A Generic Assembler Generator

Marko Caklovic

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements
of the Degree of Master of Science at the University of Glasgow

7th September 2018

Abstract

Assemblers are typically highly specialized pieces of software, capable of working with only one underlying computer architecture, and not easily modifiable. On the other hand, computer architecture researchers need a tool which allows them to quickly experiment by adding features to an architecture and generating machine code utilizing these new features. Due to the opaque design and limited breadth of existing assemblers, this is a tedious and difficult task. This paper describes the design, implementation, and evaluation of a *generic assembler*. The generic assembler allows users to easily specify the assembly language of a computer architecture, as well as its underlying machine code. It then uses this specification to parse user supplied assembly code and generate machine code. Users can then quickly and easily modify the specification and assembly code, without modifying the assembler software itself. The generic assembler enables researchers to rapidly experiment with new ideas across a wide variety of different computer architectures.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Marko Caklovic Signature: MC

Acknowledgements

Many thanks to my supervisor, Dr. John O'Donnell, for the guidance and advice he gave me as I worked on this project.

I would also like to thank my friends and family for their tireless love and support.

Contents

Chapter 1 Introduction 1

Chapter 2 Assembly Languages & Machine Code 3

Chapter 3 Review of Existing Work 5

3.1 Review of Existing Assemblers 5

3.2 Review of Existing Literature 5

Chapter 4 Requirements 7

Chapter 5 System Design and Architecture 8

5.1 System Design 8

5.2 Custom ADL Design 9

5.3 Plugin System Design 11

5.4 Architecture 12

Chapter 6 Implementation 14

6.1 Custom ADL Parsing 14

6.2 Assembly Source Code Parsing 15

6.2.1 Handling Errors While Parsing Assembly Source Code 17

6.3 The Plugin System 18

6.3.1 The int Data Type 19

6.3.2 The label Data Type 20

6.4 Bitstream Generation 21

6.4.1 First Pass: Machine Code Generation 22

6.4.2 Second Pass: Calculating Label Addresses 24

Chapter 7 Evaluation and Testing 25

7.1 Evaluating x86 Support 25

7.2 Evaluating ARM Support 26

7.3 Evaluating Sigma16 Support 27

7.4 Test Automation 27

Chapter 8 Conclusion and Future Work 28

Chapter 1 Introduction

Research in computer architecture involves rapidly experimenting with new ideas on a variety of different architectures. One way to do this is to add or remove new operations in the architecture's specification, and then to evaluate these operations by using them in different scenarios to assess their impact on performance and usability.

Each architecture has its own *assembly language*. Assembly language is human readable source code which can be directly mapped to the underlying *machine code* of the architecture (Wick, 1975). The machine code of an architecture is a stream of binary code which the computer will interpret and execute. The assembly language is simply a convenience which enables programmers to more easily read and write said machine code.

A program which translates human readable assembly code to machine readable binary code is called an *assembler*. It takes as input an *assembly source file*, which is the source code for a program in written in assembly language, and it outputs machine code.

An assembler is usually a highly specialized piece of software, targeted at only one underlying architecture, and capable of processing only one type of assembly language. When researchers need to add or remove operations to an architecture, they must also typically modify the assembler of that architecture, to be able to generate machine code which reflects the new changes in the underlying architecture.

Since assemblers are specialized pieces of software, and usually difficult to quickly modify, this can be a tedious process. Furthermore, any changes the researcher makes will only be tied to that specific architecture. If the researcher wants to make similar changes to a different architecture, they must also make modifications to the assembler programs of those different architectures. This can be slow and tedious work which makes it hard to rapidly iterate on new ideas.

To try and solve this problem, we have designed and implemented a *generic assembler*. The purpose of the generic assembler is to take a specification for the machine code of an architecture, to then use it to parse a user-supplied assembly source file, and to finally generate machine code which is a direct translation of the user's assembly source code to the machine code of the specified architecture. The idea is to allow researchers to quickly make changes to the machine code specification of an architecture, and to be able to easily utilize those changes in their assembly source code.

The generic assembler takes three inputs: the syntax for the assembly language it will parse, the specification for the machine code that it will generate, and the assembly source code for which it will generate machine code. It will output binary machine code which implements the program specified in the user's source code.

The main purpose of the generic assembler is for researchers to be able to quickly make experimental changes to an architecture by simply modifying the inputs to the generator. This is much faster and easier than modifying the source code of an assembler program and should allow for faster experimentation. Furthermore, the generic assembler is able to parse assembly and generate machine code for very different architectures. Thus, researchers will be able to experiment with diverse architectures simply by changing the inputs to the generic assembler.

To allow researchers to conveniently define and modify the specification of an architecture, a custom *domain specific language* (DSL) was invented (Fowler, 2010). This DSL will allow researchers to quickly and easily define both the syntax of the assembly language, and the machine code which should be outputted by the generic assembler.

While the DSL is both powerful and expressive, it cannot express every aspect of an architecture. To aid users with filling in the gaps where the DSL might fall short, a Python-based plugin system was designed for the generic assembler. This plugin system can be used by users to extend the generic assembler in novel new ways and allows them to work around any limitations that might be inherent in the design of the generic assembler and its custom DSL.

Finally, the generic assembler is used to define and implement short assembly language programs for three very different architectures. This demonstration of the generic assembler's capabilities shows that it meets the project goal: providing a program which is capable of easily specifying different architectures and assembling machine code for them.

Chapter 2 Assembly Languages & Machine Code

Most computers have a *central processing unit* (CPU) that is responsible for executing a stream of instructions that make the computer run and perform its function. These machine instructions are a stream of binary code, expressed as a string of 1's and 0'.

Thus, all computer code can ultimately be expressed in binary. However, binary machine code is not easily understood by humans, and writing code in binary would be unrealistically painful and inefficient. To ease the programming of computers, researchers and programmers have over the years developed programming languages.

These programming languages allow for programmers to write human-readable code, and then use a special program to translate the human-readable code into binary machine code which the CPU can understand and execute. This special program that translates source code to machine code is called a *compiler*.

Programming languages come in many different flavors, all with a different set of abstractions. These abstractions allow for programmers to utilize the resources of the computer more efficiently, without having to pay attention to the actual details of how those resources are organized and utilized by more low-level software. The more abstractions a programming language has, the more work its compiler must do to translate those abstractions into the machine code of the underlying architecture.

An *assembly language* can be thought of as a programming language with little to no abstractions. Because *assembly code* has almost no abstractions, it can be mapped directly to the binary machine code of the underlying CPU. This also means that programming in assembly code is very verbose and tedious, and typically programmers avoid doing it in favor of using some higher-level language with better abstractions. Nevertheless, because assembly code is so low-level, it is indispensable for certain types programming which would be impossible in a higher-level language, simply because the abstractions of that higher-level language get in the way of what the programmer is trying to accomplish. Furthermore, assembly code is usually used in resource constrained computing environments, where the abstractions of higher-level programming languages would lead to prohibitively high overhead in code size and performance.

A program which translates assembly code to machine code is called an *assembler*. It is similar to a compiler in that it is responsible for translating human readable source code to machine readable binary code. However, unlike a compiler, an assembler deals exclusively with low-level assembly languages, and is responsible for generating machine code for some underlying architecture. Indeed, many compilers will translate some higher-level language source code to assembly code, and then pass the generated assembly code to an assembler for final translation into machine code (Clang.llvm.org, 2018).

There are many different kinds of CPUs, and CPUs come in many different *architectures*. A CPU architecture describes the underlying implementation of

the CPU, as well as the type of machine code that the CPU is capable of understanding. Two common architectures are Intel x86 (Rodgers and Uhlig, 2017) and ARM (Arm.com, 2018). Intel x86 is the architecture used by Intel CPUs, which are of very high performance and can typically be found in desktop computers and servers. ARM CPUs on the other hand typically consume less power and have a more compact machine code. ARM CPUs are typically used in embedded or mobile devices, such as modern smartphones (Gassée, 2016).

Since Intel x86 and ARM are different CPU architectures, they each have their own assembly language, and different assembler programs to translate that assembly language to the underlying machine code of the architecture. Thus, a *generic assembler* would be a program which is capable of understanding both x86, ARM, or any other type of assembly code, and translating that assembly code to some underlying machine code.

