Realtime debugging of a softcore OpenRISC CPU

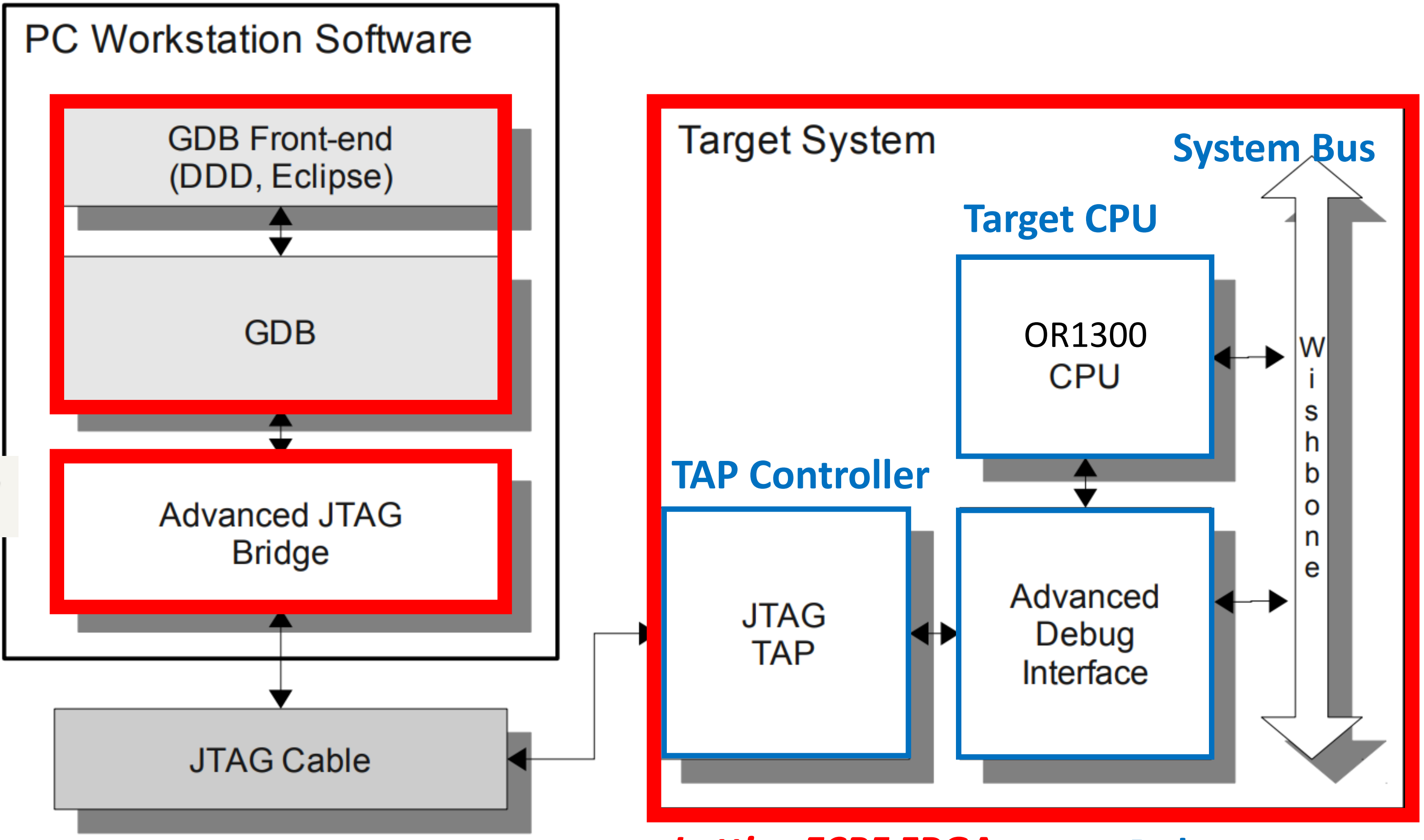
Jules Peyrat

January 5th, 2026

Context

- CS473 Course Material
  - Lattice ECP5 FPGA (« Gecko5Education »)
  - Running a softcore OpenRISC CPU (**OR1300**)
  - Toolchain to compile C programs for the *or1k* CPU
- At the moment, GDB is not compatible with the softcore CPU
- We would like students to be able to debug and upload their programs using GDB
- Step-by-step execution, check RF/memory content

