Retargetable Functional Simulator

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
Y Subhash Chandra

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

June, 1999

Certificate

This is to certify that the work contained in the thesis entitled "*Retargetable Functional Simulator*", by *Y Subhash Chandra*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

June, 1999

(Dr. Rajat Moona)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

The design of modern embedded systems require automated modeling tools for faster design and for the study of various design tradeoffs. Such tools put together constitute an integrated environment where the designer can write the high level design specifications in a language and use these tools for automatic generation of system specific tools. In this work we have designed a *Retargetable Functional Simulator (Fsimg)* for our integrated environment where the Sim-nML language is used as a base language for writing processor models. Sim-nML is an extension of nML machine description formalism and is powerful enough to describe a processor at instruction level.

The Fsimg generates a processor specific function simulator using the processor models written in Sim-nML. The generated functional simulator helps in the study of functional correctness of the design. It can also produce the instruction trace which can be used by the other tools in studying other aspects of the design. As a part of this work we have specified PowerPC 603 processor in Sim-nML. This specification includes most of user level instructions present in PowerPC 603 with pipeline, and branch prediction. We have also developed a *Macro Preprocessor (nMP)* for processing Sim-nML macros. This macro preprocessor converts the Sim-nML macros to m4 macros adding the flexibility that is not provided by the Sim-nML macros.

Acknowledgements

I am grateful to my guide, Dr. Rajat Moona, who helped me at every stage of this work. His suggestions and innovative ideas helped me a lot in completing this work. I am also grateful to Dr. Deepak Gupta and Dr. Sanjeev Aggarwal for their valuable suggestions and comments during thesis discussions. I would also like to thank other faculty members of CSE department for gaining better knowledge in different fields of computer science.

This work has been done as a part of ongoing research at Cadence Research Center at IIT Kanpur. I express my gratitude to Cadence India Ltd. for their support through fellowship.

I would like to thank my seniors Atul Bhai and Kshitiz Bhai for their help throughout my stay here. I would also like thank other members of Cadence Research Center, specially V. Rajesh and Rajiv. I am greatly indebted to all friends Sridhar, Srikar, Vasu, Prasad, Uma, Gopi, Girish, Prasanna, Nihal, Kousik, Pradip, Kapil, Manoj, Atul, well the list goes on. I would also like to thank my B.Tech friends Chandu and Naveen for their encouragement and support.

Finally I would like to thank my parents and my brother's family for their support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the design of embedded systems the use of automated modeling tools is gaining momentum. They yield fast turn-around time for the system design and simplify the process of design changes. In the past most such tools were system specific. However, with ever increasing complexity of systems and special purpose processors, a strong need is being felt for generic and modular tools. Such tools replace the system or processor specific tools and provide a generic integrated environment. Also these tools help in studying the impact of various hardware-software co-design trade-offs.

In this thesis we focus on tools that deal with the machine language of processors, such as assembler, disassembler, instruction set simulator etc. For developers of these tools, it is convenient to model the processor specific part separately and generic part separately. This is typically achieved by having processor models which tools can understand and configure themselves for a specific processor. Moreover, it is desirable to have a single processor model for all the tools. In this work we have designed a **Retargetable Functional Simulator (Fsimg)** using **Sim-nML**[17] language which is primarily an extension of the **nML**[8] language for processor modeling. **Sim-nML** is simple and powerful enough to specify a complex processor architecture with pipeline and timing parameters.

Fsimg takes the specification of the processor in the *intermediate representation*[1][10] and an executable for the processor in **ELF**[18] format and generates a *Functional simulator (Fsim)* which in turn gives the functional behaviour of the processor model for the given program. Optionally, it can also give the instruction trace

[1]From here onwards we use IR for intermediate representation.

of the program.

1.1 Motivation

Processor models are extensively used in the system design process. The system design process starts with an application and its implementation. Then the model is tested for its performance and other aspects. Designers refine the design after analysis of the performance study to tune the system according to needs. In such a scenario, an integrated environment is required for the designer where several tools exist like simulator, assembler, compiler etc. Rewriting the tools after each design change is a tedious job. Hence automatic generation of these tools is more desirable according to the design changes.

In this thesis we discuss such an integrated environment where **Sim-nML** is the base language for writing processor models. In embedded systems design it is necessary to study for the functional correctness of the model specified. This is the motivating force behind our design of *Retargetable Functional Simulator* which automatically generates a *Functional Simulator*. The *Functional Simulator* can also be used for the generation of instruction trace which can be used by several other tools for the study of various aspects of the system design.

1.2 Overview of Related Work

Using automated tools in the system design process has been a long practice. There are several functional simulation tools available today. Here we will look at some of these tools and some languages for processor modeling.

Instruction Set Description Language (ISDL)[9] is a machine description language which is similar to Sim-nML. ISDL provides constructs for specifying instruction set and other architectural features. A description in ISDL contains the machine word format used for the instruction assembly, semantics of the instruction, and constraints such as the valid combination of operations which is useful for tools like assembler to generate correct code. All theses are captured in a separate sections. Currently an automatic assembler generator has been developed.

Specification language for encoding and decoding (SLED)[15] is a language for describing the abstract, binary, and assembly-language representations for machine instructions. Using SLED a toolkit called New Jersey Machine-Code has been developed which generates bit-manipulating code for use in applications that process machine code. Programmers can write such applications at an assembly level of abstraction, and the toolkit enables the applications to recognize and emit the binary representation used by the hardware. SLED is suitable for CISC and RISC type of machines. SLED deals with the instruction representation only, but not with any other architectural details. Some tools like retargetable debugger, retargetable optimizing linker have been implemented.

SimOS[13] is a machine simulation environment designed to study large complex computer systems. SimOS simulates the computer hardware in sufficient detail and speed to run existing system software and application programs (figure 1.1). It is

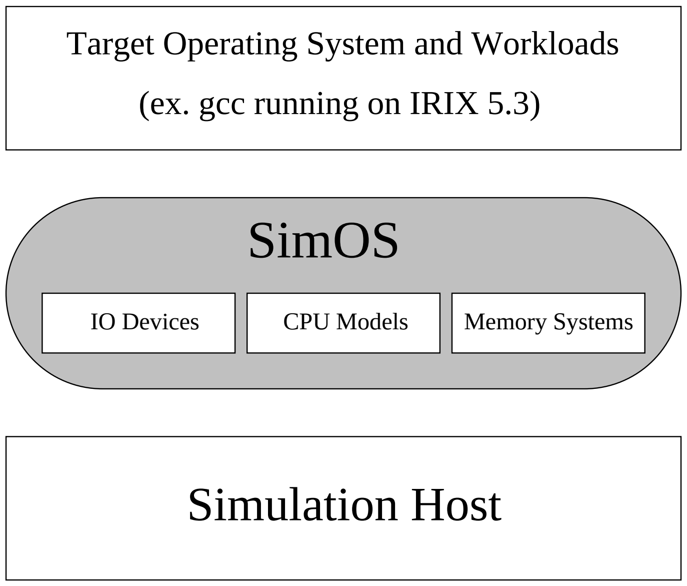
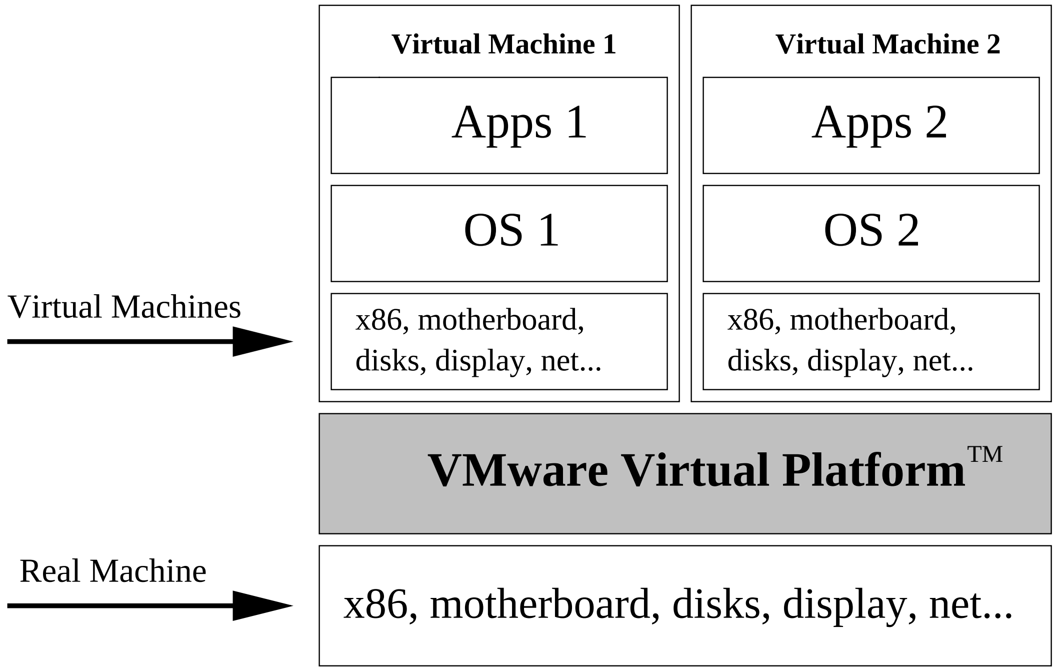

Target Operating System and Workloads
(ex. gcc running on IRIX 5.3)
SimOS
IO Devices
CPU Models
Memory Systems
Simulation Host

Figure 1.1: A SimOS System

useful to evaluate the impact of new hardware designs on the performance of the simulated hardware components. Operating system programmers can develop their software in an environment that provides the same interface as the target hardware, while taking the advantage of the system visibility and repeatability offered by a simulation environment. Currently it is able to model MIPS R4000, MIPS R10000, and Digital Alpha processor families. Some existing operating systems IRIX v5.3, IRIX v6.4, and Digital Unix are ported to SimOS environment. Work is going on a port of Linux for the Alpha.

VMware[5] is one of the new products which is similar to SimOS. It creates a virtual platform on the host operating system and allows more than one other operating systems to boot and run concurrently on it. It accomplishes this by transparently multiplexing all hardware resources into multiple virtual machines, each resembling the underlying machine. The virtual platform gives the full functional behaviour of the targeted hardware (figure 1.2). The overhead due to the virtualization is over



Virtual Machine 1
Virtual Machine 2
Apps 1
Apps 2
OS 1
OS 2
Virtual Machines
x86, motherboard, disks, display, net...
x86, motherboard, disks, display, net...
VMware Virtual Platform™
Real Machine
x86, motherboard, disks, display, net...

Figure 1.2: VMware Virtual Platform

come by it's *dual-mode personality*. The virtual machine platform run as a virtual monitor directly on hardware and a normal application running on top of the host operating system. For performance *monitor personality* is used and for device independent portions *application personality* is used. Currently it supports x86 machine as target hardware and is available for Linux and Windows NT.

vMac[4] is a Macintosh emulator which emulates Motorola 68000 based Apple Macintosh plus.

Boucs[1] is also another emulator, which emulates x86 instruction set, related AT hardware components and BIOS to boot and run various operating systems. Currently it emulates full x386 and x486 user level instructions. Supports real/virtual addressing modes, VGA color graphics, floppy and hard drive etc.

There are other small emulators which basically emulate very small environments like Sega, Nintendo game entertainment system etc. **Snes9x**[2] is an emulator for

Super Nintendo Entertainment System. It emulates 65c816 main CPU, Sony SPC700 sound CPU, DMA channels, some IRQ channels and a few other things.

All these tools try to give the hardware abstraction to the applications. The level of abstraction vary from each other depending on what kind of applications they are trying to support.

1.3 Goals Achieved

In this work we aimed at the development of integrated environment for processor performance modeling using **Sim-nML**. The development of complete environment is in progress. Many tools have been developed till now, which we will look at in Chapter 2. The goals achieved in this work are as follows.

- **PowerPC 603 Specification in Sim-nML**
  As a starting point model for **PowerPC 603**[19] processor (chapter 3) has been developed in **Sim-nML**. Around 237 instructions have been specified with resource usage model and pipeline.

- **Macro Preprocessor for Sim-nML**
  Sim-nML provides macros which help in writing the specification. A *Macro Preprocessor* has been designed and implemented.

- **Retargetable Functional Simulator**
  It has been designed and implemented which takes processor specification and a executable and generates a *Functional Simulator*.

1.4 Organization of Report

The rest of the thesis is organized as follows. In chapter 2 we discuss briefly the integrated environment. In chapter 3 we look at **PowerPC 603** architecture and its specification in **Sim-nML**. In chapter 4 we look at some aspects of *Functional Simulation* and *Trace Generation*. In chapter 5 and chapter 6 we discuss the design and implementation of *Macro Preprocessor* and *Fsimg*. Finally we conclude with the results and future work in chapter 7. In Appendix A and B, we provide the format of *ir* and the user manuals for tools developed in this thesis respectively.

