University of Glasgow | School of Computing Science

Generic Assembler

Nishad Mathur

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 25, 2016

Abstract

The demands on a processor are ever increasing, Moore's law is no longer applicable and now research is underway to extract every iota of efficiency from the hardware. One approach is design custom hardware for your needs, another is a custom extension to an existing instruction set; in both of those circumstances an assembler is needed. Developing an assembler that can keep evolve with rapidly evolving hardware is a difficult problem, fraught with challenges.

This project aims to alleviate some of these, by providing a tool optimised for rapidly creating and evolving an assembler with the architecture. The end product of this project is a table driven assembler. This assembler is designed to support a variety of architectures and syntaxes while still allowing the user the ability to implement real world architectures.

There have been many tools in this area but many have focused on different concepts, flexibility, performance or ease of use. Rarely aiming for all three. This tool is not the most flexible, nor the most powerful but it aims to strike a good balance between each of its three goals. Allowing both novice and experienced users to achieve their targets.

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 format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: __________________ Signature: ____________________

Contents

Chapter 1

Introduction

1.1 Background

In recent years the performance scaling of processors has failed to meet the lofty expectations set by Moore's Law[37, 28], an observation that the number of transistors in a processor doubles every two years and others have made follow on predictions that performance would double every 18 months (not indefinitely)[20].

Unfortunately that period of exponential scaling is reaching an end, Intel has recently increased the lifetime of their processes from 2 years to 3 [6] and now the physical limitations imposed upon processes have begun to become a limiting factor upon the scaling of processes and hence performance.

As a consequence new approaches and optimisations have to be considered as a alternative method to allow processor performance to continue to rise. One of the many approaches which is being explored is to try new alternative architectures which attempt to optimise for the real world workloads which they expect to be utilised for.

One example this approach is the Mill CPU architecture[36], which attempts to approach the limitations and performance bottlenecks with several novel solutions. The Mill is only a single approach to this problem and there are many other approaches which may be considered and alternative architectures which may be utilised in conjunction with these approaches.

A common approach to this issue is to implement extensions to existing architectures, the process of designing these extensions and developing requires that the hypothetical performance characteristics of these instructions be tested (whether it be done by simulation on on physical hardware is irrelevant). Thus the need for an assembler which can rapidly be altered to implement new instructions, alter existing ones and iterate through the process rapidly.

1.1.1 Assemblers

Assemblers in general are a key component in the process of writing almost any software for a specific architecture, they convert the low level assembly into the opcode format specific to an architecture. They handle the process of laying out the opcodes in memory, calculating offsets, instruction selection among other tasks.

The earliest assembler were created just to act as simple translators, converting instructions (also referred to as a first-generation programming language) into machine code, with no complex functionality involved. But as time as time has progressed the features expected in an assembler have increased, the features which they must implement to support the various assembly languages have grown ever more complex and sprawling; significantly complicating the assemblers themselves and complicating the process of implementing or customising an assembler.

Thus meta assemblers were created, to allow individuals to save on several factors; cost (for the case of paid assemblers), ease of modification rapid implementation for new platforms as well among other factors.

1.2 Motivation

The value of an assembler being optimised for the use case of rapid development or prototyping of a new or existing instruction set has bee established above, but the scope for such a tool extends beyond implementing new instructions for existing architectures but the issue of implementing an assembler remains the same. Extending an existing assembler or creating a new one from scratch can be a difficult, time consuming and error prone process, a set of failings which this tool is designed to ease. It is designed to do so by providing an extensible and configurable tool designed to rapidly build, test and iterate over the various instruction set architectures.

The overarching aim of this project is to make it easy for users to both a higher and lower experience levels to rapidly alter the properties of an assembler as to test the effectiveness of their instruction set architecture, processor or one of many other potential properties of a system.

The use of this tool by less experience users, for example as a tool utilised to develop an understanding of the complexities of altering an existing instruction set by providing a mechanism by which assembly language programs may be run on this newly extended processor.

The aim of this tool is then to enable users of both and and low experience levels to develop an assembler for a new or extended instruction set in a rapid and flexible enough manner as to implement a wide variety of architectures while also providing sufficient aid to the novice user.

1.3 An Explanation of Terms

This section provides an approximate explanation of how each term is used in this report.

Assembler tool which converts a textual description of a machine language program and converts it into a binary form. Not all assemblers are as primitive as that, there exist some which are in-fact fairly sophisticated and implement a variety of complex features.

Assembly Language This is a language which an assembler digests and compiles to produced an executable machine code program. The syntax for these need not necessarily to be architecture specific but may also be tied to the specific tool which compiles them, for example there are the Intel and AT&T syntaxes for x86 assembly, both of which are different in fairly significant ways (argument order, for example) but generate fairly equivalent machine code for the same instruction sets.

Instruction Set (Alternatively referred to as instruction set architecture (ISA) in this paper) Define the set of operations which a CPU can interpret and execute. For example, 'load' or 'jump'.

Architectures In the context of this document, 'architecture' refers to the instruction set architecture. Below are the various architectures chosen for the evaluation, explained in brief.

- AARCH64/ARMv8 ARM's 64bit architecture[3] . It has 31 general purpose registers and instructions are typically 32bits wide. It is a fairly large instruction set which is derived from AARCH32 (ARM's previous 32bit architecture). It design is driven in part by a desire for backwards compatibility [15]
- MIPS I[10] More specifically, MIPS R2000, which is an implementation of the MIPS 1 ISA. It is a simple 32bit instruction set which also implements 32bit wide instructions.
- Sigma16 Is a 16 bit architecture which has only 3 instruction formats. It is a very simple but consistent, well designed architecture.
- Intel x86-16 Or more specifically the 8086[22] is a 16bit implementation of Intel's x86-16 ISA and is Intel's first implementation of the ISA.

Opcodes The portion of a machine code instruction which indicates which instruction is to be executed. For example, in Sigma16 the first 4 bits indicate what the instruction to be executed is (for most instructions).

JVM Java Virtual Machine[19], the runtime on which this program runs. It is cross platform, single binary, managed runtime for multiple programming languages, such as Java, Scala, Python, Javascript, Kotlin and many others.

Kotlin[25] It is a modern language designed to be fully compatible with Java while providing modern conveniences and type safety improvements.

DSL A domain specific language in the context of this document specifically refers to embedded domain specific languages. These are effectively an approach to designing a program's API such that they are designed to optimise for readability and simplicity. They often do not follow the normal conventions of the platform and optimise for the specific use case for which they were built, describing an assembler syntax in this context.

YAML YAML (YAML is not a Markup Language)[11] is a data serialisation format optimised for human readability.

Program Specific Terminology

This is terminology which is used to refer to specific components of the actual system. Some of these are also in common use.

- Reference Reference is a blanket name which encompasses all of the various types of literals and types, for example literals, Registers, etc.
- Indexed Are a type of reference which are composed of multiple sub-references, for example a memory reference is composed of an address and an offset.
- Literals Literals are the actual numeric values used by the user. For example, hexadecimal literals (0x01), integer literals (12), etc.
- Labels This term refers to either the declaration of a label, which is used as anchor to calculate addresses or offsets or it may refer to a reference to such a label.
- Mapped These can be defined as a set of user defined literals, for example the registers may be defined using this mechanism.
- Meta Types These are the container types which encapsulate other types.

Chapter 2

Context

2.1 Related Work

There have been many extremely capable systems of this class which have been previously developed. The goals, capabilities and focus of these tools has been wide but they fail to meet the requirements outlined above. Below is a short description of each and explanation of how each differs from the aims and requirements of this projects.

2.1.1 LLVM TableGen

Clang TableGen[32, 5] is tool and DSL developed under the LLVM umbrella of projects and is designed to be an extremely flexible and powerful tool utilised inside LLVM and Clang for a variety of purposes, one of these being the basis of LLVM's assembler back ends.

TableGen offers an extremely large amount of flexibility and power due to its roots as in the C++ template language. The use of a C++-like languages gives it its power but also makes it much less approachable to new users and due to its inherent flexibility and flexibility.

TableGen's key advantage over this tool is its flexibility, but that is enabled by its DSL only architecture and the requirement that the user implement functionality which is provided automatically in the generic assembler. These can be factored into a shared library but this implements extra overhead on the user.

Despite its nature as both a very expressive and capable tool LLVM's TableGen lacks sufficient allowances for new users as to be usable on the context in which this tool is intended to be used[33].

2.1.2 Simple Generic Assembler

Simple Generic assembler[8] was an assembler designed to allow users to experiment with alterations to the DCPU-16 processor[40] but capable of being used for other architectures. DCPU-16 is a cpu designed for use in the (now abandoned) game 0x10^c [39], as the central processing unit(CPU) from which a player's space ship is controlled. It was a simple 16 bit CPU with an equally simple assembly language.

The simplicity of this language has consequently led to significant limitations in the flexibility of this tool. For example there is no mechanism by which the syntax of labels can be altered. It also offers no mechanism by which the assembler can support type checking of the arguments, macros or a variety of other features. The lack of typing checking effects both new and experienced user, in experienced users will not receive the aid they require with their errors and experienced users will not be able to implement any mechanism by which instructions can be selected on the basis of their operand types. The syntax of the description format, although very elegant, is not particularly clear, not obvious which also negatively effects its utility for inexperienced users.

The final issue is that the assemblers development appears to have ceased and it has been left in an incomplete state. This issue, in conjunction with the aforementioned issues, have rendered this tool an inappropriate fit for this use case, it offers neither the aids which would aid a new users nor the flexibility required by an experienced one.

2.1.3 Cross-32 Meta Assembler

Cross-32[30] is also a table driven meta assembler, much like the tool developed by this project, and is capable of targeting a large number of different architectures. The tool had not been updated in approximately two years (at the time of development) but is still capable of running on modern operating systems and hardware.

Despite the large number of target architectures supported, the tool makes no mention of supported assembly language dialects and as no documentation is available prior to purchase. Thus this feature could not be investigated. The only documentation available[31] indicates that the tool only supports its custom assembly dialect which is not customisable and fails to meet the requirements of this tool.

The most significant issue with this tool is its nature as a commercial software offering, a single licence retails for 79.99 which makes it infeasible to expect users to have access to the tool, hence it is not useful in a teaching context.

2.1.4 Axasm

Axasm[44] is a cross assembler written in C, it has built in support for a variety of architectures but lacks customisability in terms of supported assembler syntaxes. Its usage of the C preprocessor and the C language as its host language for its embedded DSL gives it the flexibility to support a wide variety of host architectures but makes it a non-intuitive tool to use. This lack of ease of use and the limited syntax support reduce its suitableness of the desired use cases.

2.1.5 A Generative Approach to Universal Cross Assembler Design

The paper ”A Generative Approach to Universal Cross Assembler Design”[4] describes a generative generic assembler system which accepts several files (for parsing and lexing the assembly language file) as well as several modules of code which describe how the parsed file should be transformed into an assembly language.

Issues with this system include the long setup time involved, it claims that it takes 20-30 days to implement a new assembler using this system. This alone reduces the utility of this system but its decision to use YACC and LEX to implement its lexing and parsing also significantly increases the difficulty of its use for end users.

2.1.6 Project Maxwell Assembler System

The project Maxwell Assembler System[35] is an assembler library written in Java with the intention of enabling users to write assemblers and disassemblers for a variety of complex assembly languages.

This system is implemented as a collection of tools which accept an assembly language description and generate a java assembler and decompiler for that assembly language, as well as generating an automated test suite for that assembly language. The tool was utilised for the Maxine Virtual Machine.

The tool appears to be powerful and well tested but appears to lack any parsing mechanism and is entirely designed for generating assembler for use through a library interface. This limitation and its nature as a generative assembler make it unsuitable for these requirements.

Chapter 3

Requirements

The system was designed and built at the request of Dr John O'Donnell. The basic intent of the project was to create a table driven assembler which can be reconfigured for various assembly languages.

The intent of this chapter is to provide a high level overview of the requirements, both functional and non-functional for this project as well as provide a context as to their importance to the tools effectiveness. The functional requirements section shall provide a high level overview of the feature set and capabilities of the system. The non-functional requirements will provide an overarching description of the amorphous characteristics of the system, such as speed, ease of use, etc; the more subjective constraints.

3.1 Use Cases

To highlight a particular example, one class of users who may be interested in this tool are researchers experimenting with various instruction sets. The context for this user is that they may be attempting to implement a new instruction for an existing instruction set. This instruction set may be a complex, real-world instruction set, such as ARM's ARMv8[3] (which is a modern 64bit instruction set, designed for high performance, low power devices). Alternatively it may be a synthetic architecture, custom built for the researcher's own purpose, for example Sigma16.

For this user, an alternative to implementing a generic assembler would be to extend an existing assembler for the architecture. This has several benefits over those of a generic assembler, most importantly that it already exists and is known to work correctly on the architecture. But altering an existing assembler is neither always practical, nor possible, for example being able to alter an existing assembler is conditional on having access to the source code for that assembler and for more obscure architectures, there may be no open-source assembler available to extend. Even beyond that issue lies another, which is the difficulty of learning and altering an existing code base, it is a time consuming process and it may well be that the user's task requires alterations to multiple assemblers, which necessitates learning, altering and verifying multiple code bases, any of which may not even be amiable to the required alterations, as they may have been designed to prioritise performance or other characteristics over flexibility and simplicity. The characteristics which would be ideal when extending an assembler.

3.2 Design Considerations

Both the Sigma16 and ARM architectures possess similarities, the general syntax and overarching approach to language design, but they also differ significantly, the most significant difference being the the binary form the of assembly language its self. On the whole there the syntax of the assembly languages possess far more in the way of commonalities than significant divergences, although differences most certainly do exist (the more problematic examples being related to expressions, indexing and assembler directives), the common syntax of the assembly languages offer a useful subset of the languages' functionality to create useful programs.

The more problematic features of the languages are those which lie on the intersection of syntax and behaviour. The large variety of different indexing modes in ARM are a rather apt example, it has ten or more addressing methods in the assembly language[38], but these typically map down to just 2 or 3 different instruction (pc relative and register indexed in ARMv8). These are both rather simple to handle from a syntactic point of view and to translate from a machine code point of view but the number of them makes it a rather laborious process, hence necessitating a solution (detailed further in item 3e) for a rather common problem.

Expressions are a far less common, but still very problematic feature of expression languages, x86 assembly typically features these (for example, for constant expressions). These may be used as form of self documentation, for example to demonstrate how a constant is derived and typically improve the users ability to comprehend the code. The complexity though is in parsing of the expressions and the differences in how each assembler handles each of the components of an expression.

Directives are potentially the most difficult feature to support in a manner which allows for a variety different architectures to be supported. The complexity here is primarily is related to (but not identical to) the issues faced in implementing expressions. Which is to say, each assembler requires different directives to be implemented for full support of the language and each assembler requires these to be implemented in similar but (often subtly) different manners. There are also directives which change the global behaviour of assembler or are useful for interacting with external tools, such as linkers.

The rapid iteration which is required during experimentation creates a set of implicit non-functional requirements which the tool needs to satisfy, most importantly a rapid turn around time and a simple mechanism for maintaining several similar variations of the same language and context switching between them. This biases the tools overarching design away from an assembler generator tool set, which would read the configuration file, generate an assembler into a target language, compile it and then run the assembler. Instead a more appropriate approach is that of a table driven assembler, which accepts the configuration file, parses it, configures its self for the assembly language and then loads, parses, assembles and outputs the assembled file.

The table driven approach is better suited to these requirements for several reasons, the least of which is in the simplicity of using the tool. It requires neither any intermediate files nor any external tools to utilise, which should make it simpler to pick-up and use for a new user. The lack of any form of compile stage also eliminates any time or performance overheads relating to the compiler, which in certain languages can be significant. The time saved by not running multiple command and compiling intermediate languages will improve the iteration time for the user, by reducing the time spent in the tool.

The traditional benefits offered by generated a assembler over those of offered by an interpretive one are similar to those offered by a compiler over an interpreter[1]. The most obvious similarity being the performance trades each offers, which is to say the overhead of parsing and configuring an assembler each time the generic assembler is run, instead of a slower single generation and compilation pass of the assembler. It also permits the assembler its self to be designed in a more performant manner, instead of incurring the natural overheads required for a table driven approach. Despite the performance advantage of the compiler assembler the advantage it offers is largely negated by the rapid iteration requirement of the program, which by definition necessitates frequent changes to the assembly language configuration and hence pushes the performance bottleneck from the assembling stage to the build/configuration stage. Hence, it becomes practical to consider the reconfigurable approach.

The other consideration is that the reconfigurable assembler is built to facilitate experimentation and rapid iteration, not for use in a production environment. It is not guaranteed, but the vast majority of of users for this tool will be focusing on smaller assembly language files, where any overhead from slow assembly would be subsumed by other factors, such as the assembler but also IO, etc.

Another potential use case is that of a student utilising this as a tool to aid in their understanding of computer architectures, for example implementing a new instruction for an existing architecture, or implementing a trivial architecture from scratch. In general, the requirements for this are very similar to those required by a researcher rapidly iterating on various approaches and designs but they do still require different feature sets and the non-functional requirements are the most significantly different.

One features which provides great value for this user is the ability to freely exchange the tool and the configuration files between users in a simple manner.

The student can not be counted on to have the tool installed on their system, nor can any OS be taken for granted hence the tool should ideally be cross platform; a simple requirement which the JVM fills handily, as it is cross platform for both code and the 'binary' (the JAR file in this case) which permits the student to download a single binary and use it in on which ever OS they may be using, with little to no hassle.

The system also permits easy editing and distribution of the language description using its YAML[12] configuration format, a human readable and editable data serialisation format. It is designed as a white space sensitive language which reduces the redundancy and in theory make the users experience with the tool smoother, there were unfortunately issues with this, but those are discussed further in section 5.1.3. Despite the specific issues which were raised, the general approach appears to be fairly effective for the user.

