From Processor Timing Specifications to Static Instruction Scheduling

Ed Harcourt[1], Jon Mauney[2], Todd Cook[3]

[1] Chalmers University of Technology, Göteborg, Sweden
[2] North Carolina State University, Raleigh, NC 27695, USA
[3] Rutgers University, Piscataway, NJ 08855, USA

Abstract. We show how to derive a static instruction scheduler from a formal specification of an instruction-level parallel processor. The mathematical formalism used is SCCS, a synchronous process algebra for specifying timed, concurrent systems. We illustrate the technique by specifying a hypothetical processor that shares many properties of commercial processors (such as the MIPS or SuperSparc) including delayed loads and branches, interlocked floating-point instructions, resource constraints, and multiple instruction issue.
We derive parameters necessary for instruction scheduling by developing algorithms that operate on the labeled transition systems generated by the operational semantics of SCCS. From the labeled transition system we also employ a modal logic, the modal μ-calculus to determine whether there are any illegal instruction sequences or instruction sequences that could be executed in parallel.

1 Introduction

The problem of automatically generating a code-generator from a machine description has been well studied [AGT89, Dav86, Fra89]. Giegerich [Gie90] subsequently pointed out that these descriptions were not grounded in any formal framework and, therefore, while they were useful for compiler generation they were not useful for such important tasks as hardware/compiler verification, synthesis, and simulation.

In most microprocessors, the temporal and concurrent properties of the instructions are visible to the users (compilers) of the processor. For efficiency, the user (instruction scheduler) must reorder program instructions taking these temporal and concurrent constraints into account. Since it would be useful to have an instruction scheduler automatically generated it is necessary, then, to include instruction timing properties in the machine description. In contrast with machine descriptions for code generators, there is very little research dealing with specifying the temporal properties of an architecture and deriving an instruction scheduler [BHE91, PF94] and no research on the *formal specification* of these timing properties. Also, ours is the only method to consider multiple instruction issue capabilities. As a measure of expressiveness, specifications written in the syntax of [BHE91, PF94] can be expressed in our formalism. Also, we are more general as we can specify timing constraints not expressible in [BHE91, PF94].

We present a technique for deriving instruction scheduling information from formal timing specifications of RISC style architectures. We illustrate the technique with a hypothetical processor that shares many properties of commercial processors, including delayed loads and branches, interlocked floating-point instructions, and multiple instruction issue (Superscalar). The specification is formal and suitable for use in verification and simulation [HMC93].

In this paper we briefly introduce our technique of specifying a processor (detailed in [HMC94a, HMC93]) and then show how instruction scheduling information is derived from the specification.

As our mathematical formalism, we use SCCS, a synchronous process algebra designed for specifying timed, concurrent systems [Mil89, Mil83]. There are many reasons for choosing SCCS. First, SCCS allows us to *explicitly* specify the temporal and concurrent properties of a processor. Second, SCCS is formally defined and provides a variety of techniques for proving and verifying properties about SCCS descriptions. Third, there is an available tool, the Concurrency Workbench [CPS93], which allows us to interactively experiment with, analyze, and simulate our SCCS processor descriptions. Finally, SCCS allows us to describe a processor at a variety of levels of abstraction, from a high-level specification to lower organizational and implementation levels.

In specifying the timed/parallel behavior of machine instructions we are concerned with a level of abstraction independent of hardware implementation. For example, in a delayed-load architecture the user needs only be concerned with the delay-slot following a load instruction and not with the cause for the delay. As another example, most pipelined architectures employ forwarding hardware that allows most integer register-register instructions to execute in one cycle. The user should not have to understand the forwarding hardware in order to discover that the instruction sequence

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

is legal.

This research is part of a larger project of designing a general architectural description language [HMC93, CFHM93, HMC94b, CH94].

