RISC-V External Debug Support
Version 0.11nov12

Tim Newsome <tim@sifive.com>

November 12, 2016

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

Contents

1 Introduction **5**

2 About This Document **5**
2.1 Structure 5
2.2 Terminology 5
2.3 Register Definitions 5
2.3.1 Long Name (`shortname`, at 0x123) 6

3 Background **6**

4 Supported Features **6**

5 System Overview **7**

6 Debug Transport Module (DTM) **9**

7 Debug Module(DM) **9**
7.1 Debug Bus 10
7.2 System Bus 11
7.3 Debug Interrupt Block 11
7.4 Halt Notification Block 11
7.5 Debug ROM 12
7.6 Debug RAM 12
7.7 Reset Control 13
7.8 Bus Access 13
7.9 Serial Ports 13
7.10 Security 14

7.11 Debug Module Debug Bus Registers . . . 14
7.11.1 Control (dmcontrol, at 0x10) . . . 14
7.11.2 Info (dminfo, at 0x11) . . . 16
7.11.3 Authentication Data (authdata0, at 0x12) . . . 18
7.11.4 Authentication Data (authdata1, at 0x13) . . . 18
7.11.5 Serial Data (serdata, at 0x14) . . . 19
7.11.6 Serial Status (serstatus, at 0x15) . . . 19
7.11.7 System Bus Address 31:0 (sbaddress0, at 0x16) . . . 20
7.11.8 System Bus Address 63:32 (sbaddress1, at 0x17) . . . 20
7.11.9 System Bus Data 31:0 (sbdata0, at 0x18) . . . 20
7.11.10 System Bus Data 63:32 (sbdata1, at 0x19) . . . 21
7.11.11 Halt Notification Summary (haltsum, at 0x1b) . . . 21
7.11.12 System Bus Address 95:64 (sbaddress2, at 0x3d) . . . 22
7.11.13 System Bus Data 95:64 (sbdata2, at 0x3e) . . . 22
7.11.14 System Bus Data 127:96 (sbdata3, at 0x3f) . . . 23
7.12 Debug Module System Bus Registers . . . 23
7.12.1 Clear Debug Interrupt (cleardebint, at 0x100) . . . 23
7.12.2 Set Halt Notification (sethaltnot, at 0x10c) . . . 25
7.12.3 Serial Info (serinfo, at 0x110) . . . 25
7.12.4 Serial Send 0 (sersend0, at 0x200) . . . 25
7.12.5 Serial Receive 0 (serrecv0, at 0x204) . . . 25
7.12.6 Serial Status 0 (serstat0, at 0x208) . . . 26

8 RISC-V Debug **26**
8.1 Hart IDs . . . 26
8.2 Debug Mode . . . 26
8.3 Debug ROM Contents . . . 27
8.4 dret Instruction . . . 27
8.5 Load-Reserved/Store-Conditional Instructions . . . 28
8.6 Core Debug Registers . . . 28
8.6.1 Debug Control and Status (dcsr, at 0x7b0) . . . 28
8.6.2 Debug PC (dpc, at 0x7b1) . . . 30
8.6.3 Debug Scratch Register (dscratch, at 0x7b2) . . . 30
8.6.4 Privilege Level (priv, at virtual) . . . 30

9 Trigger Module **31**
9.1 Trigger Registers . . . 31
9.1.1 Trigger Select (tselect, at 0x7a0) . . . 31
9.1.2 Trigger Data 1 (tdata1, at 0x7a1) . . . 32
9.1.3 Trigger Data 2 (tdata2, at 0x7a2) . . . 33
9.1.4 Trigger Data 3 (tdata3, at 0x7a3) . . . 33
9.1.5 Match Control (mcontrol, at 0x7a1) . . . 33
9.1.6 Instruction Count (icount, at 0x7a1) . . . 35

10 JTAG Debug Transport Module **37**
10.1 Background . . . 37
10.2 JTAG Connector . . . 38
10.3 JTAG Registers . . . 38
10.3.1 IDCODE (00001) . . . 38
10.3.2 DTM Control (`dtmcontrol`, at 10000) . . . 41
10.3.3 Debug Bus Access (`dbus`, at 10001) . . . 42
10.3.4 BYPASS (11111) . . . 44

A Debugger Implementation **45**
A.1 Debug Bus Access . . . 45
A.2 Debug RAM . . . 45
A.3 Main Loop . . . 45
A.4 Reading Memory . . . 46
A.5 Writing Memory . . . 47
A.6 Halt . . . 47
A.7 Reading Registers . . . 49
A.8 Writing Registers . . . 49
A.9 Running . . . 49
A.10 Single Step . . . 49
A.11 Handling Exceptions . . . 50

B Debug ROM Source **50**

C Trace Module **52**
C.1 Trace Data Format . . . 53
C.2 Trace Events . . . 53
C.3 Synchronization . . . 53
C.4 Trace Registers . . . 56
C.4.1 Trace (`trace`, at 0x728) . . . 56
C.4.2 Trace Buffer Start (`tbufstart`, at 0x729) . . . 58
C.4.3 Trace Buffer End (`tbufend`, at 0x72a) . . . 58
C.4.4 Trace Buffer Write (`tbufwrite`, at 0x72b) . . . 58

D Future Ideas **58**
D.1 Lightweight Brainstorming . . . 59

E Change Log **60**

List of Figures

1 RISC-V Debug System Overview . . . 8

List of Tables

1 Register Access Abbreviations 6
2 Debug Module Debug Bus Space 10
3 Debug Module System Bus Space 11
4 Debug Module Debug Bus Registers 14
5 Debug Module System Bus Registers 24
6 Debug ROM Contents 27
7 Core Debug Registers 28
8 Privilege Level Encoding 30
9 Trigger Registers 31
10 JTAG Connector Diagram 38
11 JTAG Connector Pinout 39
12 JTAG TAP Registers 40
13 Memory Read Timeline 48
14 Trace Sequence Header Packets 54
15 Trace Data Events 55
16 Trace Registers 56

Acknowledgments

I would like to thank the following people for their time, feedback, and ideas: Bruce Ableidinger, Krste Asanovic, Mark Beal, Monte Dalrymple, Peter Egold, Gajinder Panesar, Klaus Kruse Pedersen, Antony Pavlov, Ken Pettit, Megan Wachs, Stefan Wallentowitz, Ray Van De Walker, Andrew Waterman, and Andy Wright.

1 Introduction

Software contains bugs, and to help find these bugs it is critical to have good debugging tools. Unless a robust OS is running on a core, with convenient access to it (eg. over a network interface), hardware support is required to provide visibility into what is going on in that core. This document outlines how that support should be provided on RISC-V platforms.

2 About This Document

2.1 Structure

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

2.2 Terminology

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

2.3 Register Definitions

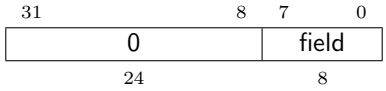
All register definitions in this document follow the format shown in Section 2.3.1. A simple graphic shows which fields are in the register. The upper and lower bit indices are shown to the top left and top right of each field. The total number of bits in the field are shown below it.

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.

Table 1: Register Access Abbreviations

R	Read-only.
R/W	Read/Write.
R/W0	Read/Write. Only writing 0 has an effect.
R/W1	Read/Write. Only writing 1 has an effect.
W	Write-only. When read this field returns 0.
W1	Write-only. Only writing 1 has an effect.

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



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

3 Background

There are two forms of external debugging. The first is *halt mode* debugging, where an external debugger will halt some or all components of a platform and inspect them while they are in stasis. Then the debugger can allow the hardware to either perform a single step or to run freely.

The second is *run mode* debugging. In this mode there is a software debug agent running on a component (eg. triggered by a timer interrupt on a RISC-V core) which communicates with a debugger without halting the component. This is essential if the component is controlling some real-time system (like a hard drive) where halting the component might lead to physical damage. It requires more software support (both on the chip as well as on the debug client). For this use case the debug interface may include simple serial ports.

A third use for the external debug interface is to use it as a general transport for a component to communicate with the outside world. For instance, it could be used to implement a serial interface that firmware could use to provide a simple CLI. This can use the same serial ports used for run-mode debugging.

4 Supported Features

The debug interface described out here supports the following features:

1. RV32, RV64, and future RV128 are all supported.
2. Any hart in the platform can be independently debugged.
3. Harts can be asked to run a short custom program and immediately return to regular execution afterwards, enabling relatively unintrusive inspection of state.

4. Optionally, a bus master can be implemented to allow memory access without involving any hart.
5. Debugging can be supported over multiple transports.
6. Code can be downloaded efficiently.
7. Each hart can be debugged from the very first instruction executed.
8. A RISC-V core can be halted when a software breakpoint instruction is executed.
9. Hardware can step over any instruction.
10. A RISC-V core can be halted when a trigger matches the PC, read/write address/data, or an instruction opcode.
11. The debug module may implement serial ports which can be used for communication between debugger and monitor, or as a general protocol between debugger and application.
12. Arbitrary instructions can be executed on a halted hart. That means no new debug functionality is needed when a core has custom instructions or registers, as long as there exist programs that can store those registers to memory.
13. The debugger doesn't need to know anything about the microarchitecture of the cores it is debugging.
14. Minimizes the additional datapath needed in the core to implement debug functionality.
15. Don't need to route a special debug bus to each core.
16. Cores don't have to become bus slaves.

5 System Overview

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

The user interacts with the Debug Host, which is running a debugger. The debugger communicates with a Debug Translator (which may include a hardware driver) to communicate with Debug Transport Hardware that's connected to the host. The Debug Transport Hardware connects the Debug Host to the Platform's Debug Transport Module (DTM). The DTM provides access to the Debug Module (DM) which contains much of the debug functionality. This interface is called the Debug Bus.

The DM allows the debugger to interrupt any hart in the platform. When a running RISC-V core is interrupted, it enters Debug Mode and jumps to the

