Processor Models for Retargetable Tools*

Rajat Moona
moona@iitk.ac.in
Department of Computer Science and Engineering,
Indian Institute of Technology, Kanpur 208 016

Abstract

This paper describes a methodology for developing processor specific tools such as assemblers, disassemblers, processor simulators, compilers etc., using processor models in a generic way. The processor models are written in a language called Sim-nML [1] which is powerful enough to capture the instruction set architecture of a processor.

We describe a few tools in this paper which can be retargeted to any processor using the high level Sim-nML model of the processor.

1. Introduction

In applications that involve the development of application specific processors, often there is a need to generate various processor specific tools such as assemblers, compilers, simulators etc. Often unavailability of such tools cause the design to fail as the development of tools is a time consuming process and results in delay in the application development.

The problem gets compounded in scenarios where several design alternatives are to be explored. It essentially involves, rewriting the tools completely as well as rewriting the application with each design alternative. While some of it can be simplified by having executable specifications at a high level, the process does not yield high accuracies in estimating costs for choosing the design alternatives.

Automatic generation of tools is an attractive option which saves design time and is cost effective. In such a process, the instruction set architecture of the processor can be described in an abstract way using a high level language and various tools can be generated using tool generators.

*This work is supported by Cadence Research Centre at IIT Kanpur, India.

2. Related work

Several works have been reported in the literature in this direction. Our own work is highly influenced by the nML work done by Freerick et.al. [2]. While it is possible to specify the instruction set architectures in the nML, it is difficult to specify processors where multiple instructions execute and interact simultaneously such as pipelined processors, superscalar processors, VLIW processors etc.

ISDL [3] work by the group at MIT aims at developing retargetable assemblers, disassemblers etc. They are also exploring the ways to synthesize the processor out of the ISDL specifications.

New Jersey tool-kit developed in the SLED [4] work essentially uses the specification level of the instruction set coding for the processors which can be used by the retargetable assemblers and disassemblers.

PlayDoh [5] architecture specification methodology developed by the HP laboratories aims at developing performance oriented compilers for the VLIW and superscalar processors. While SimOS [6] work is not directly relevant here, our work has been greatly influenced by this work in achieving high speeds of the simulation.

3. Sim-nML and Processor Models

Sim-nML [1] is a language used to describe the instruction set architecture of processors with minimal knowledge about its microarchitecture. It is possible to use the processor models developed in Sim-nML for generation of various processor specific tools such as assemblers, disassemblers, processor simulators, debuggers, functional simulators, compilers etc. in a generic way.

The processor models in Sim-nML are described using attribute grammar in a hierarchical manner. To facilitate this, Sim-nML defines two kinds of primitive rules, namely, op-rules and mode-rules. While op-rules are primarily used for describing instructions within the processor, mode-rules describe the addressing modes for the operands. A top level

op-rule, called `instruction` is used to describe the collection of all instructions of the processor in a hierarchical manner. There are two types of constructions supported in the Sim-nML. The *and*-rule constructions are used for the terminal symbol definitions. The *or*-rules are non-terminals which can expand to further *and*-rules or *or*-rules or both.

Sim-nML supports the resource tree level specification of the processor wherein the processor is viewed as a collection of several hardware resources. As the instructions descend the processor microarchitecture for execution, they utilize several resources each for a pre-defined and known amount of time. In case, a resource is not available, the instruction waits till instructions ahead of it release the resource. This model is extremely powerful and can capture the timing details of modern processors such as pipelined processors, VLIW processors, superscalar processors etc. An example model for a superscalar processor is shown in here (figures 1 and 2). In this processor, there are several resources. All instructions are read by the resource instruction fetch unit (IFU). The instructions are of variable length (16 bits or 32 bits) and accordingly take one or two units of time at the IFU. In addition, there is a branch unit, a resource that is used by the branch instruction. The target address of the branch instruction can be either PC relative or direct addressing. There are two ALUs, both alike, and are used by the ADD instruction (and other ALU instructions) or by the load store instructions. In addition there is a write back unit (wb) which is used by the instructions that need to write the result back into the registers.

