High-Level Timing Specification of Instruction-Level Parallel Processors

Ed Harcourt[1] Jon Mauney[1] Todd Cook[2]

[1] Department of Computer Science

[2] Department of Electrical and Computer Engineering

North Carolina State University

Raleigh, NC 27695

NCSU Tech Report TR-93-18

August 24, 1993

Abstract

In modern instruction set processors, the temporal and concurrent properties of the instructions are often visible to the user of the processor. To use the processor as efficiently as possible, the user needs this information. Consequently, this *instruction-level parallelism* 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.

Formal specifications are a fundemental and logical starting point for solving a variety of problems, including: verification, simulation, synthesis, and precise documentation. In addition, a formal specification aids in the design process as it requires a designer to rigorously and thoughtfully plan the design in a structured manner.

Contents

List of Figures

Chapter 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 [PH90, Joh91, HMC93]. 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 [Mil89, Mil83].

There are many reasons we have chosen SCCS as our formalism. First, SCCS allows us to *explicitly* specify the temporal and concurrent properties of a processor. Second, SCCS is formally defined and has a variety of proof techniques available that allow us to prove and verify 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. Fourth, 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.

1.1 Levels of Abstraction

There are many views of an instruction set processor [CFHM93] — a common hierarchy is:

- The *architecture* level is a functional view that represents the processor as seen by the assembly language programmer (or compiler writer). This view only includes information needed to write functionally correct programs.
- The *organization* level includes the general structure of the processor in terms of functional units which include include integer and floating point pipelines, branch units, caches, and busses.
- The *logic* level contains the low level implementation details of the functional units.

