LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Using Term Rewriting Systems to Design and Verify Processors

Computation Structures Group Memo 419
November 1998

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

To appear in IEEE Micro Special Issue on "Modeling and Validation of Microprocessors", May/June 1999.

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

Using Term Rewriting Systems to Design and Verify Processors

Arvind and Xiaowei Shen
Laboratory for Computer Science
Massachusetts Institute of Technology
arvind, xwshen@lcs.mit.edu

Abstract

We present a novel use of Term Rewriting Systems (TRS's) to describe micro-architectures. The state of a system is represented as a TRS term while the state transitions are represented as TRS rules. TRS descriptions are amenable to both verification and synthesis. We illustrate the use of TRS's by giving the operational semantics of a simple RISC instruction set. We then present another TRS that implements the same instruction set on a micro-architecture which permits register renaming and speculative execution. The correctness of the speculative implementation is discussed in terms of the ability of the two TRS's to simulate each other. Our method facilitates understanding of important micro-architectural differences without delving into low-level implementation details.

1 Introduction

Term Rewriting Systems (TRS's) offer a convenient way to describe parallel and asynchronous systems, and can be used to prove the correctness of an implementation with respect to a specification. TRS descriptions, augmented with proper information about the building blocks, also hold the promise of high-level synthesis. High-level architectural descriptions, which are both automatically synthesizable and verifiable, open up the possibility of architectural exploration at a fraction of the time and cost than what is feasible using current commercial tools.

Formal verification of microprocessors has gained considerable attention in recent years [2, 3, 7, 11]. Other formal techniques, such as Lamport's TLA and Lynch's I/O automata, can also be used to model microprocessors. While all these techniques have something in common with TRS's, we find the use of TRS's more intuitive in both architecture descriptions and correctness proofs. TRS's can be used to describe both deterministic and non-deterministic computations. Although they have been used extensively in programming language research to give operational semantics, their use in architectural descriptions is novel.

In this paper, we will use TRS's to describe a speculative processor capable of register renaming and out-of-order execution. The lack of space does not permit us to discuss a synthesis procedure from TRS's or to give sufficient details that are needed to make automatic synthesis feasible. Nevertheless, we will show that our speculative processor produces the same set of behaviors as a simple non-pipelined implementation. Though the reader is the ultimate judge, we believe that our descriptions of micro-architectures are more precise than what one may find in a modern textbook [4]. It is the clarity of these descriptions that lets us study the impact of features such as write buffers or caches, especially in multiprocessor systems [10, 9]. In fact, part of the motivation for this work came from one of the author's experience in teaching computer architectures.

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 a set of initial terms ($S_0 \subseteq S$). 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:

$$\begin{array}{ll} & s_1 \quad \text{if} \quad p\,(s_1) \\ \rightarrow & s_2 \end{array}$$

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

A rule can be used to rewrite a term if the left-hand-side pattern of the rule matches the term or one of its subterms, and the corresponding predicate is true. The new term is generated in accordance with the right-hand-side of the rule. 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. In practice, we often use abstract data types such as arrays and FIFO queues to make the descriptions more readable. More information about TRS's can be found elsewhere [1, 6].

A small but fascinating example of term rewriting is provided by the SK combinatory system, which has only two rules, and a simple grammar for generating terms. These two rules are sufficient to describe any computable function!

$$\text{TERM} \equiv \;\; \text{K} \;\|\; \text{S} \;\|\; \text{TERM.TERM}$$

$$\begin{array}{lll} \textit{K-rule:} & (\text{K}.x).y & \rightarrow \; x \\ \textit{S-rule:} & ((\text{S}.x).y).z & \rightarrow \; (x.z).(y.z) \end{array}$$

The interested reader may want to verify that, for any subterm x, the term ((S.K).K).x can be rewritten to (K.x).(K.x) by applying the S-rule. This term can be rewritten further to x by applying the K-rule. Thus, if one reads the '.' as function application then the term ((S.K).K) behaves as the identity function.
Notice the S-rule rearranges '.' and duplicates the term represented by x on the right-hand-side. In architectures where terms represent states, rules must be restricted so that terms are not restructured or duplicated.

3 AX: A Minimalist RISC Instruction Set

We will use AX, a minimalist RISC instruction set, to illustrate all the examples in this paper. The TRS description of a simple AX architecture also provides a good introductory example to the TRS notation.

In the AX instruction set (see Figure 1), all arithmetic operations are performed on registers and only the Load and Store instructions are allowed to access memory. The grammar uses '$\|$' as a meta notation to separate disjuncts. Throughout the paper 'r' represents a register name, 'v' a value, 'a' a data memory address and 'ia' an instruction memory address. An identifier may be qualified with a subscript. We do not specify the number of registers, the number of bits in a register or value, or the exact bit-format of each instruction. Such details are not necessary for a high-level description of a micro-architecture but are needed for synthesis.

INST	$\equiv$	r:=Loadc(v)	*Load-constant Instruction*
	$\|$	r:=Loadpc	*Load-program-counter Instruction*
	$\|$	r:=Op(r_1,r_2)	*Arithmetic-operation Instruction*
	$\|$	Jz(r_1,r_2)	*Branch Instruction*
	$\|$	r:=Load(r_1)	*Load Instruction*
	$\|$	Store(r_1,r_2)	*Store Instruction*

Figure 1: AX Instruction Set

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. AX is powerful enough to let us express all computations as location independent, non-self-modifying programs.

