RISC-V external debug specification simplified

2019-09-16

1 About

An official specification for the support of external debugging on RISC-V platforms has been written. At the time of writing, the document is not complete yet. The latest release is **version 0.13.2** from 2019-03-22.

The release and the working draft are available on Github: `https://github.com/riscv/riscv-debug-spec`

This document aims to present the debug architecture and mecanisms in a simplified manner without all the details. Some parts are taken directly from the official specification.

For more detailed explanations, please refer to the official specification.

2 Overview

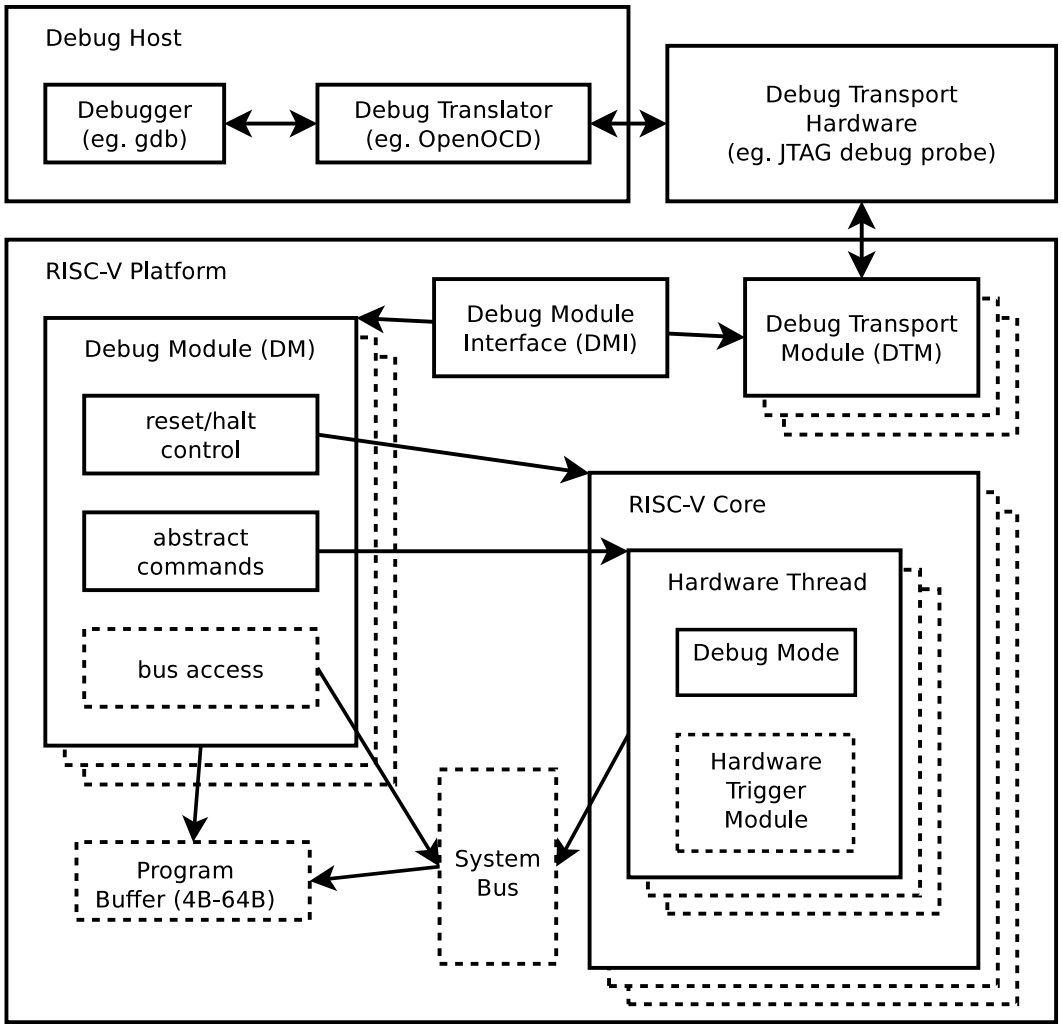

Figure 1: RISC-V Debug System Overview

The system has several elements. The ones that are external to the RISC-V platform are:

- **Debugger**: Software to test and debug the target program. gdb is the most popular. A version with RISC-V support can be found as a part of the riscv-gnu-toolchain at `https://github.com/riscv/riscv-gnu-toolchain`
- **Debug Translator**: Software to translate between the debugger and the platform. OpenOCD is often used. A fork with RISC-V support can be found at `https://github.com/riscv/riscv-openocd`
- **Debug Transport Hardware**: A way to connect the debug translator to the RISC-V platform. For example, if the system uses JTAG, it could be a JTAG probe.

