Leo De Silva

A Level Computer Science

# DESIGNING & MAKING THE SOFTWARE SUITE

for a proprietary machine code specification.

*2024, St Albans School*

# Contents

# 1 Analysis

## 1.1 Problem Defenition

The goal of this project is to design and simulate a custom CPU, requiring a suite of tools to emulate and write programs for this processor, including an Emulator (or Virtual Machine) (1.2.2), Assembler (1.2.3), and Compiler (1.2.4). The project will detail the abstract design of the computer's Instruction Set Architecture (ISA) (1.2.1) considering the internal registers, system clock, main memory, and fetch execute cycle.

The project will compose three primary parts, an emulator capable of loading machine code 'catridges' and simulating the hardware behaviour required to execute them with correct clock timing and behaviour. An assembler to translate programs written in an assembly language into binary machine code. And finally a compiler - to translate a higher level programming langauge into machine code. The compiler will require compiler optimisations in the produced object code; data structures such as arrays, objects and strings; conditional and iterative expressions; and finally functions and procedures. All together, the processor and suite surrounding it should be capable of writing and compiling complex programs such as pong or tetris, and emulating them with hardware correct timings - dealing with I/O peripherals such as a keyboard or speaker.

## 1.2 Background to the Problem Area

I have a curiosity around the lower level elements of software development, and this project will help me understand how the everyday langauges I use to program are implemented from the processor level upwards. It will look in detail at the fundemental architecture of modern computing systems and how they are developed, looking in particular at the process of designing a processor and machine code specification with an assembler and compiler to write programs for this computer. Below I will perform some initial research into what these 4 components of the system would entail:

### 1.2.1 Instruction Set Architecture

The ISA acts as an interface between the hardware and software of a computing system, it contains crucial information regarding the capabilities of a processor, including: a functional defenition of storage locations (e.g. registers and memories) as well as a description of all instructions and operations supported. An important consideration will be whether to design an 8 or 16 bit system, 16-bits allows for more complex operations to be executed in a single cycle since more bits can be processed by the CPU simultaneously, however an 8 bit system is simpler to design and emulate since considerations like whether to use little or big endian encodings can be ignored (whether to store the most significant byte of a 16-bit integer before or after the least significant).

An ISA can be classified according to its architectural comlpexity into a Complex Instruction Set Computer (CISC), or a Reduced Instruction Set Computer (RISC). A CISC processor implements a wide variety of specialized instructions in hardware (e.g. floating point arithmetic or transferring multiple registers to or from the stack), minimising the number of instructions per program at the cost of a more complex design, higher power consumption and slower execution as each instruction requires more processor cycles to complete. Joshi (2024) This is historically the most common branch of processor and often results in large instruction sets such as Intel x86's 1503 defined instructions Giesen (2016). A RISC processor however aims to simplify hardware using an instruction set consisting of a few basic instructions to load, evaluate and store data. This has the side effect of increased memory usage to store the additional instructions needed to perform the complex tasks not implemented in hardware.

### 1.2.2 Emulator

An emulator is a software program that allows the host computer to imitate the hardware of the target machine. It reads machine code instructions assembled for the target computer sequentially from memory and interprets them, mimicking the internal state of the target machine in the process, Morlan (2019a). Emulators consist of three modules, a CPU emulator, memory subsystem, and I/O device emulators, RetroReversing (2022). The simplest form of CPU emulator is an interpreter - wherein the emulator steps sequentially through each machine code instruction, and carries out the fetch-decode-execute cycle, modifying the internal state of the simulated processor in much the same manner the instruction would affect the physical hardware. The Memory Subsytem is a one dimensional array of bytes that can be addressed through the same interface as RAM, regions of memory are allocated to peripherals and subsystems, e.g. Video Random Access Memory (VRAM), the stack, and the heap. Finally, I/O device emulators translate the input from your keyboard into device specific command signals that the processor can interface with.

### 1.2.3 Assembler

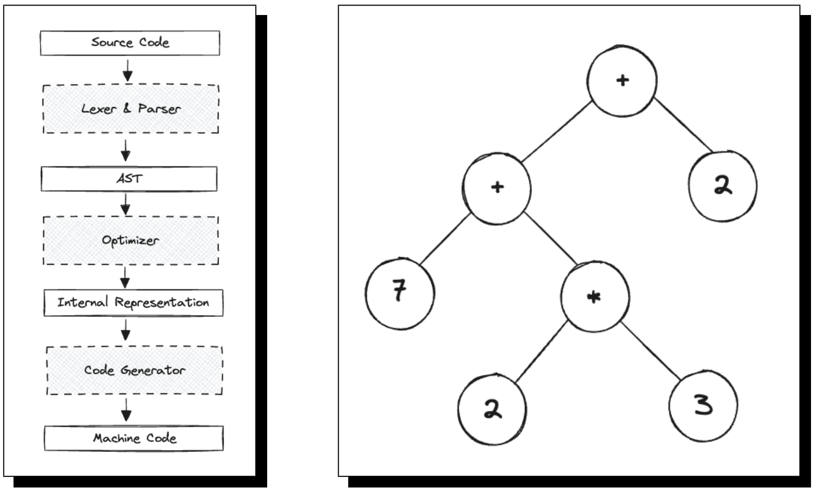
An assembler is a program that translates assembly language (a low level programming language that uses mneumonics to directly represent machine code instructions) into object code that can be executed by the processor. There are 2 types of assembler design, single-pass and multi-pass Toppr (2019). A single-pass assembler scans the source code only once to translate it into machine code, and outputs the result directly. This is the simpler type of assembler, and has faster translation speeds. However, it requires all symbols used within the program (variables, labels, etc...) to be declared before they are used - else the program will crash.

A multi-pass assembler scans the source code multiple times, on the first pass it defines a symbol and opcode table (mapping instructions and variables to their memory address which can be queried by the assembler when calculating offsets) Toppr (2019), processes pseudo instructions (compound macro instructions that are substituted during assembly with a list of fundemental instructions performing that complex task), and maintaining a location counter to store the memory address of each instruction as it would be compiled.

There are also certain abstractions a high-level assembler can translate such as `IF/THEN/ELSE/WHILE` statements and certain higher level data types (such as strings or arrays) – however this results in a complex assembler with lengthy compilation times - as well as a blurred line between the role of high level and low level languages.

### 1.2.4 Compiler

A compiler is a program that translates high level program source code into a set of machine language instructions. Some compilers translate source code into an intermediate assembly language before using an assembler to produce the machine code instructions, whereas others




(a) Compiler Pipeline

(b) Abstract Syntax Tree

compile into machine code directly. The typical pipeline to any compiler is depicted in fig. 1a Ball (2020).

A compiler is composed of three parts working in unison, Lexical analysis, Parsing, and Code Generation. The ASCII source code is tokenised by the lexer - meaning it is broken down into a list of its fundemental elements (e.g. strings, integers, keywords), and fed into the Parser where it is transformed into an Abstract Syntax Tree (AST) representing the structure of the program.

The AST is a means of breaking down the program, its statements, and order of operations into a tree representation that is easier to be processed and traversed by an algorithm. The AST representing expression ($\frac{7+2\times3}{2}$) is depicted in fig. 1b.

The optimizer may convert the AST into an Internal Representation (IR) (be that binary, textual, or another syntax tree) which is another means of representing the data in a form that lends itself better to optimisations and translation into the target language than the AST. From this new IR, optimisations may include eliminating dead code, precalculating simple arithmetic, and numerous other optimisations Ball (2020). Finally, the code generator generates the optimised code in the target language (compilation) and stores it as a file on the user's computer.

#### 1.2.4.1 Lexer

The first component of a compiler, the lexer steps through the ASCII source code character by character and builds up tokens representing the basic elements of a program such as a String, Integer, Identifier, Keyword. For example the program: `print("Result: ", (answer+1)/2)` would be tokenised as:

```
IDENTIFIER("print"), LPAREN, String("Result: "), COMMA,
    ↪ LPAREN, IDENTIFIER("answer"), ADD, INTEGER(1), RPAREN,
    ↪ DIV, INTEGER(2), RPAREN
```

This process of tokenising the program string into a series of objects makes it easier to parse into an AST and for the parser to step through by element rather than character.

#### 1.2.4.2 Parser

The process of converting the list of tokens representing the program generated by the Lexer into a tree representation (AST) that reflects the order of operations and sequence of statements is called Parsing. And is carried out by a Parser. There are two classifications of parsing algorithms, a top down parser and a bottom up parser.

A top-down parser builds its syntax tree from the root node, or highest level expressions (arithmetic operations, selective or iterative statements) and works its way down into the atomic (or leaf) nodes of the graph (individual numbers or variables). A bottom-up parser however begins with an atom such as an integer and continues to scan the source code - building up a picture of the syntax tree. For example, should the parser encounter an integer, it would continue scanning and were the next character to be an operation - the parser would know the statement must be an infix arithmetic operation. It can then transpose the graph into one representing a statement in that form (ie a root node with two children nodes for the left and right hand side of the operation). Repeating this process throughout the file builds up a syntax tree representing the program as a whole.

#### 1.2.4.3 Optimization & Code Generation

Code generation is the process of converting the AST generated from the Parser into an intermediate language which itself can be compiled down to an executable or interpreted by a virtual machine. For my NEA, the compiler will first compile down into assembly language - which will be assembled into the executable machine code - simplifying compilation through the available higher level functionality such as labels and offsets. Each higher level statement typically templates onto a standard sequence of machine code instructions, for example a program to add 2 numbers:

```
let a = 9;
let b = 5;
let c = a + b;
```

```
ldi R0, 9
ldi R1, 5
add R2, R0, R1
```

Compilations such as these involve the mapping of a potentially infinite number of variables onto a discrete number of registers, and this can be performed using such algorithms as the Linear Scan or Chatins' algorithm Geeks (2020) that take into account variable lifetimes and interactions (when the variable is in scope and when it can be freed from memory to reduce register usage). Offsets required for branch instructions that may be used in iterative or selective statements can be calculated by counting the number of instructions compiled up to the point of a particular statement (e.g. the number of machine code instructions up the the condition of a while loop) and this can be used as either an absolute or relative offset depending on the capabilities of the assembler.

## 1.3 Existing Systems

### 1.3.1 University of Washington MIPS Computer

The following system is a 16 bit MISC (Minimal Instruction Set Computer) processor designed by the University of Washington for a series of lectures as part of their computer science course Washington (2018), I will discuss its ISA and machine code encoding - in order to aid my design of an appropriate and efficient computer architecture. A MISC processsor is a subclass of the RISC processor and involves minimising the number of instructions implemented in hardware, resulting in far simpler hardware designs - where a RISC processor may have 30-70 instructions, a MISC processor may have 10-20 consisting of arithmetic, branching, loading and storing instructions. Engineering (2015).

A MIPS (Microprocessor without Interlocked Pipelined Stages) processor such as this does not overlap the execution of several instructions (pipelining), thus neglecting the potential performance gains in favor of a simpler architecture. This processor is a single-cycle implementation meaning all instructions take exactly one cycle to complete, & is achieved using a Harvard architecture in place of Von Neuman wherein instructions are stored in a seperate Read Only Memory (ROM) to data, thus both can be fetched within the same processor cycle (since a different bus is used to transfer data and instructions they can be fetched simultaneously).

The processor supports the following instructions:

1. Arithmetic `add, sub, and, or, slt (set if less than)`
2. Data Transfer `lw (load word), sw (store word)`
3. Control `beq (branch if equal to)`

Register-to-Register arithmetic instructions use the R-type encoding for their machine code representation, where `op` is the opcode of the instruction, `func` the control bits for that particular arithmetic operation, and `rs, rt`, and `rd` being the two source and destination registers respectively. This computer operates on an ALU with a 3 bit control signal supporting 5 operations that directly correspond to the `func` portion of an R type instructions binary encoding.

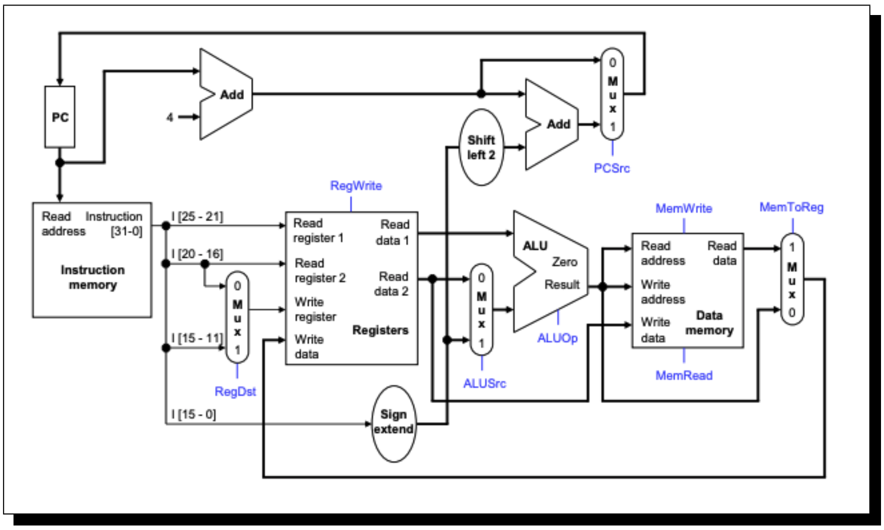
| op | rs | rt | rd | shamt | func |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

The I type encoding is the second means for which instructions can be represented, and includes the data transfer and control instructions `lw, sw`, and `beq` specified above. `address` is a signed 16 bit constant. `rt` is the destination for `lw` and source for `beq` and `sw`. `rs` is the base register for the `lw` and `sw` instructions (added to the signed constant `address` to get a data memory address) Washington (2018). In this processor design, in a `beq` instruction, the `address` field specifies not a memory address, but a signed offset from which to jump from the current PC position when executing the branch instruction.

| op | rs | rt | address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

Below is the full datapath specification for the computer, with the Instruction Memory (ROM) on the left, connected to the PC in order to address instructions. Those instructions are in turn passed through the control unit and decoded, with the opcode specifying whether an I or R type instruction is being processed and accordingly what hardware should be used to interpret and execute the instruction. This dictates the calculation (if any) that is to be performed in the ALU - the output of which is stored in a seperate data memory.

Since instructions are stored in a seperate ROM, the address of the first instruction will always begin at 0 - this simplifies the calculation of offsets and labels in the assembler – since the assumption that the first instruction begins at address 0 will always hold true. However, branch instructions are handled unusually in this computer - instead of specifying the jump address, the signed offset from the current instruction is specified instead. This has the effect of making compilation easier as branch addresses do not need to be calculated by the assembler, however renders specific jumps to memory addresses (such as the location of an interrupt service routine or bootloader) difficult.




### 1.3.1.1 Advantages & Disadvantages

The architecture described above has some notable advantages, firstly, its Harvard architecture allows the processor to operate each instruction in a single cycle – both improving performance and simplifying the design of the emulator as microinstruction cycles do not need to be simulated to accurately simulate the hardware. Secondly, by dividing the computer architecture into 2 distinct I and R type instructions, you can reduce redundant information – and thus the bits required to store machine code instructions and programs.

However, this simple architecture results in many inconveniences when writing assembly code - due to the limited instruction set, simple tasks take comparatively more instructions meaning programs are longer and more tedious to write - as well utilising more memory due to the limited number of specialized instructions who's functionality must be implemented using handwritten subroutines such as binary shifts, stack operations, or interrupt handling.

#### 1.3.1.2 Takeaways

The takeaways of this system for my project include:

1. I will consider using a Harvard architecture for my computer since all instructions can be single-cycle improving processor performance and simplifying design and emulation.
2. The encoding of instructions into meaningful machine code that directly relates to the hardware of the computer - for instance R type opcodes representing the control bits of the ALU, this makes decoding instructions more efficient - especially when implemented in hardware.
3. Secondly, the behaviour of hardware (registers, memories, flags) and the relationships between components during a single-cycle Harvard fetch-execute cycle that will have to be simulated when designing an emulator.
4. I will also expand the instruction set further than the MISC specification used in this processor to include other common instructions, and keep the memory-register seperation wherin operations are performed on register values, with 2 instructions `lw, sw` used for reading and writing to memory in order to design a more user-friendly instruction set.
5. I will also change the branch instruction to operate on absolute addresses rather than signed offsets since it offers more consistent and easily debuggable behaviour.

### 1.3.2 The Hack Computer

The Hack computer is a theoretical 16 bit computer designed by Noam Nisan and Shimon Schocken and described in their book The Elements of Modern Computing Systems Noam Nissan (2020), I will analyse its method of encoding assembly instructions into machine code - as well as the syntax of its assembly language to inform my assembler design and machine code specification. The Hack computer contains 2 16-bit registers labelled A and D, the D (data) register is a general purpose register that always acts as 1 of the 2 inputs to the ALU. Wheras the A (address) register has 2 functions: a second signed integer value for ALU operations, and a target address in instruction memory or data memory addressing. The pseudo-M (memory) register is not implemented in hardware - rather refers to the word in RAM addressed by the A register and therefore can be used to directly interact and perform calculations with memory.

```
A type: 0aaaaaaaaaaaaaaa
C type: 111accccccdddjjj
```

Hack takes a unique aproach to ISA design through its address instructions (A-type) and computational instructions (C-type). The first bit of any machine code instruction determines its type. For an A instruction - the latter 15 bits store the data (or address) as which to set the A register (`a`).

For a C-type instruction the the first 2 bits of the 15 bit operand remain unused and set to 1 by standard, this is followed by the 1 bit addressing mode (`a`) which determines whether A or M is used as the ALU's second input. Then, the computation specification

(c) composes the next 5 bits, and dictate which operation the ALU will perform, directly mapping to the ALU's control bits Noam Nissan (2020). Following, the 3 bit destination specifier (d) in turn relate to the 3 'registers' A, D, and M. Should their corresponding bit be set the ALU output will be stored in the A, D or M registers (potentially multiple). The final 3 bits describe the jump condition - each Hack C type instruction is terminated by a branch (which can be left blank). They relate to the Less Than, Equal to, or Greater Than conditions respectively, and a combination can be used to form all conditionals.

*A*-instruction: Symbolic: @*xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)
Binary: 0 *vvvvvvvvvvvvvvv* (*vv* ... *v* = 15-bit value of *xxx*)

*C*-instruction: Symbolic: *dest* = *comp* ; *jump* (*comp* is mandatory. If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted)
Binary: 111*acccccdddjjj*

| *comp* (*a* == 0) | *comp* (*a* == 1) | *c* | *c* | *c* | *c* | *c* | *c* |
|---|---|---|---|---|---|---|---|
| 0 | | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | | 1 | 1 | 1 | 1 | 1 | 1 |
| -1 | | 1 | 1 | 1 | 0 | 1 | 0 |
| D | | 0 | 0 | 1 | 1 | 0 | 0 |
| A | M | 1 | 1 | 0 | 0 | 0 | 0 |
| !D | | 0 | 0 | 1 | 1 | 0 | 1 |
| !A | !M | 1 | 1 | 0 | 0 | 0 | 1 |
| -D | | 0 | 0 | 1 | 1 | 1 | 1 |
| -A | -M | 1 | 1 | 0 | 0 | 1 | 1 |
| D+1 | | 0 | 1 | 1 | 1 | 1 | 1 |
| A+1 | M+1 | 1 | 1 | 0 | 1 | 1 | 1 |
| D-1 | | 0 | 0 | 1 | 1 | 1 | 0 |
| A-1 | M-1 | 1 | 1 | 0 | 0 | 1 | 0 |
| D+A | D+M | 0 | 0 | 0 | 0 | 1 | 0 |
| D-A | D-M | 0 | 1 | 0 | 0 | 1 | 1 |
| A-D | M-D | 0 | 0 | 0 | 1 | 1 | 1 |
| D&A | D&M | 0 | 0 | 0 | 0 | 0 | 0 |
| D\|A | D\|M | 0 | 1 | 0 | 1 | 0 | 1 |

| *dest* | *d* | *d* | *d* | Effect: store *comp* in: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | the value is not stored |
| M | 0 | 0 | 1 | RAM[A] |
| D | 0 | 1 | 0 | D register (reg) |
| DM | 0 | 1 | 1 | D reg and RAM[A] |
| A | 1 | 0 | 0 | A reg |
| AM | 1 | 0 | 1 | A reg and RAM[A] |
| AD | 1 | 1 | 0 | A reg and D reg |
| ADM | 1 | 1 | 1 | A reg, D reg, and RAM[A] |

| *jump* | *j* | *j* | *j* | Effect: |
|---|---|---|---|---|
| null | 0 | 0 | 0 | no jump |
| JGT | 0 | 0 | 1 | if *comp* > 0 jump |
| JEQ | 0 | 1 | 0 | if *comp* = 0 jump |
| JGE | 0 | 1 | 1 | if *comp* ≥ 0 jump |
| JLT | 1 | 0 | 0 | if *comp* < 0 jump |
| JNE | 1 | 0 | 1 | if *comp* ≠ 0 jump |
| JLE | 1 | 1 | 0 | if *comp* ≤ 0 jump |
| JMP | 1 | 1 | 1 | unconditional jump |

#### 1.3.2.1 Advantages & Disadvantages

This approach to ISA design being so fundementally related to the internal operations of the CPU comes with some advantages and disadvantages. Firstly, it is a very efficient design - allowing all essential operations to be carried out with a simple computer architecture. This simplicity makes it an ideal compilation target. However, the Hack assembly syntax can be unintuitive to write and understand - especially in relation to its approach to consolidating Harvard architecture and simultaneous addressing of both instructions and data from ROM or RAM respsectively. An example Hack assembly program to count to 10 might look as follows:

```
// i = 1
@i
M = 1

(LOOP)
    // if (i > 10) goto STOP
    @i
    D = M

```

```
    @10
    D = D - A

    @STOP
    D; JGT

    // i += 1
    @i
    M = M + 1

    // goto LOOP
    @LOOP
    0;JMP
(STOP)
@END
0; JMP
```

Hack's approach to assembly is also worth considering, It uses parenthesis to specify labels (points in the code from which instructions can branch to without specifying a numeric offset). The '@' character is used to specify an A type instruction - however using an identifier as the operand is a high level assembler abstraction that at compile time replaces all occurrences of the identifier with a calculated memory address representing that variable. All C-type instructions are in the form <destination(s)> = (<destination> <operation> <destination>)? (; <branch>)? where the branch expression components of the instruction are optional.

To compile this down into machine code (once labels have been replaced with offsets) - the A instruction is simply the 15 bit operand. The C instruction however is more involved. A lookup table is used to map the operations (+, -, /, *, !, &) into 5 bit opcodes (with the first bit of the 6 bit computation specified determined by whether the A or M registers are included in the operands). Then the bit corresponding to each destination specified will be set, and finally the conditional branch bits will be set depending on the mneumonic used, e.g. JGE would be replaced by 011. Together, the instruction D = D - A; JNE would be represented by the binary 111 010011 010 101.

#### 1.3.2.2 Takeaways

From this case study, there are a number of takeaways:

1. Breakdown instructions into types capable of representing a family of assembly insstructions - reducing the number of machine code instructions required to be implemented by the virtual machine (emulator).

2. I will maintain a comparatively small instruction set, relying on macro instructions (compound instructions that are substituted at compile time for a list of fundemental ones carrying out that defined task) instead, to simplify the assembly syntax and encoding of instructions into machine code.

3. Use a pseudo-register to represent the addressing behaviour of a Harvard architecture computer, simplifying operations involving memory access & compilation behaviour.
4. Represent branch conditionals through 3 bits reflecting `<, =, >` comparisons
5. Use one bit to represent each destination register allowing for a combination of destinations for a paricular instruction meaning seperate instructions need not be created for storing data in memory or registers.

### 1.3.3 Monkey

Monkey is the programming language described in Thorsten Ball's book Writing a Compiler in Go Ball (2020), I will be analysing the syntax of the language to inform my high-level language design. Monkey has a C-like syntax, variables, integers and booleans, arithmetic expressions, first class functions (functions that can be passed to other functions as parameters), strings, and arrays. Its syntax looks as follows (illustrated with an example program to calculate the nth fibonacci number):

```
let fibonacci = fn(x) {
    if (x == 0) {
        0;
    } else {
        if (x == 1) {
            1;
        } else {
            fibonacci(x - 1) + fibonacci(x - 2);
        };
    };
}

let main = fn() {
    let numbers = [1, 2, 10, 50, 9*18];
    let index = 0;

    while (index < length(numbers)) {
        print(fibonacci(numbers[index]));
        index = index + 1;
    }
}
```

#### 1.3.3.1 Advantages & Disadvantages

There are some advantages with this approach to language design, for instance its syntax lends itself to a simple and convenient to program parser, in particular, by representing functions as variables it allows you to pass functions as parameters (first order functions) without any additional logic validating return types or parameters. However, this functionality is difficult to implement in machine code. Instead, passing the address of the first instruction of the function, rather than the function itself is a more practical solution for a compiled language. References and pointers are also not present in Monkey, these permit complex functionality such as arrays and strings, whilst maintaining a simple compiler since programmers can access variables by their location in memory rather than through an identifier (providing the ability to traverse an array through consecutive memory locations for example). However, this can lead to code that is difficult to understand and takes familiarity with the hardware & implementation of the language to write.

Monkey represents variables using Go's built-in data structures, thus doesn't have to compile them into binary - meaning specifying a data type is less important, and the language can afford to be dynamically typed - this means variable types are not checked when compiling expressions, and can result in runtime errors when attempting to add an integer to a string, or assign an integer to a float type variable. Using the `let` keyword to define a variable as above(unlike python) is vital for a compiled language - since additional functionality is required to allocate a memory address (or regiser) when declaring a variable depending on its lifetime.

#### 1.3.3.2 Implementation

I will also look at the implementation of this language, in particular its Lexer and Parser as these are directly relevant to my NEA. Firstly, the Lexer. Monkey represents tokens as all deriving from an abstract class (a class to be inherrited not instantiated) `Token` defined below.

```
enum TokenType {
    LPAREN,
    RPAREN,

    STRING,
    IDENTIFIER,
    INTEGER,
    ...
}

type Token struct {
    enum TokenType
    Literal String
}
```

The code is scanned character by character and the fundemental elements of the program are stored in these token objects, for instance the string `"Hello, World!"` would be stored as `Token(TokenType::String, "Hello, World")`. A list of these token objects are returned by the lexer and used as input to the parser.

The Monkey interpreter's parser uses a top-down Pratt parser as opposed to the more common bottom-up parser. Top-down parsers are simpler and more elegant to write due to their highly recursive nature - however this can make them troublesome to debug and maintain. They avoid much of the complex graph transpositions required for a bottom up parser.

#### 1.3.3.3 Takeaways

The takeaways from this system include:

1. Using established programming language norms for defining variables, iterative statements and functions will make the programming language easier to learn due to transferable experience.

2. Designing a statically typed language would reduce program crashes and lead to a more robust compiler and programs.

3. Including references and pointers allow for the implementation of features such as arrays and strings whilst maintaining a concise and simple compiler.

4. I should consider defining variables with the 'let' keyword to tell the compiler it needs to insert additional logic calculating an appropriate memory address in which to store the variable, and store that address in a lookup table against its identifier.

5. I should consider writing a top-down parser as opposed to a bottom up parser to ensure the code is cleaner, simpler and more elegant.