Objective



GDB & User
Front End

Open On-Chip Debugger

SW/HW Bridge

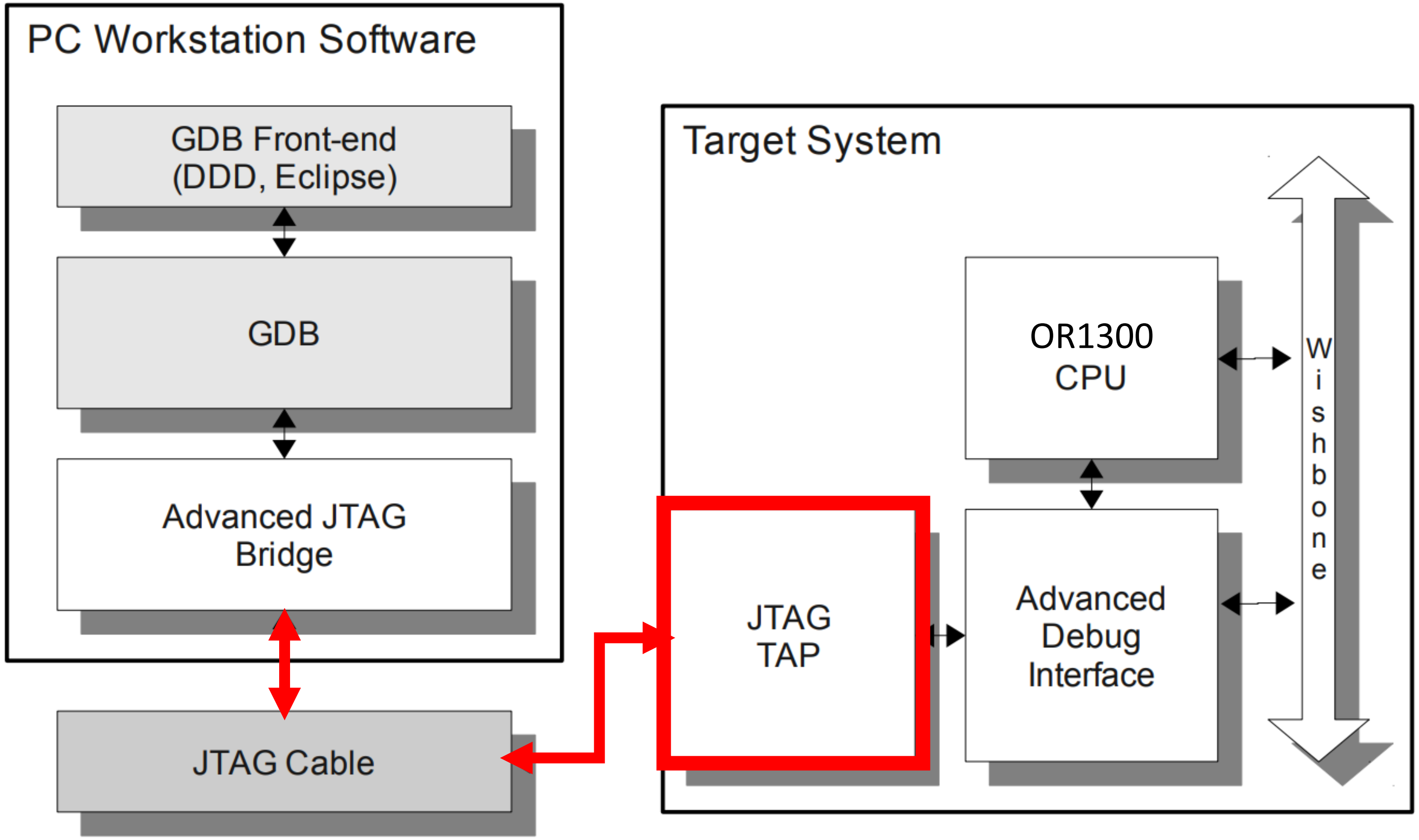
Lattice ECP5 FPGA
« Gecko5Education »

Debug
Controller

Overview

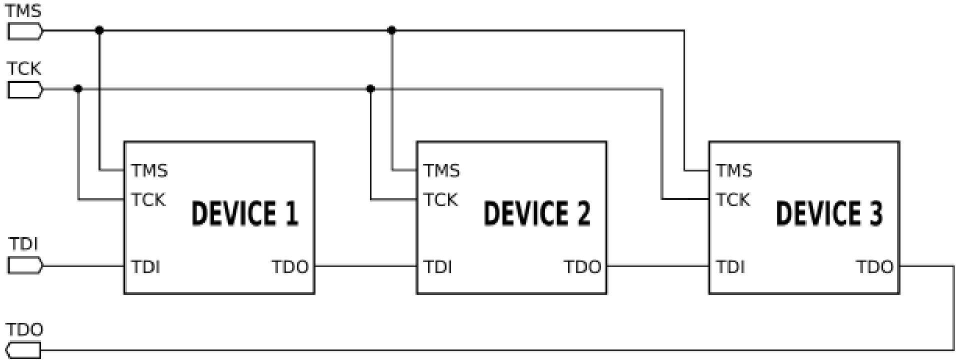
- The JTAG Standard
- OpenOCD Driver
- Top-Level Debug Controller
- Interfacing the system bus
- Interfacing the RF

