Microprocessor modeling for Retargetable Timing Simulation

A Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Technology

by
Anand Shukla
Arvind Giridhar Sarraf

under the guidance of
Dr. Rajat Moona

to the
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
May, 2001

Certificate

Certified that the work contained in the report entitled "*Microprocessor modeling for Retargetable Timing Simulation*", by *Anand Shukla* and *Arvind Giridhar Sarraf*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Rajat Moona

May, 2001

Acknowledgements

Our heartfelt thanks to Dr Rajat Moona, we couldn't have had a more patient and helpful guide. Also to Dr Deepak Gupta and Dr Sanjeev K Aggarwal, who lived with our (often half-baked) presentations in weekly meetings and made ample suggestions. To Rajiv A. for ever being so cooperative, to Souvik Basu and Soubhik Bhattacharya, the other members in the Cadence Research lab at IITK, for always being available and generously lending us papers and books.

Contents

Chapter 1

Introduction

Traditionally, microprocessor design decisions have been taken by considering the decision alternatives at Hardware description language (HDL) level, and using these descriptions to analyze designs for performance metrics such as timing performance, cost, power consumption, etc. Improving chip fabrication technology has allowed packing more logic on chips-and has been well subsumed by increasingly complex processors and systems. Success of a new system today depends a lot on demand to market time, and rapid availability of tools such as assemblers, compilers, debuggers and disassemblers.

HDLs, however, lack in this respect. These languages are designed to aid in (and almost are a necessary step of) fabrication-the descriptions are too low level to allow easy exploration of design options, and do not provide much support for tool generation.

Sim-nML project[1] was undertaken to cater to the above need. Based on an architecture description language nML [3], it allows descriptions of a target at a level customized to instruction set processors. Tools such as functional simulators, assemblers, disassemblers can automatically be generated using the specifications in this language. Our goal in this project was to *add features to the existing language that would allow timing simulation of the programs for the target processor*-in our case, an estimate of cycle count of the execution time. Several architectural details determine execution time. Our emphasis has been to obtain descriptions that are

easy to write and understand and efficient to simulate, without comprosing on the cycle count accuracy.

In this work, we propose an extension based on *Resource-usage model*. This framework specifies how hardware features that affect the execution time, and how instructions that use them are to be described. Implicit in the model is how it is to be used for timing simulation of a program.

Our emphasis has been to base our fundamental abstractions on what actually happens in real hardware. We have tried these constructs on a range of processors-from simple non-pipelined uniscalar ones to superscalar processors using advanced features such as out of order speculative execution, register renaming, branch prediction, and complex memory sub-system interface architectures. We find the intuitive nature of this formalism significantly easing descriptions. At the same time, we believe that timing simulation using this approach would be significantly easier to describe and faster than those using low-level HDLs, and at the same time a significant saving over the complex coding required to code them manually in a general purpose language.

The rest of this report is organized as follows: In chapter 2, we give a brief overview of our perspective of the problem, and our approach to the same. The remaining chapters try to explain (and hopefully, present a good enough case for) it. Chapter 3 introduces the fundamental abstractions called resources, and chapter 4 specifies how timing-critical aspects of a processor and instruction set architecture are to be captured using them. We also present how this description can be used for timing simulation. We drive home our point remarking the expressive power by describing fairly exhaustive range of processors in chapter 5. Finally, we suggest some future directions of work (chapter 6).

Chapter 2

Overview of our approach

The fundamental problem of timing simulation is this: Given a processor description μ, description of components external to it (such as memory architecture), and a program P for that processor we need to find the time that program takes on μ. In our case, it involved working out both-*how the processor is to be described, and how that description is to be used.*

Here, we try to rephrase the problem in our Sim-nML setup. In Sim-nML, processor component specifications are captured using register, memory and resource declarations. Instruction specific features are captured using its attributes-*syntax* and *image* for assembly syntax, *action* attribute for semantic action of the instruction and *uses* for the timing critical aspects of the instruction execution. We here focus on resource declaration and usage attribute for instructions.

An instruction execution can be thought of as using different processor components at different stages of execution-the exact usage depending not just on instruction type, but also on the current processor state. Each such component can be thought of as a resource and different forms of usages can be thought of as different method invocations on them. Multiple instructions may contend for the same resource. *The actual time for which processor units are used, and stalls waiting for them-both of which can be captured in the usage, the first by resource holding times, and the second by contention for them determine the program execution time.*

We begin by classifying the resource types on the basis of kind of methods that

are invoked on them. We emphasize here the point that this *classification is not based upon the exact functionality of the resource, but the way these methods affect the availability of the resources to other methods by other instructions.*

An instruction usage is composed of individual resource methods-we later give the structure in which they can be composed. Our main premise here has been to capture a wide-range of processor designs. In the process-we also make an interesting discovery-the restrictions on reasonableness of the processor also led to a simulation mechanism for this description. This we do by mapping instruction usages to what are called as *usage graphs.*

At this point, we must also make an honest confession. Though we have listed the examples towards the end of this report, our line of action has been pretty much the other way-we based our formalism (resource types, methods and the corresponding semantics) on these examples.

Chapter 3

The resources

This work hinges on, what we broadly call as a "resource usage" model. In this model, the limited pool of "resources" become a contention point for the users of the resources, in this case "instructions". Intuitively, if a set of resources, ordered in some form, are to be operated upon by multiple contenders (who themselves have some ordering), it would be some time (call it duration t) before all contenders have received what they wanted (after instructions acquire a resource, release it so that others can acquire it, go on to acquire other resource, and so on). If the ordering of resources and of instructions is well defined, and the "rules" of resource allocation are clearly and unambiguously laid out, t *would* be unique and can be figured out (our goal). The challenge is then to figure out this t: which in turn involves identifying the resources, and the ways to specify the acquisition rules.

The latter is captured by attaching some *methods* to resources, and clearly laying down scheduling rules associated with resources . These methods can be *blocking* or *non-blocking*. A non-blocking method is one for which an instruction cannot wait–the processor must make sure that the relevant resource is available when that operation is invoked. A blocking resource method blocks-usually, till the condition it is waiting for is satisfied-at which it unblocks, and the resource state is appropriately modified.

We classify resources into three categories: buffers, registers and static resources. The purpose behind these resources, their properties and the operations on them

are given below.

3.1 Static resources

A resource like an ALU, or some other execution unit-which is used for a known period of time by an instruction irrespective of availability of other units is classed under this category. The declaration of a static resource is as follows:

StaticResource *R1, R2[n]* //n instances of R2, 1 instance of R1

A static resource can have multiple instances. If a static resource R has n instances then in uses R denotes any of the n instances while R[ ] stands for all the n instances of R.

If a static resource R is to be used by an instruction for t cycles, then the corresponding invocation is denoted by 'R#t'. The duration t could be a constant, or a canonical function call that returns some integer value.

A special type of static resource, called null resource is assumed to have infinite instances. A consequence of this is that there is never a contention for this resource. Instead of using this resource as null#k, it is usually denoted as just #k, which indicates holding this resource for k cycles-this is defined as the *time* of the resource method.