Semantically, AX instructions are executed strictly according to the program order: the program counter is incremented by one each time an instruction is executed except for the Jz instruction, where the program counter is set appropriately according to the branch condition. The informal meaning of the instructions is as follows:

The load-constant instruction r:=Loadc(v) puts constant v into register r. The load-program-counter instruction r:=Loadpc puts the content of the program counter into register r. The arithmetic-operation instruction r:=Op(r_1,r_2) performs the arithmetic operation specified by Op on the operands specified by registers r_1 and r_2, and puts the result into register r. The branch instruction Jz(r_1,r_2) sets the program counter to the target instruction address specified by register r_2 if register r_1 contains value zero; otherwise the program counter is simply increased by one. The load instruction r:=Load(r_1) reads the memory cell specified by register r_1, and puts the data into register r. The store instruction Store(r_1,r_2) writes the content of register r_2 into the memory cell specified by register r_1.

We define the operational semantics of AX instructions using the $\mathrm{P_B}$ model, a single-cycle, non-pipelined, in-order execution processor. The datapath for such a system is shown in Figure 2. The processor consists of a program counter (*pc*), a register file (*rf*), and an instruction memory (*im*). The program counter holds the address of the instruction to be executed. The processor together with the data memory (*dm*) constitutes the whole system, which can be represented as the TRS term Sys(Proc(*pc*, *rf*, *im*), *dm*). The semantics of each instruction can be given as a rewriting rule which specifies how the state is modified after each instruction is executed.

It is important to realize that *pc*, *rf*, *im* and *dm* can be grouped syntactically in any convenient way. Grouping them as Sys(Proc(*pc*, *rf*, *im*), *dm*) instead of Sys(*pc*, *rf*, *im*, *dm*) provides a degree of modularity in describing the rules that do not refer to *dm*. Abstract data types can also enhance modularity. For example, *rf*, *im* and *dm* are all represented using arrays on which only two operations, selection and update, can be performed. Thus, *rf*[*r*] refers to the content of register *r*, and *rf*[*r*:=*v*] represents the register file after register *r* has been updated with value *v*. Similarly, *dm*[*a*] refers to the content of memory location *a*, and *dm*[*a*:=*v*] represents the memory with location *a* updated with value *v*.

We use the following notational conventions in the rewriting rules: all the special symbols such as ':=', and all the identifiers that start with capital letters are treated as constants in pattern matching. We use '-' to represent the wild-card term that can match any term. Notation $\underline{\mathrm{Op}}(v_1,v_2)$ represents the result of operation Op with operands v_1 and v_2.

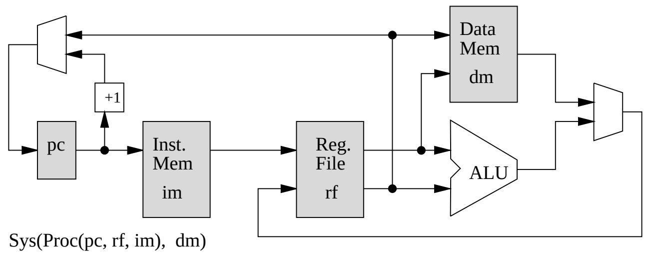


Figure 2: The $\mathrm{P_B}$ Model: A Single-Cycle In-Order Processor

Loadc Rule
$\quad$ Proc(ia, rf, im) $\quad$ if $\quad im[ia] = \; r$:=Loadc(v)
$\rightarrow$ Proc(ia+1, $rf[r:=v]$, im)

Loadpc Rule
$\quad$ Proc(ia, rf, im) $\quad$ if $\quad im[ia] = \; r$:=Loadpc
$\rightarrow$ Proc(ia+1, $rf[r:=ia]$, im)

Op Rule
$\quad$ Proc(ia, rf, im) $\quad$ if $\quad im[ia] = \; r$:=Op(r_1,r_2)
$\rightarrow$ Proc(ia+1, $rf[r:=v]$, im) $\quad$ where $\quad v = \underline{\mathrm{Op}}(rf[r_1],rf[r_2])$

Jz-Jump Rule
$\quad$ Proc(ia, rf, im) $\quad$ if $\quad im[ia] =$ Jz(r_1,r_2) $\;$ and $\; rf[r_1] = 0$
$\rightarrow$ Proc($rf[r_2]$, rf, im)

Jz-NoJump Rule
$\quad$ Proc(ia, rf, im) $\quad$ if $\quad im[ia] =$ Jz(r_1,r_2) $\;$ and $\; rf[r_1] \neq 0$
$\rightarrow$ Proc(ia+1, rf, im)

Load Rule
$\quad$ Sys(Proc(ia, rf, im), dm) $\quad$ if $\quad im[ia] = \; r$:=Load(r_1)
$\rightarrow$ Sys(Proc(ia+1, $rf[r:=dm[a]]$, im), dm) $\quad$ where $\quad a = rf[r_1]$

Store Rule
$\quad$ Sys(Proc(ia, rf, im), dm) $\quad$ if $\quad im[ia] =$ Store(r_1,r_2)
$\rightarrow$ Sys(Proc(ia+1, rf, im), $dm[a:=rf[r_2]]$) $\quad$ where $\quad a = rf[r_1]$

Since the pattern Proc(ia, rf, im) will match any processor term, the real discriminant is the instruction at address ia. In the case of a branch instruction, further discrimination is made based on the value of the condition register.

