RISC-V Run Control Debug

Understanding the alternatives and proposal to proceed

Alex Bradbury, lowRISC
Stefan Wallentowitz, Open SoC Debug

Presentation overview

We hope to make the following contributions to the ongoing discussion:

- Provide a high-level overview of the options discussed so far
- Propose a path forwards towards a unified specification
- Give a brief survey of run-control debug on other architectures and SoCs

Current Goal of the Debug Task Group

Elaborate a proposal for the first debug specification

- Focus on run-control debugging
- Basic debugging capabilities to bring up a system
- Cover wide range of cores:
  Tiny deeply embedded controllers up to heavy server OoO cores
- Don’t mandate implementation details
- Provide a common specification, so RISC-V implementers can benefit from a shared investment in debug tooling
  - Achieving this goal requires a common spec, but **not** complete uniformity

Modern Debug Methodology

Run-control debugging

- Interact with the core while software is running
- Halt software, step, read/write registers (integer, fp, control & status)
- Invasive on software execution (e.g., Heisenbugs)

Trace debugging

- Sample basic signals from the core
- Aggregate to trace events: “Instruction completed”, “Cache miss”, …
- Only observes software execution: non-invasive

Interesting reads: http://www.mad-workshop.de/2013.html

Run-control options

External debug interface

- Most common approach
- Connect with debug dongle from host
- Run, inspect and control the core harts

Self-hosted debug

- The debug tool runs as software on the core itself
- Example: Simple microprocessors, use user I/O (Load flash over UART, etc.)
- Userspace debugging (e.g., gdbserver)

External Debug Interface

Provide the debug host with access to the core resources

- The channel into the core can vary
- Often includes the transport

Options for core implementation

- Halt debugging
- Feed debug instructions into the core

Self-hosted debug

- Requires very little in the way of hardware support - just the ability to set breakpoints, potentially a ‘break’ instruction. See here http://majantali.net/2016/10/how-breakpoints-are-set/ for a brief overview of how debug typically works under Linux
- There is some conceptual overlap with instruction feeding approaches for implementing external debug
- Some cores may adopt this as the primary debug method using something like gdbstub (e.g. lm32 http://devel.milkymist.narkive.com/7mZzpby2/m-labs-devel-lm32-status-jtag-debug-interface#post5)

Where we are so far

Two mainline proposals

- Differ in terms of core implementation technique
- Use different external interfaces

Halt-debugging with a memory-mapped interface (A)

Instruction feeding (B)

Sample implementations so far have used a JTAG transport.

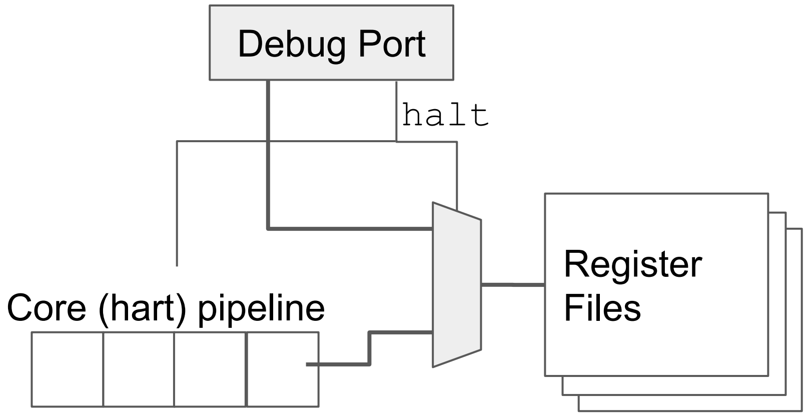
(A) Core implementation: Halt-debugging

Direct integration with the core logic

Halt the core for debug accesses

Stepping by control of pipeline

Muxing of register file(s)