The user of a processor is usually concerned with the architectural level, since the user *must* have this information to write correct programs. However the user would also like to use the processor efficiently. For example, in some RISC architectures the following instruction sequence is not the most efficient.

```
(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, which wastes cycles and the pipeline will stall. However, instructions (2) and (3) may be switched without altering the meaning of the program, and this switch would reduce the number of stall cycles.

1.2 Organization Detail vs. Timing Detail

There is no hard line that separates one level of abstraction from another. Usually the architecture level does not contain timing information but the organization level does. However, the organization level also contains a considerable amount of other detail that is of no concern to the user.

For example, a floating-point multiply may have a latency of six cycles due to the structure of the pipeline. However, to use the multiplication instruction efficiently we need

only be concerned with the latency itself, not the cause. In general, we should not expect that the user of the processor infer the existence of the timing constraints by examining the organization.

Our goal then is to develop a mathematical model of instruction timing that hides irrelevant detail of implementation.

1.3 Motivation

Formal processor specifications are useful and sometimes necessary for many reasons.

- Processor verification requires a formal specification in order to carry out proofs of correctness.
- If the specification language is executable (and SCCS is) then a timing level simulator is automatically available.
- High-level synthesis requires some form of a specification.
- Formal specifications are used for precise documentation.

Currently, we are researching deriving instruction scheduling parameters from our processor specification thereby automatically generating an instruction scheduler for the processor. Figure 1.1 shows a rather canonical view of the phases of a compiler and where our timing specification fits in.

1.4 Outline of Paper

The rest of this paper is organized as follows:

- Chapter 2 briefly reviews related research.
- Chapter 3 gives an introduction to SCCS.
- Chapter 4 introduces, informally, ToyP, a typical RISC machine.
- Chapter 5 specifies ToyP with SCCS.

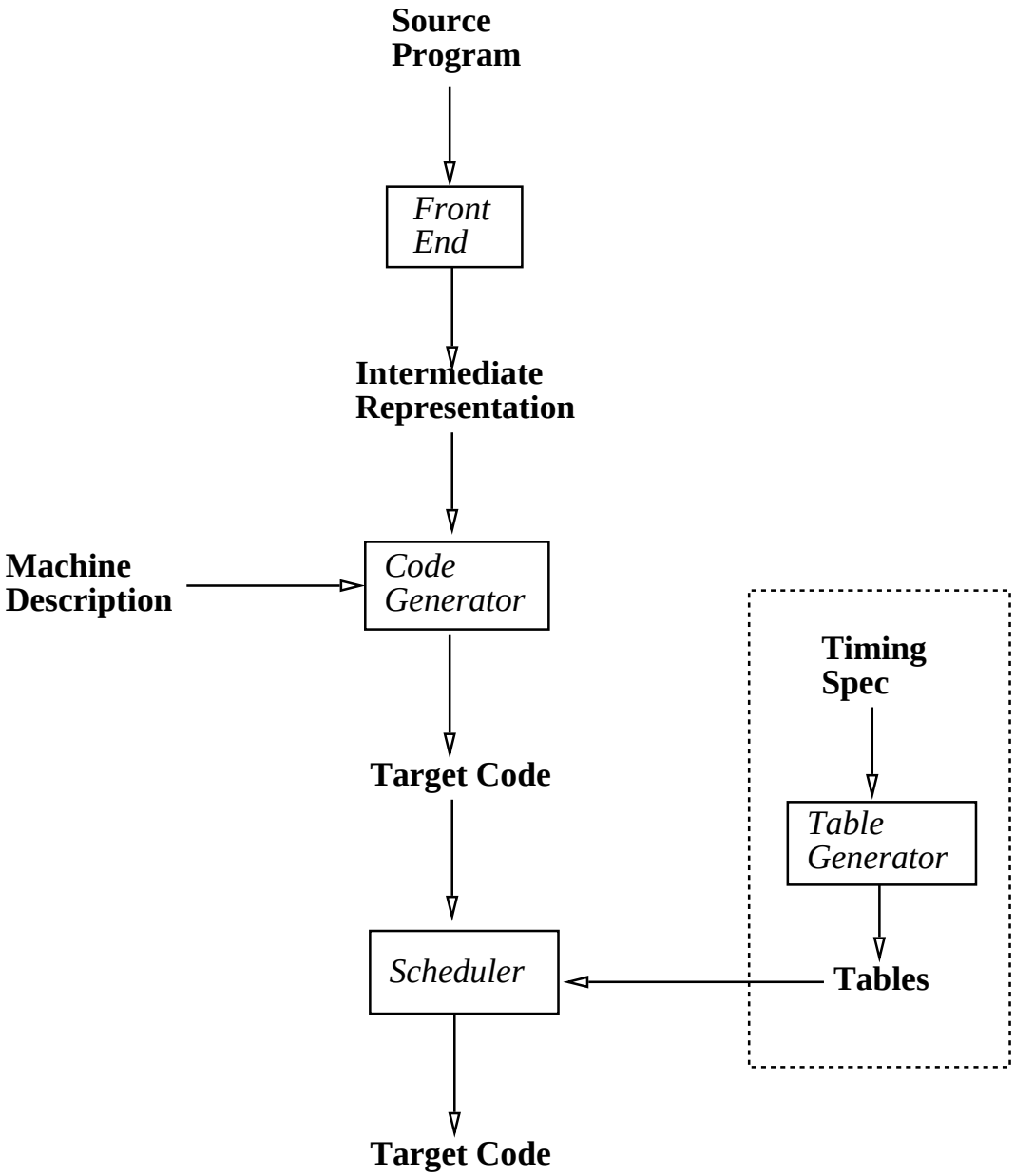

Figure 1.1: **The structure of a typical compiler.**

- Chapter 6 uses SCCS to specify various Superscalar configurations of ToyP.
- Chapter 7 we show how the Concurrency Workbench is used to simulate ToyP.
- Chapter 8 gives a brief introduction to instruction scheduling and sets the stage for, in the future, deriving instruction scheduling parameters.
- Chapter 9 concludes and discusses future research.

Chapter 2

Mathematical Specifications and Related Research

There are many formalisms available and currently being applied for specifying the intended behavior and semantics of a system. These include first order logic, higher order logic, temporal logic, equational algebra, and the lambda calculus [SMB92, BS89, LB90]. All formalisms seem to have their own benefits and deficits depending on the application domain (we have already mentioned in the introduction our criteria for choosing SCCS).

There are also a variety of formalisms for specifying asynchronous and/or synchronous concurrent systems [Mil89] including Petri Nets, CCS, SCCS, ACP, CIRCAL, and CSP [Mil89, Mil93, Hoa85, Mil85, BRR87, Dav90]. Process algebra has been used to give a semantics to a communications protocol language, LOTOS; a parallel object oriented language, POOL; a computer integrated manufacturing system; and to the specification of low-level digital hardware; these and other applications of process algebra may be found in [Mil85, Bae90, Bri88]. Of the formalisms, all except CIRCAL are for specifying asynchronous systems. CIRCAL is similar to SCCS, and our choice of SCCS is largely pragmatic: there is a large body or research to draw upon for doing formal analysis, and there are also tools available.

There has been some research into specifying instruction set architecture using functional methods [Pai90, CJL89]. In general, a functional description of an architecture specifies *final values* of an instruction and not timing or instruction interaction. The timing and

concurrent properties of a functional program are implicit, which is not what we desire. When we wish to model instructions that can interact and execute in parallel, functional methods are difficult to apply.

Chapter 3

SCCS: A Synchronous Calculus of Communicating Systems

SCCS or *Synchronous Calculus of Communicating Systems*, is a mathematical theory of communicating systems in which we can represent real systems by the *terms* or *expressions* of the model [Mil89]. SCCS allows us to directly represent the temporal and concurrent properties of the system being specified.

3.1 A Small Example

In this section we introduce SCCS through a small example of a pipeline. Consider a two stage pipeline where each stage adds one to its input; such a pipeline is depicted in figure 3.1.

Each stage is called an *agent* (what we usually call a process) in SCCS, which may have one or more communication ports. In SCCS, each agent *must* perform an action (that is, use one or more of its ports) on each clock cycle. An agent not wishing to perform an action may execute the *idle* action, written as 1. Communication between two agents occurs when, at time t, one agent wants to use a port α and the other wants to use port $\overline{\alpha}$.

One stage in our example pipeline is represented in SCCS by,

$$S(x) \stackrel{\text{def}}{=} \mathtt{in}(y)\overline{\mathtt{out}}(x) : S(y+1) \tag{3.1}$$

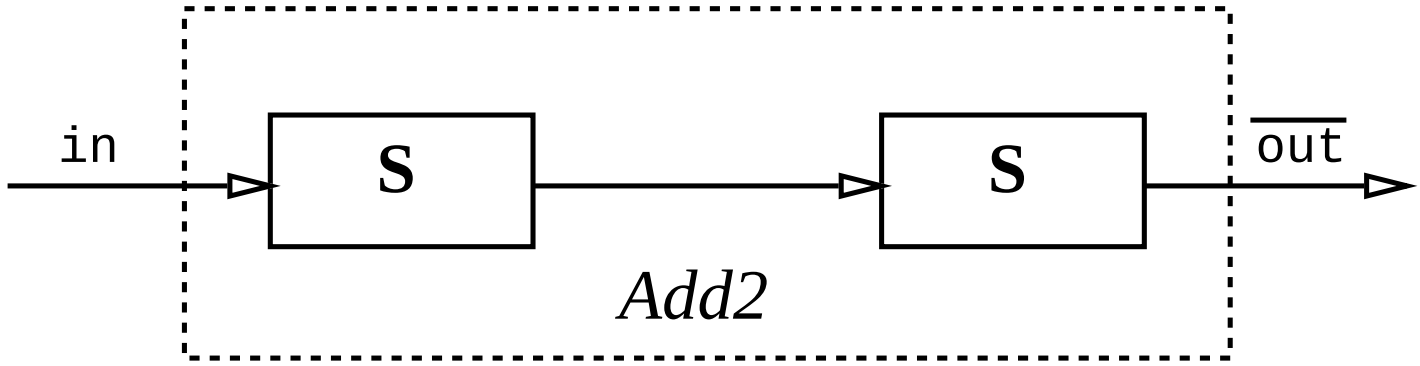

Figure 3.1: **A two stage "Add 2" pipeline constructed from two "Add 1" agents.**

Equation 3.1 specifies that on clock cycle t, S is an agent with current output x and input y and that at time $t+1$, S becomes an agent with current output $y+1$. In equation 3.1,

- ":" is called the prefix operator. In general, the expression $\mathtt{a} : P$ specifies that at time t do action $\mathtt{a}$ and then at time $t+1$ proceed with agent P.
- $\mathtt{in}(y)\overline{\mathtt{out}}(x)$ is a *product* of actions specifying that the two particulate actions, $\mathtt{in}$ and $\overline{\mathtt{out}}$, occur simultaneously. This action can also be thought of as reading y on port $\mathtt{in}$ and sending x on port $\overline{\mathtt{out}}$.
- S is defined recursively allowing for the modeling of non-terminating agents.
- S is parameterized and contains the arithmetic expression $y+1$. An important characteristic of SCCS is that the parameter of an output action may be *any* expression, using whatever functions over values we need [Mil89].

The semantics of SCCS is given formally in [Mil89].

3.2 SCCS Syntax

Systems specified by SCCS are comprised of two entities, actions and agents (or processes). Figure 3.2 gives the syntax for SCCS expressions. Actions communicate values and can

$P(x_1, x_2, \ldots, x_n) \stackrel{\text{def}}{=} E$	Parameterized agent definition
$\mathbf{1}$, $Done$	Idle agent
$\mathbf{0}$	Inactive agent
$E \times F$	Parallel composition
$E + F$	Choice of E or F
$E \rhd F$	E or F with preference for E
$a_1(x_1)a_2(x_2)\cdots a_n(x_n) : E$	Synchronous action prefix with values
$a.E$	Asynchronous action prefix
if b **then** E	Conditional
$\sum_{i \in I} E_i$	Summation over indexing set I
$\prod_{i \in I} E_i$	Composition over indexing set I
$E \uparrow L$	Action Restriction
$E \backslash\backslash L$	Particle Restriction
$E[f]$	Apply relabeling function f

Figure 3.2: **Syntax of SCCS expressions**

either be *positive* (*e.g.* $\mathtt{in}$) or negative (*e.g.* $\overline{\mathtt{out}}$). Positive actions input values and negative actions output values. Two actions α and $\bar{\alpha}$ associated with two agents running in parallel are connected by the fact that they are complements of each other.

3.3 Connecting Processes

The $\times$ combinator produces parallelism and allows for new agents to be constructed from other agents. The agent $A \times B$ represents agent A and B executing in parallel. If two agents joined by product contain complementary action names then these agents are joined by what may be thought of as wires at those ports. Hence these agents may now communicate.

Given equation 3.1 we can now construct a two stage "add 2" pipeline from two "add 1" agents. There is a problem though, the agent $S \times S$ does not contain complementary action names (that is, the pipeline stages are not connected) and the output of the first S stage must be fed into the input of the second S stage. SCCS allows us to relabel actions using a relabeling combinator. Relabeling $\overline{\mathtt{out}}$ to $\overline{\alpha}$ in the first S and $\mathtt{in}$ to α in the second occurrence of S provides the desired effect.

$$\begin{aligned} Add2(x,y) \quad &\stackrel{\text{def}}{=} \quad (S(x)[\phi_1] \times S(y)[\phi_2]) \uparrow \{\mathtt{in},\ \mathtt{out}\} \\ &\qquad \phi_1 = \mathtt{out} \mapsto \alpha, \quad \phi_2 = \mathtt{in} \mapsto \alpha \end{aligned} \tag{3.2}$$

In equation 3.2,

- ϕ_1 is a relabeling function that means change the port name $\mathtt{out}$ to α. ϕ_2 changes $\mathtt{in}$ to α.
- $S[\phi]$ means apply relabeling function ϕ to agent S.
- $S \uparrow \{\mathtt{in},\ \mathtt{out}\}$ is the *restriction* combinator applied to agent S. Restriction serves the purpose of "internalizing" ports (or "hiding" actions) from the environment and exposing others. Hence $\mathtt{in}$ and $\mathtt{out}$ are made known to the environment and α is internalized.

The net effect equation 3.2 is to construct a pipeline of two stages where each stage adds 1 to its input.

In SCCS the agent $A + B$ represent a choice of performing agent A or agent B. Which choice is taken depends upon the actions available within the environment. The agent $A_1 + A_2 + \cdots + A_n$ is abbreviated to $\sum_{i=1}^{n} A_i$.

3.4 An Algebra of Actions

Agents interact with their environment through "ports" that are identified with labels. "Port" and "label" are synonymous and every port (and label) is also an action, but as we will see the converse is not true. Actions have the following properties:

- We assume that there is an infinite set $\mathcal{A}$ of names and a set of conames $\overline{\mathcal{A}}$. The set of all labels is $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$.
- There is a binary operator $\cdot$ used to form a new action that is a "product of actions",$\alpha_1 \cdot \alpha_2$, from particulate actions α_1 and $\alpha_2 \in \mathcal{L}$.
- $\cdot$ is commutative, $\alpha \cdot \beta = \beta \cdot \alpha$.
- Often, when clarity allows, a product of actions, $\alpha_1 \cdot \alpha_2$, is written using juxtaposition (*e.g.* $\alpha_1\alpha_2$) rather than with $\cdot$.
- A product of actions, $\alpha = \alpha_1\alpha_2 \cdots \alpha_n$ denotes simultaneous occurrence of α_1,α_2,..., α_n.
- The idle action, 1, is a left and right identity on $\cdot$ such that $1\alpha = \alpha 1 = \alpha$.
- The unary operator "$^-$" is an inverse of $\cdot$ such that $\alpha\bar{\alpha} = 1$.

With this information at hand we conclude that there is an algebra of actions that form an abelian (commutative) group generated by $\mathcal{A}$, the set of particles.

$$\boxed{(Act,\ 1,\ \star,\ ^-\ ) \text{ is an Abelian Group}}$$

Therefore, every $\alpha \in Act$ can be expressed as a unique product of particulate actions

$$\alpha = \alpha_1^{z_1} \ldots \alpha_n^{z_n}$$

up to order.

3.5 Extensions to SCCS

In this section we introduce 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 can be modeled by $A \times 1 : B$. We define the binary combinator *Next* to denote this agent.

$$A \ \mathit{Next}\ B = A \times 1 : B$$

Another useful operator is the *priority sum* operator, $\triangleright$ [CW91]. Intuitively, if in the agent $A + B$ both A and B can execute, then it is non-deterministic which one is executed. We can prioritize $+$ so that if both A and B can execute, then A is preferred. Thus, $A \triangleright B$ denotes the priority sum of A and B, where A has priority over B.

3.6 Transition Graphs

The semantics of SCCS is defined in terms of a *labeled transition system.*

Definition 1 *A* **labelled transition system** *(LTS) is a triple* $\langle \mathcal{P}, \mathrm{Act}, \rightarrow \rangle$ *where* $\mathcal{P}$ *is a set of agents in SCCS,* Act *is the set of actions (as defined in section 3.4), and* $\rightarrow$ *is a subset of* $\mathcal{P} \times \mathrm{Act} \times \mathcal{P}$. *When* $p, q \in \mathcal{P}$ *and* $\alpha \in \mathrm{Act}$ *and* $(p, \alpha, q) \in \rightarrow$ *we write* $p \xrightarrow{\alpha} q$ *to mean that "agent* p *can do an* α *and evolve into* q*." Agents* p *and* q *represent the state of the system at times* t *and* $t + 1$ *respectively.*