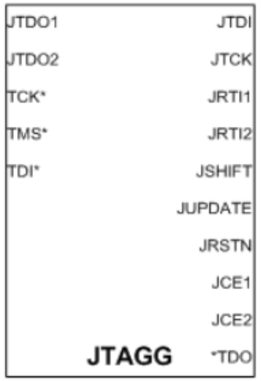
1/ The JTAG Standard



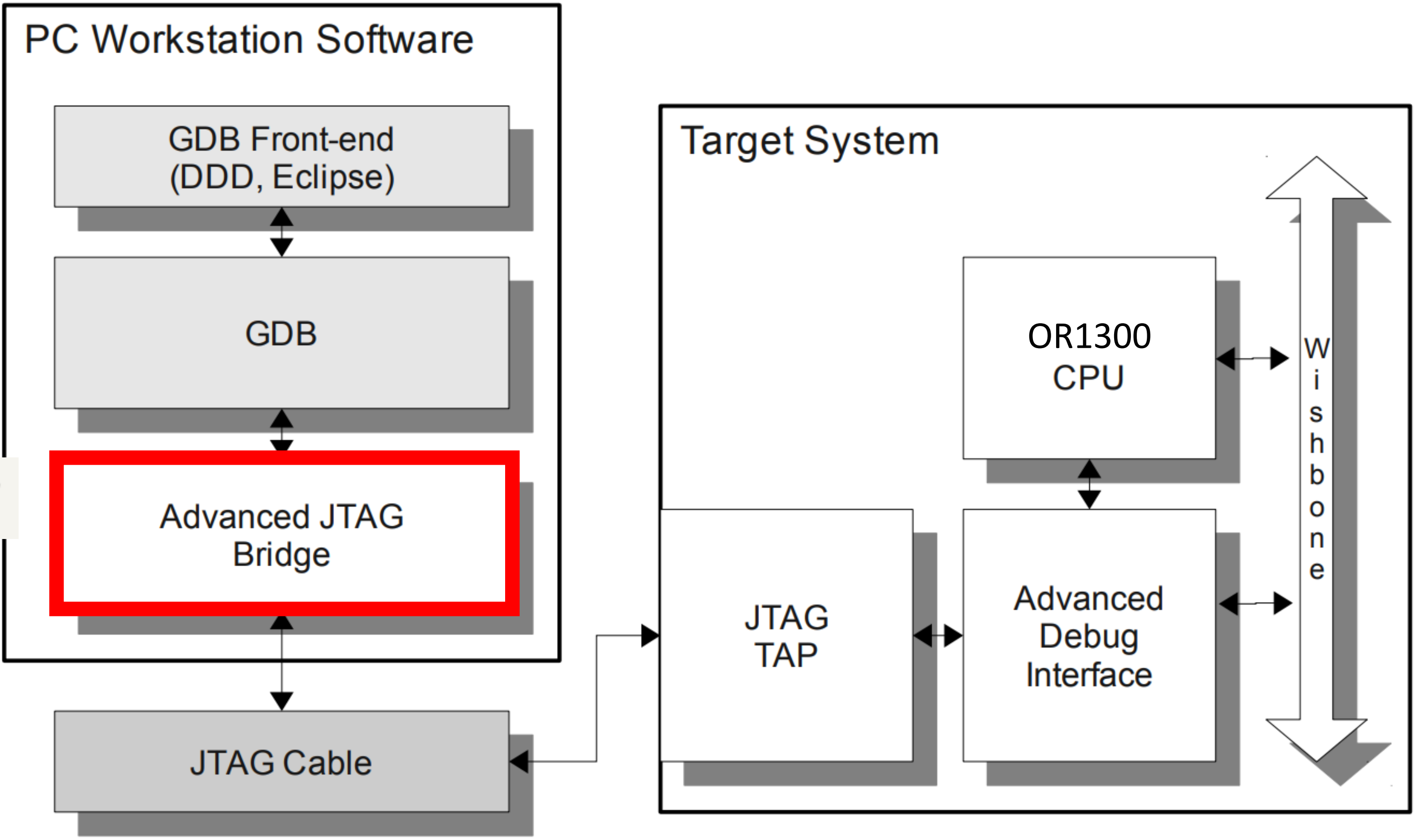
1/ The JTAG Standard

- *Join Action Test Group*
- HW communication protocol for debugging SoC’s
- Five signals
  - TCK (JTAG clock)
  - TDI, TDO (input and output bit lines)
  - TMS (next slide)
  - Async reset
- In this project we use the native JTAGG primitive of our FPGA (also used to reconfigure gates)





2/ OpenOCD



Open On-Chip Debugger

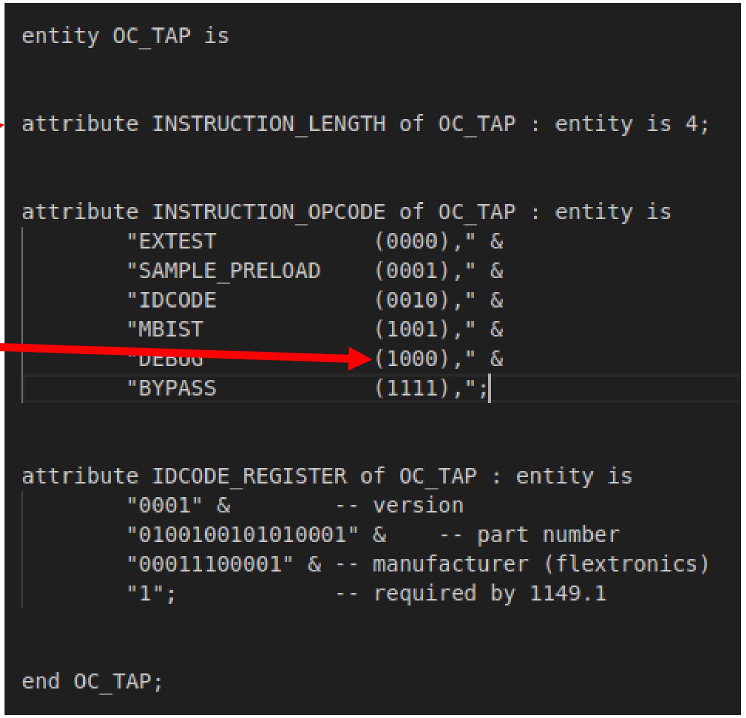
SW/HW Bridge

2/ OpenOCD

- HW/SW bridge through JTAG
- To work, OpenOCD needs
  - IR and DR shift register lengths…
  - IR instruction code…
  - Debug controller command formats (see later)