(A) Interface: Memory-mapped

Functions mapped into address space

Debug interaction is a read or write access

Example: Read register 4

Host->Target: R 0x104

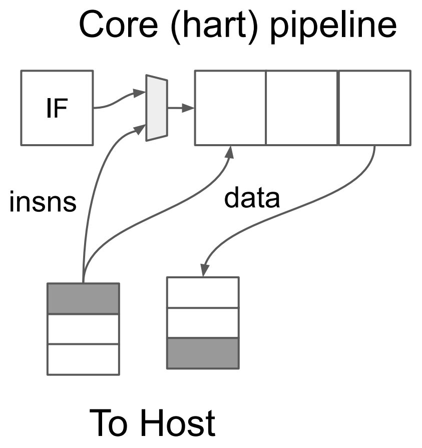
Target->Host: R4 content

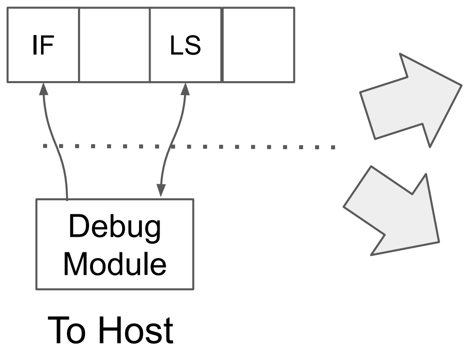
If desired, this memory-map interface could also support feeding instructions for execution by the core.

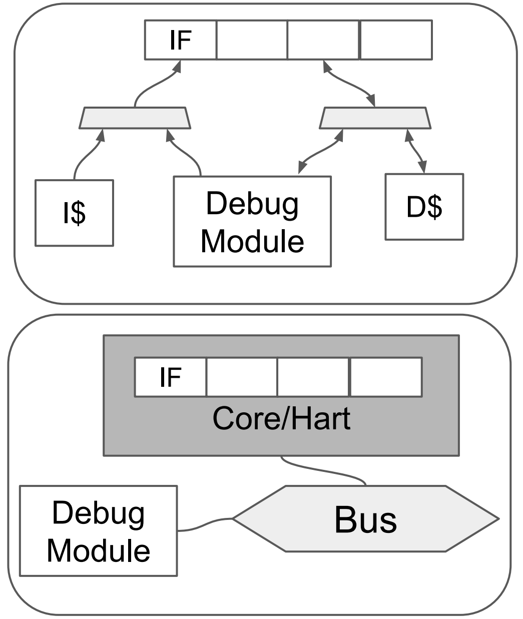
0x0	Control
0x1	Hit
0x2	Interrupt enable
0x3	Interrupt cause
0x10	HW Breakpoint #0 Control
0x11	HW Breakpoint #0 Data
...	...
0x1f	HW Breakpoint #7 Data
0x100-0x11f	Integer register file
0x400-0x71f	FP register file
0x800-0x801	NPC and PC
0x1000-0x1fff	CSRs

Similarities to ARM (), OpenRISC ()

(B) Core Implementation: Instruction Feeding







Direct/Pipeline

- Inject into pipeline
- Exchange data via CSR

Instruction Fetch

- Normal fetch, address space for debug
- Special address space reserved for communication with host
- Implementation options: tightly coupled, bus slave, in-between