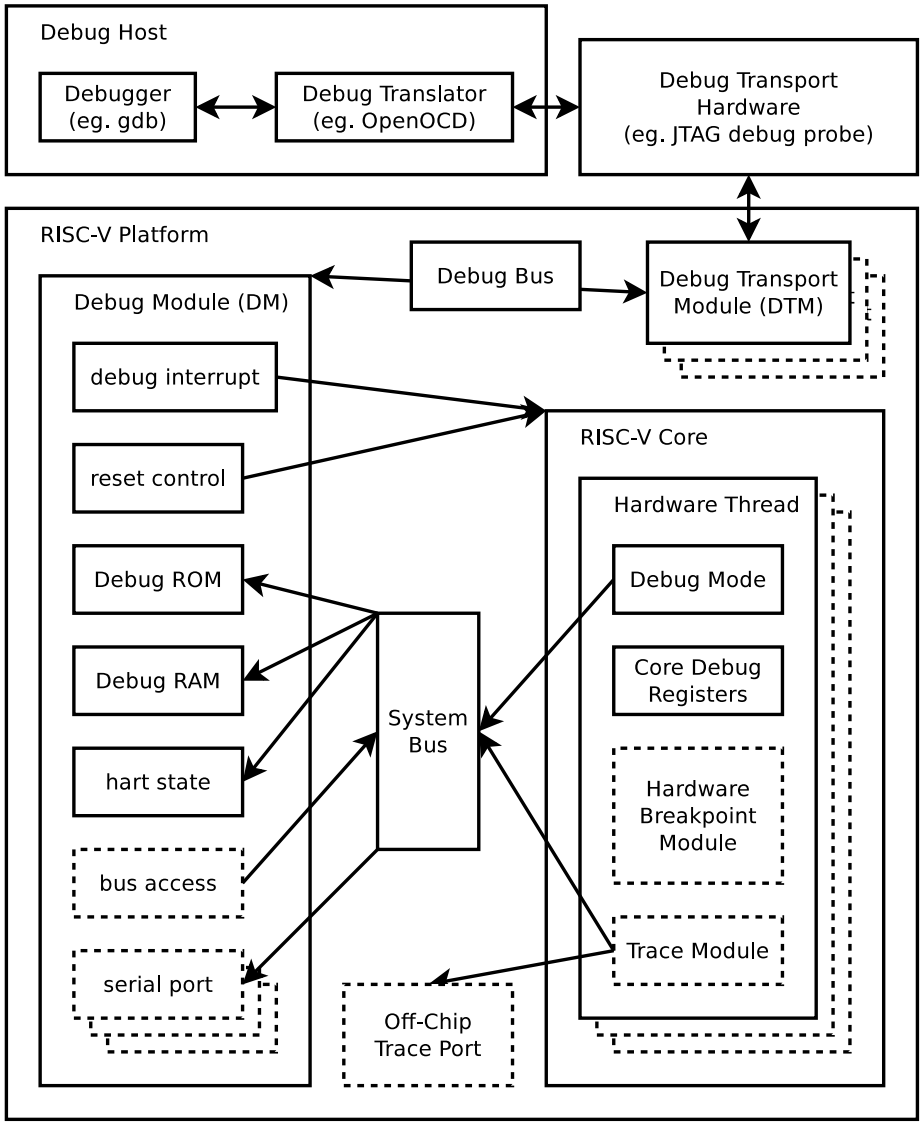


Figure 1: RISC-V Debug System Overview

Debug ROM. Generally this will cause it to execute code which has been written to the Debug RAM. The debugger can use this mechanism to access memory and registers at the cost of briefly halting one of the cores, or it can halt the hart and leave it halted.

Each RISC-V core may implement up to one Trigger Module for each hart. These can detect breakpoints as a hart is executing, which causes a hart to halt spontaneously. When that happens it notifies the DM by communicating over the System Bus. The DM tracks the hart status, and makes it available in its Debug Bus accessible registers so the debugger knows action is required. The debugger can then write appropriate instructions to Debug RAM, and send the hart another interrupt to indicate it should now jump to Debug RAM. This process is repeated until the debugger writes the code that causes the hart to leave Debug Mode.

Almost all this logic can be encoded in the Debug ROM. When a hart enters Debug Mode all it does is change the privilege mode it's running in, and jump to the beginning of Debug ROM. Code in the Debug ROM determines the cause and then either jumps straight to Debug RAM or notifies the debugger and then waits for further input from the debugger.

6 Debug Transport Module (DTM)

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

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

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

7 Debug Module(DM)

The Debug Module contains the shared functionality required to debug a RISC-V hart. It is accessed over two distinct buses that provide access to distinct register sets. DTMs communicate with the DM using the Debug Bus. Harts communicate with the DM using the System Bus. In addition, the DM drives independent reset and interrupt signals to every hart.

A single DM can debug up to 1024 harts. If more harts need to be accessible the best solution is to extend the specification.

7.1 Debug Bus

The Debug Bus uses between 5 and 32 address bits, and 34 data bits. It supports both read and write operations. The bottom of the address space is used for the DM. Extra space can be used for custom debug devices, other cores, etc. The space is laid out so that small to medium platforms can get away with just 5 address bits, while DTMs used on large systems will want to use 6 bits. 7 bits are only required for systems that want to use a large Debug RAM or have other debug modules. Details of this bus are implementation-specific.

The registers are 34 bits wide so that simple DTMs can access everything that is essential for efficient operation in a single access. This keeps the DTM simple, at the cost of a slightly awkward bus width. 34 bits are essential for an efficient bus master interface. The serial interface could work with 33 bits by jumping through some hoops involving special data *values. The Debug RAM interface could actually work fine with just 33 bits, but it's worth to having extra bit just to make the (hopefully common) bus master extension great.*

How this bus is implemented is completely left up to the designer, but in larger systems it makes sense to use a standard multi-master bus. TODO: Recommend a specific bus.

Table 2 shows the layout of the Debug Bus address space.

Table 2: Debug Module Debug Bus Space

Address	Description
0x00 – 0x0f	64 bytes of R/W Debug RAM access. Each unique address accesses 32 bits of the debug RAM, so 0x0 contains the first word, 0x1 the second word, and so on. Bit 32 of every register provides R/W0 access to the halt notification of the hart selected by hartid in `dmcontrol`. Bit 33 of every register provides R/W1 access to the debug interrupt of the hart selected by hartid in `dmcontrol`.
0x10 – 0x1b	Debug Module registers described in Section 7.11.
0x1c – 0x3b	Halt Notification Status registers. Bit 0 of the first register contains halt notification 0, while bit 31 of the last register contains halt notification 1023 (if there are that many harts attached).
0x3c – 0x3f	Debug Module registers described in Section 7.11.
0x40 – 0x6f	192 more bytes of R/W Debug RAM access for words 0x10 – 0x3f of the Debug RAM. Only implemented if Debug RAM is larger than 64 bytes. This spec doesn't present any compelling reason to implement that much Debug RAM.

Some registers might not be present, either because the feature they support doesn't exist or because there's simply empty space in the register map. Those

registers always read as 0, and writes to them are ignored.

7.2 System Bus

Harts being debugged communicate with the DM over the System Bus. The Debug Module addresses are fixed across all platforms to reduce the amount of customization required. Table 3 shows which address ranges are handled by the DM.

The debug registers and RAM are placed below 0x800 so that it is possible to access the debug space relative to `x0`*, which allows the Debug ROM code to only save a single register upon entry.*

Table 3: Debug Module System Bus Space

Address	Description
0x0 – 0xff	Reserved for custom use.
0x100 – 0x2ff	Debug Module registers described in Section 7.12.
0x400 – 0x4ff	Up to 256 bytes of Debug RAM. Each unique address specifies 8 bits.
0x800 – 0x9ff	Up to 512 bytes of Debug ROM.

This reliance on the System Bus means that debugging is not possible if the System Bus is hung for some reason. If this is a concern, then it's possible to implement a separate bus that allows each hart to access the registers listed in Table 3.

7.3 Debug Interrupt Block

This block controls interrupts from the Debug Module to a hart. It is used to halt a currently running hart, or to signal that the hart should take an action when it is already halted.

For each hart the block contains a single bit that gets set when 1 is written to `interrupt` in `dmcontrol` or Debug RAM registers. The bit gets cleared when its hart id is written to `cleardebint`. It is unspecified what happens if the bit is set and cleared at the same time.

The Debug Module conceptually has a direct connection to the debug interrupt signal of every hart that has one. Each hart must receive a signal change in no longer than 1 second. (How this is implemented is not further specified. A few clock cycles will be more typical.)

7.4 Halt Notification Block

This block tracks halt notifications from a hart to the Debug Module. It is used by harts to inform the Debug Module that they halted for a reason other than the Debug Interrupt being asserted.

For each hart, the block contains a single bit that gets set when the corresponding number is written to `sethaltnot`. The bit can be cleared using haltnot in `dmcontrol` and Debug RAM registers. It is read over the Debug Bus using `haltsum`, the halt notification section in the Debug Bus address space, and haltnot in `dmcontrol` and Debug RAM registers. It is unspecified what happens if the bit is set and cleared at the same time.

It's expected that a hart will write its hart ID to `sethaltnot` when it halts spontaneously. (Debug ROM code takes care of this.) Any other writes will likely confuse the debugger and should be avoided.

7.5 Debug ROM

The Debug ROM contains code for a RISC-V hart to execute when it enters Debug Mode. This ROM is inside the Debug Module so that it can be shared among all RISC-V harts in the system, and reduces the number of changes required to a RISC-V core to support debugging.

It is described in detail in Section 8.3.

7.6 Debug RAM

The Debug RAM is used by the debugger and the Debug ROM code to execute arbitrary instructions, and to hold data. Debug RAM must be at least 28 bytes to accommodate 32-bit RISC-V cores, 40 bytes to accommodate 64-bit RISC-V cores, and 64 bytes to accommodate 128-bit RISC-V cores.

Debug RAM is accessible over both the Debug Bus and the System Bus. When it is accessed by both simultaneously, reads may return undefined data, while writes may be ignored. If an access over the Debug Bus fails in this way, the result must be 2 (fail).

The minimum Debug RAM size is determined by the smallest debug program that can write an arbitrary value to an arbitrary location in RAM. That program consists of 4 instructions, followed by 3 XLEN-bit values (address, data, and scratch). Code for this program is shown in Section A.5. (When the compressed ISA is supported it would be possible to cut 8 bytes from this requirement, but a debugger may assume that it doesn't need to use the compressed ISA.)

Since Debug RAM resides on the System Bus, it's possible for any component to write to it at any time. Unexpected writes should only happen when a component malfunctions, but if it does happen it will definitely interfere with debugging. At the cost of more hardware, this can be resolved in two ways. If the bus knows an ID for the originator, then the Debug Module can refuse write accesses to originators that don't match the hart ID set in hartid of `dmcontrol`. If that's not feasible, a more expensive option is to include a separate Debug RAM per hart, which is only accessible from that hart. To achieve this you would likely need a separate bus that gives the DM full access and the harts

access to just their Debug RAM. hartid controls which Debug RAM the DM accesses. This is an expensive solution, but still a valid implementation of this spec.

7.7 Reset Control

This block is connected to global reset signals. The first signal resets every component in the platform. The second signal is optional and resets the non-debug portion of every component in the platform. Both resets exclude any DTMs and the Debug Module itself.

7.8 Bus Access

In a minimal configuration a debugger can access the system bus by having a RISC-V hart perform the accesses it requires. Optionally a Bus Access block may be implemented. Because the Bus Access block performs accesses directly from the DM, it only uses physical addresses.

Implementing a Bus Access block has several benefits. First, it is now possible to inspect a running system with minimal impact. The only impact now is that the bus is busy while the debugger is performing an access. Second, it may improve performance when downloading programs. There is only a benefit if JTAG TCK is a significant fraction of the RISC-V hart's clock speed. Third, it may provide access to devices that a hart does not have access to. A hart may be unable to access all devices in a system (eg. for security reasons) and in this case the debugger needs another path to access them.

To keep implementing, configuring, and using a debugger as simple as possible, systems should use the same memory map for each hart. That means that a given address maps to the same device no matter which hart performs the access. (Different harts may not all have permission to access the same devices.) If different harts do have unique memory maps then the system should provide access to all devices using the Bus Access block. This will make implementing, configuring, and using a debugger more complex so should be avoided if possible.

7.9 Serial Ports

The Debug Module may implement up to 8 serial ports. They support basic flow control and full duplex data transfer between a component and the debugger. They can be used to communicate with a debug monitor running on a hart, for the equivalent of printf debugging, to provide a simple CLI without requiring any extra peripherals, or more generally to emulate devices that aren't present. All these uses require software support, and are not further specified here.

7.10 Security

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

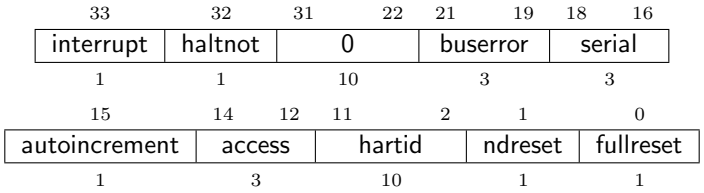
Another option is to allow the DM to be unlocked only by users who have an access key. A simple mechanism is documented in Section 7.11. When authenticated is clear, the DM must not interact with the rest of the platform in any way.

