High-Level Cycle-Accurate Specification of Microprocessors

by

Felix Sheng-Ho Chang

B.Sc., University of British Columbia, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

August 2001

Abstract

This thesis introduces a new specification style for processor microarchitectures. My goal is to produce very simple, compact, but cycle-accurate descriptions that can be automatically simulated efficiently, in order to enable early exploration of different microarchitectures and their performance. The key idea behind my approach is that one can derive the difficult-to-design forwarding and stall logic completely automatically. I have implemented a specification language for pipelined processors, along with an automatic translator that creates cycle-accurate software simulators from the specifications. I have specified a pipelined MIPS integer core in my language. The entire specification is less than 300 lines long and implements all user-mode instructions except for coprocessor support. The resulting, automatically-generated, cycle-accurate simulator achieves over 240,000 instructions per second simulating MIPS machine code. This performance is within an order of magnitude of large, hand-crafted, cycle-accurate simulators, but my specification is far easier to create, read, and modify.

Contents

List of Tables

List of Figures

Acknowledgments

I thank my advisor, Alan Hu, for helping me explore an idea — an idea that turned into a long but fulfilling research project.

Furthermore, I have been blessed with invaluable advice from many individuals, especially Mark Greenstreet whose suggestions have helped me stay on track. Finally, my gratitude goes out to Brian Winters who generously helped me get my writing started; my thesis would not have been completed otherwise.

This research was supported in part by the National Science and Engineering Research Council of Canada.

Felix Sheng-Ho Chang

The University of British Columbia

August 2001

Dedicated to my parents.

Chapter 1

Introduction

1.1 Motivation

Processor design — whether for mainstream commodity products or application-specific processor cores — involves numerous difficult trade-offs between competing goals and constraints. These trade-offs span from instruction set architecture issues to low-level circuit implementations. This thesis focuses on the microarchitectural level, the highest level where the performance effects of pipelining, stalls, functional unit latency, and similar design decisions becomes visible.

The performance effects of microarchitectural decisions are hard to predict. How much performance do we lose if we stall one-cycle for an unusual instruction combination instead of adding an expensive bypass path? How much performance do we gain with a faster functional unit? How does performance change if the typical software workload changes in some way? In practice, one relies heavily on cycle-accurate simulators in order to determine the number of clock cycles needed to execute samples of benchmark software, combined with rough timing and circuit models to estimate the clock frequency and area of a given design.

Unfortunately, writing a cycle-accurate simulator for a given microarchitecture is time-consuming and error-prone, especially since the resulting simulator must run fast enough to simulate non-trivial software. Therefore, simulation can be used to explore only a few design options; other possibilities can be considered only vaguely, using intuition and back-of-envelope calculations.

Our hypothesis is that efficient cycle-accurate simulators can often be rapidly generated from a simple, compact, but cycle-accurate style for specifying microprocessors. If our hypothesis is true, it will enable microprocessor designers to do early exploration of different design options.

1.2 Project

To test the hypothesis, we introduce a novel specification style tailored expressly for describing processor microarchitectures. The specification style allows the designer to think in terms of pipeline structure, functional unit latency, and whether or not a bypass path is provided. An automated tool can then infer all of the bypass and stall control logic, which is normally one of the most time-consuming parts of a cycle-accurate processor model to design.

To test the usefulness of our approach, we have embodied our specification style in a simple language (PSL) to describe pipelined processors. We have also written an automatic translator that reads descriptions in this language and generates a cycle-accurate simulator for the processor. The key questions to answer are (1) whether the specification style enables simpler, more compact descriptions, and (2) whether the generated simulator is fast enough to be used for meaningful performance evaluation. If the answer to both questions is affirmative, then we have a useful tool for early-stage microarchitectural design exploration.

1.3 Contributions

The work presented in this thesis makes the following contributions to microprocessor specification and simulation research:

- A novel style of specifying cycle-accurate processor models, along with an explanation of our rationale for this approach to microprocessor specification.
- A prototype language, PSL, that embodies this specification style.
- A simulator generator that automatically generates a cycle-accurate simulator from a microprocessor specification written in PSL.
- The specification of a non-trivial microprocessor (MIPS R2000) using PSL.

Chapter 3 of this thesis explains the specification style and the prototype language PSL. Chapter 4 details the simulator generator. And Chapter 5 presents the specification of MIPS R2000 as an example of using PSL.

Chapter 2

Background

2.1 Processor Microarchitecture

A microprocessor can be described at many levels of abstraction. Each level corresponds to a different point of view and allows exploration of a different set of design choices.

From an assembly programmer's point of view, a processor accepts and executes instructions one at a time. At this level (often referred to as the ISA level), the implementation details of the microprocessor are not visible: only the effect of executing instructions.

On the other end of the spectrum, a very low level of abstraction is the layout level, where a processor is described by a collection of rectangles specifying the desired structures on each layer of the fabrication process. At this level, there are enough details to determine the actual die size and the electrical properties of transistors, interconnect, and other devices. These important factors figure strongly in the manufacturing cost. The layout can be analyzed to determine the optimum clock speed supported by this design.

Between these two levels, many intermediate levels of abstraction have been proposed, named, and utilized in real world designs. Below the ISA level is a level often referred to as the microarchitecture level, where the resources and their organization on the processor are described. The next lower level is often referred to as the Register Transfer Language (RTL) level, where the operation of the processor is described by detailed pseudo-code statements using RTLs. (RTL is a simple machine-independent language used to describe the changes to processor registers). An even lower level is known as the gate and logic level, where the exact logic gates used by the processor are specified.

This thesis mainly focuses on the microarchitecture level. Microarchitecture can be defined as follows:

> "Microarchitecture refers to the set of resources and methods used to realize the architecture specification. The term typically includes the way in which these resources are organized as well as the design techniques used in the processor to reach the target cost and performance goals. The microarchitecture essentially forms a specification for the logical implementation." [13]

At this level, the layout information is not available, but the specification is detailed enough to be allow cycle-accurate simulation of the processor. Along with an estimation on the final clock speed, simulation at this level can determine the number of clock-cycles required to execute sample programs and thereby estimate the real-world performance of different microprocessor designs.

The ISA level is too high level to include most processor design decisions. Intuitively, the microarchitecture level is the highest level of abstraction that still exposes the implementation details (such as pipeline depth). Many design decisions made at this level directly affect the cost and performance of the processor. If these design choices lead to undesirable consequences, it is often difficult to undo these negative consequences at lower levels of abstraction. A typical design cycle therefore goes as follows:

1. Specification at the ISA level.
2. Specification at the microarchitecture level.
3. Cycle-accurate simulation in conjunction with intuitive estimation of hardware complexity and performance.
4. Make alternative design choices, then go back to step 2.
5. Specification at lower levels...

Conventional specification methods require verbose descriptions that take a long time to write and a lot of work to modify. Furthermore, efficient simulators usually can not be automatically generated from the specification, making it necessary to spend a significant amount of time to hand-craft a simulator from a specification. Since steps 2 and 3 are often repeated many times during the design cycle, we decided to focus our work on the microarchitecture level and ways to speed up specification and automate simulator generation at that level.

2.2 Related Work

Because of the need to simulate a processor before it is built, processor simulators have been around for decades and are too numerous to cite here. To place our simulator in the context of other simulators, we emphasize the distinction between an instruction set simulator, a cycle-accurate simulator, and a logic simulator (including cycle-based logic simulation).

An instruction set simulator simulates a processor at the instruction set architecture level, completely ignoring how the processor is implemented. Because no processor details are modeled, these simulators provide the fastest simulations (several million instructions per second) and are mainly useful to allow early software development before the processor is completed. A logic simulator simulates the actual hardware implementation (e.g., at the gate level) and is primarily used to validate the correctness of the logic level implementation. However, it runs much more slowly (e.g., a few thousand instructions per second). Cycle-accurate simulators fall somewhere in between instruction set simulators and logic simulators, providing intermediate simulation speed and accuracy. The simulator must model the processor accurately enough to compute the number of cycles needed to execute a program, but does not model the combinational logic. Simulation speed must be fast enough to run non-trivial benchmark code to evaluate processor performance.

Cycle-accurate simulators are traditionally hand-coded for a particular microarchitecture. The process of hand-coding the simulators is both time-consuming and error prone. Therefore, various projects have been launched to develop techniques for automating the generation of microprocessor simulators. The following sections list these related projects in order from high-level to low-level, each with a brief description and a comparison with PSL.

2.2.1 Hawk

Hawk [9] is an executable processor specification language designed for rapid prototyping and simulation. Since it is embedded in the functional language Haskell, it is very flexible. Therefore, specifications can be written for various levels of abstractions (such as ISA level and cycle level).

A processor is specified in Hawk by joining together *components* using *wires*. The Hawk library comes with an extensive selection of pre-built components (such as register files) and different wire types (of different delay properties). Custom components and wires can also be written by describing their behavior using a Haskell expression.

To combine these components into an executable specification, Hawk treats each component as a function that maps clock cycles into values. Essentially, each component generates an infinite list of values (one per clock cycle). Each value of a component is allowed to rely recursively on this component's past values, or on the values of other components. (Haskell's lazy semantics make it possible to execute such a specification of multiple infinite lists of values.) Simulation is then performed by inputing the specification into Haskell, and then querying the values of various components at different clock cycles.

One of the useful domain-specific feature in Hawk is the grouping of values into a *transaction*. A transaction is basically a tuple of values that can obtain additional values as it moves from component to component. It provides a natural way to model instruction contexts in a typical microprocessor: each instruction can carry a *transaction* which receives more and more refined values and states as it moves through the processor.

By allowing add-ons using a functional language, Hawk is far more flexible and dynamic than PSL. However, the flexibility comes at the price of performance. Haskell programs are normally interpreted, which is orders of magnitude slower than compiled execution. Even experimental Haskell compilers generate code that is several times slower than equivalent hand-written C code, unless extensive hints and annotations are used to reduce run-time checking and evaluation. The added effort would nullify the advantages of using Hawk in the first place.

2.2.2 Sleipnir

Sleipnir [6] is an instruction-level simulator generator that exploits domain-specificity by understanding the notion of instructions and instruction formats.

Under Sleipnir, a microprocessor is specified by a list of resources (such as registers and memory locations) and a list of instructions. Each instruction is specified by its recognition bit pattern and a section of raw C code that should be executed when the microprocessor executes the given instruction. The Sleipnir's simulator then works as follows:

```
while(1)
 {
  Identify the current instruction using the list of bit patterns.
  Execute the C code associated with the identified instruction.
 }
```

Since the instructions are simulated one at a time, Sleipnir is well suited for ISA level simulation. It is sometimes possible to approximate cycle-accuracy by inserting raw C code to calculate the effects of delays and stalls, but doing so requires same effort comparable to that of writing a hand-crafted simulator. For complex microprocessors, Sleipnir simulations are effectively restricted to the ISA level.

By using native C code for simulation, and by optimizing the simulator for the ISA level, Sleipnir boasts a higher simulation speed than most of the related simulator generators. According to the statistics cited in [6], the author's instruction-level specification of the integer portion of MIPS R2000 consists of 700 lines, and executes at 5 million instructions per second on a 250 MHz MIPS R10000.

Since Sleipnir, unlike PSL, is mainly designed for ISA level simulation, Sleipnir is well suited for early software development, but is less suited for early exploration of microarchitecture design choices.

2.2.3 UPFAST

UPFAST [10] is an integrated processor specification system that can automatically generate a low level simulator, an assembler, and a disassembler based on a processor specification.

A processor specification using UPFAST basically consists of three components: the *artifacts* (such as registers and memory locations), the pipeline stages, and the instructions. Each instruction is specified by its instruction bit pattern, its assembly instruction syntax, and the RTL statements it should perform when inside different stages. The simulator then works as follows:

```
while(1)
 {
  For each stage S
    { Execute the RTLs associated with the instruction in S }
  Move instructions forward in the pipeline if possible.
  Fetch a new instruction if the first stage is vacant.
 }
```

To facilitate rapid specification of microprocessors, UPFAST provides some domain-specific statements like *stall* (which prevents the instructions in the following stages from proceeding forward) and *freeze* (which blocks the entire pipeline).