Clear and specific error messages are a feature which was extremely desirable in general, but also in particular for this group of users. They will likely only use the tool on a limited number of occasions, thus will not have the experience to decipher the more complicated and archaic error messages which the tool may potentially emit. New users are very likely to make frequent and often unexpected errors during their usage of the tool. For example, indenting errors, spelling-mistakes, incorrect usage of language constructs and missing fields, thus the tool should attempt to make clear where the user has erred in an effort to help them avoid the error in the future and to fix the immediate issue and get the system running.

3.3 Functional Requirements

These requirements enumerate features which are desired but may have been designated out of scope for this project. The priorities are explained in section 3.3.1.

1. Configurable Assembler
   (a) **Configuration File** The system should be able to parse a configuration file, configure the assembler and then parse, assemble and output the assembly language file.
   (b) **Domain Specific Language (DSL)** The user shall be able to configure and extend the configurable assembler through means of a domain specific language, which replaces the configuration loading, parsing and assembler configuration stages of the generic assembler.
2. Handle Assembly File

   (a) **Load Assembly File** It should load the file; selecting it based either on input from the command line, or by directly passing a file handle.
   (b) **Parse Assembly File** It should take the assembly file from the previous stage and pass it into a configured assembler which then ingests it and parses it as necessary. Converting it into its internal format in the process.
   (c) **Assemble Assembly File** It should then assemble the file into machine code, handle offset, macro substitution, validation, error messages, etc and output a binary stream.
   (d) **Output Assembled File** It should optionally output the binary stream to a configured file if used in CLI mode.
   (e) **Output Assembly Listing** The system should also be be capable of generating the annotated assembly listing of the program.

3. Assemble Assembly File
   (a) **Handle Instructions and Arguments** It should be able to parse instructions in an arbitrary format, parsing the arguments into the various formats, compound types such as index operations, literals (as above), registers and other constants types. It can use the bit manipulation capabilities to combine these arguments and instructions into the defined instruction output format.
      i. **Handle Literals** It should recognise literal constants in an arbitrary format and allow them to be converted into binary output.
      ii. **Handle Registers** There should be a mechanism to declare register constants, which are converted into binary form during the output stage. There should also be a mechanism to generate a range of registers for large register files.
      iii. **Nested Types** The ability to create a type which is composed of two or more nested types.
      iv. **Handle Labels and Offsets** It should be able to detect and register labels. Calculate their offsets and then substitute in the values at references. Optionally the assembler should also support pc-offset jumps for architectures such as ARM.
   (b) **Handle Binary Output** Control how the instructions are assembled into their binary form.
      i. **Handle Endianness** It should be able to convert the binary sequences from its native java big endianness into the required endian formats required for certain architectures.
      ii. **Handle Arbitrary binary concatenation** The system should be capable to concatenating binary sequences of arbitrary length correctly and output them as a single binary stream. This is required (for example) flag fields on instructions.
      iii. **Handle Arbitrary binary subscript ranges** The system should be able to subscript an arbitrary range of bits from a binary sequence and in conjunction with the above capabilities supports capabilities such as split fields where a single argument is split into multiple bit sequences in the instructions' output stream.
      iv. **Addressing Modes** The system should be able allow label references to have their offsets calculated using various approaches, as necessary for the architecture. For example, relative addressing.
      v. **Expressions in Description** Allow the user to embed arbitrary expressions into the description file. This would be useful things such as allowing user defined addressing modes.
   (c) **Assembler Directives** These are commands to the assembler which alter its behaviour, for example it may be to treat the next instruction as a segment identifier, or inform it that the follow text is a comment, etc. The definition of what is a directive can be broad and vary between assemblers.

      These vary significantly between assemblers and architectures and their scope is potentially endless.
      i. **Segments** Are an assembler directive which inform the assembler as to which memory segment the next instruction(s) should be located within.

ii. **String to Data Directive** A directive which can convert a provided string into a binary stream, to be outputted into the binary form of the file stream.
iii. **Handle Macros** The system should be able to support a basic macro system. They should at the bare minimum be able to be defined in the architecture definition file but it may be potentially valuable to support inline macro definitions in the assembly file.
iv. **Handle Comments** It should extract comments and preserve them for the assembly listing.
v. **Expressions in Assembly** Allow the user to embed arbitrary expressions into the assembly file.

(d) **Linker Support**

i. **Linker Integration** Implement support in the assembler for emitting information to allow the output from the assembler to be digested by a linker.
ii. **Object Formats** This is a subset of linker integration, it allows the assembler to generate output in the form of preset object formats, such as Relocatable Object Module Format[21].
iii. **Import Statement** The system may permit the user to perform a direct inclusion of externals files in a similar mechanism to how macros are instantiated, but this requires more comprehensive error reporting functionality. It may include support for import tables.
iv. **Export Statement** The system may permit the user to designate symbols for export, and may support export tables.

(e) **Testing**

i. **User Evaluation**
ii. **Implement Multiple Architecture Definitions**

3.3.1 Requirement Classification

These are the requirements of the system, categorised using the MoSCoW[7] system. The specific terminology for each section is explained in brief below. These were only the preliminary classifications and were subject to change.

Must Have

The intent of this category is to enumerate the features which the assembler must have implemented to be considered a success and be feature complete.

Should Have

This section lists the features which would be considered high priority but not an absolute requirement for implementing the assembler.

Could Have

These features are the features which are ”nice to have” but not a requirement. Implementing these are entirely optional but will improve the end users quality of life.

Requirement	Comment
Configuration File	
Load Assembly File	
Parse Assembly File	
Assemble Assembly File	
Output Assembled File	
Output Assembled Listing	
Handle Literals	
Handle Registers	
Nested Types	
Handle Labels and Offsets	This does not include support for all potential addressing modes, which is a far more complex task.
Handle Comments	
User Evaluation	
Implement 1 Architecture	This is the absolute bare minimum level of testing which should be implemented.

Table 3.1: The Requirements the System Must Implement

Requirement	Comment
Handle Endianness	Having some level of support for this is extremely important, but it is not an absolute requirement.
Handle Arbitrary binary concatenation	Being able to concatenate arbitrary binary sequences is important for most architectures.
Implement 2 Architectures	This is a reasonable but still a little low of a level of testing to perform.

Table 3.2: The Requirements the System Should Implement

Won't Have

These are the features which may be desirable but lie outwith the scope of this project and will be very unlikely to be implemented.

3.4 Non-Functional Requirements

1. Detailed Error Messages The system shall be designed to provide the user with detailed and meaningful error messages to help them track down the faults in their configuration and assembly files.
2. High Level of Flexibility for Configurations The assembler should be be flexible enough to be configured for the common assembly language formats, but need not accommodate every single format; the DSL fills in that gap.

Requirement	Comment
Domain Specific Language (DSL)	This feature offers potential benefits in terms of reconfigurability but is not sufficiently so to overcome the implementation complexities.
Handle Arbitrary binary subscript ranges	This may be useful for some architecture but none of the considered architectures require this.
Addressing Modes	These may be vital for some architectures but is not a requirement for all architectures.
Segments	Segments will be implemented on a needs basis, as an architecture requires them.
String to Data Directive	This would be extremely useful but the implementation details of this vary from assembler to assembler, so like other assembler directives it will not be implemented.
Handle Macros	This too will be implemented if time allows. MIPS utilises it, but it is not a necessity to implement a useful subset of the architecture.
Implement 3 or 4 Architecture	This is the ideal level of testing to perform on the system.

Table 3.3: The Requirements the System Could Implement

Requirement	Comment
Import Statement	This could be potentially useful but is not a necessity for the initial release. Once Macros are implemented this could extend their implementation.
Export Statement	This could be potentially useful but is not a necessity for the initial release. Once Imports are implemented this could extend their implementation.
Linker Integration	This would be very nice to have but is not a requirement nor particularly useful for implementing support for bare metal architectures 9not important for use case).
Object Formats	Not a requirement for implementation of assemblers for specific use case.
Embedded Expressions	This feature would be useful for more advanced features but would be complex to implement in a useful manner (e.g. operator support for SizedBinaryArrays).
Full Compatibility with Assembler Directives	This features is infeasible to implement in the configuration format without support for embedded expressions.
Expressions in Assembly	This feature's scope is far outwith the scope of this assembler.

Table 3.4: The Requirements the System Won't Implement

3. Relatively easy to use The system should not be extremely difficult to use for users with a medium level of experience.

4. Cross Platform Support The system must run on multiple operating system, preferably OSX, Windows and Linux.

Chapter 4

Implementation

This chapter begins by describing the various stages of the implementation, the initial prototype, the development which took place for each iteration of the architecture implementation cycle. The purpose of this chapter is to give the reader an overview of the development process, as well as the decisions which were made and aid them in understanding the structure of the program.

4.1 Initial Prototype

The very first stage of this project was to create a simple prototype for the assembler. It was developed in Swift as a command-line OSX application. It was initially implemented as a very simple assembler which only handled a very trivial fixed instruction set. It only handled registers and fixed instruction and wasn't capable of converting to binary.

The initial principal for this design was to aim for the simplest possible design, with no consideration for extendibility or reuse, and the implement it. Features would then be layered on top of this underlying mechanism in a piecemeal manner until a reasonably capable tool was produced. This would then be reviewed and its limitations reflected upon, before using those as an aid to develop and to guide the overall architecture of the assembler. The intent of keeping it simple was to avoid building evermore excessive abstractions and levels of indirection into the system. A degree of over abstraction was unavoidable but regardless, minimisation was important for the goal of simplifying the architecture, while keeping it extensible.

This initial design was very limited and unable to cover even the most basic of use cases, hence it was iterated upon further. The design eventually evolved to use the ”DataSource” and ”Instruction” abstractions. Data Sources were Swift enumerations which contained the data for each each of the 4 fundamental types for the assembler: register, literal, label and memory. These 4 types are what later evolves into the ”reference” types for the generic assembler, but these are far more rigid in their design and scope.

An example of this rigidity is in the use of enumerations and pattern matching their use ensures that adding new types or altering the behaviour of a reference requires changes in a great many number of places and ties the implementation of each reference into the implementation of the assembler its self, thus reconfiguring it became an extremely error prone and time-consuming task.

Thankfully, the limited scope of the references (only the 4 type for this trivial assembler) and the lack of an extensibility requirement meant that for this prototype this mechanism was workable, but it would need to be rethought for the final assembly, where configuration was a key goal.

The instructions themselves were also implemented as an enumeration which was a more problematic design decision. It become impossible to share the implementation between each instruction and it became a significant burden to make any alterations to the assembler.

Despite the rather burdensome overhead of extending the prototype further alterations were made to implement support for new features, such as operator overloading for the instructions as well as support for a rudimentary error reporting mechanism.

This initial implementation of operator overloading was rather useful and informed the overall design of the feature (the pitfalls to avoid, as well as more suitable architectures) in the final assembler.

For example, the instructions were all generated from a single pattern matching block, which had a block of code for each of meta-instructions, where the meta identifier (the name which aliased each of the sub-instructions) and concrete instructions (a uniquely named instruction with a fixed type signature) all fed into a single code block. This block then disambiguates which concrete instruction is actually to be instantiated.

This mechanism was very simple and adding a new instruction was a trivially simple exercise, but it had several key weaknesses: the constant repetition of similar, but non-identical type signature check blocks and the other was a more significant problem, which was that the references were parsed and had their type assigned before any of this contextual information could be provided. This issue wasn't significant for literals or the other basic types, but became a significant issue for indexed references, which needed to (potentially recursively) type check their nested references.

The key takeaway form this portion of the prototype was then to defer the reference parsing to as late a stage as possible and secondly, to adopt a delegation style approach to instruction parsing, where the instruction is an object, responsible for parsing its self once it has been correctly identified.

4.2 Language Choice

When designing a system with the intent that it be extensible language choice is an important consideration as it may well be the primary interface through which the user interacts with the system. Thus some care and consideration was put into the process of deciding on the final language for the generic assembler.

There are an extremely large number of languages which could have been considered for this short list but it was completely infeasible to consider every single possible language to find the ideal fit, thus the process began from a fairly broad list of languages, which was then narrowed down to a list of five.

The five candidates were: Java[14], Python[13], Kotlin[25], Swift[2] and Haskell[34]. Java for its relative simplicity, cross-platform support, library ecosystem and large user base. Python for similar reasons as well as its meta-programming capabilities and its excellent standard library. Kotlin was a more interesting candidate as it is fully and bidirectionally compatible with Java but with a fresh take on the same features as well as additional features to improve type safety and reduce boilerplate. Swift is to Objective-C as Kotlin is to Java and offers several extra syntax features which suit this system but was not cross-platform. Finally Haskell is a purely functional programming language.

4.2.1 Java

Java offered many advantages for the assembler, the JVM being one of the most significant. The JVM offered a single write once, compile once and run anywhere platform where a single binary can be easily compiled and

```java
public class HelloWorld {

    public static void main(String... args) {
        System.out.println("Hello, World");
    }

}
```

Figure 4.1: A very simple hello world program in Java

distributed of to the users of the tool. It also possessed a rather comprehensive standard library with the facilities to implement most of the required functionality. The language also offers the benefits of familiarity both for the developer of the tool as well as for other potential users or developers of the software. One of the other benefits of Java it is a static typed language which provides a convenient self documentation mechanism as well reducing the need for unit tests in detecting errors introduced during the refactoring process.

The JVM is another useful asset of Java, it allows any JVM language to interoperate with each other, which allows the user or the tool's developer to use the most appropriate language for the task. Which allows the user to account for the weakness of Java and to interoperate with libraries from their language of choice.

The issues it possess relate to its weak meta-programming facilities and its null safety issues. The weak meta programming would not typically be a significant issue but for the case of a reconfigurable assembler it can potentially simplify several of the abstractions used in the assembler. The lack of null safety on the other hand means that it becomes a potential source of errors throughout the program. It posed a potential risk in the configuration parsing phase of the assembler where there were a significant area where nulls may have entered the system as well as in the context of using the system as a DSL. In a DSL the user can not be relied upon to know all the potential areas where nulls may or may not be returned thus is becomes a potentially significant source of errors.

Overall, Java was a strong candidate for the assembler and during the initial phases of implementing the generic assembler it was actually used as the primary implementation language. The key factors behind this decision were its cross-platform support, its library ecosystem and familiarity with the language and its ecosystem.

4.2.2 Python

```python
if __name__ == '__main__':
    print("Hello World!")
```

Figure 4.2: A very simple hello world program in Python.

Python was another strong candidate for this system, but the strengths and weakness appear of this language appear to be the mirror of those posses by Java, most significantly are its excellent meta-programming facilities but conversely, its lack of static typing and (at the time of consideration) lack of any form of type hinting is a detracting factor.

May of the advantages of Python are shared by Java, its large and comprehensive library ecosystem, a large user base, a fully featured language and the developers prior experience with the language. In fact, Java the language offers far fewer syntactic conveniences than python does, which is an important (but not key) point of consideration for this tool. Due to its lack of static typing the language also offers powerful meta-programming facilities[9] which may potentially have simplified some of the languages abstractions. For example, it may have made it feasible to use the same system for both the library and configuration system with no changes. It would also have made implementing embedded expressions trivially easy, due to pythons interpreted nature.

Despite the benefits which the language offers Python's lack of any type hinting mechanism or static typing would significantly increase the number of unit tests required, as any re-factorings are significantly higher risk for the user as there is no longer any automated type checking. The other weakness of python is that at the time of this consideration there was no mechanism for creating monolithic python executables, this makes distributing and running the tool unnecessary complicated for the end user, it was not an insurmountable issue, but it was an unnecessary complication.

Overall Python is another strong contender for the language and its meta-programming is its most attractive feature, but overall its lack of static typing and lack of a monolithic executable reduce its overall suitability to below that of Java's.

4.2.3 Kotlin

```kotlin
fun main(args : Array<String>) {
  println("Hello, world!")
}
```

Figure 4.3: A very simple hello world program in Kotlin

Kotlin, like Swift, is a very young language which was still under active development when the initial selection was being made but it offered several compelling features: a very clean syntax, improved semantics, several rather convenient syntax extensions, a modern and clean standard library[25] as well as full bi-directional compatibility with Java.

```kotlin
 val startIndex = (range["start index"] as? Number)?.toLong()
    ?: throw InvalidOption("start index", range)
```

Figure 4.4: Kotlin Configuration Parsing Code.

```java
long startIndex;

Object startIndexUntyped = range.get("start index");
if (startIndexUntyped != null && startIndexUntyped instanceof Number) {
    startIndex = ((Number) startIndexUntyped).longValue();
} else {
    throw new InvalidOption("start index", range);
}
```

Figure 4.5: Java Equivalent to the Kotlin Configuration Parsing Code.

The improved syntax of Kotlin provided several mechanisms which would ultimately prove useful in this tool, for example it has a short hand syntax for 'get-or-if-null-return/do' which was predicted to be useful in configuration parsing (fig. 4.4). This proved to be far simpler and shorter than the Java equivalent of the same feature as in fig. 4.5 which is both less easy to follow and more difficult to scale for more complex conditions. In actual usage in the system it was very heavily throughout the system as part of its error handling mechanism. The language also offers improved null safety as one its improved semantics, which offered benefits by removing an entire class of bugs. The improved standard library was also a positive signal for the language as it offered many useful convenience methods which simplify common operations. Another extremely useful feature of Kotlin over

Java is its extension method mechanism, which allow new methods to added to an existing class, which offered to make several situations significantly simpler, for example converting big-integers into byte arrays.

