Instant Toolsmith: Assembler

Generating assemblers from formal architecture specification

MATTIAS JANSSON

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

(TRITA-ICT-EX-2015:78)

Abstract

This thesis explores the use of the domain specific language Instant Toolsmith ADL to generate assemblers. The goal is to show how Instant Assembler ADL can reduce the development time and maintenance cost of the toolchain for a processor
architecture.

Instant Toolsmith ADL is used to develop Instant Assembler, a tool that generates assemblers from formal specifications. Instant Assembler is developed especially with digital signal processors (DSP) in mind. The report focuses on architectural features common for DSP processors such as Very Long Instruction Word (VLIW) and conditional execution.

As a case study an assembler for the MIPS32 processor architecture is produced using the Instant Assembler and is then extended with hypothetical additional features that are commonly found in DSP processors. The architecture is thus turned into a much more sophisticated one with little effort.

The results show that the way of working from a good architecture description language provides the possibility to achieve fast development cycles and little maintenance overhead for processor toolchains. Specifically, it is shown that overall processor functionality as well as some features that are common in DSP processors are easily modeled using Instand Toolsmith ADL and used to produce architecture-specific parts of an assembler.

Referat

Denna rapport presenterar användandet av det domänspecifika språket Instant Toolsmith ADL för att generera assemblerare. Målet är att visa hur Instant Assembler ADL can minska utvecklingstiden och underhållskostnaden av verktyg till en processorarkitektur.

Instant Toolsmith ADL används för att utveckla Instant Assembler, ett verktyg som genererar assemblerare from formella specifikationer. Instant Assembler är utvecklat särskilt för processorer för digital signalbearbetning. Rapporten fokuserar på egenskaper som är vanligt förekommande hos sådana processorer, såsom långa instruktionsord och villkorliga instruktioner.

Som fallstudie utvecklas en assemblerare för processorarkitekturen MIPS32 med hjälp av Instant Assembler och sedan utökas den med nya, hypotetiska funktioner som ofta återfinns i processorer för digital signalbearbetning. Arkitekturen blir på så vis mycket mer sofistikerad med liten ansträngning.

Resultaten visar att tillvägagångssättet att utgå från ett bra språk för arkitekturbeskrivning ger möjligheten att få snabba utvecklingscykler och minskat underhåll för verktyg relaterade till processorarkitekturer. Det framgår att övergripande processorfunktioner och även vissa funktioner som rör digital signalbearbetning är lätta att modellera med Instant Toolsmith ADL för att sedan använda till att generera de arkitekturspecifika delarna av en assemblerare.

Contents

List of Algorithms

List of Figures

Listings

Chapter 1

Introduction

This thesis presents the Instant Assembler, a tool for generating assemblers from machine readable architecture descriptions.

The tool is designed for maintainability and adaptability. The intention is to show how compiler toolchains can be made more maintainable by separating the processor architecture from the tools themselves.

The language used to describe processor architectures is called Instant Toolsmith ADL, which has been designed by myself together with Karl Johansson and Frej Drejhammar. This thesis is part of a project aiming to produce several architecture dependent tools from one and the same specification. Karl Johansson presents a sibling project in [10].

1.1 Problem Description

Every modern processor architecture that has a compiler for a higher level programming language typically also has a range of tools designed particularly for that processor architecture. There are compiler backends, assemblers, disassemblers, linkers, simulators, debuggers and possibly more.

If one was to make a change to the architecture, corresponding changes need to be done to possibly every single architecture dependent tool. This redundancy has the risk of making toolchains error prone, hard to maintain, and hard to modify.

Automatically generating architecture-specific components of an assembler from a domain-specific language reduces the manual work and, more importantly, the time required to adapt the tools to architectural changes.

1.2 Desired Outcome

Based on the problem description. The thesis will focus on the following two aspects of the outcome.

Fast development cycles
: Some of the main motivations for basing a toolchain on an ADL describing architecture dependent aspects are maintainability and extensibility. It is important that changes to the tools can be done quickly and easily by changing the description.

ADL coverage
: The architecture-dependent aspects of the Instant Assembler should, to the highest extent possible, be generated from a specification. Any aspect that can not be expressed using the ADL will need to be manually implemented in each tool that depends on it. Therefore, high ADL coverage reduces duplication of information and should increase toolchain maintainability.

1.3 Methodology

An assembler generator tool, from hereon known as Instant Assembler is built. A case study is then performed where a specification of a simple processor architecture is written, and Instant Assembler is used to create an assembler from that specification.

The specification is extended with additional features that are not part of the original processor architecture in order to evaluate development cycle speeds and ADL coverage. The case study allowed for analysis of what it would be like to use Instant Assembler when developing tools for new architectures. Strengths and weaknesses of the Instant Assembler should become apparent during the extension process.

The case study provides the basis for the evaluation where the required effort of development and level of ADL coverage is examined.

1.4 Contributions

This thesis presents the following main contributions:

- Instant Assembler design. The Instant Assembler serves as an example of how a machine-readable specification can be used to create tools where architecture-dependent parts are generated rather than manually implemented.

- Instant Assembler evaluation. The case study gives an evaluation of the tool as well as the architecture description language. Some lessons learned and future work is presented based on the evaluation.

1.5 Ethics and Sustainability

When conducting the mandatory analysis of ethical aspects to this thesis, no issues of this work regarding ethics were found. Nor any issues regarding sustainability were found.

1.6 Document Structure

Chapter 2 explains what assemblers do and how they work. Chapter 2 also mentions some related work by introducing the most commonly used assemblers and the approach those solutions have taken to describe processor architectures.

An overview of the system design of Instant Assembler is given in Chapter 3, describing all relevant components of the generated tool.

In Chapter 5, a case study is conducted where a simple processor architecture is described using the architecture description language. Instant Assembler is used to generate an assembler from the description. The architecture is given additional features, and the resulting changes to the assembler will be analysed.

Chapter 6 will present the conclusions drawn after performing the case study.

The development of the architecture description language is done by the author together with Karl Johansson and Frej Drejhammar. Appendix A is an in-depth description of the language, co-authored with Karl Johansson.

1.7 How to Read This Document

This document contains many code examples, where much of the code is generated. It is important to separate hand-written from generated code because the former is part of the method while the latter is part of the results. The results that Instant Assembler produces are assemblers.

Code listings with a single frame show hand-written code, either hypothetical examples, architecture specification code or snippets from the implementation code.

Code listings showing generated code such as generated C code, generated parser rules and the like have a double frame.

Chapter 2

Background

The creation of executable files from high level source code is usually done in several steps, where each step in the process transforms a program to a version that is closer to what a computer can execute [9, 11].

This chapter explains what assembly of programs means and what exactly is done by the assembler. Section 2.1 starts from the beginning by scratching the surface of machine instructions and processors. The role of the assembler in the context of program compilation is explained in section 2.2. Section 2.4 describes what the assembly language is and why it exists. Section 2.5 explains what object code and object files are.

2.1 Processors and Instructions

Every computer program that can be executed directly on hardware can be seen as a long series of instructions. Once a program is loaded, a processor would read instructions from memory, one by one, and execute them. The program file itself consists of little more than those instructions, some static data, and some information instructing the operating system on how to load and start the program [11, Ch. 3].

Each instruction can be considered to have at least two parts. One is the operation, which specifies what computation the processor should do. Some examples of operations are addition, subtraction, multiplication, loading data from memory, or storing data to memory [9, Ap. A].

The other part of a typical instruction is one or more operands. The operands tell the processor on which data to perform the given operation and were to store the result. Operands can be values directly specified in the instruction (usually referred to as immediate operands), references to registers or references to memory locations [9, Ap. A].

2.1.1 Pipeline

The execution of each instruction is done in several phases. All phases together form a pipeline into which instructions are fed one by one. An example execution flow may look something like this:

1. **Fetch Instruction**
   Read and decode the next instruction in order to determine what to do.

2. **Fetch Operands**
   If any operands refer to registers, read the values stored in those registers.

3. **Read Memory**
   If any operands refer to memory locations, fetch data from those locations.

4. **Execute**
   Perform the actual computation.

5. **Write Back**
   Write the result of the calculation back to a register.

Most instructions will have to go through most of these phases and in the correct order. "Write Back" can not be performed before "Execute" has completed, "Execute" can not start calculating a result before the input data is available, and so on. In this particular example, most instructions would take five clock cycles at least to complete. To speed up the process, a typical processor would queue up several instructions so that there is an instruction in each phase as much of the time as possible. Think of it like an assembly-line in a car factory. An individual instruction may still take at least five clock cycles, but every clock cycle or so, an instruction gets done. This strategy is called a pipeline and is used in all modern processors [9].

The example above is a simple 5-stage pipeline. Pipelines in reality may be much more complex. For example, an Intel Core processor has a 14-stage pipeline [7].

2.1.2 Instruction Level Parallelism

In order to achieve faster program execution, most processors today can execute several instructions in parallel. One method to achieve such parallelism is called Very Long Instruction Word, from here on referred to as VLIW. The concept of VLIW is well described in [9]. Architectures featuring VLIW allow the program to, explicitly, specify bundles (sometimes referred to as packets) of instructions to be executed in parallel. The parallelism has thus been defined statically by a compiler or programmer. VLIW architecture is common in Digital Signal Processors such as Hexagon [2].

A common alternative to VLIW, often used in desktop computers is a *s*uperscalar architecture. A superscalar processor uses run-time analysis of the program state

and upcoming instructions to determine dynamically whether an instruction can be issued in parallel with previous instructions. Superscalar architecture can give significant speedup to program execution, especially if combined with dynamic instruction reordering, called out-of-order execution.

In the context of digital signal processing, the VLIW architecture has some advantages. Avoiding too much run-time analysis may increase power efficiency and material cost. Furthermore, digital signal processing systems often have real-time constraints. VLIW architecture combines the high performance of instruction level parallelism while making program execution time more predictable.

In both VLIW and superscalar architectures, the parallelism is made possible by having multiple parts of the processor capable of executing instructions. The processor parts that perform actual computations are called functional units [9]. In the case of VLIW architectures, the way bundles are ordered may depend on the mapping of instructions to functional units. To model this, one can say that a bundle has a set of slots, from here on referred to as issue-slots. When creating a bundle, each instruction has to fit in an issue-slot, or it will not fit in the bundle. The concept of issue-slots can help model two problems when creating instruction bundle. One being limiting the number of instructions so that there are enough functional units to execute them, and another being that instructions may have to be encoded in a certain order.

2.1.3 Conditional Execution

Every nontrivial program sooner or later needs to decide what code to run depending on some condition. Regular "if" expressions and non-infinite loops are dependent on conditional code execution, sometimes referred to as instruction predication. In many processor architectures, such as MIPS32 [6], only branching or jumping can be made conditional. That way, any instructions can be made conditional by placing a conditional jump just before them. Any loop can be terminated by jumping out of it, or just not jumping back to the beginning.

There are, however, drawbacks with introducing too many jumps in the code. Consider the pipeline in section 2.1.1. The pipeline is often full of instructions on their way to be executed. A jump instruction can change what instructions are supposed to be executed next. That is the whole point of the jump. When that happens, all subsequent instructions in the pipeline may have to be removed, and the processor will have to start reading instructions from another location. Several ways exist to mitigate performance loss caused by jumps and the pipeline [9], but the problem does not go away fully.

In the case of VLIW, jumps can cause even more performance loss. Two instructions on each side of a conditional jump can not be packed together since one does not know if the latter is going to be executed or not. By introducing a high number of jumps into the code, the blocks of instructions without jumps (basic blocks), become smaller and so also the chances of having good bundle opportunities.

VLIW architectures may, therefore, exploit conditional execution of more than

just jumps. If just about any instruction can be made conditional, then bundles can be created where some instructions will always be executed and others only sometimes. Also, if a conditional non-jump instruction does not get executed, the effect on the pipeline can be made rather small.

The Intel Itanium [8] is one example of a processor architecture using instruction predication. Each instruction has a few bits allocated for a predicate that causes the instruction only to be executed if the predicate holds.

