Disassembly and Parsing Support for Retargetable Tools Using Sim-nML

by

Nitin Kumar Dahra

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

May 2007

Disassembly and Parsing Support for Retargetable Tools Using Sim-nML

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

by

Nitin Kumar Dahra

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY,KANPUR

May 2007

Certificate

This is to certify that the work contained in the thesis entitled *"Disassembly and Parsing Support for Retargetable Tools Using Sim-nML"*, by *Nitin Kumar Dahra*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May, 2007

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

Abstract

Due to the increasing design complexity, time to market constraints and other cost criteria, embedded systems designers use automated processor modeling tools for rapid design and analysis of various system design trade-offs. Given a processor description, these tools facilitate automated generation of processor specific tools.

Sim-nML [RM99] is a retargetable architecture description language used to develop processor modeling tools. In our work, we have developed a parser which takes a Sim-nML processor description as input and generates an easy and efficient to use intermediate representation in a hierarchical tree form suitable for input to tool generators. We have made a retargetable disassembler based on this intermediate representation. This disassembler has been interfaced with GDB to provide a generic debugging environment.

A traversal library [Vis06] is used to facilitate tool independent traversal of the intermediate representation hierarchy. The intermediate representation together with the traversal library provides a unique platform for development of various retargetable processor modeling tools.

Acknowledgements

I am indebted to my supervisor, Professor Rajat Moona, for his guidance throughout this project. His enthusisam and professionalism were always a source of inspiration for me. His clarity of thought and deep insight into technical matters paved the way for the completion of this thesis. I will be very happy if I am able to imbibe some of his qualities in my life.

I would like to thank Surendra for all the valuable discussions we had and helping me out when I was stuck and overall for motivating me with his hard work. I would also like to thank Amit, Bhupesh and Hemant for having some stimulating interactions and asking some tough questions which increased my own insight into this topic.

I am also thankful to all my MTech friends especially Aditya, Amit, Arjun, Ciju, Gajendra, Kaushik, Mausoom, Palak, Prashant, Rahul, Satyam, Surendra and Varun for sharing their knowledge with me and making my stay here truly enjoyable and memorable. I will always cherish the moments I spent with them.

Finally, I would like to express my gratitude towards my parents and my brother for their constant support and encouragement throughout the tough times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An embedded system is a combination of hardware and software. Generally, software is used for features and flexibility, while hardware is used for performance. The traditional design methodology is to first make the hardware and then write software for them. Due to the increasing design complexity and other constraints, this approach is no longer feasible. Some of the major issues with this approach are the following.

- The hardware design errors become increasingly costly to correct as the design progresses. If the errors are detected at an earlier stage, it will result in substantial cost savings.
- It requires the separation of functionality to implemented in hardware and software at the beginning of the design itself. This leaves no scope for further revisions. In addition to this inflexibility, this may lead to sub-optimal designs.
- The design of these systems can be subjected to many additional constraints, including performance, cost, time-to-market, power, space and reliability requirements.

These problems forced the system designers to follow new design approaches. In these, the hardware and the software are designed concurrently. This allows designers to look into various hardware software design trade-offs and chose the

ones most suited to the requirements as soon as possible in the development of the system.

1.1 Hardware Software Co-design

Hardware software co-design can be defined as cooperative design of hardware and software components. It allows the movement of fuctionality between hardware and software at various stages during the system development life-cycle. The chief goals of hardware software co-design are to shorten the time-to-market by reducing the design effort and the cost of final system by optimizing the hardware software partitioning. The hardware software co-design flow is shown in figure 1.1.

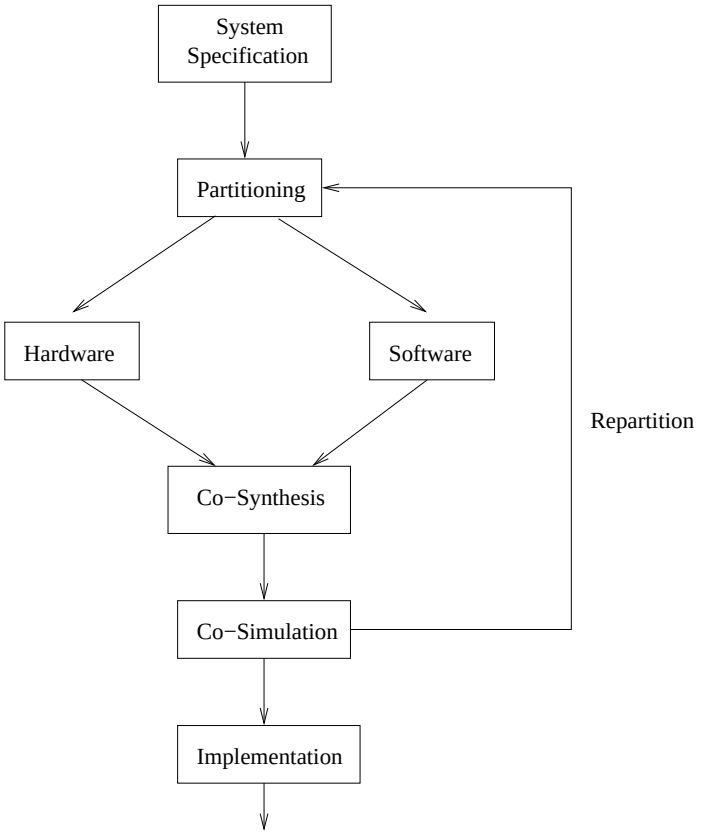

Figure 1.1: Hardware software co-design flow diagram

First step in co-design is to get a refined system specificatin from the given requirements. After this, the specification is analyzed based on various cost metrics like performance, space and power requirements. At this point, the system functionality is divided into smaller set of modules. The partitioning step identifies which functionlity module is best suited for implementation in hardware or software based on the various cost criteria. After this step we get a division between the hardware and software components from where the design and development on these go in parallel.

Hardware is designed using hardware description languages (HDLs) such as Verilog, VHDL etc. HDLs are capable of describing the hardware at the lowest level and the associated EDA tools facilitate the compilation, simulation and synthesis of these hardware designs. The software is designed using some high level language such as C, C++ etc. This process typically involves tools such as processor simulator, assembler, disassembler, debugger, compiler back-end, profiler etc. These toolsets allow the development of software for the target environment. This phase involving hardware synthesis and software compilation is known as Co-Synthesis.

The next step in co-design process is Co-Simulation. In Co-Simulation, these hardware and software components are brought together and executed in real time. To speed up the simulation time of overall system, generally emulation is used. Emulation systems map the synthesized hardware onto real programmable hardware such as FPGA (Field Programmable Gate Arrays) providing us a close prototype of the final system. Co-Simulation allows the verification of original design and we also come to know if the imposed constraints are met. If the design meets all the performance constraints and the final cost is acceptable, the Co-Simulation process stops and the hardware design is converted to real hardware and then the software design is integrated with it to make the final system. However, if the constraints are not met, a repartitioning step is performed to optimize the design and this process is repeated till we satisfy the various cost criteria.

1.2 Retargetable Processor Modelling Tools

Different embedded systems have their own unique requirements. We can either make an entirely new processor core as per application requirements (Application Specific Processors) or modify an already existing one. In the latter case we have several alternatives to chose from such as Tensilica [xten] and Xilinx Virtex-5 [xil]. But a new design requires a new set of software tools which takes a lot of time and effort to develop. This problem can be solved by automation of software toolset generation process.

Automated software toolset generators take the specification of processor model as input and generate the desired tool based on that processor model. This greatly reduces the effort required to make new tools. To accomplish this we only have to write the processor specification. However, the language used to write the processor specification must be powerful enough to model a variety of processor features. It must also be easy to understand and program in.

1.3 Overview of Retargetable Languages and Frameworks

HDLs such as Verilog [ver] and VHDL [vhd] are widely used to model processors. However, processor models written in these languages contain highly detailed and low level hardware implementation details. These details are not useful from a simulation and software verification point of view. Also, it is not possible to obtain instruction set details like assembler syntax from such low level descriptions.

SystemC [sys] is an example of high level system description language. SystemC is a set of library routines and macros implemented in C++. It contains good feature set to perform behavioral modeling of the system. However, SystemC is geared more towards system level modeling rather than processor modeling.

Architecture description languages (ADLs) are high level languages designed specifically to model processor architectures. They provide sufficient abstraction

and are suitable for making retargetable tools. Some of the ADLs and frameworks are described below.

nML [Fre93] language, developed at TU Berlin, is based on attribute grammar and describes processors at the level of instruction set. nML lacks constructs to support the resource usage and timing mechanisms. Sim-nML [RM99] language, developed at IIT Kanpur, is an extension of nML which adds these features.

ISDL (Instruction Set Description Language) [HHD97], developed at MIT, is a language mainly targeted towards VLIW architectures. It is similar to nML and describes the instruction set of the processor. ISDL has been used to generate compiler and simulator.

MDES [mde97] machine specification system is a part of Trimaran tool set [Tri]. It supports the instruction set level and structural level details including details like resource usage and latency. It also provides novel features such as predication, control and data speculation and compiler controlled management of the memory hierarchy. However, MDES allows only allows a limited retargetability.

MIMOLA (Machine Independent Microprogramming Language) [LEL99], developed at University of Dortmund, Germany, describes a processor at a lower level and is close to hardware description languages like Verilog and VHDL. The output of MIMOLA based design process is a register-transfer level description of the computer. MIMOLA has been used to make instruction set simulators and retargetable code generation systems.

LISA processor specification language [PHM00] developed at Aachen University of Tecnology, Germany, supports the description of different aspects of processor architectures like behavior, instruction set, and syntax. It also facilitates pipeline description and timing control. It has been used to to generate tools like assembler, disassembler, linker, compiler, debugger front-end etc. The language RADL [Sis98] is derived from LISA and extended to support multiple pipelines.

