Formal Specification and Simulation of Instruction-Level Parallelism

Ed Harcourt
Dept of Computer Science
Chalmers University of Technology
Göteborg, Sweden

Jon Mauney
Dept of Computer Science
North Carolina State University
Raleigh, NC 27695

Todd Cook
Dept of Electrical and Computer Engineering
Rutgers University
Piscataway, NJ 08855

Abstract

In this paper we show how to formally specify and simulate the high-level instruction timing properties of RISC/Superscalar instruction set processors. We illustrate the technique using a hypothetical processor that includes many features of commercial processors including delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue. As our formalism we use SCCS, a synchronous process algebra designed for specifying timed, concurrent systems.

1 Introduction

In modern instruction set processors, the temporal and concurrent properties of the instructions are often visible to the user of the processor. Consequently, such properties should be included in any behavioral processor specification. We present a technique for formally describing, at a *high-level*, the timing properties of pipelined, superscalar processors. We illustrate the technique by specifying and simulating a hypothetical processor that includes many features of commercial processors, including delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue.

As our mathematical formalism, we use SCCS, a synchronous process algebra designed for specifying timed, concurrent systems [Mil83]. There are many reasons for choosing SCCS. First, SCCS allows us to *explicitly* specify the temporal and concurrent properties of a processor. Second, SCCS is formally defined and provides a variety of techniques for proving and verifying properties about SCCS descriptions. Third, there is an available tool, the Concurrency Workbench [CPS93], which allows us to interactively experiment with, analyze, and simulate our SCCS processor descriptions. Finally, SCCS allows us to describe a processor at a variety of levels of abstraction, from a high-level specification to lower organizational and implementation levels.

Our goal then is to develop a mathematical model of instruction timing that hides irrelevant detail of implementation. This research is performed in conjunction with research on designing specification languages for instruction set architectures [CFHM93, HMC94, HMC93, CH94].

2 SCCS

SCCS [Mil83], or *Synchronous Calculus of Communicating Systems*, is a mathematical theory of communicating systems in which we can represent a real system by the *terms* or *expressions* of that system. SCCS allows us to directly represent the temporal and concurrent properties of the system being specified. The syntax of SCCS processes is given by the following BNF:

$$\begin{aligned} P ::= {} & \mathbf{0} \mid \mathit{Done} \mid \alpha : P \mid P_1 + P_2 \mid P_1 \times P_2 \mid P[f] \\ & \mid P \upharpoonright S \mid P_1 \rhd P_2 \mid \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \end{aligned}$$

Intuitively, the constant process $\mathbf{0}$ is the deadlocked process, *Done* is the idle process, $\alpha : P$ means do action α and proceed with P, $A+B$ means proceed with either A or B, and $A \times B$ means execute A and B in parallel. The process $P[f]$ means execute process P but relabel the actions according to the function f, and $P \upharpoonright S$ means that, when P executes, the only visible actions are those in S. Actions are drawn from a set *Act* generated by a commutative action product operator. Each action $\alpha \in \mathit{Act}$ is either atomic or of the form $\alpha_1\alpha_2$ which means action α_1 and α_2 execute in parallel. The operator $\triangleright$ is *priority choice* and is like $+$ except that preference is given to the left operand. The semantics of SCCS is given formally in [Mil83]. Two processes communicate when one wants to execute the action α and the other wants to execute $\overline{\alpha}$. There is a predefined action 1 which is the idle action and is an identity s.t. $1\alpha = \alpha$.

3 Specifying a Processor

A processor is a system of interacting processes where registers and memory interact with one or more functional units. Before we proceed in specifying instructions and their interaction, it is necessary to develop an appropriate model of registers and memory.

In SCCS, storage cells are modeled as processes. Consider the instruction `Add` $\mathtt{R}_1$`,` $\mathtt{R}_1$`,` $\mathtt{R}_1$ which reads $\mathtt{R}_1$ twice and also writes $\mathtt{R}_1$. On most processors, this instruction *effectively* executes in a single cycle because registers are read and written in different pipeline stages. However, we do not need to model pipeline stages and the organization that goes with them; instead, our process that models a register should handle parallel reads and writes. The action $\mathtt{getr}(a)\mathtt{getr}(b)$ means read the register twice putting the value into a and b. The action $\mathtt{getr}(a)\overline{\mathtt{putr}}(b)$ means read and write the register in parallel. The action $\mathtt{getr}(a)\mathtt{getr}(b)\overline{\mathtt{putr}}(d)$ means read the register twice with the value going into a and b and write d to the register, all in parallel. Only one `putr` is allowed for each action.

