Design Of Debugger For AJIT Processor

Ashfaque Ahammed <ashfaque@iitb.ac.in>

Titto Thomas <tittothomas@iitb.ac.in>

Instructor: Prof. Madhav P. Desai <madhav@ee.iitb.ac.in>

Problem Statement: Implementing GDB compatible remote debugging interface for AJIT processor.

Abstract

The debuggers are a useful tool for the user to emulate and check the working of their program on a remote machine. The user will be able to use the standard GNU Debugger (GDB) interface to communicate with the AJIT processor hardware. On considering that many of the standard commands GDB uses in its communication protocol are not hardware specific, the group proposes to use a virtual server to translate them into simpler commands.

On the platform, it is proposed to have a hardware server communicating with the server bridge through a serial line. The hardware server with it's subunits does the extraction and execution of commands delivered by the user machine. It also has control over the processor core and memory , to check / change its state at any time. This design proposes support some basic debugging commands, and could be easily modified to accommodate a lot more.

1 Introduction

Debuggers are very powerful tool for checking and verifying the working of programs on targeted hardware. They are capable of halting the execution while the program is running, read the machine state , inform the reason for stopping execution, and even upload modified code during the run. This document discuss about implementing debugging provision for AJIT processor C model.

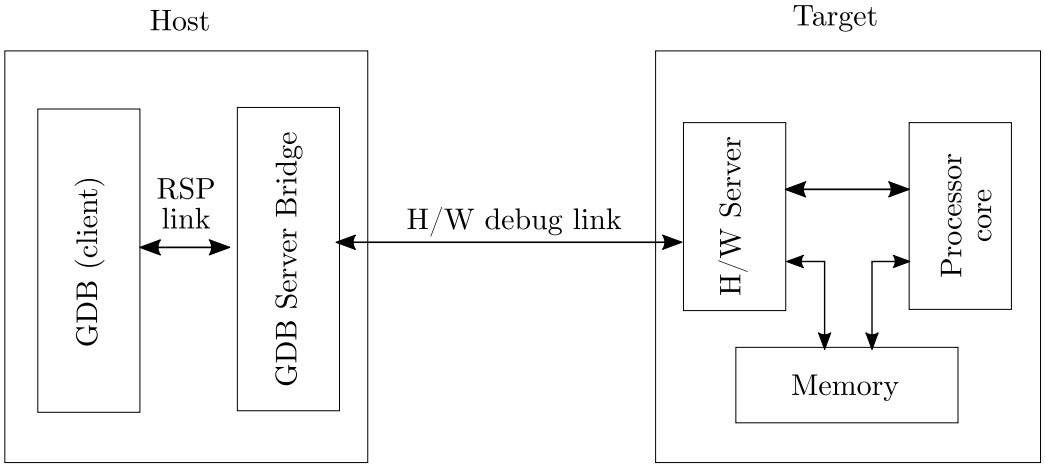

Figure 1: Over view of remote debugging

GNU Debugger (GDB) is the standard and powerful debugger in use. It supports remote debugging of the target, where user can enter a subset of standard GDB commands and get their corresponding response. The commands from GDB will be passed over a virtual port as complex Remote Serial protocol (RSP) packets. A virtual server(GDB server bridge) running on the host machine will listen to the virtual port, decode those packets and respond back accordingly. The same virtual server will communicate with the hardware server on the AJIT processor over a serial port using simple messages.

The hardware server on the target machine will listen to serial port if the target is set in debug mode. The server will receive and decode the commands, and generate the corresponding signals for CPU and MMU. The hardware server will able to halt or run the processor, read / write the registers in the processor, read / write to memory locations and support some other basic debugging options. If the target is in debug mode, the processor will accordingly respond to the hardware server.

Using two additional servers(GDB server bridge and H/W server) between the GDB and AJIT processor, the system provides a perfect debugging environment to debug the code being ran on the target system. The final aim is to support most of the standard debugging commands , and thereby make the troubleshooting much easier.

2 GNU Debugger(GDB)

GNU project Debugger (GDB) is a open source, highly powerful debugger released under GNU General Public License (GPL) which runs on most of the operating systems, and works with many programming languages like C, C++ Java etc.Mainly GDB does four kind of things[4].