EXPRESSION [HGG+99] ADL, developed at UC Irvine, supports both behavioral (instruction set) and timing model details of the processor. It uses a high level of abstraction to specify pipeline path from which reservation tables are automatically generated. It has been used to make retargetable compilers and simulators.

1.4 Overview of this Work

In this work, we have made a parser to generate an *intermediate representation (IR)* from a Sim-nML processor description. We have also developed a generic disassembler using this *IR*. It can be used to disassemble a program written for a new processor design. We have also interfaced this disassembler with GNU 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 and hides all hardware level 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.

As a processor description language, Sim-nML has various features to make description writing easy to read and write. However, due to such features it becomes harder for tool developers to directly use the processor description in tool generators. We have made a parser to convert the Sim-nML processor description into a C++ class hierarchy as an intermediate structure between the 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 information.

1.5 Previous Work with Sim-nML

Sim-nML was designed as an extension to nML. nML was not capable of handling the execution behavior of processor pipeline properly. Sim-nML addressed this issue by adding a resource usage model. A very simple and primitive instruction set simulator [Raj98] was designed at that time. 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 in the form of fixed sized tables was also introduced.

- **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:** This tool [Bha01] read a Sim-nML description in intermediate form and generated 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, we introduce Sim-nML and describe in detail the intermediate representation and mechanism to translate Sim-nML to its *IR*. In chapter 3, we discuss the design and implementation of our disassembler. In chapter 4, we look into the interface between GDB and the disassembler. In chapter 5, we conclude with results and future work. In Appendix A we provide the complete details of Sim-nML language. In appendices B and C, we provide the Sim-nML grammar and list of operators used in *IR* respectively. Finally, in appendix D we give the class hierarchy structure for the *IR*.

Chapter 2

Intermediate Representation

2.1 Introduction to Sim-nML

Sim-nML is a language for describing an arbitary processor architecture at the level of instruction set. It is flexible, easy to use and based on attribute grammar. The processor models in Sim-nML contain details of registers/memory, instructions, addresing modes and resource usage model. Instruction set in Sim-nML are described in a hierarchial, tree like structure, with sharing of common information among related instructions. These details are specified by the constructs *rules* and *declarations*. Sim-nML provides two types of *rules* - *mode-rules* and *op-rules*. *mode-rules* are used to descibe the addressing modes while *op-rules* describe the instructions. *declarations* provide the specifications of registers, memory and local variables. Sim-nML can be used to generate retargetable tools according to the given processor specification. Complete details about Sim-nML is given in the Appendix A.

2.2 Intermediate Representation

A processor specification in Sim-nML language is a human readable text file. Several constructs are provided in Sim-nML to aid the readability and clarity of the processor description. The tools based on Sim-nML need to parse the description

to remove redundant information, perform variable substitution, etc. It is not convenient for every tool to retrieve the desired information from description itself. In addition to a lot of duplicate work, inconsistency and errors could develop due to independent interpretations. Thus, it is desirable to have an interface between processor description and the input to the tool generator. We provide this interface in form of an *intermediate representation (IR)* obtained by parsing the processor description. This IR would be a collection of all the information present in the description in an efficient and easy to use format.

2.2.1 Tabular Structure vs Class Hierarchy

In the earlier work on Sim-nML [A.R99], IR was designed as a collection of tables which would hold the relevant information extracted from the processor description. These tables were designed to be simple for easy handling of the IR. Such simplifications called for the table entries to be fixed sized. This demanded addition of several other tables to prevent the table entry from being variable sized. These extra tables required indirect indexing to access the required information. This method of accessing data enhanced the complexity of tool development and a lot of effort was spent in retrieving data from the IR rather than in developing tools themselves. Additionally, this table structure had no natural resemblance to the structure of the processor specification.

In this work, we have made a parser to parse the Sim-nML processor descriptions and store the information in IR in the form of a *class hierarchy*. The detailed description of this class hierarchy as defined in [Vis06] is reproduced in Appendix D. This new IR has the following advantages

- It represents the processor description in a natural manner and avoids the indirect addressing as in the tabular structure.
- It represents the data in an object oriented manner, thereby hiding the internal details of IR implementation from the tool generators.
- It provides a generic structure to store the tool specific data in the IR, inde-

pendent of the tool generators.

The earlier IR was designed to be stored in a file. The tool generators would then read this file to get the required processor information. However, the parsing of description is fast enough and the requirement of eliminating parsing overheads by keeping the IR in a file is not necessary.

To maintain compatibility with the old tabular IR format, we have routines for conversion between these two formats.

The parser is available as a library which can be linked by the tools to achieve parsing and get the IR for the Sim-nML description.

2.3 Generating the IR

2.3.1 Simplification by Substitution

Sim-nML language allows definition of constants by using *let specification* (eg: **let REGS = 5**). During the parsing of the Sim-nML specification file, whereever a constant is referenced, its value is substituted in the IR. For example, value of the constant **REGS** i.e. 5, is substituted whererever **REGS** is used in the example figure 2.3. The definition of the symbolic names after substitution become redundant and may be removed from the IR. However, certain constants definitions might be used by the tools, for example, constants like **byte_order** may be used by tools to define the byte ordering of a processor. Therefore, a choice was made to carry out the substitution s and to additionally retain all such definitions in the IR.

Sim-nML also defines new data type definitions using basic data types and previously defined user data types. Since all user defined data types can be built using only the basic data types, all variables are redefined with only basic data types in the IR. Thus all user defined data type declarations can be eliminated from the IR. For example in figure 2.3, **byte** is used to refer to data type **card(8)**. All occurances of **byte** are replaced by **card(8)** during the IR generation. In the generated IR, no definition of **byte** will be available.

The third type of substitution that is used is in the case of enumerated data types. While the previous two substitutions are done lexically, in a way similar to the macro expansions in C, substitution of enumerated data types is done at the parser level. It also includes checks for proper compatibility. This kind of substitution is explained by an example given below.

```
type e1 = enum(low = 0, high)
type e2 = enum(zero = 0, one, two, three, four)
reg R[1, e1]


R = high;
R = two;
```

Figure 2.1: Example to show use of enumerated data types

Here, in the first statement, high will be replaced by 1. Second statement will result in parser error since R is bound to enumerated type e1 and can only take values `low` and `high`.

2.3.2 Structure of tihe Intermediate Representation

The intermediate representation is capable of storing information about constants, declarations, *or-rules*, *and-rules* and information about attributes such as *syntax*, *image*, *action* etc. These informations are represented using a hierarchy of classes. The common attributes of a particular entity are present in a base class while the differentiating attributes are stored in the derived classes. This achieves an orthogonal storage structure and avoids carrying duplicate and redundant information. The topmost class in this hierarchy is known as *IR* which encapsulates the complete intermediate representation of the processor description.

IR is specified by the tuple <`declare_list, rule_list, ctx`> (Here we show only the variables of the class. The complete details of all the classes including the data types is given in the Appendix D.)

The variables `declare_list` and `rule_list` represent the list of all declarations And-rules in the IR. In other words, IR is just a collection of list of declarations and list of rules. The complete detail of a declaration or a rule can be accessed from the relevant elements of these list. All the base classes of IR contain an object of *Context* class. This holds the information about the source code location for the corresponding entity and is used for the purpose of error reporting by tools.

Context is specified by the following tuple - <`line_start, line_end, name`>.

The variables `line_start` and `line_end` store the starting and ending line numbers in the file corresponding to that entity. The `name` is a string which stores the context string. For the constant definitions like let and type which are substituted by their values, this context string stores the name of that particular definition. For everything else, `name` will just be the name of the description file.

The *IR* class is constructed during the parsing of the Sim-nML description. This process works as shown in figure 2.2 below.

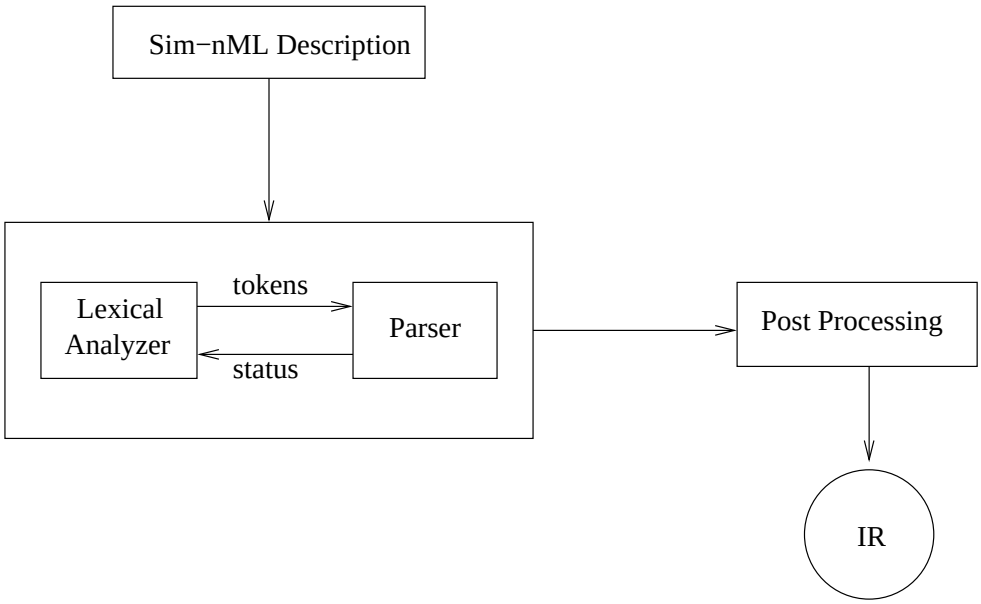

Figure 2.2: IR Generation

The input Sim-nML description is converted into a stream of tokens by lexical analyzer which are consumed by the parser. As each individual rule and declaration

gets reduced by the parser, they are added to the *IR* object. This object, however, is incomplete at this stage. This is due to the use of forward references in the processor description. For example, the sub-rules of an *or-rule* may be defined later than that rule itself. All such information is saved during the parsing phase and at the end all the stubs are filled in appropriately in the post-processing stage. After post-processing we get the complete *IR* structure which can be used by the tools.

