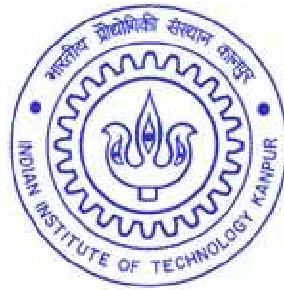
Functional Simulation Using Sim-nML

by

Surendra Kumar Vishnoi



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

May 2006

Functional Simulation Using Sim-nML

A Thesis Submitted

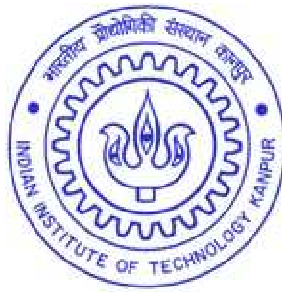
in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Surendra Kumar Vishnoi



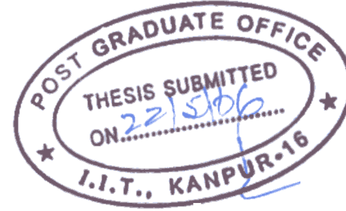
to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY,KANPUR

May 2006

Certificate



This is to certify that the work contained in the thesis entitled *"Functional Simulation Using Sim-nML"*, by *Surendra Kumar Vishnoi*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May, 2006

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

Abstract

Increasing complexity and time to market constraints in the domain of embedded systems have inspired designers to use automated processor modeling tools for rapid design and analysis of various design trade-offs. Given a processor description, these tools facilitate automated generation of processor specific tools.

Sim-nML [Raj98] is a retargetable processor description language used to develop processor modeling tools. In our work, we have developed a tool which takes a Sim-nML processor description as input and generates a functional simulator for that processor. This simulator has been interfaced with GDB to provide a generic debugging environment. The functional simulator can be used to verify correctness of new processor designs.

We have also designed a class hierarchy as an intermediate structure between the Sim-nML description and modeling tools to facilitate easy and efficient processing of the Sim-nML description by various tools. A traversal library has been developed to facilitate tool independent traversal of the class hierarchy. The class hierarchy together with the traversal library provide a unique platform for development of various processor modeling tools.

Acknowledgements

I am grateful to my guide, Dr. Rajat Moona, for his invaluable support and encouragement throughout this work. He was always patient enough to have long discussions with me and provide me innovative ideas whenever I was stuck in my work. His innovative thinking and unrelenting enthusiasm never failed to pave a way for me when I was in a tight spot.

I would like to thank Nitin, for having constructive discussions with me and giving valuable comments. Due to his liveliness and enthusiasm, work never had a dull moment. I am also thankful to B Lakshminath for having the patience to review my report and giving me valuable feedback. Arjun and Mausoom I thank for supporting me morally and making work place enjoyable and lively. Lastly, Dheeraj, Chinmay and Varun, for discussing problems I faced along the way and giving me useful hints. In short, I'd like to thank all Mtech friends for making my stay memorable.

Finally, I would like to express my gratitude toward my parents and my brother for their constant support and encouragement.

Contents

1 Introduction **1**

1.1 Hardware Software Co-design 2

1.2 Retargetable Processor Modeling Tools 3

1.3 Overview of Retargetable Languages and Frameworks 4

1.4 Overview of this Work 6

1.5 Previous Work with Sim-nML 7

1.6 Organization of Report 8

2 Sim-nML **9**

2.1 Introduction 9

2.1.1 Hierarchical tree structure for Instruction set 10

2.1.2 Example processor description 11

2.2 Syntax and semantics of Sim-nML language 14

2.2.1 Instructions 14

2.2.2 The attribute sets 17

2.2.3 Type declarations 19

2.3 Resource-usage Model 22

2.3.1 Resource declaration 23

2.3.2 Registers 23

2.3.3 Uses-attribute 24

2.3.4 Semantics of resource-usage model 25

2.4 Syntax and Semantics of attributes 28

2.4.1 Expressions 29
2.4.2 Operators 30
2.4.3 Special parametrized expressions 33
2.4.4 Sequences 34
2.5 Bit-true arithmetic 35
2.6 Type casting rules 36
2.7 Coercing rules 36

3 Intermediate Representation and Traversal Library 40
3.1 Tabular Structure vs Class Hierarchy 40
3.2 Conversion between Tabular Structure and Class Hierarchy 41
3.3 Traversal Library 42
3.3.1 Traversal of Class Hierarchy 42
3.3.2 Information Passing between Library and Tool Generators . . 43

4 Functional Simulator Generator 45
4.1 Functional Simulation 45
4.2 Processor Model 47
4.3 Processor Model Generation 48
4.3.1 Storage Model Generation 49
4.3.2 Instruction Set Model Generation 49
4.3.3 Implementation of Bit-True Arithmetic 52
4.4 Processing of Executable Binary 54
4.4.1 ELF Format 54
4.4.2 Process Image Creation 56
4.4.3 Decoding of Program 57
4.5 Simulator Generation and Operation 58

5 Interfacing Simulator with GDB 61
5.1 Overview of GDB 61
5.2 Remote-Sim interface of GDB 62

6 **Results and Conclusions** **65**
6.1 Experiments . . . 65
6.1.1 Setting for Experiments . . . 65
6.1.2 Results . . . 66
6.1.3 Analysis of Results . . . 69
6.2 Conclusions . . . 69
6.3 Future Work . . . 70

A **Grammar for Sim-nML language** **75**

B **Class Hierarchy Structure** **87**
B.1 Instruction Set Class . . . 87
B.2 IR Class . . . 87
B.3 Declaration Class . . . 88
B.4 Constant Class . . . 88
B.5 Integer Constant Class . . . 89
B.6 String Constant Class . . . 89
B.7 Real Constant Class . . . 89
B.8 Resource Class . . . 90
B.9 Storage Class . . . 90
B.10 Register Class . . . 91
B.11 Memory Class . . . 91
B.12 Variable Class . . . 91
B.13 Rule Class . . . 92
B.14 Or Rule Class . . . 92
B.15 And Rule Class . . . 93
B.16 IrAttr Class . . . 93
B.17 ImageSyntax Class . . . 94
B.18 AttrSubPart Class . . . 95
B.19 StrSubPart Class . . . 95
B.20 ParamSubPart Class . . . 95

B.21 ExprSubPart Class 96
B.22 Action Class 96
B.23 Uses Class 97
B.24 Clause Class 97
B.25 ResUnitSpec Class 98
B.26 ResUses Class 99
B.27 UsesAttr Class 99
B.28 Param Class 100
B.29 Type Class 100
B.30 BasicType Class 101
B.31 RuleType Class 101
B.32 AttrDef Class 101
B.33 DeclDef Class 102
B.34 ParamUse Class 102
B.35 TypeUse Class 103
B.36 AttrNameDef Class 103
B.37 LiteralDef Class 103
B.38 IntLiteral Class 104
B.39 RealLiteral Class 104
B.40 StrLiteral Class 104
B.41 OprDef Class 105
B.42 Traversal Library Interface 105

List of Figures

1.1 Hardware software co-design flow diagram 2

2.1 Hierarchical tree structure for Instruction set 10
2.2 Sim-nML description for a Simple hypothetical processor 14
2.3 Derivation of Add instruction from description given in figure 2.2 . . 16
2.4 Example Sim-nML description 28
2.5 Reservation table for example shown in figure 2.4 29

3.1 Interaction between traversal library and tool generator 43

4.1 Processor model generation from Sim-nML description 48
4.2 Hypothetical Sim-nML description 51
4.3 Instruction set list building from description of figure 4.2 52
4.4 Instruction set model generation from description of figure 4.2 53
4.5 Application of bit-true arithmetic operations 55
4.6 ELF object file format in two different views 56
4.7 Input file processing and decoding 57
4.8 Generation of simulator 59

5.1 GDB Structure 62

List of Tables

2.1 Type casting rules 38
2.2 Type coercion rules 39

4.1 Mapping between IS model components and Sim-nML attributes . . . 50

6.1 Number of instructions simulated for each test program 67
6.2 Performance results on **Machine1** 67
6.3 Performance results on **Machine2** 68
6.4 Operator library calls to number of simulated instructions ratio . . . 68

Chapter 1

Introduction

Complexity of embedded system design is increasing day by day. An embedded system is a combination of hardware and software. In the traditional way of design, hardware design goes first. It includes custom hardware design as well as the selection of off-the-self components. Once the hardware components are in place, software development starts on top of them. In the end, hardware and software designs are integrated together to form the final system. Owing to design complexity of embedded systems and time to market constraints, this approach is no longer feasible. Some of the major issues are the following.

- Hardware design errors become increasingly expensive to correct as design progresses. If these errors can be detected in the early phases of design, there will be a tremendous reduction in correction cost.

- This approach requires clean separation between functionality to be implemented in hardware and software at the beginning of design itself with no scope for further revisions. This type of design rigidness is not practical in modern design practices.

- Correctness is no longer the only design issue for embedded systems. They must also meet performance, time to market, power, space and flexibility requirements.

These problems forced system designers to follow new design paradigms. In these design paradigms, the design of hardware and software takes place simultaneously. This gives designers an opportunity to analyze various hardware software design trade-offs and explore different design alternatives in the early phases of design.

1.1 Hardware Software Co-design

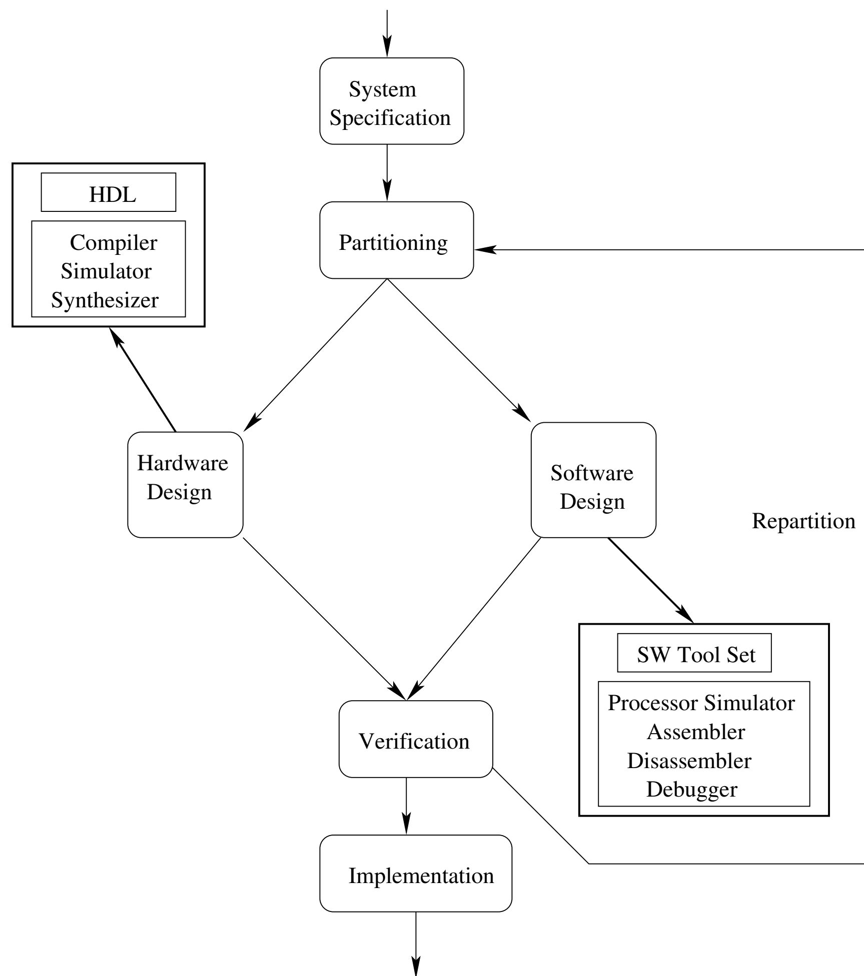


Figure 1.1: Hardware software co-design flow diagram

Hardware software co-design flow diagram is shown in figure 1.1. First step in

co-design is to generate a system specification from the given requirements. The specification is analyzed for various cost criteria such as performance, design time, power and space. After analysis, system functionality is divided into various modules. The next step in co-design is that of partitioning. In this step, decisions are made as to whether a system functionality module is to be implemented in hardware or in software. The key deciding factor in partitioning is optimal satisfaction of various cost criteria. From this point onward, design of hardware and software goes in parallel.

Hardware is designed using hardware description languages (HDLs). HDLs are capable of describing hardware at the lowest level where, the HDL tools facilitate compilation, simulation and synthesis of hardware designs. In co-design, software is developed on programmable processor cores. Programmable cores are flexible and support rapid development. These cores are associated with software tool-sets to create simulation environment of target machine on any given host machine. They typically include tools such as processor simulator, assembler, disassembler, compiler back-end, debugger and profiler. These tools facilitate compilation, simulation and debugging of software for target environment.

Hardware software co-design is followed by design verification using co-simulation. In co-simulation, hardware and software designs are simulated together with help of HDL tools and software tool sets respectively. However, during co-simulation if it turns out that current design is not able to satisfy system cost criteria, design flow loops back to partitioning to explore a new design alternative. If design works well during co-simulation, hardware design is converted to real hardware and then software design is integrated with it to form the final system.

1.2 Retargetable Processor Modeling Tools

Each and every embedded system has its own unique requirements. This forces us to either design a new processor core according to application requirements (Application Specific Processors) or modify an existing one for every new system design. In the later case, there are lots of alternatives available to choose from such as Ten-

silica [ten] and Xilinx Vertex-5 [xil]. However, a new design of a core requires a new set of software tools. Due to continuous advancements in semiconductor technologies, programmable cores are moving toward unprecedented complexity. Software tool-set design for such complex processor cores is a tedious job and requires lot of efforts and time. This problem led to automation of software tool-set generation process.

Automated software too-set generators take a processor model specification as input and generate the desired tool-set based on that processor model. This greatly reduces designers' efforts to mere specification writing. However, this solution leads to another problem of writing of the model specification. The language used to write processor model specifications should be easy to learn and use. On the other hand, it should be powerful enough to cover a large range of processor architectures. A processor description language is used to write the processor model specifications. Target specification for a particular processor is written using the processor description language and tool generators associated with that description language generate the desired tool-set.

1.3 Overview of Retargetable Languages and Frameworks

HDLs, such as Verilog and VHDL have been in use for a long time to write processor designs. However, models in these languages are used to write design specifications at highly detailed level and most of which is not relevant for software tool-sets. Moreover, certain attributes like syntax of processor instructions can't be obtained directly from such descriptions. Failure of HDLs as architecture description languages motivated people to use higher level languages.

SystemC [sys] is an example of such a high level architecture specification language. SystemC is a C++ class library for system level modeling. Not only it supports development of software algorithms, it provides constructs to model hardware and software-hardware interfaces. However, SystemC is targeted more toward

system level modeling rather than processor modeling.

Retargetable languages or architecture description languages(ADLs) are high level languages specifically designed to model processor architectures. Broadly, an ADL covers one or more of the following three models of processor description.

- **Instruction set model:** This part includes syntax, binary representation and semantic behavior of all instructions in instruction set.
- **Storage model:** This part includes memory, registers, register files with ports, aliases and local variable declarations.
- **Timing model:** This part includes resources such as functional units, buffers etc and their modeling such as pipeline flow, hazards, reservation table etc.

Some contemporary ADLs and ADL related research are described in this section.

nML [Fre93] is an ADL based on attribute grammar and describes processors at instruction set level. nML lacks constructs to describe structural details and timing model of processors. Sim-nML is an extended version of nML.

FlexWare [PS02] is an embedded system development environment. It covers all the three parts of processor description model. However, software tool development is not fully automated in FlexWare. It has been used to generate tools such as compiler, assembler, debugger, instruction-set simulator and performance analyzer.

MDes [mde97] model is a part of TRIMARAN [Tri] system. It uses MD language for writing processor descriptions. MDes processor descriptions has been used to generate low level descriptions for tools such as compilers and simulators. However, it provides very limited retargetability. It has constructs to specify resource model i.e. the reservation tables explicitly.

MIMOLA [LEL99] ADL describes a processor at a lower level. These descriptions contain RT (register transfer) type of constructs. Instruction set is extracted from this lower level description. An instruction set simulator was generated using MIMOLA description. MIMOLA descriptions are complex and laborious to write.

ISDL [HHD97] is an ADL with main focus on VLIW architectures. It also provides explicit constraints to define valid operation groupings. However, it lacks

constructs to describe detailed processor structure. ISDL has been mainly used for compiler tools.

RADL [Sis98] is designed to support detailed modeling of processor structure. It explicitly describes typical pipeline aspects. Processor structure is described in a hierarchical form, similar to nML's hierarchical description of processor instruction set. RADL is mainly focused on cycle accurate simulators.

FACILE [SHL01] ADL targets detailed micro-architecture description of processor. It's main focus is on cycle accurate simulators. It uses programming language techniques like partial evaluation and memoization to develop fast-forwarding simulators.

LISA [PHM00] ADL facilitate description of all three processor models. Its descriptions are flat without any sharing of descriptions. LISA descriptions have been used to generate tools like simulator, debugger, assembler, linker and compilers.

EXPRESSION [HGG+99] ADL supports both timing and instruction set models. It facilitates abstract structural description of processor, from which reservation tables can be automatically derived. A unique feature of EXPRESSION ADL is its support for novel memory organizations and hierarchies.

