LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Micro-architecture Exploration and Synthesis via TRS's

Computation Structures Group Memo 422
April 27, 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.

Micro-architecture Exploration and Synthesis via TRS's

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

April 27, 1999

Abstract

A new approach to designing pipelined and superscalar micro-architectures is presented. First, a precise specification of an Instruction Set Architecture (ISA) is developed using Term Rewriting Systems (TRS's). Second, a simple and intuitive behavioral description of a pipelined processor is developed by the architect, systematically, from the ISA specification. Semantically, various pipeline stages in this description communicate with each other asynchronously via bounded FIFO buffers. This pipelined architecture can be transformed further into a superscalar micro-architecture mechanically. It is shown how the TRAC compiler synthesizes synchronous hardware from these asynchronous descriptions. Using Synopsys tools, the TRAC-generated RTL (Register Transfer Language) descriptions for various micro-architectures are compared in terms of circuit sizes and delays. It is shown that these comparisons are invaluable in choosing a good pipeline design and in understanding the cost of different micro-architectures.

1 Introduction

We have used Term Rewriting Systems (TRS's) to describe speculative micro-architectures, memory models and complex cache-coherence protocols, and proven the correctness of these systems [1, 11, 10]. Recently, we have described the compilation of TRS's into a subset of Verilog that is simulatable and synthesizable by commercial tools [5]. In this paper, we discuss the design of pipelined micro-architectures in the TRS framework and show the problems that need to be solved in generating hardware for synchronous pipelines from asynchronous descriptions.

In the new design flow, a processor design begins with a specification of the processor's instruction set architecture (ISA). This specification, given in TRS, resembles the information commonly given by a processor's assembly programming manual. From this ISA specification, the architect manually arrives at a pipelined design by making high-level architectural decisions such as the depth and the locations of pipeline stages. Though the architect is responsible for exposing and resolving any data and control hazards arising from pipelining, much of this task potentially can be automated, and the correctness of the resulting TRS can be formally verified against the ISA specification. Furthermore, by taking advantage of the compositional properties of TRS rules, a TRS that corresponds to a superscalar version of the pipelined TRS can also be derived.

Both pipelining and superscalar transformations are source-to-source in the TRS language and can be compiled into Verilog RTL descriptions using TRAC, Term Rewriting Architecture Compiler. Using commercial tools, we can further compile the generated RTL to Synopsys' GTECH technology library to compare the different micro-architectures quantitatively. GTECH is a technology-independent logic representation which contains both sequential and combinational logic primitives. Every combinational path through a GTECH primitive is associated with a propagation delay in terms of logic-depth. For example, delay through a two-input *NAND* is one time unit whereas delay through a 32-bit adder is over 20 time units. Using this information, Synopsys' *RTL Analyzer* tool can estimate the propagation between two points and estimate the minimum cycle time. *RTL Analyzer* can also produce an area estimate. For the results presented, TRS-to-RTL compilation by TRAC requires less than a minute, while synthesis and analysis by Synopsys tools take tens of minutes. This type of rapid feedback provides invaluable insight in guiding the design exploration.

Paper organization: After a quick introduction to our TRS language in Section 2, we use it to describe a simple ISA in Section 3. Section 4 then explains how a pipelined processor can be derived from the initial ISA specification. In Section 5, we discuss the synthesis of synchronous pipelines from an asynchronous specification. Section 6 presents other related research in the field of architectural transformation and behavioral synthesis, as well as our conclusions.

2 TRS Hardware Descriptions

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

s *if* $p(s) \rightarrow s'$

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 rules represent state transitions.

The effect of a rewrite is atomic, that is, the whole state is "read" in one step and if the 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* 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 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 the same clock cycle without destroying the semantics that requires the rules to be fired one at a time. Without parallel execution of rules there can be no pipelining!

```
Type   PROC = Proc(PC,RF,PROG,MOUT,MIN,MBUSY)
Type     PC = Bit[16]
Type   ADDR = Bit[16]
Type  VALUE = Bit[16]
Type     RF = Array VALUE[RNAME]
Type  RNAME = Reg0 || Reg1 || Reg2 || Reg3
Type   PROG = Array INST[PC]
Type   INST = Loadc(RNAME,VALUE)
            || Loadpc(RNAME)
            || Add(RNAME,RNAME,RNAME)
            || Sub(RNAME,RNAME,RNAME)
            || Bz(RNAME,RNAME)
            || Load(RNAME,RNAME)
            || Store(RNAME,RNAME)
Type    MIN = FIFO VALUE
Type   MOUT = FIFO MCMD
Type   MCMD = MStore(ADDR,VALUE) || MLoad(ADDR)
Type  MBUSY = Busy || NotBusy
```

Figure 1: TRS grammar for AX_E, a simple processor.

