LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Hardware Synthesis from Term Rewriting Systems

Computation Structures Group Memo 421
April 9, 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.

Hardware Synthesis from Term Rewriting Systems

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

April 9, 1999

Abstract

Term Rewriting System (TRS) is a good formalism for describing concurrent systems that embody asynchronous and nondeterministic behavior in their specifications. Elsewhere, we have used TRS's to describe speculative micro-architectures and complex cache-coherence protocols, and proven the correctness of these systems. In this paper, we describe the compilation of TRS's into a subset of Verilog that is simulatable and synthesizable by commercial tools. TRAC, Term Rewriting Architecture Compiler, enables a new hardware development framework that can match the ease of today's software programming environment. TRAC reduces the time and effort in developing and debugging hardware. For several examples, we compare TRAC-generated RTL's with hand-coded RTL's after they are both compiled for Field Programmable Gate Arrays by the latest Xilinx tools. Preliminary results look very promising.

1 Motivation

Term Rewriting Systems (TRS's)[2] have been used extensively to give operational semantics of programming languages. More recently, we have used TRS's in computer architecture research and teaching. TRS's have made it possible, for example, to describe a processor with out-of-order and speculative execution succinctly in a page of text[1]. Such behavioral descriptions in TRS's are also amenable to formal verification because one can show if two TRS's "simulate" each other. This paper describes hardware synthesis from TRS's.

We describe the Term Rewriting Architecture Compiler (TRAC) that compiles high-level behavioral descriptions in TRS's into a subset of Verilog that is simulatable and synthesizable by commercial tools. The TRAC compiler enables a new hardware design framework that can match the ease of today's software programming environment. By supporting a high level of abstraction in design entry, TRAC reduces the level of expertise required for hardware design. By eliminating human involvement in the lower-level implementation tasks, the time and effort normally associated with developing and debugging hardware is reduced. These same qualities also make TRAC an attractive tool for experts to prototype large designs.

This paper describes the compilation of TRS into RTL via simple examples. Section 2 presents an introduction to TRS's for hardware descriptions. Section 3 describes TRAC's strategy for mapping TRS's to hardware and the issues in generating good hardware. Section 4 compares TRAC-generated RTL against hand-coded RTL after each is compiled for Field Programmable Gate Arrays (FPGA) using the latest Xilinx synthesis tools. Section 5 surveys related work in high-level hardware description and synthesis. Finally, Section 6 concludes with a few brief remarks.

2 Hardware Description using TRS's

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 (LHS) of a rule

matches the term or one of its subterms, and the corresponding predicate is true. The right-hand-side (RHS) 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 a few abstract datatypes such as arrays and FIFO's.

Example 1 (GCD): Euclid's Algorithm for finding the greatest common divisor (GCD) of two integers may be written as follows in TRS notation:

Type GCD = Gcd(NUM, NUM)
Type NUM = Bit[32]

GCD Mod Rule
Gcd(a, b) *if* $a \geq b$ *and* $b \neq 0 \rightarrow$ Gcd($a-b$, b)
GCD Flip Rule
Gcd(a, b) *if* $a < b \rightarrow$ Gcd(b, a)

The terms of this TRS have the form Gcd(a,b), where a and b are 32-bit integers. The answer is the first subterm of Gcd(a,b) when Gcd(a,b) cannot be reduced any further. For example, the term Gcd(2,4) can be reduced by applying the *Flip* and *Mod* rules to produce the answer 2: Gcd(2,4) $\rightarrow$ Gcd(4,2) $\rightarrow$ Gcd(2,2) $\rightarrow$ Gcd(0,2) $\rightarrow$ Gcd(2,0) □

Example 2 (GCD_2): We give yet another implementation of GCD to illustrate some modularity and types issues. Suppose we have the following TRS to implement the *mod* function.

Type VAL = Mod(NUM, NUM)
|| Val(NUM)
Type NUM = Bit[32]

Mod Iterate Rule
Mod(a, b) *if* $a \geq b \rightarrow$ Mod($a-b$, b)
Mod Done Rule
Mod(a, b) *if* $a < b \rightarrow$ Val(a)

In this TRS, VAL is a union type with two disjuncts, Val and Mod. It is because of this type declaration that the *Mod Done Rule* does not violate the type discipline - both sides of the rule have the type VAL. Using this definition of *mod*, GCD can be written as follows:

Type GCD_2 = Gcd_2(VAL, VAL)

GCD_2 Flip&Mod Rule
Gcd_2(Val(a), Val(b)) *if* $b \neq 0$
$\rightarrow$ Gcd_2(Val(b), Mod(a, b)) □

We do not permit recursive type definitions in TRS's for hardware description. This restriction, together with the requirement that both sides of a rule have the same type, guarantees that the size of each term is finite and the size does not change by applying the rewriting rules.

