The RISC-V Debug Specification

Tim Newsome, Paul Donahue (Ventana Micro Systems)

Version 1.0.0-rc3, Revised 2024-05-15: Frozen

Table of Contents

Preface

This specification is Frozen.

Change is extremely unlikely. A high threshold will be used, and a change will only occur because of some truly critical issue being identified during the public review cycle. Any other desired or needed changes can be the subject of a follow-on new extension.

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Bruce Ableidinger, Krste Asanović, Peter Ashenden, Allen Baum, Mark Beal, Alex Bradbury, Chuanhua Chang, Yen Hao Chen, Zhong-Ho Chen, Monte Dalrymple, Paul Donahue, Vyacheslav Dyachenko, Ernie Edgar, Peter Egold, Marc Gauthier, Markus Goehrle, Robert Golla, John Hauser, Richard Herveille, Yung-ching Hsiao, Po-wei Huang, Scott Johnson, L. J. Madar, Grigorios Magklis, Daniel Mangum, Alexis Marquet, Jan Matyas, Kai Meinhard, Jean-Luc Nagel, Aram Nahidipour, Rishiyur Nikhil, Gajinder Panesar, Deepak Panwar, Antony Pavlov, Klaus Kruse Pedersen, Ken Pettit, Darius Rad, Joe Rahmeh, Josh Scheid, Vedvyas Shanbhogue, Gavin Stark, Ben Staveley, Wesley Terpstra, Tommy Thorn, Megan Wachs, Jan-Willem van de Waerdt, Philipp Wagner, Stefan Wallentowitz, Ray Van De Walker, Andrew Waterman, Thomas Wicki, Andy Wright, Bryan Wyatt, and Florian Zaruba.

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 debug support on RISC-V hardware 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 hardware 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

advanced feature
: An advanced feature for advanced users. Most users will not be able to take advantage of it.

AMO
: Atomic Memory Operation.

BYPASS
: JTAG instruction that selects a single bit data register, also called BYPASS.

component
: A RISC-V core, or other part of a hardware platform. Typically all components will be connected to a single system bus.

CSR
: Control and Status Register.

DM
: Debug Module (see Chapter 3).

DMI
: Debug Module Interface (see Section 3.1).

DR
: JTAG Data Register.

DTM
: Debug Transport Module (see Chapter 6).

DXLEN
: Debug XLEN, which is the widest XLEN a hart supports, ignoring the current value of `mxl` in `misa`.

essential feature
: An essential feature must be present in order for debug to work correctly.

GPR
: General Purpose Register.

hardware platform
: A single system consisting of one or more *components*.

hart
: A hardware thread in a RISC-V core.

IDCODE
: 32-bit Identification CODE, and a JTAG instruction that returns the IDCODE value.

IR
: JTAG Instruction Register.

JTAG
: Refers to work done by IEEE's Joint Test Action Group, described in IEEE 1149.1.

legacy feature
: A legacy feature should only be implemented to support legacy hardware that is present in a system.

Minimal RISC-V Debug Specification
: A subset of the full Debug Specification that allows for very small implementations. See Chapter 3.

NAPOT
: Naturally Aligned Power-Of-Two.

NMI
: Non-Maskable Interrupt.

physical address
: address that is directly usable on the system bus.

recommended feature
: A recommended feature is not required for debug to work correctly, but it is so useful that it should not be omitted without good reason.

SBA
: System Bus Access (see Section 3.10).

specialized feature
: A specialized feature, that only makes sense in the context of some specific hardware.

TAP
: Test Access Port, defined in IEEE 1149.1.

TM

Trigger Module (see Chapter 5).

virtual address

An address as a hart sees it. If the hart is using address translation this may be different from the physical address. If there is no translation then it will be the same.

xepc

The exception program counter CSR (e.g.) that is appropriate for the mode being trapped to.

1.2. Context

This specification attempts to support all RISC-V ISA extensions that have, roughly, been ratified through the first half of 2023. In particular, though, this specification specifically addresses features in the following extensions:

1. A
2. C
3. D
4. F
5. H
6. Sm1p13
7. Ss1p13
8. Smstateen
9. V
10. Zawrs
11. Zcmp
12. Zicbom
13. Zicboz
14. Zicbop

1.2.1. Versions

Version 0.13 of this document was ratified by the RISC-V Foundation's board. Versions 0.13.*x* are bug fix releases to that ratified specification.

Version 0.14 was a working version that was never officially ratified.

Version 1.0 is almost entirely forwards and backwards compatible with Version 0.13.

1.2.1.1. Bugfixes from 0.13 to 1.0

Changes that fix a bug in the spec:

1. Fix order of operations described in sbdata0. #392
2. Resume ack is set after resume, in Section 3.5. #400

3. sselect applies to svalue . #402
4. mte only applies when action=0. #411
5. aamsize does not affect Argument Width. #420
6. Clarify that harts halt out of reset if haltreq =1. #419

1.2.1.2. Incompatible Changes from 0.13 to 1.0

Changes that are not backwards-compatible. Debuggers or hardware implementations that implement 0.13 will have to change something in order to implement 1.0:

1. Make haltsum0 optional if there is only one hart. #505
2. System bus autoincrement only happens if an access actually takes place. (sbdata0) #507
3. Bump version to 3. #512 , Require debugger to poll dmactive after lowering it. #566
4. Add pending to icount . #574
5. When a selected trigger is disabled, tdata2 and tdata3 can be written with any value supported by any of the types this trigger supports. #721
6. tcontrol fields only apply to breakpoint traps, not any trap. #723
7. If version is greater than 0, then hit0 (previously called mcontrol.`hit`) now contains 0 when a trigger fires more than one instruction after the instruction that matched. (This information is now reflected in .) #795
8. If version is greater than 0, then bit 20 of mcontrol6 is no longer used for timing information. (Previously the bit was called mcontrol.`timing`.) #807
9. If version is greater than 0, then the encodings of size for sizes greater than 64 bit have changed. #807

1.2.1.3. Minor Changes from 0.13 to 1.0

Changes that slightly modify defined behavior. Technically backwards incompatible, but unlikely to be noticeable:

1. stopcount only applies to hart-local counters. #405
2. version may be invalid when dmactive=0. #414
3. Address triggers (mcontrol) may fire on any accessed address. #421
4. All Trigger Module registers (Table 14) are optional. #431
5. When extending IR, bypass still is all ones. #437
6. ebreaks and ebreaku are WARL. #458
7. NMIs are disabled by stepie. #465
8. R/W1C fields should be cleared by writing every bit high. #472
9. Specify trigger priorities in Table 13 relative to exceptions. #478
10. Time may pass before dmactive becomes high. #500
11. Clear MPRV when resuming into lower privilege mode. #503
12. Halt state may not be preserved across reset. #504

13. Hardware should clear trigger action when dmode is cleared and action is 1. #501
14. Change quick access exceptions to halt the target in Section 3.7.1.2. #585
15. Writing 0 to tdata1 forces a state where tdata2 and tdata3 are writable. #598
16. Solutions to deal with reentrancy in Section 5.4 prevent triggers from *matching*, not merely *firing*. This primarily affects behavior. #722
17. Attempts to access an unimplemented CSR raise an illegal instruction exception. #791

1.2.1.4. New Features from 0.13 to 1.0

New backwards-compatible feature that did not exist before:

1. Add halt groups and external triggers in Section 3.6. #404
2. Reserve some DMI space for non-standard use. See custom, and custom0 through . #406
3. Reserve trigger type values for non-standard use. #417
4. Add nmi bit to itrigger. #408 and #709
5. Recommend matching on every accessed address. #449
6. Add resume groups in Section 3.6. #506
7. Add relaxedpriv . #536
8. Move scontext, renaming original to mscontext, and create hcontext. #535
9. Add mcontrol6, deprecating mcontrol. #538
10. Add hypervisor support: ebreakvs, ebreakvu, v, hcontext, mcontrol, mcontrol6, and priv. #549
11. Optionally make anyunavail and allunavail sticky, controlled by stickyunavail. #520
12. Add tmexttrigger to support trigger module external trigger inputs. #543
13. Describe mcontrol and mcontrol6 behavior with atomic instructions. #561
14. Trigger hit bits must be set on fire, may be set on match. #593
15. Add sbytemask and sbytemask to textra32 and textra64. #588
16. Allow debugger to request harts stay alive with keepalive bit in setkeepalive. #592
17. Add ndmresetpending to allow a debugger to determine when ndmreset is complete. #594
18. Add intctl to support triggers from an interrupt controller. #599

1.2.1.5. Incompatible Changes During 1.0 Stable

Backwards-incompatible changes between two versions that are both called 1.0 stable.

1. nmi was moved from etrigger to itrigger, and is now subject to the mode bits in that trigger.
2. #728 introduced Message Registers, which were later removed in #878.
3. It may not be possible to read the contents of the Program Buffer using the `progbuf` registers. #731
4. tcontrol fields apply to all traps, not just breakpoint traps. This reverts #723. #880

1.2.1.6. Incompatible Changes Between 1.0.0-rc1 and 1.0.0-rc2

Backwards-incompatible changes between 1.0.0-rc1 and 1.0.0-rc2.

1. #981 made scontext.data, mcontext.hcontext, sbytemask, and textra64.`svalue` narrower. This avoids confusion about the contents of scontext and mcontext when XLEN is reduced and increased again.

1.3. About This Document

1.3.1. Structure

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

1.3.2. ISA vs. non-ISA

This specification contains both ISA and non-ISA parts. The ISA parts define self-contained ISA extensions. The other parts of the document describe the non-ISA external debug extension. Chapters whose contents are solely one or the other are labeled as such in their title. Chapters without such a label apply to both ISA and non-ISA.

1.3.3. 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.

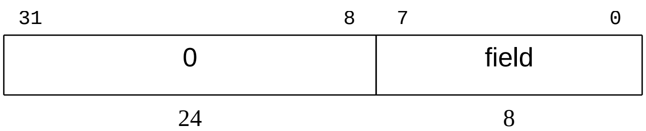
After the graphic follows a table which for each field lists its name, description, allowed accesses, and reset value. The allowed accesses are listed in Table 1. The reset value is either a constant or "Preset." The latter means it is an implementation-specific legal value.

Parts of the register which are currently unused are labeled with the number 0. Software must only write 0 to those fields, and ignore their value while reading. Hardware must return 0 when those fields are read, and ignore the value written to them.

This behavior enables us to use those fields later without having to increase the values in the version fields.

Names of registers and their fields are hyperlinks to their definition, and are also listed in the Index.

1.3.3.1. Long Name (shortname, at 0x123)



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

Table 1. Register Access Abbreviations

R	Read-only.

R/W	Read/Write.
R/W1C	Read/Write Ones to Clear. Writing 0 to every bit has no effect. Writing 1 to every bit clears the field. The result of other writes is undefined.
WARZ	Write any, read zero. A debugger may write any value. When read this field returns 0.
W1	Write-only. Only writing 1 has an effect. When read the returned value should be 0.
WARL	Write any, read legal. A debugger may write any value. If a value is unsupported, the implementation converts the value to one that is supported.

1.4. Background

There are several use cases for dedicated debugging hardware, both in native debug and external debug. Native debug (sometimes called self-hosted debug) refers to debug software running on a RISC-V platform which debugs the same platform. The optional Trigger Module provides features that are useful for native debug. External debug refers to debug software running somewhere else, debugging the RISC-V platform via a debug transport like JTAG. The entire document provides features that are useful for external debug.

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.

- Accessing hardware on a hardware platform without a working CPU. (External debug.)
- Bootstrapping a hardware platform to test, configure, and program components before there is any executable code path in the hardware platform. (External debug.)
- Debugging low-level software in the absence of an OS or other software. (External debug.)
- Debugging issues in the OS itself. (External or native debug.)
- Debugging processes running on an OS. (Native or external debug.)

1.5. Supported Features

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

1. All hart registers (including CSRs) can be read/written.
2. Memory can be accessed either from the hart's point of view, through the system bus directly, or both.
3. RV32, RV64, and future RV128 are all supported.
4. Any hart in the hardware platform can be independently debugged.

5. A debugger can discover almost [1] everything it needs to know itself, without user configuration.
6. Each hart can be debugged from the very first instruction executed.
7. A RISC-V hart can be halted when a software breakpoint instruction is executed.
8. Hardware single-step can execute one instruction at a time.
9. Debug functionality is independent of the debug transport used.
10. The debugger does not need to know anything about the microarchitecture of the harts it is debugging.
11. Arbitrary subsets of harts can be halted and resumed simultaneously. (Optional)
12. 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)
13. Registers can be accessed without halting. (Optional)
14. A running hart can be directed to execute a short sequence of instructions, with little overhead. (Optional)
15. A system bus manager allows memory access without involving any hart. (Optional)
16. A RISC-V hart can be halted when a trigger matches the PC, read/write address/data, or an instruction opcode. (Optional)
17. Harts can be grouped, and harts in the same group will all halt when any of them halts. These groups can also react to or notify external triggers. (Optional)

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

It is possible to debug code that uses software threads, but there is no special debug support for it.

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

Chapter 2. System Overview

Figure 1 shows the main components of Debug Support. Blocks shown in dotted lines are optional.

The user interacts with the Debug Host (e.g. laptop), which is running a debugger (e.g. gdb). The debugger communicates with a Debug Translator (e.g. OpenOCD, which may include a hardware driver) to communicate with Debug Transport Hardware (e.g. Olimex USB-JTAG adapter). The Debug Transport Hardware connects the Debug Host to the hardware 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 hardware 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 hardware platforms there will only be one DM that controls all the harts in the hardware platform.

DMs provide run control of their harts in the hardware 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 also 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.

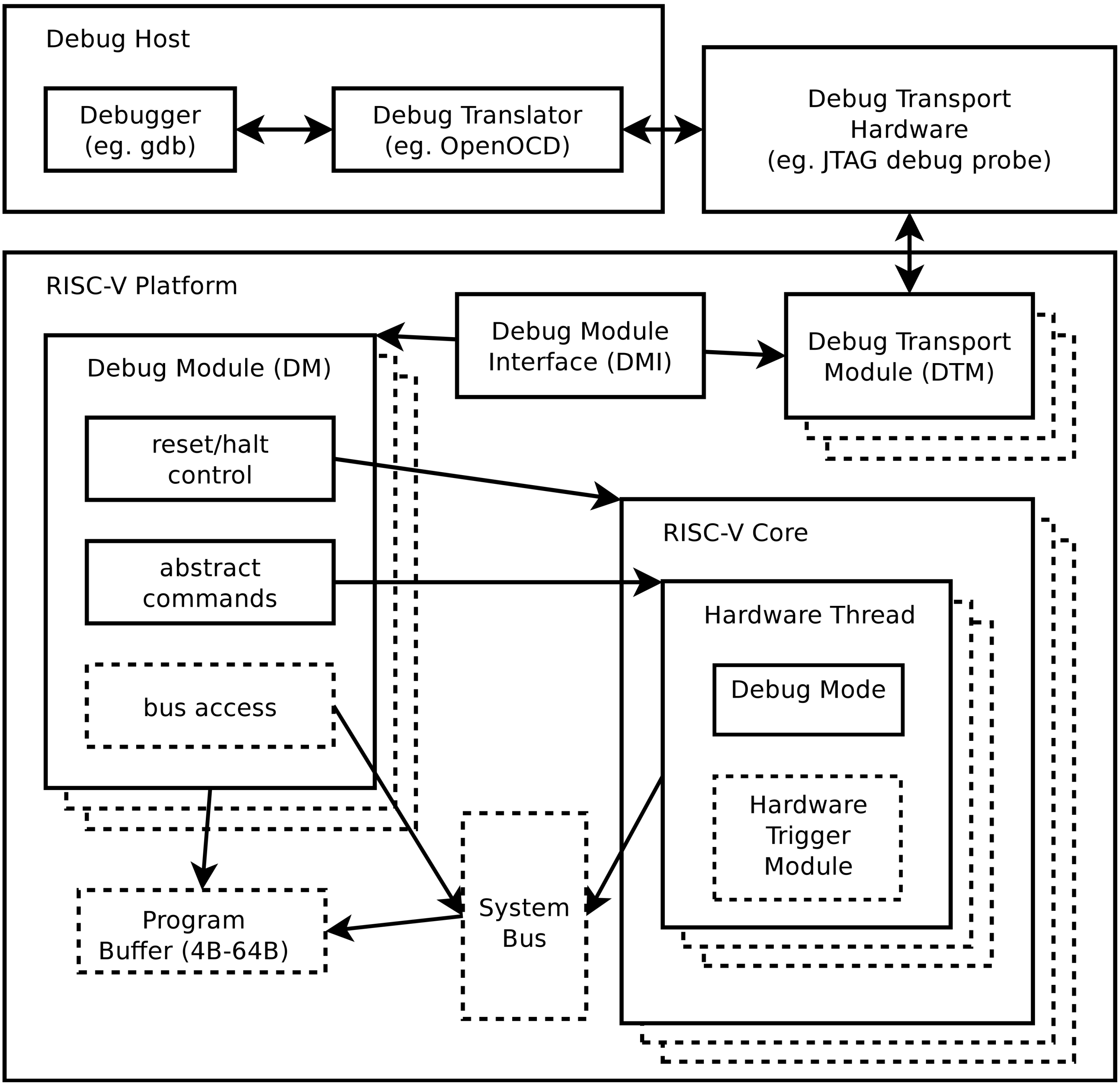


Figure 1. RISC-V Debug System Overview

Chapter 3. Debug Module (DM) (non-ISA extension)

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 abstract 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 a mechanism to allow debugging harts immediately out of reset (regardless of the reset cause). (Optional)
7. Provide abstract access to non-GPR hart registers. (Optional)
8. Provide a Program Buffer to force the hart to execute arbitrary instructions. (Optional)
9. Allow multiple harts to be halted, resumed, and/or reset at the same time. (Optional)
10. Allow memory access from a hart's point of view. (Optional)
11. Allow direct System Bus Access. (Optional)
12. Group harts. When any hart in the group halts, they all halt. (Optional)
13. Respond to external triggers by halting each hart in a configured group. (Optional)
14. Signal an external trigger when a hart in a group halts. (Optional)

In order to be compatible with this specification an implementation must:

1. Implement all the required features listed above.
2. Implement at least one of Program Buffer, System Bus Access, or Abstract Access Memory command mechanisms.
3. Do at least one of:
   a. Implement the Program Buffer.
   b. Implement abstract access to all registers that are visible to software running on the hart including all the registers that are present on the hart and listed in Table 4.
   c. Implement abstract access to at least all GPRs, dcsr, and dpc, and advertise the implementation as conforming to the "Minimal RISC-V Debug Specification", instead of the "RISC-V Debug Specification".

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

3.1. Debug Module Interface (DMI)

Debug Modules are subordinates on a bus called the Debug Module Interface (DMI). The bus manager is the Debug Transport Module(s). The Debug Module Interface can be a trivial bus with one manager and one subordinate (see Table 21), 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. Each address points at a single 32-bit register that can be read or written. 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

There are two methods that allow a debugger to reset harts. ndmreset resets all the harts in the hardware platform, as well as all other parts of the hardware platform except for the Debug Modules, Debug Transport Modules, and Debug Module Interface. Exactly what is affected by this reset is implementation dependent, but it must be possible to debug programs from the first instruction executed. hartreset resets all the currently selected harts. In this case an implementation may reset more harts than just the ones that are selected. The debugger can discover which other harts are reset (if any) by selecting them and checking anyhavereset and allhavereset.

To perform either of these resets, the debugger first asserts the bit, and then clears it. The actual reset may start as soon as the bit is asserted, but may start an arbitrarily long time after the bit is deasserted. The reset itself may also take an arbitrarily long time. While the reset is on-going, harts are either in the running state, indicating it's possible to perform some abstract commands during this time, or in the unavailable state, indicating it's not possible to perform any abstract commands during this time. Once a hart's reset is complete, `havereset` becomes set. When a hart comes out of reset and haltreq or hasresethaltreq are set, the hart will immediately enter Debug Mode (halted state). Otherwise, if the hart was initially running it will execute normally (running state) and if the hart was initially halted it should now be running but may be halted.