- Start program(which is to be debugged), along with any condition if there
- Stop program on specified conditions, (breakpoints and watchpoints)
- Examine register and memory locations etc, once the program is stopped
- change things in the program, so that one can experiment with possible reasons for a bug

GDB can debug programs which runs on the same machine(native), or programs which executes on another machine (remote).In remote debugging, GDB runs on a normal computer which named as **Host** and the machine which runs the program to be debugged is called **Target**. In target stubs used which acts as server, help to perform necessary operations on target and maintain connection with GDB on host which behaves as client. GDB can communicate with the target over a serial line, or over an IP network using TCP or UDP. Remote GDB is mostly used in embedded system debugging, and this method is most suited for AJIT processor.

2.1 Remote Debugging witn GDB

Remote debugging is used to debug programs which runs on machine that cannot run GDB in the usual way or on a small system which does not have a general purpose operating system powerful enough to run a full-featured debugger. On the GDB host machine, it needs to have a copy of the program(which is to be debugged), since GDB needs symbols and debugging information. GDB can communicates with target either by serial or IP network using TCP/UDP. Depending on how target machine is configured to communicate there are two different traditional methods, using `gdbserver` or making use of GDB stubs.

Using `gdbserver`

`gdbserver` is an auxiliary program which helps to connect program on the target to GDB on the host. Unlike gdb stubs, `gdbserver` needs an operating system to run or in fact system that runs gdb server can also run gdb itself, but `gdbserver` is a much smaller

version which make it more suitable for many systems. `gdbserver` runs as a standalone process in target machine, so it does not have privilege to access resources of other process running.This is done through a ptrace call, so that operating system gives the permission. This is why operating system support is necessary for this method. GDB and `gdbserver` communicate serial line or TCP connection using GDB remote serial protocol.

Using stubs

Debugging stub is a file containing special function subroutines that implement GDB remote serial protocol on the target. The stub must be compiled and linked together with the program(which is to be debugged). These stubs are provided with GDB, and are architecture specific, for example `sparc-stub.c` for SPARC target machines. The proposed debugger for AJIT will have to perform all the services that the stub file provides.

Implimenting a remote stub

The debugging stub provides these three subroutines. one subroutine arranges for `handle_exception` to run when the program stops execution. Another subroutine takes control when ever program stops(meeting a breakpoint for example). But it never explicitly called in the program. It begins by sending current status information to host and continue by retrieving and sending any information GDB needs until it gets any GDB command to resume program execution, then it returns back control. A breakpoint function to set breakpoints in the program.

2.2 GDB for SPARC

GDB, the GNU Debugger is used on a large number of different architectures. It can be compiled from the source code with the target architecture given as SPARC by using tool named buildroot. While configuring the GDB for installation, *–target=sparc-buildroot-linux-uclibc* option will install the cross debugger on the host machine.

2.3 Mapping GDB commands to RSP commands

While entering each command in GDB on host machine, it actually mapped to multiple RSP commands to perform corresponding operation. For example let us see how GDB command target remote and break is performed.

`target remote`

target remote command is used to establish connection with the target. the command corresponds to following RSP packet exchanges. First of all client(GDB on host) needs to know what RSP server supports, this is basically the packet size supported. Then client asks why the target halted, again this is to verify that system is halted not because of an terminate command(X) or exited(W). Then it send a qC command, which essentially returns current running thread information.[1].

Next packet (qOffset) will request any offset for loading binary data. Then it will request values of all general register by g command. Then target will provide any symbolic data as per request from host. Flow diagram showing above operations is given below. The RSP packet exchange between host and target is shown in Fig: 2

3 Remote Serial Protocol(RSP)

The GDB Remote Serial Protocol (RSP) provides a high level protocol allowing GDB to connect to any target remotely. It supports different type of connection including direct serial connection or IP network using TCP or UDP. GDB in the host acts as RSP client while target act as RSP server. client sends request to server as packets, where server responds with corresponding value or acknowledgment. The format of RSP packet is shown in Fig.3.

