Processor Modeling for Hardware Software Codesign

V. Rajesh
gvrajesh@hotmail.com

Rajat Moona
moona@iitk.ernet.in

Cadence Research Center
Indian Institute of Technology Kanpur
Department of Computer Science and Engineering
Kanpur, India

Abstract

In hardware - software codesign paradigm often a performance estimation of the system is needed for hardware - software partitioning. The tremendous growth of application specific embedded systems necessitate high level system design tools for rapid prototyping. This work involves design of a language Sim-nML *which will be the base for a high level system design environment. The language is simple, elegant and powerful enough to express the behavior of the processor at instruction level. This language is used as the base for a whole set of tools such as assembler, disassembler and simulator generator. As a part of this work, we implemented an instruction set simulator generator which takes* Sim-nML *description of the processor as input and produces* C++ *code for performance simulator. We envisage the use of the generated simulator for cycle based analysis of the processor and for performance estimation of the system. This work is primarily an extension of nML[2] language.*

1. Introduction

With the complexity of the processors growing rapidly and with increasing number of special purpose processors being developed, system designers require modeling tools with high level of abstraction for hardware - software partitioning. In addition, they want to simulate and analyze the performance impact due to various hardware - software codesign trade-offs. Toward this end, designers are using languages such as C, VHDL and Verilog. The difficulty in using these languages is that the level of abstraction provided by these languages doesn't allow rapid prototyping. Moreover, it is convenient to have one specification and use it for various applications such as simulation, assembler and disassembler generation and compiler back-end generation. Today, there are a class of special purpose machine description languages which are developed keeping in mind a particular application. GCC's .md format[7] is such a language developed for the purpose of compiler back end generation.

nML[2] is an extensible formalism developed for specifying the processor at higher level of abstraction. nML formalism models the instruction set architecture of a machine but does not provide any control flow constructs. In this paper, we have attempted to capture control flow and inter-instruction dependencies by extending nML formalism. We extended nML formalism by adding new pre-defined attribute for specifying the control flow.

This paper is not intended to be complete reference for *Sim-nML*, but it highlights the extension to nML. In Section 2, we discuss the salient features of nML. Section 3 describes the resource usage model. Section 4 describes *Sim-nML* and focuses on the extensions to nML. Section 5 gives some example models. Section 6 discusses some of the implementation issues of simulator generator. Some of the related works are reported in [1], [3],[4], [5], [6].

2. nML

nML [2] is a formalism targeted for describing arbitrary single processor computer architecture. nML works at instruction set level and hides the implementation details. In nML, the instruction set is enumerated by an *attribute grammar* [1] . The semantic action of any instruction is composed of fragments that are distributed over the whole grammar tree. i.e. the common behavior of a class of instructions is captured at the top level of the tree and the specialized behavior of sub-classes are captured in the subsequent lower levels.

[1] An attribute grammar is a context free grammar in which for each non-terminal a fixed set of attributes and for each production a set of semantic rule is given. In nML grammar, all non-terminals have to have derivations. So, we don't differentiate between productions and non-terminals.

2.1. nML Grammar

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

```
op n0 = n1 | n2 | n3 | ...
```

and *and-rule* which looks like,

```
op n0 ( p1 : t1, p2 : t2, ... )
a1 = e1  a2 = e2 ...
```

where each `ni` is a non-terminal and each `ti` is a token. Each `ai` is an attribute name and `ei` their respective definition. The `pi` are names of the parameters used in the attribute definitions.

nML grammar pre-defines three attributes namely *syntax*, *image* and *action*. The *syntax* attribute describes the textual syntax of the instruction. The *image* attribute describes the binary coding of the instruction and *action* attribute describes the semantics of an instruction.

```
type addr = card ( 32 )
type byte = card ( 8  )
mem AC [ 1, byte ]
mem PC [ 1, addr ]
mem temp [ 1, byte ]
op instruction(x:binaction, data:byte)
syntax = format("%s %d",x.syntax,data)
image=format("11%6b %8b",x.image,data)
action = {
           PC = PC + 2;
           tmp = data;
           x.action;
}
op binaction = plus | multiply
op plus ( )
syntax = "add"
image  = "000000"
action = { AC = AC + tmp; }
op multiply ( )
syntax = "mult"
image  = "000001"
action = { AC = AC * tmp; }
```

Example 1 : nML Description of a Simple Processor

The nML grammar in Example 1 describes a simple machine with two instruction. The add instruction which adds the data to the accumulator `AC` and the multiply instruction which multiplies the data to the accumulator `AC`. `PC` has a special semantics and points to the next-to-be-executed instruction.