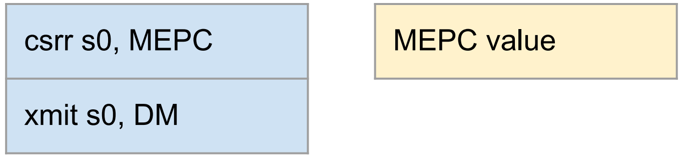
(B) Interface: Instruction Sequences

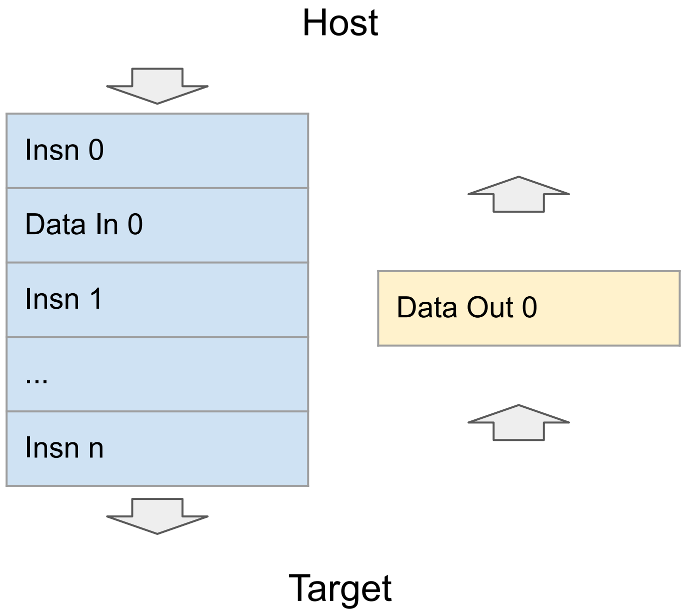
Send instructions to read/modify desired state.

Sequences contain instructions to be executed and data to be exchanged

xmit and recv are pseudo instructions, replace them on the fly with CSR access and memory access depending on implementation

Example: Read CSR MEPC





Similarities to MIPS (http://openocd.org/doc/doxygen/html/targetmips.html#ejtagacc)

Bird's-eye view on the pros and cons

Interface: memory map, **Implementation**: halt debugging

- Pros: Simplicity and clearness of interface. Memory map interface allows multiple implementation approaches, including halting and instruction feeding
- Cons: Deep core integration

Interface: Instruction feeding, **Implementation**: any

- Pros: May reduce the amount of debug-specific implementation effort when adding new extensions, likely the better implementation approach for more complex cores
- Cons: No flexibility to allow other implementation approaches (e.g. muxing register files). Doesn't allow state to be retrieved without halting the core pipeline

Can both approaches converge?

First, what do we mean with convergence?

- Beneficial if a debug software port could be reused without changes
- The spec should still be of manageable complexity
- Mandate interfaces, not implementation

For an initial approach on supporting both approaches see Stefan's proposal: *https://goo.gl/gZAejA*

Separation of Concerns

As before, separate:

- External debug interface
  - Generic interface with extensibility
  - Relevant for platform architect and debug software developer
- Core implementation technique:
  - Implementer’s discretion
  - However, a specification can ease or harden implementation, so keep in mind

Goal: define one external debug interface

Revisit the External Interface

- Base interface, physical interface is mapped to this
- Allow for other uses in the platform: As bus slave or in other debug systems
- Extensibility, in the spirit of RISC-V
- …

Want a solution that gives implementers the choice to expose new processor state via either extending the memory map or through instruction feeding.

Approach: Abstraction

Abstract from low level details and use debug messages/commands.

- Exchange messages like halt, step, set bp, read/write reg, …
- Benefit: Abstract to the layer that the debug software actually cares about
- In the memory-mapped approach messages map to mm-register accesses
- Instruction-issuing approach
  - Map messages to instruction macros (as in https://goo.gl/gZAejA)
  - Concerns about overheads by proponents of this approach

Question: Is there an advantage to this approach over providing the memory-map interface but using an instruction feeding implementation? Or is it a compromise that succeeds in keeping everyone unhappy?

Approach: Merge into one map, both optional

If overhead of mapping base debug accesses to instruction macros is no option: joint spec

Joint: Both are optional

The debug tool can query for availability

Quality host support library and clear, simple spec can help coping with extra effort

0x0	Info
0x1000-0x3fff	Base debug (optional)
0x4000-0x407f	Feeding (optional)

Approach: Merge into one map, base map

Proposed before: https://goo.gl/gZAejA

- Base map with
  - Core info and enumeration
  - Basic functionality
- Extended map for instruction feeding
  - Simple input and output map
  - Only used for extensions or advanced debug
- Problem:
  - As before: need to map the base debug to instruction macros
  - Loses some of the elegance of instruction feeding and adds overhead

0x0	Info
0x1000-0x3fff	Base debug
0x4000-0x407f	Feeding (optional)

Approach: Merge into one map, base map (cont.)

