RISC-V External Debug Support
Version 0.13
b4f1f439a57afe04aab61b8b2c42e490f3aaaf58

Tim Newsome <tim@sifive.com>

Thu Jun 8 10:20:07 2017 -0700

Preface

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

Acknowledgments

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

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

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

1.1 Terminology

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

1.2 About This Document

1.2.1 Structure

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

actual specification.

1.2.2 Register Definition Format

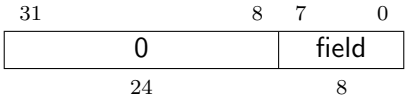
All register definitions in this document follow the format shown in Section 1.2.2. A simple graphic shows which fields are in the register. The upper and lower bit indices are shown to the top left and top right of each field. The total number of bits in the field are shown below it.

After the graphic follows a table which for each field lists its name, description, allowed accesses, and reset value. The allowed accesses are listed in Table 1.1.

Table 1.1: Register Access Abbreviations

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

Long Name (`shortname`, at 0x123)



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

1.3 Background

There are several use cases for dedicated debugging hardware, both internal to a CPU core and with an external connection. This specification defines techniques to support all of the use cases listed below. Some are optional to allow system designers to make cost vs capability tradeoffs.

- Debugging low-level software in the absence of an OS or other software.
- Debugging issues in the OS itself.
- Bootstrapping a system to test, configure, and program components before there is any executable code path in the system.

- Accessing hardware on the system without a working CPU.
- Accessing custom registers that could be added to aid in hardware debug, system bringup, etc.
- Writing code and data to memory, e.g. boot code and manufacturing constants.
- Analyzing low-level performance issues using profiling and sampling techniques.
- Providing a general transport for firmware running on a component to communicate with the outside world.

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

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

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

1.4 Supported Features

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

1. RV32, RV64, and future RV128 are all supported.
2. Any hart in the platform can be independently debugged.
3. A debugger can discover almost everything it needs to know itself, without user configuration.
4. An optional extension allows arbitrary instructions to be executed on a halted hart. That means no new debug functionality is needed when a core has additional or custom instructions or state, as long as there exist programs that can move that state into GPRs.
5. An implementation can choose to provide register access without halting.
6. An implemenation can choose to provide the ability to automatically halt and resume a hart without debugger intervention.

7. A system bus master can be implemented to allow memory access without involving any hart.
8. Debugging can be supported over multiple transports.
9. Code can be downloaded efficiently.
10. Each hart can be debugged from the very first instruction executed.
11. A RISC-V hart can be halted when a software breakpoint instruction is executed.
12. Hardware can step over any instruction.
13. A RISC-V hart can be halted when a trigger matches the PC, read/write address/data, or an instruction opcode.
14. Optional serial ports can be used for communication between debugger and monitor, or as a general protocol between debugger and application.
15. The debugger does not need to know anything about the microarchitecture of the cores it is debugging.
16. Multiple harts can be halted and resumed simultaneously.

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

Chapter 2

System Overview

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

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

The DM allows the debugger to halt any hart in the platform. Abstract commands provide access to GPRs. The optional Program Buffer allows the debugger to execute arbitrary code on the hart, which allows access to additional hart state. Alternatively, additional abstract commands can provide access to additional hart state.

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

An optional system bus access block allows memory accesses without using a RISC-V hart to perform the access.

Optional serial port blocks allow the Debug Transport to be re-used as a generic communication interface.

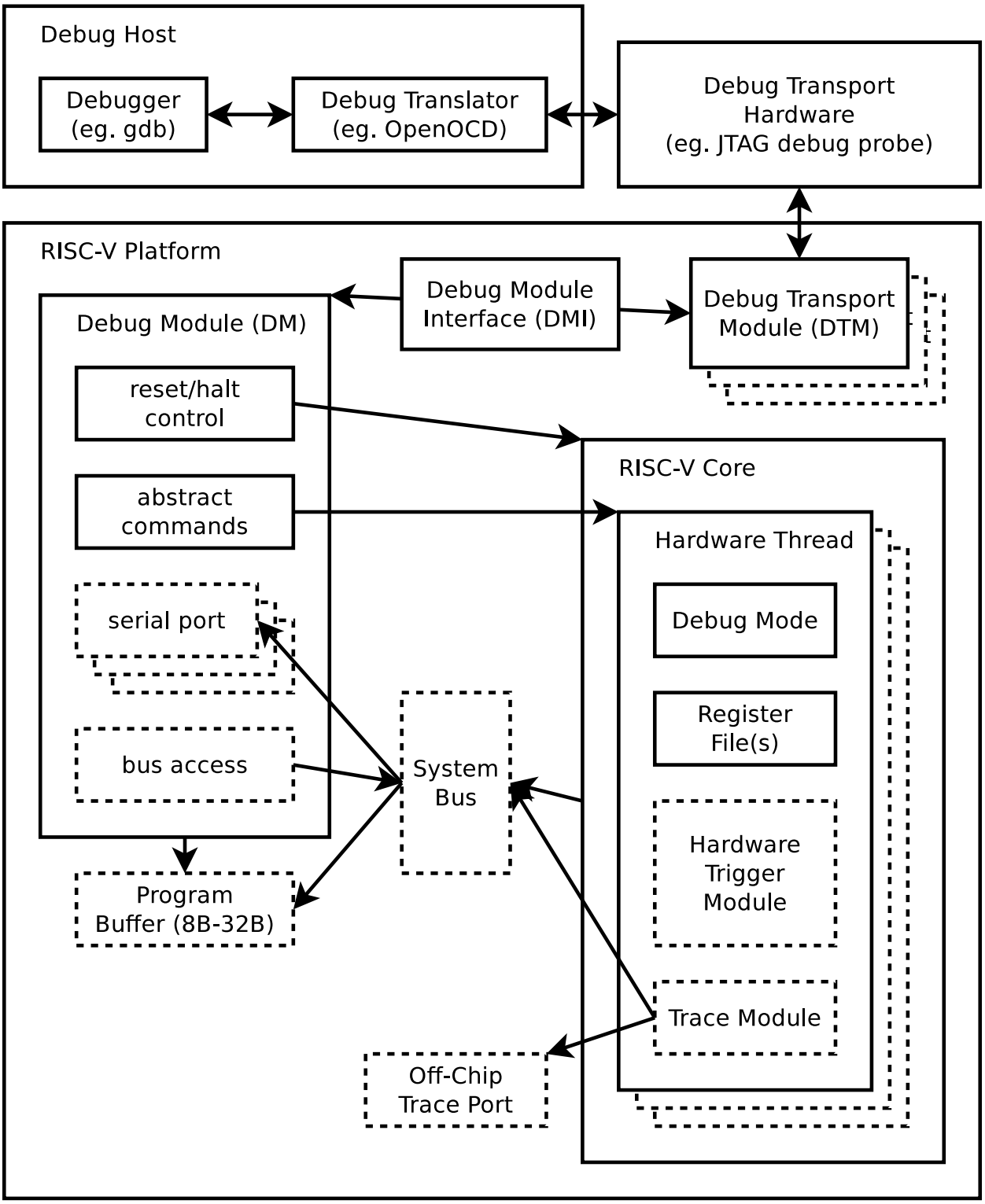


Figure 2.1: RISC-V Debug System Overview

Chapter 3

Debug Module (DM)

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

1. Give the debugger necessary information about the implementation. (Required)
2. Allow any individual hart to be halted and resumed. (Required)
3. Provide status on which harts are halted. (Required)
4. Provide read and write access to a halted hart's GPRs. (Required)
5. Provide access to a reset signal that allows debugging out of reset.(Required)
6. Provide access to other hart registers. (Optional)
7. Provide an interface to force the hart to execute arbitrary instructions. (Optional)
8. Allow multiple harts to be halted, resumed, and/or reset at the same time. (Optional)
9. Allow system memory accesses. (Optional)

A single DM can debug up to 1024 harts.

3.1 Debug Module Interface (DMI)

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

The DMI uses between 7 and 32 address bits. It supports read and write operations, which may return an error. (Errors are only returned by the optional System Bus Access and Serial Port

blocks). The bottom of the address space is used for the DM. Extra space can be used for custom debug devices, other cores, additional DMs, etc.

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

Table 3.1: Debug Module Interface Address Space

0x00 – 0x3f	Registers described in Section 3.11.
0x40 – 0x5f	There are 1024 bits here, one for each hart controlled by this debug module. If the hart is halted, the bit is 1. Otherwise the bit is 0. The bit for hart 0 is the LSB in the 32-bit word at 0x40. The bit for hart 1023 is the MSB in the 32-bit word at 0x5f.

3.2 Reset Control

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

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

3.3 Selecting Harts

Up to 1024 harts can be connected to a single DM. The debugger selects a hart, and then subsequent halt, resume, reset, and debugging commands are specific to that hart.

A debugger can enumerate all the harts attached to the DM by selecting each hart starting from 0 until anynonexistent is 1.

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

3.3.1 Selecting a Single Hart

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

3.3.2 Selecting Multiple Harts

Debug Modules may optionally implement a Hart Array Mask register to allow selecting multiple harts at once. The debugger can set bits in the hart array mask register using `hawindowsel` and `hawindow`, then apply actions to all selected harts by setting hasel. If this feature is supported, multiple harts can be halted, resumed, and reset simultaneously.

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

3.4 Halt Control

The Debug Module can halt harts and allow them to run again using the `dmcontrol` register. When a debugger wants to halt a single hart it selects it in hartsel and sets haltreq, then waits for allhalted to indicate the hart is halted before clearing haltreq to 0. Setting haltreq has no effect on a hart which is already halted.

To resume, the debugger selects the hart with hartsel, and sets resumereq. This action sets the hart's `resumeack` bit to 0. Once the hart has resumed, it sets its `resumeack` bit to 1. Thus, the debugger should wait for allresumeack to indicate the hart has resumed before clearing resumereq to 0. If a debugger reads allresumeack and allhalted in the same cycle, the hart must have resumed and then halted again.

When waiting for a hart to resume, a debugger should examine allresumeack*, not* allrunning *or* allhalted*, because the hart may immediately halt again due to trigger or step conditions.*

Setting resumereq on hart which is running clears `resumeack` but may cause a hart which halts in the future to immediately resume. Debuggers should not depend on this behavior and should not set resumereq for running harts.

To halt or resume multiple harts at the same time, the debugger first sets the hart's bits in the hart array mask register, then follows the same procedure but with hasel set to 1. Depending on the desired operation, the debugger might consider the `any*` versions of the status instead of `all*`.

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

3.5 Abstract Commands

The DM supports a set of abstract commands, most of which are optional. Depending on the implementation, the debugger may be able to perform some abstract commands even when the selected hart is not halted. Debuggers can only determine which abstract commands are supported by a given hart in a given state by attempting them and then looking at cmderr in `abstractcs` to see if they were successful.

Debuggers execute abstract commands by writing them to command. Debuggers can determine whether an abstract command is complete by reading busy in abstractcs. If the command takes arguments, the debugger must write them to the data registers before writing to command. If a command returns results, the Debug Module must ensure they are placed in the data registers before busy is cleared. Which data registers are used for the arguments is described in Table 3.2. In all cases the least-significant word is placed in the lowest-numbered data register.

