

Contents

Analysis

<u>Item</u>	<u>Page relative to section</u>
Description of program and justification of computational methods	0-7
Identification of stakeholders, research and explanation of needs	13-24
Research of problem and of existing solutions	8-13
Identification and explanation of essential features	24-29
Justification of limitations	29-31
Hardware and software requirements	31-33
Success criteria	33-42

Design

<u>Item</u>	<u>Page relative to section</u>
Justification of problem decomposition	1-3 (continued throughout section)
Defined in detail the structure of the solution	3-200
Pseudocode and algorithms	19,21,51,53,54,56,57,59,61,63,65,66-67,70,73,75,79-80,84-85,88-89,91-93,94-96,98-101,103,104,106-109,111,112,130,131-132,136,137,138,139,147-148,150,151,154-155,156-157,174,175,176,192 (continued throughout section)
Validation	201-203,6,112,115,116,118,119,120,139
Usability features	4,5,10,13-14,14-15,16-17,18,119
Key Variables	204-205
Test data	continued throughout section

Development

<u>Item</u>	<u>Page relative to section</u>
Evidence of iterative development process	continued throughout section
Evidence of prototype versions	Alpha: 2-14 Beta: 14-34 Developer preview: 34-85 Release version: 85-181
Structure and modular nature	184
Code annotation to aid maintenance	continued throughout section
Appropriate naming of variables and structures	181-183
Validation for key elements of solution	46-49,173-174,174-181,181,183
Review	13-14,15,24-25,26-27,34,45,60,70,74,105,118,126,169

Test evidence	11-13,16-19,21-24,28-33,40-44,57-59,73,82-85,91-92,93-95,95-96,98-99,99-104,110-111,116-117,126-162,171-173
Failed tests and remedial actions	12-13, 17-19,21-23,32-33,43-44,100-102,173

Evaluation

<u>Item</u>	<u>Page relative to section</u>
Success criteria evaluation and cross reference with test evidence	p1-15
Evidence, evaluation, and justification of usability features and their success	p15-23
How the program could be developed to deal with limitations	p23-24 <i>In addition, all success criteria and usability features contain a further development plan.</i>

Virtual Machine C sharp project writeup

Summary

Virtual Machine C Sharp (VMCS) enables users to execute and debug x86-64 assembly programs in a virtual environment. VMCS is written in Microsoft C# on the .NET Framework and runs on the Windows operating system. The application acts both as an IDE for assembly programmers - simplifying how they debug code (by virtue of using a high level language to control their program) and as a traditional virtual machine - allowing the execution of programs on an unsupported platform, e.g. Intel code on a non-Intel chip-set.

The application is modular in design, allowing users to easily extend the system to their own needs and includes support for testcases. Testcases allows arbitrary program samples to be executed without user interaction. This is primarily to test program functionality and reduce regression issues resulting from application changes. A testcase allows all aspects of the application to be tested in the click of a button. Testcases are used both by the application author and the end-user to automate testing and validate any modifications.

Modularity is a major focus of the project. Due to the quantity of features present in the x86-64 architecture, it will obviously not be possible to include support for everything, e.g. interacting with hardware devices. However, by designing the application using classes to abstract opcodes, the application is entirely extensible.

A further goal of the project is to abstract the various layers of presentation and logic, so that the low-level emulation process is entirely separated from the graphical user interface. This leaves open the possibility of implementing alternative methods of interacting with the application, such as remotely via a web interface.

Analysis

Problem Identification

VMCS will solve the problem of no introductory x86-64 assembly programming development environments available to users on Windows, and will allow users to execute and debug these programs in a virtualised environment.

Instructions that the user inputs will be referred to as user-instructions. The features to directly solve this problem will be a Control Unit, which will be an automaton to determine whether a given program P is a member of the set of all valid programs Σ , and if so, determine the result of the program. The set of valid programs will be referred to as the "alphabet". Each program will consist of opcodes, which are a machine level operation which less specific than an instruction. An instruction may be "MOV EAX, 0x10", and the opcode here would be MOV. A valid program is a program that when inputted to the CU, the final state of the program is reached. Whether a program reaches its final state will depend on whether it has been coded by the user correctly, hence cannot be solved by the program, so long as the program provides 1:1 emulation of the x86-64 specification. The final state will be denoted as F. For simplicity, in all examples there will only be one final state; the user-program may be coded to have multiple end points, which will still be considered valid.

For example, consider the following program P1,

BYTES	DISASSEMBLY
89 03	mov eax, ebx
8d 03	lea eax, [ebx]

In pseudo,

```
// q0, Start
EAX = EBX;
EAX = EBX;
// F
```

The CU would execute the LEA instruction, then the final state would be reached as the program is finished. This would be considered a valid program as it is a member of the set of all valid programs of length two. This can be written as,

$$P1 \in \Sigma^2$$

Consider the following P2,

BYTES	DISASSEMBLY
90	nop
e9 00 00 00 00	jmp 0

In pseudo,

```
// q0
WHILE TRUE
    // Nop = No operation, do nothing
    ELIHW;
    // F
```

This would not be considered a valid program as the program would never reach the

final state as it does not reach the final state as it loops indefinitely. In this case,

$$P2 \notin \Sigma^2$$

Another case where a program would not be considered valid would be if its instructions could not be parsed. Not all opcodes will be included in the scope of the project due to their complexity. The majority of users will only use a small subset of opcodes, therefore only what is in demand will be coded. This will be determined through stakeholder research and surveys.

Explanation and demonstration of computational approach

A computation approach will be necessary because the set of all valid programs is an infinite set; any unique program could be created by the user. This means that a hard-coded solution will not be possible, as it would take up an infinite amount of memory, which computers do not have. This means that CU will have to determine whether a program is valid or not. It will read each element of an instruction as a token. A token is an abstraction of the input, it may be a lexicon, number, or special character that the programmer will enter when creating code in any language.. For example, consider this example of high level python code,

```
def MyFunction(input):
    input += "hello";
    return input;
```

The circled parts are the parts of the code that have to be considered separately. The letters: "d", "e", "f" do not matter, rather the meaning that def implies. It was the decision of pythons creators to name this way, just as it was the developers choice to call the variable "input". It is very convenient in planning to abstract this language specific syntax and use symbols instead, because for the purposes of modelling how instructions will be parsed, all instructions will undergo a very similar treatment. There does not need to be a specific operation for decoding MOV that is different to LEA. Another concept that is abstracted at this stage is any new lines, spaces, and comments. The tokens are combined to produce a "string" that can then be processed by computational methods. Firstly lets only consider validating a program, not executing it. This can be modelled through a transition graph.

Consider the set of all valid programs(alphabet) to be any combination of tokens a and b, but not token c.

$$\Sigma \in \{ a, b, ab, aa, bb \dots a^n b^m \}$$

$$c \notin \Sigma$$

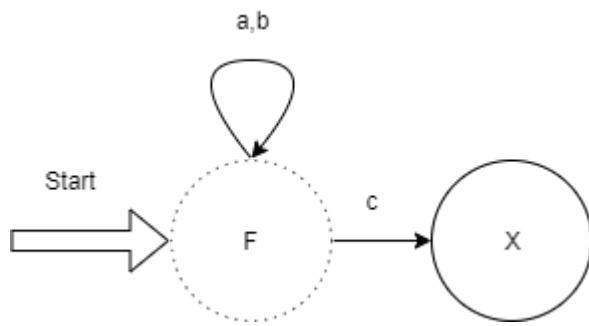
This alphabet is infinite as constants n and m approach infinity. Hence, this problem must be solved computationally as it would not be possible to manually parse this number of tokens because the entire program must be parsed to see if it contains token c. Short programs may be tolerable, but impractical at a larger scale.

Consider this pseudocode to demonstrate a program that would validate strings for this alphabet,

```
DEF ParseProgram(TOKEN[] Tokens)

    // Iterate through the entire input
    FOR Token in Tokens DO
        // Return false if the opcode is of type C because C was defined to
        // be invalid in this alphabet.
        IF Token == C THEN
            RETURN FALSE;
        FI;
    ROF;
    // Return true if the entire input string had only tokens a and b, the program
    // would have returned earlier if there was a token c.
    RETURN TRUE;
FED;
```

Consider the following transition diagram, which is equivalent to the previous pseudocode



The outlined arrow represents the initial state of the machine, the solid outlined circle is a state, the dotted outlined circle F represents the final state. This notation will be used throughout. When the start(and final) state F is reached, it will indefinitely loop through the string whilst the current token is a or b, as shown by the arrow on top. If the token is ever of type C, it will go to state X. State X is what I would describe as a “dead state”, as it will never be able to get back to the final state, which would indicate the success of a program. For a program that consists of only tokens A and B, it will loop through the entire string and finish at state F, which would indicate success. This can be described as a deterministic finite automaton; there is one path for each token to reach a state from F, if state X is reached it is certain there was a C in the string, but a string with the C token would never stay at state F.

This will work very similar when applied to assembly. For example, consider a simplified scenario where each opcode O must be followed by an input I. This alphabet would be defined as,

$$\Sigma \in \{OI, OIOI, \dots (OI)^n\}$$

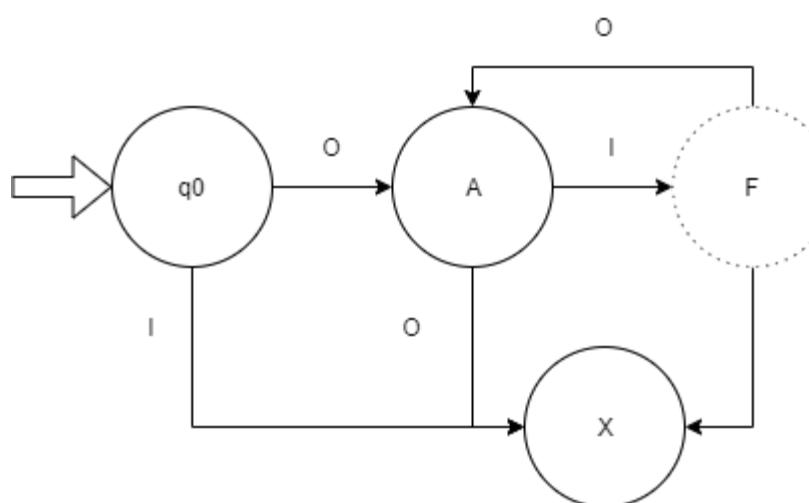
Firstly, pseudocode for a parser,

```

DEF ParseInstruction(TOKEN[] Tokens)
    // Start with the I_Token because there cannot be a valid string that starts
    // with an input I_Token and not an opcode O_Token
    Token LastToken = I_Token;
    FOR Token in Tokens DO
        IF Token == LastToken THEN
            RETURN FALSE;
        FI;
    ROF;
    RETURN TRUE;
FED;

```

This transition diagram represents the same functionality,



q0 is the start state F is the final states. If two consecutive I tokens or O tokens are read, or the first token is an I token, the dead state X is reached. At this point, the string is not read further. A valid string will end at state F, state A would be an invalid string. This is because an O token was left without an I token to follow it, an opcode had no input.

Justification of computational approach

This demonstration has justified that a computational approach is appropriate to be applied to the CU due to the length of valid strings approaching infinity, an entire string must be parsed to determine whether it is valid or not, hence present in the set of all valid programs. In reality, the program would be executed and disassembled whilst being parsed. This is the best approach as the parsing algorithm needs to be $O(1n)$ as was in the previous examples, $O(xn)$ would become very inefficient for larger programs and is unnecessary when it is no more complex than the $O(n)$. In this context, the term for this is “Virtualisation”, instructions are executed in the environment of this program, not in a standalone program. This will allow for faster and easier debugging through provided features such as displaying the values of registers, disassembling the instructions. This information would not be available if the user executed their program on the command line; they would have to design specific procedures to output information instead of having it shown to them, such that it would be harder to debug or determine the results of the program. The systematic and consistent steps taken by the Control Unit will allow faster development of user assembly programs as they do not have to link the object code or deal with any limitations that are irrelevant to the problem the user is trying to solve, which may occur in a low level development process.

Another computable feature would be the operations that opcodes perform. For example, consider addition. In high level code, this is an easy operation through add operators, however it will only work on specific lengths of variables and abstracts lots of information that is important to this situation. Consider bitwise details, such as carries and overflows. These are abstracted by the high level languages, the developer is unlikely to be told about this event unless an exception is thrown(language-specific). In an assembly program, these are stored in flags which then are often used to make decisions. This shows that use of emulation is the best option. This is because there is no mathematical procedure that can emulate this behaviour. If two large positive numbers were added in maths, and the result was negative, the result would be wrong, but in computation is a valid result. This concept can be applied to all of the opcodes, as to achieve 1:1 emulation, they need to emulate the result of running the program on another processor, not the mathematically correct answer. There are also some details that can be abstracted from the approach, such as segmentation faults and alike. A segmentation fault (segfault) occurs when memory that has not been allocated to the program has been accessed. Consider the following assembly code, where x is an arbitrary address.

```
MOV [x], 0x10
```

This would very likely result in a segfault as addresses must be allocated first. This is generally a hassle and requires the use of syscalls to request that the kernel allocates more memory, moreover would differ for every operating system. This is not a very appealing task to ask of someone who codes in high level languages, as they are not used to considering such factors. VMCS will abstract this requirement, allowing memory to be used at will, but using specific techniques to mitigate the performance impact of this, which will be discussed in the design section. This is a necessary feature because the primary aim of the project is to provide a basic introduction of low level programming to high level programmers; if they are bombarded with platform specific tasks and procedures, they will lose interest. In addition, this is not something that is part of the x86-64 specification, it is based on the operating system. If you were to write your own operating system, you would also not have to deal with this. It is an important feature for operating systems in terms of minimising memory usage, but in this context is only an obstacle.

Furthermore, a computational approach will be best because production behaviour can only be predicted reliably by virtualisation that accurately reproduces known reduces. Taking a mathematical approach only introduces error and inaccuracy, and a lot of time for larger programs. To do this, intended behaviour of opcodes will be first defined by using a debugger to step through code in a non-virtualised environment. As these instructions will be executed by the processor, the results be representative of what would happen in a production environment. Any anomalies will be cross referenced with the intel manual, favouring the manual in disagreement. Some processors have specific behaviours that are not present on other models. This is the best approach as to base results solely on one model of processor is not broad enough to be representative of the entire specification. These recorded behaviours will be stored as “testcases” for each opcode, which then introduces the need for the test handler, which will automate these testcases to ensure a modification to the code or a refactoring has not had unintended consequences. This is a necessary computable feature as it automates repetitive behaviour quickly, such that the solution is more maintainable; less time will be spent re-testing each feature individually after a major change, it can all reliably done with the test handler.

Thinking abstractly

Core operational methods of a processor can be abstracted and simplified into tasks that the high-level language of C# can achieve easily. For example, instead of having registers of a specific capacity, a byte array will be used instead to emulate the same behaviour. This is important because a general solution can then be applied to operations of all capacities, rather having a specific operation for specific sizes of integers.

Compatibility features of the specification cannot be abstracted. These include obsolete and outdated behaviours that are still present in x86-64 architecture. For example, the behaviour of the overflow flag in shift operations. Simply put, the overflow flag will be set based on certain circumstances in a way that has no apparent use, however has remained included because it was part of the older specification and was kept in case any old programs depended on this functionality. It is necessary to keep this behaviour because the user should also be aware that this flag will be modified, such that the programs they develop inside VMCS will behave the same outside.

Thinking ahead

A series of testcases will be developed pre-development. These will define intended behaviour in a systematic and computable way, such that a testcase can be tested against developmental builds of the project to find any logic errors that have appeared before updates are released. This will allow for a more stable and consistent release of versions, as new versions will not be released if they do not meet the testcase requirements. These testcases will be based on results from real hardware along with the official x86-64 specification. This will ensure that the fundamental principles of the program, to emulate x86-64 specification with 1:1 accuracy, are consistently met.

A utility library will be developed to work independently from the program, providing general solutions to problems that reoccur throughout the solution. This will save development time as a problem will never be solved more it needs to, and updates to the method will be seen program-wide, thus will not need to be tested and implemented on a case-by-case basis. To identify these recurring problems, a stepwise refinement will be used to break down the larger problem into subproblems.

Thinking procedurally

The order of procedures will be very important throughout the program. Steps must follow a sequential order that is equivalent to that of a processor. For example, consider the JMP instruction. The bytes of a JMP instruction lay out like so, [E9] [*4 bytes of relative displacement immediate operand*]. It will be important that the instruction pointer to calculate the displacement is fetched after the operand has been fetched. This is because the relative displacement is calculated once all of the bytes related to that instruction are read, not after the opcode bytes are fetched. If this were not done, relative jumps would all be 4 bytes off the actual address that should be jumped to, which would not only cause unexpected behaviour but introduce bad habits for users, as they would grow to understand that this is how the instruction works.

Thinking logically

Decisions will have to be made throughout the program. For example, the emulation of conditional codes for operands. These allow the assembly developer to test for the presence of certain flags, and can be tagged on to certain opcodes. This is the case for jump operands, where the JMP opcode is a jump without any condition, but the JZ opcode will only take a jump if the zero flag is set, hence “Jump Zero”. This is a necessary feature to include in the scope as it will allow the user to also include decisions in their programs, which is a fundamental aspect of assembly and programming in general.

Thinking concurrently

Concurrency will be necessary when computing resource intensive tasks. This is necessary because in the .NET framework, the entry thread is also the so-called UI thread that handles the drawing of the window and form controls. This means that if a resource intensive operation is performed on this thread, the user will experience lag when interacting with the application interface. This could cause freezes with the program not responding and an overall unpleasant experience when using the application. By moving the execution of instructions to another thread, this problem can be avoided. This is the most necessary aspect of the program to move to another thread as it will take the most computation power. Whilst these instructions are very fast at an assembly level, they will not be nearly as fast due to them not being translated 1:1 by the compiler, which is understandable due to the nature of how memory is managed by the .NET framework, however, at this point is far out of my control so will be mitigated through the use of concurrency. After these complex operations are complete, callbacks can be used to report back to the UI thread and display results. This will be most useful when coding the testing system discussed earlier, where many programs will

be tested against the program to ensure the accurate functionality of all features. This will be discussed thoroughly in later sections.

Conclusion

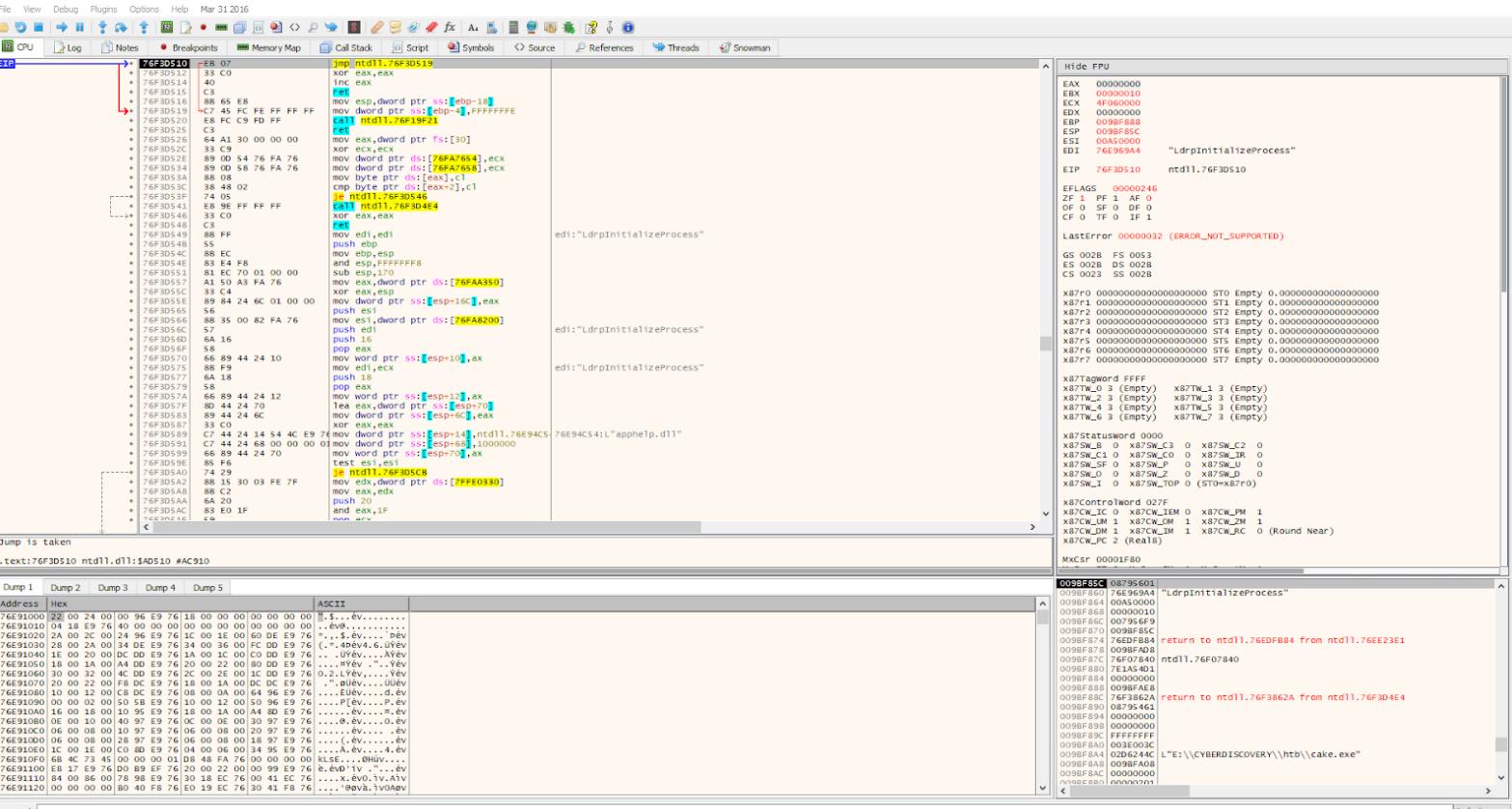
The previous subsections have demonstrated that the computational approach is most appropriate for the problem, and how computational methods can be applied to problems identified in early development stages. This will allow for the problem to be divided much further and simplified through abstraction in the design stage of development.

Research

The existing tools that are similar to VMCS are debuggers. The difference being that debuggers execute the instructions as-is; there is no virtualisation in play. This has advantages and disadvantages which have already been discussed, however, most stakeholders will likely be using this software, or would use this software if they were to learn assembly. For this reason, the best approach to research will be to compare to this software and evaluate what features were executed well and which would not be included in VMCS. The following will be a summary of independent research of the existing tools, that will then be discussed in detail afterwards.

X64DBG

An existing windows debugger, X64DBG is a processor level debugger. I.e opcodes are executed as they are in the binary. This is not convenient for my stakeholders, would have to revert to documentation that is designed for seasoned assembly programmers, not the high level programmers of which I would describe the stakeholders, however does provide exact 1:1 results of how a processor would behave with that instruction. Each processor has very slight changes from the specification, whereas my program intends to emulate the specification word for word where possible, so there may be minor deviations in what happens in my program and what happens on the user's processor due to processors sometimes acting differently to the general specification.



- Very information heavy. Lots of unnecessary information is displayed to the user. This is useful when the user is performing an advanced task, but is only overwhelming to a new user
 - Can be confusing seeing so much information displayed at once

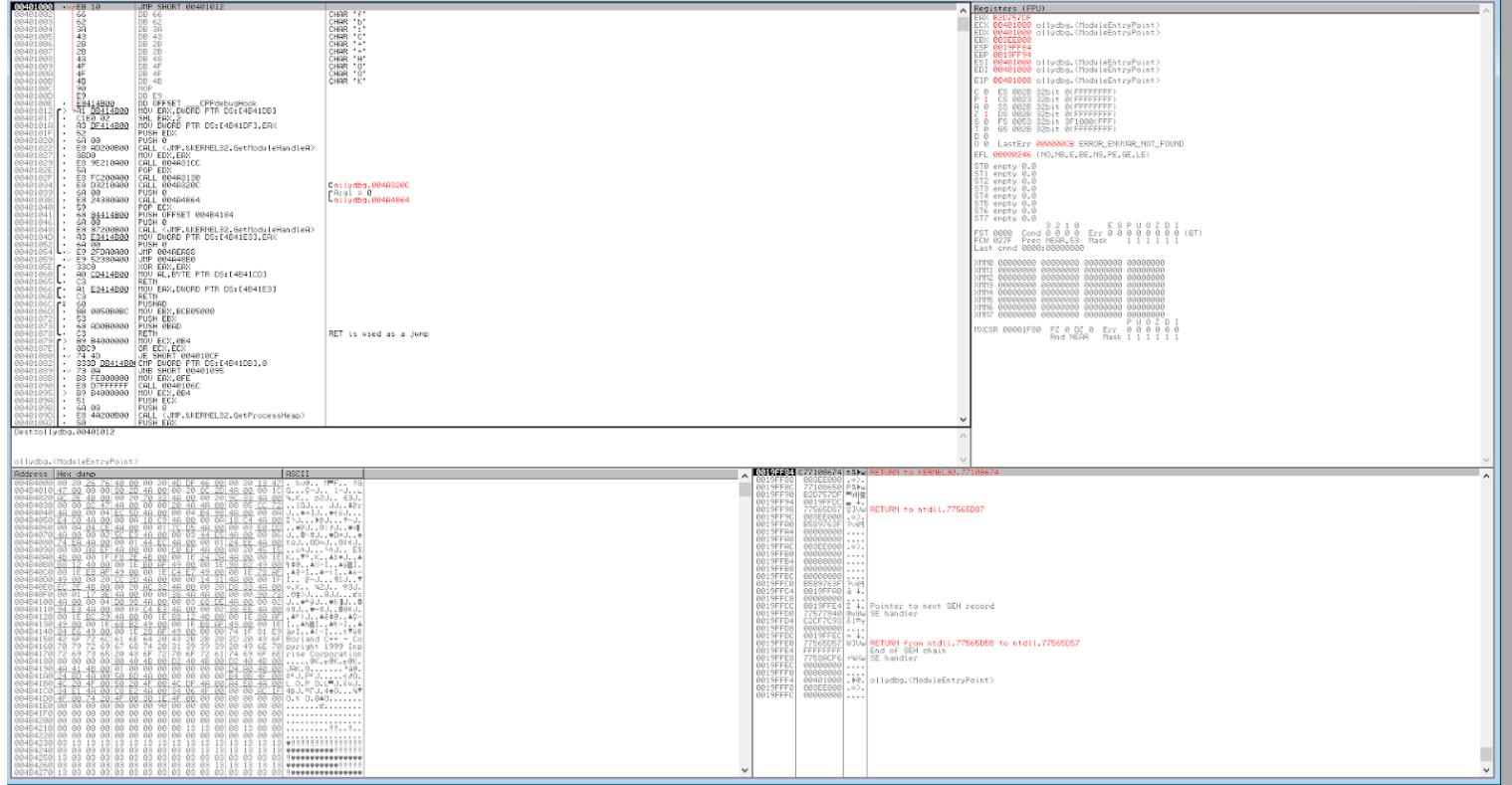
- Not user friendly to an inexperienced debugger at all. The stakeholders of VMCS would be lost using such a program.
- Does not demonstrate principles of assembly, rather just controls the flow of execution and memory. Very useful for debugging, as is its purpose, but is neither educational nor intuitive.
- Does not provide an easy way for users to load their own code. Code must be compiled into a .exe or it cannot be loaded.
- Modularity varies. It is very easy to create a script, slightly harder to create plugins, and virtually impossible to edit the source directly. Scripts are limited in their capabilities, but can perform simple repetitive tasks. Plugins require the user to be familiar with the API which the developers of X64DBG refer to as “templates”. There are lots of supported languages for writing plugins. Changing the core functionality is almost impossible without a thorough understanding of the rest of the source code. It is also very specific to the users hardware. My program will be able to have instruction sets swapped easily.
- Not virtualised. Lots of restrictions imposed on the developer, not necessarily by the program but due to the nature of executing on hardware. Memory must be allocated, which is not something a new and inexperienced assembly developer should have to deal with.

GDB

- GDB has very advanced usage through the command line. VMCS will use a GUI to be more accessible to the stakeholders, they will not have to remember commands.
- Requires GNU/Linux. Users will have to use dual boot or a virtual machine to use. An end goal of my program would be to encourage the users to move on to linux and continue using GDB there, but they must be fully equipped before committing. It would be tragic for a user to rush straight into linux, not know what they are doing, and then hold a stigma against using GNU/Linux because of a bad experience. A user learning assembly should not have to install a new operating system to do so.
- Very modular, can be changed entirely to the users needs through plugins in python, however the source code is in C which is not an object orientated language. The stakeholders of my project will find it hard to understand.
- Like x64dbg, is not virtualised. Same argument.
- Very cross compatible, can be cross compiled for different architectures, but still requires processors of said architecture to execute on.
- Can load a variety of file formats, generalised as “object code”. This includes ELF files, which are linux equivalent of .exe files, and unlinked object code generated by nasm. Still is inconvenient to generate code. Requires command line knowledge.

OllyDbg

- Less information than x64dbg, but still ample.
- Cannot debug 64bit code or executables
- Not user friendly or intuitive, requires lots of debugging/assembly experience to interpret the user interface.
- Not fully open source, only the command line version is.
- No longer supported, very out of date. Windows XP style form
- A degree of modularity. Users can create plugins which extend functionality, however the API given is very limiting to the features that plugins could provide.



Entry point of main module

- Not virtualised. Must run on the same architecture that the program was compiled for, cannot be modified to support other architectures.
- Aimed at very advanced users as icons are used instead of text, which makes it harder to navigate the interface.

Evaluation of features and drawbacks of the researched programs

"Not a virtualised environment, and so introduce processor specific anomalies that are not part of the specification"
Suitable approach: VMCS will follow the specification by letter of the law.

The best approach to solving this problem will be to base tests on results from execution on a processor, as this will reduce logical error compared to creating tests by what I would expect the result to be after reading the specification. This would cover any misinterpretations I had, however, introduces the possibility of processor model-specific behaviours that are not part of the specification. Potential cases for this in the specification are labelled as "X is undefined", for example, "The carry flag is undefined". A common misinterpretation of this would be that the carry flag is ignored; this is not the case. Undefined implies that it should be ignored, but may vary per model. It would be bad practice to base code on undefined behaviour, hence will not be included in the project. My processor that will be used to base these testcases was released after the May 2019 release of the specification, and so in theory should be up to date with the latest behaviour. An example of something that would split through quite easily would be a flag being set when it should not. To ensure this does not happen, the tests will be cross referenced with the specification.

"Not cross compatible with other architectures"

Suitable approach: VMCS will allow for the easy extensibility of opcodes and for the mapping of opcodes to be changed to allow similar register based architectures to be implemented into the project.

This is the best approach as the primary aim of the solution is to provide 1:1 emulation of the x86-64 specification, although the standard functionality of a processor will still be present. Naturally, some architectures will not be in this case applicable, where that architecture has great differences in how it operates, such as a CISC design compared to a RISC design. To enable this, the best approach will be to allow the extension and replacement of opcode tables. This would allow the user to change the opcode mappings in the solution to match the mappings of the architecture

they intend to emulate. This approach will maximise the ratio of development to demand met; there will be little demand for other architectures, but will be even smaller a task to provide an interface for the user to implement their own.

"Aimed exclusively at advanced users"

Suitable approach: VMCS will have a simple and intuitive UI that displays only information that it is necessary for the user to know.

This problem mostly involves the amount of unnecessary information that the user is given, that a beginner to assembly would not understand. This is the best approach to solving this common problem, as the people who need this information can simply use the already existing debuggers; such user is not a stakeholder of this project, whilst the people who do not need this information will be able to use VMCS to accomplish their goals easier. To evaluate which information will be important to the user, there will be a question present in the upcoming survey to the views of the stakeholders.

"Inconvenient for loading user code, all require a degree of compilation"

Suitable approach: VMCS will parse raw assembled instructions, they do not have to be linked, so long as they are in machine code.

This countermeasure has multiple advantages. The first is that convention of input and process of assembling can be abstracted from the program. For example, whether the user prefers NASM, GCC, GAS, or standard intel syntax, it does not matter. They can choose their own assembler to suit their needs, they will not be forced to program in the convention they do not like. Secondly, the development process will be much faster for the user. Running extra commands, downloading extra software, having to have multiple windows open to compile their code into an executable; it all adds up to an unwieldy amount of effort to develop and debug a program. Therefore, this is the best option, because if the VMCS does not simplify this process, it is not doing its due diligence to provide the easy and accessible assembly development solution it aims to provide.

"Hard for users to learn from code written on complex lower level languages such as C and CPP. Also extremely large source repositories can be overwhelming."

Suitable approach: VMCS, hence the name, will be written in C sharp and make use of its own code libraries to reduce the overall code size.

C# is a very common high level language that uses syntax common across many other high level languages. It also has a similar grammar to other C languages, so understanding the source will develop the users intuition of these variants. This will be the best approach because it would be hypocritical to propose an introduction to low level languages, by coding in a low level language. Out of high level languages, C# is the one I am most familiar, and one I feel is best suited to the job because I know there are specific language specifics that I will be able to take advantage of to simplify the solution, such as interfaces. C# is about as simple as a high level language can get, therefore is the best approach. The project is aimed to introduce a competent high level programmer to low level computing, not an inexperienced programmer.

"Researched solutions all had some agree of modularity, which allowed the users and community to create plugins to extend functionality"

Suitable approach: VMCS will include many different object orientated design principles as part of its success criteria, which includes making use of constructs such as interfaces, inheritance, and polymorphism, which will allow users to develop their own code that can be used in place of existing modules

This is the best approach because instead of enabling a small set of features as part of a plugin api, the community developers will have full access to the code, such that the functionality that can be extended is not limited in this way. It also means that an api would not have to be maintained independently of the solution, such that more time can be spent developing the essential features that the stakeholders want. The time invested in further developing these features could negate the need for a plugin that would implement this functionality.

Stakeholders

Summary

Ideal stakeholders for the project are university students aged 18-21 who have a background in high level/object orientated languages, so have a reasonable amount of competence in programming, and need a good introduction to low level language that isn't extreme(Low level introductions tend to be trivial or overly complicated, VMCS aims to provide a happy medium) and allows them to use their existing knowledge to understand how the concepts are applied in the high level code and graphical environment they are more familiar with to help them with a module they are taking in their course, or as general interest.

User developer – An existing high level programmer.

How they will make use of the solution

The user will modify the code and continue to develop the project in their own ways after reading and understanding the code. the goal of the program is to educate competent high level programmers in lower languages. In doing so, they will read through the source code and see how high level concepts can be used to describe low level concepts. Any user who does not have the program open alongside the source code does not fall into this category. They will make adjustments to the code, add new opcodes that they have learned about, reimplement algorithms more effectively, whilst being able to test and experiment code in a highly relaxed environment, there are no restrictions on code. Memory does not have to be paged or segmented, i.e no segmentation faults. Users can create modules of unimplemented assembly features. It meets their needs because it is in a familiar high level language, rather than diving straight into assembly code. I know this works because I was the first test subject. By going through the process myself, I now consider myself proficient in assembly code from being confident in only C#. The source code will contain ample resources for doing so that I have written myself and referenced useful information.

How the solution is appropriate to their needs

Existing resources for assembly are few and far between. Where found, they are mostly at a very advanced level. This is even further split by the differences in syntax styles and architectures. One example is the difference between GAS syntax and Intel syntax. This is the most counter-intuitive convention in assembly. GAS syntax, along with a few other differences, has disassembled instructions in the form

OPCODE SOURCE DESTINATION

whereas Intel and other syntaxes have the form,

OPCODE DESTINATION SOURCE

Consider the following

GAS: MOV %EAX, %ECX

Intel: MOV ECX, EAX

These are both identical operations, they would have the same machine code, but the disassembly output is different. This is very learner unfriendly, as some sources may use GAS syntax, some may use intel. This is rarely established by the source, there are a few key methods of identifying GAS syntax, but the user would not know these either. Therefore to be appropriate to the users needs, all information will be provided in intel syntax and kept simple but still thorough. This is important because the user shouldn't have to use other sources until they are at a point where they have enough of an understanding to interpret the other sources.

Existing assembly developers who use windows.

How they will make use of the solution

A user who is successful in using my program will be able to go on and use these advanced tools with enough intuition to understand how to use them. This user will be most dependent on the source code because they already understand how assembly works at a syntactic perspective, they understand assembly from the theoretical perspective. For example, how instructions are translated to machine code, how operands are encoded. Ultimately, they will be able to apply new techniques to their assembly code that can only be achieved by a deeper understanding. For example, rather than "MOV EAX, 0x0" they will understand the impact and inefficiency of this length instruction and more efficient ways of programming to the same result, such as using "XOR EAX,EAX" to clear the register. This information will be provided through comments in the source and documentation, and most importantly, experimentation. It will be much simpler for them to execute their code; they need not worry about assembling. In this respect, they will be more able to try new algorithms and techniques out to see how they can optimise their programs whilst yielding the same result.

How the solution is appropriate to their needs

Windows does not have many existing tools for assembly programming. The user friendly application can bridge the gap between the “deep end” in linux; most users will not yet be committed enough to change operating system for the sake of better tools. By familiarising themselves with the nature of x86-64 assembly, they will be ready to move on to advanced tools that operate in a stricter environment such as GDB. This transition will be easier because they first used the more productive GUI provided by VMCS, such that certain procedures such as linking their object code was removed from the development procedure, hence they can dedicate their time to understanding the underlying concepts of assembly rather than only the syntax and procedures to compile code.

Students looking to study computational methods of arithmetic and bitwise operations.

How they will make use of the solution

This user will be able to analyse the highly annotated and explained source code and understand how to implement macro scale operations on large byte arrays in order to achieve results that are not normally obtainable through high level arithmetic, or to develop an understanding of how addition and other arithmetic procedures can be solved through computation. They will also be able to see the effects of said operations in action, using it and applying it to algorithms written in assembly. This user will learn about more effective ways of using assembly opcodes to achieve certain goals. For example, instead of multiplying by two, shifting by one yields the same result but is easier for a computer to shift than to multiply, hence faster. This user will also understand how flags are affected by operations and how they can be used to perform operations on a greater scale, such as adding two 128 bit numbers, which cannot be done with a single operation.

How the solution is appropriate to their needs

Computational solutions to these arithmetic and bitwise operations are usually very complex; something that would be studied at university. A simple browser search shows that the majority of results are from university lectures (This will be analysed in detail in a later section), which are far too complex for the stakeholder in question. By providing annotated and simple solutions, potentially at the cost of performance, will provide a starting point for the user to establish intuition for the procedure, and after that will be able to understand the more complex adjustments that can be made to improve performance; that which was covered in the university level lectures. Another aspect would be flags. Flags are a highly undocumented aspect of x86-64 assembly. The intel documentation will say, for example, “CF is set according to the result”. Some websites give a slightly more useful answer, but even so can vary between architectures. Contrary to this, there will be a thorough explanation of when and why a flag is set and what can be done with the result, such that users can further their understanding of how these algorithms can be implemented into their own solutions. This is important because without understanding, they will only be able to provide examples of existing code, which then they will not be able to adapt to a given scenario or optimise.

Programmers interested in developing algorithms

How they will make use of the solution

Programmers can apply abstract models of algorithms and machines into a virtualised environment. Using the assembly language will make their solution more portable and demonstrative; a line of code on a high level could execute hundreds of instructions, and so the ability to implement faster and unique solutions will never be found this way. This user will have less need to analyse the source code, rather the practicality it can offer.

How the solution is appropriate to their needs

An existing assembly level debugger provides too many unnecessary details to this user. They are only concerned about the results of their program, which will be provided through flags, registers, and memory. This idea will be explored further in the research section. It will also abstract any platform specific details. For example, if I go to debug in GDB, there will already be all the environment variables on the stack, there will be kernel code at the bottom of memory, linker code at the entry point. These details are only an obstacle to this user, they have no need to worry about them, hence with them being abstracted from the VMCS design, they will see a clearer picture of the results produced by their algorithm; it will not be confused with data that already existed in memory. Another detail abstracted is the need for memory segmentation. When programming in a low level language, memory has to be allocated before it can be used by the program. This is a very platform dependent procedure, there is no general method for doing so. At an assembly level, this normally involves putting certain parameters in the registers then using the syscall opcode to perform a kernel interrupt. This is a push factor, it is not an exciting part of assembly that encourages the user; nobody wakes up wanting to segment memory. This is also perfectly valid in terms of the aims of the program. No part of the x86-64 specification details anything about specific syscall procedures as they are entirely different for each

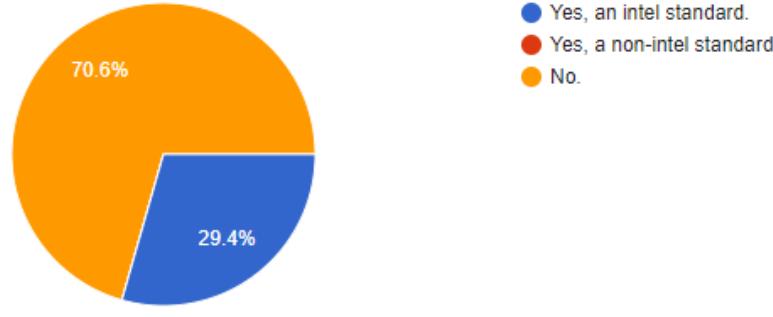
operating system, and can even be manually changed in linux. Therefore it is necessary to remove this obstacle to ensure the user can reach their goals faster through the means of VMCS than an alternative.

Stakeholder feedback

The following section will review and discuss the feedback of the analysis development stage from various stakeholders that I have chosen to participate in a short survey provided by Google Forms. Following sections will be based on this data.

Have you ever used the assembly language

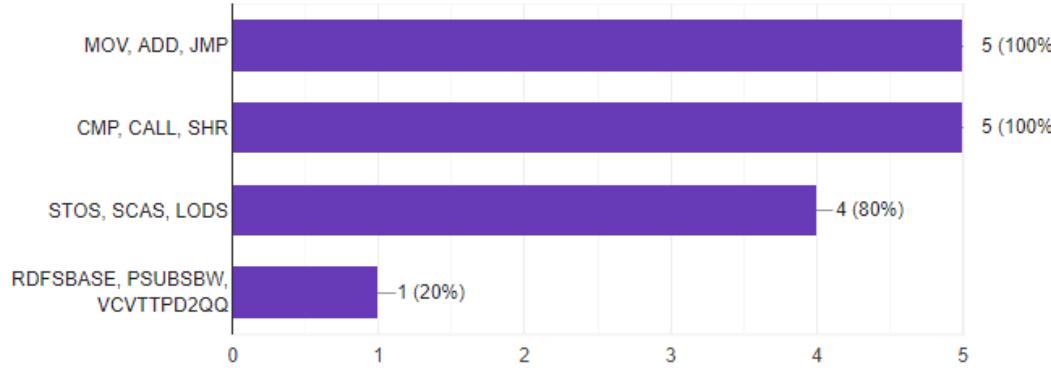
17 responses



The survey was conducted among a group of associates who I consider to fall into the various mentioned stakeholder classifications. As the data shows, a total of 17 took part in the survey. This included a mix of those who are familiar with assembly and those who are not. I felt this was important to see the viewpoints of people who have less to learn from the program and will have a more constructed evaluation on the practicality.

If so, which of the following opcodes are you familiar with?

5 responses

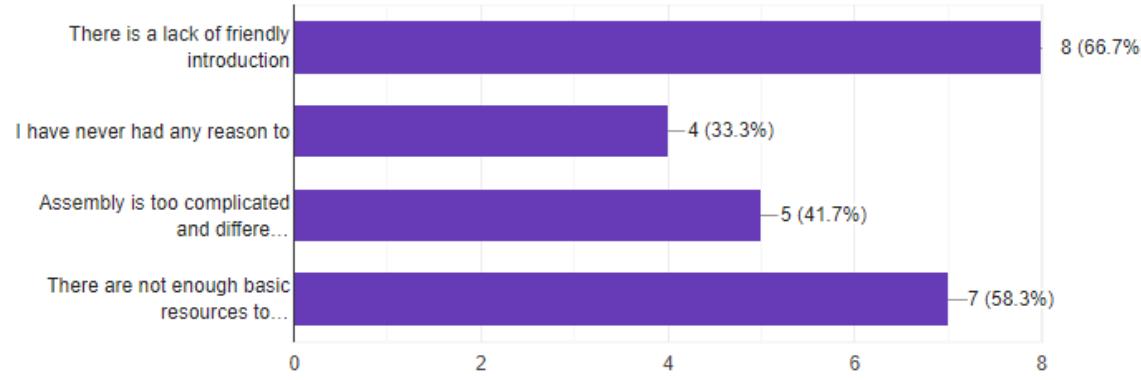


This figure shows the distribution of needs that existing programmers may have. This question and options were designed to escalate in complexity to give a bearing on the scope of opcodes that should be included in the project. This will be useful in determining where time needs to be spent most and answer questions such as "Do the stakeholders need the full instruction set, or would already be advanced enough to use alternative programs at this point?". Every programmer made use of the top two basic sets. These would have likely been included regardless, but goes to demonstrate the importance of including the basic opcodes, as it is shown statistically that no existing assembly programmer would be able to make full use of their knowledge without having these included. The second point comes to the string operations(STOS, SCAS, LODS), which are a more specialised set. As 80% of the participants were familiar with these, I consider this a large enough proportion. This could also be seen as an instance of the Pareto principle, that although the string operations are a small subset of available operands and are not exactly common assembly fundamentals, a great amount of stakeholder demand could be met by including them at

cost of a lesser amount of development time. Lastly come the relatively advanced packed integer operations and the use of developer-controlled segmentation. These would take a large amount of developer time to include in the project and only be used by 20% of the stakeholders that knew assembly.

If not so, why have you not(check all that apply)?

12 responses



The figure shows the attitudes of stakeholders that have not learned assembly, towards learning assembly. Please note that although clipped in the image, the following options were provided, and that participants could choose multiple options.

- There is a lack of friendly introduction
- I have never had any reason to
- Assembly is too complicated and different from what I am used to
- There are not enough basic resources to learn from

VMCS primarily aims to solve stakeholders that would choose option #1 and/or option #4 however options #2 and #3 are still open to be encouraged. Fortunately, the majority chose #1 and #2, suggesting that the selection of stakeholders that were interested in learning assembly was an accurate choice, and perhaps had option #1 or #4 selected as well as #2 or #3. Option #2 was designed specifically to determine what proportion of stakeholders would not be interested in learning assembly and would likely not make use of the project after development. The needs represented in options #3 and #4 will be met through documentation and source code commenting as well as simple code principles that will be discussed in later sections. This is the best approach as the stakeholders will be able to see the code in action in a high-level context that they are more familiar with. Option #1 will also likely be met through this nature as existing assembly resources are almost entirely university lecture level; a simple search query will demonstrate.

[All](#)[Shopping](#)[Images](#)[Videos](#)[News](#)[More](#)[Settings](#) [Tools](#)

About 86,600,000 results (0.47 seconds)

The **add instruction** adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location. The **inc instruction** increments the contents of its operand by one. The **dec instruction** decrements the contents of its operand by one. 19 Nov 2018

University lecture/paper

Guide to x86 Assembly

<https://www.cs.virginia.edu/~evans/guides>[About Featured Snippets](#) [Feedback](#)

Assembly - Arithmetic Instructions - TutorialsPoint

https://www.tutorialspoint.com/assembly_programming/assembly_arith.htm
Assembly - Arithmetic Instructions - The **INC instruction** is used for incrementing an operand by one ... The **ADD and SUB instructions** have the following syntax -

I find this relatively simple, but I have found this website to be a very unreliable source for assembly, there is a great lack of explanation and more statements, like a paraphrased version of the official manual

ADD: Add (x86 Instruction Set Reference)

https://c9x.me/html/file_module_x86_id_5.htm

x86 assembly tutorials, x86 opcode reference, programming, pastebin with syntax highlighting.

... The **ADD instruction** performs integer addition. It evaluates the ...

HTML version of the manual

Guide to x86 Assembly

<https://www.cs.virginia.edu/~evans/guides>

Jump to **Instructions** - The **add instruction** adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location. The **inc instruction** increments the contents of its operand by one. The **dec instruction** decrements the contents of its operand by one.

ARM not intel

Assembler User Guide: ADD - Keil

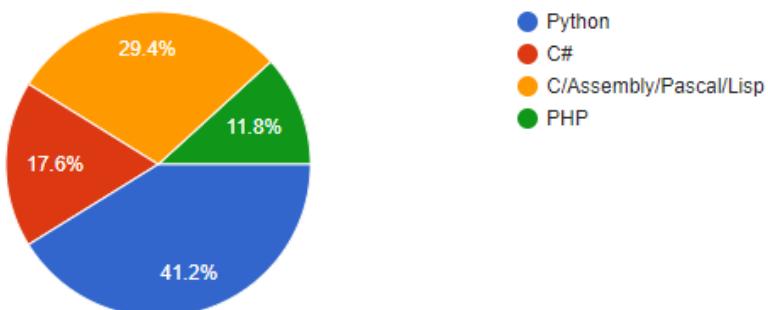
www.keil.com/home/arm_and-thumb-instructions

Operation The **ADD instruction** adds the values in Rn and Operand2 or imm12. In certain circumstances, the assembler can substitute one **instruction** for another ...

The figure explains itself. One of the first page search results was not intel standard(a new programmer searching for answers will likely not know the difference, and would then be misled into an entirely different grammar). Another was what I found to not be a very useful source of information, there are multiple posts on forums that support this claim. One was a straight rip from the manual, this is about as advanced as it can get, and lastly a repeated result of a university level course, which was the entire lecture to end from 2006. A very informative resource that I have read myself, but by no means a beginner level. This supports the claim that there is a demand for more introductory resources that is not being met.

Which of the following is the programming language you are most confident in?

17 responses

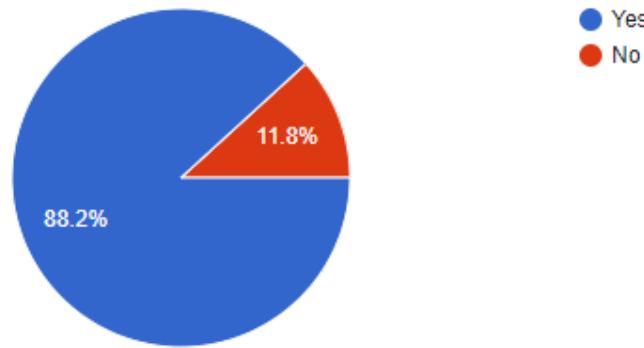


This figure demonstrates the distribution of programming skills across the stakeholders. The main take-away is that the majority of stakeholders are high-level programmers, therefore a significant amount of time should be dedicated to ensuring that the code is simple and uses high-level object orientated concepts such as classes and interfaces such that the high level programmers who are most likely to make use of the source code can do so as easily as possible.

The existing assembly programmers will also likely have less need for reading the source code as they will likely already be familiar with the included opcodes, as shown earlier.

Please download x64DBG and OllyDBG as attached in the email, or install GDB if using linux. If not possible, tick "No" on the following.

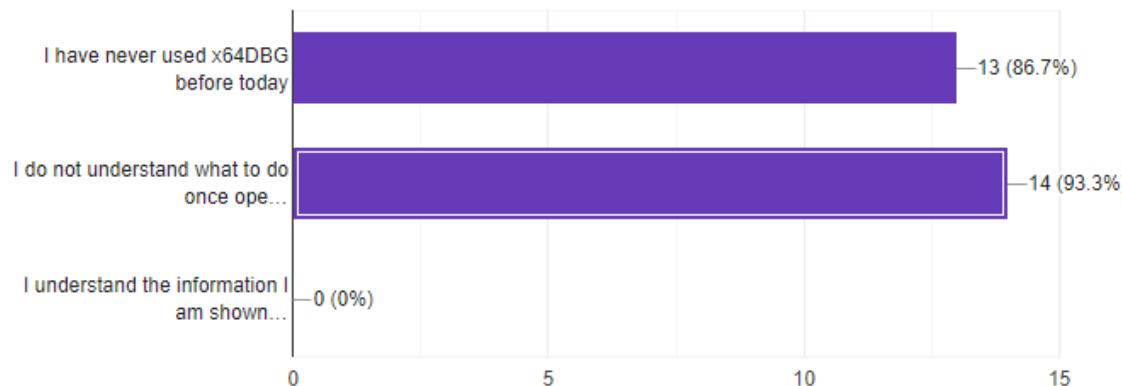
17 responses



The following figures will be more aimed to see impressions and opinions on existing alternative software. Fortunately, the vast majority were able to take part in this part of the survey, which will keep the results reliable.

After opening X64DBG, please tick as appropriate

15 responses



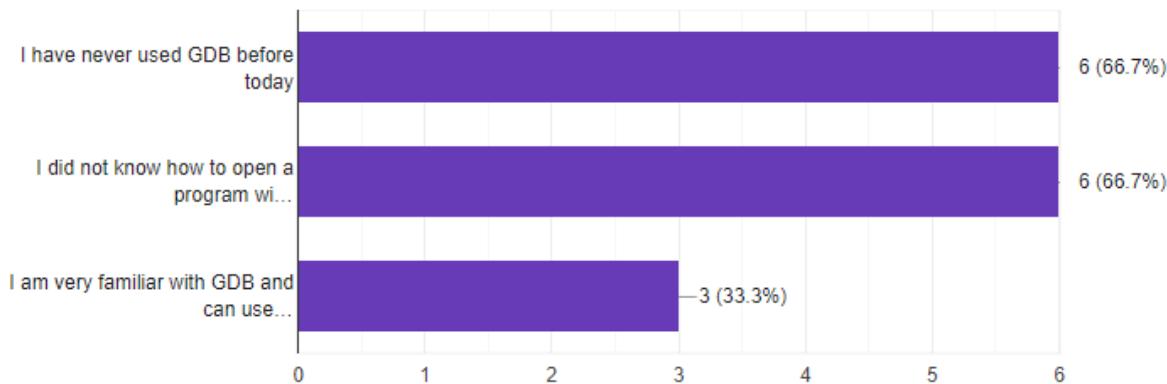
Please note that although clipped in the picture, the options were,

- I have never used x64DBG before today
- I do not understand what to do once opening
- I understand the information I am shown fully and find the layout natural

As shown in the figure, most of the stakeholders had not yet used x64DBG. This shows that most of the stakeholders were never exposed or attracted to such debugging software. If so would imply that there is a need for a more user friendly program to serve as an introductory level software as opposed to the advanced program x64DBG as predicted earlier. This is supported by the fact that even participants that had used x64DBG before, could not recall how to use it afterwards, which supports the idea that a more intuitive design with labels rather than icons is needed. This figure also shows that no user was fluent in usage of x64DBG. This suggests that they had either moved on to alternative software or used alternatives in the first place.

After running GDB, please tick as appropriate

9 responses



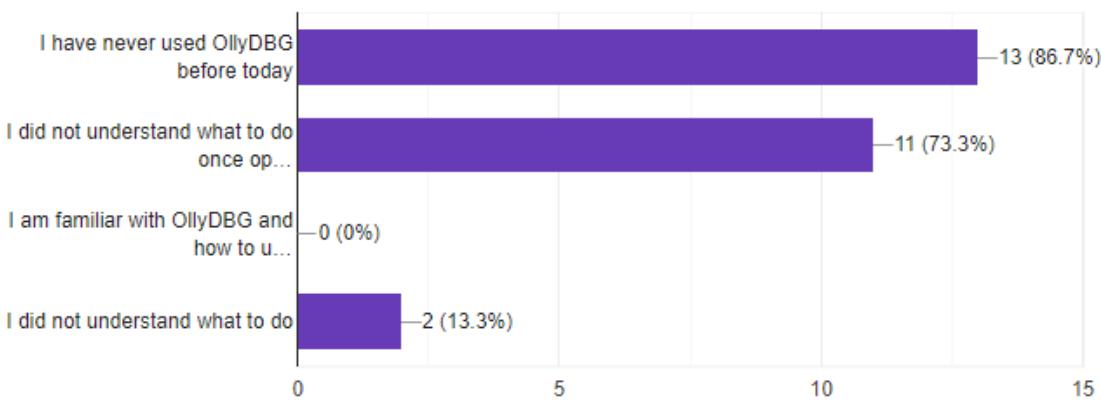
Please note that although clipped in the picture, the options were,

- I have never used GDB before today
- I did not know how to open a program without opening the man page
- I am very familiar with GDB and can use with ease.

This figure shows an interesting distribution of GDB users. Most users were either very familiar with GDB or had no idea at all. This suggests that as opposed to x64DBG, the GDB users stuck with GDB instead of moving on to alternatives. Some further research may be required here into the features that these users found most appealing in order to meet some of the more specific stakeholder demands.

After opening OllyDBG, please tick as appropriate

15 responses



Please note that option #4 and option #2 are the same, but I renamed #4 to #2 after a few participants had already taken the survey, and so were kept separate from the renamed version, and although clipped in the picture, the options were,

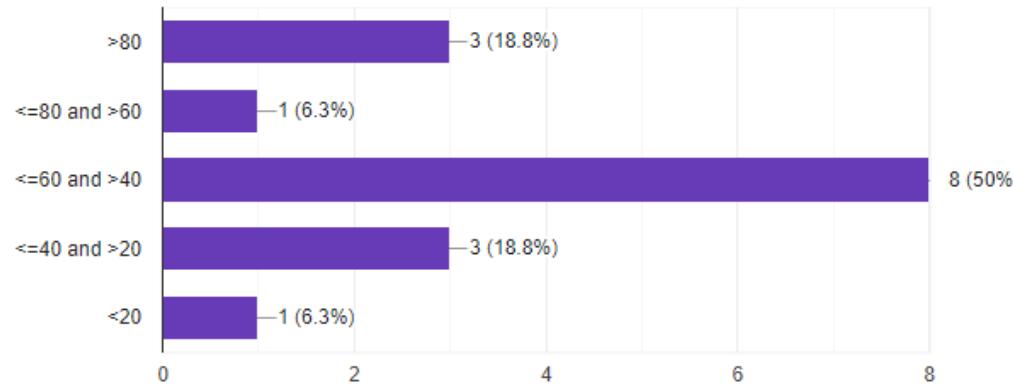
- I have never used OllyDBG before today
- I did not understand what to do once opening the program
- I am familiar with OllyDBG and how to use it, or picked up fast
- I did not understand what to do

This figure shows similar results to x64DBG. Most of the stakeholders never took the opportunity to use this software. I believe this is because these types of software describe themselves strictly as "debuggers", which is a lot less representative of what they can do. There is also reputation that they are solely used for cracking programs, which is

not true at all. To avoid these negative connotations, VMCS will be strictly marketed as an emulator. There is also the common theme of over complicated user interface design. There is little else left to say on this front except from that it will be greatly prioritised, as research has shown it is a very determining factor of whether stakeholders find use in the program or not.

(Windows only)Please head to userbenchmark.com, install the application and follow instructions there. Record your "Desktop" percentage score here.

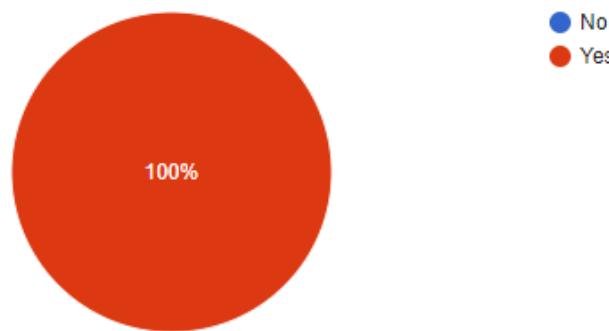
16 responses



This figure shows there is an almost normal distribution of hardware standards. The anomaly at 80-60 is likely because of the number of participants. This shows that there should be little to worry about optimisation-wise when programming. Naturally, good code implies good performance, but more specifically there is no foreseen need to spend time optimising, it will likely be possible to determine the required complexities when designing the code structure. It is also important that the algorithms are simple such that they are easier to interpret and maintain in the future. At any rate, this question demonstrates that the stakeholders do have reasonable computers that should be able to run a non-resource intensive program. In relation to userbenchmark.com specifically, 50% is the global average score, however it is important to note bias that the vast majority of people who benchmark their computer will be people with expensive gaming computers. For reference, <https://www.userbenchmark.com/UserRun/21469703>, a 60% score had a latest gen intel core i7 and an Nvidia GTX 1050-Ti, which are certainly more than what the program will need to run.

If presented an opportunity, would you be interested in learning more about low level operations of computers?

17 responses



Finally, the survey shows the all participants would be interested in learning assembly. This shows contrast to earlier, where there were more negative attitudes present. Should this option been present to them at that time, it would have likely seen a different result. Exposure to the alternative software that has been discussed shows that people are interested in the software, but need an introductory software to ease them in. There is too much commitment involved in reading 5000 page intel manuals(Not an exaggeration) or university lectures especially for programmers that

already have jobs, or as stated earlier, see currently no specific need to understand assembly. This was a good idea to include in the research because now I can conclude that there is definitely a demand for VMCS, and many advantages and disadvantages of alternative software shown through research can then be considered in further stages of analysis. Purely researching on my own provided good insight for how I will approach the problem, but stakeholder specific feedback will allow my program to do what others cannot.

Essential features

A control unit that emulates the process of fetching and decoding instructions.

Identification

The control unit will handle tasks specific to the bytes of instructions, such as converting them into opcodes. It will handle the grammar of the x86-64 architecture, the order of prefixes, and other conventions in the language. It will be responsible for calling opcodes and providing any parameters they require. It will also give them an interface to modify components such as memory. This is necessary for consistency throughout the program, the rules of execution will be defined here.

Explanation

This is an essential feature because having these input rules defined in one place will make it certain that there are no deviations or inconsistencies in the program. Also, it will be very convenient to maintain the program at a later date if something is changed in the specification, hence needs to be changed in the program. Having this procedure centralised in the CU ensures that the changes will be seen in every instance of the scenario; definition outside of the CU could be forgotten about at a later date or not obvious to a future maintainer.

Disassembler

Identification

A built-in disassembler to disassemble the instructions entered by the user. This will have its own class and provide disassembly for a given stream of instructions.

Explanation

This is an essential feature because the user will want to see which instructions are being executed at a given time, which instructions have already been executed etcetera. It would be very inconvenient to the user if they could not see what was happening in execution in a human readable format. Machine code would not suffice here as they are a lot harder to remember; there is no operation that can be inferred by a human from a byte, where as "MOV" is simpler to infer the operation "to move". If this feature were not provided, the user would likely use other means to execute their program, such as a command line, because there would be no useful information provided to the user by the program.

Graphical user interface

Identification

A GUI will display the outputs of the program to the user in a simple format, where details that the user does not need to worry about can be abstracted, whilst still being present in the program source code if the user does need to see them.

Explanation

This GUI was shown to be an essential feature because it was shown in the research section that similar software did not include a simple GUI. The GUIs provided were very advanced and displayed many details that would not be relevant to the majority of the stakeholders identified for this project. As the stakeholder feedback showed, a very small percentage of users actual made use of assembly instructions that would affect this information, therefore the GUI can be narrowed down to display only the essential details such as memory, registers, and flags.

Stateless opcodes

Identification

Stateless opcode that can be called at any time and carry out their set of procedures. Specifically, the procedures will have no necessary preconditions; they will take only inputs and then output data where applicable. Moreover, the opcodes will not depend on the previous instructions executed. Naturally if the programmer has coded wrong they will not have their desired result. The opcodes will be easy to program with the interfaces provided and can be swapped out or created easily.

Explanation

This is an essential feature because the user may want to extend functionality, adding their own opcodes, or opcodes that were not covered in the scope of the project. They could also remove and modify opcodes freely, if no opcode depends on another, there will be nothing preventing this. It will also provide more accurate emulation; The processor does not consider the order of instructions, it only works with the data it has been given. Consider the “RET” instruction. One might first assume that CALL has to be executed before RET, however, RET only pops an address off the stack and jumps to it(The address that *should* have been pushed on by the CALL opcode). In this sense, developing with dependence on order may be able produce more mathematically correct results, but would not be aligned with the primary aim of the project, 1:1 emulation.

Opcode map that the Control Unit decodes instructions from.

Identification

This opcode map will provide a centralised view of all opcodes throughout the solution. Having these in one place will make it easy to update parameters to opcodes, as will likely be required as the program develops; only a small set of features will be available initially, the necessary preconditions and inputs to the opcode may change as this feature set is expanded.

Explanation

This is an essential feature as it was shown to be a factor that was not considered by similar software, as shown in the research section. Having the map will allow the user to swap the existing opcode mappings to match a different architecture. This may also be useful for the stakeholder who is developing a microprocessor architecture, and can model this in the virtualised environment. Intel x86-64 fundamentally is a great standard for CISC instruction sets of register machines, so should easily be adaptable to virtualise a desired architecture that has a similar grammar to x86-64. They would be able to replace opcodes with their own definitions, such as an opcode class provided through a new module. This will also allow users to insert their own opcodes for experimentation, such that they do not interfere with existing opcodes; they can use a spot on the map that is not currently mapped to an opcode.

It will also provide a good visualisation of the differences between opcodes and their variants. For example, there is a specific mov instruction to move an immediate value for each general purpose register, which is done by Intel to save the memory used by arguably the most common instruction. This was hard to describe with words, but consider a visualisation similar to what the user would see in the program,

BYTE OPCODE

0xB8, Mov(Destination:Eax, Source:Immediate)
0xB9, Mov(Destination:Ecx, Source Immediate)
0xB9, Mov(Destination:Edx, Source Immediate)

This would demonstrate much more clearly the purpose of these 16 consecutive MOV opcodes on the opcode map (Two for each register, one specific setting the lower byte of that register). This is also a lot more intuitive than how it is defined in the intel manual(2B, 4-35),

B8+ rd id | MOV r32, imm32 | OI | Move imm32 to r32.

These kind of details are important facts for every assembly programmer, including all of the stakeholders, to understand. An identical mnemonic does not imply identical behaviour. A certain opcode of MOV will even automatically sign extend a dword to a qword, but still be disassembled as MOV(0xC7).

Self-testing mechanism

Identification

A self-testing mechanism to test known correct behaviour against the current revision of the code, covering all aspects of the solution. This will take common input format such as XML, such that the users/maintainers can create their own testcases, or modifying existing.

Explanation

Being able to test operations easily will encourage users to modify the code and adjust it to their personal needs. This mechanism will make it clear what has gone wrong in the testcases; this will be necessary because a undescriptive error will be no more useful than having no testing mechanism. To do this, the output will also be in XML format and demonstrate what went wrong by visually comparing the known good to the result of the test. For example, the output could look like,

```
<Test name="Add opcode">
  <Expected>"RAX=0x10","RBX=0x102232324"</Found>
  <Found>"RAX=0x5","RBX=0x102232324"</Found>
```

</Test>

This would allow future maintainers or users to then parse this output, or view it visually; some may prefer a visualisation. Many programs exist that parse XML into a more readable format, so formatting it would not be essential. By showing clearly where the operation has gone wrong, the user/maintainer will be able to quickly debug their code where they made the modification. For example, if they a new bitwise addition algorithm was implemented, then all of the testcases for opcodes that involved addition failed, it would be clear that the new code had a bug. It will also help narrow down the bug by testing many different data sizes; some bugs may only be apparent like this. A good example of this would be a carry in an addition algorithm. If a carry was being set incorrectly, but the only testcase added 4(100) and 3(011), the bug would go unnoticed; the testcase would in this sense even provide a false sense of trust. For this reason, it is not only essential that a self testing mechanism is included, but a self testing mechanism that covers all aspects of the solution.

Intermediary layer

Identification

An intermediary layer between high layer development and low layer development to provide useful data structures and routines through an API to communicate between layers and abstract complex tasks, such that the layers need not concern themselves with the specific procedures that have to take place to process data and details specific to a layer. This kind of operation will be handled by the intermediary layer. If a future maintainer wants to improve the user interface, they will not be required to know about assembly, only about the code they intend to modify. This will be done made possible through means of an “intermediary layer”; an abstraction of outputs from the low level “processing layer” where instructions are executed into a simple format that removes any unnecessary details that the user interface programmer would not need to worry about. It will also parse and validate the inputs from the user interface into a format that can be understood by the processing layer through the use of data structures.

Explanation

This will be essential to users who have are modifying the code. A user would not be modifying the code to change the entire functionality of the program; they would likely want to change a small section or routine to extend existing functionality. For example, if the user-developer implements a new opcode class, then there is a runtime error about how the interface error could not interpret the input, this would become a chore. This is where the intermediary layer could be used to instead change the new code to match the delegate types and data structures that the interface layer uses. This may sound limiting, but consider the nature of the program. There are very few concepts that would require a drastic change such that the delegate types would no longer suffice. There needs to be some kind of memory, registers, and flags for it to be of assembly nature. These kind of fundamentals could be easily modified and reimplemented by the user, but would unlikely need to be removed by the user. This also works both ways. For example, a user-developer wants to implement a new file format to be parsed by the program. They can write the input validation and parsing routines for the format in the interface layer to match to produce a result that outputs a “memory” object containing the instructions of the file, such that the object matches the delegate that the intermediary layer requires. This could be as simple as a byte array. Then, the intermediary layer will perform the procedures and routines to solve the preconditions and extra details that the processing layer requires. In this way, the user was able to only parse and validate a file without having to have any knowledge of how the processing or intermediary layer functioned. This pipelining concept is comparable to the OSI model. When working with the application layer for example, all that concerns the developer is the inputs and outputs of the presentation layer. As a further result, maintainability will be improved. A developer who is good only with windows forms will be able to offer a good input and improvements to the program, they will not have to understand the whole nine yards. It would be cumbersome to have to read the entire source before being able to offer relevant advice. A simple delegate or data structure is more simpler to understand, as the details of the source are abstracted to a selection of variables.

IO handler

Identification

An IO handler will parse and validate the user inputs to the program, specifically the loading of executable files. This will allow users to load their own code from a variety of file formats.

Explanation

The IO handler will allow the user to interact with the program without having to modify the source code. This will allow for more reliable results when inputting data, as validation routines will be in place to parse and validate the input. This

is important user input has many places to go wrong for programming in general. By having pretested routines to perform this task, it will be clear to the user whether the code is incorrect or their code is not in the correct format. Every user of the program will have cause to load instructions; it is the main functionality of the program, therefore it is essential that there is a dedicated approach to ensuring robustness.

This will also improve future maintenance as the underlying code can easily be modified without affecting the ability to load programs. The parser and validation routines will format the input into a general data structure that can be applied to all input formats. This could just be a byte array holding the instructions. If the method of loading these instructions is changed, the output byte array can be changed to match this delegate; the parsing and validation routines do not have to be changed.

Logging

Identification

A logging system to generalise exception and event handling throughout the program. It will support a variety of methods, such as displaying a message box to the user and writing to a file. This will be used by all layers of the program, as concerns is necessary meta information the user-developer will need to see for bug reporting, or even finding issues in their own code.

Explanation

This will be essential to improving the communication barrier between maintainers and users when submitting bug reports. This is because a general exception may not be clear, such as "ArrayOutOfBoundsException". This would not be very conclusive; there will be hundreds of arrays throughout the solution. This will be solved by providing the logging system that specifically provides error codes for each operation where the program is most likely to fail. For example, if there is an invalid data when validating user input, this would have a dedicated log code for that input. The routine called immediately after that which used the validated inputs would not need one, because the validation routine has already validated that data. If a user need only submit an error code and some added context, the procedure of reporting and fixing bugs will be much more streamlined.

It is essential that the logged event is both displayed to the user by a message box and written to an external file. This is because the user is likely to just click an error away when it pops up; they want to get back to what they are doing. By logging this to an external file, if the error caused further disruption afterwards, the user will still be able to report the issue. If the message box popped up and there was no significant erroneous behaviour afterwards, the user can move on. This could be the case if the user inputs an invalid file, they can amend the file and then try again. An error in the validation routine could be as simple as this, or could be parsed wrong. If the file was parsed wrong such that the user could not use a working file, in this case would the logging system be most useful to them as after closing the message box, they can fetch the details from the log file if necessary.

Limitations

Identification: The opcode map will only be covered to a reasonable scope

Explanation

Research will be done to determine which opcodes will be used by the majority of the users. This will be done in the form of a survey. A scope will be covered such that the most common features of the x86-64 assembly language can be demonstrated that the user would likely see in an executable. Opcodes will be continued to be added for as long as time permits.

Justification

This is a necessary limitation as to code every opcode would consume an excessive amount of time. This would in turn significantly affect the other areas of the program as less time would be able to be dedicated to them. This limitation is not severe because by the time that the user has explored and understood all of the implemented opcodes, they will be ready to make use of the existing advanced debuggers that provide a similar functionality to VMCS but at a very advanced level that was not accessible to them when they first used VMCS. This is the best approach because the majority of time should be spent developing the features that the stakeholders are going to use.

Identification: The solution will not include features of alternative software if the only purpose is that it is a useful feature.

Explanation

Features of other software may be included if they provide a foreseeable benefit to the aims of VMCS. A feature that would not be included would be a feature that provides great debugging functionality, but at a very advanced level.

The user will learn from the basic operations such as breakpoints and stepping to develop understanding before moving on to user advanced alternative software.

Justification

The primary purpose of VMCS is to provide high level programmers with an introduction to low level programming in a context they are familiar with. VMCS feature-wise will not be aimed at existing users of alternative software who are familiar with it. The features for these users will be in the source code, as in alternative software it was shown that the process of decoding instructions and other processor level operations was not explained. This is the best approach as it saves development time to focus on the features that will directly benefit the majority of stakeholders rather than one specific demand.

Identification: The solution will not include a built in assembler

Explanation:

Users will have to use another program to assembly instructions from written format into machine code. From the machine code format, more general solutions can be applied to parsing the instructions.

Justification

There are many different standards and conventions of x86-64 assembly. Almost every assembler will have differing syntax to the next. The differences could be small, or as large as having instructions written out entirely differently(GAS vs Intel). By not forcing a convention on the user, they will be able to choose whichever syntax they are most comfortable. This is a very subjective choice by the user and bears no impact on the produced assembly code, therefore VMCS has no place to enforce convention here.

Identification: The solution will not include methods of loading windows .exe executables.

Explanation

The solution will provide methods for loading the instructions of ELF files(linux equivalent of a windows exe), BIN files, and TXT files. TXT files will be files that have machine code written in string format, whereas BIN files are written directly in bytes/machine code format. Windows executables will not be included in the parser.

Justification

Very few windows assemblers exist; most are written for GNU/Linux. Some tools for linux will directly assemble and link a given set of instructions into an ELF file, whereas others will assemble the code into pure object code. These both would be covered by the ELF and BIN parsing procedures. On windows, all of the assemblers I have found all output to the described BIN format, where each instruction is written to a file exactly how it is written in machine code; there are no additional file headers that need to be considered. For this reason, a .EXE parsing procedure is not necessary, linux only needs special consideration because of the inconsistency between available assemblers.

Another available solution for windows users is to make use of web assemblers. Defuse.ca hosts a public web assembler that uses GCC syntax and outputs instructions in text format. This text can then be copy pasted into a .TXT file which can be read by the TXT parser.

Solution Requirements

Hardware

Specification: Monitor

Purpose: Display program windows graphically. Will not support a console based/remote shell system.

Justification

The project will use a graphical user interface to display information, as determined in the research section. Without a monitor, it would not be possible to display useful information that the solution aims to provide. In this sense, if the user has no monitor, the program would be as useful as executing the assembly program on a machine(outside of the virtualised environment).

Specification: Mouse

Purpose: User interaction will be based on mouse clicks. There will be no command line.

Justification

The program will solely use the GUI for interaction. It was determined in the research section that the programs which used keyboard interaction through commands were too complex; the user had to commit time to learning the different commands to function the program. Instead, equivalent functionality will be provided through buttons, hence the user will need a mouse to click them.

Specification: Secondary storage with at least 1MB of space available

Purpose: Data files will need to be stored on the user's system, such as testcase files.

Justification

Only a small amount of memory will be required to store the program. This is because the drawing will be done programmatically; there will be no pictures or icons stored. This requirement mostly concerns the need for the user to be able to use the secondary storage. An offline approach to the solution is better than a web based design because it does not depend on the service being always available, web hosting costs money whereas websites such as github will host the repository for free, then once downloaded can be stored forever. The only drawback to this is a user may not be able to download/use the executable in school as schools usually block executables, however they could use their own laptop.

Specification: A relatively modern 64-bit intel/AMD processor

Purpose: ULONG and LONG data types are needed, which make use of 8 byte registers, and so will be needed to emulate the 8 byte registers used in x86-64.

Justification

.NET functionality varies greatly across architectures. For example, if .NET code is executed on a big endian processor, `BitConverter.GetBytes()` will return bytes in big endian. This will not function in the program as it will exclusively work with little endian. This is the best approach because there would be a large amount of work required to work around this and the amount of processors using big endian that can run windows are almost nonexistent, and even less actually do. Big endian systems are mostly IoT devices or industrial machines, hence are not identified as potential stakeholders of the project. The processor must be 64 bit because the "long" data type will be used very often as the aim of the project is to emulate x86-64 architecture which uses QWORD pointers and registers; if the target is not capable of executing these instructions in the first place, they will not be able to emulate them.

Software

Specification: .NET Framework 4.8 minimum

Purpose: Windows forms, the UI system used, requires .NET framework.

Justification

The program will be written in C# and use the .NET Framework to include useful libraries that will save time in the development process. For example, the windows forms libraries are part of the .NET framework and will be used to draw the user interface.

Version 4.8 is the latest version at the time of writing and will be necessary to take advantage of the latest features and optimisations provided by this version, rather than compiling for an old version. It will also be important for life-cycle of the program. After a few years, Microsoft drops support for a specific version. This means that if there is a bug in the framework, or is not ported to the future versions of windows, users will be unable to use the code; they may not have the means to do so either.

Specification: Windows

Purpose: .NET framework is supported on windows only

Justification

As the program will use the .NET framework, the user will require windows. This is different to .NET Core, which is cross platform but does not provide the full feature set, nor windows forms, which as discussed will be very useful during development. Another reason was found during research, users who had GNU/Linux were generally technically advanced enough to use GDB, which has been actively developed since 1986, thus these users will be making use of the advanced features provided. Windows is the platform that the vast majority of the stakeholders will be using, therefore would be better to put development efforts into where the stakeholders will see the benefits.

Success Criteria

Summary

- The program should be able to accurately emulate a x86-64 processor from an outside perspective.
- The program should be able to execute instructions in a timely manner.
- The executable program and source code should be a useful learning tool for the stakeholders to advance their knowledge of assembly
- User interface must be intuitive.
- User interface must not be excessive
- User interface interaction will be kept minimal
- Core features will be easily accessible through the UI
- User interface will follow the Material.io material dark theme design specification
- Program will be able to load ELF files as executables
- Program will be able to interpret ASCII text into instructions
- Program will be able to display disassembly for an input program
- Program will have an testcase system that can be expanded by adding files to a directory
- Program will allow user to add breakpoints to instructions

Assembly

Criterion: The program should be able to accurately emulate a x86-64 processor from an outside perspective.

Justification

This is of great importance as the user needs to be able to take their programs and run them on other intel processors and get the exact same result. Through research, it was shown that this was an absolute given when using the alternative software, therefore stakeholders would be much more inclined to use the alternatives if they provided correct results. If users were to be misinformed, that is catastrophic. For this reason, it is essential that this criteria is met. However, the final statement bears some significance. A method more suitable to a high level language will be favoured over a low level procedure, so long as the outcome is the same. This will be discussed further in the design section.

Measurability

The testcase system will be used to measure this success; it will be used throughout development to make sure that the reliable behaviour produced from the specification and executing the instructions outside of a virtualised environment, is met with every release.

Criterion: The program should be able to execute instructions in a timely manner.

Justification

This will be of value to the stakeholders as all users should be able to have a grasp at how performant their program/algorithm is. Small scale programs will have a similar execution time as executed on hardware. It would only be misleading if the program executed a particular algorithm faster than an alternative which is faster when executed outside of the project. For this reason, it will be important to give a rough relative time relation between two pieces of code.

Measurability

This criteria will be measured through performance testing after development. To do this, the tools provided in visual studio will be used to measure how long certain procedures take. A threshold of 50ms per opcode will suffice as the user is most likely to be stepping through their program at a speed much slower than this.

Criterion: The executable program and source code should be a useful learning tool for the stakeholders.

Justification

The primary purpose of the program is to educate users and provide a smooth transition from high level programming to low level programming. This was something that was shown to be lacking from the similar software in research; complex low level procedures would be performed, but the purpose or procedure not explained. Most of the education content will lie in the source, which will establish intuition and understanding of the low level concepts. This will then allow the users to experiment in the main program. Therefore, this criteria is essential because it ensures that said primary purpose will be met.

Measurability

This criteria will be measured by the ability for existing programmers who do not know assembly to learn. To assess this, I will exhibit some class files to my classmates, then test them afterwards on the content. To keep this a reliable measure I will use a concept that they are very unlikely to have heard of before.

UI

Criterion: User interface must be intuitive.

Justification

This was a problem in similar programs found through research; the interface is too advanced for the stakeholders. This would be overwhelming to a programmer who has likely never seen a low level debugging interface before and act as a knowledge barrier. In particular, the alternative software such as OllyDbg would use icons. This was confusing as it was unclear what each button does and its purpose. This criteria is required to ensure this does not happen in VMCS.

Measurability

When planning UI designs, I will test my friends who I would consider potential stakeholders by first explaining what the program does (The user who had just downloaded the program would have downloaded it with an idea of what it does from the readme file), then task them with a basic procedure that a new user would be likely to try, such as loading a program and executing it to end; a task that should be easily guided by the UI. The number of users who completed the task will be taken into a percentage. A percentage greater than 80% will be considered a great success.

Criterion: User interface must not be excessive

Justification

This problem found in similar programs, too much information was displayed at once, most of which would be irrelevant to most situations. This would confuse a new user, if they start researching the advanced details that were displayed instead of the basics, they will not develop a good understanding of assembly and most likely move on because of complexity.

Measurability

This can be measured by showing my classmates a picture of the interface where information is displayed, then surveying them on how much of it they understood. I will do this after a reasonable amount of the A level course has been taught such that they will be better programmers at that time, hence will have a similar amount of knowledge to the potential stakeholders, but not too much that it is obvious to them.

Criterion: User interface interaction will be kept minimal

Justification

Many complex procedures can be automated for the user, such that they need not understand certain details. For example, a single file dialog will load a file, they do not have to specify the file type. To do this, magic bytes will be used to identify the file. A new user may not know what an ELF file is, for example a Windows user, but the program will still be kept accessible to the stakeholders by instead using extra procedures to automate the loading files. This approach will be taken to all other UI components.

Measurability

This criterion ensures a UI design that prioritises quality of life over functionality. Because of this, detailed user feedback will be required. I will present the interface to potential stakeholders selected from friends of mine and ask them to write a short paragraph about their thoughts on how much effort was required to perform simple tasks such as loading a file. If successful, the majority of feedback will be positive, stating that the UI in some way benefitted the development process.

Criterion: Core features will be easily accessible through the UI

Justification

User interface will include a simple way to use core features, whilst making any advanced features discrete. As shown in the research, it was sometimes hard to find basic functions such as breakpoints and stepping. This will contribute to the intuitivity of the UI; the user would not be able to navigate the program easily if the features they use often are hidden behind multiple menus.

Measurability

This criterion can be measured systematically by measuring the number of clicks required to reach a core feature. Core features will be determined by analysing which elements of the user interface are used most often. This could be done with an interface heatmap gathered by allowing users to trial a modified version of the software where each button click is recorded. This will be considered a success if core features are found in two clicks or less.

Criterion User interface will follow the Material.io material dark theme design specification

Justification

Programs researched all had a very professional style environment gray and white form design. This made the programs feel very tense and serious. Following the material dark theme will provide a more friendly atmosphere.

Measurability

The success of this criteria will be measured post-development by surveying users about how they feel about the design. This will be considered a success if the users were at minimum contented. It would be considered unsuccessful if users say that they would prefer a more serious style.

Criterion: Program will be able to load ELF files as executables

Justification

ELF files are the most common executable format for developing assembly programs. Most existing tools use ELF files. By supporting this file type, the users who are new to assembly will be introduced to the standard file types, as opposed to creating my own, which could be counterproductive. It also allows the user to debug existing programs.

Measurability

Tests will be conducted whilst this feature is added. They will include factors such as how the files are validated, and whether the input is parsed correctly and additional robustness tests such as how the program behaves when the ELF file has an abnormal configuration.

Criterion: Program will be able to interpret ASCII text into instructions

Justification

It was shown in research that alternative software require a compiled executable as an input. However, it is often found more convenient for the stakeholders to copy and paste text on screen rather than go through the process of compiling a binary, especially as they may not know how to use other tools to do so.

Measurability

This criterion will be measured by testing the feature directly. It will be considered successful if the feature works correctly with normal input, but how erroneous input is handled, e.g. meaningless text.

Criterion: Program will be able to display disassembly for an input program

Justification

It is important that the user will be able to see their code that is being executed and debugged. This will also allow for further interactions required such as clicking on instructions to set a breakpoint. By converting it into assembly rather than leaving in machine code, it will also be easier for the user to understand what the input code does.

Measurability

Each individual opcode will be written into a program, with a variation of parameters to ensure that the disassembly works for all kinds of instructions.

Criterion: Program will have an testcase system that can be expanded by adding files to a directory

Justification

It will be important for the future of the program if users and maintainers can add testcases to test their own code, and to test the functionality of the program. It will make the testing process easier as tests do not have to be directly added into the code, or performed manually in most cases.

Measurability

The testcases will use XML format. The parsers for this will be tested rigorously by testing not only valid inputs, but invalid XML files, and other erroneous file types.

Criterion: Program will allow user to add breakpoints to instructions

Justification

It is important for a debugger to allow the user to have a method of breakpoints. This will allow them to find logical errors in their code, and to see the effects of individual instructions or a particular sequence.

Measurability

Breakpoints will be configured by clicking on the desired address in the disassembly viewer. This will toggle the breakpoint. This success criterion will be successful if the breakpoint is added correctly, and responds correctly to spam clicks and other erroneous behaviour.

Additional considerations

Criterion: Complex low level concepts will be abstracted into simple data structures and delegates

Justification

The intention is that stakeholders will be able to extend functionality to meet any specific needs, such as an opcode they want to learn and understand. The best way to do this would be for them to develop the said opcodes themselves. To encourage this, the process should be made as simple as possible, providing them useful programming interfaces to interact with the complex code without having to understand every part of the solution. An example of this would be providing methods for an opcode classes to fetch operands; a repetitive yet complex procedure as it will involve reading memory, reading registers—a significant amount of tasks would need to be understood. However, by providing a method to perform this procedurally, the process itself does not have to be understood. Researched alternatives all provided an acceptable form of API that allowed for extension of functionality. This proved to be a very popular feature. For example, Pwndbg is a GDB plugin that added more commands and changes the layout of information. VMCS will provide a similar feature, except rather than through plugins, the user-developer will extend the code directly, then recompile the source. This saves time as a separate API to what I will use during development does not have to be developed and maintained alongside, and also gives more control to the users over what behaviour they can change.

Measurability

After development, I will measure this by tasking friends of mine, whom I consider to fall into the category of user-developers, with designing their own simple opcode, not necessarily to meet the specification, that is not the purpose here, rather to see how simple it is for them to adapt to the programming interface. After ten minutes, I will tell

them to stop and assess how many had met the goal they set to achieve based on how complete their code was and whether they were easily able to incorporate the data structures into their code.

Criterion: Single responsibility principle: Every module, class, and function will be designed to fit a specific requirement. A class should only have one reason to change[2]

Justification

Designing components to solve one problem rather than many will ensure robustness. After being tested, they should be able to be left alone until their requirement changes, and have reliable outputs given valid inputs. They should not need to be changed after development of that component has finished, rather the class requires the change will adapt to the input format that the callee requires, unless the callee is being changed to suit another purpose. This will be essential to not only improve maintainability, but also applies to the user's experience reading the code; They should only have to read the part of the code that they wish to understand, not the whole program.

Measurability

This will be measured by the ability of the user to read through an arbitrary class. I will test this on a selection of stakeholders by asking them to read a given class then ask how much they understood that code and may ask a few questions about it. This will be successful if the general consensus is that the code is fluent, and they did not need to read large amounts of the source to understand the given class.

Criterion: Open/Closed principle. Software entities should be open for extension but closed for modification[3].

Justification

This will be an important criterion to ensure the finished project is a maintainable solution. By closing the functioning classes for extension, there will be a consistency throughout as this class does not have to be changed again. Moreover, when a new feature is to be added, instead of extending an existing class, a new class will be developed to provide this feature. Because of this, instead of having to adapt all existing code to suit the new modifications, the new functionality will be available separated from what is known to work.

Measurability

This criterion will be measurable through means of independent peer review. I will have the code reviewed by other developers to determine not only whether the Open/Closed principle, but also whether the advantages of this design have been exploited.

Criterion Liskov substitution principle. Subtype Requirement: Let $f(x)$ be a property provable about objects of type T. Then $f(y)$ should be true for objects y of type S where S is a subtype of T.[4]

Justification

Inputs to functions will be generalised and abstracted into a form that is purely based on an interface or inheritable class, not specific to certain types. This base class/interface will define the minimum requirements for the derived type, such that new features can still implement addition functionality whilst being fully compatible with existing functions that accept the base class/interface as an input. It is necessary that interfaces are used throughout the program to ensure interoperability; so long as each has valid outputs, any input can be swapped out and replaced with one that inherits from the same interface and the program will still operate, which will allow for the future extension of functionality without having to modify the existing classes to meet the new functionality.

Measurability

This will be measured by the overall use of polymorphism. It will be considered a success if the principle is not broken; all areas of the code should accept interface or base classes as inputs rather than specific data types.

Criterion: Interface segregation principle. Clients should not be forced to depend upon interfaces that they do not use.[5]

Justification

Superclasses split into subclasses have more specific purposes. These classes will be able to have better tailored methods that are more relevant to them, rather than "fat interfaces"[5]. As the solution develops, the number of features will also scale. It will be important to keep the features separate from one another to allow this scalability. For example, if a new distinct feature is added, instead of adapting it to squeeze into an existing interface, it will have its own interface as it serves a different purpose. This will be more important as the solution scales as otherwise, the "fat interface" would eventually exceed its limit of proportionality where no possible new feature could be adapted to fit into it. Then, every existing feature would need to be reworked into a new interface. Moreover, existing features would otherwise need to be adapted to fit into the "fat interface" as the project scales. If the new methods are added to the interface, every class that implements this interface must have this method, therefore existing features would have to be stretched to fit the new interface in which they were fine before.

Measurability

This criterion can be measured by code review. I will ask several OO programmers whom I know to review my implementation of the principle and ask for written feedback about to what degree they think the criteria was met. Naturally, some cases the ISP will be impractical, as written by the creator of the principle, "ISP acknowledges that there are objects that require noncohesive interfaces"[5].

Criterion: Dependency inversion principle, " High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces). Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions." [2]

Justification

This will be a very important criteria as the solution includes a wide range of programming skills. At the lowest layer of abstraction will be the assembly processing, and at the highest the .NET forms user interface. The dependency inversion principle will be necessary to ensure that the details of the assembly code such as instructions, memory, are all abstracted into simple high level concepts before reaching the higher layers. This will be important to ensure that any developer will be able to contribute to the project or maintain it in some way, as they will only have to be proficient in one field. For example, an assembly programmer will be able to offer advice about the lower layers, without having to know about the workings of the upper layers, only the intermediary abstractions that will be inputs to the layer such as the data structures.

Measurability

This criterion can be measured through peer code review. I will have programmers who I think are particularly good at one area of the program and ask them to read the code without providing the code for the higher layers. The programmer will then feedback and state how much they could understand without seeing the higher layer code. This will be considered a success if the programmers all only had to read the abstractions and had no doubts of which the root cause was because they could not see the upper layer code.

Criterion KISS principle(Keep it stupid simple)

Justification

At all times simple code will be favoured over complex code that accomplishes the same task. As said in a similar principle, "do the simplest thing that could possibly work"[6]. This will ensure that code will be easier to comprehend by a future maintainer or user reading the source. This is necessary as even I myself will forget how code functions after time, by keeping the solution simple, the development process will in turn be simpler and more efficient. This project does not require an amazing degree of performance to meet the needs of the stakeholders, existing solutions already exist for the user that requires this.

Measurability

Code will be heavily commented then reviewed by peers. I will vary which sections are viewed by each person to cover the entire program and to prevent them from becoming overly familiar with a specific section that would cause bias. This is the best approach as seeing how another party views an explanation is the only reasonable method to conclude whether the explanations in comments have been a success; I understand it because I wrote it, but I need to be certain that other people find it simple to pick up on.

Criterion: Calling trees will be pipelined through layers without backflow where possible

Justification

Having a clear call trace whilst debugging will ensure that bugs are easy to track down and resolve. The best approach to doing this is to make sure that program flow follows a strict path in terms of layers of abstraction. For example, if an output from the bottom layer needs to be processed by the intermediary layer, there will be no "backflow"; the intermediary layer will not call methods in the lower layer to parse this information, rather the next function called must be another routine in the intermediary layer or in the interface layer. This means that the breakpoints will be able to narrow down anomalies quickly. If a breakpoint is placed in the intermediary layer and when reached the anomaly is present, it must be certain that the anomaly was from the previous layer. To persist with this design strategy from the beginning will ensure the maintainability of the code in the future.

Measurability

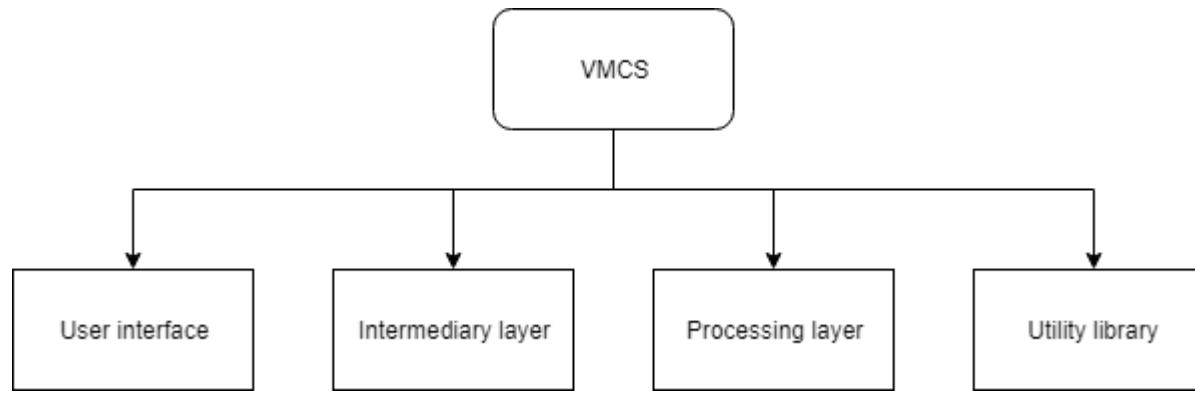
This criterion can be measured through call stack analysis. For core functions where this behaviour would be likely to happen, a breakpoint can be placed at the expected endpoint of the call tree which can be determined by looking through the source. Once the breakpoint is reached, I can analyse the call stack to look for any cases where backflow has happened. This criterion will be considered a success if the number of backflow cases is less than ten throughout the solution. These ten give some flexibility in cases where another design principle would be contradicted, such as

the KISS principle. In these ten allowed cases, each time will be fully documented and explained as to why it is a better approach.

Design

Problem decomposition

A stepwise refinement has been used to decompose the problem hence design the solution. I have taken this approach because it clearly distinguishes the steps that are required to code each module. It has also allowed the project to be separated into four different layers of abstraction, of which each can be developed separately-- This will be discussed further later. The stepwise refinement will assist in identifying where thinking ahead can be used to determine which problems are present in multiple areas of the solution, hence can be separated into a utility library such that the solution may only be coded once. This will not only save development time but make maintenance easier in the future, as if the function needs to be modified again at a later date, the changes will be present in all areas that had the problem initially, and will not need to be maintained on a case-by-case basis. Full stepwise refinement diagrams will be provided in a separate file, but throughout will be referenced



Project can be split into four distinct layers that need to be tackled separately

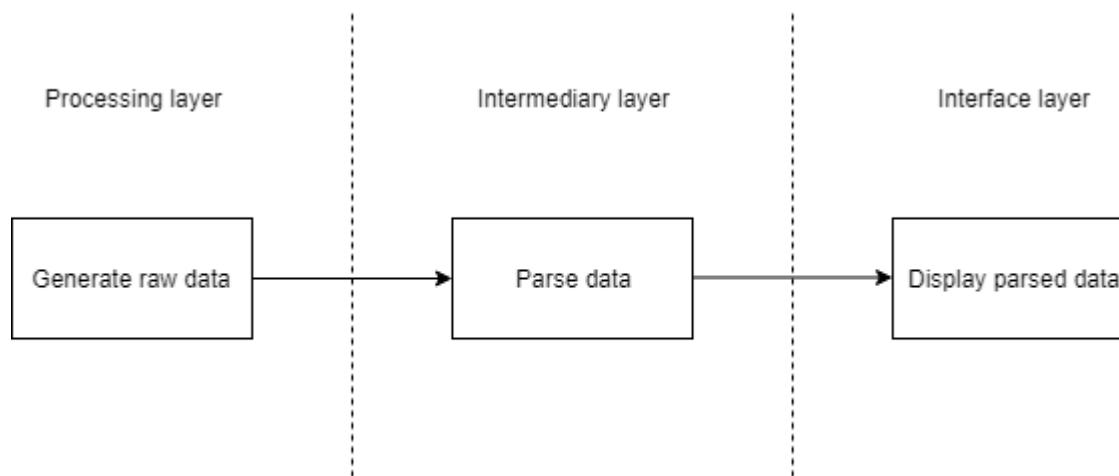
By separating them, it will be easier to maintain and develop the solution. This is because each layer will only be concerned with the inputs and outputs of adjacent layer(this is visualised in the figure), such that so long as the change matches the inputs, the adjacent layer will have no problem.

The exception to this is the utility library, which will provide solutions to problems that reoccur throughout the project or may be usable in future projects. The reoccurring problems will be identified using the stepwise refinement diagram. This will reduce overall workload and testing requirements, as a routine can be fully tested and documented and left in the utility library, such that the process does not have to be repeated when the code is reused.

The layers will also have distinct purposes. The interface layer will focus on usability features. The intermediary layer will focus on maintainability features and data structures that the interface layer can communicate effectively and simply with the processing layer. The processing layer and utility library will heavily focus on algorithms and computational methods.

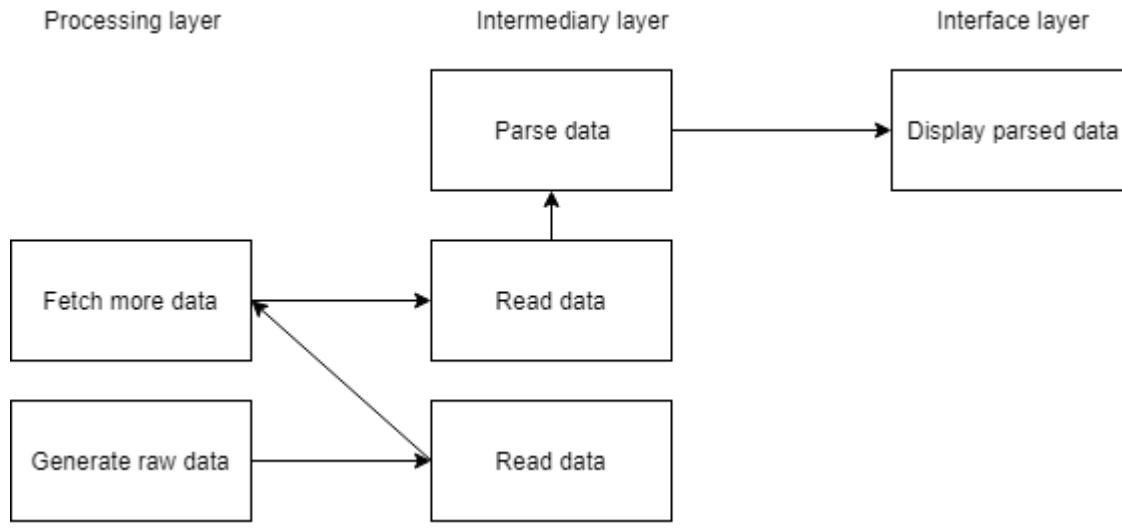
Having layers also improves maintainability and future development. A layer will be able to function with any substitute adjacent layer that has the same input and output format. In some cases, the compatibility will be even more so relaxed, such that only the same type of output is required.

Another factor that improves maintainability is that bugs will be easier to track down. A clear pipeline for an output will derivable, such that an exponentially decreasing number of functions will have to be debugged. For example, if a value is being incorrectly displayed on the screen, the programmer can first check the intermediary layer, is the

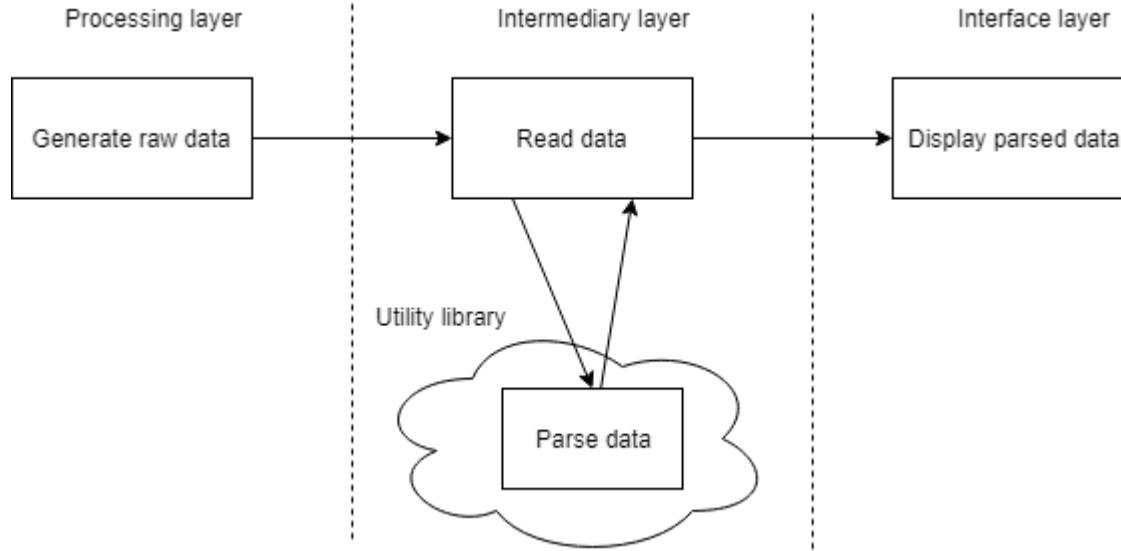


problem present at the pipeline node here? If so check the node in the processing layer that provided the data, is the absurdity present there? If not, the programmer knows the bug was in the intermediary layer. Where possible, nodes will present an ordered chronological situation.

Where each rectangle represents a node, the benefits of the layers system are much more demonstrative in tracking down a bug. As opposed to the following,



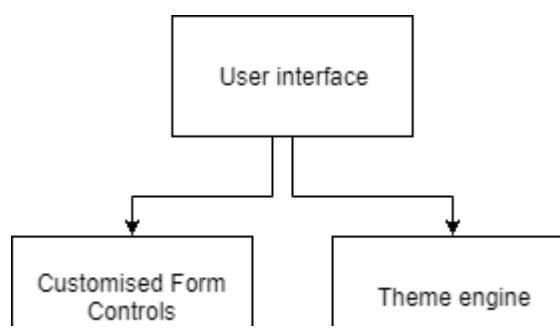
It becomes very incoherent when a pipeline contains backflow, because a layer can no longer be checked like mentioned before, the problem could have occurred in the backflow. However, a more realistic scenario than the first may look like,



This is the primary purpose of the utility library, to hold methods that have been thoroughly tested to justify a “bend in the pipeline” in order to simplify the solution and reuse components.

Due to the size of the program, some irrelevant and trivial parts of the code have not been discussed. For example, when a message box is displayed, an abstraction of a function is used in relevant parts of the pseudocode, such as PRINT(), without defining exactly how the message box has been created and styled, as these are details that are custom to the development process.

Interface layer



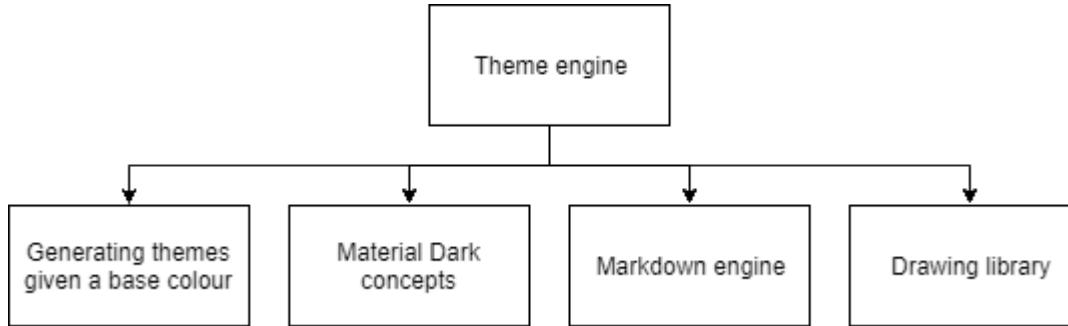
Explanation

The interface layer will provide most of the usability features of the program and handle the drawing of any forms and controls required for the program. It will also cover the procedures of what needs to happen when a button is clicked and will pass any necessary information down through the pipeline to the processing layer to perform actions at the user's request.

Justification

It is necessary to divide the user interface into two separate subproblems because there are two main parts of the interface that need to be considered. One is the drawing of forms and their colours, and the other is the procedures of directing the inputs where they need to go.

Theme engine



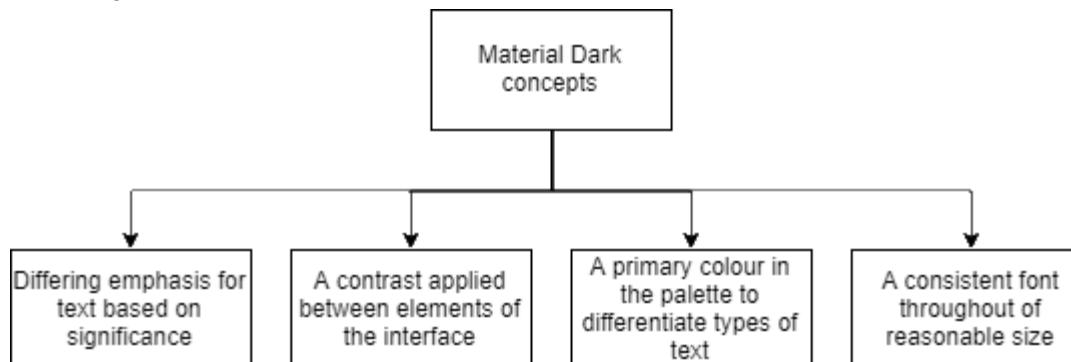
Explanation

The theme engine will generalise the themes of all controls across the program. A theme will be programmatically generated when a colour is given as a parameter by adding predefined offsets onto the colour to produce other colours that can be used as primary and secondary colours for the theme. The default will be dark, but will be customisable. There will also be a separate drawing library for specific procedures required for drawing forms. The markdown engine will apply themes to text based on special characters parsed at run-time.

Justification

It is necessary to split this problem into subproblems because there are certain tasks that can be modularised into a more general solution and others are more specific to the project. For example, the markdown engine will be applicable to any implementation, but the drawing library will draw the controls and forms for this project, so is unlikely to be reused. For this reason, the best approach is to break down the problem and tackle the subproblems to produce self-contained modules that can operate independent of one another.

Usability feature: Material Dark concepts



Explanation

The material dark design theme states 3 key principles.[7]

- Darken with grey: Use dark grey – rather than black – to express elevation and space in an environment with a wider range of depth.
- Color with accents: Apply limited color accents in dark theme UIs, so the majority of space is dedicated to dark surfaces.
- Enhance accessibility: Accommodate regular dark theme users (such as those with low vision), by meeting accessibility color contrast standards.

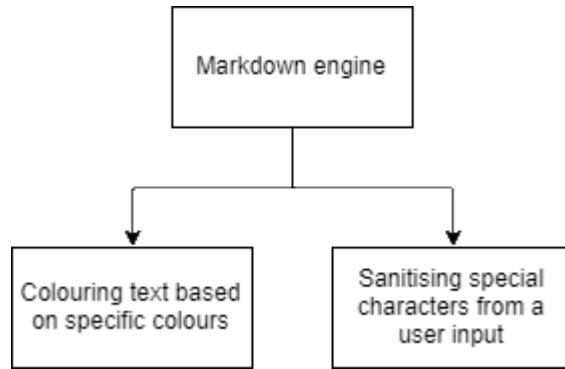
These principles will be followed throughout the theme engine design.

Justification

It is necessary to include this usability feature because it was identified through research that other programs had a white theme that gave a very professional and serious impression. This would not be attractive to a user as the program would not stand out from the rest, therefore it would only attract a limited number of users because they

would see the program as comparable to alternative softwares identified through research(X64DBG, OllyDBG). Another reason is that the light theme is very bright. This is counter-productive for long debugging sessions as staring at bright light for multiple hours is not going to be comfortable during or after the program is used and will be hard to navigate if the user is in a dark room for example, it can be hard to see text written on white in such conditions. For these reasons, the best choice is to include the material dark concepts to make it easier for the user to use the program for long periods of time as is often required when debugging code.

Usability feature: Markdown engine



Explanation

The markdown engine will draw stylised text from an input string. The input string will have special characters that dictate which style will be applied. For example, “!hello!” would draw “hello” in a different colour. The syntax will be similar to a regular expression, where sections are enclosed in a special character. The special characters will apply styles according to the Material Dark concept. This will be implemented as “Emphasis”. Text will be more/less transparent based on its emphasis. In reality, the transparency will dim the text, however programmatically the alpha channel of the colour is being lowered. This means that the default, full opacity, will only be used in circumstances where the text is to be greatly emphasised. The characters will follow a predefined mapping.

- ! : Max emphasis. Text will stand out the most, used to display the most important information
- “ : High emphasis. Text will stand out above normal text. Will be used to display text important information that does not need immediate attention.
- £ : Medium emphasis. Normal text. Will be used to display general messages that may or may not be relevant to the user.
- \$: Low emphasis. De-emphasised text. Will be used to display text that the user will rarely require attention or be changed.
- % : Background text. Text will be the same colour as used to draw certain controls. This will be used to display text that is constant and conveys an obvious meaning therefore would likely be read once or twice by the user.
- ^ : Primary colour. By default will make the text purple. Will be used to make text important text stand out differently from emphasis. For example, user input.

These characters were chosen because they follow the pattern of pressing shift + number on the keyboard, such that lower numbers have higher emphasis.

Justification

It is necessary to include the markdown engine because of the enhanced accessibility material dark principle. It will allow more important information to be clearly shown to the user as certain information may otherwise be harder for certain users to see. For example, if the user is bombarded with information, message boxes, and bright text once opening the program, it is likely to be confusing and off putting. Therefore, by including the principle of emphasised text, less relevant text can be de-emphasised and important text will be emphasised. The user may only need to read the less relevant text once such as the program name VMCS; they are unlikely to forget this information. The primary colour will also be used to show more clearly what the user is doing. The user interaction elements of the interface will be highlighted in the primary colour. This will make it clearer to see what the user is currently interacting with by highlighting the information as they do so. If the markdown engine was not included, it would not be possible to say that the material dark accessibility principle had been met, therefore this usability feature is important to include.

Validation:

Sanitising user input

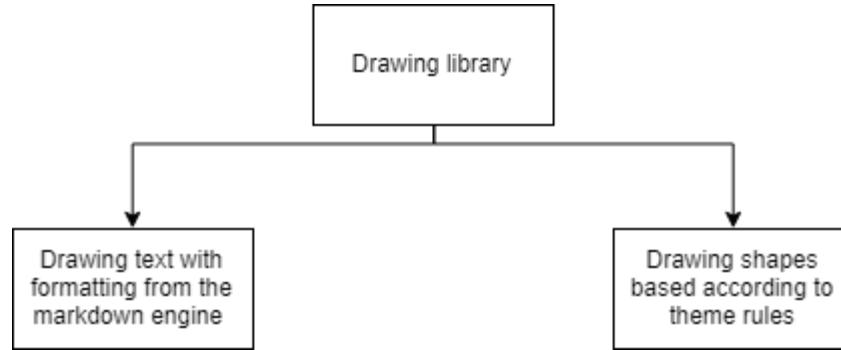
Explanation

It is possible that text entered into the program by the user may contain the special characters that cause a change in style. A validation method in the theme engine will be used to filter any of the special characters by means of a method that iterates through the string and removing any of the characters. This can be implemented by using a comparison statement, e.g. IF char IN specialchars.

Justification of validation

This validation is necessary to include because if it was not performed it would result in the styles being applied to the user input text when displayed on the screen, which would be confusing as they would not expect this to happen and think something may have gone wrong. This may cause the users to submit unnecessary bug reports or stop using the program as they would lose trust that there are no other bugs in the code. For this reason, it will be necessary to include this validation in order to handle any cases where the user enters text and the markdown engine is used to render it.

Drawing library



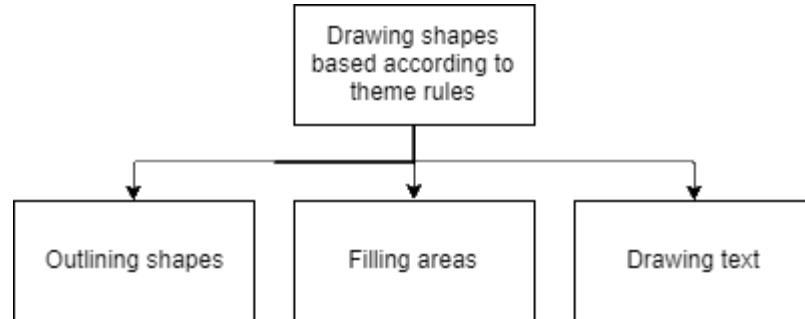
Explanation

The drawing library will provide frequently used drawing procedures that will be used by individual control and form classes. They will apply the theming principles to the form such that every control follows the theme. It will also apply the markdown engine into a method that draws the string at a specified location through parameters.

Justification

It is necessary to separate these problems into a separate library because they can be reused throughout the solution. Many of the controls will require similar procedures to use in drawing, e.g. draw a rectangle. This means that there is an opportunity for development time to be saved as code does not have to be rewritten and retested in every instance it is used and the results will be more consistent as every caller is using the exact same procedure not a specific implementation that could otherwise lead to inconsistencies.

Drawing functions



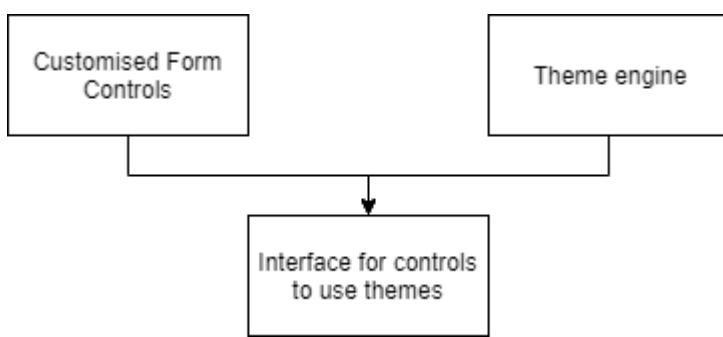
Explanation

The drawing functions will be specifically designed to apply the themes to every control. This is because instead of taking a colour as a parameter, they will take an emphasis. This will ensure that all callers are using the correct colour schemes as there is no opportunity for error(e.g. typos, different constants used).

Justification

It is necessary to solve these tasks separately because they will provide different features for different use cases. For example, outlining shapes could be used to draw a border around a box. The same task could be achieved by using filling an area twice with two different colours, however the tasks can be simplified to specific purposes by separating them into individual components. This is important because the functions performed by procedures that use the methods will be more obvious when reading and maintaining code at a later date.

Custom control interface



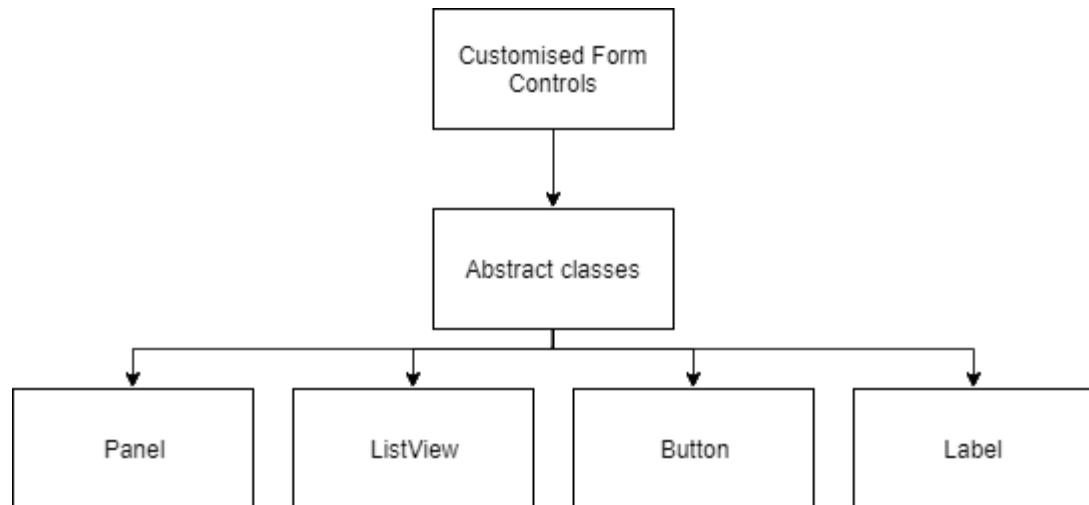
Explanation

The custom control interface will be the interface implemented into all custom control classes. The interface will include the fields “Emphasis” and “Layer”. This information will be used by the drawing library to determine which colours should be used to draw the background of the control and its text.

Justification

This interface is necessary because the utility library and base classes will apply general procedures to each control, such that the type of control will be irrelevant, only that it is a custom control. For this reason, it is necessary to have each implement an interface that has the necessary details(layer and emphasis) that can be used to draw the control appropriately.

Custom controls



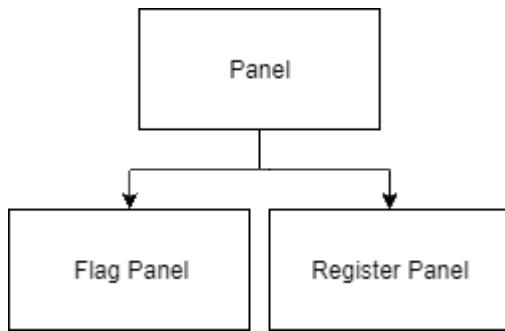
Explanation

Custom controls are form objects that are rendered on the screen and perform a specific purpose. For example, a button. Many different classes will be made to handle each specific control: one for a button, one for a textbox etc. The classes will include methods specialised to that class, performing a function that differentiates a normal textbox from a custom textbox implementation.

Justification

It is necessary to split the controls into individual classes because each control will serve a slightly different purpose to another. This links to the single responsibility success criterion as there will be few cases where classes can be reused because the methods provided are specific to a purpose, so the best approach is to separate the controls into separate classes where they have only methods that are relevant to them and that they will use.

Panel



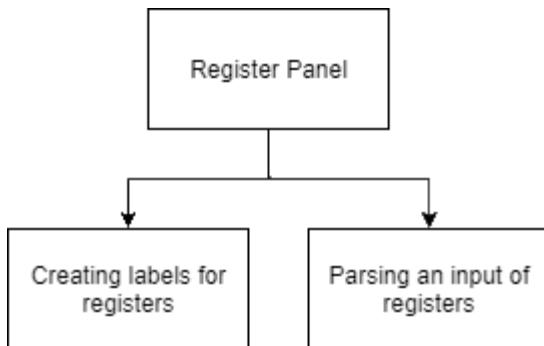
Explanation

The panel custom control will be an extension of the existing windows forms panel. It will be adapted to be drawn by the theming engine, such that the layer and emphasis of the form(fields from the interface) are used to determine how the panel is drawn rather than a specified colour. The flag panel class will display flags to the user and the register panel will display the registers to the user.

Justification

It is necessary to split the problem into two separate problems because the tasks performed by both are different. This will allow the two classes to have separate methods that are more relevant to their purpose than having general methods that may apply to both but would be less effective as they are not tailored to the needs of the class, e.g. the parsing procedure may not be the same for both. For this reason, the best approach is to separate the problem into two subproblems and tackle them separately.

Register panel



Explanation

The Register Panel will display registers in a user friendly way that clearly demonstrates which data will be relevant to the user and which can be hidden. This will be implemented through the use of the markdown engine. For example, the insignificant 00 bytes of a register will be de-emphasised such that the significant bytes will stand out. Labels for each register will be automatically generated and placed within the panel when the panel is created. A method will be provided that takes a RegisterGroup as an input then parses and displays the information in the created labels.

Justification

It is necessary to create a separate class to solve this problem because the regular panel class cannot provide enough functionality alone. This is because the class constructor will need to be used to automatically create the labels such that they are available immediately after the class is created hence do not have to be created and positioned every time by the caller. This will allow for the forms to be easily remodelled as the control will not be anchored to a specific position, i.e. if the class was moved, the labels within would always be created relative to the new position, not hardcoded to the last position so would otherwise need to be recoded to be placed at the new position.

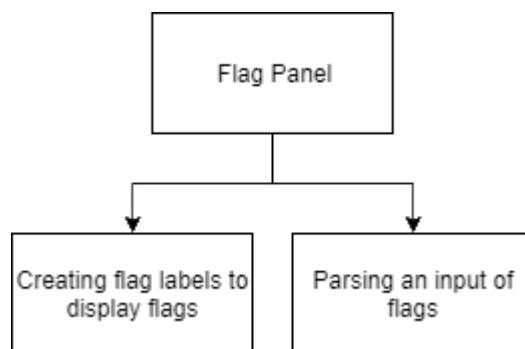
Usability feature: Register resizing

The user will be able to double click on the register panel to change the size of registers displayed. For example, double clicking on RAX will change the register to display EAX instead, which will display only 4 bytes instead of four.

Justification

This is an important usability feature because it will allow the users to adjust the amount of information that is displayed for them. The extra bytes could be irrelevant to them at that point in time, or could be confusing seeing so many numbers at once. This was shown to be important as the stakeholder research in the analysis section showed that 93.3% of the stakeholders did not understand what to do once opening x64DBG. It was concluded that this was due to the amount of excessive and unnecessary information displayed to the user. As a response to this, it is necessary that means to reduce the amount of information displayed is available to the user so they can decide what is and is not relevant to them.

Flag panel



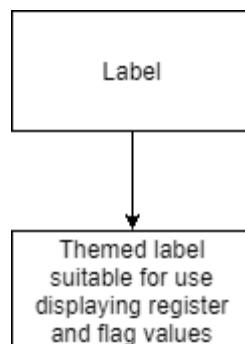
Explanation

The Flag Panel class will automatically create labels to show flags. It will have a method that takes a FlagSet as parameters. This will update all the labels to show the new values of the flags. The flags will be displayed as true or false, indicating whether they are on or not.

