Debug Support for Retargetable Simulator FSim Using GDB

Hemant Shinde

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

July, 2008

Debug Support for Retargetable Simulator FSim Using GDB

A Thesis Report Submitted

in Partial Fulfillment of the Requirement

for the Degree of

Master of Technology

by

Hemant Shinde

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

July, 2008

Certificate

This is to certify that the work contained in the thesis titled *"Debug Support for Retargetable Simulator FSim Using GDB"*, by *Hemant Shinde*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

July, 2008 ———————————————

(Prof. Dr. Rajat Moona)

Department of Computer Science & Engineering,

Indian Institute of Technology Kanpur

Abstract

Retargetable tool-chains are used by embedded system designers to reduce the cost and design time. The designers can explore several design options prior to choosing one for their system. In order to assist design exploration, the designers need processor-centric tools. The process is greatly simplified by using a single tool-chain development mechanism using processor description in architecture description language (ADL). Generally, a tool-chain is created for a particular processor whose specifications are available in an ADL. As a part of the retargetable tool-chain written for the Sim-nML [37] processor architecture specification language, a functional simulator generator (FSimg)[19] was designed and developed at IIT Kanpur. FSimg is used to generate a functional simulator (FSim) for the processor specified.

In this work, we enhanced the GNU debugger (GDB) to provide debugging support for target applications running on FSim. Such a support needs GDB to be able to change its target properties at run-time. GDB is a debugger and supports several processors as targets for debugging. This support is achieved by an abstraction for the target in GDB. Target specific functionality in GDB is provided by a set of target specific functions which interface to rest of the GDB through a common interface. These functions are compiled into GDB while building it for a specific target.

The GDB supports multiple processors in this way by using compile time reconfigurability. We introduced a dummy target *simnml* to GDB to incorporate run-time reconfigurability. We update this dummy target from Sim-nML processor specification at run-time. In our work we have introduced an fsim interface to achieve run-time reconfigurability. Our work interfaces with the rest of GDB using the *remote-sim* interface of GDB.

Acknowledgments

I would like to express my gratitude and appreciation for my thesis advisor, Prof. Dr. Rajat Moona. His invaluable guidance and support has been major catalyst in the success of this work. His innovative thinking and inspirational advices made this work a truly enjoyable learning experience.

I would like to thank Nitin, Amit G. and Bhupesh for helping me to understand this topic. I would like to thank Mayur, Arun and Vishal for their thoughtful inputs and for helping me whenever I was stuck in my work. Without their help this work wouldn't have been possible. I extend my thanks to Amit HK and Nachiket for the stimulating discussions we had on this topic.

I am thankful to all my M.Tech friends for sharing their knowledge with me and making my stay in IIT Kanpur pleasant and memorable. I will always cherish the moments I spent with them.

I am grateful towards the entire CSE department and IIT Kanpur for providing me with all the facilities and proper environment.

Dedication

I dedicate this work to my parents and my brother.

Contents

List of Figures

Chapter 1

Introduction

Embedded systems span all aspects of modern life and there are many examples of their use. An embedded system is a special-purpose system designed to perform one or a few dedicated functions. Unlike general purpose computers, embedded system's hardware is also designed according to the specific functionality it intends to perform. The traditional design methodology is to first create the hardware components and then write software for them. However, due to ever-increasing design complexity and other constraints, this approach is no longer feasible. Some of the major issues in this approach are as following.

- The hardware design errors become more and more expensive to correct as the design progresses. If the errors are detected at an earlier stage, it will result in substantial cost savings.
- It requires the separation of functionality to be implemented in hardware and software at the beginning of the design itself. This reduces the flexibility of the system and leaves very little scope for further revisions, translating into sub-optimal designs.
- The design of these systems can have many additional constraints, including performance, cost, time-to-market, power, space and reliability requirements.

These problems forced the system designers to pursue new design approaches. In these approaches, the hardware and the software are designed in tandem, leaving designers with multiple choices of hardware software designs to choose from. Designers can choose a design based on various criteria such as cost, power, space etc.

1.1 Hardware-Software co-design

In Hardware-Software co-design the design process of Hardware and Software is integrated. The system description is written without any assumption about the hardware or the software. This system description is often modular and suitable for carrying out system simulations. The estimations of cost and performance can be done by using various modeling and cost estimation techniques. The modular system description is then divided between the hardware and the software through a technique known as hardware-software partitioning. An advantage of this approach is that it allows us to fully explore the design space and to come up with an acceptable solution.

After the division of modules, the design process consists of three steps, hardware and software design, simulation of the design and verification. In the first step, the hardware is typically designed using hardware description languages (HDLs). HDLs are used to write executable hardware specification. They capture the notion of time and concurrency, which are the basic features of hardware. HDLs are used for synthesis and simulation of the hardware. In co-design approach the software may be designed for general purpose processors, for specific purpose processors or for programmable processor cores. The programmable cores allow the designers to configure the processor core according to the specific need of the application.

The hardware and software simulations are carried out and then verified for the requirements of the system. If the requirements are not fulfilled, the division of hardware and software modules may be changed. The modified design is again simulated and verified.

1.2 Retargetable processor modeling tools

Different embedded systems have their own requirements. Often general purpose CPUs are either insufficient or carry an extra baggage of hardware causing extra power being consumed. We can either make an entirely new processor core optimized for the application requirements (Application Specific Processors) or enhance an already existing one. In the second case we have several alternatives to choose from, such as Tensilica [9] and Xilinx Vertex-5 [11]. However in these cases the designer requires a new set of software tools for each alternative he may want to use. Development of these processor-centric tools takes a lot of time and effort. This problem can be solved by automation of the tool-set generation process.

The tool-set generation process can be automated using the architectural models of processors. Each tool then takes the processor model description and configures itself for the new processor. The specification language shall have the syntax and semantics to support the features, necessary in the processor description for tool-set generation. The language should be able to specify static as well as run-time behaviors of the processor.

1.3 Overview of languages for retargetable tools

The processor models used by the retargetable tools are written in some processor specification languages. They describe hardware details in such a way that the retargetable tools can be generated from them, without much user intervention. The operation of same hardware can be described by several levels of abstraction. The level of abstraction decides the complexity of the description. The three levels, in the increasing order of abstractions are switching circuit level, register transfer level (RTL) and processor instruction set level. The hardware specification languages support one or more levels of abstraction.

Hardware models of processor at circuit or register transfer level are generally described using hardware description languages (HDLs). Examples of these languages include Computer Design Language (CDL) [28], A Hardware Programming Language (AHPL) [33], Instruction Set Processor Specifications (ISPS) [17], Verilog [6] and VHDL [5]. Except ISPS, all the other HDLs describe hardware either at register transfer level or at switching circuit level. The level of detail required, makes it difficult to use this technique for complex systems. Further these models are extremely slow for the purpose of simulation.

Instruction Set Processor Language (ISPL) [16], developed at CMU, is the first to use the abstraction at instruction level to describe a processor model. The constructs introduced by this language, are still widely used by other specification languages. The language is used for generating processor simulators.

SystemC [7] is used for system description or behavior modeling, instead of hardware description. It has similar semantics as HDLs but has more syntax overhead. It provides a set of library routines and macros implemented in C++. SystemC is better used for system modeling than to write a processor description.

Architecture description languages (ADLs) are higher level languages, designed specifically to model processor architectures. They provide sufficient abstraction and are suitable for making retargetable tools. Some of the ADLs and frameworks are described below.

The nML [24] language is developed at TU Berlin. It uses the instruction level abstraction to describe the processor. It however lacks constructs to support resource usage and timing mechanisms. Sim-nML [37] language developed at IIT Kanpur is an extension of nML, which adds these features.

ISDL (Instruction Set Description Language) [27] is developed at MIT. It is a machine description language and describes the processor at instruction set level. It is mainly targeted towards VLIW architectures. ISDL has been used to generate assembler and code generator generator.

MDES (machine description language) [13] is developed at the University of Illinois in collaboration with HP. It is used in Trimaran [10] compiler infrastructure. It allows the user to develop a machine description for the HPL-PD family of processors in a high-level language, which is then translated into a low-level representation, for efficient use by the compiler. It provides limited retargetability.

MIMOLA [32] language is used for structural description of the processor. The description consists of net-lists which makes it at a lower level than the instruction set. Hence, the descriptions in this language are difficult to write and modify. However, these descriptions are used for generating retargetable compilers.

LISA processor specification language [36] was developed at Aachen University, Germany. It supports behavioral, timing as well as instruction set model of the processor. It covers architecture and pipeline details of modern DSP processors. It has been used to generate retargetable tools such as assemblers and simulators.

RADL [40] is an extension to LISA. It provides multiple pipeline support to LISA language.

EXPRESSION [29] language was developed at UC Irvine. It provides behavioral as well as structural modeling support for architecture descriptions. It is used to generate compiler and retargetable simulator. The language provides syntax for structural information, which is used to generate reservation tables, required by the compiler.

CoWare provides a design environment called CoWare Processor Designer [2], which supports the specification of heterogeneous systems and its systematic refinement for heterogeneous implementation. Hierarchical data models of CoWare allow encapsulation of existing specification languages, their simulators and design environments and forms the basis for system simulation and implementation.

1.4 Related work

The retargetable languages and frameworks have been used to generate retargetable simulators.

ReXSim [34] developed at UC Irvine, is a retargetable processor instruction set simulator. It uses the EXPRESSION ADL as its processor specification language. ReXSim carries out compiled simulations and gives superior performance than the interpretive simulators.

A fast and retargetable simulation technique is presented in DATE [26]. It improves traditional static compiled simulation by aggressive utilization of the host machine resources. Such utilization is achieved by a low level code generation interface specialized for ISA simulation. This approach requires C descriptions that are based on the internal implementation details of the simulator.

The simulator generated from FACILE [38] descriptions uses fast forwarding technique to speed up the execution. FlexWare simulator [35] uses VHDL for its description and needs low level hardware details to be specified.

SimpleScalar [20] is a popular interpretive simulator. It supports a number of contemporary architectures, but is not retargetable.

1.5 Introduction to Sim-nML

Sim-nML [37] is a language for describing arbitrary processor architectures. It provides processor description at an abstraction level of the instruction set and hides all hardware level implementation specific details. It can be used to describe processor architecture for various processor-centric tools, such as instruction-set simulator, assembler, disassembler, compiler back-end, profiler etc., in a retargetable manner. Sim-nML has been used as a specification language for generation of various processor modeling tools.

1.5.1 Previous work with Sim-nML

Sim-nML was designed as an extension to the nML language [24]. The nML language was not designed for modeling the execution time behavior of processors. It had no constructions to define the timings of operations. Sim-nML addressed this issue by adding a resource usage model [37]. Using resource usage model it is possible to extract the timing behavior of instruction execution incorporating details of pipelines, superscalar and VLIW architectures. Sim-nML has been used as a specification language for generation of various processor modeling tools, some of which are given below.

- Disassembler: A processor independent symbolic disassembler [31] was designed. To avoid processing of Sim-nML descriptions, an intermediate representation (IR) of processor description in the form of fixed size tables was also introduced. The disassembler would take the IR as an input in addition to the processor binary code. The IRs were generated from Sim-nML descriptions using Sim-nML parsers.

- Functional Simulator: A retargetable functional simulator (FSimg) [22] was designed and extensive simulations were carried out using a PowerPC 603 processor model.

- Compiler Back-End: This tool [18] reads a Sim-nML description in intermediate form and generates a partially complete GCC machine description for the processor. The tool was tested by retargeting the GCC to Sparc processor [12].

- Cache Simulator: A cache simulator [15] was developed to provide a basis for benchmarking various cache policies for a given processor and corresponding binary application.

- Memory Profiler for Functional Simulator: A memory profiler [21] was developed, which generates profiling logs for the processes running on the simulator. These logs are then analyzed for various purposes.

- Disassembler Support for Sim-nML targets in GDB: GDB is used to debug the programs on Functional Simulator targets. A Sim-nML description has assembly language syntax for each instruction. Target disassembler of GDB [23] was configured and modified to support the target specified by Sim-nML processor description.

1.6 Overview of this work

The FSimg [19] was developed as part of the thesis work of Surendra Kumar Bishnoi at IIT Kanpur. FSimg can be used to create retargetable functional simulator (FSim). In our work, we have used the GNU Debugger (GDB) [3] to provide generic debugging environment for the target applications of FSim. We have also implemented the target side of GDB in the simulator.

In our work, we have used the remote-sim interface of GDB with functional Simulator (FSim). GDB is a retargetable debugger. It can be built for any particular target. However once GDB is built, it can not be used for any other target. Hence, the GDB could not be used for retargetable simulators. We have developed an interface, such that the target architecture of the GDB can be updated from the processor specification, at run-time.

FSim is a retargetable functional simulator. It is generated by FSimg [19] from the processor specification. GDB interacts with the simulator for handling of its commands. GDB has several commands, which can be used to watch and update the values of expressions, examine memory and stack, to suspend or to continue flow of execution and to examine the call stack. These commands are implemented in GDB at the target side interface. In our work, we implement these commands in the FSim simulator to make GDB runtime retargetable.

1.6.1 Motivation

An instruction set simulator can be used to analyze the execution behavior of an application on a target processor. The highest level of abstraction provided by this simulator is of machine instruction level. These simulators are of two types, interpretive simulators and compiled simulators. Generally, the debug support is provided to interpretive simulators by creating a new debugging environment. However, these debuggers can only debug the programs at assembly instructions level. The GNU Debugger (GDB) [3] provides an interface for processor architecture simulators, using which one can debug the programs written in a higher level language on the simulated processor. The problem here is that, this approach supports only the simulators of existing GDB targets. In order to provide the debugger support for a customized processor, one needs to write a new machine description for GDB since GDB does not provide support for retargetable simulators.

Sim-nML is used for describing processor architectures. The aim of the language is to minimize the efforts needed for developing the software tool-set generation by using automation. The functional simulator generator (FSimg) [19] generates a retargetable simulator FSim from a processor description written in Sim-nML language. The FSim is used to simulate an application, which may be written in a higher level language. The addition of debug support to FSim will provide an ability to debug the applications written for customized target processor. GDB is one of the most widely used debugger with a simulator interface. We therefore use the GDB for providing the debugging support to the application simulated by FSim.

1.6.2 Organization of report

Rest of the thesis is organized as follows. In chapter 2, we describe the existing Functional Simulator and discuss its retargetability issues in detail. In chapter 3, we give the GDB overview in the context of our work. In chapter 4, we look into interface between the GDB and the FSim. We discuss how a target is added to the GDB, and how the GDB interfaces with a simulator. We also discuss the interface developed for updating the machine architecture in GDB using the corresponding Sim-nML description. In chapter 5, we discuss the implementation of the GDB commands in the simulator. In chapter 6, we conclude this work and provide results and future work.

Chapter 2

Retargetable Functional Simulator Generator (FSimg)

The Functional Simulator Generator (FSimg) [19], based on the Sim-nML descriptions, was designed and developed at IITK in an earlier M.Tech. thesis by Surendra Kumar Bishnoi. It generates functional simulator (FSim) from the Sim-nML description and the target binary executable.

2.1 Overview of FSim

FSim is a functional simulator. It simulates the program for a target processor at the instruction level. At runtime, the processor state is maintained by simulating the registers, flags and memory. Execution of every instruction modifies the target processor state in the simulator to mimic the actual processor state. Micro-architecture details of the processor are ignored, e.g. pipeline behavior of the processor is not simulated. The functional simulation is useful for verifying the correctness of programs written for new processor designs. It is also used to generate execution profile of a program, for the target processor, which is later utilized by various profile visualization tools.

2.2 Generation of FSim from a processor description

FSim is a compiled simulator. The simulator generation process is carried out in three stages. In the first stage, a processor model is generated using the target Sim-nML description. In the following step, the target binary program is decoded using the processor model. In stage three, the functional simulator is generated.

2.2.1 FSim generation stage-I

In stage-I of functional simulator generation, a processor model specified in Sim-nML is converted to simulator programs in C language. This process is carried out by using Sim-nML compiler. Every Sim-nML construct is converted into one or more C constructions. The details of Sim-nML to C conversion are given in [39].