In the RISC-V platform, there are:

- **Debug Transport Module (DTM)**: The access point. The DTM provides access to the DM from the exterior.
- **Debug Module Interface (DMI)**: A bus that connects the DTM to the DM.
- **Debug Module (DM)**: The unit that effectively implements the debug operations. The DM is controlled via register accesses to its DMI address space.
- **RISC-V Core**: The part of the platform that performs the instructions and computations. One or several hardware threads (**harts**) run on it. It can be interrupted and examined by the DM.

3 Debug Transport module (DTM)

A DTM provides acces to the DMs of the platform. There may be multiple DTMs, but only one can be active at the same time.

The transports used are not specified (e.g. JTAG or USB) and are left to the implementation. Nevertheless, a JTAG DTM has been defined in the specification.

Additional DTMs may be added in future versions of the specification.

3.1 JTAG DTM

The JTAG DTM defines a Test Access Port (TAP) that can be accessed using JTAG. The TAP uses the following registers:

Table 1: JTAG DTM TAP Registers

Address	Name	Description
0x00	BYPASS	JTAG recommends this encoding
0x01	IDCODE	To identify a specific silicon version
0x10	DTM Control and Status (`dtmcs`)	For Debugging
0x11	Debug Module Interface Access (`dmi`)	For Debugging
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

The *IDCODE* and *BYPASS* registers are standard JTAG registers.

Debug functionnalities are achieved through the *dtmcs* and *dmi* registers.

3.1.1 DTM Control and Status (dtmcs)

This register contains:

- **Version**: The specification version of the DTM. Possible values are 0.11, 0.13 (current), other.
- **Address size**: The size of the addresses used in *dmi*
- **Status**: Current status of the DTM (no error, error, busy)
- **Reset**: A way to reset or hard-reset the DTM

3.1.2 Debug Module Interface Access (dmi)

This register is allows access to the DM, through the DMI. It contains:

- **Address**: Address used for DMI access
- **Data**: The data to write in the DM, or the data to read after an operation
- **OP**: The operation (read, write, nop) or the result (success, fail, busy)

4 Debug Module (DM)

The DM is slave to the DMI. It implements a translation interface between abstract debug operations and their specific implementation.

4.1 Features

The DM must support the following features:

- Give the debugger necessary information about the implementation.
- Allow any individual hart to be halted and resumed.
- Provide status on which harts are halted.
- Provide abstract read and write access to a halted harts GPRs (general purpose registers).
- Provide access to a reset signal that allows debugging from the very first instruction after reset.
- At least one of these:
  - Provide a Program Buffer to force the hart to execute arbitrary instructions.
  - Allow direct System Bus Access.
  - Provide abstract access to non-GPR hart registers.
- At least one of these:
  - Implement the Program Buffer
  - Implement abstract access to all registers that are visible to software running on the hart
  - Minimal Debug Specification: Implement abstract access to at least all GPRs, *dcsr*, and *dpc*

4.2 Registers

The multiple functions are realized through the DM registers. These registers are controlled by the DMI.

When read, unimplemented Debug Module DMI Registers return 0. Writing them has no effect.

Table 2: Debug Module Debug Bus Registers

Address	Name
0x04	Abstract Data 0 (`data0`)
0x0f	Abstract Data 11 (`data11`)
0x10	Debug Module Control (`dmcontrol`)
0x11	Debug Module Status (`dmstatus`)
0x12	Hart Info (`hartinfo`)
0x13	Halt Summary 1 (`haltsum1`)
0x14	Hart Array Window Select (`hawindowsel`)
0x15	Hart Array Window (`hawindow`)
0x16	Abstract Control and Status (`abstractcs`)
0x17	Abstract Command (`command`)
0x18	Abstract Command Autoexec (`abstractauto`)
0x19	Configuration String Pointer 0 (`confstrptr0`)
0x1a	Configuration String Pointer 1 (`confstrptr1`)
0x1b	Configuration String Pointer 2 (`confstrptr2`)
0x1c	Configuration String Pointer 3 (`confstrptr3`)
0x1d	Next Debug Module (`nextdm`)
0x1f	Custom Features (`custom`)
0x20	Program Buffer 0 (`progbuf0`)
0x2f	Program Buffer 15 (`progbuf15`)
0x30	Authentication Data (`authdata`)
0x32	Debug Module Control and Status 2 (`dmcs2`)
0x34	Halt Summary 2 (`haltsum2`)
0x35	Halt Summary 3 (`haltsum3`)
0x37	System Bus Address 127:96 (`sbaddress3`)
0x38	System Bus Access Control and Status (`sbcs`)
0x39	System Bus Address 31:0 (`sbaddress0`)
0x3a	System Bus Address 63:32 (`sbaddress1`)
0x3b	System Bus Address 95:64 (`sbaddress2`)
0x3c	System Bus Data 31:0 (`sbdata0`)
0x3d	System Bus Data 63:32 (`sbdata1`)
0x3e	System Bus Data 95:64 (`sbdata2`)
0x3f	System Bus Data 127:96 (`sbdata3`)
0x40	Halt Summary 0 (`haltsum0`)
0x70	Custom Features 0 (`custom0`)
0x7f	Custom Features 15 (`custom15`)