Chapter 3 Review of Existing Work

3.1 Review of Existing Assemblers

Before beginning to design a new generic assembler, it is useful to review existing assemblers, to gain an understanding of how they work and what useful features and common behaviors they share. The assemblers reviewed were the GNU Assembler (Web.mit.edu, 1991), Microsoft MASM (Msdn.microsoft.com, 2015), and NASM (Nasm.us, 2015).

All of the reviewed assemblers shared several common behaviors. They all function by accepting one input file and producing an object file containing the translated machine code. Command line flags could optionally be passed to the assembler to configure its behavior. All assemblers worked on only one architecture. If there was an error in the input assembly code, the assembler would highlight the erroneous code and the line number where it was located. All of the assemblers had support for labels for easy branching and data references, directives for controlling the attributes of the output object file, and macros to create more complex abstractions in assembly code.

While Microsoft MASM and NASM could only work on assembly code for the Intel x86 architecture, the GNU Assembler is actually a family of many different assemblers (Web.mit.edu, 1991), where each assembler supports one specific architecture. Each of these assemblers is a separate program, with a separate code base for that specific architecture. Thus, while on paper the GNU Assembler project might support many different architectures, it is not a generic assembler.

3.2 Review of Existing Literature

After reviewing the features and behaviors of common assemblers, a review of academic literature was done to examine previous work on generic assemblers. The first paper reviewed was "Automatic Assembler Generator for NoGap" (Karlström et al., 2010). In it, the authors described how they created a system called AsmGen, which automatically generates an assembler program (NoGapAsm) based on a processor specification written in NoGap (Karlström and Liu, 2009). The system works by having AsmGen generate intermediate files describing an assembly language's syntax and machine code generation rules. NoGapAsm then uses the metadata from these intermediate files to parse inputted assembly code, and then generate machine code.

The first thing that's notable about NoGapAsm is that it has a two-pass parser for the input assembly language code. The first pass verifies the syntax of the input code against the assembly syntax described in the intermediate files, while the second pass processes labels (Karlström et al., 2010). A major limitation is that NoGapAsm has a general-purpose parser which assumes that all assembly programs have the same basic syntax (Karlström et al., 2010). Another limitation of NoGapAsm is that it can only support fixed size instructions, which means it cannot work on variable sized instruction architectures similar to Intel x86. NoGapAsm also has only very limited support for directives, and it was

unclear if it was capable of handling instructions with relative addressing (such as branches or relative loads).

The second paper reviewed was "Functional Simulation Using Sim-nML", a master's project by Surendra Vishnoi (2006), which was referenced from the NoGapAsm paper. Vishnoi describes how she designed a program to generate simulators from a processor architecture specified in the Sim-nML *architecture description language* (ADL) (Vishnoi, 2006, Karlström et al., 2010). While the generation of simulators is a separate problem from the generation of assemblers, there were many ideas in this paper which could be re-used by a generic assembler.

For example, the paper describes how a processor could be specified in the Sim-nML language, and how that specification could later be processed to identify instruction encodings and behaviors for use in a simulator. While the Sim-nML language has too much additional information about the processor which is irrelevant when generating an assembler, the idea of encoding information about instructions in re-usable classes and generating instruction bitstreams from specially formatted 'images' was ultimately adopted for use in the generic assembler. However, because of the verbosity of the Sim-nML languages, the generic assembler ended up using a separate ADL which was simpler and easier to work with.

A big problem when designing the generic assembler was deciding on an ADL which was simple enough to work with, and yet expressive enough to describe a wide variety of different architectures. "The ArchC Architecture Description Language and Tools" describes one such ADL which has been successfully used to describe several complex real-world architectures such as MIPS, Intel, and SPARC (Azevedo et al., 2005).

ArchC is a SystemC based language for processor description. Each instruction in an architecture has its behavior, assembler syntax, and bitstream described with SystemC source code. This SystemC code is later parsed, verified, and used to automatically generate assemblers and simulators (Azevedo et al., 2005). The syntax which ArchC uses to describe assembly instructions served as inspiration for the custom ADL which designed for the generic assembler. However, as with Sim-nML, the ArchC ADL was judged to be too verbose, and too difficult to parse, so a separate ADL was designed for use in the generic assembler project.

The final paper reviewed was "The Automatic Generation of Assemblers" (Wick, 1975). The paper is directly related to the current project and describes the design and implementation of a generic assembler generator. While the paper is more than 40 years old, it provides several useful insights into common problems with making a generic assembler, as well as hints as to how the architecture of the generic assembler should be structured. For example, the paper describes how the assembler is divided into several modules for parsing assembly code, evaluating expressions, keeping track of symbols, and code generation. The generic assembler project ultimately adopted a similar modular architecture. Finally, the paper explores the pros-and-cons of using an existing ADL as input to the generic assembler, as opposed to a custom ADL. While Wick (1975) ultimately uses an existing ADL as input to the generic assembler, our project instead opted to use a custom ADL, in the interests of simplicity and ease-of-use by end users.

Chapter 4 Requirements

Upon reviewing existing assemblers and academic literature on generic assemblers, a list of requirements was drawn up based on their feature sets and the problem statement. It was determined that the generic assembler should have many of the same features and behaviors as a regular assembler, as well as the ability to work with user-defined architectures and assembly languages. Given the limited time available for the development of this project, a list of requirements was drawn up and prioritized using the MoSCoW method and is presented below.

Must:

- Present a command-line interface to users, similar to existing assemblers.
- Be able to load a user-supplied assembly source code file and architecture specification.
- Be able to output the assembled machine code.
- Offer options for users to print diagnostics and error information.
- Have a plugin system to allow users to extend the program capabilities.

Should:

- Be able to print verbose error messages when parsing assembly code, showing what the parser expected, and what it got instead.
- Be able to print informative error messages when parsing the architecture specification.
- Be able to output the machine code as either text, a raw binary blob, or embed it in an executable object file.
- Support common assembly language abstractions such as labels and data statements.

Could:

- Be able to display the abstract syntax tree (AST) of the parsed assembly code.
- Be able to display verbose information detailing the generation of the machine code.
- Include example code, architecture specification, and documentation.

Would like to have:

- Support for M4 (Gnu.org, 2016) macro expansion
- Assembler directives and string literal data statements
- Example architecture specifications for more exotic architectures (such as the Burroughs B5500 (Kimpel, 2018))

Chapter 5 System Design and Architecture

5.1 System Design

The functionality of the generic assembler was intended to closely resemble existing assemblers, such as MASM (Msdn.microsoft.com, 2015) or NASM (Nasm.us, 2015). To that end, the generic assembler was designed to act as a command-line program, which accepts all inputs over the command line, and writes all outputs to the terminal and/or a specified output file.

As with existing assemblers, the assembler generator takes as input the assembly source code to be assembled, as well as an optional list of command line flags to enable/disable certain features. Furthermore, unlike existing assembler programs, the generic assembler also requires as input a file which describes the architecture that it is assembling for. This is required because unlike existing assemblers, the generic assembler must be able to assemble any architecture, and not just one specific architecture.

While many *architecture description languages* (ADLs) exist, for the purposes of this project all of them were judged to be too verbose and hard to work with. For this reason, a novel and simple custom ADL was designed. This custom ADL is capable of simply describing the syntax of an assembly language, as well as how the machine code of that assembly code should be generated by the assembler program. A more in-depth description of the custom ADL is provided below.

While the custom ADL is capable of describing the bulk of a computer architecture, there are a few details in every computer architecture which proved impossible to easily define using the custom ADL. These details usually deal with how integers, data, and relative addresses are encoded in the machine code instructions of the architecture. To make the generic assembler flexible enough to handle most of these architectural details, a plugin system was designed which allows users to extend this part of the generic assembler. The goal is for the plugin system to allow end-users to create plugins that can interpret and encode different data types in different architectures. Furthermore, this same plugin architecture is used by the generic assembler to implement handling for a few different default data types. The idea is that users can use these default plugin implementations as a reference for when they implement their own plugins for their own custom architectures.

Both the generic assembler program and the plugin system are implemented in Python. Python is a non-standard choice for this kind of low-level programming (with most assemblers and compilers typically being implemented in C/C++). However, due to the limited time available for this project and the fact that the generic assembler wouldn't be interoperating with existing C/C++ systems, the simplicity and development speed of Python made it more a more attractive choice than the other languages evaluated for this project. Furthermore, because the generic assembler used Python, it was very simple to create a simple and easily accessible plugin system also based on Python, which is hopefully easier to use than a plugin system created in another language.

