Specification of Instruction-Level Parallelism

Ed Harcourt* Jon Mauney* Todd Cook†

*Department of Computer Science

†Department of Electrical and Computer Engineering

North Carolina State University

Raleigh, NC 27695

July 15, 1993

Abstract

We present a technique for formally describing, at a high-level, the timing properties of Superscalar/RISC instruction set processors. We illustrate the technique by specifying a hypothetical processor that shares many properties of commercial processors including delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue (Superscalar). As our mathematical formalism we use SCCS, a synchronous process algebra used for specifying timed concurrent systems. Timing properties are specified at an abstract level without resorting to implementation detail. Such high-level specifications are useful for timing-level simulator generation, synthesis and verification of hardware and software (*e.g.* compilers, schedulers), and precise documentation. We have implemented our specification within the framework of the Concurrency Workbench, a tool for simulating and analyzing SCCS specifications.

1 Introduction

In modern computer architecture 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 architecture specification. We present a technique for formally describing, at a *high-level*, the timing properties of Superscalar/RISC instruction set processors [PH90, Joh91]. We illustrate the technique by specifying a hypothetical RISC that shares many properties of

commercial RISCs including delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue (Superscalar).

As our mathematical formalism we use SCCS, a synchronous process algebra designed for specifying timed concurrent systems [Mil89, Mil83]. SCCS allows us to *explicitly* specify the temporal and concurrent properties of a processor (and instructions). In contrast, a functional approach only allows us to specify final computations [Pai90]. We have implemented our specification on the Concurrency Workbench [CPS93, Mol92] allowing us to interactively experiment with, analyze, and simulate our processor description. This research is in conjunction with research to design a specification language for instruction set architecture [CFHM93].

1.1 Levels of Abstraction

There are many views of an instruction set processor — a common hierarchy is:

- The *architecture* level is a functional view that represents the processor as seen by the assembly language programmer.
- The *organization* level includes the general structure of the processor in terms of functional units (*e.g.,* integer and floating-point pipelines, caches, and busses).
- The *logic* level contains the low level implementation detail of the functional units.

The user of a processor is concerned with the architectural level, as they must have information to write correct programs. However the user would also like to use the processor most efficiently. For example, in some RISC architectures the following instruction sequence may possibly be programmed more efficiently.

```
(1)     Load R1, (R2)              ;R1 ← Mem[R2]
(2)     Add  R2, R2, R1            ;R2 ← R2 + R1
(3)     Add  R3, R3, #1            ;R3 ← R3 + 1
```

Instruction (2) will usually cause an interlock (on the MIPS this is an incorrect program) which wastes cycles. However, instructions (2) and (3) may be switched without altering the meaning of the program. This switch would most likely eliminate the interlock caused by (2).

There is no hard line that determines where one processor view ends and another begins. Usually the architecture level does not contain timing information and the organization level does. But the organization level also contains a considerable amount of other detail that is of no concern to the user.

A motivating example comes from the MIPS manual which states that for the load-word instruction, `LW rt, offset(base)`, "... *the contents of general register* `rt` *are undefined for time T of the instruction immediately following this load instruction*" [KH92]. Such informal descriptions are vague and imprecise and demonstrate the kind of timing constraint we wish to formalize at an abstract level — that is, hides organization detail. There may be any number of reasons for the delay in the load instruction and we only wish to specify the delay and not the underlying cause. The user should not be expected to infer the delay by studying low-level organization.

The goal of this research, then, is to develop a mathematical model of instruction timing at an abstract level that hides irrelevant detail of organization.

1.2 Extensions to SCCS

It is assumed that the reader is familiar with SCCS as presented in [Mil83, Mil89]. We use two extensions to SCCS that will aid us in writing processor specifications.

Frequently we wish to execute two agents A and B in parallel where B begins executing one clock cycle after A (*e.g.*, issuing instructions on consecutive cycles). This is modeled by the agent $A \times 1 : B$ and the expression A *Next* B denotes this agent.

Another useful operator is the *priority sum* operator, $\triangleright$ [CW91]. If in $A \triangleright B$ both A and B can execute then A is preferred.