> *There is no general, reliable way for the debugger to know when reset has actually begun.*

The Debug Module's own state and registers should only be reset at power-up and while dmactive in dmcontrol is 0. If there is another mechanism to reset the DM, this mechanism must also reset all the harts accessible to the DM.

Due to clock and power domain crossing issues, it might not be possible to perform arbitrary DMI accesses across hardware platform reset. While ndmreset or any external reset is asserted, the only supported DM operations are reading/writing dmcontrol and reading ndmresetpending. The behavior of other accesses is undefined.

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 might or might not be cleared when dmactive is low.

3.3. Selecting Harts

Up to 2^{20} harts can be connected to a single DM. Commands issued to the DM only apply to the currently selected harts.

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 hardware platform's configuration structure.

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. The state of the hart array mask register is not affected by setting or clearing hasel.

Execution of Abstract Commands ignores this mechanism and only applies to the hart selected by hartsel.

3.4. Hart DM States

Every hart that can be selected is in exactly one of the following four DM states: non-existent, unavailable, running, or halted. Which state the selected harts are in is reflected by allnonexistent, anynonexistent, allunavail, anyunavail, allrunning, anyrunning, allhalted, and anyhalted.

Harts are nonexistent if they will never be part of this hardware platform, no matter how long a user waits. E.g. in a simple single-hart hardware platform only one hart exists, and all others are nonexistent. Debuggers may assume that a hardware platform 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. Harts may be unavailable for a variety of reasons including being reset, temporarily powered down, and not being plugged into the hardware platform. That means harts might become available or unavailable at any time, although these events should be rare in hardware platforms built to be easily debugged. There are no guarantees about the state of the hart when it becomes available.

Hardware platforms 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.

Harts are running when they are executing normally, as if no debugger was attached. This includes being in a low power mode or waiting for an interrupt, as long as a halt request will result in the hart being halted.

Harts are halted when they are in Debug Mode, only performing tasks on behalf of the debugger.

Which states a hart that is reset goes through is implementation dependent. Harts may be unavailable while reset is asserted, and some time after reset is deasserted. They might transition to running for some time after reset is deasserted. Finally they end up either running or halted, depending on haltreq and hasresethaltreq.

3.5. Run Control

For every hart, the Debug Module tracks 4 conceptual bits of state: halt request, resume ack, halt-on-reset request, and hart reset. (The hart reset and halt-on-reset request bits are optional.) These 4 bits reset to 0, except for resume ack, which may reset to either 0 or 1. The DM receives halted, running, and havereset signals from each hart. The debugger can observe the state of resume ack in allresumeack and anyresumeack, and the state of halted, running, and havereset signals in allhalted, anyhalted, allrunning, anyrunning, allhavereset, and anyhavereset. The state of the other bits cannot be observed directly.

When a debugger writes 1 to haltreq, each selected hart's halt request bit is set. When a running hart, or a hart just coming out of reset, sees its halt request bit high, it responds by halting, deasserting its running signal, and asserting its halted signal. Halted harts ignore their halt request bit.

When a debugger writes 1 to resumereq, each selected hart's resume ack bit is cleared and each selected, halted hart is sent a resume request. Harts respond by resuming, clearing their halted signal, and asserting their running signal. At the end of this process the resume ack bit is set. These status signals of all selected harts are reflected in allresumeack, anyresumeack, allrunning, and anyrunning. Resume requests are 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).

The DM can implement optional halt-on-reset bits for each hart, which it indicates by setting hasresethaltreq to 1. This means the DM implements the setresethaltreq and clrresethaltreq bits. Writing 1 to setresethaltreq sets the halt-on-reset request bit for each selected hart. When a hart's halt-on-reset request bit is set, the hart will immediately enter debug mode on the next deassertion of its reset. This is true regardless of the reset's cause. The hart's halt-on-reset request bit remains set until cleared by the debugger writing 1 to clrresethaltreq while the hart is selected, or by DM reset.

If the DM is reset while a hart is halted, it is UNSPECIFIED whether that hart resumes. Debuggers should use resumereq to explicitly resume harts before clearing dmactive and disconnecting.

3.6. Halt Groups, Resume Groups, and External Triggers

An optional feature allows a debugger to place harts into two kinds of groups: halt groups and resume groups. It is also possible to add external triggers to a halt and resume groups. At any given time, each hart and each trigger is a member of exactly one halt group and exactly one resume group.

In both halt and resume groups, group 0 is special. Harts in group 0 halt/resume as if groups aren't implemented at all.

When any hart in a halt group halts:

1. That hart halts normally, with cause reflecting the original cause of the halt.
2. All the other harts in the halt group that are running will quickly halt. cause for those harts should be set to 6, but may be set to 3. Other harts in the halt group that are halted but have started the process of resuming must also quickly become halted, even if they do resume briefly.
3. Any external triggers in that group are notified.

Adding a hart to a halt group does not automatically halt that hart, even if other harts in the group are already halted.

When an external trigger that's a member of the halt group fires:

1. All the harts in the halt group that are running will quickly halt. cause for those harts should be set to 6, but may be set to 3. Other harts in the halt group that are halted but have started the process of resuming must also quickly become halted, even if they do resume briefly.

When any hart in a resume group resumes:

1. All the other harts in that group that are halted will quickly resume as soon as any currently executing abstract commands have completed. Each hart in the group sets its resume ack bit as soon as it has resumed. Harts that are in the process of halting should complete that process and stay halted.
2. Any external triggers in that group are notified.

Adding a hart to a resume group does not automatically resume that hart, even if other harts in the group are currently running.

When an external trigger that's a member of the resume group fires:

1. All the harts in that group that are halted will quickly resume as soon as any currently executing abstract commands have completed. Each hart in the group sets its resume ack bit as soon as it has resumed. Harts that are in the process of halting should complete that process and stay halted.

External triggers are abstract concepts that can signal the DM and/or receive signals from the DM. This configuration is done through dmcs2, where external triggers are referred to by a number. Commonly, external triggers are capable of sending a signal from the hardware platform into the DM, as well as receiving a signal from the DM to take their own action on. It is also allowable for an external trigger to be input-only or output-only. By convention external triggers 0-7 are bidirectional, triggers 8-11 are input-only, and triggers 12-15 are output-only but this is not required.

External triggers could be used to implement near simultaneous halting/resuming of all cores in a hardware platform, when not all cores are RISC-V cores.

When the DM is reset, all harts must be placed in the lowest-numbered halt and resume groups that they can be in. (This will usually be group 0.)

Some designs may choose to hardcode hart groups to a group other than group 0, meaning it is never possible to halt or resume just a single hart. This is explicitly allowed. In that case it must be possible to discover the groups by using dmcs2 even if it's not possible to change the configuration.

3.7. 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 (running, halted, or held in reset) by attempting them and then looking at cmderr in abstractcs to see if they were successful. Commands may be supported with some options set, but not with other options set. If a command has unsupported options set or if bits that are defined as 0 aren't 0, then the DM must set cmderr to 2 (not supported).

> *Example: Every DM must support the Access Register command, but might not support accessing CSRs. If the debugger requests to read a CSR in that case, the command will return "not supported".*

Debuggers execute abstract commands by writing them to command. They can determine whether an abstract command is complete by reading busy in abstractcs. If the debugger starts a new command while busy is set, cmderr becomes 1 (busy), the currently executing command still gets to run to completion, but any error generated by the currently executing command is lost. After completion, cmderr indicates whether the command was successful or not. Commands may fail because a hart is not halted, not running, unavailable, or because they encounter an error during execution.

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 2. 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 DXLEN where not explicitly specified.

Table 2. 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`

> *The Abstract Command interface is designed to allow a debugger to write commands as fast as possible, and then later check whether they completed without error. In the common case the debugger will be much slower than the target and commands succeed, which allows for maximum throughput. If there is a failure, the interface ensures that no commands execute after the failing one. To discover which command failed, the debugger has to look at the state of the DM (e.g. contents of data0) or hart (e.g. contents of a register modified by a Program Buffer program) to determine which one failed.*

Before starting an abstract command, a debugger must ensure that haltreq, resumereq, and ackhavereset are all 0.

While an abstract command is executing (busy in abstractcs is high), a debugger must not change hartsel, and must not write 1 to haltreq, resumereq, ackhavereset, setresethaltreq, or clrresethaltreq.

If an abstract command does not complete in the expected time and appears to be hung, the debugger can try to reset the hart (using hartreset or ndmreset). If that doesn't clear busy, then it can try resetting the Debug Module (using dmactive).

If an abstract command is started while the selected hart is unavailable or if a hart becomes unavailable while executing an abstract command, then the Debug Module may terminate the abstract command, setting busy low, and cmderr to 4 (halt/resume). Alternatively, the command could just appear to be hung (busy never goes low).

3.7.1. 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 lists all commands.

Table 3. Meaning of cmdtype

cmdtype	Command
0	Access Register Command
1	Quick Access
2	Access Memory Command

3.7.1.1. `Access Register`

This command gives the debugger access to CPU registers and allows it to execute the Program Buffer. It performs the following sequence of operations:

1. If write is clear and transfer is set, then copy data from the register specified by regno into the **`arg0`** region of **`data`**, and perform any side effects that occur when this register is read from M-mode.
2. If write is set and transfer is set, then copy data from the **`arg0`** region of **`data`** into the register specified by regno, and perform any side effects that occur when this register is written from M-mode.
3. If aarpostincrement and transfer are set, increment regno. regno may also be incremented if aarpostincrement is set and transfer is clear.
4. 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. If the failure is that the requested register does not exist in the hart, cmderr must be set to 3 (exception).

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. It is recommended that if one register in a group is accessible, then all registers in that group are accessible, but each individual register (aside from GPRs) may be supported differently across read, write, and halt status.

Registers might not be accessible if they wouldn't be accessible by M mode code currently running. (E.g. **`fflags`** might not be accessible when **`mstatus.FS`** is 0.) If this is the case, the debugger is responsible for changing state to make the registers accessible. The Core Debug Registers (Section 4.9) should be accessible if abstract CSR access is implemented.

Table 4. Abstract Register Numbers

Numbers	Group Description
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.

The encoding of aarsize was chosen to match sbaccess in sbcs.

This command modifies **`arg0`** only when a register is read. The other **`data`** registers are not changed.

31–24	23	22–20	19	18	17	16	15–0
cmdtype	0	aarsize	aarpostincrement	postexec	transfer	write	regno
8	1	3	1	1	1	1	16

Field	Description
`cmdtype`	This is 0 to indicate Access Register Command.
`aarsize`	2 (32bit): Access the lowest 32 bits of the register. 3 (64bit): Access the lowest 64 bits of the register. 4 (128bit): Access the lowest 128 bits of the register. If aarsize specifies a size larger than the register's actual size, then the access must fail. If a register is accessible, then reads of aarsize less than or equal to the register's actual size must be supported. Writing less than the full register may be supported, but what happens to the high bits in that case is UNSPECIFIED. This field controls the Argument Width as referenced in Table 2.
`aarpostincrement`	0 (disabled): No effect. This variant must be supported. 1 (enabled): After a successful register access, regno is incremented. Incrementing past the highest supported value causes regno to become UNSPECIFIED. Supporting this variant is optional. It is undefined whether the increment happens when transfer is 0.
`postexec`	0 (disabled): No effect. This variant must be supported, and is the only supported one if progbufsize is 0. 1 (enabled): Execute the program in the Program Buffer exactly once after performing the transfer, if any. Supporting this variant is optional.
`transfer`	0 (disabled): Don't do the operation specified by write. 1 (enabled): 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 aarsize or regno.

Field	Description
`write`	When transfer is set: 0 (arg0): Copy data from the specified register into `arg0` portion of `data`. 1 (register): Copy data from `arg0` portion of `data` into the specified register.
`regno`	Number of the register to access, as described in Table 4. dpc may be used as an alias for PC if this command is supported on a non-halted hart.

3.7.1.2. `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," the Program Buffer execution ends, and the hart is halted with cause set to 3.
4. If the Program Buffer executed without an exception, then resume the hart.

Implementing this command is optional.

This command does not touch the `data` registers.

31 – 24	23 – 0
cmdtype	0
8	24

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

3.7.1.3. `Access Memory`

This command lets the debugger perform memory accesses, with the exact same memory view and permissions as the selected hart has. This includes access to hart-local memory-mapped registers, etc. The command performs the following sequence of operations:

1. Copy data from the memory location specified in `arg1` into the `arg0` portion of `data`, if write is clear.
2. Copy data from the `arg0` portion of `data` into the memory location specified in `arg1`, if write is set.
3. If aampostincrement is set, increment `arg1`.

If any of these operations fail, cmderr is set and none of the remaining steps are executed. An access may only fail if the hart, running M-mode code, might encounter that same failure when it attempts the same access. An implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure.

Debug Modules may optionally implement this command and may support read and write access to memory locations when the selected hart is running or halted. If this command supports memory accesses while the hart is running, it must also support memory accesses while the hart is halted.

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

This command modifies `arg0` only when memory is read. It modifies `arg1` only if aampostincrement is set. The other `data` registers are not changed.

31–24	23	22–20	19	18–17	16	15–14	13–0
cmdtype	aamvirtual	aamsize	aampostincrement	0	write	target-specific	0
8	1	3	1	2	1	2	14

Field	Description
`cmdtype`	This is 2 to indicate Access Memory Command.
`aamvirtual`	An implementation does not have to implement both virtual and physical accesses, but it must fail accesses that it doesn't support. 0 (physical): Addresses are physical (to the hart they are performed on). 1 (virtual): Addresses are virtual, and translated the way they would be from M-mode, with `MPRV` set. Debug Modules on systems without address translation (i.e. virtual addresses equal physical) may optionally allow aamvirtual set to 1, which would produce the same result as that same abstract command with aamvirtual cleared.
`aamsize`	0 (8bit): Access the lowest 8 bits of the memory location. 1 (16bit): Access the lowest 16 bits of the memory location. 2 (32bit): Access the lowest 32 bits of the memory location. 3 (64bit): Access the lowest 64 bits of the memory location. 4 (128bit): Access the lowest 128 bits of the memory location.
`aampostincrement`	After a memory access has completed, if this bit is 1, increment `arg1` (which contains the address used) by the number of bytes encoded in aamsize. Supporting this variant is optional, but highly recommended for performance reasons.
`write`	0 (arg0): Copy data from the memory location specified in `arg1` into the low bits of `arg0`. The value of the remaining bits of `arg0` are UNSPECIFIED. 1 (memory): Copy data from the low bits of `arg0` into the memory location specified in `arg1`.
`target-specific`	These bits are reserved for target-specific uses.

3.8. 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. DMs 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`. An implementation may support an implicit `ebreak` that is executed when a hart runs off the end of the Program Buffer. This is indicated by impebreak. With this feature, a Program Buffer of just 2 32-bit words can offer efficient debugging.

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 with an `ebreak` instruction, the hart will remain in Debug Mode and the debugger will lose control of the hart.

If progbufsize is 1 then the following apply:

1. impebreak must be 1.
2. If the debugger writes a compressed instruction into the Program Buffer, it must be placed into the lower 16 bits and accompanied by a compressed `nop` in the upper 16 bits.

This requirement on the debugger for the case of progbufsize equal to 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.

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.

3.9. Overview of Hart Debug States

Figure 2 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.

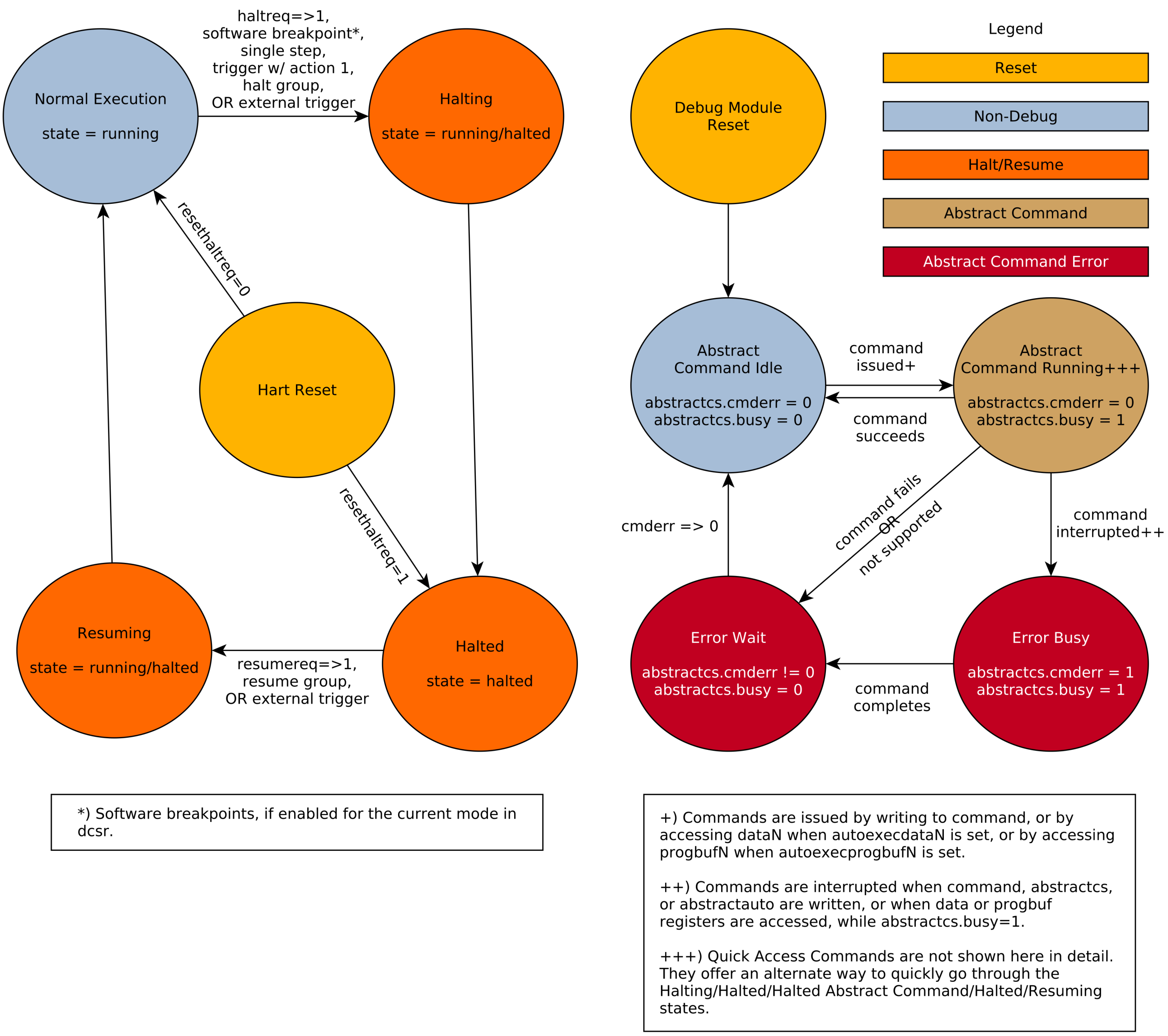


Figure 2. Run/Halt Debug State Machine for single-hart hardware platforms. As only a small amount of state is visible to the debugger, the states and transitions are conceptual.

3.10. System Bus Access

A debugger can access memory from a hart's point of view using a Program Buffer or the Abstract Access Memory command. (Both these features are optional.) 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 5 shows which bits in `sbdata` are used for each access size.

Table 5. 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 might 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. Possibilities may include writing 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.

3.11. Minimally Intrusive Debugging

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 hartinfo, this can be used to quickly perform a memory or register access. For some hardware platforms this will be too intrusive, but many hardware platforms 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.

3.12. 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 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 hardware 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.

Implementations where it's not possible to unlock the DM by using authdata should not implement that register.