It is important to understand the atomic nature of these rules. Once a rule is applied, the state specified by its right-hand-side must be reached before any other rule can be applied. For example, on an Op instruction, both operands must be fetched and the result computed and stored in the register file in one atomic action. Furthermore, the program counter must be updated during this atomic action as well. This is why these rules describe a single-cycle, non-pipelined implementation of AX.

To save space, we may use a table to describe the rules informally. For example, Figure 3 summarizes the $\mathrm{P_B}$ rules given above. It is our hope that, given proper context, the reader will be able to deduce the precise TRS rules from a tabular description.

Rule Name	Instruction at ia	Next pc	Next rf	Next dm
Loadc	r:=Loadc(v)	ia+1	$rf[r:=v]$	dm
Loadpc	r:=Loadpc	ia+1	$rf[r:=ia]$	dm
Op	r:=Op(r_1,r_2)	ia+1	$rf[r:=\underline{\mathrm{Op}}(rf[r_1],rf[r_2])]$	dm
Jz	Jz(r_1,r_2)	ia+1 (if $rf[r_1] \neq 0$) $rf[r_2]$ (if $rf[r_1] = 0$)	rf	dm
Load	r:=Load(r_1)	ia+1	$rf[r:=dm[rf[r_1]]]$	dm
Store	Store(r_1,r_2)	ia+1	rf	$dm[rf[r_1]:=rf[r_2]]$

Figure 3: Operational Semantics of AX (Current State: Sys(Proc(ia, rf, im), dm))

4 Register Renaming and Speculative Execution

There are many possible micro-architectures that can implement the AX instruction set. For example, in a simple pipelined architecture, instructions are fetched, executed and retired in order, and there can be as many as 4 or 5 partially executed instructions in the processor. Storage in the form of pipeline buffers is provided to hold these partially executed instructions. In more sophisticated pipelined architectures, there are multiple functional units, which may be specialized for integer or floating-point calculations. In such architectures, even if instructions are issued in order, they may complete out of order because of varying latencies of functional units. To preserve correctness, a new instruction is not issued when there is another instruction in the pipeline that may update any register to be read or written by the new instruction. Seymore Cray's CDC 6600, which is one of the earliest examples of such an architecture, used a *scoreboard* to dispatch and track partially executed instructions in the processor. In Cray-style scoreboard design, the number of instructions in the pipeline is limited by the number of registers in the instructions set.

The technique of register renaming was invented by Tomasulo at IBM in mid sixties to overcome this limitation on pipelining. Tomasulo assigned a renaming tag to each instruction as it was decoded. The following instructions used this tag to refer to the value produced by this instruction. A renaming tag became free, i.e., could be used again, once the instruction was completed. The micro-architecture maintained the association between the register name, the tag and the associated value (whenever the value became available). This innovative idea was embodied in IBM 360/91 in late sixties but went out of favor until late eighties for several reasons. For example, the performance gains were not considered commensurate with the complexity of the implementations. Register renaming is common place today and present in all the high-end microprocessors (PentiumPro, PowerPC 604, MIPS R10000 and later models in these product lines).

An important state element in a micro-architecture with register renaming is a reorder buffer (ROB), which holds instructions that have been decoded but have not completed their execution (see Figure 4). Conceptually, ROB divides the processor into two asynchronous parts: the first part fetches an instruction and, after decoding and renaming registers, dumps it into the next available slot in the ROB. The ROB slot index serves the purpose of the renaming tag, and the instructions in the ROB always contain tags or values instead of register names. An instruction in the ROB can be executed if all its operands are available. The second part takes any *enabled* instruction out of the ROB and dispatches it to an appropriate functional unit, including the memory system. This mechanism is very similar to the execution mechanism in dataflow architectures. Such an architecture may execute instructions out of order, especially if functional units have different latencies or there are data dependencies between instructions.

In addition to register renaming, most contemporary microprocessors also permit speculative

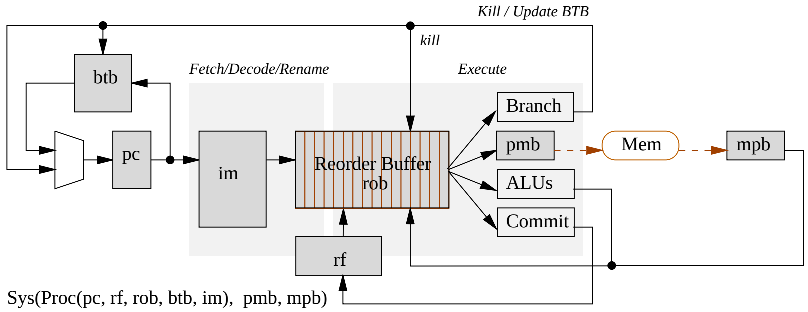


Figure 4: The $\mathrm{P_S}$ Model: A Processor with Register Renaming and Speculative Execution

execution of instructions. The speculative mechanisms predict the address of the next instruction to be issued based on the past behavior of the program. (Several researchers have recently suggested mechanisms to speculate on memory values as well but none of these have been implemented so far; we do not consider such mechanisms in this paper). The address of the speculative instruction is determined by consulting a table known as the branch target buffer (BTB), which can be indexed by the current content of the program counter. If the prediction turns out to be wrong, the speculative instruction and all the instructions issued thereafter are abandoned and their effect on the processor state nullified. The BTB is updated according to some prediction scheme after each branch resolution.