In the model, two instances of the ALU resource are available. However, the instructions that use the ALU can use any one of these two resources. This way it is simple to specify the superscalar model of the processor. A three way superscalar processor with 3 integer units can be obtained by just changing the resource declaration.

Registers are special resources declared using the `reg` declaration. For each register, there are multiple read port resources and one write port resource is declared by default. For read from the registers, any one of the read port is needed. For writing in the register, all its read ports and one write port are needed. This ensures that no instruction can read the register while some other instruction modifies the same register.

Sim-nML supports the canonical functions which are plugged from outside by the tools. In this example (figure 1), memory access times are modeled by a simple random number generator. It is assumed that the system has a data cache and it takes one unit of time to access from the cache in case of a hit to supply the data. In case of a cache miss, the memory takes 10 units of time. In our example model, we have assumed a 95% cache hit ratio.

This type of canonical function interface, permits very powerful tools to be built using generic tools. For example, an on-line cache simulator can be plugged in by making a canonical call to the simulator in each memory access.

4. Our Approach

In our approach, we specify the processor models in Sim-nML. The models thus developed are parsed, checked for the errors and converted to a compact intermediate form (IR). It is fairly simple for various tools to read the information from the IR.

The IR is organized as a collection of tables. Each table contain various fixed size records. A 'table of index' table in the beginning provides the size and locations of the tables in the file. Thus a tool need to look at only those tables which are needed by it and can ignore the rest of them. An assembler for example need to find only the syntax and image attributes for various instructions. It therefore can read only those tables which provide these informations.

Figure 3 provides a typical scenario where the Sim-nML based models are used for tools development. A tool called `irg`[7, 8] is used to generate the IR specification of the processor model from the Sim-nML based model. Various tools and meta-tools use this IR model and corresponding processor binaries.

5. Retargetable tools

In this section we describe various tools developed at cadence research centre. All these tools work with the Sim-nML based flow and are retargetable. The tools have been developed keeping in mind the endian-ness of various processors. In our approach, therefore, it is possible to have tools developed on a processor whose endian-ness differs from that of the target processor. For example, the disassembler for the PowerPC processor generated using Sim-nML model works on the Sun Sparc processor as well as on the Intel x86 platform. Even the IR generated is a universal one. Thus the IR generated on one processor can be used for the tool generation on another processor.

5.1. Retargetable Assembler Generator

The retargetable assembler generator generates assembler for a processor from its model. The assembler emulates GNU assemblers for various processors. The generated assembler uses the similar pseudo operations as in various GNU assemblers. The instruction set format used by the generated assembler is the one that is described in the Sim-nML specifications for the processor.

The assembler generator generates several files which are together used to generate the assembler. To compile the generated assembler, various tools such as GNU flex and

```
type word = card(16)
type byte = card(8)
reg R[4, word]
reg PC[1, word]
mem M[2**16, byte]
resources ifu, bu, lsu, alu[2], wb

//Addressing Modes
mode immediate(x:word) = x
  syntax = format("%d", x)
  image = format("%16b", x)
mode register(i:card(2)) = R[i]
  syntax = format("R%d", i)
  image = format("0%2b", i)
mode direct(addr:word) = M[addr]::M[addr+1]
  uses = if "rand"() < 0.95 #1 else #10
  syntax = format("%d", addr)
  image = format("%16b", addr)
mode reg_indirect(i:card(2)) = M[R[i]]::M[R[i]+1]
  uses = if "rand"() < 0.95 #1 else #10
  syntax = format("(R%d)", i)
  image = format("1%2b", i)

op instruction(x:inst_type)
  uses = x.uses
  syntax = x.syntax
  image = x.image
  action = x.action
op inst_type = addinst | branch | loadstore
op addinst(x:addtype)
  uses = x.uses
  syntax = format("ADD %s", x.syntax)
  image = format("00000001%s", x.image)
  action = x.action
op addtype = addRtoR | addItoR
op addRtoR(R1:register, R2:register)
  uses = ifu#1, alu#1, wb#1
  syntax = format("%s, %s", R2.syntax, R1.syntax)
  image = format("00%s%s", R1.image, R2.image)
  action = {
    R2 = R2 + R1;
    PC = PC + 2;
}
op addItoR(x:immediate, R:register)
  uses = ifu#2, alu#1, wb#1
  syntax = format("%s, %s", R.syntax, x.syntax)
  image = format("01%s000%s", R.image, x.image)
  action = {
    R = R + x;
    PC = PC + 4;
}
```