### 1.3.4 Jack

Jack is the high level langauge defined in book The Elements of Modern Computing Systems Noam Nissan (2020) with a syntax similar to Java. I will analyse its syntax and how it is compiled into machine code to inform my design of a compiled language. Jack is an Object-Oriented statically typed language (programs organised around objects rather than functions, and that requires the specification of a variables data-type when it is declared) similar to Java that is compiled down into the Hack machine code specification. An example Jack program may look as follows (demonstrated with a program to print the elements in a linked list) Noam Nissan (2020):

```
class List {
    // declare the class attributes
    field int data;
    field List next;

    // define a constructor to initialise a List with attributes data and next
    constructor List new (int dataParam, List nextParam) {
        let data = dataParam;
        let next = nextParam;
        return this;
    }

    method int getData() { return data; }
    method List getNext() { return next; }

```

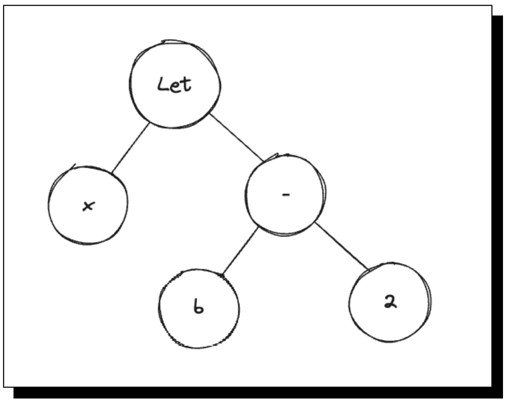
```
    method void print() {
        // declare a pointer to the first element of the list
        var List current;
        let current = this;

        // iterate through all the elements in the linked list
        while (~(current = null)) {
            do Output.printInt(current.getData());
            do Output.printChar(32) // space
            let current = current.getNext();
        }

        return;
    }
}
```

Above is the example program to define a linked list in Jack as provided in the book, and shows the similarities and differences to other popular languages. Jack has program structure very similar to that of Java, or C# - relying on a series of classes containing program logic which can be invoked using the `do` keword. Jack splits the functionality of certain keywords in typical programming languages into more specialized roles: for instance the `field` keyword used to define object attributes, the `let` keyword being used every time when assigning to variables, the `method` and `constructor` keywords typically under the umbrella of `function`, and the `do` keyword used to invoke methods. This can introduce a steeper learning curve when learning Jack and adds potentially unecessary complexity.

However the reason for differentiating field variables and regular variables, for instance, is due to their lifetimes. A copy of field variables needs to be maintained for each instance of a particular class - wheras other local variables can be freed from memory once their subroutine terminates and they are no longer used.

#### 1.3.4.1 Implementation

The code generation in the Jack language involves scanning through the Abstract syntax tree and for each Node type (e.g. Infix, Selection, Iterative) appending a template of (optimized) assembly language instructions into an array which can then be compiled down into its ASCII representation. A typical example of such compilation would be through the compilation of the statement `let x = b - 2`. The AST for this is below, and is the data structure that would be passed to the code generator:



From this, the code generator would traverse the graph using pre-order traversal, recursively calling the `compile` method on each node, for instance, the compile method would be called on the parent root node, which would recursively call the same method on each of its child nodes. During compilation of the RHS node, it would also recursively compile its child nodes - until an assembly representation of the program is built up. The compile method generates a list of assembly language instructions which perform the behaviour specified for that particular operation. The assembly generated for this program would look as follows:

```
// x variable is mapped to memory address $01
ldi a, 2
sub b, a
sw  $01, b
```

Depending on the number of working variables in memory, the register x may be assigned to one of the general purpose registers instead.

The unique method in which selection statements are compiled down into assembly code in the Jack compiler is useful to analyse due to the convenience it offers when writing a compiler - namely it allows you to ignore calculating offsets by taking full advantage of the higher level features of the assembler (a luxury afforded due to the two step compilation process). To compile selection statements in the Jack Compiler, the compiler generates a series of arbitrary labels e.g. (L1, L2) and places these after key points in the selective process in order to avoid offset calculating - a functionality that can be handled by the assembler. To compile the following:

```
if (b > 10) {
    c = b;
} elif (b % 2 == 0) {
    b = b + 1;
} else {
    b = b - 1;
}
```

The compiler will insert labels before the first instruction of each branch, and insert any code for the unconditional 'else' block after the jump instructions for any conditions (elif, and then branches). This approach avoids calculating any offsets and thus only a single pass is required to compile this program.

```
// if b > 10 goto .then
ldi a, 10
sub a, b, a
bgt .then

// elif b % 2 == 0 goto .elif
```

```
ldi a, 2
mod b, 2
beq .elif

// b = b - 1
lda a, 1
sub b, a
goto end

.then
    // c = b
    mov c, b
    goto end
.elif
    // b = b + 1
    ldi a, 1
    add b, a
.end
```

#### 1.3.4.2 Advantages & Disadvantages

The advantages of the Jack programming language include its specific keywords that offer insight into the manner in which its features are implemented - removing some of the abstraction typical higher level langauges offer. Another advantage is its type system, resulting in robust programs and reducing the edge cases a compiler would have to deal with. If an incorrect type was passed to a function or operation, an error would be thrown at compile time and no such error could occur in the compiled machine code.

However, the disadvantages of the Jack language include its Object Oriented approach making compilation difficult. Attribute variables on different instances of classes will have different lifetimes and therefore freeing the finite number of registers the computer offers to make space for newly declared variables becomes much harder a task. Secondly, Jack uses many unecessary keywords, for instance the `do` keyword functioning as an abstraction for calling a method and ignoring its return value, and the `let` keyword being required every time you assign a variable rather than for its declaration alone. This means declarations in Jack are required to be sepereate statements, increasing the volume of code required to perform the same task.

#### 1.3.4.3 Takeaways

The takeaways from this language include:

1. Use a procedural aproach to program structure rather than an object oriented one since it leaves the flexibility of program structure in the hands of the programmer.

2. Limit the number of keywords used in the final source code to only those that offer useful insight into the purpose of statements in the program.

3. Consider implementing a 2 step compilation process to take advantage of the assemblers higher level conveniences around labels and offsets.

4. Use a simplified statically typed type system closer to that of Java or Go rather than Rust or C since it is less cumbersome and more intuitive to use.

5. Compile selection and iterative statements using generated labels rather than calculated offsets to greater more robust and less error prone code.

### 1.3.5 Austin Morlan's CHIP-8 Emulator

CHIP-8 is a specification for a fictitious computer designed to provide an easy entry point into developing emulators, intended as a stepping stone before approaching more complex systems. I will analyse Austin Morlan's CHIP-8 emulator, Morlan (2019b), and discuss the manner in which he has realised the internal state of the computer through code and how this can be applied to my system.

First I will discuss the actual hardware of the CHIP-8 computer itself, in order to provide a background when discussing its implementation in code. The CHIP-8 system is an 8-bit general purpose, Von Neuman computer. It has 16 general purpose registers labelled `V0-VF` which can hold values ranging from `0x00-0xFF`, Morlan (2019a). The `VF` flag is called the flag register, and its bits are set or unset depending on the result of calculations. For example, were the result of a calculation to be negative, the corresponding negative bit in the `VF` flag would be set.

CHIP-8 contains 4096 bytes of memory (from `0x000` to `0xFFF`) subdivided as follows: `0x000-0x1FF` contains the bootloader (a program to initialise the computer's state and begin execution of general purpose programs), `0x040-0x0A0` contains the computers character set (binary data containing the pixel representations of ASCII characters), and the rest of the memory is used to store instructions and data respectively.

CHIP-8 also contains a number of special purpose internal registers including a 16 bit Index register (I) used to store memory addresses for use in operations, and a 16 bit Program Counter (PC) that holds the address of the next instruction to execute, Morlan (2019a). There is also an 8-bit Delay timer that decrements its value when non-zero - used to regularise time intervals between frames when writing games, and an 8-Bit Sound Timer that emits a buzz when its value is non-zero.

The CHIP-8 computer contains a 16-bit address stack of depth 16 referred to by an 8-bit stack pointer (SP) which keeps track of the most recent value pushed onto the stack. Whenever a `call` instruction is executed, the current value of the PC is pushed onto the top of the stack and SP incremented to point to this new value. Correpondingly, when a `ret` instruction is executed, the top value is popped off of the stack and set as the new value for the PC, causing program execution to resume after the `call` instruction.

Finally, CHIP-8 has memory-mapped I/O (where the input or output of peripherals are stored in main memory). For instance, 16 bits are used to represent the 16 keys of the CHIP-8 system with a 0 or 1 rerpesenting whether a key is held down. 2KB are used to store the 32*64 black and white monochrome display - with one bit per pixel.

Austin represented this internal state through the following class definition (shown with the respective variable names and data types):

```
#define CHARSET_ADDRESS  0x50
#define START_ADDRESS    0x200
#define MEMORY_SIZE      4096;
#define VIDEO_HEIGHT     32;
#define VIDEO_WIDTH      64;

class Chip8 {
    public:
        uint8_t registers[16];
        uint8_t memory[4096];
        uint16_t index;
        uint16_t pc;
        uint16_t stack[16];
        uint8_t sp;
        uint8_t delayTimer;
        uint8_t soundTimer;
        uint16_t opcode;
};
```

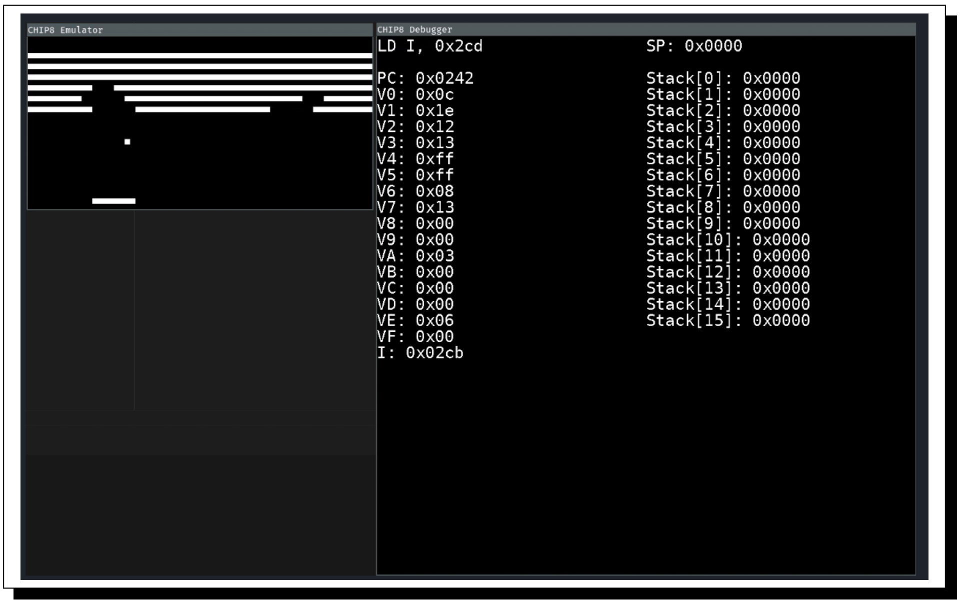
The scaffolding of Morlan's emulator revolves around an indefinite loop simulating the CPU's clock cycles, each of which contains the code to fetch, decode and execute instructions, Muller (2011). First the program fetches the 16-bit instruction from the address specified by the PC (and its following byte) and a bitmask is applied to extract the 4-bit opcode. A switch statement is then used to determine the operation and modify the internal state of the computer to carry out its behaviour accordingly by modifying register values or reading/writing to memory. Finally the timers are decremented should they be non-zero.

```
void chip8::emulateCycle() {
  // Fetch 16-bit instruction (4-bit opcode, 12-bit operand)
  instruction = memory[pc] << 8 | memory[pc + 1];

  // Decode opcode
  switch(instruction & 0xF000) {
    case 0xA000: // ANNN: Sets I to the address NNN
      I = opcode & 0x0FFF;
      pc += 2;
    break;

    [...]

    default:
      printf ("Unknown opcode: 0x%X\n", opcode);
  }

  // Update timers
```

```
19   if(delay_timer > 0)
20     --delay_timer;
21
22   if(sound_timer > 0) {
23     --sound_timer;
24   }
25 }
```

#### 1.3.5.1 User Interface

Morlan has designed the UI for his emulator with 2 distinct parts, the display on the left, and the debugger on the right. The display is a graphical representation of the contents of VRAM, consisting in this case of a 32x64 pixel monochrome display. The debugger contains a dissassembled version of the currently executing instruction represented as a mneumonic, the contents of all special purpose registers (I, PC, SP), all general purpose registers (V0-VF), and the stack. This information helps a programmer to debug their program, ensuring registers are being modified as expected.




#### 1.3.5.2 Advantages & Disadvantages

Morlan has decided to increment the PC value inside the switch statement seperately for each instruction (`emulateCycle()` line 9), reducing a potential source of bugs. If the PC is incremented automatically at the end of the `emulateCycle()` method, should the PC be modified during the execution of an instruction (e.g. branch, call or return) then automatically incrementing the PC would offset its value from that which is intended.

Morlan's debugger contains enough information to be of use to a programmer, however without a means to probe memory, lacks some of the functionality required. Furthermore, its prominent position in the UI overshadows the actual display, so a toggleable debugger would leave more room for the display itself.

Around the CHIP-8 system more generally, having memory addressable through 3 bytes

(0x000-0xFFF or 4096 distinct memory locations) frees a nibble to represent the opcode, meaning an instruction can fit in a 16 bit register - since this is the same as the word size for the CHIP-8 processor, it means the system can fetch all instructions within a single cycle, increasing efficiency.

Furthermore, using a Delay timer rather than an interrupt-request system simplifies the process of synchronising CPU operations to real world timings, e.g. when drawing frameas for a simulation at a fixed frame rate, instead of polling (listening) for requests to trigger the code to draw a frame, only executing the code to draw the frame when the delay counter is 0 would have the same effect.

#### 1.3.5.3 Takeaways

The takeaways from the CHIP-8 architecture and emulator include:

1. Incrementing the PC should be done either inside the switch statement or after fetching the instruction to avoid modifying the PC multiple times and reduce a source of bugs.

2. Design a togglable debugger that can be hidden when it is not required to prioritise space for the graphical display.

3. Using a Delay timer rather than an interrupt-request system simplifies both the hardware and software side of the comptuer, avoiding the need to define Interrupt Request Tables (mapping interrupt codes to the Interrupt Service Request (ISR) required to service them).

## 1.4 Client Proposal

My client is my uncle, a software engineer who has previously worked with lower level development. I will present him with the following proposal and assertain his thoughts on a series of questions to dictate the direction and design of my project.

My aim is to design a 16 bit processor with a RISC instruction set and the capability to execute complex programs such as tetris whilst interfacing with I/O devices such as a keyboard to handle input. I will build the suit of tools required to emulate the behaviour of such a processor. Firstly, a virtual machine, able to emulate the hardware of my processor and execute machine code programs with the correct clock timing and behaviour. This will include a simple GUI that reflects the contents of VRAM (Video RAM) allowing images and information to be communicated to the user.

Then, I will design the syntax of an assembly language that represents the machine code instructions of the processor's instruction set, and a multi-pass assembler to compile this down into binary machine code. The assembler should be able to calculate the required offsets of branch instructions from the position of labels in the source code, and potentially handle macro instruction expansions. (where pseudo-instructions represented by mneumonics can be defined - which at compile time are substituted for a list of fundemental instructions that carry out that defined task).

Finally, I will design a high level language with syntax similar to C and features such as iteration, procedures & methods, selective statements, static typing, variables, references, and pointers. The experience of programming in this language should be familiar to anyone with programming experience, however remove some of the higher level abstractions typical languages offer, providing insight into the manner in which features are implemented.

### 1.4.1 Client Interview

I will interview my client to get his views on a number of questions regarding the design of my project, including the processor design and instruction set; the assembly language syntax and machine code abstractions (macros and labels); and finally the higher level language syntax and features (OOP, first order functions, etc...).

1. **Instruction Set Architecture & Assembler**

   - Q: Do you have any low level experience?
     RFA: To determine my clients level of familiarity with my problem domain, and to target my questions towards that.
     A: *"I used assembly when writing a driver a few years back, but I've not gone much lower level than that. Although I can remember some theory from University."*
   - FU: What architecture did you use, and what were your experiences using it?
     RFA: To establish whether my client has used a RISC or a CISC architecture and inform any follow up questions that would help determine which architecture my project will use.
     A: *"I was migrating a driver in x86 to ARM so I've touched on both. x86 is definitely more powerful, but that does mean it's harder to learn because there's so many more instructions. Although once you do know it, it's really efficient to program in, and you can write complex programs relatively concisely. ARM is the opposite, its easy to understand with an obviously well thought out design that makes it nice to program in whether you're just learning or experienced."*
   - FU: Did you prefer working with a CISC (x86) or RISC (arm) Instruction Set?
     RFA: To inform whether I use a RISC or CISC architecture.
     A: *"I prefer RISC because the fewer instructions mean those that are present tend to be much more carefully thought out, it's also just easier to program without flipping back to the documentation every 5 minutes."*
   - Q: Have you ever programmed for a Harvard architecture computer?
     RFA: I want to determine whether my client is opposed to or in favour of programming for such a system.
     A: *"Most ARM processor tend to run on a Harvard architecture, although the assembly language hides that level of hardware anyway so it's not something I really have to consider."*
   - Q: What in particular did you like about the syntax of x86 or arm assembly?
     RFA: To determine what he is familiar with and thus help design the syntax of my assembly language.
     A: *"They're both quite similar aside from their register names. x86 uses eax, ebx - but arm uses r0, r1. Which I think makes more sense. I think x86 has longer menumonics as well - although that's probably because of its larger instruction set."*
   - FU: When writing assembly do you find yourself using labels or macro instructions?
     RFA: To see if my client wants me to include these higher level assembly language features in my assembler.

A: *"Labels are really important when coding, and they mean you don't have to keep recalculating offsets every time you add a new instruction. But for a RISC processor like arm - macro instructions provided by the assembler can be really helpful - they cut out a lot of the tedious programming when you write the same thing over and over."*

- Q: Would you prefer to write assembly for a 16-bit or an 8-bit system?
  RFA: To understand my client's position on the impact of the word-length of a system.
  A: *"I would prefer a 16-bit system because of the flexibility in representing large or precise numbers which you just can't do with an 8-bit system. It lets you worry less about overflow and underflow and all the quirks of binary maths."*

Takeaways:

1.1 My client favours a RISC instruction set, particularly the carefully considered instructions. I should take time when designing my instruction set to ensure there is enough breadth to cover all the desired functionality in an effective manner that is convenient to program in.

1.2 My assembly language's instructions should abstract away the quirks of the Harvard architecture's seperate data and instruction memories - meaning my client will have more transferable experience when programming in my langauge.

1.3 Registers should be named logically, either alphabetically or numerically, e.g. 'r0, r1, r2, ...' or 'a, b, c, ...'.

1.4 My client would prefer a 16-bit system.

2. **Compiler & High Level Language**

- Q: When you write code, do you prefer an Object Oriented or Procedural Style?
  RFA: To decide whether my language needs an OOP focus like Java, or a procedural approach.?
  A: *"I like the flexibility of procedural programming, even though I tend to write cleaner code when I use OOP, I think procedural is easier to pick up and code. You could do something like python where you support OOP but don't enforce it?"*
- Q: Do you prefer a simpler syntax like python, or something more like C?
  RFA: To determine which syntax style my language should use.
  A: *"I've been coding in Go for work and I like their approach. It's got the unambigious syntax of C with the flexibility in how you format your code that comes with braces and semicolons, but they've also simplified the type system so you don't have to think about integer sizes or pointers in strings."*
- FU: Would you like my language to have a simlar syntax to preexisting langauges, or to try something new?
  RFA: To see how important familiarity is to my client, and whether he'd be willing to try new ideas to see if they work.
  *"I think it's important a language is readable to someone with no experience programming in it, so I wouldn't change the format too much. But it's nice to*

*try some new things, like what go did with goroutines, or rust with the borrow checker."*

- Q: If you could design your own language, what features would be most important to you?
  RFA: To allow my client to suggest other ideas I hadn't considered that might aid the design of my language.
  A: *"I think good error messages go a long way into improving my experience with a language. They're often overlooked when you're writing a language, but for someone just learning how to code, they're make or break. I'd also say a good type system, Go and Java have pretty good approaches, although when you're writing something lower level you need that extra information about the size of your variables they just don't offer. What Rust does with it's numeric types is good, although I think their approach to strings needs refining."*

Takeaways:

2.1 My language should be a primarily procedural langauge, however offer optional elements of object oriented programming such as classes and interfaces to help organise programs.

2.2 My programming language should use the C like syntax elements of curly braces and semi colons as they offer more flexibility when formatting code. Since whitespace does not dictate control flow.

2.3 Without straying too far from the norms, I should consider alternative aproaches to syntax and langage features in order to differentiate my language, aggregating positive elements of other systems.

2.4 My compiler should produce specific and actable error messages that are actually helpful to a programmer. This may include information on how to approach correcting such an error and its location in the source code.

2.5 A strong type system is important, It should abstract the implementation details of compound data structures such as strings, however still offer the flexibility required when writing lower level programs. For example specifying an integer size or whether it is signed or unsigned.

3. **Virtual Machine**

- Q: Have you ever used a Virtual Machine before?
  RFA: To see in how much detail I can ask the follow-up questions.
  A: *"I've used a Gameboy emulator before, but I've never coded anything in one."*
- Q: What features would you expect if you were using a virtual machine to test your code?
  RFA: To see which features are the most vital to include in my emulator in order to help my client code for my computer.
  A: *"I think a good debugger is important, certainly one showing the contents of RAM, register values, and the current instruction being executed. With maybe the ability to step through a program one instruction at a time setting breakpoints."*

Takeaways:

3.1 My virtual machine should include a comprehensive debugger for testing programs, you should be able to check the internal state of the computers memory and registers to determine whether the program is functioning as intended.

This interview has affirmed that the direction in which to take my project is that of a simpler RISC processor with carefully considered instructions, relying more on macro instructions provided by the assembler to improve the development experience rather than on the hardware itself. My client also suggested a procedural language structure with syntax similar to C, a common trend with lower level languages. He also emphasised the importance of a well considered type system and error messages, so these should have careful consideration in my design section. Finally, due to the difficulty of testing machine code programs, a debugging mode in the virtual machine would greatly improve the experience of my client when writing assembly code.

## 1.5 Objectives

| Objective | Requirement | Justification | Deliverables |
|---|---|---|---|
| 1.0 ISA & Assembler | | | |
| 1.1 | A RISC (Reduced Instruction Set Computer) design philosophy. | A RISC instruction set can have better performance due to the faster and more efficient execution of it's instructions, especially when a user isn't as familiar with the instruction set of the system. (Client Interview 1.1) | |
| 1.2 | A Von Neuman computer architecture where both data and instructions share the same memory. | The shared memory between data and instructions minimises memory wastage since they can both share the same memory locations. (University of Washington MIPS Computer 1.) | |
| 1.3 | My Instruction Set should utilize 3 address operands standards. | 3 address operands is the programming standard for both arm and x86, pre-established instruction sets programemrs are already familiar with. The need to address instruction and data from different memories in a Harvard architecture is typically hidden from the programmer. (Client Interview 1.2) | Assembly instructions should take 3 operands: a destination register followed by 1-2 source registers. Programmers should interface with the branch and load instructions in the same manner when addressing instructions or data. |
| 1.4 | Branch and call instructions should calculate offsets from labels in the source code. | Labels allow programmers to write branching or selective statements without having to manually calculate memory offests. (The Hack Computer, The University of Washington MIPS Computer, Client Interview) | My assembler should replace all occurences of a label in the assembly source code with calculated offests from the current instruction to that label. |
| 1.5 | Macro instructions to perform common tasks that are not otherwise specified in the ISA. | Macro instructions minimise the need for programmers to repeat chunks of code to perform common tasks such as pushing values to the stack or calling functions. (The Hack Computer 2., Client Interview) | The assembler should substitute compound instructions such as `call` and `ret` for a list of machine code instructions that perform the same task. |

| | | | |
|---|---|---|---|
| 1.6 | A set of registers broad enough to minimse memory access. | Prioritising registers over RAM improves processor performance as calculations can be performed with reduced latency. A delay timer simplifies the process of timing CPU operations, and a memory-resident address stack simplifies the process of calling and returning from functions. (Austin Moorlan's CHIP-8 Emulator, Client Interview 1.3) | My processor should have 16 general purpose registers (r0-rF), a stack pointer (SP), program counter (PC), and delay timer (DT). |
| 1.7 | A CPU word length of 16-bits. | A 16-bit word length allow more bits to be processed in a single cycle and allows 16-bit instructions to be fetched within a single cycle improving performance. 16-bits can also represent numbers of a greater magnitude than 8-bits reducing overflow errors. (Client Interview 1.4) | Registers, ALU operations, memory locations and busses should all operate on 16-bit values. |
| 2.0 Compiler | | | |
| 2.1 | C standard syntax with semi colons and braces rather than indentation. | This makes for easier compilation and more flexibility when formatting code, as well as reporting errors such as an unterminated brace at compile time (unlike incorrect indentation which may not be detected until debugging) (Client Interview 2.2) | Lines should be terminated using a semi-colon, and curly-braces used to signify code blocks in selective or iterative expressions rather than indentation. |
| 2.2 | The programming language should be statically typed. | A type system ensures type-errors are thrown at compile time rather than during execution, leading to more robust programs that are easier to debug. Furthermore compilation becomes easier with statically typed variables as their size in memory is predetermined. (Monkey.2, Jack.4, Client Interview 2.5) | My syntax should support signed and unsigned integers, characters, booleans. The `let` keyword should be used when declaring a variable and require the type to be specified alongside its identifier. |

| | | | |
|---|---|---|---|
| 2.3 | The language should support a procedural programming paradigm. | Procedural programming is much more straightforward to compile since variable lifetimes within multiple instances and references of an object don't have to be calculated - simplifying the process of garbage collection. (Client Interview 2.1, Jack.1) | All statements should be contained within methods, with the single entry point being a compulsory `main()` method. |
| 2.4 | My language should support references and pointers to variables in memory. | Pointers allow programmers to implement arrays and strings by accessing variables through their memory location rather than an identifier. They also let you pass structs (otherwise a large data structure inefficient to pass as a copy) to a function as well as references to the first instruction of a function (allowing for first order functions) (Monkey.3) | Programmers should be able to create a pointer to a variable: (`&a`), and dereference it (`*a`). |
| 2.5 | The compiler should produce relevant error messages, pointing out the position in source code if relevant. | Relevant error messages make debugging much easier and improve the programmer's experience with a language. (Client Interview 2.4) | Error messages should include an easy to understand description of the error, its position in source code - and if possible, relevant steps to correcting it. |
| 2.6 | The language should support definite and indefinite iteration through for and while loops. | Iteration is a key element of control flow for a procedural language. | 'for' and 'while' keywords should be used for the two different types of loops. |
| 2.7 | Data should be stored in scoped variables and global constants. | Storing data in scoped variables ensures memory is efficiently managed, preventing unintended side effects from modifying global variables. Constants help improve code readability and make debugging easier. | The 'let' keyword should be used to declare variables inside a subroutine, and 'const' for declaring gloabl constants. |

| | | | |
|---|---|---|---|
| 2.8 | The compiler should support defining and calling functions. | Functions allow code to be reused, and encapsulate logic into distinct units. Furthermore, functions help with code organisation. | Programmers should be able to declare functions with typed parameters and a return type. Functions should support recursion. |
| 3.0 Virtual Machine | | | |
| 3.1 | The Virtual Machine should include a graphical display showing the contents of VRAM. | It makes programs more interactive and easily debuggable, as well as allowing programs such as simulations or games to be written for the system, expanding its capabilties. (Austin Morlan's CHIP-8 Emulator) | |
| 3.2 | The Virtual Machine should include a togglable debugger. | A debugger would help programmers locate errors and test their programs, as well as ensuring the internal state of the computer is being modified as intended. (Client Interview 3.1, Austin Morlan's CHIP-8 Emulator.2) | The debugger should show the contents of the general and special purpose registers, the currently executing instruction, and be able to probe the contents of memory. |

## 1.6 Prototyping

There are 3 main areas I am unsure how to implement and will need to explore further through prototyping:

1. The process of loading a binary machine code program into the emulator, and stepping through it instruction by instruction.
2. The data structures with which I will store Tokens and Nodes in the compiler, and from this I will develop a parser for arithmetic expressions to familiarise myself with coding a Lexer and a Parser.
3. The process of generating binary machine code from a list of objects representing assembly language instructions.

