RISC-V External Debug Support
Version 0.9jan27

Tim Newsome

January 27, 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

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

1.1 Terminology

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

1.2 Background

There are two forms of external debugging. The first is halt/freeze mode debugging, where an external debugger will halt some or all components of a platform and inspect them while everything is in stasis. Then the debugger can either let the hardware perform a single step or let it run freely. The second is run mode debugging. In this mode there is some 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.

There's a third use for the external debug interface, which 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.

1.3 Supported Features

The debug interface laid out here supports the following features:

1. Any component in the platform can be independently debugged.
2. Any core can be debugged just by using the system bus (unless a dedicated debug bus is used).
3. It's not necessary for a debugger to poll a component's state to see whether it has halted/has completed something.
4. More than one debug transport can be used. They all use a common system bus protocol to communicate with components being debugged.
5. Arbitrary instructions can be executed on a halted RISC-V core.
6. Data can be transferred between a RISC-V core and the debugger without relying on shared RAM.
7. The debug transport may implement serial ports which can be used for communication between debugger and monitor, or as a general protocol between debugger and application.
8. Code can be downloaded efficiently.
9. Each core can be debugged from very first instruction executed.
10. A RISC-V core can be halted when a software breakpoint instruction is executed.

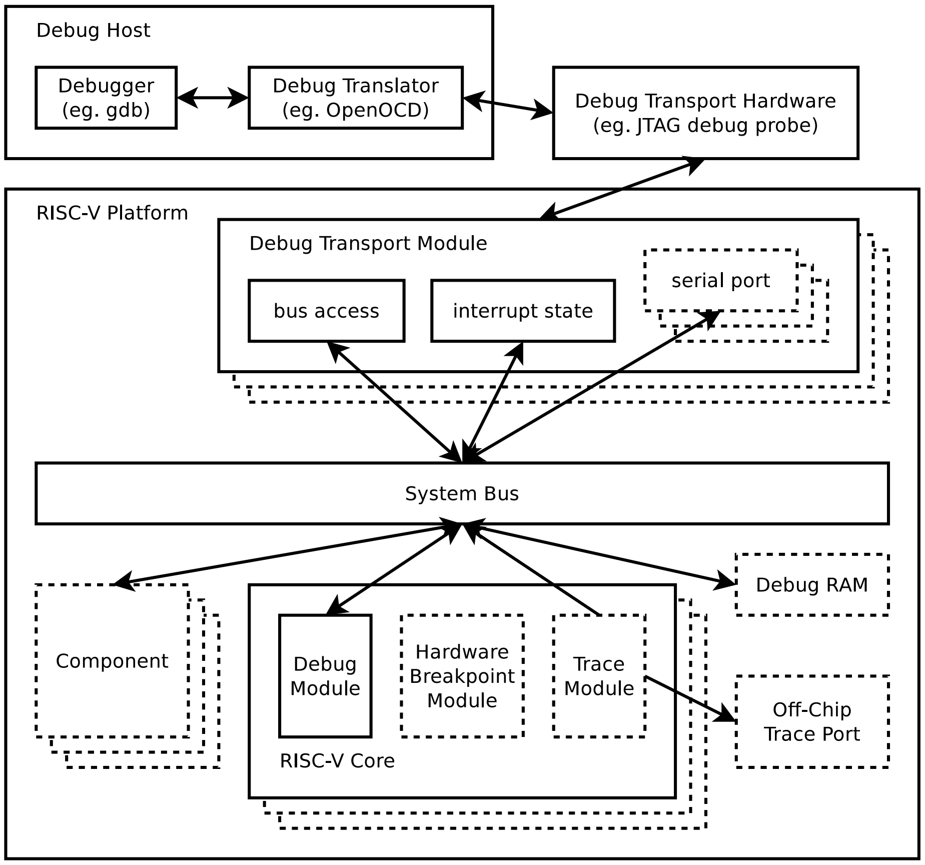


Figure 1: RISC-V Debug System Overview

11. A RISC-V core can be halted when a hardware breakpoint matches PC, or read/write address/data.
12. A RISC-V core can store an execution trace to on- or off-chip RAM.
13. The core can execute code while remaining in Debug Mode.
14. It's always possible to halt a RISC-V core, even if some other component is writing all over the system bus.

2 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. That hardware is also connected to the Platform, which contains a Debug Transport Module.

The Debug Transport Module provides bus access, keeps track of simple interrupts, and may implement serial ports to facilitate communication between code running on the core and the debugger. This bus could be the system bus as depicted, or a dedicated debug bus. Any component that supports some basic features may be debugged over that bus. For RISC-V cores, the Debug Module controls most debug features. Additionally there may be a Hardware Breakpoint Module and a Trace Module that can write trace information to the System Bus or an off-chip trace port.

The platform may contain a Debug RAM to be used when debugging RISC-V cores.

3 Debug Transport Module

Debug Transport Modules provide access to the system bus over one or more transports (eg. JTAG or USB). They also implement a simple interrupt tracking feature that helps notify debuggers of component updates without them having to poll over the system bus. Finally they may implement some serial ports.

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 would include a Debug Transport Module. This would trivially allow any platform where the system bus is used as the debug bus to be debugged over USB.

3.1 System Bus Access

While the details are left completely to the transport-specific Debug Transport Module, every DTM must support all accesses from the following list that the system bus supports: 8-bit read/write, 16-bit read/write, 32-bit read/write, 64-bit read/write, and 128-bit read/write to arbitrary addresses on the system bus.

In addition, DTM designers should keep the following common use cases in mind:

1. XLEN-bit reads from consecutive addresses.
2. XLEN-bit writes to consecutive addresses.
3. Repeatedly read and write XLEN bits at addresses that are adjacent or very close.

Some implementations may decide they don't want debug accesses to use the system bus. Instead they may run a dedicated debug bus through the platform. This has the benefit that debugging does not interfere at all with other execution, and there is no need to make RISC-V cores system bus slaves. The downsides are that an extra bus needs to be run, and it's not possible to debug a component from anything but the DTM. On a small platform with a single RISC-V core it makes sense to have a dedicated debug bus (which can be very simple if there is just a single DTM and a single component to be debugged). For more complex platforms, implementers will likely choose to use the system bus.

The existence of a dedicated debug bus is transparent to the debugger. The DTM simply designates part of the address space (only when accessed by the DTM) as debug bus space. This address space should not mask a device that a debugger might conceivably want to access. If there is no address space available to fit the debug bus in, the DTM must add another address bit which is used to select the debug bus (when 1) or the system bus (when 0).

3.2 Interrupt Tracking

To avoid a debugger constantly polling the components it's interested in (cluttering up the system bus), very simple interrupt mechanism is supported. It consists of a single register where components can set bits by writing the bit's index to a register in the DTM. The width of this register is implementation-specific. It must be at least 1 bit wide. Sensible widths are the number of debuggable components in the platform, and the width of the data bus. When a bit becomes set, the DTM should communicate that to the debugger as soon as possible.

This mechanism exists so components can let the debugger know they are now halted (eg. because a breakpoint was hit), but may have other uses. Which interrupt each component uses is configurable by the debugger by writing the component's interrupt in `ccsr`.

3.3 Serial Ports

Each DTM may implement up to 8 serial ports. They support basic flow control and full duplex data transfer between a component and the debugger. They're intended to be used for the equivalent of printf debugging, or to provide a simple CLI without requiring any extra peripherals.

3.4 Security

It may be necessary to prevent just anyone from accessing the debug interface. One option could be to add a fuse bit to the DTM that can be used to be permanently disable it. Since this is transport and technology specific, it is not further addressed in this spec.

Another option is to allow the DTM to be unlocked only by people who have the key. A simple mechanism is documented in Section 3.5. When authenticated is clear, the DTM must not perform any System Bus accesses, as well as prevent all external access to the serial ports and interrupt state.

