LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Modeling and Verification of ISA Implementations

Computation Structures Group Memo 400 (A)

Xiaowei Shen and Arvind
xwshen, arvind@lcs.mit.edu

To appear in Proceedings of the Australasian Computer Architecture Conference, Perth, Australia, Feburary 1998.

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Modeling and Verification of ISA Implementations

Xiaowei Shen and Arvind
xwshen, arvind@lcs.mit.edu

Abstract

We propose a method to precisely model implementations of Instruction Set Architectures (ISA) using term rewriting systems (TRS). Our method facilitates understanding of important micro-architectural differences without delving into low-level implementation details. More importantly, the use of TRS allows us to prove rigorously the equivalence of different implementations.

We first define $\mathcal{AX}$, a simple RISC ISA, by specifying its operational semantics using a simple in-order execution model. We then give an $\mathcal{AX}$ implementation which uses register renaming and permits out-of-order instruction execution. The equivalence of the two models is proved by showing that the two TRS's can simulate each other.

1 Introduction

Modern microprocessors embody increasingly complex micro-architectures to achieve high performance. Optimization techniques such as out-of-order and speculative execution, write buffers and split-phase bus transactions, can make the semantics of certain instructions difficult to understand. For example, from the PowerPC manual [7] it is not easy to pin down the precise semantics of memory access and memory barrier instructions given the weakly consistent storage model for shared-memory multiprocessor systems. Often the only precise description of an ISA implementation is the program (in some hardware description language) from which the actual logic gates are generated. Of course each implementation has its own hardware description program which plays an indispensable role in both the behavioral verification of the design and the debugging of the actual microprocessor chip. However, such a program contains too much implementation detail, and is not very amenable to verification as an ISA specification.

This paper takes a novel approach to descriptions of ISA implementations. We describe a computer system and its components as terms generated by a context free grammar. The operational behavior is specified as a set of rules for rewriting the terms that represent the system or its components. Term rewriting system [4] is convenient for describing parallel systems, and can be used to prove the correctness of an implementation with respect to a specification.

We give a brief introduction to TRS in Section 2. In Sections 3 and 4 we present the $\mathcal{AX}$ instruction set and define its operational semantics using a simple in-order execution processor ($\mathcal{P}_{\mathcal{B}}$). In Section 5 we give an implementation of $\mathcal{AX}$ that allows out-of-order execution by employing register renaming ($\mathcal{P}_{\mathcal{R}}$). Then in Section 6, we formally prove that $\mathcal{P}_{\mathcal{R}}$ is a correct implementation of $\mathcal{AX}$ by showing that $\mathcal{P}_{\mathcal{B}}$ and $\mathcal{P}_{\mathcal{R}}$ can simulate each other. Finally we discuss other related work and research in progress.

2 Term Rewriting Systems

A term rewriting system is defined as a tuple (S, R, S_0), where S is a set of terms, R is a set of rewriting rules, and S_0 is the set of initial terms ($\mathrm{S}_0 \subseteq \mathrm{S}$). In the architectural context, the terms and rules of a TRS represent states and state transitions, respectively. The general structure of rewriting rules is as follows:

$$\begin{array}{ll} & s_1 \quad \textit{if} \;\; p(s_1) \\ \longrightarrow & s_2 \end{array}$$

where s_1 and s_2 are terms, and p is a predicate.

A rule can be used to rewrite a term if its left-hand-side pattern matches the term or one of its subterms, and the corresponding predicate is true. If several rules are applicable, then any one of them can be applied. If no rule is applicable, then the term cannot be rewritten any further and is said to be in *normal form*.

We use $\mathrm{C}[\![\;]\!]$ to represent a *context*, which is a term with a "hole" that can be filled by a term. $\mathrm{C}[\![s]\!]$ refers to the term in which the hole is filled by term s.

We say term s_1 can be rewritten to term s_2 in one rewriting step ($s_1 \longrightarrow s_2$), if (i) there exist a context $\mathrm{C}[\![\;]\!]$ and terms s_1' and s_2' such that $s_1 = \mathrm{C}[\![s_1']\!]$ and $s_2 = \mathrm{C}[\![s_2']\!]$; and (ii) s_1' can be rewritten to s_2' according to some rewriting rule.

We say term s_1 can be rewritten to term s_2 in zero or more rewriting steps ($s_1 \twoheadrightarrow s_2$), if either (i) $s_1 = s_2$; or (ii) there exists a term s' such that $s_1 \longrightarrow s'$ and $s' \twoheadrightarrow s_2$.

A term s is *legal* if there exists $s_0 \in \mathrm{S}_0$ such that $s_0 \twoheadrightarrow s$. Since we are only interested in legal terms, we will drop the qualifier "legal" in our discussion.