2 A Synchronous Calculus of Communicating Systems

SCCS, or *Synchronous Calculus of Communicating Systems* [Mil83, Mil89], is a mathematical theory of communicating systems in which we can represent a real system by the *terms* or *expressions* of that system. SCCS allows us to directly represent the temporal and concurrent properties of the system being specified.

Systems specified by SCCS are composed of two entities, actions and processes. Processes are built from expressions whose syntax is given in figure 1. Actions communicate values and can either be *positive* (*e.g.* `in`) or negative (*e.g.* $\overline{\texttt{out}}$). Positive actions input values, and negative actions output values. Two

actions α and $\bar{\alpha}$ associated with two processes running in parallel can communicate by the fact that they are complements of the same name. Each process *must* perform an action (that is, use one or more of its ports) on each clock cycle. A process not wishing to perform an action may execute the *idle* action, written as 1.

We introduce SCCS through an example of a simple pipeline. Consider a two stage pipeline where each stage adds one to its input and has the overall effect of receiving a value v at time t and outputting $v+2$ at time $t+2$.

One stage in our example pipeline is represented in SCCS by

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

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

- ":" is called the prefix operator. In general, the expression $\mathtt{a} : P$ specifies that at time t do action $\mathtt{a}$, and then at time $t+1$ proceed with process P.
- $\mathtt{in}(y)\overline{\mathtt{out}}(x)$ is a *product* of actions specifying that the two particulate actions (or particles), $\mathtt{in}$ and $\overline{\mathtt{out}}$, occur simultaneously. This action can also be thought of as reading y on port $\mathtt{in}$ and sending x on port $\overline{\mathtt{out}}$ where x is the current state of S which was computed in the previous cycle.
- $S(x)$ is defined recursively. Recursive definitions allow for the modeling of non-terminating processes.
- S is parameterized and contains the arithmetic expression $y+1$.

The semantics of SCCS is given formally in [Mil83].

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

Fig. 1. Syntax of SCCS expressions

2.1 Connecting Processes

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

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

$$\begin{aligned} \mathit{Add2}(x, y) \stackrel{\text{def}}{=} & \; (S(x)[\phi_1] \times S(y)[\phi_2]) \uparrow \{\texttt{in, out}\} \\ & \phi_1 = \texttt{out} \mapsto \alpha, \quad \phi_2 = \texttt{in} \mapsto \alpha \end{aligned} \tag{2}$$

In Equation 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 process S.
- $S \uparrow \{\texttt{in, out}\}$ is the *restriction* combinator applied to process S. Restriction alters the scope of an action by "internalizing" (or "hiding") actions from the environment and exposing others. Hence, in this example, `in` and `out` are made known to the environment and α is internalized.

The net effect of Equation 2 is to construct a pipeline of two stages and two external ports where each stage adds 1 to its input.

SCCS includes another combinator, the + combinator, for constructing processes. The process $A + B$ represents a choice of performing process A or process B, where the choice taken depends upon the actions available within the environment. The process $A_1 + A_2 + \cdots A_n$ is abbreviated to $\sum_{i=1}^{n} A_i$.

2.2 Extensions to SCCS

We introduce two extensions to SCCS that will aid us in writing processor specifications. Frequently, we wish to execute two processes A and B in parallel, where B begins executing one clock cycle after A (*e.g.,* issuing instructions on consecutive cycles). This serial operation can be modeled by $A \times (1 : B)$. We define the binary combinator *Next* to denote this process:

$$A \; \mathit{Next} \; B \stackrel{\text{def}}{=} A \times (1 : B)$$

Another useful operator is the *priority sum* operator, $\triangleright$ [CW91]. If in the process $A + B$ both A and B can execute, then it is non-deterministic which one is executed. We can prioritize + so that if both A and B can execute, then A is preferred, denoted by $A \triangleright B$.

3 Specifying a Processor

We now briefly describe the method of specifying a processor in SCCS. More details are provided in [HMC93, HMC94a]. We model a processor as a system of interacting processes where registers and memory interact with one or more functional units. Equation 3 represents such a system at the highest level:

$$Processor \stackrel{\text{def}}{=} (Instruction\ Unit \times Memory \times Registers) \uparrow I \tag{3}$$

where a processor consists of an instruction unit operating in parallel with a memory and registers and I is the set of all instructions.