Unlike other systems programming languages, Python is dynamically typed. While this allows for much greater developer productivity, it also makes it much harder to understand interfaces written in Python, or to safely carry out refactoring. To combat these problems, the Python code in the generic assembler makes limited use of the *mypy* (Mypy-lang.org, 2018) optional static typing module for Python. Mypy allows for much better IDE assistance when typing and refactoring Python code, and it makes it much easier to understand certain complicated interfaces within the generic assembler.

5.2 Custom ADL Design

A key objective of the generic assembler is for users to be able to quickly make changes to the specification of a computer architecture, and then to be able to quickly test those changes out using the generic assembler. To that end, the generic assembler must have an interface which is both robust enough to capture the details of most computer architectures, and yet simple enough to be understood and modified quickly by users and parsed quickly by the generic assembler program.

It was judged that none of the existing ADLs met this design objective, so for the purposes of this project a new ADL was designed. The new ADL is novel, but its syntax was roughly inspired by the TatSu parser library (Tatsu.readthedocs.io, 2018), while its system of bitfield declarations and modifications is inspired by Sim-nML (Vishnoi, 2006) and ArchC (Azevedo et al., 2005). This new ADL presents an innovative new approach for describing computer architectures, in that it allows the user to define the assembly syntax and machine code generation rules at the same time. An example snippet of the custom ADL defining a subset of the x86 architecture is given below.

```
.BIT_FIELDS

name: short_opcode
size: 5

name: reg
size: 3

.ASM_INSTRUCTIONS

INSTRUCTION =
| %PUSH_INSTRUCTION%
;

PUSH_INSTRUCTION =
| push %32_BIT_REG%                    :: short_opcode=0101 0
;

32_BIT_REG =
| eax                                  :: reg=000
| ecx                                  :: reg=001
;
```

Figure 1: Custom ADL Example

The custom ADL snippet above defines the x86 *push register* instruction, where the register can be either *eax* or *ecx*. The ADL listing not only defines the syntax of the assembly code, but it also describes what the bitfields of the x86 instruction are, and how the bits should be set by the assembler to properly encode the instruction.

Each ADL listing is stored in a single text file, which is referred to as an *architecture specification*, or *spec* for short. This spec file is eventually passed to the generic assembler as input. The generic assembler then parses the ADL to build a syntax specification for the assembly language, and then it later uses that syntax specification to parse assembly source code (also passed in as input, in a separate file).

The ADL listing is divided into two separate sections, with the beginning of each section being defined by a directive. The first section begins with the *BIT_FIELDS* directive, and it is meant to describe all the possible bitfields of an instruction in the architecture being described. The second section begins with the *ASM_INSTRUCTIONS* directive, and it describes the syntax of the assembly language of the architecture, as well as which bitfields should be set in an instruction when encoding that instruction.

The *BIT_FIELDS* section is a simple list of *name* and *size* pairs, where the *name* field describes the name of a bitfield in an instruction, and the *size* field specifies how many bits are in this bitfield.

The *ASM_INSTRUCTIONS* section is much more complicated, as it describes both the assembly language syntax and how the bitfields are set during machine code generation. The *ASM_INSTRUCTIONS* section is composed of multiple *instruction definitions*, where each instruction definition begins with the name of the instruction definition, followed by a '=' character. The very first instruction to be defined is always *INSTRUCTION*, and this instruction is considered to be a parent to all possible other instructions.

Each instruction definition is composed of one or more *token patterns*. A token pattern describes a pattern of tokens which identify the instruction. Thus, these token patterns can be used to specify the syntax of the assembly language. The token patterns are listed underneath the name of the instruction definition, where each token pattern begins with a '|' character. The list of token patterns for each instruction definition is terminated with a ';' character.

There are three different types of tokens in a token pattern. The first and simplest is called a *raw token*, and it describes some string which must be present in order for the token pattern to be valid. In Figure 1 above, the "*push*" string in the "*push %32_BIT_REG%*" token pattern would be considered a *raw token*. This means that in order for this token pattern to be valid, the *push* string must always be present.

The second type of token is called a *placeholder token*. Placeholder tokens are identified by the '%' characters which surround their names. For example, in Figure 1 above, the "*%32_BIT_REG%*" string in the "*push %32_BIT_REG%*" token pattern is a placeholder token with the name "*32_BIT_REG*".

Placeholder tokens are a placeholder for another instruction definition declared elsewhere in the file, which shares the same name at the placeholder token. A placeholder token is automatically expanded by the parser into this other instruction definition. The placeholder tokens allow for the reuse of certain instruction definitions in several different parts of the specification. This cuts down on repetition in the specification, and it makes it much easier to declare certain instructions, by reusing instruction definitions from previous instructions.

The final type of token is called a *plugin token*. This is a special type of token, which looks like a raw token, except its string starts with 'int_' or 'label_'. The assembler generator can automatically identify these tokens according to their prefix, and it uses the plugin system to handle these tokens. Thus, plugin tokens allow the user to write plugins for the assembler generator which will be responsible for parsing and emitting the machine code of these tokens.

Besides token patterns, the spec in the *ASM_INSTRUCTIONS* section can optionally declare some *bitfield modifiers*. Bitfield modifiers are located on the same line as a token pattern, and each bitfield modifier begins with the '::' characters. For example, in Figure 1 above, the "*:: reg=000*" string is a bitfield modifier. The bitfield modifiers specify how a bitfield should be modified if a token pattern is matched. Each bitfield modifier is composed of a *name* and a *value*, which are separated by the '=' character.

The name of a bitfield modifier is located to the left of the '=' character and must match some bitfield name. All possible bitfield names are declared in the *BIT_FIELDS* section of the spec file. The value of a bitfield modifier is a string of '0' and '1' characters and specifies what the contents of a bitfield should be set to if a match is made for the token pattern. The size of the string of bits must match the size declared for that bitfield in the *BIT_FIELDS* section of the spec file.

Besides being a string of '0' and '1' characters, the value of a bitfield modifier can also be a placeholder value, surrounded by '%' characters. This indicates to the parser that the bitfield modifier's value will be emitted by the plugin system at a later time.

A great strength of the custom ADL is that it significantly simplifies machine code generation. Since the bitfield modifiers are tied to the token patterns, this means that it is possible to include them in the internal *abstract syntax tree* (AST) of the parser as it parses the assembly source code. Machine code generation can then be done by simply walking the AST of the parsed code and applying the bitfield modifiers at each node. More details of this approach are given later in the implementation section (chapter 6) of the report.

5.3 Plugin System Design

A key limitation of the custom ADL is that it is unable to define how data, integers, and labels of the architecture must be encoded. For example, in ARM assembly an integer immediate value is limited to any 32-bit value which can be expressed with an 8-bit base and a 4-bit rotation (McDiarmid, 2014). With more exotic architectures, these limitations on the integer values and the rules for

encoding them might be even more complicated. Also, it is impossible to predict the rules and encodings of every possible architecture.

To enable users to extend the generic assembler to handle parsing and encoding values from their own architectures, the generic assembler supports a Python plugin system based on Yapsy (Nion, 2015). Users may create their own plugin by simply adding Python code and an info file to the *plugins* folder of the generic assembler. The generic assembler then automatically detects these plugins, loads them, verifies their interface, and then uses them to parse and encode instructions and data.

There are two types of interfaces the plugin can implement. The first is an *int* interface, which handles the parsing and encoding of integer values. The second is a *label* interface, which handles encoding addresses and relative offset in instructions for that particular architecture. Specific details about these interfaces are given in the implementation section (chapter 6) of the report.

5.4 Architecture

The generic assembler is implemented in a modular architecture, where each module accepts some sort of input and returns some sort of output. Each module has a single purpose and is decoupled from the other modules. The modules can then be arranged in a pipeline, where the output from one module is fed as input to the next module. The output of the final module is the output of the program. Debug and diagnostic information can optionally be printed by each module to *stdout*. A diagram of the program architecture is given below.