The correctness of the speculative processor is not contingent upon how the BTB is maintained, as long as the program counter can be set to the correct value after a misprediction. However, different prediction schemes can give rise to very different misprediction rates and thus have profound influence on the performance. Generally, it is assumed that the BTB produces the correct next instruction address for all non-branch instructions. We will not discuss the BTB any further because the branch prediction strategy is completely orthogonal to the mechanisms for speculative execution.

Any processor that permits speculative execution has to make sure that a speculative instruction either does not modify the programmer visible state until it can be "committed", or save enough of the processor state so that the correct state can be restored in case the speculation turns out to be wrong. Most implementations use a mixture of these two ideas: speculative instructions do not modify the register file or memory until it can be determined that the prediction is correct, but are allowed to update the program counter. Both the current and the speculated instruction address are recorded so that the correctness of speculation can be determined later, and the correct program counter can be restored in case of wrong prediction. Typically, all the temporary state is maintained in the ROB itself.

5 $\mathrm{P_S}$: A Speculative Processor

We now present the rules for a simplified micro-architecture that does register renaming and speculative execution. The simplification is achieved by not showing all the pipelining and not giving the details of some hardware operations. The memory system is modeled as operating asynchronously with respect to the processor. Thus, a memory instruction in the ROB is dispatched to the memory system via an ordered processor-to-memory buffer (pmb); the memory provides its responses via a memory-to-processor buffer (*mpb*). Exactly how the memory system is organized is not discussed. However, memory system details can be added in a modular fashion without changing the processor description presented here [10, 9].

We need to add two new components, *rob* and *btb*, to the processor state. Reorder buffer *rob* is a complex device to model because different types of operations need to be performed on it. It can be thought of as a FIFO queue which is initially empty (ϵ). We use the constructor '$\oplus$', which is associative but not commutative, to represent this aspect of *rob*. It can also be considered as an array of instruction templates where array index serves the purpose of a renaming tag. It is well known that a FIFO queue can be implemented as a circular buffer using two pointers into an array. We will hide these implementation details of *rob* and assume that the next available tag can be obtained.

An instruction template in *rob* contains the instruction address, opcode, operands and some extra information needed to complete the instruction. For instructions that need to update a register, the Wr(r) field records the destination register r. For branch instructions, the Sp(pia) field holds the speculated instruction address pia which will be used to determine the correctness of the prediction. Each memory access instruction maintains an extra flag to indicate whether the instruction is waiting to be dispatched (U), or has been dispatched to the memory (D). The memory system returns a value for a load and an acknowledgment (Ack) for a store. We have taken some syntactic liberties in expressing various types of instruction templates below:

$$
\begin{array}{lll}
\text{ROB Entry} & \equiv & \text{Itb}(ia, t{:=}v, \text{Wr}(r)) \\
 & [] & \text{Itb}(ia, t{:=}\text{Op}(tv_1, tv_2), \text{Wr}(r)) \\
 & [] & \text{Itb}(ia, \text{Jz}(tv_1, tv_2), \text{Sp}(pia)) \\
 & [] & \text{Itb}(ia, t{:=}\text{Load}(tv_1, mf), \text{Wr}(r)) \\
 & [] & \text{Itb}(ia, t{:=}\text{Store}(tv_1, tv_2, mf))
\end{array}
$$

where tv stands for either a tag or a value, and the memory flag mf is either U or D. The tag used in the Store instruction template is intended to provide some flexibility in coordinating with the memory system, and does not imply updating of any register.

5.1 Instruction Fetch Rules

Each time an instruction is issued, the program counter is set to the address of the next instruction to be issued. For non-branch instructions, the program counter is simply incremented by one. Speculative execution happens when a Jz instruction is issued: the program counter is then set to the instruction address obtained by consulting the *btb* entry corresponding to the address of the Jz instruction.

When an instruction is issued, an instruction template for the issued instruction is allocated in the *rob*. If the instruction is to modify a register, an unused renaming tag (typically the index of the slot in the *rob*) is used to rename the destination register and the destination register is recorded in the Wr field. The tag or value of each operand register is found by searching the *rob* from the youngest buffer (rightmost) to the oldest buffer (leftmost) until an instruction template containing the referenced register is found. If no such buffer exists in the *rob*, then the most up-to-date value resides in the register file. The following lookup procedure captures this idea:

$$
\begin{array}{l}
\text{lookup}(r, rf, rob) \\
\qquad \equiv \quad rf[r] \qquad \text{if } \text{Wr}(r) \notin rob \\
\text{lookup}(r, rf, rob_1 \oplus \text{Itb}(ia, t{:=}\text{-}, \text{Wr}(r)) \oplus rob_2) \\
\qquad \equiv \quad t \qquad\quad \text{if } \text{Wr}(r) \notin rob_2
\end{array}
$$