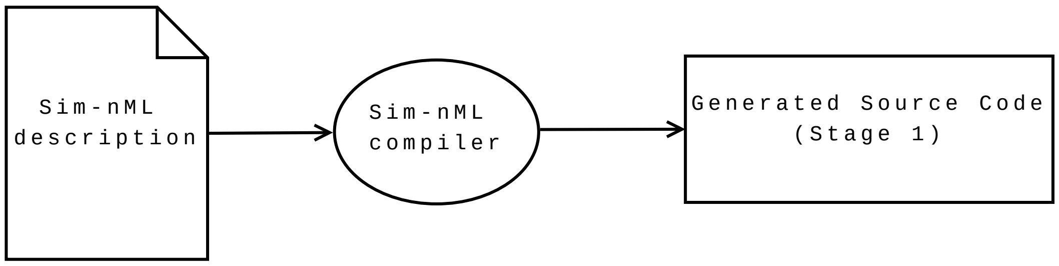


Figure 2.1: Stage-I of FSim generation

The stage-I of the FSim generation converts the Sim-nML processor specification to the following files used by subsequent stages of FSim generation.

1. type_decl.h: Every processor has different word size and address size. Every field in the instruction opcode has a different size. Hence, while creating a processor specification, certain new data types are defined. These help in better readability of the specification, after the conversion to C. These data types are defined in this file.

2. var_decl.h: The processor contains various storage objects as defined in the Sim-nML description. These include registers, temporary variables and the memory. The storage objects are converted to variables in the C program. All such variables are defined in this file.

3. inst_tbl.h: Every instruction has an opcode and some operands. The image attribute of the instruction in the Sim-nML description defines the operand and the opcode. This image attribute information is converted to a table. Each entry in the table has a bitmap and a binary mask to recognize the instruction. It also has similar masks for extracting parameters from the instruction byte stream of the program. The definition of this table is extracted for Sim-nML description and generated in this file.

4. func_def.h and inst_func.c: The semantics of instructions defined in the processor specification are converted to functions in C program for simulation purposes. The prototypes of these functions are declared in func_def.h and the function definitions are generated in inst_func.c.

2.2.2 FSim generation stage-II

In this stage, the binary executable, compiled for target processor[1] is decoded and converted to an ordered list of instructions. The opcodes of the instructions, number of operands and the instruction formats changes with the processor description. It is because of this reason that stage-II depends on the output of stage-I.

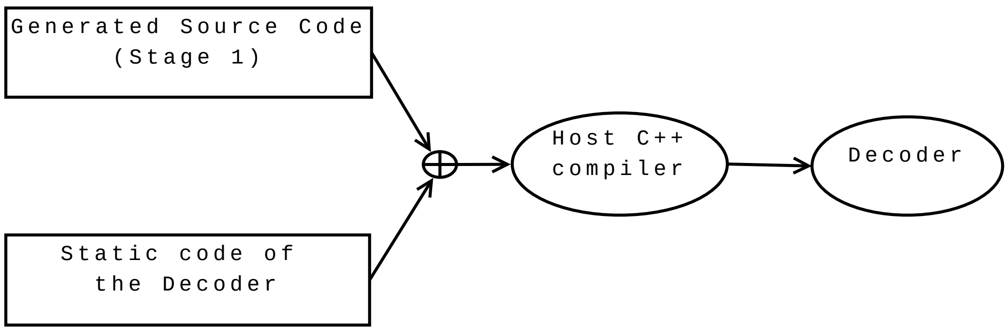


Figure 2.2: Generating a decoder for target processor

As shown in figure 2.2, a decoder is generated by combining the static code of decoder and the generated code from stage 1 of functional simulator generation. The decoder is then used for decoding of the binary executable file compiled for target

[1]Target processor: The processor, for which we write processor specification in Sim-nML.

architecture.

The executable binary file used for the decoding is supported only in the ELF (Executable and Linkable Format) [42]. An ELF file has machine instructions, linker information and loader information so that a process can be created and control can be passed to it. An executable ELF file contains ELF header, program header, section headers and the data referred to by the entries in the program and section headers. The ELF header has a field that defines the target architecture of the executable file. It also gives the location of the program header table and section header table, along with the associated number and size of entries for each table. Lastly, the ELF Header contains the location of the first executable instruction.

A process image consists of verious segments which include stack, data, bss etc. The program header table contains entries describing these segments. Each entry provides the type, file offset, physical address, virtual address, file size, memory image size, and alignment for a segment in the program.

The section header table is an array of structures. Each entry of the table correlates to a section in the file. The entry provides the name, type, memory image starting address (if loadable), file offset, the section's size in bytes, alignment, and how the information in the section should be interpreted.

An operating system creates a process from the executable file, initializes it and passes the control to the first instruction of program. However, in our case simulator has to perform these operations on its own by simulating the OS behavior. Thus along with decoding the instructions, decoder also has to create the initial process image. As shown in figure 2.3 the data from the ELF executable file is divided accordingly by the decoder. In stage-II certain C files are generated for the decoded instructions and image files are generated corresponding to the initial value of the memory for the process. The image files are in Intel hex format [30] and serve as initial image of the process in memory. A number of image files are generated, one for each memory space supprted by the processor. The simulator has to just load the image file in memory and start the execution.

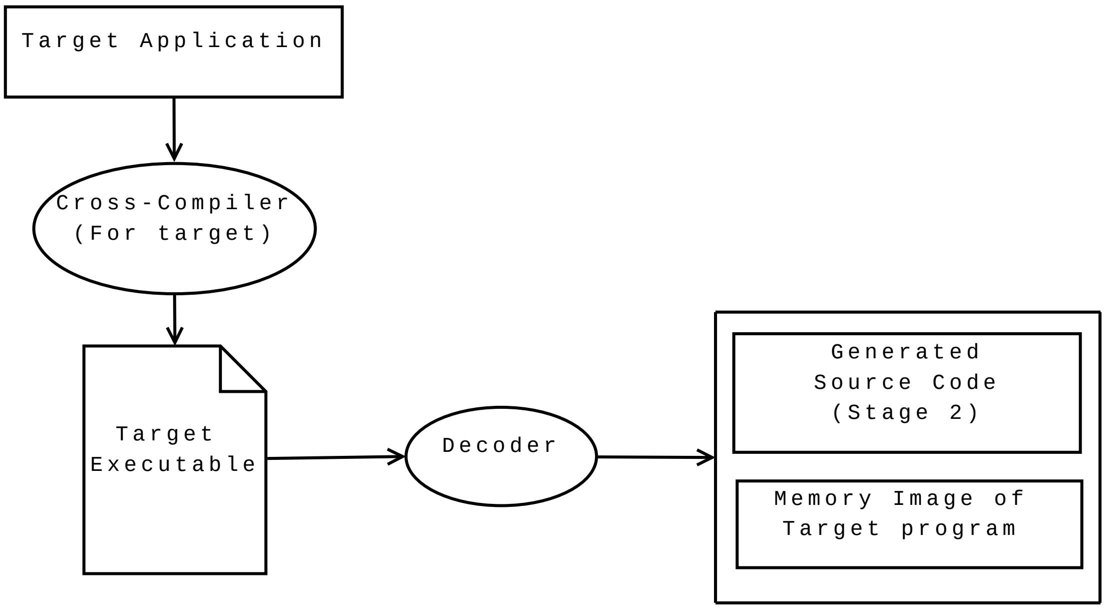

Target Application
Cross-Compiler (For target)
Target Executable
Decoder
Generated Source Code (Stage 2)
Memory Image of Target program

Figure 2.3: Decoding the target binary (Stage-II)

Following files are generated as output of stage-II.

1. defs.h: This file contains the initial values of registers of the target processor. These include stack pointer, program counter and other CPU registers. To start the target program in simulator, processor state is initialized with these values. The values of these registers are taken from the ELF binary file.

2. exec_tbl.h: In stage-I of FSim generation, each assembly instruction specified in the processor specification is converted to a subroutine definition in C language. When the instructions from the executable file are decoded, each instruction decodes to a pointer to the subroutine (generated in stage-I) corresponding to the instruction. These pointers and operands of the each instruction are stored in a table. The table is defined in this file.

3. mem.img: This file is a memory image file and is in the Intel hex format [30]. It contains process's initial state of memory when the program is loaded. Simulator reads this file and generates the program memory state at the time of execution. There may be many such files generated, one for each address space supported by the processor. All such files are then loaded by the simulator for

each memory space initialization. The name of the file is the same as the name of the address space while the extension is ".img".

2.2.3 FSim generation stage-III

As shown in figure 2.4, in stage-3 we compile together the outputs of stage-I and stage-II of FSim generation process, along with the static code to generate FSim. The FSim runs the decoded program.

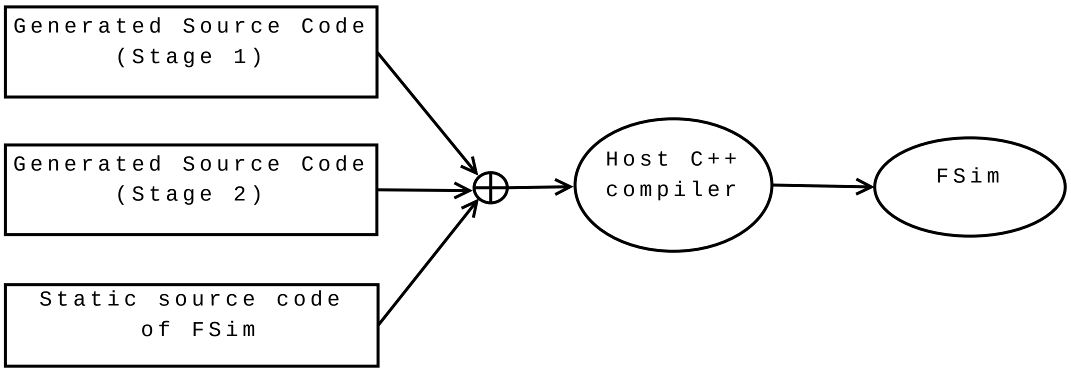

Generated Source Code (Stage 1)
Generated Source Code (Stage 2)
Static source code of FSim
Host C++ compiler
FSim

Figure 2.4: Generation of FSim (Stage-III)

2.3 FSimg organization in retargetable perspective

FSimg generates FSim in three stages. It does this by using the tools to generate the source code of FSim such as Sim-nML compiler and decoder. The generation of FSim depends on processor architecture and the OS of the target machine. The code for FSimg is organized in four parts- processor independent, processor dependent, OS dependent, and processor as well as OS dependent. A processor independent part of FSimg does not change with the change in target processor specification or the OS. A processor dependent part needs to be generated/modified for every processor specification and is usually done in stage-I of FSim generation. The OS dependent part of the FSimg needs to be updated with change in the OS. To make the FSim retargetable we generate the processor dependent information from the processor description. The OS dependency can be of the two types,

- The dependency on host OS, where FSim is generated.

- The dependency on target OS, for which the target processor executable is created.

Consider a scenario where we want to simulate MIPS processor. We write processor specification for MIPS32. We install cross-compiler for MIPS32 on i386-linux machine and cross compile programs on this machine for MIPS32 processor. Now we want to create FSim for the cross compiled MIPS32 programs intended to run on MIPS32-linux machine, on i386-linux machine.

In this scenario, host OS is i386-linux and the target OS is MIPS32-linux. The target processor is MIPS32.

2.3.1 Processor independent part of FSimg

Processor independent parts of FSimg is compiled only once for the host machine. This includes the Sim-nML compiler. The Sim-nML compiler is part of stage-I. It takes the processor description as an argument, processes it and translates it to equivalent 'C' code.

2.3.2 Processor dependent part of FSimg

A decoder is a program which reads from the binary executable file and generates the C code along with memory image. Every instruction is mapped to a C function generated in stage-1. Therefore, this mapping is different for each processor specification. For our example, this part will depend on the MIPS32 processor description and the generated decoder will decode the instructions of MIPS32 machine.

2.3.3 OS dependent part of FSimg

ELF library provides routines to read the ELF file. Here the ELF file is cross-compiled for target processor and target operating system. However the ELF file format only needs to know whether the operating system of the target is 32-bit operating system or 64-bit operating system. In our example, this part will depend on the target OS, i.e. MIPS32 Linux distribution and the ELF library will be configured for 32-bit operating system.

2.3.4 Processor and OS dependent part of FSimg

In stage-III the output of previous two stages along with static code of stage-III is used to generate the FSim. FSim mimics target processor state for which it should know the target processor architecture. The target program is compiled for the target processor architecture and target OS. The compiled program uses system calls of target operating system. During simulation, these system calls are translated to equivalent system calls provided by the native operating system. In an example, stat64 system call in MIPS32-linux is mapped to fstat system call of i386-linux machine. Therefore, the stage-III of FSimg is dependent on the target architecture, the target OS and the host OS.

2.4 FSim execution

At runtime, FSim only requires the memory image file of the target program, generated by the decoder. Simulator creates and initializes the memory of the target program using the *mem.img* file. The register values are initialized and the execution of the target program is started. An instruction is fetched from the instruction table[2] and executed. In order to do so an index in the instruction table is derived from the current PC register value. If the index is outside the range of the instruction table the simulation is aborted. During the stack operations, there may also be a stack overflow in which case also the simulation is aborted.

Flow of execution of FSim is depicted in figure 2.5.

[2]Instruction table is generated by decoder, in stage-II of FSimg.

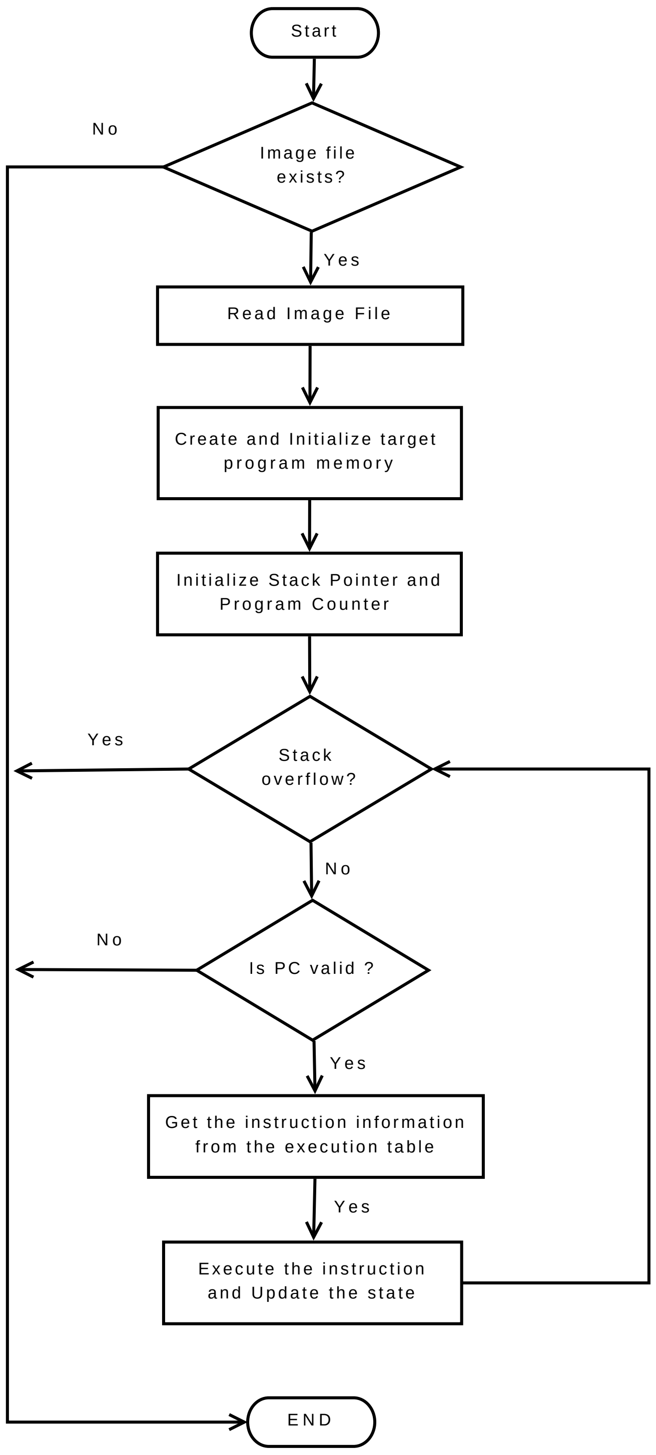

Start
Image file exists?
No
Yes
Read Image File
Create and Initialize target program memory
Initialize Stack Pointer and Program Counter
Yes
Stack overflow?
No
No
Is PC valid ?
Yes
Get the instruction information from the execution table
Yes
Execute the instruction and Update the state
END

Figure 2.5: Flow chart of FSim execution

2.5 Implementation of retargetability in FSim

A processor specification only describes the architecture of the processor. However, the simulator has to perform the functionality of an operating system as well as that of the processor. Therefore we need to distinguish between the target operating system functionality and the processor functionality.