3.2 Register

The register abstraction stands for registers in processors. Their declaration is as follows:

Register *Reg[n]* //n instances of R2, 1 instance of R1

The ith register of the above declared register file would be used as Reg[i].*operation,* where *operation* can be one of the following:

1. itR (intention to read)

2. Read
3. itW (intention to write: blocking or non-blocking)
4. forward
5. write or commit

These operations make it possible to capture various issues associated with the data: apart from read and write from the register file, data hazards (RAW, WAR, WAW) can also be handled. This is possible because of the semantics associated with these operations, which is described below.The sample data structure (see section 3.2.3) for the implementation of these operations may further help in understanding their semantics.

We define an ordering between different intentions to the same register. A register intention to read (itR) or intention to write (itW) is said to be before another intention if

- the first intention was issued before the second intention, or
- both were issued in the same clock cycle, and the first instruction is earlier in the program order than the second.

Since the above order is a total order, it implicitly defines a immediately relation among a given instruction set.

3.2.1 Semantics of itR, read and forward:

A instruction I_1 issuing a read to a register blocks till the instruction I_2which has issued an itW immediately preceding I_1's itR does a forward.

3.2.2 Semantics of itW and write

- Blocking itW
  A blocking itW blocks till all the writes corresponding to all previously issued itWs to the same register have completed, or their corresponding instructions have completed.

- Non-blocking itW

In either case, a write blocks till all the writes and reads corresponding to all previously issued itWs to the same register have completed, or their corresponding versions have completed.

R.itW by an instruction indicates that the instruction would *subsequently* be writing into the register R: the actual update is made by R.write. Now, there are two versions of itW operation: blocking and non-blocking. Two different versions are required because some processors may not allow more than one instruction from going beyond the decode (or such similar) unit if they write into the same register (WAW), whereas some other processors may not have this restriction (such as the ones using register renaming). The blocking version captures the former, while non-blocking version captures the latter.

3.2.3 A sample data structure for register operations

This sample data structure is intended to clarify the semantics of the register operations. Pictorially, it looks like in figure 1. In the diagram, $L_{i_1}$and $L_{i_2}$are linked lists, while A is an array of registers (register file R). P_{i_1}points to L_{i_1}, P_{i_2}points to an instruction I_1, P_{i_3}points to the list L_{i_2}, n_1, n_2... are nodes in the list L_{i_1}. List $L_{i_1}$is called as itW-list, L_{i_2} is itR-list. The itR-list is a list of instructions. There are similar itR-list and itW-list with all other elements of A as well. Consider the following operations on Reg[i] by an instruction I:

1. Reg[i].itW (non-blocking): This creates a new node, pointing to instruction I, and adds this node at the tail of L_{i_1}.

2. Reg[i].itW (blocking): This blocks if $L_{i_1}$is not empty. If $L_{i_1}$is empty, this operation succeeds, a new node pointing to the instruction I is created and it is attached at the head of the (empty) list L_{i_1}.

3. Reg[i].itR: Here, I is added to the tail of itR-list of the last node of itW-list of Reg[i].

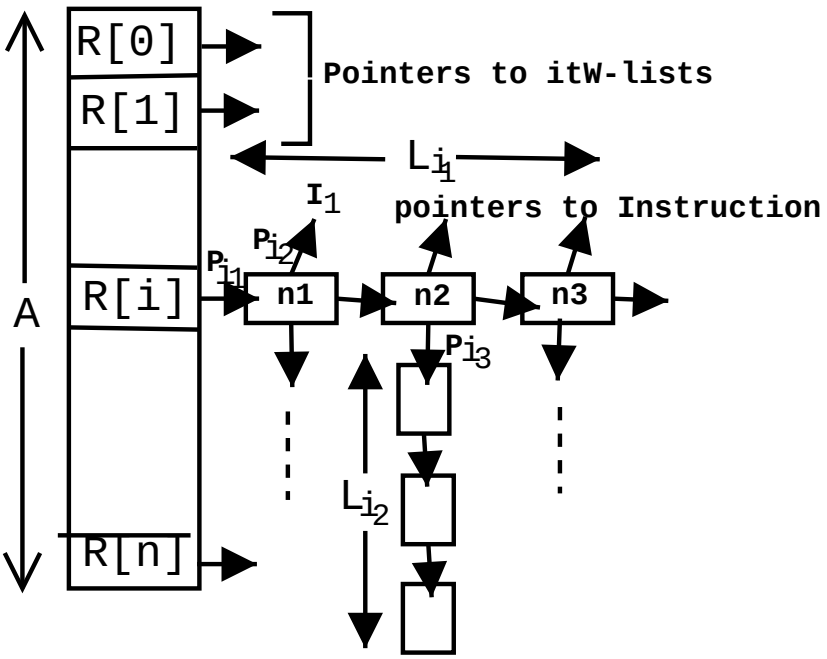

Figure 1: Data structure for implementing register operations

4. Reg[i].forward: First the node n in the itW-list of Reg[i] is located which has a pointer to I. Then in the itR list of node n, all instructions all flagged as read-available.

5. Reg[i].read: Here, I is located in the itR-lists associated with itW-list nodes of Reg[i]. (To speed up this search, the instruction data structure may contain a pointer to the itR-list). If its flag is set as "read-available", then operation succeeds, otherwise it blocks. If I can not be located in the itR-lists, then the operation succeeds.

6. Reg[i].write or Reg[i].commit: Here, a node n is located in the itW list of Reg[i]. If this node is the first node in itW list of Reg[i], then it is deleted and the operation succeeds. Otherwise it is blocked.

3.3 Buffers

This abstraction allows instructions to wait at different points in the pipeline. This may be necessary because of a variety of reasons, some of them are:

1. An instruction waiting for some resource to be available. An example of this is reservation stations associated with functional units, where instructions wait for that function unit or data.

2. Presence of units which enforce instruction reordering –usually to maintain sequential semantics. Reorder buffers (or write-back buffers, as the case may be) serve this purpose.

3. Mechanisms that allow for "speed mismatch" between different hardware components- such as memory subsystem and the processor. This compounded with the unpredictable operation delay (due to memory hierarchy) necessitates interfacing to them via buffers (typically load-store buffers).

4. The delay or wait an instruction may have to undergo may not be predictable *when a later instruction modifies the usage of the earlier instruction.* This may happen when different pipeline paths (taking different number of clock cycles) converge (in terms of instruction resource usage) to the same execution unit.

The above features seem to require support for execution-time determined delays. We capture these features in a resource abstraction called buffers. A buffer declaration looks like:

Buffer B: {slots = *n*; input = *InputOrder*; output = *OutputOrder*}

Here InputOrder can be of the form Static: R_1, R_2, ..., R_n. A buffer can have multiple incoming paths, the static ordering then defines the relative priority among those incoming paths. In the above description, R_1, R_2, ..., R_n are resources that feed into buffer B's input. Also, R_1has a higher priority than R_2, which has a higher priority than R_3, and so on. This means if multiple number of these units have

instructions waiting to enter into the buffer B, then inputs are received by B from these resources in order of their priority.

OutputOrder can be Inorder or Anyorder. This denotes the order in which instructions currently in the buffer are considered for dispatching them out of the buffer with respect to the order in which they acquire the buffer.