Problem

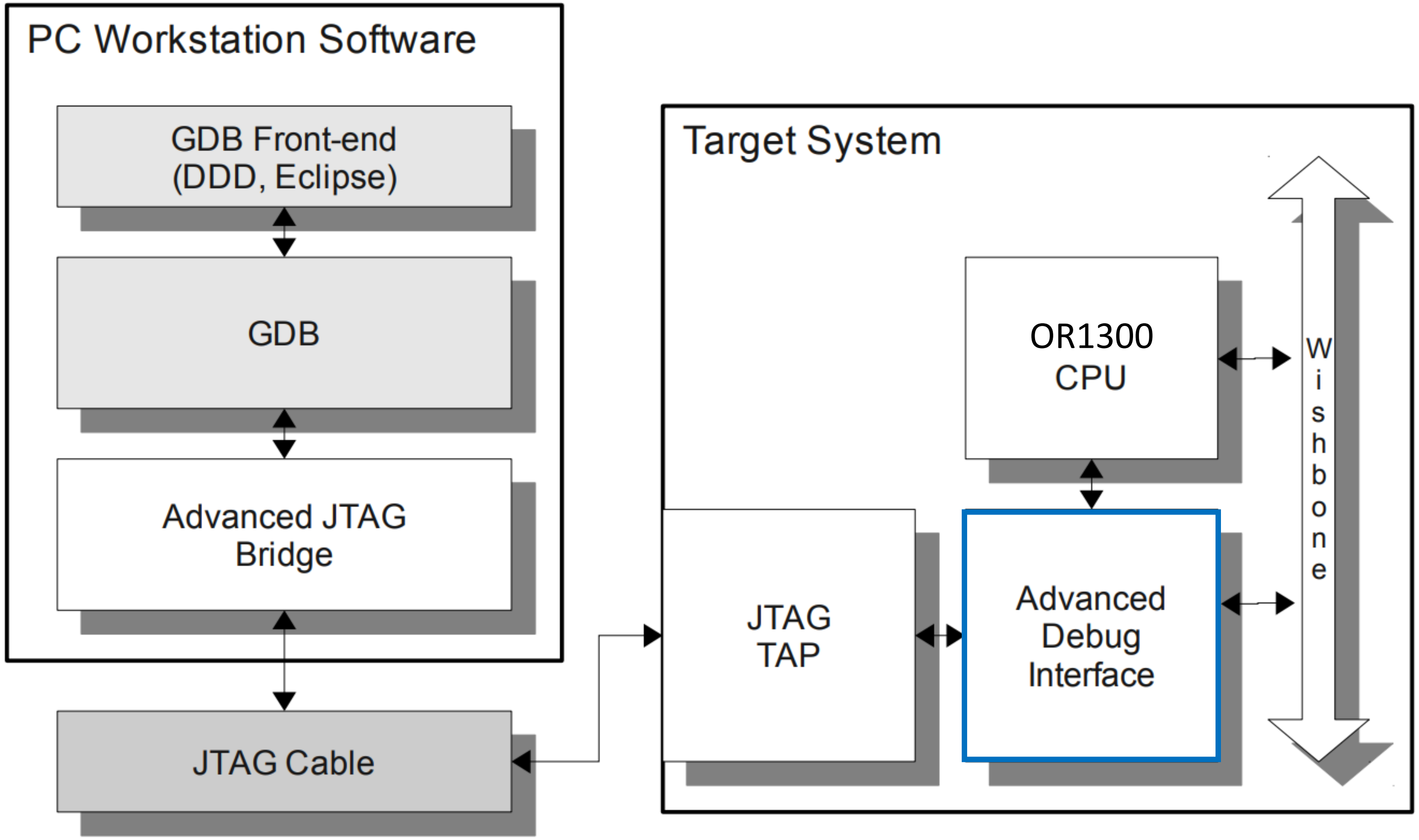
- Using a non-native TAP controller (the one from the FPGA)
- Need for an adapted OpenOCD driver
- Very close to an existing driver! (*mohor*)
- Recompile OpenOCD with that additional driver



```
entity OC_TAP is


attribute INSTRUCTION_LENGTH of OC_TAP : entity is 4;


attribute INSTRUCTION_OPCODE of OC_TAP : entity is
        "EXTEST            (0000)," &
        "SAMPLE_PRELOAD    (0001)," &
        "IDCODE            (0010)," &
        "MBIST             (1001)," &
        "DEBUG             (1000)," &
        "BYPASS            (1111),";


attribute IDCODE_REGISTER of OC_TAP : entity is
        "0001" &          -- version
        "0100100101010001" &    -- part number
        "00011100001" & -- manufacturer (flextronics)
        "1";              -- required by 1149.1


end OC_TAP;
```

