KTH Information and Communication Technology

Instant Simulator

A tool for generating simulators from machine-readable descriptions

KARL JOHANSSON

Master Thesis at ICT
Supervisor: Frej Drejhammar (SICS)
Examiner: Christian Schulte (School of ICT)

TRITA-ICT-EX-2014:106

Abstract

This thesis presents Instant Simulator, a tool for simulating digital signal processors (DSPs) to allow cycle-accurate benchmarking of programs. The purpose of the tool is to assist the Unison compiler when comparing generated machine code between Unison and industrial strength compilers

Instant Simulator is designed to reuse parts of Unison and strike a balance between performance and ease of use. DSPs are described in machine-readable specifications and compiled into simulators, partly with the help of tools developed for Unison.

A case study is conducted to evaluate Instant Simulator, where the C67x DSP from Texas Instruments is simulated during execution of small test programs. The case study ensures the correctness and completeness of the tool and the simulations executes at a clock rate between 2.6 MHz and 9.7 MHz on a modern mid-tier laptop, depending on the level of detail in simulation.

Acknowledgments

I want to express my deepest gratitude to my supervisor, Frej Drejhammar. Frej has been encouraging and supportive, not only throughout the work for this thesis, but also during my time working at SICS. Frej authored several key components in Instant Toolsmith, for which the work in this thesis is heavily dependent on.

I am very grateful for the opportunity Christian Schulte gave me by hiring me at SICS. Christian's two courses at KTH and his guidance at SICS has truly been valuable. I consider my time working at SICS as more rewarding than the last two years of studying at KTH.

I want to thank Mattias Jansson for his collaboration during the last two years, for his support in general and the sound advice he provided during the work on this thesis. Mattias' ability to detect irrelevant text helped this report become more concise.

Lastly, I am thankful to my family for their support. In particular my girlfriend Irini, who has endured unsolicited monologues about computers, among other eccentricities during the past few months.

Contents

Contents

List of Algorithms

List of Figures

Listings

Chapter 1

Introduction

This thesis presents the implementation and evaluation of Instant Simulator. Instant Simulator is a tool providing simulation of digital signal processors (DSPs), capable of providing cycle-accurate benchmarking of simple programs. The tool is designed to quickly support new DSP architectures (and revised DSP architectures) without needing to update complex underlying source code. Furthermore, its design balances ease of use with performance considerations.

1.1 Motivation

Instant Simulator is created to support the development of the compiler Unison [17]. Generated machine code from Unison will be compared to generated machine code from industrial-strength compilers on simulated DSPs. The following is required of Instant Simulator:

Support for multiple DSPs Doing comparisons between compilers on multiple DSPs strengthens the evaluation of Unison. Instant Simulator therefore needs to allow simulation of multiple DSPs.

Completeness Unison is partly developed to effortlessly support new DSP architectures and Instant Simulator must have the same goals. In particular, Instant Simulator will need to simulate DSPs at a sufficient level of detail.

Correctness Execution of programs on simulated computers must be correct, both in program behavior and in cycle accuracy.

Performance Although less essential than previous requirements, it is important that simulation is as fast as possible to increase the usefulness of the tool.

1.2 Solution

Instant Simulator reuse the tools developed to let Unison support multiple DSPs. Unison allows the needed processor-specific details of DSPs to be described in a user-friendly architecture description language (ADL), named Instant Toolsmith ADL. These machine-readable descriptions will be evaluated by a tool, generating the needed source code to make a complete compiler. Section 2.3 expands further on the usage of an ADL. Instant Toolsmith ADL is also used by Instant Simulator, allowing simulation based on the same descriptions of DSPs made for Unison.

The design of Instant Simulator is optimized for performance, by generating C source code based on the machine-readable descriptions, which can be compiled into fast simulators.

Correctness, completeness, and performance are all evaluated by conducting a case study in Chapter 6, simulating the C67x DSP from Texas Instruments [7]. Completeness is evaluated by investigating how easily the C67x DSP can be specified in Instant Toolsmith ADL and how well simulation captures the relevant details of a DSP. Correctness and performance are evaluated by running two test programs, observing at the results of execution (both in program behavior and cycle-accuracy) and the time spent executing the programs.

1.3 Contribution

The contributions of this thesis are the following:

1. The design and implementation of Instant Simulator. In particular how the implementation reuse Instant Toolsmith ADL, developed for Unison to support multiple DSPs.
2. The evaluation of Instant Simulator in a case study, demonstrating the capabilities of the tool by simulating the C67x DSP [7].

Instant Toolsmith ADL (documented in Appendix A) is co-developed with Mattias Jansson and Frej Drejhammar. Instant Simulator heavily benefits from a currently nameless evaluation tool, also co-developed with Mattias and Frej. In one part of its implementation (covered in Section 5.4), Instant Simulator makes use of Instant Disassembler, a tool solely developed by Frej.

1.4 Ethics and Sustainability

No ethical quandaries, nor any effects on sustainability surfaced when conducting the research for this thesis.

1.5 Thesis Structure

The details of simulating multiple DSPs, benefited from the use of an ADL, is introduced in Chapter 2. This chapter describes the Instant Toolsmith ADL and ends with a summary of related work in other ADLs.

Chapter 3 outlines the design of Instant Simulator, described in greater detail in Chapter 4 and Chapter 5.

Instant Simulator is evaluated in a case study in Chapter 6. Conclusions drawn from the evaluation and recommendations for future work are given in Chapter 7.

Instant Toolsmith ADL, the ADL used in machine-readable specifications of DSPs for Instant Simulator, is documented in Appendix A, co-authored with Mattias Jansson.

Chapter 2

Background

The goal of this chapter is to explain the internals of a computer at a level of detail, necessary for the purpose of computer simulation. Furthermore, this chapter aims to lay a foundation for the design and implementation of Instant Simulator. These goals are attempted by first demystifying the workings of the central processing unit (hereafter referred to as a processor) of a computer and then explain how they can be simulated in software programs. Simulation is elaborated further by motivating the support of multiple processors and introducing the benefits of an ADL. Finally, Instant Toolsmith ADL and the class of processors supported by Instant Simulator (digital signal processors) are introduced before ending this chapter with related work in ADLs.

2.1 A Metaphor for Computers

Imagine someone preparing a meal with the help of a recipe. Following the cooking instructions found in the recipe, this person shares many similarities with a computer executing a program. Figure 2.1 shows a model outlining core components providing the functionality needed to prepare a meal, now described in detail.

Cooking instructions are provided in textual form as input to a reader component, which will read, understand and transform instructions into an mental image of actions to be performed by an actor on some input.

The actor performs the actions needed to complete each cooking

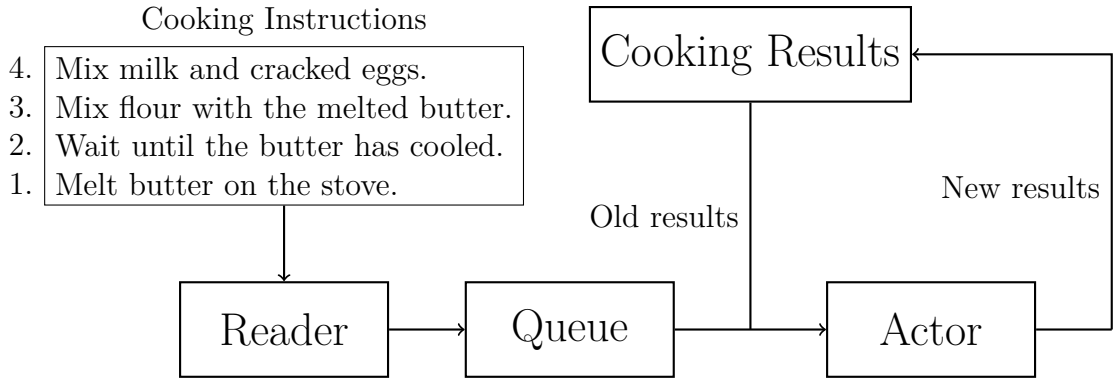

Figure 2.1: *A model of the needed functionality when preparing a meal. The reader components reads cooking instructions, which are planned in the queue before being executed by the actor. The cooking results are repeatedly updated until the preparation of the meal is completed.*

instruction, using the necessary incomplete cooking results from earlier operations. Cooking results are updated repeatedly for each cooking instruction completed, until there are no more instructions to carry out, in which case the preparation of the meal is complete.

Some cooking instructions have to be carried out before others because instructions make use of previous results. This is managed in queue structure where instructions await the completion of needed temporary results. Our model in Figure 2.1 contains example input which first instructs the cook to melt butter, then wait until the temperature drops before mixing it with flour. People devoid of skills in the culinary arts might lack the capabilities to handle this complexity, mixing flour and butter while the temperature is still too hot, resulting in inedible food. Others are skillful enough to reorder the instructions, not only preparing the meal correctly, but also reducing the time it takes to prepare the meal. For instance, they might mix milk and eggs while waiting for the melted butter to cool off.

2.2 The Internals of a Processor

Even though the metaphor used in Section 2.1 is an unnatural way of describing a cook, it captures the essential functionality of a processor in a computer. Processors receive instructions as input, which are scheduled

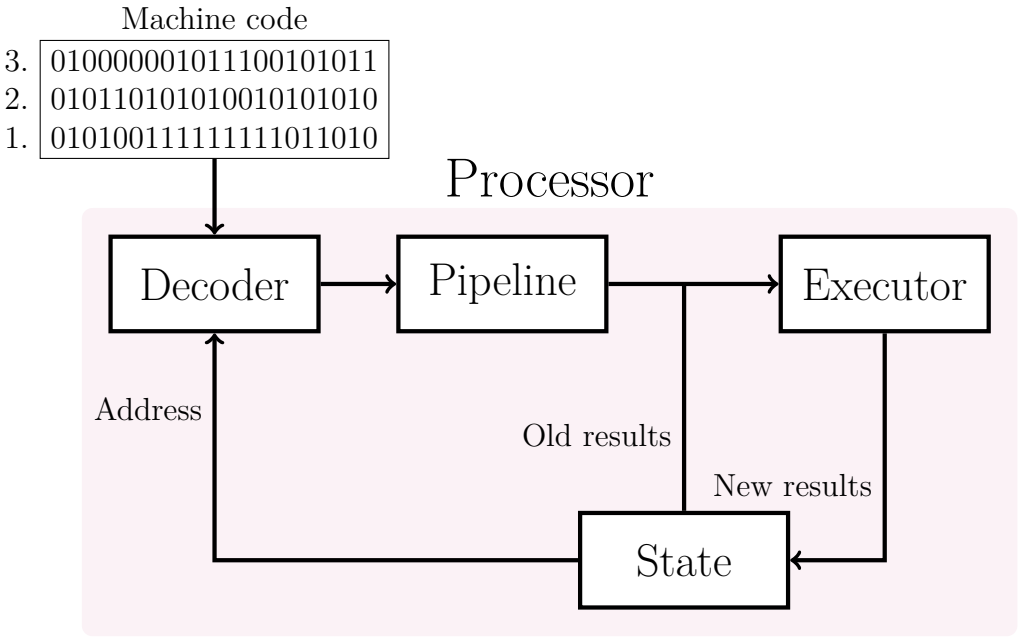

Figure 2.2: *A model of the core functionality that make up a processor. The decoder retrieves and decodes instructions at a specific address in the machine code. The decoded instruction is then scheduled in a pipeline before being carried out by an executor. State is repeatedly used as input to the instructions, while being updated as the machine code is executed.*

and carried out on previous results to get new results. This section describes the core functionality of a processor in detail by use of the model shown in Figure 2.2, based on our model of meal preparation shown in Figure 2.1. The description of computer hardware in this section is based on Hennessy et al. [6].

2.2.1 Decoding instructions

Unlike people, processors are incapable of interpreting and understanding the meaning of given instructions. However, instructions can still be decoded from one unambiguous representation into a sequence of actions. They are received as machine code in form of binary strings. Different parts of these strings represent needed information to perform an action, as shown in Figure 2.3.

Another difference between a decoder and the reader in the metaphor is the

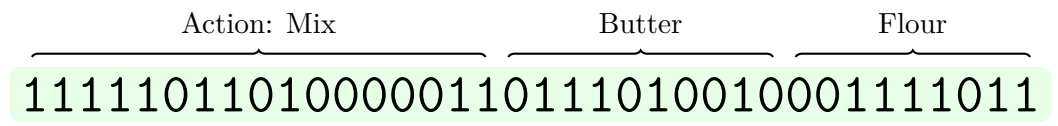

Figure 2.3: *An example of an instruction format in machine code. One binary substring is used to signify which action to take, while the rest of the binary string is used to specify the input operands. Instructions are encoded similarly in machine code (although cooking instructions are not executed in computers).*

use of two inputs rather than one. The decoder is given a buffer of instructions, similarly to the reader. The second input marks the position, or address in the machine code where the next instruction resides. Results from performing the actions corresponding to instructions may affect the address of the next instruction to be decoded.

2.2.2 Execution of instructions

Instructions represent sequences of actions carried out on temporary results to create new results, similarly to cooking instructions. One difference however, is the granularity of the actions. For instance, the third cooking instruction in Figure 2.1 involved mixing flour and melted butter. A processor would need to handle this task in smaller bites. Below are three instructions collectively achieving the same results in steps more suitable for a processor:

1. Move flour into bowl A.
2. Move butter into bowl B.
3. Add bowl A and bowl B into bowl C.

Processors require the input of instructions, known as operands, to explicitly declare storage for temporary results. Values (ingredients) are put into some form of storage (bowls) and are referred to in operations. As shown in the first two of the three instructions above, instructions may also take constants (flour and butter) as operands.

Processors do not perform computations using physical ingredients. Values are instead stored identically to the instructions found in machine code.

Section 2.2.1 described how these binary strings could represent actions, but binary strings can in fact represent nearly anything. They can represent integers, characters and even colors. The processor cannot interpret their meaning more than what is necessary to carry out computations. Whether the contents of storage A combined with storage B denotes adding 13 with 4 to create 17, or represents adding blue with red to create purple is irrelevant to the processor. It is up to the author of the recipes to keep track of what these values stand for.

2.2.3 State

Rather than bowls, pots, and pans, processors store results in a memory hierarchy. The top of the memory hierarchy is made up of registers, providing very limited storage but allow the processor to retrieve and store values extremely fast. Memory further down the hierarchy will compromise speed of usage for storage size and is called primary memory. Registers and primary memory are all considered volatile because the stored results are lost when the computer shuts down. Primary memory and memory components further down the hierarchy are not considered part of a processor in practice. Memory types further down in the hierarchy allow persistent storage, but are beyond the scope of this thesis.

2.2.4 Classes of instructions

Instructions in a processor are often separated into three categories:

Arithmetic and logic instructions These instructions take constants and registers as input and perform some operation, such as multiplication or logical comparisons.

Move instructions Values sometimes need to be moved between registers or between registers and primary memory locations, which is carried out by these instructions.

Control flow instructions To allow execution to continue from another position in the buffer of machine code, instructions of this type allow the

address of successive instructions to be changed, either unconditionally or based on some logic computation.

Instructions from different categories will use different resources in the processor when being executed, as explained in the next section.

2.2.5 Functional units

Figure 2.2 shows that instructions are moved from a pipeline into an executor component, performing the corresponding operations. In reality, this executor is composed of many functional units capable of carrying out needed actions. A cooking analogue to a functional unit is a stove or a mixer, used by the actor for some specific demand.

Processors have at least one a unit performing integer arithmetic and another logical operations. Other units perform actions required by control-flow instructions and move instructions. The need for this extra level of detail is motivated in Section 2.2.6, where dependencies between instructions are depict.

2.2.6 Pipeline

Instead of a queue, processors issue instructions into a structure known as a pipeline. A major difference between a pipeline and a queue is instructions do not only wait until they are executed. Minor tasks are performed as instructions progress through the pipeline, even after they finish executing.

