Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation

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

by
Rajiv A.R

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

December, 1999

Certificate

This is to certify that the work contained in the thesis entitled "*Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation*", by *Rajiv A.R*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

December, 1999

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

Abstract

The design process for modern embedded systems requires automated modeling tools for faster design and for the study of various design trade-offs. Such tools 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. **Sim-nML**[15] is one of the specification languages used for developing processor performance model.

In this thesis, we have developed the following towards the integrated environment.

- Designed and extended an *Intermediate Representation (IR)* of a processor specification written in *Sim-nML*. The *IR* is simple and facilitates the development of various tools such as assembler, disassembler, compiler back-end generator, instruction set simulator, trace generator, profilers etc. based on the processor specification.
- *IR-Generator*. It takes a processor specification written in *Sim-nML* and produces its intermediate representation.
- *Cache Simulator*. This provides a mechanism to simulate various caching policies. The designer can use the simulator to study the trade-offs between different caching policies.
- *Code Instrumentor*. This implements a mechanism to perform analysis and profiling of application programs through the technique of *code instrumentation*.
- *Motorola 68HC11* processor specification in *Sim-nML*.

The *Cache Simulator* and the *Code Instrumentor* were implemented on top of the *Retargetable Functional Simulator*[1]. They provide an architecture independent way of constructing profiling and analysis tools.

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, Dr. Sanjeev Aggarwal and Kshitiz Krishna for their valuable suggestions and comments during the 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 the Department of Computer Science and Engineering, IIT Kanpur. I express my gratitude to Cadence India Ltd. for their support through fellowship.

I would like to thank my seniors and fellow Cadence members specially V.Rajesh, Y. Subhash Chandra for their help throughout my work here. I would also like to thank other members of the Cadence Research Group, Prashant Pogde and Sarika Kumari. I am greatly indebted to all my comrades of the mtech'98 batch for giving me a great company during my stay here.

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

In the design of embedded systems, the use of automated modeling tools is gaining momentum. They yield fast turn-around time with lower costs for the system design and simplify the process of design changes. In the past, many 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. This way, these tools also help in studying the impact of various hardware-software co-design trade-offs. For a designer of the system, such tools are useful as they allow him to explore several alternatives early in the design phase. The benefits of such high-level processor models and processor development methodology include the availability of application development tools, simulation tools and profiling tools even before the processor is ready. An unified processor model for the generation of application development tools, profilers and simulators not only reduce the effort required but also eliminates the chances of discrepancies among different descriptions.

The *Sim-nML*[15] language is used as a model to develop an integrated processor development environment. The integrated environment would include tools like assemblers, disassemblers, compiler back-end generators, functional simulators, cache simulators, profilers, hardware synthesizers etc. The instruction set architecture of the processor at hand is described in *Sim-nML* from which these tools are generated automatically. For this purpose, we have designed an *intermediate representation*[1] for the *Sim-nML* language. The *IR* is simple but powerful enough to facilitate the development of various tools based on the processor specification. The *IR* has been designed to ease the burden of each tool to parse the *Sim-nML* language which is tedious and redundant. The *IR* encapsulates the *Sim-nML* description in a set of tables. This would allow the tools to easily extract the relevant information. We have designed a tool, *IR-generator*, which takes a processor specification in *Sim-nML* language and provides the intermediate representation of the processor model as output.

In addition, a *Cache Simulator Environment* has been developed. This would help the processor designer to study the trade-offs of implementing various caching policies for the application to be run on the processor under development.

We have also developed a profiling tool through code instrumentation mechanism. This provides the designer with code instrumentation mechanisms at the procedure level, or at the basic block level or at the instruction level. The designer can use this mechanism to study the run-time behavior of the application on the targeted processor.

Since *Sim-nML* provides a generic way of describing a processor architecture, the cache simulator and profiler generator constructed from *Sim-nML* specification allow a flexible and architecture independent way to generate profiling and analysis tools. Currently, most of such tools are architecture dependent which necessitates the development of separate set of tools for each processor model. It can be avoided with the *Sim-nML* model as once the description is ready the tools could be automatically generated and customized. Moreover, currently these profiling tools also require processor support. Our tools help to do the same without such support or even before the actual processor fabrication is done.

This work is a continuation of the *Retargetable Functional Simulator*[1] work done by **Y. Subhash Chandra**. An initial version of the *IR*[5] was designed and implemented by **Nihal Chand Jain**. The code instrumentation mechanism was inspired from ATOM[13].

[1]from now on we use the term *IR* to refer to the *intermediate representation.*

1.2 Related Work

Performance modeling of a system is a growing area and a lot of research has been pursued in this area. These previous works have resulted in a set of performance modeling tools using different languages for processor specification.

Instruction Set Description Language (**ISDL**)[4] 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. These are captured in separate sections. Currently an automatic assembler generator has been developed.

Specification language for encoding and decoding (**SLED**)[11] 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.

Visualization based Microarchitecture Workbench (**VMW**)[14] is an infrastructure which facilitates the specification of instruction set architecture and microarchitecture of a machine in a concise manner. *VMW* provides all necessary infrastructure software to the designer, including generic simulation software, visualization support software and graphical user interface software. VMW automatically integrates the machine specification and infrastructure software to generate a customized performance simulator based on the trace-driven simulation approach. Thus *VMW* provides a powerful environment for modern superscalar processor design.

SimOS[9] 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.

ATOM[13] provides a frame work for providing customized program analysis tools. It provides a common infrastructure provided in all code-instrumenting tools.

ATOM organizes the final executable such that the application program and user's analysis routines run in the same address space. *ATOM* uses no simulation or interpretation. It has been used to build a diverse set of tools for basic block counting, profiling, dynamic memory recording, instruction and data cache simulation, pipeline simulation, evaluating branch prediction and instruction scheduling.

Pixie[12] is a utility that allows you to trace, profile or generate dynamic statistics for any program that runs on a MIPS processor. It works by annotating executable object code with additional instructions that collect the dynamic information during run time.

Dinero IV[2] is a trace driven uniprocessor cache simulator for memory reference.

QPT[6][7] is profiler and tracing system. It rewrites a program's executable file (a.out) by inserting code to record the execution frequency or sequence of every basic block or control-flow edge. From this information, another program QPT_STATS can calculate the execution cost of procedures in the program.

EEL[8] (Executable Editing Library) is a C++ library that hides much of the complexity and system-specific detail of editing executables. EEL provides abstractions that allow a tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding foreign code. EEL greatly simplifies the construction of program measurement, protection, translation, and debugging tools.

1.3 Goals Achieved

In this work, we aimed at the development of an integrated environment for processor performance modeling using *Sim-nML*. The development of the 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 thesis are as follows.

- *Intermediate Representation (IR)* for *Sim-nML* language specification is extended. This is simple but powerful enough to facilitate the design of various processor specific tools. This was an extension of an earlier version([5]).

- *IR-Generator* which takes a processor specification in *Sim-nML* language and provides an intermediate representation of the processor specification as output was extended and implemented. This was an extension of an earlier version([5]).

- A *Cache Simulating Environment* has been developed to provide a basis for benchmarking various caching policies of a given processor.
- A *Code Instrumentation Mechanism* has been developed for implementing various profiling techniques.
- *Motorola 68HC11 Specification in Sim-nML*
  Model for **Motorola 68HC11**[17] processor has been developed in *Sim-nML*. All the instructions have been specified with a simple resource usage model.

1.4 Organization of Report

The rest of the report is organized as follows. In **Chapter 2** we give an overview of the *Sim-nML* integrated environment. In **Chapter 3**, we discuss the design and implementation of the *IR*. In **Chapter 4**, we look at the *Cache Simulation Environment*. In **Chapter 5**, the we discuss the *Code Instrumentation Mechanism*. A brief overview of *Motorola 68HC11* processor is given in **Chapter 6**. Finally, we conclude in **Chapter 7**. In **Appendix A** we describe the *Sim-nML* grammar and give the *IR* format in **Appendix B**.

Chapter 2

The Sim-nML Integrated Environment

2.1 Overall Structure

The base language for our environment is *Sim-nML*, a generic processor modeling language. *Sim-nML* is an extension of *nML* machine description formalism([3]). 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)*. For a tool, intermediate form is simpler and easier to read and interpret when compared to a specification in *Sim-nML*. The overall view of the environment is shown in the figure 2.1.

2.2 Sim-nML Language

2.2.1 Sim-nML Model

Sim-nML[15] is an extensible formalism designed to specify generic single processor models. *Sim-nML* works at two levels of abstraction. The processor described by the language could either be an existing one or an application specific processor being developed. The designer team, depending on the application for which the processor is being modeled, would either choose for an off-the shelf processor or design a new

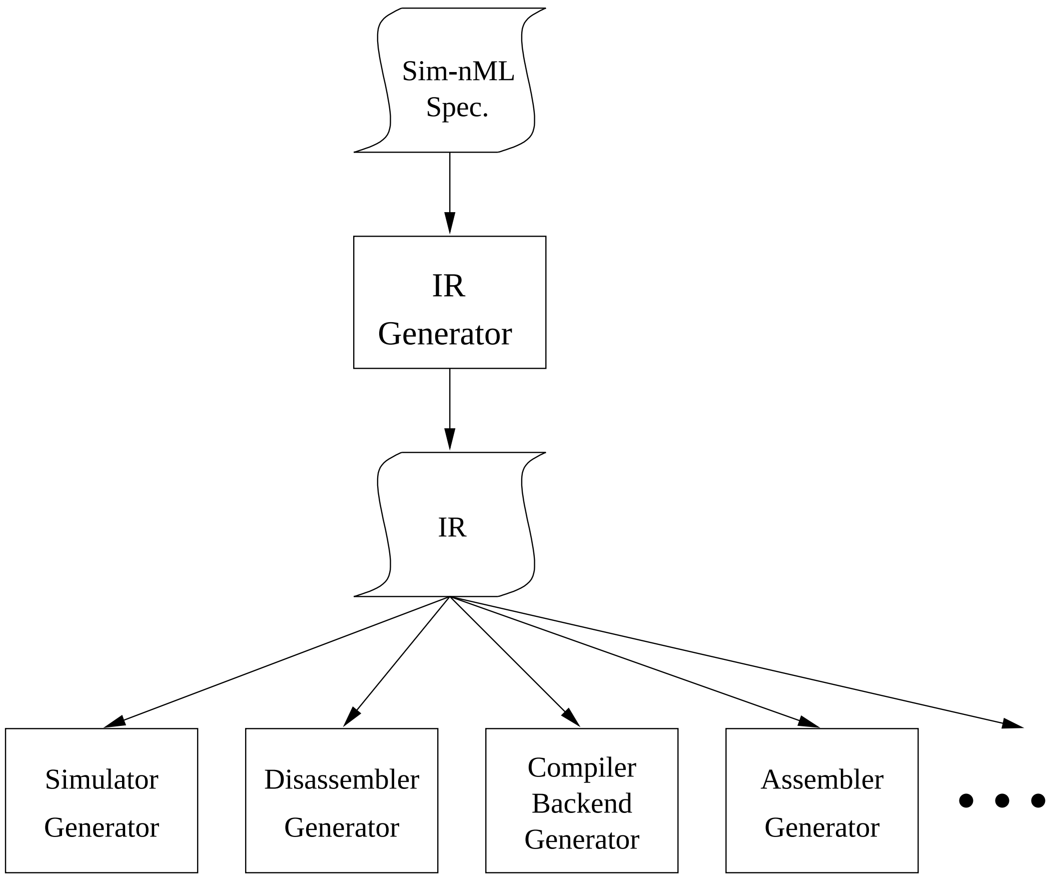

Figure 2.1: A View of Integrated Environment

one. At the functional level, the designer typically has an overview of the instruction set that the processor should support to meet the application requirements. The *Sim-nML* model is used to give an ISA level description which is the application programmer's model of the processor. While modeling an existing processor the designer would have the processor instruction set manual which he could use to describe the instruction set, both the syntax and semantics, in *Sim-nML*. If a processor does not exist, the *Sim-nML* language could be used to describe the intended instruction set semantics.

