Abstract

HARCOURT, EDWIN ALAN. Formal Specification of Instruction Set Processors and the Derivation of Instruction Schedulers. (Under the Direction of Jon Mauney and Thomas K. Miller, III.)

We present two techniques for formally specifying an instruction set processor at the *programmer's view* — the *architecture* view and the *timing* view. From the timing specification we show how to derive an instruction scheduler for the processor.

One technique addresses architecture specification, that is, the information required to write correct programs. At the architectural level we present a functional semantics that captures the property that instructions are functions from processor state to processor state.

The second specification technique addresses the programmer's view of the timing of the processor, that is, the needed information required to write temporally efficient programs. 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. Putting our specification to use, we show how to construct an instruction-scheduler from the specification by deriving appropriate parameters needed for instruction scheduling. These parameters include instruction latencies, illegal instruction combinations, resource constraints, and instructions that may be issued in parallel.

The Formal Specification of Instruction Set Processors and the Derivation of Instruction Schedulers

by

Edwin Alan Harcourt

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh
1994

Approved by:

Co-chair of Advisory Committee	Co-chair of Advisory Committee

Biography

Edwin A. Harcourt received a B.S. in Computer Science from the State University of New York at Plattsburgh in 1986. In 1989 he received an M.S. in Computer Engineering (1989) and in 1994 a Ph.D. in Computer Science both from North Carolina State University.

Contents

List of Figures

Chapter 1

Introduction

A specification of a system is a precise description of the desired behavior of the system which any implementation should provide. Specifications have traditionally been expressed in natural language which, unfortunately, can be vague and hence lead to imprecision and ambiguity. To remedy this problem many specifications are now being expressed *formally* (*i.e.,* using some mathematical framework such as first-order logic or algebraic techniques). Once a formal specification has been written, several tasks can be performed: the specification may be used to help derive an implementation, construct a simulator for the system, or aid in writing documentation.

This dissertation addresses the problem of specifying an instruction set processor in a way that is useful for building compilers. An *instruction set processor* is what we normally think of as a microprocessor or CPU. Specifically, we address formally specifying a processor at two levels: the architecture level and the instruction timing level. Also, we show how a timing specification can be used to derive an instruction scheduler for the processor.

Formal processor specifications are valuable for a variety of reasons.

- Formal specifications aid in the design process. They require the user to thoughtfully plan and design the system.

- Processor verification requires a formal specification in order to carry out proofs of correctness.

- If the specification language is executable (and both SML and SCCS are) then a timing level simulator is *automatically* available.

- High-level synthesis (whether compilers or hardware) requires some form of a specification.
- Formal specifications are used for precise documentation.

Instruction set processors are best viewed hierarchically at various levels of abstraction. A common partitioning of the hierarchy is as follows:

- The *instruction set architecture* (or just 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, buses, internal latches, etc.
- The *logic* level contains the low level implementation details of the functional units.

When discussing or making an argument about a processor one has to be clear about the level of abstraction at issue. The user of a processor is usually concerned with the architectural level, since the user must have this information to write correct programs.

1.1 The Architecture Level

At its highest level, an architecture is an *abstract data type* where memory and registers (*i.e.,* the machine *state*) constitute the data and the machine instructions are the operations defined on the data. At this level, instructions are essentially functions that map states to states. Usually, a simple *register transfer language*, or RTL, notation suffices to describe the computational effect(s) of an instruction. For example, the instruction

Add $\mathtt{R}_i$, $\mathtt{R}_j$, $\mathtt{R}_k$

may be described by the register transfer statement

```
Reg[i] := Reg[j] + Reg[k].
```

Until recently, there has been no language available for specifying processors at the architectural level. Cook [29] remedied this by designing a language specifically for designing instruction sets. Since the architecture level provides a functional view, where instructions are functions from processor state to processor state, it can be formalized using a functional language. In Chapter 3 we present a functional semantics of a processor at the architecture level using the language SML [78].

1.2 Organization Detail vs. Timing Detail

Besides writing correct programs, a user would also like to write efficient programs. Hence, the user needs more information than is contained in an architecture specification. 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 (because (2) needs to wait for `R1` to be loaded from memory by (1)), which wastes cycles and causes the pipeline to stall. On some processors the sequence is illegal which causes the value of `R1` to be undefined during instructions (1) and (2). 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 or make the sequence valid.

When one instruction can interact with another and alter the computational effect in undefined ways (*e.g.,* the load instruction above) or when we wish to capture timing information so that we can write efficient programs, then viewing instructions as functions is no longer sufficient. We consider this point further in Chapter 2.

In many modern instruction set processors, the temporal and concurrent properties of the instructions are visible to the user of the processor. Consequently, such properties should be included in a behavioral processor specification. We present a technique for formally describing, at a *high-level*, the timing properties of pipelined, superscalar processors [76, 55]. 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 (superscalar). As our mathematical formalism we use SCCS (**S**ynchronous **C**alculus of **C**ommunication **S**ystems), a synchronous process algebra designed for specifying timed, concurrent systems [68, 67]. After showing how to specify a processor we parameterize the instruction scheduling problem and demonstrate how to derive these scheduling parameters from the specification, essentially yielding an instruction scheduler for the architecture.

1.3 Static Instruction Scheduling

Once a processor has been defined at the instruction timing level there are several ways in which the specification can be used. A hardware designer could use the specification as the *requirements specification* that any implementation must meet.

However, a particular level of abstraction is both a *specification* of a lower level and an *implementation* of a higher level. For example, the organization level can be viewed as an implementation of the architecture level and as a specification for the logic level. In a similar vein our timing specification could also be viewed as an implementation of the processor for a higher level, the compiler level. It is in this way that we use our timing specification: by extracting scheduling information from the timing specification and putting it in a form suitable for use with an "off the shelf" instruction scheduler.

A subsequent goal of this research, then, is to utilize our specification by generating instruction scheduling information (*e.g.,* latencies, resource requirements) directly from the specification. We develop algorithms that analyze SCCS descriptions and yield the instruction scheduling information. The algorithms depend on the formal operational semantics of SCCS, which is defined in terms of an abstract machine known as a *labeled transition system.*

1.4 Outline of Dissertation

The rest of this dissertation is organized as follows:

- There have been a variety of methods used to specify low-level digital hardware, both formal and informal. Chapter 2 reviews this related research and further motivates our choice of using the synchronous process algebra, SCCS.

- Instructions at the highest level are, mathematically, state transformation functions. Chapter 3 gives a functional semantics to an example architecture, using the functional language SML.

- In Chapter 4 we motivate the need for expressing programmer level timing constraints in processor specifications.

- In Chapter 5 we introduce SCCS both formally (using structural operational semantics) and informally (with several examples of using SCCS to describe some simple combinational and sequential circuits).

- Chapter 6 formally specifies, using SCCS, the instructions and the their timing constraints of a hypothetical, yet realistic, RISC style processor.

- Chapter 7 shows how we can use SCCS to specify various Superscalar configurations of our example architecture.

- The operational semantics of SCCS maps SCCS programs to abstract machines. Our SCCS specification is, therefore, executable, in terms of the abstract machine it represents. In Chapter 8 we discuss how the behavior of our architecture is simulated.

- Chapter 9 gives a brief introduction to instruction scheduling and sets the stage for deriving instruction scheduling information from the SCCS specification.

- Chapter 10 describes how to derive instruction scheduling parameters, including instruction latencies, resource constraints, illegal instruction sequences (*e.g.,* delayed loads) and multiple instruction issue.

- Chapter 11 concludes and discusses future research.

- For purposes of introducing SCCS with some simple examples, Appendix A gives the SCCS implementation (using the Concurrency Workbench) of some simple digital circuits.

- Appendix B gives an SCCS implementation of our example RISC architecture using the Concurrency Workbench.

Chapter 2

Mathematical Specifications and Related Research

At first, researchers hoped that there might exist some one unique ideal specification language. But what could it be? Perhaps some kind of logic, say full first order logic? Or perhaps Horn clause logic? Or equational logic? Or LCF? What about the lambda calculus? Or perhaps we should take a state transition approach ...? But none of these seemed to work.

(Joseph Goguen – *One, None, a Hundred Thousand Specification Languages*)

There are many formalisms available and currently being applied for specifying the intended behavior and/or semantics of computer hardware and architectures. In this chapter we review hardware description languages (*e.g.*, Verilog and VHDL) [65, 3, 29, 5, 6, 40], formal specification languages such as first-order logic [18], higher-order logic [47, 11, 91, 9, 61, 10], temporal logic [80, 94, 43], equational algebra [90, 42, 39], relational algebra [85], concurrency formalisms [53, 67, 68, 69, 66, 17, 34, 44, 8, 16, 83], the Z specification language [13, 12, 87], type theory [54, 48], and the lambda calculus [45, 46, 88, 75, 21]. We also review compiler code generator generators which also provide specification languages for describing processors [1, 37, 33, 81, 15].

2.1 Hardware Description Languages

The term "hardware description language" (or HDL) is not a well-defined term. Traditionally it means "any language used to specify hardware". But this definition is too broad as this would include imperative programming languages such as C, Pascal, and Ada and abstract formalisms such as Constructive Type Theory [54] and Category Theory [85]. Usually an HDL refers to languages such as VHDL and Verilog which are imperative style programming languages extended with primitives and libraries that aid in describing hardware.

There are a variety of HDL's such as Verilog [92], VHDL [62], ISPS [5], and ELLA [72]. All of them suffer in that they lack a formal definition. HDL's are extensions of imperative programming languages and their problems can be traced to the problems of imperative languages — such as aliasing and side-effects. It has also been suggested that HDL's be abandoned altogether and that imperative programming languages be used. Using Ada as an HDL has been addressed in [6, 40]. Also, while HDL's are useful for simulating low-level digital hardware they are unsuitable for use in high-level processor specification as pointed out by Cook [29].

VHDL is one of the more popular hardware description languages [62]; Figure 2.1 shows the VHDL code that describes the behavior of a simple inverter. There is nothing too surprising about this description other than that it is a lot of code for specifying a simple inverter. A more serious problem, however, is that, since VHDL is not formally defined, attempting to ascertain the behavior of even simple circuits (let alone more complicated circuits) is dependent upon individual implementations of VHDL. For example, the keyword, **transport** in the inverter specification specifies that the delay in the **after** clause corresponds to the propagation delay associated with passing a value through a wire. If the keyword **transport** is omitted then delay corresponds, not to a propagation delay in a wire, but to the "hold time", that is, the amount of time the signal must persist in order for it to be considered as a signal. Other timing complications are introduced by the various forms of the **wait** statement on processes – **wait**, **wait on**, **wait for**, **wait until**, **wait** *time-expression*.

Another criticism frequently leveled at VHDL is that the timing primitives of VHDL are low-level and overly concrete as the programmer is required to specify timing constraints in nanoseconds or femtoseconds (one femtosecond is the smallest unit of time in VHDL). If one wishes to view time more abstractly in terms of, say, clock cycles then a "clock process" must be constructed that models a clock and every process must then perform their actions based on a signal that this clock process emits. These same criticisms apply to Verilog, the other most widely used HDL, as it is similar to VHDL.

```
— Define some enumeration types.
type Logic is ('0','1');
type Delay_flag is (Zero_delay,Const_delay);

— The interface to the inverter.
entity INV is
  generic (Time_flag: Delay_flag);
  port (X: in Logic;Y: out Logic);
end INV;

— Specify the behavior of the inverter.
architecture Behavior of INV is
begin
 p0: process
  variable delay: Time := 0 Ns;
  begin
    if Time_flag = Const_delay then delay := 0.83 Ns;
    end if;
    if X = '0' then
        Y <= transport '1' after delay;
    else
        Y <= transport '0' after delay;
    end if;
    wait on X;
  end process;
end Behavior;
```

Figure 2.1: **A VHDL description of an inverter.**

2.2 Formal Specification Languages

A formal specification language is one that is based on some rigorous mathematical framework. The specification then becomes amenable to analysis using the mathematical tools of the framework. For example, if we specify a system using first-order logic we may then be able to use resolution to perform proofs about the system.

Formal specification languages fall into two broad categories: *functional* (or relational) and *reactive.* A *functional* view of a system is one that maps an initial state to a final state. (Or in the relational view, mapping an input to a *set* of outputs). This view is appropriate for programs that accept all of their inputs when the program begins executing and yields its output when the program terminates [80]. For example, a compiler maps source programs to target programs.

Some systems have no final output and termination is viewed as a catastrophe rather than as a virtue. The behavior of these *reactive* systems is defined in terms of the ongoing interaction with their environment. For example, an operating system should not terminate and has no final output but has an ongoing interaction with users to service their requests. Any concurrent system can be viewed as reactive because, even if its overall role is functional, the system is composed of interacting components [80]. It is in this way that we view a microprocessor: reactively as a system of interacting components (instructions, registers, and functional units).

A recent overview of formal methods applied to the design and analysis of digital systems can be found in [90].

2.2.1 Sequential Formalisms

We use the term *sequential* to mean a formalism that does not include primitives to specify concurrency or time. Sequential formalisms include functional languages (based on the λ-calculus), algebraic theories, first-order logic, and higher-order logic.

Functional Specification

There has been some research into specifying hardware and architectures using functional methods. Paillet [75] presents a functional semantics of a CISC-style microprocessor at the instruction set level. No attempt was made to specify instruction timing properties or instruction interaction; rather the intent of the research was to use the specification as a basis to verify an implementation, although this was not done.

```
fun HalfAdder(a,b) = (a <> b, a  andalso b)

fun FullAdder(Cin, a, b) =
    let
       val (Sum'', Carry') = HalfAdder(a,b)
       val (Sum', Carry'') = HalfAdder(Cin,Sum'')
    in
        (Sum', Carry'  orelse Carry'')
    end
```

Figure 2.2: **A full-adder specified in SML.**

Charlton [21] presents a technique of introducing a clock into a functional specification by using "lazy lists" to represent an infinite stream of clock pulses. The research was limited to a consideration of how one could introduce a clock in a functional model; nothing was subsequently done with the specification.

Gordon [45, 46] represents finite-state systems using the λ-calculus. He then showed how one could verify circuits in the λ-calculus using *fixpoint induction*. As an example, Figure 2.2 uses the functional language SML to specify a full-adder from two half-adders.

Of the various abstraction levels of a processor the functional view is appropriate if we wish to view it at its highest level — the architecture. In this view, instructions are functions from states to states. However, a function that represents an instruction specifies *final values* of an instruction and not how this value was computed nor how long it took to compute. This view makes it difficult to specify timing or instruction interaction.

HOL — Higher-Order Logic

Probably the most widely used formalism for formally specifying hardware is *higher-order logic*; in particular, the logic embodied in the HOL system of Gordon. (See Gordon [47] for an introduction to HOL and Melham [64] for using HOL in hardware verification.) Higher Order Logic is an extension of first-order logic in which variables can range over functions and predicates. HOL uses the lambda calculus to specify functions and, consequently, subsumes functional methods.

HOL is very expressive and has been used to describe other formalisms such as a subset of VHDL [11], CCS [68] (the asynchronous version of SCCS), the π-calculus[70] (higher order CCS), and Hoare's CSP [53]. This suggests that HOL is a good meta-language and is useful for defining other formalisms within HOL. However, there is a sense in which HOL is too expressive, according to Goguen [42]:

> "most logics are too expressive! We need more restricted logics ... in order to get *executable* specification languages that can be used for rapid prototyping and even efficient programming. And we certainly want to avoid specifying systems that are unimplementable because they involve uncomputable functions or relations".

Higher order logic is undecidable. That is, it is undecidable whether a statement in higher order logic is true. Consequently, the HOL system is not a theorem prover but a proof checker and provides a "support environment" for carrying out proofs in HOL by hand. Moreover, HOL can not directly be used to simulate the system being specified as there is no notion of the operational behavior of an HOL statement [90].

As in functional methods, neither time nor concurrency are primitives in the logic but are encoded. However, because of HOL's expressiveness it is somewhat easier to represent both time and concurrency than in a functional approach. For example, to represent concurrency in HOL we do not need to talk about sets of final values but simply define two events α and β to occur in parallel if α and β are both true at time t, that is, if $\alpha(t) \wedge \beta(t)$ holds. So logical conjunction is used to compose two elements in parallel.

Typically, time is represented using the natural numbers ($0, 1, 2, 3, \ldots$ usually abbreviated ω). Consider two ports, `in` and `out`. A port can be represented as a function from time to booleans.

$$\mathtt{in}, \mathtt{out} \in \omega \rightarrow \mathit{bool}$$

We use HOL's logical operations (the standard $\vee, \wedge, \forall, \exists, =$) to specify constraints on the port values. For example, the invariant that the value at port `out` at time $t+1$ is the same as the value on port `in` at time t is specified as,

$$\forall t \in \omega.(\mathtt{out}(t+1) = \mathtt{in}(t))$$

Equation 2.1 represents a one bit unit delay element where the output at time t is the same as the input at time $t-1$. Equation 2.1 also specifies that, at time 0 (power up), the contents of the register is 0.

$$\mathit{Reg}(\mathtt{in}, \mathtt{out} \in \omega \rightarrow \mathit{bool}) \overset{\text{def}}{=} \forall t \in \omega.\mathtt{out}(t) = ((t = 0) \Rightarrow 0 \mid \mathtt{in}(t-1)) \tag{2.1}$$

The notation $\mathit{cond} \Rightarrow \mathit{true} \mid \mathit{false}$ is a conditional expression.

Equation 2.2 "connects" two registers together creating a delay circuit that inputs a bit at time t and outputs it at time $t+2$. The types of the ports `in`, `out`, ℓ are omitted for readability but are functions from time to booleans as in Equation 2.1.

$$\mathit{TwoRegs}(\mathtt{in}, \mathtt{out}) \overset{\text{def}}{=} \forall \mathtt{in}.\forall \mathtt{out}.\exists \ell.(\mathit{Reg}(\mathtt{in}, \ell) \wedge \mathit{Reg}(\ell, \mathtt{out})) \tag{2.2}$$

Equation 2.2 shows how, using conjunction, two registers are instantiated in parallel. Existentially quantifying the variable ℓ makes an internal connection between the two registers.

In [26] Cohn describes the Viper microprocessor using HOL. The Viper is a microcoded processor that does not include any instruction-level parallel properties. Also, in [52], Herbert discusses specifying and verifying microcoded microprocessors using HOL in general. Again, the techniques do not address instruction-level parallelism.

2.2.2 Formalisms of Concurrency and Time

There are well developed formalisms for explicitly dealing with reactive systems. The most common formalisms are process algebras and modal and temporal logics.

Process Calculi

There are a variety of formalisms for specifying asynchronous and/or synchronous concurrent systems including Petri Nets [16], CCS [68, 69], SCCS [68, 67], ACP [4], CIRCAL [66, 34], HOP [44], ESTEREL [8], the π-calculus [70], and CSP [53].

Process algebras have been used to give a semantics to a communications protocol language, LOTOS [17]; a parallel object oriented language, POOL [4]; a computer integrated manufacturing system [4]; to low-level digital hardware [66]; and biological ecosystems [95]. The various calculi can be divided into asynchronous and synchronous languages. The synchronous languages can be considered "more expressive" as it is typically possible to embed an asynchronous calculus within a synchronous one. For example, the asynchronous calculus CCS is definable within the synchronous calculus SCCS [67].

In an asynchronous formalism parallelism is reduced to non-determinism (or interleaving). For example, let

- $A \times B$ mean processes A and B execute in parallel;
- $A + B$ mean non-deterministically execute either A or B;
- $a.A$ mean execute the action a and then become the process A.

In an asynchronous formalism if the process A can do some action a and become the process A' we will write $A \xrightarrow{a} A'$; the process B can do some action b and become the process B' then the process $A \times B$ is equivalent to the process $a.(A' \times B) + b.(A \times B')$. That is, the process $A \times B$ can execute A for a step or B for a step but not both at the same time (forgetting for the moment the possibility that A and B might wish to communicate).

This is not the correct way to view clocked hardware as a circuit does not operate by interleaving the execution of the subcomponents of the circuit. This problem of asynchronous formalisms has been pointed out by Berry [8]. For example, if we construct a half-adder from an or-gate and an xor-gate the circuit does not operate by computing the result of the or-gate and then computing the result of the xor-gate (or vice-versa) but by *really* computing both the *or* and the *xor* at the same time. Consequently, we dismiss asynchronous calculi (*i.e.*, CCS, ACP, LOTOS, CSP, Petri Nets) as they can not directly represent the global clock associated with hardware. They have, however, been applied for representing asynchronous circuits, a current area of active research [19].

Synchronous Process Algebras

Several of the algebras cited above (SCCS, ESTEREL, CIRCAL, HOP) are designed for specifying synchronous systems, in other words, systems that have a global clock where events occur based on the clock and processes execute in lock step. Given the above example, the process $A \times B$ is equivalent to the process $a \star b.(A' \times B')$ where "$\star$" is some binary operation on actions. This means that both A and B perform their actions on the same clock cycle. Here, $\times$ is now being used as a synchronous parallel operator rather than an interleaving one.

The synchronous languages are all related in that SCCS is the father of them all. Our choice of using SCCS over the other synchronous formalisms is largely pragmatic. SCCS:

- is the most widely used and known
- is the most mature and formally developed
- has a large body of research to draw upon for doing formal analysis
- has tools available for carrying out analysis [25]
- has a simple syntax
- has a clear and concise operational semantics

- is compositional, allowing larger systems to be composed of smaller ones.

No one has yet applied SCCS to the problem of hardware specification.

Modal and Temporal Logic

There are some logics that deal explicitly with time and concurrency [43, 94, 80]. Modal logic extends classical propositional or first-order logic by by including two operators $\Box$ and $\Diamond$ that represent "necessity" and "possibility". (Really, the logic only need include one of the operators as one can always be defined in terms of the other using negation.) In a temporal context these operators are usually interpreted as "henceforth" and "eventually".

For example, the temporal logic of Pnueli [80] has a temporal operator $\Box$ that means "henceforth" and an operator $\bigcirc$ that means "next". Consider the example from the discussion on HOL where we wished to represent the invariant that the value on a port `in` at time t appeared at port `out` at time $t+1$. This can be represented in temporal logic by

$$\Box((\bigcirc\mathtt{out}) \Leftrightarrow \mathtt{in}).$$

This says that "from now on, the next output is equal to the current input."

It turns out that there is a strong connection between modal (temporal) logics and process calculi. In fact we will use a particular modal logic, the μ-calculus, to check whether our microprocessor, specified in SCCS, has certain properties. This connection between modal/temporal logic and process algebra was first established by Pnueli [80]. Briefly, the relationship is this: the operational semantics of SCCS is defined in terms of a labeled transition system (defined in Chapter 5). In turn, the labeled transition system is used as a model for formulae in the modal logic [94]. That is, the abstract machine generated by the operational semantics of one language (SCCS) is used as a model of another (the modal μ-calculus). In Chapter 10 we will discuss this in more detail.

2.3 Compiler Code Generator Generation Languages

The compiler research area of *retargetable code generation* offers another method of specifying an architecture. A retargetable code generator uses a general code generation strategy (often some type of pattern matching as in Twig [1]) parameterized by the properties of the architecture which are encapsulated in a *machine description.* However, the term "machine description" is misleading as the machine description does not describe a machine as much as it describes an intermediate language (IL). A better term would be "code generator description" as one would expect that a machine description describe an architecture's instructions directly by mapping them to some other language. But what a machine description describes is how the compiler's *intermediate language* maps to the machine's instructions. That is, we would expect

$$\textsf{Machine Description: Instructions} \longrightarrow \textsf{IL}$$

but what we really have is

$$\textsf{Machine Description: IL} \longrightarrow \textsf{Instructions}$$

This problem has been remedied by Giegerich [41] who proposes a way of *inverting* the machine description. Until recently, these retargetable code generation systems have dealt only with *instruction selection* and not instruction scheduling. That is, they have not dealt with instruction timing properties.

There has been some work by Bradlee [15] and Proebsting and Fraser [81] (called PF from here on) in compiler code generator description languages where instruction timing properties are inserted into the machine description to try and address the appropriate user-level view of instruction timing. Bradlee was the first to propose extending machine descriptions with instruction timing properties. The specification language, Marion, was subsequently used to perform retargetable instruction scheduling and as a basis for integrating register allocation with instruction scheduling. The goal of PF was to generate, from a specification of the resource requirements of a processor, a space-efficient state transition graph that can be used for instruction scheduling. Consequently PF's machine description only includes a method of specifying resource requirements and is not as expressive as Bradlee's. Since our primary interest is on specification and because Marion is more expressive than PF's description language we only consider Bradlee in detail.

2.3.1 Resource Requirements

From the time an instruction is fetched to the time it has completed typically takes several cycles. On each cycle, an instruction uses a subset of the processor's resources. The specification languages of PF and Bradlee specify these resource requirements directly. For example, the MIPS/R4000 processor's [56] floating-point unit has as its resource set an unpacker, shifter, adder, rounder, two-stage multiplier, divider, and two exception resources. These are abbreviated by Bradlee as `U`, `S`, `A`, `R`, `M1`, `M2`, `D`, `E1`, and `E2` respectively.

Using Marion's syntax the resource constraints of the MIPS R4000 processor's single precision floating-point divide instruction, `div.s`, is specified as

```
        [U; S,A; S,R; S; D*13; D,A; D,R; D,A; D,R; A; R]
```

which means that the `div.s` instruction uses the

- unpacker on the first cycle
- shifter and adder on the second cycle
- shifter and rounder on the third cycle
- shifter on the fourth cycle
- divider only for thirteen consecutive cycle starting on cycle five

and so on.

There are two main criticisms of Marion. The first criticism is that it is not formal as we only have intuitive notions of what the operators ";", ",", and "*" mean. This precludes Marion from being used for anything except for use with Bradlee's own algorithms. That is, in order to use Marion for other tasks, which we expect from a specification (*i.e.,* verification, simulation, documentation) the specification language needs to be given a formal definition.

The second criticism of Marion and this approach is that it is sometimes a too low-level way of specifying the timing constraints of an instruction. For example, the Marion specification of the resource requirements for the MIPS/R3000 processor single precision integer addition instruction, `add`, is given as:

```
        [IF; RD; ALU; MEM; WB]
```

which specifies that the instruction uses a different resource on each clock cycle, cycles 1 through 5. More specifically, `IF`, `RD`, `ALU`, `MEM`, `WB` represent the instruction fetch, register read, execute, memory access, and register write back stages of the integer pipeline respectively (though this information can only be inferred by the reader from already knowing how instruction pipelines are usually structured).

