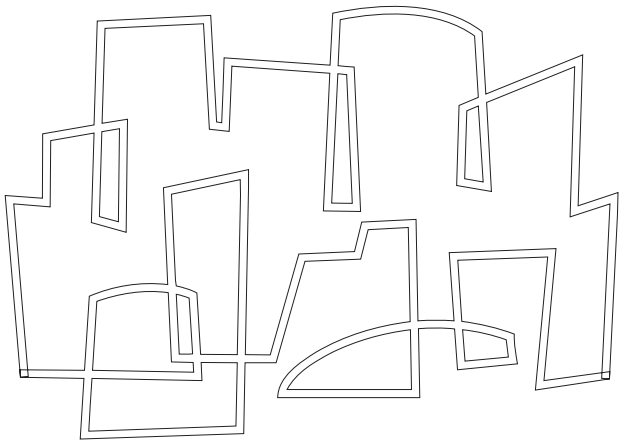
Using Term Rewriting Systems to Design and Verify Processors

Arvind, Xiaowei Shen

In IEEE Micro Special Issue on Modeling and Validation of Microprocessors

1999, May

Computation Structures Group Memo 419



Using Term Rewriting Systems to Design and Verify Processors

THE OPERATIONAL SEMANTICS OF A SIMPLE RISC INSTRUCTION SET SERVE AS AN ILLUSTRATION IN THIS NOVEL USE OF TERM REWRITING SYSTEMS TO DESCRIBE MICROARCHITECTURES.

Arvind and Xiaowei Shen
Massachusetts Institute of Technology

Term rewriting systems (TRSs) offer a convenient way to describe parallel and asynchronous systems and prove an implementation's correctness with respect to a specification. TRS descriptions, augmented with proper information about the system building blocks, also hold the promise of high-level synthesis. High-level architectural descriptions that are both automatically synthesizable and verifiable would permit architectural exploration at a fraction of the time and cost required by current commercial tools.

In recent years, considerable attention has focused on formal verification of microprocessors.[1-4] Other formal techniques, such as Lamport's Temporal Logic of Actions and Lynch's I/O automata, also enable us to model microprocessors. While all these techniques have something in common with TRSs, we find the use of TRSs more intuitive in both architecture descriptions and correctness proofs. TRSs can describe both deterministic and nondeterministic computations. Although they have been used extensively in programming language research to give operational semantics, their use in architectural descriptions is novel.

In this article, we use TRSs to describe a speculative processor capable of register renaming and out-of-order execution. We lack space to discuss a synthesis procedure from TRSs or to provide the details needed to make automatic synthesis feasible. Nevertheless, we show that our speculative processor produces the same set of behaviors as a simple nonpipelined implementation. Our descriptions of microarchitectures are more precise than those found in modern textbooks.[5] The clarity of these descriptions lets us study the impact of features such as write buffers or caches, especially in multiprocessor systems.[6,7] In fact, experience in teaching computer architectures partially motivated this work.

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 ($S_0 \subseteq S$). The state of a system is represented as a TRS term, while the state transitions are represented as TRS rules. The general structure of rewriting rules is

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

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

We can use a rule to rewrite a term if the rule's left-hand-side pattern matches the term or one of its subterms and the corresponding

0272-1732/99/$10.00 © 1999 IEEE

predicate is true. The new term is generated in accordance with the rule's right-hand side. If several rules apply, then any one of them can be applied. If no rule applies, 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. The sidebar at right shows an example of a well-known TRS. The literature offers more information about TRSs.[8,9]

The AX instruction set

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

In the following AX instruction set, all arithmetic operations are performed on registers, and only the Load and Store instructions are allowed to access memory.

INST ≡

	r := Loadc(v)	Load constant
❙	r := Loadpc	Load program counter
❙	r := Op(r_1, r_2)	Arithmetic operation
❙	Jz(r_1, r_2)	Branch
❙	r := Load(r_1)	Load memory
❙	Store(r_1, r_2)	Store memory

The grammar uses a thick vertical bar (❙) as a metanotation to separate disjuncts. Throughout the article, 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 microarchitecture but must be provided for synthesis.