BSD file of a supported TAP controller

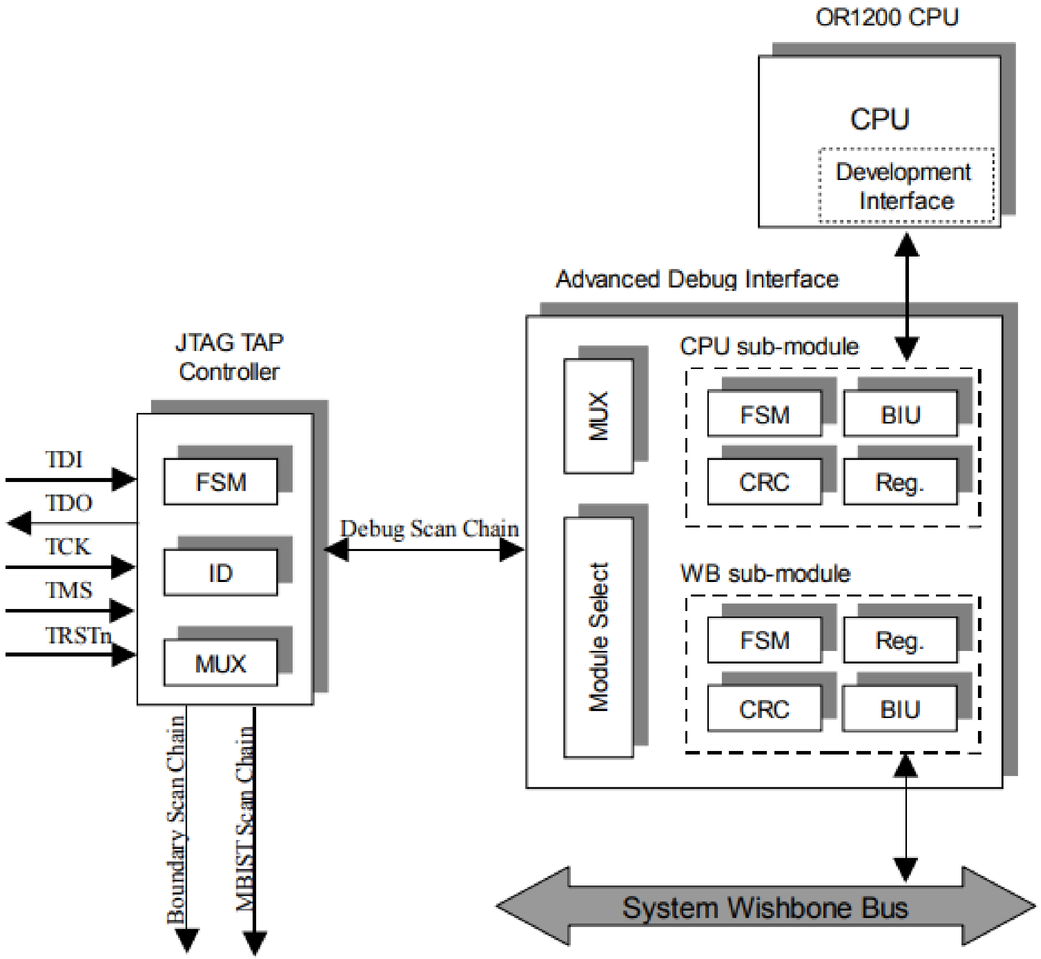
3/ Debug Controller



Debug Controller

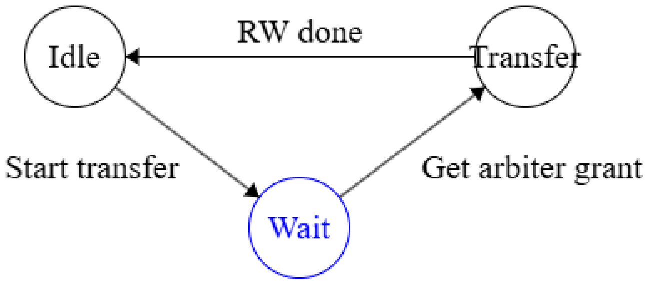
3/ Top-Level Debug Controller

- *Advanced Debug Interface*
- Composed of 4 submodules
- Submodule selection
- Forwards commands to other submodules
- Most recent implementation of a debug interface for *or1k*



4/ Interfacing the system Bus (WB module)

- Our OR1300 CPU uses a different bus architecture (=/= Wishbone)
- Some differences
  - Different buses for address and data (WB), only one for our system bus
  - Native BIU has no support for bus arbiters
- Modify the FSM of the Bus Interface Unit (BIU)
- Three states
  - One to wait grant from bus arbiter
  - One to wait ack from slave (transaction success)
- In case of bus error, goes back to *Idle* and assert signal



4/ BIU Command Formats

- One burst read = 2 distinct commands
- Setup Command → Burst Command
- Status bit used to check if the bus was available
- Up to 3 retries before giving up

Figure 6: Burst Setup command

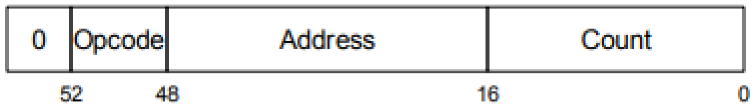


Figure 7: Burst Write format

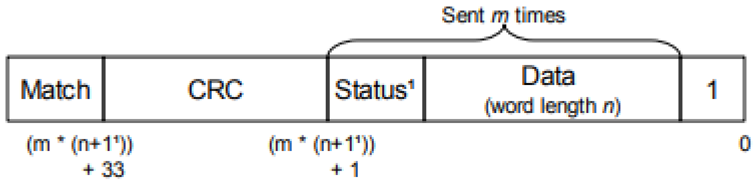
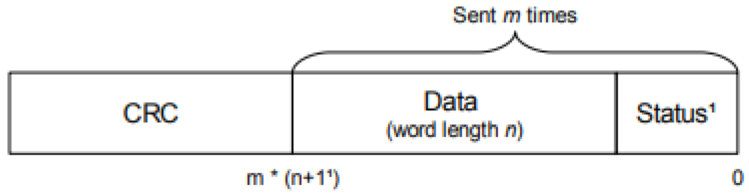