Some processors are sophisticated enough to resolve dependencies between instructions during execution to achieve better performance. For example, instructions are reordered, and executed on functional units in parallel, similarly to a skillful cook in the metaphor. Other processors are made up of simpler hardware, relying on an input of instructions for which all dependencies have already been resolved.

In the cooking metaphor, dependencies between cooking instructions are defined as results from one instruction needing to be completed before being

input in successive cooking instructions. Dependencies create unwanted situations, known as hazards in processors. These hazards exist in three shapes:

Data hazards These correspond to the instruction dependencies in the cooking metaphor, where results from instructions needed in other instructions are not yet ready.

Structural hazards The finite amount of resources in a processor, in particular functional units, limits the rate at which instructions are executed, creating these hazards.

Control hazards Instructions that change the control-flow of execution create pipeline issues because when they finish executing, the pipeline will contain successive instructions that are not meant to be executed.

To avoid issues with hazards being unsolvable during execution, processors require delay slots. Delay slots are positions in machine code that need to be filled with instructions (there cannot be empty fields in machine code and instructions are decoded at a fixed rate), but where no useful instruction fits. Processors fill these delay slots with so-called no operation (or NOP) instructions that do not perform any actions. The notion of a delay slot is needed for the evaluation of Instant Simulator in Chapter 6.

The depth of the pipeline, along with the small operations performed in each step and the hazards that can occur, are collectively known as the pipeline behavior.

2.2.7 Steps of Execution

Unlike a processor, cooks will read, plan, and execute cooking instructions in continuous time steps. Processors, being less spontaneous, execute the necessary steps of execution in (more or less) fixed time steps known as cycles. Ideally in a single cycle, a processor will decode an instruction and issue it into the pipeline, while letting all instructions currently residing in the pipeline progress one step further, executing one or more of them. Unfortunately, this view is oversimplified and naive for our purposes of understanding processors and simulators. Instructions tend to require multiple cycles to finish executing

and unresolved hazards force other instructions to wait until their input state is ready.

2.3 Simulating Processors

Not only does our model of a processor, depict in Section 2.2, share similarities with someone preparing a meal, but also with a typical software program. Indeed, software programs are often created to receive a form of input and perform computations. Figure 2.4 shows how a simulator is executed like any other program. Essentially, all that is needed to create a simulator is to define functionality for decoding instructions of machine code and how they update program state.

One problem when simulating a computer is the extra computation needed when a machine executes software intending to imitate a machine. Another issue is the fact that capturing every little behavioral and physical detail of a processor results in an extremely complex system which is redundant for many purposes of simulation. Trade-offs are considered in their designs, influenced by the intended usage. Even the brief overview of a processor in Section 2.2 is more detailed than necessary of some purposes.

Having established that the level of realism in a simulator is coupled to a specific purpose, the intended usage of Instant Simulator is in need of being recounted for in order to motivate its design, which was covered in Section 1.1.

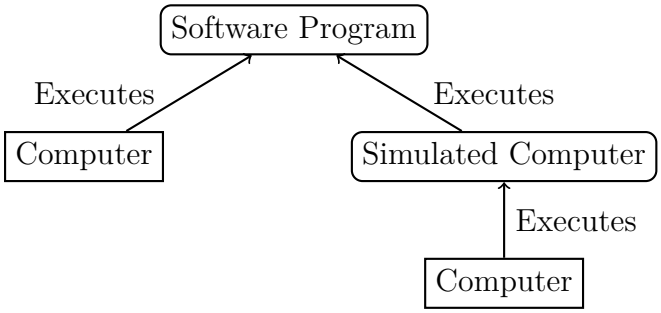

Figure 2.4: *Executing software versus simulating execution of software. Simulating a computer is no different from executing a software program. Software programs are indicated by the rounded rectangles and physical machines have rectangles with hard edges*

2.3.1 Simulation with support for multiple processors

Developing a simulator supporting multiple processors requires software reuse to avoid large amount of duplication program code. The possibilities of avoiding repetition are found by observing our current model of a processor in Figure 2.2.

Our current model of a processor contains processor-specific components: The decoder, pipeline and executor. These components collaborate with each other, maintaining processor-specific state and is thus changed between simulators of different processors.

How these components collaborate with each other, indicated by the directed paths between them, is independent of the specific details of a processor for our purposes of simulation. The wiring between components may remain unchanged between different simulators, thus allowing us to implement a generic version of a simulator. This generic simulator can then be extended with the following needed processor-specific details:

- How state is defined.
- How instructions of are decoded.
- The semantics (actions) of the instructions.
- The pipeline behavior.

Instead of interweaving processor-dependent components with the generic parts of the simulator, a clear separation in the design reduces the problematic of replacing components when supporting another processor. Figure 2.5 shows how the generic parts are separated from the processor specific components.

The process of supporting multiple processors can be simplified further. While our model of a processor contains a handful of details, in reality, processors support hundreds of different instructions, maintaining state in complex memory hierarchies. These highly repetitive and fine-grained details of a processor can be specified using a higher-level language, known as an architecture description language (ADL). Descriptions are then evaluated by a tool in order to generate source code compatible with the generic components of the simulator.

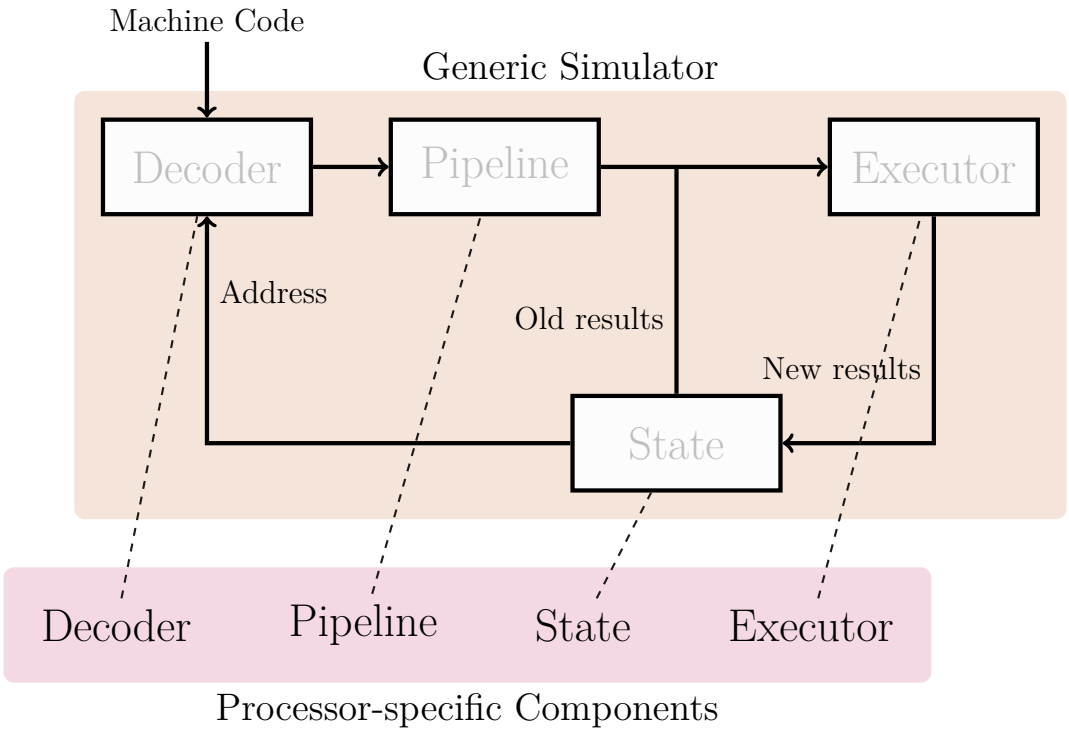

Figure 2.5: *The separation of the processor-specific parts and the generic parts of a simulator. This figure is based on Figure 2.2, where the two parts are combined.*

2.3.2 Simulation based on an ADL

The purpose of an ADL is to help describe a system (a processor in our case) to some desired level of detail [2]. These machine-readable descriptions follow grammatical and semantic rules, allowing them to be transformed into a form compatible with generic parts of various tools.

Section 2.3.1 summarized all the processor-specific parts of a typical simulator based on our current model, which an ADL must allow a processor-description to express in order to create simulators. All possible instructions combined with all allowed operands must map to unique binary strings in order for a decoder component to function. Possible hazards will also need to be described, along with instructions passage through the pipeline. Each instructions must map to a sequence of actions to perform on input. how detailed this information is depends on the purpose of the tool to be generated.

Separating processor-dependent parts from generic parts, not only in

design but also in use of language, comes at an initial cost. The generic part of simulator has to be developed together with an ADL and a tool capable of evaluating this ADL to generate compatible source code. However, the amortized cost of creating simulators for an increasing number of processors is reduced over time, compared to manually rewriting the fine-grained repetitive details in source code.

2.4 Instant Toolsmith ADL

Instant Simulator follows the design principles outlined in Section 2.3.1, allowing descriptions of processors for generating the necessary functionality. These descriptions are written in Instant Toolsmith ADL, documented in greater detail in Appendix A. This section aims to briefly show how the different parts of a processor are described in Instant Toolsmith ADL with the help of an example in Listing 2.1.

Instant Toolsmith ADL is implemented as a subset of the programming language Scheme [12], where source code is structured by the (excessive) use of parentheses. Listing 2.1 shows an example, which begins by defining state in form of registers, capable of storing binary strings with a maximum bit width of 32 bits.

The second part of Listing 2.1 defines an instruction add-regs, capable of performing addition on values stored in registers. This instruction requires 32

```
(define-reg reg1 (size 32))
(define-reg reg2 (size 32))
(define-reg reg3 (size 32))
...

(define-instruction add-regs
  ((size 32)
   (functional-unit (arith1) (arith2))
   (operands (reg dst r32) (reg src1 r32) (reg src2 r32))
   (encoding P (22 #b11110000100110011010) src1 src2 dst)
   (semantics (set-reg! dst (add 32 src1 src2)))))
```

Listing 2.1: *An example of a description in Instant Toolsmith ADL.*

bits to be represented in machine code, declared with the size attribute. It can be executed on one of two functional units, arith1 and arith2, described with the functional-unit attribute. It takes three registers as input (identified as src1, src2 and dst), storing the sum of two registers into the third register, indicated by the operands and semantics attributes. Which registers are used, is provided in the encoding attribute together with a constant binary string with a bit width of 22 bits. A P bit, explained in Section 2.5.1, is also provided in the encoding of the instruction.

Functional units carry out execution in a real processor according to Section 2.2.5. However, they are only included in Instant Toolsmith ADL to allow structural hazards to be detected. There is no way to describe the details of a functional unit besides a unique name.

Listing 2.1 captures much of the needed information to generate the processor-specific components. The definition of registers allow state to be generated. The encoding attribute and the size attribute in instructions are evaluated to generate the decoder. The semantics describes the actions taken when executing an instruction, needed for an executor component.

Unfortunately, the current description of the instruction lacks the necessary details for describing its pipeline behavior. Structural hazards are described explicitly, and data-dependency hazards can be implied, but there is currently no notion of control flow hazards. Moreover, there is no way of describing which hazards are resolved by the processor, among other needed details. Chapter 3 expands on this issue further, due to its influence on the design of Instant Simulator.

2.5 Digital Signal Processors

Instant Simulator supports a specific class of processors, known as digital signal processors, described in detail by Tan et al. [14]. Informally, a digital signal processor (DSP) can be characterized as means for carrying out numerically intensive and repetitive tasks for which there are real-time constraints. The reasoning behind this choice is that the parts of these processors (relevant to Instant Simulator) are simpler than what is found in a general purpose processor (GPP). A real example of a DSP is characterized in Section 6.2.

2.5.1 Very long instruction word

Section 2.2.6 briefly touched upon the notion of instruction-level parallelism (ILP) by mentioning how independent instructions in the pipeline are reordered and executed on functional units in parallel, to achieve better performance. DSPs also execute instructions in parallel on functional units, but will typically not reorder them due the high cost of power usage. The truly simple DSPs cannot even handle any of the three types of hazards categorized in Section 2.2.6.

DSPs achieve ILP with a technique known as very long instruction word (VLIW). Rather than decoding one instruction at a time, bundles (also known as packets) of instructions are decoded. Multiple instructions in the same bundle progress through the pipeline and execute in parallel.

VLIW requires machine code expressing bundles of instructions. Boundaries between bundles are set by letting the first bit of an instruction encoding signify whether the instructions is bundled with the previous adjacent instruction. Another variant is letting the last bit of the instruction encoding signify whether the instruction is to be bundled with the adjacent successive instruction. This indicator bit is often referred to as the P bit, where P stands for packet.

2.5.2 Conditional execution

Instructions that might change control flow, depending on the current state of computation, result in unwanted control hazards. GPPs reduce their presence by predicting whether the control flow is changed and then continues execution according to the prediction. Performance is not hurt by control flow instructions when the processors predicts the outcome correctly. If the prediction is wrong however, the results from the speculative execution are discarded and loss in performance becomes inevitable.

DSPs abilities to handle control flow instructions tend to be less sophisticated compared to GPPs. DSPs make use of a technique known as predicated execution [10], also known as conditional execution (GPPs also use this technique to a lesser extent). With conditional execution, instructions may have predicates attached to them placed adjacently in

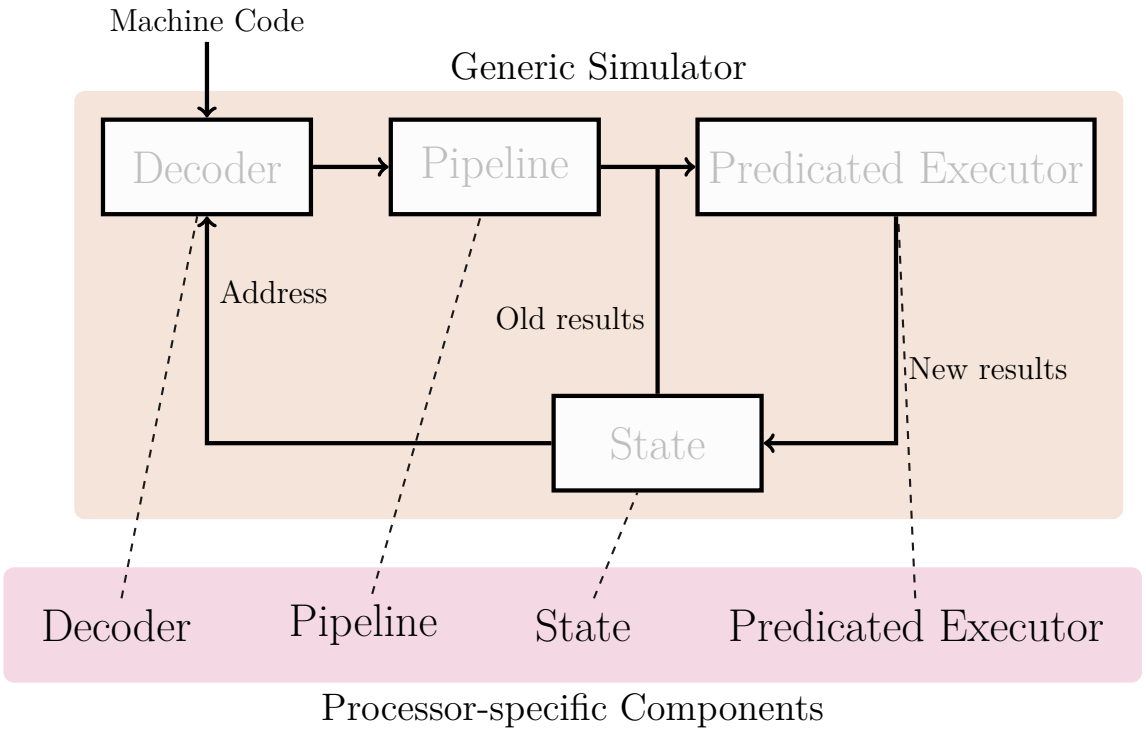

Figure 2.6: *A model of a simulator supporting predicated execution of instructions. This figure is an extension to Figure 2.5, with the use of a predicate executor component instead of an executor component, to indicate that instructions can be executed conditionally.*

machine code, mapping to operations that check the value of a specific register. These predicates will not decide if an instruction is to be executed, but whether the results of the instruction should be saved or discarded. While executing redundant instructions is wasteful, the effect control hazards have on performance is even worse [3].