To run a program on simulator, it must have following knowledge about the target processor.

1. Endianity: The endianity is the byte ordering used to represent the multi-byte data. It defines the order in which the individual bytes of data are stored in the memory. The target program memory is created and maintained by the simulator. A system call made by the target process which is simulated on the host machine, may cause a data exchange between the host OS and the target program's memory. If there is an endianity mismatch between the target and the host processor, the simulator handles the situation by transforming the data to the correct endianity. We add the endianity information of the target processor in its Sim-nML description.

2. Program counter (PC) register: The program counter register has the address of the instruction to be executed. When the simulator starts the execution of the target program, this register is initialized with the starting address of the program. However, every processor has different name for the PC register. As the FSim recognizes a register only by its name, it must be aware of the PC register name of the target architecture.

3. Stack pointer (SP) register: The stack pointer register points to the program stack. Function parameters are usually passed through the processor stack in the executable code for that processor. A frame containing the return address of the function, local variables and the previous SP register value, is also created on the processor stack. As FSim simulates the process, it also needs to simulate the stack. The initial value of the SP register varies with each target program,

and is specified in the ELF header of an executable. Similar to the PC register, the name of the register used as stack pointer is different for each processor.

4. Function calling conventions: Different platforms use different function calling conventions. This affects the initialization of the target program in the FSim, as the first function to execute might take some arguments depending upon the OS environments. These arguments must be passed by the simulator to the target process memory for which it must use the calling conventions of the target OS for which the target binary was generated. In addition, the target process makes various system calls of the target OS. We therefore need an OS emulator that must handle such system calls. The system calls also use different calling conventions. The FSimg provides a support to pass on the system calls of the target OS to host-specific functions and therefore needs to be aware of host and target calling conventions.

The registers defined in a processor specification are converted into variables in the equivalent C code. These variables are accessed by the later stages of the simulator. Therefore the FSimg can not know the variable names prior to the compilation of the Sim-nML specification. However, in the later stages, we need to access these variables in the static code of FSim. Therefore, instead of using register variable names, we use SP and PC identifiers for stack pointer and program counter register respectively. We rename these identifiers using compiler directives to the actual register names for the particular processor. The register names of PC and SP register are specified in the Sim-nML description.

A C function call can be made from Sim-nML specification using canonical function mechanism. This facility is used to provide a system call interface for the processor. In a new processor description, which needs a system call interface, a system call handler is added in the static code of FSimg. This function translates the target system calls to their equivalent host counterparts. The system call interface is used to provide OS-processor communication. It is used for parameter passing and to initialize thread data pointer, requierd by certain target OSes.

Chapter 3

GDB Overview

A debugger is an interactive software, that provides fine-grained controls over the execution of a program. It can examine and change the state of the process being executed. GNU Debugger (GDB) [3] is a free, open source software. It is one of the most commonly used source language debuggers. GDB has several commands, which can be used to watch and update the values of expressions, examine memory and stack, suspend or continue flow of execution and examine the call stack. These commands are entered through a command line interface.

3.1 GDB internals

As shown in figure 3.1, GDB can be divided into four different functionalities, user interface, symbol handling, target architecture support and core gdb handling.

User interface includes taking the commands from the user and presentation of debugging information to the user. It incorporates various routines to display the debugging information, to read the commands from user and to invoke the handler of the command given by the user. A command handler implements the debugging algorithm by dividing the task associated with the command into smaller subtasks.

Symbol handling functionality of GDB consists of object file readers, symbol table management, expression handling, source language support, source display and other activities that involve symbolic data. Binary File Descriptor (BFD) library which

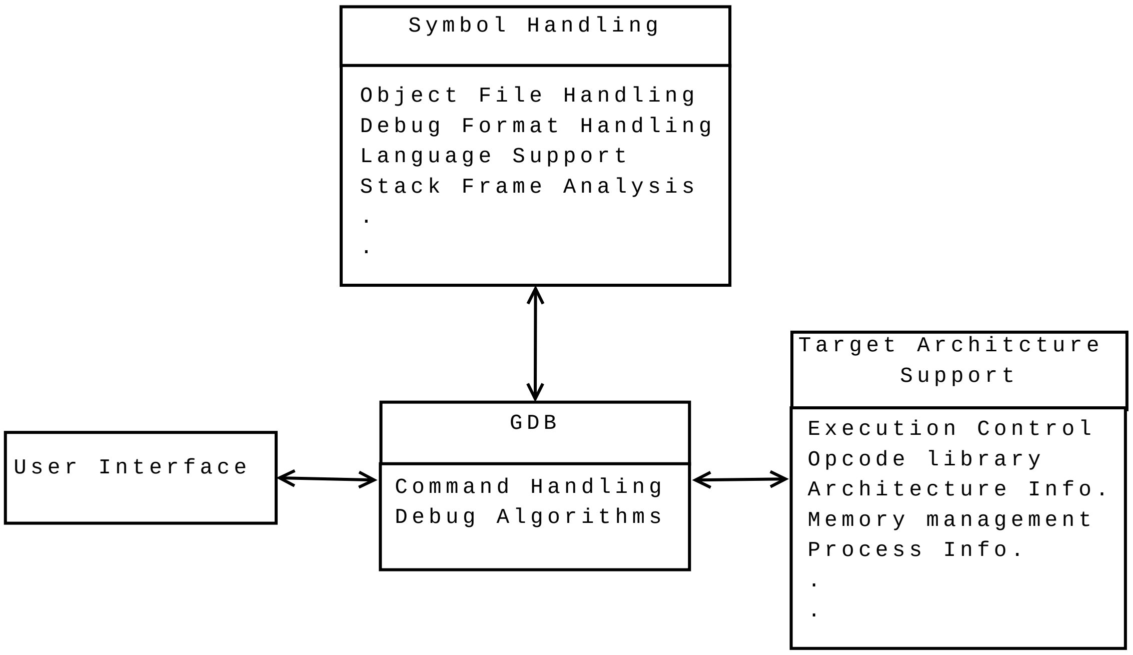


Figure 3.1: Basic GDB architecture

consists of facilities for reading various executable and object file formats (ELF, COFF, a.out etc.) is a core component of this subsystem. The symbol handling functionality also supports various debug formats (Stabs, DWARF2 etc.), which are used to express debug information.

The target architecture support consists of execution control, stack frame analysis, disassembly, opcode library, target specific routines for breakpoint, handling registers etc. This is mainly used to access and manipulate target data. GDB assumes the target machine includes a bank of registers and a block of memory. The processor registers are assumed to form an ordered list. Every register is identified and accessed by its register number, which is its index in the list. The order of all registers is fixed; it is same as that of GCC for the particular target.

The modular architecture of GDB allows GDB to be ported to large number of target architectures.

GDB also provides support for interfacing with simulators (figure 3.2) through a simulator interface which is a very helpful feature for development of debuggers for the embedded systems. The simulator provides the mechanism for target architecture support.

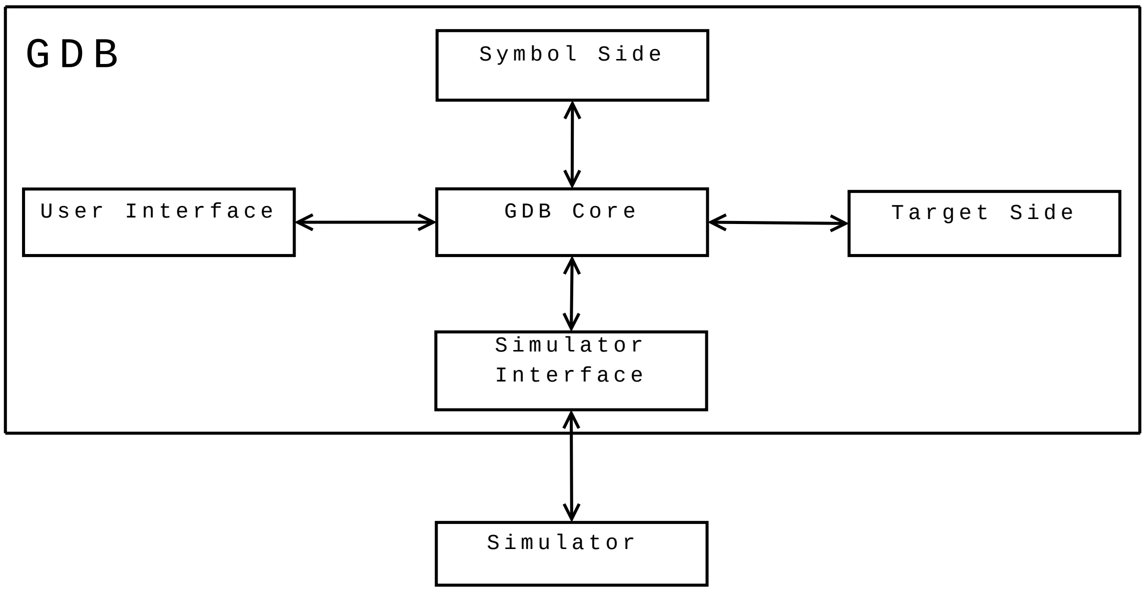


Figure 3.2: GDB architecture with simulator

3.2 Initialization of GDB

For a debugger, the host machine is where the debugger resides, and the target machine is for which the binary executables of the programs to be debugged are generated. In a typical scenario, a debugger supports single target architecture. A retargetable debugger is designed such that, it is relatively easy to modify the debugger to support different target CPU architectures. These modifications are typically related to the architecture dependent parameters of the target support functionality. A retargetable debugger can be built for any of the supported target processor architectures. GDB is a retargetable debugger and the target processor is selected during the configuration of GDB.

In retargetable tools, the retargetability is usually provided by maintaining the architecture dependent information in a set of variables. Some of the architecture dependent information includes endianity of the processor, number of bits in the address and data word of the processor, number of registers of the processor etc. In GDB, the architecture specific information is provided through a source file named

<arch>-tdep.c. Here *<arch>* is the name of the processor. A processor initialization routine in this file initializes the architecture specific information of GDB with the information of the selected target architecture.

The *<arch>-tdep.c* file describes the basic layout of target processor architecture. It contains miscellaneous code required for various operations on the target machine. These operations include breakpoint handling mechanisms, stack frame handling and register information retrieval mechanisms.

3.2.1 Target specific initialization for handling breakpoints

The implementation of breakpoint mechanism is processor dependent and is part of the target side of GDB. Calculation of breakpoint address, insertion of breakpoint, resuming the execution flow after the breakpoint and removal of breakpoint are some of the major functionalities in breakpoint mechanism. These are implemented by the target specific routines.

If a breakpoint is asked to be set on a function, it means the breakpoint should be set at the first instruction after the function prologue. The size of function prologue varies with the ABI and hence, the routine to skip function prologue while setting the breakpoint is implemented by target side. This routine gives the actual instruction address, where the execution will break due to the breakpoint.

An interrupt or a trap instruction is inserted at the breakpoint address to set the breakpoint. The opcode and size of the instruction to be inserted in place of actual instruction is different for different targets. This information is provided by a target specific routine. This routine is used during the insertion of a breakpoint.

The original instruction at breakpoint address is executed when the program resumes its execution after the occurrence of a breakpoint. This can be handled in multiple ways depending on the facilities provided by the hardware. If the hardware provides single stepping support, GDB has to insert the original instruction back to its original location, decrement the program counter to the breakpoint address, execute a single step and insert the breakpoint again. If the hardware does not support the single stepping, the original instruction at the breakpoint address is copied to an executable page in the program memory, and the instruction is followed by the code to return to the next instruction after the breakpoint. In this scenario, after the breakpoint the execution resumes at the memory location where the original instruction from the breakpoint address is copied. The routines to implement any of the above mechanism are dependent on processor architecture and hence are target specific.

Even though the implementation of above routines varies with the processor architecture, their functionalities do not change. GDB declares function pointers for each of these functionalities and their implementation is provided by the target architecture. Each routine is then registered with GDB for its functionality.

3.2.2 Target specific initialization for handling call stack

Each time a program performs a function call, certain information about the call is gathered. This information includes the location of the function call in the program, arguments of the function call and the local variables of the function being called. This information is saved in a block of memory called frame. The frames are usually allocated on the stack known as call stack.

A stack frame has data associated with a call to any function. When the program is started, the call stack has only one frame, corresponding to the function *main*. This frame is called the *initial* frame or the *outermost* frame. Each time a function is called a new frame is created. Each time a function returns, the frame for that invocation is destroyed. The frame for the function in execution (on stack top) is known as the *innermost* frame. The frames are linked together by use of frame pointers, the memory address of a frame. Each time a new frame is created, address of previous frame is stored in the new frame. Using the *backtrace* command of GDB a user can examine all the frames on the call stack. However, to retrieve the call stack information the GDB has to trace back from the *innermost* frame. The call stack structure is specific to the target architecture. A *sniffer* routine is used to retrieve previous frame information from a given frame. The process of retrieving previous stack frame from the given frame pointer is known as unwinding the stack. GDB provides *sniffer* routines for each debug format. In addition to those routines target specific unwinding routines are also provided. GDB maintains the list of unwinding routines specific to the target architecture. These sniffer routines are registered during the initialization of the target architecture.

A frame has an associated frame pointer. The frame data is accessed by using the frame pointer. The absolute address of a variable stored on the stack frame is calculated by adding its offset and the frame pointer. The offset of each local variable from the frame pointer is given in the debug information of the program. The offsets of function parameters, return address and the non-volatile registers from the frame pointer are fixed for a particular application binary interface (ABI). As the ABI is target procesor specific, these offsets are also target processor specific. This information is used while tracing the execution path of the program, i.e. to trace the calls made from the beginning to reach the current function.

3.2.3 Target specific initialization for handling CPU registers

Routines to provide name of a register, type of a register and to get the value stored in the register are specific to a target. In addtion a separate routine is provided to get the value of the program counter register. All these routines are identified during the initialization of GDB with a specific target.

After specifying these routines and initializing the architecture information, the GDB is ready to accept a command.

3.3 GDB command processing

GDB has a command processing engine which maintains the list of commands that are supported. During initialization of GDB, the supported commands are added to this list. Each command has its own syntax and associated command handler. Whenever an input command matches with one of the commands in the list, the corresponding command handler is invoked, with the rest of the input command passed as the parameter. Command handler parses the parameters and performs the actions specified by the command. The output from the command handler is printed. For example, when a command such as the one to print the value of a variable is given, GDB makes a call to read data from the memory location corresponding to the address of the variable. In this example size and address of the variable is obtained from the debug information.

The values stored in the memory are in the endianity of the target architecture. However the code to print them runs on the host processor. Hence these values are printed according to the host endianity. The conversion of endianity from the target to the host is carried out by the command handler. GDB assumes that it will get all the data in the host endianity. The command handlers therefore have to implement all target-specific functionality such as reading of the CPU registers, memory locations and parsing of CPU specific expressions.

3.4 Debug information

The debugger needs the debug information for symbolic source level debugging. GDB supports a variety of debug information formats such as stabs, COFF, DWARF-1, DWARF-2 etc. The debug information formats are processor independent. However, they are used to provide information about the target processor and application binary interface.