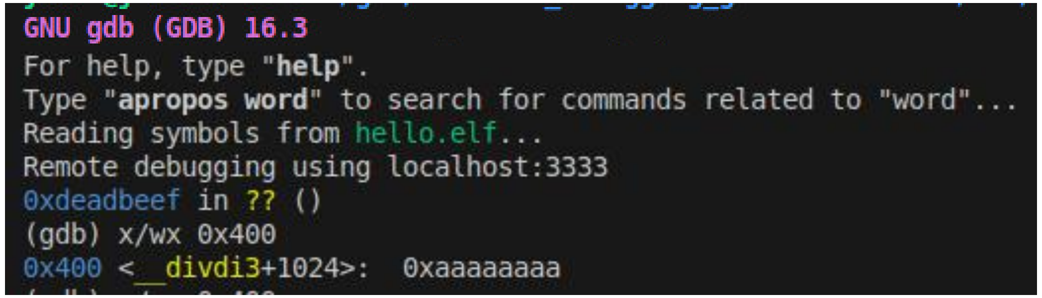
Figure 8: Burst Read format



[1] *Only sent once in hi-speed mode*

4/ Memory Reads

- Open the OpenOCD server and connect to it using GDB
- "x/wx 0x400" (read full 32-bit word in hex)
- → JTAGG, module selection, BIU work
- Memory alignment?



```
GNU gdb (GDB) 16.3
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello.elf...
Remote debugging using localhost:3333
0xdeadbeef in ?? ()
(gdb) x/wx 0x400
0x400 <__divdi3+1024>:  0xaaaaaaaa
```

GDB output

4-bytes aligned 32-bit read (OpenOCD logs)

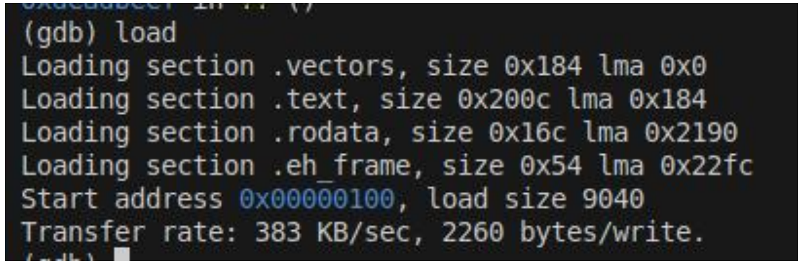
```
406 reading buffer of 4 byte at 0x00000400
407 Read memory at 0x00000400, size: 4, count: 0x1
408 Reading WB32 at 0x00000400
409 Doing burst read, size 4, count 1, start 0x400
410 CRC OK!
```

4-bytes unaligned 32-bit read

```
8118 reading buffer of 4 byte at 0x00000401
8119 Read memory at 0x00000401, size: 1, count: 0x1
[…]
8123 Read memory at 0x00000402, size: 2, count: 0x1
[…]
8127 Read memory at 0x00000404, size: 1, count: 0x1
[…]
8130 CRC OK!
```

4/ Memory Writes

- Writes suffer from CRC errors
- "set {int}0x400 = 0x12345678"
- Likely cause → CRC submodule uses the TDI input line (1 cycle late)
- Unable to upload programs at the moment



```
0xdeadbeef in ?? ()
(gdb) load
Loading section .vectors, size 0x184 lma 0x0
Loading section .text, size 0x200c lma 0x184
Loading section .rodata, size 0x16c lma 0x2190
Loading section .eh_frame, size 0x54 lma 0x22fc
Start address 0x00000100, load size 9040
Transfer rate: 383 KB/sec, 2260 bytes/write.
(gdb)
```

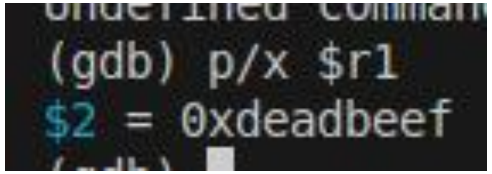
Uploading a program (disabling CRC checks)

```
20576 writing buffer of 4 byte at 0x00000400
20577 Write memory at 0x400, size: 4, count: 0x1
20578 Writing WB32 at 0x400
20579 Doing burst write, word size 4, word count 1,start address 0x00000400
20580 CRC ERROR! match bit after write is 0 (computed CRC 0x00000000)
20581 CRC ERROR! match bit after write is 0 (computed CRC 0x00000000)
20582 CRC ERROR! match bit after write is 0 (computed CRC 0x00000000)
20583 Reporting -4 to GDB as generic error
```