Our current model of a processor, outlined in Figure 2.2, does not capture the use of conditional execution, which is needed when this model used to present the design of Instant Simulator. The model is therefore extended to use a predicated executor instead of a executor, shown in Figure 2.6. Conditions are described similarly to instructions in Instant Toolsmith ADL, documented in Section A.3.

2.6 Related Work in Architecture Description Languages

This section explores relevant work in creating ADLs for simulation purposes. As the main focus is on generating simulators, only the ADLs that are relevant to Instant Simulator are covered.

2.6.1 Expression

In the development of Expression [5], the authors recognized the need for System-on-Chip designers to both rapidly explore hardware options and accurately benchmark systems. Hence, the goal has been to cover both the physical and behavioral attributes of processor descriptions in their ADL. Furthermore, the description must not only be machine-readable but also human readable. Expression therefore uses a friendly LISP-like structure, very similar to Instant Toolsmith ADL. Another similarity to Instant Toolsmith ADL is the target hardware. Expression is suited for DSP hardware and supports VLIW.

Expression's physical description of processors is overly detailed for the purposes of this thesis. This is evident in the way pipelines are specified by the use of ports, latches and signal structures. Expression could provide a basis for future work, if Instant Toolsmith ADL is to be extended to include a more structural description.

2.6.2 Harmless

Harmless [8] is an ADL designed specifically for generating simulators. The simulators can be used to cycle-accurately benchmark software programs and to ensure their correctness.

The simulated pipelines are built statically as finite state machines, optimized by categorizing instructions into more classes than found in Section 2.2.4. Unlike with dynamically managed pipelines, used by Instant

Simulator (described in Chapter 4), the authors claim there is less overhead during execution.

Harmless targets embedded hardware with less complex pipeline structures (compared to GPPs), hence the focus is put on performance rather target-hardware scope. Describing deeply pipelined processors that utilize heavy instruction-level parallelism could lead to a combinatorial explosion in the used optimization algorithms. The authors provide a way to avoid this by the use of less intuitive descriptions of the pipeline behavior.

Even though VLIW is not supported, the pipeline design in Harmless should be examined further once Instant Toolsmith ADL becomes capable of describing a pipeline.

2.6.3 MADL

Mescal Architecture Description Language (MADL) [11] is an attempt to automatically generate quality simulators and compiler tools without sacrificing the range of supported hardware types. MADL incorporates heavily abstracted structures called operation stack machines (OSM). By using an OSM, the execution flow is described using different (pipeline) states where different resources are allocated and released. Expressing resource usage helps generating constraints which then enables static generation of pipeline hazards. Moreover, this can be done at simulator compile time, resulting in a performance boost. Although a statically generated pipeline is beyond the scope of this thesis, the resource acquisition is used in the naive pipeline of Instant Simulator, covered in Section 4.3.

Chapter 3

System Design

This chapter provides an overview of the design of Instant Simulator. Chapter 2 outlined a model for a DSP and described how it is simulated in software. Usage of an ADL was motivated as a way of reducing implementation complexity and code duplication when supporting simulation of multiple processors. These concepts are now put to use in the design and implementation of Instant Simulator.

3.1 System Overview

Instant Simulator achieves simulation supporting multiple DSPs, following the design principles mentioned in Section 2.3.2, splitting the tool into two parts. Figure 3.1 outlines the design, which consists of the following two parts:

1. A generic simulator that actively drives execution, but lacks the necessary processor-specific details to simulate a concrete DSP.
2. A group of build scripts evaluating descriptions formalized in Instant Toolsmith ADL, generating source code for the processor-specific components. These components are reactive since their functionality is requested when needed by the generic simulator.

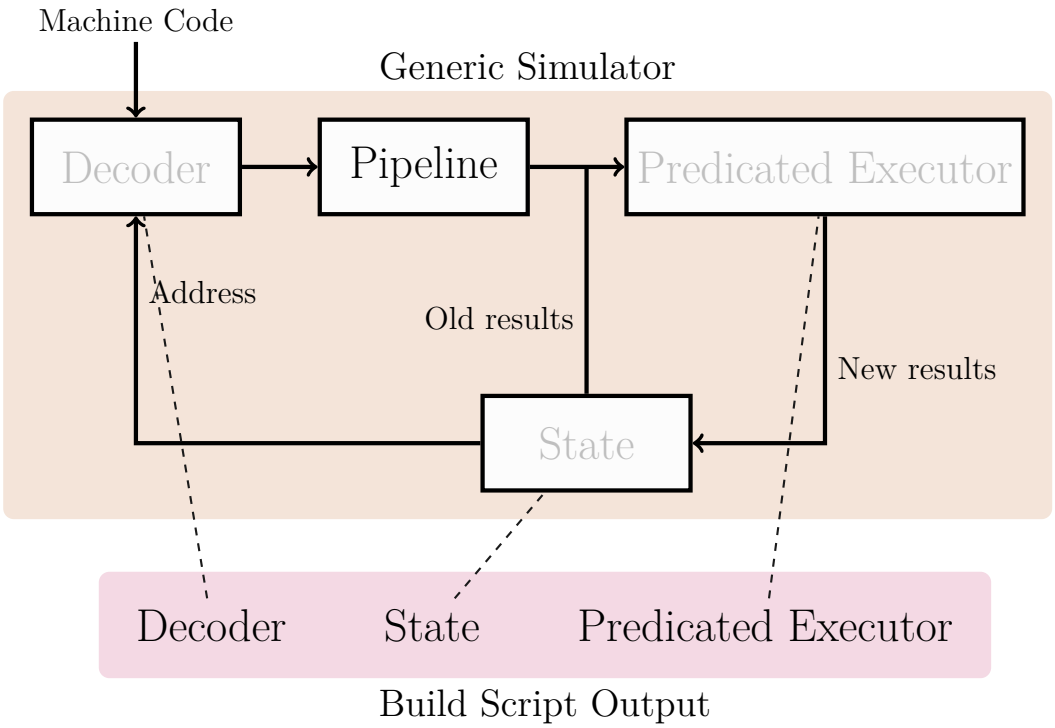

Figure 3.1: *The design of Instant Simulator. The generic simulator provides general functionality and the build scripts generate the processor-specific components. These components are plugged into the generic simulator to get a complete simulator.*

Once the source code that make up processor-specific components have been generated, they are plugged into the generic simulator. The notion of Instant Simulator being a simulator is incorrect, because it is a tool for generating simulators.

3.2 Generic Simulator

The first part of Instant Simulator is the generic simulator, covered in detail in Chapter 4. The purpose of the generic simulator is to drive simulation in each cycle in a sequence of steps:

1. Creating bundles filled with instructions and conditions using the decoder component. Results from the decoder component contain

necessary information for instructions and conditions to progress through the pipeline and for the correct operations to be carried out.

2. Issuing bundles into the pipeline, maintaining their progress with the help from the information given by the decoder. The pipeline was considered a processor-specific component in Figure 2.2, but the design of Instant Simulator takes a different approach. the generic simulator provides a pipeline (as shown in Figure 3.1) in form of an interpreter, maintaining the progress of instructions. The generic simulator does not require the use of a pipeline for generated simulators, thus allowing the paths of instructions through the pipeline to be omitted in simulation.

3. Allowing conditions and instructions in the pipeline to be executed when ready by requesting actions from the predicated executor component. This component is given the relevant state to update along with the necessary information about the instruction (or condition).

Section 2.2.1 mentioned that a processor does not understand the meaning of machine instructions and blindly maps them to actions. Furthermore, Section 2.2.2 stated that a processor is indifferent to what input values and computed results truly represent in a bigger picture. The same principles apply to the generic simulator.

The information of decoded instructions and conditions, given from the decoder, does not include the actions required to carry out execution. Instructions are instead uniquely identified by a number, which is mapped to their semantics within the predicated executor. This leap of faith, where a decoded instruction is assumed to map to actions, allows the generic simulator to only focus on driving simulation and ignore the concrete details of how execution is accomplished.

3.3 Build Scripts

The second part of Instant Simulator is made up of scripts, described in Chapter 5, generating source code for the processor-specific components. The generic simulator performs actions demanding functionality from the following components, generated by the build scripts, also shown in Figure 3.1:

- State made up of registers, defined using data structures declared in the generic simulator, described in Section 5.1.
- A predicated executor, provided with the current state and needed operands, executing an instruction to update state or checking a condition. How this component is generated is covered in Section 5.2.
- A decoder capable of receiving a buffer of machine code and a position, returning the next instruction or condition, covered in Section 5.4. As mentioned in Section 3.2, the results must capture enough information to allow instructions to progress through the pipeline and be executed with the correct operands.

There is no way of describing pipeline behavior in Instant Toolsmith ADL, so the build scripts cannot generate the necessary information required by the pipeline in the generic simulator. Once Instant Toolsmith ADL can describe the pipeline of DSPs, the build scripts can be extended to generate pipeline behavior. In the case study in Chapter 6, Instant Toolsmith ADL and the build scripts are temporarily patched to generate the necessary pipeline information.

Chapter 4

The Generic Simulator in Detail

This chapter describes how the generic simulator, outlined in Chapter 3, carries out simulation by performing distinct steps of actions in each cycle of execution. All steps are detailed in the following sections, along with the communication with the processor-specific components, generated by the build scripts (Chapter 5). This chapter ends with an implementation-focused overview of the data structures, used within the generated simulators.

4.1 Executing a Cycle

The generic simulator performs execution in steps, mimicking an execution cycle performed by a real DSP. Figure 4.1 shows how these steps can be carried out in two different ways, depending on whether hazards exists within the pipeline:

1. If the pipeline is free of hazards, the next bundle of instructions and conditions are first decoded, covered in Section 4.2. This bundle is then issued into the pipeline before the pipeline is updated to let the residing instructions progress one step further. Pipeline management is outlined in Section 4.3.

2. If hazards do exists within the pipeline, only the pipeline will updated in a cycle of execution in an attempt resolve the issue. The pipeline will signal its stall status and no more bundles can be issued until the pipeline is free of hazards, which might be resolved in the next cycle or

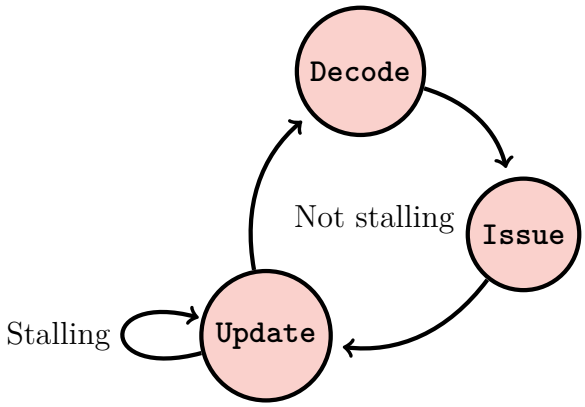

Figure 4.1: *The steps taken during execution of a pipelined simulator, which depends on whether the pipeline is stalling or not.*

after multiple cycles. How the pipeline detects and resolves hazards is also covered in Section 4.3.

4.2 Decoding Bundles of Instructions and Conditions

Bundles of instructions, some of which have conditions attached, are decoded by repeatedly querying the decoder component. The decoder is given a buffer of machine code and an address and it will return a structure containing relevant information about an instruction or a condition.

Boundaries between bundles are distinguished by a VLIW flag in the result, corresponding to the P bit explained in Section 2.5.1. The VLIW flag of an instruction either signifies that it is bundled with the adjacent successive instruction, or the previous adjacent instruction. Both ways are supported by Instant Simulator and decided upon when creating a simulator. Letting the VLIW flag signify the bundling of a previous instruction will however require the generic simulator to always decode one extra instruction for each bundle and then discard the result when the VLIW flag is not set.

When conditions are decoded, they are attached to a previous adjacent instruction if their VLIW flag is set. If the VLIW flag is not set, they will be attached to the whole bundle of instructions as a global condition.

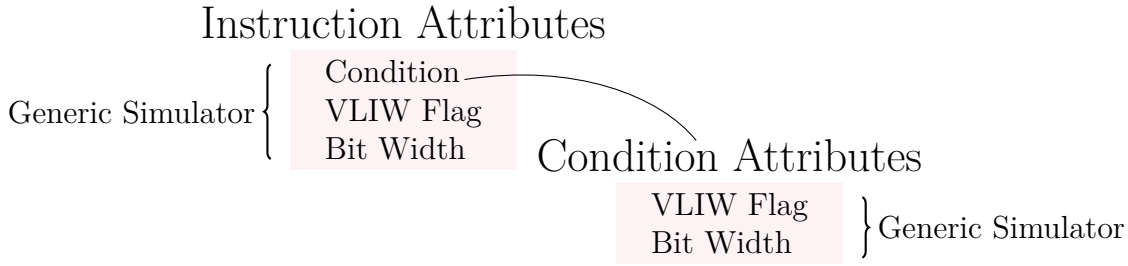

Figure 4.2: *The initial decoded instruction (and condition) information needed by the generic simulator*

More information is needed from the decoder result besides a VLIW flag. Once an instruction is decoded, the generic simulator will need to know the length of the binary substring in the machine code that represented the result. This length is used to calculate the position of the next piece do be decoded. Figure 4.2 shows the current information needed from the decoded result. Subsequent sections will further extend the fields needed in order for a simulator to work.

4.3 Maintaining Bundles in the Pipeline

Once a complete bundle has been decoded, the generic simulator will issue the bundle into the pipeline (unless the pipeline is stalling). In order for the pipeline to make use of the data representing instructions within the bundles, the data structures will need to include more fields than what is shown in Figure 4.2. These fields are now introduced in the rest of this section.

The pipeline is implemented as a small interpreter, receiving bundles of instructions containing paths in form of linked list, filled with operations on resources. The interpreter will traverse these paths, one step per cycle, and perform the operations stored in them. One example of an operation is the acquisition of a functional unit before execution. Figure 4.3 includes the path field needed by the pipeline, where these operations are stored.