3.13. Version Detection

To detect the version of the Debug Module with a minimum of side effects, use the following procedure:

1. Read dmcontrol.
2. If dmactive is 0 or ndmreset is 1:
   a. Write dmcontrol, preserving hartreset, hasel, hartsello, and hartselhi from the value that was read, setting dmactive, and clearing all the other bits.
   b. Read dmcontrol until dmactive is high.
3. Read dmstatus, which contains version.

If it was necessary to clear ndmreset, this might have the following side effects:

1. haltreq is cleared, potentially preventing a halt request made by a previous debugger from taking effect.
2. resumereq is cleared, potentially preventing a resume request made by a previous debugger from taking effect.
3. ndmreset is deasserted, releasing the hardware platform from reset if a previous debugger had set it.
4. dmactive is asserted, releasing the DM from reset. This in itself is not observable by any harts.

This procedure is guaranteed to work in future versions of this spec. The meaning of the dmcontrol bits where hartreset, hasel, hartsello, and hartselhi currently reside might change, but preserving them will have no side effects. Clearing the bits of dmcontrol not explicitly mentioned here will have no side effects beyond the ones mentioned above.

3.14. Debug Module Registers

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

Debug Module DMI Registers that are unimplemented or not mentioned in the table below return 0 when read. Writing them has no effect.

Table 6. Debug Module Debug Bus Registers

Address	Name	Section
0x04	Abstract Data 0 (data0)	Section 3.14.14
0x05	Abstract Data 1 (data1)	
0x06	Abstract Data 2 (data2)	
0x07	Abstract Data 3 (data3)	
0x08	Abstract Data 4 (data4)	
0x09	Abstract Data 5 (data5)	
0x0a	Abstract Data 6 (data6)	
0x0b	Abstract Data 7 (data7)	
0x0c	Abstract Data 8 (data8)	
0x0d	Abstract Data 9 (data9)	
0x0e	Abstract Data 10 (data10)	
0x0f	Abstract Data 11 (data11)	
0x10	Debug Module Control (dmcontrol)	Section 3.14.2
0x11	Debug Module Status (dmstatus)	Section 3.14.1
0x12	Hart Info (hartinfo)	Section 3.14.3
0x13	Halt Summary 1 (haltsum1)	Section 3.14.19
0x14	Hart Array Window Select (hawindowsel)	Section 3.14.4
0x15	Hart Array Window (hawindow)	Section 3.14.5
0x16	Abstract Control and Status (abstractcs)	Section 3.14.6
0x17	Abstract Command (command)	Section 3.14.7
0x18	Abstract Command Autoexec (abstractauto)	Section 3.14.8
0x19	Configuration Structure Pointer 0 (confstrptr0)	Section 3.14.9
0x1a	Configuration Structure Pointer 1 (confstrptr1)	Section 3.14.10
0x1b	Configuration Structure Pointer 2 (confstrptr2)	Section 3.14.11
0x1c	Configuration Structure Pointer 3 (confstrptr3)	Section 3.14.12
0x1d	Next Debug Module (nextdm)	Section 3.14.13
0x1f	Custom Features (custom)	Section 3.14.31
0x20	Program Buffer 0 (progbuf0)	Section 3.14.15
0x21	Program Buffer 1 (progbuf1)	
0x22	Program Buffer 2 (progbuf2)	
0x23	Program Buffer 3 (progbuf3)	
0x24	Program Buffer 4 (progbuf4)	
0x25	Program Buffer 5 (progbuf5)	
0x26	Program Buffer 6 (progbuf6)	
0x27	Program Buffer 7 (progbuf7)	

Address	Name	Section
0x28	Program Buffer 8 (progbuf8)	
0x29	Program Buffer 9 (progbuf9)	
0x2a	Program Buffer 10 (progbuf10)	
0x2b	Program Buffer 11 (progbuf11)	
0x2c	Program Buffer 12 (progbuf12)	
0x2d	Program Buffer 13 (progbuf13)	
0x2e	Program Buffer 14 (progbuf14)	
0x2f	Program Buffer 15 (progbuf15)	
0x30	Authentication Data (authdata)	Section 3.14.16
0x32	Debug Module Control and Status 2 (dmcs2)	Section 3.14.17
0x34	Halt Summary 2 (haltsum2)	Section 3.14.20
0x35	Halt Summary 3 (haltsum3)	Section 3.14.21
0x37	System Bus Address 127:96 (sbaddress3)	Section 3.14.26
0x38	System Bus Access Control and Status (sbcs)	Section 3.14.22
0x39	System Bus Address 31:0 (sbaddress0)	Section 3.14.23
0x3a	System Bus Address 63:32 (sbaddress1)	Section 3.14.24
0x3b	System Bus Address 95:64 (sbaddress2)	Section 3.14.25
0x3c	System Bus Data 31:0 (sbdata0)	Section 3.14.27
0x3d	System Bus Data 63:32 (sbdata1)	Section 3.14.28
0x3e	System Bus Data 95:64 (sbdata2)	Section 3.14.29
0x3f	System Bus Data 127:96 (sbdata3)	Section 3.14.30
0x40	Halt Summary 0 (haltsum0)	Section 3.14.18
0x70	Custom Features 0 (custom0)	Section 3.14.32
0x71	Custom Features 1 (custom1)	
0x72	Custom Features 2 (custom2)	
0x73	Custom Features 3 (custom3)	
0x74	Custom Features 4 (custom4)	
0x75	Custom Features 5 (custom5)	
0x76	Custom Features 6 (custom6)	
0x77	Custom Features 7 (custom7)	
0x78	Custom Features 8 (custom8)	
0x79	Custom Features 9 (custom9)	
0x7a	Custom Features 10 (custom10)	
0x7b	Custom Features 11 (custom11)	
0x7c	Custom Features 12 (custom12)	

Address	Name	Section
0x7d	Custom Features 13 (custom13)	
0x7e	Custom Features 14 (custom14)	
0x7f	Custom Features 15 (custom15)	

3.14.1. Debug Module Status (dmstatus, at 0x11)

This register reports status for the overall Debug Module as well as the currently selected harts, as defined in hasel. Its address will not change in the future, because it contains version.

This entire register is read-only.

31 25	24	23	22	21 20	19	18
0	ndmresetpending	stickyunavail	impebreak	0	allhavereset	anyhavereset
7	1	1	1	2	1	1

17	16	15	14	13	12	11
allresumeack	anyresumeack	allnonexistent	anynonexistent	allunavail	anyunavail	allrunning
1	1	1	1	1	1	1

10	9	8	7	6	5	4	3 0
anyrunning	allhalted	anyhalted	authenticated	authbusy	hasresethaltreq	confstrptrvalid	version
1	1	1	1	1	1	1	4

Field	Description	Access	Reset
`ndmresetpending`	0 (false): Unimplemented, or ndmreset is zero and no ndmreset is currently in progress. 1 (true): ndmreset is currently nonzero, or there is an ndmreset in progress.	R	-
`stickyunavail`	0 (current): The per-hart **`unavail`** bits reflect the current state of the hart. 1 (sticky): The per-hart **`unavail`** bits are sticky. Once they are set, they will not clear until the debugger acknowledges them using ackunavail.	R	Preset
`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 and reset has not been acknowledged for any of them.	R	-
`anyhavereset`	This field is 1 when at least one currently selected hart has been reset and reset has not been acknowledged for that hart.	R	-

Field	Description	Access	Reset
`allresumeack`	This field is 1 when all currently selected harts have their resume ack bit set.	R	-
`anyresumeack`	This field is 1 when any currently selected hart has its resume ack bit set.	R	-
`allnonexistent`	This field is 1 when all currently selected harts do not exist in this hardware platform.	R	-
`anynonexistent`	This field is 1 when any currently selected hart does not exist in this hardware platform.	R	-
`allunavail`	This field is 1 when all currently selected harts are unavailable, or (if stickyunavail is 1) were unavailable without that being acknowledged.	R	-
`anyunavail`	This field is 1 when any currently selected hart is unavailable, or (if stickyunavail is 1) was unavailable without that being acknowledged.	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	-
`authenticated`	0 (false): Authentication is required before using the DM. 1 (true): The authentication check has passed. On components that don't implement authentication, this bit must be preset as 1.	R	Preset
`authbusy`	0 (ready): The authentication module is ready to process the next read/write to authdata. 1 (busy): 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
`hasresethaltreq`	1 if this Debug Module supports halt-on-reset functionality controllable by the setresethaltreq and clrresethaltreq bits. 0 otherwise.	R	Preset
`confstrptrvalid`	0 (invalid): confstrptr0--confstrptr3 hold information which is not relevant to the configuration structure. 1 (valid): confstrptr0--confstrptr3 hold the address of the configuration structure.	R	Preset

Field	Description	Access	Reset
`version`	0 (none): There is no Debug Module present. 1 (0.11): There is a Debug Module and it conforms to version 0.11 of this specification. 2 (0.13): There is a Debug Module and it conforms to version 0.13 of this specification. 3 (1.0): There is a Debug Module and it conforms to version 1.0 of this specification. 15 (custom): There is a Debug Module but it does not conform to any available version of this spec.	R	3

3.14.2. 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. Debuggers must not change hartsel while an abstract command is executing.

There are separate setresethaltreq and clrresethaltreq bits so that it is possible to write dmcontrol without changing the halt-on-reset request bit for each selected hart, when not all selected harts have the same configuration.

On any given write, a debugger may only write 1 to at most one of the following bits: resumereq, hartreset, ackhavereset, setresethaltreq, and clrresethaltreq. The others must be written 0.

resethaltreq is an optional internal bit of per-hart state that cannot be read, but can be written with setresethaltreq and clrresethaltreq.

keepalive is an optional internal bit of per-hart state. When it is set, it suggests that the hardware should attempt to keep the hart available for the debugger, e.g. by keeping it from entering a low-power state once powered on. Even if the bit is implemented, hardware might not be able to keep a hart available. The bit is written through setkeepalive and clrkeepalive.

For forward compatibility, version will always be readable when bit 1 (ndmreset) is 0 and bit 0 (dmactive) is 1.

31	30	29	28	27	26	25–16
haltreq	resumereq	hartreset	ackhavereset	ackunavail	hasel	hartsello
1	1	1	1	1	1	10

15–6	5	4	3	2	1	0
hartselhi	setkeepalive	clrkeepalive	setresethaltreq	clrresethaltreq	ndmreset	dmactive
10	1	1	1	1	1	1

Field	Description	Access	Reset
`haltreq`	Writing 0 clears the halt request bit for all currently selected harts. This may cancel outstanding halt requests for those harts. Writing 1 sets the halt request bit for all currently selected harts. Running harts will halt whenever their halt request bit is set. Writes apply to the new value of hartsel and hasel.	WARZ	-
`resumereq`	Writing 1 causes the currently selected harts to resume once, if they are halted when the write occurs. It also clears the resume ack bit for those harts. resumereq is ignored if haltreq is set. Writes apply to the new value of hartsel and hasel.	W1	-
`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. While this bit is 1, the debugger must not change which harts are selected. 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.	WARL	0
`ackhavereset`	0 (nop): No effect. 1 (ack): Clears `havereset` for any selected harts. Writes apply to the new value of hartsel and hasel.	W1	-
`ackunavail`	0 (nop): No effect. 1 (ack): Clears `unavail` for any selected harts that are currently available. Writes apply to the new value of hartsel and hasel.	W1	-

Field	Description	Access	Reset
`hasel`	Selects the definition of currently selected harts. 0 (single): There is a single currently selected hart, that is selected by hartsel. 1 (multiple): There may be multiple currently selected harts — the hart 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.	WARL	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.	WARL	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.	WARL	0
`setkeepalive`	This optional field sets keepalive for all currently selected harts, unless clrkeepalive is simultaneously set to 1. Writes apply to the new value of hartsel and hasel.	W1	-
`clrkeepalive`	This optional field clears keepalive for all currently selected harts. Writes apply to the new value of hartsel and hasel.	W1	-
`setresethaltreq`	This optional field writes the halt-on-reset request bit for all currently selected harts, unless clrresethaltreq is simultaneously set to 1. When set to 1, each selected hart will halt upon the next deassertion of its reset. The halt-on-reset request bit is not automatically cleared. The debugger must write to clrresethaltreq to clear it. Writes apply to the new value of hartsel and hasel. If hasresethaltreq is 0, this field is not implemented.	W1	-
`clrresethaltreq`	This optional field clears the halt-on-reset request bit for all currently selected harts. Writes apply to the new value of hartsel and hasel.	W1	-
`ndmreset`	This bit controls the reset signal from the DM to the rest of the hardware platform. The signal should reset every part of the hardware platform, including every hart, except for the DM and any logic required to access the DM. To perform a hardware platform reset the debugger writes 1, and then writes 0 to deassert the reset.	R/W	0

Field	Description	Access	Reset
`dmactive`	This bit serves as a reset signal for the Debug Module itself. After changing the value of this bit, the debugger must poll dmcontrol until dmactive has taken the requested value before performing any action that assumes the requested dmactive state change has completed. Hardware may take an arbitrarily long time to complete activation or deactivation and will indicate completion by setting dmactive to the requested value. 0 (inactive): 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). Any accesses to the module may fail. Specifically, version might not return correct data. 1 (active): The module functions normally. No other mechanism should exist that may result in resetting the Debug Module after power up. To place the Debug Module into a known state, a debugger may write 0 to dmactive, poll until dmactive is observed 0, write 1 to dmactive, and poll until dmactive is observed 1. Implementations may pay attention to this bit to further aid debugging, for example by preventing the Debug Module from being power gated while debugging is active.	R/W	0

3.14.3. 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.

31–24	23–20	19–17	16	15–12	11–0
0	nscratch	0	dataaccess	datasize	dataaddr
8	4	3	1	4	12

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

Field	Description	Access	Reset
`dataaccess`	0 (csr): The **`data`** registers are shadowed in the hart by CSRs. Each CSR is DXLEN bits in size, and corresponds to a single argument, per Table 2. 1 (memory): 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 CSRs 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: Address of RAM where the data registers are shadowed. This address is sign extended giving a range of -2048 to 2047, easily addressed with a load or store using **`x0`** as the address register.	R	Preset

3.14.4. 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.

31 … 15	14 … 0
0	hawindowsel
17	15

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. E.g. on a hardware platform with 48 harts only bit 0 of this field may actually be writable.	WARL	0

3.14.5. 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.

31 – 0
maskdata
32

3.14.6. Abstract Control and Status (abstractcs, at 0x16)

Writing this register while an abstract command is executing causes cmderr to become 1 (busy) once the command completes (busy becomes 0).

> *datacount must be at least 1 to support RV32 harts, 2 to support RV64 harts, or 4 to support RV128 harts.*

31 – 29	28 – 24	23 – 13	12	11	10 – 8	7 – 4	3 – 0
0	progbufsize	0	busy	relaxedpriv	cmderr	0	datacount
3	5	11	1	1	3	4	4

Field	Description	Access	Reset
`progbufsize`	Size of the Program Buffer, in 32-bit words. Valid sizes are 0 - 16.	R	Preset
`busy`	0 (ready): There is no abstract command currently being executed. 1 (busy): 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
`relaxedpriv`	This optional bit controls whether program buffer and abstract memory accesses are performed with the exact and full set of permission checks that apply based on the current architectural state of the hart performing the access, or with a relaxed set of permission checks (e.g. PMP restrictions are ignored). The details of the latter are implementation-specific. 0 (full checks): Full permission checks apply. 1 (relaxed checks): Relaxed permission checks apply.	WARL	Preset

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. This field only contains a valid value if busy is 0. 0 (none): No error. 1 (busy): An abstract command was executing while command, abstractcs, or abstractauto was written, or when one of the `data` or `progbuf` registers was read or written. This status is only written if cmderr contains 0. 2 (not supported): The command in command is not supported. It may be supported with different options set, but it will not be supported at a later time when the hart or system state are different. 3 (exception): An exception occurred while executing the command (e.g. 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), or unavailable. 5 (bus): The abstract command failed due to a bus error (e.g. alignment, access size, or timeout). 6 (reserved): Reserved for future use. 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 1 — 12.	R	Preset

3.14.7. Abstract Command (command, at 0x17)

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

Writing this register while an abstract command is executing causes cmderr to become 1 (busy) once the command completes (busy becomes 0).

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.

31 – 24	23 – 0
cmdtype	control
8	24

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

3.14.8. Abstract Command Autoexec (abstractauto, at 0x18)

This register is optional. Including it allows more efficient burst accesses. A debugger can detect whether it is supported by setting bits and reading them back.

If this register is implemented then bits corresponding to implemented progbuf and data registers must be writable. Other bits must be hard-wired to 0.

If this register is written while an abstract command is executing then the write is ignored and cmderr becomes 1 (busy) once the command completes (busy becomes 0).

31 – 16	15 – 12	11 – 0
autoexecprogbuf	0	autoexecdata
16	4	12

Field	Description	Access	Reset
`autoexecprogbuf`	When a bit in this field is 1, read or write accesses to the corresponding `progbuf` word cause the DM to act as if the current value in command was written there again after the access to `progbuf` completes.	WARL	0
`autoexecdata`	When a bit in this field is 1, read or write accesses to the corresponding `data` word cause the DM to act as if the current value in command was written there again after the access to `data` completes.	WARL	0

3.14.9. Configuration Structure Pointer 0 (confstrptr0, at 0x19)

When confstrptrvalid is set, reading this register returns bits 31:0 of the configuration structure pointer. Reading the other `confstrptr` registers returns the upper bits of the address.

When system bus access 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 configuration structure from the hart with ID 0.

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

The configuration structure itself is a data structure of the same format as the data structure pointed to by `mconfigptr` as described in the Privileged Spec.

This entire register is read-only.

31 … 0
addr
32

3.14.10. Configuration Structure Pointer 1 (confstrptr1, at 0x1a)

When confstrptrvalid is set, reading this register returns bits 63:32 of the configuration structure pointer. See confstrptr0 for more details.

This entire register is read-only.

31 … 0
addr
32

3.14.11. Configuration Structure Pointer 2 (confstrptr2, at 0x1b)

When confstrptrvalid is set, reading this register returns bits 95:64 of the configuration structure pointer. See confstrptr0 for more details.

This entire register is read-only.

31 … 0
addr
32

3.14.12. Configuration Structure Pointer 3 (confstrptr3, at 0x1c)

When confstrptrvalid is set, reading this register returns bits 127:96 of the configuration structure pointer. See confstrptr0 for more details.

This entire register is read-only.

31 … 0
addr
32

3.14.13. 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.

31 … 0
addr
32

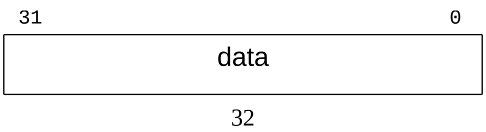
3.14.14. Abstract Data 0 (data0, at 0x04)

data0 through data11 are registers that may be read or changed by abstract commands. datacount indicates how many of them are implemented, starting at data0, counting up. Table 2 shows how abstract commands use these registers.

Accessing these registers while an abstract command is executing causes cmderr to be set to 1 (busy) if it is 0.

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

The values in these registers might 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.



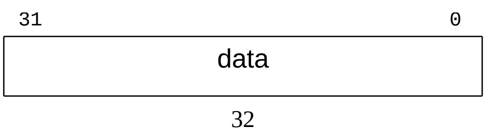
3.14.15. Program Buffer 0 (progbuf0, at 0x20)

progbuf0 through progbuf15 must provide write access to the optional program buffer. It may also be possible for the debugger to read from the program buffer through these registers. If reading is not supported, then all reads return 0.

progbufsize indicates how many `progbuf` registers are implemented starting at progbuf0, counting up.

Accessing these registers while an abstract command is executing causes cmderr to be set to 1 (busy) if it is 0.

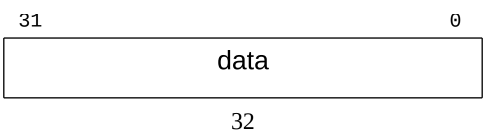
Attempts to write them while busy is set does not change their value.



3.14.16. Authentication Data (authdata, at 0x30)

This register serves as a 32-bit serial port to/from 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.