A TRS is *confluent* if, for any term s_1, if $s_1 \twoheadrightarrow s_2$ and $s_1 \twoheadrightarrow s_3$, then there exists a term s_4 such that $s_2 \twoheadrightarrow s_4$ and $s_3 \twoheadrightarrow s_4$.

A TRS is *strongly terminating* if, for any term, it can always be rewritten to a normal form using any rewriting strategy.

3 $\mathcal{AX}$ Instruction Set Architecture

$\mathcal{AX}$ is a minimalist RISC instruction set (see Figure 1), in which all arithmetic operations are performed on registers and only the Load and Store instructions can access memory. Semantically instructions are executed strictly according to the program order: the program counter is increased by one each time an instruction is executed except for the Jz instruction. The informal meaning of the instructions is as follows:

The load-constant instruction $\mathsf{r}{:=}\mathsf{Loadc}(\mathsf{v})$ puts constant v into register r. The load-program-counter instruction $\mathsf{r}{:=}\mathsf{Loadpc}$ puts the content of the program counter into register r. The arithmetic-operation instruction $\mathsf{r}{:=}\mathsf{Op}(\mathsf{r}_1, \mathsf{r}_2)$ performs an arithmetic operation on operands specified by registers r_1 and r_2, and puts the result into register r. The branch instruction $\mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2)$ sets the program counter to the target instruction address specified by

INST	$\equiv$	$\mathsf{r} := \mathsf{Loadc}(\mathsf{v})$	*Load-constant Instruction*
	$[]$	$\mathsf{r} := \mathsf{Loadpc}$	*Load-program-counter Instruction*
	$[]$	$\mathsf{r} := \mathsf{Op}(\mathsf{r}_1, \mathsf{r}_2)$	*Arithmetic-operation Instruction*
	$[]$	$\mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2)$	*Branch Instruction*
	$[]$	$\mathsf{r} := \mathsf{Load}(\mathsf{r}_1)$	*Load Instruction*
	$[]$	$\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2)$	*Store Instruction*

Figure 1: $\mathcal{AX}$ Instruction Set

register r_2 if register r_1 contains value zero (otherwise the program counter is simply increased by one). The load instruction $\mathsf{r} := \mathsf{Load}(\mathsf{r}_1)$ reads the memory cell specified by register r_1, and puts the data into register r. The store instruction $\mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2)$ writes the content of register r_2 into the memory cell specified by register r_1.

Throughout the paper, we use '$[]$' as meta notation in grammars to separate disjuncts. We use 'a' and 'ia' to represent a data address and an instruction address, respectively. We use 'r' as a register name, 't' as a register renaming tag, and 'u' and 'v' as values. Subscripts will be used to distinguish domain elements whenever necessary. To avoid unnecessary complications, we assume that the instruction address space is disjoint from the data address space, so that self-modifying code is forbidden.

4 $\mathcal{P}_\mathcal{B}$ Model: Operational Semantics of $\mathcal{AX}$

In this section, we define $\mathcal{P}_\mathcal{B}$ (base processor), a single-cycle, non-pipelined, in-order execution model, and then give the operational semantics of the $\mathcal{AX}$ instruction set. The grammar of $\mathcal{P}_\mathcal{B}$ is given in Figure 2. The system has two components, a memory and a processor. The memory consists of a set of memory cells, where each memory cell has an address and a value. The processor consists of a program counter, a register file, and a program. The program counter holds the address of the instruction to be executed. The register file is a set of registers, where each register has a register name and a value. The program is a set of instructions, in which each instruction is associated with an instruction address.

In our notation, '$|$' is a constructor that is commutative and associative. We use 'ϵ' to represent the empty term, and '-' to represent the wild-card term that can match any term. We assume that instructions in a program have distinct instruction addresses, and use notation $\mathsf{prog}[\mathsf{ia}]$ to refer to the instruction with instruction address ia in the program prog. We assume that addresses in the memory are pairwise distinct, and so are register names in the register file. Notation $\mathsf{m}[\mathsf{a}]$ refers to the content of memory cell a, and notation $\mathsf{m}[\mathsf{a} := \mathsf{v}]$ represents memory m with memory cell a updated with value v. Similarly, notation $\mathsf{rf}[\mathsf{r}]$ refers to the content of register r, and notation $\mathsf{rf}[\mathsf{r} := \mathsf{v}]$ represents the register file that differs from rf only in the content of register r.