3 AX_E: A Simple Processor

We describe AX_E, an application specific instruction processor (ASIP), with integrated instruction ROM (read-only memory), and then synthesize it to illustrate TRS's and TRAC.

Description of AX_E: Derived from AX[9], the programmer visible state of this simple architecture consists of a program counter, a register file, instruction ROM and external data memory interfaces. 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 six fields. The declaration of type INST demonstrates the use of an algebraic union to represent the AX_E instruction set.

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

```
      Proc(pc,rf,prog,mout,min,mbusy)
          if  prog[pc]=Add(rd,r1,r2)
→     Proc(pc+1,rf[rd:=rf[r1]+rf[r2]],
                    prog,mout,min,mbusy)
```

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 with small letters (e.g., *pc*, *rf*, ...). 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 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 one and register *rd* should be up-

Name	Firing Condition	New State
Load Const	*prog*[*pc*]=Loadc(*rd*,*const*)	Proc(*pc*+1,*rf*[*rd*:=*const*],*prog*,*mout*,*min*,*mbusy*)
Load PC	*prog*[*pc*]=Loadpc(*rd*)	Proc(*pc*+1,*rf*[*rd*:=*pc*],*prog*,*mout*,*min*,*mbusy*)
Add	*prog*[*pc*]=Add(*rd*,*r1*,*r2*)	Proc(*pc*+1,*rf*[*rd*:=*rf*[*r1*]+*rf*[*r2*]],*prog*,*mout*,*min*,*mbusy*)
Sub	*prog*[*pc*]=Sub(*rd*,*r1*,*r2*)	Proc(*pc*+1,*rf*[*rd*:=*rf*[*r1*]-*rf*[*r2*]],*prog*,*mout*,*min*,*mbusy*)
Branch Taken	*prog*[*pc*]=Bz(*rc*,*rt*) & *rf*[*rc*]=0	Proc(*rf*[*rt*],*rf*,*prog*,*mout*,*min*,*mbusy*)
Branch Not Taken	*prog*[*pc*]=Bz(*rc*,*rt*) & *rf*[*rc*]≠0	Proc(*pc*+1,*rf*,*prog*,*mout*,*min*,*mbusy*)
Load Start	*prog*[*pc*]=Load(*rd*,*ra*) & ¬*mbusy*	Proc(*pc*,*rf*,*prog*,*mout*;MLoad(*rf*[*ra*]),*min*,Busy)
Load Finish	*prog*[*pc*]=Load(*rd*,*ra*) & *min*=*v*;*min*'	Proc(*pc*+1,*rf*[*rd*:=*v*],*prog*,*mout*,*min*',NotBusy)
Store	*prog*[*pc*]=Store(*ra*,*r*)	Proc(*pc*+1,*rf*,*prog*,*mout*;MStore(*rf*[*ra*],*rf*[*r*]),*min*,*mbusy*)

Figure 2: TRS rules for a non-pipelined AX_E. (Current state: Proc(*pc*,*rf*,*prog*,*mout*,*min*,*mbusy*))

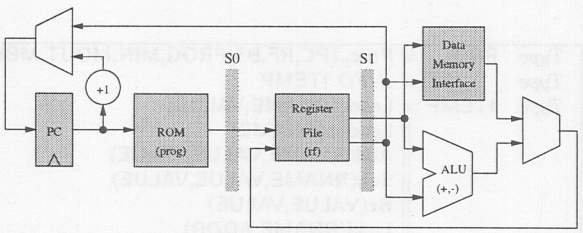

Figure 3: AX_E datapath without control. S0 and S1 are potential sites for inserting pipeline buffers.

