LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

# Deriving Superscalar Microarchitectures from Pipelined Microarchitectures

Computation Structures Group Memo 424
June 8, 1999

James C. Hoe and Arvind
MIT Laboratory for Computer Science
Cambridge, MA 02139
{jhoe,arvind}@lcs.mit.edu



This paper describes research done at the MIT Laboratory for Computer Science. Funding for this work is provided in part by the Defense Advanced Research Projects Agency of the Department of Defense under the Ft. Huachuca contract DABT63-95-C-0150.

# Deriving Superscalar Microarchitectures from Pipelined Microarchitectures

James C. Hoe and Arvind
MIT Laboratory for Computer Science
Cambridge, MA 02139
{jhoe,arvind}@lcs.mit.edu



June 8, 1999

**Abstract**

A method is given to systematically derive a superscalar microarchitecture from a pipelined microarchitecture where the microarchitectures are described using Term Rewriting Systems (TRS). The superscalar derivation is based on the rule composition property of TRS's. This automatic derivation, when coupled with a TRS-to-RTL compiler (TRAC) and commercial synthesis tools, dramatically expands the designer's ability to explore microarchitectures interactively. Computer generated estimates of circuit area and critical path delays are presented for five microarchitectures to study the cost-performance tradeoff.

## 1 Introduction

We have used Term Rewriting Systems (TRS) to describe speculative microarchitectures, memory models and complex cache-coherence protocols, and proven the correctness of these systems [1, 15, 14]. Recently, we have described the compilation of TRS's into a subset of Verilog that is simulatable and synthesizable by commercial tools [7] and shown how this synthesis technology can be used to design and explore pipelined microprocessors[8]. This follow-on paper describes how to automatically synthesize a superscalar microprocessor from a pipelined description of the processor using the *rule-composition* property of TRS's. The TRS framework permits the automation of several of the most critical steps in the processor design flow, thus reducing the amount of time and effort needed to design custom high-performance microprocessors.

TRS is a simple and intuitive formalism for describing hardware structure and behavior. In a TRS, hardware states are represented as terms generated by a grammar, and the behaviors are specified in the form of rules that specify when and how a term can be rewritten.

The rule-composition property allows new rules to be derived by composing existing rules without introducing illegal behaviors to the system.

In the TRS-based processor design flow, the architect starts by formulating a high-level specification of the processor's instruction set architecture (ISA) as a TRS. The goal at this stage is to define an ISA as precisely as possible without injecting implementation details. From such a description, TRAC, the Term Rewriting Architecture Compiler, can generate a Register Transfer Language description (RTL) of a single-issue, in-order, non-pipelined processor. The generated RTL can be simulated and synthesized by commercial tools.

Next, the architect manually transforms the ISA's TRS description into another TRS that corresponds to a pipelined mircoarchitecture. In this step, the architect makes high-level architectural decisions such as the locations of the pipeline stages. The pipeline stages are modeled as FIFO buffers, which makes most of the rules local. A rule typically dequeues a partially executed instruction from one FIFO, computes on it using only local information, and enqueues it into the next FIFO. The architect is also responsible for exposing and resolving any data and control hazards introduced by pipelining. To guard against possible errors introduced during this manual transformation, a semiautomatic verification technique can be used to show the correctness of the pipelined TRS against the original ISA specification using state-simulation techniques[1]. TRAC can take such asynchronous specifications and generate RTL's for synchronous pipelines.

Finally, the pipelined TRS is transformed into a superscalar TRS by devising composite rules. The effect of a composite rule is to apply more than one pipeline rule at each stage of the pipeline. As we will show, this can be done totally automatically once the degree of superscalarity is specified. The correctness of the resulting transformation is guaranteed because the rules derived by rule composition are always correct by TRS semantics.

Both pipelining and superscalar transformations are source-to-source in the TRS language and can be compiled into Verilog RTL descriptions using TRAC. Throughout the design flow, intermediate designs can be compiled. The RTL's of these intermediate designs can be evaluated immediately to steer design decisions in successive refinement steps. Presently, we are using a commercial tool, Synopsys's *RTL Analyzer*, to analyze the size and speed of the circuit. In addition, the operation of the processor on sample programs can be examined using a commercial Verilog RTL simulator. Based on the prompt feedback from these tools, an architect can rapidly explore a large number of architectural options and trade-offs.

**Related Work:** Over the years, a wide range of research has addressed the problem of synthesizing high-quality circuits from high-level specifications. On one end of the spectrum, commercial tool vendors are improving the capability of hardware compilation to support behavioral Verilog and VHDL[10] descriptions. On the other hand, researches in the area of Field Programmable Gate Arrays (FPGA) and Reconfigurable Computers (RC) (e.g.RAW[2], PAM[16], Splash[4]) have explored logic synthesis of algorithms expressed in software programming languages. Other efforts have focused on high-level optimizations, such as automatic pipelining [6]. Our research in processor microarchitecture synthesis is based on a higher-level design abstraction and can benefit from many of these developments.

Developments have also been made specifically in the synthesis of processor microarchitectures. The ADAS[12] environment accepts an ISA description in Prolog and emits a

pipelined VLSI implementation that is tuned for factors like instruction issue frequencies, pipeline stage latencies, etc. The ADAS design environment can be driven by ASIA[9] which automatically produces a custom instruction set architecture for an application. A similar line of research is pursued in Automatic Architecture Exploration (AAE)[5]. The Dagar[13] project accepts behavioral descriptions of digital systems in the form of dataflow graphs and outputs a customized microprogram-controlled pipelined datapath. As far as we know, automatic synthesis of superscalar microarchitectures has not been described yet.

**Paper Organization:** The next section presents the TRS-based behavioral description framework that is central to our approach. Section 3 applies the TRS description framework to describe a simple processor ISA, and Section 4 shows how to derive a pipelined processor from this initial ISA specification. Section 5 describes the transformation from a pipelined designs to its superscalar equivalent. In Section 6, we present an analysis of the superscalar design and show how the different tools can be used to guide design decisions. Finally, Section 7 gives a summary along with our conclusions.

# 2 TRS as a Hardware Description Language

A TRS consists of a set of terms and a set of rewriting rules. The general structure of a rewriting rule is:

$$s \ \mathit{if}\ p \rightarrow\ s'$$

where $s$ and $s'$ are are terms, and $p$ is a predicate on $s$.