Rule Name	Instruction at ia	New Template in rob	Next pc
Fetch-Loadc	r:=Loadc(v)	Itb(ia,t:=v,Wr(r))	ia+1
Fetch-Loadpc	r:=Loadpc	Itb(ia,t:=ia,Wr(r))	ia+1
Fetch-Op	r:=Op(r_1,r_2)	Itb(ia,t:=Op(tv_1,tv_2),Wr(r))	ia+1
Fetch-Jz	Jz(r_1,r_2)	Itb(ia,Jz(tv_1,tv_2),Sp($btb[ia]$))	$btb[ia]$
Fetch-Load	r:=Load(r_1)	Itb(ia,t:=Load(tv_1,U),Wr(r))	ia+1
Fetch-Store	Store(r_1,r_2)	Itb(ia,t:=Store(tv_1,tv_2,U))	ia+1

Figure 5: $\mathrm{P_S}$ Instruction Fetch Rules (Current state: Proc(ia, rf, rob, btb, im))

It is beyond the scope of this paper to give a hardware implementation of this procedure but it is certainly possible to do so using TRS's. Any implementation that can look up values in the *rob* using a combinational circuit would suffice.

For example, the fetch rule for an Op instruction simply puts the instruction after register renaming at the end of the *rob* as follows:

Fetch-Op Rule

$\quad$ Proc(ia, rf, rob, btb, im) $\quad$ if $\quad im[ia] = \ r$:=Op(r_1,r_2)

$\rightarrow$ Proc(ia+1, rf, $rob \oplus$Itb(ia,t:=Op(tv_1,tv_2),Wr(r)), btb, im)

where t represents an unused tag; tv_1 and tv_2 represent the tag or value corresponding to the operand registers r_1 and r_2, respectively, i.e., tv_1 = lookup(r_1, rf, rob), tv_2 = lookup(r_2, rf, rob).

The instruction fetch rules are summarized Figure 5. In any implementation, there are a finite number of *rob* entries, and the instruction fetch has to be stalled if *rob* is full. This availability checking can be easily modeled, and we leave it as a simple exercise for the interested reader. It should be noted that a fast implementation of the lookup procedure in hardware is quite difficult. Often a renaming table that keeps the association between a register name and its current tag is maintained separately.

5.2 Arithmetic Operation and Value Propagation Rules

The arithmetic operation rule states that an arithmetic operation in the *rob* can be performed if both operands are available. It assigns the result to the corresponding tag. Note that the instruction can be in any position in the *rob*.

Op Rule

$\quad$ Proc(ia, rf, $rob_1 \oplus$Itb(ia_1,t:=Op(v_1,v_2),Wr(r))$\oplus rob_2$, btb, im)

$\rightarrow$ Proc(ia, rf, $rob_1 \oplus$Itb(ia_1,t:=v,Wr(r))$\oplus rob_2$, btb, im) $\quad$ where $\quad v = \underline{\mathrm{Op}}(v_1,v_2)$

There are two value propagation rules, the forward rule and the commit rule. The forward rule sends the value of a tag to other instruction templates, while the commit rule writes the value produced by the oldest instruction in the *rob* to the destination register and retires the corresponding renaming tag. Notation $rob_2[v/t]$ means that one or more appearances of tag t in rob_2 are replaced by value v.

Value-Forward Rule

$\quad$ Proc(ia, rf, $rob_1 \oplus$Itb(ia_1,t:=v,Wr(r))$\oplus rob_2$, btb, im) $\quad$ if $\quad t \in rob_2$

$\rightarrow$ Proc(ia, rf, $rob_1 \oplus$Itb(ia_1,t:=v,Wr(r))$\oplus rob_2[v/t]$, btb, im)

Value-Commit Rule

$\quad$ Proc(ia, rf, Itb(ia_1,t:=v,Wr(r))$\oplus rob$, btb, im) $\quad$ if $\quad t \notin rob$

$\rightarrow$ Proc(ia, $rf[r:=v]$, rob, btb, im)

Rule Name	**rob** $= rob_1 \oplus \text{Itb}(ia_1, \text{Jz}(0, nia), \text{Sp}(pia)) \oplus rob_2$	**Next rob**	**Next pc**
Jump-CorrectSpec	$pia = nia$	$rob_1 \oplus rob_2$	ia
Jump-WrongSpec	$pia \neq nia$	rob_1	nia

Rule Name	**rob** $= rob_1 \oplus \text{Itb}(ia_1, \text{Jz}(v, \text{-}), \text{Sp}(pia)) \oplus rob_2$	**Next rob**	**Next pc**
NoJump-CorrectSpec	$v \neq 0,\ pia = ia_1{+}1$	$rob_1 \oplus rob_2$	ia
NoJump-WrongSpec	$v \neq 0,\ pia \neq ia_1{+}1$	rob_1	$ia_1{+}1$

Figure 6: P_S Branch Completion Rules (Current state: Proc(*ia*, *rf*, *rob*, *btb*, *im*). *btb* update is not shown)

It is worth noting that the *rob* pattern in the commit rule dictates that the register file can only be modified by the oldest instruction after it has forwarded the value to all the buffers in the *rob* that reference its tag. Restricting the register update to just the oldest instruction in the *rob* eliminates output (write-after-write) hazards, and protects the register file from being polluted by incorrect speculative instructions. It also provides a way to support precise interrupts. The commit rule is needed primarily to free up resources and to allow the reuse of the tag by the following instructions.