The transition graph of an agent p is a graphical representation of the labelled transition system induced by p. For example, the SCCS agent defined in equation 3.3 has the transition graph shown in figure 3.3.

$$E \stackrel{\text{def}}{=} a : b : \mathbf{0} + c : (d : \mathbf{0} + e : \mathbf{0}) \tag{3.3}$$

The transition graph of an agent represents its dynamic nature as it represents an agent "executing" through time.

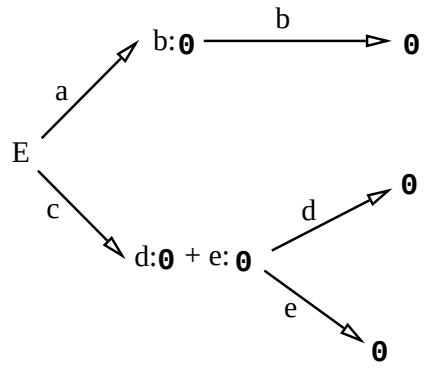

Figure 3.3: **Transition graph of agent defined in equation 3.3.**

3.7 Equational Properties

SCCS is an *algebra* that satisfies many equational laws (figure 3.4).

If $P, Q, R \in \mathcal{P}$ and $a, b \in Act$ then,

$$P \times \mathbf{1} \sim P \tag{3.4}$$

$$P + \mathbf{0} \sim P \tag{3.5}$$

$$P + P \sim P \tag{3.6}$$

$$P \times \mathbf{0} \sim \mathbf{0} \tag{3.7}$$

$$a : P \times b : Q \sim ab : (P \times Q) \tag{3.8}$$

$$P \times (Q + R) \sim P \times Q + P \times R \tag{3.9}$$

$$P + Q \sim Q + P \tag{3.10}$$

$$P \times Q \sim Q \times P \tag{3.11}$$

$$(P \times Q) \times R \sim P \times (Q \times R) \tag{3.12}$$

$$(P + Q) + R \sim P + (Q + R) \tag{3.13}$$

$$a(b : P + c : Q) \sim ab : P + ac : Q \tag{3.14}$$

$$(a : P) \uparrow A \sim \begin{cases} a : (P \uparrow A) & \text{if } a \in A \\ \mathbf{0} & \textit{if } a \notin A \end{cases} \tag{3.15}$$

$$P \uparrow A \uparrow B \sim P \uparrow (A \cap B) \tag{3.16}$$

Figure 3.4: **Equational laws of SCCS**

Chapter 4

ToyP, a 32-bit RISC

To illustrate a specification of a processor we present the instructions of a simple hypothetical RISC called ToyP. In ToyP, instructions, memory word size, registers, and addresses are thirty-two bits. Our hypothetical processor is, in fact, based on the MIPS/R4000 and closely resembles many other RISC architectures such as the SuperSPARC, Motorola 88000, and the IBM RS/6000, and DLX (a RISC presented in [PH90].

Here is an informal description of some of the ToyP instructions we describe with SCCS.

- `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 that branches to *Locn* if register i is zero. The instruction immediately after the branch is always executed before the branch is taken. 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 with a latency of six cycles. If another `Fadd` instruction tries to use the result before the current `Fadd` is finished, then instruction execution stalls until the result is ready.

Instruction Syntax	*Computational Semantics*
`Add` R_i, R_j, R_k	$R_i \leftarrow R_j + R_k$
`AddI` R_i, R_j, `#Const`	$R_i \leftarrow R_j +$ `#Const`
`Mov` R_i, R_j	$R_i \leftarrow R_j$
`Mov.f` FR_i, FR_j	$FR_i \leftarrow FR_j$
`MovFPtoI` R_i, FR_j	$R_i \leftarrow FR_j$
`MovItoFP` FR_i, R_j	$FR_i \leftarrow R_j$
`Nop`	*No operation*
`Cmp` R_i, R_j, R_k	$R_i \leftarrow R_j$? R_k
`CmpI` R_i, R_j, `#Const`	$R_i \leftarrow R_j$? `#Const`
`Fadd` FR_i, FR_j, FR_k	$FR_i \leftarrow FR_j + FR_k$
`Fmul` FR_i, FR_j, FR_k	$FR_i \leftarrow FR_j \times FR_k$
`Fdiv` FR_i, FR_j, FR_k	$FR_i \leftarrow FR_j / FR_k$
`BZ` R_i, `Locn`	if $R_i = 0$ then PC $\leftarrow$ `Locn`
`Load` R_i, R_j, `Offset`	$R_i \leftarrow$ `Mem[`$R_j +$ `Offset]`
`Load.f` FR_i, R_j, `Offset`	$FR_i \leftarrow$ `Mem[`$R_j +$ `Offset]`
`LoadI` R_i`,#Const`	$R_i \leftarrow$ `#Const`
`Store` R_i, R_j, `Offset`	`Mem[`$R_j +$ `Offset]` $\leftarrow R_i$
`Store.f` FR_i, R_j, `Offset`	`Mem[`$R_j +$ `Offset]` $\leftarrow FR_i$

Figure 4.1: **ToyP instruction set.**

4.1 Timing Constraints

There are three types of constraints that alter the programmers view of the timing view of a processor.

delayed instructions The effect of an instruction can be delayed (*e.g.*, as in ToyP's load and branch instructions) making certain instructions sequences illegal.

multicycle instructions An instruction that takes more than one cycle to calculate its result may cause the processor to interlock when a subsequent instruction needs the result before the previous instruction has finished. These are known as data hazards and will be discussed in more detail later.

limited resources Often, two or more instructions may compete for the same resource (*e.g.*, floating-point adder) and one will have to wait. This situation is known as a structural hazard.

Will will discuss all three types of ToyP's timing constraints as we encounter them.

Chapter 5

Specifying a Processor

A processor is a system of interacting processes where registers and memory interact with one or more functional units. Equation 5.1 represents such a system at the highest level.

$$Processor \stackrel{\text{def}}{=} (Instruction\ Unit \times Memory \times Registers) \uparrow I \tag{5.1}$$

$$Where\ I\ is\ the\ set\ of\ all\ instructions. \tag{5.2}$$

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

5.1 Defining the Registers

In this section we develop an abstract model of storage in which storage cells are modeled as agents. Equation 5.3 defines one cell, $Cell(y)$, holding a value y, such that an action $\texttt{putc}(x)$ executed at time t stores x in $Cell$ which is available for use at time $t+1$. The action $\overline{\texttt{getc}}(y)$ retrieves the value stored in $Cell$ and assigns this to y. If no agent wants to interact with $Cell$ using `putc` or $\overline{\texttt{getc}}$ actions then $Cell$ executes the idle action 1.

$$Cell(y) \stackrel{\text{def}}{=} \overline{\texttt{getc}}(y) : Cell(y) \ + \ \texttt{putc}(x) : Cell(x) \ + \ 1 : Cell(y) \tag{5.3}$$

This model of a storage cell is simple enough, but inadequate because $Cell$ can only perform one `getc` or `putc` at a time. Consider the instruction `Add` $\texttt{R}_1$`,` $\texttt{R}_1$`,` $\texttt{R}_1$ which accesses $\texttt{R}_1$ twice and also writes $\texttt{R}_1$. On most processors this instruction can effectively

execute in a single cycle because registers are read and written in different pipeline stages. But we do not need to model pipeline stages and all of the other organization that goes with them. What we need to do is augment the agent *Cell* so that it can handle parallel reads and writes. For example the action $\mathtt{getc}(a)\mathtt{getc}(b)$ means read *Cell* twice putting the result into a and b. The action $\mathtt{getc}(a)\overline{\mathtt{putc}}(b)$ means read and write *Cell* in parallel. The action $\mathtt{getc}(a)\mathtt{getc}(b)\mathtt{getc}(c)\overline{\mathtt{putc}}(d)$ means read *Cell* twice with the value going into a and b and write d to *Cell*. Only one `putc` is allowed for each action.

Equation 5.4 defines a new agent *Reg* that is a new version of *Cell* so that it can handle parallel reads and writes.

$$Reg1(y) \stackrel{\text{def}}{=} \sum_{j=0}^{2} \overline{\mathtt{getr}}(y)^j (1 : Reg(y) \; + \; \mathtt{putr}(x) : Reg(x)) \tag{5.4}$$

If we expand the summation in equation 5.4 we obtain equation 5.5.

$$\begin{aligned} Reg1(y) \stackrel{\text{def}}{=} & \; \overline{\mathtt{getr}}(y)^0 : Reg(y) \\ + & \; \overline{\mathtt{getr}}(y)^1 : Reg(y) \\ + & \; \overline{\mathtt{getr}}(y)^2 : Reg(y) \\ + & \; \overline{\mathtt{getr}}(y)^0 \mathtt{putr}(x) : Reg(x) \\ + & \; \overline{\mathtt{getr}}(y)^1 \mathtt{putr}(x) : Reg(x) \\ + & \; \overline{\mathtt{getr}}(y)^2 \mathtt{putr}(x) : Reg(x) \end{aligned} \tag{5.5}$$

Applying the equational law that states that for any particulate action $a \in \mathcal{A}$, $a^0 = 1$ and $a^1 = a$ equation 5.5 reduces to equation 5.6.

$$\begin{aligned} Reg1(y) \stackrel{\text{def}}{=} & \; \overline{\mathtt{getr}}(y) : Reg(y) \\ + & \; \overline{\mathtt{getr}}(y)^2 : Reg(y) \\ + & \; \mathtt{putr}(x) : Reg(x) \\ + & \; \overline{\mathtt{getr}}(y) \mathtt{putr}(x) : Reg(x) \\ + & \; \overline{\mathtt{getr}}(y)^2 \mathtt{putr}(x) : Reg(x) \\ + & \; 1 : Reg(y) \end{aligned} \tag{5.6}$$

In further sections, we will not continue to expand summations like this, as it was done here to get used to seeing how summations are used and manipulated. In fact, the algebra

allows the use of infinite sums, that is, sums over a countably infinite indexing set, which would prohibit us from expanding them completely anyway.

5.1.1 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 (instruction) should result in an error. We will augment equation 5.4 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**. Equation 5.7 modifies *Reg1* so that when an agent locks a register the register goes into a state *Locked_Reg* where the only allowable action is $\overline{\texttt{putr}}(x)\texttt{releasereg}$. 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 this out and put them in equation 5.9.

$$Reg(y) \stackrel{\text{def}}{=} Reg1(y) + \sum_{j=0}^{2} \overline{\texttt{getr}}(y)^j \texttt{lockreg} : Locked\_Reg(y) \tag{5.7}$$

$$Locked\_Reg(y) \stackrel{\text{def}}{=}$$

$$Illegal\_Access(y) + \texttt{putr}(x)\texttt{releasereg} : Reg(x) + 1 : Locked\_Reg(y) \tag{5.8}$$

$$Illegal\_Access(y) \stackrel{\text{def}}{=}$$

$$\sum_{j=0}^{2} \overline{\texttt{getr}}(y)^j \, (\overline{\texttt{getr}}(y) : \mathbf{0} + \texttt{putr}(x) : \mathbf{0} + \texttt{putr}(x)\texttt{releasereg} : \mathbf{0}) \tag{5.9}$$

Figure 5.1 shows the state transition graph of *Reg* (equation 5.7).

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

$$Registers \stackrel{\text{def}}{=} Reg_0(y) \times \cdots \times Reg_{31}(y) \tag{5.10}$$

which we abbreviate to

$$Registers \stackrel{\text{def}}{=} \prod_{i=0}^{31} Reg_i(y) \tag{5.11}$$

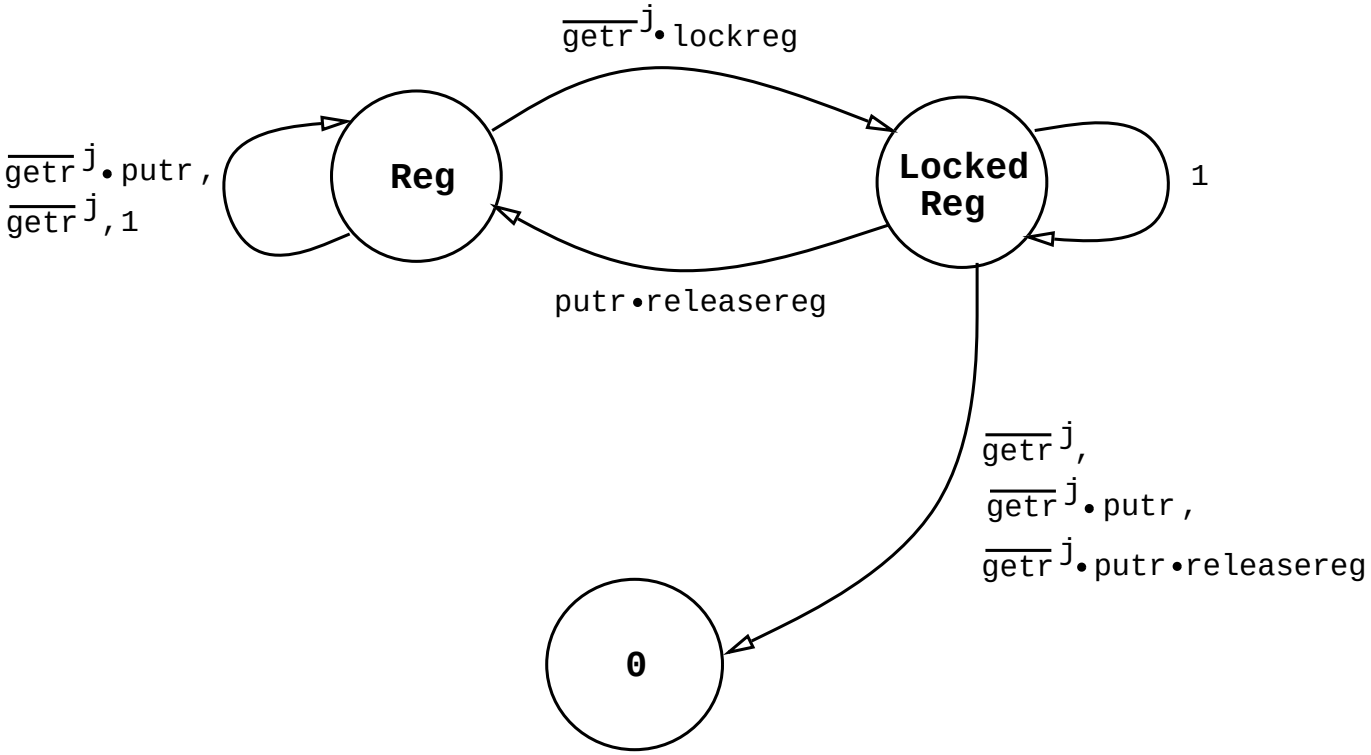

Figure 5.1: **State transition graph of agent** *Reg* **in equation 5.7.**

5.2 Defining Memory

The definition of an agent *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.

$$\begin{aligned} MEM_i(y) &\stackrel{\text{def}}{=} \mathtt{putm}_i(x) : MEM_i(x) \\ &+ \overline{\mathtt{getm}_i}(y) : MEM_i(y) \\ &+ \overline{\mathtt{getm}}(y)\mathtt{putm}(x) : MEM(x) \\ &+ 1 : MEM_i(y) \qquad (5.12) \\ MEMORY &\stackrel{\text{def}}{=} \prod_{i=0}^{2^{32}-1} MEM_i(y) \qquad (5.13) \end{aligned}$$

5.3 Instruction Pipeline

Instruction pipelines are usually described in terms of their stages of execution. For example, the agent *IPL* (for instruction pipeline)

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

defines a five-stage instruction pipeline, where *IF, ID, EX, MEM,* and *WB* represent instruction fetch, decode, execute, memory access, and writeback stages.

This is a reasonable and obvious representation, but since we are interested only in external timing 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 detailed operation of each individual stage. Since our interest is simply timing behavior we need a more abstract specification.

We should also point out that it is very useful to be able to specify a processor at this lower organizational level as this would count as an "implementation" of the processor. In fact, one of SCCS's major benefits is its ability to specify systems at various levels and compare and analyze them. This robustness is one of the resons we chose SCCS.

5.4 Instruction Issue

Given our previous definitions of *Registers* and *Memory* and using a program counter, PC, we now describe an agent $Instr(PC)$ (equation 5.14) that specifies the behavior of ToyP instructions. $Instr(PC)$ partitions instructions into two classes, *Branch* and *Non_Branch*. *Non_Branch* instructions are further divided into three classes, arithmetic (*Alu*), load and store (*Load_Store*), and floating-point (*Float*).

$$\begin{aligned} Instr(PC) &\stackrel{\text{def}}{=} (Non\_Branch(PC)\ Next\ \ Instr(PC+4)) \\ &+ \ Branch(PC) \\ &\triangleright \ Stall(PC) \qquad (5.14) \\ Non\_Branch(PC) &\stackrel{\text{def}}{=} Alu(PC) + Load\_Store(PC) + Float(PC) \qquad (5.15) \\ Stall(PC) &\stackrel{\text{def}}{=} 1 : Instr(PC) \qquad (5.16) \end{aligned}$$

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 5.14 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. Hence, the decision on what instruction to execute next is deferred (see equation 5.19).
- If no instruction can execute then the processor must stall (equation 5.16). The $\triangleright$ operator (section 3.5) is used here because the processor should stall only when no other alternative is available.

5.4.1 Arithmetic Instructions

Like most architectures, ToyP fetches instructions from memory using a program counter, PC. The action

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

represents fetching an `Add` instruction from memory.

From a user's view, the 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. (We will see, later on, how we can deduce whether instructions sequences are legal or not.)

The agent

$$Alu(PC) \stackrel{\text{def}}{=} \mathtt{getm}_{\mathrm{PC}}(\mathtt{Add\ R}_i\mathtt{,R}_j\mathtt{,R}_k)\mathtt{getr}_j(x)\mathtt{getr}_k(y)\overline{\mathtt{putr}_i}(x+y) : Done \qquad (5.17)$$

represents the execution of the `Add` instruction. At time t, source registers j and k are read (by the actions $\mathtt{getr}_j(x)\mathtt{getr}_k(y)$) and the result is written to destination register i (by the action $\overline{\mathtt{putr}_i}(x+y)$).

In fact, equation 5.17 describes the same computation as the register transfer statement

