Computer Science Non-Examined Assessment

Max Black

Candidate Number: ****
Centre Number: *****

Contents

1 Introduction **1**
1.1 The Intel 8086 . . . 1
1.2 Why build an emulator? . . . 1
1.3 Why emulate the Intel 8086? . . . 2

2 Analysis **3**
2.1 Stakeholders . . . 3
2.1.1 Interview . . . 3
2.1.2 Stakeholder Requirements . . . 3
2.2 Computational Methods . . . 4
2.2.1 Abstraction . . . 4
2.2.2 Decomposition . . . 4
2.2.3 Object-Orientation . . . 4
2.3 Features & Limitations . . . 5
2.4 Software & Hardware Requirements . . . 6
2.4.1 Hardware . . . 6
2.4.2 Software . . . 6
2.5 Existing Projects . . . 6
2.5.1 i8086emu . . . 6
2.6 8086tiny . . . 8
2.7 Success Criteria . . . 8

3 Intel 8086 Microprocessor **10**
3.1 General-Purpose and Index Registers . . . 10
3.2 Status Register/Flags . . . 10
3.3 Memory Segmentation and Segment Registers . . . 11
3.4 Instruction Encoding . . . 11
3.4.1 Opcode . . . 11
3.4.2 MOD-REG-R/M . . . 12
3.4.3 Displacement . . . 12
3.4.4 Immediate . . . 13

4 Intel 8259 Programmable Interrupt Controller (PIC) **13**
4.1 Registers . . . 13
4.1.1 Interrupt Mask Register (IMR) . . . 13
4.1.2 Interrupt Request Register (IRR) . . . 14
4.1.3 In-Service Register (ISR) . . . 14
4.2 Interrupt Vector Table . . . 14
4.3 Interrupt Firing/Handling . . . 14

5 Intel 8253 Programmable Interval Timer (PIV) **14**
5.1 Control Word . . . 15

6 Design **16**
6.1 Coding Style . . . 16
6.1.1 Use of Case . . . 16
6.1.2 Whitespace . . . 17
6.1.3 Naming Conventions . . . 17
6.1.4 Examples . . . 17
6.2 Project Structure . . . 17
6.3 Namespaces . . . 18
6.4 Build System . . . 19

6.5 GUI Design . . . 19
6.6 Instruction Representation . . . 21
6.7 Usability Features . . . 22
6.8 Testing . . . 22
6.8.1 Unit Testing . . . 22
6.8.2 Assembly Generation . . . 23
6.8.3 Instruction Execution . . . 23
6.8.4 Emulator Memory . . . 24
6.8.5 Memory Segmentation . . . 24
6.8.6 Post-Development Testing . . . 25

7 Implementation: First Stage **26**
7.1 Development . . . 26
7.1.1 Setting Up Git . . . 26
7.1.2 Primitives . . . 26
7.1.3 Memory . . . 27
7.1.4 CPU . . . 28
7.1.5 Register Indexes . . . 30
7.1.6 Registers . . . 31
7.1.7 Helper Functions . . . 32
7.2 Testing . . . 32
7.2.1 Testing of Memory . . . 32
7.2.2 Testing of Helper Functions . . . 33
7.2.3 Testing of CPU Registers . . . 34
7.3 Review . . . 34
7.3.1 Overview of Progress . . . 34
7.3.2 Success Criteria . . . 34
7.3.3 Plans . . . 34

8 Implementation: Second Stage **36**
8.1 Development . . . 36
8.1.1 Logging . . . 36
8.1.2 Instruction Opcodes . . . 38
8.1.3 Changes to CPU . . . 38
8.1.4 Conversion/Helper Functions . . . 39
8.2 Testing . . . 40
8.2.1 Testing of New Helper/Conversion Code . . . 40
8.2.2 Testing of Basic Instruction Representation . . . 41
8.3 Review . . . 41
8.3.1 Overview of Progress . . . 41
8.3.2 Success Criteria . . . 41
8.3.3 Plan . . . 41

9 Implementation: Third Stage **42**
9.1 Development . . . 42
9.1.1 Type Aliases . . . 42
9.1.2 System Memory to/from Files . . . 42
9.1.3 CPU Register System . . . 43
9.1.4 CPU Header Expanded . . . 46
9.1.5 Assembly Style . . . 47
9.1.6 Instruction Opcode Representation . . . 47
9.1.7 MOD-REG-R/M . . . 49
9.1.8 Instruction Arguments (Displacement and Immediate Values) . . . 52
9.1.9 Instruction Classes . . . 54
9.1.10 Yet More Conversion/Helper Functions . . . 58

9.1.11 CPU Implementation 60
9.2 Testing 62
9.2.1 Conversions 62
9.2.2 Instruction Representation 62
9.2.3 Stack 63
9.2.4 Unknown Instruction Handling 64
9.3 Review 64
9.3.1 Overview of Progress 64
9.3.2 Success Criteria 64

10 Example Instruction Decodings **66**

11 Full Interview **67**
11.1 Are you satisfied with the resources you currently have available for teaching about the low-level workings of computing systems? 67
11.2 Have you considered implementing practical demonstrations into such lessons? 67
11.3 If so, did you find that it enhanced the learning experience of your students? 67
11.4 Have you before considered performing demonstrations using more simplistic, early computer systems (whether physical or emulated) to help illustrating your teaching points? . 67
11.5 What features would you look for in an emulator to make it most applicable for usage in a teaching environment? 67
11.6 What would, in your opinion, be the ideal interface for such a piece of software? 67

12 Sources **68**

1 Introduction

1.1 The Intel 8086

The birth of x86 architecture - the very same architecture that the vast majority of desktop computers still use today (or at least some 64-bit variant thereof) - took place just over forty years ago with the release of Intel's revolutionary 8086 16-bit microprocessor.

While obviously the 8086 pales in comparison to modern processors with a clock-speed of 5 to 10 MHz and the ability to address a megabyte of segmented memory the 8086 was, for its time, more than capable in regards to raw processing power. This power however is not the source of the 8086's acclaim. Rather, the reason the Intel 8086 remains relevant today is because of its unique instruction set and the legacy it created as a result.

Using real addressing mode, any code written for the original 8086 should (at least theoretically) be able to run on any modern x86 processor (assuming all the required interrupts and hardware are available). Importantly, all innovations in processing technology today rely on the 8086 as a backbone.

1.2 Why build an emulator?

For a long while I have found the prospect of implementing my own emulator intriguing. Even before I began programming, I frequently experimented with running virtual machines and emulators for all manner of different hardware. I particularly enjoyed modifying values within memory of a Nintendo Entertainment System (NES) emulator to observe what effects it would have on the running game.

While the NES itself is a something of a black box, with an emulator it is possible to see the precise state of the Central Processing Unit (CPU) as it runs, change exactly how it runs and even see the disassembled code running on it in real time. In Figure 1 for example, I was able to change the colour scheme in the Legend of Zelda as well as give Link infinite health and access to the best weapons in the game simply by changing a few values in memory.

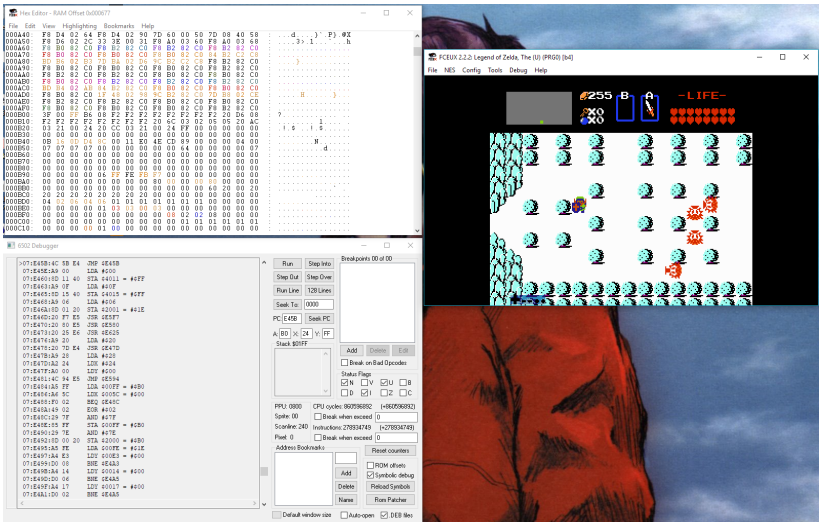

Figure 1: *The Legend of Zelda running in FCEUX with memory modifications.*

In view of this, the key elements for my own emulator are:

1. the ability for the user to view exactly what the CPU is doing at any one time, and

2. modify its behaviour whilst it is operating.

It is not the intention of this project to create a particually fast or complex implementation of the Intel 8086. It is to allow the user to understand what is happening 'under the hood'.

1.3 Why emulate the Intel 8086?

There are a number of reasons as to why I chose to replicate the 8086 specifically. These are summarised as follows:

1. The 8086 had a complexity level high enough to be challenging while not being entirely unapproachable.

2. Being so well known and having existed for quite some time, there should be a wealth of documentation available for the 8086 online and in books.

3. Even if the required documentation cannot be found, using disassemblers, hex editing tools and existing emulators, it should be feasible to reverse engineer certain components regardless should there be a need to do so.

As mentioned previously, all current x86 processors maintain backwards compatibility with the Intel 8086. This means that, should I wish to implement a more advanced x86 processor in the future, it should be possible to reuse some of the code from this project.

This was another key point that drew me to the 8086 as, assuming this project is successful, I plan to implement the 32-bit Intel 80386 sometime in the future.

2 Analysis

2.1 Stakeholders

This project is designed with computer science education in mind. As such potential stake holders could include various educational institutions as well as individual teachers and tutors. For example, the project could be used in a sixth-form or university classroom to demonstrate the workings of the x86 architecture or the more general workings of a typical Von Neumann architecture CPU.

In this particular instance, my stakeholder shall be my own A-Level computer science teacher, Mr. Sisley.

2.1.1 Interview

My interview questions for Mr Sisley (stakeholder in education) are as follows:

1. Are you satisfied with the resources you currently have available for teaching about the low-level workings of computing systems?

2. Have you considered implementing practical demonstrations into such lessons?

3. If so, did you find that it enhanced the learning experience of your students?

4. Have you before considered performing demonstrations using more simplistic, early computer systems (whether physical or emulated) to help illustrating your teaching points?

5. What features would you look for in an emulator to make it most applicable for usage in a teaching environment?

6. What would, in your opinion, be the ideal interface for such a piece of software?

My stakeholder expressed his current dissatisfaction with the teaching resources available for demonstrating the workings of low-level systems to students. He states that he is "only really aware of the little man computer simulation" which is "quite good for GCSE-level students". However, he does believe that is not really "suitable for A Level" and such feels there is a "gap in the market" for such a resource or piece of software. Indeed, this is something I hope to address with my project.

Mr Sisley also stated that practical learning demonstrations are a "real benefit when trying to get an idea across to a class" due to low-level computing being a "very dry, abstract topic". Again he mentions the problems had with the little man computer simulation, stating that it "can be rather difficult for students to follow and understand". He later elaborates on ways in which my own solution can address these issues.

2.1.2 Stakeholder Requirements

Mr Sisley's requirements were assess by means of an interview (see section 2.1.1 and appendix 11).

2.2 Computational Methods

A computational solution is appropriate here for a number of reasons. One such reason is that one of the best ways for potential stake holders (i.e. teachers) to teach about the low-level workings of an x86 system is show some such system in action. While theoretically they could indeed source an old IBM PC or similar for this purpose, doing so nowadays is rather difficult and expensive due to the rarity of such old systems. Instead, a simpler alternative is to just run emulators of these old systems using their existing modern hardware. Another bonus of doing this as an alternative is that an emulator allows for far more precise information about and control of the running of the system. Indeed, this is a key aspect that I think will allow my project specifically to be useful in the domain of education as a key focus of its design is allowing for the exposure of the inner workings of the emulated system.

2.2.1 Abstraction

Abstraction is used in the project in the sense that not all of the many features of the Intel 8086 are actual implemented as they are superfluous for the purpose of this project. In particular, the entire concept of concept of the 8086 being a physical chip with specific pins is entirely disregarded as implementing such low-level emulation would be effectively pointless. The idea of transmitting data to memory via an address bus and a data bus is also abstracted away, with the emulator instead simply having a reference to an object representing main memory which is used for interaction with emulator memory. Hardware modes (*min*, *max* mode) are also unnecessary due to the aforementioned.

2.2.2 Decomposition

One conclusion I reached straight away was that the GUI and the emulator itself should be entirely decoupled and separated as much as possible. Decomposing the entire project into these two primary components was advantageous for a couple of reasons. Firstly, it allowed my to initially focus exclusively on the implementation of the emulator without getting bogged down with to implementing GUI simultaneously. In addition, it also allowed the emulator to run without a GUI at all which would be convenient for testing via automatic unit testing. From a future-proofing perspective, it could also allow for implementation of a CLI or additional GUIs using preferred GUI libraries for certain platforms.

See figure 2 below for a detailed breakdown of the components this project is comprised of. This diagram was created using the divide & conquer method of decomposition.

2.2.3 Object-Orientation

I feel as though an emulator is particularly appropriate to an object-orientated design for a number of reasons. A computer system tends to have a set collection of components with each having a distinct function. The functioning of the system as a whole is then but a product of the interactions between these separate components. This lends itself well to an object-orientated design as each component of the emulated system (the CPU, memory, registers, etc.) can be implemented as a class with private internals and then a public interface to facilitate the interaction with other components.

The ability to set private, protected and public properties of a class also encourages better design by limiting which pieces of code may modify certain variables or access certain methods. This is applicable to this particular project as being able to identify where program state is changed is integral to maintaining the stability of the program due to the fact that an emulator has many keepers of state that when modified dramatically influence its running (segment registers, CPU flags, among others).

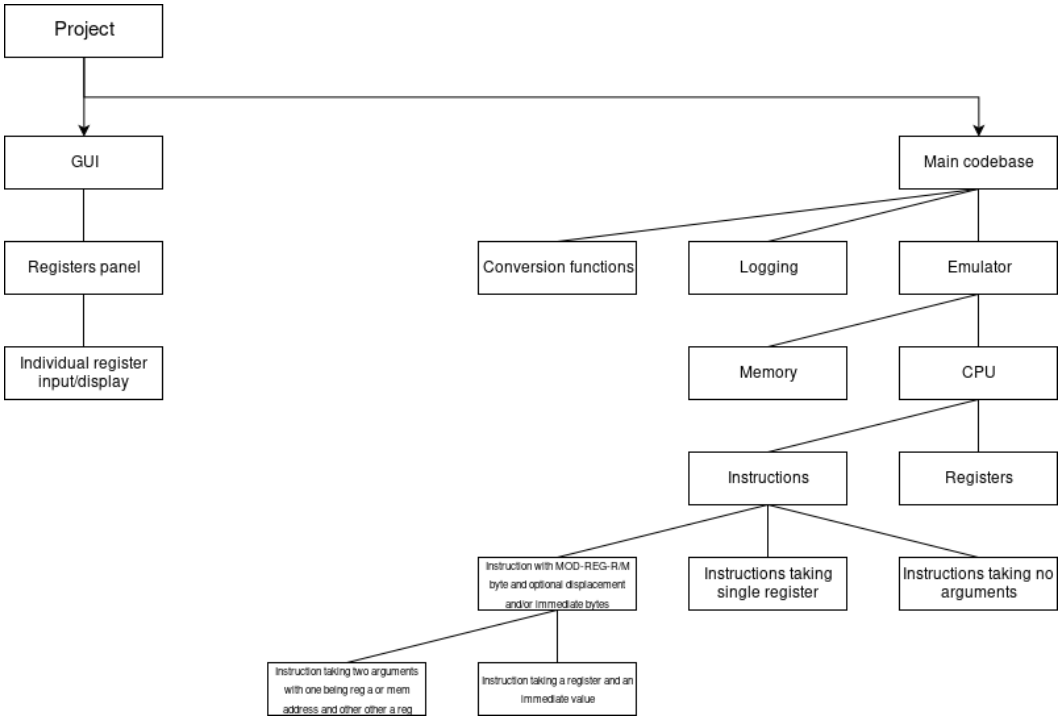

Figure 2: *Decomposition of the project.*

Object-orientated inheritance will also prove vital to allowing code reuse. In particular, classes to represent CPU instructions will benefit from inheritance especially due to the many subtypes of instruction having similarities to and sharing components of more general instruction types (see figures 3 and 2).

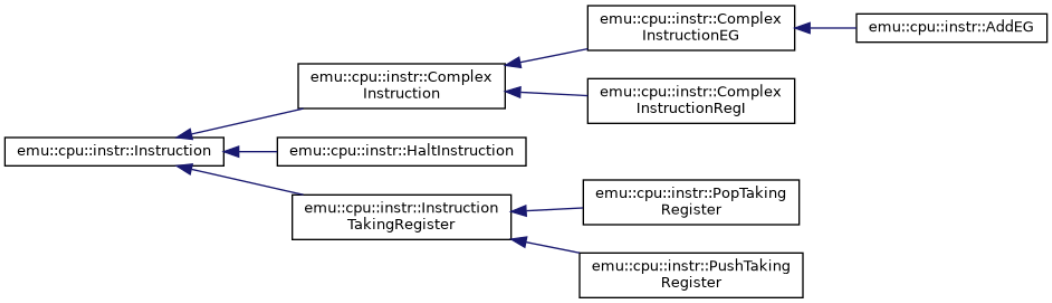

Figure 3: *Hierarchy of instruction classes.*

2.3 Features & Limitations

One feature that was required as part of the coursework specification was a Graphical User Interface (GUI). The choice of GUI library to use proved to be more challenging than anticipated - see section 6.5 of the Design portion of this document for more information.

As stated above, one key feature of the emulator should be its applicability to a teaching environment. As such, detailed information regarding the running of the emulator (state of registers, assembly expression of the current instruction, state of main memory, etc.) should be readily available. In addition, ability to modify the state of emulator should be possible to allow students to learn through the observing the consequences of any modifications they make. On the topic of modification, the source code of the project must be well organised, cleanly written, and fully documented so that students may learn through reading its implementation and by potentially making modifications to the software itself.

Speed of execution is most certainly not to be prioritised during development meaning any optimisations

that could be made to the code at the expense of readability or stability are not to be made. This lack of optimisation could potentially be presented as a limitation of the software in certain situations. However, due to emulation of such old hardware not being particularly resource-intensive as well as the project being implemented in the fast, compiled C++ language, this should not present any significant issues.

Another limitation of the solution is that fact that not all of the instructions of the Intel 8086 can be run on it. This means that complex programs that utilise more obscure instructions will be not be able to run on the emulator system. Fortunately, this should not cause significant issues due to the fact that software aimed primarily at teaching does not require advanced/obscure features.

2.4 Software & Hardware Requirements

2.4.1 Hardware

The software will have only one real hardware requirement: a computer. This computer should have a architecture which can be targeted by the chosen compiler (see section 2.4.2). At least 2 GB of DDR3 RAM is recommended (depending on the OS used).

2.4.2 Software

In terms of running the software, the only requirement is an operating system with a graphical desktop environment/window manager that supports the creation of and interaction with an SFML window.

As for compilation, there are several software requirements. The first of such requirements being CMake version 3.12 or higher. The purpose of this software is to generate an appropriate build system using whatever is available on the system. As such, appropriate software that CMake can use to handle the actual compilation process is also required. On my Linux system, CMake uses the GCC, GNU Linker, and Make programs which I also have installed. On Windows, it uses Microsoft Visual C++ for compilation. CMake will produce an error message if no appropriate compiler/build system is installed. The project's `CMakeLists.txt` file specifies that C++ Standard 17 is required and so will also produce an error message should the available compiler not support that version of C++ (for example, all features of C++ 17 are implemented in GCC version 8 and above).

Compiling the software also requires that a few libraries are installed for the purpose of creating a GUI. These include: SFML, ImGUI, and the SFML binding for ImGUI. The project also uses the Catch2 library for unit testing, however this only requires a single header file which is to be distributed with the project's source code and as such requires no additional installation steps.

2.5 Existing Projects

2.5.1 i8086emu

There are a few different existing emulators of the Intel 8086 microprocessor and associated hardware. One such existing project goes by the name of *i8086emu*. As seen in figure 4, this emulator has some some interesting features that I ultimately decided to implement into my own emulator. For example, one feature of i8086emu is its disassembler. This displays the instructions that the CPU has and will execute in a human-readable assembly format. I felt such a feature would be important to add to

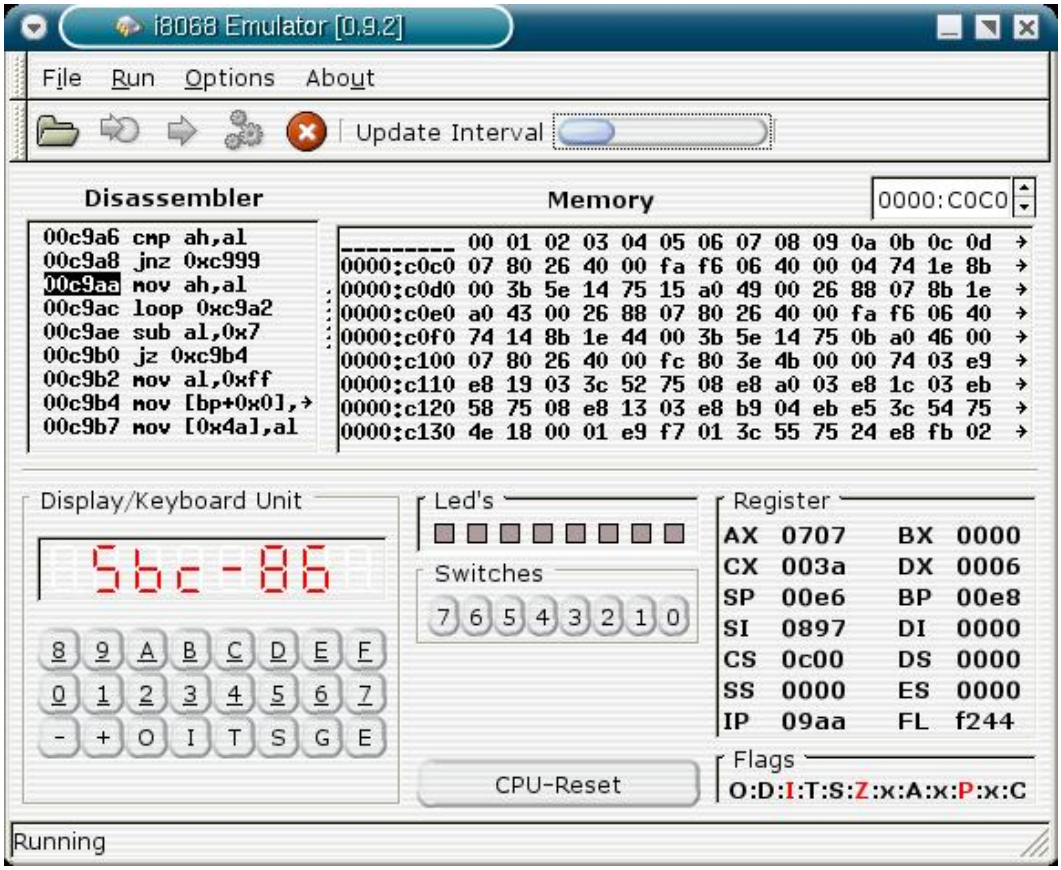

Figure 4: *Screenshot of the i8086emu emulator's GUI.*

my own emulator as it gives the user convenient insight into what each instruction in memory will do without having to manually perform opcode and MOD-REG-R/M lookups. This makes the software especially applicable to a teaching environment as it provides a basic demonstration of CPU instruction decoding.