In most of the processors addressing modes and instructions are orthogonal to each other. Therefore, describing an instruction with different addressing modes explode the size of the description. So, nML separates addressing mode descriptions. nML formalism helps to describe the processor concisely and precisely. However, nML lacks control flow constructs and cannot depict the inter-instruction dependencies. Therefore, it is not possible to use nML for performance estimation.

3. Resource Usage Model

At the time of hardware - software partitioning, designers like to study the performance impact due to their design decision. This is an active phase and the design is changed frequently. This necessitates a way to depict the design in easy way and estimate the performance with the help of simulation. We extend nML, by abstracting out the control flow with the help of *resource usage model*.

The resource usage model is based on the assumption that at any instant, an instruction on execution, holds a set of resources and does some action. The resources held by the instruction and the action taken change progressively.

In resource usage model, a resource is an abstraction of a piece of hardware such as registers, ALUs, functional blocks, etc. for which instructions contend and control flow is nothing but a way of resolving conflicts due to contention. When two instructions are waiting simultaneously for a single resource, the conflict will be resolved by FIFO order i.e. the instruction that entered the pipe earlier will be allocated with the resource. This simple model is powerful enough to model pipelines.

For example, consider our simple processor described in the Example 1. We model the processor with three pipeline stages, viz., `fetch_unit`, `execution_unit` and `retire_unit`. The primary extension made to incorporate resource usage model in nML is, the addition of a new pre-defined attribute *uses*. The *uses* attribute describes the resource usage model and the action of an instruction. The *Sim-nML* specification of the processor is given in Example 2. It specifies that all instructions first use the `fetch_unit` for one unit of time. The instructions then use the `execution_unit` for the duration dependent on the type of instruction and then the `retire_unit` for one unit of time. The `add` instruction uses the `execution_unit` for one time unit whereas the multiply instruction uses the `execution_unit` for three time units. The token *action* at the end of *uses* specifies that after the specified resources are used for the specified duration, the function specified in *action* attribute is performed. The *resources* declaration is used to declare the functional blocks such as the `fetch_unit`, the `execution_unit` and the `retire_unit`. The description of actual functionality of these resources is not in the scope of *Sim-nML* formalism and is hidden.

```
resource fetch_unit,execution_unit,
         retire_unit
reg AC [ 1, card( 8 )  ]
reg PC [ 1, card( 32 ) ]
reg temp [ 1, card ( 8 ) ]
op plus ( )
syntax = "add"
image  = "000000"
action = { AC = AC + tmp; }
uses   = execution_unit #1
op multiply ( )
syntax = "mult"
image  = "000001"
action = { AC = AC * tmp; }
uses   = execution_unit #3
op binaction = plus | multiply
op instruction(x:binaction, data:card(8))
syntax = format("%s %d",x.syntax,data)
image=format("11%6b %8b",x.image,data)
action = { tmp = data; x.action; }
preact = { PC = PC + 2; }
uses = fetch_unit : preact&#1,x.uses,
             retire_unit #1 : action
```

Example 2 : Sim-nML Description of the Simple Processor

The unit of time can be thought-of as machine clock cycle although it is not imposed by the *Sim-nML*. But, sub-unit timings are not allowed. In a nut-shell, if unit of time is same as machine clock cycles then we can estimate the number of clock cycles taken by a program to complete.

4. Sim-nML

4.1. Uses Attribute

In resource usage model, a set of resources are acquired by an instruction and the resources are held till the next set of resources are available. Therefore, a specification of sequence of resources used by an instruction results in an abstract specification of the control flow. For example, `fetch_unit, execution_unit, retire_unit` means that at first `fetch_unit` is acquired if it is free and it is held till the `execution_unit` becomes free. When `execution_unit` becomes free, `execution_unit` is acquired and `fetch_unit` is freed. The same procedure is repeated in acquiring `retire_unit`.