At an alternate level of abstraction, *Sim-nML* could be used to expose the microarchitecture details. This could be used to detail out the various units within a processor along with a timing estimate. When describing an existing processor, the designer could abstractly specify the processor's pipeline features, functional units etc. The associated timing estimates could help the designer to evaluate multiple processors in order to choose the optimal model satisfying the application at hand. When designing a new processor model, the designer could give a rough estimate of the microarchitecture features that he would like to incorporate along with the associated timings. The process could be iterated until a satisfactory description is obtained that meets the timing requirements of the application.

Sim-nML is an attributed grammar[1] with some predefined but optional attributes like *image, syntax, action and uses.* The instruction set is described in a hierarchical manner with fragments of each of the attributes being distributed over the whole grammar tree. The common behavior of a class of instructions is captured at the top level of the tree and the specialized behavior of the sub-classes are captured in the subsequent lower levels.

2.2.2 Sim-nML Grammar

Sim-nML grammar has a fixed start symbol namely *instruction* and two kinds of productions, namely *or-rule* which looks like,

op = n_0 | n_1 | n_2 | ...

and *and-rule* which looks like

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, each t_i is a token. Each a_i is an attribute name and e_i their respective definitions.

The *Sim-nML* grammar predefines four attributes - *syntax, image, action, uses.* The *syntax* describes the assembly language format of the instruction, *image* describes the binary coding of the instruction, *action* describes the semantics of the instruction while the *uses* describes the *resource-usage model.*

The *Sim-nML* grammar in example 1 (figure 2.2) describes a simple processor with two instructions - **add** and **multiply**. All of these attributes are used for adding and multiplying the contents of two general purpose registers respectively. **PC** refers to the address from which the next instruction has to be fetched. *Sim-nML* supports a special token, **$**, which is used to denote the address of the instruction in the definition of various attributes.

[1]an attribute grammar is a context free grammar in which for each non-terminal a fixed set of attributes and for each production a set of semantic rules is given. In grammar all non-teminals have to be derivations. So we don't diffrentiate between productions and non-terminals.

```
type addr   = card(32)
type byte   = card(8)

let REGS = 32
let byte_order = little

mem PC  [1, addr]
mem M   [2 ** 32, addr]
reg  R  [REGS, byte]
var tmp [1, byte]

resource Fetch_Unit, Exec_Unit[2], Retire_Unit

mode REG(index : card(5)) = R[index]
syntax = format("0%3b", index)
image  = format("R%d", index)

mode MEM(Addr : addr) = M[addr]
syntax = format("1%32b", addr)
image  = format("R%d", addr)
mode ADDRMODE = REG | MEM

op instruction(x  : binaction)
uses   = Fetch_Unit #{2}, x.uses, Retire_Unit #{2} : action
syntax = format("%s", x.syntax)
image  = format("%s", x.image)
action = {
        x.action;
}
```

Figure 2.2: Sim-nML Specification for a Simple Processor

The basic types of *Sim-nML* include *card, int, bool, enum* etc. The *type* declaration is used to declare derived types. Addressing modes in the processor are described using *mode* rule. In the above example, mode rule **REG** denotes register addressing mode where **R[i]** denotes the i^{th} register of the register file **R**. The *var* declaration is used to declare temporary variables. This processor assumes 2 instances of resource **Exec_unit**, one of which is held by the instruction under execution. If one instance

```
op binaction = plus | multiply

op plus(src : ADDRMODE, dst : ADDRMODE)
uses   = Exec_unit #{2}
syntax = format("add %s %s", src.syntax, dst.syntax)
image  = format("1010010%s%s", src.image, dst,image)
action = {
        dst = src + dst;
        PC = PC + 9;
}

op multiply(src : ADDRMODE, dst : ADDRMODE)
uses   = Exec_unit #{6}
syntax = format("mult %s %s", src.syntax, dst.syntax)
image  = format("0010101%s%s", src.image, dst.image)
action = {
        dst = src * dst;
        PC = PC + 9;
}
```

Figure 2.3: Sim-nML Specification for a Simple Processor: continued

is already acquired, then another instruction in the pipleline can acquire the second instance. The following instructions remain stalled till one of the resource instance is released. This models a simple superscalar processor with 2 execution units.

2.2.3 Resource Usage Model

The micro-architecture details of the processor can be specified using the *resource-usage model*. *Sim-nML* assumes that entities within the processor like the functional units, pipeline stages, registers, ports etc. constitutes a set of resources. The resources can be acquired/released by any instruction in execution. The *resource-usage model* is based on the assumption that at any instant, an instruction in execution, holds some set of resources and does some action. The resources held by the instruction and the action taken change progressively.

In the *resource usage model*, the resource is an abstraction of a piece of hardware such as registers, ALUs, functional blocks etc. for which instructions contend and pipeline flow is nothing but a way of resolving such conflicts. When two instructions wait simultaneously for a single resource, the conflict will be resolved by FIFO order, i.e, the instruction that entered the pipeline earlier will be alloted the resource. This model is powerful enough to describe pipelines, superscalars and other microarchitectures. The *uses* attribute describes the *resource usage model* and the action taken when the resource is acquired or released for an instruction. In Example 1 (figure 2.2), the *resource* definition is used to define the functional units like **Fetch_Unit, Exec_Unit, Retire_Unit**. It specifies that all instructions use the **Fetch_Unit** for 2 units of time, the **Exec_Unit** for time depending on the type of instruction - 2 units for *plus* instruction while 6 units for *multiply* instruction and the **Retire_Unit** for 2 units of time. The token *action* at the end of *uses* specifies that after the specified resources are used for the given time period, the function specified in the *action* attribute is performed. The unit of time can be thought of as a machine clock cycle although it is not imposed by *Sim-nML*. If an unit of time is same as machine clock cycles, then we can estimate the number of clock cycles taken by the program.

2.2.4 Specification of register ports

Processors implement registers as register files with multiple read-only or write-only ports. Access restrictions are imposed on registers within the register file depending on hardware implementation. Superscalar processors allow multiple instructions to be at the write-back stage. Two instructions with the same destination register should block and execute in the program imposed order (WAW, Write-after-Write hazards). Similarly, an instruction could read from a register which is being simultaneously written by another instruction (WAR, RAW hazards). Multiple instructions however could be allowed to read the same register as long as read ports are available. These hazards can be modeled in *Sim-nML* with the *uses* attribute.

Assume **Rfile** is a register file with 32 registers, 32 bit each. Rfile is declared to have 3 read ports and 2 write ports as follows.

```
reg Rfile[32, card(32)] ports = 3, 2
```

This would implicitly declare 2 resources **Read_Rfile[3], Write_Rfile[2]** which represent the read and write ports of the register file **Rfile** with 3 and 2 instances respectively. Apart from this, each register in the register file is assumed to have port resource instances equal to the number of read ports of the register file, 3 in this case.

Assume an instruction with register **R**[**0**] as the source register. This could have the following uses attribute

```
uses = Read_Rfile, R[0]
```

This implies that two resources are to be acquired, one resource is **Read_Rfile** and the other is any one of the port resource of register **R**[**0**]. Before reading the register contents, any free read register port **Read_Rfile** is acquired followed by register **R**[**0**] itself.

To model a write to register **R**[**0**], we could write

```
uses = Write_Rfile, R[0][]
```

where a single instance of **Write_Rfile** port while all port resource instances of register **R**[**0**] are acquired. This would prevent another instruction from accessing register **R**[**0**] while it is being written.

2.3 Current Work

Following tools have been implemented till now in our environment.

Instruction Set Simulator Generator [15] 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 [5] takes *Sim-nML* processor specification and a binary for that processor in **ELF** format and gives out the symbolic disassembly of the binary which can be assembled back to the binary.

Compiler Back-End Generator [10] takes *nML* specification and generates *LCC* machine description which can used to generate the *LCC* compiler for the specified processor.

Retargetable Functional Simulator [1] generates a functional simulator for a particular program to be run on a given processor description in *Sim-nML*.

The following tools are under development

Timing Simulator to analyze a particular program for timing performance and resource usage. A compiled code simulator generator would generate a higher performance timing simulator.

Assembler generator for a given processor described in *Sim-nML* is being developed. This would generate an assembler, confirming to *GNU* assembler syntax, which could produce *ELF* object code of the input assembly language program.

Compiler Back-End Generator to generate back-end for *GNU-C* by automatically generating *GNU*'s *md* file format description of a particular processor from *Sim-nML*.

As part of this thesis work, *Sim-nML* specification for *Motorola 68HC11*[17] was written. Its a simple 8-bit processor. This description was tested over the *Retargetable Functional Simulator*[1] and the *generic disassembler*[5]. Earlier, *Sim-nML* specifications for *PowerPc603* processor[1] and for *Intel 8085* were written.

Chapter 3

Intermediate Representation of Processor Models

One part of this thesis involves the development of an *Intermediate Representation (IR)* of the processor model. We developed a tool, *IR-Generator*, which takes a processor specification written in *Sim-nML* language as input and produces corresponding intermediate representation of the processor specification as output. In order to have the intermediate representation usable by all front-end tools such as disassembler, assembler, simulator etc., certain goals were setup behind the design of the IR as listed below.

The *IR* should

- be as simple as possible.
- should not lose any useful information which is available in the original input *Sim-nML* specification.
- not have any unnecessary or redundant information.
- be easy to understand and use.
- be easy and efficient to retrieve the required information.
- be flexible and extensible.
- facilitate the design of various processor specific tools such as assembler, disassembler, simulator, trace generator, compiler back-end generator etc.

3.1 Overview of Earlier Work on IR

The *IR* was designed and implemented as part of a master's thesis by **Nihal Chand Jain**[5]. The *IR-generator* so designed had 2 parts - the parser and the flattener. In the first phase, the parser parsed the *Sim-nML* input specification and collected the relevant information in tables. The flattener would then simplify the hierarchy.