The i8086emu emulator also shows the value of CPU registers and flags as part of its user interface. Again, such a feature would be vital to include in my emulator as it provides vital insight into the running of the emulation. However, I also decided to go a step further and allow the user to modify the values of registers via the interface during execution. This facilitates students using the emulator to experiment with and hopefully learn through observing results after changes are made to register values.

While the code of the i8086emu is fairly readable and commented consistently, all of those comments are written in German. As such, learning through the reading and modification of the source code may prove challenging for students unable to read German. Naturally, I hope to improve on this by providing clear comments in English in my own emulator.

The i8086emu project source code is listed under the GNU General Public License (GPL) which means that its code is free to be modified and redistributed while requiring any modifications are also licensed under the GPL. As someone who appreciates open-source software and have found it invaluable over the years, I feel a desire to also use a similar such license for my own project.

2.6 8086tiny

Other emulators of the Intel 8086 CPU also exist, such as *8086tiny* by GitHub user *adriancable*. This particular piece of software is claimed to be "the smallest of its kind" with the "fully-commented source"

user 25 kilobytes in size.

I have identified a fair few (of what I perceive to be) issues with this emulator that I hope to avoid while design and developing my own. One such issue is the heavy use of pre-processor macros. The 8086tiny emulator is written in the C programming language which supports a pre-processor system wherein the programmer can define constants and functions which are then partially evaluated at compile-time. The C++ programming language (which I am using for my emulator) inherits its pre-processor from C and as such allows for the definition of similar such macros. Macros do however present many issues and many of their benefits can now instead be achieved using the features available in modern C++ standards. As an example, macros essentially disregard the static typing system provided in the C and C++ languages and as such can results in thoroughly confusing error messages if values of an inappropriate type are used with them. Macros also cannot be enclosed in namespaces - they are always automatically a part of the global namespace. The main supposed benefit of macros is their speed due to their partial compile-time evaluation, however the C++ keywords `inline` and `constexpr` allow for similar compile-time expansion to be achieved. Regardless, speed optimisations are not a priority for my own emulator.

Another issue I have with 8086tiny is the obtusely-named global variables. Firstly, I plan to avoid the use of global variables as much as possible (or even entirely) by encapsulating values inside namespaces and classes. In addition, I will refrain from using variables names such as `i_rm` and `spkr_en` (as seen in 8086tiny) and instead endeavour to use more descriptive names. The 8086tiny emulator is also defined in a single source file `8086tiny.c` which is again something I will avoid as making use of multiple files/directories helps the navigation and understanding of a large code base.

A feature of 8086tiny that I however do appreciate is its fairly extensive commenting, giving explanations of the purpose of each major piece of code. This extensive commenting is certainly something I plan to incorporate into my own software.

2.7 Success Criteria

The original success criteria of this project was far too ambitious given the limited time available to be dedicated to this project. Originally, it was planned that the emulator would be considered complete only once it was capable of running a Basic Input Output System (BIOS) and then successfully booting an early version of Microsoft Disk Operating System (MS-DOS). While this of course would have been rather impressive, it would have required an entirely bug-free implementation of the entirety of the Intel 8086, Intel 8259 Programmable Interrupt Controller, Intel 8253 Programmable Interval Timer, etc.

As for more realistic, concrete success criteria, a few points have since been decided upon. These are outlined in table 1.

Success Criteria	Evidence	Justification
Has emulated memory which can be read and written to by the emulated CPU without error.	Ensure that memory read/writes correctly alter the emulator memory and that erroneous data (e.g. reading address out of bounds) is also handled appropriately (throw an exception rather than reading/editing unallocated memory).	Important as all computer systems require primary storage for storing programs and data. Issues with this storage will cause serious knock-on effects for the rest of the system also.
Capable of executing instructions, including those that have MOD-REG-R/M, immediate and/or displacement bytes.	I will write a collection of unit tests that ensure each individual class and method involved in instruction decoding and execution function as expected.	This is necessary to ensure the emulator can correctly execute the various instruction types as expected.
Capable of producing correct assembly representations of instructions, including those that have MOD-REG-R/M, immediate and/or displacement bytes.	I will a write a collection of unit tests that test the individual assembly generation for each component of an instruction is as expected.	The stakeholder's students should be able to see x86 instruction decoding in action - showing disassembly is an easy way of showing how raw bytes of data correlate to actual CPU instructions.
Emulator not crash when provided with an unknown/unimplemented opcode or instruction with invalid encoding.	Have the emulator attempt to run instructions that are known to not be supported/have invalid MOD-REG-R/M bytes and ensure that a clear error message is displayed rather than the program simply crashing or doing nothing.	This is necessary to ensuring the stability to the program - especially as users are capable of editing the data in memory that is to be executed by the CPU.

Table 1: *Success criteria for the emulator itself (not the GUI).*

3 Intel 8086 Microprocessor

This section consists of an implementation non-specific overview of the how the Intel 8086 is structured and functions at a high-level.

3.1 General-Purpose and Index Registers

The Intel 8086 has four different 16-bit general-purpose registers (AX, BX, CX, DX), which can also be accessed as twice as many 8-bit registers (AL and AH, BL and BH, CL and CH, DL and DH). These registers can be used for any purpose but are implicitly used by some instructions.

There are also four 16-bit index registers on the 8086:

- Source Index (SI) - Points to the source in the current operation.
- Destination Index (DI) - Points to the destination in the current operation.
- Base Pointer (BP) - Points to the base of the stack within the stack segment.
- Stack Pointer (SP) - Points to the most recent value on the stack within the stack segment.

Note that unlike the general-purpose registers, the index registers are 16-bit only.

My emulator solution will implement all of the registers (general purpose and index) outlined above.

3.2 Status Register/Flags

There are a set of flags in the 8086 that contain information about the state of the CPU. These flags are stored within a single 16-bit value, of which nine of those bits are used (see Table 2).

Flags		Bit	Purpose
Carry Flag	(CF)	0	Set when an arithmetic operation resulted in a bit being carried.
Parity Flag	(PF)	2	Set if the binary representation of the result of the last operation is made up of an even number of bits.
Axillary Carry Flag	(AF)	4	Set when an arithmetic carry or borrow is generated out of the four least significant bits of an operation.
Zero Flag	(ZF)	6	Set when the result of the last arithmetic operation is equal to zero.
Sign Flag	(SF)	7	Set if the result of the last mathematical operation had its most significant bit set.
Trap Flag	(TF)	8	When set the processor will generate an internal interrupt after executing each instruction. This is for debugging purposes.
Interrupt Flag	(IF)	9	The processor will only recognise interrupt requests from peripherals when this flag is set. Otherwise, requests will be ignored.
Direction Flag	(DF)	10	Specifies the ordering of bytes when handling strings.
Overflow Flag	(OF)	11	Set when the last operation resulted in integer overflow.

Table 2: *The nine different true/false flags stored in the FLAGS register.*

This project will feature the flags above however they will be represented as a collection of boolean values rather than a single 16-bit value.

3.3 Memory Segmentation and Segment Registers

The 8086 has an unusual method of addressing up to one megabyte of memory. To find an absolute/physical address, the CPU shifts the 16-bit segment register four bits left and then adds the 16-bit offset address. Simply concatenating the segment and the offset to form a 32-bit absolute address, while slightly simpler, was deemed excessive for the time and would have required more expensive external bus pins.

There are four different memory segments within the Intel 8086:

- Code Segment (CS) - The segment in which the executable program is stored. The instruction pointer is segmented within the code segment.
- Data Segment (DS) - Where any data used by the program is stored.
- Extra Segment (ES) - An additional area in which to store program data.
- Stack Segment (SS) - Where the contents of the stack is stored. All stack pointers (base pointer and stack pointer) are segmented within the stack segment.

My emulator will implement the form of memory segmentation outlined above as well as the four segment registers (code, data, extra, stack).

3.4 Instruction Encoding

The x86 instruction set is an example of a Complex Instruction Set Computing (CISC) instruction set and it most certainly lives up to such a description. Each individual instruction can span from one to six bytes in length and is made up of several components which are outlined in the subsections below.

Full i8086 instruction decoding will be a feature of my own emulator.

3.4.1 Opcode

While later x86 processors allowed for two byte opcodes, the 8086 only has 8-bit single byte opcodes. All instructions will have an opcode value and this value specifies what the instruction will do when executed as well as the ordering and types of parameters this instruction requires.

The second-to-least significant bit of an opcode is known as the direction or `d` bit and is used to indicate whether the register specified within the instruction is a source (`d=0`) or a destination (`d=1`).

The least significant bit of an opcode is known as the word or `w` bit and indicates whether the instruction will operate on 16-bit word values (`w=1`) or 8-bit byte values (`w=0`).

3.4.2 MOD-REG-R/M

Many instructions have a single MOD-REG-R/M byte immediately after the opcode. This byte specifies instruction operands and their addressing mode.

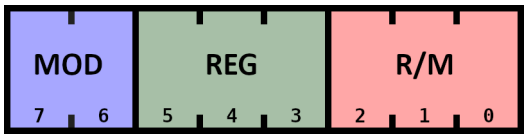

Figure 5: *The structure of a MOD-REG-RM byte.*

As shown in Figure 5 above, bits 7 and 6 of the MOD-REG-RM byte comprise the MOD component. These two bits specify the addressing mode used (see Table 3).

MOD Value	Addressing Mode
00	No displacement.
01	One byte displacement follows MOD-REG-R/M byte.
10	Two byte displacement follows MOD-REG-R/M byte.
11	R/M component is treated as a second register field (register addressing mode).

Table 3: *Shows how the MOD bits of a MOD-REG-RM byte affect the addressing mode used.*

Bits 3 to 5 make up the REG component of the MOD-REG-RM byte. This component specifies the register used (whether it is a source or destination is dependent on the 'd' bit of the opcode). Table 4 shows which register will be used based on both the REG value and whether the instruction is operating on 8-bit or 16-bit values.

REG Value	Register if 8-bit	Register if 16-bit
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Table 4: *Show which register a REG value corresponds to (or R/M value when in register addressing mode as indicated by MOD value).*

The R/M part of the MOD-REG-RM byte is found in bits 0 to 2. When in register addressing mode (`MOD=11`), the R/M component, like the REG component, indicates a specific register (see table 4). See the Displacement section below for the meaning of R/M when MOD is equal to other values.

3.4.3 Displacement

Depending on the value of MOD in the MOD-REG-RM byte, a instruction may be encode with up to two bytes of displacement. When MOD is either `01` or `10` (indicating either 8 or 16 bit displacement), the value of R/M will indicate against which index registers the displacement will be applied to (see table 5).

R/M Value	Operand Address
000	BX + SI + Displacement
001	BX + DI + Displacement
010	BP + SI + Displacement
011	BP + DI + Displacement
100	SI + Displacement
101	DI + Displacement
110	BP + Displacement
111	BX + Displacement

Table 5: *Shows the address used as an instruction operand based on R/M value (assuming MOD indicates that the R/M component is not being used as a second register field).*

Note that BX is the only general-purpose register which can also be used for indexing (as seen in table 5).

3.4.4 Immediate

An immediate data value can be encoded directly into an instruction. This value can be a single byte or a 16-bit two byte value encoded in little endian (depending on the data size of the instruction).

4 Intel 8259 Programmable Interrupt Controller (PIC)

Note that, while it was indeed originally planned, the PIC component of the emulator was never implemented due to time constraints. As such, this section is not relevant to/be a feature of the emulator itself.

The original Intel 8086 PCs relied upon a chip separate to the main Intel 8086 microprocessor called the Intel 8259. This chip managed several hardware interrupts (also known as Interrupt Requests or IRQs) and informed the processor when an interrupt required handling as well as which Interrupt Service Routine (ISR) should be called in order to do so. The Intel 8259 offered eight different interrupt pins numbered from 0 to 7 where IRQ 0 had highest priority while IRQ 7 the lowest.

The 8259 PIC functions as something of a multiplexer as its eight possible interrupt pins fire just a single maskable interrupt pin on the processor. The processor will ignore all interrupt requests from the PIC indicated by the interrupt pin if its Interrupt Flag (IF) is set to 0. In this case, all external interrupts will effectively become disabled.

4.1 Registers

The 8259 PIC has three 8-bit registers which determine its behaviour.

4.1.1 Interrupt Mask Register (IMR)

The IMR allows for the individual interrupts provided by the PIC to be masked/disabled. The 8 bits of the IMR each correspond to an IRQ pin with the least significant bit corresponding to IRQ 0 and the most significant corresponding to IRQ 7. When an interrupt's masking bit within the IMR is set to 1,

the interrupt will be masked meaning it will not be able to interrupt the processor. If an interrupt's bit is 0 however, then it will indeed interrupt the processor should the interrupt trigger.

4.1.2 Interrupt Request Register (IRR)

The IRR indicates when an interrupt has been fired. Like the IMR, each bit of the IRR corresponds to one of the eight interrupts. As soon as a device signals an interrupt then the appropriate bit in the IRR will be set to 1. This register can only be modified by the PIC itself.

4.1.3 In-Service Register (ISR)

The ISR indicates which interrupts are currently being serviced/handled (where execution has begun but is not complete).

4.2 Interrupt Vector Table

The Interrupt Vector Table holds the addresses for every interrupt handler that may be required by the system (both hardware and software interrupts). On the Intel 8086, this table always resides in memory from 0x0000 to 0x03FF and consists of 256 four-byte far pointers (two-byte segment and two-byte offset pairs).

4.3 Interrupt Firing/Handling

When an interrupt occurs, assuming the interrupt is not masked (in the IMR) and the processor has interrupts enabled (IF set), then the following takes place:

- The PIC asserts the processor's interrupt pin.
- Before it handles the interrupt, the processor finishes the execution of the current instruction.
- The processor sends a signal of acknowledgement back to the PIC who then in turn sends the vector number of the interrupt back to the processor.
- The vector number returned by the PIC is used as an index in the Interrupt Vector Table which is created by the BIOS. The corresponding entry in the interrupt vector table contains the address (segment and offset) for the appropriate interrupt service routine.
- The processor clears IF (disabling interrupts) and pushes FLAGS, CS and IP onto the stack.
- The processor then jumps to the address of the interrupt service routine by setting CS and IP to the segment address and offset respectively.

5 Intel 8253 Programmable Interval Timer (PIV)

Again, like the PIC, it was not possible to implement the PIV as originally planned due to time constraints. As such, this section is not relevant to the emulator itself.

The typical Intel 8086 system relies on a separate PIV chip in order for precise timing and counting functions. It offers three separate channels for this purpose labelled 0, 1 and 2.

In IMB PC compatible devices, Timer Channel 0 fires INT 8 on IRQ 0 (highest priority interrupt offered). This channel is implemented as a descending counter where, once the initial value is set, it will repeatedly count down from that value. Computers running MS-DOS or similar operating systems typically have the first timer channel operate at a frequency of 18.2 Hz.

5.1 Control Word

The CPU is able to set the operating mode and format for each channel of the interval timer by writing an 8-bit control word to the PIT.

The two most significant bits of the control word indicate (when not 11 as that is unused by the 8253) the channel this control word will affect (expressed in binary). The following two bits indicate the ordering of read and writes of the high/low bytes of the counter register. The next three bits indicate the timing mode while the final bit indicates whether the counter will operate in binary or binary-coded decimal (note that the latter is almost never used).

6 Design

This section of the document outlines all the key design decisions made when building WiredEmu, both internally and in terms of aesthetics and GUI design.

6.1 Coding Style

Unlike perhaps most mainstream programming languages, C++ has no real consensus in terms of coding style, use of case and naming conventions. In order to maintain consistency across the project therefore, in this section I shall outline the code style adopted across the project (of course, external libraries often use various different styles that do not align with my own but that is unfortunately unavoidable).

6.1.1 Use of Case

Identifiers that should use camelCase:

- Local and member variables `int x;`
- Namespaces `namespace ns`
- Function and method names `void func()`
- Function and method parameters `void func(int parameter)`

Identifiers that should use PascalCase:

- Class names `class ClassName`
- Enumeration names `enum EnumName`
- Structure names `struct StructureName`
- Type definitions/aliases `using TypeName = ...;`
- Template parameters representing types `template <typename TypeTemplateParameter, unsigned int notTypeParameter>`

Identifiers that should use UPPER_CASE:

- Macros `#define MACRO_NAME ...`
- Enumeration values `enum { FIRST_VALUE, SECOND_VALUE };`
- Constants where the value is known at compile-time and will never change throughout the execution of the program (this is something which is not guaranteed when using the `const` keyword depending on how and where it is used)

6.1.2 Whitespace

Spaces will be used instead of tab characters. A normal indentation should be four spaces. I chose to use spaces over regular tabs as they allow for one to be sure that more complex use of indentation will visually appear the same for all regardless of text editor used.

All mathematical operators, with the exception of unary operators, should be padded with spaces on either side (for example, `10 + (2 * a)` and `-x + 5`).

When using pointer or reference types, have the `*` or `&` attached to the type name and not the identifier name (for example, `unsigned int* x;`).

All elements in a single-line brace initialisation should be spaced as such (each element having a space on either side): `{ 5, 10, 15 }`

6.1.3 Naming Conventions

Names like `i` and `j` are acceptable for numerical counters for short loops. For longer loops or loops where a collection of items are iterated through then more descriptive names are preferred (for example, use `for(auto child : children)` instead of `for(auto i : children)`).

Names for namespaces should be kept short whenever possible so as to discourage the use of `using namespace veryLongNamespaceName;` (names like `emu` or `gui` are acceptable for namespaces though make sure to document their purpose when using such short names).

Class and type names are not recommended to be kept as short as namespace names and therefore can and should be more descriptive (for example, use `class InterruptHandler` instead of `class IntHandle`). Frequently used type aliases can have shorter names for the sake of convenience.

6.1.4 Examples

```
1  // The first enumeration item may appear on the same line as the enum keyword.
2  // In such an instance, all subsequent items should align with the first.
3  enum EnumType { FIRST,
4                  SECOND,
5                  THIRD };
6
7  // Alternatively, all enumeration items may be defined on newlines using
8  // regular indentation of four spaces.
9  enum EnumType {
10     FIRST,
11     SECOND,
12     THIRD
13 };
14
15 // If an enumeration only contains a few short items then it is acceptable to
16 // put it all on a single line.
17 enum EnumType { FIRST, SECOND, THIRD };
```

Listing 1: *Example of coding style to use when defining enumerations.*

6.2 Project Structure

As the most basic level, I opted for the typical C++ project structure wherein there are two primary directories: `src` (stores all source/implementation files) and `include` (stores all header files). It is standard practice when writing in C++ to keep the definition of classes, functions and data structures

```
class ClassName : public Base {
protected: // Note that visibility keywords should be unindented.
    ClassName(int x, int y); // The first declaration in a class should be the
                             // constructor (assuming the class has one).

public: // Next should be all public declarations with methods before member
        // variables.
    void publicMethod();

    Type method(float z) const;
    Type method(float z, bool a) const; // Group similar methods by not
                                        // separating them with an additional
                                        // newline.

    int member, otherMember; // Only define multiple variables using comma
                             // separation if the two values are closely
                             // related (e.g. pair of x and y coordinates).

protected: // Protected declarations after public.
    using MyType = unsigned int; // Using declarations come before methods.

    virtual void update() = 0;

private: // Finally, all private declarations come last.
    int overridable() override; // If a method in a base class is being
                                // overriden then the override keyword must be
                                // used.

    const MyType privateMember;
};
```

Listing 2: *Example of coding style to use when declaring a class or structure.*

entirely separate from their implementation whenever possible as this allows for significantly improved build times when only minor changes to the code are made. Definitions are found in header files (`*.hpp`) in the `include` directory while implementations are found in source files (`*.cpp`) in the `src` directory. I considered using the new module system planned for C++ 20 over the header/source method, however support for modules is still somewhat patchy as the C++ 20 standard is not yet fully finalised at the time of writing.

If in order to maintain a degree of modularity and allow for the use of the emulator core as a general-purpose library without any GUI/CLI/unit tests, I devised a system wherein the code base is split into four different groups/build targets: `common`, for all emulator and shared code; `cli`, for command-line interface code; `gui`, for graphical user interface code; and `test`, for unit testing. While `cli`, `gui` and `test` are executable build targets (i.e. they each have a `main` function and generate standalone executables), `common` is a non-executable library which is linked again each of the aforementioned executables.