Following debug information about the target architecture and application binary interface (ABI) is required by GDB and is provided by the debug format.

1. When a function is called, the control shall return to the next instruction after the execution of the function. The address of the next instruction after the function call instruction is the return address of the callee function. Many processors use a dedicated register to store return address of the function. The register in that case is known as return address register. Return address of a function is stored in the return address register and/or in the frame on the call stack. For example in MIPS,

the return address of a function is stored in the return address register, but not on the call stack. The return address is required by GDB for the execution of *finish* command. It is also used to get the call stack information in GDB. The method to get the return address value of the function is stated in the debug information. For example in MIPS, the return address of currently executing function is obtained from the return address register, while for other functions it is extracted from the stack frames. In the later case the offset of the memory location within the frame where the return address is stored is given in the debug information.

2. Each time a function is invoked, a new frame is created on the call stack. Each time a function returns, the frame for that function is removed from the stack. The debug format provides information about the size of the memory required by the frame for each function.

3. When the program execution breaks, the GDB commands can be used to display the values of variables. The variables are considered from the current frame of execution unless any other frame is selected from the call stack. From the debug information, the offset of the memory location where the variable is stored is known. The address of the variable can be computed by adding the frame pointer to this offset. Hence, to store and retrieve the values of variables GDB needs to know the frame pointer. The frame pointer is usually kept in a CPU register. Therefore, the GDB needs to know the frame pointer register for the target architecture.

4. The old frame pointer is stored on the call stack at some fixed offset in the current frame when a function is invoked. This offset is dictated by the ABI of the target processor. The value of old frame pointer is needed to unwind the stack frame. Hence GDB needs to know the offset where the previous frame pointer is stored in a stack frame.

The debug information for a function includes the extents of memory addresses where the instructions of the functions are storeed. These are indicated by two addresses known as *low-pc* and *high-pc* for each function.

When the execution of the program is broken, the value of the current PC is compared with the extents of all functions to find the function for the current instruction where the execution was broken.

Chapter 4

GDB Simulator Interface

In a normal scenario, the GDB is used to debug programs on the native machine. This is known as local debugging. GDB can also be used to debug programs running on a remote machine connected through a network connection. This scheme is known as remote debugging. GDB is also used to debug the programs running on processor instruction set simulators. For our work, we are interested in using GDB for simulator target.

4.1 Types of simulators handled by GDB

GDB provides two kinds of debugging facilities for simulators. In one kind of organization, the simulator runs on the same machine where the GDB runs and simulates the instruction set of the target machine. In the other kind of organization the GDB and the simulators reside and execute on different machines and use remote debugging facility of GDB.

4.1.1 Native simulators

GDB includes built-in instruction set simulators for several processors such as Hitachi h8300 [4], Motorola 68hc11 [8], MIPS [14], PowerPC [12] and ARM [1]. These simulators use the remote-sim interface to utilize the debugging facility of GDB. The remote-sim interface provides the functions through which the GDB can access and control the simulator for debugging a program. In this approach the GDB is built for simulator of specific target architecture. A new instruction set simulator can be interfaced to work with GDB using simulator interface of GDB. If the simulator runs on the same machine where the GDB runs, it is a native simulator. The interface between the two can be a procedural interface.

4.1.2 Simulators on remote machine

GDB remote protocol is used for remote debugging of programs. Usually this facility is used for a target machine which does not have operating system powerful enough to run a full-featured debugger. The host machine has a full featured GDB and the target machine has a program known as GDB stub which would communicate with the host GDB and would implement basic GDB functionality on the target side. Here the target machine can either be a hardware machine or a simulator of the specific target architecture for which GDB is built. The communication is performed over a serial line, or over an IP network using TCP or UDP. To use this approach, a simulator has to implement the GDB stub.

4.2 Using GDB for FSim simulator

The instruction set simulator (FSim) is generated from the processor description written in the Sim-nML language and is used to simulate execution of the program for target architecture. However the simulator does not provide any insight into the program being executed or the processor state. We have interfaced our simulator with GDB to utilize its debugging functionality.

The FSim is a retargetable simulator. In order for GDB to support FSim, the GDB must incorporate runtime retargetability. This is however not possible within the architectures of GDB. Hence we create and add a new target known as *simnml* to the existing targets of the GDB. We configure GDB to run with simulator using target *simnml*. The *simnml* target initially has dummy information to create the target architecture. During the initialization of the target architecture of GDB, the dummy information about the target is replaced by the actual information from the processor description. Hence, we need an interface which would take the target-specific information at runtime from the processor description.

We use the existing features of GDB which make it retargetable. For this we use existing interface of GDB to connect to the simulator and add a new interface to update target architecture information of GDB. This information is gathered from the processor description at runtime of GDB.

4.2.1 Addition of *simnml* target to GDB

As mentioned earlier, the GDB needs to be retargetable at its runtime to support FSim. However as per the GDB architecture, we need to specify a target to build GDB. We therefore added a new target, *simnml,* to the existing targets of the GDB. To add a new target *simnml* to GDB, we had to make some changes in the configuration files. The changes are as follows.

1. The topmost level of configuration files checks whether the particular target architecture is supported by GDB or not. We had to add the name of the target architecture in the top level configuration files.

2. The Binary File Descriptor (BFD) library is used to read the contents of the executable file which include the metadata of different sections of the file, executable code and the debug information. Some of the routines used in the BFD library depend upon the target architecture properties such as the endianity of the target. Hence, whenever we want to add a new target in GDB, we have to also add it in the configuration files of BFD. The *bfd_ arch_ info* structure in BFD stores the basic information about the target required by BFD library, and its instance for a target *<arch>* is defined in the *cpu-<arch>.c* file. An instance of *bfd_ arch_ info* structure is defined for each of the supported target architectures. For the target architecture *simnml*, we defined the instance of structure *bfd_ arch_ info* as *bfd_ simnml_ arch* in the file named as *cpu-simnml.c*. The name of the variable (*bfd_ simnml_ arch*) is then listed

in the *config.bfd* file. An array of pointers of type *struct bfd_arch_info* is defined in *archures.c* file. Each entry in this array points to the instance of *bfd_arch_info* structure defined for a certain target architecture. The *simnml* target related information is added in this array.

3. BFD library also performs object file format handling, such as ELF handling, for GDB. The internal structure of an ELF file varies with only two architecture dependent parameters, i.e., endianity and the word size of the target machine. In order to support different target-specific features of the ELF, the corresponding BFD library provides a list of byte exchange routines which are used by GDB while handling an ELF file targeted for a processor of particular endianity and word size. Two such lists for different endianities are defined in the source file *elfxx-target.h* within the GDB. During the configuration of GDB target specific list is to be used depending on the target endianity. Since we need runtime retargetability, we don't know which list to use at compile time. Hence we select a dummy ELF vector. We replace this dummy vector by the appropriate target ELF vector when the target architecture parameters are retrieved from the processor specification. An array of supported target ELF vectors is defined in the file *targets.c*. We add our new target ELF vector (*bfd_elf32_simnml_vec*) in this file.

4. Whenever we add a target to GDB, we have to add a *<arch>-tdep.c* file in the source of GDB. This file contains basic layout of target machine's processor chip (registers, stack etc). It contains the target initialization routine which is called to initialize the target architecture. In our implementation, we created *simnml-tdep.c* file for this purpose. The routines in this file initialize the target specific layout using the Sim-nML processor description at runtime.

To add a new simulator to GDB, we created a directory for the simulator target inside the *sim* directory of GDB source and provided the implementation of the interface between GDB and FSim simulator in this directory. This interface is compliant to

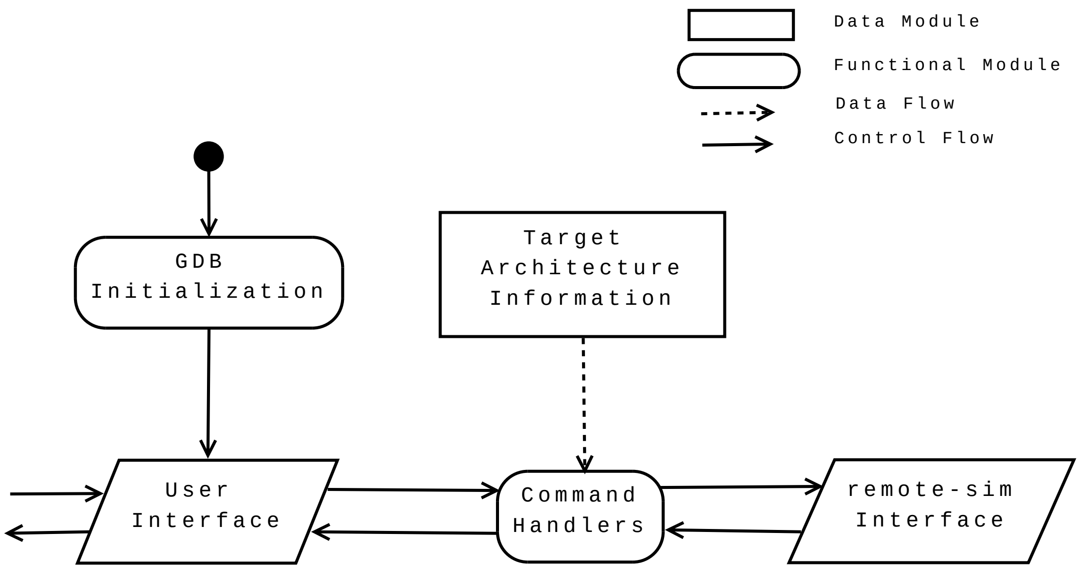


Figure 4.1: GDB with *remote-sim* interface

the remote-sim interface of GDB.

4.2.2 Implementation of the *remote-sim* interface

When the GDB includes a CPU simulator that one can use instead of hardware CPU to debug the programs, an interface is used to connect simulator to the GDB. This interface is known as remote-sim interface and is provided by GDB.

In figure 4.1 we show the execution of GDB with the remote-sim interface. After the initialization, control is passed to the user interface module. Whenever user enters a valid GDB command, appropriate command handler gets invoked. A command handler completes the task associated with the command and returns the control to the user. The command handlers use remote-sim interface to control the execution and to gather information about the state of the simulated program and the CPU. The command handlers need the information about the target architecture for their execution.

The target architecture of GDB is configured by using the information of one of the supported hardware targets. This way the target architecture information is available to GDB at compile time.

The remote-sim interface provides the following template functions which should be implemented by the target simulator. In our approach, these functions are implemented in the FSim simulator.

- SIM_OPEN: This is the main entry point for the simulator. It creates fully initialized simulator instance. This function is called when the simulator is selected from the GDB command line. In our implementation this function creates the processor model from the Sim-nML description using the first phase of FSimg and returns the success value back to GDB. From this point onward, GDB is connected to simulator.
- SIM_CLOSE: This function destroys the simulator instance created by SIM_OPEN. All resources which are allocated till that point, are released and certain other book keeping functions are performed. From this point onwards, GDB is no longer connected to the simulator. In our implementation the FSim instance generated for the particular target and the binary executable of the generated instruction set simulator (FSim) is deleted by this function.
- SIM_LOAD: This function loads binary program into simulator memory. In our implementation the second phase of FSimg is carried out in this command. This is achieved by decoding the input program and by creating the execution table. Finally, simulator is created as a dynamically linked library (DLL).
- SIM_CREATE_INFERIOR: This function prepares to run the simulated program. It should initialize target processor registers and state variables and set command line arguments.
- SIM_FETCH_REGISTER: This function returns the contents of the requested register. Register number is passed as an argument to this function.
- SIM_STORE_REGISTER: This function overwrites requested register's contents by the value given in the function parameters.
- SIM_READ: This function returns the contents of a requested memory location.

- SIM_WRITE: This function overwrites requested memory location's contents with the value provided in the function parameters.
- SIM_RESUME: This function runs the target binary program in the simulator. If the step flag which is given as argument to this function is false, simulator runs until it encounters a breakpoint or until the program ends. If it is true, the simulator simulates only one line of the source program.
- SIM_STOP: This function stops the simulation process.
- SIM_STOP_REASON: This function returns the reason why the program has been stopped. A list of valid reasons is given below.
  - EXITED: The program has been terminated. The exit status is returned through a signal.
  - STOPPED: The program has been stopped. A signal value is returned to identify the reason. It can be a breakpoint instruction, an illegal instruction, a *sim_stop* request, completion of single step, an internal error or an access to wrong memory location.
  - SIGNALED: The program has been terminated by a signal.
  - RUNNING: The program is still running.
  - POLLING: The simulator is waiting for a new command.

The above mentioned remote-sim interface is used to communicate with GDB and to create the FSim instance, using the FSimg. Once the simulator instance is created, it is driven by the remote-sim interface routines.

4.2.3 The *fsim* interface for GDB

In a usual scenario, in which a GDB is connected with a simulator using remote-sim interface, the GDB is configured for one of the supported target architectures. However remote-sim interface is insufficient to provide retargetability support to the GDB at runtime. We added fsim interface for this purpose. The fsim interface is

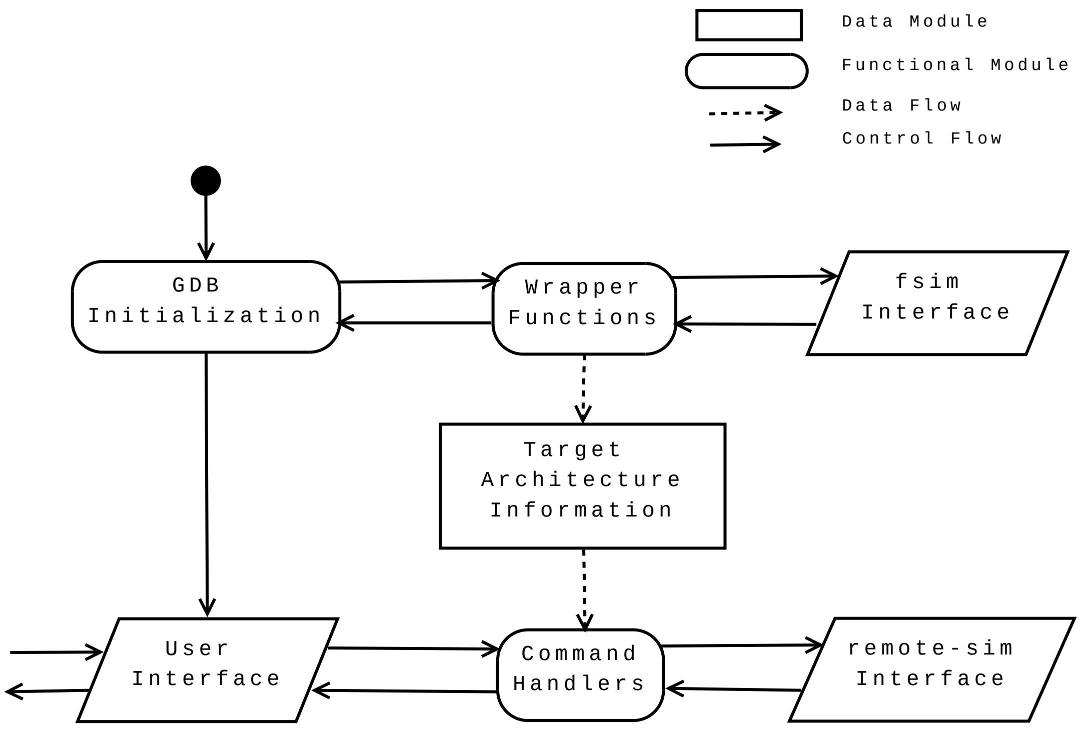


Figure 4.2: GDB With *fsim* Interface

used by the simnml target of GDB and provides the runtime retargetability of the GDB. Implementation of this interface is provided by the FSim simulator.

In figure 4.2 we show the execution of GDB with the fsim interface. As shown in figure 4.1 GDB uses the target architecture information available at compile time. GDB did not have support to change the target architecture information at runtime, which is necessory for runtime retargetability. Hence we developed the fsim interface to get the target architecture information at runtime from Sim-nML specification. The wrapper functions act as an interface between initialization routines of GDB, target architecture information and the fsim interface. The wrapper functions update the target information of GDB from the current Sim-nML target by using the fsim interface. This way once the initialization of GDB is completed, the command handlers can access the information of the target architecture specified by the Sim-nML processor specification.