We will now see how the data is stored in *IR*. We first describe the building blocks of this *IR* hierarchy and then show how declarations and rules are added using the example processor description given below.

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

[1]  let REGS = 5
[2]  let MSIZE = 2**8

[3]  type byte = card(8)
[4]  type index = card ( REGS )
[5]  type long = card ( 32 )
[6]  type slong = int ( 32 )

[7]  reg GPR [ 2 ** REGS , slong ]

[8]  mem cq_ptr [ 1 , byte ]
[9]  mem wait_id [ 1 , byte ]
[10] mem CQ [ 6 , long ]
[11] mem int_id [ 1 , byte ]
[12] mem M [MSIZE, byte]
[13] mem TMP_SHWORD [ 1 , int ( 16 ) ]
```

```
[14]  mem TMP_SHWORD_A0 [ 1 , int ( 8 ) ]
[15]       alias = TMP_SHWORD [ 0 ]
[16]  mem TMP_SHWORD_A1 [ 1 , int ( 8 ) ]
[17]       alias = TMP_SHWORD [ 8 ]

[18]  resource int_unit

 \\ ****Addressing Modes****\\
[19]  mode REG_IND ( r : index ) = GPR [ r ]
[20]     syntax = format ( "%d", r )
[21]     image  = format ( "%5b", r )

[22]  mode IMM16 ( n : int ( 16 ) ) = n
[23]     syntax = format ( ``%d'', n )
[24]     image  = format ( ``%16b'', n )

 \\ ****Instruction Set****\\

[25]  op instruction ( x : instr_action )
[26]     uses   = x.uses
[27]     syntax = x.syntax
[28]     image  = x.image
[29]     action = {
[30]                  x.action;
[31]              }

[32]  op instr_action = arith_instr
```

```
[33]                      | branch_instr

[34]  op arith_instr = add
[35]                     | sub
[36]                     | mul
[37]                     | div

