Debug support on the Aa model

Titto Thomas

May 17, 2016

1 Overview

The debug support on the Ajit processor Aa model will be provided through a `debug_daemon` block in the hardware and it's connections with `ccu.debug_daemon` communicates with with other blocks through AHIR pipes. The processor will communicate with the `debug_daemon` only if the processor is running in the `DEBUG_MODE`.

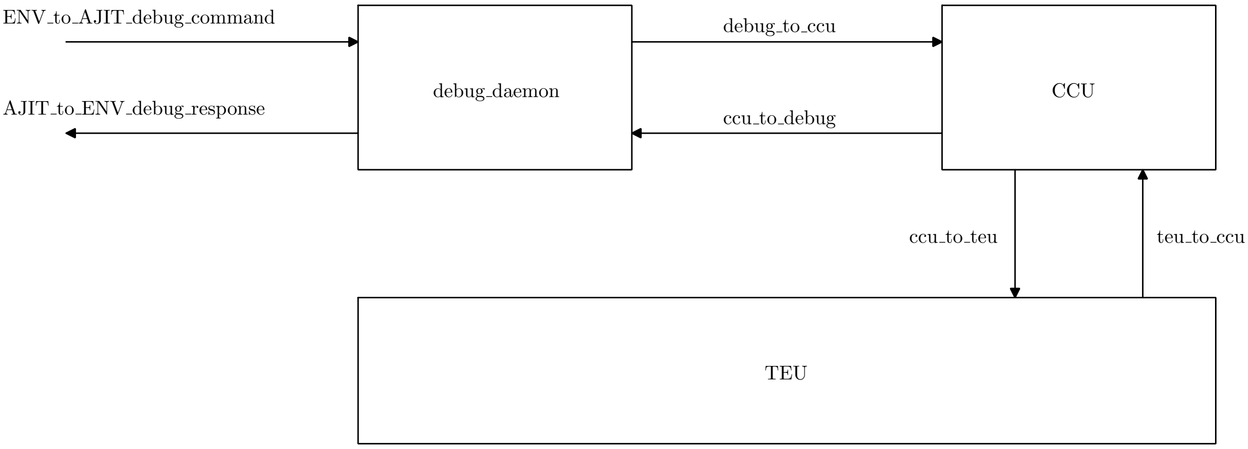


Figure 1: System architecture

The system consists of the two interfaces described below.

1. `debug_daemon` to external environment

The pipes `ENV_to_AJIT_command` and `AJIT_to_ENV_response` are used for communicating with the GDB on host PC. The message format on these pipes are same for both the C and micro-architectural model of the processor. They are described in the section 3 of this document.

2. `debug_daemon` to `ccu`

This interface with `ccu_to_debug` and `debug_to_ccu` pipes will be primarily used for updating the processor state at any instant and to be informed when the thread finishes or stops.

- The `ccu` informs `debug_daemon` if any of the following situations occur and then wait for a response. Along with the code for thread finish, the current value of PC, NPC and PSR will also be passed on.
  - Connection request (initial)
  - Breakpoint hit
  - Watchpoint hit (both read & write)
  - Processor is in error mode

  - Trap occurance
  - Returning from single step
  - Kill request executed

  After subsequent operations, when the `debug_daemon` wants to continue, it will pass the following details to `ccu`.

  - PC, NPC, PSR
  - Next processor mode

  If the host has terminated the connection, then `debug_daemon` will change the processor to normal execution mode.

- `debug_daemon` will send requests to `ccu` for reading/modifying the current processor state. The requests could be

  - Interrupt the processor execution flow when there is a *kill* request from the GDB host.
  - Read/write the iunit/fpunit/control/co-processor registers
  - Read/write the fpunit registers
  - Read/write the memory
  - Read/write the breakpoint/watchpoint registers
  - Detach the debugger and continue exeuction in normal mode