The fsim interface uses the following template functions which should be implemented to provide runtime retargetability support to GDB.

- get_sim_endianity: This function returns the endianity of the simulator target. The target endianity of the GDB is set to the endianity returned by this function. For example, target endianity of GDB for i386 target architecture will be little endian.

- get_sim_bits_per_word: This function returns the size of a word in the target architecture. For example, for a MIPS target, bits per word are 32, while for Intel 8086 architecture size of a word is 16 bits.

- get_sim_bits_per_addr: The function returns the size of address in the target architecture. For 8085 architecture this function will return 16 where as for MIPS it returns 32.

- get_sim_num_regs: The function returns the number of registers in the target.

- get_sim_pc_regnum: The function returns the index of the PC register in the register list. Here all registers of the target are considered to form an array and the index of PC register is returned.

- get_sim_sp_regnum: The function returns the index of the SP register in the register list.

- get_sim_regnames: The function returns the names of registers in the target architecture. Register names are required by GDB to communicate with the user. User can set or get register values using the register name.

- get_sim_regtypes: Type of the register specified by the index given as argument is returned by this function.

These functions form an interface between the GDB and the retargetable simulator to update the target architecture of GDB at the runtime of GDB. A retargetable simulator can use this interface along with the remote-sim interface to enable GDB support for debugging.

4.2.3.1 Wrapper functions for *fsim* interface

In our approach, we implemented the existing remote-sim interface and created a new fsim interface to enable debugging of FSim simulator. The implementation of fsim interface is generated at runtime from the Sim-nML specification and then used by the GDB. Implementation of runtime retargetability of GDB required update of certain architecture information by the Sim-nML target. At compile time these variables were set to some dummy values. The wrapper functions update these dummy values as described in section 4.2.1 with the information from current target architecture. The information is gathered from the Sim-nML processor specification through the fsim interface. We have implemented these functions in the file *simnml-tdep.c.* A list of these wrapper functions used for fsim interface is given below.

- output_reg_interface: All routines specified in the fsim interface are generated at the runtime of GDB using this function. The Sim-nML processor description is read and is used to generate the definitions of the routines of fsim interface.

- update_simnml_architecture: This function calls the routines to initialize GDB architecture from simnml target. It calls routines from BFD library to update the target architecture information available in GDB and in BFD library.

- update_gdb_sim_arch: It uses the register information functions of the fsim interface to update the register information of the target architecture of GDB. The information includes total number of registers of the target and register numbers of the stack pointer and program counter register.

- update_arch_info_type_from_simnml: This function updates the architecture information in BFD library (i.e. the *bfd_arch_info* data structure) and is defined in *cpu-simnml.c* file of BFD library.

- update_bfd_sim_vec: This function updates the target elf vector (i.e. the *bfd_elf32_simnml_vec* data structure) in the BFD library. An elf vector is chosen according to the endianity of the target and the target elf vector of GDB is modified to point to the newly selected elf vector.

GDB simulator interfaces have enabled GDB to be used for debugging the programs running on the retargetable simulator FSim. The fsim interface is used to initialize and update the target architecture of the GDB from the Sim-nML specification, whereas the remote-sim interface is used by command handlers for performing execution control and accessing the runtime debug information of the program running on simulator.

Chapter 5

Implementation of Debug Mechanisms in Simulator

GDB commands are of two types. The first type of commands control GDB behavior, while the second type of commands control and monitor behavior of the program being debugged. It is the second type of commands which are implemented by the target. In our case they are implemented by FSim simulator. These commands handle breakpoints, control the program execution and modify/display memory state. They also include the commands for register interaction.

5.1 The breakpoint mechanism

Breakpoints are used to suspend the flow of a program. Breakpoint mechanism includes creation, identification and deletion of a breakpoint. The creation of a breakpoint defines a condition to break the execution of a running program. Identification of a breakpoint means to stop the running program whenever breakpoint condition occurs. Deletion of a breakpoint causes the definition of the associated condition to be nullified.

In GDB the breakpoints can be set in two different ways.

1. Function name: The breakpoints set for a function name causes the execution

to be broken whenever the control reaches to this function. The ELF format stores the information about function (starting address and name of the function), even if the source is compiled without any debug flags. Hence, we can always set a breakpoint using a function name existing in the program.

2. File name and Line number: The breakpoint set for a source line causes the execution to be broken whenever the control reaches to that line. A line in the C program is typically converted to multiple machine instructions in the object code. We need to know the address of the first instruction for a particular line, where we want to set the breakpoint. The ELF file format does not store any line number information unless the debug information is stored. Hence, unless we have compiled the application with debug flags set, we can not set the breakpoint using this mechanism. If the file name is not given, breakpoint is set in the file containing the function corresponding to the current frame.

Once the program execution is started, it should stop when the value of program counter equals any of the breakpoint addresses. If none of the breakpoint addresses match the program counter, program execution does not halt intermediately. The execution of the program is carried out by the target processor. Therefore the breakpoint handling is done by the target side. It stops the execution whenever it encounters a breakpoint address and passes control to the breakpoint handler. The breakpoint handler routine in the core GDB then decides whether to break the execution or not. For example, in case of a conditional breakpoint, decision of breaking the execution also depends upon the evaluation of condition according to present values.

5.1.1 Debug information related to breakpoints

Different ways of setting a breakpoint need different types of debug information. When a function name is used to set a breakpoint, the information required is the starting address of the function. When a line number is used to set a breakpoint, debugger requires the mapping between the line number and associated address to set a breakpoint. In case of conditional breakpoints, debugger may also need to know the information of the variables if they are used in the conditional expression.

The mapping between the line number and corresponding address is provided as debug information in the executable file. In an ELF object file, the *.debug_line* section contains this information, where the following information is generated for every source line.

- Source file name
- Source line number
- Address of the first machine instruction translated from the source statement

The command handler matches the user specified specifications of breakpoint with the available debug information to find the starting address of the function or the source statement, and uses that address to set the breakpoint.

5.1.2 Types of breakpoints

Apart from the breakpoint command, few other commands also use the breakpoint mechanism in their implementation. These commands include *step*, *next*, *finish* and *until*. In order to handle these commands as well, the breakpoints are identified with certain types. In case of simulator we are concerned with following types of breakpoints.

- bp_breakpoint: This is a normal type of breakpoint and is created by *breakpoint* command.

- bp_until: This type of breakpoint is created by *until* command and is a temporary breakpoint.

- bp_finish: This type of breakpoint is created by *finish* command and is a temporary breakpoint.

- bp_step_resume: The *step*, *stepi*, *next* and *nexti* commands create this type of breakpoint. This type of breakpoint is a temporary breakpoint.

A temporary breakpoint is valid only in the context of the current debug command. Whenever the execution is broken due to a temporary breakpoint, the GDB removes all temporary breakpoints before it passes control to the user.

5.1.3 Implementation

In the target side, execution is stopped whenever a breakpoint instruction is encountered and control is passed to core GDB routines which handle further processing about the breakpoint. Breakpoints can be implemented on the target side in two ways.

1. In the first approach we keep a list of breakpoints. Before execution of every instruction, we compare the program counter register with the entries in the list. In case of any match, the program execution is broken and control is passed to the core functions of GDB. A flag is set which keeps track of the reason of break in execution. When the program resumes its execution, the occurrence of breakpoint is not checked for the first instruction to ensure that the same breakpoint is not encountered again and again. However this approach is an inefficient approach as it needs to search in a list of breakpoints for every instruction execution. Though this approach is generally suitable for the simulators, we use different and efficient approach in our simulator.

2. In the second approach, separate instructions such as a trap instruction, or a sequence of instructions are used to handle breakpoints. In this approach, the target side replaces the instructions at breakpoint address by these instructions. The original sequence of instruction bytes at the breakpoint address is saved in the data structure associated with the breakpoint. When a breakpoint is hit, the trap instruction or the replaced instructions are executed causing the control to be caught by GDB by exception mechanism or some similar mechanism. The program counter (i.e. return address) is then checked for breakpoints. In case a breakpoint is hit, execution stops and control is passed to core GDB routines. When the program resumes the execution, original

instruction is inserted back to the breakpoint address. The processor executes one instruction in single step mode and then GDB reinserts the trap instruction at breakpoint address for subsequent breakpoint. In case the breakpoint was a temporary one, the breakpoint is not reinserted. The program execution is then resumed in the normal mode.

We have used the second approach in our implementation. We create a breakpoint list, in which we store the actual instruction from the breakpoint address. The list is modified by 'create breakpoint' and 'remove breakpoint' commands from GDB. To insert a breakpoint, we replace the instruction at breakpoint address (in the instruction table) by a dummy function pointer, implemented in the simulator. The trap function does not increment the program counter. It only sets the breakpoint flag to true. Before executing every instruction we check the breakpoint flag. If it is set, the program execution is stopped and control is passed to GDB. When the program execution is resumed, post processing of the breakpoint is done. In this step, we execute the original instruction stored in the breakpoint table and resume the normal program execution.

5.1.4 Data structures used for breakpoint

GDB creates a breakpoint data structure for every breakpoint. It keeps a list of currently active breakpoints. The breakpoint data structure of GDB stores the following information.

- Type of breakpoint: The type of breakpoint is related to the command that caused the creation of this breakpoint as described earlier.
- Disposition Information: This specifies the action to be taken when the breakpoint is hit. For a normal breakpoint, there is no need for any additional action. For *step_ resume_ breakpoint* or *finish_ breakpoint,* the action would be to remove the temporary breakpoints.

- Number: Each breakpoint has a unique number, which is used for identification by the GDB.
- Breakpoint Location Structure: This structure stores the information related to the breakpoint location. This includes the address, number of breakpoints at this location and a pointer to the BFD section associated with this address.
- Line number and source file name.
- List of GDB commands to be executed when the breakpoint is hit. This may be created because of display command of GDB for example.
- Frame Id: Frame Id corresponds to the identification of frame on the call stack. When it is non-zero, execution breaks only when current Frame Id equals the stored value.
- Conditional Expression: This condition is checked whenever the breakpoint is hit. If condition is false, the GDB resumes the normal execution without passing the control to command handling and execution is not broken.
- Hit count: It stores the number of times a breakpoint is hit.

FSim simulates the behavior of the target side of GDB. It stores target side list of breakpoints. The breakpoint structure on the target side is simpler than its core GDB counterpart. It has following fields.

- Enabled flag: it states whether the breakpoint is valid or not.
- Breakpoint address: This field stores the breakpoint address. It is required to distinguish between two breakpoints. No two breakpoints on target side have same address. This field is compared with the program counter to identify the instruction at breakpoint address, which is to be executed when program is resumed.

- Original instruction: The function pointer corresponding to the original simulated instruction at the breakpoint address is stored in this field. The operands of the instruction are taken from the instruction table.

5.2 The watchpoint mechanism

A watchpoint is a mechanism to stop the program execution whenever the value of a pre-defined expression changes, without predicting a particular instruction address where this may happen. An expression can be a single variable or many variables combined by operators. When an expression has multiple variables, the value of the expression may change due to change in any of the variables or may not change even when multiple variables change. A watchpoint is also known as a data breakpoint. Depending on the target, watchpoints may be implemented in the software or in the hardware. If implemented in the software, GDB handles watchpoints by single stepping the program, computing the expression and noticing the change in from its previous value. In case of hardware implementation the target side monitors the access to memory addresses and breaking the execution whenever any memory location is accessed. The GDB prefers a hardware implementation if available in the target. If GDB can not set a hardware watchpoint, it sets a software watchpoint.

5.2.1 Types of watchpoints

A watchpoint is used to stop program execution by examining the memory accesses. A memory access by the program can be either to read from or to write into the watchpoint address. There are three types of watchpoints according to the type of memory access watched by the watchpoint.

1. bp_hardware_watchpoint: This type of watchpoint is set by using *watch* command. It sets a watchpoint for an expression *expr*. GDB breaks the execution when the value of the associated expression changes.

2. bp_read_watchpoint: This type of breakpoint is set by using *rwatch* com-

mand. If this type of watchpoint is set, GDB breaks the execution when the value of variables is read by the program.

3. bp_access_watchpoint: This type of breakpoint is set by using *awatch* command. If this type of watchpoint is set, GDB breaks the execution when the value of the associated variable is either read from or written into by the program.

5.2.2 Debug information about watchpoint mechanism

Watchpoints monitors an expression for the data access. This requires the knowledge of memory locations of the variables which constitute the expression. A watchpoint can also be set on an array, and as the elements of an array can also be accessed individually, the size of the memory region to be watched is also needed by the GDB. This information is generated by the compiler in the debug section.

5.2.3 Implementation

During initialization of the target architecture of GDB, a flag is set suggesting whether the target can support a hardware watchpoint or not. If hardware watchpoints are not supported the GDB uses software watchpoints. In our implementation we set the hardware watchpoint flag and support the hardware watchpoint functionality through simulator.

Target side of the GDB provides the routines to insert watchpoints and to remove them. A routine to acknowledge whether the current break in the execution is due to a watchpoint or not, is also implemented by FSim simulator. In a usual scenario, target hardware such as the x86 processor have support for hardware watchpoint, and generate an exception on the occurrence of a watchpoint to return the control to GDB. However in case of a simulator, we have to monitor every memory access and compare the requested memory address with the memory region specified by the watchpoints. If the memory address passed to the *write_memory* routine is within the range specified by the watchpoint and if the watchpoint is either *bp_ hardware_ watchpoint* or *bp_ access_ watchpoint* type then the execution is stopped for the watchpoint and control is passed back to the GDB. Similarly for the *read_ memory* routine if the memory address is within the range of the watchpoint addresses then the control is passed to the GDB if watchpoint is of either of *bp_ read_ watchpoint* or *bp_ access_ watchpoint* type. The FSim simulator keeps the watchpoints in a linked list. Hence as opposed to the watchpoints provided by the hardware targets, the number of watchpoints in FSim is not limited.

5.3 Single stepping and program trace mechanism

Step, *stepi*, *next* and *nexti* are the commands used for single stepping. *Step* and *next* commands execute single line of source code and stop the execution at next line of source code. The *stepi* and *nexti* commands execute a single machine instruction. In case of a subroutine call, the *step* and *stepi* commands stop the execution at the first line or at the first instruction in the subroutine. However, the *next* and *nexti* commands break the execution after the subroutine call is completed.

5.3.1 Debug information for single stepping

Single stepping mechanism requires certain debug information about the program being traced. The debug information required by each of these commands is discussed below.

The *step* command need to use the debug information to check whether the execution of source line is completed or not.

The *nexti* command need to use debug information to check whether the last executed instruction made a call to a function or not.

The *next* command need to use debug information for both purposes, to check whether the execution of source line is completed or not; and to check whether the last executed instruction made a call to a function or not.

In order to control line level execution, GDB needs to know the line number information of every source line. This information is generated by the compiler. A line of source code is converted to several machine instructions. The debug information about a source line includes the line number in the source file, starting address of the first instruction of the block of machine instructions to which the source line is mapped, and the starting address of the first instruction of the block of instructions to which the next source line is mapped.

In order to identify whether a particular instruction lies within a function or not, GDB needs the information about the boundary of the function. The boundary of a function starts at the starting address of first instruction of the function and ends after the last byte of the last instruction. For every function in the program the boundary information is provided by the debug format.

5.3.2 Implementation

Each of these commands controls the execution of the program until a certain condition occurs. The simplest way of controlling the execution of the simulated program would be to simulate the execution of one instruction in the simulator and then to pass the control back to the simulator engine of the GDB. The GDB would then check for the occurrence of the condition specified by the command. If it occurs, the control would be passed to the user and GDB would go into interactive mode to accept commands from the user. However, if the condition does not occur then the control would be passed to the simulator to execute next machine instruction. This process of execution control would continue until the occurrence of the condition specified by the command. This approach would work but is inefficient due to the large overhead of control switch between GDB and simulator. Therefore, we execute a chunk of machine instructions before transferring control from simulator to GDB. This is achieved by providing certain information about the debugging command to the simulator which is used to check for the occurrence of the conditions associated with the command on the simulator side itself, reducing the control switch.

