Term Rewriting System Models of Modern Microprocessors

by

Lisa A. Poyneer

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 14, 1999
[June 1999]

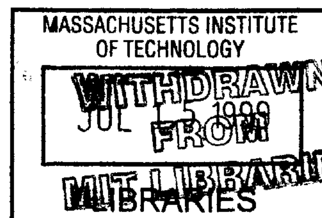
© Lisa A. Poyneer, MCMXCIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 13, 1999

Certified by ....
Arvind
Johnson Professor of Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses



ENG

Term Rewriting System Models of Modern Microprocessors

by

Lisa A. Poyneer

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Term Rewriting System models and corresponding simulators represent a powerful, high-level approach to hardware design and testing. TRS principles are discussed, along with modeling techniques such as pipelining and modularity. Two RISC Instruction Set Architectures are the focus of the modeling. Eleven models are presented, varying in complexity from a single-cycle version to a speculative, modular, out-of-order version. Principles of generating simulators for TRS models are discussed. Five simulators are presented and used on a test-suite of code as an example of the benefits of simulation.

Thesis Supervisor: Arvind
Title: Johnson Professor of Computer Science

Acknowledgments

I would like to thank Arvind for giving me the opportunity to work with him and CSG and for his many hours of both professional and technical guidance. I sincerely appreciate his efforts on my behalf. Thank you to James Hoe for being a great mentor, collaborator and role model. Thanks to Dan Rosenband for a great proof-read of my thesis and for being a good officemate. Thanks to everyone else is CSG, particularly Mitch Mitchell, Jan-Willem Maessen and Alex Caro.

Contents

1 Introduction **13**

1.1 Problem statement . . . 13
1.2 Term Rewriting Systems . . . 14
1.2.1 Definition of Term Rewriting System . . . 14
1.2.2 Applicability of TRS . . . 15
1.3 Processors . . . 16
1.3.1 Pipelining . . . 16
1.3.2 Speculative Execution . . . 17
1.3.3 Out-of-order execution . . . 17
1.4 The Instruction Sets . . . 18
1.4.1 AX ISA . . . 18
1.4.2 DLX ISA . . . 18
1.5 Basic TRS Building Blocks . . . 20
1.6 Summary . . . 20

2 Simple Non-Pipelined Models **21**

2.1 Harvard Model and design principles . . . 21
2.2 The Harvard AX Model, M_{ax} . . . 22
2.2.1 Definition . . . 22
2.2.2 Rules . . . 22
2.2.3 Example of execution of M_{ax} . . . 23
2.3 The Harvard DLX Model, M_{dlx} . . . 23
2.3.1 Definition . . . 24
2.3.2 Rules . . . 25
2.4 The Princeton Model and design principles . . . 27
2.5 The Princeton AX Model, MP_{ax} . . . 27
2.5.1 Definition . . . 27

2.5.2 Rules . . . 27
2.5.3 Example of execution . . . 29
2.6 Alternatives and discussion . . . 29
2.6.1 Definition . . . 31
2.6.2 Rules . . . 31
2.7 Summary . . . 32

3 Simple Pipelined models **35**
3.1 Pipelining in hardware and TRS models . . . 35
3.2 New types for TRSs . . . 36
3.3 The Stall Pipelined AX Model, $Mpipe_{ax}$ . . . 37
3.3.1 Definition . . . 37
3.3.2 Rules . . . 37
3.3.3 Example of execution . . . 39
3.4 The Bypass Pipelined AX Model, $Mbyp_{ax}$ . . . 41
3.4.1 Definition . . . 41
3.4.2 New Rules . . . 41
3.4.3 Example of execution . . . 43
3.5 Summary . . . 43

4 Complex models **47**
4.1 New types for TRSs . . . 48
4.2 Branch Prediction Schemes . . . 48
4.2.1 Branch Prediction Schemes . . . 49
4.3 The Speculative AX Model, $Mspec_{ax}$ . . . 49
4.3.1 Definition . . . 49
4.3.2 Rules . . . 49
4.4 Register Renaming and Multiple Functional Units . . . 53
4.5 The Register-Renaming AX Model, Mrr_{ax} . . . 53
4.5.1 Definition . . . 53
4.5.2 Rules . . . 53
4.6 Summary . . . 61

5 Simulation **63**
5.1 Principles of Simulation . . . 63
5.1.1 Clock-centric Implementations . . . 63
5.1.2 Rule-centric Implementations . . . 67

5.2 The Simulators for AX TRSs 67
5.2.1 M_{ax} 67
5.2.2 $Mpipe_{ax}$ 67
5.2.3 $Mbyp_{ax}$ 68
5.2.4 $Mspec_{ax}$ 68
5.2.5 Mrr_{ax} 68
5.3 Test results on simulations 69
5.4 Summary 70

6 Conclusions **71**

A DLX Models **73**
A.1 The Princeton DLX Model, MP_{dlx} 73
A.1.1 Definition 73
A.2 The Pipelined DLX Model, $Mpipe_{dlx}$ 75
A.2.1 Definition 75
A.2.2 Rules 76
A.3 The Speculative DLX Model, $Mspec_{dlx}$ 81
A.3.1 Definition 81
A.3.2 Rules 81

B Hardware Generation via Compilation **85**

List of Figures

1-1 Example execution trace for M_{gcd} 15

2-1 Example initial state for M_{ax} 23

2-2 Example execution trace for M_{ax} 24

2-3 Example initial state for MP_{ax} 29

2-4 Example execution trace for MP_{ax} 30

3-1 Example initial state for $Mpipe_{ax}$ 39

3-2 Example execution trace for $Mpipe_{ax}$ 40

3-3 Example initial state for $Mbyp_{ax}$ 44

3-4 Example execution trace for $Mbyp_{ax}$ 44

4-1 2-bit prediction 50

4-2 Schematic of the units of Mrr_{ax} 54

4-3 Definition of Mrr_{ax} 55

4-4 State transitions for IRB entries in ROB 60

List of Tables

5.1 Relations between $Mpipe_{ax}$ rules 68

5.2 Relations between $Mspec_{ax}$ rules 69

5.3 Instruction Mix for Testing Code 69

5.4 Results of simulators for AX models 70

Chapter 1

Introduction

1.1 Problem statement

Term Rewriting Systems (TRSs) have long been used to describe programming languages. Recent work ([1],[2] and[3]) has investigated using TRSs to describe modern computer hardware such as processors and memory systems. This research has lead toward the development of a complete system for design, testing and production of hardware using TRSs.

TRSs, used in conjunction with simulators and hardware/software compiler, will greatly improve the processor development process. TRS models are intuitive to write, are modular, and allow a complete description to be written in days or even hours. They are easy to reason with and are amenable to proofs. Simulators, which are straightforward to write, are easily modifiable if the TRS changes and, with instrumentation, allow for testing and profiling throughout the development process. Debugging, corrections and design changes can be done at high-level and in earlier design stages, instead of after the hardware is generated. The compiler will automatically generate target code (either Verilog or a high-level programming language) removing the burden of hand-coding from scratch.

Instead of designing a chip, hand-writing the Verilog code, waiting for the hardware to return and testing it and iterating the above for design changes, TRS will allow the designer to write and manipulate high level descriptions, testing in software and making changes on the fly. The compiler will automatically generate the hardware description at the end of the design process. Just as the use of high-level programming languages and their compilers was a great improvement over writing assembly code, we hope that the use of TRS and its simulators and compilers will similarly improve hardware design.

This thesis is concerned with the modeling and simulation steps described above. The Introduction discusses the basics of TRS and fundamental concepts in computer architectures that are

built on later. Chapters 2, 3 and 4 present many new models of the DLX and AX ISAs and provide discussion on the relative merits and design challenges. Chapter 5 addresses the issues involved in simulation and synthesis of hardware from these TRS models and presents several simulators, along with the results of running them on a small test-suite of code.

1.2 Term Rewriting Systems

1.2.1 Definition of Term Rewriting System

A Term Rewriting System model consists of a set of terms, a set of rules for rewriting these terms and a set of initial terms. To begin, the model has a definition, or declaration, of all the state elements by type.

The definition is followed by the rules. Each rule has an initial state and if clause, which together represent the *precondition* that must be met for the rule to be enabled. The rule also has a specification of the rewritten state, with an optional where clause to contain bindings used. (In Section 5.1 we work further with the format of a TRS) For a set of rules, a TRS executes as follows. Whenever a rule has its precondition satisfied, it has been *triggered*. Multiple rules may trigger at the same time. However, only one rule can *fire* at once, and this firing happens atomically. With multiple rules triggered, the one to fire is chosen randomly. A system proceeds until no rules are triggered. If, given a specific initial state, a system always reaches the same final state no matter the order of rule firing, it is called *confluent*.

A simple example of a TRS is one describing Euclid's GCD algorithm. This algorithm is both simple and well-known, so its proof is omitted here. This model, called M_{gcd}, begins with the two numbers and ends when just one number, the greatest common divisor, is left after rewriting.

```
PAIR  =  Pair(VAL, VAL)|| Done(NUM)
 VAL  =  Num(NUM)|| Mod(NUM, NUM)
 NUM  =  Bit[32]
```

Rule 1

```
    Pair(Num(x), Num(y))
                     if y = 0
==>       Done(x)
```

Rule 2

```
    Pair(Num(x), Num(y))
                     if y > x
==>       Pair(Num(y), Num(x))
```

Rule 3

```
    Pair(Num(x), Num(y))
                     if x ≥ y and y ≠ 0
==>       Pair(Num(y), Mod(x, y))
```

Rule 4

Mod(x, y)

if x ≥ y

⟹ Mod(v, y)

where v := x - y

Rule 5

Mod(x, y)

if x < y

⟹ Num(x)

State	Rule Applied
Pair(Num(231), Num(98))	Rule 3
Pair(Num(98), Mod(231,98))	Rule 4
Pair(Num(98), Mod(133, 98))	Rule 4
Pair(Num(98), Mod(35, 98))	Rule 5
Pair(Num(98), Num(35))	Rule 3
Pair(Num(35), Mod(98, 35))	Rule 4
Pair(Num(35), Mod(63, 35))	Rule 4
Pair(Num(35), Mod(28, 35))	Rule 5
Pair(Num(35), Num(28))	Rule 3
Pair(Num(28), Mod(35, 28))	Rule 4
Pair(Num(28), Mod(7, 28))	Rule 5
Pair(Num(28), Num(7))	Rule 3
Pair(Num(7), Mod(28, 7))	Rule 4
Pair(Num(7), Mod(21, 7))	Rule 4
Pair(Num(7), Mod(14, 7))	Rule 4
Pair(Num(7), Mod(7, 7))	Rule 4
Pair(Num(7), Mod(0, 7))	Rule 5
Pair(Num(7), Num(0))	Rule 1
Done(7)	-

Figure 1-1: **Example execution trace for** M_{gcd} The left column shows the state, the right column the triggered and fired rule. Given the numbers 231 and 98, M_{gcd} correctly calculates the gcd, which is 7. 231 has prime factorization $7 * 33$ and 91 has factorization $7 * 13$.

1.2.2 Applicability of TRS

TRSs have traditionally been used to describe software languages. A classic example is the SK combinatory system, which is generally expressed as the following two rules.

$$\begin{array}{rcl} K\ x\ y & \rightarrow & x \\ S\ x\ y\ z & \rightarrow & (x\ \ y)\ \ (y\ \ z) \end{array}$$

These rules may be expressed in our TRS notation as follows:

PAIR = Ap(E,E)
E = S || K|| Ap(E,E)

First

Ap(K, Ap(x, y)))
$\Longrightarrow$ Ap(K, x)

Second

Ap(S, Ap(x, Ap(y, z))))
$\Longrightarrow$ Ap(Ap(x, z), Ap(y,z))

SK formalism, unlike the lambda calculus, has no variables and ye has the power to express all computable functions. The study of these two rules has inspired a lot of theoretical and languages research. The TRSs considered in this thesis are much less powerful.

1.3 Processors

Modern Microprocessors are very different from the first computers that emerged forty years ago. To improve performance, most microprocessors can execute instructions in a pipelined manner. Modern microprocessor pipelines are complex and permit speculative and out-of-order execution of instructions. The models presented later in this work follow the evolution of microprocessors and become progressively more complex. Below is a summary of the key concepts and styles of implementations.

1.3.1 Pipelining

Pipelining seeks to exploit the fact that in a single-cycle implementation, most of the resources are idle during the long clock period. For example, after instruction fetch the instruction memory lies idle for the rest of the clock cycle, waiting for the register file, ALU and data memory to be used in turn. By splitting the data path into multiple stages, each with independent resources, multiple instructions can be executed at once, as on an assembly line. The reduction in combinational logic in each stage brings a corresponding reduction in the clock period. The initial pipelined model will not include floating point (FP) instructions, and will have linear order instruction issue and completion.

Pipelining, however, creates many new issues that the control logic must deal with. Execution of code in a pipelined implementation must be equivalent to execution on a non-pipelined implementation, so data, control and structural hazards must be solved. Data hazards arise when an instruction produces data necessary for a later instruction. Basic pipelines will have Read-after-Write (RAW) hazards, where one instruction reads data written by a preceding instruction. RAW data hazards can be solved with stalls or bypasses. Control hazards occur when one instruction controls what instructions are executed after it, possibly necessitating nullification of instructions already in the pipeline. Branches and interrupts cause control hazards. Structural hazards arise from competition for a single resource. Examples of structural hazards are competition for functional units. A correct pipelined implementation needs to correctly handle all of the hazards present to ensure correct

execution and handle them as efficiently as possible to improve performance. Pipelined models are presented in Chapter 3.

1.3.2 Speculative Execution

A significant percentage of instructions in normal execution are changes in the flow of control. In a simple pipelined implementation, a branch or jump will cause a few stalls during execution. In a more deeply pipelined implementation doing out of order execution, changes in control are much more difficult to deal with. All instructions before a branch must commit, while no instructions after (excepting the delay slot in DLX) may modify processor state. Changing the flow of control is also more expensive, because multiple instructions can be executing on different functional units at once.

If the target of a control flow change can be predetermined with reasonable accuracy on instruction issue, the costly flushing of the pipeline and/or delaying of instructions can be avoided. On issue some mechanism, frequently called the Branch Target Buffer, is accessed to determine the predicted next address. Execution is speculated with this new address until the actual target is resolved in later stages. If the prediction is correct, the processor wins, otherwise it undo the effect of incorrect instructions and transfer control to the correct target. If prediction has a high rate of accuracy, this strategy will be highly effective. There are many different ways to predict branch behavior, including one and two-level prediction and various adaptive strategies. Speculative models are presented in Chapter 4.

1.3.3 Out-of-order execution

In an inefficient implementation of a pipelined integer and FP data path, the long latencies of the FP units will cause a large number of stalls. By deviating from the linear paradigm, out-of-order issue and out-of-order completion can generate higher utilization of resources, with a corresponding jump in system complexity.

A modern processor has multiple functional units. These multiple units introduce a new set of problems for a pipelined implementation. Because different units will have different latencies (e.g. an add will be faster than a divide) enforcing in-order completion will cause backwards pressure on the pipeline. If out-of-order completion is allowed, care must be taken to avoid Write-after-write hazards. This can be done at either the issue stage or via write-back stage arbitration.

The variable latency of the units can also cause stalls in the issue stage while instructions wait for certain units to become available. Out-of-order execution can alleviate these delays. However, care must be taken to avoid Write-after-read anti-dependence hazards on issue. There are several common approaches to non-linear execution, most notably register renaming and score-boarding. A register renaming model is presented in Chapter 4.

1.4 The Instruction Sets

For this thesis two RISC ISAs have been chosen. RISC ISAs where chosen because the smaller number of instructions and standard instruction format would make comprehension easier, and the prevalence of RISC ISA in current industry development. AX, a minimalist RISC ISA has been used purely for research and teaching purposes. It's small size and simplicity make it ideal for introducing and illustrating new concepts. The more realistic DLX ISA, described in [4], is similar to current industry ISAs and provides an illustration of the power of TRS methods.

1.4.1 AX ISA

AX has only six instructions. It is a basic load/store architecture. Its instructions are as follows:

- Load Constant: r := Loadc(v), RF[r] ← v
- Load Program Counter: r := Loadpc, RF[r] ← pc
- Arithmetic Operation: r := Op(r1, r2), RF[r] ← rs1 op rs2
- Jump: Jz(r1, r2), if r1 == 0, pc ← RF[r2]
- Load: r:= Load(r1), RF[r] ← Memory[RF[r1]]
- Store: Store(r1, r2), Memory[RF[r1]] ← RF[r2]

1.4.2 DLX ISA

DLX features a simple load/store style architecture, and has a branch delay slot. DLX specifies the following:

- 32 1-word (32-bit) general purpose integer registers, with R0 as the bitbucket.
- 32 1-word floating point registers.
- Data types of 8-bit, 16-bit or 1-word for integers and 1-word or 2-word for floating point.

Instruction format

DLX was designed to have a fixed-length instruction format to decrease decode time. There are three instruction types: I-type, R-type and J-type.

I-Type $[opcode_{(6)}||rs1_{(5)}||rd_{(5)}||immediate_{(16)}]$

R-Type $[opcode_{(6)}||rs1_{(5)}||rs2_{(5)}||rd_{(5)}||func_{(11)}]$

J-Type $[opcode_{(6)}||offset_{(26)}]$

Types

Within the three instruction types above, there are 7 basic types of instructions that can be described in register transfer language. An example of each basic type is listed below.