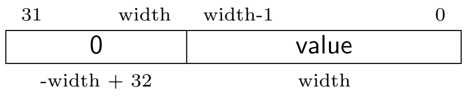
3.5 Debug Transport Module Registers

Table 1: Debug Transport Module Registers

Address	Name
0x0	Interrupt
0x4	Control
0x8	Authentication Data
0x10	Serial Info
0x20	Serial Send 0
0x30	Serial Receive 0
0x40	Serial Status 0
0x50	Serial Send 1
0x60	Serial Receive 1
0x70	Serial Status 1
0x80	Serial Send 2
0x90	Serial Receive 2
0xa0	Serial Status 2
0xb0	Serial Send 3
0xc0	Serial Receive 3
0xd0	Serial Status 3
0xf0	Serial Send 4
0x100	Serial Receive 4
0x110	Serial Status 4
0x120	Serial Send 5
0x130	Serial Receive 5
0x140	Serial Status 5
0x150	Serial Send 6
0x160	Serial Receive 6
0x170	Serial Status 6
0x180	Serial Send 7
0x190	Serial Receive 7
0x1a0	Serial Status 7

3.5.1 Interrupt (`dtminterrupt`, at 0x0)

Writes to this register set bits in the internal interrupt state.



Field	Description	Access	Reset
value	A write of value N sets bit N. The width of this register depends on the width of the internal interrupt register. (Eg. it's 5 bits wide if the internal internal register is 32 bits wide.)	W	0

3.5.2 Control (`dtmcontrol`, at 0x4)

31 24	23 21	20	19	18	17	16
abussize	0	access128	access64	access32	access16	access8
8	3	1	1	1	1	1

15 8	7 6	5	4	3 2	1	0
intbits	0	authenticated	authbusy	authtype	ndreset	fullreset
8	2	1	1	2	1	1

Field	Description	Access	Reset
abussize	Width of the address bus in bits. (This includes the extra address bit if it's required to access the debug bus.)	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
intbits	The width of the internal interrupt state is intbits $+ 1$.	R	Preset
authenticated	0 when authentication is required before using the DTM. 1 when the authentication check has passed. On components that don't implement authentication, this bit must be preset as 1.	R	Preset

Continued on next page

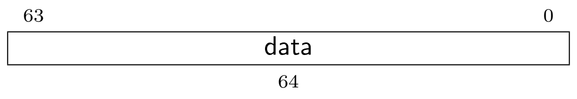
authbusy	While 1, writes to `authdata` may be ignored or may result in authentication failing. Authentication mechanisms that are slow (or intentionally delayed) must set this bit when they're not ready to process another write.	R	0
authtype	Defines the kind of authentication required to use this DTM. 0 means no authentication is required. 1 means a password is required. 2 means a challenge-response mechanism is in place. 3 is reserved for future use.	R	Preset
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.	W	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 DTM itself.	W	0

3.5.3 Authentication Data (`authdata`, at 0x8)

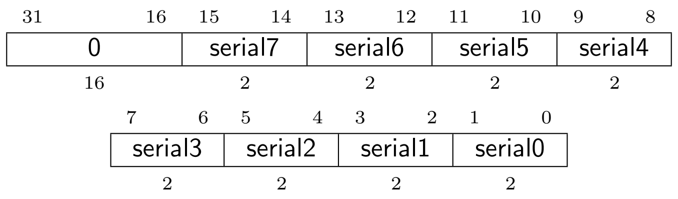
If authtype is 0, this register is not present.

If authtype is 1, writing a correct password to this register enables the DTM. The DTM is disabled either by writing an invalid password, or by resetting it. 0 must not be used as a password. Reading from the register returns 0.

If authtype is 2, things are a bit more complicated. Reading from the register reads the last challenge generated. Writing the correct response enables the DTM. The DTM is disabled either by writing an incorrect response, or by resetting it. Writing an incorrect response causes a new challenge to be generated. Depending on the implementation, there may not be a valid challenge until the first write to this register.



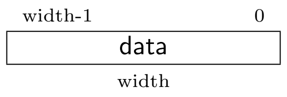
3.5.4 Serial Info (`serinfo`, at 0x10)



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	0 means serial interface 0 is not supported. 1 means serial interface 0 is supported and 32 bits wide. 2 means serial interface 0 is supported and 64 bits wide. 3 means serial interface 0 is supported and 128 bits wide.	R	Preset

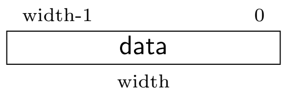
3.5.5 Serial Send 0 (`sersend0`, at 0x20)

Values written to this address are added to the send queue, unless the queue is already full.

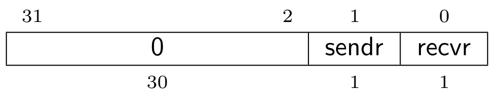


3.5.6 Serial Receive 0 (`serrecv0`, at 0x30)

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



3.5.7 Serial Status 0 (`serstat0`, at 0x40)



Field	Description	Access	Reset
sendr	Send ready. 1 when the send queue is not full. 0 otherwise.	R	1
recvr	Receive ready. 1 when the receive queue is not empty. 0 otherwise.	R	1

4 Device Tree Additions

The device tree is a data structure in ROM that all RISC-V platforms should have. It contains a variety of information about every component in the platform. (As of January 16, 2016 it is not yet part of any RISC-V spec.)

Every debuggable component should implement any applicable properties listed in Table 2 in the device tree.

Table 2: Component Device Tree Properties

`riscv,debug-address`	Address of this component's `ccsr` register.	Required
`riscv,auth-type`	Defines the kind of authentication required to use the debug logic on this component. 0 means no authentication is required. 1 means a password is required. 2 means a challenge-response mechanism is in place. Other values are reserved for future use.	Optional

If a Debug RAM is implemented, it must be listed in the device tree with its start address and size.

Each DTM must be listed in the device tree including its base address (address of `dtminterrupt`).

TODO: Update this section once there is a more general RISC-V device tree spec.

5 Component Debugging

Every component can expose arbitrary functionality as registers visible to the Debug Transport Module on the debug/system bus. This document only specifies in detail what RISC-V cores must expose.

There are a few generic features specified that any component can implement, and that even a debugger that knows nothing else about that component can use. The simplest is to freeze a component. Freezing is the simplest form of halting, effectively the same as gating the clock to a component. It should also be possible to just let the component run for a single clock cycle. Freezing a RISC-V core might allow a debugger to inspect the state of the pipeline.