```
Reg[i] ← Reg[j] + Reg[k]
```

except that the SCCS equation specifies that registers are accessed and the result is written atomically (*i.e.,* executes in a single cycle). The agent *Done* is the idle agent and represents termination of the instruction (agent).

5.4.2 Integer 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,

$$
\begin{aligned}
&\mathit{Load\_Store}(PC) \stackrel{\text{def}}{=} \\
&\quad \texttt{getm}_{\text{PC}}(\texttt{Load R}_i\texttt{, R}_j\texttt{, }\Delta)\texttt{getr}_j(B)\texttt{getm}_{\text{B}+\Delta}(V)\overline{\texttt{lockreg}}_i : \\
&\quad \overline{\texttt{putr}}_i(V)\overline{\texttt{releasereg}}_i : \mathit{Done}
\end{aligned} \tag{5.18}
$$

Equation 5.18 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 (by action $\texttt{getr}_j(B)$).
2. Memory is fetched with the value placed in the variable V (with action $\texttt{getm}_{\text{B}+\Delta}(V)$ where Δ is the offset value).
3. The destination register i is locked (using the action $\overline{\texttt{lockreg}}_i$).

At time $t+1$, two actions occur.

1. The value V is written to destination register i (with the action $\overline{\texttt{putr}}_i(V)$).
2. The destination register i is released (with the action $\overline{\texttt{releasereg}}_i$).

5.4.3 The Branch Instruction

In the following instruction sequence,