[38]  op add ( rd : index, ra : REG_IND, rb : REG_IND )
[39]    uses   = int_unit #{1}
[40]    syntax = format ("add %d,%s,%s", rd, ra.syntax, rb.syntax)
[41]    image  = format ("011111%5b%s%s01000010100",
                         rd, ra.image, rb.image<4..2>)
[42]    action = {
[43]                GPR [ rd ] = ra + rb;
[44]             }
[45]    preact = {
[46]                int_id = CQ [ wait_id ];
[47]                wait_id = wait_id + 1;
[48]             }
     //other instructions
```

Figure 2.3: Example processor description

2.3.3 Types

Types in the IR are represented by the class *Type* <arg_type>

arg_type holds the kind of *Type* stored. It can be either DATA_TYPE or RULE_TYPE. The further details are defined in the Type subclasses *BasicType* (used

when `arg_type` is DATA_TYPE) and *RuleType* (used when `arg_type` is RULE_TYPE).

BasicType class is used to represent basic data types like bool, int, card etc. It consists of the tuple <`data_type, val1, val2`>

`data_type` defines the type of this *BasicType* and `val1` and `val2` are bit widths of integer and fractional part of the data type. For example, for card(6), `data_type` will be a constant IR_MT_CARD, `val1` will be 6 and `val2` will be unused. Similarly, for fix(5,3), `data_type` will be IR_MT_FIX, `val1` will be 5, and `val2` will be 3.

The class *RuleType* <`param_rule`> is used to represent the rule type parameters.

Here `param_rule` will point to the appropriate rule. In the example description above, in rule add, parameter ra is of rule type. In this case `param_rule` will be pointer to the rule REG_IND.

2.3.4 Parameters

The *and-rule* parameters are stored in the objects of class *Param* <`no, name, type`>

The variable `no` is the unique number assigned to this particular parameter. The variable `name` holds the name of this param. Finally, the variable `type` holds the type of that particular parameter as defined in section 2.3.3. For example, in the description given in figure 2.3 in line 38, for parameter rd, `name` will be "rd" and `type` will be representation of card(5), which refers to user type index defined in line 4 of the example.

2.3.5 Expressions

Expressions form a core part of Sim-nML language. It is important to understand how they are stored in the IR. Expressions are constructed in a tree like structure and the class *AttrDef* is used to represent them.

AttrDef class is represented by <`def_type`>.

This class encapsulates the whole variety of expressions possible. The variable `def_type` denotes the type of expression that the object will have and the corresponding subclass contains the actual value. The values it can take are the following.

- **Literal:** Literals occuring in Sim-nML description are represented by the class *LiteralDef* <`literal_type`>

  The kinds of literals possible in Sim-nML are integer, real and string, as in 5, 3.63 and "hello" respectively. The variable `literal_type` denotes the type of literal and the actual value is stored in the objects of respective derived classes, *IntLiteral*<`int_val`>, *RealLiteral*<`real_val`> and *StrLiteral*<`str_val`>.

- **Declaration:** When a declaration is used in an expression, it is stored as an object of class *DeclDef*<`decl_def`>

  `decl_def` is the pointer to the actual declaration being referred. For example, in the line 47 of figure 2.3, wait_id is used in the expression. Here `decl_def` will point the memory declaration of wait_id.

- **Parameter:** The usage of parameters in an expression is stored in an object of class *ParamUse*<`param_use`>

  `param_use` is the pointer to the parameter used. For example, in line 43 of figure 2.3, an expression ra + rb is used that uses two parameters ra and rb. For the usage of ra and rb, *ParamUse* object is used where the variable `param_use` points to the respective parameters ra and rb.

- **Attribute Name:** An attribute name may occur in an expression, for example in the expression $x.image < 4..8 >$ an image attribute is used. It is stored as an object of class *AttrNameDef* <`attr_name`>

  For the above expression, `attr_name` will be "image".

- **Sub Expressions:** Most expressions will be formed by combining the simple expressions given above with some operator. For example, the expression "d + 5" is formed by combining the declared variable, d with the interger literal 5 using operator '+'. 'd' and '5' will be called the operands of the operator '+'. Such expressions are stored using an object of class *OprDef*<`type, arity, list`>.

Here `type` denotes the type of operator and `arity` is the arity of that operator. Finally, `list` contains the actual list of a number of operands that is defined by the `arity`. A complex expression is stored in hierarchial manner where the operands of an operator can themselves be operators having their own operands. Figure 2.4 gives some examples of expressions as stored in the IR. All the expressions possible are stored using *OprDef* class only. To make it possible, IR adds some operators not in Sim-nML. For example, to store an if-then-else expression, a ternary IF operator is used with the operands being if-condition, if-action and else-action respectively. Appendix C gives the complete list of operators used in IR and their arity.

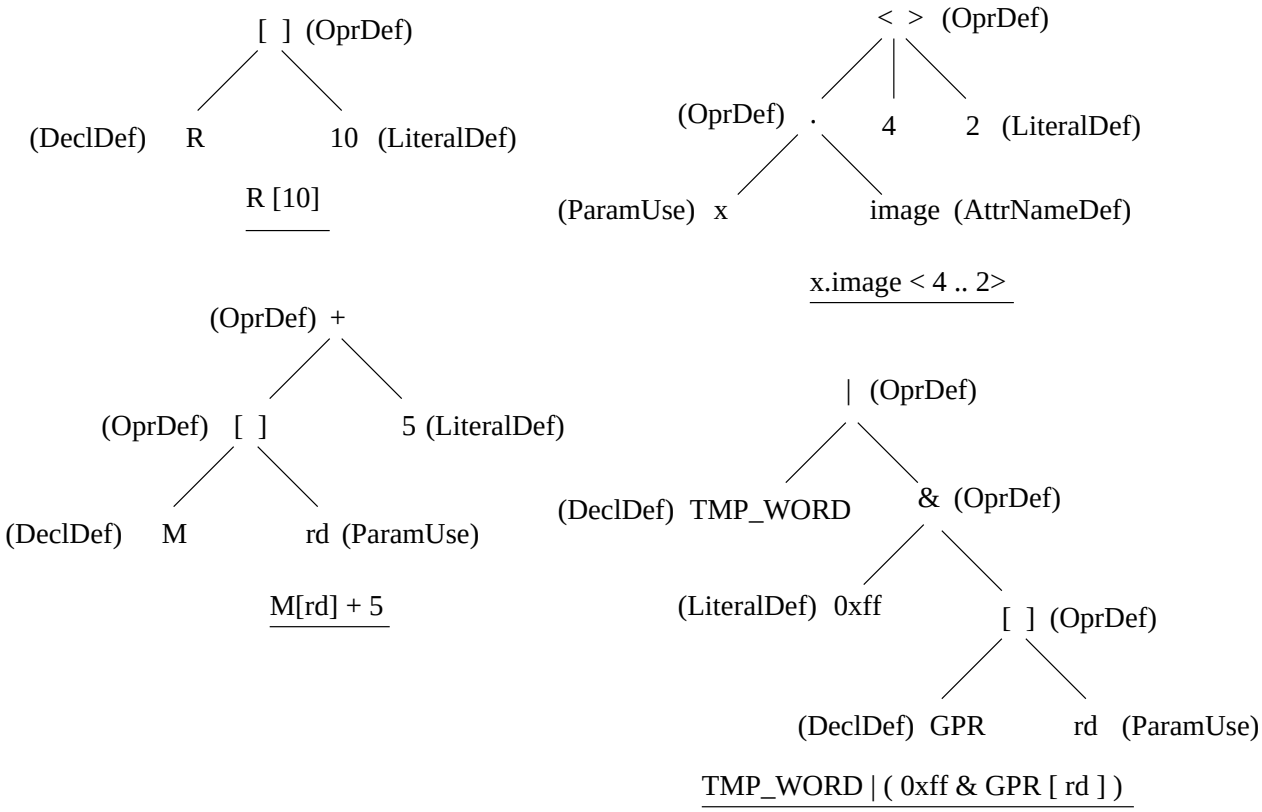

Figure 2.4: Example showing expressions

2.3.6 Adding Declarations

Declarations are used to store the processor specific infrastructure available to us as part of the description. These include processor registers, memory, resources, flags and variables local to the description. The top level class which represents all declarations is *Declaration*<`name, id, type`>.

Here, `name` is the name of the declaration, `id` is the unique number assigned to this declaration and `type` contains the type of this declaration. It can be CONST, STORAGE or RESOURCE for the three type of declarations possible in Sim-nML.

These types of declarations are stored as objects of their respective subclasses as given below.

- *Storage*<`size, data_type, stor_type`>. The variable `size` represents the size of that storage entity. The variable `data_type` is an object of class *Type* representing the type of the storage. Finally, `stor_type` is the type of the storage. We have three types of Storage entities possible and they are represented by the following subclasses.

  - *Register*<`read_ports, write_ports, init_val`> The variables `read_ports` and `write_ports` represent the number of read and write ports available for that register. The variable `init_val` provides the initial value for this register. If these values are not provided in the description, they are filled in according to the conventions specified in the Appendix A.

  - *Memory*<`attr_def`> A memory can be an alias to some other memory location as described in the Appendix A. The variable `attr_def` stores an expression providing the definition of the alias. For example, in the case of memory declaration in line 12 in figure 2.3, the value of variable `attr_def` will be NULL representing absence of an alias. However, in the declaration of TMP_SHWORD_A0, `attr_def` will contain an expression corresponding to TMP_SHWORD [0].

  - *Variable*<`attr_def`> The variable `attr_def` here provides the same purpose as in the case of memory. We do not currrently allow variables to

be aliased to some other variable. `attr_def` here is present primarily for backward compatibility.

- *Constant*<`const_type`>. The constant declarations specified in the description using *let* keyword are represented by this class. The variable `const_type` here denotes the type of Constant as described below. The actual value is stored in the objects of corresponding subclasses as given below.
  - *IntConstant*<`int_val`> for integer constant. The variable `int_val` stores the integer value.
  - *FlotConstant*<`flot_val`> for floating point constant. The value is stored in the variable `flot_val`.
  - *StrConstant*<`str_val`> for string constants. The variable `str_val` stores the string value. For example, in case of declaration **let endianity = "big"**, the complete tree representation in IR will be as shown in figure 2.5

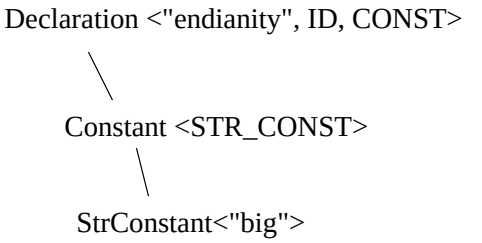

Figure 2.5: Example showing storage of a string constant in IR

- *Resource*<`no_units`> The variable `no_units` holds the number of units of this particular resource.

2.3.7 Adding Rules

Rules are the used to store the instruction set specific information present in the description. These rules are described in a hierarchial tree like structure as described

in the Appendix A. The instruction set contains two orthogonal components - mode-rules, which are used to define the addresing modes; and op-rules, which describe the instructions. From the point of view of IR there is no difference between mode and op rules as they are stored in the same data structures. Both these components are specified using production or-rules and and-rules. Or-rules are used to group together a related set of op-rules whereas an And-rule describes a particular instruction and contains its details. These rules are described in detail in the Appendix A.

The base class that represents rules is *Rule* <`name, id, type, ret_list`>

The variable `name` stores the name of the rule. The variable `id` stores the unique number assigned to this rule. The variable `type` denotes the type of the rule. It can be an OR_RULE or an AND_RULE. The variable `ret_list` is used as a placeholder for tool specific information and is actually independent of any particular tool.

OR Rules

An Or-rule is basically a list of all possible rules for that particular rule. Or rules are represented by the class *OrRule* <`no_child, or_and_list`>

The variable `no_child` stores the number of children of this or-rule. The variable `or_and_list` provides the list of possible rules this or-rule can expand to. For example, in line 34 in figure 2.3, add, sub, mul and div are the possible rules for the arith_instr. In this case the value of variable `no_child` will be 4 and the variable `or_and_list` will contain a list of Rule pointers to these individual rules. The base class variable `name` will be "arith_instr" and `type` will be OR_RULE.

And Rules

An And-rule has two components, parameters and attributes. For example, in line 38 in figure 2.3, the **add** instruction has three parameters - rd, ra and rb and five attributes - *uses*, *syntax*, *image*, *action* and *preact*.

The parameters are represnted by *Param* class as described earlier.

There are four predefined attributes for an And-rule - *image*, *syntax*, *action* and

uses. Additional attributes can be added to this list. The new attributes are all similar in syntax and sematics and use the same data structure for storage in IR as of *action* attribute. The *preact* attribute in **add** instruction is an example of such an attribute. Attributes are stored as objects of the class *IrAttr*<`name, id, type`>

The variables `name`, `id` and `type` represent the name, a unique number by which the attributes are referred to and type of the attribute, which can have the values as IMAGE, SYNTAX, ACTION and USES.

The detailed storage structure of these attributes is as follows.

- **Image and Syntax**: Image attribute specifies the binary pattern of the instruction. Syntax describes the assembler level version of the instruction. The image and syntax attributes have similar structure and are by the class *ImageSyntax* <`no_subpart, subpart_list`>.

  The variable `no_subpart` provides the number of subparts that the particular image or syntax has. The variable `subpart_list` contains the actual values stored as list of *AttrSubPart*<`type, width`>.

  Here, the variable `width` holds the number of bits that subpart takes as given in the format string. The variable `type` denotes the kind of subpart. The actual value of the subpart is stored in the corresponding subclass object according to the `type`. These are

  - *StrSubPart*<`str`> This is the simple string type subpart. The variable `str` contains the actual string value.
  - *ParamSubPart*<`specifier_type, param`> This type of subpart refers to a parameter of the current And-rule. The variable `specifier_type` denotes the kind of specifier for that subpart as given in the format string. The variable `param` stores the pointer to the corresponding parameter of the current And-rule.
  - *ExprSubPart*<`specifier_type, attr_def`> This is used when the subpart is an expression. The variable `specifier_type` is same as in *ParamSubPart*. The variable `attr_def` contains the pointer to the expression stored as *AttrDef*.

Consider the image attribute of add instruction given in line 41 of figure 2.3.

This image attribute has 5 subparts. First subpart is string type subpart with `str` equal to "011111" and `width` 6. Second subpart is a param type subpart specified by "%5b" and rd. Here `width` is 5, `specifier_type` is 'b' which stands for binary and the variable `param` stores a pointer to *Param* rd of this And-rule. Third attribute is specified by "%s" and the corresponding ra.image is an expression type subpart. In this case, `width` is unspecified and `specifier_type` is 's'. The variable `attr_def` points to the expression ra.image stored as an *AttrDef* object. Simlarly, the fouth and fifth subparts are (expression type subpart with specifier 's' and expression rb.image<4..2>) and (string type subpart with value "01000010100") respectively.

Image and Syntax can also be written as simple string or just a reference to the corresponding attribute of a parameter, for example syntax = "add" or syntax = x.syntax where x is a parameter of that rule. In this case, we form the `subpart_list` consisting of a single subpart which is of string type in former and expression type with specifier_type set as 's' in latter case.

- **Action**

  Action attribute describes the execution behaviour of an instruction. It is represented by the class *Action*<`no_def, attr_def`>

  The variable `no_def` holds the total number of attribute definitions in an action attribute. The variable `attr_def` holds the value of *action* attribute in form of tree of expressions. Action is defined in grammar as sequence of statements where each statement is delimited by a semicolon. For the **add** instruction given in figure 2.3 the trees for *action* and *preact* are shown in the figure 2.6.

- **Uses**

  The Uses attribute is used for describing resource usage pattern of a particular instruction. It is stored as a list of sequential clauses each separated by a comma. The complete details of the resource usage model are explained in the Appendix A.

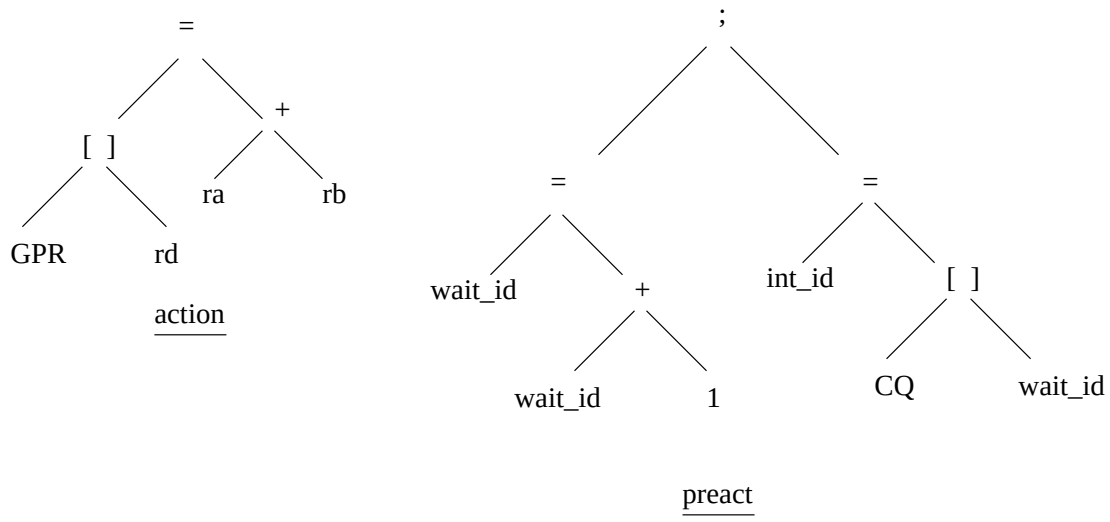

Figure 2.6: Expression tree for action and preact attributes

Uses attribute is represented by the class *Uses*<`clause_list`> where the variable `clause_list` stores the list of objects of class *Clause*.

The class *Clause* is described as the tuple <`type, cond, action, time, if_expr, clause1, clause2`>

Here, the variable `type` denotes the type of clause which can be a simple clause, an or-clause, an and-clause or an if-expression. `cond`, `action` and `time` are used to store the corresponding attribute of the clause. If the clause is of type if-expression, `if_expr` stores the expression for that. In case of an and-clause or an or-clause, `clause1` and `clause2` will have the corresponding sub-clauses which are *and*ed or *or*ed.

The simple clause is represented by the *Clause* subclass *ResUnitSpec*<`res_unit_type`>

The variable `res_unit_type` denotes the type of usage of clause which can be either Resource or a Rule. These are further represented by the following classes.

- *ResUses* <`type, opr, dec, index`> The variable `type` holds the type of resource, which can be either a resource instance or a register operation.

These register operations are - intent to read, intent to write, actual read operation, actual write operation and forwarding of register value. The variable `opr` holds the operation to be applied on this resource or register. It can take the following values - aquisition of resource, release of resource, using all resources, using a particular resource or register, single resource or register.

The variable `dec` holds the pointer to the declaration of resource or register used in this clause. The variable `index` holds the array index of this declaration.

– *UsesAttr* <`rule`>: *UsesAttr* is used in case the clause refers to the resource usage of another rule, for example `x.uses`. In this case the variable `rule` will contain pointer to the appropriate rule.

Now let us see how a complicated attribute of type uses will be stored with an example given below.

