Ariane Debug Implementation

This document describes Arian's external debug specification according to riscv-debug-spec-0.13.

The current implementation supports:

- `dcsr`, `depc` and one `dscratch0` register
- Triggers are unimplemented (they read zero according to the spec)
- Debug module (DM) is separated from debug transport module (DMI)
- DMI can be used as a raw interface or accessed with a JTAG converter which is compatible to the JTAG DMI specified in 0.13

The whole debug implementation lives within the `src/debug` folder. High level parameters can be customized in the DM's package (`dm_pkg.sv`). Link to the source code.

This document is not a replacement for the debug specification. It just talks about one possible implementation as it is done for Ariane. It is highly recommended to look at the debug spec before trying to understand the actual implementation.

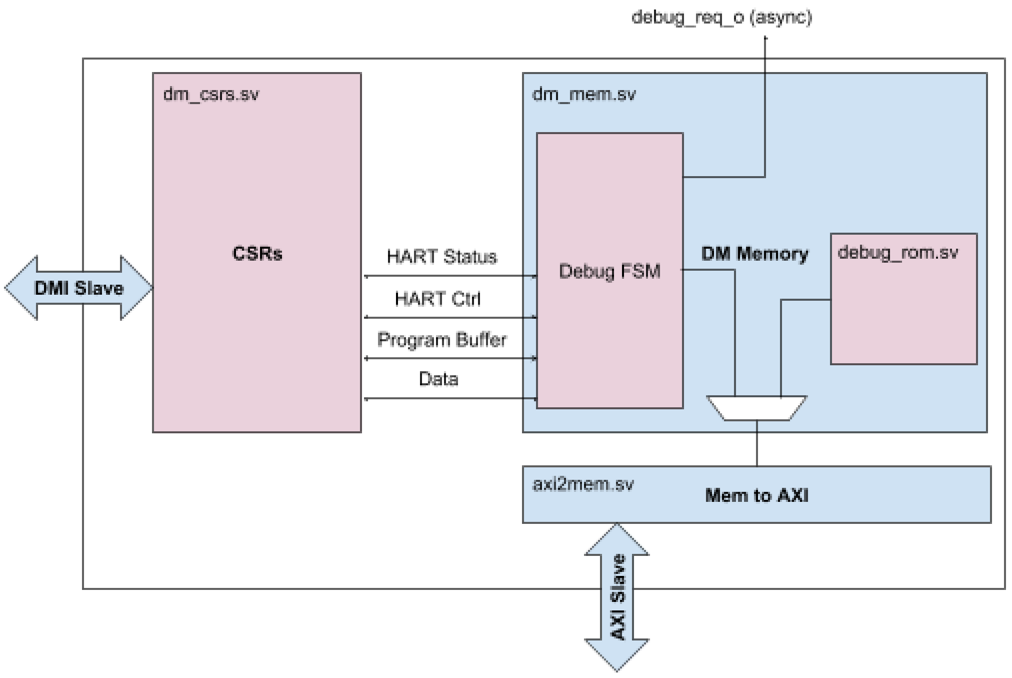

Figure 1: Overview of Debug Module

The DM largely consists of two modules:

1. The debug CSRs as specified in the 0.13 debug specification. Not all operations are supported. Currently missing are:
    a. Hart Array Window Select (making it possible to select more than one hart at a time)
    b. Device Tree Addr 0-3
    c. Next Debug Module: Only one DM per system is currently supported
    d. Authentication Data: No authentication is required.
    e. System Bus Access (sba): Accesses to the system bus must be handled through a running core (TODO).
    f. Hart Summary 1-3 (only required if more than 32 harts are handled by the same DM).
    g. Only the access abstract register command is currently implemented
2. The debug memory which handles communication with an execution based debug protocol. The debug module exposes a very basic SRAM-like interface which is then protocol translated to AXI. The simple interface was chosen in-order to make translation to different buses as easy as possible.

Note: The debug memory (dm_mem) can be swapped with a different module which facilitates communication with targets which do not support the execution based flow.

As a reference we are providing the definition of **execution based** debug vs. **abstract command based** (see implementation section of debug spec 0.13):

Abstract Command Based

Halting happens by stalling the hart execution pipeline.

Muxes on the register file(s) allow for accessing GPRs and CSRs
using the Access Register abstract command.

Memory is accessed using the Abstract Access Memory command or through System
Bus Access.

Execution Based

This implementation only implements the Access Register abstract command for GPRs on a halted hart, and relies on the Program Buffer for all other operations.