Figure 1. Addressing modes, top level instruction and flavors of ADD instruction

```
op branch(x:branchtype)
  uses = ifu#2, alu#1, bu#1, wb#1
  syntax = format("JMP %s", x.syntax)
  image = format("00010000%s", x.image)
  action = x.action
op branchtype = branchrelative | branchabsolute
op branchrelative(target:card(16))
  syntax = format("%d", target)
  image = format("00000000%16b", target −($ +4));
  action = {
    PC = target;
}
op branchabsolute(target:card(16))
  syntax = format("%d", target)
  image = format("00000001%16b", target);
  action = {
    PC = target;
}

op loadstore = load | store
var tmp[1,word]
op load(r:register, l:loadmode)
  uses = l.uses
  syntax = format("LOAD %s, %s", r.syntax, l.syntax)
  image = format("00010000%s%s", r.image, l.image)
  action = {
    l.action;
    r = tmp;
}
op loadmode = load_direct | load_indirect
op load_direct(m:direct)
  uses = ifu#2, m.uses, wb#1
  syntax = format("%s", m.syntax)
  image = format("01000%16b", m.image)
  action = {
    PC = PC + 4;
    tmp = m;
}
op load_indirect(r:reg_indirect)
  uses = ifu#1, r.uses, wb#1
  syntax = format("%s", r.syntax)
  image = format("00%s", r.image)
  action = {
    PC = PC + 2;
    tmp = r;
}
  .
  .
  .
```

Figure 2. Flavors of Branch and Load instructions

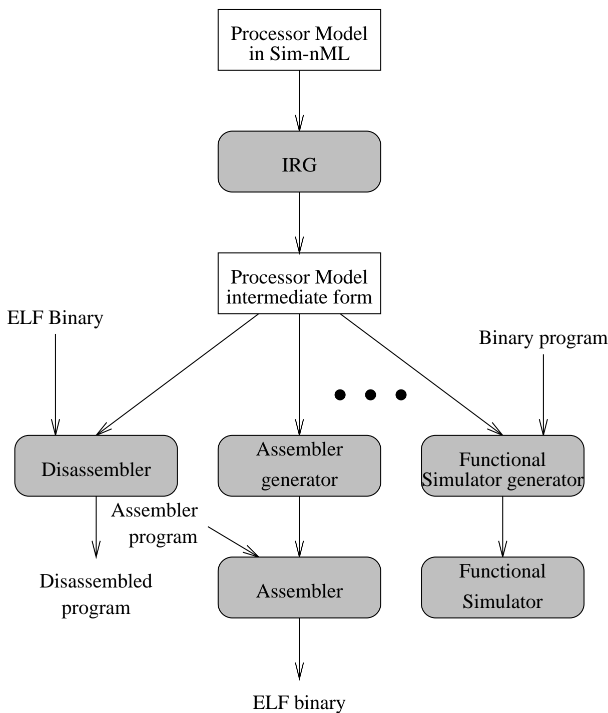

Figure 3. Typical flow for the Sim-nML based development

```
From the binary file, extract the addresses of
        various functions;
put these addresses in start_code_block list;
i=0;
while start_code_block list is not empty {
  //take out the address from the list
  address = listout(start_code_block);
  code_fragment_start[i] = address;
  follow binary till an instruction that changes PC;
  if a call instruction then
    put the target address in start_block_list;
  else
    code_fragment_end[i] = address of the
                 current instruction;
    i = i+1;
}
Generate label for each code_fragment_start;
Merge adjacent code_fragments into one;
```

Figure 4. Code block identification for the disassembler

GNU bison are used. The output of the generated assembler are the binary in the ELF [9] format and various list files. The ELF standard specifies certain processor specific values for the relocatable variables. To implement this, the assembler generator uses an external configuration specification and generate such relocation information. The generated assembler is a traditional two pass assembler.