1.4 Overview of this Work

In this work, we have developed a functional simulator generator, which accepts the description of a processor given in Sim-nML processor description language, and generates a functional simulator for that processor. The functional simulator is used to simulate processor behavior at instruction level granularity. It can be used to verify the correctness of a program written for a new processor design. We have also interfaced this simulator with GNU Project Debugger(GDB) [gdb] to provide a generic debugging environment for target applications.

Sim-nML is a language for describing an arbitrary processor architecture. It provides processor description at an abstraction level of the instruction set, thus hiding all implementation specific details. Sim-nML is flexible, easy to use and is based on attribute grammar. It can be used to describe processor architecture for

various processor-centric tools, such as instruction-set simulator, assembler, dissembler, compiler back-end etc, in a retargetable manner. Sim-nML has been used as a specification language for generation of various processor modeling tools. A brief description of these tools is given in section 1.5.

An important part of processor description is timing model. This model is required to capture cycle level details of processor's pipeline execution behavior. Cycle accurate simulators utilize this model to mimic processor's behavior at cycle level. Sim-nML captures timing model of processor using a resource usage model. This model handles processor pipeline behavior by means of a reservation table specification. In the resource usage model, the reservation table specification is not given explicitly. Instead it is derived from the model itself. The resource usage model includes the timing specification and the conditions for the use of entities such as structural units like ALU, FP Unit etc., registers and buffers. However, current model has certain shortcomings. We have modified the resource usage model to address these shortcomings. Our resource usage model supports Sim-nML hierarchical description style.

As a processor description language, Sim-nML has various features to make description writing easy and relatively error-free. However, due to such features it becomes harder for tool developers to directly use the processor description in tool generators. We have designed a C++ class hierarchy as an intermediate structure between Sim-nML processor description and tool generators. Sim-nML descriptions are first converted to this hierarchy and then it is used by various tool generators for easy and efficient processing of processor description. We have also built a traversal library on top of the class hierarchy. Using this library, tool generators can traverse the class hierarchy without bothering about its internal structure.

1.5 Previous Work with Sim-nML

Sim-nML was designed primarily as an extension to nML. As nML was not capable of handling the execution behavior of processor pipeline properly, Sim-nML addressed this issue using its resource usage model. At that time a very simple and primitive

instruction set simulator [Raj98] was designed. Since then, Sim-nML has been used as specification language for generation of various processor modeling tools. A listing of major tools generated using Sim-nML is as follows.

- **Disassembler:** A processor independent symbolic disassembler [Jai99] was designed. To avoid tedious processing of Sim-nML descriptions, an intermediate representation(IR) of processor description was also introduced. This IR was in the form of fix size tables.
- **Functional Simulator:** A retargetable functional simulator(Fsimg) [Cha99] was designed and limited instructions of **PowerPC 603** and **Motorola 68HC11** processors were tested on it.
- **Compiler Back-end:** A tool was [Bha01] designed that reads a Sim-nML specification in intermediate form and generates a partially complete GCC machine description. The tool was tested by retargeting the GCC to **Sparc** processor.
- **Cache Simulator:** A cache simulating environment [A.R99] was developed to provide a basis for benchmarking various caching policies of a given processor.

1.6 Organization of Report

Rest of the thesis is organized as follows. In chapter 2, Sim-nML language is described in detail. In chapter 3, we describe design of the *class hierarchy* and that of *traversal library*. In chapter 4, we discuss the design and implementation of our functional simulator generator. In chapter 5, we look into the interface between GDB and the simulator. Finally, in chapter 6, we conclude with results and future work. In appendices A and B, we provide Sim-nML grammar and class hierarchy structure respectively.

Chapter 2

Sim-nML

2.1 Introduction

Sim-nML is a language for describing an arbitrary processor architecture. It provides processor description at an abstraction level of the instruction set, thus hiding all implementation specific details. Sim-nML is flexible, easy to use and is based on attribute grammar. It can be used to describe processor architecture for various processor-centric tools, such as instruction-set simulator, assembler, disassembler, compiler back-end etc, in a retargetable manner.

Sim-nML description of a processor can be viewed as a programmer's model of the processor. This model consists of the following.

- Syntax and semantics of instruction
- Addressing modes
- Definition of registers and memory
- Resource usage model
- Methods for handling traps and other synchronized events

2.1.1 Hierarchical tree structure for Instruction set

In Sim-nML, an instruction set is described by a hierarchical tree like structure. The hierarchical structure facilitates sharing of description among related instructions in the instruction set. In this tree structure, any path from the root node to a leaf node constructs an individual instruction description. Each non-leaf node contains certain attributes, which can be shared by its descendants.

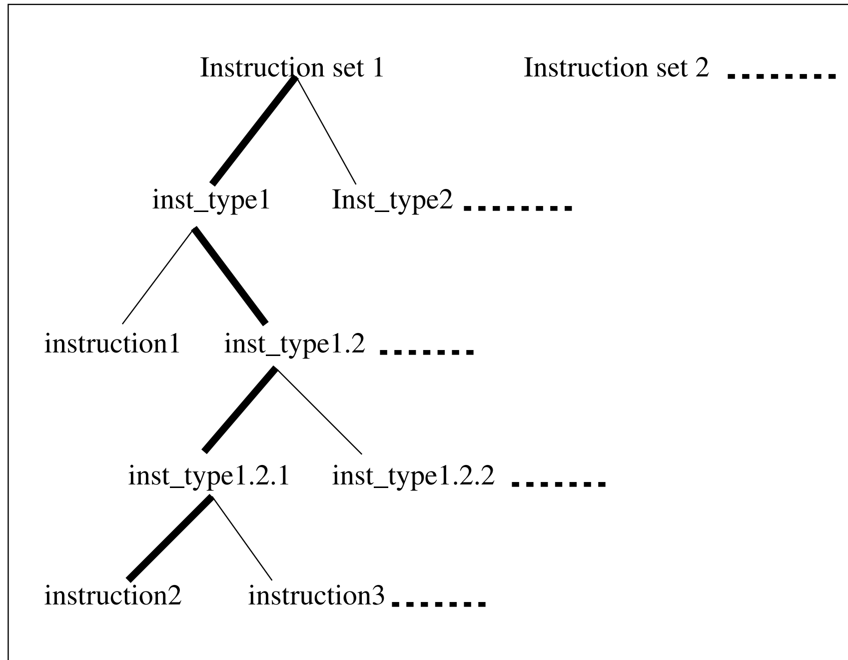


Figure 2.1: Hierarchical tree structure for Instruction set

Figure 2.1 explains description sharing by various instructions. This figure is indeed a forest, consisting of multiple instruction sets. This facilitates description of processors having more than one instruction set (e.g. ARM processor with Thumb instruction set). Let us consider the first instruction set tree in the figure. The root node provides an abstraction of the complete instruction set. Each node in between the root and the leaf nodes represents a set of instructions having certain common features, such as numeric instruction and load/store instruction. Each leaf node represents an individual instruction. It shares all the attributes of its proper

ancestors and describes only remainder of the attributes. Thus, traversal from the root node to a leaf node gives complete description of an instruction. Similar tree structures can be used for other constructs, like addressing modes in the processor architecture.

2.1.2 Example processor description

Figure 2.2 describes a simple processor architecture. This processor supports 64 bytes of external memory and has 16 registers. It supports three instructions, i.e. *Add, Sub* and *Mov*. All of these instructions operate on two operands. There are three addressing modes for operands, i.e. *MEM, REG* and *IREG*. Register PC is used to denote the value of program counter. *Fetch unit, execute unit* and *commit unit* are the available resources (or abstraction of resources) in the processor. All instructions are of 16 bit length. This example description is used as a reference to explain various features of Sim-nML language in this chapter.

```
\\************ Type declarations start ************\\

[1]  let MSIZE = 2**6
[2]  let REGS = 16
[3]  type index = card(6)
[4]  type nibble = card(4)
[5]  type byte = int(8)
[6]  mem M[MSIZE, byte]
[7]  reg R[REGS, byte]
[8]  reg PC[1, byte]
[9]  var SRC1[1, byte], SRC2[1, byte], DEST[1, byte]
[10]  resource Fetch_unit, Exec_unit[3], Commit_unit

\\************ Type declarations end ************\\
```

```
\\************ Addressing modes start ************\\

[11]  mode OPRND = MEM | REG | IREG

[12]  mode MEM(i: index)=M[i]
[13]  syntax = format("(%d)",i)
[14]  image = format("%6b",i)

[15]  mode IREG(i: nibble)=M[R[i]]
[16]  syntax = format("(R%d)",i)
[17]  image = format("00%4b",i)

[18]  mode REG(i: nibble)=R[i]
[19]  syntax = format("R%d",i)
[20]  image = format("01%4b",i)

\\************ Addressing modes end ************\\

\\************ Instruction set starts ************\\

[21] op Instruction(x: arith_mem_inst)
[22]  uses = Fetch_unit #{2}, x.uses, Commit_unit #{2}
[23]  syntax = x.syntax
[24]  image = x.image
[25]  action = x.action
```

```
[26]  op arith_mem_inst(y: Add_sub_mov, op1: OPRND, op2: OPRND)
[27]  uses = y.uses
[28]  syntax = format("%s %s %s", y.syntax, op1.syntax, op2.syntax)
[29]  image = format("%s %s 00%s", y.image, op1.image, op2.image)
[30]  action = {
[31]              SRC1 = op1;
[32]              SRC2 = op2;
[33]              y.action;
[34]              op1 = DEST;
[35]              PC = PC + 2;
[36]           }

[37]  Add_sub_mov = Add | Sub | Mov

[38]  op Add()
[39]  uses = Exec_unit #{2}
[40]  syntax = "add"
[41]  image = "00"
[42]  action = {
[43]              DEST = SRC1 + SRC2;
[44]           }

[45]  op Sub()
[46]  uses = Exec_unit #{2}
[47]  syntax = "sub"
[48]  image = "01"
[49]  action = {
```

```
[50]                DEST = SRC1 - SRC2;
[51]             }

[52]   op Mov()
[53]   uses = #0
[54]   syntax = "mov"
[55]   image = "10"
[56]   action = {
[57]                DEST = SRC2;
[58]             }

\\************ Instruction set ends ************\\
```

Figure 2.2: Sim-nML description for a Simple hypothetical processor

2.2 Syntax and semantics of Sim-nML language

Sim-nML description is based on the attribute grammar. This grammar is acyclic and each non-terminal has at least one production. Thus, any symbol in the grammar having no production rule associated with it is a terminal symbol.

2.2.1 Instructions

There are two orthogonal components in an instruction set. The addressing modes, which define the mechanisms to obtain operands for instructions and the operations performed by the instructions. In Sim-nML, addressing modes are described using *mode-rules*.

Instructions are described using operations and operands. The operations are specified as *op-rules* whereas the operands are specified as parameters to *op-rules*.

The types of parameters that define the operands are the addressing modes specified using *mode-rules*.

Mode and *Op-rules* are arranged hierarchically using production rules. There are two kinds of production rules, *OR rule* and *AND rule*.

Operations

Both the production rules for *op-rules* are as follows.

- *OR rule*

$$\text{op } n_0 = n_1 \mid n_2 \mid n_3 \ldots$$

- *AND rule*

$$\text{op } n_0(\ p_1\text{: } t_1,\ p_2\text{: } t_2,\ p_3\text{: } t_3 \ldots)$$
$$a_1 = e_1\ a_2 = e_2\ a_3 = e_3 \ldots$$

For example, line 37 in figure 2.2 defines an *OR rule*, while line 38 defines an *AND rule*.

For each instruction set of the processor, there is one start symbol (e.g. "Instruction" in line 21). Any terminal string derived from start symbol corresponds to an instruction in the instruction set. This string however does not provide any information regarding syntax and semantics of the instruction. This information inference can be made using attributes attached with the terminals of the string. An example instruction derivation and corresponding attributes are shown in figure 2.3. Here root node corresponds to the start symbol "Instruction" and leaf nodes constitute derived terminal string "Add REG REG". Attributes are shown in rectangular boxes attached with each node in the derivation tree. Dashed nodes and arrows show other possible derivation paths.

In the *AND rules*, t_i is a token (either non terminal or terminal) and is interpreted as type of parameter p_i, where p_i is the corresponding parameter name. Each (a_i, e_i) pair denotes attribute and corresponding definition, respectively for the terminal symbol n_0. For example in the *AND rule* at line 21, "arith_mem_inst" is a token, while "x" is the corresponding parameter name. There are four attributes, *uses,*

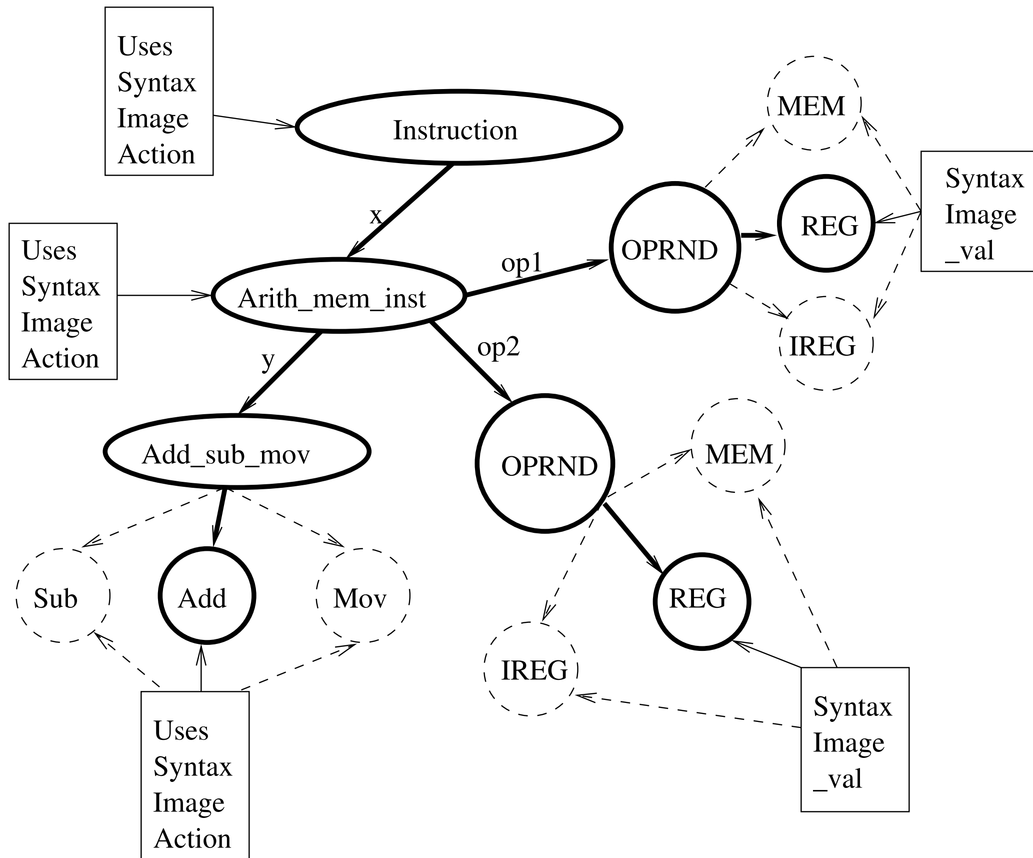


Figure 2.3: Derivation of Add instruction from description given in figure 2.2

syntax, image and *action* with appropriate definitions. More details about attributes are given in section 2.2.2.

The attributes of a descending node in the specification tree can be used while defining an attribute. This can be done using expression such as $p_i.attr$ where p_i defines the descending node. For example, definition of "Instruction" node in line 23 uses x.syntax where x is the descending node "arith_mem_inst" in the specification tree.

OR rules do not have any attribute definitions.

Addressing modes

Sim-nML facilitates easy description of various addressing modes using mode rules. Mode rules are nearly analogous to above described op rules. Keyword "mode" is used to define mode rules.

- *OR rule*

  $$\text{mode } n_0 = n_1 \mid n_2 \mid n_3 \ldots$$

- *AND rule*

  $$\text{mode } n_0(p_1\text{: } t_1,\ p_2\text{: } t_2,\ p_3\text{: } t_3 \ldots) \text{ [= value_assign]}$$
  $$a_1 = e_1\ a_2 = e_2\ a_3 = e_3 \ldots$$

The mechanism to obtain the value of the operand defined by an addressing mode is specified as an optional value assignment at the end of *AND rule*. This value can be thought of as an extra attribute for terminal symbol n_0. Moreover, this value is used as an operand in the *op-rule* whenever a parameter of corresponding addressing mode type is used. In figure 2.2, the mode rule at line 11 is an *OR rule*, while mode rules at lines 12, 15 and 18 are the *AND rules*. All the *AND rules* are followed by value assignments (M[i], M[R[i]] and R[i] at lines 12, 15 and 18 respectively).

2.2.2 The attribute sets

In Sim-nML, attributes are used to describe properties of instructions and addressing modes. Sim-nML facilitates use of arbitrary number of attributes. There are some important predefined attributes. It is the responsibility of description writer to provide appropriate definitions for both self-defined and predefined attributes. All predefined attributes except the *uses-attribute* (described later in details) are explained below.

Syntax-attribute

It describes textual(assembly) syntax of the instruction and evaluates to a string value. The definition part would consists of one of the following.