```
uses =  x.uses, ( { bu_busy == 0 } fetch_unit1 : preact1 #{1}
                | { bu_busy == 0 } fetch_unit2 : preact2 #{1}
                ), R[2].itr
```

Figure 2.7: Example of uses attribute

This uses attribute consists of 3 clauses. Its representation in IR is shown in figure 2.8

Mode Rules

In case of mode rules, we have an additional attribute called *val* attribute which are used for describing the addressing mode of an instruction. For example, in case of mode rule `REG_IND` in figure 2.3, `GPR [ r ]` will be the val attribute. The expression in *val* attribute is stored as an *AttrDef* variable. The *val* attribute itself is stored like any additional attribute defined ie like an *Action* attribute.

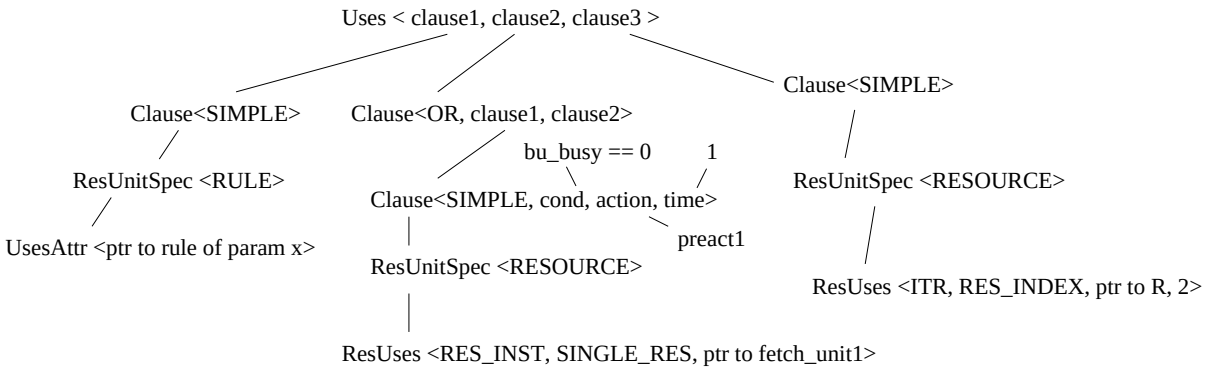

Figure 2.8: Uses attribute of figure 2.7 in IR

Chapter 3

Disassembler

3.1 Overview

A disassembler for a particular processor takes the machine code as an input and produces the assembly instructions for that processor. We have implemented a retargetable disassembler that takes the processor information from the intermediate representation of the its Sim-nML description. This disassembler is generic in nature meaning that it doesn't depend on the type of the processor and works according its processor descriptions. The disassembly works by traversing the IR hierarchy and matching the instructions with the input binary stream.

3.2 Traversing

The disassembler works in conjunction with the tool independent traversal library [Vis06]. This is co-recursive in the sense that traversal routine calls the disassembler functions which in turn call this traversal routine to process other rules. The two types of traversals possible are, (1) complete traversal of the tree in which all the nodes of the tree are visited, and, (2) guided traversal in which only a subset of nodes are visited as guided by the input data.

In this disassembler, we have used guided traversal, where the input byte stream

for machine code is used to guide the traversal down the tree. Complete traversal can be avoided here as we just need to find a single match for each machine instruction.

The disassembler needs to look at only the image and syntax attributes. For an instruction, these attributes will be distributed over the complete *and-or* IR tree. The disassembler traverses the instruction tree top-down in a depth-first fashion and uses backtracking while matching input with the image attribute. As the matches are made successfully, the corresponding syntax definitions are used to generate the assembly instruction.

To traverse the tree, disassembler needs to store some data at all the successful intermediate nodes to reach at correct node. This data is stored in the placeholder for tool specific information, represented by class *RetList*, in the *Rule* class as mentioned in the previous chapter. We store the following data in case of an *and*-rule.

- *Path*: This is the path by which we arrive at this particular rule. It is defined as the sequence of rule references required in reaching a node. It is used to identify if a particular rule was traversed earlier with the same path.
- *Bits_Matched*: This is the list of number of bits matched for each image subpart. It is used to change the input matching offset in case of backtracking.
- *Length*: This is just the sum of all the entries in the previous list.
- *Syntax*: This is the syntax of the rule constructed at the end of successful match of all image subparts.
- *Parameter_Syntax*: This is the list of syntax of each parameter reference in the image subparts. This is used to construct the final syntax string for this rule.

In case of an *or*-rule, we store *Path*, *Length* and *Syntax* as before. The lists *Bits_Matched* and *Parameter_Syntax* are empty in this case. In addition, we store a pointer to the *Next_Rule* which is just the subrule which resulted in successful traversal. This is needed because whenever we backtrack, we need an *or*-rule traversal to continue from the previous successful point. This *Next_Rule* pointer is not used and is just stored as NULL in case of an *and*-rule.

3.3 Disassembler Algorithm

The disassembler works as follows.

We iterate over the image subpart list of the rule being processed and try to match the subparts with the input binary stream. The processing of different types of subparts as follows is done as follows.

- *String*: String type subparts can be directly compared with the input.
- *Parameter*: The data type parameters are the parameters of that particular instruction. The parameter which can be, for example, a register number or an immediate value, is formed from the input equal to the length of that parameter. So in this case, the input stream subset of length equal to the number of bits of this parameter is saved.
- *Expression*: An expression can involve reference to other rules, for example in case of `x.image`. In this case we would need to traverse the rule referred by parameter `x`. This traversal is performed with the unique integer id of this parameter appened to path. In case of a successful match the syntax returned from this traversal is saved.

At each point, if the subpart is matched, the offset of tool in the input data stream is incremented by the number of bits of this subpart. But, if the match is unsuccessful, we would need to backtrack to a previous point of rule reference to try for other possibilities (Backtracking is explained in more detail in next section). If backtracking fails, the disassembler routine returns failure.

If the iteration gets completed, it means that the rule's *image* attribute has successfully matched the input binary stream. At this point we will process the *syntax* attribute and construct the rule's syntax using the information collected during the *image* processing stage. The complete data for this successful traversal is stored in the fields as mentioned above.

We will see the working of the disassembler with the example below.

```
op bran_cond_link_abs ( BO : card ( 5 ), BI : card ( 5 ), BD : SIMM )
```

```
        syntax = format ( ``bcla %d,%d,%s'', BO, BI, BD.syntax )
        image  = format ( ``010000%5b%5b%s11'', BO, BI, BD.image )
            .
            .
        // other attributes

    mode SIMM ( n : int ( 14 ) ) = n
        syntax = format ( ``%d'', n )
        image  = format ( ``%14b'', n )
```

Figure 3.1: Example to show the working of disassembler

Suppose the input binary stream is -