5.2. Retargetable Dis-assembler

The retargetable disassembler uses the intermediate representation of the processor model and performs full symbolic disassembly of an ELF binary program for that processor. The disassembler performs various analysis on the binary program. The first analysis is to find out the code and data area. To do so, the disassembler first finds out the code blocks. The algorithm given in figure 4 is used to do that. The algorithm works as follows. From the ELF binary, first find out the addresses of various functions in the program. This information is stored in the symbol table section of the binary. Assuming that each of these will be starting point of the code block, the code is traced till a branch instruction is found. The branch instruction is used to denote the end of the code block. A call instruction provides another starting point of a code block. The process is repeated as long as there is any untraced starting address. For each starting point, a label is used which is either the name provided in the symbol table of the binary file or the one that is generated internally by the disassembler.

In the second part, the data analysis is done. Depending on the access pattern of the data, it is defined as byte array (`.byte` pseudo op), or a word array, or string of characters etc.

The third part of the analysis involves the generation of the assembly language program using the binary program.

At all steps, the Sim-nML specification is used to identify various instructions. For example, the branch instructions are the ones that modify a program counter register in the Sim-nML specifications. The call instructions are the ones that save the program counter before modifying. These heuristics, though prone to failure, work wonderfully for most programs.

5.3. Retargetable Functional Simulator

The retargetable functional simulator [10, 11] takes a binary program for a processor and ‘compiles’ it into a C program that is functionally equivalent to the binary program running on the processor. Essentially, for each instruction in the binary program, corresponding C code fragment is generated that simulates the action for the instruction. The functional simulator take the processor model in its intermediate form and a processor binary in ELF to generate a

functionally similar C program. The generated functional simulator program (`fsim.c`) contains the following.

1. A set of functions, one for each instruction in the instruction set description of the processor.
2. An initialized table of function pointers, along with the parameters. The table contains the instructions for the programs including the operands as parameters.
3. The memory image of the program initialized from the binary program.
4. Registers and other memory elements of the processor constituting the visible state of the processor. These are mapped to the host data structures.
5. A driver routine that initializes the memory and registers and calls the first and subsequent functions stored in the function pointer table. Thus it implements the program's functional simulation.

In the generated simulator, various calls to the library functions can be routed to other functions in the host program. This is achieved by a configuration file that specifies the way the parameters are passed. The generator then appropriately calls the routed function passing parameters from the target address space. Similarly the values returned are put back into the target address space. Thus it is possible, for example, to route the `printf` and `scanf` functions in the target code, to the host functions that will do the input output on the host and pass the values to the target.

The simulation speeds for this simulator exceed 1 MIPS on host processors like 233 MHz Pentium for a PowerPC 603 model[10].

Various other tools have been developed around the functional simulator. These include, the instruction trace generator[10], code instrumentation tool[8], on-line cache simulation[8] etc.

5.4. Retargetable timing simulator

The timing simulator takes the processor binary and simulates each of its instruction while applying the resource graph model of the Sim-nML. As the instructions descend the processor microarchitecture they utilize various resources. The simulator takes a model wherein all instructions of the program are enabled to execute. However, because of the resource conflicts all instructions can not be allowed to execute. In such cases, the simulator takes instructions in the program order of execution and makes other instructions wait. The data dependencies are captured by assuming that the registers are also the resources. With the speeds of the simulation being around 4000 instructions per second, the simulator is rather slow and the work is going on to speed it up further.

5.5. Retargetable Compiler Back-end generation

This work, though in a very early stage, aims at generating the GNU `.md` files which are the machine description files for the GNU C compiler. Our aim here is to be able to generate most of the back-end automatically and resort to the manual generation of various routines such as floating point library etc.

5.6. Retargetable Processor Synthesis

We are also aiming at the automatic synthesis of the processor from its instruction set architecture description. We intend to synthesize the datapath as well as the controlpath of the processor. This work is again in very early stage.

