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. This necesitates the development of a suite of tools required 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 Subsystem 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

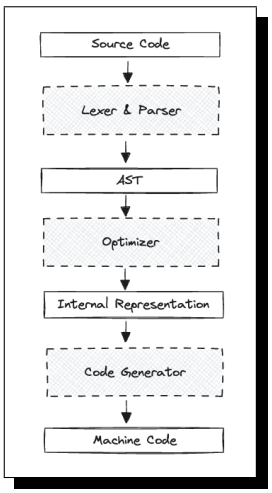
An assembler is a program that translates assembly language (a low level programming language that uses mneumonics to directly represent machine code instructions) into object code that can be executed by the processor. There are 2 types of assembler design, single-pass and multi-pass Toppr (2019). A 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.

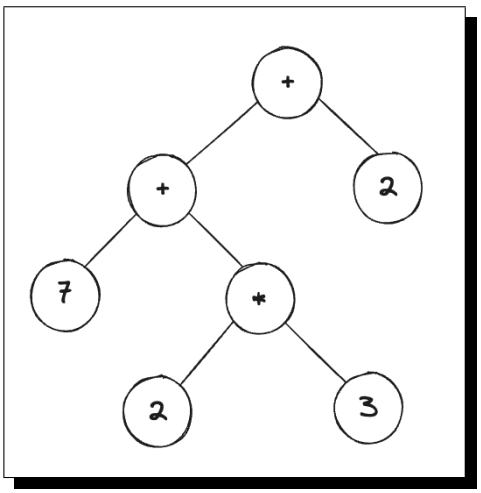
There are also certain abstractions a high-level assembler can translate such as `IF/THEN/ELSE/WHILE` statements and certain higher level data types (such as strings or arrays) – however this results in a complex assembler with lengthy compilation times - as well as a blurred line between the role of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



(a) Compiler Pipeline



(b) Abstract Syntax Tree

language before using an assembler to produce the machine code instructions, whereas others 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:

```
1 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:

```
1 let a = 9;
2 let b = 5;
3 let c = a + b;
```

```
1 ldi R0, 9
2 ldi R1, 5
3 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

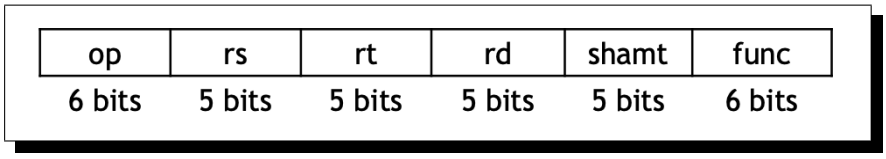
The following system is a 16 bit MISC (Minimal Instruction Set Computer) processor designed by the University of Washington for a series of lectures as part of their computer science course Washington (2018), I will discuss its ISA and machine code encoding - in order to aid my design of an appropriate 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).

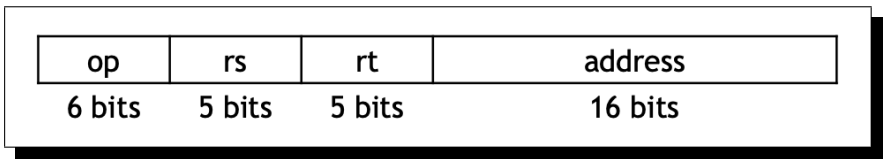
The processor supports the following instructions:

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.