In *Sim-nML*, information about an instruction is composed of fragments that are distributed over the whole specification tree with the root node named as *instruction*. To get information about one particular instruction, a complete path from root node to a leaf node is traversed with proper parameter substitution at all levels of the tree. If all such paths are traversed, then information about all possible instructions are obtained. This process is called flattening of the tree. All references to *or-rules* are eliminated from the *or-rule* and *and-rule* defintions. Elimination of *or-rule* parameters from an *and-rule* definition results in generation of new *and-rules*. All attributes of the *and-rule* remain unchanged in the new *and-rules*. To make the *IR* compact, these new *and-rules* were treated as *sub-rules* of the original *and-rule*. In the *IR*, all *sub-rules* of an *and-rule* were stored along with the *and-rule* itself. The references for the attributes in the *and-rule* were not duplicated for *sub-rules*.

The *syntax* and *image* attribute defintions were then flattened with parameter substitution and added to the syntax and image tables respectively. Corresponding to each instruction node in the hierarchy all possible instructions that can be generated were flattened and added to the syntax and image tables. The hierarchical information was maintained using *dot exprssions* while the parameter details in syntax and image fields of an instruction was represented using 3-tuples. So from a particular node in the hierarchy tree, the images and syntaxes of each subtree could be listed. The topmost node *instruction* would enumerate all possible syntaxes and images specified for that processor. The parsed and flattened information extracted from *Sim-nML* was then dumped onto tables in the output *IR*. The important tables include the *identifier table*, *and-rule table*, *memory table*. Each of these tables had a unique key to refer to each entry. The *identifier table* table had an *id-key* for each identifier. The other tables referred to each identifier using this key.

3.2 Shortcomings of Earlier IR

The earlier version of the IR[5] and *Sim-nML* was found to have the following shortcomings due to which they were extended.

- disallowed the use of expressions in **format** definitions in *syntax* and *image* attributes.
- expressions were not allowed in bit selection operators.
- each resource specified in *Sim-nML* was assumed to have a single instance.
- register and memory ports were not supported.
- the *IR* tables used multiple levels of indirection to retrieve information.
- information about the endian-ness of the generated *IR* could not be found in an easy manner.
- flattening resulted in loss of information which were required by the tools during instruction matching.

3.3 Design of an Intermediate Representation

A processor specification in *Sim-nML* language is a human readable text file. Several constructs are provided in *Sim-nML* to enhance the clarity and readability of the description. In order to retrieve the desired information from such a description, a tool needs to perform parsing of input, variable substitution etc. An intermediate representation helps in reducing such extra burden on the tool. Thus we need an intermediate representation keeping previously mentioned goals in mind. In this section, we will discuss the design of the *IR* in detail.

3.3.1 Simplification of Information by Substitution

The *Sim-nML* language allows the constant definition using *let-specification* (eg: **let REGS = 32**). In the *Sim-nML* specification file, wherever a constant is referenced, its value is substituted in the *IR*. For example, value of the constant **REGS**, i.e. 32, is substituted whereever **REGS** is used in the example given in figure 3.1. Thus constants are not referenced in the *IR* of the processor specification. Therefore all such constant declarations can be eliminated from the *IR*. However, some constants might be used by the tools, i.e., constants like **byte_order** may be used by tools to define the byte ordering of a processor. So the *IR* retains information about all constant declarations.

The *Sim-nML* language defines some basic data types and allows new data type definitions using basic data types and previously defined user data types. Since all user defined data types can be built using only basic data types, all variables are redefined with only basic data types in the *IR*. Thus all user defined data type declarations can be eliminated from the *IR*. For example in figure 3.1, **index** is used to refer to data type **card(2)**. All occurrences of **index** can be replaced by **card(2)**.

There are some other constructs in *Sim-nML* which are simplified in the *IR*. For example, names of *Sim-nML* memory variables, op-rules, attributes, parameters in *and-rules* are replaced by unique identifiers and everywhere the corresponding identifier is used for the reference. *Sim-nML* allows the use of identifier names for *op-rules* even before they are defined. This necessarily requires a tool to do multiple passes over the processor specification. Many of these identifiers are not significant at all (for example, parameter names). In the *IR*, all significant identifiers are maintained in an *identifier-table* and the index into the *identifier-table* is used for the reference. It simplifies the information retrieval from the *IR*.

3.3.2 Representation of Attribute Definition

In the *Sim-nML* processor specification, memory variables, *mode-rule* and *op-rule* declarations define attribute names and their definitions. The attribute definition is either an expression consisting of various operands and operators, or a sequence of statements separated by a semicolon. Each of these statements might be a simple assignment statement or a conditional statement or a function call or a use of an attribute from another related *op-rule*. (Refer to Appendix A for *Sim-nML* grammar)

For *syntax* and *image* attributes, definitions could be an expression which evaluates to a string. The *and-rule table* entry corresponding to these *op-rule*s would contain an index into the *syntax table* and *image table* respectively. In the *IR*, a record is stored for each *syntax* and *image* attribute definition of an *and-rule*. The record includes a string value corresponding to the expression. The string values are evaluated as in figure 3.3.

```
type index = card(2)
let REGS = 32
resource eunit[2]

reg  PC[1, card(32)]
mem R[REGS, card(32)]
mem MEM[1024, card(8)]

mode SHORT = MEM | REG

mode MEM(i:index) = M[R[i]]
syntax = format("(R%d)", i)
image = format("0%5b", i)

mode REG(i:index) = R[i]
syntax = format("R%d", i)
image = format("1%5b", i)

op instruction(x:instr_action)
syntax = x.syntax
image = x.image

op instr_action = alu_op | move_op
op alu_op(src:SHORT, dst:SHORT, aa:alu_action)
syntax = format("%s %s,%s", aa.syntax, src.syntax, dst.syntax)
image = format("1%b %b %b", aa.image, src.image, dst.image)
```

Figure 3.1: Sim-nML Program for a Hypothetical Processor

An exception to this encoding is when the *syntax/image* attribute is encoded as *SYNIMGDOT_TYPE* (refer Appendix B for *IR* types) (like P_i*.image* or P_i*.syntax*), then the *and-rule table* entry for that op rule would contain the parameter number i and the index into the attribute table corresponding to the defined attribute (refer Appendix B).

For the example *Sim-nML* processor model in figure 3.2 the *syntax table* entry for **move op rule** would be

```
move %s{0.3}, %s{1.3}
```

```
op alu_action = a_add | a_sub | jmp

op a_add()
syntax = "add"
image = "0"

op a_sub()
syntax = "sub"
image = "1"

op move_op = move | store
op move(src:SHORT, dst:SHORT)
syntax = format("move %s, %s", src.syntax, dst.syntax)
image = format("00%s%s", dst.image, src.image)

op store(src:SHORT, dst:SHORT)
syntax = format("move %s,%s", src.syntax, dst.syntax)
image = format("01%s%s", src.image<2..3>, dst.image)

op jmp(dst:card(32))
syntax = format("jmp %32b", dst)
image = format("101%32b", $+dst)
```

Figure 3.2: Sim-nML Program for a Hypothetical Processor: continued

where *src* is parameter 0 and *dst* is parameter 1. 3 stands for index into the attribute table which corresponds to the attribute *syntax*.

Similarly the *syntax table* entry corresponding to the **MEM op rule** (figure 3.2) is

`(R%d{0.-1})`

where 0 represents the first parameter (i) and since i is of a basic type (card in this case), the second integer is -1.

In a similar way, the *image table* entry for **jmp op rule** would be

`101%32b{-14.3}`

where -14 indicates an index to the *prefix attribute table* where the expression `$ + src` is stored. This expression is stored as a 3-tuple in the *prefix attribute table* (and hence the second entry is 3). $ is stored in the *prefix attribute table* as offset into the

- For a simple string it is placed as it is in the syntax/image table.
- If **format** declaration is used, for each format quantifier(like %s, %nb, etc.) a 2-tuple of the form **{X.Y}** denoting the corresponding defining parameter P_i (where P_i denotes the *ith* parameter starting from left of that *and-rule*) is embedded in the *syntax* or *image* table as follows.
  1. if the parameter P_i is a basic type(like int, card, bool) then **X** would denote the parameter number i while **Y** would be **-1**.
  2. if the parameter P_i is of an *and-rule* or *or-rule* type and is specified as P_i*.image* or P_i*.syntax* then **X** would denote the parameter number i while **Y** would be the index into the attribute table corresponding to the defined attribute.
  3. else if its an expression, then negative of **X** would be the index into the *prefix-attribute table* while **Y** denotes the number of such tuples.

Figure 3.3: Evaluating syntax and image attributes

string table, *src* is stored as *PARA_TYPE* with value 0 (parameter number).

The *image table* entry for **move op rule** would be

`01%s{-8.6}%s{1.4}`

where -8 denotes an index to *prefix attribute table* where the expression `src.image<2..3>` is stored as a 6-tuple which includes the parameter number, attribute index, and the range parameters (2 and 3).

Other attributes in *Sim-nML* are used to hold semantic action associated with the instruction. For example, to simulate the behavior of an instruction, attribute definition of *action* attribute is used. A tool such as the instruction set simulator could be made to run faster if such attribute definitions are represented differently. Usually expressions inside an attribute definition are written in an infix notation using priority and associativity rules to decode an expression uniquely. However, prefix or postfix notation is better for faster evaluation as the priority and associativity becomes implicit.

In the *IR*, prefix notation is used for all attribute definitions except *syntax* and *image* attributes. Using such a representation, tools like simulator, trace generator, compiler back-end generator etc. can be made to run fast.

3.3.3 Structure of the Intermediate Representation

The structure of the *IR* (refer Appendix B) should be capable of storing information about constants, identifiers, *or-rules*, *and-rules* and information about attributes such as *syntax*, *image*, *action* etc. They are represented in various fixed sized and variable sized data structures.