Chapter 2

Overview of Integrated Environment

In this chapter we discuss the integrated environment that we work in, the Sim-nML Language, the IR, and some existing tools.

2.1 Overall Structure

The base language for our environment is Sim-nML, an extension of nML machine description formalism. Processor models are written in Sim-nML, using which, various processor specific tools can be generated automatically. To make the tools' design easy the model specified in Sim-nML is first converted into an *intermediate representation* (IR) (section 2.4). For a tool, intermediate form is simpler and very easier to read and interpret when compared to a specification in Sim-nML. A tool called irg[10] is available that takes a Sim-nML specification and converts it to IR. The overall view of the environment is shown in the figure 2.1.

2.2 Sim-nML Language

nML[8] is an extensible formalism to describe a processor architecture. nML works at an abstract level hiding implementation details of the architecture. In nML the architecture is described at instruction level. The instruction set is enumerated as an attribute grammar in a tree hierarchy capturing the semantics of the instructions at

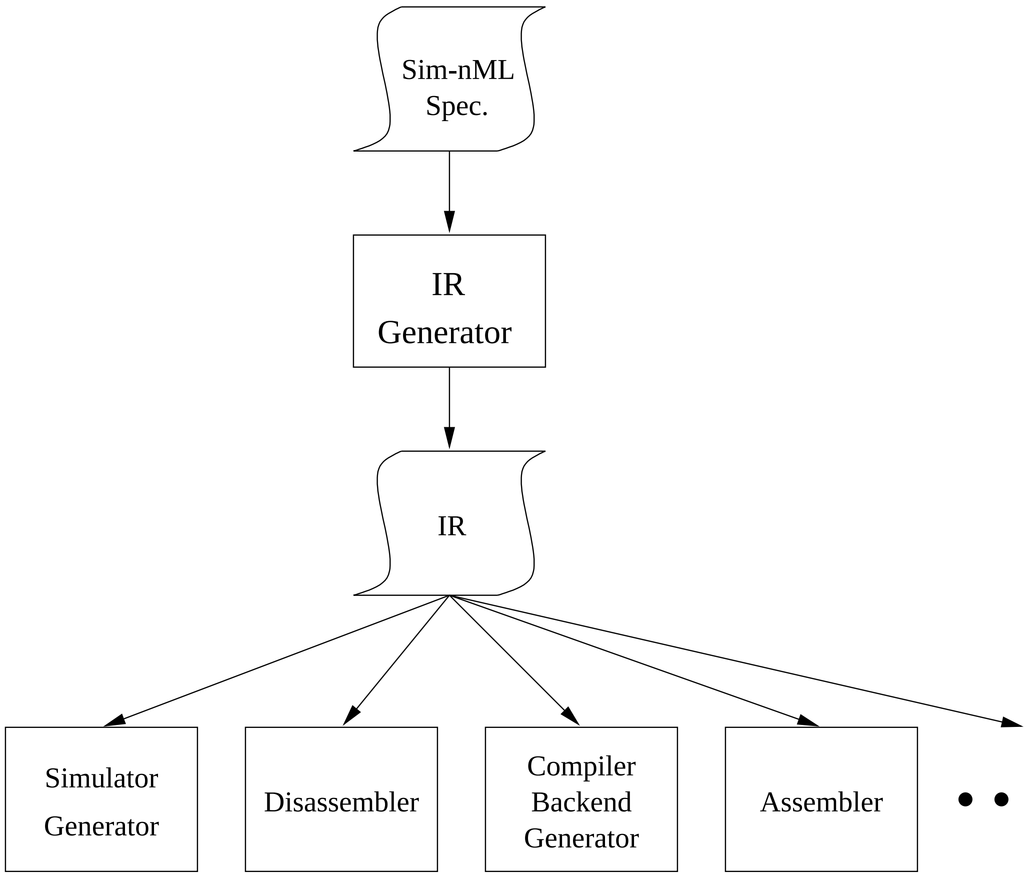


Figure 2.1: A View of Integrated Environment

different levels of the hierarchy depending on the class of instructions.

nML defines a fixed start symbol called instruction and two kinds of productions or-rule, whose syntax is as follows:

op $n_0 = n_1 \mid n_2 \mid n_3 \mid ....$

and and-rule whose syntax is as follows

op n_0 ($p_1 : t_1, p_2 : t_2, ...$)
$a_1 = e_1\ a_2 = e_2\ ...$

where each n_i is a non-terminal and each t_i is a terminal. Each a_i is a attribute and e_i is it's corresponding definition. p_is are the parameters used in the attribute definitions. We can specify the addressing modes using mode-rule which is similar to and-rule. A simple mode-rule looks like following

```
mode REG_INDIRECT ( i : card ( 5 ) ) = R [ i ]
```

Above mode-rule defines a register indirect mode. nML grammar predefines three attributes namely *syntax* which is textual syntax of the instruction, *image* which is binary coding of the instruction and *action* which is the semantics of the instruction. Following is a nML description for simple processor with two instructions add and sub.

```
mem AC [ 1 , card ( 16 ) ]
mem PC [ 1 , card ( 16 ) ]
mem REG [ 4 , card ( 16 ) ]

mode REG_INDIRECT ( i : card ( 2 ) ) = REG [ i ]
   syntax = format ( "%d", i )
   image = format ( "%2b", i )

op instruction ( x : instruction_action )
   syntax = format ( "%s", x.syntax )
   image = format ( "%s", x.image )
   action = {
                 PC = PC + 1;
                 x.action;
             }

op instruction_action = add | sub

op add ( x : REG_INDIRECT )
   syntax = format ( "add %s", x.syntax )
   image = format ( "100000%s", x.image )
   action = { AC = AC + x; }

op sub ( x : REG_INDIRECT )
   syntax = format ( "sub %s", x.syntax )
```

```
        image = **format** ( "100001%s", x.image )
        action = { AC = AC + x; }
```

The add instruction adds the contents of a register to register AC and stores the result in the AC. Similarly the sub instruction does the subtraction. PC is the program counter.

nML allows type declarations, constant declarations and macros which aid in writing clear specification. nML lacks in the control constructs and ability to specify inter-instruction dependencies. Moreover, specification of timing of operations is not possible. Therefore nML is not very useful for performance evaluation. It is difficult to specify and study a newer architecture having features like pipelines, out-of-order execution, branch prediction etc. Sim-nML[17] is an extension of nML which includes the timing of various operations and a resource usage model. The main idea behind the resource usage model is that, as an instruction executes it holds a set of resources like functional units, registers etc. Capturing this model helps in the study of the performance of the processor.

In Sim-nML resources are declared with the *resource* construct which looks like following.

```
    resource fetch_unit, execution_unit, retire_unit
```

A new attribute *uses* has been added which describes the resource usage pattern and timing parameters[1] for the instruction. The specification for the previous example with the resource use model looks like the following.

```
    reg AC [ 1 , card ( 16 ) ]
    reg PC [ 1 , card ( 16 ) ]
    reg REG [ 4 , card ( 16 ) ]

    mode REG_INDIRECT ( i : card ( 2 ) ) = REG [ i ]
        syntax = format ( "%d", i )
        image = format ( "%2b", i )

    resource fetch_unit, execution_unit, retire_unit
```

[1]Refer [17] for more detailed information.

```
op instruction ( x : instruction_action )
    uses = fetch_unit : preact #{1}, x.uses, retire_unit #{1} : action
    syntax = format ( "%s", x.syntax )
    image = format ( "%s", x.image )
    preact = { PC = PC + 1; }
    action = { x.action; }

op instruction_action = add | sub

op add ( x : REG_INDIRECT )
    uses = execution_unit & AC #{1}
    syntax = format ( "add %s", x.syntax )
    image = format ( "100000%s", x.image )
    action = { AC = AC + x; }

op sub ( x : REG_INDIRECT )
    uses = execution_unit & AC #{1}
    syntax = format ( "sub %s", x.syntax )
    image = format ( "100001%s", x.image )
    action = { AC = AC + x; }
```

The above specification says that all instructions use the *fetch_unit* for one time unit, the *execution_unit* for the time specified by the instruction and the *retire_unit* for one time unit. The add and sub instructions use the *execution_unit* for one time unit. The *action* attribute at the end of the *uses* definition specifies that after using the given resources for the mentioned duration of time, the specified function has to be performed. Sim-nML gives more constructs like declaration of exceptions[2] etc. which help in describing branch prediction and out-of-order execution.

2.3 Macro Preprocessor

nML provides macros to aid the specification writing. A typical macro definition looks

[2]Refer [17] for more information.

like,

macro name (p_1, p_2, ...) = *expr*

where p_i is a parameter and *expr* is an nML expression. The above syntax is not very flexible because the macro body allows only one nML expression. We have relaxed this restriction so that macros can be used extensively. The new syntax looks like,

macro name (p_1, p_2, ...) = macro-body

where macro-body is sequence of characters ending with a new line. If the macro definition needs to span multiple lines then each line should end with a reverse slash(\) except the last line which should end with a new line. For this purpose we have designed and implemented a macro preprocessor **nMP**[3]. The Sim-nML description containing macros is given as input to nMP which translates the macros into **m4**[16] macros. m4[4] is a generic macro processor in traditional Unix systems. It is available on most platforms. The output from m4 is Sim-nML description without macros. This is given as input to irg. nMP does not do any syntax checking on the macro-body since it is done by irg[10]. This macro processing phase appear in between Sim-nML Spec. and IR generator in the figure 2.1. We will discuss its design and implementation in section 5.1.

2.4 Intermediate Representation

Specification written in Sim-nML contains many constructs for the clarity and understanding. Tools using this specification need to parse and interpret the contents. Such an effort can be avoided if parsed Sim-nML specification is represented in an intermediate form that is easily understand by a program. IRG[10] converts Sim-nML specification into an IR.

[3]Refer Section 5.1

[4]man m4 or info m4 gives detailed information on using and writing macros in m4.

2.4.1 Structure of IR

The information in the Sim-nML specification is captured into a set of tables in IR. Each table consists of fixed or variable size records representing a particular type of information. For example all the *mem* type declarations i.e, variables are put into MEMORY TABLE. Each record in this table give the information such as type, size etc., of a variable. In the remaining tables a variable is referred by its index in the MEMORY TABLE. In this way it is easy to extract out the information needed. Finally to figure out the number of tables, no of records in each table and their location inside the IR, a special table is added called META TABLE, whose records provide information about the other tables in the IR. META TABLE is always the first table in the IR while other tables can be any where inside the IR. Information about various tables and their structures can be found in Appendix A.

2.5 Existing Tools

Following tools have been implemented till now in our environment.

Instruction Set Simulator Generator [17] takes Sim-nML specification and generates a performance simulator, which in turn takes a binary for that processor and gives the performance based results.

Disassembler [10] takes Sim-nML specification and a binary in **ELF** format and gives out the disassembly of the binary.

Compiler Back-End Generator [14] takes nML specification and generates a LCC machine description which can used to generate a LCC compiler for the specified processor.

Chapter 3

PowerPC 603 Specification in Sim-nML

In this chapter we present a brief overview of *PowerPC 603* architecture and discuss the *PowerPC 603* specifications in Sim-nML, and some issues in writing specifications in general.

3.1 Overview of Architecture

PowerPC 603[19] is a 32-bit implementation of *PowerPC architecture*[7] which consists of following components.

- **PowerPC user instruction set architecture** - This includes the base user-level instruction set, user-level registers, programming model, data types, and addressing modes.
- **PowerPC virtual environment architecture** - This describes the memory model that can be assumed by software processes and includes descriptions of the cache model, cache-control instructions, address aliasing, and other related issues.
- **PowerPC operating environment architecture** - This includes the structure of the memory management model, supervisor-level registers, and the exception model.

The 32-bit portion of PowerPC architecture provides 32-bit effective address, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits. The PowerPC 603 is a super-scalar processor capable of issuing and retiring as many as three instruction per clock. Instructions can execute out of order for increased performance; however, PowerPC 603 makes completion appear sequential.

▮ *Execution Units*

The PowerPC 603 consists of five execution units - an integer unit (IU), a floating-point unit (FPU), a branch processing unit (BPU), a load/store unit (LSU), and a system register unit (SRU) (figure 3.1). The PowerPC 603 has the ability to execute five instructions in parallel. Most of the integer instructions execute in one clock cycle. The FPU is pipelined such that a single-precision multiply-add instruction can be issued every clock cycle. It provides two independent on-chip 8-Kbyte, two-way set-associative, physically addressed caches for instructions and data and on-chip instruction and data memory management units (MMUs). MMUs contain translation look aside buffers (TLBs) for virtual memory support.