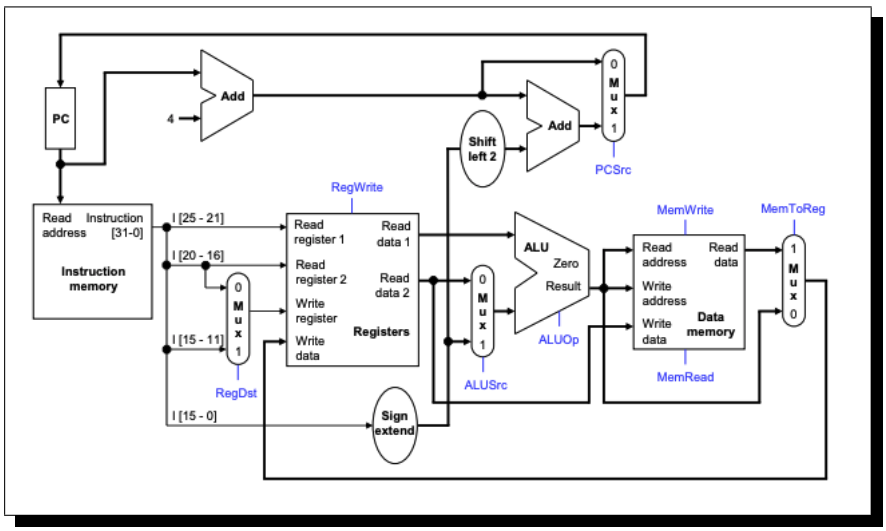


The I type encoding is the second means for which instructions can be represented, and includes the data transfer and control instructions `lw, sw`, and `beq` specified above. `address` is a signed 16 bit constant. `rt` is the destination for `lw` and source for `beq` and `sw`. `rs` is the base register 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.



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

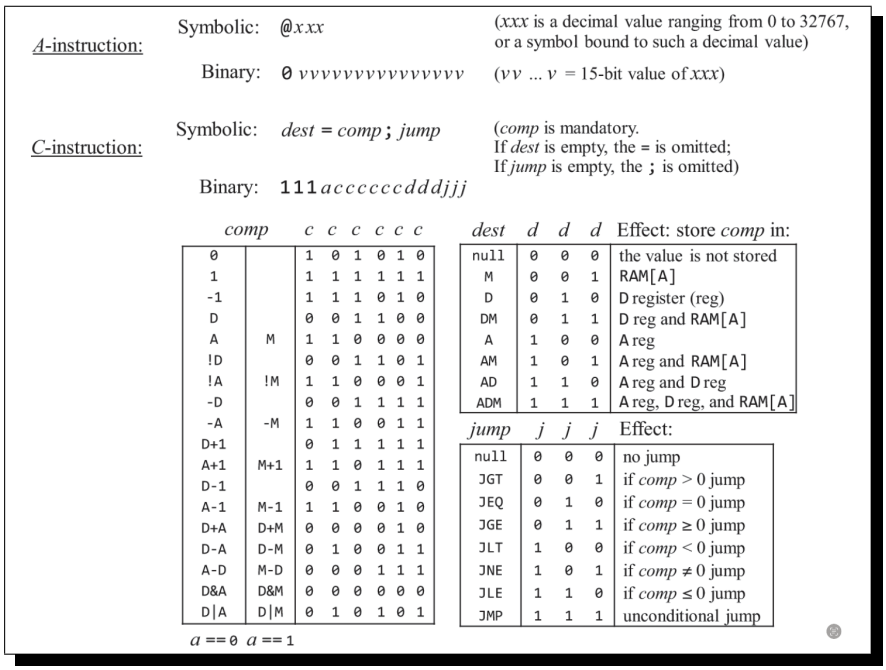
The Hack computer is a theoretical 16 bit computer designed by Noam Nisan and Shimon Schocken and described in their book The Elements of Modern Computing Systems Noam Nissan (2020), I will analyse its method of encoding assembly instructions into machine code - as well as the syntax of its assembly 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.

```
1 A type: 0aaaaaaaaaaaaaaa
2 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`*accccccdddjjj*

comp		*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

a == 0 *a* == 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:

```
1 // i = 1
2 @i
3 M = 1
4
5 (LOOP)
6     // if (i > 10) goto STOP
7     @i
8     D = M
9
```

```
10      @10
11      D = D - A
12
13      @STOP
14      D; JGT
15
16      // i += 1
17      @i
18      M = M + 1
19
20      // goto LOOP
21      @LOOP
22      0;JMP
23  (STOP)
24  @END
25  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):

```
1  class List {
2      // declare the class attributes
3      field int data;
4      field List next;
5
6      // define a constructor to initialise a List with
           ↪ attributes data and next
7      constructor List new (int dataParam, List nextParam) {
8          let data = dataParam;
9          let next = nextParam;
10         return this;
11     }
12
13     method int getData() { return data; }
14     method List getNext() { return next; }
15
```