```
01000010 01001001 11000000 11100111 10101100 11010110 ...
```

In processing the instruction *bran_cond_link_abs*, we will iterate over the image subparts.

- The first subpart is the string type subpart with value "010000". It matches the binary stream.
- Second and third subparts are data type parameters, BO and BI respectively. These are considered as matched and their values are taken from input as 10010, 01001 and stored.
- Fourth subpart is a expression type subpart with parameter BD referring to rule SIMM. This will result in another traversal where the input value 11000000111001 is matched with image. Since SIMM has now matched completely, its syntax will be computed according to the *syntax* attribute as the decimal value 12345. This data is then available in the original rule and we store "12345" as syntax for parameter BD.

- Finally, fifth subpart string "11" matches the input.

At this point we can process the *syntax* attribute and compute the syntax as `bcla 18, 9, 12345`.

The advantage of this tree matching algorithm is that complete subtrees starting at some node can be discarded without descending further into them. For example, if we know that all arithmetic instructions in our processor description have image pattern starting with 1010 and the input stream doesn't match it we can completely discard this subtree having, potentially, several instructions.

3.3.1 Backtracking

It is possible that a image subpart that refers to another rule matches some subrules while the subsequent image subparts don't match the input as shown in example in figure 3.2. This means that the choice we made at some previous rule references was wrong or that perhaps we are in a wrong subtree altogether. To correct our decision we need to change the choices made in earlier rule reference. This is accomplished by backtracking in the tree.

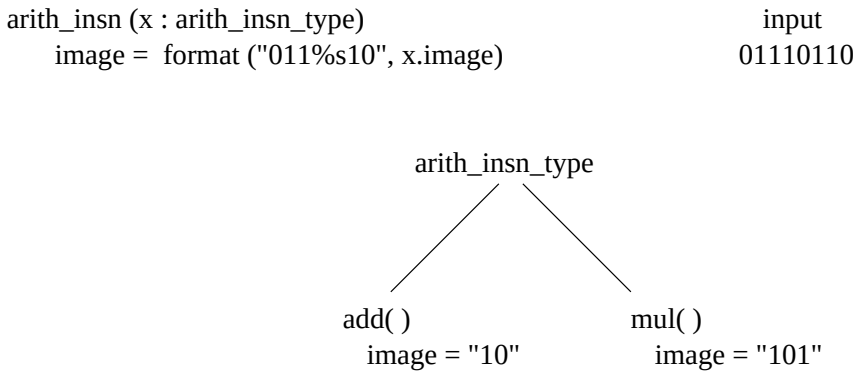

Figure 3.2: Example for showing Backtracking

For example in case of example of figure 3.2, while processing image for *arith_insn* we will succeed in matching the input with *add*. However, later part of image ie `10` does not match with input. This means that we selected wrong subrule for

arith_insn_type and we will need to backtrack and change our choice. We get the correct matching with *mul* instruction and processing of *arith_insn* succeeds. If, for example, the input stream is like ‘‘`0111XXXX...`’’, the rule *mul* would also not match it. In this case, the traversal of *arith_insn* will result in failure and we will have to backtrack to a node in IR tree that is above *arith_insn* and search in some other portion of the tree.

This differs from traditional backtracking in that we may revisit nodes that were successfully traversed earlier. For example, in figure 3.2, even though traversal of node *arith_insn_type* was successful and resulted in selection of *add* sub-rule, we had to visit this node again and chose the *mul* sub-rule due to failure ahead.

Backtracking essentially provides us with next possible choice to compare with the input. If we enumerate all the rules by flattenning out all the subrule references, we will get an order of all possible values of the image to match. Backtracking provides us with the next value in this enumeration to compare the input against. This is done without actually flattening all the rules and is made possible with the help of the data we store at all the successful rule matches.

Whenever we start to process a particular rule, we need to know if this rule was successfully traversed earlier with the same path. If this is indeed the case, it means that there was failure in matching somewhere ahead and we are backracking and have returned to this rule to explore other possibilities of match. We then restore all the information stored from the earlier successful traversal and then start backtracking in image subpart list from the end to get at next enumeration.

It is possible that the input data doesn't match with any instruction in the tree. This can be the case, for example, when a particular instruction is left out from the processor description or in case of incorrect input data being fed. In this case the traversal will result in failure and the disassembler outputs a `.byte` pseudo-op and continues matching from the next byte onwards.

Chapter 4

Interfacing Disassembler with GDB

A debugger is an interactive software containing fine-grained controls for execution of a program and examining and changing the processor states. GNU debugger (GDB) is a popular, free and open source debugger. We have interfaced our disassembler with GDB to utilize its debugging facilities.

4.1 Overview of GDB

The block diagram of GDB is shown in figure 4.1. Internally, GDB can be viewed as having 3 major subsystems: user interface, symbol handling and target system handling.

The user interface consists of various routines for display of information and other supporting code for features such as providing command line buffer etc.

The symbol side consists of object file readers, symbol table management, expression handling, language support, source display and other actibities that involve symbolic data. BFD library [BFD] which consists of facilities for reading various executable and object file formats (ELF, coff, a.out etc) is a core component of this subsystem.

The target side consists of execution control, stack frame analysis, disassembly opcode library etc. It mainly works with numerical data such as modifying program counter, manipulating breakpoint information etc.

This modular separation allows GDB to be ported to large number of target architectures.

GDB also provides support for interfacing with simulators through a simulator interface which is a very helpful feature for development of embedded systems.

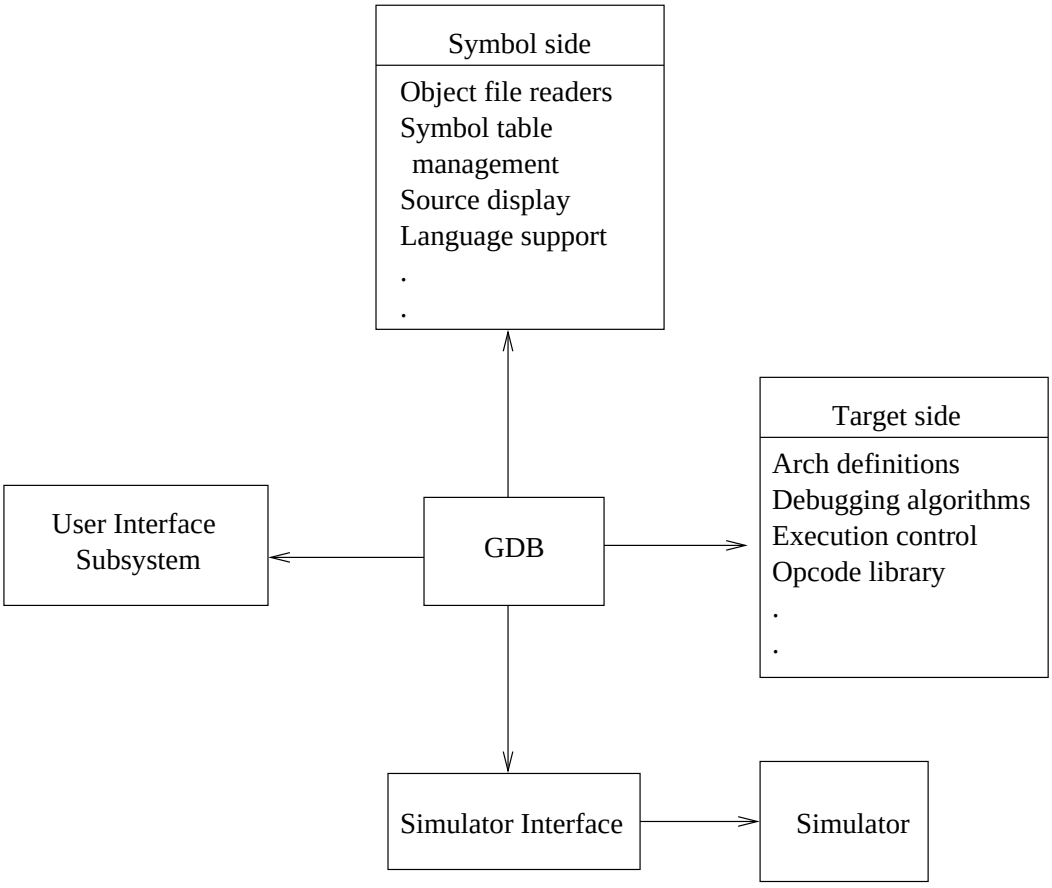

Figure 4.1: GDB structure

4.2 Interfacing with GDB

GDB provides a disassemble command which is used to examine the disassembly of a specified region of a program binary during interactive debugging. The disassembly information for all the supported target architectures is contained in the opcode library which is linked with GDB. This information is present in a hardcoded way in the opcode library.

We have changed this interface of GDB to its opcode library by plugging in our own disassembler in its place. This is shown in figure 4.2.

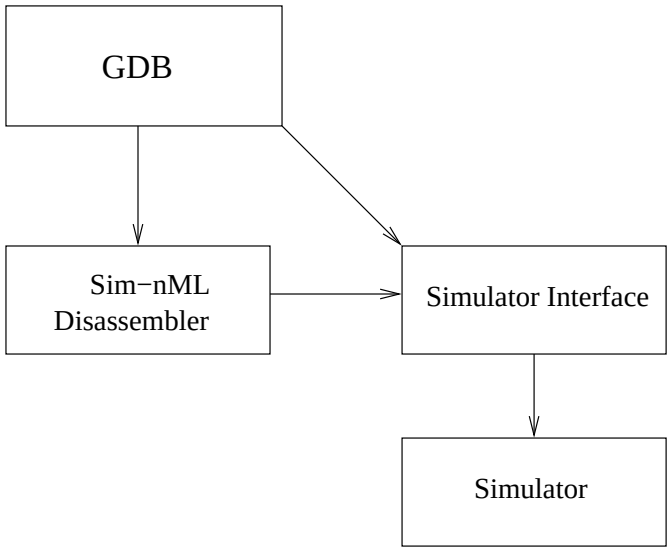

Figure 4.2: GDB Interface

The disassembler also interacts with the simulator interface [Vis06] to read the memory from the location it has to disassemble. The simulator provides the requested data from its memory management subsystem.

The signature of function that was changed in gdb is given below.