The *IR* structure is essentially a collection of various tables. Information of each type is stored in a different table. The entries in most of these tables are fixed size records. However, some tables hold variable size records. For an easy access to the tables, a *meta table* is also added in the *IR* which contains the location and name of all the tables. This simplifies the access mechanism for all tables. In brief, the *IR* consists of the following tables

- **Meta table** . This is a table of contents having a road map to know about the location and name of other tables in the IR.
- **Constant table** . This table holds all constant declarations in the *Sim-nML* processor specification. For the example given in figure 3.1, this table will contain the following.

  ```
  (Name  Type    value)
  REGS   CONST_TABLE_INT_TYPE    32
  ```

- **Resource table** . This table holds the names of the resources which are declared with *resource* declaration along with the number of instances of each resource. For the example given in figure 3.1, this table will contain the following.

  ```
  (Name Num)
  eunit 2
  ```

- **Attribute table**. This table holds the name of all distinct attributes used in the input processor specification. For the example given in figure 3.1, this table will contain the following.

  ```
  (Name)
  syntax
  image
  ```

- **Identifier table** . This table holds the name of all the identifiers (other than those specified in the **constant table** and the **resource table**). An identifier can be of *MEM_TYPE, MODE_OR_TYPE, MODE_AND_TYPE, OP_OR_TYPE, OP_AND_TYPE* etc. Depending on the types, the index into the corresponding *memory-table*, *and-rule table* or *or-rule table* is stored. For the earlier example, the following is the contents of the *identifier table.*

(Index	Name	Type)
0	PC	MEM_TYPE
0	MEM	MODE_AND_TYPE
1	REG	MODE_AND_TYPE
0	SHORT	MODE_OR_TYPE
2	instruction	OP_AND_TYPE
1	instr_action	OP_OR_TYPE
3	alu_op	OP_AND_TYPE
3	move_op	OP_OR_TYPE
2	alu_action	OP_OR_TYPE
4	a_add	OP_AND_TYPE
5	a_sub	OP_AND_TYPE
6	move	OP_AND_TYPE
7	store	OP_AND_TYPE

- **Memory table** . This table holds the information about all memory variables declared with a *reg*, *mem* or *var* declaration. It includes index into the *identifier table*, type and size of the data and information to locate various attributes (of the variable) stored in other tables. For the example shown in figure 3.1, the following is the partial contents of the *memory table.*

(Name	data-type	type	size	value1	attribute)
R	CARD_TYPE	MEM	32	32	-
M	CARD_TYPE	MEM	1024	8	-

Note that instead of storing the name of memory variable (i.e. `R`), the index into the *identifier table* is used.

- **Or-Rule table** . This table holds the information about children of all *or-rules.* The following is the partial contents of the *or-rule table* for the example shown in figure 3.1

  ```
  (Name            total_children         integer table index)
  instr_action        3                 97 (<AND_RULE_TYPE, 3(and table index)>, ...
  ```

  Note that instead of storing the name of *or op-rule* variable (i.e. `instr_action`), the index into the *identifier table* is used.

- **And-Rule table**. This table holds the information about all *and-rules.* It also holds the information to locate the attribute definitions stored in other tables. The following is the partial contents of the *and-rule table* for the example shown in figure 3.2

  ```
  (Name   total_para  total_attr  integer table index(attribute)
  store     2            2         72(<syntax,7(syntax table offset),0(len)>, ...)

  integer table index(parameter))
   78(<OR_RULE_TYPE, 0(or rule table offset), 0>, ...)
  ```

  Note that instead of storing the name of *and op-rule* variable (i.e. **store**), the index into the *identifier table* is used.

- **Syntax table** . This table holds the syntax record associated with the *syntax* attribute definitions of all *and-rules.* It also holds the information to associate the correspondence between the *and-rule table* and the *syntax table.*

- **Image table** . This table holds the image record associated with the `image` attribute definitions of all *and-rules.* It also holds the information to associate the correspondence between the *and-rule table* and the *image table.* It holds records similar to the *syntax table.*

- **String table** . This table is used for storing variable length strings (null terminated) such as identifier names. This table helps in having fixed size entries in other tables. Identifier names and strings in other tables are stored as offsets into the string table.

- **Integer table** . This table is used for storing only integer values. These integers are associated with other tables and represent different meanings in different contexts. This table helps in having fixed size entries in other table. For example in the *or-rule table* each child is represented with 2 values. The first corresponding to whether the child is an **OR-RULE-TYPE** or **AND-RULE-TYPE** and the second an index into the *identifier table.* The 2 values are stored as 2 integers in the *integer table.* The number of such 2-tuples correspond to the number of children for each *or-rule* type. Hence for *n* children there would be *2n* integer values stored in the *integer table* at the specified offset of this *or op-rule.*

- **Prefix-Attribute-Definition Table** . This table holds the attribute definition of all the attributes (except *syntax* and *image* attributes) associated with memory-variables and *and-rules.* These definitions are stored in prefix notation. Other tables store the information to locate the appropriate attribute definition correctly.

A header is prepended to the *IR* which consists of 2 fields: a four byte magic number which is currently initialised to **"IRV2"** and a field to indicate whether the *IR* file format is in **big-endian** or **little-endian** format.
In Appendix B, we present the structure of each of the tables in detail.

The conversion from *Sim-nML* to the *IR* is done in the following two passes.

3.3.4 Pass 1 : Macro Preprocessor

The *IR* does not retain any macro definitions from the source. For ease of implementation, macro processing is implemented as a separate pass over the *Sim-nML* specification file. This part has been done in another project by Y. Subhash Chandra[1]. The macro preprocessor takes the *Sim-nML* file with macro definitions as input and produces a *Sim-nML* file without macros. It gathers all macro definitions and converts them into equivalent *m4*[16] macro definitions. Then *m4*, a standard utility available on *Unix* platforms, is run on this file to get the *Sim-nML* file without any macros.

3.3.5 Pass 2 : Parsing the Hierarchy

Pass two takes a *Sim-nML* specification file without macros as input and produces the specification in the *IR*. This pass proceeds in two phases.

- The first phase involves the parsing of input file. During the parsing, all relevant information is gathered in appropriate data structures. Attribute definitions for all attributes except *syntax* and *image* attributes are converted into prefix notations during the parsing time. As soon as a definition is complete, it is stored in the prefix-attribute-definition table.
- In the second phase, the syntax and image table entries are created with appropriate 2-tuples added to define each parameter.

At the end of the second pass, all tables are written in the output file while updating the *meta table* to include information relevant to other tables.

Chapter 4

Cache Simulation Environment

As a second part of this thesis work, we implemented a *Cache Simulation Environment* for processor models described in *Sim-nML*. The motivation behind this is to implement a complete processor simulation environment. The cache simulator provides a mechanism to study the caching policies of the processor being modeled. For an application, the designer could use the simulator to study the trade-offs between different caching policies. He could measure the preformance of the processor under various caching models by varying parameters like cache size, cache line size, associativity, replacement policy, effects due to unified or split cache model, effects of multi-level caching etc. The designer can simulate to get parameters such as cache hit rates, miss rates, conflict misses, invalid misses, compulsary misses etc. The advantage of providing such a mechanism is that the designer could simulate and study the caching behavior of the application programs to be run on the processor being designed much before the actual implementation. The benchmarks could then be used to select an ideal caching policy. This mechanism provides a generic architecture independent cache performance analysis.

Cache simulation can be done on-line or off-line. On-line cache simulation tries to keep track of the instruction and data addresses depending on the caching policy at run time. This involves running the application on a processor simulator and tracing the instruction and data memory references. In off-line cache simulation, the simulator could be used to generate a trace of the memory references. The trace could then be analyzed for cache references with suitable optimizations applied to speed up the process.

The Cache Simulator is built upon the *Retargetable Functional Simulator - Fsimg* developed by **Y. Subhash Chandra**[1] as part of his master's thesis. 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 other tools in studying other aspects of the design.

As the functional simulator simulates the execution of the given program, calls could be made using *canonical functions* (refer section 4.2). The instruction or data addresses are passed as parameters to cache simulation routines which simulates the caching behavior by keeping track of the addresses.

4.1 Cache Configuration

The *Cache Simulator* uses a configuration file wherein the designer can specify the caching policies. The simulator then reads the file to create the specified caching environment before actual simulation. Figure 4.1 gives a sample configuration file.

The following are the standard definitions used in the specification file.

- **levels**: specifies the number of levels of cache. The first is named as **L1**, second as **L2** and so on.
- **addrlen**: specifies the physical address length.
- **Level**: is used to denote which level is being described. **Level n** stands for the n^{th} level.
- **type**: denotes cache type being defined. It could be **INSTRCACHE** for instruction cache or **DATACACHE** for data cache or **UNIFIED** for a unified cache architecture.
- **associativity**: specifies the associativity of the cache being described. For a direct mapped cache it is given as **1**. For a n-way set associative cache it should be n. A keyword **FULL** can be used for describing a fully associative cache.
- **size n**: specifies the cache size. **n** can be suffixed with K(kilobytes) or M(megabytes). Without the quantifier, **n** is assumed to be in bytes.
- **line n**: specifies the cache line size. **n** can be suffixed with K(kilobytes) or M(megabytes). Without the quantifier, **n** is assumed to be in bytes.

- **replace**: denotes the replacement policy for *set associative* cache systems. The policy can be **FIFO**(first in first out), **RANDOM** or **LRU**(least recently used).
- **subblock**: denotes the subblock size within a cache line.
- **write**: specifies the write policy. Could be **WB WA**(write through with write allocate) or **WB NWA**(write back - no write allocate) or **WT WA**(write through with write allocate) or **WT NWA**(write through with no write allocate).
- **writebuffer**: specifies the size of the write buffer in bytes.
- **nonblocking**: specifies the number of outstanding misses that a cache can satisfy.

4.2 Implementation

The *Sim-nML*[15] language allows the use of *canonical functions* which are user defined functions. These are used to describe features which are not directly specified within *Sim-nML*. These are entities whose semantics would be realised by the tool that processes the *Sim-nML* description. They can be used to model the external environment like memory systems, caches, interrupts etc. The *cache simulator* uses 2 predefined canonical functions - *icache* and *dcache*.

For data addresses we use

```
"dcache"(address, type)
```

where *address* is the effective memory address(data) while *type* could be **READ** or **WRITE**. This is used to specify whether the access to the given *address* is a *read* or a *write*. The specification writer adds *dcache* calls in various *action* attribute definitions in the *Sim-nML* processor specification.

For instruction addresses, we use a similar function call as follows.

```
"icache"(address, type)
```

where *address* is the effective memory address(instruction) while *type* could be **READ** or **WRITE**. The *icache* canonical function call however cannot be buried in the *Sim-nML* specification. So *icache* is called by the *functional simulator* engine which is always aware of the instruction virtual addresses as specified in the input *ELF* binary.

Another possibility is to embed *icache* as a canonical function call within the top level *instruction* node's *action* attribute.

The cache simulator is run on-line along with the *functional simulator*. The functional simulator generator *Fsimg* converts the *canonical functions* as direct *C* calls to user defined routines. During the functional simulator generation process, *Fsimg* generates *icache* calls for each instruction. While running the simulator, the canonical functions, *dcache* and *icache* are called which simulate the cache system. During the first call to *icache* or *dcache* the *cache simulator* reads the configuration file and initializes the caching environment according to the specification. As each address is passed, the cache behavior is simulated and the performance metrics are sampled.