Kotlin's most compelling feature is its bi-directional compatibility with Java, where any and all constructs of Java can be accessed in Kotlin, using Kotlin's semantics and syntax, and any feature of Kotlin's is mapped to idiomatic Java, properties to getters and setters, companion objects, extensions methods and free functions to static methods, etc. This is a very useful property as it allows any weakness or bugs in Kotlin to be overcome by using Java where appropriate. It also allows the language to use any existing Java tools with the language, such as Maven and the plethora of JVM tools (debuggers, byte-code weavers, etc) with minimal friction. It also offered the advantage that should the language have proven to be poorly suited for the tool, it would have been possible to switch languages mid implementation without discarding any previous work.

This compatibility was in fact utilised early on in the project, where the early implementation of the language was a pure Java implementation of the assembler. Initially the decision was made that despite the benefits offered by Kotlin due to its lack of stability and lack of a community Java would be the better choice for the generic assembler. But during the implementation process it gradually became apparent that the tool would benefit from many of the quality-of-life improvements which Kotlin offered, such as the null-coalescing operator and statements-as-expressions would improve the implementation. Thus the implementation switched to using Kotlin for any new components of the language and eventually IntelliJ's automatic Java-to-Kotlin translation tool[18] converted the remaining Java components of the tool into Kotlin.

The aforementioned stability issue for Kotlin was a significant issue initially as the language was still in its late beta stages when the project was undertaken and there were backwards incompatible changes made to the language on a fairly regular basis. The other, much less significant issue, was the authors lack of prior experience with the language, but due to the relative similarities to Java those were not an insurmountable concern. The compatibility breaking changes and the potential bugs in the early releases were the biggest concerns in selecting the language and what ultimate led to it being discounted in favour of Java.

Overall were it not for lack of stability, there would have been no qualms in selecting Kotlin as the primary language for the tool. But initially those concerns outweighed the potential benefits and it was not selected, later though its stability improved and value of its features became evidence and it became the primary implementation language.

4.2.4 Swift

```swift
// The function isn't required here, but its closer to the others.
func run() {
    print("hello world!!)
}

run()
```

Figure 4.6: A very simple hello world program in Swift.

Swift as language is very similar to Kotlin, and offered many of the same tradeoffs which it offered but had one significant difference from Kotlin, it is a statically compiled language which produces a single binary and requires not external dependencies, such as a runtime. But conversely the binary is unique to each operating system and at the time of initial consideration had no cross platform support.

Despite the authors relative familiarity with Swift and the several extra language feature which Swift possessed, Kotlin was the preferred language to it. The author has previously utilised Swift any several other small-medium

```swift
guard sections.count == 1 else {
    throw AssemblerError.LineParseError(
        error: "Line has more than 1 segment, you can have at most ...'"
    )
}

guard case (.register, .label) = (condition, destination) else {
    throw AssemblerError.IncorrectTypeError(
        error: "Jumpf expects its arguments in the form ..."
    )
}
```

Figure 4.7: Swift Guard.

```kotlin
 val startIndex = (range["start index"] as? Number)?.toLong()
    ?: throw InvalidOption("start index", range)
```

Figure 4.8: Kotlin Equivalent to Guard.

sized projects to great effect which may have proven to be a valuable asset in creating the tool, the prototype too was in fact created in Swift, thus it would also have a short term advantage where appropriate design patterns for the system would already have been tried and discovered. Swift also offered several language constructs that may have offered a significant benefit to the error handling, most significant was Swift's guard-let-else and guard-else expressions (fig. 4.7), which allow for a cleaner and more flexible initialisation and chained conditional checking than the null-coalescing operator (":?") in Kotlin (fig. 4.8), which is also an option in Swift ("??").

The tooling for Swift though is far less impressive than that possessed by Kotlin, which can take advantage of much of the Java and JVM ecosystems, as well as the Integrated Development Environment(IDE) created by Jetbrain's, the creator of both IntelliJ and Kotlin. The lack of automated refactoring tools and a variety of other useful tools are a significant disadvantage for Swift in relation to Kotlin, or even Java. The then unreleased and currently experimental cross platform support for Swift was also a problem. Hence Swift was discarded as a potential language for the generic assembler.

4.2.5 Haskell

```haskell
module Main where

main = putStrLn "Hello, World!"
```

Figure 4.9: A very simple hello world program in Haskell.

The final language which was seriously considered for this tool was Haskell but that was quickly discounted for two very important reasons. The first being that the author lacked any prior experience in the language (formal, or self taught) and the University was not offering any classes to provide any experience. The second being that the pool of user who know Haskell is significantly smaller than the users who know Java, Kotlin, or any other JVM compatible language, which makes it less accessible. Although it could be argued that users who are likely to use this tool are also more likely to have experience with purely functional languages such as Haskell, it was not a significant enough option to counter those concerns.

4.2.6 Hybrid Solutions

```java
/**
* Java calculator class that contains two simple methods
*/
public class Calculator {

    public Calculator(){

    }

    public double calculateTip(double cost, double tipPercentage){
        return cost * tipPercentage;
    }

    public double calculateTax(double cost, double taxPercentage){
        return cost * taxPercentage;
    }

}
```

Figure 4.10: The Java component of the a Simple Java-Python hybrid application [27].

```python
import Calculator
from java.lang import Math

class JythonCalc(Calculator):
    def __init__(self):
        pass

    def calculateTotal(self, cost, tip, tax):
        return cost + self.calculateTip(tip) + self.calculateTax(tax)

if __name__ == "__main__":
    calc = JythonCalc()
    cost = 23.75
    tip = .15
    tax = .07
    print "Starting Cost: ", cost
    print "Tip Percentage: ", tip
    print "Tax Percentage: ", tax
    print Math.round(calc.calculateTotal(cost, tip, tax))
```

Figure 4.11: The Python component of the simple Java-Python hybrid application[27].

In addition to these (primarily) homogeneous systems an alternative was considered, the use of multiple languages for the for the implementation of the assembler. The primary contenders for solution was a hybrid system which mixed Kotlin and Java in the core implementation of the assembler and utilised a python implementation for the JVM as the configuration language for the system.

The hybrid approach offered the most benefits for the assembler and had their been an actively supported implementation of Python 3 for the JVM this would likely have been the preferred approach for the assembler.

The benefits it offers are numerous, for example building an extension method for the assembler becomes a trivial exercise, where the use can build extensions in python and embed them into the configuration that they have built, without any of the trouble involved in a DSL/library style approach or a plugin approach. It also removes any concerns about de-serialisation from the assembler, as it can directly interact with the data-structures created by the python configuration program and rely on Python's type system and internal mechanisms to handle validation of the input; which would significantly improved the implementation of the system. It would also have made many of the features of the system redundant, for example the instruction-format block (an abstraction for de-duplicating similar instruction binary format descriptions) are entirely redundant, the use may create a single function which creates an instance of the instruction-format, filling in the values automatically, without any of the rigidity that the current mechanism entails. The rigidity being a consequence of limitations of the configuration parser.

The unfortunate reality though was that there was no suitable and actively developed version of Python 3 for the JVM. Python 3 was the proffered target for this project for two reasons the first of which was the desire for a future proof choice, it was not desirable to switch implementation languages when Python 2 reaches its end-of-life status (no official support nor bug fixes) and Python 3 had many new features which would have been useful for this toolkit, the most important of which is its type hinting support. Thankfully, due to the tools support for usage as a library in the future it would be fairly simple to implement a bridge between Kotlin/Java and Python. Unfortunately there was no evidence of supported variant of Python 3 JVM being released in a practical time-frame for this project, Jython 3 (an implementation of Python for the JVM) has been under development for several years but was not released at the time of this report being written. ZipPy[46] is a fast and modern implementation of Python 3 (using the Truffle framework[45]) for the JVM, it would be a viable option for a future extension to the project but as of yet is incomplete and was released midway though the project implementation, too late to be integrated into the design.

Had there been an implementation of Python which met the stability and version requirements this would likely have been the approach which was taken, but as they were not met by an existing system it was deemed infeasible to adopt this approach.

4.2.7 Reflection

This sections reflects on the final choice of languages for the systems and problems which were experience with the languages, as well as what lessons were learned from this decision.

Reflection on Kotlin

When considered holistically Kotlin has been an excellent choice of language for the system. It's standard library has made many tasks that would otherwise be be unnecessarily complex a simple utility method, vastly simplifying the code base of the assembler. Several of its language constructs and its general approach to various problems have made many situations much simpler and much more brief than would be in the equivalent Java code, which was especially important in the configuration parsing phase, which requires some rather complex flow control logic, primary to provide more detailed error messages. Generally then it can be stated that Kotlin was the correct choice for this system, but that is not to say it was without its issues.

An unfortunate limitation which Kotlin has inherited from its Java roots is its lack of amenability to "interface oriented programming" which is an approach to object oriented programming[42] which eschews class inheritance in favour of interface conformance and interface inheritance. Objects do no rely on class inheritance to share behaviour for a method but instead implement interface on classes to extend their behaviour. Although Kotlin and Java can both conform to interface oriented programming techniques (and in fact the system utilises them to a degree) the inability of those languages to implement and interface without sub-classing it limit its power, as it may only be used on classes which are directly under your control. Thus, although Kotlin could utilise this technique it is less capable of expressing its full power than a language such as Swift, which allow class/struct/etc extensions to implement arbitrary interfaces.

The most glaring of the issues posed by Kotlin was that when the project began it was still in a later stage beta phase. Which for this project meant that there were still breaking changes being made to the syntax and semantics of the language, as well as additions, removals, changes and moving around of methods in the standard library. None of these issues was excessively time consuming, but during the earliest phases of the project (September-December), before the language hit pre-release, there was typically a significant change every second week, requiring that time be spent resolving compiler issues rather, acting as a distraction from the actual project.

What was in fact more time consuming than implementing the changes was the process of discovering the new best practices as the language evolved. For example, the standard library repeatedly changed what which of the several approaches was the preferred manner to instantiate and utilise regular expressions. Another example was when the language altered the syntax for existential types, the initial iterations of this were breaking changes which the system required the code be changed before it would compile correctly and later making optional the type annotation when retrieving these objects. This later change was not significantly difficult to implement and in fact had a net positive result, but it was rather time consuming to fix as the configuration parsing code made continual use of this feature and it permeated a large portion of the code base. It was not a required change, but it made the code far more pleasant to read thus was implemented gradually and illustrates an alternative to the program wide roll-out that other changes necessitated.

The other side effect of using a still experimental language was that there a limited number of bugs, which were later fixed. On the whole, the author did not encounter any significant number of bugs when developing the system, either in the standard libraries or in the compiler its self and the ones that were encountered were trivially simple and fixed rapidly. With only a single exception, during the final stages of the testing process (in one of the release candidate builds) Jetbrains accidentally introduced a bug where using the post increment operator on the backing field in an accessors would trigger a panic in the JVM. This issue was in fact due to the compiler generating invalid byte code and was tracked down by the author, in parallel with another user, and then reported to Jetbrains for resolution. The reason this issue was tracked down so quickly (in the under a day, by a user with no knowledge of the internals of the compiler) was because of Kotlin's JVM roots. The prediction that its access to standard JVM tooling would be useful rang true and the error was eventually (after tracking down which class was causing the issue) tracked down using a JVM bytecode decompiler and then reported. The temporary workaround for the issue was a trivial issue when the cause was discovered.

The final issue of consideration was the consequence of using a language with which the author had no prior experience. On the whole the learning process for the language was not extremely complex, it had excellent documentation, useful tool assistance and is semantically similar enough to Java to provide a good starting point. In fact, one of the most useful techniques for learning the language was to write the initial version code for a class in Java and use the automatic translation tool to convert it into Kotlin as to understand how they map onto one another, this technique was rather helpful early on but was used less so as the author gained more experience with the language. The only significant hurdle was learning the new potential failure modes for the language where the most common of the issues was due to Kotlin's type inference scheme, where it became possible to accidentally introduce type unintentionally in a long chain of calls or closures, the most common cause of this was forgetting to throw an exception after instantiating it and this prorogation through the call chain and eventually becoming a non-trivial issue to track down. The main cause of this was inexperience and eventually these became rarer and rarer as the author developed the correct mental model of the system.

Overall, despite these issues, Kotlin was an excellent choice in languages and it was a far better choice than Java, the second strongest competitor.

4.3 Initial Development

The first phase of the development process was to implement an prototype assembler. This was implemented in swift and was performed with the intent of developing an understanding of the mechanics of constructing an assembler, as this is the first time the author has constructed one and felt that it would be an instructive experience. Which it ultimately was, its design guided the architecture and general design trends of the assembler its self and although no code from that initial prototype was utilised in the final build, its influence is very visible. For example in the error handling model and the general naming schemes and the general break down of how the configuration is structured, the references, the instructions and the subtypes of each.

The second phase of the development process was far more agile in nature and is interleaved with the evaluation process of the tool. The general process was a simple feedback loop, where the implementation was evaluated respective to the current target architecture, its weakness noted and an estimate was made as to what features were required, which were then implemented. Then an attempt was made to implement the architecture using the existing tools. If it wasn't possible to implement, then the required features were implemented and finally, it was re-evaluated.

This process had several positive effects on the development of the system. the most important of which was how it effected the direction of development. Prior experience in creating tools such as these has taught that implementing features in an undirected manner has a tendency towards sub-par and incomplete systems being developed, where there is a mismatch between the actual required feature set and the implemented feature set. This approach of frequent and constant iteration had the effect of guiding the development priorities away from the more intriguing and fascinating features and towards the high impact, low interest features which were far more helpful for the user.

Those tools were not the most obvious and eye catching feature set to implement but were arguably the most important features which needed to be implemented to develop a working tool. For example it guided development away from the more interesting features, such as macros and segments, towards the more critical tasks for the use case of this tool, such as the ability manually construct the binary structure of an instruction from the reference types and binary values. In that example, both features were of high interest but had it been a purely arbitrary choice then Macros would likely have been implemented first despite the significantly higher important of implementing the latter feature. The requirement to use it for the MIPS architecture is what push the author to prioritising that feature over the development of the more interesting feature, macros. Which ultimately was the correct decision and permitted the implementation to proceed faster than had macros been implemented first.

This was a blocking issue for MIPS due to the instruction formats which it used, these did not have a 1:1 correspondence to its arguments, thus it required a more comprehensive feature set to implement. Macros conversely though are used by many of the instructions in MIPS to map the instructions from their pseudo-instruction form to the full expanded form but it was possible to construct some reasonable subset of MIPS I which did not require macros (the non-pseudo instructions[10]). Thus macros should have and did take a lower priority than the more complex instruction format construction mechanism.

This was only a single example of this process in action (a relatively trivial one at that) but it demonstrates the value of this approach in developing this tool.

4.3.1 Overall Architecture

The overall architecture of this tool is influenced by the original prototype in many different ways. Most obviously in the naming scheme for the various reference types but more importantly in the structure of the reference subtypes and the line and error handling mechanisms.

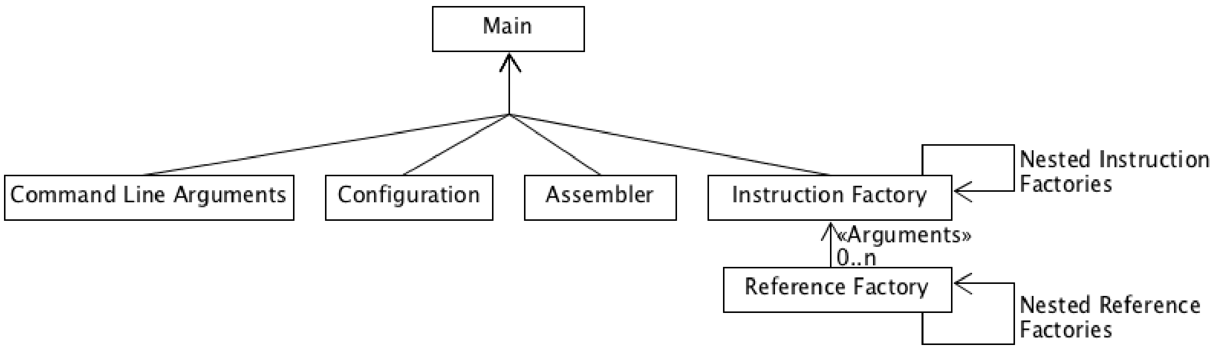

Figure 4.12: A UML diagram of the Main Class of the Tool.

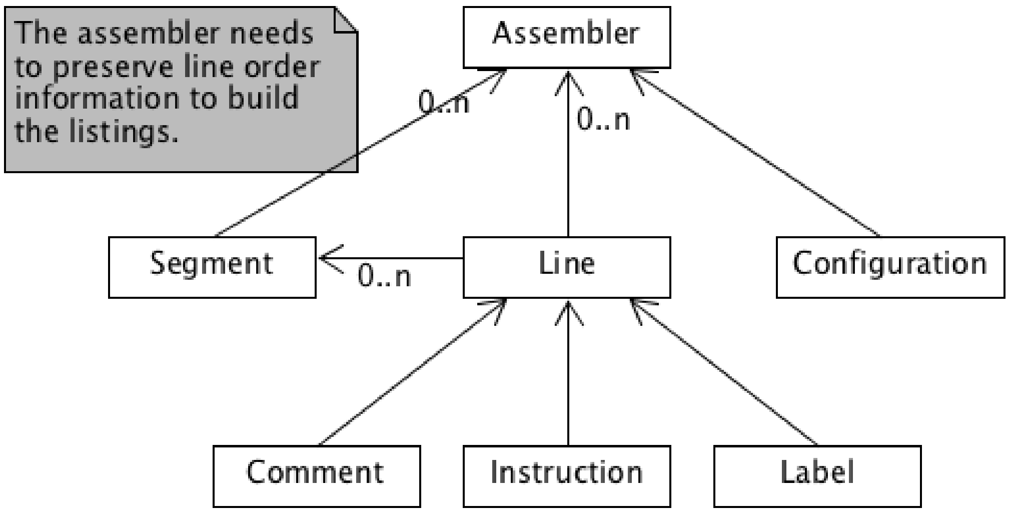
The assembler needs to preserve line order information to build the listings.
Assembler
0..n
0..n
Segment
0..n
Line
Configuration
Comment
Instruction
Label

Figure 4.13: A UML diagram of the Assembler.