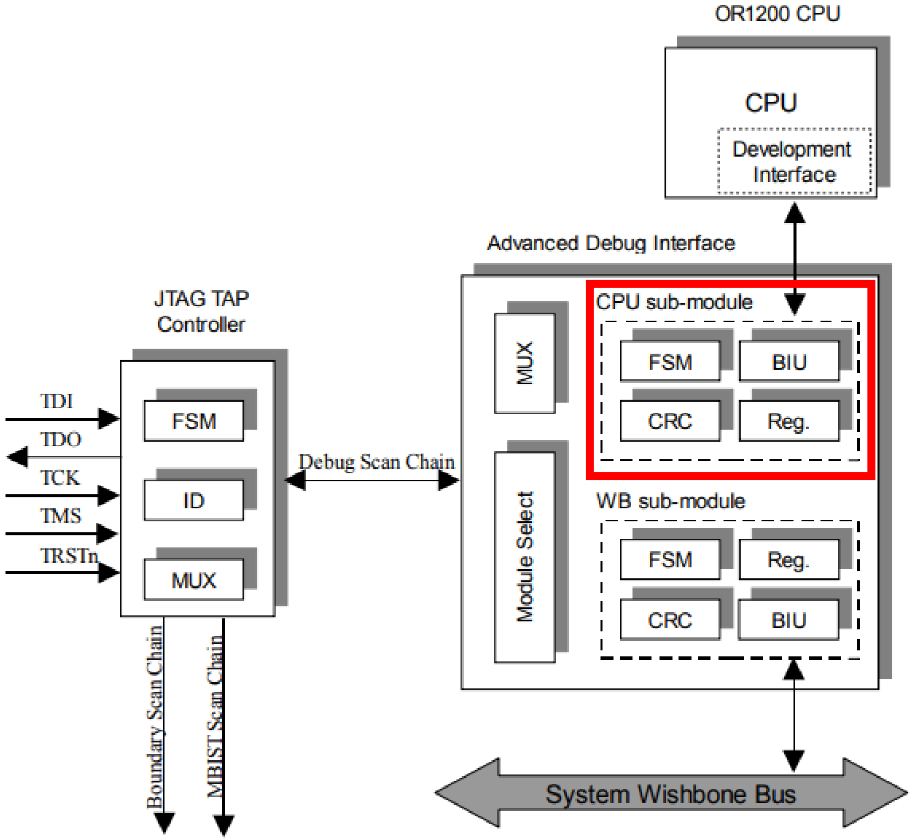
CRC errors on writes

5/ Accessing the register file

- The OR1300 register file is a simple dual port memory. Both ports are already used by the CPU
  - Duplicate the RF to get two additional read ports.
- Structure of CPU submodule very similar to WB submodule
  - Refactor BIU (simple FSM with 2 states)
- Dummy register read
  - Assert one signal to trick BIU into thinking bus is always available



```
undefined comman
(gdb) p/x $r1
$2 = 0xdeadbeef
(gdb)
```



Future work

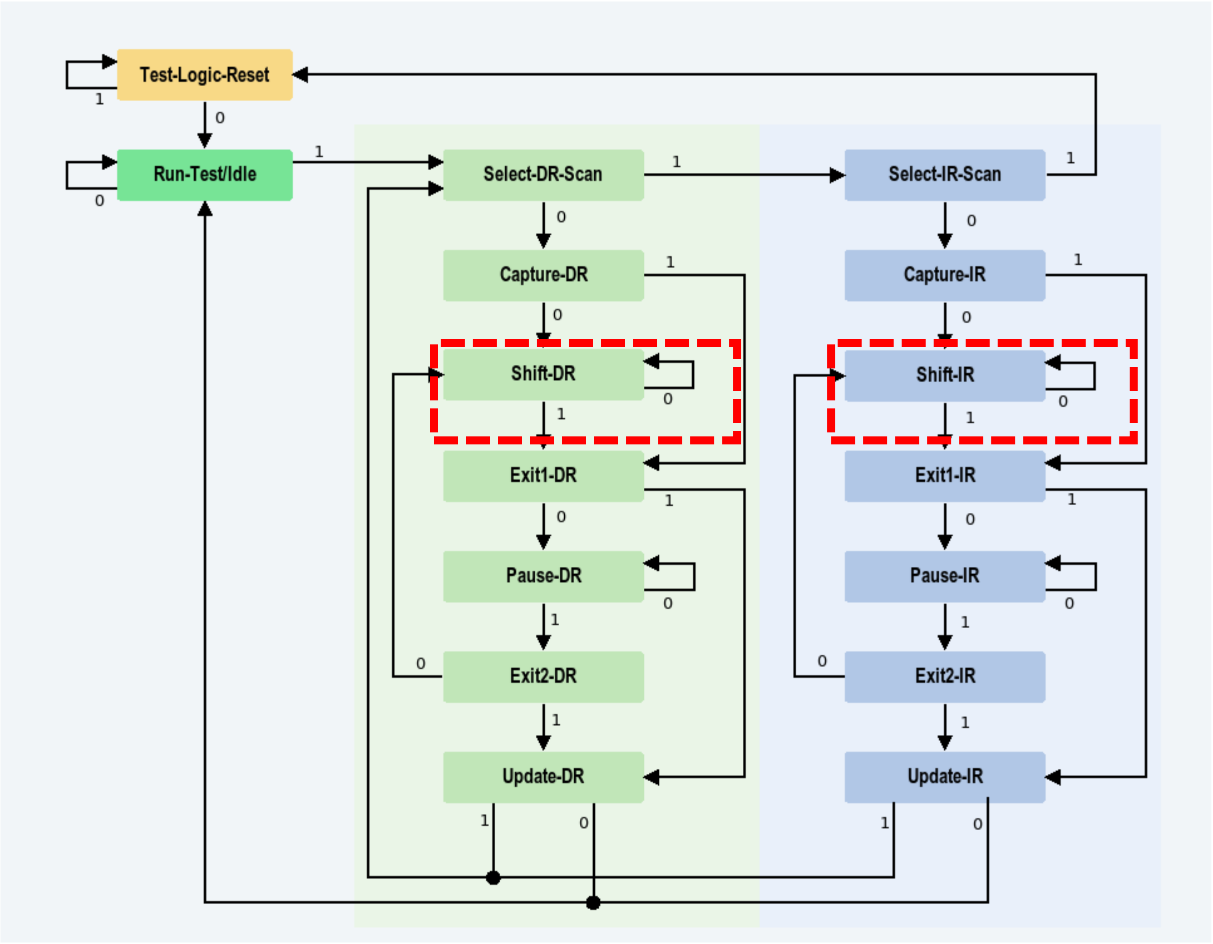
- Fix CRC errors due to TDI
- Implement RF BIU & give access to RF
- Stalling and step-by-step exécution (CPU submodule)
- Experiment with GDB frontends (DDD)