I will also use this prototyping process to help inform which language I use to code my project (the primary options being C or Rust for their low level support).

### 1.6.1 Loading & Interpreting Binary Programs

The first part of my system to prototype was the process of interpreting and decoding a binary file into a series of distinct instructions from which their behaviour can be simulated. I am going to use the CHIP-8 instruction set as a placeholder due to its simplicity, and the fact each instruction is always 2 bytes long making the process of fetching instructions easier. I will also use this to develop my knowledge of C.

Below is the code for this prototype, It takes in the filename of the ROM as a command line argument, opens the binary file and writes it to the memory array of the emulated CHIP-8 system. From there it enters an infinite loop (terminated only by the halt flag on the CPU) representing the fetch-execute cycle of the system. In each iteration, it fetches the 2 bytes of the instruction and stores it in a 16-bit unsigned integer, bitmasks the instruction to extract the opcode and then enters a case statement to act according to the opcode. For the purposes of this prototype, I just printed the name of each instruction it encounters.

```c
#include "stdio.h"
#include "stdint.h"

#define MEM_CAPACITY 4096

struct CHIP8 {
    uint8_t memory[MEM_CAPACITY];
    uint16_t pc;
    uint8_t hlt;
};

void emulate_cycle(struct CHIP8 *chip8) {
    // fetch the 2 byte instruction from memory (MSB: pc, LSB
        ↪ : pc+1) and store in a 16-bit unsigned int
    uint16_t instruction = (chip8->memory[chip8->pc] << 8) |
        ↪ chip8->memory[chip8->pc+1];
```

```
    chip8->pc += 2;

    printf("0x%04x ", chip8->pc);

    // bitmask the instruction to extract the opcode (first nibble)
    switch (instruction & 0xF000) {
        case 0x0000:
            printf("HLT");
            chip8->hlt = 1;
            break;

        case 0x1000:
            printf("JMP");
            break;

        case 0x2000:
            printf("CALL");
            break;

        case 0x3000:
            printf("SEQ");
            break;

        case 0x4000:
            printf("SNE");
            break;

        case 0x6000:
            printf("SET");
            break;

        case 0x7000:
            printf("ADD");
            break;
    }

    printf(": 0x%04x\n", instruction);
}

int main(int argc, char *argv[]) {
    // exit if user hasn't specified a ROM
    if (argc < 2) {
        printf("error: no input ROM\n");
        return 1;
    }
```

```

    // initialise CHIP8 (memory and pc) values to 0
    struct CHIP8 chip8;
    chip8.pc = 0;
    for (int i = 0; i < 4096; i++)
        chip8.memory[i] = 0;

    // read binary stream from ROM into chip-8 memory
    FILE *ptr;
    ptr = fopen(argv[1], "rb");

    fread(chip8.memory, sizeof(chip8.memory), 1, ptr);

    // simulate CPU cycles
    while(chip8.hlt != 1) {
        emulate_cycle(&chip8);
    }

    return 0;
}
```

The ROM I am using to test this program is an example on the CHIP-8 archive. `https://johnearnest.github.io/chip8Archive/`.

```
$ gcc main.c -o main && ./main "roms/Octojam 9 Title.ch8"
0x0002 SET: 0x6010
0x0004 SET: 0x620b
0x000e SNE: 0x4121
0x0010 ADD: 0x7008
0x0012 SNE: 0x4121
0x0014 SET: 0x6100
0x0016 SEQ: 0x3030
0x0018 JMP: 0x1206
0x001a CALL: 0x23e6
[...]
0x01e0 SNE: 0x4d07
0x01e2 SET: 0x6d00
0x01e4 CALL: 0x23ea
0x01e6 JMP: 0x1264
0x01e8 SET: 0x6f14
0x01ee SEQ: 0x3f00
0x01f0 JMP: 0x13ea
0x01f2 SET: 0x6f03
0x01f6 HLT: 0x00ee
```

Making this prototype exposed one vulnerability in my code and one inconvenience, I did not validate the ROM size before loading it into RAM, this could cause a buffer overflow should the ROM be larger than 4KB, and allow access to protected memory. The inconvenience however was C's default type system and the unintuitive names for variable sizes, for instance a 16-bit unsigned integer is an `unsigned short` and array of strings a `char *array[]`. This lead me to include the `stdint.h` library which offers more explicit alternatives for these names such as a `uint8_t` representing an 8 bit unsigned integer. I found this made for cleaner and more easily readable code and I will use this standard throughout my project.

### 1.6.2 Lexer

The second prototype encompased two components of the system, a Lexer and a Parser written as a subset of the final program and capable of evaluating arithmetic expressions considering the order of operations. This inital data model represents tokens as enums (rust - similarly to C, does not support typical object oriented programming paradigms, instead seperating the behaviours into enums and structs representing different behaviours of a class). I also created a SyntaxError struct which stores a single error message with the intention of expanding upon this in the final lexer to support line number and position within the source code.

```
// ==== src/token.rs ====
#[derive(PartialEq, Debug, Clone)]
pub enum Token {
    Number(u32),
    LPAREN,
    RPAREN,
    ADD,
    SUB,
    MUL,
    DIV,
    EOF,
}

#[derive(Debug)]
pub struct SyntaxError {
    pub msg: String,
}

impl SyntaxError {
    fn new(msg: String) -> Self {
        SyntaxError { msg }
    }
}
```

There were 2 approaches I considered for the lexer with regard to the data model: the first represents programs as a list of characters, with a pointer to the current position in the source code that is incremented or decremented as it scans the program. This can lead to unpredictable side effects and repeated code since everytime the pointer is used, you must ensure it has not exceeded the bounds of the list. Furthermore due to the nature of parsing multicharacter tokens such as numbers and strings, the behaviour for incrementing this pointer is not uniform and can be difficult to keep track of in the program, making code very difficult to debug.

Instead I opted to use rust's native `Peekable` class which encapsulates this behaviour at the cost of more complex variable lifetimes and memory management. I pass a reference to the Lexer struct into each subroutine to hold the current state of the program.

This program works by iterating over the source code character by character and appending its Token representation onto a Vector containing the tokenised source code. When it encounters a number, it instead appends that first digit to a numeral string and continues iterating over all consecutive digits until it has built up a numeral string representing this number.

```
// ==== src/lexer/lexer.rs ====
use std::{iter::Peekable, str::Chars};
use super::token::{Token, SyntaxError};


pub struct Lexer<'a> {
    program: Peekable<Chars<'a>>,
}

impl<'a> Lexer<'a> {
    pub fn new(program: &'a str) -> Self {
        Lexer {
            program: program.chars().peekable(),
        }
    }

    pub fn read_char(&mut self) -> Option<char> {
        self.program.next()
    }

    pub fn peek_char(&mut self) -> Option<&char> {
        self.program.peek()
    }

    pub fn tokenize(&mut self) -> Result<Vec<Token>,
        ↪ SyntaxError> {
        let mut tokens: Vec<Token> = Vec::new();

```

```
        // iterate over all characters in the source code
        while let Some(ch) = self.read_char() {
            match ch {
                ch if ch.is_whitespace() => {}
                '(' => tokens.push(Token::LPAREN),
                ')' => tokens.push(Token::RPAREN),

                '+' => tokens.push(Token::ADD),
                '-' => tokens.push(Token::SUB),
                '*' => tokens.push(Token::MUL),
                '/' => tokens.push(Token::DIV),

                '0'..='9' => {
                    // parse a numebr by collecting consecutive digits in the source code
                    // into the 'numeral' string
                    let mut numeral = String::new();
                    while let Some(ch) = self.peek_char() {
                        if !ch.is_numeric() {
                            break;
                        }

                        numeral.push(self.read_char().unwrap());
                    }

                    tokens.push(Token::Number(
                        (ch.to_string() + &numeral).parse::<u32>().unwrap(),
                    ));
                }

                _ => {
                    return Err(SyntaxError::new(format!(
                        "SyntaxError: invalid character in source code '{}'",
                        ch
                    )))
                }
            }
        }

        tokens.push(Token::EOF);
        Ok(tokens)
    }
```

```
}
```

```
// ==== src/main.rs ====
mod lexer;
use lexer::{lexer::Lexer, token::Token};

fn main() {
    let program = "(1-20)/(2-3)";
    let mut lexer = Lexer::new(program);

    let tokens: Vec<Token> = match lexer.tokenize() {
        Ok(tokens) => tokens,
        Err(err) => {
            eprintln!("{}", err.msg);
            std::process::exit(1);
        },
    };

    println!("{:?}", tokens);
}
```

```
$ cargo run
>> (-10 + -2/3)/10 - 2
LPAREN, SUB, Number(10), ADD, SUB, Number(2), DIV, Number(3),
    ↪  RPAREN, DIV, Number(10), SUB, Number(2), EOF
```

### 1.6.3 Parser

The second component of this system is the parser to convert the tokenised source code into an abstract syntax tree representing the expression. Of the 2 main parsing methods, I chose a Pratt parser for this prototype due to the clear control flow when compared to a bottom-up or recursive descent parser. Since each operation in a pratt parser is assigned a binding prefernce to determine the order of operations, I wrote a subroutine to get the pratt parser precedence from any operation Token.

```
// ==== src/lexer/token.rs ====

// convert a Token enum into a numerical value representing
// the pratt parser precedence of that operation
impl Token {
```

```
    pub fn get_precedence(&self) -> i32 {
        match self {
            Token::ADD | Token::SUB => 10,
            Token::MUL | Token::DIV => 20,
            _ => -1,
        }
    }
}
```

Since rust does not support inheritance, I used the relationships between enums to achieve a similar effect. The data model for parsed Nodes uses enums for the top level expressions (ie. expressions that alone would make for a valid program - a valid program could be any of, prefix: "-10", literal: "3", infix: "1+2").

These can take recursive parameters (contained within a `Box<>` to allocate them onto the heap and permit this recursive behaviour). At the core of the nested Infix and Prefix expressions (an infix expression is in the form a+b, and prefix -a) are Literals, these are the smallest units of the program (in this case only unsigned integers).

Decomposing expressions into multiple seperate enums (Prefix, Operation, Literal) reduces heap memory useage and ensures cleaner and more robust code since the parameters for each Node is limited to only what is valid, meaning that the nodes themselves will not have to be validated during code generation in the compiler.

```
// ==== src/parser/ast.rs ====
use crate::lexer::token::Token;

#[derive(PartialEq, Debug, Clone)]
pub enum Expression {
    LiteralExpr(Literal),
    PrefixExpr(Prefix, Box<Expression>),
    InfixExpr(Box<Expression>, Operation, Box<Expression>),
}

#[derive(PartialEq, Debug, Clone)]
pub enum Literal {
    Number(i32),
}

#[derive(PartialEq, Debug, Clone)]
pub enum Prefix {
    Minus,
}

#[derive(PartialEq, Debug, Clone)]
pub enum Operation {
    Add,
```

```
    Subtract,
    Multiply,
    Divide,
}

// implement the try_from() property to conveniently convert Tokens
// into Operation types
impl TryFrom<Token> for Operation {
    type Error = &'static str;
    fn try_from(token: Token) -> Result<Self, Self::Error> {
        match token {
            Token::ADD => Ok(Operation::Add),
            Token::SUB => Ok(Operation::Subtract),
            Token::MUL => Ok(Operation::Multiply),
            Token::DIV => Ok(Operation::Divide),
            _ => Err("Invalid Type: can only convert operators")
        }
    }
}
```

```
use super::ast::{Expression, Literal, Operation, Prefix};
use crate::{lexer::token::SyntaxError, Token};

pub struct Parser {
    tokens: Vec<Token>,
    pos: usize,
    tok: Token,
}

impl Parser {
    pub fn new(tokens: Vec<Token>) -> Self {
        let tok = tokens[0].clone();
        Parser {
            tokens: tokens,
            pos: 0,
            tok: tok,
        }
    }

    fn advance(&mut self) {
        if self.pos + 1 < self.tokens.len() {
            self.pos += 1;
```

```
            self.tok = self.tokens[self.pos].clone();
        }
    }

    fn retreat(&mut self) {
        self.pos -= 1;
        self.tok = self.tokens[self.pos].clone();
    }

    // throw a SyntaxError if the parser encounters an unexpected token (!= t)
    fn assert(&self, t: Token) -> Result<(), SyntaxError> {
        if self.tok != t {
            return Err(SyntaxError::new(format!(
                "SyntaxError: expected to encounter token of type '{:?}', instead encountered '{:?}'",
                t,
                self.tok,
            )))
        }
        Ok(())
    }

    pub fn parse(&mut self) -> Result<Expression, SyntaxError> {
        return self.parse_expression(0);
    }

    fn parse_expression(&mut self, rbp: i32) -> Result<Expression, SyntaxError> {
        // parse the left hand side of an infix operation and determine the
        // precedence of the next operation
        let mut lhs = match self.parse_atom() {
            Ok(lhs) => lhs,
            Err(e) => {
                return Err(e);
            }
        };

        self.advance();
        let mut peek_rbp = self.tok.get_precedence();

        // if an operation token is encountered, parse the tokens as an infix expression, and
```

```
        // continue to parse to the lhs if operators with a higher precedence than rbp (e.g. *, /)
        // are encountered - order of operations.
        while self.pos < self.tokens.len() && peek_rbp > rbp {
            lhs = match self.parse_infix(lhs, self.tok.clone()) {
                Ok(lhs) => lhs,
                Err(e) => {
                    return Err(e);
                }
            };

            peek_rbp = self.tok.get_precedence();
        }

        Ok(lhs)
    }

    fn parse_infix(&mut self, lhs: Expression, op: Token) -> Result<Expression, SyntaxError> {
        self.advance();

        // recursively pass the rhs which iteslf can be an expression
        let rhs = match self.parse_expression(op.get_precedence() + 1) {
            Ok(rhs) => rhs,
            Err(e) => return Err(e),
        };

        Ok(Expression::InfixExpr(
            Box::new(lhs),
            Operation::try_from(op).unwrap(),
            Box::new(rhs),
        ))
    }

    fn parse_atom(&mut self) -> Result<Expression, SyntaxError> {
        let expr = match self.tok {
            Token::Number(n) => Expression::LiteralExpr(Literal::Number(n as i32)),

            Token::SUB => {
                // parses the prefix operation (-): calls
```

```
                ↪ parse_expression() with rbp 40
            // meaning the rhs can be an expression
                ↪ itself however only one with precedence
                ↪  > 40
            // (only parenthesised expressions)
            self.advance();
            let expr = match self.parse_expression(40) {
                Ok(expr) => expr,
                Err(e) => return Err(e),
            };

            self.retreat();
            Expression::PrefixExpr(Prefix::Minus, Box::
                ↪ new(expr))
        }

        Token::LPAREN => {
            self.advance();
            // call parse_expression() with rbp 0 since
                ↪ any combination of operations can be
            // contained within parentheses and parsed as
                ↪  inside
            let expr = match self.parse_expression(0) {
                Ok(expr) => expr,
                Err(e) => return Err(e),
            };

            match self.assert(Token::RPAREN) {
                Ok(_) => {},
                Err(e) => return Err(e),
            };

            expr
        }

        _ => return Err(SyntaxError::new(
            format!(
                "SyntaxError: unexpected token
                    ↪ enocountered when parsing infix
                    ↪ expression '{:?}'",
                self.tok,
            )
        )),
    };

    Ok(expr)
```

```
    }
}
```

```
$ cargo run
    Compiling parser v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 1.14s
    Running `target/debug/parser`

>> (-10 + -2/3)/10 - 2
InfixExpr(
    InfixExpr(
        InfixExpr(
            PrefixExpr(Minus, LiteralExpr(Number(10))),
            Add,
            InfixExpr(
                PrefixExpr(Minus, LiteralExpr(Number(2))),
                Divide,
                LiteralExpr(Number(3))
            )
        ),
        Divide,
        LiteralExpr(Number(10))
    ),
    Subtract,
    LiteralExpr(Number(2))
)
```

## 1.7 Technical Decisions

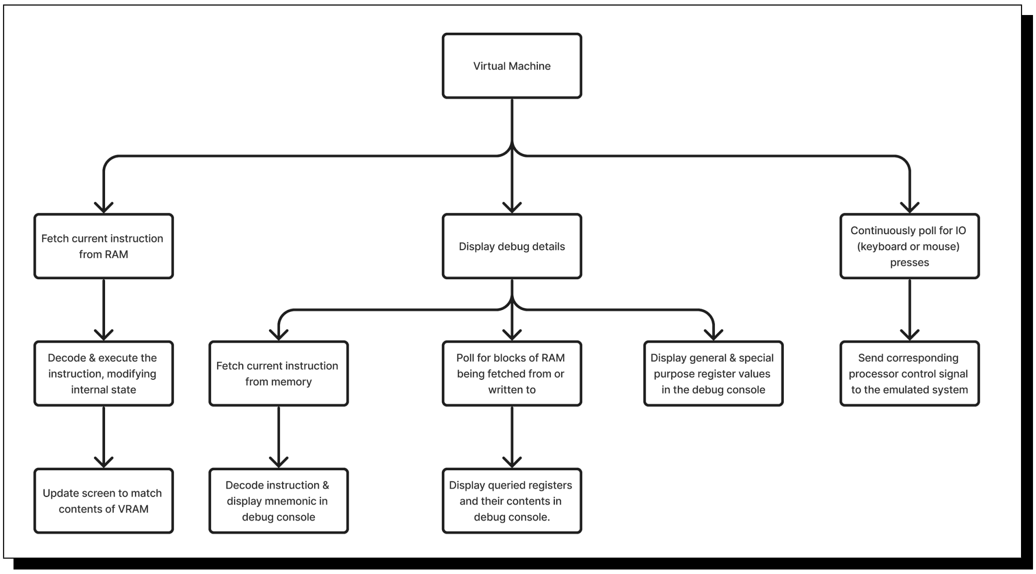
I decided to use two languages for this project, C for the virtual machine - due to the minimal overhead required for interacting with machine code. It has minimal abstractions making bitwise logic easier to follow and more concise to write. However, I will use Rust for the compiler and assembler due to the higher level features it implements, including strings and compound data types that will come in handy when working with a text file.

# 2 Design

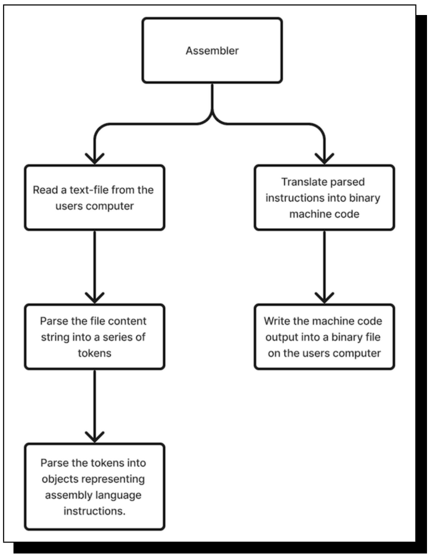
## 2.1 High Level Overview

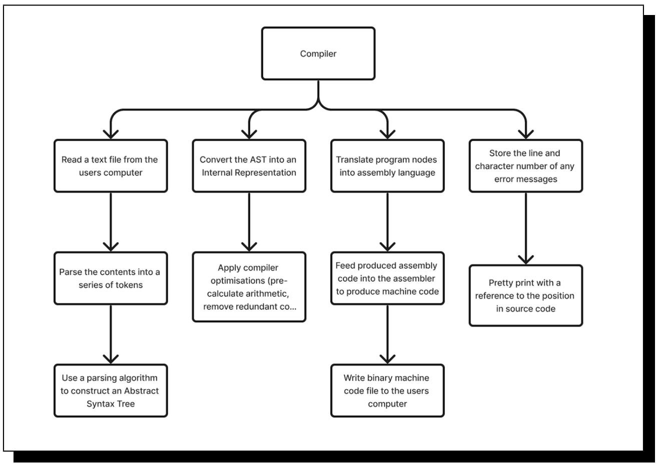
The system will comprise 3 parts required to simulate and program a proprietary processor. It will include a virtual machine to emulate the execution of binary machine code catridges, an assembler to translate higher level assembly code into machine code, and finally a compiler for a higher level language to easily program complex applications to run on the processor.

The virtual machine consists of two main processes, the debugger and interpreter. The interpreter will continiously step through memory, decoding and executing instructions sequentially whilst displaying the contents of VRAM through the pixel display.




The assembler consists of a single pipeline for transforming ASCII assembly programs into binary machine code. The files are loaded into the interpreter which stores their contents in a string. The contents are then tokenised by a lexer into a list of objects representing the foundational elements of the program (e.g. STRING, BRACE, NUMBER) and parsed into a sequence of assembly language instructions. These instructions are translated into binary machine code according to the instruction set architecture (defined in 2.2.1.4) which is then written to a file and stored on the users computer.






Much like the assembler, the compiler takes an ASCII program, converts it into tokens and parses it into an Abstract Syntax Tree (AST) representing the structure and order of operations of the program. This AST is converted into an internal representation (IR) designed to help easily locate potential optimisations in the source code (e.g. pre-calculating arithmetic or removing redundant code), these optimisations are made and the IR is converted into an intermediate assembly language due to the presence of high level optimisations such as labels and macro-instructions. Finally, this assembly code is inserted into the assembler and the produced machine code is stored as a file on the users computer.

## 2.2 Component Design

### 2.2.1 Instruction Set Architecture

#### 2.2.1.1 Computer Architecture

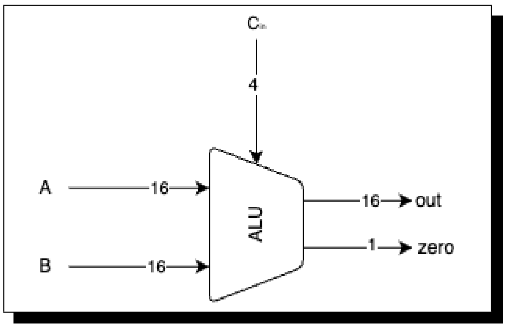
Objective 1.1 (implementing a RISC architecture) meant many tradeoffs had to be made between convenience and practicality, There is 64Kb of memory (65,526 memory locations) specified by 16-bit addresses, requiring a 16-bit Program Counter (PC). The clock speed for the CPU runs at 600Hz. Each instruction is 32-bits, meaning two processor cycles are required to fetch an instruction and store the high and low words in the Current Instruction Register (CIR).

I decided on 32 16-bit general purpose registers, any two of which can be inputs to the ALU, which itself has two outputs - a 16 bit result which can be written to a register or memory depending on the instruction, and a 1 bit zero bit which is set when the result of the calculation is 0.

There are a number of special purpose registers, which, by convention are dedicated an address in memory, one for the stack pointer (SP) and another for the sound timer (ST). There is no dedicated stack and is instead allocated a downwards growing section of memory. The stack stores register values and return addresses when calling functions, however since the instruction set contains no `call` or `ret` instructions, this must be performed manually by the programmer. Each cycle the sound timer is non-zero, the computer produces a sound and decrements ST, allowing the length of the noise to be specified.

#### 2.2.1.2 Arithmetic and Logic Unit

The core of any instruction set is the Arithmetic Logic Unit (ALU) so I began by designing an interface for that. I decided on 2 16-bit inputs to the ALU, which can either take register values, or for an Ri-type instruction, the value of the 16-bit immediate field. There are 4 ALU control bits into the ALU, whcih dictate the operation to be performed. The first two control bits negate their respective inputs, and the second two determine the arithmetic or logical operation to perform. Combinations of these ALU control bits can produce a variety of different operations, all of which are detailed in the table below. (Noam Nissan 2020)




| nx | ny | op | out |
|---|---|---|---|
| 0 | 0 | 00 | and |
| 0 | 0 | 01 | or |
| 0 | 0 | 10 | add |
| 0 | 1 | 10 | sub |
| 0 | 1 | 11 | slt |
| 1 | 1 | 00 | nor |

#### 2.2.1.3 Assembly Language

I decided to use a MIPS instruction set architecture, minimising the number of instructions supported by the processor. I settled on 2 instruction types: R-type instructions (performing ALU operations on register values), I-type instructions (operations involving both registers and immediate fields - includes jump and branch instructions) from these two types, the following instruction set can be constructed, demonstrated with a program to multiple two numbers stored in memory. ('[]' are used to indicate a memory address):

```
R-type: add, sub, and, or, nor, slt, sll, slr
I-type: addi, andi, ori, lw, sw, bge, bne
J-type: jmp, jr, jal

// multiply the numbers in memory address 0xb000 and 0xb001, and write the answer to 0xb002
li r31, 0xb000 // pointer to the start of data memory

lw r1, 0($r31)
lw r2, 1($r31)

```

```
.loop
  // if r2 is 0, break
  li r0, 0
  beq r0, r2, [.store]

  add r1, r1, r1

  addi r2, r2, -1

  jmp [.loop]

.store
  sw r1, 2($r31)
```

#### 2.2.1.4 Machine Code Encoding

Below is the breakdown of how R/I/J type instructions are represented in binary, broken down into their respective fields. All instructions have a 4-bit opcode which dictates the type of instruction (and consequentially which of 2 decoding types should be used when decoding the instruction). In the machine code, Ri, L and J type instructions are encoded with the same operands, however they are each interpreted differently by the processor.

- `00-xx`: R-type instructions *to perform operations between two registers*
  - `5-bits rs`: the first of two input registers to the ALU.
  - `5-bits rt`: the second of two input registers to the ALU.
  - `5-bits rd`: the register in which to store the result of the operation.
  - `4-bits func`: the control bits determining the ALU operation.
- `01-xx`: Ri-type instructions *(to perform operations between a register and immediate value)*
  - `5-bits rs`: the first of two input registers to the ALU.
  - `5-bits rt`: the second of two input registers to the ALU.
  - `16-bits immediate`: the data used as the second ALU input.
- `10-xx`: L-type instructions *(to load/store words from memory)*
  - `5-bits rs`: the register containing the base offset for calculating memory addresses.
  - `5-bits rt`: the register to store/read data from/into memory.
  - `16-bits immediate`: the offset from the base address for the calculating memory addresses.
- `11-xx`: J-type instructions *(to jump to an address in memory)*
  - `5-bits rs`: the register containing the memory address to branch to in RAM.