For each register, i, we define a process with actions $\mathtt{getr}_i(x)$, $\overline{\mathtt{putr}}_i(x)$, $\mathtt{lockreg}_i$, and $\mathtt{releasereg}_i$ for reading, writing, locking, and releasing register i. The lock is required to trap an instruction trying to read a register while it is being loaded from memory. The registers are specified in such a way that SCCS's bottom state, $\mathbf{0}$, is reached when another instruction tries to access a locked register.

The definition of a process *Memory* is exactly analogous to that of *Registers*, except that memory cells do not have locks associated with them. The actions $\mathtt{getm}_i(x)$ and $\overline{\mathtt{putm}}_i(x)$ read and write memory cell i. The definition of processes *Registers* and *Memory* is straightforward and is given in [HMC93].

3.1 Instruction Issue

Given our definitions of *Registers* and *Memory* and using a program counter, PC, we now describe a process $Instr(PC)$ (Equation 4) that specifies the behavior of 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*).

$$\begin{aligned} Instr(PC) &\stackrel{\text{def}}{=} (Non\_Branch(PC)\ \ Next\ \ Instr(PC+4)) \\ &+\ Branch(PC) \\ &\rhd\ Stall(PC) \end{aligned} \tag{4}$$

$$Non\_Branch(PC) \stackrel{\text{def}}{=} Alu(PC) + Load\_Store(PC) + Float(PC) \tag{5}$$

$$Stall(PC) \stackrel{\text{def}}{=} 1 : Instr(PC) \tag{6}$$

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

- A non-branch instruction may execute, in which case the next instruction to execute is at $PC + 4$; the first line of Equation 4 describes this situation.
- A branch instruction may execute. The next instruction to execute cannot be determined until it is known whether the branch will be taken or not. Consequently, the next instruction executed is controlled by $Branch(PC)$.
- If no instruction can execute, then the processor must stall (Equation 6). The $\rhd$ operator (section 2.2) is used here because the processor should stall only when no other alternative is available.

Arithmetic Instructions. Our architecture fetches instructions from memory using a program counter, PC. The action

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

represents fetching an `Add` instruction from memory. One way to understand this is that if $Memory(PC)$ contains the above instruction it now matches (or synchronizes) with the action $\texttt{getm}_{\texttt{PC}}(\texttt{Add R}_i\texttt{,R}_j\texttt{,R}_k)$ defined by $ALU(PC)$.

From a user's view, the instruction `Add` $\texttt{R}_i$, $\texttt{R}_j$, $\texttt{R}_k$ *appears* to take one cycle to execute. The underlying hardware may be more complex, but at our level we are concerned only with external behavior.

The process

$$Alu(PC) \stackrel{\text{def}}{=} \texttt{getm}_{\texttt{PC}}(\texttt{Add R}_i\texttt{,R}_j\texttt{,R}_k)\texttt{getr}_j(x)\texttt{getr}_k(y)\overline{\texttt{putr}_i}(x+y) : Done \quad (7)$$

represents the `Add` instruction: at time t, source registers j and k are read (by the actions $\texttt{getr}_j(x)\texttt{getr}_k(y)$) and the result is written to destination register i (by the action $\overline{\texttt{putr}_i}(x+y)$). In fact, Equation 7 describes the same final *computation* as the register transfer statement

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

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

Load and Store Instructions. In a delayed-load architecture (such as the MIPS R3000) an instruction may not immediately use the register being loaded. The `Load` instruction accesses memory at time t, and the result of the load is available at time $t+2$. This is represented by,

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

Equation 8 specifies that at time t, three things happen:

1. The base register j is accessed, and the base address is placed in the B.
2. Memory is accessed with the value placed in V.
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 $\overline{\texttt{releasereg}_i}$).