2 `debug_daemon` execution flow

- Wait for a connection request from the `ccu_daemon`. Will not continue forward unless the connection request arrives.
- After receiving the connection request from `ccu_daemon`, wait for a similar request from the GDB host to proceed further.
- Acknowledge the GDB host connect request and execute all the subsequent instructions such as read/write the registers/memory and set/clear the breakpoints/watchpoints.
- Once the GDB host sends a *continue* message, read the current PC, NPC, PSR values. Pass these values to the `ccu_daemon` and start program execution.
- Read the `ENV_to_AJIT_command` and `ccu_to_debug` pipes every cycle and respond if there are any valid requests on them.
  - `ccu_to_debug` can inform regarding breakpoint hit, watchpoint hit and occurance of trap.
    - ∗ Inform the GDB host regarding the breakpoint hit, watchpoint hit and occurance of trap along with the details. The address of breakpoint/watchpoint hit will be obtained from the local copies of their corresponding registers stored in debugger.
    - ∗ Execute all the subsequent instructions such as read/write the registers/memory and set/clear the breakpoints/watchpoints. Update the local copies of PC, NPC, PSR and breakpoints/watchpoint registers with corresponding messages.
    - ∗ Once the GDB host sends a *continue* message, pass it to the `ccu_daemon` along with PC, NPC, PSR and start program execution.
    - ∗ If GDB host sends a *detach* message, pass it to the `ccu_daemon` along with PC, NPC, PSR and let the program execution continue in normal mode.
  - `ENV_to_AJIT_command` inform when the GDB host wants to kill the program execution.
    - ∗ Send an interrupt signal to the `ccu_daemon` and stop the current program execution.

3 Packet format on each of the pipes

The message formats on each of the interface pipes are as shown in the following sections.

1. `ENV_to_AJIT_command` (64 bit, non-blocking)

The messages on this interface will be same for both the micro-architecture and the C model. These 32 bit messages are send through this 64 bit pipe with an additional bit to indicate whether it's valid or not (shown in fig. 2).

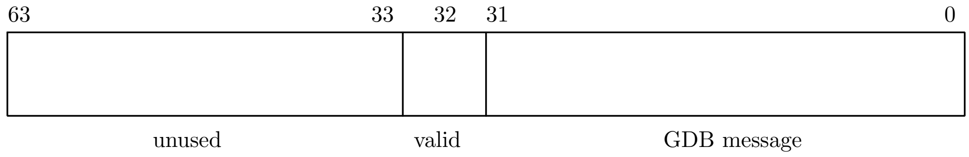


Figure 2: ENV_to_AJIT_command packets

The 32 bit GDB messages could either be commands or data for the debugger. These commands have the format shown in fig. 3.

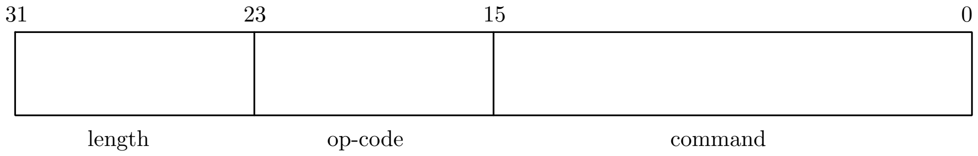
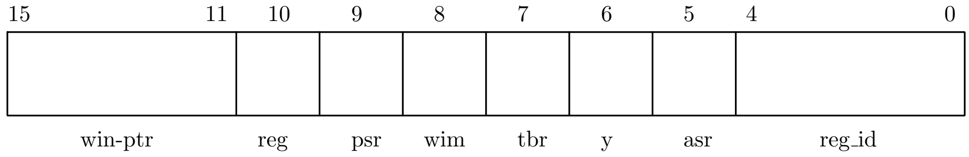


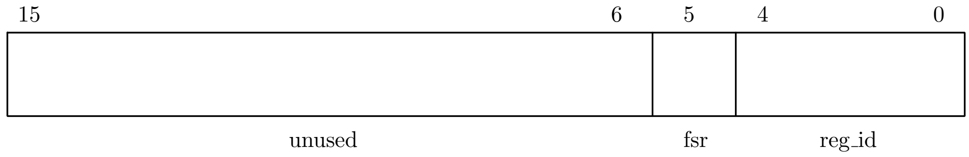
Figure 3: GDB commands for the debugger

The individual messages are as follows.

- Read `iunit` registers : op-code = 1, length = 1, command is interpreted in the following order. If the *reg* or *asr* bit is set, then the *reg_id* indicates the number of corresponding register. The *win-ptr* field corresponds to the window pointer for the register.