The simulator samples the following parameters - **HITS**, **MISSES**, **CONFLICT MISSES**, **INVALID MISSES**, **COMPULSARY MISSES**. During simulation it internally keeps track of the above metrics. At the end of the simulation, the statistics are dumped into log files. Statistics are maintained for each cache type at each cache level. They can be used later by the designer for performance analysis.

```
#Cache Configuration File

# Number of Levels
levels  3

# Address length
addrlen 32

# Description for L1 Cache
Level 1

# Description for InstrCache of L1
type INSTRCACHE
associativity 4
size 32K
line 16
replace FIFO
subblock 4      # subblock size
write WB WA     # Write Back, Write Allocate
writebuffer 32  # size of write buffer
nonblocking 2   # specifies number of outstanding misses

# Description for DataCache of L1
type DATACACHE
associativity 4
size 32K
line 16
subblock 4
replace FIFO
write WB WA
writebuffer 32
nonblocking 2
```

Figure 4.1: Sample Cache Simulation specification file

```
# Description for L2 Cache
Level 2

# Description for InstrCache of L2
type INSTRCACHE
associativity 8
size 512K
line 32
subblock 8
replace LRU
write WT NWA    # Write through, No Write Allocate

# Description for DataCache of L2
type DATACACHE
associativity 8
size 512K
line 32
subblock 8
replace LRU
write WT WA

# Description for L3 Cache
Level 3

# Description for Unified L3 Cache
type UNIFIED
size 2M
line 64
associativity FULL
replace RANDOM
subblock 16
write WT WA
```

Figure 4.2: Sample Cache Simulation specification file: continued

Chapter 5

Program Analysis and Profiling through Code Instrumentation

5.1 Introduction

Program analysis tools are extremely useful for understanding program behavior. Computer architects use such tools to evaluate how well the program performs on new architectures. Software writers need such tools to analyze their programs and identify critical pieces of code to optimize for efficiency. Compiler writers use such tools to find out how well their instruction scheduling or branch prediction algorithms are performing. As the third part of this thesis work, we implemented a mechanism to perform analysis and profiling of application programs through the technique of *code instrumentation*. This technique was inspired from **ATOM**[13] which is a framework for building wide range of customized program analysis tools. The *Retargetable Functional Simulator*[1] is used as a platform for performing program analysis.

We have tried to build a mechanism that provide architects and software developers to implement various profiling policies. These include basic block counting, instruction counting, branch behavior etc. In our approach, the profiling of code is accomplished by instrumenting application code at various points. For example, to count the basic blocks traversed at run time, a counter could be placed at the end of each basic block. Similarly to analyze branch behavior, routines could be added after conditional branch instructions.

We have tried to provide a common infrastructure using which users can build custom profiling tools. In our approach, the program is viewed as a collection of procedures each containing a collection of basic blocks each of which comprises of processor instructions. A user defined procedure for instrumenting the application program can be inserted before or after an instruction, a basic block, or a procedure. This model provides a generic processor profiling mechanism. Using such an approach, a custom profiling tool can be constructed. The *Sim-nML* language could be used to model a processor from which custom architecture independent profiling tools can be constructed.

The *Retargetable Functional Simulator - Fsimg*, generates a functional simulator of a given processor for a particular program (*compiled code simulator*). For each instruction in the input processor specification, a function is generated which is called during the simulation process. Each such function simulates the semantic action of that instruction. Code instrumentation can be done by inserting calls to the user defined procedures within each such function. The functional simulator engine maintains a table of function pointers which points to the functions each of which simulates a processor instruction. For each instruction in the input program, a pointer to the corresponding function for that instruction is maintained. Instrumentation can be done between such instruction calls. This allows instrumentation at instruction boundaries of the input binary. It can also be used for basic block profiling as well as the procedure level profiling.

The user defines the tool specific parts in a predefined *Instrument* function. A set of predefined routines - an application programming interface (*API*) is provided which allows the user to add his procedure calls before or after instructions. A set of *Basicblock analysis* routines are provided for profiling at the level of procedures, basic blocks or instructions within basic blocks. The profiling takes place in 2 phases. In the first phase, the user adds his instrumentation routines through the instrumentation-*API*. During generation of the functional simulator, the *API* calls are used to instrument the application program at appropriate places in the generated simulator. In the second phase, the user runs the simulator which executes the instrumented code while simulating the input program. This would then provide the profiling information.

5.2 Application Programing Interface - API

In order to perform code instrumentation, we provide the following instrumentation-*api* to the user.

The *api* currently provided are as follows:

1. **AddCallFuncbyName(iname, type, func, pos)**: *Fsimg* defines a function for each instruction in the input processor specification in *Sim-nML*. *AddCall-FuncbyName* adds the user procedure *func* within the function defintion corresponding to *iname*. Thus this function can be used to instrument a particular processor instruction in the application program whenever it is executed.

    - *iname*: could be the name of an instruction or a node in the *Sim-nML* hierarchy.
    - *type*: the type could be **INSTR_TYPE** or **NODE_TYPE** to specify whether *iname* is of instruction or node type.
    - *func*: is the name of the user routine which is to added.
    - *pos*: could be **BEFORE** or **AFTER** to specify whether *func* has to be executed before or after the execution of *iname* in the functional simulator.

2. **AddCallFunc(inst, func, pos)**: The functional simulator engine maintains for each instruction in the input program, a function pointer to the defining function. *AddCallFunc* is used to add *func* before or after the instruction address *inst* in the input program. Thus this function can be used to instrument the application program for a specifc address, i.e, whenever an instruction is fetched from the address *inst*.

    - *inst* - instruction address: Each instruction in the input program has an instruction address. This is the virtual address of the instruction in the input program.
    - *func*: is the name of the user routine which is to added.
    - *pos*: could be **BEFORE** or **AFTER** to specify whether *func* has to be instrumented before or after *inst*.

3. **AddTrailerFunc(func)**: The user can add any routines(*func*) to be executed after simulation. *Fsimg* adds these routines after the simulation engine. They

can be used by the user to collect the final statistics, dump profiling information etc.

4. **GetFirstProc**: Used to get the first procedure as listed in the *ELF* tables in the program.
5. **GetNextProc(p)**: Gets the next procedure after the current procedure *p*.
6. **GetFirstBlock(p)**: Gets the first basic block in procedure *p*.
7. **GetNextBlock(b)**: Gets the next basic block after the current block *b*.
8. **GetLastInst(b)**: Gets the last instruction of the basic block *b*.

Section 5.3 discusses the usage and implementation details.

5.3 Implementation

The instrumentation routines are added in 3 files - *instrument.c, bblockanal.c, userfuncs.c*. The first file contains a call to a predefined routine *Instrument* in which the user adds the *api* calls to add functions after particular instructions.
Suppose the user wants to count the occurrences of *add* instructions executed in the program, he uses

```
void Instrument()
{
    AddCallFuncbyName("add", INSTR_TYPE, "addcounter", AFTER);

    AddTrailerFunc("printaddcnt");
}
```

Here a user defined function *addcounter* is added within the *add* instruction defintion at its end. The file *userfuncs.c* contains the user defined routines. The function *addcounter* could be defined as follows:

```
long addcnt = 0;
```

```
void addcounter()
{
    addcnt++;
}
```

where *addcnt* is a global counter. *AddTrailerFunc* is used to add the user function *printaddcnt* at the end of simulation which could be defined as follows

```
void printaddcnt()
{
    printf("num of add instrucions executed : %d\n", addcnt);
}
```

The file *bblockanal.c* contains the instrumentation routines associated with basic block related analysis. It contains a call to a predefined routine *BasicblockAnal* in which the user adds the *api* calls to add functions relating to basic block profiling. Suppose the user wants to count the number of basic blocks that are traversed during program execution, he uses

```
void BasicblockAnal()
{
    Proc *p;
    Block *b;
    Inst inst;

    for (p = GetFirstProc(); p; p = GetNextProc(p)) {
        for (b = GetFirstBlock(p); b; b = GetNextBlock(b)) {
            inst  = GetLastInst(b);
            AddCallFunc(inst, "countbb", AFTER);
        }
    }
    AddTrailerFunc("printbb");
}
```

The user defined function *countbb* is added *after* the last instruction in each basic block. The user might want to call different functions at the same address boundary. Multiple user defined instructions can be engineered at address boundaries by calling *AddCallFunc* with different function names at the same instruction address.

For basic block oriented profiling, the *Fsmig* analyzes the input program to obtain basic blocks in the input program. A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. A basic block is obtained using an algorithm shown in figure 5.1.

1. Determine the set of *leaders*, the first statements of each basic block.
2. The rules used are.
   - the first statement is a leader.
   - any statement that is the target of a conditional or unconditional goto is a leader.
   - any statement that immediately follows a goto or conditional goto statement is a leader.
3. For each basic block, its basic block consists of the leader and all statements up to but not including the next leader or end of the program.

Figure 5.1: Algorithm to construct basic block

This is done by providing the *Fsimg* with the conditional and unconditional control flow instructions(branch/call/jmp) of the processor instruction set. Since *Sim-nML* is a hierarchical description, if the hierarchy allows, we can provide the top level branch node instead. The actual branch instructions can then be enumerated from this. Once a list of branch instructions are enumerated, we split the input instruction stream at procedure boundaries. For a given procedure, the basic block boundaries are marked just after every branch instruction.

To calculate the branch target addresses, a configuration file has to be provided which specifies the relevant branch instruction with the branch target calculation mechanism. A sample configuration is shown in figure 5.2

In the configuration file, **$** refers to the current instruction address, *%n* refers to

```
# configuration file for branch/jump target specification

#instruction name        target address

b                       $ + (%0 << 2)

ba                      %0 << 2
```

Figure 5.2: Branch target specification file

the n^{th} parameter of the instruction specification in *Sim-nML* **'and'** rule. Parameters are counted from left to right starting from 0.

During generation, *Fsimg* calls *Instrument* to add instrumentation routines. It then performs basic block analysis where procedures and basic blocks within each procedure are enumerated. It then calls *BasicblockAnal* to add the relevant user defined routines. The *Fsimg*[1] implements each instruction in the *Sim-nML* processor description as a function definition. The functional simulator engine contains a table of pointers to functions corresponding to the instructions in the input program. *AddCallFuncbyName* essentially modifies the function defintion for the relevant instruction by adding a call to the user defined routine. *AddCallFunc* modifies the table of function pointers by adding a call to an alternate routine which embeds a call to the relevant user defined routine along with the call to the actual instruction definition function. During simulation, the functional simulator simulates the input program by calling the routines corresponding to each instruction. The profling is done by calling the user engineered routines.

Chapter 6

Motorola 68HC11 Specification in Sim-nML

In this chapter, we briefly survey the *Motorola 68HC11*[17] processor architecture and its *Sim-nML* specification.

The *Motorola 68HC11*, is a family of micro-controller units with a simple 8-bit processor core. The programmer's model consists of the following

- **Accumulators(A, B and D)**: **A** and **B** are two general purpose 8-bit accumulators used to hold operands and results of arithmetic calculations and data manipulations. Some instructions treat the combinations of these two as a 16-bit double accumulator (accumulator **D**). The higher order byte of **D** is equivalent to accumulator **A** while the lower order byte corresponds to the accumulator **B**.

- **Index Registers (X and Y)**: The 16-bit index registers **X** and **Y** are used for index addressing mode. In the indexed addressing mode, the effective address is obtained by adding the contents of a 16-bit index register to an 8-bit immediate offset in the instruction.

- **Stack Pointer (SP)**: The M68HC11 CPU supports a program stack which may be located anywhere in the 64-Kbyte address space and may be of any size up to the amount of memory available in the system.

- **Program Counter (PC)**: The program counter is a 16-bit register that holds the address of the next instruction to be executed.