5.3 Branch Completion Rules

The branch completion rules determine if the branch prediction was correct by comparing the speculated instruction address and the resolved branch target instruction address. If they do not match (indicating that the speculation was wrong), all instructions issued after the branch instruction are aborted, and the program counter is set to the new branch target instruction. The branch target buffer *btb* is updated according to some prediction algorithm. The branch resolution cases are summarized in Figure 6.

The reader may want to ponder over the fact that the branch rules allow branches to be resolved in any order. The branch resolution mechanism becomes slightly complicated, if certain instructions needed to be killed are waiting for responses from the memory system or some functional units. In such a situation, killing may have to be postponed until no instruction in rob_2 is waiting for a response. (This is not possible for the rules that we have presented).

5.4 Memory Access Rules

Memory requests are sent to the memory system strictly in order and only when there is no unresolved branch instruction in front of it. This dispatch rules flip the U bit to D, and enqueue the memory request into the *pmb*. The memory system can respond to the requests in any order and the response is used to update the appropriate entry in the *rob*. The ';' is used to represent an ordered queue, and the '|' an unordered queue (i.e., it is both commutative and associative). The memory access rules are given in Figure 7.

We do not present the rules for how the memory system handles memory requests from the *pmb*. The table in Figure 8 shows a simple interface between the processor and the memory that ensures memory accesses are processed in order by the external memory system to guarantee sequential consistency in multiprocessor systems. More aggressive implementations of memory access operations are possible than the ones presented here, but they lead to various relaxed memory models in multiprocessor systems. A discussion of such optimizations is beyond the scope of this paper.

Rule Name	itb	pmb	Next itb	Next pmb
Load-Dispatch	Itb(ia_1,t:=Load(a,U),Wr(r)) U, Jz $\notin rob_1$	*pmb*	Itb(ia_1,t:=Load(a,D),Wr(r))	pmb;$\langle t$,Load(a)$\rangle$
Store-Dispatch	Itb(ia_1,t:=Store(a,v,U)) U, Jz $\notin rob_1$	*pmb*	Itb(ia_1,t:=Store(a,v,D))	pmb;$\langle t$,Store(a,v)$\rangle$

Rule Name	itb	mpb	Next itb	Next mpb
Load-Retire	Itb(ia_1,t:=Load(a,D),Wr(r))	$\langle t,v\rangle \| mpb$	Itb(ia_1,t:=v,Wr(r))	*mpb*
Store-Retire	Itb(ia_1,t:=Store(a,v,D))	$\langle t$,Ack$\rangle \| mpb$	ϵ (deleted)	*mpb*

Figure 7: $\mathrm{P_S}$ Memory Access Rules (Current state: Sys(Proc(ia, rf, $rob_1 \oplus itb \oplus rob_2$, btb, im), pmb, mpb))

dm	pmb	mpb	Next dm	Next pmb	Next mpb
dm	$\langle t$,Load(a)$\rangle$;pmb	*mpb*	*dm*	*pmb*	$mpb \| \langle t, dm[a]\rangle$
dm	$\langle t$,Store(a,v)$\rangle$;pmb	*mpb*	$dm[a{:=}v]$	*pmb*	$mpb \| \langle t$,Ack$\rangle$

Figure 8: Processor-Memory Interface Specification

6 Correctness of the $\mathrm{P_S}$ Model

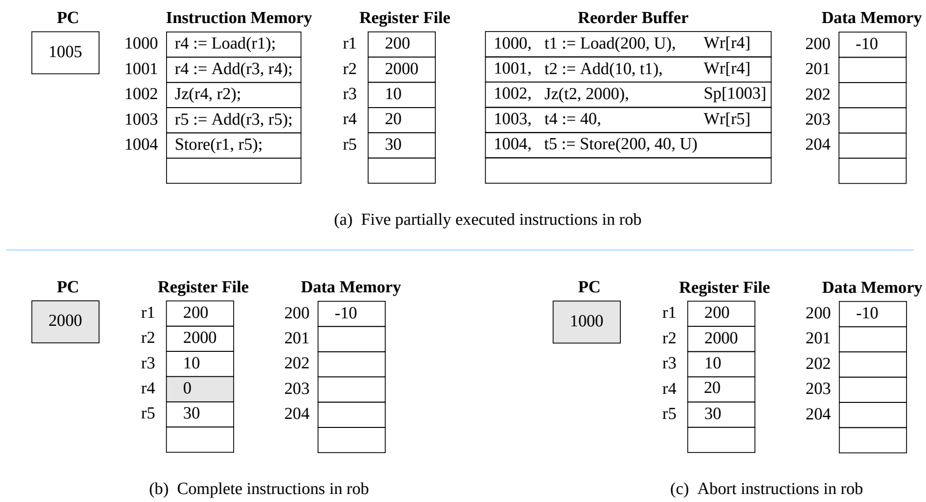
One way to prove that the speculative processor is a correct implementation of the AX instruction set is to show that $\mathrm{P_B}$ and $\mathrm{P_S}$ can simulate each other in regards to some observable property. A natural observation function is the one that can extract all the programmer visible state, including the program counter, the register file and the memory from the system. One can think of an observation function in terms of a print instruction that prints a part or the whole of the programmer visible state. If model A can simulate model B, then for any program, model A should be able to print whatever model B prints during the execution.