Justification

It is necessary to separate this feature into a separate class rather than use the existing panel class because there needs to be a procedure to update the flags such that the means to do so do not have to be repeated throughout the code. This is important because there may be a large amount of code required, e.g. checking each flag, setting the name, applying markdown. It would take a large amount of development time to maintain this because if any changes were made, the code would have to be updated to the new requirements in every place where it has been written. Because of this, the best approach is to have a separate module to generalise the procedure of performing the repetitive tasks in a single reusable procedure.

Custom Label



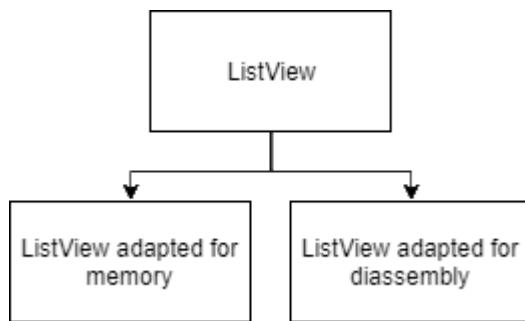
Explanation

The custom label class will be a base class that all other label classes will be derived from. It will inherit from the label class but include extra features that ensure the themes are applied correctly. This will include overriding the OnPaint() method which is called when the control is to be drawn to the screen. Instead of undergoing default control drawing procedures, it will be redirected to the theming engine.

Justification

It is necessary to include a separate label class because the text and background of the label will have to be drawn using the theme engine. This would otherwise have to be performed manually in the code by every instance of the label. This would add up fast in terms of code repetition as the register panel will need 16 labels(one for each register), the flags panel will need seven. To avoid repeating the procedures to do so in every label class, the better approach is to have a base class that provides the procedure and all other classes inherit from it. This is the best approach because it will also allow classes added in the future to automatically have the same drawing procedures as any other label such that all labels are consistent in design which will make the UI smoother and more minimalistic.

Custom ListView



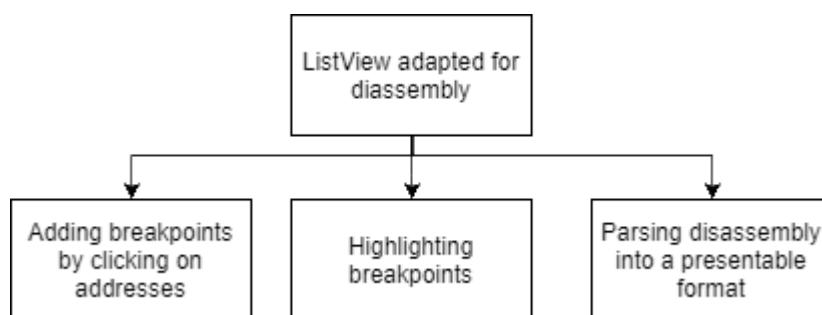
Explanation

The ListView custom class will be the base class for all derived ListView classes in the project. It will perform any necessary functions that would otherwise be repeated in the derived classes. For example, each ListView will be set to enable double buffering. This is a windows form feature that is disabled by default that caches the next frame of the listview. When a ListView has a large number of items, it gets very laggy, however double buffering is a good solution to reduce this.

Justification

It is necessary to solve this problem by splitting the ListView into two further problems because the two subproblems require very different procedures performed to operate. This is because they serve different purposes, so will be performing different tasks and solving further subproblems. This means that the best approach is to divide the problem and have a class to tackle each problem. This is because of the single responsibility principle success criterion. Each class should have a single purpose and not be designed to fit every purpose, as this would make the code harder to maintain because there will be lots of irrelevant functions in the class and classes would depend on components that they would not use.

Disassembly ListView



Explanation

The Disassembly ListView will display the disassembly pipelined from the processing layer and display it to the user. It will also show extra information such as the address where the instruction is found. It will also make use of the markdown engine by applying specific formatting to text to hide information that is not relevant such as insignificant zeros in a number. Disassembly will be loaded using a dictionary with an address as the key and a string(which holds the disassembly) as the value.

Justification

It is necessary to tackle this problem in a separate class because it will abstract the complex procedures required from a maintainer refactoring another part of the program that uses the class and will allow the class to be reused for other purposes. This is necessary because there will be many procedures that the caller does not need to know about, for example, sorting the list into address order. By including this as a separate function inside the class, the caller will only have to call a public function such as "AddLines()" to input to the class. This is important as it would otherwise require that the maintainer or user must first understand the class before using it, which is unnecessary as it is avoidable by using a separate class.

Usability feature: Adding breakpoints

The user will be able to click on address lines to add breakpoints. This will be implemented by using an event that fires a callback in the VM class that is running the instructions to add the breakpoint. This will be shown by changing the colour of the line to the primary colour of the design(purple by default).

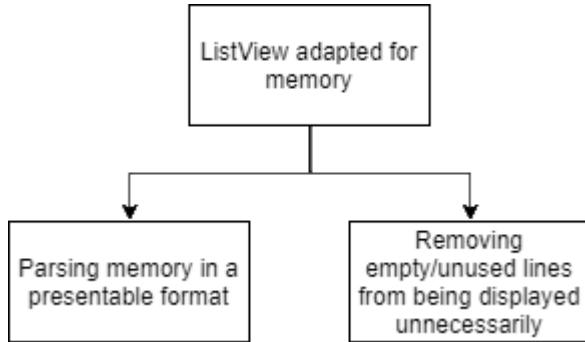
Justification of usability feature

Choosing to include this usability feature in the program is important because it will support the "Core features will be easily accessible through the UI" success criterion. Burying the ability to set a breakpoint behind many forms was shown to be a negative through the research of alternative software and stakeholder research. The vast majority of stakeholders surveyed did not know what to do when opening the program. As a countermeasure to this, the ability to

set breakpoints will be intuitive as clicking once on the instruction; no icons or specialist knowledge will be required, stakeholders will not even have to know what a breakpoint is.

Test #	Aim	How	Test data	Normal /Erroneous/ Boundary	Expected result	Justification
1	To create and remove breakpoints	An address will be clicked multiple times.	mov eax,0x100 mov ebx,0xff mov ebp,0x800000	Normal	Address changes colour	It is necessary to use this test data because there needs to be code such that there are disassembled lines to click on. This will allow the feature to be tested because it will be visibly seen whether the colour has changed or not; no coded debugging features means are necessary.
2	Test that the callback is invoked	The instructions will be executed with the Run button after setting a breakpoint.	mov eax,0x100 mov ebx,0xff mov ebp,0x800000	Normal	Program stops executing when highlighted line(breakpoint) is reached.	It is necessary to use this test data because there needs to be instructions present in memory in order to test that the execution stops at the breakpoint. It would be much harder to determine this programmatically because extra debugging procedures would have to be put in place to determine whether the callback had fired and the breakpoint was set, therefore the best approach is to use the simplest test data that will work such that debugging can be performed faster.

Memory viewer ListView



Explanation

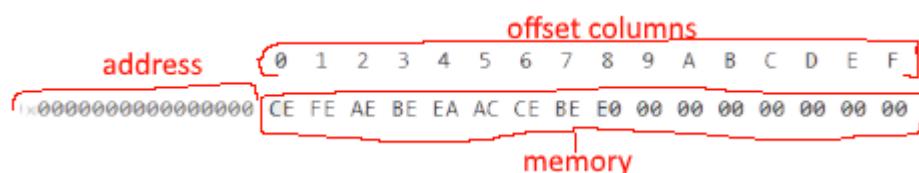
The memory viewer will provide a user interface for interacting and viewing memory. It will display all the bytes in hexadecimal, which is easier to interpret because every byte is always 2 hexadecimal characters when padded, whereas a decimal could be 1-3 digits.

Justification

It is necessary to solve this problem as a separate class because it will require procedures to parse and draw the control on screen differently to other forms. This is because the data will not just be a list, but other factors must also be considered such as drawing the items in the correct order without having to shuffle the list every time. This is necessary because inserting into a list in C# will cause the whole list to be reallocated and moved in memory. As the memory list view will have a large number of items, this would cause a great amount of lag and stutter when the user executes their program. Therefore, the best approach is to separate the ListView into its own class and program the class to display sorted by address of the data rather than by its index in the list. This class is necessary to the wider solution as it will introduce users to low level convention using hexadecimal. This is important because in stakeholder research and research of alternative software in the analysis section, it was shown through survey of the stakeholders that the majority did not know what to do when opening the program. This was concluded to be because of the great deal of irrelevant information that was shown to them. Because of this, a minimalistic memory viewer will be used that

only shows the necessary data; the address of the memory and the values in memory, with irrelevant features such as ASCII interpretations of the memory.

Usability feature: column headers

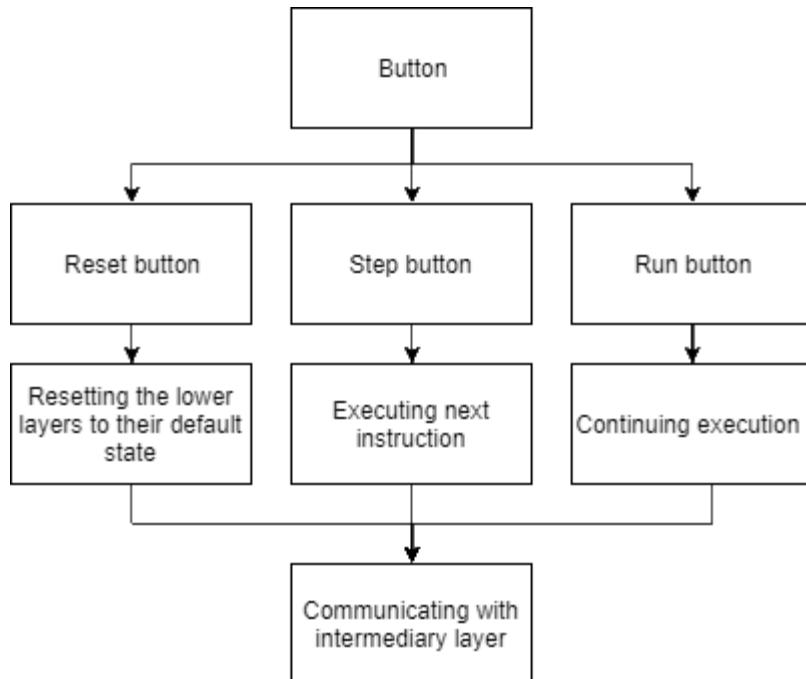


Column headers will be used to show clearly which byte is at which address. This is necessary as multiple bytes will be displayed on a single line instead of having one byte per line which would become very hard to interpret. There will be 16 bytes per line, 0x1 to 0xF. The digit offset from the address shown at the start will be placed above in line with its corresponding byte.

Justification

It is necessary to include this feature as part of the “User interface must be intuitive” success criterion. This is because having the offset columns above the memory will make it very easy to read which byte as it can take a long time for a user to develop the feel of knowing where every byte is on the line as the alternative software had lines of 16 length but did not show the columns. This is because they assumed all users would already be confident in viewing memory. Therefore, in order to make the program accessible to non-expert stakeholders, implementing this feature is the best choice as it will make the interface self explanatory rather than assuming the user knows what they are doing.

Usability feature: Control Buttons



Explanation

The control button class will provide the classes for the three key user interface interactions when using the program to execute instructions.

- Run button : Execute all the users code or until the next breakpoint is reached. This will be implemented by calling the Execute() function in the HypervisorBase that will automatically handle asynchronous execution such that the user can interact with the program still whilst their code executes. This feature will be important as the user may have just made modifications to their code and want to check the end result to see if the problem they addressed in their code has been fixed. Having a button that executes all instructions will allow them to do this as quickly as possible.
- Step button : Execute a single instruction. This will be implemented by calling the Execute() function and passing the step boolean parameter as true. This will also be called asynchronously. This button is necessary because the user may be debugging their code and need to execute line by line to track down the bugs in their code.
- Reset button : Reset the code to the beginning/entry point. This will be implemented by re-flashing the instructions onto the VM, hence resetting the processing layer; all the registers and memory will be cleared as the UpdateContext() method in the handle will be called. This is necessary to include as the user may be part way through debugging their code and realise they missed something, so providing this feature will enable to quickly get back to what they were doing without having to restart the program or select the file again.

Justification of usability feature

It is necessary to include this usability feature in order to meet the "User interface interaction will be kept minimal" success criterion. This is because the control buttons behind the scenes will be performing many necessary tasks to solve the subproblems. For example, the equivalent behaviour for the step button could be achieved by the user by setting a breakpoint on the next instruction and pressing the run button, however, this can be made more convenient for the user by performing the procedure through computational methods, hence reducing the amount of interaction required.

Tests

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	Crash test - Does the program open and layout correctly	The solution will be executed with no further input.	Will be determined during development as the layout has not been coded yet.	Normal	The program opens and does not crash, the layout of the controls is correct.	This test data is necessary as it is important to make sure that the form opens and draws properly before continuing testing as they would not be valid if the form

						did not work in the first place
2	Test whether the registers display properly	A few instructions will be executed that cause registers to change.	add rax, 0x1111 add rsi, 0x22222222 add rdx, 0x33333333	Normal	The following information will be displayed onscreen RAX : 0x1111 RCX : 0x22222222 RDX : 0x33333333	This test data is necessary to make sure that registers are displayed correctly in the interface and that there are no errors in the process of doing so, which is important as displaying the registers was described as an essential feature. The instructions will be tested before this test is conducted to ensure they are working.
3	Test whether memory displays properly	A few instructions will be executed that cause memory to change.	add byte ptr [0x30], 0x11 add word ptr [0x50], 0x2222 add dword ptr [0x1000], 0x33333333	Normal	The following information will be displayed onscreen [0x30] 0x11 [0x50] 0x2222 [0x1000] 0x33333333	It is necessary to use this test data to test that memory is displayed properly as there is a success criterion stating that essential information will be displayed to the user. This means that is important to test that this feature is working in order to meet the success criteria. The instructions will be tested before this test is conducted to ensure they are working.
4	Do step/run buttons perform their intended tasks?	Instructions will be loaded into the CU. The buttons will be clicked to test that for step and run perform their intended actions.	mov eax, 0x10 mov eax, 0x20 mov eax, 0x30 mov eax, 0x40	Normal	Step button executes one instruction Run instruction executes all instructions.	This test data is appropriate for this test as it will be clear which instruction was the last to be executed, e.g. if all instructions were executed, EAX will be 0x40, and will also be clear to see the changes when stepping instructions.

Usability feature: Help menu

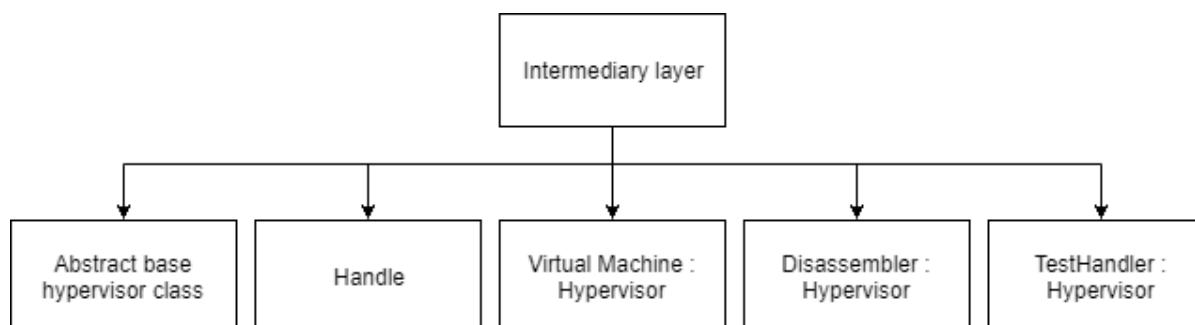
Explanation

A help menu will be available to the user. This will outline the core features of the program to a new user, and will still be viewable afterwards. It will give information on where the users can find out more about what the feature does as it may be hard to sum up in a single picture without crowding the screen.

Justification

It is necessary to include this usability feature because many of the stakeholders identified may have no assembly experience at all. This means that they would be as lost in VMCS as they would in an alternative identified in the research section. To keep the program accessible for users of any skill level, this usability feature will be required to meet the “The executable program and source code should be a great learning tool for the stakeholders.” success criterion.

Intermediary layer



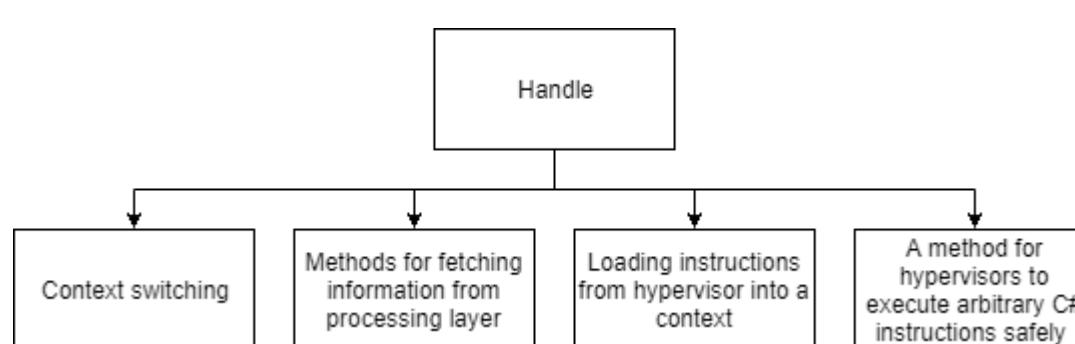
Explanation

The intermediary layer contains abstractions of the processing layer that can be used by the interface layer to perform operations more effectively without having to understand the specific details of the processing layer that will be irrelevant.

Justification

It is necessary to perform this abstraction as part of the dependency inversion principle success criterion. This is because a large part of the program will require assembly knowledge to develop, however, this means that a maintainer would also have to know assembly. To counter this problem, the intermediary layer will abstract the assembly concepts into high level objects and classes and provide simple functions that can be used by the interface layer to perform large scale operations without having to spend time understanding the lower layers of abstraction. This means that it will be easier for a future maintainer to develop the code and for a stakeholder reading a specific section of the code without having to understand the whole solution.

Handle



Explanation

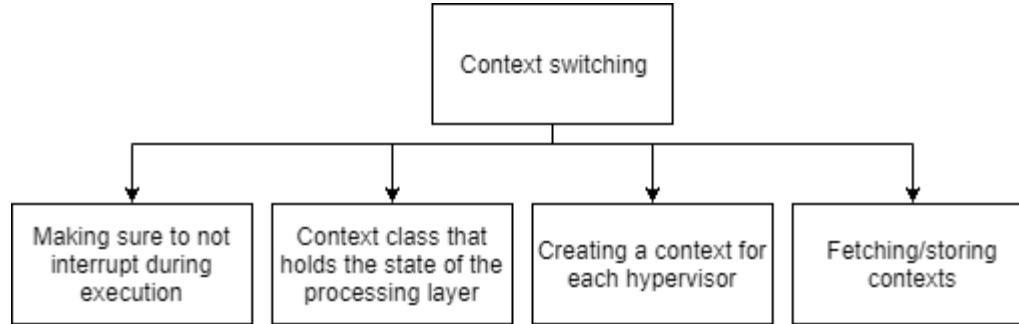
The Handle object will act as an interface for the intermediary layer to have more control over the CU in order for the classes to achieve their specific tasks. To implement this, the handle will be a subclass to the CU class. Classes in the intermediary layer will create a handle object that will allocate them a context in the CU such that they can load and execute their instructions without having to interact with the CU directly.

Justification

The handle class is necessary as an extension to the intermediary layer interface because there needs to be a way for multiple classes to execute instructions of the CU concurrently. This will be achieved through a context switching mechanism which will require the handle class such that the context swapped out of the CU when a new context is loaded is stored somewhere, then can be retrieved when is required to be re-loaded into the CU.

It is necessary to implement the Handle as part of the CU as many key variables that have been mentioned will be private variables. This means that external classes cannot see or use them. Therefore, in order to provide the extra level of control, the handle will be a subclass of the CU so it can interact with the private variables.

Context switching



Explanation

Context switching is the process of unloading one context from the CU and storing it, then loading in the next context. This will be provided through a method in the handle, where the handle will switch out the current context for its own context.

Justification

Context switching is necessary because of the design of the CU. The opcode interface interacts with the CurrentContext key variable mentioned in the CU. If another context is to be used, the current context must be swapped out with the new context before the caller handle can begin interacting with the CU, or the new context will not be affected. Therefore, this procedure is necessary as otherwise there would be no way to have multiple functional handles at once.

Making sure to not interrupt during execution

Explanation

The current context may be executing at the time the context swap is called. If the context were to be swapped at this moment, there would be a race condition between the two threads trying to use the CU. To prevent this, a thread-safe boolean will be used to determine whether a thread is using the context.

```
BOOL Busy;  
DEF WaitNotBusy()  
  
    // Wait whilst another thread is using the CU.  
WHILE Busy == true  
    // Wait 100ms  
    WAIT 0.1;  
    ELIHW;  
  
    // Reserve the CU for the caller thread  
    Busy = true;  
  
    RETURN;  
FED;
```

A thread will use the WaitNotBusy() method to wait until it is safe to execute instructions. This will automatically set the CU as busy, then will be unset once the thread finishes executing instructions.

Justification

It is necessary to tackle this problem because having the race condition between threads will result in erroneous behaviour, which could be an exception, or would result in the instructions of the new context being executed on the previous context. This would create a lot of bugs that are unsolvable in other parts of the program, therefore is very important to tackle this problem in the handle class.

Testing

Test #	Aim	How	Test data	Expected result	Justification
1	Test the process of waiting for the CU to no longer be in use	The Busy boolean will be set to true for a measured time period, and during the time period another thread will call WaitNotBusy().	<pre> Busy = true new Thread(Wait(10s); Busy = false;); // This will execute whilst // the previous thread waits WaitNotBusy(); // An obvious indicator that // will show whether it took // 10 seconds or not print("Done!"); </pre>	"Done" will be printed after ten seconds, not immediately after.	This test data is necessary because there will be many occasions where threads will be acting independently of one another. This will lead to multiple threads trying to use the CU at the same time. This test data will show whether the threads are coordinated properly to prove that the next thread will wait for the Busy boolean to be false before interacting with the CU, which is tested using two separate threads, one which sets busy to false after a timer(pretending that the CU is in use) and the other to wait for the other thread to do so.

Context class that holds the state of the processing layer

Explanation

The handle class will reuse the context class to simplify the process of swapping information in and out of the CU. This is because the context class has all the variables necessary to hold the important information of the CU that will affect the result. An instance of a context can be created then the information in the CU loaded into each of its variables.

Justification

It is necessary to tackle this problem separately because it allows for the reuse of existing classes in order to simplify the steps needed to solve the problem of storing the information of the CU. This will reduce the amount of code that needs to be written and maintained, therefore save development time. Moreover, other parts of the handle class will be able to interact more easily with the variables that are stored in one object, the context, than have many separate variables to hold the equivalent information.

Key variable: Static handle-context dictionary

Explanation

The dictionary will be used to retrieve the context of a given handle. This will allow the contexts to be stored once switched out of the CU, and easily retrieved when a context is to be swapped in. This can be achieved by creating a key of the desired handle, then using that handle object to access the dictionary by index accessor in order to interact with the associated context.

Justification

This key variable is necessary as it will solve the subproblem of Fetching/storing contexts. By making the variable static, it will be consistent across all instances of the class. This solves the problem completely because every handle will be accessing the same data, such that the correct context will always be retrieved/set.

Creating a context for each hypervisor

Explanation

A new context will be created for each hypervisor and stored in the handle-context dictionary. To create the handle, the hypervisor will input a MemorySpace as a parameter. This will contain all the instructions provided by the caller, such as instructions entered by the user.

Justification

This solution is important for other parts of the handle solution. By assigning each handle a context, it will be simple to determine which context needs to be loaded when a caller requests a context switch and where to store the current context. Therefore, it will be possible to achieve the goal of having multiple classes execute instructions concurrently. The best approach to this problem is to only require a MemorySpace as an input. This is because the caller will generally want to start with all other key variables, such as Registers and flags, empty. If they have a specific need otherwise, they can change the variables using the interface api.

Methods for fetching information from the processing layer

Explanation

The handle class will include an extension of some of the intermediary api methods that can be used to more effectively obtain information from the processing layer in a consistent format and without having to interact with it directly, or providing a safe way to interact with it.

- Run() : The run function will incorporate the execute method in the intermediary api and the WaitNotBusy() in Handle.

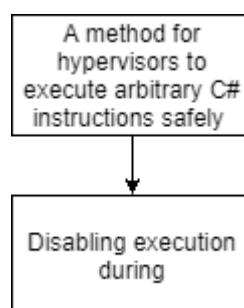
DEF Run()

```
WaitNotBusy();
IsBusy = true;
CU.Execute();
IsBusy = false;
RETURN;
```

FED;

This function will wait until the Busy boolean in Handle is false then set Busy to true, execute, then set busy back to false to allow the CU to be used in the future. This function will be necessary to avoid handles running on top of each other, which would result in erroneous behaviour.

- Invoke() : The invoke function will use the WaitNotBusy() in the handle to execute arbitrary code safely.



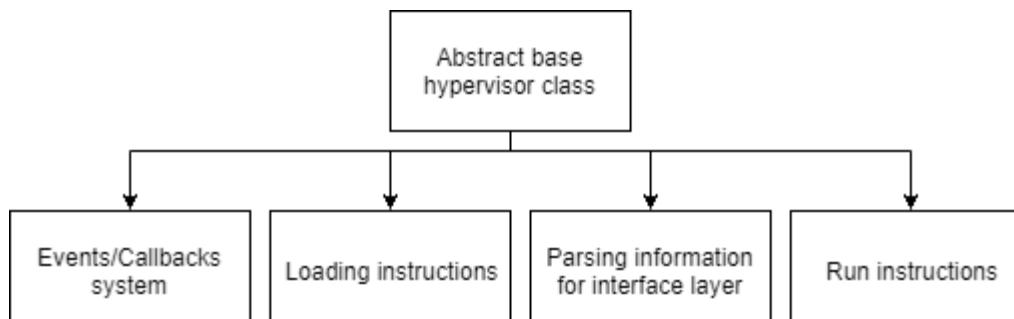
The programmer will input a C# Action object as a parameter. The Busy variable will be set true(which will prevent other callers from executing), the action will be executed, then set false afterwards. This will allow arbitrary code to be executed safely such that it is certain the CU is not executing instructions at that given time.

- UpdateContext() : The update context function will allow a caller to update the context of their handle with a new context. For example, the caller would be able to load new instructions from user input without destroying its handle.

Justification

It is necessary to solve this problem separately to ensure modularity in the processing layer. The intermediary api methods in the handle class are dependent on the handle class, they could not exist otherwise. Because of this, the best approach is to include the modular procedures in the CU that are not dependent on other classes and keep the classes dependent on Handle separate. This will allow smoother design transitions and maintenance in the future because it is clear which functions will be affected by specific changes. By doing so, several other subproblems in the stepwise refinement have been solved.

HypervisorBase



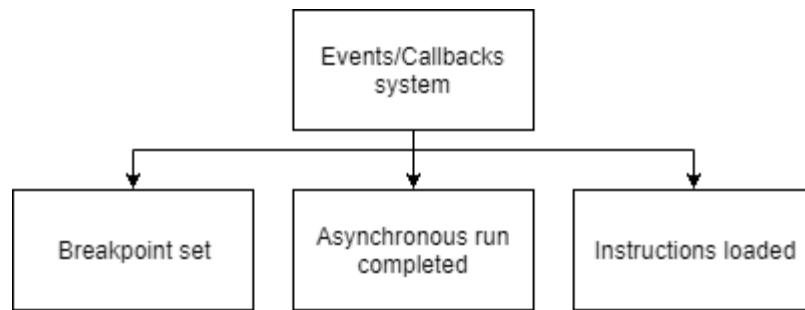
Explanation

The HypervisorBase class will be the base class for all hypervisor classes. Hypervisor classes are classes that use the CU in a specific way to perform a different task. The HypervisorBase class will provide many useful procedures that would otherwise be repeated throughout the hypervisor classes.

Justification

It is necessary to use a base class for this purpose to reduce code. There are many procedures that every hypervisor class will use in order to function. For example, every hypervisor will need a handle. Instead of coding this separately, a general function in the base class can be called in the base constructor such that the procedure does not have to be rewritten and retested for every class, hence saving development time.

Events/Callbacks system



Explanation

The events/callbacks system will provide multiple C# events that can be used by the caller that instanced the class. These will allow for specific callbacks once a specific task has been completed.

(Implementations of events will be further discussed in their relevant sections)

- Breakpoint set : There will be an event for when a breakpoint is set. This event is necessary because a callback will be required to refresh the GUI in the interface layer to show the new breakpoint differently. (Breakpoints will be highlighted in the disassembly)
- Asynchronous run completed : There will be an event for when an async run has been completed. This event is necessary because a callback will be used to refresh the interface to fetch the new register values etc and update the information in the UI.
- Instructions loaded : There will be an event for when instructions have been loaded into the context of the handle. This will be necessary for the Disassembler hypervisor. By having an event to listen for this, instructions must only be disassembled once per program loaded.

Justification

It is necessary to tackle this problem separately because there are certain operations in other classes that would otherwise have very inefficient solutions if it were not for an events system. For example, the UI would have to continuously check for any new changes to registers. This would be very performance inefficient because the checking intervals would have to be very small in order for the program to feel fluid. If this were not implemented, there would be universal delays across all platforms no matter how performant they are because of hard coded interval conditions. To have program lag would not show the program as attractive to the stakeholders.

Loading instructions

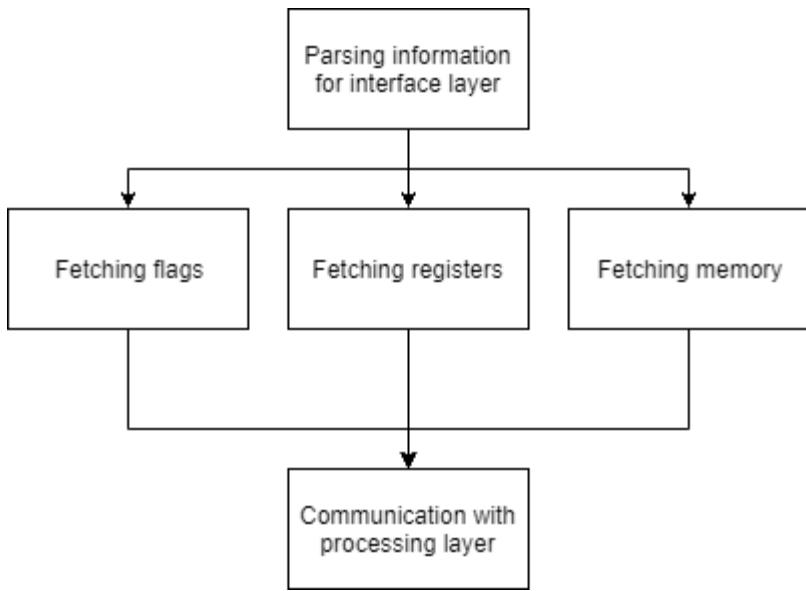
Explanation

A method will enable callers to load a new MemorySpace into the context of their handle. For example, this could be a new set of instructions for the CU. This will raise the Instructions Loaded event and automatically create a new context for the handle method.

Justification

It is necessary to tackle this problem separately from the UpdateContext method provided in the handle because elements of the problem can be abstracted from the caller. This includes creating a context from the MemorySpace, which will automatically be performed in the method instead of rewriting the procedure to do so in every caller class. This will also allow the Instructions Loaded event to be invoked, as the handle method would not be designed to perform this task as the modules are independent of one another; they are only concerned of the inputs and outputs to functions not the existence of the class.

Parsing information for interface layer



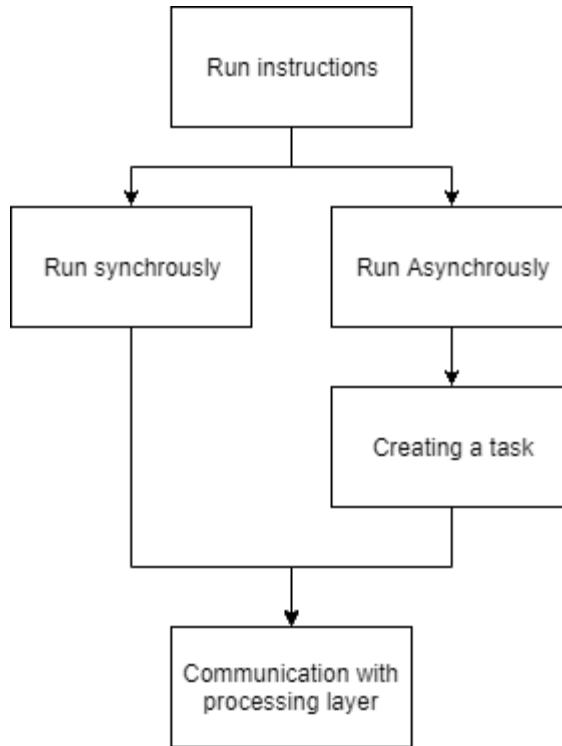
Explanation

Methods will be provided to specifically tailor outputs of the CU intermediary api to the inputs of the interface layer. For example, fetching registers, fetching memory, and fetching flags. Any required parsing or casting will also be performed here, such that parsing procedures do not have to be repeated in the interface layer as it will be provided in the format it is required.

Justification

It is necessary to tackle this problem separately from the fetching methods in the CU intermediary api because these methods will be context and thread safe. This means that the Busy variable will be used to check whether the correct context is loaded in the CU that the caller requires and loading in the correct context as required. If this was not checked, the caller would receive the information of another context rather than the context of its own, which would result in incorrect information being displayed in the UI.

Run instructions



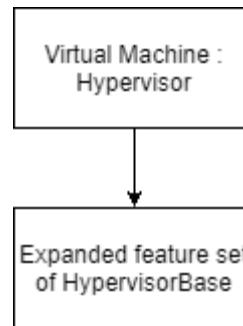
Explanation

Asynchronous and synchronous run methods will be provided in the hypervisor base class. The run asynchronous method will create a C# Task object that creates a new thread to call Handle.Run(), then raise the RunComplete event. The run synchronous call the Handle.Run() method without creating an extra thread and also raise the RunComplete event afterwards.

Justification

It is necessary to tackle these two problems separately because there are certain cases where it would be optimal to run synchronously rather than run asynchronously. For example, this would be the case for small individual tasks such as TestHandler testcase programs where the overhead to create a new thread would be greater than to run synchronously. For this reason, the problems should be tackled separately to optimise the performance of the program, as execution in the processing layer is where the most calculations and algorithms will be performed, hence needs more optimisation than other parts of the solution.

VM



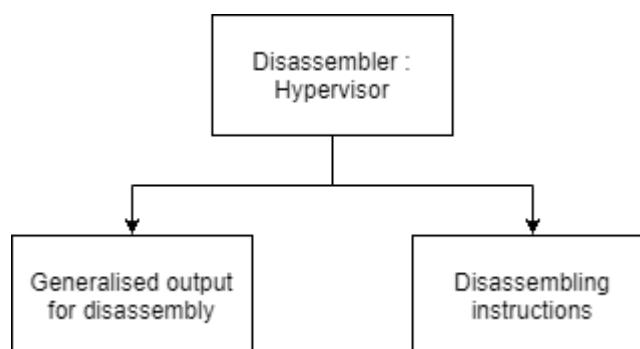
Explanation

The VM class will provide an instanceable derived class of the hypervisor base class. It will be a general purpose hypervisor that has a few extra features. The VM class will be the hypervisor that executes the user's instructions as its features are not specific to one purpose such as the Disassembler hypervisor.

Justification

It is necessary to have a separate class for the VM because there are many features that would be excessive if included in the base class as they would be included. For example, there will be a method to jump to a specific address. These are unnecessary in other hypervisor classes, hence as part of the interface segregation principle success criterion, it will be necessary to have a split the HypervisorBase class and have a separate VM class that has more specific functionality.

Disassembler



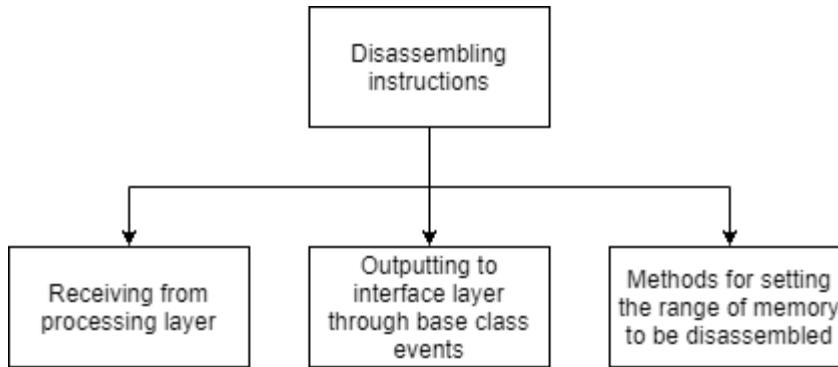
Explanation

The Disassembler class will inherit from the hypervisor class and be a hypervisor specific to disassembling the instructions of another hypervisor class then use events to output the results. It will perform specific operations in order to achieve this. Instructions will be disassembled in specific ranges where the users code is stored. This will be implemented by checking the SegmentMap of the MemorySpace of the target hypervisor to see the range in which the instructions are stored. The instructions will be disassembled then output an address range object and a dictionary of address string pairs. This will allow for instructions ranges to be searched for by entering their address and then the desired line will be returned

Justification

It is necessary to tackle this problem in a separate class as part of the single responsibility principle success criterion, as disassembly is a specific task general to any hypervisor. This shows that if it was part of a single class, other modules that may need the feature in the future will not be able to without significant code modification. If it was included in the base class, the hypervisors that do not require disassembly would be forced to depend on methods which are irrelevant to them. For these reasons, the best approach is to have a separate module that can be developed separately.

Disassembling instructions



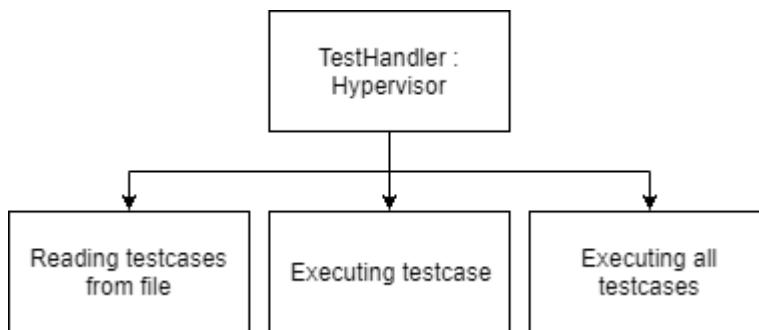
Explanation

The sub-problem of disassembling instructions will be solved by splitting the problem into three tasks. Receiving input from the processing layer will be achieved by using the existing events in the HypervisorBase class. The “Instructions Loaded” event in the hypervisor base will be used to listen for when new instructions are loaded into the target, then pipeline them through the disassembly procedures. The disassembled instructions will be output by events specific to the Disassembler class, e.g. “Disassemble complete”. These will send the new disassembled instructions to callback procedure and allow further processing. A method will be provided to disassemble a specific area of memory instead of only the main segment. It will take an address range as an input.

Justification

It is necessary to tackle these three tasks separately because they can be created by reusing other methods from other modules. For example, the loaded instructions event. This is the best approach to this task because it will mean that only new instructions have to be parsed as opposed to continually disassembling the instructions on the off chance that new instructions are there to be disassembled, hence saving a large amount of performance as disassembly will be a resource-heavy procedure since instructions must be decoded and executed again. This solution is the same for the output to the interface using events, as instead of the interface having to continually check for new disassembly, the event will be used, hence having the same advantages as mentioned. It is also necessary to solve the problem of disassembling a range in this subproblem as it will save a large amount of performance. This is because instead of disassembling the entire MemorySpace(which is potentially the size of the user computer’s RAM), only the relevant sections will be disassembled.

TestHandler



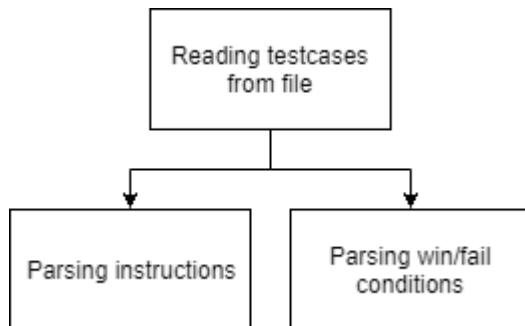
Explanation

The TestHandler class will perform heuristic code self-testing and analysis on the overall solution. This specifically includes testing the processing layer where logical errors, discrepancies, and anomalies in results are most likely to occur. This will be performed by executing pre-defined programs on the handle of the TestHandler hypervisor that aim to test whether the outputs produced are correct. These programs will be developed and executed in a non-virtualised environment where the results will be accurate and realistic. This is important to ensure that the correct results are compared against the code hence will definitively show whether code is correct or not. The TestHandler will be able to execute the testcases on a macroscale. This will involve executing all testcases at once, such that changes to the code can immediately be ruled as erroneous or functioning. As this will return a large number of results, the option to execute a single testcase will pertain as there may be cases where a certain self-contained module is under development, such that effects on other parts of the program are improbable. Testcases will be read from a file from XML format and parsed into Checkpoints, which are similar to breakpoints, except when broken, a series of conditions will be tested to compare the data in the CU and the data from the input testcase.

Justification

It is important to solve this problem because as the solution increases with scale, it will become a significant task to track down minor bugs and logical errors that may arise under very specific conditions or in a separate module unrelated to the module under development. Tracking down these bugs would cost a great amount of development time and they may be hard to find if they are present in a rarely used part of the solution. As a countermeasure to this, the TestHandler will ensure that these parts of the solution are continually tested after every modification to the code such that the small logical errors and bugs will be ironed out and an overall more robust solution will be produced as a result, allowing the essential features to be developed more thoroughly as there will be more development resources available to be allocated to them.

Reading testcases from a file



Explanation

Testcases will be read from the testcases folder. Every XML file in that folder will first be validated to make sure that the file has the correct base element, “Testcase”, for the testcase program(incase the user has any other files in that directory). There will be several elements within the root element that are parsed(others will be ignored),

- <Hex> - The hex element will hold the program instructions for the testcase, stored as “hex characters” where each byte is written in the ASCII representation of a byte in hex. These will be read, validated, and stored in a MemorySpace ready to be loaded into the handle.
- <Checkpoint> - The Checkpoint element will divide the testcase into separate tests and contain sub elements that determine what is tested in the checkpoint, however the element itself will have a “tag” and “position” attribute. The sub-checkpoint elements will be referred to as the win/fail conditions of the testcase. The tag attribute will hold the name of the checkpoint which will make it easier to identify which parts of the testcase were success. The position attribute will be the address to break at and test the conditions; the position at which the breakpoint will be inserted.
- <Register> - The Register element as part of a checkpoint will hold information to be tested against a specific register when the checkpoint is reached. This will have “id”, “size” attributes, and the value to compare to as the value of the element. The “id” attribute will hold the name of the register, e.g. A, B, SP. The size will then hold the size of that register to test against, as certain tests may only want to test a specific part of the register.
- <Memory> - The Memory element as part of a checkpoint will hold information to test a specific address in memory. This will have “offset_register” and “offset” attributes, that allow for flexibility in how the address is determined, such as using an offset from the stack pointer to test memory stored in the stack. This would be performed by setting the offset_register attribute to “SP” and the “offset” attribute to the sign-extended 8 byte offset from that location.
- <Flag> - The Flag element as part of a checkpoint will hold information to test a flag in the CU. The attribute, “name”, will hold the name of the flag to test, e.g. “Carry”. The value of the element will be “1” if the flag should be on and “0” if the flag should be off.

As the testcase executes, the checkpoints will be tested and stored as an output XML. This will include the comparison of the expected value and the value found. This will allow for bugs to be tracked down faster as it will be clear which function that was tested is causing the bug. There will also be a “passed” field for each checkpoint and testcase. This will allow the maintainer to quickly scan through the file looking for any “passed=false” attributes to quickly determine whether a testcase was successful or not. If any of the testcases failed, this will be displayed in a message so a result can quickly be dismissed if there were no failures in any testcase.

Justification

It is necessary to solve this problem through modular testcases because it will be easier to adapt and update the testcases as the development progresses. It will also allow for clear demonstration that each testcase fully tests all the elements of the feature to the greatest extent as the tests will be split up into easily visible testcases rather than one large testcase. Because of this, bugs will be tracked down faster as there will be more relevant information to that testcase; the information of other testcases will not obstruct the maintainer. This will also allow new tests to be easily implemented in the future. This could be done by either user or maintainer. This is because the tests will automatically be loaded and tested, such that instead of requiring the user/maintainer to fully understand the workings and details of the function they are testing, they can instead only have to understand the expected behaviour, create a testcase to define that behaviour, then use that perform the test autonomously instead of having to manually test the code every time, hence saving great amounts of development time.

Validation of testcases

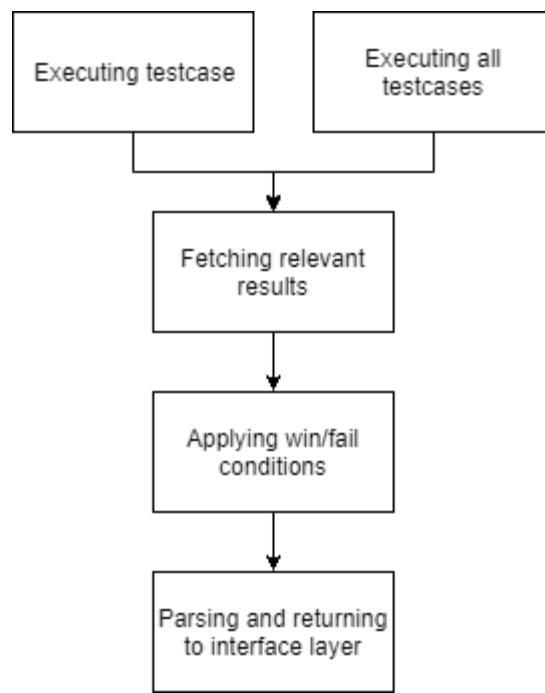
Validation procedures can be reused from other parts of the solution. For example, when reading the instructions from the testcase file, the same procedure used to validate the ASCII representations of hex input bytes in the IO utility

library will be used. This will save development time because new code does not have to be developed independently and tested again. To validate flag names in flag elements, the same validation procedure defined in the FlagSet struct can be reused, which will result in consistent flag names throughout the solution. Register tags must be validated through a new procedure because there is no other place in the solution where register names are validated. This will be a simple test against a known list of correct register names. This new procedure can be reused in the validation of the memory offset_register attributes as they have the same input format.

Justification of validation

It is necessary to perform this validation as there is a large possibility that the user may enter invalid inputs into the file. This would result in erroneous and undefined behaviour, which would be entirely detrimental in the TestHandler. For example, an invalid input may lead to false-positives in the output, giving the impression that functions are working, when in reality they are not, leading to the bugs going undetected and unsolved. For this reason, it is very important to perform the necessary validation checks in order to ensure the TestHandler is a productive feature, not counterproductive.

Executing testcases



Explanation

Testcases will be executed in the CU under the same conditions as they would be executed as if they were the user's program. The instructions to be executed will first be loaded into the handle context, then breakpoints inserted where each checkpoint is to be tested. Once the executions breaks(The CU.Execute() methods returns on break), the win/fail conditions for the respective checkpoint will be tested and stored in an output object. After all checkpoints have been tested, the output objects will be parsed into XML format using the same procedures used to decode the input XML and returned to the caller.

Justification

It is necessary to break this problem down into steps because it needs to be executed in a specific order to determine which win conditions for a checkpoint has been met at that particular step. This has to be done using the CU because the results would not be representative of what may be expected from the user's program if they were not executed exactly like how the user's program would be executed. For this reason, going through extra steps to break down the problem into individual sub-checkpoints will be important in making sure the tests are reliable as possible.

Iterative testcase data for opcodes

Add/Adc

m	How	Test data	Expected result
test addition of immediate byte with 'A' register	The opcode variant to add an immediate with the 'A' register will be used.	add al,0x10 add ax,0x2000 add eax,0x30000 add rax,0x40000000	<Register id="A" size="1">10</Register> <Register id="A" size="2">2010</Register> <Register id="A" size="4">40032010</Register>
test addition of immediate memory with memory	The opcode variant to add an immediate with memory at an address will be used.	add bl,0x10 add bh,0x10 add bx,0x1000 add ebx,0x30000 add rbx,0x40000000	<Register id="B" size="1">10</Register> <Register id="B" size="2">2010</Register> <Register id="B" size = "8">40032010</Register>
test addition of sign extended word with word	The opcode variant to sign extend and add an immediate with a dword will be used.	add dx,0xffff0 add ecx,0xffffffff2	<Register id="D" size="2">FFF0</Register> <Register id="C" size="4">FFFFFFF2</Register>

test addition of register with memory	The opcode variant to add a register with a memory address will be used. The values must first be moved into registers because adding immediates uses a separate opcode to adding registers to memory.	<pre>mov eax,0x10 mov ebx,0x2000 mov ecx,0x30000000 mov edx,0x0 add BYTE PTR [rsp],al add WORD PTR [rsp+0x1],bx add DWORD PTR [rsp+0x3],ecx</pre>	<Memory offset_register="SP">10</Memory> <Memory offset_register="SP" offset="1">0020</Memory> <Memory offset_register="SP" offset="3">00000030</Memory>
test addition of memory with register	The opcode variant to add a memory at an address with a register will be used. (Note that the data mov'd into the registers from the previous checkpoint is reused)	<pre>add BYTE PTR [rsp],al add WORD PTR [rsp+0x1],bx add DWORD PTR [rsp+0x3],ecx add al,BYTE PTR [rsp]</pre>	<Register id="A" size="1">20</Register> <Register id="B" size="2">4000</Register> <Register id="C" size="4">60000000</Register> <Register id="D" size="8">30000000</Register>
test addition of two registers	The opcode variant to add two registers will be used	<pre>mov eax,0x0 mov ebx,0x0 mov ecx,0x0 mov edx,0x33332211 add al,dl add bx,dx add ecx,edx</pre>	<Register id="A" size="1">11</Register> <Register id="B" size="2">2211</Register> <Register id="C" size="4">33332211</Register>
test addition with expected overflow	The add opcode will be used with operands crafted such that an overflow will be expected in the result	<pre>mov eax,0x7fffffff add eax,0x1 (will overflow from positive to negative)</pre>	<Flag name="Overflow">1</Flag>
test addition with expected carry	The add opcode will be used with operands crafted such that a carry will be expected in the result	<pre>mov eax,0x80000000 add eax,0x80000001 (will overflow the maximum value for a qword)</pre>	<Flag name="Carry">1</Flag>
test addition with ADC opcode	The ADC opcode will be used instead of the add opcode, with the CF set initially	<pre>mov eax,0xf000 mov ebx,0x0 add ax,0x1000(cause CF to be set) adc bx,0x0 (should be one because CF set)</pre>	<Register id="B" size="2">1</Register> <Flag name="Carry">0</Flag>

Justification of test data for add

It is necessary to include this test data for add because there are many different variants of the opcode, meaning that it is more likely for something to go wrong in a variant that is rarely used, hence would go unnoticed. For this reason it is important to include this test data to ensure the module is fully functional. This test data is appropriate for the problem because it has been specifically designed to the aim of the test. For example, adding 1 onto 0x7FFF..FF, will cause an overflow into the sign bit(MSB), therefore on the overflow test #7, there should always be an overflow, so would be clear if this part of the function was not working properly. Tests 1-7 test every size of operand, e.g. QWORDS, WORDS, etc. This will indirectly test the procedures of determining the operand size and for setting the registers, which is important to the problem as an error in one of the key data types would potentially lead to errors and exceptions, hence is important to raise flags when these functions are not working, such that the developer can draw conclusions about where to track down bugs faster.

Sub/sbb

m	How	Test data	Expected result
sub 'A' register by immediate	The opcode variant to subtract an immediate from the 'A' register will be used.	<pre>movabs rax,0x4444444433332211 sub al,0x22 sbb al,0x44 sub ax,0x4400 sbb ax,0x8800 sub eax,0x66660000 sbb eax,0xbbbb0000</pre>	<Register id="A" size="4">111256a8</Register>

Sub 'A' register by sign extended word immediate	The opcode variant to subtract a sign extended dword immediate from the 'A' register will be used.	<pre>movabs rax,0xf1f1f1f122222222 sub rax,0xffffffff80000000 sbb rax,0xffffffff8fffffe</pre>	<Register id="A" size="8">f1f1f1f212222223</Register> <Flag name="Carry">1</Flag>
Sub register by immediate	The opcode variant to subtract an immediate from a register will be used.	<pre>movabs rbx,0x4444444433332211 sub bl,0x22 sbb bl,0x44 sub bx,0x4400 sbb bx,0x8800 sub ebx,0x66660000 sbb ebx,0xbbbb0000</pre>	<Register id="B" size="4">111256a8</Register>
Sub register by sign extended word immediate	The opcode variant to subtract a sign extended dword immediate from a register will be used.	<pre>movabs rbx,0xf1f1f1f122222222 sub rbx,0xffffffff80000000 sbb rbx,0xffffffff8fffffe</pre>	<Register id="B" size="8">f1f1f1f212222223</Register> <Flag name="Carry">1</Flag>
Sub/sbb register by sign extended byte	The opcode variant to subtract a sign extended byte immediate from a register will be used. Both sbb and sub opcodes are tested in this checkpoint.	<pre>movabs rax,0x4444444433332211 sub ax,0xff80 sub eax,0xfffff90 sbb rax,0xfffffffffffffa0</pre>	<Register id="A" size="8">33332360</Register> <Flag name="Carry">1</Flag>
Sub/sbb memory at address register	The opcode variant to subtract a register from a value in memory at an address will be used. Both sbb and sub opcodes are tested in this checkpoint.	<pre>movabs rax,0x4444444433332211 movabs rbx,0x8888888866664422 mov QWORD PTR [rsp],0xffffffffbbbb8844 sub al,bl sbb BYTE PTR [rsp],al sub ax,bx sbb WORD PTR [rsp],ax sub eax,ebx sbb DWORD PTR [rsp],eax sub rax,rbx sbb QWORD PTR [rsp],rax</pre>	<Register id="A" size="8">7777777866675689</Register> <Register id="B" size="8">8888888866664422</Register> <Memory offset_register="SP">50b8868887888888</Memory>

Justification of test data for sub

This test case data is appropriate for the sub opcode as it can reuse the test checkpoints for the SBB opcode by including both in the test program. This tackles both problems at once such that individual testcases do not have to be made for both of the opcodes as they are both very similar and use the same procedures, hence bugs in one will be visible in the other. Therefore it is appropriate to the test data as it applies to both rather than having to test both separately which would otherwise increase the volume of data that needs to be reviewed when analysing test results. The test data is also specifically designed to use certain variants of the opcode to show their functionality. When assembled, the test data in tests #4 and #5 will automatically be shortened into their sign extended immediate form, which is not visible in disassembly, therefore the extra variants will still be tested.

Mov

How	Test data	Expected result
The opcode variant to move an immediate into a register will be used.	<pre>mov al,0xf1 mov bx,0xf2f2 mov ecx,0xf33f3f3 movabs rdx,0xf4444444f444f4f4</pre>	<Register id="A" size="1">f1</Register> <Register id="B" size="2">f2f2</Register> <Register id="C" size="4">f33f3f3</Register> <Register id="D" size="8">F4444444F444F4F4</Register>

ove reg to mem	The opcode variant to move a register into a memory location will be used.	mov BYTE PTR [rsp],al mov WORD PTR [rsp-0x2],bx mov DWORD PTR [rsp-0x6],ecx mov QWORD PTR [rsp-0xe],rdx	<Memory offset_register="SP">F1</Memory> <Memory offset_register="SP" offset="FE">F2F2</Memory> <Memory offset_register="SP" offset="FA">F3F333F3</Memory> <Memory offset_register="SP" offset="F2">f4f444f4444444f4</Memory>
ove with sign extension, g to reg	The opcode variant to move a register into another register and sign extend when doing so will be used.	movsx bx,cl movsx edx,cx movsx rcx,bl movsx ecx,bx movsx rbx,cx movsxd rax,ebx	<Register id="A" size="8">fffffffffffff3</Register> <Register id="B" size="8">fffffffffffff3</Register> <Register id="C" size="4">fffffff3</Register> <Register id="D" size="4">fffff3f3</Register>
ove with zero extension, g to reg	The opcode variant to move a register into another register and zero extend when doing so will be used.	movzx bx,cl movzx edx,cx movzx eax,bl movzx ecx,bx movzx rbx,cx	<Register id="A" size="8">f3</Register> <Register id="B" size="8">f3</Register> <Register id="C" size="4">f3</Register> <Register id="D" size="4">fff3</Register>

Justification of test data for mov

The test data for the mov operand is necessary as it tests every variant of the opcode, and the movsx and movzx opcodes. This will as a result also test that the Bitwise.SignExtend() and Bitwise.ZeroExtend() methods work correctly as movsx/movzx depend on those functions respectively. This will allow more of the solution to be tested through a single testcase whilst keeping the scope of the testcase narrow and specific to the mov opcode, such that the mov testcase failing will still show that a task as part of the procedure of executing the mov opcode has failed, and is not an irrelevant module, such that the benefits of tracking down bugs in the code faster are still apparent.

Lea

m	How	Test data	Expected result
load effective address to WORD register	The opcode variant to load the effective address of a SIB byte into a register will be used.	mov rax,0x123 lea rbx,[rax*8+0xffff]	<Register id="B" size="8">1000917</Register>
load effective address to WORD register	The opcode variant to load the effective address of a SIB byte into a dword register will be used.	mov rax,0x123 lea ecx,[rax*2+0x33332200]	<Register id="C" size="8">33332446</Register>
load effective address to WORD register, and cause overflow	The opcode variant to load the effective address of a SIB byte into a word register will be used, such that value is lost when doing so because of the capacity of a WORD.	mov rax,0x123 lea dx,[rax*4+0xfc00]	<Register id="D" size="8">8c</Register>

Justification of test data for lea

This test data is necessary as it demonstrates all the possible functions of the LEA instruction. This means that if the test case passes, it is certain that the opcode will work in the general case. For example, the events that take place when an overflow is present must be tested. This is because it is a boundary condition; it would not be unsurprising to have logical errors in this part of the code even if the first two testcases passed. For this reason, it is important to test the behaviour when this happens as otherwise underlying bugs could go unnoticed in the solution.

Mul

m	How	Test data	Expected result
ul two bytes in registers	The opcode variant to multiply two bytes in registers will be used.	mov bl,0x11 mov al,0x2 mul bl	<Register id="A" size="2">0022</Register> <Register id="D" size="2">0</Register> <Flag name="Carry">0</Flag> <Flag name="Overflow">0</Flag>
ul two words in registers	The opcode variant to multiply two words in registers will be used.	mov bx,0x2222 mov ax,0x2211 mul bx	<Register id="A" size="2">C842</Register> <Register id="D" size="2">048A</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
ul two words in register	The opcode variant to multiply two dwords in registers will be used.	mov ebx,0x33333333 mov eax,0x33332211 mul ebx	<Register id="A" size="4">8f5c2c63</Register> <Register id="D" size="4">0a3d6d36</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
ul a WORD register and a value in emory	The opcode variant to multiply a qword in the 'A' register and a qword in memory will be used.	mov rbx,0xb mov QWORD PTR [rsp],rbx movabs rax,0x4444444433332211 mul QWORD PTR [rsp]	<Register id="A" size="8">eeeeeeee333276bb</Register> <Register id="D" size="8">2</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>

Justification of test data for mul

This test data is necessary and adequate to test the mul opcode as all combinations of inputs have been tested that could have an affect on the result. For example, the all operands sizes have been tested, which would outline any errors present in the Bitwise.Multiply() method. Moreover, in testcase 1 it is tested whether the CF and OF are set correctly. This is necessary in order to achieve 1:1 emulation as stated in the success criteria, all the small details should be emulated exactly as they would outside of the program. The success criterion would not be fully met if this was not tested, as it would be uncertain whether the flag was being set correctly. The flag is set when the upper bytes of the results are used, which is not the case for $0x2 * 0x11$ because $0x22$ fits in one byte, whereas $0x2222 * 0x2211 = 048AC842$, does not fit in one word, hence CF and OF should be set.

Jmp/test/cmp

m	How	Test data	Expected result
est condition checking of CMP/TEST/ CMP opcodes	Include a short program in a testcase that will use the three opcodes in a context where the outputs will be wrong if they are not functional	mov rbp,rsp mov rcx,0x20 // counter dec rcx // Decrement counter mov BYTE PTR [rbp+rcx*1+0x0],cl test rcx,rcx // check if counter == -1 jg 0xa // jump to dec cmp ecx,0xffffffff // check if counter == -1 jne 0xa // jump to dec	<Memory offset_register="BP" offset="FF">FF000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F</Memory>

Justification of test data for JMP/TEST/CMP

This test data is important as it tests three opcodes that will all depend on a certain function, the TestCondition() method in the opcode base class. The testcase uses iteration with rcx as a counter. The loop will repeat $0x20$ times, twice. The loop comparison will first be performed with CMP, then TEST. The loop will execute until a memory error if JMP does not work. This shows that this testcase is necessary as any logical errors present in one of the opcodes would also be present in the others, therefore by using this test data to tackle the problems at the same time, there will be less time spent debugging multiple classes at once, as it will be obvious where the error lies.

Negate

m	How	Test data	Expected result
test the bitwise negation procedure	The testcase will contain assembly code that negates predefined values that can be then compared with known correct values	<pre> mov rax,0xfffffffffffffff neg rax mov ebx,0x11111111 neg rbx neg ecx </pre>	<Register id="A" size="8">1</Register> <Register id="B" size="8">ffffffffeeeeeeeef</Register>

Justification of test data for negate

This test data is necessary as there is no other opcode that uses the Negate() method of the bitwise library. Because of this, the problem needs to be tackled in a separate testcase in order to verify that the procedure is working as this will not be shown in other testcases. This test data will also verify that the opcode works with the two distinguishable inputs that exist for this function. This is zero and any other number. Zero is tested on the last line of the program as ECX is never assigned. This is necessary because it is a boundary condition, so unexpected problems could occur if this is not tested throughout development.

Bitwise OR

m	How	Test data	Expected result
R immediate to the 'A' register	Use the opcode variant of OR that implicitly uses the A register and an immediate	<pre> movabs rax,0xa5a5a5a5a5a5a5a5 or al,0x5a or ax,0x5a00 or eax,0x5a5a0000 or rax,0xffffffff80000000 </pre>	<Register id="A" size="8">fffffffffffff</Register>
R immediate to register	Use the opcode variant of OR that uses a register and an immediate	<pre> movabs rbx,0xa5a5a5a5a5a5a5a5 or bl,0x5a or bx,0x5a00 or ebx,0x5a5a0000 or rbx,0xffffffff80000000 </pre>	<Register id="B" size="8">fffffffffffff</Register>
R immediate to memory	Use the opcode variant of OR that uses an immediate and a location in memory	<pre> mov ebx,0x11 mov BYTE PTR [rsp],bl mov ebx,0x2233 mov WORD PTR [rsp+0x1].bx mov ebx,0x44556677 mov DWORD PTR [rsp+0x4],ebx movabs rbx,0x8899aabcccddeeff mov QWORD PTR [rsp+0x9],rbx or BYTE PTR [rsp],0xee or WORD PTR [rsp+0x1],0xddcc or DWORD PTR [rsp+0x4],0xbbaa9988 or QWORD PTR [rsp+0x9],0xffffffff3221100 </pre>	<Memory offset_register="SP">FF</Memory> <Memory offset_register="SP" offset="1">FFFF</Memory> <Memory offset_register="SP" offset="4">FFFFFFF</Memory> <Memory offset_register="SP" offset="9">FFFFFFFFFFFFFF</Memory>
R register to memory	Use the opcode variant of OR that uses a register and a location in memory	<pre> mov ebx,0x11 mov BYTE PTR [rsp],bl mov ebx,0x2233 mov WORD PTR [rsp+0x1].bx mov ebx,0x44556677 mov DWORD PTR [rsp+0x4],ebx movabs rbx,0x8899aabcccddeeff mov QWORD PTR [rsp+0x9],rbx mov bh,0xee or BYTE PTR [rsp],bh mov bx,0xddcc or WORD PTR [rsp+0x1],bx mov ebx,0xbbaa9988 or DWORD PTR [rsp+0x4],ebx movabs rbx,0x7766554433221100 or QWORD PTR [rsp+0x9],rbx </pre>	<Memory offset_register="SP">FF</Memory> <Memory offset_register="SP" offset="1">FFFF</Memory> <Memory offset_register="SP" offset="4">FFFFFFF</Memory> <Memory offset_register="SP" offset="9">FFFFFFFFFFFFFF</Memory>

Justification of test data for OR

This test data is necessary to test the behaviour of the logical operators in the program. This will apply to Bitwise.AND() and Bitwise.XOR() also because they use the same algorithm but with a different operator. Therefore, by using this test data, it is possible to apply a single testing procedure that will test multiple functions of the program accurately, such that if a bug is found in one, it can be fixed in the other methods, so will save development time testing all of the procedures separately.

Bit shifts

m	How	Test data	Expected result
test the shift left algorithm	The SHL opcode will be used which uses the ShiftLeft() bitwise library method. The input will be crafted to only use the bare features of the algorithm, e.g. no OF or CF used.	<pre>mov cl,0x8 mov rax,0xff mov rbx,rax shl bl,1 mov rdx,rax shl dx,cl mov QWORD PTR [rsp],rax shl QWORD PTR [rsp],0x3f</pre>	<Register id="B" size="1">FE</Register> <Register id="D" size="2">FF00</Register>
test the setting of the OF in the shift left algorithm	The SHL opcode will be used which uses the ShiftLeft() bitwise library method. The input will be crafted to cause the OF to be set in the output.	<pre>mov rbx,rax shr bl,1</pre>	<Register id="B" size="1">7F</Register> <Flag name="Overflow">1</Flag>
test the setting of the CF in the shift left algorithm	The SHL opcode will be used which uses the ShiftLeft() bitwise library method. The input will be crafted to cause the CF to be set in the output.	<pre>mov rdx,rax shr dx,cl</pre>	<Register id="D" size="1">0</Register> <Flag name="Carry">1</Flag>
test the setting the shift right algorithm	The SAR opcode will be used which uses the ShiftRight() bitwise library method. The input will be crafted to only use the bare features of the algorithm, e.g. no OF or CF used.	<pre>mov rdx,rax sar bl,1</pre>	<Register id="B" size="1">3f</Register>
test the of the CF in the shift right algorithm	The SAR opcode will be used which uses the ShiftRight() bitwise library method. The input will be crafted to set the CF in the output.	<pre>mov rdx,rax sar dx,cl mov QWORD PTR [rsp],rax sar QWORD PTR [rsp],0x4</pre>	<Register id="D" size="1">0</Register> <Flag name="Carry">1</Flag>

Justification of test data for bit shifts

This test data is necessary as it will test the shifting algorithms in both directions. It also allows for the two tests to be combined together as the algorithms to do so are very similar, so a bug in one would suggest the other also needs inspection. Because of this, less time will be spent analysing the testcase results and responding to any bugs because the relevant information will be grouped together, such that the process of finding the bug can be found in a single testcase rather than tracking many.

CBW/CWDE/CDQE

Test #	Aim	How	Instructions used as input	Expected result

1	Test converting a byte into a word	The CBW opcode will be used as an instruction.	(Used throughout entire testcase) mov ebx,0x80008080 mov al,bl cbw	<Register id="A" size="2">FF80</Register>
2	Test converting a word into a dword	The CWDE opcode will be used as an instruction.	mov ax,bx cwde	<Register id="A" size="4">FFFF8080</Register>
3	Test converting a dword into a qword	The CDQE opcode will be used as an instruction.	mov eax,ebx cdqe	<Register id="A" size="8">ffffffff80008080</Register>

IMUL

Test #	Aim	How	Instructions used as input	Expected result
1	Test signed multiplication of a byte	The IMUL instruction will be used with two byte registers, one implicitly being the 'A' register.	mov bl,0x11 mov al,0x2 imul bl	<Register id="A" size="2">0022</Register> <Register id="D" size="2">0</Register> <Flag name="Carry">0</Flag> <Flag name="Overflow">0</Flag>
2	Test signed multiplication of a word	The IMUL instruction will be used with two word registers, one implicitly being the 'A' register.	mov bx,0x2222 mov ax,0x2211 imul bx	<Register id="A" size="2">C842</Register> <Register id="D" size="2">048A</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
3	Test signed multiplication of a dword from memory	The IMUL instruction will be used with a memory and implicitly the 'A' register	mov ebx,0x33333333 mov DWORD PTR [rsp],ebx mov eax,0x33332211 imul DWORD PTR [rsp]	<Register id="A" size="4">8f5c2c63</Register> <Register id="D" size="4">0a3d6d36</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
4	Test signed multiplication of a qword	The IMUL instruction will be used with two qword registers, one implicitly being the 'A' register.	mov rbx,0xb movabs rax,0x444444443332211 imul rbx	<Register id="A" size="8">eeeeeeee333276bb</Register> <Register id="D" size="8">2</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
5	Test signed multiplication of two registers(not implicit A register)	The IMUL instruction will be used with two registers of all sizes.	movabs rax,0x444444443332211 mov bx,0x2222 imul bx,ax mov ecx,0x33333333 imul ecx,eax movabs rdx,0x4444444444444444 imul rdx,rax	<Register id="B" size="8">c842</Register> <Register id="C" size="8">08f5c2c63</Register> <Register id="D" size="8">320fedcbf259084</Register>
6	Test signed multiplication of a reg and immediate, output into a different register(3 operand variant)	The IMUL instruction will be used with an immediate multiplied by a register, and output into a different register..	movabs rax,0x444444443332211 imul bx,ax,0x3 imul ecx,eax,0x3 mov QWORD PTR [rsp],rax imul rdx,QWORD PTR [rsp],0x3	<Register id="B" size="8">6633</Register> <Register id="C" size="8">99996633</Register> <Register id="D" size="8">cccccc99996633</Register>

Negate

Test #	Aim	How	Instructions used as input	Expected result
1	Test negation of dword and qword registers.	Move data into two separate registers then use the negate opcode on them. (Applicable to use one single testcase as not many registers are involved).	mov rax,0xffffffffffff neg rax mov ebx,0x11111111 neg rbx neg ecx	<Register id="A" size="8">1</Register> <Register id="B" size="8">ffffffffeeeeeeef</Register> <Flag name="Carry">1</Flag>

		ECX is empty and negated to test that the CF is set.		
--	--	--	--	--

Push and pop

Test #	Aim	How	Instructions used as input	Expected result
1	Push/pop word and qword at a pointer in memory	Push the data onto the stack at a pointer to a byte/word using the opcode variant that performs this task. (The specific sizes of memory are used because pop/push only has variants for these specified sizes)	movabs rax,0aaaaaaaaaaaaaaaaaaaa push rax ; (Push from RIP) push WORD PTR [rip+0x0] push QWORD PTR [rip+0x0] pop rax pop bx	<Register id="A" size="8">2918B848905B6658</Register> <Register id="B" size="8">35ff</Register> <Memory offset_register="SP" offset="F6">58665B9048B81829FF35</Memory>
2	Push/pop register word and qword	Push the data onto the stack from a word/qword register using the opcode variant that performs this task.	movabs rax,0x4f4e4d4c3b3a2918 push ax push rax pop rbx pop cx	<Register id="B" size="8">4f4e4d4c3b3a2918</Register> <Register id="C" size="8">2918</Register>
3	Push/pop immediate byte, word, and dword	Push the data onto the stack from a word/qword immediate using the opcode variant that performs this task.	push 0xffffffffffff2 pushw 0x1234 push 0x12345678 pop rax pop bx pop rcx pop rdx	<Register id="A" size="8">12345678</Register> <Register id="B" size="2">1234</Register> <Register id="C" size="8">ffffffffffff2</Register> <Register id="D" size="8">aaaaaaaaaaaaaaaaaa</Register> <Memory offset_register="SP" offset="F8">aaaaaaaaaaaaaaaaaa</Memory> <Register id="SP" size="8">800000</Register>

Ret and call

Test #	Aim	How	Instructions used as input	Expected result
1	Test whether call and ret jump to and from the called function correctly	A small function was created to use in the test data that is called at the beginning of the program. If the functions returns as expected(0xA0 in the A register), the testcase passed.	mov rbx,rspl add rbx,0xffff mov rax,0x80 push rax call 0x1b cmp rbx,rspl nop pop rbp pop rax add rax,0x20 push rbp ret 0xffff	<Register id="A" size="8">A0</Register> <Flag name="Zero">1</Flag> <Flag name="Parity">1</Flag>

Rotate

Test #	Aim	How	Instructions used as input	Expected result
1	Test left rotation of the A register, cl times	The variant of the ROL opcode that implicitly uses the CL register as the rotation count will be used in the test data..	; Setup for testcases 1-4 movabs rax,0x4444444433332211 mov ebx,eax mov WORD PTR [rsp],ax mov dh,ah ; mov cl,0x44 rol rax,cl	<Register id="A" size="8">4444444433332211</Register><Flag name="Carry">0</Flag>
2	Test left rotation of a dword, when the number of rotations is stored in an immediate	The variant of the ROL opcode that rotates a register by an immediate will be used in the test data.	rol ebx,0x10	<Register id="B" size="8">22113333</Register><Flag name="Carry">1</Flag>
3	Test left rotation of value in memory pointed to by address	The variant of the ROL opcode that uses a pointer in memory and rotates by 1 will be used in the test data.	rol WORD PTR [rsp],1	<Memory offset_register="SP">2244</Memory><Flag name="Overflow">0</Flag><Flag name="Carry">0</Flag>
4	Test left rotation of a full rotation loop(result same as input)	The ROL opcode will be used such that the entire register is rotated back to its original value.	rol dh,0x4	<Register id="D" size="2">2200</Register><Flag name="Carry">0</Flag>
5	Test left rotation of register using carry in rotation pool	The variant of the RCL opcode that implicitly uses the CL register as the rotation count will be used in the test data..	; Setup for tests 5-8 stc movabs rax,0x4444444433332211 mov ebx,eax mov WORD PTR [rsp],ax mov dh,ah ; mov cl,0x44 rcl rax,cl	<Register id="A" size="8">4444444433332211a</Register>
6	Test left rotation of a dword, when the number of rotations is stored in an immediate, with the carry flag in the pool	The variant of the RCL opcode that rotates a register by an immediate will be used in the test data.	rcl ebx,0x10	<Register id="B" size="8">22111999</Register><Flag name="Carry">1</Flag>
7	Test left rotation of value in memory pointed to by address, with the carry flag	The variant of the RCL opcode that uses a pointer in memory and rotates by 1 will be used in the test data. The CF is set by the previous test.	rcl WORD PTR [rsp],1	<Memory offset_register="SP">2344</Memory><Flag name="Overflow">0</Flag><Flag name="Carry">0</Flag>

	in the pool.			
8	Test left rotation of value stored in upper register with CF in pool	The RCL opcode will be used such that the entire register is rotated back to its original value.	rcl dh,0x4	<Register id="D" size="2">2100</Register><Flag name="Carry">0</Flag>
9	Test right rotation of the A register, cl times	The variant of the ROR opcode that implicitly uses the CL register as the rotation count will be used in the test data..	<pre> movabs rax,0x4444444433332211 mov ebx,eax mov WORD PTR [rsp],ax mov dh,ah mov cl,0x44 ror rax,cl </pre>	<Register id="A" size="8">144444443333221</Register><Flag name="Carry">0</Flag>
10	Test right rotation of a dword, when the number of rotations is stored in an immediate	The variant of the ROR opcode that rotates a register by an immediate will be used in the test data.	ror ebx,0x10	<Register id="B" size="8">22113333</Register>
11	Test right rotation of value in memory pointed to by address	The variant of the ROR opcode that uses a pointer in memory and rotates by 1 will be used in the test data.	ror WORD PTR [rsp],1	<Memory offset_register="SP">0891</Memory><Flag name="Overflow">0</Flag><Flag name="Carry">1</Flag>
12	Test right rotation of a full rotation loop(result same as input)	The ROR opcode will be used such that the entire register is rotated back to its original value.	ror dh,0x4	<Register id="D" size="2">2200</Register><Flag name="Carry">0</Flag>
13	Test right rotation of register using carry in rotation pool	The variant of the RCR opcode that implicitly uses the CL register as the rotation count will be used in the test data..	<pre> stc movabs rax,0x4444444433332211 mov ebx,eax mov WORD PTR [rsp],ax mov dh,ah mov cl,0x44 rcr rax,cl </pre>	<Register id="A" size="8">344444443333221</Register>
14	Test right rotation of a dword, when the number of rotations is stored in an immediate, with the carry flag in the pool	The variant of the RCR opcode that rotates a register by an immediate will be used in the test data.	rcr ebx,0x10	<Register id="B" size="8">44223333</Register>
15	Test right rotation of value in memory pointed to by address, with the	The variant of the RCR opcode that uses a pointer in memory and rotates by 1 will be used in the test data. The CF is set by the	rcr WORD PTR [rsp],1	<Memory offset_register="SP">0811</Memory><Flag name="Overflow">0</Flag><Flag name="Carry">1</Flag>

	carry flag in the pool.	previous test.		
16	Test right rotation of value stored in upper register with CF in pool	The RCR opcode will be used such that the entire register is rotated back to its original value.	rcr dh,0x4	<Register id="D" size="2">5200</Register><Flag name="Carry">0</Flag>

SetCC

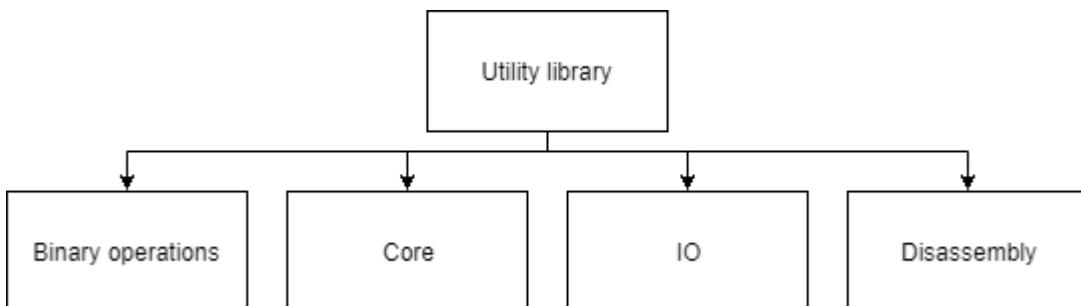
Test #	Aim	How	Instructions used as input	Expected result
1	Test whether SetCC sets the specific byte based on a condition successfully	The SetCC operand will be used in the input instructions. This only has to be tested with a couple of condition codes because the condition testing methods will be tested separately.	<pre> mov bl,0x10 cmp bl,0x10 sete BYTE PTR [rsp] mov bl,0xff cmp bl,0x10 setge BYTE PTR [rsp+0x1] </pre>	<Memory offset_register="SP">0100</Memory>

Xchg

Test #	Aim	How	Instructions used as input	Expected result
1	Test that both register values are swapped when using the XCHG opcode	The instructions containing the XCHG opcode will be the input. They will swap around the registers a few times, then the outcome will depend on whether the results match the expected results.	<pre> movabs rax,0x4444444433332211 mov bl,0x44 mov ch,0x44 mov DWORD PTR [rsp],eax xchg bx,ax xchg ch,ah xchg rdx,rax xchg DWORD PTR [rsp],eax </pre>	<Register id="A" size="8">33332211</Register> <Register id="B" size="2">2211</Register> <Register id="C" size="2">0</Register> <Register id="D" size="8">4444444433334444</Register> <Memory offset_register="SP">0</Memory>

Utility library

Structure and decomposition



The utility library will be a pretested library of routines to be used to solve common problems throughout the program that have been identified through the stepwise refinement. The utility library will consist of four sub-libraries: Binary Operation, Core, IO, and Disassembly. Another attribute of the utility library will be modularity; each sub-library will be re-usable in other projects that also have problems in common.

Explanation

Each sub-library will cater to a specific set of problems based on the nature of the problem, but not tied to a particular layer. For example, the Disassembly library will provide routines and methods for disassembling instructions, which would mostly be used by the Processing layer. However, if the Interface layer had a particular reason to do so, it would still be conventional for it to do so. This is because the purpose of the Utility Library is to solve common problems to simplify the solution, therefore the backflow of layers concept does not apply here.

To enforce modularity, a general solution will be applied to the problems, rather than a solution specific to this program. For example, consider the following scenario. A variable called Name in class MyClass is required in the utility routine Welcome().

```
DEF Welcome()
    print("Welcome, " MyClass.Name);
FED;
```

The figure shows pseudocode that would function in the program, but would not function if it was moved to another program. Instead, the following approach would be taken,

```
DEF Welcome(string InputName)
    print("Welcome, " + InputName);
FED;
```

This approach will be taken to every routine, favouring extra inputs over hard-coded variable usage. This will allow the code to be reused in other projects, and would only have to provide the correct input to the function. I will refer to this approach as a “parameter-heavy” approach.

Justification

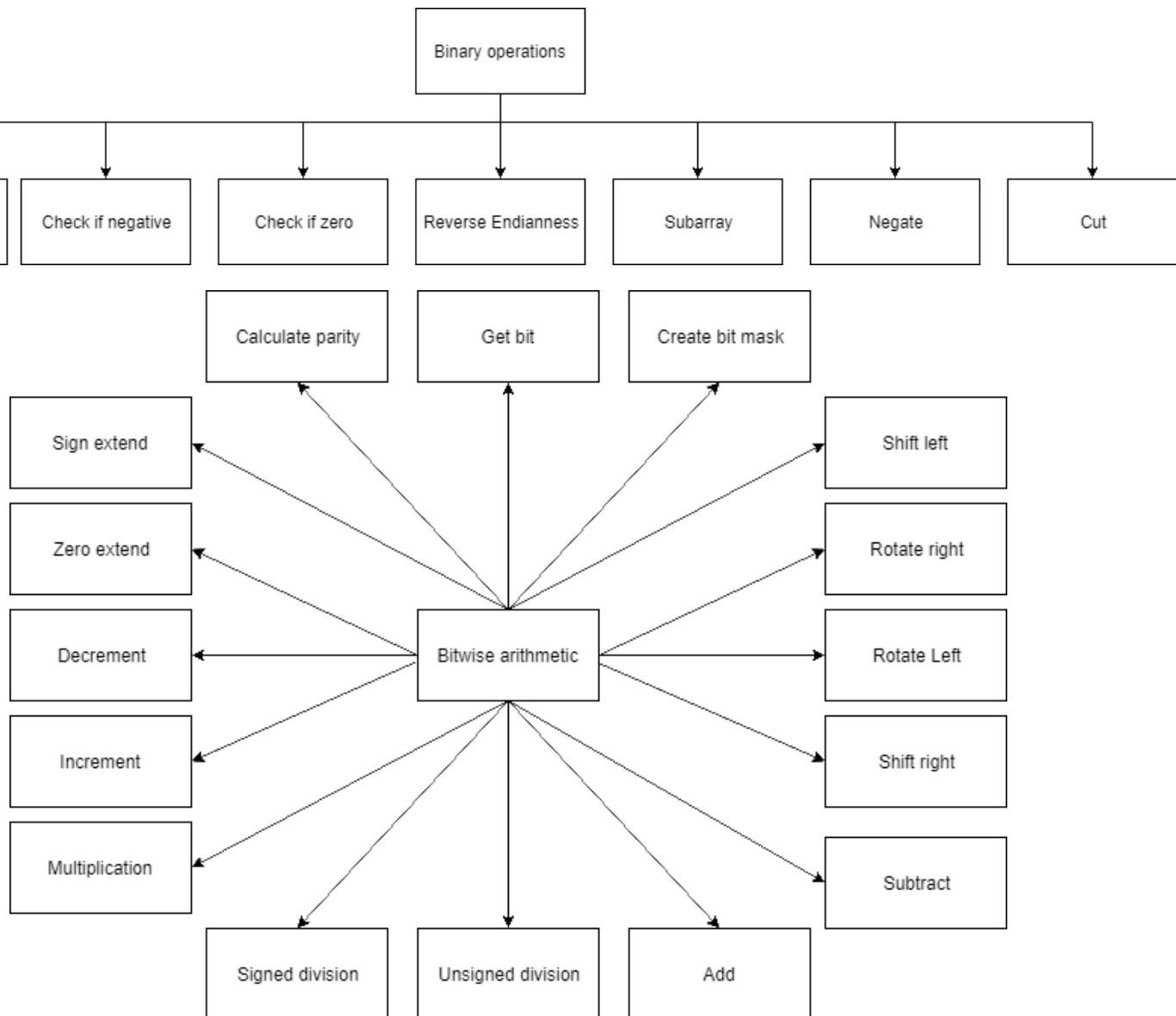
By not restricting the libraries to a specific layer and taking the parameter heavy approach, the problems that occurred multiple times in the stepwise refinement will be all solvable by a single general solution.. This is necessary because solutions will only have to be tested once, albeit a thorough test. For example, if a routine is thoroughly tested in the utility library and determined fit for purpose, it can be left untouched until there is a particular change in the purpose of the method. This links back to the Single Responsibility Principle as part of the success criteria. A class should only have one reason to change, which would be when the purpose of the function has changed. This will make the program more robust because it will ensure that the solutions are specific to the problems and do not aim to solve more than one problem. This will keep code maintainable, as the code will follow an algorithm and be intuitive, rather than an incomprehensible mess. This also supports the justification for a utility library, as problems can be decomposed into further sub-problems, which each have a dedicated function to solve them in the library. This is a better approach than having too many functions spread across classes, where routines would be hard to locate methodically by another developer who is reviewing the code.

Moreover, having a pretested library of routines developed from the beginning will establish a “known good” throughout the development process. This means that I will have a consistent solution to refer back to once improvements and refactorings are made to cross reference when developing improved solutions. It is easier to make a correct program fast than a fast program correct[1], hence by thinking ahead and planning for future revisions of code, development time will be saved both during and post development.

Another benefit due to the library is that improvements to the library routines will be seen across the whole program where the library routines are used. If this were not the case, optimisations and refactorings would have to be made on a case-by-case basis for each class, which would exponentially increase development time as opposed to developing and maintaining the general solution stored in the utility library and improving that.

Bitwise library

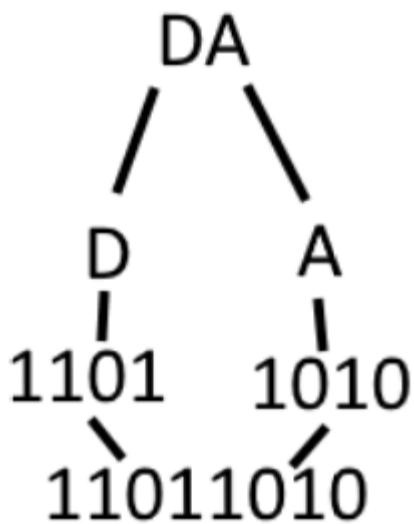
Explanation



The bitwise library will provide predefined solutions to many bitwise arithmetic operations, and other procedures that may be relevant. The procedures be independent of any layers of abstraction, rather will depend on the solution wide data type abstractions, such as a FlagSet, and C# primitives such as byte arrays. Byte arrays will be used to store numbers instead of integer types. Essentially, numbers will be stored in base-256. This is identical to how numbers work in hex, except split up byte by byte and stored in little endian.

Dec	2222
Hex	5CA140
Byte[]	[0x40, 0xA1, 0x5C]

This will work exactly like how hex digits in a hex number can be split up, then joined back together in base 2. For example,



This exact same procedure can be applied except on a larger scale because the lowest column of units goes up to 0xFF not 0xF.

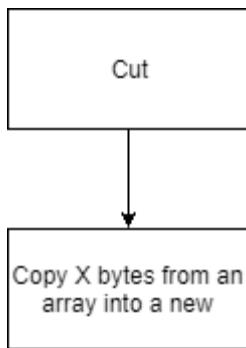
To avoid confusion when comparing two boolean variables, a boolean “equals” will be referred to more formally as XNOR.

A	B	A XNOR B
1	0	0
1	1	1
0	0	1
0	1	0

Justification

The Bitwise library is necessary as part of the Single Responsibility Principle success criterion. For example, the compare(CMP) and subtract(SUB) opcode classes will depend on subtracting two byte arrays. This problem can be solved in both cases by creating the subtract routine in the utility library, then both call the function there. such that the code only has to be created once. This is the best approach to the re-occurring problems because the code in the bitwise class is likely to change often; a better arithmetic algorithm may be found. To deploy the new algorithm faster, such that it only has to be tested and developed in one place, having the callers depend on the library class than each have a separate implementation is a better option. Storing numbers as byte arrays is the better option than to use integer types such as ints and longs because it allows for a flexible input size, as would be likely in an assembly program; adding EAX and adding AX would be two separate methods otherwise, as EAX has the capacity of “int”, and AX has the capacity of a “short”. This allows for more maintainable methods as instead of having to maintain and develop a method for each size of data type, the procedure can be combined into one. It will also allow numbers greater than the maximum size of a ulong to be operated on. For example, the MUL opcode can add two 16 byte numbers. This would not be possible without using a byte[]. The bytes should be stored in little endian format because the intel specification states, “ Intel 64 and IA-32 processors are “little endian” machines; this means the bytes of a word are numbered starting from the least significant byte”. The best option will be to go with what is said in the specification rather than create routines that use big endian and convert afterwards as this adds unnecessary steps in solving the solution and the solution will be harder to maintain because parts will be in big endian and other parts in little endian.

Cut algorithm



Pseudocode

```

DEF Cut(byte[] input, int newSize)
    byte[] Result = new byte[newSize];
    int i = 0;

    // Need to stop if would go out of array bounds, or if desired size reached
    FOR i < newSize AND i < input.Length DO
        Result[i] = input[i];
        i++;
    ROF;
    RETURN Result;
FED;

```

Explanation

The cut method will cut out the beginning N bytes of an array into a new array, or zero where there are no more bytes. This function will be necessary for making sure user input meets a specific length. The desired size of the input can be “cut” to the correct length to ensure other procedures will work.

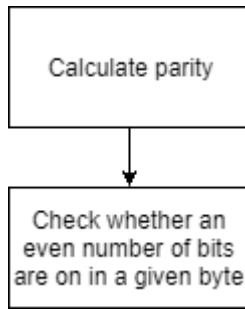
Justification

This algorithm will provide a complete solution to the problem because the task of copying bytes from the input array into the result array has been solved. This is because there will never be a case where the desired length is not met; if the input is less, the result array was initialised to “newSize” not `input.Length`, hence will always be the correct size except with trailing zeroes.

Iterative testing

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
ut a byte array	Hardcode a call to the function with set parameters that can be checked after the test. A breakpoint will be set after the function call to test this.	The input array will be size 4 and all the bytes will be equal to 0xAA, newSize will be 2.	Normal	Output = [0xAA, 0xAA]	This test is necessary to make sure the function works with normal data. This is important to make it possible to deduce whether erroneous results in other tests are because of a fundamental design error or only bugs; to prove that the function works at all.
ut a byte array with erroneous inputs	The function will be called with a byte array that has a size lower than the input size.	The input array will be of size 4, the newSize parameter will be 5.	Erroneous	Output = input	This test is necessary to make sure that the size check in the if condition is coded properly, such that there will not be an error. The FOR loop should never be entered because the condition is not met. This test will show that it is entered if there is an error.

GetParity algorithm



Pseudocode

```

DEF GetParity(byte input)
    // Start assuming even parity
    bool Even = true;

    // Iterate through every bit in the byte. This is done by using the counter to create a
    // bitmask
    int i = 0;
    FOR i < 8 DO
        IF (input AND (1 << i)) > 0 THEN
            Even ^= true;
        FI;
        i += 1;
    ROF;
    RETURN Even;
FED;

```

Explanation

GetParity will return true if and only if there are an even number of bits in the byte and false otherwise. This can be checked by iterating through each bit in the byte and flipping the value of even every time. In other words, if Even is true, then XORed by true, it will become false. If Even is false, then XORed by true, it will become true. The bits can be iterated by using the counter to create a bitmask every iteration. The bitmask is created by shifting one by the counter(which starts at zero therefore no further action is required), then used to AND input. This means that the only bit that could possibly be set after the AND is the bit that is \$i places from the start of the byte, the \$i+1th bit(starting from one not zero). This method will be used by the FlagSet struct to apply the generic procedure of checking parity to the input byte array, as many operations will set the parity flag automatically based on the same conditions that this algorithm determines.

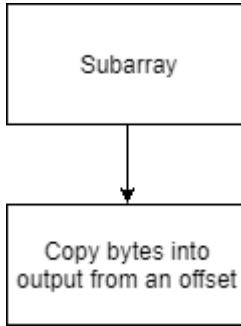
Justification

This is a complete solution to the problem as a computational solution has been applied that will work for any case. The even parity is determined by whether the number of bits set on in the byte are even. By initialising even to be true, this algorithm will work if there are no bits set on in the byte at all, as zero is an even number the function will return true; the even boolean will be unchanged after its initial assignment. Moreover, by iterating through every bit in the byte and flipping the value of even every time the bit is on, the IF statement uses a greater than condition which will therefore work for any position of the bit in the byte as these values are not deweighted; they are still the powers of two they represent, not the desired on or off state. The on or off state can be determined by checking if it is greater than zero, as any bit set will have a value greater than zero.

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test for even parity	The input to the function will be a known even parity. A breakpoint after the function call will allow me to check whether the output is true.	Input = 5 5d = 0101b, 2 ones is even parity.	Normal	Output = true	This test is necessary to make sure that the function will return a value and that there are no runtime errors caused by incorrect code. Other tests will not be reliable if basic functionality is not confirmed.

Test for odd parity	The input to the function will be a known odd parity. A breakpoint after the function call will allow me to check whether the output is false.	Input = 4 4d = 0100b, 1 one is odd parity	Normal	Output = false	This test is necessary to make sure that the even value is being affected. If only Test #1 was performed, it is not certain that the function works because there is the possibility of a logical error in the if conditional statement that would cause the even boolean to never change.
Test for even parity when input zero	The function will be called with a zero. A breakpoint can be used to check after that the output is true.	Input = 0 0 = 0000b, 0 ones is even parity	Boundary	Output = true	This test is necessary to make sure that the function works properly on the boundary condition. This is because the code in the IF statement should never be executed when the input is zero, the boolean should stay true until the end of the function.

Subarray algorithm



Pseudocode

```

DEF Subarray(byte[] input, int offset)
    // The length of bytes returned will be the length minus the offset, because
    // $offset number of bytes are skipped
    byte[] Result = new byte[input.Length - offset];
  
```

```

    // Start at the beginning of the results array
    int index = 0;
  
```

```

    // Read then set every byte after the index
    FOR index < Result.Length DO
  
```

```

        // The input array must be offset by the value of the input offset, otherwise
        // the bytes will be read from the beginning of the array
        Result[index] = input[index + offset];
    
```

```

    ROF;
  
```

```

    RETURN Result;
  
```

FED;

Complexity

O(n) Theta(n) Omega(1) - The algorithm will only need to iterate the remaining bytes in the input, therefore the best case would be that there is one byte to read, the worst case is that the entire input needs to be iterated

Explanation

The algorithm works like a subarray of a string. After and including a given offset index to start on, all remaining bytes will be added onto an output. This method will be useful for dealing with transforming a byte array to split across multiple registers. For example, the unsigned multiply(MUL) opcode has two registers it outputs to, the A register and the D register, where the lower bytes of the result go to the A register and the upper to the D register. This method will be useful at taking the upper bytes from the result, starting at half way, which can then be used to set the D register.

Having this in the utility library will allow other opcodes that use the same procedure, such as DIV, to also use this solution without having to code it multiple times.

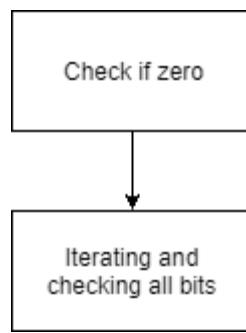
Justification

This algorithm provides a full solution to the problem because the task of copying bytes into the output from an offset has been completed by iteratively copying the bytes from the input array at an offset into the result array.. If the offset is outside of the array, there will be an error. The best approach is to leave let this happen because there is no sensible procedure to work around the invalid input; it is best that the developer is told so they can fix their code.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
check that the function works for normal input	The function will be called with non-erroneous input and a breakpoint afterwards will be used to check that the output is correct.	input = [AA,AA,FF,FF] offset = 2	Normal	output = [FF,FF]	This test is necessary to make sure that the function works under normal conditions. This will test for logical errors such as the offset not being used correctly or incorrect values from the array not being copied into the result.
check that an error is thrown when erroneous parameters are used	The function will be called with an offset greater than the length of the byte array.	input = [AA], offset = 1	Erroneous	ArrayOutOfBoundsException	This test is necessary to make sure that the exception is thrown when erroneous parameters are entered, otherwise bugs may show up in the development process where I expected the method to tell me of erroneous input, but did not.

IsZero algorithm



Pseudocode

```

DEF IsZero(byte[] input)
    int i = 0;
    // Iterate every element in the array
    FOR i < input.Length DO
        // If it is not zero, return false, because the value of the array will definitely not be
        // zero
        IF input[i] != 0 THEN
            RETURN false;
        FI;
        i++;
    ROF;
    RETURN true
FED;

```

Complexity

O(n) Theta(n) Omega(1) - The algorithm will exit immediately once a non-zero is found. The best-case scenario would be if the first element in the array is nonzero, the time will be 1. The worst and average case would be in linear time as the iterations would be linear.

Explanation

The algorithm works by checking every element of the array, checking if it is zero. If the entire array was iterated, it would be true because the FOR loop would have returned earlier otherwise.

Justification

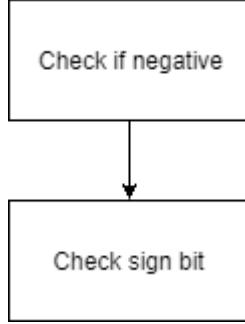
This algorithm is a complete solution to the problem because the sub-problem of iterating and checking all bits has been solved. This has been solved by using a for loop to iterate through every index of the array and check if the value is zero. This is a complete solution to the problem because in unsigned and signed two's complement numbers(which are the two methods of signedness permitted in the x86-64 specification), both have one value for zero, zero, as opposed to one's complement which has two, but is not part of the x86-64 specification therefore the only valid input would be two's complement signed numbers and unsigned numbers.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test to see if zero valued inputs output true	A byte array filled with zeroes will be used as the input and the result will be checked using a breakpoint.	input = [00, 00]	Normal	True	This test is necessary to make sure that the evaluate zero values properly, as other tests will not be reliable if the function does not work with a normal input because this would likely be the cause of any other erroneous behaviour
test to see if non-zero valued inputs output false	A byte array filled with non-zero values will be used as the input and the result will be	input = [AA, AA]	Normal	False	This test will make sure that the IF statement is functional, such that there is no logical in the condition code that would make the function always

	checked using a breakpoint.				output true(zero).
test to see if boundary non-zero valued inputs output false	A byte array filled with zeroes, but the last byte a non-zero will be used and the result will be checked using a breakpoint.	input = [00, 00, 00, AA]	Boundary	False	This test data is necessary to make sure that the entire array is iterated and detect any logical errors such as the condition in the FOR loop being incorrect.

IsNegative algorithm



Pseudocode

```

DEF IsNegative(byte[] input)
    // If the MSB is set(the sign bit), it has the potential to be a negative.
    IF MSB(input.Length[input.Length-1]) == 1 THEN
        // If one is subtracted from a power of two(acceptable word sizes for
        // a two's complement number), and the result has no bytes set in
        // common, the input was a power of two, therefore must be a
        // signed negative.
        IF (input.Length AND input.Length-1) == 0 THEN
            RETURN true;
        FI;
    FI;
    RETURN false;
FED;
  
```

Explanation

There are two conditions to determine whether an input is a negative in any case. The sign bit must be set and the input must be a power of two. An input whose size is not a power of two cannot be defined as a signed negative by definition.

Justification

This is a complete solution to the problem because the sub-problem of checking the sign bit has been solved, hence a negative will be detected in every case. This had to be extended in practice a little bit to cover inputs that could be invalid, where the sign bit has been cut out of the input.

Two's complement tells us that if the sign bit is set, the value is negative. The second condition is necessary because the actual size of the input must be known when determining whether a value is a signed negative or not, which in this program should only be a word size as defined by the RegisterCapacity enum. This is also defined in the Intel x86-64 manual, "Signed integers are signed binary numbers held in a byte, word, doubleword, or quadword. All operations on signed integers assume a two's complement representation. The sign bit is located in bit 7 in a byte integer, bit 15 in a word integer, bit 31 in a doubleword integer, and bit 63 in a quadword integer". This means that an input isn't a word size(a power of two <= 8), the caller has an incomplete array and so the best course of action is to assume that the array was supposed to be the next word size up but zeroes had been cropped off, therefore cannot be a negative because the sign bit in the input is not truly the intended sign bit. The method of taking one to determine a power of two can be proven by looking at the pattern of known powers of two,

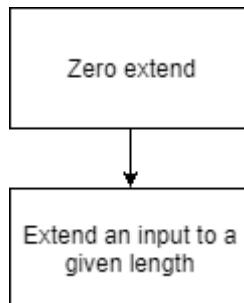
8	4	2	1	
<hr/>				
1	0	0	0	-1
0	1	1	1	

A number one less than a power of two will have all bits before said power of two set, therefore the AND of the two values will be zero.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test a signed negative outputs true	A signed negative will be entered as the input parameter	input = [FF, FF]	Normal	True	This test is necessary to make sure that the function works under normal conditions as other tests could not be reliable if the algorithm does not work on valid input.
test a signed positive outputs false	A signed positive will be used as the input parameter	input = [FF, 0F]	Normal	False	This test will make sure that the algorithm for detecting the sign of the input works for both negatives and positives. Test #1 would not reveal the possibility that true is output every time due to a logical error.
test an erroneous put with a size that not a multiple of two	An input that has the MSB set but is not a valid two's complement signed integer will be used.	input = [FF, FF, FF]	Erroneous	False	This test is necessary to ensure that the second IF condition(nested condition) correctly detects when the input is not a size that is a multiple of two.

ZeroExtend algorithm



```

DEF ZeroExtend(byte[] input, byte newLength)
// Leave the input as it was given if its length is not less than newLength
IF input.Length < newLength THEN
    // Assume that the byte is initially all zeroes
  
```

```

byte[] Result = new byte[newLength];

// Copy the bytes from the input into the result
int i = 0;
FOR i < input.Length DO
    Result[i] = input[i];
    i += 1;
ROF;

RETURN Result;
ELSE
    RETURN input;
FI;
FED;

```

Explanation

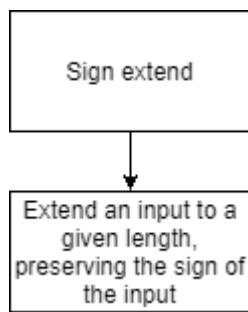
This algorithm works by creating a new array of the specified length then copying the bytes from the input into the new array, then returning the new array.

Justification

This is a complete solution to the problem because solves the task described in the decomposition. The input is extended to the given length and retains its former value, so long as it is an unsigned number. It will also work for a signed positive, however SignExtend would be the better option for signed numbers, and ZeroExtend used for unsigned numbers. This is because the sign bit isn't preserved, such that if a signed negative was zero extended, it would now be a large positive instead.

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Test that a normal input will produce the correct output	An input array with a newLength greater than the length of the input array will be used as parameters.	input = [AA, AA] newLength = 4	Normal	Output = [AA, AA, 00, 00]	This test is necessary to make sure that the function works under normal conditions as other tests will not be reliable if an error is innately caused by the algorithm rather than
Test that an erroneous input will output the same as the input	An input array with newLength less than the length of the array will be used as parameters.	input = [AA, AA] newLength = 1	Erroneous	Output = [AA,AA]	This test will make sure that the if condition at the start of the procedure will detect this erroneous condition and return only the input, instead of executing the code in the for statement and causing an ArrayOutOfBoundsException in the FOR loop as input.Length will be greater than Result.Length
Test that boundary input is unchanged	An input array with newLength equal to the length of the array will be used as parameters.	input = [AA,AA] newLength = 2	Boundary	Output = [AA,AA]	This test is necessary to check that when the length of input and newLength is equal, the input will be left unchanged. This should be caught by the condition at the start of the routine that checks if input.Length is less than newLength, however a logical error may cause this to not be the case.

SignExtend algorithm



Pseudocode

```

DEF SignExtend(byte[] input, byte newLength)
    // Leave the input as it was given if its length is not less than newLength
    IF input.Length < newLength THEN
        // The index to start writing the new bytes is at the end of the string such
        // that the bytes in the input are not overwritten, this is one more than the
        // greatest index because it is zero based, rather than the Length which starts
        // at one.
        int startIndex = input;
        // First zero extend to the new size
        input = ZeroExtend(input, newLength);

        // If the sign of the input is negative, the zeros need to be changed to
        // 0xFFs to preserve the negative value. No further action is required if the
        // input was positive; the procedure was the same as zero extension in this
        // specific instance.
        IF IsNegative(input) THEN
            // Start at the first byte of the resized bytes.
            int i = startIndex;
            // Replace the new bytes with 0xFF(all bits set)
            FOR i < newLength DO
                input[i] = 0xFF;
                i += 1;
            ROF;
        FI;
    FI;
    return Input;
FED;

```

Explanation

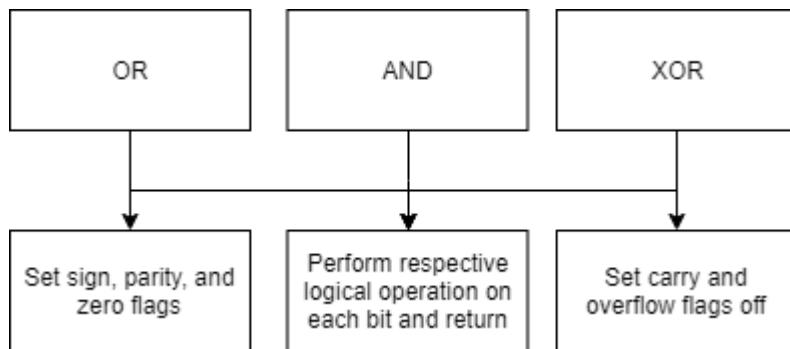
This algorithm works by first zero extending the input, then checking afterwards if the input needs have its sign changed to match the input by reusing the `IsNegative()` function, which returns true if the input is negative. This means that if a negative number is sign extended, it will have the same value as the input. A large unsigned number (with the msb set on) will not output the correct results in this procedure because there would be no negative weight in the value, therefore the result would be much greater than the input, therefore the `ZeroExtend` function should be used for unsigned numbers and the `SignExtend` function for signed numbers.

Justification

This is a complete solution to the problem as the identified component to the solution has been solved. The input has been extended to the given length by reusing the `ZeroExtend` function to extend the array. The sign of the input has been preserved by overwriting the zeroes from the zero extension if the input was negative and overwriting them with `0xFF`. This preserves the sign of the output because for any given two's complement signed number, the MSB is the only negative weighted bit. To make sure that the negative weight is not increased (the value does not decrease by sign extension), all the other bytes between must be set to `0xFF` as well to balance out, resulting in the same value as the input.

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Sign extension of a signed negative	The input to the function will be a signed negative and a newLength greater than the size of the input array.	input = [AA,AA] newSize = 4	Normal	Output = [AA,AA,FF,FF]	This test is necessary to ensure the function works under normal conditions. Other tests will not be valid if the function does not work for normal input. This will detect any logical errors in the code that may cause an error for any given input, such as the \$i counter exceeding the bounds of newLength.
Sign extension of a signed positive	The input to the function will be a signed positive and newLength greater than the size of the array.	input = [AA,0A] newSize = 4	Normal	Output =[AA, 0A,00,00]	This test will make sure that the sign extension is does not append 0xFF bytes every time, rather will output 0x00 bytes as the extension instead.
Test that erroneous input is handled correctly	The input to the function will be an array that is longer than newLength.	input = [AA,AA] newSize = 1;	Erroneous	Output = [AA,AA]	This test will check whether the IF statement at the start of the procedure will successfully detect when the erroneous condition where newSize is less than the length of the existing array, which should mean that the main algorithm is not executed, rather the input is immediately returned back.

AND, XOR, OR algorithms



Pseudocode

```

DEF And(byte[] input1, byte[] input2)
    // Create a new byte array to store the result
    byte[] Result = new byte[input1.Length];

    // Iterate through every index in the input
    int i = 0;
    FOR i < input1.Length
        // Here AND can be substituted for XOR/OR for the individual methods
        Result[i] = input1[i] AND input2[i];
        i++;
    ROF;

    // In AND, XOR, and OR, the carry and overflow flags are always set off in the
    // intel specification
  
```

```

FlagSet ResultFlags = new FlagSet()

// Use the in-built procedure to determine sign parity and zero flags.
ResultFlags.AutoDetermine(Result);

ResultFlags.Carry = FlagState.OFF;
ResultFlags.Overflow = FlagState.OFF;

RETURN Result, ResultFlags;

```

FED;

Complexity

This algorithm will always finish in linear time $O(n)$ theta(n) omega(n) because the entire array has to be iterated.

Explanation

AND, XOR, OR can be solved with the general solution of iterating through every index in the array and apply the logical procedure to that given index, then storing the result in the result array.

Justification

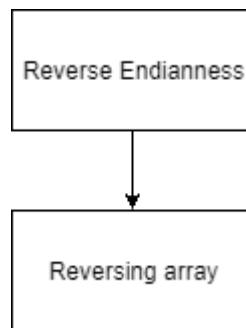
These three functions provide a complete solution to the problem because they each have a general method that solves all three sub problems. The sub-problem of setting the sign, parity, and zero flags is already solved in the FlagSet class, therefore the provided method is used. The task of performing the logical operation on each bit of the input has also been solved. This uses operators that are included in the C# language, AND OR and XOR. These have been tested in thousands of programs and have been shown to work for any valid input. The task of setting the carry and overflow flags has also been solved. This is done simply by setting the flags to FlagState.OFF in the ResultFlags variable.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test AND of two inputs	Two inputs will be entered as parameters, then the output will be compared to a known correct output sourced reliably from a calculator.	Input1 = [AA, AA, AA, AA] Input2 = [CC, CC, CC, CC]	Normal	Output = [88,88,88,88]	This test is necessary to make sure that the algorithm applies the logical operator correctly and detects any obvious logical errors such as an <code>ArrayOutOfBoundsException</code> or similar exceptions that are due to the algorithm not the input.
test XOR of two inputs	Two inputs will be entered as parameters, then the output will be compared to a known correct output sourced reliably from a calculator.	Input1 = [AA, AA, AA, AA] Input2 = [CC, CC, CC, CC]	Normal	Output = [66,66,66,66]	This test will check that the XOR operator is applied to each input and that no extra erroneous behaviour was introduced when adapting the AND algorithm to XOR instead.
test OR of two inputs	Two inputs will be entered as parameters, then the output will be compared to a known correct output sourced reliably from a calculator.	Input1 = [AA, AA, AA, AA] Input2 = [CC, CC, CC, CC]	Normal	Output = [EE,EE,EE,EE]	This test will ensure that the OR operator is applied correctly and that no extra erroneous behaviour was introduced when adapting the AND algorithm to perform an OR.
test AND on erroneous input parameters	The two inputs entered as parameters both have different sizes	Input1 = [AA] Input2 = [CC,CC]	Erroneous	ArrayOutOfBoundsException	This test is necessary to make sure that the procedure will throw an exception with erroneous input instead of output incorrect results that would be harder to track down as bugs later in the development. This will apply to XOR and OR as they use the

				same algorithm save minor adaptations.
--	--	--	--	--

Reverse endian algorithm



Pseudocode

```

DEF ReverseEndian(byte[] input)
    byte[] Result = new byte[input.Length];
    int i = 0;
    // Start at the last element of result and the beginning of input. This will reverse
    // the order of the input array and store it in the result array, hence reverse endian.
    FOR i < input.Length DO
        Result[Result.Length-1-i] = input[i];
    ROF;
    return Result;
FED;
  
```

Explanation

To reverse an endian, the order of the input is reversed. There is no way to determine whether an input is in little or big endian, therefore the general procedure “ReverseEndian” is the only applicable method. This algorithm will be useful when outputting data to the interface layer because it is convention to output immediates in disassembly and the values of registers in big endian, therefore to remove any ambiguity, the common convention will be followed.

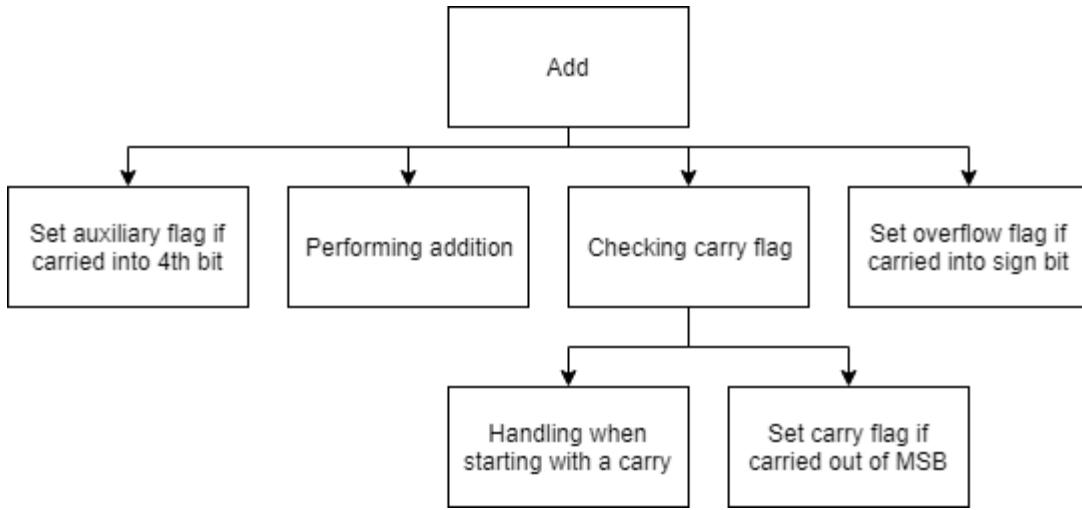
Justification

This will form a complete solution because the sub-problem of reversing the array has been solved, such that it is valid for any given input because the only difference between big endian and little endian in this context is the order of the bytes. Any big endian number can be represented in little endian by reversing the order of the bytes, therefore, there is no given input that will not work.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test that the output is the reverse of the input	An asymmetric byte array will be entered as an input and the output will be checked after using a breakpoint.	input = [AB,CD,EF]	Normal	Output = [EF,CD,AB]	This test data is necessary to make sure that the algorithm produces correct results and there are no logical errors in the implementation, such as the for loop iterating outside of the bounds of the array. It is also important that the test data is asymmetric because otherwise it would not be possible to tell whether the array had been reversed.

Add algorithm



Pseudocode

```

DEF Add(byte[] input1, byte[] input2, bool carry)

int i = 0
byte[] Result = new byte[input1.Length]
// Iterate through every byte in the input as everything needs to be added
FOR i < input1.Length DO
    // Add the two existing values together. Must be an integer because
    // carries need to be found.
    int sum = input1[i] + input2[i];

    // If the last addition carried(could have been from the previous
    // add instruction if specified in the arguments), add 1 on, as this is
    // the maximum value that can be carried.
    IF carry == true THEN
        sum += 1;
    FI;
    // If the sum is greater than the maximum value of a byte(sum is an int), then there must be a carry.
    IF sum > 0xFF THEN
        // Store the remainder of the value in the current column
        Result[i] += (byte)(sum % 0x100);
        // Carry one onto the next column
        carry = true;
    ELSE
        // Otherwise the sum can be added as normal
        // If a carry was set before, it is unset here.
        Result[i] += (byte)sum;
        carry = false;
    FI;
    i++;
ROF;
// Use provided procedure to determine SF, PF and ZF.
FlagSet ResultFlags = new FlagSet().AutoDetermine(Result);

// If there was a carry out of the MSB(the last byte that was iterated over carried), set the flag as the specification requires,
// otherwise turn it off.
IF carry == true THEN
    ResultFlags.Carry = FlagState.ON;
ELSE
    ResultFlags.Carry = FlagState.OFF;
FI;

```

```

// If the two inputs have the same sign, and the result had a different sign, the sum must have overflowed. E.g.
adding two positives gave a negative, therefore set the overflow flag.

IF (input1.IsNegative() XNOR input2.IsNegative())
    AND (Result.IsNegative() XOR input1.IsNegative())
THEN
    ResultFlags.Overflow = FlagState.ON;
ELSE
    ResultFlags.Overflow = FlagState.OFF;
FI;

// An auxiliary carry is defined as a carry into the 4th bit of the inputs.

// The lowest 3 bits can be obtained by ANDing by 7. If the sum of the lowest
of the inputs is greater than 8, it // 3 bits
// must have carried into the 4th bit in the result, therefore set the flag.

IF ((input1[0] AND 7) + (input2[0] AND 7) > 7) THEN
    ResultFlags.Auxiliary = FlagState.ON;
ELSE
    ResultFlags.Auxiliary = FlagState.OFF;
FI;
RETURN Result, ResultFlags;

```

FED;

Complexity

$O(n)$ Theta(n) omega(n). The algorithm completes in linear time for all inputs because the entire input byte arrays need to be iterated once simultaneously.

Explanation

The addition algorithm is similar to a long addition algorithm you may use to add two numbers on paper. Each column is added, then if the result of that column is greater than what can be stored(10 $>=$ in decimal, in real life. In a byte array, 256 $>=$ because a byte can go up to 255) a carry needs to be taken into account on the next column. The remainder is found using the modulo operator, which will return the number of bytes that the sum went over 0x100(the maximum value of a byte). The carry is passed as a parameter to allow the carry to be specified as already set. In context, this case covers the “ADC” Add with carry opcode, which will include the current state of the carry flag as a carry in the addition, which proves useful for adding 128-bit numbers.

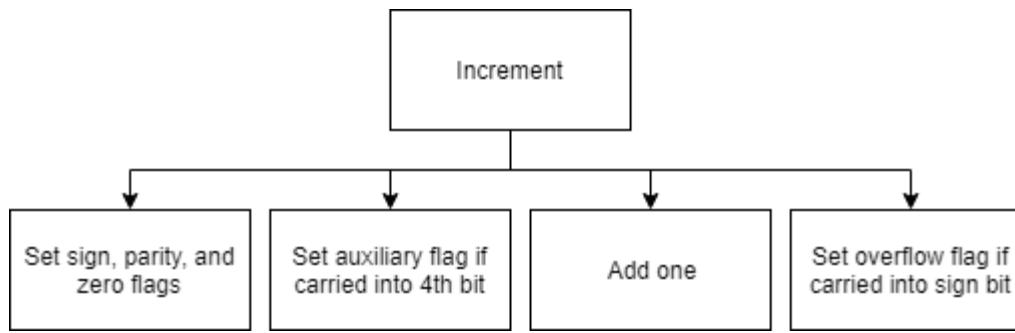
Justification

The add function is a complete solution to the problem because all problems identified through the stepwise refinement have been solved. The task of performing addition has been solved. This is because all valid inputs will produce valid results. This includes two's complement signed negatives, two's complement signed positives, and unsigned integers. This is certain to work because it uses a well-known method of addition, long addition, which is used all the time in real life, therefore a simple approach to solving the problem. The problem of checking the carry flag has also been solved. The task of handling when starting the carry has been solved by allowing the carry to be specified as a parameter. This will add the carry in the first iteration, then addition continues as normal. The task of setting the carry flag has also been solved. If the carry boolean is true after the iteration has finished, there must have been a carry out of the MSB, the carry flag in the FlagSet output is set in the IF statement which checks if the addition had an overflow. This will have occurred if and only if the input had two of the same sign, and the output had a different sign. For example, two positive numbers were added and the result was negative or vice versa, there was an overflow. The two inputs must have the same sign to overflow because the maximum size of a two's complement negative and positive would not allow this to happen. The boundary example in this case is 127 + -128. This is not even close to overflowing into the sign bit(the result should be negative, the sign bit would be overflowed if the result overflowed back to a positive), therefore this algorithm solves the problem in every case. The task of setting the auxiliary flag is also solved. This is a complete solution to this sub-problem because a suitable computable solution has been created by using computational methods such as iteration and selection to determine when the procedure has finished and to perform the addition using conditions. If the first three bits of both inputs added together are greater than seven, it is clear that the 4th bit was carried into, therefore the auxiliary flag should be set per the intel x86-64 specification.

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification

Test whether two inputs are added together correctly	Two normal inputs will be used as parameters to the function with the carry parameter false.	Input1 = [11,11] Input2 = [22,22] Carry = false	Normal	Output = [33,33] PF = true	This test is necessary to ensure that the algorithm works under normal conditions as it will be clear whether other tests show bugs or fundamental flaws in the algorithm. If the algorithm works in the general case, it could be deduced that there is a logical error in the code, which will speed up the bug identification and fixing process.
Test whether two inputs are added together correctly, starting with a carry	Two normal inputs will be used as parameters to the function with the carry parameter true.	Input1 = [11,11] Input2 = [22,22] Carry = true	Normal	Output = [34,33] All flags off	This test will make sure that the algorithm handles starting with a carry correctly when specified in the parameter, such that the carry is applied to the LSB of the result.
Check that inputs of different sizes will throw an exception	Two erroneous inputs that have different sizes to each other.	Input1 = [11] Input2 = [22,22]	Erroneous	ArrayOutOfBoundsException	This test will show whether the exception is thrown when the inputs are invalid instead of producing an invalid result that could cause bugs later in execution that would be harder to find as opposed to having the exception point it out clearly.
Test that the carry boolean will carry into the next byte properly	Two normal inputs will be used as the input that will cause the carry boolean to be used where the bit needs to be carried into the next byte.	Input1 = [FF, 00] Input2 = [01,00] Carry = false	Normal	Output = [00,01] PF = true AF = true	This test will be necessary to show that the carry boolean part of the algorithm is carrying the bits correctly. This kind of logical error would be hard to detect based on the other tests because they do not cause a carry as their purpose is different. Having two separate tests will round down the possibilities causing the problem to a greater extent.
Check whether a carry out of the MSB sets the carry flag	Two normal inputs that will cause the MSB to carry out, in turn resulting in the CF set in the result flags.	Input1 = [00, FF] Input2 = [00, 01] Carry = false	Normal	Output = [00,00] CF = true PF = true ZF = true	This test will be necessary to show that the IF condition that determines whether the CF is set in the output has any logical errors. It will also make sure that there is no strange behaviour as the MSB is carried out, e.g an exception.
Check whether a carry into the sign bit/MSB sets the overflow flag	The input will be two normal inputs that will cause a carry into the sign bit, resulting in the OF set in the result flags.	Input1 = [7F] Input2 = [01] Carry = false	Normal	Output = [80] SF = true OF = true	This test will show whether the predicate to detect an overflow is successful or not at detecting when the addition of a two of the same sign results in the other sign. This has to be separated from other tests as they may unintentionally affect the OF, resulting in an invalid test.