- **Condition Code Registers (CCR)**: This register contains five status indicators, two interrupt masking bits, and a STOP disable bit. The five flags reflect the results of arithmetic and other operations. The five flags are half carry (H), negative (N), zero (Z), overflow (V) and carry/borrow (C).
- **Addressing Modes**:
  1. **Immediate**. The actual argument is contained in the byte(s) immediately following the instructions.
  2. **Extended**. The effective address of the operand appears explicitly in the two bytes following the opcode.
  3. **Direct**. The least significant byte of the effective address of the instruction appears in the byte following the opcode. The higher order byte of the effective address is 0.
  4. **Indexed (INDX, INDY)**. The effective address is the contents of either of the index registers **X**, **Y** plus a fixed 8-bit unsigned offset contained in the instruction.
  5. **Inherent**. Contains implicit operands. For example, the instruction **ABA** adds the contents of accumulator **A** with accumulator **B** and stores the results in **A**.
  6. **Relative**. For branch instructions, the target address is the address of the next instruction plus a 8-bit signed offset specified in the instruction.

6.1 Overview of the Specifications

We have specified the *Motorola 68HC11* specification in *Sim-nML*. A simple resource usage model has been assumed. A single instance of a resource **exec_unit** is declared. Any instruction in execution acquires this resource for the time period depending on the number of clock cycles required for execution of that instruction.

The instruction set of the *68HC11* CPU is organized in a hierarchy in the *Sim-nML* specification. The description hierarchy is as follows. Top level node is the *instruction.* Instruction can be arithmetic, stack control, program control, conditional instructions, load-store instructions. Arithmetic instructions can furthur be classified into add-subtract, multiply-divide, shift-rotate, data test bit instructions. Program control instructions consist of branch, jump, subroutine calls etc. These instructions operate on both 8-bit and 16-bit data. Instructions involving external interfaces like interrupts, serial/parallel data transfers have not been specified.

Chapter 7

Results and Conclusion

In this chapter we discuss a few sample cache profiling and code instrumentation mechanisms along with their performance impacts on the speed of the functional simulator.

7.1 Results

The caching and profiling mechanisms were tested on **PowerPC603** *Sim-nML* input specification. The test cases were run on an

- Intel P-II 233MHz, a little-endian processor with 32MB RAM running GNU-Linux Kernel 2.2.13.

Following are the test programs written in **C**. The *PowerPC603 ELF* binaries were created using the GNU-C cross-compiler.

- **mmul.c** : Matrix multiplication program. This program initializes two integer matrices of 100x100 size and multiplies these two.
- **bsort.c** : Bubble 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.

The total number of dynamically executed instructions during the simulation of each of these programs are given in the table 7.1 and the performance of the functional simulator without the cache simulation and profiling is given in table 7.2.

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.

Program	Total Time in Seconds	Instructions per second
mmul.c	62	1,476,322
bsort.c	106	573,198
qs.c	109	741,044
fmmul.c	64	1,439,549
nqueen.c	225	910,741

Table 7.2: Performance Results of the functional simulator

7.1.1 Caching Example

We have used the sample configuration file as specified in the figure 7.1 and the corresponding output metrics measured are given in table 7.3. In table 7.3,

- *Cache type* indicates whether the cache is a data or instruction cache.

- *Level* denotes the cache level in the cache hierarchy.
- *Hits* denote the % of the total memory access that resulted in a cache hit.
- *Misses* denote the % of the total memory access that resulted in a cache miss.
- *Conflict Miss* denote the % of the total memory access that resulted in a conflict miss (i.e, the cache entry was marked valid) in the cache.
- *Invalid Miss* denote the % of the total memory access that resulted in a invalid miss (i.e, the cache entry was marked invalid) in the cache.
- *Compulsary Miss* denote the % of the total memory access that resulted in a compulsary or cold miss (i.e, the address was being accessed for the first time) in the cache.

Program	Cache type	Level	Hits (%)	Misses (%)	Conflict misses(%)	Invalid misses(%)	Compulsary misses(%)
mmul.c	Data	1	98.5	1.3	1.3	0	.02
	Instr	1	99.95	0.04	0.04	0	0
bsort.c	Data	1	99.9	0.001	0.001	0	0
	Instr	1	99.9	0.001	0.001	0	0
qs.c	Data	1	99.75	0.24	.24	0	0.01
	Instr	1	99.89	0.1	0.1	0	0
fmmul.c	Data	1	98.5	1.3	1.3	0	0.02
	Instr	1	99.9	0.04	0.04	0	0
nqueen.c	Data	1	99.9	0	0	0	0
	Instr	1	99.9	0	0	0	0

Table 7.3: Results of profiling output for test programs.

The performance of the functional simulator with on-line cache simulation given in table 7.4.

7.1.2 Profiling Example

We have implemented a simple profiling tool which counts the number of basic blocks that are traversed at run time. At the same time, the number of *PowerPC* **addi**

```
#PowerPC603 Cache Configuration

# Number of Levels
levels  1

# Address length
addrlen 32

# Description for L1 Cache
Level 1

# Description for InstrCache of L1
type INSTRCACHE
associativity 2
size 8K
line 32
replace LRU
write WB WA     # Write Back, Write Allocate

# Description for DataCache of L1
type DATACACHE
associativity 2
size 8K
line 32
replace LRU
write WB WA
```

Figure 7.1: PowerPC603 Cache configuration file

Program	Total Time in Seconds	Instructions per second	slowdown factor
mmul.c	549	166,72	8.8
bsort.c	546	111,280	5.1
qs.c	809	99,844	7.4
fmmul.c	522	176,498	8.1
nqueen.c	1138	180,06	5.0

Table 7.4: Performance results of cache profiling for test programs.

instructions that are executed is also found. The code instrumentation technique that is used is specified in section 4.2.

The profiling output is given in table 7.5.

Program	Total No. of basic block traversed	Total No: of addi instructions executed
mmul.c	2081207	1030305
bsort.c	4506008	2253005
qs.c	7315513	242144
fmmul.c	2081207	1110305
nqueen.c	40030204	60766515

Table 7.5: Profiling output for test programs.

The performance of the *functional simulator* with this profiling is given in table 7.6.

Program	Instructions per second	Slowdown factor
mmul.c	1,452,888	1.01
bsort.c	573,198	1
qs.c	734,307	1.01
fmmul.c	1,439,549	1
nqueen.c	898,758	1.01

Table 7.6: Performance results of profiling of test programs.

7.1.3 IR-Generator

The *IR* fulfills all the goals that were setup behind the design and extension of the IR. The shortcomings of the earlier *IR* were removed.

The *IR-generator* was tested for processor models of *PowerPC603*, *Motorola 68HC11* & *Intel 8085*. It was run on *Linux/Intel* and *Solaris/Ultrasparc* platforms.

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 generic tools - assembler, simulator, compiler, etc. can be automatically generated using *Sim-nML* processor models.

As part of this thesis work, we have extended the *IR* for processor description using *Sim-nML* language. The *IR* simplifies the development of tools like compiler back-end generators, assemblers, disassemblers, simulators etc. An *IR generator* has been developed which takes the *Sim-nML* specification as input and produces the *IR* of the processor specification. We have also implemented a mechanism for program analysis. This includes a mechanism for cache simulation and an infrastructure for program profiling through code instrumentation. These tools help in generating a processor independent platform for program analysis. It was implemented over the *Retargetable Functional Simulator - Fsimg.* The tools were tested for the *PowerPC603* specification.

7.3 Future Work

We visualize the following that can be used to build a complete processor simulation environment.

A complete simulation of the external environment of the processor can be done. This would involve developing separate simulation modules for memory, cache systems, bus etc. which would interact with the processor functional or timing simulator. The functional or timing simulator would then only simulate the processor. This would make the system more modular, scalable and flexible.

Bibliography

[1] Chandra, Y. S. Retargetable Functional Simulator. Master's thesis, Department of Computer Science and Engg., IIT Kanpur, June 1999. http://www.cse.iitk.ac.in/research/mtech1997/9711121.html.

[2] Edler, J., and Hill, M. D. Dinero IV Trace-Driven Uniprocessor Cache Simulator.

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

[4] 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.

[5] Jain, N. C. Disassembler 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.

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

[7] Larus, J. R., and Ball, T. Rewriting Executable Files to Measure Program Behavior. *Software Practice & Experience 24*, 2 (Feb 1994), 197–218.

[8] Larus, J. R., and Schnarr, E. EEL: Machine-Independent Executable Editing. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 1995).

[9] 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.

[10] 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.

[11] 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.

[12] Smith, M. D. Tracing with Pixie. *Memo from Center for Integrated Systems, Stanford Univ.* (April 1991).

[13] Srivastava, A., and Wall, D. ATOM: A system for building customized analysis tools. *Proceedings of the SIGPLAN '94 Conference of Programming Language Design and Implementation (PLDI)* (June 1994), 196–205.

[14] Trung A., D., and John Paul, S. VMW: A Visualization-Based Microarchitecture Workbench. *IEEE Computer* (Dec 1995), 57–64.

[15] 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.

[16] *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.

[17] *M68hc11 Reference Manual.* Motorola Inc., 1994. http://mot-sps.com/mcu/documentation/pdf/hc11rmr3.pdf.