2.2 From Source Code to Executable

The assembler is typically one in a chain of tools used to produce executable files from source code. In the common case, programmers would write their programs using a high level language such as C, which is then run through a compiler to produce assembly code. An assembler would then use the generated assembly code to produce object files, which are then linked together using a linker to produce a complete executable file. Figure 2.2 illustrates the process for a program of three source files.

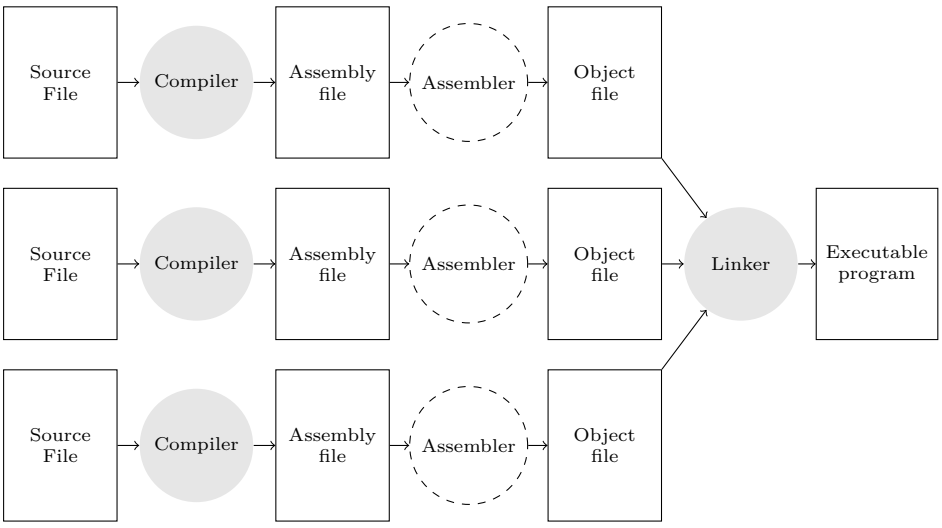

Figure 2.1. Illustration of the most common process of program compilation and the role of the assembler.

2.3 Typical Assembler Structure

The overall process of assembly can be considered a pipeline-like structure, where each step in the pipeline transforms input data to be used in the next step. The details of the process is described throughout this chapter. A simple overview is presented in Figure 2.3. If the input assembly code is represented by text it is parsed in an initial step in order to interpret the contents of the program. From the parsed program, a typical assembler determines which instructions, with what data and so forth is meant to be used. In Figure 2.3, this step is referred to as instruction selection. When the assembler has determined what the assembled program should contain, the program is translated into a binary format. This step is referred to as encoding. Finally, the assembled program is written to file. That step is referred to as object writing.

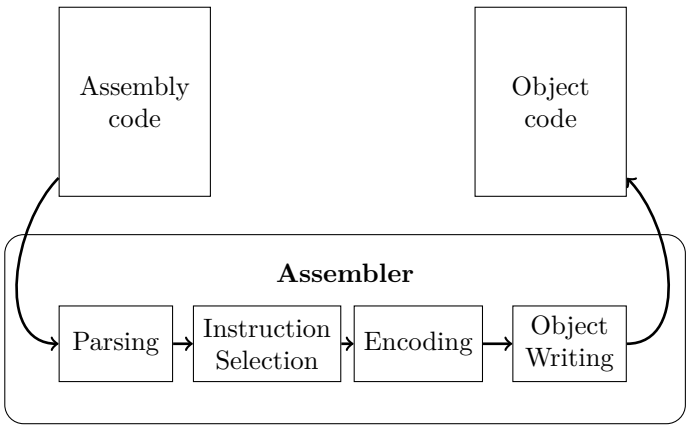

Figure 2.2. Basic structure of an assembler. Data is being transformed in steps through a pipeline-like structure. Input assembly code is read by a parser. Instructions and data are then determined during instruction selection. The binary version of the program is produced during encoding. An object file writer formats the resulting program on file.

2.4 Assembly Language

Assembly languages are typically dependent on the assembler being used as well as the processor architecture on which the program is intended to run. Most assembly languages are however quite similar and share a common set of features. Every assembly language has an instruction set, a way to define data areas, labels that enable the program to refer to specific points in the code and more. The mentioned features are described in this section to provide overview of how a typical assembly language works. There are usually many more features of an assembly language than described here. A good description of assembly language can be found in Reis [16]. To learn how to write full programs in assembly code, refer to the manual of the assembler and the processor that is to be used.

2.4.1 Instructions

The instructions are a list of operations telling a computer what to do, when to do it, and with which data to do it. Each processor architecture has its set of instructions that it can perform, from here on referred to as the instruction set. Some instructions in an assembly file could look something like Listing 2.1:

These instructions could mean to tell the computer to:

1. Store the value 12 in register r1
2. Store the value 14 in register r2
3. Add the value stored in r1 to the value stored in r2, and store the result in register r3

2.4.2 Labels

Labels are a way to name certain points in the program in order to refer to them. For example, Listing 2.2 shows one way to calculate the factorial of 3.

```
1 mov r1, 12
2 mov r2, 14
3 add r3, r1, r2
```

Listing 2.1. Example section of assembly code

```
1 mov r1, 3       ; Store the value 3 in r1
2 mov r2, 1       ; Store the value 1 in r2
3 loop:           ; A label
4 mul r2, r1, r2  ; Multiply r1 and r2, store result in r2
5 sub r1, r1, 1   ; Subtract 1 from r1, store in r1
6 jnz r1, #loop   ; If r1 is not zero, go to the ''loop'' label
```

Listing 2.2. Example section of assembly code with label and a jump instruction

Having a computer execute these instructions would result in the following execution scenario:

1. Store the value 3 in register $r1$
2. Store the value 1 in register $r2$
3. Calculate $3 \cdot 1$ and store the result (4) in $r2$
4. Subtract 1 from r1 and store the result (2) in $r1$
5. $r1$ does not contain the value 0, jump back to "loop"
6. Calculate $2 \cdot 3$ and store the result (6) in $r2$
7. Subtract 1 from $r1$ and store the result (1) in $r1$
8. $r1$ does not contain the value 0, jump back to "loop"
9. Multiply $1 \cdot 6$ and store the result (6) in $r2$
10. Subtract 1 from $r1$ and store the result (0) in $r1$
11. $r1$ contains the value 0. Done

Labels can be used, as shown above, to express loops. They can also be used to define functions. Functions in assembly code are basically a piece of code with a label before it so that the function can be accessed from other parts of the program.

There may be more actions that need to be taken in order for the caller and callee to not overwrite each other's register data. How that is done is a matter of convention.

2.4.3 Data

The data section in an assembly file contains all the statically defined data in the program. In C, this would be all the global constants. The data is stored in the file quite similarly to the instructions and labels can be used much in the same way. To refer to a piece of data stored in the data section, just use a label.

2.5 Object Code

The output of an assembler is typically object code, which can be viewed as separate parts of a final executable. Instructions and data are encoded and aligned so that a target processor can run it, and enough information is included so that a linker can link object files and static libraries into a complete executable program. The format of an object file is dependent on the operating system that is meant to load and execute it, but most have a few concepts in common. Object files and many object file formats are covered in depth in [11].

2.5.1 Symbols

When producing an object file, the assembler usually resolves and encodes as many labels and expressions as possible to actual numerical addresses or offsets [11, Ch. 1]. References to labels in other files can however never be resolved by the assembler since files are assembled individually. Resolving such expressions is one of the jobs for a linker.

Any location in an object file that other files should be able to refer to are put in a symbol table, along with all symbols the file cannot resolve on its own [11, Ch. 5]. One can call them imported and exported symbols. During linking, symbol tables from several files can be compared so that an exported symbol in one file can be matched to an imported symbol in another one. When linking several files into one, those symbols can then be resolved to addresses within the file [11, Ch. 5].

2.5.2 Relocation information

When writing an individual object file, the assembler does not know how the final executable will be linked. Any code reference within the object file is likely to be wrong when multiple files are linked together [11, Ch. 3].

For example, assume there is a jump instruction in one object file that jumps to address 4. Assume that all the object code in that file is placed 40 bytes into the combined file. If the jump instruction still jumps to address 4, it will be off by 40 bytes.

The linker has to take care of this problem by adjusting all code references in all files so that they still refer to the same thing, even when the code has moved. This task is called relocation [11, Ch. 3].

The problem is that the linker can not know what is a code reference and what is not. At this point, there is not any difference between the address 4 and the number of Ninja Turtles. It is just a value. The linker needs a way to find out that the value is a code reference. The assembler preserves this information by finding all the points in the code that will need relocation. That is then stored in the object file. In the example above, the linker just needs to read the relocation information and add 40 to each value marked for relocation.

In many operating systems, the code will need to be relocated again when being loaded into memory for execution [11, Ch. 2], but that is outside the scope of this thesis.

2.6 Parsing and Parser Generators

In order to manage and transform assembly code, the first step for an assembler is to read and interpret the textual code. This section briefly explains the steps necessary for a program to parse text in general. The focus will be on how to use a parser generator to do this. To give a better understanding of the concepts, we will use a simple example language. The example language accepts a single mathematical

```
1 2 + (3 − 4) + 5
```

Listing 2.3. A valid expression in the example language

```
1 INT PLUS LPAREN INT MINUS INT RPAREN PLUS INT
```

Listing 2.4. Lexical analysis of the expression in Listing 2.3

expression using integers, addition, subtraction, and brackets. Whitespace is ignored. The expression in Listing 2.3 would be valid in this language.

2.6.1 Lexical analysis

Before the contents of a machine-readable text is examined more closely, a parser normally performs a first pass of the text to identify the lexical structure. The lexical analysis identifies numbers, text, symbols, and other constructs and breaks them into a list of tokens. The token list for the expression above is shown in Listing 2.4.

A program that performs this tokenization, from here on referred to a lexer, can be generated using a lexer generator. In this thesis, Flex [17] is used as the lexer generator. Listing 2.5 gives an example of the input to Flex for the example language.

Lines 2-3 are regular C code. The generated lexer will be a C file, and the code put there will end up at the beginning of that file. Lines 6-10 list some definitions that will be used. The definitions are regular expressions that the lexer will attempt to match to the input text. Lines 14-24 are the rules. Each rule consists of a pattern and an optional action. The pattern can be a regular expression formed with the use of the definitions. The action is C code defining what the lexer is meant to do when a rule matches. The variable "yytext" is always set to refer to the text that matched the expression. The "yylval" structure is generated by the parser generator and is described in Section 2.6.2. Each "return" statement returns a token, which is also described in Section 2.6.2.

2.6.2 Syntactic analysis

When the lexical analysis has transformed the input text to a series of tokens, the next step is syntactic analysis, or parsing. There are several parsing algorithms used today of varying complexity. As in the case of the lexer, the parser can normally be generated using a parser generator. The parser generator used in this thesis is Bison [4] which is a YACC-compatible [16] parser generator. Bison mainly generates parsers that implement the LALR(1) [16, 3] parsing algorithm. The parsing algorithm itself is outside the scope of this thesis.

```
1  %{
2  #include <math.h>
3  #include ''example_lang.tab.h''
4  %}
5
6  LPAREN ''(''
7  RPAREN '')''
8  PLUS ''+''
9  MINUS ''-''
10 DIGIT ''[0-9]''
11
12 %%
13
14 {DIGIT}+ {
15   yylval.integer = atoi( yytext );
16   return INT;
17 }
18
19 LPAREN { return LPAREN; }
20 RPAREN { return RPAREN; }
21 PLUS   { return PLUS; }
22 MINUS  { return MINUS; }
23
24 [ \t\n]+          /* Ignore whitespace */
25
26 %%
```

Listing 2.5. Flex input file for the example language

To specify the language to be parsed to Bison, a context-free grammar [16] is used. The context-free grammar defines which constructions of tokens are valid and what they represent. A grammar file like Listing 2.6 is used with Bison to generate a parser for the example language.

The structure of this input file resembles the input for Flex. Lines 2-3 is C code. In this case, we assume that there is a header file called "intermediate_representation.h" which defines the struct "expression_t" and the enum "operator_t". Lines 6-10 specify the data structure of the parsing result. It should be a union of all types of data the grammar will match. Lines 12-13 define all the non-terminal types, which means types that are built up from other types. The type of data returned when those types are matched are specified between the '<' and the '>'. They refer to elements in the union. Lines 15-19 define terminals, which are types that can be deduced directly from tokens from the lexer. Data types are specified as for non-terminals.