Increment



Pseudocode

```

DEF Increment(byte[] input)
    // Copy value not reference
    byte[] Result = (value)input;
    int i = 0;
    FOR i < Result.Length DO
        Result[i]++;
        IF Result[i] != 0x00 THEN
            // Break out of for loop
            BREAK;
        FI;
        i++;
    ROF;

    // Set SF, ZF, PF using pre-built routine
    FlagSet ResultFlags = new FlagSet(Result).AutoDetermine();
    // Carry is never set in the INC instruction, hence not in the increment function
    // Overflow can be detected by an XOR of the sign bits; input was incremented
    // and had a different sign bit. The only possible case for this is -1
    IF (Result.IsNegative() == true) AND (input.IsNegative() == false) THEN
        ResultFlags.Overflow = FlagState.ON;
    ELSE
        ResultFlags.Overflow = FlagState.OFF;
    FI;
    // The auxiliary carry is set when the 4th bit of the first byte is overflowed.
    // A very particular circumstance, but it is defined in the intel manual as
    // conventional behaviour. This is because of how binary coded decimal works,
    // however the specifics of this can be abstracted from the solution.
    IF (input[0] AND 7) == 7 THEN
        ResultFlags.Auxiliary = FlagState.ON;
    ELSE
        ResultFlags.Auxiliary = FlagState.OFF;
    FI;
    RETURN Result, ResultFlags;
FED;
  
```

Complexity

$O(n)$ $\Theta(n)$ $\Omega(1)$. The worst case is that the entire input must be iterated. The best is that only the first element of the array must be incremented(no carry).

Explanation

The increment function increments an input by one. The pseudocode assumes a carry is likely, therefore the procedure of incrementing is wrapped in a for loop. This handles the case where a carry is needed. For example, the input is a two's complement minus one. The whole input array would need to be iterated through to handle the increment, and the carry. All of the 0xFF bytes would need to be incremented to get zero. This is the worst case, the zeroth index could be incremented, then since the result of the increment was not zero(where 0xFF is incremented to get 0x00, an overflow occurs), the for loop will be broken immediately after and the function returned.

Justification

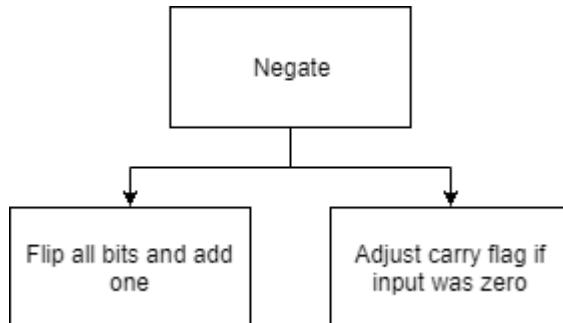
The increment function is a complete solution to the problem because all sub problems have been solved by the algorithm. The setting of the sign, parity, and zero flags has been solved by a suitable computational solution in the flag set class which will determine the flag states for the result. The auxiliary flag is set if the fourth bit has been carried into. This is because if the input has the first three bits set on, then one is added, it is certain that the fourth bit will be carried into(could then carry over again to the fifth which is still a case where the auxiliary flag would be set). The algorithm does add one. This is most suitable when separated from the add function because the procedure is a lot simpler than having nested iterations which are only necessary if and only if both inputs are greater than ten, and the increment function does not set the carry flag. This is defined in the intel manual, therefore to meet the primary success criterion of 1:1 emulation, it must be separated. The overflow flag is set when the sign bit is overflowed. This is detected by checking if the sign of the result is negative(`Result.IsNegative == true`) and the sign of the input is positive(`input.IsNegative == false`). This would occur when the input is positive, which after incremented, was negative, hence an overflow.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Test the incrementation of the input	The function will be called on a byte array and a breakpoint will be used afterwards to check if the result is one more than the input.	Input = [00,00]	Normal	Output = [01,00]	This test is necessary to check whether the algorithm is functional or has flaws. This will be important when comparing other tests as it will be clear whether bugs can be traced back to the design of the algorithm or are a special case found by that test, such that the process of identifying and fixing bugs in the algorithm is faster.
Test the incrementation of the input with a carry	The function will be called on a byte array that requires a carry and a breakpoint will be used afterwards to check if the result is one more than the input and that the carry was applied to the next byte successfully	Input = [FF, 00]	Normal	Output = [00,01]	This test will show whether the carry algorithm is working as it is separate from Test #1 because the FOR loop will be iterated multiple times in this test. This will outline any logical errors such as the condition being incorrect clearly in the test, which may be harder to trace outside of the test because I cannot spot mistakes in adding hexadecimal by eye very easily.
Test of incrementation of -10	An array of 0xFF bytes will be passed as the parameter to the function, then a breakpoint will be used to record the output afterwards.	Input = [FF, FF]	Boundary	Output = [00,00]	This test will be necessary to show how the procedure handles the special case where the carry is carried across the whole input. This is the case where the condition control of the FOR loop will no longer be met instead of the IF condition breaking the loop. This will make sure that the condition in the FOR loop is correct, as too many times could cause an <code>ArrayOutOfBoundsException</code>
Test of incrementation with signed overflow	0x7F will be used as an input to the procedure then a breakpoint will be used to record the value of the output and the result flagset.	Input = [7F]	Boundary	Output = [80] OF = true	This test will demonstrate whether the overflow flag is being set correctly, as with this input there would be a carry into the MSB, which would turn the number from a signed positive into a signed negative. This test will make sure the procedure to test this is correct. This can be certain because the only input that will cause the overflow is an input

one less than the value of the MSB. E.g MSB of a byte is negative 128d, 0x80, therefore one less is 127 or 0x7F.

Negate algorithm



Pseudocode

```

DEF Negate(byte[] input)
    byte[] Result = Xor(input, -1);
    Result = Increment(Result);
FlagSet ResultFlags = new FlagSet();
IF IsZero(Result) THEN
    ResultFlags.Carry = FlagState.ON;
ELSE
    ResultFlags.Carry = FlagState.OFF;
FI;
RETURN ResultFlags, Result
FED;
  
```

Complexity

O(n) Theta(n) Omega(n) - The algorithm will always finish in linear time because the array is being once in the Xor function, and worst-case one full iteration in the increment function.

Explanation

The negate routine will provide a two's complement negation of an input. To do this, flip all the bits(XOR by -1), then add one. The Carry flag is set if the result is zero(hence unchanged) as per the intel specification.

XORing by -1 works because -1 in two's complement binary is all 1s.

To display visually,

	Dec	Binary
$10 \oplus -1$	-1	1111 1111 1111 1111
	10	0000 0000 0000 1010
=	-11	1111 1111 1111 0101

As the figure shows, to complete the two's complement negation, the result of the XOR has to be incremented.

Justification

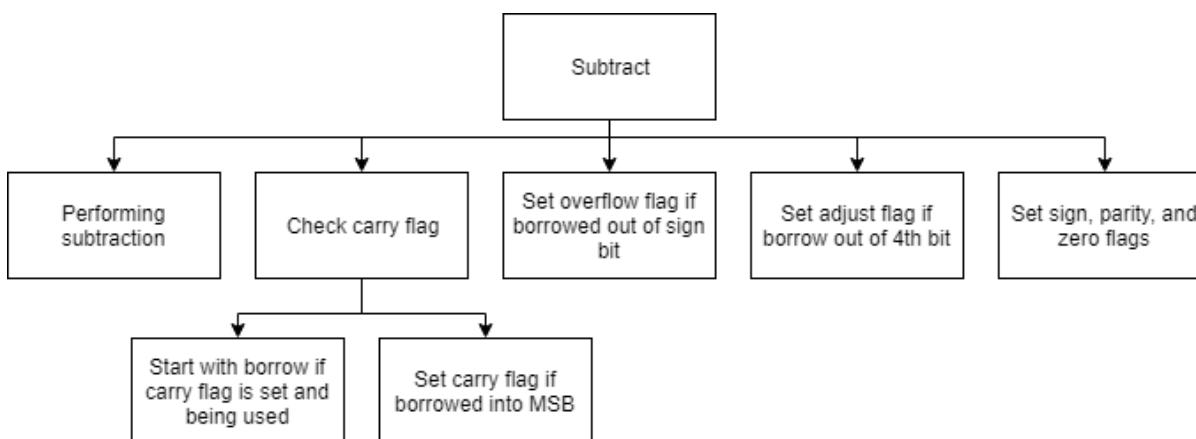
This algorithm forms a complete solution to the problem because all sub-problems in the stepwise refinement have been solved. The flipping of bits task has been solved as it follows the two's complement negation principle; Flip all bits and add one. Instead of having to recreate an algorithm for negate, the problem can be simplified by using XOR to flip all of the bits instead as XORing by 11...1111b2 will flip every bit in the input. Increment is then used to add one, such that carries are handled by the existing function. The problem of setting the carry flag has also been solved

by checking if the result is zero using the IsZero function, then setting the carry flag if the result was zero and otherwise setting it to off.

Iterative test data

Test case	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Test the procedure with a signed positive input	The function will be called with a signed positive that will be negated to become a negative of the same magnitude, which can be checked using a breakpoint.	Input = [33,33]	Normal	Output = [CC,CC]	This test will be necessary to check whether the core functionality of the input works and detect any flaws in the algorithm as the procedure cannot be expected to work with special cases if it does not work for the general case. This will reduce development time because it will be clear which parts of the function work and which parts need to be changed.
Test the procedure with a signed negative input	The function will be called with a signed negative that will be negated to become a positive of the same magnitude, which can be checked using a breakpoint.	Input = [AA,AA]	Normal	Output = [55,55]	This test will be necessary to check that the procedure works with negative inputs as well as positive inputs as identified in Test #1. This would outline any fundamental flaws in the algorithm as it mostly depends on other procedures. If something were to go wrong, the algorithm itself would likely need correcting.
Test the procedure with an array of zeros	The function will be called with an array of zeros that will be the same as the output because zero negated is still zero.	Input = [00,00]	Normal	Output = [00,00] CF = true	This test will determine whether the procedure will work for the special case that the input is zero. This includes identifying any logical errors that would cause the carry flag not to be set true or if the algorithm did not work properly with zero as an input.

Subtract algorithm



Pseudocode

```

DEF Subtract(byte[] subtractee , byte[] input2, bool borrow)
    // result = subtractee - subtractor
    int i = 0;
    byte[] Result = new byte[subtractee .Length]
    FOR i < subtractee.Length DO
  
```

```

int sum = subtractee[i] - subtractor[i];
IF borrow THEN
    sum -= 1;
FI;
IF sum < 0 THEN
    // Borrow 256 from the next byte. 1 will be deducted from the
    // next byte next iteration.
    Result[i] = (byte)(sum + 256);
    borrow = true;
ELSE
    Result[i] = (byte)sum
    borrow = false;
i++;
ROF;
// Use provided procedure to determine SF, PF and ZF.
FlagSet ResultFlags = new FlagSet().AutoDetermine(Result);

IF borrow == true THEN
    // The carry also represents a borrow in this case. The carry flag
    // is not strictly only for carries, just an outdated name kept for
    // consistency, yet necessary to keep in the program because still used
    // today.
    ResultFlags.Carry = FlagState.ON;
ELSE
    ResultFlags.Carry = FlagState.OFF;
FI;
// Condition true if both inputs are different signs and the sign of the result is not
// the same as the sign of the input. If a positive was subtracted from a negative,
// and the result was a positive, and integer overflow occurred, therefore the overflow
// flag needs to be set. Principle also applied to +ve - -ve.
IF (subtractee.IsNegative() XOR subtractor.IsNegative())
    AND (subtractee .IsNegative XOR Result.IsNegative())
    THEN
        ResultFlags.Overflow = FlagState.ON;
ELSE
    ResultFlags.Overflow = FlagState.OFF;
FI;
// Determine auxiliary flag by checking if the 4th bit was changed in the result, as this
// is the condition to set the AF defined in the intel specification.
// To do this, AND each of the least significant bytes by 8, such that the only bit left is
// the 4th bit(8), then check if both of the values are the same. They are ORed
// because if either of the bits were set, then not the same in the result, that would
// be the definition of the 4th bit changing in this context. This is to do with binary
// coded decimal, however the details of this do not matter as the only thing to
// consider is whether the specification states that I should, and this is the case.
IF ((input1[0] AND 8) OR (input2[0] AND 8)) XNOR (Result[0] AND 8) THEN
    ResultFlags.Auxiliary = FlagState.ON;
ELSE
    ResultFlags.Auxiliary = FlagState.OFF;
FI;
RETURN Result, ResultFlags;
FED;
Complexity
O(n) Theta(n) omega(n). The algorithm completes in linear time for all inputs because the entire input byte arrays
need to be iterated once simultaneously.
Explanation

```

To implement the subtraction function, a long subtraction method will be used, as you would subtract numbers in real life. The inputs are iterated through and each column in the subtractor subtracted from the same column in the subtractee. When the result of the column subtraction is less than 0, a borrow of one must be made from the next most significant column(which is worth 256 in the current column because the byte arrays represent base 256). On the next iteration, one will be subtracted because a value of one was borrowed in the previous column. Then, the borrow boolean will be false unless another borrow is needed from the next column, hence the process repeats. In the case that there is a borrow into the most significant bit, the carry flag will be set. If there is a borrow out of the MSB(into the second most significant bit), the overflow flag will be set hence the sign has changed. The borrow is passed as a parameter to allow the procedure to begin with a borrow. In context, this will be used with the SBB subtract with borrow opcode where the state of the carry flag will determine whether the subtraction starts with a borrow or not.

Justification

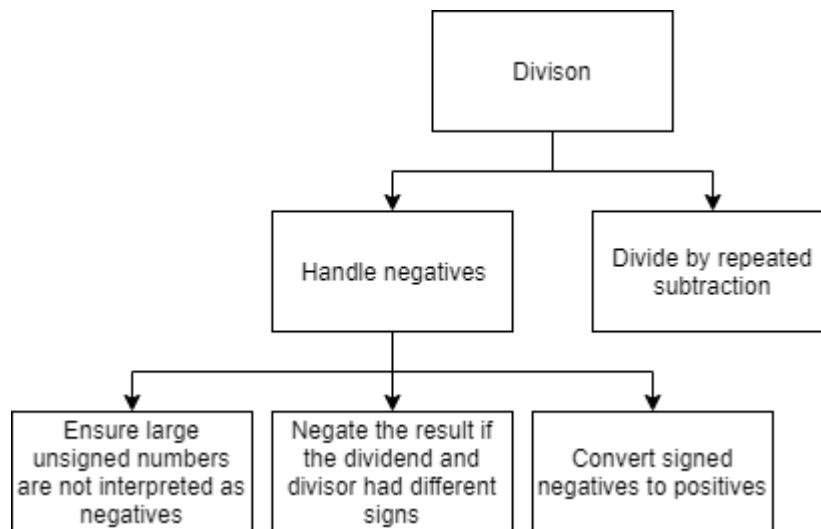
Like with addition, using the simplest method will have benefits including easier maintainability, as less skill and time is required to refactor the code, and easier readability, such that it will benefit the stakeholders reading it more so than if the code was too complex to understand. The inputs will also have to be the same size for the same reason, there is no general approach to dealing with numbers that could be signed or unsigned. A possibility would be to have two functions for each sign, however, the problem arises when implementing opcodes as the "ADD" opcode in x86-64 is not specific to any sign. This is because two's complement allows for a suitable computable solution when subtracting/adding two numbers that are either signed or unsigned, provided they are of the same length or otherwise the signedness is known, which can be performed on any sized inputs as the process is repeated through iteration produce the result. This algorithm is however a complete solution for all valid inputs, as explained before, two's complement allows for addition and subtraction to be generalised into one solution, therefore there is no other input that would be invalid. Moreover, there is no instance where the solution would not apply to a problem that requires it; result flags are required if necessary, but can be ignored if not.

Iterative test data

Test Case ID	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Test 1: Check if two inputs are subtracted correctly.	Two normal inputs will be used as parameters to the function with the borrow parameter false.	Input1 = [22,22] Input2 = [11,11] Borrow = false	Normal	Output = [11,11]	This test is necessary to ensure that the algorithm works under normal conditions as it will be clear whether other tests show bugs or fundamental flaws in the algorithm. If the algorithm works in the general case, it could be deduced that there is a logical error in the code, which will speed up the bug identification and fixing process.
Test 2: Check if two inputs are subtracted correctly, starting with a borrow.	Two normal inputs will be used as parameters to the function with the borrow parameter true.	Input1 = [11,11] Input2 = [22,22] Borrow = true	Normal	Output = [10,11]	This test will make sure that the algorithm handles starting with a carry correctly when specified in the parameter, such that the borrow causes the LSB to decrease by one(which may cause further borrows)..
Test 3: Check that inputs of different sizes will throw an exception.	Two erroneous inputs will be used; they will have different sizes to each other.	Input1 = [11] Input2 = [22,22]	Erroneous	ArrayOutOfBoundsException	This test will show whether the exception is thrown when the inputs are invalid instead of producing an invalid result that could cause bugs later in execution that would be harder to find as opposed to having the exception point it out clearly.
Test 4: Check that the borrow boolean will borrow from next byte after where the borrow was needed.	Two normal inputs that will cause the borrow boolean to be used to decrement the next byte by one which can be identified by using a breakpoint after the function call and checking the output.	Input1 = [00, 01] Input2 = [01,00] Borrow = false	Normal	Output = [FF,00]	This test will be necessary to show that the carry boolean part of the algorithm is correctly performing the borrow operation on the next byte in the array. Separating this test from the others will show clearly whether an error lies in the function of the borrow in the algorithm or a more general part of the algorithm. For example if this test is failed, but Test #1 is passed, it would be clear that there is a bug in this part of the algorithm.
Test 5: Check whether a borrow out of the MSB sets the carry flag in the output.	Two normal inputs that will cause there to be a borrow into the MSB, resulting in the CF set in the result flags.	Input1 = [00, 01] Input2 = [00, 02] Borrow = false	Normal	Output = [FF,FF] CF = true	This test will be necessary to show that the borrow boolean is not handled correctly when the subtraction has finished such as the IF condition that determines whether the CF is set in the output may not be coded correctly, e.g the flag is set under the wrong condition. It will also make sure that there is no strange behaviour when there is a borrow into the MSB, e.g an exception, which would be not identified in other tests where there is no borrow into the MSB.

check whether a borrow out of the sign bit/MSB sets the overflow flag	The input will be two normal inputs that will cause a borrow out of the sign bit, resulting in the OF set in the result flags.	Input1 = [80] Input2 = [01] Borrow = false	Normal	Output = [7F] OF = true	This test will show whether the predicate to detect an overflow is functional. This would be the case when a positive is subtracted from a negative and the result is a positive or vice versa. This would not be found in other tests as they do not test this part of functionality, so it is necessary to have a test that covers this aspect of the algorithm or bugs will be harder to track down in the future if present when this part of the solution is developed.
---	--	--	--------	----------------------------	--

Divide algorithm



Pseudocode

```

DEF Divide(byte[] dividend, byte[] divisor, bool signed);
    // The output is two byte arrays the same length as the input.
    Quotient = new byte[dividend.Length];
    Modulo = new byte[dividend.Length];

    IF divisor.IsTrueZero() THEN
        // Divide by zero error
    FI;

    // A repeated subtraction method will be used to divide, therefore once the sign
    // changes, the division will have finished. LastSign will be used to determine this.
    // LastSign is true if the dividend is a two's complement negative
    // NegativeDivided is true if and only if the dividend is a two's complement negative
    bool LastSign = dividend.IsTrueNegative();
    bool NegativeDividend = signed AND LastSign;

    // A general computable solution cannot be applied to division due to how it will be determined that
    // the division is finished(sign change works differently for unsigned compared to signed). To allow a computable solution, instead
    // the negative input will
    // be negated, then the result will be negated later if and only if the divisor was not
    // negative. This is because a +ve divided by a +ve is always positive and hence is a
    // -ve divided by a -ve also always positive, therefore negating later would not be necessary.
    IF NegativeDividend == true THEN
        dividend = Negate(dividend);
        LastSign = false;
    
```

```

FI;

// Same applied to divisor
bool NegativeDivisor = divisor.IsNegative;
IF NegativeDivisor == true THEN
    divisor = Negate(Divisor);
FI;

WHILE TRUE DO
    // Subtract divisor from dividend and store in temporary buffer
    byte[] DividendBuffer = Subtract(dividend, divisor);

    // Store new sign
    bool NewSign = DividendBuffer.IsNegative();

    // If the signs are the same, division is not yet complete
    IF (LastSign XNOR NewSign) OR LastSign == true THEN
        // When LastSign is already true but != New sign. Read explanation
        LastSign = NewSign;

        // Update dividend
        dividend = DividendBuffer;

        // For each time the division is performed, the quotient increases
        Quotient = Increment(Quotient);
    ELSE
        // The division is complete due to a change in sign
        BREAK;
    FI;

ELIHW;

IF (signed == true) AND (NegativeDividend XOR NegativeDivisor) THEN
    // Flip the quotient and the divisor because the input had mixed signs,
    // therefore the result is negative.
    Quotient = Negate(Quotient);
FI;

// The remainder of the dividend that would go below zero if the divisor was subtracted
// once more.
Modulo = Dividend;

// If one of the inputs was negative, the remainder will be negative because it rounds towards
// zero.
IF (signed == true) AND (NegativeDividend OR NegativeDivisor) THEN
    Modulo = Negate(Modulo);
FI;

// Divisor affects no flags.
RETURN Quotient, Modulo;
FED;
Complexity
The repeated subtraction algorithm is in linear time, O(n) theta(n) omega(1) as the iterations are a constant(this is the
result quotient) multiplied by the elements in the array(iterated in the subtraction method). However, this is
misrepresentative of the realistic time, as the constant will likely be a very high number. Because of this, O(n) is not
necessarily the best algorithm possible.
Explanation

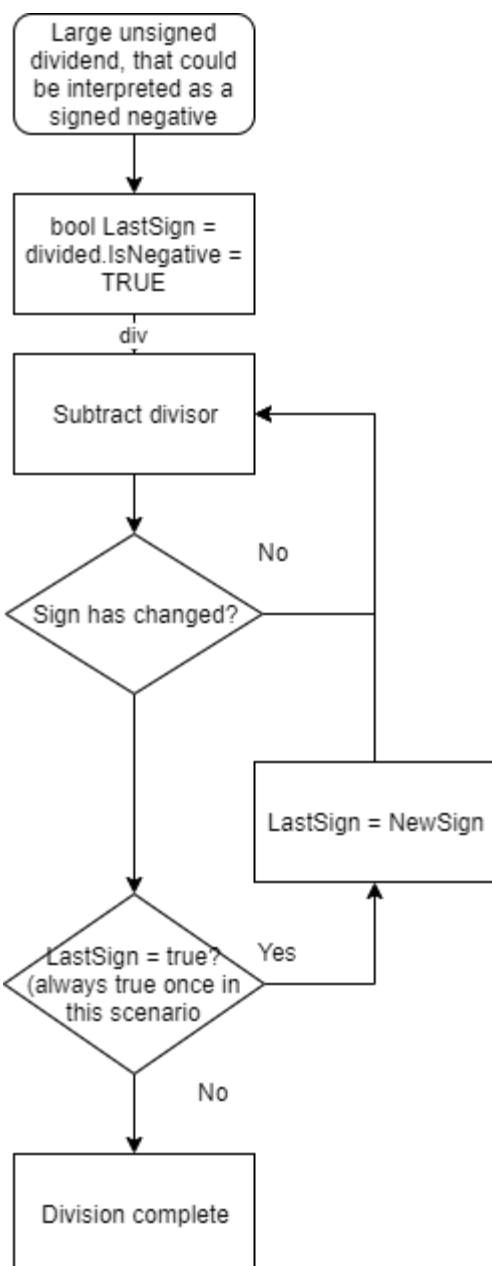
```

The division algorithm works by repeated subtraction, as is the fundamental concept of division. The divisor is subtracted from the dividend until the dividend is less than the divisor, then the remainder is the modulo. In division, the sign is important. This is because when the divisor is subtracted, it would be impossible to determine whether the dividend turned negative, or was just a really big unsigned number to begin with(more specifically with the sign bit already on). For this reason, the signedness must be passed as a parameter to the function. Even that the negatives have been detected, there still can be no solution because subtracting a positive repeatedly from a negative would loop until an overflow in the wrong direction. Therefore, all signed negatives be flipped before performing the division, then the sign can be determined afterwards. The sign can be determined by referring back to the signs of the inputs. If both are the same sign, the modulus and quotient will always be positive. If both have different signs, the modulus and quotient will be negative.

Justification

The division algorithm is a complete solution to the problem because all sub-problems have been solved. Division by repeated subtraction has been solved by using a while loop to iterate while the dividend is greater than the divisor(dividend - divisor > 0), then subtracting the divisor from the dividend. The sub-problem of handling negatives has also been solved. This has been done by abstracting the concept of negatives from the repeated division algorithm. Any signed negatives in the input have been negated, hence converted to positives at the start of the function. The result has been negated at the end if the dividend and divisor had different signs. This is detected using an XOR, which is true if and only if one of the inputs had a negative sign. The sub problem of ensuring large unsigned numbers are not interpreted as negatives has been solved by the use of the LastSign variable. The LastSign variable is used to identify a change in sign. If the input dividend is a signed negative(will be negated) or a small unsigned number, a change in sign will indicate that the dividend buffer has gone below zero, therefore the division is complete, because the divisor cannot be subtracted any more. However, if the dividend is a large unsigned number, such that it could be interpreted as a signed negative, this will not work because there will be two changes in sign. Once from signed negative(but actually a large unsigned number) to a signed positive, then finally from a signed positive to a signed negative; the point where the division really has finished. To further generalise the method, initialising LastSign to dividend.IsNegative() is important. If the dividend is a signed negative, the dividend is negated and last sign is set to false. Therefore, the only case that LastSign is true on the first iteration(true implies negative) is if the input is a large unsigned number that could be interpreted as a signed negative. To handle the first transition from "could be signed negative" to "could be signed positive", the "OR LastSign == true" condition is needed. This will ignore LastSign when it is true initially, but remember that LastSign XNOR NewSign is only true if both are the same, not specifically testing for NewSign to be true(hence negative). Therefore, the last sign is true, there will always be another subtraction, however LastSign will still be updated to NewSign, hence will go back to false at some point. At this point, the division continues as if it was a small unsigned number.

The following flowchart shows a simplified execution flow of this routine when the input is a large unsigned number that could be interpreted as a negative signed number.

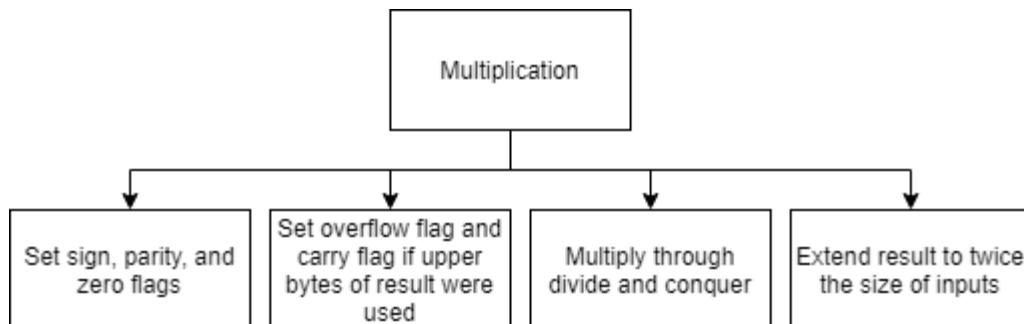


Iterative test data

Test case	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Test #1: Divide two unsigned numbers with no remainder	Two unsigned numbers which are evenly divisible will be passed as parameters to the function that represent the dividend and divisor. A breakpoint will be set after the function call to check the result.	Dividend = [00,01] Divisor = [10,00] Signed = false	Normal	Quotient = [10,00] Modulo = [00,00]	This test is necessary to check whether the algorithm is functional with the most basic of inputs. This will test that the algorithm will work under the conditions that introduce the least possibility of an error. For example the algorithm becomes more complex when signed negatives are used as inputs as more blocks of code are executed, e.g. negating the inputs.
Test #2: Divide two unsigned numbers with a remainder	Two unsigned numbers which are not evenly divisible will be passed as parameters to the function that represent the dividend and divisor.	Dividend = [00,01] Divisor = [11,00] Signed = false	Normal	Quotient = [0F,00] Modulo = [01,00]	This test will make sure that the modulo part of the algorithm is functional as the quotient was already tested in Test #1. This test will outline any logical errors in the code that are only present when there is a non-zero remainder, which would be hard to track down if this test was not

					performed
Test the division of two signed negatives with no remainder	Two unsigned numbers which are evenly divisible will be passed as parameters to the function that represent the dividend and divisor.	Dividend = [F6, FF] Divisor = [FB,FF]	Normal	Quotient = [05, 00] Modulo = [00,00]	This test is necessary to demonstrate whether the negation process will be handled correctly when dealing with negatives. There is a higher margin for error in this test as more code is being executed(the process of negation and converting back where applicable). This test will show if the conditions to detect negative inputs are correct, and whether the part of the algorithm that handles them(negation) is coded correctly.
Test the division of two signed negatives with a remainder	Two unsigned numbers which are not evenly divisible will be passed as parameters to the function that represent the dividend and divisor.	Dividend = [F5,FF] Divisor = [FB, FF] Signed = true	Normal	Quotient = [02 , 00] Modulo = [FF,FF]	This test will check that the algorithm is working with negatives as well as remainders, which is not tested in any other test so needs to be tested separately. This will also show whether there are any logical errors in the process of negating the modulus(e.g. an incorrect condition).

Multiplication algorithm



Pseudocode

```

DEF Multiply(byte[] input1, byte[] input2, bool signed, int size)
    // Inputs need to be sign extended because the result is twice the size
    // of the input(specific to x86-64). Because of this, everything needs to be
    // doubled. This is because carries can be many bits, not just a single bit like in
    // addition. This means that the concept of adding on the carry next iteration like with
    // addition cannot be applied here.
    IF signed == true THEN
        SignExtend(input1,size*2);
        SignExtend(input2, size*2);
    ELSE
        ZeroExtend(input1,size*2);
        ZeroExtend(input2,size*2);
    FI;
    // Twice the new size of inputs.
    Result = new byte[size*4]
    int ColumnPos = 0;
    // Iterate through each byte of every operand
  
```

```

FOR ColumnPos < size*2 DO
    int BytePos = 0;
    FOR BytePos < size * 2 DO
        // As the carry can be many bits, the result may already have bytes set from previous iterations, therefore
        // the existing value must be added on, but not multiplied.
        int mul = (input1[ColumnPos] * input2[BytePos]) + Result[BytePos + ColumnPos];
        Result[BytePos + ColumnPos] = (byte)(mul % 100);
        IF mul > 0xFF THEN
            // Add requires operands of the same size
            byte[] carry = new byte[size*4];

            // Handle the powers of 256 in each column like how you
            // add a zero when moving onto the next column in long
            // multiplication, except no longer just worth one in the next
            // column.
            carry[ColumnPos + BytePos + 1] = (byte)(mul / 0x100)

            // Add carry onto the result
            Result = Add(Result, carry);
        FI;
        BytePos++;
    ROF;
    ColumnPos++;
ROF;

// Resize the result back down to the size of the output
Result = Cut(Result, size*2);

// Set SF, ZF, PF using pre-built routine
FlagSet ResultFlags = new FlagSet().AutoDetermine(Result);

// The carry and overflow flags are set on if the upper bytes of the output were used.
// To determine this, I will check if the sign extension of half the result(lower bytes)
// equals the entire result. If they are not equal, there must be something other than
// sign extension in the upper bytes of the result, therefore used. This has to be
// sign extension because the intel manual defines that neither 0xFF bytes in a negative
// number are not significant, nor zeroes in a signed positive number.
IF signed == true THEN
    IF SignExtend(Cut(Result, size), size*2) == Result THEN
        ResultFlags.Overflow == FlagState.OFF;
        ResultFlags.Carry == FlagState.OFF;
    ELSE
        ResultFlags.Overflow == FlagState.ON;
        ResultFlags.Carry == FlagState.ON;
    FI;
ELSE
    IF ZeroExtend(Cut(Result, size), size*2) == Result THEN
        ResultFlags.Overflow == FlagState.OFF;
        ResultFlags.Carry == FlagState.OFF;
    ELSE
        ResultFlags.Overflow == FlagState.ON;
        ResultFlags.Carry == FlagState.ON;
    FI;
FI;
RETURN Result, ResultFlags;
FED;
Complexity

```

$O(n^2)$ Theta(n^2) Omega(n^2). The algorithm will have to iterate through every combination of multiplications between the top and bottom row.

Explanation

The algorithm works like long multiplication that you would write on paper, except with a few shortcuts that allow a faster computation. Firstly, the algorithm works by the principle that a number can be split by digits so long as their magnitude is conserved. For example, $432 = 400 + 30 + 2$. This also applies to base 256(arrays of bytes), where the digits are split up into $256^0, 256^1, 256^2$ instead of $10^0, 10^1, 10^2$. As the inputs are byte arrays, the digits are already split up. $\text{input1}[0]$ holds the digits that have a weight of 256^0 , $\text{input1}[1]$ holds the digits that have a weight of 256^1 etc. This means that the only step left is to perform the multiplications. This is done by iterating through each index of input1 and iterating again to multiply it by every index of input2 . This could be the other way round, either way would not matter.

For example,

$$\begin{array}{r} 123 \times \\ 45 \\ \hline 615 \\ 4920 \\ \hline 5535 \end{array}$$

The digit in the units column, 5, must be multiplied by 1, 2, and 3, then written beneath, then the process is repeated with the tens column, 4, which represents 40 not 4; ($4 * 10^1$). In the pseudocode, the process of writing the multiplication results beneath can be skipped and can just be added straight onto the result array, such that they do not have to be added later.

Unlike with add, the carries of this can get very large fast because they are no longer only worth one in the next column. Hence, the best approach is to use the pre-existing add function to add the carry onto the result. This is why the existing value of the results column must be considered in the next iteration. The iteration will loop back round to the beginning of the results array when the iteration of input2 has finished(the value at ColumnPos index in input1 has been multiplied all elements of input2), therefore the existing value must be added on afterwards to make sure that the result of the previous iteration is not overwritten. This also has to be included in the assignment of "mul" because the existing value could carry into the next row, which is handled by the carry condition($\text{mul} > 0xFF$).

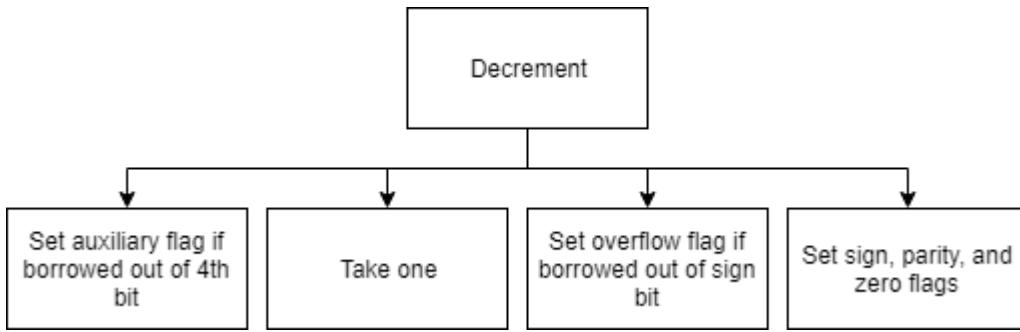
Justification

This is a complete solution because all sub-problems have been solved. The task of extending the result to twice the size of the inputs has been solved by using sign extension when the parameters to the function specify signed multiplication, and zero extended when specified as unsigned. This is a complete solution to the task because it will preserve the value of any given input. The task of setting sign, parity, and zero flags has been solved by using the method provided by the FlagSet data type, as this was a common problem across the solution. The problem of setting the overflow and carry flag has been solved by checking if the upper bytes contain significant bytes. Significant bytes are bytes that affect the value of the result. Zeros are significant if and only if the result is a signed negative, as the sign extension for negatives is purely 0xFFs, otherwise for any other number, the sign extension is purely 0x00s(hence any non-zero is significant). Therefore, this has been solved by checking if the upper bytes were significant or not. This task is necessary as it is defined in the x86-64 specification for MUL and IMUL opcodes. The problem of multiplication has been solved by a divide and conquer approach. This has been done by splitting the problem into easier solve components, which are the powers of 256 implied by the column index for each element in the input. This is the same as long multiplication, where the units column is multiplied first, then the tens column is multiplied with an extra zero at the start where the numbers to add are written. This allowed for a much simpler solution than multiplying all at once.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test the multiplication of two unsigned numbers	Two unsigned numbers will be passed as the parameters to the function and the result will be compared afterwards by using a breakpoint.	Input1 = [50,00] Input2 = [60,00] Signed = false Size = 2	Normal	Output = [B8,0B,00,00]	This test will be necessary to show whether the algorithm works under normal conditions; there is little room for error because there are no special cases required for the inputs, such that a failure in this test will demonstrate a flaw in the algorithm that may indirectly cause all other tests to fail.
test the multiplication of two signed positive numbers	Two signed positive numbers will be passed as the parameters to the function.	Input1 = [AA,00] Input2 = [BB,00] Signed = true Size = 2	Normal	Output = [2E, 7C, 00, 00]	This test will show whether the procedures applied to signed numbers affect the result in a way that they should not. This test will outline the step between signed negative and unsigned multiplication, as it will show whether a bug affects all signed numbers, or only signed negative numbers, which will allow the location of the bug to be deduced faster, saving development time.
test the multiplication of two signed negative numbers	Two signed negative numbers will be passed as the parameters to the function.	Input1 = [CE,FF] Input2 = [C4,FF] Signed = true Size = 2	Normal	Output = [B8,0B,00,00]	This test will show whether the multiplication of negative numbers is correct, as there are different aspects in the algorithm that will be tested by multiplying negatives opposed to multiplying positives. For example, there will be a carry out of the MSB, which will cause the positive into a negative, a circumstance that has been isolated from other test but still needs to be tested.
test the multiplication of two inputs of different sizes	Two input arrays will hold an unsigned value and of different lengths, will be passed as the parameters to the function. This test will be a success if the result is correct and there are no exceptions such as an ArrayOutOfBoundsException as might be expected from this input.	Input1 = [10,00] Input2 = [20] Signed = false Size = 2	Normal	Output = [00,02,00,00]	This test will be necessary to check whether the condition at the beginning of the algorithm successfully causes inputs to be sign extended to the length of the Size variable. This will be important because there are many cases where this condition will be used, for example, larger multiplication that requires the entire array to store results may experience an ArrayOutOfBoundsException if this test is not performed.
test the conditions of the CF and OF being set	Two signed positive numbers that will cause the upper bytes of the output to be used will be passed as the parameters to the function.	Input1 = [11,11] Input2 = [22,22] Signed = false Size = 2	Normal	Output = [42, 86, 46, 02] CF = true OF = true	This test will be necessary to test whether the conditions to detect the upper bytes being used are programmed/implemented correctly and outline any logical flaws in this procedure. This is also not tested in any other test, therefore necessary to have a separate test.

Decrement algorithm



Pseudocode

```

DEF Decrement(byte[] input)
    // Copy value not reference
    byte[] Result = (value)input;
    int i = 0;
    FOR i < Result.Length DO
        Result[i]--;
        IF Result[i] != 0xFF THEN
            // Break out of for loop
            BREAK;
        FI;
        i++;
    ROF;

    // Set SF, ZF, PF using the provided routine
    FlagSet ResultFlags = new FlagSet(Result).AutoDetermine();

    // Like in increment, the carry flag is never set.
    // The overflow will be set if the result is positive but the input was negative
    // which indicates an invalid result.
    IF (Result.IsNegative() == false) AND (input.IsNegative() == true) THEN
        ResultFlags.Overflow = FlagState.ON;
    ELSE
        ResultFlags.Overflow = FlagState.OFF;
    FI;
    // Determine auxiliary flag by checking if the 4th bit was changed in the result, as this
    // is the condition to set the AF defined in the intel specification.
    // Similar procedure to as done in subtract, however the subtractor is known to be
    // one therefore the procedure can be simplified. If the input and the result have a
    // different 4th bit (where 1 is bit one), it must have changed, therefore set the AF.
    IF (((input[0] AND 8) XOR (Result[0] AND 8)) == 8) THEN
        ResultFlags.Auxiliary = FlagState.ON;
    ELSE
        ResultFlags.Auxiliary = FlagState.OFF;
    FI;
    RETURN Result, ResultFlags;
FED;

```

Explanation

This algorithm has the same idea as the increment algorithm. The for loop allows the possibility of any number of borrows to be required. If the index was decremented and the index did not turn to 0xFF ($0x00 - 1 == 0xFF$), no borrow is required, which means one has been decremented from the input hence break out the loop and exit. Otherwise, the for loop will continue this iteration until there are no more borrows are required, or there is a borrow out of the MSB(if the input was full of zeros and was decremented, the result would be -1 which has all bits set, [0xFF] [0xFF] [0xFF] ...)

Justification

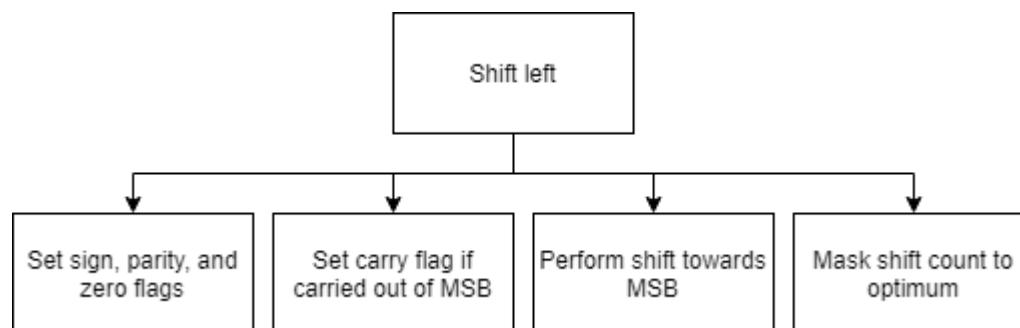
The decrement algorithm is a complete solution to the problem because all of the identified sub-problems have been solved. The task of subtracting one has been solved by the for loop, which will iteratively borrow until one has been decremented at the index which does not need a borrow. This means that when the input is zero, all the bytes will be

borrowed from to get minus one, which is the correct value. This will work like how the decrement function is implemented in x86-64, therefore completely solves the problem in that respect.

The task of setting the SF PF and ZF has been solved using the method provided by the FlagSet data structure which will apply a suitable computable solution to the problem. The task of setting the overflow flag has been solved by checking if the input was negative and the result was positive. This would indicate an overflow because if a negative number is decremented, the expected result would also be negative, therefore the result must have overflowed. As the value is always decremented by one, this would apply specifically to 0x80 (extended to length of input), where there would be a borrow out of the sign bit. The task of setting the auxiliary flag has been solved by checking if the 4th bit (starting from bit 1, with a value of 8) is the same in the input as it is in the result. If they are different, then the XOR of the two arrays by mask 8 will result in 8, hence the auxiliary flag will be set.

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Decrement a number that requires no borrows	A number will be passed as a parameter to the function and a breakpoint will be set afterwards to check the result	Input = [11]	Normal	Output = [10]	This test will be necessary to show whether the function works on a general level, as other tests may not be applicable if the algorithm itself is not implemented properly
Decrement a number that requires a borrow	A number which has its LSB equal to 00 will be used as a parameter to the function	Input = [00,01]	Normal	Output = [FF,00]	This test will show whether the FOR loop in the algorithm is implemented correctly, as a logical error would cause the borrow to not be calculated correctly, revealing a bug that was not shown in Test#1 because in this case the function will not return on the first iteration.
Decrement a number that causes an overflow	0x80 will be used as the input parameter to the function and the result will be checked after.	Input = [80]	Boundary	Output = [7F] OF = true	This test will be significant to show whether the special case of the integer overflow present in the algorithm will be handled correctly. This includes setting the overflow flag as the input was a negative, then once decremented returned a positive, which indicates an overflow. This is not present in other tests because it is an isolated condition that needs to be tested separately so I can be certain about its behaviour.

ShiftLeft algorithm



Pseudocode

DEF ShiftLeft(byte[] input, byte count)

// If shifting more than 63(0x3F) times on a byte array size 8(64 bits) or greater, shifting 63 or
// more times will do nothing because the result will be all zeros. The same applies to

```

// size 4(32 bits) with 31(0x1F) shifts. As per the intel manual, this is only defined for
// these two conditions; word and byte sizes do not have their own, therefore in order
// to maintain accuracy, this behaviour is included.

IF input.Length >= 8 THEN
    count = count AND 0x3F;
ELSE
    count = count AND 0x1F;
FI;

// This will be used later
int StartCount = count;

// If not shifting at all, exit early without affecting the flags. This is defined in the intel
// x86-64 specification, " If the count is 0, the flags are not affected"
IF count == 0 THEN
    return input, new FlagSet();
FI;

// Copy the input by value, then shift this instead of affecting the input array.
byte[] Result = (value)input;

// Shift equivalent of a carry, except named different to not confuse the two.
bool Pull = false;

// Repeat each shift separately. E.g a shift of 20 will shift by 1, twenty times.
FOR count > 0 DO
    // Pulls do not last across shifts, bits left shifted out of the MSB are not preserved
    Pull = false;
    // Iterate through the result array
    int i = 0;
    FOR i < Result.Length DO
        // If the MSB of the previous byte has been shifted out, it is shifted
        // into the first bit of the next byte in the array. This is done by ORing
        // the first bit with a mask when required.
        int PullMask;
        IF Pull == true THEN
            PullMask = 1;
        ELSE
            PullMask = 0;
        FI;

        // If the MSB of the result is set, then it is about to be shifted out of this
        // byte. This means that it has to be ORed onto the first bit of the next
        // byte as the array will not know itself that it represents a single value
        IF MSB(Result[i]) == 1 THEN
            Pull = true;
        ELSE
            Pull = false;
        FI;
        // Shift the byte by one and OR the MSB of the previous byte onto the first bit
        // where required.
        Result[i] = (Result[i] << 1) OR PullMask;
    ROF;

    // Decrement counter as a shift is complete.
    count -= 1;
ROF;

```

```

// Auto determine SF,ZF,PF with provided method
FlagSet ResultFlags = new FlagSet().AutoDetermine(Result);

// If the last iteration pushed a 1 out of the MSB of the result, the carry flag is
// set on.
IF Pull == true THEN
    ResultFlags.Carry = FlagState.ON;
ELSE
    ResultFlags.Carry = FlagState.OFF;
FI;

// A very specific condition that is defined in the intel specification, but has no
// apparent purpose, but included as part of the success criteria for 1:1 emulation.
// If the shift was only by 1 bit, the overflow flag is set on if and only if one of the
// carry flag or the MSB of the result are true(At this point, the carry flag and
// pull variable have the same value).
IF StartCount == 1 THEN
    IF (MSB(Result) == 1) XOR (Pull == true) THEN
        ResultFlags.Overflow = FlagState.ON;
    ELSE
        ResultFlags.Overflow = FlagState.OFF;
    FI;
FI;

// Return result and flags
RETURN Result, ResultFlags;

```

FED;

Explanation

This algorithm works by iteratively shifting for each shift required and decrementing the counter to count the number of shifts remaining. To handle MSBs of individual bytes being shifted out, the pull boolean is used to indicate that the previous byte had a byte pushed out of the MSB, therefore should be shifted into the LSB of the current byte. It is important that the push mask is ORed after the shift because otherwise it would be included in the shift and not be in the LSB, rather the bit above the LSB.

Justification

The shift left algorithm is a complete solution to the problem because all sub-problems identified through the stepwise refinement have been solved. The task of setting the SF PF and ZF has been solved by using the extension method provided by the FlagSet data structure, which will automatically define the flags based on a set of rules. The task of masking the shift count to the optimum value has been solved to the extent that the intel specification permits. This is defined in their pseudocode in the official intel x86-64 manual,

IF 64-Bit Mode and using REX.W

THEN

countMASK ← 3FH;

ELSE

countMASK ← 1FH;

FI

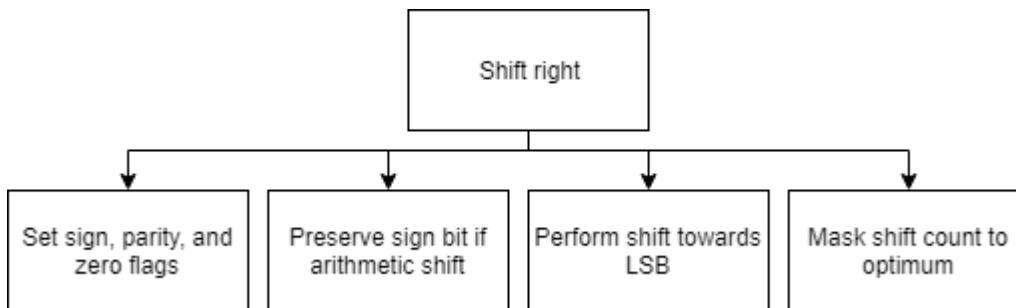
tempCOUNT ← (COUNT AND countMASK);

This is identical to the algorithm except the concept of 64-bit mode and Rex bytes have been abstracted from the function, however the same conditions can be detected using the length of the array. A Rex.W prefix indicates that the operand is 8 bytes (and so also requires 64-bit mode), which is the same as checking if the input array is 8 bytes in length. Therefore, the task has been solved as this algorithm will be able to be used in the SHL/SAR opcodes. The task of shifting towards the MSB has been solved through the iteration block, which will perform single shifts until the input has been shifted the required amount of times that was specified in the parameters. The task of setting the carry flag is the MSB was carried out of has been solved using the pull boolean. If the boolean is true after the iteration block has completed, this means that there was shift out of the MSB. Per the intel specification, this is only considered for the last shift performed. This is where the if condition statement is used to check if the value of the pull boolean is true, if so the carry flag is set.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
to test the iterative shifting procedure in the algorithm	The core algorithm will be tested separately from other tests. The parameter will be a byte array shifted twice	Input = [10,10] ShiftCount = 2	Normal	Output = [40,40]	This test will be necessary to determine whether the algorithm is implemented correctly. This means that it is important to use a simple test initially to show which parts of the algorithm are known to function.
To test whether the flags will be changed when the input shift count is zero	The ShiftCount parameter will be set to zero and a breakpoint will be used to check the flags afterwards.	ShiftCount = 0 (Input irrelevant)	Boundary	Empty flag set returned	This test data is necessary for this test because it is the only case where the behaviour will appear. This will be testing whether this special case will be handled correctly, as no other test would be able to demonstrate this.
Test the output of the carry flag	The input will be an array that will have its MSB pushed out of the array on the last iteration, such that the carry flag will be expected to be set	Input = [FF] ShiftCount = 1	Normal	Output = [FE] CF = True SF = True (because signed negative result)	This test data will always be an input that should result in the CF being set in the output. Therefore, this data is appropriate for this test. The test will identify any implementation bugs of the conditions that determine the carry flag such that any bugs can be found and fixed faster than if they had to be identified by other means.

ShiftRight algorithm



Pseudocode

```

DEF ShiftRight(byte[] input, byte count, bool isArithmetic)
    // Same procedure as in shift left, except the purpose is that this number of
    // shifts will shift all bytes out of the array in the direction of the LSB instead of the
    // MSB
    IF input.Length >= 8 THEN
        count = count AND 0x3F;
    ELSE
        count = count AND 0x1F;
    FI;
    int StartCount = count;

    // Return early if there is no shift to perform. The x86-64 specification defines that
    // all flags are not changed if this is the case (therefore return an empty flagset)
  
```

```

IF count == 0 THEN
    return input, new FlagSet();
FI;

// Copy the input into the result array by value to avoid modifying the input array
byte[] Result = (value)input;

FOR count > 0 DO
    bool Push = false;

    // ShiftRight will work backwards instead. -1 because length is one-based
    // and the array index is zero based.
    int i = Result.Length-1;
    FOR i >= 0 DO
        // Instead of masking the LSB, the MSB needs to be masked because
        // pulled from the LSB of the previous byte into the MSB of      this byte.          // the bit was
        int PushMask
        IF Push == true

THEN
    PushMask = 0x80; // (128, the MSB of a byte)
    FI;
    // If the LSB of the current byte is set, it is about to be pulled from this
    // next byte(the next index less than the current index).          // byte into the
    IF (Result[i] AND 1) == 1

THEN
    Push = true;
ELSE
    Push = false;
    FI;

    // Shift the current byte right by one, ORing on the push mask afterwards
    Result[i] = (Result[i] >> 1) OR PushMask;
    i++;
    ROF;
    count -= 1;
    ROF;

    // If the function is an arithmetic right shift, preserve the sign bit of the input
    // in the sign bit of the result by ORing it back on afterwards. E.g if the input
    // is negative, the output will too in an arithmetic shift.

    IF isArithmetic == true
    THEN
        Result[Result.Length-1] |= MSB(Input);
    FI;

    // Use provided method to determine SF PF and ZF
    FlagSet ResultFlags = new FlagSet().AutoDetermine(Result);

    // Another flag setting included only to accurately implement the x86-64 specification.
    IF StartCount == 1
    THEN
        IF isArithmetic == true
        THEN
            ResultFlags.Overflow = FlagState.OFF;
        ELSE
            // Overflow flag is set to the sign of the input
            IF MSB(Input) == 1

```

```

    THEN
        ResultFlags.Overflow = FlagState.ON;
    ELSE
        ResultFlags.Overflow = FlagState.OFF;
    FI;
FI;
RETURN Result, ResultFlags;

```

FED:

Explanation

The ShiftRight function works by iterating through each of the bytes in the input, applying the right shift function, and deciding whether the LSB was set and pushed into the next byte beneath that will be iterated next. Essentially, all bits will be shifted down one place, which is repeated for the number of shifts to perform.

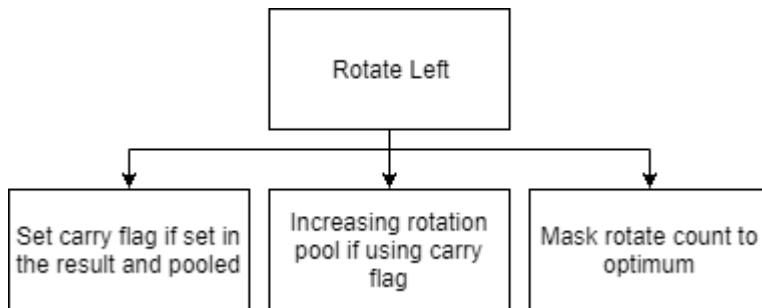
Justification

This is a complete solution to the shift right problem because all sub-problems identified through the stepwise refinement have been solved. The task of masking the shift count to the optimum value is performed at the start, as the count parameter is masked by the highest value that would affect the procedure. For example, right shifting a 4 byte input 500 times would be the same as shifting 4 times because all the bytes will be zero after the 4th shift. The task of performing the shift towards the LSB has been solved by using repeated iteration to apply shifts on each byte. Instead of the iteration counter increasing, the counter decreases because it allows the pull to be detected and then solved in the next iteration by ORing the MSB with the LSB of the previous iteration. The task of setting sign, parity, and zero flags has been solved by using the method provided by the FlagSet data type, as this was a common problem across the solution. The task of preserving the sign bit in an arithmetic shift has been solved by ORing the MSB of the result by the MSB of the input if the isArithmetic boolean was true in the parameters. This means that the two will have the same sign; if the input was positive, the OR did nothing as it would not be possible for the MSB to be non-zero after a right shift, otherwise the OR would set the MSB on when the input is negative.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test the iterative right shift procedure contained in the for loop	The function will be called with pre-defined input parameters then a breakpoint will be set afterwards to check the result	Input = [12,34] ShiftCount = 4	Normal	Output = [41,03]	This test data will demonstrate whether the implementation of the algorithm is successful. This will involve outlining any logical errors, which are likely as the algorithm gets complex with nested iterations. Therefore, it is appropriate to have a test specifically designed to check whether this has been performed correctly before being able to do other tests as they will depend on this element of the algorithm to work.
to test whether the flags will be changed when the input shift count is zero	The ShiftCount parameter will be set to zero and a breakpoint will be used to check the flags afterwards.	ShiftCount = 0 (Input irrelevant)	Boundary	Empty flag set returned	This test data is necessary for this test because it is the only case where the behaviour will appear. This will be testing whether this special case will be handled correctly, as no other test would be able to demonstrate this.
test the effect of the arithmetic boolean parameter	The function will be called with a negative signed input and the arithmetic boolean true, then the output checked afterwards using a breakpoint	Input = [CC,FF] ShiftCount = 1 arithmetic = true	Normal	Output = [E6, FF]	This test is necessary as the arithmetic boolean is used in conditional statements to execute other pieces of code, which would otherwise go untested. Therefore it is important to use an input that is crafted to trigger these conditions in order to check that they work.

RotateLeft algorithm



Pseudocode

```
DEF RotateLeft(byte[] input, byte rotateCount, bool useCarry, bool carryPresent)
    // A similar procedure to as in the shift instructions to reduce the number of
    // iterations necessary is used.
    byte StartCount;

    // Check if the carry flag is included as part of the rotation pool
    IF useCarry == true
        THEN
            // When reading the following, it is important to consider where identical
            // outputs are for different values of rotateCount. For example, rotating a
            // byte 9 times will loop it back to its original value, therefore to modulo
            // by nine will get the effective offset from the start(remember that the carry flag is used in the rotation pool,
            hence is nine not eight). This is all defined in the           // intel x86-64 specification therefore should be included in the method
            such
            // that the results of this method are realistic.
            SWITCH (RegisterCapacity)input.Length
                CASE BYTE
                    StartCount = (rotateCount AND 0x1F) % 9;
                    ESAC;
                CASE WORD
                    StartCount = (rotateCount AND 0x1F) % 17;
                    ESAC;
                CASE DWORD
                    StartCount = (rotateCount AND 0x1F);
                    ESAC;
                CASE QWORD
                    StartCount = (rotateCount AND 0x3F);
                    ESAC;
                HCTIWS;
            ELSE
                // If the carry flag is not included as part of the rotation pool, the same masks
                // as applied in the shift methods are applied
                IF (RegisterCapacity)input.Length == RegisterCapacity.QWORD
                    THEN
                        StartCount = bitRotates AND 0x3F;
                    ELSE
                        StartCount = bitRotates AND 0x1F;
                    FI;
                FI;
                // If the start count is zero, return early without changing any flags per the intel manual
                IF StartCount == 0
                    return input, new FlagSet();
```

```

FI;

// Copy the value of the input such that the input array is not modified directly
Result = (value)input;

// Pre-set if there is already a carry flag. This is the value that would be pushed into LSB
// I only have to worry about this if I am using the carry flag, and the carry flag is set, as will be shown in the algorithm.
bool Pull = carryPresent && useCarry;

// Perform the rotate operation on every bit, StartCount times. This is looped through
// this outer for loop. The following algorithm is very similar to the left shift algorithm
// except the important difference is that the bytes loop around instead of being
// discarded when shifted out of the MSB.
int RotatesPerformed = 0;
FOR RotatesPerformed < StartCount
DO
    int i = 0;
    FOR i < input.Length
    DO
        byte PullMask;
        // If pull is true, the same principle as in shift left applied. The
        // MSB of the previous index byte is going to be pulled into this
        // byte.
        IF Pull == true THEN
            PullMask = 1;
        ELSE
            PullMask = 0;
        FI;

        // If the MSB is set, it is about to be pulled into the next byte, next
        // iteration, as otherwise it would be cut off.
        IF MSB(Result[i]) > 0
THEN
        Pull = true;
ELSE
        Pull = false;
    FI;

    // Shift the existing byte and apply the mask, which will set the LSB on
    // after the shift if the MSB of the previous byte was shifted into this byte
    Result[i] = (Result[i] << 1) | Mask;
    i += 1;
    ROF;

    // If carry flag is in the rotation pool, move the value of the carry flag into
    // the LSB, then the value of the last MSB to be shifted(stored in the pull
    // boolean) into the carry flag, then continue the next iteration if applicable.
    IF useCarry == true THEN
        IF Carry == true THEN
            Result[0] = Result[0] OR 1;
        FI;
        Carry = Pull;
    ELSE IF Pull == true THEN
        // Otherwise, the value of the last MSB to be shifted is simply moved
        // to the first(assuming bits are not zero based) bit. (If there pull is
        // false, this would be implied already by the shift)
        Result[0] = Result[0] OR 1;
    FI;
}

```

```

FI;

// Increment counter
RotatesPerformed += 1;

ROF;

// "The SF, ZF, AF, and PF flags are always unaffected." - Intel manual May 2019 edition
FlagSet ResultFlags = new FlagSet();

// Apply the result value of the carry flag to the flag in the results, as the actual
// carry flag was not affected during the operation, only a boolean representing it.
// This is because it is unnecessary to depend on the control unit to set the actual
// carry flag as the value of the CF will not be read until after the function finishes.

IF useCarry == true THEN
    IF Carry == true THEN
        ResultFlags.Carry = FlagState.ON;
    ELSE
        ResultFlags.Carry = FlagState.OFF;
    FI;
ELSE
    // A special case defined in the x86-64 specification, "For ROL and ROR
    // instructions, the original value of the CF
    // is not part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other"
    IF LSB(Result) > 0 THEN
        ResultFlags.Carry = FlagState.ON;
    ELSE
        ResultFlags.Carry = FlagState.OFF;
    FI;

    // Another special case defined in the specification. The overflow flag is
    // set to the XOR of the MSB of the result and the carry flag if and only if
    // the input bits were rotated by one place only.
    IF StartCount == 1 THEN
        IF MSB(Result) ^ ResultFlags.Carry > 0 THEN
            ResultFlags.Overflow = FlagState.ON;
        ELSE
            ResultFlags.Overflow = FlagState.OFF;
        FI;
    FI;
FI;

// Return results afterwards
RETURN Result, ResultFlags;

```

FED;

Explanation

The Rotate Left algorithm works by shifting all bits one place to the right but preserving the LSB by moving it into the CF/MSB depending on the parameters passed into the function, unlike the shift left operation which will discard of this bit. If useCarry is true, the CF will be included in the rotation pool(used as an extra bit, MSB will be rotated into the CF). If carryPresent is true, the Carry variable will start true, then on the first rotate will be rotated into the LSB. This procedure will be used in the RCL/ROL opcodes. FlagSet.AutoDetermine() is not used in this algorithm as the intel manual states, "The SF, ZF, AF, and PF flags are always unaffected."

Justification

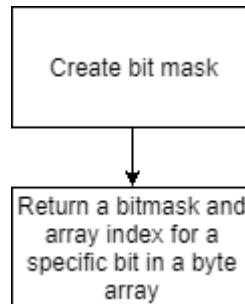
The RotateLeft algorithm is a complete solution because all sub-problems identified in the decomposition have been solved. The task of masking the rotate count was solved at the beginning of the procedure. The pseudocode adapted for the algorithm implements the same conditions as the pseudocode in the official intel manual, therefore this task has been solved successfully. The task of left rotation has been solved by adapting the left shift algorithm to instead of discarding of the MSB after each shift, after each shift is performed, the MSB is ORed onto the LSB of the result or

into the CF, whichever applicable. The task of setting the carry flag after the operation is complete has been solved by using an if statement to check if the local variable “Carry” is true at the end of the procedure.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test the iterative left rotation procedure contained in the for loop	The function will be called with pre-defined input parameters then a breakpoint will be set afterwards to check the result	Input = [12,34] RotateCount = 4	Normal	Output = [23, 41]	This test data will demonstrate whether the implementation of the algorithm is successful. This will involve outlining any logical errors, which are likely as the algorithm gets complex with nested iterations. Therefore, it is necessary to test the components of the problem separately such that it can be identified at a later date which component contains flaws or bugs.
to test whether the flags will be changed when the input rotate count is zero	The RotateCount parameter will be set to zero and a breakpoint will be used to check the flags afterwards.	RotateCount = 0 (Input irrelevant)	Boundary	Empty flag set returned	This test data is necessary for this test because it is the only case where the behaviour will appear. This will be testing whether this special case will be handled correctly, as no other test would be able to demonstrate this.
test the effect of the useCarry boolean parameter	The function will be called with the useCarry and carryPresent variables set and the input containing bytes with the MSB set to rotate, then the output checked afterwards using a breakpoint	Input = [00,80] RotateCount = 1 useCarry = true carryPresent = true	Normal	Output = [01, 00] CF = true OF = true	This test is necessary to check whether the implementation of the algorithm works correctly when the CF is used in the rotation pool. This test data is appropriate because it has been specially crafted to test every part of the algorithm that involves the carry flag. As the CF is on in the input(carryPresent), it is expected that the bit will be rotated into the LSB of the array. The array also having the MSB set will cause the CF to be set in the output, such that this test covers the entire scope of the carry flag in the procedure.

GetBitMask algorithm



Pseudocode

```
DEF GetBitMask(int bit)
    // Get the index of the byte that the bit would be in the byte array. 8 bits in a byte so
    // the quotient of the bit and 8 will give the zero-based index in the array.
    byte index = bit / 8;

    // Create a mask of where the bit is in the byte.
    byte mask = 1 << ((bit-1) % 8);
    return index, mask;
```

FED;

Explanation

Return an index and mask used to locate the value of a specific bit in a byte. For example, GetBitMask(4) would return (0, 16), because the 5th bit lies in the first byte, and $2^4 = 16$. (The input to the function is a zero based index, therefore the GetBitMask(0) returns bit 1)

This works by dividing by 8 to get the index(bytes in this context are all 8 bits), then creating a mask to find the bit in that byte. The mask is constructed by shifting a 1 bit

to that position(therefore if you were to AND by this mask, the only bit that could be set in the result would be this byte. This is done by subtracting one from the bit then modulo by 8(take the remainder). This works because divide was already used to find the index of the byte in the array where the bit lies, so the remainder will find the position in that byte. One has to be subtracted because when shifting by one, there is already a value of 1(the first bit) set, so essentially one shift is already performed by having 1 already set, but shifting zero initially would not work because the result would always be zero.

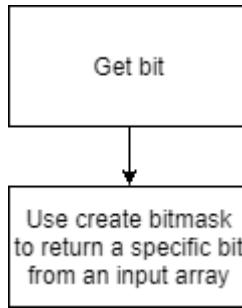
Justification

This is a complete solution to the problem because it is a suitable computational solution to any input, so long as there are 8 bits in a byte. The computational solution used to solve the sub-problem identified in the stepwise refinement is suitable as the array index can be found by dividing by 8 using integer division, then the bitmask is crafted using the remainder, which would not depend on any specific range of inputs as this process is applicable to any sized integer and can be used on any sized input, performing calculations that could not be performed effectively manually as arithmetic operations such as division that computers are very fast at have been used.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test the function for bit in the first byte	The input parameter will be an integer less than 8	bit = 3	Normal	index = 0 mask = 4	This test will be necessary to check whether the algorithm to produce a mask works, which will help in identifying where bugs are present in Test #2, as if Test#1 fails, it will likely cause Test #2 to fail. This test data will always result in the index being zero because there are 8 bits in a byte.
test the function for bit outside the first byte	The input parameter will be an integer greater than 8	bit = 32	Normal	index = 3 mask = 0x80	This test will test the overall function of the procedure, as the same code is executed for every input, therefore this test will show fully whether the algorithm is working.

GetBit algorithm



Pseudocode

```

DEF GetBit(byte[] input, int bit)
    // Use get mask to find the position in the array
    int Index, int Mask = GetBitMask(bit);

    // Use the mask to get the value of the bit. Note that this bit will still represent a power
    // of two, i.e. could be 128, 64, 32, 16, 8, 4, 2, or 1.
    int WeightedBit = input[index] AND Mask;
    // This if statement will deweight the bit, which is what the caller would want as it is
    // isn't exactly representative of anything outside of the byte it is in. It would not
    // cover all the columns of powers of two outside of the fetched byte, only the single byte it was fetched from. The only
    possible outputs of the algorithm will be 1 if the bit was
    // set and 0 if it was not
    IF WeightedBit > 0 THEN
        RETURN 1;
    ELSE
        RETURN 0;
    FI;
FED
  
```

Explanation

This algorithm makes use of `GetBitMask()` to simplify a procedure that is the most probable application of `GetBitMask()`, but still allowing it to be used for other cases. The algorithm will determine if a specific bit is set in the given input array. This is done using `GetBitMask()` and ANDing the byte in the input at the returned index by the returned mask. This bit will still be a power of two, hence the if statement will remove this value before returning the

value to the caller, such that the state of the bit will be returned rather than its value i.e. 1 for on 0 for off, independent of position in the byte.

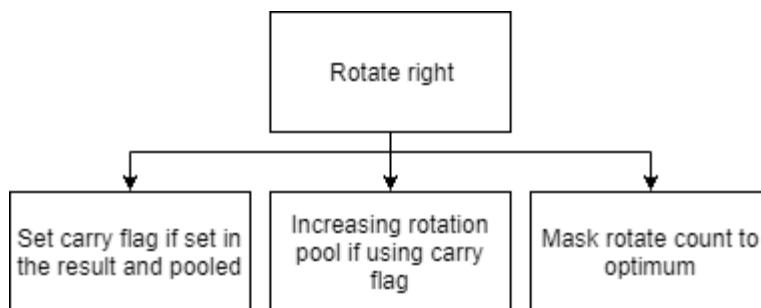
Justification

This algorithm is a complete solution to the problem because it makes use of the existing GetBitMask(), which was already shown to be a complete solution, in order to solve the problem. GetBitMask() is used to find the position and mask used to locate the byte in the array, which is then deweighted. The deweighting procedure is a complete solution because it will work for bits in any position of the byte. This is because instead of explicitly stating weights, such as IF WeightedBit == 1, the greater than condition will cover all the powers of two, therefore if there are any bits set in the masked result, the bit must be set so one should be returned. Unlike GetBiskMask(), this requires the inputs to be correct. If the bit is outside of the array, there will be an exception because the array will attempt to be accessed outside of its bounds. The best course of action is to leave this exception as there is no other approach that can be taken that would provide the correct output to the caller, therefore it is best that the developer/maintainer is told rather than allowing bugs in the code.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test whether a bit can be found under normal conditions	The bit parameter will point to a bit present in the input array	(bit parameter is zero based) bit = 10 input = [00, 04]	Normal	output = 1	This test data will be necessary to demonstrate that the input will work with valid inputs as the algorithm is identical for all valid inputs, hence only requires this one test to show that it is working. This would still show any logical errors, such as if the wrong index was checked.
test the event of erroneous parameters	The bit parameter will point to a bit not present in the input array	bit = 10 input = [00]	Erroneous	ArrayOutOfBoundsException	This test data will show that the function will result in an exception when the parameters are invalid. This will be important as if there is no exception, the input must be invalid, which would not be apparent until the code is used in production or reused in another project.

RotateRight algorithm



Pseudocode

```

DEF RotateRight(byte[] input, byte rotateCount, bool useCarry, bool carryPresent)
    // The same masking criteria in RotateLeft is applied
    byte StartCount;
    // Check if the carry flag is included as part of the rotation pool
    IF useCarry == true
        THEN
            SWITCH (RegisterCapacity)input.Length
                CASE BYTE

```

```

        StartCount = (rotateCount AND 0x1F) % 9;
        ESAC;
        CASE WORD
            StartCount = (rotateCount AND 0x1F) % 17;
        ESAC;
        CASE DWORD
            StartCount = (rotateCount AND 0x1F);
        ESAC;
        CASE QWORD
            StartCount = (rotateCount AND 0x3F);
        ESAC;
    HCTIWS;
ELSE
// If the carry flag is not included as part of the rotation pool, the same masks
// as applied in the shift methods are applied
IF (RegisterCapacity)input.Length == RegisterCapacity.QWORD
THEN
    StartCount = bitRotates AND 0x3F;
ELSE
    StartCount = bitRotates AND 0x1F;
FI;
FI;

// If the start count is zero, return early without changing any flags per the intel manual
IF StartCount == 0
    return input, new FlagSet();
FI;

// Copy the value of the input such that the input array is not modified directly
Result = (value)input;

// Pre-set if there is already a carry flag. This is the value that would be pushed into LSB
// If there is a CF set, it will become the MSB of result. Rotation of the byte array pool
// alone can be handled inside the algorithm.
bool Push = carryPresent && useCarry;

// Perform a right shift on every byte, except after each iteration, instead of discarding
// the LSB, rotate it into the MSB of the result.
int RotatesPerformed = 0;
FOR RotatesPerformed < StartCount
DO
    int i = input.Length-1;
    FOR i >= 0 DO
        byte PushMask;
        // If push is true, the same principle as in shift right applied. The
        // LSB of the previous byte is going to be rotated into the MSB of
        // the current byte. 0x80, 128d, is the highest bit in a single byte,
        // therefore ORing the shifted byte by it will set it. The MSB will be
        // empty after the shift because all bits were moved one place to the
        // right.
        IF Push == true THEN
            PushMask = 0x80;
        ELSE
            PushMask = 0;
        FI;
        // If the MSB is set, it is about to be pulled into the next byte, next
        // iteration, as otherwise it would be cut off.

```

```

IF MSB(Result[i]) > 0
THEN
    Push = true;
ELSE
    Push = false;
    FI;

    // Shift the existing byte and apply the mask, which will set the LSB on
    // after the shift if the MSB of the previous byte was shifted into this byte
    Result[i] = (Result[i] << 1) | Mask;
    i += 1;
ROF;

// If carry flag is in the rotation pool, move the value of the carry flag into
// the MSB, then the value of the last LSB to be shifted(stored in the pull
// boolean; the LSB of the first byte before the shift), into the carry flag, then continue the next iteration if
applicable.

IF useCarry == true THEN
    IF Carry == true THEN
        Result[Result.Length-1] = Result[0] OR 0x80;
        FI;
        Carry = Push;
    ELSE IF Pull = true THEN
        // Otherwise, OR the MSB by the previous LSB(which was set
        // because pull was true, it is already zero if false so no further
        // action is required).
        Result[Result.Length-1] = Result[Result.Length-1] OR 0x80;
        FI;
    // Increment counter
    RotatesPerformed += 1;
ROF;

// "The SF, ZF, AF, and PF flags are always unaffected." - Intel manual May 2019 edition
FlagSet ResultFlags = new FlagSet();
// Apply the result value of the carry flag to the flag in the results, as the actual
// carry flag was not affected during the operation, only a boolean representing it.
// This is because it is unnecessary to depend on the control unit to set the actual
// carry flag as the value of the CF will not be read until after the function finishes.
IF useCarry == true THEN
    IF Carry == true THEN
        ResultFlags.Carry = FlagState.ON;
    ELSE
        ResultFlags.Carry = FlagState.OFF;
    FI;
ELSE
    // In a rotate right operation without the CF in the pool, the CF is set to the
    // MSB of the result.
    IF MSB(Result) > 0 THEN
        ResultFlags.Carry = FlagState.ON;
    ELSE
        ResultFlags.Carry = FlagState.OFF;
    FI;

    // Another special case defined in the specification. The overflow flag is
    // set to the XOR of the MSB of the result and the second to last bit of the
    // result.

```

```

IF StartCount == 1 THEN
    // *8 because 8 bits per byte. -1 because the function is zero based
    IF MSB(Result) ^ GetBit(Result, Result.Length*8-1) > 0 THEN
        ResultFlags.Overflow = FlagState.ON;
    ELSE
        ResultFlags.Overflow = FlagState.OFF;
    FI;
FI;

// Return results afterwards
RETURN Result, ResultFlags;

```

FED;

Explanation

The RotateRight algorithm works by shifting all bits in the input array one place to the right, then rotating the LSB that would have been discarded in a shift right algorithm into the MSB/CF depending on whether the CF is in the rotation pool. This is repeated for as many iterations as specified in the rotateCount parameter. FlagSet.AutoDetermine() is not used in this algorithm as the intel manual states, “The SF, ZF, AF, and PF flags are always unaffected.”

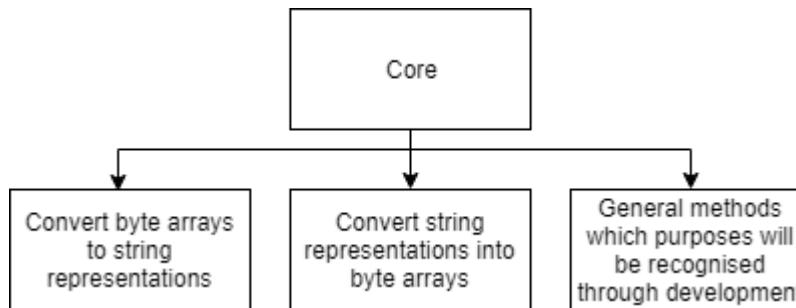
Justification

This is a complete solution to the problem because all of the sub-tasks have been solved. The sub-task of masking the rotate count to optimum has been performed by re-using the code from the shift left algorithm. This was more appropriate than creating another method because these are the only two instances that the code will be repeated and are not likely to change in the development process as their purpose is to comply with the x86-64 specification. The sub-task of increasing the rotation pool if using the carry flag has been solved by using a carry variable to represent the carry flag, then making decisions based on its state after performing rotate on all the bytes, such as ORing the MSB if carry is true, because in reality the CF represents the MSB, but an individual bit cannot be appended onto a byte array, therefore the boolean data type is used to represent the on and off state. The task of rotating right has been solved by adapting the shift right algorithm to preserve the LSB of the shifted array by rotating it into the MSB, hence “rotate right” as all bits are rotating in the direction of the LSB, then looping back round. This is done through a for loop that iteratively performs the shift right procedure on each byte, then afterwards evaluates whether the MSB is to be set on(in the case that the LSB was true before). The sub-task of setting the carry flag in the result has been solved by using the carry boolean variable to determine whether the CF should be set on or off. If and only if the carry boolean is true, on the last rotation, a one bit(the LSB set on) was rotated into the carry, therefore in this case the CF should be set on in the result, and otherwise set off in the result.

Iterative test data

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
test the iterative right rotation procedure contained in the for loop	The function will be called with pre-defined input parameters then a breakpoint will be set afterwards to check the result	Input = [12,34] RotateCount = 3	Normal	Output = [82,46]	This test data will demonstrate whether the implementation of the algorithm is successful. This will involve outlining any logical errors, which are likely as the algorithm gets complex with nested iterations. Therefore, it is necessary to test the components of the problem separately such that it can be identified at a later date which component contains flaws or bugs.
to test whether the flags will be changed when the input rotate count is zero	The rotateCount parameter will be set to zero and a breakpoint will be used to check the flags afterwards.	RotateCount = 0 (Input irrelevant)	Boundary	Empty flag set returned	This test data is necessary for this test because it is the only case where the behaviour will appear. This will be testing whether this special case will be handled correctly, as no other test would be able to demonstrate this.
test the effect of the useCarry boolean parameter	The function will be called with the useCarry and carryPresent variables set and the input containing bytes with the LSB set to rotate, then the output checked afterwards using a breakpoint	Input = [01,00] RotateCount = 1 useCarry = true carryPresent = true	Normal	Output = [00, 80] CF = true	This test is necessary to check whether the implementation of the algorithm works correctly when the CF is used in the rotation pool. This test data is appropriate because it has been specially crafted to test every part of the algorithm that involves the carry flag. As the CF is on in the input(carryPresent), it is expected that the bit will be rotated into the MSB of the array. The array also having the LSB set will cause the CF to be set in the output, such that this test covers the entire scope of the carry flag in the procedure.

Core library



Explanation

The core library will mostly hold any utility functions whose need is identified during the development process, such as solving problems specific to the C# language that could not be identified when planning the design.

Justification

There must be a distinct place for these functions be stored in order for the design to make sense. This is because they could be very general purpose and used in all parts of the solution, but would not make sense to include them in the bitwise library for example, where then every class that implements them has to import the entire bitwise class, which could be misleading about the purpose of said class.

Convert byte arrays to string representations

Pseudocode

```

DEF BytesToString(byte[] input)
    string Output = "";
    bool Significant = false;
    int i = 0;
    FOR i < input.Length DO
        // Start from the end array because the most significant bytes are at
        // the end of the array.
        byte CurrentByte = input[input.Length - i - 1];

        // Once the first non-zero byte in the array is read, all the bytes including
        // and after that will be significant.
        IF Cursor != 0 THEN
            Significant = true;
        FI;

        // Only add bytes to the end that affect the value
        IF Significant == true
        THEN

            // Convert byte to a hex string
            Output += CurrentByte.ToString("X");
        FI;
        i++;
    ROF;
    RETURN Output;
FED;

```

Explanation

Each byte in the array is iterated and appended to the output string. The byte array is iterated in reverse because the algorithm will not include insignificant zeros. For example instead of outputting “00000012”, the output will be “12”. This would be the case when the input byte array is greater than the size necessary to hold that value.

Justification

This is a complete solution to the problem because it will work for any input, which is necessary because many layers of the program will call the function with many different parameters. This is the case because the for loop iterates through the length of the byte array in reverse, such that the most significant bytes are read first. This will determine whether they are significant or not because after the first non-zero, the significant variable becomes true and from then on all bytes will be considered significant. For example, the zero in 10 is significant because otherwise the value would be 1, but the first zero in 010 is not.

Validate and convert string representations into byte arrays

```

DEF CharToByte(char input)
    // If the character is one of: 0,1,2,3,4,5,6,7,8,9 , subtract the ascii value of ‘0’ to
    // get the value
    IF (input >= ‘0’) AND (input <= ‘9’) THEN
        RETURN (input - ‘0’);

    // If the character is one of: A,B,C,D,E,F, subtract the ascii value of ‘A’ to get
    // the respective value but between one and ten, so add 0xA on afterwards to
    // get the desired value.
    ELSE IF (input >= ‘A’) AND (input <= ‘F’) THEN
        RETURN (input - ‘A’ + 0xA);

    // The same as the previous except will work for lowercase letters.
    ELSE IF (input >= ‘a’) AND (input <= ‘f’) THEN
        RETURN (input - ‘a’ + 0xA);

    FI;
    // Indicate invalid user input if it did not meet the above criteria
    RETURN false;
FED;

```

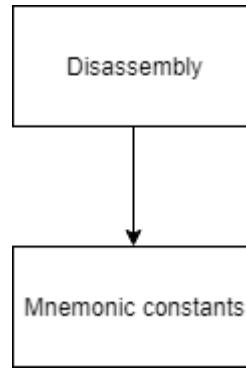
Explanation of validation

This validation algorithm works by subtracting the ascii values of the input to convert a character into the numerical form it represents. Another way to think about it is that it calculates the offset between '0' and the character, or 'A'/'a' and the character. If this value was not in between or one of 0 and 9 for any of the cases, it must not be a valid character.

Justification of validation

This is a complete solution to the problem because it allows for all forms of valid hexadecimal; upper and lower case, and also will tell the caller that there is an invalid user input. This is possible by performing the operation character by character, such that the caller can use iteration to iterate through the string then handle the invalid inputs on a case-by-case basis rather than being told that the entire string was invalid without any further information. This will be useful when handling file inputs which contain text, such that if the file contains information that cannot be parsed, they will know which character it was rather than having to search through an entire file.

Disassembly library



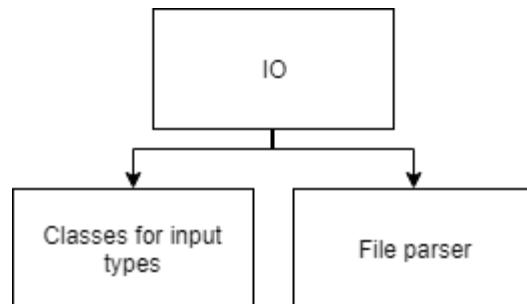
Explanation

The disassembly library will contain the mnemonic constants for various operations of assembly such as condition codes and word sizes.

Justification

Storing the mnemonic constants in one place will reduce code repetition and save storage space as multiple copies of the same string will not have to be stored. This is a complete solution to the problem because as stated at the beginning of the design section, the utility library will be used by all layers of the program instead of having the layers depend on each other, which can be achieved by instead of keeping disassembly constants spread around the program, they can all be stored in the disassembly library. In this way, when one needs to be changed or added, the changes will be seen throughout the entire solution rather than having inconsistencies when one part of the program uses a different mnemonic to another.

IO library

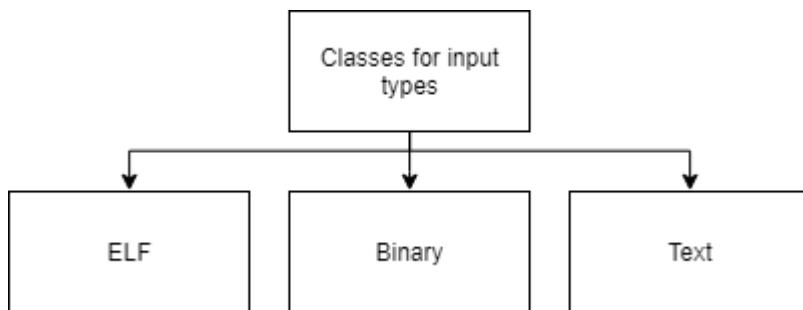


Explanation

The IO library will provide methods for reading and parsing files that can be reused throughout the solution

Justification

The input files must meet certain criteria to be able to be interpreted. In order to generalise this criteria, and hence generalise what a "valid file" is in the solution, the parsing of files will be centralised in the IO class, which will also automate the procedures to extract required information from the files.



Classes for input types

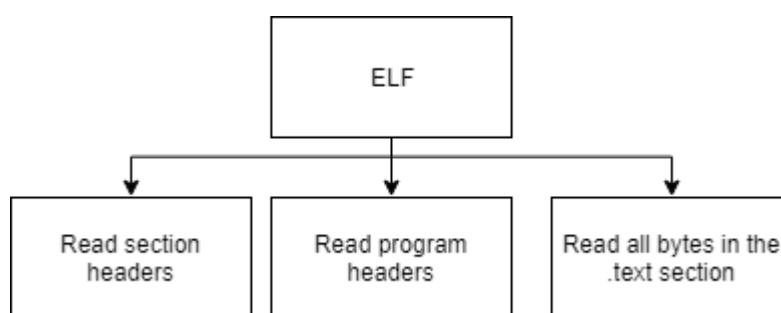
Explanation

Each input type will be broken down into a smaller and more manageable module that will interpret the files individually. For example, given a path, the ELF class will load and parse the ELF file.

Justification

This is a suitable extent of decomposition because the procedures for parsing different file types range greatly, there is no general method for loading any file. Input for one file may be valid, but invalid for any other file type. For this reason, the problem needs to be modulated and tackled separately, using different routines to handle different files.

ELF Class



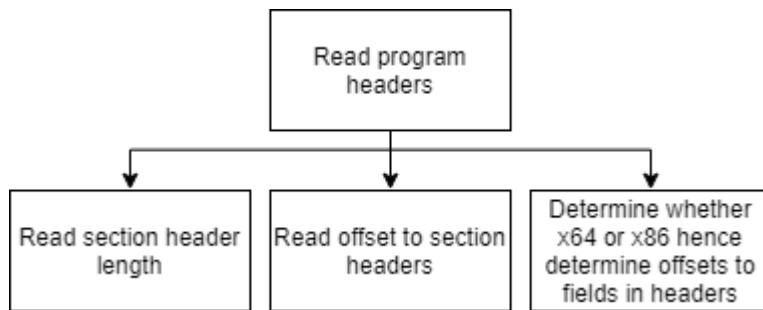
Explanation

The ELF file class will parse a given ELF file into parsed instructions that can be executed by the control unit.

Justification

This problem has to be divided into three subproblems. This is because their order of execution is important. The program headers must be read first in because they contain the location of the section headers. The section headers must be read second because they contain the location of the ".text" section; the section that contains the user's code. For this reason, they must be separated and pipelined into one and other.

Read program headers



Explanation

The program headers are located at the start of the ELF file. The program headers must be read in order to determine where the ".strtab" section is located in the file. Then it will be possible to determine where the .text section is by using the section header index provided by the strtab(the index of the strtab is provided in the program header, but no other).

Justification

This needs to be divided into three different tasks because the ELF header section contains a lot of information that is irrelevant in this context. For this reason, it is necessary to abstract the other headers and focus only on the required information. To do this, the best approach will be to locate the required headers individually rather than construct the location all at once as many procedures will be required to solve the problem.

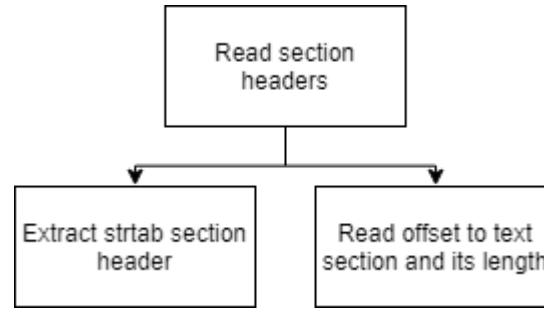
Validation

The location of the desired headers is dependent on the “ELF class header”, which will allow decisions to be made in order to determine whether the ELF file is x86-64, x86, or another architecture. The offsets to the headers after the class header will change depending on the value of the class header. If the elf class is not supported, the event will be logged and the procedure will abort.

Justification of validation

This is the best approach to validating the ELF file because the class header is always at a fixed position. This means that for any ELF, supported or unsupported in the program, will be able to be validated procedurally. This will allow for more meaningful log messages because the reason why the input will not be accepted is known, rather than a general statement such as “Invalid input”. It is necessary that the procedure aborts because there could be exceptions later in the processing of the ELF otherwise. Moreover the instructions would not be interpreted properly by the CU because they are not designed for this architecture; the instruction set is completely different. For this reason, the user has no reason to use the ELF file of another architecture so the best approach is to alert them of the event as it likely they clicked on the wrong file.

Read section headers



Explanation

The section headers will be used to locate the strtab section header. This will be done by cutting out the portion of the file that contains the strtab and iterating through the section names until “.text” is found. Once the .text is found, its index in the strtab header will be the same as its index in the section header table.

Justification

This problem has to be divided because extracting the strtab can act as an input to the finding “.text” method, such that the process can then be pipelined. The extracted strtab section header can be used as an input to the method that will find the index of the text section. This will allow for easier maintenance and development in the future as the procedure of extracting the strtab will not change; it has existed like this for decades. However, in the future it is possible that more sections will be extracted for different purposes that are not currently in the scope of the project. This will allow for extensibility as mentioned in the Open/Closed principle success criterion as another class can inherit from the ELF class and use the strtab extension method without having to use the same method of parsing.

Validation

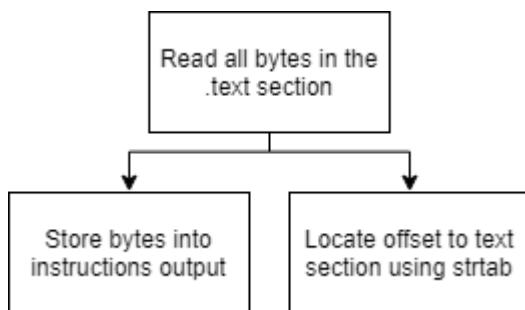
Validation will have to be applied to this solution as there is a possibility that there is no text section. This would imply that the ELF file is corrupt, or not an ELF file. To do this, iteration will be used to search the strtab. For example,

```
DEF ValidateStrtab(string strtab)
    IF strtab.Contains(“.text”) THEN
        RETURN true;
    ELSE
        RETURN false;
    FI;
FED;
```

Justification of validation

If validation is not applied to the solution of this sub-problem, there is the possibility of an error after the function returns when the text section is attempted to be read. For this reason, the procedure is required, and afterwards the caller can log an error. The program does not need to exit in this instance because the user will be able to go back and select the correct file, or fix the file and input it again.

Reading the text section



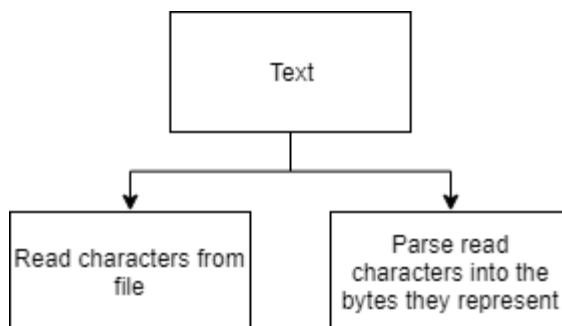
Explanation

The task of reading the text section can be divided into two different problems. The problem that must be tackled first is to locate where to read the text file from. This will be used from the output of the section header parse routine which returned the index. The index can then be used to calculate the offset by multiplying it by the length of every section header and adding on the offset to which all section headers are stored (They are all stored together, these headers were found in the program header table). After the offset to the text section has been found, the Start Address and End Address headers can be read which will indicate where memory needs to be read from. Storing the bytes into the instructions output must be handled afterwards, because it depends on this information. This problem can be solved by using the start address and end address and extracting the bytes in between from the file. This will contain all of the users instructions, and can then be used as an input to the control unit.

Justification

This problem has to be divided because the start address and end address must be read before the instructions can be read from the file, else it would not be known where the instructions are in the file. Because of this, pipelining can be used to pass the output of the procedure that reads the .text section headers and returns the start and end address, into the input extracts the instructions and store them in a variable that the user can then use as the output.

TXT file



Explanation

The TXT file class will read a text file and interpret as instructions. For example, the string "01D8" would decode to 0x01, 0xD8, then be parsed as the instruction it represents. The text stored in the file is encoded in ASCII; the bytes are not the same as the character they represent. To perform this conversion, the predefined procedure, CharToByte, in the core library will be used.

Justification

This problem has to be divided into two sub-problems because certain validation has to be performed before the file can be parsed. To be parsed as hexadecimal, the file must be a multiple of two in length. This means that the file must be read into an array, then extended to the nearest multiple of two if applicable before the array can be used as an input to the parsing procedure.

Usability feature

The TXT class is a usability feature. This is because it saves the user's time as they do not have to assemble the code into machine code or install the software to do so. This will allow them to use the output of web assemblers, such as defuse.ca, copy the output into a file and load the code into the program.

Justification of usability feature

This is a necessary choice of usability feature because the stakeholders are new users to assembly. This means that they will not have experience with the programs that would be required to assemble the code, such that they will face the same issues as the alternative software identified in the research section as they would still have to use advanced software to use VMCS.

Usability feature

Another usability feature will be the ability to load code from the clipboard. This will be a similar procedure to loading a text file, except data will be extracted from the windows clipboard instead of loading a file.

Justification of usability feature

It is important to include this usability feature as it will allow instructions to quickly be loaded into the program. For example, the user could copy the ascii code from another application and load it directly, without having to open a file editor or locate the file in the file dialog, however the option to load from a file will still be open for users, for example, in the case that they download the file from the internet.

Validation

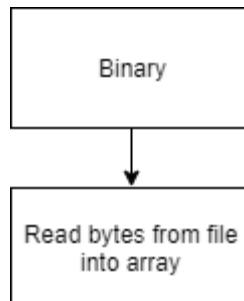
Validation must be applied to this sub-problem. This is because the input file may not contain valid hex characters. The validation applied will be flexible, ignoring any invalid characters and only parsing valid hexadecimal characters, e.g

```
private string ValidChars = "1234567890abcdefABCDEF";
DEF Validate(string text)
    int i = 0;
    string Output = "";
    FOR i < text.Length DO
        IF ValidChars.Contains(text[i]) THEN
            Output += text[i];
        FI;
        i++;
    ROF;
    RETURN Output;
FED;
```

Justification of validation

This is the best approach to validating the input because the likely cause of invalid characters is where the user has copied the text from. There may be a newline at the end or a whitespace. It would be counter-productive to force users to remove these characters from the string as being able to import files in text is a usability features designed to improve their experience.

Binary file



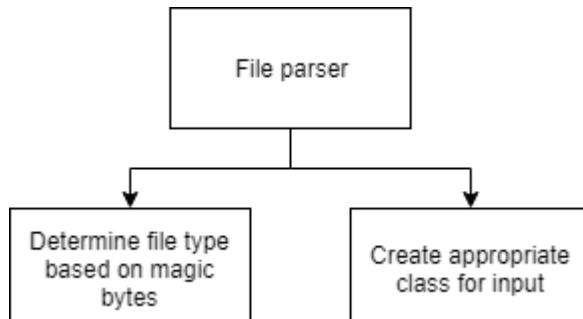
Explanation

The binary file class will read a file and parse all the bytes it contains as instructions, without any extra validation or parsing procedures. This is because the source the user obtained the file from may have assembled the instructions into a file. This can still be read by interpreting the file as raw instructions, which is the same result that every file type will be parsed into. For that reason, there is no parsing or validation required in this instance.

Justification

This task needs to be separated from the other classes because the other classes such as ELF and TXT will apply validation procedures before the file parsed, therefore if a binary file was used in place of one of these classes, there may be an exception when the class tries to parse it, as binary files are not intended to be parsed.

Parser



Explanation

The file parser class will use the file type classes to provide a general interface that can be used by other classes to abstract the need for them to understand the different file types available.

Justification

This sub-problem has to be split into two different tasks, as the file type must be determined before the appropriate class can be used to parse the file. Because of this, the two tasks will be separate procedures, which will allow for the first task to be pipelined into the second task to simplify the procedure.

Determine file type based on magic bytes

Explanation

The file type will be determined using the magic bytes of a file. This can be determined by reading the first few bytes of a file, then comparing it against a predefined list to check if it is a known sequence of magic bytes. For example, an ELF file always begins with the string “ELF”.

Justification

This task will be necessary to abstract the need for user or maintainer to understand the available file types. They can select a file and the program will load the instructions. This cannot be done for binary files and text files because they have no signature; they are just bytes in a file, such that the user will have to specify which they would like to interpret the input as.

Validation

There must be validation in this procedure because it has to be checked that there are enough bytes in the file to read the signature(4 bytes). The following algorithm could be used,

```
DEF ValidateFile(FILE inputFile)

    // Check if enough bytes in the file for at least a signature
    IF inputFile.Size < 4 THEN
        RETURN false;
    ELSE
        RETURN true;
    FI;
FED;
```

The algorithm will check if the file is big enough to read the signature from. If the size is equal to 4, there will likely be an error handled later as there will only be a file signature; no code.

Justification of validation

This validation method is appropriate because at this point of time in the code, only the magic byte signature needs to be read before continuing. Once the file type is determined, further validation can take place separately as each file will have different procedures required to validate that it is in the correct format. However, if the validation was not performed, there could be an exception when the magic signature is attempted to be read, but there are not enough bytes in the file. This exception would otherwise go unhandled and cause the program to crash. By validating it before this has the possibility to happen, more appropriate handling procedures can be taken such as telling the user that the file is incorrect and to select another.

Create appropriate class for input

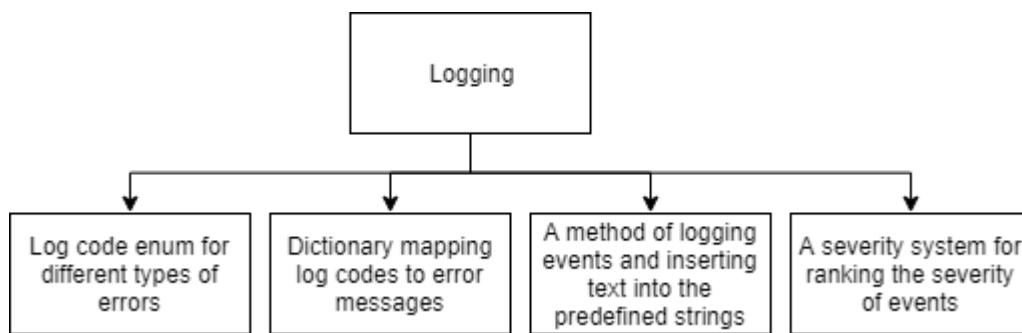
Explanation

A dictionary will be used to map a sequence of magic bytes to a file class. This will then automatically parse the file and store the output in a variable that can be accessed by the caller.

Justification

This task is necessary to simplify the process of creating new file type modules and abstracts the need for the programmer of another class to understand how the file type classes function, as the only input will be a byte array containing the instructions, rather than having to parse the file themselves.

Logging library



Explanation

The logging library will be used to handle any exceptions or events throughout the solution. For example, if the user inputs an invalid file, this would go through the logging system, such that they can go back and see what happened in the log file at a later date.

Justification

It is necessary to keep the logging modules separate from the rest of the code in order to generalise certain routines. For example, the date format must be consistent. If the event was written to the log file by the module which the event took place, a different function might be used to determine the timestamp, such that it would be confusing to use the log file.

Log code enum

Explanation

The log code enum will hold a unique event id for every event. For example, there will be an event along the lines of "File not found", which will use the same event in every module that opens files. This will allow the user to search through the log file by an error code to find the instances of this event.

Justification

This enum will be necessary to make the purpose of code clearer and more maintainable. Having integers that represent error codes in the code will get very confusing; the maintainer would have to be familiar with every error code. Instead, the enum will assign a name to each error code that is more descriptive of its purpose than an arbitrary number.

Dictionary mapping log codes to error messages

Explanation

The dictionary will use a log code(enum member) as a key and a string as a value. Using the key will return the string of the error that is to be displayed to the user, which would state what had gone wrong, e.g "Directory does not exist".

Justification

This is a necessary approach because the error messages must be consistent to simplify the log file and generalise error messages. For example, if every module had a different wording for "File not found", "Could not find file", etc, it would become very hard to locate information in the log file.

A severity system for ranking the severity of events

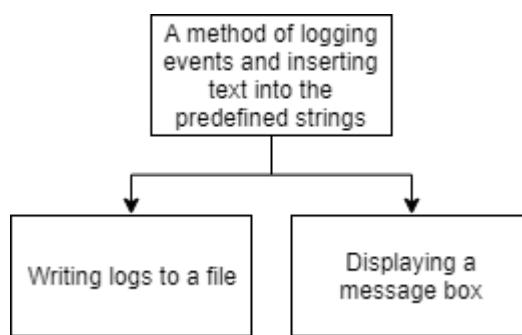
Explanation

Another enum will be used to represent the different severities of events. For example, one event could be critical, one could be a warning, etc. This will also have another dictionary that maps events to their severity. Different severities may change how the event is handled; a critical event may cause the program to exit, a warning may only show a message box to the user.

Justification

This will be necessary to give the user a general impression of how the error may affect the running program. For example, if the error is just a warning, they can likely move on and continue what they are doing, but if the error is of critical severity, the user will be more inclined to make sure they are prepared for a crash.

A method of logging events and inserting text into the predefined strings



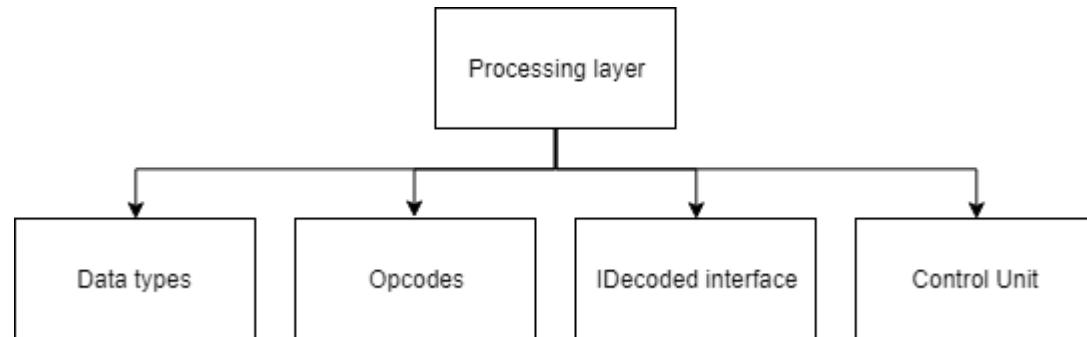
Explanation

The string will have formatting literals stored that are stored in the logcode-string dictionary. These will allow the caller to specify some changes to the string that are more descriptive of the error whilst maintaining consistency in the main error statement. For example, instead of "File not found", "File not found: input.txt does not exist". These formatting literals will be replaced using `string.Format()`, which will replace them with the caller-specified text. The necessary procedures to alert the user will then be taken, such as writing the new and complete message to a file and displaying a message box.

Justification

This is a necessary design feature because callers will be able to offer some more details or guidance to the user on how to handle the error. For example, the error may indicate a bug in the add function, however the error message may only be an `ArrayOutOfBoundsException` error message. To add meaning to the message, the caller will specify what may be affected by the error and whether the user should restart the program. It is also necessary to display these to the user and write them to the file. This is because the user may decide to take action immediately, in which case they need to be shown a message box, as the error may go unnoticed if they don't keep checking the log file. Conversely, if the user ignores the pop-up message box to continue what they are doing, then they later discover that the error was significant, they can revert back to the log file which stored a copy of the error message.

Processing layer



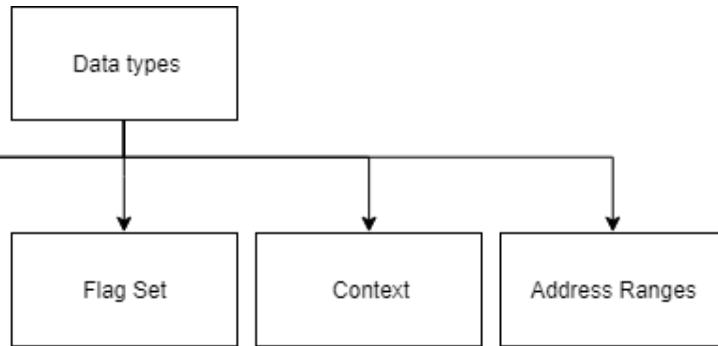
Explanation

The processing layer serves to solve four key areas, data types, opcodes, `IDecoded` objects, and the Control Unit. The control unit will be referred to as the "CU". The processing layer will be responsible for all lower layer procedures that involve assembly decoding and execution.

Justification

The processing layer is split into four different sub-problems that need to be tackled separately. This is because of the modular nature of the design. The opcodes and data types will be interchangeable. Users will be able to add their own opcodes into the solution through simple methods that aside from their own code, will only take a few modifications to the solution such as registering them to the opcode map. For this reason, the components need to be standalone such that being modified or swapped out by the user will not affect other parts of the program.

Data types



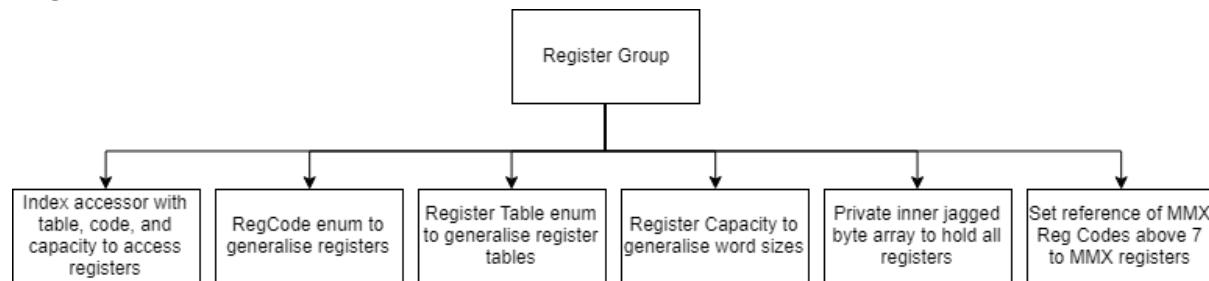
Explanation

The data types sub-problem will aim to abstract complex assembly routines and concepts that are better represented through structures in order to simplify the problem and allow for less code elsewhere as the complex procedures surrounding the particular concept are handled by a general solution in the data structure

Justification

It is necessary to divide the data types into five different types. This is because all the types solve a very different problem. This is the best approach as opposed to having many data types because it allows for structures to be reused around the code, such that the computational methods used to solve a particular problem inside the structure can be used to solve multiple related problems. This can also be seen as an instance of the Liskov substitution principle success criterion. Each data type structure will specify the minimum requirements for a class of its purpose. This means that in the future, new code can be developed designed to replace the existing data types without having to remove them completely, or recode the solution to work around the new change. This is because inheritance can be used, then methods overridden to provide the new functionality.

RegisterGroup



Explanation

RegisterGroup solves the problem of register implementation throughout the program. It allows for consistency and modularity, such that each method will generalise how registers are to be managed by the CU, but also could be adapted to fit future needs such that changes would be seen program-wide.

Justification

A RegisterGroup will be important to meet the requirements of the Single Responsibility Principle success criterion. This will be met as instead of other classes having to manage registers on their own; to have multiple purposes, a more robust solution will be to have a separate class to handle all the necessary procedures. This will benefit the solution because by generalising how registers are used, complex assembly operations can be abstracted by using a set of enums to describe behaviour. This will simplify the procedure of identifying which register is to be set in an operation without having to maintain and develop code for each individual caller. This will save development time as the procedure may evolve as development continues; a more efficient method may be found, which would mean otherwise that every class that uses registers would have to be modified rather than only the RegisterGroup.

RegCode enum to generalise registers:

An enum will be used to distinguish registers, such that a code will be assigned to each. It is necessary to do this as it will greatly benefit other parts of the program as you will see later, for example where registers are encoded in ModRM bytes as a code from 0-8. This code for example would be 0 for the A register and 1 for C the register. This will be covered further later. More reasons will be shown throughout this section.

Register Table enum to generalise register tables:

A register table is a x86-64 concept for changing the meaning of a register code. In theory, a register code is useless without a register table along with it. For example, the A register has code 0 on the General Purpose(GP) table, but the MM0 register has code 0 on the MMX table. It will be necessary to do this to contain the problems associated at

processing layer, as because of the pipeline principle, all these extra details will be abstracted before passed to the intermediary layer.

Register Capacity to generalise word sizes:

The Register Capacity enum will generalise word sizes. Word sizes can be one of BYTE, WORD, DWORD, and QWORD, meaning 1,2,4,8 respectively (make a table?). This will provide a degree of convention, as only these sizes of operands are ever operated on in x86-64; you cannot set or fetch an arbitrary number of bytes of a register. The RegisterCapacity enum will be used across the program, as word sizes make often occurrence throughout assembly; integers generally have a size of 1,2,4, or 8 bytes. It will also aid in disassembly, as a Register Capacity tied to the other attributes of a register will enable the possibility of full disassembly. With only a Reg Code and Register Table, the only property that could be determined would be the register, such as the A register or C register. With only a Register Capacity, only the size of the register could be determined, such as E-X or R-X. Having both together allows the necessary disassembly that can take place to get a full mnemonic such as EAX, allowing all the technical details that have been mentioned already to be abstracted before moving to the next layer. Another benefit being that they abstract the need for extra register tables. Technically, each capacity of register is on a different table. This is confusing enough of a detail as-is with MMX registers, therefore in order to assist with future maintenance, this will be abstracted entirely in favour of accessing by Register Capacity.

Private inner jagged byte array to hold all registers

RegisterGroup will provide a wrapper of useful methods around a jagged byte array that the caller cannot see. This will allow for an efficient storing mechanism as the value of a register can be accessed through its Reg Code and Register Table. As Register Tables have no code specifically, it is implied by an opcode, I will assign a meaningful value to the enum entries, of 0 for GP, 16 for MMX, and 32 for SSE. This will allow the register table to be accessed internally by an offset RegCode + Register Table, therefore can all be stored in one array. This method needs to be hidden externally because the design of the class needs to be abstracted from the class before it reaches the caller, such that a new method of storing the registers can be adopted in the future without affecting any callers as the callers depend on methods not the array itself. This will be done through the index accessor.

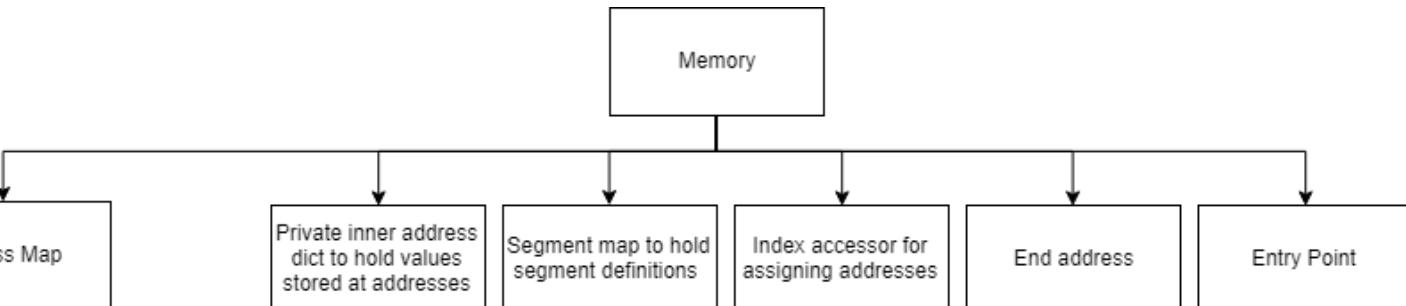
Index accessor

Having an index accessor makes for a simple way to access registers. As mentioned earlier, this will require a Reg Code, and a Register Table, but also a Register Capacity. This will provide some degree of convention, ensuring the caller can only set an array of bytes which is equal to a word size. In the end, registers need to be accessible in some shape or form to provide the core functionality of the class, doing this through either a method or an index accessor would be appropriate, however I believe an index accessor is just slightly neater.

Set reference of MMX Reg code

As there are only 8 MMX registers(MM0-MM7), attempting to access with a greater register code in x86-64 will return the MMX register of that register code minus 8. Essentially, the registers loop back round. This behaviour can be emulated by assigning the same reference to the first and second register codes that refer to that register. This is necessary because although it might seem strange; it is still documented behaviour therefore would be against the success criteria if I did not aim to emulate it.

Memory space



Explanation

The MemorySpace will provide an abstraction of RAM during emulation. Callers will be able to access byte values at specific addresses and modify them to their needs. They will also be able to access an address map, which can be

used to identify which addresses have been assigned or used, which then can narrow down a search to smaller subsets of memory.

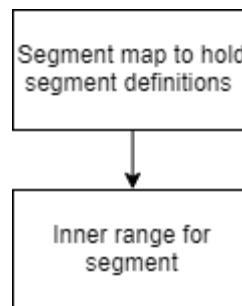
Justification

The MemorySpace class will be necessary to meet the requirements of the Dependency inversion principle criterion. Instead of callers working on a dictionary(under these circumstances a low level procedure), they will work on an abstraction, MemorySpace. This will benefit the program because useful methods to complete reoccurring problems will be provided, such as overwriting a region of memory. This will also give opportunity to handle specific procedures that need to be performed every time, such as adding the address to the address map where necessary. This will reduce dependence on the caller and changes to the procedures will be seen in every instance of usage rather than having to modify a procedure for every caller.. This is necessary because memory will need to be loaded from multiple different sources, and innately will take up a large portion of RAM, so countermeasures need to be taken where possible. Many techniques will be discussed on how to provide a better degree of abstraction. In this case I consider better abstraction to be less memory stored(hence less details on top of memory itself are being stored), however public methods are a given. The following details cannot be directly abstracted

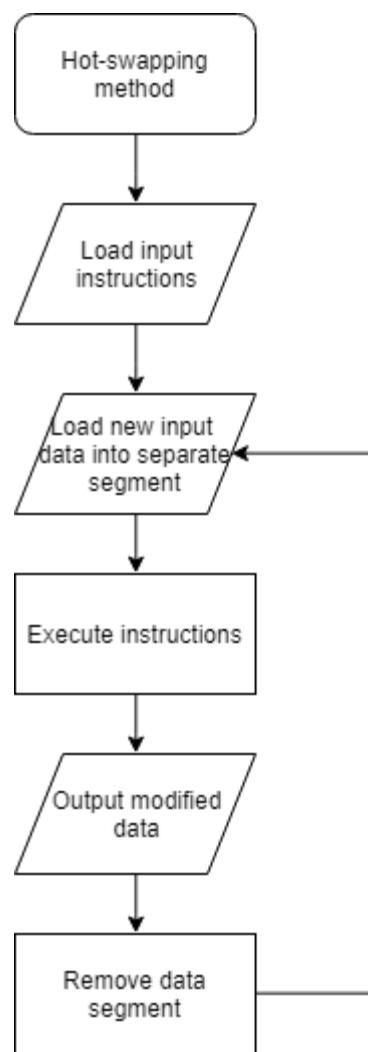
- Addresses
- Data

However, approaches to abstracting unnecessary instances of this data will be discussed in the following sections

Segmentation



A segmentation approach will be taken to loading bytes into the MemorySpace(Not to be confused with segmentation of memory at an assembly level). This includes loading memory as a given byte array, or “segment”, then allocating it a place in the entire memory map, such that the programmer does not have to concern themselves with deciding where the segment should go in the memory. There will be an option for hard coding an address, as this will be necessary to separate the stack from the main segment so they do not overlap easily. This will also allow segments to be dynamically loaded.



For example, if a series of instructions were used to operate on data, the data could go in a separate segment, then a module could be used to swap out the results, then swap in more data. This will be useful because including data in assembly programs is generally a pain. The method to do this with the least effort would be to use NASM specific “declare byte” syntax to tell the assembler to include the data in the output object code. This greatly narrows down the number of options the user has available to assemble their code. However, a module could easily be implemented that allows this hot swapping of data by loading and unloading input data segments, whilst the assembled instructions remain static in memory. In implementation, a segment will be an address range associated with an array of bytes, where the address range will represent the position of the segment in memory. Some validation will have to be done to make sure that the lengths add up. A simple criterion could be,

DEF ValidateRange(AddressRange Range)

```

IF (Range.End - Range.Start) < Data.Length THEN
    // Invalid range
FI;
FED;

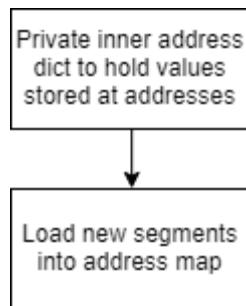
```

More flexibility could be allowed for ranges that are greater than the given data, as it is more acceptable to extend data with zeros than crop the end of data off, and another method will be mentioned in the next section that negates any impact of these extra zeroes. Using this address range will allow the benefits of the address range concept mentioned in the previous section to be applied to MemorySpace.

Segments will be given string names that allow other classes and modules to interact with this feature. In reality, the segments will be part of a dictionary, where said strings are the keys that return the segments as values. Taking this approach as opposed to a hard-coded approach allows segments to be loaded at runtime rather than at compile time, such that as many as there are no limitations to how many the user can add. If there was a predefined number of segments, this would only limit the freedom of the user. When creating the MemorySpace initially, some segments could be added by implication. For example, the input memory used to create the memory space would be implied as entry instructions. Another implied segment could be the stack space, which could be initialised to have no data, but at a specific position such that the stack is always at the same address for every program. This would be a quality of life feature for the user as they would never have to worry about the stack overlapping with their code, provided the stack is not used excessively, as they are still present on the same address map and would have potential to overlap no matter the distance between the two segments, however the stack conventionally should not be used for such large operations, so is not a behaviour that should be encouraged by the program. This allocation is done by modern kernels when executed outside of the virtualised environment but is not specific to any kind of x86-64 specification, rather kernel specific, but still gives the accurate message about how the two will be separated when they execute their programs in production. In summary, segments are an important feature,

- To improve the extensibility offered to modular components
- To simplify removing large sections of memory from the address space
- To access specific areas of memory dynamically at run time

Inner dictionary



As the primary method of storing data, a dictionary will be used. This is the most effective way of storing the memory because it uses dynamic memory allocation behind the scenes. Consider that a x86-64 program can take up a maximum of 4 GB ram, thus so could a user loaded program. If an array was used, this capacity would have to be initialised from beginning, then addresses referenced by index. As there will be many parts of the program throughout the memory space, this would be unavoidable. For example consider the stack, which typically starts at 0x7fffffff (this is very platform dependent, but in this ballpark). This is already an array of length 2^{32} . Therefore, arrays are a no-go because too much memory is consumed for unused addresses. (Note that all enumerable data structures in .NET are based on arrays, but managed cleverly by the framework, so any potential solution using arrays would be practically identical to the high level enumerable). The second potential solution would be to use a list, which would solve the memory usage problem. However, it would become very hard to locate addresses. Say address 0x10 was set a value through List.Add(), then 0x9 was added afterwards through List.Add(), how would it be possible to then change the value of 0x10? This introduces the same problems that arrays had. However, a method which I have found to have no such drawbacks is the dictionary. An address can be assigned as a key, then a byte as its value. This allows the address to then be accessed through the index accessor of the dictionary. It also brings the same advantage as the list, where addresses can be added as they are first assigned by the program, therefore saving memory. However, a further abstraction can be introduced to decrease memory usage, which is to assign a default

returned value. For example, if the most common byte used can be identified, it can be hard coded to be returned as the default value for an address rather than ever having to store it, and also addresses that are assigned the value can be removed from the dictionary, even freeing memory, as they would be returned as the default value regardless. I have decided to use zero(0x00) as this default value, as it makes the most sense as unassigned addresses are zero by default in memory(on a hardware level with paging in scope, this is not the case, but this is beyond the scope of assembly). Choosing zero for this will also allow the idea of a full address space to be simulated more realistically, accessing arbitrary unassigned addresses would never use more memory but still return the expected value. This concludes that dictionaries are the most suitable choice of data structure to hold the addresses.

Another problem that needs to be addressed by the inner dictionary is how data will be loaded into the memory space. This is where segmentation will be taken advantage of. There will be a procedure called AddSegment to do the following,

```
PUBLIC Dictionary<AddressRange, byte> InnerDict;
DEF AddSegment(segmentName, segmentObject)
    // Add the segment to the map mentioned in the previous section
    SegmentMap.Add(segmentName, segmentObject);

    // Add the range to the address map
    AddressMap.Add(segmentObject.Range);

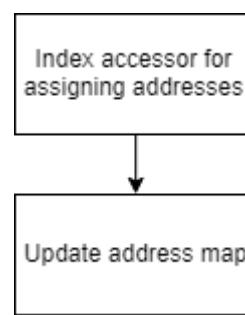
    // Set the data in the inner dict to the data in the segment
    INT Cursor = 0;

    // Start at zero and finish at one less than the end of the range because
    // the lower bound is inclusive but the upper exclusive
    WHILE segmentObject.Range.Start + Cursor < segmentObject.Range.End
        // Set the effective address in memory to the corresponding
        // value in the array.
        InnerDict.Set(segmentObject.Range.Start + Cursor, segmentObject.Data[Cursor]);
    ELIHW;

FED;
```

This will be a useful procedure that will ensure no steps are missed when adding segments and their data to the inner dict. If for example, the segment was not added to the map, it would never again be accessible after being added. By having a method specifically to do this, it is certain that all the necessary steps will be taken before continuing execution at the caller.