Define a set of ‘levels’ for implementers to conform to. For example:

- No debug
- Level 0: Bare minimum. Support for halting a core and reading/writing state via memory map
- Level 1: +hardware breakpoints and memory watchpoints
- Level 2: +instruction feeding
- Level 3: +trace

The above is a strawman proposal - the details can be argued.

Problem: some of these features are orthogonal (e.g. exposing an instruction feeding interface)

What else to discuss?

- Not much time left for specification
- No clear picture, some things may need to be experimented before spec’ing

Topics

- Physical interface and transport protocols
  - Strong focus on JTAG so far, but reduced pin-count interfaces are important for standalone microcontrollers
- Simple core sampling interfaces for trace debugging
- Debug Monitor: Is important while not mandatory for the proposed

Summary: Options to Proceed

We see the following options (in order of our personal preference, preferred first):

1. Memory-mapped is the baseline interface, extension instruction feeding
   - Includes message-based variant, virtually the same
2. Both memory-mapped and instruction feeding are optional
   - Up to implementer to decide what to support
3. Instruction feeding only

Appendix

Perspective

Discussions so far have focused on a tiny part of the debug problem. Although it makes sense to get this right, there's a danger of losing perspective.

In reality, debug software and SoCs implementing RISC-V will have huge differences beyond this question of 'direct' vs 'instruction' access to GPRs:

- Different debug transports and transport configurations. Low-pin interfaces may encourage different interfaces to minimise bandwidth requirements.
- Debug software initially ported to a particular SoC implementation. Functionality such as `hartid` selection is unlikely to be implemented if not needed for that device (see: current openocd port)
- Potentially large differences in available trigger/breakpoint/watchpoint functionality. To address varying requirements, and for product differentiation
- Mechanism for uploading code (e.g. writing to on-chip flash)
- Amount of state exposed and how. E.g. for custom extensions, or to expose cache information that isn't typically architecturally visible.

Extensibility

Support wide range of cores: RV32E microcontroller to RV64/RV128 high performance

Also allow for open or proprietary extensions

- Can add different classes of register files
- Can add CSRs

How to cope with the diversity? (Basically a generic RISC-V question)

Proposal: Discover as much as possible by the debug host

Extensibility: Discovery

Straight-forward option: Vendor ID:Product ID

- The tool can lookup the memory map, extra instructions, the CSR map, etc.
- Does not scale well (leads to multiple tool repositories and other nightmares)

Alternative: More modern memory map design, with machine-readable description

- Basic static part to allow very simple implementations
- Dynamic memory maps (offset fields in info, etc.)
- A machine-readable description could also specify instruction templates to retrieve state via instruction feeding

Debug survey: ARM

Support for two debug modes [1]

Halt debugging

- Conventional JTAG debugging

Self-hosted debug

- Generally used for userspace debugging
- Also used for minimal debug support. Also external? [2]

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/ch18s01s02.html
[2] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488h/way1382455546942.html

Debug survey: MIPS

- See documentation at https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MD00355-2B-24KPRG-PRG-04.63.pdf and http://openocd.org/doc/doxygen/html/targetmips.html#ejtagacc
- Instruction feeding is used. Instructions are written into the ‘dmseg’ memory space.
- A fast data transfer mode is supported

Debug survey: OpenRISC

- Uses the memory-map interface and implementation technique
- See the OpenOCD target implementation:
  - https://github.com/ntfreak/openocd/blob/master/src/target/openrisc/or1k.c
  - https://github.com/ntfreak/openocd/blob/master/src/target/openrisc/or1k_du_adv.c
  - This implementation is a good example for how one OpenOCD target could support different access methods
- The debug unit exposes multiple modules (CPU and wishbone)