Lines 25-38 are the rules. As one can see, the rules are constructed using terminals and non-terminals. The action following the rule will be taken when a rule matches. In the case of the first rule, the rule matches when there is any expression followed by an operator and another expression. The action taken will be to allocate a new "struct expression_t" element, assign the "left" member to the leftmost expression, the "op" member to the operation and the "right" member to the rightmost expression. Any input containing addition or subtraction will this

```
1  %{
2  #include <stdlib.h>
3  #include ''intermediate_representation.h''
4  %}
5
6  %union {
7    int integer;
8    struct expression_t *expr;
9    enum operator_t op;
10 }
11
12 %type <expr> expression
13 %type <op> operator
14
15 %token LPAREN
16 %token RPAREN
17 %token <op> PLUS
18 %token <op> MINUS
19 %token <integer> INT
20
21 %left PLUS MINUS
22
23 %%
24
25 expression : expression operator expression
26                 {
27                     $$ = malloc(sizeof(struct expression_t));
28                     $$.left = $1;
29                     $$.op = $2;
30                     $$.right = $3;
31                 }
32             | LPAREN expression RPAREN
33                 { $$ = $2; }
34             | INT
35                 { $$ = $1; }
36
37 operator : PLUS { $$ = op_plus; }
38          | MINUS { $$ = op_minus; }
39
40 %%
```

Listing 2.6. Grammar file for Bison to generate a parser for the example language

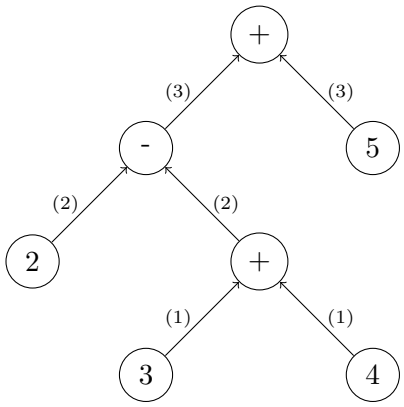

Figure 2.3. An abstract syntax-tree as it would be constructed by the example parser generator with the input "2 - (3 + 4) + 5". The semantic meaning of this abstract syntax tree could now be interpreted by traversing the tree recursively. A natural interpretation of the tree would be the following: **1:** the values 3 and 4 are to be added together. **2:** The result of the addition is subtracted from the value 2. **3:** The value 5 is added to the result of the subtraction. The steps are marked out by the arrows in the figure.

way form a tree-like structure, usually referred to as an abstract syntax-tree.

One very important line has not been mentioned yet. Take a look at line 21. That line specifies what the parser should do when it, for example, encounters an input of the form "A - B + C". Looking at the grammar above, the parser will either match "A - B" first and construct an expression of that which is then used as the leftmost expression of the "+" operator, or the other way around. In short, it will either parse it as equivalent to "(A - B) + C" or equivalent to"A - (B + C)". The latter is not what we intended, so we must instruct the parser to match the leftmost expression first. One usually refers to such tokens as left-associative. Line 21 defines the plus and minus tokens as left-associative.

Figure 2.6.2 shows what the abstract syntax-tree would look like if the parser was given the input "2 - (3 + 4) + 5".

2.7 Commonly Used Assemblers

The goal of this thesis is to develop a solution for the quick development of assemblers for new processor architectures and adapting to change with low overhead. This section briefly mentions two of the most commonly used assembler solutions and describes the way the two handle architecture-specific information.

2.7.1 GNU Assembler

The GNU Assembler (GAS) is the assembler of the GNU Binutils [5] tool set. Although supporting a wide range of processor architectures, GAS does not generate

code from any architecture description language. In order to support a new processor architecture, one needs to hand-write the architecture specific parts.

2.7.2 LLVM MC

The LLVM project [14] is a large collection of compiler and compiler-related technologies that are meant to be used to create tools. The Clang [12] compiler is one example of a tool created using LLVM.

LLVM Machine Code (LLVM MC) is a subproject of LLVM that handles assembly, disassembly, and other tasks dealing with machine and assembly code. LLVM MC can generate much of the architecture-specific code using TableGen [13] specifications.

TableGen in itself is a flexible language for expressing different types of domain-specific information. It is mainly a way for developers to define records of information. The actual meaning of the information defined is determined by the entity reading it.

LLVM MC can use TableGen records for generating some instruction specific information, such as instruction syntax, operands, and some encoding information. However, not all architecture-dependent information can be captured using the TableGen specifications for LLVM MC. In order to implement a somewhat complex architecture, quite some manual code needs to be written, both for assembly code parsing and for instruction encoding.

2.8 Instant Toolsmith ADL

The Instant Toolsmith ADL is developed by Karl Johansson, Frej Drejhammar and the author in the Ericsson-sponsored Instant Toolsmith project. The language is originally intended to support the description of DSP architectures and, therefore, has built in support for VLIW and instruction predication. The Instant Toolsmith ADL is the language that is used in Instant Assembler. Appendix A describes the language in more detail.

Chapter 3

System Design

This chapter presents the design decisions made when developing the Instant Assembler. Section 3.1 briefly goes through the overall structure of the generated assembler. The rest of the chapter describes in more detail how each component works.

3.1 Overall Structure

The assemblers generated by Instant Assembler can be thought of as a set of components operating at a certain stage of the assembly process. These components and their relations are illustrated in Figure 3.1. First, the input assembly code is fed to the lexer for lexical analysis. Tokens from the lexer are then passed on to the parser. See sections 2.6.1 and 2.6.2 on what that implies. The parser will produce an in-memory representation of the program. In this representation, many of the instructions have not been fully decided. The matcher will inspect instruction operands and decide on the actual instructions to use. When the instructions have been selected, the bundle arranger will check what issue slots the instructions in each bundle can be placed in and order them accordingly. The ordered bundles are then sent to the instruction encoder that computes the binary representation of the instructions. Once all instructions are encoded, the object writer can produce the actual object file.

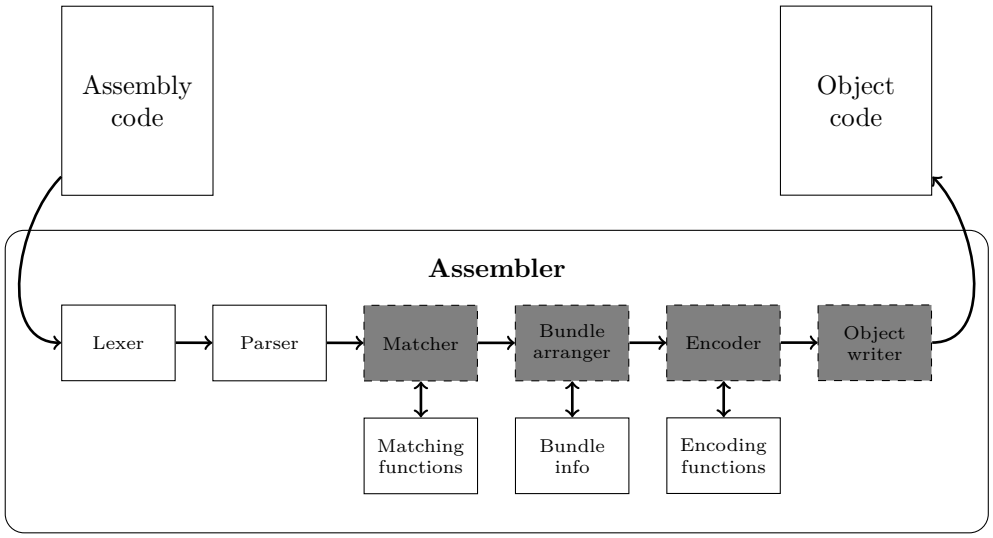

Figure 3.1. The main components of an assembler generated by Instant Assembler. The gray rectangles with dashed frames represent components that are not generated and thus should be as architecture independent as possible. The brown rectangles with solid frames represent components generated from ISD. Arrows show the general application flow.

3.2 Lexer

The first step in the assembly process is the lexical analysis as described in Section 2.6.1. In order for the lexer to read an assembly file, the token specifications for it need to be generated from the ISD.

Many tokens such as integer literals and other common lexical structures do not need to be generated. The same goes for the general structure and much of the syntax of the token specification. They are independent of the processor architecture. The set of register names are, however, different for each processor architecture and do need to be generated from the ISD. The same goes for instruction syntax.

A template is used that contained common code and placeholders for architecture-dependent code. When Instant Assembler is run, it traverses all register names and the instruction syntax of every instruction. Any unique token encountered in the ISD that consists of letters and numbers are collected and put into the token specification.

It is important that every generated token be unique. If any one occurs twice, it may be recognized as different tokens when reading assembly files. Tokens are easily kept unique by first processing the entire instruction and register set, continuously removing any duplicates, and then writing them to the token specification.

3.3 Parser

The syntactical analysis of the assembly code is in part hand written and in part generated from the ISD. The parts of the language that are not architecture-dependent are taken directly from an existing assembler. Examples of such parts are section dividers, comments, mathematical expressions, labels, and more.

The syntax of architecture-dependent parts of the language such as registers and instructions are generated by Instant Assembler. This section describes the method of generating the productions for registers and instructions.

3.3.1 Parsing registers

The instruction specifications in the ISD specify which register class a certain register operand must be part of to be valid syntax. A first version of the parser would output one grammar production for each register in each register class. When the parser expected a certain register class, it would automatically verify that the next token is a register of that class.

This method works well when describing MIPS32 because there is no need for registers that occur in more than one register class. For more complex architectures, registers often need to belong to several register classes. When that happens, the parser can not be sure which production to match the register token. This is a form of reduce-reduce conflict [4].

With the help of Frej Drejhammar, the grammar generator was changed first to find all register classes that have an overlap of registers. Each set of register classes that have such an overlap would be grouped together into what one may call a virtual register class, removing the ambiguity in the grammar. The drawback of this method is that the parser can no longer be certain whether an encountered register is in the correct register class. That check is therefore left to the matcher to do.

3.3.2 Parsing instructions

The instruction productions are generated using the assembly syntax field in the ADL. As described in Section 3.2, all sequences of letters and numbers in all registers, conditions, and assembly syntax fields have been collected in a set of tokens. Some other constructs, such as immediate values, parenthesis, and some commonly occurring symbols are put in the grammar by hand. Register operands are recognized as described in Section 3.3.1.

Generating the grammar production for an instruction is done by analyzing the assembly syntax string of the instruction description and constructing a production of the statically defined tokens and the nonterminals of each operand.

Much in the same way as in Section 3.3.1, this method has the risk of introducing reduce-reduce conflicts. If the syntax of each instruction is unique, it will work fine. There are however cases where two or more instructions can have the same syntax.

A common case for this to happen is jump or branch instructions. It is common for a processor architecture to have several branch instructions, each of a different size. The reason for this is to allow the assembler to choose a smaller instruction when the operand value is small, but have the possibility to have larger values by switching to a larger version of the instruction.

The is also very similar to Section 3.3.1, and also done with the help of Frej Drejhammar. The grammar generator will first check the syntax of all instructions. If any set of instructions has the same syntax, they will only result in a single production in the grammar file. When the parser hits an instruction production, it will return a set of all instructions that match that production. These instructions are assumed to be semantically equivalent and interchangeable. If they were not, then the assembly language itself would be too ambiguous to parse.

3.4 Matcher

The parser will not always be entirely sure which instructions can be used and not. For example, every immediate operand has boundaries for what values they may take. The parser does not check such boundaries. Also, as described in Section 3.3, the parser may not be sure about what instruction to pick, or whether a register is in the right register class.

The matcher checks this and removes incompatible instructions by calling the matching function for each possible instruction. The matching function will check that all operands are compatible and reply back to the matcher that will select an appropriate instruction to use.

A noteworthy detail is that labels, which refer to a certain point in the program, may be resolved to a different value depending on the size of the instructions around it. If the matcher switches some instruction from a smaller to a larger one or vice versa, the label may need to be reevaluated. Any instruction that refers to a reevaluated label may then become incompatible, or another instruction may become a better pick. The matcher must be aware of this and run the matching functions iteratively until all instructions, and all labels converge.