Index accessor



For external classes and modules to interact with the addresses in memory, there will be an index accessor that allows the memory space to be used as if it was the massive array it mimics. For example,

```
DEF AssignMemory()
    MemorySpace ms = new MemorySpace(instructions);
    ms[0x100] = 0x40;
FED;
```

This would assign the byte at address 0x100(in hex) to the value 0x40. This will provide a simple way for modules to access and modify the MemorySpace, abstracting the need for them to know the inner workings of the segments and address ranges. Naturally, they will need to know about segments if they want to make use of them, but it is an unnecessary detail in this case because the segment does not directly affect the data present in its range. In the index accessor, the implementation of the “default value” concept mentioned in the previous section will be implemented. This would take a very specific C# syntax, so the following is in a heavily C# orientated form of pseudocode.

```
public Dictionary<AddressRange, byte> InnerDict;
// Index accessor that takes a ulong in the index and a byte as the input
public byte this[ulong address]
{
    // When the index accessor is used as a variable where its value is used, not assigned
    get {
        // If the dictionary contains the address already return the value
        // Otherwise assume the default value, 0.
        if (InnerDict.ContainsKey(address))
        {
            return InnerDict[address];
        }
        else
        {
            Return 0;
        }
    }

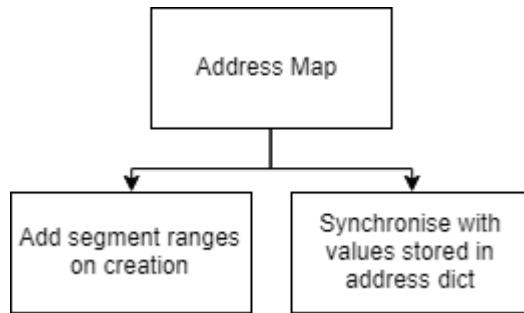
    set
    {
        /* Note that “value” is the value used to assign the index */
        // If the value is zero, it isn't going to be set
        if(value == 0)
        {
            // The address set to zero can be removed because 0
            // is the default implied value, so the result would be the
            // same except removing would save memory.
            if(InnerDict.ContainsKey(address))
            {
                InnerDict.Remove(address);
            }
            else
            {
                // Do nothing because an address that hasn't
                // already been assigned a value is being set to
                // the default value, which means that this
                // assignment can be ignored.
            }
        }
        else
        {
            // Add the meaningfully assigned value to the address
            // map. Remember that the end is an exclusive bound,
            // so the extra one needs to be added to include address
            AddressMap.Add(new AddressRange(Start: address, End: address+1);

            // Set the key in the dictionary to the input
            InnerDict.Set(address, value);
        }
    }
}
```

}

}

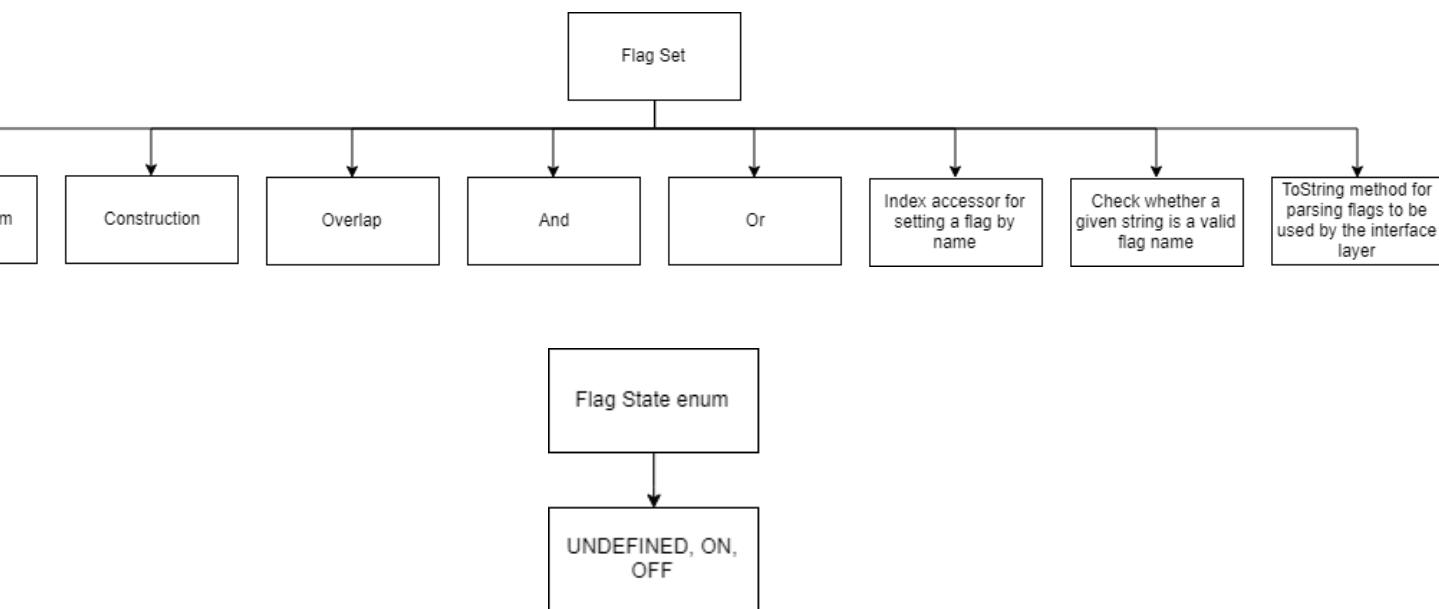
This is the basic concept that will be used to implement this, there may be further improvements and optimisations found at the time of coding. As shown in the previous pseudocode, this index accessor is where the address map will be updated.



The common problem of adding segment ranges was solved in the AddSegment() procedure, so this leaves the latter to the index accessor. This was outlined in the pseudocode, but the new range is created for the new address, such that the address map can be kept coherent. The importance of this has already been shown in the address map section. In summary, the MemorySpace class is important,

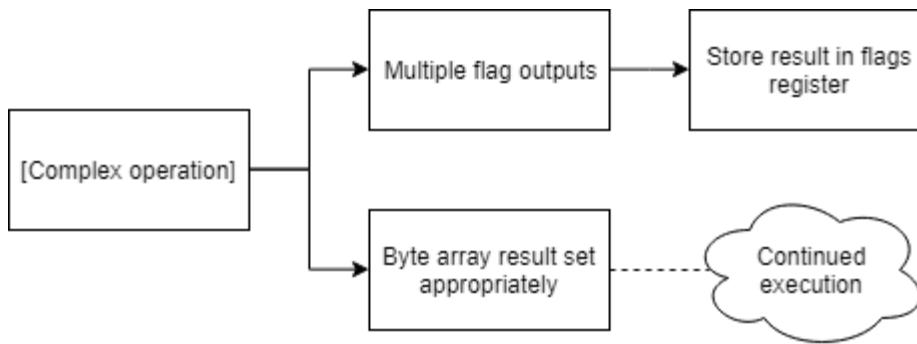
- To heavily optimise and enable the possibility of memory usage and assignment by user programs(Dictionary and Zeros trick)
- To provide an interface for memory allocation and assignment to future modules(Index accessor)
- To abstract the needs of specific routines and procedure orders from callers in order to simply maintenance and future development, reducing code and logical error possibilities elsewhere in the program.

FlagSet



Explanation

The FlagState and FlagSet will serve as abstractions for the storage for flags, providing a useful programming interface throughout the solution to simplify potentially complex procedures. The FlagSet will contain a variable for each flag of type FlagState, which can then be accessed programmatically. The FlagState will have an ON, OFF, or UNDEFINED state. The need for ON and OFF is clear, as a flag can either be on or off at an assembly level, however due to the nature of how the struct will be used, there will be a need for an UNDEFINED state. This will tell the struct to ignore this flag in certain operations that will be explained shortly. FlagSet will provide the method to modify flags in the CU, such that a new FlagSet is constructed every time rather than setting individual boolean-esque properties of a static FlagSet. This can greatly simplify the process for a routine that wants to set multiple flags at once, see the following flow diagram,

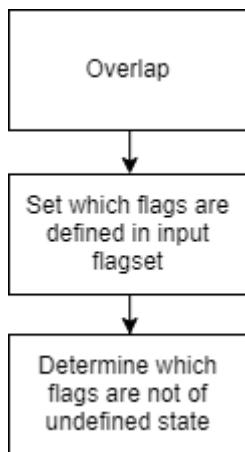


As you can see, a complex operation will require multiple flags to be set. Using an entire FlagSet to set the result is not only more elegant but also more maintainable and modular. For example, the complex operation did not need to be explained at all, the procedure matched a standard delegate given in the flow diagram of having two forms of outputs, the flags and some sort of byte array result. The byte array is another story and will be discussed thoroughly in later sections. By having the output be a FlagSet, the CU can handle everything else in its own way, handling exactly how the flags are set(which need not be complicated, but must be generalised for each operation). For instance, any “complex operation” that had this output format of FlagSet and byte array could be implemented very simply into an opcode or other means, and then in the future would not be dependent on the behind-the-scenes of the CU. This is where the undefined state comes into play. Not every given complex operation will determine each flag, rather only affect specific flags. To put into context, take the INC instruction. This instruction affects a few different flags, but specifically never the carry flag; it is ignored completely. This kind of behavior could never be handled without the undefined state because whether the flag was set the on or off in the FlagSet, it would one way or another change the state of the carry flag. Any intention of ignoring the carry flag would be completely lost and abstracted once reaching the CU, which purely takes the inputs and outputs. Having the undefined flag gives absolute leverage in this scenario, as a flag can default to undefined, therefore when setting flags in the CU, the default behavior will be to only modify flags that the developer has specifically coded to change through use of setting variables. This is by far the most modular implementation, as consider this alternative: To use methods of setting flags in the complex operation. This greatly contrasts with the principles of the success criteria, as if the procedure for changing flags ever changed, all these methods would have to be adapted to the new calling procedure, rather than only changing the FlagSet. It also supports the modularity and layered abstraction model, as the complex procedures are to be generalised into reusable methods in the utility library. By convention the utility library should have very little dependence on the CU. Naturally there will be scenarios where this is not possible, as the purpose of the utility library is to provide pretested commonly used routines, if there is a specific repeated procedure that uses the CU and makes sense to be part of the library, there may be no avoiding it. As for the layered abstraction, the utility library should never be specific to the CU, only to the data types provided in the program, i.e. the CU should be able to be swapped out without affecting the utility library. If the methods in the utility library had to call CU methods themselves, this would not align with the principle.

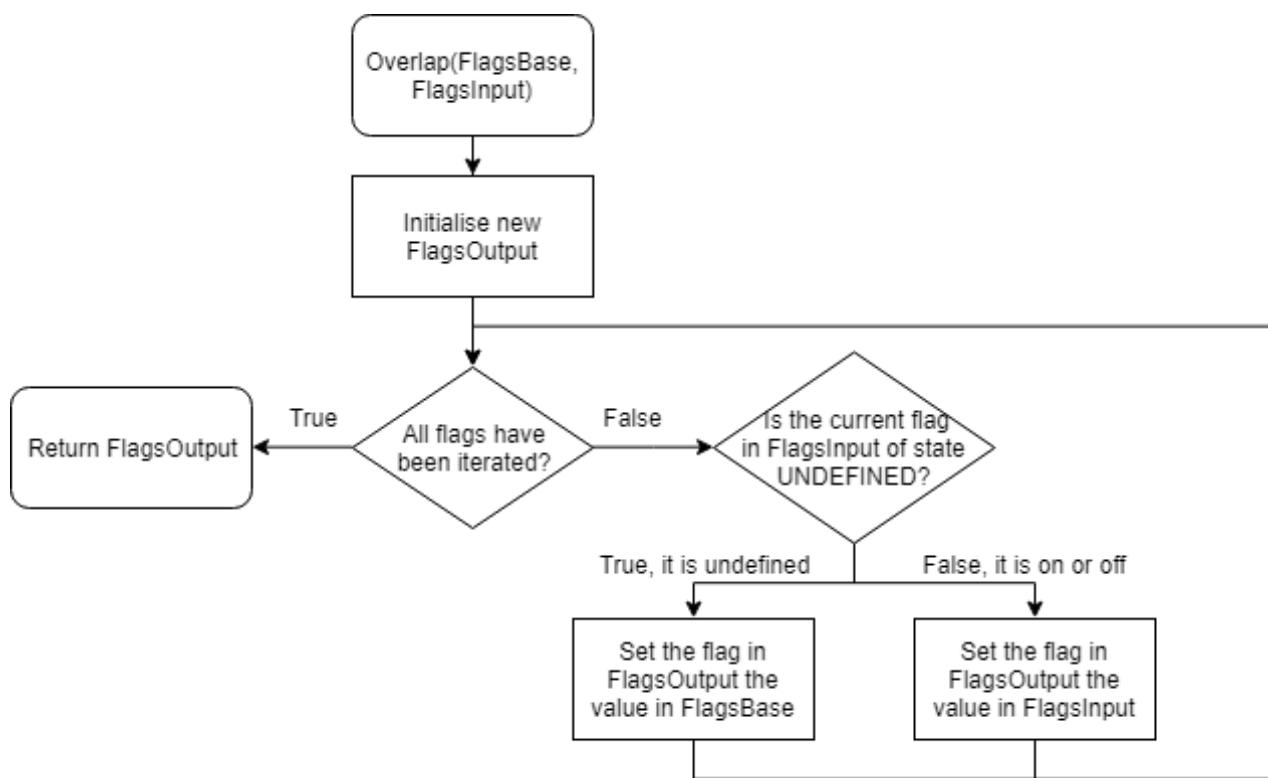
Justification

This abstraction will be necessary to meet the Dependency Inversion Principle success criterion. The FlagSet is an example of an abstraction that does not depend on details, rather the details depend on it. Every opcode will have a different reason to set a flag indicating a different result; the carry flag is not reserved only for carries. For this reason, a data structure that does not depend on the reasons for setting the flag will be a good abstraction to simplify the procedure of setting a flag.

Overlap



The overlap function of the FlagSet will outline the benefits of the undefined state best. This will provide the implementation of the concept of ignoring undefined flags and only setting defined flags. Consider the following flow diagram,



As shown in the figure, each flag that is defined(I will use defined to describe a flag that is not of state UNDEFINED), is favoured when determining the output, and that if it is not defined, the value in the base FlagSet is used. Naturally there is the possibility of this still being undefined, of which there is no better alternative than to leave it as undefined. Due to the nature of flags existing as variables rather than an iterable set in the program, it may not be necessary to have such a loop, but the flow diagram demonstrates the intention. Consider this pseudocode as a more realistic implementation,

```
DEF Overlap(FlagSet inputFlags)
    FlagSet ResultFlags = new FlagSet();
    IF inputFlags.Carry == FlagState.UNDEFINED THEN
        ResultFlags.Carry = this.Carry;
    ELSE
        ResultFlags.Carry = inputFlags.Carry;
    FI;
    IF inputFlags.Overflow == FlagState.UNDEFINED
```

```

...
/* Where ... denotes the repeated procedure. In language-specific implementation, the ternary operator will be of use rather than
having many IF statements..
RETURN ResultFlags;
FED;

```

The specifics of this syntax will be thoroughly covered in source code comments and revisited in later sections. Note that a new instance of the FlagSet is being returned rather than performing the operation on a specific instance. This is because the FlagSet will be a struct rather than a class, as I personally consider it better practice to use structs where possible as they perform much better in memory than classes. This is due to the nature of structs compared to classes. .NET references will be covered in depth throughout the source to detail the differences between the two.

ToString

A ToString override method will be implemented into the struct. This will allow the interface layer to also have a general way of handling flags, as the structures themselves will abstracted from the interface layer. This method will purely be a concatenation of all present flags into a single string such as "CF" if only the carry flag is present, or "CFOF" if the overflow flag and carry flag are present. This simple representation is all that is required for the interface layer as to test for the presence of a flag, a method such as string.Contains() can be used. I favour this over testing by index accessor methods as the position of each flag in the string will only be by the nature of how they are concatenated. For example consider this pseudo code,

```

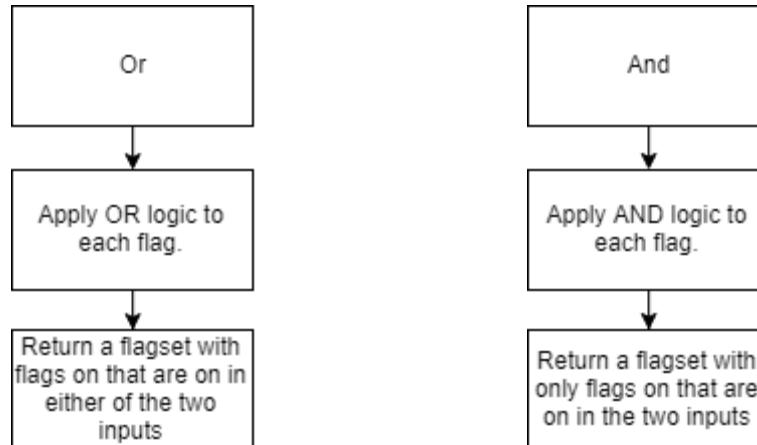
DEF ToString()
    string Output = "";
    IF this.Carry == FlagState.ON THEN
        Output += "CF";
    FI ;
    IF this.Overflow == FlagState.ON THEN
        Output += "OF";
    FI;
...
    RETURN Output;

```

FED;

If the carry flag is on, it will always be appended first, but if the carry flag is off and the overflow is on, the first two characters will be "OF" not CF, moreover the second and third indexes will not be "OF", they will be blank or another flag, therefore not a good way to determine whether a flag is set. String.Contains() does not care about order like this so using String.Contains("OF") would be much better practice. The interface will never have a need to modify or make use of the FlagSet struct, that should all be done in the lower layers, so it could be said that passing a string does enforce a degree of convention.

Bitwise operators



OR and AND operators will be available for the FlagSet. This will provide a much greater degree of control and flexibility over the structure. It is not a structure used to convey convention, it is just a tidy delegate input type for passing flags information to the CU, and so is free to provide as many useful operations as deemed necessary. This may mean that more methods may be added at development time because of unforeseen circumstances. Despite

this, I can see one specific need for the AND operator. If you were to compare two FlagSets, and AND operator would definitely be a useful tool in doing so. For example, it would give the base for comparison of FlagSets, which will be very useful when it comes to the TestHandler. Performing an AND along with ToString() will be an easy way to display which flags were found that were tested for. Ignore the example of TestHandler for now as it will be explained in depth in the TestHandler section, however consider the case that you wanted to compare two flag sets and output only the flags that were compared against in the input. For example, I need to test whether the OF, SF, and CF are set, and output it as a string. If I take the current FlagSet in the CU and call ToString on that, the output will include other flags such as the Parity Flag(PF) appended, which really should have been abstracted from the output because I am only concerned about the three aforementioned flags. So, this can be achieved using AND. Consider the following pseudocode,

```
DEF CompareFlags(FlagSet inputFlags)
    FlagSet ComparisonFlagSet = new FlagSet()
    {
        Carry = FlagState.ON
        Sign = FlagState.ON
        Overflow = FlagState.ON
    }

    // This FlagSet contain only the flags that were on in the ComparisonFlagSet and
    // inputFlags. If this result is equal to ComparisonFlagSet, it will mean that the
    // CF, SF, and OF were on in the inputs well as in the ComparisonFlagSet.
    FlagSet ComparisonResult = AND(ComparisonFlagSet, inputFlags);

    IF ComparisonResult == ComparisonFlagSet THEN
        Print("SF, CF and OF were on");
    ELSE

        // Only ON flags are included in the ToString(). Because of the AND
        // including the constant ComparisonFlagSet, it will only ever be able to
        // have the Carry, Sign, or Overflow flags set on.
        Print("Only, " + ComparisonResult.ToString() + " were set");
    FI;
FED;
```

As you can see, AND was necessary here as otherwise if any other flags were set in the input, it would have never been possible to determine equality as the two flag sets were not equal from a general perspective. This is only a single use case, but it is highly possible that it will be used more in development, and having a predefined method ready will be useful. I believe that OR will serve a similar purpose, and be useful to future developers who may carry on the project. Due to how these are very specific operations on the FlagSet, they are best placed in the FlagSet struct itself rather than in a utility library, which also gives the option of extension methods where possible, for example MyFlagSet.Overlap(x), which better portrays the nature of the methods and how certain inputs are acted upon differently to others of the same type.

Index accessor

An index accessor will provide a simpler way for certain callers to set a flag that cannot be determined at compile time. For example, when TestHandler will need to read from a file, it will not be known which flags need to be set in the FlagSet for comparison(This comparison is the same as in the previous section). To do this, a string accepting index accessor will be used. Consider the following pseudocode,

```
DEF PrintFlags(string FilePath)
    ReadFromFile(FilePath);
    String[] ReadFlagNames;
    ...
    /* Irrelevant parsing */
    ...
    WHILE $i < ReadFlags.Length
        IF CU.Flags[ReadFlagNames[i]] != FlagState.ON THEN
```

```

Print("Flag not present");
RETURN;
FI;
i++;
ELIHW;
FED;

```

Obviously this is a very stripped down version of how testing would work, but demonstrates the idea of using the user inputs as inputs to check flags, rather than having some kind of hard coded solution, such as a series of IF blocks checking if a flag is present in the ReadFlagNames array and in the CU. Such design would be tricky to maintain, as if any of the surrounding conditions were changed: the names of the flags in the files, what happens when a flag is not present-- a significant amount of code would have to be rewritten and retested as opposed to a single block.

Validation of flag names

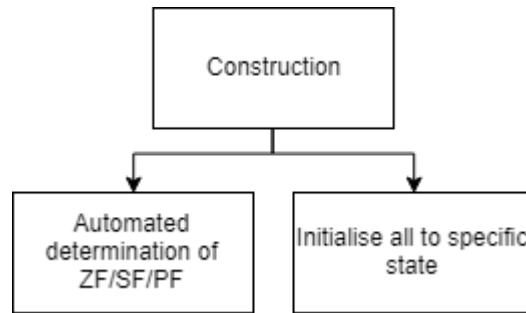
As user input is being handled, it is very important that it is validated, and in this case to test whether it is a valid flag name or not. This will be important as the pseudocode in the previous section would throw an exception if ReadFlagNames[i] was not a valid flag name. Ideally it is something that would be picked up on whilst user input is parsed, rather than when it is being processed. It is better to have freedom when processing data, it should not be at this point that I have to worry about user inputs, which will allow more efficient processing of the data as the algorithms do not have to accomodate invalid inputs. As for implementation, due to the small number of flags that exist, a simple check against constant strings will suffice. For example,

```

DEF ValidateFlag(string InputString)
IF inputString != "Carry" OR "Overflow" OR "Sign" OR ... OR ... (Etc) THEN
    RETURN FALSE;
ELSE
    RETURN TRUE;
FI;
FED;

```

Construction

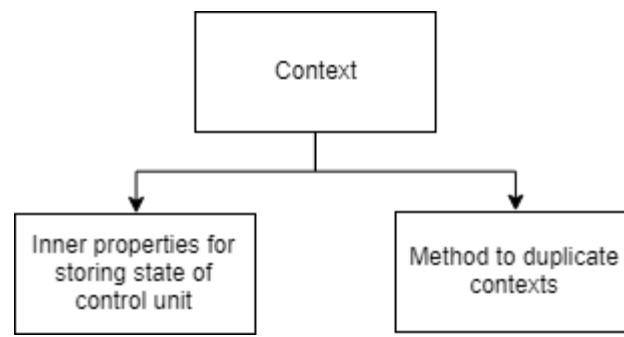


The constructor for the FlagSet will underpin some of the fundamentals of the FlagSet, but it was necessary to explain their purposes in previous sections. The constructor will mostly be the implementation of these. There will be two constructors. The first will take a FlagState as an input. This will initialise all of the flags to said state. This will be useful as in most cases, it will be left as undefined, for advantages explained previously, however there is also cause to have the flags specifically defined. Consider the Control Unit, there is no reason for the flags to ever be undefined there, they should always be defined as whether a flag is defined or not is purely a high level abstraction and not a part of the x86-64 assembly standard, hence the CU will initialise the flags as OFF instead.

The second constructor will replace a common procedure used in operations that return a FlagSet. This is to set the Sign, Zero, and Parity flags to their respective values, as they serve a more self-evident purpose than other flags, which are used at the will of the opcode. Normally these flags are either set using this procedure, or not affected at all, so will be used in most cases where these flags are affected. This constructor will also mimic some behaviour of the other as it will default all other flags to undefined. There is no reason not to do this as the flags will almost certainly be used to overlap on the CU.

These methods will be part of the utility library, and so implementation will be discussed in a later section, however it is important to note the meaning of each flag set in this case. If the ZF is on, the input was zero. If the SF is set, the input was a Two's complement signed negative. If the PF was set, the lower byte of the input had an even number of bits set.

Context



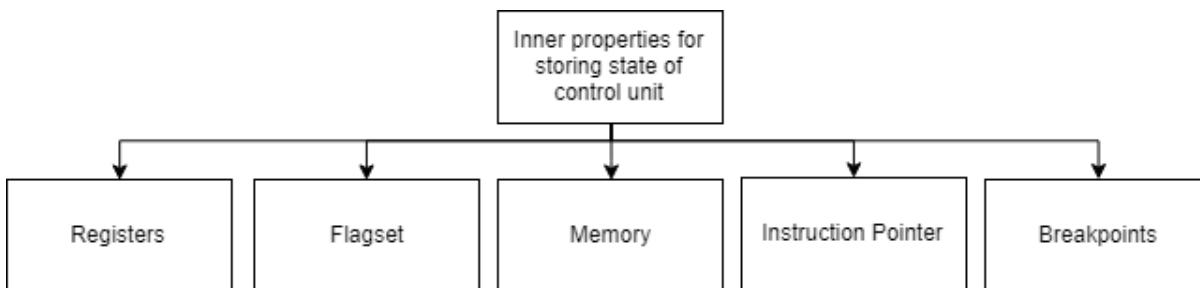
Explanation

A context class will be a container for all necessary information to set up the ControlUnit of a specific state. Consider the Context as the 4-tuple of a FlagSet, MemorySpace, RegisterGroup, InstructionPointer (F,M,R,IP). These together would provide the set of states the ControlUnit could be in, and so could be seen as the parameters to a transition function that would map the start state of the CU (most likely empty except for memory), to the end state; the end of the user program. With this definition, the transition function parameters would be the user's assembly code and the transition function itself the Control Unit, but the possible outcomes of this function are infinite as it depends on what the user inputs, therefore must be solved by computation. This is where the need for a computable solution is apparent. This is exactly what the context can provide, where the CU itself acts the transition function to map the 4-tuple to the final state. This also proves the computability of the problem. A computable problem can be formally defined as a procedure, that given a n-tuple input T, can produce the value f(T). In this case, f(T) will be the n-tuple of the input, which has been operated on throughout the execution of the user's program. The tuple will be transformed throughout the execution of the users code. For example, they may be changing the values of registers.

In some cases, f(T) may be incomputable, which would be where the user-program is invalid. In this case, the CU would never reach the final state, and so would either be stuck in an infinite loop or exit by a C# exception. This definition then supports the idea earlier of the CU being the process of performing the transition function.

Justification

The best way to implement the 4-tuple (F,M,R,IP) is simply to abstract the tuple concept and have a data structure, Context, holding all of the elements that were elements of the tuple as variables . In this way, the dependency inversion principle success criterion will be met. This is because the abstraction does not depend on the details of the caller; the Context can be used in many purposes such as a context switching procedure or sourced from user input to the program. This will allow the Context to serve as a skeleton for the CU and allow more unique procedures such as context switching; swapping out a context stored in the CU to then execute instructions of another context. This is possible because the CU would not need to depend on the details of the abstraction, rather only have the necessary inputs(the 4-tuple discussed earlier) it needs to start execution.



It is necessary that these are the four values of the tuple as they all have a direct impact on the final state of the CU. Consider the following,

The user creates an adding program. If the program started with values in registers, the end result would be different.

The memory holds the instructions, therefore it is clear that if these are changed, the series of instructions executed will be different. In effect, a different program will execute.

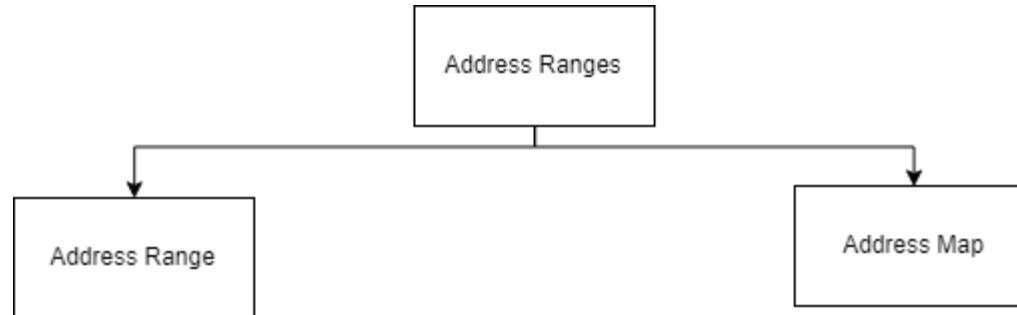
Execution begins on a selection with the carry flag set. The first JC (jump carry) is taken, hence a different series of instructions were taken to produce the result, therefore the result is different.

Instruction pointer. Another take on this would be to consider when the instruction pointer is points to the byte next to the address it should be pointing to (This will be referred to as a “derailed instruction pointer”). This could very well mean that every instruction read be different from the input instructions, as the next byte over will be interpreted as the opcode instead of the intended entry point. Since instructions are in the majority of cases more than one byte, entirely different instructions will be interpreted. If this is not clear, the decoding of instructions will be covered in a later section, but it remains obvious that if the instruction pointer is different, different instructions will be executed.

This has demonstrated the need for the Context, however, most of its functionality will be shown through other classes and modules in later sections. For now, it is important to understand that the concept of context switching will be used. This serves a very similar purpose to what a hardware processor would have to context switch but is implemented very differently. When another module attempts to use the CU, it will have to wait for the current caller to finish executing instructions, swap out that context with its own, then execute its own instructions. This allows multiple different modules to use the CU concurrently. This will be discussed further in the ControlUnit and Handle sections.

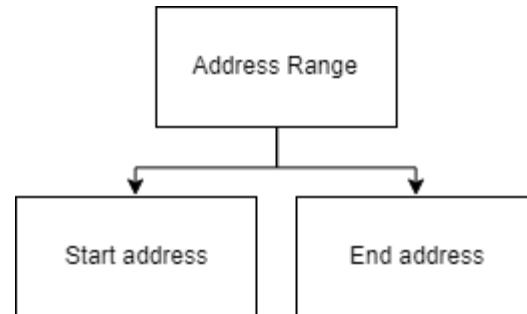
The context is the best approach to implementing the 4-tuple discussed earlier as it also supports the Liskov substitution principle as part of the design success criteria. For example, a future module “S” that is inherits from MemorySpace. As the index accessor is a property of the MemorySpace, S being a subtype means that it will also have this property, therefore can be used in place of a MemorySpace. Then to add unique functionality, the index accessor can be overridden.

Addressing structures



In this section, two data structures will be covered, the AddressRange struct and the AddressMap class.

Address Range



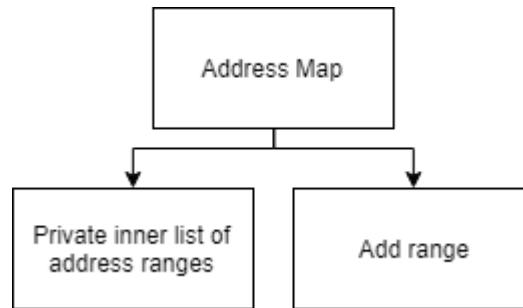
Explanation

The AddressRange struct will be a data structure with a Start and End unsigned long values. These must be unsigned long values as they represent address in a 64 bit memory space, meaning that they can be a value from 0 to 2^{64} . Its purpose will be made apparent through the AddressMap. One important thing to note is that the End value will not be inclusive. For example, a range of 0-5 will represent the set $\{0, 1, 2, 3, 4\}$. This has no technical significance, but inclusivity of ranges is a very important aspect of communication when maintaining code, therefore should be established clearly.

Justification

Having a separate structure will make for more maintainable code in the future as the Single Responsibility Principle is being met as the Start and End variables are being separated into a separate structure that allows it to be used for other purposes throughout the program, rather than only in the AddressMap. The values for Start and End would be hard to interpret if not in a separate data structure; they would otherwise have to be stored in a 2D array which would be more complex to understand and use programmatically than to store the start and end address together in a struct.

AddressMap



Explanation

The AddressMap class will be a container class for an internal list of AddressRanges, and will provide methods for interacting with it, in what will be referred to as a “coherent way”. The main principle here is to keep the internal list ordered. This is easier said than done when dealing with the abstract notion of a “range”. The following rules need to be defined,

What happens when two ranges overlap?

What happens when one range is a subset of another?

How will it be indicated that a range does not exist currently exist and needs to be added?

Aside from basic ordering principles, these behaviours need to be abstracted entirely from the caller, which is the exact purpose of the AddressMap, it is designed to do nothing greater nor lesser, hence will also require some cooperation by the caller. This is because the class is very flexible in purpose, and so will be seen constantly throughout the source.

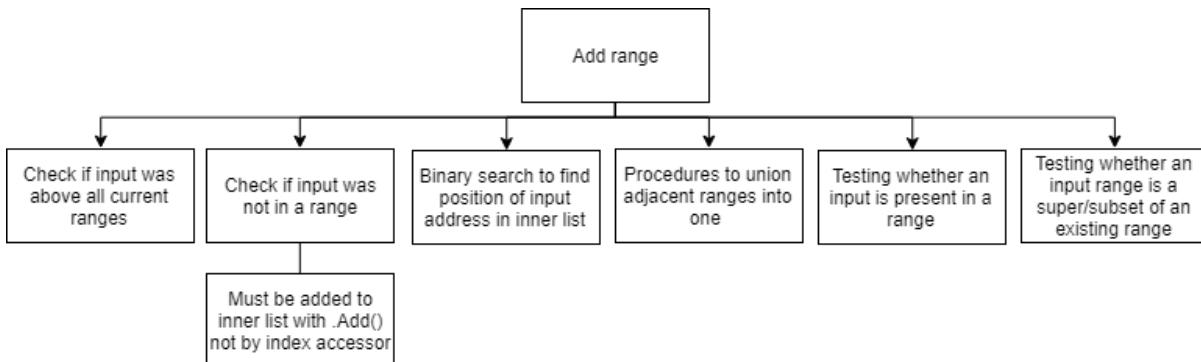
Inner list

This list will be the list of address ranges that is abstracted from the caller. They will still be able to view address ranges at indexes through use of the index accessor that will forward all indexes retrievals to the list. This must be done to enforce usage of the provided wrapper methods in the AddressMap class in order to keep the list coherent.

Justification

The AddressMap is a necessary class to provide a better method of storing address ranges than array. This can be seen as an example of the Single Responsibility Principle success criterion being met. The problem of sorting the ranges would be a problem for every usage of an array of address ranges. This procedure can be generalised and automated inside the AddressMap class instead of having to recreate the sorting algorithm and implementation for each usage of multiple address ranges, hence the purpose of the AddressMap will solely be to keep the inner address range collection in a meaningful order.

AddRange



Add range will handle all of the adding procedures and conditions for the caller, defining the rules mentioned before. Firstly, all rules will be ignored if there are no ranges in the list already. This is because there is only one possible outcome, and unnecessary processing can be skipped in this way and can return early. If this is not the case, a binary search will be performed to determine the position of the given range in the map(the inner list). I will discuss the details of the binary search shortly, but first understand the outputs that are required of this binary search, as there are extra rules in play and the concept of a range, more things must be considered than in an ordinary binary search. This output will take form as the BinarySearchResult struct and contain the following values,

INT Index;
 RESULTINFO Info;

Index will hold the value where the input placed in the inner list. This may be accessible through the index accessor, e.g. AddressMap[\$index], but it is entirely possible that it was present in no range at all, in which case needs to be inserted manually. This is one of the factors that the AddRange function aims to abstract, however in order to achieve this desired abstraction, the ResultInfo enum must be used to pass the details about the caller. In most cases, AddRange() will be the only caller that listens to this, but is entirely possible that a module in the future might not. I think this is the best approach to doing this as it keeps the binary search and AddRange() nicely separated, which will allow for simple implementation of new methods in the future that make use of the same binary search algorithm. This is extremely important because if separate, extreme care must be taken that each binary search outputs the same values every time or the inner list will never be coherent if multiple wrapper methods are used. For this reason, a single concrete binary search method with a generalised output will be used.

Result info represents the following enum,

- OUT_OF_BOUNDS represents an address that above/below all existing ranges. This is an attribute that will be useful to some callers that may need to check if this was infact the case, then do something with that. This is a useful detail that I have decided to include because I feel there may be a specific case where it will be useful and may get too far out of the abstraction model if it is not included initially.
- PRESENT represents the more general case that a result is present in an existing range. This is necessary because of interactions with the inner list. If a range is not present, then needs to be added, List.Add() must be used. Otherwise if it is not present, and it going to be updated, it can be accessed with the index accessor. Including this in the enum is the most reasonable approach to this, an additional boolean variable is not necessary when memory can be saved by using an enum.
- NONE will be used to check for a value that falls in to no category. This is also important for identifying a value that is not present, because without NONE, the default value for the enum would be OUT_OF_BOUNDS, which would obviously not be accurate. Through code I will use more demonstrative conditions such as testing for PRESENT not to be set, rather than testing for NONE. This is going to make complex code more maintainable because the meaning of NONE can be very vague and have multiple connotations in this context.

Back to AddRange(), a series of conditions will be tested based on this BinarySearchResult, that will define the behaviour of the mentioned rules.

Firstly, overlap will be tested for. This is necessary because the binary search will not work properly if ranges overlap, the list would not be truly ordered because there is not a one to one function mapping an address to a range, as there would be multiple possibilities. In order to always be in the optimal scenario, this must be done. To test for this, the following predicate will be used.

```
Result.Info==ResultInfo.PRESENT && AddressRanges[Result.Index].End<inputRange.End
```

Firstly, only present ranges are considered. This means that only situations where the start address of the input range is within a range are considered by this predicate. The condition where the end address lies above is considered afterwards. Secondly, it must be tested whether the input range is a subset of the current range at the index. If it is, then it doesn't meet the rule that was being tested for. This is dealt with in another condition. If this predicate was met, then the address range should be updated with a new range that is the union of the input range and the existing range. Due to it being known that the start of the range is present in the current range, it must be the end of the input range that lies outside of the range. Therefore, the following approach could be taken,

```
Map[index] = new AddressRange(Start:Map[index].Start, End:Input.End);
```

The next step would be to test the other case, where the input range had a start value outside of the range, but an end address in an existing range. This must be handled differently because the start address and end address shown in the previous rule will be the other way round. This is essentially the only difference, but needs to be handled separately to the other case because of this. In terms of implementation I think there is no better way, because it is unavoidable that these need to be handled differently, and feel like more complex ways of doing this, such as creating a variable to hold the range and then set that, only add unnecessary complexity to the program.

The last case would be that the address is purely a standalone range, such that this predicate is met,

```
Result.Info != ResultInfo.PRESENT
```

This relies on the previous predicates not being met, therefore will require an if else block logic to implement this. In this case, the range will purely be inserted at that index.

Finally, the last rule would be to ignore adding ranges that are subsets of an existing range. This behaviour can exist purely through exhaustion, that no other predicate being met, such that the method can return as normal because it has deduced that the range must be a subset, in which event it would not be added; nothing would happen.

Binary Search

The binary search will search for the position of a single address in the inner list. This can provide more than enough information to deduce how to act on the result. It is not necessary to search an entire range, as shown above, using the start and some extra checks afterwards will suffice. The main objectives of the binary search were already implicitly stated previously, such that the following will more detail the implementation. The result is very flexible such that there is not one specific way to interpret the results, in the future the outputs may be used to determine different conditions altogether.

Firstly, if the inner list is empty, return index 0(Remember that not being present is the default, the enum will initialise to NONE by itself).

Secondly comes an index, which will be the index used in the return value. This will be initialised the following,

```
INT index = (InnerList.Count - 1) / 2;
```

Note that 1 must be subtracted because solely indexes are being dealt with, which start at 0, as opposed to a Count which will return the number of items in the list.

Now enters the binary search itself. This will search very similarly to a binary search, hence the name, but will make use of some information that is only known in the scope of the search to provide more insight, that can be used more

effectively than if an ordinary binary search was used, as demonstrated in AddRange. Consider the following pseudo code as the binary search,

PUBLIC List<AddressRange> Map;

DEF BinarySearch()

```
INT index = (InnerList.Count - 1) / 2;
WHILE(TRUE)
    INT prev_index = index;

    // If the address is above the end of the current, it has nothing to do
    // with this index, so advance upwards in a binary search fashion.
    IF Map[index].End <= inputAddress THEN
        Index += index / 2;

    // If the address is beneath the start of the current, likewise with the
    // previous condition, except advance downwards.
    ELIF Map[index].Start > inputAddress THEN
        Index /= 2;
    FI;
    IF index >= Map.Count THEN
        // *done + out of bounds*
    FI;

    IF index == prev_index THEN
        // *done*
    FI;
ELIHW;
```

FED;

This specific design of binary search is purely based on how I code my binary searches, some prefer a “right” and “left” approach, with two index values approaching each other, however this isn’t as easy to apply in the context of ranges, nor is it how I am familiar with writing binary searches, so I will be using this design when it comes to coding the solution. Having the index in a single value also makes it very easy to then return this value afterwards, there has to be no extra steps to determine the index, it has already been incorporated into the algorithm. To aid those who are more used to the right and left concept, this method, like others, will be heavily documented.

Another thing to note about this design is the WHILE(TRUE). This is generally not a good idea, but rest assured that it is without a doubt that the loop will at some point break, I just opted to shift the conditions inside the loop because there are multiple that are going to indicate slightly different results, but are still exit conditions. This design is a lot neater and maintainable than testing for multiple conditions in the while parameters, then determining them again afterwards, a mere comment will clear up any confusion here. The last part of the binary search to consider is the additional information, this was omitted from the previous pseudocode because it was not relevant there.

DEF BinarySearch()

```
...
/* prev code */
// Due to half of the index being added on every time, there is a
// possibility that due to the rounding of integer division (or more
// specifically the quotient value being taken), it could sometimes
// go two above the count. This can just be ignored entirely by using
// more general terms such as >= rather than ==. Finally the index is
// set to the count of the list, because although it is not present,
// it is to be inserted at that index, even if it was above this, it would be
// inserted at the top regardless. This is just how List.Insert() works,
// the input is the value you want the index to be. It is necessary to
// separate this from the next loop because checks will be done using
// the value of index in the list accessor. There would be an exception
// because it would be attempting to access an index that does not
// exist.
IF index >= Map.Count THEN
```

```

    RETURN new BinarySearchResult(
        Index:AddressRanges.Count,
        Info:OUT_OF_BOUNDS);
FI;

// If the binary search has converged on a value, the division yielded 0
// change to the value and so looped round again with the same index
// and did not change after that.
IF index == prev_index THEN
    // Check if the address lies within the bounds, in this case it is present.
    IF Map[index].Start <= inputAddress < Map[index].End THEN
        RETURN new BinarySearchResult(Index:index, Info:PRESENT);
    FI;

    // The previous case for out of bounds checking cannot be
    // applied to addresses that fall into the “beneath all others”
    // category, but can be easily checked manually by checking
// if the address is less than the start address of the first element of the array.
    IF inputAddress < Map[0].Start THEN
        RETURN new BinarySearchResult(Index:0, Info:OUT_OF_BOUNDS);
    FI;

    // Finally, if not returned already, return the address as not
    // present, with the index to insert it at.
    RETURN new BinarySearchResult(Index:index);
FI;
FED;

```

Conclusion and justification of methods

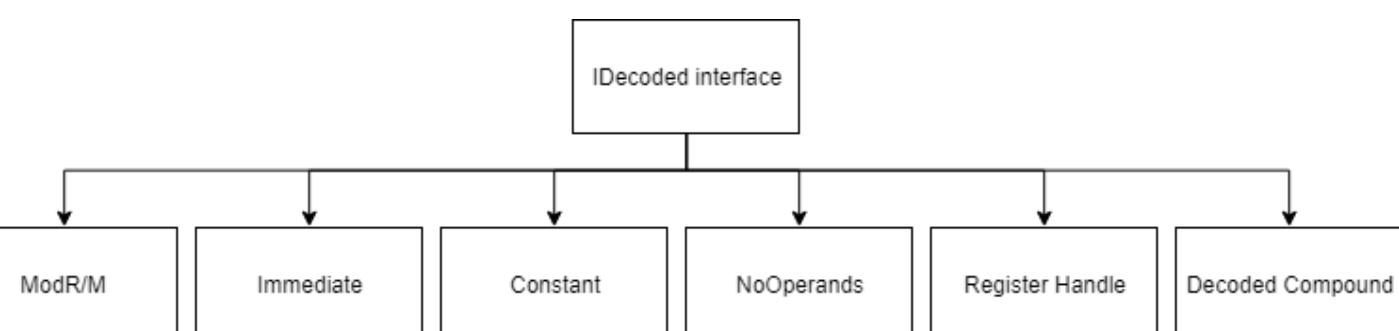
This concludes the AddressRange section. As for choice of search algorithm, I think this is the best possible and the best necessary for the program. By using this algorithm, a lot of useful information is retained, and is can be used by the caller functions, as demonstrated in Add range. It will also be very maintainable as the search algorithm has been isolated from the evaluation of inputs as shown in the pseudocode; it will be very easy for a future developer who understands a binary search to develop the code. Of course, the binary search will also be explained to a developer who does not understand a binary search. In terms of using the address range, the main thing is to understand how it works, it will be entirely modular in the way that it can be transferred from program to program, but it is to be used as more of an aid to accompany an existing collection rather than have any intrinsic value. For example, as will be seen shortly, if I have a collection of memory that could hold addresses from 0-2^64, it will prove invaluable in narrowing down searches to small and manageable address ranges by keeping track of which addresses are in use. In summary, the AddressRange and AddressMap will be important,

To greatly improve the efficiency of other collections that store a large amount of data with gaps in order to search for data quickly and keep track of where data is stored in these collections; information that is not provided by the collections themselves.

To separate certain ranges of collections from others such that only a small relevant piece of information has to be processed as opposed to the entire collections, allowing for more performant usage of these collections from external callers.

IDecoded interface

Explanation



The IDecoded interface will provide the generalisation of inputs to an opcode. The primary purpose will be to allow all opcodes to accept this input, such the need to decode the operands is abstracted from the opcode entirely such that they can accept any input that implements the interface. The next sections will cover the fields necessary for the interface to implement.

Size of the operand

Explanation

For example, the size of register fetched. Sometimes RAX, the 8 byte A register will be fetched, sometimes EAX; the 4 byte A register(which points to the same register, but only covers the lower 4 bytes of it), et al.

Justification

This is necessary because operations may depend on this size, for example to sign extend an output. If the output is sign extended to a shorter length than it should, the negative sign would not be retained because the sign bit(MSB) would not have been set. If the output was sign extended to a longer length, there may be errors in other functions that did not anticipate this input, and rightfully so.

Disassemble method

Explanation

Every operand will perform disassembly differently as they represent values stored differently. The disassembly of an immediate operand will be different to the disassembly of a ModRM. However, every operand will be required to provide some kind of disassembly in order to generalise the solution. Consider the following pseudocode method for a class that performs disassembly of instructions,

DEF DisassembleOpcode(OPCODE Opcode)

string Disassembly;

// Add the mnemonic onto the start of the string, e.g MOV

 Disassembly.Append(Operand.Mnemonic);

 // Iterate through all operands appending the disassembly of each
 // using the interface provided methods.

 FOR IDecodedClass in Opcode.Operands DO

 Disassembly.Append(" " + IDecodedClass.Disassembly);

 ROF;

 // Return the complete string

 RETURN Disassembly;

FED;

Justification

This kind of generalisation would provide a very simple solution for performing such an operation. Moreover this would increase the degree of abstraction in upper layers as they do not have to handle IDecoded classes and perform specific routines per IMyDecoded class type, rather use iteration to disassemble each in the same way. This is necessary to simplify the task of making modifications to the disassembly of the operand and the addition of new operands in the future. The methods will also only need to be tested and refined once per class; once working code has been found, there would be little need to change its outputs in the future. This is exponentially less work than having each opcode have its own disassembly procedures.

Fetch method

Explanation

The interface will implement a method field such that all classes that implement the IDecoded interface will have a fetch method that returns a list containing all of the opcodes that IDecoded operand provides. Returning a list is necessary because it will allow operands which return more than one value to function properly. For example, a ModRM byte has both a source operand and a destination operand, therefore two values must be returned. An immediate has only one operand, so does not require the list, however, would still function with one, whereas a ModRM could not function without one.

Justification

This field is necessary because every operand will have a dramatically different way of fetching their values. For example, some may have to fetch from memory, some may use registers, all very different operations. This means

that a general solution cannot be applied, they all must be solved independently. This is because of the nature of different operands, if they could all be fetched in the same way, there would be no reason to distinguish them. However, all will output one or more byte arrays, so the output of each can be generalised like this. This can allow for a general solution to be applied further down the line when the opcode fetches the values of all operands, similar to the pseudocode in the previous section.

DEF FetchOperands(OPCODE Opcode)

```
// Create a list to store the operand values
LIST<BYTE[]> FetchedValues = new LIST<BYTE>();

// Use the interface methods of the DecodedClass to fetch
// all the operand values. Note that DecodedClass.Fetch returns a list
FOR IDecodedClass in Opcode.Operands DO
    FetchedValues.AddRange(DecodedClass.Fetch());
ROF;

RETURN FetchedValues;
FED;
```

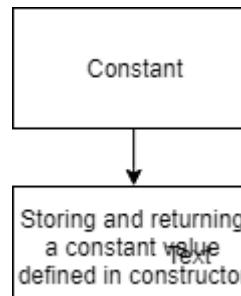
Likewise with disassembly, using this algorithm as a general method to fetch operands greatly abstracts the need for the opcode class and maintainer to understand how operands are fetched from the caller.

To set/modify the value of an operand. This will allow the opcode classes to easily interact with the operands, such as storing results in memory or in a register. This is necessary because it abstracts concepts such as the MemorySpace and RegisterGroup from the opcode classes, which would only increase code size and would be harder to maintain the repeated code across every opcode class if a bug was found or an improvement was to be made. By providing this in the IDecoded classes, as there will be much fewer than the opcode classes, there will be less code to maintain. Moreover, less unnecessary code will be written because each operand class is going to store this data slightly differently, e.g a RegisterHandle would store the result in a register. This will be covered in the next sections. In some cases, this function will not be available for an operand. For example, an immediate value cannot be set, and would have no reason to be. For this reason, the immediate class and others who may have the same behaviour, will throw exceptions in this case. This would not be at the fault of the class, but at the fault of another place in the code. There is no valid case where an opcode would receive an immediate, then attempt to set its value again, so the developer would have made a mistake in the code.

Justification

This interface will be necessary, as it links back to the Liskov substitution principle as part of the design success criteria. Doing so will improve development and testing, as less needs to be developed and less needs to be tested if a common general solution is provided as opposed to relying on every single opcode class to parse inputs by itself. Another design success criteria that is met by this interface is the Interface segregation principle, hence will also make it easier to improve the operand classes in the future as the callers will only be dependent on the output data they provide through the interface methods, not the actual implementation of the operand. (However it will still be necessary that the updated operand class is coded correctly). The actual implementation of these features will be present in other classes (hence other sections) as it is an interface, not a class, so can only provide the required properties and methods that every derived class has, but the necessity for this interface has now been made apparent.

Constant



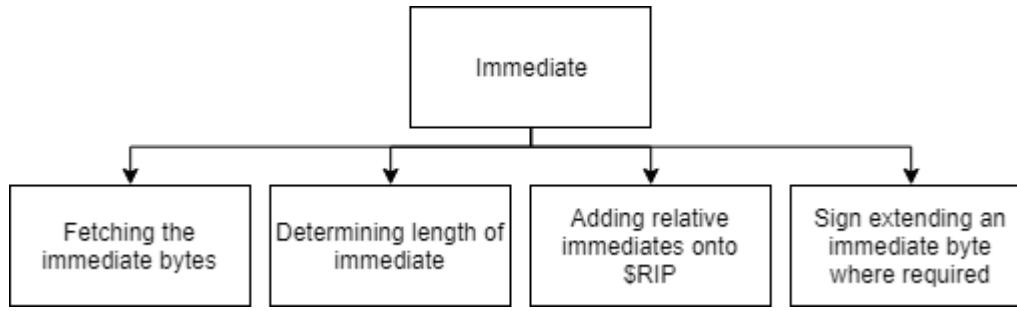
Explanation

The constant operand will hold constant values as operands. Their value will be hardcoded, not defined at run time. The disassembly method will return the constant value in hex. The fetch method will return the constant value, and the set method will throw an exception.

Justification

It is necessary that the task is solved separately because certain opcodes require a constant value as an operand. This will simplify the creation of these opcodes because they do not have to hard code methods themselves that define the constant whilst being able to accept other IDecoded types as an input as well. The best response to the value being set is to throw an exception. This is because the value is defined as a constant value, the value would be discarded after the opcode has finished executing even if possible, so it is clear that the developer has made a logical error in their code; throwing an exception will help them address this.

Immediate value



Explanation

An immediate is a value that is stored in the instructions next to the opcode at the time of assembly. For example if I had the instruction ,

ADD EAX, 0x10

The 0x10 would be an immediate stored next to the opcode. These are comparable to constant values in high level languages, which have values known at compile time. They also come in the same sizes as registers, such that the RegisterCapacity enum discussed in the RegisterGroup can be used to generalise immediate sizes, which further allows a more general solution across the program as only one enum for data sizes is required, which will make the program easier to maintain in the future because less data types have to be managed.

Justification

The immediate class will be necessary to implement this concept because there is potential for error in the order of instructions when considering immediates. For example, if the immediate bytes are fetched before another operand is(immediates can be combined with other operands in certain cases), the instruction would be interpreted incorrectly and likely result in a derailed pointer. More specifically, the last byte of the immediate would be interpreted as the next instruction. Another justification would be that immediates follow a very specific set of rules that would give a good opportunity to abstract these details from the caller. These rules are not an important detail to the caller because they only determine certain properties of the immediate. The rules will be covered in the following sections.

Determining the length of the immediate

Explanation

The Size property of the interface will be used to store the size. When the caller specifies a register capacity to use as the word size, it will have to be validated to make sure it abides by the implied immediate settings. These are all tied to opcodes, so will have to be provided by the caller. By default, Immediates have a maximum size of a DWORD. I.e. if no other rule is met, the immediate is a DWORD.

Justification

It is necessary to solve this problem as a separate task because performing the validation inside the class ensures that it is always performed and abstracts the need of the caller to know how immediate values are validated. As this is a common operation, it is best that the immediate sub-problem is modularised, such that the common solution and rule set can be applied in any context, without having to write new algorithms.

Fetching immediates

Explanation

The settings implied by an opcode change the behaviour of the immediate. This will define how the immediate is fetched, so conditions are required to determine how to handle each setting. The settings can be one of the following,

- Sign extended byte. Certain opcodes will take a single byte as an immediate operand, which is then sign extended to match the size of the other operand. Some instructions have this opcode, most do not. Having this ability will allow the sign extension procedure (covered shortly) to be performed before the opcode class receives the operand, and so would not even know it happens. This is a necessary abstraction as it links back to the single responsibility principle mentioned in the success criteria, the opcode classes themselves will only need to worry about the data in the operand, not how the operand is interpreted, which is the responsibility of the immediate class.
- 8 Byte immediates. In the entire x86-64 standard, only a single opcode of MOV can be used to with an 8 byte immediate as an operand. However, there may well be the case that the user wants to allow other operands to use 8 byte immediates. In this case, the best design would be to provide the setting in the immediate class that allows an 8 byte immediate. This is because it keeps code general, the MOV class doesn't have to have a specific exception for the specific opcode that accepts this. This design is also supported by the Open/Closed principle from the design success criteria as it provides the ability to extend functionality of a the new opcode (allowing it to have a 8 byte immediate) without changing the functionality of the any other classes (MOV or the immediate class). As a result of this, enabling the 8 byte immediate will not affect any opcodes, which may not be the case if it was required to modify the immediate class. For example, if a list was hard coded inside of the immediate class that dictated which opcodes could have 8 byte immediates, this would mean that the developer would have to modify the immediate class to add the new opcode, which introduces the possibility of breaking the immediate class and affecting the other opcodes that depend on it.

To implement these rules, the following selection procedure will be used,

```
00 PUBLIC BOOL IsSignExtendedByte;
01 PUBLIC BOOL Allow8ByteImmediate;
02 PRIVATE BYTE[] FetchedBytes = null;
03 DEF GetImmediate(REGISTERCAPACITY Size)
04
05     // Check if immediate has already been fetched
06     IF FetchedBytes != null THEN
07         RETURN FetchedBytes
08     FI;
09
10    BYTE MaximumSize;
11    IF IsSignExtendedByte == true THEN
12        MaximumSize = 1;
13    ELIF Allow8ByteImmediate == true THEN
14        MaximumSize = 8;
15    ELSE
16        // Default to 4 bytes
17        MaximumSize = 4;
18    FI;
19    // If the size is within the inclusive maximum value, that number of bytes can be
20    // fetched.
21    IF Size <= MaximumSize THEN
22        FetchedBytes = CU.FetchNext(Size);
23    ELSE
24        //See following
25        FetchedBytes = CU.FetchNext(MaximumSize);
26        FetchedBytes.SignExtend(ToLength: Size);
27    FI;
28    RETURN FetchedBytes;
29 FED;
```

This pseudocode algorithm will provide a complete solution to this subproblem because all of the tasks required to be performed by the fetching algorithm have been met

- Storing immediate bytes and returning the same every fetch

This task is important as it prevents immediates from being fetched multiple times, such that the instruction pointer is incremented multiple times, which will lead to the next instruction being interpreted as the immediate. To prevent this, the bytes are stored in the private variable FetchedBytes. This means that the value stored will remain after the function returns, where as a local variable would be erased. On line 6, a condition is tested to see if the FetchedBytes array is still null(the value it is assigned to). This will return the previously fetched bytes if the immediate has already been fetched, such that the immediate will only be fetched once, the first time the method is called.

- Checking to only allow QWORD immediates where applicable

This is done by checking if the boolean "Allow8ByteImmediate" is true on line 13. This boolean will be set in the constructor, or by other means irrelevant to the algorithm. The MaximumSize will be set to 8 which will mean that the condition on line 24 will now be true if the input size is a QWORD.

- Sign extending to DWORD when cropped down from a QWORD

This is task is performed on line 25 and 26. Instead of fetching the specified size, the maximum size possible will be fetched. The fetched value is then sign extended to the desired length. This will provide consistency as the caller will expect the size they specified to be returned as they may have cause to iterate it later. Instead, the value will be sign extended to the desired size, such that the value is not changed.

When the caller specified size is greater than the maximum size, as discussed earlier, must be cut to the maximum size, purely because of the x86-64 specification. Here is the quote suggesting so,

"In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use. "

This is done by fetching the maximum size instead, then sign extending it to the expected size. This is purely because the x86-64 says so, and the assembly section of the success criteria includes "1:1 emulation of a x86-64 processor", so this is not a detail that can be abstracted. However, it can also be justified as necessary in the program. If it were not to be sign extended, the two operands would have different lengths. This would be a devastating inconsistency, because the two arrays would no longer be able to be operated on in parallel. This means iterating through one and being able to use the same iterator for both. Consider the following,

```
DEF ParallelCopyExample(BYTE[] InputOne, BYTE[] InputTwo, REGCAP Size)
    INT i = 0;
    WHILE i < SIZE
        InputTwo[i] = InputOne[i];
        i++;
    ELIHW;
    RETURN InputTwo;
FED;
```

In this example, it could quite easily be worked around, however when more complex algorithms are discussed later, this would be extremely detrimental. If the length of both arrays was not the same as Size, an ArrayOutOfBoundsException would be thrown. Working with arrays like this has great advantages in the bitwise operators in the utility library.

Relative immediates

Pseudocode

```
BOOL IsRelative;
DEF CheckRelative(BYTE[] Input)
```

```
// Check if the immediate is a relative immediate
IF IsRelative == true THEN
    // Sign extend to match the size of the instruction pointer
    Input = Bitwise.SignExtend(Input, 8);
```

```

// Use the bitwise library to add the two
RETURN Bitwise.Add(Input, CU.InstructionPointer);
ELSE
    // Return to procedure as normal if not a relative immediate
    RETURN Input
FI;
FED;

```

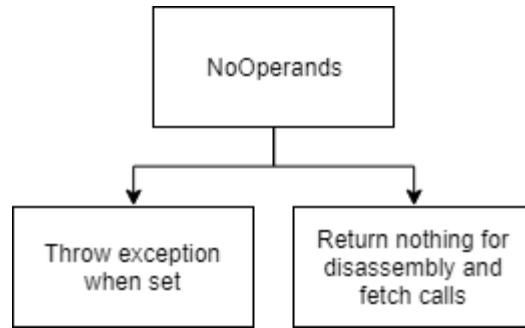
Explanation

Certain immediates can be implied as “Relative immediates”. This is where the value of the instruction pointer is added onto the immediate. To do this, existing library functions can be reused. Bitwise.Add() and Bitwise.SignExtend() are used to perform the addition. Input must be sign extended because it needs to match the size of the instruction pointer to be added, which will always be 8 bytes. The sign is also always preserved in a relative immediate. This is because the x86-64 specification allows for negative offsets to the instruction pointer. This task is implicitly performed by the SignExtend function.

Justification

This is the best approach to solving the problem because it ensures that the instruction pointer is added after the immediate has been read, not whilst. This would be a problem otherwise because the instruction pointer would not point to the intended location at that point in time, rather point to just before the immediate, which would not be the correct address to point to.

NoOperands



Explanation

The NoOperands IDecoded class will allow opcodes to opt to take no operands as their input. This would be the case for opcodes such as RET(Return), which does not have an explicit operand.

Justification

It is necessary to solve this problem in a separate class in order to comply with the IDecoded interface, such that the opcodes which do not take any operands can still be used in the code the same way that other operands are. This will allow the general solution of handling operands by interface rather than type to still apply throughout the solution.

Throw an exception when set

Explanation

When the NoOperands operand is attempted to be set through the interface method, an exception should be thrown.

Justification

This is the best approach to the sub-problem because if the opcode using the NoOperands operand is attempting to set its value, there is something going wrong, as the data would not go anywhere. This is supported by the fact that no opcode in assembly takes NoOperands as their operand type as well as another opcode, therefore there is no reason why the interface method should be used. In the case that it is used, the developer has likely made a mistake somewhere so the exception will help them fix the bug in their code

Return nothing for disassembly and fetched calls

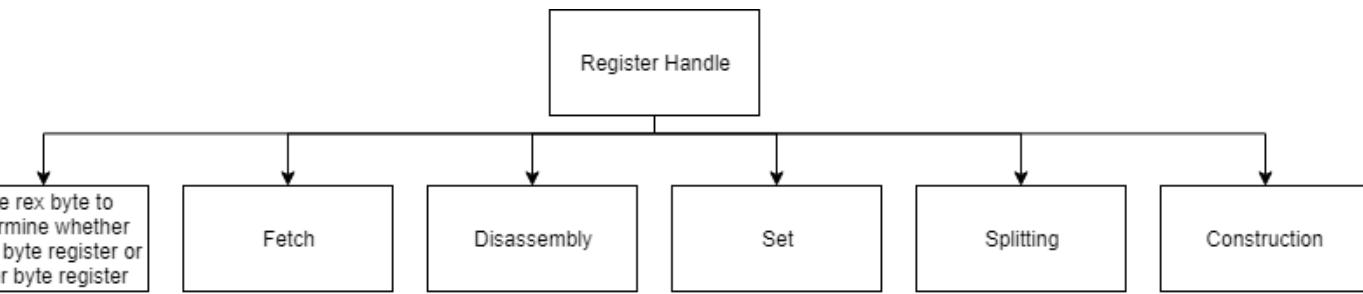
Explanation

There is no disassembly or returned value for no operands. This will be implemented by returning empty lists for those methods.

Justification

This is the best approach to solving the problem because if a null value was returned, or an empty string, there would likely be errors elsewhere in the code because those modules expected the input to be the same as any other operand. To fit in with the behaviour of other operands, the empty list will be returned as when the caller uses the AddRange() function for example, their list will not be changed because the list returned was empty, however there is also no NullReferenceException because there is still a list.

Register Handle



Explanation

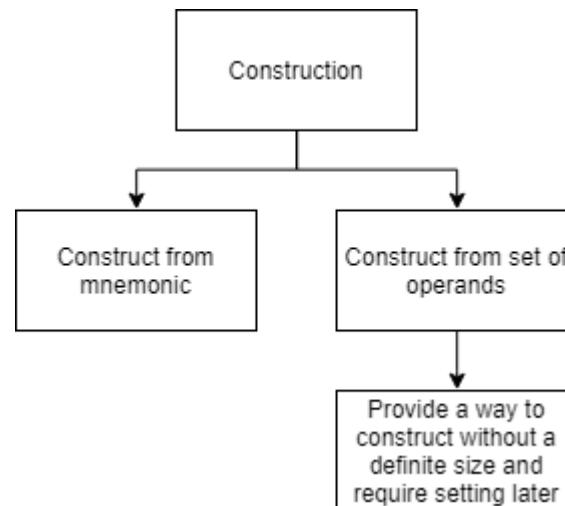
The register handle class will allow registers to be fetched and modified. Its usage will not be limited to operands; other places in the code may have reason to use a register, but it will implement the IDecoded interface so that it can.

Justification

This is the best approach to solving the problem because it will allow for more consistent results and method usage. For example, one of the classes have slight differences in return values compared to the other. Having two separate classes for accessing registers will be harder to maintain and will introduce more bugs, as changes to the code will have to be applied to both classes. This shows that the best approach is to combine the two and allow this class to be used for all access to the registers. Direct access to the RegisterGroup class should not be allowed because it introduces:

- Bad code practice. Higher level modules will be using the low level data structures instead of the abstraction methods that simplify the solution provided in the intermediary layer.,
- Greater dependence on the RegisterGroup class. If the method of operation is changed for the RegisterGroup class, all instances where the old methods are used will have to be re-coded and tested again, instead of only having to code and test the methods in the RegisterHandle.
- Code repetition because the useful re-usable methods that solve common problems will be provided in the RegisterHandle, such as fetching and disassembling registers. Using the pre-tested and pre-coded solutions will save development time.

Construction



Explanation

The RegisterHandle class will have two construction methods. One way will be to construct from mnemonic. A dictionary will store the mnemonic in a key and the value associated with it will hold the necessary parameters to construct the register handle from it. This includes a RegCode, a RegisterCapacity, and a RegisterTable, as all of these can be implied from the mnemonic. These parameters will be stored and used by other methods to access the register after the constructor has finished.

The RegisterHandle can also be constructed by parameters. These would be the same parameters that would be implied by the mnemonic. A RegisterCapacity will not be necessary to construct the RegisterHandle in this way, but may be specified if required.

Justification

It is necessary to handle the construction from a mnemonic separately as it will be useful when reading from the testcase file, as there will need to be a way to validate and interpret the user input when specifying which register to compare the value of certain registers against the testcase. Constructing from parameters is also a necessary sub-problem as it will allow for RegisterHandles to be created programmatically at run-time through variables. Without this, a separate routine would be required to construct the mnemonic from the variables, which only adds an extra conversion process that has to be performed, just to parse the string back to those variables again. It is important that the RegisterCapacity does not have to be specified because some opcodes have an “Implicit register” as their operand. This is where a specific opcode will always use a certain register, but its size is not implied. In this case, the register handle can be hard-coded as an operand to the opcode, then at run-time the size will be set by the caller(as is necessary to use the other methods, but does not specifically have to be done at initialisation).

Validation

Validating the input mnemonic to the constructor can be done with the dictionary. As the number of possibilities for a correct register are small, the correct mnemonics can be hard coded into the dictionary. The user input can then be validated by using the Dictionary.ContainsKey() method. If the dictionary does not contain the key, it must be an invalid mnemonic.

Justification of validation

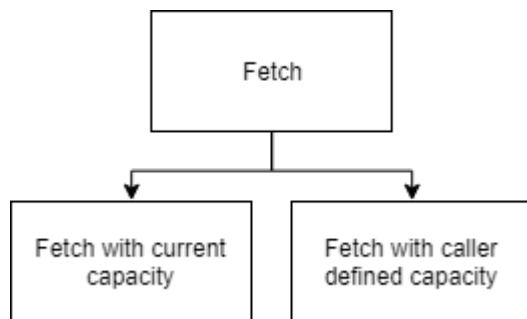
It is necessary to perform this validation because there will be an exception if the dictionary is accessed with a key that is not present. This will cause the program to crash on the invalid input, whereas by imposing a validation method, the user can be told that the input was invalid, which is more informative than a “KeyNotFoundException”. This is the best approach to validating the input as it is very simple and allows for mnemonics to be changed and added easily.

Iterative test data

Test Case Description	How	Test Data	Normal /Erroneous/Boundary	Expected Result	Justification
Test the construction of a register by mnemonic	A constructor call with a hard coded parameter, then compare with the intended results. A breakpoint will be placed after the function call so it is certain that no other sources have changed the register after the test.	“EAX”, “AX”, “RDX”, will be used as parameters	Normal	For EAX, RegisterCode = 0, RegisterCapacity = DWORD, RegisterTable = GP For AX, RegisterCode = 0, RegisterCapacity = WORD, RegisterTable = GP For RDX, RegisterCode = 2, RegisterCapacity = QWORD, RegisterTable = GP	This test is necessary to check whether the constructor will create the correct settings given the mnemonic. This will make sure that the RegisterTable, RegisterCapacity, and RegisterCode variables are set properly.
Test invalid construction of a register by mnemonic	A constructor call with a hard coded parameter that is erroneous, such that it is not present in the dictionary.	“NotARegister” will be used as the input parameter	Erroneous	An error will be logged and displayed to the user by a message box	This test is necessary to make sure that the error is thrown when the input is invalid. If it was not tested, there may be bugs that go undetected later in the development process
Test the construction of a register by parameters	Parameters will be used to construct a register. This can be done using a function call with hard coded	new RegisterHandle (RegCode.A, RegisterTable, GP, RegisterCapaci	Normal	The Size, Table, and Code variables will all be set to the input values.	This test is necessary to make sure that registers are being constructed properly. The program would not work if the registers were not being assigned to the parameters in

	parameters, then checking if the variables where the parameters are stored are set after the constructor finishes execution	ty.WORD)		the input. This would be hard to track down in a bug if not tested as the parameters would be generated programmatically, where as the comparison is easy when tested with hard-coded inputs.
--	---	----------	--	---

Fetch



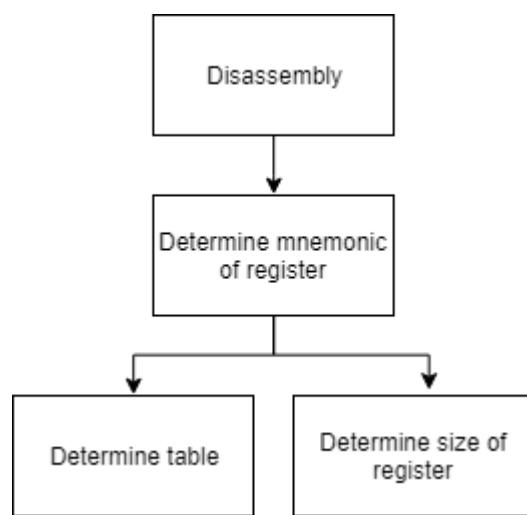
Explanation

The fetch routine will be the implementation of the IDecoded interface fetch field. This will fetch the register from the RegisterGroup in the CU and apply any necessary procedures that have to be performed. Two variants of the function will be included, one that uses the Size field inherited from the interface, and one that allows the caller to choose the capacity.

Justification

This method has to be included as part of the interface, and to allow registers to be used across all layers(as the RegisterHandle is not restricted to the processing layer as it is an abstraction). It is necessary that the two variants of the method are included because it will allow for cleaner code. If the caller is accessing the same register multiple times, they can use the method that allows them to choose the capacity such that they do not have to create a new instance of the handle every time. It is also necessary that a method fetches with the current capacity because the interface field does not specify a size in the function call. In this case, the size field used by the interface will be used, such that procedure that decides the register capacity for the opcode can decide which RegisterCapacity is to be used instead of the caller having to specify.

Disassembly



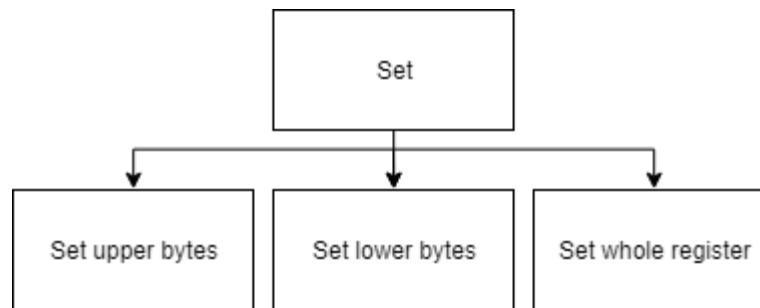
Explanation

The disassembly method will output the mnemonic of the register based on the parameters stored by the constructor. This will be the inverse of the procedure performed by the mnemonic constructor. The parameters will be used as an input to the dictionary to determine which key has this value(as the values will all be distinct). The table and size of the register will affect the mnemonic, so must also be used to find the key in the dictionary as the RegCode will be used by many other registers, such as the registers of other word sizes.

Justification

This is the best approach to disassembling the register because as the RegisterHandle class is the only class used to access registers, there will be no repeated code or constants. This is because the same dictionary can be used to construct the handle, can also be used to disassemble it. If a procedure in another class was used, the constants would have to be stored twice. Not only would this be more memory consuming, but if there was ever cause to change or update the mnemonics, it would have to be done in multiple places, such that there could be inconsistencies which would be confusing as it would be not immediately obvious to the user that two different mnemonics represent the same register, therefore it is important that the disassembly is centralised.

Set



Explanation

The set procedure will allow the value of the register to be set by the caller. There will be different methods to set different parts of the register, as some operations may require. Setting the upper bytes will set the upper half of the register to a given input. Setting the lower bytes will set the lower half of the register. These two operations will not affect the other half of the register that they do not set. Setting the whole register will be the interface field that opcodes will mostly use to interact. This will set the whole register, that is limited by the Size field.

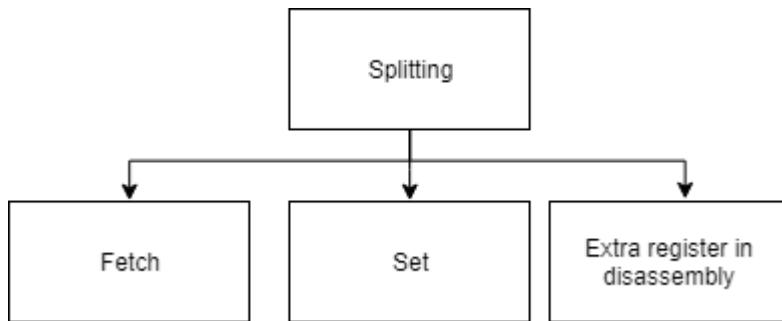
Justification

The division of this problem is necessary as certain operations will require different methods of setting the outputs. For example, when the DIV instruction instruction is performed on a byte register, the modulo goes into the upper byte of the A register, and the quotient goes into the lower byte. Using the SetUpperBytes and SetLowerBytes methods will allow this to be performed. In every other case, the whole register will be set. This is necessary for general usage such as storing the result of an operation.

m	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
Set the upper bytes of a register	Use the SetUpperBytes method to check the upper bytes are set.	Clear EAX, set the lower word to AAAA(by using the visual studio debugger, not programmatically), create a register handle to EAX, then use the SetUpperBytes method with a byte array in the parameter that is 2 bytes long, containing FFFF.	Normal	The upper word will be FFFF, the lower word will be AAAA	This test is necessary to make sure that the correct number of bytes are set in the right position, as it could be easy to code this wrong where the bytes are set at the wrong offset. It will also be important to make sure that the lower bytes are not cleared in the processor, hence is important to check the lower word is still AAAA.
Set lower bytes of the register	Use the SetLowerBytes() method to check the lower bytes are set.	Clear EBX and set the upper bytes to FFFF, create a register handle to EBX, then use the SetLowerBytes method with a byte array in the parameter	Normal	The upper word will be FFFF and the lower word will be AAAA	This test is necessary to make sure that the lower bytes are set properly, and do not overwrite the upper bytes. This would be an easy place to make a logical error, such as setting the wrong number of bytes or at the wrong position.

		that is 2 bytes long, containing AAAA.		
Set upper bytes and lower bytes of two registers erroneously	Use the SetUpperbytes() and SetLowerBytes() methods to attempt to erroneously set the EAX and EBX registers by more than half a register.	EAX and EBX will be cleared. SetUpperBytes() will be called on EAX with an array of size 4(the size of EAX) containing all FF bytes. SetLowerBytes() will be called on EBX with an array size of 4, containing all FF bytes.	Erroneous	The upper word of EAX will be FFFF and the lower word will be 0000. The upper word of EBX will be 0000 and the lower word will be FFFF.

Splitting



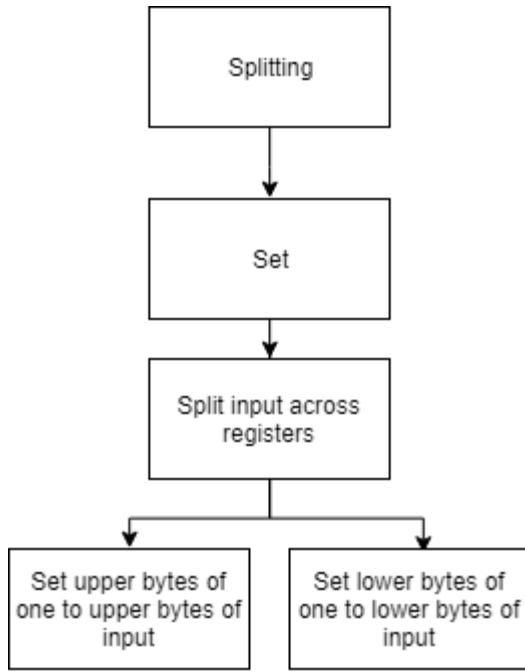
Explanation

Register splitting will allow multiple registers to be combined as one operand, such that they can be fetched and set together, without having to code the algorithm in each instance it is needed. The extra register in disassembly will show that two registers are being used for the opcode. This will be displayed like "A:B" where A is the upper bytes and B are the lower bytes

Justification

Register splitting will be necessary for opcodes such as DIV and MUL. MUL stored the result of a multiplication in two registers, where the upper bytes go into the D register and the lower bytes go into the A register. DIV uses this the other way round, the upper bytes of the D register are used as the upper bytes of the dividend, and the A register is used as the lower bytes of the dividend. For this reason, both separate procedures are needed. Having the extra register in disassembly is necessary as it could be confusing to a new programmer, such as a stakeholder of the project, if multiple registers are being modified even though only one is shown in the disassembly(as would be shown in another debugger). For this reason, concatenating the two with clear notation is an appropriate solution to the problem.

Set split registers



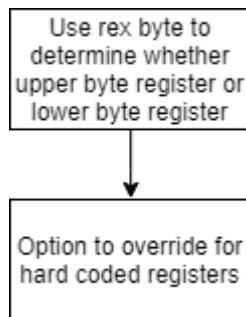
Explanation

To split the input across two registers, there are two different problems to handle. The upper bytes have to be selected from the input and then used to set the register that holds the upper bytes. The lower bytes have to be selected from the input then used to set the register that sets the lower bytes. This can be done with the Bitwise.Subarray() and Bitwise.Cut() methods.

Justification

It is necessary to separate this problem from the general register set method because the procedures of setting the bytes are very different. The general set method will set the full register capacity to a value, but the split register set method will split the input into two registers, therefore a very different procedure that must be tackled separately because of this. This is the best approach to solving this sub-problem because existing methods from the utility library can be reused to solve the problem.

Using rex byte to identify registers



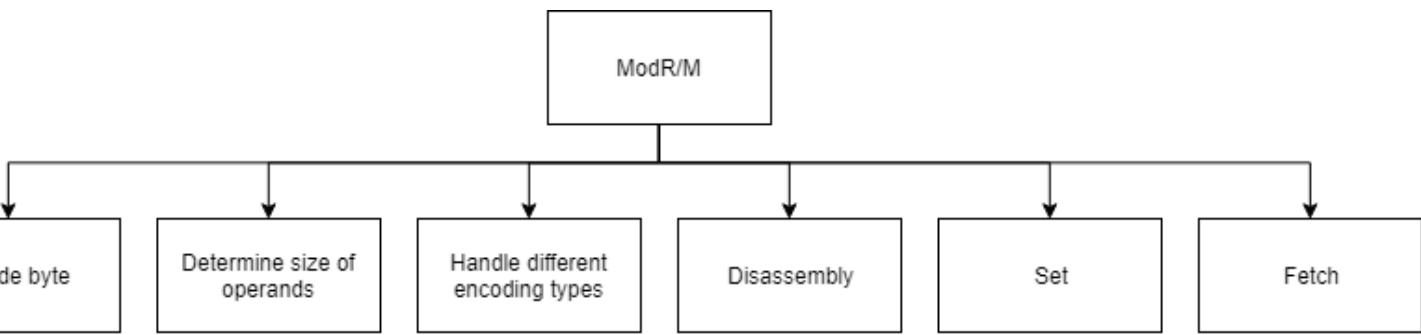
Explanation

The REX byte will determine which registers are used in some cases. For example, whether EAX is used or R8. For this reason, it is important that the CU is checked to check the REX byte, such that the correct register is selected. There will also be an option to override this.

Justification

It is necessary to tackle this problem separately because it needs to be done automatically; the caller will not consider the REX byte. This caller could be the procedure that creates instances of the operand types. In this case, due to the modularity of the program (there is class independence), the details of the REX byte will be abstracted from that class. Because of this, the procedure must be automated inside the RegisterHandle because it is part of the CU, therefore is permissible to depend on the CU according to the success criteria as it is part of it. The option to override this procedure will be necessary as RegisterHandles that are hard-coded in other modules to use a specific register will want to use that register regardless of the REX byte, hence an option for them to ignore it is required.

ModRM



Explanation

The ModRM byte will handle the decoding of ModRM bytes, which contain “Mod”, “Reg”, and “Memory” fields. These are used to decide which register will be used as an operand and whether it is a pointer or not, plus a few implied settings that will be discussed when relevant. The class will aim to reproduce this table in a code-effective manner,

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(r) r16(r) r32(r) mm(r) xmm(r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--] disp32 ²		100	04	0C	14	1C	24	2C	34	3C
[ESI]		101	05	0D	15	1D	25	2D	35	3D
[EDI]		110	06	0E	16	1E	26	2E	36	3E
		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8 ³	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Justification

It is necessary that this sub-problem is separated from the rest of the solution because it is a commonly performed tasks and very lengthy. By abstracting the class into pure inputs and outputs through means of the IDecoded interface, the specific procedures of decoding bytes is abstracted outside of the class.

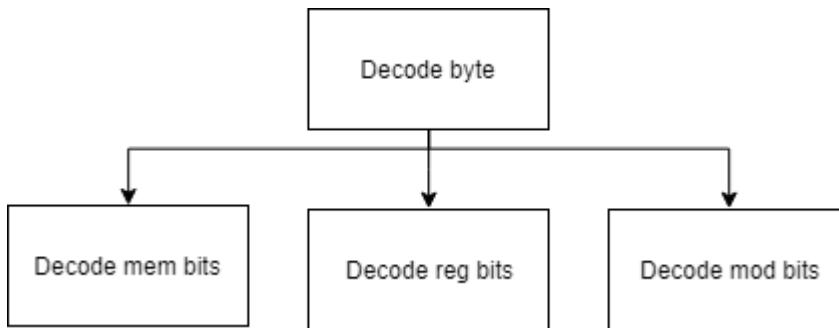
Key variables

The key variables in the ModRM class will be the source and destination variables. These will be a ulong values that either represent a reg code(stored as a long) or a pointer in memory. The mem bits will be stored alongside this value and used by the methods in the class to determine whether the destination and source variables is to be treat as a pointer or as a register. This will require casting the variables to a RegCode where applicable, or left as ulongs when referencing memory.

Key variables justification

These variables are necessary in the class because they will allow the complex procedure of mapping the mem and reg bits to their correct registers to be contained in the constructor. This will allow the mem and reg bits to be abstracted from the class after the constructor finishes executing, such that abstractions can be used instead. The source and destination variables are this abstraction as they represent the stored output of the procedure, e.g. a pointer into memory. This means that the byte will not have to be decoded more than once.

Decode byte



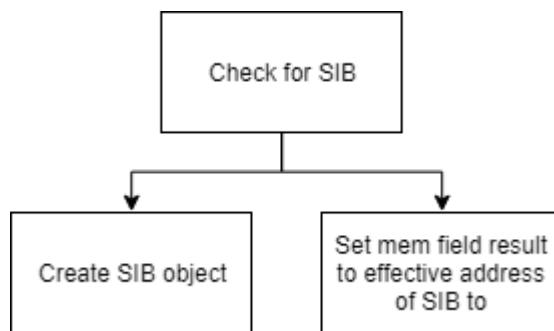
Explanation

The task of decoding the input byte can be split into three separate sub-problems. These can all be handled separately as they are not dependent on each other, such that they can be simplified by applying a general solution to each. Each of these subtasks will perform the necessary procedures to map the portion of the input byte that corresponds to their designated field, to the position in the table.

Justification

It is necessary that this problem is split into three sub-problems because it allows for a general solution to be applied to each. This is a better approach to solving the problem because a hard-coded solution would have a much larger code size as there are many different possibilities.

Check for SIB



Explanation

As part of decoding the byte, a SIB will have to be checked for. A SIB is an extra byte following a ModRM that allows for more specific operands. This will be done by creating a subclass to hold the information of the SIB and serve the purpose of handling the inputs and outputs of the SIB. Like the ModRM, it will map the bytes to their position on a table. This can be done by having procedures that determine which bits of the map byte to a certain position, then pass the results back to the ModRM which can then apply these changes to the mem field.

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

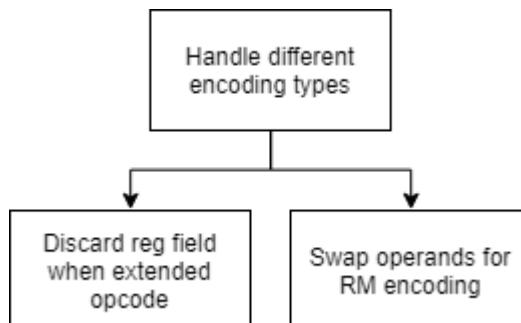
Justification

It is necessary to have this task performed separately because it can be abstracted into a subclass as part of the ModRM class. This can be done as the SIB byte is only ever used by the ModRM. This will abstract the need for outside callers to understand the SIB class or interact with it as the ModRM class will handle all the interactions and handle the outputs of the SIB appropriately, such that the results are immediately used to determine the outputs of the ModRM, rather than having to deal with two separate classes. This is the best approach as code will be simpler to maintain and classes will not have to depend on functions that they do not use, such as the decoding procedures in the SIB class.

Test data for SIB

Test case	How	Test data	Expected result	Justification
Test register plus offset	Test data containing specific instructions containing a ModMR byte that implies a register plus offset SIB operand.	(These three instructions will be used in every test) mov eax,0x100 mov ebx,0xff mov ebp, 0x8000000 mov DWORD PTR [eax+0x50],ebx	<Memory offset="150">FF</Memory>	This test data is necessary as it will test the basic functionality of the SIB as only an offset is applied. An error in this test would indicate a fundamental logical error in the algorithm as opposed to a specific part of it. Because of this, less development time will have to be spent debugging and tracking down bugs as it will give a good starting point on where to start looking.
Test scaled register and offset	Test data containing specific instructions containing a ModMR byte that implies a scaled register plus offset SIB operand.	mov DWORD PTR [eax*2+0x0],ebx mov DWORD PTR [eax*2+0x100],ebx mov DWORD PTR [eax*4+0x20000000],ebx	<Memory offset="200">FF</Memory> <Memory offset="300">FF</Memory> <Memory offset="20000400">FF</Memory>	This test data is necessary as it will test the multiplication performed by the scale coefficient. The conditions for this operation to be performed are specific, making it a special case. For this reason, it is important to make sure that there is test data that proves the functionality of the special case in the algorithm.
Test multiple register addition with offset and scale	Test data containing specific instructions containing a ModMR byte that implies a base register added to a scaled register plus offset SIB operand.	mov DWORD PTR [ebx+eax*8],ebx mov BYTE PTR [eax+ebx*1+0x100],bl mov DWORD PTR [ebp+eax*2+0x10],ebx mov cl,BYTE PTR [ebp+eax*2+0x10]	<Memory offset="08FF">FF</Memory> <Memory offset="2FF">FF</Memory> <Memory offset_register="BP" offset="210">FF</Memory> <Register id="C" size="1">FF</Register>	This test data is necessary because it will test all the features provided by the SIB byte at once. This will allow test data to exist that is conclusive enough to assume that the SIB procedures will work in the general case.

Handling different encoding types



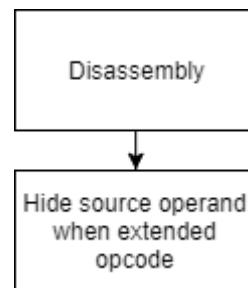
Explanation

There are different encodings of ModRM that will change the order of the operands. The RM encoding will require the mem and reg fields to be swapped. This is so the reg field can also be a pointer(the reg field is the source, mem field is the destination). This will be implemented by having two variables called “Source” and “Destination”. When the RM encoding is used, the two will be swapped from what they are normally. These variables will be used in all the operations that use the source and destination. When an extended opcode is used, the reg field is used as extra bits to determine the opcode and are not an operand. This means that a boolean parameter will have to be used, such as “IsExtendedOpcode” to determine whether this is the case and will be specified by the callers.

Justification

It is important to tackle these problems separately because they can still have the general solution of mapping bytes applied to them but they imply a few preconditions that need to be handled before the mapping procedure is performed. This is the best approach as it will result in a shorter code size as the procedures will not have to be repeated in every instance, and the results of the code will always be the same, i.e. a bug will be present in all encodings of opcode, not specific to a single one. This will be helpful in identifying and fixing any bugs as they will be very noticeable, not hidden away in a function that is rarely used.

Disassembly



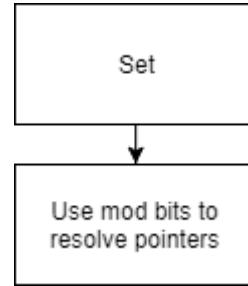
Explanation

The disassembly task will handle the general disassembly of the ModRM bytes. It will cover the method required by the IDecoded interface, allowing it to be included in any general solutions that use the Disassembly method of an IDecoded class.

Justification

Disassembly is a necessary problem to handle separately because it will perform a routine that is used often and requires a significant amount of code. The data that the mem/reg bits point to has to be determined. For example, a pointer will appear differently in disassembly than a register. For this reason, it is important that the task is handled in a self-contained method such that results are consistent.

Set



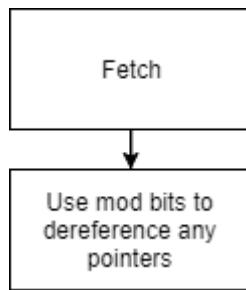
Explanation

The Set function will allow the destination of the ModRM byte to be set. This will be the implementation of the IDecoded interface Set method. This will allow for general solutions in other modules to use the interface method. The mod bits will be used to determine whether the destination is a pointer or a register. This will allow one method to solve both problems with no effect on the caller.

Justification

The set function will be necessary to generalise the procedure of setting the destination. It is necessary to tackle this problem separately because there are specific conditions based on the mod bits that determine whether the destination is a memory location or a register. These two locations require two different procedures to set their memory. For this reason, the best approach is to handle the process in the ModRM class allowing the details irrelevant to the caller, such as the mod bits, to be abstracted.

Fetch



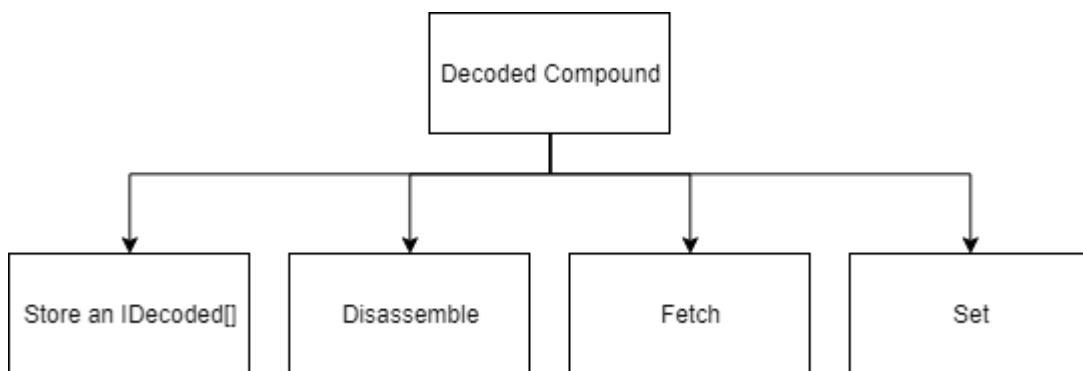
Explanation

The Fetch function will fetch the source and destination of the ModRM byte. This will be the implementation of the IDecoded interface Fetch method. The mod bits are used in the same way as the set function to determine whether the source is a pointer or a register.

Justification

It is necessary to separate the fetch task from other parts of the class because it will use the variables stored in the key variables source and destination, in order to generalise the procedure based on the outputs of the constructor. This means that the fetch condition will only have to use an if condition to determine whether the variables represent pointers or register RegCode by using the stored mod bits. The best approach to this task is to return both the destination and source because most operation will require this information. For example, to add, both numbers must be known in the addition. Returning both the destination and the source will allow for a greater feature set available to the opcode classes.

DecodedCompound



Explanation

The DecodedCompound class will store multiple IDecoded objects and allow them to be concatenated into one input. This will allow multiple operands to a single opcode. This will be used for opcodes such as IMUL which can have a ModRM and an immediate value that have to be handled separately.

Justification

It is necessary that this sub-problem is handled as a separate problem because it will be possible to have a general solution rather than a solution specific to individual operands or opcodes. This will allow the solution to be applied to future opcodes that are implemented post-development without having to redesign or deprecate code and modules. It will also make use of the interface methods provided by the IDecoded interface such that the operands are accessed as if they were the only opcode, such that there is no affect on their behaviour.

Store an IDecoded[]

Explanation

The constructor will take an IDecoded[] as a parameter and then store the operands in a private variable such that they can be accessed by the functions of the class. This will allow the operands to perform iterative operations, such as calling a single interface method on all the IDecoded objects in the array and returning them to the caller in concatenated form.

Justification

This is a necessary task as it will allow the main features of the class to be used dynamically, such that they can be called at any time and produce results, i.e. is not restricted to a single usage or function call.

Discriminative vs indiscriminative operations

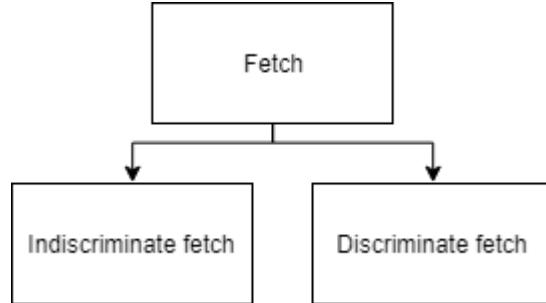
Explanation

Two forms of iteration can be applied to an operand in the IDecoded[], an indiscriminative operation or discriminative operation. The difference is that in an indiscriminative operation, only one operand will be taken from each input before moving to the next. In a discriminative operation, if an operand returns multiple values for an operation(as IDecoded methods all return lists), the entire list will be appended not just the single item.

Justification

It is necessary to separate these two tasks as different callers will require different methods of selecting the operands. This applies only to callers which know they will receive a DecodedCompound as a parameter as these methods are not present in the interface. This disambiguation is necessary for opcodes that affect both the source and destination operands. For example, the XCHG opcode swaps the values of the destination and the source.

Fetch



Explanation

The fetch method will have both discriminate and indiscriminate forms. The indiscriminate form will take only the first operand returned of each IDecoded, e.g.

```
IDecoded[] InternalArray;  
DEF Indiscrim_Fetch()  
    List<byte[]> OperandValues = new List<byte[]>();  
    FOREACH Operand in InternalArray DO  
        // Add only the first index of the returned value  
        OperandValues.Add(Opcode.Fetch()[0]);  
    HCAEROF;  
    RETURN OperandValues;  
FED;
```

This algorithm will loop through each index of the internal array and add the first element of the returned list from the Opcode.Fetch() method, effectively ignoring any extra elements on after it. This is a complete solution to this task as it does exactly as the task describes, to allow each operand to return no more than one operand.

The discriminative fetch will iterate adding every returned value, not just the first.

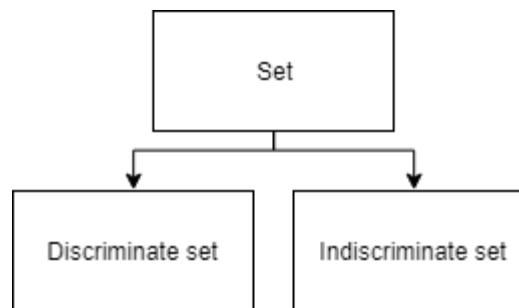
```
IDecoded[] InternalArray;  
DEF Discrim_Fetch()  
    List<byte[]> OperandValues = new List<byte[]>();  
    FOREACH Operand in InternalArray DO  
        // Add all returned items to the result list  
        OperandValues.AddRange(Opcode.Fetch());  
    HCAEROF;  
    RETURN OperandValues;  
FED;
```

This algorithm will loop through each index adding all of the elements of the returned array to the output list rather than only the first. This is a complete solution to this task as it handles all situations that a discriminative search would cover because the general AddRange() extension method for a list can be leveraged to simplify this problem.

Justification

It is necessary to separate these two tasks because their returned values are different. For example, an operand has a ModRM and an immediate as an operand. If an indiscriminative fetch is used, only the destination of the ModRM and the value of the immediate will be returned, whereas if a discriminative fetched is used, the source and destination of the ModRM will be returned as well as the value of the immediate. Because of this, two separate procedures are required to be available to each caller that requires the specific functionality. However, the interface method that is used in general by opcodes is the discriminative fetch because it is most common behaviour to expect all operands to be concatenated.

Set



Explanation

The set method will have both discriminate and indiscriminate forms. The indiscriminate form will set the IDecoded at a specific index in the array, e.g.

```

IDecoded[] InternalArray;
DEF Indiscrim_Set(byte[] data, int index)
    InternalArray[index] = data;
    RETURN;
FED;
  
```

This algorithm will set the index of the array at the parameter "index" to the value of data. This is exactly what the indiscriminative set should do as each operand is being treated as only one operand; not holding multiple values. The discriminative set is more complex. Each operand will be iterated and a counter will be incremented by one if the operand has only a destination, or by two if it has a destination and a source. This will effectively treat sources of operands as individual operands.

```

IDecoded[] InternalArray;
DEF Discrim_Set(byte[] data, int index)
    int i;

    // Iterate through all elements of InternalArray
    FOR i < InternalArray.Length DO
        // If the target index is reached,
        IF i == index THEN
            // Use IDecoded interface method
            InternalArray[i].Set(data);
        FI;

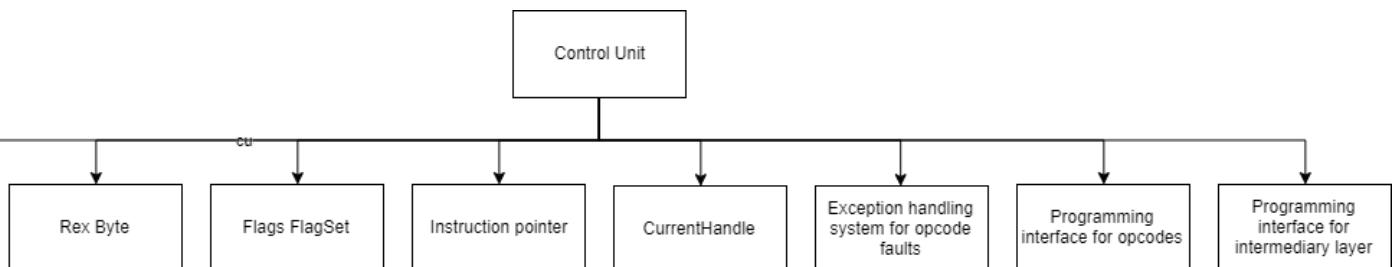
        // If the current index has a source, it needs to be treated separately.
        IF InternalArray[index].HasSource THEN
            i++;
            IF i == index THEN
                InternalArray[i].SetSource(data);
            FI;
            FI;
            i++;
        ROF;
        RETURN OperandValues;
    FED;
  
```

This algorithm will iterate through all the elements of the InternalArray, incrementing by one when the operand has only one value, but if the HasSource boolean is true, the source will be accounted for. This is a complete solution as the scope of discriminancy covers only one or two returned values per opcode, as the maximum that needs to be handled in the solution is two, which is in the ModRM byte. No operand in the x86-64 standard has the potential to imply more than two sub-operands.

Justification

It is necessary to separate these two tasks because their returned values are different. If the wrong method is used, it is likely that the caller will be setting the wrong value. Therefore, it is important that the difference between the two is clear. It is also important that the same range of functionality is offered to the set method as is offered to fetch method, as if both are not included the other would essentially be useless.

ControlUnit



Explanation

The ControlUnit will handle the fundamental procedures that need to happen in order to emulate successfully. For this most part this will include decoding instructions and calling other modules as appropriate, also providing a programming interface for other modules to control the flow of execution and manipulate the virtual memory space.

Justification

It is necessary to have the CU as a separate module as it will define which procedures can take place at a given time and impose the rules that enforce the x86-64 specification. These need to be separated to ensure that they are always performed and are consistent in doing so. Having each class decide what is considered a valid operation would lead to code that is hard to maintain as a change in the specification would require the entire solution to be modified as opposed to one class.

Key variables

Instruction Pointer

Explanation

The instruction pointer will hold the address of the next instruction to be read. This is the abstraction of the RIP x86-64 register. It will be a ulong type and be a public variable that can be read by other modules

Justification

This variable is a necessary abstraction as it does not have to be stored as a byte array as it rarely used in calculations and is always 8 bytes long, as opposed to registers where different word sizes can be manipulated. This allows for the IP to be abstracted as a ulong value instead, which will allow for simpler usage as operators provided by the language can be used on it, such that the maintainer does not have to understand any other modules such as the Bitwise library to understand it.

Rex Byte

Explanation

The REX byte will hold specific information related to operands that will be used by other modules to determine the state of the CU and what procedures need to be done because of it. This is a concept of the x86-64 architecture, but the variable itself will be a public variable in the class. There are four different fields in the byte which are assigned to certain bits of the byte. These are called W,R,X,B. These fields will be generalised into an enum.

Justification

The REX byte is a necessary variable because it will allow other modules to check whether they need to adjust their behaviour based on the current situation in order to emulate accurately. An enum class is most appropriate because it will allow for the fields to be combined, such that a Rex.W and Rex.R etc can be present at the same time, as is possible and very common in the x86-64 architecture, hence is essential to include.

Flags

Explanation

The Flags variable will hold all the current flags of the CU. This can then be set by modules to update the flags, such as setting the results of a bitwise operation. It can also be checked by modules to determine whether a certain condition is met, such as a conditional jmp, allowing greater functionality provided to opcode classes through the programming interface

Justification

This variable is necessary as it will be a required abstraction that can then be made use of by other modules to allow the modules to interact with the flags in a simple

and code efficient manner, making use of the procedures provided by the FlagSet class.

PrefixBuffer

Explanation

The prefix buffer will provide generalisation to the storage and conditions to determine whether a specific instruction prefix is present. Internally it will hold an array of size 4 to hold the maximum capacity of prefixes possible. This is because certain prefixes will replace others when decoded afterwards. This is generally an erroneous case, as there is no benefit to having multiple prefixes of the same type preceding an instruction; the first will just be ignored per the Intel specification.

Justification

This variable is necessary as it will impose the conventions of the x86-64 specification by having set procedures to determine the steps taken when certain conditions are met such as multiple prefixes of the same type leading, to consistent behaviour of the buffer, as opposed to having a list or similar and allowing callers to check by themselves, which would also use more code.

CurrentHandle

Explanation

The CurrentHandle variable will hold the current handle object that is using the CU at a given time. It will be public and so can be checked by external callers. The variable will be set by the Handle class when function to invoke the CU is used.

Justification

This variable is necessary to allow other modules to check which handle is executing and whether it is safe for them to assume control of the CU, which if in use already would cause undefined behaviour for the caller and the thread of the current handle. It is necessary that this variable is stored in the CU class as it is a static class so the variable will be the same for every caller, which will be important for having the same value shown to every caller, rather than an instance based solution.

CurrentContext

Explanation

The CurrentContext variable will hold a reference to the current context. This will be the context used by the functions in the opcode api and intermediary api to interact with the required tuple input to the CU in order to execute(discussed in the context section). This reference will be to the instance of the context owned by the current handle that is being operated on in the current frame of execution. If another handle is to use the CU, the current context will be swapped with a reference to the context of the new handle.

Justification

It is necessary to have this variable because it will allow for a simple way to have the same context used by all modules at a given time; there will be no hard-coded usage of a context to a specific handle. This is essential as it will allow for the main aspects of modularity to be included at the processing layer, as instead of modules depending on many classes, they only have to depend on the CU to provide a context that it has to work on.

Exception handling for assembly level exceptions

Explanation

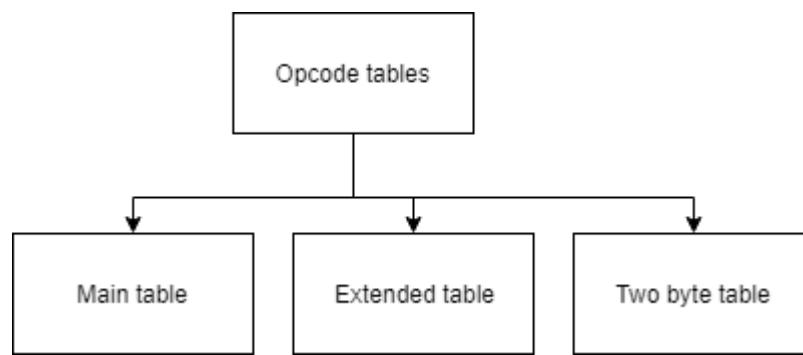
In the x86-64 architecture, there are exceptions that can happen as part of the emulation that would tell the user of an assembly level fault caused by their program. These are the same exceptions that would be seen if the user's code was executed in a non-virtualised environment. The CU will provide a method of raising an exception and define the necessary procedures in the event of an exception.

Justification

This feature is necessary to create a better representation of reality for the user. For example, if the IDIV opcode is used to divide by zero, the #DE fault will be raised instead of raising an exception in the C# code that would cause the program to crash. This is the best approach as the user will need to understand these errors if they intend to use their

code in a production environment. The fault will be displayed to the user by means of a messagebox, which is the best approach as it is not intrusive and does not require them to restart the program.

Opcode table

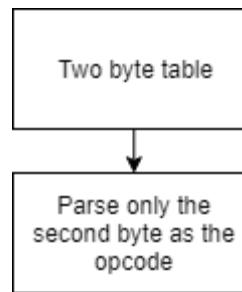


Explanation

Opcode tables will hold all the mappings of bytes to their corresponding opcode, such that the CU can call the table, supplying a byte as a parameter to decode and execute the opcode. This problem can be split into three sub problems: the main table, extended table, and two byte table. These are all different tables that hold opcode mappings. The one to be used is determined by certain preconditions based on how the instruction was decoded. For example, the byte 0x20 might map to opcode A on the main table, but opcode B on the extended table, hence requires a clear disambiguation. To solve these problems, a general solution will be used. A dictionary will be used to store every opcode mapping, then the CU will specify which table is to be used when the function is called. This abstracts the details of the preconditions from the opcode table.

Some elements of these sub-problems will need to be handled differently,

Two byte table



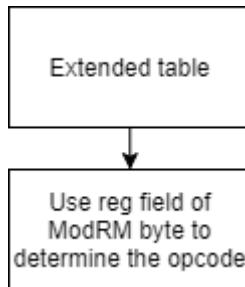
Explanation

The two byte table is a separate table that is only used if and only if the previous byte decoded was 0x0F. Hence, it is called the two byte table because it takes two bytes to store the opcode instead of one as in the normal table. This means that the 0x0F will have to be ignored, then a boolean set to indicate that the two byte table will be used when the next instruction is read.

Justification

It is necessary to handle this problem separately because it requires a condition to be checked to determine whether the two byte table is the table that will be used to map the next instruction or the main table. The process of fetching the mapped opcode beyond that will follow the same method as the main table, therefore will still be possible to make use of the decoding procedures used by the main table, except using the two-byte table to map the opcode instead.

Extended table



Explanation

In the extended table, the reg field of the following ModRM byte is used as part of the opcode, such that a single byte can imply 8 different opcodes based on the reg field. This will be handled by having a nested dictionary. For example, when the byte 0x20 is the parameter byte used to requested to be mapped to an opcode, instead of returning the opcode, another dictionary will be returned which will hold keys 0-8 that can be used to select which of the 8 different opcodes is to be used.

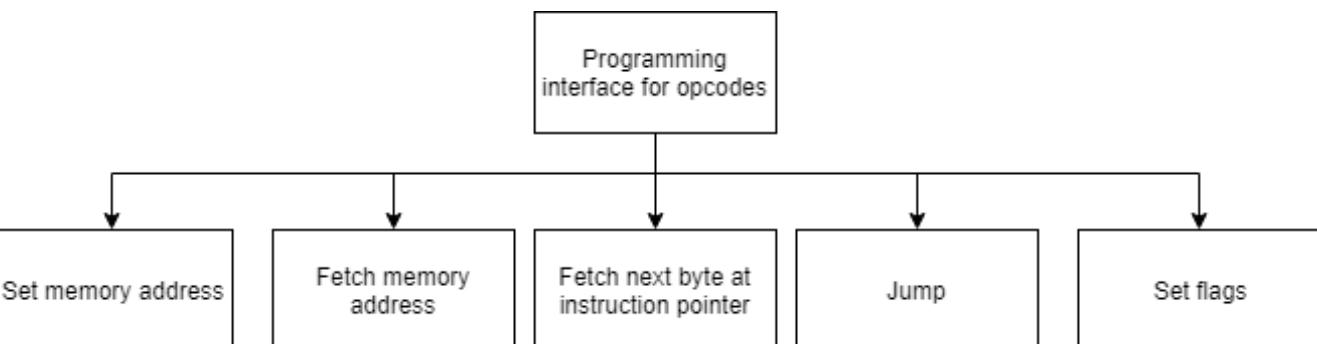
Justification

It is necessary to handle this task separately as it cannot have the same general method applied to it that can be applied to the main table and two byte table; an extra step of checking the returned dictionary has to be performed instead of having the opcode returned the first time.

Justification

It is necessary to handle this task separately from the other modules because it will allow for simple extension and modification of the table mappings as opposed to having the mappings spread throughout the program, which would not be as defined or easy to interpret. Because of this, having all the mappings in one place is the best approach in order to improve maintainability.

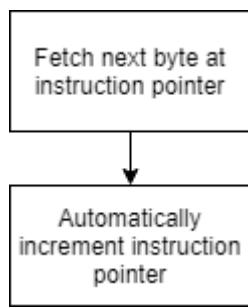
Opcode api



Explanation

The opcode api will provide pre-defined procedures and routines to the opcode modules that can be used to interact with the CU in a safe and conventional way, such that further abstractions can be applied to the data structures that tunes there functionality more accordingly to conventional x86-64 behaviour rather than absolute freedom of control. An example of this would be to only let word sizes of registers to be set rather than an arbitrary number of bytes.

- Set memory address: A function to set memory to a given byte, at a given address. This will interact with the MemorySpace object of the current context, performing any preconditions required before the procedure takes place. It will also ensure that the current context is always the context used to make the changes. For example, the function will be called externally as ControlUnit.SetMemory(address,data), not MyContext.SetMemory(address,data), so there is less margin for error here.
- Fetch memory address: A function to fetch an arbitrary length of bytes specified in the parameters from the memory of the current context. This will extend functionality of the MemorySpace data type allowing more than one byte to be fetched at once, allowing for less code repetition in the processing layer as often more than one byte will have to be read.
- Fetch next byte at IP:



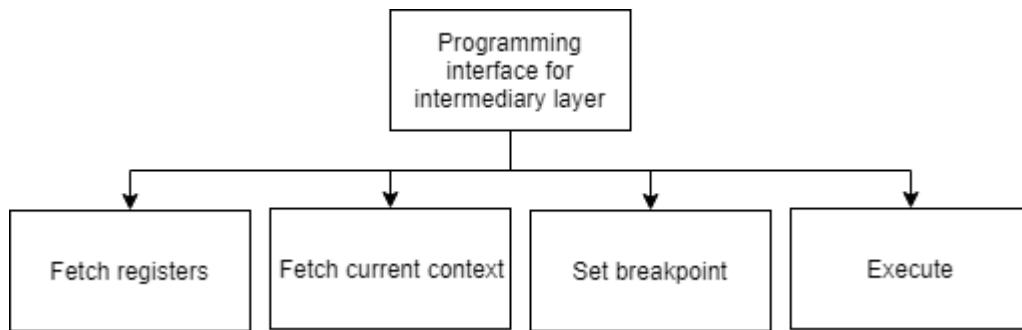
This procedure will make use of the existing fetch memory procedure but provide some procedures that would otherwise be repeated often to be performed at once in the function, which will in turn reduce code size. This includes fetching only the next byte at the IP(as is generally what is required when fetching at the instruction pointer), then incrementing the instruction pointer to point to the next address. This would be a procedure repeated very often in the code without the function, for example when reading the next instructions. For this reason, it is best to have its own method.

- Jump: This procedure will allow a module change the flow of execution by changing the instruction pointer to a specified value. This will allow the cases where the instruction pointer is set and where it is fetched to be separated, as different approaches may be needed when performing the tasks such as the jump call being ignored under specific conditions where a jump should not be taken, for example to an invalid address.
- Set flags: A procedure will be programmed to perform the procedure of setting the flags to a new value. This will make use of the FlagSet.Overlap() method, however an important point to consider is that the FlagSet is a struct, therefore requires to be assigned a new value every time as it has no reference, which would be an easy mistake to make when programming, therefore it is necessary that the method is provided to simplify this process whilst still having the advantages of using a struct.

Justification

It is necessary to solve this problem in the CU as it will require access to the private variables and functions that are restricted from outside callers as they may have potentially erroneous effects if used incorrectly, or require certain preconditions to be met before they can be executed that can be solved with a procedure in the api that will make sure this condition is always met to avoid future bugs when the developer forgets to meet these preconditions and code repetition in the meeting of them.

Intermediary layer api



Explanation

The intermediary layer api will provide functions that are to be used by the intermediary layer such as repeated procedures identified through the stepwise refinement and to output variables that external callers do not have direct access to.

Justification

This will be necessary in order to provide the simplified form of outputs that the intermediary will then parse into a more readable format that can be used in the interface layer through pipelining. There are still some details that need to be abstracted in the processing layer and not pipelined to the intermediary layer, which can be performed in this procedure. This would include references to objects that could then be modified by the interface layer and have unintended consequences on the processing layer.

Fetch context

Explanation

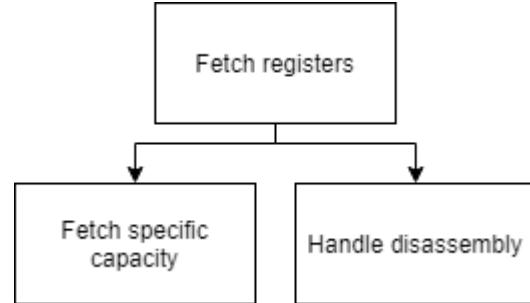
The fetch context method will allow the intermediary layer to have direct access to the context when needed, such that extra functionality can be implemented to introduce features that are not strictly x86-64. For example, this could include setting breakpoints in the user's assembly code.

Justification

It is necessary that this task is performed separately as there are some circumstances where the caller will not want to modify the memory as it will result in unintentional modifications to the same reference used by the CU, which would introduce undefined and erroneous behaviour. As a countermeasure, it is necessary to separate this from the other functions, but still include the functionality as the separate fetch context method.

Fetch registers

Explanation

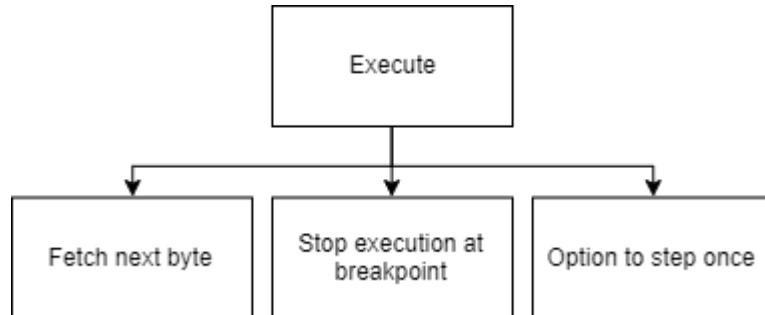


The fetch registers procedure will allow all the registers to be fetched at once and disassembled then returned to the caller. This procedure can be generalised by defining the outputs in the CU rather than returning the raw forms of the registers and parsing it in the intermediary layer. This is because a lot of unnecessary details can be abstracted here, such register codes, which can be replaced with more relevant and less complex identifiers such as mnemonics.

Justification

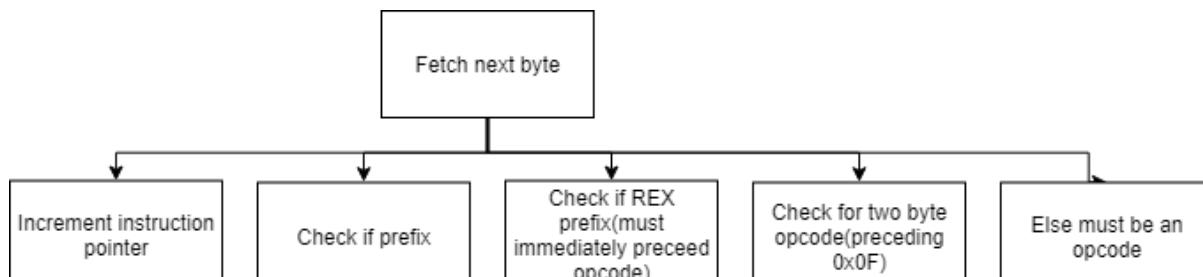
It is necessary to perform this task in the CU and not the intermediary layer because the registers must be fetched and iterated over in this method regardless of whether they are abstracted here, therefore the best approach would be to apply the necessary abstractions and removal of irrelevant details in this iteration rather than having to iterate through the registers again in the intermediary layer, which would introduce code repetition and harder to trace down where the procedure is performed because it would be split across layers and functions.

Execute



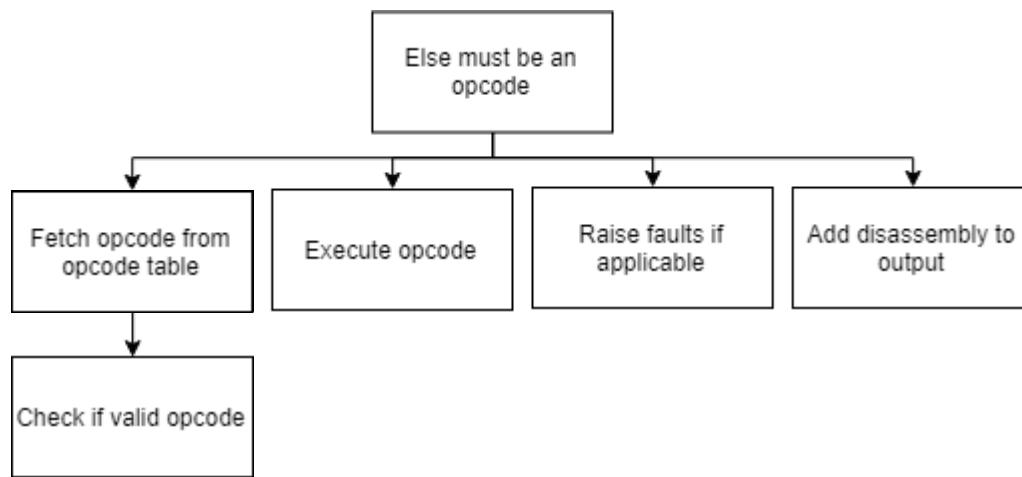
Explanation

The execute method will allow the intermediary layer to begin executing instructions at the instruction pointer. Execution will stop at a breakpoint, or if the step boolean parameter is true, only one instruction will be executed before returning to the caller.



The fetch next byte sub-task will be the most significant part of this problem, therefore has to be broken down into further problems. These sub-problems can be solved with reusable general methods provided by existing classes.

- When the next byte is fetched, the instruction pointer must be incremented such that the next instruction is executed afterwards, not the same one continually. This can be done by reusing the method to Fetch byte at IP method in the CU, which will solve this repeated problem.
- The byte fetched will have to be checked to determine whether it is a prefix or not. The PrefixBuffer will perform this check by comparing it to the comprehensive list of prefix bytes(which is short), then writing the prefix to the buffer if applicable.
- The REX prefix will be checked separately to other prefixes because it requires further operations to deduce the fields set and the implied behaviour. This can be done using a condition to check whether the byte is greater than or equal to 0x40, and less than 0x50 as this range is all covered by rex bytes. The REX byte key variable will then be assigned to the fetched byte and checked by other modules to determine behaviour. To be sure it immediately precedes the opcode, it will be checked after the prefix is checked, then the code that treats the next byte as an opcode will be executed afterwards instead of performing the prefix checks again.
- The 0x0F two byte opcode prefix will be implemented by having an IF condition check if the byte is 0x0F, then if so set a boolean that will tell the next procedure that the opcode is to be fetched from the two byte opcode map, not the main opcode map. This byte will be after the REX byte(as it is part of the opcode), so this check will be performed after the REX byte check.
- If other conditions were not met that indicated otherwise, the byte must be an opcode.



In this case, the opcode table will be called with the parameters being the byte that was fetched and the opcode table to fetch the byte from(defaulting to the main table). The opcode object fetched from the table will then be executed. After this, any results of the opcode will have been set already through the opcode api, so no further interaction with the opcode is required. If there was no mapping of the bytes to an opcode, a #UD fault will be raised, indicating to the user that the instruction was skipped over because it was unrecognised, which could be a result of corruption.

Finally the disassembly returned by the opcode class will be returned back through the pipeline to the interface layer, such that the disassembly(which is an abstraction of the instruction in string format) can be used in the higher layer functions as it is in a primitive data type hence no further abstraction is required.

Justification

The execute task has to be separated from the other components because there are many necessary conditions and preconditions that need to be checked before executing the opcode, as it may not necessarily be an opcode. This is an appropriate place to provide this abstraction because it necessary for neither the intermediary layer nor the interface layer to understand the specific conditions in which this would be the case, therefore by handling this procedure in the processing layer, irrelevant details can be abstracted from the upper layers so they can focus on the important outputs of the execution.

Set breakpoint

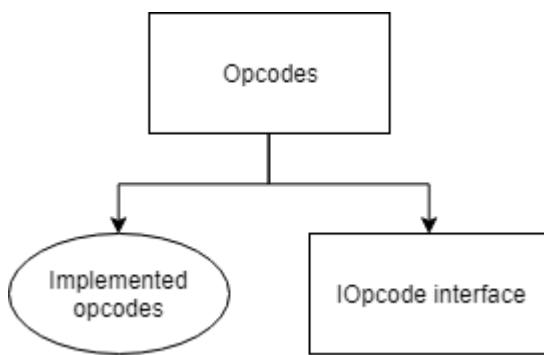
Explanation

The set breakpoint method will allow the intermediary layer to set a breakpoint in the user's assembly code, such that the execute method will return once the breakpoint is met(if step mode is not enabled).