	Non-pipe. area (%)	2-Stage area (%)	3-Stage area (%)
Prog. Counter	321 (7.4)	321 (4.9)	321 (4.1)
Reg. File	1786 (41.6)	1792 (27.4)	1792 (22.8)
Mem. Interface	963 (22.4)	963 (14.7)	963 (12.3)
ALU	796 (18.5)	796 (12.2)	796 (10.1
Pipe. Buffer(s)	0 (0)	1613 (246.6)	2862 (36.4)
Other	431 (10.0)	1066 (16.3)	1127 (14.3)
Total	4297 (100)	6551 (100)	7861 (100)

Figure 4: The area distribution of three AX_E processors excluding the instruction ROM. (Unit area = 2-input *NAND* gate)

dated by the sum of *rf*[*r1*] and *rf*[*r2*]. Figure 2 gives the complete set of rules for AX_E in an abbreviated format.

Synthesis of AX_E: The AX_E TRS can be viewed as a behavioral description of an implementation which processes each rule in one clock cycle. Thus, the processor executes instructions in a non-pipelined manner such that each instruction takes one cycle except the Load instruction which takes two. The clock of such a processor is likely to be slow because of the long combinational delay. We can confirm this hunch by compiling this TRS using TRAC to produce a synthesizable RTL description in Verilog[5]. A block diagram of the generated RTL is shown in Figure 3. The two columns under the "non-pipelined" heading in Figure 5 report the GTECH propagation delay in the critical path of this non-pipelined AX_E. The instruction ROM lookup delay is denoted by the variable X since it varies with the ROM's size and implementation. Column 2 of Figure 4 reports the GTECH area estimates of this AX_E. To put the GTECH estimates in context, the non-pipelined AX_E utilizes 26% of a XC4036XL-08 Field Programmable Gate Array and has a cycle time of 28 ns (ignoring the instruction ROM's contribution to propagation delay and area).

4 Pipelining Transformations

The architecture in Figure 3 can be pipelined by splitting each rule in its TRS (Figure 2) into multiple rules, each describing 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 queues. 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 isolation; most rules dequeue an input from one FIFO queue and enqueue the result into another FIFO queue. 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 6. $PROC_p$ 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). Furthermore, we assume simple pre-fetching of instructions, that is, the PC is always incremented by one. Under these assumptions the original Add rule may be replaced by the following two fetch and execute rules:

$$\begin{aligned} &\mathsf{Proc}_p(pc,rf,bs,prog,mout,min,mbusy) \\ &\quad \textit{if}\ \ prog[pc]=\mathsf{Add}(rd,r1,r2) \\ &\quad \textit{and}\ \ r1 \notin Target(bs)\ \textit{and}\ r2 \notin Target(bs) \\ &\rightarrow \mathsf{Proc}_p(pc{+}1,rf,bs;\mathsf{Add}(rd,rf[r1],rf[r2]), \\ &\qquad prog,mout,min,mbusy) \end{aligned}$$

$$\begin{aligned} &\mathsf{Proc}_p(pc,rf,\mathsf{Add}(rd,v1,v2);bs, \\ &\qquad prog,mout,min,mbusy) \\ &\rightarrow \mathsf{Proc}_p(pc,rf[rd:=v1{+}v2],bs, \\ &\qquad prog,mout,min,mbusy) \end{aligned}$$

Splitting a rule into smaller rules destroys the

	Non-pipelined		2-Stage				3-Stage					
			Stage 1		Stage 2		Stage 1		Stage 2		Stage 3	
	delay	cum.	delay	cum.	delay	cum.	delay	cum.	delay	cum.	delay	cum.
Program Counter	0	0	0	0	–	–	0	0	–	–	–	–
Instruction Fetch	X	X	X	X	–	–	–	–	–	–	–	–
S0	–	–	–	–	–	–	NA	20	0	0	–	–
Instruction Decode	11	11+X	21	21+X	–	–	–	–	14	14	–	–
S1	–	–	7	28+X	0	0	–	–	7	21	0	0
32-ALU	24	35+X	–	–	23	23	–	–	–	–	23	23
Write Back	3	38+X	–	–	3	26	–	–	–	–	3	26

Figure 5: Propagation delay in the critical paths of non-pipelined and pipelined AX_E's.(Unit delay = 2-input *NAND* gate)

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 with extra conditions to prevent fetching when a RAW (read-after-write) hazard is detected. Instruction fetch is stalled if the current instruction's operands depend upon the target register of any instruction waiting in the *bs* for execution. If the architect makes mistakes in the transformation, the errors would be revealed when an attempt is made to verify the equivalence of the pipelined processor against the initial specification via TRS simulation [1, 2].