Appendix A

Grammar of Sim-nML Language

Following is the Context Free Grammar for *Sim-nML* language.

```
MachineSpec :
        |  MachineSpec  LetDef
        |  MachineSpec  TypeSpec
        |  MachineSpec  MemorySpec
        |  MachineSpec  RegisterSpec
        |  MachineSpec  VarSpec
        |  MachineSpec  ModeSpec
        |  MachineSpec  OpSpec
        |  MachineSpec  ResourceSpec
        |  MachineSpec  ExceptionSpec
        |  MachineSpec  error
        ;
LetDef :  LET ID
          '=' LetExpr
        ;
ResourceSpec:  RESOURCE
               ResourceList
        ;
ResourceList:
            ID
            | ID '[' CARD_CONST ']'
            | ResourceList ',' ID
            | ResourceList ',' ID '[' CARD_CONST ']'
        ;
ExceptionSpec:  EXCEPTION
                IdentifierList
        ;
IdentifierList:
                ID
                | IdentifierList  ','  ID
        ;
TypeSpec: TYPE ID
          '=' TypeExpr
        ;
TypeExpr: BOOL
          | INT  '(' LetExpr')'
          | CARD '(' LetExpr')'
          | FIX  '(' LetExpr ',' LetExpr')'
          | FLOAT  '(' LetExpr ',' LetExpr')'
          | '[' LetExpr DOUBLE_DOT LetExpr ']'
          | ENUM '(' IdentifierList ')'
    ;
LetExpr:  Expr
        ;
MemorySpec: MEM ID
            '[' MemPart ']' OptionalMemAttrDefList
        ;
RegisterSpec: REG ID
              '[' RegPart ']' OptionalMemAttrDefList
        ;
VarSpec: VAR ID
         '[' RegPart ']'
    ;
MemPart: LetExpr ',' Type
         | LetExpr
    ;
RegPart: LetExpr ',' Type
         | Type
        ;
Type : TypeExpr
       | ID
    ;
OptionalMemAttrDefList:
                       | MemAttrDefList
        ;
MemAttrDefList:
               MemAttrDef
               | MemAttrDefList MemAttrDef
        ;
MemAttrDef:
           VOLATILE '=' LetExpr
           | PORTS '=' CARD_CONST ',' CARD_CONST
           | ALIAS '=' MemLocation
           | INITIALA '=' LetExpr
           | USES '=' UsesDef
        ;
MemLocation :
            ID Opt_Bit_Optr
            | ID '[' Expr ']' Opt_Bit_Optr
        ;
ModeSpec:  MODE ID
           ModeSpecPart
        ;
ModeSpecPart: AndRule OptionalModeExpr  AttrDefList
              | OrRule
        ;
OptionalModeExpr :
                  | '='
                  Expr
        ;
OpSpec: OP ID
        OpRulePart
        ;
OpRulePart: AndRule AttrDefList
            | OrRule
        ;
OrRule: '='
        Identifier_Or_List
        ;
Identifier_Or_List:
                   ID
                   | Identifier_Or_List  '|'  ID
        ;
AndRule: '(' ParamList ')'
        ;
ParamList:
          | ParamListPart
          | ParamList ',' ParamListPart
        ;
ParamListPart:
              ID
              ':' ParaType
        ;
ParaType : TypeExpr
           | ID
        ;
AttrDefList:
            | AttrDefList   AttrDef
        ;
AttrDef :
         ID '='  AttrDefPart
         | SYNTAX '=' AttrExpr
         | IMAGE '=' AttrExpr
```

```
          | ACTION '=' '{' Sequence '}'
          | USES '=' UsesDef
         ;
AttrDefPart:
             Expr
             | '{' Sequence '}'
         ;
AttrExpr :
           ID '.' SYNTAX
           | ID '.' IMAGE
           | STRING_CONST
           | FORMAT  '(' STRING_CONST  ','  FormatIdlist ')'
         ;
FormatIdlist:
              FormatId
              | FormatIdlist  ',' FormatId
         ;
FormatId:
          ID
          | ID '.' IMAGE OptBitSelect
          | ID '.' SYNTAX
          | DOLLAR '+' ID
         ;
OptBitSelect:
              | BIT_LEFT CARD_CONST DOUBLE_DOT CARD_CONST BIT_RIGHT
         ;
Sequence:
          | StatementList ';'
         ;
StatementList:
               Statement
               | StatementList ';' Statement
         ;
Statement:
```

```
            | ACTION
            | ID
            | ID '.' ACTION
            | ID '.' ID
            | Location  '='  Expr
            | ConditionalStatement
            | STRING_CONST '(' ArgList ')'
            | ERROR '(' STRING_CONST ')'
        ;
ArgList :
         | Expr
         | ArgList ',' Expr
        ;
Opt_Bit_Optr :
              | BIT_LEFT Bit_Expr DOUBLE_DOT Bit_Expr BIT_RIGHT
        ;
Location :
          ID Opt_Bit_Optr
          | ID '[' Expr ']' Opt_Bit_Optr
          | Location DOUBLE_COLON Location
        ;
ConditionalStatement:
                     IF  Expr  THEN  Sequence OptionalElse   ENDIF
                     | SWITCH '(' Expr ')' '{' CaseList '}'
        ;
OptionalElse:
              | ELSE  Sequence
        ;
CaseList:
         CaseStat
         | CaseList   CaseStat
        ;
CaseStat:
        CaseOption ':' Sequence
        ;
CaseOption:
           CASE  Expr
           | DEFAULT
        ;
Expr :
      COERCE '(' Type ','
      Expr')'
      | FORMAT '(' STRING_CONST ',' ArgList ')'
      | STRING_CONST '(' ArgList ')'
      | ID '.' SYNTAX
      | ID '.' IMAGE
      | ID '.' ID
      | Expr DOUBLE_COLON Expr
      | ID '[' Expr ']' Opt_Bit_Optr
      | ID Opt_Bit_Optr
      | Expr '+' Expr
      | Expr '-' Expr
      | Expr '*' Expr
      | Expr '/' Expr
      | Expr '%' Expr
      | Expr  DOUBLE_STAR Expr
      | Expr LEFT_SHIFT Expr
      | Expr RIGHT_SHIFT Expr
      | Expr ROTATE_LEFT Expr
      | Expr ROTATE_RIGHT Expr
      | Expr '<' Expr
      | Expr '>' Expr
      | Expr LEQ Expr
      | Expr GEQ Expr
      | Expr EQ Expr
      | Expr NEQ Expr
      | Expr '&' Expr
      | Expr '^' Expr
```

```
        | Expr '|' Expr
        | '!' Expr
        | '~' Expr
        | '+' Expr %prec '~'
        | '-' Expr %prec '~'
        | Expr AND Expr
        | Expr OR Expr
        | '(' Expr ')'
        | FIXED_CONST
        | CARD_CONST
        | STRING_CONST
        | DOLLAR
        | BINARY_CONST
        | HEX_CONST
        | IF  Expr  THEN  Expr OptionalElseExpr   ENDIF
        | SWITCH '(' Expr ')' '{' CaseExprList '}'
         ;
Bit_Expr :
          ID
          | Bit_Expr '+' Bit_Expr
          | Bit_Expr '-' Bit_Expr
          | Bit_Expr '*' Bit_Expr
          | Bit_Expr '/' Bit_Expr
          | Bit_Expr '\%' Bit_Expr
          | Bit_Expr  DOUBLE_STAR Bit_Expr
          | '(' Bit_Expr ')'
          | FIXED_CONST
          | CARD_CONST
          | STRING_CONST
          | BINARY_CONST
          | HEX_CONST
        ;
CaseExprList:
             CaseExprStat
```

```
            | CaseExprList   CaseExprStat
        ;
CaseExprStat:
            CaseOption ':' Expr
        ;
OptionalElseExpr:
                | ELSE  Expr
        ;
UsesDef:
        UsesOrSequence
        | UsesDef ',' UsesOrSequence
        ;
UsesOrSequence:
               UsesIfAtom
               | UsesOrSequence '|' UsesIfAtom
        ;
UsesIfAtom:
           UsesIndirectAtom
           | IF Expr THEN UsesIfAtom OptionalElseAtom ENDIF
        ;
OptionalElseAtom :
                 | ELSE UsesIfAtom
        ;
UsesIndirectAtom:
                 UsesCondAtom
                 | ID '.' USES
                 | '(' UsesDef ')'
                 | UsesLocationList AND  ID  '.'  USES
                 | UsesLocationList AND  '('  UsesDef  ')'
        ;
UsesCondAtom:
             UsesAndAtom
             | '{' Expr '}' UsesAndAtom
        ;
```

```
UsesAndAtom :
             UsesLocationList  UsesActionList
        ;
UsesActionList :
                | ActionTimeList OptionalAction
                | TimeActionList  OptionalTime
        ;
ActionTimeList :
                '#' '{' Expr '}'
                | ActionTimeList  ':' UsesActionAttr '#' '{' Expr '}'
        ;
TimeActionList :
                ':' UsesActionAttr
                | TimeActionList  '#' '{' Expr '}' ':' UsesActionAttr
        ;
OptionalAction :
                | ':' UsesActionAttr
        ;
OptionalTime :
              | '#' '{' Expr '}'
        ;
UsesActionAttr:
               ID
               | ACTION
        ;
UsesLocationList :
                  UsesLocation
                  | UsesLocationList  '&'  UsesLocation
        ;
UsesLocation :
              ID Opt_Bit_Optr
              | ID '[' Expr ']' Opt_SecDim Opt_Bit_Optr
        ;
```

```
Opt_SecDim :
           | '[' ']'
```

Appendix B

File Format of Intermediate Representation

In this appendix, we will discuss the layout of the file for the intermediate representation. The file consists of two parts. The first part is the *IR* header and the second part is essentially a collection 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. All other tables can reside anywhere in the second part 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 ATTRIBUTE DEFNITION TABLE"

Each table consists of an array of records. Each record in a table constitutes of various fields. The fields might be stored either in *little-endian* or *big-endian* encoding using the native data storage order of the host processor.

- **Convention :** Each table is described by defining its record format. We have used a C-like struct definition to describe a record. Refer to *tables.h* for complete definition for the structures and predefined constants. In describing the record, following data types are being used.

`uint8`	=	*unsigned char - 8 bits*
`uint16`	=	*unsigned integer - 16 bits*
`uint32`	=	*unsigned integer - 32 bits*
`int8`	=	*signed char - 8 bits*
`int16`	=	*signed integer - 16 bits*
`int32`	=	*signed integer - 32 bits*

B.1 IR Header

The IR header contains 2 fields as shown below. The first is a magic number. For the current version it is set to "IRV2". The second field is used to denote the endianness of the host processor on which the *IR* was created. The possible values for *endian* are **LITTLE_END** and **BIG_END**, two constants defined in *tables.h.*

```
typedef struct {
        uint8 magic[4];
        uint8 endian;
} IR_Header;
```

B.2 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 {
        uint8 table_name[32];
        uint32 table_size;
        uint32 table_offset;
        uint32 total_records;
        uint32 record_size;
} MetaTable_t;
```

- 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 :**META_TABLE, CONST_TABLE, ATTRIBUTE_TABLE, RESOURCE_TABLE, IDENTIFIER_TABLE, MEMORY_TABLE, AND_RULE_TABLE, OR_RULE_TABLE, SYNTAX_TABLE, IMAGE_TABLE STRING_TABLE, INTEGER_TABLE, PREFIX_TABLE**. The first entry in the table is for *META TABLE* itself.
- 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 from the beginning of the file.
- total_record : This field holds the number of fixed size records stored in a table. Tables with variable size records like *string table*, *integer table* have this field set to 0.
- record_size : This field holds the size of a fixed size record (in bytes) in a table. Tables with variable size records like *string table* and *integer table* have this field set to 0.

B.3 Constant Table

Each record of the *constant table* holds the informations about the constant expressions in the following format.

```
typedef struct {
        uint32 id_name;
        int8 val_type;
        int32 value;
} ConstTable_t;
```

- id_name: This field holds the index into the *string table*. The *string table* holds null terminated strings. Thus this field represents a reference to the constant name.
- val_type: This field indicates the type of the value associated with the constant. Currently it can be one of the two constants **CONST_TBL_INT_TYPE, CONST_TBL_STRING_TYPE** as defined in *tables.h*
- value: If the *val_type* field represents **CONST_TBL_INT_TYPE**, then this field holds the corresponding *int32* value. If the *val_type* is **CONST_TBL_STRING_TYPE**, then this field holds the index into the *string table*.

B.4 Resource Table

Entries of this table hold the information about *resource*. Each entry indicates the resource name and the number of instances of each resource. Each record has the following format.