Table 3.2: Use of Data Registers

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

Table 3.3: Abstract Register Numbers

0x0000 – 0x0fff	CSRs
0x1000 – 0x101f	GPRs
0x1020 – 0x103f	Floating point registers
0xc000 – 0xffff	Reserved for non-standard extensions and internal use.

3.5.1 Abstract Command Listing

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

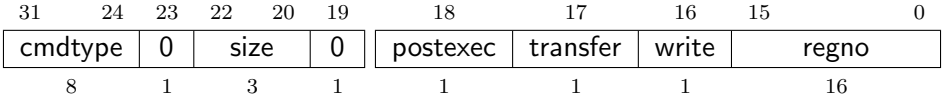
Access Register

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

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

If any of these operations fail, cmderr is set and none of the remaining steps are executed. An implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure.

Debug Modules must implement this command and must support accessing GPRs when the selected hart is halted. Debug Modules may optionally support accessing other registers, or accessing registers when the hart is running.



Field	Description
cmdtype	This is 0 to indicate Access Register Command.
size	2: Access the lowest 32 bits of the register. 3: Access the lowest 64 bits of the register. 4: Access the lowest 128 bits of the register. If size specifies a size larger than the register's actual size, then the access must fail. If a register is accessible, then reads of size less than or equal to the register's actual size must be supported.
postexec	When 1, execute the program in the Program Buffer exactly once after performing the transfer, if any.
transfer	0: Don't do the operation specified by write. 1: Do the operation specified by write.
write	When transfer is set: 0: Copy data from the specified register into `arg0` portion of `data`. 1: Copy data from `arg0` portion of `data` into the specified register.
regno	Number of the register to access, as described in Table 3.3. `dpc` may be used as an alias for PC if this command is supported on a non-halted hart.

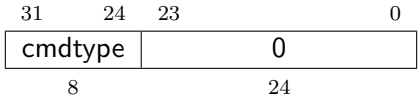
Quick Access

Perform the following sequence of operations:

1. Halt the hart. If the hart is already halted, the entire command fails.
2. Execute the Program Buffer.
3. Resume the hart. If the hart is already running, the entire command fails.

If any of these operations fail, cmderr is set and none of the remaining steps are executed. An implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure.

Implementing this command is optional.



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

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

3.6 Program Buffer

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

A debugger can write a small program to the Program Buffer, and then execute it exactly once with the Access Register Abstract Command, setting the postexec bit in `command`. If progsize is 1, the Program Buffer may only hold a single instruction. This can be a 32-bit instruction, or a compressed instruction in the lower 16 bits accompanied by a compressed `nop` in the upper 16 bits. If progsize is greater than 1, the debugger can write whatever program it likes, but the program must end with `ebreak` or `ebreak.c`.

While these programs are executed, the hart does not leave Debug Mode (see Section 4.1). If an exception is encountered during execution of the Program Buffer, no more instructions are executed, the hart remains in Debug Mode, and cmderr is set to 3 (`exception error`).

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

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

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

3.7 System Bus Access

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

Implementing a System Bus Access block has several benefits even when a Debug Module also implements a Program Buffer. First, it is possible to access memory in a running system

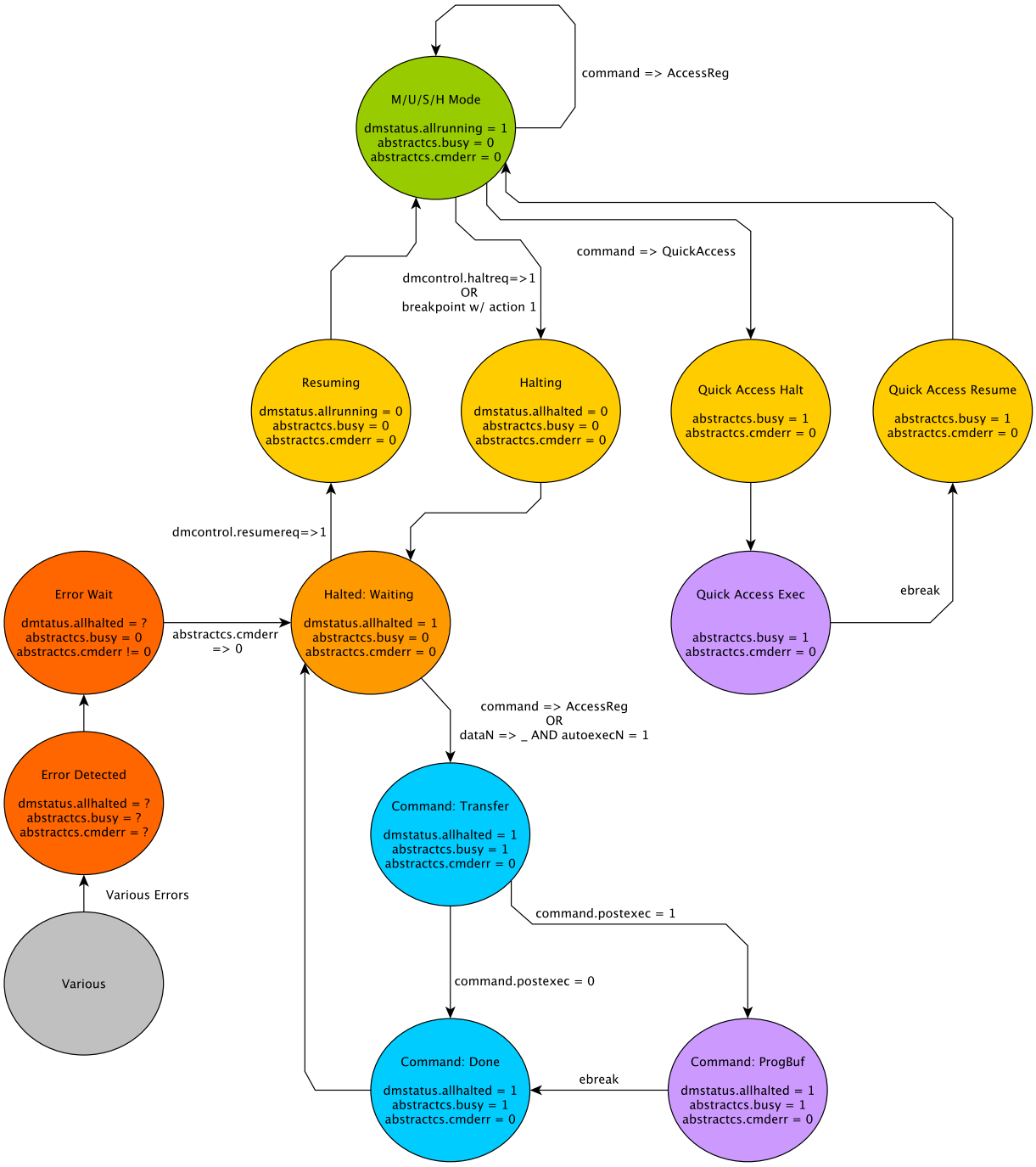


Figure 3.1: Run/Halt Debug State Machine. As only a small amount of state is visibile to the debugger, the states and transitions are conceptual.

with minimal impact. Second, it may improve performance when accessing memory. Third, it may provide access to devices that a hart does not have access to.

3.8 Quick Access

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

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

Second, the Quick Access abstract command can be used to halt a hart, quickly execute the contents of the Program Buffer, and let the hart run again. Combined with instructions that allow Program Buffer code to access the `data` registers, as described in 3.11, this can be used to quickly perform a memory or register access. For some systems this will be too intrusive, but many systems that can't be halted can bear an occasional hiccup of a hundred or less cycles.

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

3.9 Security

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

Another option is to allow the DM to be unlocked only by users who have an access key. A few bits in `dmstatus` and `authdata` can support an arbitrarily complex authentication mechanism. When authenticated is clear, the DM must not interact with the rest of the platform in any way.

3.10 Serial Ports

The Debug Module may implement up to 8 serial ports. They support basic flow control and full duplex data transfer between a component and the debugger, essentially allowing the Debug Transport to be used to communicate with a debug monitor running on a hart, or more generally emulate devices which aren't present. All these uses require software support, and are not further specified here. Only the DMI side of the Debug Module serial registers are defined in this specification as the core side interface should look like a peripheral device.

Table 3.4: Debug Module Debug Bus Registers

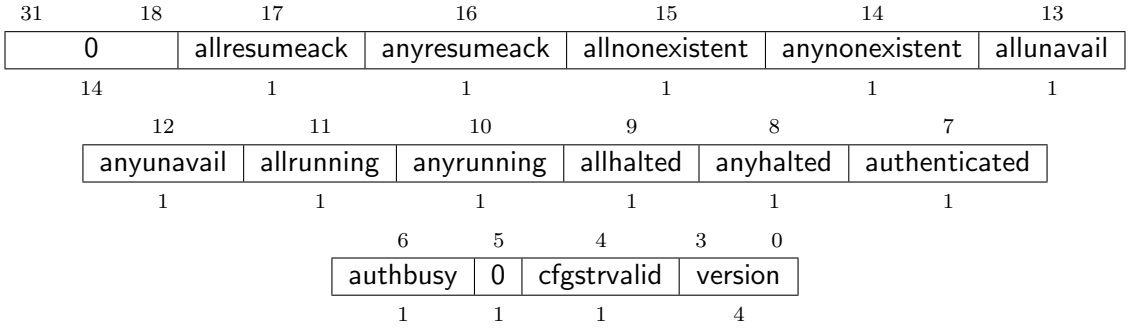
Address	Name	Page
0x04	Abstract Data 0	23
0x0f	Abstract Data 11	
0x10	Debug Module Control	17
0x11	Debug Module Status	16
0x12	Hart Info	19
0x13	Halt Summary	19
0x14	Hart Array Window Select	20
0x15	Hart Array Window	20
0x16	Abstract Control and Status	21
0x17	Abstract Command	22
0x18	Abstract Command Autoexec	22
0x19	Configuration String Addr 0	23
0x1a	Config String Addr 1	
0x1b	Config String Addr 2	
0x1c	Config String Addr 3	
0x20	Program Buffer 0	23
0x2f	Program Buffer 15	
0x30	Authentication Data	24
0x34	Serial Control and Status	24
0x35	Serial TX Data	25
0x36	Serial RX Data	25
0x38	System Bus Access Control and Status	25
0x39	System Bus Address 31:0	27
0x3a	System Bus Address 63:32	27
0x3b	System Bus Address 95:64	27
0x3c	System Bus Data 31:0	28
0x3d	System Bus Data 63:32	28
0x3e	System Bus Data 95:64	29
0x3f	System Bus Data 127:96	29

3.11 Debug Module DMI Registers

Debug Module Status (`dmstatus`, at 0x11)