A rule can be used to rewrite a term if the pattern implied by the left-hand-side of a rule matches the term or one of its subterms, and the corresponding predicate is true. The right-hand-side specifies the resulting term. In hardware descriptions, the terms represent states and the rules represent state transitions.

The effect of a rewrite is atomic, that is, the whole state is "read" in one step and if the rule is applicable then the state is updated in the same step. If several rules are applicable, then any one of them can be applied, and afterwards, all rules are re-evaluated for applicability on the new term. Starting from an initial term, successive rewriting progresses until the term cannot be rewritten using any rule.

All terms in a TRS have a *type*, and each rule is constrained to have the same type for the terms on both sides of the '$\rightarrow$'. The TRS notation accepted by TRAC includes built-in integers and common arithmetic and logical operators, and product and disjoint (non-recursive) union types. Two important abstract datatypes, arrays and bounded FIFO buffers, are also included to facilitate hardware description and synthesis.

Arrays are used to model register files and memories, and have only two operations defined on them. Syntactically, if *rf* is an array then *rf*[*r*] gives the value stored in location *r*, and *rf*[*r*:=*v*] gives the new value of the array after location *r* has been updated by value *v*. FIFO buffers provide the primary means of communication between different modules and pipeline stages. Syntactically, a buffer *bs* containing three elements is represented as *b1*;*b2*;*b3*. The two main operations on FIFO's are *enqueuing* and *dequeuing*. Enqueuing *b*

```
Type    PROC = Proc(PC,RF,IMEM,DMEM)
Type      PC = Bit[n]
Type    ADDR = Bit[n]
Type   VALUE = Bit[n]
Type      RF = Array VALUE[RNAME]
Type   RNAME = Reg0 || Reg1 || Reg2 || Reg3 || . . . . Regm
Type    IMEM = Array INST[PC]
Type    DMEM = Array VALUE[ADDR]
Type    INST = Loadc(RNAME,VALUE)
             || Loadpc(RNAME)
             || Op(MINOR,RNAME,RNAME,RNAME)
             || Bz(RNAME,RNAME)
             || Load(RNAME,RNAME)
             || Store(RNAME,RNAME)
Type   MINOR = Add || Sub || . . . .
```

Figure 1: TRS grammar for AX parameterized by the datapath width $n$, number of general purpose registers $m$, and the minor opcodes.

to *bs* yields *bs*;*b* while dequeuing from *b*;*bs* leaves the buffer in state *bs*. We also permit any "read" operation on the elements of FIFO buffers.

Although TRS's provide great flexibility in specifying state and state transitions, the TRS language with the restrictions described above essentially has the power of a finite state machine (FSM) because its terms cannot "grow". This allows TRAC to map a TRS into a synchronous FSM by (1) mapping TRS terms to storage elements (*e.g.*, registers, register files) and (2) mapping TRS rules to combinational logic that generates data and latch-enable signals for storage elements. TRAC follows this idea to generate a subset of Verilog that is simulatable and synthesizable by commercial tools. The main challenge for TRAC is in scheduling - how to fire the maximum number of rules in a given clock cycle without destroying the semantics that requires the rules to be fired one at a time.

# 3 AX: A Simple Processor

We use AX[1], a simple RISC instruction set, to demonstrate our synthesis procedure. The programmer visible state of AX consists of a program counter, a register file, instruction ROM (read-only memory) and data RAM (read-write memory). These states can be represented using the terms generated by the grammar in Figure 1. Type PROC is a product type with the constructor symbol Proc and four fields. The declaration of type INST demonstrates the use of an algebraic union to represent the AX instruction set. For simplicity, the program and data memory are modeled as storage arrays internal to the processor. However, when we discuss synthesis of these descriptions in Section 6, the memory arrays will be replaced by external memory interfaces represented as FIFO's.

A set of rewrite rules define AX's dynamic behavior. For example, the following rule describes the effect of executing an Add instruction:

| | | |
|---|---|---|
| Proc(*pc*,*rf*,*im*,*dm*) | | |
| *if* *im*[*pc*]≡Loadc(*rd*,*const*) | → | Proc(*pc*+1,*rf*[*rd*:=*const*],*im*,*dm*) |
| *if* *im*[*pc*]≡Loadpc(*rd*) | → | Proc(*pc*+1,*rf*[*rd*:=*pc*],*im*,*dm*) |
| *if* *im*[*pc*]≡Op(*op*,*rd*,*r1*,*r2*) | → | Proc(*pc*+1,*rf*[*rd*:=*op*(*rf*[*r1*],*rf*[*r2*])],*im*,*dm*) |
| *if* *im*[*pc*]≡Bz(*rc*,*rt*) & *rf*[*rc*]≡0 | → | Proc(*rf*[*rt*],*rf*,*im*,*dm*) |
| *if* *im*[*pc*]≡Bz(*rc*,*rt*) & *rf*[*rc*]≠0 | → | Proc(*pc*+1,*rf*,*im*,*dm*) |
| *if* *im*[*pc*]≡Load(*rd*,*ra*) | → | Proc(*pc*+1,*rf*[*rd*:=*dm*[*rf*[*ra*]]],*im*,*dm*) |
| *if* *im*[*pc*]≡Store(*ra*,*r*) | → | Proc(*pc*+1,*rf*,*im*,*dm*[*rf*[*ra*]:=*rf*[*r*]]) |

Figure 2: TRS rules for a non-pipelined AX.

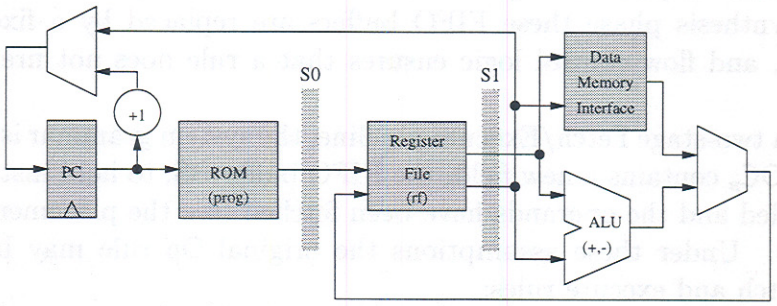


Figure 3: AX datapath without control. S0 and S1 are potential sites for inserting pipeline buffers.