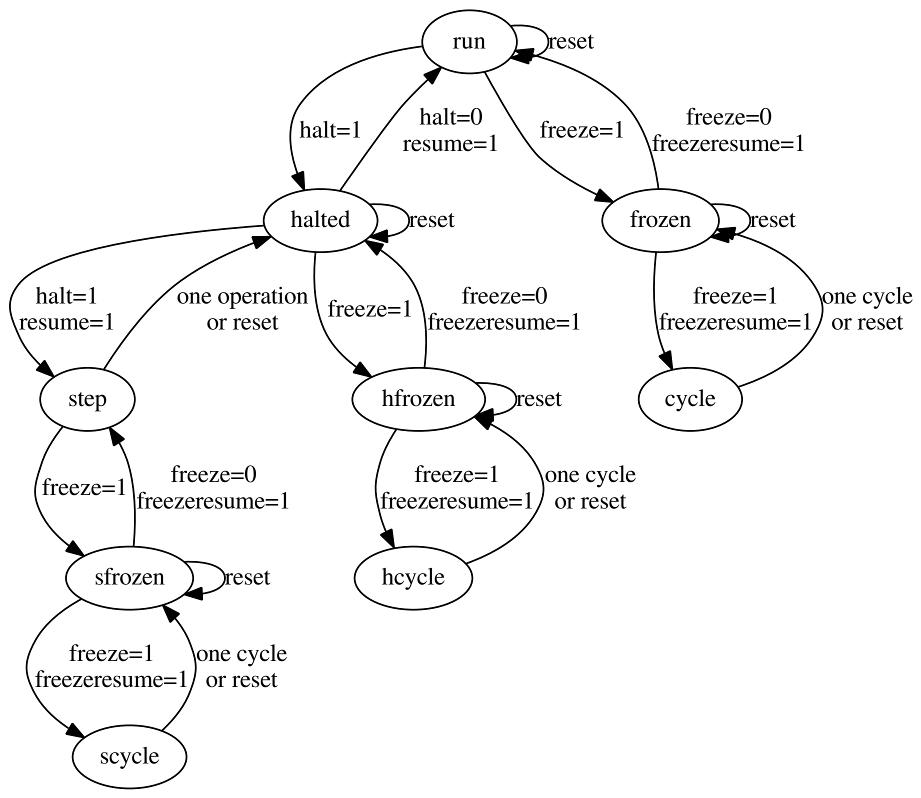


Figure 2: Component Run States

Components that support freezing must implement freeze and freezeresume in `ccsr`.

A more complicated alternative to freezing is halting. Halting usually happens on a boundary that is meaningful (eg. an instruction being fetched or completely executed), and may even put the component into a special Debug Mode. When a RISC-V core is halted, it's possible to let the core execute arbitrary instructions. It should also be possible to let the component take a meaningful step (eg. execute a single instruction). Components that support halting must implement halt and resume in `ccsr`. Their use is summarized in Figure 2.

Freezing and halting are orthogonal to each other, so a component may be both frozen and halted. In this case freezing could be used to debug halting.

In addition to freezing and halting, there are also 2 kinds of reset supported: The first is a traditional reset that resets the entire component. The second is a non-debug reset, which only resets that part of the component that are not part of the debug logic. The second is used so you can reset a component but remain halted/frozen.

5.1 Interrupt Tracking

When cdisable is clear, a component can send an interrupt to the DTM by writing interrupt to `dtminterrupt` as described in Section 3.2.

5.2 Security

Some components may contain intellectrual property that should not be disclosed, even to people who may debug other parts of the system. To help with this there are registers specified to support a simple authentication scheme that enables an authorized debugger to unlock the debug logic on a component.

This mechanism does not affect accesses over the system bus at all. It's up to the component designer to ensure that no IP is leaked over the system bus. This could be done by carefully designing the interface, or by only granting components that really need it the relevant system bus access. The latter option depends on the system bus to support that kind of functionality, and additionally requires that the components that have access be similarly secured.

5.3 Component Debug Registers

Table 3: Component Debug Registers

Address	Name
0x0	Component Control and Status
0x8	Authentication Data
0x10	DTM Interrupt Address

5.3.1 Component Control and Status (`ccsr`, at 0x0)

See Figure 2 for more information about how freeze, halt, and reset bits interact.

31	30	29 28	27	26	25
authenticated	authbusy	0	ndreset	fullreset	stopcycle
1	1	2	1	1	1

24	23	22	21	20	19
stoptime	frozen	freezesup	freeze	freezeresume	halted
1	1	1	1	1	1

18	17	16	15 8	7 0
haltsup	halt	resume	0	interrupt
1	1	1	8	8

Field	Description	Access	Reset
authenticated	0 when authentication is required before talking to the debug interface on this component. 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 `authdata` 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
ndreset	Every time this bit is written as 1, it triggers a reset of the non-debug logic in this component.	W	0
fullreset	Every time this bit is written as 1, it triggers a reset of the component including the debug logic.	W	0
stopcycle	Controls the behavior of any counters while the component is halted. When 1, counters are stopped when the component is halted/frozen. 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 halted. When 1, timers are stopped when the component is halted/frozen. 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
frozen	1 when the component is currently frozen.	R	0
freezesup	1 when freeze and freezeresume are supported.	R	Preset
freeze	When this bit is 1 and the component is not frozen, it becomes frozen. If this bit is 1 when the component is reset, the component should be frozen before it has performed any operations. If this bit is 1 when the freezeresume is written as 1, the component will only execute a single cycle before becoming frozen again. Setting this bit to 0 does not have an immediate effect.	R/W	0
freezeresume	If this bit is written as 1 while the component is frozen, the component becomes unfrozen.	W1	0
halted	1 when the component is currently halted.	R	0
haltsup	1 when halt and resume are supported.	R	Preset
halt	When this bit is 1 and the component is not halted, the component will enter Debug Mode. If this bit is 1 when the component is reset, the component must go directly to Debug Mode. If this bit is 1 when the component leaves Debug Mode, the component will perform one operation (eg. execute a single instruction, or perform a single cycle in a state machine) before re-entering Debug Mode. Setting this bit to 0 does not have an immediate effect.	R/W	0

Continued on next page

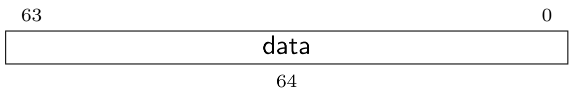
resume	If this bit is written as 1 while the component is in Debug Mode, the component leaves Debug Mode.	W1	0
interrupt	The value this component should use when writing to `dtminterrupt`. Depending on intbits in the DTM, the top bits of this field may be hardwired to 0. This value may be hardcoded if there are few components in the system. The debugger must read this value back after writing it to confirm whether it was written or not.	R/W	Preset

5.3.2 Authentication Data (`authdata`, at 0x8)

If authtype is 0, this register is ignored.

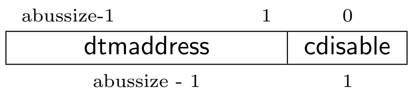
If authtype is 1, writing a correct password to this register enables the debug functionality. The functionality is disabled either by writing an invalid password, or by resetting it. 0 should not be used as a password. Reading from the register returns 0.

If authtype is 2, things are a bit more complicated. Reading from the register reads the last challenge generated. Writing the correct response enables the debug functionality. The functionality is disabled either by writing an incorrect response, or by resetting it. Writing an incorrect response causes a new challenge to be generated. Depending on the implementation, there may not be a valid challenge until the first write to this register.



5.3.3 DTM Interrupt Address (`cdtmaddress`, at 0x10)

On platforms with only a single DTM, this register may be read-only, pointed at that DTM.



Field	Description	Access	Reset
dtmaddress	Bits abussize-1:1 of the address of `dtminterrupt`. On some components this register will be less wide than abussize. In order for those components to send messages to the DTM, the DTM needs to be accessible to them using however many address bits (often 32) that actually fit in the register.	R/W	Preset
cdisable	When 1, the component won't write anything to the DTM. Set to 0 to enable the component writing to the DTM.	R/W	Preset

6 RISC-V Debug Module

6.1 Bus Interface

Each RISC-V debug module uses 17 bits of address space on the debug/system bus to provide access to debug features. The address space is divided up as show in Table 4.

6.2 Debug Mode

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

When entering Debug Mode:

1. The core should write its interrupt number to the DTM's `dtminterrupt`.

While in Debug Mode:

1. Regular program execution is suspended.
2. The contents of general purpose registers are accessible over the debug interface.
3. All operations happen in machine mode.
4. All interrupts are masked.
5. No hardware breakpoints are triggered.
6. Trace is disabled.

When leaving Debug Mode:

1. Regular program execution resumes at the address in `dpc`.

6.3 Debug Registers

6.3.1 Component Control and Status (`ccsr`, at 0x0)

This is the exact same register as is described in Section 5.3.1, but the description here is slightly more specific to RISC-V cores.