– `5-bits rt`: the register to store the return address of the jump.
– `16-bits immediate`: the address to branch to in RAM

| | | | | | |
|---|---|---|---|---|---|
| R-format | opcode [4] | rs [5] | rt [5] | rd [5] | func [4] |
| I-format | opcode [4] | rs [5] | rt [5] | immediate [16] | |
| J-format | opcode [4] | immediate [16] | | | |

Next to each assembly instruction below is its machine code encoding, showing the relationship between the assembly language operands and the encoding fields.

```
addi $r5, $r0, 14     // 0110 00000 00101 00000000 00001110
lw $r4, 10($r30)      // 1000 11110 00100 00000000 00001010


slt $r0, $r1, $r2z    // 0000 00001 00010 00000 0111
bne $r0, $r4, [0xb020] // 1101 00000 00100 10110000 00100000
jmp [0x8080]        // 1111 10000000 10000000
```

Since each instruction is 32-bits and contains two words, an instruction must be stored across two memory locations. Little-endian and Big-endian are different standards for storing data across multiple bytes, for little endian - the LSB of the instruction is stored in the first memory location and the MSB in the second, vise versa for big endian. The modern standard has become little-endian encoding, and is what I will use in this project. Hence, the instruction `0110110000000101 0000000000001110` would be stored in memory as:

```
  [0]: 0000000000001110
  [1]: 0110110000000101
```

#### 2.2.1.5 High Level Language

Following on from my client interview, the high level language should take inspiration from Go for its syntax and type system, although perhaps with a greater focus on pointers - due to the additional flexibility they open up for the programmer. I decided on the `func` keyword to define functions, and `->` to indicate the return type of a function. I also decided to mandate the data type in a variable declaration (specified by the `var` keyword, to aid parsing). The syntax for pointers is similar to that of C or C++, pointers are created with the `*` symbol and dereferenced with `&`. Arrays and strings are null terminated and can be indexted using square bracket syntax (which is interchangable with `&(array + index)` when working with memory addresses and pointers).

```
func println(str: char*) {
```

```
  // copy character bytes from string to memory addresss 0
      ↪ x8000+
  // either &() or [] syntax can be used to write to or read
      ↪ from a pointers address
  for (var ptr: i16 = 0; str[ptr] != 0; ptr++) {
    var addr: u16* = 0x8000;
    &(addr + ptr) = str[ptr];
  }
}

func main() -> int {
  var x: i16 = 10;
  var y: i16 = 5;
  var z: i16;

  if (x > y) {
    z = 1;
  } else {
    z = 0;
  }

  // strings are null terminated
  var str: char* = "Hello, World!"
  println(str);
}
```

## 2.3 Virtual Machine

The virtual machine should act as an interpreter, initialising a virtual processor - with an interface mimicking the registers, memory, and clock of the hardware descriptor. It needs to load a binary program cartridge into an array of bytes represening the computers memory, and steps through it word by word, decoding each instruction encountered and carrying out the subsequent operations accordingly. The data structure representing the processor will look as follows:

```
STRUCT lion
  SIGNED WORD[65535] memory
  SIGNED WORD[32] registers
  UNSIGNED WORD pc
  UNSIGNED WORD elapsed_cycles;
ENDSTRUCT
```

The processor has $2^{16}$ different 16-bit memory locations, both this and the 32 general-purpose registers can be represented by an array of signed words. The program counter must remain unsigned however, as its sole purpose is to point to locations in memory. The elapsed cycles field will be used to keep track of the number of cycles each instruction takes to execute - and to tell the emulator how long to wait for to maintain correct clock timing.

### 2.3.1 Algorithms

#### 2.3.1.1 Fetch-Execute Cycle

The virtual machines primary directive is to simulate the fetch-decode execute cycle of the processor. This involves fetching an instruction from memory, incrementing the program counter, and then depending on the opcode of the fetched instruction, executing logic to handle that instruction. The `elapsed_cycles` field on the struct can be used to keep a record of the time the program should wait after each instruction to maintain consistancy with the clock speed of the processor. This will be set depending on the instruction according to the processor architecture, e.g. a jump instruction may only take 2 cycles to execute, wheras a branch on equals would take 4. The basic structure for the fetch-execute cycle procedure should look as follows:

```
PROCEDURE FETCH_EXECUTE()
  READ BINARY CARTRIDGE INTO MEMORY ARRAY

  WHILE PROCESSOR IS RUNNING
    IF CPU CYCLES BACKLOG IS EMPTY
      FETCH INSTRUCTION
      INCREMENT PC

      MATCH ON INSTRUCTION OPCODE
        R:
          MAP func FIELD TO OPERATION
          FETCH REGISTER VALUES
          PERFORM OPERATION ON REGISTERS
          STORE RESULT IN REGISTER
          SET CPU CYCLES BACKLOG
        LW:
          FETCH REGISTER VALUE
          SUM ADDRESS AND REGISTER
          SET ADDRESS TO RESULT
          FETCH VALUE FROM MEMORY LOCATION
          STORE THE RESULT IN REGISTER
          SET CPU CYCLES BACKLOG
        JAL:
          STORE PC VALUE IN REGISTER
          SET PC TO IMMEDIATE FIELD
          SET CPU CYCLES BACKLOG
    ELSE
      DECREASE CPU CYCLES BACKLOG
```

```

    WAIT CLOCK CYCLE
ENDPROCEDURE
```

First the words from `memory[pc]` and `memory[pc+1]` must be fetched and stored together in a 32-bit current instruction register. From here, the 4-bit opcode should be extracted and used to determine how the instruction is executed. Each instruction will take multiple clock cycles to execute, and the field `elapsed_cycles` on the `lion struct` is used to keep track of this, after each instruction is executed, the processor should wait for `elapsed_cycles` clock cycles before proceeding to the next.

Since each instruction is 32-bits, instructions span 2 memory locations and are stored according to the little endian convention, where the least significant byte is stored in the first location. Below is the pseudocode for fetch-execute cycle algorithm:

```
PROCEDURE FETCH_EXECUTE
  READ_FILE_INTO_MEOMRY()

  WHILE (NOT hlt) DO
    // only begin the next instruction cycle once the
       ↪ previous has finished executing
    IF (elapsed_cycles == 0) THEN
      // LSB stored before the MSB in memory
      // binary shift the MSB to fit the LSB
      WORD cir = (memory[pc+1] << 16) | memory[pc]
      pc += 2

      EXECUTE_INSTRUCTION(cir)
    ELSE
      elapsed_cycles -= 1
    ENDIF
z‘

    WAIT 1/CLOCK_SPEED
  ENDWHILE
ENDPROCEDURE

PROCEDURE EXECUTE_INSTRUCTION(u32 instruction)
  NIBBLE opcode = instruction & 0xF000
  MATCH (opcode)
    CASE 0x0000:
      UNSIGNED BYTE rs = instruction[4:9]
      UNSIGNED BYTE rt = instruction[10:15]
      UNSIGNED BYTE rd = instruction[16:21]
      UNSIGNED BYTE func = instruction[22:26]

```

```
      SIGNED WORD OP() = HASHMAP[func]
      registers[rd] = OP(rs, rt)
      elapsed_cycles = 4
      BREAK

    CASE 0100: // lw rt, i16($rs)
      SIGNED WORD offset = instruction[16:]
      UNSIGNED BYTE rs = instruction[4:9]
      UNSIGNED BYTE rt = instruction[10:15]

      UNSIGNED WORD address = offset + registers[rs]
      registers[rt] = memory[address]
      elapsed_cycles = 4
      BREAK;

    CASE 0x1111: // jal i16
      UNSIGNED BYTE rt = instruction[10:15]
      registers[rt] = pc
      pc = instruction[16:]
      elapsed_cycles = 3
      BREAK
  END
END
```

#### 2.3.1.2 Reading a Binary File

The virtual machine needs to be able to load a binary program from a file on the users computer into the processors memory array. This should default to the first address in memory since the pc (program counter) should point to the first instruction and the computer lacks a bootloader. The algorithm should open the file on the users computer from the filename provided in the CLI arguments, and reads it through two bytes at a time, loading each word into the processors memory array - until the end of the file has been reached.

```
PROCEDURE READ_FILE_INTO_MEMORY()
  IF (LEN(ARGS) <= 1) THEN
    THROW "Missing filename argument"
  ENDIF

  // get cartridge filename from cli arguments
  STRING filename = ARGS[1]

  // open the file and read in the first word (2 bytes)
  FILE file = OPEN(filename)

  // since the file size is in bytes, and iterating through
```

```
    ↪ words, SIZE(file)/2
  FOR (WORD addr = 0; addr < SIZE(file)/2; addr++) DO
    memory[addr] = READ_WORD(file)
  ENDFOR
ENDPROCEDURE
```

## 2.4 Assembler

The assembler takes in an assembly program as a text file, and outputs the corresponding binary machine code. It consists of three parts: a lexical analysis of the text that produces an array of textual elements (e.g. NUMBER, LABEL, COLON); a parser that combines these program tokens into a sequence of instructions; and finally a compiler to synthesise machine code instructions. The broad pipeline for the assembler should look like the following:

```
PROCEDURE assemble()
  READ TEXT FILE INTO STRING

  // COMPILE PROGRAM STRING INTO A TOKEN ARRAY
  TOKEN LIST = lex()

  // COMBINE TOKENS INTO A SERIES OF INSTRUCTIONS
  INSTRUCTION LIST = parse()

  // CONVERT INSTRUCTIONS INTO MACHINE CODE
  MACHINE CODE = compile()

  WRITE MACHINE CODE TO FILE
ENDPROCEDURE
```

### 2.4.1 Data Structures

#### 2.4.1.1 Token

I need a token data structure to represent the program elements in the lexical stage of the assembler, it needs to be addressed in a uniform manner regardless of the type, and since different tokens can take different arguments - this logic needs to be split into an enum. The token struct should contain the position of the token (both start and end position since tokens can vary in length), and this can be used to generate error messages that point to the particular location in the file in which the error occured.

```
STRUCT TOKEN
  TOKEN_TYPE t_type
  UNSIGNED INTEGER s_pos
  UNSIGNED INTEGER e_pos
```

```
ENDSTRUCT

ENUM TOKEN_TYPE
  NUMBER(i16),
  LABEL(String),
  REGISTER(String),
  LPAREN,
  RPAREN,
  LSQUARE,
  RSQUARE
  DOLLAR,
  COMMA,
  DOT,
  ADD
  BNE,
  JAL
ENDENUM
```

#### 2.4.1.2 SyntaxError

One of the objectives for this project (Objective 2.6) was the production of relevant error messages - pointing out the position in source code in which they occur. I decided on the format below for syntax errors.

```
SyntaxError: invalid character '%', line: 2
lw $r0, 10(%r0)
       ^
```

For this: two things are required, the error message itself, and the position of the error (start and end position). The program string itself instead of being included in the error can be passed as a parameter when displaying the message and the display method should be able to reference the line and character(s) on which the error occured from this.

```
STRUCT SYNTAX_ERROR
  STRING msg
  UNSIGNED INTEGER s_pos
  UNSIGNED INTEGER e_pos
ENDSTRUCT
```

The algorithm to display syntax errors should convert the s_pos and e_pos variables into a line number and column tuple, (initially an index in the 1 dimensional program string). It should then identify the column of the line in which the error occurs and print a carrat under that character. Since s_pos and e_pos refer to indexes in the program string, the line number and column in which they occur must be determined: after iterating through the string line by line and decrementing the variables by the number of characters on each line, when the variables are less than the length of the subsequent line, the (line, col) position of the error has been found.

```
PROCEDURE DISPLAY_SYNTAX_ERROR(program: STRING)
  SPLIT PROGRAM INTO LINES
  ITERATE THROUGH EACH LINE
    DECREMENT s_pos and e_pos BY EACH LINE LENGTH

    WHEN s_pos < THIS LINE LENGTH
    SET (s_line, s_col) TUPLE

    WHEN e_pos < THIS LINE LENGTH
    SET (e_line, e_col) TUPLE

  PRINT msg, "line: ", line
  PRINT line
  PRINT " " * (s_col - 1) // SPACE BEFORE COL AND ERR
    + "^" * (e_col - s_col + 1) // LENGTH OF ERR
ENDPROCEDURE
```

### 2.4.2 Algorithms

#### 2.4.2.1 Lexical Analysis

The lexer needs to step through the program string one character at a time and produce a vector of tokens containing the elements of the program (e.g. numbers, brackets or commas) that the parser can easily iterate through.

```
PROCEDURE lex()
  ITERATE THROUGH THE PROGRAM CHARACTER BY CHARACTER
  CREATE A VECTOR OF TOKENS

  MATCH CHARACTER
    FOR A SINGLE CHARACTER TOKEN e.g. ('(', '$',',')
    APPEND THE TOKEN TO THE VECTOR

    FOR A MULTI CHARACTER TOKEN e.g. numbers
    CREATE A NUMBER STRING
    WHILE character IS A NUMBER
      APPEND character TO NUMBER STRING
    ENDWHILE
    PARSE NUMBER STRING TO INTEGER
    APPEND TOKEN TO THE VECTOR
```

```
  ENDMATCH

  APPEND EOF TOKEN TO VECTOR
ENDPROCEDURE
```

I will create a lexer struct to store the internal state of the lexer: e.g. the current character, read position, program string. The lexer should have functions to advance (eat) a character, and peek at the next character.

```
STRUCT LEXER
  STRING program
  UNSIGNED INTEGER pos
  CHAR ch
ENDSTRUCT

PROCEDURE LEXER.eat()
  IF SELF.pos < LEN(SELF.program) THEN
    SELF.pos += 1
    SELF.ch = SELF.program[SELF.pos]
  ELSE
    SELF.ch = EOF
  END
ENDPROCEDURE

FUNCTION LEXER.peek() RETURNS CHAR
  IF SELF.pos + 1 < LEN(SELF.program) THEN
    RETURN SELF.program[SELF.pos + 1]
  ELSE
    RETURN NONE
  ENDIF
ENDFUNCTION
```

The `lex()` method should use the above functions to iterate through each character in the program string, if it encounters a single character token e.g. '(', '$', that should be appended to the token vector. Should it encounter a alphabetic character, it should parse an identifier and build up a string of all consecutive alphanumeric characters. This identifier should be matched against the mneumonics and return a token specifying the specific instruction e.g. LW, or ADD. Should it encounter a number, if the first two characters are '0b' or '0x' the following digits should be parsed as binary or hexidecimal respectively - else they should be interpreted as denary.

```
PROCEDURE lex()
  TOKEN[] tokens

```

```
WHILE SELF.ch != EOF DO
  TOKEN tok_type;
  UNSIGNED INTEGER s_pos = SELF.pos
  MATCH SELF.ch
    CASE '(':
      tok_type = LPAREN
    CASE ')':
      tok_type = RPAREN
    CASE ',':
      tok_type = COMMA
    //[...]

    CASE '.':
      IF SELF.peek() is ALPHABETIC:
        tok_type = lex_identifier()
      ELSE:
        tok_type = DOT

    case SELF.ch is NUMERIC:
      UNSIGNED INTEGER base = 10
      IF SELF.ch == '0' AND SELF.peek() == 'b':
        // point to the first character of the
        // digit string for lexing
        SELF.eat()
        SELF.eat()
        base = 2
      IF SELF.ch == '0' AND SELF.peek() == 'x':
        SELF.eat()
        SELF.eat()
        base = 16
      ENDIF

      digit_str = lex_number()
      tok_type = NUMBER(INT(digit_str, base))

    case SELF.ch is ALPHABETIC:
      STRING identifier = lex_identifier()
      IF identifier IN mneumonics:
        tok_type = mneumonics[identifier]
      ELSE
        THROW "SyntaxError: unexpected mneumonic"

  tokens.PUSH(
    Token(tok_type, s_pos: s_pos, e_pos: SELF.pos)
  )
ENDWHILE
```

```

  RETURN tokens
ENDPROCEDURE

PROCEDURE lex_identifier() RETURNS STRING
  STRING str = SELF.ch
  WHILE SELF.peek() is ALPHANUMERIC DO
    SELF.eat()
    str += SELF.ch
  ENDWHILE

  // pos will point to the last character of the string
  // meaning the next character will be skipped when
  // eat() is called
  SELF.rewind()
  RETURN str
ENDPROCEDURE
```

#### 2.4.2.2 Parsing

Once the program has been converted into a list of tokens, the next step is to parse these tokens into a series of structs representing the assembly language instructions. The program should iterate through the tokens, and once it encounters a mneumonic, parse the operands of the instruction into a struct representing the encoding type. The function should return a list of these instruction structs. There are two valid types of statements the parser could encounter, a label definiton or an instruction, and therefore an interface is required for these two types under a the branch "Statement".

There are two formats I'm considering for instructions, either each instruction struct is identical and contains a vector of variable length of operands. This has the advantage of making parsing easier, however compilation will be more tedious.

Else, each instruction has a mneumoic field, and then one of 2 format types: I or R depending on the instruction being parsed. Since the `format` field could hold either `R_format` or `I_format`, and rust doesn't support inheritance - an enum can be used instead.

```
TRAIT Statement

STRUCT Label : Statement
  String label
  UNSIGNED INTEGER s_pos
  UNSIGNED INTEGER e_pos
ENDSTRUCT

STRUCT Instruction : Statement
  Token mneumonic;
  InstructionFormat format
```

```

  UNSIGNED INTEGER s_pos,
  UNSIGNED INTEGER e_pos,
ENDSTRUCT

ENUM InstructionFormat
  R(R_format),
  I(I_format),
ENDENUM

STRUCT R_format
  Register rs
  Register rt
  Register rd
ENDSTRUCT

STRUCT I_format
  Register rs
  Register rt
  Number immediate
ENDSTRUCT
```

When parsing, the parser should iterate through each token - depending on which mneumonic it encounters, it should pass a different format of operands, e.g. a `LW` instruction should take the form `lw` $rt, imm($`rs`$)$ wheras a `jmp` instruction `jmp imm`. The `eat_expect()` function helps to simplify the code - as it both advances a token and throws an error if the token it encountered was not the one expected.

```
STRUCT Parser
  Token token
  Token[] tokens
  UNSIGNED INTEGER pos
ENDSTRUCT

FUNCTION Parser.eat_expect(Token token) RETURNS Token | Error
  self.eat()
  IF self.token != token THEN
    THROW "Unexpected token encountered"
  ELSE
    RETURN self.token
  ENDIF
ENDFUNCTION

PROCEDURE parse()
  WHILE self.token != EOF
```

```
    instruction = MATCH self.TOKEN
      CASE LABEL:
        LABEL

      CASE ADD, SUB, AND [...]:
        parse_R_format()

      CASE ADDI, ANDI, BEQ, BNE [...]:
        parse_Ri_format()

      CASE LW, SW:
        parse_L_format()

      CASE JR:
        [...]

      CASE JAL:
        [...]
    ENDMATCH

    self.eat()
    IF self.token != NEWLINE OR self.token != EOF
      THROW "expected newline"
    ENDIF
  ENDWHILE
ENDPROCEDURE

// mneumonic rd rs rt
FUNCTION parse_R_format() RETURNS Instruction
  Token mneumonic = self.token

  Token rd = self.eat_expect(Register)
  self.eat_expect(COMMA)

  Token rs = self.eat_expect(Register)
  self.eat_expect(COMMA)

  Token rt = self.eat_expect(Register)

  RETURN Instruction(mneumonic, R_format(rs, rt, rd))
ENDFUNCTION

// addi $r0, $r1, 0xf0ba
FUNCTION parse_Ri_format() RETURNS Instruction
  Token mneumonic = self.token

```

```
  Token rt = self.eat_expect(Register)
  self.eat_expect(COMMA)

  Token rs = self.eat_expect(Register)
  self.eat_expect(COMMA)

  Token immediate = self.eat_expect(Number)

  RETURN Instruction(mneumonic, I_format(rs, rt, immediate))
ENDFUNCTION

// lw $rt, 10($rs)
FUNCTION parse_L_format() RETURNS Instruction
  Token mneumonic = self.token

  Token rt = self.eat_expect(Register)
  self.eat_expect(COMMA)

  Token immediate = self.eat_expect(Number)

  self.eat_expect(LPAREN)
  Token rs = self.eat_expect(Register)
  self.eat_expect(RPAREN)

  RETURN Instruction(mneumonic, I_format(rs, rt, immediate))
ENDFUNCTION
```

#### 2.4.2.3 Compilation

The compiler has broadly the same structure as the assembler: the file is fed through a lexer and then a parser to build up an abstract syntax tree, and from there, a code generator. However whilst the lexer is broadly similar to the assemblers, the parser is much more complicated. It needs to consider the order of operations of statements and expressions, parse prefix, infix and postfix expressions - multi character tokens, and consider the context in which any of these are used. The parsing algorithm I will use for this is a Pratt parser: it provides each operation with a binding precedence (e.g. multiplication binds stronger than addition) and parses expressions recursively - stopping a particular branch when it encounters an operator with a lower binding preference.

In a pratt parser programs are broken up into statements and expressions: statements are terminated by a semicolon and tend to be inflexible in their usage (such as a function definition or variable declaration), unlike expressions which can be passed around and evaluated. The grammar for my language will look as follows:

```
statement:   function
             | declaration
```

```

function:    FUNC identifier LPAREN (binding COMMA)* binding? RPAREN ( -> type)? block

declaration: VAR binding EQ expression SEMICOLON

binding:     identifier : type

block:       LBRACE ( declaration | expression SEMICOLON )* expression? RBRACE

expression:  call
             | binary
             | prefix
             | postfix
             | literal
             | variable
             | assign
             | break
             | return
             | while
             | for
             | if

literal:     int | char | bool
assign:      expression EQ expression
infix:       expression BINOP expression
postfix:     expression POSTOP
prefix:      PREOP expression
if:          IF LPAREN expression RPAREN block (ELSE block)?
for:         FOR LPAREN ( declaration? expression? SEMICOLON expression? SEMICOLON) block
```

```
ENUM STATEMENT
  CONST {
    binding: Binding,
    expression: EXPRESSION
  }

  FUNCTION {
    identifier: Identifier,
    bindings: Binding,
    return: Type,
    body: Block
```

```
  }
ENDENUM

ENUM EXPRESSION
  INFIX {
    op: BINOP
    lhs: EXPRESSION,
    rhs: EXPRESSION,
  }

  ASSIGN {
    lhs: EXPRESSION,
    rhs: EXPRESSION
  }

  IF {
    condition: EXPRESSION,
    consequence: Block,
    alternative: Block?,
  }

  EMPTY // '()'
ENDENUM

STRUCT Type
  type: TYPE
  size: int
ENDSTRUCT

ENUM TYPE
  INT,
  CHAR,
  VOID,
  BOOL,
  STRUCT(Vec<TYPE>),
  ARRAY(TYPE, int),
  PTR(TYPE),
ENDENUM

STRUCT Binding
  type: Type
  identifier: Identifier
ENDSTRUCT

STRUCT Block
  statements: Vec<STATEMENT>
```

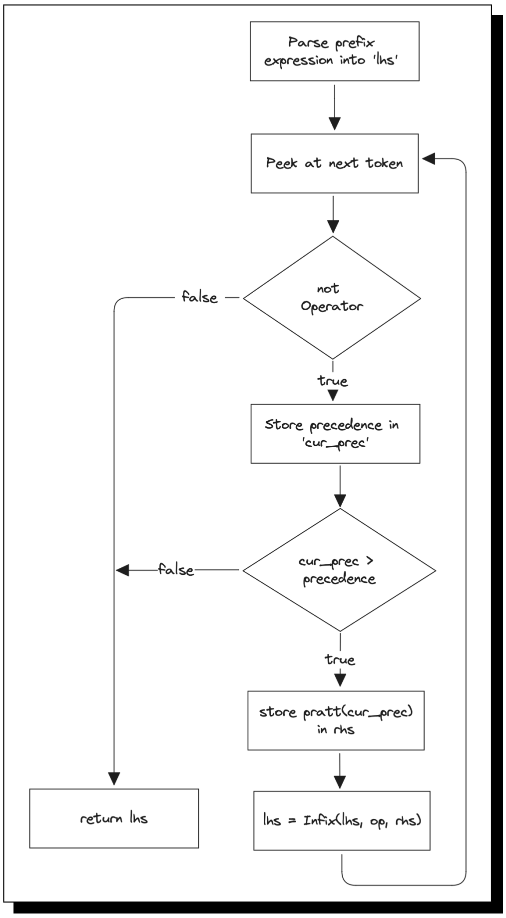
```
  expression: EXPRESSION
ENDSTRUCT
```

When iterating through a the list of tokens, the first thing to parse would be a statement, be that a function definition or variable declaration.

```
FUNCTION parse_statement() RETURNS STATEMENT
  SWITCH self.token
    CASE Token::FUNC:
      parse_func();

    CASE Token::CONST:
      parse_const();

    DEFAULT:
      THROW "unexpected token encountered"
ENDFUNCTION

// <IDENTIFIER>(<BINDING>,?)* (-> TYPE)? { BLOCK }
FUNCTION parse_func() RETURNS STATEMENT
  self.eat()
  Identifier identifier = self.parse_identifier()

  self.eat_assert(LPAREN)

  // continue parsing parameter bindings until a ')' is encountered
  Binding[] params = [];
  WHILE self.token != RPAREN AND self.token != COMMA
    self.parse_binding()
  ENDWHILE
  self.eat_assert(RPAREN)

  // optionally parse a return type if '->' is encountered
  Type type;
  IF self.eat(RARROW) THEN
    type = self.parse_type()
  ELSE
    type = VOID
  ENDIF

  Body body = self.parse_block()

  RETURN Statement::FUNCTION(
    identifier,
```

```
    parameters,
    type,
    body
  )
ENDFUNCTION

// CONST <BINDING> = <EXPRESSION>
FUNCTION parse_const() RETURNS STATEMENT
  self.eat()
  Binding binding = self.parse_binding()
  self.eat_assert(EQ)

  EXPRESSION expr = self.parse_expr()
  self.eat_assert(SEMICOLON)

  RETURN Statement::CONST(
    binding,
    expr
  )
ENDFUNCTION
```

The `parse_expression()` function will contain the meat of the compiler, it should first determine whether the token is a keyword (IF, FOR, WHILE): and if so, parse that accordingly. Otherwise, it should invoke the `parse_pratt_expression()` function.




A pratt parser works by assigning each operation a binding preference, (e.g. multiplication binds with a higher binding power than addition, which itself binds with a higher power than logical operations like '=='). It recursively parses the right hand side of an infix operation (two operands seperated by an operator), whilst the binding power of the current operation is greater that than its parent.

