

Debugging and Interrupt Support for Educational RISC-V Core

LinuxDays 2026

Jakub Jurčík
jurcijak@fel.cvut.cz

Pavel Píša
pisa@fel.cvut.cz

based on the bachelor thesis on the same topic by Jakub Jurčík
License: CC-BY-SA



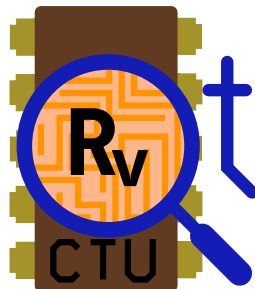
4. October, 2026

Outline

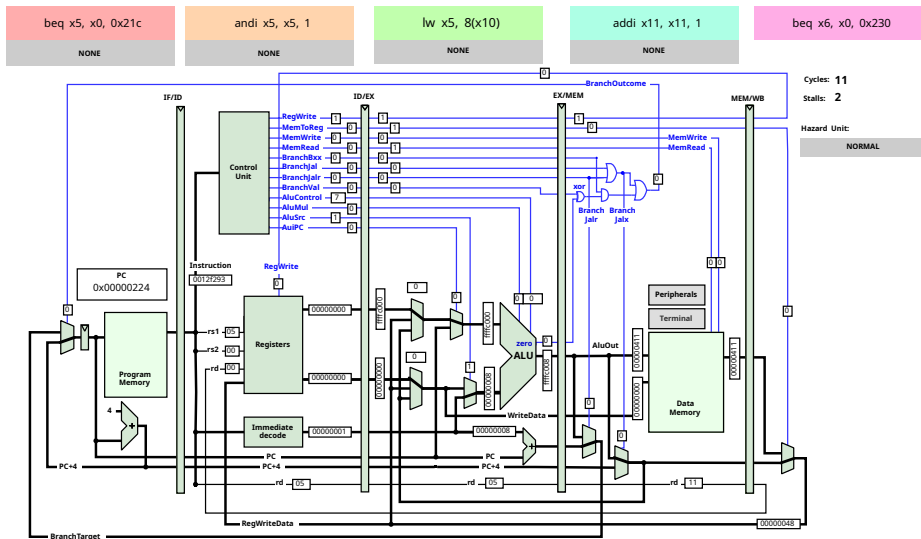
- 1 RVapo
- 2 Control and Status Registers
- 3 Exceptions and Interrupts
- 4 Debugging Support
- 5 Connecting GDB
- 6 GDB Showcase

RVapo – Educational VHDL RISC-V CPU

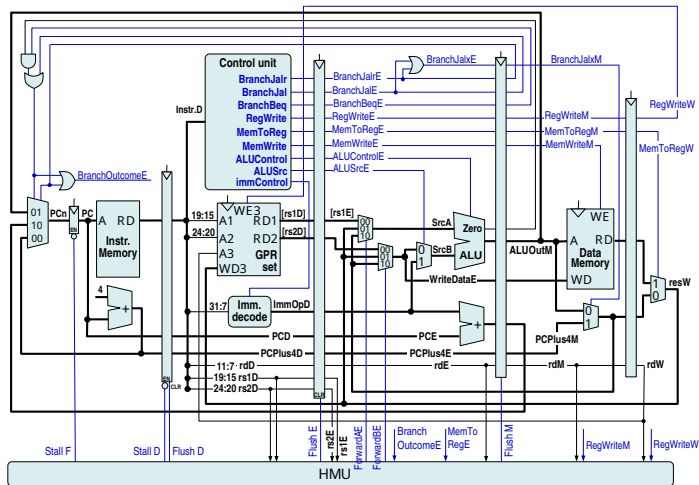
- Softcore RISC-V processor written in VHDL with 5-stage pipeline
- Authors: Eduard Lavuš, Damir Gruncl, Jakub Jurčík
- Design mirrors that of QtRvSim to provide an FPGA alternative
- Version with debugging and proper interrupts uses GHDL simulation
- <https://gitlab.fel.cvut.cz/otrees/fpga/rvapo-vhdl>



QtRvSim – five stage pipeline



Original Processor Schematic



Not exactly 1:1, taken from BIE-APS 2021 slides

Outline

- 1 RVapo
- 2 Control and Status Registers**
- 3 Exceptions and Interrupts
- 4 Debugging Support
- 5 Connecting GDB
- 6 GDB Showcase