Buffers have two basic operations: acquire (denoted by $\lfloor B$) and release($\rfloor$). Each buffer keeps track of number of free slots. Acquiring a buffer reduces the number of free slots by 1, releasing it increases it. Both these methods are potentially blocking- an acquire can block in case the buffer has no free slots, and release if it violates the output scheduling policy of the buffer.

Special case of buffers: Latches Latches are buffers with just 1 slot (and so no output scheduling policy is needed). An input scheduling policy may be needed, however, as multiple pipeline paths may be leading into the same latch.

Latches occur in hardwares just before resource units (such as ALU, fetch unit etc) where instruction may wait until the resource unit is free. If there are no latches (or buffers) in a part of a pipeline, then instructions can not wait at any point in that part of the pipeline.

Chapter 4

Instruction Description in the Resource Framework

4.1 Usage Grammar: At the conceptual level

Conceptually, an instruction's resource description is of the following form (the exact grammar is given in Appendix B)

Usage: Φ
| *list*(*and*(AndUsage), Usage)
| *list*(*or*(condition, Usage, Usage), Usage)

AndUsage: ResourceMethod
| *ResourceMethod* & AndUsage

An *instruction usage* is a list of and and or usages.

An *and usage* is list of operations on resources each having equal time-it is schedulable at a given time if all the resource operations that are a part of it are schedulable. The *time* of an *and usage* is the time of any of the resource methods that constitute it.

An *or usage* will have two alternative usages along with a condition. The condition may be a an expression involving processor state, and canonical function calls-in

which case it is evaluated just before pushing the instruction into the pipeline to give *usage of that instruction instance*, and the path corresponding to the boolean value of the condition value must be schedulable. Or it may be a wildcard (*) which means that any of the two resource usages may be schedulable.

Sim-nML notations

We saw in Section 3.3 that a buffer B's acquire was represented as $[B$ and B's release as $]$ (the releases $]$ were mapped to corresponding acquires by the nesting). In Sim-nML grammar, an *and usage* is represented by corresponding methods connected by &, and an *or usage* by its usages separated by /.

and(u_1,u_2,...,u_n) = u_1& u_2& ...&u_n

or(*condition*, u_1,u_2) =((u_1)/(u_2)) *if* (*condition* = *)

or(*condition*, u_1,u_2) = *if* (*condition*) *then* u_1 *else* u_2 *endif*

4.2 Usage Graph

A convenient way to think of a usage is using a directed acyclic graph (DAG) representation called *Usage Graph*. Crudely speaking, each node of this graph corresponds to an and usage-containing the information about the corresponding resource methods. Each node u may have some successors denoted by *Succ(u)* (defining the directed edges of the graph):

1. If a node has no successor, it marks the end of the instruction execution.
2. If a node has no predecessor, it marks the start usage of the instruction execution.
3. If a node has more than 1 successors, it means that the instruction's usage can be satisfied in more than 1 way (something like an or usage).

4.2.1 Notation

Pictorially, we represent resource operations as shown (2, 3, 4)

Figure 2: Static resource

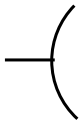

Figure 3: Buffer acquire

A node of the usage graph is represented by the corresponding operations at the same point.

We give some example descriptions in chapter 5.

Appendix C.1 gives the syntax directed translation for converting a usage specification to a Usage graph-for our future purposes, we will use these graphs.

4.2.2 Some simple properties required from usage graphs for valid instruction description

Not all Usage descriptions accepted by the Grammar would qualify as valid from the point of view of simulation. Here we list down some simple requirements of them:

- A buffer acquire must be matched by corresponding buffer release along all instruction usages that an instruction can take from the given start point, and vice versa.
- A resource acquire operation must be preceded by a wait point.
  For example,
  [FU, #1], DU#1, [EX, #1]
  will not qualify as a valid usage description.

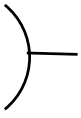

Figure 4: Buffer release

The rationale for this is given in section 4.3.

4.2.3 Definitions

A resource method is said to be *schedulable* if when issued at that time, it does not block.

A usage graph node (an *and node*) is said to be *schedulable* if all the resource operations in that and node can be initiated in that clock cycle. Its is said to have been scheduled if those resource operations have been performed to update the resource state.

A *stage* in instruction usage is a two-tuple of the form of the form (u, Succ(u)) or of the form (Pred(u), u) for some node u from Usage graph. Barring the fact that one of the elements of the above two-tuple is a set, and the other is not-the stage captures the all possible nodes that may have been scheduled last, and all possible future different next usages for the future.

A *wait point* of a usage graph is a stage of the (u, Succ(u)) where u has a buffer acquire, or of the form (Pred(v), v), where v contains a buffer release.

Intuitively, the first case corresponds to the fact that the instruction has just acquired a resource (in the and usage for node u), but hasn't yet "decided" which of the possibly several paths of execution (corresponding to different members of Succ(u)) it will take. The second case corresponds to the fact that an instruction has decided to release a buffer (in the usage for node v)-the usage pattern so far could have been along any path to v (hence, Pred(v)).

A path from (v_1,...v_n) from stage (S1, S2) to stage (T1, T2) is said to be *waitpoint free* iff

1. for no $1<i\leq n$, (Pred(v_i), v_i) is a waitpoint
2. for no $1\leq i<n$, (v_i, Succ(v_i)) is a waitpoint

Intuitively, a wait point free path from one stage to another represents one particular usage in middle of which an instruction can't block (no wait points), and hence-when that path has been decided to be taken, it must be ensured that all of the

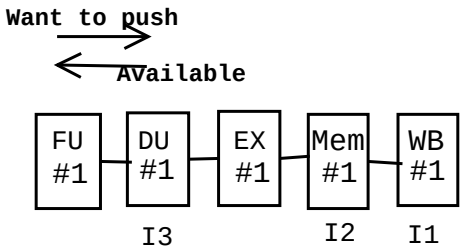

Figure 5:

operations can be performed at corresponding time. This idea is captured in the following definition.

A path from an edge stage (S_1, S_2) to (T_1, T_2) is said to be *schedulable at time t along a wait-point free path* v_1, v_2,... v_n, if either $v_1 = S_2$, or $v_1 \in S_2$, and either $v_1 = T_1$or $v_1 \in T_1$, and v_i is schedulable at time $t + \sum_{1 \leq j < i} time(v_j)$

An *instruction instance's current usage* is said to be at position at (u,Succ(u)), if its some path from any start usage point to (including) that corresponding to u has been scheduled, and none of the following successors of u have yet been scheduled. Similarly, an instruction instance's usage is said to be at (Pred(v), v) if some path from any start usage to (including) a node u in Pred(v) has been scheduled, but the usage of v has not yet been scheduled.

4.3 Timing Simulation

The idea behind our timing simulation algorithm is based on our following understanding of pipelined instruction execution: Lets say at some point between clock cycle boundaries t and t+1, the occupancy of pipeline stage is shown in figure 5.

Further, let us assume all instructions I_1,.. I_4 to be associated with relevant stages for one clock cycle. The instructions then actually use the corresponding stage's units for something less than 1 (say 1-δ) cycles. For the remaining part of the clock cycle, they check for the availability of the next stage. If available, the values are released on the lines to be latched at the next clock edge.