Prefix and postfix operations can also be supported in a pratt parser by verifying the token when parsing the literal on the left hand side of the equation. For example, when parsing `-a + b`, usually the parser would look for a literal token (a varaible name or number) and throw an error if it does not exist. However another condition to run `parse_pratt_expression()` once more, after it ecnounters a prefix operation with a higher binding preference and storing that expression as the left hand side allows brackets and prefix operations to be supported.

```
FUNCTION parse_pratt_expression(INT precedence)
  EXPRESSION lhs

  // parse LHS & any prefixes (parenthesis, unary operations, literals)
  SWITCH self.token
    CASE NUMBER:
      lhs = NUMBER(number)
    CASE RPAREN:
      lhs  = parse_pratt_expression(0)
      eat_assert(RPAREN)
    CASE OP:
      EXPRESSION rhs = parse_pratt_expression(OP.prefix_binding_power)
      lhs = UNARY(OP, rhs)
  ENDSWITCH

  self.eat()
  INT cur_precedence = self.token.binding_power()
  WHILE self.token.is_operation() AND cur_precedence > precedence DO
    self.eat()
    EXPRESSION rhs = parse_pratt_expression(self.token.binding_power + 1)
    cur_precedence = op.binding_power()
    lhs = INFIX(lhs, op, rhs)
  ENDWHILE

  RETURN lhs
ENDFUNCTION
```

#### 2.4.2.4 Optimizations & Sentiment Analysis

The third component of the compiler verifies the abstract syntax tree & performs any optimisations it can. This involves checking variable types & scopes, folding constants into a single literal, ensuring the presence of a `main()` function, and validating the left hand side of any assignment operations.

Constant folding is the first operation the compiler will perform. For each expression in the abstract syntax tree, the program should recursively evaluate the left and the right hand side of the expression. Should an operation be performed on two literal nodes who's value is known at compile time (e.g. ints or chars) - the program should precalculate and store the result of the operation in the abstract syntax tree.

```
FUNCTION fold_constant(expr: EXPRESSION) RETURNS EXPRESSION
  IF expr is LITERAL
    RETURN expr
  ELSE IF expr is INFIX
    lhs = fold_constant(expr.lhs)
    rhs = fold_constant(expr.rhs)

    SWITCH op
      CASE '+':
        return lhs + rhs
      CASE '*':
        return lhs * rhs
      // [...]
    ENDSWITCH
  ENDIF
ENDFUNCTION
```

#### 2.4.2.5 Scoped Variables & The Symbol Table

To keep track of variables within the program, I will create a scope table. It should contain a list of `Scope` structs, addressible by a unique ScopeID for each scope. A scope needs to contain a hashmap of all the variables declared within that scope (stored with their respective types) and a reference to a potential parent scope. The data structures will look as follows:

```
STRUCT SymbolTable
  // store constants and function interfaces
  symbols: HashMap<Identifier, SYMBOL>
  scopes: HashMap<ScopeID, Scope>
ENDSTRUCT

STRUCT Scope
  variables: HashMap<Identifier, Variable>,
  parent: Option<Scope>
ENDSTRUCT

STRUCT Variable
  type: TYPE,
ENDSTRUCT
```

Since code inside a particular scope can access variables declared within a larger, parent scope. Recursion can be used to help resolve a variable given an identifier and a current scope. The `resolve_variable()` function should, given a scope and variable identifier, return the Variable struct (containing the datatype and space in memory) corresponding to the variable with that particular identifier from that scope or any parent scopes.

```
FUNCTION resolve_variable(scope: ScopeID, identifier: Identifier) RETURNS Option<Variable>
  current_scope = self.scopes.get(scope)

  IF current_scope.parent == NULL
    AND !current_scope.contains(identifier) THEN
    RETURN None
  ELSE IF current_scope.contains(identifier)
    RETURN current_scope.get(identifier)
  ELSE
    // recursively call with parent scope
    RETURN resolve_variable(current_scope.parent, identifier)
  ENDIF
ENDFUNCTION
```

#### 2.4.2.6 Type Checking

Each expression in my language has a type, be that `int, void, char, bool` or a pointer to any of the previous. Assignment expressions have a void type. When validating the types of the AST, the program needs to recursively resolve the type of each expression, and ensure any contained expressions (e.g. lhs and rhs in an infix expression) are of compatible types. Further checks need to be performed to ensure that any conditionals resolve down too a boolean and a function is returning the correct type from within its function block. Depth first tree traversal can be used to ensure all nodes are being checked in the correct order.

When type checking a BLOCK statement, all statements internally need to be verified, however the overall 'type' of the block statement is determined by any contained 'return' expressions. Should no return statements exist, the type defaults to VOID.

```
FUNCTION typeck(expr: EXPRESSION) RETURNS TYPE
  SWITCH expr
    // base case for recursive calls
    CASE LITERAL:
      RETURN expr.type
    CASE INFIX:
      lhs_ty = typeck(expr.lhs)
      rhs_ty = typeck(expr.rhs)
      IF lhs_ty != rhs_ty THEN
        THROW unexpected types
      ENDIF
  ENDSWITCH
ENDFUNCTION
```

```
FUNCTION typeck_block(block: BLOCK) RETURNS TYPE
  return_ty = VOID
  FOR stmt in block
    SWITCH stmt
      CASE expr:
        // verify types are consistent and ignore the result
        typeck(stmt AS expr)
      CASE RETURN:
        ty = typeck(stmt)
        IF return_ty == VOID THEN
          return_ty = ty
        ELSE IF return_ty != ty THEN
          THROW inconsistent return types
        ENDIF
    ENDSWITCH
  ENDFOR

  RETURN return_type
ENDFUNCTION
```

#### 2.4.2.7 Intermediate Representation & Code Generation

I will use a tuple based IR for this compiler as it most closely resembles an assembly language, making the code generation phase simpler. Each tuple represents an instruction performed on a set of registers, labels or immediate values described earlier in my instruction set.

```
ENUM Block
  // add $rs + $rt and store in $rd
  ADD(Register, Register, Register)

  // add $rs and immediate value, store in $rd
  ADDI(Register, Register, i16)

  LABEL(String)

  // jump to label and store return address in $rd
  JAL(Register, Label)

  // jump to the address stored in a particular register
  JR(Regsiter)

  // jump to register if $rs == $rt
  BEQ(Register, Register, Label)

```

```
  // set the contents of $rd to $rs
  MOV(Register, Register)

  // set the contents of $rd to an immediate value
  LI(Register, i16)

  // load the contents of memory in the address of immediate + $rs into $rd
  LW(Register, Register, i16)
ENDENUM
```

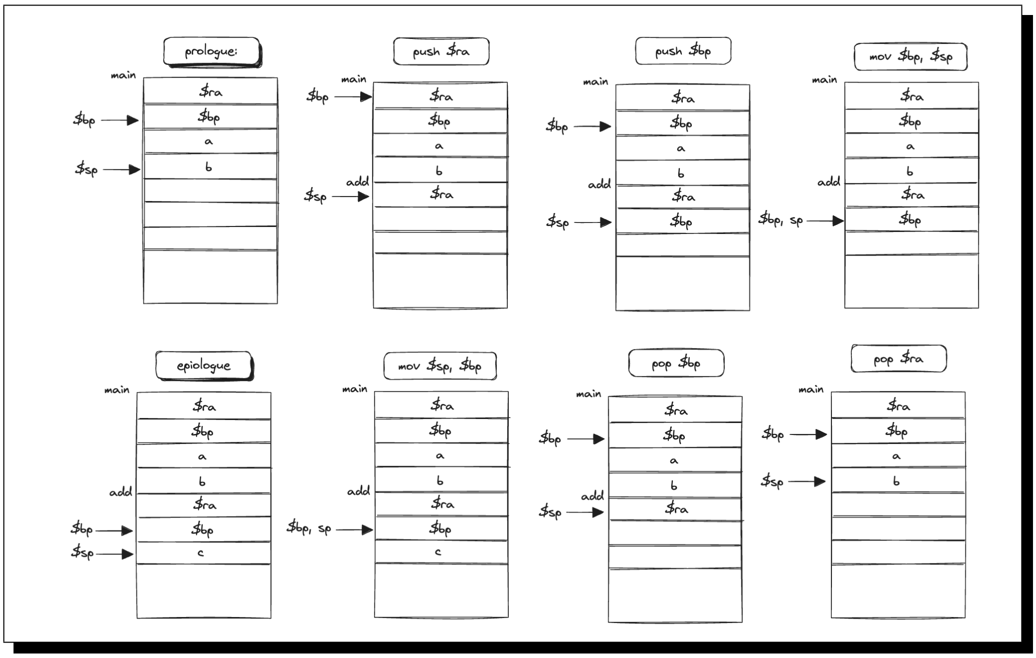
The program should iterate through the abstract syntax tree and build up a list of tuples representing each node in the program. Some statements can be represented by a standard template such as an IF node. Whilst the condition, consequence and alternative must all be compiled recursively, their location in the assembly program can be determined by the following template:

```
// === High Level ===
if a < b {
  [...consequence]
} else {
  [...alternative]
}

// === Assembly ===
SLT $t0, a, b
BEQ $t0, $zero, [.else]

[...consequence]
jmp [.endif]

[.else]
[...alternative]

[.endif]
```

When handling complex expressions you need to store the result of any intermediate calculations. There are two methods I could use for this: the first is a register based approach where the program keeps track of any free registers and assigns each to an intermediate step of the calculation. The second is a stack based approach where after each step the result of the calculation is pushed onto the stack and retrieved when it is required. The register based approach is more memory efficient, however limits how complicated expressions can be since there are only a finite number of registers. I will use the registers `$r0` and `$r1` as a temporary registers to hold the results of calculations popped from the stack before they have been operated upon.

```
// the result of the expression should be stored in $r0
FUNCTION translate_expression(EXPRESSION expr)
  SWITCH expr
    CASE LITERAL:
      "LI $r0, [LITERAL]"

    CASE INFIX:
      translate_infix(expr)
  ENDSWITCH
ENDFUNCTION

FUNCTION translate_infix(EXPRESSION expr)
  translate_expression(lhs)
  PUSH $r0

  translate_expression(rhs)
  POP $r1 // retrieve the value of the lhs from the stack

  SWITCH op
    CASE ADD:
      ADD $r0, $r1, $r0

    CASE SUB:
      SUB $r0, $r1, $r0

    [...]
  ENDSWITCH
ENDFUNCTION
```

For the compiler to reference locally scoped variables, they must be assigned a slot in the current stack frame. The function prologue needs to prepare the stack frame for any locally scoped variables and parameters and push the return address of the function onto the stack (so further function calls will not overwrite the original return stored in the register). This can be done by setting the base pointer (references the top of the stack) to the stack pointer and storing the old base pointer to retrieve when the function completes. This way when new elements are added to the stack, the compiler can assure they will not overwrite the previous functions scope. Once the function has terminated the function epiologue is responsible for returning the stack to the exact state it was before the function is called.




Once the stack has been prepared local variables can be stored internally as an offset from the base pointer of the current stack frame. e.g. in the example above, since the stack grows downwards, the address of variable `b` in the `main()` function could be calculated by adding 2 to the address in the base pointer.

For the compiler to reference locally scoped variables by offset, a data structure containing the context of a function scope is required. It needs a reference to the return label, a reference to the current stack position, and a scope ID so variable types can be looked up in the symbol table.

When a locally scoped variable is declared, a new record in the symbol table needs to be added - storing the stack slot for that particular variable.

```
STRUCT Fctx
  ret_label: STRING
  stack_pointer: i16
  scopeID: ScopeID
ENDSTRUCT

PROCEDURE translate_declaration(binding: BINDING, expr: EXPRESSION)
  // translate the expression and push the result to the top of the stack
  translate_expression(expr)
  PUSH $r0

  // update stack pointer variable after pushing a value
  fctx.stack_pointer -= 1

  // store the stack slot in symbol table
```

```
  symtbl[fctx.scope][binding.identifier].stack_slot = fctx.stack_pointer
ENDPROCEDURE
```

Since the stack slot is an offset from the base pointer, the absolute memory location of any variable can be calculated by adding together the stack slot offset stored in the symbol table and the value of the `$bp` register. This can be used to implement the pointer prefix operation:

```
PROCEDURE translate_prefix(op: UNOP, rhs: EXPRESSION)
  SWITCH op
    CASE DEREF
      // evaluate the rhs of the expression and store in $r0
      translate_expression(rhs)

      // load the value in the memory address given by $r0 into $r0
      LW $r0, $r0, 0

    CASE PTR // rhs is of type "VARIABLE"
      stack_slot = symtbl[fctx.scope][rhs.identifier].stack_slot
      // add the offset to the base pointer and store the calculated memory address in $r0
      ADDI $r0, $bp, stack_slot
ENDPROCEDURE
```

# 3 Technical Solution

## 3.1 A Level Standard

| | Technical Skill | Evidence |
|---|---|---|
| Group A | | |
| 1.1 | Complex User Defined Algorithm: a recursive Pratt parsing algorithm is used to convert a linked list of Tokens into an Abstract Syntax Tree data structure representing the program and order of operations within. | `parse_pratt()` (p. 101) |
| 1.2 | A tree data structure is used to store parsed nodes in an Abstract Syntax Tree. | `SYMBOL` (p. 93) |
| 1.3 | A depth first search graph traversal algorithm is used when evaluatting the result of an arithmetic operation at compile time, and substituting it with a constant. | `Walker` (p. 105) |
| 1.4 | A stack data structure is used to store return addresses of the PC (program counter) from subroutines in the emulator during `call` and `ret` instructions. | `translate_fn()` (p. 119) |
| 1.5 | Complex user-defined use of object oriented programming using both composition, aggregation, and interfaces to relate common behaviour between nodes in the abstract syntax tree. | `SYMBOL`, `EXPRESSION`, `Binding` (p. 93) |
| 1.6 | Dynamic Generation of objects is used in both the parser and the lexer when compiling user's programs and representing them as nodes or tokens. | `parse_symbol()`, (p.96), `parse_rri_format()` (p.89) |
| Group B | | |
| 2.1 | Source code programs are read in from a text file and compiled, the machine code programs are written to a binary file | `LION_read_file()`, (p.75) |

## 3.2 Virtual Machine

The main method and entry point for the virtual machine is responsible for initialising SDL2 (the graphics library) and the LION processor struct, as well as timing the CPU's execution. The first part of the code validates SDL2 is installed on the users computer, and that a binary file has been provided to the virtual machine, throwing respective errors should either condition not be met. Following this, the main loop runs constantly until the halt flag is set on the CPU. The loop provides four functions:

1. **Times refresh rate**: The elapsed time for each cycle is calculated by subtracting the time at the end and start of each loop. The expected time to wait to achieve a 60Hz refresh rate is then calculated with `1000/FPS` and the program waits the difference between the two, ensuring CPU execution slows to the correct speed.

2. **Batches CPU cycles**: Since the rendering code cannot keep up with the 40MHz clock speed requirement, and runs itself at 60Hz, multiple CPU cycles have to be executed per rendering cycle. Calculated by dividing the clock speed by refresh rate. `CLOCK_SPEED/FPS`

3. **Updates the display when VRAM is modified**: the `is_buffered` flag on the LION struct is set when a write instruction updates the region of memory representing VRAM. This prevents the graphics code throttling execution since it is only updated when required.
4. **Decrements the delay timer**: each frame, the delay timer is decremented meaning programs can use the delay timer to maintain track of time.

```c
#include <stdio.h>
#include "lion.h"
#include "screen.h"

int main(int argc, char *argv[]) {
  if (SDL_init() != 0) {
    printf("error initialising SDL2\n");
    return -1;
  }

  if (argc < 2) {
    printf("Incorrect arguments: expected filename");
    return -1;
  }

  struct LION *lion = LION_init();
  lion->is_running = true;
  lion->is_buffered = true;

  LION_read_file(lion, argv[1]);

  while (lion->is_running) {
    uint64_t start_time = SDL_GetPerformanceCounter();
    SDL_Event e;

    while (SDL_PollEvent(&e)) {
      switch (e.type) {
        case SDL_QUIT:
          lion->is_running = false;
          break;
      }
    }

    // batch cycle execution to execute at 40MHz with a refresh rate of 60Hz
    for (int i = 0; (i < CLOCK_SPEED/FPS) && lion->is_running; i++)
      LION_emulate_cycle(lion);
```

```c

    // only update the display if instructions have modified VRAM
    if (lion->is_buffered) {
      SDL_update(lion);
      lion->is_buffered = false;
    }

    // calculate the time elapsed executing CPU cycles and wait for the remaining time
    // to ensure the refresh rate stays a constant 60Hz
    uint64_t end_time = SDL_GetPerformanceCounter();
    uint64_t elapsed_time = (end_time - start_time) / (float) SDL_GetPerformanceFrequency() * 1000.0f;
    SDL_Delay(floor((1000/FPS) - elapsed_time));

    // decrement the delay timer every frame
    if (lion->memory[DT] > 0) {
      lion->memory[DT] -= 1;
    }
  }

  SDL_close();

  return 0;
}
```

Below is the LION struct definition, and the two methods called during initalisation. The first prevents garbage values being preset into the memory or register arrays by initialising all values to 0, and the second reads a binary file one word at a time into the first `n` memory locations.

```c
struct LION {
  uint16_t pc;   // store the address of the next instruction
  uint32_t cir;  // store the current instruction being executed
  uint16_t r[REG_NUM];
  uint16_t memory[RAM_SIZE];
  bool keypad[16];

  bool is_running;
  bool is_buffered; // determines when to update screen
};

struct LION *LION_init(void) {
```

```c
  struct LION *lion = malloc(sizeof(struct LION));

  // initialise all RAM and Registers to 0, preventing indeterminate values
  for (int i = 0; i < RAM_SIZE; i++)
    lion->memory[i] = 0;

  for (int i = 0; i < REG_NUM; i++)
    lion->r[i] = 0;

  lion->pc = 0;
  return lion;
}

void LION_read_file(struct LION *lion, char *filename) {
  FILE *fileptr;
  long filesize;

  // get the length of file in bytes
  fileptr = fopen(filename, "rb");
  fseek(fileptr, 0, SEEK_END);

  filesize = ftell(fileptr);
  rewind(fileptr);

  // iterate through the file 2 bytes at a time, storing them together as a word in memory
  for (int address = 0; address < filesize/2; address++) {
    uint16_t word = 0;
    fread(&word, 1, 2, fileptr);
    lion->memory[address] = word;
  }

  fclose(fileptr);
}
```

After the LION struct has been initialised, it is responsible for carrying out the cycle-by-cycle execution of the processosr. This is handled by two methods: `LION_emulate_cycle()` and `LION_emulate_instruction()`. The former reads an instruction in two parts from two addresses in from memory and logically combines them into a single 32-bit representation. It also increments the program counter before calling the second method which is responsible for executing that instruction. `LION_emulate_instruction()` initially decodes the instruction into any potential memory addresses, immediate values, or registers and switches on the opcode. Each possible instruction has its own branch which contains the code required to simulate its behaviour within the LION processor.

```
void LION_emulate_cycle(struct LION *lion) {
  // combine the two words into a 32 bit instruction
  lion->cir = (lion->memory[lion->pc+1] << 16)
    | lion->memory[lion->pc];
  lion->pc += 2;

  LION_emulate_instruction(lion);
}

void LION_emulate_instruction(struct LION *lion) {
  uint32_t opcode = lion->cir >> 28;

  // decode instruction into components
  uint32_t rs = extract_bits(lion->cir, 5, 5);
  uint32_t rt = extract_bits(lion->cir, 10, 5);
  uint32_t rd = extract_bits(lion->cir, 15, 5);
  int16_t immediate = extract_bits(lion->cir, 15, 16);
  uint32_t func = extract_bits(lion->cir, 20, 4);

  switch (opcode) {
    case HLT:
      LION_display_registers(lion);
      lion->is_running = false;
      break;

    case R: // (func) $rd, $rs, $rt
      switch (func) {
        case FUNC_AND:
          lion->r[rd] = lion->r[rs] & lion->r[rt];
          break;

        case FUNC_OR:
          lion->r[rd] = lion->r[rs] | lion->r[rt];
          break;

        case FUNC_ADD:
          lion->r[rd] = lion->r[rs] + lion->r[rt];
          break;

        case FUNC_SUB:
          lion->r[rd] = lion->r[rs] - lion->r[rt];
          break;

        case FUNC_SLT:
          lion->r[rd] = (int16_t)lion->r[rs] <  (int16_t)lion
```

```
->r[rt];
        break;

      case FUNC_NOR:
        lion->r[rd] = ~(lion->r[rs] | lion->r[rt]);
        break;
    }
    break;

  case ANDI: // andi $rt, $rs, i16
    lion->r[rt] = lion->r[rs] & immediate;
    break;

  case ORI : // ori $rt, $rs, i16
    lion->r[rt] = lion->r[rs] | immediate;
    break;

  case ADDI: // addi $rt, $rs, i16
    lion->r[rt] = lion->r[rs] + immediate;
    break;

  case SLTI: // addi $rt, $rs, i16
    lion->r[rt] = (int16_t)lion->r[rs] < (int16_t)immediate ;
    break;

  case LW: // lw $rt, i16($rs)
    lion->r[rt] = lion->memory[lion->r[rs] + immediate];
    break;

  case SW: // sw $rt, i16($rs)
    lion->memory[lion->r[rs] + immediate] = lion->r[rt];

    // set is_buffered if accessing locations in VRAM
    if (lion->r[rs] + immediate >= 0x8000 && lion->r[rs] + immediate <= 0x8800)
      lion->is_buffered = true;
    break;

  case BEQ: // beq $rt, $rs, i16
    if (lion->r[rs] == lion->r[rt])
      lion->pc += immediate << 1;
    break;

  case BNE: // bne $rt, $rs, i16
    if (lion->r[rs] != lion->r[rt])
```

```
      lion->pc += immediate << 1;
    break;

  case JMP: // jmp i16
    lion->pc = immediate;
    break;

  case JAL: // jmp $rt, i16
    lion->r[rt] = lion->pc;
    lion->pc = immediate;
    break;

  case JR: // jmp $rs
    lion->pc = lion->r[rs];
    break;
  }
}
```

### 3.3 Assembler

The first step of the assembler (and compiler) is to tokenize the program string of characters into a series of data structures that can be processed easier by the parser. I've represented each token as an enum with a type and parameter.

```
#[derive(Debug, PartialEq, Eq, Hash, Clone)]
pub enum Token {
  Number(u16),
  Label(String),
  Register(u16),

  EOF,
  NEWLINE,
  COMMA,

  LPAREN,
  RPAREN,
  LSQUARE,
  RSQUARE,

  BEQ,
  AND,
  XOR,
  \\[...]
  JAL,
```

```
  PUSH,
  POP,
}
```

In order to meet one of my objectives, my assembler and compiler both require pretty-printing of error messages. This means keeping track of the position of each and every token in source code. I created a generic `Span<T>` wrapper struct which takes in a generically typed parameter (and therefore can be reused both for lexer tokens and parser nodes). Each `Span` references a `Loc`, which itself stores the start position and length of a particular token. Since all source locations are stored as a one dimensional index, the `get_pos()` function converts this into a line and column number, meaning errors can point to specific characters within the source code.

```
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub struct Loc {
  pub pos: usize,
  pub len: usize,
}

impl Loc {
  pub fn new(s_pos: usize, e_pos: usize) -> Self {
    Loc {
      pos: s_pos,
      len: e_pos - s_pos + 1,
    }
  }

  pub fn get_pos(&self, program: &str) -> Option<(usize, usize)> {
    let mut pos = self.pos;
    for (line_number, line) in program.lines().enumerate() {
      if pos <= line.len() + 1 {
        return Some((line_number, pos));
      }

      // + 1 accounts for newline characters not included in len()
      pos -= line.len() + 1;
    }

    None
  }
}

impl Add for Loc {
```

```
  type Output = Loc;

  fn add(self, rhs: Loc) -> Loc {
    Loc {
      pos: self.pos,
      len: rhs.pos + rhs.len - self.pos,
    }
  }
}

#[derive(Clone, PartialEq, Eq)]
pub struct Span<T> {
  pub v: T,
  pub loc: Loc
}

impl<T> Span<T> {
  pub fn new(t: T, s: Loc) -> Span<T> {
    Span {
      v: t,
      loc: s,
    }
  }
}

// helper functions to simplify memory management when
// wrapping data types with the Span struct
impl<T: Copy> Copy for Span<T> {}

impl<T> Deref for Span<T> {
  type Target = T;
  fn deref(&self) -> &T {
    &self.v
  }
}

impl<T> DerefMut for Span<T> {
  fn deref_mut(&mut self) -> &mut T {
    &mut self.v
  }
}

impl<T: fmt::Debug> fmt::Debug for  Span<T> {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "{:?}", self.v)
  }
```

```
}
```

I have represented the source code as a linked list of characters inside the lexer, through which it can advance, peek and retreat. Whenever the lexer encounters a character, it enters a switch statement. Could the character form part of a longer, multicharacter spanning token (e.g. '+=' or '!=') - the next character in the source code will be viewed to determine whether to return a single character token, or advance another place in the source code and combine them. e.g. should a '+' be encountered, and the next character in the source is '=' they would be tokenised together as `ADD_EQ` rather than `ADD`.

Should the lexer encounter an alphabetic character (`read_identifier()`), it will continue scanning the source code while it encounters alphanumeric characters, appending each characeter to an identifier string. Once the string has been built up from all consecutive alphanumeric characters, the lexer will determine whether it is a label or mneumonic. Mneumonics are represented by their own tokens - wheras labels are all under the banner of a `LABEL` token.