3.5 Bundle Arranger

Bundle arrangement is the process of reordering instruction bundles in such a way that they conform to the order expected by the hardware. To model the order that the hardware expects, the ISD has the concept of issue slots. Issue slots are named positions in a bundle that may be occupied by at most one instruction. The assembler is responsible for assigning issue slots to instructions and rearranging bundles so that the order of the instructions for each bundle conform to the order of the issue slots they use.

Depending on the hardware architecture, different orderings of some instructions in a bundle may not be semantically equivalent. For example, the architecture might

support bundling of several stack operations (push, pop) with semantics implied by their order. In that case, it would be the responsibility of the programmer or compiler to decide the order of instructions in the bundle. It is important not to reorder such instructions in order to retain the semantics of the program.

One way to achieve that is to have the ability to define instructions in such a way that the assembler never reorders them. That method could, however, be a bit too restrictive. One may want to reorder them with respect to other instructions, but not each other.

In practice, it often makes little sense for this type of instructions to support different sets of issue slots. For example, defining a push instruction to support issue slot 1 and 2 and pop to support 2 and 3 would imply that a pop can never come before a push. If two or more instructions should retain whatever order they were in before reordering, they are likely to support the same set of issue slots. A simple solution is to make sure that any instructions supporting the same set of issue slots are never reordered with respect to each other. The solution comes with an inherent constraint: instructions that must not be reordered with respect to each other must support the same set of issue slots, or never be in the same bundle. This constraint does not seem to be very limiting in practice.

The following conditions give a more formal definition of what to consider a correct bundle ordering given the accepted limitations:

Precondition:
All instructions in a bundle must be assignable to an issue slot.

Postcondition 1:
All instructions have an issue slot assigned.

Postcondition 2:
Any two instructions, which have the same set of possible issue slots, must retain the order in which they occurred before reordering.

3.5.1 Naïve greedy bundle arrangement

The simplest and most obvious way to perform bundle arrangement is a greedy selection algorithm that, for each issue slot, iterates over the bundle and selects the first possible instruction for that bundle. If no instruction is found, that would mean that the slot will be unused, and the algorithm continues. The pseudocode in Algorithm 1 describes this algorithm:

This implementation was considered for implementation, but will not work. Consider the following:

Assume two instructions, **A** and **B**. Assume instruction **A** can use issue slot 1 or 2 and that **B** can use issue slot 2 or 3. Now, if there is a bundle that contains the instructions **B A A**, then the second instruction, (**A**), will be placed on issue slot 1. Then, the instruction (**B**) will be placed in issue slot 2, and there will be no viable issue slot left for the third instruction.

Algorithm 1 Naïve greedy bundle arrangement algorithm

```
for all slot ∈ NewBundle.slots do
    for all instr ∈ Bundle do
        if instr can use slot AND instr.placed == false then
            slot.instr := instr
            instr.placed := true
        end if
    end for
end for
if not all placed then
    fail
end if
```

If the second instruction had been placed in issue slot 3, then the bundle could have been formed correctly. The precondition holds but not postcondition 1. This algorithm can therefore not be considered correct.

3.5.2 Greedy arrangement with priorities

This version is a simple expansion of the naïve greedy bundle arrangement. In the previous example, one could give instruction **A** a higher priority than instruction **B**. Then, **A** would be placed before **B** if possible, and the ordering would be correct for that particular case.

This extension will, however, not solve the general case without constraints. The following example shows one potential problem with this:

Assume three instructions, **A** and **C**. **A** can use issue slot 1 or 2. **C** can use issue slot 1 or 3. If there is a bundle that contains the instructions **A C C**, they should be arranged as **C A C**. Giving **C** a higher priority than **A** will give this behavior. However, consider a bundle that contains the instructions **A C A**. If **C** has higher priority than **A**, then **C** will be placed first, and the subsequent instructions would not fit in the remaining slots. In this case, **A** should have had higher priority than **C** to give a correct order.

The problem here lies in the gap in the set of possible issue slots for instruction **C** (more specifically, the possibility to use issue slot 1 or 3 but not 2). As seen here, that gap may cause instructions to need different priorities depending on context.

The author is not aware of cases where a priority can not be determined if there are no such gaps.

Another limitation to this is that it does not capture arbitrary context dependent ordering rules, for example, rules such as: "**B** should be placed before **A**, but only if there are no other instructions". That type of rule will not be captured using this solution.

If there are no issue slot gaps and no arbitrary arrangement rules, this method is sufficient. The author is not aware of any VLIW architectures where such gaps exist.

3.5.3 Search-based algorithms

If one needs to be able to express more complex arrangement rules in a general manner, a search-based approach that explores more possible arrangements could provide more flexibility. It could solve arrangement problems even with gaps and arbitrary ordering rules.

These methods were however deemed outside the scope of the project since the priority based approach meets the requirements of any VLIW architecture encountered by the author.

3.6 Encoder

The encoding process can be seen as a process on several levels with the highest level being the program, and the lowest level being the encoding of an individual operand. When the encoder is started, it will iterate over the entire list of instructions and bundles in the program and invoke encoding functions for each one. As shown in Figure 3.1, the encoder itself can be made architecture-independent. The encoding functions that encode particular instructions or operands can not. They are generated from the ISD.

3.6.1 Bundle encoding

The encoding of one bundle is almost as simple as encoding all the instructions within it. In the encoded output, the bundle will consist of the encoded version of each instruction, with some information that they belong to the same bundle. In the current solution, the only supported way to encode this information is using the bundle flag. How to use the bundle flag is shown in Appendix A.2.6.

The bundle flag is a single bit that is allocated in each VLIW instruction. If the flag is set, it means that the instruction belongs to the same bundle as the instruction before it. To encode a bundle, all one has to do is to clear the bundle flag of the first instruction and set all the others. The instruction encoder will point out to the bundle encoder where that bit is. The bundle encoder can then set the flags properly.

Since this is currently the only supported bundle encoding mechanism, it is hand-coded into the encoder, except for the position of the flag, which is defined by the ISD.

3.6.2 Instruction encoding

Each instruction is encoded into a predetermined number of bits, as defined by the ISD. Some of the bits of a particular instruction are fixed and will always be the same for that instruction. Other fields are set by the encoding of the instruction's operands. Appendix A.2.6 describes this in more detail.

For each instruction defined in the ISD, there will be one generated encoding function in the assembler. The encoding function will set the values of all fixed bits as defined in the ISD, mark out the position of the bundle flag and call encoding functions for each of the function's operands. When instruction encoding function is done, the result will be a bit-string representing the encoded version of the instruction.

3.6.3 Operand encoding

There are several types of operands that an instruction can take. There are registers, immediate values, labels, and conditions. Each of them has its encoding mechanism. The instruction encoder should call the operand encoder for each operand. The resulting value is then used by the instruction encoder to set the correct bits of the instruction.

Registers

A register may be part of several register classes as described in Appendix A.1.6. The encoding of one register is determined by the register class specified by the instruction. For example, assume an instruction that takes a single operand and the description specifies that the operand the instruction takes must be a register of register class **C1**. When encoding such an instruction, the register **R1** is encountered. Register **R1** is a member of several register classes, but the encoding of the register for this particular instruction must be the encoding specified in register class **C1**.

The generated instruction encoder will know this from the specification of the instruction. The instruction encoder must call the encoder of that register class. Each class' register encoder is in turn simple. It is just a mapping from a register ID to a bit-string as defined in the ADL.

Immediate operands

The type and value of an immediate operand are set during parsing, and the encoding of them is assumed to be the same regardless of processor architecture. When encoding an immediate operand of a given type, the only parameter that differs from instruction to instruction is the width. That is, the number of bits that the operand should take up. The width of an immediate operand is defined in the instruction's description in the ISD.

Labels

Labels are similar to immediate operands. In fact, when an instruction is being executed, it will be an immediate operand. Remember from Section 2.5 that the assembler tries to resolve as many labels as possible to numeric values. If a label has been resolved to a numeric value, it should be treated as an immediate operand. If

it has not, the encoding can not be determined. Putting the correct value in there is then the job of the linker (or loader).

3.6.4 Condition encoding

As described in Appendix A.3, a condition has quite a few things in common with instructions. They both have an assembly syntax, a predetermined size, an encoding description that is very similar to that of the instruction, and zero or more operands.

The encoding of conditions is thus done much in the same way as encoding of instructions. A noteworthy detail is that conditions, just as instructions, have a bundle flag. It does however not have the same meaning in the case of conditions.

3.7 Object Writer

The object writer is the component that produces the actual output of the assembler, the object files. Section 2.5 describes what object files are and what they contain.

The object writer is mainly dependent on the target operating system and the environment. Each object file format would require a different object writer. This thesis covers the processor architecture aspect of the assembler as opposed to the operating system aspect of it. Implementation of the object writer is thus outside the scope of this thesis.

Existing assemblers usually have object writers for most of the established formats, and object files do not change as often as processor architectures do. The easiest way to produce object files today would be to use existing implementations.

Chapter 4

Implementation

This chapter describes how the Instant Assembler is implemented. The two programming languages that are mainly used for this thesis are C [1] and Racket [15]. The generated output of Instant Assembler is C code. Racket, a language in the Lisp family, is used for generating said C code.

4.1 Separating the General From the Generated

Some components of an assembler, as described in 3, are not architecture specific. That notion gives a good opportunity to reuse such components from an existing assembler. The implementation of Instant Assembler is based on a manually implemented assembler provided by the telecom company Ericsson.

Any components of the assembler that are general enough to be used for any processor architecture, such as the object writer, are kept as is. Others are modified to separate out any code that should be generated. For example, the instruction encoder is changed so that it finds and calls the generated encoding functions instead of using the existing, hand written, encoder.

Figure 3.1 gives some idea of how some components are divided into a general part and a generated part.

4.1.1 Lexical and Syntactic Analysis

The lexers and parsers that Instant Assembler produces are generated using Flex and Bison respectively (Sections 2.6.1 and 2.6.2). More specifically, the parts of the Flex and Bison input files are generated by separating the general assembly language constructs (labels, arithmetic expressions, etc.) from the specific ones (instructions, registers, etc.). To combine the two, a template engine from the Racket standard library is used where general parts are written directly into the template with placeholders for the generated parts to be inserted.

4.2 Existing ADL Parser

A tool for parsing the Instant Toolsmith ADL into an in-memory database was created prior to this thesis. It was written largely by Karl Johansson with small modifications by Frej Drejhammar and the author.

4.3 Code Generating Scripts

The parser that was used allowed for the architecture description to be loaded as an in-memory database in the Racket virtual machine. The database is then available for any racket program to be loaded and gain access to the data.

All generated parts of the Instant Assembler is done so by loading the description into the Racket virtual machine and then running a code-generating script. Typically there is one script for each major generated part. Once all scripts have been loaded and run, the newly generated assembler is ready to be compiled by any C compiler.

Chapter 5

Case Study: Extending a simple processor architecture

One of the reasons for generating toolchains from machine-readable specifications is that it gives flexibility and fast development cycles. This speed and flexibility can, in turn, provide good input for architectural decision-making. Evaluating a small change or extension to an architecture could be done by making a change to the specification, generating the entire toolchain, producing new binaries and evaluating them.

This chapter serves as a more detailed description of how the Instant Assembler works as well as an evaluation of the method. The MIPS32 architecture has been selected as a base architecture due to its simplicity and popularity.

5.1 Evaluation Methodology

In order to evaluate and demonstrate the properties of Instant Assembler, the MIPS32 architecture is used as a starting point, and additional features to the architecture are added throughout the chapter. One can see this as an experiment showing how one could go about trying to find out what it would be like if MIPS32 are turned into a DSP. The evaluation focuses on the following aspects:

Fast development cycles
: The amount of effort required to make a change to a processor architecture and produce new tools in order to test it is one of the major motivations for the project. The development cycles are evaluated by the amount of effort required to do changes and extensions.

ADL coverage
: One goal of the project is to be able to specify as many aspects of a processor architecture as possible in a machine-readable specification. The less code that needs to be written by hand, the less amount of effort it should take to

create, maintain, and revise a toolchain. A goal of the case study is to expose some limitations of the method.

