Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign

Mark R. Hartoog, James A. Rowson, Prakash D. Reddy, Soumya Desai, Douglas D. Dunlop, Edwin A. Harcourt, Neeti Khullar

Alta Group of Cadence Design Systems, Inc.

Abstract

An experimental set of tools that generate instruction set simulators, assemblers, and disassemblers from a single description was developed to test if retargetable development tools would work for commercial DSP processors and microprocessors. The processor instruction set was described using a language called nML. The TMS320C50 DSP processor and the ARM7 microprocessor were modeled in nML. The resulting instruction set models execute about 25,000 instructions per second, and compiled instruction set simulation models execute about 150,000 instructions per second. The viability of this approach and the deficiencies of nML are discussed.

1. Introduction

Models and development tools for target processors are an important part of a Hardware/Software Codesign system. Commercial processors generally have instruction set models, assemblers, debuggers and compilers supplied by the vendor. We have integrated some of these vendor models into a Hardware/Software Codesign system. Each of these vendor supplied sets of development tools is different, and whenever a user switches to a new processor he must learn to use a different set of development tools with different capabilities. It would be much better if a consistent set of retargetable development tools were available for a variety of processors.

Currently vendors devote significant effort to producing a set of good development tools for their processors. There has been much discussion about custom or application specific processors, but the effort to create the development tools for these custom processors has so far held back the wide spread acceptance of this approach. If it were possible to significantly reduce the effort required to produce a suite of development tools through a retargetable approach, this would appeal to many vendors of standard and custom processors.

While we are interested in models and development tools for embedded microprocessors and DSP processors, considerably less work is being done on DSP processors and they have many features that are not found in microprocessors. For this reason we decided to focus our study on retargetable development tools for DSP processors, but evaluate them for microprocessors as well.

Several approaches have been taken to the problem of retargetable DSP processor models. The FlexWare tools set [1] included a retargetable instruction set model called Insulin [1, 2]. Insulin was built around a partially reconfigurable VHDL simulation model of a generic processor. It could be reconfigured for bit width, number of registers, number of ALUs, etc. Insulin cross assembled target processor instructions into micro instructions for the generic processor, which were then executed on the generic VHDL model. The processor description was a list of generic processor micro instructions to execute each target processor instruction.

A different approach was used by the Technical University of Berlin and IMEC. The Technical University of Berlin (TUB) developed a language called nML [3, 4] for describing processor instruction sets. TUB developed an instruction set simulator called SIGH/SIM [5], which worked from the nML description and a code generator called CBC [6]. IMEC independently developed a code generator called CHESS [7] which used nML and an instruction set simulator called CHECKERS.

During our initial investigations into DSP code generation, it became clear that an instruction set level description was necessary for a number of tasks. We decided to develop a number of tools using nML as an experimental method to describe processors. The main objective was to determine whether a good suite of development tools could be generated for commercial DSP processors and microprocessors using this kind of a retargetable approach.

2. nML

The nML language [4] relates three different views of an instruction set: the behavior, the assembly syntax, and the object code image. It also allows the instruction set to be broken down into classes of instructions with shared code, syntax, and image. Addressing modes are also supported explicitly.

nML is a form of attributed grammar. An attributed grammar is a declarative description of a language with attributes added onto the "rules" of the grammar. Each view of the processor (action, syntax, and image) is an attribute attached to the rule for that instruction or addressing mode.

In addition to the rules that describe the instruction set, nML by necessity has some other ways to declare types, memories, etc.

The nML language has 3 important language features:

- Declarations
- Addressing Modes expressed as rules
- Instructions expressed as rules

Declarations are used to define the internal states of the processor. nML assumes that all internal states are memories of some bit-width. A single register will be a memory of size 1.

Both instructions and addressing modes can have two forms: an AND form that requires all the arguments, and an OR form that selects from alternatives.

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..$3.50

Addressing modes are used to compute effective addresses. The behavior for an addressing mode is an expression. Addressing modes can appear on the left or right hand side of the behavior in an instruction, so the expressions that are legal are those that can be on the left hand side of an assignment.

Instructions have an RTL-like behavior. Because the formal argument to an instruction can be an OR addressing mode, a single instruction rule can represents quite a rich collection of actual instructions.

Both instructions and addressing modes have “syntax” and “image” attributes. The syntax attribute defines the assembler syntax for the instruction or addressing mode. The image attribute defines the object code encoded as a string of 1’s and 0’s.