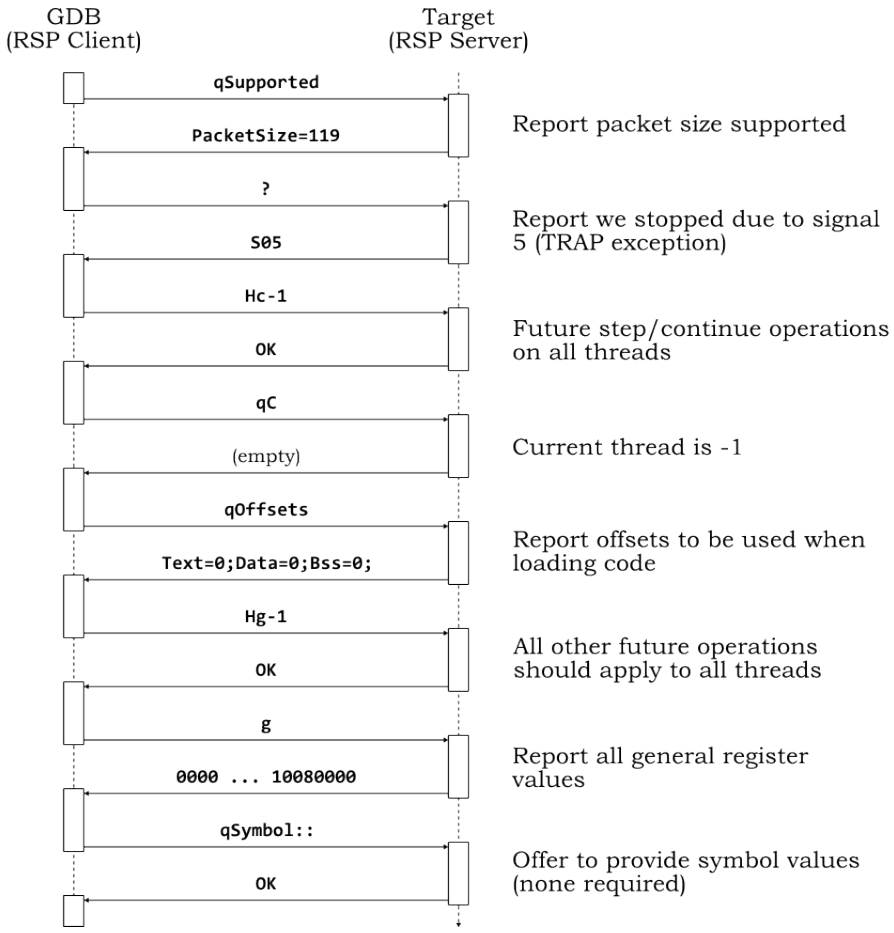

Figure 2: RSP packet exchanges for the GDB `targetremote` command. (This figure is taken from *GDB: Remote serial protocol* by Jeremy Bennett)[1]

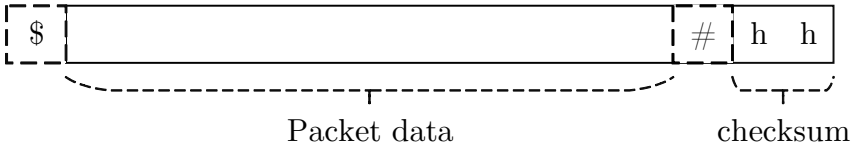

Figure 3: RSP packet format

where $ indicates starting and # indicates end of the packet. Checksum is unsigned sum of all characters in the packet modulo 256, represented by a pair of hexadecimal digits. Each packet is acknowledged by a + if satisfactory or - indicating packet to be re transmitted. Depending on the acknowledgment sent back packets divided to different types(categories).

1. Packets requiring no acknowledgment.
2. Packets requiring simple acknowledgment like OK or Enn("nn" denotes error number)
3. Packets that returns data or an error code.