7.11 Debug Module Debug Bus Registers

Table 4: Debug Module Debug Bus Registers

Address	Name
0x10	Control
0x11	Info
0x12	Authentication Data
0x13	Authentication Data
0x14	Serial Data
0x15	Serial Status
0x16	System Bus Address 31:0
0x17	System Bus Address 63:32
0x18	System Bus Data 31:0
0x19	System Bus Data 63:32
0x1b	Halt Notification Summary
0x3d	System Bus Address 95:64
0x3e	System Bus Data 95:64
0x3f	System Bus Data 127:96

7.11.1 Control (`dmcontrol`, at 0x10)

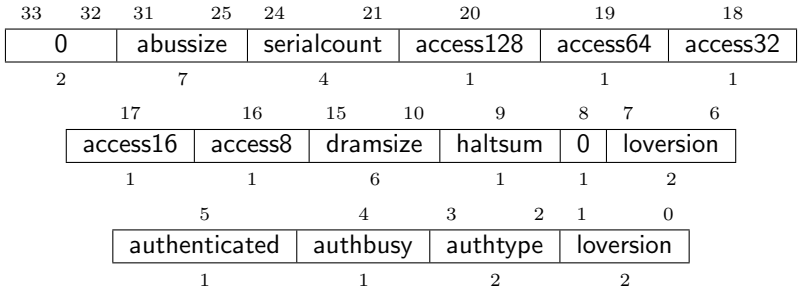


Field	Description	Access	Reset
interrupt	This field contains the Debug Interrupt bit for the hart selected by hartid. Writes apply to the new value of hartid. This field is also accessible when accessing Debug RAM.	R/W1	0
haltnot	This field contains the Halt Notification bit for the hart selected by hartid. Writes apply to the new value of hartid. This field is also accessible when accessing Debug RAM.	R/W0	0
buserror	When the debug bus master causes a bus error, this field gets set. It remains set until 0 is written to any bit in this field. Until that happens, the bus master is busy and no more accesses can be initiated. 0: There was no bus error. 1: There was a timeout. 2: A bad address was accessed. 3: There was some other error (eg. alignment).	R/W0	0
serial	Select which serial port is accessed by `serdata`.	R/W	0
autoincrement	When 1, the internal address value (used by the bus master) is incremented by the access size (in bytes) selected in access after every bus access.	R/W	0
access	Select the access size to use for system bus accesses triggered by writes to the `sbaddress` registers or `sbdata0`. 0: 8-bit 1: 16-bit 2: 32-bit 3: 64-bit 4: 128-bit If an unsupported access size is written here, the DM may not perform the access, or may perform the access with any access size	R/W	2
hartid	The ID of the hart to select. The halt notification and debug interrupt of the selected hart are accessible in haltnot and interrupt in this register as well as every Debug RAM register.	R/W	0

Continued on next page

ndreset	Every time this bit is written as 1, it triggers a full reset of the non-debug logic on the platform. This bit exists so that, for debugging purposes, reset behavior can be different from the standard behavior. For instance, a core could be forced into Debug Mode right out of reset.	W1	0
fullreset	Every time this bit is written as 1, it triggers a full reset of the platform, including every component in it and the debug logic for each component. It also resets the DM itself.	W1	0

7.11.2 Info (`dminfo`, at 0x11)



Field	Description	Access	Reset
abussize	Width of the address bus in bits. (0 indicates there is no bus access support.)	R	Preset
serialcount	Number of supported serial ports.	R	Preset
access128	1 when 128-bit bus accesses are supported.	R	Preset
access64	1 when 64-bit bus accesses are supported.	R	Preset
access32	1 when 32-bit bus accesses are supported.	R	Preset
access16	1 when 16-bit bus accesses are supported.	R	Preset
access8	1 when 8-bit bus accesses are supported.	R	Preset
dramsize	Size of the Debug RAM, in 32-bit words minus 1. Eg. if Debug RAM is 32 bytes, it's encoded here as 7 ($32/4 - 1$). A debugger must not access any Debug RAM locations that fall outside the range specified here.	R	Preset
haltsum	1 when `haltsum` is implemented.	R	Preset
loversion	Bits 3:2 of the 4-bit version field.	R	0
authenticated	0 when authentication is required before using the DM. 1 when the authentication check has passed. On components that don't implement authentication, this bit must be preset as 1.	R	Preset
authbusy	While 1, writes to `authdata0` and `authdata1` may be ignored or may result in authentication failing. Authentication mechanisms that are slow (or intentionally delayed) must set this bit when they're not ready to process another write.	R	0

Continued on next page

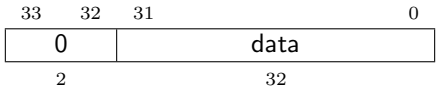
authtype	Defines the kind of authentication required to use this DM. 0: No authentication is required. 1: A password is required. 2: A challenge-response mechanism is in place. 3: Reserved for future use.	R	Preset
loversion	Bits 1:0 of the 4-bit version field. The combined version field is interpreted as follows: 0: There is no Debug Module present. 1: There is a Debug Module and it conforms to version 0.11 of this specification. Other values are reserved for future use.	R	1

7.11.3 Authentication Data (authdata0, at 0x12)

If authtype is 0, this register is not present.

If authtype is 1, writing a correct password to this register and authdata1 enables the DM. The DM is disabled either by writing an invalid password, or by resetting it. 0 must not be used as a password. If an implementation wants to use a well-known password, the recommended value is 0x5551212. Reading from the register returns 0.

If authtype is 2, things are a bit more complicated. Reading from the register pair reads the last challenge generated. Writing the correct response to authdata1 and authdata0 enables the DM. The DM is disabled either by writing an incorrect response, or by resetting it. Writing to authdata0 triggers validation, so if a 64-bit value is required then authdata1 must be written first. If the combined value in authdata0 and authdata1 is not a valid response after writing authdata0, then a new challenge must be generated. Depending on the implementation, there may not be a valid challenge until the first write to this register.



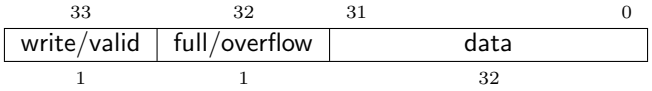
7.11.4 Authentication Data (authdata1, at 0x13)

This register contains the upper 32 bits of a 64-bit password or challenge/response as described in authdata0.

7.11.5 Serial Data (`serdata`, at 0x14)

If serialcount is 0, this register is not present.

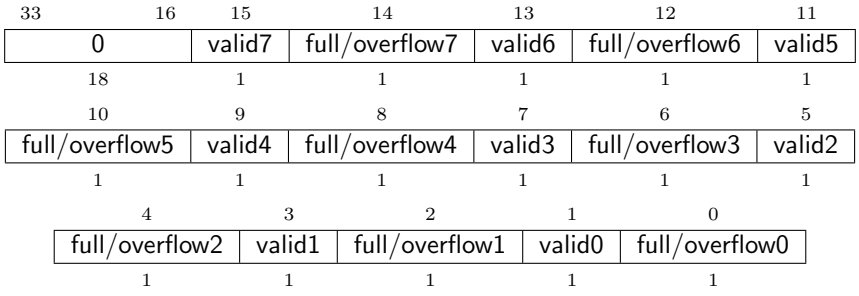
All the fields in this register apply to the serial port selected by serial in `dmcontrol`.



Field	Description	Access	Reset
write/valid	Set this bit to write data to the debugger-to-core queue. Read this bit to determine whether the register contains valid data from the core-to-debugger queue.	R/W	0
full/overflow	0: The debugger-to-core queue is not full. The next write will be accepted. 1: The debugger-to-core queue is currently full, or the debugger has previously attempted to write to the queue when it was full. To clear this state, the debugger must write 0 to this bit. (The queue may still be full, in which case the bit will remain high.)	R/W0	0
data	This field contains the oldest value in the core-to-debugger queue if write/valid reads as 1.	R/W	0

7.11.6 Serial Status (`serstatus`, at 0x15)

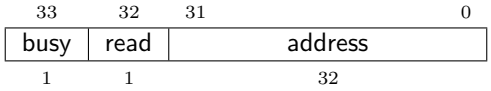
If serialcount is 0, this register is not present.



Field	Description	Access	Reset
valid0	1 when the core-to-debugger queue for serial port 0 is not empty.	R	0
full/overflow0	full/overflow for serial port 0.	R/W0	0

7.11.7 System Bus Address 31:0 (sbaddress0, at 0x16)

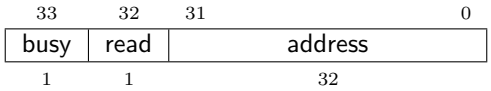
If abussize is 0, then this register is not present.



Field	Description	Access	Reset
busy	When 1, the bus master is busy and will ignore any writes to the System Bus registers. Don't write to this register without reading busy as 0 first.	R	0
read	If written as 1, the bus master will start to read after updating the address from address. The access size is controlled by access in dmcontrol.	W	0
address	Accesses the lower 32 bits of the internal address.	R/W	0

7.11.8 System Bus Address 63:32 (sbaddress1, at 0x17)

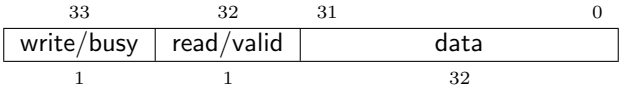
If abussize is less than 33, then this register is not present.



Field	Description	Access	Reset
busy	The same as busy in sbaddress0.	R	0
read	The same as read in sbaddress0.	W	0
address	Accesses bits 63:32 of the internal address (if the system address bus is that wide).	R/W	0

7.11.9 System Bus Data 31:0 (sbdata0, at 0x18)

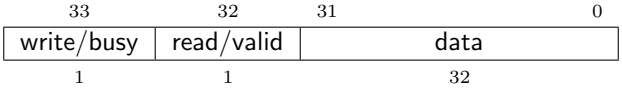
If all of the access bits in dminfo are 0, then this register is not present.



Field	Description	Access	Reset
write/busy	When 1, the bus master is busy and will ignore any writes to the System Bus registers. If written as 1, the bus master will start to write after updating the data from data. The access size is controlled by access in `dmcontrol`.	R/W	0
read/valid	When 1, the register contains the result of a successful memory read. The valid state is automatically cleared every time a new bus access is started. If written as 1, the bus master will start to read after updating the address from address. The access size is controlled by access in `dmcontrol`.	R/W	0
data	Accesses bits 31:0 of the internal data.	R/W	0

7.11.10 System Bus Data 63:32 (`sbdata1`, at 0x19)

If access64 and access128 are 0, then this register is not present.