Control and Status Registers

- Special registers defined in the Zicsr extension
- They are primarily used for privileged architectures
- However, they are necessary for proper exception/interrupt (trap) handling and debugging
- Machine mode and debug mode registers implemented

Control and Status Registers

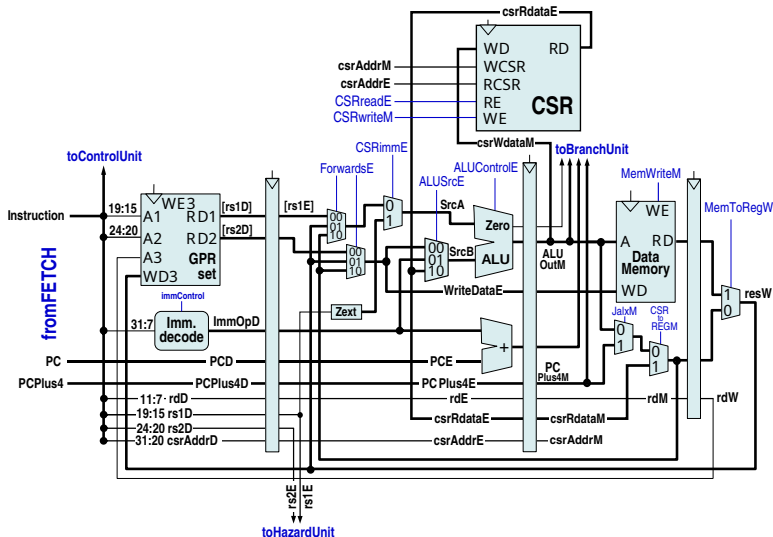
CSR	Address	Description
mstatus	0x300	Status of the core.
misa	0x301	Supported instruction set architecture.
mie	0x304	Machine interrupt enable.
mtvec	0x305	Machine trap vector.
mcountinhibit	0x320	Machine counter inhibit.
mscratch	0x340	Machine scratch register.
mepc	0x341	Machine exception program counter.
mcause	0x342	Cause of the trap.
mip	0x344	Machine interrupt pending.
mcycle	0xB00	Counter of clock cycles.
minstret	0xB02	Counter of completed instructions.

Chosen machine-mode CSRs

CSR Instructions

- The Zicsr extension introduces six new instructions performing read-modify-write on CSRs:
 - `csrrw rd, csr, rs1` - read, write
 - `csrrs rd, csr, rs1` - read, set
 - `csrrc rd, csr, rs1` - read, clear
 - `csrrwi rd, csr, imm` - read, write immediate
 - `csrrsi rd, csr, imm` - read, set immediate
 - `csrrci rd, csr, imm` - read, clear immediate
- The x0 register has special behaviour:
 - `rd = x0` → no read performed
 - `csrrs/csrrc` and `rs1 = x0` → no write performed
- It also provides pseudoinstructions for only reading or writing CSRs:
 - `csrr rd, csr`
 - `csrw csr, rs1`

Fitting CSR in the Pipeline



Handling CSR Hazards

- Implementing CSRs into the pipeline introduces new hazards
- CSR read happens in EXE, CSR write happens in MEM
- We need special behaviour for reading a CSR:
 - Read-after-write hazard with two subsequent CSR instructions
 - CSR can be modified as side effects
 - Sound forwarding would be difficult
 - Solution: one-cycle stall for CSR reads
- And also for writing a CSR:
 - Writing a CSR can cause side effects that change the behaviour of following instructions
 - Solution: pipeline flush with CSR writes
- Stall/flush is not always necessary, but doing so anyway simplifies the hardware

Handling CSR Hazards

Cycle	PC	IF	ID	EXE	MEM	WB
n	PC + 12	csrr	inst1	csrw	-	-
$n + 1$	PC + 4	inst2	csrr	inst1	csrw	-
$n + 2$	PC + 8	inst1	-	-	-	csrw
$n + 3$	PC + 12	csrr	inst1	-	-	-
$n + 4$	PC + 16	inst2	csrr	inst1	-	-
$n + 5$	PC + 16	inst2	csrr	-	inst1	-
$n + 6$	PC + 20	inst3	inst2	csrr	-	inst1
$n + 7$	PC + 24	inst4	inst3	inst2	csrr	-

Timing table for CSR hazard handling

Outline

- 1 RVapo
- 2 Control and Status Registers
- 3 Exceptions and Interrupts**
- 4 Debugging Support
- 5 Connecting GDB
- 6 GDB Showcase