We enrich the TRS notation by introducing several *abstract datatypes*. The two of greatest interest are *arrays* and *bounded FIFO buffers*. Arrays are used to model register files and memories, and have only two operations defined on them. 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. Generally pipeline latches are modeled as FIFO buffers whose size is restricted to be one. 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.

Example 3 (Single-Cycle RISC Processor): The state of a unpipelined, simple RISC processor is captured by its program counter (pc), register file (rf) and memory (mem). Such a processor can be specified as a TRS by giving a rewrite rule for each instruction. The following rule is for the execution of the *Add* instruction.

$Proc_s(pc, rf, mem)$
if $mem[pc] \equiv$ Add(rd,$r1$,$r2$)
$\rightarrow$ $Proc_s(pc+1, rf[rd:=v], mem)$
where $v = rf[r1]+rf[r2]$

The processor which we synthesized has four 32-bit general purpose registers and 7 instructions: move PC to register, load immediate, register-to-register addition and subtraction, branch if zero, memory load and store. □

Example 4 (Pipelined RISC Processor): The processor in Example 3 can be pipelined by introducing FIFO's as pipeline-stage buffers and by systematically splitting each rule into local rules for various pipeline stages. In the following, we introduce the

pipeline buffer *bs* and split the *Add* and *Bz* instruction rules into *Fetch* and *Execute* stage rules:

Fetch Rule
$\mathsf{Proc}_p(pc,\ rf,\ bs,\ mem)$
$\rightarrow\ \mathsf{Proc}_p(pc{+}1,\ rf,\ bs;mem[pc],\ mem)$
Add Rule
$\mathsf{Proc}_p(pc,\ rf,\ \mathsf{Add}(rd,r1,r2);bs,\ mem)$
$\rightarrow\ \mathsf{Proc}_p(pc,\ rf[rd{:=}\ v],\ bs,\ mem)$
$\qquad\qquad where\ v = rf[r1]{+}rf[r2]$
Branch-Taken Rule
$\mathsf{Proc}_p(pc,\ rf,\ \mathsf{Bz}(rc,ra);bs,\ mem)\ if\ rf[rc]{\equiv}0$
$\rightarrow\ \mathsf{Proc}_p(rf[ra],\ rf,\ \epsilon,\ mem)$
Branch-Not-Taken Rule
$\mathsf{Proc}_p(pc,\ rf,\ \mathsf{Bz}(rc,ra);bs,\ mem)\ if\ rf[rc]{\neq}0$
$\rightarrow\ \mathsf{Proc}_p(pc,\ rf,\ bs,\ mem)$

Notice the *Fetch* rule is always ready to fire. At the same time one of the execute stage rules may be ready to fire as well. This is the first time we have seen an example where more than one rule can be enabled on a given state. Even though according to TRS semantics, only one rule should be fired in each step, we will see that our compiler tries to fire as many rules in parallel as possible. Parallel firing is possible when rules are "conflict free". Without parallel firing of rules we won't get the pipelining effect we want.

Since there is a race to update the *pc* between the *Fetch* and the *Branch Taken* rules, the above rules can exhibit nondeterministic behavior. Specification of microprocessors and cache-coherence protocols often entails nondeterminism, even though a given realization is usually completely deterministic. Our compiler can handle such nondeterministic TRS's. □

In addition to the TRS-to-RTL compilation to be described shortly, we are developing source-to-source TRS transformations that can achieve the kind of pipelining described in Example 4. The dependence between the rules has to be analyzed carefully to ensure the correctness of all such transformations. Presently, human intervention is required to guide the transformation process at the high level. It is also possible to automatically derive the rules for a superscalar version of the pipelined processor in Example 4.

TRS, the way we have presented it, is a closed system, but we are experimenting with different notation and semantics to support input and output (I/O). The two GCD examples only require mechanisms for initializing the starting state of a new computation and for observing progress of the state changes. The processor examples requires more sophisticated I/O mechanisms where precise timing can be controlled to handshake with external memory.