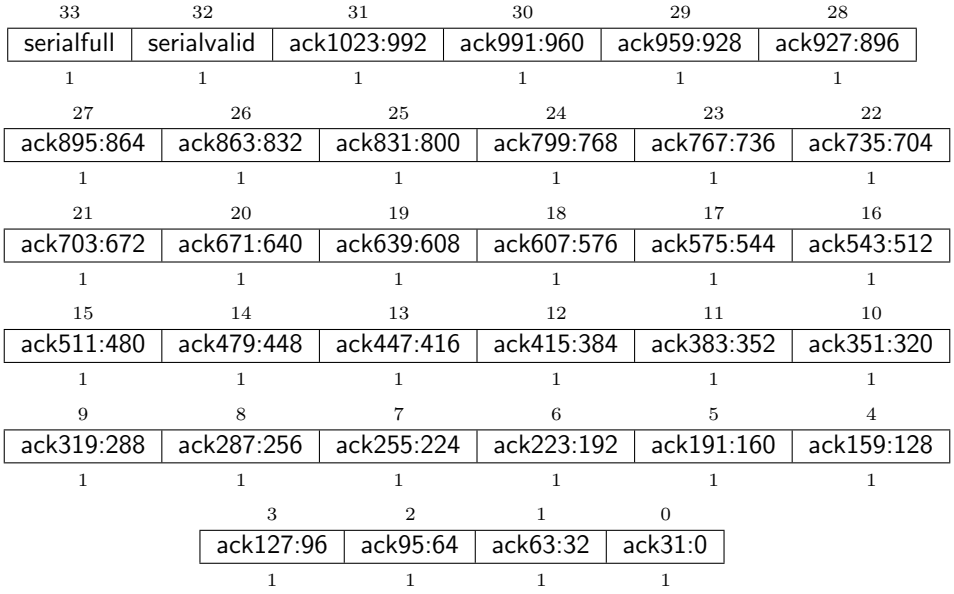


Field	Description	Access	Reset
write/busy	The same as write/busy in `sbdata0`, except if an access is triggered the access size is 64 bits instead of what access selects.	R/W	0
read/valid	The same as read/valid in `sbdata0`, except if an access is triggered the access size is 64 bits instead of what access selects.	R/W	0
data	Accesses bits 63:32 of the internal data (if the system bus is that wide).	R/W	0

7.11.11 Halt Notification Summary (`haltsum`, at 0x1b)

If implemented, this register contains a summary of which halt bits are set. This register should be implemented if there are more than 64 harts, or if there are more than 32 harts and more than 0 serial ports.

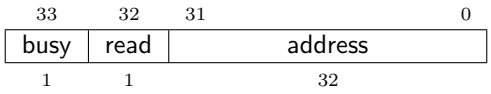
Each of the lower 32 bits contains the logical OR of 32 consecutive halt bits. When there are a large number of harts in the system, the debugger can first read this register, and then the specific registers to find the exact halt bit that's asserted.



Field	Description	Access	Reset
serialfull	Logical OR of all the full bits in `serstatus`.	R	0
serialvalid	Logical OR of all the valid bits in `serstatus`.	R	0

7.11.12 System Bus Address 95:64 (`sbaddress2`, at 0x3d)

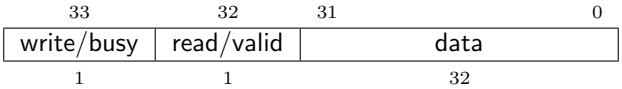
If abussize is less than 65, then this register is not present.



Field	Description	Access	Reset
busy	The same as busy in `sbaddress0`.	R	0
read	The same as read in `sbaddress0`.	W	0
address	Accesses bits 95:64 of the internal address (if the system address bus is that wide).	R/W	0

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

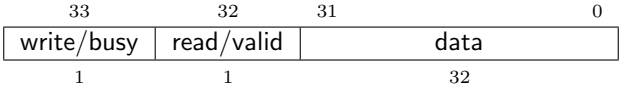
This register only exists if access128 is 1.



Field	Description	Access	Reset
write/busy	The same as write/busy in `sbdata0`, except if an access is triggered the access size is 128 bits instead of what access selects.	R/W	0
read/valid	The same as read/valid in `sbdata0`, except if an access is triggered the access size is 128 bits instead of what access selects.	R/W	0
data	Accesses bits 95:64 of the internal data (if the system bus is that wide).	R/W	0

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

This register only exists if access128 is 1.



Field	Description	Access	Reset
write/busy	The same as write/busy in `sbdata0`, except if an access is triggered the access size is 128 bits instead of what access selects.	R/W	0
read/valid	The same as read/valid in `sbdata0`, except if an access is triggered the access size is 128 bits instead of what access selects.	R/W	0
data	Accesses bits 127:96 of the internal data (if the system bus is that wide).	R/W	0

7.12 Debug Module System Bus Registers

7.12.1 Clear Debug Interrupt (`cleardebint`, at 0x100)

Writes to this register clear the debug interrupt corresponding to the number written. To avoid a race, the DM must not complete the write access on the System Bus until the change in the debug interrupt value has been propagated to the relevant hart.

A hart must write its hart ID to this register to indicate that it has completed executing a debug program. (The code to do this is already in the Debug ROM.)

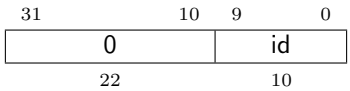


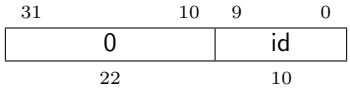
Table 5: Debug Module System Bus Registers

Address	Name
0x100	Clear Debug Interrupt
0x10c	Set Halt Notification
0x110	Serial Info
0x200	Serial Send 0
0x204	Serial Receive 0
0x208	Serial Status 0
0x20c	Serial Send 1
0x210	Serial Receive 1
0x214	Serial Status 1
0x218	Serial Send 2
0x21c	Serial Receive 2
0x220	Serial Status 2
0x224	Serial Send 3
0x228	Serial Receive 3
0x22c	Serial Status 3
0x230	Serial Send 4
0x234	Serial Receive 4
0x238	Serial Status 4
0x23c	Serial Send 5
0x240	Serial Receive 5
0x244	Serial Status 5
0x248	Serial Send 6
0x24c	Serial Receive 6
0x250	Serial Status 6
0x254	Serial Send 7
0x258	Serial Receive 7
0x25c	Serial Status 7

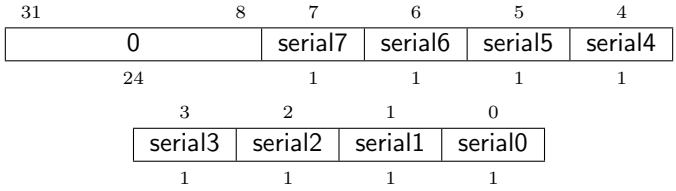
7.12.2 Set Halt Notification (`sethaltnot`, at 0x10c)

Writes to this register set the halt notification bit corresponding to the number written.

A hart must write its hart ID to this register to indicate to the debugger that it has halted spontaneously. (The code to do this is already in the Debug ROM.)



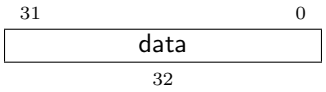
7.12.3 Serial Info (`serinfo`, at 0x110)



Field	Description	Access	Reset
serial7	Like serial0.	R	Preset
serial6	Like serial0.	R	Preset
serial5	Like serial0.	R	Preset
serial4	Like serial0.	R	Preset
serial3	Like serial0.	R	Preset
serial2	Like serial0.	R	Preset
serial1	Like serial0.	R	Preset
serial0	1 means serial interface 0 is supported.	R	Preset

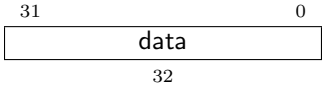
7.12.4 Serial Send 0 (`sersend0`, at 0x200)

Values written to this address are added to the core-to-debugger queue, unless the queue is already full.

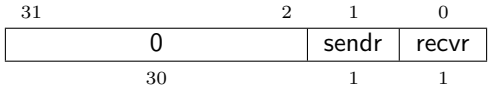


7.12.5 Serial Receive 0 (`serrecv0`, at 0x204)

This register contains the oldest value in the debugger-to-core queue. Reading the register removes that value from the queue. If the queue is empty, reading this register returns an undefined value.



7.12.6 Serial Status 0 (serstat0, at 0x208)



Field	Description	Access	Reset
sendr	Send ready. 1 when the core-to-debugger queue is not full. 0 otherwise.	R	1
recvr	Receive ready. 1 when the debugger-to-core queue is not empty. 0 otherwise.	R	1

8 RISC-V Debug

Modifications to the RISC-V core to support debug are kept to a minimum. There is a special execution mode (Debug Mode) and a few extra CSRs. The code in Debug ROM and resources in the Debug Module take care of the rest.

8.1 Hart IDs

External debug imposes a few limits on hart IDs. Every hart in the system must have a unique ID. (There could be additional harts that reuse IDs, but only one of the harts that share an ID can be debugged.) One of the harts must use ID 0. The debugger needs this to access the Device Tree to enumerate the remaining harts in the system. Hart IDs should be less than 128 if the Debug Bus address is 5 bits wide, or less than 1024 if that address is 6 or more bits wide.

8.2 Debug Mode

Debug Mode is a special processor mode used only when the core is halted for external debugging.

To enter Debug Mode the hart:

1. Saves `pc` to `dpc`.
2. Sets `cause` in `dcsr`.
3. Sets `pc` to 0x800.

While in Debug Mode:

1. All operations happen in machine mode.
2. `mprv` in `mstatus` is ignored.
3. All interrupts are masked. Whether slow watchdog timers (10s or longer) are masked is left to the implementation.
4. All exceptions don't update any registers, and cause the hart to jump to `exception` in Debug ROM. That means no `cause`, `epc`, and `badaddr` registers are changed. `mstatus` isn't updated either.
5. No trigger actions are taken.

6. Trace is disabled.
7. Cycle counters may be stopped, depending on stopcycle in dcsr.
8. Timers may be stopped, depending on stoptime in dcsr.
9. The wfi instruction either acts as nop, or stalls the hart until the Debug Interrupt is set. It ignores any other interrupts.
10. Instructions that change the privilege level have undefined behavior. This includes ecall, ebreak, mret, hret, sret, and uret. The only exception is dret, described in Section 8.4. (To change the privilege level, the debugger can write prv in dcsr.)

8.3 Debug ROM Contents

Table 6: Debug ROM Contents

Address	Name	Pseudocode
0x800	entry	If cause indicates a debug interrupt, jump to Debug RAM. Otherwise, write mhartid to sethaltnot (to notify the debugger), set halt (to track the reason for entry), wait for debugint to be set, and jump to Debug RAM.
0x804	resume	Write 0 to the last word in Debug RAM. Write mhartid to cleardebint (to notify the debugger the hart is back in Debug ROM). If halt is set, wait for debugint to be set, and jump to Debug RAM. Otherwise restore saved registers and resume normal execution at dpc.
0x808	exception	Just like resume, but writes 0xffffffff to the last word in Debug RAM instead of 0.
0x80c	Reserved	Reserved for future standard use.

The Debug ROM (part of the Debug Module) contains the code required for a debugger to communicate with a hart while in Debug Mode. Table 6 summarizes the contents of the Debug ROM, while sample Debug ROM source can be found in Appendix B.

When entering Debug RAM, s0 is saved in dscratch and s1 is saved at the very end of Debug RAM. In between calls to Debug RAM s0 and s1 will change, but all other registers keep their value. Debug ROM code restores both s0 and s1 registers from those locations before leaving Debug Mode.

It is expected that the code in Debug RAM finishes with a jump to resume in Debug ROM.

8.4 dret Instruction

To return from Debug Mode, a new instruction is required: dret. It has an encoding of 0x7b200073. Executing the instructions changes pc to the value

stored in `dpc`. The current privilege level is changed to what's specified by prv in `dcsr`. cause in `dcsr` is cleared since the hart is no longer in Debug Mode.

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

8.5 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.

8.6 Core Debug Registers

The Core Debug Registers must be implemented for each hart being debugged.

These registers are only accessible from Debug Mode.