What resources are acquired (and returned), at any point during execution, is maintained by the pipeline, thus allowing hazards to be detected. If a hazard is detected, for instance when a path of an instruction attempts to use a register as an input operand when another instruction has acquired the same

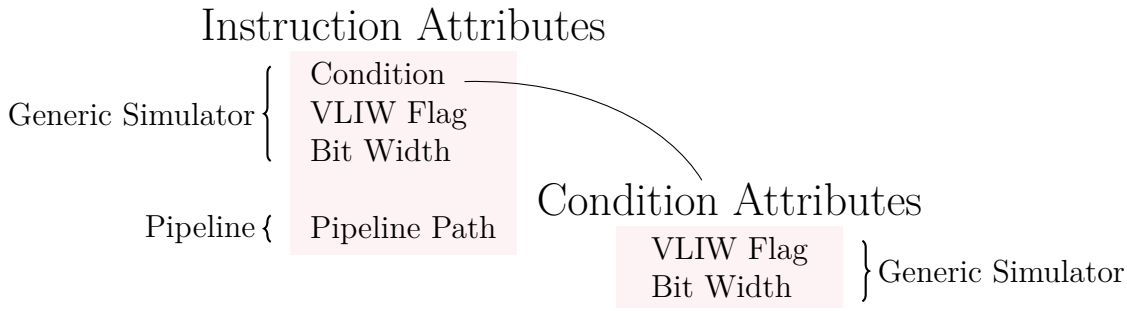

Figure 4.3: *An extension to the initial decoded information. The pipeline in the generic simulator needs the pipeline behavior of individual instructions.*

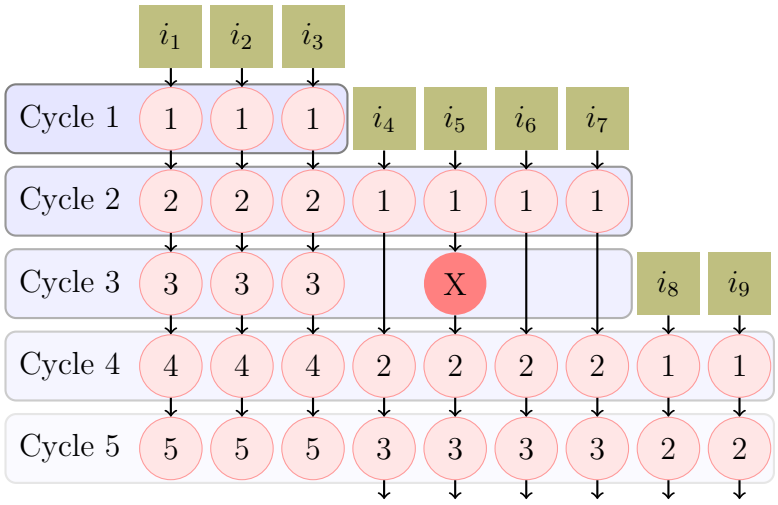

Figure 4.4: *Pipeline state during five cycles of execution, where three bundles and a total of nine instructions (squares) are issued. The pipeline updates path steps (circles) left to right (according to the figure). In the second bundle, the fifth instruction (i_5) causes the pipeline to stall one cycle. No instructions in that bundle, nor any succeeding instructions, will update state during stall cycles.*

register for updating state, the pipeline will stall. The path of the instruction where the hazard is found will not be updated in the pipeline during that cycle, along with all the instruction in the same bundle, as shown in Figure 4.4.

The following operations contained in the steps of the paths of instructions, are handled by the pipeline:

Nothing Path steps that do nothing are needed to simulate events the simulator does not really care about. For instance, fetching instructions is not really done from the pipeline, so the simulator does nothing instead.

Lock This operation provides mutually exclusive use of registers and functional units. Instructions that write to registers need to lock them so that data-dependency hazards are detected. Non pipelined functional units also need to be locked to allow for structural hazards. Already locked units and registers result in stall cycles.

Release Registers and functional units that have been locked need to be released when instructions have passed through the pipeline.

Read register When this operation is carried out, a read of a specific register is attempted. If the register is locked from a preceding operation, the pipeline stalls.

Execute This operation carries out the real execution of instructions. If an instruction contains a condition, the result of first executing the condition decides if the instruction is to be executed. Instructions within a bundle need not all do execution in the same cycle.

Lock pipeline Sometimes, DSPs require their pipeline to stall for various reasons. A classic example is when an DSP lacks speculative execution. A decoded conditional-jump instruction will then create a control hazard because the processor cannot feed the pipeline with more instructions until the result of the conditional-jump instruction is known.

Release pipeline This operation will allow the pipeline to be fed instructions again, after a forced stall.

Flush pipeline DSPs lacking branch prediction sometimes require the pipeline to be flushed after executing jump instructions. This operation removes all successor instructions in the pipeline.

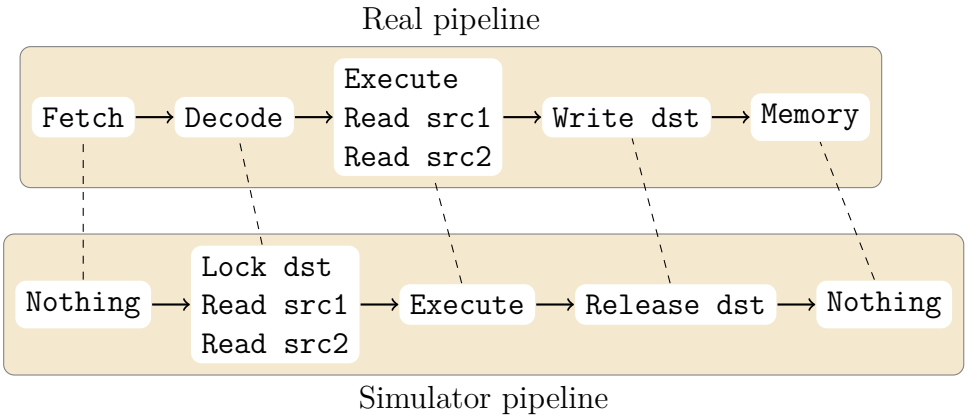

Figure 4.5: *The stages of a typical five-stage pipeline. The version at the top illustrates how a (very simple) pipeline is organized in hardware. The version beneath outlines how this pipeline would function in a simulator created by Instant Simulator. The dashed paths connect the corresponding stages of the pipelines.*

The pipeline in the generic simulator differs from what is seen in a real processor. Making the pipeline overly realistic would introduce redundancies (for example an explicit write-back stage). A more realistic description can still be expressed with an ADL and then compiled into the format used by the generic simulator.

To end this section, an example is now presented to show how different the pipeline is in the generic simulator compared to what is found in real hardware. Figure 4.5 shows two different versions of a simple and ideal five-stage pipeline model usually associated with a Mips architecture [6]. The version above is a typical illustration and the version below shows how it would be structured in a simulator generated by Instant Simulator.

4.4 Executing Instructions and Conditions

One of the operations used in the paths, interpreted by the pipeline, is the execution of an instruction. This operation is now covered in this section.

Before execution of an instruction takes place, a check is made to see whether a condition is attached to the instruction. If that is the case, the condition is provided to the predicated executor in a request to evaluate the

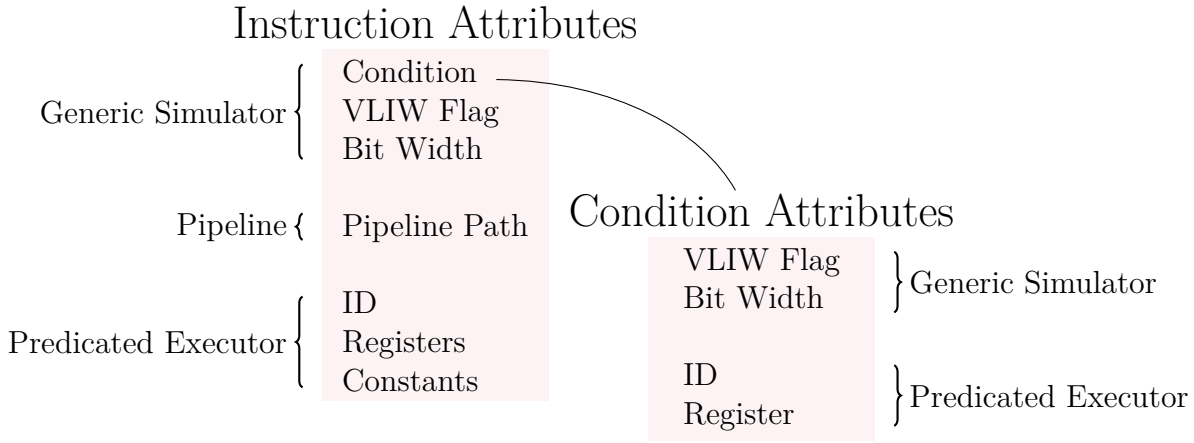

Figure 4.6: *All of the decoded information needed for simulation. The predicated executor requires a unique ID to map to actions and input operands for updating state.*

condition. The current fields in the structure representing a condition, shown in Figure 4.3, cannot help identify which condition is to be evaluated. The structure will therefore need to include a unique ID number for each condition. Furthermore, conditions are evaluated by using the content of a register which must also be provided. Figure 4.6 shows extended condition structure with the additional attributes.

Result given from evaluating a condition help the generic simulator decide whether the instruction the condition belongs should be carried out. Section 2.5.2 mentioned that in real DSPs, instructions will always be executed and the condition decides whether the results will be discarded. The generic simulator will instead skip the execution of instructions if the predicate turns out to be false.

If the predicated executor returns a positive value, the instruction will be executed. Similarly to the condition, the instruction data structure will need to be extended with an ID and operands. Instructions allow multiple registers as operands, unlike conditions, along with multiple integer constants. These fields are added to the extended data structure for an instruction, as shown in Figure 4.6.

Instructions are executed by being given to the predicated executor, together with the current state of the simulator, in a request by the generic simulator to update state. The current state is needed because some

instructions modify more state beyond the provided operands (for example, move instructions load data from primary memory). The actions performed within the predicated executor is beyond the scope of the generic simulator and instead covered by the build scripts in Chapter 5.

4.5 Data Structures

The generic simulator provides a set of data structures for storing information and communicating with the processor-specific components. An example of such information is the data structures for instructions and conditions in Figure 4.6. These structures are utilized both internally within the generic simulator and by the processor-specific components generated by the build scripts.

Concrete implementation details will now be in focus, unlike in previous language-agnostic sections. The rest of this section presents relevant data structures using C code, the programming language simulators are generated in by Instant Simulator.

4.5.1 Registers

Registers are represented with a simple data structure containing information about the bit width, a current value and a unique identifier. Figure 4.7 shows a very basic register hierarchy and Listing 4.1 presents the constructed result. The overlaps between the registers, documented in Section A.1.4, are not captured by the data structure in the generic simulator. Instead, this is managed by register functions, which are generated by the build scripts (described in Section 5.3).

4.5.2 Functional units

Functional units are made up of the least complicated data structure in the generic simulator. They only need a unique (numeric) identifier for representation. Their purpose is not to simulate execution of operations.

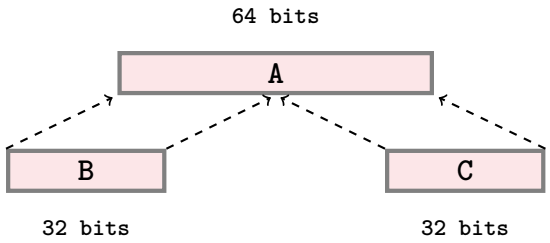

Figure 4.7: *The register hierarchy of the three registers created in code listing 4.1, one 64 bit register A and two overlapping 32 bit registers B and C. The dashed arrows indicate the sections where an overlapping register shares data with a base register.*

```
1  sim_register_t* a_64 = alloc_reg(64);
2  sim_register_t* b_32 = alloc_reg(32);
3  sim_register_t* c_32 = alloc_reg(32);
4
5  a_64->id = 0;
6  b_32->id = 1;
7  c_32->id = 2;
```

Listing 4.1: *Construction of three registers from the hierarchy shown in Figure 4.7. 32 bit registers B and C both overlap 64 bit register A, but this information is lost.*

```
1  /* the literal parameter 1 represents
2  the identity of the allocated unit */
3  sim_unit_t* MAC = alloc_unit(1);
```

Listing 4.2: *The uneventful allocation of a functional unit, to be identified by the value 1.*

Instead, they are used by the pipeline to account for structural hazards. The generic simulator provides means for declaring a unit type that, besides including an identifier, could include run time statistics in future extensions of the tool. Listing 4.2 shows how a unit is allocated.

```
1  instr_info_t* instr = alloc_instr();
2
3  // Needed by Predicated Executor to map to the
4  // correct actions.
5  instr->id = ADDI;
6
7  // This instruction took 32 bits to decode.
8  instr->size = 32;
9
10 // Operands for Predicated Executor
11 instr->regs[0] = state->regs[0];
12 instr->regs[1] = state->regs[1];
13 instr->imms[0] = 42;
14
15 // Flag for generic simulator
16 instr->vliw = SIM_TRUE;
```

Listing 4.3: *The construction of an instruction performing addition on a register and an immediate value.*

4.5.3 Instructions

Throughout this chapter, the needed fields in the data structure representing instructions have been extended to cover the needs of different parts of a simulator, fully shown in Figure 4.6. Listing 4.3 shows how some of this information is stored in C for a simple add immediate instruction. The pipeline path of the instruction is omitted to avoid clutter. The data structures that make up these paths are not implemented graciously, because they are intended to be utilized via generated code rather than hand-written code.

4.5.4 Primary memory

Although a complete memory model is beyond the current scope of Instant Simulator, the effort required to implement primary memory is small enough to include it. The simulator state contains a reference field which can be used to manually allocate a chunk of data. By referencing a memory area from the simulator state structure, instructions (which are passed the simulator state during execution) can manipulate this area and implement the processor stack

behavior. This is is done in the case study in chapter 6.

4.5.5 Memory management of data structures

Internally, simulators will dynamically allocate and deallocate memory for the data structures presented in this section (and other internal structures as well), repeatedly during execution. A minor optimization has been made in the generic simulator to reduce memory allocation overhead.

All data structures are created and destroyed in a memory-management layer via allocation- and deallocation functions. In the current version of Instant Simulator, this layer utilizes an array-based free list to avoid constant invocations of system calls. Test results show a significant speedup for a small development effort.

Chapter 5

The Build Scripts in Detail

This chapter describes how the build scripts of Instant Simulator generate the missing processor-specific components, outlined in Chapter 3. Descriptions of DSPs, formalized in Instant Toolsmith ADL (briefly presented in 2.4 and documented in Appendix A) are provided to a group of scripts. These scripts evaluate the content and generate the needed components in compatible C source code. The following components are generated by the build scripts:

- Simulator state (Section 5.1).
- A predicated executor (Section 5.2).
- A decoder (Section 5.4).

The build scripts will also generate smaller functions to simplify the implementation details of the generated components. One example is a group of register functions, described in Section 5.3, for updating the values in registers while ensuring consistency among register overlaps.

5.1 Building Simulator State

The build scripts need to initiate the data structures that contain the state in the simulator so that state can be updated when executing instructions. Furthermore, the state will be used by the pipeline to detect hazards (covered in Section 4.3).

```
1 #define SIM_NUM_REGS ...
2 #define SIM_NUM_UNITS ...
3
4 sim_state_t* get_state()
5 {
6   sim_state_t* state = alloc_state(get_registers(),
7                                    SIM_NUM_REGS,
8                                    get_units(),
9                                    SIM_NUM_UNITS,
10                                   alloc_pipeline());
11  return state;
12 }
```

Listing 5.1: *The generated source code for creating simulator state. Constants SIM_NUM_REGS and SIM_NUM_UNITS are extracted from the specification. The pipeline is allocated via the generic simulator directly because no processor-specific initialization is needed*

The only two parts requiring processor-specific initialization are the registers and the functional units (the initial construction of a pipeline does not rely on a specification). Registers and functional are evaluated in a specification documented in Section A.1 and Section A.4.5 respectively, and then transformed into C source code, as described in Section 4.5.

Once the C source code for creating registers and functional units is generated by a build script, the last step is to generate a procedure linking everything together and constructing the state. This procedure is shown in Listing 5.1.

5.2 Building Executable Instructions and Conditions

This section describes how the predicated executor (outlined in Chapter 3) is generated into C source code, capable of carrying out the actions of instructions and conditions. The semantics of instructions and conditions are transformed from a Instant Toolsmith ADL specification into C source code in a handful of stages. Once all instructions and conditions have had their

```
(define-util-fun (update-msb-flag! reg)
  "updates the most-significant-bit flag of
  the provided 32 bit register"
  (set-reg! (reg-ref reg 'msb) (icmp 32 'slt reg  0)))

(define-instruction MAX
  ((operands (reg rs r32) (reg rt r32) (reg rd r32))
   (semantics
     (begin
      (set-reg! rd (rif (icmp 32 'sgt rs rt)
                        rs
                        rt))
      (update-msb-flag! rd)))))
```

Listing 5.2: *A made-up instruction which serves an example throughout the stages of transformation described in this section. The instruction stores the largest value out of two registers into a target register. A flag that signifies the most significant bit of the target register is then updated.*

semantics transformed into source code compatible with the generic simulator, they are wrapped into a function that maps the ID numbers of instructions and conditions to the correct actions.