The result is not available until time $t+2$.

Equation 8 specifies both the computation of the load instruction and its temporal properties. For instruction scheduling we are only interested in the timing information but, to be complete, we also include the computational aspect of the instructions.

3.2 Interlocked Floating-Point Instructions

Our processor has a separate set of thirty two floating-point registers. Actions $\texttt{lockfreg}_i$, $\texttt{releasefreg}_i$, $\overline{\texttt{putfr}}_i(x)$, and $\texttt{getfr}_i(x)$ lock, release, write, and read floating-point register i. (We should note that the actions $\texttt{lockfreg}_i$ and $\texttt{releasefreg}_i$ model the *scoreboard* register of pipelined processors. The scoreboard is a bit vector where each register has a corresponding bit in the vector which is, essentially, a semaphore.) The definitions are similar to the integer registers and can be found in [HMC93]. One difference, however, is that accessing a locked floating-point register is not illegal, as is the case for the integer registers. Accessing a locked floating-point register causes the processor to stall (with the process *Stall*).

One situation that arises with multi-cycle floating-point instructions is that they usually share a limited set of internal resources, R (*e.g.,* adder, multiplier, rounder, etc.). A particular instruction requires resources from R at various times during its execution. A scheduler must know the resource requirements of the instructions so that it can schedule them appropriately. We model each resource $r \in R$ as a process with two actions, $\texttt{get}_r$, and $\texttt{release}_r$.

The `Fdiv` Instruction. Having defined interlocked floating-point registers and resources, we can specify the behavior of the floating-point divide instruction:

$$
\begin{aligned}
\mathit{Float}(PC) \stackrel{\text{def}}{=}\ & \texttt{getm}_{\text{PC}}(\texttt{Fdiv FR}_i\texttt{,FR}_j\texttt{,FR}_k)\overline{\texttt{lockfreg}_i}\texttt{getfr}_j(x)\texttt{getfr}_k(y) : \\
& \overline{\texttt{get}}_{\texttt{adder}} \cdot \overline{\texttt{release}}_{\texttt{adder}} : \\
& \overline{\texttt{get}}_{\texttt{divider}} : (1:)^6\overline{\texttt{release}}_{\texttt{divider}} : \\
& \overline{\texttt{get}}_{\texttt{adder}} : \overline{\texttt{release}}_{\texttt{adder}} : \\
& \overline{\texttt{putfr}}_i(x/y)\overline{\texttt{releasefreg}_i} : \mathit{Done}
\end{aligned}
\tag{9}
$$

The `Fdiv` instruction

1. accesses its source registers ($\texttt{getfr}_j(x)\texttt{getfr}_k(y)$);
2. locks its destination register ($\overline{\texttt{lockfreg}_i}$);
3. uses the adder for one cycle;
4. uses the divider for eight consecutive cycles;
5. uses the adder for two consecutive cycles;
6. writes the result in the destination register ($\overline{\texttt{putfr}}_i(x/y)$); and
7. releases the destination register ($\overline{\texttt{releasefreg}_i}$).

The abbreviation $(1:)^n$ represents the n-cycle delay, $1:1\ldots1$, which is interpreted as n-cycles of internal computation.

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

4 A Two-Issue Superscalar Processor

This section describes a superscalar 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 is specified by Equation 10.

$$(Float(PC) \times Alu(PC+4)) + (Alu(PC) \times Float(PC+4)) \tag{10}$$

Equation 11 extends Equation 10 by using a summation notation that succinctly allows us to specify n-way parallelism (where $i \neq j$). Equation 11 also continues execution at $PC + 8$:

$$Do\_Two(PC) \stackrel{\text{def}}{=} \left(\sum_{i,j \in \{0,4\}} (Alu(PC+i) \times Float(PC+j)) \right) \; Next \;\; Instr(PC+8) \tag{11}$$

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

If we rename equation 4 as Do_One, the dual issue case can be written as:

$$Instr(PC) \stackrel{\text{def}}{=} Do\_Two(PC) \rhd Do\_One(PC) \rhd Stall(PC) \tag{12}$$

Notice the use of the priority choice operator, $\rhd$ (section 2.2) instead of $+$; whenever it is possible to do Do_Two, it is also possible to do Do_One, and issuing two instructions should take priority over issuing one when possible.