---

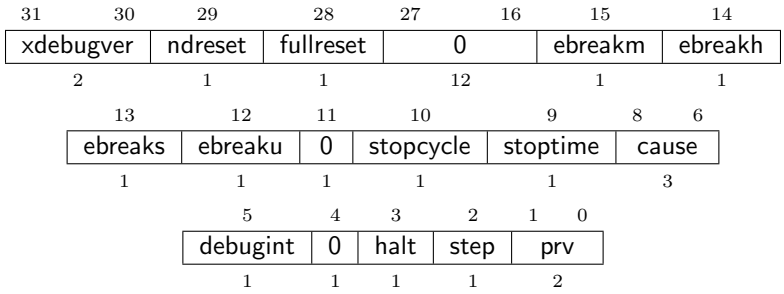
step, halt, *and* prv *all lie in the lower 5 bits so a debugger can manipulate them using* `csrsi` *and* `csrci`.

debugint *and* cause *all lie within a 12-bit immediate so Debug ROM can check them using integer instructions that use immediates.*

Table 7: Core Debug Registers

Address	Name
0x7b0	Debug Control and Status
0x7b1	Debug PC
0x7b2	Debug Scratch Register
virtual	Privilege Level

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



Field	Description	Access	Reset
xdebugver	0: There is no Debug Mode support. 1: Debug Mode exists as it is described in this document. Other values are reserved for future standards.	R	Preset
ndreset	Every time this bit is written as 1, it triggers a full reset of the hart except for the halt bit in this register. This enables a debugger to reset a hart and debug it from the very first instruction executed.	W1	0
fullreset	Every time this bit is written as 1, it triggers a full reset of the hart, including the debug logic.	W1	0
ebreakm	When 1, `ebreak` instructions in Machine Mode enter Debug Mode.	R/W	0
ebreakh	When 1, `ebreak` instructions in Hypervisor Mode enter Debug Mode.	R/W	0
ebreaks	When 1, `ebreak` instructions in Supervisor Mode enter Debug Mode.	R/W	0
ebreaku	When 1, `ebreak` instructions in User/Application Mode enter Debug Mode.	R/W	0
stopcycle	Controls the behavior of any counters while the component is in Debug Mode. This includes the `cycle` and `instret` CSRs. When 1, counters are stopped when the component is in Debug Mode. Otherwise, the counters continue to run. An implementation may choose not to support writing to this bit. The debugger must read back the value it writes to check whether the feature is supported.	R/W	1

Continued on next page

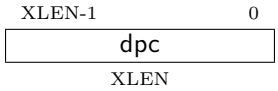
stoptime	Controls the behavior of any timers while the component is in Debug Mode. This includes the `time` and tt timeh CSRs. When 1, timers are stopped when the component is in Debug Mode. Otherwise, the timers continue to run. An implementation may choose not to support writing to this bit. The debugger must read back the value it writes to check whether the feature is supported.	R/W	0
cause	Explains why Debug Mode was entered. When there are multiple reasons to enter Debug Mode in a single cycle, the cause with the highest priority is the one written. 1: A software breakpoint was hit. (priority 3) 2: The Trigger Module caused a halt. (priority 4) 3: The debug interrupt was asserted by the Debug Module. (priority 2) 4: The hart single stepped because step was set. (priority 1) 5: halt was set. (priority 0) Other values are reserved for future use.	R	0
debugint	This bit contains the current value of the debug interrupt signal.	R	0
halt	When this bit is set, the hart enters Debug Mode immediately if it is not already in Debug Mode. The bit is used to enter Debug Mode straight out of reset, and to ensure that spontaneous entries into Debug Mode don't get lost in a race with the debugger.	R/W	0
step	When set and not in Debug Mode, the hart will only execute a single instruction, and then enter Debug Mode. Interrupts are disabled when this bit is set.	R/W	0
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is describe in Table 8. A debugger can change this value to change the hart's privilege level when exiting Debug Mode. Not all privilege levels are supported on all harts. If the encoding written is not supported, the hart may ignore the value, or may change to any supported privilege level.	R/W	0

Table 8: Privilege Level Encoding

Encoding	Privilege Level
0	User/Application
1	Supervisor
2	Hypervisor
3	Machine

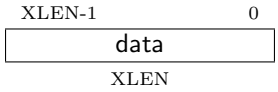
8.6.2 Debug PC (`dpc`, at 0x7b1)

When entering Debug Mode, the current PC is copied here. When leaving Debug Mode, execution resumes at this PC.



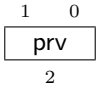
8.6.3 Debug Scratch Register (`dscratch`, at 0x7b2)

Register reserved for Debug ROM where it can save `s0`.



8.6.4 Privilege Level (`priv`, at virtual)

Users of the debugger shouldn't need to know about the debug registers, but might want to inspect and change the privilege level that the hart was running in when the hart halted. To facilitate this, debuggers should expose the privilege level in this virtual register. (A virtual register is one that doesn't exist directly in the hardware, but that the debugger exposes as if it does.)



Field	Description	Access	Reset
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is describe in Table 8. A user can write this value to change the hart's privilege level when exiting Debug Mode.	R/W	0

9 Trigger Module

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

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

1. Write 0 to `tselect`.
2. Read back `tselect` to confirm this trigger exists. If not, exit.
3. Read `tdata1`, and possible `tdata2` and `tdata3` depending on the trigger type.
4. If `type` in `tdata1` was 0, then there are no more triggers.
5. Repeat, incrementing the value in `tselect`.

9.1 Trigger Registers

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

Table 9: Trigger Registers

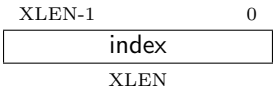
Address	Name
0x7a0	Trigger Select
0x7a1	Trigger Data 1
0x7a1	Match Control
0x7a1	Instruction Count
0x7a2	Trigger Data 2
0x7a3	Trigger Data 3

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

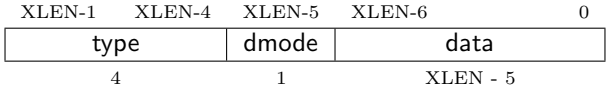
This register determines which trigger is accessible through the other trigger registers. The set of accessible triggers must start at 0, and be contiguous.

Writes of values greater than or equal to the number of supported triggers result in an undefined value in `tselect`. Debuggers should read back the value to confirm that what they wrote was a valid index.

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



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



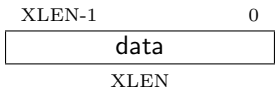
Field	Description	Access	Reset
type	0: There is no trigger at this `tselect`. 1: The trigger is a legacy SiFive address match trigger. These should not be implemented and aren't further documented here. 2: The trigger is an address/data match trigger. 3: The trigger is an instruction count trigger. 15: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

Continued on next page

data	Trigger-specific data.	R/W	Preset

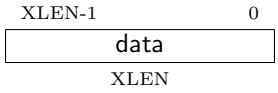
9.1.3 Trigger Data 2 (tdata2, at 0x7a2)

Trigger-specific data.



9.1.4 Trigger Data 3 (tdata3, at 0x7a3)

Trigger-specific data.

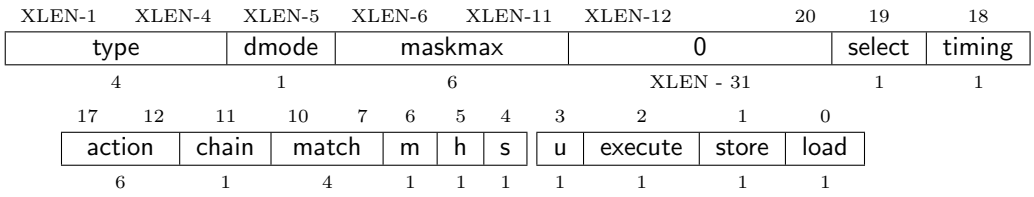


9.1.5 Match Control (mcontrol, at 0x7a1)

This register is accessible as tdata1 when type is 2.

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

Address and data trigger implementation are heavily dependent on how the processor core is implemented. To accommodate various implementations, load and store address and data triggers may fire at whatever point in time is most convenient for the implementation. Following the suggestions in the definitions of store and load will lead to the best user experience, however.



Field	Description	Access	Reset
maskmax	Specifies the largest naturally aligned powers-of-two (NAPOT) range supported by the hardware. The value is the logarithm base 2 of the number of bytes in that range. A value of 0 indicates that only exact value matches are supported (one byte range). A value of 63 corresponds to the maximum NAPOT range, which is 2^{63} bytes in size.	R	0
select	0: Perform a match on the address. 1: Perform a match on the data value loaded/stored, or the instruction executed.	R/W	0
timing	0: The action for this trigger will be taken just before the instruction that triggered it is executed, but after all preceding instructions are are committed. 1: The action for this trigger will be taken after the instruction that triggered it is executed. It should be taken before the next instruction is executed, but it is better to implement triggers and not implement that suggestion than to not implement them at all. Most hardware will only implement one timing or the other, possibly dependent on select, execute, load, and store. This bit primarily exists for the hardware to communicate to the debugger what will happen. Hardware may implement the bit fully writable, in which case the debugger has a little more control. Data load triggers with timing of 0 will result in the same load happening again when the debugger lets the core run. For data load triggers debuggers must first attempt to set the breakpoint with timing of 1. A chain of triggers that don't all have the same timing value will never fire (unless consecutive instructions match the appropriate triggers).	R/W	0

Continued on next page

action	Determines what happens when this trigger matches. 0: Raise a debug exception. (Used when software wants to use the trigger module without an external debugger attached.) 1: Enter Debug Mode. (Only supported when dmode is 1.) 2: Start tracing. 3: Stop tracing. 4: Emit trace data for this match. If it is a data access match, emit appropriate Load/Store Address/Data. If it is an instruction execution, emit its PC. Other values are reserved for future use.	R/W	0
chain	0: When this trigger matches, the configured action is taken. 1: While this trigger does not match, it prevents the trigger with the next index from matching.	R/W	0
match	0: Matches when the value equals `tdata2`. 1: Matches when the top M bits of the value match the top M bits of `tdata2`. M is XLEN-1 minus the index of the least-significant bit containing 0 in `tdata2`. 2: Matches when the value is greater than or equal to `tdata2`. 3: Matches when the value is less than `tdata2`. 4: Matches when the lower half of the value equals the lower half of `tdata2` after the lower half of the value is ANDed with the upper half of `tdata2`. 5: Matches when the upper half of the value equals the lower half of `tdata2` after the upper half of the value is ANDed with the upper half of `tdata2`. Other values are reserved for future use.	R/W	0

Continued on next page

m	When set, enable this trigger in M mode.	R/W	0
h	When set, enable this trigger in H mode.	R/W	0
s	When set, enable this trigger in S mode.	R/W	0
u	When set, enable this trigger in U mode.	R/W	0
execute	When set, the trigger fires on the address or opcode of an instruction that is executed.	R/W	0
store	When set, the trigger fires on the address or data of a store.	R/W	0
load	When set, the trigger fires on the address or data of a load.	R/W	0

9.1.6 Instruction Count (`icount`, at 0x7a1)

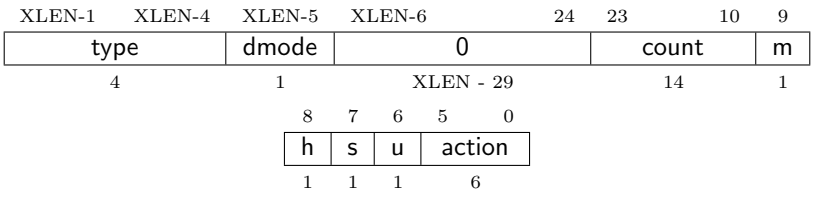
This register is accessible as `tdata1` when type is 3.