```
Locn:
        ⋮
        BZ R1, Locn
        Add R2, R2, #-1
        ⋮
```

the **Add** instruction after the branch is always executed before the jump to **Locn**. If the branch is not taken then the **Add** instruction is skipped and the instruction below **Add** is executed. A **BZ** instruction may not be immediately followed by another **BZ** instruction.

Equation 5.19 specifies the behavior of the BZ instruction.

$$
\begin{aligned}
\mathit{Branch}(PC) \quad \stackrel{\text{def}}{=} \quad & \mathtt{getm}_{\mathrm{PC}}(\mathtt{BZ\ R}_i\mathtt{,}\ \mathit{Locn})\mathtt{getr}_i(V): \\
& \quad \textbf{if } V = 0 \textbf{ then} \\
& \quad\quad \mathit{Non\_Branch}(PC+4)\ \mathit{Next}\ \ \mathit{Instr}(\mathit{Locn})) \\
& \quad\quad + \ \mathtt{getm}_{\mathrm{PC}+4}(\mathtt{BZ\ R}_i\mathtt{,}\ \mathit{Locn}) : \mathbf{0} \\
& \quad \textbf{else} \\
& \quad\quad \mathit{Instr}(PC+8)
\end{aligned}
\tag{5.19}
$$

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 $\mathtt{R}_i$ is not zero then execution continues with the instruction after the branch delay slot.
- at time $t+1$, if the value of $\mathtt{R}_i$ 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.

5.5 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 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.

5.5.1 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 here is that accessing

a locked integer register is illegal and accessing a locked FP-register 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.

$$\begin{aligned}
Freg_i(y) &\stackrel{\text{def}}{=} \sum_{j\in\{0,1,2\}} \overline{\texttt{getfr}_i}(y)^j \texttt{lockfreg}_i : Locked_Freg_i \\
&+ \sum_{j\in\{1,2\}} \overline{\texttt{getfr}_i}(y)^j : Freg_i(y) \\
&+ 1 : Freg_i(y) \\
Locked_Freg_i &\stackrel{\text{def}}{=} \texttt{putfr}_i(x)\texttt{releasefreg}_i : Freg_i(x) \\
&+ 1 : Locked_Freg_i
\end{aligned}$$

Thirty-two FP-registers are constructed analogously to the integer registers.

$$FP\_Registers \stackrel{\text{def}}{=} \prod_{j=0}^{31} Freg_j(y) \tag{5.20}$$

5.5.2 The `Fadd` instruction

Now that interlocked registers are defined we can define the behavior of the floating point add instruction. The `Fadd` instruction must,

1. access its source registers and
2. lock its destination register
3. compute the addition
4. write the result in the destination register
5. release the destination register

Equation 5.21 specifies Toyp's `Fadd` instruction.

$$\begin{aligned}
Float(PC) \stackrel{\text{def}}{=}\ & \texttt{getm}_{\text{PC}}(\texttt{Fadd, FR}_i\texttt{, FR}_j\texttt{, FR}_k)\overline{\texttt{lockfreg}_i}\texttt{getfr}_j(x)\texttt{getfr}_k(y) : \\
& (1:)^5 \\
& \overline{\texttt{putfr}_i}(x+y)\overline{\texttt{releasefreg}_i} : Done
\end{aligned} \tag{5.21}$$

The abbreviation $(1)^n$ represents the n-cycle delay, $\overbrace{1:1\ldots1}^{n \text{ times}}$, which is interpreted as n-cycles of internal computation. The processor stalls when an instruction wishes to access a locked FP-register. This happens because the instruction will not be able to access the FP-register and the only other option is to execute the agent *Stall* (equation 5.14). (Remember in the definition of $Instr(PC)$ in equation 5.14 that ToyP continues executing instructions after `Fadd` is starts.)

5.6 Structural Constraints

Processors often reuse functional units. For example, a floating-point unit may have only one adder that is used by the addition, multiply, and division instructions. This "failure" to fully replicate the resource for each instruction that needs it gives rise to *structural hazards* which can alter the timing characteristics of the instructions that require the resource. Moreover, an instruction may require a resource several times for various lengths of time during its execution, hence, making the resource constraints complex to describe.

As an example, the MIPS/R4000 floating-point unit has a floating-point adder, divider, rounder, and shifter (it also has several other functional units in the FPU such as an exception checker, but we will keep it simple). The single precision divide instruction, `FDIV`, requires the:

- floating-point adder and shifter on clock cycle 2
- rounder and the shifter on cycle 3
- shifter on cycle 4
- divider on cycles 5 through 36
- divider and adder on cycles 37 and 39
- divider and rounder on cycles 38 and 40
- adder on cycle 41
- rounder on cycle 42.

5.7 Modeling Finite Resources

To model limited resources, we introduce a generic agent *Resource* that models a resource which instructions can acquire and release. When an instruction acquires a resource that is being used by another, the processor stalls. Equation 5.22 defines a generic agent *Resource* that can be acquired (with the action `get_resource`) and released (with the action `release_resource`). This agent will be replicated however many times is needed, once for each resource, and suitably relabeled.

$$\begin{aligned} \mathit{Resource} &\stackrel{\text{def}}{=} \texttt{get\_resource} : \mathit{Locked\_Resource} \\ &+ \texttt{get\_resource} \cdot \texttt{release\_resource} : \mathit{Resource} \\ &+ 1 : \mathit{Resource} \qquad (5.22) \\ \mathit{Locked\_Resource} &\stackrel{\text{def}}{=} \texttt{release\_resource} : \mathit{Resource} \\ &+ 1 : \mathit{Locked\_Resource} \qquad (5.23) \end{aligned}$$

Notice that the functionality of a resource is not being modeled; only an instruction's capability to use the resource exclusively. Hence, modeling a floating-point adder is exactly like modeling a floating-point multiplier. We could specify the (pipelined) floating-point unit in detail if we desired, but this would mire us in irrelevant organizational detail.

In ToyP, the floating-point unit has three resources that must be shared: an adder, multiplier, and a divider. Equation 5.24 specifies this situation by replicating *Resource* three times and relabeling its actions appropriately.

$$\begin{aligned} \mathit{FPU} &\stackrel{\text{def}}{=} \mathit{Resource}[\phi] \times \mathit{Resource}[\psi] \times \mathit{Resource}[\theta] \qquad (5.24) \\ \text{where} \quad & \phi = \texttt{get\_resource} \mapsto \texttt{get\_multiplier}, \\ & \qquad \texttt{release\_resource} \mapsto \texttt{release\_multiplier} \\ & \psi = \texttt{get\_resource} \mapsto \texttt{get\_adder}, \\ & \qquad \texttt{release\_resource} \mapsto \texttt{release\_adder} \\ & \theta = \texttt{get\_resource} \mapsto \texttt{get\_divider}, \\ & \qquad \texttt{release\_resource} \mapsto \texttt{release\_divider} \end{aligned}$$

We assume that an agent acquires a resource on the first cycle that it needs it and releases

the resource on the last cycle that it needs it. For example, if an instruction needs the adder for one cycle only, then it will perform the action product $\texttt{get\_adder}\cdot\texttt{release\_adder}$. If an instruction needs the adder for two consecutive cycles then it should do $\texttt{get\_adder}:\texttt{release\_adder}$ and not acquire and release the adder on each cycle as in the following.

$$\texttt{get\_adder}\cdot\texttt{release\_adder}:\texttt{get\_adder}\cdot\texttt{release\_adder}$$

5.7.1 Multi-cycle Floating-point Instructions

Now that there are several floating-point instructions competing for shared resources the `Fadd` instruction defined in equation 5.21 needs to be altered. We redefine the agent $\mathit{Float}(PC)$ to the following.

$$\mathit{Float}(PC) \stackrel{\text{def}}{=} \mathit{FADD}(PC) + \mathit{FMUL}(PC) + \mathit{FDIV}(PC)$$

The `Fadd` instruction requires the adder for two cycles after the operands are accessed.

$$\begin{aligned}\mathit{FADD}(PC) \stackrel{\text{def}}{=}\ & \texttt{getm}_{\text{PC}}(\texttt{Fadd, FR}_i\texttt{, FR}_j\texttt{, FR}_k)\overline{\texttt{lockfreg}_i}\texttt{getfr}_j(x)\texttt{getfr}_k(y):\\ & \overline{\texttt{get\_adder}}:(1:)^3\overline{\texttt{release\_adder}}:\\ & \overline{\texttt{putfr}_i}(x+y)\overline{\texttt{releasefreg}_i}:\mathit{Done}\end{aligned} \tag{5.25}$$

The `Fmul` and `Fdiv` can now similarly defined. The `Fmul` instruction requires the adder for one cycle, then the multiplier for two cycles, then the adder again for one cycle.

$$\begin{aligned}\mathit{FMUL}(PC) \stackrel{\text{def}}{=}\ & \texttt{getm}_{\text{PC}}(\texttt{Fmul, FR}_i\texttt{, FR}_j\texttt{, FR}_k)\overline{\texttt{lockfreg}_i}\texttt{getfr}_j(x)\texttt{getfr}_k(y):\\ & \overline{\texttt{get\_adder}}\cdot\overline{\texttt{release\_adder}}:\\ & \overline{\texttt{get\_multiplier}}:\overline{\texttt{release\_multiplier}}:\\ & \overline{\texttt{get\_adder}}\cdot\overline{\texttt{release\_adder}}:\\ & \overline{\texttt{putfr}_i}(x*y)\overline{\texttt{releasefreg}_i}:\mathit{Done}\end{aligned} \tag{5.26}$$

The `Fdiv` instruction requires, after the operands are accessed, the adder for one cycle, the divider for eight cycle, and then the adder again for two cycles.

$$\mathit{FDIV}(PC) \stackrel{\text{def}}{=} \texttt{getm}_{\text{PC}}(\texttt{Fdiv, FR}_i\texttt{, FR}_j\texttt{, FR}_k)\overline{\texttt{lockfreg}_i}\texttt{getfr}_j(x)\texttt{getfr}_k(y):$$

$$\overline{\texttt{get\_adder}} \cdot \overline{\texttt{release\_adder}} :$$

$$\overline{\texttt{get\_divider}} : (1 :)^6 \overline{\texttt{release\_divider}} :$$

$$\overline{\texttt{get\_adder}} : \overline{\texttt{release\_adder}} :$$

$$\overline{\texttt{putfr}_i}(x/y)\overline{\texttt{releasefreg}_i} : \mathit{Done} \tag{5.27}$$

Essentially, we are modeling a resource as a binary semaphore. This technique can be used to handle any kind of resouce that needs to be exclusively accessed (*e.g.,* register/memory ports, busses, etc.).

A processor may have more than one copy of a particular resource. For example, there may be two independent floating-point adders that can be used by any of the floating-point instructions. In this case we duplicate the resource using the same label for each. For example, two adders would be specified as

$$\begin{aligned} &\mathit{Resource}[\phi] \times \mathit{Resource}[\phi] \\ &\qquad \text{where } \phi = \texttt{get\_resource} \mapsto \texttt{get\_adder} \end{aligned} \tag{5.28}$$

and when an agent wishes to acquire one of them then there are three possibilities:

1. Both adders are free and one of them is non-deterministically chosen.
2. If one adder is being used and the other is free, then the free adder is acquired.
3. If both adders are busy then the instruction cannot continue and the pipeline stalls (as in the case for the interlocked floating-point registers).

Chapter 6

Superscalar Versions of ToyP

6.1 An *Integer* × *Float* Superscalar

This section describes a superscalar version of ToyP that can issue one floating-point and one integer instruction per cycle. If two instructions can be issued in parallel, then we have either 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 6.1.

$$\begin{aligned} & (\mathit{Float}(PC) \times \mathit{Alu}(PC+4)) \\ + \quad & (\mathit{Alu}(PC) \times \mathit{Float}(PC+4)) \end{aligned} \tag{6.1}$$

We can rewrite this sum as equation 6.2.

$$\sum_{i,j \in \{0,4\}} (\mathit{Alu}(PC+i) \times \mathit{Float}(PC+j)) \tag{6.2}$$

Assuming an instruction is not both an integer and floating-point, equation 6.2 represents a folding of equation 6.1. Equation 6.2 is a sum of four terms, however, when $i = j$ then $\mathit{Alu}(PC+i)$ and $\mathit{Float}(PC+j)$ refer to the same memory location and we can conclude that

$$\mathit{Alu}(PC+i) \times \mathit{Float}(PC+j) \sim \mathbf{0}$$

and only two terms remain (equation 6.1). We use the summation notation because it enables us to succinctly specify n-way instruction parallelism.

Equation 6.3 extends equation 6.2 to continue execution at $PC + 8$.

$$Do\_Two(PC) \stackrel{\text{def}}{=} \left(\sum_{i,j \in \{0,4\}} (Alu(PC + i) \times Float(PC + j)) \right) \; Next \;\; Instr(PC + 8) \tag{6.3}$$

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

6.1.1 Instruction Issue

Our top-level instruction issue equation (equation 5.14) must now be modified to take this new two-issue capability into account. For reference, we restate *Instr* (equation 5.14), and rename it *Do_One*.

$$\begin{aligned} Do\_One(PC) \quad &\stackrel{\text{def}}{=} \quad (Non\_Branch(PC) \; Next \;\; Instr(PC + 4)) \\ &+ \quad Branch(PC) \end{aligned} \tag{6.4}$$

The processor can execute two, one, or zero (*i.e.,* stall) instruction(s) per cycle, which we capture by,

$$Instr(PC) \stackrel{\text{def}}{=} Do\_Two(PC) \rhd Do\_One(PC) \rhd Stall(PC) \tag{6.5}$$

Notice here the use of the priority choice operator, $\rhd$ (section 3.5) 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.

6.2 An *Integer* × *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

$$Alu(PC) \times Alu(PC + 4) \tag{6.6}$$

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. Before we continue, we need to introduce the various types of data hazards that can arise.