Table 4: Debug Bus Memory Space

Address	Value
0x0000 – 0x0100	Debug registers described in Section 6.3. These are always accessible.
0x0100 – 0x01c3	Debug registers described in Section 6.3. These are only be accessible when the core is halted.
0x01c4 – 0x01ff	Supported register map. This indicates which registers are directly accessible to a debugger. Each register gets a single bit, so each 32-bit word maps to 256 bytes of address space starting at 0x100. The LSB in each word maps to the first 128-bit word in its 256-byte space. Since the general purpose registers must always be supported, the word at 0xc4 always contains 0xffffffff. These may only be accessible if the core is halted.
0x0200 – 0x03ff	General purpose registers (`x0`–`x31`), each 16 bytes in size. These may only be accessible if the core is halted.
0x0400 – 0x05ff	Floating point registers (`f0`–`f31`), each 16 bytes in size. (Optional, even when floating point is supported by the core.) These may only be accessible if the core is halted.
0x0600 – 0x3fff	Reserved for future official extensions.
0x4000 – 0x7fff	Reserved for future debug standards.
0x8000 – 0xffff	Reserved for custom use to support platform-specific functions.
0x10000 – 0x1ffff	CSR registers. Each CSR register gets 16 bytes of space, to make it look like the registers are just laid out in a contiguous memory section (assuming they're all 128 bits in size). While halted, all registers must be accessible. When not halted, some or all registers may not be accessible.

Table 5: Debug Bus Registers

Address	Name
0x0	Component Control and Status
0x10	DTM Interrupt Address
0x20	Debug Control and Status
0x30	PC Sample
0x100	Stuff Instruction
0x110	Debug Jump
0x120	PC

31	30	29 28	27	26	25
authenticated	authbusy	0	ndreset	fullreset	stopcycle
1	1	2	1	1	1

24	23	22	21	20	19
stoptime	frozen	freezesup	freeze	freezeresume	halted
1	1	1	1	1	1

18	17	16	15 8	7 0
haltsup	halt	resume	0	interrupt
1	1	1	8	8

Field	Description	Access	Reset
authenticated	See Section 5.3.1.	R	Preset
authbusy	See Section 5.3.1.	R	0
ndreset	See Section 5.3.1.	W	0
fullreset	See Section 5.3.1.	W	0
stopcycle	Controls the behavior of any counters while the component is halted. This includes the counters read with `rdcycle` and `rdinstret`. When 1, counters are stopped when the component is halted/frozen. 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
stoptime	Controls the behavior of any timers while the component is halted. This includes the timer read with `rdcycle`. When 1, timers are stopped when the component is halted/frozen. 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

Continued on next page

frozen	See Section 5.3.1.	R	0
freezesup	See Section 5.3.1.	R	Preset
freeze	See Section 5.3.1.	R/W	0
freezeresume	See Section 5.3.1.	W1	0
halted	See Section 5.3.1.	R	0
haltsup	See Section 5.3.1.	R	Preset
halt	When this bit is 1 and the core is running, the core will enter Debug Mode, setting `dpc` to the address of the next instruction to be executed. If this bit is 1 when the core is reset, the core must go directly to Debug Mode. Do not execute any instructions. Do not collect $200. When entering Debug Mode, `dpc` is set to the reset vector. If this bit is 1 when the core leaves Debug Mode, the core will keep interrupts disabled and execute one instruction before re-entering Debug Mode. `dpc` is set to the address of the next instruction to be executed. Setting this bit to 0 does not have an immediate effect.	R/W	0

Continued on next page

resume	If this bit is written as 1 while the core is in Debug Mode, the core leaves Debug Mode and resumes execution at `dpc`.	W1	0
interrupt	See Section 5.3.1.	R/W	Preset

6.3.2 DTM Interrupt Address (`cdtmaddress`, at 0x10)

See Section 5.3.3.

6.3.3 Debug Control and Status (`dcsr`, at 0x20)

31	30	29 28	27 16	15 8	7
pcsample	haltinterrupt	xdebugver	hwbpcount	0	debug
1	1	2	12	8	1

6 4	3	2	1	0
cause	ebreakm	ebreakh	ebreaks	ebreaku
3	1	1	1	1

Field	Description	Access	Reset
pcsample	1 when the `pcsample` is implemented. 0 otherwise.	R	Preset
haltinterrupt	1 when the core will write interrupt to `dtminterrupt` when it halts. 0 otherwise.	R	Preset
xdebugver	0 means there is no external hardware debug support. 1 means hardware debug support exists as it is described in this document. Other values are reserved for future standards.	R	Preset
hwbpcount	Number of hardware breakpoints this core supports.	R	Preset
debug	1 when the core is in Debug Mode. 0 otherwise.	R	0
cause	Explains why Debug Mode was entered. 0 means the core is not in Debug Mode. 1 means a software breakpoint was hit. 2 means a hardware breakpoint was hit. 3 means halt in `ccsr` was set. 4 means the core single stepped.	R	0

Continued on next page

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

6.3.4 PC Sample (`pcsample`, at 0x30)

This optional register contains a recent value of `pc`. It can be repeatedly polled by a debugger to get some idea of where execution spends most of its time.

XLEN-1 0
pc
XLEN

6.3.5 Stuff Instruction (`dstuff`, at 0x100)

While halted, a write to this register will result in the 32-bit instruction written being executed exactly once. (If the debugger needs to execute instructions that aren't 32 bits wide, it should use `djump`.) Stuffing instructions that change the PC (eg. branch and jump) leads to undefined behavior.

This may interfere with the value in `dpc`, so the debugger should save it first.

The instruction executed must not read or write `s8`– `s11`. They are reserved for debug logic in the core.

31 0
instruction
32

6.3.6 Debug Jump (`djump`, at 0x110)

While halted, a write to this register will result in a jump to the address written. The core remains in Debug Mode, but is no longer halted. When the core encounters an `ebreak` instruction it halts again. If haltinterrupt is set, the core must write interrupt to `dtminterrupt` when it halts again.

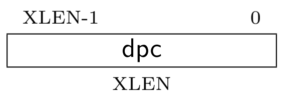
This may interfere with the value in `dpc`, so the debugger should save it first.

If the code executed wants to use `s8`– `s11`, it has to save and restore them. They are reserved for debug logic in the core.

XLEN-1 0
address
XLEN

6.3.7 PC (`dpc`, at 0x120)

After entering Debug Mode, this register contains the PC of the next instruction to be executed when Debug Mode is left.



7 Hardware Breakpoint Module

Hardware breakpoints 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 hardware debug module present, so the Hardware Breakpoint Module is broken out as a separate piece that can be implemented separately.

A core may support up to 4095 hardware breakpoints, although 4 is a more typical number. Each hardware breakpoint may support a variety of features. A debugger can build a list of all hardware breakpoints and their features by selecting each one in turn using `bpselect`, and then querying `bpcontrol`.

TODO: Clearly spec behavior when both M mode and Debug Mode want to use the same hardware breakpoint.

7.1 Hardware Breakpoint Registers

TODO: Is it worth specifying some kind of state machine for triggering, to get functionality more on par with a logic analyzer? Would you implement that?

These breakpoint registers are only accessible in machine mode, to prevent untrusted user code from causing entry into Debug Mode without the OS's permission.

Table 6: Hardware Breakpoint Registers

Address	Name
0x780	Breakpoint Select
0x781	Breakpoint Control
0x782	Breakpoint Low Address
0x783	Breakpoint High Address
0x784	Breakpoint Low Data
0x785	Breakpoint High Data

7.1.1 Breakpoint Select (`bpselect`, at 0x780)

Since CSR space is limited, and each hardware breakpoint may have a lot of configuration options, this register determines which hardware breakpoint is accessible through the other breakpoint registers.

