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 4

2 About This Document 4
   2.1 Structure ................................................. 4
   2.2 Terminology .............................................. 4
   2.3 Register Definitions .................................... 4
      2.3.1 Long Name (shortname, at 0x123) ................. 5

3 Background 5

4 Supported Features 5

5 System Overview 6

6 Debug Transport Module 8
   6.1 System Bus Access ....................................... 8
   6.2 Interrupt Tracking ....................................... 9
   6.3 Serial Ports ............................................. 9
   6.4 Security .................................................. 9
   6.5 Debug Transport Module Registers ..................... 9
      6.5.1 Interrupt (dtminterrupt, at 0x0) .................... 9
      6.5.2 Control (dtmcontrol, at 0x4) ....................... 11
      6.5.3 Authentication Data (authdata, at 0x8) ............ 12
      6.5.4 Serial Info (serinfo, at 0x10) ..................... 13
      6.5.5 Serial Send 0 (sersend0, at 0x20) ................ 13
      6.5.6 Serial Receive 0 (serrecv0, at 0x30) ............... 13
      6.5.7 Serial Status 0 (serstat0, at 0x40) ............... 13
7  Device Tree Additions

8  Component Debugging 14
8.1  Interrupt Tracking 16
8.2  Security 16
8.3  Component Debug Registers 16
  8.3.1  Component Control and Status (ccsr, at 0x0) 16
  8.3.2  Authentication Data (authdata, at 0x8) 19
  8.3.3  DTM Interrupt Address (cdtmaddress, at 0x10) 19

9  RISC-V Debug Module 20
9.1  Bus Interface 20
9.2  Debug Mode 20
9.3  Debug Registers 22
  9.3.1  Component Control and Status (ccsr, at 0x0) 22
  9.3.2  DTM Interrupt Address (cdtmaddress, at 0x10) 24
  9.3.3  Debug Control and Status (dcsr, at 0x20) 24
  9.3.4  PC Sample (pcsample, at 0x30) 25
  9.3.5  Stuff Instruction (dstuff, at 0x100) 25
  9.3.6  Debug Jump (djump, at 0x110) 25
  9.3.7  PC (dpc, at 0x120) 26

10 Hardware Breakpoint Module 26
10.1  Hardware Breakpoint Registers 26
  10.1.1  Breakpoint Select (bpselect, at 0x780) 26
  10.1.2  Breakpoint Control (bpcontrol, at 0x781) 27
  10.1.3  Breakpoint Low Address (bploaddr, at 0x782) 29
  10.1.4  Breakpoint High Address (bphiaddr, at 0x783) 30
  10.1.5  Breakpoint Low Data (bplodata, at 0x784) 30
  10.1.6  Breakpoint High Data (bphidata, at 0x785) 30

11 Trace Module 30
11.1  Trace Data Format 31
11.2  Trace Events 31
11.3  Synchronization 31
11.4  Trace Registers 34
  11.4.1  Trace (trace, at 0x788) 34
  11.4.2  Trace Buffer Start (tbufstart, at 0x789) 36
  11.4.3  Trace Buffer End (tbufend, at 0x78a) 36
  11.4.4  Trace Buffer Write (tbufwrite, at 0x78b) 36

12 JTAG Debug Transport Agent 36
12.1  Background 37
12.2  JTAG Registers 37
  12.2.1  IDCODE (00001) 37
  12.2.2  DTM Control (dtmcontrol, at 10000) 37
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.

2 About This Document

2.1 Structure

This document contains 2 parts. The main part of the document is the specification, which is given in the numbered sections. Additionally there’s an appendix consisting of sections with letters instead of numbers. The information in the appendix is intended to clarify and give 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.

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 in 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>
<thead>
<tr>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>Read/Write.</td>
</tr>
<tr>
<td>R</td>
<td>Read-only.</td>
</tr>
<tr>
<td>W</td>
<td>Write-only. When read this field returns 0.</td>
</tr>
<tr>
<td>R/W0</td>
<td>Read/Write. Only writing 0 has some effect.</td>
</tr>
<tr>
<td>W1</td>
<td>Write-only. Only writing 1 has some effect.</td>
</tr>
</tbody>
</table>

Description of what this register is about.
Table 2: Sample Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x123</td>
<td>Long Name</td>
</tr>
</tbody>
</table>

2.3.1 Long Name (shortname, at 0x123)

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>Description of what this field is used for.</td>
<td>R/W</td>
<td>15</td>
</tr>
</tbody>
</table>

3 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 (e.g., 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.

4 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.

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.

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. 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.
Figure 1: RISC-V Debug System Overview
6 Debug Transport Module

Debug Transport Modules provide access to the system bus over one or more transports (e.g. 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.

6.1 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 aligned 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).
6.2 Interrupt Tracking

To avoid a debugger constantly polling the components it’s interested in (cluttering up the system bus), a very simple interrupt mechanism is supported. It consists of a single register where the DTM tracks which components have pending interrupts. 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.

Components can set bits in this register by writing the bit’s index to **dtminterrupt**. Alternatively, an implementation may choose to use interrupt signals so it is possible to debug a component that cannot write DTM registers.

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 may be configurable by the debugger by writing the component’s **interrupt** in **ccsr**.

6.3 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.

6.4 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 6.5. 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.