The directory structure inside of both `src` and `include` mirror the structure of the project's nested namespaces for the most part. For example, the definition of the class `emu::cpu::Intel8086` (part of the `common` build target) can be found in `include/common/emu/cpu/intel8086.hpp` and its implementation in `src/common/emu/cpu/intel8086.cpp`.

6.3 Namespaces

- `emu` - Contains all emulator code. Part of `common` build target.
  - `emu::cpu` - Contains emulator code specific to the Intel 8086 Central Processing Unit.
    - `emu::cpu::reg` - Code related to CPU registers.
    - `emu::cpu::instr` - Code specific to the representation of CPU instructions.
- `convert` - Contains functions and function templates for the conversion between different types and formats of data. Part of `common` build target.
- `gui` - Contains all GUI code. Part of `gui` build target.

6.4 Build System

C++ is known for being notoriously complex and inconsistent to build across different platforms. Fortunately, tools like CMake make this *somewhat* easier. CMake is a tool essentially for configuring whatever build system may be available on the given system. For example, on my Linux computer, CMake writes a Makefile which utilises the GCC compiler but, on Windows, it generates a Visual C++ Project file which uses Microsoft's Visual C++ compiler.

CMake requires a configuration file called `CMakeLists.txt` which, despite having the `.txt` file extension, contains code in a domain-specific language unique to CMake. This allows one to specify the details of where header files are contained, which source files to compile, libraries to link against, and any other factors that may need to managed in order to build a codebase.

My initial `CMakeLists.txt` was rather simple - it stated that the `include` directory contained header files, to recursively look in the `src` directory for source files, and that an executable called `WiredEmu` should be created.

Naturally, as the project grew more complex, so did its build script. While initially the emulator had only a command-line interface, I soon decided it would be necessary to implement a full graphical user interface. In addition, it was believed that adding unit testing functionality would benefit the stability of the program and assist with debugging. The current build script and version of CMake used did not support such features. As such, I updated CMake and began to make use of its modern features - the primary of which being its build target system. It was this feature that allowed for the building of the various different executables described in section 6.2.

Modern CMake allows the creation of distinct build targets which can each have unique properties for both themselves as well as for those they are linked against/depended on by. This is done by specifying public, private and interface properties with private properties applying only to the target; interface properties only applying to other targets dependent on the target; and public properties which are equivalent to applying both of the former. As an example, this would allow for a library to be built using C++ 17 (private property) but only require targets that depend on it to be built with a minimum of C++ 11 (interface property).

6.5 GUI Design

As you can see in the initial design in Figure 6, the GUI for this project was based primarily around three main components: a hex editor (shown in the top-left of the design), a status/control panel (bottom-left), and various register panels (right).

The hex editor would display and allow the user to alter system memory, the status/control panel would show the disassembled instruction currently being run and will the user to play/pause execution as well as change execution speed, while finally the register panels will each display and will allow for the editing of the values of all the different register groups (general-purpose, segments, index registers, stack registers and FLAGS register).

A new design was created in order to be more applicable to use with the chosen GUI library (ImGui). See figure 7. This design differs from the previous one by having separate 'sub-windows' for the hex editor, control panel, and registers panels. In addition, this revised GUI design shows more of the specifics of the layout within even panel (the original design instead only showed the general arrangement of the panels themselves).

Down the left-hand side of this are collection of 4 register-related GUI panels. The first (top-left) shows

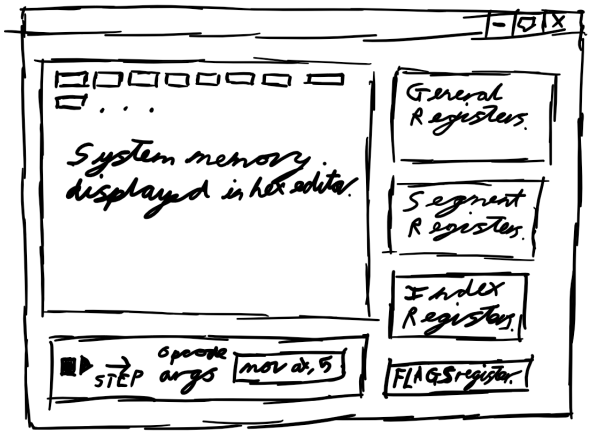

Figure 6: *Initial GUI design drawn on a Surface Pro tablet.*

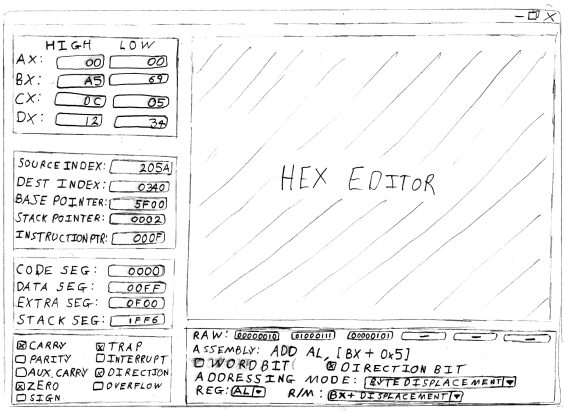

Figure 7: *Revised GUI design drawn using pencil and paper.*

the CPU's general-purpose 16-bit registers registers split into their two 8-bit components each (high and low bytes) in a 2 by 4 grid. This grid layout is ideal as it reduces the visual clutter of having three labelled fields each (AX, AH, AL, etc.) for what is effectively just a set of four single registers. Indeed, reducing the complexity of the GUI while still providing a high level of functionality is key, especially for the stakeholder's needs (use in a learning environment).

Below the general-purpose registers panel are the index and segment register panels. Note how unlike many other emulators, my emulator will give the full name of each segment and index register rather than just the acronyms (for example, 'SOURCE INDEX' instead of just 'SI'). This felt necessary so that students did not have to constantly look up what each individual acronym means. In addition, since index and segment registers can only be accessed as full 16-bit values (not as individual high/low bytes like with the general-purpose registers), these panels do away with the 2 columns used in the general-registers panel.

The bottom-left panel of the revised GUI design shows the CPU flags. While the nine status flags on the real Intel 8086 processor are stored as bits of a single 16-bit FLAGS register, my solution instead stores them as just a collection of nine boolean values. This GUI design follows suit by simply having each flag next to a checkbox toggle to clearly show whether it is or is not enabled. Full names of flags are also displayed instead of the shorthands/acronyms seen in many pieces of documentation (for example, 'PARITY' instead of 'PF').

6.6 Instruction Representation

Decoded instructions are represented using complex data structures making heavy use of object-orientation and inheritance. This allows for logical encapsulation and for more efficient efficient code reuse. Note that the eplanations below do not provide full namespaces of the referenced objects/functions/etc so as to aid readability.

The most basic instruction class is simply titled `Instruction` and holds basic attributes that are components of every x86 instruction. These include an opcode (represented using a `Opcode` object) and an assembly identifier (e.g. `add`, `jnz`). Note that this class is purely virtual/abstract and cannot be initiated without the overriding of its three virtual methods.

One method of `Instruction` that should be overridden is:

```
virtual OffsetAddr execute(Intel8086& cpu, Mem& memory) = 0;
```

As the name would suggest, it is the role of this method to executed the instruction using the given `cpu` and `memory` references. Note that these are mutable (i.e. non-constant) references meaning modification is allowed (an instruction unable to modify CPU or memory state would naturally have very limited use). This is in contrast to the `toAssembly` method discussed shortly which takes constant references so as to prevent accidental modification of the passed CPU and memory.

The `execute` method is required to return an `OffsetAddr` (an alias for a unsigned 16-bit integer type). The CPU's instruction pointer will be set to this returned value on execution completion. For most instructions other than jump and interrupt instructions, the address of the next instruction (and thus the appropriate return value) will simply by the current instruction pointer plus the length of the encoded instruction in bytes. As such, a helper method is provided:

```
OffsetAddr nextAddress(const Intel8086& cpu) const;
```

The above helper method allows most instructions to end their `execute` method simply with the statement:

```
return nextAddress(cpu);
```

The `Instruction` class also has the following virtual method:

```
virtual std::string toAssembly(const Intel8086& cpu, const assembly::Style& style) const;
```

6.7 Usability Features

For a piece of software that is supposed to be used by students not yet acutely familiar with processor architecture, the usability of said software is a key concern.

The first major consideration when it comes to usability, is the general layout of the software's GUI. One issue with many similar solutions is an intimidating and complex-to-navigate user interface. This is something I hope to address properly with my own solution. Please see section 6.5 for a breakdown for the GUI design process and an outline of some more general usability features of the software.

After discussions with the shareholder, we agreed on a particular feature designed to aid usability of the software to quite an extent. This feature is the introduction of 'tool tips' - short paragraphs of description displayed when the user hovers their mouse over a particular area of the GUI. This is beneficial as it allows for details on appropriate software usage to be provided in a streamlined manner (e.g. without having to refer to separate usage instructions). In addition, having this information only appear on mouse hover prevents additional clutter in what is already a somewhat intimidating user interface.

6.8 Testing

Naturally, it is difficult to decide on the specifics of testing before any code has actually been written. The current plan is to write a collection of unit tests during the iterative development process in order to test the internals of the software.

6.8.1 Unit Testing

When working on larger projects with expansive code bases, it can quickly become inconvenient to manually test each individual portion of said code base. Fortunately, unit testing allows for a degree of automation in the process of ensuring each component or 'unit' that makes up a piece of software functions as it should. This is typically done by writing tests that give certain inputs to pieces of code and will only pass should the expected outputs are received. Should a test fail then a report explaining which portion of code failed, what it was supposed to do, and what it actually did is generated.

Initially I wrote my own unit testing framework. This framework was built around constructing an object holding a 'test function' (lambda/anonymous function) which performs an assertion by either returning true to indicate a pass and false to indicate failure. This unit test object then had various methods available which allowed one to pass input to the testing function and then have the result of its assertion handled appropriately.

While this method did indeed work, it presented a couple of issues. The first of which being the limited information given when a test failed. I had made a `CREATE_TEST` macro which passed information such as in which file and function the test was created to the the unit test object, which gave some degree of insight into how to test failed. More information such as the precise value of arguments to the test

function however, would be helpful to provide for debugging purposes. While I was able to just use parameter pack expansion in order the write most argument values to standard output, more complex values (such as `std::vector` values or values of custom types) cannot be handled be `std::cout` by default. While I could have written a custom output stream derived from `std::ostream` to handle these types, doing so would be rather time consuming.

Another issue with my own unit testing framework was the lack of categorisation of tests. The implementation did not allow for the tagging or running of individual tests and instead always ran every test defined. This was inefficient and did not allow for optimal organisation when a large quantity of tests are defined.

After realising that improving my own unit testing framework could be an entire project in and of its self, I decided to utilise a third-party framework so that I would be able to place all focus on coding the emulator portion of the project. The framework I chose after a few days of deliberation was **Catch2**. Features that drew me to this particular framework include the ability to search for and tag tests, the fact that far more detailed information about failed tests are provided, and the ability provided by this framework to split tests into separate subsections.

6.8.2 Assembly Generation

Every CPU cycle, the instruction at the point in memory indicated by the instruction pointer is converted into its human-readable assembly representation. This assembly representation is displayed to the user so that it is clear to them what an instruction will do before it is actually run. I concluded the best way to test the validity of the emulator's generated assembly would be compare it to that of a pre-existing disassembler.

The GNU project provides a piece of software `objdump` which allows for the in-depth analysis of object code. Using the command-line `-D` option, `objdump` will produce assembly code from the specified object file. The `-Mintel` option is employed also so as to specify the use Intel x86 syntax rather than AT&T as the former is the style I have programmed the emulator to output. The argument `-m i8086` is used to specify the Intel 8086 as the target architecture. For the sake of completeness, the `-b binary` option is also used to specify the input file's binary format (including this option is not really necessary as `objdump` is quite good at inferring the format used).

Disassembling the code in a file called `a.out` could therefore be done with the following command:

```
objdump -Mintel -D -b binary -m i8086 a.out
```

I would perform these tests based on each general type of instruction so as to ensure each variety is functional.

6.8.3 Instruction Execution

Ensuring instructions actually perform the correct changes to CPU state (i.e. execute correctly) is to be done using a collection of unit tests. As mentioned in the Analysis section, I will use the Catch2 library for creating, managing, and running unit tests.

```
1  SECTION("Test arithmetic instructions.") {
2      cpu.generalRegisters.set(cpu::reg::CX_REGISTER, 5);
3      cpu.generalRegisters.set(cpu::reg::BX_REGISTER, 10);
4
5      cpu.performRelativeJump(0); // Reset instruction pointer to 0.
6      memory.write(0, { 0b00000001, 0b11011001 }); // add cx, bx
7
```

Test Data	Type	Description
write byte to address `0x0F`	normal	write byte value to valid memory address (within bounds)
write byte to address `0xFE`	boundary	write byte value to address just within memory bounds
write byte to address `0x5FF`	erroneous	attempt to write memory value that is out-of-bounds so exception should be thrown
write several bytes beginning from address `0x10`	normal	to valid memory addresses (within bounds)
write several bytes beginning from address `0xFD`	erroneous	not all memory addresses will be within bounds so exception should be thrown
read byte from address `0x0F`	normal	read byte from valid memory address (within bounds)
read byte from address `0xFE`	boundary	read byte from address just within memory bounds
read byte from address `0xABCD`	erroneous	attempt to read byte that is out-of-bounds so exception should be thrown
read several bytes from address `0x15`	normal	from valid memory addresses (within bounds)
read several bytes from address `0xFD`	erroneous	not all memory addresses will be within bounds so exception should be thrown

Table 6: *Table of emulator memory test data.*

```
8       auto add = cpu.fetchDecodeInstruction(0, memory);
9       cpu.executeInstruction(add, memory);
10
11      REQUIRE(add->toAssembly(cpu, assembly::Style()) == "add cx, bx"); // Test assembly generation.
12      REQUIRE(cpu.generalRegisters.get(cpu::reg::CX_REGISTER) == 15); // Test execution.
13  }
```

Above is an example of what one such unit test may look like. This approach is ideal as it allows me to test all instructions by simply running the generated unit testing executable.

6.8.4 Emulator Memory

Having memory representation for the emulated system that functions as expected is also vital as any system, emulated or otherwise, will frequently write to and read from memory. Again, unit testing is ideal in this situation.

Testing data table 6 works on the assumption that memory consists of a total of 0xFF separate 8-bit values.

6.8.5 Memory Segmentation

As before, unit testing will also be used for the testing of memory segmentation. See table 7.

Segment Register Value	Offset Value	Expected Absolute Address	Test Data Type
0000	0000	00000	normal
00FF	0000	000FF	normal
00FF	000F	00FFF	normal
FFFF	000E	FFFFE	boundary
FFFF	FFFF	-	erroneous

Table 7: *Table of example memory segmentation testing data.*

6.8.6 Post-Development Testing

...

7 Implementation: First Stage

7.1 Development

7.1.1 Setting Up Git

By nature of using Git for version control, the first step to beginning the project is to create a `.gitignore` file:

```
*.out
*.exe
build/
```

Now the project folder can be initialised as a git repository:

```
max@virtual ~/Project git init
Initialized empty Git repository in /home/max/Project/.git/
max@virtual ~/Project  master git add .gitignore
max@virtual ~/Project  master + git commit -m "Initial commit."
[master (root-commit) 606aa9c] Initial commit.
 1 file changed, 3 insertions(+)
 create mode 100644 .gitignore
max@virtual ~/Project  master
```

As mentioned in the Design section, the project code is split into two primary directories: `include/` and `src/` with header files stored in the former and source/implementation files in the latter. Inside both of those directories are directories both called `common`. It is in those directories where the general emulator code is kept (i.e. code not specific to the GUI, testing or CLI).

7.1.2 Primitives

The first step taken in beginning the actual implementation was to define a collection of primitive types. This was done for a number of reasons. Firstly, the typical built-in types provided in C++ (e.g. `unsigned int`) have a size that is dependent on the target system and compiler used. The standard library header `cstdint` provides types such as `uint16_t` which are of a definite size (unsigned 16-bit integer in the case of the aforementioned). The issue with these however is that, depending on the compiler, they may be found under the `std::` namespace only or also in the global namespace. In ensure compatibility across all compilers, one would have to use these types with the `std::` prefix or make use of the `using namespace` command for each type they wish to use (so that they are guaranteed to be in the global namespace).

To resolve this issue, as an alternative to making use of the aforementioned 'using' command, I decided to create type aliases for each of these primitive types with their `std::` namespace prefix specified. This also allowed me to give cleaner names without the `_t` extension in order to aid readability.

The code of the `primitives.hpp` header is found in the `include/common/` directory and is shown below:

```
#pragma once

#include <cstdint>

using u8 = std::uint8_t;
using u16 = std::uint16_t;
using u32 = std::uint32_t;

```

```
using i8 = std::int8_t;
using i16 = std::int16_t;
using i32 = std::int32_t;
```

7.1.3 Memory

An element of the C++ compilation system is that it is advantageous in terms of compile time to keep the declaration of class separate from the implementation of any of its member functions. Class declarations are defined in header files (`.cpp`) while class implementations are defined in source files (`.hpp`). In the case of class templates however, the compiler does not allow the implementation to be kept separate from the declaration. This is seen in the `Memory` class template that follows:

```
#pragma once

#include <memory>
#include <vector>
#include <exception>

namespace emu {
    template <typename Value, typename Address>
    class Memory {
    public:
        /**
         * Exception thrown when a call to read or write is supplied with an address that is out of bounds.
         */
        class OutOfBounds : public std::exception {
        public:
            OutOfBounds(Address addr) : address(addr) {}

            const Address address;
        };

    public:
        Memory(Address memorySize) : size(memorySize), mem(new Value[size]) { fill(); }

        /**
         * Check if the address passed is within bounds of the memory allocated.
         *
         * @param address The address check.
         * @return Whether the address is within bounds or not.
         */
        bool withinBounds(Address address) const {
            return address < size && address >= 0;
        }

        /**
         * Fill all memory with the specified value (defaults to 0).
         *
         * @param value Value to fill memory with.
         */
        void fill(Value value = 0) {
            for(Address addr = 0; addr < size; addr++)
                write(addr, value);
        }

        /**
         * Read the value held in memory at the given address.
         *
         * @param address The address of the value to read.
         * @return The value read.
         */
        Value read(Address address) const {
            assertWithinBounds(address);
            return mem[address];
        }

        /**
         * Read multiple values from memory.
         *
         * @param startAddress The starting address to begin reading from.
         * @param amount Number of values to read.
         * @return Vector of values read.
         */
        std::vector<Value> read(Address startAddress, Address amount) const {
            std::vector<Value> values;

            Address address;
            Value value;

            for(Address offset = 0; offset < amount; offset++) {
                address = startAddress + offset;
                value = read(address);

                values.push_back(value);
            }

            return values;
        }

        /**
         * Write a value to memory at the given address.
         *
```

```
81          * @param address The address to be written to.
82          * @param value The value to write.
83          */
84         void write(Address address, Value value) {
85             assertWithinBounds(address);
86             mem[address] = value;
87         }
88
89         /**
90          * Write multiple values to memory beginning from the specified address.
91          *
92          * @param address The startAddress from which to begin writing.
93          * @param values Vector of values to write.
94          */
95         void write(Address startAddress, std::vector<Value> values) {
96             Address address;
97             Value value;
98
99             for(Address offset = 0; offset < values.size(); offset++) {
100                address = startAddress + offset;
101                value = values[offset];
102
103                write(address, value);
104            }
105        }
106
107        // TODO: Add methods for reading from/writing to files.
108
109        const Address size;
110
111    protected:
112        /**
113         * Checks if the given address is within bounds, throwing an OutOfBounds exception if that is not the case.
114         *
115         * @param address The address to check.
116         */
117        void assertWithinBounds(Address address) const {
118            if(!withinBounds(address)) throw OutOfBounds(address);
119        }
120
121    private:
122        std::unique_ptr<Value[]> mem;
123    };
124}
```

It was decided that this would indeed be a class template simply to make the code as generic as possible. Should, in some later stage of development, be decided that the values stored in memory or memory addresses would be a different numerical type, the use of this class template makes that an easy change to make.

There is an exception class that derives from `std::exception` defined within the `Memory` class template. Nesting these classes was done so that the exception class could have a member that is the correct type to hold the out-of-bounds address that caused the exception to be thrown.

The memory values themselves are stored as a raw array, the size of which is not known at compile-time (meaning it has to be dynamically-allocated). This is array is kept within a `std::unique_ptr` so that memory will be deallocated automatically when no longer needed. The pointer to this array is declared as private meaning it can only be used within the class itself. This class makes use of encapsulation in the sense that only methods that indirectly modify the internal data can be used from the outside. This has the advantage of ensuring safety as it means users of the class cannot tamper with memory or cause issues by providing invalid values (e.g. attempting to write a value to memory beyond the amount allocated).

In terms of the public interface, methods for reading from and writing to memory are provided as well as the methods to check whether a given address is within memory bounds or to fill/clear all memory values.