Yet, there are *no* timing constraints on the MIPS's add instruction and, in a program, the add instruction can be used in any context. This is impossible to ascertain from the description because we do not know precisely what the description means. As it turns out, the reason that there are no constraints is that there are other resources — forwarding hardware — which are left unspecified. It is not possible to specify these extra resources as it does not suffice to just enumerate the forwarding hardware as resources in the specification language. One must specify *how* the forwarding resources are *connected* to the other resources and how they interact. This capability is currently beyond the specification languages of both PF and Bradlee.

2.3.2 A Marion Extension

To remedy the situation where Marion's resource vector does not accurately reflect the timing properties of the instruction, Bradlee associates a delay value with the instruction. In his notation, the add instruction is now specified as:

```
        [IF; RD; ALU; MEM; WB] (1)
```

Here, the `(1)` represents the latency of the instruction, or how many cycles need to pass before a subsequent instruction can use the result produced by the add. Since the latency is only one, the very next instruction may use the result. Now, in this case, the resource vector does not yield any useful information about the instruction's timing constraint, which is contained in the extra parameter — the delay value `(1)`. So sometimes specifying resources is unnecessary as the timing constraints of the instruction do not depend on them. However, if an instruction's timing constraints do depend on resource constraints then resources need to be included in the instruction specification. While omitting unproblematic resources is not possible in Marion, we will see that our specification technique using SCCS will allow us to specify latency values with or without resource information depending on whether it is needed or not.

The situation is actually more complicated. Bradlee also associates with each instruction a value that represents the number of delay slots for the instruction, which, for most instructions is zero. For example, the statement

```
        [IF; RD; ALU] (2,1)
```

represents the timing constraints of a delayed branch instruction. The tuple `(2,1)` represents the latency and the number of required delay slots respectively. Furthermore, a negative value for the number of delay slots indicates that the instructions in the delay slots are only executed if the branch is taken. A positive value indicates that the instructions are always executed.

2.3.3 Other Processor Features

Some processors use priority schemes to resolve resource conflicts For example, the Motorola 88000 uses gives priority of certain instruction over others if they both require the data bus on the same cycle [2]. Expressing this is not possible in the framework of Bradlee and PF but is in SCCS (and process algebras in general) [20].

Also, superscalar processors execute more than one instruction on each cycle. Specifying this will be possible in SCCS but is not within the frameworks of Bradlee and PF.

2.4 Summary — The Specification Language Zoo

In this chapter we have reviewed two areas of processor specification each adopting a particular viewpoint: 1) hardware description languages (HDLs) and 2) compiler code generator machine descriptions. We argued that the language we require should be formal and be able to explicitly specify the temporal and concurrent properties of instructions. We then argued that no one has addressed formally specifying instruction timing at the appropriate user level.

In the compiler code generation view we indicated that, while there were attempts to specify architectures at the correct user's timing level, the attempts were ad hoc and informal and were only partially successful at hitting the correct level of abstraction. With these criticisms at hand we then justified our choice of using SCCS as our specification language.

Figure 2.3 shows a classification of the various languages and formalisms from the point of processor specification as discussed in this chapter. From our perspective, every language is a processor specification language which is categorized as either a hardware description language or a code generator description language (Marion and Twig). VHDL, Verilog, and ELLA are informal hardware description languages. SML and Haskell are two particular implementations of the λ-calculus. HOL is an implementation of higher-order-logic. CTL (Computational tree logic, a particular temporal logic) and the μ-calculus are two particular modal logics that are related to process algebra in that they are used to describe reactive systems. SCCS, CIRCAL, and ESTEREL are synchronous process algebras while CCS, Petri Nets, and CSP are asynchronous process formalisms. The tree diagram represents the current situation and does not mean to imply that there will never be such a thing as a formal code generator description language. Also there has been some work at giving VHDL a formal semantics [74] (currently only subsets of VHDL) which could move it into

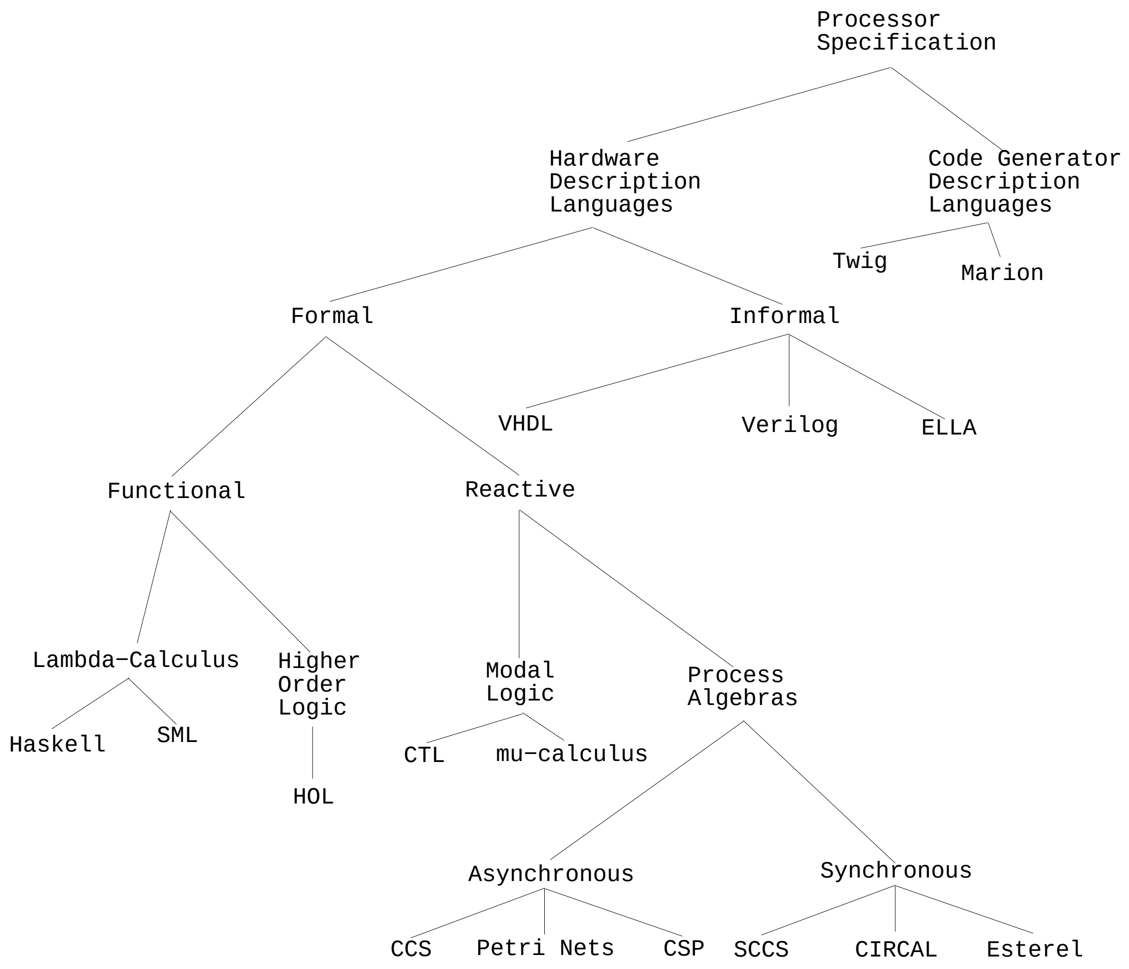


Figure 2.3: **The classification of specification languages.**

the realm of a formal hardware description language.

Chapter 3

Functional ISA Specification

In this chapter we present a methodology for formally specifying a denotational (or functional) semantics of an instruction set architecture. More specifically, since there are basic architectural operations common to all architectures (*e.g.*, reading and writing registers and memory) we create an *abstract data type* of primitive architectural operations suitable for representing a wide variety of architectures. This functional view specifies final computations of instructions and not instruction timing properties.

3.1 Denotational Semantics

A denotational semantics of a programming language represents each syntactic construct of the language in terms of mathematical objects (*e.g.*, sets, functions, relations). We can view an architecture as a low-level programming language (but a high-level view of a processor) in which each instruction is a syntactic object that represents an operation on the state of the processor. That is, an instruction is a state transformation function

$$\mathcal{I} : \mathit{State} \rightarrow \mathit{State}$$

and *State* is the type of a function from locations (register numbers or memory addresses) to values (integers).

$$\begin{aligned} \mathit{State} &= \mathit{Locations} \rightarrow \mathit{Values} \\ \mathit{Locations} &= \mathit{RegNums} + \mathit{Addresses} \\ \mathit{Values} &= \mathit{integer} \end{aligned}$$

If ρ is the state function then $\rho(x)$ is the value stored at location x. A new state can be built from an old state ρ with the state update function $\rho[x \mapsto d]$ which means "the state ρ with x updated to d". Given this definition of state we can begin to describe the semantics of individual instructions. For example, the MIPS `Add` instruction is defined by

$$I[\![\ \texttt{Add R}_i\texttt{, R}_j\texttt{, R}_k\ ]\!]\rho = \rho[\texttt{R}_i \mapsto \rho(\texttt{R}_j) + \rho(\texttt{R}_k)]$$

The notation $I[\![\ \texttt{-}\ ]\!]$ represents the valuation function. The metabrackets "$[\![\ \texttt{-}\ ]\!]$" surround the syntax of an instruction and separate the language being defined from the defining language. The state function ρ is an argument to the valuation function and an updated valuation function is returned. This coincides with our definition of instructions being functions from state to state.

The advantage of a denotational semantics is that it provides a facility to reason about specifications and to mathematically prove properties about the specification. For example, consider the definition of an instruction that increments by one the value stored in the register

$$I[\![\ \texttt{Inc R}_i\ ]\!]\rho = \rho[\texttt{R}_i \mapsto \rho(\texttt{R}_i) + 1]$$

Given the definitions of `Add` and `Inc` it is now an easy matter to prove that

$$\texttt{Inc R}_i \equiv \texttt{Add3 R}_i\texttt{, R}_i\texttt{, 1}$$

by checking that

$$I[\![\ \texttt{Inc R}_i\ ]\!] \equiv I[\![\ \texttt{Add3 R}_i\texttt{, R}_i\texttt{, 1}\ ]\!]$$

By substituting each side of the "$\equiv$" with its definition we see that

$$\rho[\texttt{R}_i \mapsto \rho(\texttt{R}_i) + 1] \equiv \rho[\texttt{R}_i \mapsto \rho(\texttt{R}_i) + 1]$$

This property of being able to substitute equals for equals is known as *referential transparency* and is a property of functional specifications not shared by traditional imperative programming languages and some hardware specification languages such as VHDL or Verilog [9].

3.2 Semantic Algebras

We make our denotational specifications more readable and easier to work with by creating a library of architectural primitives. This library is essentially an abstract data type and is sometimes called a *semantic algebra* [84]. The operations defined by the semantic algebra are called *semantic actions.* Rather than specify the architecture directly as a denotational semantics we first translate it into semantic actions defined by the semantic algebra. The semantic algebra is a *register transfer language*, or RTL, which is then specified denotationally.

This specification technique is unique in that:

- Specifying the semantic actions occurs once. When this is done, many architectures can be specified using the predefined actions.

- The semantics of the actions need not be specified denotationally, but could be specified using any formal semantic method (*e.g.* operational or algebraic semantics). This gives our semantic specification an abstractness and modularity that is absent from other semantic specification methods. This style of semantics is called a *separated semantics* [60].

3.3 The Semantics of ISA's

The denotational semantics of an ISA will comprise the three parts of a semantics: a syntactic domain, a semantic domain, and a valuation function.

- **Syntactic domain** — The syntactic domain is the instruction formats of the architecture being specified.

- **Semantic domain** — The semantic domain is a set of *semantic actions.* In this paper we use actions that implement a *register transfer language* (RTL).

- **Valuation function** — A valuation function specifies how an item in the syntactic domain maps to an item in the semantic domain. In this case, the valuation function formally describes an instruction in terms of semantic actions.

To illustrate the process we use the PDP-11 as an example ISA.

3.3.1 Notation

It is common to use a modern functional language to specify a denotational semantics [60], and in our case we will use Standard ML. While SML is not completely referentially transparent, due to its imperative features, its semantics is formally specified. Also, if we restrict ourselves to the purely functional subset we keep referential transparency.

Traditionally, the syntactic domain is specified with a context-free grammar. Since the syntax of instructions is simple, SML's algebraic data type constructor (**datatype**) is used to specify the abstract syntax.

The semantic domain is a set of semantic actions that represent basic architectural operations and, as was mentioned before, is RTL. The semantic actions constitute a language with a syntax and a semantics and, to be complete, both must be specified.

The valuation function in a traditional denotational description will be described using SML functions. Since our semantics is directly coded into SML, a simulator for the architecture is immediately available.

3.3.2 The Syntactic Domain

The syntactic domain consists of the ISA's instruction formats. Figure 3.1 shows the SML definition of the syntax of the PDP-11 instruction formats. For example, the PDP-11 instruction `Add R0,(R1)+` is represented by the SML construct,

```
TwoOp1(ADD, RegDirect(0), AutoInc(1))
```

and the instruction `Add 10(R2), #10` is represented by,

```
TwoOp1(ADD, Indexed(10,2), Immediate(10))
```

Here, **TwoOp1**, **RegDirect**, **AutoInc**, **Indexed**, and **Immediate** are type constructors (or tags).

3.3.3 The Semantic Actions

This section describes the syntax of the language of semantic actions (RTL). We defer their implementation until Section 3.4. The choice of the semantic actions is important: they must be capable of specifying the low-level semantics of a machine instruction and also be suitable as an intermediate representation for the front end of a compiler. Register transfer language (RTL) satisfies both criteria[32]. Figure 3.2 shows the SML representation of the RTL syntax. The RTL semantic actions constitute an *abstract data type* that separates the RTL syntax from its semantics. Now, the syntax can be used effectively for pattern matching. This keeps the low-level implementation details of the RTL operators hidden.

The RTL actions are of two kinds — *values* and *imperatives.* Value actions produce values (*e.g.,* memory fetching, register access, addition, etc.) and imperative actions alter the state (*e.g.*, assignment and statement sequencing). A valid RTL program is a sequence

```
type RegNum = int
type Offset = int
type ImmediateValue = int

(* Instruction formats *)
datatype Instruction =

    TwoOp1  of (TwoOpInstr1 * Operand * Operand)  |
    TwoOp2  of (TwoOpInstr2 * int  * Operand)  |
    OneOp   of (OneOpInstr  * Operand)           |
    Branch  of (BranchInstr * int)

(* Instructions allowed for each format *)
and  TwoOpInstr1 = ADD | MOV  | MOVB | SUB | CMP | CMPB
and  TwoOpInstr2 = XOR | MUL  | DIV  | ASH | ASHC
and  OneOpInstr  = CLR | CLRB | INC  | DEC
and  BranchInstr = BR  | BNE  | BEQ  | BGT | BGE | BLT | BLE

(* Operand formats and addressing modes *)
and  Operand = RegDirect       of int                |
               RegIndirect     of int                |
               AutoInc         of int                |
               AutoDec         of int                |
               Immediate       of int                |
               Indexed         of (int * int)        |
               AutoIncIndirect of int                |
               AutoDecIndirect of int                |
               IndexedIndirect of (int * int)
```

Figure 3.1: **Syntactic domain that specifies PDP-11 instruction formats.**

```
datatype
  Mode = CC | Address | Int8 | Int16 | Int32 | Int64 | Float | DoubleFloat
and
  Value =

     (* Arithmetic operators *)
     Mem           of (Mode * Value)                   |
     Reg           of (Mode * int)                     |
     Add           of (Mode * Value * Value)           |
     Sub           of (Mode * Value * Value)           |
     Mul           of (Mode * Value * Value)           |
     Div           of (Mode * Value * Value)           |
     Mod           of (Mode * Value * Value)           |
     Uminus        of (Mode * Value)                   |
     Integer       of (Mode * int)                     |
      ⋮
     (* Comparison operators *)
     Compare       of (Mode * Value * Value)           |
     Eq            of (Mode * Value * Value)           |
     Neq           of (Mode * Value * Value)           |
     Gt            of (Mode * Value * Value)           |
     Gteq          of (Mode * Value * Value)           |
     Lt            of (Mode * Value * Value)           |
     Lteq          of (Mode * Value * Value)           |
     If_Then_Else  of (Value * Value * Value)          |

     (* Program counter and condition codes. *)
     PC                                        |
     N                                         |
     V                                         |
     Z                                         |
     C
and
  Imperative =
     Sequence   of (Imperative list) |
     Parallel   of (Imperative list) |
     Assign     of (Value * Value) |
     Call       of  Value |
     Nop |
     Return
```

Figure 3.2: **Syntax of semantic actions that specify RTL.**

of imperatives. The `Parallel` operator specifies that an instruction has multiple effects on the state that occur simultaneously. For example, an instruction `ADD` *Dest, Source* performs the operation $Dest = Dest + Source$ and, in parallel, assigns condition codes based on $Dest + Source$.

3.3.4 The Valuation Function

The valuation function maps syntactic objects (instructions) to semantic actions (RTL). We specify the valuation function in two parts, addressing modes and instructions. The valuation functions build *prefix operator terms* (or *abstract syntax trees*) that represent the effect of the operations. It is these terms that can either be given a semantics (as we will do in the next section), matched for code selection [1, 32], or analyzed for code optimizer generation [36].

In the following discussion, for the sake of clarity, and adhering to common notational conventions, we have strayed slightly from strict SML syntax.

Addressing Modes— A valuation function for an addressing mode returns a `Value` (figure 3.2) that describes how an operand is accessed. Some addressing modes (e.g., auto-increment/decrement modes) cause side effects on the processor state. Consequently, an addressing mode also returns an `Imperative` that specifies this side-effect. Where no side-effects take place, the imperative `Nop` is returned. The type for the addressing mode valuation function `Op` is:

$$\texttt{Op:}\quad \texttt{Operand} \rightarrow \texttt{Mode} \rightarrow (\texttt{Value} \times \texttt{Imperative})$$

Where the type `Operand` (figure 3.1) represents an operand including its addressing mode and `Mode` (figure 3.2) is the size of the data being accessed. Function `Op` is higher-order: `Op` takes `Operand` and returns a function from `Mode` to a tuple.

For example, consider the auto-increment addressing mode. The valuation function `Op` is

```
Op ⟦  AutoInc(RegNum) ⟧ Mode =
   let
    val r = Reg(Addr, RegNum)
   in
    (Mem(Mode, r),
     Assign(r,Add(Addr,r,Int(Int8,Sizeof(Mode))))
   end
```

`Op` returns a tuple, the first item of which is how the operand is accessed and the second item is the side-effect of the auto-increment addressing mode. "`AutoInc(RegNum)`" is abstract syntax and `Reg`, `Mem`, `Assign`, `Add`, and `Integer` are semantic actions. The remainder of the PDP-11's addressing modes are similarly defined and are shown in figure 3.3.

Instructions— A valuation function for an instruction constructs an `Imperative` by combining the actions of the operands with the action that represents the semantics of the instruction. The function `Instr` has type

$$\texttt{Instr} : \texttt{Instruction} \rightarrow \texttt{Imperative}$$

For example, the PDP-11 `MOV` instruction is specified by the following valuation function, `Instr`:

```
Instr ⟦  (TwoOp1(MOV, DestMode, SrcMode)) ⟧ =
  let
    val (Dest, S') = Op ⟦  DestMode ⟧ Int16
    and (Src, S'') = Op ⟦  SrcMode ⟧ Int16
  in
   Sequence([
     Parallel[
       Assign(Dest,Src),
       Assign(Z,Eq(Int16,Src,Integer(Int16,0))),
       Assign(N,Lt(Int16,Src,Integer(Int16,0)))],
     Parallel[S', S'']])
  end
```

Two subtrees, S' and S", are constructed that represent side-effects from both operand accesses. These subtrees are combined with the subtree that represents the effect or meaning of the operation.

An instruction can alter the state in three ways.

- Explicitly
- Through a side effect of the addressing mode.
- By setting condition codes.

The remainder of the PDP-11 instructions are similarly defined and are shown in figures 3.4, 3.5, and 3.6.

```
fun Op (RegDirect(RegNum)) Mode =                    (* Register Direct *)
     (Reg(Mode, RegNum), Nop)

  | Op (RegIndirect(RegNum)) Mode =                  (* Register Indirect *)
     (Mem(Mode, Reg(Address, RegNum)), Nop)

  | Op (AutoInc(RegNum)) Mode =                      (* Auto-increment *)
      let
         val r = Reg(Address, RegNum)
      in
        (Mem(Mode, r),
         Assign(r, Add(Address, r, Integer(Sizeof(Mode)))))
      end


  | Op (AutoDec(RegNum)) Mode =                      (* Auto-decrement *)
      let
         val r = Reg(Address, RegNum)
      in
        (Mem(Mode, r),
         Assign(r, Sub(Address, r, Integer(Sizeof(Mode)))))
      end

  | Op (Indexed(Offset, RegNum)) Mode =              (* Indexed *)

     (Mem(Mode, Add(Address, Reg(Address, RegNum),
                           Integer(Offset))), Nop)

  | Op (Immediate(ImmediateValue)) Mode =            (* Immediate *)

     (Integer(ImmediateValue), Nop)
  | Op (AutoIncIndirect(RegNum)) Mode =  (*  Autoincrement indirect *)

      let
         val r = Reg(Address, RegNum)
      in
        (Mem(Mode, Mem(Address, r)),
         Assign(r, Add(Address, r, Integer(Sizeof(Address)))))
      end
```

Figure 3.3: **The Semantics of the PDP-11 Addressing Modes**

```
(* Add instruction *)
fun Instr (TwoOp1(ADD, DestMode, SrcMode)) =
   let
     val (Dest, S') = Op(DestMode) Int16
     and (Src, S'') = Op(SrcMode) Int16
   in
     let
        val Result = Add(Int16, Dest, Src)
     in
       Sequence([ Parallel[ Assign(Dest,Result),
                            Assign(Z, Eq(Int16, Result, Integer(0))),
                            Assign(N, Lt(Int16, Result, Integer(0)))],
                  Parallel[S', S'']])
     end
   end


(* Compare instruction  *)
  | Instr (TwoOp1(CMP, DestMode, SrcMode)) =
   let
     val (Dest, S') = Op(DestMode) Int16
     and (Src, S'') = Op(SrcMode) Int16
   in
     let
        val Temp = Sub(Int16, Dest, Src)
     in
       Sequence( [Parallel[Assign(Z, Eq(Int16, Temp, Integer(0))),
                           Assign(N, Lt(Int16, Temp, Integer(0)))],
                  Parallel[S', S'']])
     end
   end

(* Move instruction *)
  | Instr (TwoOp1(MOV, DestMode, SrcMode)) =
   let
     val (Dest, S') = Op(DestMode) Int16
     and (Src, S'') = Op(SrcMode) Int16
   in
    Sequence([ Parallel[Assign(Dest,Src),
                        Assign(Z, Eq(Int16, Src, Integer(0))),
                        Assign(N, Lt(Int16, Src, Integer(0)))],
               Parallel[S', S'']])
   end
```

Figure 3.4: **Semantics of PDP-11 Instructions.**

```
(* Branch on not equal to zero *)
  | Instr (Branch(BNE, Offset)) =
        Assign(PC, If_Then_Else(Eq(CC, Z, Integer(0)),
                                Add(Address, PC, Integer(Offset)), PC))


(* Divide.  Dest ← Dest / Src.  Dest must be an even numbered register *)
(* The dividend is put in Dest and the remainder in Dest + 1.          *)
  | Instr (TwoOp2(DIV, RegNum, SrcMode)) =

    let
      val (Src, S') = Op(SrcMode) Int16
    in
      let
        val Quotient  = Div(Int16, Reg(Int16, RegNum), Src)
        and Remainder = Mod(Int16, Reg(Int16, RegNum), Src)
      in
        if even(RegNum)  then
         Sequence([ Parallel([ Assign(Reg(Int16, RegNum), Quotient),
                              Assign(Reg(Int16, RegNum+1), Remainder),
                              Assign(Z, Eq(Int16, Quotient, Integer(0))),
                              Assign(N, Lt(Int16, Quotient, Integer(0)))]), S'])
        else Nop
      end
    end


(* Multiply. *)
  | Instr (TwoOp2(MUL, RegNum, SrcMode)) =

    let
       val (Src, S') = Op(SrcMode) Int16
       val Result = Mul(Int32, Reg(Int16, RegNum), Src)
    in
       let
         val LoBits = Assign(Reg(Int16, RegNum), LoPart(Int16, Result))
         and HiBits =  if even(RegNum)  then
                         Assign(Reg(Int16, RegNum+1), HiPart(Int16, Result))
                       else Nop
       in
        Sequence([ Parallel([ LoBits, HiBits,
                              Assign(Z, Eq(Int16, Result, Integer(0))),
                              Assign(N, Lt(Int16, Result, Integer(0)))]), S'])
       end
    end
```

Figure 3.5: **Semantics of PDP-11 Instructions** *(continued)*.

```
(* Clear *)
  | Instr (OneOp(CLR, SrcDestMode)) =
    let
       val (SrcDest, S') = Op(SrcDestMode) Int16
       val Result = Assign(SrcDest, Integer(0))
    in
      Parallel([Result, S'])
    end

(* Increment *)
  | Instr (OneOp(INC, SrcDestMode)) =
    let
       val (SrcDest, S') = Op(SrcDestMode) Int16
       val Result = Assign(SrcDest, Add(Int16, SrcDest, Integer(1)))
    in
      Parallel([Result, S'])
    end


(* Decrement *)
  | Instr (OneOp(DEC, SrcDestMode)) =
    let
       val (SrcDest, S') = Op(SrcDestMode) Int16
       val Result = Assign(SrcDest, Add(Int16, SrcDest, Integer(-1)))
    in
      Parallel([Result, S'])
    end
```

Figure 3.6: **Semantics of PDP-11 Instructions** *(continued)*.

3.4 Action Implementation

This section outlines a denotational semantics of the RTL semantic actions used in the previous section. The implementation of the semantic actions that describe the RTL only needs to be done once, the point being that once the RTL is implemented we can easily describe a variety of architectures.