We can also model multiple instruction issue when the instructions have data dependencies between them. For example, if the processor has two integer units, and can issue two integer instructions in parallel, we now have the problem that the two instructions can have data dependencies between them. This is done by using the restriction operator, $\uparrow$, to allow or disallow certain instruction sequences [HMC93].

5 Instruction Scheduling

We now introduce the instruction scheduling problem and show what parameters need to be extracted from the specification. The next section shows how these parameters are derived from the SCCS specification.

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 respect the semantics of S.

There are two types of scheduling constraints—precedence and resource. 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. Data dependencies fall into three categories: true (or forward dependency), anti-, and output. Hardware designers know these dependencies as the three types of data hazards: RAW (read-after-write), WAR (write-after-read), and WAW (write-after-write).

Dependencies are represented graphically by a *directed acyclic graph* (DAG) G where 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 from the program and a directed edge, (i, j), from vertex i to vertex j means j depends on i. The DAG is augmented by labeling each edge, e, with a *minimal latency* (delay), $d(e)$, that represents the least amount of time, in cycles, that must pass after i begins executing before j can begin.

5.1 Resource Constraints

An architecture consists of a *multiset*, R, of resources. Each instruction uses, on each clock cycle that it is executing, resources from R. First, we need a definition.

Definition 1 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.

If an instruction i executes for *length*(i) cycles then the *resource usage function* for instruction i, ρ_i, maps clock cycles to subsets of R (*i.e.*, $\rho_i : t \rightarrow Pow(R)$ *s.t.* $0 \leq t \leq \mathbb{N}$). When $t >$ *length*(i) then $\rho_i(t) = \emptyset$. For example, $\rho_{\mathtt{Add}}(2)$ 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 is not greater than the available resources.

As an example, the resource set of the MIPS/R4000 floating-point unit is given below [KH92].

$R =$ {`unpack, divider, shifter, adder, rounder, mult_stage1, mult_stage2, exception`}

5.2 Deriving an Instruction Scheduler

To build an instruction scheduler we need to determine the following information:

- a table of latencies for all possible pairs (i, j) of instructions. This allows us to annotate each edge in the program dependence graph with $d(e)$ (where $e = (i, j)$), the minimal latency.

– for each instruction, i, 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 it takes 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 an instruction pair (i, j) in the presence of a hazard h, where $h \in \{\texttt{RAW}, \texttt{WAR}, \texttt{WAW}\}$.

In order to "execute" an instruction and capture useful information we need to know how to simulate an SCCS agent. An SCCS agent determines an automaton (in our case finite), called a *labeled transition system* or LTS. It is the LTS that our algorithms use to determine the scheduling information.

Definition 2 *A* **labeled transition system** *(LTS) is a triple* $\langle \mathcal{P}, \mathrm{Act}, \rightarrow \rangle$ *where* $\mathcal{P}$ *is the set of agents in SCCS,* Act *is the set of actions, and* $\rightarrow$ *is the transition relation, a subset of* $\mathcal{P} \times \mathrm{Act} \times \mathcal{P}$*. A state,* σ_0*, can be designated as the start state.*

When $p, q \in \mathcal{P}$ and $\alpha \in Act$ and $(p, \alpha, q) \in \rightarrow$ we write $p \xrightarrow{\alpha} q$ to mean that "agent p can do an α and evolve into q." Agents p and q represent the state of the system at times t and $t + 1$ respectively.