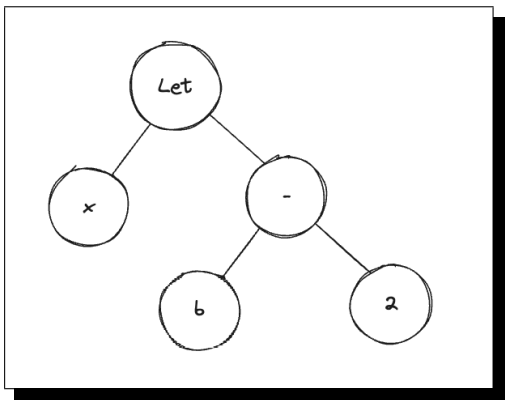
```
16      method void print() {
17          // declare a pointer to the first element of the list
18          var List current;
19          let current = this;
20
21          // iterate through all the elements in the linked
                ↪ list
22          while (~(current = null)) {
23              do Output.printInt(current.getData());
24              do Output.printChar(32) // space
25              let current = current.getNext();
26          }
27
28          return;
29      }
30  }
```

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:

```
1 // x variable is mapped to memory address $01
2 ldi a, 2
3 sub b, a
4 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:

```
1 if (b > 10) {
2     c = b;
3 } elif (b % 2 == 0) {
4     b = b + 1;
5 } else {
6     b = b - 1;
7 }
```

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.

```
1  // if b > 10 goto .then
2  ldi a, 10
3  sub a, b, a
4  bgt .then
5
6  // elif b % 2 == 0 goto .elif
7  ldi a, 2
8  mod b, 2
9  beq .elif
10
11 // b = b - 1
12 lda a, 1
13 sub b, a
14 goto end
15
16 .then
17     // c = b
18     mov c, b
19     goto end
20 .elif
21     // b = b + 1
22     ldi a, 1
23     add b, a
24 .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.

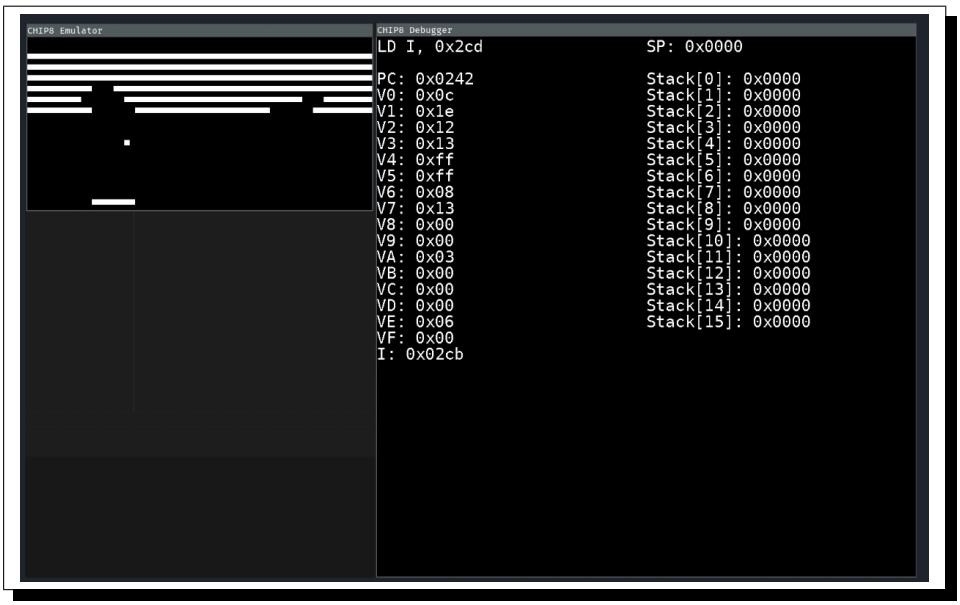
```
void chip8::emulateCycle() {
  // Fetch 16-bit instruction (4-bit opcode, 12-bit operand)
  instruction = memory[pc] << 8 | memory[pc + 1];

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