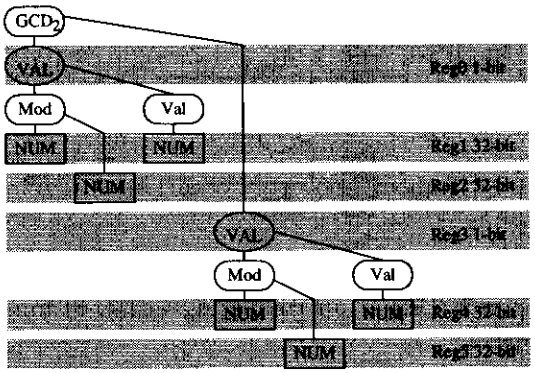

Figure 1: A graph representation of the GCD_2 grammar from Example 2 and its mapping to registers

3 Basic Synthesis Strategy

Although TRS's provide great flexibility in specifying state and state transitions, a TRS description should be reminiscent of a finite state machine (FSM). TRAC maps a TRS to 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.

3.1 Mapping Terms to Storage Elements

To extract state registers, it is convenient to view the term grammar as the tree shown in Figure 1. Each Bit node at the leaf implies a register. A union type node has a branch for each of the disjuncts. At any time, only the registers in one branch of a union node hold meaningful data — a VAL term is used either as $\mathsf{Val}(x)$ or $\mathsf{Mod}(y, z)$ but not both. Thus, a union node implies a *tag* register (one-bit wide in this example) to record which branch is active. As an optimization, registers from different branches can share the same physical register. In Figure 1, the registers aligned horizontally are mapped to the same register.

TRAC does not synthesize arrays and other abstract datatypes. The user or the library is expected to provide a Verilog module in RTL for each abstract datatype. When an abstract datatype is used in a TRS, TRAC instantiates the corresponding Verilog module in the RTL and makes appropriate connections to the interface.

3.2 Mapping Rules to Logic

A TRS rule '$s\ if\ p(s) \rightarrow s'$' has two components, π, the firing condition, and δ, the change-of-state func-

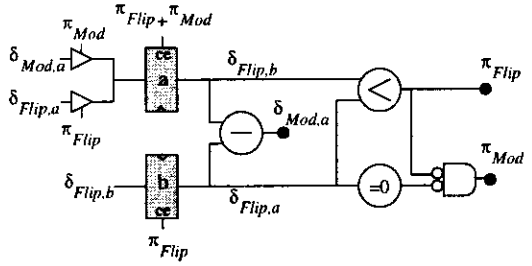

Figure 2: Circuit for computing Gcd(a, b) from Example 1.

tion. π, has the signature $typeof(s) \rightarrow Bool$, and is based on the predicate $p(s)$ and the LHS pattern. δ has the signature $typeof(s) \rightarrow typeof(s)$ and is based on the primitive functions used on the RHS to define the next state. δ can be implemented using only combinational logic. Therefore, if s represents the current state then s', the state to be latched at the next clock edge, is defined as follows:

$$if\ \pi(s)\ then\ s' = \delta(s)\ else\ s' = s$$

A rule generally modifies only a part of the state. Therefore, for each part of a term that is rewritten by a rule R, δ_R generates the next state value and π_R is fed to the latch-enable signal of all the affected registers. The π and δ functions for the GCD in Example 1 are given below. The input for these functions is the term Gcd(a,b), which is really just the two registers a and b.

$$\begin{aligned} \pi_{Mod} &= a \geq b \text{ and } b \neq 0 \\ \pi_{Flip} &= a < b \\ \delta_{Mod,a} &= a - b \\ \delta_{Mod,b} &= b \\ \delta_{Flip,a} &= b \\ \delta_{Flip,b} &= a \end{aligned}$$

Notice the *Mod* rule does not rewrite b. Hence $\delta_{Mod,b}$ is a no-op, and there is no need to generate it and store it back into register b in case π_{Mod} is true. In this example both the rules update the state variable a but only one of them can actually fire in a given state. Whenever the firing conditions are mutually exclusive, the latch enable can be simply the logical-OR of the firing conditions (*e.g.* , $\pi_{Mod}+\pi_{Flip}$ in this example), and the next state values can selected by their own π's at the multiplexer. Figure 2 shows the FSM generated by interconnecting π and δ for Example 1.

However, in general, several π's could be asserted, i.e., several rules could be applicable. In the simplest solution, a new set of disjoint triggers $\phi_1, ..., \phi_n$ can be generated using a round-robin priority encoder fed by $\pi_1, ..., \pi_n$. ϕ's replace π's at the multiplexers and at the latch enable OR-gates. This arbitration is simple and correct, but the circuit is inefficient and only allows one rewrite per cycle.