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

The load-constant instruction r := Loadc(v) puts value v into register r. The load-program-counter instruction r := Loadpc puts the program counter's content 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 incremented 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 P_B model, a single-cycle, nonpipelined, in-order execution processor. Figure 1 (next page) shows the data path for such a system. 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 rewrit-

SK combinators: a TRS example

The SK combinatory system, which has only two rules and a simple grammar for generating terms, provides a small but fascinating example of term rewriting. The two rules are sufficient to describe any computable function.

Term ≡ K ❙ S ❙ Term.Term

K-rule: (K.x).y → x
S-rule: ((S.x).y).z → (x.z).(y.z)

We can 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. We can rewrite this term further to x by applying the K-rule. Thus, if we read the dot as a function application, then the term ((S.K).K) behaves as the identity function.

Note that the S-rule rearranges the dot and duplicates the term represented by x on the right-hand side. For architectures in which terms represent states, rules must be restricted so that terms are not restructured or duplicated as in the S and K rules.

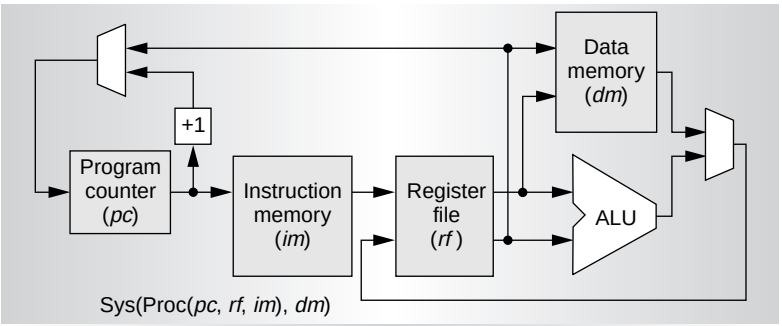


Figure 1. The P_B model: a single-cycle, in-order processor.

Loadc rule
Proc(*ia*, *rf*, *im*) if $im[ia] = r := \text{Loadc}(v)$
$\rightarrow$ Proc(ia+1, $rf[r := v]$, *im*)

Loadpc rule
Proc(*ia*, *rf*, *im*) if $im[ia] = r := \text{Loadpc}$
$\rightarrow$ Proc(ia+1, $rf[r := ia]$, *im*)

Op rule
Proc(*ia*, *rf*, *im*) if $im[ia] = r := \text{Op}(r_1, r_2)$
$\rightarrow$ Proc(ia+1, $rf[r := v]$, *im*) where $v = \underline{\text{Op}}(rf[r_1], rf[r_2])$

Jz-Jump rule
Proc(*ia*, *rf*, *im*) if $im[ia] = \text{Jz}(r_1, r_2)$ and $rf[r_1] = 0$
$\rightarrow$ Proc($rf[r_2]$, *rf*, *im*)

Jz-NoJump rule
Proc(*ia*, *rf*, *im*) if $im[ia] = \text{Jz}(r_1, r_2)$ and $rf[r_1] \neq 0$
$\rightarrow$ Proc(ia+1, *rf*, *im*)

Load rule
Sys(Proc(*ia*, *rf*, *im*), *dm*) if $im[ia] = r := \text{Load}(r_1)$
$\rightarrow$ Sys(Proc(ia+1, $rf[r := dm[a]]$, *im*), *dm*) where $a = rf[r_1]$

Store rule
Sys(Proc(*ia*, *rf*, *im*), *dm*) if $im[ia] = \text{Store}(r_1, r_2)$
$\rightarrow$ Sys(Proc(ia+1, *rf*, *im*), $dm[a := rf[r_2]]$) where $a = rf[r_1]$

Figure 2. These rules for the P_B model define the operational semantics of the AX instruction set.

ing rule specifying how the state is modified after each instruction executes.

Note 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 special symbols, such as :=, and all identifiers that start with capital letters are treated as constants in pattern matching. We use a hyphen (-) to represent the wildcard term that can match any term. Notation $\underline{\text{Op}}(v_1, v_2)$ represents the result of operation Op with operands v_1 and v_2. The rules for the P_B model are given in Figure 2 and define the operational semantics of the AX instruction set.

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 based on the condition register's value.

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. This is why these rules describe a single-cycle, nonpipelined implementation of AX.

To save space, we use tables to describe the rules informally. For example, Table 1 summarizes the P_B rules given in Figure 2. Given proper context, it should be easy to deduce the precise TRS rules from a tabular description.