3.14.17. Debug Module Control and Status 2 (dmcs2, at 0x32)

This register contains DM control and status bits that didn't easily fit in dmcontrol and dmstatus. All are optional.

If halt groups are not implemented, then group will always be 0 when grouptype is 0.

If resume groups are not implemented, then grouptype will remain 0 even after 1 is written there.

The DM external triggers available to add to halt groups may be the same as or distinct from the DM external triggers available to add to resume groups.

31 12	11	10 7	6 2	1	0
0	grouptype	dmexttrigger	group	hgwrite	hgselect
20	1	4	5	1	1

Field	Description	Access	Reset
`grouptype`	0 (halt): The remaining fields in this register configure halt groups. 1 (resume): The remaining fields in this register configure resume groups.	WARL	0
`dmexttrigger`	This field contains the currently selected DM external trigger. If a non-existent trigger value is written here, the hardware will change it to a valid one or 0 if no DM external triggers exist.	WARL	0
`group`	When hgselect is 0, contains the group of the hart specified by hartsel. When hgselect is 1, contains the group of the DM external trigger selected by dmexttrigger. The value written to this field is ignored unless hgwrite is also written 1. Group numbers are contiguous starting at 0, with the highest number being implementation-dependent, and possibly different between different group types. Debuggers should read back this field after writing to confirm they are using a hart group that is supported. If groups aren't implemented, then this entire field is 0.	WARL	preset

Field	Description	Access	Reset
`hgwrite`	When 1 is written and hgselect is 0, for every selected hart the DM will change its group to the value written to group, if the hardware supports that group for that hart. Implementations may also change the group of a minimal set of unselected harts in the same way, if that is necessary due to a hardware limitation. When 1 is written and hgselect is 1, the DM will change the group of the DM external trigger selected by dmexttrigger to the value written to group, if the hardware supports that group for that trigger. Writing 0 has no effect.	W1	-
`hgselect`	0 (harts): Operate on harts. 1 (triggers): Operate on DM external triggers. If there are no DM external triggers, this field must be tied to 0.	WARL	0

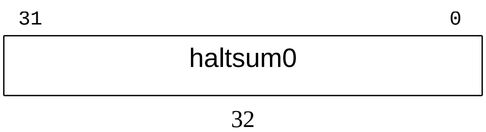
3.14.18. Halt Summary 0 (haltsum0, at 0x40)

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.

This register might not be present if fewer than 2 harts are connected to this DM.

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.



3.14.19. 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 might not be present if fewer than 33 harts are connected to this DM.

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.

31 - 0
haltsum1
32

3.14.20. 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 might not be present if fewer than 1025 harts are connected to this DM.

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.

31 - 0
haltsum2
32

3.14.21. 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 might not be present if fewer than 32769 harts are connected to this DM.

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.

31 - 0
haltsum3
32

3.14.22. System Bus Access Control and Status (sbcs, at 0x38)

31 - 29	28 - 23	22	21	20	19 - 17	16
sbversion	0	sbbusyerror	sbbusy	sbreadonaddr	sbaccess	sbautoincrement
3	6	1	1	1	3	1

15	14 - 12	11 - 5	4	3	2	1	0
sbreadondata	sberror	sbasize	sbaccess128	sbaccess64	sbaccess32	sbaccess16	sbaccess8
1	3	7	1	1	1	1	1

Field	Description	Access	Reset
`sbversion`	0 (legacy): The System Bus interface conforms to mainline drafts of this spec older than 1 January, 2018. 1 (1.0): 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 set, no more system bus accesses can be initiated by the Debug Module.	R/W1C	0
`sbbusy`	When 1, indicates the system bus manager 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. Writes to sbcs while sbbusy is high result in undefined behavior. A debugger must not write to sbcs until it reads 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 (8bit): 8-bit 1 (16bit): 16-bit 2 (32bit): 32-bit 3 (64bit): 64-bit 4 (128bit): 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 4.	R/W	2
`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

Field	Description	Access	Reset
`sberror`	When the Debug Module's system bus manager encounters an 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 (none): There was no bus error. 1 (timeout): There was a timeout. 2 (address): A bad address was accessed. 3 (alignment): There was an alignment error. 4 (size): An access of unsupported size was requested. 7 (other): 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

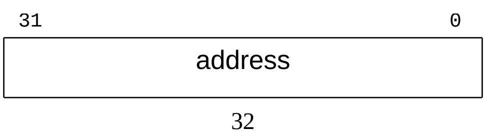
3.14.23. System Bus Address 31:0 (sbaddress0, at 0x39)

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

When the system bus manager 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.

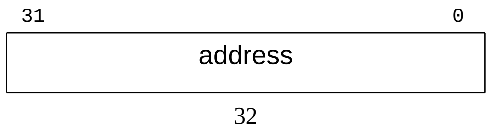


Field	Description	Access	Reset
`address`	Accesses bits 31:0 of the physical address in `sbaddress`.	R/W	0

3.14.24. System Bus Address 63:32 (sbaddress1, at 0x3a)

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

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

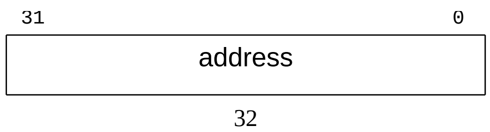


Field	Description	Access	Reset
`address`	Accesses bits 63:32 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

3.14.25. System Bus Address 95:64 (sbaddress2, at 0x3b)

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

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

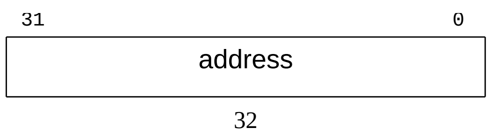


Field	Description	Access	Reset
`address`	Accesses bits 95:64 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

3.14.26. System Bus Address 127:96 (sbaddress3, at 0x37)

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

When the system bus manager 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

3.14.27. 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 either sberror or sbbusyerror isn't 0 then accesses do nothing.

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

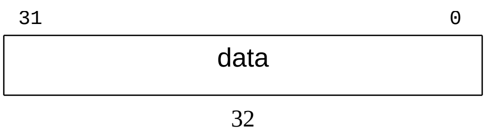
Writes to this register start the following:

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 sbreadondata is set:
   a. Perform a system bus read from the address contained in `sbaddress`, placing the result in `sbdata`.
   b. If sbautoincrement is set and the read was successful, increment `sbaddress`.
4. 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.

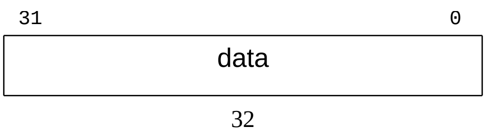


Field	Description	Access	Reset
`data`	Accesses bits 31:0 of `sbdata`.	R/W	0

3.14.28. System Bus Data 63:32 (sbdata1, at 0x3d)

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

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

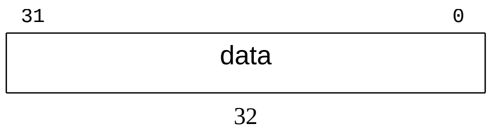


Field	Description	Access	Reset
`data`	Accesses bits 63:32 of `sbdata` (if the system bus is that wide).	R/W	0

3.14.29. System Bus Data 95:64 (sbdata2, at 0x3e)

This register only exists if sbaccess128 is 1.

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

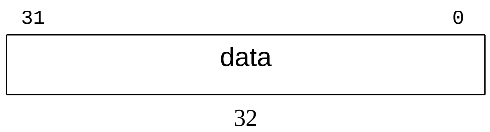


Field	Description	Access	Reset
`data`	Accesses bits 95:64 of `sbdata` (if the system bus is that wide).	R/W	0

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

This register only exists if sbaccess128 is 1.

If the bus manager 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

3.14.31. Custom Features (custom, at 0x1f)

This optional register may be used for non-standard features. Future version of the debug spec will not use this address.

3.14.32. Custom Features 0 (custom0, at 0x70)

The optional custom0 through custom15 registers may be used for non-standard features. Future versions of the debug spec will not use these addresses.

Chapter 4. Sdext (ISA Extension)

This chapter describes the Sdext ISA extension. It must be implemented to make external debug work, and is only useful in conjunction with external 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.

In order to be compatible with this specification an implementation must implement everything described in this chapter that is not explicitly listed as optional.

If Sdext is implemented and Sdtrig is not implemented, then accessing any of the Sdtrig CSRs must raise an illegal instruction exception.

4.1. Debug Mode

Debug Mode is a special processor mode used only when a hart is halted for external debugging. Because the hart is halted, there is no forward progress in the normal instruction stream. How Debug Mode is implemented is not specified here.

When executing code due to an abstract command, the hart stays in Debug Mode and the following apply:

1. All implemented instructions operate just as they do in M-mode, unless an exception is mentioned in this list.
2. All operations are executed with machine mode privilege, except that additional Debug Mode CSRs are accessible and `mprv` in `mstatus` may be ignored according to mprven. Full permission checks, or a relaxed set of permission checks, will apply according to relaxedpriv.
3. All interrupts (including NMI) are masked.
4. Traps don't take place. Instead, they end execution of the program buffer and the hart remains in Debug Mode. Because they do not trap to M-mode, they do not update registers such as , `mepc`, `mcause`, `mtval`, `mtval2`, and `mtinst`. The same is true for the equivalent privileged registers that are updated when trapping to other modes. Registers that may be updated as part of execution before the exception are allowed to be updated. For example, vector load/store instructions which raise exceptions may partially update the destination register and set `vstart` appropriately.
5. Triggers don't match or fire.
6. If stopcount is 0 then counters continue. If it is 1 then counters are stopped.
7. If stoptime is 0 then `time` continues to update. If it is 1 then `time` will not update. It will resynchronize with `time` after leaving Debug Mode.
8. Instructions that place the hart into a stalled state act as a `nop`. This includes `wfi`, `wrs.sto`, and `wrs.nto`.
9. Almost all instructions that change the privilege mode have UNSPECIFIED behavior. This includes `ecall`, `mret`, `sret`, and `uret`. (To change the privilege mode, the debugger can write prv and v in dcsr). The only exception is `ebreak`, which ends execution of the Program Buffer when executed.
10. All control transfer instructions may act as illegal instructions if their destination is in the Program Buffer. If one such instruction acts as an illegal instruction, all such instructions must act

as illegal instructions.

11. All control transfer instructions may act as illegal instructions if their destination is outside the Program Buffer. If one such instruction acts as an illegal instruction, all such instructions must act as illegal instructions.
12. Instructions that depend on the value of the PC (e.g. `auipc`) may act as illegal instructions.
13. Effective XLEN is DXLEN.
14. Forward progress is guaranteed.

> *When mprven, the external debugger can set MPRV and MPP appropriately to have hardware perform memory accesses with the appropriate endianness, address translation, permission checks, and PMP/PMA checks (subject to relaxedpriv). This is also the only way to access all of physical memory when 34-bit physical addresses are supported on a Sv32 hart. If hardware ties mprven to 0 then the external debugger is expected to simulate all the effects of MPRV, including any extensions that affect memory accesses. For these reasons it is recommended to tie mprven to 1.*

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. A higher level debugger may choose to automate this.*

4.3. Wait for Interrupt Instruction

If halt is requested while `wfi` is executing, then the hart must leave the stalled state, completing this instruction's execution, and then enter Debug Mode.

4.4. Wait-on-Reservation-Set Instructions

If halt is requested while `wrs.sto` or `wrs.nto` is executing, then the hart must leave the stalled state, completing this instruction's execution, and then enter Debug Mode.

4.5. Single Step

4.5.1. Step Bit In Dcsr

This method is only available to external debuggers, and is the preferred way to single step.

An external debugger can cause a halted hart to execute a single instruction or trap and then re-enter Debug Mode by setting step before resuming. If step is set when a hart resumes then it will single step, regardless of the reason for resuming.

If control is transferred to a trap handler while executing the instruction, then Debug Mode is re-entered immediately after the PC is changed to the trap handler, and the appropriate `tval` and `cause` registers are updated. In this case none of the trap handler is executed, and if the cause was a pending interrupt no instructions might be executed at all.

If executing or fetching the instruction causes a trigger to fire with action=1, 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 occur 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.

If the instruction being stepped over would normally stall the hart, then instead the instruction is treated as a `nop`. This includes `wfi`, `wrs.sto`, and `wrs.nto`.

4.5.2. Icount Trigger

Native debuggers won't have access to dcsr, but can use the icount trigger by setting count to 1.

This approach does have some limitations:

1. Interrupts will fire as usual. Debuggers that want to disable interrupts while stepping must disable them by changing `mstatus`, and specially handle instructions that read `mstatus`.
2. `wfi` instructions are not treated specially and might take a very long time to complete.

This mechanism cleanly supports a system which supports multiple privilege levels, where the OS or a debug stub runs in M-Mode while the program being debugged runs in a less privileged mode. Systems that only support M-Mode can use icount as well, but count must be able to count several instructions (depending on the software implementation). See Section B.3.1.

4.6. Reset

If the halt signal (driven by the hart's halt request bit in the Debug Module) or hasresethaltreq are 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.

4.7. Halt

When a hart halts:

1. cause is updated.
2. prv and v are set to reflect current privilege mode.
3. dpc is set to the next instruction that should be executed.
4. If the current instruction can be partially executed and should be restarted to complete, then the relevant state for that is updated. E.g. if a halt occurs during a partially executed vector instruction, then `vstart` is updated, and dpc is updated to the address of the partially executed instruction. This is analogous to how vector instructions behave for exceptions.

5. The hart enters Debug Mode.

4.8. Resume

When a hart resumes:

1. `pc` changes to the value stored in dpc.
2. The current privilege mode and virtualization mode are changed to that specified by prv and v.
3. If the new privilege mode is less privileged than M-mode, `MPRV` in `mstatus` is cleared.
4. The hart is no longer in debug mode.

4.9. Core Debug Registers

The supported Core Debug Registers must be implemented for each hart that can be debugged. They are CSRs, accessible using the RISC-V `csr` opcodes and optionally also using abstract debug commands.

Attempts to access an unimplemented Core Debug Register raise an illegal instruction exception.

These registers are only accessible from Debug Mode.

Table 7. Core Debug Registers

Address	Name	Section
0x7b0	Debug Control and Status (dcsr)	Section 4.9.1
0x7b1	Debug PC (dpc)	Section 4.9.2
0x7b2	Debug Scratch Register 0 (dscratch0)	Section 4.9.3
0x7b3	Debug Scratch Register 1 (dscratch1)	Section 4.9.4

4.9.1. Debug Control and Status (dcsr, at 0x7b0)

Upon entry into Debug Mode, v and prv are updated with the privilege level the hart was previously in, and cause is updated with the reason for Debug Mode entry. Other than these fields and nmip, the other fields of dcsr are only writable by the external debugger.

Table 8 shows the priorities of reasons for entering Debug Mode. Implementations should implement priorities as shown in the table. For compatibility with old versions of this spec, resethaltreq and haltreq are allowed to be at different positions than shown as long as:

1. resethaltreq is higher priority than haltreq
2. the relative order of the other four causes is maintained

Table 8. Priority of reasons for entering Debug Mode from highest to lowest.

cause encoding	Cause
5	resethaltreq
6	halt group
3	haltreq

cause encoding	Cause
2	trigger (See Table 13 for detailed priority)
1	ebreak
4	step

Note that mcontrol/mcontrol6 triggers which fire after the instruction which hit the trigger are considered to be high priority causes on the subsequent instruction. Therefore, an execute trigger with timing=after on an ebreak instruction is lower priority than the ebreak itself because the trigger will fire after the ebreak instruction. For the same reason, if a single instruction is stepped with both icount and step then the step has priority. See Table 13 for the relative priorities of triggers with respect to the ebreak instruction.

Most multi-hart implementations will probably hardwire stoptime to 0, as the implementation can get complicated and the benefit is small.

This CSR is read/write.

31–28	27	26–24	23–20	19	18	17	16	15	14
debugver	0	extcause	0	cetrig	0	ebreakvs	ebreakvu	ebreakm	0
4	1	3	4	1	1	1	1	1	1

13	12	11	10	9	8–6	5	4	3	2	1–0
ebreaks	ebreaku	stepie	stopcount	stoptime	cause	v	mprven	nmip	step	prv
1	1	1	1	1	3	1	1	1	1	2

Field	Description	Access	Reset
`debugver`	0 (none): There is no debug support. 4 (1.0): Debug support exists as it is described in this document. 15 (custom): There is debug support, but it does not conform to any available version of this spec.	R	Preset
`extcause`	When cause is 7, this optional field contains the value of a more specific halt reason than "other." Otherwise it contains 0. 0 (critical error): The hart entered a critical error state, as defined in the Smdbltrp extension. All other values are reserved for future versions of this spec, or for use by other RISC-V extensions.	R	0

Field	Description	Access	Reset
`cetrig`	This bit is part of Smdbltrp and only exists when that extension is implemented. 0 (disabled): A hart in a critical error state does not enter Debug Mode but instead asserts the critical-error signal to the platform. 1 (enabled): A hart in a critical error state enters Debug Mode instead of asserting the critical-error signal to the platform. Upon such entry into Debug Mode, the cause field is set to 7, and the extcause field is set to 0, indicating a critical error triggered the Debug Mode entry. This cause has the highest priority among all reasons for entering Debug Mode. Resuming from Debug Mode following an entry from the critical error state returns the hart to the critical error state. *When cetrig is 1, resuming from Debug Mode following an entry due to a critical error will result in an immediate re-entry into Debug Mode due to the critical error. The debugger may resume with cetrig set to 0 to allow the platform defined actions on critical-error signal to occur. Other possible actions include initiating a hart or platform reset using the Debug Module reset control.*	R/W	0
`ebreakvs`	0 (exception): `ebreak` instructions in VS-mode behave as described in the Privileged Spec. 1 (debug mode): `ebreak` instructions in VS-mode enter Debug Mode. This bit is hardwired to 0 if the hart does not support virtualization mode.	WARL	0
`ebreakvu`	0 (exception): `ebreak` instructions in VU-mode behave as described in the Privileged Spec. 1 (debug mode): `ebreak` instructions in VU-mode enter Debug Mode. This bit is hardwired to 0 if the hart does not support virtualization mode.	WARL	0
`ebreakm`	0 (exception): `ebreak` instructions in M-mode behave as described in the Privileged Spec. 1 (debug mode): `ebreak` instructions in M-mode enter Debug Mode.	R/W	0

Field	Description	Access	Reset
`ebreaks`	0 (exception): `ebreak` instructions in S-mode behave as described in the Privileged Spec. 1 (debug mode): `ebreak` instructions in S-mode enter Debug Mode. This bit is hardwired to 0 if the hart does not support S-mode.	WARL	0
`ebreaku`	0 (exception): `ebreak` instructions in U-mode behave as described in the Privileged Spec. 1 (debug mode): `ebreak` instructions in U-mode enter Debug Mode. This bit is hardwired to 0 if the hart does not support U-mode.	WARL	0
`stepie`	0 (interrupts disabled): Interrupts (including NMI) are disabled during single stepping with step set. This value should be supported. 1 (interrupts enabled): Interrupts (including NMI) are enabled during single stepping with step set. Implementations may hard wire this bit to 0. In that case interrupt behavior can be emulated by the debugger. The debugger must not change the value of this bit while the hart is running.	WARL	0
`stopcount`	0 (normal): Increment counters as usual. 1 (freeze): Don't increment any hart-local counters while in Debug Mode or on `ebreak` instructions that cause entry into Debug Mode. These counters include the `instret` CSR. On single-hart cores `cycle` should be stopped, but on multi-hart cores it must keep incrementing. An implementation may hardwire this bit to 0 or 1.	WARL	Preset
`stoptime`	0 (normal): `time` continues to reflect `mtime`. 1 (freeze): `time` is frozen at the time that Debug Mode was entered. When leaving Debug Mode, `time` will reflect the latest value of `mtime` again. While all harts have stoptime=1 and are in Debug Mode, `mtime` is allowed to stop incrementing. An implementation may hardwire this bit to 0 or 1.	WARL	Preset

Field	Description	Access	Reset
`cause`	Explains why Debug Mode was entered. When there are multiple reasons to enter Debug Mode in a single cycle, hardware should set cause to the cause with the highest priority. See Table 8 for priorities. 1 (ebreak): An **`ebreak`** instruction was executed. 2 (trigger): A Trigger Module trigger fired with action=1. 3 (haltreq): The debugger requested entry to Debug Mode using haltreq. 4 (step): The hart single stepped because step was set. 5 (resethaltreq): The hart halted directly out of reset due to resethaltreq It is also acceptable to report 3 when this happens. 6 (group): The hart halted because it's part of a halt group. Harts may report 3 for this cause instead. 7 (other): The hart halted for a reason other than the ones mentioned above. extcause may contain a more specific reason.	R	0
`v`	Extends the prv field with the virtualization mode the hart was operating in when Debug Mode was entered. The encoding is described in Table 11. A debugger can change this value to change the hart's virtualization mode when exiting Debug Mode. This bit is hardwired to 0 on harts that do not support virtualization mode.	WARL	0
`mprven`	0 (disabled): **`mprv`** in **`mstatus`** is ignored in Debug Mode. 1 (enabled): **`mprv`** in **`mstatus`** takes effect in Debug Mode. Implementing this bit is optional. It may be tied to either 0 or 1.	WARL	Preset
`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
`step`	When set and not in Debug Mode, the hart will only execute a single instruction and then enter Debug Mode. See Section 4.5.1 for details. The debugger must not change the value of this bit while the hart is running.	R/W	0

Field	Description	Access	Reset
`prv`	Contains the privilege mode the hart was operating in when Debug Mode was entered. The encoding is described in Table 11. A debugger can change this value to change the hart's privilege mode when exiting Debug Mode. Not all privilege modes 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 mode.	WARL	3

4.9.2. 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 9.