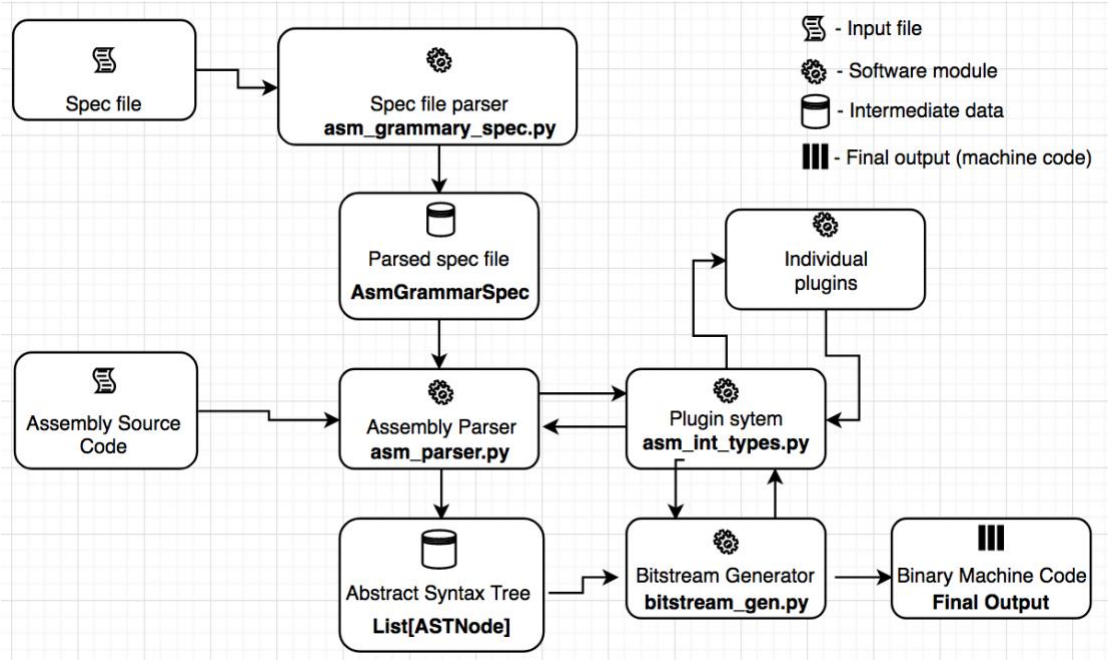

Figure 2: Software architecture diagram

Below is a list of all modules, along with a description of their purpose, inputs, and outputs. Further details about the implementation of each module are given in chapter 6 of the report.

main.py – Responsible for accepting all user inputs and passing them to the rest of the program. Also responsible for orchestrating all modules, by passing their inputs, getting their outputs, and passing that output to the next module.

asm_grammar_spec.py – Responsible for parsing the architecture specification (spec file) passed in as input by the user. Outputs an **AsmGrammarSpec** object which contains all syntactical information about the assembly language that will be parsed, as well as information about how the machine code should be generated for this architecture.

asm_parser.py – Responsible for parsing the assembly source code passed as input by the user. Uses information from the **AsmGrammarSpec** object to correctly parse the assembly code or highlight any syntax errors that might be present in the code. Relies on the **asm_int_types.py** module to communicate with the plugin interface to parse integers in the assembly code. The output of this module is an abstract syntax tree (AST) of the parsed assembly code.

asm_int_types.py – Responsible for acting as a bridge in between the plugin interface and the rest of the program. Upon startup, this module will detect and load all plugins, and verify that their interfaces are correctly implemented. This module also presents a simple interface for parsing integers and labels to the rest of the program and is responsible for routing calls from the program to the correct plugin.

bitstream_gen.py – Responsible for translating the AST of the parsed assembly code to a stream of bits (a bitstream). This bitstream is the machine code representation of the assembly code and is the final output of the generic assembler program.

obj_writer.py - A helper module which allows the user to inject the assembled machine code into an executable object file. The user can then execute this object file to run and test their assembled code. Template object files for Windows, OSX and Linux are provided in the *bin_templates* folder of the generic assembler program.

plugins – This is a folder which houses all plugins used for parsing and encoding integers and labels. The plugins in this folder are automatically detected, loaded, and verified by the **asm_int_types.py** module. By default, the folder contains a **builtin_types.py** file which implements some int and label types which the generic assembler can use. This file can also serve as a reference implementation for any future plugin development.

parse_utils.py – Utility class implementing methods which are helpful during parsing.

ast_utils.py – Utility class implementing methods to help print and display the AST of the parsed assembly code.

Chapter 6 Implementation

6.1 Custom ADL Parsing

Any execution of the generic assembler begins with at least two inputs: a spec file describing the syntax of the assembly language that will be parsed, and an assembly source code file written in that assembly language. Before the generic assembler can do anything, it must build a representation of the assembly language it is expected to parse. Thus, the very first thing the generic assembler program does is parse the input spec file which contains a specification of the architecture written in the custom ADL described in section 5.2. This parsing of the custom ADL is done by the **asm_grammar_spec.py** module.

The custom ADL parser works by parsing the input file character-by-character, and then attempting to match the characters against expected tokens. The very first thing the parser does is locate the *BIT_FIELDS* directive, and then attempt to parse any bitfield declarations defined underneath the directive. If there is a syntax error in the file, or some sort of unexpected data, the parser will print a detailed error message with a line number.

The ADL parser will continue parsing bitfield declarations until it reaches the *ASM_INSTRUCTIONS* directive. At this point it will start parsing instruction definitions, which describe the syntax of the assembly language. All instruction definitions begin with a name, followed by a list of token patterns. If there is a syntax error while parsing the instruction definition, the parser will print an informative error message along with the line number, and then exit.

All parsed instruction definitions are stored in an **AsmInstructionDefinition** object with the following fields:

```
class AsmInstructionDefinition:

    def __init__(self, name, line_num):
        self.name = name
        self.line_num = line_num
        self.spec_patterns = []  # type: List[DefinitionPattern]
        self.only_raw_values = True
        return
```

Figure 3: Custom ADL Instruction Definition Object

The *spec_patterns* field is a list of possible token patterns for the current instruction definition. Each token pattern also has an optional list of bitfield modifiers, which define what changes should be applied to the bits of an encoded instruction if that particular token pattern is present. Each pair of token patterns and bitfield modifiers is stored in a **DefinitionPattern** object.

```
class DefinitionPattern:

    def __init__(self, token_patterns, bitfield_modifiers):
        self.token_patterns = token_patterns
        self.bitfield_modifiers = bitfield_modifiers
```

Figure 4: Object to store token patterns and corresponding bitfield modifiers

All instruction definitions are stored in a Python dictionary, where the key of the dictionary is the name of the instruction definition, while the value is the **AsmInstructionDefinition** object which describes the instruction definition. This allows for easy and fast lookups of an instruction definition by its name. This dictionary is later passed to the assembly parser module as part of the assembly language specification (or *spec* for short). The assembly parser will do extensive lookups in this dictionary while it's parsing the input assembly code, so it was imperative that the interface for these lookups be simple and efficient.

When the parsing of the spec file is done, all the information is stored in an **AsmGrammarSpec** object as defined below:

```
class AsmGrammarSpec:

    def __init__(self):
        self.parsed_asm_instruction_types = False
        self.parsed_bitfields_definitions = False
        self.spec = {} # type: Dict[str, AsmInstructionDefinition]

        self.bitfields = []  # type: List[BitfieldDefinition]
        self.bitfield_indexes_map = {}  # type: Dict[str, int]
```

Figure 5: Object storing parsed specification file

The most important fields in this object are *spec,* which allows for each instruction definition in the custom ADL to be looked up by name, and *bitfields* which is a list of all possible bitfields (and their sizes) in the architecture being specified. Since the order of the bitfields matters, they are stored in a list, and a separate dictionary called *bitfield_indexes_map* allows the program to look up a bitfield by its name (by associating the name of a bitfield with its index in the *bitfields* list).

After the parsing of the spec is done, the finished **AsmGrammarSpec** object is verified for errors (such as references to undefined instruction definitions), and then it is passed to the assembly parser to be used in parsing the input assembly source code.

6.2 Assembly Source Code Parsing

The parsing of the input assembly source code is handled by the **asm_parser.py** module. This module takes as input the **AsmGrammarSpec** object produced by the custom ADL parser, and the input assembly source code. It produces as output an AST which can later be used to generate the bitstream of the assembled instructions.

The assembly parser is implemented in two passes. The first pass is responsible for parsing any labels in the file, and the second pass will parse the actual instructions. Two different types of label declarations can be recognized and parsed by the assembly parser. The first type of label is of the form '*string*:', where the label name is some alpha-numeric string followed by a ':' character. The second type of label parsing, which must be enabled with a special command-line flag, handles so-called Sigma16 labels. These labels must be the very first thing present on a line of assembly source code, and they are separated from the rest of the assembly code with whitespace characters. There is no ':' terminating

character. The names and line locations of all parsed labels are stored in a dictionary, which is then utilized in the second pass of the assembly parser.

