In-System debugger for AJIT Microprocessor

Submitted in partial fulfillment of the requirements

of the degree of

Master of Technology

by

Titto Thomas
(Roll No. 133079015)

Under the guidance of

Prof. Madhav P. Desai

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2016

Dissertation Approval

The dissertation entitled **In-System debugger for AJIT Microprocessor** by Titto Thomas (Roll No. 133079015) is approved for the degree of **Master of Technology** in **Microelectronics**.

Examiners

____

____

____

Supervisor

____

Chairperson

____

Date : ________________

Place : ________________

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources.

I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified and idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Titto Thomas
(Roll No. 133079015)

Date : 28th June 2016
Place : IIT Bombay

Acknowledgement

I would like to express my sincere gratitude toward my thesis advisor, Prof. Madhav P Desai for his guidance and encouragement throughout this work. His advice and suggestions were critical for the progress of the work and gave me an excellent opportunity to explore and push the boundaries. The methods and philosophies I learned from him will remain close to me throughout my professional life.

I would like to thank my colleagues Mr. Ashfaque Ahammed, Mr. Piyush Soni and Ms. Neha Karanjkar with whom I have collaborated during this project. The discussions we had was a great learning experience and provided me valuable inputs regarding the project.

I would also like to thank Ms. Krupa Bathia and all the students working with Prof. Madhav P Desai for their help during different stages of the project.

I would like thank the Electrical Engineering Department, IIT Bombay for providing me all the facilities and resources for research.

I would like to thank my friends from Wadhwani Electronics Laboratory for the support and all the good times we shared. My life at IIT Bombay would have been much less exciting if not for my friends here, Nithin Sankar, Wilson A N, Anakha V B, Renju Boben, Abhijith V S, Reshma Krishnan, Vineesh V S, Sreejith K P, Ramzan Basheer, and Achuth P V. I would like to thank them for the support and all the memories.

To my family, all of this would have been impossible without your love, support and faith in me.

Titto Thomas

Abstract

As the complexity of embedded systems increase, providing debugging support becomes a crucial requisite. AJIT processor project is an attempt to deliver a microprocessor platform that can cater to different embedded applications. The design and implementation of a complete debug infrastructure for this processor is the principal goal of this project.

One of the most powerful and commonly used debuggers, Gnu Debugger (GDB) is chosen as the front end of the system. Provision of a non-intrusive view and the modifiability of processor states are the requisites of this debug system. Software server running on the host computer will work as translator between the GDB and hardware. The processor hardware description is modified to include additional modules and support debug operations. The processor has been implemented as two separate models with in-built debug support. Addition of a hardware server and the modified processor core constitute the debug functionality in the software model. Two debug units are added to the micro-architecture for the same purpose.

The implementation is verified across all the models and the final FPGA prototype is verified with several test programs. This system is mainly being used for the development of the Linux kernel for the AJIT processor. The scope of the design includes supporting a JTAG interface between the computer and processor hardware.

Contents

List of Figures

Chapter 1

Introduction

The AJIT processor project aims to develop a 32-bit processor platform compliant with the IEEE standard 1754-1994[1] and SPARC V8 Architecture Manual[2] from SPARC International Inc. This work attempts creation of an efficient debug infrastructure and provision of real time debug capability on the processor, even when it is working as part of a larger system. Along with this in-system debugging, it can also be used for verification during the different stages of processor development. The proposed system with integrated hardware-software architecture provides a non-intrusive view of the processor and the ability to remotely modify it at any instant.

1.1 Motivation

As the embedded microprocessors are getting more powerful and are being used to build complex systems, their debug support needs improvement. An efficient debugger requires low level control on the program execution and complete visibility in hardware. Such debug capabilities are necessary to deliver quality systems and meet production goals.

According to 2014 results of UBM Tech's annual comprehensive survey [3] on embedded systems industry, debugging tools and support were the most lacking aspect in embedded design activities. The conventional hardware-independent debugging techniques are very restricted and do not provide any information about the system internals and peripherals (how they behave in a system). This work is motivated towards creating an efficient in-system debugger for the in-house developed AJIT processor and across its different implementation models. Such a system would enable the users to evaluate both the on-chip and in-system effects on the working hardware.

Even though there are proprietary commercial tools from different companies, very few of such debuggers exist in the open source domain. Gnu debugger (GDB) is the main one among them that supports many different architectures. The aim is to create an efficient system by synchronizing one of these standard software debuggers with this specific hardware. The system needs to provide real time debug capability on the actual processor hardware and support application development.

This in-system debugger will be an essential utility throughout the AJIT processor hardware and software development to remain as the only way of external control in the final hardware.

1.2 Organization of the thesis

Chapter 1 is the introduction. In Chapter 2, the existing solutions for embedded system debugging are briefly discussed. Chapter 3 describes the system architecture and design of in-system debugger across different AJIT processor models. Chapter 4 deals with the implementation and validation details of the proposed system. Chapter 5 discusses the results and future work.

Chapter 2

Debug solutions for microprocessors

Debugging programs on embedded systems is really different from how its done during traditional software development. The limited resources, memory and processing power on board restrict the presence of a full featured debugger on the target hardware. Another challenge in developing debug solutions for such systems is that they may not even be running on completely verified hardware.

Because of these challenges and restrictions, most of the embedded system debugging is done remotely. It is the process of debugging programs running on a smaller target hardware using a powerful host computer and they communicate through any of the standard interfaces.

2.1 Standard debugging techniques

2.1.1 Logic Analyzers and Trace based debugging

These methods are useful for older, small hardware systems and does not require any external computer for debugging. The main idea is to add tightly coupled hardware in the processor to provide a trace of what is happening inside, for the outside world. After the processor completes executing the program, the trace information can be compared against expected values to find out the errors.

2.1.2 In-circuit emulators

In-circuit emulators are available for a few microprocessor architectures, providing a software based debugging solution. Most such emulators are proprietary and support execution of programs on top of the standard microprocessors models. This solution offers both passive and active debugging, gives a non-intrusive view of the program flow, and allows fine control over program execution, CPU state and memory contents. But, they cannot be used for debugging if the actual hardware is involved.

2.1.3 Debug stubs

Debug stubs are the programs that run on target hardware and communicate with the debug software on host computer. Initially the stub program establishes connection with the host before handing over the control to the main function. During execution, stubs are called whenever exceptions occur. Being a software solution, its unsuitable for early development stages where the initialization code itself has to be debugged. Since this does not require any additional hardware, debug stubs are preferred for systems with a functioning operating system.

2.1.4 Integrated software and hardware solutions

Integrated debug hardware gives the power of in-circuit emulators with much higher flexibility. The target microprocessors will contain debug functionality in hardware, that could connect to a host computer through a serial communication channel. Using the host computer, user will be able to remotely debug the programs running on target hardware. This is the best possible debug solution, but involves additional hardware and increases design complexity along with a cost overhead.

2.2 Existing debug solutions

Since debugging is a critical part of embedded software development, there are many commercial and open source tools readily available. Most of them are for either x86 or ARM[1] architectures, the commonly used microprocessors in the industry. A few of the existing debug solutions for the architecture of AJIT processor are described in the following sections.

OpenOCD : The Open On-Chip Debugger (OpenOCD)[2] is an open source tool for debugging, in-system programming and boundary-scan testing for embedded target devices created by Dominic Rath[4]. The debugging is done with the assistance of a hardware adapter, which is a small module that communicates from the target being debugged. Multiple such

[1]http://www.arm.com/
[2]http://www.openocd.org/

adapters have been developed (even as dongles) over the time, and many of them support different communication protocols as well. Currently the project supports ARM7, ARM9, XSCALE, MARVELL[3] and MIPS processors and use JTAG interface to communicate with the target.

Advanced on-chip debug support on LEON : LEON processor is a 32-bit processor core based on the SPARC V8 architecture, developed by Aeroflex Gaisler[4]. They have a debug monitor named GRMON2[5] that provides most of the advanced debugging features like read/write access to all registers and memory, breakpoint and watchpoint management, etc.. over a variety of interfaces like PCI, USB, Ethernet, JTAG, UART and SpaceWire[5]. This debugger is part of the supporting applications commercially available for the LEON family of microprocessors.

Oracle Solaris Studio dbxtool : An easy-to-use graphical interface combined with the debugging functionality of the Oracle Solaris Studio[6] dbx debugger ,the tool `dbxtool` was created[6]. The program allows users to debug programs running on an executable or core file or by attaching to a running process. The tool also has the ability to save trace output, manage events, detect run time errors, modify source code, and save and re-run a debugging run as well.

[3]http://www.marvell.com/
[4]http://www.gaisler.com/
[5]http://spacewire.esa.int/
[6]https://www.oracle.com/solaris/studio/

Chapter 3

System Architecture

The proposed system with integrated hardware-software architecture provides a non-intrusive view of the processor and the ability to remotely modify it at any instant. It has a software subsystem with the standard interface of GNU Debugger (GDB)[7] that communicates with processor hardware. The other part is a hardware subsystem consisting of additional modules with the processor core for supporting these debug commands from the user.