- *Strings:* Defined simply by putting value in double quotes, as shown in line 40 of figure 2.2.
- *Parameter attribute:* Defined using notation "Parameter.attr", where "attr" is of syntax type. For example, x.syntax at line 23 in figure 2.2.
- *Format expression:* Defined using expression "format" (such as in line 28 of figure 2.2), having similar interpretation as of C function "printf". "Format" expression is described later in details.

Image-attribute

It describes binary coding of the instruction and evaluates to a string of 0s and 1s. White spaces are allowed in resulting binary string for improved readability. The definition part would consists of one of the following.

- *Strings:* Defined Simply by putting value in double quotes, as shown in line 41 of figure 2.2.
- *Parameter attribute:* Defined using notation "Parameter.attr", where "attr" is of image type. For example, x.image at line 24 in figure 2.2.
- *Format expression:* Defined using expression "format"(such as in line 29 of figure 2.2), having same interpretation as of C function "printf". "Format" expression is described later in details.

Action-attribute

It describes semantics of the instruction in terms of sequence of register transfer statements. The definition part consists of either register transfer statements or"Parameter.attr", where "attr" is also of type action. In figure 2, lines 25, 30, 42, 49 and 56 show various definitions for action attribute.

2.2.3 Type declarations

Sim-nML facilitates declaration of constants and macros, data types, memory, registers, temporary variables and resources.

Constants and Macros

In Sim-nML constants are declared by using the following statement.

```
let C=100
```

Here *let* is a reserved word, *C* is the name of the constant and *100* is its value.

After this declaration, C is a global constant and can be used in any context. Global constants are defined only once. Some of these constants may be used to define the behavior of processor tools. For example, a processor simulator may use a constant "ENDIANITY" to implement endianity of the processor. This constant need to be defined in the Sim-nML description of that processor.

Macros are used to define a short hand for arbitrary expressions. They may have parameters embedded in their definition. Macros terminate with a new line. However, they can span multiple lines by adding "\" character at the end of each line except the last one, which obviously terminates with a new line. The nMP preprocessor tool can translate Sim-nML macro definitions to standard m4 macros. A Simple macro definition with two parameters is as shown below.

```
macro comp(A, B) if (A)==(B) then 0 else -1 endif
```

Data types

A data type specifies a range of values for the declared object. A Simple type declaration in Sim-nML is as follows.

type addr = card(32)

Here *type* is a reserved word, *addr* is an object name and *card(32)* is its data type. Sim-nML supports the following primitive data types. (For more examples see line numbers 3 to 5 in figure 2.2)

- **int(n):** This is signed integer data type. Negative numbers are stored in 2's complement form. Here n is number of bits and the possible range of values is $[-2^{n-1} \ldots\ 2^{n-1} - 1]$.

- **card(n):** This is unsigned integer data type. Here n is number of bits and possible value range is $[0 \ldots\ 2^n - 1]$.

- **float:** This is IEEE 754 floating point number.

- **fix(n,m):** This is signed fix format number, having n and m bits before and after the binary point, respectively. The value of a real number r represents $\lfloor r * 2^m \rfloor$ as int(n+m).

- **[n..m]:** This specifies integer or cardinal number in range (n,m) (where n≤m).

- **enum(id_1, id_2, ... id_i):** Defines an enumeration type, where constants id_1=0, id_2=1, ... id_i=i-1. Type of enum will be card($\lceil log_2(i) \rceil$).

- **bool:** This is Boolean data type with two predefined constant values: *false* and *true*. If one coerces these constants to int or card type, *true* is coerced to 1 and *false* is coerced to 0. In the reverse direction, integer 0 is treated as *false* and every other value is treated as *true*.

Memory, Registers and variables

In Sim-nML, memory is modeled as an external entity while registers are internal to the processor. Both represent the user visible state of the processor and across the execution of two instructions only this state is carried. Variables represent temporary storage for facilitating the compact processor description. They do not represent the externally visible state of the processor.

- **Memory**

  A typical memory declaration statement is as given below.

  mem M[N, type] [optional-properties]

  In this declaration, *M* is the name of the memory , *N* is the number of memory locations and *type* is the data type of each location. If no data type is specified, card(8) is default type. Successive memory locations can be accessed using expressions like M[0], M[1] . . . M[n-1]. Memory declarations can have certain optional properties, which are explained below.

  – **Alias**: Describes declared memory as an alias of some other memory as well. Thus, both memories will refer to the same address but with different type interpretations.

    mem A[6, int(32)]
    mem M[3, card(32)] alias = A[3]

    In this example, memory locations A[3], A[4] and A[5] can also be referenced as M[0], M[1] and M[2] respectively. However, memory locations of A are interpreted as 32 bit signed integers while that of M are interpreted as unsigned 32 bit numbers.

- **Register**

  In Sim-nML registers can be declared in the following manner.

  reg R[N, type] [optional-properties]

  In this declaration, *R* is a register file name, *N* is an optional parameter representing the number of registers in the register file and *type* is the data type of each register. If only type is specified, number of registers is taken as 1.

Successive registers are accessed using expressions like R[0], R[1] ... R[n-1]. Register declaration can also have optional attributes, as explained below.

- **Ports**: Describes number of read and write ports for register file.

  reg R[16, int(8)] port = 3, 2

  In above example, register file R has 3 write ports and 2 read ports. Moreover, each register in R consists of 2 read ports, equal to the read ports declared for R itself and one write port. These ports are treated as resources and are used to define the instruction dependencies.

- **Initial**: Describes the initial value for declared register.

  reg R[1, card(32)] initial = 100

- **Variables**

  Temporary variables are typically declared as shown below.

  var TEMP[N, type]

  In this declaration, *TEMP* is a variable array name, *N* is the number of variables in the array and *type* is the data type of each variable. If only *type* is specified, number of variables is taken as 1. Successive variables are accessed using expressions like TEMP[0], TEMP[1] ... TEMP[n-1]. Unlike memory and register declarations, variable declarations do not have any optional attributes. Also, the values of variables are not carried across two instructions.

2.3 Resource-usage Model

In Sim-nML, an instruction is described by associating it with the following two views (of which the timing model view is optional).

- **Instruction semantics**

  In this view, an instruction is described in terms of the operations it will perform, operands of these operations and the resulting value. For this, Sim-nML provides *syntax, image* and *action* attributes as described earlier.

- **Timing model**

  This view describes an instruction by its execution sequence and timing specifications. For this purpose a resource-usage model is used.

The resource-usage model is described using the following constructs.

2.3.1 Resource declaration

A resource is an abstraction of hardware units within a processor, through which an instruction flows during execution. It is not necessarily the hardware implementation, but may be an approximation used to define the timing of execution.
Sim-nML facilitates the declaration of various resource units. A typical example of a declaration is given below.

```
resource Exec_unit[2]
```

In this declaration, *resource* is a reserved word and *Exec_unit* is the name of a resource unit in the processor. Using an optional number after the resource unit name, more than one instances of that particular unit can be declared. The default value is one.

2.3.2 Registers

In Sim-nML, registers and associated ports are also considered as resources. In addition, registers are grouped in a single register file which is also considered a resource. To read a register, one register read port and one register file read port should be available. As stated previously, each register in a register file has number of read ports equal to the total number of read ports for that register file and write

ports equal to one. Thus, number of parallel read operations on a single register is equal to total number of read ports for the corresponding register file. To write a register, all the read ports and write port of that register and one write port of the corresponding register file should be available. Thus, a register should be written exclusively. To capture the behavior of registers as resources, following 5 operations are defined on them.

- **itR:** This declares intention for reading a register value and implicitly demands one read port of the register and one read port of the corresponding register file. Actual read operation can take place only after acquisition of these resources.
- **itW:** This declares intention for writing a register value and implicitly demands one write port and all the read ports of the register and one write port of the corresponding register file. Actual write operation can take place only after acquisition of these resources.
- **Rdone:** This declares actual read operation.
- **Forward:** This declares forwarding of register value to certain other resource unit.
- **Wdone:** This declares actual write operation.

Actual implementation of these operations is dependent on the tool-generator. For example, *itW* operation can be blocking or non-blocking. In former case, instruction is made to wait if resources required for actual *write* operation are not available at that time. While in latter case, instruction will proceed independent of the availability of required resources. However, actual read and write operations could not progress without the availability of demanded resources. These operations together with declared resources describe the complete resource-usage model.

2.3.3 Uses-attribute

This describes the resource-usage model of instructions in a hierarchical manner. An instruction specification includes all the resources required by the instruction in a

timing sequence. As explained in section 2.1, in Sim-nML, instructions are described in a tree-like hierarchical structure. Individual instructions, which are present at leaf nodes, share the attributes of their ancestor nodes. This sharing applies to *usage-attribute* as well. While describing the instruction set for a processor, the resource requirements for every node are specified directly at that node or as a reference to the resource requirements of its children nodes. This description is continued recursively, until it reaches leaf nodes. For example, consider the usage attribute definition in line 22 of figure 2.2. It says that an instruction will require the fetch unit for two units of time, followed by the resources required by parameter "x" (determined by token arith_mem_inst) and in the end, the commit unit for 2 units of time. Parameter "x" acquires resources from parameter "y" (determined by token Add_sub_mov), which corresponds to an *OR* rule and forks into three operations(*Add, Sub* and *Mov*). The *uses-attribute* definitions for these three operations are given at line number 39, 46 and 53 respectively. Both *Add* and *Sub* require the execution unit for 2 units of time, while *Mov* uses no resources.

2.3.4 Semantics of resource-usage model

Semantics of *resource-usage model* can be explained with the help of following constructs.

- **Clauses**
  Resource requirements of an instruction can be modeled by a sequence of resource-use-clauses, separated by ",". An individual clause in the sequence corresponds to one or more resources with different semantics attached to it. An example Sim-nML description for the resource requirements of three instructions is shown in figure 2.4. Instructions acquire and release the resources specified in the clauses in a sequence during execution. In figure 2.4, the resource acquisition sequence for *instruction1* is as follows. It will first acquire either of the two *fetch unit* (clause1), followed by *execution unit* (clause2) and in the end, it requires *store buffer* and one of the *commit unit* (clause3). Thus, a clause may correspond to a single resource or a Boolean combination of

multiple resources.

- **Timing modeling**

  Resources are acquired for fixed units of time, which typically corresponds to the multiple of clock cycles of the processor. However, time unit is an abstract quantity and any other mapping to machine cycles may be assumed. For example, *instruction1* requires all the resource units except *store buffer* for 2 units of time. Multiple resources in a single clause can be acquired for different units of time. For example, in the clause3 of *instruction1*, *store buffer* is required for 1 unit of time while *commit unit* is required for 2 units of time.

- **Conflict resolution**

  All the resources specified in a single clause are either acquired simultaneously or none. However, until an instruction acquires all the resources in the current clause, it will hold the resources of the previous clause. If more than one instruction contends for the same resource, then the conflict is resolved in *FIFO* order. All instructions except the one which acquires the resource, wait for the release of that resource. Moreover, if waiting instructions already have some resources of the current clause, these resources are released.

  The resource reservation table for the example description is shown in figure 2.5 (for simplicity *instruction3* is ignored). As shown in the table, both *instruction1* and *instruction2* contend for the *execution unit* at time unit 3. The conflict is resolved in favor of *instruction1* according to *FIFO* order and *instruction2* is stalled for 2 units of time.

- **Acquisition from multiple choices**

  An instruction can request for more than one resource alternative in a single clause. If at least one resource out of all the specified alternatives is free, then the instruction will not stall. In case more than one of the alternatives is available, allocation is done arbitrarily. In figure 2.4, *instruction1* specifies each of the two *fetch units* as alternatives in the clause1. An alternative syntax for the same is to use "|" operator, as shown in the clause1 and clause3

of *instruction2*. Similarly, instructions can request for more than one resource simultaneously. In this case, either all these resources are allocated simultaneously or none are allocated at all. For example, *instruction2* requires both *commit unit* as well as *store buffer* in the clause3.

- **Conditional acquisition**

  Resource acquisition can be conditional. In this case, an instruction will acquire the resources in a clause if and only if certain conditions specified in the clause hold. In figure 2.4, *instruction2* will acquire the resources in the clause1 if and only if the condition (pipeline == 1) holds. In order to specify such a condition, both conditional expressions and the *if-then-else* construct can be used. However, only constant values are allowed for specifying conditions. Conditional resource acquisition is useful to model certain optional resources in the target processor. For example, *instruction2* can model both the pipelined and unpipelined processor depending on the outcome of the condition specified in the clause1.

- **Book-keeping actions**

  Uses-definition also facilitates the description of an optional *action* after each resource request in a clause. The specified *action* takes place either after resource acquisition or after resource release, depending on where it is declared. In the example description, an action *branch_handler* is specified with *execution unit* (clause2) of both *instruction1* and *instruction2*. However, in the case of *instruction1* the action will take place just after the acquisition of *execution unit*, whereas for *instruction2* it will take place after the release of *execution unit*. *Actions* in the *uses-definition* do not have any semantic meaning attached to them in the context of execution of instructions. They are mainly used for book keeping purpose. Typically such actions can be used for branch prediction and management of cache replacement policy.

- **Instruction sequencing**

  To specify advanced pipeline features like out of order execution, *uses-definition* provides *'['* and *']'* operators. *[* operator marks the arrival order of incoming

instructions. *]* operator regulates the departure order of outgoing instructions according to the marking done by matching *[* operator. All out of order instructions are made to wait until they become aligned with the marked order of arrival. In between a single *[...]* operator pair any order of execution is allowed. For example, *instruction3* has 4 clauses with a unique resource in each clause. Clause2 and Clause3 are enclosed in a *[...]* operator pair. Thus, in the pipeline, *[* operator will note the arrival order of incoming instructions from *fetch unit* to *decode unit*, while *]* will enforce the same order on outgoing instructions from *execution unit* to *commit unit*. Transition of instructions from *decode unit* to *execution unit* can be out of order.

```
Resource Fetch_unit[2], Decode_unit,  Exec_unit, Commit_unit[2]

  Instruction1:
     uses = (Fetch_unit #{2}), Exec_unit : branch_handle #{2},
     (Store_buffer #{1} & Commit_unit  #{2})

  Instruction2:
     uses = { pipeline ==1} (Fetch_unit[1] #{1} | Fetch_unit[2] #{1}),
     Exec_unit #{1} : branch_handle,
    (Commit_unit[1] #{2} | Commit_unit[2] #{2})

  Instruction3:
     uses = Fetch_unit[1]#{1}, [Decode_unit#{1}, Exec_unit#{1}],
     Commit_unit[1]#{2}
```

Figure 2.4: Example Sim-nML description

2.4 Syntax and Semantics of attributes

In Sim-nML, attribute definitions can contain expressions and statement sequences. There are certain assumptions in the language, which one should keep in mind

Resource \ Time	1	2	3	4	5	6	7
Fetch_unit 1	isnt1	isnt1					
Fetch_unit 2		inst2					
Exec_unit			isnt1	isnt1	inst2		
Commit_unit1					isnt1	inst1	
Commit_unit2						inst2	inst2
Store_buffer					isnt1		

Figure 2.5: Reservation table for example shown in figure 2.4

when writing the attribute definitions. In the subsequent sections these issues are explored.

2.4.1 Expressions

An expressions can be one of the following.

- **Constant:** one of the following type.
  - string (e.g. "al")
  - binary (e.g. 0b101010)
  - decimal (e.g. 4 or 65.4)
  - hexadecimal (e.g. 0x34FA3D)
- **Identifier:** any possible combination of alphabets, numbers and _. For example byte, M etc.
- **Attribute reference:** an attribute of an identifier, expressed as "ID.attr". For example x.action, y.image etc.
- **Parametrized expressions:** one of the following types.

- operators and operands like $a + b$
- identifier and parameters like *format("%s", "mov")*
- canonical function like *"sin"(30)*

- **Indexed expression:** one of the following.
  - Memory location (e.g. M[2] or M)
  - Register (e.g. R[3] or PC)
  - Variable (e.g. V[1] or V)
  - Any of the above with bit select (e.g. R[3]< . . >)

- **Conditional:** a conditional statement with if-then-else construct.
  If A<B then . . . else . . .

- **Switch case:**
  switch choice {
  case 0: "Zero byte instruction"
  case 1: "One byte instruction"
  case 3: "Three byte instruction"
  default: "Invalid instruction"
  }

- **Macro:** a macro call corresponding to macro definition given in description.
  Macro call: DIV(6,3)
  Macro definition: macro DIV(a,b) = a/b

2.4.2 Operators

Sim-nML provides variety of operators for easy, flexible and speedy description of the processor. Following is the list of operators with syntax and semantics explained.

- **Binary +, −**
  These are the usual addition and subtraction operators which operate on two

operands. For FLOAT and FIX data types, both operands must be of the same type. In the case of operand mismatch for INT and CARD types, following rules apply.

  - If operands are of different bit width, result will have bit width at most 2 more than the larger bit width.
  - If one operand is INT type and other one is CARD type, result will be of INT type.

- **Unary +, −**

  These operators are used only for INT, FLOAT and FIX data types.

- ***, /, %**

  These are usual multiplication, division and remainder operators, which operate on two operands. If operands are of INT or CARD types, rules Similar to that for binary +,- are applied. However, maximum bit width of result can be equal to twice of the larger bit width. In the case of FLOAT and FIX type operands, mixing with INT or CARD data types is allowed, result type being of that FLOAT or FIX type numbers.

- ******

  This is a double star operator for exponentiation operation. Out of two operands, first can be of any type but second must be a constant. Bit width of result can be determined by assuming this operation to be equal to multiple * operations.