UPFAST is the closest related work to PSL in its model of simulation and its approach to processor specification. However, unlike PSL, all forwarding and hazard detections in UPFAST have to be detected and performed explicitly. For example, data forwarding requires each instruction to check the value fields in the neighbor stages and then explicitly fetch them as replacement of the instruction's own outdated values. And hazard detection requires each instruction to check for possible hazards and then issue the *stall* command explicitly.

The low level nature is reflected in the size of processor description. According to [10], the author's specification of the integer portion of MIPS R2000 consists of 5000 lines, and executes at 200K cycles per second on a 200 MHz Pentium Pro.

2.2.4 LISA

LISA [11] is an extensive processor specification system that allows specifications written in LISA to be automatically translated into a simulator, an assembler, and a disassembler. The simulation can be done at the ISA level, the cycle level, and even the phase level (where each cycle is divided into sub-cycles). Unlike PSL, one of LISA's goal is to be general enough for vastly diverse architectures; therefore there are fewer assumptions in LISA, making specifications in LISA more verbose.

A microprocessor specification in LISA consists mainly of the hardware resources and the processor operations. While specifications in LISA can also include additional information (like the assembly language syntax that may be used to automatically generate an assembler), these features are not directly related to processor simulation, and therefore will not be described here.

In LISA, each *processor operation* is described by its syntax (for assemblers and disassemblers), its behavior (a block of RTL-like statements), and an optional *activation* section that describes the timing behavior of the operation. LISA's timing model is based on discrete *control steps*; processor designers can use these control steps to represent instructions, clock cycles, and even clock phases of the processor. Simulation then proceeds as follows:

```
while(1)
 {
  For each activated operation: execute its associated RTLs.
  Activate more operations if possible.
 }
```

LISA also supports domain-specific commands like *stall* (which blocks the activation of following stages) and *flush*.

However, in order to be general enough to accommodate vastly different architectures, LISA does not contain built-in support for automatic forwarding and automatic stalling. For example, mandatory stalling has to be explicitly coded in each operation's activation condition. Therefore, specifications in LISA are necessarily more verbose than those in PSL.

2.2.5 Loosely-Coupled Pipelined Circuits

Marinescu and Rinard [8] propose a style of specification based on the decoupling of pipelined circuits into modules that are loosely connected by FIFO queues. The specification can then be automatically translated into Verilog. Basically a pipelined circuit is specified by a set of global resources, a collection of modules, and the queues that connect the modules.

Each module is specified by a set of rewrite rules using the notation of Term Rewriting Systems [3]. Each rule is a rewrite rule that modifies the current machine state. The semantics are that during each clock cycle, an enabled rule will be arbitrarily chosen and executed until there are no more enabled rules. Then at the end of a clock cycle, the module may insert items into its output queue. Furthermore, if there is no space in the output queue of a module, then the entire module will be disabled with all rewrite rules disabled.

Verilog generation is performed by symbolically evaluating each module's rewrite rules into a single transition expression, and then converting these transition expressions into Verilog. The queues are converted into inter-module buffers that are only readable at the beginning of clock cycles. To optimize the size and speed of the synthesized circuit, the *relaxation* algorithm given in [8] can be used to identify rules that can be executed earlier or concurrently without breaking the semantics. By relaxing the enabling conditions of some rules, the transition expressions can be simplified, reducing the final circuit size.

The emphasis of this approach is to facilitate modular designs. By describing the inter-module buffers as queues, designers can concentrate on building each of the individual modules without worrying about the interaction with other modules. However, unlike PSL, this specification style is not specifically designed for microprocessors and thus lacks some domain-specific features contained in PSL. While automatic stalling is provided when output queues fill up, there is no automatic forwarding and no other automatic stalling. For example, in the sample pipeline processor specification cited in [8], the operand fetch stage is described by code that explicitly forwards data from a later module by explicitly searching for conflicting WRITE instructions in the later module's input queue and then reading the updated value if found. Therefore, specifications written in this style are more verbose than in PSL.

2.2.6 TRAC

Hoe and Arvind [5] propose an operation-centric style of specification using the notation of Term Rewriting Systems [3], and introduce TRAC as a synthesis tool that translates a specification in this style into Verilog directly. Essentially, a machine is specified as the initial machine state and a set of rules that rewrite the current state into the next state.

The *state* of a machine is the combination of all visible values in the model. For example, given a 5-stage pipelined microprocessor like MIPS R2000, the machine state could be modeled by the combined state of the registers, memory locations, and latch values in each stage.

To specify how the machine evolves from cycle to cycle, *transition rules* need to be specified. Basically each rule is a guarded rewrite command that transforms the current state into the next state given suitable preconditions. For example, the fetching of instructions could be modeled by having a rule that transforms (PC,QUEUE,IMEM) into (PC+1,QUEUE:IMEM[PC],IMEM) where PC is the program counter, QUEUE is the list of fetched instructions, and IMEM is the instruction memory. Simulation then proceeds as follows:

```
while(1)
 {
  Apply all non-conflicting enabled rules to the current state.
 }
```

Hoe and Arvind present several theorems in [5] which can be used to optimize the specification. For example, a theorem is given that determines when multiple enabled rules can be applied simultaneously and yet achieve a result that is equivalent to atomically applying each rule in some order. By utilizing these optimization analysis, TRAC is able to go directly from specification to Verilog code without too much penalty. According to the statistics cited in [5], the automatically synthesized MIPS integer core is only 26% larger than the equivalent core generated by hand-written RTL.

TRAC's emphasis is on feasible and efficient synthesis. While PSL specifications also contain enough information for Verilog synthesis, our higher-level semantics require more optimization efforts to generate efficient Verilog. To achieve feasible and efficient synthesis, TRAC descriptions are basically RTL-level, but with a more natural syntax to describe the detailed interaction between hardware components. In contrast with PSL, there are no built-in features to automate stalling, forwarding, or even movement of instructions through the processor. Every feature of the hardware is specified explicitly. This ensures efficient hardware synthesis, but it requires much lower-level specifications than PSL, making it less suitable for early exploration of different design choices.

Chapter 3

PSL: A Processor Specification Language

This chapter explains the specification style and its design objectives. We have embodied this specification style into a prototype language called PSL, which is introduced in Section 3.3. The detailed PSL reference and its grammar can be found in Appendices A and B.

3.1 Design Objectives

Our main objective is to allow rapid specification of processors and automatic generation of processor simulators. This means the specification style must be high-level enough to allow concise and intuitive specifications that are both easy to write and to modify. At the same time, the specification style must be specific and detailed enough to allow automatic generation of cycle-accurate simulators. In order to satisfy both goals simultaneously, we have decided to sacrifice generality by optimizing the specification style only for microprocessors.

Being domain-specific allows the specification language to include built-in support for common microprocessor design features. This enables processor designers to describe a processor with sufficient detail for cycle-accurate simulations (meeting the second goal) while using intuitive and concise language elements (meeting the first goal). In the following section, we present the exact conceptual framework of our specification style.

3.2 Specification Framework

Our entire approach hinges on two assertions:

Assertion 1 (Sequential Execution Semantics) *Processors are designed to provide the same programmer-visible behavior as if individual instructions were executed completely, one-at-a-time, one-after-another.*

Assertion 2 (Producer Knows Best) *The functional unit that produces (or fails to produce) a result is the clearest place in a description to control access to that result.*

The Sequential Execution Semantics assertion means that a description style, by default, should enforce the normal execution semantics that software programmers expect and that processors typically provide. For example, if an instruction depends on the result of the preceding instruction, a simulator generator should determine automatically that a stall or bypass is necessary. In contrast, typical processor descriptions languages require the model to describe explicitly all stall and bypass conditions needed to enforce sequential execution semantics. Describing these conditions is error-prone and time-consuming, and needlessly complicates the description. Note that our specification style can handle small violations of this assertion — for example, our MIPS R2000 model in Chapter 5 accurately models delayed branches — but if a processor completely disregards sequential execution semantics, e.g., a completely non-interlocked VLIW processor, our specification style provides no advantage over existing methods.

The Producer Knows Best assertion means that the description code to determine which instructions have access to which results should be localized to the portion of the description that produces the result, rather than scattered throughout the description among all possible places a result might be needed. Other description styles typically require an explicit check of whether a result is available in any code that accesses that result. In our style, those checks are implicit and can be automatically inferred by a simulator generator. For example, consider the code sequence:

```
addi $t0, $t1, 1
add  $t0, $t0, $t2
sw   $t0, 0($t3)
```

(MIPS assembly convention has the destination register as the first operand, except for store instructions.) In our description style, the add-immediate and add instructions would simply announce whenever their results were ready, and the Sequential Execution Semantics assertion would ensure that the add and store-word instructions get the proper data values, via forwarding or stalling, without any explicit checks for data readiness in the description.

More concretely, our description style requires specifying the functional units of the processor, what each functional unit does, and how instructions flow from one functional unit to the next (or are retired). Instructions carry with them their *context*, i.e., the instruction's view of the values of all machine state. The use of contexts is similar to other formal models of pipelined execution (e.g., [1, 9]) and greatly simplifies the semantics of the description. Functional units will rely on the newest available value for operands, automatically forwarding or stalling as necessary, unless otherwise specified. Results are written into the instruction's context, where they are available to subsequent instructions automatically. Before an instruction retires, its results are committed to a global context, unless the instruction was killed due to exceptions, misprediction, etc.

To support the Sequential Execution Semantics, we introduce the concept of *containers* and *access operators*. Containers are like variables and correspond to all programmer-visible machine state, such as registers, the program counter, condition codes, etc., as well as any additional bookkeeping information needed to process the instruction, such as the instruction register, the opcode, etc. An instruction's context is simply an assignment of values to containers. In addition to their normal values, containers can be assigned the special values UNAVAILABLE, which indicates that the given value is not yet available, and TRANSPARENT, which indicates that the value of the container should be taken from the preceding context. Typically, an instruction's context will initialize with all containers being UNAVAILABLE, and then all containers that the instruction won't modify will be changed to TRANSPARENT. When the instruction's result is available for forwarding to subsequent instructions, the result container will be changed from UNAVAILABLE to the result value. When the instruction is ready to retire, the result value will be written to the global context. (See Figure 3.1.) We call assignments to the current context *announcements*; we call assignments to the global context *commitments*.

When a functional unit accesses the value of a container, the default semantics is to return the most recent non-TRANSPARENT value for that container. The value might be forwarded from a recent instruction or taken from the global context.

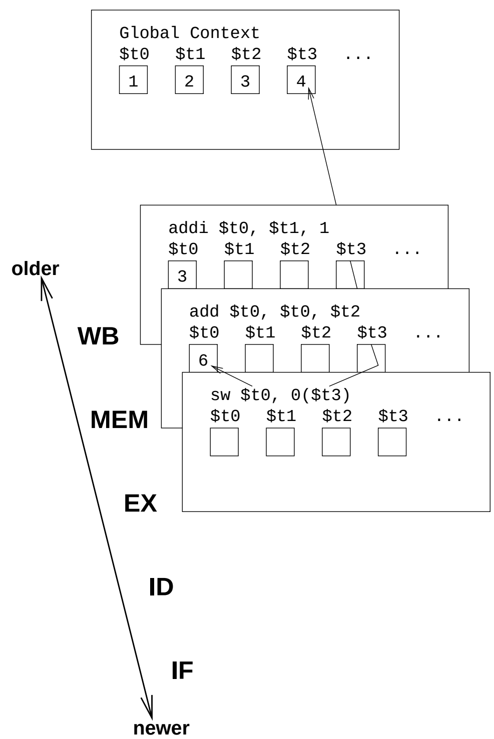

Figure 3.1: The store-word instruction wants to execute and needs to access the values of the registers. It gets the values from the most recent context with a non-TRANSPARENT value, so it gets $t0 from the preceding instruction, but $t3 from the global context, because the add instructions assigned TRANSPARENT to their $t3 containers (because they knew they didn't modify the value of $t3). IF, ID, EX, MEM, and WB refer to MIPS R2000 pipeline stages.

If the most recent value is UNAVAILABLE, the functional unit will stall. This default behavior automatically enforces sequential execution semantics. Each instruction always gets the most current value for its operands. Stalling occurs automatically if a needed result has not yet been announced. Forwarding occurs automatically if a needed result has been announced, but not yet committed. Precise exceptions are easy to implement because uncommitted values disappear when instructions are killed/retired. No description effort is spent implementing any of these effects.