7.1.4 CPU

Next, design of the general Intel 8086 class began:

```
1  #pragma once
2
3  #include <memory>
```

```
#include "common/primitives.hpp"
#include "common/emu/memory.hpp"
#include "common/emu/cpu/registerindexes.hpp"
#include "common/emu/cpu/registers.hpp"

namespace emu::cpu {
    /// Absolute address on the 8086 are 20-bit however no 20-bit unsigned integer type exists in C++ so a 32-bit
    /// unsigned integer is used instead.
    using AbsAddr = u32;

    /// Offset addresses within a given segment are 16-bit wide.
    using OffsetAddr = u16;

    /// Values stored in memory are 8-bit wide.
    using MemValue = u8;

    class Instruction; // TODO: Temporary!

    /**
     * Class representing the main Intel 8086 microprocessor. Handles decoding and execution of instructions fetched
     * from memory.
     */
    class Intel8086 {
    public:
        /**
         * Takes a 16-bit memory offset and a 16-bit segment register and returns an absolute 20-bit address (which is
         * stored in an unsigned 32-bit integer since there is no 20-bit integer type available in C++).
         *
         * @param offset The memory offset within the given segment.
         * @param segment Segment register index indicating which segment to resolve the offset within.
         * @return Absolute 20-bit address within memory.
         */
        AbsAddr resolveAddress(OffsetAddr offset, SegmentIndex segment) const;

        /**
         * Calculate the address of the next instruction in memory based on the value of the instruction pointer and the
         * code segment.
         *
         * @return Address of next instruction.
         */
        AbsAddr nextInstructionAddress() const;

        /**
         * Fetches and decodes the next instruction.
         *
         * @param addr The absolute address of the instruction to fetch and decode.
         * @param memory Reference to the memory to fetch the instruction data from.
         * @return Decoded instruction object.
         */
        std::unique_ptr<Instruction> fetchDecodeInstruction(AbsAddr address, Memory<MemValue, AbsAddr>& memory) const;

        /**
         * Executes a decoded instruction on this CPU.
         *
         * @param instruction Reference to the std::unique_ptr holding the instruction.
         */
        void executeInstruction(std::unique_ptr<Instruction>& instruction);

    private:
        /// The instruction pointer is an offset within the code segment that points to the next instruction in memory.
        OffsetAddr instructionPointer = 0;

        RegistersLowHigh<GeneralIndex> generalRegisters;
        Registers<SegmentIndex, u16> segmentRegisters;
    };
}
```

This began with defining some type aliases to make it more explicit what a value's purpose exactly is. For example, it isn't immediately obvious why the `resolveAddress` method returns an unsigned 32-bit integer, so having it return such a type under the alias `AbsAddr` indicates that it is returning an absolute memory address and therefore reduces ambiguity.

Completing fetch-decode-execute cycle with this class design requires a few steps:

1. Fetching the address of the next instruction (i.e. instruction pointer segmented within the code segment) by calling the `nextInstructionAddress` method.

2. Passing that address and a `Memory` object to the `fetchDecodeInstruction` method which will return the decoded instruction at that address as an object inheriting from the `Instruction` (and stored within a `std::unique_ptr`).

3. The `Instruction`-derrived object can then be passed to `executeInstruction` where it is finally executed.

This provides a fair degree of flexibility - the separation of decoding and executing gives the freedom to

decode an instruction at a given address (so that its assembly representation may be seen, for example) without then being forced to execute it.

At this stage in development, the `Intel8086` class is only declared in a header and is not yet implemented.

7.1.5 Register Indexes

You may have noticed that the CPU class outlined in section 7.1.4 has register members which are to be declared now. However, before the class templates for the registers themselves are defined, I begin with creating the `RegisterIndex` class:

```
#pragma once

#include <string>

namespace emu::cpu {
    enum RegisterPart { FULL_WORD, LOW_BYTE, HIGH_BYTE };

    /**
     * Class used to indicate which register is required within a collect of registers. Should be used somewhat like a
     * Java-style enum by extending this call and then having each enum item as a static const instance of that same
     * class.
     */
    class RegisterIndex {
    protected:
        RegisterIndex(std::string indexName, std::string indexDescription = "");

    public:
        /**
         * Converts this register index to its name in x86 assembly. Since Intel syntax is used the short-hand name
         * is simply returned (e.g. 'ax').
         *
         * @return Assembly representation of this register index.
         */
        std::string toAssembly() const;

        /**
         * Builds a string displaying information about this register index.
         *
         * @return Informational string.
         */
        std::string getInfo() const;

    private:
        const std::string name, description;
    };

    /**
     * Used in a fashion similar to RegisterIndex but with the added option to have separate names depending on whether
     * the entire register is accessed or just its high or low bytes.
     */
    class RegisterIndexLowHigh : public RegisterIndex {
    protected:
        RegisterIndexLowHigh(std::string name, std::string description = "");
        RegisterIndexLowHigh(std::string name, std::string low, std::string high, std::string description = "");

    public:
        std::string toAssembly(RegisterPart part = FULL_WORD) const;

    private:
        const std::string lowName, highName;
    };

    class GeneralIndex : public RegisterIndexLowHigh {
    public:     static const GeneralIndex AX, BX, CX, DX;
    protected:  using RegisterIndexLowHigh::RegisterIndexLowHigh;
    };

    class SegmentIndex : public RegisterIndex {
    public:     static const SegmentIndex CODE, DATA, EXTRA, STACK;
    protected:  using RegisterIndex::RegisterIndex;
    };
}
```

This `RegisterIndexes` class is to function somewhat like the type of `enum` one can define in Java - in other words, an enumeration but with additional properties also stored. In this instance, those 'additional properties' are the assembly identifiers and brief descriptions of each register. This class functions like a enumeration by having its constructor be protected and then each enumeration value defined as a static constant member of that class.

In the above header, indexes for each of general-purpose and segment registers of the Intel 8086 are

declared.

The implementation of this class is fairly simple also:

```
#include "common/emu/cpu/registerindexes.hpp"

namespace emu::cpu {
    RegisterIndex::RegisterIndex(std::string indexName, std::string indexDescription)
    : name(indexName), description(indexDescription) {}

    std::string RegisterIndex::toAssembly() const {
        return name;
    }

    std::string RegisterIndex::getInfo() const {
        if(description.size()) return name + " - " + description + " Register";
        else return name + " - Register";
    }

    RegisterIndexLowHigh::RegisterIndexLowHigh(std::string name, std::string description)
    : RegisterIndex(name, description), lowName(name + "(low)"), highName(name + "(high)") {}

    RegisterIndexLowHigh::RegisterIndexLowHigh(std::string name, std::string low, std::string high, std::string description)
    : RegisterIndex(name, description), lowName(low), highName(high) {}

    std::string RegisterIndexLowHigh::toAssembly(RegisterPart part) const {
        switch(part) {
        case LOW_BYTE: return lowName;
        case HIGH_BYTE: return highName;
        default: return toAssembly();
        }
    }

    const GeneralIndex GeneralIndex::AX("AX", "AL", "AH"),
                       GeneralIndex::BX("BX", "BL", "BH"),
                       GeneralIndex::CX("CX", "CL", "CH"),
                       GeneralIndex::DX("DX", "DL", "DH");

    const SegmentIndex SegmentIndex::CODE("CS"),
                       SegmentIndex::DATA("DS"),
                       SegmentIndex::EXTRA("ES"),
                       SegmentIndex::STACK("SS");
}
```

7.1.6 Registers

Registers are also represented using class templates so that the aforementioned `RegisterIndex` may be specified as type arguments - the size of an individual register can also be specified by providing the appropriate type.

```
#pragma once

#include <map>
#include "common/primitives.hpp"
#include "common/conversion.hpp"
#include "common/emu/cpu/registerindexes.hpp"

namespace emu::cpu {
    /**
     * Generic class template for a collection of registers.
     *
     * @tparam Index Type of indexes for specifying the desired register (should be a RegisterIndex index).
     * @tparam Value Type of value stored in each register (usually numerical).
     */
    template <typename Index, typename Value>
    class Registers {
    public:
        /// Register value getter.
        Value get(Index index) { return regs[index]; }

        /// Register value setter.
        void set(Index index, Value value) { regs[index] = value; }

    private:
        std::map<Index, Value> regs;
    };

    template <typename Index>
    class RegistersLowHigh : public Registers<Index, u16> {
    public:
        /// Get least significant byte of 16-bit register.
        u8 getLow(Index index) {
            u16 value = get(index);
            return conversion::getLowByte(value);
        }

        /// Get most significant byte of 16-bit register.
        u8 getHigh(Index index) {
            u16 value = get(index);
            return conversion::getHighByte(value);
```

```
        }

        /**
         * Fetch a specific part of a register. Note that return value will always be 16-bit wide even if only a single
         * byte of a register is accessed.
         */
        u16 get(Index index, RegisterPart part) {
            switch(part) {
            case LOW_BYTE: return getLow(index);
            case HIGH_BYTE: return getHigh(index);
            default: return get(index);
            }
        }

        void setLow(Index index, u8 value) {}
        void setHigh(Index index, u8 value) {}
        void set(Index index, RegisterPart part, u16 value) {}
    };
}
```

Internally, registers are stored using `std::map` which maps a `RegisterIndex` to a register's value. The `RegistersLowHigh` class makes use of helper functions `getHighByte` and `getLowByte` which are outlined in section 7.1.7.

7.1.7 Helper Functions

Due to the Intel 8086 having general-purpose registers that can be accessed as individual high or low bytes, it is necessary that the program can extract the most or least significant byte from a 16-bit value. This is what prompted me to create a helper function namespace called `conversion`.

```
#pragma once

#include "common/primitives.hpp"

namespace conversion {
    /**
     * Returns the most significant byte of the given value.
     *
     * @param value Value to fetch high byte of.
     * @return Most significant byte of value.
     */
    u8 getHighByte(u16 value);

    /**
     * Returns the least significant byte of the given value.
     *
     * @param value Value to fetch the low byte of.
     * @return Least significant byte of value.
     */
    u8 getLowByte(u16 value);
}
```

While it currently has only two functions, it is likely that this namespace will become more populated as the project progresses.

```
#include "common/conversion.hpp"

namespace conversion {
    u8 getHighByte(u16 value) {
        return value >> 8;
    }

    u8 getLowByte(u16 value) {
        return value & 0xFF;
    }
}
```

7.2 Testing

7.2.1 Testing of Memory

The `Memory` has been fully implemented and is as such ready for proper testing. Below is the unit testing code for this class (using the Catch2 framework):

```
TEST_CASE("Test emulator memory.", "[emu][memory]") {
    // Define some type aliases for convenience:
    using Address = u32;
    using Value = u16;
    using Memory = emu::Memory<Value, Address>;

    Memory memory(0xF); // Create instance of memory class which is used by all tests below.

    SECTION("Ensure accurate checking of whether an address is out of bounds.") {
        REQUIRE(memory.withinBounds(0)); // normal
        REQUIRE(memory.withinBounds(memory.size - 1)); // boundary
        REQUIRE_FALSE(memory.withinBounds(memory.size)); // erroneous
    }

    SECTION("Test read/writing of memory.") {
        constexpr Value value = 123;

        // Normal and boundary (as every memory address is written to and read from):
        for(Address addr = 0; addr < memory.size; addr++) {
            memory.write(addr, value);
            REQUIRE(memory.read(addr) == value);
        }

        // Erroneous read/write (exception should be thrown):
        REQUIRE_THROWS_AS(memory.write(memory.size, value), Memory::OutOfBounds);
        REQUIRE_THROWS_AS(memory.read(memory.size), Memory::OutOfBounds);
    }

    SECTION("Test reading/writing of multiple values from/to memory.") {
        std::vector<Value> values = { 12, 34, 56 };

        // Boundary read/write:
        Address addr = memory.size - values.size() - 1;
        memory.write(addr, values);
        REQUIRE(memory.read(addr, values.size()) == values);

        // Erroneous (exception should be thrown):
        REQUIRE_THROWS_AS(memory.write(memory.size - 2, values), Memory::OutOfBounds);
        REQUIRE_THROWS_AS(memory.read(memory.size - 3, 5), Memory::OutOfBounds);
    }

    SECTION("Test filling of all memory.") {
        constexpr Value value = 456;

        memory.fill(value);

        for(Address addr = 0; addr < memory.size; addr++) {
            REQUIRE(memory.read(addr) == value);
        }
    }
}
```

The results of running these tests:

```
 max@virtual  ~/Wired86/emulator/build  master ●  ./test "[memory]" -d yes
0.000 s: Ensure accurate checking of whether an address is out of bounds.
0.000 s: Test emulator memory.
0.000 s: Test read/writing of memory.
0.000 s: Test emulator memory.
0.000 s: Test reading/writing of multiple values from/to memory.
0.000 s: Test emulator memory.
0.000 s: Test filling of all memory.
0.000 s: Test emulator memory.
===============================================================================
All tests passed (38 assertions in 1 test case)
```

7.2.2 Testing of Helper Functions

The two functions of the `conversion` namespace were also tested.

```
TEST_CASE("Tests conversions.", "[conversions]") {
    using namespace conversion;

    SECTION("Test the fetching of high and low bytes from 16-bit values.") {
        REQUIRE(getHighByte(0) == 0);
        REQUIRE(getHighByte(0xAB) == 0);
        REQUIRE(getHighByte(0xABCD) == 0xAB);

        REQUIRE(getLowByte(0) == 0);
        REQUIRE(getLowByte(0xAB) == 0xAB);
        REQUIRE(getLowByte(0xABCD) == 0xCD);
    }
}
```

These tests also all ran successfully.

7.2.3 Testing of CPU Registers

While the CPU itself is not yet ready to be tested, the CPU register system is complete and therefore ready.

```
TEST_CASE("Test CPU registers.", "[emu][cpu][registers]") {
    enum Index { REG };
    emu::cpu::Registers<Index, u32> regs;

    SECTION("Ensure registers are initialised to 0.") {
        REQUIRE(regs.get(REG) == 0);
    }

    SECTION("Test setting/getting register values.") {
        regs.set(REG, 0xBED);
        REQUIRE(regs.get(REG) == 0xBED);
    }

    emu::cpu::RegistersLowHigh<Index> regsLowHigh;

    SECTION("Test individual access of high/low bytes of registers.") {
        regsLowHigh.setLow(REG, 0xA);
        regsLowHigh.setHigh(REG, 0xB);

        REQUIRE(regsLowHigh.getLow(REG) == 0xA);
        REQUIRE(regsLowHigh.get(REG, emu::cpu::LOW_BYTE) == 0xA);

        REQUIRE(regsLowHigh.getHigh(REG) == 0xB);
        REQUIRE(regsLowHigh.get(REG, emu::cpu::HIGH_BYTE) == 0xB);

        REQUIRE(regsLowHigh.get(REG) == 0x0B0A);
        REQUIRE(regsLowHigh.get(REG, emu::cpu::FULL_WORD) == 0x0B0A);
    }
}
```

Running these tests shows that the CPU register system devised functions as expected.

7.3 Review

7.3.1 Overview of Progress

After completion of the first stage of development, I now have a working development environment and a codebase that, though it may not yet provide any use to the user, is a solid foundation on which to build. In terms of that foundation, a proper build system with distinctly separate locations for code (common, GUI, CLI, and testing) is provided. For 'common' code, several key emulator components - including the CPU, memory and registers - are now implemented (or at least declared).

7.3.2 Success Criteria

This first stage of development was able to fully address one emulator success criteria (see table 1): *Has emulated memory which can be read and written to by the emulated CPU without error.*

For evidence that this success criteria was indeed met, please see section 7.2.1. In said section, one can see that all unit tests designed to ensure emulator memory functions correctly run successfully.

7.3.3 Plans

There are a few key areas of software that should be addressed next. To begin with, I believe that the software would benefit greatly from some form of proper logging system as currently output is just

written to standard out without any form of colour coding, time-stamping, or other such information. In addition, the `Intel8086` class needs to be implemented. After which I can begin the difficult task of devising a system which can decode the complex x86 instruction set.

8 Implementation: Second Stage

8.1 Development

8.1.1 Logging

To begin the second stage of development, it was decide that a proper, flexible logging system was integral to for clearly providing information regarding the status of emulator (at least, before the GUI is implemented). I decided that an object-orientated approach would be most appropriate as multiple instance of a `Logger` class can be created for each logger type (e.g. general information, error, warning, etc.)

Aside from the colour-coding and time-stamping one would expect from such a system, additional information can be provided to logging output via the `ADDITIONAL_LOGGING_INFO` macro and the `LoggingInfo` structure. Through the use of the former as the final argument to a logging call, I can have that logging message include information about from where in the code it was called from (line number, file, and from what function).

The logging system is also not hard-coded to output via standard console output. Instead, it makes use of the C++ standard library's `std::ostream` object to allow output via any stream (console output, to a file, or even over a network). This is beneficial as it allows logging output to be redirected to a file for later analysis with very little extra effort.

```
1   #pragma once
2
3   #include <string>
4   #include <ostream>
5   #include <vector>
6   #include <iostream>
7
8   /*
9    * __PRETTY_FUNCTION__ produces the function name along with argument types and namespaces though it is unfortunately
10   * only available on GCC. If GCC is not being used then __func__ is used instead (which is defined as part of the C++ 11
11   * standard).
12   */
13  #ifdef __GNUG__
14      #define ADDITIONAL_LOGGING_INFO { __LINE__, __FILE__, __PRETTY_FUNCTION__ }
15  #else
16      #define ADDITIONAL_LOGGING_INFO { __LINE__, __FILE__, __func__ }
17  #endif
18
19  namespace logging {
20      /**
21       * Structure holding information about a call to a logging method. Holds the line number, file and function name
22       * from which a log call is made. Should not be made manually - use the ADDITIONAL_LOGGING_INFO macro instead as a
23       * second argument to a logging method in Logger.
24       */
25      struct LoggingInfo {
26          unsigned long line;
27          std::string file;
28          std::string function;
29      };
30
31      /**
32       * Allows for the logging of information, warning and errors to multiple output streams.
33       */
34      class Logger {
35      public:
36          /**
37           * Initialise a new logger object.
38           *
39           * @param loggerLogType A string specifying the type of log message. This is prefixed to and displayed before every log message.
40           * @param loggerEscapeSequence Specify the escape sequence applied to each log message (usually a colour - see static constant expressions of Logger
41                     class).
42           * @param initialStream The first stream with which
43           */
44          Logger(std::string loggerLogType, std::string loggerEscapeSequence, std::ostream& initialStream = std::cout);
45
46          /**
47           * Display message via all output streams.
48           *
49           * @param message Message to display (should not contain timestamps or logging type as that information will be
50           *                added automatically).
51           */
52          void operator()(std::string message);
53
54          /**
55           * Display message with additional information about where the call was made to all output streams.
56           *
57           * @param info Additional information about where the call to log was made. Should not be created manually -
```

```
         *            instead use the ADDITIONAL_LOGGING_INFO macro.
         */
        void operator()(std::string message, LoggingInfo info);

        /// Add a new output stream to this logger.
        void addStream(std::ostream& stream);
        /// Remove an output stream from this logger.
        void removeStream(std::ostream& stream);
        /// Check if this logger has at least 1 output stream.
        bool hasStreams() const;

        static constexpr auto MESSAGE_END = "\033[0m\n"; // Reset formatting/colouring plus newline.

        static constexpr auto CYAN_ON_BLACK_TEXT    = "\033[36;40m",
                              WHITE_ON_BLACK_TEXT   = "\033[37;40m",
                              YELLOW_ON_BLACK_TEXT  = "\033[33;40m",
                              RED_ON_BLACK_TEXT     = "\033[31;40m";

    protected:
        /// Output a string through every output stream used by this logger.
        void outThroughAllStreams(std::string msg);

        /// Returns a string of the current time expressed in HH:MM:SS format.
        std::string fetchCurrentTimeString() const;

    private:
        std::vector<std::ostream*> streams;
        const std::string logType, escapeSequence;
    };

    extern Logger info, /// For logging general information.
                  success, /// For logging information indicating an operation completed successfully.
                  warning, /// For logging warning messages (non-fatal).
                  error; /// For logging errors (fatal).
}
```

```
#include "logging.hpp"

#include <iomanip>
#include <ctime>
#include <sstream>
#include <algorithm>

namespace logging {
    // Define the 4 standard logger types:
    Logger info("INFO", Logger::WHITE_ON_BLACK_TEXT);
    Logger success("SUCCESS", Logger::CYAN_ON_BLACK_TEXT);
    Logger warning("WARNING", Logger::YELLOW_ON_BLACK_TEXT, std::cerr);
    Logger error("ERROR", Logger::RED_ON_BLACK_TEXT, std::cerr);

    Logger::Logger(std::string loggerLogType, std::string loggerEscapeSequence, std::ostream& initialStream)
    : logType(loggerLogType), escapeSequence(loggerEscapeSequence) {
        addStream(initialStream); // Add the logger's first output stream (defaults to standard out).
    }

    void Logger::operator()(std::string message) {
        outThroughAllStreams(escapeSequence +
                             "[" + logType + " - " + fetchCurrentTimeString() + "] " +
                             message + MESSAGE_END);
    }

    void Logger::operator()(std::string message, LoggingInfo info) {
        outThroughAllStreams(escapeSequence +
                             "[" + logType + " - " + fetchCurrentTimeString() +
                             " - line " + std::to_string(info.line) +
                             " of " + info.file + " - " + info.function + "] " +
                             message + MESSAGE_END);
    }

    void Logger::addStream(std::ostream& stream) {
        streams.push_back(&stream);
    }

    void Logger::removeStream(std::ostream& stream) {
        auto pos = std::remove(streams.begin(), streams.end(), &stream);
        streams.erase(pos, streams.end());
    }

    bool Logger::hasStreams() const {
        return streams.size() > 0;
    }

    void Logger::outThroughAllStreams(std::string msg) {
        if(hasStreams()) {
            for(std::ostream* stream : streams) {
                // Since streams are stored as raw pointers here, it is best to check that each pointer is not null before outputting.
                if(stream) (*stream) << msg;
            }
        }
    }

    std::string Logger::fetchCurrentTimeString() const {
        // std::time is an old function provided by the C standard library that also included in the C++ STD.
        std::time_t t = std::time(nullptr);
        // Convert the epoch time (seconds since 00:00:00 UTC January 1st 1970) into a calendar time expressed in local time:
        std::tm tm = *std::localtime(&t);

        // Convert the structure returned by std::localtime to a string in format `hours:minutes:seconds`:
        std::stringstream ss;
        ss << std::put_time(&tm, "%H:%M:%S");

        return ss.str(); // Convert std::stringstream to regular std::string.
    }
```

```
68 }
```