We do not have room here to present the semantics of the actions but we will briefly outline the type signatures of the actions and the kinds of values that they operate on. We first define an abstract notion of *state*.

The State— Our state consists of memory (byte addressable), registers (32-bit), and a status register (for the condition codes C, N, V, and Z) that operate on the data types given by `Mode`.

```
Mode = Addr + Int8 + Int16 + Int32 + Int64 + Float + CC
State = Memory × Registers × StatusReg
```

Each of `Memory`, `Registers`, and the `StatusReg` are stores. Abstractly, a store is a function from locations (addresses, register numbers, condition codes) to data. This suggests the following definition for the three stores:

$$\begin{aligned} \texttt{Memory} &= \texttt{Address} \rightarrow \texttt{Int8} \\ \texttt{Registers} &= \texttt{RegNum} \rightarrow \texttt{Int32} \\ \texttt{StatusReg} &= \texttt{Flags} \rightarrow \texttt{Bit} \end{aligned}$$

Action Signatures— The `Mem` action takes two parameters: a mode (`Mode`) that specifies how many bytes to fetch, and a value producing action (`Value`) that yields an address. `Mem` itself is a value producing action. Its signature is given by the following equation.

$$\texttt{Mem:}\quad \texttt{Mode} \times \texttt{Value} \rightarrow \texttt{Value}$$

An example of an imperative action (`Imperative`) is `Assign`. `Assign` takes two value actions, the first of which must be an L-value that will be assigned the data the second action produces (the R-value). The signature of `Assign` is:

$$\texttt{Assign:}\quad \texttt{Value} \times \texttt{Value} \rightarrow \texttt{Imperative}$$

A value action is consequently a function from a state to a denotation. An imperative action is a function from a state to a state.

$$\begin{aligned}
\texttt{Imperative} &= \texttt{State} \rightarrow \texttt{State} \\
\texttt{Value} &= \texttt{State} \rightarrow \texttt{Denotation} \\
\texttt{Denotation} &= \texttt{CC of Bit} + \texttt{Address of int} \\
&+ \texttt{Int8 of int} + \texttt{Int16 of int} + \texttt{Int32 of int}
\end{aligned}$$

To be complete, we must specify the semantics of individual actions. We only note that there is an SML function for each of the actions in figure 3.2. For example, there is an SML function `Assign` that has the following signature.

$$\texttt{Assign} : \texttt{Value} \times \texttt{Value} \rightarrow \texttt{Imperative}$$

Simulation — SML is, essentially, an implementation of the call-by-value typed λ-calculus. The simulator for the architecture is the SML interpreter. The simulation should be viewed as computational; that is, we model the computation of the individual instructions as functions described in the λ-calculus.

3.5 Code Generation

A retargetable code generator operates by matching portions of the intermediate language (IL) produced by the front-end to the language that describes instructions in the machine description. This of course assumes that the IL generated by the front-end is the same as the language that describes instructions. Fortunately, the RTL presented in this chapter suffices for both. (Using RTL for an IL was first proposed by Davidson [32] and is embodied in the popular retargetable C compiler GCC [89].)

Our specification can be used in a code generator because the RTL used to describe instructions can also be used as an intermediate language. For example, the instruction `Add R1, (R2)+` determines the RTL syntax tree (linearized) in figure 3.7. (Recall that `Int16` and `Address` are types that describe the kind of value in the register or memory cell.) Pattern matching code generators work by matching portions of the intermediate language generated by the front-end with patterns that represent instructions (figure 3.7). Twig [1] is one such system that could be used here.

```
Sequence([
   Parallel([
      Assign(Reg(Int16,1), Add(Int16,
                               Reg(Int16,1),
                               Mem(Int16,Reg(Address,2))))
      Assign(Z, Eq(Int16, Mem(Int16, Add(Int16,
                                         Reg(Int16,1),
                                         Mem(Int16,Reg(Address,2))))))
      Assign(N, Lt(Int16, Mem(Int16, Add(Int16,
                                         Reg(Int16,1),
                                         Mem(Int16,Reg(Address,2))))))])
   Parallel([Nop,
             Assign(Reg(Address,2),
                    Add(Address,
                        Reg(Address,2), Integer(2)))])])
```

Figure 3.7: **RTL Syntax tree corresponding to the instruction `Add R1, (R2)+`.**

3.6 Discussion

An instruction set architecture is, in a sense, a programming language and can be treated as such. Using denotational semantics we have specified, formally, the semantics of an instruction set architecture which allows for the design of modular, readable, and usable architectural specifications. We have implemented the semantics using the functional language SML which makes our specification executable, automatically providing a simulator for the instruction set architecture.

Chapter 4

Instruction Timing

In this chapter we motivate our thesis that the timing-properties of instructions that are needed for program efficiency (to perform instruction scheduling) should be included in a high-level processor specification. We have refrained from using the the term "architecture" as this level contains only information required to write *correct* programs.

4.1 Architecture and Organization

The distinction between the levels of abstraction of a processor are ill-defined and somewhat artificial. Consider the architecture and organization levels; it is often difficult keeping the details of organization from creeping into the architecture. As an extreme example, the Intel i860 processor [63] makes no such separation and one can even argue that the organization *is* the architecture. Consequently, in order to program the i860 properly it is necessary that the user understand the structure of the pipelines and how the internal latches are used.

Ideally, an architecture specification describes *final values* that the instructions compute and the organization describes *how* those values are computed. That is, the architecture is an abstraction whereas the organization is an implementation. To this end, the architecture level does not contain timing information but the organization level does. However, when considering the efficiency of a program, the organization level does contain information that is of use for the programmer. Yet the organization 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. There are many such examples of this complex *instruction interaction* which we now describe.

4.2 Delayed Instructions

Some instructions have an *architecturally defined* delay. The most common examples are delayed loads and branches. In a delayed instruction the effect of the instruction always occurs n instructions (cycles) after the instruction begins, where n represents the number of required *delay slots*. For example, some memory load instructions have an architecturally defined delay to them (it is not reasonable to expect a word from memory be fetch in a single cycle without slowing the processor clock speed) as in the following instruction sequence, where $n = 1$);

```
(1)       Load R1, (R2)                ;R1 := Mem[R2]
(2)       Add  R2, R2, R1              ;R2 := R2 + R1
```

On some processors, the use of register `R1` in `(2)` is illegal and sends the processor into an undefined state. For delayed load instructions, program correctness depends on the programmer who must ensure that the instruction executing immediately after a delayed load instruction does not reference the register being loaded.

Another situation arises with a delayed branch. For some processors, in the following instruction sequence,

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

the `Add` instruction after the branch (`BZ`) is always executed before a jump to `Locn`. If the branch is not taken a couple of different behaviors can be defined. One is to always execute the `Add` instruction whether the branch succeeds or fails. The other is to skip the `Add` instruction when the branch fails and execute the instruction immediately below the `Add` (This is sometimes called *nullification* of the delay-slot). Another constraint is that a branch instruction may not be immediately followed by another branch instruction.

4.3 Multicycle Instructions

Some instructions take several cycles to execute. In this situation, an instruction may have to wait until a previous instruction finishes. This "wait" or *data-dependent delay* occurs in the following instruction sequence,

```
(1)     Fmul FR1, FR2, FR3
(2)     Fadd FR4, FR4, FR1
```

where the `Fadd` instruction has to wait until the `Fmul` instruction finishes due to the data dependency on register `FR1`. The delay suffered in this instruction pair is not the same as that encountered in the delayed load instruction as the delay in the load instruction is "architecturally defined". That is, the delay is constant across implementations whereas the delay in the above instruction pair is dependent upon a particular implementation of the processor.

4.4 Resource Constraints

A processor has a limited set of resources. When one instruction needs a resource that is currently being used by another instruction a structural hazard results and an interlock (or delay) occurs. For example, if a processor has only one floating-point adder then, in the following sequence,

```
(1)     Fadd FR1, FR2, FR3
(2)     Fadd FR4, FR5, FR6
```

the second `Fadd` instruction will have to wait for the first `Fadd` to finish using the adder, even though the two instructions are data independent.

4.5 Multiple Instruction Issue

On some processors, when instructions have no data dependencies nor resource conflicts the processor will execute the instructions in parallel. For example, integer and floating-point instructions typically use mutually exclusive portions of the hardware and the following instructions

```
(1)    Fmul FR1, FR2, FR3
(2)    Add  R1, R2, R3
```

can execute in parallel.

4.6 Motivation

All of the timing constraints described above arise because instructions overlap in execution. This overlap is called *instruction-level parallelism* (or ILP). These constraints constitute the *programmers view of instruction timing.* The architecture level generally does not contain timing information but the organization level does. However, the organization level also contains many more details of implementation that are irrelevant to the user. Consequently, we are interested in a new level of abstraction that describes the programmer's view of instruction timing. This new level abstracts away from implementation details which we simply call the *instruction timing view.*

The main motivation for describing a processor at the timing level is that, in general, we should not expect that the user of the processor infer the existence of timing constraints by examining the organization. One of our goals then is to develop a mathematical model of instruction timing that hides irrelevant details of implementation.

4.7 Functions Don't Work

Given the above discussion on the various ways instructions can interact, the behavior of an instruction can no longer be specified in isolation of other instructions. Consequently, the typical technique of using register-transfer statements can no longer be applied. However, as we have mentioned, on some processors it is vital that the compiler know the timing constraints and the concurrent properties of the processor to use the processor efficiently and, in some cases, correctly (*e.g.,* architecturally defined delayed loads/branches and instruction latencies).

It is well-known that functional methods do not extend well to include timing constraints or concurrency. In fact, it is widely known in the programming language community that functional methods and concurrency are at odds, as has been pointed out by Milner [69], Hoare [53], and Pnueli [80]. This is because a concurrent system does not necessarily have a functional behavior as concurrency can introduce non-determinism into a system. The definition of being a *function* implies that an input has only one corresponding output.

Consider again the delayed-load instruction of the MIPS R3000 [56]. On the MIPS, for the following instruction sequence,

```
(1)       Load R1, (R2)                ;R1 := Mem[R2]
(2)       Add  R2, R2, R1              ;R2 := R2 + R1
```

the contents of register R1 is undefined at instruction (2). That is, the value of R1 is non-deterministic. The above load instruction, then, is can not be described by a function as R1 can receive one of two values:

1. The correct value stored at the memory location in R2 or

2. some undefined value.

Consequently, since R1 is undefined R2 will also be undefined, possibly causing a ripple-effect of undefined registers. This implies that if the contents of any register becomes undefined then the entire processor state must be considered to be undefined.

Even if a concurrent system does have an overall functional behavior it may be more appropriate to model the system using a concurrency-based formal method rather than a functional one. This is because the concurrent system is composed of interacting components which can yield irregular, yet deterministic behavior which can be difficult to specify functionally. For example, if the load instruction above did not include an architecturally defined delay but was interlocked instead (*i.e.,* if the processor stalled and waited for R1 to be loaded), as is done on the newer MIPS R4000, then we have to model the stall also. But modeling the stall is cumbersome using functional methods as one would have to somehow include a stall-value in the output. This is, as we will see, straightforward using a calculus of concurrency. So we will forego using a functional language in favor of a particular calculus of concurrency, SCCS.

Chapter 5

SCCS: A Synchronous Calculus of Communicating Systems

SCCS, or *Synchronous Calculus of Communicating Systems* [67, 68], is a mathematical theory of communicating systems in which we can represent systems by the *terms* or *expressions* in a simple language given by the theory. SCCS allows us to directly represent the temporal and concurrent properties of the system being specified.

5.1 A Small Example

In this section we introduce SCCS through a small example involving a pipeline. Consider a two stage pipeline where each stage adds one to its input; such a pipeline is depicted in figure 5.1.

Each stage is modeled using an *agent* (process) in SCCS, which may have one or more

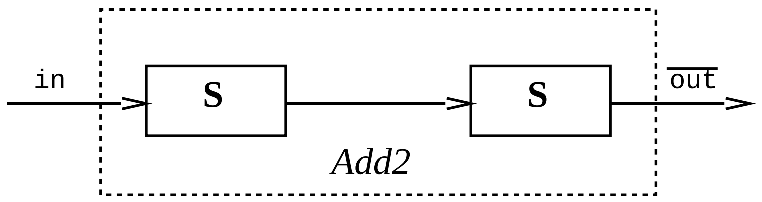


Figure 5.1: **A two stage "Add 2" pipeline constructed from two "Add 1" agents.**

communication ports. In SCCS, each agent *must* perform an action (that is, use one or more of its ports) on each clock cycle, or 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 complementary port $\overline{\alpha}$.

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

$$S(x) \overset{\text{def}}{=} \mathtt{in}(y)\overline{\mathtt{out}}(x) : S(y+1) \tag{5.1}$$

Equation 5.1 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 5.1,

- ":" is the prefix operator. The expression $\mathtt{a} : P$ represents the process that, at time t, can do the action $\mathtt{a}$ and become the process P at time $t+1$.

- $\mathtt{in}(y)\overline{\mathtt{out}}(x)$ is a *product* of actions specifying that the two particulate actions, `in` and $\overline{\mathtt{out}}$, occur simultaneously. This action can also be thought of as reading y on port `in` and sending x on port $\overline{\mathtt{out}}$.

- S is defined recursively allowing for the modeling of non-terminating agents.

- S is parameterized by 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 [68].

The semantics of SCCS is given formally in [68].

5.2 SCCS Syntax

Systems specified by SCCS consist of two entities, actions and agents (or processes). A process is an SCCS expression (Figure 5.2 gives the syntax for SCCS expressions). Actions communicate values and can either be *positive* (*e.g.* `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 the same name.

5.3 Connecting Processes

The $\times$ combinator models parallelism. The agent $A \times B$ represents agents A and B executing in parallel. If two agents joined by product contain complementary action names then these

$P(x_1, x_2, \ldots, x_n) \stackrel{\text{def}}{=} E$	Parameterized agent definition
$\mathbf{1}$, Done	Idle agent
$\mathbf{0}$	Inactive agent
$E \times F$	Parallel composition. E and F execute in parallel.
$E + F$	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
if b **then** E_1 **else** E_2	Conditional
$\sum_{i \in I} E_i$	Summation over indexing set I
$\prod_{i \in I} E_i$	Composition over indexing set I
$E \upharpoonright L$	Action Restriction
$E \backslash\backslash L$	Particle Restriction
$E[f]$	Apply relabeling function f

Figure 5.2: **Syntax of SCCS expressions**

agents are joined by what may be thought of as wires at those ports. Hence these agents may now communicate.

Given Equation 5.1 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), yet the output of the first S stage must be fed into the input of the second S stage. To model this kind of connection, SCCS provides a mechanism for *relabeling* actions. Relabeling $\overline{\texttt{out}}$ to $\overline{\alpha}$ in the first S and `in` to α in the second occurrence of S provides the desired effect.

$$\begin{aligned} \mathit{Add2}(x,y) &\stackrel{\text{def}}{=} (S(x)[\phi_1] \times S(y)[\phi_2]) \uparrow \{\texttt{in, out}\} \\ & \phi_1 = \texttt{out} \mapsto \alpha, \quad \phi_2 = \texttt{in} \mapsto \alpha \end{aligned} \tag{5.2}$$

In Equation 5.2,

- ϕ_1 is a relabeling function that means change the port name `out` to α. ϕ_2 changes `in` to α.
- $S[\phi]$ means apply relabeling function ϕ to agent S.
- $S \uparrow \{\texttt{in, out}\}$ 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 `in` and `out` are made known to the environment and α is internalized.

Combining restriction and relabeling is a way of achieving scoping in SCCS.

The net effect Equation 5.2 is to construct a pipeline of two stages where each stage adds 1 to its input.

In SCCS the agent $A + B$ represents 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 $\sum_{i=1}^{n} A_i$.

5.4 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:

- 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". This, $\alpha_1 \cdot \alpha_2$, is the product of actions α_1 and $\alpha_2 \in \mathcal{L}$.
- $\cdot$ is commutative and associative: $\alpha \cdot \beta = \beta \cdot \alpha$ and $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$.
- 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 operation: $\alpha \cdot \bar{\alpha} = 1$.

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.

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

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

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

up to order.

5.5 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.

$$A \ \mathit{Next}\ B = A \times (1 : B) \tag{5.3}$$

Next is right-associative. That is,

$$A \ \mathit{Next}\ B \ \mathit{Next}\ C = A \ \mathit{Next}\ (B \ \mathit{Next}\ C)$$

Intuitively, the agent A *Next* B *Next* C should begin executing A at time t, B at time $t+1$, and C at time $t+2$. We can see that this is the case by expanding it using Equation 5.3.

$$\begin{aligned}
A \textit{ Next } B \textit{ Next } C &= A \textit{ Next } (B \textit{ Next } C) \\
&= A \times 1 : (B \textit{ Next } C) \\
&= A \times 1 : (B \times 1 : C) \\
&= A \times 1 \cdot 1 : (B \times 1 : C) \qquad \text{(1 is identity)} \\
&= A \times 1 : B \times 1 : 1 : C \qquad \text{(by 5.9)}
\end{aligned}$$

Another useful operator is the ***priority sum*** operator, $\rhd$ [20]. Intuitively, if in the agent $A+B$ both A and B can execute, then it is non-deterministic as to which is executed. Often, we would like to prioritize $+$ so that if both A and B can execute, then A is preferred. Thus, $A \rhd B$ denotes the priority sum of A and B, where A has priority over B.

5.6 Transition Graphs

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

Definition 1 *A* **labeled transition system** *(LTS) is a triple* $\langle \mathcal{P}, \mathrm{Act}, \longrightarrow \rangle$ *where* $\mathcal{P}$ *is a set of states,* Act *is a set of actions, and* $\longrightarrow$ *is the transition relation, a subset of* $\mathcal{P} \times \mathrm{Act} \times \mathcal{P}$.

When $p, q \in \mathcal{P}$ and $\alpha \in \mathit{Act}$ and $(p, \alpha, q) \in \longrightarrow$ we write $p \xrightarrow{\alpha} q$ to mean that "state p can do an α and evolve into q." States p and q represent the state of the system at times t and $t+1$ respectively. Figure 5.4 gives the inference rules for SCCS that map an SCCS term to a labeled transition system, where states correspond to SCCS expressions. In the transition $p \xrightarrow{\alpha} q$, q is called an "α-derivative" of p or just a "derivative" of p.

A state t is *reachable* from state s if there exist states $s_0, \ldots, s_n$ and actions $\alpha_1, \ldots, \alpha_n$ such that

$$s = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} s_n = t.$$

In this case we call $\alpha_1, \ldots, \alpha_n$ a *trace* of s.

The transition graph of an agent p is a graphical representation of the LTS induced by p. For example, the SCCS agent defined in Equation 5.4 has the transition graph shown in figure 5.3.

$$E \overset{\text{def}}{=} a : b : \mathbf{0} + c : (d : \mathbf{0} + e : \mathbf{0}) \tag{5.4}$$

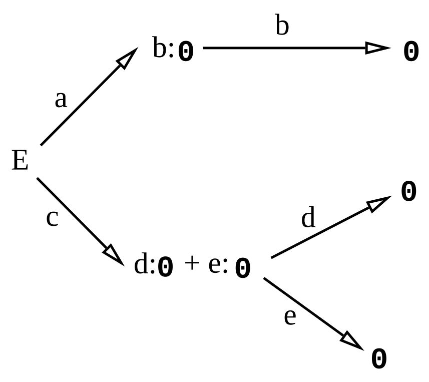


Figure 5.3: **Transition graph of agent defined in Equation 5.4.**

The transition graph of an agent represents its dynamic nature as it represents an agent "executing" through time. The LTS itself is a structure that represents all possible traces of the system. For example, the LTS in figure 5.3 of the SCCS process in Equation 5.4 identifies the set $\{(a,b),(c,d),(c,e)\}$ as the possible traces of E of length two. Analysis of SCCS processes (*e.g.,* simulation, equivalence testing, model checking) is carried out on the LTS rather than SCCS terms directly.

5.7 The Operational Semantics of SCCS

In general, an operational semantics of a language maps terms of the language to some *abstract machine.* A *structural operational semantics* (developed by Plotkin [79]) uses syntax directed rules to map terms of the language to a transition system (or labeled transition system in our case). The structural operational semantics (or SOS [79]) for SCCS is given in figure 5.4.

Each rule in figure 5.4 is an inference rule structured on the syntax of an SCCS term. The following conveys the intuition behind the rules:

Action — In the process $\mathsf{a} : 0$ the rule **Action** says that $\mathsf{a} : 0 \xrightarrow{\mathsf{a}} 0$.

Product — If P can do an action α *and* Q can do an action β then $P \times Q$ can do an action $\alpha \cdot \beta$. For example, consider the process $\mathsf{a} : \mathbf{0} \times \mathsf{b} : \mathsf{b} : \mathbf{0}$. The rule **Product** says that

1. since $\mathsf{a} : \mathbf{0} \xrightarrow{\mathsf{a}} \mathbf{0}$ and

$$\textbf{Sum1}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \textbf{Sum2}\ \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\textbf{Action}\ \frac{}{\alpha : P \xrightarrow{\alpha} P}$$

$$\textbf{Product}\ \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q'}{P \times Q \xrightarrow{\alpha \cdot \beta} P' \times Q'} \qquad \textbf{Restriction}\ \frac{P \xrightarrow{\alpha} P'}{P \uparrow A \xrightarrow{\alpha} P' \uparrow A}\ \alpha \in A$$

$$\textbf{Relabeling}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \textbf{Definition}\ \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ A \overset{\text{def}}{=} P$$

Figure 5.4: **Operational semantics of SCCS.**

2. since $\mathsf{b} : \mathsf{b} : \mathbf{0} \xrightarrow{\mathsf{b}} \mathsf{b} : \mathbf{0}$ then
3. then $\mathsf{a} : \mathbf{0} \times \mathsf{b} : \mathsf{b} : \mathbf{0} \xrightarrow{\mathsf{a} \cdot \mathsf{b}} \mathbf{0} \times \mathsf{b} : \mathbf{0}$.

Sum — The **Sum1** rule says that if P can do an action α and become P' then $P + Q$ can do an α and become P'. The rule **Sum2** handles the other case.

Relabeling — If P can do an α then the process $P[f]$ can do an $f(\alpha)$.

Restriction — If P can do an α then $P \uparrow A$ can do an α provided $\alpha \in A$. The predicate $\alpha \in A$ is called a *side condition.*

Definition — The definition rule allows us to bind a process to an identifier. It says that if P can do an α then so can any process identifier A where $A \overset{\text{def}}{=} P$.

The inference rules define all possible transitions for any SCCS term.

SCCS is an *algebra* that satisfies many equational laws (figure 5.5). The equational laws are based on the definition of *bisimulation equivalence* in SCCS which is defined in [67]. We will not pursue this here.

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

$$P \times \mathbf{1} \sim P \tag{5.5}$$

$$P + \mathbf{0} \sim P \tag{5.6}$$

$$P + P \sim P \tag{5.7}$$

$$P \times \mathbf{0} \sim \mathbf{0} \tag{5.8}$$

$$a : P \times b : Q \sim ab : (P \times Q) \tag{5.9}$$

$$P \times (Q + R) \sim P \times Q + P \times R \tag{5.10}$$

$$P + Q \sim Q + P \tag{5.11}$$

$$P \times Q \sim Q \times P \tag{5.12}$$

$$(P \times Q) \times R \sim P \times (Q \times R) \tag{5.13}$$

$$(P + Q) + R \sim P + (Q + R) \tag{5.14}$$

$$a(b : P + c : Q) \sim ab : P + ac : Q \tag{5.15}$$

$$(a : P) \upharpoonright A \sim \begin{cases} a : (P \upharpoonright A) & \text{if } a \in A \\ \mathbf{0} & \text{if } a \notin A \end{cases} \tag{5.16}$$

$$P \upharpoonright A \upharpoonright B \sim P \upharpoonright (A \cap B) \tag{5.17}$$

Figure 5.5: **Equational laws of SCCS**

5.8 Examples

Here we present some examples using SCCS to specify various simple circuits. More examples and their implementation in the Concurrency Workbench are given in Appendix B. The purpose of this section is to familiarize the reader with SCCS and how it is used. Since we will eventually be specifying the timing properties of a RISC-style microprocessor we use digital hardware in our examples.

5.8.1 A Zero-Delay and a Unit-Delay Wire

The simplest circuit is a wire. One end is the input and the other is the output which we label with the actions `in` and $\overline{\texttt{out}}$.

In all of the examples, variables range over booleans 0 and 1. Equation 5.18 represents a zero-delay wire. At all times t the current input at `in` is equal to the output at $\overline{\texttt{out}}$.

$$\mathit{Wire0} \stackrel{\mathrm{def}}{=} \texttt{in}(i)\overline{\texttt{out}}(i) : \mathit{Wire0} \tag{5.18}$$

The process *Wire1* below represent a wire that has a unit-delay which means it has a current state j. At the time t the input i on `in` becomes, at time $t+1$, the output j on port $\overline{\texttt{out}}$. (Compare the HOL equation for this same circuit, Equation 2.1.)

$$\mathit{Wire1}(j) \stackrel{\mathrm{def}}{=} \texttt{in}(i)\overline{\texttt{out}}(j) : \mathit{Wire1}(i)$$

We can also implement a wire that has a delay of two clock cycles:

$$\mathit{Wire2}(i,j) \stackrel{\mathrm{def}}{=} \texttt{in}(a)\overline{\texttt{out}}(j) : \mathit{Wire2}(a,i)$$

We could have also specified the two-delay wire by hooking together two unit-delay wires:

$$\mathit{Wire2Impl}(i,j) \stackrel{\mathrm{def}}{=} (\,\mathit{Wire1}(i)[\overline{\texttt{alpha}}/\overline{\texttt{out}}] \times \mathit{Wire1}(j)[\texttt{alpha}/\texttt{in}]) \uparrow \{\texttt{in}, \texttt{out}\}$$

One important aspect of SCCS is that now we can *prove* that the agents specified by *Wire2* and *Wire2Impl* are equivalent. In fact, the Concurrency Workbench can establish this for us automatically.