Should a numerical character be encountered (`read_number()`), the lexer will first determine whether a prefix such as '0x' or '0b' has been encountered. Once the lexer knows which base it expects the number to be in, it continues iterating whilst it encounters a valid digit, and multiplies the digit by its base and offset from the start, adding it to a running total that represents the base 10 of the tokenized number.

```
pub struct Lexer<'a> {
  source: Chars<'a>,
  pos: usize,
  ch: char,
}

impl<'a> Lexer<'a> {
  pub fn new(mut source: Chars<'a>) -> Self {
    Self {
      ch: source.next().unwrap(),
      pos: 0,
      source,
    }
  }

  // advance the pointer through the source code,
  // self.ch is set to a null byte at the end of the file
  pub fn eat(&mut self) -> char {
    self.pos += 1;
    self.ch = self.source.next().unwrap_or('\0');
    self.ch
  }

  // advance only if the next character is expected
```

```
  // used to lex multi-character tokens e.g. !, !=
  pub fn eat_if(&mut self, ch: char) -> bool {
    self.peek() == ch && { self.eat(); true }
  }

  // return a copy of the next character in the source code
  pub fn peek(&self) -> char {
    self.source.clone().next().unwrap_or('\0')
  }

  pub fn tokenize(source: Chars<'a>) -> Vec<Span<Token>> {
    let mut lexer = Lexer::new(source);
    let mut tokens: Vec<Span<Token>> = Vec::new();

    while lexer.ch != '\0' {
      let s_pos = lexer.pos;
      if let Some(token) = lexer.tokenize_char() {
        tokens.push(Span::new(token, Loc::new(s_pos, lexer.
          ↪ pos)));
      }
      lexer.eat();
    }

    // terminate lexer output with EOF token
    tokens.push(Span::new(
      Token::EOF,
      Loc::new(lexer.pos, lexer.pos)
    ));

    tokens
  }

  pub fn tokenize_char(&mut self) -> Option<Token> {
    match self.ch {
      '(' => Some(Token::LPAREN),
      ')' => Some(Token::RPAREN),
      '{' => Some(Token::LBRACE),
      '}' => Some(Token::RBRACE),
      ',' => Some(Token::COMMA),
      '^' => Some(Token::XOR),
      ':' => Some(Token::COLON),
      ';' => Some(Token::SEMICOLON),

      '+' => Some(
        if self.eat_if('+') { Token::INC }
        else if self.eat_if('=') { Token::ADDEQ }
```

```
        else { Token::PLUS }
      ),

      '=' => Some(if self.eat_if('=') { Token::EE } else { Token::EQ }),
      '<' => Some(if self.eat_if('=') { Token::LTE } else { Token::LT }),

      ch if ch.is_ascii_digit() => self.tokenize_number(),
      ch if ch.is_alphabetic() => self.tokenize_identifier(),
      ch if ch == '\'' => self.tokenize_char_literal(),

      ch if ch.is_whitespace() => None,

      _ => fatal_at!(
        format!("Syntax Error: unexpected character in lexer {:?}", self.ch),
        Loc::new(self.pos, self.pos)
      ),
    }
  }

  fn tokenize_char_literal(&mut self) -> Option<Token> {
    let ch = self.eat();
    if !self.ch.is_ascii() || self.peek() != '\'' {
      fatal_at!("Syntax Error: invalid character literal", Loc::new(self.pos - 1, self.pos - 1))
    }

    self.eat();
    Some(Token::Char(ch))
  }

  fn tokenize_identifier(&mut self) -> Option<Token> {
    let identifier = self.read_identifier();
    if let Some(tok) = Token::from_identifier(&identifier) {
      Some(tok)
    } else {
      Some(Token::Identifier(identifier))
    }
  }

  fn tokenize_number(&mut self) -> Option<Token> {
    // should the prefix 0x or 0b be encountered, parse the subsequent
    // digit string into an integer with base 16, 2 or 10
```

```
        respectively

    let number = match (self.ch, self.peek()) {
      ('0', 'x') => {
        self.eat();
        self.eat();
        self.read_number(16)
      },
      ('0', 'b') => {
        self.eat();
        self.eat();
        self.read_number(2)
      },
      _ => self.read_number(10)
    };

    Some(Token::Number(number))
  }

  fn read_number(&mut self, base: u32) -> u16 {
    let mut sum: u16 = self.ch.to_digit(base).unwrap_or_else(||
      fatal_at!(
        "Syntax Error: expected digit after base prefix",
        Loc::new(self.pos, self.pos)
      )) as u16;

    // convert the next character into an integer provided it is a digit
    // of the correct base and shift the previous total 1 place to the left
    // (sum * base) and add the newly parsed digit
    while let Some(n) = self.peek().to_digit(base) {
      sum = (sum * base as u16) + n as u16;
      self.eat();
    }

    sum
  }

  fn read_identifier(&mut self) -> String {
    let mut identifier = String::from(self.ch);
    while self.peek().is_alphanumeric() || self.peek() == '_' {
      self.eat();
      identifier += &self.ch.to_string();
```

```
    }

    identifier
  }
}
```

### 3.3.1 Parser

Once the source code has been tokenized, the parser needs to convert it into a series of data structures that can be easily compiled down into machine code. Each assembly program is composed of statements, a statement can be either a label declaration or an instruction. Each instruction can take one of two formats, I-format or R-format, each of which I have represented as an type in an Instruction enum.

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Statement {
  Label(String),
  Instruction(Instruction),
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Instruction {
  IFormat {
    mneumonic: Span<Token>,
    rs: Register,
    rt: Register,
    immediate: Span<Immediate>,
  },

  RFormat {
    mneumonic: Span<Token>,
    rs: Register,
    rt: Register,
    rd: Register,
  },
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Register(pub u16);

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum Immediate {
  Label(String),
  Number(u16),
```

```
}
```

The parser iterates through the tokens generated by the lexer, and depending on the mneumonic encountered, expects to find a different instruction format.

```
pub fn parse(&mut self) -> Result<Vec<Span<Statement>>, Exception> {
  let mut statements: Vec<Span<Statement>> = Vec::new();

  while *self.tok != Token::EOF {
    let statement : Span<Statement> = match &*self.tok {
      Token::Label(label) => {
        Span::new(
          Statement::Label(label.clone()),
          self.tok.loc,
        )
      }

      Token::ADD | Token::SUB => self.parse_rrr_format()?,
      Token::BEQ  | Token::BNE => self.parse_rri_format()?,
      Token::SW | Token::LW => self.parse_rir_format()?,

      _ => return Err(Exception::new(
        format!("Syntax Error: unexpected token encountered, expected LABEL or INSTRUCTION, got '{:?}'", *self.tok),
        self.tok.loc,
      ))
    };

    statements.push(statement);
    self.eat();
  }

  Ok(statements)
}
```

I write two functions to parse the different formats of operand, registers and immediates respectively. A register can optionally be encased in square brackets as is convention when addressing memory, however is handled the same regardless. An immediate can take two formats, either a 16 bit number, or a label contained within square brackets. The function switches based on whether it encounters a square bracket or number (throwing an unexpected operand exception otherwise) and parses each situtation seperately.

```rust
// [.label], n16
fn parse_immediate(&mut self) -> Result<Span<Immediate>, Exception> {
  self.eat();

  match *self.tok {
    Token::Number(n) => Ok(Span::new(
      Immediate::Number(n),
      self.tok.loc
    )),

    Token::LSQUARE => {
      let label = self.eat_expect(
        Token::Label(String::new())
      )?.try_into().unwrap();

      self.eat_expect(Token::RSQUARE)?;
      Ok(label)
    }

    _ => Err(Exception::new(
      format!("Syntax Error: unexpected token, expected Number or '[Label]', got: '{:?}'", *self.tok),
      self.tok.loc
    ))
  }
}

// $rx, [$rx]
fn parse_register(&mut self) -> Result<Register, Exception> {
  self.eat();
  match *self.tok {
    Token::Register(reg) => Ok(Register(reg)),
    Token::LSQUARE => {
      let register = self.eat_expect(Token::Register(0))?
        .try_into().unwrap();

      self.eat_expect(Token::RSQUARE)?;
      Ok(register)
    }

    _ => Err(Exception::new(
      format!("Syntax Error: unexpected token, expected Register or '[Register]', got: '{:?}'", *self.tok),
```

```
      self.tok.loc
    ))
  }
}
```

When a particular mneumonic is encountered in the top level of the parser, it looks up the corresponding instruction format and calls the corresponding function. Each parsing function uses the `parse_register()` and `parse_immediate()` functions to handle the operands. Should any expected tokens be missing (e.g. a comma), a syntax error is thrown by the `eat_expect()` function. Finally, an Instruction Statement is outputted with a `Span` container that points to the start and end of the instruction in source code.

```
// addi $rt, $rs, [label] | immediate
fn parse_rri_format(&mut self) -> Result<Span<Statement>, Exception> {
  let mneumonic = self.tok.clone();

  let rt = self.parse_register()?;
  self.eat_expect(Token::COMMA)?;

  let rs = self.parse_register()?;
  self.eat_expect(Token::COMMA)?;

  let immediate = self.parse_immediate()?;

  Ok(Span::new(
    Statement::Instruction(Instruction::IFormat {
      mneumonic: mneumonic.clone(),
      rs,
      rt,
      immediate
    }),
    self.tok.loc - mneumonic.loc
  ))
}
```

### 3.3.2 Code Generation

The compiler runs on a two-pass basis. The first pass is responsible for storing the numerical offsets for each label in the symbol table, and expanding macro instructions into one or more primative instructions. The second pass is responsible for translating the instruction objects into binary machine code. The first pass creates a vector to store all primative (expanded) machine code instructions. It then iterates through the parsed program, keeping an offset of the number of instruction encountered. When a label is encountered, a new entry in the Symbol table is created, mapping the label identfier to the instruction offset. This can be used in the second pass when calculating jump addresses. Should an instruction be countered, the `expand_macro()` function is called. This function returns a list of expanded instructions that is appended to the `machine_instructions()` vector. The `expand_macro()` instruction first determines whether a mneumonic belongs to a primative or macro instruction. If it is a primative instruction, that instruction is returned alone. Else, it uses the template defining each macro instruction to determine the sequence of instructions to return.

```
fn first_pass(&mut self) -> Result<Vec<Span<Instruction>>, Exception> {
  let mut machine_instructions: Vec<Span<Instruction>> = Vec::new();

  let mut offset = 0;
  for stmt in self.statements.clone() {
    match &*stmt {
      // store offset to label in a symbol table
      Statement::Label(label) => {
        self.symbol_table.insert(label.clone(), offset);
      }

      Statement::Instruction(..) => {
        let mut expanded_macro = self.expand_macro(
          stmt.as_instruction().unwrap()
        )?;

        // increment instruction counter by the length of
        // the expanded macro instruction

        offset += (expanded_macro.len() * 2) as u16;
        machine_instructions.append(&mut expanded_macro);
      }
    }
  }

  Ok(machine_instructions)
}
```

On the second pass of the compiler, the binary representation for each field in the machine code needs to be calculated. For a register, this is simply the register number (stored in the `.0` field on the register struct). For the opcode or func fields, these are properties of the mneumonic keyword token. And the immediate representation is calculated by the `compile_immediate()` function, which determines whether the operand is a label or immediate value. Should it be a label, either a relative or absolute offset is calculated using the label and current instruction address.

```
pub fn compile_instruction(&mut self, instr: Span<Instruction>) -> Result<u32, Exception> {
  match &*instr {
    Instruction::IFormat {
      mneumonic,
      rs,
      rt,
      immediate,
    } => {
      let opcode = mneumonic.get_opcode()?;
      let rs = rs.0 as u32;
      let rt = rt.0 as u32;
      let immediate = self.compile_immediate(
        &immediate,
        // set the is_offset flag for B operations
        **mneumonic == Token::BEQ || **mneumonic == Token::BNE,
      )?;

      // combine fields into instruction
      Ok(((opcode) << 28) | (rs << 23) | (rt << 18)
        | (immediate << 2))
    }

    Instruction::RFormat {
      mneumonic,
      rs,
      rt,
      rd,
    } => {
      let opcode = mneumonic.get_opcode()?;
      let func = mneumonic.get_func()?;
      let rs = rs.0 as u32;
      let rt = rt.0 as u32;
      let rd = rd.0 as u32;

      // combine fields into instruction
      Ok(((opcode) << 28) | (rs << 23) | (rt << 18)
        | (rd << 13) | (func << 9))
    }
  }
}

pub fn compile_immediate(
  &self,
```

```
  immediate: &Span<Immediate>,
  as_offset: bool,
) -> Result<u32, Exception> {
  match &**immediate {
    Immediate::Label(label) => {
      // lookup label in symbol table
      if let Some(address) = self.symbol_table.get(&*label.clone()) {
        // if offset, calc relative distance from label
        // to the current instruction, else return the address itself
        Ok(if as_offset {
          ((*address - self.word) / 2 - 1) as u32
        } else {
          *address as u32
        })
      } else {
        Err(Exception::new(
          format!("Syntax Error: undefined label {:?}", *label),
          immediate.loc,
        ))
      }
    }

    Immediate::Number(number) => Ok(*number as u32),
  }
}
```

## 3.4 Compiler

### 3.4.1 Parser

Once the source code has been tokenised by the lexer (which can be reused from the assembler), the tokens need to be parsed into an abstract syntax tree following the grammar rules defined in the parser specification in the design section. I encapsulated the parser code in a struct, containing the list of tokens, current token, current position and current scope. The current scope is unique for each Block the parser parses, and is used to reference a particular block scope in the Scope Table.

```
pub struct Parser {
  tokens: Vec<Span<Token>>,
  tok: Span<Token>,
  pos: usize,
  scope: u32,
```

```
}

impl Parser {
  pub fn new(tokens: Vec<Span<Token>>) -> Self {
    Self {
      tok: tokens.get(0).unwrap().clone(),
      tokens: tokens,
      pos: 0,
      scope: 0,
    }
  }

  pub fn parse(tokens: Vec<Span<Token>>) -> Vec<Span<SYMBOL>>
    {
    let mut parser = Parser::new(tokens);
    let mut source = vec![];

    while *parser.tok != Token::EOF {
      source.push(parser.parse_symbol());
    }

    source
  }
}
```

The parser outputs an Abstrat Syntax Tree composed of nodes, each node representing a program element. The formal grammar that dictates the nodes was designed in the Design section of the pratt parser.

```
#[derive(Debug, Clone, Eq, PartialEq)]
pub enum SYMBOL {
  Const {
    binding: Span<Binding>,
    expr: Box<Span<EXPRESSION>>,
  },

  Function {
    ident: Span<Identifier>,
    bindings: Vec<Span<Binding>>,
    ret_ty: TYPE,
    body: Box<Span<Block>>,
  }
}

#[derive(Debug, Clone, PartialEq, Eq)]
```

```
pub enum STATEMENT {
  Declaration {
    binding: Span<Binding>,
    expr: Box<Span<EXPRESSION>>,
  },

  If {
    cond: Box<Span<EXPRESSION>>,
    conseq: Box<Span<Block>>,
    altern: Option<Box<Span<Block>>>
  },

  While {
    cond: Box<Span<EXPRESSION>>,
    body: Box<Span<Block>>,
  },

  For {
    init: Box<Span<STATEMENT>>,
    cond: Box<Span<EXPRESSION>>,
    inc: Box<Span<EXPRESSION>>,
    body: Box<Span<Block>>,
  },
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum EXPRESSION {
  Literal { lit: Span<LITERAL> },

  Call {
    func: Box<Span<EXPRESSION>>,
    args: Vec<Span<EXPRESSION>>,
  },

  Infix {
    lhs: Box<Span<EXPRESSION>>,
    op: BINOP,
    rhs: Box<Span<EXPRESSION>>,
  },

  Prefix {
    op: UNOP,
    rhs: Box<Span<EXPRESSION>>
  },
}

```

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum LITERAL {
  Int(u16),
  Char(char),
  Bool(bool),
}

impl LITERAL {
  // resolve the numerical value of the literal
  pub fn unwrap(&self) -> u16 {
    match self {
      LITERAL::Char(ch) => *ch as u16,
      LITERAL::Bool(b) => *b as u16,
      LITERAL::Int(n) => *n,
    }
  }

  // resolve the data type of the Literal
  pub fn ty(&self) -> TYPE {
    match self {
      LITERAL::Char(_) => TYPE::CHAR,
      LITERAL::Bool(_) => TYPE::BOOL,
      LITERAL::Int(_) => TYPE::INT,
    }
  }
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Block {
  pub stmts: Vec<Span<STATEMENT>>,

  // reference to block scope in the Scope Table
  pub scope: ScopeId,
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Binding {
  pub ident: Span<Identifier>,
  pub ty: TYPE,
}
```

There are two functions to help iterate through the source tokens, an `eat()` function and an `assert()`. The eat function advances one token, setting the values of the `self.tok` and `self.pos` fields. The assert function takes an expected token as a parameter and throws a syntax error if the current token is not the same.

```rust
pub fn eat(&mut self) {
  if self.pos + 1 < self.tokens.len() {
    self.pos += 1;
    self.tok = self.tokens.get(self.pos).unwrap().clone();
  }
}

pub fn assert(&mut self, expect: Token) {
  if *self.tok != expect {
    fatal_at!(
      format!("Syntax Error: expected {:?}, got {:?}", expect
          ↪ , *self.tok),
      self.tok.loc
    )
  }
}
```

For each iteration of the `parse()` loop, the `parse_symbol()` function is called, generating a node in the Abstract Syntax Tree which is appended to the `source` vector. The `parse_symbol()` function throws an error if the statement is neither a function declaration or constant, otherwise calls their respective parsing functions.

```rust
fn parse_symbol(&mut self) -> Span<SYMBOL> {
  match *self.tok {
    Token::FN => self.parse_fn(),
    Token::CONST => self.parse_const(),
    _ => fatal_at!(
      format!(
        "Syntax Error: expected 'fn' or 'const', got {:?}",
        *self.tok
      ),
      self.tok.loc
    ),
  }
}
```

When parsing the parameters of a function, I created a vector to hold the parameter bindings. After the parser encounters a '(', it continues to parse bindings while it encounters a comma, breaking once the ')' character has been reached.

```rust
// FN <ident> LPAREN (<ident>: <type>)* RPAREN (-> <type>)? <
    ↪ block>
fn parse_fn(&mut self) -> Span<SYMBOL> {
```

```
  let s_pos = self.tok.loc;

  self.eat();
  let ident = self.parse_identifier();

  self.assert(Token::LPAREN);
  self.eat();

  let mut bindings: Vec<Span<Binding>> = Vec::new();
  while *self.tok != Token::RPAREN {
    bindings.push(self.parse_binding());

    if *self.tok != Token::COMMA {
      break;
    }

    self.eat();
  }

  self.eat();

  // if a '->' token follows the parameters, parse the return type,
  // else default to a VOID type
  let ret_ty = if *self.tok == Token::ARROW {
    self.eat();
    self.parse_type()
  } else {
    TYPE::VOID
  };

  let body = self.parse_block();

  Span::new(
    SYMBOL::Function {
      ident,
      bindings,
      ret_ty,
      body: Box::new(body),
    },
    s_pos + self.tok.loc,
  )
}

// CONST <ident>: <type> = <expr>;
fn parse_const(&mut self) -> Span<SYMBOL> {
```

```
  let s_pos = self.tok.loc;

  self.eat();

  let binding = self.parse_binding();
  self.assert(Token::EQ);
  self.eat();

  let expr = self.parse_expression();
  self.assert(Token::SEMICOLON);
  self.eat();

  Span::new(
    SYMBOL::Const {
      binding: binding,
      expr: Box::new(expr),
    },
    s_pos + self.tok.loc,
  )
}
```

A block consists of a sequence of semi colon terminated satements contained between two curly braces.

```
fn parse_block(&mut self) -> Span<Block> {
  let s_pos = self.tok.loc;
  self.assert(Token::LBRACE);
  self.eat();

  let mut stmts: Vec<Span<STATEMENT>> = Vec::new();
  while *self.tok != Token::RBRACE {
    stmts.push(self.parse_statement());
  }

  self.eat();

  // assign a unique scope to the block at creation
  Span::new(
    Block {
      stmts,
      scope: self.next_scope()
    },
    s_pos + self.tok.loc
  )
}
```

When entering the `parse_statement()` function, use the keyword token to determine how to parse the subsequence tokens. Should a statement have no keyword preceeding it, it is parsed as an expression terminated by a semi colon.

```
fn parse_statement(&mut self) -> Span<STATEMENT> {
  match *self.tok {
    Token::LET => self.parse_declaration(),
    Token::RETURN => self.parse_return(),
    Token::IF => self.parse_if(),
    Token::WHILE => self.parse_while(),
    Token::FOR => self.parse_for(),

    _ => {
      let expr = self.parse_expression();
      self.assert(Token::SEMICOLON);
      self.eat();
      Span::new(
        STATEMENT::Expression {
          expr: Box::new(expr.clone()),
        },
        expr.loc,
      )
    }
  }
}

// while <expression> <block>
fn parse_while(&mut self) -> Span<STATEMENT> {
  let s_pos = self.tok.loc;
  self.eat();
  let cond = self.parse_expression();
  let body = self.parse_block();

  Span::new(
    STATEMENT::While {
      cond: Box::new(cond),
      body: Box::new(body)
    },
    s_pos + self.tok.loc
  )
}

// if <condition> <block> (else (<block> | <if>))?
```

```
fn parse_if(&mut self) -> Span<STATEMENT> {
  let s_pos = self.tok.loc;

  self.eat();
  let cond = self.parse_expression();
  let conseq = self.parse_block();
  let mut altern = None;

  // parse alternative if an else block exists
  // else set altern to None
  if *self.tok == Token::ELSE {
    self.eat();
    // if ELSE IF, then recursively parse the if block as the
    // alternative attribute, otherwise parse a block
    if *self.tok == Token::IF {
      altern = Some(Box::new(
        Span::new(Block {
          stmts: vec![self.parse_if()],
          scope: self.next_scope(),
        }, s_pos + self.tok.loc)
      ));

      self.scope += 1;
    } else {
      altern = Some(Box::new(self.parse_block()));
    }
  }

  Span::new(
      STATEMENT::If {
        cond: Box::new(cond),
        conseq: Box::new(conseq),
        altern
      },
      s_pos + self.tok.loc
  )
}
```

The parse_expression() handles expressions ranging from arithmetic operations to variable assignments. It parses the left hand side of any operation as a full pratt expression, meaning statements like &(vram + 0x0b) = 10 can be represented with arithmetic operations on the left hand side of an assignment. It then determines whether the folowing token is an '=' or increment operation such as '+=', if it encounters one of the two, it is an assignment operation and handled by the parse_assign() function, else it simply returns the expression itself. The parse_assign() function simply parses the right hand side as another pratt expression and returns the result in an `EXPRESSION::Assign` enum variant.

```
fn parse_expression(&mut self) -> Span<EXPRESSION> {
  let expr = match *self.tok {
    _ => self.parse_pratt(0),
  };

  match &*self.tok {
    Token::EQ => self.parse_assign(expr),
    tok if BINOP::from_assign_op(tok.clone()).is_some() => self.parse_assign_op(expr),
    _ => expr,
  }
}

fn parse_assign(&mut self, lhs: Span<EXPRESSION>) -> Span<EXPRESSION> {
  let s_pos = self.tok.loc;
  self.eat();
  let rhs = self.parse_expression();

  Span::new(
    EXPRESSION::Assign {
      lhs: Box::new(lhs),
      rhs: Box::new(rhs)
    },
    s_pos + self.tok.loc
  )
}
```

#### 3.4.1.1 Pratt Parser

Below is the impelmentation of the Pratt parsing algorithm designed in the previous section. I extrapolated the prefix code into the `parse_atom()` function.

```
fn parse_pratt(&mut self, rbp: i32) -> Span<EXPRESSION> {
  let s_pos = self.tok.loc;

  // parse LHS, including prefixes (parenthesis, literals)
  // e.g. ++x, (10+2), &x
  let mut lhs = self.parse_atom();


  // parse all postfix operations for the LHS
```

```
  // f(), a[], x++
  while let Some(op) = POSTOP::from((*self.tok).clone()) {
    self.eat();
    lhs = Span::new(
      EXPRESSION::Postfix { lhs: Box::new(lhs), op },
      s_pos + self.tok.loc,
    );
  }

  // continue parsing into the RHS of the expression while
  // binary operations of a higher precedence are encountered
  while let Some(op) = BINOP::from((*self.tok).clone()) {
    if op.get_precedence() < rbp {
      break;
    }

    self.eat();
    let rhs = self.parse_pratt(op.get_precedence() + 1);

    // set LHS to newly parsed expression so it can be built
    // off of each iteration
    lhs = Span::new(
      EXPRESSION::Infix {
        lhs: Box::new(lhs),
        op: op,
        rhs: Box::new(rhs),
      },
      s_pos + self.tok.loc,
    );
  }

  lhs
}
```

The `parse_atom()` function handles parenthesis that contain a regular pratt expression. Implementing this as an atom allows the order of operations to be followed since whatever is contained within the brackets will be evaluated before any superceeding operations. It also handles unary operations that come before an expression, e.g. (`-10`, `&x`). Each prefix operation is given a binding precedence, and this binding precedence is used as the base precedence for the `parse_pratt_expression()` call, preventing `*x - 4` being parsed as `DEREF(x - 4)` instead of `DEREF(x) - 4`.

```
fn parse_atom(&mut self) -> Span<EXPRESSION> {
  let s_pos = self.tok.loc;

```

```
  let expr = match &*self.tok.clone() {
    Token::LPAREN => {
      self.eat();
      let expr = self.parse_expression();
      self.assert(Token::RPAREN);
      self.eat();
      expr
    }

    tok if UNOP::from(tok.clone()).is_some() => {
      let prefix = UNOP::from((*self.tok).clone()).unwrap();
      self.eat();
      let rhs = self.parse_pratt(prefix.get_precedence());

      Span::new(
        EXPRESSION::Prefix {
          op: prefix,
          rhs: Box::new(rhs)
          },
          s_pos + self.tok.loc
       )
    }

    _=> self.parse_literal(),
  };

  // if '(' after atom, parse as a function call
  match *self.tok {
    Token::LPAREN => self.parse_call(expr),
    _ => expr
  }
}
```

The most fundemental unit of the abstract syntax tree is the program literal. Each one corresponds to a single token therefore all the `parse_literal()` function has to do is map the token to its corresponding expression literal.

```
fn parse_literal(&mut self) -> Span<EXPRESSION> {
  let s_pos = self.tok.loc;
  let expr = match &*self.tok {
    Token::Number(n) => EXPRESSION::Literal {
      lit: Span::new(LITERAL::Int(*n),
      self.tok.loc)
    },

```

```
    Token::Char(ch) => EXPRESSION::Literal {
      lit: Span::new(LITERAL::Char(*ch),
      self.tok.loc)
    },

    Token::TRUE => EXPRESSION::Literal {
      lit: Span::new(LITERAL::Bool(true), self.tok.loc)
    },

    // [...]

    _ => fatal_at!(
      format!(
        "Syntax Error: expected program literal, got {:?}",
        *self.tok
      ),
      self.tok.loc
    ),
  };

  self.eat();
  Span::new(expr, s_pos)
}
```

Finally, the last component when parsing a program is to handle the function calls. They are composed of an identifier, followed by '(' with 0 or more arguments and a terminating ')'. The arguments consist of a series of expressions seperated by a comma. Iteratively parsing expressions whilst a comma is encountered, and storing the result in a vector is sufficient to collect the arguments which are passed as an attribute into the `EXPRESSION::Call` variant.

```
fn parse_call(&mut self, func: Span<EXPRESSION>) -> Span<EXPRESSION> {
  let s_pos = self.tok.loc;

  self.assert(Token::LPAREN);
  self.eat();

  // continue collecting arguments while a ',' is encountered
  // and before the terminating ')'
  let mut args: Vec<Span<EXPRESSION>> = Vec::new();
  while *self.tok != Token::RPAREN {
    args.push(self.parse_expression());

    if *self.tok != Token::COMMA {
      break;
```

```
    }

    self.eat();
  }

  self.eat();
  Span::new(EXPRESSION::Call {
    func: Box::new(func),
    args: args
  }, s_pos + self.tok.loc)
}
```

### 3.4.2 Optimisations and Sentiment Analysis

#### 3.4.2.1 AST Traveral

The optimisations require the ability to traverse every node in the abstract syntax tree. I created a trait to implement a depth first search traversal that could be itself implemented by the optimisation structs. The `Walker` trait contains `walk()` functions that recursively iterate over all child nodes for a particular node in the abstract syntax tree, and call abstract `visit()` methods that are themselves overwritten in the optimisation code to handle that particular node. The `walk()` method is called on the root node of the AST, and the Walker trait recursively visits each child node, calling the corresponding `visit()` method for each.

```
pub trait Walker {

fn walk(&mut self, ast: &mut Vec<Span<SYMBOL>>) {
  for sym in ast {
    self.visit_symbol(sym);
  }
}

fn walk_symbol(&mut self, symbol: &mut Span<SYMBOL>) {
    match &mut **symbol {
      SYMBOL::Const { binding, expr } => {
        self.visit_binding(binding);
        self.visit_expression(&mut **expr);
      },

      SYMBOL::Function { ident, bindings, ret_ty, body } => {
        self.visit_identifier(&mut *ident);
        for binding in bindings {
          self.visit_binding(&mut *binding);
        }