The second pass of the assembly parser is responsible for reading the assembly source code line by line and parsing the assembly instruction (if any) on each line. The parser begins by checking if the line begins with a label, and then skipping it. It then attempts to parse the instruction on the line, using all possible token patterns defined for an *INSTRUCTION* in the **AsmGrammarSpec** object. If the instruction is successfully parsed, it creates an **ASTNode** object and adds it to the AST of the parsed code. A shortened version of the **ASTNode** object definition is given below:

```
class ASTNode:

    def __init__(self, token_type=None, token_value=None,
child_nodes=None, bm=None):
        self.token_type = token_type        # type: TokenTypes
        self.token_value = token_value      # type: str
        self.child_nodes = child_nodes      # type: List[ASTNode]
        self.bm = bm              # type: List[BitfieldModifier]
```

Figure 6: Definition for a node of the AST

The AST of the parsed assembly code is really a list of **ASTNode** objects. Each entry in the AST list corresponds to a line of parsed assembly code. Each entry in the AST is a root node which is of type *INSTRUCTION,* since every spec file's definition for an assembly instruction must begin with an *INSTRUCTION* definition. In turn, this *INSTRUCTION* root node has child AST nodes which correspond to the token patterns which were matched against the assembly input being parsed. For example, consider this AST which is produced by parsing the *push eax* instruction, for which an example specification was given in Figure 1:

```
INSTRUCTION
    PUSH_INSTRUCTION        :: short_opcode=01010
        'push'
        32_BIT_REG          :: reg=000
            'eax'
```

Figure 7: Example AST after parsing *push eax*

The AST begins with the *INSTRUCTION* node, which is the root node (and root definition in the specification) for all possible instructions. Next, we see that the parser was able to match the input against the *PUSH_INSTRUCTION placeholder* token pattern. This token pattern is composed of a *raw token* with the value 'push', followed by a *32_BIT_REG* placeholder token pattern. The *32_BIT_REG* token pattern is in turn matched with the 'eax' raw token.

Another thing to note about the above AST is the bitfield modifiers given in the right-hand column. These bitfield modifiers allow the assembler to easily build the bitstream of an instruction by simply walking across all the children of an AST node. More details about this approach are given in section 6.4.

Once the second pass of the assembly parser is done, the generic assembler will have a list of **ASTNode** objects, where each **ASTNode** in the list is the root AST node of some parsed instruction. This list is then passed to the bitstream

generation module, which will use it to generate the bitstream of the assembled machine code.

6.2.1 Handling Errors While Parsing Assembly Source Code

In order for the generic assembler to be usable and friendly to human developers, it needs to display useful error messages if it encounters any errors while parsing the input assembly source code. Most traditional assemblers are only responsible for parsing one flavor of assembly code, so they have special code in their parser to detect errors and display helpful error messages regarding them (Msdn.microsoft.com, 2015, Nasm.us, 2015).

Unfortunately, this approach is not possible with the generic assembler, due to the fact that it does not operate on a single well-defined assembly grammar, but instead on a custom grammar that has been specified and inputted by the user. Thus, the generic assembler cannot have special code to handle specific errors, because it doesn't even know what the input assembly grammar might look like.

To solve this problem, the parser of the generic assembler keeps track of every token pattern that it attempts to match against the input text, by pushing the token onto a special stack referred to as the *expected stack*. If it matches a token in the pattern, it leaves that token on the stack, and then attempts to match the next token in the token pattern. However, if it fails to match the token against the input text, if pops the token off the stack, and then attempts to match the next token pattern. If the generic assembler fails to match any token pattern, then it will display an error message with the longest stack it produced during parsing. The error message shows what it was able to parse, what it *expected* to parse next, and what it got instead which triggered the error.

The behavior of the stack can be best visualized with an example. Consider trying to parse the *push eax* instruction, where the parser has successfully matched the 'push' string and is attempting to match 'eax' string. The error stack of the parser can be visualized as follows:

```
['%PUSH_INSTRUCTION%', "'push'", "' '", '%32_BIT_REG%']
```

Figure 8: Expected stack before parsing 'eax' in 'push eax'

The stack shows that the parser is attempting to match a *PUSH_INSTRUCTION* instruction definition. Per the instruction definition given in Figure 1, the instruction begins with the 'push' and ' ' raw tokens, followed by a *32_BIT_REG* placeholder token. Next, the parser will attempt to match 'eax' as specified by the *32_BIT_REG* placeholder token. The stack will then look like this:

```
['%PUSH_INSTRUCTION%', "'push'", "' '", '%32_BIT_REG%', "'eax'"]
```

Figure 9: Expected stack after parsing 'eax' in 'push eax'

After the 'eax' string is match, all the tokens in the *PUSH_INSTRUCTION* token pattern have been matched, and the input string is exhausted. Thus, the parser has successfully parsed the *push eax* instruction, and it will discard the expected stack and proceed to parse the next instruction.

However, consider that instead of the *push eax* instruction, the parser is attempting to parse an instruction with a typo in it, such as *parse exx*. As with the above example, if the parser has already parsed the 'push' string then the error stack will look like this:

```
['%PUSH_INSTRUCTION%', "'push'", "' '", '%32_BIT_REG%']
```

Figure 10: Expected stack before attempting to parse 'exx' in 'push exx'

Next, per the *32_BIT_REG* instruction definition from the spec, it will attempt to match 'eax':

```
['%PUSH_INSTRUCTION%', "'push'", "' '", '%32_BIT_REG%', "'eax'"]
```

Figure 11: Expected stack when trying to match expected 'eax' to actual 'exx'

Since 'eax' does not match 'exx', the match will fail, so the parser will pop 'eax' off of the expected stack. Next, per the *32_BIT_REG* instruction definition, the parser will attempt to match 'ecx', which will result in the following expected stack:

```
['%PUSH_INSTRUCTION%', "'push'", "' '", '%32_BIT_REG%', "'ecx'"]
```

Figure 12: Expected stack when trying to match expected 'ecx' to actual 'exx'

Again, since 'ecx' does not match 'exx', the match will fail, and the parser will pop 'ecx' off of the expected stack. The parser will then attempt to match the instruction against a few other token patterns, but none of them will produce a match or an expected stack larger than the one from Figure 11. Since the instruction *push exx* fails to match any known token pattern from the input spec, the parser will fail to parse it, and the generic assembler will fail with the following error message:

```
Assembler ERROR: Unable to parse INSTRUCTION on line 1
PARSED: %PUSH_INSTRUCTION% 'push' ' ' %32_BIT_REG%
EXPECTED: 'eax'
INSTEAD GOT: exx
```

Figure 13: Error message displayed when failing to parse 'push exx'

In this error message, the user can see what the parser was able to parse from the erroneous instruction, what it expected next ('eax'), and what it got instead ('exx'). This allows the user to quickly locate the error in the instruction (line 1, 'exx'), and understand why the parser was unable to parse the text (expected 'eax' or some other register, got 'exx' instead, which is probably a typo).

This demonstration shows how the error handling mechanism is flexible enough to handle any possible grammar which might be inputted, while still being able to show useful error messages based on that grammar.

6.3 The Plugin System

The generic assembler implements a plugin system, which allows users to extend its parsing and instruction emitting modules. The purpose of the plugin system is

to allow users to extend the generic assembler to implement the details of an architecture which are otherwise too complex to be defined in the spec file. Like the generic assembler, the plugin system is implemented in Python, and all plugins must be written in Python as well.

The generic assembler uses the YAPSY (Nion, 2015) Python library to help implement the plugin system. YAPSY is a Python plugin framework which greatly simplifies the discovery and loading of plugins. Each execution of the generic assembler begins with it using YAPSY to detect and load all plugins in the *plugins* directory.

Each plugin is expected to implement a *get_registered_types* function, which returns a list of supported data types. After a plugin is loaded by YAPSY, the generic assembler will query it using *get_registered_types* to get a list of data types it supports, and then it will verify that the plugin correctly implements the interface for each data type.

There are two possible data types in the plugin system, each with their own interface which must be implemented by the plugin. The first is called an *int* type, and it allows the plugin system to define an interface for parsing and emitting bits for a custom type of integer. The second type is called a *label* type, and it allows the plugin to calculate and emit the bits for memory addresses and relative offsets.