- Register-register ALU operations (R-type): rd := rs1 func rs2
- Register-immediate ALU operations (I-type): rd := rs1 op immediate
- Loads (I-type): rd := Mem[(rs1 + immediate)]
- Stores (I-type): Mem[(rs1 + immediate)] := rd
- Conditional branches (I-type): if rs1, pc := pc + 4 + immediate else pc := pc + 4
- Jumps register (I-type): pc := rs1
- Jumps (J-type): pc := pc + 4 + displacement

DLX Instructions

Name	Description	RTL
ADD(U), SUB(U), MUL(U), DIV(U)	arithmetical operations	rd = rs1 op rs2
AND, OR, XOR	logical operations	rd = rs1 logicalop rs2
SLL, SRL, SRA	Shifts, logical and arithmetical	rd = rs1 << (rs2)
SLT, SGT, SLE, SGE, SEQ, SNE	Set logical operations	if (rs1 logicalop rs2) {rd1 = 1} else rd1 = 0
ADD(U)I, SUB(U)I, MUL(U)I, DIV(U)I	arithmetic immediate operations	rd = rs1 op immediate
ANDI, ORI, XORI	Logical immediate operations	rd = rs1 logicalop immediate
SLLI, SRLI, SRAI	Shifts by immediate, logical and arithmetic	rd = rs1 << immediate
BEQ, BNEZ	Branch (not) equal to	if (rs1) {pc = pc + 4 + immediate} else pc = pc + 4
J, JAL	Jump (optional link to r31)	pc = pc + 4 + immediate
JR, JALR	Jump register (optional link to r31)	pc = rs1
LB	Load byte	
LH	Load half-word	
LW	Load word	rd = Mem[(rs1 + immediate)]
LF, LD	Load single or double precision floating point	
SB	Store byte	
SH	Store half-word	
SW	Store word	Mem[(rs1 + immediate)] = rd

1.5 Basic TRS Building Blocks

There are several basic elements that we will use throughout this thesis in TRSs. There are two basic types that we will use: the single element (e.g. a register) and the labeled collection of these elements (e.g. a memory or register file.) A single element is represented as a single term of the type of data it contains. The program counter is therefore just a term of type ADDR. For a collection of elements, we use an abstract data type and define simple operation on it.

$$\begin{aligned} \text{PC} &= \text{ADDR} \\ \text{RF} &= \text{Array[RNAME] VAL} \\ \text{RF} &= \text{Array[ADDR] INST} \\ \text{RF} &= \text{Array[ADDR] VAL} \end{aligned}$$

For both register files and memories we adopt a shorthand convention for reading and writing elements. To mean "the value of the second element of the pair with label r" we write `rf[r]`. To mean "set the value of the second element of the pair with label r to be v" we write `rf[r := v]`

The types RNAME, VAL, ADDR, and INST must also be defined. For our purposes we have a 32-bit address space in memory, 32 registers store data in 32 bit blocks. To be thorough, ADDR should be the conjunction of 2^{32} different terminals (i.e. ADDR $= 0 \parallel 1 \ldots \parallel 2^{32} - 1$) but as shorthand we instead say Bit[n], where 0 is the first and $2^n - 1$ is the last terminal. The INST type is specified as being one of six different instructions.

$$\begin{aligned} \text{ADDR} &= \text{Bit[32]} \\ \text{INST} &= \text{Loadc(RNAME, VAL)} \parallel \text{Loadpc(RNAME)} \parallel \\ & \text{Op(RNAME, RNAME, RNAME)} \parallel \text{Load(RNAME, RNAME)} \parallel \\ & \text{Store(RNAME, RNAME)} \parallel \text{Jz(RNAME, RNAME)} \\ \text{RNAME} &= \text{Reg0} \parallel \text{Reg1} \parallel \text{Reg2} \parallel \ldots \text{Reg31} \\ \text{VAL} &= \text{Bit[32]} \end{aligned}$$

As we introduce new terms to cope with increasing model complexity, they will be discussed.

1.6 Summary

This Chapter has presented the basic concepts of Term Rewriting Systems, modern computer processor architecture techniques and two ISAs. The statement that TRS techniques and models present a powerful new way to design hardware will be justified in the following chapters. Chapters 2, 3 and 4 will present many models, increasing in complexity of both model and hardware for AX and DLX. Chapter 5 will discuss how to simulate these models and provide an example of the advantages of easily generated simulators.

Chapter 2

Simple Non-Pipelined Models

In this chapter the simple beginnings of TRS processor models are discussed. We begin with the most basic model, the Harvard model, and then use the Princeton model to show how to break functionality up across rules.

2.1 Harvard Model and design principles

A Harvard style implementation has separate memories for instructions and data. A very basic processor can be designed using this style and a single-cycle combinational circuit that executes one instruction per clock cycle. It can be easily designed by laying down the hardware necessary for each instruction. For example, a register add would require connections from the PC to the register file to read the operands then connections to the ALU. The ALU result and the target register from the PC are also connected to the register file.

Conceptually, the Harvard model is very simple, and the TRS description is likewise. For this model the state is the different state elements of the machine: the pc, register file, instruction memory and data memory. The functions that the hardware performs, such as addition or resetting the pc on a jump, are embodied in the where clauses of the rules. For each instruction that has a different opcode (e.g. add versus load) a different rule is needed to describe the different hardware action.

First the Harvard model for the AX ISA, M_{ax}, is presented, along with a sample execution trace to illustrate the execution of a TRS model. Next, the considerations for modeling the more complex DLX ISA are discussed along with that model, M_{dlx}.

2.2 The Harvard AX Model, M_{ax}

As described in Section 1.4 AX has only six instructions. M_{ax} has only seven rules (two for the Jz), with each rule containing all the functionality of a complete instruction execution.

2.2.1 Definition

This model is very simple. It contains only a program counter, register file and instruction and data memories. For definitions and discussion of the building block types, see Section 1.5

PROC = Proc(PC, RF, IM, DM)

2.2.2 Rules

The Loadc instruction is the simplest. The value specified in the instruction is stored to the target register. The pc is incremented to proceed with linear program execution.

Rule 1 - Loadc
Proc(pc, rf, im, dm)
if im[pc] == Loadc(r, v)
$\Longrightarrow$ Proc(pc + 1, rf[r := v], im, dm)

The Loadpc instruction stores the current pc into the target register.

Rule 2 - Loadpc
Proc(pc, rf, im, dm)
if im[pc] == Loadpc(r)
$\Longrightarrow$ Proc(pc + 1, rf[r := pc], im, dm)

The Op instruction adds the values stored in the two operand registers and stores the result in the target register.

Rule 3 - Op
Proc(pc, rf, im, dm)
if im[pc] == Op(r, r1, r2)
$\Longrightarrow$ Proc(pc + 1, rf[r := v], im, dm)
where v := Op applied to rf[r1], rf[r2]

The Load instruction reads the data memory at the address specified by the contents of register r1 and writes the data to register r.

Rule 4 - Load
Proc(pc, rf, im, dm)
if im[pc] == Load(r, r1)
$\Longrightarrow$ Proc(pc + 1, rf[r := v], im, dm)
where v := dm[rf[r1]]

The Store instruction stores the value of register reg at the address is data memory specified by

register ra.

Rule 5 - Store

Proc(pc, rf, im, dm)

if im[pc] == Store(ra, r1)

$\Longrightarrow$ Proc(pc + 1, rf, im, dm[ad := v])

where ad := rf[ra] *and* v := rf[r1]

If the value of register rc is 0, the branch is taken. In the successful case, pc is set to the contents of register ra. Otherwise the pc is incremented as normal.

Rule 6 - Jz taken

Proc(pc, rf, im, dm)

if im[pc] == Jz(rc, ra) *and* rf[rc] == 0

$\Longrightarrow$ Proc(rf[ra], rf, im, dm)

Rule 7 - Jz not taken

Proc(pc, rf, im, dm)

if im[pc] == Jz(rc, ra) *and* rf[rc] $\neq$ 0

$\Longrightarrow$ Proc(pc + 1, rf, im, dm)

2.2.3 Example of execution of M_{ax}

An example of M_{ax} from a initial start state is given as follows. Keep in mind that rules fire atomically and if many rules can fire, one is randomly chosen to fire. In this simple case, at most one rule can fire for any given state. independence and is discussed further in Chapter 5.

State	Value
pc	0
rf	all zeros
im	im[0] = Loadc(r0, 1)
	im[1] = Loadc(r2, -1)
	im[2] = Loadc(r1, 16)
	im[3] = Load(r3, r1)
	im[4] = Loadpc(r10)
	im[5] = Op(r3, r3, r3)
	im[6] = Jz(r0, r1)
	im[7] = Op(r0, r2, r0)
	im[8] = Jz(r0, r10)
	im[9] = Op(r3, r0, r3)
dm	dm[16] = 5

Figure 2-1: **Example initial state for M_{ax}**

2.3 The Harvard DLX Model, M_{dlx}

As described in Section 1.4, DLX is a more complex ISA. At this simple stage this just means more rules to write for each different type of instruction. The main difference from AX, besides the larger

State			fired
pc	rf	im	
0		im[0] = Loadc(r0, 1)	1
1	rf[0] = 1	im[1] = Loadc(r2, -1)	1
2	rf[2] = -1	im[2] = Loadc(r1, 16)	1
3	rf[1] = 16	im[3] = Load(r3, r1) = 5	4
4	rf[3] = 5	im[4] = Loadpc(r10)	2
5	rf[10] = 4	im[5] = Op(r3, r3, r3)	3
6	rf[3] = 10	im[6] = Jz(r0, r1)	7
7		im[7] = Op(r0, r2, r0)	3
8	rf[0] = 0	im[8] = Jz(r0, r10)	6
4		im[4] = Loadpc(r10)	2
5	rf[10] = 4	im[5] = Op(r3, r3, r3)	3
6	rf[3] = 20	im[6] = Jz(r0, r1)	6
16		im[16] = nop	-

Figure 2-2: **Example execution trace for M_{ax}** The current state is shown in the first four columns, followed by the triggered rules and single rule that fires. The next state is shown in the following line of the table.

number of instructions, is the branch delay slot. This is dealt with by using an extra term to store the next pc.

2.3.1 Definition

There are only a few elements of state necessary in the Harvard model: the PC, the register file, the next PC and the instruction and data memories. This next PC field is necessary to implement the branch delay slot required by DLX. Our TRS model of a Harvard processor is then:

$$\begin{aligned} \text{PROC} &= \text{Proc(PC, NEXT, RF, IM, DM)} \\ \text{NEXT} &= \text{ADDR} \end{aligned}$$

We need new definitions of instructions because this is a different instruction set:

$$\begin{aligned}
\text{INST} &= \text{REGREGOP} \parallel \text{SETLOGOP} \parallel \text{REGIMMOP} \\
&= \text{JUMPOP} \parallel \text{MEMOP} \\
\text{REGREGOP} &= \text{Regregop(RNAME, RNAME, RNAME, RRTYPE)} \\
\text{RRTYPE} &= \text{Add} \parallel \text{Sub} \parallel \text{Mul} \parallel \text{Div} \parallel \text{Addu} \parallel \text{Subu} \\
&\quad \text{Mulu} \parallel \text{Divu} \parallel \text{And} \parallel \text{Or} \parallel \text{Xor} \parallel \text{Sll} \parallel \text{Srl} \parallel \text{Sra} \\
\text{SETLOGOP} &= \text{Setlogop(RNAME, RNAME, RNAME, SLTYPE)} \\
\text{SLTYPE} &= \text{Slt} \parallel \text{Sgt} \parallel \text{Sle} \parallel \text{Sge} \parallel \text{Seq} \parallel \text{Sne} \\
\text{REGIMMOP} &= \text{Regimop(RNAME, VAL, RNAME, RITYPE)} \\
\text{RITYPE} &= \text{Addi} \parallel \text{Subi} \parallel \text{Muli} \parallel \text{Divi} \parallel \text{Addui} \parallel \text{Subui} \\
&\quad \text{Mului} \parallel \text{Divui} \parallel \text{Andi} \parallel \text{Ori} \parallel \text{Xori} \parallel \text{Slli} \parallel \text{Srli} \parallel \text{Srai} \\
\text{JUMPOP} &= \text{Beqz(RNAME, VAL)} \parallel \text{Bnez(RNAME, VAL)} \\
&= \text{J(VAL)} \parallel \text{Jal(VAL)} \parallel \text{Jr(RNAME)} \parallel \text{Jalr(RNAME)} \\
\text{MEMOP} &= \text{Lw(RNAME, VAL, RNAME)} \parallel \text{Lh(RNAME, VAL, RNAME)} \\
&= \text{Lb(RNAME, VAL, RNAME)} \parallel \text{Sw(RNAME, VAL, RNAME)} \\
&= \text{Sh(RNAME, VAL, RNAME)} \parallel \text{Sb(RNAME, VAL, RNAME)}
\end{aligned}$$

2.3.2 Rules

The rules for a Harvard style implementation are straightforward to write. The Instructions can be divided into three semantic groups and the rules are a direct translation from the instruction set.

Register Instructions

The arithmetic and logical operations simply apply the operator to the two register values (or one value and immediate) and save the result in the register file.

Reg-Reg Op

Proc(ia, nxt, rf, im, dm)
  if im[ia] == Regregop(rs1, rs2, rd, rrtype)
⟹ Proc(nxt, nxt + 1, rf[rd, v], im, dm)
  where v := rrtype(rf[rs1], rf[rs2])

Set-Logical

Proc(ia, nxt, rf, im, dm)
  if im[ia] == SetlogOp(rs1, rs2, rd, sltype) *and* sltype(rf[rs1], rf[rs2]) == true
⟹ Proc(nxt, nxt + 1, rf[rd, 1], im, dm)

Proc(ia, nxt, rf, im, dm)
  if im[ia] == Setlogop(rs1, rs2, rd, sltype) *and* sltype(rf[rs1], rf[rs2]) == false
⟹ Proc(nxt, nxt + 1, rf[rd, 0], im, dm)

Reg-Imm Rule

Proc(ia, nxt, rf, im, dm)
  if im[ia] == RegimmOp(rs1, imm, rs2, ritype)
⟹ Proc(nxt, nxt + 1, rf[rd, v], im, dm)
  where v := ritype(rf[rs1], imm)

Control Flow Instructions

Due to DLX's branch delay slot, the instruction after a branch or jump is always executed. The following jumps modify the next pc instead of the pc to account for that.

BEQZ

Proc(ia, nxt, rf, im, dm)
  if im[ia] == Beqz(rs1, imm) *and* rf[rs1] == 0
⟹ Proc(nxt, ia + 1 + imm, rf, im, dm)

Proc(ia, nxt, rf, im, dm)
  if im[ia] == Beqz(rs1, imm) *and* rf[rs1] ≠= 0
⟹ Proc(nxt, nxt + 1, rf, im, dm)

BNEZ

Proc(ia, nxt, rf, im, dm)
  if im[ia] == Bnez(rs1, imm) *and* rf[rs1] ≠= 0
⟹ Proc(nxt, ia + 1 + imm, rf, im, dm)

Proc(ia, nxt, rf, im, dm)
  if im[ia] == Bnez(rs1, imm) *and* rf[rs1] == 0
⟹ Proc(nxt, nxt + 1, rf, im, dm)

Jump

Proc(ia, nxt, rf, im, dm)
  if im[ia] == J(imm)

$\Longrightarrow$ Proc(nxt, ia + 1 + imm, rf, im, dm)

Jump and Link

Proc(ia, nxt, rf, im, dm)
if im[ia] == Jal(imm)
$\Longrightarrow$ Proc(nxt, ia + 1 + imm, rf[r31, nxt + 1], im, dm)

JumpRegister

Proc(ia, nxt, rf, im, dm)
if im[ia] == Jr(rs1)
$\Longrightarrow$ Proc(nxt, rf[rs1], rf, im, dm)

JumpRegister and Link

Proc(ia, nxt, rf, im, dm)
if im[ia] == Jalr(rs1)
$\Longrightarrow$ Proc(nxt, rf[rs1], rf[r31, nxt + 1], im, dm)

Memory Instructions

Memory operations are straightforward. The half-word and byte load operations return a padded version of the low two or one bytes of that memory location. The store versions write only part of the word by loading the current whole word and combining the new data before writing. Note that these store rules present a problem in implementation because they both read and write data memory in a single cycle.

Load Word

Proc(ia, nxt, rf, im, dm)
if im[ia] == Lw(rs1, imm, rd)
$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, dm[addr]], im, dm)
where addr := rf[rs1] + imm

Load Half-Word

Proc(ia, nxt, rf, im, dm)
if im[ia] == Lh(rs1, imm, rd)
$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, v], im, dm)
where v := LogicalAnd(0x0011, dm[addr]) *and* addr := rf[rs1] + imm

Load Byte

Proc(ia, nxt, rf, im, dm)
if im[ia] == Lb(rs1, imm, rd)
$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, v], im, dm)
where v := LogicalAnd(0x0001, dm[addr]) *and* addr := rf[rs1] + imm

Store Word

Proc(ia, nxt, rf, im, dm)
if im[ia] == Sw(rs1, imm, rd)
$\Longrightarrow$ Proc(nxt, nxt + 1, rf, im, dm[addr, rf[rd]])
where addr := rf[rs1] + imm

Store Half-Word

Proc((ia, nxt, rf), im, dm)
if im[ia] == Sh(rs1, imm, rd)
$\Longrightarrow$ Proc(nxt, nxt + 1, rf, im, dm[addr, v])
where addr := rf[rs1] + imm *and* v := xor(LogicalAnd(0x1100, dm[addr]), LogicalAnd(0x0011, rf[rd]))

Store Byte
Proc((ia, nxt, rf), im, dm)
if im[ia] == Sb(rs1, imm, rd)
$\Longrightarrow$ Proc(nxt, nxt + 1, rf, im, dm[addr, v])
where addr := rf[rs1] + imm *and* v := xor (LogicalAnd(0x1110, dm[addr]) ,LogicalAnd(0x0001, rf[rd]))

Further DLX models are all in Appendix A.

2.4 The Princeton Model and design principles

The Princeton model was another of the original style models. It has identical data and instruction memories, (Note that the proliferation of instruction and data caches make the Harvard-model assumption that instructions and data are stored in different memories more appropriate.) Having only one memory prevents more than one memory access per clock cycle. Therefore whenever an instruction access data memory (i.e. load or store) it cannot also access the instruction memory in the same cycle. this leads to a resource conflict for the memory.

This problem gives us our first modeling challenge. How do we solve this resource conflict? We solve this by creating the TRS equivalent of a two-state controller. In the first state, we fetch the instruction. In the second, we execute it. Every instruction takes two states (or cycles) to execute.

The Princeton model for the AX ISA, MP_{ax}, is now presented, followed by a sample execution trace to illustrate the execution of a TRS model.

2.5 The Princeton AX Model, MP_{ax}

2.5.1 Definition

In addition to the four state elements from M_{ax}, we add a flag to indicate which state we are in and a register to hold the fetched instruction.

PROC	=	Proc(PC, RF, MEM, INST, FLAG)
MEM	=	Array[ADDR] VI
VI	=	VAL || INST
FLAG	=	fetch || execute

2.5.2 Rules

First comes the instruction fetch state. Here we fetch the current instruction, and toggle the flag. The notation of - indicates that we don't care about that variable in the term.

Rule 0 - Fetch
Proc(pc, rf, mem, -, fetch)

$\Longrightarrow$ Proc(pc, rf, mem, mem[pc], execute)

The rules in the execute state are very similar to the ones in M_{ax}. We simply execute the instruction and increment the pc. The Loadc instruction is the simplest. The value specified in the instruction is stored to the target register.

Rule 1 - Loadc
Proc(pc, rf, mem, inst, execute)
if inst == Loadc(r, v)
$\Longrightarrow$ Proc(pc + 1, rf[r := v], mem, -, fetch)

The Loadpc instruction stores the current pc into the target register.

Rule 2 - Loadpc
Proc(pc, rf, mem, inst, execute)
if inst == Loadpc(r)
$\Longrightarrow$ Proc(pc + 1, rf[r := pc], mem, -, fetch)

The Op instruction adds the values stored in the two operand registers and stores the result in the target register.

Rule 3 - Op
Proc(pc, rf, mem, inst, execute)
if inst == Op(r, r1, r2)
$\Longrightarrow$ Proc(pc + 1, rf[r := v], mem, -, fetch)
where v := rf[r1] + rf[r2]

The Load instruction reads the data memory at the address specified by the contents of register r1 and writes the data to register r. Here, if a load is detected, the instruction is saved to the special register, the flag is set and the pc incremented.

Rule 4 - Load
Proc(pc, rf, mem, inst, execute)
if inst == Load(r, r1)
$\Longrightarrow$ Proc(pc + 1, rf[r := v], mem, -, fetch)
where v := mem[rf[r1]]

The Store instruction stores the value of register reg at the address is data memory specified by register ra. Here, if a store is detected the instruction is saved to the buffer and the flag is set.