The overarching structure of the program is best considered hierarchically; the root of the assembler is the main file, which is responsible for 4 tasks: first it is the command-line argument parsing, secondly it delegates the configuration parsing to the various methods, instantiating the compiler from the assembler and then delegating the assembling process to the assembler and handling IO for it. Each of the stages of that process: command-line argument parsing, configuration parsing, assembly file parsing & compilation and the IO are handled independently of one another in an attempt to decouple each of the stages from the other, in this architecture they each operate independently and without knowledge of one another, which allows for the user to independently replace each component of the tool (for example the configuration parser may be replace with the DSL) or the IO routines and command line interface may be replaced with a web interface or a plugin for a system such as IntelliJ and other such possibilities.

References Figure 4.14 illustrates the overall structure of the references package. It is composed of 2 primary hierarchies of classes, the factory classes, which are responsible for instantiating and parsing specific instances of

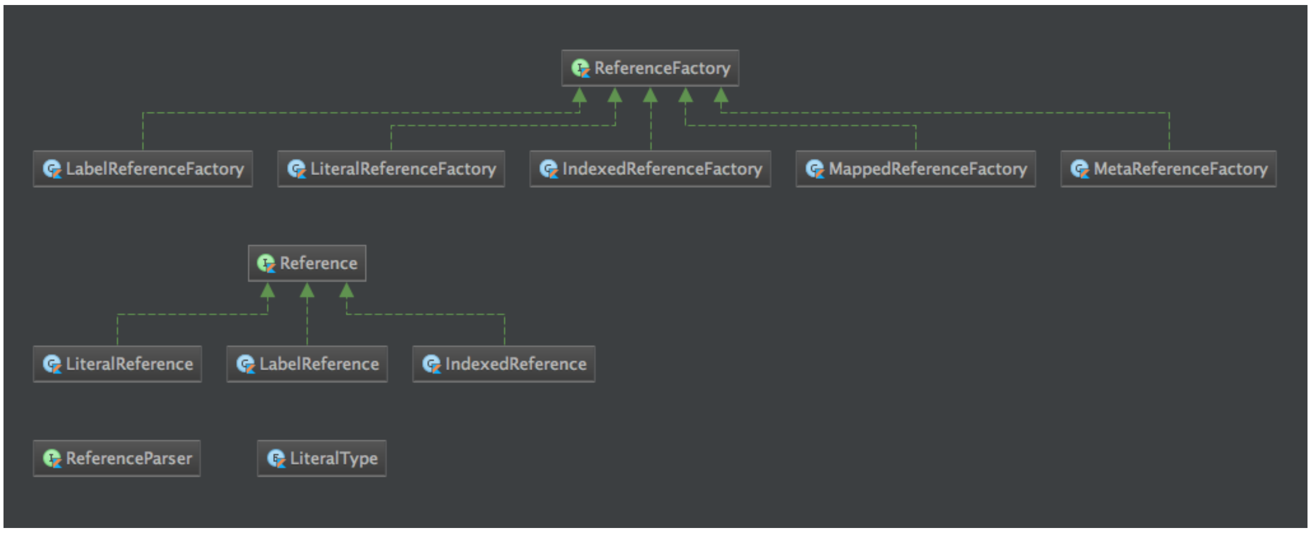

Figure 4.14: A UML diagram of the References package.

a reference using the parsed configuration to create a specific reference from the second tree, the reference tree. This tree contains classes which actually encapsulate the the information for that specific reference, for example for a binary literal it would contain the type of the literal, the raw string form of the literal, its size and it would be responsible for actually converting the literal into a binary array of the correct size.

The basic patterns utilised by this package are the factory pattern and the chain of responsibility pattern.

The factory pattern is the mechanism by which the configuration is used to parse and instantiate the actual reference objects. The configuration parser or the DSL instantiates the factory object using the configuration, passing in the parameters which permit the factory to instantiate the instructions and parse the assembler file. The factory implements three methods and properties, a type property which the system uses to get the name of the type for generating the help string and for generating debug messages for the user. It also implements a method which accepts a string (which the system has determined should be a reference and may potentially match the type of this reference, this is the ”checkIsMatch” method which is used to determine whether or not this is the relevant reference. If it is then the ”getInstanceIfIsMatch” method is called on the factory and this instantiates the concrete reference.

There are 5 different types of reference factories in use in the system, LiteralReferences, IndexedReferences, LabelReferences, MappedReferences and MetaReferenceFactory. These different types of references exist due to the differing requirements for extracting these factories and the different data which each of the references needs in order to be instantiated, for example the label factory requires the label table to be passed in, where as no other requires this.

Literal references are the simplest type of reference available in the system, they are used to represent literal types such as integers (1, 2, 3, etc), hexadecimal (0x1, 0x2, 0xB, etc) and others.

The second are MappedReferences, which are designed for a similar use case as the literal references, but are designed for use with small but arbitrary literal values, for example register references, which may be literally substituted by an integer value. These mapped values in-fact are so similar to literal references that they both instantiate the same LiteralReferences class and differ only in the implementation of their factory.

Indexed references were a more complicated type of reference, they were designed to encapsulate nested reference types such as occurs in indexed memory accesses such as in this sample of a Sigma16 assembly file fig. 4.15 which on line 2 illustrates an example of an indexed reference, where the compound type first uses the first parameter (’x’) as a memory location and the value on the second parameter (’R3’) as the offset from that

```
; Taken from the ArraySum.asm provided for Compute Architecture
         load  R5,x[R3]          ; R5 = x[i]

; Data area
n        data   6
sum      data   0
x        data  -1
         data   0
         data   1
         data   2
         data   3
         data   4
```

Figure 4.15: A fragment of a Sigma16 Assembly Language Program (from the Computer Architecture Course).

address. This type was the most complicated to implement due to the nesting requirements it may face. In the prototype the types of the nested values were in fact hard coded into the regular expression of the compound type (and to a certain degree there is still some interdependence on one another's regular expressions) and this was not a scalable solution, thus this type had to be fundamentally rethought.

The first stage was to decouple the type checking in this type from the regular expression which identified that the reference matched the structure of the indexed reference. So first the checker identified that the structure of the reference could be a match for this indexed expression, it then extracted the candidate nested references and then it recursively (in the case of another nested reference) checked the type of those nested references, returning true if they were of the expected type, false otherwise.

The design of these was driven by the specific issues experienced during the prototyping phase, where the direct instantiation of the various types led to a very heavy interdependency between the various reference and instruction types and the actual instantiation code. This led to the noted difficulties relating to altering or expanding the system and these concerns, as well as the desire for an extensible system, led to the usage of the factory design pattern.

On the whole the design pattern was an effective solution to the architectural issues encountered in the prototype and it facilitated a configurable and extensible approach in the assembler in an understandable and effective manner, such as in the case mapped references and literal references sharing a single reference type implementation, despite their different factory types.

The other design pattern which was used in the system was the chain of responsibility pattern, which offers each item in a collection of event handlers the opportunity to handle the event (perhaps recursively passing it onto its children or instead directly). This pattern is used in two separate places for references, the first being the mechanism by which the reference hierarchy is searched for the correct matching reference type and the second being the more program wide method of converting the assembly language program into a byte stream.

The reference factory hierarchy is composed of 2 primary types of nodes, the leaf nodes, such as LiteralReferenceFactory, IndexedReferenceFactory, LabelReferenceFactory and MappedReferenceFactory, these map directly to a concrete reference type. Alternatively there are the branch nodes of the hierarchy which are represented by the MetaReferenceFactory class, which are a container reference type for a collection of reference factories and may be used to reference multiple reference types in a single node. This is useful in situations such as allowing literals of multiple types, such as allowing integer and hexadecimal literals to be used for an argument. These branch nodes recursively offer each of their child nodes the opportunity to instantiate the reference. Only the first reference encountered is used here and the priority is based on order of declaration.

This decision unfortunately precluded the usage of an ambiguity warning (in the case that 2 or more reference types match on a single candidate reference) as the decision was made to optimise for the typical use case, where the user has intentionally introduced ambiguous definitions in an effort to account for an existing assembly language syntax which utilises the same mechanism. It may have been possible to provide an optional warning for this situation but it was not implemented in time for the submission, despite the potential utility of this feature in detecting potential accidental selection of the wrong reference type. There are also potential issues relating to false positive warnings, which unfortunately discount this feature as a possibility.

This pattern was such a success that it has in fact percolated throughout the entire system, it is also used to convert the assembly language program into a literal. Each object which can be represented in a literal form (either directly, or by through its children) implements the ”RawLiteralConvertible” interface which indicates that it will convert its self into a binary array, or delegate the task to its children. This abstraction had the effect of creating a single consistent method of accessing the compiled binary form of an array and has made the program mode consistent and easier to understand as well as facilitating the usage of the SizedBinaryArray class.

The final interface which these classes utilise is the ReferenceParser interface, which is used to create a common function on each objects companion object[26](static methods, properties, etc for a class) for the purposes of parsing the configuration sections for the references.

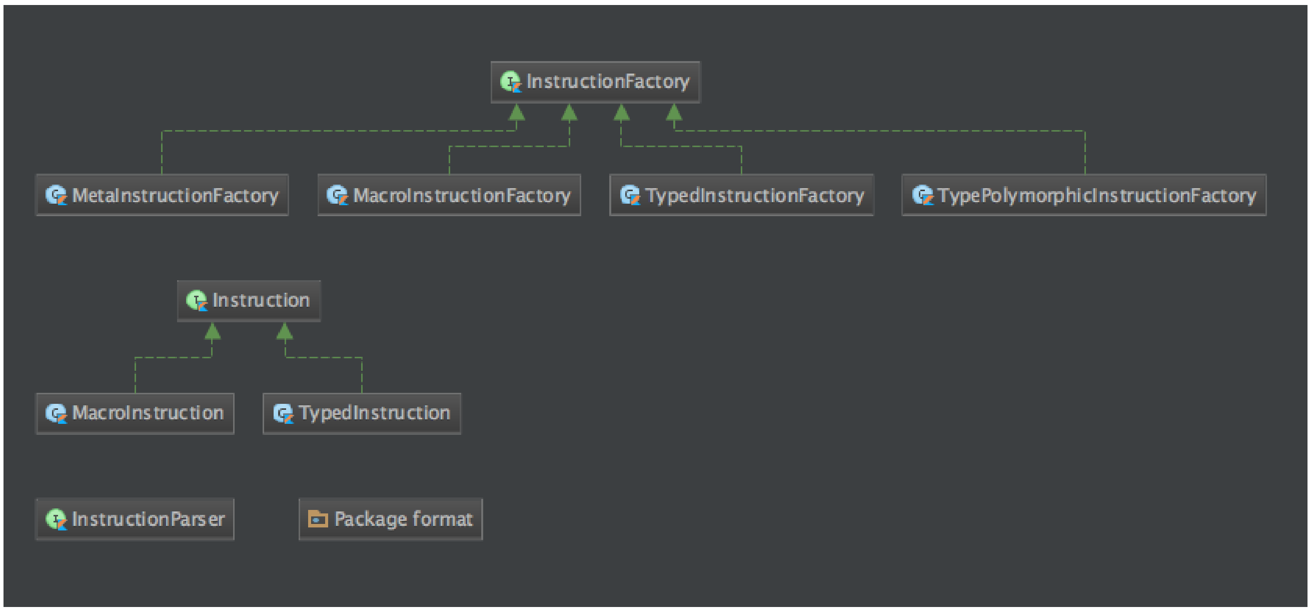

Figure 4.16: A UML diagram of the Instructions package.

Instructions The instructions package (as displayed in fig. 4.16) has a very similar architecture to that used by the references package for very similar reasons.

The most clear difference between the instructions package and the references package is in the actual classes which are implemented, they follow the same design patterns, naming schemes and have a similar purpose but they implement different behaviour and interfaces. The four main groups of classes are typed instruction factory, type overloaded instruction factory, the macro instruction factory and the meta instruction factory.

The typed instruction factory group of classes (”TypedInstructionFactory” and ”TypedInstruction”) are the basic building block of the instruction classes, unlike with references, instructions only have a single leaf node type and instead have multiple branch/container nodes. This is due to each instruction requiring the same infrastructure to parse and assemble but the way these are combined tend to differ. The basic mode of operation for this class is very similar to that of the reference classes, it has two methods for checking if the instruction is potential candidate for parsing this instruction and the second checks if the types of the arguments is correct and then there is the method which actually returns a concrete instruction object from the assembly.

The logic for type resolution is primarily deferred to the each of references (using the chain of command pattern) which greatly simplifies the the instruction implementations and demonstrates the value of that pattern to this system. The sole complication being how instruction formats are handled, specifically because they require support for aliases (in the case of the name internal to the instruction format and the name of the argument do not match) which alias the path segment for that level onto one-another, allowing for the reference path resolution process to refer to the instruction by either name.

The alternative to implementing this aliasing system (and the complications it creates) would have been to force the user to ensure the names for the argument match the name for segment in the instruction format. This was very much a trade off for additional complexity in design, implementation and (optionally usage) in return for poorer debugging and error messages. The system currently utilises the argument names in conjunction with types to help inform the user of errors in the assembly language and not implementing this feature would have been a net loss to usability. The decision was made then to implement this feature despite extra complexity, due to the second use case of this system. Students using the system for the first time would benefit from the additional information, but also it would also help the first class of users (researchers) by acting as a form of self documentation, as it would be far clearer as to the purpose of an argument if it is named descriptively. Hence the decision was made to implement this feature despite the trade-offs required.

The MetaInstruction group of classes is relatively straightforwards and exists purely for usage as the root node for instructions, there is no user facing interface for this class.

The TypeOverloadedInstruction type exists for similar reasons, it is a container class which selects a specific instruction based purely on the type signature of the class and if the identifier of the instruction matches the identifier of the TypeOverloadedInstructionFactory class, this exists for the case when several instructions of the same name overload a single a name and are selected for based argument type. This was an important feature to implement due to the use of this feature in many assembly languages but as mentioned previously, it can cause false positive values for certain classes of error detection, thus precluding their use. Despite this problem, the criticality of this feature required that it be implemented.

The most interesting of these instruction classes is the ”MacroInstructionFactory” which the most complicated of types and necessitated a great deal of additional machinery in the assembler to support. The current implementation has several limitation feature set, most notably that it only support macros which are hard-coded into the assembler description. Implementing support for macros in the body of the assembly file would require a fundamental rework of how the lexing stage is performed (it may have been possible to utilise a simple regular expression to capture macros in most languages, but this avenue was not explored).

Currently the macros are also designed to be fully hygienic[29] and hence labels defined within macros cannot be referenced, this was a temporary trade off made for the implementation of this project. The use of hygienic macros is an attempt to avoid naming collisions between the macro body and the user code, as well as collisions between instances of the macro its self. The unfortunate side effect is that the isolation is complete and there is no mechanism in the current implementation to allow the label definitions to escape to the outer context. This is not a fundamental limitation of the system which currently implements the macros as an entirely separate instance of the assembler which can access its own nested label table (as well as recursively resolve the label from the parent assembler’s scope) but it is not implemented due to the lack of other related features.

Ultimately, this feature although powerful in its current form is hamstrung by the overall approach to development which this project has adopted, its nature of implementing features as and when they are needed has meant certain arbitrary limitations exist in the system, but the sheer nature of the system implies that only only useful features are prioritised and as this feature was not prioritised, it was not useful. Hence it could be argued this not implementing it was in fact preferable to not implementing something more critical.

Overall the basic architectural features of this system are common to each of the major components of the system the two most common of which are the factory pattern and the chain of responsibility pattern both of which have had a positive effect on the design of the system relative to the design prior to both their introduction and the prototype system. The system is designed with the intent that components may be added or removed independently of one another and the system is fairly capable in that regard. This was an important characteristic to support the researcher use case, where they may be required to implement features beyond the scope of the existing design to implement their required assembler.

Error Handling

One of the important concerns for this system was how it would handle errors, both those in the configuration and those in the actual assembly language program. The reason that this topic was a significant concern was due to the target use case of this tool. Both of the classes of user (researcher and student) would have benefited strongly from clear and precise error messages.

The researchers benefited as clear error messages ensure that they can pursue their research at a high speed as it permits them to rapidly iterate over their design for the system and fix the errors in short order of their occurring, had the system not given clear error messages to the user they would struggle with finding the cause of errors and waste time on an unnecessary issue, slowing down the iteration process.

The students benefited from this in a more obvious manner, as new users they are far more likely to make errors while creating their configurations and the assembly language file and far less likely to understand cryptic and domain specific errors and they would benefit very strongly from more clear and specific error messages than any other user.

The error handling mechanism for the assembler is both fairly simple and sufficiently powerful. The basic mechanic which it utilises is Kotlin's exception mechanism[23]. It is identical to the one utilised by Java, with the exception of the lack of support for checked exceptions. The system possess 4 separate classes of errors, each of which are designed for use in separate contexts. They are customised as need be for the situation in which they will be use, for example the ”AssemblerError” class hierarchy extends the AssemblerError class (fig. 4.17) and the differentiating factor for that error is that is possess a line number property, which allows the system to display more detailed debugging diagnostics in the case of an error.

