

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 | 2 |
| 1.1 | Problem Defenition | 2 |
| 1.2 | Background to the Problem Area | 2 |
| 1.2.1 | Instruction Set Architecture | 2 |
| 1.2.2 | Emulator | 3 |
| 1.2.3 | Assembler | 3 |
| 1.2.4 | Compiler | 4 |
| 1.3 | Existing Systems | 6 |
| 1.3.1 | University of Washington MIPS Computer | 6 |
| 1.3.2 | The Hack Computer | 8 |
| 1.3.3 | Monkey | 11 |
| 1.3.4 | Jack | 13 |
| 1.3.5 | Austin Morlan's CHIP-8 Emulator | 17 |
| 1.4 | Client Proposal | 20 |
| 1.4.1 | Client Interview | 21 |
| 1.5 | Objectives | 24 |
| 2 | Design | 27 |
| 3 | Technical Solution | 27 |
| 4 | Testing | 27 |
| 5 | Evaluation | 27 |

1 Analysis

1.1 Problem Defenition

The goal of this project is to design the hardware specification for a custom CPU, and develop the suite of tools required for a programmer to write programs for this processor. This will involve writing an Emulator (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) and its implementation in hardware, 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. This will require compiler optimisations in the produced object code; complex 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 capeable 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 keen 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 corresponding machine code specification with an assembler and compiler to allow a programmer to write programs to run on 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 definition 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 would allow for more complex instructions and operations to be executed in a single cycle since more bits can be processed by the CPU simultaneously, however an 8 bit system is more efficient and 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 complexity 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) Thus optimisations in performance occur on the hardware level. This is historically the most common branch of processor and often results in large instruction sets, for example the 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, as memory is required to store the additional instructions needed to perform the complex tasks not implemented in hardware.

1.2.2 Emulator

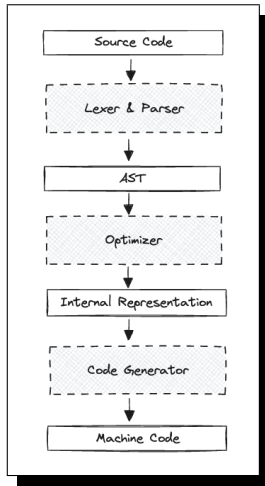
An emulator is a software program that allows the host computer to replicate the hardware of the target machine. It reads machine code instructions assembled for the target computer from memory and interprets them, mimicking the internal state of the target machine in the process, Morlan (2019a). Emulators 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 the same manner the instruction would affect the physical hardware. The Memory Subsystem is a one dimensional array of bytes that can be addressed in the same manner as RAM, regions of memory are allocated to peripherals and subsystems as with a real processor, 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

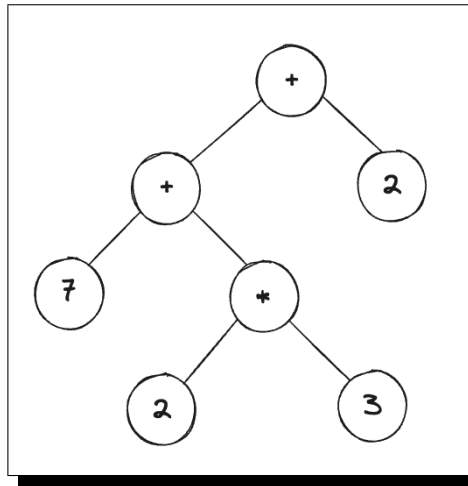
An assembler is a program that translates assembly language (a low level programming language that uses mnemonics to directly represent machine code instructions) into object code that can be executed by the processor. There are 2 types of assembler design, single-pass and multi-pass Toppr (2019). A 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 however 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 fundamental 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 higher level abstractions a high-level assembler can translate such as IF/THEN/ELSE/WHILE statements and certain higher level data types – however this results in a complex assembler and lengthy compilation times - as well as a blurred line between the role of high level and low level languages.