Warning! `icount` is just a proposal. So far nobody has commented on it, so it could very easily be removed or changed in the future.

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

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

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



Field	Description	Access	Reset
count	When count is decremented to 0, the trigger fires. Instead of changing count from 1 to 0, it is also acceptable for hardware to clear m, h, s, and u. This allows count to be hard-wired to 1 if this register just exists for single step.	R/W	1
m	When set, every instruction completed in M mode decrements count by 1.	R/W	0
h	When set, every instruction completed in H mode decrements count by 1.	R/W	0
s	When set, every instruction completed in S mode decrements count by 1.	R/W	0
u	When set, every instruction completed in U mode decrements count by 1.	R/W	0
action	Determines what happens when this trigger matches. 0: Raise a debug exception. (Used when software wants to use the trigger module without an external debugger attached.) 1: Enter Debug Mode. (Only supported when dmode is 1.) 2: Start tracing. 3: Stop tracing. 4: Emit trace data for this match. If it is a data access match, emit appropriate Load/Store Address/Data. If it is an instruction execution, emit its PC. Other values are reserved for future use.	R/W	0

10 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).

10.1 Background

JTAG refers to IEEE Std 1149.1-2013. It is a standard that defines test logic that can be included in an integrated circuit to test the interconnections between integrated circuits, test the integrated circuit itself, and observe or modify circuit activity during the components normal operation. We're using it for the third case here. The 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.

10.2 JTAG Connector

Every target's JTAG connector seems to have its own pinout. To make it easy to acquire debug hardware, this spec recommends a connector that is compatible with the Cortex Debug Connector, as described below.

The connector is a .05"-spaced, gold-plated male header with .016" thick hardened copper or beryllium bronze square posts (SAMTEC FTSH-105 or equivalent). Female connectors are compatible 20 µm gold connectors in order to prevent oxide build-up on tin connectors.

Viewing the male header from above (the pins pointing at your eye), a target's connector looks as it does in Table 10. The function of each pin is described in Table 11.

Table 10: JTAG Connector Diagram

VCC	1	2	TMS
GND	3	4	TCK
GND	5	6	TDO
KEY	7	8	TDI
N/C	9	10	RESET

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

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

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

10.3 JTAG Registers

JTAG DTMs should use a 5-bit JTAG IR. 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 12. The only regular JTAG registers a debugger might use are BYPASS and IDCODE, but the JTAG standard recommends a lot of other instructions so we leave IR space for them. If they are not implemented, then they must select the BYPASS register.

10.3.1 IDCODE (00001)

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.

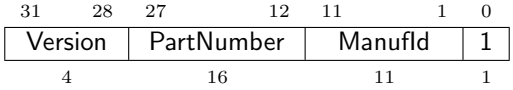


Table 11: JTAG Connector Pinout

1	VCC	Power provided by the target, which may be used to power the debug adapter. Must be able to source at least 25mA. This signal also serves as the reference voltage for logic high. This pin must be clearly marked in both male and female headers.
2	TMS	JTAG TMS signal, driven by debug adapter.
3	GND	Target ground.
4	TCK	JTAG TCK signal, driven by the debug adapter.
5	GND	Target ground.
6	TDO	JTAG TDO signal, driven by the target.
7	KEY	This pin should be clipped in male connectors, and plugged in female connectors. Electrically it must not be connected.
8	TDI	JTAG TDI signal, driven by the debug adapter. This pin may be used by a target to sense a debugger at reset by weakly pulling this signal high during a brief detection period at reset. Debuggers should drive TDI low when the interface is idle.
9	N/C	Not connected in either target or debug adapter. May be used in future specs.
10	RESET	Reset signal, driven by the debug adapter. This may be active low or active high, depending on the target's requirements. A debug adapter must accommodate either option. Asserting reset should reset any RISC-V cores as well as any other peripherals on the PCB. If not implemented in a target, this pin must not be connected.

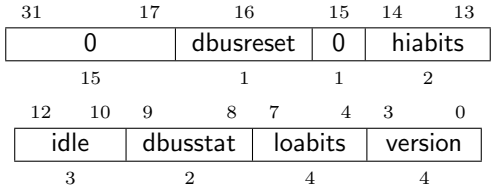
Table 12: JTAG TAP Registers

Address	Name	Description
00000	BYPASS	JTAG recommends this encoding
00001	IDCODE	JTAG recommends this encoding
00010	SAMPLE	JTAG requires this instruction
00011	PRELOAD	JTAG requires this instruction
00100	EXTEST	JTAG requires this instruction
00101	CLAMP	JTAG recommends this instruction
00110	CLAMP_HOLD	JTAG recommends this instruction
00111	CLAMP_RELEASE	JTAG recommends this instruction
01000	HIGHZ	JTAG recommends this instruction
01001	IC_RESET	JTAG recommends this instruction
01010	TMP_STATUS	JTAG recommends this instruction
01011	INIT_SETUP	JTAG recommends this instruction
01100	INIT_SETUP_CLAMP	JTAG recommends this instruction
01101	INIT_RUN	JTAG recommends this instruction
01110	Unused (BYPASS)	Reserved for future JTAG
01111	Unused (BYPASS)	Reserved for future JTAG
10000	DTM Control	For Debugging
10001	Debug Bus Access	For Debugging
10010	Reserved (BYPASS)	Reserved for future RISC-V debugging
10011	Reserved (BYPASS)	Reserved for future RISC-V debugging
10100	Reserved (BYPASS)	Reserved for future RISC-V debugging
10101	Reserved (BYPASS)	Reserved for future RISC-V standards
10110	Reserved (BYPASS)	Reserved for future RISC-V standards
10111	Reserved (BYPASS)	Reserved for future RISC-V standards
11000	Unused (BYPASS)	Reserved for customization
11001	Unused (BYPASS)	Reserved for customization
11010	Unused (BYPASS)	Reserved for customization
11011	Unused (BYPASS)	Reserved for customization
11100	Unused (BYPASS)	Reserved for customization
11101	Unused (BYPASS)	Reserved for customization
11110	Unused (BYPASS)	Reserved for customization
11111	BYPASS	JTAG requires this encoding

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

10.3.2 DTM Control (`dtmcontrol`, at 10000)

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



Field	Description	Access	Reset
dbusreset	Writing 1 to this bit resets the dbus controller, clearing any sticky error state.	W1	0
hiabits	Bits 5:4 of abits, which describes the size of address in `dbus`.	R	Preset
idle	The number of cycles a debugger needs to send the target through Run-Test/Idle after every dbus scan. 0: It's 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

Continued on next page

dbusstat	0: No error. 2: An operation failed. 3: An operation was attempted while a bus access was still in progress.	R	0
loabits	Bits 3:0 of abits, which describes the size of address in dbus.	R	Preset
version	0 indicates the version described in this document. Other values are reserved for future use.	R	0

10.3.3 Debug Bus Access (dbus, at 10001)

This register allows access to the Debug Bus.

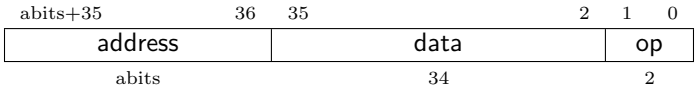
In Update-DR, the DTM starts the operation specified in op unless the current status reported in op is sticky.

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

See Section A.1 and Table 13 for examples of how this plays out.

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 Debug Bus access. In Update-DR this value is sent to the DM.	R/W	0
data	The data to send to the DM during Update-DR, and the data returned from the previous operation to the DM.	R/W	0
op	When the debugger writes this field, it has the following meaning: 0: Ignore data. (nop) 1: Read from address. (read) 2: Read from address. Then write data to address. (write) 3: Reserved. When the debugger reads this field, it means the following: 0: The previous operation completed successfully. 1: Reserved. 2: The previous operation failed. The data scanned into `dbus` in this access will be ignored. This status is sticky and can be cleared by writing dbusreset in `dtmcontrol`. 3: The previous operation is still in progress. The data scanned into `dbus` in this access will be ignored. This status is sticky and can be cleared by writing dbusreset in `dtmcontrol`. If a debugger sees this status, it needs to give the target more time between Update-DR and Capture-DR. The simplest way to do that is to add extra transitions in Run-Test/Idle.	R/W	0

10.3.4 BYPASS (11111)

1-bit register that has no effect. It's used when a debugger wants to talk to a different TAP in the same scan chain as this one.

A 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. All these examples assume a 32-bit core but it should be easy to adapt the examples to 64- or 128-bit cores.

A.1 Debug Bus Access

To read an arbitrary Debug Bus register, select `dbus`, 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 error condition must be cleared by writing `dbusreset` in `dtmcontrol`, and then the operation must be tried again. This time the debugger should allow for more time between Capture-DR and Update-DR.

To write an arbitrary Debug Bus register, select `dbus`, 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 also performed right after the read. The operation isn't considered complete until the write has happened.

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

A.2 Debug RAM

All operations are executed by writing a debug program to Debug RAM, and having the core execute it. This comes down to a series of Debug Bus accesses as described above.

First, select the hart of interest by writing its ID to `hartid` in `dmcontrol`. Then write the program to Debug RAM with a series of writes to 0x0 – 0xf. On the last write, set the interrupt bit. This triggers program execution. Perform a read to poll the interrupt bit to wait for it to clear. Typically JTAG will be so slow that the interrupt bit will be clear on the very next scan. An efficient debugger can optimistically assume all operations will complete immediately. If it discovers that is not the case (by seeing `interrupt` is still set on a scan) it can adapt by increasing the delay between scans.

That means if eg. you are doing a block write and have the program mostly set up, you can write a new data value and set the interrupt bit every time through the loop using a write operation.

A.3 Main Loop

A debugger continuously monitors `haltsum` to see if any harts have spontaneously halted. While this is going on, a debugger might perform a quick

operation (most likely a memory access) by writing a debug program that ends with a jump to `resume`, and asserting a hart's Debug Interrupt.

To halt a hart, the debugger writes a debug program that sets halt and ends with a jump to `resume`. Then it asserts the hart's Debug Interrupt.

Once halted, the debugger can write debug programs in exactly the same way that it can when the hart is running. The only difference is that in this case the jump to `resume` ends in up in Debug ROM code that waits until the debug interrupt is set again, instead of by continuing normal execution. To resume normal execution, the debugger writes a debug program that clears halt before jumping to `resume`.

Performing operations then all comes down to writing the appropriate program to Debug RAM, so the sections below mostly consist of short program listings. The Debug ROM already takes care of saving `s0` and `s1` so those registers can be used without any extra precautions.

A.4 Reading Memory

Execute the following program, and then read the value that was read from `data`.

```
        lw      s1, address
        lw      s0, 0(s1)
        sw      s0, data
        j       resume
address:.word   ADDRESS
data:   .word   DATA
```

A shorter program is possible by hardcoding the address in a set of `lui`/`lw` instructions, but this technique would be limited to 32-bit cores. This version also has the nice property that to immediately read from a different address, only 1 word in Debug RAM needs to be changed.

A slightly different program can be used to read memory very efficiently:

```
        lw      s1, data
        lw      s0, 0(s1)
        sw      s0, data
        j       resume
data:   .word   ADDRESS
```

In this case `data` is used both for the address and data. After this program is executed, the data value is in `data`. Since a JTAG bus write first performs a read, a single scan can read this data value, write the next address value, and assert the Debug Interrupt. As many of those scans can be used as necessary, and every scan results in 32 bits of data being read. The debugger does need to make sure interrupt is clear in each read. If it is not the data cannot be trusted, and the same address should be read again.