Proc(*pc*,*rf*,*im*,*dm*) *if* *im*[*pc*]≡Op(Add,*rd*,*r1*,*r2*)
→ Proc(*pc*+1,*rf*[*rd*:=*rf*[*r1*]+*rf*[*r2*]],*im*,*dm*)

This rule can be examined in three parts: the match template (left-hand-side), the rewrite template (right-hand-side), and the predicate. The free variables in the match template begin are in italicized lower-case letters (e.g., *pc*, *rf*, etc.). Since the match template has no constants, it matches any term that has PROC's signature. The predicate will hold if the program counter points to an instruction memory location containing Op(Add,*rd*,*r1*,*r2*). When a term satisfies both the match template and the predicate, the rule's rewrite template specifies that the *pc* field should be incremented by 1 and register *rd* should be updated by *rf*[*r1*]+*rf*[*r2*]. Figure 2 gives the complete set of rules for AX in an abbreviated format. When synthesized, this TRS roughly corresponds to the datapath shown in Figure 3.

# 4 Pipelining Transformations

The simple datapath in Figure 3 can be pipelined by splitting each rule in Figure 2 into multiple sub-rules. Each sub-rule describes a sub-operation that uses its own set of resources. For example, in a two-stage pipeline design, the processing of an instruction can be broken down into separate fetch and execute steps. A pipelined design needs buffers to hold partially executed instructions. We model such buffers between pipeline stages as FIFO's of an unspecified but finite size. In a behavioral description, it is convenient if the operation of each stage can be described without reference to other stages. FIFO buffers provide this

| | | |
|---|---|---|
| *Type* | $\mathsf{PROC}_p$ | $= \mathsf{Proc}_p(\mathsf{PC},\mathsf{RF},\mathsf{BS},\mathsf{IMEM},\mathsf{DMEM})$ |
| *Type* | BS | $= \mathit{FIFO}\ \mathsf{ITEMP}$ |
| *Type* | ITEMP | $= \mathsf{Op}(\mathsf{MINOR},\mathsf{RNAME},\mathsf{VALUE},\mathsf{VALUE})$ |
| | | $\parallel \mathsf{Bz}(\mathsf{VALUE},\mathsf{VALUE})$ |
| | | $\parallel \mathsf{Load}(\mathsf{RNAME},\mathsf{ADDR})$ |
| | | $\parallel \mathsf{Store}(\mathsf{ADDR},\mathsf{VALUE})$ |

Figure 4: TRS grammar for a 2-stage pipelined AX.

isolation; most rules dequeue an input from one FIFO and enqueue the result into another FIFO. In the synthesis phase these FIFO buffers are replaced by a fixed-depth FIFO or simply registers, and flow control logic ensures that a rule does not fire if the destination FIFO is full.

To describe a two-stage Fetch/Execute pipeline, the system grammar is modified as shown in Figure 4. $\mathsf{PROC}_p$ contains a new field, the FIFO buffer BS, to hold instructions after they have been decoded and the operands have been fetched (see the placement of pipeline stage S1 in Figure 3). Under these assumptions the original Op rule may be replaced by the following two fetch and execute rules:

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,bs,im,dm) \\ & \quad \mathit{if}\ \ im[pc]{\equiv}\mathsf{Op}(op,rd,r1,r2)\ \mathit{and}\ r1{\notin}\mathit{Target}(bs)\ \mathit{and}\ r2{\notin}\mathit{Target}(bs) \\ \rightarrow & \mathsf{Proc}_p(pc{+}1,rf,bs;\mathsf{Op}(op,rd,rf[r1],rf[r2]),im,dm) \end{array}$$

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,\mathsf{Op}(op,rd,v1,v2);bs,im,dm) \\ \rightarrow & \mathsf{Proc}_p(pc,rf[rd{:=}\underline{op}(v1,v2)],bs,im,dm) \end{array}$$

Splitting a rule into smaller rules destroys the atomicity of the original rule and thus, can cause new behaviors which may not conform to the original specifications. Therefore, in addition to determining the appropriate division of work across the stages, the architect must also resolve any newly created hazards. For example, the fetch rule's predicate has been extended to check if the source register names are in $\mathit{Target}(bs)$, a shorthand for the set of target register names in $bs$. This condition prevents fetching when a RAW (read-after-write) hazard exists. If the architect makes a mistake in the transformation, the error would be revealed when an attempt is made to verify the equivalence of the pipelined processor against the initial specification via TRS simulation [1, 3].

As another example, consider the pair of Bz rules in Figure 2. Again, we can split the rules into their fetch and execute components. Both rules share the following instruction fetch rule:

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,bs,im,dm) \\ & \quad \mathit{if}\ \ im[pc]{\equiv}\mathsf{Bz}(rc,rt)\ \mathit{and}\ rc{\notin}\mathit{Target}(bs)\ \mathit{and}\ rt{\notin}\mathit{Target}(bs) \\ \rightarrow & \mathsf{Proc}_p(pc{+}1,rf,bs;\mathsf{Bz}(rf[rc],rf[rt]),im,dm) \end{array}$$

The two execute rules for the Bz instruction are:

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,\mathsf{Bz}(vc,vt);bs,im,dm)\ \mathit{if}\ vc{\equiv}0 \\ \rightarrow & \mathsf{Proc}_p(vt,rf,\epsilon,im,dm) \end{array}$$

| | | |
|---|---|---|
| $\mathsf{Proc}_p(pc,rf,itemp;bs,im,dm)$ | | |
| $if\ itemp{\equiv}\mathsf{Op}(op,rd,v1,v2)$ | $\rightarrow$ | $\mathsf{Proc}_p(pc,rf[rd{:=}\overline{op}(v1,v2)],bs,im,dm)$ |
| $if\ itemp{\equiv}\mathsf{Bz}(vc,vt)\ \&\ vc{\equiv}0$ | $\rightarrow$ | $\mathsf{Proc}_p(vt,rf,\epsilon,im,dm)$ |
| $if\ itemp{\equiv}\mathsf{Bz}(vc,vt)\ \&\ vc{\neq}0$ | $\rightarrow$ | $\mathsf{Proc}_p(pc,rf,bs,im,dm)$ |
| $if\ itemp{\equiv}\mathsf{Load}(rd,va)$ | $\rightarrow$ | $\mathsf{Proc}_p(pc,rf[rd{:=}dm[va]],bs,im,dm)$ |
| $if\ itemp{\equiv}\mathsf{Store}(va,v)$ | $\rightarrow$ | $\mathsf{Proc}_p(pc,rf,bs,im,dm[va{:=}v])$ |

Figure 5: Execute rules for a two-stage AX

$\mathsf{Proc}_p(pc,rf,\mathsf{Bz}(vc,vt);bs,im,dm)\ if\ vc{\neq}0$
$\rightarrow\ \mathsf{Proc}_p(pc,rf,bs,im,dm)$

In the fetch phase, the processor performs a weak form of branch speculation by incrementing *pc* without knowing the branch resolution. Consequently, in the execute phase, if the branch is resolved as taken, besides restarting *pc* at the correct value, we need to discard the speculatively fetched instructions in *bs*. This is indicated by setting *bs* to $\epsilon$ in the Bz-taken rule.