Equation 1 defines a process, *Reg*, that can handle parallel reads and writes.

Register Locking — It is possible that a register is going to be updated some time in the future (*e.g.,* delayed loads), and any attempt to read or write the register by another process should result in an error. We augment Equation 1 to allow a process to reserve a register for a future update by using the action `lockreg`; then at some point in the future, the register can be written (with `putr`) and released with the action `releasereg`.

Equation 2 modifies *Reg1* so that when a process locks a register, the register goes into the state *Locked_Reg* where the only allowable action is $\overline{\mathtt{putr}}(x)\mathtt{releasereg}$. All other combinations of `getr` and `putr` in the locked state lead to the inactive process **0**, specified in *Illegal_Access*, which, for brevity, is ommitted.

Given the definition of one register, a family of registers (*Reg*$_1$, *Reg*$_2$, *etc.*) is now defined by subscripting each of the actions by a register number. For example, the action $\mathtt{putr}_i(x)$ represents writing x to register i.

The definition of a process *Memory* is exactly analogous to that of *Registers*, except that memory cells do not have locks associated with them. For brevity, we omit the definition of *Memory* and just note that the actions $\mathtt{getm}_i$ and $\mathtt{putm}_i$ read and write memory cell i.

3.1 A 32-bit RISC

To illustrate a processor specification, we present the instructions of a hypothetical 32-bit RISC. The instructions we specify are a single-cycle integer add, a delayed-load, a delayed-branch, and a multi-cycle floating-point add instruction.

Given our previous definitions of *Registers* and *Memory* and using a program counter, *PC*, we now describe a process *Instr*(PC) (Equation 4) that specifies the behavior of the instructions. *Instr*(PC) partitions instructions into two classes, *Branch* and *Non_Branch*. *Non_Branch* instructions are further partitioned into arithmetic (*Alu*), load and store (*Load_Store*), and floating-point (*Float*) instructions.

There are three possible alternatives of *Instr*(PC):

- A non-branch instruction may execute, in which case the next instruction to execute is at $PC+4$; the first line of Equation 4 describes this situation.
- A branch instruction may execute, in which case the next instruction to execute cannot be determined until it is known whether the branch will be taken or not.
- If no instruction can execute, then the processor must stall (Equation 6). The $\triangleright$ operator is used here because the processor should stall only when no other alternative is available.

3.1.1 Integer Instructions

From a user's view, the instruction `Add` $\mathtt{R}_i$`,` $\mathtt{R}_j$`,` $\mathtt{R}_k$ *appears* to take one cycle to execute, that is, behaviorally, there is no problem with writing `R1` and reading `R1` in consecutive instructions. The user does not and should not need to understand bypass hardware in order to know that the above instruction sequence is legal. The process in Equation 7 describes the integer `Add` instruction, where: at time t, source registers j and k are read and the result is written to register

$$\mathit{Reg1}(y) \overset{\text{def}}{=} \sum_{j=0}^{2} \overline{\texttt{getr}}(y)^j (1 : \mathit{Reg}(y) \; + \; \texttt{putr}(x) : \mathit{Reg}(x)) \tag{1}$$

$$\mathit{Reg}(y) \overset{\text{def}}{=} \mathit{Reg1}(y) \; + \; \sum_{j=0}^{2} \overline{\texttt{getr}}(y)^j \texttt{lockreg} : \mathit{Locked\_Reg}(y) \tag{2}$$

$$\mathit{Locked\_Reg}(y) \overset{\text{def}}{=} \mathit{Illegal\_Access}(y) \; + \; \texttt{putr}(x)\texttt{releasereg} : \mathit{Reg}(x) \; + \; 1 : \mathit{Locked\_Reg}(y) \tag{3}$$

Figure 1: Specification of integer registers.

i. The process *Done* is the idle process and represents termination of the instruction.