Table 13 shows the scans involved in reading a single word using this method. In this table dram[n] refers to the location in Debug RAM with address n. If more words need to be read, then this can by pipelined, by changing the scan in step 6 to a write of the next address.

A.5 Writing Memory

To write a single word:

```
        lw      s1, data
        lw      s0, address
        sw      s1, 0(s0)
        j       resume
address:.word   ADDRESS
data:   .word   DATA
```

To efficiently write a block of memory, the debug program can take care of incrementing the address. First save `t0` and load the start address into it:

```
        sw      t0, data
        lw      t0, address
        j       resume
address:.word   ADDRESS
data:   .word   0
```

Then write the following program with the first data value, and assert the Debug Interrupt. Additional words can be written by writing the next data value and asserting the Debug Interrupt.

```
        lw      s0, data
        sw      s0, 0(t0)
        addi    t0, t0, 4
        j       resume
data:   .word   DATA
```

After the second program is written, each word can be written to the target in 43 TCK cycles. That's 75% efficient, and translates to a download speed of 908KB/s at a 10MHz TCK. That should be good enough that it's not worth making the JTAG interface more complex to improve the efficiency. (This assumes the Debug Bus uses 5 address bits and that the debugger never has to wait for the core.)

A.6 Halt

To halt a hart, first write the code to be executed to Debug RAM. Then assert the relevant Debug Interrupt.

The code to be executed must set `halt`. Reading `pc` and `dcsr` is optional, but it's extremely common for a debugger to do this immediately after halting.

Table 13: Memory Read Timeline

	JTAG State	Activity
1	Shift-DR	Debugger shifts in write of 0x41002403 to dram[0], and gets back the result of whatever happened previously.
	Update-DR	DTM starts read from dram[0], followed by write to dram[0].
2	Capture-DR	JTAG DTM captures results of read from dram[0].
	Shift-DR	Debugger shifts in write of 0x42483 to dram[1], and gets back the old contents of the first word in Debug RAM.
	Update-DR	DTM starts read from dram[1], followed by write to dram[1].
3	Capture-DR	JTAG DTM captures results of read from dram[1].
	Shift-DR	Debugger shifts in write of 0x40902823 to dram[2], and gets back the old contents of the second word in Debug RAM.
	Update-DR	DTM starts read from dram[2], followed by write to dram[2].
4	Capture-DR	JTAG DTM captures results of read from dram[2].
	Shift-DR	Debugger shifts in write of 0x3f80006f to dram[3], and gets back the old contents of the third word in Debug RAM.
	Update-DR	DTM starts read from dram[3], followed by write to dram[3].
5	Capture-DR	JTAG DTM captures results of read from dram[3].
	Shift-DR	Debugger shifts in write of the address the user wants to read from to dram[4], while also asserting the Debug Interrupt. The old contents of the fourth word in Debug RAM are shifted out.
	Update-DR	DTM starts read from dram[4], followed by write to dram[4] which also asserts Debug Interrupt. The hart will respond to the Debug Interrupt by executing the program in Debug RAM which in this case will read the address written, and replace the entry in Debug RAM with the data at that address.
6	Capture-DR	JTAG DTM captures results of read from dram[4].
	Shift-DR	Debugger shifts in read from dram[4], and gets back the old contents of the fourth word in Debug RAM. (This is the value that was there just before the address was written there.)
	Update-DR	DTM starts read from dram[4].
7	Capture-DR	JTAG DTM captures results of read from dram[4].
	Shift-DR	Debugger shifts in nop, and gets back the contents of the fourth word in Debug RAM. This is the value that was there during the previous Update-DR, which is the result of the Debug Program execution.

```
dpc:    csrsi   DCSR, DCSR_HALT_MASK
dcsr:   csrr    s1, DPC
        sw      s1, dpc
        csrr    s1, DCSR
        sw      s1, dcsr
        j       resume
```

A.7 Reading Registers

Eg. how to read `f1`:

```
        fsw     f1, data
        j       resume
data:   .word   0
```

A.8 Writing Registers

Eg. how to write `mepc`.

```
        lw      s0, data
        csrw    MEPC, s0
        j       resume
data:   .word   DATA
```

A.9 Running

To let the core run once it's halted, the debugger needs to first clear the Halt Notification using the debug bus directly. If the debugger used any registers besides `s0` and `s1` as scratch registers, now is the time to restore them. Finally:

```
        csrci   DCSR, DCSR_HALT_MASK
        j       resume
```

A.10 Single Step

A debugger can single step the core by setting a breakpoint on the next instruction and letting the core run, or by asking the hardware to perform a single step. The former requires the debugger to have much more knowledge of the hardware than the latter, so the latter is preferred.

Using the hardware single step feature is almost the same as regular running. The debugger just sets `step` in `dcsr` before leaving Debug Mode. The core behaves exactly as in the running case, except that interrupts are left off and it only fetches and executes a single instruction before re-entering Debug Mode.

A.11 Handling Exceptions

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

When an exception occurs in Debug Mode no registers are updated, but Debug ROM will write 0xffffffff to the last word of Debug RAM. If the debugger thinks an exception may have occurred it should check for that. If no exception occurred in the last entry to Debug RAM, then the word must contain 0. If there was an exception, it's left to the debugger to know what must have caused it.

B Debug ROM Source

```
#include "riscv/encoding.h"

#define DEBUG_RAM               0x400
#define DEBUG_RAM_SIZE          64

#define CLEARDEBINT             0x100
#define SETHALTNOT              0x10c

        .global entry
        .global resume
        .global exception

        # Automatically called when Debug Mode is first entered.
entry:  j       _entry
        # Should be called by Debug RAM code that has finished execution and
        # wants to return to Debug Mode.
resume:
        j       _resume
exception:
        # Set the last word of Debug RAM to all ones, to indicate that we hit
        # an exception.
        li      s0, ~0
        j       _resume2

_resume:
        li      s0, 0
_resume2:
        fence
```

```
        # Restore s1.
        csrr    s1, CSR_MISA
        bltz    s1, restore_not_32
restore_32:
        lw      s1, (DEBUG_RAM + DEBUG_RAM_SIZE - 4)(zero)
        j       finish_restore
restore_not_32:
        slli    s1, s1, 1
        bltz    s1, restore_128
restore_64:
        ld      s1, (DEBUG_RAM + DEBUG_RAM_SIZE - 8)(zero)
        j       finish_restore
restore_128:
        nop     #lq     s1, (DEBUG_RAM + DEBUG_RAM_SIZE - 16)(zero)

finish_restore:
        # s0 contains ~0 if we got here through an exception, and 0 otherwise.
        # Store this to the last word in Debug RAM so the debugger can tell if
        # an exception occurred.
        sw      s0, (DEBUG_RAM + DEBUG_RAM_SIZE - 4)(zero)

        # Clear debug interrupt.
        csrr    s0, CSR_MHARTID
        sw      s0, CLEARDEBINT(zero)

check_halt:
        csrr    s0, CSR_DCSR
        andi    s0, s0, DCSR_HALT
        beqz    s0, exit
        j       wait_for_interrupt

exit:
        # Restore s0.
        csrr    s0, CSR_DSCRATCH
        dret

_entry:
        # Save s0 in DSCRATCH
        csrw    CSR_DSCRATCH, s0

        # Check why we're here
        csrr    s0, CSR_DCSR
        # cause is in bits 8:6 of dcsr
        andi    s0, s0, DCSR_CAUSE
        addi    s0, s0, -(DCSR_CAUSE_DEBUGINT<<6)
        bnez    s0, spontaneous_halt
```

```
jdebugram:
        # Save s1 so that the debug program can use two registers.
        fence.i
        csrr    s0, CSR_MISA
        bltz    s0, save_not_32
save_32:
        sw      s1, (DEBUG_RAM + DEBUG_RAM_SIZE - 4)(zero)
        jr      zero, DEBUG_RAM
save_not_32:
        slli    s0, s0, 1
        bltz    s0, save_128
save_64:
        sd      s1, (DEBUG_RAM + DEBUG_RAM_SIZE - 8)(zero)
        jr      zero, DEBUG_RAM
save_128:
        nop     #sq      s1, (DEBUG_RAM + DEBUG_RAM_SIZE - 16)(zero)
        jr      zero, DEBUG_RAM

spontaneous_halt:
        csrr    s0, CSR_MHARTID
        sw      s0, SETHALTNOT(zero)
        csrsi   CSR_DCSR, DCSR_HALT

wait_for_interrupt:
        csrr    s0, CSR_DCSR
        andi    s0, s0, DCSR_DEBUGINT
        beqz    s0, wait_for_interrupt

        j       jdebugram
```

C Trace Module

This part of the spec needs work before it's ready to be implemented, which is why it's in the appendix. It's left here to give a rough idea of some of the issues to consider.

Aside from viewing the current state of a core, knowing what happened in the past can be incredibly helpful. Capturing an execution trace can give a user that view. Unfortunately processors run so fast that they generate trace data at a very large rate. To help deal with this, the trace data format allows for some simple compression.

The trace functionality described here aims to support 3 different use cases:

1. Full reconstruction of all processor state, including register values etc. To achieve this goal the decoder will have to know what code is being executed, and know the exact behavior of every RISC-V instruction.

2. Reconstruct just the instruction stream. Get enough data from the trace stream that it is possible to make a list of every instruction executed. This is possible without knowing anything about the code or the core executing it.

3. Watch memory accesses for a certain memory region.

Trace data may be stored to a special on-core RAM, RAM on the system bus, or to a dedicated off-chip interface. Only the system RAM destination is covered here.

C.1 Trace Data Format

Trace data should be both compact and easy to generate. Ideally it's also easy to decode, but since decoding doesn't have to happen in real time and will usually have a powerful workstation to do the work, this is the least important concern.

Trace data consists of a stream of 4-bit packets, which are stored in memory in 32-bit words by putting the first packet in bits 3:0 of the 32-bit word, the second packet into bits 7:4, and so on. Trace packets and their encoding are listed in Table 14.

Several header packets are followed by a Value Sequence, which can encode values between 4 and 64 bits. The sequence consists first of a 4-bit size packet which contains a single number N. It is followed by N+1 4-bit packets which contain the value. The first packet contains bits 3:0 of the value. The next packet contains bits 7:4, and so on.

C.2 Trace Events

Trace events are events that occur when a core is running that result in trace packets being emitted. They are listed in Table 15.

C.3 Synchronization

If a trace buffer wraps, it is no longer clear what in the buffer is a header and what isn't. To guarantee that a trace decoder can sync up easily, each trace buffer must have 8 synchronization points, spaced evenly throughout the buffer, with the first one at the very start of the buffer. A synchronization point is simply an address where there is guaranteed to be a sequence header. To make this happen, the trace source can insert a number of Nop headers into the sequence just before writing to the synchronization point.

Aside from synchronizing a place in the data stream, it's also necessary to send a full PC, Read Address, Write Address, and Timestamp in order for those to be fully decoded. Ideally that happens the first time after every synchronization point, but bandwidth might prevent that. A trace source should attempt to send one full value for each of these (assuming they're enabled) soon after each synchronization point.