```
Add R1, R1, R1
Add R2, R3, R4
```
(a) No dependencies

```
Add R1, R1, R1
Add R2, R1, R3
```
(b) Read-After-Write hazard

```
Add R2, R1, R1
Add R1, R3, R3
```
(c) Write-After-Read hazard

```
Add R1, R2, R3
Add R1, R3, R3
```
(d) Write-After-Write hazard

Figure 6.1: **Possible data dependencies in instruction sequences.**

6.2.1 Data Dependencies (or Data Hazards)

The instructions in figure 6.1 represent all of the possible dependencies that can exist between any two instructions[1].

The instructions in 6.1a can be executed in parallel because they are data independent while those in 6.1b cannot be executed in parallel because of the RAW hazard on `R1`.

In 6.1c parallel execution is possible if the processor can do register renaming thus eliminating the WAR hazard. However, we can specify that the instructions *must* be executed in parallel without having to specify the renaming hardware as we don't wish to overspecify. The instructions in 6.1d present a rare (but possible) WAW hazard. Here, the final value of `R1` must be `R3 + R3`. Two solutions are possible. First, since the hazard is rare, execute the instructions sequentially. Second, execute them in parallel and insure that `R1` gets the result of the second instruction. For simplicity, we will choose the first option.

6.2.2 Specifying Data Hazards

Using restriction, we can force equation 6.6 to apply only to legal integer instruction sequence of length two. If the first integer instruction writes register i then the second integer

[1] There is a confusion in terminology with regards to data hazards. Computer engineers refer to them as RAW, WAR, and WAW hazards, while computer scientists refer to them as *forward*, *anti-*, and *output* dependencies respectively. Forward dependencies are sometimes referred to as *true* dependencies.

instruction cannot write or read register i. For example, given two integer instructions, if the first instruction writes register 0 then equation 6.7 represents the legal integer instruction sequences.

$$\begin{aligned} Alu(PC)\backslash\backslash A \quad &\times \quad Alu(PC+4)\backslash\backslash B \\ \textbf{where} \qquad & A = \{\mathtt{putr}_0, \mathtt{getr}_0, \ldots, \mathtt{getr}_{31}\} \\ & B = \{\mathtt{putr}_1, \ldots, \mathtt{putr}_{31}, \mathtt{getr}_1, \ldots, \mathtt{getr}_{31}\} \end{aligned} \tag{6.7}$$

Here, the agent $P\backslash\backslash S$ represents *particle restriction* on the agent P where S is a set of particles that P may execute [Mil83]. In equation 6.7 the restriction on the first instruction by the set A specifies that only register zero is a possible destination register while the restriction on the second instruction by the set B specifies that register zero cannot be a source register nor a destination register.

Summing over all possible destination registers of the first instruction yields the desired result.

$$\begin{aligned} Do\_Two(PC) \quad &\stackrel{\text{def}}{=} \quad \sum_{i=0}^{31} (Alu(PC)\backslash\backslash A \times Alu(PC+4)\backslash\backslash B) Next(PC+8) \\ \textbf{where} \qquad & A = \{\mathtt{putr}_i, \mathtt{getr}_0 \ldots \mathtt{getr}_{31}\} \\ & B = \{\mathtt{getr}_0 \ldots \mathtt{getr}_{31}, \mathtt{putr}_0 \ldots \mathtt{putr}_{31}\} - \{\mathtt{putr}_i, \mathtt{getr}_i\} \end{aligned} \tag{6.8}$$

Equation 6.8 represents all of the allowable integer instruction sequences of length two that may execute in parallel.

6.3 An *Integer* × *Integer* × *Float* Superscalar

In section we describe a version of ToyP that can execute three instructions in parallel: two of which can be integer instructions and the other may be a floating-point instruction. Now that we have already specified two dual-issue versions (sections 6.2 and 6.1) the three issue version follows nicely.

In this case, however, the easiest way to write the equation is as a sum of three terms, one for each possible position of the floating-point instruction (ignoring data hazards, for

```
Add R1, R1, R1
Add R2, R3, R4
Fadd FR1, FR2, FR3
```
(a) No dependencies

```
Add R1, R1, R1
Fadd FR1, FR2, FR3
Add R2, R1, R3
```
(b) RAW hazard

Figure 6.2: **Possible three-issue instruction sequences.**

the time being).

$$\begin{aligned} & (Alu(PC) \times Alu(PC+4) \times Float(PC+8)) \\ + \quad & (Alu(PC) \times Float(PC+4) \times Alu(PC+8)) \\ + \quad & (Float(PC) \times Alu(PC+4) \times Alu(PC+8)) \end{aligned} \tag{6.9}$$

Adding data hazard constraints to equation 6.9 as in equation 6.8 gives us equation 6.10.

$$Do\_Three(PC) \stackrel{\text{def}}{=} \sum_{i=0}^{31} \left(\begin{array}{lllllll} & Alu(PC)\backslash\backslash A & \times & Alu(PC+4)\backslash\backslash B & \times & Float(PC+8) \\ + & Alu(PC)\backslash\backslash A & \times & Float(PC+4) & \times & Alu(PC+8)\backslash\backslash B \\ + & Float(PC) & \times & Alu(PC+4)\backslash\backslash A & \times & Alu(PC+8)\backslash\backslash B \end{array} \right) Next\ Instr(PC+8)$$

$$\begin{aligned} \textbf{where} \quad & A = \{\mathtt{putr}_i, \mathtt{getr}_0 \ldots \mathtt{getr}_{31}\} \\ & B = \{\mathtt{getr}_0 \ldots \mathtt{getr}_{31}, \mathtt{putr}_0 \ldots \mathtt{putr}_{31}\} - \{\mathtt{putr}_i, \mathtt{getr}_i\} \end{aligned} \tag{6.10}$$

The top-level issue equation now allows executing three, two, one, oe zero instructions on a cycle.

$$Instr(PC) \stackrel{\text{def}}{=} Do\_Three(PC) \triangleright Do\_Two(PC) \triangleright Do\_One(PC) \triangleright Stall(PC) \tag{6.11}$$

As an example, the instruction sequence in figure 6.2a can be executed in parallel according to equation 6.10 while the sequence in figure 6.2b cannot because of the hazard.

The situation is much the same for specifying four-issue, five-issue, etc.

Chapter 7

Simulation

In this section we show how our SCCS specification of ToyP is simulated. The simulation occurs within the framework of the Concurrency Workbench [CPS93] which allows us to experiment with, simulate, and analyze SCCS specifications.

A simulation of our processor specification amounts to loading a program into memory (with `putm` actions) and then running the that represents the processor. That is, We can observe the behavior of the program by calculating the transition graph of an agent.

Recall from section 3.6 that the transition graph of an agent P is comprised of transitions of the form $A \xrightarrow{\mathtt{a}} B$.

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

Because ToyP represents such a large system, it is not feasible to write down the entire graph.

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 agents have been omitted. In a complete graph each particle

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

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

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

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

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

Figure 7.1: **Derivation of program executing on ToyP**

on an edge would have a corresponding complementary action. At the nodes, unchanged register and memory cells have been replaced by an ellipsis.

7.1 A Simple Example

Figure 7.1 shows the transition graph of the following ToyP program.

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

We assume that the program is loaded into memory (with $\overline{\texttt{putm}}$ actions) starting at PC.

In the graph in figure 7.1 notice that

- Time t is the initial state of the processor.
- Time $t+1$ is the state of the processor after executing the `Add` instruction.
- Time $t+2$ is the final state of the processor after the `Mov` instruction is executed.

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

$[\texttt{Load R}_1\texttt{, R}_2\texttt{, \#8}][\texttt{getr}_2(Base)][\texttt{getm}_{Base+8}(V)][\texttt{lockreg}_1]$

$t+1$: $\boxed{Instr(PC+4) \times (\overline{\texttt{putr}}_1(V)\overline{\texttt{releasereg}}_1 : Done) \times Locked\_Reg_1(y) \times \cdots}$

$[\texttt{Mov R}_3\texttt{, R}_1][\texttt{getr}_1(x)][\overline{\texttt{putr}}_3(x)][\overline{\texttt{putr}}_1(V)][\overline{\texttt{releasereg}}_1]$

$t+2$: $\boxed{\mathbf{0}}$

Figure 7.2: **Derivation of illegal program executing on ToyP**

7.2 Example: An Illegal Instruction Sequence

Executing an illegal instruction sequence on our specification should lead to the inactive agent **0**. Figure 7.2 traces the following illegal instruction sequence.

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

On the second transition the particle $\texttt{getr}_1(x)$ causes the agent $Locked\_Reg_1$ to change to the agent **0** (see equation 5.9).

7.3 Example: A Floating-Point Vector Sum

For a more complete example, we trace our processor's behavior on a program that calculates the vector sum of a floating-point array (figure 7.3). For this example, we use the superscalar version of ToyP defined in section 6.1.

There are instructions in figure 7.3 which we have not specified. `LoadI`, `AddI`, and `CmpI` are load, add, and compare instructions where the third operand is an immediate constant. `LoadI`'s timing properties are the same as `Load`, and `AddI` and `CmpI` take one cycle to execute