5.8.2 Logic Gates

A unit-delay inverter is specified in Equation 5.19.

$$\begin{aligned} \mathit{Not}(0) &\stackrel{\mathrm{def}}{=} \texttt{in}(0)\overline{\texttt{out}}(0) : \mathit{Not}(1) + \texttt{in}(1)\overline{\texttt{out}}(0) : \mathit{Not}(0) \\ \mathit{Not}(1) &\stackrel{\mathrm{def}}{=} \texttt{in}(0)\overline{\texttt{out}}(1) : \mathit{Not}(1) + \texttt{in}(1)\overline{\texttt{out}}(1) : \mathit{Not}(0) \end{aligned} \tag{5.19}$$

Using a little notational freedom for specifying the functionality of an agent we can abbreviate Equation 5.19 with Equation 5.20.

$$Not(j) \overset{\text{def}}{=} \texttt{in}(i)\overline{\texttt{out}}(j) : Not(\neg i) \tag{5.20}$$

Connecting two inverters together we get Equation 5.21.

$$TwoNots(i, j) \overset{\text{def}}{=} (Not(i)[\overline{\texttt{alpha}}/\overline{\texttt{out}}] \times Not(j)[\texttt{alpha}/\texttt{in}]) \uparrow \{\texttt{in}, \texttt{out}\} \tag{5.21}$$

Connecting two inverters together cancel each other out, except for the delay. We should expect that agent *TwoNots* be equivalent to agent *Wire2* (and consequently *Wire2Impl*). Indeed, this is the case.

Other logic gates are similarly defined (of course all we really need is a nand or a nor and we can define all of them in terms of this operation and prove that they meet their specification).

$$And(j) \overset{\text{def}}{=} \texttt{in1}(a)\texttt{in2}(b)\overline{\texttt{out}}(j) : And(a \wedge b) \tag{5.22}$$

$$Or(j) \overset{\text{def}}{=} \texttt{in1}(a)\texttt{in2}(b)\overline{\texttt{out}}(j) : Or(a \vee b) \tag{5.23}$$

$$Xor(j) \overset{\text{def}}{=} \texttt{in1}(a)\texttt{in2}(b)\overline{\texttt{out}}(j) : Xor(a \oplus b) \tag{5.24}$$

$$Nor(j) \overset{\text{def}}{=} \texttt{in1}(a)\texttt{in2}(b)\overline{\texttt{out}}(j) : Nor(a \textbf{ nor } b) \tag{5.25}$$

These processes are all unit-delay with j being the "default" initial state.

5.8.3 Flip-Flop

A flip-flop is easily constructed by connecting together two *Nor* gates (figures 5.6 and 5.7). Each Nor gate has an output that feeds back into the other Nor gate. In order to make the desired connections we have make two copies of a Nor gate and suitably relabel some ports. Finally the restriction operator internalizes the two feedback connections.

$$FF(m, n) \overset{\text{def}}{=}$$
$$(Nor(m)[\texttt{s}/\texttt{in1},\ \ \texttt{g} \cdot \texttt{in1}/\texttt{out}] \times Nor(n)[\texttt{r}/\texttt{in2},\ \ \texttt{in2} \cdot \texttt{d}/\texttt{out}]) \uparrow \{\texttt{s}, \texttt{r}, \texttt{g}, \texttt{d}\}$$

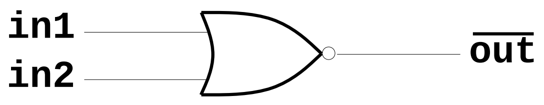


Figure 5.6: **A nor gate.**

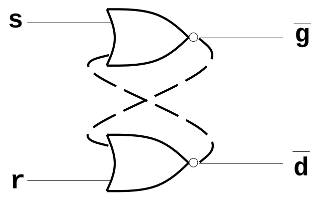


Figure 5.7: **A flip-flop constructed from two nor-gates (Figure 5.6).**

Chapter 6

Specifying a Processor

In this chapter we present a specification of the programmer's timing view for hypothetical RISC-style processor, based on DLX and the MIPS from Patterson and Hennessy [76, 77, 56].

6.1 Our example microprocessor

In this RISC, instructions, memory word size, registers, and addresses are thirty-two bits. Figure 6.1 gives the instruction syntax and computational semantics of our processor. What follows is an informal description of some of our RISC instructions that we subsequently describe with SCCS.

- `Add` $\mathtt{R}_i$`,` $\mathtt{R}_j$`,` $\mathtt{R}_k$ adds registers j and k and puts the result in register i. The instruction executing immediately after an `Add` may use register i.

- `Load` $\mathtt{R}_i$`,` $\mathtt{R}_j$`, #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` $\mathtt{R}_i$`, #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` $\mathtt{FR}_i$`,` $\mathtt{FR}_j$`,` $\mathtt{FR}_k$ 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,

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 6.1: **RISC instruction set.**

then instruction execution stalls until the result is ready.

The table in Figure 6.1 gives the entire instruction set along with the RTL statement describing the functional semantics of each instruction. This functional semantics can be easily represented using the techniques from chapter 3.

6.2 Timing Constraints

There are three types of constraints that alter the programmer's view of the timing of a processor.

delayed instructions — The effect of an instruction can be delayed (*e.g.,* as in our RISC'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.

We will discuss all three types of timing constraints as we encounter them.

6.3 The SCCS Specification

A processor is a system in which registers and memory interact with one or more functional units. Equation 6.1 represents such a system at the highest level in SCCS.

$$\mathit{Processor} \overset{\text{def}}{=} (\mathit{Instruction\ Unit} \times \mathit{Memory} \times \mathit{Registers}) \uparrow I \qquad (6.1)$$

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

Equation 6.1 says that the only visible actions are instructions. That is, the labels on the labeled transition system generated by 6.1 are entities like `Add R1, R2, R2`, `BZ R1,` *Locn*, etc.

6.4 Instruction Formats as Actions

We will represent the syntax (format) of each instruction as an SCCS action. The instructions described in Figure 6.1 determine a set of actions, one for each instruction. For example, the instruction `Add R`$_1$`, R`$_2$`, R`$_3$ is represented in SCCS by the action $\mathtt{AddR_1R_2R_3}$. More generally, the instruction `Add R`$_i$`, R`$_j$`, R`$_j$ determines a set of actions, I_{Add}, given by Equation 6.2.

$$I_{\text{Add}} \overset{\text{def}}{=} \mathtt{Add\ R_i,\ R_j,\ R_j} = \bigcup_{i,j,k=0}^{31} \mathtt{AddR_iR_jR_k} \tag{6.2}$$

The set, I_{op} can be determined for each opcode, $op \in \mathit{Opcodes}$, yielding the set I of all instructions (Equation 6.3).

$$I \overset{\text{def}}{=} \bigcup_{op \in \mathit{Opcodes}} I_{\text{op}} \tag{6.3}$$

For readability, we continue to comma separate an instruction's operands, always keeping in mind that an instruction is represented by an action. (We could also use the bit representation of the instructions but this would only make the specification less readable.)

6.5 Defining the Registers

Before we proceed in specifying instructions and their interaction, it is necessary to develop an appropriate model of registers and memory. In this section we develop an abstract model of storage in which storage cells are modeled as agents. Equation 6.4 defines one cell, $\mathit{Cell}(y)$, holding a value y, such that an action $\mathtt{putc}(x)$ executed at time t stores x in *Cell* which is available for use at time $t+1$. The action $\overline{\mathtt{getc}}(y)$ retrieves the value stored in *Cell* and assigns this to y. If no agent wants to interact with *Cell* using `putc` or $\overline{\mathtt{getc}}$ actions then *Cell* executes the idle action 1.

$$\mathit{Cell}(y) \quad \overset{\text{def}}{=} \quad \overline{\mathtt{getc}}(y) : \mathit{Cell}(y) \quad + \quad \mathtt{putc}(x) : \mathit{Cell}(x) \quad + \quad 1 : \mathit{Cell}(y) \tag{6.4}$$

This model of a storage cell is simple but inadequate because *Cell* can only perform one `getc` or `putc` at a time. Consider the instruction `Add R`$_1$`, R`$_1$`, R`$_1$ which accesses $\mathtt{R_1}$ twice and also writes $\mathtt{R_1}$. On most processors this instruction can effectively execute in a single cycle because registers are read and written in different pipeline stages and there is appropriate forwarding hardware, renaming registers, etc.. 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* three times with the cell's value placed in a, b and, and c and also write d to *Cell.* Only one `putc` is allowed for each action.

Equation 6.5 defines an agent *Reg* that is a new version of *Cell* that can accommodate parallel reads and writes.

$$\mathit{Reg1}(y) \overset{\text{def}}{=} \sum_{j=0}^{2} \overline{\mathtt{getr}}(y)^j(1 : \mathit{Reg}(y) \ + \ \mathtt{putr}(x) : \mathit{Reg}(x)) \tag{6.5}$$

Notice that we can change the upper bound on the summation to allow an arbitrary number of readers of the registers (*i.e.,* `getr`'s)

If we expand the summation in Equation 6.5 we obtain Equation 6.6.

$$\begin{aligned} \mathit{Reg1}(y) \quad \overset{\text{def}}{=} \quad & \overline{\mathtt{getr}}(y)^0 : \mathit{Reg}(y) \\ + \quad & \overline{\mathtt{getr}}(y)^1 : \mathit{Reg}(y) \\ + \quad & \overline{\mathtt{getr}}(y)^2 : \mathit{Reg}(y) \\ + \quad & \overline{\mathtt{getr}}(y)^0\mathtt{putr}(x) : \mathit{Reg}(x) \\ + \quad & \overline{\mathtt{getr}}(y)^1\mathtt{putr}(x) : \mathit{Reg}(x) \\ + \quad & \overline{\mathtt{getr}}(y)^2\mathtt{putr}(x) : \mathit{Reg}(x) \end{aligned} \tag{6.6}$$

Applying the equational law that states that for any particulate action $a \in \mathcal{A}$, $a^0 = 1$ and $a^1 = a$ Equation 6.6 reduces to Equation 6.7.

$$\begin{aligned} \mathit{Reg1}(y) \quad \overset{\text{def}}{=} \quad & \overline{\mathtt{getr}}(y) : \mathit{Reg}(y) \\ + \quad & \overline{\mathtt{getr}}(y)^2 : \mathit{Reg}(y) \\ + \quad & \mathtt{putr}(x) : \mathit{Reg}(x) \\ + \quad & \overline{\mathtt{getr}}(y)\mathtt{putr}(x) : \mathit{Reg}(x) \\ + \quad & \overline{\mathtt{getr}}(y)^2\mathtt{putr}(x) : \mathit{Reg}(x) \\ + \quad & 1 : \mathit{Reg}(y) \end{aligned} \tag{6.7}$$

In further sections, we will not continue to expand summations like this; it was done here to indicate 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.

6.5.1 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 6.5 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 6.8 modifies *Reg1* so that when an agent locks a register the register goes into a state *Locked_Reg* where the only allowable action is $\overline{\mathtt{putr}}(x)\mathtt{releasereg}$. 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 this out and put them in Equation 6.10. Notice again, that the upper-bound of the summation in equations 6.8 and 6.10 can be arbitrarily increased to include any number of readers.

$$\mathit{Reg}(y) \overset{\text{def}}{=} \mathit{Reg1}(y) + \sum_{j=0}^{2} \overline{\mathtt{getr}}(y)^j \mathtt{lockreg} : \mathit{Locked\_Reg}(y) \tag{6.8}$$

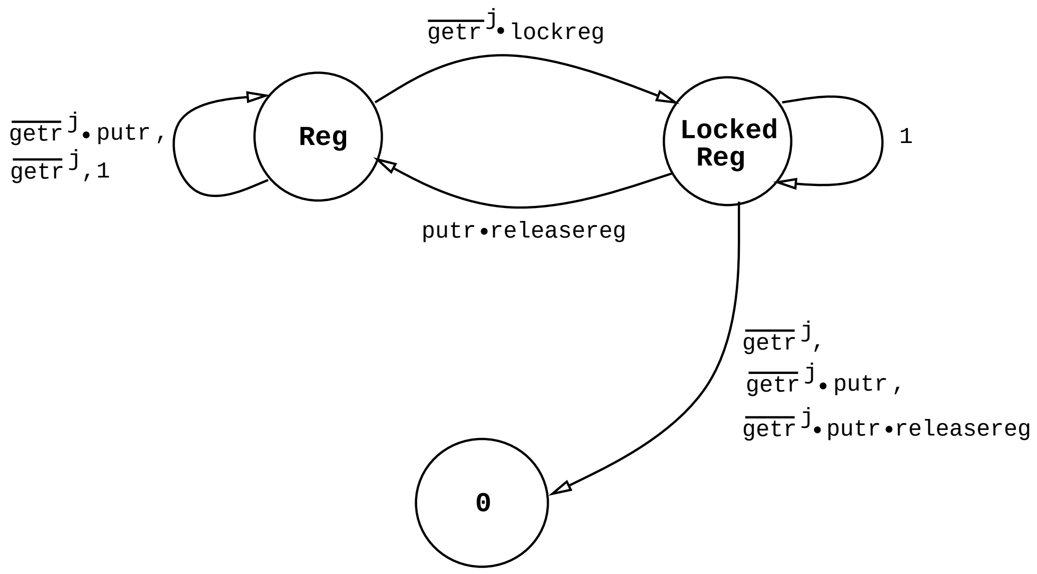
$$\begin{aligned}&\mathit{Locked\_Reg}(y) \overset{\text{def}}{=} \\ &\quad \mathit{Illegal\_Access}(y) + \mathtt{putr}(x)\mathtt{releasereg} : \mathit{Reg}(x) + 1 : \mathit{Locked\_Reg}(y)\end{aligned} \tag{6.9}$$

$$\begin{aligned}&\mathit{Illegal\_Access}(y) \overset{\text{def}}{=} \\ &\quad \sum_{j=0}^{2} \overline{\mathtt{getr}}(y)^j \left(\overline{\mathtt{getr}}(y) : \mathbf{0} + \mathtt{putr}(x) : \mathbf{0} + \mathtt{putr}(x)\mathtt{releasereg} : \mathbf{0}\right)\end{aligned} \tag{6.10}$$

Figure 6.2 shows the state transition graph of *Reg* (Equation 6.8) which is the labeled transition system generated by the operational semantics of SCCS. The figure shows that, from the state *Reg*, any number of **getr**'s (including zero) and zero or one **putr**'s will leave us in the state *Reg*. However, any number of **getr**'s and a **lockreg** action will put us in the *Locked_Reg* state. Figure 6.2 also shows that any action other than a $\mathtt{putr} \cdot \mathtt{releasereg}$ will put us in the deadlocked state **0**.

Given the definition of one register, a family of registers (Reg_1, Reg_2, *etc.*) is now defined by subscripting each of the actions by a register number. For example, the action $\mathtt{putr}_i(x)$ represents writing x to register i. Thirty-two registers are constructed by

$$\mathit{Registers} \overset{\text{def}}{=} \mathit{Reg}_0(y) \times \cdots \times \mathit{Reg}_{31}(y) \tag{6.11}$$



$\overline{\text{getr}}^j \bullet \text{lockreg}$
$\overline{\text{getr}}^j \bullet \text{putr}$,
$\overline{\text{getr}}^j, 1$
Reg
Locked Reg
1
putr•releasereg
$\overline{\text{getr}}^j$,
$\overline{\text{getr}}^j \bullet \text{putr}$,
$\overline{\text{getr}}^j \bullet \text{putr} \bullet \text{releasereg}$
0

Figure 6.2: **State transition graph of agent *Reg* in Equation 6.8.**

which we abbreviate to

$$Registers \overset{\text{def}}{=} \prod_{i=0}^{31} Reg_i(y) \tag{6.12}$$

(we have not changed SCCS at all, this is just a shorthand notation).

6.6 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.

$$\begin{aligned} MEM_i(y) &\overset{\text{def}}{=} \mathtt{putm}_i(x) : MEM_i(x) \\ &+ \overline{\mathtt{getm}_i}(y) : MEM_i(y) \\ &+ \overline{\mathtt{getm}}(y)\mathtt{putm}(x) : MEM(x) \\ &+ 1 : MEM_i(y) \qquad (6.13) \\ MEMORY &\overset{\text{def}}{=} \prod_{i=0}^{2^{32}-1} MEM_i(y) \qquad (6.14) \end{aligned}$$

6.7 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 write back 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 a more abstract specification will suffice.

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 reasons we chose SCCS.

6.8 Instruction Issue

Given our previous definitions of *Registers* and *Memory* and using a program counter, PC, we now describe an agent $Instr(PC)$ (Equation 6.15) that specifies the behavior of our processor's 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*).

$$\begin{aligned}
Instr(PC) &\stackrel{\text{def}}{=} (Non\_Branch(PC)\ Next\ \ Instr(PC+4)) \\
&\quad + \ Branch(PC) \\
&\quad \triangleright \ Stall(PC) && (6.15) \\
Non\_Branch(PC) &\stackrel{\text{def}}{=} Alu(PC) + Load\_Store(PC) + Float(PC) && (6.16) \\
Stall(PC) &\stackrel{\text{def}}{=} 1 : Instr(PC) && (6.17)
\end{aligned}$$

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 6.15 describes this situation.

- A branch instruction may execute, in which case the next instruction 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 6.22).

- If no instruction can execute then the processor must stall (Equation 6.17). The $\triangleright$ operator (section 5.5) is used here because the processor should stall only when no other alternative is available.

6.8.1 Arithmetic Instructions

Like most processors, ours 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. (Recall the slight abuse of notation with the comma separated operands.)

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. Normally, a user would *expect* this instruction sequence to be legal and the user should not need to understand the details of the instruction pipeline that might make the sequence *illegal* nor the bypass hardware that makes the sequence behave as originally expected.

The agent

$$\mathit{Alu}(PC) \stackrel{\mathrm{def}}{=} \mathtt{getm}_{\mathrm{PC}}(\mathtt{Add\ R}_i\mathtt{,R}_j\mathtt{,R}_k)\mathtt{getr}_j(x)\mathtt{getr}_k(y)\overline{\mathtt{putr}_i}(x+y) : \mathit{Done} \qquad (6.18)$$

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 6.18 describes the same computation as the register transfer statement

$$\mathtt{Reg[i]} \leftarrow \mathtt{Reg[j]} + \mathtt{Reg[k]}$$

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). The other arithmetic instructions are analogous.

6.8.2 Integer Load and Store Instructions

The following instruction sequence,

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

is illegal in our processor 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 $t+2$.

The computational behavior of the load instruction is the following RTL statement.

$$\texttt{Load R}_i\texttt{, R}_j\texttt{, Offset} \equiv \texttt{Reg[i]} \leftarrow \texttt{Mem[Reg[j] + Offset]}$$

This is represented by,

$$\begin{aligned} &\mathit{Load}(PC) \stackrel{\text{def}}{=} \\ &\quad \texttt{getm}_{\mathrm{PC}}(\texttt{Load R}_i\texttt{, R}_j\texttt{, }\Delta)\texttt{getr}_j(B)\texttt{getm}_{\mathrm{B}+\Delta}(V)\overline{\texttt{lockreg}_i} : \\ &\quad \overline{\texttt{putr}_i}(V)\overline{\texttt{releasereg}_i} : \mathit{Done} \end{aligned} \tag{6.19}$$

Equation 6.19 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 $\texttt{getm}_{\mathrm{B}+\Delta}(V)$ 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 $\overline{\texttt{releasereg}_i}$).

In the event that an instruction attempts to read or write a locked register the processor reaches the deadlocked state $\mathbf{0}$. This is because our process that represents a register traps any such operations.

The Store Instruction

The store operation effectively takes one cycle to execute. For example, the following code segment is legal.

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

The computational behavior of the store instruction is given by the following RTL statement.

$$\texttt{Store R}_i\texttt{, R}_j\texttt{, Offset} \equiv \texttt{Mem[Reg[i] + Offset]} \leftarrow \texttt{Reg[j]}$$

It is unlikely, however, that `R1` has finished being written to memory. The hardware takes care of the difficulties allowing the next instruction to access the memory location just written. Equation 6.20 represents the store instruction.

$$\begin{aligned} &\mathit{Store}(PC) \overset{\text{def}}{=} \\ &\quad \texttt{getm}_{\text{PC}}(\texttt{Store R}_i\texttt{, R}_j\texttt{, }\Delta)\texttt{getr}_j(B)\texttt{getr}_i(V)\texttt{putm}_{\text{B}+\Delta}(V) : \mathit{Done} \end{aligned} \tag{6.20}$$

The important characteristic of Equation 6.20 is that it specifies that the `Store` instruction effectively takes one cycle to execute. Equation 6.21 combines the two agents *Load* and *Store* for Equation 6.15.

$$\mathit{Load\_Store}(PC) \overset{\text{def}}{=} \mathit{Load}(PC) + \mathit{Store}(PC) \tag{6.21}$$

6.8.3 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 6.22 specifies the behavior of the `BZ` instruction.

$$\begin{aligned}
\mathit{Branch}(PC) \quad \overset{\text{def}}{=} \quad & \mathtt{getm}_{\mathrm{PC}}(\mathtt{BZ}\ \mathtt{R}_i\mathtt{,}\ \mathit{Locn})\mathtt{getr}_i(V): \\
& \quad \textbf{if } V = 0 \textbf{ then} \\
& \quad\quad \mathit{Non\_Branch}(PC + 4)\ \mathit{Next}\ \ \mathit{Instr}(\mathit{Locn})) \\
& \quad\quad + \ \mathtt{getm}_{\mathrm{PC+4}}(\mathtt{BZ}\ \mathtt{R}_i\mathtt{,}\ \mathit{Locn}) : \mathbf{0} \\
& \quad \textbf{else} \\
& \quad\quad \mathit{Instr}(PC + 8)
\end{aligned} \tag{6.22}$$

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.

6.9 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. As inserting `Nop` instructions would significantly increase code size, therefore, floating-point instructions are typically interlocked.

6.9.1 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 while accessing a locked FP-register causes the processor to stall.

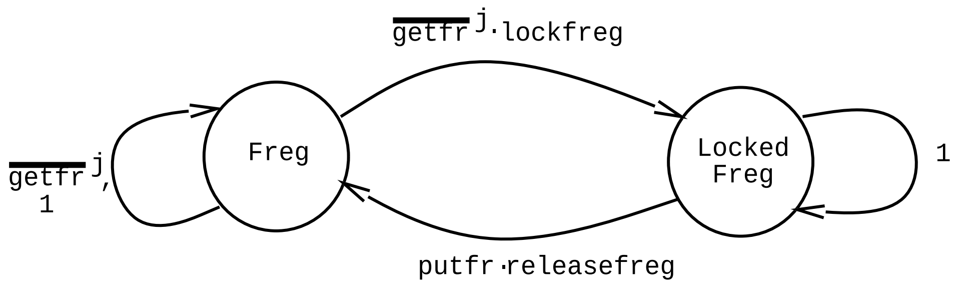

$\overline{\texttt{getfr}}^{j}\cdot\texttt{lockfreg}$
Freg
Locked Freg
1
$\overline{\texttt{getfr}}^{j}$, 1
putfr·releasefreg

Figure 6.3: **State transition graph of a floating-point register,** ***Freg.***

Our processor 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.

$$
\begin{aligned}
\mathit{Freg}_i(y) &\overset{\text{def}}{=} \sum_{j\in\{0,1,2\}} \overline{\mathtt{getfr}_i}(y)^j\mathtt{lockfreg}_i : \mathit{Locked\_Freg}_i \\
&+ \sum_{j\in\{1,2\}} \overline{\mathtt{getfr}_i}(y)^j : \mathit{Freg}_i(y) \\
&+ 1 : \mathit{Freg}_i(y) \\
\mathit{Locked\_Freg}_i &\overset{\text{def}}{=} \mathtt{putfr}_i(x)\mathtt{releasefreg}_i : \mathit{Freg}_i(x) \\
&+ 1 : \mathit{Locked\_Freg}_i
\end{aligned}
$$

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

$$
\mathit{FP\_Registers} \overset{\text{def}}{=} \prod_{j=0}^{31} \mathit{Freg}_j(y) \qquad (6.23)
$$

Figure 6.3 shows the state transition graph for a floating-point register. For the case of the integer registers, it was illegal to access a locked register, and trying to do so was trapped by directly putting the processor in the deadlocked state. But, for the floating-point registers, there is no deadlocked state. A process (instruction) is still not able to read a locked floating-point register, but trying to do so causes the processor to stall as the only possible transition will be the *Stall* process (Equation 6.17).

6.9.2 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 6.24 specifies our processor's **Fadd** instruction.

$$\begin{aligned} \mathit{Float}(PC) \quad \overset{\text{def}}{=} \quad & \mathtt{getm}_{\mathrm{PC}}(\mathtt{Fadd},\ \mathtt{FR}_i,\ \mathtt{FR}_j,\ \mathtt{FR}_k)\overline{\mathtt{lockfreg}_i}\mathtt{getfr}_j(x)\mathtt{getfr}_k(y): \\ & (1:)^5 \\ & \overline{\mathtt{putfr}_i}(x+y)\overline{\mathtt{releasefreg}_i}: \mathit{Done} \end{aligned} \tag{6.24}$$

The abbreviation $(1:)^n$ represents the n-cycle delay, $\overbrace{1:1:\ldots:1}^{n\text{ times}}$, 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 6.15). (Remember in the definition of *Instr*(PC) in Equation 6.15 that our processor continues executing instructions after the **Fadd** instruction has started.)

6.10 Structural Constraints

Processors often reuse functional units. For example, a floating-point unit may have only one adder that is used by the addition, multiplication, 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 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 omit for the sake of simplicity). 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.

6.10.1 Modeling Finite Resources

We need a way to include resources in our model. We could leave them out — doing so would still give us a good "approximation" of the timing behavior of our processor (that is we could still model delayed load, branches, and latencies) but if we wish to be more accurate then we should include resources. Also, instruction schedulers consider resource constraints.