In the initial system term, the program counter is the address of the first instruction to be executed, and all registers and memory cells have the undefined value '$\perp$'. The following

SYS	$\equiv$	Sys(MEM, PROC)	*System*
MEM	$\equiv$	ϵ [] Cell(a,v) | MEM	*Memory*
PROC	$\equiv$	Proc(PC, RF, PROG)	*Processor*
PC	$\equiv$	ia	*Program Counter*
RF	$\equiv$	ϵ [] Reg(r, v) | RF	*Register File*
PROG	$\equiv$	ϵ [] Inst(ia, INST) | PROG	*Program*

Figure 2: $\mathcal{P}_\mathcal{B}$ Model

rewriting rules specify the operational semantics of the $\mathcal{AX}$ instruction set:

Loadc Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{r := Loadc(v)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf[r := v],\ prog)} \end{array}$$

Loadpc Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{r := Loadpc} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf[r := ia],\ prog)} \end{array}$$

Op Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{r := Op(r_1, r_2)} \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf[r := v],\ prog)} \quad \textit{where} \quad \mathsf{v} = \underline{\mathsf{Op}}\mathsf{(rf[r_1], rf[r_2])} \end{array}$$

Jz-Jump Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{Jz(r_1, r_2)} \quad \textit{and} \quad \mathsf{rf[r_1]} = 0 \\ \longrightarrow & \mathsf{Proc(rf[r_2],\ rf,\ prog)} \end{array}$$

Jz-NoJump Rule

$$\begin{array}{ll} & \mathsf{Proc(ia,\ rf,\ prog)} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{Jz(r_1, r_2)} \quad \textit{and} \quad \mathsf{rf[r_1]} \neq 0 \\ \longrightarrow & \mathsf{Proc(ia{+}1,\ rf,\ prog)} \end{array}$$

Load Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{r := Load(r_1)} \\ \longrightarrow & \mathsf{Sys(m,\ \ Proc(ia{+}1,\ rf[r := m[a]],\ prog))} \quad \textit{where} \quad \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

Store Rule

$$\begin{array}{ll} & \mathsf{Sys(m,\ \ Proc(ia,\ rf,\ prog))} \quad \textit{if} \quad \mathsf{prog[ia]} = \mathsf{Store(r_1, r_2)} \\ \longrightarrow & \mathsf{Sys(m[a := rf[r_2]],\ \ Proc(ia{+}1,\ rf,\ prog))} \quad \textit{where} \quad \mathsf{a} = \mathsf{rf[r_1]} \end{array}$$

Notice the memory access rules involve both the processor and the memory (*i.e.* the system), while other rules only deal with the processor. Notation $\underline{\mathsf{Op}}(\mathsf{v}_1, \mathsf{v}_2)$ represents the result of operation Op with operands v_1 and v_2.

5 $\mathcal{P}_\mathcal{R}$ Model: An Implementation with Register Renaming

Micro-architectures that do register renaming have a register renaming table and a set of instruction template buffers to hold instructions that have been issued and assigned register

SYS	$\equiv$	Sys(MEM, PROC)	*System*
MEM	$\equiv$	ϵ $[\!]$ Cell(a,v) | MEM	*Memory*
PROC	$\equiv$	Proc(PC, RF, ITBs, PROG)	*Processor*
PC	$\equiv$	ia $[\!]$ Stall	*Program Counter*
RF	$\equiv$	ϵ $[\!]$ Reg(r, tv) | RF	*Register File*
ITBs	$\equiv$	ϵ $[\!]$ ITB(ia, IT) $\oplus$ ITBs	*Instruction Template Buffers*
IT	$\equiv$	t := tv_1	
	$[\!]$	t := Op(tv_1, tv_2)	
	$[\!]$	Jz(tv_1, tv_2)	
	$[\!]$	t := Load(tv_1)	
	$[\!]$	Store(tv_1, tv_2)	*Instruction Template*
tv	$\equiv$	t $[\!]$ v	*Tag or Value*
PROG	$\equiv$	ϵ $[\!]$ Inst(ia, INST) | PROG	*Program*

Figure 3: $\mathcal{P}_{\mathcal{R}}$ Model

renaming tags but have not yet completed execution. Any instruction in the instruction template buffers can be executed if all its operands are available (as will be seen in a moment, some extra restrictions may apply to the execution of memory access operations). A natural consequence of register renaming is that instructions can be executed in a different order from the program order. $\mathcal{P}_{\mathcal{R}}$ (processor with renaming), our implementation for such a micro-architecture, uses the register file itself as the register renaming table. This is a common implementation trick in the absence of speculative execution.