Rule 5 - Store
Proc(pc, rf, mem, inst, execute)
if inst == Store(ra, reg)
$\Longrightarrow$ Proc(pc + 1, rf, mem[ad := v], -, fetch)
where ad := rf[ra] *and* v := rf[reg]

If the value of register rc is 0, the branch is taken. In the successful case, pc is set to the contents of register ra. Otherwise the pc is incremented as normal.

Rule 6 - Jz taken
Proc(pc, rf, mem, inst, execute)

if inst == Jz(rc, ra) *and* rf[rc] == 0

$\Longrightarrow$ Proc(rf[ra], rf, mem, -, fetch)

Rule 7 - Jz not taken

Proc(pc, rf, mem, inst, execute)

if inst == Jz(rc, ra) *and* rf[rc] $\neq$ 0

$\Longrightarrow$ Proc(pc + 1, rf, mem, -, fetch)

2.5.3 Example of execution

One might argue: what happens if the flag is set to be true and the instruction in inst is not a load or store? Then there is not a rule that can fire and MP_{ax} will halt. Ensuring that the initial state of the terms has the flag set to false, then the flag will be set to true only concurrently with the instruction buffer being set to a valid load or store.

Initialization of terms is necessary for any simulation or hardware execution, though initial terms are not a part of a TRS. In the case of either hardware or software implementations, an initialization is just contents of the memories, register file and pc (usually set to the first point in the instruction memory.)

An example of MP_{ax} from a initial start state is given as follows. Keep in mind that rules fire atomically and if many rules can fire, one is randomly chosen to fire. Note how the 'modes' toggle back between fetch and execute, with each instruction taking two rules to be executed.

State	Value
pc	0
rf	all zeros
im	im[0] = Loadc(r0, 1)
	im[1] = Loadc(r2, -1)
	im[2] = Loadc(r1, 16)
	im[3] = Load(r3, r1)
	im[4] = Loadpc(r10)
	im[5] = Op(r3, r3, r3)
	im[6] = Jz(r0, r1)
	im[7] = Op(r0, r2, r0)
	im[8] = Jz(r0, r10)
	im[9] = Op(r3, r0, r3)
dm	dm[16] = 5

Figure 2-3: **Example initial state for MP_{ax}**

2.6 Alternatives and discussion

There is an alternative to the two state MP_{ax} just presented. That model always takes two states to execute instructions that can be executed in only one, since there is no resource conflict. This

State					fired
pc	rf	inst	flag	dm	
0	-	-	fetch	-	0
0	-	Loadc(r0, 1)	execute	-	1
1	rf[0] = 1	-	fetch	-	0
1	-	Loadc(r2, -1)	execute	-	1
2	rf[2] = -1	-	fetch	-	0
2	-	Loadc(r1, 16)	execute	-	1
3	rf[1] = 16	-	fetch	-	0
3	-	Load(r3, r1)	execute	-	4
4	rf[3] = 5	-	fetch	-	0
4	-	Loadpc(r10)	execute	-	2
5	rf[10] = 4	-	fetch	-	0
5	-	Op(r3, r3, r3)	execute	-	3
6	rf[3] = 10	-	fetch	-	0
6	-	Jz(r0, r1)	execute	-	7
7	-	-	fetch	-	0
7	-	Op(r0, r2, r0)	execute	-	3
8	rf[0] = 0	-	fetch	-	0
8	-	Jz(r0, r10)	execute	-	6
4	-	-	fetch	-	0
4	-	Loadpc(r10)	execute	-	2
5	rf[10] = 4	-	fetch	-	0
5	-	Op(r3, r3, r3)	execute	-	3
6	rf[3] = 20	-	fetch	-	0
6	-	Jz(r0, r1)	execute	-	6
16	-	-	fetch	-	-

Figure 2-4: **Example execution trace for MP_{ax}** The current state is shown in the first four columns, followed by the rule that fires. The next state is shown in the following line of the table.

version, $MP - alt_{ax}$, instead tries to execute every instruction completely, and breaks into two cycle mode only when confronted with a load or store.

2.6.1 Definition

In addition to the four state elements from M_{ax}, we add a flag to indicate which state we are in and a register to hold the fetched instruction.

PROC	=	Proc(PC, RF, MEM, INST, FLAG)
FLAG	=	regular || special
MEM	=	ϵ || Mem(ADDR, VI);MEM
VI	=	VAL || INST

2.6.2 Rules

The Loadc instruction is the simplest. The value specified in the instruction is stored to the target register. The pc is incremented to proceed with linear program execution. Here we introduce the notation of - to indicate don't care in the term.

Rule 1 - Loadc
Proc(pc, rf, mem, -, regular)
if mem[pc] == Loadc(r, v)
$\Longrightarrow$ Proc(pc + 1, rf[r, v], mem, -, regular)

The Loadpc instruction stores the current pc into the target register.

Rule 2 - Loadpc
Proc(pc, rf, mem, -, regular)
if mem[pc] == Loadpc(r)
$\Longrightarrow$ Proc(pc + 1, rf[r, pc], mem, -, regular)

The Op instruction adds the values stored in the two operand registers and stores the result in the target register.

Rule 3 - Op
Proc(pc, rf, mem, -, regular)
if mem[pc] == Op(r, r1, r2)
$\Longrightarrow$ Proc(pc + 1, rf[r, v], mem, -, regular)
where v := rf[r1] + rf[r2]

The Load instruction reads the data memory at the address specified by the contents of register r1 and writes the data to register r. Here, if a load is detected, the instruction is saved to the special register, the flag is set to special.

Rule 4a - Load
Proc(pc, rf, mem, -, regular)
if mem[pc] == Load(r, r1)
$\Longrightarrow$ Proc(pc, rf, mem, inst, special)

where inst := Load(r, r1)

In the second step, if the flag is set to special, the pc is ignored and the instruction in the buffer is executed. The flag is then set to regular, the pc is incremented and execution will continue as normal.

Rule 4b - Load

Proc(pc, rf, mem, inst, special)

if inst == Load(r, r1)

⟹ Proc(pc + 1, rf[r, v], mem, -, regular)

where v := mem[rf[r1]]

The Store instruction stores the value of register reg at the address is data memory specified by register ra. Here, if a store is detected the instruction is saved to the buffer and the flag is set to special.

Rule 5a - Store

Proc(pc, rf, mem, -, regular)

if mem[pc] == Store(ra, reg)

⟹ Proc(pc, rf, mem, inst, special)

where inst := store(ra, reg)

In the second step, the store is executed and the flag reset to false.

Rule 5b - Store

Proc(pc, rf, mem, inst, special)

if inst == Store(ra, reg)

⟹ Proc(pc + 1, rf, mem[ad, v], -, regular)

where ad := rf[ra] *and* v := rf[reg]

If the value of register rc is 0, the branch is taken. In the successful case, pc is set to the contents of register ra. Otherwise the pc is incremented as normal.

Rule 6 - Jz taken

Proc(pc, rf, mem, -, regular)

if mem[pc] == Jz(rc, ra) *and* rf[rc] == 0

⟹ Proc(rf[ra], rf, mem, -, regular)

Rule 7 - Jz not taken

Proc(pc, rf, mem, -, regular)

if mem[pc] == Jz(rc, ra) *and* rf[rc] ≠ 0

⟹ Proc(pc + 1, rf, mem, -, regular)

2.7 Summary

With these first simple models we have laid the foundation for future work. M_{ax}, M_{dlx}, MP_{ax} are very short and elegant descriptions of simple implementations of the two instruction sets. The next chapters add complexity in both modeling techniques and hardware concepts. MP_{ax} presented an

important idea - breaking the functionality of one rule into many and using intermediate storage. The obvious next step to take is to pipeline these two states (or stages) if the model is to become more efficient. This idea will be expanded to deal with pipelining in Chapter 3.

Chapter 3

Simple Pipelined models

In this section we introduce the technique for pipelining TRS models. The fact that processors are frequently pipelined makes comprehension of this method easy, but it can be applied to any type of model, not just one that we would normally think of as being pipelined. We begin by discussing general principles and then describe three pipelined models, followed by discussion of pipelining strategies.

3.1 Pipelining in hardware and TRS models

In hardware, pipelining seeks to increase parallelism by exploiting idle functional units. In a single cycle circuit, most the circuit lies idle at any given point. By breaking the circuit into stages separated by registers, multiple instructions (or input) can be executed in parallel in a lockstep fashion on the circuit. Though this cannot decrease the latency of a single instruction through the circuit (and in general increases latency because the clock cycle must be long enough for the longest-latency part to complete) it does dramatically increase the throughput of the circuit. In the best case where there are no hazards between stages, throughput increases from 1 to the number of pipeline stages.

In a TRS model, pipelining similarly tries to break large computations, usually expressed in a rule's where clause, into smaller units across many rules. In order to do this, state elements to maintain the intermediate stages must be created. We have modeled these with queues. For convenience of notation, not semantic necessity, we have modeled the queues as unbounded in size. In practice, only bounded size queues can be implemented.

Though any hardware implementation or simulation of a TRS written with unbounded queues will be correct but not complete. Some behavior will be lost, but none new will be gained. For example, supposed we bound queues to be of length three. It is possible that a TRS execution on a given state could have more than three elements. Our simulation will not capture these behaviors,

but will capture all those execution traces that never exceeded the queues bounds. In simulation we are interested in implementing one possible execution order, not all of them.

Since multiple rules check the same queue, deadlock is a consideration. Deadlock will not occur in a pipelined TRS if the pipelines don't have circular dependencies and if no rule examines more than the top element of the FIFO. Specifically, we have only forward dependences (i.e. a specific stage does not depend on the behavior of a previous one) and only examine the heads of the queues.

The general pipelining strategy is to decide where to make the breaks, and insert queues in between them. Each stage them reads from its input queue and must forward the instruction with any new state to the next stage, meeting all requirements. To be correct, the model must be equivalent to an unpipelined version. Though the proof is omitted here, intuitively the pipeline must not have forward dependencies preventing instructions from draining completely.

Another hardware similarity is the choice between resolving RAW data hazards by stalling or by bypassing. Stalling means waiting until the register is written to and then proceeding; bypassing means finding the new value as soon as it appears in the pipeline and using that value (bypassing the writeback wiring.) These choices are written in to the TRS itself. Therefore two variations on $Mpipe_{ax}$ are presented.

3.2 New types for TRSs

For storing information between stages, queues are used. This is the standard first-in-first-out (FIFO) buffer. For its definition, another abstract data type is used. A queue is an ordered collection of elements, concatenated with ;'s. A queue of two elements e_1, e_2 is written as $e_1; e_2$ In our notation e_i can be either have the type of a single element or a queue of that type of element. Enqueueing an element e to the end of a queue q is written as $q; e$. Removing an element from the head of the queue $e; q$ leaves q as the queue. Queues here are modeled as having unbounded length. Note that a valid queue can be empty. ELEM is whatever type the queue needs to contain.

Also needed now is a buffer for storing instructions in the queue: the instruction buffer. This buffer holds an instruction and address. Later on we will introduce different buffers to deal with increased complexity.

$$\text{IB} \quad = \quad \text{Ib(ADDR, INST)}$$

As instructions move through the pipeline, the register names become values. Therefore, a few modifications to the previous definition of INST (see Section 1.5) are necessary. First, the standard instructions can now hold RNAMEs or VALs. Second, a new instruction is introduced to represent writing single value to a register. This instruction is called Reqv (or "r equals v"). The new definition if INST is as follows:

```
INST   = Loadc(RNAME, VAL) || Loadpc(RNAME)||
         Op(RNAME, RV, RV) || Load(RNAME, RV)||
         Store(RV, RV) || Jz(RV, RV) || Reqv(RNAME, VAL)
RV     = RNAME || VAL
RNAME  = Reg0 || Reg1 || Reg2 || .. Reg31
VAL    = Bit[32]
```

3.3 The Stall Pipelined AX Model, $Mpipe_{ax}$

In $Mpipe_{ax}$, the standard choice was made to break the instruction execution into five stages. These five stages are the instruction fetch, where the instruction memory is read at the pc value; the decode stage, where the register file is read and instruction type determined; the execute stage, where arithmetic operations and other tests are performed; the memory stage, where the data memory is either read or written; the writeback stage, where the register file is written to.

In this stall version, instructions cannot move through the Decode stage until there are no RAW hazards (i.e. no instruction farther along the pipeline writes to a register that needs to be read.)

3.3.1 Definition

```
PROC               = Proc(PC, RF, BSD, BSE, BSM, BSW, IM, DM)
BSD, BSE, BSM, BSW = Queue(IB)
```

3.3.2 Rules

In the Fetch stage the next instruction is fetched, added to the bsD queue, and the pc is incremented. When Jz's are fetched, the pc is still incremented instead of stalling until the branch target is determined. (This is a passive form of speculative execution, which is discussed in Chapter 4.)

Rule 1 - Fetch

Proc(ia, rf, bsD, bsE, bsM, bsW, im, dm)

⟹ Proc(ia+1, rf, bsD;Ib(ia, inst), bsE, bsM, bsW, im, dm)

where inst := im[ia]

In the decode stage the different instruction types are determined. In this stall version, the registers to be read from the register file are checked against those to be written in later queues. The decode rules fire only if there are no RAW hazards.

Rule 2a - Decode Op

Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)

if inst1 == Op(r, r2, r3) ***and*** r2, r3 are not dests in (bsE, bsM, bsW)

⟹ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)

where inst2 := Op(r, rf[r2], rf[r3])

Rule 2b - Decode Loadc

Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)

if inst1 == Loadc(r, v)

⟹ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)

where inst2 := Reqv(r,v)

Rule 2c - Decode Loadpc
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Loadpc(r)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Reqv(r, sia)

Rule 2d - Decode Load
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Load(r, r1) *and* r1 is not dest in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Load(r, rf[r1])

Rule 2e - Decode Store
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Store(r1, r2) *and* r1, r2 are not dests in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Store(rf[r1], rf[r2])

Rule 2f - Decode Jz
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Jz(r1, r2) *and* r1, r2 are not dests in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Jz(rf[r1], rf[r2])

In the execute stage Op instructions undergo the equivalent of ALU use, and Jz's are resolved. If a Jz is taken, the pc is reset and the now-invalid queues bsD and bsE are flushed, otherwise the failed Jz is discarded. Memory instructions and already completed value determinations (loadc and loadpc) are passed on to the next stage untouched.

Rule 3 - Exec Op
Proc(ia, rf, bsD, Ib(sia, Op(r, v1, v2);bsE, bsM, bsW, im, dm)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE, bsM;Ib(sia, ReqV(r, v)), bsW, im, dm)
where v := Op applied to (v1, v2)

Rule 4 - Exec Jz taken
Proc(ia, rf, bsD, Ib(sia, Jz(0, nia);bsE, bsM, bsW, im, dm)
$\Longrightarrow$ Proc(nia, rf, ϵ, ϵ, bsM, bsW, im, dm)

Rule 5 - Exec Jz not taken
Proc(ia, rf, bsD, Ib(sia, Jz(v, -);bsE, bsM, bsW, im, dm)
if $v \neq 0$
$\Longrightarrow$ Proc(ia, rf, bsD, bsE, bsM, bsW, im, dm)

Rule 6 - Exec Copy
Proc(ia, rf, bsD, Ib(sia, it);bsE, bsM, bsW, im, dm)
if it $\neq$ Op(-, -, -) or Jz(-, -)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE, bsM;Ib(sia, it), bsW, im, dm)

In the memory stage, data memory is accessed by Load and Store instructions. All other instructions (which have the form r=v) are passed on to the writeback stage.

Rule 7 - Mem Load
sys(Proc(ia, rf, bsD, bsE, Ib(sia, Load(r, a));bsM, bsW, im), pg, dm)

$\Longrightarrow$ sys(Proc(ia, rf, bsD, bsE, bsM, bsW;Ib(sia, Reqv(r, v)), im), pg, dm)
where v := dm[a]

Rule 8 - Mem Store
sys(Proc(ia, rf, bsD, bsE, Ib(sia, Store(a, v));bsM, bsW, im), pg, dm)
$\Longrightarrow$ sys(Proc(ia, rf, bsD, bsE, bsM, bsW, im), pg, dm[a:=v])

Rule 9 - Mem Copy
Proc(ia, rf, bsD, bsE, Ib(sia, ReqV(r, v));bsM, bsW, im, dm)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE, bsM, bsW;Ib(sia, Reqv(r, v)), im, dm)

In the final writeback stage, values, determined by Op, Loadc, Loadpc or Load instructions, are written to the register file.

Rule 10 - Writeback
Proc(ia, rf, bsD, bsE, bsM, Ib(sia, Reqv(r, v));bsW, im, dm)
$\Longrightarrow$ Proc(ia, rf[r := v], bsD, bsE, bsM, bsW, im, dm)

3.3.3 Example of execution

An example of M_{ax} from a initial start state is given as follows. Keep in mind that rules fire atomically and if many rules can fire, one is randomly chosen to fire. As displayed in the trace, this pipelined version does not actually exhibit the standard lock-step progression of instructions through the stages. To do this in simulation we need to fire multiple rules at once. This will be discussed in Chapter 5.

State	Value
pc	0
rf	all zeros
im	im[0] = Loadc(r0, 1)
	im[1] = Loadc(r2, -1)
	im[2] = Loadc(r1, 16)
	im[3] = Load(r3, r1)
	im[4] = Loadpc(r10)
	im[5] = Op(r3, r3, r3)
	im[6] = Jz(r0, r1)
	im[7] = Op(r0, r2, r0)
	im[8] = Jz(r0, r10)
	im[9] = Op(r3, r0, r3)
dm	dm[16] = 5
bsD	-
bsE	-
bsM	-
bsW	-

Figure 3-1: **Example initial state for $Mpipe_{ax}$**

State						Can fire	Did
pc	rf	bsD	bsE	bsM	bsW		
4		Ld(r3,r1);Ldc(r1,16)	r2=-1	-	r0=1	1, 2b, 6, 10	2
4		Ld(r3,r1)	r1=16; r2=-1	-	r0=1	1, 6, 10	6
4		Ld(r3,r1)	r1=16	r2=-1	r0=1	1, 6, 9, 10	1
5		Ldpc(r10);Ld(r3,r1)	r1=16	r2=-1	r0=1	1, 6, 9, 10	10
5	r[0]=1	Ldpc(r10);Ld(r3,r1)	r1=16	r2=-1		1, 6, 9	9
5		Ldpc(r10);Ld(r3,r1)	r1=16		r2=-1	1, 6, 10	6
5		Ldpc(r10);Ld(r3,r1)	-	r1=16	r2=-1	1, 9, 10	10
5	r[2]=-1	Ldpc(r10);Ld(r3,r1)	-	r1=16	-	1, 9	9
5		Ldpc(r10);Ld(r3,r1)	-	-	r1=16	1, 10	10
5	r[1]=16	Ldpc(r10);Ld(r3,r1)	-	-	-	1, 2d	2
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
9		Jz(r0,r10)	-	r0=0	r3=10	1, 9, 10	9
9		Jz(r0,r10)	-	-	r0=0;	1, 10	1
–		more of previous line	-	-	r3=10		
10		Op(r3,r0,r3);Jz(r0,r10)	-	-	r0=0;	1, 10	10
–		more of previous line	-	-	r3=10		
10	r[3]=10	Op(r3,r0,r3);Jz(r0,r10)	-	-	r0=0	1, 10	10
10	r[0]=0	Op(r3,r0,r3);Jz(r0,r10)	-	-	-	1, 2f	2
10		Op(r3,r0,r3)	Jz(0,4)	-	-	1, 2f, 4	2
10		-	r3=Op(0,10);Jz(0,4)	-	-	1, 4	4
4		-	-	-	-	1	1
5		Ldpc(r10)	-	-	-	1, 2c	2

Figure 3-2: **Example execution trace for** $Mpipe_{ax}$ The current state is shown in the first four columns, followed by the triggered rules and single rule that fires (the rule to fire is chosen randomly.) The next state is shown in the following line of the table. Note that r:=v is used as a shorthand for Reqv(r, v) to save space. The first window shows stalling occurring. Rule 2 is not triggered because r1, which the Load instruction reads, is being written to farther on in the pipeline by the Loadc instruction. The second window shows the reset of the pc and flush of earlier pipeline stages when the Jz resolves. *Note that some instruction names have been abbreviated to save space. Two lines have been split over multiple rows also.*

3.4 The Bypass Pipelined AX Model, $Mbyp_{ax}$

This model is the same as the previous, with the exception of the six Decode stage rules. These rules have been changed to implement bypass instead of stall. The definition and other rules remain the same are are not repeated.