In unusual circumstances, we would like to bypass the sequential execution semantics, so we provide alternative access operators. To date, we have needed only four: Given a container `c`, the default access `c` denotes the most recent non-TRANSPARENT value of the container, starting from the present context; `c'` denotes the most recent non-TRANSPARENT value of the container, starting from the *previous* context; `c$` denotes the most recent non-TRANSPARENT value of the container from the previous context *after all functional units further in the pipeline have completed the current clock cycle* ; and `c#` denotes the value of the container in the global context. For example, the MIPS instruction set architecture specifies delayed branches — the instruction following a branch always executes, regardless of whether the branch is taken or not. Our specification style most naturally produces non-delayed branches in which the Fetch unit is stalled until the branch is resolved. To get delayed branches, we simply add a branch target container and a branch taken flag. The fetch unit fetches instructions normally, unless the branch taken flag signals a branch. The branch taken flag is accessed using the # access operator, so the fetch unit doesn't stall even though the branch decision isn't available yet. (The functional unit that resolves the branch commits the decision to the global context.)

Note that existing description styles make the delayed branch case easy to describe, as it is a natural artifact of the pipelining, but makes normal execution hard to describe. Our description style makes the common case (sequential execution semantics) simple, at the cost making the uncommon cases somewhat harder.

3.3 Language Description

To demonstrate these concepts, we have developed a prototype language, PSL, that embodies this specification style. This section contains a brief description of the language. The detailed language reference of PSL is available in Appendix A, and its grammar is listed in Appendix B.

Before we begin describing the language, it is useful to introduce a sample specification using PSL. Below is a specification of a simple, fictitious processor:

```
container Mem[0..0xFFFF], PC, Op;

enum "raise":=1, "lower":=2;

constructor Fetch
 {
  true: { Op <- Mem#[PC'] ; PC:=PC'+1; goto ALU; }
 }

stage ALU
 {
  Op == "raise" : { syscall(7,0,0); }
  Op == "lower" : { syscall(3,0,0); }
  true          : { retire;         }
 }
```

The above specification describes a simple microprocessor with two stages: an instruction fetch stage and an ALU stage. The main memory holds exactly 65536 words, and there are two processor state variables (`PC` and `Op`). During each clock cycle, an instruction is fetched from the memory and the `PC` is incremented by 1. The previously fetched instruction then moves into the ALU stage where it will make one of two possible syscalls (based on the instruction opcode) and then retire.

In general, a processor specification in PSL includes a list of containers and a list of stages. Each stage has a name and a list of guarded commands. Furthermore, one and exactly one of the stage must be declared as a constructor stage. In the above example, there are three containers (Mem, PC, and Op) and two stages (constructor stage Fetch and the non-constructor stage ALU).

The simulation begins with a global context (where all containers are initialized to zero or some other default values) and a list of stages which are all vacant (meaning no instruction is in any of the stages). During each clock cycle, every occupied stage will execute all enabled commands using the given instruction's context. For example, if an enabled command refers to the container `PC`, then by default the value it uses is the value from the instruction's private context.

Legal statements include *announcement* statements using the `<-` operator (which modify the values in the current instruction context), *commitment* statements using the `:=` operator (which modify the values in the global context), the `goto` statement (which indicates where the instruction should go in the next cycle), and the `retire` statement (which indicates the instruction should exit the processor after this cycle). It is illegal for an instruction to execute both a `goto` statement and a `retire` statement in the same cycle.

The semantics of the multiple announcements and commitments is that these context modifications happen atomically in parallel; it is illegal to make more than one announcement or commitment to the same container during any given cycle. If an instruction encounters unavailable values when evaluating one or more of the guards, or if an enabled statement requires a value that is unavailable, then the instruction is deemed to have *failed* in the current cycle. That means none of its context modifications will happen, and it stays in its current stage regardless of any `goto` or `retire` statements. On the other hand, if an instruction does not encounter any unavailable values during a cycle, but its destination stage is occupied in the next cycle, then this instruction is deemed to have *stalled* in the current cycle. That means all of its announcements and commitments do happen, but the instruction stays in the current stage.

At the end of a clock cycle, all instructions that have an enabled `retire` command will exit the processor, while all instructions that have an enabled `goto` command will attempt to move into their destination stages. The movement of instructions happen atomically with the instructions furthest in the pipeline having the top priority. This means given stages 1, 2, 3, 4, and 5 in a pipeline, if the instruction in stage 5 wishes to retire, then the instruction in stage 4 can move into stage 5 within the same cycle as the retirement. Furthermore, whenever the constructor stage would be vacant for the next cycle, a new instruction will be created and placed into the constructor stage just before the beginning of the next cycle. The newly created instruction has a per-instruction context where all containers are initialized to the special value UNAVAILABLE.

All four access operators described in Section 3.2 are available in PSL. The syntax of the expressions are modeled after the C language. For the detailed language reference, please refer to Appendix A.

Chapter 4

Implementation of a Simulator-Generator

4.1 Design Goals

To test our specification approach and its embodiment in PSL, we have developed a simple proof-of-concept tool that automatically translates specifications into cycle-accurate simulators. There are two objectives for the simulator-generator: the generation of the simulator must be fully automatic, and the generated simulator should be as fast as possible.

The first goal is achievable because the PSL language has been defined with this goal in mind; that means barring a few extreme cases (such as division by zero), the language has no implementation-dependent nor any undefined behavior.

The second goal is achieved by employing aggressive compile-time analysis to eliminate redundant computation, and by optimizing the simulation engine for the most common cases. More explanation of the optimization techniques developed for this project can be found in Section 4.4. In the following sections, we present the prototype implementation of the simulator-generator.

4.2 Data Structures and Algorithms

There are 3 main data structures used in the generated simulator: the *global* array (which stores the global context of all containers), the *context* singly-linked list (which has one node for each live instruction currently in the simulator), and the *future* array (which stores the atomic parallel commitments during a cycle).

The *global* array is simply an array that stores the global context of all containers during the simulation. It has the same number of elements as containers, and all elements are initialized to 0 before the simulation begins. The simulator can then optionally load an image file that contains initial values for the global containers.

In our simulator engine, the *context* singly-linked list is used to store per-instruction context. It is basically a linked list of nodes, one node for each live instruction. As instructions are fetched in the constructor stage, a new empty node is allocated for the new instruction and inserted at the head of the list. As instructions retire, their nodes are removed from the list. So in effect, the nodes of the list are sorted in program order, with the head being the youngest instruction and the tail being the oldest.

Conceptually, the per-instruction context is just an array that stores the values of all containers in the private context. However, since most instructions only operate on a small subset of the list of containers, it is inefficient to store the per-instruction context as a large array. Each node contains the full list of announcements in the context of its associated instruction. An empty node means this instruction has not made any announcements; therefore, all containers are regarded as UNAVAILABLE. Context modification is done by appending or removing an announcement from the list. As an optimization, each entry in the list can represent the value of multiple adjacent containers if they share the same value.

During each cycle, an instruction that attempts to make announcements and commitments may *fail* when it encounters UNAVAILABLE values. When that happens, all the context modifications it makes in that cycle have to be canceled. That is accomplished by storing all per-instruction context modifications into the instruction node's temporary array, and all global context modifications into the *future* array as each instruction executes. If the instruction can successfully execute all of its enabled commands, then the entries in the *future* array will be written to the actual global context, and the entries in the instruction node's temporary array will be written into the per-instruction context. Otherwise, the *future* array and the per-node temporary array will be purged, canceling all announcements and commitments by this instruction.

4.3 Simulation Engine

In order to be as portable as possible, the automatically generated simulation engine is a C source file that uses only standard C functions. The simulator source code then simply contains three major data structures (as explained in the previous section), an initialization function (which can read image files as initial values for global containers), and a main simulation function (which contains the simulation code for the entire processor). The pseudo-code for the main simulation function is shown in Figure 4.1.

One thing worth mentioning is how a reference to a container is evaluated at

```
while(true)
 {
  Erase all entries from the "future" array.
  For each stage that contains a live instruction
   {
    Remember the current size of the "future" array.
    Erase all entries from the instruction's "temporary" array.
    Evaluate all guards.
    If an unavailable value is encountered
     {
      Clean up. Skip the rest of this stage.
     }
    Else
     {
      Execute all enabled statements.
      If an unavailable value is encountered
       {
        Clean up. Skip the rest of this stage.
       }
     }
   }
  For each entry in the "future" array
   {
    Commit the changes into the "global" array.
   }
  For each entry in each instruction's "temporary" array
   {
    Make the changes in the per-instruction context.
   }
  For each live instruction from oldest to youngest
   {
    If the instruction wishes to retire,
       remove its node from the "context" linked list.
    Otherwise,
       move into the destination stage if vacant.
   }
  If the constructor stage becomes vacant
   {
    Create a new instruction with an empty "context" node.
    Insert the new node into the "context" linked list.
    Insert the new instruction into the constructor stage.
   }
 }
```

Figure 4.1: Pseudo-code for the main simulation function

run time. Recall from Section 3.3 that there are four access operators in PSL, each corresponding to a different sequence of context search. The pseudo-code for the `C` container reference is shown in Figure 4.2.

```
For(node=ThisContext; node!=NULL; node=node->next /* earlier instruction */)
 {
  For each entry (Container E, Value V) in node->AnnouncementTable
   {
    If C == E
     {
      If V is an integer, then the value of the entire expression is V.
      Otherwise, if V is UNAVAILABLE, then the entire expression is UNAVAILABLE.
     }
   }
 }
If a value still hasn't been found, then the GLOBAL context is queried.
```

Figure 4.2: Pseudo-code for a reference to container C

The second access operator, `C'`, differs in that it searches for the most recent non-TRANSPARENT value starting from the *previous* context instead of the *current* context. The pseudo-code for the `C'` container reference is shown in Figure 4.3.

The third access operator, `C$`, differs in that it waits for all later stages to be completed and then searches for values in both the per-instruction context and in the *future* and temporary tables. The pseudo-code for the `C$` container reference is shown in Figure 4.4.

The fourth and final access operator, `C#`, does not depend on any instruction context. Instead, it bypasses per-instruction private context and directly queries the *global* context for the value. It is quite straight-forward; therefore, no pseudo-code is listed here.

```
For(node=ThisContext->next; node!=NULL; node=node->next /* earlier instruction */)
 {
  For each entry (Container E, Value V) in node->AnnouncementTable
   {
    If C == E
     {
      If V is an integer, then the value of the entire expression is V.
      Otherwise, if V is UNAVAILABLE, then the entire expression is UNAVAILABLE.
     }
   }
 }
If a value still hasn't been found, then the GLOBAL context is queried.
```

Figure 4.3: Pseudo-code for a reference to container C'

```
For(node=ThisContext->next; node!=NULL; node=node->next /* earlier instruction */)
 {
  For each entry (Container E, Value V)
      in node->AnnouncementTable, node->TemporaryArray, and the future array
   {
    If C == E
     {
      If V is an integer, then the value of the entire expression is V.
      Otherwise, if V is UNAVAILABLE, then the entire expression is UNAVAILABLE.
     }
   }
 }
If a value still hasn't been found, then the GLOBAL context is queried.
/* Since the stages are evaluated from the end of the pipeline to the front,
   all later stages would have completed. Thus, querying the Temporary
   and Future arrays provides the desired most recent value. */
```

Figure 4.4: Pseudo-code for a reference to container $C\$$

4.4 Optimization Trade-offs

Due to time constraints, we have not used any sophisticated techniques for accelerating processor simulation, such as predecoding instructions, cross compilation, or strength reduction. The simulation engine currently uses only simple data structures (like arrays and linked lists). The few optimization techniques we do use are listed below.

The first optimization technique is basic *constant folding* and *common subexpression elimination*. These optimizations are performed by the parser as it reads in the specification file. More information about performing these simple optimizations can be found in [2].