```
9         pc += 2;
10      break;
11
12      [...]
13
14      default:
15        printf ("Unknown opcode: 0x%X\n", opcode);
16    }
17
18    // 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 Harvard computer architecture where data and instructions are stored in seperate memories.	Data and instructions can be fetched during the same processor cycle, so execution is more time efficient and emulation easier (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, strings, and integers of different sizes. 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 langauge should offer structures to group related variables in memory.	Structures offer a way to transparently organise data in a structured manner in memory - without the complexity of attaching methods, constructors, public and private variables etc... Structures can be passed between functions by reference as a parameter to contain program state and produce cleaner code. (Monkey)	The `struct` keyword should be used to define a struct, followed by a list of all attributes and their data types. Structs should be instantiated using curly braces and a list of properties.
2.5	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`).
2.6	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.
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.

```
1  #include "stdio.h"
2  #include "stdint.h"
3
4  #define MEM_CAPACITY 4096
5
6  struct CHIP8 {
7      uint8_t memory[MEM_CAPACITY];
8      uint16_t pc;
9      uint8_t hlt;
10 };
11
12 void emulate_cycle(struct CHIP8 *chip8) {
13     // fetch the 2 byte instruction from memory (MSB: pc, LSB
       ↪ : pc+1) and store in a 16-bit unsigned int
14     uint16_t instruction = (chip8->memory[chip8->pc] << 8) |
       ↪ chip8->memory[chip8->pc+1];
```

```
15      chip8->pc += 2;
16
17      printf("0x%04x ", chip8->pc);
18
19      // bitmask the instruction to extract the opcode (first
            ↪ nibble)
20      switch (instruction & 0xF000) {
21          case 0x0000:
22              printf("HLT");
23              chip8->hlt = 1;
24              break;
25
26          case 0x1000:
27              printf("JMP");
28              break;
29
30          case 0x2000:
31              printf("CALL");
32              break;
33
34          case 0x3000:
35              printf("SEQ");
36              break;
37
38          case 0x4000:
39              printf("SNE");
40              break;
41
42          case 0x6000:
43              printf("SET");
44              break;
45
46          case 0x7000:
47              printf("ADD");
48              break;
49      }
50
51      printf(": 0x%04x\n", instruction);
52  }
53
54  int main(int argc, char *argv[]) {
55      // exit if user hasn't specified a ROM
56      if (argc < 2) {
57          printf("error: no input ROM\n");
58          return 1;
59      }
```

```
60
61      // initialise CHIP8 (memory and pc) values to 0
62      struct CHIP8 chip8;
63      chip8.pc = 0;
64      for (int i = 0; i < 4096; i++)
65          chip8.memory[i] = 0;
66
67      // read binary stream from ROM into chip-8 memory
68      FILE *ptr;
69      ptr = fopen(argv[1], "rb");
70
71      fread(chip8.memory, sizeof(chip8.memory), 1, ptr);
72
73      // simulate CPU cycles
74      while(chip8.hlt != 1) {
75          emulate_cycle(&chip8);
76      }
77
78      return 0;
79  }
```

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

```
1  $ gcc main.c -o main && ./main "roms/Octojam 9 Title.ch8"
2  0x0002 SET: 0x6010
3  0x0004 SET: 0x620b
4  0x000e SNE: 0x4121
5  0x0010 ADD: 0x7008
6  0x0012 SNE: 0x4121
7  0x0014 SET: 0x6100
8  0x0016 SEQ: 0x3030
9  0x0018 JMP: 0x1206
10 0x001a CALL: 0x23e6
11 [...]
12 0x01e0 SNE: 0x4d07
13 0x01e2 SET: 0x6d00
14 0x01e4 CALL: 0x23ea
15 0x01e6 JMP: 0x1264
16 0x01e8 SET: 0x6f14
17 0x01ee SEQ: 0x3f00
18 0x01f0 JMP: 0x13ea
19 0x01f2 SET: 0x6f03
20 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.

```
1  // ==== src/token.rs ====
2  #[derive(PartialEq, Debug, Clone)]
3  pub enum Token {
4      Number(u32),
5      LPAREN,
6      RPAREN,
7      ADD,
8      SUB,
9      MUL,
10     DIV,
11     EOF,
12 }
13
14 #[derive(Debug)]
15 pub struct SyntaxError {
16     pub msg: String,
17 }
18
19 impl SyntaxError {
20     fn new(msg: String) -> Self {
21         SyntaxError { msg }
22     }
23 }
```

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.

```
1  // ==== src/lexer/lexer.rs ====
2  use std::{iter::Peekable, str::Chars};
3  use super::token::{Token, SyntaxError};
4
5
6  pub struct Lexer<'a> {
7      program: Peekable<Chars<'a>>,
8  }
9
10 impl<'a> Lexer<'a> {
11     pub fn new(program: &'a str) -> Self {
12         Lexer {
13             program: program.chars().peekable(),
14         }
15     }
16
17     pub fn read_char(&mut self) -> Option<char> {
18         self.program.next()
19     }
20
21     pub fn peek_char(&mut self) -> Option<&char> {
22         self.program.peek()
23     }
24
25     pub fn tokenize(&mut self) -> Result<Vec<Token>,
       ↪ SyntaxError> {
26         let mut tokens: Vec<Token> = Vec::new();
27
```