The decision regarding the availability of next stage propagate backwards. In the above example, the WB unit "knows" that I_1 will be leaving it at the end of

cycle. It signals the previous stage (Mem) regarding its availability-and decides to latch I_2 to WB at the next clock edge. This decision propagates backwards right through the pipeline in the backward direction.

Our timing simulator algorithm attempts to capture this ordering among the buffers, and then use it "schedule" instructions at each of them. The central theme when it comes to scheduling is to enforce the availability of operations needed to perform resource operations-an instruction waits only at wait points-just after acquiring a buffer, and just before releasing for a time just enough to ensure that what it actually issues any resource method in between, it does not block.

Part 1: To get an ordered list of resources in which they are to be considered

Construct a resource graph $G_R = (V_R, E_R)$, where V_R = Set of buffers = $\{b_1,...b_n\}$ and E_R = { (b_i, b_j): if there exists an instruction I_k whose usage graph has a path from a node containing b_i.acquire to b_j.acquire with no other buffer acquire in between}
(The way this graph can be generated from usage graphs of all instructions is given in Appendix C.2.)

The presence of an edge e = (b_i, b_j) in the edge set E_R denotes that scheduling of an instruction occupying b_i may depend upon the availability of b_j-hence, it must occur after b_j.

For each buffer b having static input schedule $c_1....c_m$, add an edge (c_{i+1}, c_i) for $0 \leq i < m$ in E_R.
(The justification here is that since c_i carries precedence over c_{i+1} when both contend for b, it looks *reasonable* to suggest that c_i must be scheduled before c_{i+1}.)

If the graph G_R constructed above has cycles, return invalid usage description.

Let B = [b_1'...b_n'] be the *any* reverse topological sort of G–we will consider the resources in this order.

Part 2: To simulate the instructions flow through the processor components

```
clock count = 0;

while (there are instructions ready to be executed or in the execution
pipeline)
{
      for (each buffer b ∈B considered in the same order as in B)
      {
            for (each instruction I that is in the second wait point at b
               and that has the next schedule time equal to the current time
                schedule (I);
            for (each instruction I that is in the first wait point at b
                and that have the next schedule time equal to the current
time)
                schedule(I);

             for (all instructions I that are yet to be pushed into the
execution pipeline)
             {
                  U = instructions instance I's usage using the current
machine stage;
                  if (path till the some first wait point of U is schedulable)
                  {
                        perform I's actions;
                        schedule(I);
                  }
```

```
            }
        }
        clock count ++;
}

schedule (I)
{
        w = current stage of the instruction
        for (all possible wait points w' or instruction end point along all
possible sub-paths from w)
        {
                if (I can be scheduled at the sub-path p)
                {
                        perform the corresponding the resource operations;
                        if (w' is not an end point)
                                set I's current stage to w';
                        if (time(p) == 0)
                                schedule(I);
                        else
                                next schedule time of I += time(p)
                        return;
                }
        }
        // cannot be scheduled
        next schedule time of I += 1;
}
```

Chapter 5

Examples

Here, we will try to present some sample processor and instruction architectures and how they will be represented in our framework.

We start with a simple non-pipelined processor with fixed memory access time, and gradually add features such as pipelining, control and data hazard handling, data bypassing, branch delay slots, multiple functional units, out of order execution and register renaming, branch prediction and complex memory interfaces.

For uniformity, the initial models assume that instruction execution can be divided into five distinct stages (similar to DLX [8]): fetch stage (fetch unit-FU), decode stage (decode unit-DU), execute stage (execute unit-EX), memory stage MU), and write-back (WB).

5.1 A simple non-pipelined processor

The simplest way to describe simple non-pipelined processor with each instruction taking a fixed number of clock cycles (say 5-as in our five stage model above) would be to have a static resource called processor, and all instructions use it for the duration equal to the execution clock cycle count (figure 6).

Figure 6: Non-pipelined

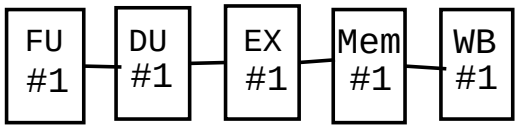

Figure 7: simple pipeline

DECLARATION:

StaticResource proc

USES ATTRIBUTE FOR ALL INSTRUCTIONS:

proc#5

5.2 Simple pipelined processors

Without any hazard handling Now we pipeline the five stages-without adding any handling mechanism for data or control hazard. (Note that the program execution model breaks away from the strictly sequential order of instructions-both in the data values that are read by instructions, and when branches show up-but then, but we are not proposing that design here!)

Since there are no interlocks, all stages are non-blocking. Having static resource for each resource does make sense here (figure 7).

DECLARATION:

StaticResource FU, DU, EX, MU, WB

USES ATTRIBUTE FOR ALL INSTRUCTIONS:

FU#1, DU#1, EX#1, MU#1, WB#1

With data hazard handling and forwarding, but no control hazard handling yet. This gives us an opportunity to introduce register resources-a register resource array for each register file. As we carry on, unless otherwise specified-we will assume that all instructions have one source register rs, and a destination register rd. Describing instructions with more sources/destinations will not result in any gain at the conceptual level. On the contrary, actual features we are interested in may, on the other hand, get suppressed (figure 8).

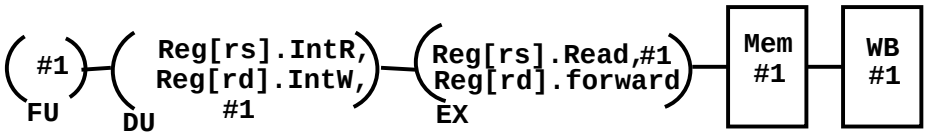

Figure 8: With data hazard handling (forwarding)

DECLARATION:

Buffer FU, DU, EX, MU

StaticResource WB

USES ATTRIBUTE FOR NON-LOAD INSTRUCTIONS:

[FU, #1], [DU, Reg[rs].itR & Reg[rd].itW, #1], [EX, Reg[rs].read, #1, Reg[rd].forward], MU#1, WB#1, Reg[rd].commit

USES ATTRIBUTE FOR LOAD INSTRUCTION:

[FU, #1], [DU, Reg[rs].itR & Reg[rd].itW, #1], [EX, #1], [MU, Reg[rs].read, #1, Reg[rd].forward], WB#1, Reg[rd].commit

An important point must be mentioned here: We have FU, DU, MU, EX as buffers instead of static resource of the previous example. This is because the execute stage (memory stage in case of a load instruction) issues a blocking operation read. In case the required register value is not available, the instruction stalls here. A pipeline stall at EX or MU propagates backwards-thus, all of them need to have the ability to hold the instruction for potentially unpredictable period (and this is what buffers are meant to capture).

Handling control hazard in addition to data hazards Control hazards arise when a branch instruction enters a pipeline-the PC value which is required by the following instructions is not available till (say) the EX stage of the branch.

We take this as a clue in our method of specifying such architectures–by having a separate PC buffer resource. Branch instruction acquires it at the fetch stage and does not release it till its execute stage. All other instructions try to acquire it during the fetch stage. Thus, instructions following a branch are stalled (figures 9 and 10) .