Table 14: Trace Sequence Header Packets

0000	Nop	Packet that indicates no data. The trace source must use these to ensure that there are 8 synchronization points in each buffer.
0001	PC	Followed by a Value Sequence containing bits XLEN-1:1 of the PC if the compressed ISA is supported, or bits XLEN-1:2 of the PC if the compressed ISA is not supported. Missing bits must be filled in with the last PC value.
0010	Branch Taken	
0011	Branch Not Taken	
0100	Trace Enabled	Followed by a single packet indicating the version of the trace data (currently 0).
0101	Trace Disabled	Indicates that trace was purposefully disabled, or that some sequences were dropped because the trace buffer overflowed.
0110	Privilege Level	Followed by a packet containing whether the cause of the change was an interrupt (1) or something else (0) in bit 3, PRV[1:0] in bits 2:1, and IE in bit 0.
0111	Change Hart	Followed by a Value Sequence containing the hart ID of the hart whose trace data follows. Missing bits must be filled in with 0.
1000	Load Address	Followed by a Value Sequence containing the address. Missing bits must be filled in with the last Load Address value.
1001	Store Address	Followed by a Value Sequence containing the address. Missing bits must be filled in with the last Store Address value.
1010	Load Data	Followed by a Value Sequence containing the data. Missing bits must be filled in by sign extending the value.
1011	Store Data	Followed by a Value Sequence containing the data. Missing bits must be filled in by sign extending the value.
1100	Timestamp	Followed by a Value Sequence containing the timestamp. Missing bits should be filled in with the last Timestamp value.
1101	Reserved	Reserved for future standards.
1110	Custom	Reserved for custom trace data.
1111	Custom	Reserved for custom trace data.

Table 15: Trace Data Events

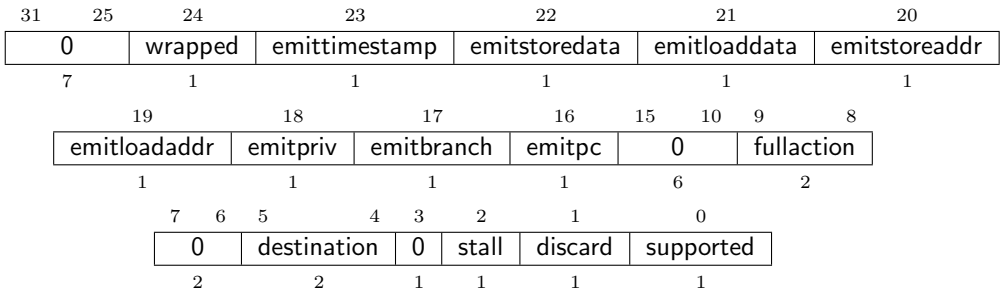
Opcode	Action
`jal`	If emitbranch is disabled but emitpc is enabled, emit 2 PC values: first the address of the instruction, then the address being jumped to.
`jalr`	If emitbranch is disabled but emitpc is enabled, emit 2 PC values: first the address of the instruction, then the address being jumped to. Otherwise, if emitstoredata is enabled emit just the destination PC.
BRANCH	If emitbranch is enabled, emit either Branch Taken or Branch Not Taken. Otherwise if emitpc is enabled and the branch is taken, emit 2 PC values: first the address of the branch, then the address being branched to.
LOAD	If emitloadaddr is enabled, emit the address. If emitloaddata is enabled, emit the data that was loaded.
STORE	If emitstoreaddr is enabled, emit the address. If emitstoredata is enabled, emit the data that is stored.
Traps	`scall`, `sbreak`, `ecall`, `ebreak`, and `eret` emit the same as if they were `jal` instructions. In addition they also emit a Privilege Level sequence.
Interrupts	Emit PC (if enabled) of the last instruction executed. Emit Privilege Level (if enabled). Finally emit the new PC (if enabled).
CSR instructions	For reads emit Load Data (if enabled). For writes emit Store Data (if enabled).
Data Dropped	After packet sequences are dropped because data is generated too quickly, Trace Disabled must be emitted. It's not necessary to follow that up with a Trace Enabled sequence.

C.4 Trace Registers

Table 16: Trace Registers

Address	Name
0x728	Trace
0x729	Trace Buffer Start
0x72a	Trace Buffer End
0x72b	Trace Buffer Write

C.4.1 Trace (`trace`, at 0x728)



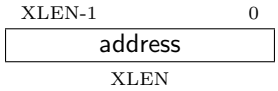
Field	Description	Access	Reset
wrapped	1 if the trace buffer has wrapped since the last time discard was written. 0 otherwise.	R	0
emittimestamp	Emit Timestamp trace sequences.	R/W	0
emitstoredata	Emit Store Data trace sequences.	R/W	0
emitloaddata	Emit Load Data trace sequences.	R/W	0
emitstoreaddr	Emit Store Address trace sequences.	R/W	0
emitloadaddr	Emit Load Address trace sequences.	R/W	0
emitpriv	Emit Privilege Level trace sequences.	R/W	0
emitbranch	Emit Branch Taken and Branch Not Taken trace sequences.	R/W	0
emitpc	Emit PC trace sequences.	R/W	0
fullaction	Determine what happens when the trace buffer is full. 0 means wrap and overwrite. 1 means turn off trace until discard is written as 1. 2 means cause a trace full exception. 3 is reserved for future use.	R/W	0
destination	0: Trace to a dedicated on-core RAM (which is not further defined in this spec). 1: Trace to RAM on the system bus. 2: Send trace data to a dedicated off-chip interface (which is not defined in this spec). This does not affect execution speed. 3: Reserved for future use. Options 0 and 1 slow down execution (eg. because of system bus contention).	R/W	Preset

Continued on next page

stall	When 1, the trace logic may stall processor execution to ensure it can emit all the trace sequences required. When 0 individual trace sequences may be dropped.	R/W	1
discard	Writing 1 to this bit tells the trace logic that any trace collected is no longer required. When tracing to RAM, it resets the trace write pointer to the start of the memory, as well as wrapped.	W1	0

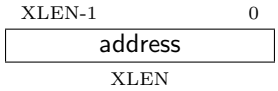
C.4.2 Trace Buffer Start (tbufstart, at 0x729)

If destination is 1, this register contains the start address of block of RAM reserved for trace data.



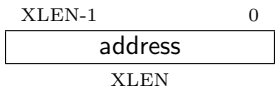
C.4.3 Trace Buffer End (tbufend, at 0x72a)

If destination is 1, this register contains the end address (exclusive) of block of RAM reserved for trace data.



C.4.4 Trace Buffer Write (tbufwrite, at 0x72b)

If destination is 1, this read-only register contains the address that the next trace packet will be written to.



D Future Ideas

Some future version of this spec may implement some of the following features.

1. The spec defines several additions to the Device Tree which enable a debugger to discover hart IDs and supported triggers for all the cores in the system.

2. DTMs can function as general bus slaves, so they would look like regular RAM to bus masters.

3. Harts can be divided into groups. All the harts in the same group can be halted/run/stepped simultaneously. When a hart hits a breakpoint, all the other harts in the same group also halt within a few clock cycles.

4. DTMs are specified for protocols like USB, I2C, SPI, and SWD.

5. Core registers can be read without halting the processor.

6. The debugger can communicate with the power manager to power cores up or down, and to query their status.

7. Serial ports can raise an interrupt when a send/receive queue becomes full/empty.

8. The debug interrupt can be masked by running code. If the interrupt is asserted, then deasserted, and then asserted again the debug interrupt happens anyway. This mechanism can be used to eg. read/write memory with minimal interruption, making sure never to interrupt during a critical piece of code.

9. The debugger can non-intrusively sample a recent PC value from any running hart.

D.1 Lightweight Brainstorming

At least one person has expressed interest in an absolute minimal gate count debug spec. Here are some ideas that take this existing design, and attempt to minimize its gate count while retaining at least its spirit.

This proposal preserves the focus on having the debugger feed the hart instructions, as well as not adding any slave interfaces to each hart.

Debug Mode is not like other mode changes. (This is a major difference with the full-featured spec.) Instead, in Debug Mode all instruction fetches come from address $0x480 + 8 * \mathsf{hart\ id}$ regardless of what the PC is. The core will keep updating the PC as usual. (Eg. an ALU instruction will increment it, and a jump instruction will change it to the jump destination.) Caches are disabled, as in the current spec. There may not be any speculative instruction fetching. When the PC hits the max value, it must wrap to 0 when incremented.

While in Debug Mode, the hart can exchange data with the debugger by accessing address $0x484 + 8 * \mathsf{hart\ id}$.

0x480 is chosen as a base, so a Debug Module can support a mixture of harts that use the Lightweight and Optimized debug interface.

When the Debug Module has a new instruction for a hart, it returns that instruction on an instruction fetch. Any other time it will return a jump-to-self instruction, or keep the access alive until it has an instruction.

When a data write to the Debug Module happens, it must accept and remember this write. If necessary it must keep the access alive until the value has been seen by the debugger. Likewise for reads it must present the hart with a value it has not yet read.

The Debug Module can tell that a hart is in Debug Mode, because if it is then it'll be performing memory access to it.

A debugger can see this happening, and when it does, it sends:

```
sw      s0, 0x484(zero)
# read s0 from data slot
auipc   s0, 0
sw      s0, 0x484(zero)
# read "pc" from data slot
```

to save s0, and then to read the PC. Note that the actual PC where the target was halted will be 4 (8?) less than the stored value.

No exceptions are taken in Debug Mode, so there is no way to check whether a load/store raised an exception. Unless we add status bits somewhere? Ditto for accessing non-existent CSRs.

dret instruction leaves debug mode, leaving the PC exactly where it is. So to resume:

```
lw      s0, 0x484(zero)
# write "pc" to data slot
jr      s0
dret
```

Simple Debug Module will just pass each access onto the DTM. Ie. set some bits indicating what kind of access is pending. Then the polling DTM will come along, read/write a value, and indicate the access can be completed. The DM could just store bits 6:2 of the address and read/write bit.

RV64. It would be nice to keep the Debug Module 32-bit. That does mean to store a register value you have to sw/shift/sw. Keeping the Debug Module 32-bit allows for more harts (since less address space is needed per store), and less bits used for JTAG shift registers etc. It's also one less parameter needed to instantiate it.

Read memory:

```
lw      s0, 0x484(zero)
# write address to data slot
lw      s1, 0(s0)
sw      s1, 0x484(zero)
# read data from data slot
addi    s1, s1, 4
# repeat... 3 scans per word
```

E Change Log

Revision	Date	Author(s)	Description

0.10	Jul 11	TN	Version shared for 4th RISC-V Workshop
0.11	Jul 17	TN	Updated Bus Access section
0.11	Jul 29	TN	`mcontrol` is intended to alias `tdata0`. Remove conditional writes.
0.11	Aug 11	TN	Document `wfi` behavior in Debug Mode. Core Debug Registers are only accessible in Debug Mode. Removed hwbpcount field from `dcsr`.
0.11	Aug 22	TN	Clarify meaning of `mode` in `tselect`. All triggers are either locked by the debugger, or none are.
0.11	Aug 23	TN	Triggers are locked on an individual basis again. See `dmode` in `tdata0`. The definitions of `chain` in `mcontrol` has been changed to be easier to explain. Changed recommendations on when store hardware breakpoints fire.
0.11	Aug 25	TN	Add `timing` to `mcontrol`. Changed encoding of `action`.
0.11	Aug 26	TN	Document `lr`/`sc` behavior. Rename `tdata0`–`tdata2` to `tdata1`–`tdata3`.
0.11	Sep 6	TN	Change Debug Bus to only support simple read and write operations.
0.11	Sep 7	TN	Clarify `op` in `dbus`.
0.11	Sep 13	TN	Add rough idea of instruction count triggers that could be used for single step.
0.11	Sep 27	TN	M-mode writes to triggers with dmode=1 are ignored instead of raising an exception.
0.11	Oct 24	TN	Change serial full bit to full/overflow.
0.11	Nov 2	TN	There actually can be side effects to reading Debug Bus registers (specifically reading serial data).