4.3 Run Control

The DM can control the states of its harts with these registers:

- **dmstatus**: status of the DM and the selected harts
- **dmcontrol**: hart selection, halting and resuming
- **hawindow**: selecting multiple harts
- **hartinfo**: information about the selected harts
- **haltsum0-3**: which harts are halted

4.4 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.

- **progbuf0-15**: the program buffer

4.5 Abstract Commands

The DM supports a set of abstract commands:

- **Acces Register**: read or write CPU registers and execute the Program Buffer. R/W access to all GPRs is mandatory; accessing other registers is optional.
- **Quick Access** (optional): halt the hart, execute the Program Buffer, and resume the hart
- **Access Memory** (optional): R/W access in the memory available to the selected hart

The following DM registers are used:

- **abstractcs**: status of abstract commands (free, busy, error)
- **command**: select a command, its arguments, and execute it
- **abstractauto**: autoexecution of commands
- **data0-11**: arguments and return values

4.6 System Bus Access

A DM may include a System Bus Access block to provide memory access without involving a hart.

- **sbcs**: status and options
- **sbadress0-3**: set bus address and can trigger a bus read
- **sbdata0-3**: bus data (from a bus read, or for a bus write) and can trigger a bus write

5 RISC-V Core

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.

5.1 Debug Mode

When a hart is halted for external debugging, it enters Debug Mode.

In Debug Mode, the hart can execute the program Buffer with changes from normal execution:

- Machine mode privilege level
- All interrupts are masked
- Exceptions don't update any registers
- Triggers are ignored
- Counters and timers may be stopped
- Some instructions behave differently
- Effective XLEN is DXLEN
- Instruction `dret` returns from Debug Mode

5.2 Core Debug Registers

The supported Core Debug Registers must be implemented for each hart that can be debugged. They are CSRs, accessible using the RISC-V `csr` opcodes and optionally also using abstract debug commands.

These registers are only accessible from Debug Mode.

Table 3: Core Debug Registers

Address	Name
0x7b0	Debug Control and Status (`dcsr`)
0x7b1	Debug PC (`dpc`)
0x7b2	Debug Scratch Register 0 (`dscratch0`)
0x7b3	Debug Scratch Register 1 (`dscratch1`)

- **dcsr**: information about Debug Mode. It is possible to execute a single instruction (single step) by setting a field of this register.
- **dpc**: virtual address of the next instruction to be executed
- **dscratch0-1** (optional): scratch registers that can be used by implementations that need it

5.3 Trigger Module

A hart may include a Trigger Module. Triggers can cause a breakpoint exception, entry into Debug Mode, or a trace action without having to execute a special instruction.

5.3.1 Registers

The Trigger Module uses registers that are CSRs. They are accessible using the RISC-V `csr` opcodes and optionally also using abstract debug commands

Table 4: Trigger Registers

Address	Name
0x7a0	Trigger Select (`tselect`)
0x7a1	Trigger Data 1 (`tdata1`)
0x7a1	Match Control (`mcontrol`)
0x7a1	Instruction Count (`icount`)
0x7a1	Interrupt Trigger (`itrigger`)
0x7a1	Exception Trigger (`etrigger`)
0x7a2	Trigger Data 2 (`tdata2`)
0x7a3	Trigger Data 3 (`tdata3`)
0x7a4	Trigger Info (`tinfo`)

- **tselect**: select a trigger
- **tdata1-3**: trigger type and type-specific data
- **mcontrol**: operations for address/data match triggers
- **icount**: operations for instruction count triggers
- **itrigger**: operations for interrupt triggers
- **etrigger**: operations for exception triggers
- **tinfo**: information about supported trigger types