Debug survey: Intel Quark

- See: Intel Quark documentation on debug http://www.intel.com/content/dam/www/public/us/en/documents/guides/quark-x1000-debug-operations-user-guide.pdf
- For debug, the processor is put into probe mode. A subset of x86 instructions are supported by the core in this mode (page 19)
  - Architectural registers are copied to an internal SRAM upon entry to probe mode (page 19)
  - The Probe Mode Instruction Register is used to submit instructions for execution
  - Do the "pseudo opcodes" for retrieving register values (page 23) correspond to real x86 instructions?
  - Reading and writing memory is done through submitting instructions (page 24)

Debug survey: MSP430

- Supports shifting in instructions and data https://github.com/travisgoodspeed/goodfet/blob/master/firmware/apps/jtag/jtag430.c http://www.ti.com/lit/ug/slau320x/slau320x.pdf
- 8-bit JTAG instructions are submitted (page 18). For some functionality, such as setting the program counter of the core, MSP430 instructions are shifted in directly over JTAG to be executed by the core.
- Reading/writing memory is done without transferring instructions to be executed by the core

Debug survey: Freescale/Motorola BDM

- Background Debug Mode (BDM), ‘background debug controller’ [1, page 145]
  - “Processor is not aware of the BDM engine, it is not seen as an exception or interrupt.” [2]
- Single-wire interface
  - High level commands/messages
  - Similar to the abstraction alternative on slide 17
- Clear separation between
  - Data that can be accessed without halting the CPU, and
  - Data which can be accessed while user programs are running
- Memory read/write instructions don’t require halting the core, but are executed “using the CPU circuitry” ([1] page 174)

[1] http://cache.freescale.com/files/microcontrollers/doc/ref_manual/HCS08RMV1.pdf
[2] http://www.intel.com/content/www/us/en/embedded/testing-and-validation/jtag-101-ieee-1149x-paper.html

Debug survey: Motorola/Freescale Extensions

- Embedded PowerPC BDM
  - Used in MPC5xx, MPC8xx
  - Similar to the instruction feeding proposal discussed here
  - Instruction feeding during a debug exception
- On Chip Emulation (onCE)
  - Internal debug support (monitor) described in “Book E” [1]
  - External debug up to implementer, here: onCE by Freescale [2]
  - Debug registers shared between software and external debug
  - Control, like stepping, via the external debug interface
  - Access registers and memory by feeding instructions
  - Good examples in AN [3]

[1] http://www.nxp.com/assets/documents/data/en/user-guides/BOOK_EUM.pdf
[2] http://www.nxp.com/assets/documents/data/en/reference-manuals/e200z3RM.pdf
[3] http://www.nxp.com/assets/documents/data/en/application-notes/AN3283.pdf

Debug survey: Xtensa (Tensilica 108Mini)

- See openocd implementation https://github.com/espressif/openocd-esp32/blob/master/src/target/esp108.c
- The debug data register (DDR) is used for communication between the core and debugger
- Instructions such as RUR (read user register) and RSR (read special register) are executed by the core to transfer processor state
- Memory accesses are also executed on the core

Debug survey: Leon/SPARC

- See documentation at http://ams.aeroflex.com/pagesproduct/datasheets/leon/UT699LEON3FTFunctionalManual.pdf (page 158)
- The ‘debug support unit’ (DSU) is accessed through the chosen debug transport (JTAG, PCI, SpaceWire, …) and has a private debug interface to the processor core(s)
- Register files, PC etc are exposed through the DSU memory map

Debug survey: Nios II

- See documentation at https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf
- The JTAG debug module has its own interface to the Nios core and doesn't sit on the system interconnect (page 11)
- Asserting the `debugreq` signal transfers execution to the 'break address'. This can occur while an exception is being handled
- Debug support is classified by 'levels'. See page 120 for an overview of the levels and LE requirements
- r25 is reserved for use by the debugger (breakpoint temporary)
- TODO: it's not clear what the interface is exposed to the host debugger, or what the code at the break address looks like. I suspect there is a debug ROM