3.3 Exploiting Parallelism

To exploit parallelism in a TRS, one would like to execute as many non-interfering rewrites as possible on each clock cycle. In general it is not safe to allow two arbitrary rule instances to execute in the same clock cycle even if they modify disjoint parts of the state. This is because executing one of them can alter the value of the π or the δ function of the other. Under such a condition the rules must be executed in some order.

Based on this observation, we can define a conflict-free relationship ($\mathcal{CF}$) between two rules such that it is safe to execute $\mathcal{CF}$ rules in the same clock cycle using the FSM infrastructure defined in the previous section.

Conflict-Free Rules: Two rule instances, R_i and R_j, are $\mathcal{CF}$, if and only if, $\forall\ s$, if $\pi_i(s) \cdot \pi_j(s)$ *then*

1. $\pi_i(\delta_j(s)) \cdot \pi_j(\delta_i(s))$
2. $\delta_i(\delta_j(s)) \equiv \delta_j(\delta_i(s)) \equiv (\delta_i(s) \uplus \delta_j(s))$

These conditions stipulate that if two $\mathcal{CF}$ rule instances are ever applicable to the same state s then (1) the application of one rule does not cancel the applicability of the other rule on the term obtained by applying the first rule, (2) the order in which the rules are applied does not affect the final term, and furthermore, the final term can be constructed by merging $\delta_i(s)$ and $\delta_j(s)$. The merge operation ($\uplus$) can be defined such that, if only δ_i (or only δ_j) produces a candidate value for a register, the next-state for that register can be taken directly from $\delta_i(s)$ (or $\delta_j(s)$). If both δ_i and δ_j produce candidate values for the same register, the next-state value is taken from $\delta_i(s)$ if $\delta_i(s) \equiv \delta_j(s)$. Otherwise, the merged next-state is undefined.

In general, an exact test for $\mathcal{CF}$ relationship between two arbitrary rule instances is expensive (Finding an s such that $\pi_i(s)$ *and* $\pi_j(s)$ is like solving SAT). Instead, TRAC performs several conservative tests to find as many $\mathcal{CF}$ relationships as possible. First, two rule instances that read and write non-overlapping parts of the systems are $\mathcal{CF}$. If two rule instances do not rewrite the same registers, and if none of the registers affected by the δ of one is used by the π of the other, then the two rules are $\mathcal{CF}$ since this condition is stronger than the requirement for $\mathcal{CF}$. Lastly, TRAC symbolically analyzes pairs

of π's to conservatively determine when a pair can never be satisfied simultaneously and thus are $\mathcal{CF}$ by default.

After TRAC has establish $\mathcal{CF}$ relationships between as many rule instances as possible, a graph of rule instances can be constructed by adding an edge between each non-$\mathcal{CF}$ pairs. Scheduling groups is formed by partitioning the graph into connected components. Different groups never interfere and can be scheduled independently. For each group, a round-robin priority encoder can be used to map π to ϕ for arbitration. For a small group, an $n \times n$ look-up table can be computed off-line to encode π to ϕ where more than one ϕ can be asserted if the asserted π's are $\mathcal{CF}$.

Based on the analysis suggested above and taking into account some properties of FIFO buffers, it can be shown that the rules of Example 4 are $\mathcal{CF}$ except for the *Fetch* and the *Branch-Taken* rule. However, it can be shown that the *Branch-Taken* rule dominates the *Fetch* rule in the sense that the effect of applying the *Branch-Taken* rule after the *Fetch* rule is the same as not applying the *Fetch* rule at all ($\delta_{BzN}(s) \equiv \delta_{BzN}(\delta_{Fetch}(s))$). Thus, instead of arbitrating between these two rules, the compiler gives priority to the *Branch-Taken* rule.

4 Performance Evaluation

TRAC generates RTL Verilog that can be synthesized to a variety of technologies by commercial tools like Synopsys. In this paper, we evaluate the quality of the TRAC-generated RTL's against hand-coded RTL when compiled for Xilinx FPGA's. In future, we will also compile for gate arrays and standard cells technologies. These less restrictive technologies should provide a more accurate comparison.

Synthesis of the GCD Circuit: Both Example 1 and 2 are compiled by TRAC. The compile time is less than 2 sec on a 166MHz PowerPC604e. As a reference, our colleague, Daniel L. Rosenband, provided a hand-optimized Verilog RTL for GCD that uses only two 32-bit registers, a single subtracter, and simple boolean logic gates. The three RTL's are compiled for XC4010XL-09 FPGA using Xilinx Foundation 1.5i tools. We report the number of flip-flops and the overall utilization of the FPGA. In addition to the maximum clock frequency, we also report the number of clock cycles needed to compute GCD(53857×10957,91159×10957).