After the pipeline flush, when the first instruction enters the pipeline, all the resources are immediately available. Therefore, to control the flow of instructions, it is also necessary to specify the minimum amount of time for which each resource is held. For example, `fetch_unit #1, execution_unit #1, retire_unit #1` means that at first the `fetch_unit` is acquired. Although, `execution_unit` is available immediately, the instruction waits in `fetch_unit` for one time unit before acquiring the `execution_unit`. Then, it holds `execution_unit` for one time unit. In the similar manner, before completion, it acquires `retire_unit` and holds it for one time unit.

In some cases, it is not possible to specify the resource hold times statically. The instruction has to wait till a condition becomes true. For example, to model in-order-retirement of instructions, an instruction before completion should wait till the completion of all the instructions that precede it. These kind of models can be described by the specification of conditional waits in *uses* attribute. For example, `execution_unit, retire_reg == present_instruction_id, retire_unit` means that `execution_unit` is acquired and held till the condition `retire_reg == present_instruction_id` becomes true and `retire_unit` is free. Then `retire_unit` is acquired and `execution_unit` is freed.

Modern processors has multiple execution units to increase the performance. Specification of these processor models, is facilitated by the *or* construct of *uses* attribute. For example, `fetch_unit, execution_unit1 | execution_unit2, retire_unit` means that the `fetch_unit` is acquired at first and held till either `execution_unit1` or `execution_unit2` becomes free. Then the free execution unit is acquired. If both execution units are free, then `execution_unit1` is acquired.

In some cases, the resources used depends on a condition. For example, time for memory access depends on cache hit and cache miss. This can be specified by the *if* construct of *uses* attribute. For example, `fetch_unit, memory_unit, if cache_miss == 1 then #10 else #2, retire_unit` means that `memory_unit` is held for atleast 10 units of time in case of `cache_miss == 1` is true and atleast 2 units of time in case of `cache_miss == 1` is false.

The specification of the action that has to be carried out at any instant can be specified by ':' operator. For example, `fetch_unit : preact` means that once `fetch_unit` is acquired, the action specified in `preact` attribute is executed.

4.2. Declarations

- *Resources* such as `fetch_unit`, `execution_unit` can be declared with `resource` key word. For example,

  ```
  resource fetch_unit, execution_unit
  ```

- *Exceptions* are useful to capture asynchronous behaviors such as branch error, interrupts etc. Exceptions can be declared with `exception` keyword. For example,

  ```
  exception overflow, branch_error
  ```

- *Registers* are same as any memory except that they are assumed to be inside the processor. Moreover, a sequence of memory locations specified by `mem` declaration is considered to be one resource whereas a sequence of memory locations specified by `reg` declaration are considered to be different resources. For example,

  ```
  reg R [ 32, card ( 64 ) ]
  ```

  in the above declaration R [1] is a different resource than R [2] i.e. R [1] can be acquired and freed independent of R [2].

- *Instruction Identifier* type is useful to uniquely identify an active instruction. This is helpful to streamline the retirement of instructions in a superscalar processor. `MAX_INSTR_COUNT` constant is related with this type. This constant specifies the number of instructions that can be present inside the processor at any instant. This constant should be appropriately specified by the user.

4.3. Predefined Canonical Functions

Canonical functions[2] are those functions whose semantics are known only to the entity that reads the description.In our instruction set simulator generator, other than few pre-defined canonical functions, all other canonical functions are mapped directly to `C++` functions.

Handling exceptional conditions is complicated because it is likely that the exceptions are handled differently at different machine states. Therefore, it is necessary to provide mechanism to change the handler at any instant. In our model, we propose the following canonical functions related to the exceptions.

- `sethandler` - used to set a new handler for a particular exception
- `ignore` - used to ignore a particular exception
- `raise` - used to broadcast the occurance of an exception to all instructions in execution.
- `abort` - used to abort an instruction on execution and free the resources held by the instruction immediately.

In addition to the exception related canonical functions we propose a canonical function to uniquely identify an instruction.

- `instid` - returns a key which uniquely identifies the calling instruction. The return value is of instruction identifier type.