3.4.1 Definition

The bypassed version has the same term definition as the stalled as is not repeated here.

INST	=	Loadc(RNAME, VAL) || Loadpc(RNAME)||
		Op(RNAME, RV, RV) || Load(RNAME, RV)||
		Store(RV, RV) || Jz(RV, RV) || Reqv(RNAME, VAL)
RV	=	RNAME || VAL
RNAME	=	Reg0 || Reg1 || Reg2 || .. Reg31
VAL	=	Bit[32]

3.4.2 New Rules

To determine how to bypass, we must figure out two things. First, at which pipeline stage are the values of the source register needed? In a stall model all operands are read from the register file in the decode stage. The instructions vary, however, on when the operands are actually used. Op and Jz use them in the execute stage. Load uses them in the Memory stage and Store in the Memory and Writeback stage. Loadc and Loadpc have no register operands. Therefore Op cannot proceed from the decode stage without both operands. Load and Store can proceed to the Execute stage but no farther without operands.

Second, when are new values for registers produced? Loadc and Loadpc produce the values in the decode stage (since the values are encoded in the instruction.) Op produces a value in the Execute stage. Load produces a value in the Memory stage. Jz and Store produce no values. Because we have written the model such that whenever a value for a register is computed, the instruction in converted to r = v, we simply look down the pipeline for the newest r = v statement for the register we want.

The rules for Loadc and Loadpc remain the same since they have no need for bypassing.

Rule 2b - Decode Loadc
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Loadc(r, v)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Reqv(r, v)

Rule 2c - Decode Loadpc
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Loadpc(r)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := ReqV(r, sia)

The four bypassed instructions have the additional rules below. Note that due to the non-deterministic execution of TRS models, we cannot let any instruction get past the Decode stage without having values for all its register operands. This is because even if a r = v for the register operand will appear after another rule fires, it can then go through the pipeline and disappear before the bypass rule can be triggered!

For ease of rule writing, we define x $\in_1$ y to be the first occurrence of an item matching x's type occurring in y, which is ordered. Not that for two operand we have four possibilities for the registers being current in the register file.

Rule 2a-1 - Decode Op, no bypass
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Op(r1, r2, r3) *and* r2, r3 are not dests in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Op(r1, rf[r2], rf[r3])

Rule 2a-2 - Decode Op, bypass r2
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Op(r1, r2, r3) *and* r2 == v2 $\in_1$ bsE;bsM;bsW *and* r3 == v3 is not dest in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Op(r1, v2, rf[r3])

Rule 2a-3 - Decode Op, bypass r3
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Op(r1, r2, r3) *and* r3 == v3 $\in_1$ bsE;bsM;bsW *and* r2 == v2 is not dest in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Op(r1, rf[r2], v3)

Rule 2a-4 - Decode Op, bypass both
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Op(r1, r2, r3) *and* r2 == v2 $\in_1$ bsE;bsM;bsW *and* r3 == v3 $\in$ bsE;bsM;bsW
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Op(r1, v2, v3)

Rule 2d-1 - Decode Load, no bypass
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Load(r, r1) *and* r1 is not dest in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Load(r, rf[r1])

Rule 2d-2 - Decode Load, bypass
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Load(r, r1) *and* r1 == v1 $\in_1$ bsE;bsM;bsW
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Load(r, v1)

Rule 2e-1 - Decode Store, no bypass
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Store(r1, r2) *and* r1, r2 are not dests in (bsE, bsM, bsW)
$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Store(rf[r1], rf[r2])

Rule 2e-2 - Decode Store, bypass r1
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)

if inst1 == Store(r1, r2) *and* r1 == v1 $\in_1$ bsE;bsM;bsW *and* r2 == v2 is not dest in bsE;bsM;bsW

$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Store(v1, rf[r2])

Rule 2e-2 - Decode Store, bypass r2
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Store(r1, r2) *and* r2 == v2 $\in_1$ bsE;bsM;bsW *and* r1 == v1 is not dest in bsE;bsM;bsW

$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Store(rf[r1], v2)

Rule 2e-4 - Decode Store, bypass both
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Store(r1, r2) *and* r1 == v1 $\in_1$ bsE;bsM;bsW *and* r2 == v2 $\in_1$ bsE;bsM;bsW

$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Store(v1, v2)

Rule 2f-1 - Decode Jz, no bypass
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Jz(r1, r2) *and* r1, r2 are not dests in (bsE, bsM, bsW)

$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Jz(rf[r1], rf[r2])

Rule 2f-2 - Decode Jz, bypass r1
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Jz(r1, r2) *and* r1 == v1 $\in_1$ bsE;bsM;bsW *and* r2 == v2 is not dest in bsE;bsM;bsW

$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Jz(v1, rf[r2])

Rule 2f-3 - Decode Jz, bypass r2
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Jz(r1, r2) *and* r2 == v2 $\in_1$ bsE;bsM;bsW *and* r1 == v1 is not dest in bsE;bsM;bsW

$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Jz(rf[r1], v2)

Rule 2f-4 - Decode Jz, bypass both
Proc(ia, rf, Ib(sia, inst1);bsD, bsE, bM, bsW, im, dm)
if inst1 == Jz(r1, r2) *and* r1 == v1 $\in_1$ bsE;bsM;bsW *and* r2 == v2 $\in_1$ bsE;bsM;bsW

$\Longrightarrow$ Proc(ia, rf, bsD, bsE;Ib(sia, inst2), bsM, bsW, im, dm)
where inst2 := Jz(v1, v2)

3.4.3 Example of execution

An example of M_{ax} from a initial start state is given as follows. Keep in mind that rules fire atomically and if many rules can fire, one is randomly chosen to fire.

3.5 Summary

In this Chapter we explored pipelining and separating functionality in TRS models. $Mpipe_{ax}$ was a standard stalled pipeline. $Mbyp_{ax}$ was a bypassed version. In the next Chapter we expand the

State	Value
pc	0
rf	all zeros
im	im[0] = Loadc(r0, 1)
	im[1] = Loadc(r2, -1)
	im[2] = Loadc(r1, 16)
	im[3] = Load(r3, r1)
	im[4] = Loadpc(r10)
	im[5] = Op(r3, r3, r3)
	im[6] = Jz(r0, r1)
	im[7] = Op(r0, r2, r0)
	im[8] = Jz(r0, r10)
	im[9] = Op(r3, r0, r3)
dm	dm[16] = 5
bsD	-
bsE	-
bsM	-
bsW	-

Figure 3-3: **Example initial state for** $Mbyp_{ax}$

State						Can Fire	Did
pc	rf	bsD	bsE	bsM	bsW		
4		Load(r3,r1);Loadc(r1,16)	r2=-1	-	r0=1	1, 2b, 6, 10	2
4		Load(r3,r1)	r1=16; r2=-1	-	r0=1	1, 2d, 6, 10	6
4		-	r3=Load(16);r1=16	r2=-1	r0=1	1, 2d, 6, 9, 10	2d
4		-	r3=Load(16);r1=16	r2=-1	r0=1	1, 6, 9, 10	1
5		Loadpc(r10)	r3=Load(16);r1=16	r2=-1	r0=1	1, 2c, 6, 9, 10	10
5	r[0]=1	Loadpc(r10)	r3=Load(16);r1=16	r2=-1		1, 2c, 6, 9	9

Figure 3-4: **Example execution trace for** $Mbyp_{ax}$ The current state is shown in the first four columns, followed by the triggered rules and single rule that fires. The next state is shown in the following line of the table. Note that r:=v is used as a shorthand for Reqv(r, v) to save space.The first window shows bypassing occurring on the Load instruction. Compare this with the execution example for $Mpipe_{ax}$

use of queues as communication mechanisms between modules instead of within the same (single) module here.

Chapter 4

Complex models

In this chapter we discuss using modularity in TRSs, and use it to model more complex hardware designs. Part of the benefit of modularity is conciseness in writing out models. Another part is the standard benefits of modularity in software design - it allows for the clean interfacing between separately developed parts and isolates changes. This allows current research (e.g. the cache coherence work done by Xiaowei Shen, Arvind and Larry Rudolph [5]) to be simply plugged in to these processor models. Then two important but orthogonal concepts are discussed: speculative execution and register renaming.

Speculative execution seeks to reduce the delay penalty incurred while waiting for a branch target to be resolved in a pipeline. As pipeline length increases, the delay until branch target resolution increases as well. Because many program structures, such as loops and function calls, exhibit regular behavior we can exploit this regularity to predict the future.

First the method of speculation and correction of errors must be defined. This is discussed more in depth and proofs of correctness are given in [2]. In the fetch stage the branch target buffer (described below) is consulted and a predicted target for a jump is used as the next pc. When the branch resolves later on in the pipeline, any incorrect execution must be fixed. If the prediction was correct, nothing happens. If the prediction was incorrect, all instructions behind the branch in the pipeline must be flushed. Because the branch is resolved before and state is changed there are no worries about fixing state.

Secondly, the method of prediction needs to be defined. The method (and accuracy) of prediction has no bearing on the overall correctness of the speculative execution. Though a bad prediction method will generate more inefficient execution, it will not generate incorrect execution. Common methods are discussed in Section 4.2.

Register renaming stems from a different hardware constraint. As multiple functional units are introduced (e.g. floating-point units, multiple ALUs) instructions can complete out-of-order.

Data hazards place constraints on whether instructions can be issued. The number of register also nominally constrains the number of instructions simultaneously active: since two instructions cannot change the same register, the number of register limits the number of instructions that can execute.

Register renaming works around this problem, but has a corresponding increase in effort elsewhere. Each time an instruction is issued, its target register is assign a tag, and its operands are read as either a tag or real value from the register file. Note that an operand having a tag instead of a value indicates an instruction writing that register has not yet resolved. When all operands have values instead of tags, the instruction can be executed. Upon completion the target tag is updated in all following instructions to be of the new value. A more in depth discussion of this can be found in [4].

4.1 New types for TRSs

Here the two new modules are defined. For speculative execution the BTB stores the predictions for branch targets based on pc. It is a memory using the Array abstract type.

BTB	=	Array[ADDR] Entry(BITS ADDR)
BITS	=	Bit[32]

Operations on the btb are defined similarly to for memory and have the following semantics. Looking up an instruction address returns a predicted address. If the the instruction address ia is not in the btb, $ia + 1$ is returned. If ia is in the BTB, $ia + 1$ is returned if the prediction is 'not taken' and pia (the predicted target) is returned if the prediction is 'taken'. Note that in practice BTBs are implemented using some hash of the provided address.

Updating the BTB is done by providing it with the instruction address it predicted for, the predicted address and the result (correct or not correct) of the prediction. Note that since we are creating an interface here and not specifying either the internal methods of storage in the BTB or the method of prediction (which are discussed in Section 4.2.

For the register renaming model the ROB (re-order buffer) is introduced. It is an ordered list that we can access both the head and tail of (like a queue) as well as scanning the contents of. The templates inside hold all the necessary information about each instruction, including the tag of the target register and the state of completion. Details are discussed more fully in Section 4.5.

ROB	=	List IRB	*Re-Order Buffer*
IRB	=	Itb (TAG, PC, INSTTEMP, STATE)	*Element of ROB*

4.2 Branch Prediction Schemes

In order for speculative execution to be effective, the prediction must be highly accurate. There are several different schemes possible for branch prediction. In the models in this chapter we have

chosen to model the branch target buffer as a black box.

4.2.1 Branch Prediction Schemes

Here two prediction methods are presented. It is important to note that the correctness of speculative execution is independent of the prediction scheme chosen. The two described below (and many more that are possible) differ in performance, not correctness.

1-bit prediction Rules

This is the simplest scheme. It uses the heuristic that the most likely branch target is the target from the most recent execution of the instruction. If the last instance of the jump at ia was taken, the last branch target is predicted. If the last instance was not taken, $ia + 1$ is predicted.This prediction scheme provides good predictive accuracy for long loops.

2-bit prediction Rules

The 2-bit method is another common approach to branch prediction. This method uses the information from the previous two jumps instead of previous one to predict the outcome.

The two-bit name comes from the fact that the two bits represent the behavior of the last two occurrences. Two states represent sure predictions of taken or not taken. Two others represent marginal confidence.

This is shown in Figure 4-1.

4.3 The Speculative AX Model, $Mspec_{ax}$

4.3.1 Definition

Because we are speculating, we need to keep track of what the speculated address was so that it can be checked later on in the pipeline. To do this we modify the buffer to hold three elements, not two.

$$\begin{array}{rcl} \text{PROC} & = & \text{Proc(PC, RF, BSD, BSE, BSM, BSW, IM, DM, BTB)} \\ \text{BSD, BSE, BSM, BSW} & = & \text{Queue(ITB)} \\ \text{ITB} & = & \text{Itb(ADDR, ADDR, INST)} \end{array}$$

4.3.2 Rules

This model introduces speculative, in-order execution. Only two significant changes are necessary from $Mpipe_{ax}$: modify the fetch stage to speculate and modify the jump resolution rules to correct any mistakes. First, Jz instructions must be speculated on in the fetch stage. Here, the distinction is made between Jz and non-Jz instructions to reduce size of the btb. Fetch of a Jz necessitates

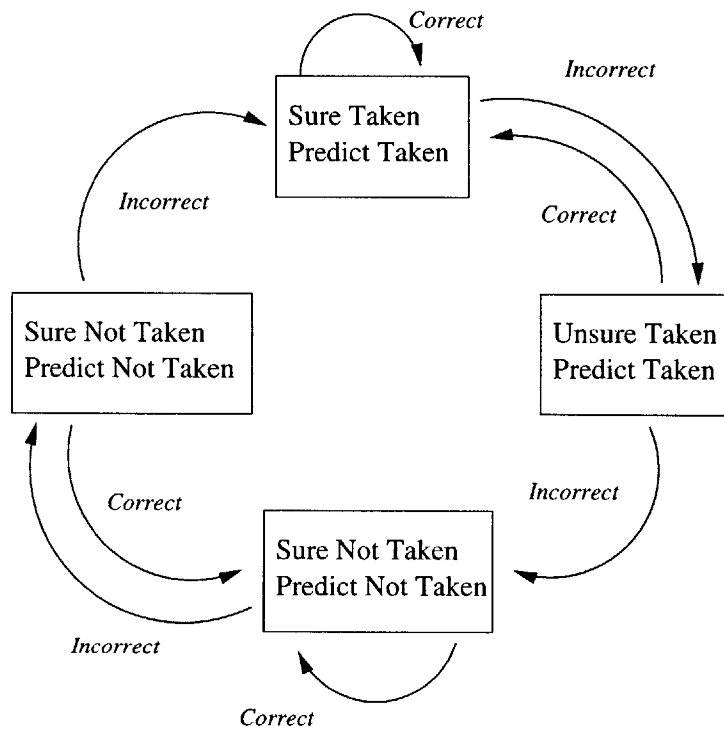


Figure 4-1: **2-bit prediction scheme** Note how it takes two mispredictions in a row to change the next prediction.

consultation with the btb, which produces the next pc. Fetches on other instructions proceed as normal.

Rule 1a - Fetch Jz
Proc(ia, rf, bsD, bsE, bsM, bsW, im, dm, btb)
if inst == Jz(-,-)
⟹ Proc(nia, rf, bsD;Itb(ia, nia, inst), bsE, bsM, bsW, im, dm, btb)
where inst := im[ia], nia := lookup(btb, ia)

Rule 1b - Fetch, not Jz
Proc(ia, rf, bsD, bsE, bsM, bsW, im, dm, btb)
if inst ≠= Jz(-,-)
⟹ Proc(ia+1, rf, bsD;Itb(ia, ia+1, inst), bsE, bsM, bsW, im, dm, btb)
where inst := im[ia]

Rule 2a - Decode Op
Proc(ia, rf, IB(sia, -, inst1);bsD, bsE, bM, bsW, im, dm, btb)
if inst1 == Op(r1, r2, r3) *and* r2, r3 are not dests in (bsE, bsM, bsW)
⟹ Proc(ia, rf, bsD, bsE;Itb(sia, -, inst2), bsM, bsW, im, dm, btb)
where inst2 := Op(r1, rf[r2], rf[r3])

Rule 2b - Decode Loadc
Proc(ia, rf, Itb(sia, -, inst1);bsD, bsE, bM, bsW, im, dm, btb)
if inst1 == Loadc(r, v)
⟹ Proc(ia, rf, bsD, bsE;Itb(sia, -, inst2), bsM, bsW, im, dm, btb)
where inst2 := Reqv(r, v)

Rule 2c - Decode Loadpc
Proc(ia, rf, Itb(sia, -, inst1);bsD, bsE, bM, bsW, im, dm, btb)
if inst1 == Loadpc(r)
⟹ Proc(ia, rf, bsD, bsE;Itb(sia, -, inst2), bsM, bsW, im, dm, btb)
where inst2 := Reqv(r, sia)

Rule 2d - Decode Load
Proc(ia, rf, Itb(sia, -, inst1);bsD, bsE, bM, bsW, im, dm, btb)
if inst1 == Load(r1, r2) *and* r2 is not dest in (bsE, bsM, bsW)
⟹ Proc(ia, rf, bsD, bsE;Itb(sia, -, inst2), bsM, bsW, im, dm, btb)
where inst2 := Load(r1, rf[r2])

Rule 2e - Decode Store
Proc(ia, rf, Itb(sia, -, inst1);bsD, bsE, bM, bsW, im, dm, btb)
if inst1 == Store(r1, r2) *and* r1, r2 are not dests in (bsE, bsM, bsW)
⟹ Proc(ia, rf, bsD, bsE;Itb(sia, -, inst2), bsM, bsW, im, dm, btb)
where inst2 := Store(rf[r1], rf[r2])

Rule 2f - Decode Jz
Proc(ia, rf, Itb(sia, -, inst1);bsD, bsE, bM, bsW, im, dm, btb)
if inst1 == Jz(r1, r2) *and* r1, r2 are not dests in (bsE, bsM, bsW)
⟹ Proc(ia, rf, bsD, bsE;Itb(sia, -, inst2), bsM, bsW, im, dm, btb)
where inst2 := Jz(rf[r1], rf[r2])

The second change occurs in the execution stage. In rules 4a,b and 5a,b, the correct Jz target is determined and two things must occur - the state must be made correct (e.g. wipe any invalid instructions, reset pc) and the btb updated to improve future predictive performance. In Rules 4a and 5a, the prediction is correct, so only the btb needs to be updated. In rules 4b and 5b, the

prediction was wrong, so bsD and bsE are flushed, the pc set to the correct value and btb updated.