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 can share the following instruction fetch rule:

$$\begin{aligned} &\mathsf{Proc}_p(pc,rf,bs,prog,mout,min,mbusy) \\ &\quad \textit{if} \quad prog[pc]{=}\mathsf{Bz}(rc,rt) \\ &\quad \textit{and} \quad rc \notin Target(bs) \ \textit{and}\ rt \notin Target(bs) \\ \rightarrow \quad &\mathsf{Proc}_p(pc{+}1,rf,bs;\mathsf{Bz}(rf[rc],rf[rt]), \\ &\qquad\qquad prog,mout,min,mbusy) \end{aligned}$$

The two execute rules for the Bz instruction are:

$$\begin{aligned} &\mathsf{Proc}_p(pc,rf,\mathsf{Bz}(vc,vt);bs,prog,mout,min,mbusy) \\ &\textit{if } vc{=}0 \ \rightarrow \mathsf{Proc}_p(vt,rf,\epsilon,prog,mout,min,mbusy) \\ &\textit{if } vc{\neq}0 \ \rightarrow \mathsf{Proc}_p(pc,rf,bs,prog,mout,min,mbusy) \end{aligned}$$

In the fetch phase, we do a weak form of branch speculation by incrementing the *pc* without knowing the branch resolution. Consequently, in the execute phase, if the branch is resolved to be taken, we need to discard the speculatively fetched instructions in *bs* and restore the *pc* to the correct value. This is indicated by setting the *bs* to ϵ in the rule.

The execute rules for the two-stage pipeline is given in Figure 7. It should be noted that pipelines with different number of stages or placement of FIFO's can be described in a similar manner. In particular, we also synthesized a three-stage pipeline where the third stage is created by inserting FIFO S0, shown in Figure 3. The GTECH area and timing estimates for non-pipelined, 2-stage and 3-stage AX_E's are reported in Figure 4 and Figure 5.

```
Type  PROC_p = Proc_p(PC,RF,BS,PROG,MIN,MOUT,MBUSY)
Type      BS = FIFO ITEMP
Type   ITEMP = Loadc(RNAME,VALUE)
             || Loadpc(RNAME)
             || Add(RNAME,VALUE,VALUE)
             || Sub(RNAME,VALUE,VALUE)
             || Bz(VALUE,VALUE)
             || Load(RNAME,ADDR)
             || Store(ADDR,VALUE)
```

Figure 6: TRS grammar for a 2-stage pipelined AX_E.

It is also possible to derive an n-way superscalar micro-architecture from any of these pipelined TRS's. The lack of space does not permit us to explain the details of this mechanical transformation. However, we have included the synthesis results for a 2-way superscalar version of the 3-stage pipelined processor in Figure 8.

Synthesis Results for Pipelined AX_E: 2254 units (or 52%) of additional area is needed to transform AX_E from non-pipelined to 2-stage pipelined operation. This overhead includes RAW hazard detection logic and pipeline buffer (S1) for the 69-bit instruction template. The minimum clock period is 28+X time units, limited by the critical path in the Fetch stage. Instruction and register operand fetch actually only require 21+X time units. The additional time is needed by the RAW hazard detection circuit. The critical path in the execute stage is 26 time units which is dominated by 23 time units in the 32-bit adder/subtracter. If X is 15 time units, we can expect upto 53% improvement in peak processor throughput, but the throughput can still be improved by further pipelining the Fetch stage.

In the 3-stage AX_E, we created separate pipeline stages for instruction fetch and decode by inserting buffer S0 (as shown in Figure 3). An additional 1310 units of area is needed. The new critical paths in the Fetch and Decode stage is 20 and 21 time units respectively. Assuming X is less than 20, the new critical path in the Fetch stage is now dominated by the 32-bit *pc* incrementer. The new minimum clock period, 26 time units required by the Execute stage, corresponds to a 104% improvement from the peak

throughput of the non-pipelined AX_E. This is the best we expect to do without pipelining the adder, incrementer or the ROM modules themselves.

Comparing the 2-way superscalar AX_E to the 3-stage AX_E, we see a significant increase in propagation delay of the Decode Stage due to the more sophisticated dual-instruction dispatch logic. In addition to a larger 4-read, 2-write port register file, we also observe a doubling of areas utilized by ALU and pipeline buffers. This feedback from synthesis, in conjunction with RTL simulation of application traces, should aid the architect in judging the merit of this design change.