(a) Compiler Pipeline



(b) Abstract Syntax Tree

1.2.4 Compiler

A compiler is a program that translates high level program source code into a set of machine language instructions. Some compilers translate source code into an intermediate assembly language before using an assembler to produce the machine code instructions, whereas others 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 fundamental 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 $(\frac{7+2 \times 3}{2})$ 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 is 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, thus can 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 then 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. 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 to 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), and I will discuss its ISA and machine code encoding - to aid my design of an appropriate and efficient computer architecture. A MISC processor 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 merely 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, this is achieved using a Harvard architecture in place of Von Neuman wherein instructions are stored in a separate Read Only Memory (ROM) to data, thus both can be fetched within the same processor cycle.

The processor supports the following instructions:

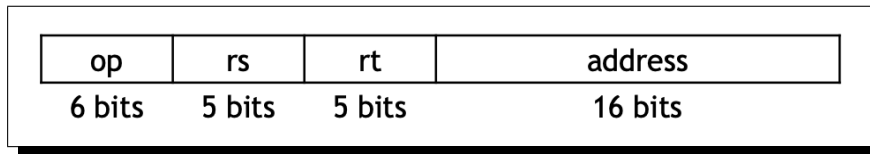
1. Arithmetic **add**, **sub**, **and**, **or**, **slt** (set if less than)
2. Data Transfer **lw** (load word), **sw** (store word)
3. Control **beq** (branch if equal to)

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 source and destination registers respectively. This computer operates on an ALU with a 3 bit control signal supporting 5 operations that directly correspond to the **func** portion of an R type instructions binary encoding.

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

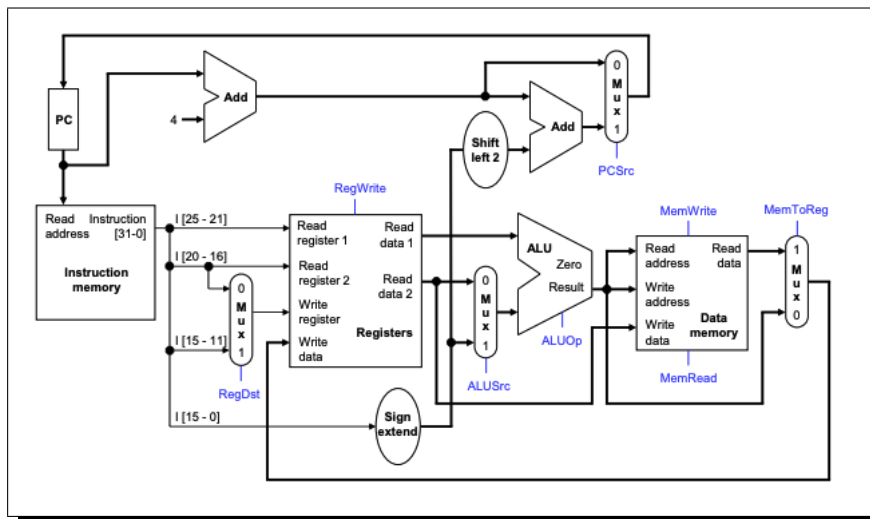
The I type encoding is the second means for which instructions can be represented, and includes the data transfer and control instructions **lw**, **sw**, and **beq** specified above. **address**

is a signed 16 bit constant. **rt** is the destination for **lw** and source for **beq** and **sw**. **rs** is the base register for the **lw** and **sw** instructions (added to the signed constant **address** to get a data memory address) Washington (2018). In this processor design, in a **beq** instruction, the **address** field specifies not a memory address, but a signed offset from which to jump from the current PC position when executing the branch instruction.



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 separate data memory.

Since instructions are stored in a separate 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 whose 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, thus 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 separation wherein operations are performed on register values, with 2 instructions `lw`, `sw` used for reading and writing to memory.
5. I will also change the branch instruction to operate on absolute addresses rather than signed offsets.