The IU contains a fetch unit, instruction queue, dispatch unit, and BPU provides the centralized control of instruction flow to the execution units. The IU determines the address of the next instruction to be fetched based on information from the sequential fetcher and from the BPU. The IU fetches the instructions from the instruction cache and places them in the instruction queue. The BPU extracts branch instructions from the fetcher and uses static branch prediction on an unresolved conditional branch to enable IU continue fetching instructions from predicted target while the conditional branch is evaluated. The instruction queue holds as many as six instructions and loads up to two instructions from IU in a single cycle. The instructions are dispatched to their respective execution units from dispatch unit at a maximum rate of two instructions per cycle. The completion unit tracks instructions from dispatch through execution, and then retires or completes them in program order. For this purpose a first in first out (FIFO) queue of five completion buffers is used. At the time of dispatching an instruction a completion buffer is allocated to that instruction. If no buffer is available then the instruction dispatch stalls until a completion buffer is available along with other resources needed for the dispatch. A maximum of two instructions per cycle are completed in order from the queue.

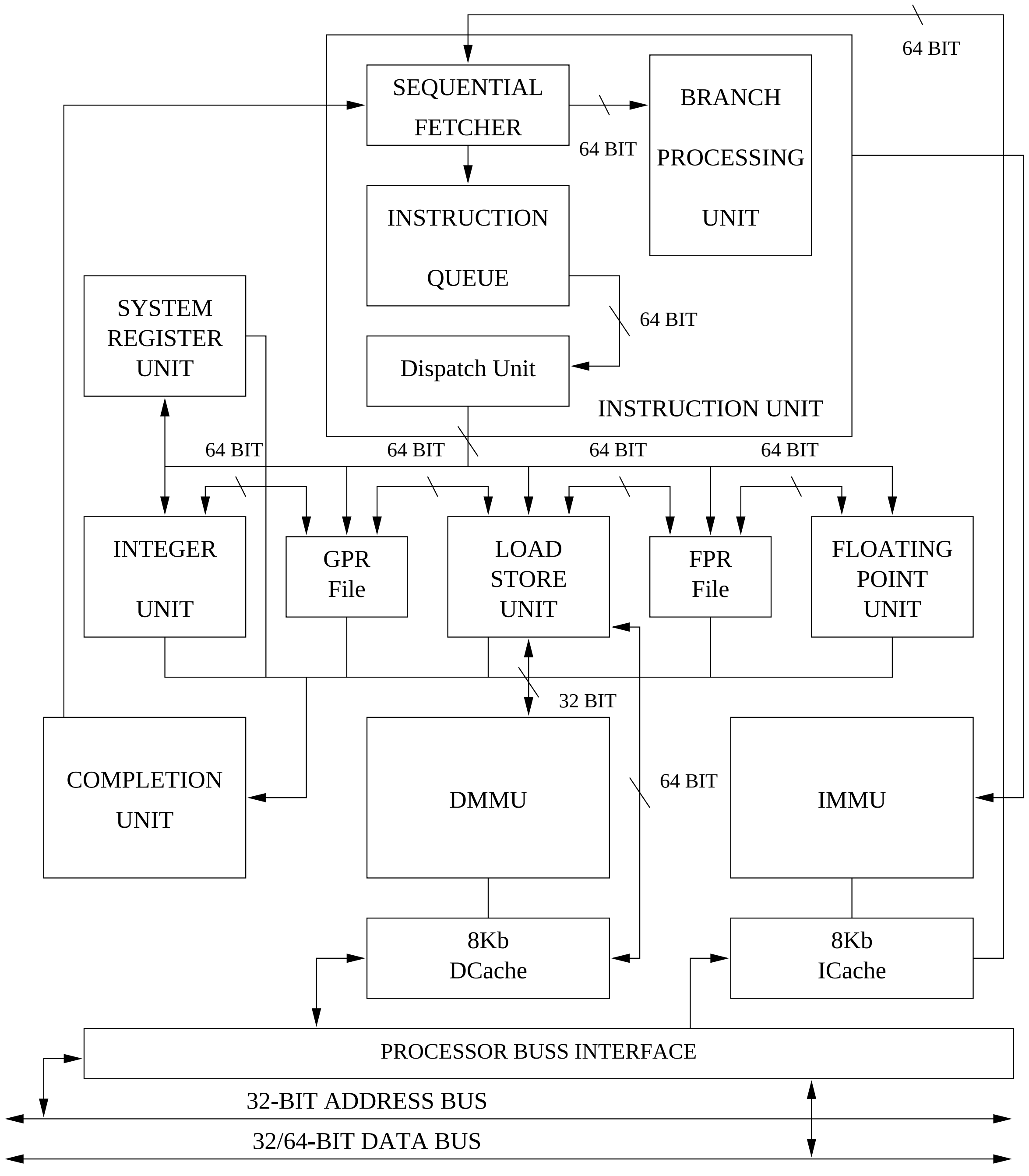


Figure 3.1: PowerPC 603 Microprocessor Block Diagram

▮ *Registers*

The PowerPC 603 contains 32 user-level, general-purpose registers (GPRs). These registers are 32-bit wide. The GPRs serve as the data source or destination for all integer instructions. The PowePC 603 also contains 32 user-level, 64-bit floating-point registers (FPRs). These serve as the data source or destination for floating-point instructions. FPRs can contain data objects in either single or double precision floating-point formats. Apart for the GPRs and FPRs, PowerPC 603 also contains some conditional registers, segment registers and a few special-purpose registers.

▮ *Instruction Set*

All PowerPC 603 instructions are encoded in 32-bits. Various types of instruction formats are consistent, permitting efficient decoding to occur in parallel with operand accesses. The PowerPC 603 instruction set is categorized into the following categories.

- **Integer instructions** - These include computational and logical instructions.
- **Floating-point instructions** - These include floating-point computational instructions.
- **Load/store instructions** - These include integer and floating-point load and store instructions.
- **Flow control instructions** - These include branch instructions and other instructions that affect the instruction flow.
- **Processor control instructions** - These instructions are used for synchronizing memory accesses and management of caches, TLBs and segment registers.
- **Memory control instructions** - These instructions provide control of caches, TLBs, and segment registers.

▮ *Addressing Modes*

The *PowerPC* architecture supports at least two simple addressing modes for load and store instructions.

1. Register indirect with immediate index

2. Register indirect with index

Addressing modes available in PowerPC 603 implementation are as follows. The load and store instructions have the following three categories of addressing modes.

1. **Register indirect with immediate index** - The effective address is calculated by adding one register value and an immediate value in the instruction.

2. **Register indirect with index** - The effective address is computed by adding two register contents specified in the instruction.

3. **Register indirect** - The effective address is provided in the register specified in the instruction.

Branch instructions have the following three categories of addressing modes.

1. **Immediate** - The effective address is calculated using the immediate value from the instruction. Branch relative and branch absolute modes fall in this category.

2. **Link register indirect** - The target address is provided in a special register called link register (LR).

3. **Count register indirect** - The target address is provided in a special register called count register (CTR).

3.2 Overview of Specification

We have specified the PowerPC 603 in Sim-nML. The specifications consist of user level instructions which includes *integer*, *floating-point*, *load/store*, *flow control* instructions, and user level instructions in *processor control* category. Branch prediction, pipelines, resource usage, and instruction timing have been included.

The description of instruction hierarchy is as follows. Top level node is instruction, and all the instructions are partitioned into two categories, branch and non-branch instructions. The branch instructions are further categorized into conditional and unconditional branch instructions. The unconditional branch instructions are of three types, PC relative, absolute, and branch and link. Branch and link instructions are used for procedure calls and save the return address in a register (this is called linking) before branching to the target address. The conditional branch instructions test the branch condition on bit-fields in the condition register. These conditional branch instructions can be speculated or non-speculated depending on the availability of condition register at the time of the branch resolution. The resource usage model handles this speculation part.

The remaining instructions fall into non-branch instructions. The load and store instructions cover integer and floating-point load and stores for loading and storing byte(8-bits), half word(16-bits), and word(32-bits) of integer types and single precision(32-bits) and double precision(64-bits) of floating point data types. Integer instructions include integer arithmetic and logical instructions. Floating-point instructions include floating-point arithmetic, multiply-add, compare, and move instructions. Flow control instructions include instructions to set bit-fields in the condition register on which branch instructions test for a particular condition. Processor control instructions have instructions for moving to and from special purpose registers. About 235 instructions in the PowerPC 603 have been specified in its Sim-nML specifications.

3.3 Some Issues in Writing Specification

▮ *Bit Fields*

Sim-nML gives bit-range operator to select arbitrary bit fields of memory structures. The syntax of this operator is as follows.

memory-name ⟨*lsb..msb*⟩

The *lsb* is the least significant bit and *msb* is the most significant bit. In Sim-nML the bits of memory locations are numbered from right to left starting with 0 (little-endian). Following example copies a least significant byte from half word memory location to a byte memory location.

```
mem REG1 [ 1 , card ( 16 ) ]
mem REG2 [ 1 , card ( 8 ) ]
REG2 = REG1 ⟨0..7⟩;
```

In the above example if *msb* and *lsb* are interchanged (i.e. REG2 = REG1 ⟨7..0⟩;) then the bit-field is reversed and put into the destination. This brings the issue of writing specification for a big-endian processor, where the bits in memory locations are numbered from left to right. In such cases the specification writer has to convert the bit numbering to Sim-nML numbering and use the corresponding numbers in the specification. This is needed since tools follow Sim-nML conventions. The above example for a big-endian processor without conversion looks like the following.

REG2 = REG1 ⟨15..8⟩;

This expression extracts the most significant byte, reverses it and puts in the destination. This is not what exactly we wanted. So the specification writer has to convert theses bit numbers for the correct interpretation by a tool.

▮ *Instruction Hierarchy*

The specification of the instruction set should strictly follow a tree structure, i.e. it should not contain any cycles. Cycles imply that there are different paths in the hierarchy to reach an instruction from top level node, which is not correct. This causes tools to report errors.

▮ *Program Counter and Current Instruction Pointer*

The semantics of incrementing the program counter (PC) should not appear in the *action* part of any instruction. Only branch instructions should modify the PC in the *action* part. The task of incrementing the PC will be taken care of by the tools (e.g. simulator). Tools increment the PC before the instruction is executed. If the instruction uses the current instruction address then it can not use PC, since it is already incremented. If the instruction set contains such instructions then another variable say OLD_PC can be used for keeping track of the current instruction address in the *action* part.

Chapter 4

Functional Simulation and Trace Generation

Repeated performance analysis and testing is an essential part of a system design phase. Performance analysis tools play major role in this phase. The complexity of modern processors is due to the performance enhancement techniques they use. This makes the simulation a time consuming process. The simulator is of no use if it is unable to simulate at a speed close to that of the real processor. It is not a easy task to make the simulator to run at that speed. Several techniques have been developed to speed up the simulation.

4.1 Simulation Methods

The existing simulation methods can be classified into two categories depending on the technique they employ.

4.1.1 Based on Resource Management

Simulation is done by managing various resources between all the instructions. A resource can be a register-file, functional unit, pipeline stage etc. There are two primary methods in this category. First one is *Cycle Based* in which the resources are managed between instructions each cycle analogous to a processor clock. The second one is *Event Based* in which the resources are managed between instructions based on the events happening like release of a resource by an instruction etc.

4.1.2 Based on Instruction Sequence

Simulation is done by the dynamic execution sequence of instructions. There are two primary methods in this category. First one is *Execution Driven* in which the actual functionality of instruction is simulated along with pipeline etc. Simulation of functionality gives the dynamic sequence of instructions. The second one is *Trace Driven* in which the dynamic sequence of instructions is obtained earlier, then the simulation of pipelines etc. is done on using this sequence. It is clear that trace driven simulation is faster than the execution driven simulation.

In our environment we have an instruction set simulator[17] which uses cycle based simulation method. The speed of this simulator is, however, low. A trace driven simulator is a choice for improving the speed of simulation. Trace driven simulator requires a dynamic instruction sequence. A trace generator is needed for obtaining the instruction sequence. This is one of the reasons behind our design of retargetable functional simulator which can also produce dynamic instruction sequence or the instruction trace.

4.2 Functional Simulation

Simulation of functionality of the instructions is called functional simulation. This includes keeping track of registers contents, memory contents etc. Generally the functional simulation is very fast, due to the fact that the complex simulation of the pipelines and other architectural features is not done. The functional simulation of a program should produce the same results as of the program running on the actual processor. This is very helpful in verifying the design of the instruction set of the processor, it can also be used for the generation of instruction trace of a program which is used by various other tools like trace driven simulator etc.