The main additional optimization technique used in our prototype simulator-generator is the elimination of redundant computation by memoization at compile time. Since all commitments and announcements happen in parallel at the end of a clock cycle, the values of expressions do not change during a cycle (with the exception of expressions that use the `$` access operator). Therefore, if the same expression appears more than once in a specification, the simulator should evaluate the expression only once per cycle, memoize the value into a variable pre-determined at compile time, and query the variable each time the expression needs to be evaluated again. To facilitate this, our prototype simulator-generator has a tunable parameter that determines how many computed values can be memoized during a clock cycle. The trade-off here is that larger values of N allows more memoization and reduces redundant computation, but if the value of N exceeds the number of general registers available on the CPU running the simulation, then these excess values would spill into memory and slows down the simulation.

In practice, a typical value of N that is roughly equal to the number of general registers seems to work well. Since the automatic generation of simulator only takes a fraction of a second, and since the simulator-generator can report the number of required computational steps, it is recommended that multiple values of N be attempted to find the value that results in the fastest simulation.

4.5 Future Optimization Possibilities

As noted in the previous section, the prototype simulator-generator currently does not use any sophisticated techniques for accelerating processor simulation. Many future optimization approaches are possible.

We believe the first future optimization should be to replace the linked-list traversal with a combination of compile-time data flow analysis (to locate the most recent value of a container at compile time rather than at run time) and, where that's not possible, represent the context by a faster data structure (such as a hash table or a balanced search tree). The specification contains both data flow and control flow information, which should enable some amount of compile time analysis of forwarding and stalling.

To simplify data flow analysis, we should restrict the language slightly. Right now, the language allows an instruction to declare a container as a Don't-Care (TRANSPARENT) and then take away the Don't-Care flag later. Furthermore, an instruction can announce a valid value for a container, and then take away the value by announcing TRANSPARENT, UNAVAILABLE, or even a different integer value. This flexibility means an instruction can potentially modify any container at any clock cycle. However, this not only makes the simulation harder to optimize, but it also hampers future efforts at formal verification. We have found no need for this flexibility and believe the language should make all announcements and commitments irrevocable.

Looking at other related projects provides insight for additional optimization possibilities. For example, the Sleipnir [6] simulator uses a decode cache that holds decoded instruction information for later cycles. A similar approach could be adapted for our simulator, in that a retired instruction context could be retained for its next incarnation. Actual experiments are needed to determine whether this provides a positive impact on simulation speed or not.

Another interesting optimization technique can be found in the LISA system. The idea is that quite frequently it is desirable to run a simulation of a small, fixed program for many clock cycles. This is true for much DSP code, and it is also sometimes true for microprocessor benchmarks. By taking into consideration a specific memory image, the simulator-generator can omit unused instructions, hardware resources, and bypass paths. Furthermore, control and data flow analysis can be performed on both the processor specification and the memory image to derive more container values and instruction stalling information ahead of time. By making this trade-off, the authors of LISA report the speed is sometimes improved by more than one order of magnitude.

However, given our higher-level semantics, it is not clear if we can match the performance of simulators generated from more explicit specifications. For example, our simulator performs run-time hazard detection at every container reference, even though some hazards are never possible. On the other hand, it may be possible to eliminate some of these unnecessary computations through more careful analysis of the model. Furthermore, our simulator generator might be able to exploit domain-specificity to generate carefully optimized code for specific aspects of the simulator. Additional optimizations is an area for further research.

Chapter 5

Example: MIPS R2000

We have implemented the integer core of the MIPS R2000 [7] microprocessor, including all user-mode instructions except coprocessor support, floating point instructions, and non-aligned loads/stores. The model we present here does not implement exceptions, although they could be added easily.

MIPS R2000 is a classic, 5-stage RISC pipelined processor. Extensive documentation is available, and the microarchitecture has been thoroughly studied, so it makes an excellent benchmark example. Although such a design is no longer leading edge, it is a very representative processor of its generation. Furthermore, such a design is typical of the application-specific cores used today, for which there are currently numerous design starts and for which early-stage microarchitectural exploration would be useful. We intend to extend our language to handle more aggressive microarchitectures (superscalar, VLIW, etc.) in the future.

5.1 Specification of MIPS R2000

Our entire processor specification is 290 lines long, of which 150 lines are devoted to the instruction decoder. The description starts by declaring containers for the main memory (Mem), 32 general-purpose registers (Reg), multiplication and division registers (Hi and Lo), program counter (PC), and a handful of instruction fields and temporary results. The R2000 has five pipeline stages, so the description also declares five pipeline stages. Each of these 5 stages will be explained in the following subsections.

5.1.1 Fetch Stage

The first pipeline stage is the instruction fetch stage:

```
constructor FetchStage
 {
  PF# == 0: { INS <- Mem#[ (PC' + 4) >>L 2 ]; PC <- PC'+4 ;         }
  PF# == 1: { INS <- Mem#[ (PN#    ) >>L 2 ]; PC <- PN#   ; PF:=0; }
  true    : { Reg[0] <- tr; goto DecodeStage; }
 }
```

The first two lines basically attempt to read a new instruction from memory (the Mem array) into the instruction register (INS). Intuitively, the instruction is fetched from either the next instruction in memory (PC + 4), or from the branch target address (PN) if the branch taken flag (PF) is set. However, in order to get the correct semantics, there are a few subtleties that need to be observed.

The first issue involves the modeling of the system memory. In this specification, we have decided to ban non-aligned memory access. Given this restriction, it is natural to implement the memory as a word-addressed rather than byte-addressed array; that is, `Mem[0]` refers to the first four bytes and `Mem[1]` the next four bytes, etc. However, the program counter (`PC`) and the branch target register (`PN`) both store byte addresses. To compensate for the difference, the offsets are divided by 4 before being used as `Mem` array indices.

Another memory simplification is that our specification does not accurately model self-modifying code. Therefore, instructions can be fetched using the `#` operator without stalling even when a memory write is in the pipeline.

There is only one branch taken flag and one branch target register; they are global variables used by the branch instructions to override the default "program order semantics". That means all references to `PF` and `PN` must use the `#` operator, since the other 3 access operators are all bound by the program order semantics. In addition, since all references to `PF` and `PN` are done via the global context, all modifications should be done using *commitment* statements rather than *announcement*.

On the other hand, the program counter can be considered a per-instruction value in that each instruction is fetched from a particular memory address. In the branch-not-taken case, the new instruction announces its `PC` value by adding 4 to the previous instruction's `PC` value. Here we see the need for a second access operator (the `'` operator) which returns the most updated value of a container in the context of the previous instruction. The statement `PC<-PC'+4` takes the previous instruction's `PC` value, adds 4 to it, then immediately announces it as the new `PC` value so that the instruction fetch at the next cycle does not stall.

The third line of the `FetchStage` stage introduces the usage of the keyword `tr` (which represents the special value TRANSPARENT). In MIPS R2000, register 0 is always 0 regardless of the instruction that reads it. This fact is known before the instruction is even decoded; therefore, we can announce it in the fetch stage to ensure that all references to register 0 go directly to the global context without stalling.

Finally, the `goto` statement is used to indicate the new instruction should attempt to go into the decode stage in the next clock cycle. Recall our language provides automatic stalling when needed. So here we do not need to check for stalling; if the current instruction in the decode stage is stalled, this newly fetched instruction won't be able to advance and will stall automatically.

5.1.2 Decode Stage

The most difficult and tedious part of the processor specification was the instruction decoder. Decoding instructions involves nothing more than extracting bit fields and pattern matching, but the decode stage encompasses over half of the entire description. Our research is not directed at instruction decoding, so this part of the model is simply a necessary evil. Fortunately, other researchers have developed high-level techniques for specifying instruction decoders [12], so a real, high-level processor specification language should combine that work with ours. The decode stage description consists of numerous cases such as this one:

```
// ADD[U],SUB[U],AND,OR,XOR,NOR,SLT[U]
(INS & 0xFC000030) == 0b100000:
 {
  Mem[..] <- tr;
  Reg[..INS[15..11]-1] <- Reg[INS[15..11]+1..] <- Hi <- Lo <- tr;
  T2 <- 0;
  V1 <- Reg$[INS[25..21]]; Op <- INS[ 5.. 0];
```

```
 V2 <- Reg$[INS[20..16]]; T  <- INS[15..11];
}
```

The guard checks the opcode of the instruction register `INS` to see if it is one of the specified instructions. In this case, all these three-register-instructions (ADD, ADDU, etc.) share a common instruction format and can be detected by a simple bitwise test. Furthermore, they share the same basic behavior:

1. These instructions do not modify the memory nor the multiplication and division registers (`Hi` and `Lo`). Therefore, we can immediately announce the special TRANSPARENT value for `Hi`, `Lo`, and the entire `Mem` array. Furthermore, we clear the `T2` flag to indicate the `Hi` and `Lo` registers will not be modified.

2. These instructions only modify one of the 32 general purpose registers. In particular, the target register is always the one indicated by bits 15 to 11 in the instruction. Therefore, we can store the target register's identity into an internal variable (`T`) and announce the special TRANSPARENT value for the other 31 registers. Here the instruction register (`INS`) is referenced using the third access operator, the default operator, to query the current instruction context. Since `INS`'s value is announced in the fetch stage already, reading it here does not cause stalling.

3. Since these instructions all get their operands from the same register locations, we can fetch the operands in this common command block. Here, MIPS R2000 architecture requires that if an instruction in the ALU stage (the stage after the decode stage) produces a new value for an operand, that new value must be forwarded into the decode stage in the same cycle. This is accomplished in our specification by using the fourth access operator (the `$` operator) which waits until all other stages are completed and then forwards the latest value from the previous instruction's context.

4. The difference among these instructions is the ALU operation they represent. The choice of a specific ALU operation depends on bits 5 through 0 of the instruction register. Therefore, these bits need to be extracted then stored in an internal variable (`Op`).

The decode stage contains similar guarded command blocks for other instructions. It also contains the following `goto` statement

```
true: { goto ALUstage; }
```

directing all instructions to proceed to the ALU stage.

5.1.3 ALU Stage

The ALU stage is the second longest stage, simply because there are many possible ALU operations. Here is the entire ALU stage description:

```
stage ALUStage
 {
  Op=="ALU-SLL"   : { Reg[T ? T : 32] <-  (V1 <<  V2) ; }
  Op=="ALU-SRA"   : { Reg[T ? T : 32] <-  (V1 >>A V2) ; }
  Op=="ALU-SRL"   : { Reg[T ? T : 32] <-  (V1 >>L V2) ; }
  Op=="ALU-MFHI"  : { Reg[T ? T : 32] <-  (Hi'      ) ; }
  Op=="ALU-MFLO"  : { Reg[T ? T : 32] <-  (Lo'      ) ; }
  Op=="ALU-MTHI"  : { HiTemp          <-  (V1       ) ; }
  Op=="ALU-MTLO"  : { LoTemp          <-  (V1       ) ; }
  Op=="ALU-ADD"   : { Reg[T ? T : 32] <-  (V1 +   V2) ; }
  Op=="ALU-ADDU"  : { Reg[T ? T : 32] <-  (V1 +   V2) ; }
  Op=="ALU-SUB"   : { Reg[T ? T : 32] <-  (V1 -   V2) ; }
```

```
  Op=="ALU-SUBU"  : { Reg[T ? T : 32] <-  (V1 -   V2) ; }
  Op=="ALU-AND"   : { Reg[T ? T : 32] <-  (V1 &   V2) ; }
  Op=="ALU-OR"    : { Reg[T ? T : 32] <-  (V1 |   V2) ; }
  Op=="ALU-XOR"   : { Reg[T ? T : 32] <-  (V1 ^   V2) ; }
  Op=="ALU-NOR"   : { Reg[T ? T : 32] <- ~(V1 |   V2) ; }
  Op=="ALU-SLT"   : { Reg[T ? T : 32] <-  (V1 <S  V2) ; }
  Op=="ALU-SLTU"  : { Reg[T ? T : 32] <-  (V1 <U  V2) ; }
  Op=="ALU-LUI"   : { Reg[T ? T : 32] <-  (V2 <<  16) ; }
  Op=="LINK"      : { Reg[T ? T : 32] <-  (PC +    8) ; }

  Op=="ALU-MULT"  : { HiTemp<-(V1 *SH V2); LoTemp<-(V1 *SL V2); }
  Op=="ALU-MULTU" : { HiTemp<-(V1 *UH V2); LoTemp<-(V1 *UL V2); }
  Op=="ALU-DIV"   : { HiTemp<-(V1 %S  V2); LoTemp<-(V1 /S  V2); }
  Op=="ALU-DIVU"  : { HiTemp<-(V1 %U  V2); LoTemp<-(V1 /U  V2); }

  (Op & 512)==512 : { Offset <- (V1 + V2); } // MEMORY

  true: { goto MemoryStage; }
 }
```

The ALU stage description simply computes the desired operation and specifies that instructions proceed to the memory stage next. A few points are worth mentioning:

1. Almost all ALU instructions use `V1` and `V2` as operands. All forwarding and stalling is implicit in the language and need not be spelled out.