To gain a better understanding of the incremental transformation of instruction semantics, a made-up instruction is used as an example. The relevant parts of the specification for this instruction is found in Listing 5.2. Even though the instruction itself might seem contrived, it is designed to make the most use out of the transformation stages while minimizing the amount of generated source code.

When generating executable conditions, the same principles apply as when generating executable instructions. The only difference is that the semantics found in conditions return a boolean value. Conditions are therefore omitted in this section to avoid repetition.

```
(define-instruction MAX
  ((operands (reg rs r32) (reg rt r32) (reg rd r32))
   (semantics
    (begin
      (set-reg! rd (rif (icmp 32 'sgt rs rt)
                        rs
                        rt))
      ;; The utility function is now inlined into the
      ;; semantics. A let form takes care of the
      ;; argument binding.
      (let ((reg rd))
        (set-reg! (reg-ref reg 'msb)
                  (icmp 32 'slt reg 0))))))
```

Listing 5.3: *First transformation (out of four). The semantics of the example-instruction (defined in Listing 5.2) after the utility function has been inlined.*

5.2.1 Inlined semantics

Before transforming instruction semantics, described in Instant Toolsmith ADL, some pre-processing is carried out. Each instruction and utility function will have their semantics updated, with calls to macro functions and utility functions (described in Section A.4) in-lined. Instant Toolsmith ADL disallows any type of recursion, thus making this step possible by simply traversing the semantics and applying the substitution model. After having a utility function in-lined in its semantics, the new version example instruction is shown in Listing 5.3.

5.2.2 Simplified static single assignment form

In this stage of source code transformation, the instruction semantics are flattened into a simpler version of the static single assignment form. Temporaries are introduced for variables that are not registers.

The result after this transformation is found in Listing 5.4. To characterize the source code as being flattened might seem ironic. This form is best suited for a more procedural language, for example the intermediate representation

```
(define-instruction MAX
  ((operands (reg rs r32) (reg rt r32) (reg rd r32))
   (semantics
    (begin
      (let ((tmp1 rd)
            ;; Find the largest register out of rt and
               rs.
            (tmp2 (let ((tmp3 (let ((tmp4 32)
                                    (tmp5 'sgt)
                                    (tmp6 rs)
                                    (tmp7 rt))
                                (icmp tmp4 tmp5 tmp6
                                   tmp7))))
                    (rif tmp3 rs rt))))
        ;; Store largest register in target register rd.
        (set-reg! tmp1 tmp2))

      ;; Update flag of target register rd omitted for
         clarity
      ... ))))
```

Listing 5.4: *Second transformation (out of four). The semantics for the example-instruction after being flattened. The source code created in this pass still contains valid Instant Toolsmith ADL expressions. The flag-update is omitted in the example to minimize source code size.*

used in LLVM [9], a just-in-time compiler framework.

5.2.3 Statically typed imperative representation

Once the semantics are flattened and filled with temporaries, an environment is built by traversing the expressions while storing bindings in a key-value structure. The usage of this environment is motivated below.

The source code is traversed yet again, but this time following a bottom-up approach. Expressions that update state are searched for. These expression include register updates, memory stores and control-flow updates. When the relevant expressions are found, dependencies for the temporaries used in these expressions are recursively added before including the state-modifying

```
;; Store the max of rs and rt into rd
(if (number (icmp (number 32)
                  (symbol sgt)
                  (register rs)
                  (register rt)))
    (then (set-reg (register rd) (register rs)))
    (else (set-reg (register rd) (register rt))))
 ;; Update the most-significant bit flag for register rd.
 (set-reg-ref (register rd)
              (symbol msb)
              (number (icmp (number 32)
                            (symbol slt)
                            (register rd)
                            (number 0))))
```

Listing 5.5: *Third transformation (out of four) which is the final transformation before C source code is generated. This representation is closer to C than Instant Toolsmith ADL. The if-statement has been lifted out from the second argument of the set-reg statement to top-level.*

expression. To find these dependencies, the pre-built environment is put to use.

Listing 5.5 presents the resulting representation of the semantics. This new representation includes type information, needed for the next stage, where C source code finally is generated. The current representation includes few constructs allowing for simple interpretation of the values. For instance, when the second operand of a *set-reg* construct is of the type register, the actual value is extracted rather than a reference to the register itself.

Unlike Instant Toolsmith ADL (and functional languages in general), C cannot make use of if-statements containing blocks of statements anywhere. If-statements are therefore lifted to the current block-level in the source code.

5.2.4 C source code

The final transformation step creates C source code. The final version of the instruction used as an example is shown in Listing 5.6. Any redundancies and source code duplication are left to the mercy of a C compiler.

```
1 void instr0(sim_state_t* state,uint64_t
    old_pc,instr_info_t* instr) {
2   sim_register_t* rd = instr->regs[0];
3   sim_register_t* rt = instr->regs[1];
4   sim_register_t* rs = instr->regs[2];
5
6   if(llvm_icmp(32, SIM_SYMBOL_sgt, rs->value, rd->value))
7   {
8     set_register(state,rd,rs->value);
9     ;
10  } else {
11    set_register(state,rd,rt->value);
12    ;
13  }
14  set_register_ref(state,
15                   rd,
16                   SIM_SYMBOL_MSB,
17                   llvm_icmp(32,
18                             SIM_SYMBOL_slt,
19                             rd->value,
20                             0));
21 }
```

Listing 5.6: *Fourth transformation (out of four). The semantics has been transformed into a C function. The function will be executed by the generic simulator.*

Some of the functions used in Listing 5.6 are prefixed with llvm. These functions carry out operations corresponding to functions found in the LLVM framework [9]. No other usage nor resemblance of LLVM is found in Instant Simulator.

The generated C code does not modify the values of registers directly. Updating registers is instead done in separate functions (set_register, set_register_ref), described next in Section 5.3.

5.3 Building Register Functions

The generated functions executing instructions need to retrieve and update values from registers to modify simulator state. This section describes the needed functionality to achieve this.

5.3.1 Aliased register-retrieval

Registers related to instruction operands sometimes need to be referred to in instruction semantics. For instance, assume an architecture where, if an addition operation overflows, the overflow-flag belonging to the destination register needs to be set. Different registers have different overflow-flags, but the instruction semantics cannot specify which register the overflow-flag refers to, unless only a single register can act as destination operand.

A build script builds a function for retrieving an aliased register during runtime, based on a reference register and a unique symbol. For each possible reference made from each register, a unique constant definition is created. The definitions, together with a unique register identification number, provide the necessary tools for finding the aliased register. A simplified version of the result is shown in Algorithm 1.

Note that there is no error handling, as it is assumed that the evaluation tools in Instant Toolsmith ADL checks for errors before the simulator is built. The simulator will assume that references made in instruction semantics have valid aliased registers for all registers that are allowed as operands.

5.3.2 Ranged register-value retrieval

Retrieving a bit field within a register is made possible by use of a utility function from the generic simulator, presented in Chapter 4. The implementation is presented in Algorithm 2. This function does not contain any processor-specific components and could therefore be implemented once in the generic simulator. For consistency, the build scripts will generate the procedure, together with the other register functions.

```
1: procedure REFVALUE(refRegister, symbol)
2:     # Unrolled in the generated source code.
3:     for all reg ∈ registers do
4:         if reg.id = refRegister.id then
5:             # Unrolled in the generated source code.
6:             for all ⟨alias, targetRegister⟩ ∈ reg.references do
7:                 if alias = symbol then
8:                     return targetRegister
9: end procedure
```

Algorithm 1: *A procedure for retrieving a register by alias. The number of registers and their references are known when generating the simulator, hence the loops are unrolled into double-nested switch blocks.*

```
1: procedure RANGEDREGISTERVALUE(register, from, to)
2:     return register.value[from : to]
3: end procedure
```

Algorithm 2: *A procedure for retrieving a bitfield of a register.*

5.3.3 Register value update

Updating the value of registers requires the largest and most complicated implementation, compared to the other register functions described in this section.

Relevant bit fields in overlapping registers need to be updated to ensure consistency. When generating the function, information about overlaps of each register is used so that the correct registers are updated. Algorithm 3 provides a high-level strategy for updating overlapping registers.

An unfortunate side effect of this implementation is that highly overlapped registers contribute to a combinatorial explosion of register updates. This function tends to be thousands lines long in source code.

```
1: procedure setRegister(register, value)
2:     # Unrolled in the generated source code.
3:     for all reg ∈ registers do
4:         if reg.id = register.id then
5:             register.value ← value
6:             # Unrolled in the generated source code.
7:             for all ⟨targetRegister, D_s, D_e, S_s, S_e⟩ ∈ reg.overlaps do
8:                 targetRegister.value[D_s : D_e] ← value[S_s : S_e]
9:             return
10: end procedure
```

Algorithm 3: *A procedure for updating the value of a register, together with overlapping registers. The registers and their overlaps are known when the simulator is generated, so no iterative source code exists in the generated function. This algorithm makes use of for-constructs for clarity reasons only.*

```
1: procedure setRegisterRange(register, from, to, value)
2:     # Only update relevant bit field in register.
3:     register.value[from : to] ← value
4:     # Propagate update on overlapping registers.
5:     setRegister(register, register.value)
6: end procedure
```

Algorithm 4: *Ranged write to register. setRegister from Algorithm 3 is reused to simplify the implementation, at the cost of performance. Propagating value updates via setRegister could result in redundant register updates when the used bit fields are not within the ranges of the register's overlaps.*

5.3.4 Ranged register value update

Updating a bit field within a register is achieved by first using bit operations to only modify the relevant bits in the register. Once the new value is set, the *set_register* function is called to propagate the changes to overlapping registers. Some register updates will be redundant, depending on how large the range is.

```
1: procedure setAliasedRegister(refRegister, symbol, value)
2:     aliasRegister ← refValue(refRegister, symbol)
3:     setRegister(aliasRegister, value)
4: end procedure
```

Algorithm 5: *A procedure for updating the value of a register, referenced as an aliased by another register. Previously described functionality is utilized create this compound procedure. setRegister is found in Algorithm 3. refValue is found in Algorithm 1.*

5.3.5 Aliased register value update

Updating the value of a register, which is referred to by an alias from another register requires no new functionality, as previous register-related functions suffice. From the implementation in Section 5.3.1, retrieving an aliased register is made possible. Section 5.3.3 described the implementation where registers are updated. Using these two functions, the function can be implemented as shown in Algorithm 5.

5.4 Building a Decoder

The last functionality generated by the build scripts, presented in this chapter, is the decoder. Chapter 2 introduced the decoder as a component capable of reading inputs of instructions, transforming them into a representation understood by the simulator. The decoder is given a buffer of machine code together with an address and is expected to return a data structure filled with information (covered in Section 4.5.3).

Due to the existence of Instant Disassembler, a tool to developed by Frej Drejhammar to aid the process of decoding instructions and conditions, a large part of the work required to build a decoder is already done. Decoding of instructions is carried out in three steps. First the correct instruction is found. The needed information is extracted, detailed in. Finally, the extracted information is returned, together with a tag indicating what was decoded.

```
(match 6
  ((0 (skip 15 (match 11
                 ((32 (instruction ADD))
                  (34 (instruction SUB))))))
   (4 (instruction BEQ))
   (5 (instruction BNE))
   (8 (instruction ADDI))))
```

Listing 5.7: *A small trie from a subset of MIPS instruction set, created by Instant Disassembler. The first match construct in this example evaluates the six next bits to one of the four values. The position in the machine code buffer is then moved 6 bits if the value 0 was matched. The skip construct increment the buffer position 15 bits. The instruction construct signifies that a matching instruction is found.*

5.4.1 Finding the right instruction

The first step, finding out the next instruction, requires little work. Instant Disassembler generates a large trie in form of symbolic expressions, which are transformed into nested switch blocks in C source code. Listing 5.7 shows an example for a subset of instructions for the MIPS architecture [6].

Machine code is supplied to the generic simulator in form of a buffer, as seen in the design of Instant Simulator in Chapter 3. The transformed trie is traversed via a 64 bit chunk of data found of the provided address in this buffer. Because the encoding of instructions may vary in bit width, the provided address might not be aligned to 64 bits. Thus, in order to get the next 64 bits, the endianess of both the machine code and the host computer requires consideration. Currently, the build scripts only supports instructions with bit widths of 16 bits through 64 bits.

5.4.2 Decoding instruction operands and flags

Once an instruction is found, the needed information will need to be decoded. Constants are encoded as their binary value. Register operands are based on register classes. As presented in Section A.1.6, register classes contain registers and their corresponding encoding. This allows a single register to have multiple encoding depending on which instruction it occurs in. An example of where this is useful is found in the Qualcomm Hexagon architecture [15], where

some instruction encodings are allowed to be compressed. These compressed encodings allow fewer bits for representing register operands. Each register class is transformed to a register decoder, in the form of a function containing a large switch block.

As mentioned in Chapter 3, the pipeline paths cannot be decoded from the decoders generated by the build scripts, since Instant Toolsmith ADL cannot capture pipeline behavior.

5.4.3 Returning the decoded result

In the last step, if found, an instruction (or condition) has been decoded and its information extracted. The instruction is returned together with a tag, describing whether the result was a condition, an instruction or something unrecognizable. A result indicating neither a instruction nor a condition will halt execution in a simulator.

Chapter 6

Evaluation by Case Study: C67x DSP

This chapter presents an evaluation of Instant Simulator, to assert whether the tool fulfills its purpose and also to explore possibilities for future research. The C67x DSP is described in a machine-readable specification, using Instant Toolsmith ADL. Simulators are then generated by Instant Simulator. Results from running simple programs in these simulators are analyzed with respect to relevant aspects. The information needed to describe the C67x DSP in Instant Toolsmith ADL is retrieved from the instruction-set manual [7].

6.1 Evaluation Methodology

This section motivates the criteria used for evaluation. Important assumptions are provided, together with a description of the environment in which generated simulators are evaluated.

6.1.1 Key aspects

Instant Simulator provides means for comparing the quality of generated machine code from different compilers. The tool is therefore evaluated with respect to three key aspects relevant to its purpose. The following aspects are considered:

Correctness Machine code needs to be executed correctly, but also

measured correctly (in cycles). Correctness of Instant Simulator is therefore validated by running two small programs in this evaluation. Their output (for a specific input) is checked to ensure that the behavior of generated simulators reflects what is expected. One of the programs will be used to evaluate cycle-accuracy.

Performance Generated simulators might have performance constraints for some use cases. Execution time (in form of cycles per second) is therefore measured. This measurement is often referred to as the clock rate, or clock frequency.

Completeness It is important to understand how complete Instant Simulator is in order to make use of the tool and to establish future work. Attempting to describe and simulate real hardware allows the most relevant limitations to be exposed.

6.1.2 Assumptions

A few assumptions are made in this evaluation. Implementing a subset of the C67x DSP, rather than trying to capture as much as possible, is partly done by choice. It is assumed that if a subset of the instruction set works, then the whole instruction set will work. For instance, there might be over ten versions of addition in the instruction set, all with small variations. Explicitly showing that they all work is considered redundant.

The speed at which generated simulators execute is highly dependent of their host systems. The computer which is used for this evaluation, presented in the Section 6.1.3, will act as a reference. That is, the performance evaluation is relevant to modern mid-end computers.