2 Specifying a Processor

A processor is a system of interacting processes where registers and memory interact with one or more functional units. Equation 1 represents such a system at the highest level.

$$\begin{aligned} \mathit{Processor} &\stackrel{\text{def}}{=} (\mathit{Instruction\ Unit} \times \mathit{Memory} \times \mathit{Registers}) \uparrow I \qquad (1) \\ \textbf{where } I &= \{\text{Instructions from section 2.4.}\} \end{aligned}$$

Before we proceed in specifying instructions and their interaction it is necessary to develop an appropriate model of registers and memory.

2.1 Defining the Registers

In this section we develop an abstract model of storage in which storage cells are modeled as agents. The agent *Reg1*(y) defines one register holding a value y, such that an action $\mathtt{putr}(x)$ executed at time t stores x in the register which is available for use at time $t+1$. The action $\overline{\mathtt{getr}}(y)$ retrieves the value stored in the register and assigns this to y. Another action, a product of two particulate actions, $\overline{\mathtt{putr}}(x)\mathtt{getr}(y)$, allows *Reg1* to be read and written simultaneously. The value read is the old value in *Reg1* not the new one being written. Since reading a register does not alter its value, multiple **getr** actions are allowed on the same register. The action $\mathtt{getr}(y)\mathtt{getr}(y)$ represents reading the register twice, which we abbreviate to $\mathtt{getr}(y)^2$.

$$Reg1(y) \stackrel{\text{def}}{=} \sum_{j\in\{1,2\}} \overline{\mathtt{getr}}(y)^j : Reg(y) + \sum_{j\in\{0,1,2\}} \overline{\mathtt{getr}}(y)^j\mathtt{putr}(x) : Reg(x) + 1 : Reg(y) \quad (2)$$

2.1.1 Register Locking

The actions **getr** and **putr** are atomic. It may be 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 agent should result in an error. We augment equation 2 by allowing an agent to reserve a register for future writing using the action **lockreg** and then, at some point in the future, by writing the register (with **putr**) and releasing it with the action **releasereg**. When an agent locks a register the register goes into a 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 agent **0**. This need to trap all of the other illegal action sequences complicates matters so we have factored them into equation 4.

$$\begin{aligned} Reg(y) &\stackrel{\text{def}}{=} Reg1(y) + \mathtt{lockreg} : Locked\_Reg(y) \\ Locked\_Reg(y) &\stackrel{\text{def}}{=} Illegal\_Access(y) + \mathtt{putr}(x)\mathtt{releasereg} : Reg(x) \\ &+ 1 : Locked\_Reg(y) \end{aligned} \quad (3)$$

$$Illegal\_Access(y) \stackrel{\text{def}}{=} \sum_{j\in\{1,2\}} \overline{\texttt{getr}}(y)^j : \mathbf{0} + \sum_{j\in\{0,1,2\}} \overline{\texttt{getr}}(y)^j \texttt{putr}(x) : \mathbf{0}$$
$$+ \sum_{j\in\{1,2\}} \overline{\texttt{getr}}(y)^j \texttt{putr}(x) \texttt{releasereg} : \mathbf{0} \quad (4)$$

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 $\texttt{putr}_i$ represents writing register i. Thirty-two registers are constructed by

$$Registers \stackrel{\text{def}}{=} \prod_{0}^{31} Reg_i(y) \quad (5)$$

Notice that when no **putr** or $\overline{\texttt{getr}}$ action is requested the registers are idling.

2.2 Defining Memory

Given the definition of the register *Reg1* (non-locking version), a similar definition of an agent *Memory* is straightforward. Analogously, actions **getm** and **putm** read and write memory cells and the agent *Memory* is defined to be a product of individual memory cells.

2.3 Instruction Pipeline

Instruction pipelines are usually described in terms of its stages of execution, for example: fetch, decode, execute, memory access, write back (abbreviated IF, ID, EX, MEM, WB). *IPL* (for instruction pipeline) defines a model of an instruction pipeline.

$$IPL \stackrel{\text{def}}{=} IF \times ID \times EX \times MEM \times WB$$

This is a reasonable and obvious representation, but if we are interested only in *external behavior* it is over specified. We should resist attempting to specify an architecture's timing behavior in terms of individual stages as this commits us to describe the functionality of each individual stage which would have to include, for example, forwarding hardware and latches. We should strive for a more abstract specification.