2. MIPS R2000 ignores all writes to register 0. In our specification, we have declared a Don't Care register (`Reg[32]`) that is not read by anyone. All writes to

register 0 are then simply redirected to register 32.

3. To allow concise specification of the ALU stage, PSL provides a huge list of operators that perform most of the common ALU operations. For example, there are separate operators for signed and unsigned division (`/S` and `/U`, respectively). If the specification of a particular microprocessor requires an arithmetic operation that can not be done by combining existing PSL operators, a new operator can be easily added to PSL.

4. There are fewer cases in the ALU stage than in the decode stage because many similar instructions (such as the register and immediate forms) have been folded into a common form by the decode stage. Since exceptions are not implemented, the with- and without-exception forms are folded as well. The branch instructions are taken care of entirely in the fetch and decode stages, and thus are not handled in the ALU stage.

5.1.4 Memory Stage

The memory stage accesses memory if necessary:

```
stage MemoryStage
 {
  // The next stage to go to is the WriteBack stage.
  true: { goto WriteBackStage; }

  // Command block for each different read instructions.
  Op=="MEM-LW"                  :{ Reg[T?T:32]<- Mem[Offset >>L 2]         ; }
  Op=="MEM-LH"  && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2][[31..16]]; }
  Op=="MEM-LH"  && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2][[15.. 0]]; }
  Op=="MEM-LHU" && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2] [31..16] ; }
  Op=="MEM-LHU" && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2] [15.. 0] ; }
  Op=="MEM-LB"  && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2][[31..24]]; }
  Op=="MEM-LB"  && (Offset&3)==1:{ Reg[T?T:32]<- Mem[Offset >>L 2][[23..16]]; }
```

```
Op=="MEM-LB"  && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2][[15.. 8]]; }
Op=="MEM-LB"  && (Offset&3)==3:{ Reg[T?T:32]<- Mem[Offset >>L 2][[ 7.. 0]]; }
Op=="MEM-LBU" && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2] [31..24] ; }
Op=="MEM-LBU" && (Offset&3)==1:{ Reg[T?T:32]<- Mem[Offset >>L 2] [23..16] ; }
Op=="MEM-LBU" && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2] [15.. 8] ; }
Op=="MEM-LBU" && (Offset&3)==3:{ Reg[T?T:32]<- Mem[Offset >>L 2] [ 7.. 0] ; }

// Common commands shared by all memory writes.
// Memory writes can be recognized by (Op & 520)==520.
(Op & 520)==520:{ Mem[..((Offset >>L 2)-1)]<-Mem[((Offset >>L 2)+1)..]<-tr;}

// Command block for each different write instructions.
Op=="MEM-SW": { Mem[Offset >>L 2] <- V3; }

Op=="MEM-SB" && (Offset&3)==0:
  { Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0x00FFFFFF) | ((V3 & 0xFF)<<24); }

Op=="MEM-SB" && (Offset&3)==1:
  { Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFF00FFFF) | ((V3 & 0xFF)<<16); }

Op=="MEM-SB" && (Offset&3)==2:
  { Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFFFF00FF) | ((V3 & 0xFF)<<8); }

Op=="MEM-SB" && (Offset&3)==3:
  { Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFFFFFF00) | (V3 & 0xFF); }

Op=="MEM-SH" && (Offset&3)==0:
  { Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0x0000FFFF) | (V3 << 16); }

Op=="MEM-SH" && (Offset&3)==2:
  { Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFFFF0000) | (V3 & 0xFFFF); }
}
```

Even though we do not model non-aligned accesses, we do model MIPS R2000's ability to load/store half words and bytes. We model the load instructions by reading the entire word and extracting the desired bits, sign-extending them if necessary using the `[[ ]]` operator described in Section A.5.8. The store instructions are modeled by reading the entire word, modifying parts of it, then writing the entire word back. In addition, the store instructions must remember to declare TRANSPARENT for the untouched words in order to prevent unnecessary stalling.

5.1.5 WriteBack Stage

The last stage is the WriteBack stage. It is where all register modifications are actually committed.

```
stage WriteBackStage
 {
  true: { retire; }

  // If T is not zero, that means a new value for
  // register T has been announced.
  // Therefore, it needs to be committed.
  T: { Reg[T] := Reg[T]; }

  // If T2 is not zero, that means new Hi and Lo values
  // are available in HiTemp and LoTemp.
  // They must be announced and committed to Hi and Lo.
  T2: { Hi:=HiTemp; Hi<-HiTemp; Lo:=LoTemp; Lo<-LoTemp; }

  // If it's a memory write instruction, the announced
  // memory write must be committed.
```

```
  // In this specification, commitments to memory are delayed until
  // the final stage, so that flushing an instruction automatically
  // undoes all its effects on the global containers.
  (Op & 520)==520: { Mem[Offset >>L 2] := Mem[Offset >>L 2]; }

  Op=="SYSCALL": { Reg[2] := syscall(V1,V2,V3); }
 }
```

Note the use of commitment using the `:=` operator so that committed values affect future instructions, even after the current instruction is retired. In addition, since we have not implemented exceptions in our model, we need a place to implement the `syscall` interface. We have chosen to place it in the WriteBack stage arbitrarily. A custom handler can be used along with the simulator to allow statistics gathering, system I/O emulation, etc.

5.1.6 Exceptions

The model we present here does not implement exceptions, although they could be added easily. One approach is to use a new container (`FlushFlag`) to indicate whether the instructions in the pipeline should flush or not. Every stage would check to see if `FlushFlag#` is set or not, and if so, executes a `retire` command immediately. (The guard that checks the `FlushFlag` must use the `#` access operator because it is the only operator that can bypass the "program order semantics"). With the above addition to the model, an exception can be raised by setting the flag. At the next cycle, all instructions in the pipeline will flush; since none of the instructions make it to the WriteBack stage, their announcements will be discarded, satisfying the requirements for precise exceptions.

5.2 Results

The above is the entire description of this processor. Except for the decode stage, we can see how our description style is vastly simpler than previous processor descriptions. The remaining question is whether we can generate efficient simulators from these descriptions.

We selected the Dhrystone synthetic benchmark, Version 2.1 [14] to provide a non-trivial, integer workload to test the simulator. Dhrystone is too small for general-purpose processors, but is still widely used to benchmark embedded processors. We chose the Dhrystone benchmark because we ran into compiler problems with larger benchmarks: namely, we were unable to get `gcc` to generate only MIPS R2000 code. With a small enough benchmark like Dhrystone, we were able to manually substitute the unsupported instructions with equivalent R2000 instructions.

Because of our limited system call interface, we omitted the initial log-file access of the Dhrystone benchmark, but included the rest of the benchmark. We used the `gcc` C compiler version 2.95.2, with options "`-G 104857600 -mgpopt -mfp32 -mgp32 -s -static`" to compile the benchmark into MIPS object code The compiler generated some instructions not supported on the R2000 — being a more modern compiler — so we had to substitute equivalent R2000 instructions manually. (The steps we took to compile the Dhrystone binary on an SGI machine and then simulate it with our MIPS R2000 simulator are shown in Figure 5.1).

Automatic generation of the simulator took less than a second. The resulting simulator was compiled using `gcc` version egcs-2.91.66 with the `-O3` optimization option. We ran our experiments on a 733Mhz Pentium III with 256KB cache and 128MB memory, running Linux version 2.2.16-22.

We consistently obtained over 240,000 simulated cycles/second. Direct com-

```
1) LINUX# make
   --> This builds the simulator-generator "sim"

2) LINUX# ./sim < mips.psl > mips.c
   LINUX# cat main.inc >> mips.c
   LINUX# gcc -O3 -o mips mips.c
   --> This builds the MIPS R2000 simulator "mips"

3) SGI# gcc -G 104857600 -mgpopt -mfp32 -mgp32 -s -static -S -o dhry.s dhry.c
   --> This builds dhrystone.s

4) SGI# gas -o dhry.o dhry.s
   --> This builds dhrystone.o
   --> You may have to edit dhry.s manually beforehand to...
   A) Replace non-R2000 instructions with R2000 equiv. (Such as BEQL, DADDU)
   B) Replace non-gas pseudoinstructions. (Such as ".popsection")
   #
   # For this version of the tweaked "dhry.c", there are 4 things to do:
   # A) Replace the only BEQL with BEQ
   # B) The next line has the only "BREAK". Replace it with NOP.
   # C) There are many ".popsection"  ".section bss" pairs. Remove them.
   # D) The only remaining ".popsection" will be at the end. Remove it.

5) SGI# ld -o dhrystone.image dhrystone.o && strip dhrystone.image
   --> This generates the image file.
   --> It warns "missing _start". It's okay, since we moved "main()" to top.

6) SGI# elfdump -R dhrystone.image
   --> Determine the initial "GP value" needed by this image.

7) SGI# elfdump -h dhrystone.image
   --> Determine how many segments the image file has, and how long
   --> each segment is, and where each segment should be loaded to.
   # For example, it may say (paraphrased):
   # .code    ShouldBeAt 0x400018 , IsAtFileOffset 0x500 , Length 0x30
   # .rodata  ShouldBeAt 0x410000 , IsAtFileOffset 0x530 , Length 0x50
   # .sdata   ShouldBeAt 0x410100 , IsAtFileOffset 0x580 , Length 0x10

8) LINUX# ./mips 2000 100 \ #Run 2000 cycles without DEBUG, then 100 with DEBUG
         s 3 29 0x500000  \ #Make R29 big enough to give a large enough stack
           3 28 0x414950  \ #Set R28 to the "GP value" shown in "elfdump -R"
     f dhrystone.image 0x400018 0x500 0x30 \ #Load EACH segment in "elfdump -h"
     f dhrystone.image 0x410000 0x530 0x50 \
     f dhrystone.image 0x410100 0x580 0x10
```

Figure 5.1: Steps needed to simulate Dhrystone

parison with other published results is difficult because of differences in modeling and host machine speed. As a very rough comparison, Önder and Gupta [10] reported 200K cycles/second for their UPFAST system simulating a cycle-accurate, 5-stage, complete MIPS on a 200Mhz Pentium Pro. The description was 5782 lines long. Jeremiassen [6] reported 5.1M instructions/second for his Sleipnir system, simulating an *instruction-level*, integer-only MIPS core on a 250Mhz MIPS R10000. The description was 700 lines long. Apparently, our performance is within an order of magnitude of more carefully tuned simulators, and our descriptions are much smaller and simpler.

Chapter 6

Conclusions and Future Work

We have introduced a novel approach for specifying processor microarchitectures. The approach centers on two key ideas: enforcing sequential execution semantics automatically and localizing access to results in the code that generates the values. We have demonstrated the potential of our approach by implementing a simple description language with an automatic simulator generator, and then creating a MIPS R2000 model in our language. Our description is vastly simpler and shorter than competing methods, and our simulation performance is within an order of magnitude of high-performance, purpose-built simulators. We believe our approach will be very useful for early-stage microarchitecture design exploration.

The work presented in this thesis makes the following contributions to microprocessor specification and simulation research:

- A novel style of specifying cycle-accurate processor models, along with an explanation of our rationale for this approach to microprocessor specification.
- A prototype language, PSL, that embodies this specification style.

- A simulator generator that automatically generates a cycle-accurate simulator from a microprocessor specification written in PSL.

- The specification of a non-trivial microprocessor (MIPS R2000) using PSL.

The most obvious future work is to improve the performance of our generated simulators. There is considerable literature on techniques to improve simulation performance, and we should be able to harness some of it. Ideas and future directions on optimization are explored in Section 4.5.

