ultrasoc

RISCV Debug Specification Tutorial

May 2018

Outline

- Debug Working Group Introduction
- Why does RISC-V need an External Debug Spec?
- Overview of the Debug Spec v. 013
- Future tasks of Debug Working Group

Debug Working Group Introduction

- History of the current Specification
  - Started out as an email list
  - Task group formed August 2016
  - Regular meetings
  - Lots of discussion around two basic mechanisms
  - Took a non-binding “opinion poll” of the member companies
  - Strong desire for a unified spec
    - To avoid fragmentation of RISC-V ecosystem
    - Working group agreed to pursue that and see where it led

Debug Working Group Introduction

- History of the current Specification
  - Specification source is available in github:

    https://github.com/riscv/riscv-debug-spec
  - Pre-compiled PDFs also available in github:

    https://github.com/riscv/riscv-debug-spec

Debug Working Group Intro

- Chair
  - Megan Wachs, SiFive Inc.

- Active members from:
  - Many member companies
    - Hardware
    - Software
  - Many individuals
  - Many research institutions

Why does RISC-V need a Debug Spec?

- Essential tool for any real hardware
- Debug embedded software on simple systems
- Debug kernel issues on more complex systems
- Perform bring up and test before SW is up and running
- NOT intended to find HW faults/bugs
  - But can be used to narrow them down!

Why does RISC-V need a Debug Spec?

- “Software is King”
- Develop HW and SW debuggers that can work for any RISC-V core
- Not dependent on the quality of a Vendor’s debugging toolchain

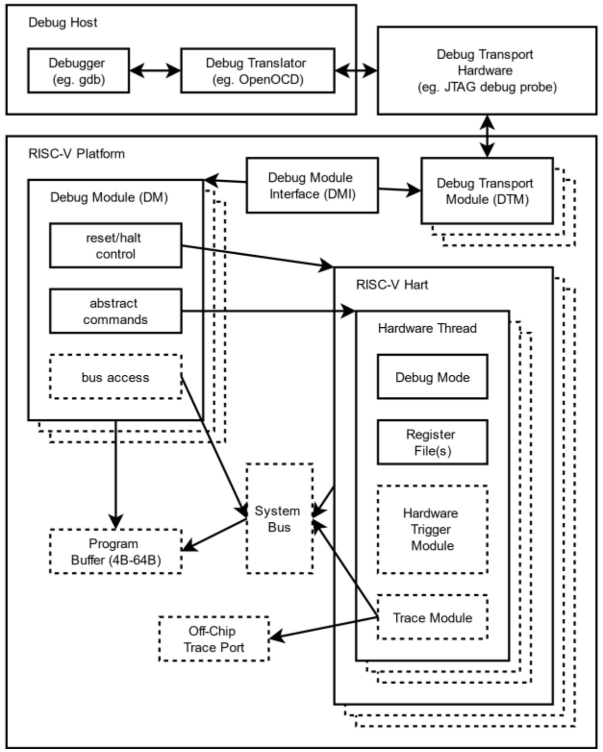
Goal of specification:

A debugger can connect “blind” to any RISC-V platform, and discover everything it needs to know.

Many features

- There are several features in the debug specification.
- Only a small set is mandatory
- A number are optional
- This allows for different implementations for different uses
  - Don’t tax implementations with features that are not needed.
- The specification does not mandate an implementation