3.1 Standard RSP commands (packets)

RSP Command [4]	Description [4]	Response	Type
qSupported	Return the feature name followed by '+' if it is supported, '-' if it is not supported and value if necessary. As a minimum, just the packet size can be reported	PacketSize=600;multiprocess-;qRelocInsn-;	2*
?	Report why the target halted. The reply should be the POSIX signal ID of the reason , Eg. 05 for BP exception, 10 for Bus error..	S *signal ID*	3
qC	Current thread being executed	T0	1
Hc*, Hg*	Specifying the threads for which the operations are applicable	OK	2
qOffsets	The offsets for relocating the downloaded code. As we don't want any offset, they should be set to 0.	Text=0;Data=0;Bss=0;	2
g	Read all the internal registers of processor. The order for SPARC is G0-G7, O0-O7, L0-L7, I0-I7, F0-F31, Y, PSR, WIM, TBR, PC, NPC, FPSR, CPSR	register contents in order	3
qSymbol::	Request any symbol table data	OK	1
X*add*, $offset$: *data*	Load binary *data* from *add* to $add + offset$. If fails, GDB will do the same using M (utilized in the proposed model)	Empty	1
M*add*, *count* : *data*	Write *data* from *add* to $add + count$	OK	3
m*add*, *count*	Read data from memory locations *add* to $add + count$	memory contents in order	3
vCont?	Report if vCont actions are supported. Return vCont followed by 'c' for continue and 't' for stop, if they are supported.	vCont;c;t	1
s	Single step execution on the target. The response will be the signal (same as for ?) from the hardware after halting execution when it finishes one instruction.	S *signal ID*	3
c	Continue execution	S *signal ID*	3
Zn*add*	Set breakpoint or Watch-point at a particular *add*. 'n' can be 0 for software breakpoint and 1 for hardware breakpoint, 2 for write watch point, 3 for read watch-point, 4 for access watch-point	OK	3
zn*add*	Clear breakpoint at *add* and 'n' is same as in the previous case	OK	3

D	Detach from the remote server	OK	3
P*number* = *value*	Write *value* to the register *number* in the standard SPARC sequence (given for command 'g')	OK	3
G*values*	Write *values* to the registers in the standard SPARC sequence	OK	3

4 GDB to GDB-server bridge: RSP link

GDB and GDB server bridge communicate through sockets on the host machine. The proposed implementation uses stream sockets (connection-oriented sockets), which use Transmission Control Protocol (TCP). The TCP socket on the machine can be addressed by IPv4 address of 127.0.0.1 and it's port number, by any program running on it. Any allowed, free port can be chosen for communication, and the proposed one is 8888.

Linux offers standard functions for creating and communicating through sockets. These standard functions can be used by including `sys/socket.h` header file in the c source code. These functions are

`socket()`
: The function to create a socket as per given specification. The specifications can include communications domain, type of socket, and protocol to be used with the socket.

`connect()`
: The connect() function shall attempt to make a connection on a socket specified by it's address and port number.

`bind()`
: bind() assigns a local socket address to a given socket that has no local socket address assigned.

`send(), recv()`
: This functions allows to send or receive data through a specified socket.

5 GDB-server bridge

GDB-server bridge is a program that runs on the host machine and acts as a translator between the GDB and the target hardware server. It continuously listens to a virtual port, for the incoming RSP packets sent from the GDB. The program will then process those packets, and retrieve the underlying information. Once the information is obtained, the server will sent the corresponding command to the target over a serial port. Once the target acknowledges the server, it will then respond to GDB in the standard RSP packet format. If the command is a valid GDB command and is unsupported by our target, then an error message will be sent back.

The protocol that GDB use for communication(RSP) is very complex and has a lot of varieties of commands that the receiver will have to understand. It is partly because the GDB is designed to support a lot of varieties of hardwares, and it needs to support a lot of commands on all of them. But, here the target hardware is fixed and a lot of the GDB commands can be filtered / simplified. As the supported commands are only a subset of the normal one, the server does this translation. Since debug hardware have to respond simple commands instead of relatively complex RSP packets. GDB server bridge always listen to the port, and if any RSP packet is received first checks is that request in the packet is supported by this system, if not sends back an empty packet which corresponds to not

supported. If request in the packet was supported, it will generate corresponding commands for the debug hardware. These are simple custom commands which makes design of debug hardware design easier. The debug hardware and software bridge are connected through a custom serial link.

GDB sever bridge is a program written in C which implements a single thread. The program receives RSP commands from the GDB, responds to them and translates them to debug interface commands. The program can be split to several different tasks which could be implemented using functions.