In addition, this thesis is not directed at instruction decoding. As a result, our tool does not facilitate concise specification of the instruction decoding. As noted in Chapter 5, the decode stage occupies more than half of our MIPS R2000 specification. We believe harnessing existing research on specifying instruction decoders [12] should greatly improve the ease-of-use of our system.

More generally, we believe our ideas extend easily to superscalar processors and are working on formalizing the concepts and implementing a tool chain. The main difference is more dynamic constructs for directing instructions to functional units.

It may eventually be possible to synthesize efficient implementations from our descriptions. This goal seems rather distant, but research on automatic pipelining is promising [4, 8, 5]. Because the logic required to enforce sequential execution semantics is currently extremely challenging to design correctly, being able to synthesize from a small, obviously correct description is a very attractive long-term target.

Bibliography

[1] Mark Aagaard and Miriam Leeser. Reasoning about pipelines with structural hazards. In *Theorem Provers in Circuit Design*, pages 13–32. Springer, 1994. Published in LNCS Vol. 901 in 1995.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] Soha Hassoun and Carl Ebeling. Architectural retiming: Pipelining latency-constrained circuits. In *33rd Design Automation Conference*, pages 708–713. ACM/IEEE, 1996.

[5] James C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *International Conference on Computer-Aided Design*, pages 511–518. IEEE/ACM, 2000.

[6] Tor E. Jeremiassen. Sleipnir — an instruction-level simulator generator. In *International Conference on Computer Design*, pages 23–31. IEEE, 2000.

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

[8] Maria-Cristina Marinescu and Martin Rinard. High-level specification and efficient implementation of pipelined circuits. In *Asia and South Pacific Design Automation Conference*, 2001.

[9] John Matthews, John Launchbury, and Byron Cook. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, pages 90–101. IEEE, May 1998.

[10] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *International Conference on Computer Languages*, pages 80–89. IEEE, May 1998.

[11] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA — machine description language for cycle-accurate models of programmable DSP architectures. In *36th Design Automation Conference*, pages 933–938. ACM/IEEE, 1999.

[12] Norman Ramsey and Mary Fernández. The New Jersey machine-code toolkit. In *USENIX Technical Conference*, pages 289–302, January 1995. The current version is available at `http://www.eecs.harvard.edu/~nr/toolkit/`.

[13] Bruce Shriver and Bennett Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. IEEE Computer Society Press, 1998.

[14] R. Weicker. Dhrystone 2.1. *SIGPLAN Notices*, 23(8), August 1988. We downloaded from `ftp://ftp.nosc.mil/pub/aburto`.

Appendix A

PSL Language Reference

A.1 Introduction

PSL is a prototype language developed to embody the specification style detailed in Chapter 3 of this thesis. This appendix explains the syntax and semantics of the language, and Appendix B contains the complete PSL grammar.

A specification written in PSL consists of enumeration declarations, container declarations, and stage declarations. Further explanations are given in the following sections.

A.2 Keywords and Identifiers

An identifier is a case-sensitive sequence of alphabets, digits, and underscores where the first character is not a digit. For example, `10units` is not a legal identifier, and `buffer1` and `Buffer1` are distinct legal identifiers.

In PSL, identifiers are used as names for stages and containers. The stage names and container names exist in the same name space; that means the same identifier can not be used to identify both a stage and a container. In addition, the following sequences of characters are reserved keywords that can not be used as stage or container identifiers:

```
constructor  container  enum     false  goto
na   retire  stage      syscall  tr     true
```

The usage of these keywords will be explained in the following sections.

A.3 Constants

An integer constant is represented by one or more digits. If the first digit is 0, the entire constant is considered octal, otherwise the constant is considered decimal. Binary and hexadecimal integer constants can be specified by adding the prefix 0b and 0x, respectively.

Often it is more convenient to refer to integer constants by names. To allow this, a specification in PSL can contain enumeration declarations using the `enum` keyword which assigns an integer constant to a string literal. While the same integer can be represented by many string literals, each string literal can only represent one integer constant. For example, the following segment declares the string literal "Dozen" to represent the number 12:

```
enum "Dozen" := 12;
```

In addition, the keywords `false` and `true` are pre-defined constants that represent the integer constants 0 and 1, respectively.

The special values UNAVAILABLE (see Section A.6) and TRANSPARENT (see Section A.5.8) can be specified by the keywords `na` and `tr`, respectively.

A.4 Containers

There are two types of containers in PSL: *scalar containers* and *array containers*. A container that holds exactly one value is called a *scalar container* and is declared by using the `container` keyword followed by a unique identifier. A container that holds multiple values is called an *array container*. An array container's declaration requires two additional parameters: the smallest and largest index values. For example, the following section of code declares a scalar container `PC` and an array container `Table` where the valid entries are 2 to 10:

```
container PC;
container Table[2..10];
```

The integer values contained in a container must always have the exact same size as the native word size of the simulator's environment. Furthermore, the integer value can be interpreted in two different ways: as an unsigned integer or as a 2's-complement signed integer. The choice of interpretation depends only on the operator that operates on the value.

During the simulation of a processor, there is a global context where each container is associated with an integer value, as well as a per-instruction context for each instruction in which each container is associated with either an integer value or one of the two special values (UNAVAILABLE or TRANSPARENT). These two special values will be explained in later sections.

A.5 Expressions

This section describes the expression operators and their semantics. The precedence of all operators are listed in Table A.1 from highest precedence to lowest precedence.

Precedence	Operators							
Highest	`+`	`-`	`!`	`~`	— unary operators			
	`*SH`	`*SL`	`*UH`	`*UL`	`/S`	`/U`	`%S`	`%U`
	`+`	`-`						
	`<<`	`>>A`	`>>L`					
	`<S`	`<U`	`<=S`	`<=U`	`>S`	`>U`	`>=S`	`>=U`
	`==`	`!=`						
	`&`							
	`^`							
	`|`							
	`&&`							
	`^^`							
	`||`							
Lowest	`?:`							

Table A.1: Context diagrams

A.5.1 Unary Operators

There are 4 unary operators: the unary `+` operator, unary `-` operator, unary `!` operator, and unary `~` operator.

The unary `+` operator does nothing. It is provided for symmetry with the unary `-` operator. The unary `-` operator treats its operand as a signed integer, and the result is the negation of the integer. The only exception is when the operand is the minimum negative integer, the unary `-` operator does nothing.

The unary ~ operator treats its operand as a unsigned integer, and it complements every bit of the value. The unary ! operator returns 1 if its operand is equal to 0. Otherwise the result is 0. In this case, it does not matter if the operand is treated as a signed or unsigned integer.

A.5.2 Additive and Multiplicative Operators

The + (add) and - (subtract) operators compute the sum and difference of the operands, respectively. It does not matter if the operands are treated as signed or unsigned integers.

The `*SH` and `*SL` operators treat the operands as signed integers and return the upper word and lower word of the result, respectively. The `*UH` and `*UL` operators treat the operands as unsigned integers and return the upper word and lower word of the result, respectively.

The `/S` and `/U` operators treat the operands as signed and unsigned integers, respectively. The result is the division of these two numbers. If the second operand is 0, the result is implementation dependent.

The `%S` operator treats the first operand as a signed integer and the second operand as an unsigned integer. The operator computes the remainder of these two numbers. If the first operand is negative, or if the second operand is 0, the result is implementation dependent.

The `%U` operator treats both operands as unsigned integers and computes the remainder of these two numbers. If the second operand is 0, the result is implementation dependent.

A.5.3 Shift Operators

The `<<` and `>>L` operators treat the operands as unsigned integers and perform a bitwise shift to the left and to the right, respectively. If the second operand is larger than the number of bits, the result of the expression is zero.

The `>>A` operator treats the first operand as a signed integer and the second operand as an unsigned integer. The result is the arithmetic shift to the right. That means the incoming bit is 1 when the original first operand is negative, and 0 if the original first operand is positive. If the second operand is larger than the number of bits, the result of the expression is 0 if the first operand is positive, and -1 if the first operand is negative.

A.5.4 Comparison Operators

The `==` (equal to), `!=` (not equal to), `<` (less than), `<=` (less than or equal to), `>` (greater than), and `>=` (greater than or equal to) operators will return 1 if the specified comparison is true, and 0 if the specified comparison is false.

For the `==` and `!=` operators it does not matter whether the operands are treated as signed or unsigned integers. However, the other operators require a suffix (`S` and `U`) to specify whether the operands should be considered signed or unsigned integers, respectively.

A.5.5 Bitwise Operators

The `&`, `|`, and `^` operators treat the operands as unsigned integers and compute the bitwise AND, bitwise OR, and bitwise XOR, respectively.

A.5.6 Logical Operators

The `&&` operator returns 0 if at least one of the operands is equal to 0. Otherwise it returns 1. The `||` operator returns 1 if at least one of the operands is not equal to 0. Otherwise it returns 0. The `^^` operator returns 0 in two situations: when both operands are equal to 0, and when neither operand is equal to 0. Otherwise the `^^` operator returns 1.

A.5.7 If-Then-Else Operator

The `?:` operator takes three arguments. To evaluate the expression `a?b:c`, the operand `a` is evaluated first. If `a` is not equal to 0, then the third operand `c` is ignored, and the entire expression equals the value of `b`. On the other hand, if `a` is equal to 0, then the second operand `b` is ignored, and the entire expression equals the value of `c`.

A.5.8 Container References

A scalar container is referenced by its identifier. An array container is always referenced by its identifier and an integer index into the array. It is impossible to refer to multiple entries of an array as a single expression.

By default, the container reference uses the default access mode. An optional suffix can be attached after the identifier to specify one of three alternative access operators. Given a container `c`, the default access `c` denotes the most recent non-TRANSPARENT value of the container, starting from the present context; `c'` denotes the most recent non-TRANSPARENT value of the container, starting from the *previous* context; `c$` denotes the most recent non-TRANSPARENT value of the container from the previous context *after all functional units have completed the current clock cycle*;

and `c#` denotes the value of the container in the global context.

Furthermore, a range of bits can be extracted by using the `[..]` operator. Given a container reference `e`, the expression `e[a..b]` treats `e` as an unsigned integer and returns the range of bits from bit `a` to bit `b` inclusively with bit `b` as the least significant bit. If the default word size is bigger than `a-b+1` bits, the unspecified bits will be 0.

The `[[..]]` operator performs the same operation as `[..]`, except the `[[..]]` operator sign-extends the expression. Given a container reference `e`, the expression `e[[a..b]]` treats `e` as an unsigned integer and returns the range of bits from bit `a` to bit `b` inclusively with bit `b` as the least significant bit. If the default word size is bigger than `a-b+1` bits, the unspecified bits will be equal to bit `a`.

A.5.9 Syscalls

The generated simulator may interact with an outside environment via the `syscall` interface. Whenever the expression `syscall(a,b,c)` is evaluated, the three arguments will first be evaluated into integers, then the syscall handler will be invoked with the three integers as parameters. A custom syscall handler can be supplied to the generated simulator to provide additional capabilities such as statistics gathering, system I/O emulation, etc.

A.6 Statements

There are four different types of statements in PSL: announcements, commitments, goto statements, and retire statements. The announcement statement is an announcement that modifies the current instruction's context. It does not modify the context of other instructions nor the global context. For example, the following guarded statement will announce `PC` to be 0 if the expression `ExceptionRaised` evaluates to a non-zero value:

```
ExceptionRaised: { PC <- 0; }
```

The commitment statement is used to modify the global context shared by all instructions. It does not modify the context of any instruction. For example, the following guarded statement changes the value of `Lo` in the global context to 10:

```
true: { Lo := 10; }
```

The goto statement is used to specify the destination stage that the current instruction should go into if possible. For example, the following guarded statement indicates that the instruction should move into the ALU stage before the next cycle if the expression `OperandFetched` evaluates to a value greater than 2:

```
OperandFetched > 2 : { goto ALU; }
```

The retire statement is used to indicate the instruction no longer exists, and that its instruction context should be discarded. For example, the following guarded statement indicates that the instruction should be discarded before the next cycle:

```
true: { retire; }
```

During each clock cycle, each instruction evaluates all of its guard expressions. Each instruction then attempts to execute all enabled statements. If an instruction encounters an UNAVAILABLE value during its evaluation of guards or the execution of statements, then this instruction is deemed to have *failed* (meaning it will discard all of the changes it made during this cycle, stall, and attempt to re-execute in the next cycle). However, if an instruction completes all evaluation and execution, but its destination stage is occupied, then the instruction will stall with all changes intact.