4.3 Trace Generation

Traces are of two types. One is the instruction trace which is a sequence of addresses of instructions executed, and the other one is the memory trace which is a sequence of addresses of memory locations the program refers to while executing. Memory traces are used to simulate caches and memory systems. Here in this work we generate only the instruction trace.

4.3.1 Issues in Trace Generation

The difficulties in getting the complete program trace comes from the high cost of recording every instruction and data address as the application program executes and from the large size of resulting trace file. A simple tracing system examines every instruction as a program executes. This approach is inefficient and makes the program run slowly. A 10-million-instruction-per-second (MIPS) processors produces nearly up to 70 megabytes of trace per second of execution. This makes the trace for long time execution of the program difficult to store.

4.3.2 Trace Generation Methods

Generally the computational overhead in tracing can be reduced by modifying the computer hardware along with the computer application software to record address. Although such approaches can reduce computational overhead in tracing, the size of the trace remains a big problem. Even if the traces can be compressed using standard compression utilities by a factor of about 10, trace size remains large. Some tracing systems avoid the storage of trace by sending the trace directly to the consuming tool. This is called on-line-tracing. The difficulty with this method is that traces are not sharable.

An efficient tracing system should reduce the tracing time as well as the trace size. This can be achieved by recording minimal number of events while tracing the program. One such technique was given by Ball and Larus[6]. A tool called **qpt**[12] has been implemented using this technique combined with another technique called *abstract execution*[11] for the MIPS 2000 system. This technique uses compiler-based techniques. Tracing of a program is done in two phases. In the first phase a compiler-style analysis is done on the program which helps in reducing the information to collect during the program's execution. This phase identifies a small subset of a trace that suffices to reproduce the full trace. Only the events in this subset, called the *trace record*, is recorded while the program runs. In the second phase a *trace regeneration* process produces the full trace from the trace records.

This is achieved by analyzing the program and instrumenting it to generate the token records. The instrumentation is done in such a way that less number of token records are generated when the program is executed. Then the program is executed to get these token records. The full trace can be obtained form these token records using trace regenerating tool.

4.3.3 Our Approach

All the existing methods require modification in the compiler, or the program, or some other application, or the hardware to produce the trace, and the methods are dependent on the system. For a retargetable system, such an approach is not possible. Therefore, we generate traces by functional simulation of the program. In our approach, the functional simulator, fsim, can generate traces for any program. Since the fsim itself is generated using Sim-nML model for a processor, our set of tools are retargetable. The trace generated, however, is not compressed.

Chapter 5

Design and Implementation

In this chapter we discuss the design and implementation of the *Macro Preprocessor (nMP)* and *Retargetable Functional Simulator (Fsimg)*.

5.1 Macro Preprocessor (nMP)

As we discussed in section 2.3 that nMP is not really a macro processor but a macro translator, which translates the Sim-nML macros into m4[16] macros. This decision was made to simplify the design and implementation of nMP by making use of widely available powerful macro processor m4. Moreover, by this approach we can also use the features available in m4 such as macro calls within macros, recursive macros etc. This encourages the use of macros in an extensive manner.

nMP recognizes only the macro definitions in the given Sim-nML input. The macro calls are not considered because nMP does not do any macro expansion. nMP works in two modes. The first one is the passive mode. In this mode nMP looks for the macro definitions in the input and copies simultaneously the input to the output. By default nMP works in the passive mode. The second mode is the active mode. nMP switches to the active mode when ever it finds a macro definition in the input. In the active mode nMP stores the macro definition in its internal data structures. When the macro definition is completed the Sim-nML macro is translated to the m4 macro and nMP switches back to the passive mode.

nMP is written using *flex* and *bison* in about 500 lines of code. nMP takes input file with macros as a command line argument and writes the output (Sim-nML description with m4 macros) to the file whose name is derived by suffixing ".m4" to the input file name. If no file name is given, nMP reads its input from the standard input and writes its output to the standard output. A shell script has been provided for running m4 on nMP generated output to get the Sim-nML specification without macros.

5.2 Retargetable Functional Simulator (Fsimg)

Fsimg takes as input, the processor description in an intermediate form and a program in **ELF** (section 6.1) executable format and generates a functional simulator (Fsim) for this program. The code generated for Fsim is a **C** code. First we look at the overall process of Fsim generation, then the basic structure of Fsim and details on Fsim generation.

5.2.1 Overview of Fsim Generation

The *action* attribute in the Sim-nML specification captures the semantics of the instructions. Fsimg converts all instructions semantics into respective functions by flattening the hierarchical description of *action* attribute. The Fsimg then decodes the instructions from the program, extracts the parameters for the instruction and generates a call to the respective function with these parameters. All these calls to functions are captured into a table called function-pointer-table whose entries are basically a set of parameters and a function pointer pointing to the respective function. The entries in this table are in the order of the instructions in the program. The simulation starts by calling the function for the first instruction along with its parameters. The called function return the index of the next instruction into the function-pointer-table. In this way simulation continues till the program is terminated. Along with this table Fsimg generates data structures for the memory, registers and other memory elements in the processor, and a driving routine for the simulation which initializes the memory and registers. The driving routine calls the first instruction of the program.

5.2.2 Fsim Structure

Fsim has the following five components (figure 5.1).

1. A set of functions one for each instruction in the processor description.

2. A function-pointer-table corresponding to instructions in the program.
3. The memory image of the program.
4. Data structures for registers and other memory elements.
5. The driving routine.

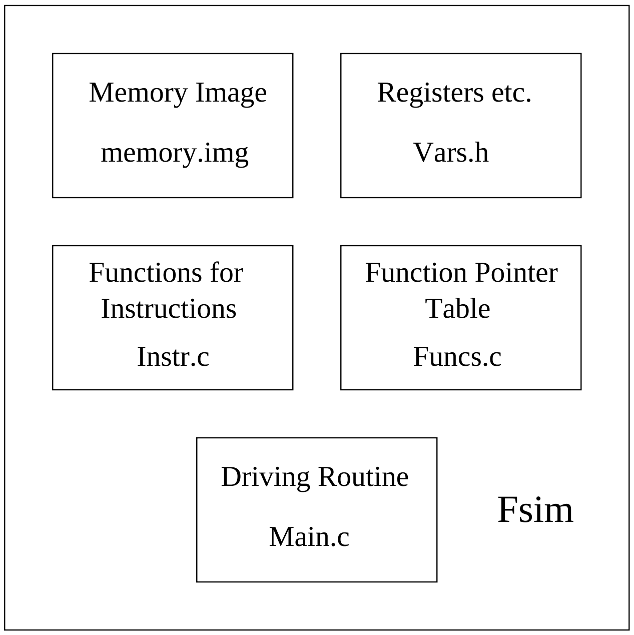


Figure 5.1: Fsim Components

The function-pointer-table is an array of structure whose members are an array of parameters and a pointer to a function. The **C** declaration of function-pointer-table is as follows.

```
struct func_ptr {
    uint64 p [MAX_PARAMS];
    int (*func)(uint64 *);
};

struct func_ptr Func_Pointers [MAX_POINTERS] = {
    {{13, 0, 388}, Fun38},
    {{13, 13, 41916}, Fun146},
            ⋮
};
```

The function pointed to by the function pointer take a pointer to the parameters and return the index of the next instruction. When the simulation starts the driver routine initializes the conceptual program counter (PC), stack pointer (SP), and memory. At this point PC points to the first instruction of the program. The index of this instruction into the function-pointer-table is calculated differently for a processor with fixed instruction length and processor with variable instruction length. For a processor with fixed instruction length, the index calculation is as shown below,

```
index = ( PC - CODE_BASE ) / INSTR_LEN;
```

where CODE_BASE is the address of the first instruction in the program's code segment and INSTR_LEN is the length of the instruction in bytes. Each entry in the function-pointer-table represents an instruction with parameters.

In the case of a processor with variable instruction length, the index calculation is not possible in this manner. In our simplistic approach each entry in the function-pointer-table for such a processor represents only one byte of the instruction. That is for a three byte instruction three entries are used in which first one represents the instruction and the remaining two are not used. With this kind of function-pointer-table the index calculation is as shown below.

```
index = (PC - CODE_BASE);
```

The following example shows the function pointer table for a variable length processor where the length of the first instruction is two bytes and that for the second is three bytes.

```
struct func_ptr Func_Pointers [MAX_POINTERS] = {
    {{2}, Fun190},
    {{}, Dummy_Cisc},
    {{4100}, Fun78},
    {{}, Dummy_Cisc},
    {{}, Dummy_Cisc},
            ⋮
};
```

After computing the index, the driver routine calls the function at this index with the parameters. The function called performs the semantic action associated with the instruction and returns the next instructions index to the driver routine. The function calculates this index as described earlier using the program counter (PC), the length of the instruction, and the base address of the first instruction. The driver routine uses this index to call the function at that index. In this way the instructions get executed (simulated) until the program terminates. The code for the driver routine looks like the following.

```
index = (MAIN_ENTRY − CODE_BASE) / INSTR_LEN;
while((index = Func_Pointers[index].func(Func_Pointers[index].p)) != -1);
```

5.2.3 Code Generation for Fsim

▮ *Extracting instructions and Hashing*

At first Fsimg tries to find out the information about the instructions in the description. All the tables in the IR are read into the memory and syntax and image attributes for all instructions are extracted. Syntax attributes contain the information about the parameters. Image attributes are used to decode the instructions in the program. For this image masks are computed for each image as described here. Generally some of the bits in the image are fixed and others come as parameters. Image mask is basically a bit string that has ones for the fixed bit positions and zeros in the parameter bit positions. In order to decode an instruction, we and it bit-wise with each instruction mask and compare the result with the image attribute in the IR. This process of decoding is, however time consuming. To improve the decoding performance the images are hashed into a hash table.

First a global mask is computed by bit-wise anding all instruction masks. The global mask therefore represents the opcode field of the instructions. First level hashing is done based on the opcode field. However it is not very useful because instructions may not have distributed evenly to all the buckets. It may result in one bucket having large number of instructions while some other buckets having no instructions. For this reason instructions in each bucket are further hashed based on the remaining fixed fields of those instructions. This comes from the observation that when we hash on opcode all instruction of particular type say integer instructions gets hashed to same bucket. Now these instruction have additional fields to identify different instructions amongst them self. These fields are called sub-opcodes. In this way hashing is done several levels until single instruction is hashed to a bucket.

▮ *Generating Functions for instructions (Action Flattening)*

After instruction hashing, Fsimg generates the functions for the instructions in the description. The semantics of an instruction are captured in the *action* attribute, which is hierarchically spread over the path from the top level node to the instruction. A set of dot-expressions[10] in the IR provides the information about this path for each instruction. Starting at the top node till the instruction, the definition in the *action* attributes is captured as a **C** function. All the attribute definitions are available in the PREFIX ATTRIBUTE DEFINITION table in the IR. The definitions are in the prefix notation which are converted in to the infix notation during the code generation. Sim-nML operators like bit-field, left and right rotate, bit-concatenation etc., are converted into library calls (see 6.3.1).

In the definition of *action* attributes the operands of the instruction may appear. The image got from the IR provides the method to find this information. These statements may also contain calls to other attribute definitions. In this case the definition of that attribute is substituted in that place. Before generating the code for the attribute, a unique label is placed so that if the definition recursively calls the same attribute then it can be converted to a goto statement. This situation may arise because Sim-nML lacks loop constructs and the specification writer may need a loop while describing a instruction like load string bytes[1]. This can be written by calling an attribute recursively on a condition. The generated functions take pointer to parameters and returns the index of next instruction into the function-pointer-table.

[1]load string bytes instruction loads n consecutive bytes from a memory address, where n and memory address are operands of this instruction.

▮ *Instruction Decoding and Function Pointer Table Generation*

The Fsimg reads the required information from the given program which is in ELF format into the internal data structures. Depending on the processor type number of function-pointer-table entries are calculated. If it is a processor with fixed instruction length, then the calculation is as follows.

no-of-entries = text-section-size-in-bytes / instruction-length

If it is a processor with variable instruction length, then the calculation is as follows.

no-of-entries = text-section-size-in-bytes

Now the decoding of the instructions in the program is done using the hash table created earlier. Once a instruction is recognized the operand values are extracted from the instruction and a function-pointer-table entry is generated with these values and the corresponding function for the instruction. In case of a processor with variable instruction length, number of entries generated are equal to the length of the instruction in bytes. In this manner all the instructions in all text sections of the program are decoded and function pointer table is generated. If an instruction gets unrecognized then a dummy entry is created in that position of the function-pointer-table. If the control reaches this entry during the simulation, then it generates an error message and simply returns the index of next instruction to the driving routine. This may lead to incorrect results and unpredicted behaviour of the Fsim. To avoid this the specification has to cover all the instructions needed for running the program.