8.1.2 Instruction Opcodes

Finally, it was time to begin the complex task of creating a system capable of representing the complex instruction encoding used by the Intel 8086 CPU. The first step here was to create a class to represent instruction opcodes.

```
1  #pragma once
2
3  #include "primitives.hpp"
4
5  namespace emu::cpu::instr {
6      enum DataSize { WORD_DATA_SIZE, BYTE_DATA_SIZE };
7      enum RegDirection { REG_IS_SOURCE, REG_IS_DESTINATION };
8
9      class Opcode {
10     public:
11         Opcode(u8 opcodeValue);
12
13         /**
14          * Indicates whether the data size of this opcode is a word (when the w-bit is 1/true) or a byte (when the w-bit
15          * is 0/false). This bit is the least significant bit of the opcode.
16          */
17         bool getWordBit() const;
18         DataSize getDataSize() const;
19
20         /**
21          * Indicates whether the REG component of a MOD-REG-R/M byte is the source or destination for data handled by
22          * the instruction. This bit is the second-to-least significant bit of the opcode.
23          */
24         bool getDirectionBit() const;
25         RegDirection getDirection() const;
26
27         const u8 value;
28     };
29 }
```

Aside from of course indicating which operation an instruction will carry out, the opcode of an instruction also indicates (where applicable) whether the instruction operates on byte values or 16-bit values, as well as the 'direction' of the instruction (whether the REG component of the instruction's MOD-REG-R/M byte is acting as a source or destination for the operation). In the `Opcode` class header above, it can be seen that these are represented using human-readable enumerations `DataSize` and `RegDirection` as an alternative to the bit values they are actually stored as.

```
1  #include "emu/cpu/instr/opcode.hpp"
2
3  #include "convert.hpp"
4
5  namespace emu::cpu::instr {
6      Opcode::Opcode(u8 opcodeValue) : value(opcodeValue) {}
7
8      bool Opcode::getWordBit() const {
9          return convert::getBitFrom(value, 0); // Least significant bit.
10     }
11
12     DataSize Opcode::getDataSize() const {
13         return getWordBit() ? WORD_DATA_SIZE // w=1
14                             : BYTE_DATA_SIZE; // w=0
15     }
16
17     bool Opcode::getDirectionBit() const {
18         return convert::getBitFrom(value, 1); // Second-to-least significant bit.
19     }
20
21     RegDirection Opcode::getDirection() const {
22         return getDirectionBit() ? REG_IS_DESTINATION // d=1
23                                  : REG_IS_SOURCE; // d=0
24     }
25 }
```

8.1.3 Changes to CPU

In the first stage of development, a class by the name of `Intel8086` was declared. At this second stage, that class is partially implemented:

```cpp
#include "emu/cpu/intel8086.hpp"

namespace emu::cpu {
    AbsAddr Intel8086::resolveAddress(OffsetAddr offset, SegmentIndex segment) const {
        OffsetAddr segmentAddress = segmentRegisters.get(segment);

        // Calculate 20-bit absolute address by applying 4-bit left shift to segment address and then adding the offset:
        return (segmentRegisters << 4) + offset;
    }

    AbsAddr Intel8086::nextInstructionAddress() const {
        // The next instruction for execution will be at the instruction pointer segmented within the code segment:
        return resolveAddress(instructionPointer, SegmentIndex::CODE);
    }

    std::unique_ptr<Instruction> Intel8086::fetchDecodeInstruction(AbsAddr address, const Memory<MemValue, AbsAddr>& memory) const {
        // ...
    }

    void Intel8086::executeInstruction(std::unique_ptr<Instruction>& instruction, const Memory<MemValue, AbsAddr>& memory) {
        // ...
    }
}
```

8.1.4 Conversion/Helper Functions

Aside from the `conversion` namespace being renamed to just `convert`, a few additional helper functions and function templates were introduced into said namespace. For example, in section 8.1.2 you will see the use of templates by the name of `getBitFrom` and `getBitsFrom` respectively. As their names would imply, these template functions allow for fetching specific bits contained in a value from a given 'index'.

```cpp
#pragma once

#include "primitives.hpp"

namespace convert {
    /**
     * Returns the most significant byte of the given value.
     *
     * @param value Value to fetch high byte of.
     * @return Most significant byte of value.
     */
    u8 getHighByte(u16 value);

    /**
     * Returns the least significant byte of the given value.
     *
     * @param value Value to fetch the low byte of.
     * @return Least significant byte of value.
     */
    u8 getLowByte(u16 value);

    /**
     * Create a 16-bit word from a low and a high byte.
     *
     * @param low The least significant byte.
     * @param high The most significant byte.
     * @return 16-bit word constructed from the low/high bytes.
     */
    u16 createWordFromBytes(u8 low, u8 high);

    /**
     * Fetch a specific bit of a numerical value (expressed as a boolean).
     *
     * @tparam T Numerical type to fetch bit from.
     * @param value Value to fetch bit of.
     * @param index Specify which bit to fetch. Indexing begins from 0 with the least significant bit up to the most
     *              significant.
     * @return The fetched bit expressed as a boolean value.
     */
    template <typename T>
    bool getBitFrom(T value, unsigned int index) {
        return (value >> index) & 1;
    }

    /**
     * Fetch multiple bits from a numerical value.
     *
     * @tparam T Numerical type to fetch bits from.
     * @param value Value to fetch bits from.
     * @param index Specify the beginning of the sequence of bits to fetch (indexing beginning from 0).
     * @param count The number of bits to fetch starting from the specified index.
     * @return The fetched sequence of bits.
     */
    template <typename T>
    T getBitsFrom(T value, unsigned int index, unsigned int count) {
        T mask = (1 << count) - 1; // (2 ^ count) - 1
        return (value >> index) & mask;
    }
}
```

In addition, the function `createWordFromBytes` was also introduced. This function simply combines a low and high byte in order to produce a resulting 16-bit value.

```
#include "convert.hpp"

namespace convert {
    u8 getHighByte(u16 value) {
        return (value >> 8) & 0xFF;
    }

    u8 getLowByte(u16 value) {
        return value & 0xFF;
    }

    u16 createWordFromBytes(u8 low, u8 high) {
        return (high << 8) + low;
    }
}
```

8.2 Testing

8.2.1 Testing of New Helper/Conversion Code

In this second stage of development, additional code was introduced into the `convert` namespace. Naturally, to ensure the stability of the program, said code should be tested.

```
...

    SECTION("Test the creation of 16-bit values from a high/low 8-bit byte.") {
        REQUIRE(createWordFromBytes(0, 0) == 0);
        REQUIRE(createWordFromBytes(0xAB, 0) == 0xAB);
        REQUIRE(createWordFromBytes(0, 0xCD) == 0xCD00);
        REQUIRE(createWordFromBytes(0xAB, 0xCD) == 0xCDAB);
    }

    SECTION("Test function templates for fetching specific bits of numerical values.") {
        REQUIRE(getBitFrom<u8>(0b10110, 4));
        REQUIRE_FALSE(getBitFrom<u8>(0b101, 1));

        REQUIRE(getBitsFrom<u16>(0b101010100, 2, 7) == 0b1010101);
        REQUIRE(getBitsFrom<u8>(0b10111000, 3, 3) == 0b111);
    }
}
```

The above code shows the new unit tests added to the pre-existing section wherein the code of the `covert` namespace is checked. It was by running these new unit tests that I realised the original implementation of the `getBitsFrom` function template did not work as I had intended:

```
/home/max/Wired86/emulator/src/test/testcommon.cpp:28: FAILED:
  REQUIRE( getBitsFrom<u16>(0b101010100, 2, 7) == 0b1010101 )
with expansion:
  0 == 85

===============================================================================
test cases:  1 |  0 passed | 1 failed
assertions: 13 | 12 passed | 1 failed
```

The above image shows the output when the tests were run and one failed. From this I was able to identify the `getBitsFrom` function template as containing the faulty code. Upon investigation, I realised the following line had an off-by-one-error:

```
T mask = 1 << count;
```

This was causing issues as creating an appropriate bit-mask requires raising 2 to the power of the number of bits and then taking away one. As such, the code was easily corrected:

```
T mask = (1 << count) - 1; // (2 ^ count) - 1
```

8.2.2 Testing of Basic Instruction Representation

In terms of CPU instructions, only the `Opcode` class is fully-implemented at this stage in development.

```
1  TEST_CASE("Test CPU instruction representation.", "[emu][cpu][instructions]") {
2      using namespace emu::cpu;
3
4      SECTION("Test checking the direction and data size of instruction based on opcode value") {
5          instr::Opcode firstOpcode(0b10);
6
7          REQUIRE_FALSE(firstOpcode.getWordBit());
8          REQUIRE(firstOpcode.getDataSize() == instr::BYTE_DATA_SIZE);
9
10         REQUIRE(firstOpcode.getDirectionBit());
11         REQUIRE(firstOpcode.getDirection() == instr::REG_IS_DESTINATION);
12
13         instr::Opcode secondOpcode(0b01);
14
15         REQUIRE(secondOpcode.getWordBit());
16         REQUIRE(secondOpcode.getDataSize() == instr::WORD_DATA_SIZE);
17
18         REQUIRE_FALSE(secondOpcode.getDirectionBit());
19         REQUIRE(secondOpcode.getDirection() == instr::REG_IS_SOURCE);
20     }
21 }
```

All these tests ran successfully.

8.3 Review

8.3.1 Overview of Progress

The second stage of development began with the implementation of a more robust logging system which should help with the diagnosis of issues with as well as the general usage of the emulator. In addition the first step towards full instruction decoding was made with the introduction of opcode representation and further improvements to the CPU code.

8.3.2 Success Criteria

While it does not yet meet it, this stage of development brought the project closer to fulfilling the following success criteria (see table 1): *Capable of executing the chosen subset of instructions successfully including cases where those instructions may include MOD-REG-R/M, immediate and/or displacement bytes.*

To elaborate, the introduction of the `Opcode` class is the first key step to meeting the above criteria - said class has been implemented and tested.

8.3.3 Plan

As for the following stage of development, completing the instruction representation system is a key priority. Said system should handle execution of instructions as well as the creation of assembly representations. In addition, reading/writing memory data to/from files needs to be implemented.

9 Implementation: Third Stage

9.1 Development

9.1.1 Type Aliases

To begin the third stage of development, a small collection of type aliases were defined inside their own header for the purpose of improving code readability.

```
#pragma once

#include <memory>
#include "primitives.hpp"
#include "emu/memory.hpp"

namespace emu {
    /// Absolute address on the 8086 are 20-bit however no 20-bit unsigned integer type exists in C++ so a 32-bit
    /// unsigned integer is used instead.
    using AbsAddr = u32;

    /// Values stored in memory are 8-bit wide.
    using MemValue = u8;

    /// Memory type for memory storing 8-bit byte values ad taking 32-bit addresses.
    using Mem = Memory<MemValue, AbsAddr>;

    /// Offset addresses within a given segment are 16-bit wide.
    using OffsetAddr = u16;

    /// Type of values stored by standard registers.
    using RegSize = u16;
}
```

9.1.2 System Memory to/from Files

Being able to save and load the state of the emulated system memory is vital for allowing the user to continue working with a specific program over multiple sessions. To allow this, I introduced to new methods to the `Memory` class that respectively allow reading and writing of all memory data to a raw binary file of a given path.

```
...

        /**
         * Load data from a binary file into emulator memory.
         *
         * @param path The path of the file from which data should be loaded.
         * @param offset The offset in memory to which data should be loaded.
         * @return Whether the file could be opened successfully or not.
         */
        bool loadFromFile(std::string path, Address offset = 0) {
            std::ifstream file;

            file.open(path,
                      std::ios::in | // Input/read mode.
                      std::ios::binary | // Binary mode.
                      std::ios::ate); // Place cursor at end of file so that `tellg` will return file size.

            if(file.is_open() && size > offset) {
                Address fileSize = file.tellg();
                Address readSize = std::min(fileSize, size - offset); // Either the entire file will be read or a
                                                                      // number of bytes equal to the size of memory
                                                                      // available will be read (whichever is smaller).

                if(readSize > 0) {
                    file.seekg(0, std::ios::beg);
                    auto ptr = reinterpret_cast<char*>(mem.get() + offset);
                    file.read(ptr, readSize);
                }

                file.close();
                return true;
            }

            return false;
        }

        /**
         * Save all data in emulator memory to a binary file.
         *
         * @param path The path of the file to write to. Note that if a file already exists at this path, it will be
         *        overwritten.
```

```
42          * @return Whether the file could be opened successfully or not.
43          */
44         bool saveToFile(std::string path) const {
45             std::ofstream file;
46
47             file.open(path,
48                       std::ios::out | // Output/write mode.
49                       std::ios::binary | // Binary mode.
50                       std::ios::trunc); // Overwrite existing file contents should it already exist.
51
52             if(file.is_open()) {
53                 auto ptr = reinterpret_cast<char*>(mem.get()); // Cast pointer to the raw char pointer the `write`
54                                                                // method expects.
55                 file.write(ptr, size);
56
57                 file.close();
58                 return true;
59             }
60
61             return false;
62         }
63
64         const Address size;
65
66 ...
```

As mentioned previously, the `Memory` class is actually a class template so the above methods are declared and implemented directly in the header file.

9.1.3 CPU Register System

The existing register system, while functional, has several flaws. One such flaw being the clunky system with which register indexes are defined. Having worked with the language previously, I had come to like Java's interpretation of enumerations (in effect, treating them more like classes) as it allowed each value in an enumeration to have its own sub-properties and even methods. While my attempt to translate such a system to C++ did work, it was very awkward to use and came with drawbacks regarding efficiency and the ability to compare enumeration values. Ultimately, I have now decided that using regular C++ enumerations is the better option.

```
1  #pragma once
2
3  #include <map>
4  #include "primitives.hpp"
5  #include "convert.hpp"
6
7  namespace emu::cpu::reg {
8      enum RegisterPart { FULL_WORD, LOW_BYTE, HIGH_BYTE };
9
10     /**
11      * Generic class template for a collection of registers.
12      *
13      * @tparam Index Type of indexes for specifying the desired register (should be a RegisterIndex index).
14      * @tparam Value Type of value stored in each register (usually numerical).
15      */
16     template <typename Index, typename Value>
17     class Registers {
18     public:
19         /**
20          * Register value getter.
21          *
22          * @param index The index of the register to get.
23          * @return The value stored at the specified register.
24          */
25         Value get(Index index) const {
26             // Cannot have method declared `const` when using the operator[] as that method of indexing will create a
27             // default constructed value if one is not found at the index (thus modifying `this`). As such, the map will
28             // be searched for the index and will return a default value should a value not be found.
29             auto value = regs.find(index);
30
31             if(value != regs.end()) return value->second;
32             else return Value(); // Return default-constructed value if no actual value is found in map.
33         }
34
35         /**
36          * Register value setter.
37          *
38          * @param index The index of the register to set.
39          * @param value The value to set the specified register to.
40          */
41         void set(Index index, Value value) { regs[index] = value; }
42
43         /**
44          * Get the assembly identifier of the specified register. Is pure virtual and must be overriden by subclasses.
45          *
46          * @param index The index of the register assembly identifier to fetch.
47          * @return String assembly identifier.
48          */
```

```
        virtual std::string getAssemblyIdentifier(Index index) const = 0;

    private:
        std::map<Index, Value> regs;
    };

    template <typename Index>
    class RegistersLowHigh : public Registers<Index, u16> {
    public:
        using Registers<Index, u16>::get;
        using Registers<Index, u16>::set;

        /**
         * Get least significant byte of 16-bit register.
         */
        u8 getLow(Index index) const {
            u16 value = get(index);
            return convert::getLeastSigByte(value);
        }

        /**
         * Get most significant byte of 16-bit register.
         */
        u8 getHigh(Index index) const {
            u16 value = get(index);
            return convert::getMostSigByte(value);
        }

        /**
         * Fetch a specific part of a register. Note that return value will always be 16-bit wide even if only a single
         * byte of a register is accessed.
         */
        u16 get(Index index, RegisterPart part) const {
            switch(part) {
            case LOW_BYTE: return getLow(index);
            case HIGH_BYTE: return getHigh(index);
            default: return get(index);
            }
        }

        /**
         * Set least significant byte of 16-bit register (most significant byte unaffected).
         */
        void setLow(Index index, u8 low) {
            u16 high = getHigh(index);
            u16 value = convert::createWordFromBytes(low, high);
            set(index, value);
        }

        /**
         * Set most significant byte of 16-bit register (least significant byte unaffected).
         */
        void setHigh(Index index, u8 high) {
            u16 low = getLow(index);
            u16 value = convert::createWordFromBytes(low, high);
            set(index, value);
        }

        /**
         * Set a specific part of a register. Note that the value argument is 16-bit wide but will be cast to 8-bits
         * when only a setting a high or low byte of a register and not the entire 16-bit value.
         */
        void set(Index index, RegisterPart part, u16 value) {
            u8 byte = static_cast<u8>(value);

            switch(part) {
            case LOW_BYTE: setLow(index, byte); break;
            case HIGH_BYTE: setHigh(index, byte); break;
            default: set(index, value);
            }
        }

        /**
         * Get the assembly identifier of a specific register index and part.
         *
         * @param index The index of the register assembly identifier to fetch.
         * @param part The specific register part to fetch the identifier for.
         * @return String assembly identifier.
         */
        virtual std::string getAssemblyIdentifier(Index index, RegisterPart part) const = 0;
    };
}
```

With this generic system created, I am now able to create the specific set of registers that make up the Intel 8086:

```
#pragma once

#include "primitives.hpp"
#include "emu/cpu/reg/registers.hpp"

namespace emu::cpu::reg {
    enum GeneralRegister {
        AX_REGISTER,
        BX_REGISTER,
        CX_REGISTER,
        DX_REGISTER,
        SOURCE_INDEX,
        DESTINATION_INDEX,
        BASE_POINTER,
```

```
        STACK_POINTER
    };

    class GeneralRegisters : public RegistersLowHigh<GeneralRegister> {
    public:
        std::string getAssemblyIdentifier(GeneralRegister index) const override final;
        std::string getAssemblyIdentifier(GeneralRegister index, RegisterPart part) const override final;
    };

    enum SegmentRegister {
        CODE_SEGMENT,
        DATA_SEGMENT,
        EXTRA_SEGMENT,
        STACK_SEGMENT
    };

    class SegmentRegisters : public Registers<SegmentRegister, u16> {
    public:
        std::string getAssemblyIdentifier(SegmentRegister index) const override final;
    };

    enum Flag {
        CARRY_FLAG,
        PARITY_FLAG,
        AUX_CARRY_FLAG,
        ZERO_FLAG,
        SIGN_FLAG,
        TRAP_FLAG,
        INTERRUPT_FLAG,
        DIRECTION_FLAG,
        OVERFLOW_FLAG
    };

    class Flags : public Registers<Flag, bool> {
    public:
        std::string getAssemblyIdentifier(Flag) const override final;
    };
}
```

```
#include "emu/cpu/reg/registers8086.hpp"

#include "logging.hpp"

namespace emu::cpu::reg {
    constexpr auto UNKNOWN_INDEX = "<unknown register index>";

    std::string GeneralRegisters::getAssemblyIdentifier(GeneralRegister index) const {
        switch(index) {
        case AX_REGISTER: return "ax";
        case BX_REGISTER: return "bx";
        case CX_REGISTER: return "cx";
        case DX_REGISTER: return "dx";
        case SOURCE_INDEX: return "si";
        case DESTINATION_INDEX: return "di";
        case BASE_POINTER: return "bp";
        case STACK_POINTER: return "sp";
        }

        return UNKNOWN_INDEX;
    }

    std::string GeneralRegisters::getAssemblyIdentifier(GeneralRegister index, RegisterPart part) const {
        switch(part) {
        case LOW_BYTE:
            switch(index) {
            case AX_REGISTER: return "al";
            case BX_REGISTER: return "bl";
            case CX_REGISTER: return "cl";
            case DX_REGISTER: return "dl";
            default: break;
            }
            break;

        case HIGH_BYTE:
            switch(index) {
            case AX_REGISTER: return "ah";
            case BX_REGISTER: return "bh";
            case CX_REGISTER: return "ch";
            case DX_REGISTER: return "dh";
            default: break;
            }
            break;

        default: break;
        }

        return getAssemblyIdentifier(index);
    }

    std::string SegmentRegisters::getAssemblyIdentifier(SegmentRegister index) const {
        switch(index) {
        case CODE_SEGMENT: return "cs";
        case DATA_SEGMENT: return "ds";
        case EXTRA_SEGMENT: return "es";
        case STACK_SEGMENT: return "ss";
        }

        return UNKNOWN_INDEX;
    }

    std::string Flags::getAssemblyIdentifier(Flag) const {
        return "flags";
    }
```

```
}
```

9.1.4 CPU Header Expanded

It was realised during this stage that the CPU class would require some additional methods and members:

```hpp
#pragma once

#include <memory>
#include "emu/types.hpp"
#include "emu/cpu/instr/instruction.hpp"
#include "emu/cpu/reg/registers8086.hpp"

namespace emu::cpu {
    /**
     * Class representing the main Intel 8086 microprocessor. Handles decoding and execution of instructions fetched
     * from memory. Also holds all CPU registers.
     */
    class Intel8086 {
    public:
        /**
         * Takes a 16-bit memory offset and a 16-bit segment register and returns an absolute 20-bit address (which is
         * stored in an unsigned 32-bit integer since there is no 20-bit integer type available in C++).
         *
         * @param offset The memory offset within the given segment.
         * @param segment Segment register index indicating which segment to resolve the offset within.
         * @return Absolute 20-bit address within memory.
         */
        AbsAddr resolveAddress(OffsetAddr offset, reg::SegmentRegister segment) const;

        /**
         * Returns the relative address of the instruction pointer (i.e. before it is segmented within the code
         * segment).
         */
        OffsetAddr getRelativeInstructionPointer() const;

        /**
         * Calculate the address of the next instruction in memory based on the value of the instruction pointer and the
         * code segment. Does not increment instruction pointer - that is done on execution.
         *
         * @return Address of next instruction.
         */
        AbsAddr getAbsoluteInstructionPointer() const;

        /**
         * Fetches and decodes the next instruction.
         *
         * @param address The absolute address of the instruction to fetch and decode.
         * @param memory Reference to the memory to fetch the instruction data from.
         * @return Decoded instruction object (will be empty if instruction decoding fails).
         */
        std::unique_ptr<instr::Instruction> fetchDecodeInstruction(AbsAddr address, const Mem& memory) const;

        /**
         * Executes a decoded instruction on this CPU.
         *
         * @param instruction Reference to the std::unique_ptr holding the instruction.
         * @param memory Reference to the memory of the emulated system.
         * @return Whether execution was successful or not.
         */
        bool executeInstruction(std::unique_ptr<instr::Instruction>& instruction, Mem& memory);

        /**
         * Push values onto the stack. Stack pointer decremented.
         */
        void pushToStack(MemValue value, Mem& memory);

        /**
         * Pop values from the stack. Stack pointer incremented.
         */
        MemValue popFromStack(const Mem& memory);

        /**
         * Push a 16-bit word value onto the stack.
         */
        void pushWordToStack(u16 value, Mem& memory);

        /**
         * Pop 16-bit word value off the stack.
         */
        u16 popWordFromStack(const Mem& memory);

        /**
         * Sets the instruction pointer without altering any segment addresses.
         */
        void performRelativeJump(OffsetAddr offset);

        reg::GeneralRegisters generalRegisters; /// CPU general-purpose registers.
        reg::SegmentRegisters segmentRegisters; /// CPU segment registers.

        bool halted = false; // Whether the CPU is in a halted state or not.

    private:
        /**
```

```
89          * @param opcode The instruction opcode.
90          * @return The decoded instruction or an empty unique pointer if decoding failed.
91          */
92         std::unique_ptr<instr::Instruction> fetchDecodeWithoutModRegRm(const instr::Opcode& opcode) const;
93
94         /**
95          * @param opcode The instruction opcode.
96          * @param address The absolute address from which the given opcode was fetched.
97          * @return The coded instruction or an empty unique pointer if decoding failed.
98          */
99         std::unique_ptr<instr::Instruction> fetchDecodeWithModRegRm(const instr::Opcode& opcode, AbsAddr address,
100                                                                     const Mem& memory) const;
101
102        /// The instruction pointer is an offset within the code segment that points to the next instruction in memory.
103        OffsetAddr instructionPointer = 0;
104
105        /// CPU flag register. Declared private as not all flags should be directly modifiable by all.
106        reg::Flags flags;
107    };
108}
```

Firstly, you may have noticed that the type aliases that were declared in this class previously have been moved to their own header file (as mentioned in section 9.1.1). This was done as it allows `Instruction` and its derived classes to make use of said type aliases without causing a circular dependency (previously, `Instruction` would have had to include the `Intel8086` class but this would cause issues as the latter already includes the former).