- Write `iunit` registers : op-code = 2, length = 2, command is interpreted in the same way as the previous one. This command will be followed by the data to be written.
- Read `fpunit` registers : op-code = 3, length = 1, command is interpreted as following. If the *fsr* bit is not set, then the register with *reg_id* will be read.



- Write `fpunit` registers : op-code = 4, length = 2, command is interpreted in the same way as the read command. This command will be followed by the data to be written.
- Read from the memory : op-code = 6, length = 1, command is interpreted as follows.

15 12	11 8	7 0
unused	number of bytes	asi

- Write to the memory : op-code = 7, length = 2, command is interpreted in the same way as the previous one. This command will be followed by the data to be written.
- Set breakpoint : op-code = 8, length = 2, command is interpreted as follows. The *reg_id* field corresponds to the breakpoint register where the address will be stored. The command is followed by the address to be stored in the register.

15 4	3 0
unused	reg_id

- Remove breakpoint : op-code = 9, length = 1, command is interpreted as the previous one.
- Set watchpoint : op-code = 10, length = 2, command is interpreted as follows. The type indicates the properties of the watchpoint. It could be a write watchpoint (type = 2), read watchpoint (type = 3) or access watchpoint (type = 4). The command is followed by the address to be stored in the register.

15 8	7 4	3 0
unused	type	reg_id

- Remove watchpoint : op-code = 11, length = 1, command is interpreted similarly to the breakpoint removal.
- Read control registers : op-code = 13, length = 1, command is decoded as follows.

15 2	1	0
unused	npc	pc

- Detach : op-code = 15, length = 1, requests to stop debugger and continue normal execution.
- Continue : op-code = 16, length = 1, requests to resume program execution.
- Read co-processor status register : op-code = 17, length = 1, command is unused.
- Write co-processor status register : op-code = 18, length = 1, command is unused.
- Kill the current thread : op-code = 19, length = 1, command is unused.

2. `AJIT_to_ENV_response` (32 bit, blocking)

These pipes will use the same message formats of the C model, described the Debugger design document[1].

3. `debug_to_ccu` (64 bit, non-blocking)

This pipe has to request the read/write to the general registers/memory/breakpoint registers/watchpoint registers and also allow the `ccu` to continue after a halt (with new PC, NPC, PSR, and mode).

The message format is almost same as that of the `ENV_to_AJIT_command` commands. The structure in fig. 2 is retained and the commands are also same except the following two.

- Continue : op-code = 16, length = 4, command will contain the new processor mode (same values used in the `ccu`). Followed by 3 packets with new PC, NPC, PSR values.
- Kill the current thread : op-code = 19, length = 4, command will be same as the previous one. Followed by 3 packets with new PC, NPC, PSR values.

4. `ccu_to_debug` (64 bit, non-blocking))

This pipe has to inform the `debug_daemon` when it stops and send the register/memory contents when required. The MSB indicates if the pipe has a valid message or not and the rest 32 bits have the following format shown in fig. 4.

63 33	32	31 24	23 16	15 0
unused	valid	length	op-code	command

Figure 4: ccu_to_debug interface packet response

- Connect request : op-code = 1, length = 1;
- Breakpoint hit : op-code = 2, length = 4. Packet is followed by PC, NPC, PSR values.
- Watchpoint hit (read) : op-code = 3, length = 4. Packet is followed by PC, NPC, PSR values.
- Watchpoint hit (write) : op-code = 4, length = 4. Packet is followed by PC, NPC, PSR values.
- Thread finished : op-code = 5, length = 4. The reason for the thread being finished is encoded in the command.
  - command [0] : Trap occured
  - command [1] : Returning after single step
  - command [2] : Interrupt
  - command [3] : Returning after a kill request

  Multiple bits on the command could be '1' at the same time.
- Entered error mode : op-code = 6, length = 1.
- Register/memory content : If the pipe contains a response for one of the register or memory reads, then the then the least significant 32 bits will be the content.

References

[1] Titto Thomas, Ashfaque Ahammed, Prof. Madhav Desai, “Design Of Debugger For AJIT Processor,” 2015.