6. Conclusion

We have shown the effectiveness of the high level processor models in developing various retargetable processor specific tools. With the increase in the number of application specific processors, this approach provides a very attractive solution to the tools generation problem.

We have developed the processor models for the PowerPC 603[12], Motorola 68HC11[13], Intel 8085[14] processors. Various tools developed work for all such processors. Currently we are also developing processor models for Sun Sparc[15], ARM[16], DLX[17] and ADSP 2100 [18] series processors.

Much of the information about the work is available at `http://www.cse.iitk.ac.in/sim-nml`. Various tools are also available in public domain downloadable from the same site.

Acknowledgement

The author would like to acknowledge Prof. S.K. Aggarwal, Prof. Deepak Gupta, V. Rajesh, N. C. Jain, K. Krishna, Y. Subhash Chandra, Rajiv A.R., Sarika Kumari, P. Pogde and others in the Cadence Research Centre for their contribution in the Sim-nML based projects and to make them a reality.

References

[1] V. Rajesh, *A Generic Approach to Performance Modeling and its Application to Simulator Generator* Master's thesis, Department of CSE, IIT Kanpur, 1998. (`http://www.cse.iitk.ac.in/research/mtech1996/9611132.html`).

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

[3] G. Hadjiyiannis, S. Hanono and S. Devadas, *ISDL: An Instruction Set Description Language for Retargetability*, Proceedings of the 34th Design Automation Conference, 1997.

[4] N. Raksey and Fernandez, *Specifying representations of machine instructions*, ACM transactions on Programming Languages and Systems, 19(3), May 1997.

[5] Vinod Kathail, Michael Schlansker, B. Ramakrishna Rau, *HPL PlayDoh Architecture Specifications: Version 1.0* HP Laboratories Technical Report, HPL-93-80, February 1994.

[6] Mendel Rosenblum, Edouard Bugnion, Scott Devine and Stephen A.Herrod, *Using the SimOS Machine Simulator to Study Complex Computer Systems*, ACM Transactions on Modeling and Computer Simulation, Jan 1997, vol. 7, no. 1, pp. 78-103.

http://simos.stanford.edu

[7] N.C. Jain, *Disassembler using high level processor models*, Master's thesis, Department of CSE, IIT Kanpur, 1999,

(http://www.cse.iitk.ac.in/research/mtech1997/9711113.html)

[8] Rajiv A.R., *Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation*, Master's thesis, Department of CSE, IIT Kanpur, 1999.

(ftp://www.cse.iitk.ac.in/pub/moona/sim-nml/simnml-rajiv-thesis.ps.gz)

[9] *Executable and Linkable Format (ELF), version 1.1*, Tools interface Standards (TIS), Portable formats specifications.

[10] Subhash Chandra, Rajat Moona, *Retargetable Functional Simulator Using High Level Processor Models*, Proceeding of 13th International Conference on VLSI Design, January 2000. (ftp://www.cse.iitk.ac.in/pub/moona/sim-nml/simnml-fsimg-vlsi2000.ps.gz)

[11] Y. Subhash Chandra, *Retargetable functional simulator*, Master's thesis, Department of CSE, IIT Kanpur, 1999,

(http://www.cse.iitk.ac.in/research/mtech1997/9711121.html)

[12] IBM Microelectronics and Motorola, *PowerPC 603 RISC Microprocessor User's Manual*, 1995.

(http://mot-sps.com/powerpc)

[13] Motorola Inc., *M68HC11 Reference Manual*, 1994

(http://mot-sps.com/mcu/documentation/pdf/hc11rmr3.pdf)

[14] Gaonkar R.S., *Microprocessor Architecture, Programming and Application with 8085/8080A*, New Age International Publication, 1995

[15] David L. Weaver, Tom Germond, *The SPARC Architecture Manual v9*, Prentice Hall, 1994.

[16] D.V. Jaggar, *Advanced RISC Machines Architecture Reference Manual*, Prentice Hall, London, 1996

also, http://www.arm.com/.

[17] J.L. Hannessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1996.

[18] Analog Devices Inc., *ADSP-2101/2 User's Manual*, 1988.