```
28          // iterate over all characters in the source code
29          while let Some(ch) = self.read_char() {
30              match ch {
31                  ch if ch.is_whitespace() => {}
32                  '(' => tokens.push(Token::LPAREN),
33                  ')' => tokens.push(Token::RPAREN),
34
35                  '+' => tokens.push(Token::ADD),
36                  '-' => tokens.push(Token::SUB),
37                  '*' => tokens.push(Token::MUL),
38                  '/' => tokens.push(Token::DIV),
39
40                  '0'..='9' => {
41                      // parse a numebr by collecting
                            ↪ consecutive digits in the source
                            ↪ code
42                      // into the 'numeral' string
43                      let mut numeral = String::new();
44                      while let Some(ch) = self.peek_char() {
45                          if !ch.is_numeric() {
46                              break;
47                          }
48
49                          numeral.push(self.read_char().unwrap
                                ↪ ());
50                      }
51
52                      tokens.push(Token::Number(
53                          (ch.to_string() + &numeral).parse::<
                                ↪ u32>().unwrap(),
54                      ));
55                  }
56
57                  _ => {
58                      return Err(SyntaxError::new(format!(
59                          "SyntaxError: invalid character in
                                ↪ source code '{}'",
60                          ch
61                      )))
62                  }
63              }
64          }
65
66          tokens.push(Token::EOF);
67          Ok(tokens)
68      }
```

```
}
```

```
// ==== 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 {
```

```
6       pub fn get_precedence(&self) -> i32 {
7           match self {
8               Token::ADD | Token::SUB => 10,
9               Token::MUL | Token::DIV => 20,
10              _ => -1,
11          }
12      }
13  }
```

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.

```
1   // ==== src/parser/ast.rs ====
2   use crate::lexer::token::Token;
3
4   #[derive(PartialEq, Debug, Clone)]
5   pub enum Expression {
6       LiteralExpr(Literal),
7       PrefixExpr(Prefix, Box<Expression>),
8       InfixExpr(Box<Expression>, Operation, Box<Expression>),
9   }
10
11  #[derive(PartialEq, Debug, Clone)]
12  pub enum Literal {
13      Number(i32),
14  }
15
16  #[derive(PartialEq, Debug, Clone)]
17  pub enum Prefix {
18      Minus,
19  }
20
21  #[derive(PartialEq, Debug, Clone)]
22  pub enum Operation {
23      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;
```

```
23              self.tok = self.tokens[self.pos].clone();
24          }
25      }
26
27      fn retreat(&mut self) {
28          self.pos -= 1;
29          self.tok = self.tokens[self.pos].clone();
30      }
31
32      // throw a SyntaxError if the parser encounters an
            ↪ unexpected token (!= t)
33      fn assert(&self, t: Token) -> Result<(), SyntaxError> {
34          if self.tok != t {
35              return Err(SyntaxError::new(format!(
36                  "SyntaxError: expected to encounter token of
                        ↪ type '{:?}', instead encountered '{:?}'
                        ↪ ",
37                  t,
38                  self.tok,
39              )))
40          }
41          Ok(())
42      }
43
44      pub fn parse(&mut self) -> Result<Expression, SyntaxError
            ↪ > {
45          return self.parse_expression(0);
46      }
47
48      fn parse_expression(&mut self, rbp: i32) -> Result<
            ↪ Expression, SyntaxError> {
49          // parse the left hand side of an infix operation and
                ↪  determine the
50          // precedence of the next operation
51          let mut lhs = match self.parse_atom() {
52              Ok(lhs) => lhs,
53              Err(e) => {
54                  return Err(e);
55              }
56          };
57
58          self.advance();
59          let mut peek_rbp = self.tok.get_precedence();
60
61          // if an operation token is encountered, parse the
                ↪ tokens as an infix expression, and
```