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

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



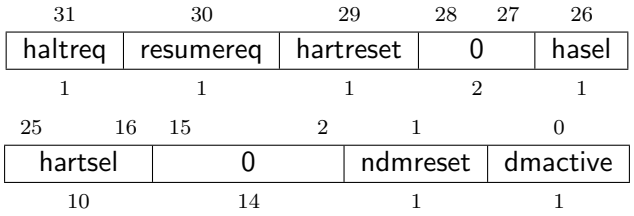
Field	Description	Access	Reset
allresumeack	This field is 1 when all currently selected harts have acknowledged the previous resumereq.	R	-
anyresumeack	This field is 1 when any currently selected hart has acknowledged the previous resumereq.	R	-
allnonexistent	This field is 1 when all currently selected harts do not exist in this system.	R	-
anynonexistent	This field is 1 when any currently selected hart does not exist in this system.	R	-
allunavail	This field is 1 when all currently selected harts are unavailable.	R	-
anyunavail	This field is 1 when any currently selected hart is unavailable.	R	-
allrunning	This field is 1 when all currently selected harts are running.	R	-
anyrunning	This field is 1 when any currently selected hart is running.	R	-
allhalted	This field is 1 when all currently selected harts are halted.	R	-
anyhalted	This field is 1 when any currently selected hart is halted.	R	-

Continued on next page

authenticated	0 when authentication is required before using the DM. 1 when the authentication check has passed. On components that don't implement authentication, this bit must be preset as 1.	R	Preset
authbusy	0: The authentication module is ready to process the next read/write to `authdata`. 1: The authentication module is busy. Accessing `authdata` results in unspecified behavior. authbusy only becomes set in immediate response to an access to `authdata`.	R	0
version	0: There is no Debug Module present. 1: There is a Debug Module and it conforms to version 0.11 of this specification. 2: There is a Debug Module and it conforms to version 0.13 of this specification.	R	2

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

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



Field	Description	Access	Reset
haltreq	Halt request signal for all currently selected harts. When set to 1, each selected hart will halt if it is not currently halted. Writing 1 or 0 has no effect on a hart which is already halted, but the bit should be cleared to 0 before the hart is resumed. Setting both haltreq and resumereq leads to undefined behavior. Writes apply to the new value of hartsel and hasel.	R/W	0

Continued on next page

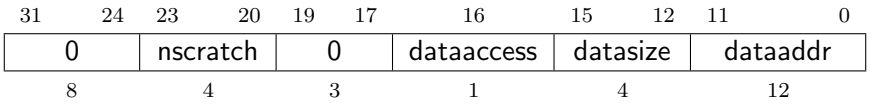
resumereq	Resume request signal for all currently selected harts. When set to 1, each selected hart will resume if it is currently halted. Setting both haltreq and resumereq leads to undefined behavior. Writes apply to the new value of hartsel and hasel.	R/W	0
hartreset	This optional bit controls reset to all the currently selected harts. To perform a reset the debugger writes 1, and then writes 0 to deassert the reset signal. If this feature is not implemented, the bit always stays 0, so after writing 1 the debugger can read the register back to see if the feature is supported. Writes apply to the new value of hartsel and hasel.	R/W	0
hasel	Selects the definition of currently selected harts. 0: There is a single currently selected hart, that selected by hartsel. 1: There may be multiple currently selected harts – that selected by hartsel, plus those selected by the hart array mask register. An implementation which does not implement the hart array mask register should tie this field to 0. A debugger which wishes to use the hart array mask register feature should set this bit and read back to see if the functionality is supported.	R/W	0
hartsel	The DM-specific index of the hart to select. This hart is always part of the currently selected harts.	R/W	0
ndmreset	This bit controls the reset signal from the DM to the rest of the system. To perform a system reset the debugger writes 1, and then writes 0 to deassert the reset. This bit must not reset the Debug Module registers. What it does reset is platform-specific (it may reset nothing).	R/W	0
dmactive	This bit serves as a reset signal for the Debug Module itself. 0: The module's state, including authentication mechanism, takes its reset values (the dmactive bit is the only bit which can be written to something other than its reset value). 1: The module functions normally. No other mechanism should exist that may result in resetting the Debug Module after power up, including the platform's system reset or Debug Transport reset signals. A debugger may pulse this bit low to get the debug module into a known state. Implementations may use this bit to aid debugging, for example by preventing the Debug Module from being power gated while debugging is active.	R/W	0

Hart Info (`hartinfo`, at 0x12)

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

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

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

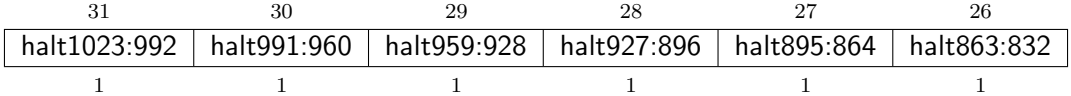


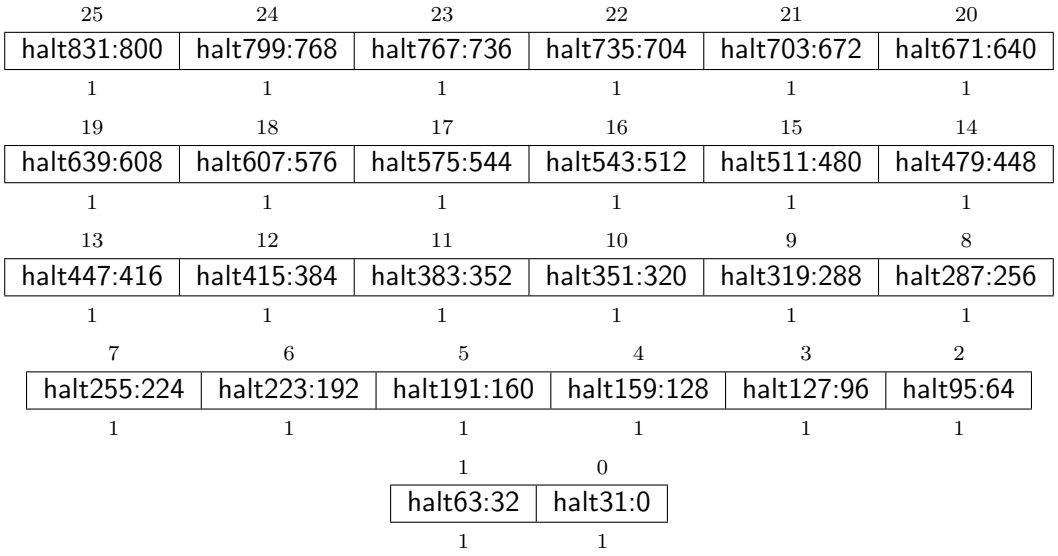
Field	Description	Access	Reset
nscratch	Number of `dscratch` registers available for the debugger to use during program buffer execution, starting from `dscratch0`. The debugger can make no assumptions about the contents of these registers between commands.	R	Preset
dataaccess	0: The `data` registers are shadowed in the hart by CSR registers. Each CSR register is XLEN bits in size, and corresponds to a single argument, per Table 3.2. 1: The `data` registers are shadowed in the hart's memory map. Each register takes up 4 bytes in the memory map.	R	Preset
datasize	If dataaccess is 0: Number of CSR registers dedicated to shadowing the `data` registers. If dataaccess is 1: Number of 32-bit words in the memory map dedicated to shadowing the `data` registers.	R	Preset
dataaddr	If dataaccess is 0: The number of the first CSR dedicated to shadowing the `data` registers. If dataaccess is 1: Signed address of RAM where the `data` registers are shadowed.	R	Preset

Halt Summary (`haltsum`, at 0x13)

This register contains a summary of which harts are halted.

Each bit contains the logical OR of 32 halt bits. When there are a large number of harts in the system, the debugger can first read this register, and then read from the halt region (0x40–0x5f) to determine which hart is the one that is halted.

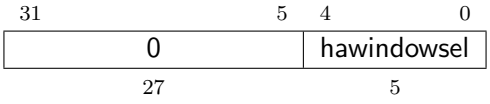




Hart Array Window Select (`hawindowsel`, at 0x14)

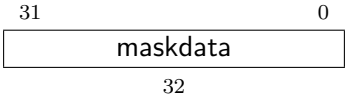
This register selects which of the 32-bit portion of the hart array mask register is accessible in `hawindow`.

The hart array mask register provides a mask of all harts controlled by the debug module. A hart is part of the currently selected harts if the corresponding bit is set in the hart array mask register and hasel in `dmcontrol` is 1, or if the hart is selected by hartsel.

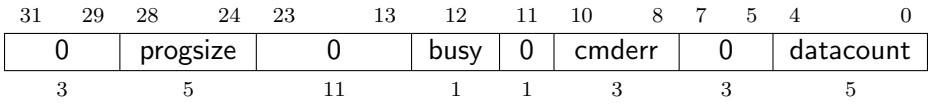


Hart Array Window (`hawindow`, at 0x15)

This register provides R/W access to a 32-bit portion of the hart array mask register. The position of the window is determined by `hawindowsel`.



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



Field	Description	Access	Reset
progsize	Size of the Program Buffer, in 32-bit words. Valid sizes are 0 - 16. TODO: Explain what can be done with each size of the buffer, to suggest why you would want more or less words.	R	Preset
busy	1: An abstract command is currently being executed. This bit is set as soon as `command` is written, and is not cleared until that command has completed.	R	0
cmderr	Gets set if an abstract command fails. The bits in this field remain set until they are cleared by writing 1 to them. No abstract command is started until the value is reset to 0. 0 (none): No error. 1 (busy): An abstract command was executing while `command`, `abstractcs`, `abstractauto` was written, or when one of the `data` or `progbuf` registers was read or written. 2 (not supported): The requested command is not supported. A command that is not supported while the hart is running may be supported when it is halted. 3 (exception): An exception occurred while executing the command (eg. while executing the Program Buffer). 4 (halt/resume): An abstract command couldn't execute because the hart wasn't in the expected state (running/halted). 7 (other): The command failed for another reason.	R/W1C	0

Continued on next page

datacount	Number of `data` registers that are implemented as part of the abstract command interface. Valid sizes are 0 - 12.	R	Preset

Abstract Command (`command`, at 0x17)

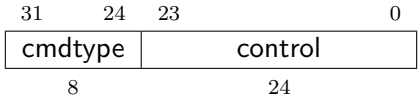
Writes to this register cause the corresponding abstract command to be executed.

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

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

---

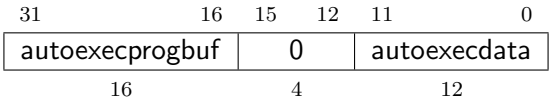
cmderr *inhibits starting a new command to accommodate debuggers that, for performance reasons, send several commands to be executed in a row without checking* cmderr *in between. They can safely do so and check* cmderr *at the end without worrying that one command failed but then a later command (which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



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

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

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



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

Continued on next page

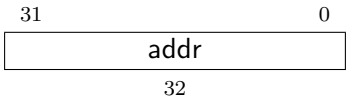
autoexecdata	When a bit in this field is 1, read or write accesses the corresponding `data` word cause the command in `command` to be executed again.	R/W	0

Configuration String Addr 0 (`cfgstraddr0`, at 0x19)

The Configuration String is described in the RISC-V Priviledged Specification. When `cfgstrvalid` is set, reading this register returns bits 31:0 of the configuration string address. Reading the other `cfgstraddr` registers returns the upper bits of the address.