Methods for interacting with the stack as well as for modifying the instruction pointer (i.e. performing jumps) have also been added.

9.1.5 Assembly Style

In order to support different styles and types of assembly language (e.g. Intel vs AT&T syntax), a collection of structures and enumerations are defined which indicate the exact appearance of the generated assembly code:

```
1   #pragma once
2
3   #include <string>
4
5   namespace assembly {
6       enum NumericalRepresentation {
7           HEX_REPRESENTATION,
8           BINARY_REPRESENTATION,
9           DENARY_REPRESENTATION
10      };
11
12      enum NumericalStyle {
13          WITH_PREFIX,
14          WITH_SUFFIX,
15          WITHOUT_SUFFIX_OR_PREFIX
16      };
17
18      struct Style {
19          NumericalRepresentation numericalRepresentation;
20          NumericalStyle numericalStyle;
21
22          std::string argumentSeparator = ", ";
23
24          std::string displacementBegin = "[";
25          std::string displacementEnd = "]";
26          std::string displacementAdd = " + ";
27
28          std::string hexPrefix = "0x";
29          std::string hexSuffix = "h";
30
31          std::string binaryPrefix = "0b";
32          std::string binarySuffix = "b";
33      };
34  }
```

9.1.6 Instruction Opcode Representation

Some changes have been made to opcode representation at this stage of development. For example, a method would produces a string representation of the opcode value with additional chunks of information

is now provided by a `toString` method. In addition, two more methods were also introduced where one returns the 6-bit opcode value (ignoring the direction and word bits), while the other calculates what the size of the immediate value encoded in the instruction (should there be one) should be (either 1 or 2 bytes).

```
#pragma once

#include <string>
#include "emu/types.hpp"
#include "primitives.hpp"

namespace emu::cpu::instr {
    /// Indicated by the W-bit of opcode.
    enum DataSize { WORD_DATA_SIZE, BYTE_DATA_SIZE };

    /// Indicated by the D-bit of opcode.
    enum RegDirection { REG_IS_SOURCE, REG_IS_DESTINATION };

    /**
     * Holder for an instruction opcode value. Has methods for fetching the direction and data size of the opcode based
     * on the opcode's D-bit and W-bit respectively.
     */
    class Opcode {
    public:
        Opcode(u8 opcodeValue);

        /**
         * Produces a string useful for debugging purposes (shows the opcode value in binary as well as the state of
         * its direction and word bits).
         */
        std::string toString() const;

        /**
         * Fetch the 6 unique bits of this opcode (removes the word and direction bits).
         */
        u8 getUniqueValue() const;

        /**
         * Indicates whether the data size of this opcode is a word (when the w-bit is 1/true) or a byte (when the w-bit
         * is 0/false). This bit is the least significant bit of the opcode.
         */
        bool getWordBit() const;
        DataSize getDataSize() const;

        /**
         * Will return the length in bytes that any immediate instruction component should be. This is based of the data
         * size of the instruction such that this function will return 2 for a word data size and 1 for a byte data size.
         *
         * Note that this method will not return the appropriate value in order to read a displacement component from
         * memory as whether a displacement component is one or two bytes is determined by the MOD-REG-R/M addressing
         * mode and not the opcode data size.
         */
        AbsAddr getImmediateReadLength() const;

        /**
         * Indicates whether the REG component of a MOD-REG-R/M byte is the source or destination for data handled by
         * the instruction. This bit is the second-to-least significant bit of the opcode.
         */
        bool getDirectionBit() const;
        RegDirection getDirection() const;

        const u8 value;
    };
}
```

```
#include "emu/cpu/instr/opcode.hpp"

#include "convert.hpp"

namespace emu::cpu::instr {
    Opcode::Opcode(u8 opcodeValue) : value(opcodeValue) {}

    std::string Opcode::toString() const {
        bool d = getDirectionBit(), w = getWordBit();

        return convert::toHexString(value) +
               " (" + convert::toBinaryString<6>(getUniqueValue(), "") + "dw : d=" + convert::bitAsStr(d) +
               ", w=" + convert::bitAsStr(w) + ")";
    }

    u8 Opcode::getUniqueValue() const {
        return convert::getBitsFrom(value, 2, 6);
    }

    bool Opcode::getWordBit() const {
        return convert::getBitFrom(value, 0); // Least significant bit.
    }

    DataSize Opcode::getDataSize() const {
        return getWordBit() ? WORD_DATA_SIZE  // w=1
                            : BYTE_DATA_SIZE; // w=0
    }

    AbsAddr Opcode::getImmediateReadLength() const {
        return getWordBit() ? 2  // Word data size (i.e. read 2 bytes).
                            : 1; // Byte data size (i.e. read 1 byte).
    }

```

```
34      bool Opcode::getDirectionBit() const {
35          return convert::getBitFrom(value, 1); // Second-to-least significant bit.
36      }
37
38      RegDirection Opcode::getDirection() const {
39          return getDirectionBit() ? REG_IS_DESTINATION // d=1
40                                   : REG_IS_SOURCE;     // d=0
41      }
42  }
```

9.1.7 MOD-REG-R/M

The MOD-REG-R/M byte of an instruction is encoded with a fair amount of important data which heavily influences the running of that particular instruction. It was for this reason that a fair amount of time was spent ensuring the `ModRegRm` class is simple to use.

The header below not only declares the class itself, but also some enumerations used to indicate modes of addressing and modes/types of displacement.

```
1   #pragma once
2
3   #include "primitives.hpp"
4   #include "emu/cpu/instr/opcode.hpp"
5   #include "emu/cpu/reg/registers8086.hpp"
6
7   namespace emu::cpu::instr {
8       enum AddressingMode {
9           NO_DISPLACEMENT,
10          BYTE_DISPLACEMENT,
11          WORD_DISPLACEMENT,
12          REGISTER_ADDRESSING_MODE
13      };
14
15      enum DisplacementType {
16          BX_SI_DISPLACEMENT,
17          BX_DI_DISPLACEMENT,
18          BP_SI_DISPLACEMENT,
19          BP_DI_DISPLACEMENT,
20          SI_DISPLACEMENT,
21          DI_DISPLACEMENT,
22          BP_DISPLACEMENT,
23          BX_DISPLACEMENT
24      };
25
26      class ModRegRm {
27      public:
28          ModRegRm(u8 modRegRmValue);
29
30          /**
31           * Fetch the three bits that comprise the R/M component of this MOD-REG-R/M byte.
32           */
33          u8 getRmBits() const;
34
35          /**
36           * Fetch the three bits that make up the REG component of this MOD-REG-R/M byte.
37           */
38          u8 getRegBits() const;
39
40          /**
41           * Fetch the pair of bits that comprise the MOD component of this MOD-REG-R/M byte.
42           */
43          u8 getModBits() const;
44
45          /**
46           * Return the appropriate `AddressingMode` enumeration value based on the MOD component bits.
47           */
48          AddressingMode getAddressingMode() const;
49
50          AbsAddr getDisplacementReadLength() const;
51
52          /**
53           * Get the appropriate register index based on the value of the R/M component.
54           * Only relevant when using register addressing mode.
55           */
56          reg::GeneralRegister getRegisterIndexFromRm(DataSize size) const;
57
58          /**
59           * Returns the appropriate register part based on the value of the R/M component.
60           * Only relevant when using register addressing mode.
61           */
62          reg::RegisterPart getRegisterPartFromRm(DataSize size) const;
63
64          /**
65           * Get the assembly identifier of the register indicated by the R/M component.
66           */
67          std::string getRegisterIdentifierFromRm(const reg::GeneralRegisters& registers, DataSize size) const;
68
69          /**
70           * Get the appropriate register index based on the value of the REG component.
71           */
72          reg::GeneralRegister getRegisterIndexFromReg(DataSize size) const;
```

```
73
74          /**
75           * Returns the appropriate register part based on the value of the REG component.
76           */
77          reg::RegisterPart getRegisterPartFromReg(DataSize size) const;
78
79          /**
80           * Get the assembly identifier of the register indicated by the REG component.
81           */
82          std::string getRegisterIdentifierFromReg(const reg::GeneralRegisters& registers, DataSize size) const;
83
84          /**
85           * Returns the appropriate displacement type based on the value of the R/M component.
86           */
87          DisplacementType getDisplacementType() const;
88
89          /**
90           * Returns true when R/M indicates that either byte displacement or word displacement is to be used. Otherwise,
91           * returns false.
92           */
93          bool isDisplacementUsed() const;
94
95          const u8 value;
96
97      private:
98          /**
99           * Returns the appropriate register index based on 3 bits given and the data size.
100          *
101          * @param The 3 bits of either a REG or R/M component that specify a register.
102          * @param size The data size handled by this instruction (16-bit word or 8-bit byte).
103          * @return A general register index.
104          */
105         reg::GeneralRegister getRegisterIndex(u8 bits, DataSize size) const;
106
107         /**
108          * Returns the appropriate register part (low byte, high byte, or full word) based on the 3 bits given and the
109          * the data size. Will always return full word when given a 16-bit data size option.
110          *
111          * @param The 3 bits of either a REG or R/M component that specify a register.
112          * @param size The data size handled by this instruction (16-bit word or 8-bit byte).
113          * @return A register part.
114          */
115         reg::RegisterPart getRegisterPart(u8 bits, DataSize size) const;
116     };
117 }
```

```
1   #include "emu/cpu/instr/modregrm.hpp"
2
3   #include "convert.hpp"
4   #include "logging.hpp"
5
6   namespace emu::cpu::instr {
7       ModRegRm::ModRegRm(u8 modRegRmValue) : value(modRegRmValue) {}
8
9       u8 ModRegRm::getRmBits() const {
10          return convert::getBitsFrom(value, 0, 3);
11      }
12
13      u8 ModRegRm::getRegBits() const {
14          return convert::getBitsFrom(value, 3, 3);
15      }
16
17      u8 ModRegRm::getModBits() const {
18          return convert::getBitsFrom(value, 6, 2);
19      }
20
21      AddressingMode ModRegRm::getAddressingMode() const {
22          u8 bits = getModBits();
23
24          switch(bits) {
25          case 0b00: return NO_DISPLACEMENT;
26          case 0b01: return BYTE_DISPLACEMENT;
27          case 0b10: return WORD_DISPLACEMENT;
28          case 0b11: return REGISTER_ADDRESSING_MODE;
29          }
30
31          logging::warning("Invalid addressing mode specified by MOD component of MOD-REG-R/M byte: " +
32                           convert::toBinaryString<8>(bits));
33          return NO_DISPLACEMENT;
34      }
35
36      AbsAddr ModRegRm::getDisplacementReadLength() const {
37          switch(getAddressingMode()) {
38          case BYTE_DISPLACEMENT: return 1; // Read 1 byte.
39          case WORD_DISPLACEMENT: return 2; // Read word (i.e. 2 bytes).
40
41          default: return 0; // Neither byte nor word displacement is actually being used.
42          }
43      }
44
45      reg::GeneralRegister ModRegRm::getRegisterIndexFromRm(DataSize size) const {
46          return getRegisterIndex(getRmBits(), size);
47      }
48
49      reg::RegisterPart ModRegRm::getRegisterPartFromRm(DataSize size) const {
50          return getRegisterPart(getRmBits(), size);
51      }
52
53      std::string ModRegRm::getRegisterIdentifierFromRm(const reg::GeneralRegisters& registers, DataSize size) const {
54          return registers.getAssemblyIdentifier(getRegisterIndexFromRm(size),
55                                                 getRegisterPartFromRm(size));
56      }
57
```

```
58      reg::GeneralRegister ModRegRm::getRegisterIndexFromReg(DataSize size) const {
59          return getRegisterIndex(getRegBits(), size);
60      }
61
62      reg::RegisterPart ModRegRm::getRegisterPartFromReg(DataSize size) const {
63          return getRegisterPart(getRegBits(), size);
64      }
65
66      std::string ModRegRm::getRegisterIdentifierFromReg(const reg::GeneralRegisters& registers, DataSize size) const {
67          return registers.getAssemblyIdentifier(getRegisterIndexFromReg(size),
68                                                 getRegisterPartFromReg(size));
69      }
70
71      DisplacementType ModRegRm::getDisplacementType() const {
72          u8 bits = getRmBits();
73
74          switch(bits) {
75          case 0b000: return BX_SI_DISPLACEMENT;
76          case 0b001: return BX_DI_DISPLACEMENT;
77          case 0b010: return BP_SI_DISPLACEMENT;
78          case 0b011: return BP_DI_DISPLACEMENT;
79          case 0b100: return SI_DISPLACEMENT;
80          case 0b101: return DI_DISPLACEMENT;
81          case 0b110: return BP_DISPLACEMENT;
82          case 0b111: return BX_DISPLACEMENT;
83          }
84
85          logging::warning("Invalid displacement type specified by R/M component of MOD-REG-R/M byte: " +
86                           convert::toBinaryString<8>(bits));
87          return BX_SI_DISPLACEMENT;
88      }
89
90      bool ModRegRm::isDisplacementUsed() const {
91          return getAddressingMode() == BYTE_DISPLACEMENT ||
92                 getAddressingMode() == WORD_DISPLACEMENT;
93      }
94
95      reg::GeneralRegister ModRegRm::getRegisterIndex(u8 bits, DataSize size) const {
96          if(size == BYTE_DATA_SIZE) {
97              switch(bits) {
98              case 0b000: // AL
99              case 0b100: // AH
100                 return reg::AX_REGISTER;
101
102             case 0b001: // CL
103             case 0b101: // CH
104                 return reg::CX_REGISTER;
105
106             case 0b010: // DL
107             case 0b110: // DH
108                 return reg::DX_REGISTER;
109
110             case 0b011: // BL
111             case 0b111: // BH
112                 return reg::BX_REGISTER;
113             }
114         }
115
116         if(size == WORD_DATA_SIZE) {
117             switch(bits) {
118             case 0b000: return reg::AX_REGISTER;
119             case 0b001: return reg::CX_REGISTER;
120             case 0b010: return reg::DX_REGISTER;
121             case 0b011: return reg::BX_REGISTER;
122             case 0b100: return reg::STACK_POINTER;
123             case 0b101: return reg::BASE_POINTER;
124             case 0b110: return reg::SOURCE_INDEX;
125             case 0b111: return reg::DESTINATION_INDEX;
126             }
127         }
128
129         logging::warning("Invalid register specified by component of MOD-REG-R/M byte: " +
130                          convert::toBinaryString<8>(bits));
131         return reg::AX_REGISTER;
132     }
133
134     reg::RegisterPart ModRegRm::getRegisterPart(u8 bits, DataSize size) const {
135         if(size == BYTE_DATA_SIZE) {
136             switch(bits) {
137             case 0b000: // AL
138             case 0b001: // CL
139             case 0b010: // DL
140             case 0b011: // BL
141                 return reg::LOW_BYTE;
142
143             case 0b100: // AH
144             case 0b101: // CH
145             case 0b110: // DH
146             case 0b111: // BH
147                 return reg::HIGH_BYTE;
148             }
149         }
150
151         if(size == WORD_DATA_SIZE && bits <= 0b111) // Ensure no more than 3 bits are passed.
152             return reg::FULL_WORD;
153
154         logging::warning("Invalid register part specified by component of MOD-REG-R/M byte: " +
155                          convert::toBinaryString<8>(bits));
156         return reg::FULL_WORD;
157     }
158 }
```

9.1.8 Instruction Arguments (Displacement and Immediate Values)

The displacement and immediate values that an instruction may be encoded with are similar in the sense that they are both simply either 1 or 2 bytes included the instruction. As such, it seemed logical that the classes `Immediate` and `Displacement` would share a common base class. That base class, named `DataArgument`, would handle the storage of the data while its two subclasses would allow that data to function as either an immediate or displacement value respectively.

```
#pragma once

#include <vector>
#include "emu/cpu/instr/modregrm.hpp"
#include "emu/types.hpp"
#include "assembly.hpp"

namespace emu::cpu::instr {
    /**
     * Represents an additional argument to an instruction (either an immediate value or displacement value - not a
     * MOD-REG-R/M component or the opcode).
     */
    class DataArgument {
    public:
        /**
         * Create a data argument containing the specified raw data (little endian format).
         */
        DataArgument(std::vector<u8> raw);

        /**
         * Convert this instruction argument into assembly. This method is pure virtual but overriden in the Immediate
         * and Displacement classes.
         *
         * @param size The data size handled by the instruction this data argument is a part of (usually indicated by
         *        data size bit of instruction opcode).
         * @param modRegRm Constant reference to the instruction's MOD-REG-R/M component.
         */
        virtual std::string toAssembly(DataSize size, const ModRegRm& modRegRm, const reg::GeneralRegisters& registers,
                                       const assembly::Style& style) const = 0;

        /**
         * Return a constant reference to the raw data of this data argument.
         */
        const std::vector<u8>& getRawData() const;

        /**
         * Will return the value of this immediate instruction component as either 8-bits or 16-bits (note the return
         * value will be cast to u16 regardless) based on the data size specified.
         */
        u16 getValueUsingDataSize(DataSize size) const;

        u8 getByteValue() const;
        u16 getWordValue() const;

    private:
        std::vector<u8> rawData;
    };



    class Immediate : public DataArgument {
    public:
        using DataArgument::DataArgument;

        std::string toAssembly(DataSize size, const ModRegRm&, const reg::GeneralRegisters&,
                               const assembly::Style& style) const override final;
    };



    class Displacement : public DataArgument {
    public:
        using DataArgument::DataArgument;

        std::string toAssembly(DataSize, const ModRegRm& modRegRm, const reg::GeneralRegisters& registers,
                               const assembly::Style& style) const override final;

        /**
         * Gives the displacement value of this displacement component based on the given addressing mode (either byte
         * or word displacement).
         */
        u16 getValueUsingAddressingMode(AddressingMode mode) const;

        /**
         * Returns an absolute memory address based on the displacement value and displacement type.
         *
         * @param mode The addressing mode specified by the MOD-REG-R/M byte (note that register addressing mode will
         *        be treated the same as no displacement).
         * @param type The displacement type as specified by the MOD-REG-R/M byte.
         * @param registers The general-purpose and indexing CPU registers.
         * @return The resolved absolute memory address.
         */
        AbsAddr resolve(AddressingMode mode, DisplacementType type, reg::GeneralRegisters& registers) const;
    };
}
```

```cpp
#include "emu/cpu/instr/argument.hpp"

#include "convert.hpp"

namespace emu::cpu::instr {
    /*
     * DataArgument implementation:
     */

    DataArgument::DataArgument(std::vector<u8> raw) : rawData(raw) {
        if(rawData.empty()) rawData.push_back(0u); // Require the vector to have at least 1 element.
        if(rawData.size() > 2) rawData.resize(2); // Ensure vector has no more than 2 elements.
    }

    const std::vector<u8>& DataArgument::getRawData() const {
        return rawData;
    }

    u16 DataArgument::getValueUsingDataSize(DataSize size) const {
        if(size == WORD_DATA_SIZE) return getWordValue();
        else return static_cast<u16>(getByteValue());
    }

    u8 DataArgument::getByteValue() const {
        return rawData[0];
    }

    u16 DataArgument::getWordValue() const {
        u8 low = rawData[0];
        u8 high = rawData.size() > 1 ? rawData[1] : 0; // Get rawData[1] if present. Otherwise, assume high byte is 0.

        return convert::createWordFromBytes(low, high);
    }

    /*
     * Immediate implementation:
     */

    std::string Immediate::toAssembly(DataSize size, const ModRegRm&, const reg::GeneralRegisters&,
                                      const assembly::Style& style) const {
        u16 value = getValueUsingDataSize(size);

        return convert::numberToAssembly(value, style);
    }

    /*
     * Displacement implementation:
     */

    std::string Displacement::toAssembly(DataSize, const ModRegRm& modRegRm,
                                      const reg::GeneralRegisters& registers, const assembly::Style& style) const {
        std::string offsetString;

        switch(modRegRm.getDisplacementType()) {
        case BX_SI_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::BX_REGISTER) + style.displacementAdd +
                            registers.getAssemblyIdentifier(reg::SOURCE_INDEX);
            break;

        case BX_DI_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::BX_REGISTER) + style.displacementAdd +
                            registers.getAssemblyIdentifier(reg::DESTINATION_INDEX);
            break;

        case BP_SI_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::BASE_POINTER) + style.displacementAdd +
                            registers.getAssemblyIdentifier(reg::SOURCE_INDEX);
            break;

        case BP_DI_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::BASE_POINTER) + style.displacementAdd +
                            registers.getAssemblyIdentifier(reg::DESTINATION_INDEX);
            break;

        case SI_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::SOURCE_INDEX);
            break;

        case DI_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::DESTINATION_INDEX);
            break;

        case BP_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::BASE_POINTER);
            break;

        case BX_DISPLACEMENT:
            offsetString += registers.getAssemblyIdentifier(reg::BX_REGISTER);
            break;
        }

        u16 displacementValue = getValueUsingAddressingMode(modRegRm.getAddressingMode());

        // Only add displacement value assembly if there actually is a displacement value stored:
        if(displacementValue != 0)
            offsetString += style.displacementAdd + convert::numberToAssembly(displacementValue, style);

        return style.displacementBegin + offsetString + style.displacementEnd;
    }

    u16 Displacement::getValueUsingAddressingMode(AddressingMode mode) const {
        switch(mode) {
        case BYTE_DISPLACEMENT:
            return static_cast<u16>(getByteValue());
```

```
105
106         case WORD_DISPLACEMENT:
107             return getWordValue();
108
109         default: return 0; // Invalid addressing mode so just return 0.
110         }
111     }
112
113     AbsAddr Displacement::resolve(AddressingMode mode, DisplacementType type, reg::GeneralRegisters& registers) const {
114         u16 displacementValue = getValueUsingAddressingMode(mode);
115
116         switch(type) {
117         case BX_SI_DISPLACEMENT:
118             return registers.get(reg::BX_REGISTER) +
119                    registers.get(reg::SOURCE_INDEX) +
120                    displacementValue;
121
122         case BX_DI_DISPLACEMENT:
123             return registers.get(reg::BX_REGISTER) +
124                    registers.get(reg::DESTINATION_INDEX) +
125                    displacementValue;
126
127         case BP_SI_DISPLACEMENT:
128             return registers.get(reg::BASE_POINTER) +
129                    registers.get(reg::SOURCE_INDEX) +
130                    displacementValue;
131
132         case BP_DI_DISPLACEMENT:
133             return registers.get(reg::BASE_POINTER) +
134                    registers.get(reg::DESTINATION_INDEX) +
135                    displacementValue;
136
137         case SI_DISPLACEMENT:
138             return registers.get(reg::SOURCE_INDEX) + displacementValue;
139
140         case DI_DISPLACEMENT:
141             return registers.get(reg::DESTINATION_INDEX) + displacementValue;
142
143         case BP_DISPLACEMENT:
144             return registers.get(reg::BASE_POINTER) + displacementValue;
145
146         default: // BX_DISPLACEMENT
147             return registers.get(reg::BX_REGISTER) + displacementValue;
148         }
149     }
150 }
```