This method uses the hart's existing pipeline and ability to execute from arbitrary memory locations to avoid modifications to a hart's datapath. When the halt request bit is set, the Debug Module raises a special interrupt to the selected hart(s). This interrupt causes each hart to enter Debug Mode and jump to a defined memory region that is serviced by the DM. When taking this exception, `pc` is saved to `dpc` and cause is updated in dcsr.

Core Changes

Changes to the core include support for:

- **Debug mode**: A special mode which behaves according to the debug specification.
- **Debug request**: An interrupt like request. The core waits until it retires a valid instruction and associates the debug request to it. As a consequence of a debug request the core enters debug mode and saves the current PC to the `dpc` CSR. Control flow will be redirected to `HaltAddress`.
- **Debug CSRs**: Ariane currently has support for `dcsr`, `dpc` and `dscratch0`.
- **Single Stepping**: If the single-step bit is set in dcsr and the core leaves debug mode it will retire exactly one instruction and will enter debug mode again. Single stepping behaves according to the debug spec.
- **Privilege modes**: In debug mode the global privilege level will change to M. Exceptions as specified in the spec apply.
- `dret`: A special instruction which ends debug mode and resumes execution at the PC stored in `dpc`. This instruction largely uses the same control logic as `[m|s]ret`.

In debug mode the core bypasses the icache. This makes execution slower but alleviates the problem of the core not seeing the latest debug instructions.

Debug Module

Program buffer and data registers provide means of exchanging data between the debugger and the hart. It is r/w from the DMI side as well as from the core side as it is mapped into the global memory space. The fact that the data registers are mapped into global memory space is represented in the `HartInfo` struct which is readable by the DM.