To model limited resources, we introduce a generic agent *Resource* that models a resource that instructions can acquire and release. When an instruction acquires a resource that is being used by another, the processor stalls. Equation 6.25 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.

$$\begin{aligned} \mathit{Resource} &\stackrel{\text{def}}{=} \mathtt{get\_resource} : \mathit{Locked\_Resource} \\ &+ \mathtt{get\_resource} \cdot \mathtt{release\_resource} : \mathit{Resource} \\ &+ 1 : \mathit{Resource} \qquad (6.25) \\ \mathit{Locked\_Resource} &\stackrel{\text{def}}{=} \mathtt{release\_resource} : \mathit{Resource} \\ &+ 1 : \mathit{Locked\_Resource} \qquad (6.26) \end{aligned}$$

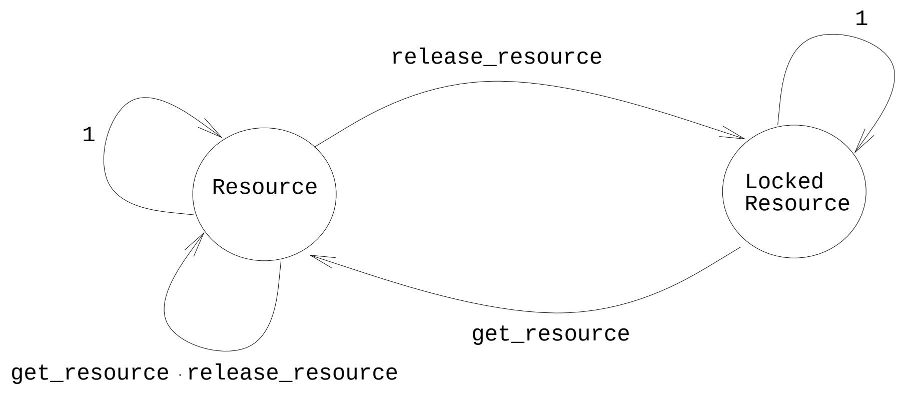


Figure 6.4: **State transition graph of a *Resource* in Equation 6.25.**

Figure 6.4 shows the transition graph for the agent *Resource*.

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

In our processor, the floating-point unit has three resources that must be shared: an adder, multiplier, and a divider. Equation 6.27 specifies this by replicating *Resource* three times and relabeling its actions appropriately.

$$\begin{aligned} FPU \;&\overset{\text{def}}{=}\; Resource[\phi] \times Resource[\psi] \times Resource[\theta] \qquad (6.27)\\ \text{where}\;\; &\phi = \texttt{get\_resource} \mapsto \texttt{get\_multiplier},\\ &\quad\;\; \texttt{release\_resource} \mapsto \texttt{release\_multiplier}\\ &\psi = \texttt{get\_resource} \mapsto \texttt{get\_adder},\\ &\quad\;\; \texttt{release\_resource} \mapsto \texttt{release\_adder}\\ &\theta = \texttt{get\_resource} \mapsto \texttt{get\_divider},\\ &\quad\;\; \texttt{release\_resource} \mapsto \texttt{release\_divider} \end{aligned}$$

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 **get_adder·release_adder**. If an instruction needs the adder for two consecutive cycles then the instruction should specify this with the agent **get_adder:release_adder**. The instruction should not acquire and release the adder on each cycle as in the following.

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

By requiring that resource requirements be specified this way we are imposing a normal form on the specification. However, this normal form is reasonable and we justify it by noting that:

- processor manuals indicate resource requirements in this fashion (for example see this MIPS processor manual [56]);
- the instruction scheduling problem, as presented in the literature defines resource requirements in this manner.

6.10.2 Multi-cycle Floating-point Instructions

Now that there are several floating-point instructions competing for shared resources the `Fadd` instruction defined in Equation 6.24 needs to be altered. We redefine the agent *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.

$$\begin{aligned}\mathit{FADD}(PC) \stackrel{\text{def}}{=}\ & \mathtt{getm}_{\mathrm{PC}}(\mathtt{Fadd},\ \mathtt{FR}_i,\ \mathtt{FR}_j,\ \mathtt{FR}_k)\overline{\mathtt{lockfreg}_i}\mathtt{getfr}_j(x)\mathtt{getfr}_k(y) : \\ & \overline{\mathtt{get\_adder}} : (1 :)^3\overline{\mathtt{release\_adder}} : \\ & \overline{\mathtt{putfr}_i}(x+y)\overline{\mathtt{releasefreg}_i} : \mathit{Done} \end{aligned} \qquad (6.28)$$

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.

$$\begin{aligned}\mathit{FMUL}(PC) \stackrel{\text{def}}{=}\ & \mathtt{getm}_{\mathrm{PC}}(\mathtt{Fmul},\ \mathtt{FR}_i,\ \mathtt{FR}_j,\ \mathtt{FR}_k)\overline{\mathtt{lockfreg}_i}\mathtt{getfr}_j(x)\mathtt{getfr}_k(y) : \\ & \overline{\mathtt{get\_adder}} \cdot \overline{\mathtt{release\_adder}} : \\ & \overline{\mathtt{get\_multiplier}} : \overline{\mathtt{release\_multiplier}} : \\ & \overline{\mathtt{get\_adder}} \cdot \overline{\mathtt{release\_adder}} : \\ & \overline{\mathtt{putfr}_i}(x*y)\overline{\mathtt{releasefreg}_i} : \mathit{Done} \end{aligned} \qquad (6.29)$$

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

$$\begin{aligned} \mathit{FDIV}(PC) \quad \overset{\text{def}}{=} \quad & \mathtt{getm}_{\mathrm{PC}}(\mathtt{Fdiv},\ \mathtt{FR}_i,\ \mathtt{FR}_j,\ \mathtt{FR}_k)\overline{\mathtt{lockfreg}_i}\mathtt{getfr}_j(x)\mathtt{getfr}_k(y) : \\ & \overline{\mathtt{get\_adder}} \cdot \overline{\mathtt{release\_adder}} : \\ & \overline{\mathtt{get\_divider}} : (1 :)^6\overline{\mathtt{release\_divider}} : \\ & \overline{\mathtt{get\_adder}} : \overline{\mathtt{release\_adder}} : \\ & \overline{\mathtt{putfr}_i}(x/y)\overline{\mathtt{releasefreg}_i} : \mathit{Done} \end{aligned} \tag{6.30}$$

Essentially, we are modeling a resource as a binary semaphore. This technique can be used to handle any kind of resource that needs to be accessed exclusively (*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

$$\begin{gathered} \mathit{Resource}[\phi] \times \mathit{Resource}[\phi] \\ \text{where } \phi = \mathtt{get\_resource} \mapsto \mathtt{get\_adder} \end{gathered} \tag{6.31}$$

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).

Non-determinism is an important aspect of specification. The potential non-determinism introduced above, while not implementable at the hardware level, helps keep a description from being *over specified.* For example, if a processor has two identical adders, at the specification level, we may not want to dictate which of the two adders the instruction uses.

6.11 A Normal Form

This chapter has demonstrated a method of specifying the timing properties of instructions. Our approach requires that instructions be specified in a certain manner so we present the following normal form for the specification. As we will see the restrictions are reasonable.

6.11.1 Register Locking

An instruction that locks a register must eventually unlock that same register. That is, a process that describes an instruction that lockes a register i must execute the following sequence:

$$\cdots \longrightarrow \sigma \xrightarrow{\texttt{lockreg}_i} \cdots \xrightarrow{\texttt{releasereg}_i} \sigma' \longrightarrow \cdots$$

6.11.2 Resource Requirements

An instruction that needs a resource will eventually release that same resource. That is, a process that describes an instruction that acquires a resource r must execute the following sequence:

$$\cdots \longrightarrow \sigma \xrightarrow{\texttt{get}_r} \cdots \xrightarrow{\texttt{release}_r} \sigma' \longrightarrow \cdots$$

Moreover, if an instruction needs resource r for one cycle then it has the sequence:

$$\cdots \longrightarrow \sigma \xrightarrow{\texttt{get}_r \cdot \texttt{release}_r} \sigma' \cdots$$

6.11.3 Undefined Instruction Sequences

Two situations arose in our processor definition where instruction sequences were undefined; 1) a branch instruction followed by another branch or 2) an instruction referencing a register that was currently being loaded by a load instruction. In both of these situations we trapped the offending sequence by having a transition to SCCS's undefined state, **0**. If an instruction sequence $i_1 \cdots i_n$ is illegal then executing that sequence should cause a transition to **0**. That is the following transition must occur:

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

The general way of specifying an illegal sequence is to have the instructions communicate through an intermediate process. This was the technique we used when specifying illegal *Load - Store* combinations where the intermediate process was a lockable register.

6.12 Summary

In this chapter we have presented a technique for specifying the "programmer's timing view" of RISC-style architectures. We handled architecturally defined delayed loads and branches, interlocked floating-point instructions, and resource constraints. The technique for specifying resources is very general and can be used to specify a variety of constraints. For example, if we had a dual-ported register file and if, for some reason, two ports were not enough and could lead to a structural hazard, we could model a port as a resource. As another example, the Motorola 88000 has a structural hazard on a write-back data bus in which it is possible that up to three instructions can try to use the write-back bus on the same cycle (as explained in [2]). The hazard is resolved by giving priority to integer instructions over floating-point instructions. In this situation we could model the write-back bus as a resource and use our priority-sum operator, $\rhd$ to correctly model how the hazard is resolved.

Chapter 7

Multiple Instruction-Issue Processors

7.1 An *Integer* $\times$ *Float* Superscalar

This section describes a superscalar (multiple instruction-issue) version of our processor 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 7.1.

$$\begin{aligned} & (\mathit{Float}(PC) \times \mathit{Alu}(PC+4)) \\ + \quad & (\mathit{Alu}(PC) \times \mathit{Float}(PC+4)) \end{aligned} \tag{7.1}$$

We can rewrite this sum as Equation 7.2.

$$\sum_{i,j \in \{0,4\}} (\mathit{Alu}(PC+i) \times \mathit{Float}(PC+j)) \tag{7.2}$$

Assuming an instruction is not both an integer and floating-point, Equation 7.2 represents a folding of Equation 7.1.

Equation 7.2 is a sum of four terms,

$$\begin{aligned} & (\mathit{Alu}(PC+0) \times \mathit{Float}(PC+0)) \\ + \quad & (\mathit{Alu}(PC+0) \times \mathit{Float}(PC+4)) \end{aligned}$$

$$\begin{aligned} &+ \quad (\mathit{Alu}(PC+4) \times \mathit{Float}(PC+0)) \\ &+ \quad (\mathit{Alu}(PC+4) \times \mathit{Float}(PC+4)) \end{aligned} \tag{7.3}$$

However, when $i = j$ then $\mathit{Alu}(PC+i)$ and $\mathit{Float}(PC+j)$ refer to the same memory location and it is impossible for a memory location to contain both an *Alu* instruction and a *Float* instruction. We can conclude then that

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

and only two terms remain (Equation 7.1). We use the summation notation because it enables us to succinctly specify n-way instruction parallelism.

Equation 7.4 extends Equation 7.2 to continue execution at $PC + 8$.

$$\begin{aligned} &\mathit{Do\_Two}(PC) \overset{\text{def}}{=} \\ &\quad \left(\sum_{i,j \in \{0,4\}} (\mathit{Alu}(PC+i) \times \mathit{Float}(PC+j)) \right) \ \mathit{Next} \ \ \mathit{Instr}(PC+8) \end{aligned} \tag{7.4}$$

There are no data dependencies to worry about because each instruction accesses separate register files. That is, because processes *Alu* and *Float* use disjoint register files, and the only way data dependencies arise is when two or more instructions use the same register, then it follows that an integer instruction and a floating-point instruction can not have a data dependency between them.

7.1.1 Instruction Issue

Our top-level instruction issue equation (Equation 6.15) must now be modified to take this new two-issue capability into account. For reference, we restate *Instr* (Equation 6.15), and rename it *Do_One*.

$$\begin{aligned} \mathit{Do\_One}(PC) \ &\overset{\text{def}}{=} \ (\mathit{Non\_Branch}(PC) \ \mathit{Next} \ \ \mathit{Instr}(PC+4)) \\ &+ \ \ \mathit{Branch}(PC) \end{aligned} \tag{7.5}$$

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

$$\mathit{Instr}(PC) \ \overset{\text{def}}{=} \ \mathit{Do\_Two}(PC) \rhd \mathit{Do\_One}(PC) \rhd \mathit{Stall}(PC) \tag{7.6}$$

Notice here the use of the priority choice operator, $\rhd$ (section 5.5) 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. Similarly, if we can not execute any instruction then the processor must stall.

7.2 An *Integer* × *Integer* Superscalar

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

$$Alu(PC) \times Alu(PC + 4) \tag{7.7}$$

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.

7.2.1 Data Dependencies (or Data Hazards)

The instructions in Figure 7.1 represent all of the possible dependencies that can exist between any two instructions[1].

The instructions in 7.1a can be executed in parallel because they are data independent while those in 7.1b cannot be executed in parallel because of the *read-after-write* (or RAW) hazard on `R1`. In 7.1c parallel execution is possible if the processor can do register renaming to eliminate the *write-after-read* (or 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 7.1d present a rare (but possible) *write-after-write* (or 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.

7.2.2 Specifying Data Hazards

Using restriction, we can force Equation 7.7 to apply only to legal integer instruction sequence of length two. If the first integer instruction writes register i then the second integer instruction cannot write or read register i. For example, given two integer instructions, if the first instruction writes register 0 then Equation 7.8 represents the legal integer instruction

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

```
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 7.1: **Possible data dependencies in instruction sequences.**

sequences.

$$\begin{aligned} \mathit{Alu}(PC)\backslash\backslash A_0 \quad &\times \quad \mathit{Alu}(PC+4)\backslash\backslash B_0 \\ \textbf{where} \quad & A_0 = \{\mathtt{putr}_0, \mathtt{getr}_0, \ldots, \mathtt{getr}_{31}\} \\ & B_0 = \{\mathtt{putr}_1, \ldots, \mathtt{putr}_{31}, \mathtt{getr}_1, \ldots, \mathtt{getr}_{31}\} \end{aligned} \tag{7.8}$$

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 [67]. In Equation 7.8 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.

$$\begin{aligned} \mathit{Do\_Two}(PC) \quad &\stackrel{\text{def}}{=} \quad \sum_{i=0}^{31} (\mathit{Alu}(PC)\backslash\backslash A_i \times \mathit{Alu}(PC+4)\backslash\backslash B_i)\ \mathit{Next}\ (PC+8) \\ \textbf{where} \quad & A_i = \{\mathtt{putr}_i, \mathtt{getr}_0 \ldots \mathtt{getr}_{31}\} \\ & B_i = \{\mathtt{getr}_0 \ldots \mathtt{getr}_{31}, \mathtt{putr}_0 \ldots \mathtt{putr}_{31}\} - \{\mathtt{putr}_i, \mathtt{getr}_i\} \end{aligned} \tag{7.9}$$

Equation 7.9 represents all of the allowable integer instruction sequences of length two that may execute in parallel.

7.3 An *Integer* × *Integer* × *Float* Superscalar

In this section we describe a version of our processor that can execute three instructions in parallel, two of which can be integer instructions and the other of which may be a floating-point instruction. Now that we have already specified two dual-issue versions (sections 7.2 and 7.1) 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 the time being).

$$\begin{aligned} & (\mathit{Alu}(PC) \times \mathit{Alu}(PC+4) \times \mathit{Float}(PC+8)) \\ + \; & (\mathit{Alu}(PC) \times \mathit{Float}(PC+4) \times \mathit{Alu}(PC+8)) \\ + \; & (\mathit{Float}(PC) \times \mathit{Alu}(PC+4) \times \mathit{Alu}(PC+8)) \end{aligned} \tag{7.10}$$

Adding data hazard constraints to Equation 7.10 as in Equation 7.9 gives us Equation 7.11.

$$\begin{aligned} & \mathit{Do\_Three}(PC) \stackrel{\text{def}}{=} \\ & \sum_{i=0}^{31} \left(\begin{array}{lllll} & \mathit{Alu}(PC)\backslash\backslash A_i & \times & \mathit{Alu}(PC+4)\backslash\backslash B_i & \times & \mathit{Float}(PC+8) \\ + & \mathit{Alu}(PC)\backslash\backslash A_i & \times & \mathit{Float}(PC+4) & \times & \mathit{Alu}(PC+8)\backslash\backslash B_i \\ + & \mathit{Float}(PC) & \times & \mathit{Alu}(PC+4)\backslash\backslash A_i & \times & \mathit{Alu}(PC+8)\backslash\backslash B_i \end{array} \right) \mathit{Next} \;\; \mathit{Instr}(PC+12) \\ & \quad \textbf{where } A_i = \{\mathbf{putr}_i, \mathbf{getr}_0 \ldots \mathbf{getr}_{31}\} \\ & \qquad\quad B_i = \{\mathbf{getr}_0 \ldots \mathbf{getr}_{31}, \mathbf{putr}_0 \ldots \mathbf{putr}_{31}\} - \{\mathbf{putr}_i, \mathbf{getr}_i\} \end{aligned} \tag{7.11}$$

The top-level issue equation now allows executing three, two, one, or zero instructions each cycle.

$$\mathit{Instr}(PC) \stackrel{\text{def}}{=} \mathit{Do\_Three}(PC) \rhd \mathit{Do\_Two}(PC) \rhd \mathit{Do\_One}(PC) \rhd \mathit{Stall}(PC) \tag{7.12}$$

As an example, the instruction sequence in Figure 7.2a can be executed in parallel according to Equation 7.11 while the sequence in Figure 7.2b cannot because of the hazard. The situation is much the same for specifying four-issue, five-issue, etc.

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

Figure 7.2: **Possible three-issue instruction sequences.**

Chapter 8

Simulation

In this section we show how our SCCS specification of our example processor is simulated. The simulation occurs within the framework of the Concurrency Workbench [25] which allows us to experiment with, simulate, and analyze SCCS specifications.

8.1 The Reactive View

A reactive system is one that interacts with its environment. This is opposed to the functional view in which a system is viewed extensionally as a mapping of initial inputs to final outputs. A common way to think of a reactive system is to view it as a black box with buttons, where the buttons represent the actions that the user (or environment) can perform.

The reactive view of our SCCS processor description is one where instructions represent the buttons. Initially, at time t_0, any button can be pressed (*i.e.,* any instruction can be executed). At time t_1 some buttons can be pressed and some cannot. The instructions that can't be executed (the button representing the instruction can't be pressed) must be "waiting" for something from the instruction that was initiated at time t_0.

For example, if the **Fadd FR$_0$, FR$_1$, FR$_2$** button is pressed at time t_0, then at time t_1, any button that uses **FR$_0$** cannot be pressed.

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

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

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

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

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

Figure 8.1: **Derivation of program executing on the processor.**

8.2 Simulation

A simulation of our processor specification amounts to loading a program into memory (with `putm` actions) and then running the agent 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 5.6 that the transition graph of an agent P consists of transitions of the form $A \xrightarrow{\alpha} B$.

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

Because our processor represents such a large system, it is not feasible to write down the entire graph so we will abbreviate by only showing pertinent states.

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 particles are enclosed with "[ ]" (*e.g.*, $[\mathbf{getr}_1(x)][\overline{\mathbf{putr}_2}(x)]$). Also, the particles that appear between the $\langle \cdots \rangle$ do not really appear on the transition because each particle has synchronized with either a register or a resource. We label the transitions with these particles anyway just to show all of the internal details. To simplify the graph many actions and processes have been omitted. At the nodes, unchanged register and memory cells have been replaced by an ellipsis.

8.3 A Simple Example

Figure 8.1 shows the transition graph of the following program.

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

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

In the graph in Figure 8.1:

- 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.

8.4 Example: An Illegal Instruction Sequence

Executing an illegal instruction sequence on our specification should lead to the inactive agent **0**. Figure 8.2 traces the following illegal instruction sequence.

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

On the second transition the particle $\mathbf{getr}_1(x)$ causes the agent $Locked\_Reg_1$ to change to the agent **0** (see Equation 6.10).

8.5 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 8.3). For this example, we

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

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

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

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

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

Figure 8.2: **Derivation of illegal program executing.**

```
      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 8.3: **Program that calculates a vector sum.**

t: $\boxed{\mathit{Instr}(PC) \times \cdots}$

$[\texttt{LoadI R}_0\texttt{, \#0}]\langle\!\langle[\texttt{lockreg}_0]\rangle\!\rangle$

$t+1$: $\boxed{\mathit{Instr}(PC+4) \times \overline{\texttt{putr}}_0(0)\overline{\texttt{releasereg}}_0 : \mathit{Done} \times \mathit{Locked\_Reg}_0(y) \times \cdots}$

$[\texttt{LoadI R}_1\texttt{, \#0}]\langle\!\langle[\texttt{lockreg}_1][\overline{\texttt{putr}}_0(0)][\overline{\texttt{releasereg}}_0]\rangle\!\rangle$

$t+2$: $\boxed{\mathit{Instr}(PC+8) \times \overline{\texttt{putr}}_1(0)\overline{\texttt{releasereg}}_1 : \mathit{Done} \times \mathit{Locked\_Reg}_1 \times \cdots}$

$[\texttt{LoadI R}_2\texttt{, \#Vec}]\langle\!\langle[\texttt{lockreg}_2][\overline{\texttt{putr}}_1(0)][\overline{\texttt{releasereg}}_1]\rangle\!\rangle$

$t+3$: $\boxed{\mathit{Instr}(PC+12) \times \mathit{Locked\_Reg}_2(y) \times \overline{\texttt{putr}}_2(\#\texttt{Vec})\overline{\texttt{releasereg}}_2 : \mathit{Done} \times \cdots}$

$[\texttt{MovItoFP FR}_0\texttt{, R}_0]\langle\!\langle[\texttt{getr}_0(x)][\overline{\texttt{putfr}}_0(x)][\overline{\texttt{putr}}_2(\#\texttt{Vec})][\overline{\texttt{releasereg}}_2]\rangle\!\rangle$

$t+4$: $\boxed{\mathit{Instr}(PC+16) \times \cdots}$

$[\texttt{Load.f FR}_1\texttt{, R}_2\texttt{ \#0}]\langle\!\langle[\texttt{lockfreg}_1][\texttt{getr}_2(B)][\texttt{getm}_B(V)]\rangle\!\rangle$

$t+5$: $\boxed{\mathit{Instr}(PC+24) \times \overline{\texttt{putfr}}_1(V)\overline{\texttt{releasefreg}}_1 \times \mathit{Locked\_FP\_Reg}_1 \times \cdots}$

$[\texttt{AddI R}_2\texttt{, R}_2\texttt{, \#4}]\langle\!\langle[\overline{\texttt{putfr}}_1(V)][\overline{\texttt{releasefreg}}_1][\texttt{getr}_2(x)][\overline{\texttt{putr}}_2(x+4)]\rangle\!\rangle$

$t+6$: $\boxed{\mathit{Alu}(PC+28) \times \mathit{Float}(PC+32) \cdots}$

$[\texttt{AddI R}_1\texttt{, R}_1\texttt{, \#1}]\langle\!\langle[\texttt{getr}_1(x)][\overline{\texttt{putr}}_1(x+1)]\rangle\!\rangle$

$[\texttt{Fadd FR}_0\texttt{, FR}_0\texttt{, FR}_1]\langle\!\langle[\texttt{gefr}_0(x)]\ [\texttt{gefr}_1(y)][\overline{\texttt{lockfreg}}_0]\rangle\!\rangle$

$t+7$: $\boxed{\mathit{Instr}(PC+36) \times (1:1:1:1:1:\overline{\texttt{putfr}}_0\overline{\texttt{releasefreg}}_0 : \mathit{Done}) \times \mathit{Locked\_FP\_Reg}_0 \times \cdots}$

$\vdots \qquad \vdots \qquad \vdots$

Figure 8.4: **Transition graph of vector sum from Figure 8.3.**

use the superscalar version of our processor defined in Section 7.1. Note that there are instructions in Figure 8.3 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 (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 8.4 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.

In conclusion, since the operational semantics of SCCS maps SCCS terms to abstract machines (*i.e.,* labeled transition systems) we have a direct way of examining the behavior of an SCCS specification.

Chapter 9

Instruction Scheduling

This chapter gives a brief overview and formalization of instruction scheduling. As an example, we present an algorithm for the most common form of instruction scheduling, *List Scheduling.* We need a formal definition of the instruction scheduling problem so that we can deduce what information an instruction scheduler requires. Knowing what to look for then leads us into extracting this information from our specification.

Instruction scheduling is primarily carried out on RISC/Superscalar style architectures that have the following characteristics:

- Load/Store architecture. Only two instructions, load and store, are allowed to access memory. All other instructions operate on registers.
- Instructions are fixed format. That is, every instruction is encoded in the same number of bits. A typical RISC has 32-bit instruction formats.
- A new instruction can, potentially, be issued every cycle.

9.1 Instruction Scheduling

Given a sequence of instructions, S, *instruction scheduling* is the problem of reordering S into S' such that S' has two properties:

1. the semantics of the original program S is unaltered, and
2. the time to execute S' is minimal with respect to all permutations of S that preserve the semantics of S.

The instruction scheduling problem is NP-complete in general [59, 38].

Before we proceed we need a definition.

Definition 2 The **length** of an instruction i is the minimum number of cycles needed to execute i.

In our case, *length*(i), is the number of cycles needed to execute i in the absence of all hazards. Intuitively, one way to compute *length*(i) is to execute i in isolation.

9.1.1 Constraints

We introduce a machine model that has two types of scheduling constraints—precedence and resource.

Precedence Constraints

A precedence constraint (or data dependency) is a requirement that a particular instruction i execute before another instruction j due to a data dependency between i and j. As mentioned in Section 7.2.1, data dependencies are of three types — true or forward, anti-, and output — which correspond to RAW, WAR, and WAW hazards.

The objects to be scheduled are individual instructions of a program. If I is the set of instructions in a program, then the precedence constraints induce a *partial order*, *Prec*, on P such that, $\mathit{Prec} \subseteq I \times I$. If there is a precedence constraint between two instructions, i and j, such that i must execute before j then $(i, j) \in \mathit{Prec}$.

The partial order, *Prec*, is usually represented graphically by a *directed acyclic graph* (DAG) G. G is composed of a set of vertices V and a set of edges E, $G = (V, E)$. Each vertex of the graph is an instruction, and if $(i, j) \in \mathit{Prec}$ then there is a directed edge, (i, j), from vertex i to vertex j. As an example, Figure 9.1 shows the dependency graph for the loop body (a basic block) of the vector sum program in Figure 8.3.

If instruction i must execute before j we do not usually require that i run to completion before j can begin. Hence, we augment the DAG by labeling each edge, e, with a *minimal latency* (delay), $d(e)$. This latency represents the least amount of time, in cycles, that must pass after i begins executing before j can begin.