The grammar of $\mathcal{P}_{\mathcal{R}}$ is given in Figure 3. Compared with the $\mathcal{P}_{\mathcal{B}}$ model, the main points to be noted are: (i) a new component, the instruction template buffers (ITBs), has been added to the processor; (ii) the program counter can either hold an instruction address, or be in the Stall state if the next instruction address has not been resolved yet; (iii) a register can hold either a value or a renaming tag. The ITBs is a sequence of instruction template buffers where each buffer consists of an instruction template and the associated instruction address. An instruction template is an instruction in which all register names have been appropriately replaced by either values or renaming tags. The ITBs is typically maintained as an ordered queue. We represent the queue using the constructor '$\oplus$', which is associative but not commutative. Initially the ITBs is empty.

5.1 Instruction Issue Rules

Instructions are issued in-order. When an instruction is issued, an instruction template is created in the ITBs with operand register names replaced with the corresponding values or renaming tags from the register file. If the instruction is to modify certain register, an unused renaming tag (t) is used to rename the destination register. This tag is placed in the destination register, which is overwritten later by either some value that is "committed" to the register, or another tag when some other instruction with the same destination register is issued.

$\mathcal{P}_{\mathcal{R}}$-Loadc-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{r} := \mathsf{Loadc}(\mathsf{v}) \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{t}],\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}, \mathsf{t} := \mathsf{v}),\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Loadpc-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{r} := \mathsf{Loadpc} \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{t}],\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}, \mathsf{t} := \mathsf{ia}),\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Op-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{r} := \mathsf{Op}(\mathsf{r}_1, \mathsf{r}_2) \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{t}],\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}, \mathsf{t} := \mathsf{Op}(\mathsf{rf}[\mathsf{r}_1], \mathsf{rf}[\mathsf{r}_2])),\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Jz-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{Jz}(\mathsf{r}_1, \mathsf{r}_2) \\ \longrightarrow & \mathsf{Proc}(\mathsf{Stall},\ \mathsf{rf},\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}, \mathsf{Jz}(\mathsf{rf}[\mathsf{r}_1], \mathsf{rf}[\mathsf{r}_2])),\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Load-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{r} := \mathsf{Load}(\mathsf{r}_1) \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf}[\mathsf{r} := \mathsf{t}],\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}, \mathsf{t} := \mathsf{Load}(\mathsf{rf}[\mathsf{r}_1])),\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Store-Issue Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{ia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{prog}[\mathsf{ia}] = \ \mathsf{Store}(\mathsf{r}_1, \mathsf{r}_2) \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia}{+}1,\ \mathsf{rf},\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}, \mathsf{Store}(\mathsf{rf}[\mathsf{r}_1], \mathsf{rf}[\mathsf{r}_2])),\ \mathsf{prog}) \end{array}$$

Notice that for the Jz instruction, the program counter is set to Stall, which will block the instruction issue until the target instruction address becomes available. The content of the program counter (ia) is recorded in the ITBs so that the address of the next instruction (ia+1) can be computed in case of "no jump".

In any implementation, there are a finite number of instruction template buffers and renaming tags. Instruction issue has to be stalled if all instruction template buffers are occupied, or no unused renaming tag is available to rename the destination register. This availability checking can be easily modeled, and we leave it as a simple exercise for the interested reader.

5.2 Arithmetic Operation and Value Propagation Rules

The arithmetic operation rule states that an arithmetic operation in the ITBs can be performed if both operands are available. It assigns the result to the corresponding tag.

$\mathcal{P}_{\mathcal{R}}$-Op Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{Op}(\mathsf{v}_1, \mathsf{v}_2)) \oplus \mathsf{itbs}_2,\ \mathsf{prog}) \\ \longrightarrow & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{v}) \oplus \mathsf{itbs}_2,\ \mathsf{prog}) \quad \textit{where} \quad \mathsf{v} = \underline{\mathsf{Op}}(\mathsf{v}_1, \mathsf{v}_2) \end{array}$$

The value propagation rules forward the value of a tag to other instruction templates and the register that contains this tag. Notation $\mathsf{itbs}_2[\mathsf{v}/\mathsf{t}]$ means that one or more appearances of tag t in itbs_2 are replaced by value v. Similarly, notation rf[v/t] refers to register file rf in which the register that contains tag t is overwritten with value v.

$\mathcal{P}_{\mathcal{R}}$-Value-Forward Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{v}) \oplus \mathsf{itbs}_2,\ \mathsf{prog}) \quad \textit{if} \quad \mathsf{t} \in \mathsf{itbs}_2 \\ \longrightarrow & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{v}) \oplus \mathsf{itbs}_2[\mathsf{v}/\mathsf{t}],\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Value-Commit Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{v}) \oplus \mathsf{itbs}_2,\ \mathsf{prog}) \quad \textit{if} \ \ \mathsf{t} \in \mathsf{rf} \\ \longrightarrow & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf}[\mathsf{v}/\mathsf{t}],\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{v}) \oplus \mathsf{itbs}_2,\ \mathsf{prog}) \end{array}$$

We also need the following rule so that a renaming tag can be retired and the associated instruction template buffer can be freed.