9.1.9 Instruction Classes

There are a wide variety of different types of instruction supported by the Intel 8086, meaning that having just a single class to represent an instruction is far from sufficient. Instead, there is an abstract base class `Instruction` from which various other classes inherit.

Each instruction will always have an assembly identifier (e.g. 'hlt', 'add') and an opcode (an instance of the `Opcode` class outlined previously) - see the `Instruction` class below. In addition, a fair few instructions also take a register that is actually indicated by the opcode itself rather than a MOD-REG-R/M byte - for such instructions, see the `InstructionTakingRegister` class below.

```
1   #pragma once
2
3   #include <string>
4   #include <vector>
5   #include "emu/types.hpp"
6   #include "emu/cpu/instr/opcode.hpp"
7   #include "emu/cpu/instr/modregrm.hpp"
8   #include "emu/cpu/instr/argument.hpp"
9   #include "assembly.hpp"
10
11  namespace emu::cpu { class Intel8086; } // Declared as incomplete type as including the CPU header here would cause a
12                                          // circular dependency.
13
14  namespace emu::cpu::instr {
15      class Instruction {
16      public:
17          /**
18           * Create a new instruction representation object.
19           *
20           * @param instrIdentifier Assembly code identifier for this instruction (e.g. "mov").
21           * @param instrOpcode The Opcode object of this instruction.
22           */
23          Instruction(std::string instrIdentifier, Opcode instrOpcode);
24
25          /**
26           * Execute this instruction using the given CPU internal values.
27           *
28           * Is pure virtual and must therefore be overridden by subclasses.
29           *
30           * It is the role of this method to return the appropriate instruction pointer value so that the correct next
31           * instruction is executed. Unless a jumping/calling instruction, in most instances it is best to return the
```

```
         * current instruction pointer plus the value returned by Instruction::getRawSize.
         *
         * @param cpu Reference to the CPU on which this instruction is being executed.
         * @param memory Reference to the memory managed by the CPU.
         * @return The new instruction pointer value after completing execution.
         */
        virtual OffsetAddr execute(Intel8086& cpu, Mem& memory) = 0;

        /**
         * Disassemble this instruction into assembly code (expressed as std::string).
         *
         * Is virtual and can therefore be overrriden. By default, simply returns instruction identifier.
         *
         * @param cpu Takes a constant reference to the CPU that this CPU was/will be executed on.
         * @param style The assembly style use for the disassembly of this instruction (constant reference to
         *        assembly::Style struct instance).
         * @return The assembly representation of this instruction.
         */
        virtual std::string toAssembly(const Intel8086& cpu, const assembly::Style& style) const;

        /**
         * Fetch the raw 8-bit values that make this instruction include the opcode value.
         *
         * By default, only returns the opcode raw value. More complex instructions that take a MOD-REG-R/M byte,
         * displacement value or immediate value should override this method.
         */
        virtual std::vector<u8> getRawData() const;

        /**
         * Fetch the raw 8-bit values that make up this instruction expressed as a string.
         *
         * @param separator The separating string placed between each binary value in the string representation.
         */
        std::string getRawDataString(std::string separator = ", ") const;

        /**
         * Returns the number of bytes that make up this instruction.
         */
        OffsetAddr getRawSize() const;

        /**
         * Returns the address of the instruction that should be run after this one (assuming that this instruction is
         * not a jump instruction or otherwise modifies the instruction pointer) based on the value of the instruction
         * pointer and the return of Instruction::getRawData method.
         */
        OffsetAddr nextAddress(const Intel8086& cpu) const;

        const std::string identifier;
        const Opcode opcode;
    };

    /**
     * For representing instructions taking a single specific register that is not specified by a MOD-REG-R/M byte.
     *
     * Examples:
     * - PUSH ES (0x06)
     * - INC AX (0x40)
     * - PUSH BP (0x55)
     */
    class InstructionTakingRegister : public Instruction {
    public:
        /**
         * @param generalReg The register index that this instruction takes as an argument.
         * @param part The register part used by this instruction (defaults to using the full 16-bit register).
         */
        InstructionTakingRegister(std::string instrIdentifier, Opcode instrOpcode, reg::GeneralRegister generalReg,
                                  reg::RegisterPart part = reg::FULL_WORD);

        /**
         * Convert this instruction to assembly (simply the instruction identifier followed by the register identifier).
         */
        std::string toAssembly(const Intel8086& cpu, const assembly::Style& style) const override;

    protected:
        const reg::GeneralRegister registerIndex;
        const reg::RegisterPart registerPart;
    };

    /**
     * This instruction halts the execution of the CPU (assembly 'hlt' identifier).
     */
    class HaltInstruction : public Instruction {
    public:
        HaltInstruction(Opcode instrOpcode);

        OffsetAddr execute(Intel8086& cpu, Mem& memory) override;
    };
}
```

```
#include "emu/cpu/instr/instruction.hpp"

#include <functional>
#include "emu/cpu/intel8086.hpp"
#include "logging.hpp"

namespace emu::cpu::instr {
    Instruction::Instruction(std::string instrIdentifier, Opcode instrOpcode)
    : identifier(instrIdentifier), opcode(instrOpcode) {}

    std::string Instruction::toAssembly(const Intel8086&, const assembly::Style&) const {
        return identifier; // By default, simply return the instruction identifier instead of proper assembly.
    }

```

```
    std::vector<u8> Instruction::getRawData() const {
        return { opcode.value }; // By default, only the opcode value is returned as raw data.
    }

    std::string Instruction::getRawDataString(std::string separator) const {
        std::vector<u8> raw = getRawData();
        std::function<std::string(u8)> convertFunction = [](u8 value) { return convert::toBinaryString<8>(value); };

        return convert::vectorToString(raw, convertFunction, separator);
    }

    OffsetAddr Instruction::getRawSize() const {
        std::size_t size = getRawData().size(); // Fetch the raw data vector in order to then obtain its size expressed
                                                // as std::size_t type.
        return static_cast<OffsetAddr>(size); // Cast from std::size_t at compile time (static cast).
    }

    OffsetAddr Instruction::nextAddress(const Intel8086& cpu) const {
        return cpu.getRelativeInstructionPointer() + getRawSize();
    }



    InstructionTakingRegister::InstructionTakingRegister(std::string instrIdentifier, Opcode instrOpcode,
                                                         reg::GeneralRegister generalReg, reg::RegisterPart part)
    : Instruction(instrIdentifier, instrOpcode),
      registerIndex(generalReg), registerPart(part) {}

    std::string InstructionTakingRegister::toAssembly(const Intel8086& cpu, const assembly::Style&) const {
        auto registerIdentifier = cpu.generalRegisters.getAssemblyIdentifier(registerIndex, registerPart);

        return identifier + " " + registerIdentifier;
    }



    HaltInstruction::HaltInstruction(Opcode instrOpcode) : Instruction("hlt", instrOpcode) {}

    OffsetAddr HaltInstruction::execute(Intel8086& cpu, Mem&) {
        cpu.halted = true;

        return nextAddress(cpu);
    }
}
```

The above classes are capable of handling instructions that do not take a MOD-REG-R/M byte. For those that do, I created an abstract `ComplexInstruction` class. This class has a number of pure virtual methods which are called depending on the value of the MOD-REG-R/M byte. This is an example of abstraction as it allows subclasses to simply override the various different virtual methods as appropriate without considering the actual value of the MOD-REG-R/M byte.

```
#pragma once

#include <optional>
#include "emu/cpu/instr/instruction.hpp"
#include "emu/cpu/instr/modregrm.hpp"

namespace emu::cpu::instr {
    /**
     * A 'complex' instruction is one which has a MOD-REG-R/M byte and may also have a displacement and/or immediate
     * value encoded as part of the instruction.
     */
    class ComplexInstruction : public Instruction {
    public:
        ComplexInstruction(std::string instrIdentifier, Opcode instrOpcode, ModRegRm instrModRegRm,
                           std::optional<Displacement> displacement = {}, std::optional<Immediate> immediate = {});

        OffsetAddr execute(Intel8086& cpu, Mem& memory) override final;

        std::string toAssembly(const Intel8086& cpu, const assembly::Style& style) const override final;

        std::vector<u8> getRawData() const override final;

    protected:
        /**
         * Pure virtual method that converts instruction arguments into assembly when the MOD-REG-R/M component
         * indicates no displacement addressing mode.
         */
        virtual std::string argumentsToAssemblyNoDisplacement(const Intel8086& cpu,
                                                              const assembly::Style& style) const = 0;

        /**
         * Pure virtual method that converts instruction arguments into assembly when the MOD-REG-R/M component
         * indicates either byte or word displacement addressing mode.
         */
        virtual std::string argumentsToAssemblyDisplacement(const Intel8086& cpu,
                                                            const assembly::Style& style) const = 0;

        /**
         * Pure virtual method that converts instruction arguments into assembly when the MOD-REG-R/M component
         * indicates register addressing mode.
         */
        virtual std::string argumentsToAssemblyRegisterAddressingMode(const Intel8086& cpu,
                                                                      const assembly::Style& style) const = 0;

        /**
```

```
         * Arrange two assembly instruction arguments based on the register direction value given.
         *
         * @param reg The assembly argument derrived from the REG component of the MOD-REG-R/M byte.
         * @param rm The assembly instruction argument derrived from the R/M component.
         * @param direction The direction of the instruction (usually indicated by the opcode).
         */
        std::string argumentsToAssemblyOpcodeDirection(std::string reg, std::string rm, const assembly::Style& style,
                                                       RegDirection direction) const;

        virtual void executeNoDisplacement(Intel8086& cpu, Mem& memory) = 0;
        virtual void executeByteDisplacement(Intel8086& cpu, Mem& memory) = 0;
        virtual void executeWordDisplacement(Intel8086& cpu, Mem& memory) = 0;
        virtual void executeRegisterAddressingMode(Intel8086& cpu, Mem& memory) = 0;

        const ModRegRm modRegRm;
        const std::optional<Displacement> displacementValue;
        const std::optional<Immediate> immediateValue;
    };
}
```

```
#include "emu/cpu/instr/complexinstruction.hpp"

#include "emu/cpu/intel8086.hpp"
#include "logging.hpp"

namespace emu::cpu::instr {
    ComplexInstruction::ComplexInstruction(std::string instrIdentifier, Opcode instrOpcode, ModRegRm instrModRegRm,
                                           std::optional<Displacement> displacement, std::optional<Immediate> immediate)
    : Instruction(instrIdentifier, instrOpcode), modRegRm(instrModRegRm),
      displacementValue(displacement), immediateValue(immediate) {}

    OffsetAddr ComplexInstruction::execute(Intel8086& cpu, Mem& memory) {
        switch(modRegRm.getAddressingMode()) {
        case NO_DISPLACEMENT:
            executeNoDisplacement(cpu, memory); break;

        case BYTE_DISPLACEMENT:
            executeByteDisplacement(cpu, memory); break;

        case WORD_DISPLACEMENT:
            executeWordDisplacement(cpu, memory); break;

        case REGISTER_ADDRESSING_MODE:
            executeRegisterAddressingMode(cpu, memory); break;
        }

        return nextAddress(cpu);
    }

    std::string ComplexInstruction::toAssembly(const Intel8086& cpu, const assembly::Style& style) const {
        std::string argumentsStr;

        switch(modRegRm.getAddressingMode()) {
        case NO_DISPLACEMENT:
            argumentsStr = argumentsToAssemblyNoDisplacement(cpu, style); break;

        case REGISTER_ADDRESSING_MODE:
            argumentsStr = argumentsToAssemblyRegisterAddressingMode(cpu, style); break;

        default: // Byte or word displacement:
            argumentsStr = argumentsToAssemblyDisplacement(cpu, style); break;
        }

        return identifier + " " + argumentsStr;
    }

    std::vector<u8> ComplexInstruction::getRawData() const {
        std::vector<u8> data = { opcode.value, modRegRm.value };

        if(displacementValue) convert::extendVector(data, displacementValue->getRawData());
        if(immediateValue) convert::extendVector(data, immediateValue->getRawData());

        return data;
    }

    std::string ComplexInstruction::argumentsToAssemblyOpcodeDirection(std::string reg, std::string rm,
                                                                       const assembly::Style& style,
                                                                       RegDirection direction) const {
        std::string arguments = "";

        switch(direction) {
        case REG_IS_SOURCE:
            arguments = rm + style.argumentSeparator + reg; break;

        case REG_IS_DESTINATION:
            arguments = reg + style.argumentSeparator + rm; break;
        }

        return arguments;
    }
}
```

With this system, I was able to begin the (somewhat tedious) process of creating classes deriving from these instruction classes for each individual instruction. With the flexibility of the system created, this process was thankfully quite simple. Indeed, for most instructions, what the instruction actually does is not overly complex, it is just that the act of decoding it tends to be rather involved.

9.1.10 Yet More Conversion/Helper Functions

One may have noticed in the above code samples that some new conversion/helper functions need to be introduced. These are outlined in the header and source files below:

```hpp
#pragma once

#include <string>
#include <sstream>
#include <bitset>
#include <vector>
#include <functional>
#include <optional>
#include "assembly.hpp"
#include "primitives.hpp"

namespace convert {
    /**
     * Returns the most significant byte of the given value.
     *
     * @param value Value to fetch high byte of.
     * @return Most significant byte of value.
     */
    u8 getMostSigByte(u16 value);

    /**
     * Returns the least significant byte of the given value.
     *
     * @param value Value to fetch the low byte of.
     * @return Least significant byte of value.
     */
    u8 getLeastSigByte(u16 value);

    /**
     * Create a 16-bit word from a low and a high byte.
     *
     * @param low The least significant byte.
     * @param high The most significant byte.
     * @return 16-bit word constructed from the low/high bytes.
     */
    u16 createWordFromBytes(u8 low, u8 high);

    /**
     * Convert a bit (expressed as a boolean) into either the string "0" (false) or "1" (true).
     *
     * @param bit Boolean bit value.
     * @return Either the string "1" or "0".
     */
    std::string bitAsStr(bool bit);

    /**
     * Convert a vector of items into a single string.
     *
     * @tparam T Type used by the vector.
     * @param items The vector to convert to string.
     * @param convertFunction A function for converting each item in the vector to a string.
     * @param separator The string placed between the string representation of each item in the vector.
     * @return The string representation of the given vector.
     */
    template <typename T>
    std::string vectorToString(const std::vector<T> &items, std::function<std::string(T)> convertFunction,
                               std::string separator = ", ") {
        std::string str;
        const auto size = items.size();

        for(unsigned int i = 0; i < size - 1; i++)
            str += convertFunction(items[i]) + separator; // Add all elements (except the final one) with separator.

        str += convertFunction(items[size - 1]); // Add final element (no separator).

        return str;
    }

    /**
     * Take an existing vector and extend it by the values in a second vector.
     *
     * @tparam T Type handled by the vectors used.
     * @param base Reference to the vector which will be extended.
     * @param extra Constant reference to the vector of values to be added to the base vector.
     */
    template <typename T>
    void extendVector(std::vector<T>& base, const std::vector<T>& extra) {
        base.insert(base.end(), extra.begin(), extra.end());
    }

    /**
     * Convert a numerical value to a string representation in binary format. This is done using a `std::bitset`.
     *
     * @tparam T Type of numerical value to convert to binary string.
     * @tparam bitCount The number of bits of the given value to display.
     * @param value Value to convert to binary string.
     * @param prefix Prefix string value to prefix (defaults to an empty string).
     * @param suffix Suffix string value to append (defaults to an empty string).
     * @return The binary string representation of the given value.
     */
    template <std::size_t BitCount, typename T>
    std::string toBinaryString(T value, std::string prefix = "", std::string suffix = "") {
        std::bitset<BitCount> bits(value);

        std::stringstream stream;
```

```
96          stream << prefix << bits << suffix;
97
98          return stream.str();
99      }
100
101     /**
102      * Convert a numerical value to a string representation in hexadecimal format. This is done using the `std::hex`
103      * into a string stream.
104      *
105      * @tparam T Type of numerical value to convert to hexadecimal string.
106      * @param value Value convert to hexadecimal string.
107      * @param prefix Prefix string value to prefix (defaults to an empty string).
108      * @param suffix Suffix string value to append (defaults to an empty string).
109      * @return The hexadecimal string representation of the given value.
110      */
111     template <typename T>
112     std::string toHexString(T value, std::string prefix = "", std::string suffix = "") {
113         std::stringstream stream;
114
115         stream << std::hex << std::uppercase << std::noshowbase
116                << prefix
117                << +value // The '+' prefix ensures 'char' types are interpreted as numerical rather than as characters.
118                << suffix;
119
120         return stream.str();
121     }
122
123     /**
124      * Convert a numerical to string using a style/format specified by a given assembly::Style struct instance. Relies
125      * on convert::toHexString for conversions to hexadecimal, on convert::toBinaryString for conversions to binary, and
126      * std::to_string otherwise.
127      */
128     template <typename T, std::size_t BitCount = 16>
129     std::string numberToAssembly(T value, const assembly::Style& style) {
130         switch(style.numericalRepresentation) {
131         case assembly::HEX_REPRESENTATION:
132             switch(style.numericalStyle) {
133             case assembly::WITH_SUFFIX:
134                 return toHexString<T>(value, "", style.hexSuffix);
135
136             case assembly::WITH_PREFIX:
137                 return toHexString<T>(value, style.hexPrefix, "");
138
139             default: // assembly::WITHOUT_SUFFIX_OR_PREFIX
140                 return toHexString<T>(value, "", "");
141             }
142
143         case assembly::BINARY_REPRESENTATION:
144             switch(style.numericalStyle) {
145             case assembly::WITH_SUFFIX:
146                 return toBinaryString<BitCount, T>(value, "", style.binarySuffix);
147
148             case assembly::WITH_PREFIX:
149                 return toBinaryString<BitCount, T>(value, style.binaryPrefix, "");
150
151             default: // assembly::WITHOUT_SUFFIX_OR_PREFIX
152                 return toBinaryString<BitCount, T>(value, "", "");
153             }
154
155         default: // assembly::DENARY_REPRESENTATION
156             return std::to_string(value);
157         }
158     }
159
160     /**
161      * Convert a string representation of a number in any base to an actual numerical type. Utilises the std::stoull
162      * function added in C++11 in order to facilitate this. Will return an empty optional should the aforementioned
163      * function throw an exception.
164      *
165      * @tparam T Type of numerical type to be converted to.
166      * @param str Constant reference to the string.
167      * @param base Numerical base (defaults to 10).
168      * @return An optional that will contain the value expressed in the given string should the conversion complete
169      *         successfully.
170      */
171     template <typename T>
172     std::optional<T> fromString(const std::string& str, int base = 10) {
173         try {
174             T value = static_cast<T>(std::stoull(str, nullptr, base));
175
176             return std::make_optional<T>(value);
177         }
178         catch(std::invalid_argument&) {}
179         catch(std::out_of_range&) {}
180
181         return {};
182     }
183
184     /**
185      * Calls convert::fromString with a base 2 argument (i.e. binary).
186      */
187     template <typename T>
188     std::optional<T> fromBinaryString(const std::string& str) {
189         return fromString<T>(str, 2);
190     }
191
192     /**
193      * Calls convert::fromString with a base 16 argument (i.e. hexadecimal).
194      */
195     template <typename T>
196     std::optional<T> fromHexString(const std::string& str) {
197         return fromString<T>(str, 16);
198     }
199
200     /**
```

```
     * Fetch a specific bit of a numerical value (expressed as a boolean).
     *
     * @tparam T Numerical type to fetch bit from.
     * @param value Value to fetch bit of.
     * @param index Specify which bit to fetch. Indexing begins from 0 with the least significant bit up to the most
     *              significant.
     * @return The fetched bit expressed as a boolean value.
     */
    template <typename T>
    inline bool getBitFrom(T value, unsigned int index) {
        return (value >> index) & 1;
    }

    /**
     * Fetch multiple bits from a numerical value.
     *
     * @tparam T Numerical type to fetch bits from.
     * @param value Value to fetch bits from.
     * @param index Specify the beginning of the sequence of bits to fetch (indexing beginning from 0).
     * @param count The number of bits to fetch starting from the specified index.
     * @return The fetched sequence of bits.
     */
    template <typename T>
    inline T getBitsFrom(T value, unsigned int index, unsigned int count) {
        T mask = (1 << count) - 1; // (2 ^ count) - 1
        return (value >> index) & mask;
    }
}
```

```
#include "convert.hpp"

namespace convert {
    u8 getMostSigByte(u16 value) {
        return (value >> 8) & 0xFF;
    }

    u8 getLeastSigByte(u16 value) {
        return value & 0xFF;
    }

    u16 createWordFromBytes(u8 low, u8 high) {
        return (high << 8) + low;
    }

    std::string bitAsStr(bool bit) {
        return bit ? "1" : "0";
    }
}
```