Definition 3 The **latency** between two instruction i and j is the *least* number of cycles that must pass between initiating i and then initiating j.

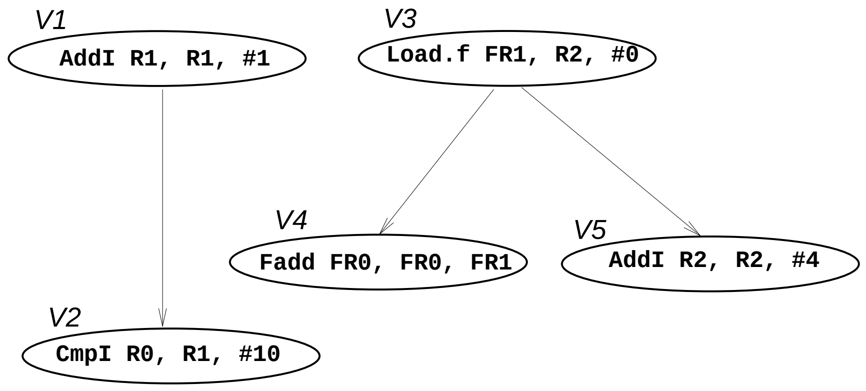


Figure 9.1: **Dependency graph of vector sum program from Figure 8.3.**

Resource Constraints

An architecture consists of a *multiset*[1], R, of resources. On each clock cycle that it is executing, each instruction uses a subset of resources from R, intuitively, the resources being used by the instruction at that time. If an instruction i executes for $\mathit{length}(i)$ cycles then the *resource usage function* for instruction i, ρ_i, maps clock cycles to subsets of R. That is, $\rho_i : t \rightarrow \mathit{Pow}(R)\ \mathit{s.t.}\ t \in \omega$ where ω represents the set of natural numbers. When $t > \mathit{length}(i)$ then $\rho_i(t) = \emptyset$. For example, $\rho_{\texttt{Add}}(2)$ represents the multiset of resources used on clock cycle 2 by the `Add` instruction.

The scheduling constraint on resources then is that, at any particular time t, the resources needed by the instructions executing at time t do not exceed the available resources.

As an example, recall the description of the MIPS/R4000 floating-point unit from Section 6.10. The resource set R is

{`divider`, `shifter`, `adder`, `rounder`, `mult_S1`, `mult_S2`, `unpacker`, `exception` }

and the partial resource usage function for the `FDIV` instruction is given in Figure 9.2.

9.1.2 Instruction Scheduling Algorithms

We now define what an *instruction schedule* is and formalize the *instruction scheduling problem.*

[1]A multiset is a set that allows duplicate elements. For example, there might be two or more floating-point adders. Multisets are also known as "bags".

$$\begin{aligned}
\rho_{\mathrm{FDIV}}(2) &= \{\texttt{adder}, \texttt{shifter}\} \\
\rho_{\mathrm{FDIV}}(3) &= \{\texttt{rounder}, \texttt{shifter}\} \\
\rho_{\mathrm{FDIV}}(4) &= \{\texttt{shifter}\} \\
\rho_{\mathrm{FDIV}}(5) &= \{\texttt{divider}\} \\
&\vdots
\end{aligned}$$

Figure 9.2: **Part of resource usage function for MIPS/R4000 `FDIV` instruction.**

Definition 4 An *instruction schedule*, σ, for a program DAG is an injective function that assigns each node $v \in V$ an integer that represents its position in the list of instructions ($\sigma : V \to \omega$). That is, σ is a topological ordering to G.

For example, consider the DAG from Figure 9.1. One possible instruction schedule would be:

$$\sigma(V_3) = 1, \ \ \sigma(V_1) = 2, \ \ \sigma(V_2) = 3, \ \ \sigma(V_5) = 4, \ \ \sigma(V_4) = 5$$

which we could abbreviate as $V_3; V_1; V_2; V_5; V_4$. Another possible schedule is $V_1; V_2; V_3; V_4; V_5$.

Definition 5 The *instruction scheduling problem* is to find the shortest schedule, σ, such that both precedence (Equation 9.1) and resource (Equation 9.2) constraints are met[2].

$$\forall e = (u, v) \in E, (\sigma(v) - \sigma(u)) \geq d(e) \tag{9.1}$$

$$\forall t \in \omega, \bigcup_{v \in V} \rho_v(t - \sigma(v)) \subseteq R \tag{9.2}$$

Intuitively, if Equation 9.1 is true then distance between any two dependent instructions is greater than or equal to the required minimum latency. If Equation 9.2 is true then the schedule σ has not overcommitted the available resources. A schedule is not overcommitted if, at any time t, the union of all the resource requirements of all the instructions still executing at time t (those that started at $t - 1, t - 2, \ldots$) is a subset of R, the available resources of the machine.

[2]In Equation 9.2, $\cup$ and $\subseteq$ refer to multiset union and subset.

The above characterization of instruction scheduling is the basis for all instruction scheduling methods, including list scheduling, trace scheduling, and software pipelining [73, 14, 15, 57, 58, 82, 7, 55, 35].

At this point we have defined the instruction scheduling problem and we know what information we need to do instruction scheduling. To make the definition more concrete we discuss the most common instruction scheduling algorithm, List Scheduling.

List Scheduling

To see what information is needed to parameterize instruction scheduling, we now describe List Scheduling, which we have taken from Lam's thesis [58].

The main idea of list scheduling is straightforward. There is a list of "ready to execute" instructions called the *ready list* which is, initially, the root nodes of V, that is, the nodes that have no incoming edges. Instructions are selected from the ready list, using some type of priority scheme, until the ready list is empty. After each instruction is scheduled, new nodes are added to the ready list. Altering the priority heuristic produces many different types of scheduling algorithms. Figure 9.3 gives the list scheduling algorithm.

We need not go into the details of function *List_Schedule* but only note that it is a function of three things:

- the program DAG G
- the table of latencies, d, used in *Satisfy_Precedence_Constraints*
- the resource usage function ρ used in *Satisfy_Resource_Requirements*

Scheduling algorithms work on a section of straight-line code, called a basic block. Trace scheduling and software pipelining are two techniques that schedule instructions *beyond* basic blocks. Without going into the details, we mention that both trace scheduling and software pipelining use list scheduling as a basis.

Figure 9.4 depicts a canonical view of the phases of compilation and shows where this research fits in (highlighted in the dotted box). In the figure, each bold-faced node not enclosed in a box is "data" for some portion of the compilation system. In our case, the timing specification is input to a program that generates parameters for a generic instruction scheduler.

```
function List_Schedule($G$ as $(V, E), d, \rho$)
    let
        $Ready$ = root nodes of $V$ and
        $Scheduled = \emptyset$
    in
        while $Ready \neq \emptyset$ do
            $v$ := highest priority node in $Ready$;
            $Lb$ := Satisfy_Precedence_Constraints($v, Scheduled, \sigma, d$);
            $\sigma(v)$ := Satisfy_Resource_Constraints($v, Scheduled, \sigma, \rho, Lb$);
            $Scheduled := Scheduled \cup \{v\}$;
            $Ready := Ready - \{v\} \cup \{u \mid u \notin Scheduled \wedge \forall (w, u) \in E, w \in Scheduled\}$;
        end while
        return($\sigma$);
    end let
end List_Schedule

function Satisfy_Precedence_Constraints($v, Scheduled, \sigma, d$)
    return($\max_{u \in \mathrm{Scheduled}}(\sigma(u) + d(u, v))$)

function Satisfy_Resource_Constraints($v, Scheduled, \sigma, \rho, Lb$)
    for $t = Lb$ to $\infty$ do
        if $\forall 0 \leq j \leq length(v).((\bigcup_{u \in \mathrm{Scheduled}} \rho_u(i + j - \sigma(u))) \cup \rho_v(j)) \subseteq R$ then
            return($t$)
```

Figure 9.3: **Generic list scheduling algorithm.**

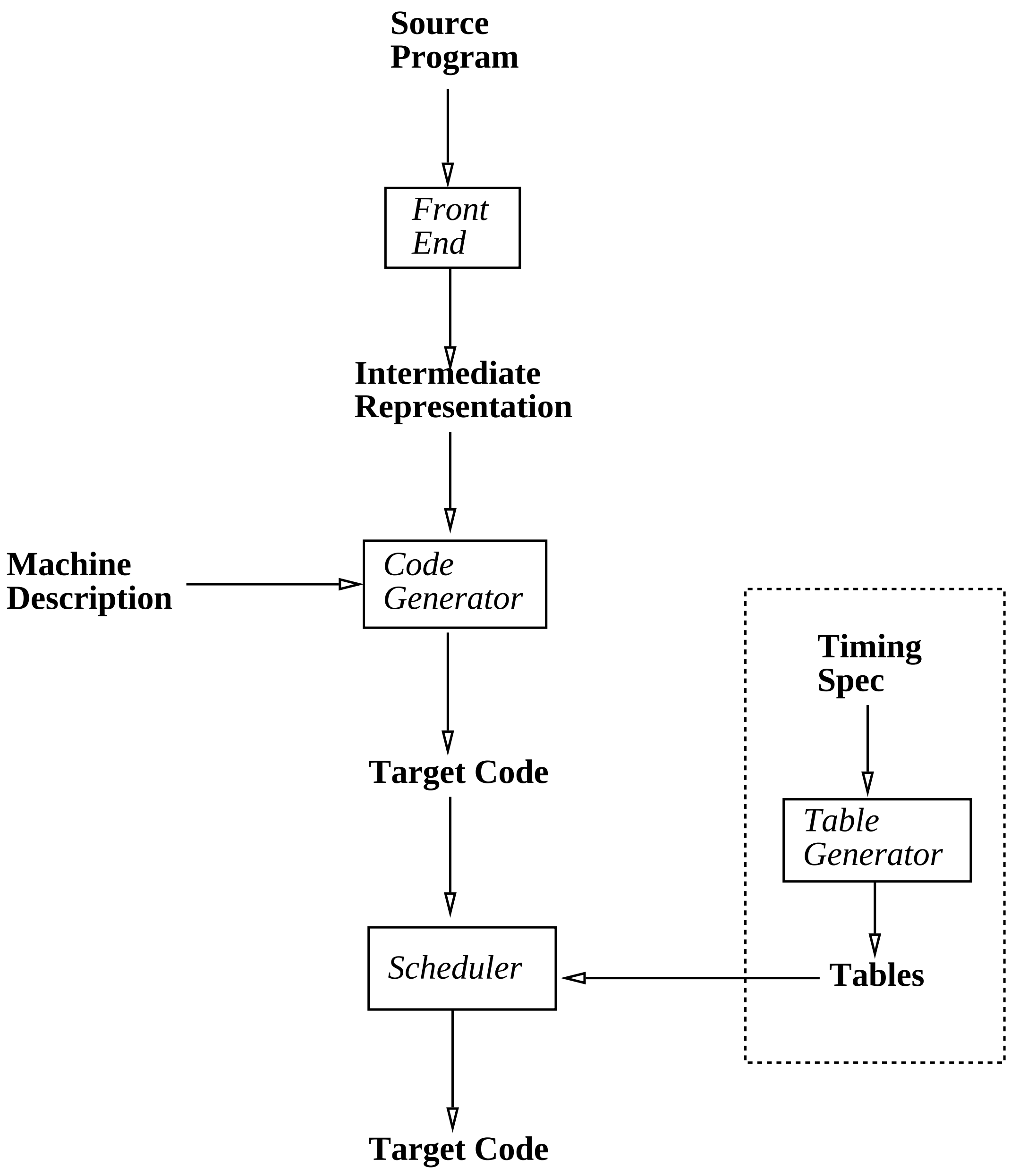


Figure 9.4: **The structure of a typical compiler.**

Chapter 10

Deriving Instruction Scheduling Parameters

This chapter describes how instruction scheduling information is derived from a timing specification, written in SCCS. The general idea is that the parameters d and ρ from the previous chapter are derivable, automatically, from the specification yielding an instruction scheduler for the processor.

10.1 Preliminaries

Recall, from chapter 9, that the length of an instruction i is the minimum number of cycles required to execute i. Intuitively, one way to compute $\mathsf{length}(i)$ is to execute i in isolation. In SCCS this amounts running the SCCS agent that describes i and counting cycles. Unfortunately, in general, SCCS agents do not have to terminate. For example, our SCCS processor description may not terminate. However, termination is a reasonable assumption to make about our SCCS agents that describe individual machine instructions. Moreover, so we can "observe" this termination, each instruction will indicate its termination with a special action, $\overline{\mathtt{done}}$.

We will require that each instruction, of our SCCS description, indicate its termination using the action $\overline{\mathtt{done}}$. To do this we redefine the agent *Done* to the following:

$$Done \overset{\mathrm{def}}{=} \overline{\mathtt{done}} : \mathbf{1}$$

The action $\overline{\mathtt{done}}$ is a termination signal — no other agent should do a $\mathtt{done}$ and synchronize with $\overline{\texttt{done}}$.

In addition, we borrow the notion of a *well-terminating* agent from Milner [68].

Definition 6 An agent P is *well-terminating* if,

1. for every $P \xrightarrow{\alpha} P'$ $\alpha \neq \texttt{done}$.

2. if $P' \xrightarrow{\overline{\text{done}}} P'$ then $P' \sim \mathit{Done}$.

The first part of the definition, (1), says the action $\overline{\texttt{done}}$ is reserved to indicate termination and that no other agent can execute the action $\texttt{done}$ and synchronize with $\overline{\texttt{done}}$. The second part of the definition says, that, if any agent does perform a $\overline{\texttt{done}}$, it is the last action it performs before changing to the idle agent **1**. The definition does not require that P terminate, only that, if it does terminate, then it indicates its termination with the action $\overline{\texttt{done}}$.

Now we can precisely compute $\mathsf{length}(i)$ inductively on the structure of SCCS terms for any agent that is well-terminating.

$$
\begin{aligned}
\mathsf{length}(\mathbf{0}) &= 0 \\
\mathsf{length}(\mathit{Done}) &= 0 \\
\mathsf{length}(\alpha : P) &= 1 + \mathsf{length}(P) & \alpha \neq \overline{\texttt{done}} \\
\mathsf{length}(P + Q) &= \max(\mathsf{length}(P), \mathsf{length}(Q)) \\
\mathsf{length}(P \times Q) &= \max(\mathsf{length}(P), \mathsf{length}(Q)) \\
\mathsf{length}(P \upharpoonright S) &= \mathsf{length}(P) \\
\mathsf{length}(P[f]) &= \mathsf{length}(P) \\
\mathsf{length}(A \overset{\text{def}}{=} P) &= \mathsf{length}(P)
\end{aligned}
$$

For example,

$$\mathsf{length}(\mathsf{a}\!:\!\mathsf{b}\!:\!\mathsf{c}\!:\!\mathbf{0} + d : \mathbf{0}) = 3$$

.

10.2 Derivation of Scheduling Parameters

Given the definition of the instruction scheduling problem from chapter 9, there are two tasks:

- Given a program dependence graph $G = (V, E)$ label each edge $e \in E$ with $d(e)$, the minimal latency.
- For each instruction, i, construct the resource usage function, ρ_i.

We first present an algorithm for deriving the delay function d from our processor specification. Essentially, the delay for an instruction pair (i, j) is calculated by initiating i and then observing how long until j can begin. Most architectures resolve WAR and WAW hazards and schedulers only have to deal with RAW hazards. We will be more general and calculate the delay for every instruction pair (i, j) in the presence of a hazard h, where $h \in \{\texttt{RAW}, \texttt{WAR}, \texttt{WAW}\}$. Now that we have a table of latencies we can correctly label a program DAG.

10.2.1 The Algorithm

The algorithm to calculate instruction latencies from the SCCS processor description is given in Figure 10.1. *Construct_Latency_Function* works by executing an instruction i and counting how long until a subsequent instruction j can begin after i starts. It does this for all possible pairs of opcodes for each data hazard. Recall that the instruction latency is the amount of time between initiating i and initiating j (definition 3). Notice, however, that it is still possible that after j begins it may stall for some other reason (which will most likely be a resource hazard).

Equivalence With Respect to Hazards

In order to decrease the number of instruction pairs that we need to calculate latencies for we do not generate all possible pairs of instructions but all possible pairs modulo the three types of hazards. That is, we consider two instruction pairs to have the same latency if they are the equivalent up to hazards. For example, we expect that the two instruction pairs below have the same latency even though they use different integer registers.

```
Add R1, R1, R1                  Add R2, R1, R1
Mul R2, R1, R1                  Mul R3, R2, R2
```

Both instruction pairs have a RAW hazard; the pair on the left on `R1` and the pair on the right on `R2`. The assumption is that the register file has a reasonable amount of orthogonality as we would expect that the time it takes to access register i is equal to the time it takes to access register j. If the architecture does not possess this orthogonality then we could inspect all register pairs for all combinations of registers. But the search space is reduced considerably if we treat all registers equivalently.

To see this, if an architecture has ten 3-operand instructions and 32 registers, the number of possible instruction pairs is $(10 \cdot 32^3)^2 = 1,073,217,600$. Examining instruction pairs up to hazards equivalence only requires that we check $3 \cdot 10^2 = 300$ pairs.

Definition 7 Let I be the set of instructions, let function $\mathsf{Opcode}(i \in I)$ return the opcode of instruction i, and function $\mathsf{Hazard}(i, j)$ return the hazard occurring between instruction i followed by instruction j. Two instruction pairs (i_1, i_2) and (j_1, j_2) are said to be *equivalent up to hazard*, written

$$(i_1, i_2) \overset{\text{haz}}{=} (j_1, j_2)$$

if

- $\mathsf{Opcode}(i_1) = \mathsf{Opcode}(j_1)$ and $\mathsf{Opcode}(i_2) = \mathsf{Opcode}(j_2)$
- $\mathsf{Hazard}(i_1, i_2) = \mathsf{Hazard}(j_1, j_2)$

Theorem 10.1 $\overset{\text{haz}}{=}$ is an equivalence relation.
Proof: We need to show that $\overset{\text{haz}}{=}$ is reflexive, symmetric, and transitive.

(reflexive) Clearly $(i_1, i_2) \overset{\text{haz}}{=} (i_1, i_2)$ as the $\mathsf{Opcode}(i_1) = \mathsf{Opcode}(i_1)$ and $\mathsf{Hazard}(i_1) = \mathsf{Hazard}(i_1)$.

(symmetric) Show that if $(i_1, i_2) \overset{\text{haz}}{=} (j_1, j_2)$ then $(j_1, j_2) \overset{\text{haz}}{=} (i_1, i_2)$. If $(i_1, i_2) \overset{\text{haz}}{=} (j_1, j_2)$ then $\mathsf{Opcode}(i_1) = \mathsf{Opcode}(j_1)$ and $\mathsf{Hazard}(i_1) = \mathsf{Hazard}(j_1)$, which is all we need to show.

(transitive) Show that if $(i_1, i_2) \overset{\text{haz}}{=} (j_1, j_2)$ and $(j_1, j_2) \overset{\text{haz}}{=} (k_1, k_2)$ then $(i_1, i_2) \overset{\text{haz}}{=} (k_1, k_2)$. By the antecedent of the "if" $\mathsf{Opcode}(i_1) = \mathsf{Opcode}(j_1)$ and $\mathsf{Opcode}(j_1) = \mathsf{Opcode}(k_1)$. Then it follows that $\mathsf{Opcode}(i_1) = \mathsf{Opcode}(k_1)$. Also by the antecedent of the "if" $\mathsf{Hazard}(i_1) = \mathsf{Hazard}(j_1)$ and $\mathsf{Hazard}(j_1) = \mathsf{Hazard}(k_1)$. Then it follows that $\mathsf{Hazard}(i_1) = \mathsf{Hazard}(k_1)$. This is all we need to show.

$\square$

Algorithm Construct_Latency_Function

Algorithm *Construct_Latency_Function* requires, as input, the labeled transition system $\langle \mathcal{P}, \sigma_0, Act, \longrightarrow \rangle$ (see Section 5.6) of the processor specification. Consider an instruction pair (i, j) having hazard h between them. Essentially, we initiate instruction i and execute $\mathtt{Nop}$ instructions until we reach a state σ'' such that the transition $\sigma'' \xrightarrow{j} \sigma'''$ is possible. That is, if we execute i at time t then we subsequently try to execute j at time $t + 1$, and if we cannot then we try executing j at time $t + 2$ and so on. Eventually j will be able to execute at some future time $t + n$ and we can conclude that the latency, $d(i, h, j)$, is n.

If we let $(\xrightarrow{\alpha})^*$ mean zero or more consecutive occurrences of the action α then we can visualize the latency between two instructions, i and j, in terms of the transition system:

$$\overbrace{\sigma_0 \xrightarrow{i} \sigma'}^{\text{first cycle of } i} \underbrace{(\xrightarrow{1})}_{\text{latency}}{}^{*} \overbrace{\sigma'' \xrightarrow{j} \sigma'''}^{\text{first cycle of } j} \longrightarrow \cdots \tag{10.1}$$

We now prove two theorems about algorithm *Construct_Latency_Function.* For all of the remaining theorems we assume that the processor is specified in the normal form laid out at the end of chapter 6 — that when an instruction acquires a register or resource it eventually releases the same register or resource.

Lemma 10.1 Algorithm *Construct_Latency_Function* terminates.

Proof: Clearly, the outer for-loop terminates as the set, $\mathsf{Opcodes} \times \mathsf{Hazards} \times \mathsf{Opcodes}$ is finite.

Consider an arbitrary instruction pair (i', j'). The only way for the while-loop to not terminate is if instruction j' is blocked forever. That is, no σ' exists for which $\sigma' \overset{j'}{\not\longrightarrow}$. Since the only thing that can block j' is a locked register or resource and, by our assumption, every register/resource must eventually be unlocked then eventually $\sigma' \xrightarrow{j'} \sigma''$ and the while-loop will terminate.

□

In the worst case, j' can begin after i' terminates. Since i' has terminated i' must have unlocked all of its registers/resources and the transition $\sigma' \xrightarrow{j'} \sigma''$ is possible and the while-loop terminates. In this case,

$$d(i, h, j) = \mathsf{length}(i)$$

However, we would like to show a stronger result; that $d(i, h, j)$ is the *minimal* latency.

```
function Construct_Latency_Function($\langle \mathcal{P}, \sigma_0, Act, \longrightarrow \rangle$)
    let
        Opcodes = {Add, Fadd, BZ, ...} and
        Hazards = {RAW, WAR, WAW}
    in
        for each $(i, h, j) \in$ Opcodes $\times$ Hazards $\times$ Opcodes do
            1) Construct an instruction pair, $(i', j')$ s.t. $i'$ uses opcode $i$,
                $j'$ uses opcode $j$, and hazard $h$ exists from $i'$ to $j'$.
            2) let $\sigma'$ be state s.t. $\sigma_0 \xrightarrow{i'} \sigma'$
            3) delay := 1;
            while there is no transition $\sigma' \xrightarrow{j'} \sigma''$ do
                1) delay := delay + 1
                2) let $\sigma_{\text{next}}$ be state s.t. $\sigma' \xrightarrow{1} \sigma_{\text{next}}$
                3) $\sigma' := \sigma_{\text{next}}$
            end while
            $d(i, h, j) =$ delay
        end for
    end let
    return($d$)
end Construct_Latency_Function
```

Figure 10.1: **Algorithm that derives instruction latencies.**

Theorem 10.2 Algorithm *Construct_Latency_Function* constructs d, the minimal latency function.

Proof: We need to show that an arbitrary triple $(i, h, j) \in \mathsf{Opcodes} \times \mathsf{Hazards} \times \mathsf{Opcodes}$ has a minimal latency, $d(i, h, j)$. Let the instruction pair (i', j') be the pair constructed in the first statement of the for-loop. Instruction i' can start executing (*i.e.,* statement 2 of the for-loop is valid) since no other instructions are currently executing that may block i'. There are two cases to consider: 1) instruction i' is not blocking j' and 2) instruction i' is blocking j'.

case 1: i' is not blocking j'. In this case, the while-loop is never entered as the only transitions possible are

$$\sigma_0 \xrightarrow{i'} \sigma' \xrightarrow{j'} \sigma''$$

and $d(i, h, j) = 1$.

case 2: i' is blocking j'. We only need to show that j' starts as early as possible. That is, that the latency is not too large. If $\sigma \xrightarrow{i'}$ at time t then we try to execute j' at time $t + 1$. If $\sigma' \overset{j'}{\not\rightarrow}$ then we issue the idle action and reach a state σ' and try to execute j' again. Eventually, $\sigma' \xrightarrow{j'} \sigma''$ at time n because, by the assumption that i' is in normal form, it will eventually release all of its registers/resources that j' depends on.

The second case of the proof essentially says that, if i' started execution at time t, then i' will eventually unlock register r at some time $t + n$ (by the assumption that instruction i' is in normal form). The latency is then $n + 1$. If the current cycle of i' is n (*i.e.,* time $t + n$) then the variable $\mathsf{delay} = n$. Now, $\sigma' \overset{j'}{\not\rightarrow}$ and we enter the body of the while loop and:

1. delay is set to $n + 1$;
2. the algorithm issues the idle action 1 (*i.e.,* the transition $\sigma' \xrightarrow{1} \sigma''$ is taken). But now, on this transition i' unlocks r (because this is cycle n) and, therefore, in state σ'' (time $t + n + 1$), r is unlocked and i' is no longer blocking j'.

The algorithm tries to execute the loop body again but the transition $\sigma' \xrightarrow{j'} \sigma''$ is possible and the loop terminates with $\mathsf{delay} = n + 1$.

□

Notice that the latency between two instructions, i' and j', that suffer a data hazard may be equal to the length of i'. In the second case, it is possible that i' may lock a resource causing j' to stall after j' has begun executing. Here, j' has already started, but that is all that is needed to calculate the latency since the latency represents the least amount of time, in cycles, that must pass after i' *begins* executing before j' can *begin*. The resource vectors of the instructions will yield information allowing the scheduler to optimize for resource conflicts.