- **<, >, <=, >=, ==, ! =**

  These are usual comparison operators and return a Boolean type of result. For *true* value 1 is returned and 0 is returned for *false* value.

- **<<, >>, &, |, ∧, ∼**

  These are usual bit level operators. To perform left shift and right shift, << and >> are used, while &, |, ∧ and ∼ are used for bit-wise *and, or, xor* and *complement* respectively.

- **<<<, >>>**
  These are left and right rotate operators.

- &&, ||, **!**
  These are logical *and, or* and *not* operators respectively. Non-zero operand is treated as having true value while zero is treated as having false value. After application of these logical operators, result is always of Boolean type.

- **::**
  This is a binary concatenation operator. Operands can be arbitrary expressions. Operands on right side are concatenated and resulting value is assigned to left side. In case, bit width of expression on left side is greater than that of on right side, right side result is sign extended or zero extended before it is assigned to left side. On the other hand, if bit width of left side expression is less than that of on right side, it is assigned the required bits from the second operator in **::** operation. However, if bit width of left side expression is greater than the second operand, the first operand is used for remaining bits.

  M[1] = R[0] :: R[1]

- **Bit-field operator**
  The general signature for bit-field operations is: *location<left_expr . . right_expr>*
  Here location can be a memory location, register or temporary variable. Left_expr and right_expr evaluate to non-negative values, which specify range for bit selection. An example for copying the lower 16 bits of a word to the upper 16 bits is shown below.

  R[0]<16 . . 31> = R[0]<0 . . 15>

2.4.3 Special parametrized expressions

- **coerce(type, value)**

  This expression takes two arguments, value to be coerced and resulting type of the value after coercion. Coercion may not be precise, in that case the value is coerced to the best approximation in coerced type. For example if a floating point number is coerced to an integer type, then fractional part of the floating point number is discarded. Similarly if a signed number is coerced to unsigned one, then 2's complement representation of former is as such copied to latter type. An example to coerce a register of card type to int type is shown below.

  reg R[1, card(32)]
  coerce(int(32), R)

- **format(format-string, args. . .)**

  This expression takes as parameters a format-string and the corresponding list of arguments. It returns a string value. A format specifier is written as %nC, where n is the optional field-width and C can be one of the following.

  - **d:** is used for decimal values to describe the syntax of instruction.
  - **b:** is used for binary values to describe instruction image.
  - **x:** is used for hexadecimal values to describe instruction syntax.
  - **s:** is used for string values to describe both the syntax and image of instruction. However, in the case of instruction image, only binary string is allowed.

  A simple example is shown below.

  format("%s %s %d", "Add", "R[1]", 30)

- **canonical(string, args. . .)**

  or

  "string"(args. . .)

They are known as canonical functions. These type of functions are not predefined in Sim-nML language. It is assumed that description processing tools know their semantics and can handle them. Canonical functions are only used in the definitions of action type of attributes. They, by themselves can't define any attributes directly. In above two styles of writing canonical functions, the first one is obsolete. A simple example of a canonical function to calculate log-base-2 is given below.

"log"(100,2)

2.4.4 Sequences

All attributes in Sim-nML except syntax and image attributes are defined using sequences. A sequence is composed of register-transfer like statements, enclosed in braces ({,}) and separated by semi-colons (;).

```
Sequence = {
        statement1;
        statement2;
        statement3;
        ...
}
```

For example

```
...
action = {
        num = M[0];
        denum = "log"(100, 2);
        if(denum != 0)
                result = num/denum;
        ...
```

```
}
...
```

A statement is one of the following.

- An assignment statement like num = M[0].
- A reference to attribute, either direct like "action" or indirect like "ID.action".
- A call to canonical or error function.
- A conditional statement which is similar to the conditional expression, except, instead of expressions, sequences are used in if and else parts.
- A switch statement which is Similar to the switch expression, except, instead of expressions, sequences are used in case parts.

2.5 Bit-true arithmetic

Bit-true arithmetic is used to resemble the target processor's arithmetic operations as closely as possible. Sim-nML facilitates the declaration of data objects having arbitrary bit length. In arithmetic operations, any of the declared data objects can be used as source and destination operands. This leads to the operands having different bit length.

Consider the following example addition operation.

```
var Result[int(7)];
var Src1[card(3)];
var Src2[card(3)];
...
Result = Src1 + Src2;
...
```

In the above example, two variables of *card(3)* type (*Src1* and *Src2*) are added and result is stored in *Result* of type *int(7)*. Addition of two 3 bit numbers of *card* type gives a result of same type having bit length of at most 4. Thus, before storing the resultant value in the *Result* data object, it will be type casted to int(7). (Type casting rules are explained in section 2.6).

Now consider the second example given below.

```
var Result[int(3)];
var Src1[card(6)];
var Src2[card(6)];
...
Result = Src1 + Src2;
...
```

In this operation, resultant value after the addition operation will be of type *card(7)*. Again, before storing the resultant value in the *Result* data object, it will be type casted to int(3).

2.6 Type casting rules

In Sim-nML, whenever two incompatible data types (either in size or type or both) are used in an assignment statement, the casting rules shown in table 2.1 apply. Each table entry corresponds to a type casting rule between source and destination data type. In all the rules, truncation and zero extension start from the most significant bits.

2.7 Coercing rules

As explained in section 2.4.3, one data type can be explicitly converted to another data type using expression *coerce*. Type coercion rules are shown in table 2.2. In

all the rules except one, truncation and zero extension start from the most significant bits. In the exceptional rule, truncation and zero extension start from least significant bits and it applies between card(m) and card(n) types.

Source type					
Destination type		**int(m)**	**card(m)**	**fix(m,k)**	**float**
	int(n)	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Sign extend source to n bits	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Zero extend source to n bits	Treat fix(m,k) as int(m+k) and apply the rule for int(n) and int(m+k). For example, value 1.25 is 001.01 in fix(3,2) format and after casting to int(6) it becomes 000101 (=5). On the other hand, if it is casted to int(2), it becomes 01 (=1).	Treat float as int(32) and cast to int(n)
	card(n)	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Sign extend source to n bits	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Zero extend source to n bits	Treat fix(m,k) as int(m+k) and apply the rule for card(n) and int(m+k). For example, value -1.25 is 111.01 in fix(3,2) format and after casting to card(6) it becomes 111101 (=61). On the other hand, if it is casted to card(3), it becomes 101 (=5).	Treat float as int(32) and cast to card(n)
	fix(n,l)	Treat fix(n,l) as int(n+l) and apply the rule for int(n+l) and int(m)	Treat fix(n,l) as int(n+l) and apply the rule for int(n+l) and card(m)	Treat fix(n,l) as int(n+l) and fix(m,k) as int(m+k). Apply rule for int(n+l) and int(m+k). For example, value 3.75 is 011.11 in fix(3,2) format and after casting to fix(2,3) it becomes 01.111 (=1.8125).	Treat float as int(32) and cast to fix(n,l)
	float	Treat float as int(32) and cast int(m) to it	Treat float as int(32) and cast card(m) to it	Treat float as int(32) and cast fix(m,k) to it	No operation

Table 2.1: Type casting rules

		Source type			
		int(m)	**card(m)**	**fix(m,k)**	**float**
Destination type	**int(n)**	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Sign extend source to n bits	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Zero extend source to n bits	Discard the fraction part and apply the rule between int(n) and int(m) for integer part	If the float value is f then put $\lfloor f \rfloor$ into int(n)
	card(n)	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Sign extend source to n bits	Case(m=n): Copy source as such Case(m>n): Truncate source to n bits Case:(m<n): Zero extend source to n bits	Discard the fraction part and apply the rule between card(n) and int(m) for integer part	If the float value is f then put $\lfloor f \rfloor$ into int(n) and then coerce that value to card(n). For example, to coerce float -3.75 to card(4), first put $\lfloor -3.75 \rfloor$=-4 into int(4)=1100 and then coerce it to card(4)=1100 (=12)
	fix(n,l)	Make fraction part =0 and apply the rule between int(n) and int(m) for integer part	Make fraction part =0 and apply the rule between int(n) and card(m) for integer part	Apply the rule between int(n) and int(m) for integer part and the exception rule between card(l) and card(k) for fraction part. For example, value 1.75 is 001.11 in fix(3,2) format and after coercing to fix(2,3) it becomes 01.110 (=1.75).	If the float value is f then put $\lfloor f \rfloor$ into int(n) and fractional part into int(l)
	float	Convert to float	Convert to float	Convert to float	No operation

Table 2.2: Type coercion rules

Chapter 3

Intermediate Representation and Traversal Library

As a processor description language, Sim-nML has various features to make description writing easy and relatively error-free. However, such features also make it harder for tool developers to directly use the processor description as input to tool generators. To avoid this problem, tool generators convert Sim-nML description to a low level intermediate form, which would be relatively easy and efficient to process. However, in that case the conversion is needlessly repeated by each tool generator. This suggests that an interface is required between the processor description and input to the tool generator. We provide this interface in form of an *intermediate representation (IR)* obtained by parsing the processor description.

3.1 Tabular Structure vs Class Hierarchy

In previous work on Sim-nML [A.R99], the IR was organized as a collection of tables that store the relevant information extracted from a processor description. It was designed to keep the tables simple for easy handing of IR. Such simplifications demanded that all entries in a given table be of the fix size. In such scheme, various other tables were introduced to keep the information that would otherwise require a table entry to be of variable size. These extra tables necessitate indirect indexing to

access the processor information. As an end result the process of tool development became complicated and tool developers spent efforts in IR information access rather than developing the features of the tool. Moreover, the table structure has no natural resemblance to the structure of the processor description. In the new version of IR, we have IR information in form of a *class hierarchy*. The detailed description of the class hierarchy is given in appendix B. The new IR has the following advantages.

- It represents the processor description in a natural way and avoids the indirect indexing of the table format.
- It supports object oriented design, which encourages hiding of internal details of the IR from tool generators.
- It provides a generic structure to store tool specific information in the IR, independent of tool generator.

The earlier IR was designed to keep the IR in the file, a feature that was thought necessary at that time. However, parsing of the description is fast enough and hence the requirement of eliminating parsing overhead by keeping the IR in a file is not necessary. In this design, the IR is in-memory data structure and is used by the tool generators by parsing Sim-nML description.

3.2 Conversion between Tabular Structure and Class Hierarchy

To maintain compatibility with previous IR format, we have conversion routines to convert the tabular structure to the class hierarchy. In the tabular structure, IR tables are dumped into a file after parsing Sim-nML description. This file is processed by tool generators to read the tabular structure. We are using an intermediate class structure to read tabular IR information in our approach. The structure of intermediate classes is similar to that of IR tables. After reading the IR file into the intermediate structure, the class hierarchy is created using tabular information. The class hierarchy is designed in such a way that it can be created from processor

description using a parser. Moreover, the same intermediate class structure is also used to convert the class hierarchy back to the tabular structure.

3.3 Traversal Library

Tool generators need to traverse the class hierarchy for processing of the IR information. In spite of having its specific information requirements, a tool generator has to go through the complete class hierarchy as the required information may be scattered throughout the hierarchy. In order to avoid traversal by all tool generators, we have designed a traversal library. Although, the traversal of class hierarchy is a fairly straight-forward task, real challenge lies in making this library generic enough to handle traversal request of all tool generators.

3.3.1 Traversal of Class Hierarchy

Sim-nML class hierarchy is an *and-or* tree structure. An *and* node's parameters are its children, while an *or* node has its children as possible alternatives to choose from. The ultimate goal of traversal is to go from the root node to leaf nodes and collect desired information during this process. This goal is achieved by collaboration between the traversal library and the tool generator. The library provides the desired path in the hierarchy tree while the tool generators retrieves the desired information. A tool generator might require either of the following two types of traversals.

- **Guided traversal:** In guided traversal, path selection from root node to a certain leaf node is guided by the tool generator. For example, an disassembler generator will guide traversal according to image of current instruction while collecting the corresponding syntax information. Thus, in this case the library provides a path of the choice of the tool generator and then the tool generator retrieves the desired information from this path.

- **Complete traversal:** In complete traversal, the library enumerates all possible paths in the hierarchy tree to the tool generators and the tool generator

retrieves the desired information from them. For example, to generate all possible instruction images in a processor description, complete traversal is the right choice.

During the traversal process, *or* nodes act as decision making points. Their children provide possible alternatives for the traversal process to continue on. In case of guided traversal, tool generator helps library to choose one of many such alternatives. However, in complete traversal, all possible alternatives are explored.

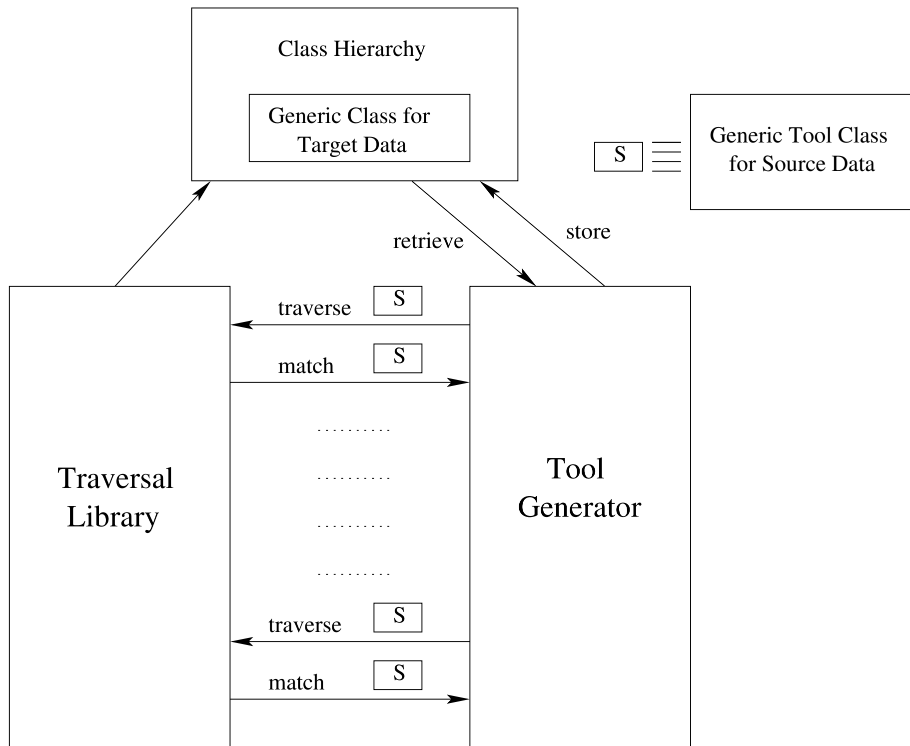


Figure 3.1: Interaction between traversal library and tool generator

3.3.2 Information Passing between Library and Tool Generators

Tool generators rely on two types of data - source data and target data. During traversal, both of these data items change dynamically. In guided traversal, source

data is used to guide the traversal while target data is collected during this process. However, in complete traversal, source data has no significance and traversal is used to collect target data. Source data is an axis around which traversal is carried out while the target data is the information collected during this traversal. Both source and target data are specific to tool generators. The traversal library is independent of these data. However, traversal is a recursive process and it supports backtracking in the case of guided traversal. This necessitates passing of both source and target data through the traversal library.

Data passing between the traversal library and a tool generator is shown in figure 3.1. To handle source data, we pass a generic tool class pointer between library routines and tool routines as a parameter. This class contains source data and other tool specific data and functions. All tool generators derive their specific derivations from this generic class. On the other hand, to handle target data, we provide a generic class in the class hierarchy itself. This class works as a space holder for target data. Tool generators derive their specific derivations from this generic class. The *Rule class* in the class hierarchy holds a list of pointers to this class.

Chapter 4

Functional Simulator Generator

We have implemented a functional simulator generator based on Sim-nML descriptions.

4.1 Functional Simulation

In functional simulation, the behavior of a processor is simulated at the granularity of instruction level for a given input program. Simulator mimics the processor state after execution of each instruction in the program. A processor's state is defined in terms of values contained in its registers, flags and memory. Micro-architecture details of the processor are ignored, e.g. pipeline behavior of the processor is not simulated at all. Functional simulation is useful for verifying the correctness of programs written for new processor designs. It can also be used to generate execution trace of a program for the target processor which can be later utilized by various tools like profiler, cycle accurate simulator, cache simulator etc.

Two most widely used simulation techniques are explained below.

- **Interpretive Simulation**
  An interpretive simulator mimics standard processor execution sequence. This is the traditional technique of simulation. In this technique, the simulator iterates over the following three steps until simulation ends.

  - **Fetch**: In this step, next instruction is fetched from program memory for execution.
  - **Decode**: In this step, instruction is decoded to find out operation to be performed and involved operators.
  - **Execute**: In this step, the designated operation is performed and processor state is updated. Results are stored in appropriate memory locations or registers.

  Interpretive simulators are inherently slow, as the overhead of fetch and decode steps is of the order of total number of instructions executed in the program.

- **Compiled Simulation**
  A compiled simulator breaks a program simulation in the following two phases.
  - **Decoding**: In this phase, all the instructions in the program are decoded and corresponding operations and operands are stored in a table.
  - **Execution**: In this phase, all the instructions are executed using the table created during the decoding phase. The *fetch* step is modeled as selection of the index of next execution entry in the table.

  Compiled simulators improve simulation speed significantly [RMD03] in comparison to interpretative simulators as decoding overhead is of the order of total number of instructions in the program rather than that of total number of instructions executed in the program. However, they do not support self-modifying codes as decoding is done prior to the program execution. If we modify an instruction later during the execution, that modification will be ignored as decoded entry is still the old one.