External Debug Definition

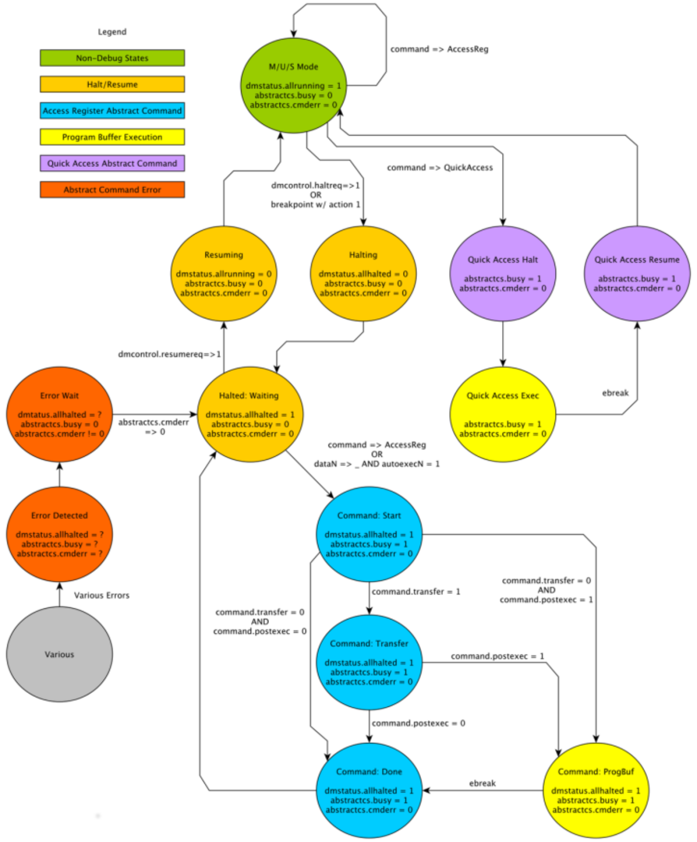


- Provide visibility into system from external hardware interface
- Access hart registers, memory, devices
- RISC-V Core and associated harts are only part of a platform, but the Spec focuses on the Harts.
- Share debug hardware between multiple harts

Debug Spec v0.13

- Basic Features
  - Selecting Harts
  - Halt - Resume
  - Abstract Commands
  - Program Buffer
  - Single - Stepping
  - Debugging across reset / power down
  - Triggers

Run/Halt State Machine



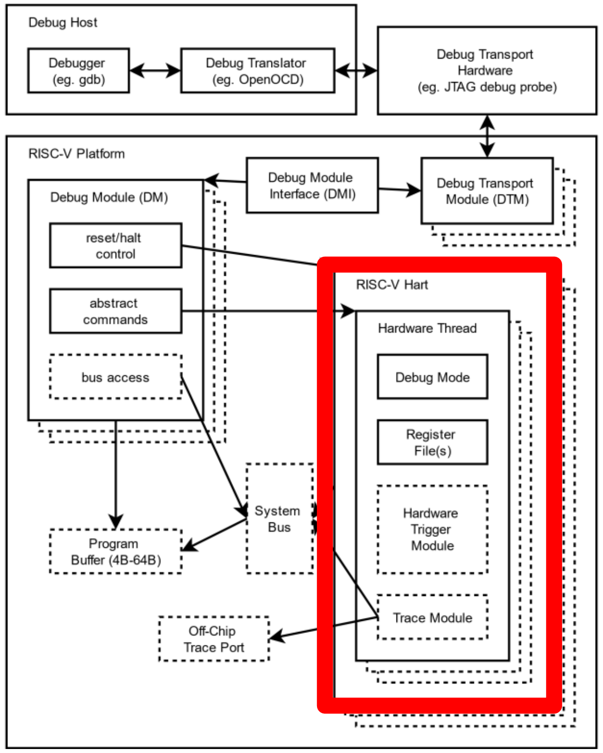
Some definitions

- `XLEN`, width of an `x` register in bits
- Hart, a hardware thread
- `WARL`. Write-Any Read-Legal field

Debugging Across Reset/Powerdown

- Debug logic reset behavior clearly specified
- Debugger can set `dmcontrol.ndmreset`
  - What parts of system are reset is implementation-specific
  - Debug logic is not affected by external reset
- Debugger must set `dmcontrol.dmactive`
  - Implementations can use this indicator to prevent power gating, etc

Requirements on RISC-V Harts



- "Debug" execution mode
  - Waiting for instruction from debugger
  - Generally acts like M-Mode
  - Interrupts are disabled
  - Exceptions handled by debugger
  - Can be implemented with simple pipeline stall