All of the AX rules can be partitioned into separate fetch and execute rules to represent a two-stage pipeline. A generic instruction fetch rule is:

$\mathsf{Proc}_p(pc,rf,bs,im,dm)$
$\quad if\ \ im[pc]{\equiv}inst\ and\ (Source(inst) \cap Target(bs) \equiv \emptyset)$
$\rightarrow\ \mathsf{Proc}_p(pc{+}1,rf,bs;Decode(inst),im,dm)$

*Source*(*inst*) is a shorthand for extracting the source register names from *inst*. *Decode*(*inst*) is the shorthand that maps *inst* to instruction templates where the register operands have been fetched. For example, *Decode*(Op(*op*,*rd*,*r1*,*r2*)) is Op(*op*,*rd*,*rf*[*r1*],*rf*[*r2*]). The grammar does not provide separate templates for Loadc and Loadpc instructions. Instead, *Decode*(Loadc(*rd*,*const*))=Op(Add,*rd*,*const*,0) and Loadpc(*rd*)=Op(Add,*rd*,*pc*,0). The execute rules for the two-stage pipeline are given in Figure 5. It should be noted that pipelines with different number of stages or placement of FIFO's can be described in a similar manner.

# 5 Transformation for Superscalar Execution

For superscalar execution, a pipelined microarchitecture processes multiple instructions in each pipeline stage. To achieve two-way superscalar execution, one needs to compose two rules that specify operations in the same pipeline stage into a new composite rule that combines the state transitions of both rules. Since the TRAC compiler generates RTL that executes the transitions of a rule in a single clock cycle, the compilation of composite rules results in RTL that can execute two instructions at a time. We illustrate this idea by extending the two-stage pipeline microarchitecture from Section 4 for two-way superscalar execution. After explaining the idea informally through an example composite rule, we will systematically derive the two-way superscalar rules.

## 5.1 Example of a Composite Rule

Consider the following Op and Bz fetch rules:

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,bs,im,dm) \\ & \quad \textit{if} \quad im[pc]{\equiv}\mathsf{Op}(op,rd,r1,r2) \quad \textit{and} \quad r1{\notin}\mathit{Target}(bs) \quad \textit{and} \quad r2{\notin}\mathit{Target}(bs) \\ \rightarrow & \mathsf{Proc}_p(pc{+}1,rf,bs;\mathsf{Op}(op,rd,rf[r1],rf[r2]),im,dm) \end{array}$$

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,bs,im,dm) \\ & \quad \textit{if} \quad im[pc]{\equiv}\mathsf{Bz}(rc,rt) \quad \textit{and} \quad rc{\notin}\mathit{Target}(bs) \quad \textit{and} \quad rt{\notin}\mathit{Target}(bs) \\ \rightarrow & \mathsf{Proc}_p(pc{+}1,rf,bs;\mathsf{Bz}(rf[rc],rf[rt]),im,dm) \end{array}$$

If we write the Bz fetch rule as if it was being applied to the term on the RHS of the Op fetch rule, it will look like the following rule:

$$\begin{array}{ll} & \mathsf{Proc}_p(pc{+}1,rf,bs;\mathsf{Op}(op,rd,rf[r1],rf[r2]),im,dm) \\ & \quad \textit{if} \quad im[pc{+}1]{\equiv}\mathsf{Bz}(rc,rt) \\ & \quad \textit{and} \quad rc{\notin}\mathit{Target}(bs;\mathsf{Op}(op,rd,rf[r1],rf[r2])) \quad \textit{and} \quad rt{\notin}\mathit{Target}(bs;\mathsf{Op}(op,rd,rf[r1],rf[r2])) \\ \rightarrow & \mathsf{Proc}_p((pc{+}1){+}1,rf,bs;\mathsf{Op}(op,rd,rf[r1],rf[r2]);\mathsf{Bz}(rf[rc],rf[rt]),im,dm) \end{array}$$

This rule is more specific than the general Bz fetch rule because it requires *bs* to contain a partially executed Op instruction. *Any specific instance of a TRS rule is guaranteed to be correct because it fires under fewer conditions.* Now we can combine the effect of the Op and Bz fetch rules into a single atomic rule as follows:

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,bs,im,dm) \\ & \quad \textit{if} \quad im[pc]{\equiv}\mathsf{Op}(op,rd,r1,r2) \quad \textit{and} \quad r1{\notin}\mathit{Target}(bs) \quad \textit{and} \quad r2{\notin}\mathit{Target}(bs) \\ & \quad \textit{and} \quad im[pc{+}1]{\equiv}\mathsf{Bz}(rc,rt) \\ & \quad \textit{and} \quad rc{\notin}\mathit{Target}(bs;\mathsf{Op}(op,rd,rf[r1],rf[r2])) \quad \textit{and} \quad rt{\notin}\mathit{Target}(bs;\mathsf{Op}(op,rd,rf[r1],rf[r2])) \\ \rightarrow & \mathsf{Proc}_p((pc{+}1){+}1,rf,bs;\mathsf{Op}(op,rd,rf[r1],rf[r2]);\mathsf{Bz}(rf[rc],rf[rt]),im,dm) \end{array}$$

The above rule is an example of a derived rule, that is, it can be derived from other TRS rules. *A derived rule is guaranteed to be correct, that is, it cannot introduce observable behaviors which were not permitted by the original rules.* However, if the derived rule replaces the rules from which it was derived, the system may not show some behaviors which were permitted otherwise. Although this error does not lead to illegal state transitions, it could result in a deadlock. Hence, unless other provisions are made, each new composite rule should be simply added to the original set of rules and should not replace any of the original rules.