We have implemented a compiled simulation approach in our functional simulator generator. The simulator generation process is carried out in two steps. In the first step, a processor model is generated using the target Sim-nML description. In the following step, after processing of the target program, it is decoded using the processor model and the functional simulator is generated. Detailed explanation of these steps is given in the following sections.

4.2 Processor Model

Functional simulator takes as input a binary file compiled for the target processor and then simulates it instruction by instruction. To simulate an instruction, simulator must know its execution behavior and then according to that behavior it should change processor state. We need to derive this information from Sim-nML description of the target processor. From functional simulator generation point of view, we are interested in the following models of Sim-nML description.

- **Storage model**
  - Processor registers
  - Processor flags
  - Processor memory
  - Variables local to description

  The storage model of a processor defines its visible state. However, local variables in the description are exception as they do not contribute to the processor state.

- **Instruction set model** Instruction set model is created by collecting the following entities for every instruction in the processor instruction set.
  - *Binary image*: It represents instruction binary format. However, parameter fields are always set to 0s.
  - *Binary mask*: It is obtained from instruction binary format by setting opcode field to all 1s and parameter fields to all 0s. If we *bit-wise and* the binary mask with instruction binary format, we get binary image.
  - *Binary masks of all parameters*: It is obtained from instruction binary format by setting everything except the target parameter field to 0s. The target parameter field is set to all 1s.
  - *Execution behavior*: It represents execution semantics of an instruction.

Instruction set model defines the complete instruction-set of a processor. The first three entries of the instruction set model are relevant from instruction decoding point of view whereas the last one is from instruction execution point of view. Given the instruction set model, we can decode and execute any instruction supported by Sim-nML processor description.

4.3 Processor Model Generation

The processor model generation process and involved modules are shown in figure 4.1. We have used C as host language to specify processor model.

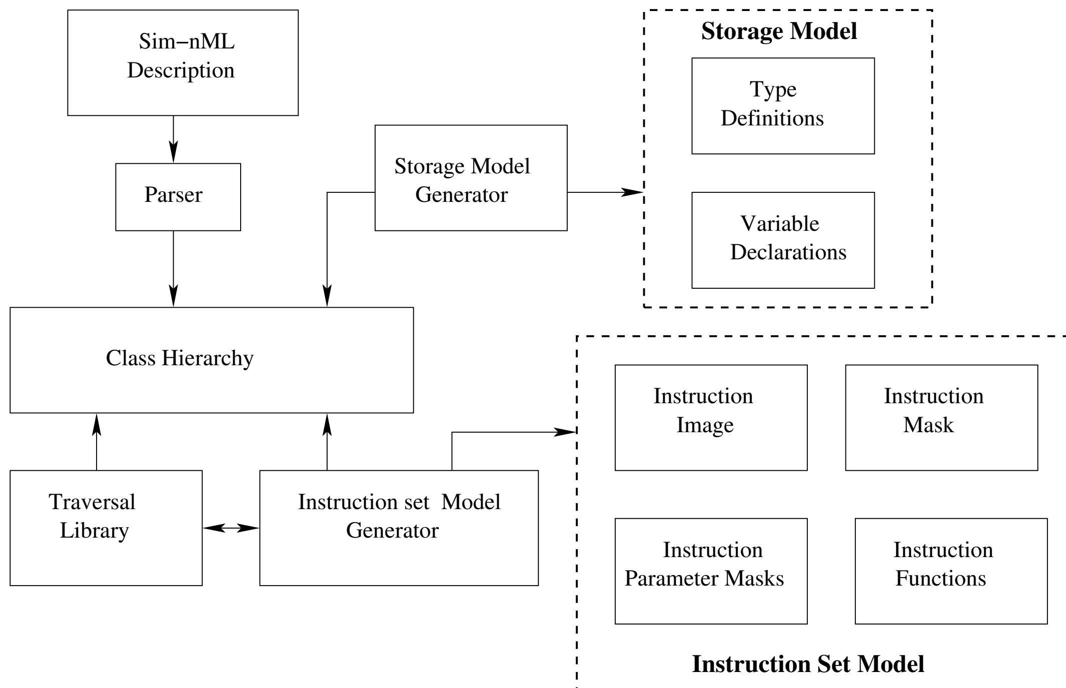


Figure 4.1: Processor model generation from Sim-nML description

The first step is to convert Sim-nML description into the class hierarchy. For this purpose, either we can use a parser capable of generating the class hierarchy directly from description or a parser which will generate the tabular structure from description. In the later case, tabular structure will be converted to class hierarchy

using the intermediate class structure. This conversion process need not be repeated for other tool generators as long as they are using the same description. Thus, given a Sim-nML description, the conversion step is carried out only once and the generated class hierarchy can be used by other tool generators as well.

4.3.1 Storage Model Generation

Steps involved in the storage model generation are as follows.

- The storage model generator module accesses the class hierarchy to generate type definitions and variable declarations. Type definitions are representation of Sim-nML data types in equivalent host language types. If there is no equivalent host language data type for a given Sim-nML data type, we map that Sim-nML data type to next higher host language data type. For example, a 3 bit integer of Sim-nML is mapped to a *signed char* in the host language.

- Sim-nML variable declarations are converted to equivalent declarations in host language. These declaration utilize type definitions generated in the previous step.

4.3.2 Instruction Set Model Generation

The instruction set model components are derived from the attributes and the parameter of Sim-nML *and rules*. The mapping between the instruction set model components and the corresponding Sim-nML attributes used to obtain these components is shown in table 4.1.

As an example let us consider the hypothetical Sim-nML description shown in figure 4.2.

The mapping for the ADD rule in this figure is the following.

- *Binary image*: The value of binary image is $< 0000010000000000 >$. It is obtained by concatenating the opcode field $< 000001 >$ with parameter fields $< 00000 >$ and $< 00000 >$ of x and y respectively.

Instruction set model	**Sim-nML attributes**
instruction image	image attribute
instruction mask	image attribute
parameter masks	image attribute
execution semantics	action attribute

Table 4.1: Mapping between IS model components and Sim-nML attributes

- *Binary mask*: The value of binary mask is $< 1111110000000000 >$. It is obtained by concatenating the opcode mask $< 111111 >$ with parameter field masks $< 00000 >$ and $< 00000 >$ of x and y respectively.

- *Parameter mask:* The value of the parameter mask for x is $< 0000001111100000 >$. It is obtained by concatenating the opcode mask $< 000000 >$ with parameter field masks $< 11111 >$ and $< 00000 >$ of x and y respectively. Similarly, the value of the parameter mask for y is $< 0000000000011111 >$. It is obtained by concatenating the opcode mask $< 000000 >$ with parameter field masks $< 00000 >$ and $< 11111 >$ of x and y respectively.

- *Execution semantics*: The execution semantics is $< R[x] \; = \; R[x] \; + \; R[y] >$.

The goal of the instruction set model generator is to collect the above mentioned Sim-nML constructs for all instructions in the target processor instruction-set. It utilizes the class hierarchy and traversal library's complete traversal utility for this purpose. Steps involved in the instruction set model generation are as follows.

- Instruction set model generator starts traversal of Sim-nML instruction-set tree from the root node. After reaching a leaf node, it starts building the partial lists of parameters and image and action attributes toward the root node. In case of an *or* node, the partial lists of its children are concatenated together to form its partial list. On the other hand, in case of an *and* node, its partial list is constructed through the partial lists of its parameters. This process continues till we reach the root node. The root node contains the lists

```
op BIN_OP = ADD | SUB

op ADD{x : int(5), y : int(5))
image = format("000001%5b%5b", x, y)
syntax = format("add R%d, R%d", x, y);
action = {
      R[x] = R[x] + R[y];
}

op SUB{a : int(5), b : int(5))
image = format("000010%5b%5b", a, b)
syntax = format("sub R%d, R%d", a, b);
action = {
      R[a] = R[a] – R[b];
}
```

Figure 4.2: Hypothetical Sim-nML description

of rule parameters and the image and action attributes in Sim-nML description for all possible instructions in the target processor instruction set.

- Parameter lists together with image attribute lists are used to generate instruction image, instruction mask and parameter masks for all possible instructions in the instruction set of target processor.

- Parameter lists together with action lists is used to generate a list of instruction functions for all possible instructions in the instruction set of the target processor. Action attributes are stored as expression trees in the class hierarchy. To generate instruction functions from action attribute, its expression tree is traversed in in-order sequence. However, to handle certain sub expressions, traversal might temporarily go out of in-order sequence. During this traversal, the parameter list is used to generate parameter references in action attribute. The instruction functions are generated in the host language.

The process of building of instruction-set list and generation of instruction-set model

using the example Sim-nML description given in figure 4.2 is shown in figure 4.3 and figure 4.4 respectively.

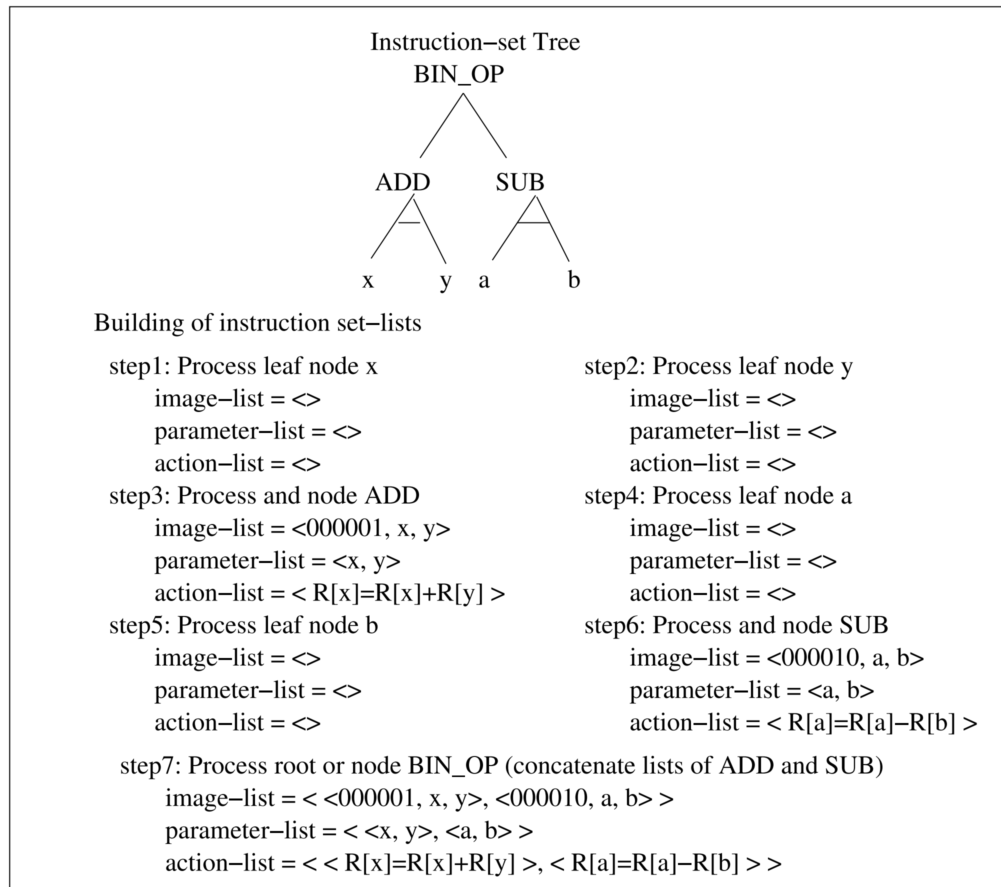


Figure 4.3: Instruction set list building from description of figure 4.2

4.3.3 Implementation of Bit-True Arithmetic

Sim-nML supports bit-true description of a processor architecture. To support bit-true evaluation of all expressions in our simulator, we generate C code that evaluates the correct expression. The evaluation is carried out using the following operations.

- **Type Pull**: Given an operator and its operands along with their types and bit widths, this operation calculates the type and bit width of the result of

```
Instruction–set model generation
    image–list = < <000001, x, y>, <000010, a, b> >
    parameter–list = < <x, y>, <a, b> >

    instruction–image =>  < <0000010000000000> <0000100000000000> >
    instruction–mask =>   < < 1111110000000000> <1111110000000000> >
    parameter–masks =>  < < <0000001111100000>, <0000000000011111> >,
                             < <0000001111100000>, <0000000000011111> > >

     parameter–list = < <x, y>, <a, b> >
     action–list = < < R[x]=R[x]+R[y] >, < R[a]=R[a]–R[b] > >

     action–semantics =>   < < R[p1]=R[p1]+R[p2] >,   (p1=x, p2=y)
                              < R[p1]=R[p1]–R[p2] > >  (p1=a, p2=b)
```

Figure 4.4: Instruction set model generation from description of figure 4.2

that operation. For example, if two 3 bit integers are added, result will be an integer of 4 bit. In an expression tree, this operation is applied from bottom to top. As data types for leaf nodes are known in advance, data types of internal nodes are calculated bottom up from leaf nodes to the root node in an incremental way.

- **Type Correction**: After application of a type pull operation and calculation of resulting data type for an expression, if it is found that host language data types are not capable of producing a result of that precision, operands of that expression are type casted to new host language data types capable of producing the desired result. For example, if two 32 bit integers are added, type pulled result will be an integer of 33 bit. However, in C language, addition of two 32 bit integers will always be a 32 bit integer on a 32 bit machine. Thus, to avoid loss of one bit in result, both operands are sign extended to 64 bit data types. In an expression tree, this operation is applied from top to bottom. Type correction of a node is applied to all its descendants in a recursive manner.

- **Type Push**: In an assignment operation, if bit width of l-value is smaller than that of r-value, we can possibly save computation. In type push operations, data types of operands are truncated to an extent that precision of resulting data type is maintained. For example, if two 32 bit numbers are added and assigned to a 8 bit number, both numbers can be truncated to 8 bits without compromising the accuracy of result. In an expression tree, this operation is applied from top to bottom. Type push of a node is applied to all its descendants in a recursive manner.

- **Type Coerce**: This operation is used to implement *type coerce* operation of Sim-nML language. In this operation, type and bit width specified in coerce operation are forcefully applied to the target operand.

An example of above mentioned operations is shown in figure 4.5.

4.4 Processing of Executable Binary

Currently, our simulator supports ELF object files. It is targeted for statically linked binaries.

4.4.1 ELF Format

ELF [TIS95] stands for "executable and linkable format". It is a portable object file format and extends across multiple operating environments. There are the following three types of object files.

- **Relocatable File**: It holds code and data suitable for linking with other object files to create an executable or a shared object file.

- **Executable File**: It holds code and data suitable for execution and provides the information to create a program's process image.

- **Shared Object File**: It holds code and data, which can be either linked with other relocatable and shared object file to create a new relocatable file or with other executable and shared file to create a new executable file.

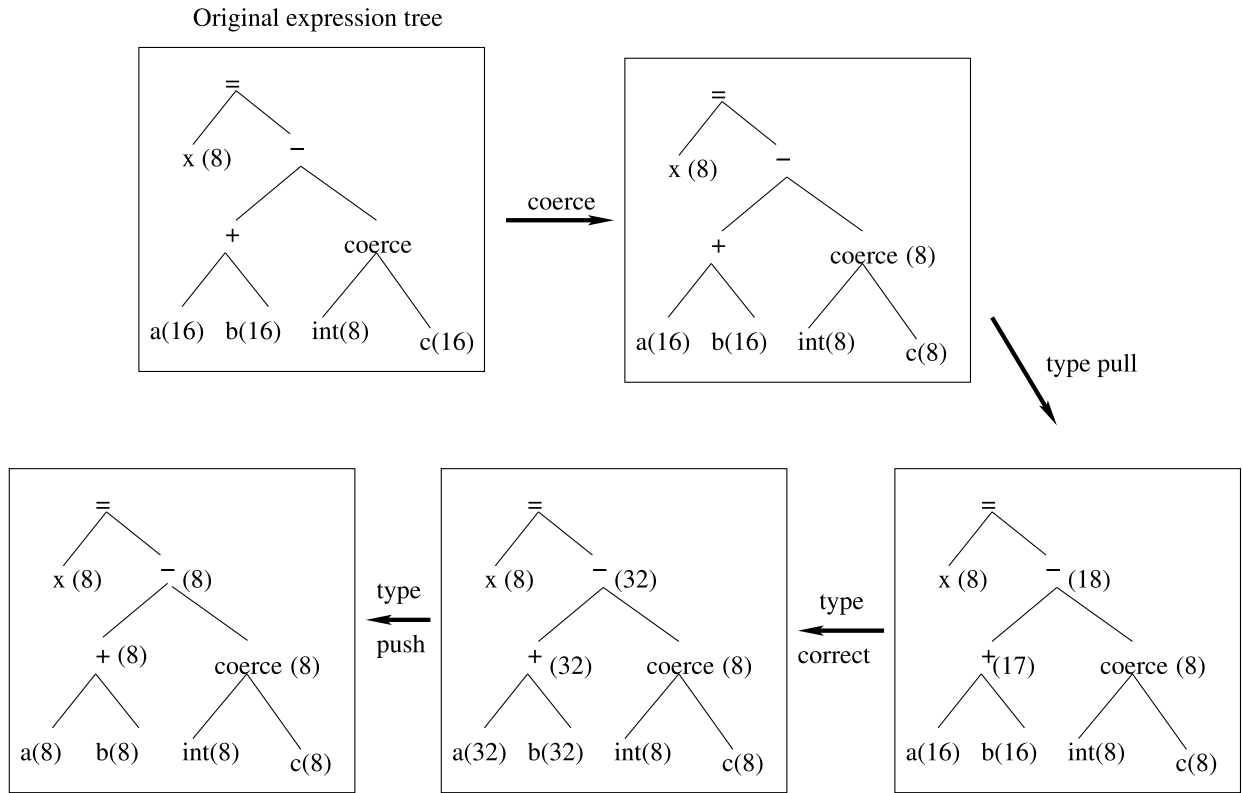


Figure 4.5: Application of bit-true arithmetic operations

ELF object file format is shown in in figure 4.6[1] for two different views.