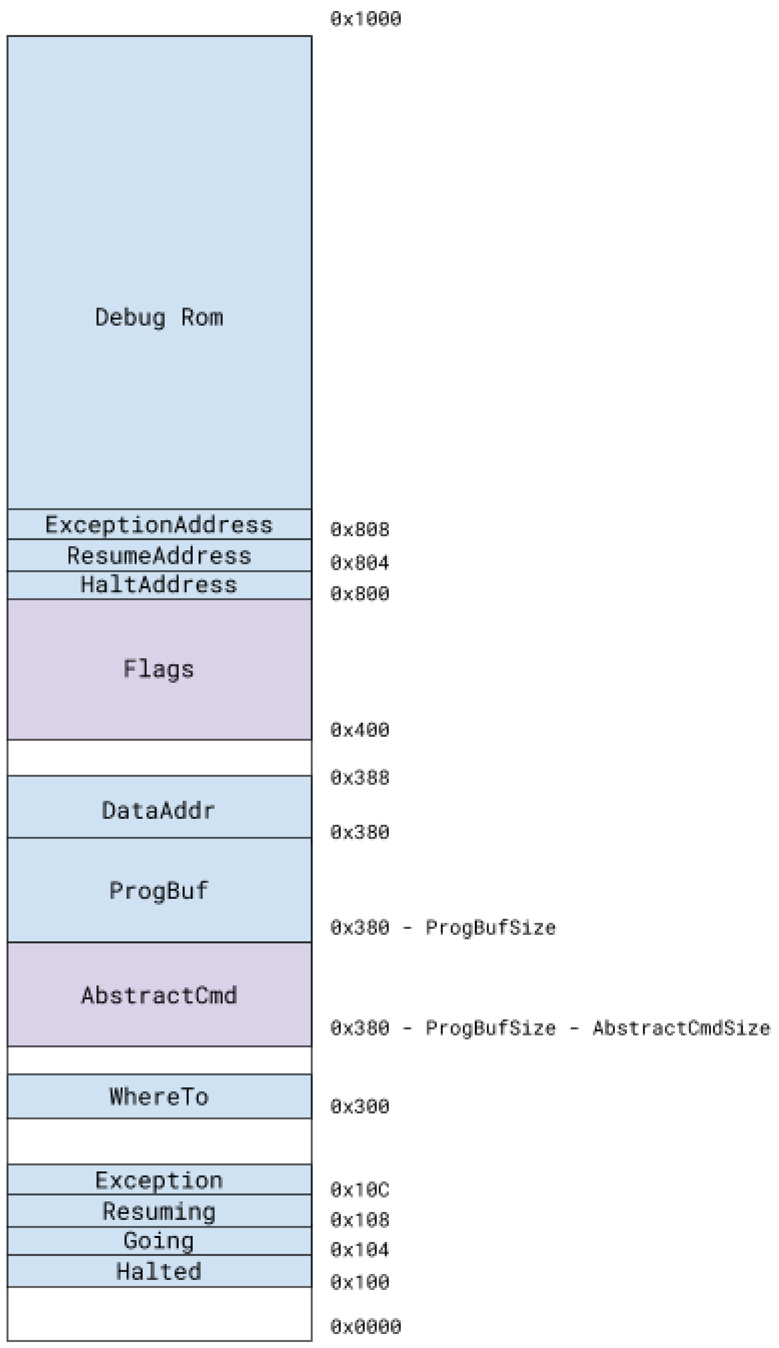

Figure 2: Debug Module (DM) Memory Map

Boot ROM

The boot rom is a first stage bootloader which provides a jump to the base of the DRAM (e.g.: `0x80000000`). Furthermore it also provides an idle loop in case we encountered an unhandled exception. The default exception handler (e.g.: `mtvec` CSR) will point to the base of the ROM +

`0x40`. The bootrom is necessary for the debugger to work because we need a valid instruction which we can associate with the request for entering debug mode.

Debug ROM

The debug ROM provides static hart control infrastructure. In particular it provides a:

1. Entry loop (a.k.a park loop). In this loop the hart informs the debug module that it is halted. Furthermore it continuously monitors a particular memory region controlled by the debug module (the region is called Flags and provides one byte per hart). Those flags indicate whether the hart should:
    a. **Go** and execute the given command
    b. **Resume** normal execution.
2. As soon as the debugger sets the `GO` flag the hart jumps to a debug controlled whereto section which the debug module populates with either:
    a. A jump to `ResumeAddress`
    b. A jump to `AbstractCmd` base
    c. A jump to `ProgBuf` base (depending on whether the transfer bit in the abstract command is set or not)

```
entry:
       jal zero, _entry
resume:
       jal zero, _resume
exception:
       jal zero, _exception

_entry:
        // This fence is required because the execution may have written something
        // into the Abstract Data or Program Buffer registers.
        fence
        csrw CSR_DSCRATCH, s0  // Save s0 to allow signaling MHARTID

        // We continue to let the hart know that we are halted in order that
        // a DM which was reset is still made aware that a hart is halted.
        // We keep checking both whether there is something the debugger wants
        // us to do, or whether we should resume.
entry_loop:
        csrr s0, CSR_MHARTID
        sw   s0, HALTED(zero)
        lbu  s0, FLAGS(s0) // 1 byte flag per hart. Only one hart advances here.
        andi s0, s0, (1 << FLAG_GO)
        bnez s0, going
        csrr s0, CSR_MHARTID
        lbu  s0, FLAGS(s0) // multiple harts can resume  here
        andi s0, s0, (1 << FLAG_RESUME)
        bnez s0, resume
        jal  zero, entry_loop

_exception:
        sw      zero, EXCEPTION(zero) // Let debug module know you got an exception.
        ebreak

going:
        csrr s0, CSR_DSCRATCH          // Restore s0 here
        sw zero, GOING(zero)           // When debug module sees this write, the GO flag is
reset.
        jalr zero, zero, %lo(whereto)
_resume:
        csrr s0, CSR_MHARTID
        sw   s0, RESUMING(zero) // When Debug Module sees this write, the RESUME flag is reset.
        csrr s0, CSR_DSCRATCH   // Restore s0
        dret

        // END OF ACTUAL "ROM" CONTENTS. BELOW IS JUST FOR LINKER SCRIPT.

.section .whereto
whereto:
        nop
        // Variable "ROM" This is : jal x0 abstract, jal x0 program_buffer,
        //                or jal x0 resume, as desired.
        //                Debug Module state machine tracks what is 'desired'.
        //                We don't need/want to use jalr here because all of the
        //                Variable ROM contents are set by
        //                Debug Module before setting the OK_GO byte.
```

Abstract Command

The debug specification mandates that at least the read register abstract command is implemented. The abstract command is translated to a series of RISC-V machine codes. In particular:

Reading GPR (x?)	Writing GPR (x?)
`sd x?, DataAddr(x0) // aarsize = 8`	`ld x?, DataAddr(x0) // aarsize = 8`
`sw x?, DataAddr(x0) // aarsize = 4`	`lw x?, DataAddr(x0) // aarsize = 4`
`sh x?, DataAddr(x0) // aarsize = 2`	`lh x?, DataAddr(x0) // aarsize = 2`
`sb x?, DataAddr(x0) // aarsize = 1`	`lb x?, DataAddr(x0) // aarsize = 1`

Reading FPR (x?)	Writing FPR (x?)
`fsd x?, DataAddr(x0) // aarsize = 8`	`fld x?, DataAddr(x0) // aarsize = 8`
`fsw x?, DataAddr(x0) // aarsize = 4`	`flw x?, DataAddr(x0) // aarsize = 4`

Reading CSR (csr?)	Writing CSR (csr?)