3.1 Software subsystem on host

The software subsystem provides sophisticated debugging options to the user with an intuitive interface. As shown in Figure 3.1 it consists of two main components. GDB is chosen as the user end debugging utility because of the open source nature and acceptance among software developers. It offers extensive facilities for tracing and altering the execution of programs across different architectures and programming languages. The debugging information will be added to the object file if appropriate options are used for compiling. These details could be the correspondence between the lines of source code and the executable code , and the memory address of all the variables. GDB has to be provided with the same executable running on the AJIT processor, but compiled with debug flags.

The other part of software subsystem is a server bridge that pretends to be the processor hardware for GDB and then translates its complex messages into a simple set of commands to be sent over to the hardware. The software server bridge and GDB could be running on different computers, and their communication link is termed as *RSP interface.* Server bridge allows the users to choose the type of physical connection to be used for *Hardware debug link* that connects it to the AJIT processor.

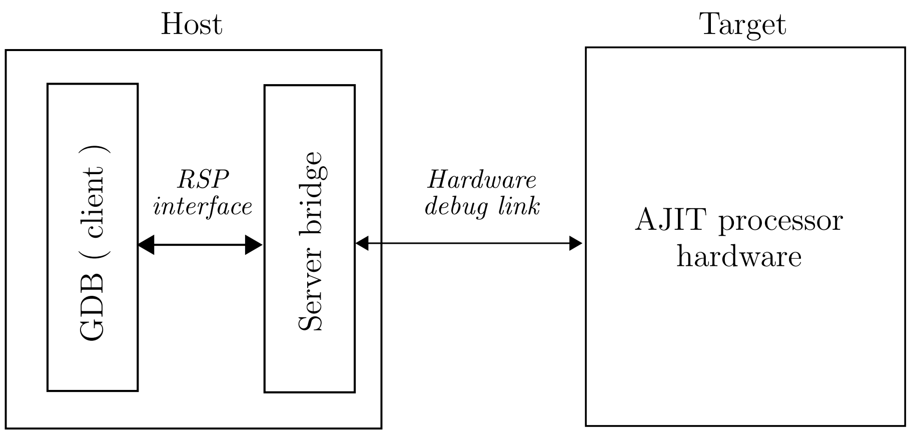


Figure 3.1: In-system debugger : System architecture

3.1.1 GNU Debugger (GDB)

The GNU Debugger[1] (GDB) is a standard open source debugger extensively used by software developers. It allows users to view and control the execution of other programs and is released under GNU General Public License (GPL)[2] .

Functionalities and usage

GDB allows the users to monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behaviour. Tracing the program execution and altering the hardware state at any instant becomes very easy with GDB. It offers quite a lot of functionalities, mainly the following four with a few other supporting ones.

- Start the program by specifying all the details for running
- Stop the program execution on certain conditions like break-points, watch-points, exceptions etc. and advance only when user instructs so
- Examine / modify the register & memory contents while the execution is stopped temporarily
- Change values and conditions in the program, to experiment and figure out the possible reasons for bug

GDB can debug programs running on the same machine (native), or on another machine (remote). In remote debugging, GDB runs on a normal full fledged computer (termed as *Host*) and program gets executed on a smaller machine (termed as *Target*) and they communicate using the Remote Serial Protocol (RSP)[8]. The communication interface between the two machines could be a serial line, or an IP network using TCP or UDP as the underlying protocol.

[1]https://www.gnu.org/software/gdb/
[2]https://www.gnu.org/licenses/

Most of the embedded system debugging is done remotely, and the same will be used in the proposed architecture.

Remote Debugging

Remote debugging is used to debug programs running on machines that cannot run GDB in the standard way or smaller systems without the resources to run full-featured debugger on it. The GDB on host machine requires a copy of the program (which is to be debugged) executable with all the debug symbols generated by the compiler. Target machine can be configured to communicate in two different methods:

- **Using gdbserver** : This is a much smaller version of the standard GDB that helps to connect the target hardware with the host and communicate. It needs an underlying operating system on the target hardware to work.
- **Using debug stubs** : Debug stubs are a collection of files containing special function subroutines that support GDB remote serial protocol on the target. All the services provided by stubs can also be implemented as dedicated hardware modules on target, as being done in this work.

3.1.2 Remote Serial Protocol (RSP) interface

The RSP interface existing between GDB and the software server bridge uses the same RSP data transfer scheme used for communicating with remote targets. Instead of the actual hardware, here a moderator program receives all the RSP commands and responds back.

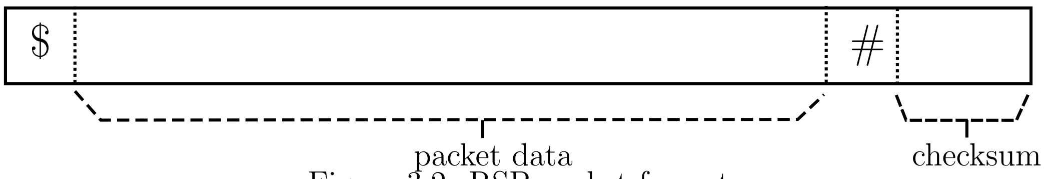


Figure 3.2: RSP packet format

All the communication interfaces supported by GDB could potentially be used here for RSP interface. Here the GDB on the host side acts as RSP client for the software server bridge. Client can send requests to the server as packets, expecting values or acknowledgement message in return. The format of a single RSP packet is shown in Figure 3.2.

Here $ and # function as the head and tail characters respectively. The 8-bit checksum is calculated as modulo 256 of unsigned sum of all characters in the packet, and represented by a pair of hexadecimal digits. Each packet is acknowledged by a '+' on satisfactory reception and '-' to request a re-transmission. The packets can be categorized into three different categories.

1. Expecting no response message.
2. Requires simple acknowledgement messages or error code.
3. Expects only data in the response packet.

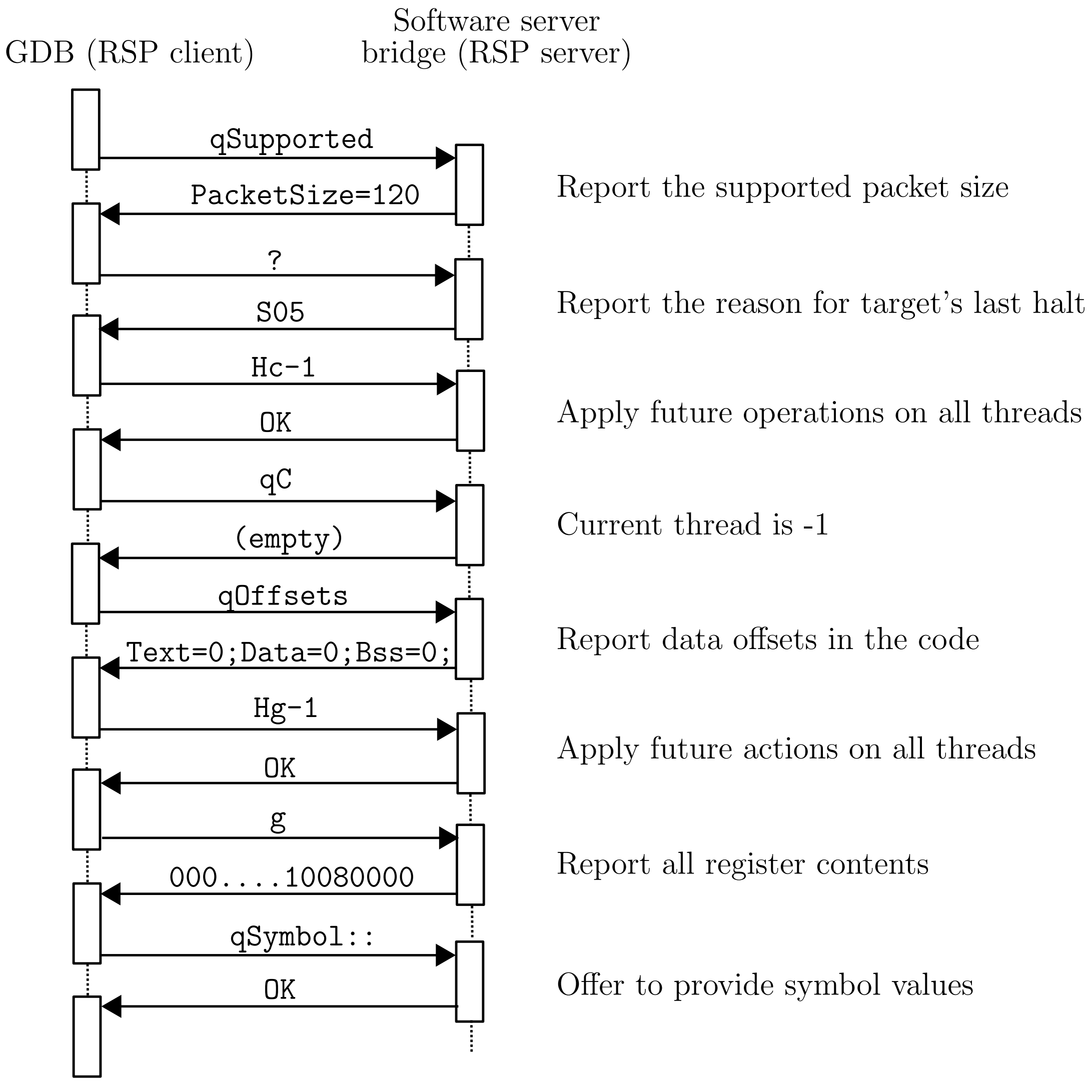

GDB (RSP client)
Software server bridge (RSP server)
qSupported
PacketSize=120
Report the supported packet size
?
S05
Report the reason for target's last halt
Hc-1
OK
Apply future operations on all threads
qC
(empty)
Current thread is -1
qOffsets
Text=0;Data=0;Bss=0;
Report data offsets in the code
Hg-1
OK
Apply future actions on all threads
g
000....10080000
Report all register contents
qSymbol::
OK
Offer to provide symbol values

Figure 3.3: RSP packet exchange

Each command entered by user in GDB on the host machine is actually mapped to multiple RSP commands for carrying out corresponding operation. An example RSP communication of connection establishment is shown in Figure 3.3. The simple command on GDB triggers a series of RSP communication between the client and server.