```

```
            self.visit_type(&mut *ret_ty);
            self.visit_block(body);
        }
    }
}

fn walk_block(&mut self, block: &mut Span<Block>) {
    for stmt in &mut block.stmts {
        self.visit_statement(stmt);
    }
}

//[...]
}
```

#### 3.4.2.2 Const Folding

The first optimisation my compiler performs involves pre calculating the result of expressions and storing the result as a node in place of the infix node. The walker visits each node in the abstract syntax tree. If it encounters an infix node, it recursively calls the `fold_constant()` function on the left and right hand side of the expression. The fold constants function checks if the lhs and rhs have been evaluated as constants, if they have - it performs the calculation and stores the result in a Literal node.

```
pub struct ConstFolding;
impl ConstFolding {

pub fn run(ast: &mut Vec<Span<SYMBOL>>) {
    let mut cf = ConstFolding;
    cf.walk(ast);
}

fn fold_constant(&mut self, e: &Span<EXPRESSION>)
    -> Option<Span<EXPRESSION>> {
    match &**e {
        EXPRESSION::Literal { .. } => return Some(e.clone()),

        // recursively fold the lhs and rhs of an expression
        // if both fold to constants, combine them and
        // replace the current node
        EXPRESSION::Infix { lhs, op, rhs } => {
            if let (
                Some(EXPRESSION::Literal { lit: lhs }),
                Some(EXPRESSION::Literal { lit: rhs }))
            = (self.fold_constant(lhs), self.fold_constant(rhs)) {
```

```
        let result = match op {
          BINOP::ADD => Some(LITERAL::Int(lhs + rhs)),
          BINOP::MUL => Some(LITERAL::Int(lhs * rhs)),
          BINOP::DIV => Some(LITERAL::Int(lhs / rhs)),
          BINOP::OR  => Some(LITERAL::Int(lhs | rhs)),
          BINOP::XOR => Some(LITERAL::Int(lhs ^ rhs)),
          BINOP::AND => Some(LITERAL::Int(lhs & rhs)),
          _ => None,
        };

        // return the folded literal node
        if let Some(lit) = result {
          return Some(Span::new(
            EXPRESSION::Literal {
              lit: Span::new(lit, e.loc)
            }, e.loc
          ))
        }
      } else {
        return self.fold_constant(lhs)
      }
    },

    _ => {},
  }

  None
}
}

// Walker visits each node in the AST, visit_expression()
// is called on each infix node to fold the lhs and rhs
impl Walker for ConstFolding {
  fn visit_expression(&mut self, e: &mut Span<crate::ast::EXPRESSION>) {
    if let Some(lit) = self.fold_constant(e) {
      *e = lit;
    } else {
      self.walk_expression(e);
    }
  }
}
```

#### 3.4.2.3 Type Checking

The type checker visits every node in the AST and uses the `resolve()` function to determine what data type the child nodes evaluate into. This process uses the symbol table (a HashMap containing all the variables, constants, and function declarations accessible within a particular scope) to resolve the type of variables. For an infix expression, `resolve()` is called twice, for the lhs and rhs respectively, and their types are compared to determine compatability.

```rust
pub struct Typeck<'a> {
  // a reference to the symbol table generated after parsing
  symtbl: &'a mut SymbolTable,

  // contains the current scope from which to resolve variables
  scope: Option<ScopeId>,
}

impl<'a> Typeck<'a> {
  fn new(symtbl: &'a mut SymbolTable) -> Self {
    Self {
      symtbl: symtbl,
      scope: None,
    }
  }

  pub fn run(ast: &mut Vec<Span<SYMBOL>>, symtbl: &'a mut SymbolTable) {
    let mut typeck = Typeck::new(symtbl);
    typeck.walk(ast);
  }
}

impl<'a> Walker for Typeck<'a> {
  fn visit_symbol(&mut self, s: &mut Span<SYMBOL>) {
    match &**s {
      SYMBOL::Function {
        ret_ty,
        body,
        ..
      } => {
        self.resolve_block(body, Some(&ret_ty));
      }

      SYMBOL::Const { binding, expr } => {
        self.resolve_expression(expr, Some(&binding.ty));
```

```
      }
    };
  }
}
```

The following function is called to determine whether an encountered type is of the expected type.

```
fn verify(&self, got: &TYPE, expected: Option<&TYPE>, loc: Loc) -> TYPE {
  if let Some(expected) = expected {
    if got != expected {
      fatal_at!(
        format!(
          "Linking Error: expected type '{}', got '{}'",
          expected, got
        ),
        loc
      );
    }
  }

  got.clone()
}
```

```
fn resolve_statement(
  &mut self,
  stmt: &Span<STATEMENT>,
  expected: Option<&TYPE>) -> TYPE {

  let got = match &**stmt {

    // validate that the expression evaluates to
    // the binding type, return VOID since declarations
    // do not return a value
    STATEMENT::Declaration { binding, expr } => {
      self.resolve_expression(&**expr, Some(&binding.ty));
      TYPE::VOID
    },

    STATEMENT::If {
      cond,
      conseq,
```

```
      altern,
    } => {
      // ensure the condition evaluates to a boolean
      self.resolve_expression(cond, Some(&TYPE::BOOL));
      let conseq_ty = self.resolve_block(&**conseq, None);

      // ensure the if-then and else branches evaluate
      // to the same type
      if let Some(altern) = altern {
        let altern_ty = self.resolve_block(&**altern,
           ↪ expected);

        if altern_ty != conseq_ty {
          fatal_at!(
            format!("Linking Error: mismatched types")
            stmt.loc
          );
        }
      }

      conseq_ty
    }
  };

  self.verify(&got, expected, stmt.loc)
}
```

```
fn resolve_block(
  &mut self,
  block: &Span<Block>,
  expected: Option<&TYPE>) -> TYPE {

  // set current scope to the block scope
  self.scope = Some(block.scope.clone());
  let mut ret_ty: Option<TYPE> = None;

  for stmt in &block.stmts {
    let stmt_ty = self.resolve_statement(&stmt, None);

    // the only statement that doesn't return VOID is return
    // verify the return statement if of the correct type
    // and store as the return type for the block
    if stmt_ty != TYPE::VOID {
      self.verify(&stmt_ty, expected, stmt.loc);
```

```
      ret_ty = Some(stmt_ty);
    }
  }

  // now block has been evaluated, return to parent scope
  self.scope = self.symtbl.parent_scope(self.scope.unwrap());

  match ret_ty {
    Some(ty) => ty,
    _ => TYPE::VOID,
  }
}
```

```
fn resolve_expression(
  &mut self,
  expr: &Span<EXPRESSION>,
  expected: Option<&TYPE>) -> TYPE {

  let got = match &**expr {
    // base case for recursive expression
    EXPRESSION::Literal { lit } => lit.ty(),

    EXPRESSION::Infix { lhs, rhs , op}
      => self.resolve_infix(lhs, op, rhs),

    // [...]

    EXPRESSION::Assign { lhs, rhs } => {
      // ensure rhs and lhs are of the same type
      let lhs_ty = self.resolve_expression(lhs, None);
      self.resolve_expression(rhs, Some(&lhs_ty));
      TYPE::VOID
    }

    // lookup the variable in the current scope
    EXPRESSION::Variable { ident } => self
      .symtbl
      .resolve_variable(self.scope, ident)
      .expect("could not resolve variable").ty
  };

  self.verify(&got, expected, expr.loc)
}
```

```
fn resolve_call(
  &mut self,
  func: &Span<EXPRESSION>,
  args: &Vec<Span<EXPRESSION>>,
  loc: Loc,
) -> TYPE {

  // ensure a valid function call
  let ident = if let EXPRESSION::Variable { ident } = &**func {
    ident
  } else {
    fatal!("Linking Error: attempt to call non-identifier");
  };

  let (bindings, ret_ty) = self.symtbl.resolve_fn(ident).unwrap();

  // ensure f() is called with the correct number of arguments
  if args.len() != bindings.len() {
    fatal_at!(
      format!("Linking Error: mismatched argument count [...] ",
      args.len()),
    loc)
  }

  // iterate over corresponding pairs of parameters
  // and argsuments, ensuring they are of the same type
  for (arg, binding) in args.iter().zip(bindings.into_iter()) {
    self.resolve_expression(arg, Some(&binding.ty));
  }

  ret_ty
}
```

```
fn resolve_infix(
  &mut self,
  lhs: &Span<EXPRESSION>,
  op: &BINOP,
  rhs: &Span<EXPRESSION>,
```

```
) -> TYPE {
  let lhs_ty = self.resolve_expression(lhs, None);
  let rhs_ty = self.resolve_expression(rhs, None);

  match op {
    BINOP::ADD | BINOP::SUB  => {
      // verify the LHS and RHS are of compatible types
      // e.g. an int can be added to a pointer
      self.verify_compatible(
        &rhs_ty,
        vec![
          &TYPE::INT, &TYPE::CHAR,
          &TYPE::PTR(Box::new(TYPE::INT)),
          &TYPE::PTR(Box::new(TYPE::CHAR))
        ],
        rhs.loc
      );

      //[verify rhs]
    } ,


    BINOP::LT | BINOP::LTE | BINOP::GT  |
    BINOP::GTE | BINOP::EE | BINOP::NE  => {
      self.verify_compatible(
        &lhs_ty,
        vec![&TYPE::INT, &TYPE::CHAR],
        lhs.loc
      );

      // [verify rhs]

      // comparative operations return a bool regardless
      // of their input types
      TYPE::BOOL
    } ,

    // ensure the lhs and rhs are both bools
    BINOP::LAND | BINOP::LOR =>
      self.verify(&lhs_ty, Some(&TYPE::BOOL), lhs.loc)
  }
}
```

```
fn resolve_prefix(
```

```
  &mut self,
  op: &UNOP,
  rhs: &Span<EXPRESSION>
) -> TYPE {
  match op {
    // resolve the type of the expression within the ptr
    // create a ptr to that type, can be defined recursively
    UNOP::PTR => TYPE::PTR(
      Box::new(self.resolve_expression(rhs, None))
    ),

    UNOP::DEREF => {
      // only pointers can be derefed, else throw an error
      let ty = self.resolve_expression(rhs, None);
      if let TYPE::PTR(ty) = ty {
        *ty
      } else {
        fatal_at!(
            format!("Cannot dereference '{}'", ty),
            rhs.loc
        );
      }
    }
  }
}
```

#### 3.4.2.4 Scope Table Builder

The scope table contains the local scope for each Block expression contained within the program. Each scope is referenced by a unique `ScopeId`, and contains a `HashMap` mapping identifiers to variable bindings which can be used to resolve the data type or stack slot of a particular variable.

```
#[derive(Debug, Clone)]
pub struct SymbolTable {
  // symbols refer to constants and functions
  symbols: HashMap<Identifier, SYMBOL>,

  // local variables (stack assigned) are stored in scopes
  scopes: HashMap<ScopeId, Scope>
}

#[derive(Debug, Clone)]
pub struct Scope {
  pub variables: HashMap<Identifier, Variable>,
```

```
  pub parent: Option<ScopeId>,
}

#[derive(Debug, Clone)]
pub struct Variable {
  pub ty: TYPE,
  // offset from stack's base pointer in the stack frame
  pub stack_slot: Option<i16>,
}

#[derive(Debug, Clone, PartialEq, Eq, Hash, Copy)]
pub struct ScopeId(pub u32);
```

The `ScopeTableBuilder` iterates over each node in the program, when it enters a block, it first determines whether the block is a function scope or a block scope (function scopes have no parent scope). If it is a function scope, it registers a function scope in the SymbolTable, unique from a block scope since it correpsonds to a unique stack frame when the code is compiled. It sets the current scope of the `SymbolTableBuilder` to the Block's local scope, and recursively resolves all statemetns contained within.

```
fn visit_block(&mut self, b: &mut Span<Block>) {
  // create new Scope with curr Scope ID in Symbol Table
  self.symtbl.register_scope(&b.scope);

  // since functions have no parent scope, if this block has
  // no parent scope, define a new function scope, otherwise
  // block exists within a function (e.g. for or while)
  if let Some(parent) = &self.scope {
    self.symtbl.set_parent_scope(b.scope, *parent);
  } else {
    self.register_function_scope(b.scope);
  }

  let prev_scope = self.scope;
  self.scope = Some(b.scope);
  self.walk_block(b);
  self.scope = prev_scope;
}
```

Should the `Walker` encounter a declaration statement, it creates an entry in the local Block Scope `HashMap` mapping the identifier to its respective data type. And recursively resolves the rhs expression.

```
fn visit_statement(
```

```
  &mut self,
  s: &mut Span<crate::ast::STATEMENT>
) {
  if let STATEMENT::Declaration { binding, expr }
    = &mut **s {

    // register variable in local block scope
    self.symtbl.register_variable(
      self.scope.as_ref().unwrap(),
      &*binding
    );

    self.visit_expression(expr);
  }

  self.walk_statement(s);
}
```

Should the `Walker` encounter an expression, it determines whether the expression is a Call or a Variable. Should the expression be a variable, it uses the SymbolTable to determine whether the variable has been previously declared in the program before it was used. Else it throws an exception. Should it be a call, it determines whether the left hand side resolves to an identifier, and that that identifier corresponds to a predeclared function symbol.

```
fn visit_expression(
  &mut self,
  e: &mut Span<crate::ast::EXPRESSION>
) {
  match &mut **e {
    // if the expression is a call, determine whether
    // the identifier corresponds to a declared subroutine

    EXPRESSION::Call { func, args } => {

      // ensure the LHS is an identifier
      let ident = if let EXPRESSION::Variable { ident }
        = &mut ***func {
        ident
      } else {
        fatal_at!("Attempt to call a non-function")
      };

      // ensure identifier corresponds to a declared symbol
      let symbol = if let Some(symbol)
        = self.symtbl.lookup_symbol(ident) {
```

```
        symbol
      } else {
        fatal_at!("Attempt to call undeclared function")
      };

      // ensure the symbol is a function not a constant
      match symbol {
        SYMBOL::Function { .. } => {},
        _ => fatal_at!("Cannot call constant {ident}")
      }

      for arg in args.iter_mut() {
        self.visit_expression(arg);
      }

      // don't visit sub expressions
      return;
    },

    // ensure the variable is declared before usage
    EXPRESSION::Variable { ident } => {
      self.symtbl.resolve_variable(
        self.scope.unwrap(), ident).unwrap_or_else(||
        fatal_at!(
          format!("use of undeclared identifier '{ident}'"),
      ));
    },

    _ => {},
  };

  self.walk_expression(e);
}
```

### 3.4.3 Code Generation

Code generation is performed in two steps, the first step generates an intermediate representation of the parsed abstract syntax tree - consisting of a list of `BLOCK`'s each corresponding to machine code instructions. The second step iterates through each `BLOCK` and generates the machine code for that particular block, inserting labels where required. Each block doesn't neccessarily represent one machine code instruction, for example, `BLT` will expand into a `SLT` and `BNE` instruction.

```
#[derive(Debug, Clone)]
```

```
pub enum BLOCK {
  ADD(Register, Register, Register),
  SUB(Register, Register, Register),
  AND(Register, Register, Register),
  OR(Register, Register, Register),
  XOR(Register, Register, Register),

  ADDI(Register, Register, i16),

  EE(Register, Register, Register),
  NE(Register, Register, Register),
  LT(Register, Register, Register),
  GT(Register, Register, Register),
  LTE(Register, Register, Register),
  GTE(Register, Register, Register),

  NOR(Register, Register, Register),
  NOT(Register, Register),

  LABEL(Label),
  JAL(Register, Label),
  JMP(Label),
  JR(Register),
  HLT,

  BEQ(Register, Register, Label),
  BLT(Register, Register, Label),

  MOV(Register, Register),
  LI(Register, Number),
  LW(Register, Register, i16), // lw $rt, offset($rs)
  SW(Register, Register, i16), // sw rt, offset($rs)

  PUSH(Register),
  POP(Register),
}

#[derive(Debug, Clone)]
pub struct Number(pub u16);

#[derive(Debug, Clone)]
pub struct Label(pub String);

#[derive(Debug, Clone)]
pub struct Register(pub String);
```

The IR struct (wrapping the IR generating code) has a current function context reference, symbol table reference, hashmap for used labels, and an output vector of `BLOCK`'s. Since the generator reuses labels (e.g. `.endif`) and each label has to be unique, the IR stores the number of times each label has been used. This is then appended onto the label to generate a unique instance. (e.g. `.endif-2`).

The function context corresponds to the current stack frame the compiler is working under. This stores a return label, so any `return` instructions inside the function know where to exit the function. A current stack pointer pointing to the next free stack slot (offset from the base pointer), when anything is added onto the stack, it is stored in the memory location given by `stack_pointer` and `stack_pointer` is decremented (since the stack grows downwards). The function context also stores a reference to the block scope of the function, containing all local variables and their offsets from the base pointer.

```
pub struct Ir<'a> {
  ir: Vec<BLOCK>,
  fctx: Option<Fctx>,
  symtbl: &'a mut SymbolTable,
  labels: HashMap<String, u16>,
}

#[derive(Debug)]
struct Fctx {
  l_ret: Label,

  // offset from $bp (grows downwards,
  // thus negate when accessing memory)
  stack_pointer: i16,
  scope: ScopeId,
}
```

This method takes in a parsed function node, with an identifier, set of parameter bindings, and body; and outputs the sequence of intermediate instructions used to represent this. It creates a new function context and prepends the function prologue, setting up the stack frame for that particular function. It then assigns each parameter a slot on the stack from which it can be referenced. It recursively calls `translate_block()` to compile the function body, before inserting the function epiologue.

```
fn translate_fn(
  &mut self,
  ident: &Span<Identifier>,
  bindings: &Vec<Span<Binding>>,
  body: &Span<Block>
) {
  self.fctx = Some(Fctx{
```

```
    stack_pointer: -1, // since 0 points to $bp
    l_ret: self.label("ret"),
    scope: body.scope,
  });

  // function prologue
  //   .[ident]
  //   push $ra
  //   push $bp
  //   mov $bp, $sp
  self.ir.push(BLOCK::LABEL(Label(ident.ident.clone())));
  self.ir.push(BLOCK::PUSH(Register::from("$ra")));
  self.ir.push(BLOCK::PUSH(Register::from("$bp")));
  self.ir.push(BLOCK::MOV(
    Register::from("$bp"),
    Register::from("$sp")
  ));


  // register each binding in the local stack frame
  for (i, binding) in bindings.iter().enumerate() {
    let scope = self.fctx().scope;

    // evaluate argument and push to stack
    self.ir.push(BLOCK::LW(
      Register::from("$r0"),
      Register::from("$bp"),
      2 + i as i16)
    );

    self.push(Register::from("$r0"));

    let sp = self.fctx().stack_pointer + 1;

    // register parameter stack slot
    self.symtbl.set_slot(scope, &binding.ident, sp);
  }

  self.translate_block(body);

  // insert function epiologue
  //   .return
  //   mov $sp, $bp
  //   pop $bp
  //   pop $ra
  //   jr $ra
```

```
  let l_ret = self.fctx().l_ret.clone();

  self.ir.push(BLOCK::LABEL(l_ret));
  self.ir.push(BLOCK::MOV(
    Register::from("$sp"),
    Register::from("$bp")
  ));

  self.ir.push(BLOCK::POP(Register::from("$bp")));
  self.ir.push(BLOCK::POP(Register::from("$ra")));
  self.ir.push(BLOCK::JR(Register::from("$ra")));
}
```

The following function translates an 'if' node. It evaluates the conditional expression first, jumping to the 'else' clause should the result be 0. The subsequent 'then' and 'else' statements can also be if nodes, meaning that if-elseif statements can be compiled recursively.

```
fn translate_if(
  &mut self,
  cond: &Span<EXPRESSION>,
  conseq: &Span<Block>,
  altern: &Option<Box<Span<Block>>>
) {
  let l_else = self.label("else");
  let l_end = self.label("endif");

  // condition == 0 => branch .else
  self.translate_expression(cond);
  self.ir.push(BLOCK::BEQ(
    Register::from("$r0"),
    Register::from("$zero"),
    l_else.clone()
  ));

  // [...then]
  self.translate_block(conseq);

  // jmp .end
  self.ir.push(BLOCK::JMP(l_end.clone()));

  // .else
  // [...else]
  self.ir.push(BLOCK::LABEL(l_else.clone()));
  if let Some(altern) = altern {
    self.translate_block(altern);
```

```
  }

  // .end
  self.ir.push(BLOCK::LABEL(l_end));
}
```

To translate a declaration statement, I first evaluate the result of the expression which is stored in register `$r0`. I push this value onto the stack and make a note of its memory address, storing its offset from the base pointer in the function context symbol table, and decrementing the stack pointer accordingly to point to the next free slot.

```
fn translate_declaration(
  &mut self,
  binding: &Span<Binding>,
  expr: &Span<EXPRESSION>
) {
  // translate expression, store in $r0,
  // push $r0 to the stack and record offset from $bp
  self.translate_expression(expr);

  // decrement stack pointer and store variable on stack
  let sp = self.fctx().stack_pointer;
  self.ir.push(BLOCK::ADDI(
    Register::from("$sp"),
    Register::from("$sp"),
    -1
  ));

  self.ir.push(BLOCK::SW(
    Register::from("$r0"),
    Register::from("$bp"),
    sp)
  );

  self.fctx().stack_pointer -= 1;

  let scope = self.fctx().scope;

  // update variables stack slot entry in symbol table
  self.symtbl.set_slot(scope, &binding.ident, sp);
}
```

To evaluate a variable, it will either be a constant or a local variable. The symbol table stores a HashMap mapping each constant to a value, so this can simply be subtituted in.

For a variable, a record will exist in the function's context mapping that identifier to a stack slot. To resolve the variable from memory, I load the word at the memory location given by (`base pointer + stack slot`) into `$r0`.

```
// LOAD $r0, stack_slot($bp)
fn translate_variable(&mut self, ident: &Span<Identifier>) {
  let scope = self.fctx().scope;

  if let Some(stack_slot) = self.symtbl
    .resolve_variable(scope, ident)
    .expect("reference to undeclared variable")
    .stack_slot {

    // if a variable has been assigned a stack slot
    // load from memory into $r0
    self.ir.push(BLOCK::LW(
      Register::from("$r0"),
      Register::from("$bp"),
      stack_slot)
    );

  } else if let Some(SYMBOL::Const { expr , ..}) = self.symtbl.lookup_symbol(ident){
    self.translate_expression(&expr);
  }
}
```

The following function evaluates any prefix expressions, namely dereferences and a pointers. To dereference a variable, the right hand side of the expression is evlauted, and whatever value in memory exists at that address is written into the register `$r0`. A pointer however, resolves to the memory address of the variable you are referencing. The stack slot is looked up in the symbol table, and the memory address is calculated by adding that offset to the base pointer. `$r0` is set to the result of this calculation.

```
fn translate_prefix(
  &mut self,
  op: &UNOP,
  rhs: &Span<EXPRESSION>
) {
  match op {
    // evaluate the rhs (stores address of variable in $r0)
    // load word from the address in $r0
    UNOP::DEREF => {
      self.translate_expression(rhs);
```

```
      self.ir.push(BLOCK::LW(
        Register::from("$r0"),
        Register::from("$r0"),
        0
      ));
    },

    // PTR (load memory address of vairable into $r0)
    // memory address of variable is its offset from the
    // base pointer + the address of the base pointer
    UNOP::PTR => {
      if let EXPRESSION::Variable { ident } = &**rhs {
          let scope = self.fctx().scope;
          let stack_slot = self.symtbl
            .resolve_variable(scope, ident)
            .stack_slot

          // stores mem addr of variable in $r0
          self.ir.push(BLOCK::ADDI(
            Register::from("$r0"),
            Register::from("$bp"),
            stack_slot
          ));

        } else if let EXPRESSION::Literal { lit } = &**rhs{
          self.translate_literal(lit);
        }
    },

    UNOP::NEG => {
      self.translate_expression(rhs);

      // to negate a value, subtract it from 0
      self.ir.push(BLOCK::SUB(
        Register::from("$r0"),
        Register::from("$zero"),
        Register::from("$r0")
      ));
    },

    // [...]
  }
}
```

The following function translates any infix nodes. It recursively evaluates the lhs of the expression, pushing the result onto the stack. It then evaluates the rhs into `$r0` and pops the lhs back into `$r1`. The operation is then performed on those two registers.

```
fn translate_infix(
  &mut self,
  lhs: &Span<EXPRESSION>,
  op: &BINOP,
  rhs: &Span<EXPRESSION>
) {
  // evaluate lhs and store on the stack
  self.translate_expression(lhs);
  self.push(Register::from("$r0"));

  // evaluate rhs into $r0 and pop lhs back into $r1
  self.translate_expression(rhs);
  self.pop(Register::from("$r1"));

  // perform the operation on registers $r0 and $r1
  match op {
    BINOP::ADD => self.ir.push(BLOCK::ADD(
      Register::from("$r0"),
      Register::from("$r1"),
      Register::from("$r0")
    )),

    // [...]

    BINOP::GTE => self.ir.push(BLOCK::GTE(
      Register::from("$r0"),
      Register::from("$r1"),
      Register::from("$r0")
    )),
  };
}
```

The call statement is responsible for preparing the arguments required to call a function, and in turn removing them from the stack once execution has completed. It evaluates and pushes each argument onto the stack in order (thus storing the arguments in the stack slots the function has pre-assigned for parameters). It then jumps to the address of the first instruction, storing the return address in `$ra`. Finally, once the function has been executed, it decrements the stack pointer by the number of arguments, returning the stack to its initial state before the function was called.

```
fn translate_call(
```

```
  &mut self,
  func: &Span<EXPRESSION>,
  args: &Vec<Span<EXPRESSION>>
) {
  // fetch the function identifier
  let ident = EXPRESSION::Variable { ident } = &**func

  // evaluate and push each argument onto the stack
  for arg in args.into_iter().rev() {
    self.translate_expression(arg);
    self.ir.push(BLOCK::PUSH(Register::from("$r0")));
  }

  // jump to the label given by the function identifier
  // store the return address in $ra
  self.ir.push(BLOCK::JAL(
    Register::from("$ra"),
    Label(ident.ident.clone())
  ));

  // after function execution decrement $sp to pop
  // arguments off of the stack and restore state
  self.ir.push(BLOCK::ADDI(
    Register::from("$sp"),
    Register::from("$sp"),
    args.len() as i16
  ));
}
```