When system bus mastering is implemented, this should be the address that should be used with the System Bus Access module. Otherwise, this should be the address that should be used to access the config string when hartsel =0.

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



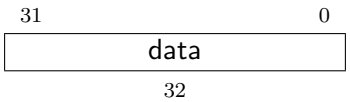
Abstract Data 0 (`data0`, at 0x04)

Basic read/write registers that may be read or changed by abstract commands.

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

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

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

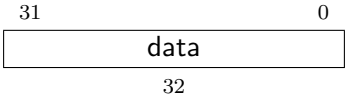


Program Buffer 0 (`progbuf0`, at 0x20)

The `progbuf` registers provide read/write access to the optional program buffer.

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

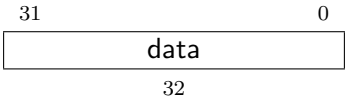
Attempts to write them while busy is set does not change their value.



Authentication Data (`authdata`, at 0x30)

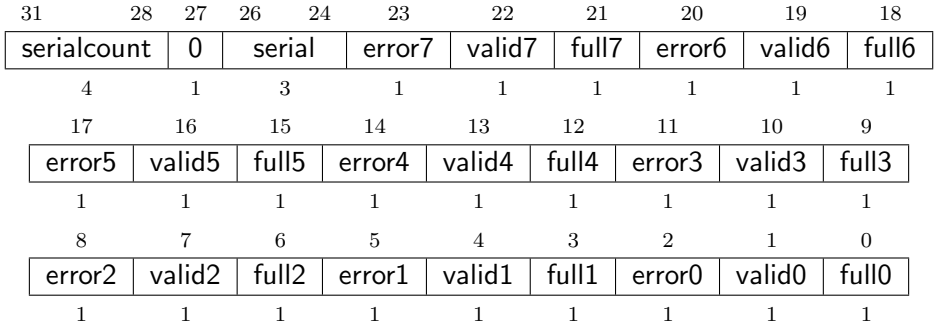
This register serves as a 32-bit serial port to the authentication module.

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



Serial Control and Status (`sercs`, at 0x34)

If serialcount is 0, this register is not present.



Field	Description	Access	Reset
serialcount	Number of supported serial ports.	R	Preset
serial	Select which serial port is accessed by `serrx` and `sertx`.	R/W	0
error0	1 when the debugger-to-core queue for serial port 0 has over or underflowed. This bit will remain set until it is reset by writing 1 to this bit.	R/W1C	0
valid0	1 when the core-to-debugger queue for serial port 0 is not empty.	R	0
full0	1 when the debugger-to-core queue for serial port 0 is full.	R	0

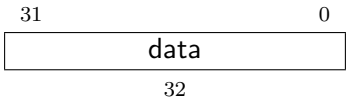
Serial TX Data (`sertx`, at 0x35)

If serialcount is 0, this register is not present.

This register provides access to the write data queue of the serial port selected by serial in `sercs`.

If the `error` bit is not set and the queue is not full, a write to this register adds the written data to the core-to-debugger queue. Otherwise the `error` bit is set and the write returns error.

A read to this register returns the last data written.

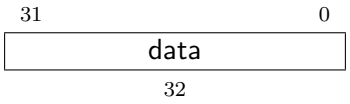


Serial RX Data (`serrx`, at 0x36)

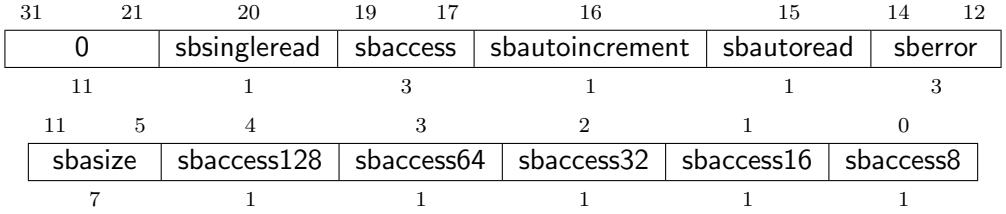
If serialcount is 0, this register is not present.

This register provides access to the read data queues of the serial port selected by serial in `sercs`.

If the `error` bit is not set and the queue is not empty, a read from this register reads the oldest entry in the debugger-to-core queue, and removes that entry from the queue. Otherwise the `error` bit is set and the read returns error.



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



Field	Description	Access	Reset
sbsingleread	When a 1 is written here, triggers a read at the address in `sbaddress` using the access size set by sbaccess.	W1	0
sbaccess	Select the access size to use for system bus accesses triggered by writes to the `sbaddress` registers or sbdata0. 0: 8-bit 1: 16-bit 2: 32-bit 3: 64-bit 4: 128-bit If an unsupported system bus access size is written here, the DM may not perform the access, or may perform the access with any access size.	R/W	2
sbautoincrement	When 1, the internal address value (used by the system bus master) is incremented by the access size (in bytes) selected in sbaccess after every system bus access.	R/W	0
sbautoread	When 1, every read from sbdata0 automatically triggers a system bus read at the new address.	R/W	0
sberror	When the debug module's system bus master causes a bus error, this field gets set. The bits in this field remain set until they are cleared by writing 1 to them. While this field is non-zero, no more system bus accesses can be initiated by the debug module. 0: There was no bus error. 1: There was a timeout. 2: A bad address was accessed. 3: There was some other error (eg. alignment). 4: The system bus master was busy when one of the `sbaddress` or `sbdata` registers was written, or the `sbdata` register was read when it had stale data.	R/W1C	0

Continued on next page

sbasize	Width of system bus addresses in bits. (0 indicates there is no bus access support.)	R	Preset
sbaccess128	1 when 128-bit system bus accesses are supported.	R	Preset
sbaccess64	1 when 64-bit system bus accesses are supported.	R	Preset
sbaccess32	1 when 32-bit system bus accesses are supported.	R	Preset
sbaccess16	1 when 16-bit system bus accesses are supported.	R	Preset
sbaccess8	1 when 8-bit system bus accesses are supported.	R	Preset

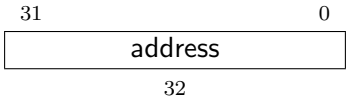
System Bus Address 31:0 (`sbaddress0`, at 0x39)

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

When the system bus master is busy, writes to this register will return error and sberror is set.

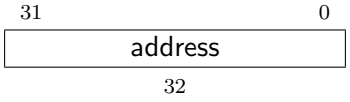
If sberror is 0 and sbautoread is set then the system bus master will start to read after updating the address from address. The access size is controlled by sbaccess in `sbcs`.

If sbsingleread is set, the bit is cleared.



Field	Description	Access	Reset
address	Accesses bits 31:0 of the internal address.	R/W	0

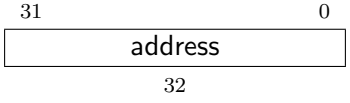
System Bus Address 63:32 (`sbaddress1`, at 0x3a)



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

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

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



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

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

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

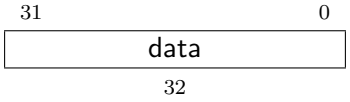
If sberror isn't 0 then accesses return error, and don't do anything else.

Writes to this register:

1. If the bus master is busy then accesses set sberror, return error, and don't do anything else.
2. Update internal data.
3. Start a bus write of the internal data to the internal address.
4. If sbautoincrement is set, increment the internal address.

Reads from this register:

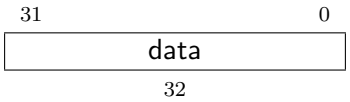
1. If bits 31:0 of the internal data register haven't been updated since the last time this register was read, then set sberror, return error, and don't do anything else.
2. "Return" the data.
3. If sbautoincrement is set, increment the internal address.
4. If sbautoread is set, start another system bus read.



Field	Description	Access	Reset
data	Accesses bits 31:0 of the internal data.	R/W	0

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

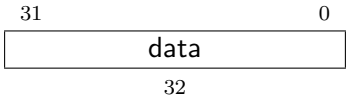
If sbaccess64 and sbaccess128 are 0, then this register is not present.



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

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

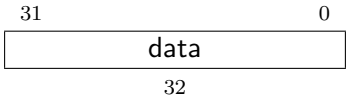
This register only exists if sbaccess128 is 1.



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

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

This register only exists if sbaccess128 is 1.



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

Chapter 4

RISC-V Debug

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

4.1 Debug Mode

Debug Mode is a special processor mode used only when the core is halted for external debugging. How Debug Mode is entered is implementation-specific.

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

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

4.2 Load-Reserved/Store-Conditional Instructions

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

4.3 Reset

If the halt signal is asserted when a core comes out of reset, the core must enter Debug Mode before executing any instructions, but after performing any initialization that would usually happen before the first instruction is executed.

4.4 Core Debug Registers

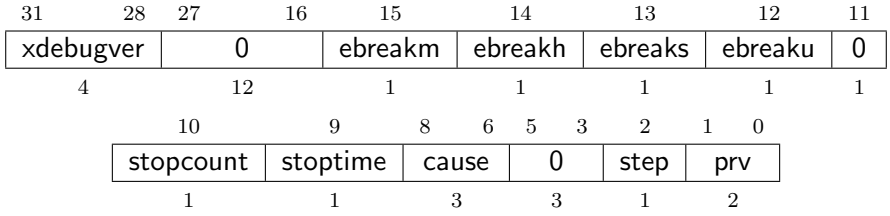
The supported Core Debug Registers must be implemented for each hart that can be debugged.

These registers are only accessible from Debug Mode.

Table 4.1: Core Debug Registers

Address	Name	Page
0x7b0	Debug Control and Status	32
0x7b1	Debug PC	34
0x7b2	Debug Scratch Register 0	
0x7b3	Debug Scratch Register 1	

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



Field	Description	Access	Reset
xdebugver	0: There is no external debug support. 4: External debug support exists as it is described in this document.	R	Preset

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
stopcount	0: Increment counters as usual. 1: Don't increment any counters while in Debug Mode. This includes the `cycle` and `instret` CSRs. This is preferred for most debugging scenarios. An implementation may choose not to support writing to this bit. The debugger must read back the value it writes to check whether the feature is supported.	R/W	Preset
stoptime	0: Increment timers as usual. 1: Don't increment any hart-local timers while in Debug Mode. An implementation may choose not to support writing to this bit. The debugger must read back the value it writes to check whether the feature is supported.	R/W	Preset
cause	Explains why Debug Mode was entered. When there are multiple reasons to enter Debug Mode in a single cycle, the cause with the highest priority is the one written. 1: An `ebreak` instruction was executed. (priority 3) 2: The Trigger Module caused a halt. (priority 4) 3: haltreq was set. (priority 2) 4: The hart single stepped because step was set. (priority 1) Other values are reserved for future use.	R	0

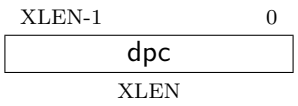
Continued on next page

step	When set and not in Debug Mode, the hart will only execute a single instruction and then enter Debug Mode. Interrupts are disabled when this bit is set. If the instruction does not complete due to an exception, the hart will immediately enter Debug Mode before executing the trap handler, with appropriate exception registers set.	R/W	0
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is described in Table 4.3. A debugger can change this value to change the hart's privilege level when exiting Debug Mode. Not all privilege levels are supported on all harts. If the encoding written is not supported or the debugger is not allowed to change to it, the hart may change to any supported privilege level.	R/W	0

Debug PC (`dpc`, at 0x7b1)

Upon entry to debug mode, `dpc` is written with the virtual address of the instruction that encountered the exception.