The algorithm to calculate instruction latencies from the SCCS processor description is given in figure 2. *Construct_Latency_Function* works by executing an instruction i and counting cycles until a subsequent instruction j can begin. It does this for all possible pairs of opcodes for each data hazard. The instruction latency is the amount of time between initiating i and initiating j. Notice, however, that it is still possible that after j begins it may stall for some other reason, which will be, most likely, a resource hazard.

Algorithm *Construct_Latency_Function* requires, as input, the labeled transition system $\langle \mathcal{P}, \sigma_0, Act, \rightarrow \rangle$. Essentially, instruction i is initiated and `1` actions are executed 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, $d(i, j)$, is n.

The 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.

5.3 Illegal Instruction Sequences

Algorithm *Construct_Latency_Function* does not detect illegal instruction sequences (*e.g.,* using a busy register in the load delay slot, or having another branch instruction in the branch delay slot). Since illegal instruction sequences cause the SCCS process to deadlock, detecting these amounts to detecting deadlock in the specification (which we can do for our finite-state system). We do this by executing all instruction pairs on the specification and identifying which

function Construct_Latency_Function($\langle \mathcal{P}, \sigma_0, Act, \rightarrow \rangle$)
  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 σ' be state s.t. $\sigma_0 \xrightarrow{i'} \sigma'$
      3) delay := 0;
      while there is no transition $\sigma' \xrightarrow{j'} \sigma''$ **do**
        1) delay := delay + 1
        2) let σ_{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

Fig. 2. Algorithm that derives instruction latencies.

ones deadlock. To do this we use the *modal μ-calculus*, a modal logic defined on the labeled transition systems of SCCS. The formulae of the μ-calculus are generated by the following grammar, where $K \in Act$.

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

The logic is essentially the propositional calculus 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 fixpoint operator $\mu x.A$ which allows us to write recursive logic equations where the variable x may occur in A.

Immediate deadlock (a process cannot execute any action) is expressed by the term

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

which means that a process cannot perform any action. Here, $-$ is a "wildcard" that represents the entire action set Act.

If CPU is the transition system generated by our SCCS specification then

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

implies that our processor cannot execute the above `Load-Add` sequence. In this situation we consider the latency between the two instructions to be the error

value, $\perp$. That is, $d(i,h,j) = \perp$. The operator eventually has a standard encoding using the fixpoint operator μ [Win93].

As in algorithm *Construct_Latency_Function* we can now selectively generate all pairs of instructions and check whether the pair deadlocks. We easily extend this for all instruction sequences of length n. This can potentially lead to combinatorial problems but, fortunately, n is quite small (less than 10).

5.4 Multiple Issue Instructions

Construct_Latency_Function does not consider multiple issue instructions. That is, we would like to know which combinations of instructions can execute in parallel. These again are identified from the labeled transition system by executing instruction sequences in parallel. 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. Detecting parallel instruction sequences is similar to deriving latencies except that, instead of executing instructions sequentially and counting cycles, we execute the instructions in parallel.

Again, we can employ the μ-calculus to determine if $CPU \models \langle i \cdot j \rangle \mathbf{true}$. For example, if our processor can execute an integer instruction in parallel with a floating-point instruction then we expect

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

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

6 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. An instruction's resource requirements are specified in the instruction with actions $\texttt{get}_r$ and $\texttt{put}_r$. In order to determine an instruction's resource requirements we will need to analyze the actions executed by the instructions. In order to do this analysis we need some definitions.

Every action, α, is uniquely expressible as a product of powers of particulate actions, that is, $\alpha = \alpha_1^{n_1}\alpha_2^{n_2}\cdots\alpha_k^{n_k}$. We denote the particles of an action α as $Part(\alpha)$, where $Part(\alpha) = \{\alpha_1, \ldots, \alpha_k\}$.

Definition 3 The *action sort* of an agent P, denoted $ActSort(P)$, is the set of actions that P executes s.t. if P executes action α then $\alpha \in ActSort(P)$.