A.7 Stages

A stage can be declared in PSL using the keyword `stage` followed by the identifier of the stage. Inside a stage's declaration is a list of guarded statements. For example, the following is a valid stage declaration:

```
stage Compute
 {
  Flag' < 40 : { Flag <- Flag' + 1;             }
  !Exception : { Ready <- true; goto Dispatch; }
  Exception  : { retire;                        }
 }
```

In addition, one and exactly one stage must be declared as a constructor by using the `constructor` keyword instead of the `stage` keyword. For example, the following is a valid constructor declaration:

```
constructor Fetch
 {
  true: { OpCode <- Mem#[PC'];  PC <- PC' + 4;  goto Decode; }
 }
```

Intuitively, a *stage* represents a typical pipeline stage in a processor. During each clock cycle, exactly zero or one instruction may be inside any given stage. If an instruction is present, then all enabled statements in that stage will be executed with respect to that instruction's context. If during the execution, some of the expressions can not be evaluated, then the entire instruction stalls and none of the statements will be executed. Otherwise, at the end of the cycle the instruction will either move into another stage (if the desired destination stage will be vacant in the next cycle clock) or the instruction will stay in the current stage (if the desired destination stage will be occupied in the next cycle). Furthermore, whenever the constructor stage becomes vacant, a new instruction context (where all containers are labeled as UNAVAILABLE) will be created and moved into the constructor stage before the beginning of the next clock cycle.

Appendix B

PSL Grammar

Below is the grammar for the prototype language PSL. Nonterminal symbols are represented in italic style, whereas the terminals are represented in teletype style inside boxes. The three undefined terminal symbols (Identifier, IntegerConstant, and StringLiteral) are explained in Appendix A.

file →

    declaration file
  | *declaration*

declaration →

    enum-declaration
  | *container-declaration*
  | *stage-declaration*

enum-declaration →

    $\boxed{\texttt{enum}}$ *enum-entry enum-declarations*

enum-declarations →

    $\boxed{\texttt{;}}$
  | $\boxed{\texttt{,}}$ *enum-entry enum-declarations*

enum-entry $\rightarrow$

StringLiteral

| *StringLiteral* $\boxed{\texttt{:=}}$ *IntegerConstant*

container-declaration $\rightarrow$

$\boxed{\texttt{container}}$ *container-entry container-declarations*

container-declarations $\rightarrow$

$\boxed{\texttt{;}}$

| $\boxed{\texttt{,}}$ *container-entry container-declarations*

container-entry $\rightarrow$

Identifier

| *Identifier* $\boxed{\texttt{[}}$ *IntegerConstant* $\boxed{\texttt{..}}$ *IntegerConstant* $\boxed{\texttt{]}}$

stage-declaration $\rightarrow$

$\boxed{\texttt{constructor}}$ *Identifier* $\boxed{\{}$ $\boxed{\}}$

| $\boxed{\texttt{constructor}}$ *Identifier* $\boxed{\{}$ *stage-body* $\boxed{\}}$

| $\boxed{\texttt{stage}}$ *Identifier* $\boxed{\{}$ $\boxed{\}}$

| $\boxed{\texttt{stage}}$ *Identifier* $\boxed{\{}$ *stage-body* $\boxed{\}}$

stage-body $\rightarrow$

block

| *block stage-body*

block $\rightarrow$

expression-right $\boxed{\texttt{:}}$ $\boxed{\{}$ *statements* $\boxed{\}}$

| *expression-right* $\boxed{\texttt{:}}$ $\boxed{\{}$ $\boxed{\}}$

statements $\rightarrow$

statement

| *statement statements*

statement →

expression-left `<-` *expression-right* `;`

| *expression-left* `:=` *expression-right* `;`

| `goto` *identifier* `;`

| `retire` `;`

expression-left →

Identifier

| *Identifier* `[` *expression-right* `]`

| *Identifier* `[` `..` `]`

| *Identifier* `[` `..` *expression-right* `]`

| *Identifier* `[` *expression-right* `..` `]`

| *Identifier* `[` *expression-right* `..` *expression-right* `]`

expression-right →

expression-if-then-else

| *expression-logical-or*

expression-if-then-else →

expression-logical-or `?` *expression-right* `:` *expression-right*

expression-logical-or →

expression-logical-xor

| *expression-logical-or* `||` *expression-logical-xor*

expression-logical-xor →

expression-logical-and

| *expression-logical-xor* `^^` *expression-logical-and*

expression-logical-and →

expression-bitwise-or

| *expression-logical-and* `&&` *expression-bitwise-or*

expression-bitwise-or $\rightarrow$

expression-bitwise-xor

$|$ *expression-bitwise-or* $\boxed{\texttt{|}}$ *expression-bitwise-xor*

expression-bitwise-xor $\rightarrow$

expression-bitwise-and

$|$ *expression-bitwise-xor* $\boxed{\texttt{\^{}}}$ *expression-bitwise-and*

expression-bitwise-and $\rightarrow$

expression-equality

$|$ *expression-bitwise-and* $\boxed{\texttt{\&}}$ *expression-equality*

expression-equality $\rightarrow$

expression-inequality

$|$ *expression-inequality* $\boxed{\texttt{==}}$ *expression-inequality*

$|$ *expression-inequality* $\boxed{\texttt{!=}}$ *expression-inequality*

expression-inequality $\rightarrow$

expression-shift

$|$ *expression-shift* $\boxed{\texttt{<S}}$ *expression-shift*

$|$ *expression-shift* $\boxed{\texttt{<U}}$ *expression-shift*

$|$ *expression-shift* $\boxed{\texttt{<=S}}$ *expression-shift*

$|$ *expression-shift* $\boxed{\texttt{<=U}}$ *expression-shift*

$|$ *expression-shift* $\boxed{\texttt{>S}}$ *expression-shift*

$|$ *expression-shift* $\boxed{\texttt{>U}}$ *expression-shift*

$|$ *expression-shift* $\boxed{\texttt{>=S}}$ *expression-shift*

$|$ *expression-shift* $\boxed{\texttt{>=U}}$ *expression-shift*

expression-shift $\rightarrow$

expression-sum

$|$ *expression-shift* $\boxed{\texttt{<<}}$ *expression-sum*

$|$ *expression-shift* $\boxed{\texttt{>>A}}$ *expression-sum*

$|$ *expression-shift* $\boxed{\texttt{>>L}}$ *expression-sum* *expression-sum* →

expression-product
| *expression-sum* `+` *expression-product*
| *expression-sum* `-` *expression-product*

expression-product →

expression-unary
| *expression-product* `*SH` *expression-unary*
| *expression-product* `*SL` *expression-unary*
| *expression-product* `*UH` *expression-unary*
| *expression-product* `*UL` *expression-unary*
| *expression-product* `/S` *expression-unary*
| *expression-product* `/U` *expression-unary*
| *expression-product* `%S` *expression-unary*
| *expression-product* `%U` *expression-unary*

expression-unary →

expression-post
| `+` *expression-unary*
| `-` *expression-unary*
| `!` *expression-unary*
| `~` *expression-unary*

expression-post →

`syscall` `(` *expression-right* `,` *expression-right* `,` *expression-right* `)`
| *constant*
| `(` *expression-right* `)`
| *expression-container-ref*
| *expression-container-ref* `[` *expression-right* `..` *expression-right* `]`
| *expression-container-ref* `[` `[` *expression-right* `..` *expression-right* `]` `]`