The TRAC compiler will synthesize very different circuits for composite and non-composite rules. Since the effect of a composite rule takes place in one cycle, significantly more resources and circuitry are required to implement composite rules. Using its understanding of the abstract data type operations, the compiler also tries to simplify the predicate. For example, the predicate in the above rule can be simplified as follows:

$$\begin{array}{ll} & \mathsf{Proc}_p(pc,rf,bs,im,dm) \\ & \quad \textit{if} \quad im[pc]{\equiv}\mathsf{Op}(op,rd,r1,r2) \quad \textit{and} \quad im[pc{+}1]{\equiv}\mathsf{Bz}(rc,rt) \\ & \quad \textit{and} \quad r1{\notin}\mathit{Target}(bs) \quad \textit{and} \quad r2{\notin}\mathit{Target}(bs) \\ & \quad \textit{and} \quad rt{\notin}\mathit{Target}(bs) \quad \textit{and} \quad rc{\notin}\mathit{Target}(bs) \quad \textit{and} \quad rc{\neq}rd \quad \textit{and} \quad rt{\neq}rd \\ \rightarrow & \mathsf{Proc}_p((pc{+}1){+}1,rf,bs;\mathsf{Op}(op,rd,rf[r1],rf[r2]);\mathsf{Bz}(rf[rc],rf[rt]),im,dm) \end{array}$$

Complete superscalar fetching of all possible instruction pairs would require the com-

position of all combinations of the original fetch rules from the 2-stage pipelined microarchitecture. In general, given a pipeline stage with $N$ rules, a superscalar transformation leads to an $O(N^s)$ increase in the number of rules where $s$ is the degree of superscalarity. Fortunately, the mechanical nature of this tedious transformation makes it easy to handle by a computer aided synthesis system. Superscalar transformation also implies duplication of hardware resources such as register file ports, ALU's and memory ports. Hence, one may not want to compose all combinations of rules in a stage. For example, we may not want to compose any other execute rules with memory load or store rules if the memory interface can only accept one operation per cycle.

## 5.2 Deriving Composite Rules

A TRS rule $r$ on the set of terms $T$ can be expressed as a function $f$ whose domain $D$ and image $I$ are subsets of $T$. Given a rule $s$ *if* $p \rightarrow s'$, function $f$ may be written as follows:

$$f(s) = \mathit{if}\ \pi(s)\ \mathit{then}\ \delta(s)\ \mathit{else}\ s$$

where $\pi$ represents the firing condition derived from the predicate $p$ and the LHS pattern, and $\delta$ represents the function to compute the next state. (Please see [7] for detail.) The composition of two rules, $r_{1,2}$, i.e., rule $r_1$ followed by $r_2$, is described by the function $f_{1,2} = f_2 \circ f_1$, which may be written as follows:

$$\begin{aligned} f_{1,2}(s) &= \mathit{if}\ \pi_1(s)\ \mathit{then}(\mathit{if}\ \pi_2(\delta_1(s))\ \mathit{then}\ \delta_2(\delta_1(s))\ \mathit{else}\ s)\ \mathit{else}\ s \\ &= \mathit{if}\ \pi_1(s)\ \&\ \pi_2(\delta_1(s))\ \mathit{then}\ \delta_2(\delta_1(s))\ \mathit{else}\ s \end{aligned}$$

The domain $D_{1,2}$ of the composite function is only the subset of $D_1$ that produces the restricted image $(I_1 \cap D_2)$ using $f_1$. By this definition of composition, adding $r_{1,2}$ to a TRS that already contains $r_1$ and $r_2$ does not introduce any new behaviors because all transitions admitted by $r_{1,2}$ could be simulated by consecutive applications of $r_1$ and $r_2$. However, $r_1$ and $r_2$ cannot be replaced by $r_{1,2}$ because some transitions could be eliminated.

It is more convenient to express rule composition as a purely syntactic operation, so that the resulting composite rule can be expressed using standard TRS syntax. Thus, given the following two rules:

$s_1$ *if* $p_1 \rightarrow s_1'$ $\quad (r_1)$
$s_2$ *if* $p_2 \rightarrow s_2'$ $\quad (r_2)$

we first derive an instance of the second rule that is directly applicable to the RHS of the first rule such that

$s_1'$ *if* $p_2' \rightarrow s_2''$ $\quad (\mathit{instance\ of}\ r_2)$

This instance of rule 2 can then be composed with rule 1 as follows:

$s_1$ *if* $p_1'$ *and* $p_2' \rightarrow s_2''$ $\quad (r_{1,2})$

## 5.3 Derivation of Two-Way Superscalar Rules

To transform the 2-stage pipelined microarchitecture into a two-way superscalar microarchitecture requires derivation of a composite rule for each pair in the cross product of rules for each pipeline stage. We will present each composite rule under the general assumption that

there are sufficient ALU's and register file ports available. To make the rule derivation more interesting, we will assume the data memory is single ported.

### Superscalar Fetch rule

The superscalar version of the fetch rule that is given at the end of Section 4 can be composed with itself to produce the following rule:

$$
\begin{array}{ll}
& \mathsf{Proc}_p(pc,rf,bs,im,dm) \\
& \quad \mathit{if} \quad im[pc] \equiv inst \quad \mathit{and} \quad im[pc+1] \equiv inst' \\
& \quad \mathit{and} \quad Source(inst) \cap Target(bs) \equiv \emptyset \\
& \quad \mathit{and} \quad Source(inst') \cap (Target(bs) \cup Target(inst)) \equiv \emptyset \\
\rightarrow & \mathsf{Proc}_p((pc+1)+1, rf, bs; Decode(inst); Decode(inst'), im, dm)
\end{array}
$$

Superscalar execute rules are derived next. All legal combinations of the rules in Figure 5 are being enumerated in the following cases.