5.2 Implementing a Standard MIPS32 Assembler

The starting point for the case study is a standard MIPS32 assembler. A specification of most of the MIPS32 instruction set is written, and an assembler generated from that. This section presents the steps of the process and shows some of the resulting generated code. The language used to describe the architecture is described in more detail in Appendix A.

5.2.1 Defining registers

The set of MIPS32 registers is rather simple. There is almost no partially overlapping registers, and the register sizes are always 32 bits, except for a single 64-bit register. However, different MIPS32 assemblers name registers differently. For example, register 'r1' is sometimes referred to as register 'at'. In this case, both name schemes are implemented. Implementing both name schemes is easily done by full register overlap, which means that they are defined as two registers but they share the same bits. There is also an explicit alias feature in the Instant Toolsmith ADL if, for some tool, the implicit alias by register overlap is not enough. Listing 5.1 shows how some of the registers are defined. The other registers are defined very similarly to these. Since one register may be part of several register classes, and encoded differently depending on the instruction, the encoding information is found in the register classes. In the case of MIPS32, it so happens that one register is always encoded the same way, so the register classes are few. Listing 5.2 shows a part of the register class that is used for the standard registers.

```
1  (define-abstract-reg rX ((size 32) (access "rw")))
2
3  (define-reg temp ((asm "temp") (size 64) (access "rw")))
4
5  (define-reg r0   ((extends rX) (asm "r0") (access "r")))
6  (define-reg r1   ((extends rX) (asm "r1") (overlap (temp 32 63))))
7  (define-reg r2   ((extends rX) (asm "r2")))
8  (define-reg r3   ((extends rX) (asm "r3")))
9
10 ;; -- REGISTER ALIASES --
11
12 (define-reg zero ((extends rX) (overlap (r0 0 31)) (asm "zero")))
13 (define-reg at   ((extends rX) (overlap (r1 0 31)) (asm "at")))
14 (define-reg v0   ((extends rX) (overlap (r2 0 31)) (asm "v0")))
15 (define-reg v1   ((extends rX) (overlap (r3 0 31)) (asm "v1")))
```

Listing 5.1. A portion of the MIPS32 register set definition. The 'temp' register is the only one that is not 32 bits wide, and register 'r1' is the only register that partially overlaps another one. Register aliases are implemented by full register overlap. Common properties are defined in an abstract register 'rX'.

```
1 (define-reg-class r32 ((r0    (encoding (5 #b00000)))
2                        (r1    (encoding (5 #b00001)))
3                        (r2    (encoding (5 #b00010)))
4                        (r3    (encoding (5 #b00011)))
5 ...
6 ))
```

Listing 5.2. A portion of one of the MIPS32 register classes. The register class definition shows the size and encoding of each register in the class, and gives a name to the class for reference in the instructions.

5.2.2 Defining instructions

As in the case of registers, the instruction set of MIPS32 is rather simple. Most instructions share several properties. For example, they are all 32 bits wide. Common properties such as these make the abstract instructions in Instant Toolsmith ADL very useful. Listing 5.3 shows the definitions of a few instructions. Only properties that are of value to an assembler have been defined, except for 'meaning' which is added for readability. Refer to Appendix A for a description of all properties that can be defined using the Instant Toolsmith ADL. Noteworthy details of listing 5.3 are the connection between the 'asm' field that defines the assembly syntax, and the 'operands' and 'encoding' fields which use the same identifiers for operands. The sizes and values of the encoded register operands are determined by the register classes that they belong to, as defined in the 'operands' section. In the instructions shown, all registers are defined to be part of the r32 register class which is partially shown in Listing 5.2. The sizes and values of encoded immediate values and address operands are defined by their range of possible values. The immediate operand in the 'ADDI' instruction will be recognized as a 16-bit signed integer.

5.2.3 The lexer

The first part of the assembler that is generated is the lexer. The Instant Assembler will generate a Flex input file that will then be fed into Flex in order to generate the lexer. Refer to section 2.6.1 for some more details on how this works. Listing 5.4 shows a couple of sections of the architecture specific parts of the Flex input. Each rule is prepended by a *start condition*, meaning that they should only match when the lexer is in a certain state. For example, in Listing 5.4, each rule says to match the register names only when the lexer is in the state of parsing an instruction row or a condition parameter. Otherwise, the string "r3", for example, may be matched as a register name even if it is intended as a label. The start conditions prevent that. Note the "BEGIN" statement in each rule, those statements are what tells the lexer to go into a certain state. The second block in Listing 5.4 defines some instruction mnemonics. When these mnemonics are encountered, either in the initial state or inside an instruction row, the state of the lexer goes to "INSTR_ROW", allowing the register rules to apply from that point.

The lexer and the parser share some data structures and the entire set of tokens.

```
1  (define-abstract-instruction base_cpu_instr
2    ((functional-unit (fu))
3     (issue-slot (is))
4     (size 32)
5     (latency 1)
6     (throughput 1)))
7
8  (define-instruction ADD
9    ((extends base_cpu_instr)
10    (meaning "Add Word")
11    (asm "ADD \$\%rd, \$\%rs, \$\%rt")
12    (encoding (6 #b000000) rs rt rd (11 #b00000100000))
13    (operands (reg rs r32) (reg rt r32) (reg rd r32))))
14
15 (define-instruction ADDI
16   ((extends base_cpu_instr)
17    (meaning "Add Immediate Word")
18    (asm "ADDI \$\%rt, \$\%rs, \%immediate")
19    (encoding (6 #b001000) rs rt immediate)
20    (operands (reg rt r32) (reg rs r32) (imm immediate -32768 32767))))
21
22 (define-instruction BEQ
23   ((extends base_cpu_instr)
24    (meaning "Branch on Equal")
25    (asm "BEQ \$\%rs, \$\%rt, \%offset")
26    (encoding (6 #b000100) rs rt offset)
27    (operands (reg rs r32) (reg rt r32) (rel-address offset -32768 32767))))
28
29 (define-instruction NOP
30   ((extends base_cpu_instr)
31    (meaning "No Operation")
32    (asm "NOP")
33    (encoding (32 #b00000000000000000000000000000000))))
```

Listing 5.3. A portion of the MIPS32 instruction set definition. Only properties relevant for an assembler are shown. Note that several properties are defined in the abstract instruction, making the definitions of the concrete instructions rather small.

The "yylval.string" variable is set by the lexer to the string value that was read when producing the token. Line 17 of Listing 5.4 shows how the lexer sets the value of "yylval.string". That string value is then available to the parser.

5.2.4 The parser

The parser is generated using the Bison parser generator as described in section 2.6.2. The Bison input file has a couple of interesting parts that are generated by Instant Assembler. One is the list of tokens that are to be used by both the parser and the lexer, and the other is the productions that define the grammar of instructions and registers. Listing 5.5 shows a subset of the token list.

Using the complete list of register and mnemonic tokens along with hard-coded common tokens such as brackets, commas and other symbols, the grammar of the assembly language can be defined. As mentioned in sections 3.3.1 and 3.3.2, there is a possibility of ambiguity if two instructions share the same syntax, or if one

```
1   ...
2
3  <INSTR_ROW>       "r3" { BEGIN(ADDR_EXPR); return REG_R3; }
4  <CONDITION_PARAM>"r3" { BEGIN(CONDITION); return REG_R3; }
5  <INSTR_ROW>       "r2" { BEGIN(ADDR_EXPR); return REG_R2; }
6  <CONDITION_PARAM>"r2" { BEGIN(CONDITION); return REG_R2; }
7  <INSTR_ROW>       "r1" { BEGIN(ADDR_EXPR); return REG_R1; }
8  <CONDITION_PARAM>"r1" { BEGIN(CONDITION); return REG_R1; }
9  <INSTR_ROW>       "r0" { BEGIN(ADDR_EXPR); return REG_R0; }
10 <CONDITION_PARAM>"r0" { BEGIN(CONDITION); return REG_R0; }
11 <INSTR_ROW>       "at" { BEGIN(ADDR_EXPR); return REG_AT; }
12 <CONDITION_PARAM>"at" { BEGIN(CONDITION); return REG_AT; }
13
14 ...
15
16 <INITIAL,INSTR_ROW>"ADD"
17 { BEGIN(INSTR_ROW); yylval.string = strdup(yytext); return MNEMONIC_ADD; }
18
19 <INITIAL,INSTR_ROW>"ADDI"
20 { BEGIN(INSTR_ROW); yylval.string = strdup(yytext); return MNEMONIC_ADDI; }
21
22 <INITIAL,INSTR_ROW>"BEQ"
23 { BEGIN(INSTR_ROW); yylval.string = strdup(yytext); return MNEMONIC_BEQ; }
24
25 <INITIAL,INSTR_ROW>"NOP"
26 { BEGIN(INSTR_ROW); yylval.string = strdup(yytext); return MNEMONIC_NOP; }
27
28  ...
```

Listing 5.4. A portion of the MIPS32 lexer input. The lexer matches tokens to the input text, alters the state of the lexer, and makes the read string values available to the parser through the "yylval.string" variable.

```
1  ...
2
3  %token REG_R3
4  %token REG_R2
5  %token REG_R1
6  %token REG_R0
7  %token REG_AT
8
9  ...
10
11 %token <string> MNEMONIC_ADD
12 %token <string> MNEMONIC_ADDI
13 %token <string> MNEMONIC_BEQ
14 %token <string> MNEMONIC_NOP
15
16 ...
```

Listing 5.5. A portion of the MIPS32 token list. All instruction mnemonics and register names are defined as tokens that are expected by the lexer. Note the "<string>" statement in the mnemonic token definitions. This is what tells both the lexer and the parser that a string value should be made available during the lexer matching. (See Listing 5.4)

```
1 register_set_0 :
2           REG_R3 { $$ = ID_REG_R3; }
3         | REG_R2 { $$ = ID_REG_R2; }
4         | REG_R1 { $$ = ID_REG_R1; }
5         | REG_R0 { $$ = ID_REG_R0; }
6         | REG_AT { $$ = ID_REG_AT; }
7 ...
```

Listing 5.6. Part of the register parsing rule for the MIPS32 registers. All registers that share at least one register are grouped into one rule.

register is a member of more than one register class. To avoid such problems, all register classes that share at least one register and all instructions that share the same syntax are grouped together in the parser. Listing 5.6 shows what the production for some register looks like and Listing 5.7 shows how that is used in the instruction productions. Note that the ambiguity has not really disappeared. It just makes the parser give more ambiguous results. The task of identifying exactly which instruction to select is a job for the matcher.

5.2.5 The instruction matcher

Once parsing is done, the assembler will perform label- and other expression resolution. In this step, the symbols and mathematical expressions are evaluated to actual values. The parsing rule of the 'BEQ' instruction in Listing 5.7 will match no matter to what value the constant expression will be evaluated. In the specification of the instruction in Listing 5.3, there are however limits as to what values are valid. Also, as mentioned in section 3.4, the values of labels may change depending on the size of instructions. Resolving values and matching instructions must, therefore, be done together. By iteratively reevaluating expressions and rematching instructions, the program will eventually (likely after one or two iterations) find viable instructions and label values. Listing 5.8 shows the matching instruction that is used for matching of the 'ADD' instruction.

The *compatibility function* of each instruction with the correct mnemonic is called in the selector function. The compatibility function will check that the operands are valid by checking that registers are in the expected register classes, and that constant expressions are within the expected boundaries. Listing 5.9 shows the *compatibility function* for the 'ADDI' function.