6.1.3 Testing environment

GCC [4] is used with optimization flags -O2 for compiling the source code of the generated simulators. Testing during development of Instant Simulator has

shown that using optimization is especially important for performance because the C source code is generated from scripts. These scripts are designed under the assumption that a compiler will clean up naive source code.

The host system used in this evaluation is a Macbook Air (model year 2013). This laptop has a dual-core Intel Core i5, model i5-4250U with default clock frequency. Primary memory consists of 8 GB of 1600 MHz LPDDR3 SDRAM.

When measuring execution time, the UNIX program time [16] is used and wall-clock time is noted. Programs are run a handful of times, until the margin of error converges to a point where the results are statistically significant. The median value is then chosen as the resulting value.

6.2 A Brief Overview of the C67x DSP

The C67x DSP [7] possesses interesting characteristics affecting its specification in Instant Toolsmith ADL. This section attempts to capture the most interesting features.

Registers are spread out over three register files. Two of these files each contain an identical set of general-purpose registers and are referred to as side 1 (or A) and side 2 (or B). These registers can be used as 16 pairs of 64 bit registers or 32 single 32 bit registers. The third file contains registers handling control flow (program counter, interrupts.).

Functional units are more exposed to programmers in the C67x DSP than what is seen in other architectures. They are named M, L, S and D and carry out different operations (with some functionality overlap). The aforementioned general-purpose register files are both constrained to their own set of these functional units. There are therefore eight of them in total: M1, M2, L1, L2 and so on. The register file for program-flow shares the two versions of the functional unit D with the other register files.

Assembly-programmers need to explicitly state which functional units are to be used in their source code. Furthermore, there are constraints on how a register can be read. Typically, when using a functional unit on side 1, only registers from side 1 are used, though exceptions are allowed via so-called

cross-paths between the register files. However, the number of allowed value-retrievals from the register file is very limited in each instruction bundle.

The instruction-set is sparse, both in quantity and in creativity (which is viewed as a positive property). Instructions that require significantly more cycles than basic arithmetic are implemented at the software level rather than at the hardware level. Two examples where a lack of instructions stand out are division- and modulus operations. Neither call instructions, nor explicit instructions for stack manipulation exists.

Conditions are embedded into instructions in the form of a register and a flag that dictates whether the register is checked for a zero or a non zero value.

The simple pipeline behavior of the C67x DSP, compared to other DSPs, such as the Qualcomm Hexagon DSP [15], almost removes any need to simulate it. No mechanisms allowing pipeline stalling are found in the hardware. Instead, any structural, control flow, or data-dependency hazards not resolved during compilation (or assembling) result in undefined behavior during execution.

6.3 Limitations and Work-arounds

A few major issues have been discovered when describing the C67x DSP in Instant Toolsmith ADL, which reduce the quality of case study. These issues now summarized.

6.3.1 Encoding issues

Unique bit field values helping decoders identify the correct instructions (also known as operation codes) are placed among the least significant bits of an encoded instruction in the C67x DSP. Instant Disassembler, used by the build scripts to generate the decoder (Section 5.4.1), cannot handle this format well. All operation codes (and the VLIW flag) are therefore moved to the most significant bits in the specification of the DSP to allow the C67x DSP to be simulated. Machine code for the real C67x DSP differ for the machine

code compatible with generated simulators for C67x DSP, as a result of this change.

6.3.2 Lack of assembler

By changing the encoded representation of instructions to cope with encoding issues in Section 6.3.1, the vendor assembler for the C67x DSP will not work (Instant Assembler is being developed by Mattias Jansson in parallel with Instant Simulator). To avoid having to hard code bits, a small program has been written for assembling a symbolic representation of code to executable machine code.

6.3.3 Test programs

Due to a lack of a full-fledged assembler, new test programs will need to be written instead of reusing programs. Texas Instruments provides a set of test programs which they run in product benchmarks [1]. These programs are applicable to the real world tasks for the C67x DSP, but they are currently provided in C source code, not assembly source code.

When comparing the test programs used in this case study to assembly code provided by Texas Instruments for different DSP architectures, it appears that the numeric intensity in their code is higher. Their machine code consists of more arithmetic and logic instructions and less move instructions (explained in Section 2.2.4). The two test programs in this case study might therefore not truly reflect typical execution in the C67x DSP.

6.3.4 Lack of pipeline complexity

A more convenient issue lies in the simplicity of the pipeline behavior of the C67x DSP. A lack of complexity in the hardware (that is relevant to this project) means Instant Simulator's features cannot be shown to the fullest. This is redeemed by utilizing one of the key-points behind the motivation for Instant Simulator: Changing the specification of the hardware and quickly generating a new simulator. After evaluating a basic version of the C67x DSP,

the specification will be extended with new pipeline features and explicit stack manipulation instructions.

6.4 Describing the C67x DSP in Instant Toolsmith ADL

This section summarizes the specification of the C67x DSP, formalized in Instant Toolsmith ADL, together with some of the issues that were discovered during the process. The final specification of the C67x DSP, allowing simulators to be generated for the two test programs in Section 6.5 and Section 6.6, has a size of nearly 300 lines of (Instant Toolsmith ADL) code.

6.4.1 Registers and register classes

Registers are defined in a straight-forward manner, as shown in Listing 6.1. Some registers appear in two register classes because they are also allowed as operands for instruction conditions.

```
(define-abstract-reg general-purpose
  ((size 32)
   (access "rw")))

(define-reg A1A0
  ((extends general-purpose)
   (size 64)
   (asm "A1:A0")))
;; ... More 64 bit registers omitted

(define-reg A0
  ((extends general-purpose)
   (asm "A0")
   (overlap (A1A0 0 31))))
;; ... More 32 bit registers omitted
```

Listing 6.1: *A few of the register-definitions for one of the three register files in c67x. The other register-file uses the letter B as a prefix for the register names.*

```
(declare-functional-units (L1 L2 S1 S2 M1 M2 D1 D2))
```

Listing 6.2: *The declaration of functional units in the architecture. There are two groups of four functional units identified by a suffix.*

```
(define-util-fun (check-condition Z reg)
  "Checks if a condition holds. returns 1 (true) if both
     Z is set and reg is cleared or vice versa.
     Otherwise 0 (false) is returned"
  (rif (icmp 1 'eq Z 1)
       (icmp 32 'eq reg 0)
       (icmp 32 'ne reg 0)))
```

Listing 6.3: *A utility function that checks the condition for conditional instructions. Conditions are embedded in instructions and do not require explicit definitions. This utility function is called in the semantics of all instructions before updating the state.*

6.4.2 Functional units

Listing 6.2 presents the declaration of functional units. Units suffixed with the number 1 can use the first register file while the units suffixed with the number 2 use the second. There exists some limited functionality to allow execution of instructions where both register files are used. The register-usage constraints, noted in Section 6.2, are not coupled to the declaration in Instant Toolsmith ADL. Instead, the constraints will be exposed in the definitions of all the instructions.

6.4.3 Conditions

Since conditions are embedded into instructions in form of operands, no external conditions are defined for the basic version of the C67x DSP. A utility function is defined (Listing 6.3) and then used in every instruction to check whether state should be updated.

6.4.4 Instructions

When defining instructions for the C67x DSP, the constraints of the functional units and how they are coupled to register files reveal an ugly side of Instant Toolsmith ADL. Each instruction needs to be defined twice to cover both functional units. The instruction field describing which functional units are usable cannot help us because a different functional unit requires a different register file. Moreover, the use of cross paths (when applicable) requires that the instruction count doubles again.

6.4.5 Pipeline

The seven-stage (for single-cycle execution) pipeline of the C67x DSP cannot be described in standard Instant Toolsmith ADL due to the incompleteness of the ADL. Instead, instructions will include meta-data containing temporary pipeline behavior. The build scripts of Instant Simulator are patched to parse the meta data and create pipeline information the generic simulator can interpret.

Since hazards are managed during compile time, all path steps except the ones executing the instructions are empty. There is no need to acquire and release resources. A pipeline is still required however, so that delay-slots are executed correctly. The specification of the C67x DSP is extended in Section 6.6 to utilize pipeline paths with register acquisitions and pipeline interlocking.

6.5 Testing the C67x DSP with Fibonacci

This section presents how a small implementation calculating the Fibonacci-sequence is used to assert correctness of a generated simulator for the C67x DSP. The program is chosen to be as simple as possible, while trying to avoid being dull. Describing the C67x DSP means there are no hazards detected during execution. Unused delay slots are filled with NOP instructions, not carrying out any actions.

```
1: procedure FIBONACCI(N)
2:     a ← 0
3:     b ← 1
4:     for all i ← 0..N do
5:         a, b ← b, a + b
6:     return a
7: end procedure
```

Algorithm 6: *An non-recursive version of Fibonacci, from Sussman et al. [13], which serves as a initial example due to its simplicity.*

6.5.1 Program description

Algorithm 6 provides a high-level description of how Fibonacci sequences are calculated and Listing 6.4 presents the corresponding assembly source code. The algorithm is taken from Sussman's et al.[13].

The source code consist of nine bundles, four of which are single NOP instructions. Since the pipeline has a depth of seven stages, six extra cycles are executed after the last instruction is fetched. A total of 15 cycles is therefore needed for fibonacci(0). Each iteration in the loop executes three bundles of arithmetic and moves, and then five bundles of delay slots. This means that every iteration increases the number of cycles with eight and $cycles(fibonacci(N)) = 15 + 8 \cdot N$ cycles.

6.5.2 Results

The number of cycles executed, relative to the number of iterations, are shown in Table 6.1. These results reflect what was expected from the previously calculated numbers. Section 6.7 provides the results from running the test program on a large input and measuring cycles per second.

```
1          ADDK .S2        N, B1
2 ||       ADDK .S1        1, A2
3 ITER:
4          MV .L1          A4, A3
5    [B1]  MV .L1          A2, A4
6          ADD .D1         A3, A2, A2
7 || [B1]  B .S2           ITER
8    [B1]  ADDK .S2        -1, B1
9          NOP             4
```

Listing 6.4: *. The C67x DSP's assembly source code for iterative Fibonacci. No hazards will be detected and all instructions, except for the program jump, are executed in a single cycle. The N:th Fibonacci number is allocated to register B1 and the final value is stored in A4.*

fibonacci(N)	Result	Cycles
fibonacci(0)	0	15
fibonacci(1)	1	23
fibonacci(1)	1	23
fibonacci(2)	1	31
fibonacci(3)	2	39
fibonacci(5)	5	55
fibonacci(8)	21	79
fibonacci(13)	233	119
fibonacci(21)	10946	183

Table 6.1: *The results from calculating different Fibonacci numbers and the corresponding number of executed cycles.*

6.6 Extending the C67x DSP for Recursive Factorial

This section tests a more complex program than Section 6.5. The aim is to show how the C67x DSP can be extended so that the assembly code is simplified.

A stack is first implemented to allow for recursion. The nearly non existent pipeline behavior is then extended by enhancing the pipeline paths of instructions to allow for stalls. Assembly source code will thus be simplified

by removing explicit NOP instructions.

6.6.1 Program description

To test a program that calculates factorial recursively, an example is taken from Sussman et. al [13] again. Their book provides both a high-level implementation (using their symbolic assembly language) and a thorough description.

Assembly source code for this program is found in Listing 6.9. How the program manages to calculate factorial is not crucial to this evaluation. The most important instructions in this example are the NOP instructions filling out all delay slots to avoid issues with hazards, because these will be removed as the specification of the C67x DSP is revised.

6.6.2 Adding stack manipulation

The test program is recursive and therefore requires a stack for saving state in primary memory (lines 9 through 12 in Listing 6.5) and restoring state (lines 19 through 22 in Listing 6.5). As described in Section 4.5.4, Instant Simulator partially provides means for implementing a stack, but the primitives for load and store instructions need to be supplied manually. Primitives are documented in Section A.4.1. The needed primitives are declared as shown in Listing 6.6. Their definitions need to be supplied in C source code, presented in Listing 6.7. These declarations and definitions allow the two primitives to be used in the semantics of instructions.

6.6.3 Managing data hazards

The first extension to the C67x DSP is to introduce detection of data-dependency hazards. By introducing mutual exclusion of registers that are updated, the pipeline will detect instructions attempting to read registers before their values have been stored. The path steps that belong to the instructions, which previously did nothing, are updated to include locking, releasing, and reading of registers.

```
1  PROLOGUE:
2           MVK       .S2       FACT_DONE, B3
3           ADDK      .S2       N, B1
4
5  FACT_LOOP:
6           CMPEQ     .L2       1, B1, B2
7     [B2]  B         .S2       BASE_CASE
8           NOP                 5
9           STW       .D2       B3, *B15(0)
10          ADDK      .S2       4, B15
11          STW       .D2       B1, *B15(0)
12          ADDK      .S2       4, B15
13          ADDK      .S2       -1, B1
14          MVK       .S2       AFTER_FACT, B3
15          B         .S2       FACT_LOOP
16          NOP                 5
17
18 AFTER_FACT:
19          ADDK      .S2       -4, B15
20          LDW       .D2       *B15(0), B1
21          ADDK      .S2       -4, B15
22          LDW       .D2       *B15(0), B3
23          NOP                 4
24          MPY       .M1x      B1, A1, A1
25 ||       B         .S2       B3
26          NOP                 5
27
28 BASE_CASE:
29          ADDK      .S1       1, A1
30          B         .S2       B3
31          NOP                 5
32
33 FACT_DONE:
```

Listing 6.5: *C67x DSP assembly source code for recursive factorial. The aim of Section 6.6 is to remove as many of the NOP instructions and instead let the pipeline detect hazards and stall.*

```
(define-primitive
   true (ldw addr offset) "custom_ldw" 0)
(define-primitive
   false (stw addr offset value) "custom_stw" 0)
```

Listing 6.6: *Declarations of custom C67x DSP primitives which will allow their usage in instruction semantics. Their definitions are found in Listing 6.7.*

```
1  uint32_t
2  custom_ldw(sim_state_t* state,
3             uint64_t addr,
4             uint64_t offset)
5  {
6    return state->memory[(addr/4)+offset];
7  }
8
9  void
10 custom_stw(sim_state_t* state,
11            uint64_t addr,
12            uint64_t offset,
13            uint32_t value)
14 {
15   state->memory[(addr/4)+offset] = value;
16 }
```

Listing 6.7: *Definitions of custom C67x DSP primitives in C. These are needed to implement stack operations in the test source code.*

Instructions with no delay slots will utilize the path shown in Figure 6.1. Load instructions have four delay slots and require a longer path, shown in Figure 6.2. These different paths helps us eliminate the explicit delay slots that are inserted after restoring the stack, found on line 23 in Listing 6.5. Listing 6.8 shows an updated version of the block of assembly code, where this data-dependency hazard will occur and be handled by the pipeline during execution.

Removing some of the explicit delay slots simplified the assembly source code. However, the five delay slots following control flow instructions are still needed because they are there to avoid possible control flow hazards. The next section will reduce the assembly source code complexity further by eliminating

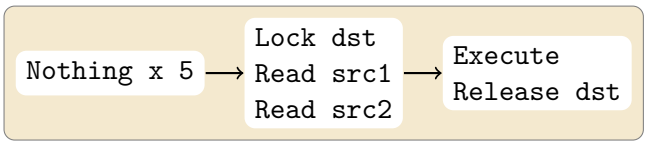

Figure 6.1: *Instruction path for single-cycle instructions in the C67x DSP.*

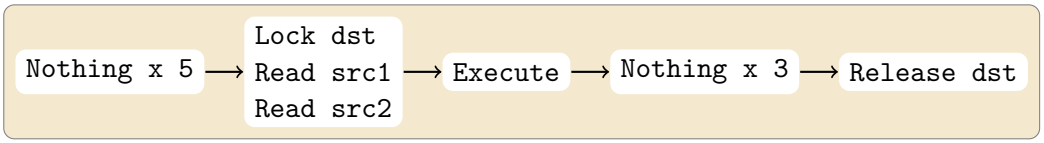

Figure 6.2: *Instruction path with four delay slots in the C67x DSP.*

```
1  PROLOGUE:
2          ...
3
4  FACT_LOOP:
5          ...
6
7  AFTER_FACT:
8          ADDK     .S2     -4, B15
9          LDW      .D2     *B15(0), B1
10         ADDK     .S2     -4, B15
11         LDW      .D2     *B15(0), B3
12         ;; 4 NOP instructions removed here
13         MPY      .M1x    B1, A1, A1
14 ||      B        .S2     B3
15         NOP              5
16
17 BASE_CASE:
18         ...
19 FACT_DONE:
```

Listing 6.8: *Assembly of recursive factorial with implicit data dependencies. The four NOP instructions that followed stack restoration are removed. They are instead handled by stalling the pipeline at run time.*

most of the remaining explicit delay slots filled with NOP instructions.