### Composing Op and other rules

If the first instruction is an Op instruction then the second instruction can always be executed. Most rules in the following table require additional read ports in the register file. Some combinations also require two write-ports.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{Proc}_p(pc,rf,\mathsf{Op}(op,rd,v1,v2);itemp;bs,im,dm)} \\
if\ itemp \equiv \mathsf{Op}(op',rd',v1',v2') & \rightarrow & \mathsf{Proc}_p(pc,(rf[rd:=\underline{op}(v1,v2)])[rd':=\underline{op}'(v1',v2')],bs,im,dm) \\
if\ itemp \equiv \mathsf{Bz}(vc,vt)\ \&\ vc \equiv 0 & \rightarrow & \mathsf{Proc}_p(vt,rf[rd:=\underline{op}(v1,v2)],\epsilon,im,dm) \\
if\ itemp \equiv \mathsf{Bz}(vc,vt)\ \&\ vc \neq 0 & \rightarrow & \mathsf{Proc}_p(pc,rf[rd:=\underline{op}(v1,v2)],bs,im,dm) \\
if\ itemp \equiv \mathsf{Load}(rd',va) & \rightarrow & \mathsf{Proc}_p(pc,(rf[rd:=\underline{op}(v1,v2)])[rd':=dm[va]],bs,im,dm) \\
if\ itemp \equiv \mathsf{Store}(va,v) & \rightarrow & \mathsf{Proc}_p(pc,rf[rd:=\underline{op}(v1,v2)],bs,im,dm[va:=v])
\end{array}
$$

### Composing Bz-Taken and other rules

There is no valid composition because the RHS of Bz-taken rule produces an empty FIFO buffer ($\epsilon$). Every execute-stage rule requires the FIFO buffer to look like "*inst*;*bs*" for some value of *inst*. Since (*inst*;*bs*) $\neq \epsilon$, no execute rule can fire immediately after a branch is taken.

### Composing Bz-Not-Taken and other rules

Executing a Bz-not-taken rule has no side-effects other than removing its template from the head of *bs*. Hence, composing a Bz-not-taken rule with any other rule produces a composite rule that is nearly identical to the second rule in the composition. This is true even if the second rule being composed is Bz-taken or Bz-not-taken.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{Proc}_p(pc,rf,\mathsf{Bz}(vc,vt);itemp;bs,im,dm)} \\
if\ itemp \equiv \mathsf{Op}(op,rd,v1,v2)\ \&\ vc \neq 0 & \rightarrow & \mathsf{Proc}_p(pc,rf[rd:=\underline{op}(v1,v2)],bs,im,dm) \\
if\ itemp \equiv \mathsf{Bz}(vc',vt')\ \&\ vc \neq 0\ \&\ vc' \equiv 0 & \rightarrow & \mathsf{Proc}_p(vt',rf,\epsilon,im,dm) \\
if\ itemp \equiv \mathsf{Bz}(vc',vt')\ \&\ vc \neq 0\ \&\ vc' \neq 0 & \rightarrow & \mathsf{Proc}_p(pc,rf,bs,im,dm) \\
if\ itemp \equiv \mathsf{Load}(rd,va)\ \&\ vc \neq 0 & \rightarrow & \mathsf{Proc}_p(pc,rf[rd:=dm[va]],bs,im,dm) \\
if\ itemp \equiv \mathsf{Store}(va,v)\ \&\ vc \neq 0 & \rightarrow & \mathsf{Proc}_p(pc,rf,bs,im,dm[va:=v])
\end{array}
$$

### Composing Load and other rules

Since we have assumed a single ported memory, it is not possible to compose a memory access rule with another memory access rule.

$$
\begin{array}{|lcl|}
\hline
\mathsf{Proc}_p(pc,rf,\mathsf{Load}(rd,va);itemp;bs,im,dm) & & \\
if\ itemp{\equiv}\mathsf{Op}(op,rd',v1,v2) & \rightarrow & \mathsf{Proc}_p(pc,(rf[rd{:=}dm[va]])[rd'{:=}\underline{op}(v1,v2)],bs,im,dm) \\
if\ itemp{\equiv}\mathsf{Bz}(vc,vt)\ \&\ vc{\equiv}0 & \rightarrow & \mathsf{Proc}_p(vt,(rf[rd{:=}dm[va]]),\epsilon,im,\overline{dm}) \\
if\ itemp{\equiv}\mathsf{Bz}(vc,vt)\ \&\ vc{\neq}0 & \rightarrow & \mathsf{Proc}_p(pc,(rf[rd{:=}dm[va]]),bs,im,dm) \\
\hline
\end{array}
$$

### Composing Store and other rules

$$
\begin{array}{|lcl|}
\hline
\mathsf{Proc}_p(pc,rf,\mathsf{Store}(va,v);itemp;bs,im,dm) & & \\
if\ itemp{\equiv}\mathsf{Op}(op,rd,v1,v2)\}\ & \rightarrow & \mathsf{Proc}_p(pc,rf[rd{:=}\underline{op}(v1,v2)],bs,im,dm[va{:=}v]) \\
if\ itemp{\equiv}\mathsf{Bz}(vc,vt)\ \&\ vc{\equiv}0 & \rightarrow & \mathsf{Proc}_p(vt,rf,\epsilon,im,\overline{dm}[va{:=}v]) \\
if\ itemp{\equiv}\mathsf{Bz}(vc,vt)\ \&\ vc{\neq}0 & \rightarrow & \mathsf{Proc}_p(pc,rf,bs,im,dm[va{:=}v]) \\
\hline
\end{array}
$$

Please note that these composite rules do not replace the original rules. All five rules in Figure 5 are still needed in case there is only one instruction in *bs*.

# 6 RTL Synthesis and Analysis

All TRS descriptions of AX processors and the derived pipelined and superscalar processors can be compiled into RTL Verilog descriptions using TRAC, the Term Rewriting Architecture Compiler[7]. Using commercial tools, these TRAC generated RTL's can be synthesized for a number of implementation technologies ranging from Xilinx's FPGA's to Synopsys' GTECH, a technology-independent logic representation. The latter target, when coupled with Synopsys' *RTL Analyzer*, has proved useful in architectural exploration because it provides quantitative information about circuit sizes and delays. The compilation times including both TRS-to-RTL and RTL-to-GTECH are short enough to make the exploration possible in real time.

## 6.1 TRS to RTL Compilation

Prior to synthesis, the datapath width ($n$) and the number of GPR's ($m$) in the parameterized TRS descriptions must be fixed. For the results in this paper, we have chosen to synthesize a 32-bit machine ($n$=32) with 4 general purpose registers ($m$=4) and two minor opcodes Add and Sub. Furthermore, we have chosen to externalize the instruction and data memory to present results for processor synthesis as opposed to a system-on-a-chip.