Rule 3 - Exec Op
Proc(ia, rf, bsD, Itb(sia, -, Op(r, v1, v2);bsE, bsM, bsW, im, dm, btb)
⟹ Proc(ia, rf, bsD, bsE, bsM;Itb(sia, -, Reqv(r, v)), bsW, im, dm, btb)
where v := Op applied to (v1, v2)

Rule 4a - Exec Jz taken correct
Proc(ia, rf, bsD, Itb(sia, pia, Jz(0, nia));bsE, bsM, bsW, im, dm, btb)
if nia == pia
⟹ Proc(ia, rf, bsD, bsE, bsM, bsW, im, dm, btb')
where btb' := update(btb, sia, nia, correct)

Rule 4b - Exec Jz taken incorrect
Proc(ia, rf, bsD, Itb(sia, pia, Jz(0, nia));bsE, bsM, bsW, im, dm, btb)
if nia ≠= pia
⟹ Proc(nia, rf, e, e, bsM, bsW, im, dm, btb')
where btb' := update(btb, sia, nia, incorrect)

Rule 5a - Exec Jz not taken, correct
Proc(ia, rf, bsD, Itb(sia, pia, Jz(v, -));bsE, bsM, bsW, im, dm, btb)
if v ≠= 0 *and* pia == sia + 1
⟹ Proc(ia, rf, bsD, bsE, bsM, bsW, im, dm, btb')
where btb' := update(btb, sia, nia, correct)

Rule 5b - Exec Jz not taken incorrect
Proc(ia, rf, bsD, Itb(sia, pia, Jz(v, -));bsE, bsM, bsW, im, dm, btb)
if v |*not* == 0 *and* pia !== sia + 1
⟹ Proc(sia+1, rf, e, e, bsM, bsW, im, dm, btb')
where btb' := update(btb, sia, nia, incorrect)

Rule 6 - Exec Copy
Proc(ia, rf, bsD, Itb(sia, -, it);bsE, bsM, bsW, im, dm, btb)
if it ≠= op or Jz
⟹ Proc(ia, rf, bsD, bsE, bsM;Itb(sia, -, it), bsW, im, dm, btb)

Rule 7 - Mem Load
Proc(ia, rf, bsD, bsE, Itb(sia, -, Load(r, a));bsM, bsW, im, dm, btb)
⟹ Proc(ia, rf, bsD, bsE, bsM, bsW;Itb(sia, -, Reqv(r, v)), im, dm, btb)
where v := dm[a]

Rule 8 - Mem Store
Proc(ia, rf, bsD, bsE, Itb(sia, -, Store(a, v));bsM, bsW, im, dm, btb)
⟹ Proc(ia, rf, bsD, bsE, bsM, bsW, im, dm[a:=v], btb)

Rule 9 - Mem Copy
Proc(ia, rf, bsD, bsE, Itb(sia, -, Reqv(r, v));bsM, bsW, im, dm, btb)
⟹ Proc(ia, rf, bsD, bsE, bsM, bsW;Itb(sia, Reqv(r, v)), im, dm, btb)

Rule 10 - Writeback
Proc(ia, rf, bsD, bsE, bsM, Itb(sia, -, Reqv(r, v));bsW, im, dm, btb)
⟹ Proc(ia, rf[r:=v], bsD, bsE, bsM, bsW, im, dm, btb)

4.4 Register Renaming and Multiple Functional Units

This model represents two major changes from the previous ones. First, it has several functional units. This modularity is useful not only in accurately modeling current designs, but in providing an easily changeable model. Secondly, it uses register renaming.

In this register renaming model, instructions continuously enter the reorder buffer (ROB) Upon entry, all registers are looked up in the buffer. For the operands, if a register is not being written to already, the value of that register is assign. Otherwise the tag (or new name) that is being used by a previous operation is assigned. Destination register are assigned fresh tags. Whenever an instruction has values for all of its operands, it is dispatched to a functional unit. When a value is received for the destination register, that value is propagated through the ROB to previous entries.

Branch resolution in the ROB is more complicated then in a pipelined model. Just as before, the pc needs to be reset and all instruction following the branch need to be removed. In the ROB two additional steps must now be taken. To reset the pc a message must be sent to the Fetch unit. To remove invalid instructions, all the following entries in the ROB must removed. This is done once all of them have completed and returned from other modules (e.g Memory).

4.5 The Register-Renaming AX Model, Mrr_{ax}

4.5.1 Definition

Mrr_{ax} contains five main functional units. Communication between these units is done with queues. The flow of information among these units is illustrated in Figure 4-2. The Funit contains the program counter, instruction memory and Branch Target Buffer (BTB.) Instructions are fetched from the memory and sent to the decode unit. The BTB provides predictions of the next pc. The Decode contains the register file, Re-Order Buffer (ROB) and the reset counter. The ROB does most of the work in the decode unit. It uses register renaming, assigning tags to register and keeping track of updating tags and registers with values. From the Decode, instructions are dispatched for execution to the three smaller units: Exec for ALU operations, Mem for Memory accesses and BP for branch resolution. The results are returned to Decode, where instructions are either committed if correctly speculated, or removed if incorrectly speculated, with appropriate reseting of the instruction fetch. The grammar for Mrr_{ax} is defined in Figure 4-3.

4.5.2 Rules

The rules below are formatted to simplify comprehension. In each old TRS expression the key triggering terms are in bold face. In each new expression the changed terms are in bold face. Operations described in the *where* clauses are abstractions for concepts discussed in the text.

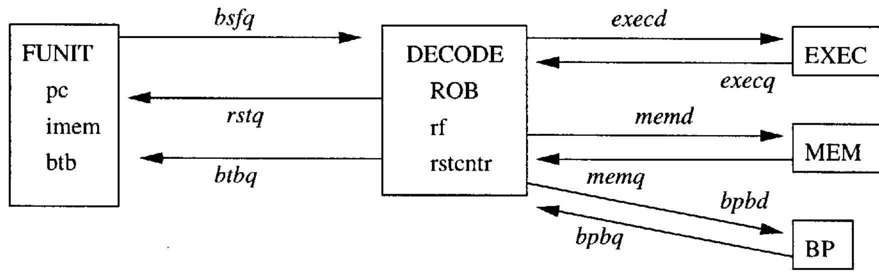


Figure 4-2: **Schematic of the units of Mrr_{ax}.** Information flows through the processor through the queues connecting the units. Speculative instruction fetch and branch prediction occur in Funit. Instruction decode and dispatch, as well as committal and completion occur within the Decode Unit. The Re-Order Buffer (ROB) takes care of register renaming. The three smaller functional units take care of actual execution of the instructions.

Fetch

The Funit optimistically fetches rules, speculating with the predictions of the BTB. When a prediction has been determined to be incorrect, a message arrives from the Decode unit via the reset queue (rstq), causing instruction fetch to continue at the new, correct pc provided in the message. Update messages for the BTB are received from the branch target buffer queue (btbq.) Though we do not discuss them here, there are many different branch prediction schemes possible.

Instruction Fetch

Funit(pc, btb, prog): btbq, ϵ : bsfq

$\Longrightarrow$ Funit(pc', btb, prog): btbq, ϵ : **bsfq;Ib(pc, pc', inst)**

where pc' = btb[pc] *and* inst = prog[pc]

Restart at new PC

Funit(pc, btb, prog): btbq, **(newpc);rstq** : bsfq

$\Longrightarrow$ Funit(newpc, btb, prog): btbq, rstq : **Restart;bsfq**

Update branch prediction

Funit(pc, btb, prog): **(pc', npc, res);btbq** , rstq : bsfq

$\Longrightarrow$ Funit(pc, **btb'**, prog): **btbq**, rstq : bsfq

where btb' = updateBTB(btb, pc', npc, res)

Decode

The Decode unit receives instructions from the Funit through the instruction fetch queue (bsfq.) Instructions are speculatively decoded and enqueued in the Re-Order Buffer (ROB) unless a non-zero reset counter (rstcntr) indicates that the incoming instructions are known to be invalid. Invalid instructions are discarded until a reset acknowledgment is received.

Discard mispredicted fetches

Decode (rf, rob, **cntr**) : **Ib(-,-,-);bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if **cntr $\neq$ 0**

$\Longrightarrow$ Decode (rf, rob, cntr) : **bsfq**, execq, memq, bpq : btbq, rstq , execd, memd, bpd

SYS	=	Sys(<FUNIT: BTBQ, RSTQ: BSFQ>, <DECODE: BSFQ, EXECQ, MEMQ, BPQ: BTBQ, RSTQ, EXECD, MEMD, BPD>, <EXEC: EXECD: EXECQ>, <MEM: MEMD: MEMQ>, <BP: BPD: BPQ>)	*System*
FUNIT	=	Funit(PC, BTB, IMEM)	*Fetch Unit*
DECODE	=	Decode(RF, ROB, RSTCNTR)	*Decode Unit*
EXEC	=	Exec	*Execution Unit*
MEM	=	Mem(V)	*Data Memory Unit*
BP	=	Bp	*Branch Resolution Unit*
ROB	=	List IRB	*Re-Order Buffer*
BTBQ	=	Queue (IA, IA, RES)	*Decode to Funit*
RSTQ	=	Queue (IA)	*Decode to Funit*
BSF	=	Queue IB	*Funit to Decode*
EXECQ	=	Queue IDB	*Exec to Decode*
MEMQ	=	Queue IDB	*Mem to Decode*
BPQ	=	Queue IDB	*BP to Decode*
EXECD	=	Queue IDB	*Decode to Exec*
MEMD	=	Queue IDB	*Decode to Mem*
BPD	=	Queue IDB	*Decode to Bp*
IB	=	Ib (PC, PC, INST) || Restart	*Instruction Buffer*
IDB	=	Itb (TAG, INSTTEMP)	*Instruction Template Buffer*
IBPB	=	Itb (TAG, PC, INSTTEMP)	*Instruction Template Buffer*
ROB	=	List IRB	*Re-Order Buffer*
IRB	=	Itb (TAG, PC, INSTTEMP, STATE)	*Element of ROB*
STATE	=	Wait || Exec || Done || Miss || Kill	*States used in IRB*
RES	=	Correct || Miss	*Result of prediction*
INST	=	r:=loadc(tv) || r:=loadpc || r:=Op(r1, r2) || Jz(rc, ra) || r:=load(ra) || Store(ra, rv)	*Instructions*
INSTTEMP	=	r:=loadc(tv) || r:=loadpc || r:=Op(tv1,tv2) || Jz(tvc,tva,ppc) || r:=load(a) || Store(a,tv) || r:=tv || JzIncorrect(pc) || JzCorrect() || StoreDone || v	*Instructions*

Figure 4-3: **Definition of** Mrr_{ax}

Restart on cue

Decode (rf, rob, **cntr**) : **Restart;bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
⟹ Decode (rf, rob, **cntr - 1**) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd

If the rstcntr is zero, instructions are decoded. There are two cases for enqueueing instructions in the ROB. If the value of an assignment can be determined immediately (e.g. r := Loadc(v) or r := LoadPC()) there is no need for the instruction to be dispatched to an execution unit. The ROB assigns the target register a tag and enqueues that instruction with state Done.

Decode LoadC instructions

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Loadc(v));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
⟹ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
where rob' = enqueue_done(pc, insttemp) *and* insttemp = v

Decode LoadPC instructions

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Loadpc());bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
⟹ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
where rob' = enqueue_done(pc, insttemp) *and* insttemp = pc

The second case is when the instruction requires an execution unit for the result to be determined. In this case, tags are assigned to the target register if necessary, and the instruction is enqueued with state Wait. The register operands of the instruction are looked up in the ROB and the correct tag or value is returned. The enqueued instruction will then wait to be dispatched to the appropriate functional unit.

Decode Op instructions

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Op(r1,r2));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
⟹ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
where rob' = enqueue(pc, insttemp) *and* insttemp = r: = Op(vt1,vt2) *and* vt1 = lookup(r1,rob,rf) *and* vt2 = lookup(r2,rob,rf)

Decode Jz instructions

Decode (rf, rob, **0**) : **Ib(pc, ppc, Jz(rc,ra));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
⟹ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
where rob' = enqueue(pc, insttemp) *and* insttemp = Jz(vtc,vta,ppc) *and* vtc = lookup(rc,rob,rf) *and* vta = lookup(ra,rob,rf)

Decode Load instructions

Decode (rf, rob, **0**) : **Ib(pc, -, r: = Load(ra));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
⟹ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
where rob' = enqueue(pc, insttemp) *and* insttemp = r: = load(vta) *and* vta = lookup(ra,rob,rf)

Decode Store instructions

Decode (rf, rob, **0**) : **Ib(pc, -, Store(ra,rv));bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
⟹ Decode (rf, **rob'**, 0) : **bsfq**, execq, memq, bpq : btbq, rstq, execd, memd, bpd
where rob' = enqueue(pc, insttemp) *and* insttemp = store(vta,vtv) *and* vta = lookup(ra,rob,rf) *and* vtv = lookup(rv,rob,rf)

Dispatch

Instructions with state Wait are dispatched to the appropriate functional units and changed to state Exec only when certain conditions are met. First, all the operands must be values, not tags. Second, if any instructions ahead in the ROB are in state Miss (incorrectly predicted jump) then the instructions are not dispatched because they are sure to be discarded. Finally, memory operations cannot be dispatched until they are at the head of the ROB. This strict memory model insures that memory reads and writes will only occur when the instruction is known to be correctly speculated, and in proper order with all other reads and writes. Instructions to the Execute Unit are dispatched through the execute dispatch queue (execd.) Instructions to the Memory Unit and sent through the memory dispatch queue (memd). Likewise, instructions to the BP are sent through the bp dispatch queue (bpd.)

Dispatch Op instructions

Decode (rf, rob1;**Itb(tag, pc, insttemp, Wait)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = r := Op(v1,v2) ***and*** no_miss(rob1)

⟹ Decode (rf, rob1;**Itb(tag, pc, insttemp, Exec)**;rob2, cntr) : bsfq , execq, memq, bpq : btbq, rstq, **execd;Idb(tag, insttemp)** , memd, bpd

Dispatch Jz instructions

Decode (rf, rob1;**Itb(tag, pc, insttemp, Wait)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = Jz(vc,va,ppc) ***and*** no_miss(rob1)

⟹ Decode (rf, rob1;**Itb(tag, pc, insttemp, Exec)**;rob2, cntr) : bsfq , execq, memq, bpq : btbq, rstq, execd, memd, **bpd;Idb(tag, pc, insttemp)**

Dispatch Load instructions

Decode (rf, **Itb(tag, pc, insttemp, Wait)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = r: = Load(a)

⟹ Decode (rf, **Itb(tag, pc, insttemp, Exec)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, **memd;Idb(tag, insttemp)**, bpd

Dispatch Store instructions

Decode (rf, **Itb(tag, pc, insttemp, Wait)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if insttemp = Store(a,v)

⟹ Decode (rf, **Itb(tag, pc, insttemp, Exec)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, **memd;Idb(tag, insttemp)**, bpd

Complete

Instructions are received from the functional units in three queues. These queues are the execute queue (execq), memory unit queue (memq) and bp queue (bpq.) The results returned in these queues are used to update the ROB. If the instruction returned a value (i.e. was Load or Op) that value is given to the ROB, which updates all occurrences of the destination tag following it in the ROB. These instructions and StoreDone acknowledgments are marked as Done in the ROB.

Complete an Op Instruction

Decode (rf, rob, cntr) : bsfq, **Idb(tag, insttemp);execq**, memq, bpq : btbq, rstq, execd, memd, bpd

⟹ Decode (rf, **rob'**, cntr) : bsfq, **execq**, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = updaterob(rob, tag, insttemp)

Complete an Load/Store Instruction

Decode (rf, rob, cntr) : bsfq, execq, **Idb(tag, insttemp);memq**, bpq : btbq, rstq, execd, memd, bpd

⟹ Decode (rf, **rob'**, cntr) : bsfq, execq, **memq**, bpq : btbq, rstq, execd, memd, bpd

where rob' = updaterob(rob, tag, insttemp)

Results returning from the BP fall into one of three cases. If the jump was correctly predicted, the instruction is marked Done. If the result was incorrectly predicted, the ROB searches for a Missed jump ahead. If there is another misprediction ahead in the ROB, the current misprediction is ignored and marked as Done. If there is not a misprediction, the flow of control must be changed, so a Reset message is sent to the Funit via the rstq, the rstcntr is incremented and the state of the instruction is marked Miss.

Complete a JzCorrect Instruction

Decode (rf, rob, cntr) : bsfq, execq, memq, **Idb(tag, JzCorrect());bpq** : btbq, rstq, execd, memd, bpd

⟹ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, **bpq** : btbq, rst q, execd, memd, bpd

where rob' = updaterob(rob, tag, insttemp)

Complete a JzInCorrect() Instruction

Decode (rf, rob1;**Itb(tag, pc, -, Exec)**;rob2, cntr) : bsfq, execq, memq, **Idb(tag, JzInCorrect(newPC));bpq** : btbq, rstq, execd, memd, bpd

if not no_miss(rob1)

⟹ Decode (rf, rob1;**Itb(tag, pc, insttemp, Done)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

Complete a JzInCorrect() Instruction

Decode (rf, rob1;**Itb(tag, pc, -, Exec)**;rob2, cntr) : bsfq, execq, memq, **Idb(tag, JzInCorrect(newPC));bpq** : btbq, rstq, execd, memd, bpd

if no_miss(rob1)

⟹ Decode (rf, rob1;**Itb(tag, pc, insttemp, Miss)**;rob2, **cntr + 1**) : bsfq, execq, memq, bpq : btbq, **rstq;(newPC)**, execd, memd, bpd

Rewind

When an instruction with state Miss is in the ROB, the instructions following it must be removed. This can only occur if no instructions following it are still being executed. If Exec state instructions were removed, the functional units could return with values and attempt to update the ROB with non-existent tags and values. Despite having to wait for all the rules currently being executed to return, this rule can be assured of firing. This is because no new instructions will be decoded (since the reset counter is non-zero) and no Waiting instructions will be dispatched (because there is an instruction in state Wait ahead.) Therefore, the safe_to_kill operation on the ROB used below will always eventually be true.