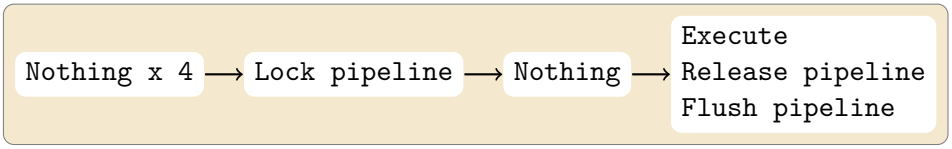

Figure 6.3: *Instruction path of branch instructions for the C67x DSP*

6.6.4 Managing control hazards

The five delay slots following branch instructions require five NOP instructions (if nothing else can fill the void). By adding pipeline locking in the decode-phase of control flow instructions, only three delay slots are needed. The three delay slots are left because an instruction is not decoded (and is therefore unknown) until the fourth pipeline stage.

After incorporating pipeline interlocking for control flow instructions, more extensions can be introduced to avoid NOP instructions. Flushing the pipeline on unconditional program jumps removes the last three delay slots. Figure 6.3 shows the final path for control flow instructions.

Conditional control flow instructions cannot utilize pipeline flushing because the current pipeline implementation will flush unconditionally, thus removing relevant instructions if a jump would not be carried out. While the flush command is still in conditional control flow instructions, three delay slots are needed to avoid the change in program behavior.

Listing 6.9 shows the updated assembly source code, after incorporating the means for detecting control hazards. The first branch still needs the three delay slots because it is conditional.

```
1  PROLOGUE:
2           ...
3
4  FACT_LOOP:
5           ...
6     [B2]  B         .S2     BASE_CASE
7           NOP               3, r8
8           ...
9           B         .S2     FACT_LOOP
10
11 AFTER_FACT:
12          ...
13 ||       B         .S2     B3
14
15 BASE_CASE:
16          ...
17          B         .S2     B3
18
19 FACT_DONE:
```

Listing 6.9: *Optimized and partially folded assembly source code for the C67x DSP's recursive factorial. All unconditional branches have had their delay slots removed. Only three delay slots (down from five) are left after the conditional branch.*

6.7 Performance Results

This section gives the performance results from executing the two test programs. Two versions of the two test programs are used to measure execution speed in form clock rate. Table 6.2 provides the results for the different versions.

For the first test program (fibonacci), a generated simulator manages to execute 2.6 million cycles per second when calculating larger Fibonacci numbers. Since no run-time hazard exist, all the pipeline contributes with is extra overhead in execution time. To compare, the simulator executes 9.7 million cycles per second without a pipeline. The NOP instructions are moved in front of the jump instruction to include the delay slots in the execution time when a pipeline is not used.

Program	Pipeline	Million cycles/second	Coeff. of variation
fibonacci	No	9.7	0.39%
fibonacci	Yes (no stalling)	2.6	0.35%
factorial	Yes (no stalling)	4.0	0.37%
factorial	Yes	4.8	0.36%

Table 6.2: *The number of cycles per second during execution of the test programs. The first test is executed on simulator without a pipeline. Pipelined simulators with no stalling mechanisms use explicit NOP instructions, as presented in previous sections.*

The number of cycles rise to four million cycles per second when measuring factorial in a pipelined simulator. Performance benefits from the lack of instruction level parallelism (explained in Section 2.5.1) in in the source code, compared to the previous test program. After extending the pipeline to detect hazards and use stalling, performance increases further as the cycle count rise to 4.8 million cycles per second.

6.8 Analysis

This section provides an analysis of the results from describing the C67x DSP and testing the two programs with generated simulators, with respect to the key aspects provided in Section 6.1.1.

6.8.1 Correctness

While the results given from running the fibonacci program were as expected, the program itself might be too simplistic to draw any major conclusions on correctness with respect to cycle-accuracy. The program that calculates factorial is deemed too complex to calculate cycles manually (and comparing with the execution results).

Manually calculating the number of cycles required to execute a program (for some specific input) is tedious and error prone. Benchmark suites

from hardware manufactures tend to include this information, which further motivates the usage of their source code over small hand-made examples.

6.8.2 Performance

An unflattering side of the generated simulators is shown during benchmarking, as low throughput means less work per cycle and therefore more cycles per second are executed. From a performance perspective, simulators prefer as little parallelism as possible in order to achieve a high cycle count per second.

A similar problem lies in the pipeline, which slows down execution time when used in the Fibonacci program. Performance increased when pipeline stalls were introduced in the second test program (recursive factorial).

The influence from parallelism in programs and the behavior of a pipeline on execution time indicates that the performance of Instant Simulator is highly dependent on the specification of the processor. The performance of simulators generated by Instant Simulator might therefore be unstable when descriptions of DSPs are updated frequently.

6.8.3 Completeness

When describing the C67x DSP in Instant Simulator ADL, a combinatorial explosion of instruction variations turns out to be bittersweet, because it reveals a lack of flexibility in how Instant Toolsmith ADL. However, it also shows that Instant Toolsmith ADL can sometimes manage more structurally focused specifications, where register files and functional units are exposed.

The pipeline extensions made for the recursive factorial program in Section 6.6 reveal a lack of important features for realistic simulation. Modern DSPs will not lock the pipeline, but instead use branch prediction and speculative execution. However, this is the beyond the current scope of Instant Simulator.

The most limiting problem in this case study is how instructions are encoded. When moving the operation codes of instructions to the most

significant bits, the vendor assembler could not be used. No real assembler means that no relevant testing programs could be executed on generated simulators. Unfortunately, the quality of this evaluation is therefore reduced.

Chapter 7

Conclusions

This report presented a tool for generating simulators based on specifications of DSPs formalized in Instant Toolsmith ADL, with the purpose of comparing generated machine code from different compilers. The tool was evaluated in a case study, where the C67x DSP [7] was specified and different simulators were generated. These simulators were tested with small programs. Two main contributions have been identified:

1. A tool for generating simulators from specifications of DSPs formalized in Instant Toolsmith ADL.

2. Assertion of key properties that motivates the existence of Instant Simulator for its purpose, but also reveal its limitations.

The rest of this chapter provides a more in-depth view of the conclusions drawn from evaluating the Instant Simulator. This chapter ends with suggestions for future research.

7.1 Key Properties of Instant Simulator

Several important properties of Instant Simulator were shown in the case study in Chapter 6. These properties are now summarized.

7.1.1 Correctness

The correctness of the generated simulators was tested by evaluating simple programs. Problems with asserting cycle-accuracy highlighted the importance of using real benchmark suites from manufactures, as they often include vital information that otherwise need to be calculated by hand. While using benchmark programs is currently impossible with Instant Simulator, future evaluation will benefit greatly once Instant Toolsmith ADL support specifications of DSPs with enough information to use real applications.

7.1.2 Performance

Drawing conclusions on performance is difficult when no concrete execution speed constraints are coupled to the purpose of Instant Simulator. Performance is still a relevant aspect of evaluation because it influences the usefulness of the tool.

The case study revealed a potential issue as performance was heavily affected by the specification of a DSP. This unpredictable performance might be mitigated by a static pipeline, similar to the techniques presented in related work (Section 2.6).

7.1.3 Completeness

Instant Simulator's level of completeness was shown in the process of specifying a real DSP (albeit a subset) in Instant Toolsmith ADL. Using a real-world DSP motivated the potential in Instant Simulator as a real tool, rather than being an academic exercise.

The case study exposed some caveats when the more structural parts of the C67x DSP were specified, leading to redundancies. However, within the current scope of Instant Simulator, instruction-set simulation without pipeline behavior has been shown to be supported.

7.2 Future Work

The previous section presented both limitations and promising aspects of Instant Simulator. These findings provide a basis for future research which is now presented.

Focusing on the purpose of Instant Simulator (providing cycle-accurate benchmarking with simulators), future work should focus on maximizing the accuracy of these simulators. Two very beneficial features currently missing in Instant Simulator are more detailed memory hierarchies and pipelines generated from specifications in Instant Toolsmith ADL.

A realistic memory simulation is important for cycle accuracy, as execution time is heavily affected by the latencies involved with moving data from different regions in the memory hierarchy of a computer.

Once Instant Toolsmith ADL can capture descriptions of pipeline behavior, the current naive pipeline implementation found in the generic simulator (Chapter 4) should be replaced by an optimized version. The related work presented Section 2.6 provides a starting point to achieve optimized cycle-accurate simulation.

Bibliography

[1] *Benchmarks - C67x DSP - TI.Com*. URL: `http://www.ti.com/lsds/ti/dsp/c6000_dsp/c67x/benchmarks.page` (visited on 04/17/2014).

[2] Paul C. Clements. "A survey of architecture description languages". In: *Proceedings of the 8th international workshop on software specification and design*. 1996, p. 16.

[3] Antoine Colin and Isabelle Puaut. "Worst case execution time analysis for a processor with branch prediction". In: *Real-Time Systems* 18.2-3 (2000), 249–274.

[4] *GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)*. URL: `http://gcc.gnu.org/` (visited on 05/25/2014).

[5] Peter Grun et al. "EXPRESSION: An ADL for system level design exploration". In: *Department of Information and Computer Science, University of California, Irvine, Technical Report* (1998), 98–29.

[6] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[7] Texas Instruments. *TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide, November 2006*.

[8] Rola Kassem et al. "Harmless, a hardware architecture description language dedicated to real-time embedded system simulation". In: *Journal of Systems Architecture* 58.8 (2012), 318–337.

[9] *LLVM Language Reference Manual — LLVM 3.4 documentation*. URL: `http://llvm.org/docs/LangRef.html` (visited on 04/01/2014).

[10] Scott A. Mahlke et al. "A comparison of full and partial predicated execution support for ILP processors". In: *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. 1995, 138–149.

[11] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. "A formal concurrency model based architecture description language for synthesis of software development tools". In: *ACM SIGPLAN Notices*. Vol. 39. 2004, 47–56.

[12] Michael Sperber et al. *Revised [6] report on the algorithmic language Scheme*. Vol. 19. Cambridge University Press, 2010.

[13] Gerry Sussman, Hal Abelson, and Julie Sussman. *Structure and interpretation of computer programs*. Vol. 10. 1983.

[14] Edwin J. Tan and Wendi B. Heinzelman. "DSP architectures: past, present and futures". In: *ACM SIGARCH Computer Architecture News* 31.3 (2003), 6–19.

[15] Qualcomm Technologies Inc. "Hexagon V4 Programmer's Reference Manual". In: (Aug. 2013).

[16] *time(1) - Linux man page*. URL: `http://linux.die.net/man/1/time` (visited on 05/25/2014).

[17] *Unison - Robust, Scalable, and Open Code Generation by Combinatorial Problem Solving | SICS*. URL: `https://www.sics.se/projects/unison` (visited on 09/15/2013).

Appendix A

Instant Toolsmith ADL

This appendix describes the architecture description language (ADL) used in the Instant Toolsmith project. Architectures are described using this ADL in order for various tools to be generated. The appendix is written as joint work by Mattias Jansson and Karl Johansson.

A.1 Describing Registers

The register specification is a model of the available registers in the architecture. It describes their properties of as well as relationships between registers such as overlap and grouping. This section describes the way to describe a set of registers so that they can be referred to and used correctly.

This section will assume a simple processor architecture with 4 registers: *AX*, *AH*, *AL* and *PC*. Registers *AH* and *AL* will be assumed to occupy the same bits as the upper and lower half of *AX*, respectively. Register *PC* is a read-only register.

A.1.1 Assembler syntax

The assembler syntax property is simply the name that the register will have when referred to in assembly source code. In the specification, each register will have an assembler syntax as well as a unique identifier for use by the

```
(define-reg ax
  ((asm "AX")))
```

Listing A.1: *The most basic register definition.*

```
(define-reg ax
  ((asm "AX")
   (size 32)))
```

Listing A.2: *Register with a defined size*

toolset. The main difference between the identifier and the assembler syntax is that the assembler syntax does not have to be unique (only unique within the register class.) Listing A.1 will set the identifier and assembler syntax for *AX*.

A.1.2 Size

The *size* property is simply the number of bits that the register consists of. Listing A.2 shows an example by extending the specification for *AX* from Section A.1.1.

A.1.3 Access

The *access* property tells the toolset whether a register can be written to, read from, or both. This is specified using the strings "r", "w", "rw". The registers *AX* and *PC* are extended with the *access* property in listing A.3.

A.1.4 Overlapping registers

As mentioned in the beginning of this section, there are certain registers that overlap others. Without this information in the specification, all registers would be treated as independent and writing to one would not affect any other. Using the *overlap* property, such dependencies can be specified as in

```
(define-reg ax
  ((asm "AX")
   (size 32)
   (access "rw")))

(define-reg pc
  ((asm "PC")
   (size 32)
   (access "r")))
```

Listing A.3: *Ax and PC declared with different access properties.*

```
(define-reg ax
  ((asm "AX")
   (size 32)
   (access "rw")))

(define-reg al
  ((asm "AL")
   (size 16)
   (access "rw")
   (overlap (ax 0 15))))

(define-reg ah
  ((asm "AH")
   (size 16)
   (access "rw")
   (overlap (ax 16 31))))
```

Listing A.4: *Sub-registers with specified overlap.*

listing A.4. This specifies that AH overlaps the upper 16 bits of AX and AL overlaps the lower 16 bits. Note how AX is referred to using its identifier in the *overlap* property. Overlaps can be defined in an arbitrary way. A register may overlap, and be overlapped by, several other registers.

```
(define-reg ax
  ((asm "AX")
   (size 32)
   (access "rw")))

(define-reg al
  ((asm "AL")
   (size 16)
   (access "rw")
   (overlap (ax 0 15))
   (ref (ah 'my-sibling) (ax 'my-parent))))

(define-reg ah
  ((asm "AH")
   (size 16)
   (access "rw")
   (overlap (ax 16 31))
   (ref (al 'my-sibling) (ax 'my-parent))))
```

Listing A.5: *Sub-registers extended with sibling references.*

A.1.5 References

For some instructions, there may be a need to refer to registers via an arbitrary relation to another one. For example, one could say that AL and AH are sibling registers since they are part of the same larger registers. They are related, but do not overlap. The *ref* property enables this type of arbitrary relations. In listing A.5, sibling and parent relationships have been given clear names.

The names of the relationships can be anything. In this case, they have purposely been given the same names for all the 16 bit registers. This way, when dealing with any of the 16 bit registers, one can refer to the sibling or the parent of the register.