```
       LoadI R0, #0
       LoadI R1, #0
       LoadI R2, #Vec
       MovItoFP FR0, R0
Loop:  Load.f FR1, R2, #0
       AddI R2, R2, #4
       AddI R1, R1, #1
       Fadd FR0, FR0, FR1
       CmpI R0, R1, #10
       BZ R0, Loop
       Nop
```

Figure 7.3: **ToyP program that calculates a vector sum.**

(as in `Add`). `MovItoFP` moves data from an integer register to a FP-register and takes one cycle to execute. `Load.f` is a delayed floating-point load instruction.

Figure 7.4 shows a partial transition graph of the vector sum program up to the first execution of the `Fadd` instruction. We can see from the transition graph that,

- at least one new instruction is issued every cycle. That is, on every transition there is an action that represents an instruction.
- at time $t+1$, $t+2$, $t+3$, and $t+5$ we can observe the effect of the delayed load instructions by observing a left-over agent that writes and releases the destination register. At these nodes we can also see that the destination register of each load is in its locked state *Locked_Reg*.
- From time $t+6$ and $t+7$ we can observe that two instructions are executing in parallel, one integer and one floating-point.

inputschedule

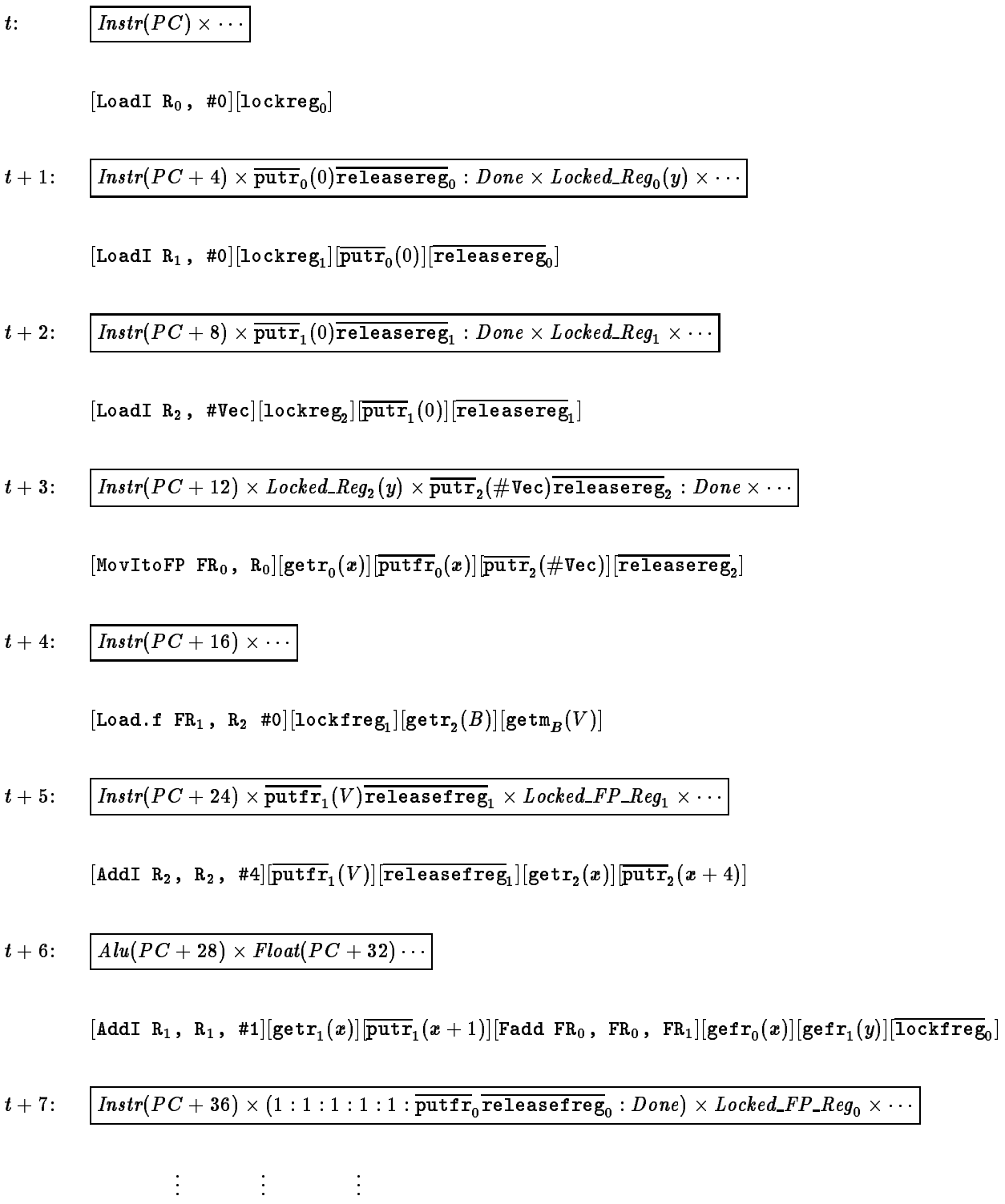

Figure 7.4: **Transition graph of vector sum from figure 7.3.**

Chapter 8

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 about instruction latencies. We are currently researching automatic derivation of instruction scheduling parameters from the transition graphs.

We should also stress that formal specifications are valuable in their own right and are a starting point for a variety of applications (as is explained in the introduction). One benefit is that they require the designer to thoughtfully plan the design in a structured manner.

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.

Appendix A

ToyP in the Concurrency Workbench

In this chapter we list how the SCCS specification of ToyP is implemented using the Concurrency Workbench [CPS93, Mol92].

A few notes about the SCCS description and the CWB SCCS description. The Concurrency Workbench implements the "basic" SCCS calculus. Consequently, this means that there is no generalized summation (*e.g.,* $\sum$) in the Workbench. In order to keep the description tangible, I have given only a "timing" specification and have excised all values, register and memory values.

There is also some notational differences.

- Action product is specified with # instead of juxtaposition. For example, the agent `ab` : **0** is `a#b`:**0**.
- Action complement is done using a single quote character (`'out` instead of $\overline{\texttt{out}}$).
- Parallel composition is, for example, $A|B$ instead of $A \times B$.

A.1 The CWB Listing

```
************************************************************************
**** Some auxillary definitions
************************************************************************
bi DONE 1:DONE

************************************************************************
**** Definition of a memory cell.  There can be 0, 1, or 2
**** readers of the register.  There is no problem with having more,
**** they're just not defined.  Only one writer is allowed and it may
**** be simultaneous with a read.
****
**** This is basically an unfolding of a summation.
************************************************************************
bi   Reg0'   putr0:Reg0                              \
          + 'getr0:Reg0                              \
          + 'getr0^2:Reg0                            \
          + 'getr0^3:Reg0                            \
          + 'getr0^4:Reg0                            \
          +  putr0#'getr0:Reg0                       \
          + 'getr0^2#putr0:Reg0                      \
          + 'getr0^3#putr0:Reg0                      \
          + 'getr0^4#putr0:Reg0                      \
          +  1:Reg0

bi   Reg0    Reg0' + lockreg0:Locked_Reg0

bi   Locked_Reg0   Illegal_Access0                   \
                 + putr0#releasereg0:Reg0            \
                 + 1:Locked_Reg0
```

```
**** Have to enumerate all of the possible illegal accesses.
bi   Illegal_Access0  'getr0:0                                          \
                    + 'getr0^2:0                                        \
                    + 'getr0^3:0                                        \
                    + 'getr0^4:0                                        \
                    + 'getr0#putr0:0                                    \
                    + 'getr0^2#putr0:0                                  \
                    + 'getr0^3#putr0:0                                  \
                    + 'getr0^4#putr0:0                                  \
                    +  putr0:0                                          \
                    + 'getr0#putr0#releasereg0:0                        \
                    + 'getr0^2#putr0#releasereg0:0                      \
                    + 'getr0^3#putr0#releasereg0:0                      \
                    + 'getr0^4#putr0#releasereg0:0


************************************************************************
**** Do the same thing for another register, Reg1, as in Reg0.
**** Should have used relabeling but when running the Workbench the output
**** is easier to look at if there is not alot of relabeling.
************************************************************************
bi   Reg1'   putr1:Reg1         \
          + 'getr1:Reg1         \
          + 'getr1^2:Reg1       \
          + 'getr1^3:Reg1       \
          + 'getr1^4:Reg1       \
          + 'getr1#putr1:Reg1   \
          + 'getr1^2#putr1:Reg1 \
          + 'getr1^3#putr1:Reg1 \
```