2.4 ToyP, a Toy Processer

To construct a specification of a processor we present the instructions of a hypothetical RISC, ToyP, that shares many features of commercial RISCs. ToyP is loosely based on the MIPS architecture [KH92]. ToyP instructions, memory word size, registers, and addresses are thirty two bits. ToyP is a Load/Store architecture with three-operand arithmetic instructions.

Here is an informal description of the semantics and timing behavior of some ToyP instructions.

- **Add** R_i, R_j, R_k adds registers j and k and puts the result in register i. The instruction executing immediately after an **Add** may use register i.
- **Load** R_i, R_j, **#Const** is a delayed load instruction. Register i is being loaded from memory at the base address in register j with offset **#Const**. The instruction executing immediately after **Load** cannot use register i.
- **BZ** R_i, **#Locn** is a delayed branch instruction. The instruction immediately after the branch is always executed before the branch is taken (*if* $R_i = 0$). If the branch is not taken then instruction after the branch is not executed. Another **BZ** instruction may not appear in the branch delay slot.
- **Fadd** FR_i, FR_j, FR_k is an interlocked floating-point add that takes six cycles before the result can be used. If another **Fadd** instruction tries to use the result before the current **Fadd** is finished then instruction execution stalls until the result is ready.

2.5 Instruction Issue

Given our definitions of *Registers* and *Memory* we now describe an agent $Instr(PC)$ (equation 6) that specifies the behavior of ToyP instructions off of program counter PC. Instructions are divided into four classes: arithmetic, load and store, branch, and floating-point and are described by agents *Alu*, *Load_Store*,*Branch*, and *Float*.

$$\begin{aligned} Instr(PC) &\stackrel{\text{def}}{=} (Non\_Branch(PC)\ Next\ \ Instr(PC+4)) \\ &+ \ Branch(PC) \rhd Stall(PC) \qquad (6) \\ Non\_Branch(PC) &\stackrel{\text{def}}{=} Alu(PC) + Load\_Store(PC) + Float(PC) \qquad (7) \\ Stall(PC) &\stackrel{\text{def}}{=} 1 : Instr(PC) \qquad (8) \end{aligned}$$

There are three possible execution paths of $Instr(PC)$.

- A non-branch instruction executes and the next instruction to execute is at $PC + 4$.
- A branch instruction may execute. Here, the decision on what instruction to execute next is deferred.
- If no instruction can execute then the processor must stall. The $\triangleright$ operator (section 1.2) is used here because the processor should stall only when no other alternative is available.

2.5.1 Arithmetic Instructions

All Von Neumann architectures are based on the "stored program model" and fetch instructions from memory using a program counter which we call, PC. The action

$$\mathtt{getm}_{\mathtt{PC}}(\mathtt{Add\ R}_i\mathtt{,R}_j\mathtt{,R}_k)$$

represents fetching an **Add** instruction from memory. And in fact, from a user's view, an instruction **Add** $\mathtt{R}_i$, $\mathtt{R}_j$, $\mathtt{R}_k$ *appears* to take one cycle to execute. In the following instruction sequence,

```
Add R1, R2, R3
Mov R2, R1
```

the **Add** instruction executes at time t and the **Mov** executes at time $t + 1$. From a behavioral view 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 discover that the above instruction sequence is legal.

The agent

$$Alu(PC) \stackrel{\text{def}}{=} \mathtt{getm}_{\mathtt{PC}}(\mathtt{Add\ R}_i\mathtt{,R}_j\mathtt{,R}_k)\mathtt{getr}_j(x)\mathtt{getr}_k(y)\overline{\mathtt{putr}_i}(x+y) : DONE$$

represents the execution of the **Add** instruction specifying that registers are accessed and the result is written atomically. The agent $DONE$ is the idle agent and has the effect of representing termination of the instruction. (In [Mil83, Mil89] the idle agent is called **1**. We use $DONE$ to avoid confusion with the idle action **1**.)