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



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

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

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

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

4.5 Virtual Debug Registers

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

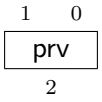
Users of the debugger shouldn't need to know about the core debug registers, but may want to change things affected by them. A virtual register is one that doesn't exist directly in the hardware, but that the debugger exposes as if it does.

Table 4.2: Virtual Core Debug Registers

Address	Name	Page
virtual	Privilege Level	35

Privilege Level (`priv`, at virtual)

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



Field	Description	Access	Reset
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is described in Table 4.3. A user can write this value to change the hart's privilege level when exiting Debug Mode.	R/W	0

Table 4.3: Privilege Level Encoding

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

Chapter 5

Trigger Module

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

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

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

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

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

5.1 Trigger Registers

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

Table 5.1: Trigger Registers

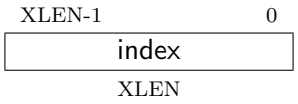
Address	Name	Page
0x7a0	Trigger Select	38
0x7a1	Trigger Data 1	38
0x7a1	Match Control	39
0x7a1	Instruction Count	43
0x7a2	Trigger Data 2	39
0x7a3	Trigger Data 3	39

Trigger Select (`tselect`, at 0x7a0)

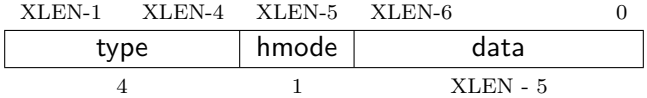
This register determines which trigger is accessible through the other trigger registers. The set of accessible triggers must start at 0, and be contiguous.

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

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



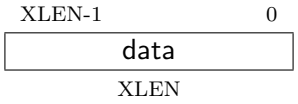
Trigger Data 1 (`tdata1`, at 0x7a1)



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

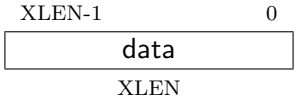
Trigger Data 2 (`tdata2`, at 0x7a2)

Trigger-specific data.



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

Trigger-specific data.



Match Control (`mcontrol`, at 0x7a1)

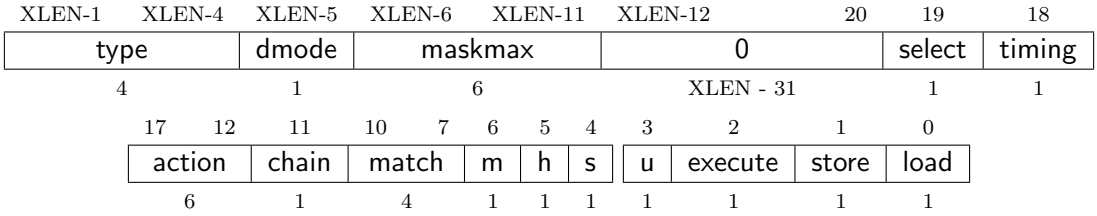
This register is accessible as `tdata1` when type is 2.

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

Address and data trigger implementation are heavily dependent on how the processor core is implemented. To accommodate various implementations, execute, load, and store address/data triggers may fire at whatever point in time is most convenient for the implementation. The debugger may request specific timings as described in timing. Table 5.2 suggests timings for the best user experience.

Table 5.2: Suggested Breakpoint Timings

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



Field	Description	Access	Reset
maskmax	Specifies the largest naturally aligned powers-of-two (NAPOT) range supported by the hardware. The value is the logarithm base 2 of the number of bytes in that range. A value of 0 indicates that only exact value matches are supported (one byte range). A value of 63 corresponds to the maximum NAPOT range, which is 2^{63} bytes in size.	R	Preset

Continued on next page

select	0: Perform a match on the virtual address. 1: Perform a match on the data value loaded/stored, or the instruction executed.	R/W	0
timing	0: The action for this trigger will be taken just before the instruction that triggered it is executed, but after all preceding instructions are are committed. 1: The action for this trigger will be taken after the instruction that triggered it is executed. It should be taken before the next instruction is executed, but it is better to implement triggers and not implement that suggestion than to not implement them at all. Most hardware will only implement one timing or the other, possibly dependent on select, execute, load, and store. This bit primarily exists for the hardware to communicate to the debugger what will happen. Hardware may implement the bit fully writable, in which case the debugger has a little more control. Data load triggers with timing of 0 will result in the same load happening again when the debugger lets the core run. For data load triggers, debuggers must first attempt to set the breakpoint with timing of 1. A chain of triggers that don't all have the same timing value will never fire (unless consecutive instructions match the appropriate triggers).	R/W	0

Continued on next page

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

Continued on next page

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

Instruction Count (`icount`, at 0x7a1)

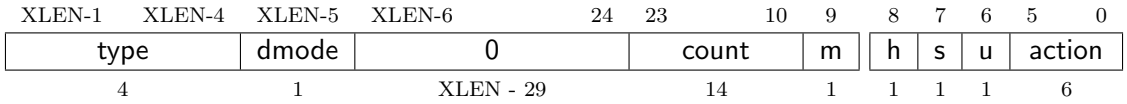
This register is accessible as `tdata1` when type is 3.

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

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

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

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



Field	Description	Access	Reset
count	When count is decremented to 0, the trigger fires. Instead of changing count from 1 to 0, it is also acceptable for hardware to clear m, h, s, and u. This allows count to be hard-wired to 1 if this register just exists for single step.	R/W	1

Continued on next page

m	When set, every instruction completed or exception taken in M mode decrements count by 1.	R/W	0
h	When set, every instruction completed or exception taken in in H mode decrements count by 1.	R/W	0
s	When set, every instruction completed or exception taken in S mode decrements count by 1.	R/W	0
u	When set, every instruction completed or exception taken in U mode decrements count by 1.	R/W	0
action	Determines what happens when this trigger matches. 0: Raise a breakpoint exception. (Used when software wants to use the trigger module without an external debugger attached.) 1: Enter Debug Mode. (Only supported when hmode is 1.) 2: Start tracing. 3: Stop tracing. 4: Emit trace data for this match. If it is a data access match, emit appropriate Load/Store Address/Data. If it is an instruction execution, emit its PC. Other values are reserved for future use.	R/W	0

Chapter 6

Debug Transport Module (DTM)

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

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

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

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

Appendix A

JTAG Debug Transport Module

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

A.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. This specification uses the latter functionality. The JTAG standard defines a Test Access Port (TAP) that can be used to read and write a few custom registers, which can be used to communicate with debug hardware in a component.

A.2 JTAG Registers

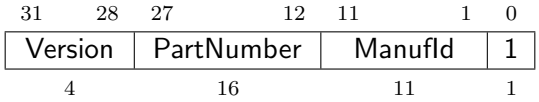
JTAG TAPs used as a DTM must have an IR of at least 5 bits. When the TAP is reset, IR must default to 00001, selecting the IDCODE instruction. A full list of JTAG registers along with their encoding is in Table A.1. If the IR actually has more than 5 bits, then the encodings in Table A.1 should be extended with 0's in their most significant bits. The only regular JTAG registers a debugger might use are BYPASS and IDCODE, but this specification leaves IR space for many other standard JTAG instructions. Unimplemented instructions must select the BYPASS register.

IDCODE (at 0x01)

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

Table A.1: JTAG DTM TAP Registers

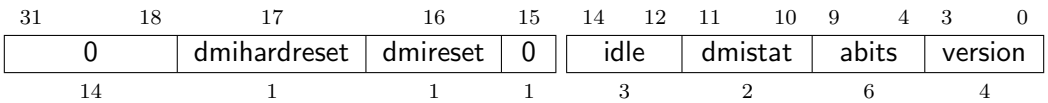
Address	Name	Description	Page
0x00	BYPASS	JTAG recommends this encoding	
0x01	IDCODE	JTAG recommends this encoding	
0x10	DTM Control and Status	For Debugging	48
0x11	Debug Module Interface Access	For Debugging	50
0x12	Reserved (BYPASS)	Reserved for future RISC-V debugging	
0x13	Reserved (BYPASS)	Reserved for future RISC-V debugging	
0x14	Reserved (BYPASS)	Reserved for future RISC-V debugging	
0x15	Reserved (BYPASS)	Reserved for future RISC-V standards	
0x16	Reserved (BYPASS)	Reserved for future RISC-V standards	
0x17	Reserved (BYPASS)	Reserved for future RISC-V standards	
0x1f	BYPASS	JTAG requires this encoding	



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

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

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



Field	Description	Access	Reset
dmihardreset	Writing 1 to this bit does a hard reset of the DTM, causing the DTM to forget about any outstanding DMI transactions. In general this should only be used when the Debugger has reason to expect that the outstanding DMI transaction will never complete (e.g. a reset condition caused an inflight DMI transaction to be cancelled).	W1	0
dmireset	Writing 1 to this bit clears the sticky error state and allows the DTM to retry or complete the previous transaction.	W1	0
idle	This is a hint to the debugger of the minimum number of cycles a debugger should spend in Run-Test/Idle after every DMI scan to avoid a 'busy' return code (dmistat of 3). A debugger must still check dmistat when necessary. 0: It is not necessary to enter Run-Test/Idle at all. 1: Enter Run-Test/Idle and leave it immediately. 2: Enter Run-Test/Idle and stay there for 1 cycle before leaving. And so on.	R	Preset

Continued on next page

dmistat	0: No error. 1: Reserved. Interpret the same as 2. 2: An operation failed (resulted in op of 2). 3: An operation was attempted while a DMI access was still in progress (resulted in op of 3).	R	0
abits	The size of address in dmi.	R	Preset
version	0: Version described in spec version 0.11. 1: Version described in spec version 0.13 (and later?), which reduces the DMI data width to 32 bits. Other values are reserved for future use.	R	1

Debug Module Interface Access (dmi, at 0x11)

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

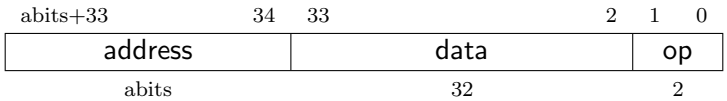
In Update-DR, the DTM starts the operation specified in op unless the current status reported in op is sticky.