6.5 Debug Transport Module Registers

6.5.1 Interrupt (**dtminterrupt**, at 0x0)

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

<table>
<thead>
<tr>
<th>31</th>
<th>width</th>
<th>width-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-width + 32</td>
<td>width</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3: Debug Transport Module Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Interrupt</td>
</tr>
<tr>
<td>0x4</td>
<td>Control</td>
</tr>
<tr>
<td>0x8</td>
<td>Authentication Data</td>
</tr>
<tr>
<td>0x10</td>
<td>Serial Info</td>
</tr>
<tr>
<td>0x20</td>
<td>Serial Send 0</td>
</tr>
<tr>
<td>0x30</td>
<td>Serial Receive 0</td>
</tr>
<tr>
<td>0x40</td>
<td>Serial Status 0</td>
</tr>
<tr>
<td>0x50</td>
<td>Serial Send 1</td>
</tr>
<tr>
<td>0x60</td>
<td>Serial Receive 1</td>
</tr>
<tr>
<td>0x70</td>
<td>Serial Status 1</td>
</tr>
<tr>
<td>0x80</td>
<td>Serial Send 2</td>
</tr>
<tr>
<td>0x90</td>
<td>Serial Receive 2</td>
</tr>
<tr>
<td>0xa0</td>
<td>Serial Status 2</td>
</tr>
<tr>
<td>0xb0</td>
<td>Serial Send 3</td>
</tr>
<tr>
<td>0xc0</td>
<td>Serial Receive 3</td>
</tr>
<tr>
<td>0xd0</td>
<td>Serial Status 3</td>
</tr>
<tr>
<td>0xf0</td>
<td>Serial Send 4</td>
</tr>
<tr>
<td>0x100</td>
<td>Serial Receive 4</td>
</tr>
<tr>
<td>0x110</td>
<td>Serial Status 4</td>
</tr>
<tr>
<td>0x120</td>
<td>Serial Send 5</td>
</tr>
<tr>
<td>0x130</td>
<td>Serial Receive 5</td>
</tr>
<tr>
<td>0x140</td>
<td>Serial Status 5</td>
</tr>
<tr>
<td>0x150</td>
<td>Serial Send 6</td>
</tr>
<tr>
<td>0x160</td>
<td>Serial Receive 6</td>
</tr>
<tr>
<td>0x170</td>
<td>Serial Status 6</td>
</tr>
<tr>
<td>0x180</td>
<td>Serial Send 7</td>
</tr>
<tr>
<td>0x190</td>
<td>Serial Receive 7</td>
</tr>
<tr>
<td>0x1a0</td>
<td>Serial Status 7</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>value</td>
<td>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.)</td>
</tr>
</tbody>
</table>

### 6.5.2 Control (dtmcontrol, at 0x4)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>abussize</td>
<td>Width of the address bus in bits. (This includes the extra address bit if it’s required to access the debug bus.)</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>access128</td>
<td>1 when 128-bit bus accesses are supported.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>access64</td>
<td>1 when 64-bit bus accesses are supported.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>access32</td>
<td>1 when 32-bit bus accesses are supported.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>access16</td>
<td>1 when 16-bit bus accesses are supported.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>access8</td>
<td>1 when 8-bit bus accesses are supported.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>intbits</td>
<td>The width of the internal interrupt state is intbits + 1.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>authenticated</td>
<td>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.</td>
<td>R</td>
<td>Preset</td>
</tr>
</tbody>
</table>

Continued on next page
### 6.5.3 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.

<table>
<thead>
<tr>
<th>authbusy</th>
<th>While 1, writes to <code>authdata</code> 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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>authtype</td>
<td>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.</td>
</tr>
<tr>
<td>ndreset</td>
<td>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.</td>
</tr>
<tr>
<td>fullreset</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
### 6.5.4 Serial Info (serinfo, at 0x10)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial7</td>
<td>Like serial0.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>serial6</td>
<td>Like serial0.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>serial5</td>
<td>Like serial0.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>serial4</td>
<td>Like serial0.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>serial3</td>
<td>Like serial0.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>serial2</td>
<td>Like serial0.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>serial1</td>
<td>Like serial0.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>serial0</td>
<td>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.</td>
<td>R</td>
<td>Preset</td>
</tr>
</tbody>
</table>

#### 6.5.5 Serial Send 0 (sersend0, at 0x20)

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

\[
\text{width-1} \quad 0 \\
\begin{array}{|c|}
\hline
\text{data} \\
\hline
\end{array}
\]

#### 6.5.6 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.

\[
\text{width-1} \quad 0 \\
\begin{array}{|c|}
\hline
\text{data} \\
\hline
\end{array}
\]

#### 6.5.7 Serial Status 0 (serstat0, at 0x40)

\[
\begin{array}{|c|c|c|}
\hline
31 & 2 & 1 & 0 \\
\hline
0 & sendr & recvr \\
\hline
30 & 1 & 1 \\
\hline
\end{array}
\]
### 7 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 4 in the device tree.

#### Table 4: Component Device Tree Properties

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendr</td>
<td>Send ready. 1 when the send queue is not full. 0 otherwise.</td>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>recvr</td>
<td>Receive ready. 1 when the receive queue is not empty. 0 otherwise.</td>
<td>R</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

### 8 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.
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 (e.g., 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 (e.g., 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.
8.1 Interrupt Tracking

When \texttt{cdisable} is clear, a component can send an interrupt to the DTM by writing \texttt{interrupt} to \texttt{dtminterrupt} or by using a hard-wired interrupt line as described in Section 6.2.

8.2 Security

Some components may contain intellectual 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.