5.1 General functional outline

1. Socket connection
   When the GDB server bridge is initialized the first task is to create a socket with the required specifics. After creating the socket it just waits for some other process to connect to that port. When `target remote ip:port` command is entered in GDB connection get established between these two programs.

2. Packet reception and decoding
   Once the connection is established the second task is to receive and decode the RSP packets sent by GDB. Beginning from the start character($) each successive character is entered in a buffer till receiving the end character(#). If the calculated checksum of the packet matches with the received checksum value, the packet is acknowledged with a "+" character and the command is decoded.

   If the decoded command does not require any information from the hardware server, then the reply packet is sent directly to GDB. If it has any commands for hardware server, a further level of translation is required.

3. Translation to debug link commands
   The task of translating the RSP commands to the debug link commands is a simple one. There is a one to one mapping between the commands, the difference being in the syntaxes. As the debug link command set is limited, some RSP commands needs to be translated into multiple debug link commands. After translation, they are packeted and passed on to the output buffer.

4. Serial port connection
   The final task is to pass the commands over the debug link to the hardware server. It involves establishing a serial port connection to the hardware (with a predefined baud rate), and then passing the date in the output buffer serially. In the 'C' model these unidirectional serial channels will be implemented through pipes.

The GDB server bridge will receive 3 types of commands from the GDB, each of which causes different actions.

1. A lot of GDB commands are not supported by the current implementation, and there could be packets that contain such commands. They will be replied with an empty packet ($#00) to indicate that they are invalid.

2. Second type of commands are simple ones, which could be answered by the GDB server bridge itself. The best example would be the `qSupported` RSP command requesting the supported packet size. The maximum packet size for AJIT processor is fixed as 120 ((G0-G7, O0-O7, L0-L7, I0-I7, F0-F31, Y, PSR, WIM, TBR, PC, NPC, FPSR, CPSR)$\times$ 8 = 576) which the server bridge itself can send.

3. The third type commands are the complex ones, which needs data from the hardware. For such commands, the GDB server bridge translates and send them over to the hardware server and waits for a reply. When the reply comes from the hardware, server decodes it and then resumes communication with the GDB.

5.2 Implementation structure

GDB-Server-bridge will be implemented as multiple C files, later compiled together to form a single library. In the C model this bridge is referred as GPB(GDB to Processor Bridge)

StartGPBThreads.c initializes and starts the main thread. *gpb.c* contains the main thread that listens on the socket and carries out the gdb-server-bridge functionality. *GDBtoAJITbridge.c* contains all the function definitions necessary to communicate over RSP interface(GDB to Server) and Server to processor interface(SPI). Each of these three will have corresponding header files with the function and data structure declarations

5.2.1 Data structures

The proposed implementation will have two data structures, one for representing RSP commands and other for representing the debug link commands.

rsp_cmd
Tis data structure will have four fields, the first field will be an integer indicating the type (same as described in the previous section). The second field will be a character containing the RSP command information. The third and fourth will be optional 32 bit address and data informations, depending on the RSP command on the second field.

dbg_cmd
The *dbg_cmd* will contain 4 fields, the first field will contain debug link command, where second field will hold the register_id. Third and fourth field will contain address and data informations of corresponding debug link command of first field.

5.2.2 Functions

gpb_init(port_no, socket)
This function will try to create a socket on port number *port_no*, returns an error if that port is already occupied by some other processes. If that port is free bind to that socket and initializes communication.

gsb_receive()
gsb_receive function will wait / listen for input packet, will verify the checksum, if found valid then the contents will be decoded(translated) to **gsb_cmd** data structure and is returned from the function.

gsb_compute_response(`gsb_cmd` command_in)
This function will first check the type of the **gsb_cmd** data structure, if it is type-1, it will return an null character (which denotes un supported). If type is 2, will return a predefined response, if type is 3 then they will be translated to single or multiple debug link commands and passed to *send_gpb_spi_cmd()* using **gpb_spi_cmd** data structure. It will acquire responses using *recv_gpb_spi_cmd()* function with **gpb_spi_rsp** data structure, combines them and return back to main function.