The line error mechanism is implemented through annotation of the error class midway through the exception throw. The assembler catches the error at its point of occurrence and annotates the error before re-throwing the error and allowing the system to handle the exception as needed. This allows the actual exception handling mechanism to be decoupled from the implementation of the assembler, thus permitting the system to be used in a more flexible manner, for example referring back to the earlier example of implementing the assembler as a website, the error messages should not have been output to standard out as is the case of the application in command-line usage but instead recoded, serialised and sent to the web client which should handle them as need be, perhaps highlighting the erroneous line; it would also be useful in the context of a DSL or if the tool were to be integrated into a plugin for an IDE (such as IntelliJ[24]).

Figure 4.17 also illustrates the many different forms of error which the assembler support, each of which is documented and is intended for use a different context. The decision to use different errors in different contexts, instead of a single error class with unique messages was an effort to allow two things: future alterations to these exceptions should be possible without altering every other instance of that exception, for example if there was a spelling mistake in the error message altering it in the message should alter it elsewhere; secondly it permits each error to be handled uniquely if need be, for example it has extra parameters in its constructor so that it can automatically generate the extremely commonly used error string. The unique names also have effect of adding redundant information to indicate what exactly is wrong with the users input. Which makes it harder for

```kotlin
open class AssemblerError(message: String) : Exception(message) {
    public var line: Line? = null

    override fun toString(): String {
        return "An error occurred while parsing line: \n\t" +
            "${line?.lineNumber} '${line?.line}' \n\n" +
            "Producing the error:\n\t" +
            "${super.message}\n\n" +
            super.toString() +
            "\n"
    }
}

class InstructionParseError(message: String) : AssemblerError(message)
class DataSourceParseError(message: String) : AssemblerError(message)
class LineParseError(message: String) : AssemblerError(message)
class UndeclaredLabelError(message: String) : AssemblerError(message)
class IncorrectTypeError(message: String) : AssemblerError(message)
class AbstractInstructionInstantiationError(message: String) :
 ↪  AssemblerError(message)
class PathResolutionError(path: String) : AssemblerError("Path '$path' is
 ↪  referenced but cannot be resolved.")
class IncorrectTypeSignatureError(message: String) :
 ↪  AssemblerError(message)
```

Figure 4.17: The Assembler Error Class and related errors

unobservant users to miss what the error is, or if there is a failure in the error handling mechanism to still provide some information, regardless of any other failures.

There are also specific errors for command line parsing, configuration parsing and in the case there is an internal error relating to any potential bugs in the assembler or a misconfiguration in the DSL.

Figure 4.18 is an example of an error which the assembler can produce. This particular example is due to an error by the user when they were writing the assembly language file. They had mistakenly attempted to refer to a register which does not exist, in this case ”r99”. This error message illustrates several of features of the assembler, for example it attempts to aid the user in ascertaining the exact cause of or an error by highlighting the specific line on which the error occurred but it is not granular enough to highlight exactly which argument is at fault. This limitation is unfortunately a consequence of the systems architecture, specifically how it resolves the overloaded instructions. The system only knows that the instruction does not match any of the known instructions and that is due to its arguments but it can not discern which argument it at fault. There was no easy correction to this error without removing support for overloaded instructions. which was not a feasible approach as those were a high priority item.

Another weakness of the system is that there is no analogue of the mechanism which reports line numbers for the configuration parser. This limitation is due to the lack of support for this in the underling library used to parse the YAML files, as that does not report errors at that granular a level. The solution to this would be to either replace the system with a different or custom implementation or alter the library to add this information to its exceptions. As neither was feasible in the time constrains of the project instead an effort was made to provide as much information as possible to the user when an error does occur.

Overall the error handling mechanism is fairly useful, it provides sufficient information to the user as to permit

```
Exception in thread "main" An error occurred while parsing line:
        4 '       load  R99,n[R0]'

Producing the error:
        load with arguments 'R99', 'n[R0]' could not be parsed correctly,
         ↪ it should be in the form 'load <destination:register>
         ↪ <address:memory>'

com.nishadmathur.errors.InstructionParseError: load with arguments 'R99',
 ↪ 'n[R0]' could not be parsed correctly, it should be in the form 'load
 ↪ <destination:register> <address:memory>'

        at com.nishadmathur.instructions.TypedInstructionFactory
    .getInstanceIfIsMatch(TypedInstructionFactory.kt:70)
        at com.nishadmathur.instructions.InstructionFactory

...
```

Figure 4.18: An example of an Error.

them locate and correct the issue despite the weakness highlighted above. Improvements could be made in the future.

YAML Configuration

The YAML configuration format was intended as the primary mode of usage for the application. It utilises standard YAML syntax and makes no use of explicit type declarations in the code (to permit automatic de-serialisation of the objects into Java objects)[43].

```
instructions:
  - !com.nishadmathur.assembler.MetaInstruction
    name: .data
    instructions:
        - !com.nishadmathur.assembler.SimpleTypesInstruction
          name: .data1
          byteSequence: 0x0
          size: 0
          arguments:
              data0: literal
        - !com.nishadmathur.assembler.SimpleTypesInstruction
          name: .data
          byteSequence: 0x0
          size: 0
          arguments:
              data0: literal
              data1: literal
```

Figure 4.19: An example of YAML Beans using type annotations.

The decision was made not to pursue this approach for a number of reasons; the most important of which

```yaml
instructions:
  - name: .data
    kind: meta
    instructions:
      - name: .data1
        byte sequence: 0x0
        size: 0
        arguments:
          data0: literal
      - name: .data2
        byte sequence: 0x0
        size: 0
        arguments:
          data0: literal
          data1: literal
```

Figure 4.20: An example of YAML without explicit type annotations.

are ease of use related, these type annotations are purely redundant information to make the compiler simpler without any benefit to the end user, which was judged to be an inappropriate trade off. Figure 4.19 demonstrates an example of code using explicit type annotations to allow automatic denationalisation. This example has several downsides, firstly it requires explicit knowledge of the assembler structure and thus explicitly couples the serialisation format to the internal structure of the assembler. Secondly, the object paths tend to be lengthy and thus clutter up the code with long class paths which runs counter to the purpose of selecting YAML, simplicity, ease of use and being easy to comprehend.

This approach was also lacking in other areas, such as the limitations imposed upon the design of the system. The most problematic of these being the impracticality of supporting multiple object schemas for a single class. An example where this limitation would have come into play is in instruction formats, as there are two separate declaration syntaxes available for different use cases, one is the ”simple case” which attempts to automatically infer the binary format of an instruction using the order of arguments in the instruction. The other format requires that the user explicitly state the binary structure of the instruction. The utility of the ”simple case” is potentially debatable but it acts an illustrative example of a situation in which this limitation can degrade user experience.

The final issue with this approach is impedance mismatch between the JavaBeans[17] approach which YAML Beans uses and the semantics of Kotlin which favours safety and brevity. The limitations are specifically due to Kotlin’s null safety, which requires that properties be explicitly initialised and requires that either the checks be disabled or all properties marked optional, neither of which is a desired approach as they undermine many of the safety and brevity improvements which Kotlin provided to the assembler.

Thus the tool uses an explicit parsing approach for parsing the configuration files utilised by the tool. This had the cost of significantly more code inside the tool but had the side effect of simplifying the configuration format and avoid issues relating to utilising JavaBeans within Kotlin, which is was a worthwhile trade-off. As demonstrated by fig. 4.20 which shows a more complex hierarchy which avoids the use of explicit type annotations and is more easly understood than the equivalent in fig. 4.19, the mode advanced form compares less favourably than even this trivial example.

The usage of manual parsing also offers the opportunity for more specific error messages both in the current implementation and in future extensions to the project.

Thus, despite the increased implementation complexity of this method, the manual configuration parsing approach was adopted.

4.4 Architecture Implementation Iterations

The later stages of the tools implementation were performed in an iterative manner. It was performed this way in an effort to guide the development of the tool and to allow it to focus most heavily on the features which are blocking its use over those more interesting but less critical features which may instead have been selected.

To perform this stage a selection of architectures were chosen for to be implemented using the tool. These architectures were chosen as to represent a broad spectrum of different architectures and assembly languages. The final candidate list for the architectures chosen was (in order of implementation): Sigma16, MIPS R2000, AARCH64 and Intel 8086.

4.4.1 General Approach

The general approach of this section of the implementation involved first extending the implementation of the system with the features which will likely be required. The implementation then begins by first researching the architecture and developing an initial understanding of it and an initial attempt is made to implement the architecture. If this proves infeasible due to the scope of the project a reasonable subset of the assembly language is chosen to be implemented and then a new attempt is made, implementing features until is is capable of expressing that subset.

The purpose of this project is not to create a perfect recreation of the assembly languages them selves but as a proof of concept to ensure that the assembler is capable of expressing those languages sufficiently as to allow a usable end result. This tool did not and could not implement every feature of every assembler but it can implement the core functionality. There may also have been errors in some of the language description files but as the purpose is to prove that the approach is viable and not to produce a working production assembler that was deemed to be a non-issue for this use case. Regardless of this, efforts were made to correct any detected errors.

4.4.2 Iterations

Sigma16

The first architecture which was implemented for this process was Sigma16, it was an architecture developed specifically for research and teaching purposes and has no commercialised version nor external distribution. The assembly language was designed with simplicity and ease in mind which made it an appropriate starting point for this process, as it was optimised specifically for the purposes which this tool is designed for.

The Sigma16 assembly language has a very simple and clean assembly language and makes no use of more complicated features of the assembler, such as segments.

There were 3 forms of instructions for this instruction set, RRR, RX and XX. Unfortunately there was no information provided for the XX format thus it was not implemented. The RRR instruction set is the simplest form of instruction, it is named for the data type of its arguments, ”Register, Register, Register” where the first register is destination and the second and third are source for the arguments, it is a 16bit instruction. The RX format on the other hand is a 32 bit instruction where the first 4 bits are the opcode which specifies that it is an RX instruction and the next two sets of 4 bits are the registers for that instruction followed by 4 bits of opcode which identifies which RX instruction it is and lastly in the second word is a 16 value, typically the address of a label for example.

Implementing this architecture was a rather straightforward affair. The initial assembler only had the simple form of instruction formats implemented and this was sufficient to implement the entire RRR instruction set, but it

```

; Initialise

       lea   R1,1[R0]          ; R1 = constant 1
       load  R2,n[R0]          ; R2 = n
       lea   R3,0[R0]          ; R3 = i = 0
       lea   R4,0[R0]          ; R4 = max = x[0]

; Top of loop, determine whether to remain in loop

loop
       cmplt R5,R3,R2          ; R5 = (i<n)
       jumpf R5,done[R0]       ; if i>=n then goto done

; if x[i] > max

       load  R5,x[R3]          ; R5 = x[i]

```

Figure 4.21: An exert from a simple Sigma16 program (Source: Computer Architecture Materials).

was not capable of representing the RX instruction set. This was due to the second opcode which was required for the process to identify which RX instruction was being executed.

There were three potential solutions to this problem, the first was to replace the simple form of the format with the more advanced form of the instruction format, as discussed in section 4.5.3 which contrasted that approach with the second option, to add the an advanced approach for more complex cases and the final approach considered was to extend the simple case with more advanced functionality. The merits of the first two options have already been considered and the option to extend the functionality of the first case was studied but deemed to not offer sufficient flexibility due to the coupling between the order of arguments in the instruction and order of arguments in the output. In this case the specific problem was that the address and offset variable had to be broken up into two separate sections, the address going in the second word and the offset register going in bits 8-12 which would not be practical using an extension of the simple scheme.

Hence it was decided to implement an additional instruction format specifier method, which eventually become the advanced form. This specific approach permitted the use of 'paths' which allow the specific arguments and nested arguments (may be nested to an arbitrary degree) to be referenced.

This architecture was validated by compiling code provided as part of the computer architecture course and running that on the provided M1 implementation of the Sigma16 virtual machine and testing its execution as well as comparing its result to provided pre-compiled version of the program. There was also manual inspection performed where the program was compiled and than manually compared to a provided listings of the program. After several bugs were fixed they produced the same results.

Overall the assembler was easily able to implement this assembler and its related syntax and is easily capable of handling assembly languages of this level of complexity.

MIPS R2000

MIPS R2000[10] is an implementation of the MIPS I instruction set, which was a 32bit fully RISC instruction set with 3 forms of instruction. MIPS is a real world architecture which is is still a subset set of the MIPS MIPS32 and

MIPS64 architectures which are still in use today. MIPS I was the first architecture produced by MIPS Computer Systems. Its usage as a real world architecture makes it a suitable test case and introduces sufficient complexities to require a new architecture.

This iteration produced an implementation of the full architecture of the non-pseudo MIPS I instructions. The architecture required several new features to fully implement the architecture of the MIPS instruction set. As well as an implementation of one pseudo instruction.

The basic implementation of MIPS was performed entirely using the advanced instruction format as the basic format could not represent it. Overall its usage of the 3 distinct instruction forms highlighted the utility of a formal mechanism for sharing instruction formats between the various instructions to reduce amount of configuration required for each instruction.

Implementing support for this architecture required that several new features be implemented: pc relative offset, placement control and a primitive implementation of segments, which is strongly related to the placement control. The different forms of offset are implement through use of the delayed reference type detection mechanism, where the different addressing modes are implemented as different types of label. This simplifies their implementation as the instruction can just declare a label of a set type, for example it can opt to use a ”pc-relative label” (for example) or it can choose to utilise a global label, which ever is appropriate in the context. The architecture also required support for a very primitive implementation of segments which are use purely to control the placement of code and data in the memory of the architecture.

```yaml
 - name: blt
   kind: macro
   arguments:
     lhs: register
     rhs: register
     destination: literal
   template: |
     slt $1, $lhs, $rhs
     bne $1, $0, $destination
```

Figure 4.22: An exert from the MIPS I architecture definition file which presents the ”blt” instruction, a macro instruction.

One feature which was implemented as a result of this architecture was the macro system. Although the subset of the language which was chosen does not require it was deemed to be of sufficient importance to be implemented for use here. The system was built to support the pseudo instruction which are used in the MIPS assembly language and was tested by implementing an example of such an instruction, ”blt”. Known issues and limitations for macros are discussed further in section 4.6.

Overall the assembler appears to bee capable of implementing the entire subset of MIPS I which was evaluated and comparison of output with a third party assembler appears to confirm that statement.

ARMv8

ARMv8[3] is a very new RISC format built-upon several generations of previous architectures, which creates a fairly complex RISC architecture (an interesting contradiction of terms). The legacy issues create some interesting requirements which provided a useful stress test of the bit-manipulation and indexed reference type destructuring capabilities; which were fairly complex to implement and test. It faced the same requirements as MIPS and added the complications of legacy, which made it an appropriate test case.

A subset of this architecture has been implemented using the description found in the ARMv8 manual[3]. Implementing this architecture was extremely time consuming and finding an assembler which supports the implemented syntax has proven beyond my capabilities, thus this the output of this configuration was manually checked. The process is clearly open to errors of interpretation on the authors part so there may be incorrect assumptions made which cause incorrect output to be generated.

To implement support for this architecture several new features were implemented, most importantly the instruction format specifier which allows multiple instructions to share a single definition and also register ranges, which allow the user to declare the start point for a rage of registers, number to generate and the textual form and it will synthesise them according to the specification. These features would have been useful in the previous assemblers but did not prove to be vital thus implementation was deferred until this iteration.

Overall, this architecture appears to have been implemented correctly, but any fundamental misunderstandings on the authors part may have introduced errors into this. Regardless though the tool in theory should be able to represent this architecture but it does not allow for a short implementation.

Intel 8086

Intel 8086[22] differs form MIPS Sigma16 in the number of addressing modes which it supports. It has 17 different modes[16], each of which is very time consuming to implement.

Thus the decision was to attempt to implement portions of the architecture in an exploratory manner to discover the capabilities if the tool without implementing a fully funcitoning implementation of 8086. Overall it highlighted the flexibility and power of the reference system by allowing for support of each of the addressing modes which was tested (indexed, register, displacement, Based Indexed and Based Indexed Plus Displacement) each of which is fairly complex. Implementing these highlighted the weakness of the reference system which make it impossible to reference more than 2 nested types in an ”indexed type” this necessitated the use of multi-layered definitions which were unnecessarily complex. The inability to set default values on arguments had also complicated the implementation process of instructions, it required that each instruction be implemented once for each addressing mode, so approximately 5-17 times for 50-70 instructions, which is extremely time consuming and would be better suited for the DSL.

Implementing support for this required that the ability access a specific range of bits in a reference. It also required the addition of a segment local addressing mode.

Overall the tools is powerful enough to represent this language but it is not powerful enough to do so in a concise manner. This is a limitation of the system but also a demonstration of its flexibility. In the future allowing a more brief implementation should be investigated.

Rejected Architectures

A variety of other architectures were considered for the testing process as well, but ultimately the decision was made to narrow down the selection of different architectures to just 4. The following architectures were rejected for a variety of reasons but the complexity in implementing them is a recurring theme.

4.5 Trade-offs

There were many trade-offs which were considered and implemented in this system, this report focuses on the most interesting and important of those trade-offs.

4.5.1 Reference Type Detection/Checking

The type detection system for arguments is suitable for its purposes from a technical point of view and is the best choice when that was the only consideration but there was another tradeoff to be made in that area; upfront complexity versus difficult to debug edge cases.

The type detection system deferred the actual process of determining the type of a reference until as late in the assembling process as possible, during the type checking of the instruction. The alternative which was considered was to type the reference immediately (as was done in the prototype) and then check the if they matched the type of the instruction.