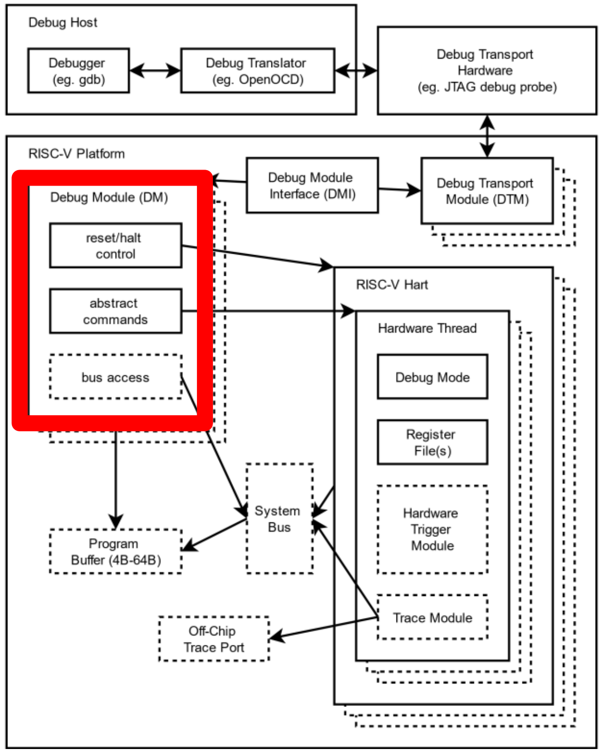
7 May 2018

Requirements on RISC-V Harts : CSRs

- `dcsr`
  - Reports cause of entering debug mode
  - Configures what ‘`ebreak`’ does
  - Controls Single-step
  - Controls counters and timers operation in debug mode

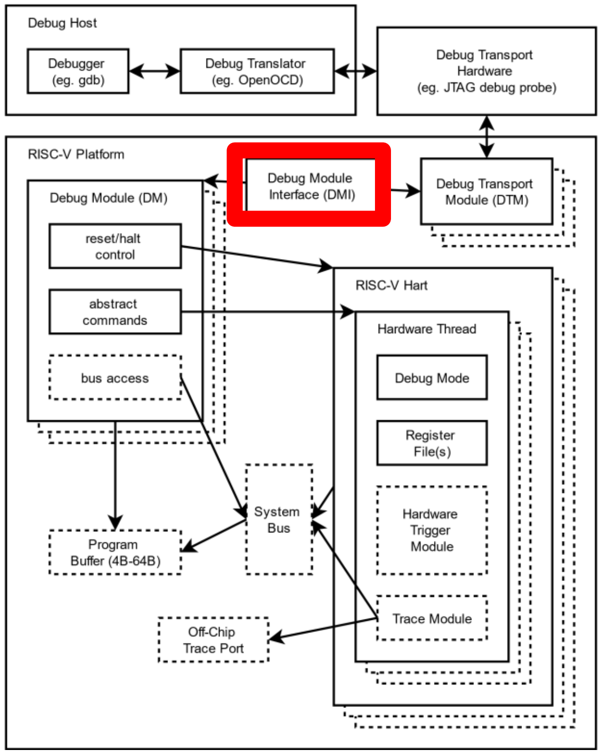
- `dpc` (`XLEN` bits, optional)
  - Holds a copy of `pc`
  - May alias to `pc`
- `dscratch`(0..N) (`XLEN` bits, optional)
  - Can be used by debug module/ debugger

Debug Module



- 32-bit register space to control debug functionality
- Specification focuses on this register space
- Details between Debug Module & Hart left up to implementation

Debug Module Interface



- Conceptual 32-bit bus
- Connects Debug Transport(s) to Debug Module
- Provides a register-based interface to Debug Module
- Abstraction: could be combined in hardware

Selecting Harts

- Single debug module can support up to 2^20 harts
- Debugger writes 20-bit hart selector to `dmcontrol.hartselhi` and `dmcontrol.hartsello`
- Hart Selector != `mhartid`, but easy to discover mapping
- Optional feature to allow selecting multiple harts