A.1.6 Register classes

Registers are normally grouped into classes. For example, some instructions may operate only on 16-bit registers while others operate only on 32-bit

```
(define-register-class <id>
  ((<reg> (encoding (<width> <value>)))  ... ))
```

Listing A.6: *Syntax of register class definition.*

```
(define-reg-class large-reg
  ((ax (encoding (1 0)))
   (bx (encoding (1 1)))))

(define-reg-class small-reg
  ((al (encoding (2 0)))
   (ah (encoding (2 1)))
   (bl (encoding (2 2)))
   (bh (encoding (2 3)))))
```

Listing A.7: *Example of register class definitions.*

registers. Some instructions only operate on a very select set of registers. Moreover, the binary representation of a register might be different depending on the instruction using it. To deal with this, the registers are grouped into register classes. A register class is simply a list of all registers that are in the class and how to binary encode each particular register in that particular class. The syntax is described in listing A.6. Id is the unique identifier of the register class, *reg* is an identifier of an existing register, width is the number of bits that the register will be encoded with and value is the actual binary encoding for the register in that register class.

Starting from the specification in Section A.1.5, listing A.7 how the register classes could be defined. When defining an instruction, the operands for that instruction can are specified by referring to a register class.

A.1.7 Abstract registers

In the examples given throughout this section there is some code duplication. In a more advanced processor architecture, there would be much more. To mitigate this, one can define abstract registers. These are registers which can not be referred to or used except for when extending them into regular registers. Listing A.8 shows a set of register definitions which are equivalent to the example in Section A.1.5 (with the addition of PC).

```
(define-abstract-reg large-reg
  ((size 32)
   (access "rw")))

(define-abstract-reg small-reg
  ((size 16)
  (access "rw")))

(define-reg ax
  ((asm "AX")
   (extends large-reg)))

(define-abstract-reg ax-child
  ((extends small-reg)
   (ref (ax 'my-parent))))

(define-reg al
  ((asm "AL")
   (extends ax-child)
   (overlap (ax 0 15))
   (ref (ah 'my-sibling))))

(define-reg ah
  ((asm "AH")
   (extends ax-child)
   (overlap (ax 16 31))
   (ref (al 'my-sibling))))

(define-reg pc
  ((asm "PC")
   (extends large-reg)
   (access "r")))
```

Listing A.8: *Registers defined using abstract registers.*

Some of the properties of *AX* are captured in the abstract register *large-reg*. The small registers do not inherit directly from *small-reg*, but indirectly via *ax-child*. They inherit *size* and *access* properties from *small-reg* and also a register reference from *ax-child*.

PC has its size set by extending large-reg, but overrides the access property to make it read-only.

A.2 Describing Instructions

Instructions define the behavioural part of processors. Their operation-semantics are described using a subset of Scheme [12] together with special forms created for Instant toolsmith. Instructions also expose structural constraints to some extent, by describing valid functional units, encoding size, operands and other properties.

This section presents how instructions are described by use of an initially empty example instruction, which is extended as more propterties are introduced. We begin with the empty instruction shown in Listing A.9.

A.2.1 Size

The size field reflects the total size of the encoded instruction. This information is also available in the field decribing how an instruction is encoded. Our example instruction is extended to include the size property in Listing A.10. The size describes how many bits are needed to encode the instruction. Varying instruction-lengths are allowed and expected for the type of architectures Instant Toolsmith targets.

```
(define-instruction add-max
  ....)
```

Listing A.9: *An initially empty instruction, which will be filled with properties as they are introduced in this section.*

```
(define-instruction add-max
  ((size 32)))
```

Listing A.10: *An instruction extended with the size property. The encoding of this instruction is 32 bits long.*

```
(define-instruction add-max
  ((size 32)
   (functional-unit (arith1) (arith2))))
```

Listing A.11: *An instruction extended with a property that describes which functional units can execute it. In this example, there are two potential functional units available.*

A.2.2 Functional units

This property describes what combinations of functional units are capable of executing an instruction. Listing A.11 shows how this property has been added to the example instruction.

Only functional units that are declared, as described in Section A.4.5, are available to an instruction. These combinations of functional units are expressed as disjunctions of conjunctions. For instance, the combination *((A B) (B C))* expresses that either both functional units A and B are used or both B and C are used.

A.2.3 Issue slots

When combining several instructions into packets (or bundles), there may be constraints on the order in which the instructions are to appear in the packet. There may also be constraints on the number of different types of instructions that can be issued in the same packet.

The concept of issue slots is a way to model some such constraints. The available issue slots are globally declared as described in Section A.4.6 and instructions will be issued in slots during packet reordering and encoding.

For example, if the two issue slots *is1* and *is2* are declared in that order.

```
(define-instruction add-max
  ((size 32)
   (functional-unit (arith1) (arith2))
   (issue-slot (is1) (is2))))
```

Listing A.12: *An instruction extended with the issue-slot property. In this example, the instruction can be issued in either is1 or is2.*

Then, if there is an instruction packet with two instructions which use *is2* and *is1* respectively, they will be reordered in the packet so that the instruction using *is1* occurs first in the packet.

Furthermore, if only those two issue slots are available, then no more than two instructions can be issued in the same packet. Also, two instructions that only support issue slot *is1* can not be issued in the same packet.

An example of issue slot usage is given in Listing A.12.

A.2.4 Operands

Instructions are required to specify which operands are used in the semantics. The example instruction has been extended with operands in Listing A.13. All operands are defined by the name they are referred to in semantics together with some type-dependent information. The required information is documented below:

(reg <name> <register-class>)
: The register class is needed in order encode and decode registers.

(imm <name> <min value> <max value>)
: Immediates require information about the minimum and maximum values. For encoding, immediate-operand sizes are implied based on these two values.

(zimm <name> <min value> <max value>)
: Non-zero immediates are similar to normal immediates, except a zero value represents the larger value within the range instead. This allows the maximum value allowed for non-zero immediates to be one higher than normal immediates of same bitlength.

```
(define-instruction add-max
  ((size 32)
   (functional-unit (arith1) (arith2))
   (issue-slot (is1) (is2))
   (operands (reg dst r32) (reg src1 r64) (reg src2
      r64))))
```

Listing A.13: *An instruction that describes valid input operands. For this instruction, only register operands are used.*

(imm <name> ((<sub-name> <min-value> <max-value>) ...))
Immediates often function as input flags in instructions. An immediate operand can be composed of more than one flag, which would require bitmasking to extract relevant bits. With this composite operand, the masking is no longer needed as sub-fields are bound to separate identifiers.

(rel-address <name> <min value> <max value>)
Relative addresses are used in instructions that do flow control based on offsets. They require a numeric minimum and maximum value, even though they in assembly are given as labels.

(abs-address <name> <min value> <max value>)
Absolute addresses are similar to relative addresses, except program jumps are not based on an offset but are absolute addresses.

A.2.5 Assembly syntax

The *asm* property describes how instructions are presented in assembly source code. The currently supported format only allows for representations beginning with a mnemonic, followed by the operands. Listing A.14 shows how the example instruction is extended to include an assembly syntax.

Operands are prefixed with a % escape character to identify where to insert their assembly representation. For registers, this representation is found in their definition. Unlike registers, immediates are literals, hence their syntactical representation is their value.

```
(define-instruction add-max
  ((size 32)
   (functional-unit (arith1) (arith2))
   (issue-slot (is1) (is2))
   (operands (reg dst r32) (reg src1 r64) (reg src2 r64))
   (asm "addmax %src1, %src2, %dst")))
```

Listing A.14: *An instruction after adding its assembly representaion. The % prefix indicates that the assembly code string found in the operand should be inserted.*

```
(define-instruction add-max
  ((size 32)
   (functional-unit (arith1) (arith2))
   (issue-slot (is1) (is2))
   (operands (reg dst r32) (reg src1 r64) (reg src2 r64))
   (asm "addmax %src1, %src2, %dst")
   (encoding P (16 #b0111100110101101) src1 src2 dst )))
```

Listing A.15: *An instruction with encoding information added.*

A.2.6 Encoding

Instructions need an encoded representation so that they can be both assembled and disassembled. Listing A.14 shows how the example instruction is extended with an encoding property.

The encoding property is a list containing an optional packet-flag, constants, and operands that collectively contribute to a number of bits that reflects the value of the size property. Operands are referred to by the alias used in the semantics.

Register operands have their encoded representation based on the register class they belong to. Immediates are simply constants and thus their numeric value is used in the encoding. Constants require an extra field which specifies their lengths. For VLIW architectures, a P symbol reflects a packet flag of 1 bit.

A.2.7 Semantics

Semantics are defined using a subset of Scheme. Features that belong in the functional-programming paradigm have been removed to simplify evaluation of the semantics. Special forms that are not mentioned in this section are disallowed.

The special-forms found in Scheme: *let*, *let**, and *begin* are valid expressions. The Lambda form cannot be used and high-order functions are disallowed. Moreover, No form of recursion nor any type of iterative source code can be used directly in the semantics (but similiar functionality can be implemented using macro functions, described in Section A.4.3). There are three special-forms used for retrieving and updating registers:

(set-reg! <register> <value>)
: Updates a register with a new value. Both operands can be registers, with the implication that the second operand refers to the value of the register and not the register itself.

(reg-ref <register> <symbol>)
: Retrieves a register based on the provided register and an alias.

(reg-range <register> <start> <end>)
: Retrieves a subrange of a register which still contains its parent register's properties. This function can be used for either updating a register or using its value.

Instant toolsmith cannot use the primitive if-expressions because a compiler will need to know when predicates depend on runtime state and when the outcome of predicates can be configured statically. There are two version of this form instead, *rif* and *cif* for if-statements resolved at runtime and if-statements resolved at configure-time respectively.

A.2.8 Abstract instructions

Instructions that share similar properties can use an abstract instruction as a template, to avoid source code duplication. These abstract intructions are defined with *define-abstract-instruction*, using properties identically to

```
(define-instruction add-max
  ((size 32)
   (functional-unit (arith1) (arith2))
   (issue-slot (is1) (is2))
   (operands (reg dst r32) (reg src1 r64) (reg src2 r64))
   (asm "addmax %src1, %src2, %dst")
   (encoding P (22 #b01111111111000010011100) src1 src2
      dst )
   (semantics
     (let*
         ;; Get the maximum register of src1 and src2
        ((src (rif (icmp 64 'sgt src1 src2) src1 src2))
         ;; Retrieve the flag-register that dictiates
            whether to saturate
         (saturation-flag (reg-ref dst 'saturation)))

        ;; If saturation-flag  set during configuration,
           use saturated add.
        (cif (icmp 1 'eq 1 saturation-flag)
             (set-reg! dst (satadd 32 dst src))
             (set-reg! dst (add 32 dst src)))))))
```

Listing A.16: *The example instruction after adding instruction semantics. The largest source register is first chosen. A saturation flag is retrieved to decide whether the addition is saturing or if not.*

those found in normal instructions. Listing A.17 shows an example where the functional unit and size of instructions have been abstracted to a base instruction, because so many instructions share these properties.

Unlike instructions, any property can be omittied, as long as sub-instructions extend their implementations with the missing properties. Similarly to macro functions, described in Section A.4.3, they are unhygienic in the sense that they may provide semantics without providing operands.

```
(define-abstract-instruction base-instruction
  ((size 32)
   (functional-unit (arith1) (arith2))
   (issue-slot (is1) (is2))))

(define-instruction add
  ((extends base-instruction)
   ...)) ;; omitted properties
```

Listing A.17: *The definition of an abstract base instruction. The size and functional-unit property is provided from this base instruction so the extending instruction need not do so.*

A.3 Describing Conditions

Conditions are described in a way nearly identical to describing instructions, except that the special-form is *define-condition*. Furthermore, unlike instructions, abstract conditions cannot be defined. Listing A.14 shows a typical condition that checks whether a general-purpose register is non-zero.

Compared to instructions, there are less properties to provide when defining conditions. *Size*, *encoding*, *operands*, and *semantics* are all valid attributes. Semantics need to return a numeric value, although side effects (such as register updates) can occur.

```
(define-condition reg-not-zero
  ((asm "[%cond-reg]")
   (size 16)
   (encoding P (9 #b1101001111) cond-reg)
   (operands (reg cond-reg r32))
   (semantics (icmp 'ne 32 cond-reg 0))))))
```

Listing A.18: *A condition that returns true if the only register operand is not zero.*

A.4 Miscellaneous Information

This section provides important features of Instant Toolsmith ADL that did not fit into previous sections.

A.4.1 Primitive functions

Primitive functions are used as building blocks by semantics to perform computations. Typical examples of these operations are arithmetic and program jumps. Besides common operations, custom primitive functions can be supplied. Two examples of custom primitives are shown in Listing A.19. The special form *define-primitive* takes three parameters. The first parameter is a boolean value that signifies whether the primitive returns a value. The second parameter is signature of the primitive function. The last argument is the name of the function in C source code, so that a generated simulator can utilize it.

A.4.2 Utility functions

Utility functions provide means for avoiding source code duplication. They are defined as typical functions that perform operations and optionally return a value. The same rules apply to utility functions as for instruction semantics. An example of a utility function is provided in Listing A.20.

Utility functions can make use of previously defined utility functions and macro functions, but recursion is disallowed.

```
(define-primitive true (load-byte addr offset)
   "llvm_ldb")
(define-primitive false (store-byte addr offset value)
   "llvm_stb")
```

Listing A.19: *A definition of two primitives which allows semantics to assume that these functions exist. The first definition will return a value while the second definition will not.*

```
(define-util-fun (swap-halves! register)
  "Swaps the lower 16 bits with the higher
   16 bits in a 32 bit register"
  ;; Need to do add because tmp should store a numeric
     value, not a register
  (let ((tmp (add 16 (reg-range register 16 31) 0)))
    (set-reg! (reg-range register 16 31)
              (reg-range register 0 15))
    (set-reg! (reg-range register 0 15) tmp)))
```

Listing A.20: *A utility function that swaps the lower and upper halves of a 32 bit register. Instructions defined after this utility function can make use of it. Utility functions require a documentation string.*

A.4.3 Macro functions

Macro functions help circumvent the strict rules associated with defining instruction semantics. With macro functions, any valid Scheme source code can be used as long as the functions return something that expands into valid semantics. Two examples where macro functions shine are now presented:

1. Semantics that are best described iteratively within a constant range can be defined in macro functions which will then unroll the source code.

2. Typically during saturating operations, register flags that signify overflow need to be updated based on the result of the operation before saturation takes place. This requirement, which is also needed by other types of flags, can utilize macro functions to avoid large source code duplication. A macro function that takes the operation, target register, and the relevant flags as input can perform the unsaturating version of the operation, update the flags and then perform the saturating operation.

Listing A.21 shows an example of a macro function that can be invoked from a instruction, condition or utility function.

```
(define-macro-fun (random-set-r5!)
  ;; Assume the random function is defined only in
     Racket.
  (let ((value (random #xdeadbeef)))
    ;; Expanded values are stored in an extra list.
    `((set-reg! r5 ,value))))
```

Listing A.21: *A macro function for updating register r5 to a random value between 0 and hexadecimal value deadbeef. The value will be randomized everytime the specification is loaded. The target register need not be provided to the macro function because the macro-system in Instant Toolsmith is unhygenic.*

```
(declare-functional-units
  (MAC1 MAC2 Load-Store FP-Arith))
```

Listing A.22: *Declaration of functional units for a hypothetical architecture. After being declared, they can be put to use in the definitions of instructions.*

A.4.4 Loading specifications

No form of recursion is allowed and specifications follow a flat module structure. They are loaded top-down similar to the old *load* mechanics in Scheme.

A.4.5 Functional units

The existence of functional units is declared in a simple statement, as shown in Listing A.22. Their functionalities and constraints are not described explicitly, but instead specified collectively in their usage, found in the definitions of all instructions.

A.4.6 Issue slots

Issue slots are used for VLIW packet ordering as briefly described in Section A.2.3. Instructions will be placed in packet slots in the order that

```
(declare-issue-slots
  (is1 is2 is3))
```

Listing A.23: *Example declaration of issue slots. This example allows for packets of three instructions, but only if the instructions themselves allow all the slots to be used.*

they appear in the issue slot declaration. Listing A.23 shows how issue slots are declared.