Table 9. Virtual address in DPC.

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	The address of the next instruction to be executed at the time that debug mode was entered. If the trigger is mcontrol and timing is 0 or if the trigger is mcontrol6 and `hit1` is 0, this corresponds to the address of the instruction which caused the trigger to fire.
halt request	Address of the next instruction to be executed at the time that debug mode was entered.

Executing the Program Buffer may cause the value of dpc to become UNSPECIFIED. 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 dpc to become UNSPECIFIED upon Program Buffer execution allows for direct implementations that don't have a separate PC register, and do need to use the PC when executing the Program Buffer.*

If the Access Register abstract command supports reading dpc while the hart is running, then the value read should be the address of a recently executed instruction.

If the Access Register abstract command supports writing dpc while the hart is running, then the executing program should jump to the written address shortly after the write occurs.

The writability of dpc follows the same rules as `mepc` as defined in the Privileged Spec. In particular, dpc must be able to hold all valid virtual addresses and the writability of the low bits depends on IALIGN.

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.

This CSR is read/write.

DXLEN-1 0
dpc
DXLEN

4.9.3. 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).

This CSR is read/write.

DXLEN-1 0
dscratch0
DXLEN

4.9.4. 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).

This CSR is read/write.

DXLEN-1 0
dscratch1
DXLEN

4.10. Virtual Debug Registers

A virtual register is one that doesn't exist directly in the hardware, but that the debugger exposes as if it does. Debug software should implement them, but hardware can skip this section. Virtual registers exist to give users access to functionality that's not part of standard debuggers without requiring them to carefully modify debug registers while the debugger is also accessing those same registers.

Table 10. Virtual Core Debug Registers

Address	Name	Section
virtual	Privilege Mode (priv)	Section 4.10.1

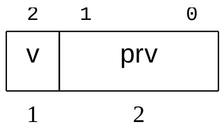
4.10.1. Privilege Mode (priv, at virtual)

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

This register contains prv and v 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 11. Privilege Mode and Virtualization Mode Encoding

H extension supported	v	prv	Abbreviation	Name
No	0	0	U-mode	User mode
No	0	1	S-mode	Supervisor mode
No	0	3	M-mode	Machine mode
Yes	0	0	U-mode	User mode
Yes	0	1	HS-mode	Hypervisor-enabled supervisor mode
Yes	0	3	M-mode	Machine mode
Yes	1	0	VU-mode	Virtual user mode
Yes	1	1	VS-mode	Virtual supervisor mode



Field	Description	Access	Reset
`v`	Contains the virtualization mode the hart was operating in when Debug Mode was entered. The encoding is described in Table 11, and matches the virtualization mode encoding from the Privileged Spec. A user can write this value to change the hart's virtualization mode when exiting Debug Mode.	WARL	0
`prv`	Contains the privilege mode the hart was operating in when Debug Mode was entered. The encoding is described in Table 11, and matches the privilege mode encoding from the Privileged Spec. A user can write this value to change the hart's privilege mode when exiting Debug Mode.	R/W	0

Chapter 5. Sdtrig (ISA Extension)

This chapter describes the Sdtrig ISA extension, which can be implemented independently of functionality described in the other chapters. It consists exclusively of the Trigger Module (TM).

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.

If Sdtrig is implemented, the Trigger Module must support at least one trigger. Accessing trigger CSRs that are not used by any of the implemented triggers must result in an illegal instruction exception. M-Mode and Debug Mode accesses to trigger CSRs that are used by any of the implemented triggers must succeed, regardless of the current type of the currently selected trigger.

A trigger matches when the conditions that it specifies (e.g. a load from a specific address) are met. A trigger fires when a trigger that matches performs the action configured for that trigger.

Triggers do not fire while in Debug Mode.

5.1. Enumeration

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. If this results in an illegal instruction exception, then there are no triggers implemented.
2. Read back tselect and check that it contains the written value. If not, exit the loop.
3. Read tinfo.
4. If that caused an exception, the debugger must read tdata1 to discover the type. (If type is 0, this trigger doesn't exist. Exit the loop.)
5. If info is 1, this trigger doesn't exist. Exit the loop.
6. Otherwise, the selected trigger supports the types discovered in info.
7. Repeat, incrementing the value in tselect.

The above algorithm reads back tselect so that implementations which have 2^n triggers only need to implement n bits of tselect.

The algorithm checks tinfo and type in case the implementation has m bits of tselect but fewer than 2^m triggers.

5.2. Actions

Triggers can be configured to take one of several actions when they fire. Table 12 lists all options.

Table 12. action encoding

Value	Description
0	Raise a breakpoint exception. (Used when software wants to use the trigger module without an external debugger attached.) `xepc` must contain the virtual address of the next instruction that must be executed to preserve the program flow.
1	Enter Debug Mode. dpc must contain the virtual address of the next instruction that must be executed to preserve the program flow. This action is only legal when the trigger's dmode is 1. Since tdata1 is WARL, hardware must prevent it from containing dmode=0 and action=1. This action can only be supported if `Sdext` is implemented on the hart.
2	Trace on, described in the trace specification.
3	Trace off, described in the trace specification.
4	Trace notify, described in the trace specification.
5	Reserved for use by the trace specification.
8 - 9	Send a signal to TM external trigger output 0 or 1 (respectively).
other	Reserved for future use.

Actions 8 and 9 are intended to increment custom event counters, but these signals could also be brought to outputs for use by external logic.

5.3. Priority

Table 13 lists the synchronous exceptions from the Privileged Spec, and where the various types of triggers fit in. The first 3 columns come from the Privileged Spec, and the final column shows where triggers fit in. Priorities in the table are separated by horizontal lines, so e.g. etrigger and itrigger have the same priority. If this table contradicts the table in the Privileged Spec, then the latter takes precedence.

This table only applies if triggers are precise. Otherwise triggers will fire some indeterminate time after the event, and the priority is irrelevant. When triggers are chained, the priority is the lowest priority of the triggers in the chain.

Table 13. Synchronous exception priority in decreasing priority order.

Priority	Exception Code	Description	Trigger
Highest	3 3 3 3		etrigger icount itrigger mcontrol/mcontrol6 after (on previous instruction)
	3	Instruction address breakpoint	mcontrol/mcontrol6 execute address before
	12, 20, 1	During instruction address translation: First encountered page fault, guest-page fault, or access fault	

Priority	Exception Code	Description	Trigger
	1	With physical address for instruction: Instruction access fault	
	3		mcontrol/mcontrol6 execute data before
	2 22 0 8, 9, 10, 11 3 3	Illegal instruction Virtual instruction Instruction address misaligned Environment call Environment break Load/Store/AMO address breakpoint	mcontrol/mcontrol6 load/store address before, store data before
	4, 6	Optionally: Load/Store/AMO address misaligned	
	13, 15, 21, 23, 5, 7	During address translation for an explicit memory access: First encountered page fault, guest-page fault, or access fault	
	5, 7	With physical address for an explicit memory access: Load/store/AMO access fault	
	4, 6	If not higher priority: Load/store/AMO address misaligned	
Lowest	3		mcontrol/mcontrol6 load data before

When multiple triggers in the same priority fire at once, hit (if implemented) is set for all of them. If more than one of these triggers has action=0 then **`tval`** is updated in accordance with one of them, but which one is UNSPECIFIED . If one of these triggers has the "enter Debug Mode" action (1) and another trigger has the "raise a breakpoint exception" action (0), the preferred behavior is to have both actions take place. It is implementation-dependent which of the two happens first. This ensures both that the presence of an external debugger doesn't affect execution and that a trigger set by user code doesn't affect the external debugger. If this is not implemented, then the hart must enter Debug Mode and ignore the breakpoint exception. In the latter case, hit of the trigger whose action is 0 must still be set, giving a debugger an opportunity to handle this case. What happens with trace actions when triggers with different actions are also firing is left to the trace specification.

5.4. Native Triggers

Triggers can be used for native debugging when action=0. If supported by the hart and desired by the debugger, triggers will often be programmed to have m=0 so that when they fire they cause a breakpoint exception to trap to a more privileged mode. That breakpoint exception can either be taken in M-mode or it can be delegated to a less privileged mode. However, it is possible for triggers to fire in the same mode that the resulting exception will be handled in.

In these cases such a trigger may cause a breakpoint exception while already in a trap handler. This might leave the hart unable to resume normal execution because state such as `mcause` and `mepc` would be overwritten.

> *In particular, when action=0:*
>
> 1. *mcontrol and mcontrol6 triggers with m=1 can cause a breakpoint exception that is taken from M-mode to M-mode (regardless of delegation).*
> 2. *mcontrol and mcontrol6 triggers with s=1 can cause a breakpoint exception that is taken from S-mode to S-mode if* `medeleg` *[3]=1.*
> 3. *mcontrol6 triggers with vs=1 can cause a breakpoint exception that is taken from VS-mode to VS-mode if* `medeleg` *[3]=1 and* `hedeleg` *[3]=1 .*
> 4. *icount triggers with m=1can cause a breakpoint exception that is taken from M-mode to M-mode (regardless of delegation).*
> 5. *icount triggers with s=1 can cause a breakpoint exception that is taken from S-mode to S-mode if* `medeleg` *[3]=1 .*
> 6. *icount triggers with vs=1 can cause a breakpoint exception that is taken from VS-mode to VS-mode if* `medeleg` *[3]=1 and* `hedeleg` *[3]=1.*
> 7. *etrigger and itrigger triggers will always be taken from a trap handler before the first instruction of the handler. If etrigger/itrigger is set to trigger on exception/interrupt X and if X is delegated to mode Y then the trigger will cause a breakpoint exception that is taken from mode Y to mode Y unless breakpoint exceptions are delegated to a more privileged mode than Y.*
> 8. *tmexttrigger triggers are asynchronous and may occur in any mode and at any time.*

Harts that support triggers with action=0 should implement one of the following two solutions to solve the problem of reentrancy:

1. The hardware prevents triggers with action=0 from matching or firing while in M-mode and while `MIE` in `mstatus` is 0. If `medeleg` [3]=1 then it prevents triggers with action=0 from matching or firing while in S-mode and while `SIE` in `sstatus` is 0. If `medeleg` [3]=1 and `hedeleg` [3]=1 then it prevents triggers with action=0 from matching or firing while in VS-mode and while `SIE` in `vstatus` is 0.
2. mte and mpte in tcontrol is implemented. `medeleg` [3] is hard-wired to 0.

> *The first option has the limitation that interrupts might be disabled at times when a user still might want triggers to fire. It has the benefit that breakpoints are not required to be handled in M-mode.*
>
> *The second option has the benefit that it only disables triggers during the trap handler, though it requires specific software support for this debug feature in the M-mode trap handlers. It can only work if breakpoints are not delegated to less privileged modes and therefore targets primarily implementations without S-mode.*
>
> *Because tcontrol is not accessible to S-mode, the second option can not be extended to accommodate delegation without adding additional S-mode and VS-mode CSRs.*
>
> *Both options prevent etrigger and itrigger from having any effect on exceptions and interrupts that are handled in M-mode. They also prevent triggering during some initial portion of each handler. Debuggers should use other mechanisms to debug these cases,*

> *such as patching the handler or setting a breakpoint on the instruction after* `MIE` *is cleared.*

5.5. Memory Access Triggers

mcontrol and mcontrol6 both enable triggers on memory accesses. This section describes for both of them how certain corner cases are treated.

5.5.1. A Extension

If the A extension is supported, then triggers on loads/stores treat them as follows:

1. `lr` instructions are loads.
2. Successful `sc` instructions are stores.
3. It is UNSPECIFIED whether failing `sc` instructions are stores or not.
4. Each AMO instruction is a load for the read portion of the operation. The address is always available to trigger on, although the value loaded might not be, depending on the hardware implementation.
5. Each AMO instruction is a store for the write portion of the operation. The address is always available to trigger on. Whether data store triggers match on AMOs is UNSPECIFIED.
6. If the destination register of any load or AMO is `zero` then it is UNSPECIFIED whether a data load trigger will match.

5.5.2. Combined Accesses

Some instructions lead a hart to perform multiple memory accesses. This includes vector loads and stores, as well as `cm.push` and `cm.pop` instructions. The Trigger Module should match such accesses as if they all happened individually. E.g. a vector load should be treated as if it performed multiple loads of size SEW (selected element width), and `cm.push` should be treated as if it performed multiple stores of size XLEN.

5.5.3. Cache Operations

Cache operations are infrequently performed, and code that uses them can have hard-to-find bugs. For the purposes of debug triggers, two classes of cache operations must match as stores:

1. Cache operations that enable software to maintain coherence between otherwise non-coherent implicit and explicit memory accesses.
2. Cache operations that perform block writes of constant data.

Only triggers with size=0 and select=0 will match. Since cache operations affect multiple addresses, there are multiple possible values to compare against. Implementations must implement one of the following options. From most desirable to least desirable, they are:

1. Every address from the effective address rounded down to the nearest cache block boundary (inclusive) to the effective address rounded up to the nearest cache block boundary (exclusive) is a compare value.
2. The effective address rounded down to the nearest cache block boundary is a compare value.

3. The effective address of the instruction is a compare value.

Cache operations encoded as HINTs do not match debug triggers.

> *The above language intends to capture the trigger behavior with respect to the cache operations to be introduced in a forthcoming I/D consistency extension.*
>
> *For RISC-V Base Cache Management Operation ISA Extensions 1.0.1, this means the following:*
>
> 1. `cbo.clean`, `cbo.flush`, *and* `cbo.inval` *match as if they are stores because they affect consistency.*
> 2. `cbo.zero` *matches as if it is a store because it performs a block write of constant data.*
> 3. *The prefetch instructions don't match at all.*

5.5.4. Address Matches

For address matches without a mask, tdata2 must be able to hold all valid addresses in all supported translation modes. That means that after writing any of these valid addresses, the exact same value XLEN-wide value is read back, including any high bits. An implementation may be able to optimize the storage required, depending on the widest addresses it supports.

> *If physical addresses are less than XLEN bits wide, they are zero-extended. If virtual addresses are less than XLEN bits wide, they are sign-extended. tdata2 must be implemented with enough bits of storage to represent the full range of supported physical and virtual address values when read by software and used by hardware.*

5.5.4.1. Invalid Addresses

If tdata2 can hold any invalid addresses, then writes of an invalid address that can not be represented as-is should be converted to a different invalid address that can be represented.

For invalid instruction fetch addresses and load and store effective addresses, the compare value may be changed to a different invalid address.

In addition, an implementation may choose to inhibit all trigger matching against invalid addresses, especially if there is no support for storage of any invalid address values in tdata2.

5.6. Multiple State Change Instructions

An instruction that performs multiple architectural state changes (e.g., register updates and/or memory accesses) might cause a trigger to fire at an intermediate point in its execution. As a result, architectural state changes up to that point might have been performed, while subsequent state changes, starting from the event that activated the trigger, might not have been. The definition of such an instruction will specify the order in which architectural state changes take place. Alternatively, it may state that partial execution is not allowed, implying that a mid-execution trigger must prevent any architectural state changes from occurring.

Debuggers won't be aware if an instruction has been partially executed. When they resume execution, they will execute the same instruction once more. Therefore, it's crucial that partially executing the instruction and then executing it again leaves the hart in a state closely resembling the state it would have been in if the instruction had only been executed once.

5.7. Trigger Module Registers

These registers are CSRs, accessible using the RISC-V `csr` opcodes and optionally also using abstract debug commands. They are the only mechanism to access the triggers.

Almost all trigger functionality is optional. All `tdata` registers follow write-any-read-legal semantics. If a debugger writes an unsupported configuration, the register will read back a value that is supported (which may simply be a disabled trigger). This means that a debugger must always read back values it writes to `tdata` registers, unless it already knows what is supported. Writes to one `tdata` register must not modify the contents of other `tdata` registers, nor the configuration of any trigger besides the one that is currently selected.

The combination of these rules means that a debugger cannot simply set a trigger by writing tdata1, then tdata2, etc. The current value of tdata2 might not be legal with the new value of tdata1. To help with this situation, it is guaranteed that writing 0 to tdata1 disables the trigger, and leaves it in a state where tdata2 and tdata3 can be written with any value that makes sense for any trigger type supported by this trigger.

As a result, a debugger can write any supported trigger as follows:

1. Write 0 to tdata1. (This will result in containing a non-zero value, since the register is WARL.)
2. Write desired values to tdata2 and tdata3.
3. Write desired value to tdata1.

Code that restores CSR context of triggers that might be configured to fire in the current privilege mode must use this same sequence to restore the triggers. This avoids the problem of a partially written trigger firing at a different time than is expected.

Attempts to access an unimplemented Trigger Module Register raise an illegal instruction exception.

The Trigger Module registers, except mscontext, scontext, and hcontext, are only accessible in machine and Debug Mode to prevent untrusted user code from causing entry into Debug Mode without the OS's permission.

In this section XLEN refers to the effective XLEN in the current execution mode. On systems where XLEN values can differ between modes, this is handled as follows. Fields retain their values regardless of XLEN, which only affects where in the register these fields appear (e.g. type). Some fields are wider when XLEN is 64 than when it is 32 (e.g. svalue). The high bits in such fields retain their value but are not readable when XLEN is 32. A modification of a register when XLEN is 32 clears any inaccessible bits in that register.

Table 14. Trigger Module Registers

Address	Name	Section
0x5a8	Supervisor Context (scontext)	Section 5.7.8
0x6a8	Hypervisor Context (hcontext)	Section 5.7.7
0x7a0	Trigger Select (tselect)	Section 5.7.1
0x7a1	Trigger Data 1 (tdata1)	Section 5.7.2
0x7a1	Match Control (mcontrol)	Section 5.7.11
0x7a1	Match Control Type 6 (mcontrol6)	Section 5.7.12

Address	Name	Section
0x7a1	Instruction Count (icount)	Section 5.7.13
0x7a1	Interrupt Trigger (itrigger)	Section 5.7.14
0x7a1	Exception Trigger (etrigger)	Section 5.7.15
0x7a1	External Trigger (tmexttrigger)	Section 5.7.16
0x7a2	Trigger Data 2 (tdata2)	Section 5.7.3
0x7a3	Trigger Data 3 (tdata3)	Section 5.7.4
0x7a3	Trigger Extra (RV32) (textra32)	Section 5.7.17
0x7a3	Trigger Extra (RV64) (textra64)	Section 5.7.18
0x7a4	Trigger Info (tinfo)	Section 5.7.5
0x7a5	Trigger Control (tcontrol)	Section 5.7.6
0x7a8	Machine Context (mcontext)	Section 5.7.9
0x7aa	Machine Supervisor Context (mscontext)	Section 5.7.10

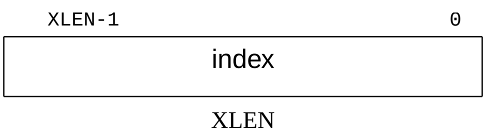
5.7.1. Trigger Select (tselect, at 0x7a0)

This register determines which trigger is accessible through the other Trigger Module registers. It is optional if no triggers are implemented. The set of accessible triggers must start at 0, and be contiguous.