Halt, Resume

To halt, debugger:

1. Selects desired hart(s)
2. Sets `dmcontrol.haltreq`
3. Waits for `dmstatus.allhalted`
4. Clears `dmcontrol.haltreq`

To resume, debugger:

1. Selects desired hart(s)
2. Sets `dmcontrol.resumereq`
3. Waits for `dmstatus.allresumeack`

Halt multiple Harts

To halt harts 33, 37, and 5, the debugger:

1. Write 0x1 to `dmcontrol.hasel` // Select multiple hart control mode
2. Writes 0x1 to `hawindowsel` // Select second set of 32 mask bits
3. Writes 0x22 to `hawindow` // sets bits 1 and 5 to select harts 33 and 37
4. Writes 0x5 to `dmcontrol.hartsello` // select hart 5
5. Writes 0x0 to `dmcontrol.hartselhi`
6. Sets `dmcontrol.haltreq` // halt
7. Waits for `dmstatus.allhalted`
8. Clears `dmcontrol.haltreq`

Abstract Commands

- Simple Abstraction for Common Operations
    - Read/Write GPRs -- REQUIRED
    - Read/Write CSRs -- Optional
    - Read/Write FPRs -- Optional
    - Can be supported on running harts -- Optional

  To perform an abstract command:

  1. For a write command the Debugger writes argument(s) into `data` registers
  2. Debugger writes `command` register
  3. Debugger waits for `abstractcs.busy` = 0
  4. For a read command the Debugger reads results from `data` registers

Program Buffer

- Optional Extension of Abstract Command
  - Allows arbitrary RISC-V Code to be executed by hart
  - Flexible way to implement any desired behavior, including future extensions

To execute Program Buffer:

1. Write desired code into `progbuf` registers (ending with ‘`ebreak`’)
2. Write `command` register with ‘`postexec`’ bit set
3. Wait for `abstractcs.busy` = 0

Program Buffer

- Optional Extension of Abstract Command
  - Program Buffer may be addressable RAM
  - Debugger can determine this by executing programs
  - If it is, more flexibility (can use Program Buffer to pass more data)
  - `data` registers may be mapped as RAM or CSRs, reported in `hartinfo` reg.

Batching Commands

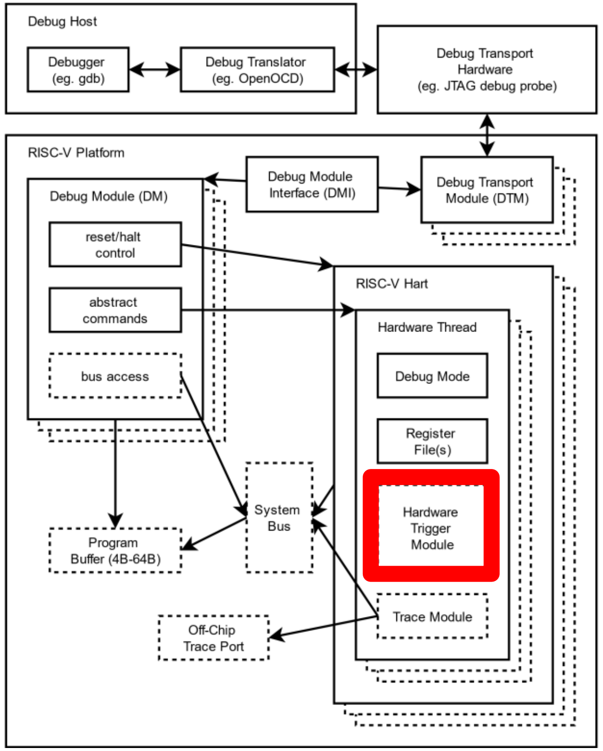
- Supported for Program Buffer and Abstract Commands
  - Debugger can write bursts of commands without checking busy
    - Check `dmstatus.cmderr` at the end, replay if necessary
  - `autoexec` functionality replays commands with different data
  - Low-overhead burst data transfers