31 12	11 0
0	bp
20	12

Field	Description	Access	Reset
bp	Select this hardware breakpoint.	R/W	0

7.1.2 Breakpoint Control (`bpcontrol`, at 0x781)

This register contains information about what the selected breakpoint supports, and allows any of its features to be enabled.

Breakpoint match logic is as follows:

$$
\begin{aligned}
&\mathsf{amatch} = (\overline{\mathsf{aen}} \wedge \overline{\mathsf{arangeen}} \wedge \overline{\mathsf{amask}}) \vee \\
&\quad (\mathsf{aen} \wedge address = \mathtt{bploaddr}) \vee \\
&\quad (\mathsf{arangeen} \wedge address \geq \mathtt{bploaddr} \wedge address < \mathtt{bphiaddr}) \vee \\
&\quad (\mathsf{amask} \wedge (address \& \mathtt{bphiaddr}) = \mathtt{bploaddr}) \\
&\mathsf{dmatch} = (\overline{\mathsf{den}} \wedge \overline{\mathsf{drangeen}} \wedge \overline{\mathsf{dmask}}) \vee \\
&\quad (\mathsf{den} \wedge data = \mathtt{bplodata}) \vee \\
&\quad (\mathsf{drangeen} \wedge data \geq \mathtt{bplodata} \wedge data < \mathtt{bphidata}) \vee \\
&\quad (\mathsf{dmask} \wedge (data \& \mathtt{bphidata}) = \mathtt{bplodata}) \\
&\mathsf{omatch} = (\mathsf{loaden} \wedge access\_is\_load) \vee \\
&\quad (\mathsf{storeen} \wedge access\_is\_store) \vee \\
&\quad (\mathsf{execen} \wedge access\_is\_exec) \\
&\mathsf{match} = \mathsf{amatch} \wedge \mathsf{dmatch} \wedge \mathsf{omatch}
\end{aligned}
$$

31 27	26	25	24	23	22	21	20	19
0	loadsup	storesup	execsup	asup	arangesup	amasksup	0	dsup
5	1	1	1	1	1	1	1	1

18	17	16	15	14 12	11	10	9	8
drangesup	dmasksup	0	matched	action	0	loaden	storeen	execen
1	1	1	1	3	1	1	1	1

7	6	5	4	3	2	1	0
aen	arangeen	amasken	0	den	drangeen	dmasken	0
1	1	1	1	1	1	1	1

Field	Description	Access	Reset
loadsup	This breakpoint supports matching on data load.	R	Preset
storesup	This breakpoint supports matching on data store.	R	Preset
execsup	This breakpoint supports matching on instruction execution.	R	Preset
asup	This breakpoint supports exact address matches. (It would be a strange breakpoint that doesn't.)	R	Preset
arangesup	This breakpoint supports range address matches.	R	Preset
amasksup	This breakpoint supports masked address matches.	R	Preset
dsup	This breakpoint supports exact data matches.	R	Preset
drangesup	This breakpoint supports range data matches.	R	Preset
dmasksup	This breakpoint supports masked data matches.	R	Preset
matched	Set to 1 when this hardware breakpoint matched. The debugger is responsible for clearing this bit once it has seen it's set.	R/W	0
action	Determines what happens when this breakpoint matches. 0 means nothing happens. 1 means cause a debug exception. 2 means enter Debug Mode. 3 means start tracing. 4 means stop tracing. 5 means emit trace data for this match. (If it's a data access match, emit appropriate Load/Store Address/Data. If it's an instruction execution, emit its PC.) Other values are reserved for future use.	R/W	0
			Continued on next page

loaden	Set to enable this breakpoint for data loads.	R/W	0
storeen	Set to enable this breakpoint for data stores.	R/W	0
execen	Set to enable this breakpoint for instruction execution. When an execution breakpoint is hit on an address match, the core enters Debug Mode immediately before the instruction is executed.	R/W	0
aen	Set to cause this breakpoint to match when *address* equals `bploaddr`.	R/W	0
arangeen	Set to cause this breakpoint to match when `bploaddr` $<=$ *address* $<$ `bphiaddr`.	R/W	0
amasken	Set to cause this breakpoint to match when *address*&`bphiaddr` $=$ `bploaddr`.	R/W	0
den	Set to cause this breakpoint to match when *data* equals `bplodata`.	R/W	0
drangeen	Set to cause this breakpoint to match when `bplodata` $<=$ *data* $<$ `bphidata`.	R/W	0
dmasken	Set to cause this breakpoint to match when *data*&`bphidata` $=$ `bplodata`.	R/W	0

7.1.3 Breakpoint Low Address (`bploaddr`, at 0x782)

Used for exact match or lower bound (inclusive) of the address match for this breakpoint.

XLEN-1 0
loaddress
XLEN

7.1.4 Breakpoint High Address (`bphiaddr`, at 0x783)

Used for upper bound (inclusive) of the address match for this breakpoint, or as address mask.

XLEN-1 0
hiaddress
XLEN

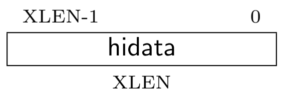
7.1.5 Breakpoint Low Data (`bplodata`, at 0x784)

Used for exact match or lower bound (inclusive) of the data match for this breakpoint.

XLEN-1 0
lodata
XLEN

7.1.6 Breakpoint High Data (`bphidata`, at 0x785)

Used for upper bound (inclusive) of the data match for this breakpoint, or as data mask.



8 Trace Module

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.

This part of the spec is functional, but could certainly be improved a lot.

8.1 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 7.

Is it an improvement to add a count after Branch Taken/Not Taken headers?

Table 7: 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	Reserved	Reserved for future standards.
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.

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.

8.2 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 8.

Table 8: 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.

8.3 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 must attempt to send one full value for each of these (assuming they're enabled) soon after each synchronization point.

8.4 Trace Registers

Table 9: Trace Registers

Address	Name
0x788	Trace
0x789	Trace Buffer Start
0x78a	Trace Buffer End
0x78b	Trace Buffer Write

8.4.1 Trace (`trace`, at 0x788)

31 25	24	23	22	21	20
0	wrapped	emittimestamp	emitstoredata	emitloaddata	emitstoreaddr
7	1	1	1	1	1

19	18	17	16	15 10	9 8
emitloadaddr	emitpriv	emitbranch	emitpc	0	fullaction
1	1	1	1	6	2

7 6	5 4	3	2	1	0
0	destination	0	stall	discard	supported
2	2	1	1	1	1

Field	Description	Access	Reset

Continued on next page

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 to trace to a dedicated on-core RAM (which is not further defined in this spec). 1 to trace to RAM on the system bus. Both those options may slow down execution (eg. because of system bus contention). 2 to send trace data to a dedicated off-chip interface (which is not defined in this spec). This does not affect execution speed. 3 is reserved for future use.	R/W	Preset
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

8.4.2 Trace Buffer Start (`tbufstart`, at 0x789)

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

XLEN-1 0
address
XLEN

8.4.3 Trace Buffer End (`tbufend`, at 0x78a)

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

XLEN-1 0
address
XLEN

8.4.4 Trace Buffer Write (`tbufwrite`, at 0x78b)

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

XLEN-1 0
address
XLEN

9 JTAG Debug Transport Agent

This Debug Transport Agent 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).

9.1 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. It is the third case that we're primarily concerned with 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.

9.2 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 10. 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.