`csrw s0, dscratch0`
`csrr s0, csr?`
`sd s0, DataAddr(x0)`
`csrr s0, dscratch0`
	`csrw s0, dscratch0`
	`ld s0, DataAddr(x0)`
	`csrw s0, csr?`
	`csrr s0, dscratch0`

Optionally the last word of the abstract command buffer is filled with `ebreak` (only if `postexec` is set to false). The program buffer is logically located at the end of the abstract command buffer so if no `ebreak` is executed execution will run into the program buffer.

If a command fails (for example if the necessary CSR isn't implemented) the execution will throw an exception which is captured by the DM. If an exception occurs in debug mode the core will jump to `ExceptionAddress` in the debug ROM. The exception handler will make sure that the core write its core id to `Exception`.

DMI

The DMI is a simple interface which provides handshaking on the request channel as well as on the response channel. DMI data width is 32-bit. Address width is currently 7-bit which is sufficient to address all DM registers. DMI should be easy enough to translate to various other transport methods. For the moment we only provide a JTAG implementation. But it can be also attached to other buses (USB, AXI, APB, etc.).

DMI currently doesn't throw any bus errors.

JTAG DMI

We provide a JTAG communication interface with the debug transport module. It is compatible to the protocol specified in version 0.13 of the external debug spec. In particular it implements the following registers:

Address	Name	Description
`0x00`	BYPASS	JTAG BYPASS, as specified by the IEEE standard
`0x01`	IDCODE	JTAG IDCODE, as specified by the IEEE standard
`0x10`	DTM Control and Status	For Debugging, debug spec 0.13
`0x11`	Debug Module Interface	For Debugging, debug spec 0.13
`0x1f`	BYPASS	JTAG BYPASS, as specified by the IEEE standard

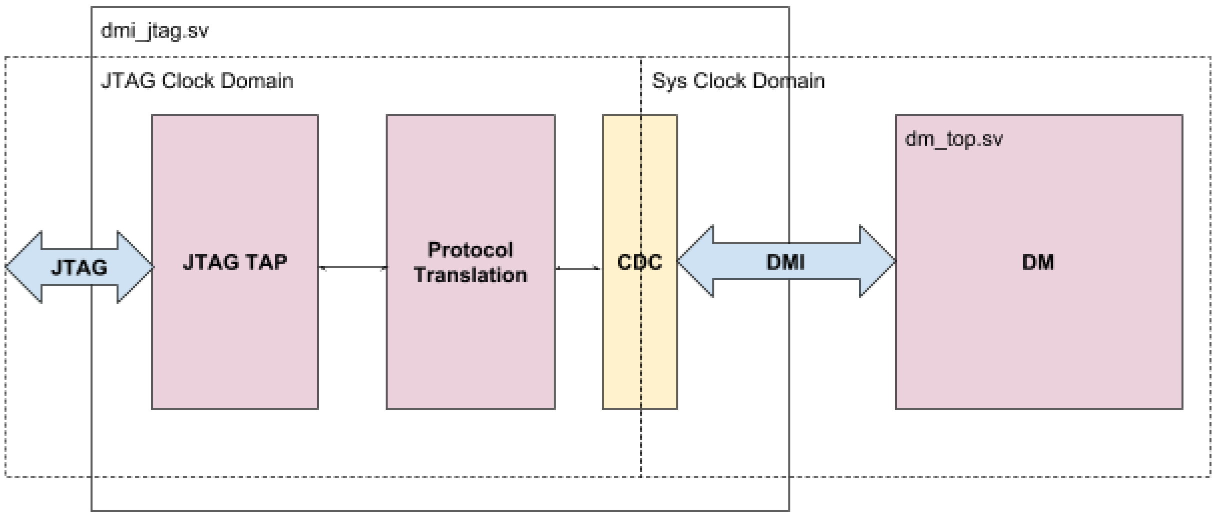

Figure 3: Overview of Debug Transport Module implementing JTAG

IDCODE value is `0x249511C3`. This might need to be changed.