8.3 Component Debug Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Component Control and Status</td>
</tr>
<tr>
<td>0x8</td>
<td>Authentication Data</td>
</tr>
<tr>
<td>0x10</td>
<td>DTM Interrupt Address</td>
</tr>
</tbody>
</table>

8.3.1 Component Control and Status (ccsr, at 0x0)

See Figure 2 for more information about how freeze, halt, and reset bits interact.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticated</td>
<td>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.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>authbusy</td>
<td>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.</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>ndreset</td>
<td>Every time this bit is written as 1, it triggers a reset of the non-debug logic in this component.</td>
<td>W</td>
<td>0</td>
</tr>
<tr>
<td>fullreset</td>
<td>Every time this bit is written as 1, it triggers a reset of the component including the debug logic.</td>
<td>W</td>
<td>0</td>
</tr>
<tr>
<td>stopcycle</td>
<td>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.</td>
<td>R/W</td>
<td>1</td>
</tr>
<tr>
<td>stoptime</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>frozen</td>
<td>1 when the component is currently frozen.</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>freezesup</td>
<td>1 when freeze and freezersume are supported.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>freeze</td>
<td>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 freezersume 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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>freezersume</td>
<td>If this bit is written as 1 while the component is frozen, the component becomes unfrozen.</td>
<td>W1</td>
<td>0</td>
</tr>
<tr>
<td>halted</td>
<td>1 when the component is currently halted.</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>haltsup</td>
<td>1 when halt and resume are supported.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>halt</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

Continued on next page
8.3.2 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.

```
63 0
```

8.3.3 DTM Interrupt Address (cdtmaddress, at 0x10)

On platforms with only a single DTM, this register may be read-only, pointed at that DTM.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtmaddress</td>
<td>Bits <code>abussize-1:1</code> of the address of <code>dtminterrupt</code>. On some components this register will be less wide than <code>abussize</code>. 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.</td>
<td>R/W</td>
<td>Preset</td>
</tr>
<tr>
<td>wired</td>
<td>When 1, the component has a hard-wired interrupt line to the DTM, and any writes to <code>dtmaddress</code> are unnecessary and must be ignored.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>cdisable</td>
<td>When 1, the component won’t write anything to the DTM, and won’t use its dedicated interrupt line if it has one. Set to 0 to enable the component writing to the DTM.</td>
<td>R/W</td>
<td>1</td>
</tr>
</tbody>
</table>

9 RISC-V Debug Module

The RISC-V Debug Module implements all the functionality of a debuggable component as described in Section 8.

9.1 Bus Interface

Each RISC-V debug module has a bus slave exposing a 17 bit address space. It can be connected to the system bus or a debug bus. All accesses to this space must be aligned with the size of the access. n-bit registers are only accessible using n-bit accesses. The lower 4 bits of every address are always 0, so an implementation only needs to wire up 13 address bits. The address space is shown in Table 6.

TODO: Work with Stefan Wallentowitz to make the address map the same as https://github.com/opensocdebug/hardware/blob/mmio-relocate/modules/cdm_ads/doc/specification.md#memory-map

9.2 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.
Table 6: Debug Bus Memory Space

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 – 0x0100</td>
<td>Debug registers described in Section 9.3. These are always accessible.</td>
</tr>
<tr>
<td>0x0100 – 0x01c3</td>
<td>Debug registers described in Section 9.3. These are only accessible when the core is halted.</td>
</tr>
<tr>
<td>0x01c4 – 0x01ff</td>
<td>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.</td>
</tr>
<tr>
<td>0x0200 – 0x03ff</td>
<td>General purpose registers (x0–x31), each 16 bytes in size. These may only be accessible if the core is halted.</td>
</tr>
<tr>
<td>0x0400 – 0x05ff</td>
<td>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.</td>
</tr>
<tr>
<td>0x0600 – 0x3fff</td>
<td>Reserved for future official extensions.</td>
</tr>
<tr>
<td>0x4000 – 0x7fff</td>
<td>Reserved for future debug standards.</td>
</tr>
<tr>
<td>0x8000 – 0xffff</td>
<td>Reserved for custom use to support platform-specific functions.</td>
</tr>
<tr>
<td>0x10000 – 0x1ffff</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 \texttt{dpc}.

9.3 Debug Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Component Control and Status</td>
</tr>
<tr>
<td>0x10</td>
<td>DTM Interrupt Address</td>
</tr>
<tr>
<td>0x20</td>
<td>Debug Control and Status</td>
</tr>
<tr>
<td>0x30</td>
<td>PC Sample</td>
</tr>
<tr>
<td>0x100</td>
<td>Stuff Instruction</td>
</tr>
<tr>
<td>0x110</td>
<td>Debug Jump</td>
</tr>
<tr>
<td>0x120</td>
<td>PC</td>
</tr>
</tbody>
</table>

9.3.1 Component Control and Status (\texttt{ccsr}, at 0x0)

This is the exact same register as is described in Section 8.3.1, but the description here is slightly more specific to RISC-V cores.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Access</th>
<th>Preset</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopcycle</td>
<td>Controls the behavior of any counters while the component is halted. This includes the counters read with <code>rdcycle</code> and <code>rdinstret</code>. 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.</td>
<td>R/W</td>
<td>1</td>
</tr>
<tr>
<td>stoptime</td>
<td>Controls the behavior of any timers while the component is halted. This includes the timer read with <code>rdcycle</code>. 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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>frozen</td>
<td></td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>freezesup</td>
<td></td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>freeze</td>
<td></td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>freezeresume</td>
<td></td>
<td>W/I</td>
<td>0</td>
</tr>
<tr>
<td>halted</td>
<td></td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>haltsup</td>
<td></td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>halt</td>
<td>When this bit is 1 and the core is running, the core will enter Debug Mode, setting <code>dpc</code> 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, <code>dpc</code> 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. <code>dpc</code> is set to the address of the next instruction to be executed. Setting this bit to 0 does not have an immediate effect.</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>
If this bit is written as 1 while the core is in Debug Mode, the core leaves Debug Mode and resumes execution at dpc.

interrupt See Section 8.3.1.