Register renaming and speculative execution

Many possible microarchitectures can implement the AX instruction set. For example, in a simple pipelined architecture, instructions are fetched, executed, and retired in order, and the processor can contain as many as four or five partially executed instructions. Storage in the form of pipeline buffers holds these partially executed instructions. More sophisticated pipelined architectures have multiple functional units that can be specialized for integer or floating-point calculations. In such architectures, instructions issued in order may nevertheless complete out of order because of varying functional-unit latencies. An implementation preserves correctness by ensuring

Table 1. Operational semantics of AX (current state: Sys(Proc(*ia, rf, im*), *dm*)).

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 := Op(rf [r_1], rf [r_2])]	dm
Jz	Jz(r_1, r_2)	ia+1 (if rf [r_1] ≠ 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]]

that 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. Cray's CDC 6600, 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 registers in the instruction set limits the number of instructions in the pipeline.

In the mid-sixties, Robert Tomasulo at IBM invented the technique of register renaming to overcome this limitation on pipelining. He 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 and could be used again once the instruction was completed. The microarchitecture maintained the association between the register name, the tag, and the associated value (whenever the value became available). This innovative idea was embodied in the IBM 360/91 in the late sixties but went out of favor until the late eighties for several reasons. One is that performance gains were not considered commensurate with the implementation complexity. The complexity issue became less relevant as the register renaming technique became better understood and extra transistors became available on chips. Another problem was that Tomasulo's specific technique for register renaming resulted in a machine with imprecise interrupts. This problem was later solved by introducing speculative execution in architectures. By the mid-nineties, register renaming had become commonplace and is now present in all high-end microprocessors.

An important state element in a microarchitecture with register renaming is a reorder buffer (ROB), which holds instructions that have been decoded but have not completed execution (see Figure 3). Conceptually, a ROB divides the processor into two asynchronous

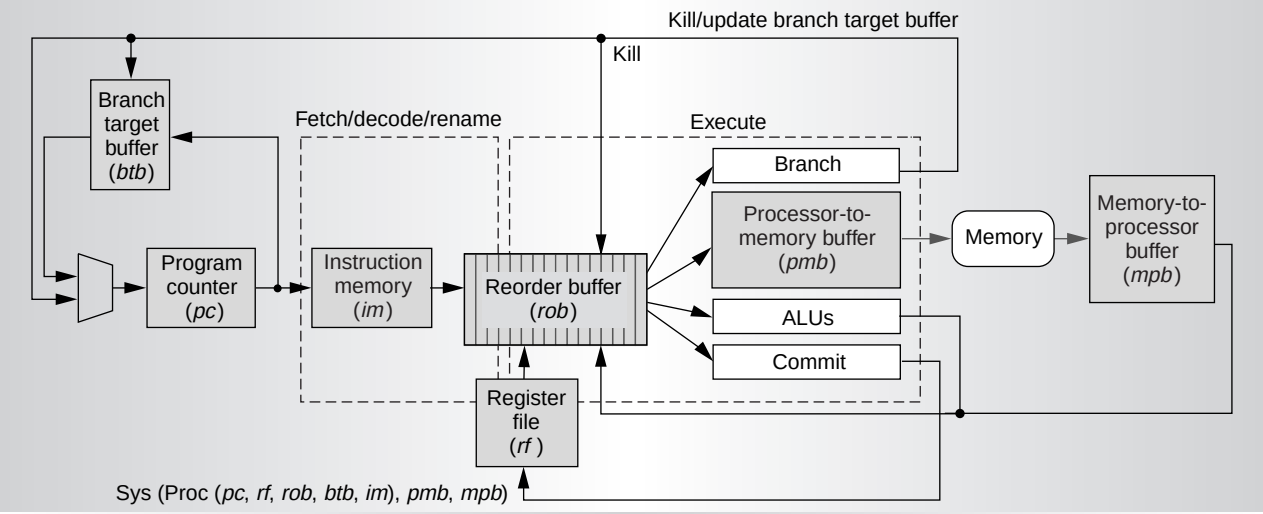


Figure 3. The P_S model: a processor with register renaming and speculative execution.