This register is **WARL**. 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 or may point to a trigger where type=0. To verify that what they wrote is a valid index, debuggers can read back the value and check that tselect holds what they wrote and read tdata1 to see that type is non-zero.

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

This CSR is read/write.



5.7.2. Trigger Data 1 (tdata1, at 0x7a1)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is optional if no triggers are implemented.

Writing 0 to this register must result in a trigger that is disabled. If this trigger supports multiple types, then the hardware should disable it by changing type to 15.

This CSR is read/write.

XLEN-1 XLEN-4	XLEN-5	XLEN-6 0
type	dmode	data
4	1	XLEN - 5

Field	Description	Access	Reset
`type`	0 (none): There is no trigger at this tselect. 1 (legacy): The trigger is a legacy SiFive address match trigger. These should not be implemented and aren't further documented here. 2 (mcontrol): The trigger is an address/data match trigger. The remaining bits in this register act as described in mcontrol. 3 (icount): The trigger is an instruction count trigger. The remaining bits in this register act as described in icount. 4 (itrigger): The trigger is an interrupt trigger. The remaining bits in this register act as described in itrigger. 5 (etrigger): The trigger is an exception trigger. The remaining bits in this register act as described in etrigger. 6 (mcontrol6): The trigger is an address/data match trigger. The remaining bits in this register act as described in mcontrol6. This is similar to a type 2 trigger, but provides additional functionality and should be used instead of type 2 in newer implementations. 7 (tmexttrigger): The trigger is a trigger source external to the TM. The remaining bits in this register act as described in tmexttrigger. 12—14 (custom): These trigger types are available for non-standard use. 15 (disabled): This trigger is disabled. In this state, tdata2 and tdata3 can be written with any value that is supported for any of the types this trigger implements. The remaining bits in this register, except for dmode, are ignored. Other values are reserved for future use.	WARL	Preset

Field	Description	Access	Reset
`dmode`	If type is 0, then this bit is hard-wired to 0. 0 (both): Both Debug and M-mode can write the `tdata` registers at the selected tselect. 1 (dmode): 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. In ordinary use, external debuggers will always set this bit when configuring a trigger. When clearing this bit, debuggers should also set the action field (whose location depends on type) to something other than 1.	WARL	0
`data`	If type is 0, then this field is hard-wired to 0. Trigger-specific data.	WARL	Preset

5.7.3. Trigger Data 2 (tdata2, at 0x7a2)

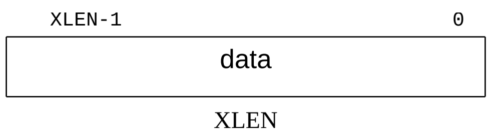
This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

Trigger-specific data. It is optional if no implemented triggers use it.

If the trigger is disabled, then this register can be written with any value supported by any of the trigger types supported by this trigger.

If XLEN is less than DXLEN, writes to this register are sign-extended.

This CSR is read/write.



5.7.4. Trigger Data 3 (tdata3, at 0x7a3)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

Trigger-specific data. It is optional if no implemented triggers use it.

If the trigger is disabled, then this register can be written with any value supported by any of the trigger types supported by this trigger.

If XLEN is less than DXLEN, writes to this register are sign-extended.

This CSR is read/write.

XLEN-1 0
data
XLEN

5.7.5. Trigger Info (tinfo, at 0x7a4)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is optional if no triggers are implemented, or if type is not writable and version would be 0. In this case the debugger can read the only supported type from tdata1.

Writing this read/write CSR has no effect.

XLEN-1 32	31 24	23 16	15 0
0	version	0	info
XLEN - 32	8	8	16

Field	Description	Access	Reset
`version`	Contains the version of the Sdtrig extension implemented. 0 (0): Supports triggers as described in this spec at commit 5a5c078, made on February 2, 2023. In these older versions: 1. mcontrol6 has a timing bit identical to timing 2. hit0 behaves just as hit. 3. hit1 is read-only 0. 4. Encodings for size for access sizes larger than 64 bits are different. 1 (1): Supports triggers as described in the ratified version 1.0 of this document.	R	Preset
`info`	One bit for each possible type enumerated in tdata1. Bit N corresponds to type N. If the bit is set, then that type is supported by the currently selected trigger. If the currently selected trigger doesn't exist, this field contains 1.	R	Preset

5.7.6. Trigger Control (tcontrol, at 0x7a5)

This optional register is only accessible in M-mode and Debug Mode and provides various control bits related to triggers.

This CSR is read/write.

XLEN-1 8	7	6 4	3	2 0
0	mpte	0	mte	0
XLEN - 8	1	3	1	3

Field	Description	Access	Reset
`mpte`	M-mode previous trigger enable field. mpte and mte provide one solution to a problem regarding triggers with action=0 firing in M-mode trap handlers. See Section 5.4 for more details. When any trap into M-mode is taken, mpte is set to the value of mte.	WARL	0
`mte`	M-mode trigger enable field. 0 (disabled): Triggers with action=0 do not match/fire while the hart is in M-mode. 1 (enabled): Triggers do match/fire while the hart is in M-mode. When any trap into M-mode is taken, mte is set to 0. When `mret` is executed, mte is set to the value of mpte.	WARL	0

5.7.7. Hypervisor Context (hcontext, at 0x6a8)

This optional register may be implemented only if the H extension is implemented. If it is implemented, mcontext must also be implemented.

This register is only accessible in HS-Mode, M-mode and Debug Mode. If Smstateen is implemented, then accessibility of in HS-Mode is controlled by `mstateenzero[57]`.

This register is an alias of the mcontext register, providing access to the hcontext field from HS-Mode.

5.7.8. Supervisor Context (scontext, at 0x5a8)

This optional register is only accessible in S/HS-mode, VS-mode, M-mode and Debug Mode.

Accessibility of this CSR is controlled by `mstateenzero[57]` and `hstateenzero[57]` in the Smstateen extension. Enabling scontext can be a security risk in a virtualized system with a hypervisor that does not swap scontext.

This CSR is read/write.

XLEN-1 32	31 0
0	data
XLEN - 32	32

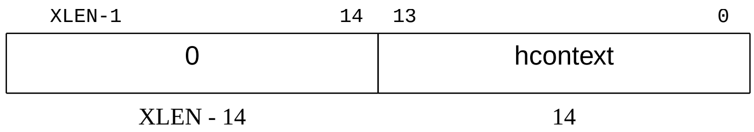
Field	Description	Access	Reset
`data`	Supervisor mode software can write a context number to this register, which can be used to set triggers that only fire in that specific context. An implementation may tie any number of high bits in this field to 0. It's recommended to implement 16 bits on RV32 and 32 bits on RV64.	WARL	0

5.7.9. Machine Context (mcontext, at 0x7a8)

This register must be implemented if hcontext is implemented, and is optional otherwise. It is only accessible in M-mode and Debug mode.

> *hcontext is primarily useful to set triggers on hypervisor systems that only fire when a given VM is executing. It is also useful in systems where M-Mode implements something like a hypervisor directly.*

This CSR is read/write.



Field	Description	Access	Reset
`hcontext`	M-Mode or HS-Mode (using hcontext) software can write a context number to this register, which can be used to set triggers that only fire in that specific context. An implementation may tie any number of upper bits in this field to 0. If the H extension is not implemented, it's recommended to implement 6 bits on RV32 and 13 bits on RV64 (as visible through the mcontext register). If the H extension is implemented, it's recommended to implement 7 bits on RV32 and 14 bits on RV64.	WARL	0

5.7.10. Machine Supervisor Context (mscontext, at 0x7aa)

This optional register is an alias for scontext. It is only accessible in S/HS-mode, M-mode and Debug Mode. It is included for backward compatibility with version 0.13.

> *The encoding of this CSR does not conform to the CSR Address Mapping Convention in the Privileged Spec. It is expected that new implementations will not support this encoding and that new debuggers will not use this CSR if scontext is available.*

5.7.11. Match Control (mcontrol, at 0x7a1)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata1 when type is 2. This trigger type is deprecated. It is included for backward compatibility with version 0.13.

This trigger type only supports a subset of features of the newer mcontrol6. It is expected that new implementations will not support this trigger type and that new debuggers will not use it if mcontrol6 is available.

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 15 suggests timings for the best user experience.

A chain of triggers that don't all have the same timing value will never fire. That means to implement the suggestions in Table 15, both timings should be supported on load address triggers that can be chained with a load data trigger.

The Privileged Spec says that breakpoint exceptions that occur on instruction fetches, loads, or stores update the `tval` CSR with either zero or the faulting virtual address. The faulting virtual address for an mcontrol trigger with action=0 is the address being accessed and which caused that trigger to fire. If multiple mcontrol triggers are chained then the faulting virtual address is the address which caused any of the chained triggers to fire.

If textra32 or textra64 are implemented for this trigger, it only matches when the conditions set there are satisfied.

This CSR is read/write.

XLEN-1 XLEN-4	XLEN-5	XLEN-6 XLEN-11	XLEN-12 23	22 21	20	19	18
type	dmode	maskmax	0	sizehi	hit	select	timing
4	1	6	XLEN - 34	2	1	1	1

17 16	15 12	11	10 7	6	5	4	3	2	1	0
sizelo	action	chain	match	m	0	s	u	execute	store	load
2	4	1	4	1	1	1	1	1	1	1

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 match 1 is not supported. A value of 63 corresponds to the maximum NAPOT range, which is 2^{63} bytes in size.	R	Preset
`sizehi`	This field only exists when XLEN is at least 64. It contains the 2 high bits of the access size. The low bits come from sizelo. See sizelo for how this is used.	WARL	0

Field	Description	Access	Reset
`hit`	If this bit is implemented then it must become set when this trigger fires and may become set when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	WARL	0
`select`	This bit determines the contents of the XLEN-bit compare values. 0 (address): There is at least one compare value and it contains the lowest virtual address of the access. It is recommended that there are additional compare values for the other accessed virtual addresses. (E.g. on a 32-bit read from 0x4000, the lowest address is 0x4000 and the other addresses are 0x4001, 0x4002, and 0x4003.) 1 (data): There is exactly one compare value and it contains the data value loaded or stored, or the instruction executed. Any bits beyond the size of the data access will contain 0.	WARL	0

Field	Description	Access	Reset
`timing`	0 (before): The action for this trigger will be taken just before the instruction that triggered it is retired, but after all preceding instructions are retired. `xepc` or dpc (depending on action) must be set to the virtual address of the instruction that matched. If this is combined with load and select=1 then a memory access will be performed (including any side effects of performing such an access) even though the load will not update its destination register. Debuggers should consider this when setting such breakpoints on, for example, memory-mapped I/O addresses. If an instruction matches this trigger and the instruction performs multiple memory accesses, it is UNSPECIFIED which memory accesses have completed before the trigger fires. 1 (after): The action for this trigger will be taken after the instruction that triggered it is retired. It should be taken before the next instruction is retired, but it is better to implement triggers imprecisely than to not implement them at all. `xepc` or dpc (depending on action) must be set to the virtual address of the next instruction that must be executed to preserve the program flow. 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. If a trigger with timing of 0 matches, it is implementation-dependent whether that prevents a trigger with timing of 1 matching as well.	WARL	0

Field	Description	Access	Reset
`sizelo`	This field contains the 2 low bits of the access size. The high bits come from sizehi. The combined value is interpreted as follows: 0 (any): The trigger will attempt to match against an access of any size. The behavior is only well-defined if select=0, or if the access size is XLEN. 1 (8bit): The trigger will only match against 8-bit memory accesses. 2 (16bit): The trigger will only match against 16-bit memory accesses or execution of 16-bit instructions. 3 (32bit): The trigger will only match against 32-bit memory accesses or execution of 32-bit instructions. 4 (48bit): The trigger will only match against execution of 48-bit instructions. 5 (64bit): The trigger will only match against 64-bit memory accesses or execution of 64-bit instructions. 6 (80bit): The trigger will only match against execution of 80-bit instructions. 7 (96bit): The trigger will only match against execution of 96-bit instructions. 8 (112bit): The trigger will only match against execution of 112-bit instructions. 9 (128bit): The trigger will only match against 128-bit memory accesses or execution of 128-bit instructions. An implementation must support the value of 0, but all other values are optional. When an implementation supports address triggers (select=0), it is recommended that those triggers support every access size that the hart supports, as well as for every instruction size that the hart supports. Implementations such as RV32D or RV64V are able to perform loads and stores that are wider than XLEN. Custom extensions may also support instructions that are wider than XLEN. Because tdata2 is of size XLEN, there is a known limitation that data value triggers (select=1) can only be supported for access sizes up to XLEN bits. When an implementation supports data value triggers (select=1), it is recommended that those triggers support every access size up to XLEN that the hart supports, as well as for every instruction length up to XLEN that the hart supports.	WARL	0

Field	Description	Access	Reset
`action`	The action to take when the trigger fires. The values are explained in Table 12.	WARL	0
`chain`	0 (disabled): When this trigger matches, the configured action is taken. 1 (enabled): While this trigger does not match, it prevents the trigger with the next index from matching. A trigger chain starts on the first trigger with `chain`=1 after a trigger with `chain`=0, or simply on the first trigger if that has `chain`=1. It ends on the first trigger after that which has `chain`=0. This final trigger is part of the chain. The action on all but the final trigger is ignored. The action on that final trigger will be taken if and only if all the triggers in the chain match at the same time. Debuggers should not terminate a chain with a trigger with a different type. It is undefined when exactly such a chain fires. Because chain affects the next trigger, hardware must zero it in writes to mcontrol that set dmode to 0 if the next trigger has dmode of 1. In addition hardware should ignore writes to mcontrol that set dmode to 1 if the previous trigger has both dmode of 0 and chain of 1. Debuggers must avoid the latter case by checking chain on the previous trigger if they're writing mcontrol. Implementations that wish to limit the maximum length of a trigger chain (eg. to meet timing requirements) may do so by zeroing chain in writes to mcontrol that would make the chain too long.	WARL	0

Field	Description	Access	Reset
`match`	0 (equal): Matches when any compare value equals tdata2. 1 (napot): Matches when the top `M` bits of any compare value match the top `M` bits of tdata2. `M` is `XLEN-1` minus the index of the least-significant bit containing 0 in tdata2. Debuggers should only write values to tdata2 such that `M` + maskmax ≥ `XLEN` and `M` > 0, otherwise it's undefined on what conditions the trigger will match. 2 (ge): Matches when any compare value is greater than (unsigned) or equal to tdata2. 3 (lt): Matches when any compare value is less than (unsigned) tdata2. 4 (mask low): Matches when $\frac{XLEN}{2}-1:0$ of any compare value equals $\frac{XLEN}{2}-1:0$ of tdata2 after $\frac{XLEN}{2}-1:0$ of the compare value is ANDed with `XLEN-1`:$\frac{XLEN}{2}$ of tdata2. 5 (mask high): Matches when `XLEN-1`:$\frac{XLEN}{2}$ of any compare value equals $\frac{XLEN}{2}-1:0$ of tdata2 after `XLEN-1` :$\frac{XLEN}{2}$ of the compare value is ANDed with `XLEN-1` :$\frac{XLEN}{2}$ of tdata2. 8 (not equal): Matches when match=0 would not match. 9 (not napot): Matches when match=1 would not match. 12 (not mask low): Matches when match=4 would not match. 13 (not mask high): Matches when match=5 would not match. Other values are reserved for future use. All comparisons only look at the lower XLEN (in the current mode) bits of the compare values and of tdata2. When select=1 and access size is N, this is further reduced, and comparisons only look at the lower N bits of the compare values and of tdata2.	WARL	0
`m`	When set, enable this trigger in M-mode.	WARL	0
`s`	When set, enable this trigger in S/HS-mode. This bit is hard-wired to 0 if the hart does not support S-mode.	WARL	0
`u`	When set, enable this trigger in U-mode. This bit is hard-wired to 0 if the hart does not support U-mode.	WARL	0

Field	Description	Access	Reset
`execute`	When set, the trigger fires on the virtual address or opcode of an instruction that is executed.	WARL	0
`store`	When set, the trigger fires on the virtual address or data of any store.	WARL	0
`load`	When set, the trigger fires on the virtual address or data of any load.	WARL	0

5.7.12. Match Control Type 6 (mcontrol6, at 0x7a1)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata1 when type is 6.

Implementing this trigger as described here requires that version is 1 or higher, which in turn means tinfo must be implemented.

This replaces mcontrol in newer implementations and serves to provide additional functionality.

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.

Table 15 suggests timings for the best user experience. The underlying principle is that firing just before the instruction gives a user more insight, so is preferable. However, depending on the instruction and conditions, it might not be possible to evaluate the trigger until the instruction has partially executed. In that case it is better to let the instruction retire before the trigger fires, to avoid extra memory accesses which might affect the state of the system.

Table 15. Suggested Trigger 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

A chain of triggers must only fire if every trigger in the chain was matched by the same instruction.

The Privileged Spec says that breakpoint exceptions that occur on instruction fetches, loads, or stores update the `tval` CSR with either zero or the faulting virtual address. The faulting virtual address for an mcontrol6 trigger with action=0 is the address being accessed and which caused that trigger to fire.

If multiple mcontrol6 triggers are chained then the faulting virtual address is the address which caused any of the chained triggers to fire.

In implementations that support match mode 1 (NAPOT), not all NAPOT ranges may be supported. All NAPOT ranges between 2^1 and $2^{maskmax6}$ are supported where $maskmax6 \geq 1$. The value of maskmax6 can be determined by the debugger via the following sequence:

1. Set match=1.
2. Read match. If it is not 1 then NAPOT matching is not supported.
3. Write all ones to tdata2.
4. Read tdata2. The value of maskmax6 is the index of the most significant 0 bit plus 1.

If textra32 or textra64 are implemented for this trigger, it only matches when the conditions set there are satisfied.

uncertain and uncertainen exist to accommodate systems where not every memory access is fully observed by the Trigger Module. Possible examples include data values in far AMOs, and the address/data/size of accesses by instructions that perform multiple memory accesses, such as vector, push, and pop instructions.

While the uncertain mechanism exists to deal with these situations, it can lead to an unusable number of false positives. Users will get a much better debug experience if the TM does have perfect visibility into the details of every memory access.

This CSR is read/write.