2.5.2 Load and Store Instructions

The following instruction sequence,

```
Load R1, R2, #8
Mov  R3, R1
```

is illegal in ToyP because of the use of R1 immediately after the Load. The Load instruction accesses memory at time t and the result of the load is available at time $t + 2$. This is represented by,

$$Load\_Store(PC) \stackrel{\text{def}}{=}$$

$$\texttt{getm}_{\text{PC}}(\texttt{Load R}_i\texttt{, R}_j\texttt{, }\Delta)\texttt{getr}_j(B)\texttt{getm}_{\text{B}+\Delta}(V)\overline{\texttt{lockreg}_i} : \overline{\texttt{putr}_i}(V)\overline{\texttt{releasereg}_i} : DONE$$

The Store instruction is similarly defined except that the result is ready immediately (presumably because of forwarding hardware).

2.5.3 The Branch Instruction

Equation 9 specifies the behavior of the delayed branch instruction, BZ.

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

The BZ instruction has the effect that

- at time t, a BZ instruction is fetched and register R_i is accessed.
- at time $t + 1$, if the value of R_i is not zero then execution continues with the instruction after the branch delay slot.
- at time $t + 1$, if the value of 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 $t + 2$.

- If another BZ instruction is in the delay slot then we reach the inactive agent **0**, which represents an error state.

2.6 Interlocked Floating-Point Instructions

The floating-point add instruction **Fadd** takes six cycles to compute its result. For instructions that have a large latency it is generally unreasonable to expect the programmer (or scheduler) to find enough independent instructions to execute until the **Fadd** is complete. Inserting **Nop** instructions would significantly increase code size, therefore, floating-point instructions are typically interlocked.

2.6.1 Floating-Point Registers

We associate a lock with each floating-point register as in the integer registers. The difference though is that an attempt to read or write an integer register while it is locked is illegal while reading or writing a floating-point register while it is locked causes the processor to stall. ToyP 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.

$$Freg_i(y) \stackrel{\text{def}}{=} \mathtt{lockfreg}_i : Write_i + \sum_{j\in\{1,2\}} \overline{\mathtt{getfr}}_i(y)^j : Freg_i(y) + 1 : Freg_i(y)$$

$$Write_i \stackrel{\text{def}}{=} \mathtt{putfr}_i(x)\mathtt{releasefreg}_i : Freg_i(x) + 1 : Write_i$$

2.6.2 The Fadd instruction

Having defined interlocked fp-registers we can specify the behavior of the **Fadd** instruction. The **Fadd** instruction must (1) access its source registers and lock its destination register using the action $\overline{\mathtt{lockreg}}$; (2) compute the addition; (3) write the result in the destination register and release the destination register using the action $\overline{\mathtt{releasereg}}$. Equation 10 specifies ToyP's **Fadd** instruction.

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

The processor stalls when an instruction wishes to access a locked fp-register. Since the action will not be available the execution of the instruction is suppressed and the only available action

then is to execute the idle action of agent *Stall.*

3 A Two-Issue Superscalar ToyP

This section describes variations of ToyP that can issue and execute multiple instructions per cycle. Such *multiple issue* processors are commonly referred to as *superscalar* processors. Two or more instructions can be executed in parallel if they are data independent and can also be issued to separate functional units.

Usually, one floating-point and one integer instruction can be issued in parallel as they use separate functional units. Also, if they use disjoint register files, they are guaranteed to be data independent. Two integer instructions can be issued in parallel only if there are two or more integer functional units. In this case, the problem is complicated somewhat because data dependencies between two integer instructions can arise inhibiting their parallel execution.

3.1 A Float $\times$ Integer Superscalar

In this variant of ToyP one integer and one floating point instruction can be issued in parallel. Branch and Load/Store instructions must be issued sequentially.