First of all the client (GDB on host) requests the maximum packet size supported through the link. Second query is to know why the target halted, and verify that it did not happen because of an terminate command (X) or exited (W). Then it send a `qC` command, to get the information about the thread being executed. `qOffset` requests the memory offsets for loading binary data. Then the values of all registers are requested by `g` command. Finally the target will provide any symbolic data as per the request from host.

All the standard RSP packets are supported by the proposed in-system debugger and they are described in Appendix I.

3.1.3 Software server bridge

Software server bridge program runs indefinitely on the host machine and acts as a translator between the GDB and the AJIT processor. The server continuously listens on the RSP interface expecting GDB message packets which are then processed to retrieve the underlying information. Based on the decoded message, the software server bridge will generate commands for the hardware and send them. Now it will wait for a valid response from the target which then has to be encoded in RSP packet format and sent to GDB. If the server bridge receives an invalid RSP command, then it will reply back immediately.

The standard RSP protocol is very complex and contains a lot of commands that server needs to understand and support. It is partly because the GDB is designed to support different types of hardware and the functionalities have been greatly improved by the open source community. But, here the target hardware is fixed and a lot of these GDB commands can be filtered and simplified further. The software server bridge carries out this filtering and simplification operation. Communication with the actual hardware is can be limited for the necessary set of GDB requests. Essentially the server bridge takes care of all the following tasks.

1. **RSP connection :** Establish connection with GDB using the type of RSP interface the user has selected.
2. **Packet reception and decoding :** Receive and decode the RSP packets sent by GDB. Message content extraction and checksum verification are done for all the RSP packets. If the command does not require any information from hardware, then the reply is sent back immediately and if not, further levels of translation are required. The messages could be
    1. Unsupported commands that will be replied with an empty packet ($#00) to indicate they are invalid.
    2. Commands that could always get a constant response from AJIT processor will be directly replied back by the server bridge itself.
    3. The third type commands requires actual data from the hardware.
3. **Translation to debug link commands :** The third type of commands mentioned above

are translated into single or multiple debug link commands.

4. **debug link connection :** The final task is to pass the commands over the debug link and read the responses back if necessary.

The algorithm of software server bridge is given in Algorithm 1.

Algorithm 1 Software server bridge

```
1: function SSB_DAEMON
2:     $rsp\_interface \leftarrow$ user input
3:     $debug\_link\_type \leftarrow$ user input
4:     RSP_CONNECT($rsp\_interface$)                                    ▷ Connect to the RSP interface
5:     while $rsp\_msg \neq CONNECT$ do                                  ▷ Wait till GDB connects
6:         $rsp\_msg \leftarrow recv\_rsp\_interface$
7:     end while
8:     $send\_rsp\_interface \leftarrow$ DEBUG_LINK_CONNECT($debug\_link\_type$)        ▷ HW connect
9:     while true do                                                     ▷ Infinite loop
10:        $rsp\_msg \leftarrow recv\_rsp\_interface$                    ▷ generate response for RSP command
11:        $rsp\_msg\_decoded \leftarrow$ RSP_DECODE($rsp\_msg$)
12:        if type($rsp\_msg\_decoded$) = hardware_dependant then
13:            $dbg\_lnk\_msg \leftarrow$ RSP_TO_DEBUG($rsp\_msg\_decoded$)
14:            $debug\_link\_reply \leftarrow$ SEND_DEBUG_LINK_AND_GET_RESPONSE(dbg_lnk_msg)
15:        else if type($rsp\_msg\_decoded$) = standard then
16:            $debug\_link\_reply \leftarrow$ STANDARD_REPLY($rsp\_msg\_decoded$)
17:        else
18:            $debug\_link\_reply \leftarrow$ INVALID
19:        end if
20:        $send\_rsp\_interface \leftarrow$ DEBUG_TO_RSP($debug\_link\_reply$)
21:    end while
22: end function
```

3.1.4 Hardware debug link

The debug link between the software server bridge and the target hardware is treated as a standard serial communication link for simplicity. Different physical connections and protocols could actually be used, but the message format will remain the same. One debug link command could be one, two or three words long depending on its type. Simple commands like *connect* and *detach* requires just a single word, where as memory/register read/write will need 2/3 words.

Each of these packets have one instruction word followed by one or two optional data words. Format of instruction word is shown in Figure 3.4. First eight bits of the word specifies the total length (1, 2 or 3) of the packet. The next eight bits contains the opcode, and the next 16 bits are interpreted differently for each opcode. All the standard debug link commands are described in Appendix II.

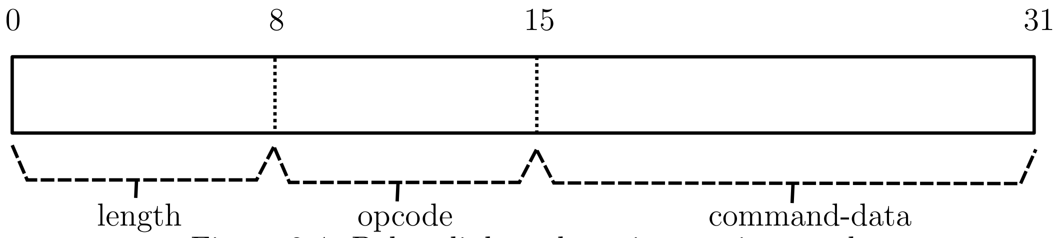


Figure 3.4: Debug link packet : instruction word

For 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 the corresponding special function register will be accessed. In break-point and watch-point commands (opcodes 8,9,10,11) the reg-id can have only values only 0 to 4, limiting the support up to 4 watch-point and break-point registers each. The second command word will specify the data for all register & memory writes and break-points & watch-point commands, address for all memory reads and instructions for instruction execution. The third command specifies data for memory writes.

3.2 Hardware subsystem on target

The commands coming through debug link are executed by the debug supporting units included in the processor hardware. Design of these processor specific units heavily depend on the implementation structure. The AJIT processor development has progressed in two parallel directions to make the verification process easier. The final objective is to create the models of processor at different levels, satisfying the guidelines of SPARC V8 architecture manual[2].

First model of the processor is completely implemented in C [9] [10] including the caches and MMU, that could execute binaries compiled for SPARC targets. It is an attempt to create a completely verified, ISA level simulator of the architecture to support future developments. The second model description faithfully represents the actual processor hardware including the pipeline. This representation could be converted into the hardware description, simulated and eventually ported to FPGA. Individual blocks of this model can be modified at different levels to improve the overall efficiency. The hardware subsystem was designed for both the models of AJIT processor according to their structure.

3.2.1 Hardware subsystem in ISA-C model

ISA level model of the AJIT processor has all the individual components modelled with parallel threads. There is a testbench that mimics the external environment of the processor, and provide the initial memory map to start the execution. The model also has specific driver functions to read from and write to the standard I/O.

The main CPU core thread implements instruction fetch, decode and execute along with exceptions checks. The implemented memory subsystem contains Virtually Indexed Virtually Tagged (VIVT) instruction and data caches since they are the easiest and fastest design to implement. Model uses paging technique to use virtual memory, and refer a 3-level page table to get mapping of a virtual address. Both TLB and CPU caches are fully associative, with dual port TLB handling request from both instruction and data caches. The primary aim of this model is to work as a simulator for executing SPARC binaries and provide a reference behaviour for the hardware models. The debug support in ISA-C model is provided by a hardware server connected to the processor core as shown in Figure 3.5.

Hardware server

The Hardware server receives the debug link commands from the software server bridge. It has the ability to communicate with the processor core and memory separately. When the processor is turned on in debug mode, it will inform the hardware server and wait for a *continue* signal back. The hardware server will now wait for the connection request from GDB and once its established, all subsequent commands (read/write registers, set/clear break-points etc..) will be executed. The hardware server will allow the processor to run when it gets a *continue* message from software server bridge.