Justification

It is necessary to provide this as a separate function as the interface layer will need to set breakpoints very often. For this reason, the intermediary layer should be given a simple way to allow this functionality. By coding this in the processing layer where there is more control of the CU, less net work has to be done in the intermediary layer.

Opcodes



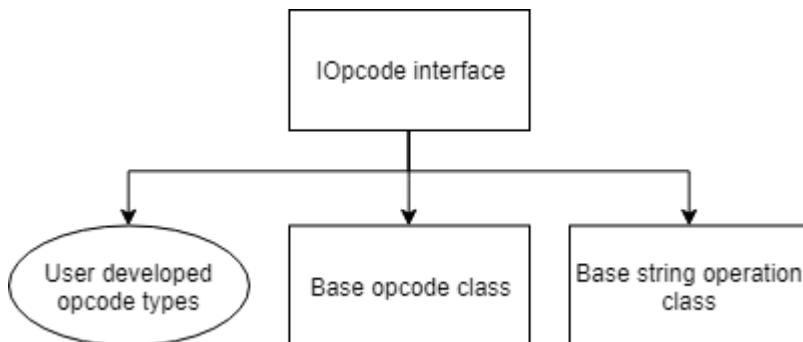
Explanation

Opcodes are assembly level instructions. Each opcode will work different to another. For this reason, opcodes will have their own classes. Opcodes can also have variants which are not seen to the assembly programmer. For example, one variant of MOV will move bytes only, and the other will move words, dwords, and qwords, but both are shown as "MOV". To apply this same abstraction at a development level, each opcode will be generalised to work with any variant using one single approach. This can be achieved because the high level language of C# abstracts these details from the programmer. This will be done by separating the opcodes from the operand types, such that the operands always output a general result instead of the opcodes decoding the operands in the Opcode class, which would require a lot of code to be repeated. Opcodes will implement the IOpcode interface that will have methods for interacting with the opcode, such as fetching it or disassembling it.

Justification

It is necessary to handle the opcodes in separate classes in order to abstract the procedural details of what the opcode performs from the CU, such that every opcode will be another IOpcode interface that can have the same execution procedure applied to it as any other opcode. This will make the problem more suitable to a computational approach because specific algorithms for special cases do not have to be designed which further allows modularity in this aspect of the solution as opcodes can easily be added or removed in the future.

IOpcode Interface



Explanation

The IOpcode interface will generalise the inputs and outputs to opcode such that the procedure in the CU that execute and disassembly the opcodes can work with a general solution rather than having specific procedures for each opcode.

Justification

It is important to have this interface as part of the Liskov substitution principle success criterion. This will allow for a more effective computational approach as each opcode can be treated as a generic subtype instead of specific class. One property in common is the IOpcode.Execute() method. Because of this, the CU will be able to use an algorithm that calls FetchedOpcode.Execute() as a general approach to executing any opcode and have the opcode handle its own procedures in its class constructor to meet any required preconditions before the execute method is called. This is a more suitable computable problem as it can be handled through means of computational methods, such as iteration because the opcodes can be executed in a single loop rather than having special conditions for each type. This will also allow any modifications to the process of executing opcodes very simple because the only dependence is on the interface, therefore it will be easier to perform any code maintenance tasks such as refactoring and optimisation on this part of the solution post development.

User developed opcode types

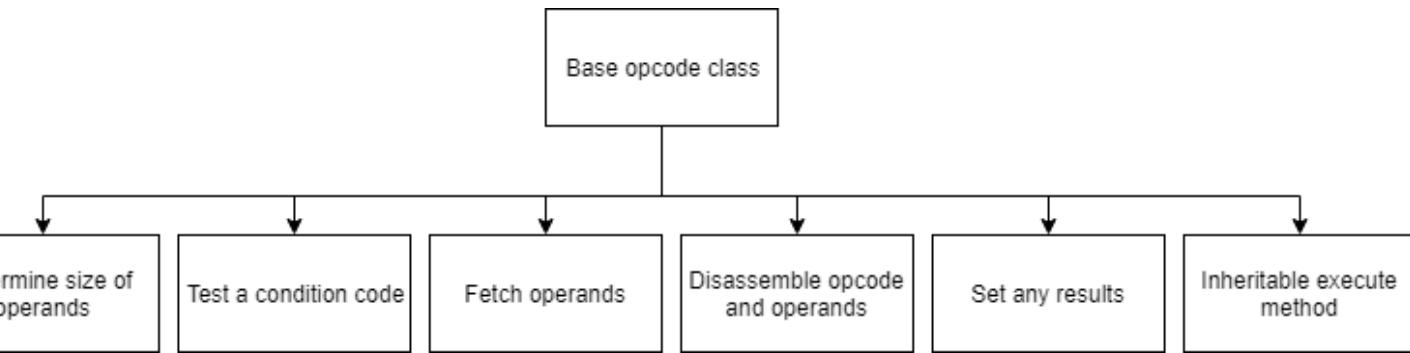
Explanation

The code will include simple ways for users to extend the functionality of the code to meet any specific requirements they have which are not included as part of the scope of the project. For example, they want to create their own opcode. This will be made simple through the use of the IOpcode interface, which they will be able to implement in their own code and have it processed by the CU using the same execution procedures that are called on the included opcodes without any further modification required.

Justification

This is a necessary problem to tackle as a separate module as it addresses the Open/Closed principle software criterion. By including this interface, the code will be open for extension by writing new opcodes that use the interface, and also removing any need for modification of existing code as the general methods are applied to the IOpcode interface, not a specific type of opcode, which will as a result reduce the amount of code that has to be written and maintained.

Base opcode class



Explanation

The base opcode class will be base class of all opcodes included in the project, including many useful solutions to reoccurring problems throughout the code that would otherwise have to be coded on an individual basis. This base class will implement the IOpcode interface such that the derived classes can be processed with general solutions.

Justification

This class is an important part of the solution as it will reduce the amount of code that has to be written and tested. This will allow for a more effective computational approach in the individual opcode classes as code will only have to be written once as opposed to writing for every individual opcode class as general procedures will be provided for each opcode, which will save development time and simplify the process of implementing new opcodes throughout development.

Determine size of operands

Explanation

The opcode class will contain a method for determining the capacity of the operands for the opcode by checking certain conditions of the CU. The default size is a DWORD, and there are 3 other cases. If there is a REX.W byte present, the operands will be QWORDS. This can be checked by checking the REX key variable in the CU with a condition that will be true if the W field is set. The second case is when there is a SIZEOVR prefix present in the PrefixBuffer, the operands will be WORDs. This can be tested by having a condition that is only met if there is a SIZEOVR prefix in the PrefixBuffer. The third case, if the opcode was an opcode that handles bytes only(implied by the opcode variant used), the operands will always be a byte. There will also be an option for opcodes to override this procedure and use their own, or specify a certain capacity. This is because not every opcode follows this procedure, but the majority do, so a general solution is appropriate.

Justification

It is necessary to solve this problem separately because it allows for a more suitable computational approach. This is because an algorithm is applied to the problem to determine the size of the operands, which is kept in the opcode base class because most of the opcodes will inherit from it. This means that the code will not have to be rewritten for each opcode, as they will use the general solution as a default. Also, opcodes that do not follow this general solution can use other methods to specify the operand size through a constructor parameter.

Test a condition code

Explanation

Many opcodes will need to test conditions based on the x86-64 specification. This includes checking if the CF is set, checking if the OF is set, or specific combinations of others. To solve this problem, a procedure will be inherited by the

derived opcode classes that allows any valid condition to be tested using a set of predefined rules and conditions to determine whether the condition is met, and then returned as a boolean.

Justification

It is necessary to tackle this problem separately in the base opcode class because it is more suited to a computational approach. This is because there are a large amount of conditions available that the opcode could use which could not be processed manually. This would otherwise mean that each opcode class that needs to test a condition would have to repeat the same code used by other classes in order to do so, which would increase code size hence be harder to maintain if a new procedure for testing conditions were to be implemented.

Fetch Operands

Explanation

Interactions with operands will be simplified through procedures included in the base class. The Fetch() method will allow opcodes to call the method and have all the operands returned in byte[] format, where any necessary preconditions or parsing is performed by the base class instead of having to repeat the procedures to do so across the solution.

Justification

It is necessary to handle this task separately because it is more suited to a computational approach. This is because it uses the IDecoded interface, which abstracts specific operand types from each opcode and will instead use iteration to loop through all the operands and handle every subtype in the same way. This is important because the classes will depend on the methods provided in the base class, which will reduce their dependence on other modules as if any change is required, it can be made to the opcode base class instead of having to update and test every opcode afterwards.

Set operands

Explanation

Another interact with operand that can be simplified is to set the operands to a value provided by the opcode class. This is a means of storing memory and registers using a general approach, as the operands will contain the source, destination, or both(depends on opcode), which the IDecoded interface then allows the reverse operation to be applied where the operand class will handle what necessary procedures are to be performed in order to do so.

Justification

This is a necessary problem to solve in the base class because it will allow the IDecoded class to be used to create a more suitable computational solution to be used in this part of the solution in order to abstract the need for individual opcode classes to understand how and where results should be stored, which instead will be handled by the operand class. This is the best approach because the operand class will contain all the details that were abstracted by the IDecoded interface, which will be necessary to store the result. As a result of this, the opcode can use one general method to set the source operand rather than have to use individual procedures and routines for each operand, which would otherwise require a large amount of code repetition across the opcode classes that would be hard to maintain and update with new code if a new operand were to be introduced or an existing opcode changed during development.

Disassemble opcode and operands

Explanation

The opcode and operands both have separate disassembling procedures which return the full disassembly of the instruction. The opcode base class will handle the process of combining the two and handling any required preconditions before doing so. For example, the operand must have a size set before disassembling, because otherwise the disassembly would be incomplete. The procedure will make sure that this precondition is met before the disassembly function proceeds.

Justification

It is necessary to solve this problem separately to the existing methods because it allows a simpler computational approach to be taken. This is because the operand and opcode modules are entirely abstracted from each other; the operand does not know which opcode it serves, and the opcode does not depend on the operand. For this reason, a computational solution is appropriate as it will tie the two together without losing the benefits of having the modules self-contained as only the opcode base class will have to change if modifications are made rather than all operand and all opcode classes. Therefore, the solution will be easier to maintain in the future and no design features will be compromised by using a general approach, as would be required if the individual classes were adapted.

Inheritable execute method

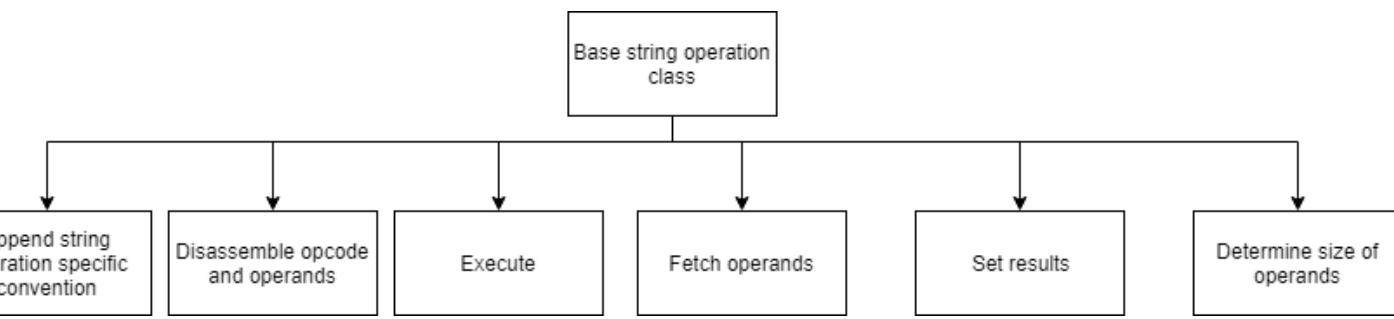
Explanation

In order to allow derived opcode classes to implement their unique functionality, the execute method required by the IOpcode interface will be left as an abstract method. In C#, this means that the derived class must implement this method, rather than defining it in the base class.

Justification

This is a necessary task required to solve the problem because each opcode will need to implement its own code. By providing a method for all the classes to override and implement their function in their own way, a computational approach can be used in other parts of the solution when handling the opcodes as they can be interacted with by using the methods required by the IOpcode interface. For example, it will be possible to use iteration to loop through an array of Opcode objects and call their execute method instead of having to address each specific class individually. This is important because otherwise every opcode would be the same, they would not have a way to execute their own code. This will be possible as the execute method that made an abstract method will be the same method that is used by interface methods, such that the responsibility of implementing this method is passed down to the derived class rather than having to define it in the base class.

String operation class



Explanation

The string operation class will be another base class for opcodes that still implements the IOpcode interface but provide a different feature set used by string operation base class, as they require procedures different the procedures included in the opcode class.

Justification

This base class is necessary as part of the interface segregation principle success criterion. The string operation derived classes will not depend on the methods in the opcode base class, so will not be forced to use them. Instead of squeezing the relevant methods in the opcode base class, a separate base class will be used that provides methods tailored specifically to the relevant class. This will allow for more suitable computational solutions to be implemented by breaking down the problem into subproblems and tackling each separately rather than reusing the methods that are not relevant to this class.

Determine size of operands

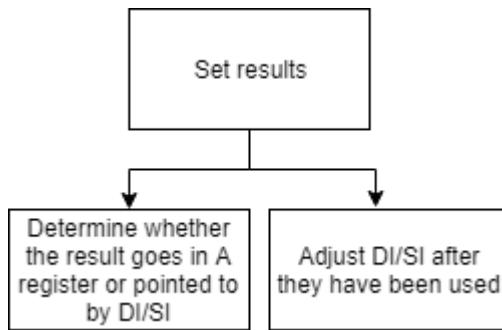
Explanation

The conditions for the size of operands is the same for string operations as for other opcodes. However, there will be specific changes to the mnemonic of the opcode based on the size and instead of having the optional parameter to specify the size, string operations always use the general procedure to determine the size.

Justification

It is necessary to tackle this problem separate to the similar problem in the opcode base class because it will allow for a more suitable computational approach in the algorithm as there are extra steps that need to be taken in the string operation class that do not need to be taken in the opcode base class. If there were to be a general method, the conditions to check the size would have to be repeated regardless. Therefore by having the two methods distinct, the conditions do not need to be checked twice in the string operation class and it will be easier to read the code when the short selection process for each class is in one place rather than split across functions.

Set results



The procedure for setting results is different to the procedure of setting results in the base opcode class. String operations have no explicit operands, it is always implied by the opcode. For this reason, instead of using general procedures from the IDecoded class, code has to be written to define the procedures of what will happen when data is set. The source of the destination operand is either DI or A, both can be of any size. This will be specified in the constructor by the opcode class and stored in a variable. The variable will be read by the set procedure. If the destination is DI, it is a pointer, so must be set in the memory, otherwise it is a register. This can be checked by a condition and the appropriate opcode api method in the CU will be used. Another task that must be performed in this case is that the DI register(whichever applicable) will be incremented/decremented by the length of bytes set. Whether this value is added or subtracted is based on the state of the direction flag. See pseudocode.

```

REGCODE DestinationCode
VAR Destination;
DEF Set(byte[] data)
    // Check if the destination is to be used as a pointer or not.
    IF DestinationCode == RegCode.DI THEN
        // Cast the variable to a ulong because it will be used as a pointer
        CU.SetMemory((ulong)Destination, data);

        // Increment/decrement the DI register by the number of bytes used.
        // If the DF is set, the register will be decremented, otherwise incremented
        IF CU.DF == FlagState.ON THEN
            DecrementDI(data.Length);
        ELSE
            IncrementDI(data.Length);
        FI;
    ELSE
        // In any other case it is not a pointer, but used as a register
        RegisterHandle.Set(DestinationCode, data);
    FI;
    RETURN;
FED;

```

This pseudocode shows the procedure of checking whether the destination will be used as a pointer, or the actual value in the register changed. It is a complete solution to this problem because when the data is to be set, the destination will be a pointer if and only if the destination code is DI. In any other case, the value of the register is changed.

Justification

It is necessary to solve this problem separately to the opcode base class because the procedures required cannot be included in the existing code without changing the behaviour of the opcode class. As there are only a small number of string operation opcodes, the best approach is to create this separate function such that when the code needs to be improved or a bug fixed, it will be a lot easier to do so without having to otherwise navigate through a god function that serves more purposes than a function should.

Fetch operands

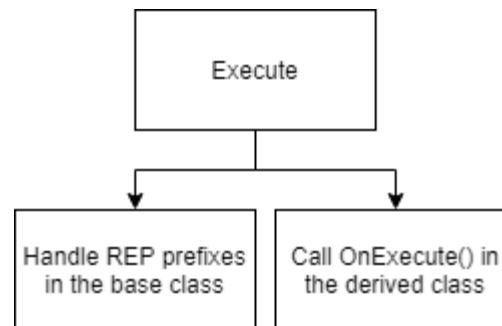
Explanation

Operands are either the A register, or DI/SI pointers. This means that if the A register is used, the value of the register will be returned to the caller. If the register is DI or SI, the value at the memory location pointed to be the address in the register will be returned. This will be performed by checking a variable to see which register is used, then using the applicable opcode api method.

Justification

It is necessary to tackle this as a separate subproblem because it will allow for a specific computational solution to be applied to the problem that is relevant to this class rather than to reuse the solution in the opcode base class. This is because the existing procedure in the opcode base class cannot be reused for this problem. This is because it does not automatically resolve pointers in the DI/SI registers. This adaptation is not possible in the opcode base class either because other opcodes do not have this same behaviour when fetching the DI/SI registers. For this reason, the best approach is to have two separate functions.

Execute



Explanation

The execute method has multiple responsibilities. Unlike the opcode base class method, it will not be an abstract method. The derived classes will instead implement another abstract method, OnExecute(). This is because the base class needs to handle the REP prefix, which cannot be done if it does not have control of the flow of execution in the Execute() method that is called externally. This is because when a REP prefix is present, the opcode is to be repeated until the ECX register is zero. To do this, OnExecute will be called every iteration then ECX decremented. The REP prefix can be checked by examining the PrefixBuffer in the CU.

Justification

This is the best approach to solving this problem because it allows the base class to still implement the IOpcde interface such that it will be processed by the CU in the same way as opcode base class would. This means that the same computational algorithms and solutions in other parts of the program can still be used to interact with the class in the same way that the opcode base class is interacted with and still allows the REP prefix functionality to be implemented in a way that does not depend on the derived opcode class. This means that the REP prefix will work for any string operation, which will be useful when implementing new opcodes; less code will be required to do so as the base class will handle the problems common to all string operations. Moreover, the class will be able to be operated on through computational methods in other parts of the solution because it will be accessed as part of an interface rather than a specific class, such that the need for class specific interactions has been abstracted and instead, more suitable approaches such as iteration can be used to call the method instead of otherwise having to use condition statements to determine whether an opcode is a string operation or not, which would be inefficient and hard to maintain.

Disassembly and applying convention

Explanation

There are differences in the disassembly of string operations and the disassembly of any other opcode. On the end of the mnemonic of a string operation, the size of the implied operands is appended at the end. For example, movsd is the mnemonic for move string dword, movsq is the mnemonic for move string qword, etc. Another convention is that unlike any other prefix, the REP prefix is always shown at the beginning of the mnemonic. This means that the disassembly method will require extra procedures in order to form a complete solution to the task. These procedures will check conditions to determine whether the REP prefix is present, and to check the size of the operand, then apply necessary changes to the disassembly.

Justification

It is necessary to tackle this problem in the string operations base class rather than have a general solution applicable to both the string operations base class and the opcode base class because there are many steps and conditions that must be checked in the string operation disassembly that do not have to be checked in the opcode disassembly, and the opcode disassembly procedure also calls the IDecoded operand object to retrieve the disassembly; string operations to not use the IDecoded class. For this reason, the best approach is to have two separate methods in order to avoid bloat in a function as it would unnecessarily serve many different purposes, which would be harder to maintain and track down bugs in.

Implemented opcodes

Explanation

Opcodes will be included as part of the solution such that the program has functionality once production ready. Features for users to implement their own opcodes will be entirely optional and in most cases unused. Each opcode will have its own class and implement the IOpcode interface, or inherit from a class that does. They will use the utility libraries, opcode api, and base class methods, in order to carry out their procedure in a simple manner and save development time by using reusable solutions to recurrent problems.

Justification

It is necessary to have the opcodes in separate classes in order to include modularity, as opcodes will easily be able to be added, removed, and modified, during and after development without affecting other parts of the program save an incorrect result. This links to the single responsibility principle success criterion. Each opcode class will perform the specific actions of that opcode, for that purpose only. Because of this, they will not need to be changed post development aside from refactoring, as instead of a new class requiring the opcode class code to change, the new class will be able to adapt to the existing opcode class inputs.

Post development test data

Assembly criteria

Name	Type	Size
_Example.xml	XML File	1 KB
Adcadd.xml	XML File	3 KB
Bitshifts.xml	XML File	2 KB
Cbw.xml	XML File	1 KB
Div.xml	XML File	1 KB
Imul.xml	XML File	2 KB
Jmptestcmp.xml	XML File	1 KB
Lea.xml	XML File	1 KB
Mov.xml	XML File	2 KB
Mul.xml	XML File	2 KB
Negate.xml	XML File	1 KB
Or.xml	XML File	2 KB
Pushpop.xml	XML File	2 KB
Retcall.xml	XML File	1 KB
Rotate.xml	XML File	4 KB
Sbbsub.xml	XML File	2 KB
Setbyte.xml	XML File	1 KB
SIB.xml	XML File	1 KB
stringops.xml	XML File	3 KB
Xchg.xml	XML File	1 KB

Explanation

Twenty(excluding example) testcases have been designed to test the functionality of the processing layer. These tests will be executed by the TestHandler hypervisor class which will carry out the testing procedures; comparing found results with expected results, and will programmatically determine whether the tests were successful. At the end of development, all these tests should pass.

Justification

It is necessary to use this test data to test these criteria as it is designed to test the assembly functionality of the solution. This will used to measure the relevant 1:1 emulation criteria. This test data is appropriate as it will allow the criteria to be measured systematically and quickly such that rules are concrete; the testcases follow the specification by letter, so it is ensured that the criteria will be measured properly.

VMCS Post-development survey

(Please download and run VMCS.exe attached in the email) Did you participate in the previous VMCS active research survey?

Yes

No

(Compared to the alternative software reviewed previously), how intuitive was the design?

1 2 3 4 5

Not intuitive Very intuitive

How much information displayed did you consider to be excessive or irrelevant?

1 2 3 4 5

None A lot

How often did you have to interact with the UI unnecessarily?

1 2 3 4 5

Never Very often

How simple was it to access core functionality such as loading a program

1 2 3 4 5

Very hard Very simple

Which of the following statements would you use to describe the UI?

- Intuitive, natural, and relaxed.
- Not serious enough, unprofessional
- Indifferent

SUBMIT

Explanation

A survey has been designed to measure the success of the design success criteria. This will be given to the same stakeholders who took part in the initial stakeholder research that took place in the analysis stage. Each question in the survey is aimed at a particular success criterion, for example, the first question is aimed at the “User interface must be intuitive.”

Justification

It is appropriate to use a survey as test data as it uses quantitative data to represent opinions. This will give a clearer view in the evaluation on the stakeholders opinions when the project is released, as an opinion such as “It was good” is not informative enough to determine the success of the criteria, whereas a scale will clearly show which criteria were met and which were not.

Design criteria

(Please open the github link provided in the email) How much programming experience would you say you have?

	1	2	3	4	5	
None	<input type="radio"/>	A lot				

(Open the "SRP" link in the email, read the information, and click the button to be directed to a random class file) Read the class file and briefly describe its purpose here and explain one method.

Your answer

(Open the "Open/Closed" link in the email, read, button) Select the box that best describes the class you read

	Polymorphism was used to extend the class, there were additional classes that expanded on the class without changing its code	Polymorphism was used to extend the class, but required code to be modified in the base class to fit certain situations	Polymorphism was not used to extend the class
The class fully supported polymorphism features such as inheritance and could easily be derived from and adapted.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The class moderately supported polymorphism features such as inheritance and could quite easily be derived from and adapted.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The class partially supported polymorphism features such as inheritance and could be derived from and adapted.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The class did not support polymorphism features such as inheritance and could not be derived from and adapted.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

(Open the "LSP" link) Identify up to three places in the class where an interface or base class has been used as an input to a function rather than a specific class(excluding private methods)

Your answer

(If you are not an experienced OO programmer, please skip this question) Please take time to read the source code and leave a comment about the extent of the Interface Segregation Principle used in the solution,

Your answer

At the bottom of the email, there is a selection of class files to choose from. Choose the one that sounds like you would have the most expertise. Write a short comment about the purpose and actions performed by that class, its name, and any parts you did not understand.

Your answer

Explanation

A survey has been designed to measure to what extend the design success criteria have been met. This survey will be sent to professional/competent high level programmers I know to evaluate the design patterns and structures used throughout the code.

Justification

This test data is suitable as it will measure multiple criteria in a way that value feedback can be received and used to improve the program post-development to meet any stakeholder needs that were not met through development. In this case, a qualitative survey is most appropriate because there are too many aspects and depth required to evaluate a response in this context, expert opinions on design are needed which go into more detail than opinions on user interface(e.g.). It is also suitable because it tests all aspects of the criteria. It will be possible to reliably evaluate each success criteria based off the results of the survey.

Username password dialog

Explanation

There will be a username/password dialog present in the program. It will activate after a set time, locking all access to the program. This will be useful as the user may leave their desk with important code open being debugged inside VMCS, hence will prevent anyone from seeing the code. The username and password will be chosen by the user when they open the program, lasting for that session.

The dialog could be as follows,

```
DEF InputLogin()
    OUTPUT "Enter username and password";
    INPUT username,password;
    PresenceCheck(username,password)
```

```

LengthCheck(username,password)
..
// Call all validation routines
..
// Save username and password so they can be checked again later in program
Save(username,password);
FED;

```

Validation

These validation routines will be carried out when the user is inputting their username and password details.

Length check

The username and password must be shorter than 8 characters. This is necessary to reduce the chances of the user forgetting their details and losing their work. The short password length not a security risk as the user will choose a new password frequently, and there are no online systems, so no data is retained when the user closes the program.

Pseudocode

```

DEF LengthCheck(password,username)
    IF password.Length > 8 OR username.Length > 8 THEN
        OUTPUT "Username and password must be shorter than 8 characters";
    ELSE
        // Proceed with storing details and other validation routines
    FI;
FED;

```

Type check

The username and password must both be strings only. This will prevent any potential character bugs with different type sets that could lock the user out the program or cause a crash.

Pseudocode

```

DEF TypeCheck(password,username)
    FOR character in password
        // Check if any characters are not letters
        IF character.IsAlphabetic == FALSE;
            OUTPUT "Password must contain only letters";
            InputLogin(); // Re-open input box
        ROF;
    FOR character in username
        IF character.IsAlphabetic == FALSE;
            OUTPUT "Username must contain only letters";
            InputLogin(); // Re-open input box
        ROF;
    // Proceed with storing details and other validation routines
FED;

```

Presence check

A presence check will be required to ensure there are no errors in other validation routines or in the program due to the lack of input.

Pseudocode

```

DEF PresenceCheck(password,username)
    // Check if username and password are empty strings
    IF password == "" OR username = ""
        InputLogin(); // Re-open input box. Assume user accidentally pressed enter.
    FI;
FED

```

Key variables

Name	Justification
CU.CurrentHandle	Stores reference to current handle so other modules can interact with current context
CU.CurrentContext	A shallow copy of the current context, allowing the actual context to be modified directly.

CU.Flags	The abstraction of the flags register, stores individual values for each flag
CU.IP	Instruction pointer, stores address of next instruction to be executed
CU.RexByte	Stores the current rex byte, allowing low level abstractions to determine what actions are performed(e.g. ModRM size is determined by the REX byte).
Handle.HandleName	A string interpretation of a handle name specified in the handle constructor call, allows for simple debugging
Handle.HandleID	An integer unique to each handle in the current execution of VMCS. Handles are managed programmatically by this variable, not the string name.
ModRM.Destination	Address of the ModRM destination
ModRM.Source	A register handle, representing the source register of the ModRM
VM.Breakpoints	List of all breakpoint addresses
Opcode.OpcodeTable	Dictionary mapping opcode byte to opcode class
Opcode.OpcodeSettings	An enum with each opcode setting, e.g. signedness
Opcode.Capacity	A WORD size that the opcode is operating in, used by subsequent IDecoded objects
Opcode.Input	IDecoded object responsible for handling input of opcode.
Context.Flags	Structure of flags in context. Cloned by CU when used.
Context.Memory	MemorySpace abstraction, storing all of the data of the instance represented by the context.
Context.Registers	RegisterSpace abstraction that holds all of the register values of the context.
Context.InstructionPointer	Stores address of next instruction to be executed. Cloned by CU when used.
StringOperation.SrcPtr	Value of RSI register copied from current context, which is saved back into the register when the opcode finishes executing.
StringOperation.DestPtr	Value of RDI register copied from current context, which is saved back into the register when the opcode finishes executing.

Development

Introduction of overall strategy to develop system

To develop the project, an iterative development cycle will be used. A feature will be added in a version, then reviewed by stakeholders. If the stakeholders are satisfied with the feature, there will be no requirement to change the feature. When a change is needed, it will be made accordingly in the next version. This will rarely be the case because research was done in the analysis and design sections to ensure that each feature is meeting the needs of stakeholders. A similar procedure will be used for tests. If a test fails, remedial action will be taken immediately after the test and be in effect in the release of the version in which the bug was found, instead of waiting for the next version. This is important because bugs are specific problems in with the program, not with the design, so it is important that the stakeholders have functioning versions of the program to review, as it could otherwise bias how they feel about the feature. For example, they may think that a design feature needs to change, but really the part they did not like was because of a bug.

Alpha -> Beta -> Developer preview -> Release candidate

A software release cycle will be used to structure the development process. The first version will be the alpha version which will include the first set of tests. These tests will be done by myself pretending to be a user and otherwise with stakeholders using black box techniques(trying features without considering how they work). This version will include very basic features, but allow something working to be seen early in the development. The structure of the program is likely to change in later versions to adapt to accommodate new features. The beta version will expand on the features of the alpha version and also include new features. The program will be mostly feature complete but may have a lot of bugs. Because of this, there will also be testing done in this stage. This testing will be similar to alpha testing, but will be more focused on covering a large number of tests planned in the design section and making sure they are working, otherwise taking any remedial actions needed. The next version will be the developer preview. This will include a fully functioning base set of features that will be demoed to a group of stakeholders. This version will focus on making sure the feature set is as complete as possible so the stakeholders get a full picture of the project's capabilities, however this is subject to time constraints. In this version, less focus will be on UI and accessibility features. The final version will be the release candidate. This will be the production release of the program. It will mostly focus on final touches and is unlikely to include major features. At this point the project will be considered complete and afterwards evaluated.

Alpha version

Date: 4th July 2019

Date: 18 July 2013

This version includes the CU coded to a basic level with a few opcodes

This version includes **OpcodesTable**

```
// Assign all opcodes to the opcode table. This associates their byte that would be read with
// the opcode that they are decoded to.
OpcodeTable.Add(1, new Dictionary<byte, Action>()
{
    { 0x00, () => new Add8().Execute() },
    { 0x01, () => new Add32().Execute() },
    { 0x02, () => new Add8(Swap:true).Execute() },
```

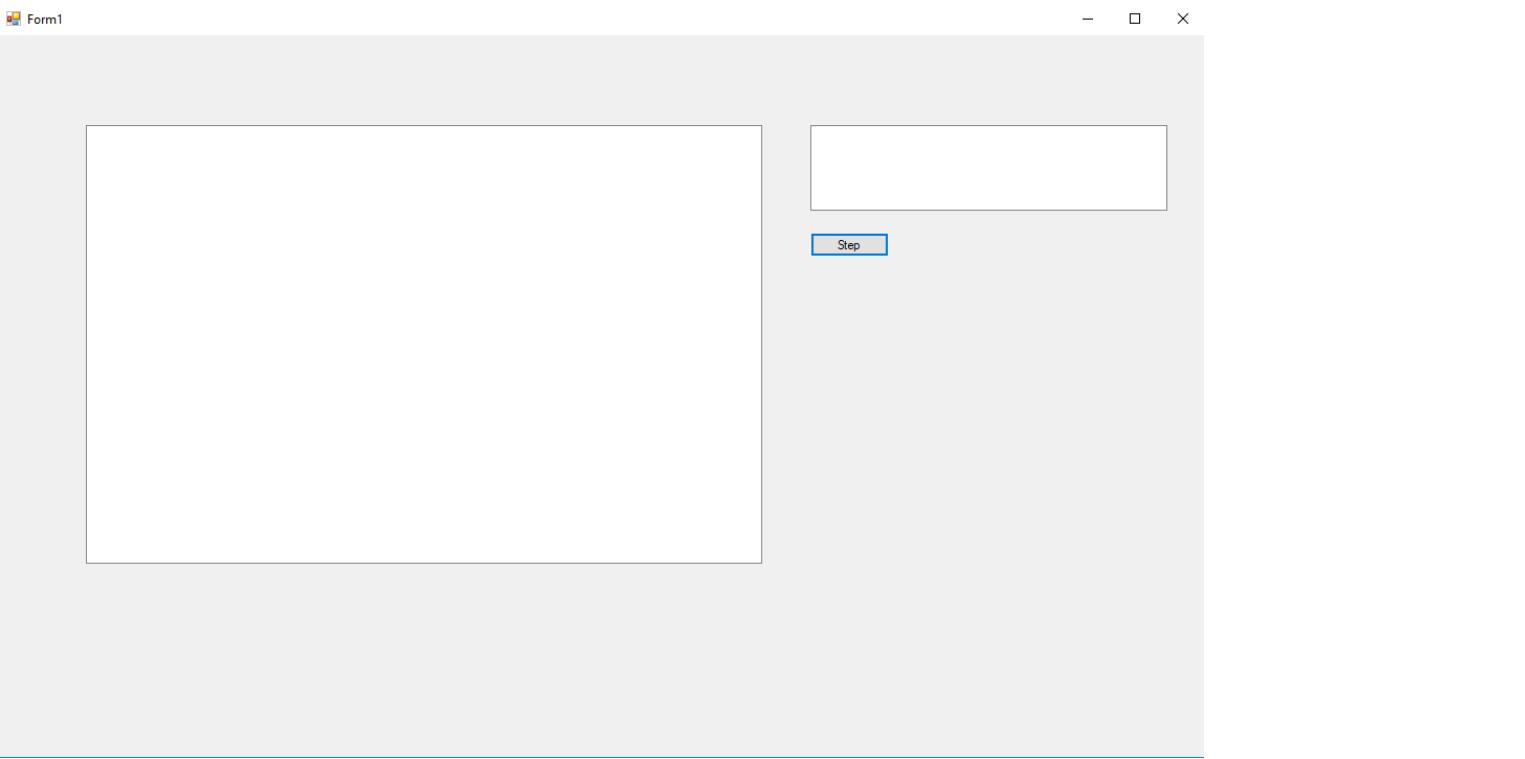
Explanation

The OpcodeTable discussed in the Design section has been implemented to a basic level. It uses the dictionary as planned to translate a byte to the opcode associated with it, which is then executed. The implied parameters are included in the opcode table. The dictionary is a member of another dictionary which holds all opcode tables. This will be used for the 2 byte opcode table and any other opcode tables to be included such as the extended opcode tables. This is performed by using a delegate method, which executes automatically once fetched.

Justification & Relation to break down of the problem

The best approach to this problem at this point of time was to use the dictionary as planned and include a small number of opcodes. This is because the functionality of the CU has to catch up before more opcodes can be coded. If too many opcodes were coded initially, the opcodes will be dependent on their own methods rather than using the opcode api procedures that have not been implemented yet, which would cause them to use outdated functions rather than the latest refactored and optimised functions in the opcode api that will have the most attention during development. This has also allowed me to think ahead and set up the design to use nested dictionaries to include multiple opcode tables in the same variable. This has allowed the 2 byte opcode table to be set up ready for when two byte opcodes are added.

User interface



Explanation

The user interface is work in progress. Forms are being designed. Currently there is only a step button which is used for debugging purposes. The groundwork for the user interface features such as the disassembly viewer and memory viewer are currently work in progress. The utility library is still being developed to implement these.

Justification & Relation to break down of the problem

It is necessary to focus on the processing layer and allow the UI to lag behind slightly because there will be many changes throughout the development to the inputs and outputs of functions and how certain features will be implemented is not yet clear as there are many modules in the utility library that are still in development phase. For this reason, the best approach is to create a robust processing layer to maximise functionality, develop code structure, and code the necessary data structures and abstractions that will be used to communicate with the interface layer. This will save development time as I will not have to code the interface multiple times to work with new versions of functions; the functions that the interface layer was shown to depend on in the design section will first be created and tested before the functional interface layer is implemented.

Registers

```
// Store each 64 bit register as a variable that can be accessed from any class(hence public).
public static Register64 RIP = 0x0000000000000000;
public static Register64 RAX = 0x0000000000000000;
public static Register64 RBX = 0x0000000000000000;
public static Register64 RCX = 0x0000000000000000;
public static Register64 RDX = 0x0000000000000000;
public static Register64 RSP = 0x0000000000000000;
public static Register64 RBP = 0x0000000000000000;
public static Register64 RSI = 0x0000000000000000;
public static Register64 RDI = 0x0000000000000000;

72 references
public static byte[] FetchRegister(ByteCode ByteCode, RegisterCapacity WorkingBits)
{
    // Switch the register capacity, which defines which size of register will be fetched
    switch (WorkingBits)
    {
        // Check if it is a WORD register
        case RegisterCapacity.X:
            // Perform a switch case to fetch the value of the register.
            switch (ByteCode)
            {
                // Check if A register, B register, C register etc.
                case ByteCode.A:
                    // Convert the register to bytes, then take the first two elements of the byte array and return them
                    return BitConverter.GetBytes(AX).Take(2).ToArray();
                case ByteCode.B:
                    return BitConverter.GetBytes(BX).Take(2).ToArray();
                case ByteCode.C:
                    return BitConverter.GetBytes(CX).Take(2).ToArray();
                case ByteCode.D:
                    return BitConverter.GetBytes(DX).Take(2).ToArray();
                case ByteCode.AH:
                    return BitConverter.GetBytes(SP).Take(2).ToArray();
                case ByteCode.BH:
                    return BitConverter.GetBytes(BP).Take(2).ToArray();
                case ByteCode.CH:
                    return BitConverter.GetBytes(DI).Take(2).ToArray();
                case ByteCode.DH:
                    return BitConverter.GetBytes(SI).Take(2).ToArray();
                }
                break;
        // Check if it is a DWORD register
        case RegisterCapacity.E:
            switch (ByteCode)
            {
public class Register64
{
    private ulong lValue;
    1 reference
    public Register64(ulong R)
    {
        // Assign the private value to the parameter passed to the constructor
        lValue = R;
    }

    // Implicitly access the object like a byte array can.
    // E.g., MyRegister64[0] will return the first index of the array returned by
    // the implicit byte array operator.
    1 reference
    public byte this[int i] => this[i];
    public static implicit operator ulong(Register64 R)
    {
        // Allow the class to be used as a ulong by implicitly converting it when accessed as one,
        // e.g.,
        // ulong MyUlong = MyRegister64 + 10;
        // This kind of code will work without any casting as it is implicitly converted to a ulong
        // (which just calls this method and returns the value below).
        return R.lValue;
    }
    public static implicit operator byte[] (Register64 R)
    {
        // Allow the class to be implicitly used as a byte array using the same means as above except
        // converting to a byte array instead of a ulong.
        return BitConverter.GetBytes(R.lValue);
    }

    public static implicit operator Register64(ulong lVal)
    {
        // Allow two Register64 objects to be used like ultongs, e.g.
        // Register64 OutputRegister = InputRegister1 + InputRegister2;
        return new Register64(lVal);
    }
}
```

Explanation

A heuristic approach has been taken to registers initially. Registers can be used as either byte arrays or as ulong values. The “implicit operator byte[]” will automatically convert the value to a byte array when used as one. Registers are currently accessed by exhaustion, they are compared against all combinations of register capacity and register code(called “ByteCode”, as it is their value in byte form). This procedure is temporarily repeated across FetchRegister() and SetRegister() functions.

Justification & Relation to break down of the problem

It is necessary to take a temporary heuristic approach to solving this problem. This is because there is a lot more groundwork in the processing layer that needs to be implemented before the intermediary layer can be coded. For this reason, the best approach is to use less efficient means of registers that is functional but is not as efficient because this could be developed in a very short amount of time, which allowed the problems that need to be tackled first to have more attention during development. This means that instead of having to code a functional and effective register system, the heuristic approach can be used before the intermediary layer is ready and has the necessary functions required to support the RegisterHandle class.

ControlUnit

```
1 reference
private static void _step()
{
    // Function purpose: Perform one iteration of the Von Neumann Fetch Decode Execute cycle.
    // This means that a single instruction will be executed.

    // Set the byte pointer to the instruction pointer
    BytePointer = RIP;

    // Fetch the next byte, stored at the instruction pointer
    byte Fetched = FetchNext();

    // Decode will return false if an instruction was read, otherwise it could have
    // been a prefix. In this case, it should neither be executed as there would be an error,
    // nor the prefix buffer reset as this only happens every instruction(it would reset
    // any prefixes read beforehand).
    if (!_decode(Fetched)) // no op
    {
        _execute(Fetched);
        Prefixes = new List<PrefixBytes>();
    }

    // Update the instruction pointer to the byte pointer
    RIP = BytePointer;
}

1 reference
private static bool _decode(byte Fetched)
{
    // Function purpose: Apply general procedures to a byte to determine whether or not it is
    // a prefix byte. This will change whether the byte is executed like an instruction, or added
    // to the prefix buffer to be used by other modules. Returns true if it is a prefix.

    // If the fetched byte was 0x0F, this is the prefix for a two byte opcode, so the
    // variable containing the size of the current opcode is set to two.
    if (Fetched == 0x0F)
    {
        _opbytes = 2;
    }
    // If the PrefixBytes enum contains a definition for this byte, it must be a prefix.
    else if (Enum.IsDefined(typeof(PrefixBytes), (int)Fetched))
    {
        // Add it to the prefix buffer so the opcodes can see the prefix is set.
        Prefixes.Add((PrefixBytes)Fetched);

        // Return true as this was a prefix.
        return true;
    }

    // If the previous conditions were not met, it must be an opcode.
    return false;
}

1 reference
private static void _execute(byte Fetched)
{
    // Function purpose: Execute the delegate method(that will call the respective opcode class execute method),
    // that the byte is mapped to in the opcode table.
    OpcodeLookup.OpcdeTable[_opbytes][Fetched].Invoke();
}
```

Explanation

The ControlUnit currently follows the fetch decode execute cycle similar to a Von Neumann architecture. The next instruction byte is fetched in the `_step()` method(which is the procedure that performs one cycle), then `_decode()` is called to test the byte against certain predefined conditions that determine whether the byte is a prefix byte. In this case it needs to be added to the prefix buffer and not be executed. If the byte was not a prefix(`_decode()` returned false), it is executed by calling the delegate method associated with that byte in the opcode table.

Justification & Relation to break down of the problem

This is the best approach to implementing the CU at this stage of the development. This is because the methods have been split into separate subproblems and tackled separately, as discussed in the design plan and stepwise refinement, the intermediary api will require a method to allow external callers to interact with the CU, which in turn has allowed certain elements such as handling prefix bytes to be abstracted from the caller. This will allow the procedures to be expanded on and updated without affecting one another as they only depend on their inputs and outputs. This has allowed the basis of the intermediary api to be developed as there is a method to programmatically start execution in the CU so other classes can run their code. This will allow upcoming versions to include prototypes of the intermediary layer as it will have the necessary functions required to start programming it available.

MemorySpace

```
public class MemorySpace
{
    // Internal dictionary to hold address to byte mappings.
    private Dictionary<ulong, byte> _memory = new Dictionary<ulong, byte>();

    // Segment map that can be used by other modules to locate data.
    public Dictionary<string, Segment> SegmentMap = new Dictionary<string, Segment>();

    // Information that other classes can use to perform other operations on a MemorySpace such as
    // find what information is at the first address. This can be done by fetching at the EntryPoint.
    public ulong Size;
    public ulong EntryPoint;
    public ulong LastAddr;

    public class Segment
    {
        // Segment data structure that holds the necessary information about a segment, e.g. where it is and the data it holds.
        // Note that the data byte array wil not be updated with new data as the program executes, it is only used to load the
        // segment data into memory when the memory is first initialised.
        public ulong StartAddr;
        public ulong LastAddr;
        public byte[] Data = null;
    }

    public MemorySpace()
    {
        // Function purpose: Testing constructor to allow the MemorySpace to be created with no additional parameters.
    }
    public MemorySpace(byte[] _rawinputmemory)
    {
        // Function purpose: Create a memory space object from an input of bytes. They will automatically be written to the
        // ".main" segment that can be accessed through the SegmentMap.

        // Initialise the entry point to 0x100000(arbitrary, allows some extra room at low addresses if needed).
        EntryPoint = 0x100000;

        // Initialise some default segments that can be used by other modules to locate the data that they need faster.
        // You can access this information by using the name of the segment as a key in the SegmentMap dictionary,
        // e.g. SegmentMap[".main"].StartAddr; is perfectly valid
        SegmentMap.Add(".main", new Segment());
        { StartAddr = EntryPoint, LastAddr = EntryPoint + (ulong)_rawinputmemory.LongLength, Data = _rawinputmemory };
        SegmentMap.Add(".heap", new Segment() { StartAddr = 0x400001, LastAddr = 0x0 });
        SegmentMap.Add(".stack", new Segment() { StartAddr = 0x800000, LastAddr = 0x0 });

        // Create memory regions for each segment
        foreach (Segment seg in SegmentMap.Values)
        {
            // If the segment has no data assigned to it, only add a single 0 byte to indicate its presence
            // (Removing this condition will result in a NullReferenceException in the next block of code)
            if (seg.Data == null)
            {
                _memory.Add(seg.StartAddr, 0x0);
            }
            else
            {
                // Iterate through every address in the segment data and add it into where the
                // segment lies in memory.
                for (ulong i = 0; i < (seg.LastAddr - seg.StartAddr); i++)
                {
                    _memory.Add(i + seg.StartAddr, seg.Data[i]);
                }
            }
        }
    }
}
```

```

public bool ContainsAddr(ulong Address)
{
    // Function purpose: Check if an address is registered in the internal dictionary.
    // This can be used to safely access the memory space as there could be a KeyNotFoundException
    // otherwise. There are no validation procedures yet because the class could change a lot in the
    // early stages of development.
    return (_memory.ContainsKey(Address)) ? true : false;
}

public void Set(string SegName, ulong Offset, byte Data)
{
    // Function purpose: Set a byte at an offset from a segment to a given value. This could be
    // used to swap out the instructions.
    Set(SegmentMap[SegName].StartAddr + Offset, Data);
}
public void Set(ulong Address, byte Data)
{
    // Function purpose: Set a byte at an address to a value specified in the byte parameter.

    // Check first if the key already exists (a value has already been assigned to the address).
    // This is required because different methods will have to be used to set the value per the
    // preconditions of the .NET dictionary object (Keys cannot be overwritten with .Add()).
    // This is the best approach as the entire dictionary does not have to be initialised to
    // empty values that would only consume memory on the developer's computer, which would make
    // debugging very slow.
    if (ContainsAddr(Address))
    {
        // Assign by index if it exists
        _memory[Address] = Data;
    }
    else
    {
        // Add the new key if it has not yet been assigned to.
        _memory.Add(Address, Data);
    }
}

public byte this[ulong Address]
{
    // This is an index accessor that is used exactly like an array,
    // e.g. MyMemorySpace[0x10] will return the byte at address 0x10
    // Return the byte at the input address
    get
    {
        return _memory[Address];
    }
    set
    {
        // Allow the index accessor to be used to assign data too, reusing the Set() method.
        Set(Address, value);
    }
}

```

Explanation

The MemorySpace data structure is currently being implemented. It can be constructed using a byte array, which will automatically create segments for the data to be stored in, and segments that opcodes will use such as the stack. Iteration is used to add data from the segment into the memory. This will be useful later on when segments can be loaded into the MemorySpace rather than only plain byte arrays, but this feature would not be necessary at the start of development so other functions have been prioritised. The MemorySpace can be modified and retrieved through the use of the index accessor. This will automatically call the Set function. The Set function will be used more internally by the methods in the class because as the solution develops, it will perform different functions that may not be safe to be used by an outside caller, especially when the AddressMap and AddressRange are implemented, as it will be necessary to have a method of setting data without modifying the map. Because of this, the best approach is to think ahead and provide the index accessor for the majority of outside callers as specific details such as the address map (which is not yet implemented) will be abstracted.

Justification & Relation to break down of the problem

It was necessary to focus on developing this module initially more so than other modules of the program because it was required that public methods and routines reach a certain standard before implementation of the CU can begin. This is because it needs to be developed and tested thoroughly before means of setting and fetching memory can be used. This is because it was identified in the stepwise refinement that many of the functions of the processing layer and intermediary layer will interact with the MemorySpace, such as setting memory and fetching memory. In order to provide the necessary foothold to develop the solutions to these problems, it was necessary to first prioritise the development of the MemorySpace. This data structure is very important in initially as it is the only means of communicating between the interface and processing layer using an abstraction. This is because of the dependency inversion principle success criterion identified in the analysis section. This abstraction is appropriate to meet this criterion as instead of the interface layer depending on the CU directly, it will instead use the MemorySpace object,

which is rarely going to have any external methods changed, whereas the CU will have its inputs, outputs, and preconditions changed frequently because it requires a lot more development to reach a reasonable state where it can be used as a basis for later development stages. Therefore it was necessary to focus development on this module such that it has the methods necessary to continue development.

Opcodes API

```
public class AddImm8 : Opcode // 04=AL, 80
{
    public AddImm8(byte[] InputData = null) { _data = InputData; _string = "ADD"; }
    public override void Execute()
    {
        // Set local capacity of the opcode(used by local functions within the class and base class)
        _regcap = RegisterCapacity.B;
        // The output of an extended byte dest operand is stored in the MultiDefOutput.lMod variable
        // If there was no provided input data, use it must be stored in the multi def output
        ByteCode Dest = (_data == null) ? (ByteCode)MultiDefOutput.lMod : (ByteCode)_data[0];
        // Fetch the immediate
        byte Immediate8 = ControlUnit.FetchNext();
        // Fetch the value stored at the immediate
        byte DestVal = ControlUnit.FetchRegister(Dest, _regcap)[0];
        // Add the immediate onto the memory location
        ControlUnit.SetRegister(Dest, (byte)(Immediate8 + Destval));
    }
}
```

Explanation

The opcode api is in early stages of development; there are few functions available by the CU. A heuristic approach has been taken, certain procedures have not yet been generalised, so have been repeated throughout classes. For example, the opcode has to set and fetch the operands itself. This is because the opcode api/IDecoded operand classes have not been implemented yet.

Justification & Relation to break down of the problem

It was necessary to use a heuristic approach because it is necessary to begin introducing some opcodes in order to test other parts of the program as they provide a simple and realistic measure of what to expect when an instruction is executed. This is important as part of the 1:1 emulation criterion in the analysis section. 1:1 emulation is not possible if the users cannot make use of it, testing individual methods will not determine whether the pipelined large-scale processes work as a whole, such as the FDE cycle in the CU which are most likely to be used by the user. By including opcodes, all the functionality can be tested at once in a realistic setting rather than an engineered ideal situation with perfect inputs.

Test data

As the TestHandler class has not been implemented yet because the intermediary api is not yet sufficient in providing the necessary procedures to execute a testcase, the tests must be performed manually by using a breakpoint in the IDE.

Test #	Aim	How	Test data	Expected result
1	Test addition of immediate byte with 'A' register	The opcode variant to add an immediate with the 'A' register will be used.	<pre> add al,0x10 add ax,0x2000 add eax,0x30000 add rax,0x40000000 </pre>	<Register id="A" size="1">10</Register> <Register id="A" size="2">2010</Register> <Register id="A" size="4">40032010</Register>

Result:

The screenshot shows a debugger interface with the following components:

- Assembly View:** Displays the assembly code for the `ClockStart` method. A breakpoint is set at line 53, and the instruction `_step();` is highlighted.
- Registers View:** Shows the current state of registers. The RAX register is highlighted in blue, showing its value as `0x0000000040032010`.
- Watch 1 Window:** Contains a table with three rows:

Name	Type	Value
Opcodetable		error CS0103: The name 'Opcodet...
(ulong)RAX	ulong	0x0000000040032010
(ulong)RIP	ulong	0x00000000000100020
- Call Stack Window:** Shows the call stack with the following entries:

Name
debugger.ControlUnit.ClockStart(debugger.N...
debugger.VM.Run.AnonymousMethod_0()
System.Threading.ThreadHelper.ThreadStart
System.Threading.ExecutionContext.RunInte...
System.Threading.ExecutionContext.Run(Sys...

Justification & Relation to break down of the problem

After all instructions have been executed, the output value is correct(in the RAX register at the bottom left). No remedial action is necessary.

Test #	Aim	How	Test data	Expected result
2	Test addition of immediate with memory	The opcode variant to add an immediate with memory at an address will be used.	<pre> add bl,0x10 add bh,0x10 add bx,0x1000 add ebx,0x30000 add rbx,0x40000000 </pre>	<Register id="B" size="1">10</Register> <Register id="B" size="2">2010</Register> <Register id="B" size = "8">40032010</Register>

The screenshot shows a debugger interface with the following details:

- Assembly View:**

```

38     public static void ClockStart(MemorySpace _memory)
39     {
40         // Set BytePointer to beginning of code
41         BytePointer = _memory.EntryPoint;
42         // Update the memory reference in the CU to point to the memory passed as a parameter
43         // This is the memory that will be modified by the instructions.
44         Memory = _memory;
45         // Set up SP and BP to point to start of stack
46         RSP = _memory.SegmentMap[".stack"].StartAddr;
47         RBP = RSP;
48         // Begin executing at start of user's code
49         RIP = Memory.SegmentMap[".main"].StartAddr;
49         // Indefinitely execute instructions
50         while (true)
51         {
52             _step(); s 1ms elapsed
53         }
54     }
55     private static void _step()
56     {
57         // Function purpose: Perform one iteration of the Von Neumann Fetch Decode Execute cycle.
58         // This means that a single instruction will be executed.
59
60         // Set the byte pointer to the instruction pointer
61         BytePointer = RIP;

```
- Watch List (Watch 1):**

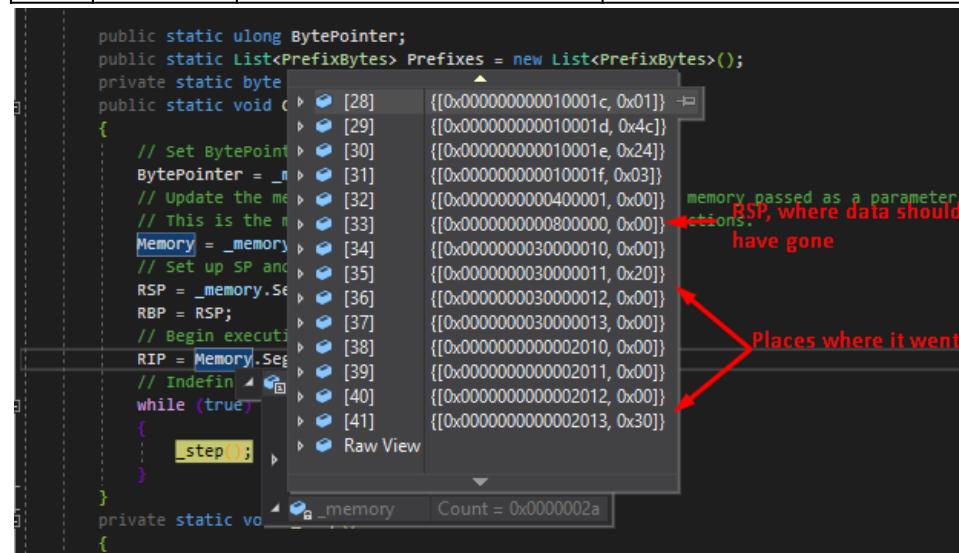
Name	Type
OpcodeTable	error CS0103: The name 'OpcodeT...' C
(ulong)RBX	ulong
(ulong)RIP	ulong
Add item to watch	
- Call Stack:**

Name
debugger.ControlUnit.ClockStart(debugger.MemorySpace)
debugger.VM.Run.AnonymousMethod_0()
System.Threading.ThreadHelper.ThreadStart_Context(object)
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext, object, object)
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, object, object)
System.Threading.ExecutionContext.Run(System.Threading.ExecutionContext, object, object)

Explanation

The results(in the RBX register at the bottom left) are the same as the expected results. No remedial action is necessary.

Test #	Aim	How	Test data	Expected result
4	Test addition of register with memory	The opcode variant to add a register with a memory address will be used. The values must first be moved into registers because adding immediates uses a separate opcode to adding registers to memory.	<pre> mov eax,0x10 mov ebx,0x2000 mov ecx,0x30000000 mov edx,0x0 add BYTE PTR [rsp],al add WORD PTR [rsp+0x1],bx add DWORD PTR [rsp+0x3],ecx </pre>	<Memory offset_register="SP">10</Memory> <Memory offset_register="SP" offset="1">0020</Memory> <Memory offset_register="SP" offset="3">00000030</Memory>



Explanation

This test did not pass, the data was inserted at arbitrary locations instead of at the stack pointer.

Remedial action

```

// Check if destination is an address(needs to be set differently)
if (DestSrc.IsAddress)
{
    // Interpret the ModRM as a pointer
    ulong lDestAddr = DestSrc.lMod;
    // Fetch the source register, taking only the first byte because it is a byte instruction
    byte bSrc = ControlUnit.FetchRegister(bcSrcReg, _regcap)[0];
    // Fetch the destination address
    byte bDest = ControlUnit.Fetch(lDestAddr)[0];
    ControlUnit.SetMemory(lDestAddr, (byte)(bSrc + bDest)); 2.
}
else
{
    // Interpret the ModRM as a register
    ByteCode bcDestReg = (ByteCode)DestSrc.lMod;
    // Fetch the source register, taking only the first byte because it is a byte instruction
    byte bSrc = ControlUnit.FetchRegister(bcDestReg, _regcap)[0]; 3.
    // Fetch the destination register
    byte bDest = ControlUnit.FetchRegister(bcDestReg, _regcap)[0];
    // Set the register with the sum of the two operands added together
    ControlUnit.SetRegister(bcDestReg, (byte)(bSrc + bDest));
}
else
{
    // Check if destination is an address(needs to be set differently)
    if (DestSrc.IsAddress)
    {
        // Interpret the ModRM as a pointer
        ulong lDestAddr = DestSrc.lMod;
        // Fetch the source register, taking only the first byte because it is a byte instruction
        byte bSrc = ControlUnit.FetchRegister(bcSrcReg, _regcap)[0];
        // Fetch the destination address
        byte bDest = ControlUnit.Fetch(lDestAddr)[0];
        // set the register with the sum of the two operands added together
        ControlUnit.SetRegister(bcSrcReg, (byte)(bSrc + bDest));
    }
}

```

Red annotations indicate potential issues:

- A red arrow labeled "1." points to the first assignment of bDest in the else block.
- A red arrow labeled "2." points to the assignment of bDest in the first if block.
- A red arrow labeled "3." points to the assignment of bSrc in the second if block.

Explanation

The source of the bug has been located to be a typo in the ADD opcode class file. In the procedure to set the result, a condition is used to check whether the operand is a memory address or a register as there are two separate procedures for setting them. The bug was identified at #1, where the wrong parameter had been called to do this, which treat the address as a register instead of an address, causing the strange behaviour. The procedure marked at #2 should have instead been called. In the process of debugging, a bug at #3 was identified where the wrong parameter was used, instead “bcSrcReg” should have been used.

Justification

This is the best course to take as a remedial action because logical errors have been identified in the code, therefore it is almost certain that these caused, or at minimum contributed to the erroneous behaviour. As a result of this, the best approach is to fix the identified bugs, then re-test the code to look for any more bugs that were not found. This will be performed in the next version test of the iterative cycle.

Review

Criterion	My opinion	Stakeholder opinion
The program should be able to accurately emulate a x86-64 processor from an outside perspective.	This criterion has not yet been met. There are data structures such as the FlagSet, which hold the outputs of an opcode, that have not been implemented, so it has not been possible to qualify this criterion yet.	“The procedures are accurate in terms of outputs from a general perspective; adding two numbers gives the correct answer, however there is not a large enough feature set to consider this criterion met.”
The program should be able to execute instructions in a timely manner.	The instructions execute relatively fast, there is no noticeable delay.	“There is no cause for concern here, no overhead was noticeable. It also scaled relatively well, I could execute many instructions quickly.”
The executable program and source code should be a great learning tool for the stakeholders.	There are many comments to contribute to this, although the heuristic approach could make it hard for stakeholders to visualise the processes and concepts as they are mostly long IF and SWITCH blocks.	“Lots of comments throughout the code that explained the processes but there was not much to learn assembly-wise because the bitwise procedures were abstracted into ulong integer data types. It would be nice to see binary operations implemented”.
Complex low level concepts will be abstracted into simple data structures and delegates	The MemorySpace class is functional, it does provide some of the abstraction it is designed to perform such as making the dictionary appear to be an entire memory map, whilst only storing necessary bytes, however other abstractions such as the RegisterGroup have not yet been implemented.	“There was one abstraction present, the MemorySpace, however static structures were used to store registers. This will not a good design choice as it will restrict the amount of concurrency possible, so an abstracted object would be a better approach in the upcoming versions”
Liskov substitution principle	The apis necessary to create the interfaces and classes have not yet been developed, to produce makeshift functions now would only consume more time later when they have to be removed.	“There were no interfaces present in the code, so objects could not be accessed by subtypes. This should be addressed as soon as possible otherwise the code will have rooted dependencies on the prototype code.”

Beta version

Date: 11th July 2019

Version explanation

This version includes more user interface functionality; a register viewer and memory viewer. Lots of repeated code has been generalised. AND, XOR ,and OR opcodes have been added. JMP opcodes have been implemented, including all condition codes. The Bitwise library has started to develop, implementing arithmetic procedures that are frequently used by opcodes. In the UI, register panel, the memory viewer, and the flags panel have now been implemented.

JMP opcode

```
// Two byte opcode table
OpcodeTable.Add(2, new Dictionary<byte, Action>()
{
    { 0x80, () => new Jmp(Relative:true, Bytes:4, Opcode:"J0").Execute() },
    { 0x81, () => new Jmp(Relative:true, Bytes:4, Opcode:"JNO").Execute() },
    { 0x82, () => new Jmp(Relative:true, Bytes:4, Opcode:"JC").Execute() },
    { 0x83, () => new Jmp(Relative:true, Bytes:4, Opcode:"JNC").Execute() },
    { 0x84, () => new Jmp(Relative:true, Bytes:4, Opcode:"JZ").Execute() },
    { 0x85, () => new Jmp(Relative:true, Bytes:4, Opcode:"JNZ").Execute() },
    { 0x86, () => new Jmp(Relative:true, Bytes:4, Opcode:"JNA").Execute() },
    { 0x87, () => new Jmp(Relative:true, Bytes:4, Opcode:"JA").Execute() },
    { 0x88, () => new Jmp(Relative:true, Bytes:4, Opcode:"JS").Execute() },
    { 0x89, () => new Jmp(Relative:true, Bytes:4, Opcode:"JNS").Execute() },
    { 0x8A, () => new Jmp(Relative:true, Bytes:4, Opcode:"JP").Execute() },
    { 0x8B, () => new Jmp(Relative:true, Bytes:4, Opcode:"JNP").Execute() },
    { 0x8C, () => new Jmp(Relative:true, Bytes:4, Opcode:"JL").Execute() },
    { 0x8D, () => new Jmp(Relative:true, Bytes:4, Opcode:"JGE").Execute() },

    switch(_string)
    {
        // speed codesize tradeoff, we could have it test for each one, make a list of opcodes that would take the jump
        // given the eflags and jump if the given opcode is in that list
        // jmps are used alot so its probably best we dont
        // wrapped ifs in !(-) rather than simplifying for readability, return on negative case so we dont repeat rip = dest
        case "JA": //77 jump above ja, jnb UNFOR SIGNED
            if(!(!EFlags.Carry && !EFlags.Zero)) { return; }
            break;
        case "JNC": //73 jump no carry jnc, jae, jnb FOR UNSIGNED
            if(!(!EFlags.Carry)) { return; }
            break;
        case "JC": // 72 jump carry: jb, jc, jnae FOR UNSIGNED
            if (!(!EFlags.Carry)) { return; }
            break;
        case "JNA": //76 JNA jna UNFOR SIGNED
            if(!(!EFlags.Carry || !EFlags.Zero)){ return; }
            break;
        case "JRCXZ":
            //E3 jump rcx zero, with an addr32 prefix this becomes jecxz (ecx zero) , jcxz is 32bit only because in 32bit mode jcxz uses the addr32 prefix not jecxz
            if(!_prefixes.Contains(PrefixBytes.ADDR32) && ECX == 0) || !(RCX==0)){ return; } //where jecxz then has none
            break;
        case "JZ": //74 jump zero
            if(!(!EFlags.Zero)) { return; }
            break;
        case "JNZ": //75 jump not zero, jne
            if (!(!EFlags.Zero)) { return; }
            break;
        case "JG": // 7F jump > ,JNLE FOR SIGNED
            if(!(!EFlags.Zero && EFlags.Sign == EFlags.Overflow)) { return; }
            break;
        case "JGE": // 7D j >= , jnl FOR SIGNED
            if(!(!EFlags.Sign == EFlags.Overflow)) { return; }
            break;
        case "JL": //7C j < ,jnge FOR SIGNED
            if(!(!EFlags.Sign != EFlags.Overflow)) { return; }
            break;
        case "JLE": //7E j <= , jng FOR SIGNED
            if(!(!EFlags.Zero || EFlags.Sign != EFlags.Overflow)) { return; }
            break;
        case "J0": //70 jump overflow
            if (!(!EFlags.Overflow)) { return; }
            break;
        case "JNO": // 71 jump no overflow
            if(!(!EFlags.Overflow)) { return; }
            break;
        case "JS": // 78 jump sign jump negative
            if(!(!EFlags.Sign)) { return; }
            break;
        case "JNS": // 79 jump not sign/ jump positive, jump >0
            if(!(!EFlags.Sign)) { return; }
            break;
        case "JP": // 7a jump parity, jump parity even
            if(!(!EFlags.Parity)) { return; }
            break;
        case "JNP": // 7b jump no parity/odd parity
            if (!(!EFlags.Parity)) { return; }
            break;
    }
    // Jump to new address
    ControlUnit.BytePointer = DestAddr;
}
```

Explanation

JMP conditional opcodes have been added to the two byte opcode table. Other opcodes were added to the OpcodeTable, nothing has needed to change there. The mnemonic is passed to the JMP code as parameter, which is then used later to determine the condition to test by a switch case statement, which will break out of the method before RIP is changed if the condition is not met. The relative boolean indicates that the jump is an offset from the current RIP, not an exact pointer to an address. The bytes parameter indicates that the operand size is four bytes.

Justification & Relation to break down of the problem

This is the best approach to implementing the JMP opcode because there are many different condition codes available in the x86-64 specification. It was identified in the design and analysis sections that a TestCondition() method in the opcode base class would be implemented. As the intermediary layer is not yet fully developed, a heuristic approach is the most justifiable because it took a small amount of development time and will still be easy to transition into using the new method when it is available, whilst still being able to show to the stakeholders that more of the x86-64 functionality is being introduced.

Review

Self review - The code is very effective and demonstrative of purpose, it is clear to see what is going on and the conditions for each requirement to be met are clear. As stated in a comment, it would not be possible to optimise code any further because there are a lot of conditions that need to be considered one way or another.

Stakeholder review, an existing high level programmer who expressed interest in assembly - "This code is very understandable, conditions for each opcode have been labelled clearly through comments. The comments are very helpful in general, for example, it is labelled that JNL(Jump not less than) and others are specifically for signed numbers, which I did not know before, and JC/JNC(Jump (no) carry) are for unsigned numbers."

Bitwise library

```
public static byte[] Add(byte[] baInput1, byte[] baInput2, RegisterCapacity _regcap, bool bCarry)
{
    /*int iSum; // revisit this
    byte[] baResult = new byte[baInput1.Length];
    for (int i = baInput1.Length-1; i >= 0; i--)
    {
        iSum = baInput1[i] + baInput2[i];
        if(iSum > 255)
        {
            baResult[i - 1] += 1;
            baResult[i] += (byte)(iSum - 255);
        } else
        {
            baResult[i] += (byte)iSum;
        }
    }*/
    // Convert to integers, then add(if there is a carry, one is added here), then convert back.
    byte[] baResult = BitConverter.GetBytes(
        Convert.ToInt64(
            BitConverter.ToInt64(baInput1, 0)
            + BitConverter.ToInt64(baInput2, 0)
            + (ulong)(bCarry && Eflags.Carry ? 1 : 0)
        ));
    // Set the flags using the opcode API method
    Opcode.SetFlags(baResult, Opcode.FlagMode.Add, baInput1, baInput2);
    // Trim the result to the correct size(as it was resized to a ulong)
    return Trim(baResult, _regcap);
}
```

Explanation

Bitwise methods such as Add, Subtract, Divide, and Multiply have been added to the bitwise library. The functions have temporarily been implemented by converting the values into integer types then performing the relevant operation. A true bitwise operation(that works on byte arrays only) is currently in development, but a working prototype has been kept in place instead.

Justification & Relation to break down of the problem

This is the best approach to implementing these arithmetic operations because it allows them to be reused throughout the code at an early stage. It was identified in the analysis and design sections that the processing layer would use the bitwise methods to implement opcodes such as the ADD, MUL, SUB opcodes, as their primary

purpose is arithmetic capability. As these opcodes are being implemented into the project currently, this approach is most suitable because the new code will already be dependent on the utility class rather than its own procedures. Not only does this save code being rewritten, hence saves development time, but also means that when the procedures are updated in the future, the existing usages will not have to adapt as the inputs and outputs will be the same. This will allow the essential features such as opcodes to be deployed faster because the procedure does not have to be retested for their individual usage, rather the method in the library can be used instead, which have already been tested.

Tests

Test #	Aim	How	Test data	Expected result
1	Test whether two inputs are added together correctly	Two normal inputs will be used as parameters to the function with the carry parameter false.	Input1 = [11,11] Input2 = [22,22] Carry = false	Output = [33,33] PF = true

```

public static byte[] Add(byte[] baInput1, byte[] baInput2, RegisterCapacity _regcap, bool bCarry)
{
    /*int iSum; // revisit this
byte[] baResult = new byte[baInput1.Length];
for (int i = baInput1.Length-1; i >= 0; i--)
{
    iSum = baInput1[i] + baInput2[i];
    if(iSum > 255)
    {
        baResult[i - 1] += 1;
        baResult[i] += (byte)(iSum - 255);
    } else
    {
        baResult[i] += (byte)iSum;
    }
}*/
```

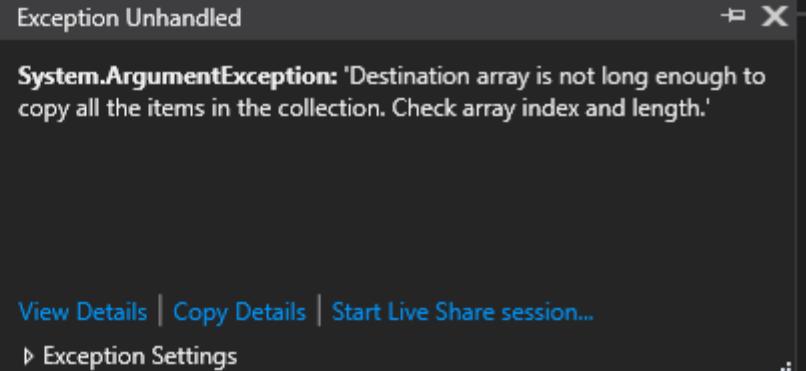
// Convert to integers, then add(if there is a carry, one is added here), then convert back.

```

byte[] baResult = BitConverter.GetBytes(
    Convert.ToInt64(
        BitConverter.ToInt64(baInput1, 0) +
        BitConverter.ToInt64(baInput2, 0) +
        (ulong)(bCarry && Eflags.Carry ? 1 : 0)));

```

// Set the flags using the opcode API method
Opcode.SetFlags(baResult, Opcode.FlagMode);
// Trim the result to the correct size(as required)
return Trim(baResult, _regcap);
}



Explanation

A method was hardcoded at the start of the program to call the Add method with the required parameters. There was an exception, therefore the testcase did not pass, so remedial action is required.

Remedial actions

The problem can be identified as the array being a different size to what the ToUInt64 method expects. This is because it requires the array to be of length 8.

There are two possible solutions to this problem, one is to use Array.Resize() and resize the array to match the required size of the BitConverter method. The other is to create a new method to convert the input to a ulong that does not require the precondition that there are exactly 8 bytes. To compare the two, a performance test was performed.

```

7  {
8      var sw = new Stopwatch();
9      byte[] asd = new byte[] { 49,2,4,0,0,0,0,0 };
10
11
12      asd = new byte[] { 49,2,4 };
13      sw.Start();
14      ulong a = 0;
15      for(int i=0; i < 1000000; i++) {
16          a = ToULong(asd);
17      }
18      sw.Stop();
19      Console.WriteLine("toulong:::::" + sw.Elapsed + " " + a);
20      sw.Reset();
21
22      asd = new byte[] { 49,2,4,0,0,0,0,0 };
23      sw.Start();
24      for(int i=0; i < 1000000; i++) {
25          a = BitConverter.ToInt64(asd,0);
26      }
27      sw.Stop();
28      Console.WriteLine("bitconverter:" + sw.Elapsed + " " + a);
29      sw.Reset();
30
31
32 }
33
34
35     public static ulong ToULong(byte[] baInput)
36     {
37         ulong lResult = 0;
38         for (int i = 0; i < baInput.Length; i++) //ba works like base 256 [256^0] [256^1] [256^2]
39         { // 49 2 4
40             lResult += (ulong)(baInput[i] << 8*i); // (1 * 49) + (256*2) + (256^2 * 4) = 262705
41         }
42         return lResult;
43     }
44 }
```

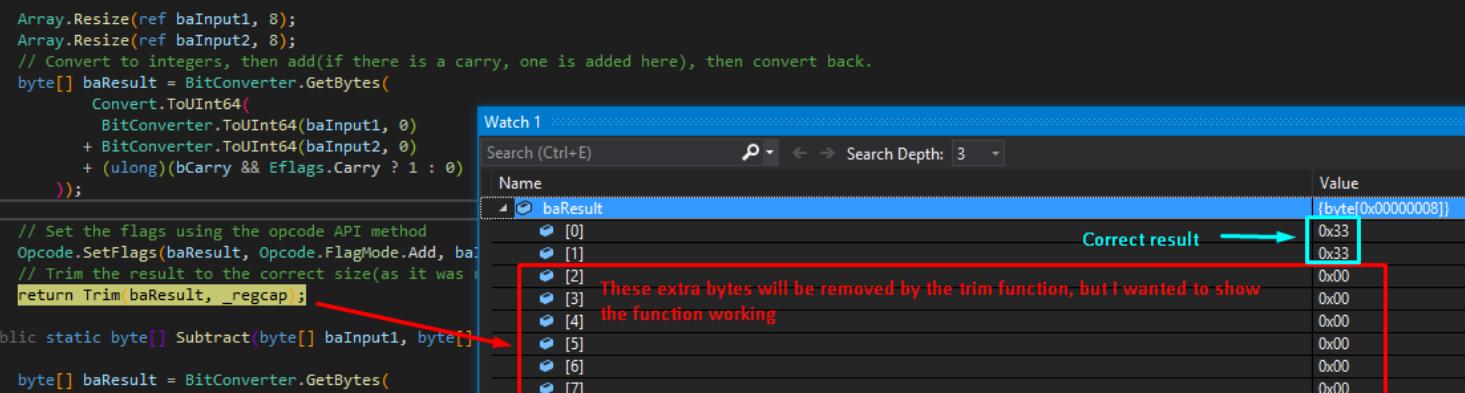
toulong:::::00:00:00.0228321 262705
bitconverter:00:00:00.0097821 262705

The test shows that the ToULong method was much slower. This would not scale very well as the program develops as the code will be using inefficient methods frequently. Soon a new addition algorithm will be implemented, so it is not justifiable to change the existing code to use a completely new method. For this reason, the array will be resized.

```

Array.Resize(ref baInput1, 8);
Array.Resize(ref baInput2, 8);
// Convert to integers, then add
byte[] baResult = BitConverter.C
```

In order to fix this, two lines have been added to the start of the function that resize the input array to the correct size. This will allow inputs of any size to work with the function. The function has then been tested again to confirm this.



```

Array.Resize(ref baInput1, 8);
Array.Resize(ref baInput2, 8);
// Convert to integers, then add(if there is a carry, one is added here), then convert back.
byte[] baResult = BitConverter.GetBytes(
    Convert.ToInt64(
        BitConverter.ToInt64(baInput1, 0)
        + BitConverter.ToInt64(baInput2, 0)
        + (ulong)(bCarry && Eflags.Carry ? 1 : 0)
    )
);

// Set the flags using the opcode API method
Opcode.SetFlags(baResult, Opcode.FlagMode.Add, baInput1);
// Trim the result to the correct size(as it was longer)
return Trim(baResult, _regcap);
}

public static byte[] Subtract(byte[] baInput1, byte[]
{
    byte[] baResult = BitConverter.GetBytes(
        Convert.ToInt64(
            BitConverter.ToInt64(baInput1, 0)
            - BitConverter.ToInt64(baInput2, 0)
            - (ulong)(bCarry && Eflags.Carry ? 1 : 0)
        )
    );
}
```

The test now passes as the result is the same as the expected result, so no further remedial action is required.
Justification of remedial actions

Remedial action

Justification

Trying other solutions	It was necessary to evaluate other options to solve the problem because there could be a better approach that could be identified that could have been more performant or maintainable. However, this was not the case as it was slower and had no advantages over resizing the arrays
Resizing the input arrays	It was necessary to add resize to the start in order to meet the preconditions of the BitConverter.ToInt64() method. This is because it will ensure that every input will always meet the preconditions, regardless of whether it is too big or too small, which will assist in future versions that may use arbitrary sized inputs, not just 8 bytes.
Repeating tests	It was necessary to test the function again afterwards to make sure that the changes were effective and corrected the bugs entirely. The new method could have introduced further bugs or caused exceptions. For this reason, repeating the tests was a suitable remedial action to take.

OR, XOR, AND

```

public static byte[] Or(byte[] baData1, byte[] baData2)
{
    // Create a bit string out of the inputs, e.g. "11010101"
    string sBits1 = GetBits(baData1);
    string sBits2 = GetBits(baData2);
    // Pad the strings to the same size, using 0's so the value is not changed
    // This is necessary so they can be iterated over together in a for loop.
    PadEqual(ref sBits1, ref sBits2);
    // Create an output string of bits to store the result
    string sOutput = "";
    // Iterate through every character in the bit string
    for (int i = 0; i < sBits1.Length; i++)
    {
        // If either string has a one at this index, append a 1 to the output, otherwise a zero
        // (Exactly as a inclusive or does)
        if (sBits1[i] == '1' || sBits2[i] == '1')
        {
            sOutput += "1";
        }
        else
        {
            sOutput += "0";
        }
    }
    // Convert the output string back into bytes
    byte[] baResult = GetBytes(sOutput);
    // Set any necessary flags(exact details abstracted from here)
    Opcode.SetFlags(baResult, Opcode.FlagMode.Logic, baData1, baData2);
    // Return the result.
    return baResult;
}

```

```

public class Xor : MyOpcode
{
    ModRM DestSrc;
    byte[] baDestBytes;
    byte[] baSrcBytes;
    public Xor()
    {
        // Fetch the current capacity of operands that will be used
        ControlUnit.CurrentCapacity = GetRegCap();
        // Fetch the ModRM operand
        DestSrc = ControlUnit.ModRMDDecode(ControlUnit.FetchNext());
        // Store the mnemonic in the opcode base class
        _string = "XOR";
        // Fetch the destination bytes
        baDestBytes = FetchDynamic(DestSrc, IsSwap);
        // Determine and fetch the source
        ByteCode bcSrcReg = (ByteCode)DestSrc.lSource;
        baSrcBytes = ControlUnit.FetchRegister(bcSrcReg);
    }

    public override void Execute()
    {
        // XOR the source and destination and set the result
        SetDynamic(DestSrc, Bitwise.Xor(baDestBytes, baSrcBytes), IsSwap);
    }
}

```

Explanation

The OR, XOR, AND methods have been introduced to the bitwise class. Their respective opcodes have also been coded. The OR function(and alike) work by creating a bit string from the inputs and iterating through the string instead and applying the logical operator whilst doing so. The individual opcode classes are designed to first fetch all the necessary information they need in the constructor, then perform their operation and store the result in the Execute() method.

Justification & Relation to break down of the problem

This is the best approach to implementing the logical operators because it is much clearer to see how the operation is performed. As the KISS principle success criterion discussed in the analysis section states that code should be kept as simple as possible in order to allow the solution to be more accessible to stakeholders, the best approach is to trial this algorithm(as it is essentially the same as the planned algorithm in the design section except with extra steps) and ask for the stakeholders opinions in a conducted review. This will allow for a more accurate conclusion on how skilled the average stakeholder is; would they prefer the bitwise approach, or the simpler string based approach. Thinking ahead is used by separating the opcodes into the constructor method and execute method. This is because when the disassembler class is complete, it will have to perform a similar setup procedure as when instructions are executed normally. Separating this into two distinct sections will make the transition into two separate functions, Execute() and Disassemble(), much smoother as the code will already be designed to store the variables in the class-scope, such that they can be accessed when either disassembly or execution is being performed. The two separate methods were planned in the design section, however it would not make sense to program disassemble now because certain details about how disassembly will be performed are not yet clear, such as the order in which disassembly tasks must be performed. Therefore, the best approach is to plan for the smooth transition without directly coding it, as it would be highly likely that the code would have to be rewritten because of factors that are not foreseeable in the current state of the program.

Tests

Test #	Aim	How	Test data	Expected result
1	Test AND of two inputs	Two inputs will be entered as parameters, then the output will be compared to a known correct output sourced reliably from a calculator.	Input1 = [AA, AA, AA, AA] Input2 = [CC, CC, CC, CC]	Output = [88,88,88,88]

```

187     public void TestingProcedure()
188     {
189         byte[] Result = Util.Bitwise.And(new byte[] { 0xAA, 0xAA, 0xAA, 0xAA }, new byte[] { 0xCC, 0xCC, 0xCC, 0xCC });
190         return; // Result {byte[0x00000004]} [0] 0x88 [1] 0x88 [2] 0x88 [3] 0x88
191     }
192 }
193
194

```

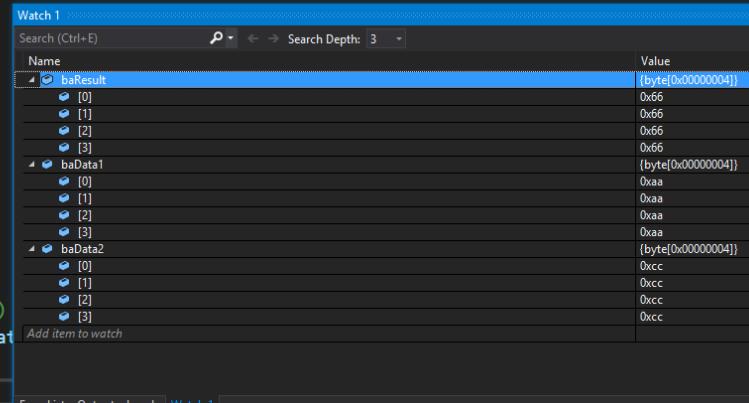
Explanation

The testing procedure is executed at the start of the program and a breakpoint was used to check the result of the AND method. The two parameters were “AND”ed, the result stored in Result. Result was the same as the expected result, hence the test passed, so no remedial action is needed.

Test #	Aim	How	Test data	Expected result
2	Test XOR of two inputs	Two inputs will be entered as parameters, then the output will be compared to a known correct output sourced reliably from a calculator.	Input1 = [AA, AA, AA, AA] Input2 = [CC, CC, CC, CC]	Output = [66,66,66,66]

```
public static byte[] Xor(byte[] baData1, byte[] baData2)
```

```
{
    // Create a bit string out of the inputs, e.g. "11010101"
    string sBits1 = GetBits(baData1);
    string sBits2 = GetBits(baData2);
    // Pad the strings to the same size, using 0's so the value is not changed
    // This is necessary so they can be iterated over together in a for loop.
    PadEqual(ref sBits1, ref sBits2);
    // Create an output string of bits to store the result
    string sOutput = "";
    for (int i = 0; i < sBits1.Length; i++)
    {
        // Append a 1 to the output if and only if there is a single one in the inputs, otherwise a zero
        // (Exactly as a exclusive or does)
        if (sBits1[i] == '0' ^ sBits2[i] == '0')
        {
            sOutput += "1";
        }
        else
        {
            sOutput += "0";
        }
    }
    // Convert the output string back into bytes
    byte[] baResult = GetBytes(sOutput);
    // Set any necessary flags(exact details abstracted from here)
    Opcode.SetFlags(baResult, Opcode.FlagMode.Logic, baData1, baData2);
    // Return the result.
    return baResult;
}
```



Error List Output Locals Watch 1

Watch 1

Name	Value
baResult	{byte[0x00000004]}
[0]	0x66
[1]	0x66
[2]	0x66
[3]	0x66
baData1	{byte[0x00000004]}
[0]	0xaa
[1]	0xaa
[2]	0xaa
[3]	0xaa
baData2	{byte[0x00000004]}
[0]	0xcc
[1]	0xcc
[2]	0xcc
[3]	0xcc

Explanation

A breakpoint was positioned at the return statement in the XOR procedure. A method call was crafted at the start of the program to call the XOR function with the test data specified. At the breakpoint, the result is not equal to the expected result of the test, therefore the test did not pass and further remedial action is required.

Remedial actions

```
public static byte[] Xor(byte[] baData1, byte[] baData2)
{
    // Create a bit string out of the inputs, e.g. "11010101"
    string sBits1 = GetBits(baData1);
    string sBits2 = GetBits(baData2);
    // Pad the strings to the same size, using 0's so the value is not changed
    // This is necessary so they can be iterated over together in a for loop.
    PadEqual(ref sBits1, ref sBits2);
    // Create an output string of bits to store the result
    string sOutput = "";
    for (int i = 0; i < sBits1.Length; i++)
    {
        // Append a 1 to the output if and only if there is a single one in the inputs, otherwise a zero
        // (Exactly as a exclusive or does)
        if (sBits1[i] == '0' ^ sBits2[i] == '0') ← Should be '1' ^ '1'
        {
            sOutput += "1";
        }
        else
        {
            sOutput += "0";
        }
    }
    // Convert the output string back into bytes
    byte[] baResult = GetBytes(sOutput);
    // Set any necessary flags(exact details abstracted from here)
    Opcode.SetFlags(baResult, Opcode.FlagMode.Logic, baData1, baData2);
    // Return the result.
    return baResult;
}
```

Explanation

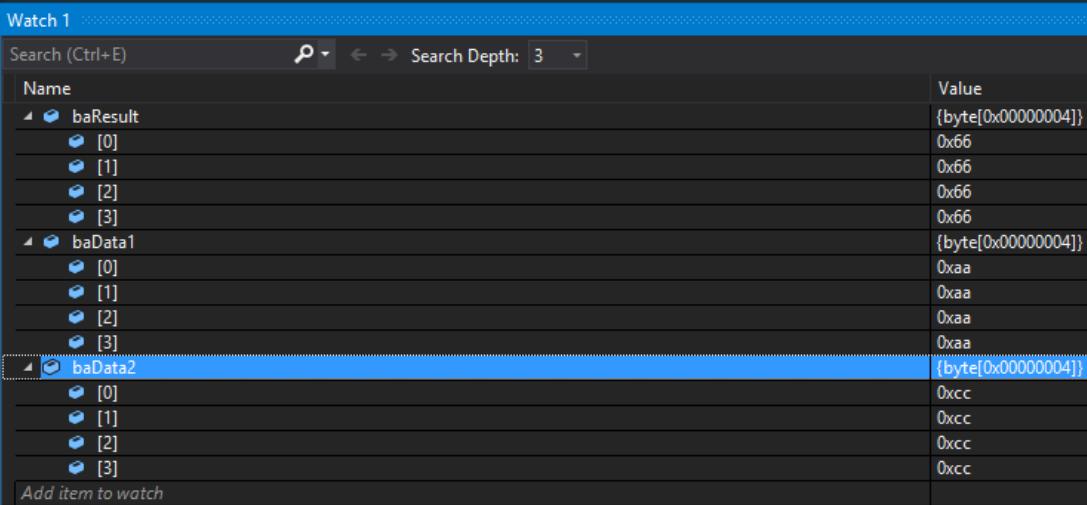
A logical error was found in the code. This is because the implementation of an XOR was wrong. Instead of checking if there was only a single one in the inputs at position \$i in the array, it checked if there was only a single zero. This caused the incorrect output in the testcase as the expected result was the result of an XOR. As remedial action, the code has been corrected to '1's.

```
if (sBits1[i] == '1' ^ sBits2[i] == '1')
```

The code has then been retested against the test to make sure that there are no further issues.

The result is now the same as the expected result, hence the test passed, so no further remedial action is required.

```
for (int i = 0; i < sBits1.Length; i++)
{
    // Append a 1 to the output if and only if there is a single one in the inputs, otherwise
    // (Exactly as a exclusive or does)
    if (sBits1[i] == '1' ^ sBits2[i] == '1')
    {
        sOutput += "1";
    }
    else
    {
        sOutput += "0";
    }
}
// Convert the output string to bytes
byte[] baResult = GetByt
// Set any necessary flags
Opcode.SetFlags(baResult);
// Return the result.
return baResult;
```



Name	Value
baResult	{byte[0x00000004]}
[0]	0x66
[1]	0x66
[2]	0x66
[3]	0x66
baData1	{byte[0x00000004]}
[0]	0xaa
[1]	0xaa
[2]	0xaa
[3]	0xaa
baData2	{byte[0x00000004]}
[0]	0xcc
[1]	0xcc
[2]	0xcc
[3]	0xcc

Justification of remedial action

Remedial action	Justification
Using test data to track down the bug	The test data itself had narrowed down the function where the error occurs. This allowed me to quickly be able to track down the bug by analysing the source code of the method and find the logical error. This action was most appropriate as it used the advantages of the test design to reduce the amount of development time that has to be spent unnecessarily searching through other code modules to find an error, therefore allowing the fix to be applied as fast as possible such that I could continue developing the essential features.
Testing the function again	It was appropriate to test the function again afterwards because there could have been other logical errors in the code that had not been identified. If the test failed afterwards, there would be cause for further remedial action. This ensured that the function was fully functional before releasing the version.

Test #	Aim	How	Test data	Expected result
3	Test OR of two inputs	Two inputs will be entered as parameters, then the output will be compared to a known correct output sourced reliably from a calculator.	Input1 = [AA, AA, AA, AA] Input2 = [CC, CC, CC, CC]	Output = [EE,EE,EE,EE]

```

public static byte[] Or(byte[] baData1, byte[] baData2)
{
    // Create a bit string out of the inputs, e.g. "11010101"
    string sBits1 = GetBits(baData1);
    string sBits2 = GetBits(baData2);
    // Pad the strings to the same size, using 0's so the value is not changed
    // This is necessary so they can be iterated over together in a for loop.
    PadEqual(ref sBits1, ref sBits2);
    // Create an output string of bits to store the result
    string sOutput = "";
    // Iterate through every character in the bit string
    for (int i = 0; i < sBits1.Length; i++)
    {
        // If either string has a one at this index, append a 1 to the output, otherwise a zero
        // (Exactly as a inclusive or does)
        if (sBits1[i] == '1' || sBits2[i] == '1')
        {
            sOutput += "1";
        }
        else
        {
            sOutput += "0";
        }
    }
    // Convert the output string back into bytes
    byte[] baResult = GetBytes(sOutput);
    // Set any necessary flags(exact details abstracted from here)
    Opcode.SetFlags(baResult, Opcode.FlagMode.Logic, baData1, baData2);
    // Return the result.
    return baResult;
}

```

Name	Value
baResult	{byte[0x00000004]}
[0]	0xee
[1]	0xee
[2]	0xee
[3]	0xee
baData1	{byte[0x00000004]}
[0]	0xaa
[1]	0xaa
[2]	0xaa
[3]	0xaa
baData2	{byte[0x00000004]}
[0]	0xcc
[1]	0xcc
[2]	0xcc
[3]	0xcc

Name	Value
baResult	{byte[0x00000004]}
[0]	0xee
[1]	0xee
[2]	0xee
[3]	0xee
baData1	{byte[0x00000004]}
[0]	0xaa
[1]	0xaa
[2]	0xaa
[3]	0xaa
baData2	{byte[0x00000004]}
[0]	0xcc
[1]	0xcc
[2]	0xcc
[3]	0xcc

Explanation

A breakpoint was positioned at the return statement in the OR procedure. A method call was crafted at the start of the program to call the OR function with the test data specified. At the breakpoint, the result is equal to the expected result of the test, therefore the test passed and no remedial action is required.

Review

In this case, it will be more relevant to have two reviews from stakeholders rather than my own as it directly concerns them; I understand the code but they are the ones who may not.

Stakeholder 1, an existing high level programmer - “I like the idea of using strings, it was very easy to grasp, however I feel that it is too great of an abstraction. As this software advertises itself as a learning opportunity, it would be better to implement the realistic algorithm and add further explanation than to provide the abstracted algorithms that will not be useable in many circumstances”

Stakeholder 2, an existing high level programmer - “It would be much better to include a true bitwise procedure in my opinion. Using strings in this way feels to superficial and unnecessary costs to performance. I personally found it harder to interpret, however I already know about how logical operations”

Generalisation of repeated code

```
public static byte[] FetchDynamic(ModRM ModRMBYTE, bool Swap=false)
{
    // Function purpose: Fetch the source of a ModRM
    // Check if the ModRM is MR encoded(the source and dest are swapped)
    if (Swap)
    {
        // If the ModRM holds an address, the source needs to be fetched as a pointer,
        // otherwise a register, so this condition determines the appropriate method to use
        if (ModRMBYTE.IsAddress)
        {
            return ControlUnit.Fetch(ModRMBYTE.lDest, ControlUnit.CurrentCapacity);
        }
        else
        {
            return ControlUnit.FetchRegister((ByteCode)ModRMBYTE.lDest);
        }
    }
    // If not a swap, the source is always stored in a register
    else
    {
        return ControlUnit.FetchRegister((ByteCode)ModRMBYTE.lSource);
    }
}
```

```
public static void SetDynamic(ModRM ModRMBYTE, byte[] baData, bool Swap=false)
{
    // Function purpose: Set the destination of a ModRM byte
    // Check if the ModRM holds an address
    if (ModRMBYTE.IsAddress)
    {
        // Check if the ModRM is MR encoded(the source and dest are swapped), in this case,
        // the source would be a register(because the address is in the other operand,
        // IsAddress does not account for swaps).
        if(Swap)
        {
            ControlUnit.SetRegister((ByteCode)ModRMBYTE.lSource, baData, ControlUnit.currentCapacity);
        }
        // Otherwise, it is RM encoded(the destination is a pointer to memory)
        else
        {
            ControlUnit.SetMemory(ModRMBYTE.lDest, baData);
        }
    } else
    {
        // Check if the ModRM is MR encoded(the source and dest are swapped), in this case,
        // the source would be a register, in the source variable
        if (Swap)
        {
            ControlUnit.SetRegister((ByteCode)ModRMBYTE.lSource, baData);
        }
        // Otherwise the source is stored in the dest variable
        else
        {
            ControlUnit.SetRegister((ByteCode)ModRMBYTE.lDest, baData);
        }
    }
}
```

```

/*Dynamic functions turn..
* byte[] baResult;
byte[] baSrcData = ControlUnit.FetchRegister(bcSrcReg);
byte[] baDestData;

if (DestSrc.IsAddress)
{
    ulong lDestAddr = DestSrc.lDest;
    baDestData = ControlUnit.Fetch(lDestAddr, ControlUnit.CurrentCapacity);
    baResult = Bitwise.Add(baSrcData, baDestData, ControlUnit.CurrentCapacity);
    if (IsSwap)
    {
        ControlUnit.SetRegister(bcSrcReg, baResult);
    }
    else
    {
        ControlUnit.SetMemory(lDestAddr, baResult);
    }
}
else
{
    ByteCode bcDestReg = (ByteCode)DestSrc.lDest;
    baDestData = ControlUnit.FetchRegister(bcDestReg);
    baResult = Bitwise.Add(baSrcData, baDestData, ControlUnit.currentCapacity);
    if (IsSwap)
    {
        ControlUnit.SetRegister(bcSrcReg, baResult);
    }
    else
    {
        ControlUnit.SetRegister(bcDestReg, baResult);
    }
}
* ...into
* byte[] baSrcData = ControlUnit.FetchRegister(bcSrcReg);
byte[] baDestData = FetchDynamic(DestSrc);
SetDynamic(DestSrc, Bitwise.Add(baSrcData, baDestData, ControlUnit.CurrentCapacity), IsSwap);
*

```

Explanation

The FetchDynamic and SetDynamic methods are a temporary implementation of operand decoding procedures. They work by using the inputs necessary to perform frequently used procedures(used in every opcode) to perform the procedure in an opcode in very few lines of code by calling the method in the opcode api library.

Justification & Relation to break down of the problem

This is the best approach to a temporary implementation of the operands because it saves great amounts of development time because the specific tasks required to perform this procedure were changing very frequently because the methods in the CU were being developed, hence required more sophisticated inputs. This meant that code across many opcodes had to be rewritten, which caused more time to be spent maintaining code than developing the new features. In terms of the problem break down, building dependency on these functions will aid the transition to the IDecoded operand types as the opcode classes themselves are building dependency on a returned byte array, or a set method. The preconditions and inputs to these methods are very unlikely to change. This means that when operands classes are introduced, the only code that will have to be changed is the functions that are called instead of having to restructure each opcode class to work from a single procedure and not handle decoding on its own; each opcode class will already be developed to handle a byte array, so changing this now will save development time later as there will be many more opcodes by the time operands are ready to be implemented.

Review

Self review - This has greatly reduced the amount of work that has to be done to code an opcode, which will reduce the time taken to code and test an opcode allowing more time to be spent developing the essential features rather than repeating and maintaining code.

Peer review from a competent programmer friend of mine- "This is a step in the right direction towards meeting the open/closed principle as code is being prepared for the new interface classes. This means that you will only have to change a small amount of code when the time comes, not rewrite every opcode. It will also be possible to reuse this code in the operand classes as they are implemented; it is already tested so there is no reason not to. This would prove useful if another programmer was to write a module for your program as they will already have the methods they need to begin coding, the details such as "SetRegister()" and "FetchRegister()" have been abstracted entirely."

User interface

```

private void Step_Click(object sender, EventArgs e)
{
    // Execute one instruction
    Emulator.Step();
    // Refresh the UI to show the new results.
    _refresh();
}

```

Explanation

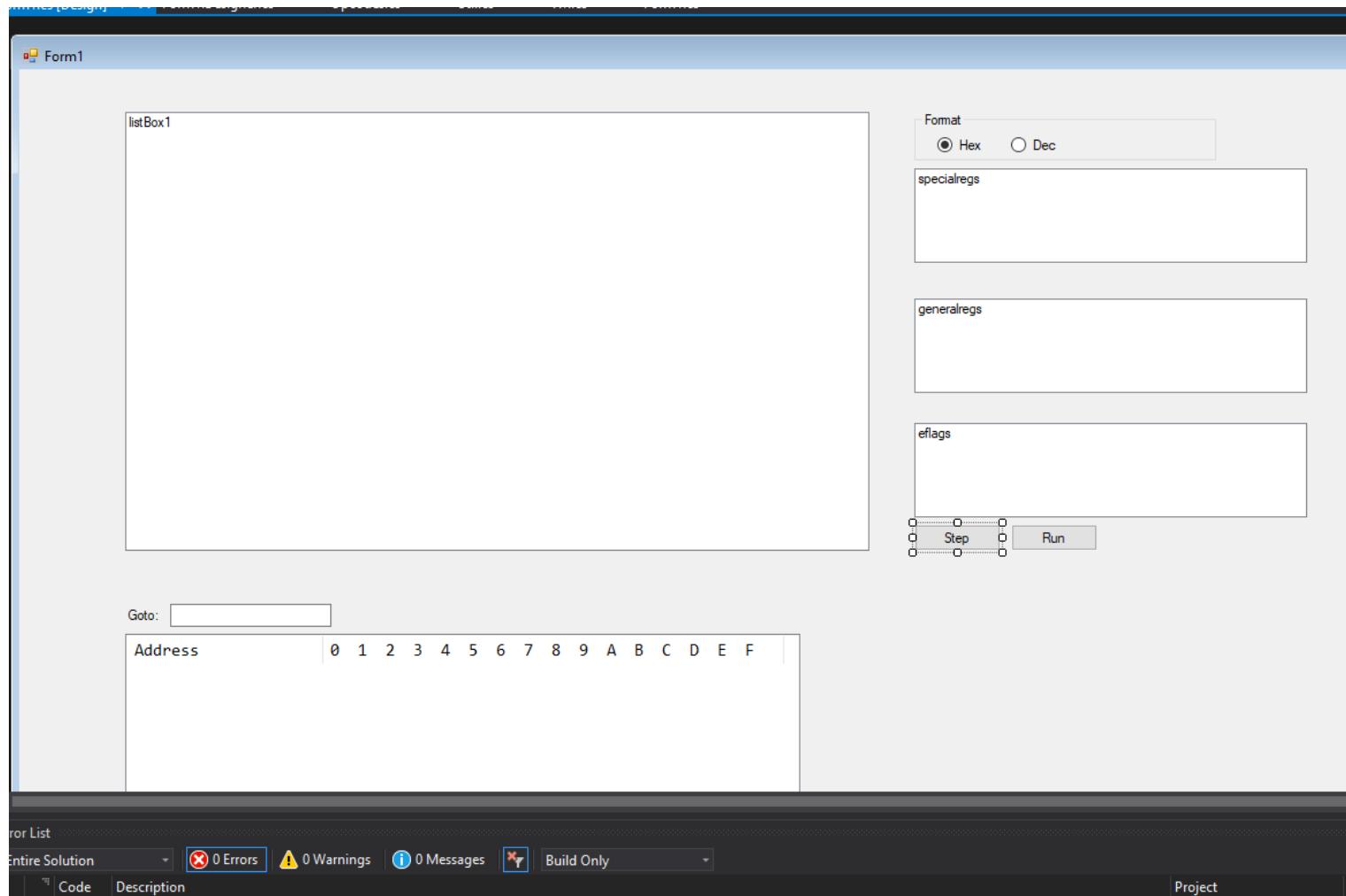
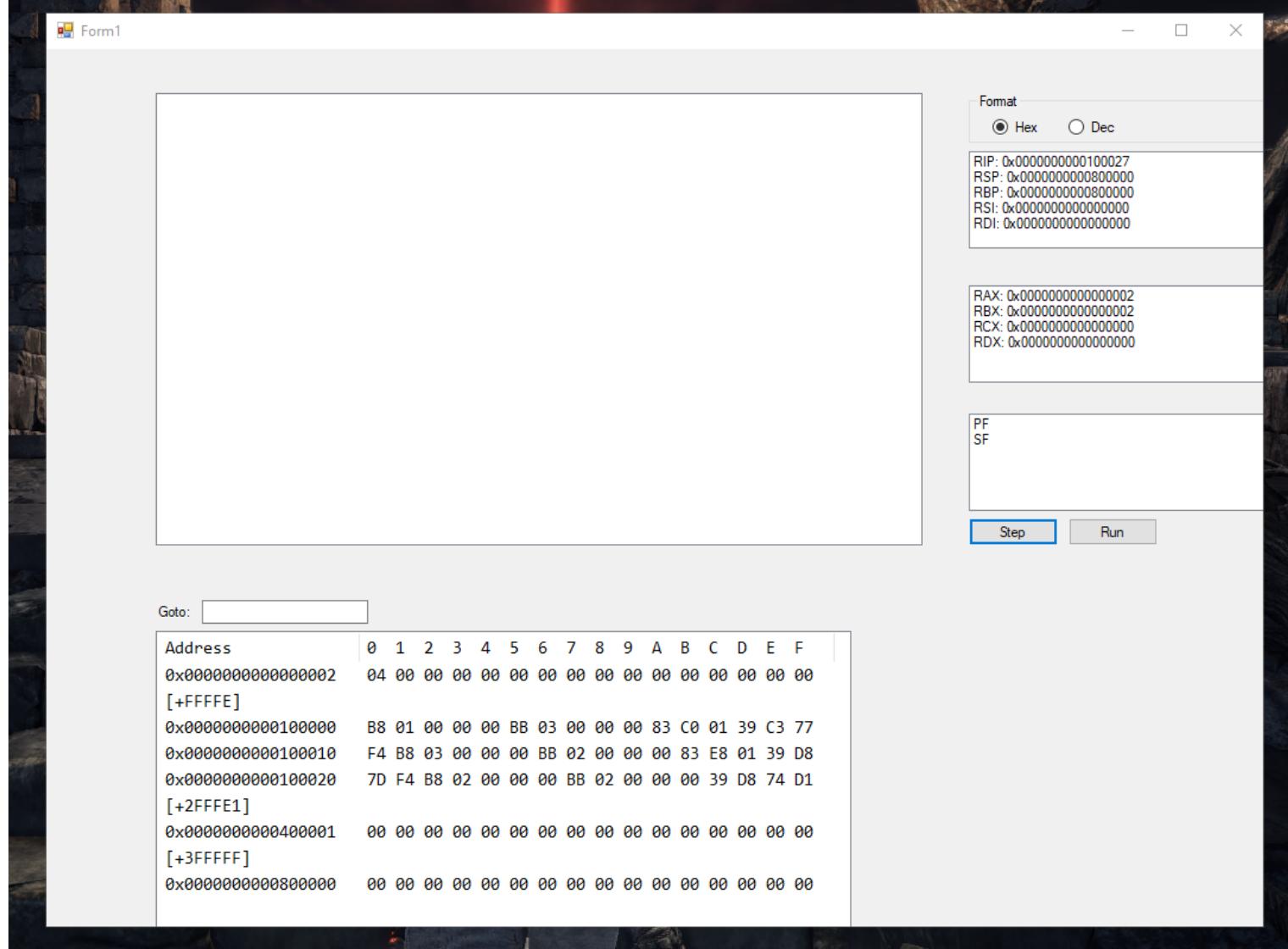
The user interface has started to be developed. The registers are displayed on the right, they can be displayed as either hexadecimal or decimal. Memory is displayed on the bottom left. Empty gaps in address space have been omitted as scrolling through a long listview became a chore. The gaps between addresses are shown between square brackets for easy interpretation. The step button will execute one instruction, the run button will execute to the end of the program or until a breakpoint is reached.

Justification & Relation to break down of the problem

This is currently the best approach to implementing the user interface because the methods are now available that automatically parse data from the processing layer such as registers. As it is too early in the development to commit to a specific layout, the material dark theme or theme engine has not been implemented as it is a feature that can be developed later in the project since it is more important to initially focus on the base functionality rather than the quality of life features as the program is useful to noone if it doesn't work. It was necessary to take this approach as it is the first implementation of the intermediary layer pipeline discussed in the design and analysis sections. The registers are first passed by the CU to the intermediary layer(the Emulator, VM class). The VM class parses the information into a high level format, a dictionary. This then allows the interface layer to process the data much more effectively as irrelevant details such as the RegCode have been abstracted at this point. This is important to include as it will aid in meeting the "Calling trees will be pipelined through layers without backflow where possible" success criterion. By implementing the pipeline early, there will be no need for a difficult transition later in the development when code is more rooted and harder to adapt due to dependencies on future modules.

Tests

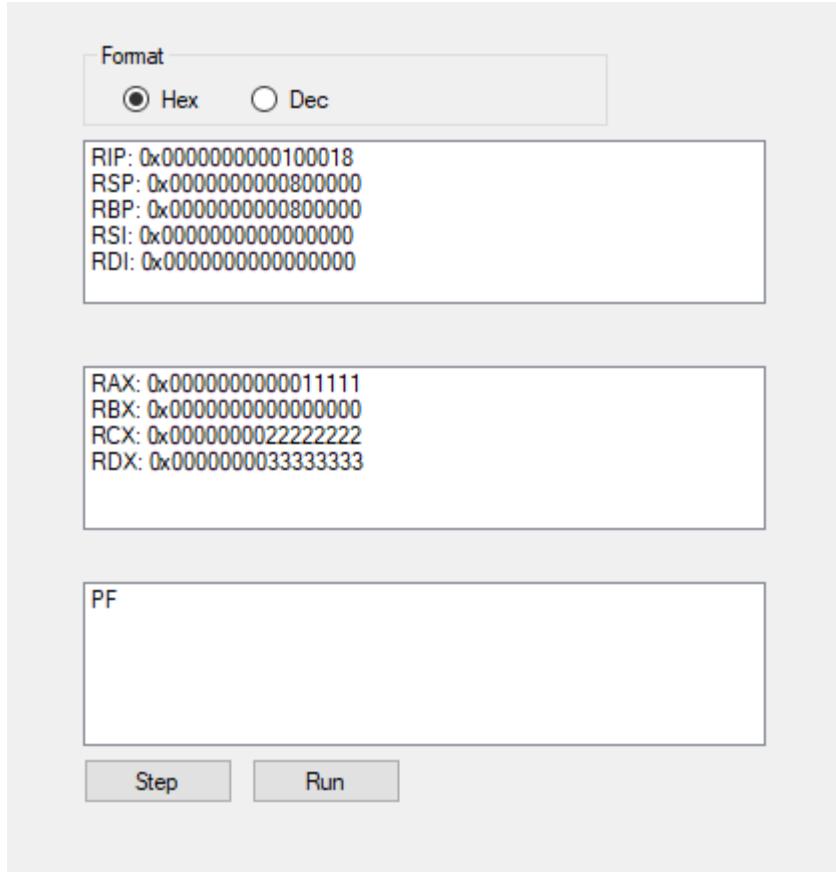
Test #	Aim	How	Test data	Normal /Erroneous /Boundary	Expected result	Justification
1	Crash test - Does the program open and layout correctly	The solution will be executed with no further input.	Will be determined during development as the layout has not been coded yet.	Normal	The program opens and does not crash, the layout of the controls is correct.	This test data is necessary as It is important to make sure that the form opens and draws properly before continuing testing as they would not be valid if the form did not work in the first place



Explanation

The first picture is the application running, the second is the application layout designed in the visual studio designer. The layout is perform correctly when launched, but I had to resize the window to fit it on the page. As the application works fine on launch, no remedial actions are required.

Test #	Aim	How	Test data	Normal /Erroneous /Boundary	Expected result	Justification
2	Test whether the registers display properly	A few instructions will be executed that cause registers to change.	add rax, 0x11111 add rcx, 0x22222222 add rdx, 0x33333333	Normal	The following information will be displayed onscreen RAX : 0x11111 RCX : 0x22222222 RDX : 0x33333333	This test data is necessary to make sure that registers are displayed correctly in the interface and that there are no errors in the process of doing so, which is important as displaying the registers was described as an essential feature. The instructions will be tested before this test is conducted to ensure they are working.



Explanation

The displayed results are the same as the expected results, hence the test passed, therefore no remedial action is required.

Test #	Aim	How	Test data	Normal /Erroneous /Boundary	Expected result	Justification
3	Test whether memory displays properly	A few instructions will be executed that cause memory to change.	add byte ptr [0x30], 0x11 add word ptr [0x50], 0x2222 add dword ptr [0x1000], 0x33333333	Normal	The following information will be displayed onscreen [0x30] 0x11 [0x50] 0x2222 [0x1000] 0x33333333	It is necessary to use this test data to test that memory is displayed properly as there is a success criterion stating that essential information will be displayed to the user. This means that is important to test that this feature is working in order to meet the success criteria. The instructions will be tested before this test is conducted to ensure they are working.

Goto:

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x000000000000000030 [+FFFFD0]	11	66	81	04	00	00	00	00	00	00	00	00	00	00	00	00
0x000000000001000000 [+2FFFF1]	80	04	25	30	00	00	00	11	66	81	04	25	50	00	00	00
0x000000000001000100 [+3FFFF]	22	22	81	04	25	00	10	00	00	33	33	33	33	00	00	00
0x000000000004000001	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x000000000008000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Explanation

This test did not pass. Extra data was copied to the memory location, data that should have been the next opcode but more than a single byte was read. However, this is very unlikely to be caused by the memory viewer and more likely to be an error in the mov opcode. This means that remedial actions are required to investigate further.

Remedial actions

This has been identified to be an issue in the structure of the opcode table. The function is not told that it is an 8 bit instruction until the constructor has executed.

```
Dictionary<int, Action> _80 = new Dictionary<int, Action>
{
    { 0, () => new AddImm(MultiDefOutput) { Is8Bit=true }.Execute() },
```

The Is8Bit boolean variable is set after the constructor executes. As a remedial action, the boolean is changed to be a parameter of the function.

```
{ 0, () => new AddImm(MultiDefOutput, Is8Bit:true).Execute() },
```

Now the function should only fetch one byte instead of four. To verify this, the test will be repeated.

Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000000000000030	11	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
[+20]	22	22	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0000000000000050	33	33	33	33	00	00	00	00	00	00	00	00	00	00	00	00
[+FB0]	33	33	33	33	00	00	00	00	00	00	00	00	00	00	00	00
0x0000000000010000	80	04	25	30	00	00	00	11	66	81	04	25	50	00	00	00
[+FF000]	22	22	81	04	25	00	10	00	00	33	33	33	33	00	00	00
0x0000000000100000	80	04	25	30	00	00	00	11	66	81	04	25	50	00	00	00
[+FFFF1]	22	22	81	04	25	00	10	00	00	33	33	33	33	00	00	00

The test now outputs the same results as the expected results, hence the bug is fixed, so no further remedial action is required.

Justification of remedial actions

Remedial action	Justification
Modifying the opcode table	This was necessary to solve the problem as the logical error in the code meant that the Is8Bit would not be passed to the function in time, such that the function would have already fetched the operands before the boolean is set. This means that the opcode table had to be modified in order to make sure that the boolean is passed in time by means of a parameter. This ensures that the class is using the correct information. Because of this, the intended function will now function since the correct amount of bytes are being read.
Re-testing the code	In case there was any further bugs that were not identified, the test had to be performed again to show that the requirements of the test were met; the expected results were displayed in the memory viewer. There could have been other bugs in the code aside from the identified problem which would have otherwise not been identified and would have been included in the version release.

Review

Reviewer	Comment
Self review	This is an appropriate solution to this sub-problem at this stage of the development because it will allow me to get early stakeholder feedback on how the design should be implemented and how much information is necessary. This communication with stakeholders will ensure that the expectations of the user interface stay grounded; it would be easy to go off track and create an over sophisticated user interface, which was shown to have many disadvantages on the target stakeholders in the analysis research section. The current interface is on track to meet the requirements of the success criteria; information displayed is neither excessive nor complex. There are additional features required that will be included in the upcoming versions.

Stakeholder, an existing Python programmer

"I like this GUI design because the purpose of every element is very clear. It feels very natural to have all the distinct groups of data together, such as registers, and at the same time does not feel crowded. The address gap interval shown in the memory viewer was very interesting, I have never before seen this kind of feature implemented. It can be very confusing to see many zeros filling the memory and very easy to get lost. This feature would definitely assist in developing assembly code as the outputs will be shown very clearly; there is no excess information such as environment variables bloating the memory. I think the next UI feature to be added should be the disassembler, it would be very helpful."

Developer preview version

Date: 12th August 2019

Release notes

- HypervisorBase added
- SIB bytes can be used to extend ModRM functionality allowing for more variations of addresses to be used.
- TestHandler. Automates the execution of testcases allowing for quick debugging. Useful for checking if modifying a piece of code caused it to break. Input/Output is in XML format.
- Contexts introduced. CU now exclusively uses contexts to operate from. They are used as both inputs and outputs to and from the CU.
- Handle context switching added. Hypervisors all use a handle to create their context on the CU.
- Disassembly implemented into UI. Instructions are automatically disassembled and displayed.
- Material dark theme introduced. Still under development, but a prototype will be useful for gaining user feedback.
- FlagSet data structure added. Some basic procedures included but needs more development..

```
public abstract class HypervisorBase
{
    protected Handle Handle { get; private set; }
    public string HandleName { get => Handle.HandleName; }
    public int HandleID { get => Handle.HandleID; }
    public delegate void OnRunDelegate(Status input);
    public delegate void OnFlashDelegate(Context input);
    public event OnRunDelegate RunComplete = (input) => { };
    public event OnFlashDelegate Flash = (context) => { };
    public HypervisorBase(string inputName, Context inputContext, HandleParameters handleParameters = HandleParameters.NONE)
    {
        // Automatically create a new handle for the derived class. The derived class should rarely have to worry about handles,
        // but for the scope of advanced usage in the future, it is left as protected. I don't wish to restrict the modular
        // capabilities of the program.
        Handle = new Handle(inputName, inputContext, handleParameters);
    }
    protected virtual void OnFlash(Context input)
    {
        // In general, it is good .NET convention to let the derived class be the
        // first to know of an event, such that it can be handled before an event
        // is invoked.
        Flash.Invoke(input);
    }
    protected virtual void OnRunComplete(Status result)
    {
        // See OnFlash()
        RunComplete.Invoke(result);
    }
    public virtual Status Run(bool Step = false) => Handle.Run(Step);
    public virtual async Task<Status> RunAsync(bool Step = false)
    {
        // Task.Run() is generally accepted as the best way to perform asynchronous tasks.
        Status Result = await Task.Run(() => Run(Step));

        // Raise event
        OnRunComplete(Result);
    }
}
```

HypervisorBase

```

public virtual void FlashMemory(MemorySpace input)
{
    // Create a new context from $input.
    // Automatically assign the stack and base pointer registers to the stack start.
    Context newContext = new Context(input.DeepCopy())
    {
        Registers = new RegisterGroup(new Dictionary<XRegCode, ulong>()
        {
            { XRegCode.BP, input.SegmentMap[".stack"].Range.Start },
            { XRegCode.SP, input.SegmentMap[".stack"].Range.Start },
        }),
    };

    // Use Handle to update the context with the new
    Handle.UpdateContext(newContext);

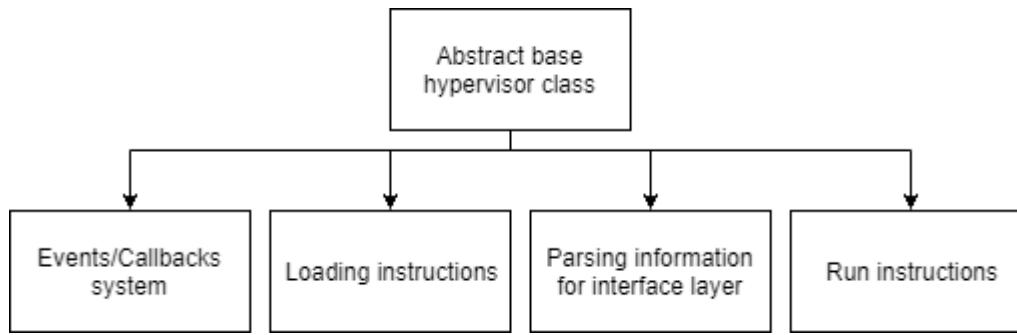
    // Raise OnFlash event
    OnFlash(newContext);
}

public Dictionary<string, bool> GetFlags()
{
    // Create a copy of the flags. ShallowCopy() a much better performance than DeepCopy(). As
    // FlagSet is a struct, there is no reason not to. Use ShallowCopy() where possible.
    FlagSet VMFlags = Handle.ShallowCopy().Flags;

    // Return string representations of each flag state.
    return new Dictionary<string, bool>()
    {
        {"Carry", VMFlags.Carry == FlagState.ON},
        {"Parity", VMFlags.Parity == FlagState.ON},
        {"Auxiliary", VMFlags.Auxiliary == FlagState.ON},
        {"Zero", VMFlags.Zero == FlagState.ON},
        {"Sign", VMFlags.Sign == FlagState.ON},
        {"Overflow", VMFlags.Overflow == FlagState.ON},
        {"Direction", VMFlags.Direction == FlagState.ON},
    };
}
public MemorySpace GetMemory() => Handle.ShallowCopy().Memory;

```

Structure explanation

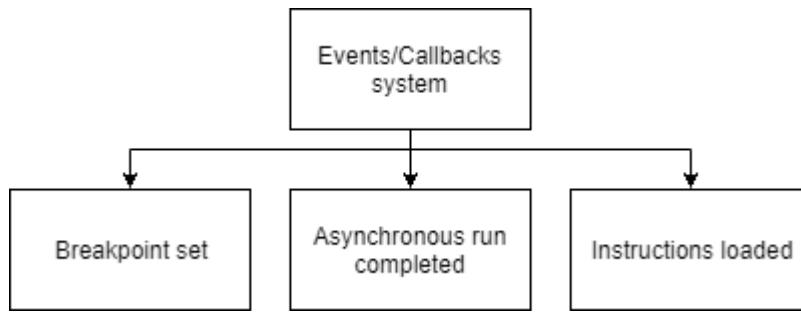


The HypervisorBase class has been coded using an abstract class. This means that it can only be inherited instead of being instanced

Structure justification

This was identified as a necessary measure in the design section. This design pattern will be clear to future maintainers that the intention is that the VM class is used for general purposes and HypervisorBase class is to be derived from for new hypervisor modules and reduces the overall code size as frequently used procedures can be coded in the HypervisorBase class hence only have to be tested once. This links to the Liskov substitution success criterion, as the procedures such as Run() will be usable in methods that interact with the HypervisorBase type instead of a specific derived class such that all subtypes of the HypervisorBase can be used in the method instead of only a single type. This will be useful when new modules are implemented post development as they can easily replace old modules without having to modify other modules.