```
typedef struct {
        uint32 res_name;
        uint32 res_num;
} ResourceTable_t;
```

- res_name : This field holds the index into the *string table* where the resource name is stored.
- res_num : This field stores the number of instances of this resource (≥ 1).

B.5 Identifier Table

This table holds the information about all the identifiers used in the processor specification file (other than those specified in the *constant table* and the *resource table*). Each record has the following format.

```
typedef struct {
        uint32 id_ptr;
        uint32 id_name;
        uint32 id_type;
} Identifier_t;
```

- 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_type : This field indicates the type of the identifier and may have one of the following values as defined in *tables.h*.

UNDEFINED	Undefined Identifier
MEM_TYPE	Memory Variable
MODE_OR_TYPE	Mode Or Rule
MODE_AND_TYPE	Mode And Rule
OP_OR_TYPE	Op Or Rule type.
OP_AND_TYPE	Op And Rule type.
EXCEPTION_TYPE	Exception

- id_ptr : This field holds the pointer to other tables depending on *id_type* value assigned to the identifier.

MEM_TYPE:	index into the *memory table*
MODE_OR_TYPE:	index into the *or rule table*
MODE_AND_TYPE:	index into the *and rule table*
OP_OR_TYPE:	index into the *or rule table*
OP_AND_TYPE:	index into the *and rule table*

B.6 Attribute Table

Each entry of this table holds the name of an *attribute*. Each record has the following format.

```
typedef struct {
        uint32 attr_name;
} Attribute_t;
```

- attr_name : This field holds an index into the *string table* where the attribute name is stored

B.7 Memory Table

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

```
typedef struct {
        uint32 id_index;
        uint32 size;
        uint32 total_attr;
        uint8 type;
        uint8 data_type;
        uint32 value1;
        uint32 value2;
        uint32 attr_list_index;
} MemTable_t;
```

- id_index : This field stores the index into the *identifier table.*
- size : A memory declaration defines a memory base, i.e., a set of memory locations accessible with a name and an index. This field specifies the number of such locations.
- total_attr : A memory declaration may also define values for some predefined attributes. This field specifies how many attributes are defined for the memory variable.
- type : This field holds a constant which can be **REG** if the identifier is declared using *reg* specification, **MEM** if the identifier is declared using *mem* specification and **VAR** if the identifier is declared using *var* declaration.
- data_type, 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_type*, specifies what type of values can be stored in a memory location. Second and third field stores the value according to the *data_type* field. *data_type* can be **BOOL_TYPE, CARD_TYPE, INT_TYPE, FIX_TYPE, FLOAT_TYPE, RANGE_TYPE, ENUM_TYPE**.

  Table B.1 shows the possible of other two fields.
- attr_list_index : If the *total_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 tuple indicates <*index, offset, len*> where *index*, is the index into the *attribute table* corresponding to that attribute. The second field of the tuple, *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 the triple, *len*, is the number of tuples for its attribute definition. Each tuple in the *prefix attribute table* is of type **PrefixTuple_t** (refer section B.14).

  For *mode* specification (refer *Sim-nML specification[15]*), one new attribute, *_val_*, is defined to store the optional expression associated with the *mode* specification. The expression is declared using =. For example, in figure 2.2, the *and* *rule* `mode REG(index : card(5)) = R[index]`, has an associated attribute *_val_* which defines the expression `R[index]`.

Data Type	**data_type**	**value1**	**value2**
bool	BOOL_TYPE	0	0
card(n)	CARD_TYPE	n	0
int(n)	INT_TYPE	n	0
fix(n, m)	FIX_TYPE	n	m
float(n, m)	FLOAT_TYPE	n	m
range[$n..m$]	RANGE_TYPE	n	m
enum(id_1...id_m)	ENUM_TYPE	0	$m - 1$

Table B.1: Encoding of data types

B.8 And-Rule Table

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

```
typedef struct {
        uint32 id_index;
        uint32 total_para;
        uint32 total_attr;
        uint32 attr_list_index;
        uint32 para_list_index;
} AndTable_t;
```

- id_index : This field holds the index into the *identifier table* corresponding to this *and-rule.*
- total_para : This field holds the number of parameters associated with 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 <*index, offset and len*> similar to the one described in the *memory table.* There are two exceptions here. If *index* refers to a *syntax* or *image* attribute, then *offset* field contains the index into the *syntax table* or the *image table*, as the case might be, and *len* field is 0.

- 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. Each integer triple indicates <*data_type, value1, value2*> i.e. the data type of parameter. *data_type* takes the same value as of *memory table* data types. In addition it could take the value **AND_RULE_TYPE, OR_RULE_TYPE**. Table B.2 shows possible values for fields of the triples.

Data Type	**data_type**	**value1**	**value2**
bool	BOOL_TYPE	0	0
card(n)	CARD_TYPE	n	0
int(n)	INT_TYPE	n	0
fix(n, m)	FIX_TYPE	n	m
float(n, m)	FLOAT_TYPE	n	m
range[$n..m$]	RANGE_TYPE	n	m
enum(id_1...id_m)	ENUM_TYPE	0	$m - 1$
and-rule	AND_RULE_TYPE	*and table index*	0
or-rule	OR_RULE_TYPE	*or table index*	0

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

B.9 Or-Rule Table

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

```
typedef struct {
        uint32 id_index;
        uint32 total_child;
        uint32 child_list_index;
} OrTable_t;
```

- id_index : This field holds the index into the *identifier table* corresponding to this *or-rule.*
- total_child : This field holds the number of children for this *or rule.*
- child_list_index : This field holds the index into the *integer table* where a list of integer values are stored. For each child 2 integers are stored. The first integer indicates the child type which could be **AND_RULE_TYPE** or **OR_RULE_TYPE**. The second integer denotes the index into the *and rule table* or *or rule table* depending on the child type.

B.10 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 {
        uint8 type;
        uint32 str_len;
        uint32 str_off;
} SynImg_t;
```

- type : This field holds the type of the syntax record. It could be **SYNIMG-DOT_TYPE** or **SYNIMGSTR_TYPE** For example, if the syntax attribute is defined as `syntax = x.syntax`

  where x is a parameter, then the *type* is **SYNIMGDOT_TYPE**, else if it is defined as a string or using the *format* keyword then the *type* is **SYN-IMGSTR_TYPE**.

- str_len : If *type* is **SYNIMGDOT_TYPE** then this field holds the parameter number (of x in the above example). If *type* is **SYNIMGSTR_TYPE** then this field holds the format string length.

- str_off : If *type* is **SYNIMGDOT_TYPE** then this field holds the index into the attribute table (of *syntax* in the above example) while if *type* is **SYNIMGSTR_TYPE** then this holds the offset into the *string table* where the format string is stored.

B.11 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 {
        uint8 type;
        uint32 str_len;
        uint32 str_off;
} SynImg_t;
```

- type : This field holds the type of the image record. It could be **SYNIMGDOT_TYPE** or **SYNIMGSTR_TYPE**. For example, if the image attribute is defined as `image = x.image`

  where x is a parameter, then the *type* is **SYNIMGDOT_TYPE**, else if it is defined as a string or using the *format* keyword then the *type* is **SYNIMGSTR_TYPE**.

- str_len : If *type* is **SYNIMGDOT_TYPE** then this field holds the parameter number (of x in the above example). If *type* is **SYNIMGSTR_TYPE** then this field holds the format string length.

- str_off : If *type* is **SYNIMGDOT_TYPE** then this field holds the index into the attribute table (of *image* in the above example). If *type* is **SYNIMGSTR_TYPE** then this field holds the offset into the *string table* where the format string is stored.

B.12 String Table

This table holds null terminated character sequences, commonly called *strings*. These strings are referred to by an index into the *string table*. 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 B.3 and the *strings* associated with various indices in table B.4.

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

Table B.3: Example of the String Table

Index	*string*
0	identifier
11	PC
14	instruction

Table B.4: Interpretation of the String Table

B.13 Integer Table

This table holds list of *signed or unsigned integer* values (`int32 or uint32` type). These integers represent different meanings in different contexts. The integers are referred in other tables by an *index* into the *integer table*. The *index* refers to the starting offset(index) into the integer table where the list of integers is stored.

B.14 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. It contains an array of records where each record of the prefix expression is stored as follows.

```
typedef struct {
        uint16 type;
        int32 value;
} PrefixTuple_t;
```

- type : This field holds an integer value to indicate the type of tuple, i.e., an *operator tuple* or *operand tuple.* For a tuple of *operand type*, this field also encodes the type of the operand.
- value : This field holds an integer value whose interpretation depends on the value of the *type* 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 tuples in the prefix notation of the definition. Table B.5 shows the possible values of *type* field and the corresponding interpretation for the *value* field. If the *type* field is set to a value 0, then the tuple is an *operator tuple.* In all other cases, the tuple is an *operand tuple.* If the tuple is an *operator* tuple, then the *value* field holds an integer which indicates operator's name and its arity. Table B.6 shows all possible values for this field and the corresponding arity.

There are as many operands available as needed for an operator. Since the arity for an operator is known a-priori, the number of its arguments is implicit. For example, an expression ' $PC = PC + 2$ ' is represented as ' $= PC + PC\ 2$ ' in prefix notation. The expression has 5 items. The first item is an operator '=' with arity 2. The second item is a memory variable with the value field being the index into the *memory table.* The third item is again an operator '+'. The fourth item is a memory variable while the last item is a *fixed-constant* with value 2.

The detailed description of each operator is given in 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 $opd1 < opd2..opd3 >$. It is considered as a ternary operator with three parameters as *opd1, opd2* and *opd3* for prefix notation.
- The second case is for "if then else". It is considered as a ternary operator *IF*. If there is no operand in *else* part, then NULL operator (0-ary) (see table B.6) is used in its place.

Type of the tuple	**type** field	**value** field
Operator	0	operator number (see table B.6)
Fixed constant	1	*int32* value of operand
Card constant	2	*uint32* 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	index of the identifier as assigned in the *identifier table*
Attribute type	7	index of the attribute name in the *attribute table*
Parameter type	8	parameter number (left most is assigned number 0).
Resource type	9	index of the resource name as assigned in the *resource table*
Exception type	10	index of the identifier as assigned in the *identifier table*

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

- The third case is when there is no attribute expression for an attribute. The NULL operator is used 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,
```

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	Greator 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	default	0-ary
35	NULL	nothing	0-ary
36	Hash	#	Binary
37	Comma	,	Binary
38	Condition	{}	Unary
39	Colon	:	Binary

Table B.6: Operators Used in Prefix Attribute Definition

```
Expr_1,          Sequence_1,
Expr_2,          Sequence_2,
....
default, 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 (see table B.6).

- 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)
    ("function name" string, n, Arg1, Arg2,........Argn)
  ```

  The first item is a function operator. Second tuple is a string constant type (*type* = String constant, *value* = byte offset into the string table where function name is stored). Next item n is a simple operand tuple with *type* as Card constant and *value* as n. Following which, 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 B.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 is two more than the actual number of parameters for each occurence of *coerce* function.