$\mathcal{P}_{\mathcal{R}}$-Tag-Retire Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \text{-}) \oplus \mathsf{itbs}_2,\ \mathsf{prog}) \quad \textit{if} \ \ \mathsf{t} \notin \mathsf{rf}, \mathsf{itbs}_2 \\ \longrightarrow & \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{itbs}_2,\ \mathsf{prog}) \end{array}$$

It is worth noting that the *$\mathcal{P}_{\mathcal{R}}$-Value-Commit* and *$\mathcal{P}_{\mathcal{R}}$-Tag-Retire* rules cannot create or destroy any other redex. Without them, the implementation would still be correct. However, an unbounded number of instruction template buffers and renaming tags would then be required.

5.3 Branch Completion Rules

The branch completion rules set the program counter appropriately according to the resolved branch condition.

$\mathcal{P}_{\mathcal{R}}$-Jz-Jump Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{Stall},\ \mathsf{rf},\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{Jz}(0, \mathsf{nia})),\ \mathsf{prog}) \\ \longrightarrow & \mathsf{Proc}(\mathsf{nia},\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Jz-NoJump Rule

$$\begin{array}{ll} & \mathsf{Proc}(\mathsf{Stall},\ \mathsf{rf},\ \mathsf{itbs} \oplus \mathsf{ITB}(\mathsf{ia}_1,\ \mathsf{Jz}(\mathsf{v}, \text{-})),\ \mathsf{prog}) \quad \textit{if} \ \ \mathsf{v} \neq 0 \\ \longrightarrow & \mathsf{Proc}(\mathsf{ia}_1{+}1,\ \mathsf{rf},\ \mathsf{itbs},\ \mathsf{prog}) \end{array}$$

5.4 Memory Access Rules

The memory access rules restrict the execution of a Load or Store operation to cases where there is no other memory access instructions ahead in the ITBs.

$\mathcal{P}_{\mathcal{R}}$-Load Rule

$$\begin{array}{ll} & \mathsf{Sys}(\mathsf{m},\ \ \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{Load}(\mathsf{a})) \oplus \mathsf{itbs}_2,\ \mathsf{prog})) \\ & \qquad\qquad \textit{if} \ \ \mathsf{Load}, \mathsf{Store} \notin \mathsf{itbs}_1 \\ \longrightarrow & \mathsf{Sys}(\mathsf{m},\ \ \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{t} := \mathsf{m}[\mathsf{a}]) \oplus \mathsf{itbs}_2,\ \mathsf{prog})) \end{array}$$

$\mathcal{P}_{\mathcal{R}}$-Store Rule

$$\begin{array}{ll} & \mathsf{Sys}(\mathsf{m},\ \ \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{ITB}(\mathsf{ia}_1, \mathsf{Store}(\mathsf{a}, \mathsf{v})) \oplus \mathsf{itbs}_2,\ \mathsf{prog})) \\ & \qquad\qquad \textit{if} \ \ \mathsf{Load}, \mathsf{Store} \notin \mathsf{itbs}_1 \\ \longrightarrow & \mathsf{Sys}(\mathsf{m}[\mathsf{a} := \mathsf{v}],\ \ \mathsf{Proc}(\mathsf{pc},\ \mathsf{rf},\ \mathsf{itbs}_1 \oplus \mathsf{itbs}_2,\ \mathsf{prog})) \end{array}$$

Memory access instructions can be implemented more aggressively while still preserving the semantics for single processor systems. For example, allowing a Load operation to be performed with outstanding Store operations can effectively model FIFO write buffers. Furthermore, allowing a Store operation to be performed with outstanding Store operations on different addresses allows the write buffers to be non-FIFO. However, these implementations can produce very different storage models in multiprocessor systems. We do not have space to explore this issue any further. Interested readers are referred to [3] [8] for a thorough discussion.

Discussion: The $\mathcal{P}_{\mathcal{R}}$ model, a micro-architecture that uses register renaming and allows out-of-order instruction execution, is remarkably simpler and more precise than what one may find in a modern textbook. A part of the simplicity arises from the fact that we have intentionally ignored some architectural features such as precise pipeline stages, potential structural hazards and the mechanism of finding the corresponding register for a renaming tag. The $\mathcal{P}_{\mathcal{R}}$ model can be extended to include all such features as well as speculative instruction execution capability. TRS's seem to be a very natural way to model parallel and asynchronous systems.