XLEN-1 XLEN-4	XLEN-5	XLEN-6 27	26	25	24	23	22	21	20 19	18 16
type	dmode	0	uncertain	hit1	vs	vu	hit0	select	0	size
4	1	XLEN - 32	1	1	1	1	1	1	2	3

15 12	11	10 7	6	5	4	3	2	1	0
action	chain	match	m	uncertainen	s	u	execute	store	load
4	1	4	1	1	1	1	1	1	1

Field	Description	Access	Reset
`uncertain`	If implemented, the TM updates this field every time the trigger fires. 0 (certain): The trigger that fired satisfied the configured conditions, or this bit is not implemented. 1 (uncertain): The trigger that fired might not have perfectly satisfied the configured conditions. Due to the implementation the hardware cannot be certain.	WARL	0
`vs`	When set, enable this trigger in VS-mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0
`vu`	When set, enable this trigger in VU-mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0

Field	Description	Access	Reset
`hit0`	If they are implemented, hit1 (MSB) and hit0 (LSB) combine into a single 2-bit field. The TM updates this field when the trigger fires. After the debugger has seen the update, it will normally write 0 to this field to so it can see future changes. If either of the bits is not implemented, the unimplemented bits will be read-only 0. 0 (false): The trigger did not fire. 1 (before): The trigger fired before the instruction that matched it was retired, but after all preceding instructions are retired. This explicitly allows for instructions to be partially executed, as described in Section 5.6. `xepc` or dpc (depending on action) must be set to the virtual address of the instruction that matched. 2 (after): The trigger fired after the instruction that triggered and at least one additional instruction were retired. `xepc` or dpc (depending on action) must be set to the virtual address of the next instruction that must be executed to preserve the program flow. 3 (immediately after): The trigger fired just after the instruction that triggered it was retired, but before any subsequent instructions were executed. `xepc` or dpc (depending on action) must be set to the virtual address of the next instruction that must be executed to preserve the program flow. If the instruction performed multiple memory accesses, all of them have been completed.	WARL	0
`select`	This bit determines the contents of the XLEN-bit compare values. 0 (address): There is at least one compare value and it contains the lowest virtual address of the access. In addition, it is recommended that there are additional compare values for the other accessed virtual addresses match. (E.g. on a 32-bit read from 0x4000, the lowest address is 0x4000 and the other addresses are 0x4001, 0x4002, and 0x4003.) 1 (data): There is exactly one compare value and it contains the data value loaded or stored, or the instruction executed. Any bits beyond the size of the data access will contain 0.	WARL	0

Field	Description	Access	Reset
`size`	0 (any): The trigger will attempt to match against an access of any size. The behavior is only well-defined if select=0, or if the access size is XLEN. 1 (8bit): The trigger will only match against 8-bit memory accesses. 2 (16bit): The trigger will only match against 16-bit memory accesses or execution of 16-bit instructions. 3 (32bit): The trigger will only match against 32-bit memory accesses or execution of 32-bit instructions. 4 (48bit): The trigger will only match against execution of 48-bit instructions. 5 (64bit): The trigger will only match against 64-bit memory accesses or execution of 64-bit instructions. 6 (128bit): The trigger will only match against 128-bit memory accesses or execution of 128-bit instructions. An implementation must support the value of 0, but all other values are optional. When an implementation supports address triggers (select=0), it is recommended that those triggers support every access size that the hart supports, as well as for every instruction size that the hart supports. Implementations such as RV32D or RV64V are able to perform loads and stores that are wider than XLEN. Custom extensions may also support instructions that are wider than XLEN. Because tdata2 is of size XLEN, there is a known limitation that data value triggers (select=1) can only be supported for access sizes up to XLEN bits. When an implementation supports data value triggers (select=1), it is recommended that those triggers support every access size up to XLEN that the hart supports, as well as for every instruction length up to XLEN that the hart supports.	WARL	0
`action`	The action to take when the trigger fires. The values are explained in Table 12.	WARL	0

Field	Description	Access	Reset
`chain`	0 (disabled): When this trigger matches, the configured action is taken. 1 (enabled): While this trigger does not match, it prevents the trigger with the next index from matching. A trigger chain starts on the first trigger with `chain`=1 after a trigger with `chain`=0, or simply on the first trigger if that has `chain`=1. It ends on the first trigger after that which has `chain`=0. This final trigger is part of the chain. The action on all but the final trigger is ignored. The action on that final trigger will be taken if and only if all the triggers in the chain match at the same time. Debuggers should not terminate a chain with a trigger with a different type. It is undefined when exactly such a chain fires. Because chain affects the next trigger, hardware must zero it in writes to mcontrol6 that set dmode to 0 if the next trigger has dmode of 1. In addition hardware should ignore writes to mcontrol6 that set dmode to 1 if the previous trigger has both dmode of 0 and chain of 1. Debuggers must avoid the latter case by checking chain on the previous trigger if they're writing mcontrol6. Implementations that wish to limit the maximum length of a trigger chain (eg. to meet timing requirements) may do so by zeroing chain in writes to mcontrol6 that would make the chain too long.	WARL	0

Field	Description	Access	Reset
`match`	0 (equal): Matches when any compare value equals tdata2. 1 (napot): Matches when the top `M` bits of any compare value match the top `M` bits of tdata2. `M` is `XLEN-1` minus the index of the least-significant bit containing 0 in tdata2. tdata2 is WARL and if bits `maskmax6-1:0` are written with all ones then bit `maskmax6-1` will be set to 0 while the values of bits `maskmax6-2:0` are UNSPECIFIED. Legal values for tdata2 require M + `maskmax6` ≥ `XLEN` and `M` > 0. See above for how to determine maskmax6. 2 (ge): Matches when any compare value is greater than (unsigned) or equal to tdata2. 3 (lt): Matches when any compare value is less than (unsigned) tdata2. 4 (mask low): Matches when $\frac{XLEN}{2}-1:0$ of any compare value equals $\frac{XLEN}{2}-1:0$ of tdata2 after $\frac{XLEN}{2}-1:0$ of the compare value is ANDed with `XLEN-1`:$\frac{XLEN}{2}$ of tdata2. 5 (mask high): Matches when `XLEN-1`:$\frac{XLEN}{2}$ of any compare value equals $\frac{XLEN}{2}-1:0$ of tdata2 after `XLEN-1`:$\frac{XLEN}{2}$ of the compare value is ANDed with `XLEN-1`:$\frac{XLEN}{2}$ of tdata2. 8 (not equal): Matches when match `=0` would not match. 9 (not napot): Matches when match `=1` would not match. 12 (not mask low): Matches when match `=4` would not match. 13 (not mask high): Matches when match `=5` would not match. Other values are reserved for future use. All comparisons only look at the lower XLEN (in the current mode) bits of the compare values and of tdata2. When select=1 and access size is N, this is further reduced, and comparisons only look at the lower N bits of the compare values and of tdata2.	WARL	0
`m`	When set, enable this trigger in M-mode.	WARL	0

Field	Description	Access	Reset
`uncertainen`	0 (disabled): This trigger will only match if the hardware can perfectly evaluate it. 1 (enabled): This trigger will match if it's possible that it would match if the Trigger Module had perfect information about the operations being performed.	WARL	0
`s`	When set, enable this trigger in S/HS-mode. This bit is hard-wired to 0 if the hart does not support S-mode.	WARL	0
`u`	When set, enable this trigger in U-mode. This bit is hard-wired to 0 if the hart does not support U-mode.	WARL	0
`execute`	When set, the trigger fires on the virtual address or opcode of an instruction that is executed.	WARL	0
`store`	When set, the trigger fires on the virtual address or data of any store.	WARL	0
`load`	When set, the trigger fires on the virtual address or data of any load.	WARL	0

5.7.13. Instruction Count (icount, at 0x7a1)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata1 when type is 3.

This trigger matches when:

1. An instruction retires after having been fetched in a privilege mode where the trigger is enabled. This explicitly includes all RET instructions from various modes.
2. A trap is taken from a privilege mode where the trigger is enabled. This explicitly includes traps taken due to interrupts.

If more than one of the above events occur during a single instruction execution, the trigger still only matches once for that instruction.

> *For use in single step, icount must match for traps where the instruction will not be reexecuted after the handler, such as illegal instructions that are emulated by privileged software and the instruction being emulated never retires. Ideally, icount would not match for traps where the instruction will later be retried by the handler, such as page faults where privileged software modifies the page tables and returns to the faulting instruction which ultimately retires. Trying to distinguish the two cases leads to complex rules, so instead the rule is simply that all traps match. See also Section 4.5.2.*

When count is greater than 1 and the trigger matches, then count is decremented by 1.

When count is 1 and the trigger matches, then pending becomes set. In addition count will become 0 unless it is hard-wired to 1.

The only exception to the above is when the instruction the trigger matched on is a write to the icount trigger. In that case pending might or might not become set if count was 1. Afterwards count contains the newly written value.

When count is 0 it stays at 0 until explicitly written.

When pending is set, the trigger fires just before any further instructions are executed in a mode where the trigger is enabled. As the trigger fires, pending is cleared. In addition, if count is hard-wired to 1 then m, s, u, vs, and vu are all cleared.

If the trigger fires with action=0 then zero is written to the `tval` CSR on the breakpoint trap.

> *The intent of pending is to cleanly handle the case where action is 0, m is 0, u is 1, count is 1, and the U-mode instruction being executed causes a trap into M-mode. In that case we want the entire M-mode handler to be executed, and the debug trap to be taken before the next U-mode instruction.*

> *This trigger type is intended to be used as a single step for software monitor programs or native debug. Systems that support multiple privilege modes that want to debug software running in lower privilege modes don't need to support count greater than 1.*

If textra32 or textra64 are implemented for this trigger, it only matches when the conditions set there are satisfied.

This CSR is read/write.

XLEN-1 XLEN-4	XLEN-5	XLEN-6 27	26	25	24	23 10	9	8	7	6	5 0
type	dmode	0	vs	vu	hit	count	m	pending	s	u	action
4	1	XLEN - 32	1	1	1	14	1	1	1	1	6

Field	Description	Access	Reset
`vs`	When set, enable this trigger in VS-mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0
`vu`	When set, enable this trigger in VU-mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0
`hit`	If this bit is implemented, the hardware sets it when this trigger fires. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) fires. If the bit is not implemented, it is always 0 and writing it has no effect.	WARL	0
`count`	The trigger will generally fire after count instructions in enabled modes have been executed. See above for the precise behavior.	WARL	1
`m`	When set, enable this trigger in M-mode.	WARL	0
`pending`	This bit becomes set when count is decremented from 1 to 0. It is cleared when the trigger fires, which will happen just before executing the next instruction in one of the enabled modes.	R/W	0
`s`	When set, enable this trigger in S/HS-mode. This bit is hard-wired to 0 if the hart does not support S-mode.	WARL	0

Field	Description	Access	Reset
`u`	When set, enable this trigger in U-mode. This bit is hard-wired to 0 if the hart does not support U-mode.	WARL	0
`action`	The action to take when the trigger fires. The values are explained in Table 12.	WARL	0

5.7.14. Interrupt Trigger (itrigger, at 0x7a1)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata1 when type is 4.

This trigger can fire when an interrupt trap is taken.

It can be enabled for individual interrupt numbers by setting the bit corresponding to the interrupt number in tdata2. The interrupt number is interpreted in the mode that the trap handler executes in. (E.g. virtualized interrupt numbers are not the same in every mode.) In addition the trigger can be enabled for non-maskable interrupts using nmi.

> *If XLEN is 32, then it is not possible to set a trigger for interrupts with Exception Code larger than 31. A future version of the RISC-V Privileged Spec will likely define interrupt Exception Codes 32 through 47. Some of those numbers are already being used by the RISC-V Advanced Interrupt Architecture.*

Hardware may only support a subset of interrupts for this trigger. A debugger must read back tdata2 after writing it to confirm the requested functionality is actually supported.

When the trigger matches, it fires after the trap occurs, just before the first instruction of the trap handler is executed. If action=0, the standard CSRs are updated for taking the breakpoint trap, and zero is written to the relevant `tval` CSR. If the breakpoint trap does not go to a higher privilege mode, this will lose CSR information for the original trap. See Section 5.4 for more information about this case.

If textra32 or textra64 are implemented for this trigger, it only matches when the conditions set there are satisfied.

This CSR is read/write.

XLEN-1 XLEN-4	XLEN-5	XLEN-6	XLEN-7 13	12	11	10	9	8	7	6	5 0
type	dmode	hit	0	vs	vu	nmi	m	0	s	u	action
4	1	1	XLEN - 19	1	1	1	1	1	1	1	6

Field	Description	Access	Reset
`hit`	If this bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	WARL	0

Field	Description	Access	Reset
`vs`	When set, enable this trigger for interrupts that are taken from VS mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0
`vu`	When set, enable this trigger for interrupts that are taken from VU mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0
`nmi`	When set, non-maskable interrupts cause this trigger to fire if the trigger is enabled for the current mode.	WARL	0
`m`	When set, enable this trigger for interrupts that are taken from M mode.	WARL	0
`s`	When set, enable this trigger for interrupts that are taken from S/HS mode. This bit is hard-wired to 0 if the hart does not support S-mode.	WARL	0
`u`	When set, enable this trigger for interrupts that are taken from U mode. This bit is hard-wired to 0 if the hart does not support U-mode.	WARL	0
`action`	The action to take when the trigger fires. The values are explained in Table 12.	WARL	0

5.7.15. Exception Trigger (etrigger, at 0x7a1)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata1 when type is 5.

This trigger may fire on up to XLEN of the Exception Codes defined in `mcause` (described in the Privileged Spec, with Interrupt=0). Those causes are configured by writing the corresponding bit in tdata2. (E.g. to trap on an illegal instruction, the debugger sets bit 2 in tdata2.)

If XLEN is 32, then it is not possible to set a trigger on Exception Codes higher than 31. A future version of the RISC-V Privileged Spec will likely define Exception Codes 32 through 47.

Hardware may support only a subset of exceptions. A debugger must read back tdata2 after writing it to confirm the requested functionality is actually supported.

When the trigger matches, it fires after the trap occurs, just before the first instruction of the trap handler is executed. If action=0, the standard CSRs are updated for taking the breakpoint trap, and zero is written to the relevant `tval` CSR. If the breakpoint trap does not go to a higher privilege mode, this will lose CSR information for the original trap. See Section 5.4 for more information about this case.

If textra32 or textra64 are implemented for this trigger, it only matches when the conditions set there are satisfied.

This CSR is read/write.

XLEN-1 … XLEN-4	XLEN-5	XLEN-6	XLEN-7 … 13	12	11	10	9	8	7	6	5 … 0
type	dmode	hit	0	vs	vu	0	m	0	s	u	action
4	1	1	XLEN - 19	1	1	1	1	1	1	1	6

Field	Description	Access	Reset
`hit`	If this bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	WARL	0
`vs`	When set, enable this trigger for exceptions that are taken from VS mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0
`vu`	When set, enable this trigger for exceptions that are taken from VU mode. This bit is hard-wired to 0 if the hart does not support virtualization mode.	WARL	0
`m`	When set, enable this trigger for exceptions that are taken from M mode.	WARL	0
`s`	When set, enable this trigger for exceptions that are taken from S/HS mode. This bit is hard-wired to 0 if the hart does not support S-mode.	WARL	0
`u`	When set, enable this trigger for exceptions that are taken from U mode. This bit is hard-wired to 0 if the hart does not support U-mode.	WARL	0
`action`	The action to take when the trigger fires. The values are explained in Table 12.	WARL	0

5.7.16. External Trigger (tmexttrigger, at 0x7a1)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata1 when type is 7.

This trigger fires when any selected TM external trigger input signals. Up to 16 TM external trigger inputs coming from other blocks outside the TM, (e.g. signaling an hpmcounter overflow) can be selected. Hardware may support none or just a few TM external trigger inputs (starting with TM external trigger input 0 and continuing sequentially). Unsupported inputs are hardwired to be inactive.

If the trigger fires with action=0 then zero is written to the `tval` CSR on the breakpoint trap. This trigger fires asynchronously but it is subject to delegation by medeleg[3] like the other triggers.

The TM external trigger input can signal when the trigger is prevented from firing due to one of the mechanisms in Section 5.4. An implementation may either ignore the signal altogether when it cannot fire (dropping the trigger event) or it may hold the action as pending and fire the trigger once it is legal to do so.

intctl is intended to be used by the `clicinttrig` *mechanism from the Core-Local*

Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions.

This CSR is read/write.

XLEN-1 XLEN-4	XLEN-5	XLEN-6	XLEN-7 23	22	21 6	5 0
type	dmode	hit	0	intctl	select	action
4	1	1	XLEN - 29	1	16	6

Field	Description	Access	Reset
`hit`	If this bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	WARL	0
`intctl`	This optional bit, when set, causes this trigger to fire whenever an attached interrupt controller signals a trigger.	WARL	0
`select`	Selects any combination of up to 16 TM external trigger inputs that cause this trigger to fire.	WARL	0
`action`	The action to take when the trigger fires. The values are explained in Table 12.	WARL	0

5.7.17. Trigger Extra (RV32) (textra32, at 0x7a3)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata3 when type is 2, 3, 4, 5, or 6 and XLEN=32.

If DXLEN >= 64, then this register provides access to the low bits of each field defined in textra64. Writes to this register will clear the high bits of the corresponding fields in textra64.

All functionality in this register is optional. Any number of upper bits of mhvalue and svalue may be tied to 0. mhselect and sselect may only support 0 (ignore).

Byte-granular comparison of scontext to svalue allows scontext to be defined to include more than one element of comparison. For example, software instrumentation can program the scontext value to be the concatenation of different ID contexts such as process ID and thread ID. The user can then program byte compares based on sbytemask to include one or more of the contexts in the compare.

Byte masking only applies to scontext comparison; i.e when sselect is 1.

Note that sselect and mhselect filtering apply in all modes, including M-mode and S-mode. If desired, debuggers can use a trigger's mode filtering bits to restrict the matching to modes where it considers ASID/VMID/scontext/hcontext to be active.

This CSR is read/write.

31 26	25 23	22 20	19 18	17 2	1 0
mhvalue	mhselect	0	sbytemask	svalue	sselect
6	3	3	2	16	2

Field	Description	Access	Reset
`mhvalue`	Data used together with mhselect.	WARL	0
`mhselect`	0 (ignore): Ignore mhvalue. 4 (mcontext): This trigger will only match or fire if the low bits of mcontext/hcontext equal mhvalue. 1, 5 (mcontext_select): This trigger will only match or fire if the low bits of mcontext/hcontext equal {mhvalue, mhselect[2]}. 2, 6 (vmid_select): This trigger will only match or fire if VMID in hgatp equals the lower VMIDMAX (defined in the Privileged Spec) bits of {mhvalue, mhselect[2]}. 3, 7 (reserved): Reserved. If the H extension is not supported, the only legal values are 0 and 4.	WARL	0
`sbytemask`	When the least significant bit of this field is 1, it causes bits 7:0 in the comparison to be ignored, when sselect=1. When the next most significant bit of this field is 1, it causes bits 15:8 to be ignored in the comparison, when sselect=1.	WARL	0
`svalue`	Data used together with sselect. This field should be tied to 0 when S-mode is not supported.	WARL	0
`sselect`	0 (ignore): Ignore svalue. 1 (scontext): This trigger will only match or fire if the low bits of scontext equal svalue. 2 (asid): This trigger will only match or fire if: • the mode is VS-mode or VU-mode and ASID in `vsatp` equals the lower ASIDMAX (defined in the Privileged Spec) bits of svalue. • in all other modes, ASID in `satp` equals the lower ASIDMAX (defined in the Privileged Spec) bits of svalue. This field should be tied to 0 when S-mode is not supported.	WARL	0

5.7.18. Trigger Extra (RV64) (textra64, at 0x7a3)

This register provides access to the trigger selected by tselect. The reset values listed here apply to every underlying trigger.

This register is accessible as tdata3 when type is 2, 3, 4, 5, or 6 and XLEN=64. The function of the fields are defined above, in textra32. This register retains its value when XLEN changes. When XLEN=32 some of the bits can be accessed through textra32.

Byte-granular comparison of scontext to svalue in textra64 allows scontext to be defined to include more than one element of comparison. For example, software instrumentation can program the scontext value to be the concatenation of different ID contexts such as process ID and thread ID. The user can then program byte compares based on sbytemask to include one or more of the contexts in the compare.

Byte masking only applies to scontext comparison; i.e when sselect is 1.

This CSR is read/write.

63 51	50 48	47 40	39 36	35 34	33 2	1 0
mhvalue	mhselect	0	sbytemask	0	svalue	sselect
13	3	8	4	2	32	2

Field	Description	Access	Reset
`sbytemask`	When the least significant bit of this field is 1, it causes bits 7:0 in the comparison to be ignored, when sselect=1. Likewise, the second bit controls the comparison of bits 15:8, third bit controls the comparison of bits 23:16, and fourth bit controls the comparison of bits 31:24.	WARL	0

Chapter 6. Debug Transport Module (DTM) (non-ISA extension)

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

There may be multiple DTMs in a single hardware platform. Ideally every component that communicates with the outside world includes a DTM, allowing a hardware platform to be debugged through every transport it supports. For instance a USB component could include a DTM. This would trivially allow any hardware 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.