Table 10: 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
00100	INIT_SETUP_CLAMP	JTAG recommends 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_RUN	JTAG recommends this instruction
01110	Unused (BYPASS)	Reserved for future JTAG
01111	Unused (BYPASS)	Reserved for future JTAG
10000	DTM Control	DTM Control
10001	DTM Authentication Data	DTM Authentication
10010	JTAG Bus Control	For bus access
10011	JTAG Bus Address	For bus access
10100	JTAG Bus Data	For bus access
10101	JTAG Status	For interrupts/serial
10110	JTAG Status Control	For interrupts/serial
10111	JTAG Serial Data	For serial
11000	Reserved (BYPASS)	Reserved for future RISC-V debugging
11001	Reserved (BYPASS)	Reserved for future RISC-V debugging
11010	Reserved (BYPASS)	Reserved for future RISC-V debugging
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

9.2.1 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.

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

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

9.2.2 DTM Control (`dtmcontrol`, at 10000)

48 32	31 0
serinfo	dtmcontrol
17	32

Field	Description	Access	Reset
serinfo	Contains the lower 16 bits of `serinfo` as described in Section 3.5.4.	R	Preset
dtmcontrol	Contains `dtmcontrol` as described in Section 3.5.2.	R/W	Preset

9.2.3 DTM Authentication Data (`authdata`, at 10001)

This register is the JTAG view of the DTM register described in Section 3.5.3. It only exists if authtype isn't 0.

9.2.4 JTAG Bus Control (`jbusc`, at 10010)

Unlike the other registers, it's possible to write this one while the JTAG bus master is busy. If the debugger chooses to do so, it should write error as 1 so it won't disable an error that occurs during the scan.

12 11	10 6	5 3	2 0
error	dbits	abits	size
2	5	3	3

Table 11: JTAG Access Size

Encoding	size	loabits
0	8	0
1	16	1
2	32	2
3	64	3
4	128	4
other	reserved	reserved

Table 12: JTAG Address Bits

Encoding	hiabits
0	$\min(7, \mathsf{abussize} - 1)$
1	$\min(11, \mathsf{abussize} - 1)$
2	$\min(15, \mathsf{abussize} - 1)$
3	$\min(23, \mathsf{abussize} - 1)$
4	$\min(31, \mathsf{abussize} - 1)$
5	$\min(63, \mathsf{abussize} - 1)$
6	$\min(127, \mathsf{abussize} - 1)$
other	reserved

Table 13: Serial Ports in `jstatus`

Encoding	hiabits
0	$\min(7, \mathsf{abussize} - 1)$
1	$\min(11, \mathsf{abussize} - 1)$
2	$\min(15, \mathsf{abussize} - 1)$
3	$\min(23, \mathsf{abussize} - 1)$
4	$\min(31, \mathsf{abussize} - 1)$
5	$\min(63, \mathsf{abussize} - 1)$
6	$\min(127, \mathsf{abussize} - 1)$
other	reserved

Field	Description	Access	Reset
error	0 means no error has been seen. 1 means a timeout error has been seen. 2 means some other bus error has been seen. The DTM updates this field with an error when one occurs. It is cleared when the debugger writes 0.	R/W0	0
dbits	Set hidbits to (dbits + 1) ∗ 4 − 1. If hidbits is set to be larger than size, the extra bits scanned into data will be ignored.	R/W	7
abits	Set hiabits per Table 13.	R/W	5
size	Set size and loabits per Table 11.	R/W	2

9.2.5 JTAG Bus Address (`jaddress`, at 10011)

hiabits-loabits+2 … 2	1	0
update	autoincrement	read/busy
hiabits - loabits + 1	1	1

Field	Description	Access	Reset
update	Update address[hiabits:loabits] with the value in update.	R/W	0
autoincrement	When set, increment address by size/8 after every scan of `jdata`.	R/W	0
read/busy	Set this bit to perform a read at the updated address. Read this bit to determine whether the JTAG bus master is busy. If this bit reads as 1 then writes to this register are ignored.	R/W	0

9.2.6 JTAG Bus Data (`jdata`, at 10100)

hidbits+2 … 2	1	0
data	write/valid	read/busy
hidbits + 1	1	1

Field	Description	Access	Reset
data	If write/valid reads as 1, this contains the data from the successful read. Data written to this register will be written to address if write/valid is written as 1. If size is larger than hidbits − 1 then data will be sign extended before being written. If size is smaller than hidbits − 1 then the extra data bits are ignored.	R/W	0
write/valid	Set this bit to write data to the current address. Read this bit to determine whether the register contains data from a successful read.	R/W	0
read/busy	Set this bit to perform a read at the (possibly post-incremented) address. Read this bit to determine whether the JTAG bus master is busy. If this bit reads as 1 then writes to this register are ignored.	R/W	0

9.2.7 JTAG Status (`jstatus`, at 10101)

serbits+intbits-1 intbits	intbits-1 0
serial	interrupt
serbits	intbits

Field	Description	Access	Reset
serial	Bit 0 is 1 when serial port 0 is ready for the debugger to send data to it. Bit 1 is 1 when serial port 0 has data in the queue for the debugger to read. Bits 2 and 3 do the same for serial port 1, and so on.	R	. . . 01
interrupt	Contains the intbits lower bits of the internal interrupt state. Scanning this register clears the entire internal interrupt state.	R	0

Continued on next page

9.2.8 JTAG Status Control (`jstatc`, at 10110)

6 5	4 3	2 0
serscan	intscan	serselect
2	2	3

Field	Description	Access	Reset
serscan	Select the number of serial ports that show up in `jstatus`. serbits is $4 * \text{serscan} + 4$. (There are 2 bits per serial port.)	R/W	0
intscan	Select the number of interrupt status bits that show up in `jstatus`. intbits is $4 * 2^{\text{intscan}}$.	R/W	0
serselect	Select which serial port `jserial` accesses.	R/W	0

9.2.9 JTAG Serial Data (`jserial`, at 10111)

width+1 2	1	0
data	write/valid	busy
width	1	1

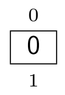
Field	Description	Access	Reset
data	If write/valid reads as 1, this contains the oldest value in the core-to-debugger queue. After this scan that value will be removed from the queue. Data written to this field will be written to the debugger-to-core queue if write/valid is written as 1. The width of this field depends on the width of the underlying serial port. It can be discovered by reading serinfo in `dtmcontrol`.	R/W	0

Continued on next page

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
busy	Read this bit to determine whether the core-to-debugger queue is full. If this bit reads as 1 then writes to this register are ignored.	R/W	0

9.2.10 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.

To keep the text readable, these examples assume that the debugger is slower than the core/DTM so never has to wait. A real implementation should always check read/busy, write/valid, etc.

A.1 Setup

The first thing a debugger should do when connecting to a RISC-V platform is to read and parse the device tree. The device tree should be located at a known address. If not, then the user will have to tell the debugger where it is located.

Next the debugger should identify each component it wants to debug. Each of those components need to be pointed at the DTM the debugger is using by writing `cdtmaddress`, and then assigned an interrupt by writing interrupt in `ccsr`. Components that won't be debugged could all be put on a single interrupt, leaving the remaining interrupts for components that are debugged. If that's not enough, interrupts will have to be shared.

A.2 Reading Memory

To read memory, first set up the memory access size (with corresponding hidbits) and abits. (If they already contain the correct value, this scan can be skipped.)

Next, scan `jaddress` filling out the address to be read, and setting read/busy. If read/busy is already set then the DTM is busy and this scan will have to be repeated until it's no longer busy. The memory access will start when the TAP is in the Update-DR state.

Finally scan `jdata`, which will contain the data that was read, assuming write/valid is set. If it is not set the scan will have to be repeated. If write/valid is clear and read/busy is also clear, that indicates there was some kind of error. The debugger should scan `jbusc` to find out what the error was.

To immediately read the same address again, set read/busy in the `jdata` scan.

To immediately read the next address, set autoincrement in the `jaddress` scan and read/busy in the `jdata` scan.

A.3 Writing Memory

To write memory, first set up the memory access size (with corresponding hidbits) and abits. (If they already contain the correct value, this scan can be skipped.)