5.2.6 The encoder

In the case of MIPS32, the encoder needs to know how to encode registers, immediate operands, and the instructions themselves. The encoding of immediate operands is assumed to be architecture independent and is not generated. The encoding of registers and instructions, on the other hand, are architecture dependent and are based on the specification.

```
1  instruction:
2    MNEMONIC_ADD register_set_0 COMMA register_set_0 COMMA register_set_0 {
3     $$ = make_instruction(@1.last_line, $1, 3, one_of_many_78596);
4     set_register_operand($$, 0, $2);
5     set_register_operand($$, 1, $4);
6     set_register_operand($$, 2, $6);
7    }
8
9    | MNEMONIC_ADDI register_set_0 COMMA register_set_0 COMMA HASH const_expr {
10    $$ = make_instruction(@1.last_line, $1, 3, one_of_many_78614);
11    set_register_operand($$, 0, $2);
12    set_register_operand($$, 1, $4);
13    set_const_expr_operand($$, 2, $7);
14   }
15
16   | MNEMONIC_BEQ register_set_0 COMMA register_set_0 COMMA HASH const_expr {
17    $$ = make_instruction(@1.last_line, $1, 3, one_of_many_78611);
18    set_register_operand($$, 0, $2);
19    set_register_operand($$, 1, $4);
20    set_const_expr_operand($$, 2, $7);
21   }
22
23   | MNEMONIC_NOP {
24    $$ = make_instruction(@1.last_line, $1, 0, one_of_many_78633);
25   }
26
27 ...
```

Listing 5.7. Some of the instruction parsing rules. Note the "one_of_many_NNNNN" identifier in the production actions. Since the production has been made so that more than one instruction can hit the same production, there is a generated selector function passed with each one. That function will decide at a later stage which functions do or do not match the given operands.

Encoding registers

The encoding of registers is specified in the register classes. That way, the same register may be encoded differently in different instructions so long as the instructions operate on different register classes. The encoding itself is just a mapping between registers and the values they take in encoded form. Listing 5.10 shows the encoding function of the 'r32' register class in MIPS32.

Encoding instructions

As in the case of register encoding, the instruction encoding is a direct translation of the encoding field of instructions in the specification. The encoding procedure is slightly more complex since the output is composed of several fields (static fields and operands). The operand values are fetched using the operand encoding functions and then shifted into the correct position. Listing 5.11 shows how this encoding is done for the 'ADDI' instruction. Note that the encoding functions are direct translations of the encoding fields of the instructions (Listing 5.3). The positioning

```
1  int one_of_many_78596(const binary_isdT *isd, const asm_sessionT *session,
2                        statementT *stmnt, instructionT *instr,
3                        bool use_far_pcu)
4  {
5    /* We select a candidate by picking the smallest matching instruction */
6    encoder_funT encoder = NULL;
7    lister_funT lister = NULL;
8    int smallest_size = INT_MAX;
9    int current_size = INT_MAX;
10   uint32_t issue_slots;
11   int bundle_priority;
12 /* l:32 q:15/32 noof-user-bits: 15 */
13   if (gfs_compatible_with_ADD(session, stmnt, instr, &current_size) >
14       INCOMPATIBLE) {
15     if (smallest_size > current_size) {
16       smallest_size = current_size;
17       encoder = gfs_encode_ADD;
18       lister = list_ADD;
19       issue_slots = ISSUE_SLOT_ALL;
20      bundle_priority = 1;
21     }
22   }
23   if (smallest_size == INT_MAX)
24     return 1; /* We failed to find an instruction */
25   instr->nof_bits = smallest_size;
26   instr->encoder_fun = encoder;
27   instr->lister_fun = lister;
28   instr->issue_slots = issue_slots;
29   instr->bundle_priority = bundle_priority;
30   stmnt->address_dependency |= 0;
31   return 0;
32 }
```

Listing 5.8. The selector function of all instructions with the 'ADD' mnemonic. Since the MIPS32 instruction set only has one such instruction, it will only check for compatibility on one instruction. Lines 12-20 compose a code block that checks compatibility of one instruction. If there are more than one candidate for selection, there would be additional such blocks, ordered by instruction size. The function checks compatibility of each candidate and selects the instruction with the smallest size. If no instruction is found, an error is indicated by a positive return value.

and widths of fields are generated from the instruction encoding and register widths.

```
1  compatibilityT
2  gfs_compatible_with_ADDI(const asm_sessionT *session,
3                           statementT *stmnt, instructionT *instr,
4                           int *nof_bits)
5  {
6    compatibilityT compatibility = COMPATIBLE;
7    int op DONT_WARN_UNUSED = 0;
8    if (instr->condition != NULL)
9      return INCOMPATIBLE;
10   gfs_op_is_in_register_class_r32(instr->operands[op++], &compatibility);
11   if (compatibility == INCOMPATIBLE)
12     return compatibility;
13   gfs_op_is_in_register_class_r32(instr->operands[op++], &compatibility);
14   if (compatibility == INCOMPATIBLE)
15     return compatibility;
16   op_is_compatible_with_immed(session, stmnt, instr,  instr->operands[op++],
17                               0, 65535, &compatibility);
18   if (compatibility > INCOMPATIBLE) {
19     *nof_bits = 32;
20   }
21   return compatibility;
22 }
```

Listing 5.9. The compatibility function for the 'ADDI' instruction. It checks that the registers are in the expected register class, and that immediate values are within the boundaries supported by the instruction. If the instruction is compatible, the size of the instruction is indicated so that the selector function may select the most appropriate instruction depending on size.

```
1  uint64_t gfs_encode_operand_for_r32(register_idT reg_id)
2  {
3    switch(reg_id)
4    {
5      case ID_REG_ZERO: return 0;
6      case ID_REG_R0: return 0;
7      case ID_REG_R1: return 1;
8      case ID_REG_AT: return 1;
9      case ID_REG_V0: return 2;
10     case ID_REG_R2: return 2;
11     case ID_REG_V1: return 3;
12     case ID_REG_R3: return 3;
13 ...
14     default:
15       assert(0 && "Register is not member of register class r32\n"); return 0;
16   }
17 }
```

Listing 5.10. The register encoding function for the 'r32' register class. Each register maps to a value in the register class. The values are directly fetched from the specification. Note that registers may map to the same value. That would in effect make them aliases.

```
1  int gfs_encode_ADDI(statementT *stmnt, instructionT *instr)
2  {
3    /* ADDI */
4    uint64_t op DONT_WARN_UNUSED;
5    op = instr->operands[2]->op.c_expr.const_expr->eval_result;
6    opcode_set_field(instr->opcode, op, ((((32 - 6) - 5) - 5) - 16), 16);
7    op = gfs_encode_operand_for_r32(instr->operands[0]->op.reg);
8    opcode_set_field(instr->opcode, op, (((32 - 6) - 5) - 5), 5);
9    op = gfs_encode_operand_for_r32(instr->operands[1]->op.reg);
10   opcode_set_field(instr->opcode, op, ((32 - 6) - 5), 5);
11   opcode_set_field(instr->opcode, 0x8, (32 - 6), 6);
12   return 0;
13 }
```

Listing 5.11. Encoding function for the 'ADDI' instruction. Each operand is encoded, and the resulting value is put in the correct place within the instruction. Statically defined fields in the specification are put directly into the encoding function as done on line 11 in this example.

5.3 Extension: VLIW Bundles

If one was to experiment with a VLIW extended MIPS32 architecture, there are several additions needed in the architecture specification. The architecture needs more than one functional unit in order to execute more than one instruction. It also needs more than one issue slot since the issue slots are where the instructions are placed in the bundle. The bundle syntax, or more precisely, the delimiter of instructions in bundles is defined in the parser. Each instruction that can be part of a bundle needs means of encoding them as such. This section will extend the MIPS32 architecture and inspect the newly generated output.

5.3.1 Functional units and issue slots

The concept of functional units are only important when instructions are executed, not when they are assembled. Hence, a simulator would need to take them into consideration while an assembler does not need that. The assembler only needs to focus on issue slots in order to produce the instruction bundles. For this example, the extended MIPS32 architecture will have two functional units for basic arithmetic such as addition and subtraction, two functional units for multiplication and division and one for branching and jumping. To make the example easy to follow, the set of issue slots will be analogous to the set of functional units. Listing 5.12 shows a specification of functional units and issue slots that matches this description. The instructions of each bundle will need to be ordered as the issue slots are ordered. That is, though not all issue slots need to be occupied in each bundle, any branch instruction must occur after any multiplication instruction. Any multiplication instruction must occur after any basic arithmetic instruction.

```
(declare-functional-units (ba1 ba2 mul1 mul2 bra))

(declare-issue-slots (iba1 iba2 imul1 imul2 ibra))
```

Listing 5.12. An example of an extended set of functional units and issue slots. The order in which the issue slots are listed is the order that bundles should be ordered.

```
(define-instruction ADD
  ((extends base_cpu_instr)
   (meaning "Add Word")
   (asm "ADD $\%rd, $\%rs, $\%rt")
   (issue-slot (iba1 iba2))
   (encoding P (13 #b0000000000000) rs rt rd (11 #b00000100000))
   (operands (reg rs r32) (reg rt r32) (reg rd r32))))

(define-instruction MUL
  ((extends base_cpu_instr)
   (meaning "Multiply Word to GPR")
   (asm "MUL $\%rd, $\%rs, $\%rt")
   (issue-slot (imul1 imul2))
   (encoding P (13 #b0000000011100) rs rt rd (11 #b00000000010))
   (operands (reg rs r32) (reg rt r32) (reg rd r32))
   (semantics (set-reg! rd (mul 32 rs rt)))))

(define-instruction BEQ
  ((extends base_cpu_instr)
   (meaning "Branch on Equal")
   (asm "BEQ $\%rs, $\%rt, \%offset")
   (issue-slot (ibra))
   (encoding P (13 #b0000000000100) rs rt offset)
   (operands (reg rs r32) (reg rt r32) (rel-address offset -32768 32767))))
```

Listing 5.13. A few instructions extended for a VLIW version of MIPS32. The specification has been extended to define the issue slots used by each instruction and the P bit has been introduced in the encoding field. The instructions have been made 8 bits wider to make room for the P bit.

5.3.2 Extending instruction properties

When the issue slots and functional units exist, the instructions need to be extended to use them. Each instruction that can be part of a bundle needs to list the issue slots in which is can be placed and it needs to specify the position of the P bit. The P bit is a single bit that is defined to have the value 1 when the instruction is in the same bundle as the instruction before it. In the encoded form, bundles are recognized as a list of instructions where every instruction but the first one has a P bit value of 1. Listing 5.13 shows a few instructions that have been extended with the 'issue-slots' field and with P bits. In order to make room for the P bit, they have also been extended to 40 bits per instruction instead of the 32 bits they were before.

```
1  %token SEMICOLON
2
3  ...
4
5  %left SEMICOLON
6
7  ...
8
9  par_instr_delim : SEMICOLON
10                 ;
11
12 instr_row       : instruction
13                   {   /* new instruction row: */
14                     ...
15                   }
16                 | instr_row par_instr_delim instruction
17                   {   /* add instruction to row: */
18                     ...
19                   }
20                 ;
21
22 ...
```

Listing 5.14. Excepts from the modified grammar file. The semicolon is listed as a token so that the lexer can recognize it. It is defined as left associative to avoid an ambiguous grammar, and the instruction row is defined as either a single instruction or an instruction row followed by the delimiter and then another instruction.

5.3.3 Setting the instruction delimiter

The Instant Assembler is built with the assumption that an instruction bundle is syntactically defined as a series of instructions with a defined character as instruction delimiter. In this example, the extended VLIW MIPS32 architecture defines its bundles using a single semicolon as delimiter. The ADL does not provide a mechanism to define this using a specification. The hand-written part of the grammar file needs a slight modification. Listing 5.14 shows a few excerpts of the modified grammar. One also needs to make sure that there is a lexer rule that will match against a single semicolon and return the appropriate token.