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. Whereas 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: 0aaaaaaaaaaaaa
2 C type: 111acccccddjjj
```

Hack takes a unique approach 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.

| | | |
|-----------------------|------------------------------|---|
| <u>A-instruction:</u> | 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) |
| <u>C-instruction:</u> | Symbolic: dest = comp ; jump | (comp is mandatory. If dest is empty, the = is omitted; If jump is empty, the ; is omitted) |
| | Binary: 111acccccddjjj | |
| | comp c c c c c | dest d d d Effect: store comp in: |
| 0 | 1 0 1 0 1 0 | null 0 0 0 the value is not stored |
| 1 | 1 1 1 1 1 1 | M 0 0 1 RAM[A] |
| -1 | 1 1 1 0 1 0 | D 0 1 0 D register (reg) |
| D | 0 0 1 1 0 0 | DM 0 1 1 D reg and RAM[A] |
| A | 1 1 0 0 0 0 | A 1 0 0 A reg |
| !D | 0 0 1 1 0 1 | AM 1 0 1 A reg and RAM[A] |
| !A | 1 1 0 0 0 1 | AD 1 1 0 A reg and D reg |
| -D | 0 0 1 1 1 1 | ADM 1 1 1 A reg, D reg, and RAM[A] |
| -A | 1 1 0 0 1 1 | jump j j j Effect: |
| D+1 | 0 1 1 1 1 1 | null 0 0 0 no jump |
| A+1 | 1 1 0 1 1 1 | JGT 0 0 1 if comp > 0 jump |
| D-1 | 0 0 1 1 1 0 | JEQ 0 1 0 if comp = 0 jump |
| A-1 | 1 1 0 0 1 0 | JGE 0 1 1 if comp ≥ 0 jump |
| D+A | 0 0 0 0 1 0 | JLT 1 0 0 if comp < 0 jump |
| D-A | 0 1 0 0 1 1 | JNE 1 0 1 if comp ≠ 0 jump |
| A-D | 0 0 0 1 1 1 | JLE 1 1 0 if comp ≤ 0 jump |
| D&A | 0 0 0 0 0 0 | JMP 1 1 1 unconditional jump |
| D A | 0 1 0 1 0 1 | |
| a == 0 a == 1 | | |

1.3.2.1 Advantages & Disadvantages

This approach to ISA design being so fundamentally 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, Hack assembly can be unintuitive to write and understand - especially in relation to its Harvard architecture and thus simultaneous addressing of both instructions and data from ROM or RAM respectively. An example Hack assembly program 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 mnemonic 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 instructions - reducing the number of machine code instructions required to be implemented by the virtual machine.

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 fundamental ones carrying out that defined task) instead.
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 particular instruction meaning separate 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:

```
1 let fibonacci = fn(x) {
2     if (x == 0) {
3         0;
4     } else {
5         if (x == 1) {
6             1;
7         } else {
8             fibonacci(x - 1) + fibonacci(x - 2);
9         };
10    };
11 }
12
13 let main = fn() {
14     let numbers = [1, 2, 10, 50, 9*18];
15     let index = 0;
16
17     while (index < length(numbers)) {
18         print(fibonacci(numbers[index]));
19         index = index + 1;
20     }
21 }
```

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, functionality as such 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 - however 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 register) 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 `Token` defined below.

```
1  enum TokenType {
2      LPAREN ,
3      RPAREN ,
4
5      STRING ,
6      IDENTIFIER ,
7      INTEGER ,
8      ...
9  }
10
11 type Token struct {
12     enum TokenType
13     Literal String
14 }
```

The code is scanned character by character and the fundamental 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 language 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 similar to Java that is compiled down into the Hack machine code specification. An example Jack program may look as follows 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
7     ↪ attributes data and next
8     constructor List new (int dataParam, List nextParam) {
9         let data = dataParam;
10        let next = nextParam;
11        return this;
12    }
13
14    method int getData() { return data; }
15    method List getNext() { return next; }
16
17    method void print() {
18        // 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 }

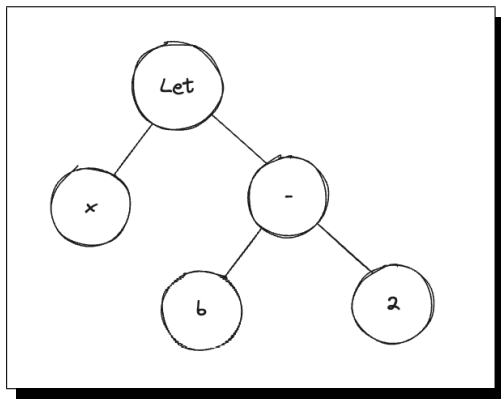
```

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` keyword. 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 unnecessary 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 - whereas 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 languages 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 unnecessary 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 separate 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 approach to program structure rather than an object oriented one.
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.
5. Compile selection and iterative statements using generated labels rather than calculated offsets.