▮ *Generation of Types and Memory Image*

The Sim-nML types are converted into corresponding **C** types, like unsigned int for card and int for int etc. But the problem comes with the sizes of these declarations. Sim-nML allows the declaration of variables of arbitrary bit sizes. Consider the following Sim-nML declaration.

mem TEMP [1 , **int** (4)]

This declares TEMP as a memory location of type integer and size 4 bits. We have to allocate exactly 4 bits for the correctness of the value held by this location. For this the **C** feature of bit-fields inside the structure declaration is used. For the above declaration the code generated is as follows.

```
typedef char int8;

typedef struct {
   int8 val:4;
} Int4;

Int4 TEMP;
```

Whenever TEMP occurs in any attribute definition, TEMP.val is generated in that place. Thus whenever a variable is declared which is not a multiple of 8 bits, nearest C-data structure larger than the one being used in Sim-nML, for example, a 12 bit variable in Sim-nML is declared using int16 type.

Fsimg composes the memory image for Fsim by combining all the data sections and text sections of the program and is written to a file. When Fsim starts it loads this memory image in to its memory. All memory references are redirected relative to the location where it is loaded.

Finally code for the driver routine is generated which consists of the code that initializes the PC, SP, and memory and the code for the simulation as we have seen earlier.

Chapter 6

External interface to Fsimg and Fsim

6.1 Executable and Linking Format (ELF)

ELF was originally developed and published by UNIX Systems Laboratories (USL) as part of the Application Binary Interface (ABI). The Tool Interface Standards committee (TIS)[3] has selected the evolving ELF standard as a portable object file format that works on 32-bit architecture environments for a variety of operating systems.

There are three main types of ELF object files.

- **Relocatable file** - This type of ELF file holds the code and data for linking with other object files to create an executable or a shared object file.
- **Executable file** - This type of ELF file holds a program suitable for program execution. This file specifies how to create the program's process image.
- **Shared Object file** - This type of ELF file holds the data suitable for linking. Linking is done in two ways. The static linking done by the link editor (ld) requires processing of several relocatable and shared object files to create another object file. The dynamic linking involves the combining of an executable file with other shared object files to create a process image.

Object files participate in program linking and execution. The object file format has different views in these two different contexts. The file starts with a machine independent header called ELF header which describes the remaining file organization. A linking view has a set of sections which provide the information needed for linking such as instructions, data, symbol table, relocation information etc. A section header table gives the information related to these sections. An execution view has a set of segments and a program header table which provides the information about how to create a process image.

6.2 Dynamic Library Calls

ELF executable files are two types depending on the way they are linked with the library. A dynamically linked executable file contains references to the library functions which reside inside the shared object files. The dynamic linker resolves these references while creating the process image for this executable. This enables the sharing of same library by many programs. A statically linked executable contains all the code including the library functions and its size may be very large.

Simulation of interactive programs is very difficult because interactive programs interact with the operating system, devices etc. Typically programs use standard library functions for interaction. These library calls can be diverted to the simulator host's library calls for simulating interactive programs. For this dynamically linked executables are more suitable as they are easy to identify in the code.

6.2.1 Handling Dynamic Calls

The Fsimg has the capability to identify the dynamic calls in the program and generates code for Fsim which diverts these calls to the host system's library calls. Fsimg achieves this through the external interface. The list of dynamic calls that may be used by the program can be specified through a configuration file. In this file, for each dynamic function to be diverted, a corresponding user function with the parameters and their size is specified. These user functions substituted for the diverted dynamic calls are linked with Fsim. Fsimg passes the parameters to user functions by reference. The user function extracts the parameters and call the host library function. The return value of the host library function should be modified and returned by the user function to Fsim. The handling of the return value is also specified in the configuration file.

The issues of endianness is also important. The simulated processor may have different endianness than the simulating host. In such cases the parameters which are passed to the user function are converted to reflect the simulating host endianness and return value has to be converted back from host endianness to the simulating processor's endianness. Fsimg detects this mismatch of endianness in parameters and generates code for changing the endianness of parameters before passing to the user function. Similarly return value is also converted. For this Fsimg needs to know the sizes of the parameters and return value which are optionally specified in the configuration file.

In addition Fsimg needs to know the various instructions through which the simulated program may call dynamic library functions. This information can be given through command line options. The call instructions typically modify the PC and save the old PC. Using the destination address of the call instructions, Fsimg generates code for diverting the calls.

6.2.2 Specifying Dynamic Calls

The configuration file has following syntax given in *yacc* style grammar. A terminal symbol starts with a upper case letter and a non-terminal symbol start with a lower case letter.

```
function_definitions :
                     | function_definitions function_definition;

function_definition : Lib_Function => user_function;

user_function : return_variable = Function_Name ( parameters )
              | Function_Name ( parameters );

return_variable : Name
                | Name [ expr ]
                | Name : Size
                | Name [ expr ] : Size;
```

```
        parameters :
                   | parameter_definition , parameters;

        parameter_definition : parameter
                             : parameter : size_info;

        parameter : Name
                  | Name [ expr ];

        size_info : parameter_size
                  | parameter_size $ size_info;

        parameter_size : No_Of_Elements - Element_Size;

        expr : Name
             | Name [ expr ]
             | Constant
             | expr + expr
             | expr - expr
             | expr * expr
             | expr / expr;

Where,

        Name : [a-zA-Z_][a-zA-Z0-9_]*
        Constant : [0-9]+
        Size : [0-9]+
        No_Of_Elements : [0-9]+
        Element_Size : [0-9]+
        Lib_Function : [a-zA-Z_][a-zA-Z0-9_]*
        Function_Name : [a-zA-Z_][a-zA-Z0-9_]*
```

The grammar specifies that configuration file is a sequence of function definitions. Each function definition starts with the name of the dynamic function (the function called in the binary executable file being simulated) followed by a token "=>". The dynamic function is diverted to the user function specified on the right side of "=>". The number of parameters to the dynamic function as well as to the user function is the same. All parameters are specified at the right side. According to grammar the return value specifications need not be given if the user function does not return any value (or, value is to be discarded). The optional size information for a variable can be given after the variable name followed by a ":" token. All parameter specifications to the user function are given within "(" and ")". If the function does not take any parameters, the parameter information can be left out. A parameter is given by its name followed by optional size information. Since Fsimg passes parameters by reference, passing of structures is also possible. When passing structures the size information gives the number of structure elements and their respective sizes. The following example explains this.

```
myfunc =>
   R[0] : 4 = usr_myfunc ( M[SP] : 1-4 $ 1-2 $ 2-4, M[SP+14] : 1-4 )
```

The above example depicts that a user function called "usr_myfunc" should be called instead of simulating a call to a dynamic function "myfunc". The user function takes two parameters from the stack. The return value should be placed in register R0 whose size is 4 bytes. The first parameter of the user function is a structure starting at address given by the register SP in the processor being simulated. The structure has four members with their respective sizes being 4, 2, 4, and 4 bytes. The syntax x-y stands for x number of elements each of size y bytes. The $ in the specification separates these element declarations. Basically $ separates the different size member declarations. If all members of a structure are of the same size then it can be given by a single x-y declaration, for example, 4-2 stands for four members of two bytes each.

The parameter size information is used by Fsimg to generate code for changing the endianness of the parameters before passing them to the user function if the endianness of the simulating processor is not same. If no size information is given Fsimg does not generate the code to change the endianness of the parameters. A sample configuration file for PowerPC 603 is shown below.

```
printf =>
```

```
    GPR[3] = lib_printf( GPR[3], GPR[4], GPR[5],
                         GPR[6], GPR[7], GPR[8], GPR[9] )

memcpy =>
    GPR[3] = lib_memcpy( GPR[3], GPR[4], GPR[5] )

open =>
    GPR[3] = lib_open( GPR[3], GPR[4] )

read =>
    GPR[3] = lib_read( GPR[3], GPR[4], GPR[5] )
```

The parameters to the functions are passed through registers from GPR3 onwards and the return value is put in GPR3. The size information for parameters is not given so Fsimg does not generate any endianness change code for parameters.

In general if the parameters are passed through the registers then there is no need for endianness change. This is because the registers are simulated in simulator memory. The load instructions take care of endianness while loading them properly so there is no need for endianness change. Whereas the parameters in the memory (e.g. stack) needs the endianness change, because the memory image which is loaded from file is in the endianness of the processor being simulated.

6.3 Fsim Library

The Fsim library contains functions that are needed by the Fsim. They are implementation of certain Sim-nML operators corresponding operators for which is not present in **C**. There are also a few other miscellaneous functions.

6.3.1 Sim-nML Operators

Following are the functions for Sim-nML operators present in the library. These operators are for integer data types only. In the current implementation of the library, these operators do not work with floating-point arguments. In general bit-level operators are rarely used on floating-point data types.

- **OpLeftRotate** - This function implements the Sim-nML left rotate operator "$\langle\langle\langle$".
- **OpRightRotate** - This function implements the Sim-nML right rotate operator "$\rangle\rangle\rangle$".
- **OpBitField** - This function implements the Sim-nML bit-field select operator "$\langle lsb..msb\rangle$".
- **OpSetBitField** - This function implements the Sim-nML bit-field operator on the left size of expression for setting selected bits.
- **OpExp** - This function implements the Sim-nML exponentiation operator "$**$".
- **OpBitConcat** - This function implements the Sim-nML bit-concatenation operator "$mem1 :: mem2$".
- **OpSetBitConcat** - This function implements the Sim-nML bit-concatenation operator on the left side of the expression.

6.3.2 Miscellaneous Functions

There are a few miscellaneous functions needed by Fsim. The first one is **InitMem** which initializes the memory before start of the simulation. The second one is **EndianChange** which is used for changing the endian of data.

If the Sim-nML specification contains any canonical functions then the user has to provide those functions also in the library.

6.4 Input Information

The Fsimg needs some information regarding the specification to generate code for Fsim. This information is given through command line options.

Stack Size and its Direction of growth

The stack size for the program is given through the command line option '−S size'. Where size is in kilo-bytes. The default direction of stack growth is from higher address to lower address. This can be changed by giving the negative value to size. If this option is not given default stack size and direction are used.

▮ *Program Counter, Stack Pointer and Current Instruction Pointer*

The program counter (PC) is the variable name used in the Sim-nML description for the program counter of the processor. The stack pointer (SP) is the variable name used for the stack pointer. Some processors do not have any special register for stack pointer. In such cases compiler uses one of the general purpose register as a stack pointer. In that, case the variable name of the register as stack pointer should be given. The current instruction pointer is a dummy program counter used for branch instructions which use the current instruction address as described in the section 3.3. This information is needed for correct code generation. This information can be given by the following command line options.

−p program_counter_name

−s stack_pointer_name

−P current_instruction_pointer_name

▮ *Call instructions and Configuration file*

The information about the call instructions can be given with option '−f call-instruction-node'. Where call-instruction-node is the top *and* or *or-rule* node for call instructions in the specification. The configuration file for dynamic functions can be given with '−c file' option.

6.5 Constraints

Fsimg has certain limitations, due to which it puts some restrictions on writing specifications for Fsimg.

6.5.1 Writing Specification for Fsimg

Sim-nML gives many features for specification writes which are some times difficult to implement. Here we discuss the restrictions in writing Fsimg.

▮ *Data Types*

1. In Sim-nML the specification writer can use data types of any length. The Fsimg allows only maximum length up to the size supported by the simulating host. For example, if one declares an integer of 128-bits, and the simulating host supports only 64-bit integers then the Fsimg would not allow this.

2. Bit operations are not allowed on floating-point data types.

3. Enumerated data type of Sim-nML is not supported.

▮ *Operators*

The sizes of the operands to the bit-concatenation operator when it is used on the left side of the expression should be the natural size of the simulating host data types. That is if the machine supports 8, 16, 32-bit integers then the arguments should only any of these sizes.