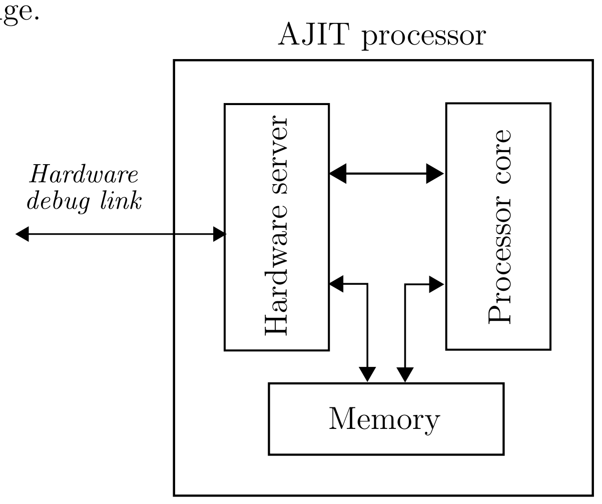


Figure 3.5: Hardware server on ISA-C model

In every execution cycle after instruction fetch, processor core will inform the current state and interrupt status to the hardware server and wait for *continue*. Now the hardware server compares the current Program Counter (PC) with all the valid break-point registers and memory address with all the valid watch-point registers. If there is no match, then continue signal is sent right away. Processor continues to write-back the results after getting the continue message. In cases of watch-point or break-point hit, the message is encoded and passed on to the software server bridge. Processor core will inform hardware server in the case of exceptions and interrupts as well. After receiving any such message, server bridge can send commands to view/modify the processor register/memory contents. It can also command the hardware server to set up new break and watch points, which the hardware server stores in one of the 8 local registers.

3.2.2 Debug units in micro-architecture model

The micro-architecture model faithfully represents the actual processor hardware including the pipeline. It has three main parts : CPU Control Unit (CCU), Thread Execution Unit (TEU) and Memory Interface Unit (MIU). They are shown in Figure 3.6 and described below.

1. **CPU Control Unit :** When the processor is powered on, the control will be with the CPU Control Unit (CCU). The processor can be ran in normal or debug or single step mode based on the reset value provided. After initializing the processor registers, CCU transfers control (along with the current PC, NPC values) to the TEU.
   CCU will obtain the control back only when exception occur in the TEU. Then it will execute the trap handler and inform the TEU to stop fetching further instructions. After the CCU completes its part, TEU is allowed to continue execution and provided with PC, NPC values. CCU sends out the current processor mode every cycle for the testbench to check if it has entered the error mode.
2. **Thread Execution Unit :** TEU is essentially the actual processor pipeline that decides the actual processor throughput. It starts execution when the CCU hands over the PC, NPC set. The model has an elastic pipeline with subsequent stages connected using interlocking FIFOs (AHIR-V2 pipes) independent of any timing assumptions as shown in Appendix III.
   Instructions in execution stage could be out of order, but they get ordered at the retire stage, making this an in-order pipeline. All instructions are given an 8-bit slot id by the fetch stage to preserve the order. Checks for interrupts are done during instruction fetch and all the load/store operations are strictly maintained to be in order. All exceptions are treated as precise, which requires all blocks to be synchronized. Exceptions occurring

at intermediate stages of pipeline are passed on to the retire stage and finally to CCU. All the subsequent instructions will pass through the pipeline without altering the processor state. Finally, the execution will continue normally when the CCU hands over a new set of PC, NPC values.

3. **Memory Interface Unit :** MIU is just the interface for the processor to read / write data from / to the main memory.

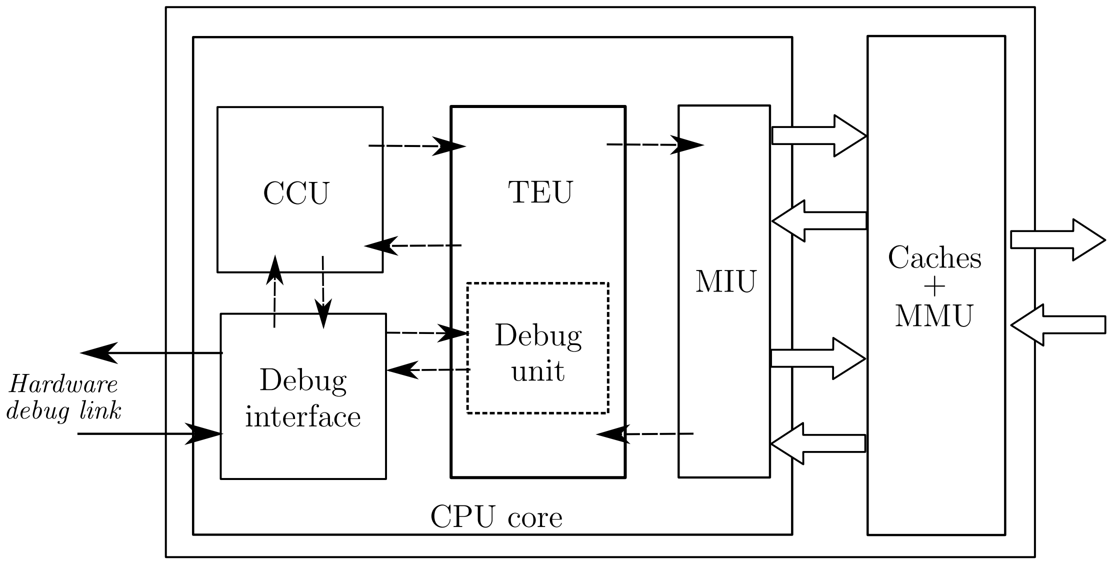


Figure 3.6: Debug units in micro-architecture model

The CCU and TEU were modified and two additional modules were added to the processor for supporting the debug functionalities proposed in this work and the final system is shown in Figure 3.6. The Debug interface unit is added to receive messages through the debug link, decode all the valid commands and generate their response. The break-point & watch-point related requests are handled by the the TEU debug unit and the rest by CCU.

Debug interface unit

The debug interface unit receive all the messages coming through the debug link and send the responses back. It filters out all the break-point and watch-point related messages and forward them to the TEU debug unit. All the watch-point registers values are also locally stored. The requests for memory/register read/write operations are passed on to the CCU. Their responses are then sent back to the software server bridge through the debug link. When there is a break-point or watch-point hit, trap occurrence, interrupt etc.. the debug interface will be informed by the CCU for passing it on to the software server bridge. In cases of a watch-point hit, its stored address will also be sent as a second the packet.

Algorithm 2 Debug interface unit

```
1: function DEBUG_INTERFACE_DAEMON
2:     while $ccu\_msg \neq CONNECT$ or $gdb\_msg \neq CONNECT$ do        ▷ CCU, GDB connect
3:         $ccu\_msg \leftarrow ccu\_to\_debug\_pipe$
4:         $gdb\_msg \leftarrow debug\_link\_in$
5:     end while
6:     $debug\_to\_ccu\_pipe, debug\_link\_out \leftarrow Acknowledge\_OK$
7:     while 1 do                                                        ▷ Infinite loop
8:         $ccu\_msg \leftarrow ccu\_to\_debug\_pipe$                      ▷ Message from CCU
9:         if (ccu_msg is valid) then
10:            Decode ccu_msg
11:            if (breakpoint or watchpoint or trap hit or interrupt) then
12:                $stored\_PC, stored\_NPC, stored\_PSR \leftarrow ccu\_to\_debug\_pipe$
13:            end if
14:            $debug\_link\_out \leftarrow$ Encode $ccu\_msg$ for GDB
15:            if (watchpoint(x) hit) then
16:                $debug\_link\_out \leftarrow$ stored adddress of x
17:            end if
18:        end if
19:        $gdb\_msg \leftarrow debug\_link\_in$                           ▷ Message from Debug link
20:        if ($gdb\_msg$ is valid) then
21:            Decode $gdb\_msg$
22:            $debug\_to\_ccu\_pipe \leftarrow gdb\_msg$
23:            if (msg_length = 2) then
24:                $gdb\_msg\_2 \leftarrow debug\_link\_in$
25:                $debug\_to\_ccu\_pipe \leftarrow gdb\_msg\_2$
26:            end if
27:            if (msg_length = 3) then
28:                $gdb\_msg\_3 \leftarrow debug\_link\_in$
29:                $debug\_to\_ccu\_pipe \leftarrow gdb\_msg\_3$
30:            end if
31:            if (PC or NPC or PSR write) then
32:                $stored\_PC$ or $stored\_NPC$ or $stored\_PSR \leftarrow gdb\_msg\_2$
33:            else if (watchpoint(x) write) then
34:                stored adddress of x $\leftarrow gdb\_msg\_2$
35:            else if (read the memory / register) then
36:                $ccu\_data \leftarrow ccu\_to\_debug\_pipe$
37:                $debug\_link\_out \leftarrow ccu\_data$
38:            else if (CONTINUE or DETACH) then
39:                $debug\_to\_ccu\_pipe \leftarrow stored\_PC, stored\_NPC, stored\_PSR$
40:            end if
41:            if ((set/clear the breakpoint/watchpoint) or (write memory / register)) then
42:                $debug\_link\_out \leftarrow Acknowledge\_OK$
43:            end if
44:        end if
45:    end while
46: end function
```

Details of the unit are given in Algorithm 2. Every time the CCU stops and the debug interface is notified, the current values of Program Counter (PC), Next Program Counter (NPC), and Program Status Register (PSR) values are also send along with it. The debug interface unit locally stores these values and modifies them if user requests so. Finally when it receives a *continue* message, the debug interface sends back the their updated values to the CCU.