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

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

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

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



Field	Description	Access	Reset
address	Address used for DMI access. In Update-DR this value is used to access the DM over the DMI.	R/W	0
data	The data to send to the DM over the DMI during Update-DR, and the data returned from the DM as a result of the previous operation.	R/W	0
op	When the debugger writes this field, it has the following meaning: 0: Ignore data and address. (nop) Don't send anything over the DMI during Update-DR. This operation should never result in a busy or error response. The address and data reported in the following Capture-DR are undefined. 1: Read from address. (read) 2: Write data to address. (write) 3: Reserved. When the debugger reads this field, it means the following: 0: The previous operation completed successfully. 1: Reserved. 2: A previous operation failed. The data scanned into dmi in this access will be ignored. This status is sticky and can be cleared by writing dmireset in dtmcs. This indicates that the DM itself responded with an error, e.g. in the System Bus and Serial Port overflow/underflow cases. 3: An operation was attempted while a DMI request is still in progress. The data scanned into dmi in this access will be ignored. This status is sticky and can be cleared by writing dmireset in dtmcs. If a debugger sees this status, it needs to give the target more TCK edges between Update-DR and Capture-DR. The simplest way to do that is to add extra transitions in Run-Test/Idle. (The DTM, DM, and/or component may be in different clock domains, so synchronization may be required. Some relatively fixed number of TCK ticks may be needed for the request to reach the DM, complete, and for the response to be synchronized back into the TCK domain.)	R/W	2

BYPASS (at 0x1f)

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

A.3 JTAG Connector

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

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

Viewing the male header from above (the pins pointing at your eye), a target's connector looks as it does in Table A.2. The function of each pin is described in Table A.3.

Table A.2: JTAG Connector Diagram

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

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

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

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

Table A.3: JTAG Connector Pinout

1	TCK	JTAG TCK signal, driven by the debug adapter. This pin must be clearly marked in both male and female headers.
5	TMS	JTAG TMS signal, driven by debug adapter.
9	TDI	JTAG TDI signal, driven by the debug adapter.
3	TDO	JTAG TDO signal, driven by the target.
8	TRSTn	Test Reset (optional, only used by some devices. Used to reset the JTAG TAP Controller).
4	VCC	Power provided by the target, which may be used to power the debug adapter. Must be able to source at least 25mA. This signal also serves as the reference voltage for logic high.
2, 10	GND	Target ground.
6	SRSTn	Active-low reset signal, driven by the debug adapter. Asserting reset should reset any RISC-V cores as well as any other peripherals on the PCB. It should not reset the debug logic. Although connecting this pin is optional, it is recommended as it allows the debugger to hold the target device in a reset state, which may be essential to debug some scenarios. If not implemented in a target, this pin must not be connected.

Appendix B

Hardware Implementations

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

B.1 Abstract Command Based

Halting happens by stalling the processor execution pipeline.

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

B.2 Execution Based

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

This method uses the processor's existing pipeline and ability to execute from arbitrary memory locations to avoid modifications to a processor's datapath. When haltreq is set, the Debug Module raises a special interrupt to the selected hart. This interrupt causes the hart to enter Debug Mode and jump to a memory region that is serviced by the DM and execute a "park loop". When taking this exception, `pc` is saved to `dpc`. In the park loop the hart writes its `mhartid` to a memory location within the Debug Module to indicate that it is halted. To allow the DM to individually control one out of several halted harts, each hart polls a specific memory location or bit in a `dscratch` CSR to determine whether the debugger wants it to continue.

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

To implement the abstract 32-bit GPR access instructions, the debugger causes the hart to execute `lw <gpr>, 0x400(zero)` or `sw 0x400(zero), <gpr>`. 64- and 128-bit accesses use `ld`/`sd` and

`lq`/`sq` respectively.

To execute the Program Buffer, the debugger causes the hart to execute `j dm_program_buffer`. When `ebreak` is executed (indicating the end of the Program Buffer code) the hart jumps back to its park loop. If an exception is encountered, the hart jumps to its debug exception address, which writes to an address in the Debug Module which indicates exception, then contains a jump back to the hart's park loop. The DM infers from the write that there was an exception, and sets cmderr appropriately.

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

Appendix C

Debugger Implementation

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

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

C.1 Debug Module Interface Access

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

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

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

C.2 Main Loop

A debugger continuously monitors `haltsum` to see if any harts have spontaneously halted.

C.3 Halting

To halt a hart, the debugger sets hartsel and haltreq. Then it waits for allhalted to become 1.

C.4 Accessing Registers

Using Abstract Command

Read `s0` using abstract command:

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

Write `mstatus` using abstract command:

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

Using Program Buffer

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

Write `mstatus` using program buffer:

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

Read `f1` using program buffer:

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

C.5 Reading Memory

Using System Bus Access

Read a word from memory using system bus access:

Op	Address	Value	Comment
Write	sbaddress0	address	
Write	sbcs	sbaccess = 2, sbsingleread	Perform a read
Read	sbdata0	-	Value read from memory

Read block of memory using system bus access:

Op	Address	Value	Comment
Write	sbaddress0	address	
Write	sbcs	sbaccess = 2, sbsingleread, sbautoread, sbautoincrement	Turn on autoread and autoincrement, and perform a read
Read	sbdata0	-	Value read from memory
Read	sbdata0	-	Next value read from memory
...	...	...	...
Write	sbcs	0	Clear sbautoread
Read	data0	-	Get last value read from memory.

Using Program Buffer

Read a word from memory using program buffer:

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

Read block of memory using program buffer:

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

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

Table C.1: Memory Read Timeline

	JTAG State	Activity
TODO	TODO	TODO

C.6 Writing Memory

Using System Bus Access

Write a word to memory using system bus access:

Op	Address	Value	Comment
Write	sbaddress0	address	
Write	sbdata0	value	

Write block of memory using system bus access:

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

Using Program Buffer

Write a word to memory using program buffer:

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

Write block of memory using program buffer:

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

C.7 Running

First, the debugger should restore any registers that it has clobbered. Once that's done, it can let the core run by setting resumereq.

C.8 Single Step

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

Using the hardware single step feature is almost the same as regular running. The debugger just sets step in dcsr before letting the core run. 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.

C.9 Handling Exceptions

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

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

C.10 Quick Access

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

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

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

Appendix D

Trace Module

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

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

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

1. Full reconstruction of all processor state, including register values etc. To achieve this goal the decoder will have to know what code is being executed, and know the exact behavior of every RISC-V instruction.
2. Reconstruct just the instruction stream. Get enough data from the trace stream that it is possible to make a list of every instruction executed. This is possible without knowing anything about the code or the core executing it.
3. Watch memory accesses for a certain memory region.

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

D.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 D.1.

Table D.1: Trace Sequence Header Packets

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

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.

D.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 D.2.

Table D.2: 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.

D.3 Synchronization

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

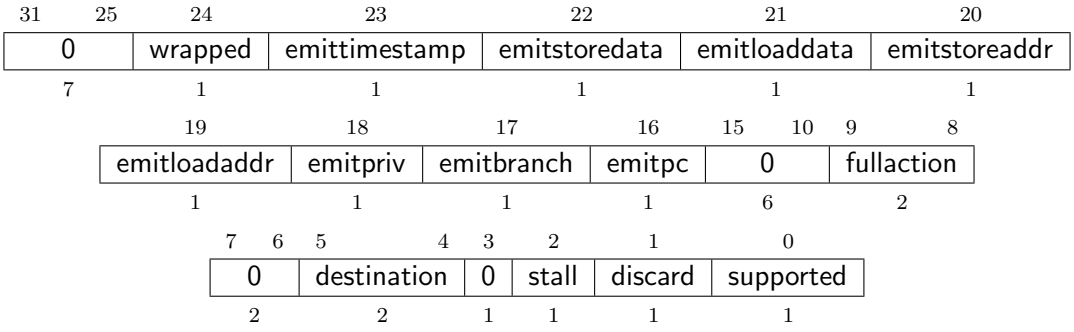
Aside from synchronizing a place in the data stream, it's also necessary to send a full PC, Read Address, Write Address, and Timestamp in order for those to be fully decoded. Ideally that happens the first time after every synchronization point, but bandwidth might prevent that. A trace source should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

D.4 Trace Registers

Table D.3: Trace Registers

Address	Name	Page
0x728	Trace	66
0x729	Trace Buffer Start	67
0x72a	Trace Buffer End	67
0x72b	Trace Buffer Write	68

Trace (`trace`, at 0x728)



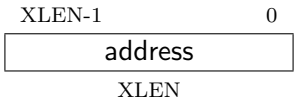
Field	Description	Access	Reset
wrapped	1 if the trace buffer has wrapped since the last time discard was written. 0 otherwise.	R	0

Continued on next page

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

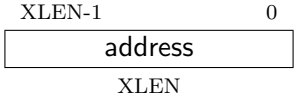
Trace Buffer Start (`tbufstart`, at 0x729)

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



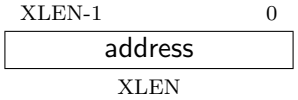
Trace Buffer End (`tbufend`, at 0x72a)

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



Trace Buffer Write (`tbufwrite`, at 0x72b)

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



Appendix E

Future Ideas

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

1. The spec defines several additions to the Device Tree which enable a debugger to discover hart IDs and supported triggers for all the cores in the system.
2. DTMs can function as general bus slaves, so they would look like regular RAM to bus masters.
3. Harts can be divided into groups. All the harts in the same group can be halted/run/stepped simultaneously. When a hart hits a breakpoint, all the other harts in the same group also halt within a few clock cycles.
4. DTMs are specified for protocols like USB, I2C, SPI, and SWD.
5. Core registers can be read without halting the processor.
6. The debugger can communicate with the power manager to power cores up or down, and to query their status.
7. Serial ports can raise an interrupt when a send/receive queue becomes full/empty.
8. The debug interrupt can be masked by running code. If the interrupt is asserted, then deasserted, and then asserted again the debug interrupt happens anyway. This mechanism can be used to eg. read/write memory with minimal interruption, making sure never to interrupt during a critical piece of code.
9. The debugger can non-intrusively sample a recent PC value from any running hart.