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 as a 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 of the processor 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 execution of instructions. On the basis of the program's past behavior, the speculative mechanisms predict the address of the next instruction to be issued. (Several researchers have recently suggested mechanisms to speculate on memory values as well, but none have been implemented so far; we do not consider them in this article.) The speculative instruction's address is determined by consulting a table known as the branch target buffer (BTB). The BTB can be indexed by 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 is nullified. The BTB is updated according to some prediction scheme after each branch resolution.

The speculative processor's correctness is not contingent on 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 profoundly influence performance. Generally, we assume that the BTB produces the correct next instruction address for all nonbranch instructions. We needn't discuss the BTB any further because the branch prediction strategy is completely orthogonal to the mechanisms for speculative execution.

Any processor permitting speculative execution must ensure that a speculative instruction does not modify the programmer-visible state until it can be "committed." Alternatively, it must 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 combination 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 they may update the program counter. Both the current and the speculated instruction address are recorded. Thus, speculation correctness can be determined later, and the correct program counter can be restored in case of a wrong prediction. Typically, all the temporary state is maintained in the ROB itself.

The P_S speculative processor model

We now present the rules for a simplified microarchitecture that performs register renaming and speculative execution. We achieve this simplification 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*). We do not discuss the exact memory system organization. However, memory system details can be added in a modular fashion without changing the processor description presented here.[6,7]

We need to add two new components to the processor state: *rob* and *btb*, corresponding to ROB and BTB. The reorder buffer 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 that is initially empty (ε). We use 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 with an array index serving as 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 buffer (*itb*) 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 destination register r. For branch instructions, the Sp(*pia*) field holds the predicted instruction address *pia*, which will be used to determine the prediction's correctness. 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:

ROB entry ≡
  Itb(*ia*, *t* := *v*, Wr(*r*))
  ❙ Itb(*ia*, *t* := Op(tv_1, tv_2), Wr(*r*))
  ❙ Itb(*ia*, Jz(tv_1, tv_2), Sp(*pia*))
  ❙ Itb(*ia*, *t* := Load(tv_1, *mf*), Wr(*r*))
  ❙ Itb(*ia*, *t* := Store(tv_1, tv_2, *mf*))

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 any register updating.

Table 2. P_s instruction fetch rules (current state: Proc(*ia, rf, rob, btb, im*)).

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

Instruction fetch rules

Each time the processor issues an instruction, the program counter is set to the address of the next instruction to be issued. For nonbranch instructions, the program counter is simply incremented by one. Speculative execution occurs 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 Jz instruction's address.

When the processor issues an instruction, an instruction template is allocated in the *rob*. If the instruction is to modify a register, we use an unused renaming tag (typically the index of the *rob* slot) 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:

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

It is beyond the scope of this article to give a hardware implementation of this procedure, but it is certainly possible to do so using TRSs. Any implementation that can look up values in the *rob* using a combinational circuit will suffice.

As an example of instruction fetch rules, consider the Fetch-Op rule, which fetches an Op instruction and after register renaming simply puts it at the end of the *rob* as follows:

Fetch-Op rule
  Proc(*ia*, *rf*, *rob*, *btb*, *im*)
    if *im*[*ia*] = *r* := Op(r_1, r_2)
→ Proc(*ia*+1, *rf*,
  rob ⊕ Itb(*ia*, *t* := Op(tv_1, tv_2), Wr(*r*)),
  btb, *im*)

where *t* represents an unused tag, and tv_1 and tv_2 represent the tag or value corresponding to the operand registers r_1 and r_2, respectively; that is, tv_1 = lookup(r_1, *rf*, *rob*), and tv_2 = lookup(r_2, *rf*, *rob*). Table 2 summarizes the instruction fetch rules.

Any implementation includes a finite number of *rob* entries, and the instruction fetch must be stalled if *rob* is full. This availability checking can be easily modeled and makes a simple exercise for those interested. A fast implementation of the lookup procedure in hardware is quite difficult. Often a renaming table that retains the association between a register name and its current tag is maintained separately.

Table 3. P_S arithmetic operation and value propagation rules (current state: Proc(*ia, rf, rob, btb, im*)).