Rewind

Decode (rf, rob1;**Itb(tag, pc, insttemp, Miss);rob2**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if no_miss(rob1) *and* safe_to_kill(rob2)

⟹ Decode (rf, rob1;**Itb(tag, pc, insttemp, Done)**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

Commit

Commitment of an instruction only occurs when it reaches the head of the ROB. This is to ensure that an instruction is never committed when it should not be. If an interrupt or misprediction occurs in front of a Done instruction, it should not be committed. All committed instructions are removed from the ROB and the tag is freed. Committed jumps send update information back through the btbq. Committed assignment to a register produces actual writing to the RF.

Commit to register and remove

Decode (rf, **Itb(tag, pc, v, Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

⟹ Decode (**rf'**, **rob'**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rf' = update(rf,r,v) *and* rob = dequeue(rob)

Commit a store completion

Decode (rf, **Itb(tag, pc, Store(-,-), Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

⟹ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

where rob' = dequeue(rob)

Commit a Jz complete, and update btb

Decode (rf, **Itb(tag, pc, JzCorrect(), Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

⟹ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, bpq : **btbq;(pc ,newpc, Correct)**, rstq, execd, memd, bpd

where rob' = dequeue(rob)

Commit a Jz complete, and update btb

Decode (rf, **Itb(tag, pc, JzInCorrect(newpc), Done)**;rob, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

⟹ Decode (rf, **rob'**, cntr) : bsfq, execq, memq, bpq : **btbq;(pc ,newpc, InCorrect)**, rstq, execd, memd, bpd

where rob' = dequeue(rob)

Throughout these previous sections, the state of an ROB entry changes many times. Figure 4-4 shows these transitions.

Interrupt

Mrr_{ax} must handle precise interrupts correctly. Precise interrupts are defined as happening at a specific point - all the instructions before the one that generated the interrupt or exception have fully completed execution, and none of the ones after it have begun. To ensure precise interrupts, all the instructions ahead of the target instruction in the ROB must complete and all those after it must not complete. For synchronous interrupts, control flow is changed to the interrupt handler address. The Rewind rule then cleans up the invalid instructions after the target.

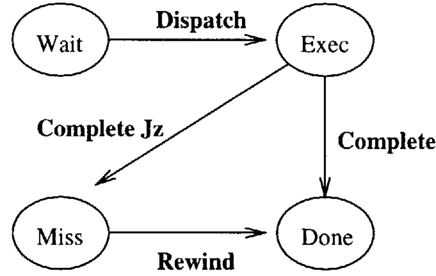


Figure 4-4: **State transitions for IRB entries in ROB** Decode of an instruction with a value as an argument creates an IRB with state Done. Decode of other instructions creates the IRB with the state Wait. Dispatch rules transition to the Exec state. If the IRB is an incorrect jump, the Completion rules transition the state to Miss. Other IRBs move to Done with Completion rules. The Rewind rule moves from Miss to Done. IRB's with the state Done are retired by the Commit rules.

Interrupt Invalid

Decode (rf, rob1;**Itb(tag, pc, insttemp, state)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if synchronous_interrupt(insttemp) *and* not no_miss(rob1)

$\Longrightarrow$ Decode (rf, rob1;**Itb(tag, pc, insttemp, Done)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

Interrupt Valid

Decode (rf, rob1;**Itb(tag, pc, insttemp, state)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd

if synchronous_interrupt(insttemp) *and* no_miss(rob1)

$\Longrightarrow$ Decode (**rf'**, rob1;**Itb(tag, pc, insttemp, Miss)**;rob2, **cntr + 1**) : bsfq, execq, memq, bpq : btbq, **rstq;(interruptHandlerPC)**, execd, memd, bpd

where rf' = update(rf, excep_pointer, pc)

Execute

The three execution units execute specific types of instructions and return results. In an ISA with more types of instructions, this model could be simply expanded to include floating point or other specialized units. In Mrr_{ax}, the execute unit applies the operation to the arguments. The Memory unit reads or writes to memory. The BP resolves branches and determines whether or not the speculation made in the Funit was correct.

Exec

Exec : **Itb(tag, r: = Op(v1,v2));execd** : execq

$\Longrightarrow$ Exec : execd : **execq;Itb(tag, v)**

where v = Op(v1,v2)

Mem Load

Mem mem : **Itb(tag, r: = Load(a));memd** : memq

$\Longrightarrow$ Mem mem : memd : **memq;Itb(tag, v)**

where v = mem[a]

Mem Store

Mem mem : **Itb(tag, Store(a,v));memd** : memq

$\Longrightarrow$ Mem mem : memd : **memq;Itb(tag, StoreDone)**
where mem[a = v]

Bp Taken Correct
Bp : **Itb(tag, -, Jz(0,va,ppc));bpd** : bpq
if va = ppc
$\Longrightarrow$ Bp : bpd : **bpq;Itb(tag, JzCorrect())**

Bp Taken Incorrect
Bp : **Itb(tag, -, Jz(0,va,ppc));bpd** : bpq
if va $\neq$ ppc
$\Longrightarrow$ Bp : bpd : **bpq;Itb(tag, JzInCorrect(va))**

Bp Not Taken Correct
Bp : **Itb(tag, pc, Jz(1,va,ppc));bpd** : bqp
if pc+1 = ppc
$\Longrightarrow$ Bp : bpd : **bpq;Itb(tag, JzCorrect())**

Bp Not Taken Incorrect
Bp : **Itb(tag, Jz(1,va,ppc));bpd** : bpq
if pc+1 $\neq$ ppc
$\Longrightarrow$ Bp : bpd : **bpq;Itb(tag, JzInCorrect(pc+1))**

Kill procedure

The jump incorrect completion rules and interrupt rule can be viewed as special cases of the kill function. This function resets the pc to that of a given instruction, and trashes all instructions after it, without changing the behavior of the processor. For the P_S model, the rule would be as follows:

Kill for P_S
(ia, rf, ROB(t, ia', it);rob2, btb, im)
$\Longrightarrow$ (ia', rf, ϵ, btb, im)

This function can be extended to Mrr_{ax}. Provided there are no misses in front of the target instruction in the ROB, the pc can be reset. The instructions following this kill are cleaned up by the rewind rule.

Kill for Mrr_{ax}
Decode (rf, rob1;**Itb(tag, pc, insttemp, state)**;rob2, cntr) : bsfq, execq, memq, bpq : btbq, rstq, execd, memd, bpd
if kill(insttemp) *and* no_miss(rob1)
$\Longrightarrow$ Decode (**rf**, rob1;**Itb(tag, pc, insttemp, Miss)**;rob2, **cntr + 1**) : bsfq, execq, memq, bpq : btbq, **rstq;(pc)**, execd, memd, bpd

The Jump Incorrect Completion rule uses the kill rule triggered by a JumpIncorrect() received in the bpq and resets the pc to the correctly branch target. The Interrupt rule uses the kill rule triggered by an interrupt and resets the pc to the interrupt handler address.

4.6 Summary

In this final Chapter on modeling techniques we explored a key idea - modularity. Just as in programming, modularity in TRS models provides the benefits of breaking down complexity and allowing independent development of separate but compatible modules. The $Mspec_{ax}$ and model introduced

the btb with transparent modularity. Mrr_{ax}, with its several functional units, represented a truly modular model. Work on subsystems such as memory units can easily be 'plugged in' to this model if they have the correct communication interface.

Chapter 5

Simulation

Simulation of TRS models in software is a key part of the development process. Techniques for producing accurate simulations of both the TRS itself and the anticipated hardware version are presented. Several simulators are discussed and used on a test suite to show how effective simulation can be from a design standpoint.

5.1 Principles of Simulation

There are several different strategies possible for the implementation of a TRS model in software. Each has its benefits and specific uses. Two general strategies, rule-centric and clock-centric, are discussed below. Clock-centric seeks to simulate the execution of a TRS turned into standard hardware, i.e. a instep-execution of a pipeline. Rule-centric seeks to simulate the actual execution behavior of a TRS model, i.e. multiple rules triggering simultaneously and one firing.

5.1.1 Clock-centric Implementations

A clock-centric implementation simulates how a clocked hardware implementation of the TRS would behave. A clock-centric implementation tries to execute as many rules as possible during a single clock cycle, while preserving the semantics of an atomic TRS. This requires analysis of the rules to ensure that the parallel execution of the rules is equivalent to the true serial, atomic execution. James Hoe is using this method for the TRS compiler [6].

To conduct this analysis of a TRS, we first need to make some definitions and functions describing a TRS. Recall that in Section 1.2 it was discussed that a rule has a precondition (defined by the initial state and optional if clause) and rewrites specified by the new state and optional where clause. For analysis purposes, we here define two functions for a rule, π_r and δ_r. π_r is a function of the state of the system and is true when the rule's precondition is satisfied and false otherwise.

$$\pi_r : state \rightarrow \{true, false\}$$

δ_r is a function of the state of the system and produces the new state of the system after the rewrite.

$$\delta_r : state \rightarrow state$$

For illustration, recall Rule 1 of M_{gcd} (the Euclid GCD model in Section 1.2).

Rule 1

Pair(Num(x), Num(y))
if y = 0
$\Longrightarrow$ Done(x)

For this rule, π_r is true only when $y = 0$ and false otherwise. δ_r simply rewrites the pair (x, y) to be x. For M_{gcd}, as any other TRS model, all the rules have π_r and δ_r. There are three important relationships between pairs of rules that are essential for producing a correct clock-centric implementation. These three relationships are independence, being in conflict (and conversely being conflict free) and dominance.

Independence

If two rules are independent, they never fire at the same time. (Note that mutual independence, just as in probability is different than pairwise independence.) Formally, two rules r1 and r2 are independent if for all possible states, the preconditions of r1 and r2 and never both true::

$$\forall s, (\pi_{r1}(s) \wedge \pi_{r2}(s)) = false$$

If two rules are independent, they will never both fire at once, so no problems can possibly arise. Independent rules can be grouped together in code in a case statement. If rules are not independent, problems can arise if they execute simultaneously.

Conflict-free

If rules are not independent, we try to further classify them by testing to see if they are conflict-free. Intuitively, the execution of two rules in parallel needs to be equivalent to either sequential execution. First, one rule can never cancel the execution of the other, or else firing them together could be incorrect. Secondly, the final state must be easily derivable from the individual changes done in parallel. Formally, two rules r1 and r2 are conflict-free if for all states where both rules have

satisfied preconditions: 1) firing of one rule always triggers the other (no 'cancelling') and 2) the final state of the two sequential executions satisfy some least upper bound (are 'easily mergible'):

$$\begin{aligned}\forall s, (\pi_{r1}(s) \wedge \pi_{r2}(s)) &\rightarrow \\ \pi_{r1}(\delta_{r2}(s)) \wedge \pi_{r2}(\delta_{r1}(s)) &= \mathit{true} \\ \mathit{least_upper_bound}(\delta_{r1}(s), \delta_{r2}(s)) &= \delta_{r1}(\delta_{r2}(s)) \wedge \delta_{r2}(\delta_{r1}(s))\end{aligned}$$

Determining if two rules are conflict can be a straightforward process. In practice with processor models, conflict-free rules have the following characteristics. To satisfy the first condition, the rules just need not modify the state element the other's precondition depends on. To satisfy the state upper bound, the rules modify disjoint parts of the state. Since proving two rules conflict free is not necessarily so simple, a way to cope with conflicting rules is needed.

Domination

Dominance applies to rules in conflict. Intuitively, one rule dominating the other means that the affects of one rule are erased by the other. Formally, r1 dominates r2 ($r1 > r2$) if for all states where both rules have satisfied preconditions: 1) r1's predicate is always true after r2's execution and 2) execution of r1 on the initial state is equivalent to executing r2 then r1:

$$\begin{aligned}\forall s, (\pi_{r1}(s) \wedge \pi_{r2}(s)) &\rightarrow \\ \pi_{r1}(\delta_{r2}(s)) &= \mathit{true} \\ \delta_{r1}(s) &= \delta_{r1}(\delta_{r2}(s))\end{aligned}$$

If r1 dominates r2 and both predicates are true, only rule 1 is fired. In practice, dominator occurs when one rule modifies a state element the the other's predicate depends on. Jump instructions resetting the pc dominate over instruction fetch rules. Note that dominance must be dealt with only on a pairwise basis and is not transitive:

$$r1 > r2, r2 > r3 \not\rightarrow r1 > r3$$

If none of the above conditions are met, the rules conflict. In this case, an arbitrary choice on each cycle is made and only one rule executes. This choice can be either random or in a round-robin fashion to avoid starvation. Though the TRSs presented here are written to not have conflicting rules, in the general case conflicting rules are possible (and even probable.)

Implementation Details

Using the three principles above, a parallel execution can be constructed from an atomic, serial TRS. Each pair of rules must be analyzed. Independent pairs can be ignored, since they will never fired in parallel. Though this adds nothing to increase the parallelism of the implementation, by integrating independent rules into one case statement or thread can speed up pre-condition checking. Of non-independent pairs, most rules (in our TRSs, not in general) are conflict-free. Conflict-free pairs can safely be executed in parallel given the easy way to merge the new states. Rules that do conflict fall into two classes: dominating and non-dominating. A pair with domination can still be executed in parallel by cancelling the dominated rule. Nothing within our analysis above can ameliorate non-dominating, conflicting rules. In this case, a non-deterministic choice must be made as to which rule of the pair will fire during that cycle while the other rule is delayed.

Actually writing code to do what is described above is not very easy. Analysis by a human can be done quickly, but automation is much more complex. One approach, developed jointly with James Hoe for use in his TRS compiler (need to cite his thesis here?) defines sets for the ranges (R(f)) and domains (D(f)) of the π and δ functions and uses simple set operations can classify the rules. The domain of π or δ is the set of state elements that affect the result. The range of δ is the state elements it modifies.

IN this analysis, independence is not checked, since independent rules will 'take care of themselves' and not cause problems. Determining if two rules are conflict-free using the following heuristic: if each rule does not modify any state the other's precondition depends on and the rules do not modify the same portions of state:

$$
\begin{aligned}
(D(\pi_{r1}) \cap R(\delta_{r2})) &= \emptyset \\
(D(\pi_{r2}) \cap R(\delta_{r1})) &= \emptyset \\
(R(\delta_{r1}) \cap R(\delta_{r2})) & \quad\ \emptyset
\end{aligned}
$$

Determining if r1 dominates r2 is done with the following heuristic: 1) r2 does not invalidate r1 by modifying any element in r1's domain, 2) r2 modifies only elements of state the r1 does:

$$
\begin{aligned}
R(\delta_{r2})) \cap (D(\pi_{r1}) &= \emptyset \\
R(\delta_{r2})) \subset R(\delta_{r1})) &= \mathit{true}
\end{aligned}
$$

These methods will quickly generate a correct, but not optimal, implementation.

5.1.2 Rule-centric Implementations

In a rule-centric implementation, each rule fires as soon as its precondition becomes true. In order to have all rules checking at once, a multi-threaded implementation is needed. To enforce the atomic model of TRS rewrites, as well as prevent the data races inherent in non-controlled multi threading, it is necessary to use some form of concurrency control on all state elements. In Java this can be done with synchronized methods and `waiting` and `signaling`. This approach, by implementing the true semantics of an atomic TRS, requires no analysis of the rules. This makes a rule-centric model easy to implement. A rule-centric implementation, however, lacks many basic features that a hardware implementation would have, such as a clock. Therefore, while it may serve effectively to simulate a TRS, it provides no information about how a hardware implementation of a TRS would behave.

Many of the same principles discussed in the previous section for use in clock-centric implementations can be used to optimize performance in a rule-centric version. For example, rules that are independent can all be placed in the same thread for execution. This means one thread can check the conditions for firing for all the rules at once and then execute the single one that will fire. The Mrr_{ax} model discussed later used this technique.

5.2 The Simulators for AX TRSs

All simulators except for the Mrr_{ax} simulator, are clock-centric. These first 4 four simulators are used in Section 5.3 for testing.

5.2.1 M_{ax}

All rules in M_{ax} are independent, since each is just a different case for the value if im[pc].

5.2.2 $Mpipe_{ax}$

$Mpipe_{ax}$ involves the first real analysis of the rules. Here the rules in each pipeline stage are independent. This is obvious since at each stage it is a 'dispatch' on the type of the instruction at the head of the queue. Rules between pipe stages are generally conflict free since they involve different resources. Conflicts do arise with Rule 4, the Jz Taken rule. Rule 4 modifies the pc and flushes bsD and bsE, which cause conflict with Rules 1 and 2a-f (fetch and decode). However, Rule 4 dominates these other rules. The table below illustrates the relationships.

	Fetch	Decode	Exec				Mem			WB
Rule	1	2a-f	3	4	5	6	7	8	9	10
1	-	cf	cf	<	cf	cf	cf	cf	cf	cf
2a-f		-	cf	<	cf	cf	cf	cf	cf	cf
3			-	I	I	I	cf	cf	cf	cf
4				-	I	I	cf	cf	cf	cf
5					-	I	cf	cf	cf	cf
6						-	cf	cf	cf	cf
7							-	I	I	cf
8								-	I	cf
9									-	cf

Table 5.1: **Relations between $Mpipe_{ax}$ rules** Independent pairs are marked **I**, conflict-free pairs **cf** and domination by < or >.

5.2.3 $Mbyp_{ax}$

$Mbyp_{ax}$ is fairly similar to $Mpipe_{ax}$. The addition of the bypass rules to the decode stage poses an interesting situation. The dependence of the bypass rules on the presence of instructions in the later queues causes conflicts. Referring back to $Mbyp_{ax}$ as described in Section 3.4, consider the following example: Load(r0, r1) is at the head of bsD, ready to be decoded. At the head of bsW is the instruction r1 = 5. In this case both Rule 2d-2 (Decode Load with bypass) and Rule 10 (Writeback) are triggered. If Rule 10 fires first, it will invalidate Rule 2 by removing the instruction that is the bypass source. Because of this cancelling of Rule 2, domination cannot apply here either?

The solution to this problem involves a closer look at the implementation. Just as all rules in the same pipeline stage are independent, so are all the rules for bypassing a certain instruction type. In $Mbyp_{ax}$, Rules 3, 4, 5 and 6 of the Exec stage are independent, as are Rules 2e-1, -2, -3 and -4 in the Decode stage.

5.2.4 $Mspec_{ax}$

$Mspec_{ax}$ is very similar to $Mpipe_{ax}$, except that only the Exec Jz rules that flush the queues (4b, 5b) dominate over Fetch and Decode. The Exec Jz rules that don't (4a, 5a) disappear, and have no effect on the state.

5.2.5 Mrr_{ax}

The fully featured model is the only rule-centric simulator and is quite a departure from the others. First, it was developed at an early stage before the simulation methodology explained above was formulated. All the intuitive notions in these concepts were used in the development of the simulator, however. This model uses multi-threading to implement concurrency, instead of the single master thread firing rules simultaneously. This multi threaded approach was chosen to effectively model

	Fetch	Decode	Exec						Mem			WB
Rule	1a,b	2a-f	3	4a	4b	5a	5b	6	7	8	9	10
1a,b	-	cf	cf	cf	$<$	cf	$<$	cf	cf	cf	cf	cf
2a-f		-	cf	cf	$<$	cf	$<$	cf	cf	cf	cf	cf
3			-	I	I	I	I	I	cf	cf	cf	cf
4a				-	I	I	I	I	cf	cf	cf	cf
4b					-	I	I	I	cf	cf	cf	cf
5a						-	I	I	cf	cf	cf	cf
5b							-	I	cf	cf	cf	cf
6								-	cf	cf	cf	cf
7									-	I	I	cf
8										-	I	cf
9											-	cf

Table 5.2: **Relations between $Mspec_{ax}$ rules** Independent pairs are marked **I**, conflict-free pairs **cf** and domination by $<$ or $>$.

the modularity of the TRS and to deal with concurrency since the ROB, in particular, is accessed by many different rules simultaneously.