5.3.4 Effects on the assembler

When the extensions and modifications of this section are in place, the parser detects instruction bundles and the encoder is able to encode them properly. The modifications to the parser are not derived from the specification so the generated parser code will not differ. The main difference from before can be seen in the instruction encoding function. Listing 5.15 shows the new encoding function for the 'ADD' instruction.

```
1  int gfs_encode_ADD(statementT *stmnt, instructionT *instr)
2  {
3    /* ADD */
4    uint64_t op DONT_WARN_UNUSED;
5    opcode_set_field(instr->opcode, 0x20, ((((((40 - 1) - 13) - 5) - 5)
6                                            - 5) - 11), 11);
7    op = gfs_encode_operand_for_r32(instr->operands[0]->op.reg);
8    opcode_set_field(instr->opcode, op, (((((40 - 1) - 13) - 5) - 5) - 5), 5);
9    op = gfs_encode_operand_for_r32(instr->operands[2]->op.reg);
10   opcode_set_field(instr->opcode, op, ((((40 - 1) - 13) - 5) - 5), 5);
11   op = gfs_encode_operand_for_r32(instr->operands[1]->op.reg);
12   opcode_set_field(instr->opcode, op, (((40 - 1) - 13) - 5), 5);
13   opcode_set_field(instr->opcode, 0x0, ((40 - 1) - 13), 13);
14   opcode_set_field(instr->opcode, 0, (40 - 1), 1);
15   instr->p_bit_position = (40 - 1);
16   return 0;
17 }
```

Listing 5.15. The encoding function of the modified 'ADD' instruction. At this stage, the P bit is set to 0 because the bundle arrangement and setting of P bits are done at a later stage. Note that the encoder now defines 40 bits of data and that the position of the P bit is pointed out by the encoder function.

5.4 Extension: Conditional Execution

As mentioned in 2.1.3, some DSP processors have the ability to make just about any instruction conditional by attaching predicates to them. In this section, the extended MIPS32 architecture will be given yet another extension. A couple of conditions will be defined, and instructions will be made to support them.

In this example, the syntax of conditionally executed instructions will be the instruction itself followed by a comma, the word "if", the condition predicate, a colon, and then a register operand. Listing 5.16 shows an example of the 'ADD' instruction with a condition expressed with the described syntax.

5.4.1 Defining conditions

Conditions are defined similarly to instructions. Section A.3 describes the ADL features available for doing so. In this example, consider the two newly defined conditions in Listing 5.17. The size of the condition should be interpreted as the number of bits by which a conditional instruction will grow.

```
1  ADD r3, r1, r2, if gtz: r1
```

Listing 5.16. The MIPS32 'ADD' instruction with a condition appended to it. The intended effect of the condition is for the instruction to only be executed if the register 'r1' is greater than zero (hence "gtz").

```
1  (define-condition condition_gtz
2    ((meaning "Register is greater than zero.")
3     (operands (reg regi r32))
4     (asm "if gtz: %regi")
5     (size 8)
6     (encoding P (3 #b101) regi)
7     (semantics (icmp 32 'sgt regi 0))))
8
9  (define-condition condition_etz
10   ((meaning "Register is equal to zero.")
11    (operands (reg regi r32))
12    (asm "if etz: %regi")
13    (size 8)
14    (encoding P (3 #b110) regi)
15    (semantics (icmp 32 'eq regi 0))))
```

Listing 5.17. Two instruction conditions. Conditions are defined similarly to instructions and share several properties.

5.4.2 Defining conditional instructions

Now that conditions exist in the extended architecture, what remains is to make the instructions support them. That is not done entirely automatically since architectures may exist that only allow some instructions to be conditional. Explicitly listing all possible conditions also gives more flexibility. For instance, an architecture may have a set of conditions that are only valid for floating point operations.

Listing 5.18 illustrates the 'ADD' instruction from Listing 5.13, but with the 'conditions' attribute added to it. That attribute tells the code generator that the instruction may come with a condition appended to it. Note that the size of the instruction has not changed. In fact, the instruction has not changed at all because this will now generate two instructions. One conditional one, and one which is exactly as before.

A noteworthy detail is that it may be impractical to list all conditions in every instruction. That is easily remedied using abstract instructions. For example, in this case one could make all integer operations conditional by extending the 'base_cpu_instr'. The instructions in the example are extended directly for the sake of clarity.

5.4.3 Resulting output

The extensions made in this section will affect the parser, the instruction selector, and the encoder. We will have a look at all of them one by one.

Parser

With the new input, the parser has been extended with new rules for both conditions and instructions. Listing 5.19 shows the new rules for the 'ADD' instruction. There are now two of them: The conditional version and the unconditional one. The conditions themselves have also been added to the parser, as shown in Listing 5.20.

```
(define-instruction ADD
  ((extends base_cpu_instr)
   (conditions condition_gtz
               condition_etz)
   (meaning "Add Word")
   (asm "ADD $%rd, $%rs, $%rt")
   (issue-slot (iba1 iba2))
   (size 40)
   (encoding P (13 #b0000000000000) rs rt rd (11 #b00000100000))
   (operands (reg rs r32) (reg rt r32) (reg rd r32))
   (semantics (nop))))
```

Listing 5.18. The 'ADD' instruction with a 'conditions' field added to it. That field instructs Instant Assembler that the two specified conditions are valid for the instruction.

```
| MNEMONIC_ADD register_set_0 COMMA register_set_0 COMMA register_set_0 {
    $$ = make_instruction(@1.last_line, $1, 3, one_of_many_78898);
    set_register_operand($$, 0, $2);
    set_register_operand($$, 1, $4);
    set_register_operand($$, 2, $6);
}

| MNEMONIC_ADD register_set_0 COMMA register_set_0 COMMA register_set_0
  COMMA condition {
    $$ = make_instruction(@1.last_line, $1, 3, one_of_many_78899);
    set_register_operand($$, 0, $2);
    set_register_operand($$, 1, $4);
    set_register_operand($$, 2, $6);
    set_condition($$, $8);
}
```

Listing 5.19. The two production rules for the 'ADD' instruction after addition of conditions. The first rule is the same as before. The second one with a condition after it. The action for the second rule differs only in that it adds a condition to the instruction data structure, and they do not use the same instruction selector function since the syntax is not equal.

```
condition :
  IF COND_PRED_ETZ COLON register_set_0  {
    /* This is one of (condition_zero) */
    $$ = make_register_condition(@1.last_line, CPRED_ETZ, $4, 0, 0);
  }
  | IF COND_PRED_GTZ COLON register_set_0  {
    /* This is one of (condition_gt) */
    $$ = make_register_condition(@1.last_line, CPRED_GTZ, $4, 0, 0);
  }
  ;
```

Listing 5.20. The entire condition parsing code. One production rule for each condition.

```
1  compatibilityT
2  gfs_compatible_with_conditional_ADD(const asm_sessionT *session,
3                                      statementT *stmnt, instructionT *instr,
4                                      int *nof_bits)
5  {
6    compatibilityT compatibility = COMPATIBLE;
7    int op DONT_WARN_UNUSED = 0;
8    gfs_op_is_in_register_class_r32(instr->operands[op++], &compatibility);
9    if (compatibility == INCOMPATIBLE)
10     return compatibility;
11   gfs_op_is_in_register_class_r32(instr->operands[op++], &compatibility);
12   if (compatibility == INCOMPATIBLE)
13     return compatibility;
14   gfs_op_is_in_register_class_r32(instr->operands[op++], &compatibility);
15   if (compatibility == INCOMPATIBLE)
16     return compatibility;
17   if (instr->condition == NULL)
18     return INCOMPATIBLE;
19   gfs_condition_is_compatible_with_one_of_condition_etz_condition_gtz(instr,
20                                                                      &compatibility);
21   if (compatibility > INCOMPATIBLE) {
22     *nof_bits = 48;
23   }
24   return compatibility;
25 }
```

Listing 5.21. The compatibility function for the conditional 'ADD' instruction. Note that the function checks that the condition attached to it is either 'etz' or 'gtz'. Another noteworthy detail is the size of the instruction, 48, which is the size of the regular instruction plus the size of the condition.

Instruction selector

As shown in Listing 5.19, the newly added rule uses a different instruction selection function than the non-conditional counterpart. The new selector functions do, however, not differ from the other ones except for the compatibility functions they use. Since there are in effect new instructions, there is a new set of compatibility functions. The compatibility function for the conditional 'ADD' instruction is shown in Listing 5.21.

Encoder

As in the case of parsing and selection of instructions, the encoding process treats the conditional instructions as whole new instructions. For each instruction that can be conditional, two encoding functions are made. The unconditional version still looks exactly as before. The conditional one is similar but with a different width and with a call to the condition encoder. Listing 5.22 shows what the conditional 'ADD' instruction looks like. Note the condition encoder call at the end of the function. The condition encoder for the 'etz' condition is shown in Listing 5.23.

```
1  int gfs_encode_conditional_ADD(statementT *stmnt, instructionT *instr)
2  {
3    /* conditional_ADD */
4    uint64_t op DONT_WARN_UNUSED;
5    opcode_set_field(instr->opcode, 0x20, ((((((48 - 1) - 13) - 5) - 5)
6                                            - 5) - 11), 11);
7    op = gfs_encode_operand_for_r32(instr->operands[0]->op.reg);
8    opcode_set_field(instr->opcode, op, (((((48 - 1) - 13) - 5) - 5) - 5), 5);
9    op = gfs_encode_operand_for_r32(instr->operands[2]->op.reg);
10   opcode_set_field(instr->opcode, op, ((((48 - 1) - 13) - 5) - 5), 5);
11   op = gfs_encode_operand_for_r32(instr->operands[1]->op.reg);
12   opcode_set_field(instr->opcode, op, (((48 - 1) - 13) - 5), 5);
13   opcode_set_field(instr->opcode, 0x0, ((48 - 1) - 13), 13);
14   opcode_set_field(instr->opcode, 0, (48 - 1), 1);
15   instr->p_bit_position = (48 - 1);
16   instr->condition->encoder(instr->opcode, instr->condition, (0 + 8));
17   return 0;
18 }
```

Listing 5.22. The encoding function of the conditional 'ADD' instruction. Its only difference from an unconditional instruction is the call to the condition encoder in the end.

```
1  void condition_encode_condition_etz(bitvecT opcode[2],
2                                      conditionalT *condition, int bit_pos)
3  {
4    uint64_t op DONT_WARN_UNUSED;
5    op = gfs_encode_operand_for_r32(condition->param.reg);
6    opcode_set_field(opcode, op, (((bit_pos - 1) - 3) - 5), 5);
7    opcode_set_field(opcode, 0x6, ((bit_pos - 1) - 3), 3);
8    opcode_set_field(opcode, 1, (bit_pos - 1), 1);
9  }
```

Listing 5.23. The encoding function the 'etz' condition. The encoding process is very similar to that of instruction encoding. The operand fields and the static fields are encoded the same. Note that the bit vector used is that of the instruction. The condition encoder is instructed as to where to write its encoded bits using the 'bit_pos' parameter.

5.5 Manual implementational changes

The ADL itself did not pose any difficulties to expressing the extended MIPS32 architecture. All features of the hypothetical architecture could without modifications be expressed using the Instant Toolsmith ADL as shown in this chapter.

There are, however, some details in the parsing of the specification and in the C code generation that did need some modification for the new assembler syntax to be accepted. For instance, the Instant Assembler did not expect a dollar sign before register operands. That was, however, easily fixed. Condition syntax was a concern that needed some more attention. Both the specification parser and the C code generator did make assumptions about the condition syntax, which is different for the processor architecture toward which the ADL was originally developed.

5.6 Development Effort

This section presents the amount of specification code that was needed to generate the assembler as well as making the extensions presented in this chapter. Although amount of code does not always map directly to effort, rough numbers should give some idea on the matter.

The instruction specifications of the MIPS32 architecture is written in about 2500 lines of specification. This includes blank lines and also a textual description of each instruction meant for generating documentation. The description also contains specification of the semantics of each instruction. Semantics and textual descriptions are not relevant for the making of an assembler.

With textual descriptions removed, the instruction specification is about 1400 lines long. With the 155 instructions that are defined in the specification, there are about 9 lines per instruction. There are on average 1 blank line per instruction and most have a semantics description of 1-3 lines. Defining MIPS32 instructions using Instant Toolsmith ADL thus requires about 5-7 lines per instruction.

Defining registers requires less code than that. Each register is defined using one line of Instant Toolsmith ADL, plus one additional line for each register class that the register is a member of. In the case of MIPS32 each register is member of one register class.

5.6.1 VLIW extension

As shown in Section 5.3, the example extension required two lines to declare the issue slots and functional units. Each conditional instruction then required one additional line. This additional line can, however, often be removed by defining them in abstract registers.

5.6.2 Conditional Execution

As shown in Section 5.4, defining conditions to be used in conditional instructions required on average 7 lines per condition. The number of additional lines per instruction depends on the number of conditions that are available for each instruction. By defining conditions in abstract instructions, the number can in practice become very small (much less than one line per instruction).