9.3.2 DTM Interrupt Address (cdtmaddress, at 0x10)

See Section 8.3.3.

9.3.3 Debug Control and Status (dcsr, at 0x20)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcsample</td>
<td>1 when the pcsample is implemented. 0 otherwise.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>haltinterrupt</td>
<td>1 when the core will write interrupt to dtminterrupt when it halts. 0 otherwise.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>xdebugver</td>
<td>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.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>hwbpcoun</td>
<td>t Number of hardware breakpoints this core supports.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>debug</td>
<td>1 when the core is in Debug Mode. 0 otherwise.</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>cause</td>
<td>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.</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

Continued on next page
### 9.3.4 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.

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Recent value of PC</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

### XLEN-1 0 pc XLEN

### 9.3.5 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 (e.g., 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.

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction</td>
<td>32-bit instruction</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

### XLEN-1 0 instruction XLEN

### 9.3.6 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 dtinterrupt 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.

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Jump address</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

### XLEN-1 0 address XLEN
9.3.7 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.

\[
\begin{array}{c}
\text{dpc} \\
\hline
\text{XLEN-1} \\
\text{0}
\end{array}
\]

10 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 \texttt{bpselect}, and then querying \texttt{bpcontrol}.

10.1 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 and debug mode,
to prevent untrusted user code from causing entry into Debug Mode without
the OS’s permission.

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x780</td>
<td>Breakpoint Select</td>
</tr>
<tr>
<td>0x781</td>
<td>Breakpoint Control</td>
</tr>
<tr>
<td>0x782</td>
<td>Breakpoint Low Address</td>
</tr>
<tr>
<td>0x783</td>
<td>Breakpoint High Address</td>
</tr>
<tr>
<td>0x784</td>
<td>Breakpoint Low Data</td>
</tr>
<tr>
<td>0x785</td>
<td>Breakpoint High Data</td>
</tr>
</tbody>
</table>

10.1.1 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.
10.1.2 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:

\[
\begin{align*}
\text{amatch} &= (\text{aen} \land \text{arangeen} \land \bar{\text{amask}}) \lor \\
& \quad (\text{aen} \land \text{address} = \text{bploadaddr}) \lor \\
& \quad (\text{arangeen} \land \text{address} \geq \text{bploadaddr} \land \text{address} < \text{bphiaddr}) \lor \\
& \quad (\text{amask} \land (\text{address} \& \text{bphiaddr}) = \text{bploadaddr}) \\
\text{dmatch} &= (\text{den} \land \text{drangeen} \land \bar{\text{dmask}}) \lor \\
& \quad (\text{den} \land \text{data} = \text{bplodata}) \lor \\
& \quad (\text{drangeen} \land \text{data} \geq \text{bplodata} \land \text{data} < \text{bphidata}) \lor \\
& \quad (\text{dmask} \land (\text{data} \& \text{bphidata}) = \text{bplodata}) \\
\text{omatch} &= (\text{loaden} \land \text{access is load}) \lor \\
& \quad (\text{storeen} \land \text{access is store}) \lor \\
& \quad (\text{execen} \land \text{access is exec}) \\
\text{match} &= \text{amatch} \land \text{dmatch} \land \text{omatch}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>bp</td>
<td>Select this hardware breakpoint.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Access</td>
<td>Reset</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>owner</td>
<td>Hardware breakpoints can be used both by an external debugger, and by the software running on a core. This field exists to prevent conflicts between the two. 0 means the breakpoint is up for grabs by anybody. 1 means the breakpoint is owned by software. 2 means the breakpoint is owned by the debugger. When a breakpoint is owned by the debugger, the hardware may prevent changes to the registers of that breakpoint while not in Debug Mode. The converse is not true. The debugger can always modify a breakpoint owned by software, but a debugger must not do that without explicit direction from the user that it’s OK to modify a software-owned breakpoint.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>loadsup</td>
<td>This breakpoint supports matching on data load.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>storesup</td>
<td>This breakpoint supports matching on data store.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>execsup</td>
<td>This breakpoint supports matching on instruction execution.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>asup</td>
<td>This breakpoint supports exact address matches. (It would be a strange breakpoint that doesn’t.)</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>arangesup</td>
<td>This breakpoint supports range address matches.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>amasksup</td>
<td>This breakpoint supports masked address matches.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>dsup</td>
<td>This breakpoint supports exact data matches.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>drangesup</td>
<td>This breakpoint supports range data matches.</td>
<td>R</td>
<td>Preset</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Access</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmasksup</td>
<td>This breakpoint supports masked data matches.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>matched</td>
<td>Set to 1 when this hardware breakpoint matched. The debugger is responsible for clearing this bit once it has seen it’s set.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>action</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>loaden</td>
<td>Set to enable this breakpoint for data loads.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>storeen</td>
<td>Set to enable this breakpoint for data stores.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>execen</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>aen</td>
<td>Set to cause this breakpoint to match when address equals bploaddr.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>arangeen</td>
<td>Set to cause this breakpoint to match when bploaddr &lt;= address &lt; bphiaddr.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>amasken</td>
<td>Set to cause this breakpoint to match when address&amp;bphiaddr = bploaddr.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>den</td>
<td>Set to cause this breakpoint to match when data equals bplodata.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>drangeen</td>
<td>Set to cause this breakpoint to match when bplodata &lt;= data &lt; bphidata.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>dmasken</td>
<td>Set to cause this breakpoint to match when data&amp;bphidata = bplodata.</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