It turns out that this model exhibits true non-determinism. There are rules that conflict and do not dominate. In this case a choice must be made as to which rule to fire.

There are problems with this implementation. Occasionally a simulation will loop forever due to the starvation of certain threads. The finite length queues do not accurately model the infinite length queues for the TRS.

5.3 Test results on simulations

For the testing below I constructed one program designed to be as 'average' as possible. Tables in [4] show that, on average, instructions types break down as follows: Load, 21% – 26%; Store 9% – 12%; Branch, 18% – 24%; ALU/other, 43% – 47%. Simulation of this program on M_{ax} revealed the following instruction frequencies shown in Table 5.3.

The simulators were all instrumented, allowing counts of the number of rule executions and dominance situations. Further instrumentation can easily be added.

Inst type	Number	Percentage
Load	47020	25.0
Store	24680	13.1
Branch	33479	17.8
Other	82708	44.0

Table 5.3: **Instruction Mix for Testing Code** The code was designed to have an instruction mix comparable to the average to provide a more accurate measurement of performance.

Clock time for M_{ax} is $t_{mem} + t_{rf} + t_{alu} + t_{mem} + t_{wb}$. Clock time for $Mpipe_{ax}$ and $Mspec_{ax}$ is

$\text{Max}(t_{mem}, t_{rf}, t_{alu}, t_{mem}, t_{wb})$. [7]

	M_{ax}	$Mpipe_{ax}$	$Mbyp_{ax}$	$Mspec_{ax}$
T_{clock}	4	1	1	1
# Cycles	187887	364452	266067	328876
# Rules Fired	187887	986468	908118	988720
Utilization	-	0.54	0.68	0.60
Normalized Time	1.00	0.48	0.35	0.44

Table 5.4: **Results of simulators for AX models** For each program there are two important criteria: how fast is the model and how efficient? How fast is measured by the number of cycles taken and total time. Efficiency is measured by the utilization of each stage. Since rules correspond to stages (in groups) for a five stage pipeline 5 rules would execute for every cycle. Note that due to starting and finishing up the instruction flow, no program will show 100% utilization on any multistage pipeline.

5.4 Summary

In this chapter two important methods of simulation of TRS models were discussed. Both have been used and can easily and quickly generate correct software simulations of either the pure TRS or an anticipated hardware version. The usefulness of these simulators was demonstrated by using them to run a performance comparison between different implementations of the AX ISA. Formalizations of these simulation principles are the groundwork for automatic generation via a compiler.

Chapter 6

Conclusions

The research presented in this thesis is a key part of the development of a complete system for design, testing and production of hardware using TRSs. The hardware models that were presented exemplify the descriptive power of TRS models and provide a suite of structures and techniques for creating new TRSs of more complex hardware. These techniques include modularity and pipelining. The new structures that were presented in this thesis include queues for communication and interfaces for connecting modules.

Eleven TRS models were developed for two ISAs. They spanned the range in complexity from non-pipelined version to variations on simple pipelines, speculative execution and register renaming.

The discussion in Chapter 5 of rule analysis lays the foundation for systematic hardware synthesis of TRSs. The many simulators demonstrate the testing possible at a software level for hardware designs. The techniques presented provide for either rule- or clock-centric focused simulation and show the ease of instrumentation and comparison of results.

Beyond this work on modeling and simulation is efforts on memory models and cache coherence and hardware synthesis. We hope that the TRS method will improve hardware design just as the use of high-level programming languages and corresponding compilers was a great improvement over writing assembly code.

Appendix A

DLX Models

A.1 The Princeton DLX Model, MP_{dlx}

Just as in the previous case, changing to DLX from AX just adds more rules and no significant complexity. In MP_{dlx} the choice was made to use the second method for dealing with the memory resource conflict, that of always fetching the next instruction during the current execution.

A.1.1 Definition

This Princeton version is analogous to MP_{ax}, with the added consideration of the branch delay slot.

PROC	=	Proc(PC, NEXT, RF, MEM, INST, FLAG)
MEM	=	Array[ADDR] VI
VI	=	VAL || INST
FLAG	=	fetch || execute

Register Instructions

The arithmetic and logical operations simply apply the operator to the two register values (or one value and immediate) and save the result in the register file.

Fetch
Proc(pc, nxt, rf, mem, -, fetch)
$\Longrightarrow$ Proc(pc, nxt, rf, mem, mem[pc], execute)

Reg-Reg Op
Proc(ia, nxt, rf, mem, inst, execute)
if inst == Regregop(rs1, rs2, rd, rrtype)
$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, v], mem, -, fetch)
where v := $\underline{\text{rrtype}}$(rf[rs1], rf[rs2])

Set-Logical
Proc(ia, nxt, rf, mem, inst, execute)
if inst == SetlogOp(rs1, rs2, rd, sltype) *and* $\underline{\text{sltype}}$(rf[rs1], rf[rs2]) == true
$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, 1], mem, -, fetch)

Proc(ia, nxt, rf, im, dm)
if inst == Setlogop(rs1, rs2, rd, sltype) *and* sltype(rf[rs1], rf[rs2]) == false
⟹ Proc(nxt, nxt + 1, rf[rd, 0], mem, -, fetch)

Reg-Imm Rule
Proc(ia, nxt, rf, mem, inst, execute)
if inst == RegimmOp(rs1, imm, rs2, ritype)
⟹ Proc(nxt, nxt + 1, rf[rd, v], mem, -, fetch)
where v := ritype(rf[rs1], imm)

Control Flow Instructions

Due to DLX's branch delay slot, the instruction after a branch or jump is always executed. The following jumps modify the next pc instead of the pc to account for that.

BEQZ
Proc(ia, nxt, rf, mem, inst, execute)
if inst == Beqz(rs1, imm) *and* rf[rs1] == 0
⟹ Proc(nxt, ia + 1 + imm, rf, mem, -, fetch)
Proc(ia, nxt, rf, mem, inst, execute)
if inst == Beqz(rs1, imm) *and* rf[rs1] ≠= 0
⟹ Proc(nxt, nxt + 1, rf, mem, -, fetch)

BNEZ
Proc(ia, nxt, rf, mem, inst, execute)
if im[ia] == Bnez(rs1, imm) *and* rf[rs1] ≠= 0
⟹ Proc(nxt, ia + 1 + imm, rf, im, dm)
Proc(ia, nxt, rf, mem, -, fetch)
if im[ia] == Bnez(rs1, imm) *and* rf[rs1] == 0
⟹ Proc(nxt, nxt + 1, rf, mem, -, fetch)

Jump
Proc(ia, nxt, rf, mem, inst, execute)
if inst == J(imm)
⟹ Proc(nxt, ia + 1 + imm, rf, mem, -, fetch)

Jump and Link
Proc(ia, nxt, rf, mem, inst, execute)
if inst == Jal(imm)
⟹ Proc(nxt, ia + 1 + imm, rf[r31, nxt + 1], mem, -, fetch)

JumpRegister
Proc(ia, nxt, rf, mem, inst, execute)
if im[ia] == Jr(rs1)
⟹ Proc(nxt, rf[rs1], rf, mem, -, fetch)

JumpRegister and Link
Proc(ia, nxt, rf, mem, inst, execute)
if im[ia] == Jalr(rs1)
⟹ Proc(nxt, rf[rs1], rf[r31, nxt + 1], mem, -, fetch)

Memory Instructions

Memory operations are straightforward. How do I deal with signed and unsigned?

Load Word

Proc(ia, nxt, rf, mem, inst, execute)

if inst == Lw(rs1, imm, rd)

$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, mem[addr]], mem, -, fetch)

where addr := rf[rs1] + imm

Load Half-Word

Proc(ia, nxt, rf, mem, inst, execute)

if inst == Lh(rs1, imm, rd)

$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, v], mem, -, fetch)

where v := LogicalAnd(0x0011, mem[addr]) *and* addr := rf[rs1] + imm

Load Byte

Proc(ia, nxt, rf, mem, inst, execute)

if inst == Lb(rs1, imm, rd)

$\Longrightarrow$ Proc(nxt, nxt + 1, rf[rd, v], mem, -, fetch)

where v := LogicalAnd(0x0001, mem[addr]) *and* addr := rf[rs1] + imm

Store Word

Proc(ia, nxt, rf, mem, inst, execute)

if inst == Sw(rs1, imm, rd)

$\Longrightarrow$ Proc(nxt, nxt + 1, rf, mem[addr, rf[rd]], -, fetch)

where addr := rf[rs1] + imm

Store Half-Word

Proc(ia, nxt, rf, mem, inst, execute)

if inst == Sh(rs1, imm, rd)

$\Longrightarrow$ Proc(nxt, nxt + 1, rf, mem[addr, rf[rd]], -, fetch)

where addr := rf[rs1] + imm *and* v := xor(LogicalAnd(0x1100, dm[addr]), LogicalAnd(0x0011, rf[rd]))

Store Byte

Proc(ia, nxt, rf, mem, inst, execute)

if inst == Sb(rs1, imm, rd)

$\Longrightarrow$ Proc(nxt, nxt + 1, rf, mem[addr, rf[rd]], -, fetch)

where addr := rf[rs1] + imm *and* v := xor(LogicalAnd(0x1110, dm[addr]), LogicalAnd(0x0001, rf[rd]))

Currently missing rules for LHU, LBU and the floating operations.

A.2 The Pipelined DLX Model, $Mpipe_{dlx}$

$Mpipe_{dlx}$ has the standard division of the circuit into five stages - Instruction memory, register read, ALU operation, data memory access and write back to the register file.

A.2.1 Definition

For this model we add in queues to connect the pipelines stages (see Section 3.2). We add the instruction buffer to hold the current pc, next pc and instruction through the pipeline. The instructions need to be modified so that they can hold either register names or values, depending on pipeline stage.

PROC	=	Proc(PC, NEXT, IM, DM, RF, BSD, BSE, BSM, BSW)
BSD, BSE, BSM, BSW	=	Queue(IB)
ITB	=	Itb(ADDR, ADDR, INST)
INST	=	REGREGOP || SETLOGOP || REGIMMOP
		JUMPOP || MEMOP || Reqv(RNAME, VAL)
REGREGOP	=	Regregop(RV, RV, RNAME, RRTYPE)
SETLOGOP	=	Setlogop(RV, RV, RNAME, SLTYPE)
REGIMMOP	=	Regimmop(RV, VAL, RNAME, RITYPE)
JUMPOP	=	Beqz(RV, VAL) || Bnez(RV, VAL)
		J(VAL) || Jal(VAL) || Jr(RV) || Jalr(RV)
MEMOP	=	LOADOP || LOAD2OP || STOREOP|| STORE2OP
LOADOP	=	Loadop(RV, VAL, RNAME, LTYPE)
LOAD2OP	=	Load2op(VAL, RNAME, LTYPE)
LTYPE	=	Lw || Lh || Lb
STOREOP	=	Storeop(RV, VAL, RNAME, STYPE)
STORE2OP	=	Store2op(VAL, RNAME, STYPE)
STYPE	=	Sw || Sh || Sb
RV	=	RNAME || VAL

A.2.2 Rules

Instruction Fetch

When rules are fetched, they are simply passed on to the decode stage.

Fetch

Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, wbQ)
$\Longrightarrow$ Proc(next, next + 1, im, dm, rf, decQ;Itb(pc, next, inst), exeQ, memQ, wbQ)
where inst := im[pc]

Instruction Decode

Since pipelines have hazards, the operands of the inst are read only if they are not being written by some instruction later on in the pipeline. So if any of the three following queues have a register being written, then the pipeline stalls. Note here that this first level model includes only reading the register file in this stage, not do any calculations. Therefore conditional branches are not determined until the execute stage, which is rather wasteful.

Decode R-type

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)
if inst == Regregop(rs1, rs2, rd, rrtype) *and* rs1, rs2 $\notin$ Dest(exeQ, memQ, wbQ)
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)
where ninst := Regregop(v1, v2, rd, rrtype) *and* v1 := rf[rs1]; v2 := rf[rs2]

Decode R-type

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)
if inst == Setlogop(rs1, rs2, rd, sltype) *and* rs1, rs2 $\notin$ Dest(exeQ, memQ, wbQ)
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)
where ninst := Setlogop(v1, v2, rd, sltype) *and* v1 := rf[rs1]; v2 := rf[rs2]

Decode I-type, Reg-Immed Arith

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)
if inst == Regimmop(rs1, immed, rd, ritype) *and* rs1 $\notin$ Dest(exeQ, memQ, wbQ)
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := Regimmop(v1, immed, rd, ritype) *and* v1 := rf[rs1]

Decode I-type, Loads

Proc(pc, next, im, dm, rf, (ia, inst);decQ, exeQ, memQ, wbQ)

if inst == Loadop(rs1, immed, rd, ltype) *and* rs1 $\notin$ Dest(exeQ, memQ, wbQ)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, ninst), memQ, wbQ)

where ninst := Loadop(v1, immed, rd, ltype) *and* v1 := rf[rs1]

Decode I-type, Stores

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)

if inst == Storeop(rs1, immed, rd, stype) *and* rs1, rd $\notin$ Dest(exeQ, memQ, wbQ)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := Storeop(v1, immed, vd, stype) *and* v1 := rf[rs1], vd := rf[rd]

Decode I-type, Branches

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)

if inst == Beqz(rs1, immed) *and* rs1 $\notin$ Dest(exeQ, memQ, wbQ)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := Beqz(v1, immed) *and* v1 := rf[rs1]

Decode I-type, Branches

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)

if inst == Bnez(rs1, immed) *and* rs1 $\notin$ Dest(exeQ, memQ, wbQ)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := Bnez(v1, immed) *and* v1 := rf[rs1]

Decode I-type, Jump Registers

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)

if inst == Jr(rs1) *and* rs1 $\notin$ Dest(exeQ, memQ, wbQ)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := Jr(v1) *and* v1 := rf[rs1]

Decode I-type, Jump Registers

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)

if inst == Jalr(rs1) *and* rs1 $\notin$ Dest(exeQ, memQ, wbQ)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := Jalr(v1) *and* v1 := rf[rs1]

Decode J-type, Jumps

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)

if inst == J(immed)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := J(immed)

Decode J-type, Jumps

Proc(pc, next, im, dm, rf, (ia, nia, inst);decQ, exeQ, memQ, wbQ)

if inst == Jal(immed)

$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ;(ia, nia, ninst), memQ, wbQ)

where ninst := Jal(immed)

ALU Execution

Here in the execute stage, the vales read in decode are used. For arithmetic operations, the values are calculated. Jumps and branches are resolved here. Branches have the condition tested, and register jumps have the target resolved. When a change of control flow occurs, the decQ and exeQ

are emptied. When we trash the instructions following us here on a change of control, we have to carefully find the instruction occupying the branch delay slot and **not** trash it.

Execute ArithOp

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
  if inst == Regregop(v1, v2, r, rrtype)
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ;(ia, nia, ninst), wbQ)
  where ninst := Reqv(r, v) *and* v := rrtype(v1, v2)

Execute ArithOp

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
  if inst == Regimmop(v1, v2, r, ritype)
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ;(ia, nia, ninst), wbQ)
  where ninst := Reqv(r, v) *and* v := ritype(v1, v2)

Execute SetLogicalOp

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
  if inst == Setlogop(v1, v2, r, sltype) *and* sltype(v1, v2) == true
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ;(ia, nia, ninst), wbQ)
  where ninst := Reqv(r,1)

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
  if inst == Setlogop(v1, v2, r, sltype) *and* sltype(v1, v2) == false
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ;(ia, nia, ninst), wbQ)
  where ninst := Reqv(r, 0)

Execute BEQZ taken (delay slot in exeQ)

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);itb;exeQ, memQ, wbQ)
  if inst == Beqz(v, immed) *and* v == 0
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, ϵ, itb, memQ, wbQ)
  where addr := ia + 1 + immed

Execute BEQZ taken (delay slot in decQ)

Proc(pc, next, im, dm, rf, itb;decQ, (ia, nia, inst), memQ, wbQ)
  if inst == Beqz(v, immed) *and* v == 0
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, itb, ϵ, memQ, wbQ)
  where addr := ia + 1 + immed

Execute BEQZ not taken

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
  if inst == Beqz(v, immed) *and* v $\neq$= 0
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, wbQ)

Execute BNEZ taken (delay slot in exeQ)

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);itb;exeQ, memQ, wbQ)
  if inst == Bnez(v, immed) *and* v $\neq$= 0
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, ϵ, itb, memQ, wbQ)
  where addr := ia + 1 + immed

Execute BNEZ taken (delay slot in decQ)

Proc(pc, next, im, dm, rf, itb;decQ, (ia, nia, inst), memQ, wbQ)
  if inst == Bnez(v, immed) *and* v $\neq$= 0
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, itb, ϵ, memQ, wbQ)
  where addr := ia + 1 + immed

Execute BNEZ not taken

Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
  if inst == Bnez(v, immed) *and* v == 0
$\Longrightarrow$ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, wbQ)

Execute JumpRegister (delay slot in exeQ)
Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);itb;exeQ, memQ, wbQ)
if inst == Jr(v)
⟹ Proc(v, v + 1, im, dm, rf, ϵ, itb, memQ, wbQ)

Execute JumpRegister (delay slot in decQ)
Proc(pc, next, im, dm, rf, itb;decQ, (ia, nia, inst), memQ, wbQ)
if inst == Jr(v)
⟹ Proc(v, v + 1, im, dm, rf, itb, ϵ, memQ, wbQ)

Execute JumpRegister and Link (delay slot in exeQ)
Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);itb;exeQ, memQ, wbQ)
if inst == Jalr(v)
⟹ Proc(v, v + 1, im, dm, rf, ϵ, itb, memQ;(ia, ninst), wbQ)
where ninst := Reqv(r31, ia + 2)

Execute JumpRegister and Link (delay slot in DecQ)
Proc(pc, next, im, dm, rf, itb;decQ, (ia, nia, inst), memQ, wbQ)
if inst == Jalr(v)
⟹ Proc(v, v + 1, im, dm, rf, itb, ϵ, memQ;(ia, ninst), wbQ)
where ninst := Reqv(r31, ia + 2)

Execute Jump (delay slot in exeQ)
Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);itb;exeQ, memQ, wbQ)
if inst == J(v)
⟹ Proc(addr, addr + 1, im, dm, rf, ϵ, itb, memQ, wbQ)
where addr := ia + 1 + v

Execute Jump (delay slot in DecQ)
Proc(pc, next, im, dm, rf, itb;decQ, (ia, nia, inst), memQ, wbQ)
if inst == J(V)
⟹ Proc(addr, addr + 1, im, dm, rf, itb, ϵ, memQ, wbQ)
where addr := ia + 1 + v

Execute Jump and Link (delay slot in ExeQ)
Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);itb;exeQ, memQ, wbQ)
if inst == Jal(v)
⟹ Proc(addr, addr + 1, im, dm, rf, ϵ, itb, memQ;(ia, ninst), wbQ)
where addr := ia + 1 + v *and* ninst := Reqv(r31, ia + 2)

Execute Jump and Link (delay slot in DecQ)
Proc(pc, next, im, dm, rf, itb;decQ, (ia, nia, inst);exeQ, memQ, wbQ)
if inst == Jal(v)
⟹ Proc(addr, addr + 1, im, dm, rf, itb, ϵ, memQ;(ia, ninst), wbQ)
where addr := ia + 1 + v *and* ninst := Reqv(r31, ia + 2)

Execute Loads
Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
if inst == Loadop(v, immed, r, ltype)
⟹ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ;(ia, ninst), wbQ)
where ninst := Load2op(addr, r, ltype) *and* addr := v + immed

Execute Stores
Proc(pc, next, im, dm, rf, decQ, (ia, nia, inst);exeQ, memQ, wbQ)
if inst == Storeop(v1, immed, vd, stype)
⟹ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ;(ia, ninst), wbQ)
where ninst := Store2op(addr, vd, stype) *and* addr := v1 + immed

Memory

Loads and stores to memory are done here. Everything else is passed on. Note - if we add the value read plus the offset in the decode stage, or combine with the below rules, we can combine the memory and execute stages.