Modified CCU

The CCU is modified as given in Algorithm 3 to support the debug operations. Now it has to be stopped initially (for connecting with debug interface unit), on hitting watch-points/break-points, on interrupts, and other exceptions for transferring the control over to the interface unit. The commands of register/memory read and write from the software server bridge are executed by the CCU when it is stopped. Finally when the CCU receives a *continue* message, it resumes execution and the control is finally passed on to the TEU. CCU has separate connections to the TEU for reading and modifying the register contents, and to MIU for reading and modifying the memory contents.

Algorithm 3 CCU

```
 1: function CPU_CCU
 2:     $debug\_mode \leftarrow ENABLE$
 3:     Initialize processor
 4:     $ccu\_to\_debug\_pipe \leftarrow CONNECT$
 5:     while $debug\_msg \neq Acknowledge\_OK$ do          ▷ Establish connection with debugger
 6:         $debug\_msg \leftarrow debug\_to\_ccu\_pipe$
 7:     end while
 8:     CCURespondToGDB(GDB_CONNECTED,0)                  ▷ Execute GDB commands
 9:     while 1 do                                         ▷ Infinite loop
10:         ...
11:         if (error_mode) then
12:             CCURespondToGDB(ERROR_MODE,0)
13:         end if
14:         ...
15:         if (trap) and ($debug\_mode = ENABLE$) then
16:             CCURespondToGDB(TRAP,0)
17:         end if
18:         ...
19:         $pass\_to\_teu \leftarrow 1$
20:         while $pass\_to\_teu$ do
21:             $ccu\_to\_teu \leftarrow CONTINUE$
22:             $teu\_msg \leftarrow teu\_to\_ccu\_pipe$         ▷ wait till TEU send back a response
```

```
            if (breakpoint hit) and ($debug\_mode = ENABLE$) then
                CCURespondToGDB(BP_HIT,reg)
            else if (watchpoint hit) and ($debug\_mode = ENABLE$) then
                CCURespondToGDB(WP_HIT,reg)
            else if (interrupt) and ($debug\_mode = ENABLE$) then
                CCURespondToGDB(INTR,0)
            else
                $pass\_to\_teu \leftarrow 0$
            end if
        end while
    end while
end function
function CCURespondToGDB(stop_reason,reg)
    $ccu\_to\_debug\_pipe \leftarrow$ Encode (stop_reason,reg) for debugger
    if ( stop_reason = breakpoint or watchpoint or trap hit or interrupt) then
        $ccu\_to\_debug\_pipe \leftarrow PC, NPC, PSR$
    end if
    $debug\_msg \leftarrow debug\_to\_ccu\_pipe$
    while $debug\_msg \neq CONTINUE$ do        ▷ Execute GDB commands till CONTINUE
        Decode $debug\_msg$
        if (read register x) then
            $ccu\_to\_debug\_pipe \leftarrow$ content of register x
        else if (write register x) then
            content of register x $\leftarrow debug\_to\_ccu\_pipe$
        else if (read memory) then
            $mem\_address \leftarrow debug\_to\_ccu\_pipe$
            $ccu\_to\_debug\_pipe \leftarrow$ content of $mem\_address$
        else if (write memory) then
            $mem\_address \leftarrow debug\_to\_ccu\_pipe$
            $mem\_data \leftarrow debug\_to\_ccu\_pipe$
            content of $mem\_address \leftarrow mem\_data$
        else if (set break-point/watch-point x) then
            $ccu\_to\_teu\_debug\_unit \leftarrow$ Encode (SET, break/watch, x, $debug\_to\_ccu\_pipe$)
        else if (remove break-point/watch-point x) then
            $ccu\_to\_teu\_debug\_unit \leftarrow$ Encode (REMOVE, break/watch, x)
        else if (Interrupt) then
            $ccu\_to\_teu\_debug\_unit \leftarrow$ Encode (Interrupt)
        else if (DETACH) then
            $debug\_mode \leftarrow DISABLE$
            break
        end if
        $debug\_msg \leftarrow debug\_to\_ccu\_pipe$
    end while
    if ($debug\_msg =$ CONTINUE or DETACH) then
        $PC, NPC, PSR \leftarrow debug\_to\_ccu\_pipe$
    end if
end function
```

TEU Debug unit

The commands from debug interface to set or clear the break-points and watch-points are executed by storing them locally in the TEU debug unit. During every instruction execution, the PC address is compared with these stored valid break-points. For the memory access instructions, the address will also be compared against the valid watch-points. If either of the checks succeeds or an exception or interrupt occures, the TEU debug unit will inform the retire unit of TEU to stop the program execution and transfer the control back to CCU. The debug interface can use this unit to interrupt the program execution (TEU) at any instant as per the software server bridge command. More details are given in Algorithm 4.

Algorithm 4 TEU

```
1: function TEU_DEBUG_UNIT
2:     while 1 do                                                  ▷ Infinite loop
3:         ccu_command ← ccu_to_teu_debug_unit
4:         Decode ccu_command
5:         if (ccu_command = set break-point/watch-point x) then
6:             break-point/watch-point (x) ← ccu_command_data, valid
7:         else if (ccu_command = clear break-point/watch-point x) then
8:             break-point/watch-point (x) ← invalid
9:         else if Interrupt then
10:            teu_debug_to_retire ← interrupt
11:        end if
12:        if (fetch_address = content of breakpoint register x) then
13:            teu_debug_to_retire ← Breakpoint x hit
14:        end if
15:        if (memory_address = content of watchpoint register x) then
16:            teu_debug_to_retire ← Watchpoint x hit
17:        end if
18:    end while
19: end function
20: function TEU_MAIN
21:    while 1 do                                                  ▷ Infinite loop
22:        PC, NPC ← ccu_to_teu
23:        while (bp_wp_intr not hit) and (no exceptions) and (no interrupt) do
24:            Instruction fetch & decode
25:            to_teu_debug_unit ← fetch_address, memory_address
26:            Execute
27:            bp_wp_intr ← teu_debug_to_retire
28:        end while
29:    end while
30: end function
```

Chapter 4

Implementation & validation

The proposed in-system debugger has been implemented for all the AJIT processor models and validated with several test programs. The software subsystem is described in C and the hardware subsystem descriptions are according to the corresponding processor models are designed for.

4.1 Implementation

4.1.1 Software subsystem and Debug link

The cross GDB to be used for debugging the AJIT processor programs is generated by compiling the standard GDB source files with *–target=sparc-buildroot-linux-uclibc* option. It can be invoked by *sparc-linux-gdb* command on the host machine terminal. The source files have to be compiled with the `gcc`[11] cross compiler debug flag (*-g*) to use them for debugging.

In the ISA-C model source code, the software server bridge is referred as GPB (GDB to Processor Bridge). It is implemented as single C thread, compiled to form a library named *gpb*. Later the complete processor executable is created by compiling the core model and linking to this library. Currently the software subsystem supports only one RSP interface, the unix sockets. Functions in *StartGPBThreads.c* initializes everything and starts the main thread, *gpb.c* contains the main thread that listens on the socket and carries out the all the functionalities. *GDBtoAJITbridge.c* contains all the functions necessary to communicate over RSP interface (GDB to Server bridge) and debug link (Server to Processor). The implementation uses two data structures, one for representing RSP commands (*rsp_cmd*) and other for representing the debug link commands (*dbg_cmd*). All the standard functions required for threads and socket communications are used from the `gcc` libraries.

The current implementation supports only a single type of connection for the debug link, the AHIR-V2 [12] pipes. *ENV_to_AJIT_debug_command* and *AJIT_to_ENV_debug_response* are the two 32-bit pipes used to communicate with the AJIT processor hardware. A pipe is just essentially a FIFO with handshaking signals that provide a blocking nature to the data transfer[9]. They can be used for inter-thread communication in multi-threaded programs. A read request for a pipe will not succeed before some data is written to it and vice-versa. AHIR-V2 provides special read and write functions to facilitate this behaviour. Since the execution is blocked while expecting some data on a pipe, this can also be used as a locking mechanism. This is kind of a hardware mutex similar to those used in software (mutex functions in `pthread` library).

4.1.2 Debugging on ISA-C model

Each block of the ISA-C model is described as a thread, and so is the hardware server. It has three main functional blocks : for handling debug link communications, the processor side communication and a core unit to execute the commands sent by software server bridge. Processor core ISA-C model files (*sparcCore.c*, *execute.c*) are modified to achieve the required control flow.