Rule name	rob	Next rob	Next rf
Op	$rob_1 \oplus \text{Itb}(ia_1, t := \text{Op}(v_1, v_2), \text{Wr}(r)) \oplus rob_2$	$rob_1 \oplus \text{Itb}(ia_1, t := \underline{\text{Op}}(v_1, v_2), \text{Wr}(r)) \oplus rob_2$	*rf*
Value-Forward	$rob_1 \oplus \text{Itb}(ia_1, t := v, \text{Wr}(r)) \oplus rob_2$ (if $t \in rob_2$)	$rob_1 \oplus \text{Itb}(ia_1, t := v, \text{Wr}(r)) \oplus rob_2[v/t]$	*rf*
Value-Commit	$\text{Itb}(ia_1, t := v, \text{Wr}(r)) \oplus rob$ (if $t \notin rob$)	*rob*	$rf[r := v]$

Table 4. P_S branch completion rules (current state: Proc(*ia, rf, rob, btb, im*). The *btb* update is not shown.

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	if $pia = nia$	$rob_1 \oplus rob_2$	*ia*
Jump-WrongSpec	if $pia \neq nia$	rob_1	*nia*

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

Arithmetic operation and value propagation rules

Table 3 gives the rules for arithmetic operation and value propagation. 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*. The forward rule sends a tag's value 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.

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 to free up resources and to let the following instructions reuse the tag.

Branch completion rules

The branch completion rules determine whether the branch prediction was correct by comparing the predicted instruction address (*pia*) with the resolved branch target instruction address (*nia* or ia_1+1). If they don't match (meaning 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 *btb* is updated according to some prediction algorithm. Table 4 summarizes the branch resolution cases. It is worth noting that the branch rules allow branches to be resolved in any order.

The branch resolution mechanism becomes slightly complicated if certain instructions that need 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 rob_2 does not contain an instruction waiting for a response. (This is not possible for the rules we have presented.)

Memory access rules

Memory requests are sent to the memory system strictly in order. A request is sent only when there is no unresolved branch instruction in front of it. The dispatch rules flip the U bit to D and enqueues 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*. Table 5 gives the memory access rules. The semicolon represents an ordered queue and the vertical bar (|) an unordered queue (that is, the vertical bar connective is both commutative and associative).

We do not present the rules for how the memory system handles memory requests from the *pmb*. Table 6 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

Table 5. P_S memory access rules (current state: Sys(Proc(*ia, rf, rob*$_1$ ⊕ *itb* ⊕ *rob*$_2$, *btb, im*), *pmb, mpb*)).

Rule name	itb	pmb	Next itb	Next pmb
Load-Dispatch	Itb(ia_1, t := Load(a, U), Wr(r)) (if U, Jz ∉ rob_1)	*pmb*	Itb(ia_1, t := Load(a, D), Wr(r))	*pmb*; ⟨t, Load(a)⟩
Store-Dispatch	Itb(ia_1, t := Store(a, v, U)) (if U, Jz ∉ rob_1)	*pmb*	Itb(ia_1, t := Store(a, v, D))	*pmb*; ⟨t, Store(a, v)⟩

Rule name	itb	mpb	Next itb	Next mpb
Load-Retire	Itb(ia_1, t := Load(a, D), Wr(r))	⟨t, v⟩ | *mpb*	Itb(ia_1, t := v, Wr(r))	*mpb*
Store-Retire	Itb(ia_1, t := Store(a, v, D))	⟨t, Ack⟩ | *mpb*	ε (deleted)	*mpb*

memory access operations are possible, but they often lead to various relaxed memory models in multiprocessor systems. Discussing such optimizations is beyond the scope of this article.

The correctness of the P_S model

One way to prove that the speculative processor is a correct implementation of the AX instruction set is to show that P_B and P_S can simulate each other in regard to some observable property. A natural observation function is one that can extract all the programmer-visible state, including the program counter, the register file, and the memory, from the system. We can think of an observation function in terms of a print instruction that prints part or all 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 execution.

The programmer-visible state of P_B is obvious—it is the whole term. The 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 P_S model because of the partially or speculatively executed instructions. However, if we consider only those P_S states in which the *rob*, *pmb*, and *mpb* are empty, then it is straightforward to find the corresponding P_B state. We will call such states of P_S the *drained states*.

It is easy to show that P_S can simulate each rule of P_B. Given a P_B term s_1, a 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 P_B rule, we can apply a sequence of 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, P_S can simulate each move of P_B.