One caveat in these results is that, presently, pipeline buffers are implemented as two-element FIFO's. As we will discuss next, we are working on a new scheme that will allow us to use simple registers as pipeline buffers directly. The new scheme will nearly halve the area overhead from pipelining and may improve the cycle time slightly.

5 Concurrent Firings

Using the TRS framework, important structures and concepts in pipelining can be conveyed in a concise and easy-to-understand manner. However, to achieve performance improvement, human insight and analysis of the rules are needed to place the pipeline buffers. After the pipeline stages are inserted, the designer has to guard against the potential hazards by modifying the predicates or by introducing new rules. In processor pipelining, it is also necessary to incorporate at least some capability for speculative execution to keep the pipeline filled. Last but not the least, the synthesis system has to interpret the rules properly; not all legal hardware implementations of a potentially pipelineable set of rules give the same performance. In fact, a simple minded implementation that fires only one rule in each clock cycle will show worse performance then a non-pipelined implementation.

To achieve pipelining, rules from various pipeline stages must fire together in the same clock cycle so the entire pipeline can advance synchronously. There are two challenges in achieving this effect. First, it has to be determined which rules can be fired concurrently without affecting the TRS semantics, which requires that at most one of the enabled rules fire at a time. Second, in an implementation, pipeline stage FIFO's are bounded, and thus, flow control needs to be enforced. When an enqueue operation appears in a rule, the predicate must include an implied condition that the FIFO is not full. Although, an implementation with bounded FIFO does not introduce illegal behaviors, in certain cases, it can preclude allowed behaviors, leading to deadlocks. However, in any realistic pipelining context, an acceptable implementation should exist otherwise the TRS itself must be flawed.

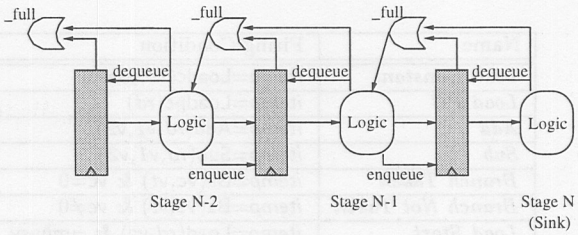

Figure 9: Synchronous pipeline firing with look ahead.

Elsewhere [5] we have given a precise definition of *conflict-free (CF) rules*, which guarantees that CF rules, if enabled, can be fired concurrently without affecting the correctness of the resulting state. There is a related notion of *dominant rule*, which states that if both rules, R_1 and R_2, are enabled in a state and the final state after applying R_2 is the same as after applying R_1 followed by R_2, then R_2 dominates R_1. In the two-stage pipeline, Branch-Taken rule dominates all fetch rules. All other rules are CF. In a pipelined description, rules from different stages should be either CF or dominating.

Consider an implementation that uses one-element FIFO's as pipeline buffers. Even for CF rules, this leads to a pipeline where every other pipelines stage contains a bubble in steady state. On each clock cycle, a parcel sees an opening (bubble) in the down stream buffer and advances into it on the clock edge. The stage vacated by the parcel cannot be filled on the same clock edge because the up-stream parcel cannot detect the vacancy until the following clock cycle. This cuts the throughput through the pipeline by a factor of two. Using a FIFO with a minimum depth of two elements effectively restores the pipeline throughput but at the cost of doubling the storage overhead.

Alternatively, instead of deciding if a FIFO can accept a new parcel solely based whether the FIFO is currently full, the test can be extended to look ahead into whether the FIFO will be dequeued at the coming clock edge, in which case it is safe to push a new parcel on the same clock edge (see a depiction in Figure 9). A stall in an intermediate stage causes all up-stream stages to stall without affecting the down-stream stages' advance. Given a pipeline with distinct source and sink, this allows us to use one-element FIFO's, i.e., registers, as pipeline buffers. It is easy for a compiler to order various pipeline stages and detect sources and sinks, as well as build the combinational logic for look-ahead.