The programmer visible state of $\mathrm{P_B}$ is obvious – it is the whole term. The $\mathrm{P_B}$ model does not have any hidden state. It is a bit tricky to extract the corresponding values of *pc*, *rf* and *dm* from the $\mathrm{P_S}$ model because of the partially or speculatively executed instructions. However, if we consider only those $\mathrm{P_S}$ states where the *rob*, *pmb* and *mpb* are empty then it is straightforward to find the corresponding $\mathrm{P_B}$ state. We will call such states of $\mathrm{P_S}$ as the *drained states*.

It is easy to show that $\mathrm{P_S}$ can simulate each rule of $\mathrm{P_B}$. Given a $\mathrm{P_B}$ term s_1, a $\mathrm{P_S}$ term t_1 is created such that it has the same values of *pc*, *rf*, *im* and *dm*, and its *rob*, *pmb* and *mpb* are all empty. Now, if s_1 can be rewritten to s_2 according to some $\mathrm{P_B}$ rule then we can apply a sequence of $\mathrm{P_S}$ rules to t_1 to obtain t_2 such that t_2 is in a drained state and has the same programmer visible state as s_2. In this manner, $\mathrm{P_S}$ can simulate each move of $\mathrm{P_B}$.

The simulation in the other direction is tricky because we need to find a $\mathrm{P_B}$ term corresponding to each term (not just the terms in the drained state) of $\mathrm{P_S}$. We somehow need to extract the programmer visible state from any $\mathrm{P_S}$ term. There are several ways in which a $\mathrm{P_S}$ term can be driven to a drained state using the $\mathrm{P_S}$ rules, and each way may lead to a different drained state. We illustrate this via an example.

Consider the snapshot shown in Figure 9(a) (we have not shown *pmb* and *mpb* and let us assume both are empty). There are at least two ways to drive this term into a drained state. One way is to stop fetching instructions and complete all the partially executed instructions. This process can be thought of as applying a subset of the $\mathrm{P_S}$ rules (i.e., all the rules except the instruction fetch



(a) Five partially executed instructions in rob

(b) Complete instructions in rob

(c) Abort instructions in rob

Figure 9: Draining the Processor

rules) to the term. After repeated application of such rules the *rob* should become empty, and the system should reach a drained state. Such a situation is shown in Figure 9(b), where in the process of draining the pipeline, it is discovered that the branch speculation was wrong. An alternative way is to rollback the execution by killing all the partially executed instructions and restoring the *pc* to the address of the oldest killed instruction. The drained state obtained in this manner is shown in Figure 9(c). Notice, this drained state is different from the one obtained by completing the partially executed instructions.

It is worth pointing out that the two draining methods represent two extremes. By carefully selecting the rules that are applied to reach the drained state, we can allow certain instructions in the *rob* to be completed and the rest to be killed. Regardless of which method is chosen for draining, we have to show that the draining method itself is correct. This is trivial when no new rules are introduced for draining. Otherwise, one will have to prove that, for example, the rollback rule does not take the system into an illegal state.

The simulation of $\mathrm{P_S}$ by $\mathrm{P_B}$ is shown in Figure 10, where '$\twoheadrightarrow$' represents zero or more rewriting steps. We have proven the following theorem using standard techniques of TRS's elsewhere [8]. Several subtle errors were discovered while proving this simulation theorem.

Theorem 1 **($\mathrm{P_B}$ simulates $\mathrm{P_S}$)** *Suppose $t_1 \twoheadrightarrow t_2$ and $t_1 \twoheadrightarrow t_{\mathrm{d1}}$ in $\mathrm{P_S}$ where t_{d1} is a drained state, and s_1 is the $\mathrm{P_B}$ term corresponding to t_{d1}. Then there exists a reduction $t_2 \twoheadrightarrow t_{\mathrm{d2}}$ in $\mathrm{P_S}$ such that t_{d2} is a drained state, and $s_1 \twoheadrightarrow s_2$ in $\mathrm{P_B}$ where s_2 is the $\mathrm{P_B}$ state corresponding to t_{d2}.*

7 Conclusions

TRS's provide a natural way to describe microprocessors and memory systems because in such systems several actions may take place asynchronously. Such systems are not amenable to sequential descriptions because sequentiality either causes over specification or does not allow one to consider situations that may arise in a real implementation. Using proper abstractions, the TRS descriptions can be given in a highly modular fashion. For example, elsewhere we have defined a new memory

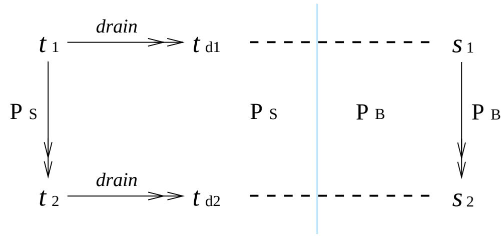


Figure 10: Simulate $\mathrm{P_S}$ in $\mathrm{P_B}$

model and associated cache-coherence protocols [10, 9]; these can be incorporated in the speculative processor model simply by replacing the processor-memory interface rules given at the end of Section 5.4. Similarly, more rules can be given to describe fully pipelined versions of both the micro-architectures described in this paper.