The Fsimg disallows the use of bit-field operator and bit-concatenation operator at the same time on the left side of the expression. The following code shows the situation which is not allowed by Fsimg.

```
mem VAR1 [ 1 , card ( 8 ) ]
mem VAR2 [ 1 , card ( 16 ) ]
mem VAR3 [ 1 , card ( 16 ) ]

 VAR2 ⟨ 0 .. 7 ⟩ :: VAR1 = VAR3;
```

▮ *Aliases*

The Fsimg allows use of aliases in a restrictive manner. Only byte level aliases are supported. In other words, size of an alias should be of a multiple of 8-bits (8, 16 etc.) and the location to which it is aliased should be byte aligned. Following example shows the various possible methods of aliasing.

```
reg AC [ 1 , card ( 32 ) ]
reg BX [ 1 , card ( 20 ) ]

mem ALIAS_1 [ 1 , card ( 8 ) ] alias = AC [ 7 ]
mem ALIAS_2 [ 1 , card ( 8 ) ] alias = AC [ 11 ]
mem ALIAS_3 [ 1 , card ( 4 ) ] alias = AC [ 3 ]
mem ALIAS_4 [ 1 , card ( 8 ) ] alias = BX [ 7 ]
```

In the above example two registers AC and BX are declared with sizes being 32-bits and 20-bits respectively. Four variables are declared which are aliased to these registers. ALIAS_1 is a valid alias definition because its size is one byte and it is aliased to the least significant byte of AC who's size is multiple of byte size. ALIAS_2 is not a valid alias definition because it is aliased to a location that is not byte aligned. ALIAS_3 is also not valid because its size is 4-bits and is aliased to invalid position. ALIAS_4 is also not valid because it is aliased to BX who's size is not multiple of byte.

Load and Store Instructions

The main memory used in the Sim-nML description is typically byte addressable. However, Sim-nML allows accessing multi-byte items with a given address. Following example explains this.

```
type byte = int ( 8 )

mem M [ 2**16 , byte ]
reg REG [ 1 , int ( 16 ) ]
mem EA [ 1 , card ( 16 ) ]

M [ EA ] = REG;
```

Although M is a byte addressable according to its declaration, the effect of the above statement is storing the REG's contents in two consecutive bytes starting from EA. This brings in the issue of endianness even in the specification. The Fsimg does not support this feature currently. Hence the specification writer has to write the code for storing these bytes separately.

The registers of any machine are always big-endian, but the memory endianness depends on the machine. The endianness has to be changed when ever a multi-byte item is loaded from memory to a register or stored to the memory from a register. In general this can be done at two places, when the simulation is done the simulator dynamically converts, or in the specification itself where the simulator would not bother about the conversion. The first method is a big overhead on the simulator. It has to keep track of may things and has to do the conversion for each memory access. The second method removes this overhead form the simulator. Since the specification writer knows the endianness of the processor he can take care of endianness while writing specification for load and store instructions. Fsimg requires the specification writer to take care of endianness. This can be achieved by using byte level aliases or bit operators. For floating-point load and store instructions use of aliases is the only possibility.

▮ *Program Termination*

To terminate the simulation gracefully, the simulated program has to call exit() at the end. Further the behaviour of exit() should be specified in the configuration file. In absence of call to exit() library function, Fsim produced exhibit unpredicted behaviour.

Chapter 7

Results and Conclusions

In this chapter, we discuss a few performance based results of the functional simulator (Fsim) and conclude this thesis.

7.1 Results

Fsimg has been tested for its retargetability for two different processors PowerPC 603 and Motorola 68HC11. Fsim has been tested for its functional simulation and performance. The functional simulation as well as the performance depends on the description. If the description is erroneous then the functional simulation can not be correct. Further even if the description is correct but the instruction semantics are specified in an inefficient manner then the performance gets affected.

▮ *Test Setup*

The simulation results have been taken on two different machines with following configurations.

- **Machine I**: Intel P-II 233MHz, a little-endian processor with 32MB RAM running GNU-Linux Kernel 2.0.32.
- **Machine II**: Sun Ultra SPARC II 250MHz, a big-endian processor with 512MB RAM running SunOS 5.5.1.

The PowerPC 603 processor description written as a part of this work has been used for the testing. Following are the test programs written in **C**. The PowerPC 603 ELF binaries were created using a GNU cross-compiler.

- **mmul.c** : Matrix multiplication program. This program initializes two integer matrices of 100x100 size and multiplies these two.
- **bsort.c** : Bobble sort program. This program initializes an array of 1500 integers in descending order and sorts them to ascending order using bubble sort algorithm.
- **qs.c** : Quick sort program. This program initializes array of 1,00,000 integers in descending order and sorts them to ascending order using quick sort algorithm.
- **fmmul.c** : Matrix multiplication for floating-point numbers. Initializes and multiplies two floating point matrices of size 100x100.
- **nqueen.c** : This program finds all the possible ways that N queens can be placed on an NxN chess board so that the queens cannot capture one another. Here N is taken as 12.

All these programs were tested for the functional correctness on both the machines. Here we present some performance based results for the above programs on these two machines. The table 7.1 gives the total number of dynamically executed instructions during the simulation for each of these programs.

Program	Total No. of Instructions
mmul.c	91,531,966
bsort.c	60,759,034
qs.c	80,773,862
fmmul.c	92,131,966
nqueen.c	204,916,928

Table 7.1: Total number of instructions simulated for test programs.

The functional simulator *Fsim* is compiled using a GCC compiler. The performance results are taken by compiling *Fsim* with optimization (optimization level 3 i.e -O3) and without optimization. In an unoptimized mode GCC tries to reduce the cost of compilation with out doing any optimizations on the generated code. In an optimized mode it tries to optimize the code for reducing the code size as well as execution speed at a higher cost of compilation.

Machine I:

Program	With no optimization		With optimization level 3	
	Total time (seconds)	Instructions per second	Total time (seconds)	Instructions per second
mmul.c	65.7	1,393,181	59.8	1,503,635
bsort.c	102.3	593,930	97.6	622,531
qs.c	111.7	723,132	108.7	743,090
fmmul.c	66.5	1,385,443	60.0	1,535,533
nqueen.c	265.2	772,688	262.4	780,933

Table 7.2: Performance Results on Machine I

Machine II:

Program	With no optimization		With optimization level 3	
	Total time (seconds)	Instructions per second	Total time (seconds)	Instructions per second
mmul.c	68.4	1,338,187	52.1	1,756,851
bsort.c	96.0	632,907	75.6	803,691
qs.c	105.7	764,180	84.4	957,036
fmmul.c	72.1	1,277,836	48.9	1,884,089
nqueen.c	261.2	784,521	234.7	873,101

Table 7.3: Performance Results on Machine II

▮ *Analysis of Results*

The results of test programs from tables 7.2 and 7.3 shows that three of these programs are simulated at 0.5 MIPS (million instructions per second) to 0.9 MIPS. But the *mmul.c* and *fmmul.c* are simulated at above 1 MIPS. After careful analysis of the instructions executing for all these programs we found that the simulator is spending variable amount of time in the library functions corresponding to Sim-nML operators. The simulation speed differs due to the varying number of calls to such library functions. The matrix multiplication programs have fewer number of instructions which in turn call these library functions. This number is higher in the other three programs which made them to slowdown.

An experimental setup has been used for finding the simulation timings of various individual instructions. We found that different instructions are having different simulation timings and the instructions having greater simulation time are having more calls to the library operators. The mix of instructions which are having greater simulation time is more in the programs whose simulation speed is less.

The conclusion we can make from this is that infrequent use of Sim-nML operators in the description can result in faster simulation. Further a better and faster implementation of Sim-nML operators can further improve the simulation.

Trace Results

The table 7.4 gives the approximate trace sizes for the test programs. Since the trace generated is not compressed and for each instruction executed a four byte address is generated, the trace size is four times the number of instructions executed.

Program	Approximate trace size (mega bytes)
mmul.c	349
bsort.c	232
qs.c	308
fmmul.c	351
nqueen.c	782

Table 7.4: Approximate trace sizes for test programs.

7.2 Conclusions

In this thesis we have discussed the Sim-nML language for modeling processors at instruction level. It is powerful enough to specify any modern processor with pipelines, branch prediction, etc. at the instruction level. We have also discussed the integrated environment where automatically tools (assembler, simulator, compiler, etc.) can be generated using Sim-nML processor models.

As a part of this work we have specified PowerPC 603 processor in Sim-nML. Around 237 instructions have been specified with resource usage model and pipeline. These instructions cover most of the user level instructions of the PowerPC 603 instruction set. We have implemented a *Macro Preprocessor (nMP)* for processing Sim-nML macros. nMP converts Sim-nML macros into m4 macros there by making the task of macro expansion simple. We have also designed and implemented a *Retargetable Functional Simulator (Fsimg)*. The Fsimg takes a processor description in an intermediate form and an executable in **ELF** format and generates a function simulator (Fsim). The Fsim simulates functionally executable program for the desired processor. It can also generate the instruction trace of the program which is useful for other tools in studying various other aspects of the design.

7.3 Future Work

Following points can be considered as an extension to this work.

- Removing the restrictions which Fsimg is imposing on the specifications writing. These include supporting aliases, allowing the use of bit-field and bit-concatenation operators at the same time on the left side of an expression, allowing operands of any size for bit-concatenation operator, supporting data types of any size, and supporting Sim-nML enumeration data type.

- Current bit-operator library supports only integer data types. This can be extended to floating-point data types also.

- The trace produced by Fsim is not compressed. It makes it difficult to handle and process trace files. Producing the compressed trace is one of the improvements that can be achieved.

- Simulation speed is one of the important issue generally we look for. Improving Fsim's speed by possibly rewriting a better and faster operator library and by changing the logic of the driving routine can be another future work.

- Although it may not be an extension to this work, Retargetable Trace Driven Simulator can be designed that can use the trace generated by Fsim for the performance study of processors.

Bibliography

[1] Bochs Software Company. http://www.bochs.com.

[2] Snes9x.com. http://207.5.92.43.

[3] The Tool Interface Standards committee (TIS). http://developer.intel.com/vtune/tis.htm.

[4] VMac group. http://leb.net/vmac/.

[5] VMware, Inc. http://www.vmware.com/products/virtualplatform.html.

[6] Ball, T., and Larus, J. R. Optimally Profiling and Tracing Programs. *ACM Transactions on Progamming Languages and Systems 16*, 4 (July 1994), 1319–1360.

[7] C.May, E.Silha, R. S., and H.Warren, Eds. *The PowerPC Architecture: A Specification for A New Family of RISC Processors.* Morgan Kanfmann, 1994.

[8] Freerick, M. The nML Machine Description Formalism, July 1993. http://www.cs.tu-berlin.de/~mfx/dvi_docs/nml_2.dvi.gz.

[9] George Hadjiyiannis, S. H., and Devadas, S. ISDL An Instruction set Description Language for Retargetability. *Proceedings of the 34th Annual Conference on Design Automation Conference* (1997), 299.

[10] Jain, N. C. Disassemble using High Level Processor Models. Master's thesis, Department of Computer Science and Engg., IIT Kanpur, Jan 1999. http://www.cse.iitk.ac.in/research/mtech1997/9711113.html.

[11] Larus, J. R. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software Practice & Experience 20*, 12 (Dec 1990), 1241–1258.

[12] Larus, J. R. Efficient Program Tracing. *Computer 26*, 5 (May 1993), 52–61.

[13] Mendel Rosenblum, Edouard Bugnion, S. D., and Herrod, S. A. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation 7*, 1 (Jan 1997), 78–103. http://simos.stanford.edu.

[14] Mondal, S. Compiler Back-end Generation using nML Machine Description. Master's thesis, Department of Computer Science and Eng., IIT Kanpur, June 1999. http://www.cse.iitk.ac.in/research/mtech1997/9711117.html.

[15] Raksey, N., and Fernandez. Specifying Representations of Machine Instructions. *ACM Transactions on Programming Langauges and Systems 19*, 3 (May 1997), 492–594. http://www.cs.virginia.edu/~nr/pubs/specifying-abstract.html.

[16] Seindal, R. GNU m4. http://www.seindal.dk/rene/gnu/.

[17] V.Rajesh. A Generic Approach to Performance Modeling and its Application to Simulator Generator. Master's thesis, Department of Computer Science and Engg., IIT Kanpur, July 1998. http://www.cse.iitk.ac.in/research/mtech1996/9611123.html.

[18] *UNIX System V Release 4, Programmers Guide : ANSI C and Programming Support Tools.* Prentice-Hall of India Private Ltd., New Delhi, 1992. Executable and Linkable Format (ELF), Tools Interface Standards (TIS), Portable Formats Specification, Version 1.1.

[19] *PowerPC^{TM}603 RISC Microprocessor User's Manual.* IBM Microelectronics, Motorola, 1994. http://www.mot.com/SPS/PowerPC/products/semiconductor/cpu/603.html.

Appendix A

File Format of Intermediate Representation

In this appendix, we will discuss the layout of the file for the intermediate representation. The file consists of various fixed or variable size tables where the name of each table is fixed. A table, named as `META TABLE`, is always the first table in the file. All other tables can reside anywhere in the file and can be located using the `META TABLE`. The following are the tables available presently in the IR.