Events/Callbacks explanation

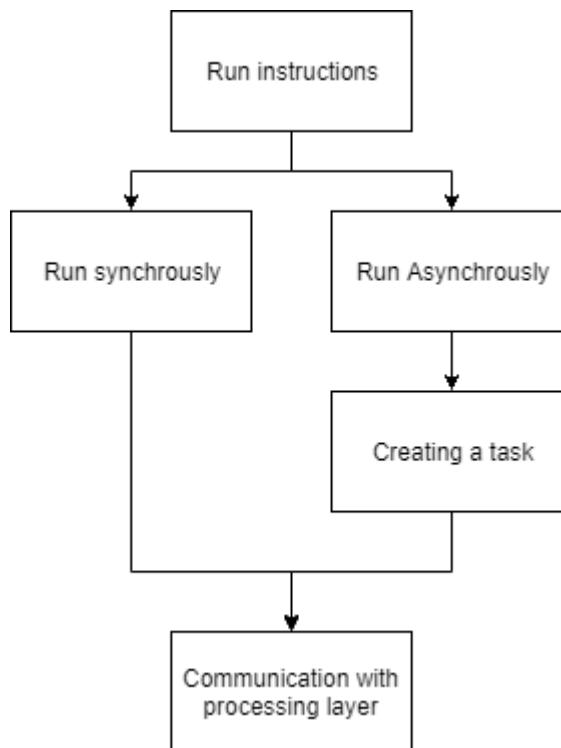


.NET events and delegates have been used to implement the callback system. An event listener can be set by the user which will trigger the callback when the event is invoked. The event is invoked during the `OnFlash()` and `OnRunComplete()` methods. A delegate is used to raise the event such that relevant information can be used as a parameter to the callback, for example, the context is passed as a parameter to allow the interface layer to update the UI with the new register/memory values. The breakpoint set event will be included in a future version because modules that require the feature have not been implemented yet, therefore it is a better choice to implement it as it is needed such that unforeseen preconditions do not affect the solution when the features need to be implemented.

Events/Callbacks justification

This subproblem was identified in the design section. The events system allows other classes to perform certain operations much more efficiently, such as waiting for a program to finish executing. This solution is suitable as it makes use of existing .NET types and methods to implement the events system such as the “event” keyword. This reduces the amount of testing and coding required as the methods required to implement this feature already exist. For example, events can be invoked by using `(event).Invoke()`. This allows more time to be spent on other features that do not already have solutions. Separating the events into two methods allows derived classes to override the methods and perform any necessary actions such as meeting preconditions that need to be performed before the event is raised to the listeners. This will be important to meet the Open/Closed principle because it will be possible to extend the functionality of a module that listens to the event without having to modify with that class directly. For example, if the run completed event is raised to update the UI with new information, the information can be parsed in the `OnRunComplete()` method before it reaches the interface layer

Run/RunAsync explanation



The `Run()` procedure allows any external class on the interface layer to start executing instructions without directly interacting with the handle such that the handle module could be swapped out at a later date and it would only need to be updated in the `HypervisorBase` class. The `RunAsync()` method creates a task to solve the same problem but will execute on a separate thread to the UI thread.

Run/RunAsync justification

This problem was identified in the design section. The subproblems of asynchronous/synchronous execution have been tackled separately using separate functions. This is the best approach because if a caller is going to use the result immediately after the function is called, there is no need to run it asynchronously as it will only create overhead coordinating the threads. For this reason, both methods can be used appropriately to maximise performance. This links to the “The program should be able to execute instructions in a timely manner.” success criterion. Executing instructions is the part of the program where the most processing is performed. Because of this, the best approach is to optimise it where possible such that the program remains accessible to users on any computer. It was identified in the stakeholder research analysis section that users have medium to high-end computers in an even distribution. By implementing these optimisations, the solution will remain accessible to the majority of stakeholders who have moderate performance computers. The subproblem of communicating with the processing layer has been tackled through the use of events. Each method will raise the `OnRunComplete()` method, which will then invoke the `RunComplete` event. This will be a necessary communication method for the interface layer to signal that new information(e.g. register, memory) has updated and needs to be fetched again to be shown to the user.

Loading instructions explanation

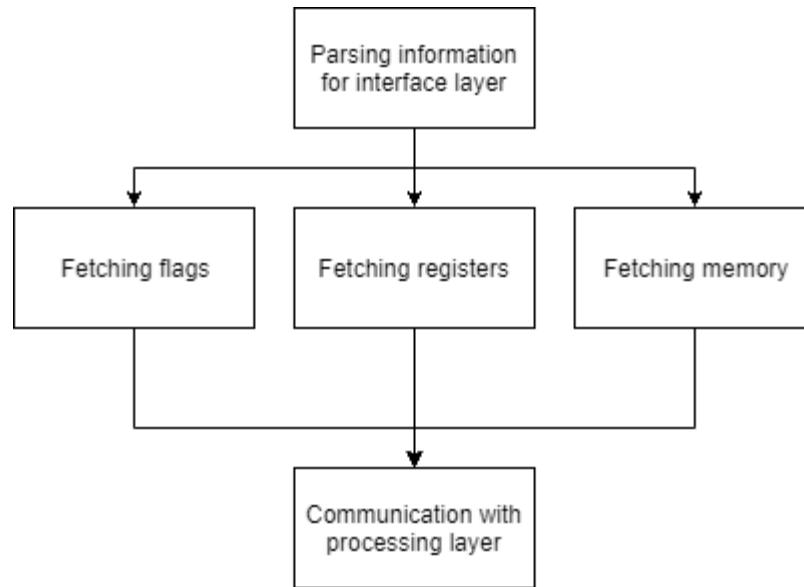
Instructions can be loaded using the `FlashMemory()` method. This will automatically handle the process of creating a context and assigning registers to their default values(The stack pointer is set to the top of the stack).

`Handle.UpdateContext()` is then used to perform a context swap.

Loading instructions justification

This task was identified in the design section. It is an important abstraction for removing complex and irrelevant details from the intermediary layer, such as handles and contexts. This will reduce the need for the interface layer to interact with the processing layer directly. This has been implemented as planned, the context is automatically created from a `MemorySpace` and the `Flash` event is raised. This will allow other modules to interact with each other using this event. For example, the disassembler will use the event to listen for new instructions being loaded into the VM. This links to the pipelining success criterion as the intermediary layer will provide more pipeline functions that divide large operations into smaller tasks such that a single method call can perform the complex procedure(creating a context, interaction with handle) without creating any dependencies on other modules and allowing the solution to be created in fewer lines of code.

Parsing information for interface layer explanation



Two functions in the `HypervisorBase` allow the interface layer to read information from the processing layer. The `GetFlags()` method works by copying the context from the handle and parses the `FlagSet`, returning the output as a dictionary of each flag name and a boolean. The `GetMemory()` method returns the `MemorySpace` stored in the context of the handle.

Parsing information for interface layer justification

This problem has been tackled as planned in the design section. Flags are parsed using names rather than mnemonics. This is important to the solution as it allows the information returned to the interface layer to be easier to understand than a mnemonic abbreviation such as “CF”. Using a dictionary as the output will allow the solution to be more maintainable as instead of using the `FlagState` enum for the output, an abstraction has been applied which removes the specific details of flag types, leaving only their name and state such that the UI can display them without having to unnecessarily handle irrelevant details. This links to the dependency inversion principle success criterion.

The method returns a dictionary class which is a very common high level class. This will allow a high level programmer who is not familiar with the rest of the program to contribute and maintain the interface layer without having to understand the processing layer.

SIB Bytes

```
public static SIB SIBDecode(int Mod)
{
    // Function purpose: Decode a SIB byte applying systematic parsing methods and return
    // the useful information to the caller in form of a SIB struct.

    // Convert the SIB byte into a string of bits that can be interpreted easier
    string SIBbits = Bitwise.GetBits(FetchNext());

    // This struct provides easily readable properties for masking the correct bits in a SIB byte to get the desired field.
    // This makes the code a lot more readable and reduces margin of error.
    // A SIB is constructed like,
    //
    // | 7 6 | 5 4 3 | 2 1 0 |
    // | SCALE | INDEX | BASE |
    //

    // So the byte 0xE2 translates to
    // Scale = 3
    // Index = 4
    // Base = 2
    // The Mod of the preceeding ModRM byte is also used to determine whether EBP is also added to the resulting pointer.
    SIB Output = new SIB()
    {
        Scale     = (byte)Math.Pow(2, Convert.ToByte(SIBbits.Substring(0, 2), 2)), // scale
        ScaledIndex = Convert.ToByte(SIBbits.Substring(2, 3), 2),
        Base      = Convert.ToByte(SIBbits.Substring(5, 3), 2),
        PointerSize = (PrefixBuffer.Contains(PrefixByte.ADDR32) ? RegisterCapacity.DWORD : RegisterCapacity.QWORD)
    };
    // If the index isn't 4, there is an index register that needs to be added to the SIB. Otherwise it is just a base or an immediate pointer.
    if (Output.ScaledIndex != 4)//4 == none
    {
        // To get the actual value of the index, multiply it by the scale, which is equal to 2^(scale bits value)
        // For example,
        // Scale bits == 3
        // 2^3 = 8
        // Index * 8
        Output.ScaledIndexValue = Output.Scale * BitConverter.ToInt64(Bitwise.SignExtend(FetchRegister((ByteCode)Output.ScaledIndex, Output.PointerSize),8),0);
    }

    // If the base isn't 5, there is a base register encoded in the SIB. The base being 5 denotes that there is only a pointer(which could be 0)
    if (Output.Base != 5) // 5 = ptr or rbp+ptr
    {
        Output.BaseValue += BitConverter.ToInt64(FetchRegister((ByteCode)Output.Base, RegisterCapacity.QWORD), 0);
    }
    // If the Mod bits of the preceeding ModRM byte do not equal 0 and the base bits equal 5, $EBP is used as the base
    else
    {
        if (Mod > 0) // mod1 mod2 = ebp+imm
        {
            Output.BaseValue += BitConverter.ToInt64(Bitwise.SignExtend(FetchRegister(ByteCode.BP, Output.PointerSize),8),0);
        }
        // The other case, _base != 5 and Mod == 0, means that there is an absolute pointer stored in the immediate 4 bytes.
    }
}

// The ModRM byte that called this constructor only uses the result of the SIB,
// but the rest is stored beyond that purely for disassembly.
Output.ResultNoOffset = Output.BaseValue + Output.ScaledIndexValue;
return Output;
```

Explanation

SIB byte handling has been implemented by using an algorithm to determine the properties of the SIB. There are certain general procedures that can be applied to determine the properties when conditions are met. For example, when `Output.Base != 5`, the SIB base is simply a pointer stored in a register. This means that the opcode api method `FetchRegister()` can be used to fetch the register stored in the base bits(which have equivalent values to `ByteCode/RegCodes`). This means that instead of having to hard code a large amount of conditions for each possible value, only one if-else condition is needed.

Justification & Relation to break down of the problem

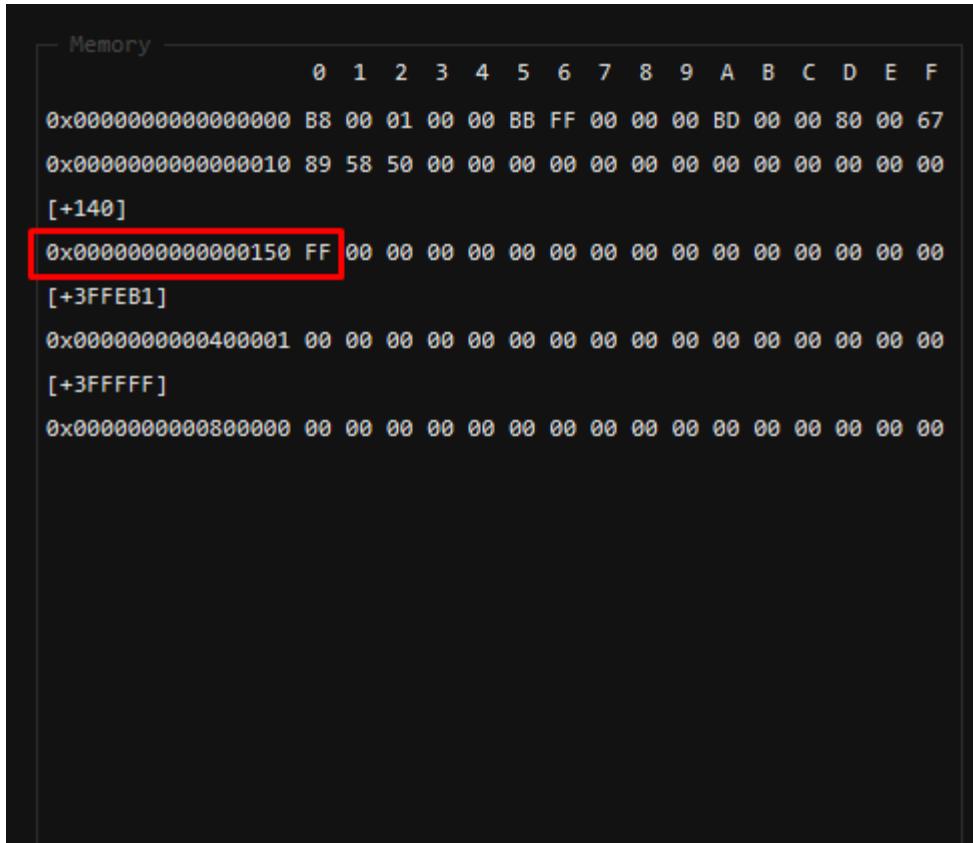
It was necessary to take this approach to the problem as it was identified in the design and analysis sections that the ModRM would require a separate class to perform the task of tackling SIBs. This has greatly abstracted the problem from the ModRM decoding procedure as only a small amount of code is required to fetch the outputs.

```
else if (Output.Mem == 4)//sib! sib always after modrm
{
    // Decode the SIB
    Output.DecodedSIB = SIBDecode(Output.Mod);
    // Store the offset(required for disassembly)
    Output.Offset = Output.DecodedSIB.OffsetValue;
    // Store the pointer to the destination parsed by the SIB
    Output.DestPtr = Output.DecodedSIB.ResultNoOffset;
}
```

It is not possible to abstract any more details because certain information is required for disassembly. For example, the immediate offset from the register. It would be confusing to have an instruction display that memory only pointed to a register is being modified, then modify memory at an offset. This would likely be confusing to the stakeholders. This links to the intuitive design success criterion. If users are expected to understand the inner workings of the class, this would not be intuitive. Therefore, this was the best approach to solving this problem in order to meet the success criteria.

Testing

Test #	Aim	How	Test data	Expected result	Justification
1	Test register plus offset	Test data containing specific instructions containing a ModMR byte that implies a register plus offset SIB operand.	(These three instructions will be used in every test) mov eax,0x100 mov ebx,0xff mov ebp, 0x800000 mov DWORD PTR [eax+0x50],ebx	<Memory offset="150">FF</Memory>	This test data is necessary as it will test the basic functionality of the SIB as only an offset is applied. An error in this test would indicate a fundamental logical error in the algorithm as opposed to a specific part of it. Because of this, less development time will have to be spent debugging and tracking down bugs as it will give a good starting point on where to start looking.



Explanation

The expected results were produced by the program when the test data was used as an input, therefore the test passed and no remedial actions are required.

Test #	Aim	How	Test data	Expected result	Justification
2	Test scaled register and offset	Test data containing specific instructions containing a ModMR byte that implies a scaled register plus offset SIB operand.	<pre>mov DWORD PTR [eax*2+0x0],ebx mov DWORD PTR [eax*2+0x100],ebx mov DWORD PTR [eax*4+0x20000000],ebx</pre>	<Memory offset="200">FF</Memory> <Memory offset="300">FF</Memory> <Memory offset="20000400">FF</Memory>	This test data is necessary as it will test the multiplication performed by the scale coefficient. The conditions for this operation to be performed are specific, making it a special case. For this reason, it is important to make sure that there is test data that proves the functionality of the special case in the algorithm.

Memory	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000000000000000000000000000 B8 00 01 00 00 BB FF 00 00 00 BD 00 00 80 00 67																
0x00000000000000000000000000000010 89 1C 45 00 00 00 00 00 67 89 1C 45 00 01 00 00 67																
0x00000000000000000000000000000020 89 1C 85 00 00 00 00 20 00 00 00 00 00 00 00 00 00																
[+1E0]																
0x0000000000000000000000200 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+100]																
0x0000000000000000000000300 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+3FFD01]																
0x0000000000000000000000400001 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+3FFFF]																
0x0000000000000000000000800000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+1F800400]																
0x000000000000000000000020000400 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																

Explanation

The expected results were produced by the program when the test data was used as an input, therefore the test passed and no remedial actions are required.

Test #	Aim	How	Test data	Expected result	Justification
3	Test multiple register addition with offset and scale	Test data containing specific instructions containing a ModMR byte that implies a base register added to a scaled register plus offset SIB operand.	<pre>mov DWORD PTR [ebx+eax*8],ebx mov BYTE PTR [eax+ebx*1+0x100],bl mov DWORD PTR [ebp+eax*2+0x10],ebx mov cl,BYTE PTR [ebp+eax*2+0x10]</pre>	<Memory offset="08FF">FF</Memory> <Memory offset="2FF">FF</Memory> <Memory offset_register="BP" offset="210">FF</Memory> <Register id="C" size="1">FF</Register>	This test data is necessary because it will test all the features provided by the SIB byte at once. This will allow test data to exist that is conclusive enough to assume that the SIB procedures will work in the general case.

Memory	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000000000000000000000000000 B8 00 01 00 00 BB FF 00 00 00 BD 00 00 80 00 67																
0x00000000000000000000000000000010 89 1C C3 67 88 9C 18 00 01 00 00 67 89 5C 45 10																
0x00000000000000000000000000000020 67 8A 4C 45 10 00 00 00 00 00 00 00 00 00 00 00 00																
[+2DE]																
0x0000000000000000000000000000002FE FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+602]																
0x000000000000000000000000000000900 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+3FF701]																
0x00000000000400001 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+3FFFFF]																
0x00000000000800000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																
[+210]																
0x00000000000800210 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00																

Explanation

There are two unexpected results as one expected result. This means that the test has failed and remedial action is required.

Remedial action

It is clear from the fact that the two expected results both contained a base register that is not EBP, that there is a bug in the code in selection statement that causes the bug to be present under certain conditions

```
// The base bits being 5 denotes that there is an immediate pointer following the SIB byte,
// this is the only way of hard coding a pointer to a specific location as of now. An immediate displacement in the ModRM and
// a SIB absolute address pointer are not mutually exclusive.
// If the base isn't 5, there is a base register encoded in the SIB. The base being 5 denotes that there is only a pointer(which could be 0)
if (Output.Base != 5) // 5 = ptr or rbp+ptr
{
    Output.BaseValue = BitConverter.ToUInt64(FetchRegister((ByteCode)Output.ScaledIndex, RegisterCapacity.QWORD), 0);
}
else
{
    // If the Mod bits of the preceding ModRM byte do not equal 0 and the base bits equal 5, $EBP is used as the base
    if (Mod > 0) // mod1 mod2 = ebp+imm
    {
        Output.BaseValue = BitConverter.ToUInt64(Bitwise.SignExtend(FetchRegister(ByteCode.BP, Output.PointerSize),8),0);
    }
    // The other case, $base != 5 and Mod == 0, means that there is an absolute pointer stored in the immediate 4 bytes.
    else
    {
        Output.OffsetValue += BitConverter.ToInt32(FetchNext(4),0);
    }
}

// The ModRM byte that called this constructor only cares about the destination
// of the SIB, but the rest is stored beyond that purely for disassembly.
Output.ResultNoOffset = Output.BaseValue + Output.ScaledIndexValue;
```

In the SIB decoding procedure, there is a condition that performs a different procedure if the register is EBP. By analysing the code, a logical error can be found.

```
if (Output.Base != 5) // 5 = ptr or rbp+ptr
{
    Output.BaseValue = BitConverter.ToUInt64(FetchRegister((ByteCode)Output.ScaledIndex, RegisterCapacity.QWORD), 0);
}
```

The scaled index is passed into the pipeline instead of the base register. This caused one of the registers in the SIB byte to be added twice, and the other not at all. This has been fixed by replacing "Output.ScaledIndex" with "Output.Base". To confirm that this change fixed the problem, the original test has been repeated.

```

Memory
0 1 2 3 4 5 6 7 8 9 A B C D E F
0x00000000000000000000000000000000 B8 00 01 00 00 BB FF 00 00 00 00 BD 00 00 80 00 67
0x00000000000000000000000000000010 89 1C C3 67 88 9C 18 00 01 00 00 67 89 5C 45 10
0x00000000000000000000000000000020 67 8A 4C 45 10 00 00 00 00 00 00 00 00 00 00 00 00
[+2DF]
0x0000000000000000000000000000002FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[+600]
0x0000000000000000000000000000008FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[+3FF702]
0x000000000000000000000000000000400001 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[+3FFFFFF]
0x000000000000000000000000000000800000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[+210]
0x000000000000000000000000000000800210 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

The outputs from the program now agree with the expected results.

Justification of remedial actions

Remedial action	Justification
Modifying the code to remove the logical error	It was necessary to take this remedial action in order to eliminate the problem. It was proven to be this part of the code causing the problem as after it was corrected, the results were now the same as the expected results. For this reason, it was necessary to take this remedial action in order to meet the test criteria.
Re-testing the code	It was a necessary action to retest the code after because it showed that the change to the code had fixed the bug in the code. If it was not re-tested afterwards, I could have been wrong initially, therefore there would likely be even more bugs present as working code would have been modified. Because of this, it was important to repeat the tests in order to ensure that the bug is not included in the latest version of the code as stakeholders would otherwise not be able to use SIB bytes which could disrupt their normal usage of the program, hence would find another program to use instead.

Review

Reviewer	Comment
Self review	This has been demonstrated as a functional feature in the program. As SIB bytes are part of the x86-64, this will contribute to the 1:1 emulation success criterion. It was also in the planned design. Implementing this early has allowed more progress towards success criteria and important abstractions that will improve development speed when other components are developed that require the SIB class functionality.
Stakeholder, an existing C# programmer	"I had never heard of SIB bytes before, however this class demonstrates their purpose very clearly. They are additional bytes that can be added on to the end of the ModRM byte to allow more control of the pointer, performing interesting operations such as multiplying a register to be part of the pointer all in one part of the instruction. The code uses very useful abstractions, such as breaking down the problem into separate fields instead of operating on a single SIB byte, which allowed variables with names to be used in place of positions in the byte such as Scale, Index, and Base. This would definitely assist in future maintenance as it would not be necessary to understand how the SIB byte is constructed in order to improve the decoding procedure since the algorithm is clearly separated and put at the start of the function code."

TestHandler: Using validation

```
static TestHandler()
{
    // Iterate each file in the Testcases folder
    foreach (string FilePath in Directory.GetFiles("Testcases"))
    {
        // Check if the file is an XML file(other files could cause errors)
        if (FilePath.Substring(FilePath.Length - 4) == ".xml")
        {
            // Use a try catch block as if any exceptions occur it will be assumed that
            // the file is invalid.
            try
            {
                // Load the file as a .NET XML object so more library methods are available.
                XDocument TestcaseFile = XDocument.Load(FilePath);

                // Iterate through every element of the XML object that is inside the root element
                // called "Testcase".
                foreach ( XElement InputTestcase in TestcaseFile.Elements("Testcase"))
                {
                    // Create a new TestcaseObject which will be used as input to the CU(is converted to a context later).
                    TestcaseObject ParsedTestcase = new TestcaseObject
                    {
                        // Use the validation method ParseHex() to attempt to parse the Hex element as bytes.
                        Memory = new MemorySpace(ParseHex(InputTestcase.Element("Hex").Value)) // memory = <hex>x</hex>
                    };
                    // Iterate through every element called "Checkpoint"
                    foreach (var InputCheckpoint in InputTestcase.Elements("Checkpoint"))//for each <checkpoint> in file
                    {
                        // Read the position_hex attribute, what the instruction pointer will be when the checkpoint is tested
                        // Essentially a breakpoint.
                        ulong Offset = Convert.ToInt64(InputCheckpoint.Attribute("position_hex").Value, 16);
                        // Create a new Checkpoint object to store the parsed checkpoint.
                        Checkpoint ParsedCheckpoint = new Checkpoint()
                        {
                            // Store the name of the checkpoint
                            Tag = InputCheckpoint.Attribute("tag").Value,
                        };
                        // Iterate through every sub-element of the checkpoint, e.g. memory tests and register tests
                        foreach ( XElement TestCriteria in InputCheckpoint.Elements())
                        {
                            // Check if the element is a register test
                            if (TestCriteria.Name == "Register") //<register>
                            {
                                // Attempt to read the attributes that describe the register
                                // "id" attribute is the mnemonic of the register.
                                // "size" attribute is the size of the register to test
                                // The Value property is the data between the tags that is the value that
                                // is expected to be read from the checkpoint, if successful.
                                // If any of these are not present, an exception will be caught by the
                                // outer try-catch block.
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
ParsedCheckpoint.ExpectedRegisters.Add(
    new TestRegister()
    {
        RegisterCode = RegisterDecodeTable[TestCriteria.Attribute("id").Value],
        Size = (RegisterCapacity)(int)TestCriteria.Attribute("size"),
        ExpectedValue = Convert.ToInt64(TestCriteria.Value, 16)
    });
}

// Check if the element is a memory test
else if (TestCriteria.Name == "Memory") //<memory>
{
    // Validate the element by checking if it has the required attribute
    // At least one of the following are required.
    if (TestCriteria.Attribute("offset") == null && TestCriteria.Attribute("offset_register") == null)
    {
        // An exception is thrown telling the user about the issue
        throw new Exception("Memory needs to have atleast an offset or offset_register attribute");
    }

    // Check if the offset is negative without a register to offset it. This would not be possible as
    // negative addresses do not exist, so validation must be performed here.
    if (TestCriteria.Attribute("offset") != null
        && Convert.ToInt64(TestCriteria.Attribute("offset").Value, 16) < 0
```

```

private static byte[] ParseHex(string hex)
{
    // Function purpose: Parse a string of hex characters into a byte array such
    // that they can have operations from other parts of the program performed,
    // such as addition.

    // There must be an even number of characters, e.g. "F" is not a byte alone,
    // but "0F" is implied. The parser needs to be able to understand this.
    if(hex.Length % 2 != 0) { hex = hex.Insert(0, "0"); }
    byte[] Output = new byte[hex.Length / 2];
    // Perform hex.Length/2 iterations because each byte is represented in two characters
    for (int ByteIndex = 0; ByteIndex < hex.Length/2; ByteIndex++)
    {
        // Convert the string at twice the index(because 2 characters were needed
        // per byte). The 16 parameter will specify that it is in base 16.
        Output[ByteIndex] = Convert.ToByte(hex.Substring(ByteIndex * 2, 2), 16);
    }
    // Return the parsed bytes.
    return Output;
}

```

```

    }
    else
    {
        // Throw exception if the input was neither memory nor register
        throw new Exception($"Unexpected item {TestCriteria.Name}");
    }
}
// Add the checkpoint to the testcase and the position to break to test the checkpoint
ParsedTestcase.Checkpoints.Add(Offset, ParsedCheckpoint);
// Add the testcase to the testcase list.
Testcases.Add(InputTestcase.Attribute("name").Value, ParsedTestcase);
}
catch (Exception e)
{
    // Alert the user that there was an IO related exception.
    MessageBox.Show($"Error reading testcase file {FilePath}:\n{e.ToString()}");
}
}

```

Explanation of validation

Validation is used by the TestHandler class to ensure that the XML files that are input to the program have the necessary elements required to parse the file as a testcase. This also includes making sure there are no unwanted elements in the XML file as they could cause erroneous behaviour. A foreach loop is used to iterate through every file in the testcase folder. At the start of the loop certain validation procedures must be applied. The file extension will be validated to make sure that the file in question is an XML file, as errors could occur if an incorrect file is loaded. The rest of the code is part of a try catch statement. This is because the XElement object is very error prone; it performs very little error checking itself. If an element is not present and is attempted to be accessed through an XElement, an exception will occur. For this reason, the code must be wrapped in a try catch block. The user is displayed a messagebox afterwards if there was an error stating that the testcase file was invalid and printing the error(without closing the program).

Justification & Relation to break down of the problem

It is necessary to perform this validation as it was identified as a problem in the design section. It was proven that there would be many opportunities where there could be parsing errors, IO errors, or invalid inputs in the XML elements. Because of this, the necessary procedures to implement the validation need to be implemented. The best place to perform this validation is in the TestHandler itself because it can be performed as the file is loaded. This means that the user can be immediately told at start up of any invalid testcase inputs in the testcases directly. It was identified in the analysis section that the procedure to perform the validation could be reused from a library. As the IO class has not yet been implemented, the library method was coded in this version, called ParseHex(). This will allow the method to be reused when the IO class is implemented.

Contexts

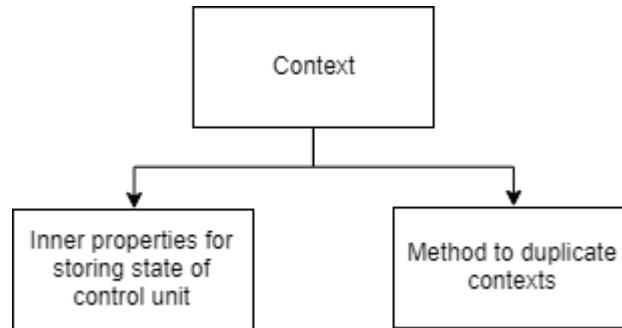
```

public class Context
{
    // Necessary variables required to execute on the CU
    // Most data structures are capable of constructing
    // themselves; others require additional inputs.
    public FlagSet Flags = new FlagSet();
    public MemorySpace Memory;
    public RegisterGroup Registers = new RegisterGroup();
    public ulong InstructionPointer;
    public List<ulong> Breakpoints = new List<ulong>();
    public Context(MemorySpace memory)
    {
        // Set up the reference to store the memory. This is the
        // only required parameter as it would be acceptable to have
        // any other variable be automatically assigned, but there
        // is no good reason to have a context without memory as the
        // context is a means of loading memory into the CU.
        Memory = memory;
        // Begin the instruction pointer at the entry point of the memory
        // such that the CU has somewhere to begin executing
        InstructionPointer = Memory.EntryPoint;
    }

    private Context(Context toClone)
    {
        // Create a value copy of the variables
        // This means that their value is separated
        // from their reference, such that they can be
        // modified by the caller without affecting the
        // classes elsewhere in the solution
        Flags = toClone.Flags.DeepCopy();
        Memory = toClone.Memory.DeepCopy();
        // Ulongs are value types; Deep copies are implied.
        InstructionPointer = toClone.InstructionPointer;
        Breakpoints = toClone.Breakpoints.DeepCopy();
        Registers = toClone.Registers.DeepCopy();
    }
    public Context DeepCopy()
    {
        return new Context(this);
    }
}

```

Explanation



Contexts have been implemented by creating an empty data structure that holds variables that are more useful when together. This is because together they contain all the necessary information required to start executing instructions in the CU. The DeepCopy procedure will produce a copy of the context that acts like a struct; a copy of the context is disassociated from its old reference. This allows more complex procedure such as disassembly to take place, where the disassembler has to deep copy the context of the VM class in order to execute its instructions, or otherwise the VM would have its instructions executed immediately after the program loads because the Disassembler class would be executing on the same MemorySpace.

Justification & Relation to break down of the problem

This is the best approach to implementing the context as it uses the justified abstractions and details explained in the analysis and design sections. This has been implemented by the use of variables in place of the complex details that the CU requires to operate. This is a result of solving the inner properties subproblem in the design section, where the details of how exactly the CU starts is abstracted from the caller, such that all the caller must concern themselves are

there relevant and necessary inputs to produce a context. This has allowed the dependency inversion principle success criterion to be met, as instead of callers being required depending on specific details such as moving MemorySpaces and RegisterGroups by their own means, the methods has been generalised by instead transporting a Context. This means that instead of the CU depending on individual classes, it will instead depend on the more modular container class Context, which will allow the components to easily be swapped out as future maintenance and development may require. The method to duplicate context solution was implemented by the deep copy. This allows copies of a context to be produced such that they can be modified without affecting the original variable. This was the best approach to implementing the solution because it reuses copying procedures that have already been made in the relevant classes, and removes the need to create a specific copy routine for the instruction pointer because it is a value type, hence passed by value regardless.

Handle

```
public struct Status
{
    // Information about why the CU executed, will be used in the
    // future to indicate an error
    public enum ExitStatus
    {
        BreakpointReached
    }
    // Initialises to breakpoint reached because it defaults to the first enum member.
    public ExitStatus ExitCode;
    // Store the last disassembled instruction in the status
    public string[] LastDisassembled;
}

public static bool IsBusy
{
    // A thread safe property for knowing whether the ControlUnit is in use by another handle.
    // This is achieved through locking and therefore returning the thread constant value of $busy.
    // This prevents other hypervisors from accidentally taking over the control unit whilst another
    // hypervisor is using it.
    get
    {
        lock (ControlUnitLock)
        {
            return _busy;
        }
    }

    private set
    {
        lock (ControlUnitLock)
        {
            _busy = value;
        }
    }
}
private static void WaitNotBusy()
{
    // A method for handles to wait until the ControlUnit is free before executing. This allows a handle to wait
    // until another is finished stepping before executing. If a handle was half way through execution and
    // $ControlUnit.CurrentHandle changed, the already running handle would now affect the new $CurrentHandle rather than its intended.
    // This avoids that scenario.
    while (IsBusy)
    {
        Thread.Sleep(10);
    }
}
```

```

// A locking object(as a value type such as a boolean cannot be locked) to ensure the value of _busy is consistent across threads.
private static readonly object ControlUnitLock = "L";
// The private value for the public property IsBusy.
private static bool _busy = false;

public struct Handle
{
    // A dictionary that pairs all existing handles with a context.
    private static Dictionary<Handle, Context> $StoredContexts = new Dictionary<Handle, Context>();

    private static int NextHandleID = 0;
    private static int GetNextHandleID
    {
        // A private property for accessing $NextHandleID such that it is incremented every time to avoid Handle ID collisions.
        get
        {
            NextHandleID++;
            return NextHandleID;
        }
        set
        {
            NextHandleID = value;
        }
    }
}

// Readonly variables that allow other classes to identify a handle and its behaviour.

// Readonly variables that allow other classes to identify a handle and its behaviour.
public readonly string HandleName;
public readonly int HandleID;
public readonly HandleParameters HandleSettings;
public Handle(string handleName, Context inputContext, HandleParameters inputSettings)
{
    // A constructor to make a handle out of a given context.
    HandleName = handleName;
    HandleID = GetNextHandleID;
    HandleSettings = inputSettings;

    // If the StoredContexts dictionary already contains an identical handle , adding it again would throw an error.
    if ($StoredContexts.ContainsKey(this))
    {
        $StoredContexts[this] = inputContext;
    }
    else
    {
        $StoredContexts.Add(this, inputContext);
    }
}

public void UpdateContext(Context inputContext)
{
    WaitNotBusy();
    IsBusy = true;
    $StoredContexts[this] = inputContext;
    IsBusy = false;
}

public static Context GetContextByID(int id)
{
    // Iterate through contexts searching for a particular id. Return null if not found.
    foreach (var KeyPair in $StoredContexts)
    {
        if (KeyPair.Key.HandleID == id)
        {
            return KeyPair.Value;
        }
    }
    return null;
}

public void Invoke(Action toExecute)
{
    // A very useful method for when creating new hypervisors, as usage of this function will likely be what differentiates one from another.
    WaitNotBusy();
    IsBusy = true;
    toExecute.Invoke();
    IsBusy = false;
}

// Return a Context that can be modified without changing the actual context.
public Context DeepCopy() => $StoredContexts[this].DeepCopy();
// Return a reference to an existing Context such that changes to the returned context are reflected in the actual context.
public Context ShallowCopy() => $StoredContexts[this];

```

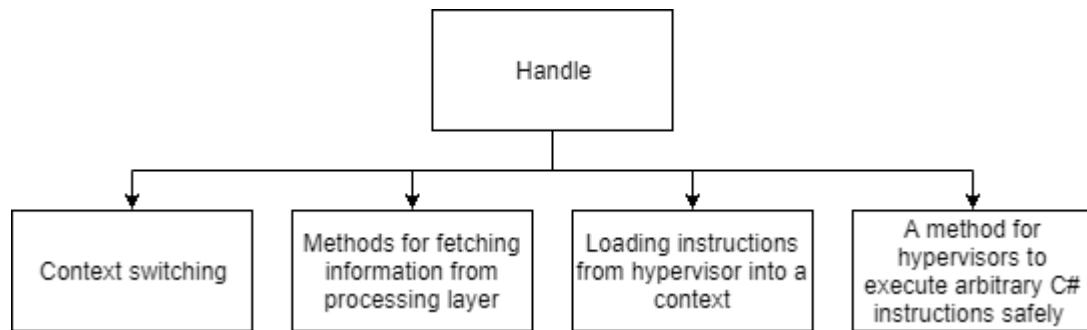
```
public Status Run(bool step)
{
    // A method for telling the ControlUnit to execute the handle. Public access to ControlUnit.Execute() would
    // not be advisable because the handle has to ensure that the ControlUnit isn't already in use and that
    // $ControlUnit.CurrentHandle is equal to the desired handle.

    WaitNotBusy();
    IsBusy = true;

    // Set this handle to be the current handle in the ControlUnit.
    CurrentHandle = this;

    // Return the $Result status struct that Execute() returned.
    Status Result = Execute(step);
    IsBusy = false;
    return Result;
}
}
```

Structure explanation

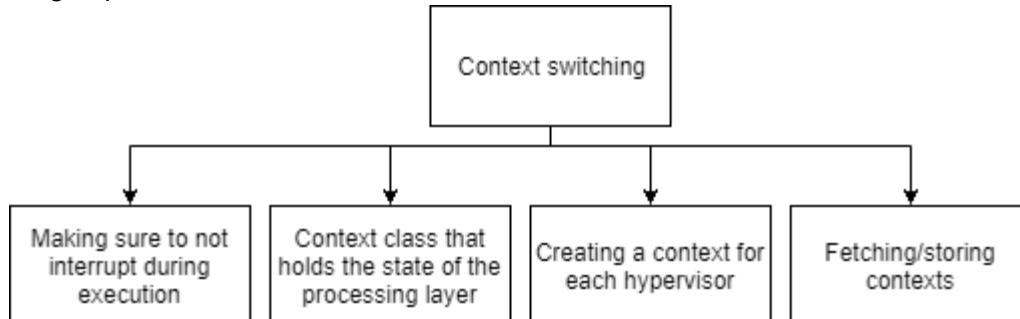


The handle class is the implementation of the pipeline of communication between the intermediary layer and the processing layer. It contains methods that can be used by modules to safely interact with the CU. Each problem has been tackled using a separate function.

Structure justification

It was proven important to separate the problems in the design section. This is because the intermediary api alone does not provide enough control over the CU and can be inconvenient at times. It was necessary to solve this problem using a data structure because there are certain variables such as HandleID that are required to coordinate context switching. The best approach to implementing this was to perform each task in a separate function, but also solve all other subproblems within that function. This allowed for irrelevant details from the ControlUnit such as the FlagState enum to be abstracted from the interface layer. This links to the pipeline success criterion as the interface layer will not be required to interact with the processing layer directly because the necessary methods to perform the complex procedures are already provided in the handle, where private variables are used to hide the irrelevant details from the caller class.

Context switching explanation

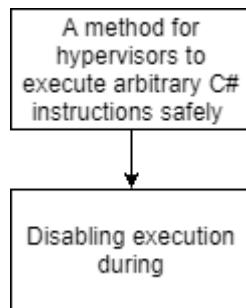


To perform a context switch, the StoredContexts dictionary has been used. The dictionary can be accessed by handle key to return the reference to the desired context. This allowed seamless transition between contexts because only the CurrentContext variable in the ControlUnit had to be changed in order to switch a context instead of having to change the value of each individual variable. The problem of making sure not to interrupt during execution has been tackled by the WaitNotBusy() method. This was identified as a subproblem in the design section and has been implemented as planned. The method will enter a while true loop that waits until the IsBusy variable is false, indicating that the CU is now available to be used.

Context switching justification

This problem was identified in the design section and has been solved by combining the problems and forming a suitable computational solution to the problem. This has been implemented by using references stored in a dictionary instead of moving large amounts of data in memory. This is the best approach to the problem because the solution is simpler and abstracted, there are no complex procedures required to implement the solution. This links to the single responsibility principle success criterion. Instead of requiring complex procedures across the processing and intermediary layer, the problem has instead been tackled using a single solution in the Handle class. This meets the requirements of the criterion because the problem has been contained to one class such that the solution will never need to be changed in the future because it has been separated into an individual function where it can solve the problem without being affected by other modules.

Method for executing arbitrary code safely explanation



This has been implemented using the `Invoke()` method. The parameter is an `Action` object, which in .NET is a function stored in a variable that can be executed when needed programmatically. This allows the `WaitNotBusy()` method to be reused in this function to ensure that the code is not executed while another thread is using the CU and also make sure that other threads do not incidentally begin executing instructions whilst the invoked action is still executing.

Method for executing arbitrary code safely justification

This is the best approach to implementing the solution to the subproblem because it allows already existing components to be reused, such as the `WaitNotBusy()` and `IsBusy` mechanism. This means that the method will require less testing because the reused methods will have already been tested, so there are less places for bugs to be present. This links to the single responsibility design principle success criterion, because the `WaitNotBusy()` method will handle all extra details that are not part of the particular problem such that the main details of the problem can be focused on in the method. This means that the method will not need to change as the solution develops because the problem has been fully solved such that its methods will not need to be updated to meet new criteria. This solution has also allowed the problem of fetching information from the processing layer to be solved. This is because intermediary layer classes will be able to use the `Invoke()` method to execute their own code to safely access the variables stored in the ControlUnit such as the memory and registers. This links to the pipelining process design criterion because the intermediary layer modules will be able to safely perform more complex procedures on behalf of the interface layer. This will remove any need for the interface layer to use processing layer because enough control is given on the intermediary layer such that the function can be included in an intermediary layer class such as a hypervisor then reused throughout the interface layer and at the same time abstracting any complex and irrelevant details in the process such as the `WaitNotBusy()` method and `IsBusy` mechanism.

Run explanation

The `Run()` function was a subproblem identified in the design section and has been implemented based on the pseudocode that was designed. The function works by using `WaitNotBusy()` to determine whether the CU is in use (and wait until it is not), then set `IsBusy` such that other functions can detect that the CU is in use. Selection is then used to ensure that the handle is the `CurrentHandle` in the CU, such that the opcodes are executed from the desired Handle's context, not another handle's context. After the `execute` method completes, the result is returned. The `Status` struct is an abstraction of the outputs of the CU that removes unnecessary details from the CU such as the status of the rex bytes and provides a simplified view of the CU.

Justification & Relation to break down of the problem

As planned in the analysis and design sections, the run method makes use of the `WaitNotBusy()` method. This will allow the hypervisors in the intermediary layer to perform more other operations while the `WaitNotBusy()` method executes on a background thread. The `Status` struct has been implemented to provide an abstraction of the outputs that the CU produces. This has allowed many irrelevant details to be abstracted, such as the result of the instruction, as this can already be found in memory. The details which are relevant to this part of the solution such as the disassembly of the instruction have been contained in the struct such that the methods in the intermediary layer will be able to parse the outputs as part of the pipeline process before returning them to the interface method. This will allow the pipeline design success criterion to be met because it will be ensured that the program flow is in one direction, the information is passed from the CU to the intermediary layer to be parsed, then returned to the interface layer. This will remove interdependence between layers, allowing for easier maintenance and improvements to large classes in the future as instead of depending on the specific classes, they will only depend defined on the abstracted inputs and outputs such as the `Status` struct.

Testing

Test #	Aim	How	Test data	Expected result	Justification
1	Test the process of waiting for the CU to no longer be in use	The Busy boolean will be set to true for a measured time period, and during the time period another thread will call WaitNotBusy().	<pre> Busy = true new Thread(Wait(10s); Busy = false;); // This will execute whilst // the previous thread waits WaitNotBusy(); // An obvious indicator that // will show whether it took // 10 seconds or not print("Done!"); </pre>	"Done" will be printed after ten seconds, not immediately after.	This test data is necessary because there will be many occasions where threads will be acting independently of one another. This will lead to multiple threads trying to use the CU at the same time. This test data will show whether the threads are coordinated properly to prove that the next thread will wait for the Busy boolean to be false before interacting with the CU, which is tested using two separate threads, one which sets busy to false after a timer(pretending that the CU is in use) and the other to wait for the other thread to do so.

```

public VM(VMSettings inputSettings, MemorySpace inputMemory) : base("VM", new Context(inputMemory)) {
    // Automatically set up stack pointer registers to start of stack
    Registers = new RegisterGroup(new Dictionary<ByteCode, Register>()
    {
        { ByteCode.SP, new Register(inputMemory.SegmentMap[".stack"].StartAddr) },
        { ByteCode.BP, new Register(inputMemory.SegmentMap[".stack"].StartAddr) } },
    });
    // Constructor class for VM

    // Store the settings input.
    CurrentSettings = inputSettings;
    // Assign the run complete event to the callput in the input
    RunComplete += CurrentSettings.RunCallback;
    // Clone the memory such that it can be used in other methods without affecting
    // the VM(which will cause errors if happens during execution, so is best to
    // create a deep copy)
    SavedMemory = inputMemory.DeepCopy();

    // The function created to perform the test, waiting the 10000 ms delay
    // on another thread whilst using the handle(this method will set IsBusy true)
    Task.Run(
        new Action(() =>
    {
        Handle.Invoke(new Action(() => Thread.Sleep(10000)));
    }));
}

```

Screenshot of a debugger interface showing assembly code, registers, memory dump, and a stopwatch overlay.

Registers:

RIP	0 0000000000000000
RAX	0 0000000000000000
RCX	0 0000000000000000
RDX	0 0000000000000000
RBX	0 0000000000000000
RSP	0 0000000008000000
RBP	0 0000000008000000
RSI	0 0000000000000000
RDI	0 0000000000000000

Memory dump (Hex dump of memory starting at address 0x0000000000000000):

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000000000000000	B8	00	01	00	00	BB	FF	00	00	BD	00	00	80	00	67
0x0000000000000010	89	1C	C3	67	88	9C	18	00	01	00	00	67	89	5C	45
0x0000000000000020	67	8A	4C	45	10	00	00	00	00	00	00	00	00	00	10
[+3FFE1]															
0x00000000000000400001	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
[+3FFFFF]															
0x00000000000000800000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Stopwatch overlay (DuckDuckGo stopwatch):

TOTAL: 00:06.58 LAP: 00:06.58

Buttons: STOP, RESET, LAP

Screenshot of a debugger interface showing assembly code, registers, memory dump, and a stopwatch overlay.

Registers:

RIP	0 0000000000000000
RAX	0 0000000000000000
RCX	0 0000000000000000
RDX	0 0000000000000000
RBX	0 0000000000000000
RSP	0 0000000008000000
RBP	0 0000000008000000
RSI	0 0000000000000000
RDI	0 0000000000000000

Memory dump (Hex dump of memory starting at address 0x0000000000000000):

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000000000000000	B8	00	01	00	00	BB	FF	00	00	BD	00	00	80	00	67
0x0000000000000010	89	1C	C3	67	88	9C	18	00	01	00	00	67	89	5C	45
0x0000000000000020	67	8A	4C	45	10	00	00	00	00	00	00	00	00	00	10
[+3FFE1]															
0x00000000000000400001	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
[+3FFFFF]															
0x00000000000000800000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Stopwatch overlay (DuckDuckGo stopwatch):

TOTAL: 00:11.74 LAP: 00:11.74

Buttons: STOP, RESET, LAP

```
// Instance VM class(calls constructor)
VMInstance = new VM(settings, ins);
// Instance disassembler immediately afterwards
Disassembler TestDisassembler = ... Disassembler(VMInstance.Handle);
```

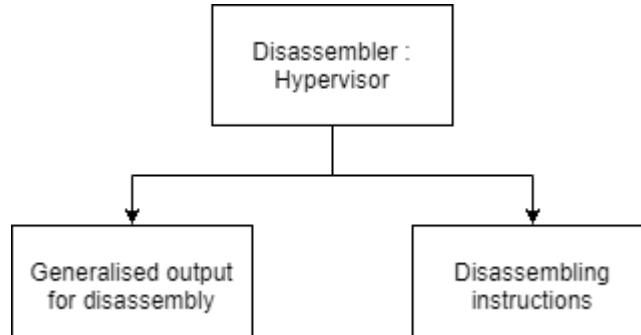
Explanation

Instead of printing done, this test can be shown to work using existing components(which proves more clearly that the method works in a realistic context).The VM class constructor is executed first. This is where the ten second timer starts on a separate thread(using Task.Run()). This means that the current thread that started the new thread continues executing and returns to the caller, where the disassembler class is instanced in the next line of code. The disassembler uses the CU to disassembly instructions. This means that it has to call WaitNotBusy() to check that it is not in use(which is called in the handle methods shown already). Therefore, disassembly should appear after the 10 second wait interval, which is shown in the pictures(The minor delay is because I had to take a screenshot plus delays between starting the stopwatch and the program, but is still very clear that the extra wait was because of this). As the expected results were produced, no remedial action is required.

Review

Reviewer	Comment
Self review	The handle object has met the existing success criteria for the code. It incorporates modularity, allowing independent modules to interact with the context switching process whilst having any complications abstracted entirely.
Experienced OO programmer 1	"You've done a great job here, [redacted]. I really like how the handle allows the invoke feature. I think this is a good example of meeting the open/closed principle discussed earlier. It would allow a new module to interact with the control unit, but doesn't require any particular modification to the control unit class code to do so."
Experienced OO programmer 2	"This class is a great abstraction of the low level concepts that this project covers. It seemed very daunting at first, but seeing the breakdown from this class really shows how you've abstracted the low level concepts entirely from the upper "interface" layer, and instead provided these methods that do all of the heavy lifting, but still allow lots of control for modules in the future. I think in the next versions I would definitely like to see more user-level features, such as more opcodes, but I understand that a good foundation to do so is required."

Disassembler



Disassembler explanation

The solution to this problem has been implemented by creating a separate class to handle all disassembly separately. The disassembling instructions subproblem has been tackled in an individual function that uses a data struct to abstract irrelevant details from the problem.

```
// When the target VM calls flash(), the context reference will change, so it is necessary to listen for this and update accordingly.
target.Flash += UpdateTarget;

public void UpdateTarget(Context targetMemory)
{
    // UpdateTarget is called when the VM instance calls FlashMemory(). This means that there are new instructions that need to be
    // disassembled, so the disassembler will, in this particular order,
    // -Clear what it already has.
    // -Flash the new instructions of the VM
    // -Disassemble the new range.
    // From this explanation it is clear why this order is needed, as the first 3 methods set up the preconditions for disassembly
    // to take place.
    // It can also be called by an external class, as there may be a time where this is necessary.
    ParsedLines.Clear();
    FlashMemory(targetMemory.Memory.DeepCopy());
    DisassembleAll();
}

public struct ParsedLine
{
    // ParsedLine is the intermediary struct for parsed DisassembledLines from ControlUnit. Mostly it is responsible for
    // the concatenation of the list of disassembled mnemonics.
    public string DisassembledLine;
    public AddressInfo Info;
    public ulong Address;
    public int Index;
}

private HypervisorBase Target;
```

The target variable is used to hold a reference to the HypervisorBase that will have its instructions disassembled. Events are used to listen for important signals from the target class that need to be handled by the disassembler. The Flash event is set because the disassembler will need to fetch the newly inserted instructions and disassemble them. This calls the UpdateTarget() procedure which clears the existing disassembly(stored in ParsedLines), copies the new memory into the context, then disassembles all the instructions back into the ParsedLines. The ParsedLine struct is an abstraction output used by the disassembler. This gives the interface layer important information such as the Info variable which will be used to tell the theme engine that a particular line is a breakpoint, hence needs to be drawn in purple.

The generalised output for disassembly problem has been solved by using a separate function that is used to join all disassembly outputs from the CU into a convention format. This is because each mnemonic in the text is a separate member of an array. The elements of the array are joined together by using iteration. The space between the first two mnemonics never has a comma, then all mnemonics after do. This is shown best in the code comments.

```

private static string JoinDisassembled(List<string> RawDisassembled)
{
    // Simply enforce a little bit of convention. Take the following examples,
    // INC EAX
    // MOV EAX, 0x10
    // IMUL EAX, EBX, 0x20
    // Each item in the input list will be a part of the resulting disassembly. E.g.,
    // { "INC", "EAX" }
    // { "MOV", "EAX", "0x10" }
    // { "IMUL", "EAX", "EBX", "0x20" }
    // As shown earlier, if there are less than 3 of these parts, there is no comma.
    // Otherwise, every comma after zero index 1 has one afterwards.
    if (RawDisassembled.Count < 3)
    {
        return string.Join(" ", RawDisassembled);
    }
    else
    {
        // Start the string with the two that will not have a comma.
        string Output = $"{RawDisassembled[0]} {RawDisassembled[1]}";

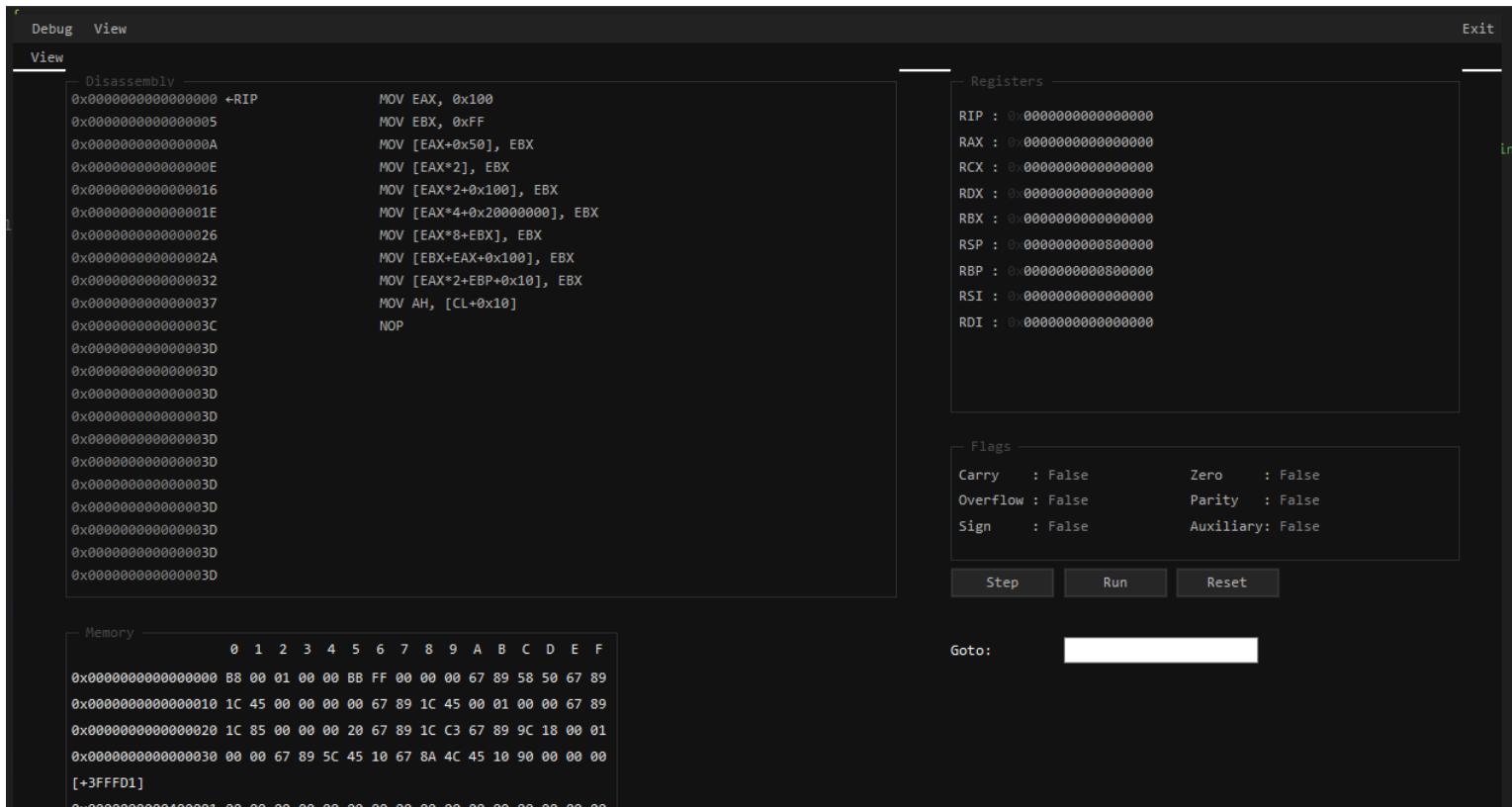
        // Append the rest with a preceeding ", ".
        for (int i = 2; i < RawDisassembled.Count; i++)
        {
            Output += ", " + RawDisassembled[i];
        }
        return Output;
    }
}

```

Disassembler justification

This is the best approach to implementing this solution because it meets the open/closed principle success criterion. This is because the task of disassembly has been separated into one individual class that uses general methods on the HypervisorBase object. This is important because it will allow future hypervisor modules to be able to use the disassembler class, such that the problem of disassembly does not have to be revisited or improved unless the purpose of the class has changed, therefore allowing the functionality of the class to be extended without having to directly modify the code. This also links to the single responsibility principle success criterion because instead of having all separate hypervisors handle disassembly in their own way, the task has been separated into a single module. This is important because it means that all disassembly will be output in the same format, which is an essential consistency in the program. It would be very confusing to an identified stakeholder such as new assembly programmer if mnemonics had different meaning across the program, or were mixed case.

User interface




```

public abstract class CustomButton : Button, IMyCustomControl
{
    public Layer DrawingLayer { get; set; }
    public Emphasis TextEmphasis { get; set; }
    public bool CustomBorder = false;
    public CustomButton(Layer drawingLayer, Emphasis textEmphasis) : base()
    {
        DrawingLayer = drawingLayer;
        TextEmphasis = textEmphasis;
    }
    protected override void OnPaint(PaintEventArgs e)
    {
        // Fill the background of the button.
        Rectangle Bounds = e.ClipRectangle;
        Drawing.FillShadedRect(e.Graphics, Bounds, DrawingLayer);

        // Create a new InnerBounds which is the border. It has to be reduced slightly to prevent clipping. The border is effectively one order
        // of layer higher than its background.
        Rectangle InnerBounds = new Rectangle(Bounds.X + 1, Bounds.Y + 1, Bounds.Width - 3, Bounds.Height - 3);
        e.Graphics.DrawRectangle(new Pen(ElevationBrushes[(int)DrawingLayer]), InnerBounds);

        // Finally draw the text in the centre of the button.
        e.Graphics.DrawString(Text, BaseUI.BaseFont, TextBrushes[(int)TextEmphasis], Drawing.GetCenter(Bounds, Text, BaseUI.BaseFont));
    }
}

public struct UISettings
{
    public readonly Font BaseFont;
    public readonly Color BackgroundColour;
    public readonly Color SurfaceColour;
    public readonly Color PrimaryColour;
    public readonly Color SecondaryColour;
    public UISettings(Font InputFont, Color InputBackgroundColour, Color InputTextColour)
    {
        BaseFont = InputFont;
        BackgroundColour = InputBackgroundColour;
        SurfaceColour = InputTextColour;

        // The suggested material dark colours have this offset, so applying it to any colour has a similar effect.
        PrimaryColour = Color.FromArgb(200, (byte)(InputBackgroundColour.R + 0xA9), (byte)(InputBackgroundColour.B + 0x74), (byte)(InputBackgroundColour.G + 0xEA));
        SecondaryColour = Color.FromArgb(240, (byte)(InputBackgroundColour.R + 0xF1), (byte)(InputBackgroundColour.B + 0xC8), (byte)(InputBackgroundColour.G + 0xB3));
    }
}

public class FormSettings
{
    public static List<SolidBrush> ElevationBrushes = new List<SolidBrush>();
    public static List<SolidBrush> TextBrushes = new List<SolidBrush>();
    public static SolidBrush LayerBrush;
    public static SolidBrush PrimaryBrush;
    public static SolidBrush SecondaryBrush;
    public static List<Color> SurfaceShades = new List<Color>();
    public static UISettings BaseUI;
    static FormSettings()
    {
        BaseUI = new UISettings(new Font("Consolas", 9), Color.FromArgb(0x12, 0x12, 0x12), Color.FromArgb(220, 255, 255, 255));

        // Create new brushes out of the BaseUI colours.
        LayerBrush = new SolidBrush(BaseUI.BackgroundColour);
        SecondaryBrush = new SolidBrush(BaseUI.SecondaryColour);
        ElevationBrushes = new List<SolidBrush>()
        {
            new SolidBrush(Color.FromArgb(0, Color.White)), // 100% transparency of layer beneath
            new SolidBrush(Color.FromArgb(17, Color.White)), // ~93%
            new SolidBrush(Color.FromArgb(22, Color.White)), // ~91%
            new SolidBrush(Color.FromArgb(30, Color.White)), // ~88%
            new SolidBrush(Color.FromArgb(38, Color.White)), // ~85%
        };
        SurfaceShades = new List<Color>()
        {
            BaseUI.SurfaceColour,
            Color.FromArgb(220, BaseUI.SurfaceColour), // ~87% transparency of text.
            Color.FromArgb(153, BaseUI.SurfaceColour), // 60%
            Color.FromArgb(97, BaseUI.SurfaceColour), // ~38%
            Color.FromArgb(17, BaseUI.SurfaceColour) // 20%
        };

        // Create TextBrushes from the surface colours.
        TextBrushes = SurfaceShades.Select(x => new SolidBrush(x)).ToList();

        PrimaryBrush = new SolidBrush(BaseUI.PrimaryColour);
    }
}

```

```

public enum Emphasis
{
    Imminent = 0,
    High = 1,
    Medium = 2,
    Disabled = 3,
    Ignored = 4
}
public enum Layer
{
    Background = 0,
    Foreground = 1,
    Surface = 2,
    Imminent = 3
}

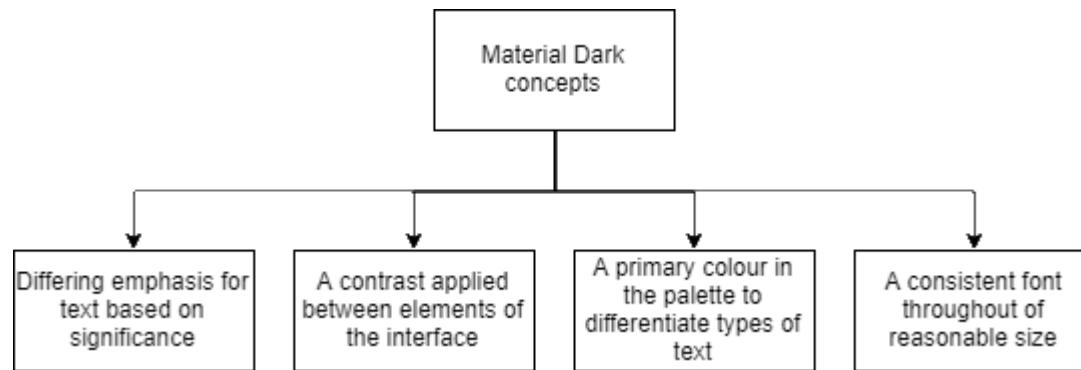
```

```

public interface IMyCustomControl
{
    Layer DrawingLayer { get; set; }
    Emphasis TextEmphasis { get; set; }
}

```

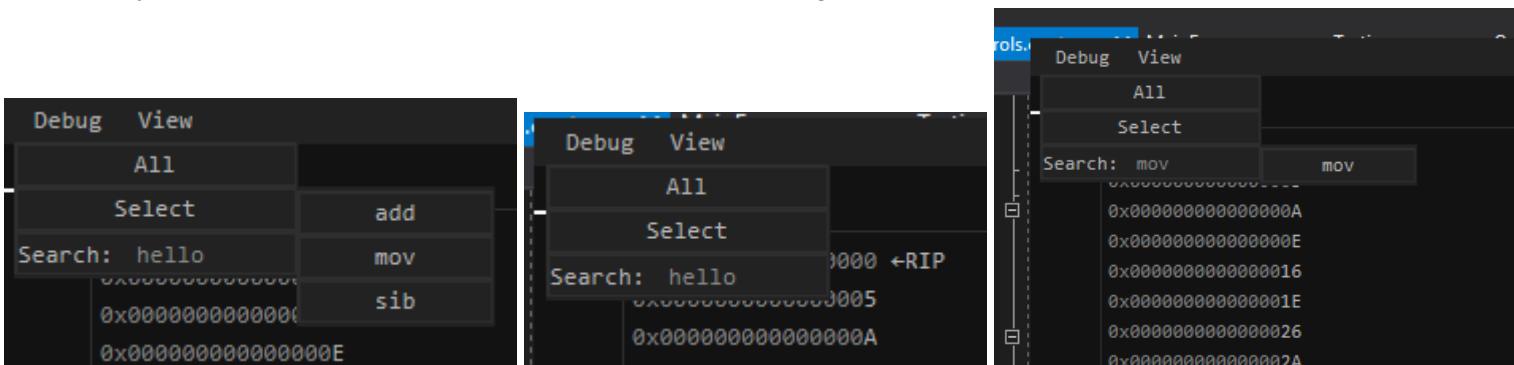
Material dark concepts explanation



Material dark concepts have been implemented by using custom classes to draw the correct colours. Each base class draws by overriding the `OnPaint()` method. This has allowed the dark theme to be drawn, using the colours provided by the `FormSettings` class. To implement a contrast between elements of the interface, the `emphasis` enum has been used as planned in the design section, and also the `layer` enum. The `layer` enum is similar except applies to the background colours of a control. These enums are fields required by the `IMyCustomControl` interface, which all custom classes implement. A primary colour is generated by adding an offset of `0xF1C8B3` onto the background colour. A secondary colour is also generated by adding `0xA974EA` onto the background colour for any additional colours required by a module.

Material dark concepts justification

This implementation of the solution to the problem identified in the design section is the best approach as it allows a fully modular implementation of controls. This is because any class that implements the `IMyCustomControl` interface must also have an `emphasis` and `layer`, such that the control will always be able to have a theme according to the material dark theme. This supports the fact that this is the best approach because it ensures that the Material dark theme design specification success criterion is always met. This is because any new modules containing controls that are added to the project will have the `DrawingLayer` and `TextEmphasis` properties such that they can be drawn by the theme engine and drawing library, hence will fit in with the theme. The best approach to implementing the primary and secondary colours is to use an offset. This offset is the same as the offset from dark to pink on the Material.io dark website, such that the offset will always provide a good contrast no matter which colour is used as the background colour. This will be useful for reusing the code in other projects as the theme engine is not strictly limited to material dark, it only depends on the brush definitions in the `FormSettings` class.



Search textBox explanation

SearchTextBox allows a string array to be searched and displayed appropriately. It is currently used to search testcases. As a constructor parameter it takes a delegate to return a string array, which will be called to fetch the array of string to search for the input and ensure the string array is always up to date. An event is invoked when a result is clicked that can be listened to by the caller. The searching method is to search by substring.

```
public SearchTextBox(GetToSearchDelegate getToSearch, Layer layer, Emphasis emphasis) : base(layer, emphasis)
{
    GetToSearch = getToSearch;

    // See CustomToolStripTextBox
    Prefix = "Search: ";

    // Add this event to the inherited keypressed event(See CustomToolStripTextBox).
    DropDown.PreviewKeyDown += KeyPressed;
}
```

The prefix variable is the part of the drawn text that cannot be deleted by user input using backspaces. The keydown event is set to call the KeyPressed method. This method will append the input character on the string.

```
public void KeyPressed(object s, PreviewKeyDownEventArgs e)
{
    // If the key pressed was a backspace and there is text to delete, do so.
    if (e.KeyCode == Keys.Back && Input.Length > 0)
    {
        Input = Input.Substring(0, Input.Length - 1);
    }
    else
    {
        // Convert the keycode into a string.
        string Key = Converter.ConvertToString(e.KeyCode).ToLower();

        // Check if the buffer size permits more characters. It's not strictly the size of the buffer but acts like so.
        // Also check the length of $Key. This is because strangely special keys such as End will be converted to "End".
        // Finally validate it as an appropriate character.
        if (Input.Length < BufferSize && Key.Length == 1 && "abcdefghijklmnopqrstuvwxyz!@#$%^&*()[];':.,/>?:@~{}1234567890".Contains(Key))
        {
            Input += Key;
        }
    }

    // Commit the changes.
    Ready();
}
```

An if statement is used to check if the key entered was a backspace and text in the input buffer than can be deleted(which has to be checked or the Prefix variable text would be deleted). When a backspace is pressed, the input is set to the substring of itself, minus one. This means that the final character is cut off the end. It is necessary to implement validation here because certain keys are considered erroneous inputs by the ConvertToString() method, which is a built-in .NET method of converting a KeyCode object to a string. For example, when the pause key(or similar) is pressed, the method will return the string "Pause". This causes strange behaviour when the user presses a key and a word is inserted. For this reason, the best approach to validating the input is to compare the returned value of ConvertToString() and check if it is a contained in a predefined string of acceptable characters, therefore, all the erroneous inputs to immediately return from the function instead of inserting the text.

```
protected override void OnPaint(PaintEventArgs e)
{
    // Draw a border
    Drawing.DrawShadedRect(e.Graphics, e.ClipRectangle, Layer.Imminent, 3);

    // Create a new bounds for the text.
    Rectangle Bounds = Drawing.GetTextCenterHeight(e.ClipRectangle);

    // Each line is drawn separately such that they can have a different emphasis. This has the following effect, http://prntscr.com/pjpw9a
    // It makes it much easier for the user to differentiate between their input and the prefix.
    e.Graphics.DrawString(Prefix, BaseUI.BaseFont, TextBrushes[(int)TextEmphasis], Bounds);

    // Offset the bounds by the width of the previous text.
    Bounds.Offset(Drawing.CorrectedMeasureText(Prefix, BaseUI.BaseFont).Width, 0);
    e.Graphics.DrawString(Input, BaseUI.BaseFont, TextBrushes[(int)TextEmphasis + 1], Bounds);
}
```

The OnPaint() method is used to draw the control. This allows for an extra feature of tinting the input text different colour, such that the user can clearly see which text they have input and which text is the prefix. This is implemented by using the next text emphasis down(less emphasis) to draw the user text.

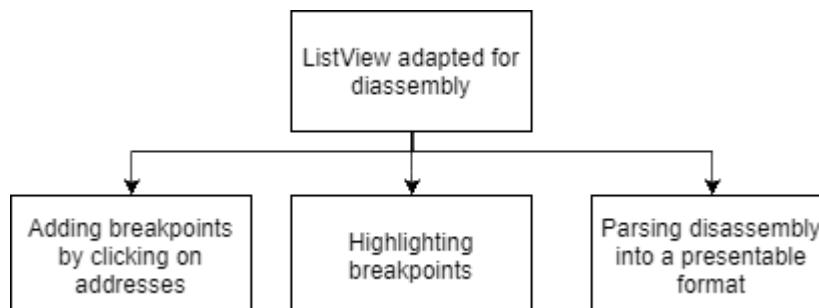
Search textbox justification

It is necessary to implement this feature as part of the “Core features will be easily accessible through UI” success criterion. This is because the users will quickly be able to search and find their testcases as the list of available testcase will be much greater by the time development has finished. This will allow them to use the TestHandler class to efficiently test their own code as they will not have to look through all the other testcases to find the testcase that they are looking for. This was the best approach to implementing this feature because it allows them to use the keyboard instead to find the testcase. This is a lot faster than clicking, therefore it is the most effective at simplifying the process.

Review

Reviewer	Comment
Self review	This user interface has been implemented successfully as many of the success criteria have already been met early in development. This has been possible because of the interface and base classes provided in the source code. It will easily be possible to finish off the user interface entirely in the upcoming versions.
Stakeholder 1 - An existing Python programmer	“The GUI is nice, it’s very easy to find the information you’re looking for as the interface is clearly divided into separate sections for each component: the disassembly, the registers, the memory, the flags. I liked the search feature for the testcases, it would be very convenient if I was testing my own plugin module that added and opcode, I would be able to use the other programming interfaces shown throughout the program such as the TestHandler to test my own code”
Stakeholder 2 - An existing assembly programmer	“This interface style is very nice. I’m glad you have implemented a dark theme and stuck to a common convention, it feels very professional but not too serious. The information is neatly organised. It is very sufficient in volume; a new assembly programmer has all the information they need to debug their code, so I would consider this interface successful as it clearly meets the needs of your target audience.”

Disassembly list view



Adding breakpoints explanation

The adding breakpoints subproblem has been solved by using events to fire callbacks as planned.

```

// Delegate and event for when an address is clicked. In the current MainForm, this sets a breakpoint.
public delegate void OnAddressClickedDelegate(ulong address);
public event OnAddressClickedDelegate OnAddressClicked = (a) => { };
  
```

```

SelectedIndexChanged += (s, a) =>
{
    // This event will be raised twice, once when it should be called, and the second when the .Clear() is called below.
    // I would strongly recommend using OnAddressClicked rather than SelectedIndexChanged because this method makes sure that
    // there is never more than one index selected, or another mysterious act from windows forms.
    if (SelectedItems.Count > 0)
    {
        // The numeric address the line represents is stored in SubItems[1] and never drawn. This saves having to do extra parsing
        // trying to select the hexadecimal from the beginning of the line. A little extra memory used but its worthwhile from a
        // maintenance and robustness perspective.
        OnAddressClicked.Invoke(ulong.Parse(SelectedItems[0].SubItems[1].Text));
    }
    SelectedItems.Clear();
};

```

The .NET SelectedIndexChanged event has been adapted to solve this problem. This will handle the detection of the event without having to code that part of the solution. However, there are some undesired side effects of the selected index changed event. As part of .NET windows forms, multiple items in the listbox can be selected. This meant that the user would have to click once to select the item, but double click to deselect, which would sometimes not work at all. To solve this problem, the item is automatically deselected once the event has been handled. This is performed by SelectedItems.Clear(). This means that the user will no longer have to click twice because OnAddressClicked event was already invoked, such that the callback has been fired as a result of this, hence it will only be a single click to reselect the item.

Adding breakpoints justification

This is the best approach to implementing the planned solution because it allows callbacks to be given relevant details(the address), whilst abstracting irrelevant details such as the disassembled lines from the problem. This will allow the callback functions to handle the outputs more effectively because there is less unnecessary parts of the problem that they have to handle. The invoke call is also passed an address stored in a ulong. This means that the callback will then be able to handle the necessary actions without having to call functions in the processing layer to obtain the address. This is important because it meets the pipeline structure design success criterion. This is because the flow of information is kept purely in one direction. In the current implementation, execution flows from the interface layer(the DisassemblyListView), then into the intermediary layer to set the breakpoint. This is important because it will improve the maintainability of code, particularly tracking bugs in the communication between the interface and intermediary layers, which will be easier as it is clear where erroneous data has originated from.

Parsing disassembly explanation

```
private void FormatAndApply(ParsedLine line)
{
    // Parse the address as hex.
    string FormattedAddress = line.Address.ToString("X").PadLeft(16, '0');

    // Make breakpoint lines purple using markdown. No other formatting is applied to these.
    if ((line.Info | Emulator.AddressInfo.BREAKPOINT) == line.Info)
    {
        FormattedAddress = $"^0x{FormattedAddress}^";
    }
    else
    {
        // Use Drawing.InsertAt..() to remove emphasis from insignificant zeroes. This is done by starting
        // with a low emphasis, "$" then inserting the normal emphasis, "f" at the significant digits.
        FormattedAddress = $"{0x${Drawing.InsertAtNonZero(FormattedAddress, "f")}}\";

    }

    // Add some spacing between the address and disassembled line.
    FormattedAddress = $"{FormattedAddress} {line.DisassembledLine}";

    // Replace some of that spacing if the line is the next to be executed.
    if ((line.Info | Emulator.AddressInfo.RIP) == line.Info)
    {
        FormattedAddress = FormattedAddress.Remove(23, 4).Insert(23, "+RIP");
    }

    // Very similar idea to as in MemoryListView. Look there for more explanation of this.
    if (Items.Count - 1 < line.Index)
    {
        Items.Add(new ListViewItem(new string[] { FormattedAddress, line.Address.ToString() }));
    }
    else
    {
        Items[line.Index].Text = FormattedAddress;

        // Add the address as a sub item that the user will never see, but will be useful in the OnAddressClicked event.
        Items[line.Index].SubItems[1].Text = line.Address.ToString();
    }
}
```

The solution to parsing disassembly and highlighting breakpoints has been implemented in a single procedure. This has been performed by a series of condition statements that are applied to each address received from the Disassembler class. The address is first converted to a string such that the markdown formatting can be applied to it and padded left with up to 16 “0”s such that address values look consistent(all the same length) without affecting their value. A breakpoint is then tested for by checking the Info variable of the input. This was set by the disassembler class in order to allow separate modules to parse the line however they require. Breakpoints are coloured by inserting carats around the line. This produces the following effect,

0x0000000000000000A	MOV EBP, 0x800000
0x0000000000000000F	MOV DWORD PTR [EAX*8+RBX], EBX
0x000000000000000013	MOV BYTE PTR [EBX+RAX+0x100], BL
0x00000000000000001B	MOV DWORD PTR [EAX*2+EBP+0x10], EBX

The next instruction to be executed is marked by writing RIP next to the address. This is also performed by using the abstraction provided by the Disassembler class in order to determine the attribute. This is done by replacing some of the spaces that were previously inserted between the address and instruction with “RIP” to create the following effect,

0x0000000000000000F	MOV DWORD PTR [EAX*8+RBX], EBX
0x000000000000000013	MOV BYTE PTR [EBX+RAX+0x100], BL
0x00000000000000001B+RIP	MOV DWORD PTR [EAX*2+EBP+0x10], EBX

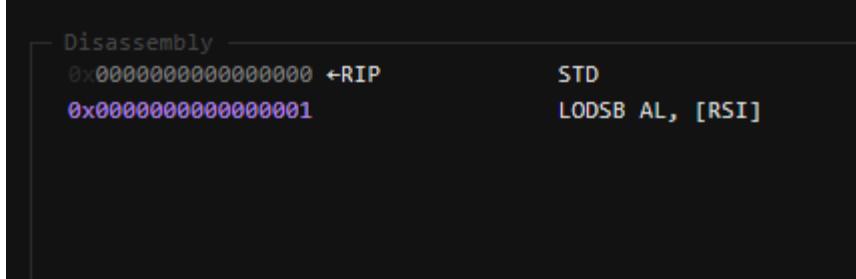
The parsed line is then inserted into the items collection, which is managed by the .NET windows forms framework from thereon. The theme engine does not draw the second subitem of the item(line.Address.ToString()), which allows other modules to then easily identify the address of the line(as shown in the previous section) without having to perform many conversion procedures(it only has to be converted into an integer; the markdown does not have to be removed).

Justification of parsing disassembly

This is the best approach to implementing the solution to this problem because it allows many details of the problem to be abstracted from other modules. For example, details of how the markdown is applied to breakpoints is abstracted by providing the address of the instruction in the second subitem. This allows components such as the adding breakpoints procedure to implement more efficient solutions to finding the details relevant to them about the particular line, only having to apply one parsing procedure. The other details that may be necessary to a different module, such as the theme engine, are also retained. This allows the markdown to still be used as the theme engine will use the first subitem of the listview item to access the full disassembled line containing markdown, hence has access to the relevant information it needs, which in this case is not the address but the disassembled line. This links to the single responsibility success criterion. This is because the procedures to handling this abstraction has been entirely delegated to this class. This means that other modules will not have to implement individual methods for obtaining the information they need, such as complex parsing routines that would select the first part of the address and attempt to manually convert it back into an integer type. This is important because less development time will have to be spent on this particular problem and can be spent on developing the essential features that allows the stakeholders to make use of this feature.

Testing

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	To create and remove breakpoints	An address will be clicked multiple times.	mov eax,0x100 mov ebx,0xff mov ebp,0x800000	Normal	Address changes colour	It is necessary to use this test data because there needs to be code such that there are disassembled lines to click on. This will allow the feature to be tested because it will be visibly seen whether the colour has changed or not; no coded debugging features means are necessary.



(Figure shows first line as gray and second line as purple)

The line changed colour, so this test passed

Review

Reviewer	Comment
Self review	The disassembly listview included in this version has been successful, it has met its necessary UI success criteria and has followed the plan from the design section, including handling all subproblems identified in the stepwise refinement.
Stakeholder 1 - Inexperienced assembly programmer	“This interface is nice in many areas, it is both functional and aesthetic. I would be more than happy to use this as a debugging interface, it has all the information I would need to develop an algorithm and has flawless disassembly output; the same results would be shown from any popular disassembler”
Stakeholder 2 - C# Programmer	“This interface seems very intuitive, it was very obvious that a breakpoint would be set by clicking on the line. I think the next best step in development will be to improve the source code maintainability through means of interfaces as the user interface has progressed far

FlagSet

```

public enum FlagState
{
    // Flags can have 3 states: on, off or never specified at all.
    // If it is undefined, the flag will be ignored when Overlap() is called
    // In previous versions, flags were stored as boolean values, which definitely
    // allowed for simpler code within the class. However having the option to leave
    // a flag as UNDEFINED greatly simplifies the usage of the FlagSet outside of the class

    UNDEFINED = 0,
    OFF = 1,
    ON = 2,
}
public struct FlagSet
{
    public FlagState Carry;
    public FlagState Auxiliary;
    public FlagState Overflow;
    public FlagState Zero;
    public FlagState Sign;
    public FlagState Parity;
    public FlagState Direction;
    public FlagState Interrupt;
    public FlagSet(FlagState initialiseAs = FlagState.UNDEFINED)
    {
        // Construct a flag set with all flags equal to $initialiseAs
        Carry = initialiseAs;
        Auxiliary = initialiseAs;
        Overflow = initialiseAs;
        Zero = initialiseAs;
        Sign = initialiseAs;
        Parity = initialiseAs;
        Direction = initialiseAs;
        Interrupt = initialiseAs;
    }
    public void Set(FlagState setTo)
    {
        // Change all the flags in the struct to $setTo
        this = new FlagSet(setTo);
    }
}

```

```

public static bool GetParity(byte input)
{
    // Function purpose: Determine odd or even parity of a given byte

    // Start assuming even parity
    bool Even = true;
    // Iterate through every bit in the byte. This is done by using the counter to create a
    // bitmask
    for (int i = 0; i < 8; i++)
    {
        // Parity can be checked by iterating through each bit in the byte and
        // flipping the value of even every time
        // The bitmask is created by shifting one by the counter
        // (which starts at zero therefore no further action is required), then used to AND input.
        if ((input & (1 << i)) > 0)
        {
            // If Even is true, then XORed by true, it will become false. If Even is false,
            // then XORed by true, it will become true
            Even ^= true;
        }
    }
    return Even;
}

```

```

public FlagSet(byte[] input)
{
    // A constructor that can set ZF, SF, PF to what they are defined as in most cases
    Carry = FlagState.UNDEFINED;
    Auxiliary = FlagState.UNDEFINED;
    Overflow = FlagState.UNDEFINED;
    Direction = FlagState.UNDEFINED;
    Interrupt = FlagState.UNDEFINED;

    // ZF is set if $input is equal to zero.
    Zero = input.IsZero() ? FlagState.ON : FlagState.OFF;

    // SF is set if $input has a negative sign in twos compliment form.
    Sign = input.IsNegative() ? FlagState.ON : FlagState.OFF;

    // PF is set if the number of bits on in the first byte of $input is even.
    // e,g
    // input[0] == 0b0000 ; PF
    // input[0] == 0b1000 ; NO PF
    // input[0] == 0b1010 ; PF
    Parity = Bitwise.GetParity(input[0]) ? FlagState.ON : FlagState.OFF;
}
public FlagSet Overlap(FlagSet input) => new FlagSet()
{
    // Return a new flag set based on $this and $input.
    // If a flag in $input is FlagState.UNDEFINED, the value of that flag in $this
    // is used instead(which could also be FlagState.UNDEFINED)
    // Otherwise, that flag in the returned FlagSet is equal to the same flag in $input.
    // For example,
    // $Input.Carry == FlagState.UNDEFINED
    // $this.Carry == FlagState.ON
    // New Carry = FlagState.ON
    // Another example,
    // $Input.Overflow == FlagState.ON
    // $this.Overflow == FlagState.OFF
    // New Carry = FlagState.ON
    Carry = input.Carry == FlagState.UNDEFINED ? Carry : input.Carry,
    Auxiliary = input.Auxiliary == FlagState.UNDEFINED ? Auxiliary : input.Auxiliary,
    Overflow = input.Overflow == FlagState.UNDEFINED ? Overflow : input.Overflow,
    Zero = input.Zero == FlagState.UNDEFINED ? Zero : input.Zero,
    Sign = input.Sign == FlagState.UNDEFINED ? Sign : input.Sign,
    Parity = input.Parity == FlagState.UNDEFINED ? Parity : input.Parity,
    Direction = input.Direction == FlagState.UNDEFINED ? Direction : input.Direction,
    Interrupt = input.Interrupt == FlagState.UNDEFINED ? Interrupt : input.Interrupt
};

```

```

public bool EqualsOrUndefined(FlagSet toCompare)
{
    // Return whether two flag sets have the same flags set ON. OFF and UNDEFINED are treat as the same.
    return toCompare.And(this).ToString() == ToString();
}
public FlagSet And(FlagSet toCompare) => new FlagSet()
{
    // Perform a bitwise AND on every flag(in the sense that ON == 1, OFF | UNDEFINED == 0).
    // If both flags of a kind are ON in $toCompare and $this, that flag will be ON in the result. Otherwise, it will be OFF(no UNDEFINED).
    Carry = (Carry & toCompare.Carry) == FlagState.ON ? FlagState.ON : FlagState.OFF,
    Overflow = (Overflow & toCompare.Overflow) == FlagState.ON ? FlagState.ON : FlagState.OFF,
    Sign = (Sign & toCompare.Sign) == FlagState.ON ? FlagState.ON : FlagState.OFF,
    Zero = (Zero & toCompare.Zero) == FlagState.ON ? FlagState.ON : FlagState.OFF,
    Auxiliary = (Auxiliary & toCompare.Auxiliary) == FlagState.ON ? FlagState.ON : FlagState.OFF,
    Parity = (Parity & toCompare.Parity) == FlagState.ON ? FlagState.ON : FlagState.OFF,
    Direction = (Direction & toCompare.Direction) == FlagState.ON ? FlagState.ON : FlagState.OFF,
    Interrupt = (Interrupt & toCompare.Interrupt) == FlagState.ON ? FlagState.ON : FlagState.OFF
};

```

```
public FlagState this[string name]
{
    // An index attribute allows a flag to be fetched by entering its name in the accessor.
    // e.g Flag["carry"] would be valid, Flag["invalid"] would not.
    get
    {
        return name.ToLower() switch
        {
            "carry" => Carry,
            "sign" => Sign,
            "overflow" => Overflow,
            "parity" => Parity,
            "zero" => Zero,
            "auxiliary" => Auxiliary,
            "direction" => Direction,
            "interrupt" => Interrupt,
            _ => throw new LoggedException(LogCode.FLAGSET_INVALIDINPUT, name)

        };
    }

    // Also provides the same for setting a flag.
    set
    {
        // Also provides the same for setting a flag.
        switch (name.ToLower())
        {
            case "carry":
                Carry = value;
                return;
            case "sign":
                Sign = value;
                return;
            case "overflow":
                Overflow = value;
                return;
            case "parity":
                Parity = value;
                return;
            case "zero":
                Zero = value;
                return;
            case "auxiliary":
                Auxiliary = value;
                return;
            case "direction":
                Direction = value;
                return;
            case "interrupt":
                Interrupt = value;
                return;
            default:
                throw new LoggedException(LogCode.FLAGSET_INVALIDINPUT, name);
        }
    }
}
```

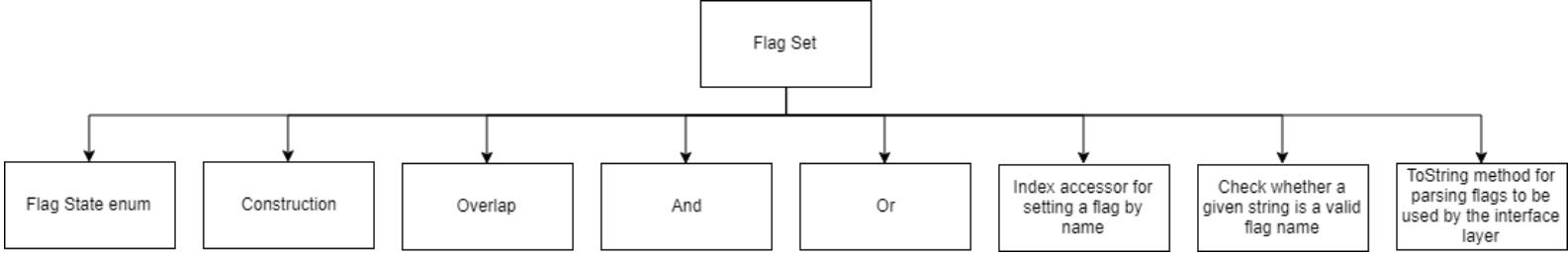
```

public static bool ValidateString(string input)
{
    // A method to check whether $input is a valid name for a flag
    // that can be used in the index accessor.
    input = input.ToLower();
    return
        input == "zero"
    || input == "carry"
    || input == "overflow"
    || input == "sign"
    || input == "parity"
    || input == "auxiliary"
    || input == "direction"
    || input == "interrupt";
}

public override string ToString()
{
    // Returns a string of flag names in the order CF,OF,SF,ZF,AF,PF,DF,IF.
    // If a flag is not set ON, it is not appended to the string.
    string Output = "";

    //If carry == FlagState.ON, append CF
    Output += Carry == FlagState.ON ? "CF" : "";
    Output += Overflow == FlagState.ON ? "OF" : "";
    Output += Sign == FlagState.ON ? "SF" : "";
    Output += Zero == FlagState.ON ? "ZF" : "";
    Output += Auxiliary == FlagState.ON ? "AF" : "";
    Output += Parity == FlagState.ON ? "PF" : "";
    Output += Direction == FlagState.ON ? "DF" : "";
    Output += Interrupt == FlagState.ON ? "IF" : "";
    return Output;
}

```



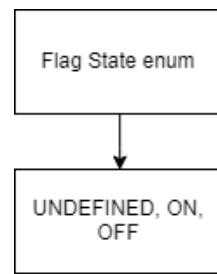
Structure explanation

As planned in the stepwise refinement, the functions that the data structure was planned to provide have been tackled separately in different methods but members of the same structure.

Structure justification

This is the best approach to implementing the solution to this problem because it allows the internal variables of the structure; the flags, to be used implicitly by functions without any interaction from the caller. This abstracts the need for the caller to understand the implementations of each function because the methods can access the class-scope variables directly, such that the caller need only understand the relevant inputs and outputs of the function. Because of this, the dependency inversion principle success criterion is met because the modules that interact with the FlagSet will only have to depend on the abstractions, the variables stored in the structure, in order to operate and use the FlagSet such that if the details on how the information to create the structure is obtained, i.e. the reason the flags were set were to change, the modules that use the class as an input will be unaffected, therefore less development time will be spent fixing bugs in other modules after improving the FlagSet class.

FlagState explanation

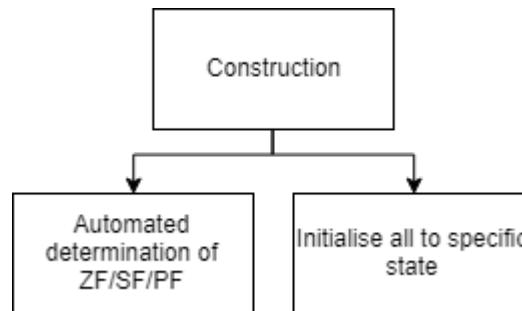


The FlagState concept has been implemented using an enum structure. The enum has three members: ON, OFF, and UNDEFINED. Each of these are used in the structure and by external callers for specific operations. Throughout the current code, UNDEFINED is assigned to a flag that is to be ignored in the operation.

FlagState justification

The best approach to implementing the solution to his problem was to use the enum data structure. This abstraction allowed for other procedures to perform operations on the variables which are much clearer than using integers or booleans. This links to the abstraction of low level concepts success criterion. This is because the flags at a low level are all stored in a single integer, where each bit is the state of a flag. By using an enum, this complex detail can be abstracted, which makes the code more maintainable as instead of the maintainer having to understand complex x86-64 principles, they can use C# language operators such as == and != when developing the structure.

Construction explanation



The construction solution has been implemented according to the plan in the design section. The variables that hold the flags can either all be initialised to a specified FlagState parameter, or can be automatically determined using several procedures from the bitwise class: Bitwise.IsNegative(), Bitwise.IsZero(), and Bitwise.GetParity(). GetParity will return true if and only if there are an even number of bits in the byte and false otherwise. This has been implemented by using iteration to loop through each bit in the byte and flipping the value of the Even boolean for every one bit.

Justification & Relation to break down of the problem

The flag set was identified as a subproblem in the stepwise refinement when planning the project. This has been solved using the same methods as planned. A method was created in the Bitwise library to apply a computational solution using the planned algorithm for GetParity. This will allow the method to be reused in the project where needed and in other projects that may require to check parity as it is not dependent on any other modules. This has been reused in the FlagSet structure constructor, where it was identified that the FlagSet would require a computational method to determine the parity in order to set the PF when necessary. This has allowed the constructor to be used in multiple places across the code such that the same procedure is applied to all callers. This contributes towards the 1:1 emulation success criterion because the exact same process will be applied to every input. This includes the abstraction of any details about the input such as signedness, which is never required in the x86-64 specification; all operations on signed/unsigned inputs are performed the same.

Validation explanation

Validation has been used to validate an input string against all valid flag names that are accepted by the index accessor. This is done by converting the string to lower(as upper and lower case are accepted in the index accessor), then comparing against all valid flag names, then true or false is returned depending on whether a match was found or not.

Validation justification

This validation was identified as a subproblem in the design section. This validation procedure is necessary for when inputs to the program will be parsed and used to reference flags. For example, in an upcoming version, the TestHandler will support testcases that also test for flags. This will be necessary to allow the flags read from the file to be validated such that the validation procedure is always up to date and modular because the TestHandler will not have to store the definitions itself, rather use the validation procedure provided. This links to the open/closed principle success criterion. This is because it will be possible to extend the functionality of the TestHandler class by implementing more valid flag names into the validation procedure, such that it will then be able to handle more flags as the procedures to handle each flag are not stored in the TestHandler. For this reason, development time can be saved as when improving the FlagSet struct, there will be no concern of the effects on other modules such as the TestHandler.

Tests

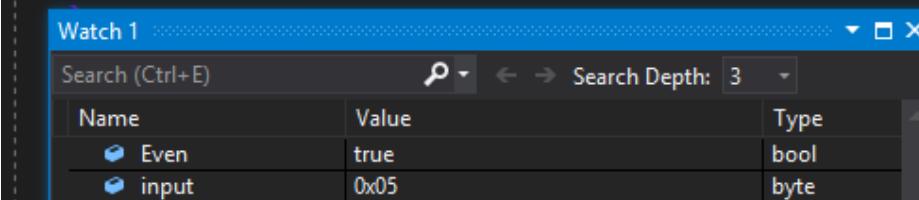
Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	Test for even parity	The input to the function will be a known even parity. A breakpoint after the function call will allow me to check whether the output is true.	Input = 5 5d = 0101b, 2 ones is even parity.	Normal	Output = true	This test is necessary to make sure that the function will return a value and that there are no runtime errors caused by incorrect code. Other tests will not be reliable if basic functionality is not confirmed.

```

public static bool GetParity(byte input)
{
    // Function purpose: Determine odd or even parity of a given byte

    // Start assuming even parity
    bool Even = true;
    // Iterate through every bit in the byte. This is done by using the counter to create a
    // bitmask
    for (int i = 0; i < 8; i++)
    {
        // Parity can be checked by iterating through each bit in the byte and
        // flipping the value of even every time
        // The bitmask is created by shifting one by the counter
        // (which starts at zero therefore no further action is required), then used to AND input.
        if ((input & (1 << i)) > 0)
        {
            // If Even is true, then XORed by true, it will become false. If Even is false,
            // then XORed by true, it will become true
            Even ^= true;
        }
    }
    return Even;
}

```



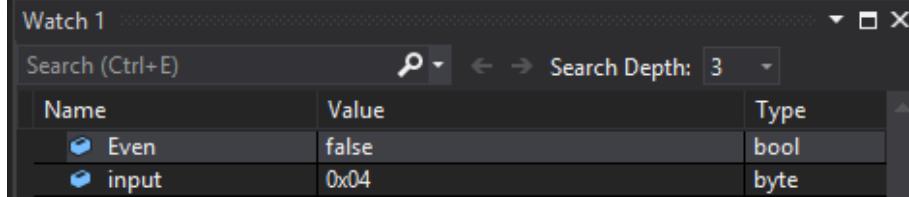
Explanation

The even boolean was true when 5 was input, which is the same as the expected result. This test has passed and no remedial action is required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
2	Test for odd parity	The input to the function will be a known odd parity. A breakpoint after the function call will allow me to check whether the output is false.	Input = 4 4d = 0100b, 1 one is odd parity	Normal	Output = false	This test is necessary to make sure that the even value is being affected. If only Test #1 was performed, it is not certain that the function works because there is the possibility of a logical error in the if conditional statement that would cause the even boolean to never change.

```
public static bool GetParity(byte input)
{
    // Function purpose: Determine odd or even parity of a given byte

    // Start assuming even parity
    bool Even = true;
    // Iterate through every bit in the byte. This is done by using the counter to create a
    // bitmask
    for (int i = 0; i < 8; i++)
    {
        // Parity can be checked by iterating through each bit in the byte and
        // flipping the value of even every time
        // The bitmask is created by shifting one by the counter
        // (which starts at zero therefore no further action is required), then used to AND input.
        if ((input & (1 << i)) > 0)
        {
            // If Even is true, then XORed by true, it will become false. If Even is false,
            // then XORed by true, it will become true
            Even ^= true;
        }
    }
    return Even;
}
```



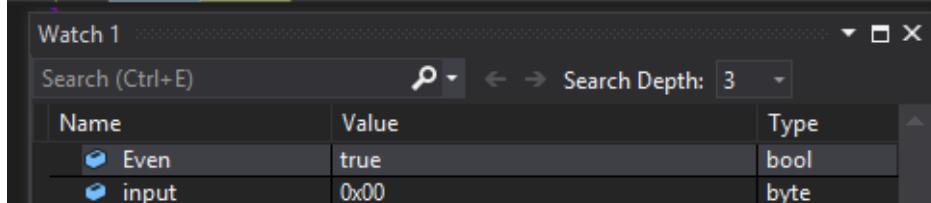
Explanation

The even boolean was false when 4 was input, which is the same as the expected result. This test has passed and no remedial action is required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
3	Test for even parity when input zero	The function will be called with a zero. A breakpoint can be used to check after that the output is true.	Input = 0 0 = 0000b, 0 ones is even parity	Boundary	Output = true	This test is necessary to make sure that the function works properly on the boundary condition. This is because the code in the IF statement should never be executed when the input is zero, the boolean should stay true until the end of the function.

```
public static bool GetParity(byte input)
{
    // Function purpose: Determine odd or even parity of a given byte

    // Start assuming even parity
    bool Even = true;
    // Iterate through every bit in the byte. This is done by using the counter to create a
    // bitmask
    for (int i = 0; i < 8; i++)
    {
        // Parity can be checked by iterating through each bit in the byte and
        // flipping the value of even every time
        // The bitmask is created by shifting one by the counter
        // (which starts at zero therefore no further action is required), then used to AND input.
        if ((input & (1 << i)) > 0)
        {
            // If Even is true, then XORed by true, it will become false. If Even is false,
            // then XORed by true, it will become true
            Even ^= true;
        }
    }
    return Even;
}
```



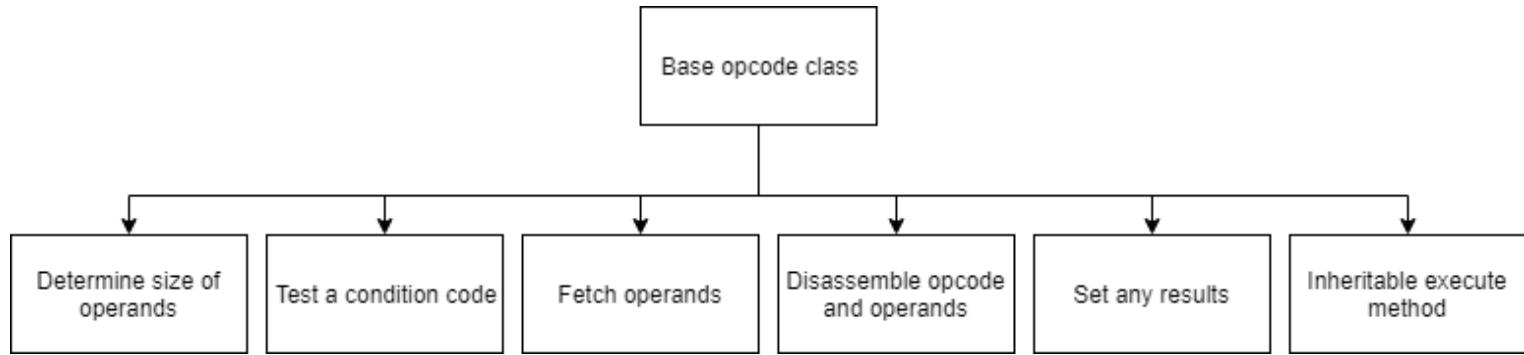
Explanation

The even boolean was true when 0 was input, which is the same as the expected result. This test has passed and no remedial action is required.

Release version

Release notes

- Many opcodes added as a response to stakeholder feedback: CBW, CLD, CLC, CMP, CMPS, LODS, MOVS, ROR, RCR, RCL, ROL, SCAS, SETcc, TEST
- Opcode base class and string operation base class
- Bitwise library completed
- Testcases added for new opcodes
- **Base opcode class**



```
// Inheritable execute method that is defined by all derived classes(hence abstract keyword)
public abstract void Execute();
```

Structure explanation

The opcode class has been implemented as planned. There is an interface used, IMyOpcode, which has Disassembly and Execute required methods which are inheritable by the derived class. Subproblems have been split into separate functions, e.g. TestCondition() tests a condition code.

Structure justification

The best approach to implementing this class is to use the design patterns demonstrated in the design section. This is because they were already shown to meet the standards of the class required by the design success criteria. For example, by having separate functions for the set task and fetch task, it will be meet the open/closed principle success criterion because it will be possible to extend the functionality of the base class by creating a derived class the introduced more features, which will only be possible as it has specific control over the order of certain procedures such as fetching, that may need to be changed in the future, however the process of how data is actually fetched still remains abstracted.

Fetching and setting explanation

```
private readonly IMyDecoded Input;
// Fetch all operands
protected List<byte[]> Fetch() => Input.Fetch();
// Discriminative set, use the operands in the order that would be expected
protected void Set(byte[] data) => Input.Set(data);
// Indiscriminative set(see documentation under "Indiscriminative operations")
protected void Set(byte[] data, int operandIndex) => ((DecodedCompound)Input).Set(data, operandIndex);
```

The operand classes have allowed the process of setting and fetching operands very simple. This is because the specific procedures of how data will be set are different for every operand, hence have been moved into their own classes. This has also made it possible to implement the discriminative and indiscriminative operations discussed in the design section as the specific decoded compound operation can be used to perform the set(as no other operand would see a difference in discriminative/indiscriminative operations)

Justification of fetching and setting

This is the best approach to implementing the solution to this problem because it allows other sections of the code that have already been coded to be reused and simplify the solution to the problem. This has allowed the Liskov substitution principle design success criterion to be met because the opcode base class will be usable by all subtypes of operands(classes that implement the IMyDecoded interface) as the IMyDecoded is used as the variable type for the Input, not a specific operand. Because of this, maintenance and improvements will be simpler because parts of the solution do not have to be recoded to fit the new features, rather the new features can be adapted to implement the interface.

Determine size of operands explanation

```
protected RegisterCapacity SetRegCap()
{
    // SetRegCap() can be used to determine the register capacity if the opcode has a
    // default working capacity of a DWORD, that can be extended to a QWORD with a rex.w
    // byte, or reduced to a WORD with a SIZEOVR legacy prefix. If the opcode is a
    // BYTE capacity variant, that can also be inferred by calling the constructor
    // with OpcodeSettings.BYTEMODE as a parameter, which will have priority over any
    // other.
    // Here are some examples,
    //     BYTES          DISASSEMBLY
    //     01 C0          ADD EAX,EAX
    //     48 01 C0       ADD RAX,RAX
    //     66 01 C0       ADD AX,AX
    //     00 C0          ADD AL,AL
    // As you can see, 01 was the byte of the opcode, ADD, C0 was the ModRM for EAX,EAX.
    // When the rex.w prefix was present(0x48), the operands became QWORD registers.
    // When the SIZEOVR prefix was present(0x66), the operands became WORD registers.
    // Finally when the opcode byte 0x00 was used, this implied that the add was for
    // two byte registers rather than two dword registers(as the first defaulted to).
    // Not every opcode exhibits this behaviour, for example
    //     BYTES          DISASSEMBLY
    //     50              PUSH RAX
    //     48 63 C0        MOVSXD RAX, EAX
    //     FF 20          JMP [RAX]
    // Therefore the constructor gives the option for the caller to override the capacity
    // with its own definitions, or SetRegCap() can be called in the function. As mentioned
    // in the summary, the input will automatically be adjusted to the new register capacity
    // when the Capacity here is changed, see the Capacity property. If there are multiple
    // conditions met that would bring ambiguity to the capacity, the priority can be inferred from
    // below, however this would be a case of invalid input, as there is no case where there should
    // be a REX.W as well as a SIZEOVR for example.

    // If byte mode is forced(This could still be overridden by setting a capacity in the constructor,
    // but there really should be no reason to)
    if ((Settings | OpcodeSettings.BYTEMODE) == Settings)
    {
        return RegisterCapacity.BYTE;
    }

    // If a Rex.W is present
    else if ((RexByte | REX.W) == RexByte)
    {
        return RegisterCapacity.QWORD;
    }

    // If a SIZEOVR is present.
    else if (LPrefixBuffer.Contains(PrefixByte.SIZEOVR))
    {
        return RegisterCapacity.WORD;
    }

    // Default to DWORD.
    return RegisterCapacity.DWORD;
}
```

This implementation works by performing a series of conditional decision statements as planned. It uses several indicators spread around the processing layer that indicate the register capacity. BYTEMODE is a setting passed as an argument to the constructor of the opcode base class which is hardcoded for certain opcodes as it is not implied through any other factors.

Determine size of operands justification

This is the best approach to implementing the solution to this problem because it will reduce the amount of code repeated throughout the opcodes. This is because the vast majority of opcodes use these set of rules to determine the capacity. It also still allows the opcodes to call their own procedure to determine the capacity by a different set of rules after the base constructor is called. This allows the open/closed principle design success criterion to be met. This is because the derived classes will still be able to extend the functionality of the base class(hence perform the operation of the opcode they are designed to implement) without being constrained by the method, however, if they require it, development time can be saved where the procedure would have to be repeated in the class as the base class method can be used.

Test condition code explanation

```
protected bool TestCondition(Condition condition)
=> condition switch
{
    // It takes some intuition to understand why these predicates work with flags like this.
    // To understand it every condition must be modelled as a subtraction, you must first
    // consider the output of A - B before considering A < B.
    // Here I will demonstrate a few,
    // If A < B, the result of A - B will be negative. This can also be seen in the converse,
    // as if B > A, the result of B - A will be positive. This is where signs can be used
    // to spot things like this. Lets focus on A < B. If A < B => A - B < 0, therefore if
    // this was in assembly, SUB A,B (pretend these are registers) would set the carry flag because
    // there was a borrow out of the MSB(Reading the Bitwise class would help understanding this).
    // This condition code would be either B(below), NAE(not above or equal), or C(carry). I generally
    // favour the latter where possible because its meaning is easier to infer, and so is used throughout
    // the program, but all three are equal.
    // There is a small detail in assembly where part of the opcode has specific bits set depending on
    // its condition. E.g if the last 4 bits of the opcode are 0100, it will have the equal/zero condition.
    // Naturally this is only for certain opcodes such as jmp, cmp, and set. Another important thing to consider
    // is the meaning of "above", "below", "less", and "greater". Above and below indicate a condition for a
    // signed number, therefore make use of the Carry and Zero flags. Less and greater indicate a condition
    // for an unsigned number, therefore make use of the Overflow, Sign, and Zero flags.
    // More about conditions can be found in Util.Disassembly.

    // If there is no carry and no zero, the subtraction must have yielded a positive non zero result,
    // therefore X - Y > 0 => X > Y.
    Condition.A => Flags.Carry == FlagState.OFF && Flags.Zero == FlagState.OFF,

    // If the above is false, (X - Y <= 0) => (X <= Y)
    Condition.NA => Flags.Carry == FlagState.ON || Flags.Zero == FlagState.ON,

    // If a carry was set, this would indicate (X - Y < 0) => (X < Y), or just that the carry flag is set.
    Condition.C => Flags.Carry == FlagState.ON,

    // Opposite of above, would imply (X - Y >= 0) => (X >= Y)
    Condition.NC => Flags.Carry == FlagState.OFF,

    // A strange, moderately undocumented condition, where a condition is taken only if RCX = 0 or ECX = 0.
    // In 32bit, an ADDROVR would denote "if CX = 0", but in x86-64 this was changed to "if ECX == 0", otherwise
    // would be "if RCX == 0". I assume this is because of ancient usage of ECX specifically as a "count register"
    // that holds the iterator during loops. This still is generally the case though, and saves a TEST ECX,ECX
    Condition.RCXZ => (LPrefixBuffer.Contains(PrefixByte.ADDROVR) && ECX.FetchOnce().IsZero()) || RCX.FetchOnce().IsZero(),

    // Zero or equal. (X - Y == 0) => (X==Y)
    Condition.Z => Flags.Zero == FlagState.ON,

    // Not zero/ not equal. (X - Y != 0) => (X != Y)
    Condition.NZ => Flags.Zero == FlagState.OFF,
```

```
// Greater than, for signed numbers.
// Once understood it will be very easy to interpret the other signed specific conditions.
// It is hard to think of mathematically, rather should be thought of logically.
// If X is a signed twos compliments negative number and Y is positive, Y - X will actually
// loop back round because of integer overflow, a borrow out of the sign bit. However, the
// result will still be positive and so will the overflow flag be set because the result
// had the same sign as its subtractee(the value being subtracted from).
// For example,
// 9 - -1 = 10 => 9 > -1 because 10 > 9
// Once understanding how flags work, it is a lot easy to apply mathematically,
// but first you consider how signed numbers work in twos compliment. See Util.Bitwise
// Now consider the case that they are not, for complexity lets say two negatives.
// In decimal,
// -1 - -10 = -9 => -1 > -10 because -9 > -10
// In this case, both the sign flag and overflow flag will be true. This works for
// both cases because the predicate is Sign == Overflow not Sign & Overflow = true
// In the following, != is used in place of XOR. They both have the same meaning, but
// saves weird casting to booleans(remember that FlagStates are enum members).
Condition.G => Flags.Zero == FlagState.OFF && Flags.Sign == Flags.Overflow,
Condition.GE => Flags.Sign == Flags.Overflow,
Condition.L => Flags.Sign != Flags.Overflow,
Condition.LE => Flags.Zero == FlagState.ON || Flags.Sign != Flags.Overflow,

// If overflow is set. Little mathematical meaning.
Condition.O => Flags.Overflow == FlagState.ON,
Condition.NO => Flags.Overflow == FlagState.OFF,

// If result negative/sign flag set.
Condition.S => Flags.Sign == FlagState.ON,
Condition.NS => Flags.Sign == FlagState.OFF,

// If even parity. Only set by some opcodes, a very rare use case.
// i.e If the number of bits set in the first byte is even.
Condition.P => Flags.Parity == FlagState.ON,
Condition.NP => Flags.Parity == FlagState.OFF,
_ => true, //Condition.None
};
```

The test condition functions works by accepting a condition code as a parameter then performing a switch selection construct to use the appropriate predicate for that condition. These conditions are mostly based on flags and are the same as defined in the intel x86-64 specification. A boolean true or false is returned based on the result of the condition.

Test condition code justification

This is the best approach to implementing this task because it allows for the solution to be reused. This implementation also allows the single responsibility design success criterion to be met. The problem of testing conditions has been delegated to this specific function in this class. This is important because many opcodes will require these tests, and it will be very consistent; all opcodes will have the same result for a condition. If this was split into different functions across opcode classes, it would be very likely that code would become outdated and begin to have incorrect and invalid outputs, causing logical errors in the affected opcodes.

Disassemble opcode and operands explanation

```
private readonly string Mnemonic;

public virtual List<string> Disassemble()
{
    // Function purpose: Associate the mnemonic of the opcode with the
    // disassembly of the operands, forming a complete disassembled instruction
    // to be pipelined into the disassembly modules

    // Start with the mnemonic
    List<string> Output = new List<string>() { Mnemonic };

    // Add the disassembly of the input
    Output.AddRange(Input.Disassemble());

    return Output;
}
```

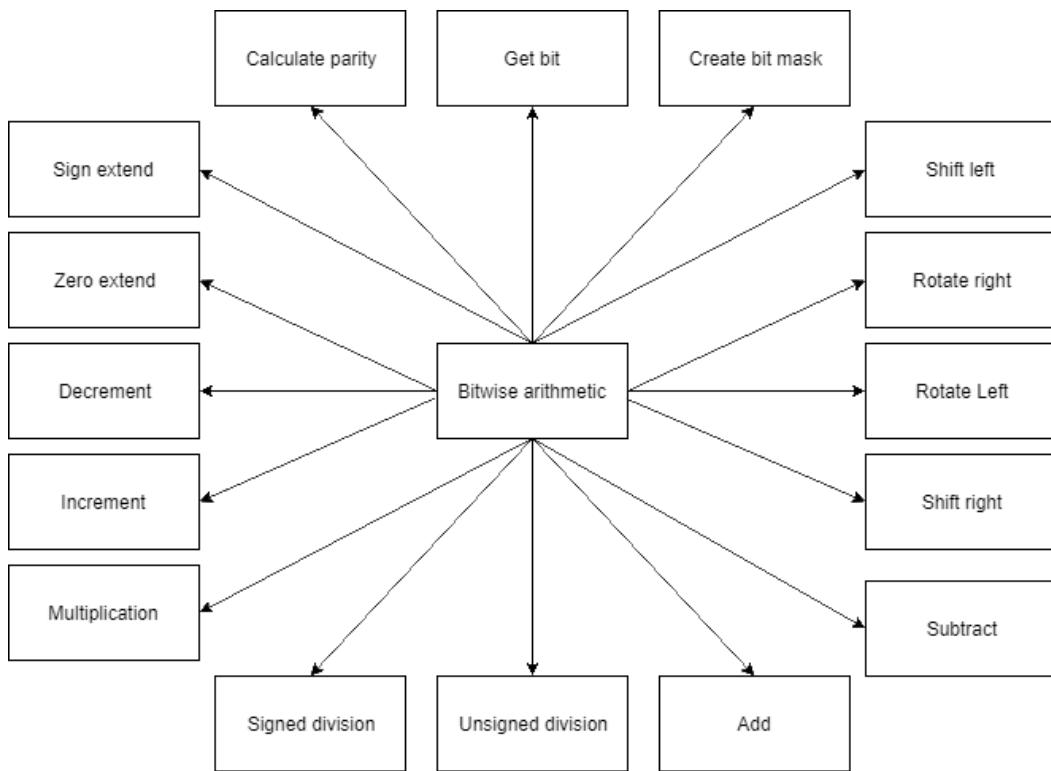
The solution to this problem has been implemented by using the methods of existing features, such as `input.Disassemble()`. The mnemonic variable is a required parameter of the constructor for the base class. This means that every opcode must define their own mnemonic, which will allow them to be disassembled. This function is used to produce the disassembled lines that are an input to the Disassembler class.

Disassemble opcode and operands justification

This is the best approach to implementing the solution to this problem because it allows the open/closed principle success criterion to be met. This is because the opcode base class uses a simple and standardised method for disassembly. This is important because any new opcodes implemented in the future will automatically have the same assembly convention applied as any existing opcode; no extra code is required. However, it is also possible to override this method as the `virtual` keyword is used. This means that any future modules that operate very differently will still be able to extend the functionality of the opcode base class because they can override this function(if required) to use their own code that solves their problem, such that the code in the opcode base class will not have to be modified after release.

Bitwise library

Structure explanation

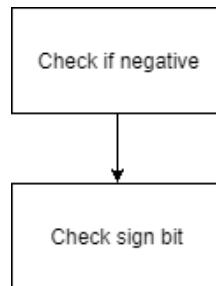


The structure of the library has been implemented as planned. The library is in a single source file as part of the util namespace. It has all the procedures that were planned in the design section.

Structure justification

This is the best approach to implementing the bitwise library because the components will be reusable throughout the solution, which will allow a lot of development time to be saved as solutions to problems do not have to be coded or tested for every individual implementation.

IsNegative explanation



```

public static bool IsNegative(this byte[] input)
{
    // Two tricks for determining if a byte array is negative or not.
    // Firstly, if $input.Length is not a power of two, it cannot be negative. This could be
    // the case where another method returned the least number of bytes in $input possible,
    // e.g instead of returning { [0x22] [0x44] [0x66] [0x88] [0xAA] [0x00] [0x00] [0x00] },
    // it returned, { [0x22] [0x44] [0x66] [0x88] [0xAA] } (cut off the trailing null bytes),
    // it would appear that the byte array represented a negative number because the MSByte (0xAA)
    // has its sign bit(MSB) set, [10101010]. This trick works by doing a bitwise AND on $input.Length
    // and $input.Length - 1. This works because any number that is a power of two has one bit set,
    // and a number that is one less than a power of two has all bits lower than said power of two set.
    // For example,
    // 0x08 = [1000]
    // 0x07 = [0111]
    // 0x08 & 0x07 = [0000]
    // or,
    // 0x80 = [10000000]
    // 0x7F = [01111111]
    // The second trick is simply to check if the MSB of $input is on. Masking by 0x80([10000000])
    // does exactly that. Naturally as seen in the example,
    // this wouldn't work if the previous trick hadn't been done prior.
    return (input.Length & input.Length - 1) == 0 && (input[input.Length - 1] & 0x80) == 0x80;
}
  
```

IsNegative justification

The best approach to implementing this algorithm was to use the two planned methods of determining a negative. This allows the KISS principle success criterion to be met; no advanced methods are required, such as converting to an integer. The code has also been heavily commented to explain what the function does and how it works incase there is an ambiguity.

IsNegative testing

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	Test a signed negative outputs true	A signed negative will be entered as the input parameter	input = [FF, FF]	Normal	True	This test is necessary to make sure that the function works under normal conditions as other tests could not be reliable if the algorithm does not work on valid input.

```
// The second trick is simply to check if the MSB of $input is on. Masking by 0x80([10000000])  
// does exactly that. Naturally as seen in the example,  
// this wouldn't work if the previous trick hadn't been done prior.  
return (input.Length & input.Length - 1) == 0 && (input[input.Length - 1] & 0x80) == 0x80;  
    (input.Length & input.Length - 1) == 0 && (inputlic static bool IsZero(this byte[] input)
```

The output is true, therefore the test passed, hence no remedial actions required

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
2	Test a signed positive outputs false	A signed positive will be used as the input parameter	input = [FF, 0F]	Normal	False	This test will make sure that the algorithm for detecting the sign of the input works for both negatives and positives. Test #1 would not reveal the possibility that true is output every time due to a logical error.

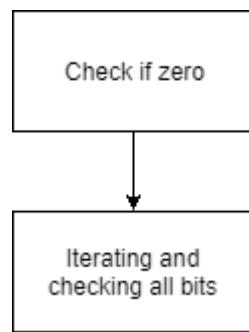
```
// The second trick is simply to check if the MSB of $input is on. Masking by 0x80([10000000]).  
// does exactly that. Naturally as seen in the example,  
// this wouldn't work if the previous trick hadn't been done prior.  
return (input.Length & input.Length - 1) == 0 && (input[input.Length - 1] & 0x80) == 0x80;
```

The output is false, therefore the test passed, hence no remedial actions required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
3	Test an erroneous input with a size that is not a multiple of two	An input that has the MSB set but is not a valid two's complement signed integer will be used.	input = [FF, FF, FF]	Erroneous	False	This test is necessary to ensure that the second IF condition(nested condition) correctly detects when the input is not a size that is a multiple of two.

```
// does exactly that. Naturally as seen in the example,  
// this wouldn't work if the previous trick hadn't been done prior.  
return (input.Length & input.Length - 1) == 0 && (input[input.Length - 1] & 0x80) == 0x80
```

IsZero explanation



```

public static bool IsZero(this byte[] input)
{
    // Iterate through every byte in the array, checking if it is zero.
    for (int i = 0; i < input.Length; i++)
    {
        // If any index in $input is not equal to zero, the byte array cannot represent a zero.
        if (input[i] != 0)
        {
            return false;
        }
    }
    return true;
}

```

This function works by using the algorithm planned in the design section, to iterate through the input and compare each element with zero.

IsZero justification

This is the best approach to implementing the solution to this problem because the algorithm is very simple, does not require any specific conversions, and can be reused throughout the project. This will allow the single responsibility principle to be met as the purpose of the function will not be repeated across the code, such that this method is always used to check for a zero, rather than having several different algorithms spread across the solution that may be less efficient and overall require more testing and development time than if a single solution was reused.

IsZero testing

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	Test to see if zero valued inputs output true	A byte array filled with zeroes will be used as the input and the result will be checked using a breakpoint.	input = [00, 00]	Normal	True	This test is necessary to make sure that the evaluate zero values properly, as other tests will not be reliable if the function does not work with a normal input because this would likely be the cause of any other erroneous behaviour

```

public static bool IsZero(this byte[] input)
{
    // Iterate through every byte in the array, checking if it is zero.
    for (int i = 0; i < input.Length; i++)
    {
        // If any index in $input is not equal to zero, the byte array cannot represent a zero.
        if (input[i] != 0)
        {
            return false;
        }
    }
    return true;
}

```

The output is true, therefore the test passed, hence no remedial actions required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
2	Test to see if non-zero valued inputs output false	A byte array filled with non-zero values will be used as the input and the result will be checked using a breakpoint.	input = [AA, AA]	Normal	False	This test will make sure that the IF statement is functional, such that there is no logical in the condition code that would make the function always output true(zero).

```
public static bool IsZero(this byte[] input)
{
    // Iterate through every byte in the array
    for (int i = 0; i < input.Length; i++)
    {
        // If any index in $input is not equal to zero, then
        if (input[i] != 0)
        {
            return false;
        }
    }
    return true;
}
```

The output is false, therefore the test passed, hence no remedial actions required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
3	Test to see if boundary non-zero valued inputs output false	A byte array filled with zeroes, but the last byte a non-zero will be used and the result will be checked using a breakpoint.	input = [00, 00, 00, AA]	Boundary	False	This test data is necessary to make sure that the entire array is iterated and detect any logical errors such as the condition in the FOR loop being incorrect.