6 Correctness Proof of the $\mathcal{P}_{\mathcal{R}}$ Model

In this section, we demonstrate that the $\mathcal{P}_{\mathcal{B}}$ and $\mathcal{P}_{\mathcal{R}}$ models can "simulate" each other in regards to the *programmer visible states*, which include the program counter, the register file and the memory. By simulation, we mean intuitively that there exists a mapping between the configurations of the two models such that if we run a program on one model and take a snapshot of the programmer visible states at any time during the execution, we can observe the same states if we run the program on the other model and take a snapshot at an appropriate time. One can imagine a print instruction that can print the content of the program counter, a register, or a memory location. If model A can simulate model B, then for any program, model A should print exactly what model B prints throughout the execution. The following proof is rigorous though some technical details have been omitted. The complete formal proof can be found in [8].

It is easy to show that the $\mathcal{P}_{\mathcal{R}}$ model can simulate the $\mathcal{P}_{\mathcal{B}}$ model. A $\mathcal{P}_{\mathcal{B}}$ system term can be mapped (lifted) to a $\mathcal{P}_{\mathcal{R}}$ system term by simply adding an empty ITBs in the processor. We call this mapping function ITBL (instruction-template-buffer-lift). Clearly, ITBL preserves the programmer visible states. It is also easy to see that any rule of $\mathcal{P}_{\mathcal{B}}$ can be simulated by a sequence of $\mathcal{P}_{\mathcal{R}}$ rules. For example, applying the *Op* rule in $\mathcal{P}_{\mathcal{B}}$ can be simulated by applying the $\mathcal{P}_{\mathcal{R}}$-*Op-Issue*, $\mathcal{P}_{\mathcal{R}}$-*Op*, $\mathcal{P}_{\mathcal{R}}$-*Value-Commit* and $\mathcal{P}_{\mathcal{R}}$-*Tag-Retire* rules in $\mathcal{P}_{\mathcal{R}}$. The simulation theorem is stated as follows:

Theorem 1 Let s_1 and s_2 be system terms in $\mathcal{P}_{\mathcal{B}}$. If $s_1 \twoheadrightarrow s_2$ in $\mathcal{P}_{\mathcal{B}}$, then $\mathsf{ITBL}(s_1) \twoheadrightarrow \mathsf{ITBL}(s_2)$ in $\mathcal{P}_{\mathcal{R}}$.

The simulation in the other direction requires some thought. We define function ITBF (instruction-template-buffer-free), which maps $\mathcal{P}_{\mathcal{R}}$ system terms to $\mathcal{P}_{\mathcal{B}}$ system terms while preserving the programmer visible states. The idea behind function ITBF is that, for any $\mathcal{P}_{\mathcal{R}}$ system term, we can rewrite the term to a normal form in which the ITBs is empty. The normal form can be mapped (projected) to a $\mathcal{P}_{\mathcal{B}}$ term by simply deleting the empty ITBs. The theorem is stated as follows:

Theorem 2 Let s_1 and s_2 be system terms in $\mathcal{P}_\mathcal{R}$. If $s_1 \twoheadrightarrow s_2$ in $\mathcal{P}_\mathcal{R}$, then $\mathsf{ITBF}(s_1) \twoheadrightarrow \mathsf{ITBF}(s_2)$ in $\mathcal{P}_\mathcal{B}$.

6.1 Instruction-Template-Buffer-Free Function

Intuitively, with instruction issue stalled, the ITBs will sooner or later become empty as instruction execution proceeds. When the ITBs becomes empty, no tag can exist in the register file, and the program counter cannot be in the Stall state. ITBF is based on the observation that we can always make a $\mathcal{P}_\mathcal{R}$ system term "instruction-template-buffer-free" by applying non-instruction-issue rules. This motivates us to define another rewriting system $\mathcal{R}_{\mathcal{ITBF}}$ which uses the same grammar as the $\mathcal{P}_\mathcal{R}$ model and includes all the $\mathcal{P}_\mathcal{R}$ rules except the instruction issue rules.

Definition 3 $\mathcal{R}_{\mathcal{ITBF}} \equiv$ { *$\mathcal{P}_\mathcal{R}$-Op, $\mathcal{P}_\mathcal{R}$-Value-Forward, $\mathcal{P}_\mathcal{R}$-Value-Commit, $\mathcal{P}_\mathcal{R}$-Tag-Retire, $\mathcal{P}_\mathcal{R}$-Jz-Jump, $\mathcal{P}_\mathcal{R}$-Jz-NoJump, $\mathcal{P}_\mathcal{R}$-Load, $\mathcal{P}_\mathcal{R}$-Store* }

It can be shown by simple induction and case analysis that for any $\mathcal{P}_\mathcal{R}$ system term, rewriting with respect to $\mathcal{R}_{\mathcal{ITBF}}$ terminates within a finite number of steps, and always reaches the same normal form regardless of the order in which the rules are applied. In TRS jargon, $\mathcal{R}_{\mathcal{ITBF}}$ is said to be strongly terminating and confluent.