USES ATTRIBUTE FOR BRANCH INSTRUCTION:

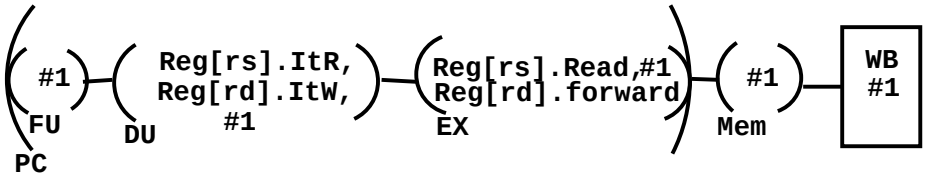

Figure 9: Handling control hazard in addition to data hazards (branch instruction)

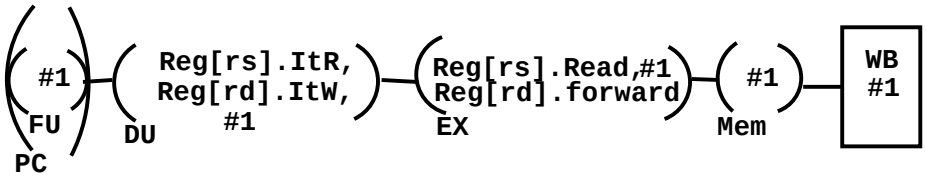

Figure 10: Handling control hazard in addition to data hazards (non- branch instruction)

[PC & [FU, #1], [DU,... #1], [EX,... #1]&]...

USES ATTRIBUTE FOR NON-BRANCH INSTRUCTIONS:

[PC & [FU, #1]&], [DU,...#1], [EX,... #1]...

Delayed branches as an optimization to branch handling In the case of delayed branches, the instruction following the branch instruction in the pipeline is executed-irrespective of whether the branch is taken or not taken (in the case of immediately successive branches, the delay slot may not be the instruction following in the program order). We handle this by allowing the branch instruction to release the PC for a cycle to prevent delay slot instruction from stalling (figure 11) .

USES ATTRIBUTE FOR BRANCH INSTRUCTION:

[PC & [FU, #1]], [DU,... #1], [PC, [EX,... #1]&]...

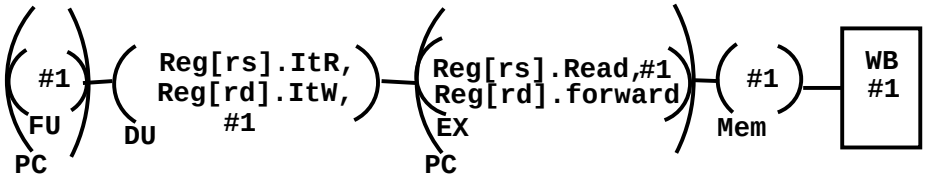

Figure 11: Delayed branches as an optimization to branch handling

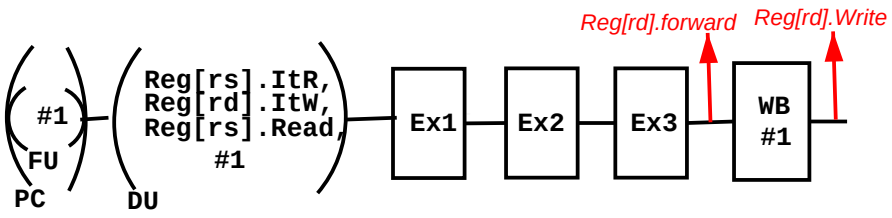

Figure 12: Multiple functional units: stalling at issue logic

At this point, one may have the following doubt. Consider the case when a branch instruction is just past the decode stage. The branch instruction contends for PC, and so does the instruction which is just about to enter the pipeline. Inorder to preserve the correct delay slot timing, the simulator must allocate the PC to the branch instruction. How is this assured?

This is where our resource scheduling order comes to rescue. The execution unit, being deeper into the pipeline than the fetch unit has its instructions considered for resource operations first. In this case, the branch instruction is considered and allocated PC first.

5.3 Processors with multiple functional units

In this case, an instruction diverges after decode stage to the relevant functional unit. However, these paths may have to merge later as writeback. As different functional units may have different delays, the writeback unit may be contended for-also, the writebacks must be serialized in program order.

There are two ways to handle this contention:

- Stall the issue logic to ensure that by the time the instructions reach the writeback, the previous writebacks are through and none of the following instructions will try to acquire it at the same time (figure 12) .

  DECLARATION:
  Buffer FU, DU
  StaticResource EX1, EX2,..,WB

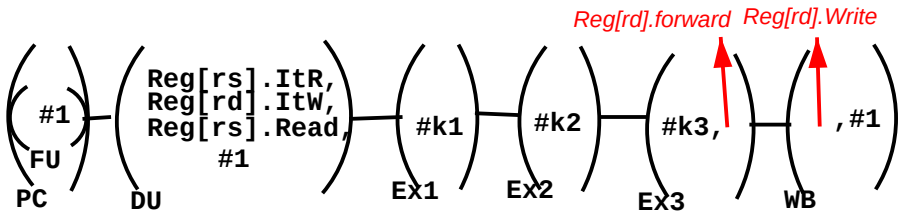

Figure 13: Multiple functional units: using latches for stalling at multiple points

USES ATTRIBUTE:
[FU, #1], [DU, ... , #1], EXi#k,..., Reg[rd].forward, WB#1, Reg[rd].commit

- Allow the instruction to go into the units as deep into the pipeline as possible- allowing them to stall anywhere in the middle of the pipeline. This happens when stall due to contention for the writeback unit propagates backwards (figure 13) .

DECLARATION:
Buffer FU, DU, EX1, EX2
StaticResource WB
USES ATTRIBUTE:
[FU, #1], [DU, ... , #1], [EX1,...], ..., [EXk, #1, Reg[rd].forward], WB#1, Reg[rd].commit

5.4 Dynamic Scheduling

Dynamic scheduling techniques allow the instructions execute out of order but at the same time maintaining the data-dependencies using some advanced techniques.

Reorder buffer[6] Reorder buffer mechanism is a common mechanism to enforce inorder completion of instructions. Here, the slots are allocated to the instructions in the program order. After execution, the results are stored either in these slots, or renamed registers (along with exception flags, if any). The completed and exception

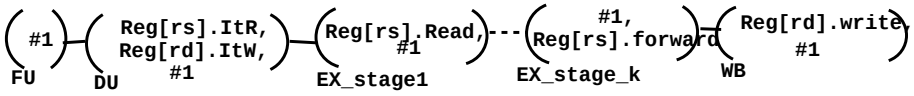

Figure 14: Scoreboarding

free instructions are completed and ejected out of the reorder buffer in program order. In case of exceptions, however, all the following instructions in the reorder buffer are killed and handler started.

DECLARATION:
Buffer FU, DU, EX, MU
StaticResource WB
USES ATTRIBUTE FOR ALL INSTRUCTIONS:
[FU, #1], [DU, ... , #1], [ROB, [EX....],], WB#1, Reg[rd].commit

Scoreboarding[5] In this technique, the issue logic blocks an instruction issue till all the previous register intending to write to the same register as the destination of the current instruction have written. The instruction then waits in the first stage of the functional unit till the data values it needs are available, and at the last stage (just before at the writeback) till all the previous instructions reading from the same register as the destination of the current instruction have actually read the values (see figure 14).