expression-container-ref →

expression-container
| *expression-container* `[` *expression-right* `]`

$expression\text{-}container \rightarrow$

$\quad Identifier$

$\quad | \; Identifier \; \boxed{\texttt{\#}}$

$\quad | \; Identifier \; \boxed{\texttt{\$}}$

$\quad | \; Identifier \; \boxed{\texttt{'}}$

$constant \rightarrow$

$\quad \boxed{\texttt{na}}$

$\quad | \; \boxed{\texttt{tr}}$

$\quad | \; \boxed{\texttt{true}}$

$\quad | \; \boxed{\texttt{false}}$

$\quad | \; IntegerConstant$

$\quad | \; StringLiteral$

Appendix C

MIPS R2000 Specification

Below is the complete specification of the integer core of the MIPS R2000 [7] microprocessor, including all user-mode instructions except coprocessor support, floating point instructions, and non-aligned loads/stores. The model we present here does not implement exceptions, although they could be added easily by following the suggestions in Section 5.1.6.

```
//==============================================================================

enum "ALU-SLL"  :=   0, "ALU-SRL"  :=   2, "ALU-SRA" :=   3;
enum "ALU-MFHI" :=  16, "ALU-MTHI" :=  17, "ALU-MFLO":=  18, "ALU-MTLO":= 19;
enum "ALU-MULT" :=  24, "ALU-MULTU":=  25, "ALU-DIV" :=  26, "ALU-DIVU":= 27;
enum "ALU-ADD"  :=  32, "ALU-ADDU" :=  33, "ALU-SUB" :=  34, "ALU-SUBU":= 35;
enum "ALU-AND"  :=  36, "ALU-OR"   :=  37, "ALU-XOR" :=  38, "ALU-NOR" := 39;
enum "ALU-SLT"  :=  42, "ALU-SLTU" :=  43, "ALU-LUI" := 501, "NOALU"   :=502;
enum "LINK"     := 503, "SYSCALL"  := 504;
enum "MEM-LH"   := 545, "MEM-LB"   := 544, "MEM-LHU" := 549, "MEM-LBU" :=548;
enum "MEM-SH"   := 553, "MEM-SB"   := 552, "MEM-SW"  := 555, "MEM-LW"  :=547;

//==============================================================================

container Mem[0..10485760];

container Reg[0..32]; // R[32] is Don't Care.

container PN,PF;
container PC,INS, T,T2, V1,V2,V3, Offset, Op, Hi,Lo, HiTemp,LoTemp;

//==============================================================================

stage WriteBackStage
```

```
 {
  true: { retire; }

  T: { Reg[T] := Reg[T]; }

  T2: { Hi:=HiTemp; Hi<-HiTemp; Lo:=LoTemp; Lo<-LoTemp; }

  Op=="SYSCALL": { Reg[2] := syscall(V1,V2,V3); }

  (Op & 520)==520: { Mem[Offset >>L 2] := Mem[Offset >>L 2]; }
 }

//==============================================================================

stage MemoryStage
 {
  // Memory Reads

  Op=="MEM-LW"                  :{ Reg[T?T:32]<- Mem[Offset >>L 2]          ; }
  Op=="MEM-LH"  && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2][[31..16]]; }
  Op=="MEM-LH"  && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2][[15.. 0]]; }
  Op=="MEM-LHU" && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2] [31..16] ; }
  Op=="MEM-LHU" && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2] [15.. 0] ; }
  Op=="MEM-LB"  && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2][[31..24]]; }
  Op=="MEM-LB"  && (Offset&3)==1:{ Reg[T?T:32]<- Mem[Offset >>L 2][[23..16]]; }
  Op=="MEM-LB"  && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2][[15.. 8]]; }
  Op=="MEM-LB"  && (Offset&3)==3:{ Reg[T?T:32]<- Mem[Offset >>L 2][[ 7.. 0]]; }
  Op=="MEM-LBU" && (Offset&3)==0:{ Reg[T?T:32]<- Mem[Offset >>L 2] [31..24] ; }
  Op=="MEM-LBU" && (Offset&3)==1:{ Reg[T?T:32]<- Mem[Offset >>L 2] [23..16] ; }
  Op=="MEM-LBU" && (Offset&3)==2:{ Reg[T?T:32]<- Mem[Offset >>L 2] [15.. 8] ; }
  Op=="MEM-LBU" && (Offset&3)==3:{ Reg[T?T:32]<- Mem[Offset >>L 2] [ 7.. 0] ; }

  // Memory Writes

  (Op & 520)==520:{ Mem[..((Offset >>L 2)-1)]<-Mem[((Offset >>L 2)+1)..]<-tr;}

  Op=="MEM-SW": { Mem[Offset >>L 2] <- V3; }

  Op=="MEM-SB" && (Offset&3)==0:
   {
    Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0x00FFFFFF) | ((V3 & 0xFF)<<24);
   }

  Op=="MEM-SB" && (Offset&3)==1:
   {
    Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFF00FFFF) | ((V3 & 0xFF)<<16);
   }

  Op=="MEM-SB" && (Offset&3)==2:
   {
    Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFFFF00FF) | ((V3 & 0xFF)<<8);
   }

  Op=="MEM-SB" && (Offset&3)==3:
   {
    Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFFFFFF00) | (V3 & 0xFF);
   }

  Op=="MEM-SH" && (Offset&3)==0:
   {
```

```
   Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0x0000FFFF) | (V3 << 16);
  }

 Op=="MEM-SH" && (Offset&3)==2:
  {
   Mem[Offset >>L 2] <- (Mem'[Offset >>L 2] & 0xFFFF0000) | (V3 & 0xFFFF);
  }

 true: { goto WriteBackStage; }
}

//===============================================================================

stage ALUStage
{
 Op=="ALU-SLL"   : { Reg[T ? T : 32] <-  (V1 <<  V2) ; }
 Op=="ALU-SRA"   : { Reg[T ? T : 32] <-  (V1 >>A V2) ; }
 Op=="ALU-SRL"   : { Reg[T ? T : 32] <-  (V1 >>L V2) ; }
 Op=="ALU-MFHI"  : { Reg[T ? T : 32] <-  (Hi'      ) ; }
 Op=="ALU-MFLO"  : { Reg[T ? T : 32] <-  (Lo'      ) ; }
 Op=="ALU-MTHI"  : { HiTemp          <-  (V1       ) ; }
 Op=="ALU-MTLO"  : { LoTemp          <-  (V1       ) ; }
 Op=="ALU-ADD"   : { Reg[T ? T : 32] <-  (V1 +   V2) ; }
 Op=="ALU-ADDU"  : { Reg[T ? T : 32] <-  (V1 +   V2) ; }
 Op=="ALU-SUB"   : { Reg[T ? T : 32] <-  (V1 -   V2) ; }
 Op=="ALU-SUBU"  : { Reg[T ? T : 32] <-  (V1 -   V2) ; }
 Op=="ALU-AND"   : { Reg[T ? T : 32] <-  (V1 &   V2) ; }
 Op=="ALU-OR"    : { Reg[T ? T : 32] <-  (V1 |   V2) ; }
 Op=="ALU-XOR"   : { Reg[T ? T : 32] <-  (V1 ^   V2) ; }
 Op=="ALU-NOR"   : { Reg[T ? T : 32] <- ~(V1 |   V2) ; }
 Op=="ALU-SLT"   : { Reg[T ? T : 32] <-  (V1 <S  V2) ; }
 Op=="ALU-SLTU"  : { Reg[T ? T : 32] <-  (V1 <U  V2) ; }
 Op=="ALU-LUI"   : { Reg[T ? T : 32] <-  (V2 <<  16) ; }
 Op=="LINK"      : { Reg[T ? T : 32] <-  (PC +    8) ; }

 Op=="ALU-MULT"  : { HiTemp<-(V1 *SH V2); LoTemp<-(V1 *SL V2); }
 Op=="ALU-MULTU" : { HiTemp<-(V1 *UH V2); LoTemp<-(V1 *UL V2); }
 Op=="ALU-DIV"   : { HiTemp<-(V1 %S  V2); LoTemp<-(V1 /S  V2); }
 Op=="ALU-DIVU"  : { HiTemp<-(V1 %U  V2); LoTemp<-(V1 /U  V2); }

 (Op & 512)==512 : { Offset <- (V1 + V2); } // MEMORY

 true: { goto MemoryStage; }
}

//===============================================================================

stage DecodeStage
{
 true: { goto ALUStage; }

 (INS & 0xFC00003F) == 12: // SYSCALL
  {
   Mem[..]<-Reg[..1]<-Reg[3..]<-Hi<-Lo<-tr; T<-0; T2<-0; Op<-"SYSCALL";
   V1 <- Reg$[2];
   V2 <- Reg$[4];
   V3 <- Reg$[5];
  }

 (INS & 0xFC000030) == 0b100000: // ADD[U],SUB[U],AND,OR,XOR,NOR,SLT[U]
```

```
 {
  Mem[..] <- Reg[..INS[15..11]-1] <- Reg[INS[15..11]+1..]<- Hi<- Lo<- tr;
  T2 <- 0;
  V1 <- Reg$[INS[25..21]]; Op <- INS[ 5.. 0];
  V2 <- Reg$[INS[20..16]]; T  <- INS[15..11];
 }

(INS & 0xFC00003C) == 0: // SLL,SRL,SRA
 {
  Mem[..] <- Reg[..INS[15..11]-1] <- Reg[INS[15..11]+1..]<- Hi<- Lo<- tr;
  V1<-Reg$[INS[20..16]];
  T2<-0;  T<-INS[15..11];  Op<-(INS&3); V2<-INS[10..6];
 }

(INS & 0xFC00003C) == 4: // SLLV,SRLV,SRAV
 {
  Mem[..] <- Reg[..INS[15..11]-1] <- Reg[INS[15..11]+1..]<- Hi<- Lo<- tr;
  V1 <-  Reg$[INS[20..16]];
  V2 <- (Reg$[INS[25..21]] & 0x1F); T<-INS[15..11]; T2<-0; Op<-INS&3;
 }

(INS & 0xFC00003C) == 0b011000: // MULT,MULTU,DIV,DIVU
 {
  T<-0; T2<-1; Op<-INS[5..0];
  V1<-Reg$[INS[25..21]]; V2<-Reg$[INS[20..16]]; Reg[..]<-Mem[..]<-tr;
 }

(INS & 0xFC00003F) == 0b010001: // MTHI
 {
  V1<-Reg$[INS[25..21]]; Reg[..]<-Mem[..]<-Lo<-tr; T<-0; T2<-1; Op<-17;
 }

(INS & 0xFC00003F) == 0b010011: // MTLO
 {
  V1<-Reg$[INS[25..21]]; Reg[..]<-Mem[..]<-Hi<-tr; T<-0; T2<-1; Op<-19;
 }

(INS & 0xFC00003D) == 0b010000: // MFHI,MFLO
 {
  Reg[..INS[15..11]-1] <- Reg[INS[15..11]+1..]<- Mem[..]<- Hi<- Lo<- tr;
  T<-INS[15..11];  T2<-0;  Op<-INS[5..0];
 }

(INS & 0xFC000000) == 0x3C000000: // LUI
 {
  Mem[..] <- Reg[INS[20..16]+1..] <- Reg[..INS[20..16]-1]<- Hi<- Lo<- tr;
  T<-INS[20..16];  T2<-0;  Op<-"ALU-LUI";  V2<-INS[15..0];
 }

// ADDI , ADDIU , SLTI , SLTIU , ANDI , ORI , XORI

(INS & 0xE0000000)==0x20000000 && (INS & 0xFC000000)!=0x3C000000:
 {
  Mem[..] <- Reg[INS[20..16]+1..] <- Reg[..INS[20..16]-1]<- Hi<- Lo<- tr;
  V1 <- Reg$[INS[25..21]]; T <- INS[20..16];  T2<-0;
 }

(INS & 0xFC000000)==0x20000000: { Op<-"ALU-ADD"  ; V2 <- INS [[15..0]] ; }
(INS & 0xFC000000)==0x24000000: { Op<-"ALU-ADDU" ; V2 <- INS [[15..0]] ; }
(INS & 0xFC000000)==0x28000000: { Op<-"ALU-SLT"  ; V2 <- INS [[15..0]] ; }
```

```
(INS & 0xFC000000)==0x2C000000: { Op<-"ALU-SLTU" ; V2 <- INS [[15..0]] ; }
(INS & 0xFC000000)==0x30000000: { Op<-"ALU-AND"  ; V2 <- INS  [15..0]  ; }
(INS & 0xFC000000)==0x34000000: { Op<-"ALU-OR"   ; V2 <- INS  [15..0]  ; }
(INS & 0xFC000000)==0x38000000: { Op<-"ALU-XOR"  ; V2 <- INS  [15..0]  ; }

//

(INS & 0xFC00003F)==0b1000: // JR
 {
  PN := Reg$[INS[25..21]];
  PF := 1; Mem[..]<-Reg[..]<-Hi<-Lo<-tr; Op<-"LINK"; T<-0; T2<-0;
 }

(INS & 0xFC00003F)==0b1001: // JALR
 {
  PN := Reg$[INS[25..21]];
  PF := 1; Op<-"LINK"; T2<-0; T<-INS[15..11];
  Mem[..]<-Reg[..INS[15..11]-1]<-Reg[INS[15..11]+1..]<-Hi<-Lo<-tr;
 }

(INS & 0xFC000000)==0x08000000: // J
 {
  PN := (PC&0xF0000000) | (INS[25..0]<<2) ;
  PF := 1; Mem[..]<-Reg[..]<-Hi<-Lo<-tr; Op<-"LINK"; T<-0; T2<-0;
 }

(INS & 0xFC000000)==0x0C000000: // JAL
 {
  PN := (PC&0xF0000000) | (INS[25..0]<<2) ;
  PF := 1; Mem[..]<-Reg[..30]<-Hi<-Lo<-tr; Op<-"LINK"; T<-31; T2<-0;
 }

(INS & 0xFC010000)==0x04000000: // BLTZ,BGEZ
 {
  PN:=PC+(INS[[15..0]]<<2)+4;  Mem[..]<-Reg[..]<-Hi<-Lo<-tr;  Op<-"LINK";
  PF:=(INS&0x10000) ^^ (Reg$[INS[25..21]] <S 0);  T<-0; T2<-0;
 }

(INS & 0xFC010000)==0x04010000: // BLTZAL,BGEZAL
 {
  PN:=PC+(INS[[15..0]]<<2)+4;  Mem[..]<-Reg[..30]<-Hi<-Lo<-tr;  Op<-"LINK";
  PF:=(INS&0x10000) ^^ (Reg$[INS[25..21]] <S 0);  T<-31; T2<-0;
 }

// BEQ / BNE / BLEZ / BGTZ

(INS & 0xF0000000)==0x10000000: //BEQ,BNE,BLEZ,BGTZ
 {
  PN:=PC+(INS[[15..0]]<<2)+4; Mem[..]<-Reg[..]<-Hi<-Lo<-tr; Op<-"NOALU";
  T<-0; T2<-0;
 }

(INS & 0xFC000000)==0x10000000: { PF:=Reg$[INS[25..21]]==Reg$[INS[20..16]]; }
(INS & 0xFC000000)==0x14000000: { PF:=Reg$[INS[25..21]]!=Reg$[INS[20..16]]; }
(INS & 0xFC000000)==0x18000000: { PF:=Reg$[INS[25..21]]<=S 0              ; }
(INS & 0xFC000000)==0x1C000000: { PF:=Reg$[INS[25..21]] >S 0              ; }

//

(INS & 0xE0000000)==0x80000000: // LW, LB(U), LH(U)
```

```
   {
    Mem[..] <- Reg[..INS[20..16]-1] <- Reg[INS[20..16]+1..]<- Hi<- Lo<- tr ;
    Op <- INS[31..26] + 512;
    V1 <- Reg$[INS[25..21]] ; V2<-INS[[15.. 0]] ; T<-INS[20..16];  T2<-0;
   }

  (INS & 0xE0000000)==0xA0000000: // SW, SWB, SWH
   {
    Op <- INS[31..26] + 512;
    V1 <- Reg$[INS[25..21]];
    V3 <- Reg$[INS[20..16]]; V2<-INS[[15..0]]; Reg[..]<-Hi<-Lo<-tr; T<-T2<-0;
   }
 }

//============================================================================

constructor FetchStage
 {
  PF# == 0 : { INS <- Mem#[ (PC'+4) >>L 2 ] ; PC<-PC'+4 ;         }
  PF# == 1 : { INS <- Mem#[ (PN#  ) >>L 2 ] ; PC<-PN#   ; PF:=0 ; }

  true: { Reg[0]<-tr; goto DecodeStage; }
 }

//============================================================================
```