Version	FF (bit)	Util. (%)	Freq. (MHz)	Elapse (cyc)
Example 1	64	20	44.2	54
Example 2	102	38	31.5	104
Hand RTL	64	16	53.1	54

The RTL generated by TRAC from Example 2 is significantly worse than the hand-coded RTL because the input TRS maps to a sub-optimal hardware structure. TRAC does not have the same ingenuity that allowed our colleague to realize the high-level transformations that leads to the smaller and simpler circuit from Example 2. However, the necessary information to achieve the same high-level transformation can be expressed at the TRS level. Given Example 1, TRAC produces an RTL that is structurally similar to the hand-coded version and compiles to within 25% of the hand-written RTL in terms of circuit size and 17% in terms of circuit speed.

Synthesis of the Unpipelined Microprocessor: Hand-optimization can often produce much more efficient implementations than machine compilation on small designs. However, as the problem size increases, the pay back of hand optimization diminishes while the effort required increases dramatically. This is evident in the synthesis of the simple microprocessor from Example 3. The TRAC generated RTL and a hand-coded Verilog RTL of the unpipelined processor, when targeted an XC4013XL-08 FPGA, are comparable both in size and speed.

Version	FF (bit)	Util. (%)	Freq. (MHz)
Example 3	161	60 %	40.0
Hand RTL	160	50 %	41.0

We also compiled a 5-stage pipelined processor from TRS descriptions. Unfortunately, the pipelined processor requires 2.5-times more resources but runs at a lower clock frequency of 32.7 MHz. We suspect this disappointing result is due to the high routing cost in FPGA's which leads to a strong negative correlation between maximum clock rate and the size of the circuit. We don't think this problem is inherent to TRAC-generated RTL's.

5 Related Work

A *behavioral* description refers to specifying a component by its input/output behavior without implementation or structural details. In industry, such descriptions are given typically in a sequential language like the behavioral portion of Verilog. Another approach

is to extend or adapt a popular software language. Programs written in object-oriented languages like Java have been suggested as a candidate for hardware synthesis.[1] Transmorgafier-C[3] and HardwareC[8] compiles hardware from a source language based on C. Some constructs in C are overloaded to convey hardware related information such as clocking and registered storage. In the Programmable Active Memory (PAM) project, Vuillemin, et al.[10] synthesize from a RTL in C++ syntax. In the Splash 2 project, Gokhale and Minnich[4] adopt a data-parallel program language to programming an array of FPGA's. The TRS-based behavioral descriptions are different from these approaches because on one hand TRS terms convey structural information about the hardware, but on the other hand, TRS rules can embody a set of behaviors in one description. This is not possible to express in any sequential language. TRS also offer a well-understood formalism which is useful in verification.

More related to TRS are hardware description languages that have been developed in the context of formal specification and verification. TRS are perhaps closest to Lamport's TLA's. Windley[11] uses the specification language from the HOL[7] theorem proving system to describe a pipelined processor. Matthews et al.[6] have developed the Hawk language to create executable specifications of processor micro-architectures. However, none of these systems has been used in synthesis to the best of our knowledge. With a somewhat different motivation, Communicating Sequential Processes have been applied to hardware-software co-design by Gupta et al.[5] and Thomas et al.[9].

6 Conclusion

When applied in conjunction with reconfigurable technologies, TRAC can drastically lower the entry cost of taking on a hardware project by people who are not hardware designers by training. Compilers like TRAC have the potential to close the traditional distinction of hardware and software by creating a continuum of trade-off between development cost and performance. We anticipate the day when all computers are shipped with a FPGA next to the CPU, and developers are just as ready to program the FPGA for a performance critical application as they would program the processor today.

[1] Martin Rinard, Slide presentations and personal communications, 1998.

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] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, Napa, CA, April 1995.

[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] R. Gupta and G. de Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, pages 29–41, September 1993.

[6] J. Matthews, J. Launchbury, and B. Cook. Microprocessor specification in hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, Chicago, IL, 1998.

[7] SRI International, University of Cambridge. *The HOL System Tutorial, Version 2*, July 1997.

[8] Stanford University. *HardwareC - A Language for Hardware Design*, December 1990.

[9] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, pages 6–15, September 1993.

[10] 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.

[11] P. J. Windley. Verifying pipelined microprocessors. In *Proceedings of the 1995 IFIP Conference on Hardware Description Languages and their Applications (CHDL)*, 1995.