Although both approaches sound functionally identical and are in fact fairly similar there are ramifications of each approach, the most important of which was that the delayed type detection allowed for simpler regular expressions for each of the types, as only the relevant potential types are checked and thus similar types which never occur in the same meta instruction do not have to consider one another when designing the regular expression (reducing the risk of ambiguity). The risk posed by this approach was that it would encourage regular expressions to be created without consideration of how types would interact in those few cases where there is potential ambiguity, which could lead to very hard to debug issues where the wrong instruction is being selected due to insufficiently specific reference types regular expressions. This issue did in fact occur during the development process of this tool which brought this consideration to light.

In the end, it was decided that despite the potentially problematic edge case posed by this approach it was preferable to the overhead of writing regular expressions that could disambiguate the types, throwing away the contextual information.

4.5.2 Declaration Brevity and Error Message Expressiveness

This was a consideration between the brevity of the declaration syntax for an instructions arguments and the improved ease of understanding for error messages.

The premise of this trade-off was should arguments be a list of types which can (if need be) referenced using their order, (e.g. ”0”, ”1”, etc.) or should it be a map of names paired with types? The decision was ultimately made to utilise names despite the extra work required by the end user due tothe improved error messages made possible by this feature.

4.5.3 One True Approach or Best of Many Approaches

This is an issue which the system has been plagued with throughout its development due to its semi-agile approach, where features are implemented as necessary. Hence features were implemented in the limited scope for which they were needed and then later the scope of these features was often expanded. But in some situations a fundamentally different approach was needed (usually a more complex approach) and the decision needed to be made as to whether the tool should follow the philosophy of Python, specifically the Zen of Python, ”There should be one–

```
1   - name: trap
2     byte sequence: 0x0D
3     size: 4
4     arguments:
5       arg1: register
6       arg2: register
7       arg3: register
8
9   - name: lea
10    byte sequence:
11      - literal: 0xF
12        size: 4
13      - path: destination
14      - path: address.offset
15      - literal: 0x0
16        size: 4
17      - path: address.source
18    arguments:
19      destination: register
20      address: memory
```

Figure 4.23: An example of the competing approaches in the assembler.

and preferably only one –obvious way to do it."[41] or if it the tool should optimise for each of the common use cases.

One of the key examples of this is in fig. 4.23 which illustrates the two competing approaches to declaring an instructions byte code format, the first (as mentioned previously) is the 'basic' format for instructions, this was implemented first to make implementing the simplest instructions as simple as possible. The second (starting on line 9) is the more advanced form in which the user explicitly lays out the instruction in memory.

The first form exists to allow the very simplest of instructions to be written in a brief and easily understood form and the second form allows for instructions which are far more complicated than is possible using the first approach. The trade off is as to whether or not the brevity of the original form was worth the increased code complexity of being able to parse both forms and the extra mental overhead on the user of knowing and understanding both forms.

The decision was made that yes, it was worth the extra overheads involved in supporting the more complicated parsing and the extra complexity for the user, for certain instruction sets (such as Sigma16) which can utilise the first form for a large number instructions allows for a far shorter definition file and the reduced maintenance burdens which that entails. Thus the disadvantages of this approach are out weighed by the shorter declarations.

4.5.4 Performance

A trade off which has been made and then repeatedly remade throughout the course of this project is as to the value of performance for this project.

The basic premise of this project is that it allows the user to rapidly iterate through various different architectures or variants of an architecture for the purpose of either extending them (for learning purposes) or for experimenting with the properties of both the architecture and the output of the assembler (for research purposes).

Although it is not an absolute property of either of this scenarios it is highly likely that the test cases used for many of these situations will be limited in size and scope. It is unlikely that an extremely large program will be utilised for this system and hence it stands to reason that the performance of this tool will likely not become a bottleneck for the situations in which is is intended for use.

Hence the decision was made not to focus heavily on the performance of the system without good cause (if it were to become a significant bottleneck).

4.6 Limitations and Known Issues

The system has several consciences and intentional limitations which limit its capabilities. They are the limited number of offset calculations modes supported 9direct, pc relative and segment relative). The system is also not capable of support all directives of an assembler, this can be remedied in the future but is beyond the current scope of this project.

The current macro system is also unable to permit the scope to leak from the macro into the enclosing scope due to its current implementation, this can be changed but it is not within the current scope of the project. Macros also do not interact correctly with listings, this is due to the architecture of the assembler (fig. 4.13) as a consequence listings cannot display the binary value of a macro due to their implementation and there is no simple solution to this issue and as of yet it remain unresolved.

4.7 Mistakes

There were several mistakes made throughout this project and most of them relatively minor, the most interesting of these are detailed below.

One rather problematic but not insurmountable issue was the systems inability to represent an arbitrary number of nested types in an ”indexed type”. This is not a fundamental issue but would require a rewrite of the indexed type hence changing this was deferred for the length of the project. Not making it flexible was a mistake made at the initial stages of the project and is a limitation inherited from the prototype.

Another issue was the drift between the naming conventions of the configuration and the internal implementation meant that for a few limited cases the behaviour of certain sections of the configuration file can be counterintuitive. This would have been best resolved by creating a disconnect between the configuration naming scheme and the implementation ensuring that a drift cannot occur as there is no direct relationship.

One significant mistake was in the implementation of the SizedBinaryArray class, which is a collection of utility class which allow the system to represent and manipulate binary arrays of arbitrary bit sizes. The initial implementation of this utilised a very fast and efficient direct binary manipulation approach to binary concatenation (for example) and this was used for the majority of the life of the project, but its extending its implementation was a time consuming and error prone process which wasted a great deal of time. Hence it was eventually rewritten to utilise binary strings instead of raw binary manipulations with no significant performance impact for the (small) scale of files which this tool handles. Overall wasting time on performance was not an efficient use of time and it would have been a better use of resources to implement it correctly first then make it fast if necessary.

Overall none of these mistakes was particularly significant but they did result in wasted time which could have been better spent in more important features and testing.

Chapter 5

Evaluation

This section of the report covers the evaluation which were performed for this system. The evaluation was performed through a user evaluation, in an effort as to observe the user experience in using the tool and assess the manual and its associated technical material.

5.1 User Evaluation

The third-party user portion of the evaluation laid out a series of tasks to the user to perform, provided them with a basic manual, a fragment of the definition file and a sheet laying out the task to them. Each user was then tasked to read (or glance through) the manual, offered a brief Q&A session with the developer to clear up any questions and then told to begin with the task, referring to the manual as necessary.

5.1.1 Task

The first task in the exercise was to complete the tutorial provided in the manual. This purpose of this task was two fold: first this task provided an initial learning exercise from which they could complete the other tasks and secondly to force them to go through the full process of building a new assembly language in this assembler, from beginning to end.

Before the task was written some basic limitations were designed for the evaluation, most importantly that the evaluation should be limited to under and hour. It was also decided that there should not be an undue burden on the user in terms of previous technical knowledge or experience.

The time considerations had a fairly strong influence on the task, as asking the user to perform a task in excess of an hour is excessively burdensome and unfair on the user, thus the potential scope of the tasks and the proportion of the system which could reasonably be tested was limited. Effort were made to maximise the exposure to different features by the evaluator, but it was not possible to test every possible vector. For example, the tutorial task asks the user to build an assembler from scratch for a trivial assembly language. Its feature case was designed as a pathological worst case language, designed to utilise as many of the key systems in the assembler as possible in as small a language as possible. For example, its designed to demonstrate each of the possible instruction formats and types, each of the different reference types and meta types for it as well, the configuration options and a host of other small features of the assembler.

```
Your task (should you choose to accept it) is to:

1. First do the tutorial, and glance through the manual to get a grasp of
 ↪  it. This skills you learn here will help you with the remaining tasks.
 ↪  Feel free to refer back to the files produced from it if you get
 ↪  stuck.

The names are related to (but not identical to the real Sigma16).

RX instructions are a fixed 16 bits.
RR instructions are a fixed 32 bits, 16 for the instruction, 16bits for
 ↪  the label offset.

Sigma16 has instructions in 2 forms, RX instructions, which have the form
 ↪  of:
 - 4 bits for opcode
 - 4 bits for destination register
 - 4 bits for regA
 - 4 bits for regB

And RR instructions:
 - 4 bits for the value: 0xF (indicates that its an RR instruction)
 - 4 bits destination register
 - 4 bits for the offset
 - 4 bits to identify which RR instruction this is.
 - 16 bits for a label address

2. Your first task is to add a new instruction definition to the Sigma16
 ↪  assembler. Your goal is to add a new instruction called 'loadxi', all
it does is combine a load instruction and an increment instruction into a
 ↪  single instruction.

 'loadxi', like load, is a RR instruction, with the identity value of
  ↪  0x7.

3. Your second task is to add a second 'add' instruction, except this one
 ↪  has a 4 bit literal as its 3rd argument, instead of a register. It
 ↪  should share the name "add" but it should select the specific
 ↪  instruction based on that argument (hint, use meta to alias the 2
 ↪  instructions onto the same mnemonic). The opcode for this is 0x3.

4. Your third task is to add a new literal, the binary type. These are
 ↪  written as |1001| and are 4 bits long (hint: the correct type is
 ↪  BINARY). It isn't fair to ask you to write the Regular expression
 ↪  without a reference, hence I have provided you with them: "|\\d+|"
 ↪  and "|(\\d+)|".
```

Figure 5.1: The task performed by the user.

The technical knowledge constraints were less influential in the design of the program, mostly limiting the scope of the task in small ways. For example the task asked the user to extend the references block with a new type, for binary literals. Which requires a regular expression to function, although this example used a fairly trivial example relying on the user to have pre-existing knowledge of regular expressions was deemed an unnecessary complication, thus a pre-made one was provided to reduce the burden on the test subject.

Some other small allowances were made for the evaluator as well, such as creating a slightly tongue-in-cheek naming convention for the language, which was deliberately chosen in an effort to keep them entertained and to keep them focused on the task.

The manual can be accessed in the appendix of this document, but provided below is a brief summary of the tutorial in the manual.

- First it provides a general overview of the assembler, what it is, what the various inputs and outputs are, etc.
- It then provides an informal description of the manual its self: its contents; how to read it and highlights components of particular import.
- It then gives the user a brief overview of the tool's command line interface.
- You then have the tutorial.
  - This begins by first illustrating the target language for the user.
  - It then briefly summarises the task of the tutorial
  - Gives a high level overview of the description language
  - Builds the Configuration block, with explanations
  - Builds the References block, with explanations
  - Builds the Instructions block, with explanations
  - Gives the user a task to implement the final instruction, based on the previously implemented instructions
  - Finally demonstrates to the user that they have built an entire assembler, from scratch in under 15 minutes.

5.1.2 Experimental Procedure

The experimental procedure for this task was fairly simple.

It involved first preparing the workspace for the user, creating a fresh copy of the files for each user. Then the files were opened for each user, in order of use (as in the task exercise), first they were shown the manual, then in a text editor there was 5 tabs, 1 for each of the files: first it was the task.md file, which is provided in fig. 5.1; then was the ”uSsembly.asm” file; then the ”uSsembly.yaml” file, which describes the architecture format; then the ”Sigma16.yaml” file, which is a textual description of Sigma16 which is to be extended by the user and finally was the ”feedback.md” file, which is where the user was asked to fill in the feedback document.

Next, the user was sat down and the basic purpose of the tool was explained to them to give them a reasonable impression of what the tool was designed to accomplish, then an overview of each of the tasks was provided to the user, to allow the user to accomplish the task in a reduced time and finally they were asked to perform the task.

After the user performed the task they were then asked to fill in the provided questionnaire and then to have a discussion with the author, in an effort to extract any other details from them and maximise their utility. They were then thanked an a reciprocal offer of evaluation was made to them, no monetary compensation was provided.

5.1.3 Results

The user evaluation was performed for 6 users. Each user was asked to perform the task and their feedback was recorded.

Written Feedback

```
# What did you think of it?
-

# What do you think is missing in the documentation?
-

# What do you think is missing in the program?
-

# What do you like in the documentation?
-

# What do you not like in the documentation?
-

# What do you like in the program?
-

# What do you not like in the program?
-

# Any other comments?
-
```

Figure 5.2: The questionnaire filled in by the user at the end of the experiment.

The overall tone of the responses was positive, but there were common complaints among many of the users primarily relating to the usage of YAML for the configuration format. The capabilities of the tool were generally not mentioned.

It is likely that several of the complaints by users in this document relate to certain aspects of the system lacking documentation relate to their inability to read the full documentation for the system, as discussed in section 5.1.3, although the complaints certainly are due consideration regardless of this.

The biggest concern for the users was YAML. Four of the six users raised concerns about the usage of YAML for this tool and specifically the indenting used to designate the structure of the YAML document. Despite two users failing to mention YAML as a problem in their feedback it was still observed as an issue for the user where each struggled with the indenting of the code samples.

There are likely multiple causes at play for this issue, it is unlikely to have a single root cause for the problem. As described in section 5.1.3 the manual is provided as PDF which was observed not to maintain the indenting characters, thus when they are copied by the user the user must manually re-indent the text, which the users struggled with. Assigning the full blame for this issue to this cause is incongruent with the observation that these issues also occurred in the independent section of the implementation, where the user was expected to implement the functionality without explicitly provided material. This may be best solved by providing the tutorial as an HTML document, which is explicitly designed to be distributed in a digital format and has no issues relating to selection.

Another of the potential causes of this issue is structure of the description format, which requires a very dense and an arguably inconsistent format. The format for example very frequently utilises lists of maps (frequently of 1 element) which may have made it difficult for an inexperienced user to scan the document and rapidly observe any flaws in indentation. The other issue may be that the document frequently uses all combinations of maps and lists where appropriate (maps of maps, maps of lists and lists of lists) which may be a source of potential confusion for the user. If this is the problem the likely solution would be create a new schema which encourages a less dense and more consistent structure for the data.

The final issue is that nested data structures may be fundamentally unsuited for indentation based description languages. Even Python, for example, utilises braces to delineate its data structure literals and does not rely on whitespace for these purposes. Which supports the hypothesis; but unfortunately proving or disproving this assertion is beyond the scope of this project. This may only be solved by pursuing an alternative markup language for the tool, or abandoning the description format in favour of a DSL.

A related suggestion which was offered was to implement an IDE plugin to provide tool assistance when developing description files. They would aid in debugging and developing the formats as it would shore up many of the issues raised during the evaluation. For example it would allow the user to receive immediate and precise information as to where they have made an error, allowing them to correct it immediately. It would also aid with the mentioned issues where the error messages do not make it immediately obvious where an error has occurred by providing explicit highlighting on the problem area. It could also provide inline documentation to aid with user comprehension and its syntax highlighting could help improve the user's comprehension of regular expressions, which two users complained about.

A user also complained about the naming scheme which is used in the description language. They felt that although it may reflect the internal structure of the tool it does not make it obvious to the user as to what exactly each property means. As this user correctly observed the names selected were chosen to match their internal naming inside the tool but as the internals of the tool have developed their names have diverged from those of the description format and now there is an impedance mismatch based on those names, where those names may no longer possess any link to either their internal purpose or any obvious metaphor to describe their purpose. These names should be reconsidered and should be considered without consideration of their internal names as to avoid this issue in the future.

People also felt that the manual should be longer and offer more material for them to help them understand the basics. This was the intent when writing the tutorial in the manual and appears not to have achieved its goal with complete success. A possible solution would be to extend the length of the tutorial with more explanations, or add ”quick hints” sections to the tutorial to provide small relevant hints to the user and provide the pertinent information in a more digestible and briefer manner than is present in the rest of the manual. Section 5.1.3 also discusses how the users failed to read the manual properly which may also be a root cause of this and is tricker to counteract, there does not appear to be any obvious solution for this issue except those already discusses, add the information to the areas of the manual which the users to read (the tutorial).

The other feedback was collated into a single document and may be reviewed in its raw farm in the appendices.

The evaluation did highlight some bugs, for example an error message missing quotation marks, which made it difficult to realise that the error was informing the user as to where the error was occurring.

Overall the user feedback has been positive towards the tool with no complaints as to the actual functionality of the tool but instead instead offers suggestions about the user suggestions. In hindsight an alternative to YAML should have been considered due to the issues which users faced with the tool, but it was not obvious at the time of the original decision that end users would react this negatively to YAML's whitespace sensitivity. Despite these issues though the users overall had a positive experience with tool.

To quote a user who was asked to describe how they felt about the tool;"Easy to use (it is simple) and effective, but hard to get to know for a starter especially when I have not done much assembly before and it was years ago"

The overall tone of the responses indicate that outwith the issues relating to YAML the majority of the complaints were relating to the initial 'hump' that is a new user's inexperience with a tool leading to difficulties understanding the system but feedback has indicated that were more "handholding" to be provided then the experience becomes reasonably straight forwards. Which allows the tool to meet its non-functional requirement of being easy to use even when YAML is considered.

When the issues are considered from the perspective of the tool having aided subjects who have never written an assembler (and some never used an assembler) before to develop an assembler in approximately 15 minutes and then independently extend upon an unseen assembler the complaints are still significant but can be viewed in terms of the bigger picture; of having succeeded in aiding inexperienced users implement an assembler.

Observations

Overall, there were several points picked up during evaluation, which a user may not mention in their written evaluation but still observe as an issue.

For example, not user mentioned it, but observation highlighted several problems with the format of the tutorial. It was provided as a PDF which by the nature of a PDF makes copy-pasting from the file, into the file very problematic, as PDF text selection was a rather problematic issue and the users had difficulty actually copying the code from the tutorial and this may have made them feel more negative about YAML than should be, but it wouldn't cause all of the negative feelings towards YAML. This can be solved in several way, but the best approach may be to provide an HTML version of the tutorial and manual, which would also avoid formatting issues (relating to line overflow).