5.3.2.1 Driver function of simulator

The machine instructions of the program are executed by the driver function of the simulator, i.e. *run_simulator*. This function has a loop in which the execution of instructions is simulated. One machine instruction is executed in each iteration of this loop. The information about the current GDB command is passed through two parameters to this function, namely *command* and *stop_address*.

-	-	-	-	-	N	SI	LLC

`LLC - Line Level Control Flag`
`SI - Single Instruction Flag`
`N - Next Flag`

Figure 5.1: The *command* parameter of simulator driver function

As shown in figure 5.1 the *command* parameter is an 8 bit number and carries three flags for three different types of controls. Remaining bits of the *command* parameter are not used and are always set to zero. The three flags are line level control flag, single instruction flag and the next flag. The line level control flag is set to 1 for the commands which control the program execution at the granularity of lines of the source code. The single instruction flag is set for the commands which control the program execution at the granularity of the machine instructions. The next flag is used to specify to step over the function calls while single stepping. Different combinations of these three flags are used for different GDB commands as shown in figure 5.2.

The value of *stop_address* parameter is valid only if the line level control flag of the command parameter is set. A valid *stop_address* value points to the starting address of the next source line.

N	SI	LLC	Commands
0	0	1	Step
1	0	1	Next
0	1	0	Stepi
1	1	0	Nexti
0	0	0	run, continue, finish

Figure 5.2: The *command* parameter flags for GDB commands

The two parameters are used to collect the information about the current debug command and to devise the terminating conditions for the loop accordingly. If there is no terminating condition associated with the current command, the *run_ simulator* function executes the instructions in an infinite loop. We show the working of the driver function in algorithm 1. The simulator state is maintained by a global variable *SIM_ STATE.* Its value is initialized to *RUN* before starting the execution of instructions. The occurrence of a breakpoint or watchpoint might change its value to *STOP.* In that case the loop terminates and the control is returned back to the GDB. For the *stepi* and *nexti* commands a single instruction is executed and control is returned to the GDB. In the current implementation of *run_ simulator* the next flag of the *command* parameter is not used. It can be used in future to enhance the implementation of the next and nexti commands on the simulator side without any change in the GDB source code.

5.3.2.2 Implementation of commands

The execution control commands are implemented in remote-sim interface and are described here.

- *step*: For the *step* command the *LLC* flag of the *command* parameter is set to 1 and the *stop_ address* parameter is set to the address of first machine instruction of the next source line. During compilation of the program, every line of source code is translated to a block of machine instructions. For the *step* command, the command handler executes the block of instructions which correspond to a single line of source code. The program execution stops whenever the program counter contains an address which is outside the block of instructions corresponding to the line to step over. This can happen due to three different reasons. First, the block of instructions gets executed and then the program counter points to the first instruction corresponding to the next line of source code. Second, the source line had a branch instruction and after its execution the program counter points to the machine instruction corresponding to

Algorithm 1 run_simulator (stop_address, command)

```
1: start_line $\leftarrow$ program counter
2: SIM_STATE $\leftarrow$ RUN
3: if command $\cap$ 0x1 then
4:    {step/next command handling}
5:    while true do
6:       Execute the current instruction
7:       if SIM_STATE = STOP then
8:          break
9:       end if
10:      if ( ( PC < stop_address ) $\cap$ ( PC > start_line ) ) is false then
11:         {Program counter points to different source line}
12:         break
13:      end if
14:   end while
15: else if command $\cap$ 0x2 then
16:    {stepi/nexti command handling}
17:    Execute the current instruction
18: else
19:    {continue/finish command handling}
20:    while true do
21:       Execute the current instruction
22:       if SIM_STATE = STOP then
23:          break
24:       end if
25:    end while
26: end if
```

some other line of source code. Third, the source line had a function call which causes the program counter to point to the machine instruction corresponding to a different function. By comparing the program counter with *start_line* and *stop_address* after the execution of each instruction, we identify the change in line and handle that accordingly.

- *next*: Handling of the *next* command is same as that of the *step* command except for the source line having a function call. As there is no mechanism available to identify the call instruction on the simulator side, in the current implementation the next flag is ignored. When the simulator executes a call instruction the program counter crosses the boundary of the current source line, and the control is passed back to the command handler. GDB then analyzes the reason of break in the execution and if it is a function call, the control is passed back to the simulator to continue the execution. In this way the program execution is then continued until the function call is completed. However, if the reason in execution break is not a function call then the control is passed back to the user.

- *stepi*: For the *stepi* command the *SI* flag of the *command* parameter is set and *LLC* flag is not set. The *run_simulator* algorithm then causes the execution of single machine instruction and returns the control back to GDB.

- *nexti*: The *nexti* command is handled by the simulator in the same way as the *stepi*. As there is no mechanism available to identify the call instruction in the simulator, the next flag is ignored. The control is passed back to the GDB after execution of every single machine instruction as in the case of the *stepi* command. In our implementation command handler on the GDB side indentifies the function call and passes the control again to the simulator in that case.

- *continue*: For this command the value of the *command* parameter is passed as 0, to indicate that the execution must be broken only upon the breakpoints.

- *finish:* Command handler of the *finish* command inserts a temporary breakpoint of type *bp_finish* at the return address of the function and then passes control to the simulator. On the simulator side, the handling of this command is same as that of the *continue* command. The value of the *command* parameter is passed as 0, indicating that the simulator should not stop until it encounters a breakpoint or a watchpoint.

Command handler of the *next* command reduces the time spent on checking whether the execution of a source line is completed by using a *step_resume* breakpoint. It inserts a *step_resume* breakpoint at *stop_address* before passing control to the target. The GDB checks for the occurrence of this breakpoint and implements the behavior of the *next* command.

5.3.2.3 Efficient implementation of *next* and *nexti*

The current implementations of *next* and *step* commands on the simulator side are identical. These commands are distinguished by the GDB by identifying the function calls and passing the control back if necessary. Similarly the implementations of *nexti* and *stepi* commands on the simulator side are also identical. The simulator does not distinguish a call instruction from the other instructions. When this capability can be built in the simulator, the *next* and *nexti* commands can be handled by the simulator itself.

5.4 Call stack analysis mechanism

During runtime, a program maintains a call stack in memory. The call stack stores the parameters, local variables and the return addresses for the active subroutines. The commands to show the execution trace or the function call trace use this call stack information. GDB also maintains a pointer to one of the frames on the call stack. This is used to show/modify variables related to that frame. Usually this pointer refers to the outermost frame. However for convenience it can be changed. This change has no effect on the execution behavior since it has no relation to the machine frame pointer register. GDB provides commands to modify this pointer, i.e., to select one of the frames on the call stack as default frame related to the display of variables.

The commands that use call stack include *frame*, *backtrace*, *up* and *down*. The frame pointer maintained by the GDB is modified by *frame, up* and *down* commands. The *backtrace* command is used to show information from all active frames.

5.4.1 Debug information related to call stack analysis

Debug information required for these commands is target architecture dependent. This includes the return address register number and the call stack unwinding mechanisms. The stack and program counter unwinding routines which are registered during the initialization of target architecture are used by command handlers of these commands. Each frame on the call stack represents execution of a function. Thus when GDB is asked to display the frame information, it also displays the information about that function. Hence debug information required to display a frame includes the name of the function, name of the source file, line number at which the execution is stopped or the line number at which subroutine of the next frame is invoked.

GDB can select one of the frames from the call stack as default frame for display of variables. For this operation GDB needs the saved register values. The information about the frame is obtained from the call stack by using unwinding routines of the GDB. The stack pointer together with the program counter serves as the frame id. A call frame also stores the value of the frame pointer referring to the previous frame. The offset of location where the previous frame pointer is stored within a frame and the size of current stack frame are given by the debug information. This information is essential as previous frame can only be recovered from the current frame pointer. Similarly the next frame pointer can be obtained by adding the size of the current stack frame to the current frame pointer. The offset of memory location where the return address is stored is constant from the frame pointer and is also given by the debug information.

5.4.2 Representation of call stack in GDB

GDB uses a structure *frame_info* to store the information about a frame on the call stack. A linked list is created to store the call stack information of the target machine in GDB, where each node of the list is of type *frame_info* and corresponds to a single frame on the call stack. Head of the list stores information about the current frame whereas the last node contains the information about the innermost frame. The *frame_info* is used to store the frame pointer, the stack pointer, the return address, the register values and the stack frame number of the stack frame in the call stack. The command handlers only have the current frame information which is used along with the debug information to extract the complete call stack of the program from the target machine.

5.4.3 Implementation

GDB accesses all registers as if they form a single register bank. A register is accessed by its index in the register bank. The order of the registers within the register bank is dictated by the compiler. The order of declaration of the registers within the processor specification is exactly the same as used by the compiler of that processor. The program counter and stack pointer registers are frequently used by the command handlers of all the commands. The indices of these registers are not specified in the debug information. This information is provided by the Sim-nML specification. During initialization of the target architecture of GDB from Sim-nML specification, these indices are calculated and registered with GDB.

GDB generates the linked list which represents the stack frames of call stack from the current values of stack pointer, frame pointer and program counter registers. Information about the current routine is used to access the previous stack frame. The unwinding routines extract the previous frame information from the current frame. The return address is used to identify the previous function and to associate static information about the frame such as the name of the function and source file. After frame unwinding, the extracted information about the previous frame is used to add a new frame to the list of frames in GDB. The process of frame unwinding is carried out until we reach the function *main*.

After the generation of linked list, remaining functionality of these commands is platform independent. Selecting a frame just need to set the values of simulator registers to that of the selected stack frame. As local variables are accessed by using the frame pointer, one can access the local variables of only currently selected stack frame.

5.5 Program state control mechanism

Program state is maintained by the data stored by the program in the memory. The program uses variables to access the data. Accessing a variable is basically to access the chunk of memory which stores the value of variable. To access a variable, GDB generates the call to load or store routines and passes them the address of the variable along with the size. Controlling the program state includes presenting the memory contents and altering them by the user defined values. Routines to access the memory of the target machine from GDB are registered during the initialization of the target architecture. In our case, FSim simulates the target memory for the simulated program and acts as target machine for GDB. Hence during the initialization of the target, the routines to access memory from simulator are registered to GDB.

The GDB is configured for target endianity at runtime. Hence the value returned from the simulator shall be in the target endianity. If there is endianity mismatch between host and target it is taken care by display routines of GDB. This point becomes important in case of arrays, because if the GDB is asked to print the value of array, it generates a combined read request for the complete array. For example, the array is of size ten and each variable has size of four bytes then the memory request will comprise of starting address of array and 40 bytes. Assuming that target and host endianity does not match, if the endianity mismatch would have been handled by the simulator, the simulator would reverse the 40 bytes and then pass it to the GDB. But the correct solution here is to reverse each four bytes of all ten elements of the array. Due to updating of target architecture from Sim-nML description, we can use this functionality of GDB.

The commands to display the memory contents take the address and the size as parameters and pass it to the memory handler routines. Instead of accessing routines for hardware target, GDB uses memory access routines provided by the GDB simulator interface.

GDB can be used to set conditional breakpoint where conditions are defined by using variables. These variables are evaluated every time GDB hits the particular breakpoint and the access to the variables is made in the same way as described above.

5.6 CPU state control mechanism

FSim is a retargetable simulator. For FSim, the information about registers such as the names and the sizes of registers change with every processor description. Hence, the information about the registers of the target architecture of the GDB also changes and it should be updated from the simulator, at runtime. The commands to access the register include *info reg* and *p $regname.* The *info reg* command displays names and values of all the registers of the target machine. A command to display the value stored in the register accepts register name as an argument. This name is compared with the list of register names made available by the Sim-nML specification for the processor. If the name is matched with any of the members of the list, the index of that member is used as the register number in the register bank and is passed as an argument to the subroutine which retrieves the value stored in the register.

The fsim interface provides the functions which return the list of register names and the value of register for the given register number. These functions are generated from the Sim-nML description at runtime and form a shared library which is then used by GDB to access the registers. These routines include load and store operations on registers, to get the value of the current PC, those for to get the register names, to get the register type and to display the values of all registers.

Other commands which use the same underlying functionality did not require any further enhancement. These commands include *display, until, finish* and *info frame*.

The disassembly of Sim-nML target was implemented successfully in an earlier thesis by Nitin Kumar Dahra. In our work we merged it with our implementation. Other GDB commands such as *list* which lists the source file are target architecture independent and no enhancement was required to tune them for the retargetable simulator.

Chapter 6

Results and Conclusions

In our work we have interfaced FSim with GDB so that the debugging facility of GDB could be used to debug the programs simulated by FSim. In this chapter we discuss the results and conclusion of this work. We have carried out our experiments on various types of applications compiled for two different target architectures. Aim of these experiments was to validate the use of GDB for the retargetable simulator FSim. The experiments included using various commands of GDB (which is built for retargetable simulator FSim) to debug the programs compiled for different target architectures and simulated by FSim.

6.1 Experimental setup

We tested the GDB and FSim integration to debug the programs compiled for two different architectures, namely, PowerPC603 and MIPS32. Sim-nML processor descriptions of MIPS and PowerPC contain all the instructions except those related to caches and processor pipelines. The instructions related to coprocessor-1 of both machines, i.e., floating point instructions are also implemented.

6.1.1 Machine settings

We have used two different configurations to test GDB. The configurations of these machines are as follows.

- Machine1: Intel Pentium-4 2.4 GHz a little endian 32-bit processor with 256 MB RAM running Linux-2.6.22
- Machine2: Intel Dual Core 2.13 GHz a little endian 64-bit processor with and 256 MB RAM running Linux-2.6.18
- Cross compiler for PowerPC603: gcc version 4.2.2, binutils version 2.18, glibc version 2.7
- Cross compiler for MIPS32: gcc version 4.1.2, binutils version 2.18, glibc version 2.6.1
- GDB version 6.3

As both cross compilers were using NPTL (native pthread library) [41], we changed the startup code provided by GNU C library (glibc) [25] to skip the pthread handling routines. We implemented few programs which perform diverse functionalities and use different programming constructs to cover a large set of instructions for these processors. These programs were then compiled for PowerPC, MIPS and the host machine. For each program, the cross compiled binary executables were debugged by the GDB for retargetable simulator, using GDB commands applicable for the program. Same set of GDB commands were then used to debug the binary executable of the native machine. The outputs of these three debug sessions were then compared for correctness.

Firstly, basic programs for checking all arithmetic and bitwise operations were implemented. These programs were then debugged successfully using GDB. Few programs for solving some basic mathematical problems were implemented. These include programs to calculate matrix multiplication, to find palindrome numbers and perfect numbers, program for finding proper factors of a number, for computing Fibonacci series, to find prime numbers and to rotate an array.

Various algorithms for sorting, searching and some basic algorithmic problems were implemented and tested. The algorithms include bubble sort, heap sort, merge sort, selection sort, quick sort, linear and binary search, tower of Hanoi, and n-queens problem.

As part of the testing, few functions from glibc library were debugged. These functions have much larger size and take help of verious system calls for their implementation. The debugged programs were using functions such as *printf*, *scanf* and the functions performing file operations such as *fopen, fwrite, fread, fprintf* etc. For file operations, we wrote a program for operations on linked list of structures which would be stored in a file and retrieved again.

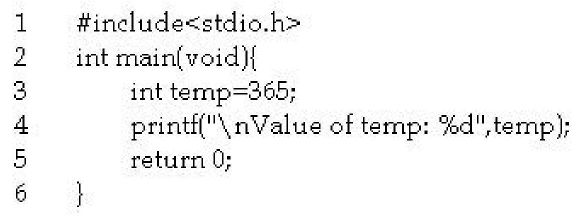
The coprocessor functionality was tested successfully using the programs for basic floating point arithmetic operations including those to multiply two matrices, to round of a real number and to calculate sine of an angle.

All these programs were debugged using GDB by creating the simulator instance for each of them for both PowerPC and MIPS machines. The GDB commands were used to perform operations on the call stack, access values of the variables and registers, display disassembly of the program and use breakpoint mechanism.