An implementation can be compatible with this specification without implementing any of this section. In that case it must be advertised as conforming to "RISC-V Debug Specification, with custom DTM." If the JTAG DTM described here is implemented, it must be advertised as conforming to the "RISC-V Debug Specification, with JTAG DTM.""

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 component's 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 16. If the IR actually has more than 5 bits, then the encodings in Table 16 should be extended with 0's in their most significant bits, except for the 0x1f encoding of BYPASS, which must be extended with 1's in the 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 16. JTAG DTM TAP Registers

Address	Name	Description	Section
0x00	bypass	JTAG recommends this encoding	
0x01	idcode	To identify a specific silicon version	Section 6.1.3
0x10	DTM Control and Status (dtmcs)	For Debugging	Section 6.1.4
0x11	Debug Module Interface Access (dmi)	For Debugging	Section 6.1.5
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	Section 6.1.6

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.

31–28	27–12	11–1	0
Version	PartNumber	ManufId	1
4	16	11	1

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.

31–21	20–18	17	16	15	14–12	11–10	9–4	3–0
0	errinfo	dtmhardreset	dmireset	0	idle	dmistat	abits	version
11	3	1	1	1	3	2	6	4

Field	Description	Access	Reset
`errinfo`	This optional field may provide additional detail about an error that occurred when communicating with a DM. It is updated whenever op is updated by the hardware or when 1 is written to dmireset. 0 (not implemented): This field is not implemented. 1 (dmi error): There was an error between the DTM and DMI. 2 (communication error): There was an error between the DMI and a DMI subordinate. 3 (device error): The DMI subordinate reported an error. 4 (unknown): There is no error to report, or no further information available about the error. This is the reset value if the field is implemented. Other values are reserved for future use by this specification.	R	4
`dtmhardreset`	Writing 1 to this bit does a hard reset of the DTM, causing the DTM to forget about any outstanding DMI transactions, and returning all registers and internal state to their reset value. 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	-
`dmireset`	Writing 1 to this bit clears the sticky error state and resets errinfo, but does not affect outstanding DMI transactions.	W1	-
`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`	Read-only alias of op.	R	0
`abits`	The size of address in dmi.	R	Preset

Field	Description	Access	Reset
`version`	0 (0.11): Version described in spec version 0.11. 1 (1.0): Version described in spec versions 0.13 and 1.0. 15 (custom): Version not described in any available version of this spec.	R	1

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

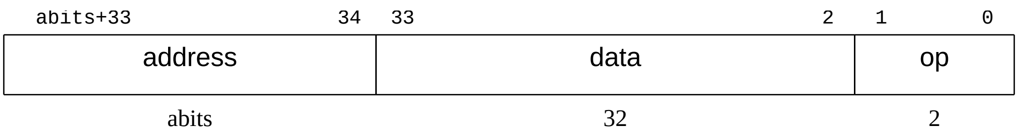
This register allows access to the Debug Module Interface (DMI).

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.2.1 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. op defines what this register contains after every possible operation.	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

Field	Description	Access	Reset
`op`	When the debugger writes this field, it has the following meaning: 0 (nop): Ignore data and address. 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. This operation leaves the values in address and data UNSPECIFIED. 1 (read): Read from address. When this operation succeeds, address contains the address that was read from, and data contains the value that was read. 2 (write): Write data to address. This operation leaves the values in address and data UNSPECIFIED. 3 (reserved): Reserved. When the debugger reads this field, it means the following: 0 (success): The previous operation completed successfully. 1 (reserved): Reserved. 2 (failed): 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 or the DMI responded with an error. There are no specified cases in which the DM would respond with an error, and DMI is not required to support returning errors. If a debugger sees this status, there might be additional information in errinfo. 3 (busy): 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.	R/W	0

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.

0
0
1

6.1.7. JTAG Connector

6.1.7.1. Recommended JTAG Connector

To make it easy to acquire debug hardware, this spec recommends a connector that is compatible with the MIPI-10 .05 inch connector specification, as described in MIPI Debug & Trace Connector Recommendations, Version 1.20, 2 July 2021.

The connector has .05 inch spacing, gold-plated male header with .016 inch thick hardened copper or beryllium bronze square posts (SAMTEC FTSH 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 17. The function of each pin is described in Table 18.

Table 17. MIPI 10-pin JTAG + nRESET Connector Diagram

VREF DEBUG	1	2	TMS
GND	3	4	TCK
GND	5	6	TDO
GND or KEY	7	8	TDI
GND	9	10	nRESET

If a hardware platform requires nTRST then it is permissible to reuse the nRESET pin as the nTRST signal, resulting in a MIPI 10-pin JTAG
nTRST connector.

6.1.7.2. Alternate JTAG Connector

The MIPI-10 connector should provide plenty of signals for all modern hardware. If a design does need legacy JTAG signals, then the MIPI-20 connector should be used. Pins whose functionality isn't needed may be left unconnected.

Its physical connector is virtually identical to MIPI-10, except that it's twice as long, supporting twice as many pins. Its pinout is shown in Table 19. The function of each pin is described in Table 18.

Table 18. JTAG Connector Pin Functions

Essential	GND	Connected to ground.
	TCK	JTAG TCK signal, driven by the debug adapter.
	TDI	JTAG TDI signal, driven by the debug adapter.
	TDO	JTAG TDO signal, driven by the target.
	TMS	JTAG TMS signal, driven by the debug adapter.
	VREF DEBUG	Reference voltage for logic high.
Recommended	nRESET	Open drain active low reset signal, usually driven by the debug adapter. The signal may be used bi-directional to drive or sense the target reset signal. Asserting reset should reset any RISC-V cores as well as any other peripherals on the PCB. It should not reset the debug logic. This pin is optional but strongly encouraged. nRESET should never be connected to the TAP reset, otherwise the debugger might not be able to debug through a reset to discover the cause of a crash or to maintain execution control after the reset.
	KEY	This pin may be cut on the male and plugged on the female header to ensure the header is always plugged in correctly. It is, however, recommended to use this pin as an additional ground, to allow for fastest TCK speeds. A shrouded connector should be used to prevent the cable from being plugged in incorrectly.
Advanced	EXT	Reserved for custom use. Could be an input or an output.
	TRIGIN	Not used by this specification, to be driven by debug adapter. (Can be used for extended functions like UART or boot mode selection by some debug adapters).
	TRIGOUT	Not used by this specification, driven by the target.
Specialized	nTRST	Test reset, driven by the debug adapter. Asserting nTRST initializes the JTAG DTM asynchronously. It is used in systems where the JTAG DTM is not ready to be used after a normal power up. This signal is sometimes called TRST*.
Legacy	RTCK	Return test clock, driven by the target. A target may relay the TCK signal here once it has processed it, allowing a debugger to adjust its TCK frequency in response. This signal should only be used to support legacy components that rely on this functionality.
	nTRST_PD	Test reset pull-down, driven by the debug adapter. Same function as nTRST, but with pull-down resistor on target. This signal should only be used to support legacy components that rely on this functionality.

Table 19. MIPI 20-pin JTAG Connector Diagram

VREF DEBUG	1	2	TMS
GND	3	4	TCK
GND	5	6	TDO
GND or KEY	7	8	TDI

GND	9	10	nRESET
GND	11	12	GND or RTCK
GND	13	14	NC or nTRST_PD
GND	15	16	nTRST or NC
GND	17	18	TRIGIN or NC
GND	19	20	TRIGOUT or GND

6.1.8. cJTAG

This spec does not have specific recommendations on how to use the cJTAG protocol.

When implementing cJTAG access to a JTAG DTM, the MIPI 10-pin Narrow JTAG connector should be used. Pins whose functionality isn't needed may be left unconnected.

Viewing the male header from above (the pins pointing at your eye), a target's connector looks as it does in Table 20.

Table 20. MIPI 10-pin Narrow JTAG Connector Diagram

VREF DEBUG	1	2	TMSC
GND	3	4	TCKC
GND	5	6	EXT or NC
GND or KEY	7	8	NC or nTRST_PD
GND	9	10	nRESET

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.

Memory is accessed using the Abstract Access Memory command or through System Bus Access.

This implementation could allow a debugger to collect information from the hart even when that hart is unable to execute instructions.

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. It 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 harts. This interrupt causes each hart to enter Debug Mode and jump to a defined memory region that is serviced by the DM and is only accessible to the harts in Debug Mode. Accesses to this memory should be uncached to avoid side effects from debugging operations. When taking this jump, `pc` is saved to dpc and cause is updated in dcsr. This jump is similar to a trap but it is not architecturally considered a trap, so for instance doesn't count as a trap for trigger behavior.

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 DM 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, the DM populates these instructions as `nop`'s. If postexec is set, execution continues to the debugger-controlled Program Buffer, otherwise the DM causes an `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 an address within the Debug Module. The code there causes the hart to write to the Debug Module indicating an 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. Typically the hart will execute a `fence` instruction before entering the park loop, to ensure that any effects from the abstract command, such as a write to data0, take effect before the DM returns busy to 0.

To resume execution, the debug module sets a flag which causes the hart to execute a `dret`. `dret` is an instruction that only has meaning while in Debug Mode and not executing from the Program Buffer. Its recommended encoding is 0x7b200073. When `dret` is executed, is restored from dpc and normal execution resumes at the privilege set by prv and v.

data0 etc. are mapped into regular memory at an address relative to 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.

The PMP must not disallow fetches, loads, or stores in the address range associated with the Debug Module when the hart is in Debug Mode, regardless of how the PMP is configured. The same is true of PMA. Without this guarantee, the park loop would enter an infinite loop of traps and debug would not be possible.

A.3. Debug Module Interface Signals

As stated in section Section 3.1 the details of the DMI are left to the system designer. It is quite often the case that only one DTM and one DM is implemented. In this case it might be useful to comply with the signals suggested in Table 21, which is the implementation used in the open-source rocket-chip RISC-V core.

The DTM can start a request when the DM sets REQ_READY to 1. When this is the case REQ_OP can be set to 1 for a read or 2 for a write request. The desired address is driven with the REQ_ADDRESS signal. Finally REQ_VALID is set high, indicating to the DM that a valid request is pending.

The DM must respond to a request from the DTM when RSP_READY is high. The status of the response is indicated by the RSP_OP signal (see op). The data of the response is driven to RSP_DATA. A pending response is signalled by setting RSP_VALID.

Table 21. Signals for the suggested DMI between one DTM and one DM

Signal	Width	Source	Description
REQ_VALID	1	DTM	Indicates that a valid request is pending
REQ_READY	1	DM	Indicates that the DM is able to process a request
REQ_ADDRESS	abits	DTM	Requested address
REQ_DATA	32	DTM	Requested data
REQ_OP	2	DTM	Same meaning as the op field
RSP_VALID	1	DM	Indicates that a valid respond is pending
RSP_READY	1	DTM	Indicates that the DTM is able to process a respond
RSP_DATA	32	DM	Response data
RSP_OP	2	DM	Same meaning as the op field

Appendix B: Debugger Implementation

B.1. C Header File

github.com/riscv/riscv-debug-spec contains instructions for generating a C header file that defines macros for every field in every register/abstract command mentioned in this document.

B.2. External 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 Section 6.1. 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.2.1. 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 Update-DR and Capture-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.2.2. Checking for Halted Harts

A user will want to know as quickly as possible when a hart is halted (e.g. due to a breakpoint). To efficiently determine which harts are halted when there are many harts, the debugger 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.2.3. Halting

To halt one or more harts, the debugger selects them, sets haltreq, and then waits for allhalted to indicate the harts are halted. Then it can clear haltreq to 0, or leave it high to catch a hart that resets while halted.

B.2.4. 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 selected harts have resumed. 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.2.5. Single Step

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

B.2.6. Accessing Registers

B.2.6.1. Using Abstract Command

Read `s0` using abstract command:

Op	Address	Value	Comment
Write	command	aarsize = 2, transfer, regno = 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	aarsize = 2, transfer, write, regno = 0x300	Write `mstatus`

B.2.6.2. 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	aarsize = 2, postexec, transfer, write, regno = 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, regno = 0x1008	read `s0`
Read	data0	-	Returns the value that was in `f1`

B.2.7. Reading Memory

B.2.7.1. Using System Bus Access

With system bus access, addresses are physical system bus addresses.

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.2.7.2. Using Program Buffer

Through the Program Buffer, the hart performs the memory accesses. Addresses are physical or virtual (depending on and other system configuration).

Read a word from memory using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`lw s0, 0(s0)`	
Write	`progbuf1`	`ebreak`	

Op	Address	Value	Comment
Write	data0	address	
Write	command	transfer, write, postexec, regno = 0x1008	Write `s0`, then execute program buffer
Write	command	regno = 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, s1, 4`	
Write	`progbuf2`	`ebreak`	
Write	data0	address	
Write	command	transfer, write, postexec, regno = 0x1008	Write `s0`, then execute program buffer
Write	command	postexec, regno = 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.

B.2.7.3. Using Abstract Memory Access

Abstract memory accesses act as if they are performed by the hart, although the actual implementation may differ.

Read a word from memory using abstract memory access:

Op	Address	Value	Comment
Write	`data1`	address	
Write	command	cmdtype=2, aamsize = 2	
Read	data0	-	Value read from memory

Read block of memory using abstract memory access:

Op	Address	Value	Comment
Write	abstractauto	1	Re-execute the command when data0 is accessed
Write	`data1`	address	
Write	command	cmdtype=2, aamsize = 2, aampostincrement = 1	
Read	data0	-	Read value, and trigger reading of next address
...	...	...	...
Write	abstractauto	0	Disable auto-exec
Read	data0	-	Get last value read from memory.

B.2.8. Writing Memory

B.2.8.1. Using System Bus Access

With system bus access, addresses are physical system bus addresses.

Write a word to memory using system bus access:

Op	Address	Value	Comment
Write	sbcs	sbaccess = 2	Configure access size
Write	sbaddress0	address	
Write	sbdata0	value	

Write a 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.2.8.2. Using Program Buffer

Through the Program Buffer, the hart performs the memory accesses. Addresses are physical or virtual (depending on and other system configuration).

Write a word to memory using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`sw s1, 0(s0)`	
Write	`progbuf1`	`ebreak`	
Write	data0	address	
Write	command	transfer, write, regno = 0x1008	Write `s0`
Write	data0	value	
Write	command	transfer, write, postexec, regno = 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, s1, 4`	
Write	`progbuf2`	`ebreak`	
Write	data0	address	
Write	command	transfer, write, regno = 0x1008	Write `s0`
Write	data0	value0	
Write	command	transfer, write, postexec, regno = 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.2.8.3. Using Abstract Memory Access

Abstract memory accesses act as if they are performed by the hart, although the actual implementation may differ.

Write a word to memory using abstract memory access:

Op	Address	Value	Comment
Write	`data1`	address	
Write	data0	value	
Write	command	cmdtype=2, aamsize=2, write=1	

Write a block of memory using abstract memory access:

Op	Address	Value	Comment
Write	data1	address	
Write	data0	value0	
Write	command	cmdtype=2, aamsize = 2, write = 1, aampostincrement = 1	
Write	abstractauto	1	Re-execute the command when data0 is accessed
Write	data0	value1	
Write	data0	value2	
…	…	…	…
Write	data0	valueN	
Write	abstractauto	0	Disable auto-exec

B.2.9. 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 might 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. All examples assume XLEN=32.

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

tdata1	0x6980105c	type=6, dmode=1, action=1, select=0, match=0, m=1, s=1, u=1, vs=1, vu=1, execute=1
tdata2	0x80001234	address

Enter Debug Mode when performing a load at address 0x80007f80 in M-mode or S-mode or U-mode:

tdata1	0x68001059	type=6, dmode=1, action=1, select=0, match=0, m=1, s=1, u=1, load=1
tdata2	0x80007f80	address

Enter Debug Mode when storing to an address between 0x80007c80 and 0x80007cef (inclusive) in VS-mode or VU-mode when hgatp.VMID=1:

tdata1 0	0x69801902	type=6, dmode=1, action=1, chain=1, select=0, match=2, vs=1, vu=1, store=1
tdata2 0	0x80007c80	start address (inclusive)
textra32 0	0x03000000	mhselect=6, mhvalue=0
tdata1 1	0x69801182	type=6, dmode=1, action=1, select=0, match=3, vs=1, vu=1, store=1
tdata2 1	0x80007cf0	end address (exclusive)

textra32 1	0x03000000	mhselect=6, mhvalue=0

Enter Debug Mode when storing to an address between 0x81230000 and 0x8123ffff (inclusive):

tdata1	0x698010da	type=6, dmode=1, action=1, select=0, match=1, m=1, s=1, u=1, vs=1, vu=1, store=1
tdata2	0x81237fff	16 upper bits to match exactly, then 0, then all ones.

Enter Debug Mode when loading from an address between 0x86753090 and 0x8675309f or between 0x96753090 and 0x9675309f (inclusive):

tdata1 0	0x69801a59	type=6, dmode=1, action=1, chain=1, match=4, m=1, s=1, u=1, vs=1, vu=1, load=1
tdata2 0	0xfff03090	Mask for low half, then match for low half
tdata1 1	0x698012d9	type=6, dmode=1, action=1, match=5, m=1, s=1, u=1, vs=1, vu=1, load=1
tdata2 1	0xefff8675	Mask for high half, then match for high half

B.2.10. Handling Exceptions

Generally the debugger can avoid exceptions by being careful with the programs it writes. Sometimes they are unavoidable though, e.g. if the user asks to access memory or a CSR that is not implemented. A typical debugger will not know enough about the hardware 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, command 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.2.11. 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	

Op	Address	Value	Comment
Write	`data1`	data	
Write	command	0x10000000	Perform quick access

This shows an example of setting the m bit in 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

B.3. Native Debugger Implementation

The spec contains a few features to aid in writing a native debugger. This section describes how some common tasks might be achieved.

B.3.1. Single Step

Single step is straightforward if the OS or a debug stub runs in M-Mode while the program being debugged runs in a less privileged mode. When a step is required, the OS or debug stub writes count=1, action=0, m=0 before returning control to the lower user program with an `mret` instruction.

Stepping code running in the same privilege mode as the debugger is more complicated, depending on what other debug features are implemented.

If hardware implements mpte and mte, then stepping through non-trap code which doesn't allow for nested interrupts is also straightforward.

If hardware automatically prevents action=0 triggers from matching when entering a trap handler as described in Section 5.4, then a carefully written trap handler can ensure that interrupts are disabled whenever the icount trigger must not match.

If neither of these features exist, then single step is doable, but tricky to get right. To single step, the debug stub would execute something like:

```
li    t0, count=4, action=0, m=1
csrw  tdata1, t0    /* Write the trigger. */
lw    t0, 8(sp)     /* Restore t0, count decrements to 3 */
lw    sp, 0(sp)     /* Restore sp, count decrements to 2 */
mret                /* Return to program being debugged. count decrements to 1 */
```

There is an additional problem with using icount to single step. An instruction may cause an exception into a more privileged mode where the trigger is not enabled. The exception handler might address the cause of the exception, and then restart the instruction. Examples of this include page faults, FPU instructions when the FPU is not yet enabled, and interrupts. When a user is single stepping through such code, they will have to step twice to get past the restarted instruction. The first time the exception handler runs, and the second time the instruction actually executes. That is confusing and usually undesirable.

To help users out, debuggers should detect when a single step restarted an instruction, and then step again. This way the users see the expected behavior of stepping over the instruction. Ideally the debugger would notify the user that an exception handler executed the first time.

The debugger should perform this extra step when the PC doesn't change during a regular step.

It is safe to perform an extra step when the PC changes, because every RISC-V instruction either changes the PC or has side effects when repeated, but never both.

To avoid an infinite loop if the exception handler does not address the cause of the exception, the debugger must execute no more than a single extra step.

Index