Another interesting observation which was made of the users execution of the tasks. The time taken by the users to complete the whole exercise fell into two categories, users who had more recent experience working with assembly languages, and completed the task in approximately 30 minutes, or less. These users had far less trouble in understanding the concepts presented by the assembler, but it was still note entirely intuitive for these users. The second class of users had not worked with an assembly languages in the last year and generally took 1 hour to complete the task, far longer than originally anticipated for even these users. They struggled more with many of the basic concepts of the assembler as well and could not grasp the concepts as intuitively. Some alterations which could be made to improve the experience for less experience users would be to place in "tip" blocks in the tutorial and in the main body of the manual, to highlight key points and give informal hints to the user.

One final interesting observation was that users tended not to read too far into the manual beyond the tutorial, they read the tutorial in depth and several read a couple of pages beyond it, but not a single one read beyond that unless they spent an inordinate proportion of their task on a single component of the task. Partially, this may be due to the advice in the introductory paragraph, but even that suggested they peruse it for interesting and useful details. The conclusion from this observation is that the user would likely prefer a more brief manual which could be more easily skimmed, and that the advice that they may skim the manual is being taken as tacit permission to ignore it. Thus that paragraph should be altered to not mention that advice.

5.2 Requirements

The tool meets all the ”must have” and ”should have” requirements as well as well as five of the eight ”could have” requirements, which is reasonable progress. Overall the implementation methodology aided in implementing these requirements and ensured they were implemented in order of need rather than arbitrarily which was a useful property, due to the authors relative inexperience in building assemblers.

Chapter 6

Conclusion

6.1 Summary

The main aim of this application was to create a tool which can permit a user to rapidly develop and iterate on various instruction set architectures and potentially share these architecture definitions with other users. The implementations of Sigma16 and MIPS have demonstrated that the system is capable of rapidly developing and iterating over the design of an assembler and the implementations of ARM and Intel 8086 have demonstrated the limitations of the system, primarily in how there can be a rapid growth in the size of a declaration file as the number of addressing modes grows.

Overall, despite the limitations it success and its ability to meet all of the ”must have” and ”should have” as well as most of the ”could have” requirements I believe the software was a success, despite the issues.

6.2 Future Work

There are many opportunities for future work in the system, the most obvious candidates are to improve the reference system to support 1 or more nested references rather than the current fixed size of 2 arguments. Other obvious candidates for implementation are linker support, full implementations of Intel 8086 and a more through test for the ARM implementation. Other features which may be desirable are fixed ranges of values for references, which may be useful (for example) for implementing jumps of various sizes and a wide variety of new directives would be useful as well as a plugin mechanism for loading a shared library of directives. Implementation of a disassembler and a companion virtual machine may also be worthwhile avenues of investigation. This is just a summary of several of the many avenues for future work.

6.3 Lessons Learned

Various lessons learned have been mentioned throughout this report but the most important take aways from this report are the importance suiting a reasonable scope. Issues in minimising the scope of the ARM and Intel 8086 reduced the time I have available to implement them fully, leading to an inability to fully test those architectures. The value of an iterative development model has also proven to be of great value as it as allowed me to prioritise the features which matter which I think has been if great benefit to me, otherwise I tend to loose momentum. Overall though I think the project has succeeded in its goals and was successful.

Acknowledgements

I would just like to thank Dr John O'Donnell for is incredible patience in handling my bumbling around and for all of the advice he has given. I can only hope I have followed it.

I would also like to thank Dr Simon Gay for his advice in the last two weeks of the project it was extremely valuble advice at an important time.

Thanks.

Appendices

Generic Assembler
Manual

Nishad Mathur

March 28, 2016

Contents

1 General Overview

1.1 About the Assembler

In brief, the purpose of this program is to provide a re-configurable assembler.

It provides 2 mechanisms through which this may be accomplished, firstly, you may use the YAML [1](a textual serialization format) configuration format in conjunction with the command line tool or you may use the system in a library format, configuring the assembler in code using it in a DSL (Domain Specific Language) like format. This manual is concerned primarily with the prior and will not cover the DSL format in any great detail.

1.2 About the Manual

I have tried to keep the manual fairly informal and brief, listing out the important information and adding (hopefully) illustrative examples along the way. After reading the 5000 page ARM architectural review manual I've gotten thoroughly sick of the dry manuals provided there, so this is my (admittedly poor) response to that.

I don't intend that you read the whole manual in one go, it may be worth skimming over the errors list 5 and the file-structure 4 and looking over them as you need them. The examples should illustrate the concepts in use if that is your preferred learning style.

There is a full example in the appendix which could be an interesting starting point. It is a mash-up of several architectures, so isn't useful on its own but it should give you a useful starting point and an idea of how it should be structured.

2 Running the Command-line Tool

Running the tool via command-line is a fairly trivial exercise the basic format of the tool is to use "`java -jar generic-assembler --input banana.asm --config lanugage.yam`

which will run the tool, parse the configuration and then output the listings to standard-out.

–help Provides a help listing of all commands, similar to this.

–input Provides the path to an assembly file to compile.

–output Provides the path to a destination file for the compiled file

–listings Outputs the listings instead of the assembly file

–config Provides the path to the configuration file.

```
// Ussembly!
// The impossible has happened, now it falls
// upon you to make sure the world knows it...
// Prepare the world!

hell:
    freeze water
    put pig0, #wings[0]
    goto #hell

animal:
    .data 0xDEADBEEF
```

Figure 1: The file (which will control the world, colloquially known as our assembly language file)!

3 Tutorial - uSemmbler for uSembly

The language is very simple, it has 3 operands, ”freeze”, ”put” and ”goto” as well as a special command called ”.data”. Not much you can do with it, but describe a lovely summers day in hell. It also has a couple of registers, called pig0 to pig 3 and water.

3.1 Getting Started

First things first, we need a couple of files to start with; lets call them `ussembly.yaml` for the assembly description file and `test.asm` for the assembly file.

Then go ahead any copy the contents of figure 1 into the `test.asm` file.

```
configuration:

references:

instructions:
```

Figure 2: The skeleton configuration file.

Next, copy the contents of figure 2 into the `ussembly.yaml` and save that.

Then we can go ahead and try that using the command `java -jar assembler.jar --config uSse`
(it should throw up an error).

3.2 Filling in the Configuration Block

So lets go ahead and flesh out that configuration block the error was complaining about. But nefore we start, I will explain what exactly it is: the configuration block is where all the global settings go, the settings which don't fit else where or effect everything.

```
configuration:
  line can start with label: true
  label bit size: 4
  word size: 16
  label regex: "^(\\w+):"
  argument separator: ",\\s*"
  comment regex: "//.*"
```

Figure 3: The Configuration Block

I'll quickly go through an explain each bit for you:

line can start with label asks whether or not a line with an instruction on it can start with a label.

label bit size the maximum number of bits a label address can take up (you can reduce this on a case by case basis).

word size declares how long a word is (this is used to calculate offset sizes for labels).

label regex asks for a regular expression in which the first group captures a label's identifier.

argument separator asks for a regex which is used to split the arguments of an instruction up.

comment regex captures a whole comment into a regular expression.

There are also, some optional options, relating to endianess, but I will leave those be for now.

Now the assembler knows how big a word is, how big a label is, what a comment looks like, what an argument looks like and what a label looks like and honestly, that's all it needs to know here.

3.3 Filling in the References Block

Now it should be complaining that the references block is missing or malformed (mostly missing in this case), so lets get started on filling that in.

Lets start with the registers then, shall we? The first part of this block should look like this:

```
references:
  - name: register
    kind: meta
    references:
      - name: filthy capitalist pigs
        kind: mapped
        size: 4
        range:
          format string: "pig{0}"
          start index: 0
          start literal: 0
          count: 4
      - name: things
        kind: mapped
        size: 4
        mappings:
          water: 0x4
```

Figure 4: The References Block - Registers

The gist of this one is that we have a bunch of registers, all of which can be refereed by their collective name of "register" (or if your feeling a little bored, "filty capitalist pigs"). Under this group are two sub groups "things" and "filthy capitalist pigs", where there is one register under things (called water, with the value of 0x4) and 4 registers under piggies (call pig0 with the value of 0x0 through to pig3 with the value 0x3).

If you're curious, the format string option expects a format string where the first item is the replaced value see the java MessageFormat docs [3] for more details.

Then we add the literals(figure 5): the numbers (like 1, 2, 100...) and the hex (like 0x0, 0x3, 0xFFFF...) literals. Here we have the literals grouped under a group called "literal" (you could call it plebs if you prefer, the name is arbitrary) and we have 2 types of literal underneath it, hex literals and integer literals. Their options are fairly self, explanatory (but if not check

```
- name: literal
  kind: meta
  references:
    - name: hexadecimal
      kind: literal
      literal type: HEXADECIMAL
      literal size: 4
      validation regex: "0x[0-9A-Fa-f]+"
      extraction regex: "0x([0-9A-Fa-f]+)"

    - name: int4
      kind: literal
      literal type: INTEGER
      literal size: 4
      validation regex: "\\d+"
      extraction regex: "(\\d+)"
```

Figure 5: The References Block - Literals

out section 4).

```
- name: label
  kind: label
  size: 4
  validation regex: "^#\\w+$"
  extraction regex: "\\w+"
```

Figure 6: The References Block - Label

This ones (figure 6) just simple, if you need to reference a label, then this describes their format, a string of characters which starts with a '#'.

This one (figure 7) is paradoxically both fairly simple and a little tricky to understand (these names may change in the future). So this is what each bit means:

source before offset controls whether the default byte order for this has the source (lhs) output before the offset (rhs) or not.

regex has 2 capture groups, the first for the source, the second for the offset.

```
  - name: memory
    kind: indexed
    source before offset: true
    regex: "(.*?)\\[(.*?)\\]"
    valid left hand types:
      - label
    valid right hand types:
      - literal
```

Figure 7: The References Block - Memory Access

valid [left—right hand types] declares which types (i.e. the references we defined earlier) can be embedded on each side.

And that is each of the bits. I will put the whole thing at the bottom, if you've had any issues, but it should be straight forward to put these pieces together.

3.4 Instruction Block

And *finally* we have the instructions them selves.

Before we actually get started on this, I'm going to go ahead and specify these properly for you.

freeze has 1 argument, a register. It's binary form has these value:

1. 4 bits identifier: 0x1
2. 4 bits register
3. 8 bits padding: 0x00

put has 2 arguments, a register and a memory access with a label and a literal offset.

1. 4 bits identifier: 0x2
2. 4 bits destination register
3. 4 bits label
4. 4 bits literal offset

goto has 1 argument, a label

1. 4 bits identifier: 0x1

2. 4 bits padding: 0x00
3. 4 bits label
4. 4 bits padding: 0x00

.data has 1 or 2 arguments, each is a literal

1. 4 bits literal

3.4.1 put

So lets start of with the conceptually simplest one ”put”. This one is straight forwards and is implemented as in figure 8. You just put the sentinel using ”byte sequence” and specify how many bits long it is. Then the arguments are extracted and then synthesized in the order in which they are defined here.

```
 - name: put
   byte sequence: 0x02
   size: 4
   arguments:
     destination: register
     value: memory
```

Figure 8: The Instructions Block - Put

3.4.2 freeze

This instructions is probably the most complicated of the 4 to implement. Mostly, because of those padding requirements. So this one differs in that you declare each literal or argument and the number of bits it takes up.

3.4.3 .data

This one is *really* easy. It doesn’t have any sentinels, or anything, it just has 1 catch, you can have 1 or 2 arguments. The solution though, is simple, remember how we had multiple registers under the same name? Well we can do the same here, with a meta block. Check out figure 10 to see how i solved it.

```
- name: freeze
  arguments:
    thing to freeze: register
  byte sequence:
    - literal: 0x1
      size: 4
    - path: thing to freeze
      size: 4
    - literal: 0x0
      size: 8
```

Figure 9: The Instructions Block - Freeze

3.4.4 Goto

This instruction should be very similar to freeze, so I'm going to use this as an opportunity to let you try this you self, and then come back and check it. I've put my version (not the only answer!) into figure 11.

3.5 Running it!

Now you should have it all put together and ready to go. Go ahead and try run it using: `jav -jar assembler --config uSsembly.yaml --input test.asm --listings`

If it all goes well you should get something out at the end which looks like figure 12. If it doesn't look like that, then go ahead and look in the co

```
- name: .data
  kind: meta
  instructions:
    - name: .data1
      byte sequence: 0x0
      size: 0
      arguments:
        data0: literal
    - name: .data2
      byte sequence: 0x0
      size: 0
      arguments:
        data0: literal
        data1: literal
```

Figure 10: The Instructions Block - Data

```
- name: goto
  arguments:
    label: label
  byte sequence:
    - literal: 0x3
      size: 4
    - literal: 0x0
      size: 4
    - path: label
      size: 4
    - literal: 0x0
      size: 4
```

Figure 11: The Instructions Block - Goto

```

```

Figure 12: The Assembled File

4 File Structure

First things first; the most important thing to understand about the configuration format is that it is just plain old YAML, so any of the clever things you can do there, you can do here so don't forget that fact and use it where you find it helpful. For example you can use anchors and aliases to deduplicate blocks of code.

Secondly, I've tried to give helpful error messages in the assembler, but errors in your configuration file can manifest in weird, wonderful and not entirely obvious ways, so remember to keep an eye out for that.

The basic gist of the file its self is that it is a YAML file (hence the '.yaml' extension) which is passed into the assembler, loaded and built into an assembler configuration. If you find it limiting, you can always extend it yourself, using the assembler as a library, but that isn't covered in this manual (your best bet there is to check out 'com.nishadmathur.main').

The file its self is composed of four top level blocks: 'configuration', 'references', 'instructions' and (optionally) 'instruction formats'.

4.1 Common Information

The intent with this configuration was to maintain consistent formats, names and levels of flexibility throughout and as a consequence there are several conventions (for the format and YAML) which are helpful to know when creating a custom configuration.

4.1.1 YAML

YAML (a recursive acronym, YAML Ain't a Markup Language)[1] is a human readable serialization/markup format optimized for readability, consistency and simplicity. It is format is very convention driven and many of those are enforced by the parser. The more fundamental of which are explained below.

4.1.2 Configuration Conventions and Requirements

Not all of these are hard requirements, but they are strongly suggested and erring from them is not thoroughly tested.

2 Space Indentation is strongly suggested format, tabs are not supported for indentation and using 2 spaces per indent level aligns neatly with the array syntax, so is the suggested depth.

Dictionary keys and values for example names, such as path and the path value may contain an arbitrary name, including spaces and punctuation, and this is the suggested format as it improves readability and expressiveness of the language.

Naked Strings are the preferred form of string, avoid using the enclosing quotes except where necessary. Certain strings may require escaping or embedding in a string literal (see the YAML documentation for further details [2]).

Space Between Sections is a suggestion which more clearly delineates each large block from one another allowing the user to better skim over the format.

Use Multiline Dictionaries and avoid dictionary literals. It is also preferred to start the dictionary on the same line as the list (e.g. `- key: value`), if the dictionary is nested in a list.

4.2 Configuration

```yaml
configuration:
  line can start with label: true
  label bit size: 16
  word size: 16
  label regex: "^(\\w+)"
  argument separator: ",\\s*"
  comment regex: ";.*"
```

This block sets up the global configurations for the assembler:

line can start with label : This switch controls whether or not the first character of the line is defined as a label.

label regex : This regular expression is used to identify and extract a label from a line, it is called on every line and the first capture group in the regex should contain the labels name.

argument separator : The instructions are split on this regular expression, so for example 'jmpf cond label' would be split on `"\^(\\w+)"` into the components 'cond' and 'label'.

comment regex : This expression should have a single capture group which matches on a comment.

word size : This used for calculating label offsets.

label bit size : This is used to control the default size of a label reference.

4.3 References

The references section defines the types which are essentially analogous variable types.

```yaml
references:
  - name: literal
    kind: meta
    references:
      - name: literal4
        kind: meta
        references:
          - name: hexadecimal
            kind: literal
            literal type: HEXADECIMAL
            literal size: 4
            validation regex: "0x\\d+"
            extraction regex: "0x(\\d+)"

          - name: int4
            kind: literal
            literal type: INTEGER
            literal size: 4
            validation regex: "\\d+"
            extraction regex: "(\\d+)"

  - name: registers
    kind: meta
    references:
    - name: general registers
      kind: meta
      references:
      - name: 64bit general registers
        kind: mapped
        size: 4
        range:
          format string: "[X|x|r]{0}"
          start index: 0
          start literal: 0
          count: 32

    - name: special registers
      kind: mapped
      size: 4
      mappings:
        PC: 0
        SP: 0
        WSP: 0
        ELR: 0

  - name: label
    kind: label
    size: 16
    validation regex: "^\\w+$"
```

Meta : Following on from the type analogy, meta types are akin to an interface, every child type (transitively) conforms to the type. For example, literal4 is a literal and int4 is a literal4 and a literal.

```yaml
references:
  - name: literal
    kind: meta
    references:
      - name: literal4
        kind: meta
        references:
          - name: hexadecimal
            kind: literal
            literal type: HEXADECIMAL
            literal size: 4
            validation regex: "0x\\d+"
            extraction regex: "0x(\\d+)"

          - name: int4
            kind: literal
            literal type: INTEGER
            literal size: 4
            validation regex: "\\d+"
            extraction regex: "(\\d+)"
```

Mapped : This type exists for named constants, for example for registers. As a note, there are two approaches to defining a mapped type.

```yaml
references:
  - name: registers
    kind: meta
    references:
    - name: general registers
      kind: meta
      references:
      - name: 64bit general registers
        kind: mapped
        size: 4
        range:
          format string: "[X|x|r]{0}"
          start index: 0
          start literal: 0
          count: 32

    - name: special registers
      kind: mapped
      size: 4
      mappings:
        PC: 0
        SP: 1
        WSP: 2
        ELR: 3
```

The first is where you define the names of a mapping as a regular expression and the literal values associated with each directly. The alternative approach is to define a 'range' block, where you define the base case for the format-string and the literal value and the number of constants that are defined. Each range mapped value has the value incremented by one.