The TRS processor description can be converted to use external memory interfaces by introducing two FIFO's and a memory busy flag. MQOUT FIFO is used for issuing MLoad and MStore commands, and MQIN FIFO for retrieving MLoad results. The MBUSY status register records whether the memory interfaces are busy. The new definition of $\mathsf{PROC}_p$ for the 2-stage pipelined processor may be given as follows:

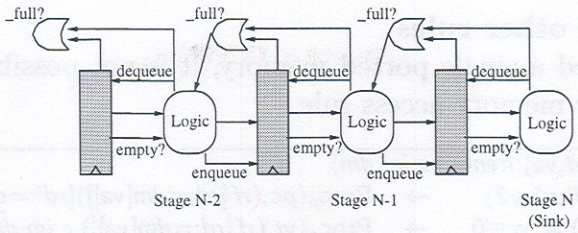


Figure 6: Synchronous pipeline firing with look ahead.

| | |
|---|---|
| *Type* | $\mathsf{PROC}_p = \mathsf{Proc}_p(\mathsf{PC},\mathsf{RF},\mathsf{BS},\mathsf{IMEM},\mathsf{MQIN},\mathsf{MQOUT},\mathsf{MBUSY})$ |
| *Type* | $\mathsf{MQIN} = \mathit{FIFO}\ \mathsf{VALUE}$ |
| *Type* | $\mathsf{MQOUT} = \mathit{FIFO}\ \mathsf{MCMD}$ |
| *Type* | $\mathsf{MCMD} = \mathsf{MStore}(\mathsf{ADDR},\mathsf{VALUE}) \parallel \mathsf{MLoad}(\mathsf{ADDR})$ |
| *Type* | $\mathsf{MBUSY} = \mathsf{Busy} \parallel \mathsf{NotBusy}$ |

During compilation, special directives are sent to TRAC to identify selected FIFO terms as synchronous or asynchronous I/O interfaces. In these results, we opted to synthesize the processor with an asynchronous instruction memory read port and a synchronous data memory read/write port. The instruction fetch takes place combinationally in a single clock cycle, while the data fetch by the Load instruction is spread over two clock cycles. In the first cycle, a MLoad command is issued on the MQOUT interface. On the next cycle, the processor retrieves the result from MQIN to complete the Load instruction. The processor can issue MStore commands on the MQOUT interface without blocking for completion.

During TRS-to-RTL compilation, TRAC maps each pipeline FIFO to a single register plus flow control logic to ensure a FIFO is not overflowed or underflowed during enqueue and dequeue operations (see the flow control depiction in Figure 6). A stall in an intermediate stage causes all up-stream stages to stall without affecting the down-stream stages' advance. During synthesis, one has to check that this combinational flow-control feedback does not become the critical path, especially in a deeply pipelined design.

## 6.2 Synthesis Results

We present the GTECH estimates of five synthesized processor designs: unpipelined, 2-stage pipelined, 2-stage 2-way superscalar processor (presented in Sections 3, 4 and 5, respectively) plus a 3-stage pipelined processor and its corresponding 2-way superscalar version also derived using our methodology. The 3-stage pipeline corresponds to the datapath in Figure 3 with both *S0* and *S1* instantiated. The superscalar transformation described in this paper is currently being implemented in the TRAC compiler, hence the superscalar TRS's have been generated by hand. Once the transformation has been implemented (expected in August, 1999) it will be straightforward to generate and report results for $s$-way ($s > 2$) superscalar implementations.

Figure 7 compares the amount of logic or area needed by the five implementations. The total area increases by 2.2 times going from an unpipelined implementation to a 3-stage 2-way superscalar pipeline. As expected, both pipelining and superscalarity increase the buffer requirements (from 0 to 3342) and control logic requirements (from 450 to 1099). Superscalarity also doubles the ALU requirements and increases the register-file size because of the increased number of ports.

| | Unpipelined | 2-stage | 2-stage 2-way | 3-stage | 3-stage 2-way |
|---|---|---|---|---|---|
| | area ( % ) | area ( % ) | area ( % ) | area ( % ) | area ( % ) |
| Prog. Counter | 321 ( 7.4) | 321 ( 5.6) | 321 ( 4.0) | 321 ( 5.1) | 321 ( 3.4) |
| Reg. File | 1786 ( 41.2) | 1792 ( 31.1) | 2157 ( 26.9) | 1792 ( 28.1) | 2157 ( 22.7) |
| Mem. Interface | 981 ( 22.6) | 985 ( 17.1) | 985 ( 12.3) | 985 ( 15.4) | 985 ( 10.4) |
| ALU | 796 ( 18.3) | 796 ( 13.8) | 1588 ( 19.8) | 796 ( 12.5) | 1588 ( 16.7) |
| Pipe. Buffer(s) | 0 ( 0.0) | 737 ( 12.8) | 1858 ( 23.2) | 1305 ( 20.4) | 3342 ( 35.2) |
| Logic | 450 ( 10.4) | 1122 ( 19.5) | 1099 ( 13.7) | 1179 ( 18.4) | 1099 ( 11.6) |
| Total | 4334 (100.0) | 5753 (100.0) | 8008 (100.0) | 6378 (100.0) | 9492 (100.0) |
| Normalized Total | 1.00 | 1.33 | 18.5 | 1.47 | 2.19 |

Figure 7: Circuit area distribution of AX processors (Unit area = 2-input *NAND* gate)

| | unpipelined | 2-stage | | 2-stage, 2-way | |
|---|---|---|---|---|---|
| | | Stage 1 | Stage 2 | Stage 1 | Stage 2 |
| Program Counter | *start* | *start* | – | *start* | – |
| Instruction Fetch | X | X | – | X | – |
| Operand Fetch | 4 | – | – | – | – |
| Raw Hazard | – | 12 | – | – | – |
| PC increment | – | – | – | 18 | – |
| *S1* | – | 6 | *start* | 8 | *start* |
| 32-ALU | 20 | – | 20 | – | 20 |
| Write Back | 6 | – | 5 | – | 7 |
| Total | 30+X | 18+X | 25 | 26+X | 27 |
| if X=20 | 50 | 38 | 25 | 46 | 27 |

Figure 8: Critical path break down of unpipelined, 2-stage pipelined and 2-stage 2-way superscalar AX processors. (Unit delay = 2-input *NAND* gate)

| | 3-stage | | | 3-stage, 2-way | | |
|---|---|---|---|---|---|---|
| | Stage 1 | Stage 2 | Stage 3 | Stage 1 | Stage 2 | Stage 3 |
| Program Counter | *start* | – | – | *start* | – | – |
| Inst Fetch or PC Inc | X | – | – | X | – | – |
| *S0* | 6 | *start* | – | 8 | *start* | – |
| Instruction Decode | – | 12 | – | – | 18 | – |
| *S1* | – | 8 | *start* | – | 11 | *start* |
| 32-ALU | – | – | 20 | – | – | 23 |
| Write Back | – | – | 5 | – | – | 8 |
| Total | 6+X | 20 | 25 | 8+X | 29 | 31 |
| if X=20 | 26 | 20 | 25 | 28 | 29 | 31 |

Figure 9: Critical path break down of 3-stage pipelined and 3-stage 2-way superscalar AX processors. (Unit delay = 2-input *NAND* gate)

Figures 8 and 9 break down the delay of each processor's critical path into contributions by different parts of the processor. For pipelined processors, separate critical paths are given for each stage. (A combinational path is assigned to a stage based on where the path starts). When a critical path involves instruction fetch, we use X to represent the instruction memory lookup delay since the actual delay will vary with the instruction memory size and implementation. The critical path analysis does not have to consider the latencies of data memory operations. If data memory latencies ever become a factor in the the critical path, we should modify the synchronous data memory interface to spend more cycles rather than lengthening the cycle time.