send_gpb_spi_cmd(`gpb_spi_cmd` data_in)
send_gpb_spi_cmd() will find the checksum, translates it as a valid debug_link_command, and by checking the available/selected communication medium (set by the main function) sends the packet through communication medium.

recv_gpb_spi_cmd(gpb_spi_rsp data_out)

This function will receive packet from RSP link and verify checksum, if verification fails, sends the command again. If checksum is verified then will give received content will be returned.

gsb_send(string response)

gsb_send() will calculate the checksum, will packetize as RSP packet, and sends through RSP_link

gpb()

gpb() will receive an argument from user which defines the communication medium selected. The *gpb()* will call all other functions. Ajit processor currently supports only serial port communication, so our program will be limited to that as well.

The file *gpb.c* will contains *gpb()* function. All the other functions are defined in *GDBtoAJITbridge.c*

6 GDB Server to Target Hardware : Debug link

The debug link between the GDB server and the target hardware is a standard serial communication link. The final implementation will have a bridge that supports serial and JTAG interface. For the current processor software model two simple pipes will be used. The pipes will be 32 bit wide and are named as `ENV_to_AJIT_serial` and `AJIT_to_ENV_serial`.

A debug command maybe of one or two or three words depending on the type of command. Simple commands like connect, detach will require only a single word, where as memory/register read/write will need 2/3 words.

Debug link : Packet Structure

In each packet, first word will be instruction followed by optional one or two data words. The structure of instruction word is shown on Fig: 4 . The top eight bits specify the total length(1, 2, 3) of the packet. The next eight bits specifies the opcode, and for each opcode the next 16 bits are interpreted differently.

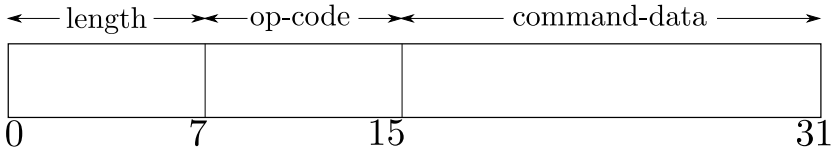

Figure 4: Debug link packet - Instruction word

Opcode label	**Op-code value**	**No: of words**	**Commnad data Interpretation**
DBG_READ_IUNIT_REG	1	1	15:11 10 9 8 7 6 5 4:0 win-ptr reg psr wim tbr y asr reg-id
DBG_WRITE_IUNIT_REG	2	2	
DBG_READ_FPUNIT_REG	3	1	15:6 5 4:0 unused fsr reg-id
DBG_WRITE_FPUNIT_REG	4	2	

DBG_READ_MEM	6	2	15:8 7:0 unused asi
DBG_WRITE_MEM	7	3	
DBG_SET_BREAK_POINT	8	2	15:4 3:0 unused reg-id
DBG_REMOVE_BREAK_POINT	9	2	
DBG_SET_WATCH_POINT	10	2	
DBG_REMOVE_WATCH_POINT	11	2	
DBG_EXECUTE_INSTRUCTION	12	2	
DBG_READ_GENERAL_REG	13	2	15:3 2 1 0 unused csr npc pc
DBG_CONNECT	14	1	
DBG_DETACH	15	1	
DBG_CONTINUE	16	1	

In the integer unit commands (opcode values 1 or 2) if the reg bit is set, then one of the 32 bit general purpose registers (specified by reg-id) is accessed and if any of the other bits are set, then that particular special function register is accessed. In break-point and watch-point commands (opcodes 8,9,10,11) the regid can have only values only 0 to 16, since hardware server supports only up to 16 watch-point/break-point registers.

The second command word will specifies data value for all register writes and break-points/watch-point commands, address for all memory accesses and instructions for instruction execution. The third command specifies data for memory writes.

7 Hardware Modules

The target hardware contains Processor core (containing the debug unit), MMU and memory along with the hardware server. Each of the blocks are explained below in detail.