The Delayed-Load Instruction — Our example RISC has a delayed-load instruction in which it is illegal for the instruction executing immediate after the load to use the register being loaded. The `Load` instruction accesses memory at time t, and the result of the load is available at time $t + 2$. This is represented in Equation 8 which specifies that at time t, three things happen: *1)* the base register j is accessed, and the base address is placed in the variable B; *2)* memory is accessed with the value placed in the variable V; and *3)* the destination register i is locked. At time $t + 1$, two actions occur: *1)* V is written to destination register i; and *2)* the destination register i is released. (The store operation is analogous and, for lack of space, has been omitted; see [Har94])

The Delayed-Branch Instruction — In a *delayed-branch* instruction the instruction after the branch is always executed before the jump. If the branch is not taken, then the instruction after the branch is skipped. An additional constraint is that a delayed-branch instruction may not be immediately followed by another delayed-branch instruction. Equation 9 specifies the behavior of the `BZ` instruction which has the effect that

- at time t, a `BZ` instruction is fetched and register $\texttt{R}_i$ is accessed.
- at time $t + 1$, if $\texttt{R}_i$ is not zero, then execution continues with the instruction after the branch delay slot.
- at time $t + 1$, if $\texttt{R}_i$ is zero, then a *non-branch* instruction is executed in the branch delay slot and execution continues with the instruction at *Locn* at time $t + 2$. However, if another `BZ` instruction is in the delay slot, then we reach the inactive process **0**, which represents an error state.

3.1.2 Floating-Point Instructions

The floating-point add instruction, `Fadd`, takes six cycles to compute its result. Instructions that have a large latency are typically interlocked, so we need to define interlocked floating-point registers.

One method of keeping instructions ordered properly is to associate a "lock" with each FP-register (as we did in the case of the integer registers). The difference is that accessing a locked integer register is illegal, and accessing a locked FP-register causes the processor to stall.

Our RISC has a separate set of thirty-two floating-point registers that are defined similarly to the integer registers, except that we add two new actions, `lockfreg` and `releasefreg`. Actions `putfr` and `getfr` are the two actions that write and read a floating-point register. Since the definitions are similar to the integer registers, we omit them here; their complete definition may be found in [Har94].

The `Fadd` instruction Now that interlocked registers are defined, we define the behavior of the floating point add instruction in Equation 10. The `Fadd` instruction: *1)* simultaneously accesses its source registers and locks its destination register; *2)* computes the addition; and *3)* simultaneously writes the result in the destination register and unlocks the destination register. The abbreviation $(1)^n$ represents an n-cycle delay, $1 : 1 \ldots 1$, which is interpreted as n-cycles of internal computation.

The processor stalls when an instruction wishes to access a locked FP-register because the instruction will not be able to access the FP-register. The only other option is to execute the process *Stall* (Equation 4).

$$\begin{aligned}
\mathit{Instr}(PC) &\stackrel{\text{def}}{=} (\mathit{Non\_Branch}(PC)\ \mathit{Next}\ \ \mathit{Instr}(PC+4)) \\
&+\ \mathit{Branch}(PC) \\
&\rhd\ \mathit{Stall}(PC) && (4) \\
\mathit{Non\_Branch}(PC) &\stackrel{\text{def}}{=} \mathit{Alu}(PC) + \mathit{Load\_Store}(PC) + \mathit{Float}(PC) && (5) \\
\mathit{Stall}(PC) &\stackrel{\text{def}}{=} 1 : \mathit{Instr}(PC) && (6) \\
\mathit{Alu}(PC) &\stackrel{\text{def}}{=} \mathtt{getm}_{\mathrm{PC}}(\mathtt{Add\ R}_i\mathtt{,R}_j\mathtt{,R}_k)\mathtt{getr}_j(x)\mathtt{getr}_k(y)\overline{\mathtt{putr}_i}(x+y) : \mathit{Done} && (7)
\end{aligned}$$

$$\begin{aligned}
&\mathit{Load\_Store}(PC) \stackrel{\text{def}}{=} \\
&\quad \mathtt{getm}_{\mathrm{PC}}(\mathtt{Load\ R}_i\mathtt{,\ R}_j\mathtt{,\ offs})\mathtt{getr}_j(B)\mathtt{getm}_{\mathrm{B+offs}}(V)\overline{\mathtt{lockreg}_i} : \overline{\mathtt{putr}_i}(V)\overline{\mathtt{releasereg}_i} : \mathit{Done} && (8)
\end{aligned}$$

$$\begin{aligned}
\mathit{Branch}(PC) \stackrel{\text{def}}{=}\ & \mathtt{getm}_{\mathrm{PC}}(\mathtt{BZ\ R}_i\mathtt{,}\ \mathit{Locn})\mathtt{getr}_i(V) : \\
& \quad \textbf{if } V = 0 \textbf{ then} \\
& \qquad \mathit{Non\_Branch}(PC+4)\ \mathit{Next}\ \ \mathit{Instr}(\mathit{Locn})) \\
& \qquad +\ \mathtt{getm}_{\mathrm{PC+4}}(\mathtt{BZ\ R}_i\mathtt{,}\ \mathit{Locn}) : \mathbf{0} \\
& \quad \textbf{else} \\
& \qquad \mathit{Instr}(PC+8) && (9)
\end{aligned}$$