Memory Load Word

Proc(pc, next, im, dm, rf, decQ, exeQ, (ia, nia, inst);memQ, wbQ)
if int == Load2op(addr, r, Lw)
⟹ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, wbQ;(ia, nia, inst))
where ninst := Reqv(r, dm[addr])

Memory Load Half-word

Proc(pc, next, im, dm, rf, decQ, exeQ, (ia, nia, inst);memQ, wbQ)
if int == Load2op(addr, r, Lh)
⟹ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, wbQ;(ia, nia, inst))
where ninst := Reqv(r, v) *and* v == LogicalAnd(0x0011, dm[addr])

Memory Load Byte Memory

Proc(pc, next, im, dm, rf, decQ, exeQ, (ia, nia, inst);memQ, wbQ)
if int == Load2op(addr, r, Lb)
⟹ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, wbQ;(ia, nia, inst))
where ninst := Reqv(r, v) *and* v == LogicalAnd(0x0001, dm[addr])

Memory Store Word

Proc(pc, next, im, dm, rf, decQ, exeQ, (ia, nia, inst);memQ, wbQ)
if inst == Store2op(addr, vd, Sw)
⟹ Proc(pc, next, im, dm[addr := vd], rf, decQ, exeQ, memQ, wbQ)

Memory Store Half-Word

Proc(pc, next, im, dm, rf, decQ, exeQ, (ia, nia, inst);memQ, wbQ)
if inst == Store2op(addr, vd, Sh)
⟹ Proc(pc, next, im, dm[addr := v], rf, decQ, exeQ, memQ, wbQ)
where v := xor(LogicalAnd(0x1100, dm[addr]), LogicalAnd(0x0011, vd))

Memory Store Byte

Proc(pc, next, im, dm, rf, decQ, exeQ, (ia, nia, inst);memQ, wbQ)
if inst == Store2op(addr, vd, Sb)
⟹ Proc(pc, next, im, dm[addr := v], rf, decQ, exeQ, memQ, wbQ)
where v := xor(LogicalAnd(0x1110, dm[addr]), LogicalAnd(0x0001, vd))

Memory Other

Proc(pc, next, im, dm, rf, decQ, exeQ, (ia, nia, inst);memQ, wbQ)
if inst == Reqv(r, v)
⟹ Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, wbQ;(ia, inst))

Register Writeback

Values are written back to the register file here.

Writeback

Proc(pc, next, im, dm, rf, decQ, exeQ, memQ, (ia, nia, inst);wbQ)
if inst == Reqv(r, v)
⟹ Proc(pc, next, im, dm, rf[r := v], decQ, exeQ, memQ, wbQ)

A.3 The Speculative DLX Model, $Mspec_{dlx}$

A.3.1 Definition

The definition now has a branch target buffer.

PROC = Proc(PC, NEXT, IM, DM, RF, BTB, BSD, BSE, BSM, BSW)
ITB = Itb(ADDR, ADDR, ADDR, INST)

A.3.2 Rules

There are only two changes here from the pipelined model. First, speculation happens on the instruction fetch. Second, the rules resolving branches and jumps need to correct any mis-speculation and deal with the branch delay slot correctly. Only these modified rules are presented. The new buffer to hold instructions through the pipeline has a fourth element, the predicted pc.

Instruction Fetch

When rules are fetched, they are simply passed on to the decode stage.

Fetch
Proc(pc, next, im, dm, rf, btb, decQ, exeQ, memQ, wbQ)
if im[pc] ≠ Beqz(-,-) or Bnez(-,-) or J(-) or Jal(-) or Jr(-) or Jalr(-)
⟹ Proc(next, next + 1, im, dm, rf, btb, decQ;ITB(pc, next, next + 1, inst), exeQ, memQ, wbQ)
where inst := im[pc]

Fetch Control change
Proc(pc, next, im, dm, rf, btb, decQ, exeQ, memQ, wbQ)
if im[pc] == Beqz(-,-) or Bnez(-,-) or J(-) or Jal(-) or Jr(-) or Jalr(-)
⟹ Proc(next, target, im, dm, rf, btb, decQ;ITB(pc, next, target, inst), exeQ, memQ, wbQ)
where inst := im[pc] *and* target := lookup(btb, pc)

Execute BEQZ taken Correct (delay slot in exeQ)
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);exeQ, memQ, wbQ)
if inst = Beqz(v, immed) *and* v = 0 *and* pred = ia + 1 + immed
⟹ Proc(pc, next, im, dm, rf, btb', decQ, execQ, memQ, wbQ)
where btb' = update(btb)

Execute BEQZ not taken Correct
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);exeQ, memQ, wbQ)
if inst = Beqz(v, immed) *and* v ≠ 0 *and* pred = ia + 2
⟹ Proc(pc, next, im, dm, rf, btb', decQ, exeQ, memQ, wbQ)
where btb' = update(btb)

Execute BEQZ taken InCorrect (delay slot in exeQ)
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);itb;exeQ, memQ, wbQ)
if inst = Beqz(v, immed) *and* v = 0 *and* pred ≠ ia + 1 + immed
⟹ Proc(addr, addr + 1, im, dm, rf, btb', ϵ, itb, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute BEQZ taken InCorrect (delay slot in decQ)
Proc(pc, next, im, dm, rf, btb, itb;decQ, (ia, nia, pred, inst), memQ, wbQ)

if inst = Beqz(v, immed) *and* v = 0 *and* pred $\neq$ ia + 1 + immed
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, btb', itb, ϵ, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute BEQZ not taken InCorrect (delay slot in exeQ)
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);itb;exeQ, memQ, wbQ)
if inst = Beqz(v, immed) *and* v $\neq$ 0 *and* pred $\neq$ ia + 1 + immed
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, btb', ϵ, itb, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute BEQZ not taken InCorrect (delay slot in decQ)
Proc(pc, next, im, dm, rf, btb, itb;decQ, (ia, nia, pred, inst), memQ, wbQ)
if inst = Beqz(v, immed) *and* v $\neq$ 0 *and* pred $\neq$ ia + 1 + immed
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, btb', itb, ϵ, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute BNEZ taken Correct (delay slot in exeQ)
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);exeQ, memQ, wbQ)
if inst = Bnez(v, immed) *and* v $\neq$ 0 *and* pred = ia + 1 + immed
$\Longrightarrow$ Proc(pc, next, im, dm, rf, btb', decQ, execQ, memQ, wbQ)
where btb' = update(btb)

Execute BNEZ not taken Correct
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);exeQ, memQ, wbQ)
if inst = Bnez(v, immed) *and* v = 0 *and* pred = ia + 2
$\Longrightarrow$ Proc(pc, next, im, dm, rf, btb', decQ, exeQ, memQ, wbQ)
where btb' = update(btb)

Execute BNEZ taken InCorrect (delay slot in exeQ)
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);itb;exeQ, memQ, wbQ)
if inst = Bnez(v, immed) *and* v $\neq$ 0 *and* pred $\neq$ ia + 1 + immed
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, btb', ϵ, itb, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute BNEZ taken InCorrect (delay slot in decQ)
Proc(pc, next, im, dm, rf, btb, itb;decQ, (ia, nia, pred, inst), memQ, wbQ)
if inst = Bnez(v, immed) *and* v $\neq$ 0 *and* pred $\neq$ ia + 1 + immed
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, btb', itb, ϵ, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute BNEZ not taken InCorrect (delay slot in exeQ)
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);itb;exeQ, memQ, wbQ)
if inst = Bnez(v, immed) *and* v = 0 *and* pred $\neq$ ia + 1 + immed
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, btb', ϵ, itb, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute BNEZ not taken InCorrect (delay slot in decQ)
Proc(pc, next, im, dm, rf, btb, itb;decQ, (ia, nia, pred, inst), memQ, wbQ)
if inst = Bnez(v, immed) *and* v = 0 *and* pred $\neq$ ia + 1 + immed
$\Longrightarrow$ Proc(addr, addr + 1, im, dm, rf, btb', itb, ϵ, memQ, wbQ)
where addr = ia + 1 + immed *and* btb' = update(btb)

Execute Jump Correct (delay slot in exeQ)
Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);exeQ, memQ, wbQ)
if inst = Jr(v) or J(v) *and* pred = v
$\Longrightarrow$ Proc(pc, next, im, dm, rf, btb', decQ, exeQ, memQ, wbQ)
where btb' = update(btb)

Execute Jump InCorrect (delay slot in exeQ)

Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);itb;exeQ, memQ, wbQ)

if inst = Jr(v) or J(v) *and* pred $\neq$ v

$\Longrightarrow$ Proc(v, v + 1, im, dm, rf, btb', ϵ, itb, memQ, wbQ)

where btb' = update(btb)

Execute Jump InCorrect (delay slot in decQ)

Proc(pc, next, im, dm, rf, btb, itb;decQ, (ia, nia, pred, inst), memQ, wbQ)

if inst = Jr(v) or J(v) *and* pred $\neq$ v

$\Longrightarrow$ Proc(v, v + 1, im, dm, rf, btb', itb, ϵ, memQ, wbQ)

where btb' = update(btb)

Execute Jump and Link Correct (delay slot in exeQ)

Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);exeQ, memQ, wbQ)

if inst = Jalr(v) or Jal(v) *and* pred = v

$\Longrightarrow$ Proc(pc, next, im, dm, rf, btb', decQ, execQ, memQ;(ia, ninst), wbQ)

where ninst = Reqv(r31, ia + 2) *and* btb' = update(btb)

Execute Jump and Link InCorrect (delay slot in exeQ)

Proc(pc, next, im, dm, rf, btb, decQ, (ia, nia, pred, inst);itb;exeQ, memQ, wbQ)

if inst = Jalr(v) or Jal(v) *and* pred $\neq$ v

$\Longrightarrow$ Proc(v, v + 1, im, dm, rf, btb', ϵ, itb, memQ;(ia, ninst), wbQ)

where ninst = Reqv(r31, ia + 2) *and* btb' = update(btb)

Execute Jump and Link InCorrect (delay slot in DecQ)

Proc(pc, next, im, dm, rf, btb, itb;decQ, (ia, nia, pred, inst), memQ, wbQ)

if inst = Jalr(v) or Jal(v) *and* pred $\neq$ v

$\Longrightarrow$ Proc(v, v + 1, im, dm, rf, btb', itb, ϵ, memQ;(ia, ninst), wbQ)

where ninst = Reqv(r31, ia + 2) *and* btb' = update(btb)

Appendix B

Hardware Generation via Compilation

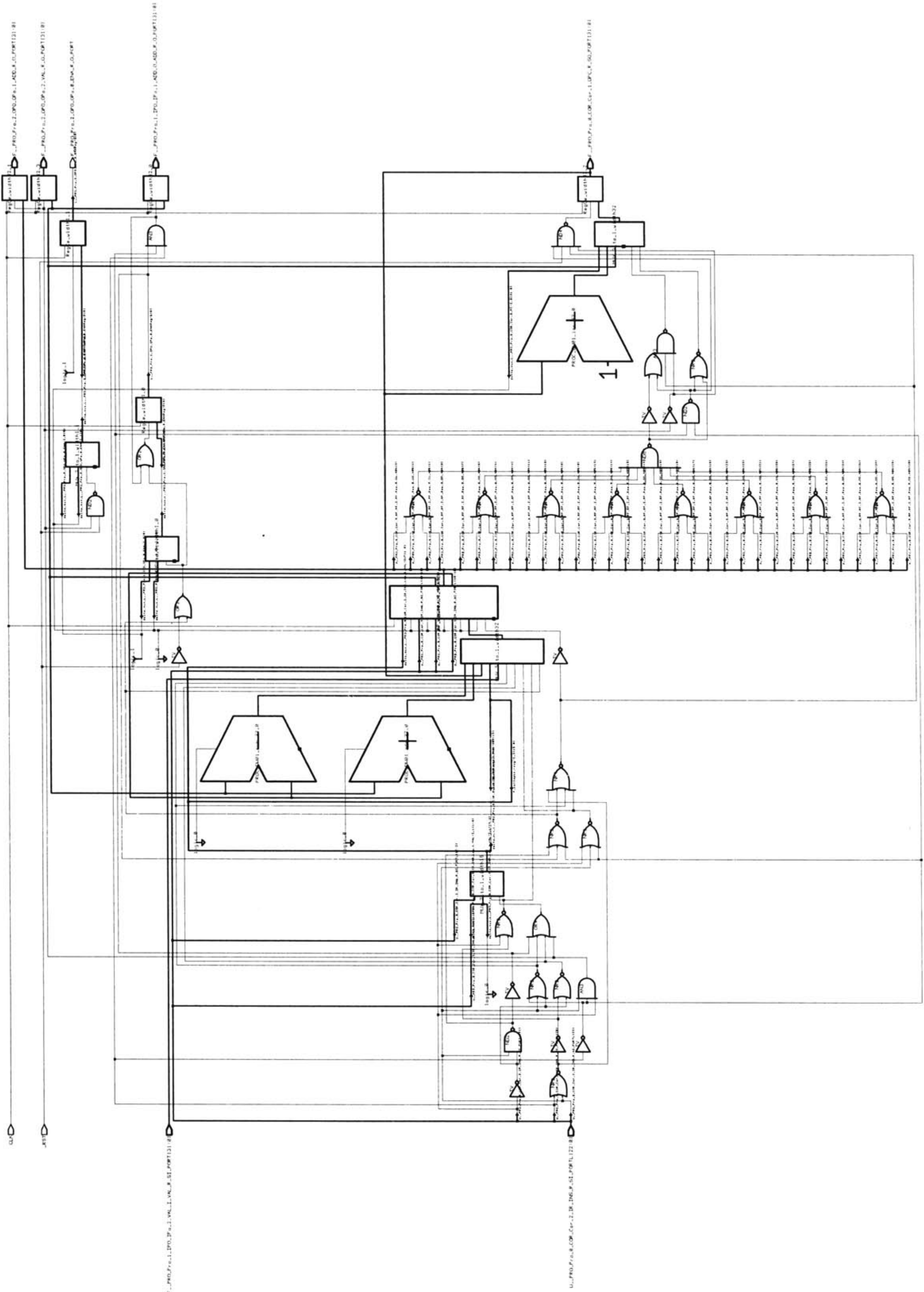
After some slight formatting changes and scaling down of some elements, M_{ax} was compiled into Verilog. James Hoe is working on this compiler which allows a hardware architect to rapidly create simulatable and synthesizable prototype designs directly from their high-level specifications. Furthermore, the combination of high-level synthesis with reconfigurable technology found in the compiler creates a new engineering trade-off point where an application developer could benefit from a hardware implementation for the same amount of time and effort as software development.

Following the TRS below is the schematic of the hardware.

```
PROC   =   Proc(PC, RF, IM, DM)
PC     =   ADDR
RF     =   Array [RNAME] VAL
IM     =   Array [ADDR] INST
DM     =   Array [ADDR] VAL
ADDR   =   Bit[32]
INST   =   Loadc(RNAME, VAL) || Loadpc(RNAME) ||
           Op(RNAME, RNAME, RNAME) || Load(RNAME, RNAME)||
           Store(RNAME, RNAME) || Jz(RNAME, RNAME)
RNAME  =   Reg0 || Reg1 || Reg2 || Reg3                       10
VAL    =   Bit[32]


"Rule 1 - Loadc"
Proc(pc, rf, im, dm)
        where   Loadc(r, v) := im[pc]
==>     Proc(pc', rf[r := v], im, dm)
        where   pc' := pc + 1
```

```
                                                              20
"Rule 2 - Loadpc"
Proc(pc, rf, im, dm)
        where   Loadpc(r) := im[pc]
==>     Proc(pc', rf[r := pc], im, dm)
        where   pc' := pc + 1


"Rule 3 - Op"
Proc(pc, rf, im, dm)
        where   Op(r, r1, r2) := im[pc]                       30
==>     Proc(pc', rf[r := v], im, dm)
        where   v := rf[r1] + rf[r2]
                pc' := pc + 1


"Rule 4 - Load"
Proc(pc, rf, im, dm)
        where   Load(r, r1) := im[pc]
==>     Proc(pc', rf[r := v], im, dm)
        where   v := dm[rf[r1]]                               40
                pc' := pc + 1


"Rule 5 - Store"
Proc(pc, rf, im, dm)
        where   Store(ra, r1) := im[pc]
==>     Proc(pc', rf, im, dm[ad := v])
        where   ad := rf[ra]
                v := rf[r1]                                   50
                pc' := pc + 1

"Rule 6 - Jz taken"
Proc(pc, rf, im, dm)
        if      rf[rc] == 0
        where   Jz(rc, ra) := im[pc]
==>     Proc(pc', rf, im, dm)
        where   pc' := rf[ra]

"Rule 7 - Jz not taken"                                       60
Proc(pc, rf, im, dm)
        if      rf[rc] != 0
        where   Jz(rc, ra) := im[pc]
```

```
==>     Proc(pc', rf, im, dm)
        where   pc' := pc + 1
```



Bibliography

[1] Xiaowei Shen and Arvind. Modeling and verification of ISA implementations. In *Proceedings of the Australasian Computer Architecture Conference*, Perth, Australia, February 1998.

[2] Xiaowei Shen and Arvind. Design and verification of speculative processors. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems*, Marstrand, Sweden, June 1998.

[3] Lisa A. Poyneer, James C. Hoe, and Arvind. A TRS model for a modern microprocessor. Computation Structures Group Memo 408, June 1998.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, San Francisco, California, second edition, 1995.

[5] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile and fences (crf): A new memory model for architects and compiler writers. In *Proceedings of the 26th ISCA*, Atlanta, Georgia, May 1999.

[6] James C. Hoe and Arvind. Micro-architecture exploration and synthesis via trs's. Computation Structures Group Memo 408, April 1999.

[7] David A. Patterson and John L. Hennessy. *Computer Organization and Design The Hardware/Software Interface.* Morgan Kaufmann Publishers, San Francisco, California, 1994.