One point to note, the format string option expects a format string where the first item is the replaced value [3].

Indexed : Index is a compound type designed specifically for label + offset references but can be re-purposed for other binary operations. The field 'source before offset' defines whether the second field (the offset field) goes first in the binary form, by default it is placed after, but this reverses it. 'valid left hand types' and 'valid right hand types' allow you to list out which types are valid in the left and right hand fields respectively.

```
  - name: memory
    kind: indexed
    source before offset: true
    regex: "(.*?)\\[(.*?)\\]"
    valid left hand types:
      - literal16
      - label
    valid right hand types:
      - registers
```

Label : This essentially defines the users label reference, which differs from the label declaration (defined in the configuration block). This for example may be used to reference a location for a jump instruction ('jmp #loop') and the offset is later calculated and the label is substituted by the binary form of the offset value.

```
  - name: label
    kind: label
    size: 16
    validation regex: "^\\w+$"
    extraction regex: "\\w+"
```

Literal : This is the other primitive type for the assembler, this one is used for literal references, for example you would use it for your integer type '1' or your hexadecimal type '0x7d' (among others).

For The 'literal type' field, you must select a type from 'BINARY', 'HEXADECIMAL' or 'INTEGER'. Each of which corresponds to the obvious formats, base 2, base 16 and base 10. These are converted directly to their binary form, so '0xB' would convert to '1011'. Binary follows the standard 2s compliment format and binary is a direct conversion. #TODO look into replacing this with a base command instead.

```
references:
- name: int4
  kind: literal
  literal type: INTEGER
  literal size: 4
  validation regex: "\\d+"
  extraction regex: "(\\d+)"
```

4.4 Instructions

Instructions are the bread and butter for an assembler, hence defining the instruction correctly properly is key to producing a working assembler. There are 2 primary concepts at play here instruction normal instruction definitions and meta instructions. 'Normal' instruction definitions encompass an instruction which is selected solely on its name (although it is still type checked) and 'Meta' instructions are instructions which can have multiple different forms, where the layout depends on the types of its arguments as well as its name.

```
- name: add
  byte sequence: 0x00
  size: 4
  arguments:
    destination: register
    lhs: register
    rhs: register

- name: mov
  kind: meta
  instructions:
    - name: mov rr
      byte sequence:
        - path: destination
          size: 5
        - literal: 0
          size: 11
        - path: source
          size: 5
        - literal: 0x2A8
          size: 11
      arguments:
        destination: 64bit general registers
        source: 64bit general registers

    - name: mov rl
      byte sequence:
        - path: destination
          size: 5
        - path: literal
          size: 16
        - literal: 0x694
          size: 11
      arguments:
        destination: 64bit general registers
        literal: literal

- name: .data
  byte sequence: 0x00
  size: 0
  arguments:
    data: literal32
```

4.5 Normal Instructions

Normal instructions will make up 80-90% of the instructions which you define. They are selected for using their name and their type information is then validated (if there is a mismatch an error is thrown). There are 2 (or really 3, but that will be explained in 4.6).

Byte Sequence Literal is the simpler form an instruction definition. It is designed for the most trivial use case of instruction literal followed by arguments (using their default sizes and formats). For this form 'byte sequence' is defined by passing a literal value and then 'arguments' is a dictionary composed of name and type pairs where name is name used for help and error messages and type is used to type-check and generate the binary values of the arguments.

```
 - name: add
   byte sequence: 0x00
   size: 4
   arguments:
     destination: register
     lhs: register
     rhs: register
```

Byte Sequence List is deigned to build up instructions which require an arbitrary (but fixed size) format and argument layout. You can place literals, arguments and sections of arguments in any order for the instruction.

The byte sequence definition here is a list of dictionaries where each dictionary defines either a literal value or a reference to an argument.

Literal is defined simply by providing the value using the field 'literal' as literal and providing the size of the segment (as a number of bits) as an integer.

Path permits you to reference any argument and (if supported) any subsection of that argument. The path is recursively resolved and will return the value at that path, or raises an error if it cannot be resolved. Currently, only the 'memory' reference type supports nested resolution and it has the fields: 'source' and 'offset' which reference the first and second fields of a reference, for example "address.offset" to reference the offset field of the address argument. For the path segment, the size is optional.

```
- name: mov rl
  byte sequence:
  - path: destination
    size: 5
  - path: literal
    size: 16
  - literal: 0x694
    size: 11
  arguments:
    destination: 64bit general registers
    literal: literal
```

4.6 Instruction Format

Instruction formats are designed for the purpose of deduplicating the byte sequence list for the **Byte Sequence List** format of the 'normal' instructions. You define the name of the instruction format and then, in the same form as in the byte sequence list, you list out the segments of the byte sequence. The paths here are resolved in the context of their instruction which references them.

```
instruction formats:
  compare and branch:
    - literal: 0xA2
      size: 8
    - path: label
      size: 19
    - literal: 0x01
      size: 1
    - path: cond
      size: 4
```

One additional feature that this supports in instructions is the 'aliases' block, where instruction paths can be defined to point to an existing argument, for example if there is a naming mismatch between the format and the argument names, or if you wish to refer to a subsection of an argument, e.g. the offset.

It may be preferable to use aliases instead of renaming the arguments as it may give far more helpful names and fields in the automatically generated help.

```
instruction formats:
  compare and branch:
    - literal: 0xA2
      size: 8
    - path: label
      size: 19
    - literal: 0x01
      size: 1
    - path: cond
      size: 4

-- This is just an example.
instructions:
  - name: jmpf
    instruction format: compare and branch
    aliases:
        cond: condition
        label: destination
    arguments:
      destination: label
      condition: register
```

5 Errors and Explanations

There are a multitude of errors types in this system but in general and they are each designed for use in a specific context, some are specifically for use during the Configuration parsing phase, some are internal errors which should never occur during normal operation and lastly some are raised during the assembly file, parsing, assembling or output stage.

One important consideration to keep in mind when hunting down as error is that certain types of mistakes in the configuration stage can manifest them selves as errors during the assembly phase, for example if your regular expressions fail to capture the expression they may manifest as run-time errors in the assembler. So, be careful about that.

The various errors are enumerated below in a non exhaustive list (some error types are currently not in use and are excluded).

Lastly, if none of this is of any help, this is a JVM program and you always have the nuclear option of using a JVM debugger to step through the program directly. This program's source should be available and if not, it employs no obfuscation techniques, so feel free to de-compile it an inspect it that way. As a note, it is a Kotlin program so some of the decompiled code is non-idiomatic in java.

5.1 Configuration Parse Errors

Due to the design of configuration parser, line level granularity of errors is not available.

IncompleteDeclarationParserError indicates one of two things. Either that you are missing or have misnamed one of the three primary blocks ('instructions', 'references' or 'configuration') or that you have referenced an instruction format which doesn't match any of your declared formats. In both cases check the spelling of both formats.

MalformedDeclaration indicates that the format of your declarations doesn't match the form that the configuration parse expects. See 4 for details.

InvalidOption is almost a catch all error type for anything for which a more specific error does not exist. But its primary purpose is for declaring missing fields in the configuration blocks where it highlights exactly which field is missing and from which block.

5.2 Internal Errors

These errors should ideally never be raised during normal operations, but if they are, contact me and I would appreciate it if you could attach the stack trace and any documents required to reproduce the error.

InvalidImplementation is raised if a module (internal or an extension) has incorrectly implemented some API.

5.3 Assembler Errors

These are the errors you see during if an error is encountered when trying to lex, parse or assemble the actual assembly language file. These errors should be annotated with the line numbers, to allow you to easily track down an error and identify its cause.

InstructionParseError is raised if an instruction is correctly identified but its arguments cannot be parsed correctly into an expected form. It should display the acceptable forms for the instruction.

DataSourceParseError occurs when a reference type fails to be parsed correctly, for example if a (potentially recursive) type such as indexed references cannot match on one of the nested types, e.g. cannot match on the offset for a[0xx1].

LineParseError generally occurs when there is text on the line which is un-handled, for example if you have a comment regular expression which does not capture the whole comment block.

UndeclaredLabelError is raised if you reference a label for which the assembler cannot find the declaration. Generally check spellings here.

IncorrectTypeError occurs when an instruction is identified and its arguments are fully parsed, but they do not match the types expected for that instruction, fro example if an integer is passed instead of a register constant.

AbstractInstructionInstantiationError is the meta-instruction counterpart to 'InstructionParseError' and 'IncorrectTypeError'. It is raised if an instruction is identified as a meta-instruction but does not match the type of any of the candidate instructions.

PathResolutionError is raised if a path from an instruction declaration cannot be resolved. For example if you reference an argument which doesn't exist, or if there is a spelling error, or if for example if you reference a nested path and make an error there.

Ideally this should have been raised during the configuration parsing phase but currently it is raised during the assembly phase and this may change in the future.

6 Frequently Asked Questions and Common 'Gotchas'

I have an error, what should I do about it? RTFM. 5 specifically discusses the errors, general conventions surrounding them, normal causes and debugging tips.

What about cyclic references? Currently, the system has no system for cyclic references or forward declarations. The best option for dealing with those is through the usage of meta-references as container types or carefully avoiding circular references where possible.

7 Appendices

7.1 Complete ”test.asm”

```
// Ussembly!
// The impossible has happened, now it falls
// upon you to make sure the world knows it...
// Prepare the world!

hell:
    freeze water
    put pig0, #wings[0]
    goto #hell

wings:
    .data 0xD, 0xE
    .data 0xA, 0xD
    .data 0xB, 0xE
    .data 0xE, 0xF
```

7.2 Full Example

```yaml
configuration:
  line can start with label: true
  label bit size: 16
  word size: 16
  label regex: "^(\\w+)"
  argument separator: ",\\s*"
  comment regex: ";.*"

references:
  - name: literal4
    kind: meta
    references:
      - name: hexadecimal
        kind: literal
        literal type: HEXADECIMAL
        literal size: 4
        validation regex: "0x\\d+"
        extraction regex: "0x(\\d+)"

      - name: int4
        kind: literal
        literal type: INTEGER
        literal size: 4
        validation regex: "\\d+"
        extraction regex: "(\\d+)"

  - name: literal16
    kind: meta
    references:
      - name: hexadecimal
        kind: literal
        literal type: HEXADECIMAL
        literal size: 16
        validation regex: "0x\\d+"
        extraction regex: "0x(\\d+)"

      - name: int4
        kind: literal
        literal type: INTEGER
        literal size: 16
        validation regex: "(\\+|-)?\\d+"
        extraction regex: "((\\+|-)?\\d+)"

  - name: register
    kind: meta
    references:
      - name: general purpose register
```

References

[1] Clark C Evans. Yaml.

[2] Clark C Evans. Yaml specification.

[3] Oracle inc. Java javadoc - message format.

```
# What did you think of it?
- Simple enough to use
- yaml was a pain in the ass.
- Could use a briefer and simpler tutorial with clearer run/show
 ↪ examples.
- Interesting exercise. Never tried defining my own instruction set
 ↪ before.
- Didnt think. Did.
- Easy to use (it is simple) and effective, but hard to get to know for a
 ↪ starter especially when I have not done much assembly before and it
 ↪ was years ago
- After some explanation, the instructions were easy enough to follow.
- main problem was wrestling with the formatting of yaml; 9/10 of my
 ↪ errors came from this, as opposed to doing something wrong elsewhere

# What do you think is missing in the documentation?
- longer example/tutorial
- Config yaml is indentation-sensitive.
- What are the funny '\\' for? Are they part of the language?
- The documentation is good apart from the fact that it would be helpful
 ↪ to define the concepts of string matching.
- Data/Field/Reference types in the yaml refer to previously defined
 ↪ identifiers.
- A bried explanation of yaml would help.
- If its used for less experienced students then the manual is missing
 ↪ the very fundamental concpets.
- How each thing does what.
- Maybe an alternate format: i.e. ensure the explanation is next to the
 ↪ code
- More examples and explanation on how to use the system - what to type
 ↪ for size; are all fields optional or are they required/mandatory;
 ↪ maybe include some links to which regex 'system' is used; include the
 ↪ different options for variable types and other lists of information
 ↪ in case they are not there (did not read the whole documentation, its
 ↪ long for a beginner)
- Some kind of basic introduction for new users (although if the tool
 ↪ isn't really aimed at newbies then this doesn't really apply)
- an appendix that contains the full file; just to make formatting easier


# What do you think is missing in the program?
- more rapid feedback.
- Bananarama
- Auto completion?
- IntelliJ support for suggestions
```

Figure 1: A summary of the feedback collected.

```
- Never used a program like this, so I wouldn't know what one
 ↪  should/shouldn't have
- maybe some form of error catching? (although if most of these errors
 ↪  are related to YAML, this is less relevant)

# What do you like in the documentation?
- Formatting is "low impedance"
- functional
- I like that there were plenty of useful examples.
- Manual did what it needed to at the end of the day

# What do you not like in the documentation?
- The slightly broken formatting is annoying.
- Although funny, some more meaningful names would help.
- Figure positioning was mucked up.
- useful at this level and doesnt go into too many tangents
- Copy Pasting from the manual was a "bit of a bugger" because its a pdf,
 ↪  other formats might be better
- regular expressions weren't well explained but were mentioned with a
 ↪  vague unexplained example
- Documentation is a bit abstract for a beginner.

# What do you not like in the program?
- YAML was irritating
- Tool could do with a little more polish
- YAML formatting is painful. Brackets would be easier.
- I think the naming is not very intuitive. It might reflect what is
 ↪  happening under the hood, but that is not entirely obvious to a
 ↪  programmer that simply wants to use the framework.
- Indenting.
- Error messages not clearly highlighting the problem area. They mention
 ↪  the problem but dont highlight it, which makes it more difficult to
 ↪  track down.
- variable types were not completely clear/mildly confusing
- regex were incomprehensible
- YAML.

# Any other comments?
- After some explanation, it was quite intuitive to work with
- Tried to goto memory or label?
- Had to rush last parts.
- The syntax is fairly easy to read and would be easy to produce given a
 ↪  simplistic IDE without any smart indentation functions.
- mac config was different but fine.
- No previous experience with typed languages
- Only python and a tiny bit of java
- spaces/indentation could be done automatically (through IntelliJ)
- From my perspective, could have done with some spoon-feeding at the
 ↪  start
- After some explanation, it was quite intuitive to work with
- Tool assistance would be handy
```

Bibliography

[1] Achal Aggarwal, Sunil K Singh, and Shubham Jain. A hybrid approach of compiler and interpreter.

[2] Apple. *The Swift Programming Language*. 2014.

[3] ARM. *ARM Architecture Reference Manual*.

[4] P. P. K. Chiu and S. T. K. Fu. A generative approach to universal cross assembler design. *SIGPLAN Not.*, 25(1):43–51, January 1990.

[5] Simon Cook. How to implement an llvm assembler.

[6] Ian Cutress. Intels tick-tock seemingly dead, becomes process-architecture-optimization.

[7] Richard Barker Dai Clegg. *Case Method Fast-Track: A RAD Approach.* 2004.

[8] Dr Dobbs. Simple generic assembler.

[9] Bruce Eckele. Python 3 metaprogramming.

[10] Daniel J. Ellard. *The MIPS R2000 Instruction Set*. 1990.

[11] Clark C Evans. Yaml.

[12] Clark C Evans. Yaml specification.

[13] Python Software Foundation. The python language reference.

[14] James Gosling, Bill Joy, Guy L Steele Jr, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley Professional, 2014.

[15] Richard Grisenthwaite. Armv8 technology preview.

[16] Clio Cardoso Guimares. Intel 8086 addressing modes.

[17] Graham Hamilton. Javabeans. Technical report, Sun Microsystems, 1996.

[18] Hadi Hariri. Mixing java and kotlin in one project.

[19] Paul Hohensee. The hotspot java virtual machine.

[20] David House. Moore's law to roll on for another decade.

[21] Intel. *8086 Relocatable Object Module Formats*, 1981.

[22] Intel. *6-BIT HMOS MICROPROCESSOR*, 10 1990.

[23] Jetbrains. Exceptions.

[24] Jetbrains. Intellij.

[25] Jetbrains. Kotlin.

[26] Jetbrains. Kotlin comanion objects.

[27] Jython. Jython and java integration.

[28] Laszlo B Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3):144–149, 2002.

[29] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, 1986.

[30] MicroProcessor Engineering Limited. Cross-32 universal assembler.

[31] MicroProcessor Engineering Limited. Cross-32 universal assembler - datasheet.

[32] LLVM. Llvm tablegen.

[33] LLVM. Llvm tablegen deficiencies.

[34] Simon Marlow et al. Haskell 2010 language report.

[35] Bernd Mathiske, Doug Simon, and Dave Ungar. The project maxwell assembler system. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, PPPJ '06, pages 3–12, New York, NY, USA, 2006. ACM.

[36] Inc Mill Computing. Mill computing - concepts.

[37] GORDON E MOORE. Cramming more components onto integrated circuits. *PROCEEDINGS OF THE IEEE*, 86(1), 1998.

[38] University of Regina. Arm addressing modes.

[39] Markus Perrson. 0x10c.

[40] Markus Perrson. Dcpu-16.

[41] Tim Peters. Zen of python.

[42] Kenneth Pugh. *Interface-Oriented Design*.

[43] Esoteric Software. Yamlbeans.

[44] Al Williams. Axasm.

[45] Christian Wimmer and Thomas Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM.

[46] Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating iterators in optimizing ast interpreters. In *ACM SIGPLAN Notices*, volume 49, pages 727–743. ACM, 2014.