Processor and hardware server use request-acknowledge mode communication through unidirectional AHIR-V2 pipes. When the processor needs to communicate something, it sends the corresponding opcode through *CPU_HWSERVER_id* (8-bit pipe) and the current processor state pointer through *CPU_to_HWSERVER_state* (64-bit pipe). The received pointer is used by the hardware server to modify the processor state. Hardware server uses *HWSERVER_to_CPU* (32-bit pipe) for sending messages back to the processor pipeline.

The processor core, hardware server and testbench are compiled together to form a single executable. The memory map of the program and the unix socket port number should be given for running the executable and starting a debug session.

4.1.3 Debugging on micro-architecture simulation model

The micro-architecture model is described in Aa[13] language, representing the actual hardware of AJIT processor. Even though it is an intermediate representation in the AHIR-V2 toolchain, it can be also used to describe hardware. Such representations could be converted into C language for verification and to VHDL for hardware realization by the AHIR-V2 compiler. It supports the use of same software testbench for hardware and software simulations.

The model has an elastic pipeline with and all the modules are interconnected by AHIR-V2 pipes. Currently the processor core is described in hardware and the memory and caches are the same ones used with ISA-C model. There are basically two simulation models derived from the micro-architecture Aa description : C, and VHDL. The C code generated from the Aa representation can be compiled together with the testbench to produce a single executable similar to the ISA-C model. The usage and everything else remains the same as the ISA-C model.

The second derived model is by the simulation of generated VHDL description using ModelSim[1]. It simulates the core processor module and the testbench connects through the VHPI (VHDL Procedural Interface) library provided by AHIR-V2. The system will behave just like the actual hardware would, and provides a good platform for design verification before moving to hardware.

4.1.4 Debugging on micro-architecture FPGA prototype

The VHDL description generated by AHIR compiler is synthesis ready. The design is synthesized with Vivado® Design suite and loaded on to the Xilinx[2] Virtex-7 VC709 FPGA card. The complete processor hardware with included debug units run at 100MHz on this FPGA prototype.

The communication medium between the testbench and processor hardware is different when it is on the FPGA. The physical interface will be the PCI Express bus, and necessary framework is provided by RIFFA 2.2 (**R**eusable **I**ntegration **F**ramework for **F**PGA **A**ccelerators)[14] platform. This open source framework provides additional hardware modules and software drivers for synchronizing and communicating FPGA based applications with computers. The complete system architecture of the FPGA prototype with RIFFA is given in Figure 4.1.

The software server bridge and the GDB communicate through RSP interface as described in the previous section. The server bridge and testbench interact through the same AHIR-V2 pipes as mentioned before. Here the testbench use RIFFA drivers for communicating with FPGA PCI Express block. The RIFFA module in hardware will interface this block with the actual AJIT processor with in-built debug modules.

4.1.5 AHIR-V2

AHIR (**A** **H**ardware **I**ntermediate **R**epresentation) compiler[12] is the high level synthesis tool set developed at IIT-Bombay. Systems can be described in higher level languages like C/C++ as

[1]http://www.model.com/

[2]http://www.xilinx.com/

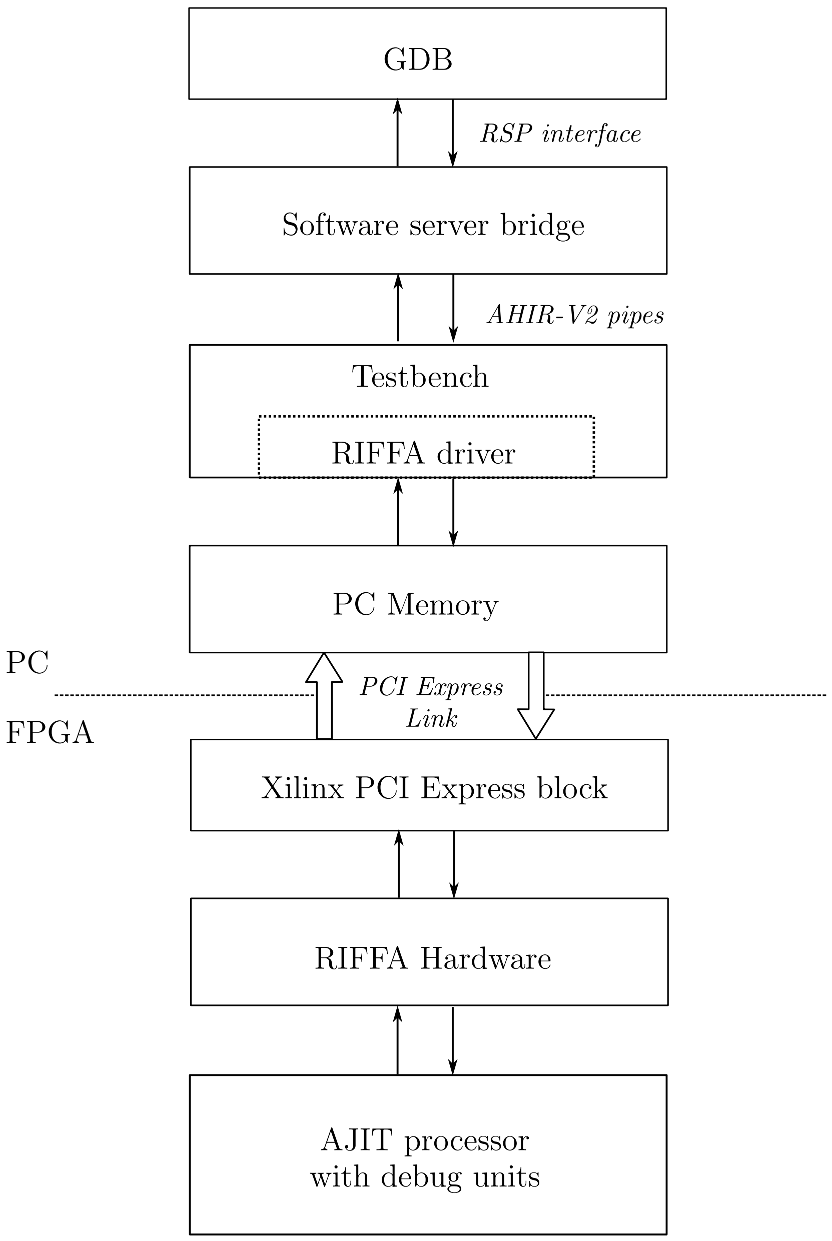


Figure 4.1: In-system debugger on FPGA prototype

algorithms and then converted into VHDL digital system descriptions using the AHIR toolchain.

AHIR tools has a two level approach for this high level synthesis process. First it converts the code written in the higher level language to byte code using LLVM[3] compiler. The byte code is subsequently converted into an intermediate representation called *Ahir Assembly* (Aa) language that uses petri nets as the basic data structure mechanism. The Aa representation is optimized and later transformed into VHDL through one more intermediate representation called *Virtual Circuit* (vC). Along with this the compiler also allows conversion of Aa representations

[3]http://llvm.org/

into C for the easiness of design verification. The complete design flow is shown in Figure 4.2. AHIR factorizes any system into three components: control path, data path and storage. AHIR framework and methodology is correct by construction. i.e., the hardware produced from the high level language is the exact functional transformation of the input.

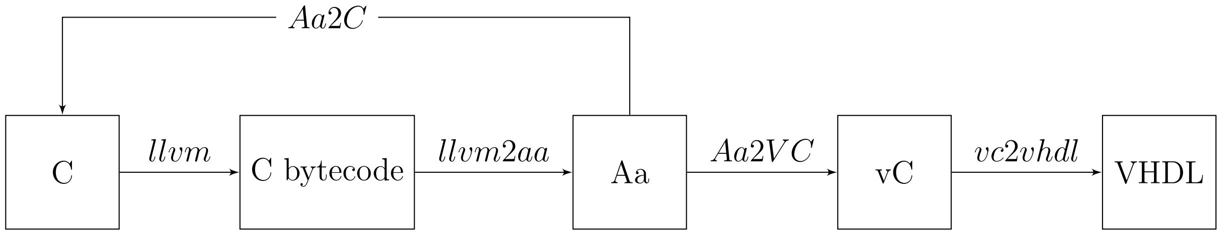


Figure 4.2: AHIR flow

4.2 Usage and Validation

4.2.1 Using in-system debugger

The program for debugging has to be compiled using `sparc-linux-gcc` cross compiler with *-g* flag for including the debug information. Then the `sparc-linux-gdb` has to be invoked by providing this executable as the first argument. Now the appropriate testbench has to be started by providing the memory map (generated using the same compiler) of the same program and the port for the debugger to connect. These testbenches could correspond to any of the four AJIT processor models available

- ISA-C model
- Micro-architecture model in C
- Micro-architecture VHDL model running in simulator
- Micro-architecture model on FPGA prototype

If the model is running on simulator, it has to be started separately. The user will now be able to connect to the AJIT processor from the GDB using the following command.

```
(gdb) target remote :<port number>
```