First header in an ELF file is the ELF header. It is the only header with a fixed position in the file. In a linking view, an ELF file is composed of sections with one section header and one optional program header. On the other hand, in an executable view, the file is composed of segments with one program header and one optional section header. We are using a open source C++ library [Fin] to read ELF files.

[1]Reproduced from [TIS95]

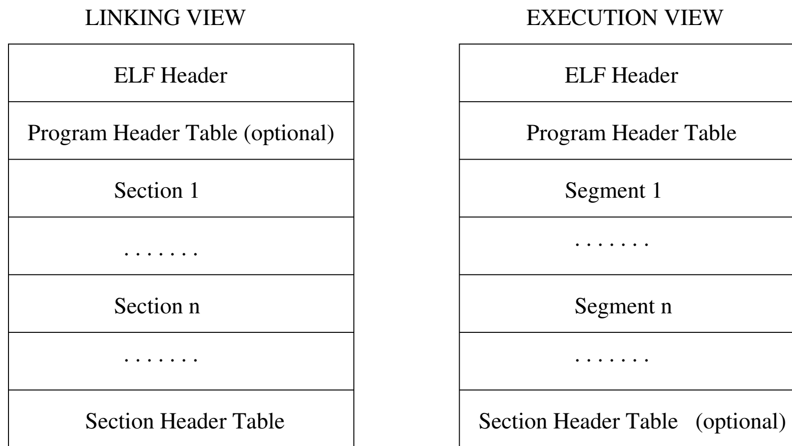


Figure 4.6: ELF object file format in two different views

4.4.2 Process Image Creation

Processing of an input file and involved modules are shown in figure 4.7. We are extracting the following information from an input program.

- *Start Address*: It is address of the start-up routine in the program.
- *Code Base*: It is the lowest virtual start address among all code sections.
- *Data Base*: It is the lowest virtual start address among all sections, which are going to hold space in process image.
- *Stack Base*: It is the start address of program stack.
- *Heap Base*: It is the start address of program heap.
- *Program Code*: Executable instructions in the program.
- *Program Data*: It includes initialized data, uninitialized data and zero initialized space for *.bss* sections.

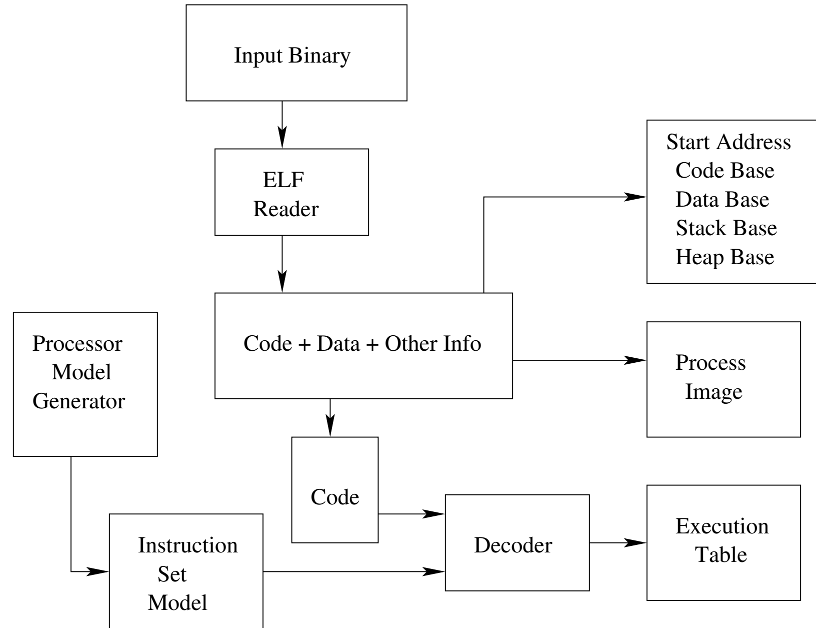


Figure 4.7: Input file processing and decoding

4.4.3 Decoding of Program

We are using compiled simulation technique. Decoded entries of a program constitutes its execution table. As shown in figure 4.1, we generate image, mask, parameter masks and function code for all possible instructions in the described instruction set. This information is stored in a instruction-set table. However, instead of instruction functions, their names are stored in the table. Actual functions are dumped into a C file, which is later used by the simulator. On the other hand, we extract program code from input binary as shown in figure 4.7. First step in the decoding process is to determine whether we are dealing with a RISC architecture or a CISC architecture. For this purpose, minimum and maximum length of an instruction is determined from all possible instruction images. If these two lengths turn out to be equal, we are dealing with a RISC machine otherwise we have a CISC machine as the target architecture. We are using different techniques to decode RISC and CISC architectures.

- **RISC Architecture:** RISC architectures have fixed length instructions. This makes decoding process relatively easy. We start with extracting number of bytes equal to an instruction's length from the program code. To decode this instruction, it is masked with an instruction mask and compared with the corresponding instruction image in the instruction-set table. We iterate over the instruction-set table till a match is found. The next step is to find instruction parameters using instruction parameter masks. This complete process is repeated till all instructions in the program code are decoded. However, if an instruction does not match with any of the instructions in the instruction-set table, a dummy decoding entry is created for this instruction.

- **CISC Architecture:** CISC architectures have variable length instructions. This makes decoding process complicated, as we do not know instruction length in advance. To handle this problem, we extract number of bytes equal to minimum instruction length in the instruction-set. These bytes are compared against the instruction-set table. If no match is found in the instruction-set table, we add one more byte from the program code to current byte stream. This process is repeated till a match is found or byte stream length equals to maximum instruction length in the instruction-set. If a match is found, we extract instruction parameters using the corresponding parameter masks and advance over those many bytes in the program code. However, in the case of a failure, we create a dummy decoding entry and advance over only one byte in the program code. This whole process is repeated till we reach to the end of the program code.

4.5 Simulator Generation and Operation

Complete simulation generation process is shown in figure 4.8. The processor model generator generates a storage model and an instruction-set model. The storage model consists of processor state variable declarations and corresponding type definitions. On the other hand, the instruction-set model consists of C functions to simulate semantics of all the instructions in the instruction-set. The binary file pro-

cessor generates the program image and corresponding state variables. The decoder incorporated inside it, generates an execution table. The execution table consists of parameters and address of target function for all instructions in the input program.

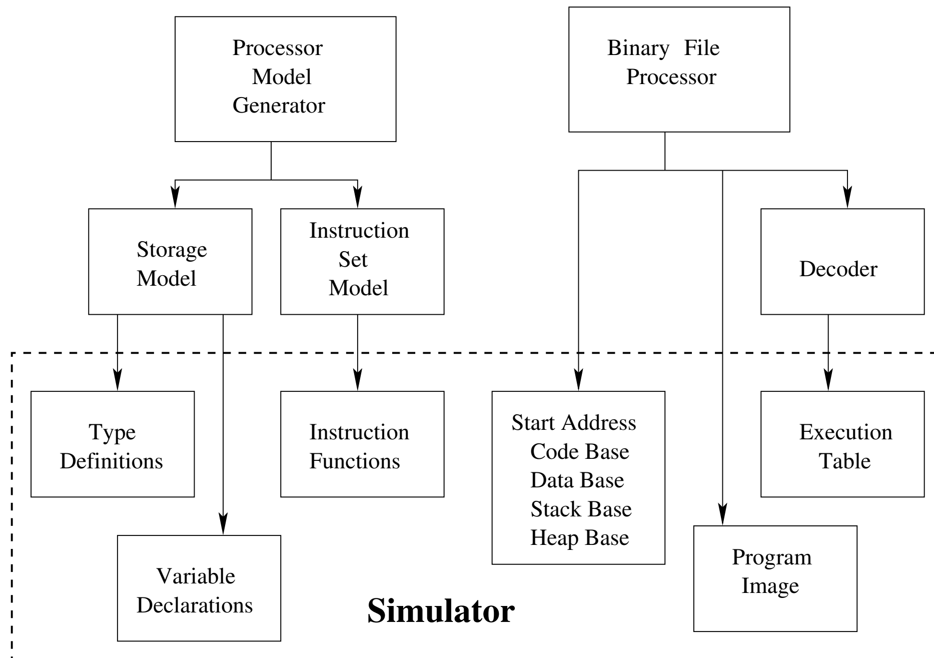


Figure 4.8: Generation of simulator

We require the following modules in addition to those mentioned above in order to make our simulator work properly.

- **Main Program**

  It is an infinite loop which calculates index of the next instruction to be executed from the execution table. Index calculation can be done using the following simple formulas.

$$index = (NIA - CODE_BASE)/INST_LEN \qquad \text{(RISC Architecture)}$$

$$index = (NIA - CODE_BASE) \qquad \text{(CISC Architecture)}$$

  where NIA = Next Instruction Address.

After index calculation, it retrieves instruction parameters from the indexed execution table entry and assigns them to a global parameter structure. All the C instruction functions use this global parameter table for their parameter references. In the next step, it calls the instruction function corresponding to the indexed table entry. Finally, it loops back to index calculation.

- **Interface to Operating System**

  This interface is used to provide a limited application binary interface (ABI) for target program. It includes initialization of certain processor state variables. One such example is initialization of the stack pointer register. One important function of this ABI is to provide a system call interface to host operating system. As the simulator is not capable to handle system calls by itself, it diverts these calls to host operating system using this interface. The simulator provides a system call number and parameters for that system call as input parameters to this interface. The interface returns the result of system call back to the simulator.

- **Canonical Functions**

  As discussed in section 2.4.3, Sim-nML facilitates specification of user defined functions through canonical functions. It is the responsibility of description writer to provide definitions of these functions to the simulator. In the current implementation, system calls are also specified through canonical functions.

- **Operator Library**

  There are certain Sim-nML operators having no corresponding operator in the host language. We handle most of these operators by expressing them as a combination of one or more host language operators. However, certain operators can not be handled in this way, as their operation depends on the information generated at the run time of simulation. To handle these operators, we need to provide operator library functions. An occurrence of such an operator in Sim-nML description generates a operator library call in the generated simulator. Currently, we need to provide operator library functions for only $< . . >$ operator.

Chapter 5

Interfacing Simulator with GDB

A functional simulator is used to verify correctness of a program for the target processor. However, it provides no insight about the program or the processor state. A debugger is a tool to examine these states at the granularity of instruction level. GNU Debugger (GDB) is a generic debugging environment provided as open source. We have interfaced our simulator with GDB to utilize its debugging functionality.

5.1 Overview of GDB

The block diagram of structure GDB is shown in figure 5.1. GDB is a portable debugger and supports a large number of target architectures. It can be viewed as structured in two parts: front-end and back-end. The back-end is completely target dependent and hence the porting of GDB to a new target requires reconfiguration of back-end for that target. It provides functionality like execution control, stack frame analysis and physical target manipulation. The GDB front-end provides all the functionality which are target independent such as user interface and symbol handling. It can be accessed by different targets using the same interface.

The modular structure of GDB significantly reduces the efforts required to port it to a new target as the bulk of GDB's functionality is in its front-end which is common for all the targets. GDB uses Binary File Descriptor(BFD) [BFD] library for handling of various types of object file formats. BFD front-end and back-end

collaborate with GDB front-end and back-end respectively.

GDB provides a remote interface to connect with simulators and use them as target execution platforms. In case of the stand-alone remote simulators, GDB target support is not required. On the other hand, remote simulators can provide limited functionality and use GDB target support for the rest of the operations.

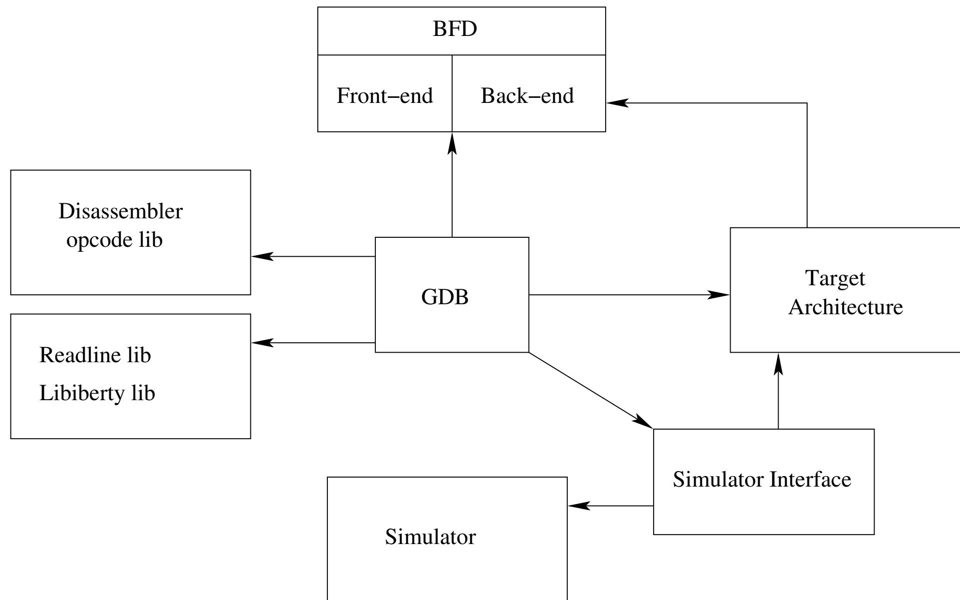


Figure 5.1: GDB Structure

5.2 Remote-Sim interface of GDB

It provides the following template functions to be implemented by the target simulator.

- **SIM_OPEN**: This is main entry point for simulator. It should create fully initialized simulator instance. In our implementation, this function generates the processor model and returns a success value back to GDB. From this point onward, GDB is connected to simulator.

- **SIM_CLOSE**: This function should destroy the simulator instance. All the resources allocated till that point are released and certain other book-keeping functions are performed. From this point onward GDB is no longer connected to simulator.
- **SIM_LOAD**: This function loads binary program into simulator memory. We decode input program and create its execution table. Finally, simulator is created as a dynamically linked library (DLL).
- **SIM_CREATE_INFERIOR**: This function prepares to run simulated program. It should initialize target processor registers and state variables and set command line arguments.
- **SIM_FETCH_REGISTERS**: This function should return the contents of requested registers.
- **SIM_STORE_REGISTERS**: This function should overwrite requested register's contents with the value provided in the function parameters.
- **SIM_READ_MEM**: This function should return the contents of requested memory location.
- **SIM_WRITE_MEM**: This function should overwrite requested memory location's contents with the value provided in the function parameters.
- **SIM_RESUME**: This function runs the program in simulator. If the *step* parameter in this function has non-zero value, simulator simulates only one instruction of the program. On the other hand, for zero value of step parameter, it runs simulator continuously till a breakpoint instruction is encountered or program ends.
- **SIM_STOP_REASON**: This function returns the reason why the program stopped. List of valid reasons is given below.
  - *EXITED*: The program has terminated. The exit status is returned through a signal.

  - *STOPPED*: The program has stopped. A signal value is returned to identify the particular reason such as a breakpoint instruction, an illegal instruction, a sim_stop request, completion of a single step, an internal error, an access to wrong memory location or a mis-aligned memory access.
  - *SIGNALED*: The program has been terminated by a signal.
  - *RUNNING*: The program is still running.
  - *POLLING*: The simulator is waiting for a new command.

- **SIM_STOP**: This function stops simulation process.

We have used the above mentioned remote-sim interface routines of GDB to create the functional simulator from the files generated by the simulator generator. Once the simulator instance is created, it is driven by these interface routines.

In GDB, implementation of breakpoint is handled by the target specific back-end. Most of the targets handle a breakpoint by writing a special instruction code at the breakpoint memory location. During the program execution, if control reaches the breakpoint memory location, a trap signal is generated by the special instruction. However, certain target architectures does not allow run-time modification of the program code. Such targets need to implement breakpoints in their own way.

Chapter 6

Results and Conclusions

In this chapter we will discuss performance results for functional simulator and conclude our thesis with insights into future work.

6.1 Experiments

We have tested our functional simulator generator by generating a functional simulator for **PowerPC 603** machine. We have implemented all **PowerPC** instructions other than those related to caches and processor pipeline.

6.1.1 Setting for Experiments

We have tested the simulator on the two different machines. The configuration of these machines is as follows.

- **Machine1**: Intel P-4 2.40 GHz, a little-endian 32 bit processor with 512 MB RAM running Linux-2.6.15-1
- **Machine2**: AMD Opteron 150 2.40 GHz, a little-endian 64 bit processor with 1 GB RAM running Linux-2.6.9-1

We have chosen a diverse set of applications to test our simulator. These applications are listed below.