Single Stepping

- Enabled by '`step`' bit in `dcsr`
- Once hart is resumed, hart executes a single instruction before returning to debug mode

Triggers (not part of Debug Module)

- CSRs in the core, shared with M-Mode
- Support up to 2^{XLEN} triggers
- Trigger select, trigger type: WARL fields
- 2 types of triggers currently defined:
  - Virtual address and/or data match
  - Instruction count
- Debugger polls `dmstatus.haltsum` to see if harts have halted



Additional Optional Features

- System Bus Mastering
- Quick Access

System Bus Mastering

- If `sbcs.sbasize` is > 0 then System Bus Mastering is supported
- To perform multiple 2x64 bit writes:

1. Debugger will read `sbcs` to check `sbcs.sbasize` >= 64
2. Check `sbcs.sbaccess64` is set
3. Set `sbcs.sbautoincrement` to 1
4. Write upper 32-bits of address to `sbaddress1`
5. Write lower 32-bits of address to `sbaddress0`
6. Wait until `sbcs.sbbusy` is zero
7. If `scbs.sbbusyerror` != 0 write `sbcs.sbbusyerror` to clear
8. If `scbs.sberror` != 0 write `sbcs.sberror` to clear
9. Write upper 32-bits of data to `sbdata1`
10. Write lower 32-bits of data to `sbdata0`
11. Goto 6

Quick Access Example

- Example of setting the `m` bit in `mcontrol` to enable a hardware breakpoint in M mode:
  - Write `progbuf0 transfer arg0, s0` // Save s0
  - Write `progbuf1 li s0, (1 << 6)` // Form the mask for m bit
  - Write `progbuf2 csrrs x0, tdata1, s0` // Apply the mask to mcontrol
  - Write `progbuf3 transfer s0, arg2` // Restore s0
  - Write `progbuf4 ebreak`
  - Write `command 0x10000000` // Perform quick access

Some Public Implementations of Debug v0.13

- Debugger Software (OpenOCD)
  - https://github.com/riscv/riscv-openocd
- Simulator (Spike)
  - https://github.com/riscv/riscv-isa-sim
  - debug-0.13 branch
- RTL (rocket-chip)
  - https://github.com/ucb-bar/rocket-chip
- SiFive E31/E51 Coreplex

Open Source Debugger SW: OpenOCD (1)

- Assumes JTAG DTM
- RV32 and RV64 Support
- Integrates with GDB to provide XML metadata about the target (e.g. which CSRs exist and which do not)
- Run/Halt/Single Step Support
- Accessing GPRs, FPRs, CSRs
  - With Program Buffer
  - With Abstract Commands
  - When hart is running or halted (if hart supports it)

Open Source Debugger SW: OpenOCD (2)

- Efficient Memory Access
  - With Program Buffer
  - With System Bus Access (prefers Program Buffer by default but can be overriden by user)
  - Fast batching for large reads and writes
- Supports multi-hart targets, either in “RTOS” or as individual targets
- Allows setting Hardware breakpoints and watchpoints
- Leverages OpenOCD support for things like NOR/NAND Flash programming
- Simple RISC-V specific commands:
  - Simple “compliance” commands for low level testing of adherence to spec
  - More sophisticated end-to-end tests can be found in riscv-tests repository
  - Low-level commands for basic DMI reads and writes, Authentication

Open Source Debugger SW: OpenOCD (3)

- Not Yet Supported:
  - Virtual address translation
  - Quick Access in spec

Current Focus of Debug Working Group

- 45 day consultation has elapsed
- Task Group now going through issues raised
- Incorporating feedback from the public review period to submit to board for ratification process

How to Get Involved:

- Read the spec:
  - https://github.com/riscv/riscv-debug-spec
  - https://www.sifive.com/documentation/risc-v/risc-v-external-debug-support/
- Join the mailing list:
  - https://workspace.riscv.org
  - debug@workspace.riscv.org
- Email :
  - megan@sifive.com