```
public static bool IsZero(this byte[] input)
{
    // Iterate through every byte in the array
    for (int i = 0; i < input.Length; i++)
    {
        // If any index in $input is not equal to zero, then
        if (input[i] != 0)
        {
            return false;
        }
    }
    return true;
}
```

The output is false, therefore the test passed, hence no remedial actions required.

ReverseEndian explanation

```
public static byte[] ReverseEndian(byte[] input)
{
    // Flip the byte order around
    Array.Reverse(input);

    return input;
}
```

The ReverseEndian method has been implemented very efficiently. The Array.Reverse() method provided by .NET has been reused to form a solution to the problem.

ReverseEndian Justification

It is still necessary to include this solution in the project because it allows code to be written more intuitively and concisely. For example,

```
if (!Core.TryParseHex(Core.ReverseEndian(input.Attribute("offset").Value), out OffsetBytes))
```

This if statement would otherwise be spread across many lines of code because `Array.Reverse` requires its own line. This means that the KISS principle can be met because the code is more readable and the meaning is more intuitive; `Array.Reverse` can be very vague, especially when it will be used in many different contexts, therefore, to separate this procedure from other methods and simplify the code, it is necessary to include it separately.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	Test that the output is the reverse of the input	An asymmetric byte array will be entered as an input and the output will be checked after using a breakpoint.	input = [AB,CD,EF]	Normal	Output = [EF,CD,AB]	This test data is necessary to make sure that the algorithm produces correct results and there are no logical errors in the implementation, such as the for loop iterating outside of the bounds of the array. It is also important that the test data is asymmetric because otherwise it would not be possible to tell whether the array had been reversed.

```
public static byte[] ReverseEndian(byte[] input)

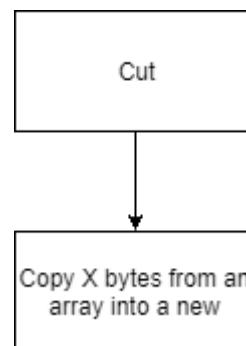
    // Flip the byte order around
    Array.Reverse(input);

    return input;
    ↗ input {byte[0x00000003]} ↘
public static byte[] Cut(byte[] input, int count)
    ↗ [0] 0xef ↗ te[] ↘ input, int count
    ↗ [1] 0xcd
    ↗ [2] 0xab ↗ Time diff ↗ linq: http://prntscr.com/od20o4
    ↗ linq: http://prntscr.com/od20vw
    ↗ my way: http://prntscr.com/od21br

    // A performant way to cut any excess bytes off whilst at the same time ensuring the desired size
    Array.Resize(ref input, count);
    return input;
}
```

The output was the same as the expected output, therefore the test passed, hence no remedial action is required.

Cut explanation



```
public static byte[] Cut(byte[] input, int count)
{
    // Time difference between this and linq.take is huge, http://prntscr.com/od20o4
    // linq: http://prntscr.com/od20vw
    // my way: http://prntscr.com/od21br

    // A performant way to cut any excess bytes off whilst at the same time ensuring the desired size
    Array.Resize(ref input, count);
    return input;
}
```

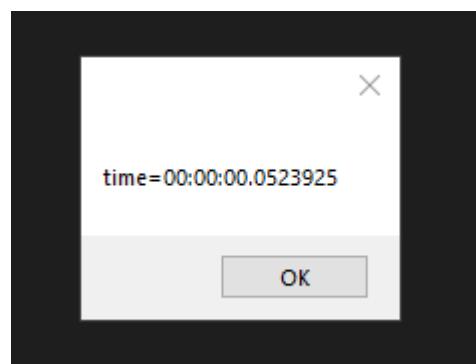
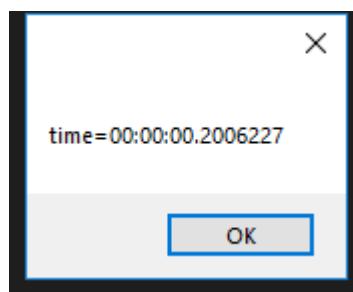
Cut works by resizing the input, hence cutting all bytes after the count off.

Cut justification

This is the best approach of implementing this function, as was shown through performance testing

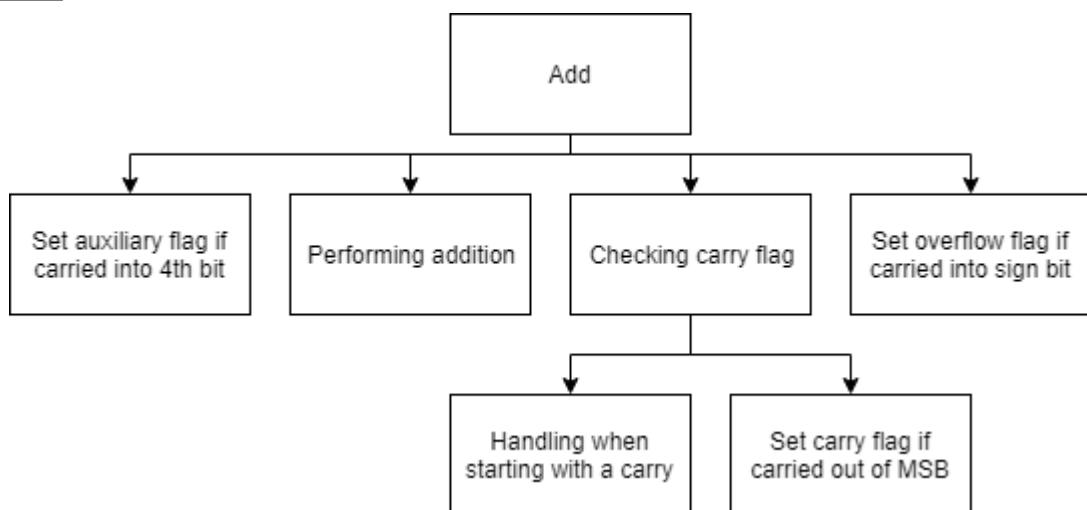
The time to execute using the algorithm planned in the design section

The time to execute using Array.Resize



Using the algorithm created in the design section turned out to be much slower than existing functions so there was no reason to use it. It is still necessary to include the function for simplicity, the same reason mentioned in reverse endian.

[Add explanation](#)



```
public static FlagSet Add(byte[] input1, byte[] input2, out byte[] Result, bool carry = false)
{
    // The two inputs must be the same length.
    Result = new byte[input1.Length];

    // Instead of using built-in methods, I use my own algorithms compatible with byte arrays increased performance
    // on critical operations that are frequently used. http://prntscr.com/ojwfs2
    for (int i = 0; i < Result.Length; i++)
    {
        // Declare a sum integer, which must be an integer because I anticipate values > 0xFF
        // , which are handled using carries shortly.
        // If there was a carry on the previous byte in the array
        // (or operation even, if the CF was set), add that to the sum.
        // Consider 9 + 1, the least significant digit overflows to 0
        // whilst the 1 carries to the next. It is impossible for the carry
        // to represent a value greater than one in the next column(e.g 9+9=18) in addition.
        int sum = input1[i] + input2[i] + (carry ? 1 : 0);
    }
}
```

```

// If the sum was greater than 0xFF, it can't be stored
// in the byte array without losing its value
if (sum > 0xFF)
{
    // Leftover value stays in the current column. For example, 9 + 5,
    // the left over value is 4, making 14. The "left over" can be calculated by
    // taking the modulo of the new sum and the greatest amount a byte can represent + 1.
    // Literally, dividing the sum and finding the remainder.
    Result[i] += (byte)(sum % 0x100);

    // Then account for this overflow by carrying to the next.
    carry = true;
}
else
{
    // Otherwise add normally, nothing special.
    Result[i] += (byte)sum;

    // If there was a carry, turn it off because it was already accounted for when the sum was calculated.
    carry = false;
}

return new FlagSet(Result)
{
    // If the last iteration carried, the addition is incomplete. Setting the flag allows the developer to handle this.
    Carry = carry ? FlagState.ON : FlagState.OFF,

    // If two numbers were added and the result had different sign, this could screw us over if we are doing signed addition.
    // For example, if I add two numbers A and B to make C, if both A and B were positive, C should also be positive. So if this is
    // the case, tell the developer by setting the OF.
    // Now if I wanted to add A and -B, there would never be a situation where the sign is incorrect. This is due to twos compliment. Say I take
    // the largest negative byte 0x80 and add the largest positive 0x7F, I get 0xFF, which proves the sum can never overflow to 0x100.
    Overflow = (input1.IsNegative() == input2.IsNegative() && Result.IsNegative() != input1.IsNegative()) ? FlagState.ON : FlagState.OFF,

    // The auxiliary flag is mostly a compatibility feature for older programs that use binary coded decimal to represent
    // numbers with decimal places(non-integers). This method used the lower nibble of a word to represent decimal digits.
    // Shortly put, BCD was not too efficient in terms of numbers, the lower nibble could only be used to represent 1-9, any greater
    // would be invalid and had undefined behaviour. To align with accurate emulation, I have implemented the flag, even though
    // BCD is actually only supported when the x86-64 processor is in compatibility mode. It is possible an end-user is dealing with BCD
    // through their own implementation--my program can do just that.
    // To check if the auxiliary flag is set, I mask the first byte of both inputs to get the lower 3 bits. If the sum of
    // these bits is greater than 0b111, or 7, it is clear that there was an overflow into the 4th bit.
    Auxiliary = ((input1[0] & 0b111) + (input2[0] & 0b111)) > 0b111 ? FlagState.ON : FlagState.OFF
};

```

The add algorithm has been implemented as planned in the design section. The for loop iterates through each column in the byte array (like long addition) adding each row. The boolean “carry” is used to denote a carry into the next byte. Like with decimal addition, the maximum single row sum, 9+9+1, is 19, therefore a boolean is appropriate because there will always be a carry of one or no carry at all (the same principle applies to base 256, or base x). The carry is then added to the sum variable in the next iteration. Sum has to be stored as an integer because if there is a carry, it will have overflowed the byte. Condition can then be used to check if there was a carry (value went above max size for a byte, 0xFF). (Flags are explained in comments)

Add justification

This is the best approach to implementing this algorithm because it has been possible to use specific data types in order to simplify the operation. For example, a true/false boolean can be used as a carry instead of having an integer, which saves space and because it has a small range of possible values, its purpose is more clear. This will be important to meeting the KISS principle design success criterion. This is because the add algorithm is one of the most complex algorithms in the source

Add testing

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	Test whether two inputs are added together correctly	Two normal inputs will be used as parameters to the function with the carry parameter false.	Input1 = [11,11] Input2 = [22,22] Carry = false	Normal	Output = [33,33] PF = true	This test is necessary to ensure that the algorithm works under normal conditions as it will be clear whether other tests show bugs or fundamental flaws in the algorithm. If the algorithm works in the general case, it could be deduced that there is a logical error in the code, which will speed up the bug

identification and fixing process.

The screenshot shows the Visual Studio Watch 1 window. It displays the variable 'Result' as a byte array [0x00000002] with elements [0] = 0x33 and [1] = 0x33. Other variables shown are 'OutputFlags' (PF) and 'InputFlags' (0x11, 0x11).

```
FlagSet OutputFlags = Util.Bitwise.Add(new byte[] { 0x11, 0x11 }, new byte[] { 0x22, 0x22 }, out byte[] Result, carry: false);
Result = Result;
// These initialisa
// use of these two
VMInstance = new VM
DisassemblerInstanc
// Draw the form.
SuspendLayout();
InitializeCompon
InitialiseCustom
```

The expected result was produced, hence no remedial action required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
2	Test whether two inputs are added together correctly, starting with a carry	Two normal inputs will be used as parameters to the function with the carry parameter true.	Input1 = [11,11] Input2 = [22,22] Carry = true	Normal	Output = [34,33] All flags off	This test will make sure that the algorithm handles starting with a carry correctly when specified in the parameter, such that the carry is applied to the LSB of the result.

The screenshot shows the Visual Studio Watch 1 window. It displays the variable 'Result' as a byte array [0x00000002] with elements [0] = 0x34 and [1] = 0x33. Other variables shown are 'OutputFlags' ({}), 'InputFlags' (0x11, 0x11), and 'Carry' (true).

```
FlagSet OutputFlags = Util.Bitwise.Add(new byte[] { 0x11, 0x11 }, new byte[] { 0x22, 0x22 }, out byte[] Result, carry: true);
Result = Result;
// These initialisa
// use of these two
VMInstance = new VM
DisassemblerInstanc
// Draw the form.
SuspendLayout();
InitializeCompon
InitialiseCustom
```

The expected result was produced, hence no remedial action required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
3	Check that inputs of different sizes will throw an exception	Two erroneous inputs that have different sizes to each other.	Input1 = [11] Input2 = [22,22]	Erroneous	ArrayOutOfBoundsException	This test will show whether the exception is thrown when the inputs are invalid instead of producing an invalid result that could cause bugs later in execution that would be harder to find as opposed to having the exception point it out clearly.

The screenshot shows the Visual Studio Watch 1 window. It displays the variable 'Result' as a byte array [0x00000001] with element [0] = 0x33. Other variables shown are 'OutputFlags' (PF) and 'InputFlags' (0x11, 0x11).

```
FlagSet OutputFlags = Util.Bitwise.Add(new byte[] { 0x11 }, new byte[] { 0x22, 0x22 }, out byte[] Result);
Result = Result;
// These initialisa
// use of these two
VMInstance = new VM
DisassemblerInstanc
// Draw the form.
SuspendLayout();
```

This test did not pass, there was no exception thrown. Remedial action is required.

Remedial actions

Evidence 1

```

public static FlagSet Add(byte[] input1, byte[] input2, out byte[] Result, bool carry = false)
{
    // The two inputs must be the same length.
    Result = new byte[input1.Length]; ←

    // Instead of using built-in methods, I use my own algorithms compatible with byte arrays i
    // on critical operations that are frequently used. http://prntscren.com/ojwfs2
    for (int i = 0; i < Result.Length; i++)
        ...
}

```

Evidence 2

```

// To represent a value greater than one in the next column (e.g. 0x10-10) in addition.
int sum = input1[i] + input2[i] + (carry ? 1 : 0); ✘

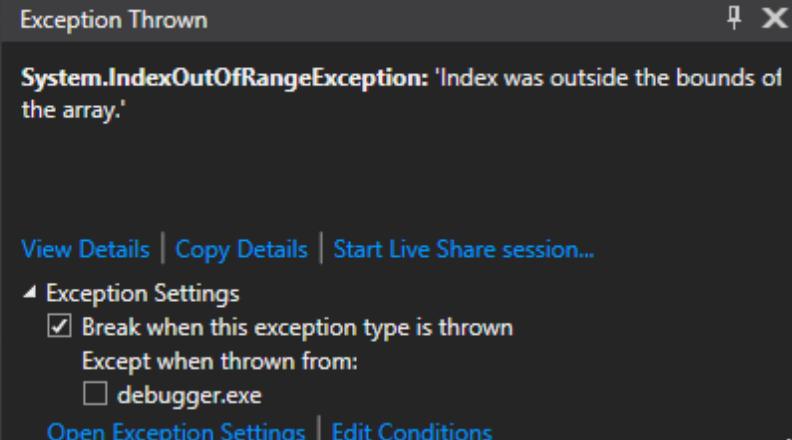
```

```

// If the sum was greater than 0xFF, it can't be stored
// in the byte array without losing its value
if (sum > 0xFF)
{
    // Leftover value stays in the current column.
    // the left over value is 4, making 14. The "length"
    // taking the modulo of the new sum and the greatest
    // Literally, dividing the sum and finding the remainder
    Result[i] += (byte)(sum % 0x100);

    // Then account for this overflow by carrying to the next column
    carry = true;
}
else
{
}

```



```

FlagSet OutputFlags = Util.Bitwise.Add(new byte[] { 0x22, 0x22 }, new byte[] { 0x11 }, out byte[] Result);
Result = Result;

```

Evidence 3

```

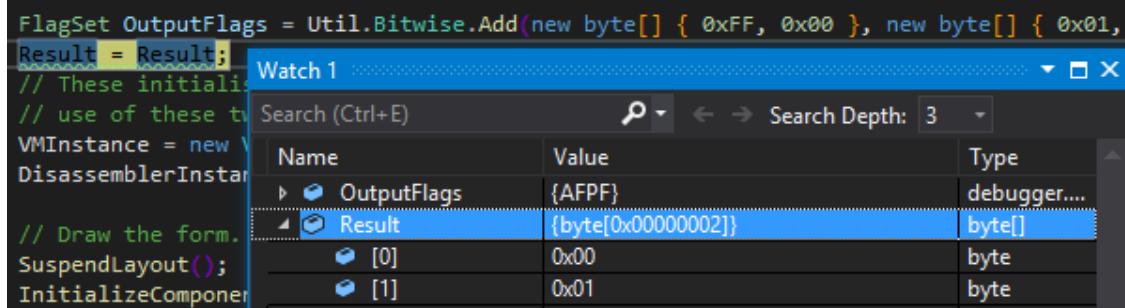
}
public static FlagSet Add(byte[] input1, byte[] input2, out byte[] Result, bool carry = false)
{
    // Validate that the two strings are both the same length.
    if(input1.Length != input2.Length)
    {
        throw new Exception("Invalid input array sizes; they are not equal length");
    }
}

```

Remedial action	Explanation	Justification
(Evidence 1) Source code analysis	By viewing the source code, it is clear why the erroneous behaviour is present. As input1 was shorter than input2, the second byte of input2 to was simply ignored in the iteration, so was undetected. To confirm this suspicion, the test will be repeated with input1 longer than input2.	It is necessary to first analyse the source code to find the source of the error. As the add function alone was being tested, the module in which the bug is present was clear. Analysing this source code allowed me to deduce that the bug was in this particular function and not another function called by this function. This allowed me to remedy the issue faster and be able to get back to development quickly as I was able to narrow down the cause of the problem very fast.
(Evidence 2) Confirming source of error	The test was performed again using the inputs swapped around, which was predicted in the previous remedial action, would cause the error(desired behaviour) in this case. This was proven correct as shown in the evidence.	It was necessary to identify that this was the source of the problem. This is because the code needs to be amended to fix the bug. If this extra test was not performed, it would not be certain that the bug intended to be fixed had been fixed. This could

		have caused further problems later if the wrong code was updated and caused more erroneous behaviour, or concealed the error to only be present in certain conditions.
(Evidence 3) Implementation of validation	As the solution to the bug is now apparent, the means to remedy the bug can be applied. This is done by adding a conditional if statement to validate that the arrays are of the same length.	This step was necessary in the remedial process because it will ensure that the developers are alerted of the erroneous function inputs. This will be important for detecting bugs throughout development because it will be clear whether or not the invalid input was present at this point, before this patch was applied there would have only been an error in certain cases where the length of input2 was greater than input1. With this solution, both can be detected.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
4	Test that the carry boolean will carry onto the next byte properly	Two normal inputs will be used as the input that will cause the carry boolean to be used where the bit needs to be carried into the next byte.	Input1 = [FF, 00] Input2 = [01,00] Carry = false	Normal	Output = [00,01] PF = true AF = true	This test will be necessary to show that the carry boolean part of the algorithm is carrying the bits correctly. This kind of logical error would be hard to detect based on the other tests because they do not cause a carry as their purpose is different. Having two separate tests will round down the possibilities causing the problem to a greater extent.



```

FlagSet OutputFlags = Util.Bitwise.Add(new byte[] { 0xFF, 0x00 }, new byte[] { 0x01, 0x00 }, out byte[] Result);
Result = Result;
// These initiali
// use of these t
VMInstance = new
DisassemblerInstan
// Draw the form.
SuspendLayout();
InitializeComponen

```

Name	Type
OutputFlags	debugger...
Result	byte[]
[0]	byte
[1]	byte

The expected result was produced, hence no remedial action required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
5	Check whether a carry out of the MSB sets the carry flag	Two normal inputs that will cause the MSB to carry out, in turn resulting in the CF set in the result flags.	Input1 = [00, FF] Input2 = [00, 01] Carry = false	Normal	Output = [00,00] CF = true PF = true ZF = true	This test will be necessary to show that the IF condition that determines whether the CF is set in the output has any logical errors. It will also make sure that there is no strange behaviour as the MSB is carried out, e.g an exception.

Name	Value	Type
OutputFlags	{CFZFPF}	debugger....
Result	{byte[0x00000002]}	byte[]
[0]	0x00	byte
[1]	0x00	byte

The expected result was produced, hence no remedial action required.

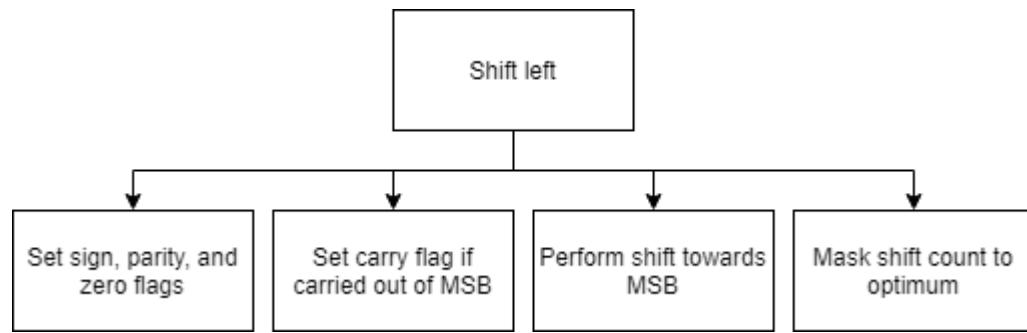
Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
6	Check whether a carry into the sign bit/MSB sets the overflow flag	The input will be two normal inputs that will cause a carry into the sign bit, resulting in the OF set in the result flags.	Input1 = [7F] Input2 = [01] Carry = false	Normal	Output = [80] SF = true OF = true	This test will show whether the predicate to detect an overflow is successful or not at detecting when the addition of a two of the same sign results in the other sign. This has to be separated from other tests as they may unintentionally affect the OF, resulting in an invalid test.

Name	Value	Type
OutputFlags	{CFZFPF}	debugger....
Result	{byte[0x00000001]}	byte[]
[0]	0x80	byte

Review

Reviewer	Comment
Self review	This function has been implemented successfully. It has been possible to meet the KISS principle. To demonstrate the success of this, I will have some stakeholders review the code.
Stakeholder 1 C# Programmer	"The implementation of this algorithm is very clear. I haven't seen the pseudocode for this algorithm yet, but based on what I've read the procedure is very simple and follows a natural approach. I think you have succeeded in keeping the code simple and is a much better structure than having a complex add algorithm. I think now you should move on to implementing your other bitwise algorithms to expand your available feature set."
Stakeholder 2 (different)C# programmer	"I like this routine, the inputs and outputs are laid out very clearly and there is lots of comments that definitely help understanding the more complex parts of the algorithm and also explain their purpose, such as why the different flags are set. I find this code very educational, I myself have learned about assembly, a language I have never used before."

ShiftLeft explanation



```
public static FlagSet ShiftLeft(byte[] input, byte count, out byte[] Result)
{
    // A massive performance boost can made here by doing *something* to $count.
    // Think about how many bits are in an 8 byte value, on x86-64, 64. If I shift 64 times, what am I doing?
    // Wasting cycles. This is really clever trick to improve performance at a hardware level. If I bitshift a QWORD
    // by any number greater than 64, I get (Hold the thought of where the carry flag is used), 0. There is no room
    // for those bits to go after 63. To shift by 63 is to make the LSB the MSB. If you are trying to zero a register,
    // I believe the best way is xor rax, rax.
    // Now for when the size isn't a QWORD, it is only "optimised" for DWORDS. If I could guess as to why, I would
    // say it's because you would hardly ever shift a WORD or BYTE, there isn't much space to move. Shifts are
    // often used to multiply by a big power of two quickly. So, WORDs and BYTEs may waste cycles, but DWORDs
    // follow the exact same logic as described for QWORDs.
    count &= (byte)(input.Length == 8) ? 0b00111111 : 0b00011111;

    // This gets a little fiddley, flags later are checked based on this, so I ought to save a copy.
    int StartCount = count;

    Result = input.DeepCopy();

    // Don't bother shifting by 0, just return now without changing the flags(as said by Intel)
    if (count == 0)
    {
        return new FlagSet();
    }

    // Pull is a term coined by myself to denote a carry in shift instructions. I think it describes the idea a little more, because it wouldn't really
    // be a carry. It has to be initialised here or compiler moans because ""it may not be initialised"". I already returned from the method if
    // count was zero!!!
    bool Pull = false;

    // So, what's my thinking with this algorithm.
    // Briefly, I will explain what a bitwise shift is.
    // Know how multiplying an integer by ten is really easy? All I do is add a zero
    // to the end. This is a cool phenomenon that happens when you multiply a number
    // by the base its represented in. So a single shift could be described as this notion
    // of multiplying by two, but most of the time we're going to want to shift a bunch
    // to get some big numbers. So, this could be written as
    // (the number I want to multiply) * 2^(the number of shifts)
    // This is exactly the same as
    // (the number I want to shift) << (the number of shifts)
    // But how does this to translate to being easy? 10 shift 2 is 40!
    // It's only easy when the result is represented in the base as well.
    // Lets show this example graphically,
    // 10 << 0 = 10 = [0001010]
    // 10 << 1 = 20 = [0010100]
    // 10 << 2 = 40 = [0101000]
    // ^ Look, these just jumped along each time
    // Or another way of thinking of it would be
    // 10 << 0 = 10 = 1010
    // 10 << 1 = 20 = 10100
    // 10 << 2 = 40 = 101000
    // They are both correct but the former really is a better way to get it into your
    // head because when we're dealing with numbers mathematically, there are no numbers
    // but we need to apply the hardware limitations(or advantages depending on how you see it)

    // Now the algorithm.
    // What does pull really describe?
    // A "pull" describes the situation where a bit has to jump from one byte in the
    // array to the next in the direction of the MSB. Nobody else has every dreamed
    // of this because it's just not a problem that exists in lower level languages.
    // Lets show see this graphically
    // I want to shift 128 by 1, but I have a problem
    // 128 = [10000000]
    // 128 << 1 = 0??? = [00000000]
    // Yes, it definitely would if I'm shifting a byte, but fortunately for purposes of
    // explanation we will be dealing with a word capacity, so more accurately 128 could be written as
    // 128 = [00000000] [10000000]
    // So pull denotes the idea of this jump, where 128 becomes 256
    // 128 << 1 = 256 = [00000001] [00000000]
```

```

// Fortunately if we are clever there is a good way to predict this.
// Each outer iteration, where count is decremented, denotes a shift. I handle one
// shift at a time, I don't think it would be impossible to do it more efficiently, but I challenge you
// to do so. So immediately it becomes obvious how I can predict if there will be a Pull, simply by the
// fact the MSB is 1. If I shift the byte as-is, I'm going to lose that MSB, so I'm going to save it
// in the Pull boolean and come back to it next nested iteration. Now I will explain how the PullMask
// works. If I shift a number, what is one thing I can be absolutely sure of? That the number will
// be even. Moreover, the LSB will be 0. Remember that I am multiplying by 2 each time, just very effectively.
// So, if I need to pull a byte up a power(remember that each index in the array represents a power of 256),
// I can bitwise OR the next index by one to preserve the bit pushed out of the previous byte by the shift.
// This is exactly how the PullMask works, if there is no Pull, the mask is 0, it does nothing.
// Take careful note of how the shift is in brackets. I don't know or care of the operator priority here,
// but I'm trying to make it realy clear that I shift BEFORE ORing. There is no guarentee that the LSB will
// be zero if I mask before shifting, and I will also get the wrong value because the bit that was already
// shifted by the pull mechanism will be shifted once more. (Also the most likely outcome in my code is
// that without brackets, 1 will be ORed by PullMask, which would always be 1)
for (; count > 0; count--)
{
    Pull = false;
    for (int i = 0; i < Result.Length; i++)
    {
        int PullMask = Pull ? 0b1 : 0;
        Pull = (MSB(Result[i]) == 1);
        Result[i] = (byte)((Result[i]) << 1) | PullMask;
    }
}

```

```

// Create some flags, use the constructor for a generic arithmetic flag set, e.g set SF, ZF, and PF as per usual.
FlagSet ResultFlags = new FlagSet(Result);
if (StartCount == 1)
{
    // The overflow flag is actually defined to be set as the MSB ^ CF but since the carry flag is
    // equal to Pull, I just substituted for ease. Think of it as the CF.
    // Here I will explain what exactly these flags represent.
    // The overflow flag is absolutely useless. I can think of no reason or purpose for
    // it here, I can only assume it meant something tens of years ago and was just kept
    // for compatibility, like you will find with a lot of things in x86_64.
    // Then the CF just shows the value of the last bit to be pushed out of the result.
    // This could still have some use on shifts greater than 1, but I imagine the most
    // common use is to check for a sign bit that got pushed out.
    ResultFlags.Overflow = (MSB(Result) == 1) ^ Pull ? FlagState.ON : FlagState.OFF;
}
ResultFlags.Carry = Pull ? FlagState.ON : FlagState.OFF;
return ResultFlags;

```

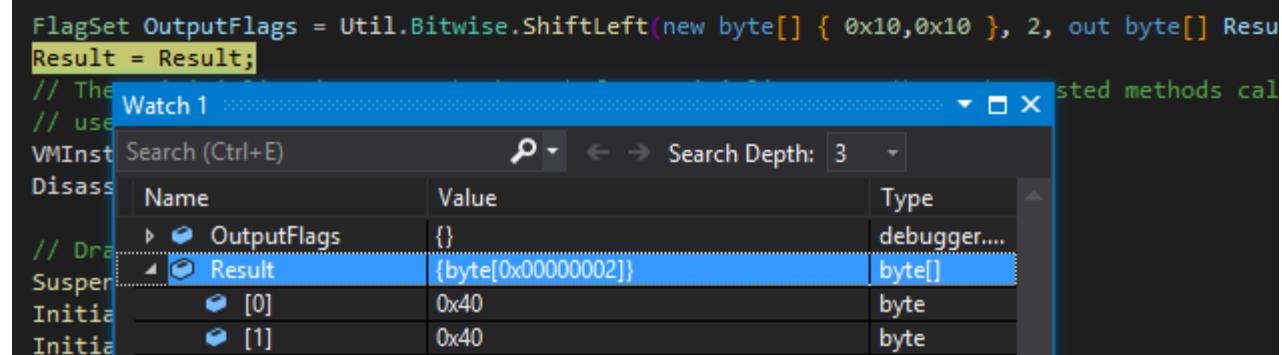
Firstly some validation must be applied in the algorithm. This is explained more in the comments, but the purpose is optimisation, and is also part of the x86-64 specification. If a number represented in bits is shifted enough times, it is certain that the output will be empty. This means that a bitmask can be used to make sure that this maximum value is not exceeded, because otherwise time would be wasted shifting zeroes. The main part of the algorithm works similar to the add algorithm. The pull boolean is used to denote carry of sort. The purpose of this is to handle when the MSB of one byte is shifted out of the byte and into the next array index, which is done by adding it onto the LSB in the next iteration.

ShiftLeft justification

This is the best approach to implementing this algorithm because it allows components to be reused and adapted. This is because the add algorithm can be taken and adapted to form the shift implementation. This is important because it links to the KISS principle design success criterion. Users reading the code and trying to understand the algorithms used will find it much easier because it is so similar to the add algorithm. Because of this they will be able to learn more from the code , therefore also contributing towards the “The executable program and source code should be a great learning tool for the stakeholders.” success criterion.

ShiftLeft testing

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	To test the iterative shifting procedure in the algorithm	The core algorithm will be tested separately from other tests. The parameter will be a byte array shifted twice	Input = [10,10] ShiftCount = 2	Normal	Output = [40,40]	This test will be necessary to determine whether the algorithm is implemented correctly. This means that it is important to use a simple test initially to show which parts of the algorithm are known to function.



The expected result was produced, hence no remedial action required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
2	To test whether the flags will be changed when the input shift count is zero	The ShiftCount parameter will be set to zero and a breakpoint will be used to check the flags afterwards.	ShiftCount = 0 (Input irrelevant)	Boundary	Empty flag set returned	This test data is necessary for this test because it is the only case where the behaviour will appear. This will be testing whether this special case will be handled correctly, as no other test would be able to demonstrate this.

```
// Don't bother shifting by 0, just return a new FlagSet()
if (count == 0)
{
    return new FlagSet();
}
```

The if block designed to detect the empty count successfully detected the zero shift, hence no remedial action required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
3	Test the output of the carry flag	The input will be an array that will have its MSB pushed out of the array on the last iteration, such that the carry flag will be expected to be set	Input = [FF] ShiftCount = 1	Normal	Output = [FE] CF = True	This test data will always be an input that should result in the CF being set in the output. Therefore, this data is appropriate for this test. The test will identify any implementation bugs of the conditions that determine the carry flag such that any bugs can be found and fixed faster than if they had to be identified by other means.

```

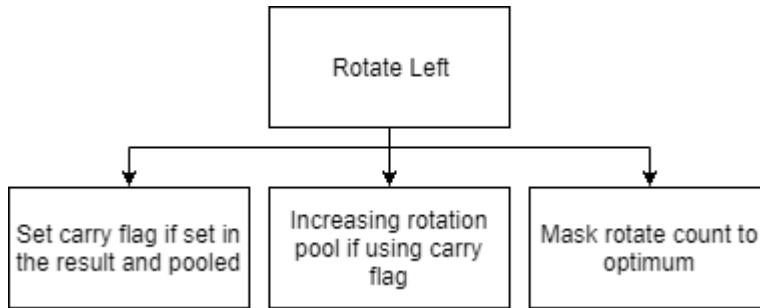
FlagSet OutputFlags = Util.Bitwise.ShiftLeft(new byte[] { 0xFF }, 1, out byte[] Result);
Result = Result; Watch 1 ...
// These initiali
// use of these t
VMInstance = new
DisassemblerInsta
// Draw the form.

```

Name	Value	Type
OutputFlags	{CFSF}	debugger....
Result	{byte[0x00000001]}	byte[]
[0]	0xfe	byte

The expected result was produced, hence no remedial action required.

RotateLeft explanation



```

public static FlagSet RotateLeft(byte[] input, byte bitRotates, bool useCarry, bool carryPresent, out byte[] Result)
{
    // Firstly, we need to make sure we know what a bitwise rotation does. If you don't know what a shift is, read ShiftLeft() first.
    // When we shift, there is a good chance our bits are going to be pushed out the result. What if we could shift a value, then
    // take all the bits pushed out and put them in the empty space created by the shift, which would be equal to the number of bits
    // pushed out. Consider we are working on some single bytes,
    // 255      = [11111111]
    // 255 << 1 = [11111110]
    // When I left shifted 255 by 1, I created an empty LSB and the MSB went into oblivion. What if I kept that MSB and said,
    // I want to wrap this back around, so if I did this mystery operation on 128, I would get 1.
    // 128      = [10000000]
    // 128 ? 1 [00000001]
    // This socalled "mystery operation" is exactly Rotate Left. Lets go back to 255,
    // 255      = [11111111]
    // 255 << 1 = [11111110]
    // 255 ROL 1 = [11111111]
    // 255 ROL 9123192389 trillion = [11111111]
    // Naturally this works for more than one bit,
    // 96      = [01100000]
    // 96 ROL 1 = [11000000]
    // 96 ROL 2 = [10000001]
    // 96 ROL 3 = [00000011]
    // 96 ROL 1027 = [00000011]
    // But how did I do that in my head? Read ahead.
    byte StartCount;
}

```

```

// What exactly is the purpose of useCarry?
// There is a certain variation of ROL called RCL, Roll carry left juggles the value that would be pushed to bit 1
// into the carry flag instead. In effect, we get an extra bit to work with.
// Lets go back to 128, the extra [] represent the CF.
// 128      = [10000000]
// 128 ROL 1 = [00000001]
// 128 RCL 1 = [1][00000000]
// 128 ROL 2 = [00000010]
// 128 RCL 2 = [0][00000001]
// This effectively gives us a 65-bit number to work with.
if (useCarry)
{
    // Like with shifting, we can save a lot of time by masking the bitRotates
    // input to eliminate any unnecessary operations. Let's say I want to ROL
    // a value by 8. Well, all the bits are going to go in a full circuit and
    // give the same result.
    // 128      = [10000000]
    // 128 ROL 8 = [10000000]
    // Consider the example earlier, where I did 96 ROL 1027 in two seconds.
    // Well, I know 1024 is some multiple of 8, so I think, how many OVER that
    // did I go, that's 3 right? 7-4 = 3. So if I modulo the count by 8, I can
    // save a tonne of time right?
    StartCount = (RegisterCapacity)input.Length switch
    {
        RegisterCapacity.BYTE => (byte)((bitRotates & 0x1F) % 9),
        RegisterCapacity.WORD => (byte)((bitRotates & 0x1F) % 17),
        RegisterCapacity.DWORD => (byte)(bitRotates & 0x1F),
        RegisterCapacity.QWORD => (byte)(bitRotates & 0x3F),
        _ => throw new Exception(),
    };
}

// Now you may have noticed, I moduloed by 9 not 8. It took me a minute
// to realise that this reduction is only done in RCL not ROL! It would be nice
// if they included this optimisation for, and I would guess they do at a hardware level,
// or is too inefficient, but let's go with what we know.
// If you're still unsure as to why its 9 and 17, think of the CF as an extra bit
// and apply the exact same principle.
// 128      = [0][10000000]
// 128 RCL 8 = [0][01000000]
// 128 RCL 9 = [0][10000000]
}

else
{
    // Unlike before, we only optimise by masking not using modulo. There is a possibility that on a hardware level, to modulo
    // by 8 is simply not worth the performance tradeoff.
    StartCount = (byte)(bitRotates & (((RegisterCapacity)input.Length == RegisterCapacity.QWORD) ? 0x3F : 0x1F));
}

// Exit early if no shifts
if (StartCount == 0)
{
    Result = input;
    return new FlagSet();
}

```

```

// Shift a deep copy of the input to prevent the input from being changed because of its reference.
Result = input.DeepCopy();

// Set this if there is a carry flag already. If the instruction is ROL, this is set but never used.
bool Carry = carryPresent;

// Pre-set if there is already a carry flag. This is the value that would be pushed into LSB
// This is done a little differently if I'm not using the carry flag. Or could be said that I only
// have to worry about this if I am using the carry flag.
bool Pull = carryPresent && useCarry;

// Each iteration of $RotateCount rotates the result by one, so do that as many times
// as I want to rotate in total.
for (byte RotateCount = 0; RotateCount < StartCount; RotateCount++)
{
    // Iterate over each byte--bits can be handle at a bigger scale.
    for (int i = 0; i < input.Length; i++)
    {
        byte Mask = (byte)(Pull ? 0b1 : 0);
        Pull = MSB(Result[i]) == 1;
        Result[i] = (byte)((Result[i] << 1) | Mask);
    }
}

// Since I'm rotating a byte, I care about every bit in said byte, so I need to handle the pull before the damage is done.
// If there is a carry,
// 1. Set the LSB of the result if there is a carry
// 2. Set the carry to be the previous MSB(before rotation). This will be stored in the pull boolean
//     and if I was shifting, I would just discard that right here because it wouldn't be needed, but
//     since I want to rotate, it loops right back round.
// If there is no carry,
// 1. Don't have to worry about step 1 of before, I just set the LSB to the MSB before rotation.
// Lastly, turn the pull off because now I'm going to do a completely different rotation.
// The algorithm isn't interdependent on iterations--you could take the result out of any $RotateCount
// iteration and get a result equal to the input array rotated $RotateCount times. Compare this to division
// and the difference becomes obvious.
if (useCarry)
{
    Result[0] |= (byte)(Carry ? 1 : 0);
    Carry = Pull;
}
else
{
    Result[0] |= (byte)(Pull ? 0b1 : 0);
}
Pull = false;
}

// "The SF, ZF, AF, and PF flags are always unaffected." - Intel manual May 2019 edition
FlagSet ResultFlags = new FlagSet();
if (useCarry)
{
    // This is where the carry flag is set in the output. I don't have to worry about setting it each
    // time in the loop, I can get away with using a boolean. This is because my program performs what is called atomic operation.
    // It really gets advanced here, but a processor can skip ahead in the code and do operations that are not interdependent on each other.
    // Let's say I have this code,
    // mov eax, 0x10
    // rol eax, 0x1F
    // mov ebx, 0x20
    // rol ebx, 0x1F
    // Lets also pretend that our processor isn't very fast. It definitely still applies to even the fastest processors
    // , little tricks like this make our flagship models really fast. But despite being really slow, our pretend processor
    // has 2 cores. We have all been told that "multiple cores don't make single processes faster" but it's really wrong.
    // Lets look a little deeper at the code given, I'm moving 10 into $eax, and rotating it left by 31, then setting $ebx
    // to 32 and doing the same. But these two operations are completely independent of each other. The first two instructions
    // affect the second two in no way at all. So, why not delegate these to core 2, so effectively we can execute this piece
    // of code theoretically in half the time. However what about with RCL?
    // 0x0 mov al, 0x80
    // 0x2 rcl al, 1
    // 0x4 mov bl, 0x80
    // 0x6 rcl bl, 1
    // 0x8 nop
}

```

```

// Remember that 0x80 in binary is [10000000]
// If we ran this atomically(in series as you see it), 0x2 would set the carry flag on. Then at 0x6 the carry flag would already
// be set, so it would loop round to the LSB, but also the MSB would be pushed into the carry flag. So at 0x8, our results would be
// $AL = 0
// $BL = 1
// $CF = 1
// And this is absolutely correct, but what if I tried this inatomically and say that I delegate instructions 0x0 and 0x2 to core 1, 0x4 and 0x6 to
// core 2. This would give a different result! Assuming that the CF was not already set, the core 2 wouldn't know that after core 1
// finishes, the CF would be set, so as a result there would have been no CF set to rotate into $BL, therefore the result would be
// $AL = 0
// $BL = 0
// $CF = 1(because the both cores would set the CF on, that doesn't mean it would be equal to 2. The CF is actually just a bit in a
// register. Think of it as setting a boolean to true twice, you did the same thing twice but nothing changes)
// So this demonstrates why both RCL instructions cannot be executed inatomically in this scenario. However, some things could.
// Whilst 0x2 is being executed by core 1, core 2 could execute 0x4, then once core 1 is finished it can move on to 0x6.
// Despite being a super cool low level concept, I don't incorporate it into my program. Why? It defeats the purpose entirely.
// Imagine trying to debug some code, then randomly another part of your program starts executing and changing stuff, that would be a massive pain.
// Most importantly, when I'm debugging assembly code, I don't care about speed. Its never going to take any amount of considerable
// time to execute a single instruction. Performance is only really important to the end-user. Also, you have to think about why you
// are learning assembly. If I didn't care about how I can take full advantage of the processor, I wouldn't be writing my code in assembly.
ResultFlags.Carry = Carry ? FlagState.ON : FlagState.OFF;
}

else
{
    // If the carry flag wasn't used in rotation because ROL was used, set it to the LSB of the result. As to why, I don't know. I can't
    // think of a scenario where I would use this, but Intel writes it as "For ROL and ROR instructions, the original value of the CF
    // is not part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other".
    ResultFlags.Carry = LSB(Result) > 0 ? FlagState.ON : FlagState.OFF;
}
}

if (StartCount == 1)
{
    // As with in ShiftLeft(), I don't know when this would be used. I assume compatibility.
    // The Intel manuals are very cookie-cutter, they don't explain much, they just state what happens.
    ResultFlags.Overflow = (MSB(Result) ^ (ResultFlags.Carry == FlagState.ON ? 1 : 0)) == 0 ? FlagState.OFF : FlagState.ON;
}
return ResultFlags;

```

At the start of the function, similar to shift, some optimisation procedures are applied per the intel specification, “ The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1)”[8]. This works on the premise that rotating a byte 8 times takes it to its original position. To modulo the value by 8(or however many bits are in the input) is a general solution to this optimisation. However, this is only performed with hardcoded values for some register capacities, as is shown in the x86-64 specification pseudocode.

The rotate implementation uses an adapted version of the shift function with extra checks after the shift. This uses selection to determine whether a bit was shifted out of the MSB, as then it would need to be looped back round to the LSB using an OR statement. Although the two operations are different, this is the only pragmatic difference between the two. If the carry parameter is used, the CF will be used as a bit in the rotation pool above the MSB(so the CF will become the MSB). Abstraction has been applied here as the CF does not need to be modified directly in the CU.Flags because nothing will access the information whilst the rotate function is executing, therefore a boolean is instead used to represent the CF then its final state stored afterwards in the result.

RotateLeft justification

This is the best implementation of the algorithm planned in the design section because it allows blocks of code to be reused. For example, the iteration block of the shift algorithm. It was not necessary to use the shift function directly because the extra information required by the selection functions after that apply the rotation effect would have been abstracted by the time the function returns. Moreover, these functions are very unlikely to be changed in the future because they are the implementation of an arithmetic algorithm; other modules will instead be adapting to suit the needs of this method. This can be seen as an instance of the single responsibility principle design success criterion. This is because the responsibility of the function will be to “Rotate left” and no further purpose. As this implementation performs its designed purpose, it will have no reason to change in the future, hence the developed solution will be more robust because the library functions will have been heavily tested and used for a long time rather than having the functions updated often, leading to less reliable code.

RotateLeft testing

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
1	Test the iterative left rotation procedure contained in the for loop	The function will be called with pre-defined input parameters then a breakpoint will be set afterwards to check the result	Input = [12,34] RotateCount = 4	Normal	Output = [23, 41]	This test data will demonstrate whether the implementation of the algorithm is successful. This will involve outlining any logical errors, which are likely as the algorithm gets complex with nested iterations. Therefore, it is necessary to test the components of the problem separately such that it can be identified at a later date which component contains flaws or bugs.

Name	Type	Value
OutputFlags	debugger...	{CF}
Result	byte[]	{byte[0x00000002]}
[0]	byte	0x23
[1]	byte	0x41

The expected result was produced, hence no remedial action required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
2	To test whether the flags will be changed when the input rotate count is zero	The ShiftCount parameter will be set to zero and a breakpoint will be used to check the flags afterwards.	RotateCount = 0 (Input irrelevant)	Boundary	Empty flag set returned	This test data is necessary for this test because it is the only case where the behaviour will appear. This will be testing whether this special case will be handled correctly, as no other test would be able to demonstrate this.

Name	Type	Value
StartCount	int	0
Result	byte[]	{byte[0x00000002]}
[0]	byte	0x23
[1]	byte	0x41

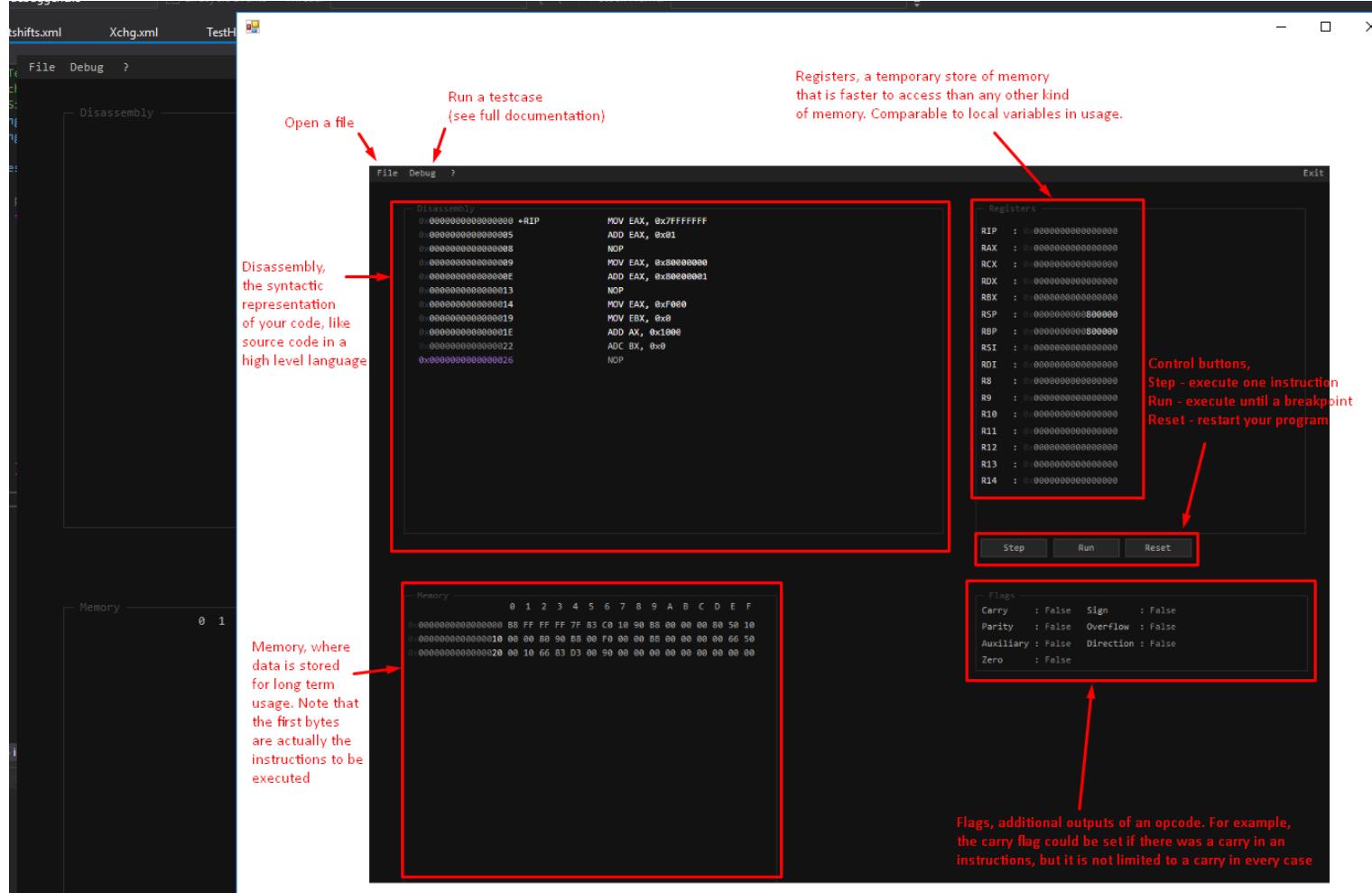
The if block designed to detect the empty count successfully detected the zero shift, hence no remedial action required.

Test #	Aim	How	Test data	Normal /Erroneous/Boundary	Expected result	Justification
3	Test the effect of the useCarry boolean parameter	The function will be called with the useCarry and carryPresent variables set and the input containing bytes with the MSB set to rotate, then the output checked afterwards using a breakpoint	Input = [00,80] RotateCount = 1 useCarry = true carryPresent = true	Normal	Output = [01, 00] CF = true OF = true	This test is necessary to check whether the implementation of the algorithm works correctly when the CF is used in the rotation pool. This test data is appropriate because it has been specially crafted to test every part of the algorithm that involves the carry flag. As the CF is on in the input(carryPresent), it is expected that the bit will be rotated into the LSB of the array. The array also having the MSB set will cause the CF to be set in the output, such that this test covers the entire scope of the carry flag in the procedure.

FlagSet OutputFlags = Util.Bitwise.RotateLeft(new byte[] { 0x00, 0x80 }, 1, true, true, out byte[] Result);
Result = Result; Watch 1

Name	Type	Value
OutputFlags	{CFOF}	
Result	byte[0x00000002]	
[0]	byte	0x01
[1]	byte	0x00

Help button



Explanation

A usability feature has been added that allows the user to click on an icon that tells them where to find features of the program and a description of what each element of the user interface represents. This is implemented by having a "?" button on the toolbar, which is a very common standard for help in a program and is easy to find for users.

Justification

Previous reviews showed that the program became too advanced for new users as new features were implemented. This was shown to be a problem in the alternative software that was researched, so it was important to add necessary features to overcome the problem that users did not understand how to use the software. This is the best way of implementing the solution as it is discrete so does not interfere with users who know how to use the program, but very accessible and obvious in purpose.

Reviewer	Comment
Classmate - python programmer	"It was a good call to add this feature and the implementation has also been successful. At early stages of development, it was easier to understand because I already had a good idea of what the program does, but I didn't keep up to date with the new features. I think this would definitely help a new user who didn't already know there way around the program or was new to assembly"

Opcodes

```
public class Cbw : Opcode
{
    public Cbw(OpcodeSettings settings = OpcodeSettings.NONE) : base("CBW", new ControlUnit.RegisterHandle(XRegCode.A, RegisterTable.GP), settings)
    {
    }
    public override void Execute()
    {
        // Fetch the operand. There will only be one because it is always the A register which is destination and source.
        byte[] Bytes = Fetch()[0];

        // Half the fetched bytes(because the capacity is that of the destination, see summary), and sign extend it
        // back to its former size.
        // E.g, in pseudo,
        // CBW()
        // {
        //     Bytes = $AX;
        //     Bytes = Cut($Bytes, 1);
        //     AX = $Bytes;
        // }
        // From here it would be very simple to create CWDE and CDQE methods.
        // Fortunately because of the interface the Opcode base class provides, this can be generalised.
        Set(Bitwise.SignExtend(Bitwise.Cut(Bytes, (int)Capacity / 2), (byte)Capacity));
    }
}
```

```
// Clx represents clc and cld; Clear carry and Clear direction. Simply each flag is set off respectively.
namespace debugger.Emulator.Opcodes
{
    public class Clc : Opcode
    {
        public Clc(DecodedTypes.NoOperands input, OpcodeSettings settings = OpcodeSettings.NONE) : base("CLC", input, settings)
        {
        }
        public override void Execute()
        {
            ControlUnit.SetFlags(new FlagSet() { Carry = FlagState.OFF });
        }
    }
    public class Cld : Opcode
    {
        public Cld(DecodedTypes.NoOperands input, OpcodeSettings settings = OpcodeSettings.NONE) : base("CLD", input, settings)
        {
        }
        public override void Execute()
        {
            ControlUnit.SetFlags(new FlagSet() { Direction = FlagState.OFF });
        }
    }
}
```

```

// Cmp; Compare. Compare two operands. It can give a range of information, not just equality, see
// Disassembly and the Opcode base class for more information on condition codes.
using debugger.Util;
using System.Collections.Generic;
namespace debugger.Emulator.Opcodes
{
    public class Cmp : Opcode
    {
        public Cmp(DecodedTypes.IMyDecoded input, OpcodeSettings settings = OpcodeSettings.NONE)
            : base("CMP", input, settings)
        {
        }

        public override void Execute()
        {
            // Fetch operands
            List<byte[]> DestSource = Fetch();

            // Subtract operand 1 from operand 0 and discard the result, keeping only the flags. This is
            // the essence of a compare in assembly.
            FlagSet Result = Bitwise.Subtract(DestSource[0], DestSource[1], out _);
            ControlUnit.SetFlags(Result);
        }
    }
}

// Cmps; compare strings. The string of bytes at DI will be compared with the string of bytes at SI.
// Most commonly will be seen with a REPZ/REPNZ prefix to determine the equality of an entire string
// rather than just one word-scale length.
using debugger.Util;
using System.Collections.Generic;
namespace debugger.Emulator.Opcodes
{
    public class Cmps : StringOperation
    {
        private FlagSet Result;
        public Cmps(StringOpSettings settings = StringOpSettings.NONE)
            : base("CMPS", XRegCode.DI, XRegCode.SI, settings | StringOpSettings.COMPARE) { }
        protected override void OnInitialise()
        {
            List<byte[]> DestSource = Fetch();

            // Carry out the subtraction and discard the results (That is how comparison works in assembly, it has the same flags as subtraction).
            Result = Bitwise.Subtract(DestSource[0], DestSource[1], out _);
        }
        protected override void OnExecute()
        {
            // Set flags
            ControlUnit.SetFlags(Result);

            // Adjust DI and SI, because both were used. See base class.
            AdjustDI();
            AdjustSI();
        }
    }
}

// Load string. Load a string pointed to by the SI register into the A register.
namespace debugger.Emulator.Opcodes
{
    public class Lods : StringOperation
    {
        public Lods(StringOpSettings settings = StringOpSettings.NONE)
            : base("LODS", XRegCode.A, XRegCode.SI, settings)
        {
        }
        protected override void OnInitialise()
        {
        }
        protected override void OnExecute()
        {
            // Set the destination to the source, in this case, set A to the bytes at SI.
            // The base class will automatically dereference the pointer.
            Set(Fetch()[1]);

            // SI was used therefore must be adjusted.
            AdjustSI();
        }
    }
}

```

```
// Move string, a string operation for moving a string of bytes from one place to another.
namespace debugger.Emulator.Opcodes
{
    public class Movs : StringOperation
    {

        public Movs(StringOpSettings settings = StringOpSettings.NONE)
            : base("MOVS", XRegCode.DI, XRegCode.SI, settings) { }

        protected override void OnInitialise()
        {

        }

        protected override void OnExecute()
        {
            // Set the destination to the source. The source is stored in Fetch()[1] by convention.
            Set(Fetch()[1]);

            // Both must be adjusted as the opcode uses both.
            AdjustDI();
            AdjustSI();
        }
    }
}
```

```
// Rotate right/rotate carry right. The carry flag will be considered as an extra bit in the rotation if RCR is used. See Bitwise.RotateRight()
using debugger.Util;
using System.Collections.Generic;

namespace debugger.Emulator.Opcodes
{
    public class Rxr : Opcode
    {
        private bool UseCarry;
        public Rxr(DecodedTypes.IMyDecoded input, bool useCarry, OpcodeSettings settings = OpcodeSettings.NONE) : base(useCarry ? "RCR" : "ROR", input, settings)
        {
            UseCarry = useCarry;
        }

        public override void Execute()
        {
            // Fetch operands
            List<byte[]> Operands = Fetch();

            // Perform the rotate and store the results. The callee will ignore "ControlUnit.Flags.Carry == FlagState.ON" if $UseCarry == false.
            byte[] Result;
            FlagSet ResultFlags = Bitwise.RotateRight(Operands[0], Operands[1][0], UseCarry, ControlUnit.Flags.Carry == FlagState.ON, out Result);

            // Set the results.
            Set(Result);
            ControlUnit.SetFlags(ResultFlags);
        }
    }
}
```

```

// Rotate left, or Rotate carry left. RCL uses the carry flag as an extra bit in the rotation. See Bitwise.RotateLeft().
using debugger.Util;
using System.Collections.Generic;

namespace debugger.Emulator.Opcodes
{
    public class Rxl : Opcode
    {
        private bool UseCarry;
        public Rxl(DecodedTypes.IMyDecoded input, bool useCarry, OpcodeSettings settings = OpcodeSettings.NONE) : base(useCarry ? "RCL" : "ROL", input, settings)
        {
            UseCarry = useCarry;
        }
        public override void Execute()
        {
            // Fetch operands
            List<byte[]> Operands = Fetch();

            // Perform rotation. See Bitwise.RotateLeft().
            byte[] Result;
            FlagSet ResultFlags = Bitwise.RotateLeft(Operands[0], Operands[1][0], UseCarry, ControlUnit.Flags.Carry == FlagState.ON, out Result);

            // Set results.
            Set(Result);
            ControlUnit.SetFlags(ResultFlags);
        }
    }
}

// Scan string, or more practically, compare a string at *DI with the value of the A register. Like all string ops,
// this size is inferred from prefixes/opcodes, specifics can be found in the base constructor. This is a comparison
// operator, therefore uses REPZ/REPNZ rather than REP.
using debugger.Util;
using System.Collections.Generic;
namespace debugger.Emulator.Opcodes
{
    public class Scas : StringOperation
    {
        FlagSet ResultFlags;
        public Scas(StringOpSettings settings = StringOpSettings.NONE)
            : base("SCAS", XRegCode.DI, XRegCode.A, settings | StringOpSettings.COMPARE)
        {
        }
        protected override void OnInitialise()
        {
            // Fetch operands
            List<byte[]> Operands = Fetch();

            // Subtract and store the result flags. Like cmp, only the flags are stored.
            ResultFlags = Bitwise.Subtract(Operands[1], Operands[0], out _);
        }
        protected override void OnExecute()
        {
            // Set flags
            ControlUnit.SetFlags(ResultFlags);

            // See base class.
            AdjustDI();
        }
    }
}

```

```

// Setcc, or set, will set a pointer or register to the result of a condition.
// If the condition is true it will be set to 0, otherwise 1. This value is sign extended to the size of the register
// or size of memory the pointer points to, whichever applicable. In many high level languages, a boolean true is -1,
// i.e., every bit in the byte is set, and all zeroes for a boolean false i.e no bit in the byte is set.
// A clever trick to mimic this with this opcode is to use the opposite condition that you are testing for, e.g.,
// if comparing which value is greater, instead of using SETA(set above), use SETBE(set below or equal), then decrement
// by one with DEC. This will give 0 if the intended test was false and -1(0xFF) if true.
using debugger.Util;

namespace debugger.Emulator.Opcodes
{
    public class Set : Opcode
    {
        byte Result;
        public Set(DecodedTypes.IMyDecoded input, Condition setCondition, OpcodeSettings settings = OpcodeSettings.NONE)
            : base("SET" + Disassembly.DisassembleCondition(setCondition), input, settings | OpcodeSettings.BYTEMODE)
        {
            // Determine the value that will be set. 1 if the condition is true, 0 if false.
            Result = (byte)(TestCondition(setCondition) ? 1 : 0);
        }
        public override void Execute()
        {
            // Set the destination operand to the zero extended value of $result.
            // This would be interchangeable will sign extending, as the value would always be non-negative.
            Set(Bitwise.ZeroExtend(new byte[] { Result }, (byte)Capacity));
        }
    }

    // Test does an AND on two operands without storing the result. I have only ever seen it used for
    // checking if a register is zero, because the zero flag will be set if the discarded result is zero.
    // Similar to CMP.
    using debugger.Util;
    using System.Collections.Generic;

    namespace debugger.Emulator.Opcodes
    {
        public class Test : Opcode
        {
            public Test(DecodedTypes.IMyDecoded input, OpcodeSettings settings = OpcodeSettings.NONE) : base("TEST", input, settings)
            {

            }
            public override void Execute()
            {
                // Fetch operands
                List<byte[]> DestSource = Fetch();

                // AND the operands and discard the result
                FlagSet ResultFlags = Bitwise.And(DestSource[0], DestSource[1], out _);

                // Set the flags to the result.
                ControlUnit.SetFlags(ResultFlags);
            }
        }
    }
}

```

Explanation

The code segments above have explanations for each of the function as its easier to convey their purpose in context. ROL RCL and ROR RCR have been merged into two single subroutines. This is because only a parameter is required to indicate whether the carry flag is to be used when calling the bitwise function, it would be an unnecessary overcomplication to have two separate subroutines.

Justification

It was necessary to include the new opcodes at this point in the development as in the review of the last iterative development cycle, it was voiced by the stakeholders that the program lacked practical functionality and that the majority of the work had gone into the processes behind the program such as improving modularity and creating the library. To decide what to implement, I went back to the initial stakeholder survey where I asked what opcodes were desirable and added not only the suggested opcodes, but other opcodes that I would consider to be on a similar skill level in terms of accessibility in order to further achieve this goal than initially planned.

Review

Reviewer	Comment
Experienced OO Programmer 1	"This code has used lots of good paradigms and OO coding principles. As you planned, this has completely met your single responsibility principle. Each opcode clearly has its own class and is modular in nature, and could be swapped out/worked on separately. A great improvement from previous versions."
Experienced OO Programmer 2	"A great use of inheritance in this version. I like how you have exploited use of override features in OO to allow the same ".Execute()" method to be called on every opcode instead of requiring separate subroutine calls for each class. I think in the next versions you should aim to improve existing features as I think you've fully met the features you intended to include"
C# programmer unfamiliar with assembly	"I like how you have abstracted the opcodes into independent classes. This works really well as instead of having a jumble of code to work through, the code has been streamlined into small separate and manageable classes."

Opcode testing

Explanation

For this version, opcodes have been tested using the test handler. This has allowed the testcases designed in the design section to be tested against the current feature set. Screenshots of evidence from the output test file are included.

Add

Test #	Aim	How	Instructions used as input	Expected result
1	Test addition of immediate byte with 'A' register	The opcode variant to add an immediate with the 'A' register will be used.	add al,0x10 add ax,0x2000 add eax,0x30000 add rax,0x40000000	<Register id="A" size="1">10</Register> <Register id="A" size="2">2010</Register> <Register id="A" size="4">40032010</Register>

```

<CheckpointResult Tag="Imm A">
  <SubCheckpointResult result="Passed">
    <Expected>$AL=0x10</Expected>
    <Found>$AL=0x10</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$AX=0x2010</Expected>
    <Found>$AX=0x2010</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$EAX=0x40032010</Expected>
    <Found>$EAX=0x40032010</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

2	Test addition of immediate with memory	The opcode variant to add an immediate with memory at an address will be used.	add bl,0x10 add bh,0x10 add bx,0x1000 add ebx,0x30000 add rbx,0x40000000	<Register id="B" size="1">10</Register> <Register id="B" size="2">2010</Register> <Register id="B" size = "8">40032010</Register>
---	--	--	--	---

```

<CheckpointResult Tag="MI">
  <SubCheckpointResult result="Passed">
    <Expected>$BL=0x10</Expected>
    <Found>$BL=0x10</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$BX=0x2010</Expected>
    <Found>$BX=0x2010</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x40032010</Expected>
    <Found>$RBX=0x40032010</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

3	Test addition of sign extended word with dword	The opcode variant to sign extend and add an immediate with a dword will be used.	add dx,0xffff0 add ecx,0xffffffff2	<Register id="D" size="2">FFF0</Register> <Register id="C" size="4">FFFFFFF2</Register>
---	--	---	---------------------------------------	--

```

<CheckpointResult Tag="Sxt b">
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0xFFFF0</Expected>
    <Found>$DX=0xFFFF0</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$ECX=0xFFFFFFFF2</Expected>
    <Found>$ECX=0xFFFFFFFF2</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

4	Test addition of register with memory	The opcode variant to add a register with a memory address will be used. The values must first be moved into registers because adding immediates uses a separate opcode to adding registers to memory.	mov eax,0x10 mov ebx,0x2000 mov ecx,0x30000000 mov edx,0x0 add BYTE PTR [rsp],al add WORD PTR [rsp+0x1],bx add DWORD PTR [rsp+0x3],ecx	<Memory offset_register="SP">10</Memory> <Memory offset_register="SP" offset="1">0020</Memory> <Memory offset_register="SP" offset="3">00000030</Memory>
---	---------------------------------------	--	--	--

```

<CheckpointResult Tag="RM">
  <SubCheckpointResult result="Passed">
    <Expected>[RSP]={10}</Expected>
    <Found>[RSP]={10}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP+0x1]={0, 20}</Expected>
    <Found>[RSP+0x1]={0, 20}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP+0x3]={0, 0, 0, 30}</Expected>
    <Found>[RSP+0x3]={0, 0, 0, 30}</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

5	Test addition of memory with register	The opcode variant to add a memory at an address with a register will be used. (Note that the data mov'd into the registers from the	add BYTE PTR [rsp],al add WORD PTR [rsp+0x1],bx add DWORD PTR [rsp+0x3],ecx add al,BYTE PTR [rsp]	<Register id="A" size="1">20</Register> <Register id="B" size="2">4000</Register> <Register id="C" size="4">60000000</Register> <Register id="D" size="8">30000000</Register>
---	---------------------------------------	--	--	--

previous checkpoint is
reused)

```
<CheckpointResult Tag="MR">
  <SubCheckpointResult result="Passed">
    <Expected>$AL=0x20</Expected>
    <Found>$AL=0x20</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$BX=0x4000</Expected>
    <Found>$BX=0x4000</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$ECX=0x60000000</Expected>
    <Found>$ECX=0x60000000</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0x30000000</Expected>
    <Found>$RDX=0x30000000</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

6	Test addition of two registers	The opcode variant to add two registers will be used	<pre>mov eax,0x0 mov ebx,0x0 mov ecx,0x0 mov edx,0x33332211 add al,dl add bx,dx add ecx,edx</pre>	<Register id="A" size="1">11</Register> <Register id="B" size="2">2211</Register> <Register id="C" size="4">33332211</Register>
---	--------------------------------	--	---	---

```
<CheckpointResult Tag="Reg to reg">
  <SubCheckpointResult result="Passed">
    <Expected>$AL=0x11</Expected>
    <Found>$AL=0x11</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$BX=0x2211</Expected>
    <Found>$BX=0x2211</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$ECX=0x33332211</Expected>
    <Found>$ECX=0x33332211</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

7	Test addition with expected overflow	The add opcode will be used with operands crafted such that an overflow will be expected in the result	<pre>mov eax,0x7fffffff add eax,0x1</pre> <p>(will overflow from positive to negative) </p>	<Flag name="Overflow">1</Flag>
---	--------------------------------------	--	---	--------------------------------

```
<CheckpointResult Tag="Overflow">
  <SubCheckpointResult result="Passed">
    <Expected>OF$FAFPF</Expected>
    <Found>OF$FAFPF</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

8	Test addition with expected carry	The add opcode will be used with operands crafted such that a carry will be expected in the result	<pre>mov eax,0x80000000 add eax,0x80000001</pre> <p>(will overflow the maximum value for a qword) </p>	<Flag name="Carry">1</Flag>
---	-----------------------------------	--	--	-----------------------------

```
<CheckpointResult Tag="OF CF">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0x1</Expected>
    <Found>$RAX=0x1</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFOF</Expected>
    <Found>CFOF</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

9	Test addition with ADC opcode	The ADC opcode will be used instead of the add opcode, with the CF set initially	<pre>mov eax,0xf000 mov ebx,0x0 add ax,0x1000(cause CF to be set) adc bx,0x0 (should be one because CF set)</pre>	<Register id="B" size="2">1</Register> <Flag name="Carry">0</Flag>
---	-------------------------------	--	---	--

```
<CheckpointResult Tag="ADC">
  <SubCheckpointResult result="Passed">
    <Expected>$BX=0x1</Expected>
    <Found>$BX=0x1</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>
```

Sub/sbb

Test #	Aim	How	Instructions used as input	Expected result
1	Sub 'A' register by immediate	The opcode variant to subtract an immediate from the 'A' register will be used.	<pre>movabs rax,0x4444444433332211 sub al,0x22 sbb al,0x44 sub ax,0x4400 sbb ax,0x8800 sub eax,0x66660000 sbb eax,0xbbbb0000</pre>	<Register id="A" size="4">111256a8</Register>

```
<CheckpointResult Tag="A_Imm">
  <SubCheckpointResult result="Passed">
    <Expected>$EAX=0x111256A8</Expected>
    <Found>$EAX=0x111256A8</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

2	Sub 'A' register by sign extended dword immediate	The opcode variant to subtract a sign extended dword immediate from the 'A' register will be used.	<pre>movabs rax,0xf1f1f1f1f122222222 sub rax,0xffffffff80000000 sbb rax,0xffffffff8fffffe</pre>	<Register id="A" size="8">f1f1f1f212222223</Register> <Flag name="Carry">1</Flag>
---	---	--	---	---

```
<CheckpointResult Tag="A_Imm_sxt_dw">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0xF1F1F1F212222223</Expected>
    <Found>$RAX=0xF1F1F1F212222223</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFSFAF</Expected>
    <Found>CFSFAF</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

3	Sub register by immediate	The opcode variant to subtract an immediate from a register will be used.	<pre>movabs rbx,0x4444444433332211 sub bl,0x22 sbb bl,0x44 sub bx,0x4400 sbb bx,0x8800 sub ebx,0x66660000 sbb ebx,0xbbbb0000</pre>	<Register id="B" size="4">111256a8</Register>
<CheckpointResult Tag="Reg Imm"> <SubCheckpointResult result="Passed"> <Expected>\$EBX=0x111256A8</Expected> <Found>\$EBX=0x111256A8</Found> </SubCheckpointResult> </CheckpointResult>				
4	Sub register by sign extended dword immediate	The opcode variant to subtract a sign extended dword immediate from a register will be used.	<pre>movabs rbx,0xf1f1f1f122222222 sub rbx,0xffffffff80000000 sbb rbx,0xffffffff8fffffe</pre>	<Register id="B" size="8">f1f1f1f12222223</Register> <Flag name="Carry">1</Flag>
<CheckpointResult Tag="Reg Imm sxt dw"> <SubCheckpointResult result="Passed"> <Expected>\$RBX=0xF1F1F1F212222223</Expected> <Found>\$RBX=0xF1F1F1F212222223</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>CFSFAF</Expected> <Found>CFSFAF</Found> </SubCheckpointResult> </CheckpointResult>				
5	Sub/sbb register by sign extended byte	The opcode variant to subtract a sign extended byte immediate from a register will be used. Both sbb and sub opcodes are tested in this checkpoint.	<pre>movabs rax,0x4444444433332211 sub ax,0xff80 sub eax,0xfffffff90 sbb rax,0xfffffffffffffa0</pre>	<Register id="A" size="8">33332360</Register> <Flag name="Carry">1</Flag>
<CheckpointResult Tag="Imm sxt b"> <SubCheckpointResult result="Passed"> <Expected>\$RAX=0x33332360</Expected> <Found>\$RAX=0x33332360</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>CFPF</Expected> <Found>CFPF</Found> </SubCheckpointResult> </CheckpointResult>				

6	Sub/sbb memory at address by register	The opcode variant to subtract a register from a value in memory at an address will be used. Both sbb and sub opcodes are tested in this checkpoint.	<pre>movabs rax,0x4444444433332211 movabs rbx,0x8888888866664422 mov QWORD PTR [rsp],0xffffffffbbbb8844 sub al,bl sbb BYTE PTR [rsp],al sub ax,bx sbb WORD PTR [rsp],ax sub eax,ebx sbb DWORD PTR [rsp],eax sub rax,rbx sbb QWORD PTR [rsp],rax</pre>	<Register id="A" size="8">7777777866675689</Register> <Register id="B" size="8">8888888866664422</Register> <Memory offset_register="SP">50b8868887888888</Memory>
---	---------------------------------------	--	---	--

```
<CheckpointResult Tag="Reg Imm sxt dw">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0xF1F1F1F212222223</Expected>
    <Found>$RBX=0xF1F1F1F212222223</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFSFAF</Expected>
    <Found>CFSFAF</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

Mov

Test #	Aim	How	Instructions used as input	Expected result
1	Move immediate into register	The opcode variant to move an immediate into a register will be used.	<pre>mov al,0xf1 mov bx,0xf2f2 mov ecx,0xf333f3f3 movabs rdx,0xf4444444f444f4f4</pre>	<Register id="A" size="1">f1</Register> <Register id="B" size="2">f2f2</Register> <Register id="C" size="4">f333f3f3</Register> <Register id="D" size="8">F4444444F444F4F4</Register>

```
<CheckpointResult Tag="Immediate">
  <SubCheckpointResult result="Passed">
    <Expected>$AL=0xF1</Expected>
    <Found>$AL=0xF1</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$BX=0xF2F2</Expected>
    <Found>$BX=0xF2F2</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$ECX=0xF333F3F3</Expected>
    <Found>$ECX=0xF333F3F3</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0xF4444444F444F4F4</Expected>
    <Found>$RDX=0xF4444444F444F4F4</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

2	Move reg into mem	The opcode variant to move a register into a memory location will	<pre>mov BYTE PTR [rsp],al mov WORD PTR [rsp-0x2],bx mov DWORD PTR [rsp-0x6],ecx</pre>	<Memory offset_register="SP">F1</Memory> <Memory offset_register="SP" offset="FE">F2F2</Memory>
---	-------------------	---	--	---

		be used.	mov QWORD PTR [rsp-0xe],rdx	<Memory offset_register="SP" offset="FA">F3F333F3</Memory><Memory offset_register="SP" offset="F2">f4f444f4444444f4</Memory>
--	--	----------	-----------------------------	--

```

<CheckpointResult Tag="Reg to mem">
  <SubCheckpointResult result="Passed">
    <Expected>[RSP]={F1}</Expected>
    <Found>[RSP]={F1}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP-0x2]={F2, F2}</Expected>
    <Found>[RSP-0x2]={F2, F2}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP-0x6]={F3, F3, 33, F3}</Expected>
    <Found>[RSP-0x6]={F3, F3, 33, F3}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP-0xE]={F4, F4, 44, F4, 44, 44, 44, F4}</Expected>
    <Found>[RSP-0xE]={F4, F4, 44, F4, 44, 44, 44, F4}</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

3	Move with sign extension, reg to reg	The opcode variant to move a register into another register and sign extend when doing so will be used.	movsx bx,cl movsx edx,cx movsx rcx,bl movsx ecx,bx movsx rbx,cx movsxd rax,ebx	<Register id="A" size="8">ffffffffffff3</Register><Register id="B" size="8">ffffffffffff3</Register><Register id="C" size="4">fffff3</Register><Register id="D" size="4">fff3</Register>
---	--------------------------------------	---	---	--

```

<CheckpointResult Tag="Sign extension">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0xFFFFFFFFFFFFFF3</Expected>
    <Found>$RAX=0xFFFFFFFFFFFFFF3</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0xFFFFFFFFFFFFFF3</Expected>
    <Found>$RBX=0xFFFFFFFFFFFFFF3</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$ECX=0xFFFFFFF3</Expected>
    <Found>$ECX=0xFFFFFFF3</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$EDX=0xFFFFF3F3</Expected>
    <Found>$EDX=0xFFFFF3F3</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

4	Move with zero extension, reg to reg	The opcode variant to move a register into another register and zero extend when doing so will be used.	movzx bx,cl movzx edx,cx movzx eax,bl movzx ecx,bx movzx rbx,cx	<Register id="A" size="8">f3</Register><Register id="B" size="8">f3</Register><Register id="C" size="4">f3</Register><Register id="D" size="4">fff3</Register>
---	--------------------------------------	---	---	--

```

<CheckpointResult Tag="Zero extension">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0xF3</Expected>
    <Found>$RAX=0xF3</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0xF3</Expected>
    <Found>$RBX=0xF3</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$ECX=0xF3</Expected>
    <Found>$ECX=0xF3</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$EDX=0xFFFF3</Expected>
    <Found>$EDX=0xFFFF3</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Lea

Test #	Aim	How	Instructions used as input	Expected result
1	Load effective address into QWORD register	The opcode variant to load the effective address of a SIB byte into a register will be used.	mov rax,0x123 lea rbx,[rax*8+0xfffffff]	<Register id="B" size="8">1000917</Register>
2	Load effective address into DWORD register	The opcode variant to load the effective address of a SIB byte into a dword register will be used.	mov rax,0x123 lea ecx,[rax*2+0x33332200]	<Register id="C" size="8">33332446</Register>
3	Load effective address into WORD register, and cause overflow	The opcode variant to load the effective address of a SIB byte into a word register will be used, such that value is lost when doing so because of the capacity of a WORD.	mov rax,0x123 lea dx,[rax*4+0xfc00]	<Register id="D" size="8">8c</Register>

```

<CheckpointResult Tag="Win">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x1000917</Expected>
    <Found>$RBX=0x1000917</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RCX=0x33332446</Expected>
    <Found>$RCX=0x33332446</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0x8C</Expected>
    <Found>$RDX=0x8C</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Mul

Test #	Aim	How	Instructions used as input	Expected result
1	Mul two bytes in registers	The opcode variant to multiply two bytes in registers will be used.	mov bl,0x11 mov al,0x2 mul bl	<Register id="A" size="2">0022</Register> <Register id="D" size="2">0</Register> <Flag name="Carry">0</Flag> <Flag name="Overflow">0</Flag>
<CheckpointResult Tag="Byte">				
				<SubCheckpointResult result="Passed"> <Expected>\$AX=0x22</Expected> <Found>\$AX=0x22</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>\$DX=0x0</Expected> <Found>\$DX=0x0</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected></Expected> <Found></Found> </SubCheckpointResult> </CheckpointResult>
2	Mul two words in registers	The opcode variant to multiply two words in registers will be used.	mov bx,0x2222 mov ax,0x2211 mul bx	<Register id="A" size="2">C842</Register> <Register id="D" size="2">048A</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
<CheckpointResult Tag="Word">				
				<SubCheckpointResult result="Passed"> <Expected>\$AX=0xC842</Expected> <Found>\$AX=0xC842</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>\$DX=0x48A</Expected> <Found>\$DX=0x48A</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>CFOF</Expected> <Found>CFOF</Found> </SubCheckpointResult> </CheckpointResult>
3	Mul two dwords in register	The opcode variant to multiply two dwords in registers will be used.	mov ebx,0x33333333 mov eax,0x33332211 mul ebx	<Register id="A" size="4">8f5c2c63</Register> <Register id="D" size="4">0a3d6d36</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>

```

<CheckpointResult Tag="Dword">
  <SubCheckpointResult result="Passed">
    <Expected>$EAX=0x8F5C2C63</Expected>
    <Found>$EAX=0x8F5C2C63</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$EDX=0xA3D6D36</Expected>
    <Found>$EDX=0xA3D6D36</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFOF</Expected>
    <Found>CFOF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

4	Mul a QWORD register and a value in memory	The opcode variant to multiply a qword in the 'A' register and a qword in memory will be used.	<pre> mov rbx,0xb mov QWORD PTR [rsp],rbx movabs rax,0x4444444433332211 mul QWORD PTR [rsp] </pre>	<Register id="A" size="8">eeeeeeee333276bb</Register> <Register id="D" size="8">2</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
---	--	--	---	---

```

<CheckpointResult Tag="Qword from mem">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0xEEEEEEEE333276BB</Expected>
    <Found>$RAX=0xEEEEEEEE333276BB</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0x2</Expected>
    <Found>$RDX=0x2</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFOF</Expected>
    <Found>CFOF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Jmp/test/cmp

Test #	Aim	How	Instructions used as input	Expected result
1	Test condition checking of JMP/TEST/ CMP opcodes	Include a short program in a testcase that will use the three opcodes in a context where the outputs will be wrong if they are not functional	<pre> mov rbp,rsp mov rcx,0x20 // counter dec rcx // Decrement counter mov BYTE PTR [rbp+rcx*1+0x0],cl test rcx,rcx // check if counter == -1 jg 0xa // jump to dec cmp ecx,0xffffffff // check if counter == -1 jne 0xa // jump to dec </pre>	<Memory offset_register="BP" offset="FF">FF000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F</Memory>

```

<CheckpointResult Tag="Win">
  <SubCheckpointResult result="Passed">
    <Expected>$CL=0xFF</Expected>
    <Found>$CL=0xFF</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RBP-0x1]=[FF, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F]</Expected>
    <Found>[RBP-0x1]=[FF, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F]</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>ZFFF</Expected>
    <Found>ZFFF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Bitwise OR

Test #	Aim	How	Instructions used as input	Expected result
1	OR immediate into the 'A' register	Use the opcode variant of OR that implicitly uses the A register and an immediate	<pre>movabs rax,0xa5a5a5a5a5a5a5a5 or al,0x5a or ax,0x5a00 or eax,0x5a5a0000 or rax,0xffffffff80000000</pre>	<Register id="A" size="8">ffffffffffff</Register>
			<pre><CheckpointResult Tag="Immediate A"> <SubCheckpointResult result="Passed"> <Expected>\$RAX=0xFFFFFFFFFFFFFF</Expected> <Found>\$RAX=0xFFFFFFFFFFFFFF</Found> </SubCheckpointResult> </CheckpointResult></pre>	
2	OR immediate into register	Use the opcode variant of OR that uses a register and an immediate	<pre>movabs rbx,0xa5a5a5a5a5a5a5a5 or bl,0x5a or bx,0x5a00 or ebx,0x5a5a0000 or rbx,0xffffffff80000000</pre>	<Register id="B" size="8">ffffffffffff</Register>
			<pre><CheckpointResult Tag="Immediate"> <SubCheckpointResult result="Passed"> <Expected>\$RBX=0xFFFFFFFFFFFFFF</Expected> <Found>\$RBX=0xFFFFFFFFFFFFFF</Found> </SubCheckpointResult> </CheckpointResult></pre>	
3	OR immediate into memory	Use the opcode variant of OR that uses an immediate and a location in memory	<pre>mov ebx,0x11 mov BYTE PTR [rsp],bl mov ebx,0x2233 mov WORD PTR [rsp+0x1],bx mov ebx,0x44556677 mov DWORD PTR [rsp+0x4],ebx movabs rbx,0x8899aabcccddeeff mov QWORD PTR [rsp+0x9],rbx or BYTE PTR [rsp],0xee or WORD PTR [rsp+0x1],0xddcc or DWORD PTR [rsp+0x4],0xbbaa9988 or QWORD PTR [rsp+0x9],0xffffffff3221100</pre>	<Memory offset_register="SP">FF</Memory> <Memory offset_register="SP" offset="1">FFFF</Memory> <Memory offset_register="SP" offset="4">FFFFFF</Memory> <Memory offset_register="SP" offset="9">FFFFFFFFFFFF</Memory>
			<pre><CheckpointResult Tag="Immediate to mem"> <SubCheckpointResult result="Passed"> <Expected>[RSP]={FF}</Expected> <Found>[RSP]={FF}</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>[RSP+0x1]={FF, FF}</Expected> <Found>[RSP+0x1]={FF, FF}</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>[RSP+0x4]={FF, FF, FF, FF}</Expected> <Found>[RSP+0x4]={FF, FF, FF, FF}</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Expected> <Found>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Found> </SubCheckpointResult></pre>	

4	OR register into memory	Use the opcode variant of OR that uses a register and a location in memory	<pre> mov ebx,0x11 mov BYTE PTR [rsp],bl mov ebx,0x2233 mov WORD PTR [rsp+0x1],bx mov ebx,0x44556677 mov DWORD PTR [rsp+0x4],ebx movabs rbx,0x8899aabcccddeeff mov QWORD PTR [rsp+0x9],rbx mov bh,0xee or BYTE PTR [rsp],bh mov bx,0xddcc or WORD PTR [rsp+0x1],bx mov ebx,0xbbaa9988 or DWORD PTR [rsp+0x4],ebx movabs rbx,0x7766554433221100 or QWORD PTR [rsp+0x9],rbx </pre>	<Memory offset_register="SP">FF</Memory> <Memory offset_register="SP" offset="1">FFFF</Memory> <Memory offset_register="SP" offset="4">FFFFFFFF</Memory> <Memory offset_register="SP" offset="9">FFFFFFFFFFFFFF</Memory>
---	-------------------------	--	--	--

```

<CheckpointResult Tag="Reg to mem">
  <SubCheckpointResult result="Passed">
    <Expected>[RSP]={FF}</Expected>
    <Found>[RSP]={FF}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP+0x1]={FF, FF}</Expected>
    <Found>[RSP+0x1]={FF, FF}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP+0x4]={FF, FF, FF, FF}</Expected>
    <Found>[RSP+0x4]={FF, FF, FF, FF}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Expected>
    <Found>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Bit shifts

Test #	Aim	How	Instructions used as input	Expected result
1	Test the shift left algorithm	The SHL opcode will be used which uses the ShiftLeft() bitwise library method. The input will be crafted to only use the bare features of the algorithm, e.g. no OF or CF used.	<pre> mov cl,0x8 mov rax,0xff mov rbx,rax shl bl,1 mov rdx,rax shl dx,cl mov QWORD PTR [rsp],rax shl QWORD PTR [rsp],0x3f </pre>	<Register id="B" size="1">FE</Register> <Register id="D" size="2">FF00</Register>

```

<CheckpointResult Tag="Shift left">
  <SubCheckpointResult result="Passed">
    <Expected>$BL=0xFE</Expected>
    <Found>$BL=0xFE</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0xFF00</Expected>
    <Found>$DX=0xFF00</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP]={0, 0, 0, 0, 0, 0, 0, 80}</Expected>
    <Found>[RSP]={0, 0, 0, 0, 0, 0, 0, 80}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFSFPF</Expected>
    <Found>CFSFPF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

2	Test the setting of the OF in the shift right algorithm	The SHR opcode will be used which uses the ShiftRight() bitwise library method. The input will be crafted to cause the OF to be set in the output.	mov rbx,rax shr bl,1	<Register id="B" size="1">7F</Register> <Flag name="Overflow">1</Flag> <Flag name="Carry">1</Flag>
---	---	--	-------------------------	--

```

<CheckpointResult Tag="SHR OF">
  <SubCheckpointResult result="Passed">
    <Expected>$BL=0x7F</Expected>
    <Found>$BL=0x7F</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFOF</Expected>
    <Found>CFOF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

3	Test the setting of the CF in the shift right algorithm	The SHR opcode will be used which uses the ShiftRight() bitwise library method. The input will be crafted to cause the CF to be set in the output.	mov rdx,rax shr dx,cl	<Register id="D" size="1">0</Register> <Flag name="Carry">1</Flag>
---	---	--	--------------------------	---

```

<CheckpointResult Tag="SHR NO OF">
  <SubCheckpointResult result="Passed">
    <Expected>$DL=0x0</Expected>
    <Found>$DL=0x0</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CF</Expected>
    <Found>CF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

CBW/CWDE/CDQE

Test #	Aim	How	Instructions used as input	Expected result
1	Test converting a byte into a word	The CBW opcode will be used as an instruction.	(Used throughout entire testcase) mov ebx,0x80008080 mov al,bl cbw	<Register id="A" size="2">FF80</Register>
			<CheckpointResult Tag="CBW"> <SubCheckpointResult result="Passed"> <Expected>\$AX=0xFF80</Expected> <Found>\$AX=0xFF80</Found> </SubCheckpointResult>
< checkpointresult><="" td=""><td></td></br><>	
2	Test converting a word into a dword	The CWDE opcode will be used as an instruction.	mov ax,bx cwde	<Register id="A" size="4">FFFF8080</Register>
			<CheckpointResult Tag="CWDE"> <SubCheckpointResult result="Passed"> <Expected>\$EAX=0xFFFF8080</Expected> <Found>\$EAX=0xFFFF8080</Found> </SubCheckpointResult>
< checkpointresult><="" td=""><td></td></br><>	
3	Test converting a dword into a qword	The CDQE opcode will be used as an instruction.	mov eax,ebx cdqe	<Register id="A" size="8">ffffffff80008080</Register>
			<CheckpointResult Tag="CDQE"> <SubCheckpointResult result="Passed"> <Expected>\$RAX=0xFFFFFFFF80008080</Expected> <Found>\$RAX=0xFFFFFFFF80008080</Found> </SubCheckpointResult>
< checkpointresult><="" td=""><td></td></br><>	

IMUL

Test #	Aim	How	Instructions used as input	Expected result
1	Test signed multiplication of a byte	The IMUL instruction will be used with two byte registers, one implicitly being the 'A' register.	mov bl,0x11 mov al,0x2 imul bl	<Register id="A" size="2">0022</Register> <Register id="D" size="2">0</Register> <Flag name="Carry">0</Flag> <Flag name="Overflow">0</Flag>

```

<CheckpointResult Tag="Byte">
  <SubCheckpointResult result="Passed">
    <Expected>$AX=0x22</Expected>
    <Found>$AX=0x22</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0x0</Expected>
    <Found>$DX=0x0</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>

```

2	Test signed multiplication of a word	The IMUL instruction will be used with two word registers, one implicitly being the 'A' register.	mov bx,0x2222 mov ax,0x2211 imul bx	<Register id="A" size="2">C842</Register> <Register id="D" size="2">048A</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
---	--------------------------------------	---	---	---

```

<CheckpointResult Tag="Word">
  <SubCheckpointResult result="Passed">
    <Expected>$AX=0xC842</Expected>
    <Found>$AX=0xC842</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0x48A</Expected>
    <Found>$DX=0x48A</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFOF</Expected>
    <Found>CFOF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

3	Test signed multiplication of a dword from memory	The IMUL instruction will be used with a memory and implicitly the 'A' register	mov ebx,0x33333333 mov DWORD PTR [rsp],ebx mov eax,0x33332211 imul DWORD PTR [rsp]	<Register id="A" size="4">8f5c2c63</Register> <Register id="D" size="4">0a3d6d36</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
---	---	---	---	---

```

<CheckpointResult Tag="Dword from mem">
  <SubCheckpointResult result="Passed">
    <Expected>$EAX=0x8F5C2C63</Expected>
    <Found>$EAX=0x8F5C2C63</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$EDX=0xA3D6D36</Expected>
    <Found>$EDX=0xA3D6D36</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFOF</Expected>
    <Found>CFOF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

4	Test signed multiplication of a qword	The IMUL instruction will be used with two qword registers, one implicitly being the 'A' register.	mov rbx,0xb movabs rax,0x444444443332211 imul rbx	<Register id="A" size="8">eeeeeeee333276bb</Register> <Register id="D" size="8">2</Register> <Flag name="Carry">1</Flag> <Flag name="Overflow">1</Flag>
---	---------------------------------------	--	---	--

```

<CheckpointResult Tag="Qword">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0xEEEEEEEE333276BB</Expected>
    <Found>$RAX=0xEEEEEEEE333276BB</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0x2</Expected>
    <Found>$RDX=0x2</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CFOF</Expected>
    <Found>CFOF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

5	Test signed multiplication of two registers(not implicit A register)	The IMUL instruction will be used with two registers of all sizes.	movabs rax,0x444444443332211 mov bx,0x2222 imul bx,ax mov ecx,0x33333333 imul ecx,eax movabs rdx,0x4444444444444444 imul rdx,rax	<Register id="B" size="8">c842</Register> <Register id="C" size="8">08f5c2c63</Register> <Register id="D" size="8">320fedcbf259084</Register>
---	--	--	--	---

```

<CheckpointResult Tag="Reg to reg">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0xC842</Expected>
    <Found>$RBX=0xC842</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RCX=0x8F5C2C63</Expected>
    <Found>$RCX=0x8F5C2C63</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0x320FEDCBF259084</Expected>
    <Found>$RDX=0x320FEDCBF259084</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

6	Test signed multiplication of a reg and immediate, output into a different register(3 operand variant)	The IMUL instruction will be used with an immediate multiplied by a register, and output into a different register..	movabs rax,0x444444443332211 imul bx,ax,0x3 imul ecx,eax,0x3 mov QWORD PTR [rsp],rax imul rdx,QWORD PTR [rsp],0x3	<Register id="B" size="8">6633</Register> <Register id="C" size="8">99996633</Register> <Register id="D" size="8">cccccccc99996633</Register>
---	--	--	---	---

```

<CheckpointResult Tag="Reg*Imm to reg">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x6633</Expected>
    <Found>$RBX=0x6633</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RCX=0x99996633</Expected>
    <Found>$RCX=0x99996633</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0xCCCCCCCC99996633</Expected>
    <Found>$RDX=0xCCCCCCCC99996633</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Negate

Test #	Aim	How	Instructions used as input	Expected result
1	Test negation of dword and qword registers.	Move data into two separate registers then use the negate opcode on them. (Applicable to use one single testcase as not many registers are involved). ECX is empty and negated to test that the CF is set.	<pre> mov rax,0xffffffffffffffffff neg rax mov ebx,0x11111111 neg rbx neg ecx </pre>	<Register id="A" size="8">1</Register> <Register id="B" size="8">ffffffffeeeeeeef</Register> <Flag name="Carry">1</Flag>

```

<CheckpointResult Tag="Win">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0x1</Expected>
    <Found>$RAX=0x1</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0xFFFFFFFFFEEEEEEF</Expected>
    <Found>$RBX=0xFFFFFFFFFEEEEEEF</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CF</Expected>
    <Found>CF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Push and pop

Test #	Aim	How	Instructions used as input	Expected result
1	Push/pop word and qword at a pointer in memory	Push the data onto the stack at a pointer to a byte/word using the opcode variant that performs this task. (The specific sizes of memory are used because pop/push only has variants for these specified sizes)	<pre>movabs rax,0aaaaaaaaaaaaaaaaaa push rax ; (Push from RIP) push WORD PTR [rip+0x0] push QWORD PTR [rip+0x0] pop rax pop bx</pre>	<Register id="A" size="8">2918B848905B6658</Register> <Register id="B" size="8">35ff</Register> <Memory offset_register="SP" offset="F6">58665B9048B81829FF35</Memory>

```
<CheckpointResult Tag="Pointer w/qw">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0x2918B848905B6658</Expected>
    <Found>$RAX=0x2918B848905B6658</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x35FF</Expected>
    <Found>$RBX=0x35FF</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP-0xA]={58, 66, 5B, 90, 48, B8, 18, 29, FF, 35}</Expected>
    <Found>[RSP-0xA]={58, 66, 5B, 90, 48, B8, 18, 29, FF, 35}</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

2	Push/pop register word and qword	Push the data onto the stack from a word/qword register using the opcode variant that performs this task.	<pre>movabs rax,0x4f4e4d4c3b3a2918 push ax push rax pop rbx pop cx</pre>	<Register id="B" size="8">4f4e4d4c3b3a2918</Register> <Register id="C" size="8">2918</Register>
---	----------------------------------	---	--	---

```
<CheckpointResult Tag="Reg w/qw">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x4F4E4D4C3B3A2918</Expected>
    <Found>$RBX=0x4F4E4D4C3B3A2918</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RCX=0x2918</Expected>
    <Found>$RCX=0x2918</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

3	Push/pop immediate byte, word, and dword	Push the data onto the stack from a word/qword immediate using the opcode variant that performs this task.	<pre>push 0xfffffffffffff2 pushw 0x1234 push 0x12345678 pop rax pop bx pop rcx pop rdx</pre>	<Register id="A" size="8">12345678</Register> <Register id="B" size="2">1234</Register> <Register id="C" size="8">fffffffffffff2</Register> <Register id="D" size="8">aaaaaaaaaaaaaaaaaa</Register> <Memory offset_register="SP" offset="F8">aaaaaaaaaaaaaaaaaa</Memory> <Register id="SP" size="8">800000</Register>
---	--	--	--	---

```

<CheckpointResult Tag="Imm b/w/dw">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0x12345678</Expected>
    <Found>$RAX=0x12345678</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$BX=0x1234</Expected>
    <Found>$BX=0x1234</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RCX=0xFFFFFFFFFFFFFF2</Expected>
    <Found>$RCX=0xFFFFFFFFFFFFFF2</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RDX=0xAAAAAAAAAAAAAAA</Expected>
    <Found>$RDX=0xAAAAAAAAAAAAAAA</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>$RSP=0x800000</Expected>
    <Found>$RSP=0x800000</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>[RSP-0x8]={AA, AA, AA, AA, AA, AA, AA, AA}</Expected>
    <Found>[RSP-0x8]={AA, AA, AA, AA, AA, AA, AA, AA}</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Ret and call

Test #	Aim	How	Instructions used as input	Expected result
1	Test whether call and ret jump to and from the called function correctly	A small function was created to use in the test data that is called at the beginning of the program. If the functions returns as expected(0xA0 in the A register), the testcase passed.	<pre> mov rbx,rsp add rbx,0xffff mov rax,0x80 push rax call 0x1b cmp rbx,rsp nop pop rbp pop rax add rax,0x20 push rbp ret 0xffff </pre>	<Register id="A" size="8">A0</Register> <Flag name="Zero">1</Flag> <Flag name="Parity">1</Flag>

```

<CheckpointResult Tag="Win">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0xA0</Expected>
    <Found>$RAX=0xA0</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>ZF=1</Expected>
    <Found>ZF=1</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

Rotate

Test #	Aim	How	Instructions used as input	Expected result
1	Test left rotation of the A register, cl times	The variant of the ROL opcode that implicitly uses the CL register as the rotation count will be used in the test data..	<pre>; Setup for testcases 1-4 movabs rax,0x444444443332211 mov ebx,eax mov WORD PTR [rsp],ax mov dh,ah ; mov cl,0x44 rol rax,cl</pre>	<Register id="A" size="8">44444443332211</Register> <Flag name="Carry">0</Flag>
			<pre><CheckpointResult Tag="rol rm, cl"> <SubCheckpointResult result="Passed"> <Expected>\$RAX=0x444444433322114</Expected> <Found>\$RAX=0x444444433322114</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected></Expected> <Found></Found> </SubCheckpointResult> </CheckpointResult></pre>	
2	Test left rotation of a dword, when the number of rotations is stored in an immediate	The variant of the ROL opcode that rotates a register by an immediate will be used in the test data.	rol ebx,0x10	<Register id="B" size="8">22113333</Register> <Flag name="Carry">1</Flag>
			<pre><CheckpointResult Tag="rol dw rm, imm"> <SubCheckpointResult result="Passed"> <Expected>\$RBX=0x22113333</Expected> <Found>\$RBX=0x22113333</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected>CF</Expected> <Found>CF</Found> </SubCheckpointResult> </CheckpointResult></pre>	
3	Test left rotation of value in memory pointed to by address	The variant of the ROL opcode that uses a pointer in memory and rotates by 1 will be used in the test data.	rol WORD PTR [rsp],1	<Memory offset_register="SP">2244</Memory> <Flag name="Overflow">0</Flag> <Flag name="Carry">0</Flag>
			<pre><CheckpointResult Tag="rol rm, 1"> <SubCheckpointResult result="Passed"> <Expected>[RSP]={22, 44}</Expected> <Found>[RSP]={22, 44}</Found> </SubCheckpointResult> <SubCheckpointResult result="Passed"> <Expected></Expected> <Found></Found> </SubCheckpointResult> </CheckpointResult></pre>	

4	Test left rotation of a full rotation loop(result same as input)	The ROL opcode will be used such that the entire register is rotated back to its original value.	rol dh,0x4	<Register id="D" size="2">2200</Register> <Flag name="Carry">0</Flag>
---	--	--	------------	--

```
<CheckpointResult Tag="rol full circle upper b rm, cl">
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0x2200</Expected>
    <Found>$DX=0x2200</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>
```

5	Test left rotation of register using carry in rotation pool	The variant of the RCL opcode that implicitly uses the CL register as the rotation count will be used in the test data..	; Setup for tests 5-8 stc movabs rax,0x4444444433332211 mov ebx, eax mov WORD PTR [rsp], ax mov dh, ah ; mov cl,0x44 rcl rax, cl	<Register id="A" size="8">444444433332211a</Register>
---	---	--	--	---

```
<CheckpointResult Tag="rcl rm, cl">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0x444444433332211A</Expected>
    <Found>$RAX=0x444444433332211A</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

6	Test left rotation of a dword, when the number of rotations is stored in an immediate, with the carry flag in the pool	The variant of the RCL opcode that rotates a register by an immediate will be used in the test data.	rcl ebx,0x10	<Register id="B" size="8">22111999</Register> <Flag name="Carry">1</Flag>
---	--	--	--------------	--

```
<CheckpointResult Tag="rcl dw rm, imm">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x22111999</Expected>
    <Found>$RBX=0x22111999</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CF</Expected>
    <Found>CF</Found>
  </SubCheckpointResult>
</CheckpointResult>
```

7	Test left rotation of value in memory pointed to by address, with the carry flag in the pool.	The variant of the RCL opcode that uses a pointer in memory and rotates by 1 will be used in the test data. The CF is set by the previous test.	rcl WORD PTR [rsp],1	<Memory offset_register="SP">2344</Memory> <Flag name="Overflow">0</Flag> <Flag name="Carry">0</Flag>
---	---	---	----------------------	---

```

<CheckpointResult Tag="rcl rm, 1(CF SET BY PREV)">
  <SubCheckpointResult result="Passed">
    <Expected>[RSP]={23, 44}</Expected>
    <Found>[RSP]={23, 44}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>

```

8	Test left rotation of value stored in upper register with CF in pool	The RCL opcode will be used such that the entire register is rotated back to its original value.	rcl dh,0x4	<Register id="D" size="2">2100</Register> <Flag name="Carry">0</Flag>
---	--	--	------------	--

```

<CheckpointResult Tag="rcl full circle upper b rm, cl">
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0x2100</Expected>
    <Found>$DX=0x2100</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>

```

9	Test right rotation of the A register, cl times	The variant of the ROR opcode that implicitly uses the CL register as the rotation count will be used in the test data..	movabs rax,0x444444443332211 mov ebx,eax mov WORD PTR [rsp],ax mov dh,ah mov cl,0x44 ror rax,cl	<Register id="A" size="8">14444444333221</Register> <Flag name="Carry">0</Flag>
---	---	--	--	--

```

<CheckpointResult Tag="ror rm, cl">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0x14444444333221</Expected>
    <Found>$RAX=0x14444444333221</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>

```

10	Test right rotation of a dword, when the number of rotations is stored in an immediate	The variant of the ROR opcode that rotates a register by an immediate will be used in the test data.	ror ebx,0x10	<Register id="B" size="8">22113333</Register>
----	--	--	--------------	---

```

<CheckpointResult Tag="ror dw rm, imm">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x22113333</Expected>
    <Found>$RBX=0x22113333</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

11	Test right rotation of value in memory pointed to by address	The variant of the ROR opcode that uses a pointer in memory and rotates by 1 will be used in the test data.	ror WORD PTR [rsp],1	<Memory offset_register="SP">0891</Memory> <Flag name="Overflow">0</Flag> <Flag name="Carry">1</Flag>
----	--	---	----------------------	---

```

<CheckpointResult Tag="ror rm, 1">
  <SubCheckpointResult result="Passed">
    <Expected>[RSP]={8, 91}</Expected>
    <Found>[RSP]={8, 91}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CF</Expected>
    <Found>CF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

12	Test right rotation of a full rotation loop(result same as input)	The ROR opcode will be used such that the entire register is rotated back to its original value.	ror dh,0x4	<Register id="D" size="2">2200</Register> <Flag name="Carry">0</Flag>
----	---	--	------------	---

```

<CheckpointResult Tag="ror full circle upper b rm, cl">
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0x2200</Expected>
    <Found>$DX=0x2200</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>

```

13	Test right rotation of register using carry in rotation pool	The variant of the RCR opcode that implicitly uses the CL register as the rotation count will be used in the test data..	stc movabs rax,0x4444444433332211 mov ebx,eax mov WORD PTR [rsp],ax mov dh,ah mov cl,0x44 rcr rax,cl	<Register id="A" size="8">344444443333221</Register>
----	--	--	--	--

```

<CheckpointResult Tag="rcr rm, cl">
  <SubCheckpointResult result="Passed">
    <Expected>$RAX=0x344444443333221</Expected>
    <Found>$RAX=0x344444443333221</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

14	Test right rotation of a dword, when the number of rotations is stored in an immediate, with the carry flag in the pool	The variant of the RCR opcode that rotates a register by an immediate will be used in the test data.	rcr ebx,0x10	<Register id="B" size="8">44223333</Register>
----	---	--	--------------	---

```

<CheckpointResult Tag="rcr dw rm, imm">
  <SubCheckpointResult result="Passed">
    <Expected>$RBX=0x44223333</Expected>
    <Found>$RBX=0x44223333</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

15	Test right rotation of value in memory pointed to by address, with the carry flag in the pool.	The variant of the RCR opcode that uses a pointer in memory and rotates by 1 will be used in the test data. The CF is set by the previous test.	rcr WORD PTR [rsp],1	<Memory offset_register="SP">0811</Memory> <Flag name="Overflow">0</Flag> <Flag name="Carry">1</Flag>
----	--	---	----------------------	---

```

<CheckpointResult Tag="rcr rm, 1">
  <SubCheckpointResult result="Passed">
    <Expected>[RSP]={8, 11}</Expected>
    <Found>[RSP]={8, 11}</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected>CF</Expected>
    <Found>CF</Found>
  </SubCheckpointResult>
</CheckpointResult>

```

16	Test right rotation of value stored in upper register with CF in pool	The RCR opcode will be used such that the entire register is rotated back to its original value.	rcr dh,0x4	<Register id="D" size="2">5200</Register> <Flag name="Carry">0</Flag>
----	---	--	------------	---

```

<CheckpointResult Tag="rcr upper b rm, cl(CF SET BY PREV)">
  <SubCheckpointResult result="Passed">
    <Expected>$DX=0x5200</Expected>
    <Found>$DX=0x5200</Found>
  </SubCheckpointResult>
  <SubCheckpointResult result="Passed">
    <Expected></Expected>
    <Found></Found>
  </SubCheckpointResult>
</CheckpointResult>

```

SetCC

Test #	Aim	How	Instructions used as input	Expected result
1	Test whether SetCC sets the specific byte based on a condition successfully	The SetCC operand will be used in the input instructions. This only has to be tested with a couple of condition codes because the condition testing methods will be tested separately.	<pre>mov bl,0x10 cmp bl,0x10 sete BYTE PTR [rsp] mov bl,0xff cmp bl,0x10 setge BYTE PTR [rsp+0x1]</pre>	<Memory offset_register="SP">0100</Memory>

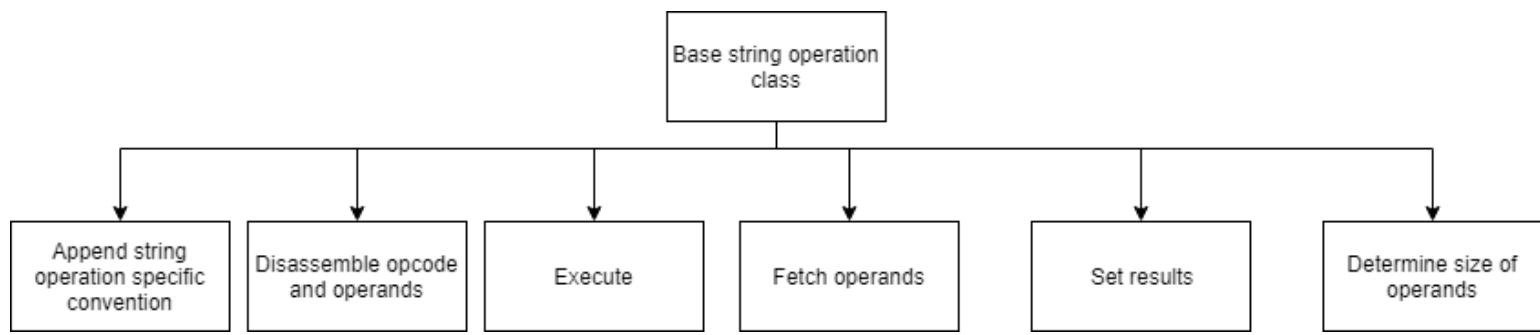
```
<TestcaseResult name="setbyte" result="Passed">
  <CheckpointResult Tag="Win">
    <SubCheckpointResult result="Passed">
      <Expected>[RSP]={1, 0}</Expected>
      <Found>[RSP]={1, 0}</Found>
    </SubCheckpointResult>
  </CheckpointResult>
</TestcaseResult>
```

Xchg

Test #	Aim	How	Instructions used as input	Expected result
1	Test that both register values are swapped when using the XCHG opcode	The instructions containing the XCHG opcode will be the input. They will swap around the registers a few times, then the outcome will depend on whether the results match the expected results.	<pre>movabs rax,0x4444444433332211 mov bl,0x44 mov ch,0x44 mov DWORD PTR [rsp],eax xchg bx,ax xchg ch,ah xchg rdx,rax xchg DWORD PTR [rsp],eax</pre>	<Register id="A" size="8">33332211</Register> <Register id="B" size="2">2211</Register> <Register id="C" size="2">0</Register> <Register id="D" size="8">4444444433334444</Register> <Memory offset_register="SP">0</Memory>

```
<TestcaseResult name="xchg" result="Passed">
  <CheckpointResult Tag="Win">
    <SubCheckpointResult result="Passed">
      <Expected>$RAX=0x33332211</Expected>
      <Found>$RAX=0x33332211</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected>$BX=0x2211</Expected>
      <Found>$BX=0x2211</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected>$CX=0x0</Expected>
      <Found>$CX=0x0</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected>$RDX=0x444444433334444</Expected>
      <Found>$RDX=0x444444433334444</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected>[RSP]={0}</Expected>
      <Found>[RSP]={0}</Found>
    </SubCheckpointResult>
  </CheckpointResult>
</TestcaseResult>
```

String operation base class



Structure explanation

The string operation class implements the IMyOpcode interface and has a similar structure to the base opcode class. It provides the same functionality except adapted to the specific requirements of a string operation instruction.

Structure justification

The best approach to implementing the solution to this problem is to separate this class from the opcode base class. This is because the single responsibility principle to be met because instead of having the opcode class become a superclass, it can be broken down into two separate classes which will allow for a more modular solution because each class will perform a specific task.

Determine size of operands explanation & applying convention

```
// Determine capacity as well as an informal mnemonic convention. I haven't seen it defined anywhere, but
// in many programs such as GDB and GCC, this is used/accepted.
// BYTE => append "B"
// WORD => append "W"
// DWORD => append "D"
// QWORD => append "Q"
// E.g,
// SCAS RAX, QWORD PTR [RSI] => SCASQ
// CMPS BYTE PTR [RDI], BYTE PTR[RSI] => CMPSB
// However in my program(as with others), the operands will remain but
// without BYTE PTR or equivalent. This does slightly contradict the purpose of
// this convention, but I don't expect everyone using the program to know the
// operands of every string operation off by heart(as each has a constant set of operands).
if ((Settings | StringOpSettings.BYTEMODE) == Settings)
{
    Capacity = RegisterCapacity.BYTE;
    mnemonic += 'B';
}
else if ((ControlUnit.RexByte | REX.W) == ControlUnit.RexByte)
{
    Capacity = RegisterCapacity.QWORD;
    mnemonic += 'Q';
}
else if (ControlUnit.LPrefixBuffer.Contains(PrefixByte.SIZEOVR))
{
    Capacity = RegisterCapacity.WORD;
    mnemonic += 'W';
}
else
{
    Capacity = RegisterCapacity.DWORD;
    mnemonic += 'D';
}
```

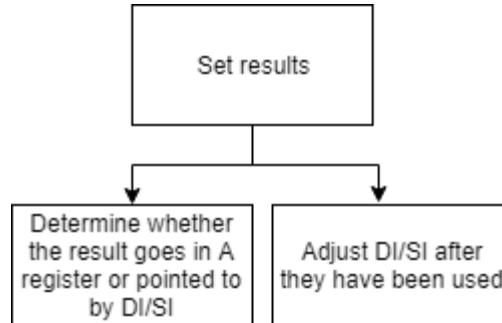
The solution to this problem has been implemented using conditional decision statements. This is because the indicators that determine the size of opcode can be detected by the value of certain variables. The required mnemonic suffix is then appended onto the end of the mnemonic specified by the opcode class.

Determine size of operands & applying convention justification

This is the best approach to implementing the solution to this problem because it will reduce the amount of code that is repeated throughout the opcode subclasses. This is because all string operations use these rules to determine their operand capacity and mnemonic prefixes. It would still be possible for modules/subclasses introduced in the future to call their own procedure to determine the capacity by a different rules if they needed to. This also allows the

open/closed principle design success criterion to be met because the derived classes will still be able to extend the functionality of the string operation base class(to perform the function of the opcode) without being limited by this method, however, if they could still reuse it if needed to reduce their code size and testing requirements.

Set results explanation



```
public void Set(byte[] data)
{
    // There are flaws in the caller if this is not already the same size.
    data = Bitwise.Cut(data, (int)Capacity);

    // If the destination handle is to DI/SI, it would be a pointer not the actual value
    if (Destination.Code == XRegCode.DI || Destination.Code == XRegCode.SI)
    {
        ControlUnit.SetMemory(DestPtr, data);
    }

    // Otherwise set the value of the register
    else
    {
        Destination.Set(data);
    }
}
```

Selection is used to determine whether the result is stored in the A register or a pointer. This has been implemented using a more general approach, interpret it as a pointer if and only if it is SI/DI. The destination(an IDecoded object) then handles the set procedure itself, otherwise, the DestPtr variable(which holds the ulong pointer stored in DI) is used to set the memory location.