Name	Firing Condition	New State
Load Constant	*itemp*=Loadc(*rd*,*const*)	Proc_p(*pc*,*rf*[*rd*:=*const*],*bs*,*prog*,*mout*,*min*,*mbusy*)
Load PC	*itemp*=Loadpc(*rd*)	Proc_p(*pc*,*rf*[*rd*:=*pc*],*bs*,*prog*,*mout*,*min*,*mbusy*)
Add	*itemp*=Add(*rd*,*v1*,*v2*)	Proc_p(*pc*,*rf*[*rd*:=*v1*+*v2*],*bs*,*prog*,*mout*,*min*,*mbusy*)
Sub	*itemp*=Sub(*rd*,*v1*,*v2*)	Proc_p(*pc*,*rf*[*rd*:=*v1*-*v2*],*bs*,*prog*,*mout*,*min*,*mbusy*)
Branch Taken	*itemp*=Bz(*vc*,*vt*) & *vc*=0	Proc_p(*vt*,*rf*,ϵ,*prog*,*mout*,*min*,*mbusy*)
Branch Not Taken	*itemp*=Bz(*vc*,*vt*) & *vc*$\neq$0	Proc_p(*pc*,*rf*,*bs*,*prog*,*mout*,*min*,*mbusy*)
Load Start	*itemp*=Load(*rd*,*va*) & $\neg$*mbusy*	Proc_p(*pc*,*rf*,*itemp*;*bs*,*prog*,*mout*;MLoad(*va*),*min*,Busy)
Load Finish	*itemp*=Load(*rd*,*va*) & *min*= *v*;*min*'	Proc_p(*pc*,*rf*[*rd*:=*v*],*bs*,*prog*,*mout*,*min*',NotBusy)
Store	*itemp*=Store(*va*,*v*)	Proc_p(*pc*,*rf*,*bs*,*prog*,*mout*;MStore(*va*,*v*),*min*,*mbusy*)

Figure 7: Execute rules for a two-stage AX_E. (Current State: Proc_p(*pc*,*rf*,*itemp*;*bs*,*prog*,*mout*,*min*,*mbusy*))

	Stage 1		Stage 2		Stage 3	
	delay	cum.	delay	cum.	delay	cum.
Program Counter	0	0	–	–	–	–
Instruction Fetch	–	–	–	–	–	–
S0	NA	26	0	0	–	–
Instruction Decode	–	–	18	18	–	–
S1	–	–	14	32	0	0
32-ALU	–	–	–	–	23	23
Write Back	–	–	–	–	8	21

	area (%)
Program Counter	321 (2.4)
Register File	2157 (15.8)
Memory Interface	963 (7.1)
ALU	1588 (11.7)
Pipeline Buffer(s)	7490 (55.0)
Other	1101 (8.1)
Total	13620 (100)

Figure 8: The GTECH results for a 2-way superscalar, 3-stage pipelined AX_E.

6 Related Work and Conclusions

The goal of high quality synthesis from high-level specifications has attracted many researchers over the years. For example, the ADAS[7] environment accepts an ISA description in Prolog and emits a VLSI implementation using a combination of tools. During behavioral synthesis phase, the Piper tool attempts to pipeline the micro-architecture while taking into account factors like instruction issue frequencies, pipeline stage latencies, etc. Another tool called ASIA[6] automatically produces a suitable instruction set architecture for ADAS from an application program. A similar research direction is being pursued in the Automatic Architecture Exploration (AAE) project by Hadjiyiannis, et al.[3]. In a slightly different direction, the Dagar[8] project accepts behavioral descriptions of digital systems in the form of dataflow graphs. For each description, the tool outputs a customized microprogram-controlled pipelined datapath and its accompanying optimized microcode. All these systems make use of high-level information to automatically determine architectural parameters like the number of arithmetic units, pipeline stages, etc. Recently, progress has also been made in automating higher-order architectural transformations such as pipelining an arbitrary synchronous circuit to reduce cycle time [4].

A true high-level specification should allow an architect to describe the functions of a design without an implementation bias. Such specifications can open up the design space for computer-aided architectural exploration and synthesis. We have shown that TRS's have this potential. On one hand TRS's can be used to describe and synthesize different micro-architectures for the same ISA using the TRAC compiler. On the other hand the TRS description of a micro-architecture can be transformed into another TRS that permits pipelining or superscalar execution. Next, we plan to automate some of these transformations and mechanically verify the correctness of some others. In near future we plan to synthesize cache-coherence engines for sophisticated cache-coherence protocols [10].

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

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

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

[5] 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).

[6] 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.

[7] 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 mi-

croprocessor design. In *Proceedings of the 29th DAC*, Anaheim, CA, June 1992.

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

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

[10] 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.

[11] 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.