5. Specification with Sim-nML

5.1. Specification of Memory Hierarchy

To avoid the severe bottleneck due to memory access and to increase the performance, modern machines use hierarchical memory. The precision of handling memory hierarchies depends on the specification. Below we model a memory system with a `data_cache` whose hit ratio is presumed to be 95%. The time for cache access is assumed to be two units of time and for that of `main_memory` is assumed to be ten units of time.

```
mem data_cache [ 1024, word ]
uses = #2
mem main_memory [ 2**16, word ]
uses = #10
mode IND( R:Address_Register )=M[ R ]
uses   = if "drand48"() < 0.95
            then data_cache.uses
            else main_memory.uses
syntax = format ( "(A%3b)", R )
image  = format ( "#3b", R )
```

In the above example, `"drand48"` denotes that it is a canonical function call. If we want more precise model in which we actually check whether the data is present in the cache. This we can do by writing a `C++` function which keeps track of the contents of the cache.

```
mode IND( R : Address_Register )=M[ R ]
uses   = if "is_hit"( data_cache, R )
            then data_cache.uses
            else main_memory.uses
syntax = format ( "(A%3b)", R )
image  = format ( "#3b", R )
```

The `is_hit` function uses the address argument to implement the cache replacement policy such as least recently used or less frequently used etc. The above example shows the ease with which the semantics of description can be increased. The implementation details are discussed in next section.

5.2. Specification of Branch Prediction

. The processor modeled below is a very simple processor with only two instructions viz. `plus_instr` and `branch_instr`. The processor contains an accumulator and a zero flag register `Z`, which is set only if the result

of computation is zero. The *Sim-nML* code, describes the plus instruction which adds an immediate value to the accumulator. In case of branch misprediction, `branch_error` exception is fired. At the time of the exception if an instruction from mispredicted path is at `fetch_unit`, then the instruction is aborted. If the instruction from mispredicted path is down below the pipe at the time of `branch_error`, then the instruction continues execution, but it does not change any registers. This is done by setting the value of `reorder_buffer` associated with the plus instruction to 255.

```
const MAX_INSTR_COUNT = 8
resource fetch, execution1,
        execution2, retire,
        branch_unit, halt_unit
exception branch_error
type addr = card ( 32 )
type byte = card ( 8  )
type bit  = card ( 1  )
reg AC [ 1, byte ]
reg PC [ 1, addr ]
reg Z  [ 1, bit  ]
mem speculated [ 1, bit ]
mem oldpc [ 1, addr ]
mem retire_reg [ 1, instid_type ]
mem reorder_buffer[ 8 , byte ]
op initial ( )
action = { PC = 0; }
op plus_instr ( data : byte )
syntax = format ( "add %d", data )
image  = format( "11000000%24b",data )
action = {
   AC = AC + data;
   if AC == 0
     then Z = 1; else Z = 0; endif;
}
uses = fetch #1,execution1 #1 |
execution2 #1,retire_reg=="instid"(),
retire & AC & Z & #1,
if reorder_buffer[retire_reg] != 255
     then retire : action
op branch_instr ( target : addr )
syntax=format( "brnz %24b" , target )
image=format( "01000000%24b",target )
preact = {
 bran_inst_id = "instid"();
 reorder_buffer[bran_inst_id] = 255;
}
action = {
 if Z == 0
    then PC = target; endif;
 speculated = 0;
}
always_taken = {
 oldpc = PC;
 PC = target;
 speculated = 1;
}
restore = { bran_inst_id = 88888; }
check_pred = {
 if Z == 1 then
     PC = oldpc;
     "raise" ( branch_error );
 endif;
}
uses=if "is_blocked" ( Z )
      then  fetch : always_taken & #1
      else  fetch : action & #1,
      branch_unit : preact,
if speculated==1 then Z : check_pred,
 retire_reg==bran_inst_id,
 retire #1 : restore

op instr = branch_instr | plus_instr

op instruction ( x : instr )
syntax = x.syntax
image  = format ( "%s", x.image )
action = { x.action; }
handler= {
 if  reorder_buffer["instid"()]==
         bran_inst_id then
  reorder_buffer [ "instid"() ] = 255;
 endif;
}
preact = {
 PC = PC + 4;
 reorder_buffer["instid"()] =
                   bran_inst_id;
 "sethandler"( branch_error,handler );
}
reorder ={ retire_reg=retire_reg+ 1; }
uses = fetch : preact , x.uses,
       retire : reorder
```