```
static int
dump_insns (struct ui_out *uiout, struct disassemble_info * di,
            CORE_ADDR low, CORE_ADDR high,
```

```
int how_many, struct ui_stream *stb)
```

Here the variables `uiout` and `stb` contain the user interface details and are used by gdb to determine how to display the disassembled data. The variables `low` and `high` contain the range of addresss to disassemble. The variable `how_many` specifies the number of disassembly lines to display. It is intended to be used by systems which use the debugger as just one small component of a larger system to control the exact amount of display. For standalone gdb, as in our case, this variable is passed as -1 which means full range of address will be disassembled and shown. To manage the display, the print functions associated with ui streams have the facility to pause after a screenful of output. The variable `di` contains architecture specific details for the current processor. Since we are using our own disassembler here, the usage of `di` is removed from this function. The function returns the number of instructions displayed.

Now whenever disassemble command is issued, it results in call to the Sim-nML disassembler. This whole process works as follows:

1. Disassemble command is given in gdb, for example, `disassemble main`
2. GDB finds out the address range of the given symbol.
3. GDB disassemble function (`dump_insns`) runs a loop from this starting address to end address of the range computed above.
   (a) In this loop Sim-nML disassembler is called with the specified address.
   (b) Sim-nML disassembler reads the data from this address through simulator interface and finds out the syntax of instruction which matches this data and prints it.
   (c) Sim-nML disassembler returns the number of bytes for this instruction.
   (d) The loop counter is incremented by this number.

This makes the whole disassembly facility in GDB retargetable with opcode information coming dynamically from the given processor description instead of in a hard-coded way.

Chapter 5

Results

In this chapter we will discuss the performance results for disassembler and conclude our thesis with insight into future work.

5.1 Experiments

We have tested our disassembler using a processor description for **PowerPC 603** architecture. The experiments conducted and the methodology is explained below.

5.1.1 Setting for Experiments

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

- **Machine1:** Intel Pentium-4 2.4 GHz processor with 512 KB cache and 256 MB RAM running Linux 2.6.12
- **Machine2:** Intel Pentium-4 3.4 GHz processor with 1MB cache and 1 GB RAM running Linux 2.6.11

We have chosen some diverse set of applications to test our disassembler. These applications are.

- **IntMatMult.c:** This is a program to multiply two integer matrices.
- **BubbleSort.c:** This is a program to sort an integer array using bubble sort algorithm.
- **Fibonacci.c:** This is a program to find the nth number in Fibonacci sequence.
- **QuickSort.c:** This is a program to sort an integer array using quick sort algorithm.
- **NQueens.c:** In this program, we have to find a way to place N queens on an NxN chessboard such that no two queens are able to capture each other.

5.2 Results

These programs were compiled using **GCC** cross compiler version 2.95.3 [gcc] for **PowerPC** and statically linked as the underlying simulator does not provide support for handling dynamic link library calls. These programs were then provided to **GDB** where we used the `disasm` command to disassemble the functions containing their core functionality. This mimics the real term scenarios as experienced in live debugging in **GBD**. The performance results obtained for **Machine1** and **Machine2** are shown in table 5.1.

5.3 Analysis of Results

We are getting an average time of disassembly per instruction in the range of 3.5-4.9 msec in case of **Machine1** and 2.5-3.8 msec in case of **Machine2**. **Machine2** clearly performs much better than **Machine1** which can be attributed to its superior configuration. We see that the relative performance of test programs is more or less the same on both the machines. One exception we find in this is that the disassembly of the program `NQueens` outperforms that of program `Bubblesort` in terms of average time per instruction in case of **Machine2** whereas it lagged in case

Program	Instructions disassembled	Total time	Avg Time per instr	Max Time of an instr	Machine
IntMatMul.c	97	356.947	3.679	18.997	Machine1
		263.96	2.721	13.998	Machine2
BubbleSort.c	117	406.937	3.478	19.997	Machine1
		292.955	2.503	13.998	Machine2
Fibonacci.c	36	175.973	4.88	19.997	Machine1
		137.981	3.832	14.998	Machine2
QuickSort.c	132	477.927	3.62	19.997	Machine1
		340.946	2.582	13.998	Machine2
NQueens.c	252	881.868	3.499	19.997	Machine1
		627.904	2.491	13.998	Machine2

Table 5.1: Performance results (All times in msec)

of **Machine1**. This is perhaps due to the larger cache and RAM size of **Machine2**. As the number of instructions are much more in case of `NQueens` the amount of available cache and memory had an increasingly important role to play.

The amount of time taken to disassemble an instruction will be directly related to where its node is located in the *IR* tree. The leftmost instruction leaf can be reached in the shortest time from the root and represents the best case traversal possible. Similarly, the rightmost leaf in the tree presents the worst case traversal scenario. This is exactly what we witnessed in the disassembly output. The leftmost leaves in this **PowerPC** *IR* (a set of unconditional branch instructions) were just four hops from the root and had a negligible and near zero traversal time. The rightmost leaves for **PowerPC** *IR* were found to be the instructions `mfspr` (move from special purpose register) and `mtspr` (move to special purpose register). To match these instructions, the traversal routines had to go through all the intermediary nodes.

These were precisely the instructions that took the maximum amount of time to disassemble as given in table 5.1.

Another interesting point to note about the results is the average time to disassemble per instructions for `Fibonacci`, which is much greater than other programs. We found out that all the functions in a **PowerPC** program contain `mfspr` & `mtspr` instructions for saving the state of special purpose register at the beginning and restoring it before returning. Since these instructions have the worst disassembly times and the `Fibonacci` program itself is very small, it results in the inflation of average disassembly time for this program.

The performance of the disassembler will also depend a lot on how the processor specification is written. Consider the example shown in figure 5.1

```
op arith_insn(x: arith_insn_type)
    image = format(``0110%s'', x.image)
op arith_insn_type =   add
                     | sub
                     | mul
                     | sub


op add (r1 : REG, r2: REG)
    image = format(``%s%s00'', r1.image, r2.image)
op sub (r1 : REG, r2: REG)
    image = format(``%s%s01'', r1.image, r2.image)
op mul (r1 : REG, r2: REG)
    image = format(``%s%s10'', r1.image, r2.image)
op div (r1 : REG, r2: REG)
    image = format(``%s%s11'', r1.image, r2.image)
```

Figure 5.1: Example instructions 1

These same set of instructions can also be written as the example shown in figure 5.2

```
op arith_insn =   add
                | sub
                | mul
                | sub