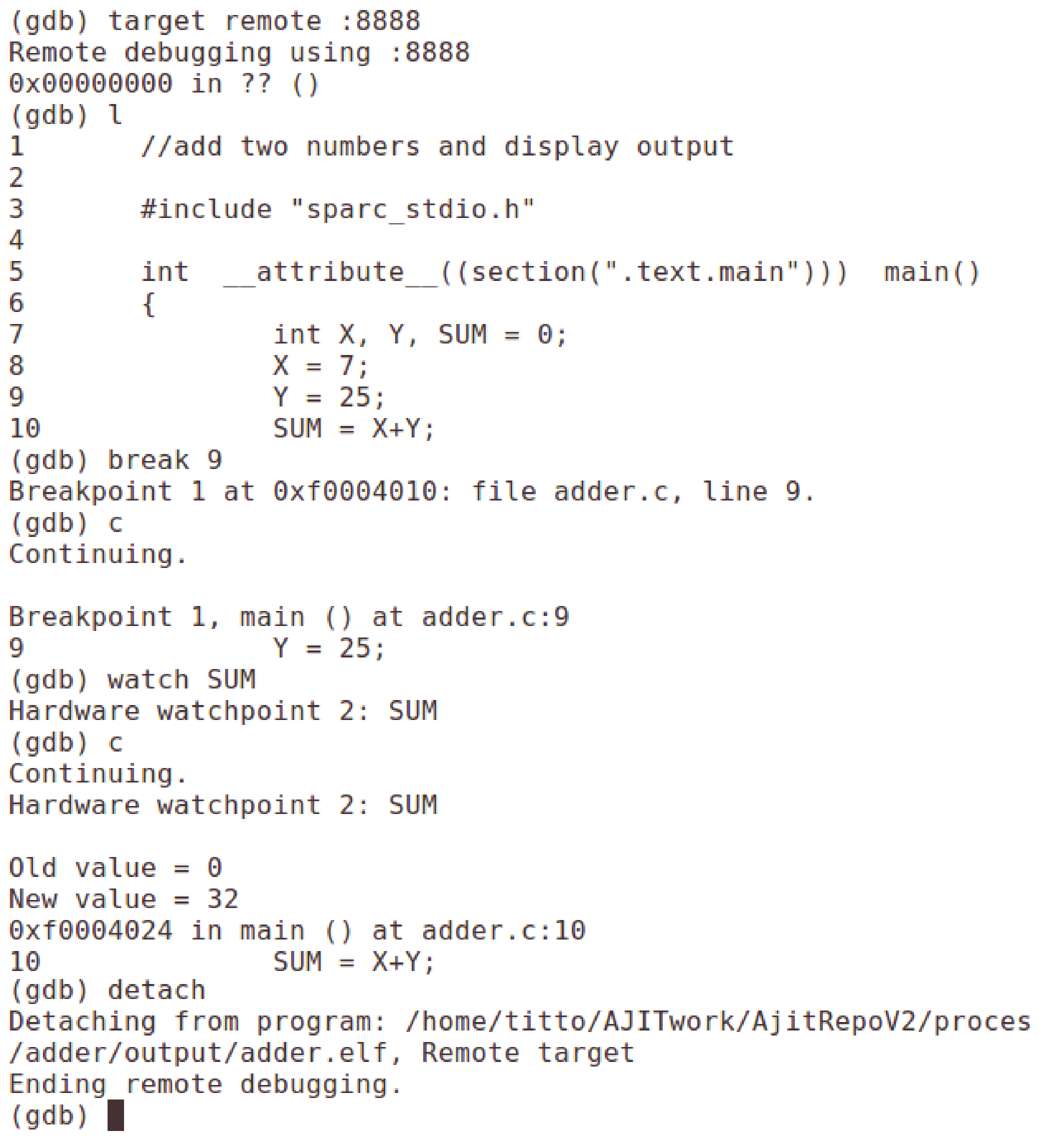
The debugging session of AJIT processor has now been started, and the users can continue using it as a regular GDB session.

4.2.2 Validation on FPGA prototype

Several test programs running on the AJIT processor FPGA hardware prototype are were debugged using this in-system debugger to validate the current implementation.

An example session of in-system debugging is shown in Figure 4.3. This indicates how the

users will be able to use this infrastructure with the simple interface of GDB. Each of these commands from the user passes through all the blocks described in previous section and gathers its response from hardware. The interactions between the host PC and the hardware for this particular example program is shown in Figure. Reading the hardware state is carried out using several commands and their responses probing all the internal registers, few memory addresses and debug registers.
In cases of traps and interrupt, the details about the type of trap and the interrupt level will not provided by the current implementation. The user has to keep their track by adding breakpoints in the corresponding interrupt and trap handlers.

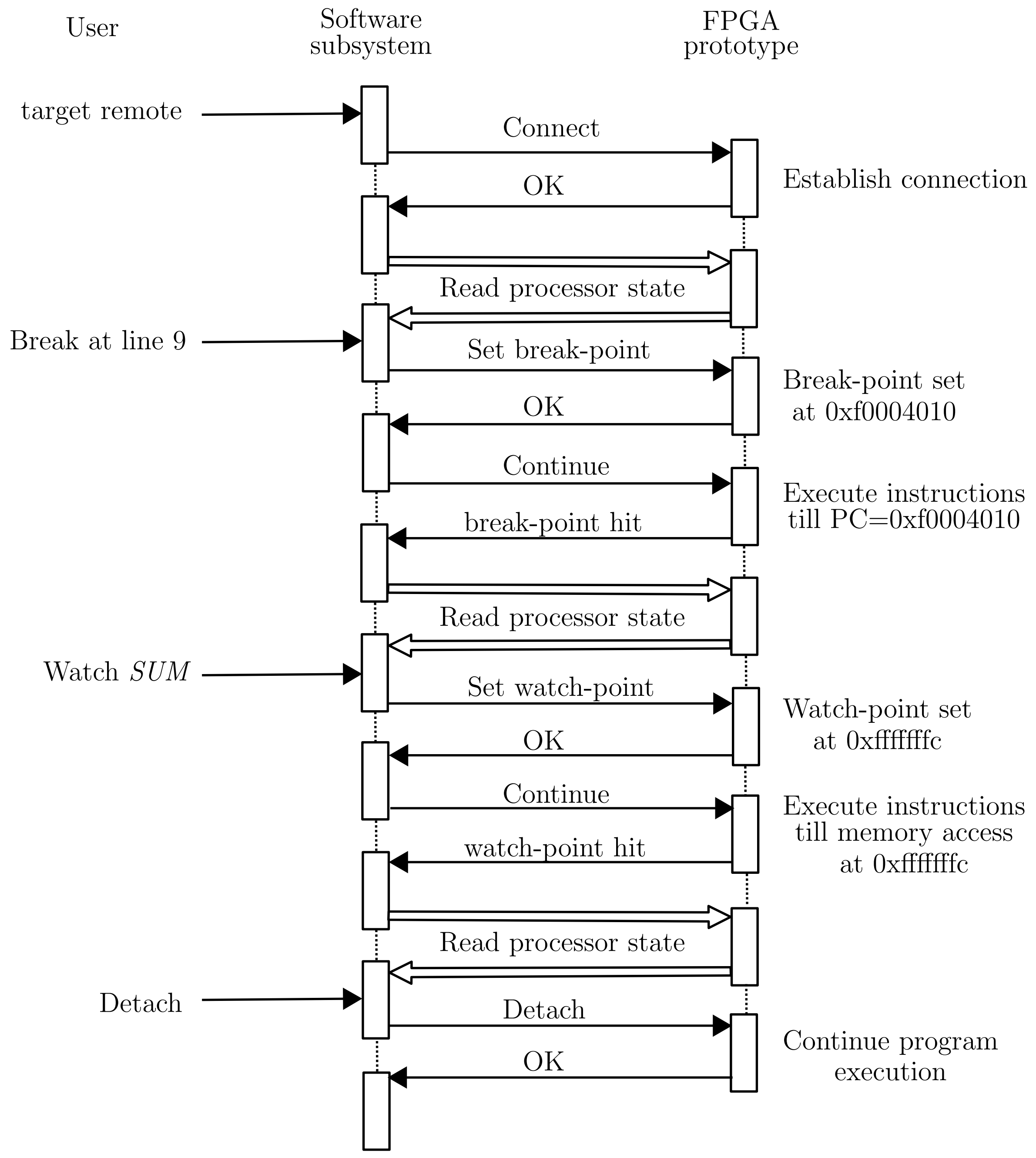


```
(gdb) target remote :8888
Remote debugging using :8888
0x00000000 in ?? ()
(gdb) l
1       //add two numbers and display output
2
3       #include "sparc_stdio.h"
4
5       int  __attribute__((section(".text.main")))  main()
6       {
7               int X, Y, SUM = 0;
8               X = 7;
9               Y = 25;
10              SUM = X+Y;
(gdb) break 9
Breakpoint 1 at 0xf0004010: file adder.c, line 9.
(gdb) c
Continuing.

Breakpoint 1, main () at adder.c:9
9               Y = 25;
(gdb) watch SUM
Hardware watchpoint 2: SUM
(gdb) c
Continuing.
Hardware watchpoint 2: SUM

Old value = 0
New value = 32
0xf0004024 in main () at adder.c:10
10              SUM = X+Y;
(gdb) detach
Detaching from program: /home/titto/AJITwork/AjitRepoV2/proces
/adder/output/adder.elf, Remote target
Ending remote debugging.
(gdb) 
```

(a) GDB debug session



(b) Message communication with hardware

Figure 4.3: Validation on FGPA prototype

Following set of features have been validated on the FPGA prototype and all the previous models with several test programs.