6.3.1 The int Data Type

All *int* data types must have a name which begins with 'int_'. The plugin must implement three different functions for each *int* data type it supports:

- `chars_<`*int type name*`>`
- `verify_<`*int type name*`>`
- `emit_<`*int type name*`>`

For example, if a plugin supports a data type called *int_sigma16*, then it must implement the following three functions:

- `chars_int_sigma16`
- `verify_int_sigma16`
- `emit_int_sigma16`

If the plugin fails to define and implement any of these three functions, the generic assembler will fail to load it and will display an error message.

The *chars* function takes no parameters and will simply return a list of characters that can be used in assembly source code to express integers of this type. This character whitelist will be used by the assembly parser to read the integers in the source code character by character.

The *verify* function accepts the string of the parsed integer as input and will return a Boolean specifying if that string is a correct text representation of the integer. The *verify* function will be called by the assembly parser module, to

verify that the custom integer string it has parsed is in a format supported by the plugin.

The *emit* function accepts the string of the integer as input and returns a string of bits which is the bitwise representation of the integer. The *emit* function is used by the bitstream generation module to convert all parsed integers to their bitwise representation, which is then embedded in the instruction bitstream.

Once a plugin has defined an *int* data type, the data type can be used in the specification of an architecture to define placeholders for integers and bitstream modifiers. For example, consider the following instruction definition:

```
DATA_STATEMENT =
| data int_sigma16_data      :: displacement=%int_sigma16_data%
;
```

Figure 14: Example instruction definition using custom *int* data type

The above specification defines a *DATA_STATEMENT*, which begins with a 'data' text string, and is followed by a custom *int* data type named *int_sigma16_data*. The parsing and verification of integer strings of this data type will be handled by the *char_int_sigma16_data* and *verify_int_sigma16_data* functions, which must be implemented in a plugin. The bits for this instruction definition will be emitted by the *emit_int_sigma16_data* function, also implemented in a plugin. The *emit* function will accept as input the string representation of the integer and will return a string of bits. The *displacement* bitfield in the instruction bitstream will then be set to this string of bits. The example illustrates how a custom *int* data type defined in a plugin can be used in a spec to extend the parsing and bitstream generation capabilities of the generic assembler.

6.3.2 The label Data Type

The second data type that can be implemented by a plugin is the *label* data type. The purpose of this data type is to enable the plugin system to handle the encoding of memory addresses and relative offsets in the instruction bitstream. For example, some branch instructions in the x86 architecture use *relative addressing*, which means that the destination of a branch is computed relative to the address of the branch instruction itself. Thus, for the generic assembler to be able to handle relative addressing correctly, it must use *label* data types to properly handle relative offsets in the instruction bitstream.

Similar to the *int* data type, all *label* data types must have a name which begins with 'label_'. Unlike the *int* data type, the plugin which defines a *label* data type must only implement a single function named "`calc_`<*label type name*>".

The *calc* function of the *label* data type is called by the bitstream generator, and it accepts two arguments. The first argument is the current memory address of the instruction in the bitstream, while the second argument is the memory address of the destination label in the bitstream. From this information, the *calc* function is able to compute the offset and emit its bits in the proper form for use in the bitstream.

The *label* data types are used in spec files in almost the same way as *int* data types:

```
JUMP_INSTRUCTION =
| jump label_sigma16[%REG_A%] :: displacement=%label_sigma16%
;
```

Figure 15: Example instruction definition using *label* data type

The above example specifies an instruction definition called *JUMP_INSTRUCTION*, which expects a 'jump' text string, followed by the name of the label, followed by a register name enclosed in brackets characters. Unlike the *int* data type, labels are simply defined as an alpha-numeric string, and the label string being parsed must match the name of some label defined elsewhere in the assembly source code.

During bitstream generation, if an instruction's bitfield modifiers make a reference to a label placeholder, the generic assembler will attempt to locate the memory address of the label being referenced. It will then pass the source instruction's address and the label's memory address to the *calc* function of the label data type, which is implemented in a plugin. The *calc* function will then compute the bits which represent the label, and will return these bits to the bitstream generator, which can them embed them in the instruction's bitstream.

Thus, for the above example, the bitstream generator will use the *calc_label_sigma16* function of the plugin to calculate the bits of the label, and then will place those bits in the *displacement* bitfield of the instruction bitstream. The example demonstrates how the plugin system is able to calculate addresses and offsets for use in the assembled machine code from label references in the assembly source code.

6.4 Bitstream Generation

Once the assembly source code has been parsed into an AST, the generic assembler will use this AST to generate the machine code of the program being assembled. The machine code is the binary representation of the assembled code, and it contains the machine instructions which the CPU will interpret to execute the program. In the generic assembler, the machine code is represented as a stream of bits, or a *bitstream* for short. The generic assembler generates the bitstream by walking the AST node by node and generating a bitstream for each individual instruction. The bitstream of all instructions is the final output of the generic assembler. All code related to bitstream generation is contained within the **bitstream_gen.py** module.

The bitstream generator module takes three inputs, and from them generates the machine code of the assembled program. The first parameter is the specification of the architecture for which it is assembling machine code. This is necessary to identify all the possible bitfields of an instruction. The second parameter is the AST of the parsed assembly source code. The final parameter is the *imagebase*, which is an integer specifying at which memory location the assembled program

will be loaded. This is necessary to correctly compute the memory addresses and offsets in the generated machine code.

The bitstream generator works in two passes. The first pass will generate the bulk of the actual machine code, while leaving placeholder values for memory addresses and offsets. The second pass will take each placeholder value and use the *label* data types of the plugin system (described in section 6.3.2) to replace the placeholder with a correctly calculated memory address or offset.

6.4.1 First Pass: Machine Code Generation

The first pass of the bitstream generation will generate most of the machine code for every instruction in the program. The assembler works by iterating across every node in the AST list of the parsed assembly code and generating a bitstream by applying the bitstream modifiers specified in every child node.

For each AST node in the AST list, the assembler begins by first reading the list of all possible bitfields from the architecture specification. For example, using the specification from Figure 1, the following bitfields can be present in an instruction:

- Name: `short_opcode.` Size: `5`
- Name: `reg.` Size: `3`

Next, the assembler will iterate across all the child nodes of a root node taken from the AST list, and will apply any bitfield modifiers it finds in each child node. If a bitfield has a bitfield modifier applied to it, then that bitfield will be present in the final assembled instruction. Otherwise, if a bitfield has no bitfield modifier referencing it, then that bitfield is ignored. For example, take the AST of the parsed *push eax* instruction:

```
INSTRUCTION
    PUSH_INSTRUCTION        :: short_opcode=01010
        'push'
        32_BIT_REG          :: reg=000
            'eax'
```

Figure 16: AST node of parsed *push eax* instruction

In the children of the root AST node, there are two bitfield modifiers:

- Name: `short_opcode.` Value: `01010`
- Name: `reg.` Value: `000`

After iterating across all the children of the root node, the following bitfields will be set in the instruction:

```
  short_opcode    reg
--------------  -----
         01010    000
```

Figure 17: Final bitfields of assembled *push eax* instruction

Thus, the final bitstream of the assembled *push eax* instruction will be 01010000, which is equivalent to 0x50h hexadecimal. This is the correct machine code representation for the x86 *push eax* instruction.

A more complicated example would be the x86 *jmp* instruction. To demonstrate the example properly, a few more bitfields must be added to the example specification from above:

- Name: `opcode`. Size: `6`
- Name: `opcode_d`. Size: `1`
- Name: `opcode_s`. Size: `1`
- Name: `short_opcode`. Size: `5`
- Name: `reg`. Size: `3`
- Name: `immediate32`. Size: `32`

The x86 *jmp* instruction is a branch instruction, which changes the instruction pointer to another memory location. When writing assembly source code, the user typically defines a label at a certain location in their code, and then uses the *jmp* instruction to branch to that location. For example:

```
jmp test_label3
nop             ; <- this code will be skipped by jmp instruction

test_label3:
nop
```

Figure 18: Assembly code using branch and label

Thus, the specification for the *jmp* instruction will specify that a label is expected after the "jmp" text string:

```
JUMP_IMMEDIATE =
| jmp label_x86_rel_32_bit_branch            :: opcode=1110 10
                                              :: opcode_d=0
                                              :: opcode_s=1
                     :: immediate_32=%label_x86_rel_32_bit_branch%
;
```

Figure 19: Example specification for x86 *jmp* instruction

Note how the specification assigns a placeholder value to the *immediate_32* bitfield. This means that the bitfield *immediate_32* will be set to a value that will later be calculated by a plugin.