Next, scan `jaddress` filling out the address to be written. If read/busy is set then the DTM is busy and this scan will have to be repeated until it's no longer busy.

Finally scan `jdata` with the data that should be written, and set write/valid. read/busy should not be set at this point since it was already cleared in the previous step. The memory access will start when the TAP is in the Update-DR state.

The debugger could poll either `jdata` or `jaddress` for read/busy to become 0 but typically the debugger will just call the write complete without waiting for that. If it's really paranoid it could check error in `jbusc` after discovering that the DTM is no longer busy.

To immediately write the same address again, simply scan `jdata` again.

To immediately write the next address, set autoincrement in the original `jaddress` scan and scan `jdata` again.

A.4 Halt

To halt a core, the debugger sets halt in `ccsr`. It can then check halted in `ccsr` to discover when the core actually halts.

A.5 Reading Registers

When halted and not running code through use of `djump`, `x0`– `x31` can be read directly from the Debug Bus Interface. Other registers are directly accessible if their corresponding bit is set in the supported register map.

For registers that are not directly accessible, an instruction will have to be executed to read it. Eg. to read `f1` first write `fmv.x.s x8, f1` to `dstuff` and then read the value of `x8` directly.

A.6 Writing Registers

When halted and not running code through use of `djump`, `x0`– `x31` can be written directly from the Debug Bus Interface. Other registers are directly accessible if their corresponding bit is set in the supported register map.

For registers that are not directly accessible writing is a 2-step process. First directly write the new value to a general purpose register (eg. `x8`). Then stuff an instruction to move the value to the appropriate register, eg. `fmv.s.x f1, x8` to read `f1`.

A.7 Custom Debug Programs

Some operations can benefit a lot from executing a small program instead of feeding instructions one at a time. Zeroing memory is a good example of this. (Depending on what a program expects, certain blocks of RAM may need to be zeroed before it is executed.)

To do this efficiently, the debugger needs a bit of RAM. This RAM can be dedicated in the platform and documented in the Device Tree, or be simply something that the user told the debugger. The debugger can write a simple program to this RAM. Eg.:

```
loop:
        sw      zero, 0(x9)
        addi    x9, x9, 4
        bne     x9, x8, loop
        ebreak
```

Then it saves the contents of `x8` and `x9` before writing the start address to `x9`, the end address to `x8`. To start execution it writes the address of the program to `djump`. The core will stay in Debug Mode but jump to the start of the code. When it encounters the `ebreak` instruction it halts again. Before the core is resumed, the debugger must restore `x8` and `x9`.

Depending on the implementation, `dpc` may be changed by doing this. The debugger must save it before writing to `djump` and restore it later.

A.8 Accessing Memory Through the Core

Typically to access memory you'd use the DTM's feature to do so directly, but sometimes some memory is only accessible from the processor itself and not available on the system bus. In that case it's necessary for the core to perform the bus access.

A.8.1 Read

Write the address to `a0`, then stuff `lw a0, 0(a0)`. Now read `a0`.

Like writing, reading a block could be done more efficiently by using a Debug Program. For instance:

```
# a0 contains the address of the serial send register.
# a1 contains the first address to read from.
# a2 contains the last address to read from.
loop:
        lw      t0, 0x10(a0)    # Load status.
        andi    t0, t0, SERSTAT_SENDR_MASK
        beqz    t0, loop
        lw      t0, 0(a1)       # Read word from RAM.
        sw      t0, 8(a0)       # Send word to serial interface.
        addi    a1, a1, 4       # Increment write pointer.
        bne     a1, a2, loop
        ebreak
```

A.8.2 Write

Write the address to `a0`, the value to `a1`, then stuff `sw a1, 0(a0)`.

If more than a few writes are needed, a more efficient option would be to write a small Debug Program and use a DTM serial port to feed it data. For instance to write a block of memory:

```
# a0 contains the address of the serial send register.
# a1 contains the first address to write to.
# a2 contains the last address to write to.
loop:
        lw      t0, 0x10(a0)    # Load status.
        andi    t0, t0, SERSTAT_RECVR_MASK
        beqz    t0, loop
        lw      t0, 8(a0)       # Read word from serial interface.
        sw      t0, 0(a1)       # Write word to RAM.
        addi    a1, a1, 4       # Increment write pointer.
        bne     a1, a2, loop
        ebreak
```

The debugger needs to save and set up the appropriate registers before executing this loop. Then it can write `djump` and start writing data to the chosen serial port.

A.9 Running

To let the core run once it's halted, the debugger should restore any registers it has modified, and then clear halt while setting resume in `ccsr`.

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 biggest difference to the user is that in the former case it is likely that a pending interrupt will be completely serviced during the "single" step (unless the debugger takes additional action to disable interrupts), and there's a chance that something goes wrong (eg. memory is changed by another core or the debugger incorrectly predicts the next PC).

Using the hardware single step feature is almost the same as regular running. The debugger just sets both halt and resume in `ccsr`. 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.

B Debug ROM Implementation

One unorthodox implementation of the RISC-V debug module is to add a bare minimum of hardware to each core, and jump to a debug ROM when the core is "halted." In this implementation Debug Mode is simply another privileged mode, like M mode. This has the benefits that a debug exception is similar to other exceptions, and that the state machine logic (which is now encoded in the ROM) is shared among all cores in the system.

B.1 Hardware Changes

Hardware needs to implement bus accesses from 0x0 – 0x7f. Everything else can be handled by ROM. (Hardware may choose to implement 0x8000–0xffff as well so CSRs can be accessed while the core is running.)

When `ebreak` causes a debug exception, the PC jumps to `entry` in Debug ROM. When `ebreak` is executed when already in debug mode (but not halted), the PC jumps to `reentry` in Debug ROM.

When `eret` is executed in Debug Mode, it restores `pc` from `dpc` and causes the core to leave Debug Mode.

To leave Debug Mode when resume is set, the hardware changes the PC to `exit` in Debug ROM.

halt in `ccsr` is set whenever the PC is in Debug ROM.

B.2 Debug ROM Registers

These are extra CSRs required for this sample Debug ROM implementation.

Table 14: Control and Status Registers

Address	Name
0x770	Bus State
0x771	Bus Address
0x772	Bus Data
0x773	Debug PC
0x774	Debug Scratch 0
0x775	Debug Scratch 1
0x776	Debug Scratch 2
0x777	Debug RAM Address
0x778	Component Control and Status
0x779	DTM Interrupt Address

B.2.1 Bus State (`busstate`, at 0x770)

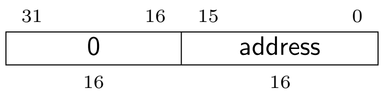
Allows code running on the core to handle debug/system bus accesses to the core.

31 … 3	2	1	0
0	complete	write	read
29	1	1	1

Field	Description	Access	Reset
			Continued on next page

complete	Write 1 to this register to complete the currently pending access.	W1	0
write	1 when a write access is pending. 0 otherwise.	R	0
read	1 when a read access is pending. 0 otherwise.	R	0

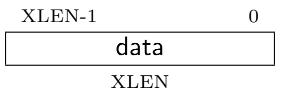
B.2.2 Bus Address (`busaddress`, at 0x771)



Field	Description	Access	Reset
address	When an access is pending, contains the address of that access.	R	0

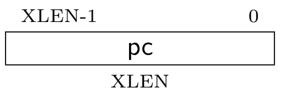
B.2.3 Bus Data (`busdata`, at 0x772)

When a write access is pending, contains the data being written. When a read access is pending, the core should write the result of that read to this register before setting complete in `busstate`.