### 10.1.3 Breakpoint Low Address (bploaddr, at 0x782)

Used for exact match or lower bound (inclusive) of the address match for this breakpoint.
10.1.4 Breakpoint High Address (bphiaddr, at 0x783)
Used for upper bound (inclusive) of the address match for this breakpoint, or as address mask.

10.1.5 Breakpoint Low Data (bplodata, at 0x784)
Used for exact match or lower bound (inclusive) of the data match for this breakpoint.

10.1.6 Breakpoint High Data (bphidata, at 0x785)
Used for upper bound (inclusive) of the data match for this breakpoint, or as data mask.

11 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.

\[\text{This part of the spec is functional, but could certainly be improved a lot.}\]

11.1 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 9.

\[\text{Is it an improvement to add a count after Branch Taken/Not Taken headers?}\]

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.

11.2 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 10.

11.3 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.
<table>
<thead>
<tr>
<th>Packet ID</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Nop</td>
<td>Packet that indicates no data. The trace source must use these to ensure that there are 8 synchronization points in each buffer.</td>
</tr>
<tr>
<td>0001</td>
<td>PC</td>
<td>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.</td>
</tr>
<tr>
<td>0010</td>
<td>Branch Taken</td>
<td>Followed by a single packet indicating the version of the trace data (currently 0).</td>
</tr>
<tr>
<td>0011</td>
<td>Branch Not Taken</td>
<td>Indicates that trace was purposefully disabled, or that some sequences were dropped because the trace buffer overflowed.</td>
</tr>
<tr>
<td>0100</td>
<td>Trace Enabled</td>
<td>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.</td>
</tr>
<tr>
<td>0101</td>
<td>Trace Disabled</td>
<td>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.</td>
</tr>
<tr>
<td>0110</td>
<td>Privilege Level</td>
<td>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.</td>
</tr>
<tr>
<td>0111</td>
<td>Reserved</td>
<td>Reserved for future standards.</td>
</tr>
<tr>
<td>1000</td>
<td>Load Address</td>
<td>Followed by a Value Sequence containing the address. Missing bits must be filled in with the last Load Address value.</td>
</tr>
<tr>
<td>1001</td>
<td>Store Address</td>
<td>Followed by a Value Sequence containing the address. Missing bits must be filled in with the last Store Address value.</td>
</tr>
<tr>
<td>1010</td>
<td>Load Data</td>
<td>Followed by a Value Sequence containing the data. Missing bits must be filled in by sign extending the value.</td>
</tr>
<tr>
<td>1011</td>
<td>Store Data</td>
<td>Followed by a Value Sequence containing the data. Missing bits must be filled in by sign extending the value.</td>
</tr>
<tr>
<td>1100</td>
<td>Timestamp</td>
<td>Followed by a Value Sequence containing the timestamp. Missing bits should be filled in with the last Timestamp value.</td>
</tr>
<tr>
<td>1101</td>
<td>Reserved</td>
<td>Reserved for future standards.</td>
</tr>
<tr>
<td>1110</td>
<td>Custom</td>
<td>Reserved for custom trace data.</td>
</tr>
<tr>
<td>1111</td>
<td>Custom</td>
<td>Reserved for custom trace data.</td>
</tr>
</tbody>
</table>
Table 10: Trace Data Events

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>jal</td>
<td>If <code>emitbranch</code> is disabled but <code>emitpc</code> is enabled, emit 2 PC values: first the address of the instruction, then the address being jumped to.</td>
</tr>
<tr>
<td>jalr</td>
<td>If <code>emitbranch</code> is disabled but <code>emitpc</code> is enabled, emit 2 PC values: first the address of the instruction, then the address being jumped to. Otherwise, if <code>emitstoredata</code> is enabled emit just the destination PC.</td>
</tr>
<tr>
<td>BRANCH</td>
<td>If <code>emitbranch</code> is enabled, emit either Branch Taken or Branch Not Taken. Otherwise if <code>emitpc</code> is enabled and the branch is taken, emit 2 PC values: first the address of the branch, then the address being branched to.</td>
</tr>
<tr>
<td>LOAD</td>
<td>If <code>emitloadaddr</code> is enabled, emit the address. If <code>emitloaddata</code> is enabled, emit the data that was loaded.</td>
</tr>
<tr>
<td>STORE</td>
<td>If <code>emitstoreaddr</code> is enabled, emit the address. If <code>emitstoredata</code> is enabled, emit the data that is stored.</td>
</tr>
<tr>
<td>Traps</td>
<td><code>scall</code>, <code>sbreak</code>, <code>ecall</code>, <code>ebreak</code>, and <code>eret</code> emit the same as if they were jal instructions. In addition they also emit a Privilege Level sequence.</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Emit PC (if enabled) of the last instruction executed. Emit Privilege Level (if enabled). Finally emit the new PC (if enabled).</td>
</tr>
<tr>
<td>CSR instructions</td>
<td>For reads emit Load Data (if enabled). For writes emit Store Data (if enabled).</td>
</tr>
<tr>
<td>Data Dropped</td>
<td>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.</td>
</tr>
</tbody>
</table>
11.4 Trace Registers