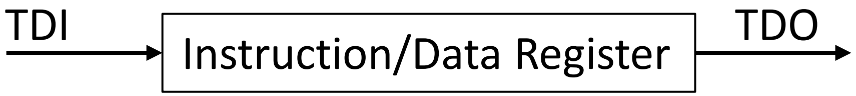
Thank you

Appendix

1/ The JTAG TAP Controller

- TAP (*Test Access Port)* Controller = implements a FSM
- Example: 32-bit stall command
  - Select DEBUG instruction (IR)
    - TMS = 1 → Select-DR-Scan
    - TMS = 1 → Select-IR-Scan
    - TMS = 0 → Capture-IR
    - TMS = → Shift-IR
    - Send data through TDI…. (TMS=0)
    - TMS = 1 → Exit-IR
    - TMS = 1 → Update-IR
    - TMS = 0 → Run-Test/Idle
  - Send data through DR
    - Same but skip cycle 2
- Use of **shift registers**



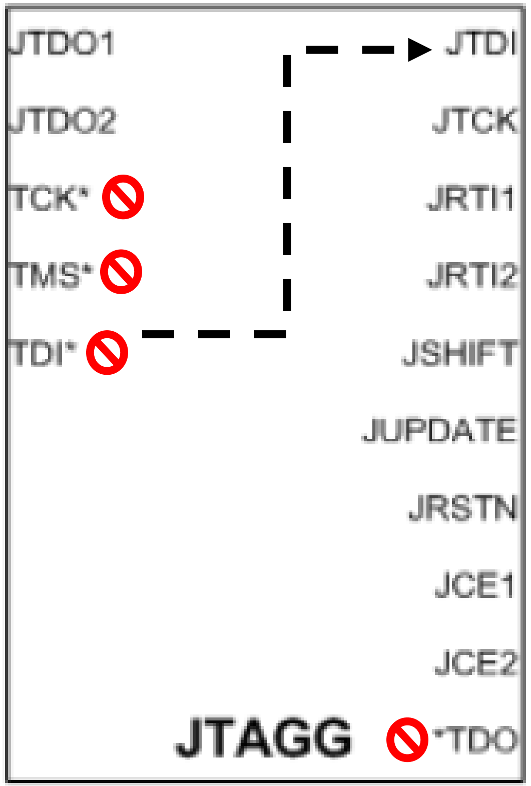

TDI

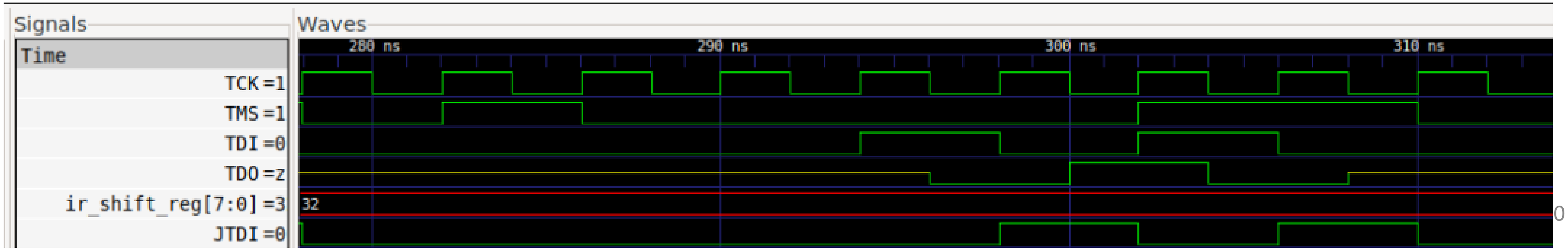
Instruction/Data Register

TDO

1/ The JTAGG Primitive

- What or1k wants us to do
  - Use a provided softcore TAP controller
  - *Artificially create a JTAG interface*
- What we want to do
  - Use the hardware JTAGG primitive of our FPGA
  - Same component used for reconfigurating the FPGA
- Problem
  - No direct access to the TDI signal
  - TDI is 1-cycle late
  - Solution = delay by 1 the Shift-DR state signal


JTDO1
JTDO2
TCK*
TMS*
TDI*
JTAGG
JTDI
JTCK
JRTI1
JRTI2
JSHIFT
JUPDATE
JRSTN
JCE1
JCE2
*TDO


Signals
Waves
Time
280 ns
290 ns
300 ns
310 ns
TCK =1
TMS =1
TDI =0
TDO =z
ir_shift_reg[7:0] =3
32
0
JTDI =0