- Stop initially and establish connection with GDB
- Read/modify the processor register and memory contents
- Set/clear break-points/watch-points and stop execution when they are hit
- Continue execution indefinitely or single step as per user command
- Interrupt the processor execution at any instant
- Detach the debugger and allow normal execution to continue

Chapter 5

Conclusion & Future work

The AJIT processor attempts to develop a quintessential embedded microprocessor using high level synthesis approaches. This work provides an in-system debugger for the AJIT processor with powerful debug capabilities and easy-to-use user interface. Users are able to remotely debug the programs running on the AJIT hardware in real time, and can even change the processor state. The complete system is implemented and tested at several levels and is currently being used for Linux kernel development of AJIT processor. The in-system debugger can also be used with the FPGA prototype by connecting it through PCIe Express bus. The hardware modules added as a part of this debugger were heavily used for verification of processor implementations. Finally this debugger will remain as the single point of AJIT processor hardware, to obtain its internal and in-system details and control it externally.

The next main challenge will be to design the communication interface between the host computer and the target hardware. As a part of another project[15] we have developed a JTAG based interface for this purpose, and it will be improved and used for this application. The debugger can also be improved to use multiple communication interfaces to iteract with the hardware and provide flexibility to the end user. By adding support for different RSP interfaces, the users will be able to debug programs from distant computers that are not directly connected with the hardware.

Appendix I

RSP Packets

RSP Command	Description	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
Xadd, $offset$: $data$	Load binary *data* from *add* to $add + offset$. If fails, GDB will do the same using M (utilized in the proposed model)	Empty	1
Madd, $count$: $data$	Write *data* from *add* to $add + count$	OK	3
madd, $count$	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
Znadd	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
znadd	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

Appendix II

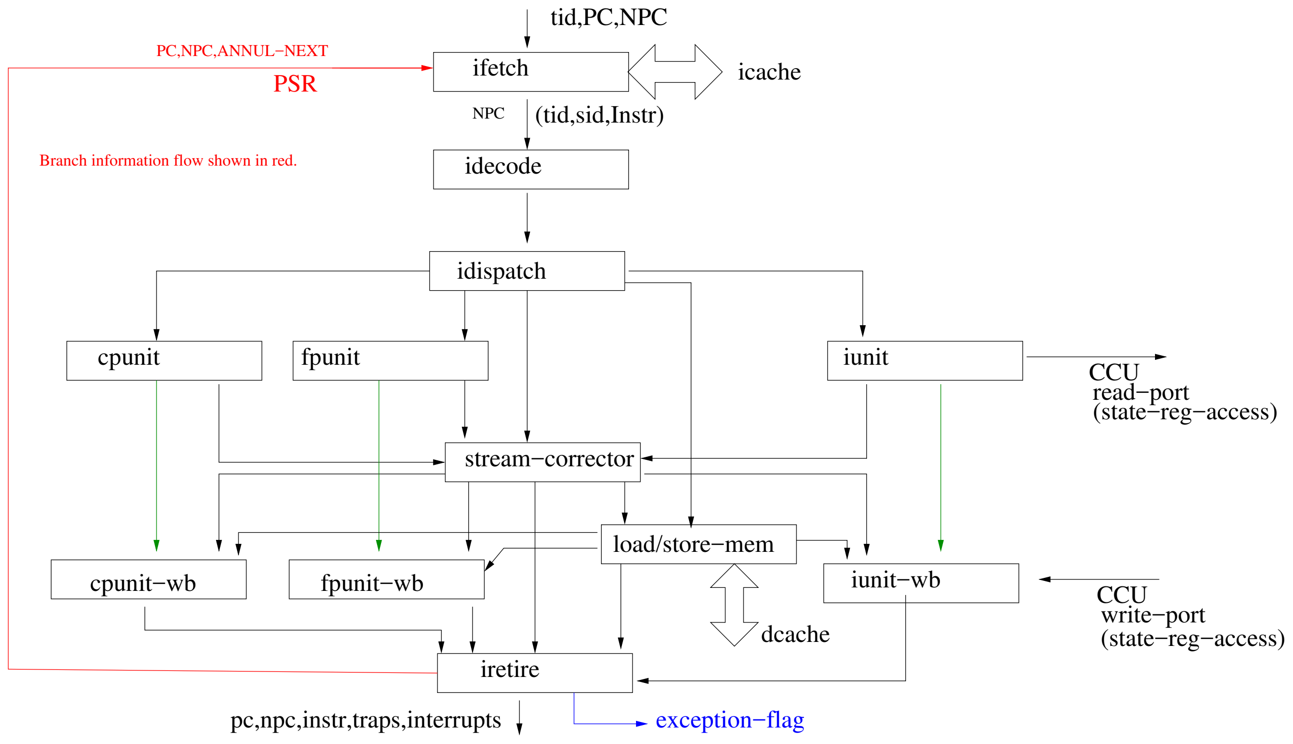
Debug link Packets

Opcode label	**Op-code value**	**No: of words**	**Commnad data Interpretation**
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
WRITE_IUNIT_REG	2	2	
READ_FPUNIT_REG	3	1	15:6 5 4:0 unused fsr reg-id
WRITE_FPUNIT_REG	4	2	
READ_MEM	6	2	15:8 7:0 unused asi
WRITE_MEM	7	3	
SET_BREAK_POINT	8	2	15:4 3:0 unused reg-id
REMOVE_BREAK_POINT	9	2	
SET_WATCH_POINT	10	2	
REMOVE_WATCH_POINT	11	2	
EXECUTE_INSTRUCTION	12	2	
READ_GENERAL_REG	13	2	15:3 2 1 0 unused csr npc pc

CONNECT	14	1	
DETACH	15	1	
CONTINUE	16	1	

Appendix III

Micro-architectural model TEU



References

[1] "IEEE standard for a 32-bit microprocessor architecture," *IEEE Std 1754-1994*, pp. 1–, 1995. DOI: `10.1109/IEEESTD.1995.79519`.

[2] SPARC International Inc., "The SPARC architecture manual v8," p. 295, 1992.

[3] U. T. Electronics, "2014 embedded market study - then, now: What's next ?," 2014 results of UBM Tech's annual comprehensive survey of the embedded systems markets worldwide., 2014.

[4] D. Rath, "Design and implementation of an on-chip debug solution for embedded target systems," Master's thesis, University of Applied Sciences, Augsburg, Department of Computer Science, May 2005.

[5] *GRMON2 user's manual*, English, version 2.0.75, Cobham Gaisler AB, 209 pp. [Online]. Available: `http://www.gaisler.com/doc/grmon2.pdf`.

[6] *Oracle solaris studio dbxtool debugger*, English, Oracle, 4 pp. [Online]. Available: `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/debuggerproductbrief-197597.pdf`.

[7] R. Stallman, R. Pesch, and S. Shebs, *Debugging with gdb*, 10th ed., Free Software Foundation, Franklin Street, Boston, USA, Feb. 2011.

[8] J. Bennett. (Nov. 2008). Howto: GDB remote serial protocol, [Online]. Available: `http://www.embecosm.com/appnotes/ean4/html/`.

[9] M. Sarath, "Implementation and validation of a SPARC-V8 CPU," Master's thesis, Indian Institue of Technology, Bombay, Mumbai, India, Jun. 2014.

[10] P. Aneesh, "Implementation & validation of memory subsystem model for SPARC V8 processor," Master's thesis, Indian Institue of Technology, Bombay, Mumbai, India, Jun. 2014.

[11] F. S. Foundation. (2015). GCC optimizations, [Online]. Available: `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[12] S. D. Sahasrabuddhe, "A competitive pathway from high-level programs to hardware specifications," PhD thesis, Indian Institute of Technology, Bombay, IIT Bombay, Jul. 2009.

[13] M. Desai, *Aa language reference manual*, Department of Electrical Engineering, Indian Institute of Technology, Mumbai, 400076, India, Mar. 2016.

[14] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, 22:1–22:23, Sep. 2015, ISSN: 1936-7406. DOI: 10.1145/2815631. [Online]. Available: http://doi.acm.org/10.1145/2815631.

[15] T. Thomas and M. P. Desai, *Scan chain based testing mechanism for digital systems*, 2nd ed., Wadhwani Electronics Lab, Electrical Engineering Department, IIT Bombay, Jan. 2015.