This is a simple example of nML addressing mode rules:

```
mode MEMORY(i:index)=M[R[i]]
syntax=format("(R%d)",i)
image=format("0 %4b",i)

mode REGISTER(i:index)=R[i]
syntax=format("R%d",i)
image=format("1 %4b",i)

mode ADDR = MEMORY | REGISTER
```

In this example `mode` is a keyword that identifies these as addressing modes. `MEMORY`, `REGISTER`, and `ADDR` are the names of addressing modes in this example. The mode `ADDR` is an OR rule which is either the `MEMORY` or `REGISTER` mode. The first two addressing modes are AND rules, which have expressions after the equal sign describing the behavior of the addressing mode. In this example `M` is a memory, `R` is a register array and `index` is a data type defined to be a four bit unsigned integer. The `MEMORY` mode expression addresses the memory location whose address is in the `i`th register. The syntax and image attributes are defined using a nML `format` statements. A `format` statement is similar to the C printf function, with the extension that `%b` is a binary format. These format statements indicate how to extract the parameter `i` from the assembler syntax and from the object code for the addressing mode.

A simple example of nML instruction rules is:

```
op add_op(src:ADDR,dst:ADDR)
action={ dst = src + dst; }
syntax = format("add %s,%s",src.syntax,
   dst.syntax)
image = format("000001 %b %b",dst.image,
   src.image)
```

In this example `op` is a keyword which identifies this rule as an instruction rule and `add_op` is the name of the rule. The parameter, `src` and `dst`, are both of type `ADDR`, which is the OR addressing mode shown above. This means that these values may be addressed by either of the addressing modes in the `ADDR` rule. The action attribute describes the RTL behavior of this instruction. When you evaluate this behavior, `src` and `dst` get replaced by the expressions for the addressing modes decoded from the image or assembly code. For example, if src is `MEMORY` mode, then you can replace src in the action by `M[R[i]]`, where `i` is the four bit register index decoded from the image or syntax. The syntax and image attributes are nML format statements, similar to the addressing rules above, but with the extension that they include the syntax or image attribute of the `src` and `dst` parameters.

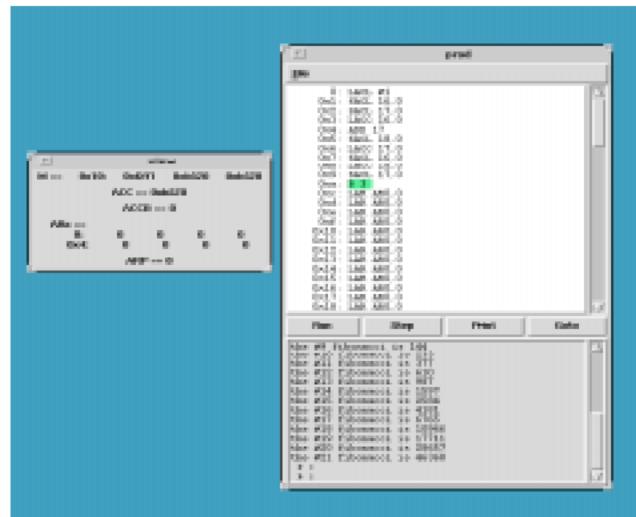

Figure 1 Debugger with register display window

3. Tools

Given this kind of declarative description of an instruction set, it is possible to generate automatically several different useful tools:

- Instruction Set Simulator (ISS) - use the image to decode the instruction and then execute the behavior
- Compiled Instruction Set Simulator (CISS) - same as above, but decoding is done at model generation time to produce a simulation mode for a processor running a specific program [8].
- Disassembler - use the image to decode the instruction and then produce the assembler syntax for that instruction
- Assembler - parse the assembler syntax for the instruction and generate the image
- Code Generator - given a piece of behavior, use the behaviors as patterns to “cover” the behavior, generating a sequence of object code images that will execute with the correct semantics
- Code Retargeter - read the image for one processor and generate an in-memory behavior, then use the behaviors from a different processor to pattern match and generate a different set of object code images

We have built the first 4 programs using nML as the input. Given an nML description we produce a C++ instruction set simulator and disassembler, which can then be compiled and run within a processor independent debugger written using C++ and Tcl/Tk (see Figure 1). An assembler was written that used the same nML description to assemble code fragments in an interpreted way (although we could have generated a YACC grammar and made a more efficient assembler). The assembler is still relatively incomplete, handling the assembly of individual instructions but with no directives, variable declarations, labels, etc.

The compiled instruction set simulator [8] decodes a program and generates a model for that program running on a processor. This has obvious restrictions, such as no self modifying code, but because the decode is done once at model generation time, it produces models that are much faster than conventional ISS models which decode each instruction every time it is executed

4. Instruction Set Models

We developed two different ISS generators. The first one, called ML, only produced ISS models. The second one, called Markus, was more general, and supported both ISS and CISS model generation.