If two instructions can be issued in parallel then it must be an integer instruction followed by a floating point instruction or vice-versa.

$$(Float(PC) \times Alu(PC+4)) + (Alu(PC) \times Float(PC+4))$$

We can rewrite this using summation and also to include continuing execution at $PC+8$.

$$Do\_Two(PC) \stackrel{\text{def}}{=} \left(\sum_{i,j \in \{0,4\}} (Alu(PC+i) \times Float(PC+j)) \right) \; Next \;\; Instr(PC+8)$$

Do_Two assumes, justifiably, that an instruction cannot be both an integer and a floating-point instruction. This implies that when $i = j$ the agent $Alu(PC+i) \times Float(PC+j)$ reduces to zero. There are no data dependencies to worry about because each instruction accesses separate register files.

3.1.1 Instruction Issue

Our top-level instruction issue equation 6 must be modified to reflect this new two-issue capability. Renaming equation 6 from *Instr* to *Do_One* our processor can now execute two, one, or zero (*i.e.*, stall) instruction(s) per cycle which we capture in equation 11.

$$Instr(PC) \stackrel{\text{def}}{=} Do\_Two(PC) \rhd Do\_One(PC) \rhd Stall(PC) \tag{11}$$

Notice the use of $\rhd$ (section 1.2) 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.

3.2 An Integer $\times$ Integer Superscalar

In this section we specify a version of ToyP that can execute two integer ALU instructions in parallel. At first glance, it would seem that

$$Alu(PC) \times Alu(PC+4) \tag{12}$$

specifies the ability to execute two integer instructions in parallel. However, because both instructions use the same register file we now have the possibility of data hazards existing between the two integer instructions. Hence, sometimes parallel execution is thwarted.

Using particle restriction ($A\backslash\backslash S$ where S is a set of particles) we can force equation 12 to apply only to legal integer instruction sequences of length two. In a legal instruction sequence the first integer instruction can write register i and the second integer instruction cannot write or read register i.

$$\begin{aligned} \sum_{i=0}^{31} (Alu(PC)\backslash\backslash A \quad &\times \quad Alu(PC+4)\backslash\backslash B) \\ \textbf{where} \qquad & A = \{\mathtt{putr}_i, \mathtt{getr}_0 \ldots \mathtt{getr}_{31}\} \\ & B = \{\mathtt{getr}_0 \ldots \mathtt{getr}_{31}, \mathtt{putr}_0 \ldots \mathtt{putr}_{31}\} - \{\mathtt{putr}_i, \mathtt{getr}_i\} \end{aligned} \tag{13}$$

Equation 13 represents all of the allowable integer instruction sequences of length two that may execute in parallel.

4 Simulation

A simulation of our ToyP specification amounts to running an agent that represents a ToyP program with our agent that represents ToyP. That is, $ToyP \times Program$. We observe the behavior of the program by calculating the transition graph of an agent. We do not have room to reproduce a transition graph here. We only note that our simulation takes place within the framework of the Concurrency Workbench which allows us to experiment with our processor specification.

5 Conclusions and Comments

In this paper we have presented a technique for 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.

With the plethora of formal specification languages (especially for hardware), the question arises why we should prefer our approach to another. The answer, as is often the case, depends on what one is interested in, and is, in our case, *explicit specification of both timing and concurrency.* The reason that this is important, is that, if the programmer wants to use the processor the most efficiently, timing and concurrency must be specified. Our SCCS processor description specifies a processor for what it really is; a communicating system of functional units.

In future research, we plan to derive instruction scheduling parameters (*e.g.,* latencies) from processor specifications allowing us to automatically synthesize instruction schedulers. The transition graphs of instructions clearly indicate when instructions begin executing. Latency information is derived by appropriately testing the specification and observing when instructions begin execution.

References

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

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffan. 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.

[CW91] Juanito Camilleri and Glynn Winskel. CCS with priority choice. In *LICS 91: IEEE Symposium on Logic in Computer Science*, pages 246–255, 1991.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1991.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

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

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mol92] Faron Moller. *The Edinburgh Concurrency Workbench (Version 6.1).* University of Edinburgh, 1992.

[Pai90] Jean-Luc Paillet. Functional semantics of microprocessors at the machine instruction level. In *Computer Hardware Description Languages and Their Applications*, pages 87–101, 1990.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, San Mateo, CA, 1990.