$$\begin{aligned}
\mathit{Float}(PC) \stackrel{\text{def}}{=}\ & \mathtt{getm}_{\mathrm{PC}}(\mathtt{Fadd,\ FR}_i\mathtt{,\ FR}_j\mathtt{,\ FR}_k)\overline{\mathtt{lockfreg}_i}\mathtt{getfr}_j(x)\mathtt{getfr}_k(y) : \\
& (1\ :)^5 \\
& \overline{\mathtt{putfr}_i}(x+y)\overline{\mathtt{releasefreg}_i} : \mathit{Done} && (10)
\end{aligned}$$

Figure 2: SCCS description of the 32-bit RISC

Resource Constraints — Most floating-point units have a finite set of resources (*e.g.*, adder, multiplier, etc.) and two or more instructions can compete for these resources thereby altering an instruction's timing behavior. We can model a resource as a lockable process and include resource requirements in our instruction specification. For example, we can specify in the `Fadd` instruction above that it uses the floating-point adder for several consecutive cycles. (Other instructions may require the floating-point adder and have to wait if it is in use.) Modeling resources is described in [Har94].

4 A Two-Issue Superscalar

This section describes a superscalar version of our RISC that can execute one floating-point and one integer instruction per cycle. If two instructions can be executed in parallel, then we either have an integer instruction followed by a floating point instruction or a floating-point instruction followed by an integer instruction. This situation is specified by Equation 11

$$\begin{aligned}
& (\mathit{Float}(PC) \times \mathit{Alu}(PC+4)) \\
+\ & (\mathit{Alu}(PC) \times \mathit{Float}(PC+4)) \qquad (11)
\end{aligned}$$

which we rewrite using a summation notation in Equation 12 and represents a folding of the expression above. We use the summation notation because it enables us to succinctly specify n-way instruction parallelism. Equation 12 also specifies that execution proceeds at $PC+8$.

There are no data dependencies to worry about because each instruction accesses separate register files.

4.1 Instruction Issue

Our top-level instruction issue equation (Equation 4) must now be modified to take this new two-issue

$$\begin{aligned} \mathit{Do\_Two}(PC) &\stackrel{\text{def}}{=} \left(\sum_{i,j \in \{0,4\}} (\mathit{Alu}(PC+i) \times \mathit{Float}(PC+j)) \right) \ \mathit{Next} \ \ \mathit{Instr}(PC+8) && (12) \\ \mathit{Do\_One}(PC) &\stackrel{\text{def}}{=} (\mathit{Non\_Branch}(PC) \ \mathit{Next} \ \ \mathit{Instr}(PC+4)) + \mathit{Branch}(PC) && (13) \\ \mathit{Instr}(PC) &\stackrel{\text{def}}{=} \mathit{Do\_Two}(PC) \rhd \mathit{Do\_One}(PC) \rhd \mathit{Stall}(PC) && (14) \end{aligned}$$

Figure 3: Dual issue superscalar specification.

capability into account. We rename our top-level instruction-issue process *Instr* (Equation 4), to *Do_One* in Equation 13. The processor can now execute two, one, or zero (*i.e.,* stall) instruction(s) per cycle, which we capture in Equation 14. Notice the use of the priority choice operator, $\rhd$ instead of $+$; whenever it is possible to do *Do_Two*, it is also possible to do *Do_One*, and issuing two instructions should take priority over issuing one when possible.

We can also model multiple instruction issue even when the instructions have data dependencies between them. This is done by using the restriction operator, $\uparrow$, to allow or disallow certain instruction sequences. Again, we do not have room to specify this here, but the details may be found in [Har94].

5 Simulation

In this section, we describe how our SCCS processor specification is simulated within the framework of the Concurrency Workbench [CPS93], a tool which allows us to experiment with, simulate, and analyze SCCS specifications.

A simulation of a processor specification amounts to loading a program into memory (with `putm` actions) and then running the process that represents the processor. The semantics of SCCS is operational and defined in terms of an abstract machine called a *labeled transition system.* The Concurrency Workbench calculates the labeled transition system of a process, allowing us to observe the behavior of the program.