DECLARATION:
Buffer FU, DU, EX_stage1, ..., EX_stagek
StaticResource WB
// Register file must have blocking intention to writes
USES ATTRIBUTE FOR ALL INSTRUCTIONS:
[FU, #1], [DU, Reg[rs].itR & Reg[rd].itW, #1], [EX_stage1, Reg[rs].read, #1, Reg[rd].forward], ... , [EX_stage1, Reg[rd].commit], WB#1

Tomasulo architecture[7] Each functional unit has a set of tagged reservation stations where instructions waiting for executing in that unit wait-for data and/or

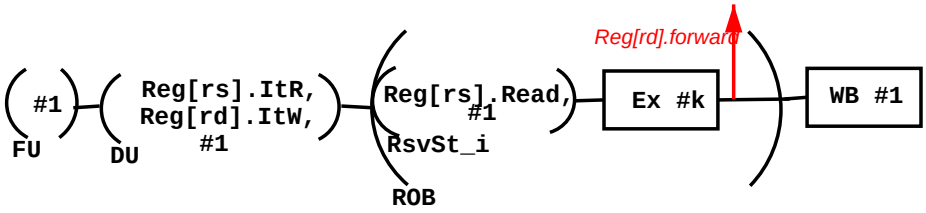

Figure 15: Reorder buffer

the unit itself. When an instruction I_1 intends to write a register R, it marks the register as busy, and register tag to its reservation station. If before I_1 completes, another instruction I_2 wants to read R-it also copies the tag of I_2. On completion, I_1 broadcasts the data along with the tag-from where it is copied into the I_2's reservation station and the register file (see figure 15).

DECLARATION:

Buffer ROB, RStation, FU, DU

StaticResource EX

USES ATTRIBUTE FOR ALL INSTRUCTIONS:

[FU, #1], [DU, Reg[rs].itR & Reg[rd].itW, #1], [ROB & [RStation, Reg[rs].read, #1], EX#k, Reg[rd].forward], WB#1, Reg[rd].commit

5.5 Branch prediction

Since varied branch prediction schemes are found in practice, we recommend pushing the actual branch prediction logic to Sim-nML canonical functions. Here we focus on handling the effect of correct or incorrect speculation-both, for uniscalars and superscalars. In actually processors, incorrect predictions involves killing speculatively executed (or in execution) instructions. Instead, we associate a fixed penalty called branch penalty with an incorrect speculation.

Uniscalars (figure 16)

DECLARATION:

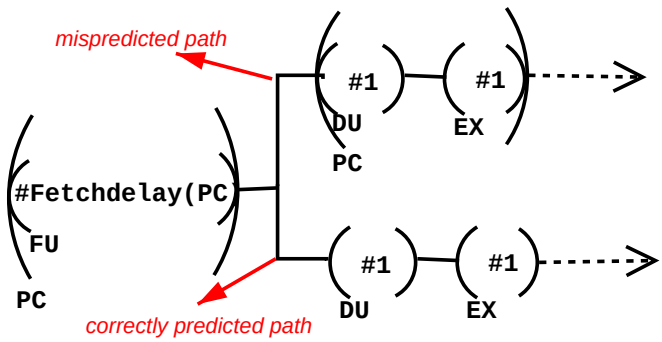

Figure 16: Branch prediction in uniscalars (branch instruction)

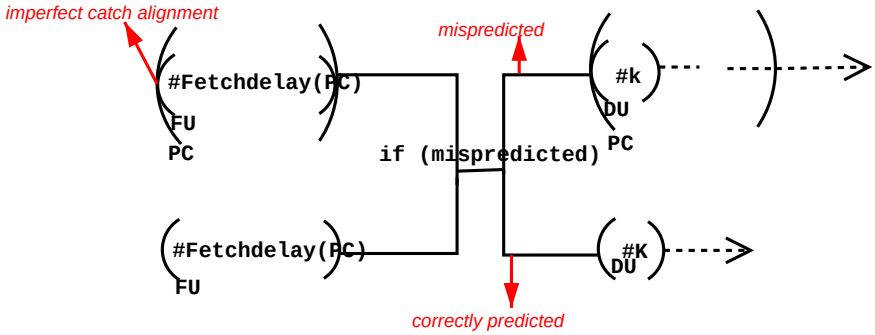

Figure 17: Branch prediction in superscalars: branch instruction

Buffer FU, PC, DU, EX

USES ATTRIBUTE FOR BRANCH INSTRUCTION:

[PC & [FU, #fetchdelay(PC)], if (mispredict) { [DU, ..., #1], [EX, ...],]} else {], [DU,...#1], [EX, ...]} endif, ...

USES ATTRIBUTE FOR NON-BRANCH INSTRUCTIONS:

[FU & [PC, #1]&], [DU, ..., , #1], [EX, ..., #1], ...

Superscalars In superscalars, multiple instructions are fetched in one clock cycle. Only the first instruction in the cache line needs the PC to be fetched-the rest ride along. The condition as to whether an instruction is in previous line fetch is captured in canonical function (see figures 17 and 18).

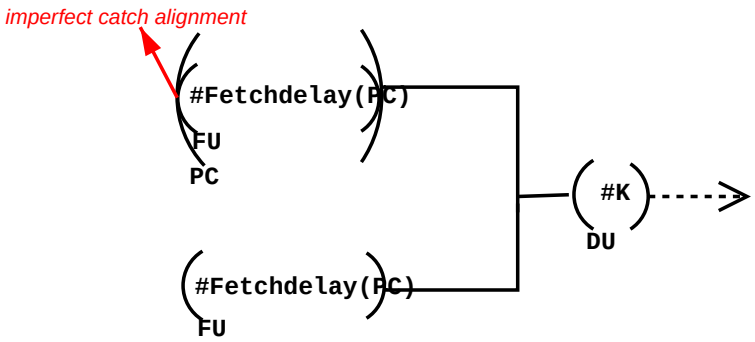

Figure 18: Branch prediction in superscalars: non-branch instruction

DECLARATION:

Buffer PC, FU[4], DU[4]..

USES ATTRIBUTE FOR BRANCH INSTRUCTIONS:

if (inprevCacheLineFetch([PC & [FU, #fdelay(PC)] &] , if (mispredict()) {[PC, [DU, ...], ...]} else {[DU, ...]... } endif, ...

USES ATTRIBUTE FOR NON-BRANCH INSTRUCTIONS:

if (inprevCacheLineFetch([PC & [FU, #fdelay(PC)] &] , [DU, ...], ...

5.6 Load-Store queues

Most current processors have an advanced memory interface-usually, the processor operations are intermediated by some queues. Here we give sample descriptions of load-store instructions of PowerPC 620[4].

Load The load instruction is issued to the reservation station where it waits for the address register value. In case of a cache-hit, the reservation station is released after one cycle. In case of a cache miss-the reservation station is released, a load buffer is acquired and held for a delay determined by memory latency (see figure 19).