The AST node of the parsed *jmp* instruction from Figure 18 is displayed below:

```
INSTRUCTION
    JUMP_IMMEDIATE    :: opcode=111010 :: opcode_d=0 :: opcode_s=1
        'jmp'         :: immediate_32=%label_x86_rel_32_bit_branch%
```

Figure 20: AST Node of parsed x86 *jmp* instruction

Finally, the assembler can produce the bitstream for this instruction by simply iterating across all the child nodes of the root AST node, and applying their bitfield modifiers to the bitfields. This gives the following bitstream:

```
  opcode    opcode_d    opcode_s                        immediate_32
--------  ----------  ----------  --------------------------------
  111010           0           1  00001000000000000000000000000000
```

Figure 21: Example bitstream of x86 *jmp* instruction

Note how the *short_opcode* and *reg* bitfields are missing from the bitstream. This is because none of the bitfield modifiers from Figure 20 referenced them, so they were not included by the assembler in the final bitstream of the instruction.

While the first pass of the bitfield generation module is responsible for generating the bulk of the machine code in the above example, the second pass is responsible for correctly populating the *immediate_32* bitfield. The next section will explore how this is done.

6.4.2 Second Pass: Calculating Label Addresses

In the previous section, Figure 18 demonstrated an x86 *jmp* instruction which branched to a label address further along in the code. This situation is particularly hard for the generic assembler to handle because it doesn't know at which memory location the label will be located until it is done generating the entire bitstream for the program. The reason for this is that x86 is a *variable length instruction set*, which means that different instructions may have different sizes. So, it's impossible to predict how many bytes will sit in between the *jmp* instruction and its destination label, until the assembler is done assembling the entire program.

To handle this situation, the bitfield generation module has a second pass, which operates on the assembled machine code. This second pass is responsible for identifying all instructions which use a label in the machine code and updating their placeholder values in the instruction bitstream with correct offsets/addresses to reference their destination labels.

The second pass functions by iterating across all the bitfield modifiers of every instruction's bitstream. If it finds a bitfield modifier which is a special type of placeholder modifier, it takes the address of the instruction and the address of the destination label and passes them to the plugin interface (as described in section 6.3.2). The plugin interface then locates the correct function to calculate the bitstream for the label offset, passes the source and destination addresses to this function, and then returns the bitstream calculated by this function. The bitstream generator finishes by setting the proper bitfield in the instruction to the calculated bitstream.

Once the second pass is finished, the bitstream generation of the program is finished, and the resulting bitstream can be outputted to a binary file. Alternatively, it can be passed to the **obj_writer.py** module to be injected into an object file, which lets the user conveniently execute and test the code that was assembled.

Chapter 7 Evaluation and Testing

The generic assembler was evaluated and tested by implementing specs and assembly programs for three different architectures: x86, ARM, and Sigma16. Each of these architectures is very different than the other two, and the generic assembler's ability to work with these architectures is evidence of its flexibility. The following sections will give a description of each architecture and how the generic assembler handles it.

7.1 Evaluating x86 Support

The Intel x86 architecture is a CISC architecture with a variable length instruction set. It is the most common CPU architecture in consumer desktop computers and is one of the most widely used architectures in the world (McGrath, 2018). CISC is short for *Complex Instruction Set Computers*, and it describes computer architectures that have very extensive and complex instruction sets which support a wide array of operations (Chen, Novick and Shimano, 2000).

To test the generic assembler, a subset of the Intel x86 instruction set was specified using the custom ADL. This subset of x86 included branch instructions (such as *jmp*), memory and data operations (such as *mov* and *add* with memory references), stack operations (such as *push*), and more. Next, several different small x86 assembly programs were written using the instructions defined in the specification. Finally, the spec file and the assembly source code were passed to the generic assembler for assembling.

The output of the generic assembler was evaluated in two different ways. First, the machine code produced by the generic assembler was passed through *Capstone*, a disassembler library capable of handling several different architectures (Quynh, 2018). The disassembly listing produced by Capstone was compared to the original assembly source code, to make sure the two matched. A test assembly program and its corresponding disassembly listing are given below:

```
Capstone Disassembly Listing:        Original Assembly Source Code:

0x1000:push eax                    | push eax
                                   | test_label1:
0x1001:mov  al, byte ptr [eax]     | test_label2: mov al, byte ptr[eax]
0x1003:xor  eax, ebx               | xor eax, ebx
0x1005:jmp  0x1012                 | jmp test_label3
0x100a:mov  word ptr [ebx], ax     | mov word ptr[ebx], ax
0x100d:mov  eax, 1                 | mov eax, 1
                                   | test_label3:
0x1012:mov  bx, 0x1337             | mov bx,   01337h
0x1016:mov  al, 0xff               | mov al, -1
0x1018:push 0x1001                 | push test_label1
0x101d:jmp  0x1001                 | jmp test_label2
```

Figure 22: Capstone disassembly vs original x86 source

The second way that the x86 integration was tested was by writing a small "Hello World" program in x86 assembly, and then using the generic assembler to assemble and inject it into an object file. The object file was then executed to make sure the assembled code ran correctly.

To inject assembled machine code into object files, a helper module called **obj_writer.py** was implemented. This helper module would be passed the assembled machine code, and a template object file. It would then inject the machine code into the object file, overwriting any previous machine code that was present. The user could then run the resulting object file to execute the assembled machine code within it. This method proved to be faster and easier than implementing a fully functional linker to produce executable object files. Template object files for Linux (32-bit ELF), OSX (32-bit MachO), and Windows (32-bit PE) are provided by the generic assembler, but the user can easily add their own.

Despite the x86 architecture having a complex instruction set with variable length instruction encodings, the custom ADL of the generic assembler was able to describe it without too many difficulties. The bitfield definition scheme of the custom ADL allows support for variable length instructions seamlessly, while the syntax description of the assembly language is robust enough to handle even complex x86 instructions. Finally, while the x86 instruction set has relatively complicated relative-offset branching capabilities, the plugin system of the generic assembler was able to describe and implement the correct encoding for x86 branch instructions with ease. Thus, the generic assembler is capable of specifying enough of the x86 architecture to implement small fully functional programs, and with a bit of work could probably handle an even larger part of the x86 architecture.

7.2 Evaluating ARM Support

The ARM architecture is a RISC architecture with fixed-size instructions. RISC is short for *Reduced Instruction Set Computer,* and it denotes a computer architecture that is simpler than CISC architectures (Chen, Novick and Shimano, 2000). This means that RISC instructions tend to have less features, but in turn have faster instruction execution speeds and a more compact and optimized instruction set. Whereas the x86 architecture dominates desktop and server computing, the ARM architecture is popular on mobile and embedded devices (Turley, 2014).

As with the x86 architecture, a subset of the ARM architecture was specified to test the generic assembler. This subset of instructions is able to handle stack operations (such as a multi-register *push*), data processing instructions (such as *mov* and *add*) and memory loads and stores (*ldr* and *str*). To test the ARM architecture support, a small assembly program was implemented and assembled with the generic assembler. The resulting machine code was then disassembled, and the disassembly listing was compared to the original assembly source code to ensure accuracy.

The ARM architecture supports two different byte orders: little-endian and big-endian. The specification was written to target big-endian ARM (in contrast to little-endian x86). This was deliberately done to evaluate the generic assembler's capabilities across two different byte orderings. In the end, there were no notable

differences in writing a spec for a big-endian architecture vs a little-endian architecture.

While the ARM instruction set is supposedly simpler than x86 (due to being RISC vs CISC), it was surprisingly difficult to describe in the custom ADL of the generic assembler. One problematic example is the multi-register *push* instruction, which can push multiple registers onto the stack at the same time. To correctly support this instruction, a token pattern had to be defined for every possible number of registers being pushed, which is incredibly repetitive.

While the syntax description facilities of the custom ADL lead to some repetition when implementing the ARM architecture specification, the plugin system was able to handle the different encodings of ARM integer representations, even the relatively complicated ARM barrel shifter (McDiarmid, 2014). This is further evidence of the plugin system's flexibility and its ability to properly implement even the most complex encoding schemes.