Theorem 10.3 The time complexity of *Construct_Latency_Function* is $O(mn^2)$ where m is length of the longest instruction and n is the number of opcodes for the architecture.
Proof: The for-loop in algorithm iterates $3n^2$ times, which is $O(n^2)$ (where $3n^2 = |\mathsf{Opcodes} \times \mathsf{Hazards} \times \mathsf{Opcodes}|$). Since instructions terminate, the while-loop iterates, at most, $m = \max_{i \in \mathrm{Opcodes}} \mathsf{length}(i)$ times, which is $O(m)$. Hence, algorithm *Construct_Latency_Function* is $O(mn^2)$.
$\square$

10.3 Determining Illegal Instruction Sequences

We need to be able to determine which instruction sequences are illegal (*e.g.,* incorrectly using the load or branch delay slots). If an illegal instruction sequence is executed then our simulated processor deadlocks (recall Figure 8.2). In our example microprocessor, referring to a register being loaded while it is locked (Equation 6.10) or executing a branch instruction in the branch delay slot (Equation 6.22) constitute illegal sequences. We would expect these to be the only situations where illegal combinations of instructions arise, but there could be others, so we must check.

10.3.1 A Modal Logic for SCCS

At the expense of some preliminary discussion we will make our algorithms more clear and concise by employing a modal logic defined in terms of labeled transition systems. The logic lets us describe properties in terms of a logical formula, we may then check whether the transition system (process) satisfies this logic formula [93]. We introduce the logic informally through an example. The following grammar generates the formulae of the logic where $K \in Act$.

$$A ::= \textbf{true} \mid \textbf{false} \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid \neg A \mid [K]A \mid \langle K \rangle A \mid \mu x.A$$

The logic is essentially the propositional calculus (boolean algebra) with two additional modal operators, $[K]$ and $\langle K \rangle$. ($[K]$ and $\langle K \rangle$ represent "necessity" and "possibility" from classical modal logic, usually denoted $\Box$ and $\Diamond$.) The calculus also includes the operator $\mu x.A$ which allows us to write recursive logic equations.

If K is a set of actions then a process P satisfies $[K]A$, written $P \models [K]A$ iff P can do every action in K and subsequently satisfy A. $\langle K\rangle$ is a dual of $[K]$ and $P \models \langle K\rangle A$ iff after *some* action in K, $P \models A$. Every process P has the property **true**. That is,

$$\forall P \in \mathcal{P}.P \models \mathbf{true}$$

and no agent has the property **false**:

$$\forall P \in \mathcal{P}.P \not\models \mathbf{false}.$$

As an example, consider a process V that represents a vending machine that can accept one quarter or a half dollar coin and yields a small (1 quarter) or large (half dollar) candy (this example is borrowed from [93]). The actions `1q` and `hd` represent inserting a quarter or a half dollar.

$$V \overset{\text{def}}{=} \mathtt{hd} : \mathtt{big} : \mathtt{collect} : V + \mathtt{1q} : \mathtt{little} : \mathtt{collect} : V$$

Here, $V \models \langle\mathtt{1q}\rangle\mathbf{true}$, that is V can perform the action `1q` (V can accept a quarter). A candy cannot be retrieved before a quarter is inserted. That is, $V \models [\mathtt{big,little}]\mathbf{false}$. Some more properties include:

- $V \models [\mathtt{hd}]([\mathtt{little}]\mathbf{false} \wedge \langle\mathtt{big}\rangle\mathbf{true})$: after a half dollar is inserted we cannot get a little candy but we can get a big one.
- $V \models [\mathtt{1q,hd}][\mathtt{1q,hd}]\mathbf{false}$: after one coin is inserted no more coins can be inserted.
- $V \models [\mathtt{1q,hd}][\mathtt{big,little}]\langle\mathtt{collect}\rangle\mathbf{true}$: after a coin is inserted and a candy is returned we can collect it.

We can express *immediate deadlock* in the logic with the formula

$$\mathit{Deadlock} \overset{\text{def}}{=} [-]\mathbf{false}.$$

Here, "—" is a wild card that represents the entire action set. If P satisfies *Deadlock* then P can't do any action. The process $\mathbf{0}$ possesses no actions and represents the deadlocked agent, consequently, $\mathbf{0} \models \mathit{Deadlock}$.

We can also encode the temporal logic operator $\mathsf{eventually}$ into the logic[1]. Here, $P \models \mathsf{eventually}(A)$ when P eventually evolves to a process P' that satisfies A. That is,

$$P \longrightarrow \cdots \longrightarrow P' \wedge P' \models A$$

$P \models \mathsf{eventually}(\mathit{Deadlock})$ if the process P eventually deadlocks (as opposed to immediately deadlocks).

10.3.2 Using the logic in a processor

We first consider illegal instruction *pairs.* Essentially, we detect these by executing all instruction pairs and examining which cause the processor to deadlock. If CPU is our SCCS process that represents our microprocessor then

$$CPU \models [\texttt{Load R1, (R2)}][\texttt{Add R3,R1,R1}]\mathsf{eventually}(\mathit{Deadlock})$$

means that CPU deadlocks when this particular `Load` instruction followed by the `Add` instruction is executed. Hence, this instruction sequence is illegal. Our intuition suggests that, for our example microprocessor that we have been studying, any instruction pair where the first instruction is a `Load` and the second instruction uses the register being loaded is illegal.

Another illegal instruction pair is a `BZ` instruction followed by a `BZ` instruction. So we expect that

$$CPU \models [\texttt{BZ R}_i\texttt{, Locn1}][\texttt{BZ R}_j\texttt{, Locn2}]\mathsf{eventually}(\mathit{Deadlock}).$$

Algorithm *Detect_Illegal_Instruction_Pairs* (Figure 10.2) examines illegal instruction *pairs.* When the algorithm detects an illegal pair (i, h, j) it records the error value $\perp$ in the table entry for that pair. That is, $d(i, h, j) = \perp$.

It is possible to extend this method to all illegal instruction sequences of length n. For example, we could check to see if, for an instruction sequence $i_1, i_2, \cdots i_n$ is legal with:

$$CPU \models [i_1] \cdots [i_n]\mathsf{eventually}(\mathit{Deadlock})$$

but this could lead to combinatorial problems. Fortunately, n is usually quite small.

[1]Encoding the operator $\mathsf{eventually}$ requires using the least fixpoint operator, μ, which represents a certain solution to recursive equations written in the logic (*e.g.,* an equation such as $Z \overset{\text{def}}{=} \langle\alpha\rangle Z$). Specifically from [98] we have,

$$\mathsf{eventually}(B) \overset{\text{def}}{=} \mu X.(B \vee (\langle - \rangle \mathbf{true} \wedge [-]X))$$

```
function Detect_Illegal_Instruction_Pairs($\langle \mathcal{P}, \sigma_0, Act, \longrightarrow \rangle, d$)
    let
        Opcodes = {Add, Fadd, BZ, ...} and
        Hazards = {RAW, WAR, WAW, NoHazard}
    in
        for each $(i, h, j) \in$ Opcodes $\times$ Hazards $\times$ Opcodes do
            1) Construct an instruction pair, $(i', j')$ s.t. $i'$ uses opcode $i$,
               $j'$ uses opcode $j$, and hazard $h$ exists from $i'$ to $j'$.
            2) if $\longrightarrow \models [i'][j']$eventually($Deadlock$) then
                 $d(i, h, j) = \perp$
        end for
    end let
    return($d$)
end Detect_Illegal_Instruction_Pairs
```

Figure 10.2: **Algorithm to detect illegal instruction pairs.**

Theorem 10.4 Algorithm *Detect_Illegal_Instruction_Pairs* correctly identifies illegal instruction pairs.

Proof: By the assumption that the specification is in the normal form laid out at the end of chapter 6 and the correctness of the satisfaction relation $\models$. The correctness of $\models$ is well-known [94] and is computable for finite-state systems.

$\square$

The only interesting part of algorithm *Detect_Illegal_Instruction_Pairs* is the second statement of the for-loop, checking the satisfaction relation, $\models$. Here we have used the modal logic to do all of the transition checking and rely on the correct implementation of the logic.

10.4 Determining Possible Multiple-Issue Instructions

We can also identify which instructions can be issued in parallel using the modal logic. An instruction is an action in SCCS, and two instructions, i and j, executing in parallel, is the product of these actions, $i \cdot j$. So if i and j can execute in parallel then the transition $\sigma' \xrightarrow{i \cdot j} \sigma''$ is possible (Figure 8.4 shows this case).

In terms of the logic then, we expect that $CPU \models \langle i \cdot j \rangle \mathbf{true}$ if i and j can execute in parallel. For example, if our processor can execute an integer instruction in parallel with a floating-point instruction then

$$CPU \models \langle \texttt{Fadd FR1, FR2, FR3} \cdot \texttt{Mov R1,R2} \rangle \mathbf{true}.$$

In this situation we consider the latency between the two instructions to be zero. That is, $d(i, h, j) = 0$.

Algorithm *Detect_Multiple_Issue_Pairs* (Figure 10.3) only examines instruction *pairs*. As in the case for illegal instruction sequences, we can extend this method to all instruction sequences of length n but, again, this leads to combinatorial problems. Fortunately, n is usually quite small and there is some evidence that there is a small lower bound (less than 10) on the amount of available instruction-level-parallelism [86, 97]. For all current superscalar architectures, n is less than five. The IBM RS/6000, which has the largest degree of parallelism, is capable of issuing four instructions/cycle.

Theorem 10.5 Algorithm *Detect_Multiple_Issue_Pairs* correctly identifies instruction pairs that can be issued simultaneously.

Proof: By the correctness of $\models$ [94].

$\square$

```
function Detect_Multiple_Issue_Pairs($(\langle \mathcal{P}, \sigma_0, Act, \longrightarrow \rangle, d)$)
    let
        Opcodes = {Add, Fadd, BZ, ...} and
        Hazards = {RAW, WAR, WAW, NoHazard}
    in
        for each $(i, h, j) \in$ Opcodes $\times$ Hazards $\times$ Opcodes do
            1) Construct an instruction pair, $(i', j')$ s.t. $i'$ uses opcode $i$,
               $j'$ uses opcode $j$, and hazard $h$ exists from $i'$ to $j'$.
            2) if $\longrightarrow \models \langle i' \cdot j' \rangle$true then
                $d(i', h, j') = 0$
        end for
    end let
    return($d$)
end Detect_Multiple_Issue_Pairs
```

Figure 10.3: **Algorithm to detect multiple instruction issue pairs.**

As in the case for checking illegal instruction sequences, the only interesting part of algorithm *Detect_Multiple_Issue_Pairs* is the second statement of the for-loop, checking the satisfaction relation, $\models$. Again we have used the modal logic to do all of the transition checking and rely on the correct implementation of the logic.

10.5 Computing the Resource Usage Functions

In this section we will compute the resource usage function, ρ_i, for each instruction i of the SCCS processor specification. First, some definitions.

10.5.1 Particulate Actions and Agent Sorts

Recall that every action, α, is uniquely expressible as a product of powers of particulate actions, that is, $\alpha = \alpha_1^{n_1}\alpha_2^{n_2}\cdots\alpha_k^{n_k}$, where each α_i is either a or $\overline{a}$. We denote the particles of an action α as $\mathsf{Part}(\alpha)$, where $\mathsf{Part}(\alpha) = \{\alpha_1, \ldots, \alpha_k\}$. We need to categorize the process by the actions that they may eventually execute. This categorization is called a *sort* and is like a "type" in a programming language.

Definition 8 The *action sort* of an agent P, denoted $\mathsf{Sort}(P)$, is the set of actions that P executes s.t. P executes action α implies $\alpha \in \mathsf{Sort}(P)$.

The converse of the implication does not have to hold so $\mathsf{Sort}(P)$ can be too big. That is, if $\mathsf{Sort}(P) = S$ and $S \subseteq T$ then $\mathsf{Sort}(P) = T$.

Definition 9 The *particle sort* of P, denoted $\mathsf{PartSort}(P)$, is the set of particles of $\mathsf{Sort}(P)$ s.t. $\mathsf{PartSort}(P) = \{\mathsf{Part}(\alpha) \mid \alpha \in \mathsf{Sort}(P)\}$.

The particle sort of an SCCS agent is defined inductively on the structure of SCCS terms.

$$\begin{array}{rcl}
\mathsf{PartSort}(\mathbf{0}) & = & \emptyset \\
\mathsf{PartSort}(Done) & = & \emptyset \\
\mathsf{PartSort}(\alpha : P) & = & \mathsf{Part}(\alpha) \cup \mathsf{PartSort}(P) \\
\mathsf{PartSort}(P + Q) & = & \mathsf{PartSort}(P) \cup \mathsf{PartSort}(Q) \\
\mathsf{PartSort}(P \times Q) & = & \mathsf{PartSort}(P) \cup \mathsf{PartSort}(Q) \\
\mathsf{PartSort}(P \uparrow S) & = & \mathsf{PartSort}(P) - \mathsf{Part}(S)
\end{array}$$

$$\begin{aligned}\mathsf{PartSort}(P[f]) &= \mathsf{Part}(f(\mathsf{Sort}(P)))\\ \mathsf{PartSort}(A \overset{\text{def}}{=} P) &= \mathsf{PartSort}(P)\end{aligned}$$

For example,

$$\mathsf{Sort}(A \overset{\text{def}}{=} \mathtt{a} : \mathit{Done} \times \mathtt{b} : \mathit{Done} \times 1 : A) = \{\mathtt{ab}, 1\}$$

and

$$\mathsf{PartSort}(A) = \{\mathtt{a}, \mathtt{b}, 1\}.$$

10.5.2 Resources and Actions

The only reason we need to introduce sorts at all is so that we can determine what resources, if any, an instruction requires. If R is the set of resources of a processor (section 9.1.1) let *PartR* be the set of particles used to acquire and release these resources (Equation 10.2).

$$\mathit{PartR} = \{\mathtt{get}_i \mid i \in R\} \cup \{\mathtt{put}_i \mid i \in R\} \qquad (10.2)$$

The set of particles that an instruction i uses to access all of its resources is defined by $\mathsf{PartSort}(P) \cap \mathit{PartR}$ where P is the SCCS agent that describes i. For example, $\mathsf{PartSort}(P) \cap \mathit{PartR}$ where P is the divide instruction of Equation 10.3 is the set

$$\{\mathtt{get\_adder}, \mathtt{put\_adder}, \mathtt{get\_divider}, \mathtt{release\_divider}\}$$

.

10.5.3 Deriving the Resource Usage Functions

Recall that the resource usage function ρ_i for an instruction i precisely specifies what resources i uses on each cycle of its execution. To derive the resource usage function for i we simply execute i, in isolation, and observe what resources i acquires and releases and also when those resources are acquired and released.

An instruction is using a resource from the time it gets that resource (using **get**) to the time it releases that resource (using **release**). We can visualize this in terms of the transition system for the instruction:

$$\sigma_0 \longrightarrow \cdots \longrightarrow \sigma \underbrace{\xrightarrow{\alpha} \cdots \xrightarrow{\beta}}_{\text{using resource } r} \sigma' \longrightarrow \cdots \longrightarrow \mathit{Done}$$

$$\textbf{where} \quad \mathtt{get}_r \in \mathsf{Part}(\alpha), \ \mathtt{release}_r \in \mathsf{Part}(\beta)$$

```
for each $i \in \mathcal{P}$ s.t. $Sort(i) \cap PartR \neq \emptyset$ do
    let $\longrightarrow$ and $\sigma_0$ be the transition relation and start state of $i$.
    $\sigma' := \sigma_0$; $\mathsf{InUse} := \emptyset$;
    for $cycle := 1$ to $\mathsf{length}(i)$ do
        let $\sigma_{\text{next}}, \alpha$ be state and action s.t. $\sigma' \xrightarrow{\alpha} \sigma_{\text{next}}$;
        $\sigma' := \sigma_{\text{next}}$;
        if $\mathbf{get}_r \in \mathsf{Part}(\alpha)$ and $\mathbf{put}_r \in \mathsf{Part}(\alpha)$ then
            $\rho_i(cycle) := \mathsf{InUse} \cup \{r\}$;
        else if $\mathbf{get}_r \in \mathsf{Part}(\alpha)$ then
            $\rho_i(cycle) := InUse \cup \{r\}$;
            $\mathsf{InUse} := \mathsf{InUse} \cup \{r\}$;
        else if $\mathbf{put}_r \in \mathsf{Part}(\alpha)$ then
            $\rho_i(cycle) := \mathsf{InUse}$;
            $\mathsf{InUse} := \mathsf{InUse} - \{r\}$
        else
            $\rho_i(cycle) := \mathsf{InUse}$
        end if
    end for
end for
```

Figure 10.4: **Algorithm to calculate resource usage functions.**

Figure 10.4 gives the algorithm for computing the resource usage functions.

The algorithm works by examining, for each instruction, the transition graph of the agent that describes that instruction. The algorithm scans the instruction recording the current set of resources that are in begin used with the variable InUse. When a $\mathbf{get}_r$ is encountered, r is added to InUse. When a $\mathbf{put}_r$ is encountered r is removed from InUse. The if-statement in the inner-most for-loop has four cases:

1. If both a $\mathbf{get}_r$ and $\mathbf{put}_r$ are needed on the same cycle then the resource r is needed for only one cycle and the current resources being used are $\mathsf{InUse} \cup \{r\}$.

2. A $\mathbf{get}_r$ (with no $\mathbf{put}_r$) signifies that the instruction begins using r for more than one cycle, so the current resources used are again $\mathsf{InUse} \cup \{r\}$. InUse is updated to include r.

3. A $\mathbf{put}_r$ signifies that the instruction is finished using r *after* the current cycle. The resource r is removed from InUse.

4. If no $\mathbf{get}_r$ or $\mathbf{put}_r$ is encountered on the current cycle then the set of resources currently being used is InUse.

Theorem 10.6 The algorithm in Figure 10.4 correctly computes the resource usage functions of the architecture.

Proof: Consider an arbitrary instruction $i \in \mathcal{P}$ in normal form, a resource $r \in R$, and the transitions of i.

$$\sigma_1 \xrightarrow{\alpha_1} \cdots \sigma_k \xrightarrow{\alpha_k} \cdots \sigma_n \xrightarrow{\alpha_n} \mathit{Done}$$

Consider an $\alpha_m \in \{\alpha_1, \ldots, \alpha_n\}$. We need to show that $r \in \rho_i(m)$ if and only if

1. $\mathbf{get}_r \in \mathsf{Part}(\alpha_m)$ or

2. $\mathbf{put}_r \in \mathsf{Part}(\alpha_m)$ or

3. $\exists \alpha_a, \alpha_b \in \{\alpha_1, \ldots, \alpha_n\}$ such that $a < m < b$ and $\mathbf{get}_r \in \mathsf{Part}(\alpha_a) \wedge \mathbf{put}_r \in \mathsf{Part}(\alpha_b) \wedge \neg\exists \mathbf{put}_r \in \{\alpha_{a+1}, \ldots, \alpha_{b-1}\}$

The first two are obvious from the first three conditions of the "if-statement" in the algorithm. The third statement follows as the transition $\sigma \xrightarrow{\alpha_a} \sigma'$ updates InUse to include r and r is not removed from InUse until the transition $\sigma \xrightarrow{\alpha_b} \sigma'$. Consequently, from the final **else** clause $r \in \rho_i(m)$.

□

Theorem 10.7 The time complexity of the algorithm in Figure 10.4 is $O(mn)$ where m is the number of instructions, n is the length of the longest instruction.

Proof: Clearly the outer for-loop executes at most m times, the number of instructions on the machine. The inner for-loop executes at most n times where n is the length of the longest instruction. Hence, the statement in the inner for-loop executes mn times, which is $O(mnr)$.

The size of R, $|R|$, will typically be small (for example on the MIPS R2000 it is 13 and on the Motorola 88000 it is 16).

An Example

Consider the floating-point divide instruction described in section 6.10.2 which we repeat here.

$$\begin{aligned} \mathit{FDIV}(PC) \overset{\text{def}}{=}\ & \texttt{getm}_{\texttt{PC}}(\texttt{Fdiv, FR}_i\texttt{, FR}_j\texttt{, FR}_k)\overline{\texttt{lockfreg}_i}\texttt{getfr}_j(x)\texttt{getfr}_k(y) : \\ & \overline{\texttt{get\_adder}} \cdot \overline{\texttt{release\_adder}} : \\ & \overline{\texttt{get\_divider}} : (1:)^6\overline{\texttt{release\_divider}} : \\ & \overline{\texttt{get\_adder}} : \overline{\texttt{release\_adder}} : \\ & \overline{\texttt{putfr}_i}(x/y)\overline{\texttt{releasefreg}_i} : \mathit{Done} \end{aligned} \tag{10.3}$$

The algorithm computes the following resource usage function for the floating-point divide instruction, `Fdiv`, described earlier.

$$\rho_{\texttt{Fdiv}}(t) = \begin{cases} \{\texttt{adder}\} & \text{if } t = 2 \\ \{\texttt{divider}\} & \text{if } 3 \le t \le 9 \\ \{\texttt{adder}\} & \text{if } 10 \le t \le 11 \\ \emptyset & \text{otherwise} \end{cases}$$

10.6 Summary

Using the labeled transition systems generated by the operational semantics of SCCS we can have shown how to:

- find instruction latencies.
- determine illegal instruction sequences.

- determine instructions that can be issued in parallel.
- derive the resource usage functions for each instruction.

Chapter 11

Conclusions and Future Research

This thesis has addressed the formal specification of two programmer views of a processor. The *architecture* level describes the processor so that the programmer can write correct programs. The *timing* level describes the processor so that the programmer can write efficient programs.

At the highest level of abstraction, the architecture level, a processor can be viewed as a set of instructions where each instruction is represented as a function from processor state to processor state. One of the contributions of this thesis was to present a way in which this functional view can be represented cleanly using the functional language SML. (Other functional languages would have sufficed, such as Haskell or Miranda, but we chose SML as it has the most advanced data abstraction facilities of any functional language.) We essentially created an abstract data type (ADT) of common architectural operations and implemented the instructions with this ADT.

11.1 Programmer's Timing View

Programmers often need more information than the architecture provides. The architecture is *minimal* in that it only includes information needed to write *correct* programs. However, programmers also want to write efficient programs. Hence, we have the programmer's timing view of instructions. Timing information includes instruction latencies, resource requirements, and multiple instruction issue capabilities.

It is natural to ask whether the functional technique used to specify architecture could be extended and applied for the timing view. It turns out that specifying the temporal and parallel behavior of systems with functional methods is difficult. Our criteria for choosing another method were that the formalism be able to specify time and parallelism explicitly and we chose the synchronous process algebra SCCS.

The second contribution of this thesis, then, is to specify, as abstractly as possible, the timing behavior of a typical RISC architecture. We also showed how to specify superscalar versions of our RISC in which more than one instruction can be issued simultaneously.

Since SCCS has a formally defined operational semantics, our specification is "executable". The abstract machine of the operational semantics, a labeled transition system, (a type of state machine) specifically describes the behavior of systems specified in SCCS. We have implemented our specification on the Concurrency Workbench, a tool for specifying and analyzing systems written in SCCS (or CCS).

11.1.1 Instruction Scheduling

Once a timing view of a processor has been specified we showed what information is required to do instruction scheduling, essentially parameterizing the instruction scheduling problem. Using the the labeled transition system of the specification we then showed how to derive the instruction scheduling parameters from the specification consequently yielding an instruction scheduler for the processor.

11.2 Future Research

The features that we have included in our example processor were those that are related to *instruction scheduling*. It is natural to ask whether SCCS is capable of specifying other features such as interrupts and more sophisticated memory hierarchies, such as instruction and data caches.

11.2.1 Interrupts

One aspect of instruction processors that we did not address are interrupts, traps, and exceptions. Any complete processor specification, however, should include them. Fortunately, SCCS and process algebras in general are capable of specifying interrupts. For example, Milner [68] discusses an interrupt operator where a process F may interrupt a process E, written $E^\wedge F$. Once the interrupt operator is included it is also possible to specify a *Restart* operator allowing processes to be restarted after they have been interrupted. The synchronous process language Esterel also includes operators useful for specifying interrupts such as its "watchdog" and "trap" statements. For example the statement

do S **watching** I

repeatedly executes the statement S until the signal I occurs. Eventually, we should address specifying processor interrupts — though this may not be useful information for compilers, users of the processor may need to know how interrupts are handled.

11.2.2 Cache Model

At present the memory hierarchy we use is only a two-level model of registers and main memory. Though it would complicate our specification of memory, we could add a cache to our specification in a transparent way, without altering the specification of individual instructions, since, in a RISC model only two instructions access memory, `Load` and `Store`. Compiling for efficient use of caches is a current area of active research [22]. Typically, program profilers generate information that describes the program memory access patterns and the compiler rearranges procedures to make efficient use of the instruction cache. Compilers can also rearrange instructions that access data so that the data cache is used more efficiently.

Once a cache model has been included in the specification it may be possible to use this specification to help generate this optimization phase of the compiler.

Other Classes of Processors

In this dissertation we have concentrated on RISC-style architectures. Certainly it would be possible to specify other classes of architecture such as CISCs or VLIWs (Very Long Instruction Word). In a VLIW, the architecture and the organization are inseparable [35].

Specifying a VLIW would essentially require us to specify the individual resources and the data-paths of the processor. Since it is up to the programmer to organize the instructions so that resource uses do not conflict it should, in principle, be easier to specify than a RISC where we needed to provide interlocks. A specification of a VLIW would require altering the algorithms to reflect the volatile nature of the resources.

11.2.3 Verification and Synthesis

One important property of SCCS that we have not used is the ability to prove systems equivalent. In SCCS it is possible to specify a high-level system and to verify that an implementation of the system meets its specification. To do this would require that we specify a processor at a lower "implementation" level.

Verification

For example, the obvious choice is to specify the organization of the processor: the pipeline stages, forwarding paths, etc. Doing this may be tedious but not difficult. The problem is state explosion. That is, the number of states in any processor is tremendous and since the notion of equivalence in SCCS relates states in the implementation with states in the specification, any complete verification would require that all of the states be checked.

There are a couple of techniques that we could use to handle the state explosion problem. The first technique exploits the fact the parts of the processor are regular, that is, they have a repetitive or inductive structure. For example, a 32-bit ALU may be constructed from thirty-two 1-bit ALU's (for example, see Patterson & Hennessy [77]). So a specification of a 16-bit ALU is almost identical to the specification of a 32-bit ALU. As Milner points [68], systems with inductive structure have proofs that are amenable to induction proofs.