Exceptions, Interrupts, Traps

- Exceptions - synchronous anomalous events that disrupt the execution of instructions and need special handling
 - Illegal instruction, unknown opcode
 - Various errors, misaligned or invalid address, page faults, ...
 - Special instructions: `ecall` and `ebreak`
 - Exceptions are always taken and handled
- Interrupts - asynchronous events that interrupt the execution of instructions
 - MEI - Machine external interrupt
 - MTI - Machine timer interrupt
 - MSI - Machine software interrupt
 - Normal interrupts can be masked with the `mie` register (and `mstatus.MIE` bit)
- In the RISC-V context, exceptions and interrupts are called **traps**

Precise Traps

- RVapo implements **precise traps** - the processor ensures both that all instruction preceding the trap commit, and none of the succeeding instructions do
- Code execution flow is unchanged from the point of view of the program (apart from timing delays or when it's desired to be observable)
- To achieve this, all traps are taken to the handler in the commit stage, in our case MEM
- Each stage up to MEM can detect a trap, and mark the current instruction as trapped
- When a trapped instruction reaches MEM, its effects are suppressed, and the Trap Unit transfers control to the handler via the Branch Unit
- Pending interrupt raises a flag that tags instructions entering the pipeline

Trap Handling

- Traps are handled by the generic trap handler starting at address `mtvec`
- The highest bit of `mcause` encodes whether the trap is an exception or interrupt
- Hardware only takes care of jumping to/from the generic handler
- Trap entry:
 - Set `mstatus.MPIE` to `mstatus.MIE`
 - Clear `mstatus.MIE` to disable interrupts
 - Set `mepc` to the trapped instruction's address
 - Set `mcause` to the cause of the trap
 - Set `mtval` to the trap-causing value
 - Jump to the handler at `mtvec`
- Trap return (`mret`):
 - Set `mstatus.MIE` to `mstatus.MPIE`
 - Set `mstatus.MPIE` to one
 - Jump back to address saved in `mepc`
- The remaining duties, ie. saving and restoring registers, is done by software

Trap Handler Example

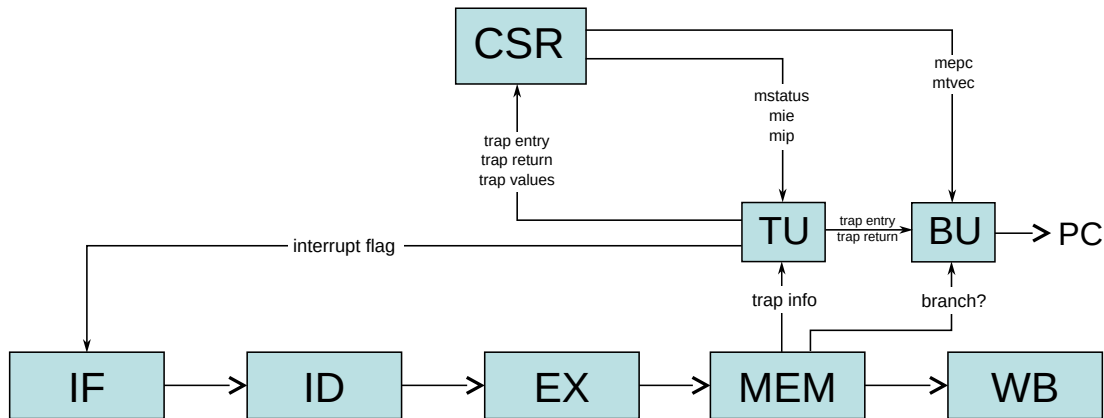
```

trap_handler:
    addi    sp, sp, -64           # make space on the stack
    sw      ra, 0(sp)            # save registers to the stack
    sw      t0, 4(sp)
    sw      t1, 8(sp)
    ...
    csrr    t0, mcause           # interrupt or exception?
    blt     t0, zero, irq_handler # branch to interrupt handler
    ...
    # handle exception

trap_return:
    lw      t1, 8(sp)            # restore registers from the stack
    lw      t0, 4(sp)
    lw      ra, 0(sp)
    addi    sp, sp, 64
    mret                          # return from the handler (pc <= mepc)