Appendix F

Change Log

Revision	Date	Author(s)	Description
b4f1f43	2017-06-08	Tim Newsome	Merge pull request #79 from riscv/cleanups
09c7f6e	2017-06-08	mwachs5	Merge remote-tracking branch 'origin/0.13' into cleanups
617da4c	2017-06-08	Megan Wachs	Update description of R/W1C
de2c56b	2017-06-08	Megan Wachs	Clarify that DCSR is also not updated on ebreak
efa615d	2017-06-07	Tim Newsome	Increase xdebugver field size to 4 bits. (#92)
c1b3e54	2017-06-07	Megan Wachs	Merge pull request #91 from riscv/ndmreset
5c7c1bb	2017-06-07	Tim Newsome	Merge branch '0.13' into cleanups
72bb874	2017-06-06	Megan Wachs	Merge branch '0.13' into ndmreset
1fbbe6e	2017-06-06	Megan Wachs	Merge pull request #90 from riscv/dpc_clarifications
89ffe50	2017-06-06	mwachs5	NDMRESET: Clarify what it may and may not do
1932da0	2017-06-06	mwachs5	DPC: Clarifications on its meaning
03bcafe	2017-06-06	Megan Wachs	Merge pull request #89 from riscv/datacount
6470fdb	2017-06-06	mwachs5	ABSTRACTCS: Correct inconsistency on the number of data words.
1f4a1fe	2017-06-06	Megan Wachs	Merge pull request #88 from riscv/W0_corrections
3ca82b4	2017-06-06	Megan Wachs	More corrections for R vs R/W1C on SERCS
9705fb8	2017-06-06	Megan Wachs	Correct a bunch of W0 registers
7531c41	2017-06-05	Megan Wachs	Merge pull request #80 from riscv/issue76
850bd87	2017-06-05	Megan Wachs	Merge branch '0.13' into issue76
43307eb	2017-06-05	Megan Wachs	Merge pull request #81 from riscv/issue63
989c60d	2017-06-05	Tim Newsome	Fix language. We can only halt harts, not cores.
517a08b	2017-06-05	Tim Newsome	Incorporate review feedback.
802be28	2017-06-05	Tim Newsome	Clarify/fix Quick Access example.
dbcaec8	2017-06-02	Tim Newsome	Merge branch '0.13' into cleanups
b8cc523	2017-06-02	Tim Newsome	Add included tex files as dependencies. (#78)
d0a5959	2017-06-02	Tim Newsome	Merge pull request #77 from riscv/pageno
15f864a	2017-06-01	Tim Newsome	Language cleanups, consistency and typo fixes.
4ecae86	2017-06-01	Tim Newsome	Add page numbers to list-of-register tables.
59b3e4a	2017-05-19	Megan Wachs	Setting up a Travis regression to check for build errors (#72)

124bf44	2017-05-17	mwachs5	Debug Module: CMDERR is Write-1-to clear, not R/W0
bb6c7f0	2017-05-17	mwachs5	SW Registers file should be XML, not TEX
d360358	2017-05-10	Megan Wachs (Temporary Acct.)	Remove virtual register from core_registers.xml
bfc64fb	2017-05-10	Megan Wachs (Temporary Acct.)	Add missing sw_registers.tex file
0512f5d	2017-05-06	mwachs5	Move virtual 'prv' register to a seperate section to make it more clear it is not a real register.
6b3c9d7	2017-05-06	mwachs5	Clarify haltreq/resumereq/resumack
0a487eb	2017-04-26	mwachs5	jtag: Change specified JTAG pinout from Coretex to AVR, to provide for TRSTn option.
93cdfaf	2017-04-26	mwachs5	DM : Clarify that DATA/PROGBUF can't be written while busy.
ef98f23	2017-04-19	mwachs5	jtag: Make it clear that a NOP is really a NOP.
a6f8efa	2017-04-17	mwachs5	single_step: Exceptions count as the 'step' completion.
bf11e9e	2017-04-17	mwachs5	resumeack: fix some LaTeX cross references
4afa081	2017-04-11	mwachs5	halt/resumereq: Clarify what setting them to 0 or 1 does
297a39b	2017-04-06	mwachs5	fix chisel build
082c499	2017-04-06	mwachs5	Rename resumed to resumeack, and add more text about what these bits mean.
909d617	2017-04-06	mwachs5	Correct some cross references after removing all the multiply listed registers
dd09914	2017-04-06	mwachs5	Add 'resumedall' and 'resumedany' bits to avoid race condition on about to resume and just halted
feb88fc	2017-04-05	mwachs5	JTAG DTM: Clarify that leading bits are 0 for more than 5-bit IR
75b96ea	2017-04-04	mwachs5	use renamed dm_registers file
9f3ec7e	2017-04-04	mwachs5	debugger_implementation: remove some old TODO and commentary.
45dd5b5	2017-04-04	mwachs5	Don't list out every single DM register for those that are just indexed versions
b8b3aa2	2017-04-04	mwachs5	remove core-side register definitions from Debug Module. Rename dm1 to dm
d979a13	2017-04-04	mwachs5	remove core-side serial port specification, as these should look like implementation-specific devices with appropriate drivers.
b56870b	2017-04-04	mwachs5	Remove the wording about 'debug exception', as it is called breakpoint exception in the RISC-V Spec.
1e9347d	2017-04-03	mwachs5	Add description of hasel
0dda84d	2017-04-03	mwachs5	JTAG DTM: Clean up TAP register descriptions
82ccde5	2017-04-03	mwachs5	JTAG DTM: Add a hard DMI bit which cancels the outstanding DMI transaction
bd2a3d1	2017-04-03	mwachs5	remove preexec

02c733a	2017-04-03	mwachs5	remove preexec from Abstract State diagram.
1e271d6	2017-04-03	mwachs5	Update Debugger implementation for DMI register access, and fix tex compile issues.
155dda4	2017-04-03	mwachs5	Rewrite HW Implementation examples to describe a pure abstract command approach, and to not rely on harts executing every instruciton which is fetched from the Debug Module
556c2be	2017-04-03	mwachs5	minor wording edits about RISC-V core registers
523c64a	2017-04-03	mwachs5	Edits to the Debug Module section.
b9a371f	2017-04-03	mwachs5	add missing trace.tex file.
58b2396	2017-04-03	mwachs5	Re-order the JTAG DTM Sections
a8827e2	2017-04-03	mwachs5	Edits to the System Overview.
c5417ce	2017-04-03	mwachs5	add more sections as seperate files.
287d5c6	2017-04-03	mwachs5	moving more files to seperate tex files.
9e873f4	2017-04-03	mwachs5	move trigger info into seperate file.
2c89a86	2017-04-03	mwachs5	move risc-v core debug info into seperate file.
e676491	2017-04-03	mwachs5	Move System Overview to seperate file
03df6ee	2017-04-03	mwachs5	Move Debug Module description to a seperate file.
5faa430	2017-04-03	mwachs5	add back in JTAG DTM in appendix
7b28b11	2017-04-03	mwachs5	Move jtag DTM to appendix. Move some text to commentary.
cc183ba	2017-04-03	mwachs5	move introduction to a seperate file. Comment out reading order.
2c83830	2017-04-03	mwachs5	Merge remote-tracking branch 'origin/0.13' into 0.13
e3cf6ab	2017-04-03	Megan Wachs	Merge pull request #18 from riscv/intro_edits
60c5a1c	2017-04-03	Megan Wachs	Merge branch '0.13' into intro_edits
f727d14	2017-04-03	mwachs5	Use Chapters vs Sections. Needs reorganization.
815951d	2017-04-03	mwachs5	Formatting updates. Make this look more like the RISC-V specs. Need to use chapter vs. section
69ffaf8	2017-03-31	mwachs5	Move XML files into a subdirectory.
b276384	2017-03-31	mwachs5	Remove debug_rom.S
112bbac	2017-03-31	mwachs5	figures: reorganize the figures into directories.
2d05746	2017-03-31	Megan Wachs	Merge pull request #50 from riscv/add_license
1e5c068	2017-03-27	Megan Wachs	Add LICENSE
0e2d08a	2017-03-22	Megan Wachs	Merge pull request #47 from poweihuang17/0.13
fc17730	2017-03-22	Po-wei Huang	Change some halt mode into debug mode.
8ccf029	2017-03-22	Po-wei Huang	All halt mode changed to debug mode to synchronize with the priv spec.
f143d9e	2017-03-21	mwachs5	Correct duplicated progbuf register names
0797ec1	2017-03-17	mwachs5	autoexec: make autoexec bits match the number of data words there really are.
8e76d93	2017-03-17	mwachs5	dm1_registers: move a few more things around. Reduce abstract data words back to 12.
f8bf292	2017-03-17	mwachs5	dm1_registers: resolve some address conflicts and inconsistencies
a74dff9	2017-03-17	mwachs5	access_register: some small bit changes
2e6b0ca	2017-03-15	mwachs5	config string: Fix LaTeX compile errors.

f83260a	2017-03-10	mwachs5	Abstract Commands: clarify that 32-bit reads should always work. This allows reading MISA.
6f9347a	2017-03-10	mwachs5	Config String: change the Abstract Command to DMI registers. Allow the same registers to be used for unspecified identifier information.
4ea10ff	2017-03-10	mwachs5	abstract: Make autoexec apply to all data and progbuf words. Make a seperate register which is optional.
5008436	2017-03-10	mwachs5	abstract: Allow up to 16 progbuf and/or data words. Inform debugger about dscratch registers available for its use.
aaa13e5	2017-03-06	mwachs5	Command: use the name 'cmdtype' not 'type' to allow easier auto-generation of Scala code.
e9bb72c	2017-03-06	mwachs5	Hart Array: Add registers for hart array.
5d17a35	2017-03-06	mwachs5	DM: Move addresses around for better seperation of functionalities in HW
25ccaa8	2017-03-06	mwachs5	CONTROL: Rename control and status registers to ___CS for consistency and to accurately reflect their functionality.
45cf6c2	2017-03-06	mwachs5	Errors: fix up the bit assignments in SERSTATUS with the addition of error bit.
38cb5a0	2017-03-06	mwachs5	Errors: Make errors write-1-to-clear.
b436d77	2017-03-03	mwachs5	triggers: Clarify that matches are against virtual addresses.
793bb85	2017-03-03	mwachs5	triggers: Add suggested timings for best user experience.
2669866	2017-03-03	mwachs5	stoptime/stopcycle: Make their functionality match their name. Allow any reset value.
c85a1cf	2017-03-01	mwachs5	config_string: Simplify the Config String Address abstract command.
a303a6b	2017-03-02	Megan Wachs	Update README.md
1951ae3	2017-03-01	Megan Wachs	Merge pull request #35 from sifive/generate_chisel
2e2dc28	2017-03-01	Megan Wachs	Merge pull request #34 from sifive/serial_addrs
c087c34	2017-03-01	mwachs5	Merge remote-tracking branch 'origin/0.13' into generate_chisel
92a4923	2017-03-01	mwachs5	serial: tweak addresses.
b09f460	2017-03-01	mwachs5	serial: tweak addresses.
6477837	2017-03-01	mwachs5	chisel: tweaks to class names.
be83e3e	2017-02-28	Tim Newsome	Clarify stoptime, stopcycle.
7f94662	2017-02-27	mwachs5	Merge remote-tracking branch 'origin/0.13' into generate_chisel
c17c17c	2017-02-27	Tim Newsome	Abstract command that returns config string addr.
096dfbc	2017-02-27	Tim Newsome	Acknowledge Alex.
c0253ab	2017-02-24	Tim Newsome	Explain tdata1 type a bit more.
e43ac2e	2017-02-24	Tim Newsome	Clarify how to enumerate triggers again.
c6e3e20	2017-02-23	Tim Newsome	Revert previous commit.
ef770bf	2017-02-23	Tim Newsome	mcontrol and icount mask tdata2, not tdata1.
27806f2	2017-02-23	mwachs5	rename 'type' to 'cmdtype' purely so my auto-generation scripts work.