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 `0xFFFF`) 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. Correspondingly, 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 representing 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:

```
1 #define CHARSET_ADDRESS 0x50
2 #define START_ADDRESS 0x200
3 #define MEMORY_SIZE 4096;
4 #define VIDEO_HEIGHT 32;
5 #define VIDEO_WIDTH 64;
6
7 class Chip8 {
8     public:
9         uint8_t registers[16];
10        uint8_t memory[4096];
11        uint16_t index;
12        uint16_t pc;
13        uint16_t stack[16];
14        uint8_t sp;
15        uint8_t delayTimer;
16        uint8_t soundTimer;
17        uint16_t opcode;
18 };
```

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.

```
1 void chip8::emulateCycle() {
2     // Fetch 16-bit instruction (4-bit opcode, 12-bit operand)
3     instruction = memory[pc] << 8 | memory[pc + 1];
4
5     // Decode opcode
6     switch(instruction & 0xF000) {
7         case 0xA000: // ANNN: Sets I to the address NNN
8             I = opcode & 0xFFFF;
9             pc += 2;
```

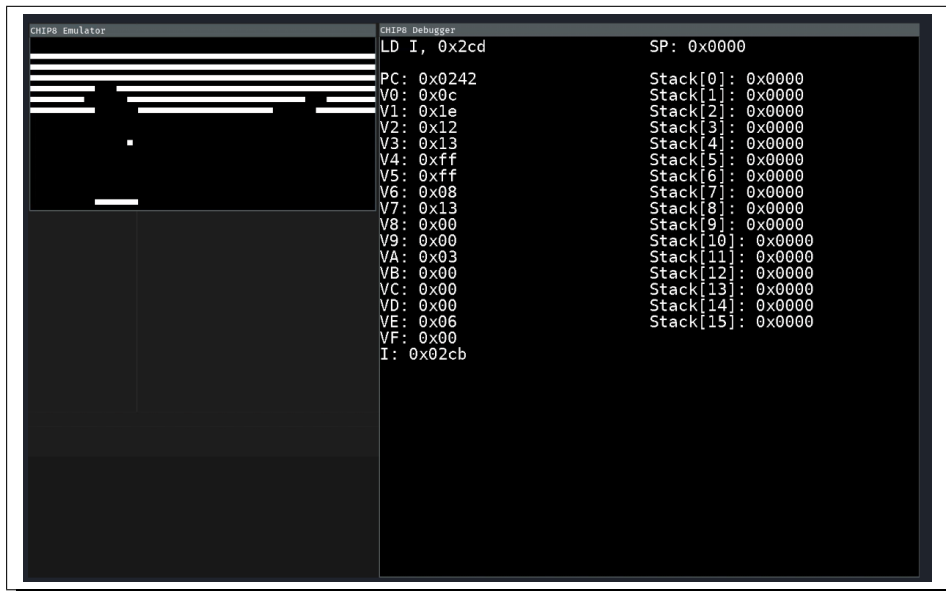
```

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 disassembled version of the currently executing instruction represented as a mnemonic, the contents of all special purpose registers (I, PC, SP), all general purpose registers (V0-VF), and the stack. This information allows a programmer to determine whether their code is running and registers being modified as expected, debugging their programs accordingly.



1.3.5.2 Advantages & Disadvantages

Morlan has decided to increment the PC value inside the switch statement separately 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 (0x000-0xFF F 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 frames 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 toggleable 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 computer, 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 a keen curiosity around lower level development. He would like to learn how the everyday languages he uses to program are implemented on the most fundamental level. I will present him with the following proposal and ascertain 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 suite 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 mnemonics can be defined - which at compile time are substituted for a list of fundamental 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

- Do you have any low level experience?
"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."
- What architecture did you use, and what were your experiences using it?
"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 has a steeper learning curve."
- Did you prefer working with a CISC (x86) or RISC (arm) Instruction Set?
"I tend to prefer RISC because the fewer instructions mean that those that are present tend to be much more carefully thought out, it's also just generally easier to program without flipping back to the documentation every 5 minutes."
- Have you ever programmed for a Harvard architecture computer?
"Most ARM processor tend to run on a Harvard architecture, although the assembly language hides most of that anyway so it's not something I really have to worry about."
- What in particular did you like about the syntax of x86 or arm assembly?
"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 mnemonics as well - although that's probably because of its larger instruction set."
- When writing assembly do you find yourself using labels or macro instructions?
"Labels are vital, I use them everywhere. 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."

- Would you prefer to write assembly for a 16-bit or an 8-bit system?
"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 separate data and instruction memories - meaning my client will have more transferable experience when programming in my language.
- 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

- When you write code, do you prefer an Object Oriented or Procedural Style?
"I like the flexibility of the procedural approach, although I tend to write cleaner code when I use OOP. I just think it's too restrictive for smaller systems."
- Do you prefer a simpler syntax like python, or something more like C?