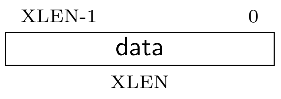
B.2.4 Debug PC (`dpc`, at 0x773)

When entering Debug Mode, the current PC is copied to this register. When leaving Debug Mode, execution resumes at the value in this register.



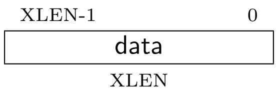
B.2.5 Debug Scratch 0 (`dscratch0`, at 0x774)

Scratch register where Debug ROM can save state while in Debug Mode.



B.2.6 Debug Scratch 1 (`dscratch1`, at 0x775)

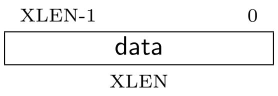
Scratch register where Debug ROM can save state while in Debug Mode.



B.2.7 Debug Scratch 2 (`dscratch2`, at 0x776)

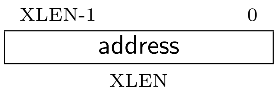
Scratch register where Debug ROM can save state while in Debug Mode.

TODO: Think about how to save state to the debugger instead of to scratch registers.



B.2.8 Debug RAM Address (`dramaddr`, at 0x777)

This register contains the address where there are 8 bytes of RAM for the Debug ROM to write to when it needs to execute an arbitrary instruction. These 8 bytes will be clobbered. They can be shared among multiple cores because the ROM code will only use the RAM during a bus access.



B.2.9 Component Control and Status (`ccsr`, at 0x778)

CSR view of `ccsr` defined in Section 5.3.1.

B.2.10 DTM Interrupt Address (`cdtmaddress`, at 0x779)

CSR view of `cdtmaddress` defined in Section 5.3.3.

B.3 Debug ROM Source

```
# This code should be functional. Doesn't have to be optimal.
# I'm writing it to prove that it can be done.

# TODO: Update these constants once they're finalized in the doc.

#define BUSSTATE                0x770
#define BUSSTATE_READ           0x1
#define BUSSTATE_WRITE          0x2
#define BUSSTATE_COMPLETE       0x4
```

```
#define BUSADDRESS              0x771
#define BUSDATA                 0x772

#define DPC                     0x773

#define SCRATCH0                0x774
#define SCRATCH1                0x775
#define SCRATCH2                0x776

#define DRAMADDR                0x777
#define CCSR                    0x778
#define CCSR_INTERRUPT_MASK     0xff

#define CDTMADDRESS             0x779

# TODO: Once address translation is specced, this code might need to be
# updated.

        .section        .debug_rom

        .balign 0x1000
entry:  j       _entry
reentry:
        j       _reentry
exit:   j       _exit

_entry:
        csrw    SCRATCH0, s8
        csrw    SCRATCH1, s9
        csrw    SCRATCH2, s10

_reentry:
        # Send an interrupt.
        csrr    s8, CCSR
        andi    s8, s8, CCSR_INTERRUPT_MASK
        csrr    s9, CDTMADDRESS
        sw      s8, 0(s9)

main:
        csrr    s9, BUSSTATE
        andi    s9, s9, BUSSTATE_READ | BUSSTATE_WRITE
        beqz    s9, main

        # Either read or write is happening.
        csrr    s8, BUSADDRESS         # Read 16 bits of address.
        andi    s9, s9, BUSSTATE_READ
        beqz    s9, write

# Handle a bus read.
read:
        li      s10, 0x90
        bne     s8, s10, rskip0
# Read PC
        csrr    s9, DPC
        j       access_done

rskip0:
        li      s10, 0xc4
        bne     s8, s10, rskip1
# Read whether GPRs are accessible. (Of course they are.)
        li      s9, 0xffffffff
        j       access_done

rskip1:
        li      s10, 0x1c0
        bne     s8, s10, rskip2
# Read s8
        csrr    s9, SCRATCH0
        j       access_done

rskip2:
        li      s10, 0x1c8
        bne     s8, s10, rskip3
# Read s9
        csrr    s9, SCRATCH1
        j       access_done

rskip3:
        li      s10, 0x1d0
        bne     s8, s10, rskip5
# Read s10
        csrr    s9, SCRATCH2
        j       access_done

rskip5:
        li      s10, 0x200
        bge     s10, s8, rskip6
        li      s10, 0x100
        blt     s8, s10, rskip6
# Read from GPR (but not s8--s10).
        # Generate "mv s9, <from>"
        # GPR number is in bits 7:3 of s8, and needs to be in bits 19:15 of
        # the instruction.
        andi    s8, s8, 0xf8
        sll     s10, s8, 19-7
        li      s8, 0xc93
        or      s8, s8, s10
        j       execute_instruction

rskip6:
        li      s10, 0x8000
        blt     s8, s10, rskip7
# Read from CSR.
        # Generate "csrr s9, <from>"
        # CSR number is in bits 14:3 of s8, and needs to be in bits 31:20 of
        # the instruction.
        sll     s10, s8, 31-14
        li      s8, 0x2c73
        or      s8, s8, s10
        j       execute_instruction

rskip7:
        li      s9, 0           # default to read 0
        j       access_done

# Handle a bus write.
write:
        csrr    s9, BUSDATA
        li      s10, 0x80
        bne     s8, s10, wskip1

# stuff instruction
        mv      s8, s9
        j       execute_instruction

wskip1:
        li      s10, 0x88
        bne     s8, s10, wskip2
# jump to address
        li      s8, BUSSTATE_COMPLETE
        csrw    BUSSTATE, s8
        jr      s9
        # At the end of the code we jump to must be an ebreak, which gets us
        # back to reentry.

wskip2:
        li      s10, 0x90
        bne     s8, s10, wskip3
# Write PC
        csrw    DPC, s9
        j       access_done

wskip3:
        li      s10, 0x1c0
        bne     s8, s10, wskip4
# Write s8
        csrw    SCRATCH0, s9
        j       access_done

wskip4:
        li      s10, 0x1c8
        bne     s8, s10, wskip5
# Write s9
        csrw    SCRATCH1, s9
        j       access_done

wskip5:
        li      s10, 0x1d0
        bne     s8, s10, wskip7
# Write s10
        csrw    SCRATCH2, s9
        j       access_done

wskip7:
        li      s10, 0x200
        bge     s10, s8, wskip8
        li      s10, 0x100
        blt     s8, s10, wskip8
# Write to GPR (but not s8--s10).
        # Generate "mv <to>, s9"
        # GPR number is in bits 7:3 of s8, and needs to be in bits 11:7 of
        # the instruction.
        andi    s8, s8, 0xf8
        sll     s10, s8, 11-7
        li      s8, 0xc8013
        or      s8, s8, s10
        j       execute_instruction

wskip8:
        li      s10, 0x8000
        blt     s8, s10, wskip9
# Write to CSR.
        # Generate "csrw <to>, s9"
        # CSR number is in bits 14:3 of s8, and needs to be in bits 31:20 of
        # the instruction.
        sll     s10, s8, 31-14
        li      s8, 0xc9073
        or      s8, s8, s10
        j       execute_instruction

wskip9:

access_done:
        # Always write BUSDATA. We need it for reads. Doesn't hurt for
        # writes.
        csrw    BUSDATA, s9
        li      s9, BUSSTATE_COMPLETE
        csrw    BUSSTATE, s9
        j       main


execute_instruction:
        # Take the instruction in s8.
        # Take the value the instruction may operate on in s9.
        # Clobber s8 and s10.
        # Jump to access_done once the instruction is executed.
        csrr    s10, DRAMADDR
        sw      s8, 0(s10)
        li      s8, 0x000c0067  # jr s8
        sw      s8, 4(s10)
        la      s8, access_done
        fence.i
        jr      s10

_exit:
        csrr    s8, SCRATCH0
        csrr    s9, SCRATCH1
        csrr    s10, SCRATCH2
        eret    # TODO: dret?
```