```

Trap Logic Block Schematic



Interrupt Example

Cycle	PC	IF	ID	EXE	MEM	WB
n	PC	inst4	lw	inst2	inst1	-
$n + 1$	PC+4	INT	inst4	lw	inst2	inst1
$n + 2$	PC+8	-	INT	inst4	lw	inst2
$n + 3$	PC+8	-	INT	inst4	lw	inst2
$n + 4$	PC+8	-	INT	inst4	lw	inst2
$n + 5$	PC+12	-	-	INT	inst4	lw
$n + 6$	mtvec	-	-	-	INT	inst4
$n + 7$	mtvec+4	hdlr1	-	-	-	-
$n + 8$	mtvec+8	hdlr2	hdlr1	-	-	-

Timing table for an interrupt after a multicycle instruction

Outline

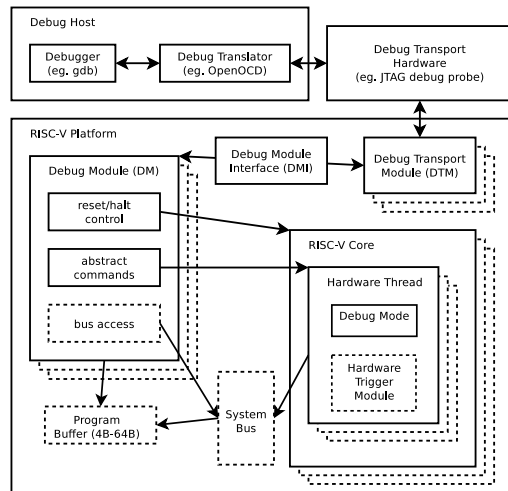
- 1 RVapo
- 2 Control and Status Registers
- 3 Exceptions and Interrupts
- 4 Debugging Support**
- 5 Connecting GDB
- 6 GDB Showcase

What a Debugger Needs

- In general, any debugger requires the core to support at least these features:
 - Halt and resume the core
 - Single step the core
 - Read/write General Purpose Registers (GPRs)
 - Read/write Control and Status Registers (CSRs)
 - Read/write memory
 - Set breakpoints
- The RISC-V Debug Specification introduces the Debug Module (DM) along others for this
- Debugging is currently only available in GHDL simulation

Debugging Flowchart

- The design features a single DM debugging a single Hardware Thread (hart)
- Hart selection mechanisms were omitted
- Debug Transport Hardware and Debug Transport Module replaced by a multithreading handshake (explained later)
- Program Buffer approach was chosen in order to reuse the existing pipeline whenever possible



Debug Module

- The Debug Module sits between the debugger and the core
- The debugger controls and inspects the state of the core using DM registers by writing either **DM Commands** or **Abstract Commands**
- Supported features are:
 - Allow the core to be halted and resumed
 - Provide status on whether the core is halted
 - Provide read/write access to GPRs
 - Provide access to a reset signal
 - Provide a Program Buffer for executing arbitrary instructions
- Everything else (memory access, CSR access) is done by the Program Buffer

Debug Module Registers and Debug CSRs

Name	Description
dmcontrol	Debug Module Control
dmstatus	Debug Module Status
command	Abstract command
abstractcs	Abstract command status
data0	Data register for passing values
progbuf0/1	Program buffer for issuing instructions
dcsr	Debug Control and Status
dpc	Debug PC
dmdata0	Debug Module data0 alias