- **IntMatMul.c**: Program to multiply two integer matrices. In this program, we initialize two 500x500 integer matrices randomly and then multiply them.
- **FloatMatMult.c**: Program to multiply two floating point matrices. In this program, we initialize two 500x500 floating point matrices randomly and then multiply them.
- **QuickSort.c**: Program to sort integer array using quick sort algorithm. In this program, we sort a randomly initialized integer array of size 5,000,000.
- **HeapSort.c**: Program to sort integer array using heap sort algorithm. In this program, we sort a randomly initialized integer array of size 5,000,000.
- **Fibonacci.c**: The Fibonacci numbers form a sequence in which each number is the sum of the two numbers before it. In our program, we calculate 40th Fibonacci number.
- **TowerHanoi.c**: In this problem, we have three pegs and N discs are stacked in order of size on peg 1, smallest at the top. The goal is to reposition the stack of disks from peg 1 to peg 3 by moving one disk at a time, and, never placing a larger disk on top of a smaller disk. In our program size of N is 27.
- **NQueen.c**: In this problem, we have to find all possible ways to put N chess queens on an NxN chessboard so that none of them is able to capture any other using the standard chess queen's moves. In our program, N is 15.

6.1.2 Results

We have compiled these programs using GCC cross compiler for **PowerPC** machine without any optimizations. We used statically linked binaries for testing as we do not provide support for handling dynamic link library calls in our simulator. Number of instructions simulated for each program is shown in table 6.1. Performance results for **Machine1** and **Machine2** are shown in table 6.2 and table 6.3 respectively.

Program name	**Number of instructions executed dynamically**
IntMatMul.c	13308993481
FloatMatMult.c	13314493435
QuickSort.c	7888705002
HeapSort.c	6391378391
Fibonacci.c	8941339490
TowerHanoi.c	6710896691
NQueen.c	3224021024

Table 6.1: Number of instructions simulated for each test program

Program name	**Time** (in seconds)	**Instructions per second**
IntMatMul.c	790	16846827
FloatMatMult.c	798	16684828
QuickSort.c	735	10732931
HeapSort.c	673	9496847
Fibonacci.c	870	10277401
TowerHanoi.c	716	9372760
NQueen.c	317	10170413

Table 6.2: Performance results on **Machine1**

Program name	**Time** (in seconds)	**Instructions per second**
IntMatMul.c	520	25594218
FloatMatMult.c	545	24430263
QuickSort.c	458	17224246
HeapSort.c	402	15898951
Fibonacci.c	560	15966677
TowerHanoi.c	440	15252037
NQueen.c	191	16879691

Table 6.3: Performance results on **Machine2**

Program name	**Ratio**
IntMatMul.c	0.645805
FloatMatMult.c	0.646439
QuickSort.c	2.020546
HeapSort.c	2.382557
Fibonacci.c	2.666666
TowerHanoi.c	3.119999
NQueen.c	2.369668

Table 6.4: Operator library calls to number of simulated instructions ratio

6.1.3 Analysis of Results

We are getting simulation speed in the range of 9-17 MIPS (million instructions per second) for the **Machine1** and 16-26 MIPS for **Machine2**. It is clear that the **Machine2** performs much better in comparison to the **Machine1**. The reason behind this huge performance gap is superior configuration of **Machine2**. As the relative performance of target applications is more or less same on both machines, application level analysis of one machine holds true for other also. We will analyze relative performance of the test applications on **Machine1**.

Among the seven chosen test applications, for five benchmarks we are getting simulation speed in the range of 9.3-10.7 MIPS. Given the diversity of application programs, this variation is expected. However, for first two applications, simulation speed is in the range of 16.5-17 MIPS. Between these two group of applications, there is a significant simulation speed gap of around 6-7 MIPS. This difference can be understood by examining the operator library call patterns of the applications. Table 6.4 shows the ratio of the number of operator library calls made by an application to the number of instructions simulated for the application for all the test applications. For first two applications the above mentioned ratio is less than 1, while for others it is in the range of 2-3.2. This explains the better performance results for the first two applications, as they are making relatively fewer operator library calls which happens to be quite expensive. These results suggest that the simulator performance can be improved significantly by optimizing the operator library functions. Similar effect can be achieved by writing the processor description containing fewer Sim-nML constructs which generate operator library calls.

6.2 Conclusions

In this thesis, we have described retargetable processor description language Sim-nML and its application in automatic generation of processor modeling software tool-sets for embedded systems. We have enhanced Sim-nML's resource usage model so that it can handle the processor timing model more effectively. We have also designed a C++ class hierarchy as an intermediate structure between Sim-nML pro-

cessor description and tool generators for easy and efficient processing of processor description. We have also built a traversal library for tool independent traversal of the class hierarchy.

We have also built a functional simulator generator based on Sim-nML processor descriptions. We have tested the simulator generated by our simulator generator for **PowerPC 603** architecture. Our simulator takes the **PowerPC** description and an ELF program binary as input and generates the simulator as a C language program. We are using compiled simulation technique for simulator generation and our target binaries are statically linked. We have also interfaced our simulator with GDB using its remote-sim interface for providing a generic debugging environment.

6.3 Future Work

We propose the following future extensions to our work.

- Although, Sim-nML can be used to describe data types of arbitrary bit-length, our implementation does not support data types having bit-length more than that of supported by the host language primitive data types. It can be extended to provide support for arbitrary bit-length data types.

- In the process of simulator generation, we are not doing any optimizations. Compiler optimization techniques used to generate optimized code can be utilized in the above process. Similarly, the simulator operator library imposes significant overhead on the performance of the simulator. It can be replaced with a more optimized operator library.

- The Sim-nML *class hierarchy* together with the traversal library provides a unique platform for development of retargetable tools. Processor modeling tools such as performance simulators, compiler back-end and disassemblers can be developed using this platform.

- Currently, our debugging environment only supports the processor models already supported by GDB. This debugger can me made generic enough to

support Sim-nML generated processor models. It requires porting of GDB as well as of BFD library to Sim-nML generated processor models.

- Currently, we are using an ELF reader library for reading the object file. Sim-nML development environment can be interfaced with BFD library for easy and efficient reading as well as writing of object file formats.

Bibliography

[A.R99] Rajiv A.R. Retargetable Profiling Tools and Their Application in Cache Simulation and Code Instrumentation. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, December 1999.

[BFD] BFD Library. http://www.gnu.org/software/binutils/manual/bfd-2.9.1/bfd.html.

[Bha01] Soubhik Bhattacharya. Generation of GCC Backend from Sim-nML Processor Description. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, July 2001.

[Cha99] Y. Subhash Chandra. Retargetable Functional Simulator. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, June 1999.

[Fin] Allan Finch. ELFIO. http://elfio.sourceforge.net/.

[Fre93] Markus Freericks. The nml machine description formalism. July 1993.

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

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

[HHD97] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 299–302, New York, NY, USA, 1997. ACM Press.

[Jai99] Nihal Chand Jain. Disassembler using High Level Processor Models. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, Jan 1999.

[LEL99] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of Interpretive and Compiled Instruction Set Simulators. *asp-dac*, 00:339, 1999.

[mde97] *The MDES User Manual*, 1997. http://www.trimaran.org.

[PHM00] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable Compiled Simulation of Embedded Processors Using a Machine Description language. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):815–834, 2000.

[PS02] Pierre G. Paulin and Miguel Santana. FlexWare: A Retargetable, Embedded-Software Development Environment. *IEEE Design and Test of Computers*, 19(4):59–69, 2002.

[Raj98] V. Rajesh. A Generic Approach to Performance Modeling and Its Application to Simulator Generator. Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, August 1998.

[RMD03] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 758–763, New York, NY, USA, 2003. ACM Press.

[SHL01] Eric C. Schnarr, Mark D. Hill, and James R. Larus. Facile: A Language and Compiler for High-performance Processor Simulators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming

language design and implementation, pages 321–331, New York, NY, USA, 2001. ACM Press.

[Sis98] Chuck Siska. A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools. In *ISSS*, pages 31–36, 1998.

[sys] SystemC. http://www.systemc.org.

[ten] Tensilica. http://www.tensilica.com.

[TIS95] TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification, May 1995. Version 1.2.

[Tri] Trimaran. http://www.trimaran.org.

[xil] Xilinx Vertex-5. http://www.xilinx.com/.

Appendix A

Grammar for Sim-nML language

```
MachineSpec           :
                      | MachineSpec  LetDef
                      | MachineSpec  TypeSpec
                      | MachineSpec  MemorySpec
                      | MachineSpec  RegisterSpec
                      | MachineSpec  VarSpec
                      | MachineSpec  ModeRuleSpec
                      | MachineSpec  OpRuleSpec
                      | MachineSpec  ResourceSpec

LetDef                : LET ID '=' LetExpr

LetExpr               : ConstNumExpr
                      | STRING_CONST
                      | IF ConstNumExpr THEN LetExpr OptionalElseLetExpr ENDIF
                      | SWITCH '(' ConstNumExpr ')' '{' CaseLetExprList '}'

OptionalElseLetExpr   :
                      | ELSE  LetExpr
```

```
CaseLetExprList         : CaseLetExprList1
                        | CaseLetExprList1 DEFAULT ':' LetExpr

CaseLetExprList1        : CaseLetExprStat
                        | CaseLetExprList1 CaseLetExprStat

CaseLetExprStat         : CASE ConstNumExpr ':' LetExpr

ResourceSpec            : RESOURCE ResourceList

ResourceList            : ID
                        | ID '[' ConstNumExpr ']'
                        | ResourceList ',' ID
                        | ResourceList ',' ID '[' ConstNumExpr ']'

TypeSpec                : TYPE ID '=' TypeExpr

TypeExpr                : BOOL
                        | INT  '(' ConstNumExpr ')'
                        | CARD '(' ConstNumExpr ')'
                        | FIX  '(' ConstNumExpr ',' ConstNumExpr ')'
                        | FLOAT  '(' ConstNumExpr ',' ConstNumExpr')'
                        | '[' ConstNumExpr DOUBLE_DOT ConstNumExpr ']'
                        | ENUM '(' IdentifierList ')'

IdentifierList          : ID
                        | ID '=' CARD_CONST
                        | IdentifierList  ','  ID
                        | IdentifierList  ','  ID '=' CARD_CONST

MemorySpec              : MEM ID '[' SizeType ']' OptionalMemVarAttr
```

```
RegisterSpec            : REG ID '[' SizeType ']' OptionalRegAttr

VarSpec                 : VAR ID '[' SizeType ']' OptionalMemVarAttr

SizeType                : TypeExpr
                        | ConstNumExpr
                        | ConstNumExpr ',' TypeExpr

OptionalMemVarAttr      :
                        | ALIAS '=' MemLocation

OptionalRegAttr         :
                        | PortsDef
                        | InitialDef
                        | PortsDef InitialDef
                        | InitialDef PortsDef

PortsDef                : PORTS '=' CARD_CONST ',' CARD_CONST

InitialDef              : INITIALA '=' ConstNumExpr

MemLocation             : ID Opt_Bit_Optr
                        | ID '[' NumExpr ']' Opt_Bit_Optr

ModeRuleSpec            : MODE ID ModeSpecPart

ModeSpecPart            : AndRule OptionalModeExpr  AttrDefList
                        | OrRule

OptionalModeExpr        :
```

```
                        | '=' Expr

OpRuleSpec              : 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 ':' TypeExpr
                        | ID ':' ID

AttrDefList             :
                        | AttrDefList   AttrDef

AttrDef                 : ID '='  AttrDefPart
                        | SYNTAX '=' ID '.' SYNTAX
                        | SYNTAX '=' AttrExpr
                        | IMAGE '=' ID '.' IMAGE
                        | IMAGE '=' AttrExpr
                        | ACTION '=' ID '.' ACTION
                        | ACTION '=' '{' Sequence '}'
```

```
                         | USES '=' UsesDef

AttrDefPart              : Expr
                         | '{' Sequence '}'

AttrExpr                 : STRING_CONST
                         | FORMAT  '(' STRING_CONST  ','  FormatIdlist ')'

FormatIdlist             : FormatId
                         | FormatIdlist  ',' FormatId

FormatId                 : ID
                         | ID '.' IMAGE OptBitSelect
                         | ID '.' SYNTAX
                         | DOLLAR '+' ConstNumExpr
                         | DOLLAR '-' ConstNumExpr
                         | ConstNumExpr '-' DOLLAR
                         | ID BinOp ConstNumExpr
                         | ConstNumExpr BinOp ID
                         | '+' ID
                         | '-' ID
                         | '~' ID

OptBitSelect             :
                         | BIT_LEFT CARD_CONST DOUBLE_DOT CARD_CONST BIT_RIGHT

Sequence                 :
                         | StatementList ';'

StatementList            : Statement
                         | StatementList ';' Statement
```

```
Statement               : ID
                        | ID '.' ACTION
                        | ID '.' ID
                        | Location  '='  Expr
                        | ConditionalStatement
                        | STRING_CONST '(' ArgList ')'
                        | ERROR '(' STRING_CONST ')'

ArgList                 :
                        | Expr
                        | ArgList ',' Expr

Opt_Bit_Optr            :
                        | Bit_Optr

Location                : LocationVal
                        | Location DOUBLE_COLON LocationVal

LocationVal             : ID Opt_Bit_Optr
                        | ID '[' Expr ']' Opt_Bit_Optr

ConditionalStatement    : IF  NumExpr  THEN  Sequence OptionalElse  ENDIF
                        | IF '(' STRING_CONST '(' ArgList ')' ')'
                          THEN Sequence OptionalElse ENDIF
                        | SWITCH '(' NumExpr ')' '{' CaseList '}'
                        | SWITCH '(' STRING_CONST '(' ArgList ')' ')'
                                 '{' CaseList '}'

OptionalElse            :
                        | ELSE  Sequence
```

```
CaseList                : CaseList1
                        | CaseList1 DEFAULT ':' Sequence

CaseList1               : CaseStat
                        | CaseList1 CaseStat

CaseStat                : CASE Expr ':' Sequence

Expr                    : NumExpr
                        | NonNumExpr

NonNumExpr              : StringExpr
                        | SyntaxImageExpr
                        | DOLLAR
                        | IF NumExpr THEN Expr OptionalElseExpr ENDIF
                        | SWITCH '(' NumExpr ')' '{' CaseExprList '}'
                        | STRING_CONST '(' ArgList ')'
                        | '(' NonNumExpr ')'

NumExpr                 : ConstNumExpr
                        | VarNumExpr

ConstNumExpr            : ConstExprVal
                        | ConstNumExpr BinOp ConstExprVal

ConstExprVal            : CARD_CONST
                        | FIXED_CONST
                        | BINARY_CONST
                        | HEX_CONST
                        | '!' ConstNumExpr
```

```
                         | '~' ConstNumExpr
                         | '+' ConstNumExpr %prec '~'
                         | '-' ConstNumExpr %prec '~'
                         | '(' ConstNumExpr ')'

VarNumExpr               : VarExprVal
                         | VarNumExpr BinOp VarExprVal
                         | VarNumExpr BinOp ConstExprVal
                         | ConstNumExpr BinOp VarExprVal

VarExprVal               : MemLocation
                         | COERCE '(' SizeType ',' Expr')'
                         | '!' VarNumExpr
                         | '~' VarNumExpr
                         | '+' VarNumExpr %prec '~'
                         | '-' VarNumExpr %prec '~'
                         | '(' VarNumExpr ')'

Bit_Optr                 : BIT_LEFT Bit_Expr DOUBLE_DOT Bit_Expr BIT_RIGHT

SyntaxImageExpr          : ID '.' SYNTAX
                         | ID '.' IMAGE
                         | ID '.' ID
                         | STRING_CONST

StringExpr               : STRING_CONST '<' STRING_CONST
                         | STRING_CONST '>' STRING_CONST
                         | STRING_CONST EQ STRING_CONST

BinOp                    : '+'
                         | '-'
```

```
                        | '*'
                        | '/'
                        | '%'
                        | DOUBLE_STAR
                        | LEFT_SHIFT
                        | RIGHT_SHIFT
                        | ROTATE_LEFT
                        | ROTATE_RIGHT
                        | '<'
                        | '>'
                        | LEQ
                        | GEQ
                        | EQ
                        | NEQ
                        | '&'
                        | '^'
                        | '|'
                        | AND
                        | OR

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            : CaseExprList1
                        | CaseExprList1 DEFAULT ':' Expr

CaseExprList1           : CaseExprStat
                        | CaseExprList1 CaseExprStat

CaseExprStat            : CASE Expr ':' Expr

OptionalElseExpr        :
                        | ELSE  Expr

UsesDef                 : NULLUSAGE
                        | Clause
                        | UsesDef ',' Clause

Clause                  : UsesSpec
                        | CondExpr '(' Clause OpAndOr UsesSpec ')' ActionTimeSpec

OpAndOr                 : '&'
                        | '|'

UsesSpec                : CondExpr ResourceUsageSpec ActionTimeSpec

CondExpr                :
                        | CondExpr '{' Expr '}'

ActionTimeSpec          :
                        | Action
```

```
                        | Time
                        | Action Time
                        | Time Action


Time                    : '#' '{' Expr '}'

Action                  : ':' ID
                        | ':' ACTION

ResourceUsageSpec       : ResourceUsage
                        | IF Expr THEN Clause OptionalElseUses ENDIF

OptionalElseUses        :
                        | ELSE Clause


ResourceUsage           : ID '.' USES
                        | ID OptReqMark
                        | RegOpr

OptReqMark              :
                        | '<'
                        | '>'
                        | '[' ']'
                        | '[' Expr ']'

RegOpr:                 ID '[' Expr ']' '.' ITR
                        | ID '[' Expr ']' '.' ITW
                        | ID '[' Expr ']' '.' FORWARD
                        | ID '[' Expr ']' '.' RDONE
```

```
| ID '[' Expr ']' '.' WDONE
| ID '.' ITR
| ID '.' ITW
| ID '.' FORWARD
| ID '.' RDONE
| ID '.' WDONE
```

Appendix B

Class Hierarchy Structure

In this appendix we are going to discuss the Class Hierarchy structure. All the base classes in hierarchy hold a pointer to a context class for error reporting.