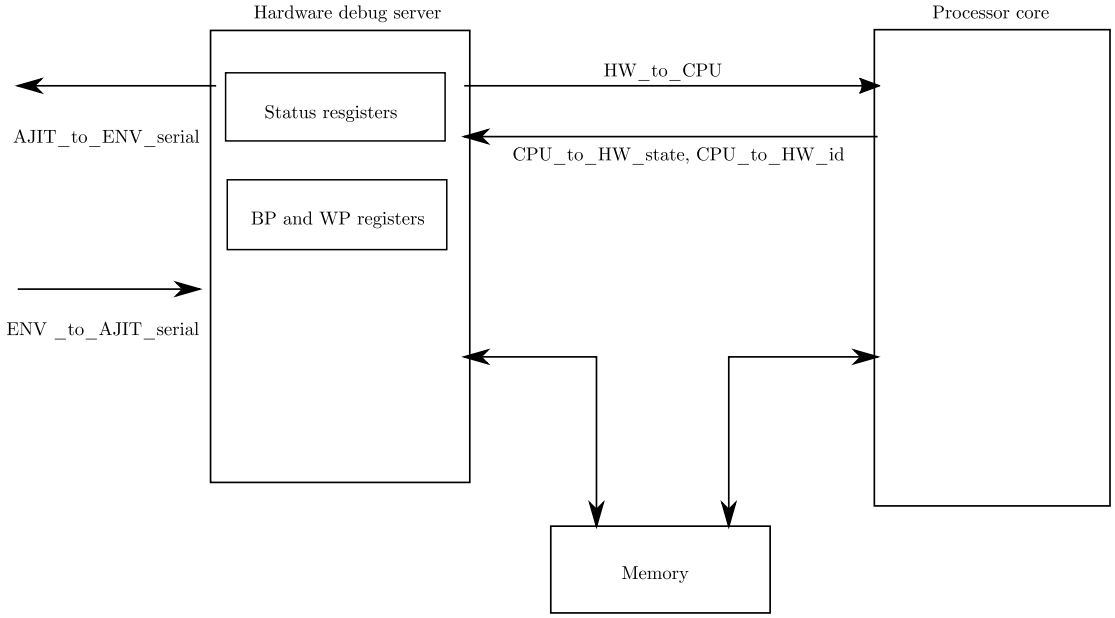

Figure 5: Hardware server connections

7.1 Hardware server

The Hardware server communicates with the GDB server bridge and receives the debug link commands. First it waits for connect request form server bridge, once it establishes connection the hardware server halts the processor before it can execute any instruction and informs that to GDB. When ever processor halts GDB communicates with hardware server to set/clear watch-points or break-points, read/write register/memory contents. When GDB sends the continue message hardware server allows processor to run the current instruction. In every processor execution cycle hardware server checks for watch-point or break-point hits, and halts the processor if there are any.

More details about hardware server are given in the supporting document [5]

Appendices

A Popular Debugging Designs

Some of the debug units from popular processors are studied here to improve the hardware debugger unit for AJIT processor.

A.1 x86 systems

From Intel 80386 CPU, all implementations of the IA-32 instruction set architecture provide debugging support through special debug registers[2]. The modern CPUs have even extended the debugging system to support sophisticated tasks. When the processor is in privileged mode, the debug registers DR0 to DR3 holds the virtual address for 4 breakpoints. If an active breakpoint

is hit on instruction loading, data access, or an I/O operation, the core raises the #DB exception. If a breakpoint should be active only for certain access-types it can be set up in the register DR7, which also allows to enable or disable the complete debugging system. In addition to the dedicated breakpoint registers the debugging system can also be configured for single-stepping, which then raises the #DB exception after every instruction, or it can be configured to trap on context switches to a certain task. Register DR6 holds information about the cause of a triggered exception.

Further breakpoint support is provided through the #BP exception, which can be triggered everywhere in the code by executing a special INIT 3 instruction. This allows the definition of breakpoints directly in the program text and allows instruction breakpoints to be set for more than four addresses.

A.2 OpenRISC

The OpenRISC 1000 architecture defines eight sets of debug registers. Additional debug register sets can be defined by the implementation itself. The debug unit is optional and the presence of an implementation is indicated by a single bit.

In the supervisor-level accessible registers Debug Value Registers (DVR0 - DVR7) the watch / breakpoints can be loaded. Debug Control Registers ...., DCR0 - DCR7) contains informations like whether the values are address or data or something else, whether comparison should be equal , less than greater than etc.. [3].