Example 3 : A Processor Model with Branch Prediction

The above code describe the branch instruction. The processor follows `always taken` policy for branch prediction. In case of speculation, the value of `Z` is checked once it is evaluated. If prediction is found to be wrong then a `branch_error` is signaled which is caught by all instructions. On catching the `branch_error`, the instructions that follow branch instruction set their associated value in `reorder_buffer` to 255. Since this model allows only one branch instruction at any instant, nested branch errors are not taken care of.

6. Implementation of Simulator Generator

The implementation of complete simulation environment involves two phases. The first phase is the implementation of instruction set simulator generator for *Sim-nML*. The simulator generator outputs a number of C++ class templates which describe the processor at instruction level. A part of code generated for Example 2 is shown below. The second phase is the implementation of a generic simulator library which is to be linked with the classes produced by the simulator generator to give a complete performance simulator.

```
template < class T0 >
class instruction {
private:
image_type  image;
T0  x;
byte        data;
public:
string syntax ( ) { ... }
string image ( )  { ... }
void action ( )
  { tmp = data; x.action ( ); }
void preact ( ) { PC = PC + 2; }
void uses(resource_list& rlist){ ... }
instruction(image_type img=0,int s=0)
    : image ( img ), x ( img, s + 2 )
{
  ...
  data=extract_bits(image,8,16);
}
};
```

The sequence of resources required by an instruction is maintained as a list of records. Each record in this list gives the resources used by an instruction at a particular instant. The states of the partially executed instructions are captured by storing their present list pointers.

To correctly implement the *Sim-nML* semantics that each instruction holds a set of resources at any instant, the generated simulator acquires next set of resources before freeing the old resources. Resource conflicts are resolved by first-instruction-first-served policy. If different policy is to be followed then the designer has to specify it by using conditional waits. In Example 3, we depict different policy by using `retire_reg`.

The simulator is implemented with the help of an event manager which keeps track of the blocked events. Whenever possible, the simulator reads next instruction from the address pointed by `PC`. Then it creates an instruction object of the class whose image best matches with the newly read instruction image. The reading of next instruction is stopped when the first resource request is not satisfiable.

7. Results

As part of this work, we developed performance models for two processors using *Sim-nML*. The first one is a performance model of a hypothetical superscalar processor employing branch prediction to reduce branch penalties. A part of *Sim-nML* specification for this model is shown in Example 3. The second one is a performance model of a DEC Alpha 211064 processor. We have modeled only a partial set of instructions. This set is sufficient to encode applications such as bubble sort. The performance simulator for these models were generated and tested. The generated simulator runs at a speed of 3,000 instructions per second.

8. Conclusion

In this paper we have proposed an environment for hardware software codesign based on a simple language *Sim-nML*. We have shown *Sim-nML* to be powerful enough to describe the complex issues such as branch prediction, hierarchical memory etc. We have implemented a simulator generator which takes *Sim-nML* description as input and produces code for performance simulator. The work for implementation of complete simulation environment is in progress. We envisage the use of this language in various other applications in hardware software codesign domain.

The simulator generator software package can be obtained from authors.

References

[1] M. R. Barbacci. Instruction Set Processor Specifications (ISPS) : The Notation and Its Application. *IEEE Trans on Computers*, Vol. C-30(No.1), Jan 1981.

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

[3] R. K. Gupta and S. Y. Liao. Using a Programming Language for Digital System Design. *IEEE Design & Test of Computers*, Apr-Jun 1997.

[4] A. Poursepanj. The PowerPC Performance Modeling Methodology. *Comm. ACM*, pages 47–55, Jan 1994.

[5] M. Reilly and J. Edmondson. Performance Simulation of an Alpha Microprocessor. *IEEE Computer*, 31(5):50–58, May 1998.

[6] M. Rosenblum and E. B. et al. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Trans on Modeling and Computer Simulation*, Vol. 7(No. 1), Jan 1997.

[7] R. M. Stallman. *Using and Porting GNU CC*.