```
62          // continue to parse to the lhs if operators with a
                ↪ higher precedence than rbp (e.g. *, /)
63          // are encountered - order of operations.
64          while self.pos < self.tokens.len() && peek_rbp > rbp
                ↪ {
65              lhs = match self.parse_infix(lhs, self.tok.clone
                    ↪ ()) {
66                  Ok(lhs) => lhs,
67                  Err(e) => {
68                      return Err(e);
69                  }
70              };
71
72              peek_rbp = self.tok.get_precedence();
73          }
74
75          Ok(lhs)
76      }
77
78      fn parse_infix(&mut self, lhs: Expression, op: Token) ->
            ↪ Result<Expression, SyntaxError> {
79          self.advance();
80
81          // recursively pass the rhs which iteslf can be an
                ↪ expression
82          let rhs = match self.parse_expression(op.
                ↪ get_precedence() + 1) {
83              Ok(rhs) => rhs,
84              Err(e) => return Err(e),
85          };
86
87          Ok(Expression::InfixExpr(
88              Box::new(lhs),
89              Operation::try_from(op).unwrap(),
90              Box::new(rhs),
91          ))
92      }
93
94      fn parse_atom(&mut self) -> Result<Expression,
            ↪ SyntaxError> {
95          let expr = match self.tok {
96              Token::Number(n) => Expression::LiteralExpr(
                    ↪ Literal::Number(n as i32)),
97
98              Token::SUB => {
99                  // parses the prefix operation (-): calls
```

```
                    ↪ parse_expression() with rbp 40
100                 // meaning the rhs can be an expression
                    ↪ itself however only one with precedence
                    ↪  > 40
101                 // (only parenthesised expressions)
102                 self.advance();
103                 let expr = match self.parse_expression(40) {
104                     Ok(expr) => expr,
105                     Err(e) => return Err(e),
106                 };
107
108                 self.retreat();
109                 Expression::PrefixExpr(Prefix::Minus, Box::
                    ↪ new(expr))
110             }
111
112             Token::LPAREN => {
113                 self.advance();
114                 // call parse_expression() with rbp 0 since
                    ↪ any combination of operations can be
115                 // contained within parentheses and parsed as
                    ↪  inside
116                 let expr = match self.parse_expression(0) {
117                     Ok(expr) => expr,
118                     Err(e) => return Err(e),
119                 };
120
121                 match self.assert(Token::RPAREN) {
122                     Ok(_) => {},
123                     Err(e) => return Err(e),
124                 };
125
126                 expr
127             }
128
129             _ => return Err(SyntaxError::new(
130                 format!(
131                     "SyntaxError: unexpected token
                        ↪ enocountered when parsing infix
                        ↪ expression '{:?}'",
132                     self.tok,
133                 )
134             )),
135         };
136
137         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 A Level Standard

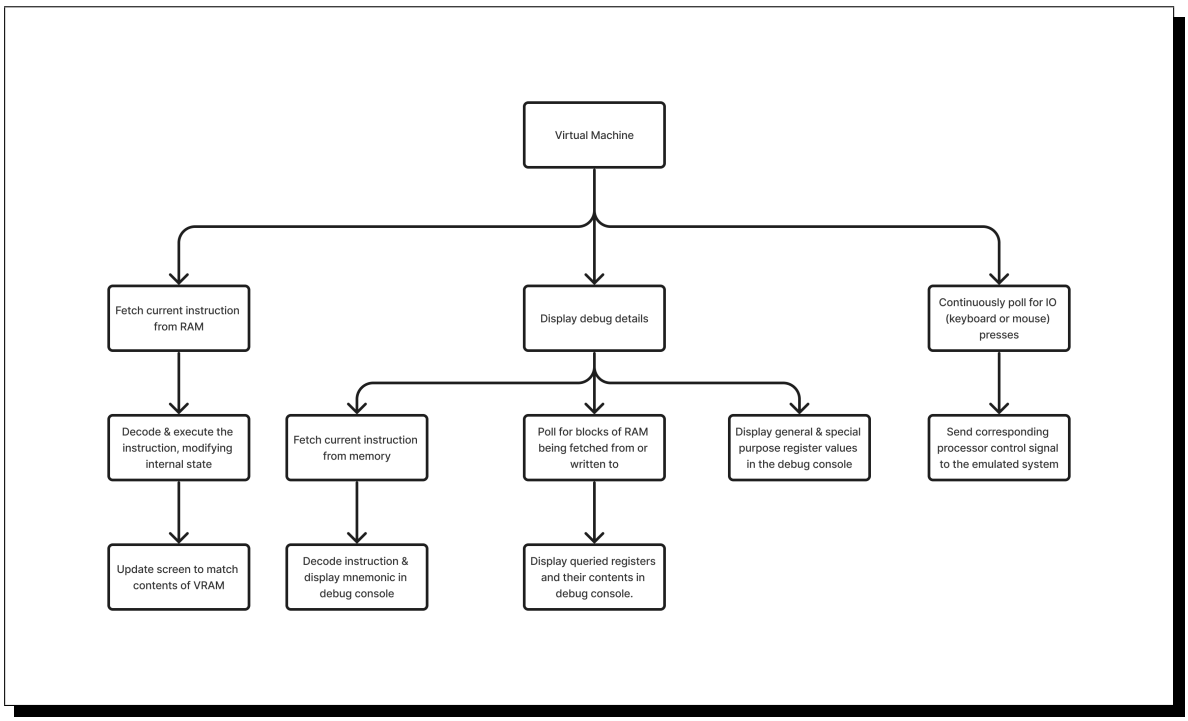
	Technical Skill	Evidence
Group A		
1.1	Complex User Defined Algorithm: a complex, heavily recursive Pratt parsing algorithm will be used to convert a linked list of Tokens into an Abstract Syntax Tree data structure representing the program and order of operations within.	
1.2	A tree data structure will be used to store parsed nodes in an Abstract Syntax Tree.	
1.3	Graph traversal algorithms will be required when optimising and compiling programs in order to locate any unconnected branches (redundant code optimisation) or to evaluate the result of an arithmetic operation at compile time, substituting it with a constant.	
1.4	A stack data structure will be used to store return addresses of the PC (program counter) from subroutines in the emulator during `call` and `ret` instructions.	
1.5	Complex user-defined use of object oriented programming using both composition, aggregation, and interfaces to relate common behaviour between distinct error structs.	
1.6	Dynamic Generation of objects will be required in both the parser and the lexer when compiling user's programs and representing them as nodes or tokens.	
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	