DM registers and debug CSRs

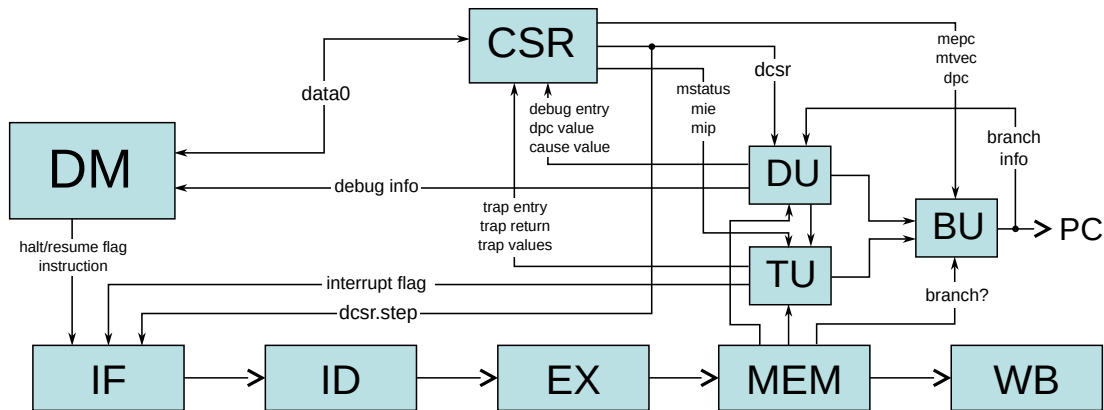
General Workflow Overview

- Typically, the debugger operations follow a similar pattern:
- Core control (halt, resume, reset):
 - Write desired command into `dmcontrol`
 - Poll `dmstatus` until the effect takes place
- State inspection or modification:
 - If writing, prepare `data0` register with data
 - Prepare `progbuf0/1` with instructions when necessary
 - Write desired command into `command`
 - Poll `abstractcs` until its `busy` bit clears
 - Check `abstractcs`'s `cmderr` field for any errors
 - If reading, read `data0` for the result

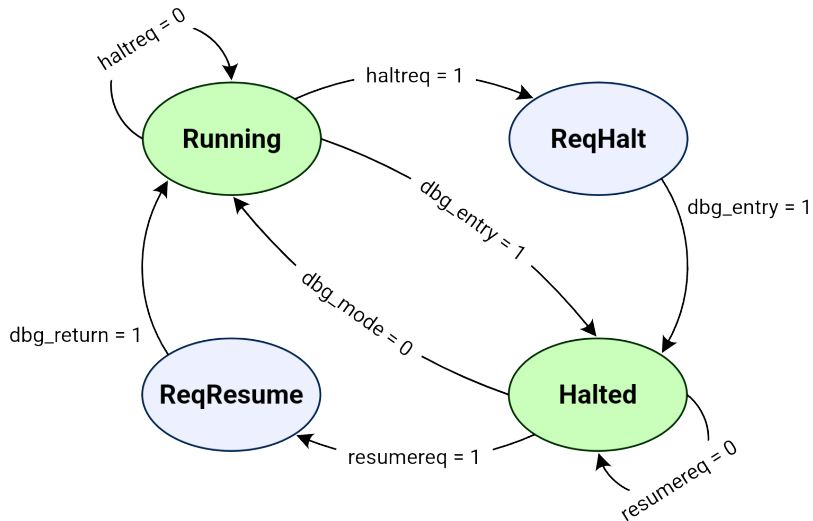
Controlling the Core

- When halted, NOPs are injected into the pipeline rather than stalling the pipeline
- This lets us issue arbitrary instructions and pass tags to later stages
- There are multiple ways to halt the core:
- Writing `dmcontrol`:
 - The DM tags instructions entering the pipeline with halt/resume request flags, similar to the interrupt request flag
 - It waits until the tagged instruction reaches MEM
 - In MEM, the Debug Unit sees the halt/resume flag and acts accordingly
- Breakpoint:
 - If enabled in `dcsr.ebreakm`, an `ebreak` instruction will halt the core once it reaches MEM
 - The debugger sets breakpoints using the `ebreak` instruction
- Single-step:
 - If `dcsr.step` is set, after resuming, the core executes one instruction and halts again
- Resetting is possible using the `ndmreset` bit of `dmcontrol`

Debugging Logic Block Schematic



Core Control State Machine



Abstract Commands

- The debugger can inspect and modify state of the core via Abstract Commands
- The Debug Specification defines three Abstract Commands:
 - Access registers (required)
 - Quick access (optional)
 - Memory access (optional)
- Only the required *Access registers* command is implemented, since it is sufficient thanks to the Program Buffer
- The command executes like so:
 - 1 If `transfer = 1` and `write = 0`, copy data from register `regno` into DM register `data0`
 - 2 If `transfer = 1` and `write = 1`, copy data from DM register `data0` into register `regno`
 - 3 If `postexec = 1`, execute the Program Buffer