Table 6. Processor-memory interface specification.

dm	pmb	mpb	Next dm	Next pmb	Next mpb
dm	⟨t, Load(a)⟩; *pmb*	*mbp*	*dm*	*pmb*	*mpb* | ⟨t, *dm*[a]⟩
dm	⟨t, Store(a, v)⟩; *pmb*	*mpb*	*dm*[a := v]	*pmb*	*mpb* | ⟨t, Ack⟩

Simulation in the other direction is tricky because we need to find a P_B term corresponding to each term of P_S (not just the terms in the drained state). We need to somehow extract the programmer-visible state from any P_S term. There are several ways to drive a P_S term to a drained state using the P_S rules, and each way may lead to a different drained state.

As an example, consider the snapshot shown in Figure 4a (we have not shown *pmb* and *mpb*; let's assume both are empty). There are at least two ways to drive this term into a drained state. One is to stop fetching instructions and complete all the partially executed instructions. This process can be thought of as applying a subset of the P_S rules (all the rules except the instruction fetch rules) to the term. After repeated application of such rules, the *rob* should become empty and the system should reach a drained state. Figure 4b shows such a situation, where in the process of draining the pipeline, we discover that the branch speculation was wrong. An alternative way is to roll back the execution by killing all the partially executed instructions and restoring the *pc* to the address of the oldest killed instruction. Figure 4c shows the drained state obtained in this manner. Note that this drained state is different from the one obtained by completing the partially executed instructions.

The two draining methods represent two extremes. By carefully selecting the rules 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 draining method is chosen, we must

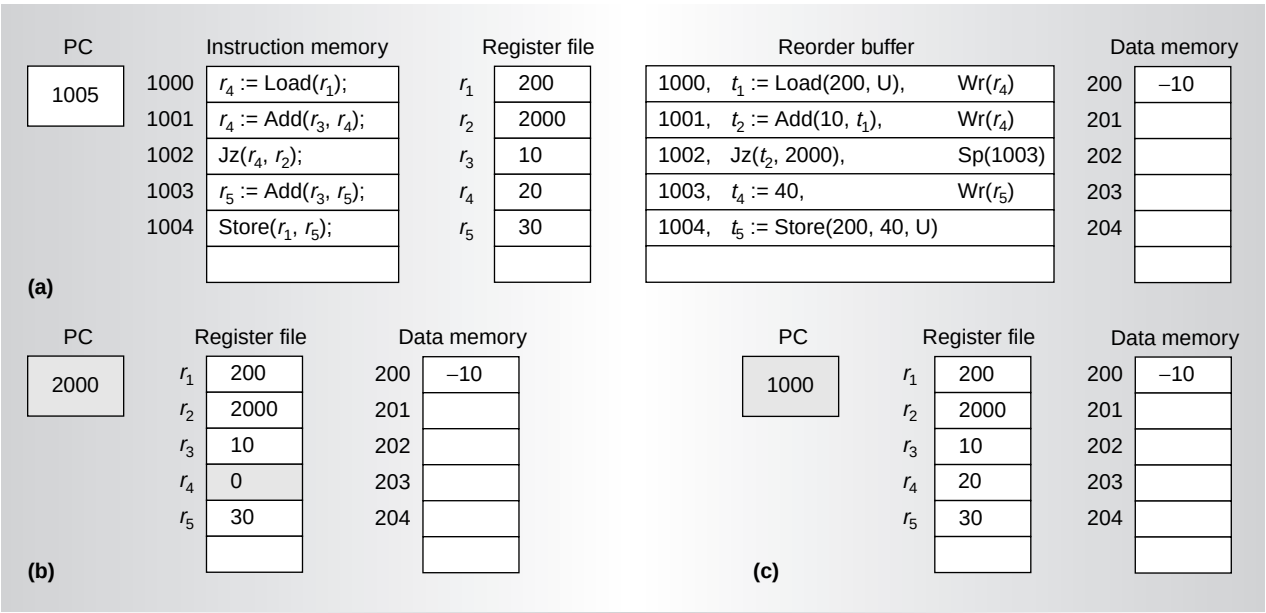


Figure 4. The processor state with five partially executed instructions (a); the drained processor state after completing (b) or aborting (c) all the partially executed instructions.