6.2 Results

The programs were compiled using GCC cross compilers for the PowerPC and MIPS machines. Our enhanced GDB for FSim was used to debug these programs. In this section we show results of debugging a small C program for both these processors using our GDB. We use various GDB commands to debug the program and compare the output of GDB for both architectures.

When we want to debug programs on a simulator using GDB, we need to give two commands before starting the debugging. These commands are *remote sim* and *load*. For an usual simulator the *remote sim* command is used to connect GDB to simulator and the *load* command is used to load the target program in the simulator. In our implementation we use GDB for retargetable simulator FSim. In our implementation the *remote sim* command performs stage-I of fsimg (described in section 2.2.1). The implementation of *load* command is used to create the simulator instance in the

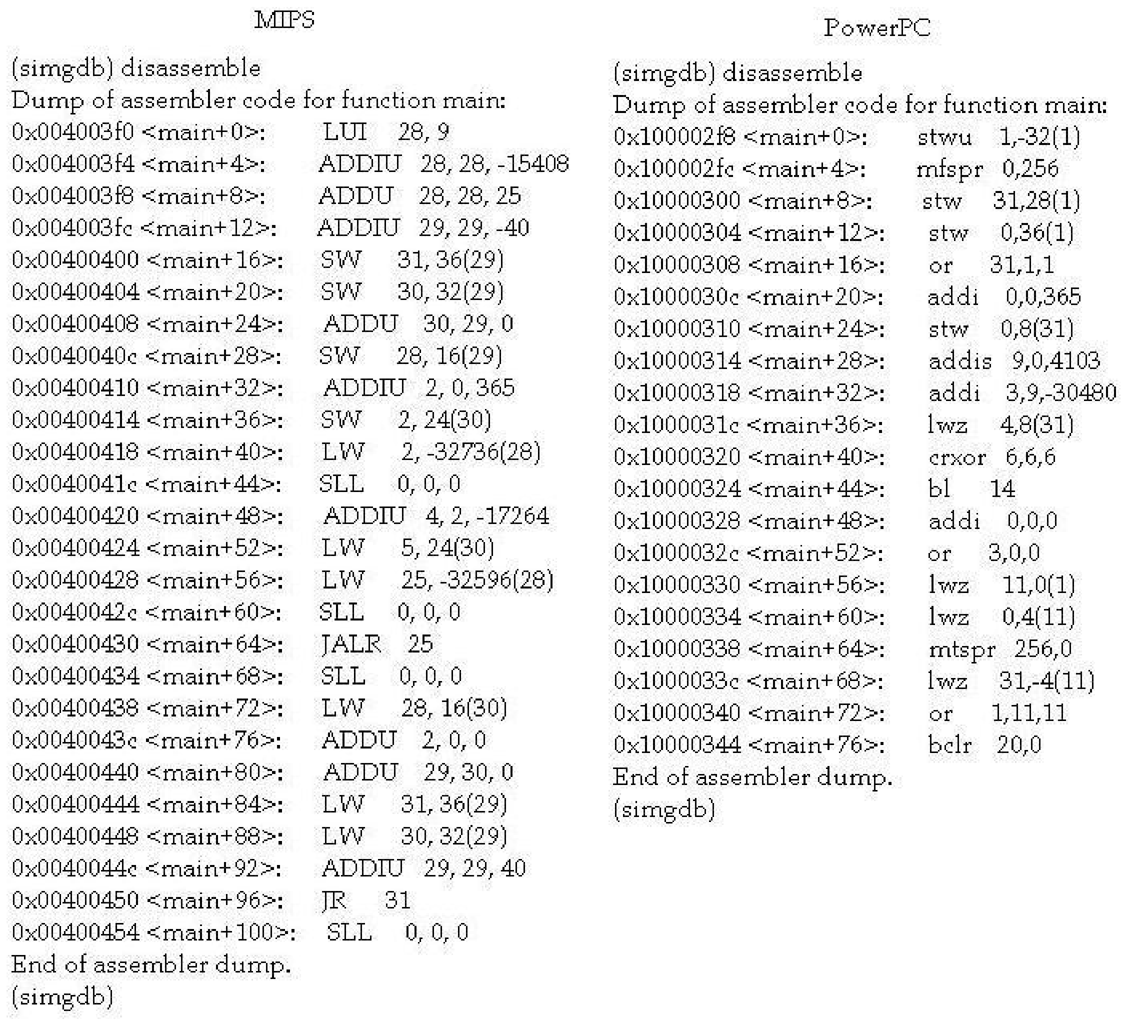


```
#include<stdio.h>
int main(void){
      int temp=365;
      printf("\nValue of temp: %d",temp);
      return 0;
}
```

Figure 6.1: Sample C source code

form of a shared library. It performs stage-II (described in section 2.2.2) and stage-III (described in section 2.2.3) of simulator generation process. So after these two commands are executed, GDB is ready to process the commands related to the program execution.

The program used is in the file *test.c* as shown in figure 6.1. It initializes a local variable *temp* with value 365 and prints it on the stdout.



MIPS

```
(simgdb) disassemble
Dump of assembler code for function main:
0x004003f0 <main+0>:     LUI    28, 9
0x004003f4 <main+4>:     ADDIU  28, 28, -15408
0x004003f8 <main+8>:     ADDU   28, 28, 25
0x004003fc <main+12>:    ADDIU  29, 29, -40
0x00400400 <main+16>:    SW     31, 36(29)
0x00400404 <main+20>:    SW     30, 32(29)
0x00400408 <main+24>:    ADDU   30, 29, 0
0x0040040c <main+28>:    SW     28, 16(29)
0x00400410 <main+32>:    ADDIU  2, 0, 365
0x00400414 <main+36>:    SW     2, 24(30)
0x00400418 <main+40>:    LW     2, -32736(28)
0x0040041c <main+44>:    SLL    0, 0, 0
0x00400420 <main+48>:    ADDIU  4, 2, -17264
0x00400424 <main+52>:    LW     5, 24(30)
0x00400428 <main+56>:    LW     25, -32596(28)
0x0040042c <main+60>:    SLL    0, 0, 0
0x00400430 <main+64>:    JALR   25
0x00400434 <main+68>:    SLL    0, 0, 0
0x00400438 <main+72>:    LW     28, 16(30)
0x0040043c <main+76>:    ADDU   2, 0, 0
0x00400440 <main+80>:    ADDU   29, 30, 0
0x00400444 <main+84>:    LW     31, 36(29)
0x00400448 <main+88>:    LW     30, 32(29)
0x0040044c <main+92>:    ADDIU  29, 29, 40
0x00400450 <main+96>:    JR     31
0x00400454 <main+100>:   SLL    0, 0, 0
End of assembler dump.
(simgdb)
```

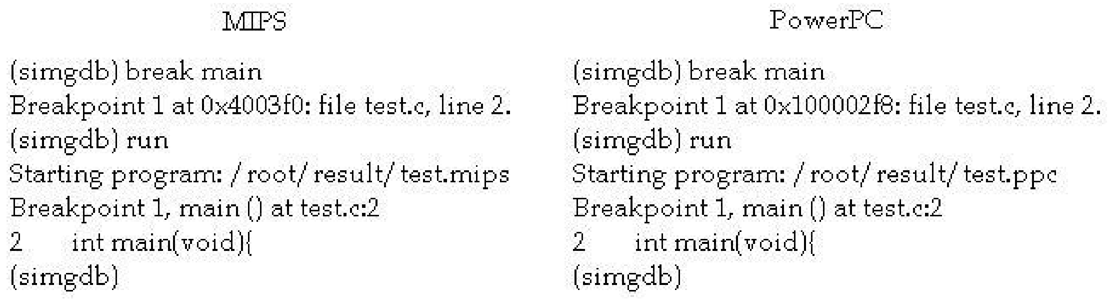
PowerPC

```
(simgdb) disassemble
Dump of assembler code for function main:
0x100002f8 <main+0>:     stwu   1,-32(1)
0x100002fc <main+4>:     mfspr  0,256
0x10000300 <main+8>:     stw    31,28(1)
0x10000304 <main+12>:    stw    0,36(1)
0x10000308 <main+16>:    or     31,1,1
0x1000030c <main+20>:    addi   0,0,365
0x10000310 <main+24>:    stw    0,8(31)
0x10000314 <main+28>:    addis  9,0,4103
0x10000318 <main+32>:    addi   3,9,-30480
0x1000031c <main+36>:    lwz    4,8(31)
0x10000320 <main+40>:    crxor  6,6,6
0x10000324 <main+44>:    bl     14
0x10000328 <main+48>:    addi   0,0,0
0x1000032c <main+52>:    or     3,0,0
0x10000330 <main+56>:    lwz    11,0(1)
0x10000334 <main+60>:    lwz    0,4(11)
0x10000338 <main+64>:    mtspr  256,0
0x1000033c <main+68>:    lwz    31,-4(11)
0x10000340 <main+72>:    or     1,11,11
0x10000344 <main+76>:    bclr   20,0
End of assembler dump.
(simgdb)
```

Figure 6.2: Example of *disassemble* command execution

The output of the *disassemble* command for the binaries for MIPS and PowerPC architectures is shown in the figure 6.2 for the function *main* of *test.c*. As it can be observed, the two architectures have the different disassembly due to different instruction sets. For example, in case of the MIPS machine the call to printf is made by the instruction at address `0x400430` and the instruction used is *jalr* (jump and link register). For the PowerPC machine the instruction is *bl* (branch and link) and its address is `0x10000324`.

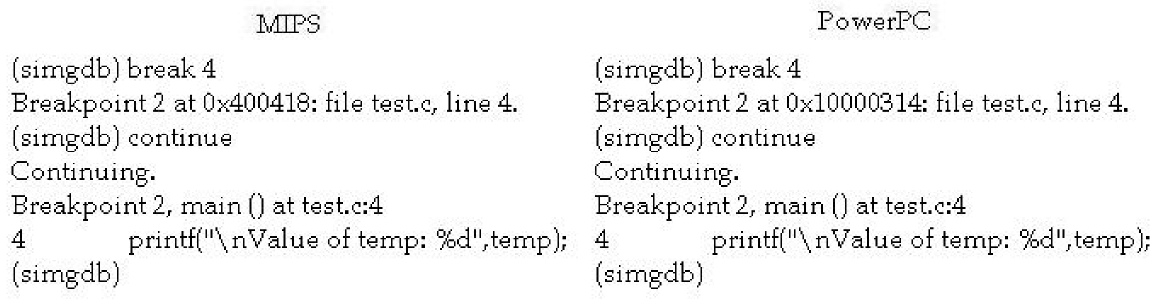
The breakpoint command is used to create a breakpoint. We have set the breakpoints at function *main*. As shown in figure 6.3 starting addresses of function *main* for both architectures are different and this can be observed from the breakpoint addresses returned by breakpoint handler. We use *run* command to start the execution. The program execution was broken at the first breakpoint, i.e. at function *main*.



```
                MIPS                                          PowerPC
(simgdb) break main                           (simgdb) break main
Breakpoint 1 at 0x4003f0: file test.c, line 2.  Breakpoint 1 at 0x100002f8: file test.c, line 2.
(simgdb) run                                  (simgdb) run
Starting program: /root/result/test.mips      Starting program: /root/result/test.ppc
Breakpoint 1, main () at test.c:2             Breakpoint 1, main () at test.c:2
2       int main(void){                       2       int main(void){
(simgdb)                                      (simgdb)
```

Figure 6.3: Example of setting a breakpoint on a function

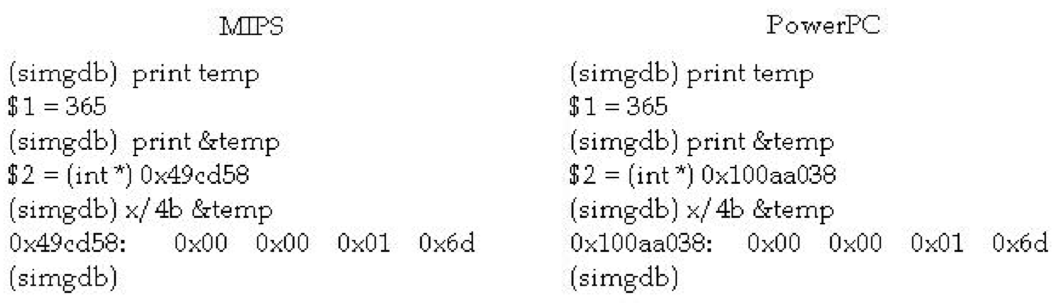
The breakpoint command can also be used with the line number and source file name as parameters. In the output shown in figure 6.4 we specify only the line number to set a breakpoint. As the current frame is from the file *test.c*, breakpoint is set on the specified line number of the same file.



```
                MIPS                                               PowerPC
(simgdb) break 4                                  (simgdb) break 4
Breakpoint 2 at 0x400418: file test.c, line 4.     Breakpoint 2 at 0x10000314: file test.c, line 4.
(simgdb) continue                                 (simgdb) continue
Continuing.                                       Continuing.
Breakpoint 2, main () at test.c:4                 Breakpoint 2, main () at test.c:4
4           printf("\nValue of temp: %d",temp);   4           printf("\nValue of temp: %d",temp);
(simgdb)                                          (simgdb)
```

Figure 6.4: Example of setting a breakpoint on a line in the source file

In the output shown in figure 6.5 we use GDB commands to display the addresses of the variable *temp* and value stored at those addresses. The endianity of the target can be observed from the displayed memory content. Both target processors use big endianity for memory.

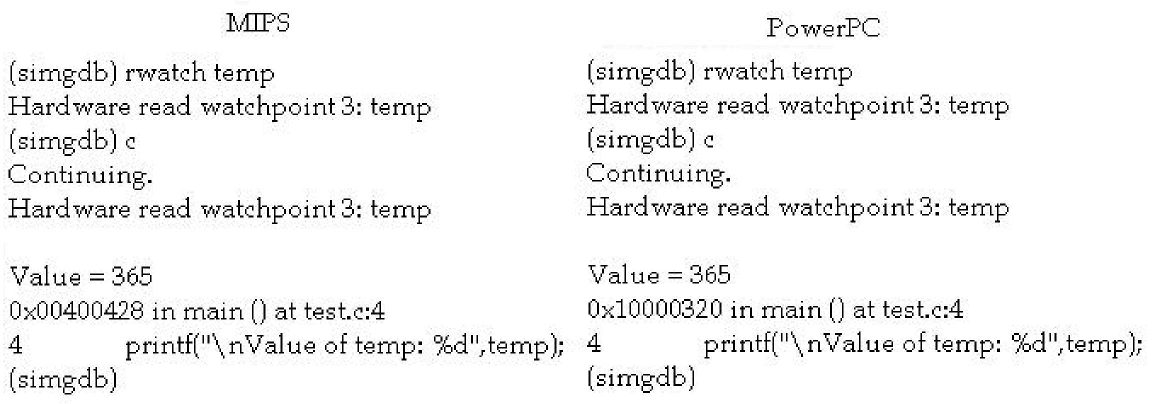


```
MIPS
(simgdb)  print temp
$1 = 365
(simgdb)  print &temp
$2 = (int *) 0x49cd58
(simgdb) x/4b &temp
0x49cd58:     0x00   0x00   0x01   0x6d
(simgdb)
```

```
PowerPC
(simgdb) print temp
$1 = 365
(simgdb) print &temp
$2 = (int *) 0x100aa038
(simgdb) x/4b &temp
0x100aa038:    0x00   0x00   0x01   0x6d
(simgdb)
```

Figure 6.5: Example of accessing a variable

In the output shown in figure 6.6 we show the execution of watchpoint command. In this program none of the variables have their values reassigned, hence we used *rwatch* command to break the execution of the program on reading the value of variable *temp*. The execution is broken after the completion of the read operation. Hence, the program counter points to the instruction next to the one which has caused memory read operation at the watchpoint address. From the disassembly of the programs we can observe that for the MIPS machine the instruction at 0x00400424 and for the PowerPC machine the instruction at 0x1000031c are load instructions. Both these instructions read the value of the variable temp and caused *read watchpoint* to break the execution at the addresses 0x00400428 and 0x10000320 respectively.