It can be furthermore proved by induction that in the normal form, the ITBs is empty, no renaming tag exists in the register file, and the program counter contains an instruction address. We define $\mathsf{ITBF}(s)$ as "compute the normal form of s with respect to $\mathcal{R}_{\mathcal{ITBF}}$ and then delete the empty ITBs". It is trivial to show that ITBF maps the initial $\mathcal{P}_\mathcal{R}$ term to the initial $\mathcal{P}_\mathcal{B}$ term. As can be seen, for any $\mathcal{P}_\mathcal{B}$ system term s, $\mathsf{ITBF}(\mathsf{ITBL}(s)) = s$.

6.2 Simulate $\mathcal{P}_\mathcal{R}$ in $\mathcal{P}_\mathcal{B}$

In the remainder of this section, we prove by induction on rewriting steps that $\mathcal{P}_\mathcal{B}$ can simulate $\mathcal{P}_\mathcal{R}$. Assume $s_1 \longrightarrow s_2$ in $\mathcal{P}_\mathcal{R}$ by applying rule α. There are two cases on α:

- $\alpha \in \mathcal{R}_{\mathcal{ITBF}}$. Needless to say, $\mathsf{ITBF}(s_1)$ and $\mathsf{ITBF}(s_2)$ are identical.

- $\alpha \notin \mathcal{R}_{\mathcal{ITBF}}$ (*i.e.* α is an instruction issue rule). In this case, we can show that an appropriate $\mathcal{P}_\mathcal{B}$ rule can be applied to $\mathsf{ITBF}(s_1)$ to yield a term which is equal to $\mathsf{ITBF}(s_2)$.

  Suppose $s_1 \longrightarrow s_3$ by applying some $\mathcal{R}_{\mathcal{ITBF}}$ rule. It can be seen by inspecting the $\mathcal{R}_{\mathcal{ITBF}}$ rules that α can also be applied to s_3. Assume $s_3 \longrightarrow s_4$ by applying α, then $s_2 \twoheadrightarrow s_4$ by applying some $\mathcal{R}_{\mathcal{ITBF}}$ rules. This follows from the fact that the $\mathcal{R}_{\mathcal{ITBF}}$ rules and the instruction issue rules are non-interfering. Let s_n be the normal form of s_1 with respect to $\mathcal{R}_{\mathcal{ITBF}}$. It can be proved by induction that α can be applied to s_n to yield s_{n+1} such that $s_2 \twoheadrightarrow s_{n+1}$ by applying $\mathcal{R}_{\mathcal{ITBF}}$ rules (see Figure 4).

  Furthermore, suppose s_{n+2} is the normal form of s_{n+1} with respect to $\mathcal{R}_{\mathcal{ITBF}}$. Since s_n and s_{n+2} both have empty instruction template buffers, it is easy to show that $\mathsf{ITBF}(s_n)$

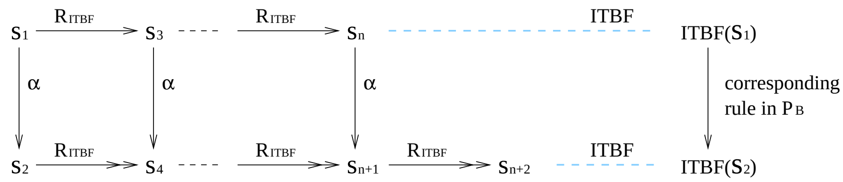

Figure 4: Simulate Instruction Issue Rule α

$\longrightarrow$ $\mathsf{ITBF}(s_{n+2})$ by applying the corresponding $\mathcal{P_B}$ rule (see Figure 4). The table below gives the correspondence between the $\mathcal{P_B}$ rules and the instruction issue rules of $\mathcal{P_R}$. Notice the $\mathcal{P_R}$-*Jz-Issue* rule corresponds to either the *Jz-Jump* or *Jz-NoJump* rule in $\mathcal{P_B}$, depending on whether the branch is taken or not.