One of the principle reasons a processor specification has so many states is that the number of states in a specification increases exponentially as the size of the memory elements increases. For example, a 32-bit register has 2^{32} different states and each 1-bit increase in word-size doubles the number of possible states. There has been some research using *abstraction techniques* (for example, see [24]) where an abstraction function is used to "merge states". For example, if we want to verify the control portion of a processor then we can ignore the values in registers and memory. Using this abstraction all 2^{32} possible states of a register are collapsed into one state. In this case the abstraction function is "congruence modulo one" which creates an equivalence class for each register on the processor. Now, each register, R_i, is the representative for all of the possible 2^{32} states of register R_i. If a processor has n 32-bit registers, the number of possible states is reduced from $(2^{32})^n$ to just n — a dramatic decrease. The majority of the states would come from the control portion of the hardware known as the *control unit.* The control unit likely includes other "state registers" which we could not abstract in the previous manner as they are integral parts of the control unit.

Synthesis

We should also note that it may be possible to synthesize hardware from a specification, since a labeled transition system, at least in the finite case, is really just a state machine and hardware synthesis is often performed from state machines. In fact, there are CAD packages that will synthesize hardware from descriptions of state machines [96].

11.3 Formal Methods

We should also stress that formal specifications are valuable in their own right and are a starting point for a variety of applications. One benefit is that they require the designer to thoughtfully plan the design in a structured manner. As Noam Chomsky points out in [23]:

> ... a formalized theory may automatically provide solutions for many problems other than those for which it was explicitly designed. Obscure and intuition-bound notions can neither lead to absurd conclusions nor provide new and correct ones, and hence they fail to be useful in two important respects.

Starting from a mature and well-defined formal approach made available a large amount of mathematical machinery. Predefined are notions of system equivalence (used for verification), labeled transition system (used in the semantics, for simulation and just about everything else), modal/temporal logic, sorts, etc. Often, whenever we needed to do something with our specification existing results were there to help.

Appendix A

Example circuits in the Concurrency Workbench

In this appendix we show the Workbench implementations some of the example circuits from chapter 4.

A few notes about the SCCS syntax and the CWB-SCCS syntax. 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 $A \times B$.
- Process definition $A \stackrel{\text{def}}{=} P$ is handled by the `bi` command which binds an identifier to a process description.

A.1 Simple Logic Gates

```
**************************************************************
**** Or gate is described by
****
****        Or(k) <= a(i)#b(j)#'out(k):Or(i or j)
****
**************************************************************

bi Or0    a0#b0#'out0:Or0 + a0#b1#'out0:Or1  \
        + a1#b0#'out0:Or1 + a1#b1#'out0:Or1

bi Or1    a0#b0#'out1:Or0 + a0#b1#'out1:Or1  \
        + a1#b0#'out1:Or1 + a1#b1#'out1:Or1

bi Or  Or0 + Or1

*************************************
**** An exclusive-or gate is similar.
*************************************
bi Xor0   a0#b0#'out0:Xor0 + a0#b1#'out0:Xor1  \
        + a1#b0#'out0:Xor1 + a1#b1#'out0:Xor0

bi Xor1   a0#b0#'out1:Xor0 + a0#b1#'out1:Xor1  \
        + a1#b0#'out1:Xor1 + a1#b1#'out1:Xor0

bi Xor  Xor0 + Xor1

**************************************************************
**** And gate is described by
****
****        And(k) <= a(i)#b(j)#'out(k):Nor(i and j)
****
**************************************************************
```

```
bi And0    a0#b0#'out0:And0 + a0#b1#'out0:And0   \
         + a1#b0#'out0:And0 + a1#b1#'out0:And1

bi And1    a0#b0#'out1:And0 + a0#b1#'out1:And0   \
         + a1#b0#'out1:And0 + a1#b1#'out1:And1

bi And   And0 + And1
```

A.1.1 Not Gates

```
**************************************************************
****   A Not gate.
****
****     Not(j) <= a(i)$'out(j):Not(not i)
**************************************************************

bi  Not0  a0#'out0:Not1 + a1#'out0:Not0
bi  Not1  a0#'out1:Not1 + a1#'out1:Not0
bi  Not   Not0 + Not1

bi  NotNot  (Not['alpha0/'out0,'alpha1/'out1] |  \
             Not[alpha0/a0,alpha1/a1])\{a0,a1,'out0,'out1}
```

A.2 Half-Adder

A.2.1 Specification

```
***************************************************************
**** Binary Half-adder specification is described by
****
**** HA_Spec(c,s) <=
****
****    a(i)#b(j)#'carry(c)#'sum(s):HA_Spec(i and j, i or j)
```

```
****
**************************************************************

**** NOTE: There is no HA_Spec10 as it is an unreachable state.

bi HA_Spec00    a0#b0#'carry0#'sum0:HA_Spec00 \
              + a0#b1#'carry0#'sum0:HA_Spec01 \
              + a1#b0#'carry0#'sum0:HA_Spec01 \
              + a1#b1#'carry0#'sum0:HA_Spec11

bi HA_Spec01    a0#b0#'carry0#'sum1:HA_Spec00 \
              + a0#b1#'carry0#'sum1:HA_Spec01 \
              + a1#b0#'carry0#'sum1:HA_Spec01 \
              + a1#b1#'carry0#'sum1:HA_Spec11

bi HA_Spec11    a0#b0#'carry1#'sum1:HA_Spec00 \
              + a0#b1#'carry1#'sum1:HA_Spec01 \
              + a1#b0#'carry1#'sum1:HA_Spec01 \
              + a1#b1#'carry1#'sum1:HA_Spec11

bi HA_Spec  HA_Spec00 + HA_Spec01 + HA_Spec11
```

A.2.2 Implementation

```
**************************************************************
**** Binary Half-adder implementation is implemented with an
**** OR-gate properly connected to an AND-gate.
****
****   HA_Impl(c,s) <= (And(c)[f] | Xor(s)[g])[h]
****
****    where  f = carry/out, g = sum/out, h = in/in#in
****                                               ^^^^^
****
****         workbench says cannot have relabel of form a/j#k,
****         but Milner's paper says you can.
```

```
****         Workbench implements []:Act --> Part, where
****         []:Act --> Act would be nice.
****
**************************************************************

**** A big set of all possible actions
basi All a0^2#b0^2#'carry0#'sum0    a0^2#b0^2#'carry1#'sum1 \
         a0^2#b0^2#'carry0#'sum1    a0^2#b0^2#'carry1#'sum0 \
         a1^2#b0^2#'carry0#'sum0    a1^2#b0^2#'carry1#'sum1 \
         a1^2#b0^2#'carry0#'sum1    a1^2#b0^2#'carry1#'sum0 \
         a1^2#b1^2#'carry0#'sum0    a1^2#b1^2#'carry1#'sum1 \
         a1^2#b1^2#'carry0#'sum1    a1^2#b1^2#'carry1#'sum0 \
         a0^2#b1^2#'carry0#'sum0    a0^2#b1^2#'carry1#'sum1 \
         a0^2#b1^2#'carry0#'sum1    a0^2#b1^2#'carry1#'sum0

bi  HA_Impl  (HA_Impl00 + HA_Impl01 + HA_Impl11)/All

bi  HA_Impl00    (And0['carry0/'out0, 'carry1/'out1] \
                | Xor0['sum0/'out0, 'sum1/'out1])

bi  HA_Impl01    (And0['carry0/'out0, 'carry1/'out1] \
                | Xor1['sum0/'out0, 'sum1/'out1])

bi  HA_Impl11    (And1['carry0/'out0, 'carry1/'out1] \
                | Xor1['sum0/'out0, 'sum1/'out1])

bi HA_Impl'  (And['carry0/'out0, 'carry1/'out1]  \
           |  Xor['sum0/'out0, 'sum1/'out1])/All
```

A.3 Wires

```
************************************************
**** Unit delay wire
```

```
****
****    Wire(j) <= in(i)#'out(j):Wire(i)
****
**** Two delay wire Specification
****
****    Wire''(i,j) <= in(a)#'out(j):Wire''(a,i)
************************************************
bi  Wire0 in0#'out0:Wire0 + in1#'out0:Wire1
bi  Wire1 in0#'out1:Wire0 + in1#'out1:Wire1
bi  Wire Wire0 + Wire1

bi  Wire''00  in0#'out0:Wire''00 + in1#'out0:Wire''10
bi  Wire''01  in0#'out1:Wire''00 + in1#'out1:Wire''10
bi  Wire''10  in0#'out0:Wire''01 + in1#'out0:Wire''11
bi  Wire''11  in0#'out1:Wire''01 + in1#'out1:Wire''11

bi Wire'' Wire''00 + Wire''01 + Wire''10 + Wire''11

************************************************************
**** Implement a two unit delay wire with two pieces of
**** unit delay wire.
****
****   Wire''Impl <= (Wire[f] | Wire[g])\{in(i),out(i)}
****
****   where f = 'alpha(i)/'out(i)  and g = alpha(i)/in(i)
************************************************************
bi Wire''Impl \
      (Wire['alpha0/'out0,'alpha1/'out1]  \
    |  Wire[alpha0/in0,alpha1/in1])\{in0,in1,'out0,'out1}

*************************************************************
**** Notice that:   Wire'' ~ Wire''Impl ~ NotNot
*************************************************************
```

A.4 Full-Adder

A.4.1 Specification

```
************************************************************
**** Full Adder specification
****
****  FA_Spec(c,s) <== a(i)b(j)Cin(k)'carry(c)'sum(s):
****         FA_Spec(if i+j+k >= 2 then 1 else 0,
****                 if i+j+k = 1 or 3 then 1 else 0)
************************************************************
bi FA_Spec  FA_Spec00 + FA_Spec01 + FA_Spec10 + FA_Spec11

bi FA_Spec00    a0#b0#cin0#'carry0#'sum0:FA_Spec00     \
              + a0#b0#cin1#'carry0#'sum0:FA_Spec01     \
              + a0#b1#cin0#'carry0#'sum0:FA_Spec01     \
              + a0#b1#cin1#'carry0#'sum0:FA_Spec10     \
              + a1#b0#cin0#'carry0#'sum0:FA_Spec01     \
              + a1#b0#cin1#'carry0#'sum0:FA_Spec10     \
              + a1#b1#cin0#'carry0#'sum0:FA_Spec10     \
              + a1#b1#cin1#'carry0#'sum0:FA_Spec11

bi FA_Spec01    a0#b0#cin0#'carry0#'sum1:FA_Spec00     \
              + a0#b0#cin1#'carry0#'sum1:FA_Spec01     \
              + a0#b1#cin0#'carry0#'sum1:FA_Spec01     \
              + a0#b1#cin1#'carry0#'sum1:FA_Spec10     \
              + a1#b0#cin0#'carry0#'sum1:FA_Spec01     \
              + a1#b0#cin1#'carry0#'sum1:FA_Spec10     \
              + a1#b1#cin0#'carry0#'sum1:FA_Spec10     \
              + a1#b1#cin1#'carry0#'sum1:FA_Spec11

bi FA_Spec10    a0#b0#cin0#'carry1#'sum0:FA_Spec00     \
              + a0#b0#cin1#'carry1#'sum0:FA_Spec01     \
              + a0#b1#cin0#'carry1#'sum0:FA_Spec01     \
              + a0#b1#cin1#'carry1#'sum0:FA_Spec10     \
              + a1#b0#cin0#'carry1#'sum0:FA_Spec01     \
              + a1#b0#cin1#'carry1#'sum0:FA_Spec10     \
              + a1#b1#cin0#'carry1#'sum0:FA_Spec10     \
              + a1#b1#cin1#'carry1#'sum0:FA_Spec11

bi FA_Spec11    a0#b0#cin0#'carry1#'sum1:FA_Spec00     \
              + a0#b0#cin1#'carry1#'sum1:FA_Spec01     \
              + a0#b1#cin0#'carry1#'sum1:FA_Spec01     \
              + a0#b1#cin1#'carry1#'sum1:FA_Spec10     \
              + a1#b0#cin0#'carry1#'sum1:FA_Spec01     \
              + a1#b0#cin1#'carry1#'sum1:FA_Spec10     \
              + a1#b1#cin0#'carry1#'sum1:FA_Spec10     \
              + a1#b1#cin1#'carry1#'sum1:FA_Spec11
```

A.4.2 Implementation

```
****************************************************************
**** Full adder implementation
****
****  FA_Impl(c,s) <== (HA[f] | HA[g] | Or[h])/{a,b,c,carry,sum}
****
****       f = cin/b, tempsum/a, 'carry'/'carry
****       g = 'tempsum/'sum, 'carry''/'carry
****       h = carry'/a, carry''/b, 'carry/'out
***************************************************************

bi HA HA_Spec

bpsi S a0 a1 b0 b1 carry0 carry1 sum0 sum1 cin0 cin1

bi HA1  HA[cin0/b0,cin1/b1,tempsum0/a0,tempsum1/a1,           \
           'carry'0/'carry0,'carry'1/'carry1]

bi HA2  HA['tempsum0/'sum0,'tempsum1/'sum1,'carry''0/'carry0, \
           'carry''1/'carry1]
```

```
bi Or'  Or[carry'0/a0,carry'1/a1,carry''0/b0,carry''1/b1,      \
           'carry0/'out0,'carry1/'out1]

bi FA_Impl (HA1 | HA2 | Or')\S
```

A.5 Flip-Flop

```
****************************************************************
**** This example of implementing a flip flop with two nor
**** gates and is borrowed from
****      R. Milner, "Calculi For Synchrony and Asynchrony",
****      Theoretical Computer Science, 1983.
****************************************************************
```

A.5.1 Implementation

```
**************************************************************
**** Nor gate is described by
****
****        Nor(k) <= a(i)#b(i)#'out(k):Nor(i nor j)
****
**** where in Sml
****
****        infix nor; fun i nor j = not (i orelse j);
**************************************************************

bi Nor0     a0#b0#'g0:Nor1 + a0#b1#'g0:Nor0 +       \
            a1#b0#'g0:Nor0 + a1#b1#'g0:Nor0

bi Nor1     a0#b0#'g1:Nor1 + a0#b1#'g1:Nor0 +       \
            a1#b0#'g1:Nor0 + a1#b1#'g1:Nor0

*************************************************************
```

```
**** Make a flip flop out of two "properly connected"
**** nor gates.
****
****     FF(m,n) <= (Nor(m)[f] | Nor(n)[g])
****
****       where f = si/ai, gi#ai/outi     where i in {0,1}
****             g = ri/bi, bi#di/outi
****
*************************************************************
bpsi S  s0 s1 r0 r1 g0 g1 d0 d1

bi FF00   (Nor0[s0/a0,s1/a1,'g0#'a0/'g0,'g1#'a1/'g1]      \
        |  Nor0[r0/b0,r1/b1,'b0#'d0/'g0,'b1#'d1/'g1])\S

bi FF01   (Nor0[s0/a0,s1/a1,'g0#'a0/'g0,'g1#'a1/'g1]      \
        |  Nor1[r0/b0,r1/b1,'b0#'d0/'g0,'b1#'d1/'g1])\S

bi FF10   (Nor1[s0/a0,s1/a1,'g0#'a0/'g0,'g1#'a1/'g1]      \
        |  Nor0[r0/b0,r1/b1,'b0#'d0/'g0,'b1#'d1/'g1])\S

bi FF11   (Nor1[s0/a0,s1/a1,'g0#'a0/'g0,'g1#'a1/'g1]      \
        |  Nor1[r0/b0,r1/b1,'b0#'d0/'g0,'b1#'d1/'g1])\S

bi FF   FF00 + FF01 + FF10 + FF11
```

A.5.2 Specification

```
************************************************************
**** A flip flop specification
****
****  FF(m,n) <= s(i)#r(j)#'g(m)#'d(n):FF(i nor n, m nor j)
****
************************************************************

**** Which unfolds to the following.
```

```
bi SpecFF00   s0#r0#'g0#'d0:SpecFF11 + s0#r1#'g0#'d0:SpecFF10  \
            + s1#r0#'g0#'d0:SpecFF01 + s1#r1#'g0#'d0:SpecFF00

bi SpecFF01   s0#r0#'g0#'d1:SpecFF01 + s0#r1#'g0#'d1:SpecFF00  \
            + s1#r0#'g0#'d1:SpecFF01 + s1#r1#'g0#'d1:SpecFF00

bi SpecFF10   s0#r0#'g1#'d0:SpecFF10 + s0#r1#'g1#'d0:SpecFF10  \
            + s1#r0#'g1#'d0:SpecFF00 + s1#r1#'g1#'d0:SpecFF00

bi SpecFF11   s0#r0#'g1#'d1:SpecFF00 + s0#r1#'g1#'d1:SpecFF00  \
            + s1#r0#'g1#'d1:SpecFF00 + s1#r1#'g1#'d1:SpecFF00

bi SpecFF  SpecFF00 + SpecFF01 + SpecFF10 + SpecFF11
```

Appendix B

Our example RISC in the Concurrency Workbench

In this appendix we list how the SCCS specification of our RISC is implemented using the Concurrency Workbench [25, 71].

B.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.
****
**** The syntax "addxyz" represent the add instruction with destination
**** register "x", and source registers "y", and "z".
************************************************************************
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

[1] Alfred V. Aho, Mahadevan Ganapathi, and Steven W.K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[2] Mitch Alsup. Motorola's 88000 family architecture. *IEEE Micro*, pages 48–66, June 1990.

[3] James H. Aylor, Ronald Waxman, and Charles Scarratt. VHDL—feature description and analysis. *IEEE Design and Test of Computers*, pages 17–27, April 1986.

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

[5] Mario R. Barbacci. Instruction set processor specifications (ISPS): The notation and its applications. *IEEE Transactions on Computers*, pages 24–40, January 1981.

[6] Mario R. Barbacci et al. Ada as a hardware description language: An initial report. In *Computer Hardware Description Languages and their Applications*, pages 272–302, 1985.

[7] David Bernstein. Global instruction scheduling for superscalar machines. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255. ACM, June 1991.

[8] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, and implementation. *The Science of Computer Programming*, 19:87–152, 1992.

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

[10] D. Borrione, editor. *Proceedings of the IFIP WG 10.2 Working Conference on, From HDL Descriptions to Provably Correct Circuit Designs.* Springer-Verlag, 1986.

[11] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages with HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design.* IFIP, North-Holland, 1992.

[12] Jonathan Bowen. Formal specification of the ProCos/safemos instruction set. *Microprocessors and Microsystems*, 14(10):631–643, December 1990.

[13] Jonathen Bowen. Formal specification and documentation of microprocessor instruction sets. *Microprocessors and Microprogramming*, pages 223–230, 1987.

[14] David Bradlee, Susan Eggers, and Robert Henry. Integrating register allocation and instruction scheduling for risc's. In *Fourth International Conference on Architectural Support Programming Languages and Operating Systems*, pages 122–131. ACM and IEEE Computer Society, April 1991.

[15] David Bradlee, Robert Henry, and Susan Eggers. The Marion system for retargetable instruction scheduling. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 229–240. ACM, June 1991.

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

[17] 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.

[18] Bishop C. Brock, Warren A. Hunt, and William D. Young. Introduction to a formally defined hardware description lang uage. In *Theorem Provers in Circuit Design*, pages 3–35, 1992.

[19] G.M. Brown. Towards truly delay-insensitive circuit realisations of process algebras. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits.* Springer-Verlag, 1991.

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

[21] 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.

[22] Chi-Hung Chi and Hank Dietz. Unified management of registers and cache using liveness and cac he bypass. In *ACM Conference on Programming Language Design and Implementation*, pages 344–355, 1989.

[23] Noam Chomsky. *Syntactic Structures.* Mouton, 1957.

[24] Edmund M Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *POPL'92, Proceedings of the* 19^{th} *annual symposium on principles of programming languages*, 1992.

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

[26] Avra Cohn. Correctness properties of the viper block model. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.

[27] Todd Cook, Paul Franzon, Ed Harcourt, and Thomas Miller. Behavioral modeling of processors from instruction set specifications. In *Proceedings of the 2nd International Verilog HDL Conference*, 1993.

[28] Todd Cook, Ed Harcourt, Thomas Miller, and Paul Franzon. LISAS: A Language for Instruction Set Architecture Specification. In *First Annual Conference on Harware/Software Co-design.*, 1992.

[29] Todd A. Cook. *Instruction Set Architecture Specification.* PhD thesis, North Carolina State University, Raleigh, NC, 1993. Department of Electrical and Computer Engineering.

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

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

[32] Jack W. Davidson. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):506–526, October 1984.

[33] Jack W. Davidson. A retargetable instruction reorganizer. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 234–241, 1986.

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

[35] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, Cambridge, MA, 1986.

[36] Christopher Fraser and Alan Wendt. Automatic generation of fast optimizing code generators. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.

[37] Christopher W. Fraser. A language for writing code generators. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 238–245, 1989.

[38] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[39] Alfons Geser. A specification of the Intel 8085 microprocessor: A case study. In *Algebraic Methods: Theory, Tools, and Applications*, pages 347–401. Springer-Verlag, 1991.

[40] Sumit Ghosh. Using Ada as an HDL. *IEEE Design and Test of Computers*, pages 30–42, February 1988.

[41] Robert Giegerich. Code selection by inversion of order-sorted derivors. *Theoretical Computer Science*, pages 177–211, 1990.

[42] Joseph A. Goguen. One, none, a hundred thousand specification languages. Technical Report CSLI-87-96, CSLI:Center for the Study of Language and Information, 1987.

[43] Robert Goldblatt. *Logics of Time and Computation.* CSLI: Center for the Study of Language and Information, 1992.

[44] Ganesh Gopalakrishnan. Specification and verification of pipelined hardware in HOP. In *Computer Hardware Description Lnaguages and their Applications*, pages 117–131. Springer-Verlag, 1990.

[45] M.J.C. Gordon. The denotational semantics of sequential machines. *Information Processing Letters*, pages 1–3, 1980.

[46] M.J.C. Gordon. Register transfer systems and their behavior. In *Computer Hardware description Languages and Their Applications*, pages 23–36, 1981.

[47] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[48] Keith Hanna, Neil Daeche, and Gareth Howells. Implementation of the veritas design logic. In *Theorem Provers in Circuit Design*, pages 77–94, 1992.

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

[50] Ed Harcourt, Jon Mauney, and Todd Cook. Formal specification and simulation of instruction-level parallelism. In *Proceedings of the 1994 European Design Automation Conference.* IEEE Computer Society Press, 1994.

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

[52] John M.J. Herbert. Incremental design and formal verification of microcoded microprocessors. In *Theorem Provers in Circuit Design*, pages 157–174, 1992.

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

[54] Paul B. Jackson. Nuprl and its use in circuit design. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design.* IFIP, North-Holland, 1992.

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

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

[57] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328. ACM, June 1988.

[58] Monica Lam. *A Systolic Array Optimizing Compiler.* Kluwer Academic Press, 1989.

[59] Eugene Lawler, Jan Lenstra, Charles Martel, Barbara Simons, and Larry Stockmeyer. Pipeline scheduling: A survey. Technical Report Computer science research report, IBM Research Division, 1990.

[60] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

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

[62] Roger Lipsett. *VHDL—Hardware Description and Design.* Kluwer Academic Press, Boston, 1989. TK7887.5 L57 1989.

[63] Neal Margulis. *i860 Microprocessor Architecture.* Intel, Osborne, McGraw-Hill, 1990.

[64] Thomas Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[65] Jean P. Mermet, editor. *Fundamentals and Standards in Hardware Description Languages*, volume 294 of *NATO ASI Series.* Kluwer Academic Press, 1993.

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

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

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

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

[70] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i and ii. *Information and Computation*, 100(1):1–77, September 1992.

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

[72] John D. Morison. ELLA, a language for the design of digital systems. In Jean P. Mermet, editor, *Fundamentals and Standards in Hardware Description Languages*, volume 294 of *NATO ASI Series*, pages 385–394. Kluwer Academic Press, 1993.

[73] Thomas Müller. Employing finite automata for resource scheduling. In *MICRO-26, Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 12–20, 1993.

[74] Serafín Olcoz and José Manuel Colom. The discrete event simulation semantics of VHDL. In *Proceedings of the International Conference on Simulation and Hardware Description Languages.* SCS Press, 1994.

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

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

[77] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufman, San Mateo, CA, 1994.

[78] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[79] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAMI FN-19, University of Aarhus, Denmark, 1981.

[80] Amir Pnueli. Linear time temporal logic. **R**esearch and **E**ducation in **C**oncurrent **S**ystems. In the **REX** School/Wokshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, the Netherlands.

[81] Todd Proebsting and Chris Fraser. Detecting pipeline structural hazards quickly. In *POPL'94, Proceedings of the* 21^{st} *annual symposium on principles of programming languages*, 1994.

[82] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 256–267. ACM, June 1991.

[83] Rami R. Razouk. The use of Petri Nets for modeling pipelined processors. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988.

[84] David A. Schmidt. *Denotational Semantics, A Methodology for Language Development.* Allyn and Bacon, 1986.

[85] Mary Sheeran. Categories for the working hardware designer. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, 1990.

[86] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on instruction-level parallelism. In *Second International Conference on Architectural Support Programming Languages and Operating Systems*, pages 290–302. ACM and IEEE Computer Society, 1989.

[87] J. M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics.* Cambridge University Press, Cambridge, UK, 1988. QA76.73 Z2 S65 1988.

[88] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, pages 52–64, September 1990.

[89] Richard Stallman. *Using and Porting GNU C.* Free Software Foundation, 1992.

[90] V. Stavridou. *Formal Methods in Circuit Design.* Cambridge University Press, 1993.

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

[92] Eliezer Sternheim, Rajvir Singh, and Yatin Trivedi. *Digital Design with Verilog HDL.* Automata Publishing Co., Cupertino, CA, 1990.

[93] Colin Stirling. An introduction to modal and temporal logics for CCS. In *Lecture Notes in Computer Science, 491*, pages 2–20. Springer-Verlag, 1991.

[94] Colin Stirling. Modal and temporal logics. In Samson Abramsky, Don Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 478–563. Oxford: Clarendon Press, 1993.

[95] Chris Tofts. Describing social insect behavior using process algebra. *Transactions of the Society for Computer Simulation*, 9(4), December 1992.

[96] C.-J. Tseng et al. A versatile finite state machine synthesizer. In *International Conference on Computer Aided Design*, pages 206–209, 1985.

[97] David W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support Programming Languages and Operating Systems*, pages 176–188. ACM and IEEE Computer Society, April 1991.

[98] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.