There are two types of assignments this language supports: values can be assigned to local variables or to memory locations directly. To assign a value o a variable, the memory address is calculated from the base pointer and stack slot and the result of the expression is stored using a `sw` instruction. When assigning directly to a memory address, the lhs of the assignment is evaluated and the result of the expression is stored directly in that memory address.

```
fn translate_assign(
  &mut self,
  lhs: &Span<EXPRESSION>,
  rhs: &Span<EXPRESSION>
) {
  // resolve address of variable:
  //   - STACK: lookup offset in the stack frame
  //   - DEREF: translate rhs, rhs is a pointer
  //            thus holds the address to write to
```

```
    self.translate_expression(rhs);

    match &**lhs {
      EXPRESSION::Variable { ident }  => {
        let scope = self.fctx().scope;

        let stack_slot = self.symtbl
          .resolve_variable(scope, ident)
          .stack_slot;

        // store $r0 in address where addr = $bp + stack slot
        self.ir.push(BLOCK::SW(
          Register::from("$r0"),
          Register::from("$bp"),
          stack_slot
        ));
      },

      // assign to dereferenced memory address
      EXPRESSION::Prefix { op: UNOP::DEREF, rhs } => {
        // push variable memory address onto the stack
        self.push(Register::from("$r0"));

        // evaluate expression and store in $r0
        self.translate_expression(rhs);

        // retrieve memory address from the stack
        self.pop(Register::from("$r1"));

        // store $r0 in the memory address in $r1
        self.ir.push(BLOCK::SW(
          Register::from("$r1"),
          Register::from("$r0"),
          0
        ));
      }
    }
}
```

## 3.5 Unit Testing

Throughout the duration of the project, I have used unit testing to ensure the system works as intended. Below, I have included a sample of the unit tests used for the assembler's lexer, and the program's output.

```
#[cfg(test)]
mod test {
  use super::*;
  #[test]
  fn test_lex_identifiers() {
    let mut lexer = Lexer::new("$a0 .label $a1\n.a$zero.1");
    let tokens = lexer.tokenize().unwrap();
    assert_eq!(
      tokens,
      vec![
        Span::new(Token::Register(1), Loc::new(0, 2)),
        Span::new(Token::Label(".label".to_string()), Loc::
            ↪ new(4, 9)),
        Span::new(Token::Register(2), Loc::new(11, 13)),
        Span::new(Token::NEWLINE, Loc::new(14, 14)),
        Span::new(Token::Label(".a".to_string()), Loc::new
            ↪ (15, 16)),
        Span::new(Token::Register(0), Loc::new(17, 21)),
        Span::new(Token::Label(".1".to_string()), Loc::new
            ↪ (22, 23)),
        Span::new(Token::EOF, Loc::new(24, 24)),
      ]
    )
  }

  #[test]
  fn test_lex_hexidecimal_numeral() {
    let mut lexer = Lexer::new("0xffff 0x00da\n0xf");
    let tokens = lexer.tokenize().unwrap();
    assert_eq!(
      tokens,
      vec![
        Span::new(Token::Number(65535), Loc::new(0, 5)),
        Span::new(Token::Number(218), Loc::new(7, 12)),
        Span::new(Token::NEWLINE, Loc::new(13, 13)),
        Span::new(Token::Number(15), Loc::new(14, 16)),
        Span::new(Token::EOF, Loc::new(17, 17)),
      ]
    )
  }

  #[test]
  fn test_error_unexpected_char() {
```

```
    let mut lexer = Lexer::new("mov %r4, $r0");
    assert_eq!(lexer.tokenize().is_err(), true);
  }
}
```

```
> assembler git:(assembler) cargo test
Compiling assembler v0.1.0 (/Users/ldesilva/Documents/School/Computer Science/NEA/NEA-Code/assembler)
  Finished test [unoptimized + debuginfo] target(s) in 1.07s
  Running unittests src/main.rs (target/debug/deps/assembler-418adfb0c6d1f1d5)

running 18 tests
test exception::tests::test_error_on_first_line ... ok
test exception::tests::test_error_on_subsequent_lines ... ok
test lexer::test::test_error_overflow_binary_numeral ... ok
test lexer::test::test_error_overflow_denary_numeral ... ok
test lexer::test::test_error_overflow_hexadecimal_numeral ... ok
test lexer::test::test_error_unexpected_char ... ok
test lexer::test::test_error_unexpected_binary_numeral ... ok
test lexer::test::test_error_unexpected_hexidecimal_numeral ... ok
test lexer::test::test_lex_decimal_numeral ... ok
test lexer::test::test_lex_comments ... ok
test lexer::test::test_lex_hexidecimal_numeral ... ok
test lexer::test::test_error_unexpected_mneumonic ... ok
test parser::test::test_parse_rir ... ok
test lexer::test::test_lex_keywords ... ok
test parser::test::test_parse_rrr ... ok
test lexer::test::test_error_unexpected_denary_numeral ... ok
test parser::test::test_parse_rri ... ok
test lexer::test::test_lex_identifiers ... ok

test result: ok. 18 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

# 4 Testing

I am going to hand compile and trace a series of programs to validate that the system is functioning correctly, as well as demonistrating various error messages the compiler can produce should it encounter syntax errors within the source code. I will also demonstrate how the graphical display and debugger for the virtual machine work - showing the robustness and performance of the system by compiling and executing a game of pong. Below is the link to the testing video:

https://www.youtube.com/watch?v=KQ6ibE77F8w

## 4.1 Testing Table

Below is a table that outlines the tests I perform in the video linked above, I discuss the parameters and expected result of each test within the video as well as a brief summary below:

| No | Test | Purpose | Timestamp |
|---|---|---|---|
| 1.1 | Assemble a program that counts to 15. | To ensure the assembler correctly assembles the program into machine code. I should expect to see the assemblers output match the hand-compiled code described in the video. | 10:53-14:30 |
| 1.2 | Assemble a program containing branch and jump instructions between labels. | To ensure jump instructions calculate absolute addresses, and branch instructions calculate relative offsets. Validates that the assembler can correctly use the position of labels within the source code to determine these values. I should expect to see the assembler correctly calculate relative and absolute addresses for any labels referenced by jump or branch instructions. | 14:30-16:26 |
| 1.3 | Assemble a program containing macro instructions. | To ensure that the assembler can correctly expand and assemble macro instructions by validating the number of instructions produced and their opcodes. In the produced machine code, I should expect to see a greater number of instructions than written in the assembly program since some instructions will be expanded before they are compiled. | 16:26-18:33 |

| | | | |
|---|---|---|---|
| 1.4 | Attempt to assemble a program containing an invalid character. | To ensure the assembler halts compilation and throws a syntax error, pointing out the position of the invalid character in source code. | 18:47-19:12 |
| 1.5 | Attempt to assemble a program containing a undefined mnuemonic. | Verifying that the assembler will not attempt to compile a program containing a mneumonic not defined within the instruction set, and instead halts compilation and throws a syntax error pointing to the location of the invalid mneumonic in source code. | 19:18-19:30 |
| 1.6 | Attempt to assemble a program containing an overflowinging integer. | To ensure the assembler throws an error specifying that the number it is attempting to assemble is too large to fit in 16 bits, instead of attempting to compile and truncating the number. | 19:30-19:40 |
| 1.7 | Attempt to assemble a program containing an unexpected register or invalid label. | To ensure the assembler throws an error specifying that the label hasn't been defined within the program, or that the register doesn't exist. | 19:40-20:15 |
| 2.1 | Trace the execution of a program to count to 15. | Verifying that the computer exhibits the correct state and control flow when interpreting a binary executable. I should expect to see the instructions within the loop execute a total of 15 times, indicating the virtual machine is correctly interpreting the loop and branch condition, and that the registers contain the expected values after execution has finished. | 21:10-22:20 |

| | | | |
|---|---|---|---|
| 2.2 | Trace the execution of a program containing jump and branch instructions. | Ensures the virtual machine correctly jumps between instructions in the program and follows the expected control flow. Using the trace function of the virtual machine, I should expect to see the PC point to the correct instruction after each jump and branch instruction. | 22:20-24:14 |
| 2.3 | Trace the execution of a program to calculate the factorial of a number. | An integration test for the assembler and virtual machine together in order to assemble and execute a more complex program involving the stack and subroutines. I should expect the final state of the computer to contain the calculated factorial of the input number, indicating the program works correctly. | 24:15-26:47 |
| 2.4 | Test the graphical display by executing programs that involve writing piexls to the screen. | Ensures the state of pixels on the screen directly correspond to the state of VRAM. Validates that writing to the screen via offsets, and within a loop work as intended. | 26:47-30:35 |
| 3.1 | Write and compile a program in the high level langauge to calculate the factorial of a number. | To demonstrate that the assembly code produced, after being copiled and executed - directly represents the high level program. I should expect to see the final state of the computer after interpreting the compiled code contain the correctly calculated factorial. | 31:50-34:17 |
| 3.2 | Attempt to compile a program containing an unexpected keyword. | Validates that the compiler halts compilation and throws a syntax error that points out the location of the error as well as suggesting alternative keywords to fix the issue. | 32:15-32:35 |

| | | | |
|---|---|---|---|
| 3.3 | Attempt to compile a program without a main subroutine | To verify that the program will not be compiled without a `main()` entry point, instead the linker will throw an error and halt compilation. | 32:30-32:38 |
| 3.4 | Attempt to compile a program attempting to assign a value of the incorrect type to a variable. | Ensuring that the type checker works as intended, and will not attempt to assign a value of an incorrect type to a variable. I should expect to see a syntax error thrown and compilation be aborted. | 32:50-32:58 |
| 3.5 | Compile and execute a program that uses VRAM and the graphical display to demonstrate passing by value and reference within my language. | Showing how dereferencing pointers to variables can be used within a subroutine to modify 'external' variables, and how to write to memory by dereferencing a memory location. Furthermore, how pointer aithmetic can be used when calculating offsets from addresses in memory. | 34:22-36:36 |
| 3.6 | Attempt to compile a program containing a reference to an undeclared subroutine. | Ensure that the compiler halts compilation and points to the location of the undeclared subroutine in the source code. | 36:36-36:47 |
| 3.7 | Compile and exeute a program that demonstrates pointer arithmetic. | Verifies that pointer arithmetic, dereferencing, and memory addresses all function correclty within the langauge. As well as testing the feature that when dereferencing a variable declared as a pointer to another variable, who's since had its contents changed - should, when dereferenced - also refer to that new value. | 36:48-38:48 |

| | | | |
|---|---|---|---|
| 3.8 | Compile and exeute pong. | An integration test that demonstrates how all three components of the system can function together to compile a complex program that involves all elements of the langauge: subroutines, stack frames, constants, conditionals, loops, pointers and references, etc. And furthermore how the CPU can use a delay timer to create a game playable in real time that interfaces with a keyboard. Ensuring the system is sufficiently optimised to execute at 60 frames per second. | 38:38-41:50 |

## 4.2 Testing Key Algorithms

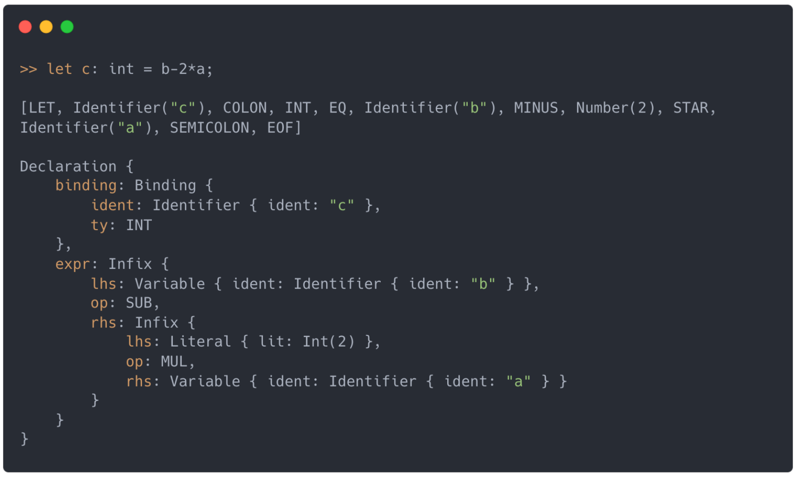
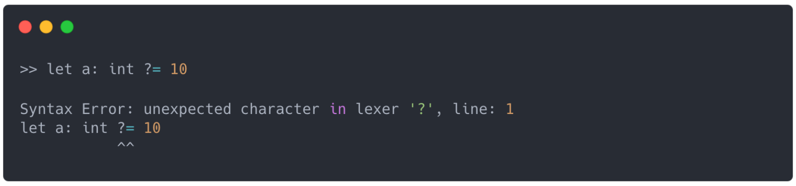
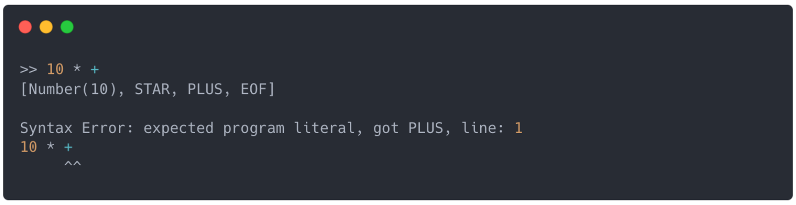
### 4.2.1 Parsing

I will run a series of tests against the key algorithms of the project in order to ensure the code works as intended. Starting with the Parser. This will ensure that each type of program statement is parsed into the correct nodes, and the system throws the expected errors when it encounters invalid syntax. Each test case includes the code being parsed, followed by the output of the lexer and parser respectively, along with any error messages.

<table>
<tr><th>Program</th><th>Result</th></tr>
<tr><td>

```
>> 1+20/2-5

[Number(1), PLUS, Number(20), SLASH, Number(2), MINUS, Number(5), EOF]

Infix {
    lhs: Infix {
        lhs: Literal { lit: Int(1) },
        op: ADD,
        rhs: Infix {
            lhs: Literal { lit: Int(20) },
            op: DIV,
            rhs: Literal { lit: Int(2) } }
        },
    op: SUB,
    rhs: Literal { lit: Int(5) }
}
```

</td><td>The expected tokens are output by the lexer, and the parser produces an AST which correctly represents the order of operations in the given expression, indicating the pratt parser is working correctly.</td></tr>
</table>

| | |
|---|---|
|   | The correct data-type has been parsed for the declaration binding, and the order of operations of the 'rhs' expression of the assignment has been parsed correctly showing that this test case has passed. |
|   | The lexer successfully threw a syntax error when it encountered an unexpected character in the source code, pointing to the specific location in the error message and passing this test. |
|   | The pratt parser determined the expression was invalid when it encountered an unexpected token (another operator). It successfully halted compilation and threw the correct error message, passing the test. |

<table>
<tr>
<td>

```
>> if (a == 10) {
    return 0;
} else if (a > 10) {
    return 1;
} else {
    return -1;
}

[IF, LPAREN, Identifier("a"), EE, Number(10), RPAREN, LBRACE, RETURN, Number(0),
SEMICOLON, RBRACE, ELSE, IF, LPAREN, Identifier("a"), GT, Number(10), RPAREN,
LBRACE, RETURN, Number(1), SEMICOLON, RBRACE, ELSE, LBRACE, RETURN, MINUS,
Number(1), SEMICOLON, RBRACE, EOF]

If {
    cond: Infix {
        lhs: Variable { ident: Identifier { ident: "a" } },
        op: EE,
        rhs: Literal { lit: Int(10) }
    },
    conseq: Block {
        stmts: [Return { expr: Literal { lit: Int(0) } }], scope: ScopeId(1)
    },
    altern: Some(Block {
        stmts: [If {
            cond: Infix {
                lhs: Variable { ident: Identifier { ident: "a" } },
                op: GT,
                rhs: Literal { lit: Int(10) }
            },
            conseq: Block {
                stmts: [Return { expr: Literal { lit: Int(1) } }], scope: ScopeId(2)
            },
            altern: Some(Block {
                stmts: [Return {
                    expr: Prefix {
                        op: NEG,
                        rhs: Literal { lit: Int(1) }
                    }
                }], scope: ScopeId(3) }) }],
        scope: ScopeId(4) })
}
```

</td>
<td>The lexer produced the expected tokens from the source code, and the parser correctly represented the if statements in the abstract syntax tree, passing the test. The 'else if' clause was interpreted as the 'altern' attribute in the top-level if statement, with the 'else' clause being the 'altern' field of the second level if satement, exhibiting the intended behaviour when parsing if-else if statements. Furthermoer each 'Block' node was assigned its own unique ScopeID which is used for referencing local variables. Finally, the last feature showing the parser passed this test was that '-1' was parsed as a Prefix node, rather than the parser attempting to interpret it as an infix expression and throwing an error.</td>
</tr>
<tr>
<td>

```
>> if (a == 10)
    return 0;

[IF, LPAREN, Identifier("a"), EE, Number(10), RPAREN, RETURN, Number(0), SEMICOLON,
EOF]

Syntax Error: expected LBRACE, got RETURN, line: 2
    return 0;
    ^^^^^^^
```

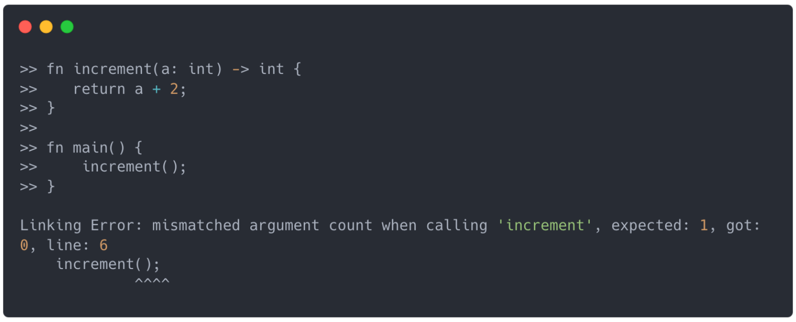
</td>
<td>When attempting to parse the if statement, the parser expects a '{' token. When, instead it encounters the 'return' keyword - it throws a SyntaxError due to the missing character. This error successfully points to the position of the unexpected token, passing the test.</td>
</tr>
</table>

| | |
|---|---|
| <pre>>> for let i: int = 0; i < 10; i += 1 {<br>>>     result += i;<br>>> }<br><br>For {<br>    init: Declaration {<br>        binding: Binding { ident: Identifier { ident: "i" }, ty: INT },<br>        expr: Literal { lit: Int(0) }<br>    },<br>    cond: Infix {<br>        lhs: Variable { ident: Identifier { ident: "i" } },<br>        op: LT,<br>        rhs: Literal { lit: Int(10) }<br>    },<br>    inc: AssignOp {<br>        lhs: Variable { ident: Identifier { ident: "i" } },<br>        op: ADD,<br>        rhs: Literal { lit: Int(1) }<br>    },<br>    body: Block { stmts: [<br>        Expression { expr:<br>            AssignOp {<br>                lhs: Variable { ident: Identifier { ident: "result" } },<br>                op: ADD,<br>                rhs: Variable { ident: Identifier { ident: "i" } }<br>            }<br>    }], scope: ScopeId(1) }<br>}</pre> | The parser assigns the correct expressions to the initialise, condition and increment fields of the for loop. Noting that whilst the initialise field is supposed to be a STATEMENT type, the other two are both EXPRESSIONs. The '+=' increment-assign operation is successfully parsed as a 'PLUS' OPERATOR and stored as an EXPRESSION in the increment field, matching the expected result for this test. |
| <pre>>> fn main() {<br>       let a: int = 10;<br>       let b: int = a + 5<br>   }<br><br><br>Syntax Error: expected SEMICOLON, got RBRACE, lines: 3-4<br>>      let b: int = a + 5</pre> | When parsing the source code, the parser notices that line 3 isn't terminated by a semicolon. It successfully throws a SyntaxError and points out the location of this error in source code, passing this test. |

### 4.2.2 Semantic Analysis

The second set of algorithms I want to test is the Sentiment Analysis phase of the compiler. Once a valid program is parsed into an abstract syntax tree, the AST is checked for further errors including references to undeclared variables or subroutines, and expressions where the data types are not compatible. Furthermore, this phase is responsible for executing the constant folding algorithm.

| Program | Result |
|---|---|

<table>
<tr>
<td>
</td>
<td>After parsing the program, the AST is fed into the type checker. The type checker notices that a multiply operation is being performed on a character and an integer, two data types which are not compatible - and throws the correponding type error. Passing this test.</td>
</tr>
<tr>
<td>
</td>
<td>The type checker is also responsible for validating the arguments passed to a function call. The testing video demonstrates when a function is called with an argument of an incorrect type, however this test demonstrates when a function is called with the incorrect number of arguments. The type checker compares the arguments passed in the function call to the parameter bindings specified in the function's definition - after noticing there is a mismatch, the type checker halts compilation and throws the correct error, pointing out the location of the invalid call in source code.</td>
</tr>
</table>

```
>> fn increment(a: int) -> int {
    if (a > 10) {
        return true;
    } else {
        return false;
    }
 }

 fn main() {
    increment(10);
 }

Linking Error: expected type 'int', got 'bool', lines: 2-7
>     if (a > 10) {
>         return true;
>     } else {
>         return false;
>     }
```

This test validates that the type checker notices when a function is attempting to return a value of an unexpected type, even when that function contains multiple return points inside a conditional expression. The type checker notices that the return values are both booleans, however the function definition expects an integer to be returned. The type checker halts compilation and throws an error, pointing out the entire conditional statement as the location of the error, passing this test.

```
>> fn main() {
      let a: int = 10;
      if (a/2) {
          a = 4;
      }
   }


Linking Error: expected type 'bool', got 'int', line: 3
    if (a/2) {
        ^^^
```

The type checker determines that the expression inside the if node's condition will evaluate to an integer and since if-nodes require their conditionals to operate on boolean values, the type checker throws an error, pointing to the location of the expression in source code. Passing this test.

```
>> fn main() {
      let a: int = 10;
      if (a == b) {
          b = 10;
      }
   }

Linking Error: use of undeclared identifier 'b', line: 3
    if (a == b) {
             ^
```

When building up the scope table during sentiment analysis, it notices that the variable ’b’ hasn’t been declared before it was used in the program. The scope table builder then successfully throws an error and halts compilation.

```
>> const resolution: int = 128 * 128;

   fn main() {
       let b: int = 10-2*3;
       let a: int = b * (18+2*5);
   }

Const {
    binding: Binding { ident: Identifier { ident: "resolution" }, ty: INT },
    expr: Literal { lit: Int(16384) }
},
Function {
    ident: Identifier { ident: "main" },
    bindings: [],
    ret_ty: VOID,
    body: Block { stmts: [
        Declaration {
            binding: Binding { ident: Identifier { ident: "b" }, ty: INT },
            expr: Literal { lit: Int(4) }
        },
        Declaration {
            binding: Binding { ident: Identifier { ident: "a" }, ty: INT },
            expr: Infix {
                lhs: Variable { ident: Identifier { ident: "b" } },
                op: MUL,
                rhs: Literal { lit: Int(28) }
            }
}], scope: ScopeId(1) } }]
```

This test case validates constant folding within the compiler. The compiler successfully noticed that the expressions '128*128', '10-2*3', and '18+2*5' could be evaluated to integers since they contain no references to variables, and replaces the entire expression with a single literal node containing its evaluated result. The expression '18+2*5' forms part of the larger expression 'b * (18+2*5)'. The algorithm is able to isolate the part of this expression that can be evaluated into an integer, and correctly replaces the expression with 'b*28'. Passing this test.

# 5 Evaluation

## 5.1 Objectives

| No | Requirement | Completed System | Timestamp |
|---|---|---|---|
| 1.1 | A RISC (Reduced Instruction Set Computer) design philosophy. | I reduced the instruction set down to 13 distinct instructions. The inclusion of the `Ri` type instructions meant that the `addi` instruction should be reused to load immediate values into registers, and move values between registers. Design decisions like this show that a RISC architecture is being used. | |
| 1.2 | A Von Neuman computer architecture where both data and instructions share the same memory. | The emulator utilises a single `memory` array to store both instructions and data, ensuring instructions are fetched sequentially from the same memory space data is stored in - consistent with a Von Neuman architecture. | |
| 1.3 | My Instruction Set should utilize 3 address operands standards. | My instruction format follows two distinct patterns: three registers, or two registers and an immediate value. Instructions can use any combination of these operands for their execution, for instance the `jr` instruction only uses a single register from the three provided by the format, whereas `bne` uses all 3. | |
| 1.4 | Branch and call instructions should calculate offsets from labels in the source code. | The assembler replaces labels in the source code with calculated offsets. Since programs are always loaded from the first memory location in the computer, the assembler can calculate absolute and relative locations of each instruction in the program for branch and jump instructions. | |
| 1.5 | Macro instructions to perform common tasks that are not otherwise specified in the ISA. | The assembler translates macro instructions like `push`, `pop`, and `xor` into their corresponding machine code sequences. This reduces repetition when programming and improves code readability. | |

| | | | |
|---|---|---|---|
| 1.6 | A set of registers broad enough to minimse memory access. | The computer includes a set of 32 registers: including temporary registers `$t0-9`, general purpose registers `$r0-12`, arguments `$a0-3`, a base pointer, stack pointer and return address. This setup reduces the need for frequent memory access, improving execution speed. | |
| 1.7 | A CPU word length of 16-bits. | All registers, ALU operations, memory locations and busses operate on 16-bit values. | |
| 2.1 | C standard syntax with semi colons and braces rather than indentation. | | |
| 2.2 | The programming language should be statically typed. | | |
| 2.3 | The language should support a procedural programming paradigm. | | |
| 2.4 | My language should support references and pointers to variables in memory. | | |
| 2.5 | The compiler should produce relevant error messages, pointing out the position in source code if relevant. | | |
| 2.6 | The language should support definite and indefinite iteration through for and while loops. | | |
| 2.7 | Data should be stored in scoped variables and global constants. | | |
| 2.8 | The compiler should support defining and calling functions. | | |
| 3.1 | The Virtual Machine should include a graphical display showing the contents of VRAM. | | |
| 3.2 | The Virtual Machine should include a togglable debugger. | | |

# References

Ball, Thorsten (2020). *Writing a Compiler in Go*. Thorsten Ball.

Engineering, DAK (2015). *Minimal Instruction Set Processor (F-4 MISC)*. URL: http://www.dakeng.com/misc.html. (accessed: 14.04.2024).

Geeks, Geeks for (2020). *Register Allocation Algorithms in Compiler Design*. URL: https://www.geeksforgeeks.org/register-allocation-algorithms-in-compiler-design/. (accessed: 21.04.2024).

Giesen, Fabian (2016). *How many x86 instructions are there?* URL: https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/. (accessed: 14.04.2024).

Joshi, Vibha (2024). *RISC and CISC in Computer Organization*. URL: https://www.geeksforgeeks.org/computer-organization-risc-and-cisc/. (accessed: 14.04.2024).

Morlan, Austin (2019a). *Building a CHIP-8 Emulator [C++]*. URL: https://austinmorlan.com/posts/chip8_emulator/. (accessed: 27.04.2024).

— (2019b). *CHIP-8 Emulator*. URL: https://code.austinmorlan.com/austin/2019-chip8-emulator. (accessed: 27.04.2024).

Muller, Laurence (2011). *How to write an emulator (CHIP-8 interpreter)*. URL: https://multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/. (accessed: 27.04.2024).

Noam Nissan, Shimon Schocken (2020). *The Elements of Modern Computing Systems: Building a Modern Computer From First Principles*. Massachusets Institute of Technology.

RetroReversing (2022). *How do Emulators work? A Deep-dive into emulator design*. URL: https://www.retroreversing.com/how-emulators-work. (accessed: 14.04.2024).

Toppr (2019). *Assembler*. URL: https://www.toppr.com/guides/computer-science/computer-fundamentals/system-software/assembler/. (accessed: 16.04.2024).

Washington, University of (2018). *A single-cycle MIPS processor*. URL: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec07.pdf. (accessed: 14.04.2024).