Table 11: Trace Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x788</td>
<td>Trace</td>
</tr>
<tr>
<td>0x789</td>
<td>Trace Buffer Start</td>
</tr>
<tr>
<td>0x78a</td>
<td>Trace Buffer End</td>
</tr>
<tr>
<td>0x78b</td>
<td>Trace Buffer Write</td>
</tr>
</tbody>
</table>

11.4.1 Trace (trace, at 0x788)

```
<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrapped</td>
<td>1 if the trace buffer has wrapped since the last time discard was written. 0 otherwise.</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>emittimestamp</td>
<td>Emit Timestamp trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>emitstoredata</td>
<td>Emit Store Data trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>emitloaddata</td>
<td>Emit Load Data trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>emitstoreaddr</td>
<td>Emit Store Address trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>emitloadaddr</td>
<td>Emit Load Address trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>emitpriv</td>
<td>Emit Privilege Level trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>emitbranch</td>
<td>Emit Branch Taken and Branch Not Taken trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>emitpc</td>
<td>Emit PC trace sequences.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>fullaction</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>destination</td>
<td>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.</td>
<td>R/W</td>
<td>Preset</td>
</tr>
</tbody>
</table>
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.

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.

<table>
<thead>
<tr>
<th>stall</th>
<th>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.</th>
<th>R/W</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>discard</td>
<td>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.</td>
<td>W1</td>
<td>0</td>
</tr>
</tbody>
</table>

### 11.4.2 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 |

### 11.4.3 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 |

### 11.4.4 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 |

### 12 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).
12.1 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.

12.2 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 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.

12.2.1 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Identifies the release version of this part.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>PartNumber</td>
<td>Identifies the designer’s part number of this part.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>Manufld</td>
<td>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.</td>
<td>R</td>
<td>Preset</td>
</tr>
</tbody>
</table>

12.2.2 DTM Control (dtmcontrol, at 10000)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>serinfo</td>
<td></td>
</tr>
<tr>
<td>dtmcontrol</td>
<td></td>
</tr>
</tbody>
</table>

37
Table 12: JTAG TAP Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>BYPASS</td>
<td>JTAG recommends this encoding</td>
</tr>
<tr>
<td>00001</td>
<td>IDCODE</td>
<td>JTAG recommends this encoding</td>
</tr>
<tr>
<td>00010</td>
<td>SAMPLE</td>
<td>JTAG requires this instruction</td>
</tr>
<tr>
<td>00011</td>
<td>PRELOAD</td>
<td>JTAG requires this instruction</td>
</tr>
<tr>
<td>00100</td>
<td>EXTEST</td>
<td>JTAG requires this instruction</td>
</tr>
<tr>
<td>00100</td>
<td>INIT,SETUP,CLAMP</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>00101</td>
<td>CLAMP</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>00110</td>
<td>CLAMP,HOLD</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>00111</td>
<td>CLAMP,RELEASE</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>01000</td>
<td>HIGHZ</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>01001</td>
<td>IC_RESET</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>01010</td>
<td>TMP,STATUS</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>01011</td>
<td>INIT,SETUP</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>01100</td>
<td>INIT,RUN</td>
<td>JTAG recommends this instruction</td>
</tr>
<tr>
<td>01110</td>
<td>Unused (BYPASS)</td>
<td>Reserved for future JTAG</td>
</tr>
<tr>
<td>01111</td>
<td>Unused (BYPASS)</td>
<td>Reserved for future JTAG</td>
</tr>
<tr>
<td>10000</td>
<td>DTM Control</td>
<td>DTM Control</td>
</tr>
<tr>
<td>10001</td>
<td>DTM Authentication Data</td>
<td>DTM Authentication</td>
</tr>
<tr>
<td>10010</td>
<td>JTAG Bus Control</td>
<td>For bus access</td>
</tr>
<tr>
<td>10011</td>
<td>JTAG Bus Address</td>
<td>For bus access</td>
</tr>
<tr>
<td>10100</td>
<td>JTAG Bus Data</td>
<td>For bus access</td>
</tr>
<tr>
<td>10101</td>
<td>JTAG Status</td>
<td>For interrupts/serial</td>
</tr>
<tr>
<td>10110</td>
<td>JTAG Status Control</td>
<td>For interrupts/serial</td>
</tr>
<tr>
<td>10111</td>
<td>JTAG Serial Data</td>
<td>For serial</td>
</tr>
<tr>
<td>11000</td>
<td>Reserved (BYPASS)</td>
<td>Reserved for future RISC-V debugg</td>
</tr>
<tr>
<td>11001</td>
<td>Reserved (BYPASS)</td>
<td>Reserved for future RISC-V debugg</td>
</tr>
<tr>
<td>11010</td>
<td>Reserved (BYPASS)</td>
<td>Reserved for future RISC-V debugg</td>
</tr>
<tr>
<td>11011</td>
<td>Unused (BYPASS)</td>
<td>Reserved for customization</td>
</tr>
<tr>
<td>11100</td>
<td>Unused (BYPASS)</td>
<td>Reserved for customization</td>
</tr>
<tr>
<td>11101</td>
<td>Unused (BYPASS)</td>
<td>Reserved for customization</td>
</tr>
<tr>
<td>11110</td>
<td>Unused (BYPASS)</td>
<td>Reserved for customization</td>
</tr>
<tr>
<td>11111</td>
<td>BYPASS</td>
<td>JTAG requires this encoding</td>
</tr>
</tbody>
</table>
Table 13: JTAG Access Size

<table>
<thead>
<tr>
<th>Encoding</th>
<th>size</th>
<th>loabits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>other</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Table 14: JTAG Address Bits

<table>
<thead>
<tr>
<th>Encoding</th>
<th>hiabits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\text{min}(7, \text{abussize} - 1)</td>
</tr>
<tr>
<td>1</td>
<td>\text{min}(11, \text{abussize} - 1)</td>
</tr>
<tr>
<td>2</td>
<td>\text{min}(15, \text{abussize} - 1)</td>
</tr>
<tr>
<td>3</td>
<td>\text{min}(23, \text{abussize} - 1)</td>
</tr>
<tr>
<td>4</td>
<td>\text{min}(31, \text{abussize} - 1)</td>
</tr>
<tr>
<td>5</td>
<td>\text{min}(63, \text{abussize} - 1)</td>
</tr>
<tr>
<td>6</td>
<td>\text{min}(127, \text{abussize} - 1)</td>
</tr>
<tr>
<td>other</td>
<td>reserved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>serinfo</td>
<td>Contains the lower 16 bits of \text{serinfo} as described in Section 6.5.4.</td>
<td>R</td>
<td>Preset</td>
</tr>
<tr>
<td>dtmcontrol</td>
<td>Contains \text{dtmcontrol} as described in Section 6.5.2.</td>
<td>R/W</td>
<td>Preset</td>
</tr>
</tbody>
</table>

12.2.3 DTM Authentication Data (authdata, at 10001)

This register is the JTAG view of the DTM register described in Section 6.5.3. It only exists if authtype isn’t 0.

12.2.4 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.
Table 15: Serial Ports in jstatus

<table>
<thead>
<tr>
<th>Encoding</th>
<th>hiabits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>min(7, abussize - 1)</td>
</tr>
<tr>
<td>1</td>
<td>min(11, abussize - 1)</td>
</tr>
<tr>
<td>2</td>
<td>min(15, abussize - 1)</td>
</tr>
<tr>
<td>3</td>
<td>min(23, abussize - 1)</td>
</tr>
<tr>
<td>4</td>
<td>min(31, abussize - 1)</td>
</tr>
<tr>
<td>5</td>
<td>min(63, abussize - 1)</td>
</tr>
<tr>
<td>6</td>
<td>min(127, abussize - 1)</td>
</tr>
<tr>
<td>other</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Field | Description | Access | Reset |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>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.</td>
<td>R/W0</td>
<td>0</td>
</tr>
<tr>
<td>dbits</td>
<td>Set hidbits to ((dbits + 1) \times 4 - 1). If hidbits is set to be larger than size, the extra bits scanned into data will be ignored.</td>
<td>R/W</td>
<td>7</td>
</tr>
<tr>
<td>abits</td>
<td>Set hiabits per Table 15.</td>
<td>R/W</td>
<td>5</td>
</tr>
<tr>
<td>size</td>
<td>Set size and loabits per Table 13.</td>
<td>R/W</td>
<td>2</td>
</tr>
</tbody>
</table>

12.2.5 JTAG Bus Address (jaddress, at 10011)

Field | Description | Access | Reset |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>Update address[hiabits:loabits] with the value in update.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>autoincrement</td>
<td>When set, increment address by size/8 after every scan of jdata.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>read/busy</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>
12.2.6 JTAG Bus Data (jdata, at 10100)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>write/valid</td>
<td>Set this bit to write data to the current address. Read this bit to determine whether the register contains data from a successful read.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>read/busy</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

12.2.7 JTAG Status (jstatus, at 10101)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>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.</td>
<td>R</td>
<td>…01</td>
</tr>
</tbody>
</table>

Continued on next page
### 12.2.8 JTAG Status Control (jstatc, at 10110)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>serscan</td>
<td>Select the number of serial ports that show up in jstatus. serbits is $4 \times serscan + 4$. (There are 2 bits per serial port.)</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>intscan</td>
<td>Select the number of interrupt status bits that show up in jstatus. intbits is $4 \times intscan$.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>serselect</td>
<td>Select which serial port jserial accesses.</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

### 12.2.9 JTAG Serial Data (jserial, at 10111)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>write/valid</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----</td>
<td>---</td>
</tr>
<tr>
<td>busy</td>
<td>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.</td>
<td>R/W</td>
<td>0</td>
</tr>
</tbody>
</table>

12.2.10 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.