2 Design

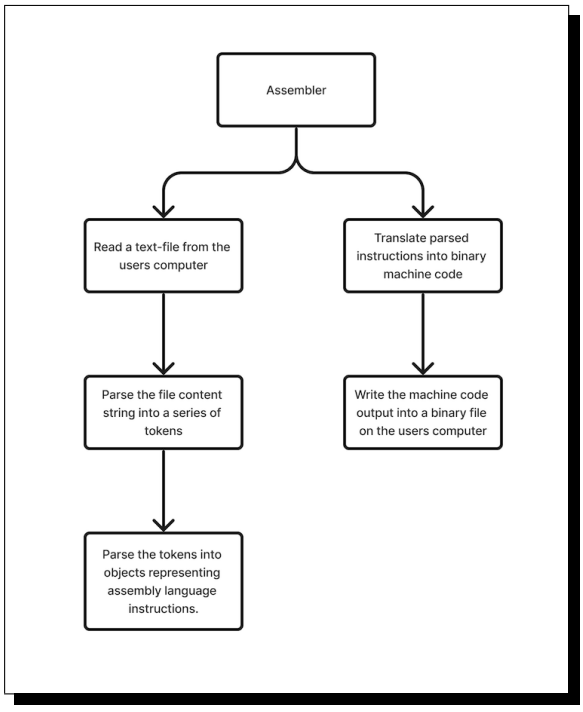
2.1 High Level Overview

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.

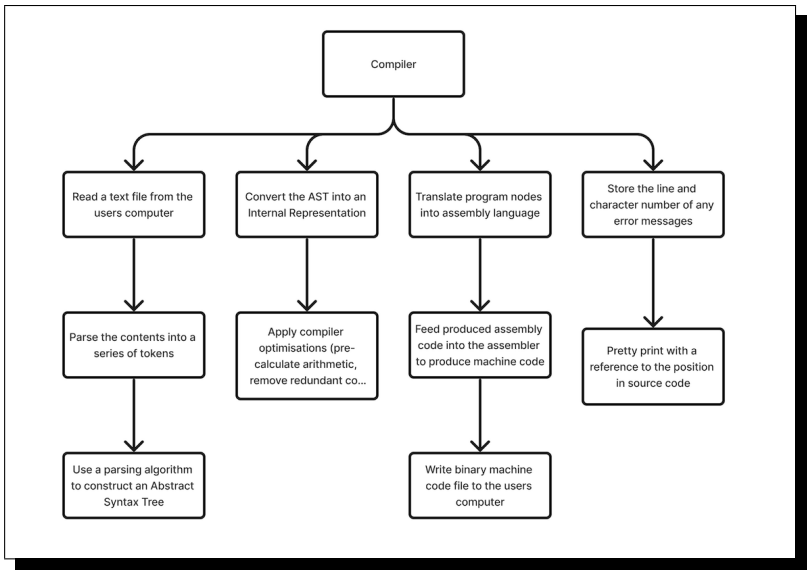


The assembler consists of a single pipeline for transforming ASCII assembly programs into binary machine code. The files are loaded into the interpreter which stores their contents in a string. The contents are tokenised and parsed into a sequence of assembly language instructions. These instructions are translated into binary machine code according to the instruction set architecture (defined in 2.2.1).