DECLARATION:

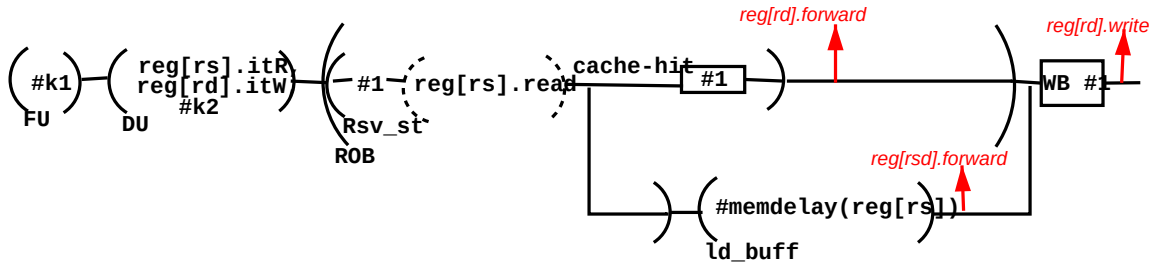

Figure 19: Load instruction in PowePC 620

Buffer ROB[...], RStation_load[...], LoadBuff, FU, DU

USES ATTRIBUTE FOR LOAD INSTRUCTION:

[FU, [DU, ..., [ROB & [RStation_load, if (cachehit(...)) { #1], } else { [LoadBuff, #memdelay(...)] } endif, ...

Store The store buffers stays in the reservation station till it gets the address value. Thereafter, it is issued to another buffer where it waits for the data value. On getting the data value-it moves to a next buffer where the instructions wait for the commit signal from the reorder buffer. On getting a commit signal, the cache line is written in a constant delay.

DECLARATION:

Buffer ROB[...], RStation_store[...], StoreBuff_1, StoreBuff_2, FU, DU
StaticResource WB

USES ATTRIBUTE FOR STORE INSTRUCTION:

[FU, #1], [DU, ...], [ROB & [RStation_store, Reg[rs_addr].read, #1], [StoreBuff_1, Regs[rs_data]. read, #1], [StoreBuff_2]&], WB#1, Reg[rd].commit

Chapter 6

Suggested future work

6.1 Scheduler optimizations

A general uses description may contain large number of latches in any pipeline path. In such cases, instead of scheduling instructions at every latch, it may often be possible to "delegate" this "responsibility" to a single (or a few number) of latches. For e.g., in the figure 20, the wait point p1 may be able to schedule instructions till the wait point p2. This can be further generalized: in the figure 21 root R is a wait point. Also, nodes n1, n2 ... are terminating nodes in the pipeline (this DAG is a simplified picture of interconnected components in a pipeline). In this case, it would be possible to delegate scheduling responsibility of all descendents of R to R. In general, a root node in a tree like topology may take up the responsibility of scheduling for all its descendents. There may be other interesting topologies where such optimizations can be applied as well, and they appear to be an interesting area

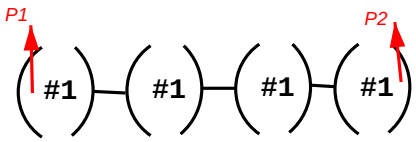

Figure 20: Point p1 can schedule all the way upto p2

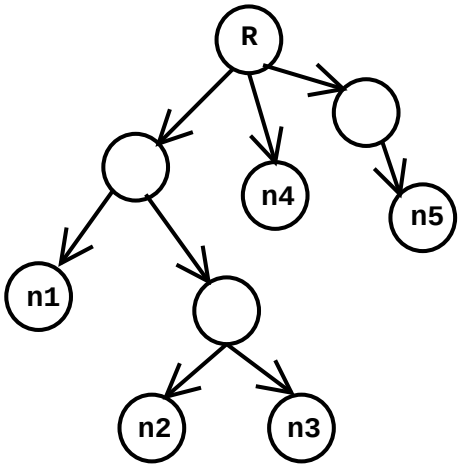

Figure 21: Root R can schedule for all its descendents

to investigate further.

6.2 Handling speculative execution (as in case of branches) and interrupts

Before elaborating on this–we first argue that speculative execution mechanisms and precise exception enforcing mechanisms on processors are intertwined–both require the ability to commit upto a particular instruction, undo the following instructions, and start program execution from a different point (interrupt handler, or branch target/follow-through instructions as the case may be). Intuitively, an instruction is not likely to killed at any stage in its execution–much less so while in the functional units. Typical killing mechanisms involve freeing up the buffers allocated to an instruction–in effect, it is as if all the killed instructions' resource usage is *snapped* (or broken) at the next scheduling point. The bonus we get here is in economy of description-we don't have to give additional specification for an instruction as to the way it is killed (which may be at various places)–the responsibility of such description lies solely with the "culprit" (killing) instruction, and the killing mechanism.

Since buffers in our model capture those wait points, they offer an attractive points where a construct like *kill* of an instruction can be defined. Annotating kill

with some qualifying predicate allows handling speculative execution: for instance one could mark the instructions that are speculatively fetched, and kill them at buffers once the speculation is known to be incorrect. At this place, it may be more appropriate to look at the above proposals from the perspective of reorder buffer mechanism or enforcing the above: All the instructions (speculatively fetched or otherwise) are residing in the reorder buffer in the program order (the processor design allocates them in program order), the one could simply specify something like "kill (free up the reorder buffer slots associated with) all instructions in the reorder buffer with PC less than equal to the PC of conditional jump instruction" in case of mispredicted branch.

6.3 Canonical function semantics

With each static resource, there is an associated delay. This delay can be some constant integer, or in a more general case, can be a canonical function. A canonical function can take arguments and returns an integer. They provide flexibility in the description as delays arising out of factors like memory or cache access, incorrect branch prediction etc can be captured using them.

Evaluation semantics There are 3 options that can be considered for their evaluation:

1. Evaluate all canonical functions in an instruction uses when the instruction is fetched.

2. Evaluate only the value of the arguments to the canonical function in the beginning, but evaluate the canonical function in the cycle it occurs.

3. Evaluate the canonical function, as also the argument values, in the cycle in which it is encountered.

The third option can be erroneous. This is because action part of subsequent instructions may update the system state, and so the arguments to the canonical

function (which may be register values) would be incorrect (as the effect of subsequent instructions should not be seen by an instruction).

The first option seems correct, and it is also the option currently being adopted in our model. It however may be limiting: the second option would allow much more flexibility. This flexibility could be needed, for example, when accurate simulation of external memory buffers is to be done using canonical functions, as now not only the sequence of requests/responses to the memory would be important, the delay between any two requests/responses may be important too. This flexibility, however, would be at a cost: an erroneous use of canonical function may be difficult to detect in these cases. It would be desirable to model external memory using canonical functions and study their timing behavior in order to have a more precise canonical function semantics.

6.4 Formal analysis

In as much as seen in the previous chapters we have been trying to analyze the expressive power of the model by writing sample descriptions in it. A complete formal description of a general processor's timing behavior remains a challenge.