```
0
0
1
```
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 cdmaddres$, 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.

```assembly
loop:
    lw t0, 0x10(a0)    # Load status.
    andi t0, t0, SERSTAT_RECRVR_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 (e.g., 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.

```markdown
Table 16: Control and Status Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x770</td>
<td>Bus State</td>
</tr>
<tr>
<td>0x771</td>
<td>Bus Address</td>
</tr>
<tr>
<td>0x772</td>
<td>Bus Data</td>
</tr>
<tr>
<td>0x773</td>
<td>Debug PC</td>
</tr>
<tr>
<td>0x774</td>
<td>Debug Scratch 0</td>
</tr>
<tr>
<td>0x775</td>
<td>Debug Scratch 1</td>
</tr>
<tr>
<td>0x776</td>
<td>Component Control and Status</td>
</tr>
<tr>
<td>0x777</td>
<td>DTM Interrupt Address</td>
</tr>
</tbody>
</table>
```

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>complete</code></td>
<td>Write 1 to this register to complete the currently pending access.</td>
<td>W1</td>
<td>0</td>
</tr>
</tbody>
</table>

Continued on next page
B.2.2 Bus Address (busaddress, at 0x771)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Access</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>When an access is pending, contains the address of that access.</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

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 Component Control and Status (ccsr, at 0x776)
CSR view of ccsr defined in Section 8.3.1.