"I've been coding in Go recently for work and I like their approach. It's got the unambiguous syntax of C with the flexibility in how you format your code, but they've also simplified the type system so you don't have to think about integer sizes or pointers in strings."
- Would you like my language to have a similar syntax to preexisting languages, or to try something new?
"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."
- If you could design your own language, what features would be most important to you?
"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 language, 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.
- 2.3 Without straying too far from the norms, I should consider alternative approaches to syntax and language 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

- Have you ever used a Virtual Machine before?
"I've used a Gameboy emulator before, but never anything for coding. I just haven't had the need."
- What features would you expect if you were using developing using a virtual machine to test your code
"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 | The processor should implement a RISC (Reduced Instruction Set Computer) design philosophy. | A RISC instruction set allows more focus to be placed on the software side (the domain of this NEA) and results in faster and more efficient execution of it's instructions resulting in strong performance benefits, especially when a user isn't as familiar with the instruction set of the system. (Client Interview 1.1) | |
| 1.2 | The processor should use 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 assembly language should support 3 address operands and hide the need to address data and instructions from different memories. | 3 address operands is the programming standard for both arm and x86, pre-established instruction sets programemrs are already familiar with. (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 | My assembler should be able to calculate offsets of branch instructions from the position of labels within the source code. | Labels provide a necessary convenience for programmers implement branching, selective statements, or functions without having to manually calculate memory offsets. (The Hack Computer, The University of Washington MIPS Computer, Client Interview) | My assembler should be able to replace all occurences of an ASCII label in the assembly source code with calculated offsets from the current instruction to that label. |

| | | | |
|--------------|--|--|--|
| 1.5 | My assembler should provide a number of macro instructions that perform common tasks (e.g. pushing an address to the stack) 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 <code>call</code> and <code>ret</code> for a list of machine code instructions that perform the same task. |
| 1.6 | My pocessor should have the following internal registers: 16 general purpose registers, a Delay timer, Program Counter, and Stack Pointer. | 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 | My CPU should operate with a word length of 16-bits. | A 16-bit word length allow more bits to be processed in a single cycle than an 8-bit system, and allows 16-bit instructions to be fetched within a single cycle improving performance. 16-bits can also represent numbers of a greater magnitude or precision using floating point representation than 8-bits mitigating the prevalence of 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 | The syntax of my language should follow C-like standards with semi colons and braces rather than relying on indentation. | This permits 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 | <p>The programming language should be statically typed.</p> | <p>A type system results in robust programs with type-errors thrown at compile time rather than during execution, meaning errors like accidentally reassigning the type of a variable will be detected by the compiler and programs 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)</p> | <p>My syntax should have support for signed and unsigned integers, strings, and integers of different sizes. The <code>let</code> keyword should be used when declaring a variable and require the type to be specified alongside its identifier.</p> |
|-----|---|--|---|

- 2 Design**
- 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*. Massachusetts 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).