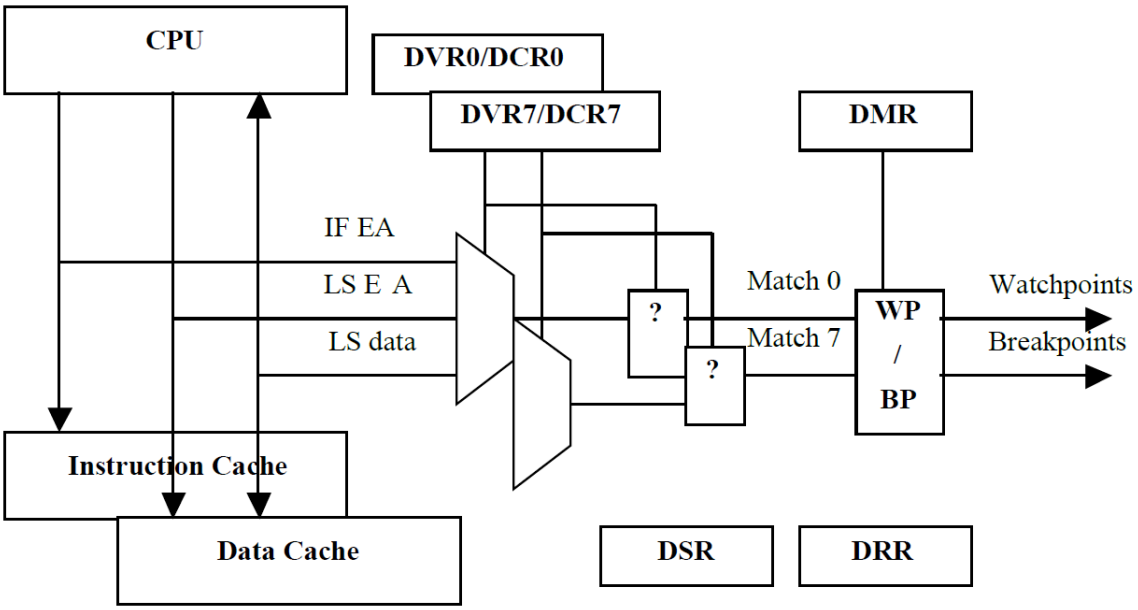

Figure 6: Block Diagram of debug support on OpenRISC 1000(This figure is taken form OpenRISC manual,[3].)

There are two Debug Mode registers to control the behavior of the the debug unit (DMR1 and DMR2). DMR1 provides a pair of bits for each of the 10 matchpoints. These specify whether the watchpoint is triggered by the associated watchpoint, by the watchpoint AND-ed with the previous watchpoint or by the matchpoint OR-ed with the previous watchpoint. Two further bits in DMR1 enable single step behavior and branch step behavior. DMR2 contains an enable bit for each counter, 10 bits indicating which watchpoints are assigned to which counter and 10 bits indicating which watchpoints generate a trap exception. It also contains 10 bits of output, indicating which watchpoints have generated a trap exception.

The Debug Stop register (DSR) is used to assign particular exceptions Current Status. Where an exception has been diverted to the development interface, the Debug Reason Register (DRR) indicates which exception caused the diversion.

References

[1] Jeremy Bennett. Howto: Gdb remote serial protocol. 2008.

[2] Intel. Volume 3b:system programming guide, part 2. *Intel 64 and IA-32 Architectures Software Developers Manual*, November 2007.

[3] Damjan Lampret, Chen-Min Chen, Marko Mlinar, Johan Rydberg, Matan Ziv-Av, Chris Ziomkowski, Greg McGary, Bob Gardner, Rohit Mathur, and Maria Bolado. Openrisc 1000 architecture manual. *Description of assembler mnemonics and other for OR1200*, 2003.

[4] Richard Stallman and Roland H Pesch. *The GDB Manual: The GNU Source-level Debugger*. Free Software Foundation, 1992.

[5] Prof. Madhav Desai Titto Thomas, Ashfaque Ahammed. *Adding Debugger to AJIT Processor C model*. 2015.