The ML model generator has a front end which parses the nML description and builds a data structure to represent it. The ISS model is then produced from the data structure. In the ISS model there is a load and a store function for each addressing mode which is passed the image bit string as an argument. The constant zero and one bits in the image attributes are used to decode the addressing mode. When an addressing mode is decoded, then parameters, like the index `i` in the example addressing modes above, are extracted from the image bits to get or set the value in the location addressed by the mode. An OR addressing mode load or store function tries each of its rules to see if the image bit string can be decoded by any of its addressing mode rules.

The ML models also has action functions for each instruction rule. These action functions are passed the image bit strings for the instruction. Again, the constant zero and one bits in the image attribute are used to decode whether this is the correct instruction. The image attribute is also used to extract the image bit strings of addressing mode parameters, like `src` and `dst` in the above example, from the image bit string of the instruction. Then the body of the action attribute is executed, with the addressing mode parameters replaced by calls to the load or store functions for the addressing mode. Each of these calls to a load or store function is passed the image bit string corresponding to its parameter.

The Markus model generator was designed to produce both ISS and CISS models. To do this the instruction decode generation and the action execution generation were decoupled so that the ISS and CISS model generation could share the same basic code. Markus has a front end which parses the nML description into an Intermediate Representation (IR). There is a decode generator which analyzes the IR and generates a C++ program that decodes an instruction by parsing the image attributes in the nML rules. The generated C++ returns a data structure that represents the decoded instruction. The action compiler uses the nML action attributes from the IR to generate C++ code to carry out the operations specified in the action. In the ISS model the data structure returned by the decode routine is passed to the action routine to carry out the instruction. In the CISS the decode of the target program is done at model generation time and the data structure returned by the decode routine is used to generate calls to the action routines with the constant arguments decoded from the program. The CISS model then is just a series of calls to the appropriate action routines to execute the target program.

The debugger, shown in Figure 1, worked with models from the ML ISS generator and could have been easily adapted to work with the Markus ISS models. Something similar to this can be generated for a CISS model [8], but our CISS model generator output C/C++ preprocessor directives which pointed to the source code of the target processor program and the nML description. This allow you to use a standard C debugger to step through the target program and the nML description of the instructions at the same time.

5. Assembler and Disassembler

The ML ISS generator included a disassembler generator. The disassembler is very similar to the ML instruction set model. When a rule and its parameters are decoded, instead of executing the action or expression, the syntax attribute is evaluated to produce a string containing the assembler instruction. This disassembler was used as part of the debugger show in Figure 1. It did not produce symbolic names for memory locations, although it could easily have been extended to do this.

The assembler currently works by matching against lists of patterns built up from the nML syntax attributes. When a match is found, the corresponding image attributes are evaluated to produce the object code. This assembler is intended to be used as part of the debugger and currently does not support labels or symbolic location names.

6. Instruction Set Model Performance

We used the TI TMS320C50 to try out our tools on a real commercial DSP processor. The C50 description is 1988 lines so far (although some instructions are incompletely implemented because of nML limitations). The ISS model and disassembler from the ML generator is 9004 lines of code. This ISS runs a very simple repetitive program (generating Fibonacci numbers) at approximately 26,500 instructions per second on a SPARC 20.

We used the ARM7 microprocessor to try our tools on a commercial microprocessor. The ARM7 description is about 1000 lines of nML. We used Markus model generator and tested both ISS and CISS models for the ARM7. The ISS model ran a simple program at about 22,000 instructions per second on a Ultra SPARC. The CISS model ran this same program at about 150,000 instructions per second on a Ultra SPARC.

The performance of these ISS models generated from nML is similar to the performance of vendor supplied ISS models for commercial DSP processors and microprocessors. The ML generator appears to produce faster ISS models than the Markus generator, perhaps because the Markus ISS models allocates data structures to describe the decoded instructions. Neither of these model generators was really tuned for performance though, and it is clear that better performance could be obtained from each approach.

The CISS models, as expected, are almost an order of magnitude faster than the ISS models. There are a number of obvious things that could be done to improve the performance of the Markus CISS models, such as inlining the actions rather than making function calls to them. With some additional work we expect the Markus CISS models could be made considerably faster.

7. Conclusions

With some enhancements to nML or a similar language, it looks feasible to develop retargetable instruction set simulators for commercial DSP processors and microprocessors.The areas that we think nML needs to be improved are:

- Delayed assignments.
- Interrupts
- Flexible PC updates
- Improved arithmetic support.
- Local variables.