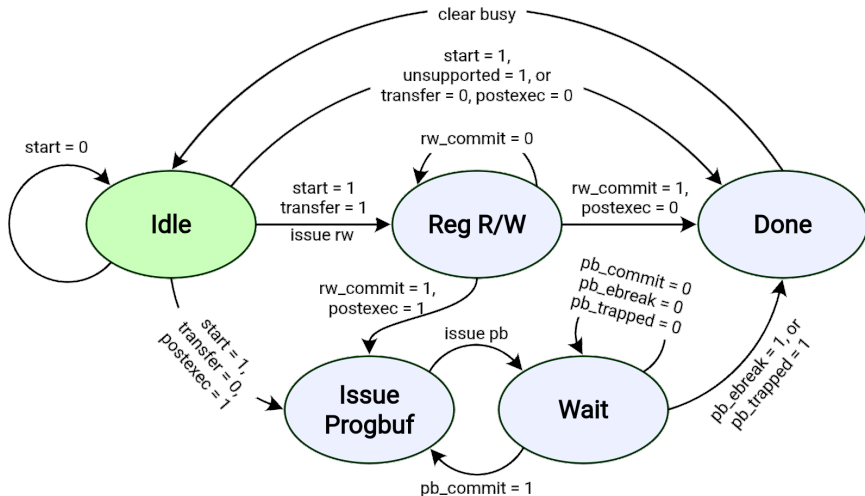
CSR data0 Alias

- The data0 alias gives the pipeline access to the register as if it were a standard CSR
- This alias allows the DM to read and write GPRs with almost no additional hardware
- The CSR block does not own the data0 register
- To read or write a register, the DM issues standard CSR instructions:
 - GPR read: `csrw data0, rs` - the GPR value is written to data0
 - GPR write: `csrr rd, data0` - the value in data0 is written to the GPR
- After issuing the instruction, the DM waits until it commits and then continues Abstract Command execution

Program Buffer

- The Program Buffer is a place in the DM where the debugger can write small programs to then execute on the core
- It is implemented directly as DM registers, as opposed to a section of RAM
- Doing so lets us use the same path that is used for injecting the instructions for GPR reads
- Each instruction injected into the pipeline must commit before the next one can start
- If an injected instruction causes an exception, the command stops and reports it in `cmderr`
- Program Buffer is used to read/write CSRs and memory → for breakpoints to work, instruction memory must be reachable through the data path (von Neumann)

Abstract Commands State Machine



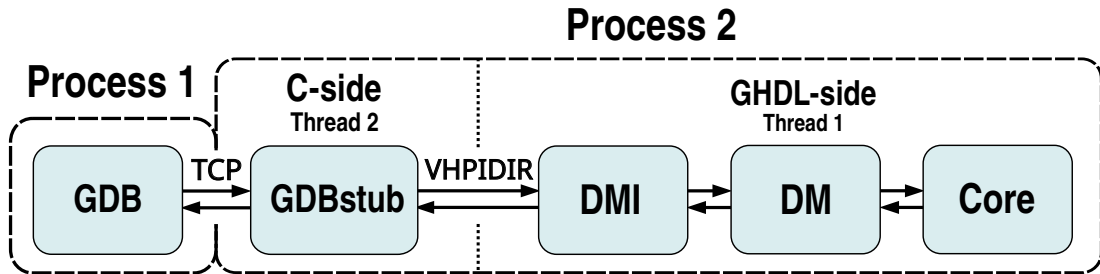
Outline

- 1 RVapo
- 2 Control and Status Registers
- 3 Exceptions and Interrupts
- 4 Debugging Support
- 5 Connecting GDB**
- 6 GDB Showcase

GDBstub and the Debug Module Interface

- GDBstub is a Debug Translator that acts as a bridge between GDB and the DM
- It receives packets from GDB using the Remote Serial Protocol and translates them into sequences of DM register accesses
- The Debug Module Interface (DMI) exposes DM registers to GDBstub
- Both the simulation and GDBstub run in the same process, each in a separate thread
- When writing/reading a register, GDBstub requests a transaction by setting a special variable
- The DMI polls said variable on each clock cycle using VHPIDIRECT