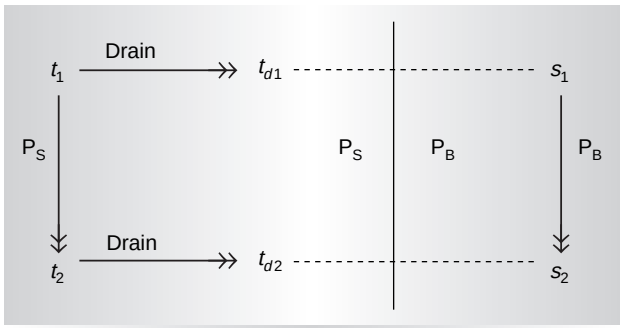


Figure 5. Simulating P_S in P_B.

show that the draining method itself is correct. This is trivial when no new rules are introduced for draining. Otherwise, we have to prove, for example, that the rollback rule does not take the system into an illegal state.

Figure 5 shows the simulation of P_S by P_B, where $\twoheadrightarrow$ represents zero or more rewriting steps. Elsewhere, we have proved the following theorem using standard TRS techniques.[10] We discovered several subtle errors while proving this simulation theorem.

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

The formulation of the correctness using drained states is quite general. For example, the states of a system with caches can be compared to those of 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 many rules, the correctness proofs can quickly become tedious. Use of theorem provers, such as PVS, and model checkers, such as Murphi, can alleviate this problem; we are exploring the use of such tools in our verification effort.

In microprocessors and memory systems, several actions may occur asynchronously. These systems are not amenable to sequential descriptions because sequentiality either causes overspecification or does not allow for consideration of situations that may arise in a real implementation. Term rewriting systems provide a natural way to describe such systems.

Using proper abstractions, we can create TRS descriptions in a highly modular fashion. For example, elsewhere we have defined a new memory model and associated cache-coherence protocols[6,7] that can be incorporated in the speculative processor model simply by replac-

ing the processor-memory interface rules given in Table 6. Similarly, we can provide more rules to describe fully pipelined versions of both microarchitectures described in this article.

We are also developing a compiler for hardware synthesis from TRSs. It translates TRSs into a standard hardware description language like Verilog.[11] 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 terms' grammar, when augmented with details such as instruction formats and sizes of various register files, buffers, memories, and so on, 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 state element. In case of such a conflict, one of the conflicting rules is selected to fire on the basis of some policy.

The synthesis area presents several challenging problems. First, good scheduling in the presence of resource constraints can be 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 scheduling can lead to implementations that perform poorly. Second, we often want to synthesize only part of the system described by a set of rules. For example, while synthesizing a microprocessor from the speculative processor rules, we may want to ignore the memory system and instead produce an interface specification for the external memory. A general solution to these problems is being studied. Nevertheless, we can 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 (see the sidebar at right).

Source-to-source transformations of TRSs also help in high-level synthesis. For example, if the generated circuit does not meet the clock requirement, then we must split each offending rule in the TRS into simpler rules. We systematically transformed the nonpipelined architecture represented by P_B into a simple five-stage pipeline and then further transformed the TRS obtained this way into a TRS representing a two-way superscalar architecture. Such source-to-source transformations dramatically reduce the number of rules a designer must write.

The promise of TRSs for computer architecture is the development of a set of integrated design tools for modeling, specification, verification, simulation, and synthesis. The conciseness and precision of TRSs, coupled with good tools, may radically alter the teaching of computer architecture. MICRO

Hardware synthesis of greatest common divisor

James C. Hoe
MIT

Euclid's algorithm for computing the greatest common divisor (GCD) of two numbers can be expressed in TRS notation as

GCD(x, y) if $x < y$ → GCD(y, x)
GCD(x, y) if $x \geq y$ and $y \neq 0$ → GCD($x-y$, y)