$\mathcal{P_R}$ instruction issue rule	corresponding $\mathcal{P_B}$ rule
$\mathcal{P_R}$-*Loadc-Issue rule*	*Loadc rule*
$\mathcal{P_R}$-*Loadpc-Issue rule*	*Loadpc rule*
$\mathcal{P_R}$-*Op-Issue rule*	*Op rule*
$\mathcal{P_R}$-*Jz-Issue rule*	*Jz-Jump/Jz-NoJump rule*
$\mathcal{P_R}$-*Load-Issue rule*	*Load rule*
$\mathcal{P_R}$-*Store-Issue rule*	*Store rule*

This completes the proof that if $s_1 \twoheadrightarrow s_2$ in $\mathcal{P_R}$, then $\mathsf{ITBF}(s_1) \twoheadrightarrow \mathsf{ITBF}(s_2)$ in $\mathcal{P_B}$. $\square$

The two simulation theorems together imply the correctness of the $\mathcal{P_R}$ model with respect to the $\mathcal{P_B}$ model. From a pragmatic point of view it is also important to specify a rewriting strategy in which a redex is rewritten within a finite number of steps. Any reasonable implementation can easily satisfy this requirement.

7 Research In Progress and Related Work

This work is a byproduct of our effort to design provably correct cache coherence protocols for distributed shared memory systems. It was motivated by our desire to incorporate program and processor behavior in the specification of memory models. In [9] we defined sequential consistency based on the $\mathcal{P_B}$ model, and designed a family of cache coherence protocols for a distributed shared-memory system with hierarchical caches. The correctness of the cache coherence protocols was proved by showing that the TRS's for the protocols and the memory model can simulate each other. Our experience shows that the technique not only makes protocol verification more systematic, but also helps us in designing adaptive protocols by successive refinement.

It is worth emphasizing that the proof technique is quite general and the definition of the mapping (lifting and projecting) functions is usually straightforward. A more sophisticated processor implementation with speculative execution was verified in [8] using the same technique. The method can also be used to verify realistic pipeline machines. Our effort is now focused on the processor-memory interface since our main research interest is to explore more aggressive implementations of memory access and synchronization instructions in multiprocessor systems.

The use of formal techniques in designing systems partially depends upon the tools available to support the technique. We have just begun the investigation of appropriate tools to support our technique so that tedious case analysis can be performed by machine. It should be possible to build or connect to a model checker type of tool to explore all the reductions of a given term. Model checkers like Murphi [2] verify assertions by exploring a finite state graph. When a problem can be expressed without using too many states, such tools have proven very useful as debuggers for engineers in verifying properties of their designs.

Many of our systems can be expressed using other formal techniques such as I/O automata [6]. Techniques based on general theorem proving systems, such as HOL, let the user express more general assertions but require more help from the user in actually doing the proofs. Like TRS, assertions in none of these formalisms can be automated fully due to the infinite number of states. Nevertheless, useful tools such as FDR are available to verify that an implementation satisfies its specifications.

Formal verification of microprocessors has gained considerable attention in recent years. For example, Burch and Dill [1] described a technique which automatically compares a pipelined implementation to an architectural specification and produces debugging information for incorrect processor design. Levitt and Olukotun [5] proposed a methodology that iteratively de-constructs a pipeline by merging adjacent pipeline stages thereby allowing verifications to be done in a number of easier steps. Windley [10] presented a case study which uses abstract theories to hierarchically verify microprocessor implementations formalized in HOL. While most of the previous work has focused on pipeline processors, one contribution of this paper is to show that features such as register renaming and write buffers can be easily modeled so that their impact in multiprocessor systems can be investigated conveniently.

Windley's methodology is similar to ours, in the sense that his correctness theorem states the implementation specification implies the behavior specification. The most critical step in the proof is the definition of the abstract mapping function such as ITBF. With our technique, the definition of this function is very straightforward. For the examples we have tried, it can always be expressed as the normal form with respect to a subset of the existing rules.

Acknowledgement We would like to thank Joe Stoy, Larry Rudolph, Alejandro Caro and James Hoe for reading an earlier draft of this paper. Members of the Computation Structures Group (CSG) have provided an extremely stimulating environment to discuss the ideas presented here.

Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Ft Huachuca contract DABT63-95-C-0150.

References

[1] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *International Conference on Computer-Aided Verification*, June 1994.

[2] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[3] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. Phd. thesis, Stanford University, 1995.

[4] J. W. Klop. Term Rewriting System. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[5] J. Levitt and K. Olukotun. A Scalable Formal Verification Methodology for Pipelined Microprocessors. In *33nd ACM IEEE Design Automation Conference*, June 1996.

[6] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[7] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for A New Family of RISC Processors.* Morgan Kaufmann, 1994.

[8] X. Shen and Arvind. Processor Models. CSG Memo 400, Laboratory for Computer Science, MIT, June 1997.

[9] X. Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. CSG Memo 398, Laboratory for Computer Science, MIT, June 1997.

[10] P. J. Windley. Formal Modeling and Verification of Microprocessors. *IEEE Transactions on Computers*, 44(1), Jan. 1995.