B.1 Instruction Set Class

Sim-nML supports description of processors with multiple instruction sets i.e. **ARM** and **ARM Thumb** instruction sets of **ARM** architecture. This class holds all the instruction-set descriptions for the target processor.

```
class IntructionSet{
   list<IR*>   ir_list;
   Context*    ctx;
}
```

- *ir_list*: This field holds a list of pointers to all instruction-sets in the Sim-nML description.

B.2 IR Class

This is the main class to represent an instruction-set description.

```
class IR{
   list<Declaration*>   ir_declare_list;
   list<Rule*>          ir_rule_list;
   Context*             ctx;
}
```

- *ir_declare_list*: This field holds a list of pointers to all the declarations in the description.
- *ir_rule_list*: This field holds a list of pointers to all the rules in the description.

B.3 Declaration Class

This is a base class to describe all the declarations in the Sim-nML description.

```
class Declaration{
protected:
   string         decl_name;
   int            decl_id;
   ir_decl_type   decl_type;
   Context*       ctx;
}
```

- *decl_name*: This field holds name of the declaration variable.
- *decl_id*: This field holds a unique id for each declaration in the description.
- *decl_type*: This field holds the type of the declaration. It can take the following values: CONST, RESOURCE, STORAGE.

B.4 Constant Class

This is a derived class from the *Declaration class* to represent constants in the description.

```
class Constant :public Declaration{
protected:
   ir_const_type const_type;
}
```

- *const_type*: This field holds type of the constant. It can take one of the following values: INT_CONST, FLOT_CONST, STR_CONST.

B.5 Integer Constant Class

This is a derived class from the *Constant class* to represent integer constants in the description.

```
class IntConst :public Constant{
   int const_num_val;
}
```

- *const_num_val*: This field holds the value of the integer constant.

B.6 String Constant Class

This is a derived class from the *Constant class* to represent string constants in the description.

```
class StrConst :public Constant{
   string const_str_val;
}
```

- *const_str_val*: This field holds the value of the string constant.

B.7 Real Constant Class

This is a derived class from the *Constant class* to represent real constants in the description.

```
class FloatConst :public Constant{
   string const_num_val;
}
```

- *const_num_val*: This field holds the value of the real constant.

B.8 Resource Class

This is a derived class from the *Declaration class* to represent resources e.g. fetch unit in the description.

```
class Resource :public Declaration{
   int res_no_units;
}
```

- *res_no_units*: This field holds the total number of instances of the resource.

B.9 Storage Class

This is a derived class from the *Declaration class* to represent storage model of the target processor.

```
class Storage :public Declaration{
protected:
   int               stor_size;
   Type*             stor_data_type;
   ir_storage_type   stor_type;
}
```

- *stor_size*: This field holds total number of storage elements.
- *stor_data_type*: This field holds data type of the storage element. It is specified through a pointer to the *Type class*.
- *stor_type*: This field holds type of the storage element. It can take one of the following values: IR_REG, IR_MEM, IR_VAR.

B.10 Register Class

This is a derived class from the *Storage class* to represent registers in the target processor.

```
class Register :public Storage{
   int   reg_read_ports;
   int   reg_write_ports;
   int   reg_init_val;
}
```

- *reg_read_ports*: This field holds the number of read ports for a register file.
- *reg_write_ports*: This field holds the number of write ports for a register file.
- *reg_init_val*: This field holds the initial value of the registers.

B.11 Memory Class

This is a derived class from the *Storage class* to represent memory in the target processor.

```
class Memory :public Storage{
   AttrDef* mem_attr_def;
}
```

- *mem_attr_def*: This field holds a pointer to the *AttrDef class*, which will hold various attributes, e.g. alias, of the memory.

B.12 Variable Class

This is a derived class from the *Storage class* to represent local variables in Sim-nML description.

```
class Variable :public Storage{
   AttrDef* var_attr_def;
}
```

- *var_attr_def*: This field holds a pointer to the *AttrDef class*, which will hold various attributes of the local variable.

B.13 Rule Class

This is a base class for all the rules in Sim-nML description.

```
class Rule{
protected:
   string          rule_name;
   int             rule_id;
   ir_rule_type    rule_type;
   list<RetList*>  ret_list;
   Context*        ctx;
}
```

- *rule_name*: This field holds the name of the rule.
- *rule_id*: This field holds a unique id for the rule.
- *rule_type*: This field holds the type of the rule. It can take one of the following values: OR_RULE, AND_RULE.
- *ret_list*: This field holds a list of pointers to the *RetList class*. It is used by various tool generators to store tool-centric information in the rule.

B.14 Or Rule Class

This is a derived class from the *Rule class* to represent *or rules* in the description.

```
class OrRule :public Rule{
   int            or_no_child;
   list<Rule*>    or_and_list;
}
```

- *or_no_child*: This field holds the total number of children for the *or rule*.
- *or_and_list*: This field holds a list of pointers to child rules.

B.15 And Rule Class

This is a derived class from the *Rule class* to represent *and rules* in the description.

```
class AndRule :public Rule{
   int            and_total_params;
   int            and_total_attrs;
   list<Param*>   and_param_list;
   list<IrAttr*>  and_attr_list;
}
```

- *and_total_params*: This field holds the total number of parameters in the *and rule*.
- *and_total_attrs*: This field holds the total number of attributes in the *and rule*.
- *and_param_list*: This field holds the list of pointers to the parameters in the *and rule*.
- *and_attr_list*: This field holds the list of pointers to the attributes in the *and rule*.

B.16 IrAttr Class

This is a base class for all the attributes of an *and rule*.

```
class IrAttr{
protected:
   string           attr_name;
   int              attr_id;
   ir_attr_type     attr_type;
   Context*         ctx;
}
```

- *attr_name*: This field holds the name of the attribute.
- *attr_id*: This field holds a unique id for the attribute.
- *ir_attr_type*: This field holds the type of the attribute. It can take one of the following values: SYNTAX, IMAGE, ACTION, USES, OTHER.

B.17 ImageSyntax Class

This is a derived class from the *IrAttr class* to represent image and syntax attributes in an *and rule*.

```
class ImageSyntax :public IrAttr{
    int                 imgsyn_no_subpart;
    list<AttrSubPart*>  imgsyn_subpart_list;
}
```

- *imgsyn_no_subparts*: This field holds the total number of subparts in an image or syntax pattern.
- *imgsyn_subpart_list*: This field holds the definitions of all the subparts in an image or syntax pattern. It is specified through a list of pointers to the *AttrSubPart class*.

B.18 AttrSubPart Class

This is a base class for defining various subparts of an image or syntax attribute.

```
class AttrSubPart{
protected:
   ir_subpart_type  subpart_type;
   int              subpart_width;
   Context*         ctx;
}
```

- *subpart_type*: This field holds the type of AttrSubPart. It can take one of the following values: STRING, PARAMETER, ATTR_DEF.
- *subpart_width*: This field holds the width of the subpart. It is more relevant for an image attribute as width will indicate the number of bits in an image pattern.

B.19 StrSubPart Class

This is a derived class from the *AttrSubPart Class* to represent strings in an image or syntax attribute.

```
class StrSubPart :public AttrSubPart{
   string subpart_str;
}
```

- *subpart_str*: This field holds value of the string subpart of an image or syntax attribute.

B.20 ParamSubPart Class

This is a derived class from the *AttrSubPart Class* to represent parameters in an image or syntax attribute.

```
class ParamSubPart :public AttrSubPart{
   char     specifier_type;
   Param*   subpart_param;
}
```

- *specifier_type*: This field holds the specifier type of the parameter. It can take one of the following values: %s , %d, %b, %o, %x.
- *subpart_param*: This field holds the definition of a parameter. It is specified through a pointer to the *Param class*.

B.21 ExprSubPart Class

This is a derived class from the *AttrSubPart class* to represent expressions in an image or syntax attribute.

```
class ExprSubPart :public AttrSubPart{
   char     specifier_type;
   AttrDef* subpart_attr_def;
}
```

- *specifier_type*: This field holds the type of parameter specifier of an expression type parameter. It can take one of the following values: %s, %d, %b, %o, %x.
- *subpart_attr_def*: This field holds the specification of an expression type parameter. It is specified through a pointer to the *AttrDef class*.

B.22 Action Class

This is a derived class from the *IrAtrr class* to represent the action attribute of an *and rule*.

```
class Action :public IrAttr{
   int      action_no_def;
```

```
   AttrDef* action_attr_def;
}
```

- *action_no_def*: This field holds the total number of attribute definitions in an action attribute.
- *action_attr_def*: This field holds a pointer to the *AttrDef class* and defines the semantics of the action attribute.

B.23 Uses Class

This is a derived class from the *IrAttr class* to represent the uses attribute of an *and rule*.

```
class Uses :public IrAttr{
   list<Clause*> uses_clause_list;
}
```

- *uses_clause_list*: This field holds a list of pointers to the *Clause class*. The uses attribute is composed of list of clauses.

B.24 Clause Class

This is a base class to represent a clause in a uses attribute. The uses attribute is composed of list of pointers to these clauses.

```
class Clause{
protected:
   ir_clause_type  clause_type;
   AttrDef*        clause_cond;
   AttrDef*        clause_action;
   AttrDef*        clause_time;
```

```
    AttrDef*        clause_if_expr;
    Clause*         clause1;
    Clause*         clause2;
    Context*        ctx;
}
```

- *clause_type*: This field holds the type of the clause. It can take one of the following values: CLAUSE_SIMPLE, CLAUSE_AND, CLAUSE_OR, CLAUSE_IF.
- *clause_cond*: This field holds a pointer to the *AttrDef class* and represents the conditional expression of the clause.
- *clause_action*: This field holds a pointer to the *AttrDef class* and represents optional book-keeping actions of the clause.
- *clause_time*: This field holds a pointer to the *AttrDef class* and represents optional timing expression of the clause.
- *clause1*: This field holds a pointer to the Clause class and represents first operand of an *and* or *or* type of clause.
- *clause2*: This field holds a pointer to the Clause class and represents second operand of an *and* or *or* type of clause.

B.25 ResUnitSpec Class

This is a derived class from the *Clause class* to specify a resource unit in the uses attribute.

```
class ResUnitSpec :public Clause{
    ir_res_unit_type    res_unit_type;
}
```

- *res_unit_type*: This field holds the type of a resource unit. It can take one of the following values: RESOURCE_USES, PARAM_USES.

B.26 ResUses Class

This is a derived class from the *ResUnitSpec* to directly specify the resource used.

```
class ResUses :public ResUnitSpec {
   ir_res_uses_type   res_uses_type;
   ir_res_uses_opr    res_uses_opr;
   Declaration*       res_uses_dec;
   AttrDef*           res_uses_index;
}
```

- *res_uses_type*: This field holds the type of resource, whether resource instance or register operation. It can take one of the following values: RES_INST, REG_ITR, REG_ITW, REG_READ, REG_FORWARD, REG_RDONE, REG_WDONE.
- *res_uses_opr*: This field holds the operation to be applied on resource or register. It can be one of the following: RES_ACQ, RES_REL, RES_ALL, RES_REG_INDEX, RES_REG_SIMPLE.
- *res_uses_dec*: This field holds the pointer to the declaration of resource or register.
- *res_uses_index*: This field holds the value of index in case res_uses_opr is equal to RES_REG_INDEX.

B.27 UsesAttr Class

This is a derived class from the *ResUnitSpec* to specify the rule referenced for the resource.

```
class UsesAttr :public ResUnitSpec {
   Rule*   rule;
}
```

- *rule*: This field is a pointer to the rule referenced for resource.

B.28 Param Class

This is a class to specify parameters in an *and rule*.

```
class Param{
protected:
   int          param_no;
   string       param_name;
   Type*        param_type;
   Context*     ctx;
}
```

- *param_no*: This field holds a unique parameter number.
- *param_name*: This field holds the name of the parameter.
- *param_type*: This field holds a pointer to the *Type class* and specifies the type of the parameter.

B.29 Type Class

This is a base class to specify both basic types and rule types.

```
class Type{
protected:
   ir_arg_type   arg_type;
   Context*      ctx;
}
```

- *arg_type*: This field holds the type specification. It can take the following two values: DATA_TYPE, RULE_TYPE.

B.30 BasicType Class

This is a derived class from the *Type class* to represent basic data types.

```
class BasicType :public Type{
   ir_var_data_type   var_data_type;
   int                var_val1;
   int                var_val2;
}
```

- *var_data_type*: This field holds the data type of a variable or parameter. All supported data types can be found in section 2.2.3.
- *var_val1*: This field holds the bit width of integer part of the data type.
- *var_val2*: This field holds the bit width of fractional part of the data type.

B.31 RuleType Class

This is a derived class from the *Type class* to represent the parameters of type rule.

```
class RuleType :public Type{
   Rule*   param_rule;
}
```

- *param_rule*: This field holds a pointer to the *Rule class*.

B.32 AttrDef Class

This is a base class to specify expressions in Sim-nML description. Expressions are stored in a tree like structure and this class is used to create the tree nodes recursively.

```
class AttrDef{
protected:
   ir_attr_def_type  attr_def_type;
   Context*          ctx;
}
```

- *attr_def_type*: This field holds the type of an expression.

B.33 DeclDef Class

This is a derived class from the *AttrDef class* to represent declared variables used in an expression.

```
class DeclDef :public AttrDef{
   Declaration*   decl_def;
}
```

- *decl_def*: This field holds a pointer to the *Declaration class* to define declared variables in an expression.

B.34 ParamUse Class

This is a derived class from the *AttrDef class* to represent parameters in an expression.

```
class ParamUse :public AttrDef{
   Param*   param;
}
```

- *param*: This field holds a pointer to the *Param class* to define parameters in an expression.

B.35 TypeUse Class

This is a derived class from the *AttrDef class* to represent data types in an expression.

```
class TypeUse :public AttrDef{
   Type* type;
}
```

- *param*: This field holds a pointer to the *Type class* to define data types in an expression.

B.36 AttrNameDef Class

This is a derived class from the *AttrDef class* to represent attribute names in an expression.

```
class AttrNameDef :public AttrDef{
   string attr_name;
}
```

- *param*: This field holds name of an attribute in an expression.

B.37 LiteralDef Class

This is a derived class from the *AttrDef class* to represent literal values in an expression.

```
class LiteralDef :public AttrDef{
protected:
   ir_literal_type literal_type;
}
```

- *literal_type*: This field holds the type of literal in an expression. It can take one of the following values: INT_LITERAL, REAL_LITERAL, STR_LITERAL.

B.38 IntLiteral Class

This is a derived class from the *Literal class* to describe integer literals in an expression.

```
class IntLiteral :public LiteralDef{
protected:
   int     int_val;
}
```

- *int_val*: This field holds the value of the integer literal in an expression.

B.39 RealLiteral Class

This is a derived class from the *Literal class* to describe real literals in an expression.

```
class RealLiteral :public LiteralDef{
   float   real_val;
}
```

- *real_val*: This field holds the value of the real literal in an expression.

B.40 StrLiteral Class

This is a derived class from the *Literal class* to describe string literals in an expression.

```
class StrLiteral :public LiteralDef{
   string  str_val;
}
```

- *str_val*: This field holds the value of the string literal in an expression.

B.41 OprDef Class

This is a derived class from the *AttrDef class* to describe sub expressions within an expression.

```
class OprDef :public AttrDef{
   ir_attr_def_opr   opr_type;
   int               opr_arity;
   list<AttrDef*>    oprnd_list;
}
```

- *opr_type*: This field holds the type of operator in a sub expression. The complete list of operators supported by Sim-nML is given in section 2.4.2.
- *opr_arity*: This field holds the value of the number of operands in a sub expression.
- *oprnd_list*: This field holds a list of pointers to the *AttrDef class* to describe the operands in a sub expression.

B.42 Traversal Library Interface

To handle tool specific data, we are providing the following two classes.

- **RetList Class**

  ```
  class RetList{

  }
  ```

  This class is not a part of class hierarchy. It is used by tool generators to store tool-centric information in it. *RetList* is a base class with no fields. Tool generators need to derive subclasses to hold their specific information.

- **ToolSpec Class**

```
class ToolSpec{
public:
    virtual list<RetList*> tool_processor(list<Type*> and_param_list,
                                          list<IrAttr*> attr_list);
}
```

> This class is used by tool generators to perform tool-specific operations. *ToolSpec* is a base class with one virtual function. This virtual function takes a parameter list and an attribute list as parameters. It returns a list of pointers to the *RetList class*. Tool generators need to derive subclasses to hold their specific information and implement the *tool_processor* function.

Traversal library interface is implemented using a *virtual function* and list of pointers to the *RetList class* in the *Rule class*. This function is implemented by both the *And class* and the *Or class*. Signature of the traversal function is given below.

```
virtual void traverse(ToolSpec* tool_spec);
```

It takes a pointer to the *ToolSpec class* as parameter to carry tool specific information.