7.3 Evaluating Sigma16 Support

The third and final architecture evaluated for use with the generic assembler is Sigma16. Unlike ARM and x86, Sigma16 is a research architecture intended for educational purposes, and is used at the University of Glasgow (O'Donnell, 2013).

Sigma16 is incredibly small and simple compared to both ARM and x86. This made it relatively simple to create a custom ADL specification which encompassed all of Sigma16. The specification was tested on the examples which ship with the Sigma16 simulator (O'Donnell, 2013). After assembling the example programs with the generic assembler, the resulting machine code was loaded and executed in the Sigma16 simulator manually, to confirm that it behaved the same as the original assembly code (which could also be assembled and executed by the simulator).

One limitation of the generic assembler that was noted during the evaluation of Sigma16 support is that it was unable to properly handle Sigma16 labels. Unlike labels in ARM and x86 assembler, Sigma16 labels can optionally appear at the beginning of a line of code, and each line can have only one label associated with it. Furthermore, Sigma16 labels do not terminate with the ':' character. To properly handle Sigma16 labels, special support for them had to be implemented in the generic assembler, and a special command line flag for enabling Sigma16 label support was added to the command line interface. The problems encountered with parsing Sigma16 labels highlighted the limitations of the custom ADL and the plugin system and should be considered in any further work done on the generic assembler.

7.4 Test Automation

The generic assembler is automatically tested with end-to-end tests, implemented in the **run_tests.py** script. This script will test every spec and assembly source file available in the *test* folder and will throw an error if any test fails. Where possible, the assembled machine code generated by a test is compared against an expected disassembly listing.

Chapter 8 Conclusion and Future Work

The generic assembler project was able to successfully describe and assemble machine code for three different architectures. On top of that, the generic assembler presents users with a custom ADL and plugin system that allows them to implement their own architectures, and further extend the generic assembler in novel new ways. However, in spite of the successes of the generic assembler, the evaluation work described in Chapter 7 revealed several limitations which should be addressed by any future work.

The first and most critical limitation is the inflexible parsing of labels. Currently the generic assembler makes several assumptions about how labels are declared and used in the assembly source code, and it's not possible for the user to change this. This was a problem that had to be coded around in the Sigma16 evaluation, and it's probable that it will be a problem with other assembly languages. In the future, this can probably be solved by extending the plugin system to handle the parsing of labels, similar to how it currently handles the parsing and validation of integers.

While the custom ADL of the generic assembler is simple and flexible, its simplicity sometimes makes it very repetitive to write architecture specifications, where lots of simple and similar operations are repeatedly specified many times due to minor changes in syntax. The custom ADL should probably be extended with further abstractions to help reduce repetition and make specifications less verbose.

The generic assembler also lacks several convenient features which are taken for granted in modern assemblers. Directives allow users to divide code into sections, to enable special features in the assembler or linker, and to define special types of data. Macros allow programmers to reduce repetitive assembly code, and to create abstractions which let them better organize their assembly source code. Implementing these features in the generic assembler would greatly increase programmer productivity and user friendliness.

Despite these limitations, the generic assembler is still a powerful tool for quickly prototyping and experimenting with computer architectures. The ability of the generic assembler to assemble programs for three very different architectures showcases its flexibility, while the simplicity of the custom ADL allows for users to quickly get started and make changes in their specifications. Thus, the generic assembler successfully fulfills its goal of providing users with an easily reconfigurable assembler tool which can be used with a wide variety of different architectures for experimental purposes.

References

Arm.com. (2018). *The ARM Architecture*. [online] Available at: https://www.arm.com/files/pdf/ARM_Arch_A8.pdf [Accessed 4 Sep. 2018].

Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., Araujo, C. and Barros, E. (2005). The ArchC Architecture Description Language and Tools. *International Journal of Parallel Programming*, 33(5), pp.453-484.

Chen, C., Novick, G. and Shimano, K. (2000). *RISC vs. CISC*. [online] Cs.stanford.edu. Available at: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ [Accessed 5 Sep. 2018].

Clang.llvm.org. (2018). *Assembling a Complete Toolchain — Clang 8 documentation*. [online] Available at: https://clang.llvm.org/docs/Toolchain.html [Accessed 4 Sep. 2018].

Fowler, M. (2010). *Domain Specific Languages*. [online] Martinfowler.com. Available at: https://martinfowler.com/books/dsl.html [Accessed 4 Sep. 2018].

Gassée, J. (2016). *A company that doesn't really make chips dethroned Intel with super savvy business moves*. [online] Quartz. Available at: https://qz.com/741078/a-company-that-doesnt-really-make-chips-dethroned-intel-with-super-savvy-business-moves/ [Accessed 4 Sep. 2018].

Gnu.org. (2016). *GNU M4 1.4.18 macro processor*. [online] Available at: https://www.gnu.org/software/m4/manual/m4.html [Accessed 4 Sep. 2018].

Karlström, P. and Liu, D. (2009). NoGAP: A Micro Architecture Construction Framework. In: *Embedded Computer Systems: Architectures, Modeling, and Simulation 9th International Workshop*. [online] Springer, pp.171-180. Available at: https://link.springer.com/chapter/10.1007/978-3-642-03138-0_18 [Accessed 4 Sep. 2018].

Karlström, P., Loganathan, S., Akhlaq, F. and Liu, D. (2010). Automatic assembler generator for NoGap. In: *6th Conference on Ph.D. Research in Microelectronics & Electronics*. [online] IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/5587152/ [Accessed 4 Sep. 2018].

Kimpel, P. (2018). *Burroughs B5500 - retroComputingTasmania*. [online] Retrocomputingtasmania.com. Available at: http://www.retrocomputingtasmania.com/home/projects/burroughs-b5500 [Accessed 4 Sep. 2018].

McDiarmid, A. (2014). *ARM immediate value encoding*. [online] Alisdair.mcdiarmid.org. Available at: https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/ [Accessed 4 Sep. 2018].

McGrath, D. (2018). *Intel's Microprocessor Share Slips Below 60%*. [online] EE Times. Available at: https://www.eetimes.com/document.asp?doc_id=1332968 [Accessed 5 Sep. 2018].

Msdn.microsoft.com. (2015). *Microsoft Macro Assembler Reference*. [online] Available at: https://msdn.microsoft.com/en-us/library/afzk3475.aspx [Accessed 4 Sep. 2018].

Mypy-lang.org. (2018). *mypy - Optional Static Typing for Python*. [online] Available at: http://mypy-lang.org/ [Accessed 4 Sep. 2018].

Nasm.us. (2015). *NASM*. [online] Available at: https://www.nasm.us/ [Accessed 4 Sep. 2018].

Nion, T. (2015). *Yapsy: Yet Another Plugin SYstem — Yapsy 1.11.223 documentation*. [online] Yapsy.sourceforge.net. Available at: http://yapsy.sourceforge.net [Accessed 4 Sep. 2018].

O'Donnell, J. (2013). Connecting the Dots: Computer Systems Education using a Functional Hardware Description Language. *Electronic Proceedings in Theoretical Computer Science*, 106, pp.20-39.

Quynh, N. (2018). *The Ultimate Disassembly Framework*. [online] Capstone-engine.org. Available at: http://www.capstone-engine.org/ [Accessed 5 Sep. 2018].

Rodgers, S. and Uhlig, R. (2017). *Intel's X86: Approaching 40 and Still Going Strong*. [online] Intel Newsroom. Available at: https://newsroom.intel.com/editorials/x86-approaching-40-still-going-strong/ [Accessed 4 Sep. 2018].

Tatsu.readthedocs.io. (2018). *竜 TatSu — 竜 TatSu 4.2.6 documentation*. [online] Available at: https://tatsu.readthedocs.io/en/stable/ [Accessed 4 Sep. 2018].

Turley, J. (2014). *Intel vs. ARM: Two titans' tangled fate*. [online] InfoWorld. Available at: https://www.infoworld.com/article/2610369/processors/intel-vs--arm--two-titans--tangled-fate.html [Accessed 5 Sep. 2018].

Web.mit.edu. (1991). *Using as - Overview*. [online] Available at: http://web.mit.edu/gnu/doc/html/as_1.html [Accessed 4 Sep. 2018].

Vishnoi, S. (2006). *Functional Simulation Using Sim-nML*. MSc. Indian Institute of Technology, Kanpur.

Wick, J. (1975). *Automatic Generation of Assemblers*. Ph.D. Yale University.