- META TABLE
- CONSTANT TABLE
- ATTRIBUTE TABLE
- RESOURCE TABLE
- IDENTIFIER TABLE
- MEMORY TABLE
- AND RULE TABLE
- OR RULE TABLE
- SYNTAX TABLE
- IMAGE TABLE

- STRING TABLE
- INTEGER TABLE
- PREFIX ATTR DEF TABLE

Each table consists of an array of records. Each record in a table constitutes of various fields. For each table, all the fields of first records are written first in the file. Then all the fields of second record are written and so on. We have used the word *record* and *entry* interchangeably. The fields might be stored either in little-endian encoding or big-endian encoding depending on the processor on which the file is created.

- Convention : Each table is described by defining its record format. We have used a C-like struct definition to describe a record. For each record, fields are written from top to bottom in the file. In describing the record, following data types are being used. Size is in *bytes*.

Type	Size	Purpose
`Byte`	1	*Unsigned Byte*
`Word`	2	*Unsigned Word*
`Dword`	4	*Unsigned Double Word*
`SByte`	1	*Signed Byte*
`SWord`	2	*Signed Word*
`SDWord`	4	*Signed Double Word*
`String`	-	Null terminated array of *SBytes*
`Address`	4	`Dword`
`Offset`	4	`Dword`

A.1 Meta Table

The `Meta table` holds the table of contents for all the tables which are present in the file. Each record of the `META TABLE` stores the information to locate a table. Each record has the following format.

```
typedef struct{
   String table_name;
   Dword table_size;
   Address table_offset;
   Dword total_record;
   Dword record_size;
}Meta_Record;
```

table_name	:	This field stores the fixed name of a table which is a 32 byte null terminated string. Name of all the tables are written earlier.
table_size	:	This field holds the size (in bytes) of a table.
table_offset	:	This field holds the starting offset (in bytes) of a table in the file.
total_record	:	This field holds the number of record stored in a table. For the `string table`, it holds the value 0.
record_size	:	This field holds the size of a record (in bytes) of a table. If a record for a table is variable in size, then this field contains the value 0.

The data encoding of the IR is dependent on the processor on which it is created i.e. data encoding can be little-endian or big-endian depending on the processor. A tool can figure out the endian-ness of the IR by reading the table of contents irrespective of the type of the machine on which the tools is running. First record of the table represent the META TABLE entries itself. Therefore the *no-of-rec* contains the total number of tables including the META TABLE, *size-of-rec* contains the size of each record in the meta table and *size-of-table* contains the total size of the meta table including the first record. A tool can read these values and check if the following equation is satisfied.

no-of-rec * *size-of-rec* = *size-of-table*

If this equation is not satisfied, then the endian-ness of the IR and the machine on which the tool is running are not the same, otherwise they are the same. In the former case, this equation must be satisfied after the endian-ness conversion of the fields values.

A.2 Constant Table

Each record of the `CONSTANT TABLE` holds the informations about the `constants` in the following format.

```
typedef struct  {
   Offset id_name;
   Dword val_typ;
   SDwor dvalue;
}Const_Record;
```

id_name	:	This field holds the index into the `STRING TABLE`. As discussed earlier, `STRING TABLE` holds null terminated strings. Thus this field represents a reference to the constant name.
val_typ	:	This field indicates type of the value associated with the constant (0 for *integer* type or 1 for a *string* type).
value	:	If the *val_typ* field represents *integer*, then this field holds the corresponding *signed integer* value. If the *val_typ* field represents *string*, then this field holds the *unsigned integer* index into the `string table` from where a null terminated string value can be retrieved.

A.3 Resource Table

Each entry of this table holds the information about a *resource*. Each *resource* is assigned a unique integer key by which it is referenced at other places. Each record has the following format.

```
typedef struct{
   Offset res_name;
   Dword res_key;
}Resource_Record;
```

res_name	:	This field holds the index into the `STRING TABLE`. In the `STRING TABLE`, the name of the *resource* is stored at this index.
res_key	:	This field holds the key value assigned to the *resource*.

A.4 Identifier Table

This table holds the informations about all the identifiers used in the processor specification file (other than those specified in the `CONSTANT TABLE` and the `RESOURCE TABLE`). Each identifier is assigned a unique integer key which is used to refer to the identifier at other places. Each record has the following format.

```
typedef struct{
   Offset id_name;
   Dword id_typ;
   Dword id_key;
}Identifier_Record;
```

id_name	:	This field holds an index into the `STRING TABLE`. The `STRING TABLE` holds a null terminated string at this index which is the name of the identifier.
id_typ	:	This field indicates the type of the identifier and may have one of the following values.

0	:	Undefined Identifier
1	:	Name of a memory Variable
2	:	Name of an *or-rule* of *mode* type
3	:	Name of an *and-rule* of *mode* type
4	:	Name of an *or-rule* of *op* type.
5	:	Name of an *and-rule* of *op* type.
6	:	Name of an Exception
others	:	Unspecified
id_key	:	This field holds the key value assigned to the identifier.

A.5 Attribute Table

Each entry of this table holds the name of an *attribute*. Each *attribute* is assigned a unique integer key to refer to it at other places. Each record has the following format.

```
typedef struct{
   Offset attr_name;
   Dword attr_key;
}Attribute_Record;
```

attr_name	:	This field holds an index into the `string table`. The `STRING TABLE` holds a null terminated string at this index which is the name of the *attribute*.
attr_key	:	This field holds the key value assigned to the *attribute*.

Note : For *mode* specification, one new attribute ,`_val_`, is defined to store the optional expression associated with =.

A.6 Memory Table

Each entry of this table holds the information about a memory variable specified with *reg* or *mem* specification construct of `Sim-nML` language. Each record has the following format.

```
typedef struct{
   Dword id_key;
   Dword siz;
   Dword tot_attr;
   Dword mem_reg;
   Dword data_typ;
   Dword value1;
   Dword value2;
   Dword attr_list_index;
}Memory_Record;
```

id_key	:	This field stores the key value associated with the identifier name of a memory variable. The key value is assigned in the `IDENTIFIER TABLE`.
siz	:	A memory declaration defines a memory base i.e. a set of memory locations accessible under a name and an index. This field specifies the number of such locations.

tot_attr	:	A memory declaration may also define values for some predefined attributes. This field specifies how many attributes are defined for the memory variable.
mem_reg	:	This field holds a value 0 if the memory identifier is declared using *Reg* specification. It holds 1 if the memory identifier is declared using *mem* specification. Both type of identifiers are similar in nature except that first type of identifiers refer to processor registers and second type of identifiers refer to memory locations.
data_typ	:	
value1	:	
value2	:	A memory location might hold values of different data types. The data type is encoded in a tuple <*data_typ, value1, value2*> First field, *data_typ*, specifies what type of values can be stored in a memory location. Second and third field stores the value according to the *data_typ* field. Table A.1 shows the possible values for these field.
attr_list_index	:	If the *tot_attr* field has a value 0, then this field is ignored and should be 0. Otherwise it specifies an index into the `integer table`. At this index, three integers are stored for each of the *attributes.* Therefore, the total number of integers are $3 * total\_attr$. Each integer triple indicates <*attr_key, offset, len*> where the *attr_key,* is the key corresponding to *attribute name* assigned in the `attribute table`. The second field of triple, *offset*, is the starting tuple number into the `PREFIX ATTRIBUTE DEFINITION TABLE` where definition of the attribute is stored in prefix notation. Third field of triple, *len*, is the number of tuples for its attribute definition.

A.7 And-Rule Table

This table holds the information about all the *and-rules* (*mode* and *op type*). It includes the information about *sub-rules* and *attributes.* The *sub-rules* of an *and-rule* are numbered from 0 to n and parameters are numbered as 0 to m from left to right. Each record has the following format.

Data Type	*data_typ*	*value1*	*value2*
bool	0	0	0
card(n)	1	n	0
int(n)	2	n	0
fix(n, m)	3	n	m
float(n, m)	4	n	m
range[$n..m$]	5	n	m
enum(id_1...id_m)	6	0	$m - 1$

Table A.1: Encoding of data types

```
typedef struct{
   Dword and_key;
   Dword id_key;
   Dword total_sub_rule;
   Dword total_para;
   Dword total_attr;
   Dword attr_list_index;
   Dword para_list_index;
}And_Rule_Record;
```

and_key	:	This field holds an integer which is a unique key assigned to an *and-rule*. This key is used later to refer to the *and-rule*.
id_key	:	This field holds the key value which is assigned to the identifier name of the *and-rule* in the `identifier table`.
total_sub_rule	:	This field holds the number of *sub-rules* generated by flattening of the *and-rule*.
total_para	:	This field holds the number of parameters taken by the *and-rule*.
total_attr	:	This field specifies the number of attributes defined for the *and-rule*.
attr_list_index	:	If *total_attr* field has value 0, then this field is ignored and has a value 0, otherwise it specifies an index into the `integer table`. At this index, three integers are stored for each of the attributes. Each integer triple indicates <*attr_key, offset and len*> similar to the one described in the `memory table`. There are two exceptions here. If *attr_key* refers to a *syntax* or *image* attribute,

then *offset* field contains the starting index in the `SYNTAX TABLE` or the `image table` and *len* field contains the total number of *syntax* or *image* records corresponding to the *and-rule*.

para_list_index : If *total_para* field has value 0, then this field is ignored. Otherwise it specifies an index into the `integer table`. At this index, three integers are stored for each of the parameter. Initially, all parameters triples of first *sub-rule* are written, then all parameter triples of second *sub-rule* are written and so on. Thus if we have `n` *sub-rules* and `m` parameters, then there will be `n*m` such integer triples. Each integer triple indicates <*data_typ, value1, value2*> i.e. the data type of parameter. Table A.2 shows possible values for fields of the triples.

Data Type	*data_typ*	*value1*	*value2*
bool	0	0	0
card(n)	1	n	0
int(n)	2	n	0
fix(n, m)	3	n	m
float(n, m)	4	n	m
range[$n..m$]	5	n	m
enum(id_1...id_m)	6	0	$m - 1$
and-rule	7	*and_key*	0

Table A.2: Parameter Type for *and-rule*

A.8 Or-Rule Table

This table holds the information of all *or-rules* (*mode* or *op* type). Each entry describes the children nodes of an *or-rule.* Each record has the following format.

```
typedef struct{
   Dword or_key;
   Dword id_key;
   Dword total_child;
   Dword child_list_index;
}Or_Rule_Record;
```

or_key	:	This field holds an integer which is a unique key assigned to an *or-rule*.
id_key	:	This field holds the key value associated with the identifier name of the *or-rule* in the `identifier table`.
total_child	:	This field holds the integer number which indicate number of children generated by the flattening procedure for the *or-rule*.
child_list_index	:	This field holds the index into the `INTEGER TABLE` where a list of *and_key* values are stored. Number of such *and_key* values is given by the value of *total_child*. These *and_key* are uses to refer to the *and-rule* (assigned in the `and-rule table`).

A.9 Syntax Table

This table holds the *syntax records* associated with the *syntax* attribute definition of all *and-rules*. Each record has the following format.

```
typedef struct{
   Dword syn_key;
   Dword dot_expr_len;
   Offset dot_expr_offset;
   Dword syn_expr_len;
   Offset syn_expr_offset;
}Syntax_Record;
```

syn_key	:	This field holds an integer which is a unique key assigned to a *syntax record*. In the `and-rule table`, the key is used to get the attribute information of *syntax* attribute.
dot_expr_len	:	This field holds the length of a character string, named as *dot-expression*.
dot_expr_offset	:	This field holds the offset in bytes into the `STRING TABLE` where actual *dot-expression* is stored as a sequence of characters.
syn_expr_len	:	This field holds the length of the character string, named as *syntax-string* of the instruction.
syn_expr_offset	:	This field holds the offset in bytes into the `STRING TABLE` where the *syntax-string* is stored as a sequence of characters.

A.10 Image Table

This table holds the *image records* associated with the *image* attribute definition of all *and-rules*. Each record has the following format.

```
typedef struct{
   Dword img_key;
   Dword dot_expr_len;
   Offset dot_expr_offset;
   Dword syn_expr_len;
   Offset img_expr_offset;
}Image_Record;
```

img_key	:	This is the unique integer assigned to each *image record*. In the `and-rule table`, this value is used to get the attribute information of *image* attribute.
dot_expr_len	:	This field holds the length of the character string, named as *dot-expression*[10].
dot_expr_offset	:	This field holds the offset in bytes into the `string table` where actual *dot-expression* is stored as a sequence of characters.
syn_expr_len	:	This field holds the length of the character string, named as *image-string* of the instruction.
syn_expr_offset	:	This field holds the offset in bytes into the `STRING TABLE` where the *image-string* is stored as a sequence of characters.