```
          + 'getr1^4#putr1:Reg1 \
          + 1:Reg1

bi   Reg1    Reg1' + lockreg1:Locked_Reg1

bi   Locked_Reg1   Illegal_Access1 + putr1#releasereg1:Reg1 + 1:Locked_Reg1

bi   Illegal_Access1   'getr1:0                                    \
                     + 'getr1^2:0                                  \
                     + 'getr1^3:0                                  \
                     + 'getr1^4:0                                  \
                     + 'getr1#putr1:0                              \
                     + 'getr1^2#putr1:0                            \
                     + 'getr1^3#putr1:0                            \
                     + 'getr1^4#putr1:0                            \
                     +  putr1:0                                    \
                     + 'getr1#putr1#releasereg1:0                  \
                     + 'getr1^2#putr1#releasereg1:0                \
                     + 'getr1^3#putr1#releasereg1:0                \
                     + 'getr1^4#putr1#releasereg1:0

************************************************************************
**** Define memory cells.  Easier than registers because no interlocks.
************************************************************************
bi   Mem0    putm0:Mem0           \
          + 'getm0:Mem0           \
          + 'getm0^2:Mem0         \
          +  putm0#'getm0:Mem0    \
          + 'getm0^2#putm0:Mem0   \
          +  1:Mem0
```

```
bi   Mem1    putm1:Mem1            \
          + 'getm1:Mem1            \
          + 'getm1^2:Mem1          \
          +  putm1#'getm1:Mem1     \
          + 'getm1^2#putm1:Mem1    \
          +  1:Mem1

*************************************************************************
**** Defining two memory cells and registers each.
*************************************************************************
bi Registers  (Reg0 | Reg1)

bi Memory (Mem0 | Mem1)

*************************************************************************
**** ------------>>> Instruction definitions <<<------------------ ****
*************************************************************************

*************************************************************************
**** There are 8 possible add instructions and the "nop" instruction.
**** Three operands with Reg0 or Reg1 possible for each operand.
*************************************************************************
bi ALU        add000#getr0#getr0#'putr0:DONE   \
          +   add001#getr0#getr1#'putr0:DONE   \
          +   add010#getr0#getr1#'putr0:DONE   \
          +   add011#getr0#getr1#'putr0:DONE   \
          +   add100#getr0#getr0#'putr1:DONE   \
          +   add101#getr0#getr1#'putr1:DONE   \
          +   add110#getr1#getr0#'putr1:DONE   \
```

```
          +   add111#getr1#getr1#'putr1:DONE   \
          +   nop:DONE

*************************************************************************
**** There are 8 load instructions and 8 store instructions
**** but to save states we are actually only going to enumerate the
**** ones where the base registers and the register being loaded or
**** stored are distinct.  That is you really don't usually want
**** to do a Load R0, (R0).
*************************************************************************
bi   Load_Store    Load + Store

*************************************************************************
**** insn[i,j,k] where i = dest reg, j = base reg, k = memory locn
*************************************************************************
bi   Load    load010#getr1#getm0#'lockreg0:'putr0#'releasereg0:DONE    \
           + load011#getr1#getm1#'lockreg0:'putr0#'releasereg0:DONE    \
           + load100#getr0#getm0#'lockreg1:'putr1#'releasereg1:DONE    \
           + load101#getr0#getm1#'lockreg1:'putr1#'releasereg1:DONE

bi   Store   store010#getr0#getr1#'putm0:DONE   \
           + store011#getr0#getr1#'putm1:DONE   \
           + store100#getr1#getr0#'putm0:DONE   \
           + store101#getr1#getr0#'putm1:DONE

*************************************************************************
**** The branch instruction has two outcomes: succede or fail.
**** Also, if the branch succedes, we can't have another branch
**** instruction.
*************************************************************************
```

```
bi Branch   Succede + Fail

bi Succede   bz#getr0:(((ALU + Load_Store) | 1:IPL) + bz:0) \
           + bz#getr1:(((ALU + Load_Store) | 1:IPL) + bz:0)

bi Fail      bz#getr0:IPL + bz#getr1:IPL

*****************************************************************
* Define a particle set of all possible instructions.
*****************************************************************
bpsi Insns add000 add001 add010 add011 add100 add101 add110 add111 \
           load010 load011 load100 load101                         \
           store010 store011 store100 store101                     \
           bz nop

bpsi Alu_Insns add000 add001 add010 add011 add100 add101 add110 add111 nop

bpsi FPU_Insns fadd000 fadd001 fadd010 fadd011     \
               fadd100 fadd101 fadd110 fadd111

*********************************************************************
**** Definition of dual ALU instruction issue.  Need appropriate
**** particle sets.  See dissertation chapter on Superscalar.
*********************************************************************
bpsi Putr0      putr0 getr0 getr1 add000 add001 add010 add011 add100  \
                   add101 add110 add111 nop
bpsi NotGetr0   putr1 getr1 add000 add001 add010 add011 add100        \
                   add101 add110 add111 nop
bpsi Putr1      putr1 getr0 getr1 add000 add001 add010 add011 add100  \
                   add101 add110 add111 nop
```

```
bpsi NotGetr1   putr0 getr0 add000 add001 add010 add011 add100 add101 \
                   add110 add111 nop

bi TwoAlus  ((ALU\Putr0 | ALU\NotGetr0) + (ALU\Putr1 | ALU\NotGetr1))

**************************************************************************
**** Floating-point registers
**************************************************************************
bi Freg1    lockfreg1:Locked_Freg1                                      \
         +  'getfr1#lockfreg1:Locked_Freg1                              \
         +  'getfr1^2#lockfreg1:Locked_Freg1                            \
         +  'getfr1:Freg1                                               \
         +  'getfr1^2:Freg1                                             \
         +   1:Freg1

bi Locked_Freg1   putfr1#releasefreg1:Freg1                             \
                + 1:Locked_Freg1

bi Freg2    lockfreg2:Locked_Freg2                                      \
         +  'getfr2#lockfreg2:Locked_Freg2                              \
         +  'getfr2^2#lockfreg2:Locked_Freg2                            \
         +  'getfr2:Freg2                                               \
         +  'getfr2^2:Freg2                                             \
         +   1:Freg2

bi Locked_Freg2   putfr2#releasefreg2:Freg2                             \
                + 1:Locked_Freg2

bi FP_Registers  (Freg1 | Freg2)
```

```
************************************************************************
**** Floating-point unit resource declarations
****
**** There is an adder, multiplier, and a divider.  All of which are
**** accessed exclusively through semaphores.
************************************************************************
bi  Resource    get_resource:Locked_Resource                        \
             +  get_resource#release_resource:Resource              \
             +  1:Resource

bi  Locked_Resource     release_resource:Resource                   \
                     + 1:Locked_Resource

bi  FPU    (Resource[get_multiplier/get_resource,                    \
                     release_multiplier/release_resource]           \
          | Resource[get_adder/get_resource,                        \
                     release_adder/release_resource]                \
          | Resource[get_divider/get_resource,                      \
                     release_divider/release_resource])



************************************************************************
**** Floating-point instructions Fdiv and Fmul not yet implemented
****
************************************************************************

************************************************************************
**** Yuck!! Enumerate all of the eight possible Fadd instrcutions.
************************************************************************
bi Float   fadd000#'lockfreg0#getfr0#getfr0:                            \
```

```
  'get_adder:1:1:'release_adder:                                    \
  'putfr0#'releasefreg0:DONE                                        \
                                                                    \
+  fadd001#'lockfreg0#getfr0#getfr1:                                \
  'get_adder:1:1:'release_adder:                                    \
  'putfr0#'releasefreg0:DONE                                        \
                                                                    \
+  fadd010#'lockfreg0#getfr1#getfr0:                                \
  'get_adder:1:1:'release_adder:                                    \
  'putfr0#'releasefreg0:DONE                                        \
                                                                    \
+  fadd011#'lockfreg0#getfr1#getfr1:                                \
  'get_adder:1:1:'release_adder:                                    \
  'putfr0#'releasefreg0:DONE                                        \
                                                                    \
+  fadd100#'lockfreg1#getfr0#getfr0:                                \
  'get_adder:1:1:'release_adder:                                    \
  'putfr1#'releasefreg1:DONE                                        \
                                                                    \
+  fadd101#'lockfreg1#getfr0#getfr1:                                \
  'get_adder:1:1:'release_adder:                                    \
  'putfr1#'releasefreg1:DONE                                        \
                                                                    \
+  fadd110#'lockfreg1#getfr1#getfr0:                                \
  'get_adder:1:1:'release_adder:                                    \
  'putfr1#'releasefreg1:DONE                                        \
                                                                    \
+  fadd111#'lockfreg1#getfr1#getfr1:                                \
  'get_adder:1:1:'release_adder:                                    \
  'putfr1#'releasefreg1:DONE
```

```
************************************************************************
**** The top-level "standard normal form" agent
************************************************************************

bi Instr    ((TwoAlus + ALU + Load_Store + Float) | 1:Instr) + Branch

bi CPU   (Registers | FP_Registers | FPU | Memory | Instr)\Insns
```

Bibliography

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge University Press, 1990.

[Bri88] E. Brinksma. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based upon the Temporal Ordering of Observational Behavior*, 1988. Draft International Standard ISO8807.

[BRR87] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Petri Nets: Applications and Relationships to Other Models of Concurrency.* Springer-Verlag, 1987.

[BS89] G. Birtwistle and P. A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[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.

[CJL89] C. Charlton, D. Jackson, and P. Leng. A functional model of clocked microarchitectures. In *MICRO-22, Proceedings of the 22nd Annual International Symposium on Microarchitecture*, 1989.

[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.

[Dav90] Bruce S. Davie. *Formal Specification and Verifiation in VLSI Design.* Edinburgh University Press, 1990.

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

[Hoa85] Tony Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

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

[LB90] M. Leeser and G. Brown, editors. *Hardware Specification, Verification and Synthesis: Mathematical Aspects.* Springer-Verlag, 1990.

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

[Mil85] George Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

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

[Mil93] Robin Milner. Elements of interaction. *Communications of the ACM*, pages 79–97, January 1993.

[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.

[SMB92] V. Stavridou, T. F. Melham, and R. T. Boute, editors. *Theorem Provers in Circuit Design*. North-Holland, 1992.