The *transition graph* of a process P is comprised of transitions of the form $A \xrightarrow{\mathsf{a}} B$, where

- A represents the state of the system, at time t;
- a is the action performed (transition); and
- B is the new state at time $t+1$.

In our transition graphs, each node is surrounded by a box, and represents the current state of the processor at a particular moment in time. Each edge is labeled with the set of actions that execute on that transition (*e.g.,* instructions, **getr**, **putr**, **lockreg**, etc.). For readability, individual actions are enclosed with "[ ]" (*e.g.,* $[\mathtt{getr}_1(x)][\overline{\mathtt{putr}}_2(x)]$). To simplify the graph, many actions and processes have been omited. In a complete graph, each particle on an edge that represents reading/writing or locking/releasing a register would have a corresponding complementary action. Figure 4 shows the transition graph of an integer add instruction followed by a move instruction.

When executing an illegal instruction sequence, $i_1 \cdots i_n$, we reach the error state $\mathbf{0}$. Here, the transition graph is

$$\sigma \xrightarrow{i_1} \cdots \xrightarrow{i_n} \cdots \longrightarrow \mathbf{0}$$

for some state σ.

If two instructions, i and j, can execute in parallel then an edge in the transition graph contains the action product $i \cdot j$. In terms of the transition graph then, $\sigma' \xrightarrow{i \cdot j} \sigma''$ for states σ' and σ''.

6 Conclusions

In this paper we have presented a technique for formally specifying the timing properties of instruction-level parallel processors using SCCS, a synchronous process calculus. The timing properties specified are delayed loads and branches, interlocked floating-point operations, and multiple instruction issue.

We have also shown how we can simulate our processor using the Concurrency Workbench. The transition graphs of the preceding section are precisely what the workbench produces. From these transition graphs we can deduce illegal instruction sequences. Also, by observing when an instruction starts and completes, the transition graphs yield information

$$t\text{:} \quad \boxed{\mathit{Instr}(PC) \times \mathit{Registers} \times \mathit{Memory} \times \mathit{FP\_Registers}}$$

$$[\texttt{Add R}_2\texttt{, R}_2\texttt{, R}_3]\ [\texttt{getr}_2(x)]\ [\texttt{getr}_3(y)]\ [\overline{\texttt{putr}}_2(x+y)]$$

$$t+1\text{:} \quad \boxed{\mathit{Instr}(PC+4) \times \mathit{Registers} \times \mathit{Memory} \times \mathit{FP\_Registers}}$$

$$[\texttt{Mov R}_2\texttt{, R}_1]\ [\texttt{getr}_1(x)]\ [\overline{\texttt{putr}}_2(x)]$$

$$t+2\text{:} \quad \boxed{\mathit{Instr}(PC+8) \times \mathit{Registers} \times \mathit{Memory} \times \mathit{FP\_Registers}}$$

Figure 4: Transition graph of `Add` followed by a `Move`.

about instruction latencies. We are also able to automatically derive instruction scheduling parameters from the transition graphs.

Another benefit of using SCCS (and the Workbench) is that we can also verify that our specification has some important properties. For example, we can verify that there exists mutual exclusion on the registers (both integer and floating-point). This is done by formulating the mutual exclusion property in a temporal logic (the Workbench also provides such a logic) and showing that the SCCS specification *satisfies* the logic expression.

References

[CFHM93] Todd A. Cook, Paul D. Franzon, Ed A. Harcourt, and Thomas K. Miller. System-level specification of instruction sets. In *ICCD 93, Proceedings of the International Conference on Computer Design*, 1993.

[CH94] Todd A. Cook and Ed Harcourt. A functional specification language for instruction set architectures. In *ICCL: Proceedings of the International Conference on Computer Languages*, 1994.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[Har94] Edwin A. Harcourt. *The Formal Specification of Instruction Set Processors and the Derivation of Instruction Schedulers.* PhD thesis, North Carolina State University, Raleigh, NC, 1994. Department of Computer Science.

[HMC93] Ed Harcourt, Jon Mauney, and Todd Cook. Specification of instruction-level parallelism. In *Proceedings of NAPAW'93, the North American Process Algebra Workshop*, 1993.

[HMC94] Ed Harcourt, Jon Mauney, and Todd Cook. Functional specification and simulation of instruction set architectures. In *Proceedings of the International Conference on Simulation and Hardware Description Languages.* SCS Press, 1994.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Journal of Theoretical Computer Science*, 25:267–310, 1983.