It is worth emphasizing that the formulation for the correctness using drained states is quite general. For example, the states of a system with caches can be compared to a system without caches using the idea of "cache flushing" to show the correctness of cache coherence protocols. The idea of a rewriting sequence that can take a system into a drained state has an intuitive appeal for designers. When a system has a large number of rules, the correctness proofs can quickly become tedious. The use of theorem provers (e.g., PVS) and model checkers (e.g., Murphi) can alleviate this problem; we are exploring the use of such tools in our verification effort.

Finally, we are also developing a compiler for hardware synthesis from TRS's. It translates TRS's into a standard hardware description language like Verilog [5]. We restrict the generated Verilog to be structural, so that commercial tools can be used to go all the way down to gates and layout. The grammar of the terms, when augmented with details like instruction formats and sizes of various register files, buffers, memories etc, precisely specifies the state elements. Each rule is then compiled such that the state is read in the beginning of the clock cycle and updated at the end of the clock cycle. This single-cycle implementation methodology automatically enforces the atomicity constraint of each rule. All the enabled rules fire in parallel unless some modify the same element of the state. In case of such a conflict, based on some policy, one of the conflicting rules is selected to fire.

There are several challenging problems in the synthesis area. First, good scheduling in the presence of resource constraints can be quite difficult. For example, rules dictate the number of concurrent ports a register file needs for single-cycle synthesis. If the register file provided has fewer ports, a rule may take several cycles to implement. Naive implementation of this idea can lead to implementations with poor performance. Second, one is often interested in synthesizing only a part of the system described by a set of rules. For example, while synthesizing a microprocessor from the speculative processor rules, one may want to ignore the memory system and instead produce an interface specification for the external memory. A general solution to these problems is under current study. Nevertheless, we are able to compile many TRS descriptions into Verilog today, and have already tested a few examples by generating FPGA code from the Verilog produced by our compiler.

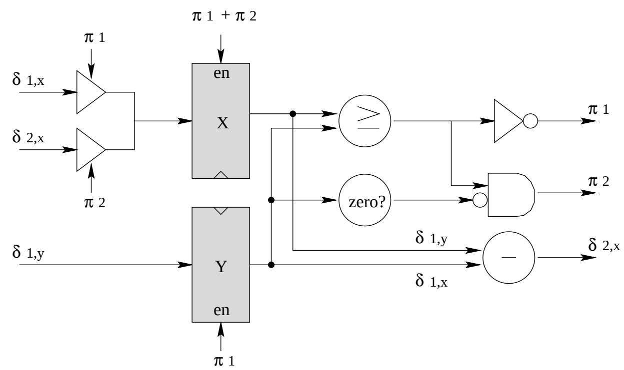


Figure 11: The GCD Circuit

Hardware Synthesis of GCD
James C. Hoe, MIT

Euclides algorithm for computing the greatest common divisor of two numbers can be expressed as follows in TRS notation:

$$\begin{array}{llll} \mathrm{GCD}(x,y) & \text{if } x < y & & \rightarrow \mathrm{GCD}(y,x) \\ \mathrm{GCD}(x,y) & \text{if } x \geq y & \text{and } y \neq 0 & \rightarrow \mathrm{GCD}(x\text{-}y,y) \end{array}$$

TRAC, Term Rewriting Architectural Compiler [5], generates a Verilog description for the circuit shown in Figure 11. The δ wires represent the new state values while the π wires represent the firing condition of the corresponding rules. After synthesis by the latest Xilinx's tools, the circuit with 32-bit x and y registers runs at 40.1 MHz using 24% of a XC4010XL-0.9 FPGA. For reference, a hand-tuned RTL code written by Daniel L. Rosenband resulted in 53 MHz and 16% utilization in the same technology.

Source-to-source transformations of TRS's also help in high-level synthesis. For example, if the generated circuit does not meet the clock requirement, then the offending rules in the TRS have to be split into simpler rules. We have systematically transformed the non-pipelined architecture represented by $\mathrm{P_B}$ into a simple 5-stage pipeline, and then further transformed the TRS obtained this way into a TRS representing a 2-way superscalar architecture. Such source-to-source transformations dramatically reduce the number of rules a designer has to write.

The promise of TRS's for computer architecture is the development of a set of integrated design tools for modeling, specification, verification, simulation and synthesis. The conciseness and preciseness of TRS's coupled with good tools may radically alter the teaching of computer architecture in future.

Acknowledgement We would like to thank James Hoe, Lisa Poyneer and Larry Rudolph for numerous discussions. 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] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

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

[3] Byron Cook, John Launchbury, and John Matthews. Specifying Superscalar Microprocessors in Hawk. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[5] James C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. CSG Memo 421, Laboratory for Computer Science, MIT, 1999.

[6] Jan Willem 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.

[7] K. L. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.

[8] 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.

[9] Xiaowei Shen, Arvind, and Larry Rodolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 13th ACM International Conference on Supercomputing, Rhodes, Greece*, June 1999.

[10] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium On Computer Architecture, Atlanta, Georgia*, May 1999.

[11] Phillip J. Windley. Formal Modeling and Verification of Microprocessors. *IEEE Transactions on Computers*, 44(1), January 1995.