Ideally, converting an unpipelined microarchitecture to a $p$-stage pipelined microarchitecture should increase the clock frequency by $p$-fold, but this is rarely achieved in practice due to unbalanced partitioning and pipeline logic overhead. In our examples, assuming X is 20 time units, the 2-stage pipelined processor only achieves 39% higher clock frequency than the unpipelined version. The 3-stage pipeline processor achieves a 92% improvement

Analysis of the superscalar designs reveals areas where the synthesis procedure can be improved. For example, in the derived 2-way superscalar pipeline, the 2-way superscalar fetch rule references both instructions: $im[pc]$ and $im[pc+1]$. In a naive implementation where the processor uses two concurrent but independent instruction memory read ports to support 2-way superscalar fetch, the worst case instruction fetch latency becomes X+15 where 15 is the additional time required to compute the second instruction fetch address $(pc+1)$. Alternatively, because the 2-way superscalar fetch rule always reference two consecutive locations, we could provide a 2-instruction-wide memory interface that only requires a single address but returns both $im[pc]$ and $im[pc+1]$. The reported superscalar results use the new memory interface. Again assuming X is 20 time units, the peak performance of the 2-stage 2-way superscalar processor is now approximately twice that of the unpipelined processor at approximately twice the cost in terms of area. The 3-stage 2-way superscalar processor appears to have the best performance/area trade-off since it has nearly 3 times the performance of the unpipelined processors while consuming only 2.2 times more area. A caveat in this analysis is that we have only estimate the processors' peak performances based on cycle time and have ignored the effect of the instruction mix. Final design selection should also depend on simulations of the TRAC-generated RTL processor descriptions running software benchmarks.

# 7 Conclusion

To enable automated architectural exploration and synthesis, we need a high-level description methodology that can precisely specify the functions of a design without injecting implementation details. In this paper, we have described how the formalism of Term Rewriting Systems can be applied in this context. We began by describing an ISA and a pipelined representation of the ISA using high-level TRS abstractions. Next, from a high-level description of an instruction pipeline, we demonstrated a mechanical procedure, based on TRS rule composition, for deriving a superscalar processor description.

Each of the microarchitectural descriptions presented were synthesized using TRAC, a TRS to RTL compiler. The quality of the generated RTL was quickly evaluated using Syn-

opsys *RTL Analyzer*. The technology independent GTECH analysis presented in this paper, not only gave insight into the area-performance tradeoffs between various architectures but also showed areas where TRAC compilation could be improved. This high-level and quick feedback can also help the designer in inserting appropriate data bypasses. In future we plan to confirm the GTECH analysis by compiling the generated RTL's to real implementation technologies such as Xilinx Field Programmable Gate Arrays and Synopsys Cell-Based Arrays. There is no technical difficulty in such synthesis, though the compile times can be considerably longer than GTECH. There is also more work needed to define memory interfaces in a way that is not only precise and abstract during description but also efficient and practical during synthesis. In related research, we are also exploring the use of rule compositions in verification. Using rule composition it is also possible to eliminate pipeline buffers (FIFO's) one at a time[11]. The ability to couple synthesis and verification to the same source description will further increase the power and utility of the TRS design framework.

# References


[1] Arvind and X. Shen. Design and verification of processors using term rewriting systems. *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, May 1999.

[2] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines '99 (FCCM '99)*, Napa Valley, CA, Apr 1999.

[3] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of Conference on Computer-Aided Verification*, Stanford, CA, June 1994.

[4] M. Gokhale and R. Minnich. FPGA computing in a data parallel C. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, Napa, CA, April 1993.

[5] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetbility. In *Proceedings of the 34th DAC*, June 1997.

[6] S. Hassoun and C. Ebeling. Architectural retiming: Pipelining latency-constrained circuits. In *Proceedings of the 33rd DAC*, Las Vegas, NV, June 1996.

[7] J. C. Hoe and Arvind. Hardware synthesis from term rewriting systems. Technical Report CSG Memo 421, Laboratory for Computer Science, MIT, April 1999. (Submitted for publication).

[8] J. C. Hoe and Arvind. Micro-architecture exploration and synthesis via TRS's. Technical Report CSG Memo 422, Laboratory for Computer Science, MIT, April 1999. (Submitted for publication).

[9] I. J. Huang, B. Holmer, and A. Despain. ASIA: Automatic synthesis of instruction-set architectures. In *Proceedings of SASIMI-'93 Workshop*, Nara, Japan, October 1993.

[10] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral synthesis methodology for HDL-based specification and validation. In *Proceedings of the 32nd DAC*, San Francisco, CA, June 1995.

[11] J. Levitt and K. Olukotun. A scalable formal verification mythology for pipelined microprocessors. In *Proceedings of the 33rd DAC*, June 1996.

[12] I. Pyo, C. Su, I. Huang, K. Pan, Y. Koh, C. Tsui, H. Chen, G. Cheng, S. Liu, S. Wu, , and A. M. Despain. Application-driven design automation for microprocessor design. In *Proceedings of the 29th DAC*, Anaheim, CA, June 1992.

[13] V. K. Raj. DAGAR: An automatic pipelined microarchitecture synthesis system. In *Proceedings of ICCD89*, 1989.

[14] X. Shen, Arvind, and L. Rudolph. CACHET: An adaptive cache coherence protocol for distributed shared-memory systems. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[15] X. Shen, Arvind, and L. Rudolph. Commit-reconcile & fences (CRF): A new memory model for architects and compiler writers. In *Proceedings of the 26th ISCA*, Atlanta, Georgia, May 1999.

[16] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI*, 4(1):56–69, March 1996.

# Contents