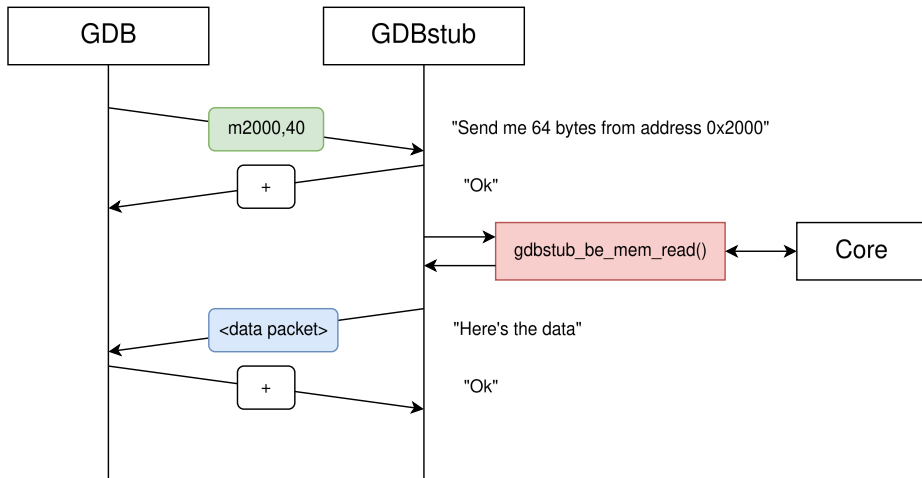
Debug Stack Overview



- In a way, VHPIDIRECT and C features replace the Debug Transport Hardware and Debug Transport Module

Example - Read Memory, Front End

- Suppose we want to execute '`(gdb) x/16xw 0x2000`':



Example - Read Memory, Back End

- `gdbstub_be_mem_read()` performs roughly the following:
 - Save registers `x30` and `x31`
 - Load `progbuf0` with `'lw x30, 0(x31)'`
 - Load `progbuf1` with `'addi x31, x31, 4'`
 - Write start address (`0x2000`) into `data0`
 - Write `command` with `regno=x31`, `transfer`, `write`, and `postexec`
 - Poll `abstractcs` until `busy` is clear
 - Loop until we read the whole range:
 - Write `command` with `regno=x30`, `transfer` and `postexec`
 - Poll `abstractcs` until `busy` is clear
 - Read `data0` containing currently read word, save it
 - Restore registers `x30` and `x31`

Outline

- 1 RVapo
- 2 Control and Status Registers
- 3 Exceptions and Interrupts
- 4 Debugging Support
- 5 Connecting GDB
- 6 GDB Showcase**

Bubblesort Demo

```
1 void bubblesort(int *A, int len);
2
3 int main (void) {
4     int a[] = {7, 15, 5, 0, 1, 19, ...};
5     int n = sizeof(a) / sizeof(a[0]);
6
7     bubblesort(a, n);
8
9     return 0;
10 }
11
12 void bubblesort(int *A, int len) {
13
14     int i, j;
15     int swapped;
16
17     for (i = 0; i < len-1; i++) {
18         swapped = 0;
```

```
19         for (j = 0; j < len-i-1; j++) {
20             if (A[j] > A[j+1]) {
21                 // swap
22                 int temp = A[j];
23                 A[j] = A[j+1];
24                 A[j+1] = temp;
25
26                 swapped = 1;
27             }
28         }
29
30         if (swapped == 0) {
31             break;
32         }
33     }
34
35     return;
36 }
```

Bubblesort Demo

```
(gdb) set arch riscv:rv32
The target architecture is set to "riscv:rv32".
(gdb) file software/c/bubblesort
Reading symbols from software/c/bubblesort...
(gdb) target remote localhost:5000
Remote debugging using localhost:5000
_start () at crt0local.S:20
20          addi    a0, zero, 0
(gdb) load
Loading section .intro, size 0x8 lma 0x0
Loading section .text, size 0x1ec lma 0x200
Loading section .rodata, size 0x28 lma 0x3ec
Start address 0x00000200, load size 540
Transfer rate: 637 bytes/sec, 180 bytes/write.
(gdb) █
```

Bubblesort Demo

```
(gdb) break bubblesort
Breakpoint 1 at 0x2dc: file prog/bubblesort.c, line 17.
(gdb) continue
Continuing.

Breakpoint 1, bubblesort (A=0x13d8, len=10) at prog/bubblesort.c:17
17         for (i = 0; i < len-1; i++) {
(gdb) print *A@len
$1 = {7, 15, 5, 0, 1, 19, 4, 13, 25, 2}
(gdb) █
```