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

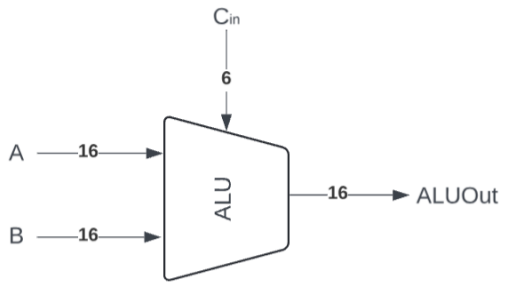
to help easily locate potential optimisations in the source code (e.g. pre-calculating arithmetic or removing redundant code), these optimisations are made and the IR is converted into an intermediate assembly language due to the presence of high level optimisations such as labels and macro-instructions. 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

The core of any instruction set is the Arithmetic Logic Unit (ALU) so I began by designing an interface for that. There are 2 16-bit inputs to the ALU, which for the purpose of my architecture will only take register values. There are 6 ALU control bits that enable further operations aside from the NOT, AND, and ADD with their own dedicated logic circuits. Different configurations of these control bits can produce various arithmetic operations as detailed in the graphic below.



zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

A=0	!A	B=0	!B	&/+	!Out
0	1	0	0	1	1

I decided to use a MIPS instruction set architecture, minimising the number of instructions supported by the processor. I settled on 3 instruction types: R-type instructions (performing ALU operations on register values), I-type instructions (load and store data between registers and memory), and J-type instructions (conditional jumps to different points in memory). From these 3 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):

```
1  R-type: add, sub, and, or, not
2  I-type: sw, lw
3  J-type: jmp, jlt, jle, jeq, jge, jgt
4
5  // multiply the numbers in memory address 0xb000 and 0xb001,
       ↪ and write the answer to 0xb002
6  lw r1, [0xb000]
7  lw r2, [0xb001]
8
9  .loop
10     // if r2 is 0, break
11     li r0, 0
12     jle r2, r0, .store
13
14     add r1, r1, r1
15
```

```
16      li r0, 1
17      sub r2, r2, r0
18      jmp .loop
19
20  .store
21      sw r1, [0xb002]
```

2.2.1.1 ISA 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 2-bit opcode which dictates the type of instruction, however only 3 instructions types are defined - leaving the fourth configuration to represent a `NOP` or `HALT` instruction. The next 6 bits of the R-type instruction are occupied by the ALU control bits which dictate the operation to be carried out. This is followed by two source registers (inputs to the ALU) and the destination register.

After the I-type opcode, an additional bit dictates whether the instruction is to be a load word (lw) or store word (sw). With a subsequent bit determining whether (for a load word instruction) the source value will be an immediate number, or the corresponding memory address (immediate or RAM[immediate]). Similarly for a store word instruction the next bit dictates whether the immediate field specifies a register holding the address in RAM, or an address by itself (RAM[register], RAM[immediate]).

Finally, for a J-type instruction 3 bits are used to specify the jump condition (¿, =, ¡) combinations of these bits can produce any condition (e.g. 101: !=, 011: ¡=). Following these there are two source registers used as inputs to the ALU for the comparison, and a 16 bit memory address to jump to should the condition be met.

Field Size	2-bits	1-bit	1-bit	1-bit	4-bits	16/4-bits
I-Format	opcode: 01	lw/sw	im/mem[?]	? = im/reg	rd	im/reg

Field Size	2-bits	6-bits	4-bits	4-bits	4-bits
R-Format	opcode: 10	c_{in}	rs	rt	rd

Field Size	2-bits	3-bits	4-bits	4-bits	16-bits
J-Format	opcode: 11	> = <	rs	rt	addr

```
1  li r5, 14
2  lw r4, [0xb000]
3  sub r6, r5, r4
4  jge r6, r4, [0x0101]
5
6  01 000 0101 00000000 00001110
7  01 010 0100 10110000 00000000
```

```
8 10 010011 0101 0100 0110
9 11 110 0110 0100 00000001 00000001
```

2.3 Virtual Machine

2.3.1 Data Structures

2.3.2 Algorithms

3 Technical Solution

4 Testing

5 Evaluation

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).