A.11 String Table

This table holds null terminated character sequences, commonly called *strings*. These strings are referred to by an index into the string table. The first byte at index zero always contains a *null* character. Similarly, the last byte also contains a *null* character, ensuring *null* termination for all strings. A string whose index is zero specifies either no name or a null name depending on the context. We show one example of the string table of size 30 bytes in table A.3 and the *strings* associated with various indices in table A.4.

null	i	d	e	n	t	i	f	i	e
r	null	P	C	null	null	i	n	s	t
r	u	c	t	i	o	n	null	1	null

Table A.3: Example of the String Table

Index	*string*
1	identifier
12	PC
16	instruction
18	struction
0	null

Table A.4: Interpretation of the String Table

A.12 Integer Table

This table holds list of *unsigned integer* values (`Dword` type). These integers represent different meanings in different contexts. The integers are referred to by an *index* into the `integer table`. The first entry always stored in this table contains 0. The *index* refers to the starting entry and not the starting offset. The offset can be found by multiplying the index and the the size of `Dword`.

A.13 Prefix-Attribute-Definition Table

This table holds various *attribute* definitions in prefix notation. All attributes except the *syntax* and *image* are converted into the prefix notation and stored in this table. Each item of the prefix expression is stored in the following record of type `Tuple_Record`.

```
typedef struct{
   Word typ;
   SDword value;
}Tuple_Record;
```

typ : This field holds an integer value to indicate the type of tuple i.e. an *operator tuple* or *operand tuple*. If tuple is of *operand type*, then this field also encodes the type of operand.

value : This field holds a integer value which will be interpreted according to the value of *typ* field.

An attribute definition is stored in the `and-rule` table and in the `MEMORY TABLE` with the starting index into the `PREFIX ATTRIBUTE DEFINITION` table and the number of items in the prefix notation of the definition. Table A.5 shows the possible values of *typ* field and corresponding interpretation of *value* field. If the *typ* field holds the value 0, then the tuple is *operator tuple*, otherwise the tuple is *operand tuple*. If the tuple is of *operator type*, then *value* field holds an integer which indicates operator name and arity. Table A.6 shows all possible values for this field and corresponding arity of the operator.

Type of the tuple	*typ* field	*value* field
Operator	0	operator number (see table A.6)
Fixed constant	1	`signed integer` value of operand
Card constant	2	`unsigned integer` value of operand
Binary constant	3	Offset into the `STRING TABLE`
Hex constant	4	Offset into the `STRING TABLE`
String constant	5	Offset into the `STRING TABLE`
Memory variable	6	key of the identifier as assigned in the `MEMORY TABLE`
Attribute type	7	key of the attribute name as assigned in the `ATTRIBUTE TABLE`
Parameter type	8	parameter number (left most is assigned number 0).
Resource type	9	key of the resource name as assigned in the `RESOURCE TABLE`
Exception type	10	Key of the identifier as assigned in the `IDENTIFIER TABLE`

Table A.5: Interpretation of the tuple used in Prefix Notation

There are as many operands available as needed for an operator. Since the arity for an operator is fixed, the number of arguments is implicit. For example, an expression $PC = PC + 2$ is $= PC + PC2$ in prefix notation and it has 5 items. The first item is an operator '='. Second is a *memory variable* with value field being the index into the `memory table`. Third item is again an operator '+'. The last field is a *fixed-constant* 2.

value	Name of Operator	Symbol	Arity of Operator
0	Addition	+	Binary
1	Subtraction	-	Binary
2	Multiplication	*	Binary
3	Division	/	Binary
4	MOD	%	Binary
5	EXP	**	Binary
6	Greater than	>	Binary
7	Less than	<	Binary
8	Equal to	==	Binary
9	Not equal to	!=	Binary
10	GEQ	>=	Binary
11	LEQ	<=	Binary
12	Logical AND	&	Binary
13	Logical OR	|	Binary
14	Logical XOR	^	Binary
15	AND	&&	Binary
16	OR	||	Binary
17	Left Shift	<<	Binary
18	Right Shift	>>	Binary
19	Rotate Left	<<<	Binary
20	Rotate Right	>>>	Binary
21	Dot	.	Binary
22	Concatenation	::	Binary
23	Indexing	[]	Binary
24	Assignment	=	Binary
25	Statement Separator	;	Binary
26	Unary Addition	+	Unary
27	UNOT OPERATOR	!	Unary
28	Unary Subtraction	-	Unary
29	Bitwise NOT	~	Unary
30	Bit Range	..	Ternary
31	IF	if then else	Ternary
32	Function	canonical function	n-ary
33	Switch	switch	n-ary
34	default Expression	default	0-ary
35	NULL	nothing	0-ary
36	Hash	#	Binary
37	Comma	,	Binary
38	Condition	{}	Unary
39	Colon	:	Binary

Table A.6: Operators Used in Prefix Attribute Definition

For detailed description of each operator, read the Sim-nML specification given in Appendix A. There are some special cases which are described here.

- The first case is for Bit Range operator which has the infix notation as $opd1 < opd2..opd3 >$.
  Equivalent prefix notation used is as follows.
  $(operator, bitrangeoperator, opd1, opd2, opd3)$.
- The second case is for "if then else". If there is no operand in *else* part, then NULL operator (0-ary) (see table A.6) is being used.
- The third case is when there is a no attribute expression for an attribute. We have used NULL operator to denote it.
- The fourth case is that of a `switch` operator. General infix notation for this is

```
switch (expr)
{
   case   Expr_1  :  Sequence_1 ;
   case   Expr_2  :  Sequence_2 ;
   .
   default       :  Sequence_i ;
   .
   case   Expr_n  :  Sequence_n ;
}
```

  The corresponding pre-fix notation is as follows :

```
(operator, switch)
   (n, expr,
       Expr_1,            Sequence_1,
       Expr_2,            Sequence_2,
       ....
       DEFAULT OPERATOR, Sequence_i,
       ....
       Expr_n,           Sequence_n)
```

The first item is an operator with operator name as `switch`. Then next item is a simple operand tuple of Card constant type and value as n. After that, expr will be again written in prefix notation. It will be followed by n-operands where each operand is an expression in prefix notation and sequence of statements in prefix notation. Default operator is a 0-ary operator so it can be taken as a pre-fix expression.

- The fifth case is that of a canonical function. General notation for this is as follows.
  "function name" $(Arg1, Arg2, Arg3, ..., Argn)$
  where each argument is again an expression. The corresponding pre-fix notation is as follows.

  ```
  (operator, function)
    (length of name, "function name" string,
     n, Arg1, Arg2,........Argn)
  ```

  The first item is a function operator. Second tuple is a string constant type (`typ` = String constant, `value` = byte offset into the string table where function name is written). Next item is a simple operand tuple with `typ` as Card constant and `value` as n. Then each argument is represented in prefix notation.

  There is one special case with function operator where the function name is *coerce*. This function takes first argument as a data type. In the IR, we convert data types to the basic data types and represent them using three numbers, *data_type, value1* and *value2* as described in table A.1. Thus, the data type parameter for the *coerce* function is converted to three integers internally. Therefore, we have two extra parameters for this function. Thus number of parameters are increased by two.

Appendix B

User manuals for nMP and Fsimg

B.1 Macro Preprocessor (nMP)

Getting nMP:

The source code of nMP can be obtained from the following FTP site.

ftp://cse.iitk.ac.in/pub/moona/simnml-nMP-0.1.tar.gz

The number 0.1 stands for the version. Future versions will be placed with higher numbers like 0.2, 0.3 etc. The file is a Unix tape archive in the compressed format.

Compiling nMP:

The downloaded file can be uncompressed and untarred to reveal the source files. Following GNU tools are required to compile nMP.

- **gcc** 2.7.2 or higher
- **flex** 2.5.4 or higher
- **bison** 1.25 or higher

This code may be compatible with lower versions or older *lex* and *yacc* tools which we have not tested. To compile the source "make" is executed. This creates an executable for nMP.

Command Line Options:

nMP does not take any command line options. It takes the input file as an argument and writes to the file with ".m4" suffixed to the input file name. If no arguments are given, it reads from the standard input and writes to the standard output.

Running m4:

A shell script "runm4" has been provided for running m4 on the output generated by nMP. Given the name of the Sim-nML file with m4 macros as the first argument and the name of the output file name as the second argument, runm4 generates the output file that contains no macros. One can also run the m4 manually using the following command.

```
% m4 -P input-file > output-file
```

Example:

```
% nMP input.nml
```

This generates a Sim-nML file "input.nml.m4" which has only m4 macros.

```
% runm4 input.nml.m4 output.nml
```

This generates a Sim-nML file "output.nml" which has no macros.

B.2 Fsimg

Getting Fsimg:

The source code of Fsimg can be obtained from the following ftp site.

```
ftp://cse.iitk.ac.in/pub/moona/simnml-Fsimg-0.1.tar.gz
```

The downloaded file can be decompressed and untarred to get the source files.

Compiling Fsimg:

Following GNU tools are required to compile Fsimg.

- **gcc** 2.7.2 or higher
- **flex** 2.5.4 or higher
- **bison** 1.25 or higher

This code may be compatible with lower versions of these tools. The lexical analyzer code is compatible with solaris *lex* tool. It may also be compatible with other tools. The parser code is not tested for other tools like *yacc*.

In order to install, first *configure* script is run, which generates a few files required for the compilation. In order to compile Fsimg, "make" is executed and to install "make install" is executed. This step installs Fsimg in the Fsimg-0.1 sub-directory within the source directory. In order to install it in another directory "–prefix=full-install-path" option may be added to the configure before compilation. Following is the contents of installed directory.

- **bin** - This directory contains the tools *fsimg*, *irview*, and *elfview.* irview and elfview are tools to look into ir and elf files.
- **lib** - This directory contains the library fsim. You have to provide the -lfsim option as well as path of this directory as a library search path while compiling Fsim.
- **include** - This directory contains the **C** header files needed by the Fsim. You have to include the path of this directory in the compiler include search path option to search this directory for header files while compiling Fsim.

Command Line Options:

Options given in [ ] are optional.

-i filename	:	Input IR file.
-e filename	:	Input ELF file.
-p pc-name	:	Name of the variable used as a program counter in the Sim-nML description.
-P cpc-name	:	Current PC name.
-m mem-name	:	Name used for memory in the Sim-nML description.
-s sp-name	:	Name used for the stack pointer in the Sim-nML description.
[-c config-file]	:	Dynamic function configuration file.
[-S size]	:	Initial stack size. Negative value to size indicates stack growth from lower address to higher address. Default higher address to lower address.
[-f call-node]	:	Call instructions. Call-node is the top *and* or *or-rule* node of call instructions in the description.
[-t]	:	Do not load text into the memory. Default is load text.

Output:

Fsimg generates following files *Instr.c, Funcs.c, Fsim.c, Defs.h, Vars.h, and Types.h.*

Compiling Fsim:

Compile the file *Fsim.c* with -lfsim and giving include search path as the *include* directory and the library search path as *lib* in the directory where Fsimg is installed. The functions for dynamic calls and any extra canonical functions have to compiled along with it.

For this purpose a "makefile" is provided in the directory where Fsimg is installed. One can edit this file to add the files(s) containing user and canonical functions to be compiled into the Fsim.

Example:

To compile Fsim manually with gcc assuming Fsimg is installed in /home/yschand/Fsimg-0.1 is as follows.

```
% gcc Fsim.c lib_dynfuncs.c canonical.c -o fsim -I. -lfsim
-I/home/yschand/Fsimg-0.1/include -L/home/yschand/Fsimg-0.1/lib
```

An option -DTRACE if given in above compilation command causes the trace to be generated. Similarly -DICOUNT option is added to configure Fsim to find out total instructions simulated. By adding -DICOUNT option defines an integer variable *Count* which can be printed at the end of simulation through the implementation of *exit* function.

To use the "makefile" first the variables DYNLIBOBJ and DYNLIBSRC are set to the file name of user functions file. Following example shows this.

```
DYNLIBOBJ = lib_dynfuncs.o
DYNLIBSRC = lib_dynfuncs.c
```

Similarly the canonical file name can be set as shown below.

```
CANONOBJ = canonical.o
CANONSRC = canonical.c
```

The "makefile" implements all the three options as described above. Running "make" results in the functional simulator fsim. Running "make count" generates the simulator which can give the count of the number of instructions simulated, and running "make trace" generates the execution trace.

Various other tools and information like IR generator, PowerPC 603 specification etc. can be obtained from `ftp://cse.iitk.ac.in/pub/moona` FTP site.