B.2.8 DTM Interrupt Address (cdtmaddress, at 0x777)
CSR view of cdtmaddress defined in Section 8.3.3.

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 CCSR 0x776
#define CCSR_INTERRUPT_MASK 0xff
#define CDTMADDRESS 0x777

# 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
_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
    andi s9, s9, BUSSTATE_READ
    beqz s9, read

    # Handle a bus read.
    read:
    li s9, 0x120
    bne s8, s9, rskip0

    # Read PC
    csrr s9, DPC
    j access_done

rskip0:
    li s9, 0x1c4
    bne s8, s9, rskip1

    # Read whether GPRs are accessible. (Of course they are.)
    li s9, 0xffffffff
    j access_done

rskip1:
    li s9, 0x380
    bne s8, s9, rskip2

    # Read s8
    csrr s9, SCRATCH0
    j access_done

rskip2:
    li s9, 0x390
    bne s8, s9, rskip5

    # Read s9
    csrr s9, SCRATCH1
j    access_done

rskip5:
    li    s9, 0x400
    bge  s9, s8, rskip6
    li    s9, 0x200
    blt  s8, s9, rskip6

# Read from GPR (but not s8--s9).
# Generate "mv s9, <from>"
# GPR number is in bits 8:4 of s8, and needs to be in bits 19:15 of
# the instruction.
    andi  s8, s8, 0x1f0
    sll  s9, s8, 19-8
    li    s9, 0xc93
    or   s8, s8, s9
    j    execute_instruction

rskip6:
    li    s9, 0x10000
    blt  s8, s9, rskip7

# Read from CSR.
# Generate "csrr s9, <from>"
# CSR number is in bits 15:4 of s8, and needs to be in bits 31:20 of
# the instruction.
    li    s9, 0xfff8
    and  s9, s9, s8
    sll  s9, s9, 31-15
    li    s8, 0x2c73
    or   s8, s8, s9
    j    execute_instruction

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

# Handle a bus write.
# s8 contains the address
write:
    li    s9, 0x100
    bne  s8, s9, wskip1

# stuff instruction
    csrr  s8, BUSDATA
    j    execute_instruction

wskip1:
li    s9, 0x110
bne  s8, s9, wskip2

# jump to address
li    s8, BUSSTATE_COMPLETE
csrw BUSSTATE, s8
csr s8, BUSDATA
jr s8

# At the end of the code we jump to must be an ebreak, which gets us back to reentry.

wskip2:
li    s9, 0x120
bne  s8, s9, wskip3

# Write PC
csrr s8, BUSDATA
csrw DPC, s8
j access_done

wskip3:
li    s9, 0x380
bne  s8, s9, wskip4

# Write s8
csrr s8, BUSDATA
csrw SCRATCH0, s8
j access_done

wskip4:
li    s9, 0x390
bne  s8, s9, wskip7

# Write s9
csrr s8, BUSDATA
csrw SCRATCH1, s8
j access_done

wskip7:
li    s9, 0x400
bge s9, s8, wskip8
li    s9, 0x200
blt s8, s9, wskip8

# Write to GPR (but not s8--s9).
    # Generate "mv <to>, s9"
    # GPR number is in bits 8:4 of s8, and needs to be in bits 11:7 of the instruction.
andi s8, s8, 0x1f0
sll s9, s8, 11-8
li s8, 0xc8013
or    s8, s8, s9
j    execute_instruction

wskip8:
    li    s9, 0x10000
    blt    s8, s9, wskip9

# Write to CSR.
# Generate "csrw <to>, s9"
# CSR number is in bits 15:4 of s8, and needs to be in bits 31:20 of
# the instruction.
    li    s9, 0xfff8
    and    s9, s9, s8
    sll    s9, s9, 31-15
    li    s8, 0xc9073
    or    s8, s8, s9
    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 BUSDATA.
    # Jump to access_done once the instruction is executed.
    la    s9, debug_ram
    sw    s8, 0(s9)
    li    s8, 0x000c0067  # jr s8
    sw    s8, 4(s9)
    la    s8, access_done
    csrr    s9, BUSDATA
    fence.i
    j    debug_ram

_exit:
    csrr    s8, SCRATCH0
    csrr    s9, SCRATCH1
    eret    # TODO: dret?
debug_ram:
    # This symbol needs to actually be the address of debug_ram, which
    # can’t be too far from the ROM so that a PC-relative jump can be used
    # to get there.

C  TODO

1. Spec DTM for 1149.7 Data Channels, if we can find a cheapish debug probe that will talk that protocol efficiently.

2. Once some kind of power controller is specified, think about debugging when some peripherals may be powered down.

3. Once some kind of interrupt controller is specified, think about how to have halt on one core trigger halt of another core.

D  Future Ideas

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

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

2. There’s a general interface for automatically sending larger blocks of data to the debugger when an interrupt happens.

3. Multiple components can be halted/run/stepped simultaneously.

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

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

6. There’s a minimal DTM for use with systems that just have a single RISC-V core and no system bus.