9.1.11 CPU Implementation

With instructions finally implemented, it is now possible to start work on the last piece of the puzzle: the implementation of the CPU itself.

```
#include "emu/cpu/intel8086.hpp"

#include "logging.hpp"
#include "emu/cpu/instr/stack.hpp"
#include "emu/cpu/instr/arithmeticlogic.hpp"

namespace emu::cpu {
    AbsAddr Intel8086::resolveAddress(OffsetAddr offset, reg::SegmentRegister segment) const {
        OffsetAddr segmentAddress = segmentRegisters.get(segment);

        return (segmentAddress << 4) + offset;
    }

    OffsetAddr Intel8086::getRelativeInstructionPointer() const {
        return instructionPointer;
    }

    AbsAddr Intel8086::getAbsoluteInstructionPointer() const {
        return resolveAddress(instructionPointer, reg::CODE_SEGMENT);
    }

    std::unique_ptr<instr::Instruction> Intel8086::fetchDecodeInstruction(AbsAddr address, const Mem& memory) const {
        MemValue opcodeValue = memory.read(address);
        instr::Opcode opcode(opcodeValue);

        std::unique_ptr<instr::Instruction> instruction;

        // First attempt to decode instruction without MOD-REG-R/M byte:
        instruction = fetchDecodeWithoutModRegRm(opcode);
        // Failing that, attempt to decode instruction with the assumption that it has a MOD-REG-R/M component:
        if(!instruction) instruction = fetchDecodeWithModRegRm(opcode, address, memory);

        if(!instruction)
            logging::warning("Encountered instruction with non-existent or currently unimplemented opcode: " +
                             opcode.toString());

        return instruction;
    }
```

```
39
40      std::unique_ptr<instr::Instruction> Intel8086::fetchDecodeWithoutModRegRm(const instr::Opcode& opcode) const {
41          switch(opcode.value) {
42
43              /*
44               * Many case statements - removed so as to not needlessly fill the document.
45               */
46
47              case 0xF4: // HLT
48                  return std::make_unique<instr::HaltInstruction>(opcode);
49              }
50
51              return {};
52          }
53
54          std::unique_ptr<instr::Instruction> Intel8086::fetchDecodeWithModRegRm(const instr::Opcode& opcode, AbsAddr address,
55                                                                                 const Mem& memory) const {
56              MemValue modRegRmValue = memory.read(address + 1); // MOD-REG-R/M byte immediately follows opcode.
57              instr::ModRegRm modRegRm(modRegRmValue);
58
59              std::optional<instr::Displacement> displacement;
60
61              if(modRegRm.isDisplacementUsed()) {
62                  auto displacementValues = memory.read(address + 2, modRegRm.getDisplacementReadLength());
63                  displacement = instr::Displacement(displacementValues);
64              }
65
66              switch(opcode.getUniqueValue()) {
67              case 0b000000: // ADD E, G
68                  return std::make_unique<instr::AddEG>(opcode, modRegRm, displacement);
69
70              /*
71               * Many case statements - removed so as to not needlessly fill the document.
72               */
73              }
74
75              return {};
76          }
77
78          bool Intel8086::executeInstruction(std::unique_ptr<instr::Instruction>& instruction, Mem& memory) {
79              if(halted) {
80                  logging::warning("Instruction could not be executed due to halted CPU state.");
81                  return false;
82              }
83
84              if(instruction) {
85                  OffsetAddr newIp = instruction->execute(*this, memory);
86
87                  if(memory.withinBounds(newIp)) {
88                      instructionPointer = newIp;
89
90                      return true; // Success!
91                  }
92                  else logging::error("Instruction returned new instruction pointer value that is out of bounds!");
93              }
94              else logging::error("Empty instruction pointer passed to CPU.");
95
96              return false;
97          }
98
99          void Intel8086::pushToStack(MemValue value, Mem& memory) {
100             OffsetAddr stackPointer = generalRegisters.get(reg::STACK_POINTER);
101
102             if(stackPointer > 0) {
103                 stackPointer--;
104
105                 AbsAddr address = resolveAddress(stackPointer, reg::STACK_SEGMENT);
106                 memory.write(address, value);
107
108                 generalRegisters.set(reg::STACK_POINTER, stackPointer);
109             }
110             else logging::warning("Stack pointer is zero so cannot successfully push value: " +
111                                   convert::toHexString(value));
112         }
113
114         MemValue Intel8086::popFromStack(const Mem& memory) {
115             OffsetAddr initialStackPointer = generalRegisters.get(reg::STACK_POINTER);
116             generalRegisters.set(reg::STACK_POINTER, initialStackPointer + 1);
117
118             AbsAddr address = resolveAddress(initialStackPointer, reg::STACK_SEGMENT);
119             return memory.read(address);
120         }
121
122         void Intel8086::pushWordToStack(u16 value, Mem& memory) {
123             pushToStack(convert::getMostSigByte(value), memory);
124             pushToStack(convert::getLeastSigByte(value), memory);
125         }
126
127         u16 Intel8086::popWordFromStack(const Mem& memory) {
128             u8 low = popFromStack(memory);
129             u8 high = popFromStack(memory);
130             return convert::createWordFromBytes(low, high);
131         }
132
133         void Intel8086::performRelativeJump(OffsetAddr offset) {
134             instructionPointer = offset;
135         }
136     }
```

9.2 Testing

9.2.1 Conversions

In this development cycle, a number of new conversion/helper functions were introduced into the codebase. Naturally, unit testing seemed the best method with which to test such code.

```
SECTION("Test conversion from a numerical value to a hexadecimal string representation.") {
        REQUIRE(toHexString<u8>(0xFF) == "FF");
        REQUIRE(toHexString<u16>(0xABCD, "0x") == "0xABCD");
        REQUIRE(toHexString<u16>(0x55A, "", "h") == "55Ah");
    }

    SECTION("Test conversion from a numerical value to a binary string representation.") {
        REQUIRE(toBinaryString<8, u8>(0b10101010) == "10101010");
        REQUIRE(toBinaryString<6, u8>(0b111111, "0b") == "0b111111");
        REQUIRE(toBinaryString<4, u16>(0xFF, "", "b") == "1111b");
    }

    SECTION("Test the extension of vectors.") {
        std::vector<u16> vec = { 0xA, 0xB },
                         extendBy = { 0xC, 0xD, 0xE },
                         expected = { 0xA, 0xB, 0xC, 0xD, 0xE };

        extendVector<u16>(vec, extendBy);
        REQUIRE(vec == expected);
    }

    SECTION("Test conversion from vector to string.") {
        std::vector<u16> values = { 0xAA, 0xBB, 0xCC };
        std::function<std::string(u16)> convert = [](auto value) { return toHexString(value); };

        REQUIRE(vectorToString(values, convert, ", ") == "AA, BB, CC");
    }

    SECTION("Test conversion from hexadecimal string representation to numeric type.") {
        REQUIRE(fromHexString<u16>("0xFF") == 0xFF);
        REQUIRE(fromHexString<u16>("    ab0  ignored") == 0xAB0);
        REQUIRE(fromHexString<u32>("\n\tabcdef") == 0xABCDEF);
    }

    SECTION("Test conversion from decimal string representation to a numeric type.") {
        REQUIRE(fromBinaryString<u16>("1010") == 0b1010);
        REQUIRE(fromBinaryString<u8>("   1111  abc") == 0b1111);
    }

    SECTION("Test conversion from numerical value to string based on assembly style specified.") {
        assembly::Style s;

        s.numericalRepresentation = assembly::HEX_REPRESENTATION;
        s.numericalStyle = assembly::WITH_PREFIX;
        s.hexPrefix = "0x";

        REQUIRE(numberToAssembly<u16>(0xFF, s) == "0xFF");

        s.numericalRepresentation = assembly::BINARY_REPRESENTATION;
        s.numericalStyle = assembly::WITHOUT_SUFFIX_OR_PREFIX;

        REQUIRE(numberToAssembly<u8, 8>(0b1010, s) == "00001010");

        s.numericalRepresentation = assembly::DENARY_REPRESENTATION;

        REQUIRE(numberToAssembly<u8>(25, s) == "25");
    }
}
```

9.2.2 Instruction Representation

Instructions are represented using a collection of classes and components which can (for the most part) function separately of one another, meaning unit testing is again an appropriate method of testing.

```
TEST_CASE("Test CPU instruction representation.", "[emu][cpu][instructions]") {
    using namespace emu::cpu;

    SECTION("Test checking the direction and data size of instruction based on opcode value.") {
        instr::Opcode firstOpcode(0b10101010);

        REQUIRE(firstOpcode.getUniqueValue() == 0b101010);

        REQUIRE_FALSE(firstOpcode.getWordBit());
        REQUIRE(firstOpcode.getDataSize() == instr::BYTE_DATA_SIZE);

        REQUIRE(firstOpcode.getDirectionBit());
        REQUIRE(firstOpcode.getDirection() == instr::REG_IS_DESTINATION);

        instr::Opcode secondOpcode(0b1010101);
```

```
16
17          REQUIRE(secondOpcode.getUniqueValue() == 0b10101);
18
19          REQUIRE(secondOpcode.getWordBit());
20          REQUIRE(secondOpcode.getDataSize() == instr::WORD_DATA_SIZE);
21
22          REQUIRE_FALSE(secondOpcode.getDirectionBit());
23          REQUIRE(secondOpcode.getDirection() == instr::REG_IS_SOURCE);
24      }
25
26      SECTION("Test MOD-REG-R/M byte representation.") {
27          SECTION("Test MOD component and fetching of addressing mode.") {
28              instr::ModRegRm noDisplace(0b00110011);
29              REQUIRE(noDisplace.getModBits() == 0b00);
30              REQUIRE(noDisplace.getAddressingMode() == instr::NO_DISPLACEMENT);
31
32              instr::ModRegRm byteDisplace(0b01010101);
33              REQUIRE(byteDisplace.getModBits() == 0b01);
34              REQUIRE(byteDisplace.getAddressingMode() == instr::BYTE_DISPLACEMENT);
35
36              instr::ModRegRm wordDisplace(0b10101010);
37              REQUIRE(wordDisplace.getModBits() == 0b10);
38              REQUIRE(wordDisplace.getAddressingMode() == instr::WORD_DISPLACEMENT);
39
40              instr::ModRegRm regAddressing(0b11001100);
41              REQUIRE(regAddressing.getModBits() == 0b11);
42              REQUIRE(regAddressing.getAddressingMode() == instr::REGISTER_ADDRESSING_MODE);
43          }
44
45          SECTION("Test REG component.") {
46              instr::ModRegRm axReg(0b11000101);
47              REQUIRE(axReg.getRegBits() == 0b000);
48              REQUIRE(axReg.getRegisterIndexFromReg(instr::WORD_DATA_SIZE) == reg::AX_REGISTER);
49              REQUIRE(axReg.getRegisterPartFromReg(instr::WORD_DATA_SIZE) == reg::FULL_WORD);
50
51              instr::ModRegRm clReg(0b11001010);
52              REQUIRE(clReg.getRegBits() == 0b001);
53              REQUIRE(clReg.getRegisterIndexFromReg(instr::BYTE_DATA_SIZE) == reg::CX_REGISTER);
54              REQUIRE(clReg.getRegisterPartFromReg(instr::BYTE_DATA_SIZE) == reg::LOW_BYTE);
55
56              instr::ModRegRm dhReg(0b11110000);
57              REQUIRE(dhReg.getRegBits() == 0b110);
58              REQUIRE(dhReg.getRegisterIndexFromReg(instr::BYTE_DATA_SIZE) == reg::DX_REGISTER);
59              REQUIRE(dhReg.getRegisterPartFromReg(instr::BYTE_DATA_SIZE) == reg::HIGH_BYTE);
60
61              instr::ModRegRm diReg(0b11111001);
62              REQUIRE(diReg.getRegBits() == 0b111);
63              REQUIRE(diReg.getRegisterIndexFromReg(instr::WORD_DATA_SIZE) == reg::DESTINATION_INDEX);
64              REQUIRE(diReg.getRegisterPartFromReg(instr::WORD_DATA_SIZE) == reg::FULL_WORD);
65          }
66
67          SECTION("Test the fetching of displacement types via R/M component.") {
68              REQUIRE(instr::ModRegRm(0b10000000).getDisplacementType() == instr::BX_SI_DISPLACEMENT);
69              REQUIRE(instr::ModRegRm(0b10111001).getDisplacementType() == instr::BX_DI_DISPLACEMENT);
70              REQUIRE(instr::ModRegRm(0b01001010).getDisplacementType() == instr::BP_SI_DISPLACEMENT);
71              REQUIRE(instr::ModRegRm(0b00101011).getDisplacementType() == instr::BP_DI_DISPLACEMENT);
72              REQUIRE(instr::ModRegRm(0b10111100).getDisplacementType() == instr::SI_DISPLACEMENT);
73              REQUIRE(instr::ModRegRm(0b01000101).getDisplacementType() == instr::DI_DISPLACEMENT);
74              REQUIRE(instr::ModRegRm(0b00101110).getDisplacementType() == instr::BP_DISPLACEMENT);
75              REQUIRE(instr::ModRegRm(0b10111111).getDisplacementType() == instr::BX_DISPLACEMENT);
76          }
77      }
78
79      SECTION("Test immediate instruction value representation.") {
80          std::vector<u8> immediateData = { 0xAA, 0xBB };
81          instr::Immediate immediate(immediateData);
82
83          REQUIRE(immediate.getRawData() == immediateData);
84          REQUIRE(immediate.getByteValue() == 0xAA);
85          REQUIRE(immediate.getWordValue() == 0xBBAA);
86      }
87
88      SECTION("Test displacement instruction value representation.") {
89          std::vector<u8> displacementData = { 0xAA };
90          instr::Displacement displacement(displacementData);
91
92          REQUIRE(displacement.getRawData() == displacementData);
93      }
94  }
```

9.2.3 Stack

In section 9.1.11, a number of additions and modifications were made to the `Intel8086` class, with one such being the introduction of methods for manipulation of the stack.

```
1   using namespace emu;
2
3   Mem memory(0xFF);
4   cpu::Intel8086 cpu;
5
6   cpu.generalRegisters.set(cpu::reg::STACK_POINTER, 0xAA);
7   cpu.segmentRegisters.set(cpu::reg::STACK_SEGMENT, 0);
8
9   SECTION("Test CPU stack handling.") {
10      cpu.pushToStack(0xAB, memory);
```

```
11      cpu.pushToStack(0xCD, memory);
12      cpu.pushToStack(0xEF, memory);
13
14      REQUIRE(cpu.popFromStack(memory) == 0xEF);
15      REQUIRE(cpu.popFromStack(memory) == 0xCD);
16      REQUIRE(cpu.popFromStack(memory) == 0xAB);
17
18      cpu.pushWordToStack(0xABCD, memory);
19      REQUIRE(cpu.popWordFromStack(memory) == 0xABCD);
20  }
```

9.2.4 Unknown Instruction Handling

One of the success criteria outlined in table 1 requires that unknown/unimplemented instructions do not cause the emulator to crash or encounter other such issues. This could be considered a form of input/data validation as it prevents the user from crashing the program by trying to make the emulated CPU execute an instruction that cannot be executed.

Using the CLI front-end interface of my emulator, I passed in a number of instructions I know are not implemented. Each time, this resulted in output similar to the following:

```
00000000 <.data>:
   0:   15 0a 00                adc    ax,0xa
[INFO - 15:05:18] --- CYCLE 1 ---
[INFO - 15:05:18] Fetching instruction from address: 0
[WARNING - 15:05:18] Encountered instruction with nonexistent or currently unimplemented opcode: 15 (000101dw : d=0, w=1)
[ERROR - 15:05:18] Failed to decode instruction - halting...
[WARNING - 15:05:18] Detected that CPU is now in halted state. Remaining cycles will not be executed.
[INFO - 15:05:18] --- TOTAL 1 OF 25 CYCLES COMPLETED ---
```

The emulator did not suddenly crash but rather display a logging warning before halting CPU execution when an unknown instruction is encountered. Thus, this fulfils that portion of the project's success criteria.

9.3 Review

9.3.1 Overview of Progress

Development of emulator is now complete - all components are now implemented and testing suggest they are functioning correctly and as expected. Users are able to load programs into the emulated system memory, see the assembly code of each instruction, and have those instructions executed by the emulated Intel 8086 microprocessor.

9.3.2 Success Criteria

All of the success criteria outlined in table 1 have now been addressed:

- The emulated CPU can access, read, and write to emulated system memory. This memory can written to and read from the file system also.
- The emulated CPU is able to execute a number of instructions, including those with MOD-REG-R/M, immediate, and/or displacement bytes.
- The program is able to produce valid assembly representations of instructions before executing them.

- The program does not crash when an attempt to execute an unknown/invalid instruction is made.

10 Example Instruction Decodings

Assembly	Opcode Byte			MOD-REG-RM Byte			Displacement Byte(s)		Immediate Byte(s)	
Intel Syntax	Opcode	D-bit (direction)	W-bit (word)	MOD	REG	R/M	Low	High	Low	High
`add cx, bx`	00000001 add	0 REG is source	1 word data size	11 register addressing	011 BX source	001 CX destination				
`add al, [bx+5]`	00000010 add	1 REG is destination	0 byte data size	01 byte displacement	000 AL dest	111 BX + displacement	00000101 displace by 5			
`mov word [bp+0x1234], 0x5678`	11000111 mov Ew, Iw	1 irrelevant	1 word data size	10 word displacement	000 unused	110 BP + displacement	00110100 0x34	00010010 0x12	01111000 0x78	01010110 0x56

Table 8: *Table showing example instruction decodings.*

11 Full Interview

11.1 Are you satisfied with the resources you currently have available for teaching about the low-level workings of computing systems?

Do you know, that I have long thought how unsatisfied I am with my computing resources for teaching. Don't go quote me word-for-word. I'm only really aware of the little man computer simulation - it's quite basic but it quite good for GCSE-level students. However there doesn't appear to be anything suitable for A Level that provides greater detail for higher-level students. Gap in the market, for sure.

11.2 Have you considered implementing practical demonstrations into such lessons?

Yes, I think practical demonstrates are a real benefit when trying to get an idea across to a class - a great asset. I tried with the little man computer however it can be rather difficult for students to follow and understand.

11.3 If so, did you find that it enhanced the learning experience of your students?

Definitely, it is potential be a very dry, abstract topic - so yes, if you can represent it visually, I think that really helps people to understand it.

11.4 Have you before considered performing demonstrations using more simplistic, early computer systems (whether physical or emulated) to help illustrating your teaching points?

(Too lazy) I have considered it yet I'm often uncertain as to where I could get such systems - whether I would have time to setup in a classroom. Any I could find tend to be expensive and unreliable (old technology).

11.5 What features would you look for in an emulator to make it most applicable for usage in a teaching environment?

I would like to see the assembly code and how that correlates into opcode, operands - some representation of this in memory. How the different registers are being used - what they're storing. Anything that provides a bit of detail, a real time view of what is going on - how programming instruction is temporarily stored and then carried out. In a way, the simpler the better - students need to clearly see that connection between a simple program and how it runs.

11.6 What would, in your opinion, be the ideal interface for such a piece of software?

I'm imagining a GUI on my screen that I can use to get a bit of a demonstration of how of the operation of the system. Maybe I could hover my mouse over something and get an indication of what that particular part does. Making it as user friendly as possible would also be ideal. Colour coding could be useful to see what is active at each moment - draw one's eye to the right bit at the right time. Some sort of menu system at the top?

12 Sources

- The primary source for the specifications of the processor: *The Intel 8086 Family: User's Manual* published in 1979 by the Intel Corporation.
- Convenient reference of Intel 8086 instructions: `www.mlsite.net/8086/`
- Helpful breakdown of instruction encoding: `www-user.tu-chemnitz.de/ heha/viewchm.php/hs/x86.chm/x86.htm`