It is currently difficult to encode behavior like delayed register assignments or delayed branches (a delayed assignment to the PC) in nML. You have to create temporary registers to save the delayed assignments, and your top level instruction has to make the assignments the correct number of clock cycles later. Modeling delayed assignment this way makes it more difficult to use the same behavior for code generation. Adding an explicit delayed assignment to nML moves this complexity out of the behavior.

Likewise, interrupts are not part of the current nML language. A test for interrupts can be added to the top level instruction, but again this makes interrupt handling part of every instruction's

behavior, which makes it more difficult to use this behavior for code generation.

The nML specification is vague about how the PC gets updated. To properly handle multi-word instructions, the PC update needs to be in each instruction, but for repeat instruction (zero overhead loops) you want to execute an instruction some number of times without updating the PC. It is possible to work around this in the behavior, but this again makes the behavior less usable for code generation.

In nML registers or memories are declared as being signed or unsigned, but whether the data in a register is treated as signed or unsigned usually depends on the instruction, not the register. Processors also have carry or overflow bits that must be set. The current nML arithmetic instruction could be improved to make it easier to set these flags, and make it clearer that these are status flags.

Finally, nML does not allow local variables or storage locations in behaviors. All temporary locations must be declared as global registers or memories. The nML behavior descriptions would be easier to understand and maintain if you could declare local variables.

Some people doing Hardware/Software Codesign think they need cycle accurate or pin accurate processor models, although only a few commercial DSP processors currently have such models. Straightforward extensions of nML would support only the simplest pipelines. While many DSP processors have simpler pipelines than microprocessors, nML still falls far short of supporting cycle accurate models, and nML was never intended to address pin accurate models.

While the assembler and disassembler we developed was very simple minded, it is clear that nML could be the bulk of the specification for a real retargetable assembler/disassembler. Some additional information is clearly required to handle labels and symbols and to specify what the object output should look like.

Both TUB and IMEC have used nML for retargetable code generation by using the behavior as a pattern to match against a program behavior. We looked at doing this, but for commercial DSP processors the instruction behavior contains many details that need to be filtered out to get an essential behavior that can be matched against a program. A general algorithm for filtering nML behavior descriptions does not seem possible to us, although one that would do most of the work looks feasible. A more general solution might be to add an essential behavior as another attribute in the nML grammar. It is also clear that good retargetable code generators for current commercial DSP processors is a much more complex problem then retargetable instruction set simulators, and may only be solved by enhancements in the architecture of commercial DSP processors.

Code generation for commercial microprocessors like the ARM7 is much simpler than for DSP processors. A code generator for these kinds of processor from nML seems more feasible, although a custom code generator might still be more efficient.

Acknowledgments

We would like to thank Markus Freericks of the Technical University of Berlin for permission to use the nML language.

References

1. P. Paulin, C. Liem, T. May, S. Sutarwala, "FlexWare: A Flexible Firmware Development Environment for Embedded Systems", in P. Marwedel, G. Goossens, *Code Generation for Embedded Processors*, Kluwer, 1995, pp. 67-84.
2. S. Sutarwala, P. Paulin, Y Kumar, "Insulin: An Instruction Set Simulation Environment", *Proc. of CHDL*, Ottawa, Canada, April 1993, pp. 355-362
3. A. Fauth, J. Van Praet, M. Freericks, "Describing Instruction Set Processors Using nML", *Proc. European Design and Test Conf.*, Paris (France), March 1995, pp. 503-507.
4. M. Freericks, "The nML Machine Description Formalism", Tech. Rep. 1991/15, TU Berlin, Fachbereich Informatik, Berlin, 1991.
5. F. Lohr, A. Fauth, M. Freericks, "SIGH/SIM - an Environment for Retargetable Instruction Set Simulation", Tech. Rep. 1993/43, TU Berlin, Fachbereich Informatik, Berlin, 1993.
6. A. Fauth, A. Knoll, "Automated Generation of DSP Program Development Tools", in *Proc. IEEE ICASSP-93*, May 1993.
7. D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, G. Goossens, "CHESS: Retargetable Code Generation for Embedded DSP Processors", in P. Marwedel, G. Goossens, *Code Generation for Embedded Processors*, Kluwer, 1995, pp. 85-102.
8. V. Zivojnovic, H. Meyr, "Compiled HW/SW Co-Simulation", in Proc. ACM/IEEE Design Automation Conference, pp. 690-695, 1996.