Definition 4 The *particle sort* of P, denoted $PartSort(P)$, is the set of particles of $ActSort(P)$ s.t. $PartSort(P) = \{Part(\alpha) \mid \alpha \in ActSort(P)\}$.

The particle sort of an SCCS agent is defined in [Mil83].

We need to determine what resources, if any, an instruction requires. If R is the set of resources of a processor let $PartR$ be the set of particles used to acquire and release these resources (equation 13).

$$PartR = \{\texttt{get}_i \mid i \in R\} \cup \{\texttt{put}_i \mid i \in R\} \tag{13}$$

The set of particles that an instruction i uses to access all of its resources is defined by $PartSort(P) \cap PartR$ where P is the SCCS agent that describes i.

6.1 Deriving the Resource Usage Functions

The resource usage function ρ_i, for an instruction i, should precisely specify what resources i uses on each cycle of its execution. To derive ρ_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 r from the time it gets that resource (using $\texttt{get}_r$) to the time it releases that resource (using $\texttt{release}_r$). We can visualize this in terms of the transition system for the instruction:

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

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

Figure 3 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 use by searching for $\texttt{get}_r$ and $\texttt{put}_r$ actions. The if-statement in the inner-most for-loop has four cases:

1. If both a $\texttt{get}_r$ and $\texttt{put}_r$ are needed on the same cycle then the resource r is needed for only one cycle.
2. A $\texttt{get}_r$ signifies that the instruction begins using r.
3. A $\texttt{put}_r$ signifies that the instruction is finished using r *after* the current cycle.
4. If no $\texttt{get}_r$ or $\texttt{put}_r$ is encountered on the current cycle then the set of resources currently being used is InUse.

The time complexity of the algorithm is $O(mnr)$ where m is the number of instructions, n is the length of the longest instruction, and r is the size of the resource set R. However, 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).

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

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

```
for each i ∈ P s.t. Sort(i) ∩ PartR ≠ ∅ do
    let ⟶ and σ0 be the transition relation and start state of i.
    σ' := σ0; InUse := ∅;
    for cycle := 1 to length(i) do
        let σnext, α be state and action s.t. σ' --α--> σnext;
        σ' := σnext;
        if get_r ∈ Part(α) and put_r ∈ Part(α) then
            ρi(cycle) := InUse ∪ {r};
        else if get_r ∈ Part(α) then
            ρi(cycle) := InUse ∪ {r};
            InUse := InUse ∪{r};
        else if put_r ∈ Part(α) then
            ρi(cycle) := InUse;
            InUse := InUse −{r}
        else
            ρi(cycle) := InUse
        end if
    end for
end for
```

Fig. 3. Algorithm to calculate resource usage functions.

7 Discussion and Conclusions

In this paper we have presented a technique for deriving instruction scheduling information from a formal specification of a RISC/Superscalar architecture. The timing properties specified are delayed loads and branches, interlocked floating-point operations, and multiple instruction issue. The mathematical formalism used is SCCS, a synchronous process algebra designed for specifying timed/concurrent systems. An instruction-level parallel processor is, in essence, a set of communicating processes (functional units) which matches the model of computation SCCS was designed to specify.

The algorithms that derive instruction scheduling information operate on the automaton induced by an SCCS process. Also, the algorithm that derives resource usage functions requires that the instructions be specified in the form detailed in section 3.2. This is not restrictive as our formalization coincides, and is based on, the informal descriptions of resource use described in processor manuals. We have implemented our processor specification on the Concurrency Workbench, a verification/simulation tool for SCCS.

References

[AGT89] 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.

[BHE91] 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.

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

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

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

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

[Dav86] Jack W. Davidson. A retargetable instruction reorganizer. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 234–241, 1986.

[Fra89] Christopher W. Fraser. A language for writing code generators. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 238–245, 1989.

[Gie90] Robert Giegerich. On the structure of verifiable code generator specifications. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 1–8, 1990.

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

[HMC94a] 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.

[HMC94b] 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.

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

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

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

[PF94] 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.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.