Chapter 6

Conclusion

This report presents the Instant Assembler as part of the Instant Toolsmith project, which is aimed at creating toolsets for processor architectures backed by a domain specific language.

The focus has been on making an assembler generator that is easy to change and maintain. This chapter analyzes the results obtained in Chapter 5 and provides suggestions for future work.

6.1 Result Analysis

The two key evaluation points are fast development cycles and ADL coverage. ADL coverage refers to both what the ADL covers, and to what extent the Instant Assembler makes use of it.

6.1.1 Fast development cycles

With a lack of something to compare the results with, the term "fast" is considered a subjective measure. That being said, the case study does show how entirely new instructions may be specified using very little code. It shows how the encoding of said instructions can be altered by changing one or two lines per instruction (Listing 5.13). Moreover, new features to the architecture such as VLIW and conditional execution are added without much effort, including a definition of issue slots that can automatically cause the assembler to order VLIW bundles in certain ways.

6.1.2 ADL coverage

Instant Assembler makes use of the ADL for defining the vast majority of architecture specific aspects of the generated assembler. It builds instruction and condition syntax based on the ADL, identifies operands and generates the entire instruction selection mechanism as well as the encoder.

Some details on syntax and encoding do remain in the implementation of the Instant Assembler. It was found in the case study that some assumptions about condition syntax made it necessary to adjust minor parts of the Instant Assembler implementation. The assembly parser also needed a minor adjustment to support the bundle syntax.

Overall, the amount of architecture specific information that could be expressed using the Instant Toolsmith ADL and used by the Instant Assembler was very high. The method described in this report is a promising way to reduce the amount of manual work required to build processor tool chains.

6.2 Future Work

The case study shows a compelling way of working with processor architectures in a way that is easy to maintain and adapt to changes. A few interesting development points have been identified which could make Instant Assembler an even easier and more effective tool for future use.

6.2.1 Wider use of the Instant Toolsmith ADL

It would be very useful for further development of the Instant Assembler as well as the Instant Toolsmith ADL to start using the tools for more processor architectures. As architectures are described using the ADL and tools are generated using Instant Assembler, they would all surely evolve into greater maturity and possibly become more general in the process. That could become beneficial for tool developers, ADL maintainers, as well as hardware architects.

6.2.2 Extensions of Instant Toolsmith ADL

During the case study presented in this thesis, a few situations arose where it would be hard or sometimes not possible to describe all changes and additions using only the Instant Toolsmith ADL. While the definitive majority of architecture dependent information was described easily, there are a few opportunities for extension.

Bundle syntax

At the time of writing, the syntax of instruction bundles is defined by hand in the Bison grammar file. This is not a major issue as long as the assembly language matches the bundle syntax in Instant Assembler. If the syntax of defining bundles is significantly different, as in the case of Hexagon [2], the parser needs to be modified by hand. Extension of the Instant Assembler and the Instant Toolsmith ADL to support defining this syntax in the specification could be very useful.

Bundle encoding

The encoding of bundles is assumed to be done using a specified bit in the

instruction encoding that will have the value of 1 in case the instruction belongs to the same bundle as the one before it. This is not true for all processor architectures. Adapting Instant Assembler to a different bundle encoding scheme would require changing the generation of the instruction encoding functions. The Instant Assembler would become more general if bundle encoding could be specified.

Syntax of conditional instructions

Instructions that have conditions bound to them are assumed to be written in the assembly code as the instruction followed by a comma and then the condition. While the instruction syntax itself and the condition syntax can be specified, the combination of the two can not be changed in the specification. In order to reach more varying processor architectures, it may be necessary to be able to handle more varying assembly syntaxes.

Bibliography

[1] Dennis M. Ritchie Brian W. Kernighan. *The C Programming Language*. English. 2nd edition. Prentice, 1988. ISBN: 8120305965.

[2] Codrescu, L., Comput. Eng., Georgia Inst. of Technol., Atlanta, GA, USA, and Anderson, W. ; Venkumanhanti, S. ; Mao Zeng ; Plondke, E. ; Koob, C. ; Ingle, A. ; Tabony, C. ; Maule, R. "Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications". English. In: *Micro, IEEE* 34.2 (May 2014), pp. 34–43. ISSN: 0272-1732. DOI: 10.1109/MM.2014.12.

[3] Franklin Lewis Deremer. *Practical translators for LR(k) languages.* eng. 1969.

[4] Free Software Foundation. *Bison - GNU Project - Free Software Foundation*. URL: http://www.gnu.org/software/bison/ (visited on 02/01/2015).

[5] Free Software Foundation. *GNU Binutils web page*. English. URL: http://www.gnu.org/software/binutils/ (visited on 01/25/2015).

[6] Imagination Technologies Ltd. *MIPS32 Architecture*. English. URL: http://www.imgtec.com/mips/architectures/mips32.asp (visited on 01/27/2015).

[7] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. English. Jan. 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf (visited on 02/01/2015).

[8] *Intel® Itanium® Architecture Software Developer's Manual*. URL: http://www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html (visited on 02/09/2015).

[9] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. English. 5th ed. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[10] Karl Johansson. *Instant Simulator*. Tech. rep. TRITA-ICT-EX-2014:106. Stockholm: KTH Royal Institute of Technology, 2014.

[11] John R. Levine. *Linkers and Loaders*. English. 1 edition. San Francisco: Morgan Kaufmann, Oct. 1999. ISBN: 9781558604964.

[12] LLVM Project. *clang: a C language family frontend for LLVM*. URL: http://clang.llvm.org (visited on 01/25/2015).

[13] LLVM Project. *TableGen documentation*. URL: `http://llvm.org/docs/TableGen/` (visited on 01/25/2015).

[14] LLVM Project. *The LLVM Compiler Infrastructure*. URL: `http://llvm.org/` (visited on 01/25/2015).

[15] PLT Inc. *The Racket Language*. URL: `http://racket-lang.org/` (visited on 06/26/2015).

[16] Anthony J. Dos Reis. *Compiler Construction Using Java, JavaCC, and Yacc*. English. 1 edition. Hoboken, N.J: Wiley-IEEE Computer Society Pr, Dec. 2011. ISBN: 9780470949597.

[17] The Flex Project. *flex: The Fast Lexical Analyzer*. URL: `http://flex.sourceforge.net/` (visited on 02/01/2015).

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, in equal parts, by Mattias Jansson and Karl Johansson.

A.1 Describing registers

The register specification is a model of the available registers for the architecture. It describes their properties of as well as relationships between registers such as overlap and grouping. This section describes the ways 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 code. In the specification, each register will have an assembler syntax as well as a unique identifier for use by the 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. More on that in A.1.6).

Listing A.1 will set the identifier and assembler syntax for **AX**.

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

Listing A.1. The most basic register definition

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

Listing A.2. Register with a defined size

```
1 (define-reg ax ((asm AX) (size 32) (access ``rw'')))
2 (define-reg pc ((asm PC) (size 32) (access ``r'')))
```

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

```
1 (define-reg ax ((asm AX) (size 32) (access ``rw'')))
2
3 (define-reg al ((asm AL) (size 16) (access ``rw'') (overlap (ax 0 15))))
4
5 (define-reg ah ((asm AH) (size 16) (access ``rw'') (overlap (ax 16 31))))
```

Listing A.4. Sub-registers with specified overlap

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 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.

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.

```
(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

```
(define-reg-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

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 registers. Maybe 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 are specified by referring to a register class (section A.2.4).

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

```
1  (define-abstract-reg large-reg ((size 32) (access ``rw'')))
2  (define-abstract-reg small-reg ((size 16) (access ``rw'')))
3
4  (define-reg ax ((asm AX) (extends large-reg)))
5
6  (define-abstract-reg ax-child ((extends small-reg)
7                                 (ref (ax 'my-parent))))
8
9  (define-reg al ((asm AL) (extends ax-child)
10                 (overlap (ax 0 15)) (ref (ah 'my-sibling))))
11
12 (define-reg ah ((asm AH) (extends ax-child)
13                 (overlap (ax 16 31)) (ref (al 'my-sibling))))
14
15 (define-reg pc ((asm PC) (extends large-reg) (access ``r'')))
```

Listing A.8. Registers defined using abstract registers

to or used except for when extending them in regular registers. In listing A.8 is a set of register definitions which are equivalent to the example in section A.1.5 (with the addition of **PC**).

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 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 code listing A.9.

```
1  (define-instruction add-max
2    ....)
```

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

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

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

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

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

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 code 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.

A.2.2 Functional units

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

Only functional units that are declared, 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 bundles (or bundles), there may be constraints on the order in which the instructions are to appear in the bundle. There may also be constraints on the number of different types of instructions that can be issued in the same bundle.

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 bundle reordering and encoding.

For example, if the two issue slots **is1** and **is2** are declared in that order. Then, if there is an instruction bundle with two instructions which use **is2** and **is1** respectively, they will be reordered in the bundle so that the instruction using **is1** occurs first in the bundle.

```
1 (define-instruction add-max
2   ((size 32)
3    (functional-unit (arith1) (arith2))
4    (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**.

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

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 code 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 provided 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 largerst value within the range instead. This allows the maximum value allowed for non-zero immediates to be one step higher than normal immediates of same bitlength.

(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 program-jumps based on offsets. They require a numeric minimum and maximum value, even though they might be 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.

```
1 (define-instruction add-max
2   ((size 32)
3    (functional-unit (arith1) (arith2))
4    (issue-slot (is1) (is2))
5    (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. There are several other types of operands available however.

```
1 (define-instruction add-max
2   ((size 32)
3    (functional-unit (arith1) (arith2))
4    (issue-slot (is1) (is2))
5    (operands (reg dst r32) (reg src1 r64) (reg src2 r64))
6    (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.

A.2.5 Assembly syntax

The *asm* property describes how instructions are presented in assembly code. The currently supported format only allows for representations beginning with a mnemonic, followed by the operands. Code listing A.15 shows how the example instruction is extended to include an assembly syntax.

Operands are prefixed with a % escapte 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.

A.2.6 Encoding

Instructions need an encoded representation so that they can be both assembled and disassembled. Code listing A.15 shows how the example instruction is extended with an encoding property.

The encoding property is a list containing an optional bundle-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 be 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. These lengths prevent leading zeroes from being stripped off. For VLIW architectures, a P symbol reflects a bundle flag of 1 bit.

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

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

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 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 controlled. There are two version of this form instead, *rif* and *cif* for if-statements resolved at runtime and if-statements resolved at compile-time respectively.

A.2.8 Abstract instructions

Instructions that share similar properties can use an abstract instruction as a template, to avoid code-duplication. These abstract intructions are defined with *define-abstract-instruction*, using properties identically to those found in normal instructions. Code listing A.17 shows an example where the functional unit and

```
(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 #b01111111111100001001100) src1 src2 dst )
   (semantic
     (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.

```
(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 that extending instruction need not do so.

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.

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 with instructions, abstract conditions cannot be defined. Code listing A.15 shows a typical condition that checks whether a general-purpose register is non-zero.

Unlike instructions, there are less properties to provide when defining conditions. *Size*, *encoding*, *operands*, and *semantics* are all valid attributes. Semantics need to

```
(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.

return a numeric value, although side effects (such as register updates) can occur.

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 is shown in code listing A.19. The special form *define-primitive* takes three parameters. The first parameter is a boolean value that signifies whether the primitiv returns a value. The second parameter is the primitive functions signature. The last argument is the name of the function in C code, so that a generated simulator can utilize it.

A.4.2 Utility functions

Utility functions provide means for avoiding 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 code 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.

```
(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.

A.4.3 Macro functions

Macro functions help circumvent the strict rules associated with defining instruction semantics. With macro functions, any valid Racket 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 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 code-duplication. A macro function that takes the opertion, 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.

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

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.

```
1 (declare-functional-units
2   (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.

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

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

A.4.5 Functional units

The existance of functional units is declared in a simple statement, as shown in code 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 bundle ordering as briefly described in section A.2.3. Instructions will be issued in bundle slots in the order that they appear in the issue slot declaration. Listing A.23 shows how issue slots are declared.