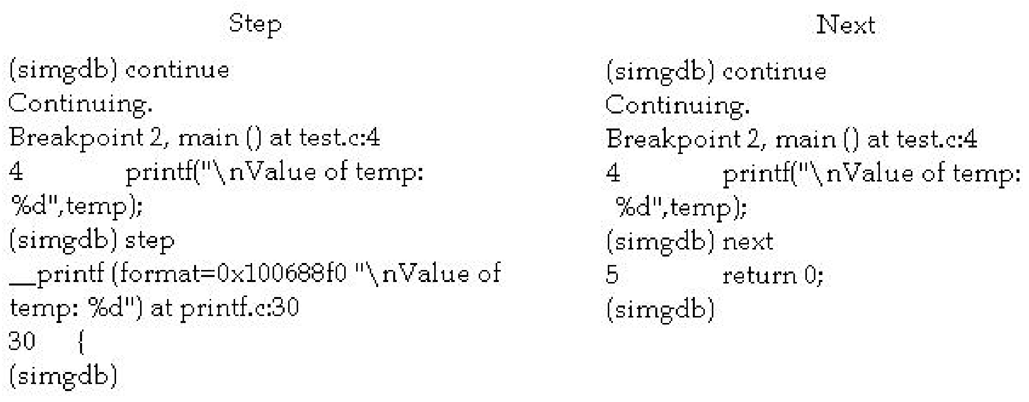


```
MIPS
(simgdb) rwatch temp
Hardware read watchpoint 3: temp
(simgdb) c
Continuing.
Hardware read watchpoint 3: temp

Value = 365
0x00400428 in main () at test.c:4
4          printf("\nValue of temp: %d",temp);
(simgdb)
```

```
PowerPC
(simgdb) rwatch temp
Hardware read watchpoint 3: temp
(simgdb) c
Continuing.
Hardware read watchpoint 3: temp

Value = 365
0x10000320 in main () at test.c:4
4          printf("\nValue of temp: %d",temp);
(simgdb)
```

Figure 6.6: Example of setting a watchpoint

In the output shown in figure 6.7 we show the execution of commands *step* and *next* for the PowerPC architecture. As shown in the figure, GDB steps over the function to execute the *next* command, whereas for the *step* command it stops execution at the first line of the callee function. The callee function here is *printf* and the caller function is *main*.



Step

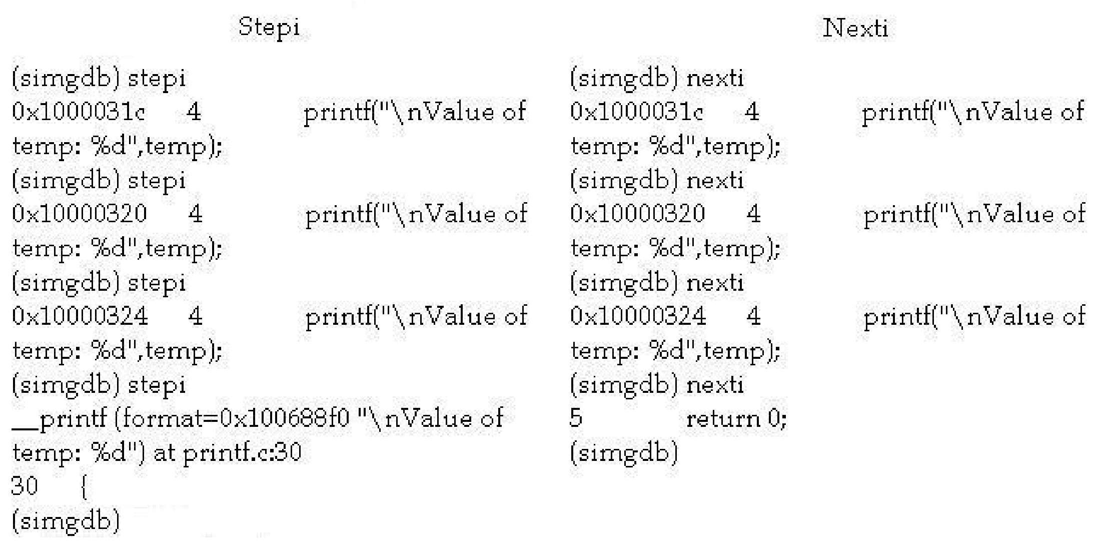
```
(simgdb) continue
Continuing.
Breakpoint 2, main () at test.c:4
4           printf("\nValue of temp:
%d",temp);
(simgdb) step
__printf (format=0x100688f0 "\nValue of
temp: %d") at printf.c:30
30    {
(simgdb)
```

Next

```
(simgdb) continue
Continuing.
Breakpoint 2, main () at test.c:4
4           printf("\nValue of temp:
 %d",temp);
(simgdb) next
5           return 0;
(simgdb)
```

Figure 6.7: Example of *step* and *next* command execution

In the output shown in figure 6.8 we show the execution of commands *stepi* and *nexti* for the PowerPC architecture. Instruction at address `0x10000324` is a call instruction with pneumonic *bl* (branch and link) which can be found from the disassembly of the function *main*. As shown in the figure, GDB steps over the function for the execution of *nexti* command, whereas for the *stepi* command it stops execution at first instruction of the callee function. Again, the callee function here is *printf* and the caller function is *main*. The similar behavior were seen for the MIPS binary programs as well.



Stepi

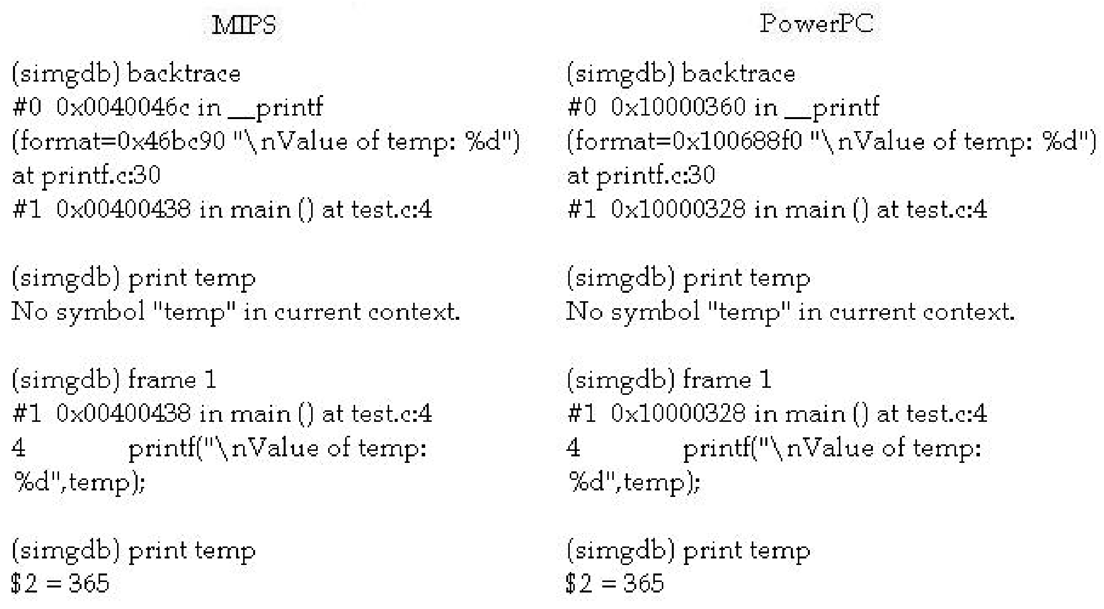
```
(simgdb) stepi
0x1000031c     4          printf("\nValue of
temp: %d",temp);
(simgdb) stepi
0x10000320     4          printf("\nValue of
temp: %d",temp);
(simgdb) stepi
0x10000324     4          printf("\nValue of
temp: %d",temp);
(simgdb) stepi
__printf (format=0x100688f0 "\nValue of
temp: %d") at printf.c:30
30     {
(simgdb)
```

Nexti

```
(simgdb) nexti
0x1000031c     4          printf("\nValue of
temp: %d",temp);
(simgdb) nexti
0x10000320     4          printf("\nValue of
temp: %d",temp);
(simgdb) nexti
0x10000324     4          printf("\nValue of
temp: %d",temp);
(simgdb) nexti
5          return 0;
(simgdb)
```

Figure 6.8: Example of *stepi* and *nexti* command execution

In the output shown in figure 6.9 we show the result of *backtrace* command of GDB for two different architectures. The value of variable *temp* can not be printed from call frame of *printf*. So we select the frame corresponding to the function *main* and display the value.



MIPS

```
(simgdb) backtrace
#0  0x0040046c in __printf
(format=0x46bc90 "\nValue of temp: %d")
at printf.c:30
#1  0x00400438 in main () at test.c:4

(simgdb) print temp
No symbol "temp" in current context.

(simgdb) frame 1
#1  0x00400438 in main () at test.c:4
4          printf("\nValue of temp:
%d",temp);

(simgdb) print temp
$2 = 365
```

PowerPC

```
(simgdb) backtrace
#0  0x10000360 in __printf
(format=0x100688f0 "\nValue of temp: %d")
at printf.c:30
#1  0x10000328 in main () at test.c:4

(simgdb) print temp
No symbol "temp" in current context.

(simgdb) frame 1
#1  0x10000328 in main () at test.c:4
4          printf("\nValue of temp:
%d",temp);

(simgdb) print temp
$2 = 365
```

Figure 6.9: Example of execution of *backtrace* command

6.3 Conclusions and future work

In this thesis we developed a debugger support for the retargetable simulator FSim. The GDB can be used to debug the programs written and compiled for multiple simulator targets. We have used the existing simulator interface of GDB to connect to the FSim simulator. The newly designed interface to update the target architecture of GDB from the target architecture of a simulator is used successfully with the FSim simulator. In our approach we update the target definition of GDB from processor specifications in Sim-nML language. We have successfully implemented the target architecture functionality of GDB in the FSim simulator.

We have tested GDB and FSim for two architecture descriptions, MIPS and PowerPC. The functional simulator generator (FSimg) along with GDB can be used to speedup the development of processor descriptions and of their simulators. This debugger can be used to study and to help mapping of operating system calls made by the target architecture to the host architecture in the simulator.

The fsim interface designed and developed as part of this thesis is used to update the target architecture definition of the GDB from the Sim-nML specification and is used to debug the retargetable simulator FSim. In future, this interface can be used by GDB along with other simulator interfaces to provide debugging support to other retargetable simulators.

Bibliography

[1] ARM Processor Overview. Website. http://www.arm.com/products/CPUs/.

[2] CoWare Processor Designer Environment. Website. http://www.coware.com/products/processordesigner.php.

[3] GNU Debugger Project. Website. http://www.gnu.org/software/gdb/.

[4] Hitachi H8/3297 Series Hardware Manual. Website. http://moss.csc.ncsu.edu/ mueller/rt/mindstorm/h3314.pdf.

[5] IEEE Std. 1076-1993: IEEE Standard VHDL Language Reference Manual. Website. http://www.eda.org/rassp/vhdl/standards/standards.html.

[6] IEEE Std. 1364-1995: IEEE Standard Verilog Hardware Description Language. Website. http://inst.eecs.berkeley.edu/ cs150/ProtectedDocs/verilog-ieee.pdf.

[7] IEEE Std. 1666-2005: IEEE Standard System C Language Reference Manual. Website. http://standards.ieee.org/getieee/1666/download/1666-2005.pdf.

[8] Motorola 68HC11 Family of Microcontrollers. Website. http://www.hc11.demon.nl/thrsim11/68hc11/.

[9] Tensilica: Configurable and Standard Processor Cores for SOC Design . Website. http://www.tensilica.com.

[10] Trimaran: An Integrated Compilation and Performance Monitoring Infrastructure. Website. http://www.trimaran.org.

[11] Xilinx Virtex-5 Multi-Platform FPGA . Website. http://www.xilinx.com/support/documentation/virtex-5.htm.

[12] IBM PowerPC 603 RISC Microprocessor User's Manual. Website, 1994. http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC.

[13] The MDES User Manual. HP Lab Technical Report, 1997.

[14] MIPS32 Architecture for Programmers. Website, March 2001. http://www.mips.com/content/Documentation/MIPSDocumentation.

[15] Rajiv A.R. Retargetable Profiling Tools and Their Application in Cache Simulation and Code Instrumentation. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, December 1999.

[16] R. M. Balzer. An Overview of The ISPL Computer Systems Design. *Commun. ACM*, 16(2):117–122, 1973.

[17] Mario R. Barbacci. Instruction Set Processor Specifications for Simulation, Evaluation, and Synthesis. In *DAC '79: Proceedings of the 16th Conference on Design automation*, pages 64–72, Piscataway, NJ, USA, 1979. IEEE Press.

[18] Soubhik Bhattacharya. Generation of GCC Backend from Sim-nML Processor Description. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, July 2001.

[19] Surendra Bishnoi. Functional Simulation Using Sim-nML. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, May 2006.

[20] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin - Madison, 1996.

[21] Bhupesh Chandra. Memory Profiler for FSim Simulator. B.Tech. Project Report, Department of Computer Science and Engineering, IIT Kanpur, May 2007.

[22] Y. Subhash Chandra. Retargetable Functional Simulator. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, June 1999.

[23] Nitin Kumar Dahra. Disassembly and Parsing Support for Retargetable Tools Using Sim-nML. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, May 2007.

[24] Markus Freericks. The nML Machine Description Formalism. TU Berlin Computer Science Technical Report - Updated and Revised Version 1.5 (Draft), June 1993.

[25] GNU C Library. Website. http://www.gnu.org/software/libc/.

[26] M. M. Gourary, S. G. Rusakov, S. L. Ulyanov, M. M. Zharov, and B. J. Mulvaney. A New Simulation Technique for Periodic Small-Signal Analysis. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 30–44, Washington, DC, USA, 2003. IEEE Computer Society.

[27] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 299–302, New York, NY, USA, 1997. ACM Press.

[28] W. Hahn. Computer Design Language - Version Munich (CDLM) A Modern Multi-Level Language. *20th Conference on Design Automation*, pages 4–11, June 1983.

[29] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM Press.

[30] Intel Corporation. Intel Standard Hex File Format Specification, Revision A, Jan 1988.

[31] Nihal Chand Jain. Disassembler using High Level Processor Models. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, Jan 1999.

[32] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of Interpretive and Compiled Instruction Set Simulators. *IEEE ASP-DAC*, pages 339–340, 1999.

[33] Masud M. and Sait S. M. Universal AHPL-A Language for VLSI Design Automation. *IEEE Circuits and Devices Magazine*, pages 8–14, September 1986.

[34] N. Bansal M. Reshadi, P. Mishra and N. Dutt. ReXSim: A Retargetable Framework for Instruction-Set Architecture Simulation. Technical Report CECS-03-05, University of California, Irvine, 2003.

[35] Pierre G. Paulin and Miguel Santana. FlexWare: A Retargetable, Embedded-Software Development Environment. *IEEE Design and Test of Computers*, 19(4):59–69, 2002.

[36] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable Compiled Simulation of Embedded Processors Using a Machine Description language. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):815–834, 2000.

[37] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. *Proceedings. Twelfth International Conference On VLSI Design*, pages 132–137, Jan 1999.

[38] Eric C. Schnarr, Mark D. Hill, and James R. Larus. Facile: A Language and Compiler for High-performance Processor Simulators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 321–331, New York, NY, USA, 2001. ACM Press.

[39] Hemant Shinde. Sim-nML To C Conversion. CS-697 Report, Department of Computer Science and Engineering, IIT Kanpur, May 2007.

[40] Chuck Siska. A processor desription language supporting retargetable multi-pipeline dsp program development tools. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 31–36, Washington, DC, USA, 1998. IEEE Computer Society.

[41] The Native POSIX Thread Library for Linux. Website. http://people.redhat.com/drepper/nptl-design.pdf.

[42] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification. Website, May 1995. http://x86.ddj.com/ftp/manuals/tools/elf.pdf.