e46798d	2017-02-22	mwachs5	Add Abstract Commands to automatic chisel
b3bb939	2017-02-21	mwachs5	Generate Chisel headers as well for Debug Module.
3d5b6f6	2017-02-22	Tim Newsome	Merge pull request #31 from sifive/abstract_command_types
c9db98c	2017-02-22	Tim Newsome	Simplify description of op statuses.
bda39cc	2017-02-22	mwachs5	Add explicit type field to Abstract Command.
34ff1d8	2017-02-22	Tim Newsome	Merge pull request #30 from sifive/more_ibuf_progbuf
f83a1ca	2017-02-22	mwachs5	Finish up replacement of ibuf-¿progbuf
ddde0a2	2017-02-22	Tim Newsome	Merge pull request #28 from sifive/inst_supply_vs_progbuf
9666e51	2017-02-22	mwachs5	IBUF-¿PROGBUF
5308ecd	2017-02-22	mwachs5	Remove last references to "Instruction Supply"
f6ebde9	2017-02-22	Tim Newsome	Move authentication to a serial protocol.
0f079c8	2017-02-22	Tim Newsome	Reserve bit for per-hart reset.
f2c93ac	2017-02-22	Tim Newsome	Clarify that dmactive resets authentication.
59154ac	2017-02-22	Tim Newsome	Merge pull request #27 from asb/clarify_reset
f5e7b1c	2017-02-22	Alex Bradbury	Clarify that the halt state of all harts is maintained through reset
3dfe8fd	2017-02-22	Tim Newsome	More Debug Mode -¿ Halt Mode.
d29fc1f	2017-02-22	Tim Newsome	Debug Mode -¿ Halt Mode
55d6030	2017-02-21	Tim Newsome	Generate debug_defines.h as part of normal make
b0e6a7f	2017-02-21	Tim Newsome	Minor clarifications.
0f9885c	2017-02-20	Tim Newsome	Various clarifications.
e443ab9	2017-02-15	Tim Newsome	Merge pull request #25 from sifive/ctrl_status
3b08e90	2017-02-15	Tim Newsome	Merge pull request #24 from sifive/sm_diagram_resumereq
0802d5a	2017-02-15	mwachs5	Use consistent 'Control and Status' naming for CS registers.
5accc7d	2017-02-15	Tim Newsome	Change all the "other" JTAG IRs to just reserved.
bcbd7da	2017-02-15	mwachs5	sm_diagram: Show using resumereq bit to resume.
18f6e55	2017-02-14	Tim Newsome	Introduce resumereq command, similar to haltreq.
fb40538	2017-02-14	Tim Newsome	Merge pull request #22 from sifive/sb_errors
4b62c40	2017-02-14	mwachs5	SystemBus: Clean up some formatting and error specification notes.
0f346e4	2017-02-14	Tim Newsome	Merge pull request #21 from sifive/sm_for_quick_access
bc97723	2017-02-14	mwachs5	quick-access: Update SM Diagram for Quick Access
d27066e	2017-02-14	Tim Newsome	Clarify haltreq bit.
6f8ec43	2017-02-14	Tim Newsome	Always generate long constants when required.
c6ac6bc	2017-02-13	Tim Newsome	Include field descriptions in C header file.
b849213	2017-02-13	Tim Newsome	Fix the build.
c82c62e	2017-02-12	Tim Newsome	Merge pull request #20 from sifive/jtag_ir_minimum
1cf8033	2017-02-12	mwachs5	jtag: More clarifications
6203bd6	2017-02-12	Megan Wachs	Update requirements– W GPRs Required
f2b43a7	2017-02-12	Megan Wachs	Remove double 'the'
2c64ef1	2017-02-12	Megan Wachs	Remove comma
f84abce	2017-02-12	Megan Wachs	Whitespace edits and address come comments

7246b44	2017-02-12	Tim Newsome	Merge pull request #19 from sifive/jtag_dtm_edits
23c2648	2017-02-11	mwachs5	jtag_dtm: ask for clarification on TAP sharing.
7020d23	2017-02-11	mwachs5	jtag_dtm: Clarifications, DBUS-¿DMI
292d49c	2017-02-11	Megan Wachs	fix indentation
55ef8d6	2017-02-11	Tim Newsome	Merge pull request #17 from sifive/prog_buffer_size
b879b86	2017-02-11	Megan Wachs	Add missing period
bbe0521	2017-02-11	mwachs5	Make comments on program buffer size match the address map.
4ceaa37	2017-02-11	mwachs5	Flesh out and edit the introduction/background Add a description of use cases this spec has in mind, and what it doesn't cover.
cbf89d6	2017-02-11	Tim Newsome	Rewrite Quick Access.
9115db1	2017-02-10	Tim Newsome	Merge pull request #16 from sifive/reduce_prog_buffer_size
170bff1	2017-02-10	Megan Wachs	Allow size 4 for the program buffer
9d46077	2017-02-10	Tim Newsome	Merge pull request #15 from sifive/dmactive
c911e6e	2017-02-10	Tim Newsome	Clarify use of dmactive.
2ca296f	2017-02-09	Tim Newsome	Reserve command register space for custom use.
e49666e	2017-02-09	Tim Newsome	Clarify hart index change per Megan's comments.
84865e9	2017-02-09	Tim Newsome	Add header prefix for abstract commands.
2434f4f	2017-02-09	Tim Newsome	Select harts by index instead of hart ID.
7bf112a	2017-02-09	Tim Newsome	Generate correct headers for ¿32-bit registers.
7f0f09a	2017-02-08	Tim Newsome	Reset dbus status to "failure" to avoid confusion.
7b1803e	2017-02-08	Tim Newsome	Merge pull request #13 from sifive/arg0_clarification
8b1c6f0	2017-02-08	Megan Wachs	Fix line wrap issue
345c33f	2017-02-08	Megan Wachs	Call out "arg0" specifically.
9f080f5	2017-02-08	Megan Wachs	Clarify "arguments" to commands
259badd	2017-02-08	Tim Newsome	Make haltsum/halt registers mandatory.
eb0f1d3	2017-02-07	Tim Newsome	Allow for early abstract command failures.
bb49bd1	2017-02-07	Tim Newsome	Clarify error handling a little.
3fc0a97	2017-02-07	Tim Newsome	Explain when abstract data regs may be clobbered.
c37167e	2017-02-07	Tim Newsome	Fix old language in description of halt registers.
6943c96	2017-02-07	Tim Newsome	Generate more useful C header files from reg defs
d7a8045	2017-02-06	Tim Newsome	Merge pull request #11 from sifive/sm_diagram
8bef40e	2017-02-05	mwachs5	Merge remote-tracking branch 'origin/0.13' into sm_diagram
98639df	2017-02-05	mwachs5	Include the SM Diagram as a figure. Also some minor capitalization fixes.
a95e4c3	2017-02-05	mwachs5	Update State Machine diagram to show uncertainty of halt bit during auto halt/resume.
ba76744	2017-02-05	Tim Newsome	Combine loabits and hiabits.
02b1d92	2017-02-05	Tim Newsome	DMI can get away with just 6 address bits.
35d6e33	2017-02-05	mwachs5	Update State machine diagram to show BUSY without HALTED
f511b05	2017-02-04	Tim Newsome	Clarify command busy bit.
a8e5ae7	2017-02-03	mwachs5	Merge remote-tracking branch 'origin/0.13' into sm_diagram
d0f8961	2017-02-03	mwachs5	Update figures

e18a68d	2017-02-03	Tim Newsome	Clarify prehalt/postresume failure.
ac3e2a9	2017-02-02	Tim Newsome	Clarify abstract command failure behavior.
ce4baee	2017-02-02	Tim Newsome	Add Quick Access section.
0490377	2017-02-02	Tim Newsome	Add prehalt and postresume to reg command.
67515bd	2017-02-02	Tim Newsome	Deal with a few minor TODOs.
96456fc	2017-02-02	Tim Newsome	Turn register names into links.
317cd98	2017-02-02	Tim Newsome	Explain what register access is required.
f3ad2f2	2017-02-01	Tim Newsome	Revert Plain Exception implementation to be simple
a0ad281	2017-02-01	Tim Newsome	execb -¿ preexec, execa -¿ postexec
1d4a2c3	2017-02-01	Tim Newsome	Limit Program Buffer sizes to 0, 1, 8.
cc40815	2017-02-01	Tim Newsome	Incorporate Po-wei's feedback.
c8b45d6	2017-02-01	Tim Newsome	Clarify how all autoexec bits work.
dbb1deb	2017-02-01	Tim Newsome	Remove stale TODO.
c5f8f59	2017-02-01	Tim Newsome	Explain why cmderr inhibits starting new commands.
5c69194	2017-02-01	Tim Newsome	Fix editing error.
50f7c48	2017-02-01	Tim Newsome	Remove empty hart info register.
781c68e	2017-02-01	Megan Wachs	Update README.md
f46b32e	2017-02-01	mwachs5	Add a diagram of Abstract Command flow.
633bd63	2017-02-01	Tim Newsome	Move Reading Order into About This Document
51ec4d1	2017-02-01	Tim Newsome	Add reading order section.
03d20ad	2017-02-01	Tim Newsome	autoexec0 applies to data0, not inst0.
c302353	2017-01-31	Tim Newsome	Don't rely on hart fetching instructions once.
2558c25	2017-01-31	Tim Newsome	Change how exceptions in Halt Mode are handled.
a36ddce	2017-01-31	Tim Newsome	Add size to abstract register command.
64de458	2017-01-31	Tim Newsome	Detail bus master reads.
c08486f	2017-01-31	Megan Wachs	reset: Add some comments (#5)
1558049	2017-01-30	Tim Newsome	Automate Change Log.
51525a4	2017-01-29	Tim Newsome	Update System Overview
7d39ac0	2017-01-29	Tim Newsome	Update Supported Features.
9e7cbea	2017-01-29	Tim Newsome	Update RISC-V Core section.
515188d	2017-01-29	Tim Newsome	Update Hardware Implementations section.
4b19ed8	2017-01-29	mwachs5	system_bus: be consistent and always call it 'System Bus'. Even if some dislike the name, we should be consistent and clear in the spec.
9ccef3d	2017-01-29	Tim Newsome	Fleshed out some debugger implementation.
04b9176	2017-01-28	Tim Newsome	Rename debug exception to breakpoint exception.
5ac4ea1	2017-01-27	Tim Newsome	WIP on big update on instruction supply.
2d9c3e2	2017-01-27	Tim Newsome	Reorganize dm registers.
de50ba8	2017-01-27	Tim Newsome	Abstract command support is already addressed.
27cb0da	2017-01-26	Tim Newsome	Merge pull request #4 from sifive/access_renames
5085046	2017-01-26	mwachs5	Rename registers and fields like 'access' that were confusingly the same name.
10bbf6f	2017-01-26	Tim Newsome	Fix #2: DM address space table
a05c582	2017-01-26	Tim Newsome	Add debugger inspection as a feature.
4062681	2017-01-24	Tim Newsome	Add publish target.
5c8bb83	2017-01-24	Tim Newsome	Clarify use of data registers.
1504da6	2017-01-24	Tim Newsome	Replace manual date with automatic git hash/date.
997f2a0	2017-01-23	Tim Newsome	Deal with unsupported abstract commands.

cb6f2b8	2017-01-23	Tim Newsome	Renumber registers to prevent duplicates.
8b4db96	2017-01-23	Tim Newsome	Don't print out addresses if they're not provided.
b00cd21	2017-01-23	Tim Newsome	Add an abstract command.
675b556	2017-01-23	Tim Newsome	Reorganize DM bits into functional group regs.
5fc7512	2017-01-23	Tim Newsome	Remove bits 33:32 from sbdata[23].
ceb5d66	2017-01-20	Tim Newsome	Starting point for a comprehensive spec