Bubblesort Demo

```
(gdb) break bubblesort.c:26
Breakpoint 2 at 0x380: file prog/bubblesort.c, line 26.
(gdb) display *A@len
1: *A@len = {7, 15, 5, 0, 1, 19, 4, 13, 25, 2}
(gdb) c
Continuing.

Breakpoint 2, bubblesort (A=0x13d8, len=10) at prog/bubblesort.c:26
26         swapped = 1;
1: *A@len = {7, 5, 15, 0, 1, 19, 4, 13, 25, 2}
(gdb) c
Continuing.

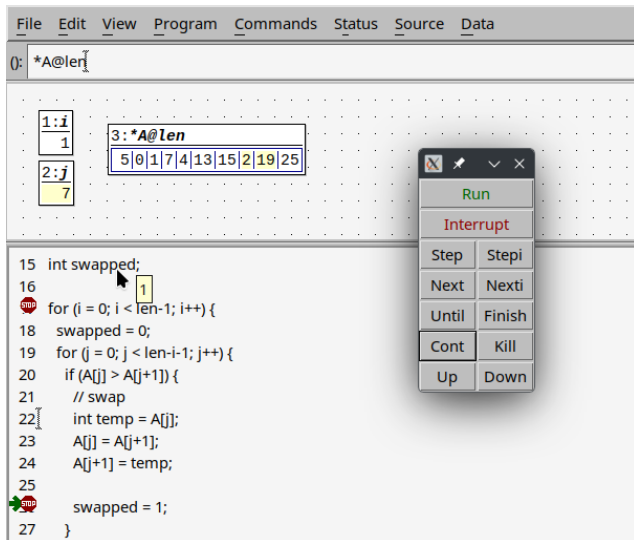
Breakpoint 2, bubblesort (A=0x13d8, len=10) at prog/bubblesort.c:26
26         swapped = 1;
1: *A@len = {7, 5, 0, 15, 1, 19, 4, 13, 25, 2}
(gdb) █
```

Bubblesort Demo

```
(gdb) disable 2
(gdb) break bubblesort.c:31
Breakpoint 3 at 0x3d4: file prog/bubblesort.c, line 31.
(gdb) c
Continuing.

Breakpoint 3, bubblesort (A=0x13d8, len=10) at prog/bubblesort.c:31
31         break;
1: *A@len = {0, 1, 2, 4, 5, 7, 13, 15, 19, 25}
(gdb) █
```

Bubblesort Demo



The screenshot displays the GDB interface during a bubblesort demonstration. The top panel shows the current command '0: *A@len' and a memory dump of the array A. The middle panel shows the source code of the bubblesort function. The bottom panel shows the GDB console output.

Memory Dump:

Address	Value
1:i	1
2:j	7
3:*A@len	5 0 1 7 4 13 15 2 19 25

Source Code:

```
15 int swapped;  
16 for (i = 0; i < len-1; i++) {  
17     swapped = 0;  
18     for (j = 0; j < len-i-1; j++) {  
19         if (A[j] > A[j+1]) {  
20             // swap  
21             int temp = A[j];  
22             A[j] = A[j+1];  
23             A[j+1] = temp;  
24         }  
25     }  
26     swapped = 1;  
27 }
```

GDB Console Output:

```
(gdb) run  
Program started.  
(gdb) step  
16 for (i = 0; i < len-1; i++) {  
(gdb) step  
17     swapped = 0;  
(gdb) step  
18     for (j = 0; j < len-i-1; j++) {  
(gdb) step  
19         if (A[j] > A[j+1]) {  
(gdb) step  
20             // swap  
(gdb) step  
21             int temp = A[j];  
(gdb) step  
22             A[j] = A[j+1];  
(gdb) step  
23             A[j+1] = temp;  
(gdb) step  
24         }  
(gdb) step  
25     }  
(gdb) step  
26     swapped = 1;  
(gdb) step  
27 }
```

Links

- QtRvSim: <https://comparch.edu.cvut.cz/>, <https://github.com/cvut/qtrvsim>
- RVapo: <https://gitlab.fel.cvut.cz/otrees/fpga/rvapo-vhdl>, debugging in branch rvapo-csr-debug
- Debugging and Interrupt Support for VHDL Pipelined Educational RISC-V Core, Bachelor Thesis, 2026: <https://dSPACE.cvut.cz/entities/publication/d0364efd-bea7-4e5b-85f5-0eb80d6c7f2b>
- BI-APS course: <https://comparch.edu.cvut.cz/courses/fit/bi-aps/>, image from lecture 4: Pipelined RISC microarchitecture
- The RISC-V Instruction Set Manual, Volume I: https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf

Links

- The RISC-V Instruction Set Manual, Volume II:
https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf
- The RISC-V Debug Specification:
<https://github.com/riscv/riscv-debug-spec/releases>
- A GDBSTUB for RISC-V by Rishiyur Nikhil:
https://github.com/bluespec/RISCV_gdbstub/tree/master