op add (r1 : REG, r2: REG)
    image = format(``0110%s%s00'', r1.image, r2.image)
op sub (r1 : REG, r2: REG)
    image = format(``0110%s%s01'', r1.image, r2.image)
op mul (r1 : REG, r2: REG)
    image = format(``0110%s%s10'', r1.image, r2.image)
op div (r1 : REG, r2: REG)
    image = format(``0110%s%s11'', r1.image, r2.image)
```

Figure 5.2: Example instructions 2

If the binary stream doesn't match the image pattern of any of these instructions, then the disassembler will need to perform four comparisons to figure this out for example 2 versus just one comparison for example 1. The way of writing instructions of example 1 is, thus, much more efficient than that of ex ample 2 in this case. So it is very important that the instruction set be written in such a way that common pattern of related instruction is as close to root as possible.

In the current **PowerPC** description, we find that this common structure of instructions is not exploited. The performance of the disassembler will improve greatly when this description is optimized as described.

Another processor description related optimization possible to improve disassembly times is to profile the target benchmark applications and find out the most

used instructions. These and related set of instructions can then be written as close to root instruction as possible in the processor description to achive a better *IR* tree. This is possible because the rules are added linearly in the *IR* as the description is parsed.

5.4 Conclusions

In this thesis, we have described the retargetable processor description language Sim-nML and its application in automatic generation of processor modeling software tool-sets for embedded systems. We have made a parser for converting the Sim-nML processor description to an Intermediate Representation as an easy and efficient interface between tool generators and the processor description.

We have also built a generic retargetable disassembler based on Sim-nML processor descriptions. We have also interfaced this disassembler with **GDB** and the Sim-nML simulator for providing a generic debugging environment. We have tested the disassembler with description of **PowerPC 603** architecture.

5.5 Future Work

We propose the following future extensions to our work.

- This is a core disassembler and works in conjunction with **GDB**. It can easily be extended to a complete standalone disassembler with the help of a driver routine.

- Our disassembler provides *raw* disassembly, for example, it gives out the actual location or offset in case of jump or branch instructions. This can be improved by giving out labels instead of actual numeric addresses. To achieve this, we will have to provide a way to identify the jump, branch and other such instructions. This can be done by providing a separate configuration file containing the details of all such instructions.

- The Sim-nML class hierarchy in the form of Intermediate Representation together with traversal library provides a unique platform for development of retargetable tools. Various processor modelling tools such as performance simulators, compiler back-end, assemblers etc can be developed using this platform.

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.

[Fre93] Markus Freericks, The nML Machine Description Formalism, TU Berlin Computer Science Technical Report, July 1993.

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

[gcc] GCC PowerPC Cross Compiler for i386. https://gforge.inria.fr/plugins/scmsvn/viewcvs.php/packages/cross-ppc-toolchain-2.95.3-1.i386.rpm?rev=13&root=wifibotlib

[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] Trimaran Tool Set, *The MDES User Manual*, 1997, http://www.trimaran.org/docs/mdes_manual.ps.

[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.

[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.

[RM99] V. Rajesh and R. Moona, Processor Modeling for Hardware Software Codesign, In *International Conference on VLSI Design*, Goa, India, Jan 1999

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

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

[xten] Tensilica, Inc., The Xtensa 7 Processor for SOC Design, http://www.tensilica.com/products/xtensa_7.htm

[Tri] Trimaran: An Infrastructure for Research in Instruction-Level Parallelism, http://www.trimaran.org.

[ver] IEEE Standard Verilog hardware description language. http://ieeexplore.ieee.org/xpl/standardstoc.jsp?isnumber=20656&isYear=2001.

[vhd] IEEE Standard VHDL Language Reference Manual. http://ieeexplore.ieee.org/iel5/7180/19335/00893288.pdf.

[Vis06] Surendra Kumar Vishnoi, Functional Simulation Using Sim-nML, Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, May 2006.

[xil] Xilinx Corp, Virtex-5 Multi-Platform FPGA, http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex5/index.htm

Appendix A

Sim-nML

A.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, dissembler, 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

A.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.

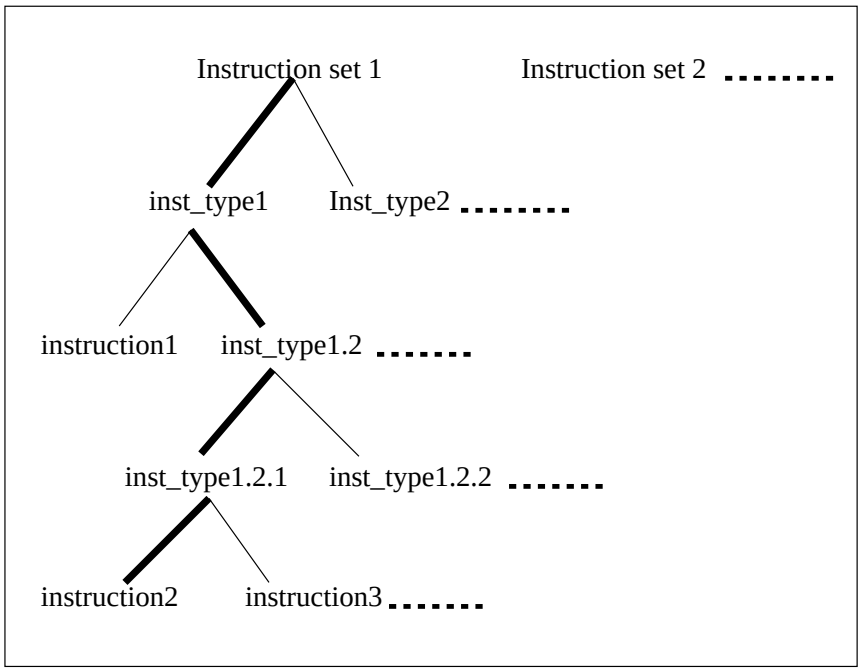

Figure A.1: Hierarchical tree structure for Instruction set

Figure A.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.

A.1.2 Example processor description

Figure A.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 A.2: Sim-nML description for a Simple hypothetical processor

A.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.

A.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*

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

For example, line 37 in figure A.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 A.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,*

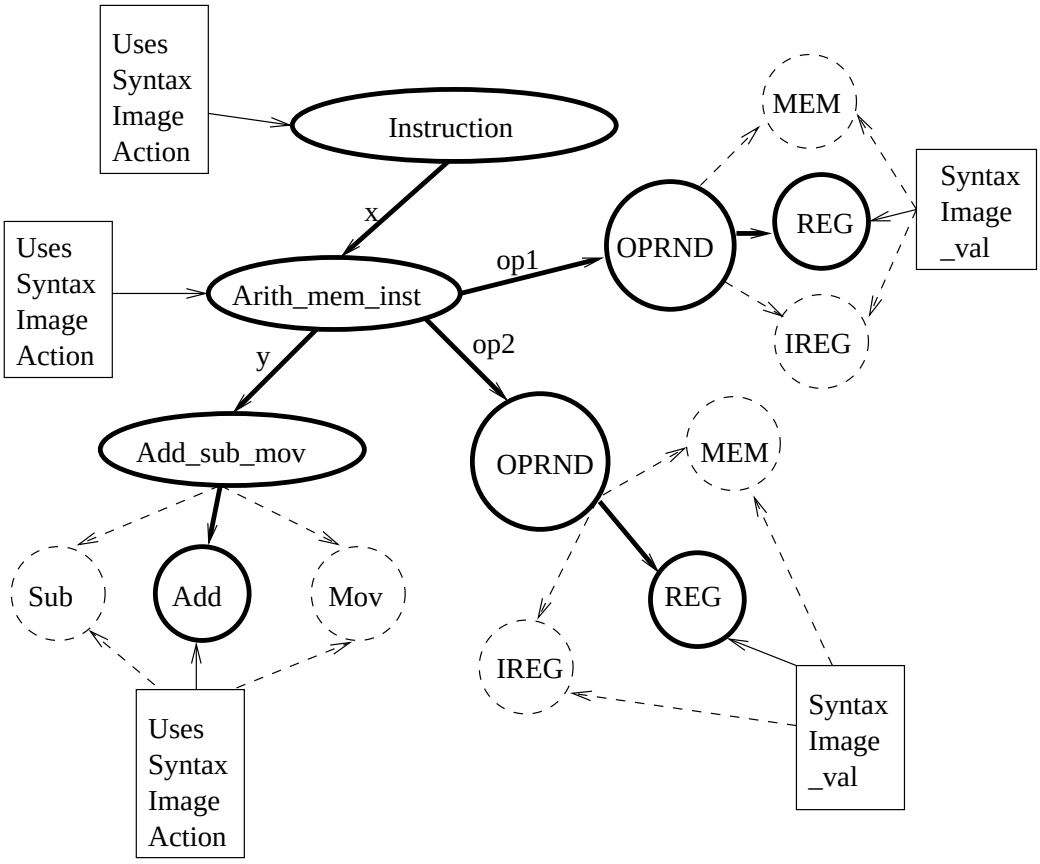

Figure A.3: Derivation of Add instruction from description given in figure A.2

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

  mode $n_0 = n_1 \mid n_2 \mid n_3 \ldots$

- *AND rule*

  mode $n_0(p_1\text{: } t_1, p_2\text{: } t_2, p_3\text{: } t_3 \ldots)$ [= 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 A.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).

A.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 A.2.
- *Parameter attribute:* Defined using notation "Parameter.attr", where "attr" is of syntax type. For example, x.syntax at line 23 in figure A.2.
- *Format expression:* Defined using expression "format" (such as in line 28 of figure A.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 A.2.
- *Parameter attribute:* Defined using notation "Parameter.attr", where "attr" is of image type. For example, x.image at line 24 in figure A.2.
- *Format expression:* Defined using expression "format"(such as in line 29 of figure A.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.

A.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 A.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**,** $\ldots$ id_i**):** Defines an enumeration type, where constants id_1=0, id_2=1, $\ldots$ 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.

A.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.

A.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.

A.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.

A.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 A.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 A.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.

A.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 A.4. Instructions acquire and release the resources specified in the clauses in a sequence during execution. In figure A.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 A.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 A.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 A.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 in-

coming 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 A.4: Example Sim-nML description

A.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

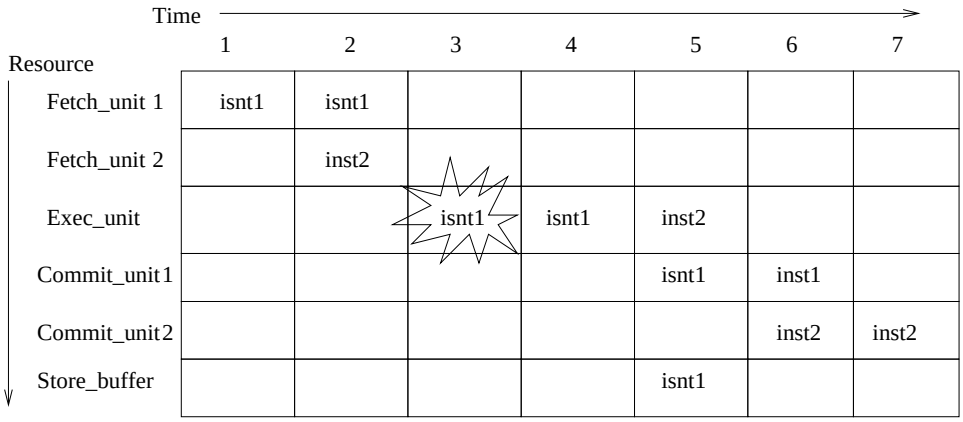

Figure A.5: Reservation table for example shown in figure A.4

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

A.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

A.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>

A.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)

A.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.

A.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 A.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).

A.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 A.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.

A.7 Coercing rules

As explained in section A.4.3, one data type can be explicitly converted to another data type using expression *coerce*. Type coercion rules are shown in table A.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			
		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	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 A.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 A.2: Type coercion rules

Appendix B

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 ')'   //Incomplete

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

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
                        | '<' ID
                        | ID '>'
                        | ID '[' ']'
                        | ID '[' Expr ']'
                        | RegOpr

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 C

Operators in IR

Name of Operator	Symbol	Arity of Operator
Addition	+	Binary
Subtraction	-	Binary
Multiplication	*	Binary
Division	/	Binary
Modulo	%	Binary
Exponentiation	**	Binary
Greater than	>	Binary
Less than	>	Binary
Equal to	==	Binary
Not equal to	!=	Binary
Greater than or equal to	>=	Binary
Less than or equal to	<=	Binary
AND	&	Binary
OR	|	Binary
XOR	∧	Binary
Logical AND	&&	Binary
Logical OR	||	Binary
Left Shift	<<	Binary

Right Shift	>>	Binary
Left Rotate	<<<	Binary
Right Rotate	>>>	Binary
Dot	.	Binary
Concatenation	::	Binary
Indexing	[ ]	Binary
Assignment	=	Binary
Statement Separator	;	Binary
Unary Plus	+	Unary
Unary Not	!	Unary
Unary Minus	-	Unary
Bitwise Complement	∼	Unary
Bit Range	..	Ternary
If	if then else	Ternary
Function	canonical function	n-ary
Switch	switch	n-ary
default expression	default	0-ary
NULL	Nothing	0-ary
Colon	:	Binary

Table C.1: List of operators used in IR

Appendix D

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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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*.

D.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.

D.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*.

D.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.

D.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.

D.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*.

D.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*.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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 A.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.

D.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*.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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 A.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.

D.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, bool traversal_type);
```

It takes a pointer to the *ToolSpec class* as parameter to carry tool specific information. The type of traversal, complete or guided, is specified by `traversal_type`.