6.5 Description of multiprocessor systems

A single chip, multiprocessor system consists of array of processor elements. It may be desirable to code in Sim-nML the individual tile elements and then interface the multiple simulators (of these elements) suitable. This would need a more detailed study of interfacing requirements among simulators in Sim-nML framework. Alternatively, it may be possible to generate a simulator for an entire RAW like processor from a single Sim-nML description.

Chapter 7

Conclusion

Our resource model, along with the attached scheduling semantics is a promising way of doing timing simulation for a processor. It can be implemented directly in the Sim-nML framework to automatically generate timing simulators, and our belief is the model would yield simulation results that would be very close to the true values. There is tremendous flexibility in the model: based on how much accuracy is desired, a designer could use it in variety of ways. For instance one could use it to capture simple, bare-bone pipeline structure of a processor, go on to handle hazards, further proceed to even specify scheduling optimizations, handle delays due to cache mis-alignment, include dynamic scheduling methodologies, and so on. A side-effect of our flexible model may be a bit of burdensome constructs for very simple descriptions: but then processor design is getting more detailed and sophisticated by the day, and a flexible model would have much more longevity.

Appendix A

Complexities in processor descriptions[8]

Pipelining

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. It resembles an assembly line: different steps are completing different parts of different instructions in parallel.

The major hurdle of Pipelining: Pipeline Hazards Hazards are situations that prevent next instruction in the instruction stream from executing during its designated cycle. There are three classes of hazards:

1. Structural hazards arise from conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution

2. Data hazards arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. Consider two instructions i and j, with i occurring before j. The possible data hazards are:

   (a) RAW (read after write): j tries to read a source before i writes it, so j incorrectly gets the old value

(b) WAW (write after write): j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination.

(c) WAR (write after read): j tries to write a destination before it is read by i, so i incorrectly gets the new value.

3. Control hazards arise from the pipelining of branches and other instructions that hazard the PC.

Other complications

Other processor features which add to their diversity include bypassing, instruction reordering, branch prediction, speculative execution, register renaming, register rotation and windowed register file.

Appendix B

Resource Declarations and Instruction Resource Usage Grammar

```
ResourceSpec: STATIC_RESOURCE[1] StaticResourceList
            | BUFFER Bufferlist
            | REGISTER Registerlist
            ;

StaticResourceList: ArrayId
                  | StaticResourceList ',' ArrayId
                  ;

BufferList: BufferDef
          | BufferList ',' BufferDef
          ;

BufferDef: ArrayId ':' '{' SLOTS '=' INTEGER ';' INPUTORDER '='
InputOrder ';' OUTPUTORDER '=' OutputOrder '}'
```

[1] All capitals used in this grammar specifications stand for keywords and/or lexical tokens such as integers.

```
        ;

InputOrder: STATIC '=' StaticInputOrderSeq
          |
          ;

StaticInputOrderSeq: ID
                   | StaticInputOrderSeq ',' ID

OutputOrder: INORDER
           | ANYORDER
           ;

RegisterList: ArrayId
            | RegisterList ',' ArrayId
            ;

UsesDef: UsesDef ']'
       | UseDef
       | UsesDef ',' UseDef

UseDef: IF '(' Expr ')' THEN UsesDef ENDIF
          | IF '(' Expr ')' THEN UsesDef ELSE UsesDef ENDIF
          | '(' UsesDef ')' '|' '(' UsesDef ')'
          | AndUseDef
          | '(' UsesDef ')'
          | ResourceId '.' USES
          ;

AndUseDef: SingleUseDef
         | AndUseDef '&' SingleUseDef
```

```
                ;

SingleUseDef: ID '#' INTEGER
                | '[' ID
                | ']'
                | ID '.' INTENDTOREAD
                | ID '.' INTENDTOWRITE
                | ID '.' READ
                | ID '.' AVAILABLE
                | ID '.' WRITE
                | '#' INTEGER
                ;
```

Appendix C

Algorithms and Pseudo-codes

C.1 Syntax directed translation for converting a Usage specification to Usage Graph

Usage: Φ
{
Usage.head = Φ;
Usage.isNullifiable = true;
Usage.last = Φ;
Usage.E = Φ;
Usage.V = Φ;
}

Usage_1 : *list*(*and*(AndUsage), Usage_2)
{
Usage_1.head = AndUsage;
Usage.$_1$isNullifiable = false;
Usage_1.last = Usage_2.last;
if (Usage_2.isNullifiable)
Usage_2.last = Usage_2.last $\cup${AndUsage};
Usage_1.E = Usage_2.E $\cup${(u, AndUsage): u $\in\text{Usage}_2$.head};
Usage_1.V = Usage_2.V $\cup${AndUsage};

}

$Usage_1$: *list*(*or*(condition, $Usage_2$, $Usage_3$), $Usage_4$)

{

$Usage_1$.head = $Usage_2$.head ∪$Usage_3$.head;

if ($Usage_2$.isNullifiable or $Usage_3$.isNullifiable)

$Usage_1$.head = $Usage_1$.head ∪$Usage_4$.head;

$Usage._1$isNullifiable = ($Usage_2$.isNullifiable or $Usage_3$.isNullifiable) and $Usage_4$.isNullifiable

$Usage_1$.last = $Usage_4$.last;

if ($Usage_4$.isNullifiable)

$Usage_1$.last = $Usage_1$.last ∪$Usage_2$.last ∪$Usage_3$.last;

$Usage_1$.V = $Usage_2$.V ∪$Usage_3$.V ∪$Usage_4$.V;

$Usage_1$.E = $Usage_2$.E ∪$Usage_3$.E ∪$Usage_4$.E∪{(u, v): u ∈$Usage_2$.last ∪$Usage_3$.last, v∈$Usage_4$.head} ;

}

C.2 Generation of Resource graph from Instruction Usage Graph

V_R= { set of buffers} ;

E_R = Φ;

for (each instruction I)

{

for (each start node s of the start nodes)

GenerateResourceOrder(UsageGraph(U), s, Φ);

}

GenerateResourceOrder(U, v, S)

{

T =set of buffers acquired at node v;

if (T≠ Φ)

```
    {
        E_R = E_R ∪{(b_i, b_j): b_i ∈S, b_j ∈T};
        S = Φ;
    }
    S = S ∪set of buffers acquired at v;
    for (each w ∈succ(v))
        GenerateResourceOrder(U, w, S);
}
```

Bibliography

[1] Sim-nML, http://www.cse.iitk.ac.in/sim-nml/

[2] Sim-nML grammar, http://www.cse.iitk.ac.in/sim-nml/moona/sim-nml/simnml-grammar.ps.gz

[3] Markus Freericks, *The nML Machine Description Formalism*, TU Berlin Computer Science Technical Report- Update and Revised Version 1.5

[4] T. A. Tiep, C. Nelson, and J. P. Shen, Performance evaluation of PowerPC 620 microarchitecture, Proceedings of 22nd Symposium of Computer Architecture, Santa Margherita, Italy, 1995

[5] G. S. Sohi, Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers, IEEE Transactions on Computers, March, 1990

[6] James E. Smith and Andrew R. Pleszun, Implemention of precise interrupts in pipelined processors

[7] R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, IBM Journal of Research and Development, January, 1967

[8] David A. Patterson and John L. Hennessy, Computer architecture: A quantitative approach, Morgan Kaufmann, 1996