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

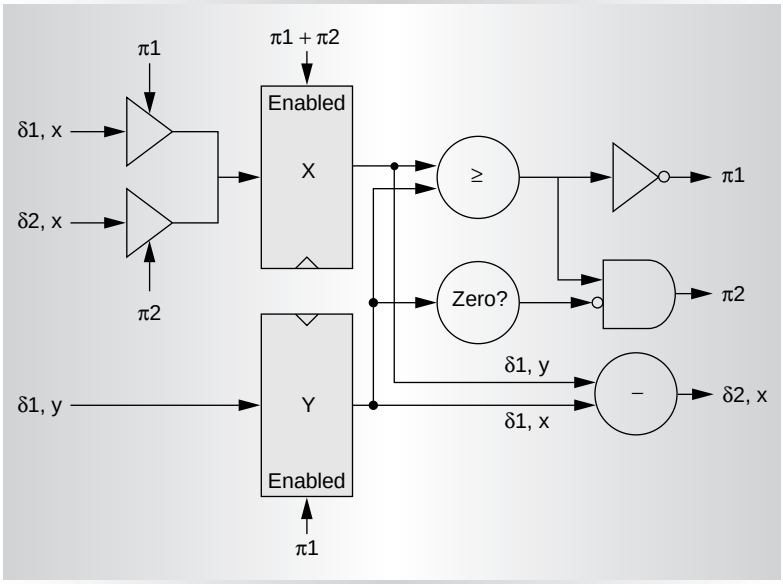


Figure A. The greatest common divisor circuit.

Acknowledgments

We 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 Fort Huachuca contract DABT63-95-C-0150.

References

1. J.R. Burch and D.L. Dill, "Automatic Verification of Pipelined Microprocessor Control," *Proc. Int'l Conf. Computer-Aided Verification*, Springer Verlag, June 1994.
2. B. Cook, J. Launchbury, and J. Matthews, "Specifying Superscalar Microprocessors in Hawk," *Proc. Workshop on Formal Techniques for Hardware and Hardware-Like Systems*, Marstrand, Sweden; published by Dept. of Computer Science, Chalmers Univ. of Technology, June 1998.
3. K.L. McMillan, "Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking," *Proc. Workshop on Formal Techniques for Hardware and Hardware-Like Systems*, Marstrand, Sweden; published by Dept. of Computer Science, Chalmers Univ. of Technology, June 1998.
4. P.J. Windley, "Formal Modeling and Verification of Microprocessors," *IEEE Trans. Computers*, Vol. 44, No. 1, Jan. 1995, pp. 54-72.
5. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, 1996.
6. X. Shen, Arvind, and L. Rudolph, "Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers," *Proc. 26th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., May 1999, pp. 150-161.
7. X. Shen, Arvind, and L. Rudolph, "CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems," *Proc. 13th ACM Int'l Conf. Supercomputing*, ACM, New York, June 1999.
8. F. Baader and T. Nipkow, *Term Rewriting and All That*, Cambridge Univ. Press, Cambridge, UK, 1998.
9. J.W. Klop, "Term Rewriting System," in *Handbook of Logic in Computer Science*, Vol. 2, S. Abramsky, D. Gabbay, and T. Maibaum, eds., Oxford University Press, 1992.
10. X. Shen and Arvind, "Design and Verification of Speculative Processors," *Proc. Workshop on Formal Techniques for Hardware and Hardware-Like Systems*, Marstrand, Sweden, June 1998; also appears as CSG Memo 400B, Laboratory for Computer Science, MIT, Cambridge, Mass., http://www.csg.lcs.mit.edu/pubs/csgmemo.html.
11. J.C. Hoe and Arvind, "Hardware Synthesis from Term Rewriting Systems," CSG Memo 421, Laboratory for Computer Sci., MIT, Cambridge, Mass., 1999; http://www.csg.lcs.mit.edu/pubs/csgmemo.html.

Arvind is Johnson professor of computer science and engineering at MIT. His current research interest is design, synthesis, and verification of architectures and protocols expressed using term rewriting systems. He has contributed to the development of dynamic dataflow architectures, the implicitly parallel programming languages Id and pH, and the compilation of these types of languages for parallel machines. Arvind received a BTech from IIT, Kanpur, India, and an MS and a PhD from the University of Minnesota. He is a member of the ACM and a fellow of the IEEE.

Xiaowei Shen is a PhD candidate in the Electrical Engineering and Computer Science Department at MIT. He is working on the development of scalable and adaptive shared-memory multiprocessor systems and the design and verification of architectures and protocols using term rewriting systems. His research interests also include compilers, networks, and many aspects of parallel and distributed computing. Shen received a BS and an MS in computer science and technology from the University of Science and Technology of China and an MS in electrical engineering and computer science from MIT.

Direct questions concerning this article to Arvind or Xiaowei Shen at the Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139; {arvind, xwshen}@lcs.mit.edu.