```
public void AdjustDI()
{
    // The direction flag in the control unit dictates whether string operations work forwards or backwards, i.e is the string
    // in memory stored like "MYSTRING" or "GNIRTSYM". In the first case it will start at the beginning M and work towards G,
    // otherwise it will start at the end M and work towards G at the beginning.
    // Only create string operand opcodes that use at least one of: SI, DI, or one of those replaced with the A register.
    // Behaviour without complying to this convention is undefined.
    if (ControlUnit.Flags.Direction == FlagState.ON)
    {
        DestPtr -= (byte)Capacity;
    }
    else
    {
        DestPtr += (byte)Capacity;
    }
}

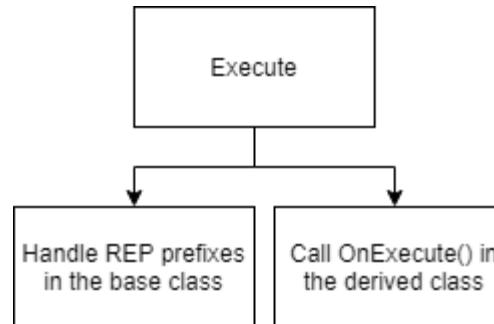
public void AdjustSI()
{
    // See DI
    if (ControlUnit.Flags.Direction == FlagState.ON)
    {
        SrcPtr -= (byte)Capacity;
    }
    else
    {
        SrcPtr += (byte)Capacity;
    }
}
```

This problem has been addressed by using two separate functions for adjusting SI and DI. This is because there are different scenarios where each would be set. For example, in an operation only using SI, DI would be unchanged. The FlagSet in the CU is used to determine whether to add or subtract from the pointer.

Set results justification

This is the best approach to implementing the solution to this problem because it gives the subclasses more control over their operands rather than to restrict. For example, if a module in the future needed the opcode to use the B register instead of the A register, this would be possible as the converse is used to determine whether the input is a pointer or a register. This links to the open/closed principle design success criterion because said modules will be able to extend the functionality of this base class without having to directly modify the code of the base class to work in the future. This is because they can use the AdjustDI and AdjustSI methods in the derived classes(because they are public), hence can use them for their own purposes in the future.

Execute explanation



```
public void Execute()
{
    // Handle a REP prefix. If the handle has NOJMP set, this is ignored.
    // The REP prefix must be understood when dealing with string operations.
    // REP simply means "repeat this instruction until ECX=0, and decrement ECX every time"
    // Some pseudocode could be,
    // while($ECX > 0)
    // {
    //     Execute();
    //     $ECX--;
    // }
    // $ECX will stop at zero, i.e never become -1.
    // CMPS and SCAS handle REP prefixes slightly differently, such that there are two forms of the prefix,
    // each with an extra condition. The two forms are REPZ and REPNZ. Both of these still follow the above,
    // but have a different additional condition. REPZ will repeat whilst the zero flag is set. This is checked
    // after the instruction is ran, the flag does not have to be set beforehand.
    // Pseudo,
    // while($ECX > 0)
    // {
    //     Execute();
    //     $ECX--;
    //     if(ZF == 0)
    //     {
    //         break;
    //     }
    // }
    // The second, REPNZ will do the opposite, and exit afterwards if the zero flag is not set. This is due to the nature
    // of the SCAS and CMPS opcodes. SCAS compares a string of bytes at *SI with $A. The derivation of length of these bytes
    // can be found in the constructor. CMPS compares a string of bytes at *SI with *DI. If two strings are being compared,
    // and one character is found to be different, it is immediately apparent that the two strings are not equal, so the whole
    // string does not have to be iterated, only until the different. If they are the same, afterwards I would recommend using
    // JZ or JE to avoid having to use two lines,(TEST $ECX, $ECX; JZ) because this information is already provided by the result
    // of the last comparison. This is concept is generalised by setting the COMPARE bit of the StringOpSettings for an opcode.
    // Derived classes must have a different design to Opcode derived classes. There must be no non-constant information assigned
    // in the constructor. This must be done in the derived OnInitialise(), and execution in the OnExecute(). This is due to REP
    // prefixes, as information must be updated with each execution, such that each execution is treat as a new instruction, but
    // without having to re-instance the class. Execute() in the base class will make sure that rep prefixes are compatible, calling
    // the derived OnInitialise(), then OnExecute() when ready.
```

```

// Check if prefix is set in buffer
if (ControlUnit.LPrefixBuffer.Contains(PrefixByte.REPZ)
    || ControlUnit.LPrefixBuffer.Contains(PrefixByte.REPNZ))
{
}

// Initialise $Count to $ECX.
uint Count = BitConverter.ToUInt32(ECX_Counter.FetchOnce(), 0);
for (; Count > 0; Count--)
{
    OnExecute();
    OnInitialise();

    // If the operation is a comparison, extra checks have to be done against the zero flag.
    // This will be ignored when not a comparison.
    // The conditions can be summarised as
    // If (REPZ && ZF != 1) break;
    // If (REPNZ && ZF != 0) break;
    if ((Settings | StringOpSettings.COMPARE) == Settings
        // repz and repnz act as normal rep if the opcode isn't cmps or scas=
        && ((ControlUnit.LPrefixBuffer.Contains(PrefixByte.REPZ) && ControlUnit.Flags.Zero != FlagState.ON)
            || (ControlUnit.LPrefixBuffer.Contains(PrefixByte.REPNZ) && ControlUnit.Flags.Zero != FlagState.OFF)))
    {
        break;
    }
}

// Set $ECX to the new count. Once the instruction is completely executed(all REPs handled), $ECX will either be
// zero, or $ECX_Before_Instruction - $Number_Of_OnExecute()s
ECX_Counter.Set(BitConverter.GetBytes(Count));
}

// If no REP prefix, the instruction can be executed as normal.
else
{
    OnExecute();
}

```

The two subproblems have been solved in the same procedure. First a REP prefix is found by checking the LPrefixBuffer. If there isn't one, execution will move into the else block(on the last picture) and execute the instruction normally. The ECX register is fetched into the Count variable, which will be used as the counter for iteration(per intel requirements). The loop repeats until the counter is zero (inclusive)or for certain opcodes, execution will continue until stop depending on whether the zero flag is set. In the loop, Execute() and Initialise() are called. This allows the function to meet any required preconditions before execute is called(for this first iteration it was already called before the loop).

Execute justification

This is the best approach to implementing the solution to this problem because it has allowed the single responsibility success criterion to be met. This is because all future string operations will be able to make use of REP prefix functionality and will not have to individually code methods for handling the prefix. This means that as the responsibility of handling REP prefixes has been delegated to the base class only, less time will be required testing future modules such as new string operations, where the procedure would otherwise have to be repeated.

Fetch explanation

```
// Store the value of the input register if present, otherwise the SI register. See constructor
private readonly byte[] ValueOperand;

public List<byte[]> Fetch()
{
    // There will be two values output.
    List<byte[]> Output = new List<byte[]>(2);

    // If the destination is DI, use it as an effective address.
    if (Destination.Code == XRegCode.DI)
    {
        Output.Add(ControlUnit.Fetch(DestPtr, (int)Capacity));
    }

    // Otherwise add its value.
    else
    {
        Output.Add(ValueOperand);
    }

    // IF the source is SI, use its as an address
    if (Source.Code == XRegCode.SI)
    {
        Output.Add(ControlUnit.Fetch(SrcPtr, (int)Capacity));
    }

    // Otherwise add its value.
    else
    {
        Output.Add(ValueOperand);
    }
    return Output;
}
```

The fetch problem has been solved by using selection. In a string operation, if the DI register is the destination, it is always a pointer. If the SI register is the source, it is also always a pointer, which can be found using predicates in the if statements.

Fetch justification

This is the best approach to solving this problem because it allows the IMyDecoded and opcode API methods to be reused to simplify the procedure. For example, Destination.Code has been used as a simple measure to determine which register the operand represents, such that extra procedures to determine this were not required. This links to the single responsibility principle design success criterion because it has removed the need for the single purpose of “fetching operands” to be spread across multiple operands by making use of existing functions. This will reduce the development time required to maintain this module because the functions it uses will not have to be updated multiple times. For example, if an update to the CU.Fetch() method is required, it will not affect this module, however, if it used its own means of accessing a register, it would be necessary to update that procedure to do so.

Feature demo and review explanation

A demonstration of features was designed to show the current state of the program to the stakeholders. This allowed stakeholders to give feedback on the current state of the program and review the current features. An assembly program that showed a lot of the current assembly features was used. It was a basic program that set counted up to 255(0xFF), then back down again, storing the results progressively in memory. The assembly program was first executed in the host OS so the users could see the expected results and gave insight to how the program was tested.

Disassembly		Registers	
0x000000000000001F	TEST CL, CL	RIP	: 0x0000000000000093
0x0000000000000021	JNZ 0x15	RAX	: 0x0000000064636261
0x0000000000000023	CMP DL, 0xFF	RCX	: 0x0000000000000000
0x0000000000000026	JZ 0x30	RDX	: 0x00000000000000FF
0x0000000000000028	MOV DL, 0xFF	RBX	: 0x0000000000000000
0x000000000000002A	MOV BL, 0x0	RSP	: 0x0000000000800000
0x000000000000002C	MOV CL, DL	RBP	: 0x0000000000800000
0x000000000000002E	JMP 0x15	RSI	: 0x0000000000800365
0x0000000000000030	MOV RSI, RSP	RDI	: 0x0000000000800369
0x0000000000000033	MOV RDI, RSI	R8	: 0x0000000000000000
0x0000000000000036	ADD RDI, 0x0200	R9	: 0x0000000000000000
0x000000000000003D	MOV ECX, 0x0100	R10	: 0x0000000000000000
0x0000000000000042	REP MOVSB [RDI], [RSI]	R11	: 0x0000000000000000
0x0000000000000044	STD	R12	: 0x0000000000000000
0x0000000000000045	SUB RSI, 0x04	R13	: 0x0000000000000000
0x0000000000000049	ADD RDI, 0xFC	R14	: 0x0000000000000000
0x0000000000000050	MOV ECX, 0x40		
0x0000000000000055	REP MOVSD [RDI], [RSI]		
0x0000000000000057	CLD		
0x0000000000000058	MOV RSI, RSP		
0x000000000000005B	ADD RSI, 0x022F		
0x0000000000000062	MOV RDI, RSI		
0x0000000000000065	ADD RDI, 0x0100		

The screenshot shows a debugger interface with two main panes. The left pane, titled 'Memory', displays a hex dump of memory starting at address 0x0000000000000090. The right pane, titled 'Flags', shows the current state of CPU flags.

Address	Value
0x0000000000000090	04 AF 90 00 00 00 00 00 00 00 00 00 00 00 00 00
0x0000000000800000	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x0000000000800010	10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
0x0000000000800020	20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
0x0000000000800030	30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
0x0000000000800040	40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
0x0000000000800050	50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
0x0000000000800060	60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
0x0000000000800070	70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
0x0000000000800080	80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
0x0000000000800090	90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
0x00000000008000A0	A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
0x00000000008000B0	B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
0x00000000008000C0	C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
0x00000000008000D0	D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
0x00000000008000E0	E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
0x00000000008000F0	F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
0x0000000000800100	FF FE FD FC FB FA F9 F8 F7 F6 F5 F4 F3 F2 F1 F0
0x0000000000800110	EF EE ED EC EB EA E9 E8 E7 E6 E5 E4 E3 E2 E1 E0

Flags

Flag	Value
Carry	: False
Parity	: True
Overflow	: False
Auxiliary	: False
Direction	: False
Zero	: True

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000000000800130	CF	CE	CD	CC	CB	CA	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0
0x0000000000800140	BF	BE	BD	BC	BB	BA	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
0x0000000000800150	AF	AE	AD	AC	AB	AA	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0x0000000000800160	9F	9E	9D	9C	9B	9A	99	98	97	96	95	94	93	92	91	90
0x0000000000800170	8F	8E	8D	8C	8B	8A	89	88	87	86	85	84	83	82	81	80
0x0000000000800180	7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70
0x0000000000800190	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	60
0x00000000008001A0	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50
0x00000000008001B0	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40
0x00000000008001C0	3F	3E	3D	3C	3B	3A	39	38	37	36	35	34	33	32	31	30
0x00000000008001D0	2F	2E	2D	2C	2B	2A	29	28	27	26	25	24	23	22	21	20
0x00000000008001E0	1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10
0x00000000008001F0	0F	0E	0D	0C	0B	0A	09	08	07	06	05	04	03	02	01	00
0x0000000000800200	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0x0000000000800210	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0x0000000000800220	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
0x0000000000800230	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
0x0000000000800240	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
0x0000000000800250	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F

Reviewer	Comment
Stakeholder - C# Programmer	“Progress is coming on very well, I’m glad to see the amount of and complexity of instructions the emulator can handle. This has definitely seen progress from the preview version I reviewed. The user interface is also

	coming along well, the theme looks very developed and comparable to a professionally-made application" <i>(This stakeholder did not review the previous version, so had not seen the interface updates in the last version)</i>
Stakeholder - (Different) C# Programmer	"This version of the software looks very production ready. There have been lots of new opcodes added, which will make the program more useful for testing programs. I think the next step is to add more options for inputting programs into the solution, as it required putting the machine code into an array to launch the program"
Stakeholder - Existing assembly programmer	"This version is very impressive. The demonstration shows that most of the opcodes I use have been implemented, and from what I have seen already the others are included but not in this demonstration. I agree with *redacted*, it did seem a hassle to input the program and there was no validation applied, so I think the next version should include a formal IO system, but I'm sure this is planned already"
Stakeholder - Python programmer	"The improvement of the feature set here is great. I would definitely be encouraged to learn in this environment, I'm glad that there are a lot of features present that would normally only be found on linux, being able to step through the code. You are definitely meeting the 1:1 emulation criterion you mentioned in the demonstration as the results were clearly the same as the results shown when executed outside of VMCS. I like the modularity of the IMyOpcode interface and the opcode base classes, they would definitely be useful if I were to create a module as I can use the provided functions."

Username and password login box

```
public static class Auth
{
    private static string StoredUsername;
    private static string StoredPassword;
    public static void InputLogin()
    {
        // Prompt asking user to input a username and password
        string Username = Microsoft.VisualBasic.Interaction.InputBox("Choose username", "VMCS", "");
        string Password = Microsoft.VisualBasic.Interaction.InputBox("Choose password", "VMCS", "");
        if(!PresenceCheck(Username, Password))
        {
            MessageBox.Show("Please enter a username and password");
            InputLogin();
        }
        else if (!TypeCheck(Username, Password))
        {
            MessageBox.Show("Usernames and passwords can only contain letters");
            InputLogin();
        }
        else if (!LengthCheck(Username, Password))
        {
            MessageBox.Show("Usernames and passwords must be less than 8 characters");
            InputLogin();
        }
        else
        {
            MessageBox.Show("Success");
            // Store details to be checked later
            StoredUsername = Username;
            StoredPassword = Password;
        }
    }

    public static void Lock()
    {
        // Prompt for username and password
        string Username = Microsoft.VisualBasic.Interaction.InputBox("Enter username", "VMCS", "");
        string Password = Microsoft.VisualBasic.Interaction.InputBox("Enter password", "VMCS", "");
        // Compare input with stored data
        if(Username != StoredUsername || Password != StoredPassword)
        {
            // Re-lock if the user enters the wrong login
            MessageBox.Show("Incorrect login");
            Lock();
        }
    }

    private static bool LengthCheck(string password, string username)
    {
        // Function returns true if validation passes

        // Check if password or username are too long
        return (password.Length < 8 && username.Length < 8);
    }
}
```

```
private static bool TypeCheck(string password, string username)
{
    // Function returns true if validation passes

    // Iterate through password checking if any characters are not letters
    foreach(char character in password)
    {
        // Return false if any character is not a letter
        if (!char.IsLetter(character))
        {
            return false;
        }
    }

    // Iterate through username checking if any characters are not letters
    foreach (char character in username)
    {
        // Return false if any character is not a letter
        if (!char.IsLetter(character))
        {
            return false;
        }
    }
    return true;
}

private static bool PresenceCheck(string password, string username)
{
    // Function returns true if validation passes

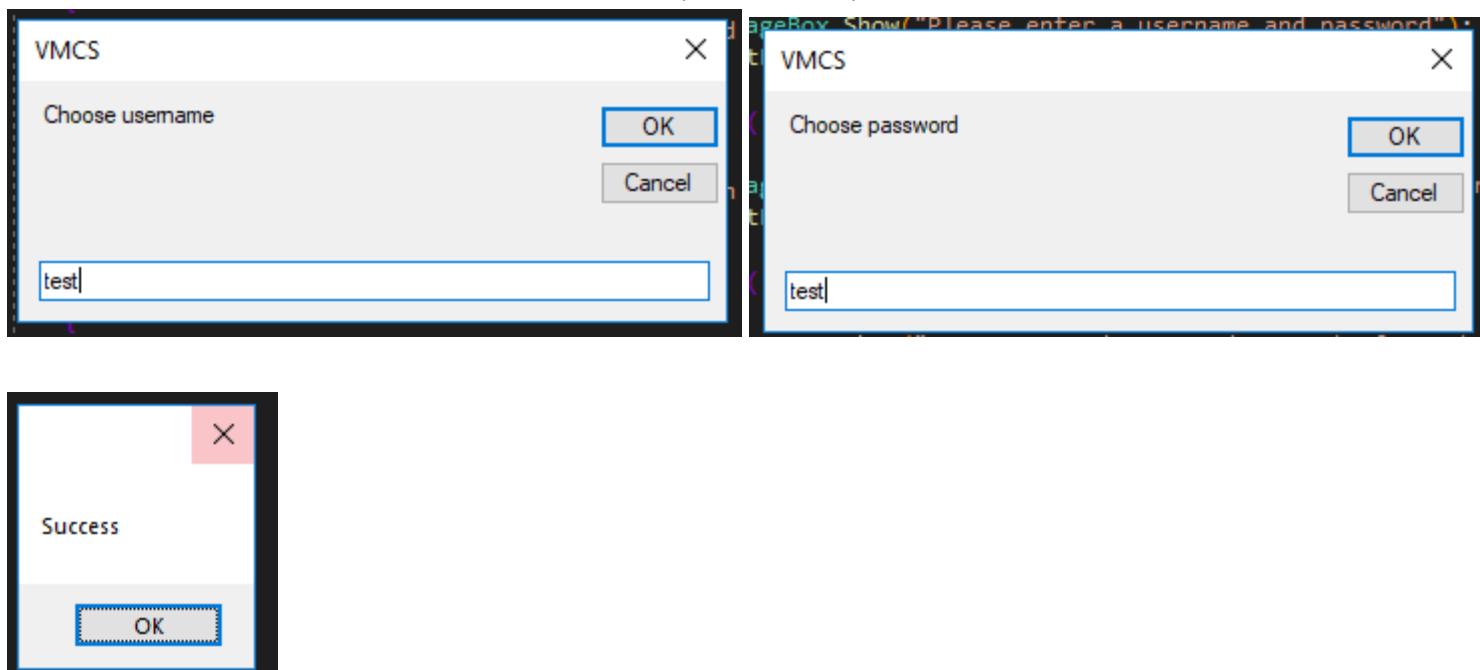
    // Return true if password and username have values
    return (password != "" && username != "");
}
```

Explanation of validation

The username and textbox login box allows the user to enter their set login details, but a sequence of validation routines are required to ensure the credentials meet requirements and are compatible with the rest of the code. A presence check ensures data was entered into each input box. A type check is used to ensure that the username and password contain only letters. The length check ensures the username and password are both less than 8 characters long.

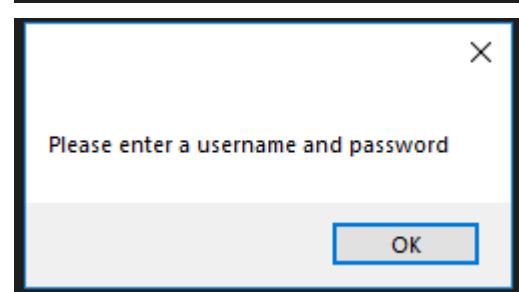
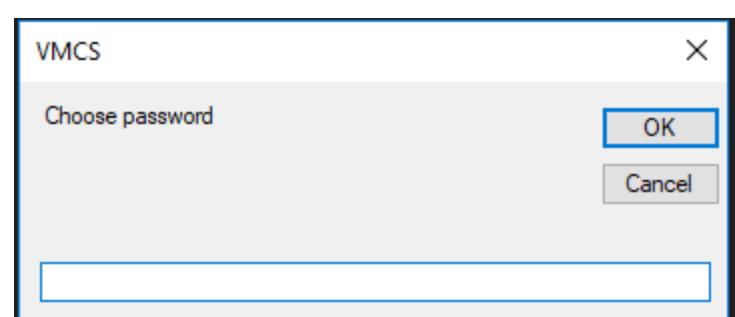
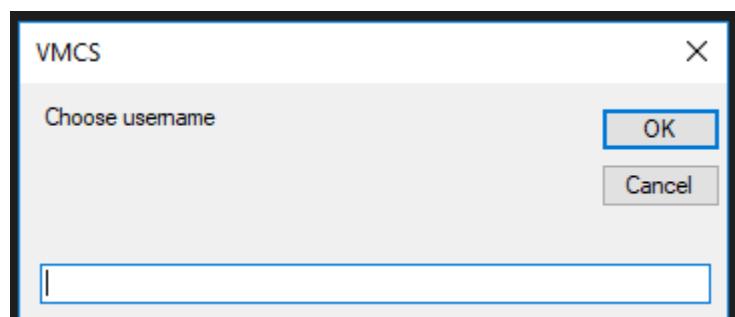
Testing validation

Test 1: Password and username less than 8 letters (Normal case)



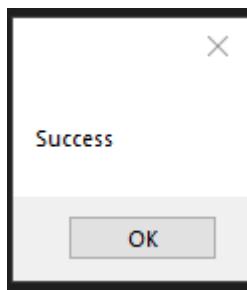
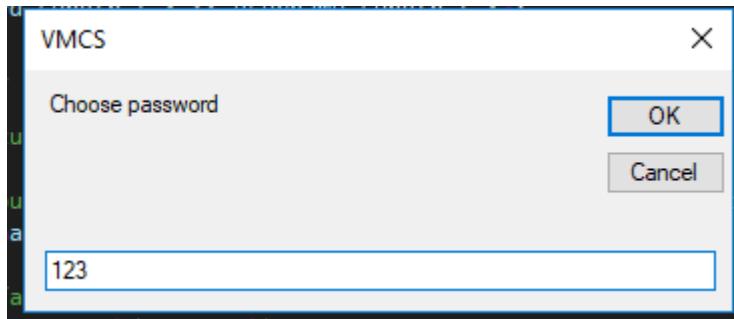
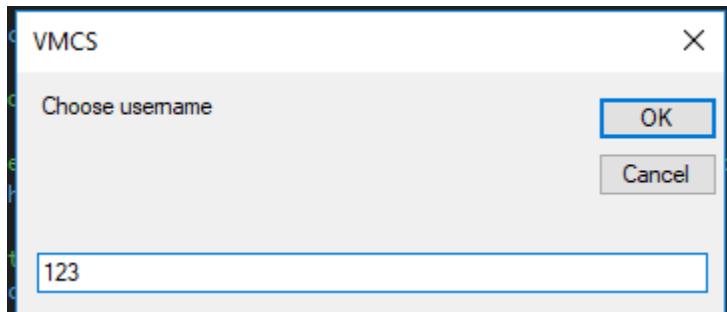
The test produced the expected results so doesn't need any remedial action.

Test 2: Empty username/password



The test produced the expected results so doesn't need any remedial action.

Test 3: Non-alphabetic username and password



This did not produce the correct result so needs remedial action.

```
if(!PresenceCheck(Username, Password))
{
    MessageBox.Show("Please enter a username and password");
    InputLogin();
    if (!TypeCheck(Username, Password))
    {
        MessageBox.Show("Usernames and passwords can only contain letters");
        InputLogin();
    }
}
```

A logical error was found in the code. The TypeCheck was nested in another if statement. This meant that the type check would only be tested if the presence check failed. This was resolved by moving the TypeCheck outside of the if block, where it should have been.

```
string Password = Microsoft.VisualBasic.Interaction.InputBox("Choose password", "VMCS");
if(!PresenceCheck(Username, Password))
{
    MessageBox.Show("Please enter a username and password");
    InputLogin();
    if (!TypeCheck(Username, Password))
    {
        MessageBox.Show("Usernames and passwords can only contain letters");
        InputLogin();
    }
}
else if (!LengthCheck(Username, Password))
{
    MessageBox.Show("Usernames and passwords must be less than 8 characters");
    InputLogin();
}
```

Loading files and IO system

```
/// FileParser provides the basis for loading and parsing external files as instructions.  
// It can be used in a number of different ways,  
// - Automatically detecting the file type through magic byte signatures  
// - Parsing as a text file  
// - Parsing as a binary file  
// The last two are necessary when the first bytes of a txt/bin file are the same as anothers signature by coincidence.  
using debugger.Logging;  
using System.Collections.Generic;  
using System.IO;  
namespace debugger.IO  
{  
    public enum ParseMode  
    {  
        AUTO = 0,  
        BIN = 1,  
        TXT = 2,  
    }  
    public enum ParseResult  
    {  
        SUCCESS = 0,  
        NOT_INFERRRED = 1,  
        INVALID = 2,  
    }  
    public class FileParser  
    {  
        private delegate IMyExecutable MagicDelegate(FileStream inputReader);  
        private static readonly Dictionary<int, MagicDelegate> SignatureTable = new Dictionary<int, MagicDelegate>()  
        {  
            // New magic file signatures for new formats can be added here(The first 4 unique bytes of a file)  
            { 0x7F454C46, ELF.Parse }  
        };  
        private readonly FileInfo TargetFile;  
        public FileParser(FileInfo inputFile)  
        {  
            if (!inputFile.Exists)  
            {  
                throw new LoggedException(LogCode.IO_FILENOFOUND, inputFile.FullName);  
            }  
            TargetFile = inputFile;  
        }  
    }  
}
```

```
public ParseResult Parse(ParseMode mode, out byte[] Instructions)  
{  
    byte[] MagicBytes = new byte[4];  
  
    // Open the file as read only.  
    using (FileStream Reader = TargetFile.Open(FileMode.Open))  
    {  
        if (mode == ParseMode.AUTO)  
        {  
            // There must be no less than 4 bytes in the file in order to determine its magic bytes. This is not necessarily true for  
            // every file type. If there are only 3 bytes in the read file, the user probably chose the wrong file anyway.  
            // Reader.Read() returns the number of bytes read.  
            if (Reader.Read(MagicBytes, 0, 4) != 4)  
            {  
                Logger.Log(LogCode.IO_INVALIDFILE, "File must be no less than 4 bytes in length");  
                Instructions = null;  
                return ParseResult.INVALID;  
            }  
  
            // Order the bytes in big endian(because was read from a file) to form the signature of magic bytes.  
            // Most magic byte signatures are 4 bytes, especially for the purposes of this program.  
            int Signature = MagicBytes[3] + (MagicBytes[2] << 8) + (MagicBytes[1] << 16) + (MagicBytes[0] << 24);  
  
            // Check if the file type can be inferred from the magic byte signature from the signatures registered in the SignatureTable.  
            MagicDelegate ResultDel;  
            if (SignatureTable.TryGetValue(Signature, out ResultDel))  
            {  
                IMyExecutable Result = ResultDel(Reader);  
                // If $Result is null after parsing, there was an error in doing so.  
                if (Result == null)  
                {  
                    Instructions = null;  
                    return ParseResult.INVALID;  
                }  
                else  
                {  
                    Instructions = ResultDel(Reader).Instructions;  
                    return ParseResult.SUCCESS;  
                }  
            }  
        }  
    }  
}
```

```

    }
    else
    {
        // If the file type cannot be inferred,
        Instructions = null;
        return ParseResult.NOT_INFERRRED;
    }
}

// Other parse modes for forcing files to be interpreted as a particular format.
else if (mode == ParseMode.BIN)
{
    Instructions = BIN.Parse(Reader).Instructions;
    return ParseResult.SUCCESS;
}
else if (mode == ParseMode.TXT)
{
    TXT Result = TXT.Parse(Reader);
    // If $Result is null after parsing, there was an error in doing so.
    if (Result == null)
    {
        Instructions = null;
        return ParseResult.INVALID;
    }
    else
    {
        Instructions = Result.Instructions;
        return ParseResult.SUCCESS;
    }
}
}

```

Explanation of FileParser

A standalone class has been created dedicated to handling the opening of files. It uses magic bytes(a set sequence of bytes at the start of every file of a certain type) to identify a file, then creates an instance of a IMyExecutable class(e.g. TXT,BIN,ELF), which handle file type specific parsing and validation procedures.

Justification of FileParser

This was needed so that callers depend on a single module that is unlikely to change, rather than having to adjust many classes to new dependencies when support for a new file type is added.

Validation of files

A length check has to be done on all files. This is because the first 4 bytes are going to be read to fetch the magic bytes. If this was done without checking if the file had at least four bytes, there would be an error.

```

// There must be no less than 4 bytes in the file in order to determine its magic bytes. This is not necessarily true for
// every file type. If there are only 3 bytes in the read file, the user probably chose the wrong file anyway.
// Reader.Read() returns the number of bytes read.
if (Reader.Read(MagicBytes, 0, 4) != 4)
{
    Logger.Log(LogCode.IO_INVALIDFILE, "File must be no less than 4 bytes in length");
    Instructions = null;
    return ParseResult.INVALID;
}

```

There also needs to be a check whether the magic bytes are valid recognised magic bytes. This is done by comparing them with a pre-defined dictionary.

```
// Check if the file type can be inferred from the magic byte signature from the signatures registered in the SignatureTable.
MagicDelegate ResultDel;
if (SignatureTable.TryGetValue(Signature, out ResultDel))
{
    IMyExecutable Result = ResultDel(Reader);
    // If $Result is null after parsing, there was an error in doing so.
    if (Result == null)
    {
        // Set null to prevent errors later
        Instructions = null;
        return ParseResult.INVALID;
    }
    else
    {
        // Fetch instructions
        Instructions = ResultDel(Reader).Instructions;
        return ParseResult.SUCCESS;
    }
}
else
{
    // If the file type cannot be inferred,
    Instructions = null;
    return ParseResult.NOT_INFERRRED;
}
```

BIN class

```
// BIN provides a simple way to load bytes in a file as instructions. Bytes are read as-is, there is no extra translation.
// The use case of this would be when the user's assembler just outputs executable code rather than an object file.
using System.IO;
namespace debugger.IO
{
    public class BIN : IMyExecutable
    {
        public byte[] Instructions { get; private set; }
        public static BIN Parse(FileStream reader)
        {
            reader.Seek(0, SeekOrigin.Begin);
            BIN Output = new BIN()
            {
                Instructions = new byte[reader.Length]
            };
            reader.Read(Output.Instructions, 0, (int)reader.Length);
            return Output;
        }
    }
}
```

The BIN class handles the opening of binary files. These are files that contain raw bytes to be interpreted as instructions. This requires no further validation than what has already been done since any bytes can be read.

TXT class

```

// TXT IMyExecutable allows a file to read that contains UTF-8 encoded text that can be interpreted as hex bytes. Behaviour
// when a different encoding could work, see Core.Htoi(), but is undefined nevertheless.
// The use case of TXT would be when the user has copied text representing bytes from the output of objdump or alike. This
// saves them having to use other programs to convert that output into actual bytes(that BIN would parse).
// If invalid data is found, said byte is ignored.
// This is a rare case where working around invalid input is better than assuming it to be evil because,
// -It is too common that a trailing whitespace will be left or newline when creating a file.
// -Other byte representations can be used, for example C style hex bytes, \x00\xC0\x90
// -The scope of invalid bytes poisoning the valid data is low, as only 0-9 and A-F will be considered valid.
// Exact details of what happens in the case of invalid input can be found in Core.TryParseHex()
using debugger.Logging;
using System.IO;
namespace debugger.IO
{
    public class TXT : IMyExecutable
    {
        public byte[] Instructions { get; private set; }
        public static TXT Parse(FileStream reader)
        {
            // Read all of the file into FileBytes.
            reader.Seek(0, SeekOrigin.Begin);

            // Make sure FileBytes[] has an even length, but not necessarily $reader.Length aswell. Essentially this rounds $FileBytes.Length
            // up to the nearest multiple of two.
            byte[] FileBytes = new byte[reader.Length / 2 + reader.Length % 2];
            reader.Read(FileBytes, 0, (int)reader.Length);
            return Parse(FileBytes);
        }

        public static TXT Parse(byte[] encoded_bytes)
        {
            byte[] ParsedBytes;

            // Try to parse the encoded bytes into their intended byte values. Returns false if none could.
            if (Util.Core.TryParseHex(encoded_bytes, out ParsedBytes))
            {
                return new TXT()
                {
                    Instructions = ParsedBytes
                };
            }
            else
            {
                Logger.Log(LogCode.IO_INVALIDFILE, "Input did not contain any hex-parsable characters.");
                return null;
            }
        }
    }
}

```

The TXT class allows bytes written out in ascii to be converted back into bytes to be used as instructions. For example, “0xAA” written out in text does not have the same value as 0xAA in memory, it is an ascii representation. TXT requires a type check validation. If no bytes can be interpreted as ASCII, then the file must be in an invalid format so has to be rejected.

ELF file

```

/// ELF IMyExecutable provides the ability to load ELF64 and ELF32 files as assembly to debug. ELF has the most practical use case, which is debugging an a linked executable; one that
// could be run from the terminal. Currently, only the .text segment is loaded(and hard coded to do so). This means
// that any linked external libraries will not work(be it statically or dynamically). ELF32 is best-effort; it works absolutely fine in terms of parsing, but the code will
// not be interpreted as x86, rather as x86-64 like an ELF64. This is only a problem for programs that use INC/DEC instructions or the ancient bcd/ascii opcodes.
// There are a couple of tested ways for creating an ELF that loads,
// Create an object file
// nasm -f elf64 file.asm
// Link an existing object file
// ld file.o -o file.out
// OR
// gcc file.o -o file.out -nostdlib -no-pie (-m32 if 32 bit)
// Like this, gcc will call ld behind the scenes. Linking without "-nostdlib" is undefined. It probably will work, but there is no glibc to call, so you will only have erroneous
// behaviour later. If you are insistent on using external libraries, a best effort would be,
// gcc file.o -o file.out -static
// then jump over any calls.
// If you notice instructions that use absolute addressing changing when viewed in the program, use the gcc command with "-no-pie" to prevent them being changed to RIP rel when linked.
using debugger.Logging;
using debugger.Util;
using System;
using System.IO;
namespace debugger.IO
{
    public class ELF : IMyExecutable
    {
        private byte Class;
        public ulong EntryPoint;
        private ulong TextLength;
        private ulong SHOffset;
        private ushort SHLength;
        private ushort SHCount;
        private ushort SHStrIndex;
        public byte[] Instructions { get; private set; }
        public static ELF Parse(FileStream reader)
        {
            ELF ParsedElf;

            // Check if the length is less than the minimum acceptable.
            if (reader.Length < 0x34)
            {
                Logger.Log(LogCode.IO_INVALIDFILE, "Incomplete ELF header");
                return null;
            }
        }
    }
}

```

```

// Determine the class of the elf. If the class byte is equal to 1, the ELF is 32 bit, 2 for 64-bit. Otherwise assume an invalid elf.
// The class byte is the 4th byte(0,1,2,3,<-) so make sure the stream is at position 4. This is required to know the length of the header.
reader.Seek(4, SeekOrigin.Begin);
byte[] Elf_class = new byte[1];
reader.Read(Elf_class, 0, 1);

// Read and parse the rest of the header.
// All bytes are offset slightly because 5 bytes(magic bytes + class) have already been parsed.
// The offsets differ between elf32 and elf64 because addresses in elf64 must be 8 bytes; 4 bytes in elf32.
byte[] FileHeader;
switch (Elf_class[0])
{
    case 0x01:
        FileHeader = new byte[0x34 - 0x05];
        reader.Read(FileHeader, 0, FileHeader.Length);
        ParsedElf = new ELF()
        {
            Class = 32,
            // The address of the section header table.
            SHOffset = BitConverter.ToInt32(FileHeader, 0x20 - 0x05),
            // The length of each header in the section header table.
            SHLength = BitConverter.ToInt16(FileHeader, 0x2E - 0x05),
            // The number of section headers in the section header table.
            SHCount = BitConverter.ToInt16(FileHeader, 0x30 - 0x05),
            // The index of the section header containing the string names of all other sections in the section header table.
            SHStrIndex = BitConverter.ToInt16(FileHeader, 0x32 - 0x05)
        };
        break;
    case 0x02:
        FileHeader = new byte[0x40 - 0x05];
        reader.Read(FileHeader, 0, FileHeader.Length);
        ParsedElf = new ELF()
        {
            Class = 64,
            SHOffset = BitConverter.ToInt64(FileHeader, 0x28 - 0x05),
            SHLength = BitConverter.ToInt16(FileHeader, 0x3A - 0x05),
            SHCount = BitConverter.ToInt16(FileHeader, 0x3C - 0x05),
            SHStrIndex = BitConverter.ToInt16(FileHeader, 0x3E - 0x05)
        };
        break;
    default:
        Logger.Log(LogCode.IO_INVALIDFILE, "Invalid class byte in ELF header");
        return null;
}

```

```

// Make sure ELF uses x86-x64 instruction set
if (FileHeader[0x0] != 0x03 && FileHeader[0x0] != 0x3E)
{
    Logger.Log(LogCode.IO_INVALIDFILE, "Input elf does not use the x86 instruction set");
    return null;
}

// Create a byte array with enough space to store the entire sh table.
byte[] SHTable = new byte[ParsedElf.SHLength * ParsedElf.SHCount];

// Seek the file to the offset of the SH table.
reader.Seek((long)ParsedElf.SHOFFSET, SeekOrigin.Begin);

// Read the entire SH table into $SHTable.
reader.Read(SHTable, 0, ParsedElf.SHLength * ParsedElf.SHCount);

// Use cut and subarray to cut out the shstrtab(section header string table; section that points to text names) from the SH table.
// Subarray cuts out the preceeding tables. The length of which is the number of tables before shstrtab(which happens to be the index of shstrtab) multiplied by
// the length of each table.
byte[] SHstrtab = Bitwise.Cut(Bitwise.Subarray(SHTable, ParsedElf.SHLength * ParsedElf.SHStrIndex), ParsedElf.SHLength);

// Get the size of shstrtab stored at offset 0x20 (elf32:0x14) of the section header and
// seek to the position in the elf file of the actual shstrtab(not the shstrtab section header)
int StrtabSize;
if (ParsedElf.Class == 32)
{
    StrtabSize = BitConverter.ToInt32(SHstrtab, 0x14);
    reader.Seek(BitConverter.ToUInt32(SHstrtab, 0x10), SeekOrigin.Begin);
}
else
{
    StrtabSize = BitConverter.ToInt32(SHstrtab, 0x20);
    reader.Seek(BitConverter.ToUInt32(SHstrtab, 0x18), SeekOrigin.Begin);
}

byte[] strtab = new byte[StrtabSize];

reader.Read(strtab, 0, StrtabSize);
// Read sh_name from every section in shtab and check if the offset($shstrtab + $offset = target) is the string ".text".
// .text is holds the user code and therefore is the only section this program cares about.
// If there isn't one, which most likely means this wasn't an elf file, so handle it because this file cannot be used.
for (int i = 0; i < ParsedElf.SHCount; i++)
{
    if (ReadString(strtab, BitConverter.ToUInt32(SHTable, i * ParsedElf.SHLength)) == ".text")
    {
        if (ParsedElf.Class == 32)
        {
            // Store the EntryPoint(in the file not when in memory) and TextLength(length of the
            // code in bytes) in the elf object.
            ParsedElf.EntryPoint = BitConverter.ToInt32(SHTable, i * ParsedElf.SHLength + 0x10);
            ParsedElf.TextLength = BitConverter.ToInt32(SHTable, i * ParsedElf.SHLength + 0x14);
        }
        else
        {
            ParsedElf.EntryPoint = BitConverter.ToInt64(SHTable, i * ParsedElf.SHLength + 0x18);
            ParsedElf.TextLength = BitConverter.ToInt64(SHTable, i * ParsedElf.SHLength + 0x20);
        }
        break;
    }
    else if (i + 1 == ParsedElf.SHCount)
    {
        Logger.Log(LogCode.IO_INVALIDFILE, "ELF File had no .text section");
        return null;
    }
}

// Finally copy all the bytes from the .text segment into $Instructions.
reader.Seek((long)ParsedElf.EntryPoint, SeekOrigin.Begin);
ParsedElf.Instructions = new byte[(int)ParsedElf.TextLength];
reader.Read(ParsedElf.Instructions, 0, (int)ParsedElf.TextLength);

return ParsedElf;
}

private static string ReadString(byte[] strtab, uint offset)
{
    string Output = "";
    for (uint i = offset; i < strtab.Length; i++)
    {
        // Check if the current byte is a null(the end of a string) and break if it is
        if (strtab[i] == 0x00)
        {
            break;
        }

        // Char makes sure that the byte is taken as-is when appended to the string. There is no atoi() or any annoying .NET stuff in the way,
        // purely the byte will be parsed as a utf-8 character. As a side effect, if for some reason the strtab has some weird section names,
        // it's not going to look right. That is however an ELF thing and the user would be doing something insane if that was the case regardless
        Output += (char)strtab[i];
    }
    return Output;
}

```

Many validation checks are required for loading an ELF file.

```
// Check if the length is less than the minimum acceptable.
if (reader.Length < 0x34)
{
    Logger.Log(LogCode.IO_INVALIDFILE, "Incomplete ELF header");
    return null;
}
```

The first is a length check. This checks that there are enough bytes in the file for an ELF header, so the input file must be at least 52 bytes.

```
// Read and parse the rest of the header.
// All bytes are offset slightly because 5 bytes(magic bytes + class) have already been parsed.
// The offsets differ between elf32 and elf64 because addresses in elf64 must be 8 bytes; 4 bytes in elf32.
byte[] FileHeader;
switch (Elf_class[0])
{
    case 0x01:
        FileHeader = new byte[0x34 - 0x05];
        reader.Read(FileHeader, 0, FileHeader.Length);
        ParsedElf = new ELF(...);
        break;
    case 0x02:
        FileHeader = new byte[0x40 - 0x05];
        reader.Read(FileHeader, 0, FileHeader.Length);
        ParsedElf = new ELF(...);
        break;
    default:
        Logger.Log(LogCode.IO_INVALIDFILE, "Invalid class byte in ELF header");
        return null;
};
```

The second check is to determine whether the ELF class header is supported, so must have a value of 1 or 2. This is always the 4th-index byte in the file, so can be read directly and compared. This is a range check since if it is outside the range of 1 to 2, the file must be invalid. The class header is 1 if the file is 32-bit, or 2 if the file is 64-bit and other values are undefined. Although the program doesn't directly support 32-bit, it can be executed as if it were 64 bit as the fundamental operations performed by the CU and other modules are the same, but for better compatibility would need a separate opcode table in the future. If the range check fails, "Invalid class byte in ELF header" is logged and a message is shown to the user.

```
// Make sure ELF uses x86-x64 instruction set
if (FileHeader[0xD] != 0x03 && FileHeader[0xD] != 0x3E)
{
    Logger.Log(LogCode.IO_INVALIDFILE, "Input elf does not use the x86 instruction set");
    return null;
}
```

The next validation routine checks if the ELF file uses the correct instruction set by comparing the instruction set header byte with known values for x86-32 and x86-64. If they are not equal to then the input file must be designed for a different instruction set, so would not be compatible with the program and could cause errors and erroneous behaviour further down the line so it is rejected.

```
// Read sh_name from every section in shtab and check if the offset($shstrtab + $offset = target) is the string ".text".
// .text is holds the user code and therefore is the only section this program cares about.
// If there isn't one, which most likely means this wasn't an elf file, so handle it because this file cannot be used.
for (int i = 0; i < ParsedElf.SHCount; i++)
{
    if (ReadString(strtab, BitConverter.ToInt32(SHTable, i * ParsedElf.SHLength)) == ".text")
    {
        if (ParsedElf.Class == 32)...
        else...
        break;
    }
    else if (i + 1 == ParsedElf.SHCount)
    {
        Logger.Log(LogCode.IO_INVALIDFILE, "ELF File had no .text section");
        return null;
    }
}
```

The last validation procedure checks for the presence of a .text section. This is done by iterating through the shtab, which is a place in the file that contains all the section header names in ASCII format. If after the iteration is complete(by checking if the next iteration is the final iteration), there is no ".text" found in the shtab, an error message is displayed and the file is rejected. The text section is needed because it is the position in the ELF file where the user's code is stored.

Loading files from clipboard

```
OpenClipboardMenu.Click += (s, a) =>
{
    // Parse the clip board as text.
    IO.TXT ParsedClipboard = IO.TXT.Parse(System.Text.Encoding.UTF8.GetBytes(Clipboard.GetText(TextDataFormat.UnicodeText)));

    // If the output is null, there was an error somewhere, but this is already handled in the TXT class.
    // Regardless, the VM will be unchanged in this case.
    if (ParsedClipboard != null)
    {
        // This should not really be handled here, rather in the main class file.
        FlashProcedure(ParsedClipboard.Instructions);
    }
};
```

When the “Open from clipboard” button is pressed, instructions are read from the clipboard as UTF8 bytes and converted into a TXT file. This allows the TXT validation procedures to be reused. An additional validation procedure is also required. The if block checks if the TXT file returned null. This means that there was a validation error in the TXT class that has already been handled, but as there are no instructions returned it needs to also be handled here. This is essentially a type check, since the clipboard contained something other text such as an image.

Post development reflection

Key variables evidence

Name	Evidence
CU.CurrentHandle	<pre>public static Handle CurrentHandle = EmptyHandle; // Or</pre>
CU.CurrentContext	<pre>private static Context CurrentContext { get => CurrentHandle.ShallowCopy(); } //</pre>
CU.Flags	<pre>public static FlagSet Flags { get => CurrentContext.Flags; set { CurrentContext.Flags = value; } }</pre>
CU.InstructionPointer	<pre>public static ulong InstructionPointer { get => CurrentContext.InstructionPointer; private set { CurrentContext.InstructionPointer = value; } }</pre>
CU.RexByte	<pre>public static REX RexByte { get; private set; } = REX.NONE; // A protected field</pre>
Handle.HandleName	<pre>public readonly string HandleName;</pre>
Handle.HandleID	<pre>public readonly int HandleID;</pre>
ModRM.Destination	<pre>private ulong Destination;</pre>
ModRM.Source	<pre>public ControlUnit.RegisterHandle Source;</pre>
VM.Breakpoints	<pre>public ListeningList<ulong> Breakpoints = new ListeningList<ulong>();</pre>
OpcodeTable	<pre>private static readonly Dictionary<byte, Dictionary<byte, OpcodeCaller>> OpcodeTable = new Dictionary<byte, Dictionary<byte, OpcodeCaller>>()</pre>
Opcode.OpcodeSettings	<pre>public readonly OpcodeSettings Settings;</pre>
Opcode.Capacity	<pre>protected RegisterCapacity Capacity { get => _capacity; set { // Re-initialise the input with the new capacity. Input.Initialise(value); _capacity = value; } }</pre>
Opcode.Input	<pre>private readonly IMyDecoded Input;</pre>
Context.Flags	<pre>public FlagSet Flags = new FlagSet(FlagState.OFF);</pre>

Context.Memory	<pre>public MemorySpace Memory;</pre>
Context.Registers	<pre>public RegisterGroup Registers = new RegisterGroup();</pre>
Context.InstructionPointer	<pre>public ulong InstructionPointer;</pre>
StringOperation.SrcPtr	<pre>// the registers every time protected ulong SrcPtr;</pre>
StringOperation.DestPtr	<pre>protected ulong DestPtr;</pre>

Key variables reflection

In the alpha stage of development, hungarian notation was used as convention for identifier naming. At the time, this was convenient as it allows me to quickly see the type of a variable when looking at the identifier. However, as development continued, variable names seemed to become more convoluted because of this, so all variables were renamed to use pascal case. This also keeps the code easier to read for someone who doesn't use hungarian notation as it could be confusing why there are seemingly arbitrary letters at the start of an identifier. (In hungarian notation, the prefix of an identifier is used to denote its type, e.g. "sMyString", the 's' indicates a string.) For parameters to subroutines, camel case is used to help differentiate between local and global variables.

Validation for key elements of solution

The key elements of the solution identified for validation during development were,

- Username and password login box p173-174
- Loading instructions from files p174-181
- Loading instructions from clipboard p181
- Reading and parsing testcase files p46-p49

These are the only sources of user input into the program, so by handling validation here, there were very few cases where further validation was necessary. Some modules that were designed to be entirely independent from the project, such as the bitwise library, used their own validation as required. It was important to perform validation at these key elements such that there would always be a high level of trust between different modules so the inputs and outputs between modules were always valid and robust, rather than requiring validation in every class/subroutine which would have taken too much development time and been less efficient than the current validation plan post-release.

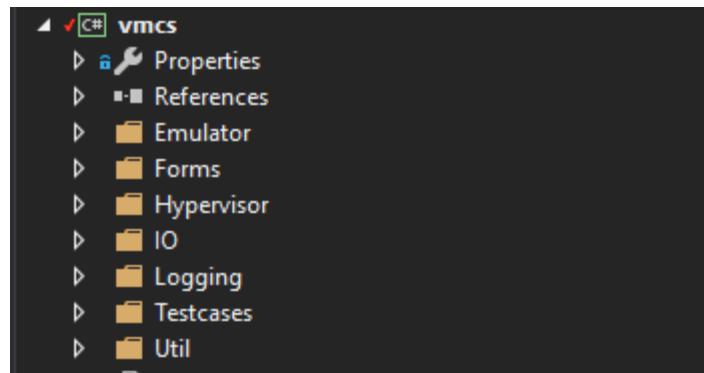
The username and login password box required validation because there were certain requirements and restrictions on the usernames and passwords that the user could have, which made sure the user was less likely to forget their credentials or cause errors in the rest of the program whenever the inputs contain obscure multi-byte unicode characters(which are now rejected by the validation).

Loading instructions from files requires validation because the user can select any file in the windows file selection menu. This could potentially allow them to input a picture into the program, causing it to crash. It could also be the case when a file has been corrupted or compiled improperly, which would make it unreadable to the internal modules of the program such as the ControlUnit. These validation procedures prevent this ever being a problem by having a whitelist of minimum requirements for a given input file, which is implemented by a sequence of presence, length, and range checks. This means that no matter the rest of the contents of the file, it will always be compatible with the rest of the program if it passes the validation routines.

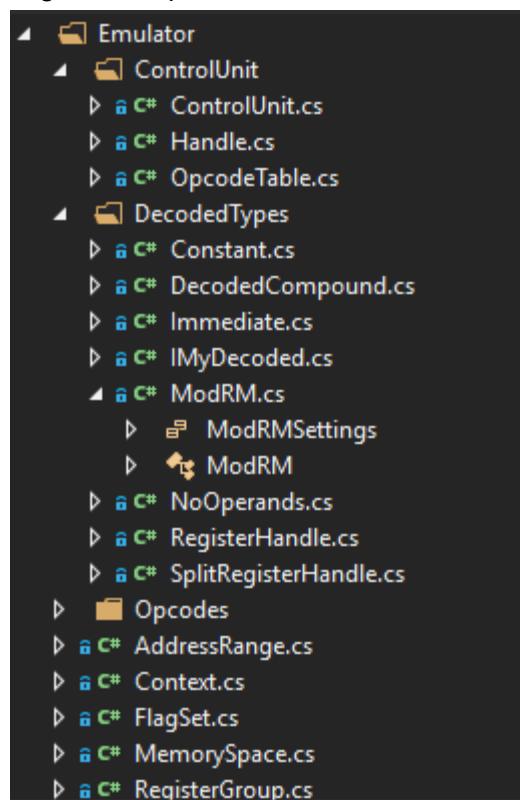
Loading instructions from the clipboard reused validation procedures from "loading instructions from files" for the most part since they were interpreted in the same way as a text file, however, some further validation was needed. This was a type check to ensure that the clipboard was text and not an image etc.

Reading and parsing testcase files required multiple validation checks because the input file type was XML. This means that the file had to be checked to ensure that it was a proper XML file in a correct and parsable format. This had to be done immediately after the file was opened because the XML reader is provided by the .NET framework and will throw exceptions if the XML file is invalid. The first check is a type check, which checks that the file extension of the file is ".xml". After that each element and attribute of the XML file is iterated and parsed using the ParseHex validation routine which iterates through the input converting bytes such as "AA" written in ASCII to 0xAA in memory/in a variable. It also checks that each element, e.g. a Checkpoint, has the minimum attributes and sub-elements required, which is done by a sequence of presence checks, though this was discussed further in the development section.

Structure and modular nature



The program consists of several namespaces. Each namespace represents a different part of the program. Namespaces contain each of the modules. This means, for example, to add a new hypervisor module, it need only be added to the Hypervisor namespace then can be used by any class that imports the hypervisor namespace. Some large namespaces are further broken down into sub-namespaces

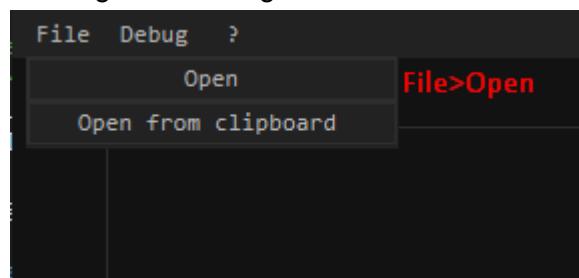


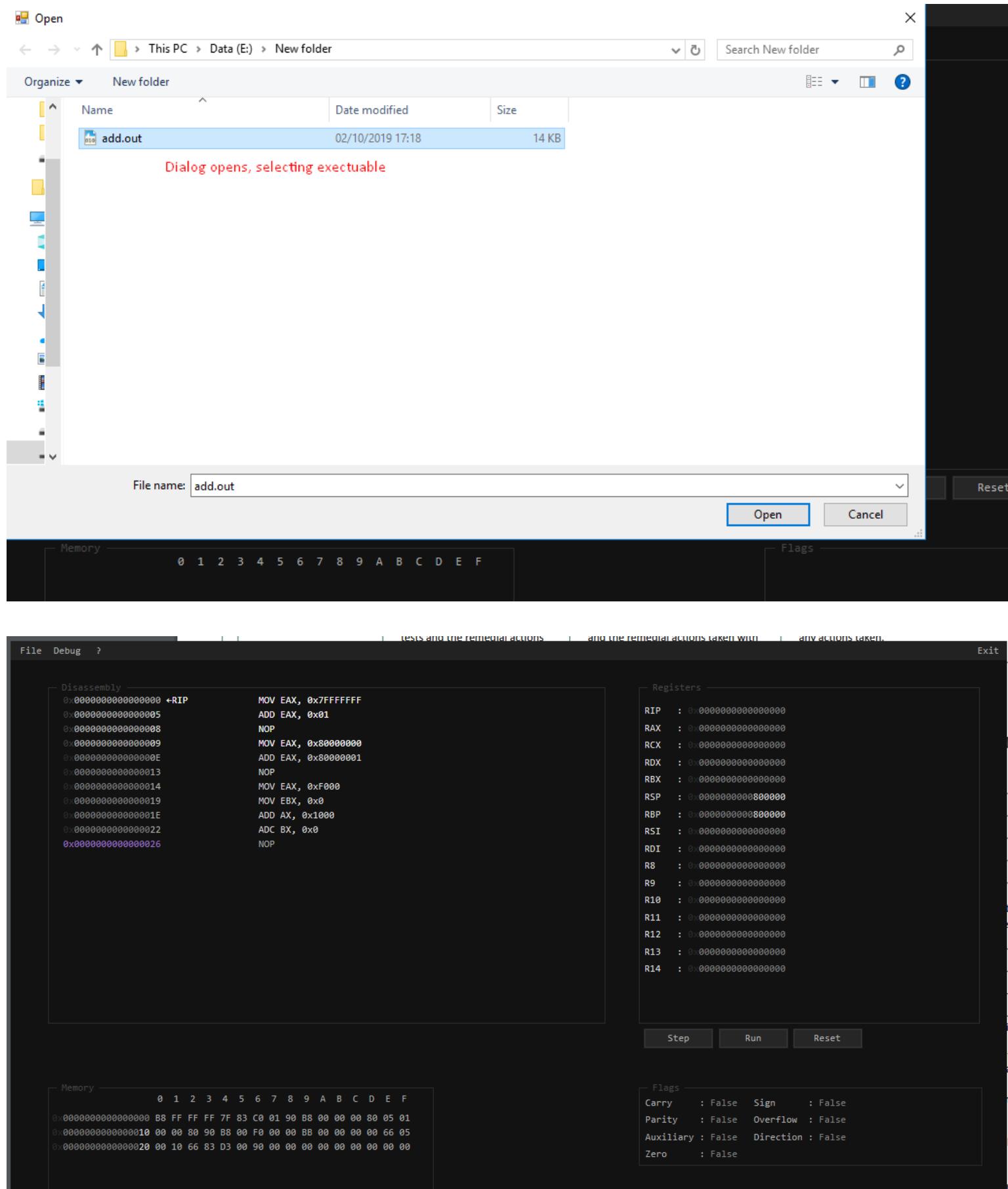
This further reduces the need for any modules to import code that they do not depend on. For example, the a class may only need the DecodedTypes objects need only import the DecodedTypes namespace to have access to all the modules contained, rather than have to import the entire Emulator namespace.

Splitting the project into many class files also improved modularity. This means that any developer or maintainer in the future could swap out a class file for a new class file and the program would continue to function with the new file(provided it was coded and tested properly).

Testing to inform evaluation/Post development testing

Loading a file testing





Annotation

GUI was used to open a file that contained instructions using the optimal method (performed by myself). As shown in the last picture, the file loaded correctly.

Usability testing of feature

A classmate/stakeholder was asked to perform this task without any further intervention (a competent python programmer, no assembly knowledge). The events of the video recording are outlined here.

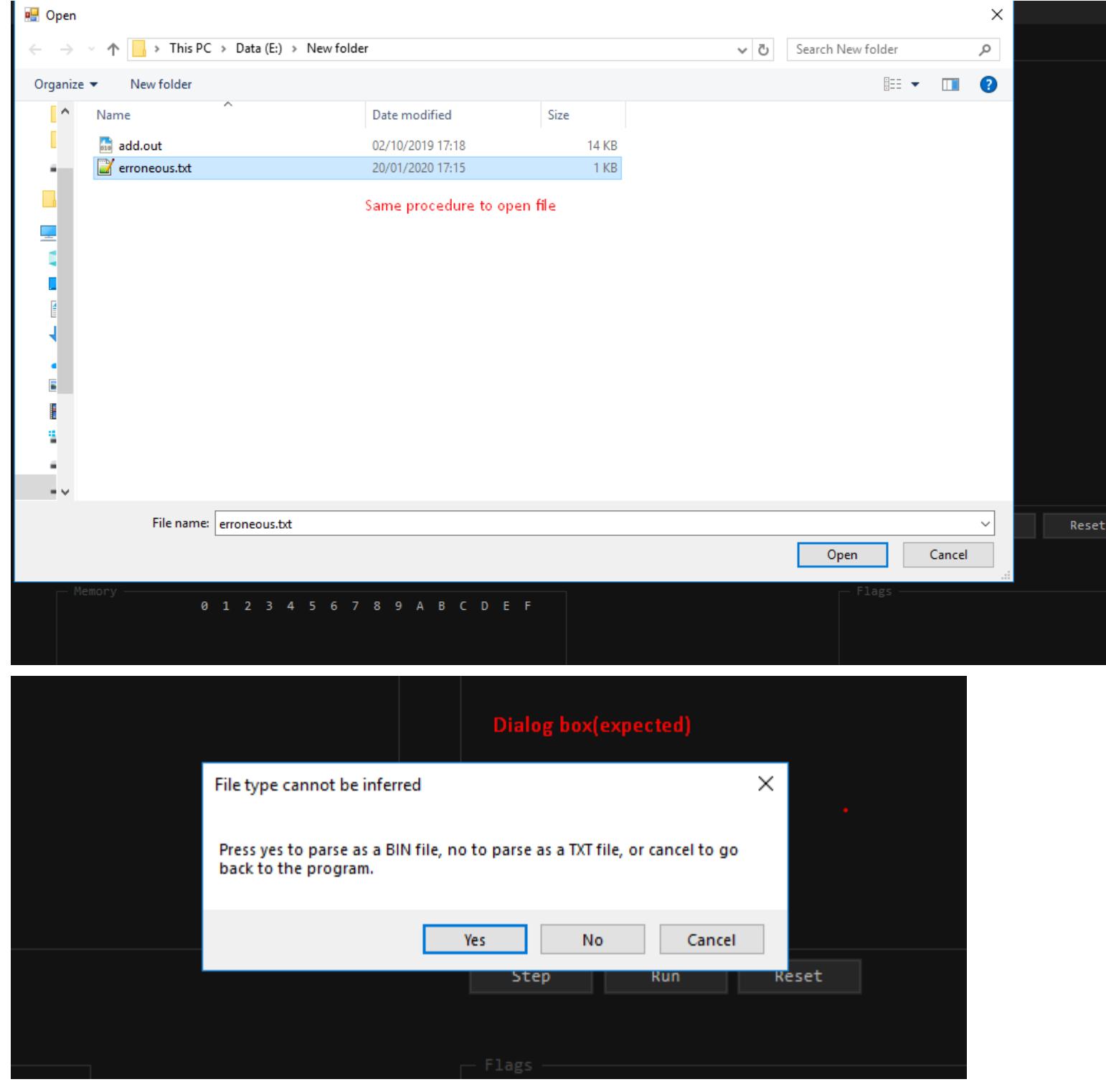
1. User opens program
2. User clicked file then open
3. User opened the file

(Once in program, 1 click to get to “open”)

The user testing the programmed followed the optimal method for opening the file; they did not need any guidance from help features or myself. This shows that the “Core features will be easily accessible through the UI” has been met and that the “User interface interaction will be kept minimal” criterion has been met without any drawbacks for this function. This is because the user who had no previous experience with the program had been able to easily reproduce the same test results that were conducted by myself.

Robustness testing of feature

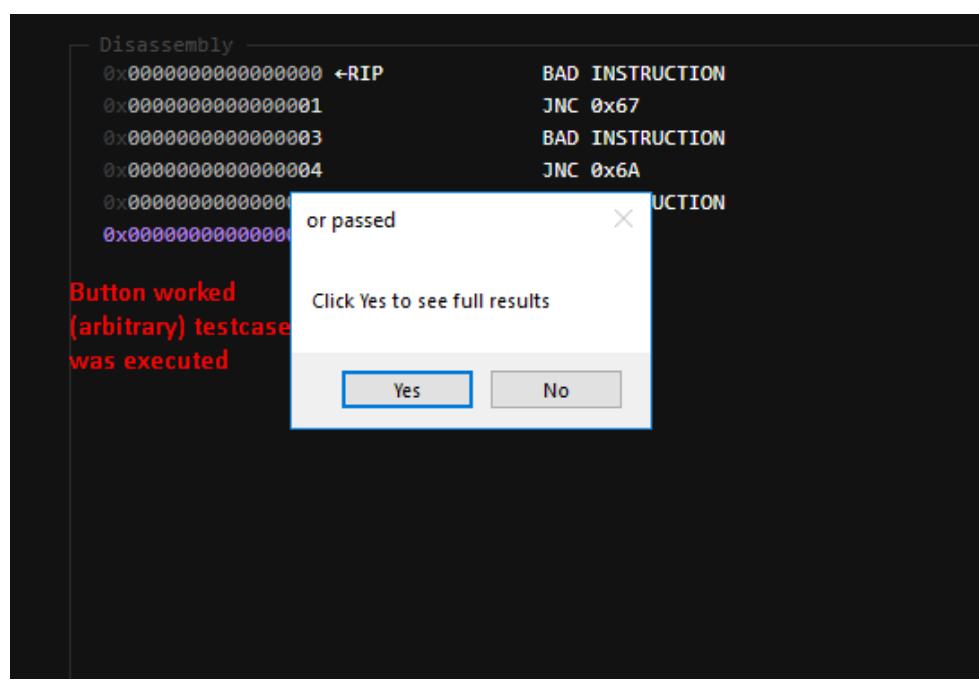
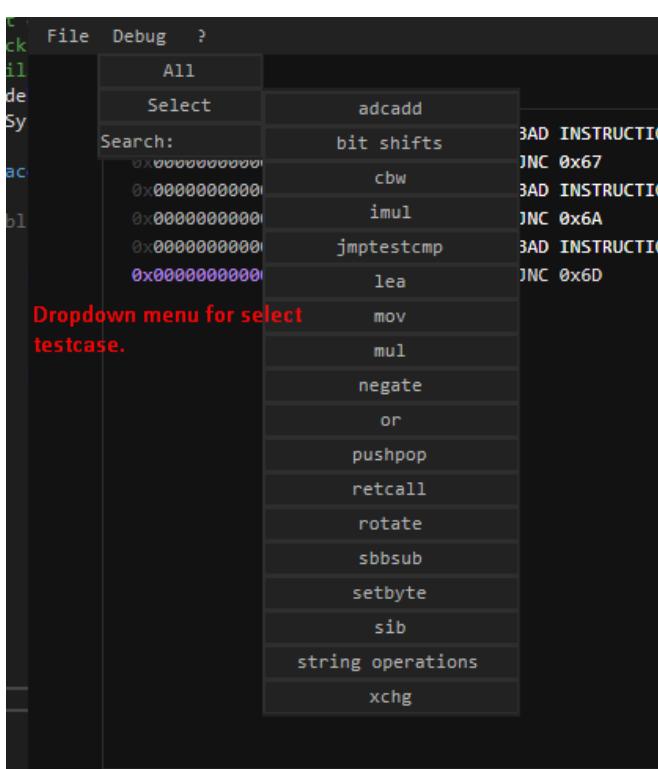
To test this, an erroneous input file was used that was plaintext.



```
est :     .text
heck File Debug ?
imil
g de
g Sy
spa
publ
{
    Disassembly
    0x0000000000000000 +RIP      BAD INSTRUCTION
    0x0000000000000001          JNC 0x67
    0x0000000000000003          BAD INSTRUCTION
    0x0000000000000004          JNC 0x6A
    0x0000000000000006          BAD INSTRUCTION
    0x0000000000000007          JNC 0x6D
```

As designed, the program still attempts to parse the file(as assembly code is a sequence of bytes, almost any file could be interpreted as assembly) then replaces any bytes it cannot parsed with “BAD INSTRUCTION” instead of crashing the program. This still gives a meaningful output to the user and shows that this feature is robust.

Executing testcases testing



cycle Events ▾ Thread:

TestHandler.cs

Core

Results were displayed
visually

0000 ←RIP

0001

0003

0004

0006

0007

0 1 2 3 4 5 6

00 61 73 64 61 73 64 61

```
<TestcaseResult name="or" result="Passed">
  <CheckpointResult Tag="Immediate A">
    <SubCheckpointResult result="Passed">
      <Expected> $RAX=0xFFFFFFFFFFFFFF</Expected>
      <Found> $RAX=0xFFFFFFFFFFFFFF</Found>
    </SubCheckpointResult>
  </CheckpointResult>
  <CheckpointResult Tag="Immediate">
    <SubCheckpointResult result="Passed">
      <Expected> $RBX=0xFFFFFFFFFFFFFF</Expected>
      <Found> $RBX=0xFFFFFFFFFFFFFF</Found>
    </SubCheckpointResult>
  </CheckpointResult>
  <CheckpointResult Tag="Immediate sign extended byte">
    <SubCheckpointResult result="Passed">
      <Expected> $AL=0x33</Expected>
      <Found> $AL=0x33</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> $RBX=0x11113333</Expected>
      <Found> $RBX=0x11113333</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> $RCX=0x1111111333333333</Expected>
      <Found> $RCX=0x1111111333333333</Found>
    </SubCheckpointResult>
  </CheckpointResult>
  <CheckpointResult Tag="Immediate to mem">
    <SubCheckpointResult result="Passed">
      <Expected> [RSP]={FF}</Expected>
      <Found> [RSP]={FF}</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> [RSP+0x1]={FF, FF}</Expected>
      <Found> [RSP+0x1]={FF, FF}</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> [RSP+0x4]={FF, FF, FF, FF}</Expected>
      <Found> [RSP+0x4]={FF, FF, FF, FF}</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> [RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Expected>
      <Found> [RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Found>
    </SubCheckpointResult>
  </CheckpointResult>
  <CheckpointResult Tag="Reg to mem">
    <SubCheckpointResult result="Passed">
      <Expected> [RSP]={FF}</Expected>
      <Found> [RSP]={FF}</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> [RSP+0x1]={FF, FF}</Expected>
      <Found> [RSP+0x1]={FF, FF}</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> [RSP+0x4]={FF, FF, FF, FF}</Expected>
      <Found> [RSP+0x4]={FF, FF, FF, FF}</Found>
    </SubCheckpointResult>
    <SubCheckpointResult result="Passed">
      <Expected> [RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Expected>
      <Found> [RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Found>
    </SubCheckpointResult>
  </CheckpointResult>
</TestcaseResult>
```

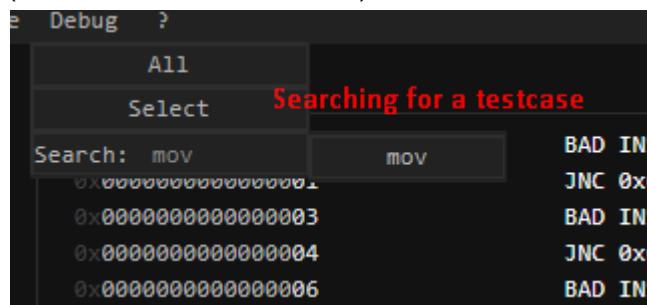
OK

orTestcase.xml - Notepad

File Edit Format View Help

```
<?xml version="1.0" encoding="utf-8"?>
<TestcaseResult name="or" result="Passed">
  <CheckpointResult Tag="Immediate A">
    <SubCheckpointResult result="Passed">
      <Expected>$RAX=0xFFFFFFFFFFFFFF</Expected>
      <Found>$RAX=0xFFFFFFFFFFFFFF</Found>
    </SubCheckpointResult>
  </CheckpointResult>
  <CheckpointResult Tag="Immediate">
    <SubCheckpointResult result="Passed">
      <Expected>$RBX=0xFFFFFFFFFFFFFF</Expected>
      <Found>$RBX=0xFFFFFFFFFFFFFF</Found>
    </SubCheckpointResult>
  </CheckpointResult>
  <CheckpointResult Tag="Immediate sign extended byte">
    <SubCheckpointResult result="Passed">
      <Expected>$AL=0x33</Expected>
    </SubCheckpointResult>
  </CheckpointResult>
</TestcaseResult>
```

(Results were written to file)

Annotation

The GUI was used to execute a testcase. Since the same button class is used for all testcases, it can be said that all testcases will be executed the same. The search feature also worked.

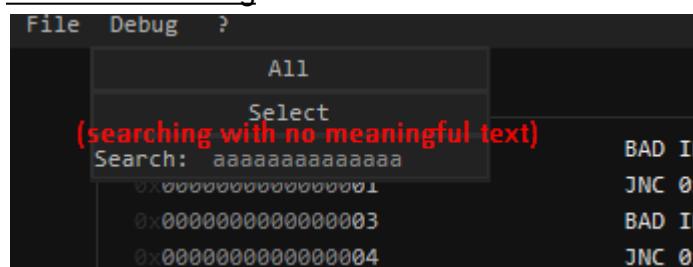
Usability testing

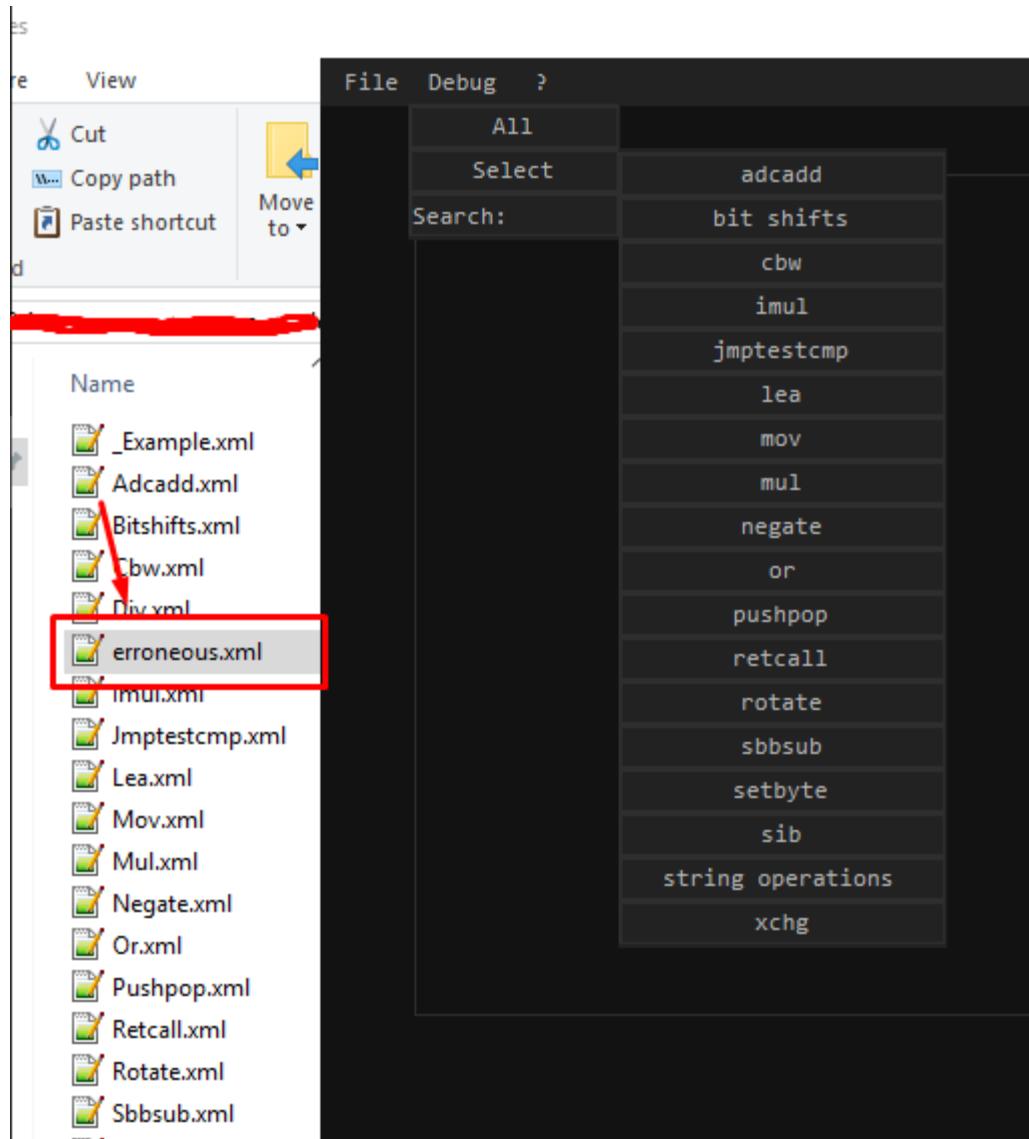
A classmate was instructed to “execute the mov testcase” with no further intervention. The events in the video are detailed as follows,

1. User opens program
2. User looks around for a few seconds then clicks on help button
3. User reads “execute a testcase” section of help
4. User goes to debug and executes mov testcase by search

(2 clicks)

This shows that the “Core features will be easily accessible through the UI” success criterion has been met in this function. This is because it did not take the user long to figure out how to access the testcase and they figured it out entirely on their own. This has sufficiently met the criterion as only two clicks were required by the user. Most interaction was done through hovering and typing, however was still easy for the user to complete the task.

Robustness testing



The box responded well when erroneous text was entered. It also ignored any non alphanumeric input as designed and there was no erroneous behaviour when pressing backspace with no input text (but hard to show in a picture). Another aspect of robustness tested was how the program handles erroneous testcase files. As designed in the testhandler section, the file was ignored (hence not included in the testcase list).

User code execution environment testing

The screenshot shows a debugger interface with a dark theme. On the left, the 'Disassembly' pane displays assembly code. The code consists of several MOV and ADD instructions, starting with 'MOV EAX, 0xFFFFFFFF' at address 0x0000000000000000 and continuing with 'ADD EAX, 0x01', 'NOP', 'MOV EAX, 0x80000000', 'ADD EAX, 0x80000001', 'NOP', 'MOV EAX, 0xF000', 'MOV EBX, 0x0', 'ADD AX, 0x1000', 'ADC BX, 0x0', and 'NOP'. In the center, the 'State when first loaded code' section is labeled in red. On the right, the 'Registers' pane lists all general-purpose registers (RIP, RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14) with their values set to 0.00000. At the bottom right, there is a 'Step' button.

Disassembly

```

0x0000000000000000    MOV EAX, 0xFFFFFFFF
0x0000000000000005<RIP ADD EAX, 0x01
0x0000000000000008    NOP
0x0000000000000009    MOV EAX, 0x80000000
0x000000000000000E    ADD EAX, 0x80000001
0x0000000000000013    NOP
0x0000000000000014    MOV EAX, 0xF000
0x0000000000000019    MOV EBX, 0x0
0x000000000000001E    ADD AX, 0x1000
0x0000000000000022    ADC BX, 0x0
0x0000000000000026    NOP

```

Registers

RIP	: 0x0000000000000000
RAX	: 0x0000000000000000
RCX	: 0x0000000000000000
RDX	: 0x0000000000000000
RBX	: 0x0000000000000000
RSP	: 0x0000000000000000
RBP	: 0x0000000000000000
RSI	: 0x0000000000000000
RDI	: 0x0000000000000000
R8	: 0x0000000000000000
R9	: 0x0000000000000000
R10	: 0x0000000000000000
R11	: 0x0000000000000000
R12	: 0x0000000000000000
R13	: 0x0000000000000000
R14	: 0x0000000000000000

Step

Disassembly

```

0: 0000000000000000    MOV EAX, 0xFFFFFFFF
0x0000000000000005    ADD EAX, 0x01
0x0000000000000008    NOP
0x0000000000000009    MOV EAX, 0x80000000
0x000000000000000E    ADD EAX, 0x80000001
0x0000000000000013    NOP
0x0000000000000014    MOV EAX, 0xF000
0x0000000000000019    MOV EBX, 0x0
0x000000000000001E    ADD AX, 0x1000
0x0000000000000022    ADC BX, 0x0
0: 0000000000000026    NOP

```

Registers

RIP	: 0x0000000000000027
RAX	: 0x0000000000000000
RCX	: 0x0000000000000000
RDX	: 0x0000000000000000
RBX	: 0x0000000000000001
RSP	: 0x0000000000000000
RBP	: 0x0000000000000000
RSI	: 0x0000000000000000
RDI	: 0x0000000000000000
R8	: 0x0000000000000000
R9	: 0x0000000000000000
R10	: 0x0000000000000000
R11	: 0x0000000000000000
R12	: 0x0000000000000000
R13	: 0x0000000000000000
R14	: 0x0000000000000000

Run pressed
(as expected, all instructions executed so RIP not visible)

Memory

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0000000000000000	B8	FF	FF	FF	7F	83	C0	01	90	B8	00	00	00	80	05	01
0x0000000000000010	00	00	80	90	B8	00	F0	00	00	BB	00	00	00	00	66	05
0x0000000000000020	00	10	66	83	D3	00	90	00	00	00	00	00	00	00	00	00

Flags

Carry	: False	Sign	: False
Parity	: False	Overflow	: False
Auxiliary	: False	Direction	: False
Zero	: False		

Step Run Reset

```

- Disassembly
0x0000000000000000 +RIP      MOV EAX, 0xFFFFFFFF
0x0000000000000005          ADD EAX, 0x01
0x0000000000000008          NOP
0x0000000000000009          MOV EAX, 0x80000000
0x000000000000000E          ADD EAX, 0x80000001
0x0000000000000013          NOP
0x0000000000000014          MOV EAX, 0xF000
0x0000000000000019          MOV EBX, 0x0
0x000000000000001E          ADD AX, 0x1000
0x0000000000000022          ADC BX, 0x0
0x0000000000000026          NOP

```

Reset pressed after run

Registers	
RIP	: 0x0000000000000000
RAX	: 0x0000000000000000
RCX	: 0x0000000000000000
RDX	: 0x0000000000000000
RBX	: 0x0000000000000000
RSP	: 0x0000000000080000
RBP	: 0x0000000000080000
RSI	: 0x0000000000000000
RDI	: 0x0000000000000000
R8	: 0x0000000000000000
R9	: 0x0000000000000000
R10	: 0x0000000000000000
R11	: 0x0000000000000000
R12	: 0x0000000000000000
R13	: 0x0000000000000000
R14	: 0x0000000000000000

Registers set
to default
values

Step Run Reset

Memory	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000000000000000	B8	FF	FF	FF	7F	83	C0	01	90	B8	00	00	00	80	05	01
0000000000000000	10	00	00	80	90	B8	00	F0	00	00	BB	00	00	00	66	05
0000000000000000	20	00	10	66	83	D3	00	90	00	00	00	00	00	00	00	00

Memory wiped without affecting
bytes of instructions

Flags	
Carry	: False
Parity	: False
Auxiliary	: False
Zero	: False
Sign	: False
Overflow	: False
Direction	: False

(Flags also reset, but could only be observed by stepping)

Annotation

The features of the run time environment were tested by loading arbitrary instructions and analysing the effects of each button pressed.

Usability testing

A classmate was used to test the usability of this feature. They were first briefed on what was meant by “Step” and “Run”. It is fair to assume that the stakeholders described in the analysis section will already understand what is meant by these terms, and it is not hard to figure out by pressing them. The classmate was asked to first run the instructions using the Run button, then execute them a second time by stepping through without restarting the program(they had to figure that they would need to use the reset button)

1. User opens program
2. User opens file(irrelevant to this test)
3. User presses run
4. User immediately pressed reset then stepped through again

4 clicks

This shows that the “User interface must be intuitive.” success criterion has been met as the user was able to figure out on their own how to use the interface with no significant hesitation.

Robustness testing

To test this, I tried various button combinations and erroneous behaviour(such as spamming reset and run). No erroneous behaviour was observed, the program did nothing when reset was pressed and only executed once when run was pressed(since run does not automatically reset the program when reaching the end).

Open file from clipboard

over, so you don't have to remove the double quotes or "\x" if you

```
B8 FF FF FF 7F 83 C0 01 90 B8 00 00 00 80 05 01  
00 00 80 90 B8 00 F0 00 00 BB 00 00 00 00 66 05  
00 10 66 83 D3 00 90
```

Copied into clipboard

Disassembly

Raw Hex (zero bytes in bold):

```
B8FFFFFF7F83C00190B800000080050100008090B800F00000BB€
```

String Literal:

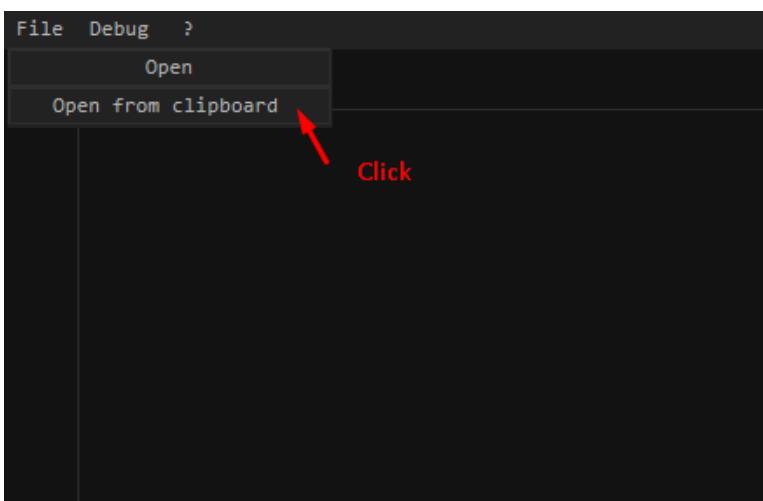
```
"\xB8\xFF\xFF\xFF\x7F\x83\xC0\x01\x90\xB8\x00\x00\x06
```

Array Literal:

```
{ 0xB8, 0xFF, 0xFF, 0xFF, 0x7F, 0x83, 0xC0, 0x01, 0x90,  
0x05, 0x01, 0x00, 0x00, 0x80, 0x90, 0xB8, 0x00, 0xF0,  
0x00, 0x00, 0x66, 0x05, 0x00, 0x10, 0x66, 0x83, 0xD3,
```

Disassembly:

```
0: b8 ff ff ff 7f          mov    eax,0xffffffff  
5: 83 c0 01                add    eax,0x1  
8: 90                      nop  
9: b8 00 00 00 80          mov    eax,0x80000000  
e: 05 01 00 00 80          add    eax,0x80000001  
13: 90                      nop  
14: b8 00 f0 00 00          mov    eax,0xf000  
19: bb 00 00 00 00          mov    ebx,0x0  
1e: 66 05 00 10              add    ax,0x1000  
22: 66 83 d3 00              adc    bx,0x0  
26: 00                      nop
```



Click

```
Disassembly
0x0000000000000000 ←RIP      MOV EAX, 0xFFFFFFFF
0x0000000000000005          ADD EAX, 0x01
0x0000000000000008          NOP
0x0000000000000009          MOV EAX, 0x80000000
0x000000000000000E          ADD EAX, 0x80000001
0x0000000000000013          NOP
0x0000000000000014          MOV EAX, 0xF000
0x0000000000000019          MOV EBX, 0x0
0x000000000000001E          ADD AX, 0x1000
0x0000000000000022          ADC BX, 0x0
0x0000000000000026          NOP

Code loaded
```

Annotation

I assembled some code online(as shown in the first image alongside the instructions) then copied the bytes(in ascii) into my clipboard. I could then load that into the program and as shown, the instructions were the same.

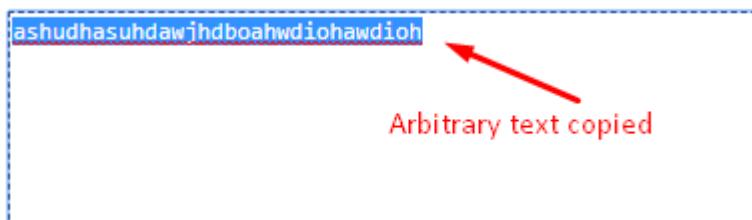
Usability testing

A classmate was asked to load code into the program, given the same website open(the assembly code was already entered by myself). They were not explicitly told about the clipboard feature.

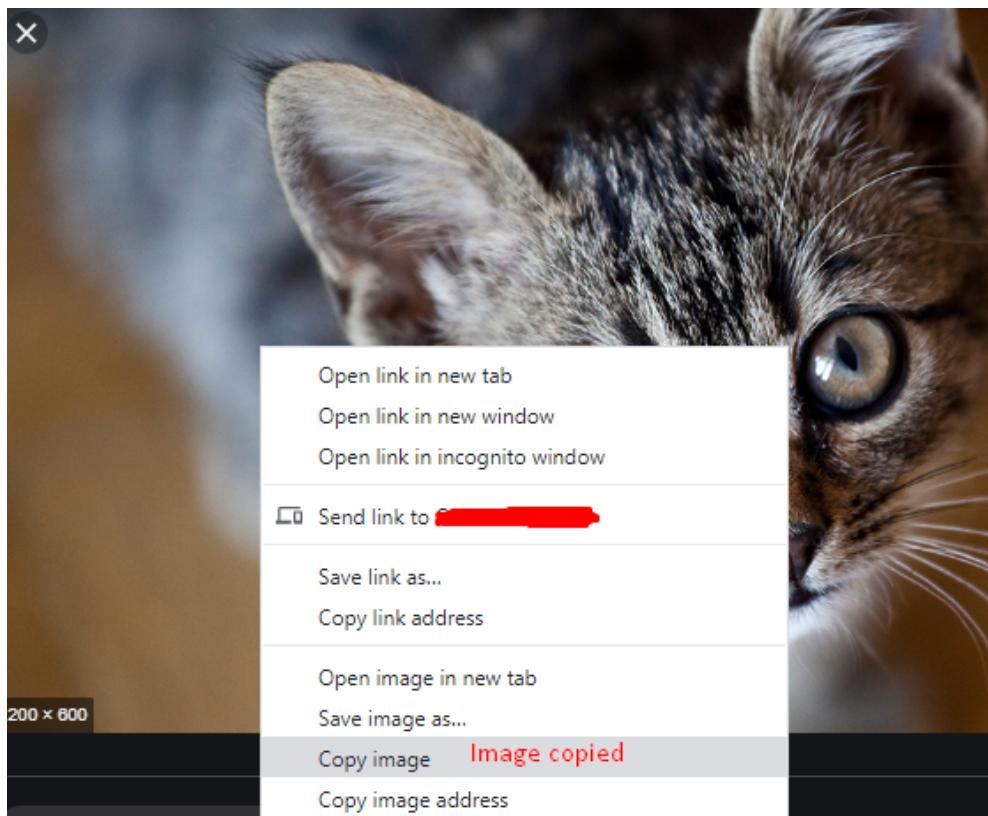
1. User opens program
2. User hovers over file and immediately sees open from clipboard
3. User presses it(without code copied)
4. User goes back to window and copies the code
5. User presses open from clipboard again

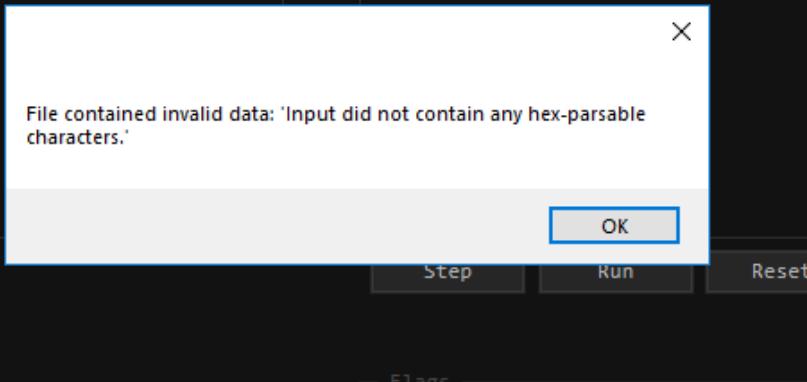
4 clicks

This shows that the “Core features will be easily accessible through the UI” success criterion has been met for this feature. This is because the user required only one click to find out about the clipboard feature(it was not hidden) and was easily able to use it. The user pressing the clipboard button the first time with nothing in their clipboard can be ruled as an anomaly.

Robustness testing

```
Disassembly
0x0000000000000000 +RIP LODSD EAX, [RSI]
0x0000000000000001 LODSD EAX, [RSI]
0x0000000000000002 LODSD EAX, [RSI]
0x0000000000000003 MOV EDX, 0xD0DA
```





To test this, erroneous data was copied into the clipboard to try produce erroneous behaviour. The program handled the data as it was designed to. Any data that could have possibly had meaning, such as the ascii text was parsed as if the instructions shown in the disassembly were intended as the code. When the input was entirely unparsable(likely metadata and windows related technicalities involved in the process of handling an image), the program displayed a warning and did not crash afterwards.

UI intuition testing

VMCS

Use this picture when answering the next questions

The figure shows a debugger interface with four numbered sections:

- 1:** Assembly code window. The code is:

```
0000000000000000 +10P      MOV RAX, 0xFFFFFFFF  
0000000000000005    ADD RAX, RDX  
0000000000000008    NOP  
000000000000000B    MOV RAX, 0x00000000  
0000000000000010    ADD RAX, 0x00000001  
0000000000000013    NOP  
0000000000000014    MOV RAX, 0x00000000  
0000000000000019    MOV RDX, RDX  
000000000000001E    ADD AX, 0x1000  
0000000000000021    ADC BX, 0x49  
0000000000000026    NOP
```
- 2:** Registers window. The registers show their initial values:

RIP	: 0000000000000000
RAX	: 0000000000000000
RCX	: 0000000000000000
RDX	: 0000000000000000
RSI	: 0000000000000000
RSP	: 0000000000000000
RCX	: 0000000000000000
RDI	: 0000000000000000
RS	: 0000000000000000
RF	: 0000000000000000
RB	: 0000000000000000
RD	: 0000000000000000
RSI	: 0000000000000000
RDI	: 0000000000000000
RSI	: 0000000000000000
RDI	: 0000000000000000
- 3:** Status flags window. The flags are:

Carry	: False
Sign	: False
Parity	: False
Overflow	: False
Auxiliary	: False
Direction	: False
Zero	: False
- 4:** Stack dump window. The stack contains:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0000000000000000	10	FF	FF	FF	7F	83	C0	01	00	00	00	00	00	00	05	01
0000000000000001	10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	95
0000000000000002	20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Option 1

- ### Option 1

What is displayed at 1

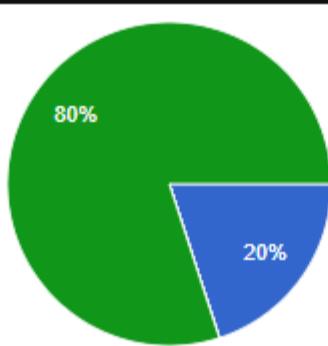
- Registers
 - Flags
 - Memory
 - Disassembly

What is displayed at 2?

- ## Registers

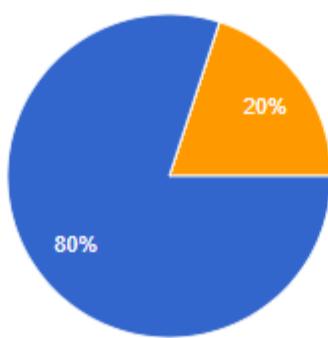
What is displayed at 1

5 responses



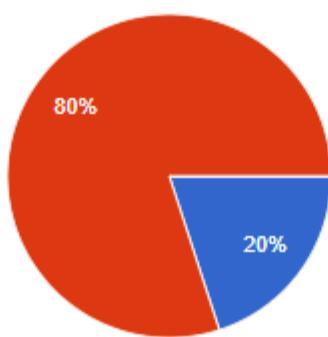
What is displayed at 2

5 responses



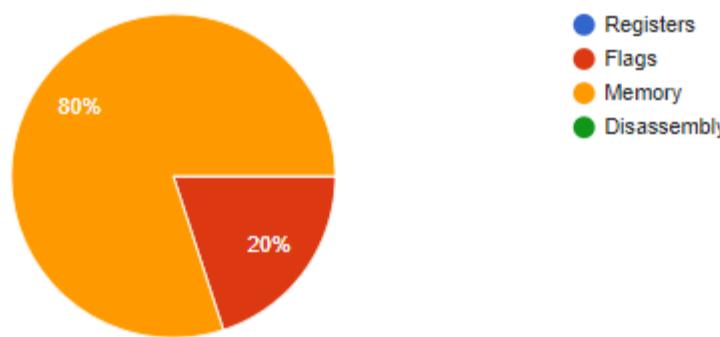
What is displayed at 3

5 responses



What is displayed at 4

5 responses



Annotation

Stakeholders selected from my classmates at school were given a picture of the interface along with some questions asking what each UI element displayed, with the name of each box hidden. The results of the questions for the test are shown above. In each case, 4 out of 5 got the correct answer.

Code abstraction testing

```
using debugger.Util;
namespace debugger.Emulator.Opcodes
{
    public class Double : Opcode
    {
        public Double(DecodedTypes.IMyDecoded input, OpcodeSettings settings = OpcodeSettings.NONE) : base("DBL", input, settings)
        {
        }
        public override void Execute()
        {
            byte[] Result;
            FlagSet ResultFlags = Bitwise.Multiply(Fetch()[0], new byte[] { 2 }, false, 8, out Result);
            ControlUnit.SetFlags(ResultFlags);
            Set(Result);
        }
    }

    using debugger.Util;
    using System.Collections.Generic;
    namespace debugger.Emulator.Opcodes
    {
        public class Square : Opcode
        {
            public Square(DecodedTypes.IMyDecoded input, OpcodeSettings settings = OpcodeSettings.NONE) : base("SQR", input, settings)
            {
            }
            public override void Execute()
            {
                List<byte[]> DestSource = Fetch();
                byte[] Result;
                FlagSet ResultFlags = Bitwise.Multiply(DestSource[0], DestSource[1], false, 8, out Result);
                ControlUnit.SetFlags(ResultFlags);
                Set(Result);
            }
        }
    }
}
```

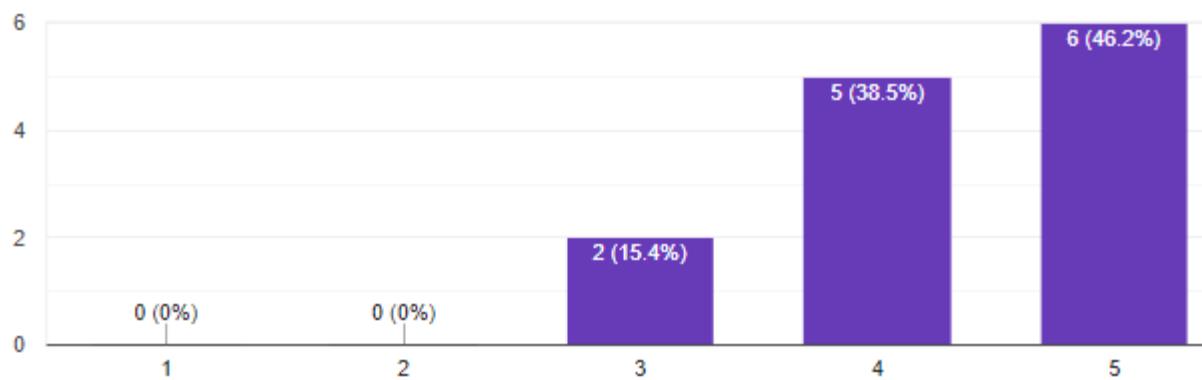
Annotation

Classmates were given the source of the project and asked to create their own opcode using the inbuilt libraries. This test was designed to show whether the users were able to quickly pick up and learn how to use the libraries in the space of one lesson by seeing them in context with other opcodes or by viewing the source of a library function.

UI review

Rating

13 responses



A great UI. Information is clearly sorted in a logical order and the buttons are placed in very natural positions. The dark theme makes the program very presentable and creates a good atmosphere. Features have clearly been highlighted appropriately, such as colouring breakpoints and dimming 00 bytes.

Looks great. Glad you were able to reduce the amount of information compared to the alternative software showcased earlier, such as removing empty lines from the memory viewer. The borders also add a really nice touch.

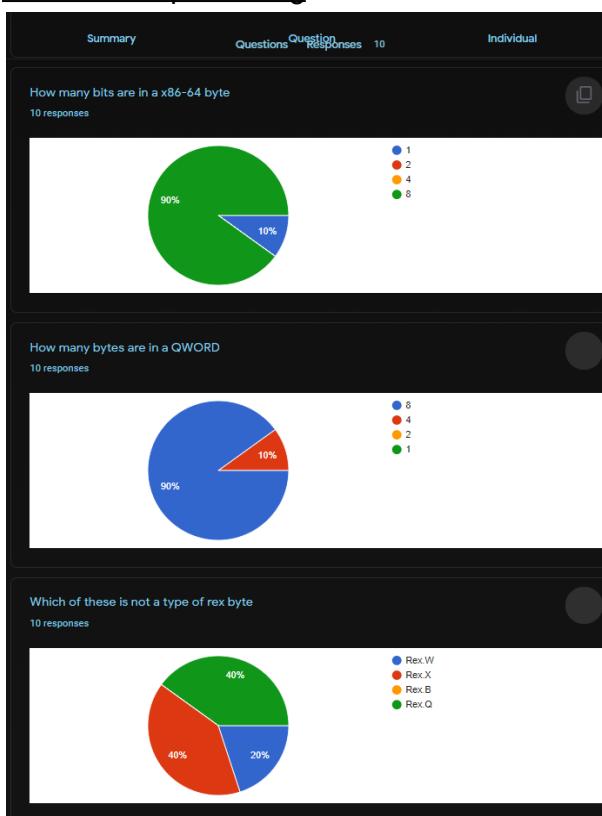
Seems fine to me. I think it would be a little better if there was something to occupy the bottom right corner as it feels a little empty, but obviously could be addressed later.

Good positioning of GUIs. I like how the name of each section is discretely put around the border, clear to see when viewing the first time but not obstructive once you know what you're doing.

Annotation

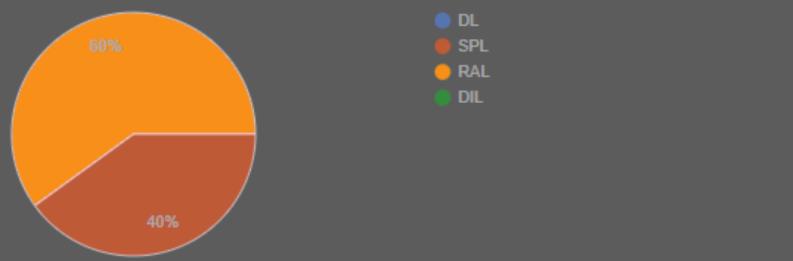
These are the results taken from a survey given to my classmates who met categories of stakeholders. This allowed stakeholders to give back on the UI, which will be important for evaluating success criteria.

Comment depth testing



Which of these is not a lower 8 bit register

10 responses



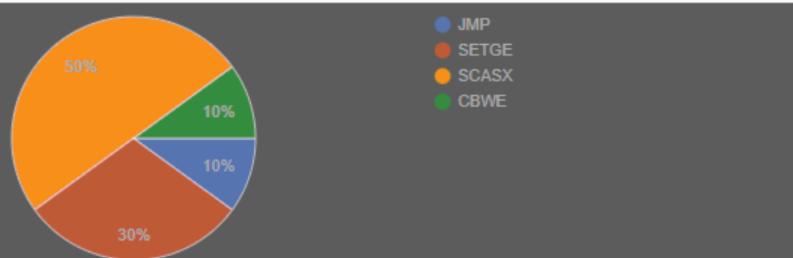
Which of these is not a flag

10 responses



Which of these is not an opcode

10 responses



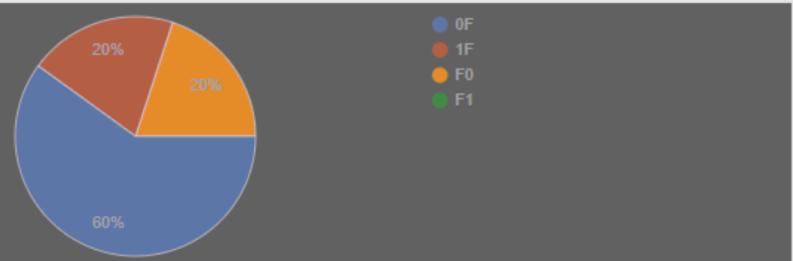
Which of these is not an opcode prefix

10 responses



Which of these is the prefix byte for a two byte opcode

10 responses



Annotation

Classmates were given time to read various parts of the code that contained enough information to deduce the answers to a short test given afterwards. Above are the test scores, where in each case, the majority or half got the correct answer.

Performance testing

```
622     public static async Task<XElement> ExecuteAll()
623     {
624         // ExecuteAll() will do some very useful things for the user. All the out-
625         // own "result" attribute for easy identification of whether all testcases
626         // time looking for a failure every time. The UI also makes use of this.
627
628         XElement RootElement = new XElement("all");
629
630         // ExecuteAll() has to do its own thing with running testcases and cannot
631         // This is because of the "result" attribute; checking whether each testca
632         bool Passed = true;
633         foreach (var Testcase in Testcases)
634         {
635             TestingEmulator Emulator = new TestingEmulator(Testcase.Value);
636             TestcaseResult Result = await Emulator.RunTestcase();
637             RootElement.Add(Result.ToXML(Testcase.Key));
638             Passed &= Result.Passed;
639         }
640         RootElement.SetAttributeValue("result", Passed ? "Passed" : "Failed");
641         return RootElement;
642     }
643     public static string[] GetTestcases() => Testcases.Keys.ToArray();
644
631     // This is because of the "result" attribute; checking whether each testcase passed or
632     bool Passed = true;
633     foreach (var Testcase in Testcases)
634     {
635         TestingEmulator Emulator = new TestingEmulator(Testcase.Value);
636         TestcaseResult Result = await Emulator.RunTestcase();
637         RootElement.Add(Result.ToXML(Testcase.Key));
638         Passed &= Result.Passed;
639     }
640     RootElement.SetAttributeValue("result", Passed ? "Passed" : "Failed"); ⏳152ms elapsed
641     return RootElement;
642 }
```

Annotation

Performance testing was based on the time to execute all testcases. This is because there is a large amount of opcodes in the testcases collectively, so can be used to measure the success criteria. A breakpoint was placed before and after the loop that loops through executing every testcase. This gave a very accurate upper bound of the time in-between the breakpoints given by visual studio.

Assembly emulation testing

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <all result="Passed">
3  <TestcaseResult name="adcadd" result="Passed">
4  <CheckpointResult Tag="Imm A">
5  <SubCheckpointResult result="Passed">
6  <Expected>$AL=0x10</Expected>
7  <Found>$AL=0x10</Found>
8  </SubCheckpointResult>
9  <SubCheckpointResult result="Passed">
10 <Expected>$AX=0x2010</Expected>
11 <Found>$AX=0x2010</Found>
12 </SubCheckpointResult>
13 <SubCheckpointResult result="Passed">
14 <Expected>$EAX=0x40032010</Expected>
15 <Found>$EAX=0x40032010</Found>
16 </SubCheckpointResult>
17 </CheckpointResult>
18 <CheckpointResult Tag="MI">
19 <SubCheckpointResult result="Passed">
20 <Expected>$BL=0x10</Expected>
21 <Found>$BL=0x10</Found>
22 </SubCheckpointResult>
23 <SubCheckpointResult result="Passed">
24 <Expected>$BX=0x2010</Expected>
25 <Found>$BX=0x2010</Found>
26 </SubCheckpointResult>
27 <SubCheckpointResult result="Passed">
28 <Expected>$RBX=0x40032010</Expected>
29 <Found>$RBX=0x40032010</Found>
30 </SubCheckpointResult>
31 </CheckpointResult>
32 <CheckpointResult Tag="Sxt b">
33 <SubCheckpointResult result="Passed">
34 <Expected>$DX=0xFFFF0</Expected>
35 <Found>$DX=0xFFFF0</Found>
36 </SubCheckpointResult>
37 <SubCheckpointResult result="Passed">
38 <Expected>$ECX=0xFFFFFFF2</Expected>
39 <Found>$ECX=0xFFFFFFF2</Found>
40 </SubCheckpointResult>
41 </CheckpointResult>
42 <CheckpointResult Tag="RM">
43 <SubCheckpointResult result="Passed">
44 <Expected>[RSP]={10}</Expected>
45 <Found>[RSP]={10}</Found>
46 </SubCheckpointResult>
47 <SubCheckpointResult result="Passed">
48 <Expected>[RSP+0x1]={0, 20}</Expected>
49 <Found>[RSP+0x1]={0, 20}</Found>
50 </SubCheckpointResult>
51 <SubCheckpointResult result="Passed">
52 <Expected>[RSP+0x3]={0, 0, 0, 30}</Expected>
53 <Found>[RSP+0x3]={0, 0, 0, 30}</Found>
54 </SubCheckpointResult>
55 </CheckpointResult>
56 <CheckpointResult Tag="MR">
57 <SubCheckpointResult result="Passed">
58 <Expected>$AL=0x20</Expected>
59 <Found>$AL=0x20</Found>
60 </SubCheckpointResult>
61 <SubCheckpointResult result="Passed">
62 <Expected>$BX=0x4000</Expected>
63 <Found>$BX=0x4000</Found>
64 </SubCheckpointResult>
65 <SubCheckpointResult result="Passed">
66 <Expected>$ECX=0x60000000</Expected>
67 <Found>$ECX=0x60000000</Found>
68 </SubCheckpointResult>
69 <SubCheckpointResult result="Passed">
70 <Expected>$RDX=0x30000000</Expected>
71 <Found>$RDX=0x30000000</Found>
72 </SubCheckpointResult>
73 </CheckpointResult>
74 <CheckpointResult Tag="Reg to reg">
75 <SubCheckpointResult result="Passed">
76 <Expected>$AL=0x11</Expected>
```

```
77     <Found>$AL=0x11</Found>
78   </SubCheckpointResult>
79   <SubCheckpointResult result="Passed">
80     <Expected>$BX=0x2211</Expected>
81     <Found>$BX=0x2211</Found>
82   </SubCheckpointResult>
83   <SubCheckpointResult result="Passed">
84     <Expected>$ECX=0x33332211</Expected>
85     <Found>$ECX=0x33332211</Found>
86   </SubCheckpointResult>
87 </CheckpointResult>
88 <CheckpointResult Tag="Overflow">
89   <SubCheckpointResult result="Passed">
90     <Expected>OF$FAFPF</Expected>
91     <Found>OF$FAFPF</Found>
92   </SubCheckpointResult>
93 </CheckpointResult>
94 <CheckpointResult Tag="OF CF">
95   <SubCheckpointResult result="Passed">
96     <Expected>$RAX=0x1</Expected>
97     <Found>$RAX=0x1</Found>
98   </SubCheckpointResult>
99   <SubCheckpointResult result="Passed">
100    <Expected>CFOF</Expected>
101    <Found>CFOF</Found>
102  </SubCheckpointResult>
103 </CheckpointResult>
104 <CheckpointResult Tag="ADC">
105   <SubCheckpointResult result="Passed">
106     <Expected>$BX=0x1</Expected>
107     <Found>$BX=0x1</Found>
108   </SubCheckpointResult>
109   <SubCheckpointResult result="Passed">
110     <Expected></Expected>
111     <Found></Found>
112   </SubCheckpointResult>
113 </CheckpointResult>
114 </TestcaseResult>
115 <TestcaseResult name="bit shifts" result="Passed">
116   <CheckpointResult Tag="Shift left">
117     <SubCheckpointResult result="Passed">
118       <Expected>$BL=0xFE</Expected>
119       <Found>$BL=0xFE</Found>
120     </SubCheckpointResult>
121     <SubCheckpointResult result="Passed">
122       <Expected>$DX=0xFF00</Expected>
123       <Found>$DX=0xFF00</Found>
124     </SubCheckpointResult>
125     <SubCheckpointResult result="Passed">
126       <Expected>[RSP]={0, 0, 0, 0, 0, 0, 0, 80}</Expected>
127       <Found>[RSP]={0, 0, 0, 0, 0, 0, 0, 80}</Found>
128     </SubCheckpointResult>
129     <SubCheckpointResult result="Passed">
130       <Expected>CFSFPF</Expected>
131       <Found>CFSFPF</Found>
132     </SubCheckpointResult>
133 </CheckpointResult>
134 <CheckpointResult Tag="SHR OF">
135   <SubCheckpointResult result="Passed">
136     <Expected>$BL=0x7F</Expected>
137     <Found>$BL=0x7F</Found>
138   </SubCheckpointResult>
139   <SubCheckpointResult result="Passed">
140     <Expected>CFOF</Expected>
141     <Found>CFOF</Found>
142   </SubCheckpointResult>
143 </CheckpointResult>
144 <CheckpointResult Tag="SHR NO OF">
145   <SubCheckpointResult result="Passed">
146     <Expected>$DL=0x0</Expected>
147     <Found>$DL=0x0</Found>
148   </SubCheckpointResult>
149   <SubCheckpointResult result="Passed">
150     <Expected>CF</Expected>
151     <Found>CF</Found>
```

```
152 </SubCheckpointResult>
153 </CheckpointResult>
154 <CheckpointResult Tag="SHR">
155   <SubCheckpointResult result="Passed">
156     <Expected>[RSP]={1F, 0, 0, 0, 0, 0, 0}</Expected>
157     <Found>[RSP]={1F, 0, 0, 0, 0, 0, 0}</Found>
158   </SubCheckpointResult>
159   <SubCheckpointResult result="Passed">
160     <Expected>CF</Expected>
161     <Found>CF</Found>
162   </SubCheckpointResult>
163 </CheckpointResult>
164 <CheckpointResult Tag="SAR NO OF">
165   <SubCheckpointResult result="Passed">
166     <Expected>$BL=0x3F</Expected>
167     <Found>$BL=0x3F</Found>
168   </SubCheckpointResult>
169   <SubCheckpointResult result="Passed">
170     <Expected>CF</Expected>
171     <Found>CF</Found>
172   </SubCheckpointResult>
173 </CheckpointResult>
174 <CheckpointResult Tag="SAR">
175   <SubCheckpointResult result="Passed">
176     <Expected>$DL=0x0</Expected>
177     <Found>$DL=0x0</Found>
178   </SubCheckpointResult>
179   <SubCheckpointResult result="Passed">
180     <Expected>[RSP]={F, 0, 0, 0, 0, 0, 0}</Expected>
181     <Found>[RSP]={F, 0, 0, 0, 0, 0, 0}</Found>
182   </SubCheckpointResult>
183   <SubCheckpointResult result="Passed">
184     <Expected>CF</Expected>
185     <Found>CF</Found>
186   </SubCheckpointResult>
187 </CheckpointResult>
188 </TestcaseResult>
189 <TestcaseResult name="cbw" result="Passed">
190   <CheckpointResult Tag="CBW">
191     <SubCheckpointResult result="Passed">
192       <Expected>$AX=0xFF80</Expected>
193       <Found>$AX=0xFF80</Found>
194     </SubCheckpointResult>
195   </CheckpointResult>
196   <CheckpointResult Tag="CWDE">
197     <SubCheckpointResult result="Passed">
198       <Expected>$EAX=0xFFFF8080</Expected>
199       <Found>$EAX=0xFFFF8080</Found>
200     </SubCheckpointResult>
201   </CheckpointResult>
202   <CheckpointResult Tag="CDQE">
203     <SubCheckpointResult result="Passed">
204       <Expected>$RAX=0xFFFFFFFF80008080</Expected>
205       <Found>$RAX=0xFFFFFFFF80008080</Found>
206     </SubCheckpointResult>
207   </CheckpointResult>
208 </TestcaseResult>
209 <TestcaseResult name="imul" result="Passed">
210   <CheckpointResult Tag="Byte">
211     <SubCheckpointResult result="Passed">
212       <Expected>$AX=0x22</Expected>
213       <Found>$AX=0x22</Found>
214     </SubCheckpointResult>
215     <SubCheckpointResult result="Passed">
216       <Expected>$DX=0x0</Expected>
217       <Found>$DX=0x0</Found>
218     </SubCheckpointResult>
219     <SubCheckpointResult result="Passed">
220       <Expected></Expected>
221       <Found></Found>
222     </SubCheckpointResult>
223   </CheckpointResult>
224   <CheckpointResult Tag="Word">
225     <SubCheckpointResult result="Passed">
226       <Expected>$AX=0xC842</Expected>
227       <Found>$AX=0xC842</Found>
```

```
228 </SubCheckpointResult>
229 <SubCheckpointResult result="Passed">
230   <Expected>$DX=0x48A</Expected>
231   <Found>$DX=0x48A</Found>
232 </SubCheckpointResult>
233 <SubCheckpointResult result="Passed">
234   <Expected>CFOF</Expected>
235   <Found>CFOF</Found>
236 </SubCheckpointResult>
237 </CheckpointResult>
238 <CheckpointResult Tag="Dword from mem">
239   <SubCheckpointResult result="Passed">
240     <Expected>$EAX=0x8F5C2C63</Expected>
241     <Found>$EAX=0x8F5C2C63</Found>
242   </SubCheckpointResult>
243   <SubCheckpointResult result="Passed">
244     <Expected>$EDX=0xA3D6D36</Expected>
245     <Found>$EDX=0xA3D6D36</Found>
246   </SubCheckpointResult>
247   <SubCheckpointResult result="Passed">
248     <Expected>CFOF</Expected>
249     <Found>CFOF</Found>
250   </SubCheckpointResult>
251 </CheckpointResult>
252 <CheckpointResult Tag="Qword">
253   <SubCheckpointResult result="Passed">
254     <Expected>$RAX=0xFFFFFFFF333276BB</Expected>
255     <Found>$RAX=0xFFFFFFFF333276BB</Found>
256   </SubCheckpointResult>
257   <SubCheckpointResult result="Passed">
258     <Expected>$RDX=0x2</Expected>
259     <Found>$RDX=0x2</Found>
260   </SubCheckpointResult>
261   <SubCheckpointResult result="Passed">
262     <Expected>CFOF</Expected>
263     <Found>CFOF</Found>
264   </SubCheckpointResult>
265 </CheckpointResult>
266 <CheckpointResult Tag="Reg to reg">
267   <SubCheckpointResult result="Passed">
268     <Expected>$RBX=0xC842</Expected>
269     <Found>$RBX=0xC842</Found>
270   </SubCheckpointResult>
271   <SubCheckpointResult result="Passed">
272     <Expected>$RCX=0x8F5C2C63</Expected>
273     <Found>$RCX=0x8F5C2C63</Found>
274   </SubCheckpointResult>
275   <SubCheckpointResult result="Passed">
276     <Expected>$RDX=0x320FEDCBF259084</Expected>
277     <Found>$RDX=0x320FEDCBF259084</Found>
278   </SubCheckpointResult>
279 </CheckpointResult>
280 <CheckpointResult Tag="Reg*Imm to reg">
281   <SubCheckpointResult result="Passed">
282     <Expected>$RBX=0x6633</Expected>
283     <Found>$RBX=0x6633</Found>
284   </SubCheckpointResult>
285   <SubCheckpointResult result="Passed">
286     <Expected>$RCX=0x99996633</Expected>
287     <Found>$RCX=0x99996633</Found>
288   </SubCheckpointResult>
289   <SubCheckpointResult result="Passed">
290     <Expected>$RDX=0xCCCCCCCC99996633</Expected>
291     <Found>$RDX=0xCCCCCCCC99996633</Found>
292   </SubCheckpointResult>
293 </CheckpointResult>
294 </TestcaseResult>
295 <TestcaseResult name="jmpitestcmp" result="Passed">
296   <CheckpointResult Tag="Win">
297     <SubCheckpointResult result="Passed">
298       <Expected>$CL=0xFF</Expected>
299       <Found>$CL=0xFF</Found>
300     </SubCheckpointResult>
301     <SubCheckpointResult result="Passed">
```

```
302     <Expected>[RBP-0x1]={FF, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14,
303     <Found>[RBP-0x1]={FF, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15,
304   </SubCheckpointResult>
305   <SubCheckpointResult result="Passed">
306     <Expected>ZFPF</Expected>
307     <Found>ZFPF</Found>
308   </SubCheckpointResult>
309 </CheckpointResult>
310 </TestcaseResult>
311 <TestcaseResult name="lea" result="Passed">
312   <CheckpointResult Tag="Win">
313     <SubCheckpointResult result="Passed">
314       <Expected>$RBX=0x1000917</Expected>
315       <Found>$RBX=0x1000917</Found>
316     </SubCheckpointResult>
317     <SubCheckpointResult result="Passed">
318       <Expected>$RCX=0x33332446</Expected>
319       <Found>$RCX=0x33332446</Found>
320     </SubCheckpointResult>
321     <SubCheckpointResult result="Passed">
322       <Expected>$RDX=0x8C</Expected>
323       <Found>$RDX=0x8C</Found>
324     </SubCheckpointResult>
325   </CheckpointResult>
326 </TestcaseResult>
327 <TestcaseResult name="mov" result="Passed">
328   <CheckpointResult Tag="Immediate">
329     <SubCheckpointResult result="Passed">
330       <Expected>$AL=0xF1</Expected>
331       <Found>$AL=0xF1</Found>
332     </SubCheckpointResult>
333     <SubCheckpointResult result="Passed">
334       <Expected>$BX=0xF2F2</Expected>
335       <Found>$BX=0xF2F2</Found>
336     </SubCheckpointResult>
337     <SubCheckpointResult result="Passed">
338       <Expected>$ECX=0xF333F3F3</Expected>
339       <Found>$ECX=0xF333F3F3</Found>
340     </SubCheckpointResult>
341     <SubCheckpointResult result="Passed">
342       <Expected>$RDX=0xF4444444F444F4F4</Expected>
343       <Found>$RDX=0xF4444444F444F4F4</Found>
344     </SubCheckpointResult>
345   </CheckpointResult>
346   <CheckpointResult Tag="Reg to mem">
347     <SubCheckpointResult result="Passed">
348       <Expected>[RSP]={F1}</Expected>
349       <Found>[RSP]={F1}</Found>
350     </SubCheckpointResult>
351     <SubCheckpointResult result="Passed">
352       <Expected>[RSP-0x2]={F2, F2}</Expected>
353       <Found>[RSP-0x2]={F2, F2}</Found>
354     </SubCheckpointResult>
355     <SubCheckpointResult result="Passed">
356       <Expected>[RSP-0x6]={F3, F3, 33, F3}</Expected>
357       <Found>[RSP-0x6]={F3, F3, 33, F3}</Found>
358     </SubCheckpointResult>
359     <SubCheckpointResult result="Passed">
360       <Expected>[RSP-0xE]={F4, F4, 44, F4, 44, 44, 44, F4}</Expected>
361       <Found>[RSP-0xE]={F4, F4, 44, F4, 44, 44, 44, F4}</Found>
362     </SubCheckpointResult>
363   </CheckpointResult>
364   <CheckpointResult Tag="Sign extension">
365     <SubCheckpointResult result="Passed">
366       <Expected>$RAX=0xFFFFFFFFFFFFFFF3</Expected>
367       <Found>$RAX=0xFFFFFFFFFFFFFFF3</Found>
368     </SubCheckpointResult>
369     <SubCheckpointResult result="Passed">
370       <Expected>$RBX=0xFFFFFFFFFFFFFFF3</Expected>
371       <Found>$RBX=0xFFFFFFFFFFFFFFF3</Found>
372     </SubCheckpointResult>
373     <SubCheckpointResult result="Passed">
374       <Expected>$ECX=0xFFFFFFFFF3</Expected>
375       <Found>$ECX=0xFFFFFFFFF3</Found>
376     </SubCheckpointResult>
377     <SubCheckpointResult result="Passed">
```

```
380      </SubCheckpointResult>
381    </CheckpointResult>
382    <CheckpointResult Tag="Zero extension">
383      <SubCheckpointResult result="Passed">
384        <Expected>$RAX=0xF3</Expected>
385        <Found>$RAX=0xF3</Found>
386      </SubCheckpointResult>
387      <SubCheckpointResult result="Passed">
388        <Expected>$RBX=0xF3</Expected>
389        <Found>$RBX=0xF3</Found>
390      </SubCheckpointResult>
391      <SubCheckpointResult result="Passed">
392        <Expected>$ECX=0xF3</Expected>
393        <Found>$ECX=0xF3</Found>
394      </SubCheckpointResult>
395      <SubCheckpointResult result="Passed">
396        <Expected>$EDX=0xFFFF3</Expected>
397        <Found>$EDX=0xFFFF3</Found>
398      </SubCheckpointResult>
399    </CheckpointResult>
400  </TestcaseResult>
401  <TestcaseResult name="mul" result="Passed">
402    <CheckpointResult Tag="Byte">
403      <SubCheckpointResult result="Passed">
404        <Expected>$AX=0x22</Expected>
405        <Found>$AX=0x22</Found>
406      </SubCheckpointResult>
407      <SubCheckpointResult result="Passed">
408        <Expected>$DX=0x0</Expected>
409        <Found>$DX=0x0</Found>
410      </SubCheckpointResult>
411      <SubCheckpointResult result="Passed">
412        <Expected></Expected>
413        <Found></Found>
414      </SubCheckpointResult>
415    </CheckpointResult>
416    <CheckpointResult Tag="Word">
417      <SubCheckpointResult result="Passed">
418        <Expected>$AX=0xC842</Expected>
419        <Found>$AX=0xC842</Found>
420      </SubCheckpointResult>
421      <SubCheckpointResult result="Passed">
422        <Expected>$DX=0x48A</Expected>
423        <Found>$DX=0x48A</Found>
424      </SubCheckpointResult>
425      <SubCheckpointResult result="Passed">
426        <Expected>CFOF</Expected>
427        <Found>CFOF</Found>
428      </SubCheckpointResult>
429    </CheckpointResult>
430    <CheckpointResult Tag="Dword">
431      <SubCheckpointResult result="Passed">
432        <Expected>$EAX=0x8F5C2C63</Expected>
433        <Found>$EAX=0x8F5C2C63</Found>
434      </SubCheckpointResult>
435      <SubCheckpointResult result="Passed">
436        <Expected>$EDX=0xA3D6D36</Expected>
437        <Found>$EDX=0xA3D6D36</Found>
438      </SubCheckpointResult>
439      <SubCheckpointResult result="Passed">
440        <Expected>CFOF</Expected>
441        <Found>CFOF</Found>
442      </SubCheckpointResult>
443    </CheckpointResult>
444    <CheckpointResult Tag="Qword from mem">
445      <SubCheckpointResult result="Passed">
446        <Expected>$RAX=0xFFFFFFFF333276BB</Expected>
447        <Found>$RAX=0xFFFFFFFF333276BB</Found>
448      </SubCheckpointResult>
449      <SubCheckpointResult result="Passed">
450        <Expected>$RDX=0x2</Expected>
451        <Found>$RDX=0x2</Found>
452      </SubCheckpointResult>
453      <SubCheckpointResult result="Passed">
454        <Expected>CFOF</Expected>
455        <Found>CFOF</Found>
```

```
456     </SubCheckpointResult>
457   </CheckpointResult>
458 </TestcaseResult>
459 <TestcaseResult name="negate" result="Passed">
460   <CheckpointResult Tag="Win">
461     <SubCheckpointResult result="Passed">
462       <Expected>$RAX=0x1</Expected>
463       <Found>$RAX=0x1</Found>
464     </SubCheckpointResult>
465     <SubCheckpointResult result="Passed">
466       <Expected>$RBX=0xFFFFFFFFFFFFFFE</Expected>
467       <Found>$RBX=0xFFFFFFFFFFFFFFE</Found>
468     </SubCheckpointResult>
469     <SubCheckpointResult result="Passed">
470       <Expected>CF</Expected>
471       <Found>CF</Found>
472     </SubCheckpointResult>
473   </CheckpointResult>
474 </TestcaseResult>
475 <TestcaseResult name="or" result="Passed">
476   <CheckpointResult Tag="Immediate A">
477     <SubCheckpointResult result="Passed">
478       <Expected>$RAX=0xFFFFFFFFFFFFFF</Expected>
479       <Found>$RAX=0xFFFFFFFFFFFFFF</Found>
480     </SubCheckpointResult>
481   </CheckpointResult>
482   <CheckpointResult Tag="Immediate">
483     <SubCheckpointResult result="Passed">
484       <Expected>$RBX=0xFFFFFFFFFFFFFF</Expected>
485       <Found>$RBX=0xFFFFFFFFFFFFFF</Found>
486     </SubCheckpointResult>
487   </CheckpointResult>
488   <CheckpointResult Tag="Immediate sign extended byte">
489     <SubCheckpointResult result="Passed">
490       <Expected>$AL=0x33</Expected>
491       <Found>$AL=0x33</Found>
492     </SubCheckpointResult>
493     <SubCheckpointResult result="Passed">
494       <Expected>$RBX=0x11113333</Expected>
495       <Found>$RBX=0x11113333</Found>
496     </SubCheckpointResult>
497     <SubCheckpointResult result="Passed">
498       <Expected>$RCX=0x1111111133333333</Expected>
499       <Found>$RCX=0x1111111133333333</Found>
500     </SubCheckpointResult>
501   </CheckpointResult>
502   <CheckpointResult Tag="Immediate to mem">
503     <SubCheckpointResult result="Passed">
504       <Expected>[RSP]={FF}</Expected>
505       <Found>[RSP]={FF}</Found>
506     </SubCheckpointResult>
507     <SubCheckpointResult result="Passed">
508       <Expected>[RSP+0x1]={FF, FF}</Expected>
509       <Found>[RSP+0x1]={FF, FF}</Found>
510     </SubCheckpointResult>
511     <SubCheckpointResult result="Passed">
512       <Expected>[RSP+0x4]={FF, FF, FF, FF}</Expected>
513       <Found>[RSP+0x4]={FF, FF, FF, FF}</Found>
514     </SubCheckpointResult>
515     <SubCheckpointResult result="Passed">
516       <Expected>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Expected>
517       <Found>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Found>
518     </SubCheckpointResult>
519   </CheckpointResult>
520   <CheckpointResult Tag="Reg to mem">
521     <SubCheckpointResult result="Passed">
522       <Expected>[RSP]={FF}</Expected>
523       <Found>[RSP]={FF}</Found>
524     </SubCheckpointResult>
525     <SubCheckpointResult result="Passed">
526       <Expected>[RSP+0x1]={FF, FF}</Expected>
527       <Found>[RSP+0x1]={FF, FF}</Found>
528     </SubCheckpointResult>
529     <SubCheckpointResult result="Passed">
```

```
530 <Expected>[RSP+0x4]={FF, FF, FF, FF}</Expected>
531 <Found>[RSP+0x4]={FF, FF, FF, FF}</Found>
532 </SubCheckpointResult>
533 <SubCheckpointResult result="Passed">
534   <Expected>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Expected>
535   <Found>[RSP+0x9]={FF, FF, FF, FF, FF, FF, FF, FF}</Found>
536 </SubCheckpointResult>
537 </CheckpointResult>
538 </TestcaseResult>
539 <TestcaseResult name="pushpop" result="Passed">
540   <CheckpointResult Tag="Pointer w/qw">
541     <SubCheckpointResult result="Passed">
542       <Expected>$RAX=0x2918B848905B6658</Expected>
543       <Found>$RAX=0x2918B848905B6658</Found>
544     </SubCheckpointResult>
545     <SubCheckpointResult result="Passed">
546       <Expected>$RBX=0x35FF</Expected>
547       <Found>$RBX=0x35FF</Found>
548     </SubCheckpointResult>
549     <SubCheckpointResult result="Passed">
550       <Expected>[RSP-0xA]={58, 66, 5B, 90, 48, B8, 18, 29, FF, 35}</Expected>
551       <Found>[RSP-0xA]={58, 66, 5B, 90, 48, B8, 18, 29, FF, 35}</Found>
552     </SubCheckpointResult>
553   </CheckpointResult>
554   <CheckpointResult Tag="Reg w/qw">
555     <SubCheckpointResult result="Passed">
556       <Expected>$RBX=0x4F4E4D4C3B3A2918</Expected>
557       <Found>$RBX=0x4F4E4D4C3B3A2918</Found>
558     </SubCheckpointResult>
559     <SubCheckpointResult result="Passed">
560       <Expected>$RCX=0x2918</Expected>
561       <Found>$RCX=0x2918</Found>
562     </SubCheckpointResult>
563   </CheckpointResult>
564   <CheckpointResult Tag="Imm b/w/dw">
565     <SubCheckpointResult result="Passed">
566       <Expected>$RAX=0x12345678</Expected>
567       <Found>$RAX=0x12345678</Found>
568     </SubCheckpointResult>
569     <SubCheckpointResult result="Passed">
570       <Expected>$BX=0x1234</Expected>
571       <Found>$BX=0x1234</Found>
572     </SubCheckpointResult>
573     <SubCheckpointResult result="Passed">
574       <Expected>$RCX=0xFFFFFFFFFFFFFF2</Expected>
575       <Found>$RCX=0xFFFFFFFFFFFFFF2</Found>
576     </SubCheckpointResult>
577     <SubCheckpointResult result="Passed">
578       <Expected>$RDX=0xAAAAAAAAAAAAAAA</Expected>
579       <Found>$RDX=0xAAAAAAAAAAAAAAA</Found>
580     </SubCheckpointResult>
581     <SubCheckpointResult result="Passed">
582       <Expected>$RSP=0x800000</Expected>
583       <Found>$RSP=0x800000</Found>
584     </SubCheckpointResult>
585     <SubCheckpointResult result="Passed">
586       <Expected>[RSP-0x8]={AA, AA, AA, AA, AA, AA, AA, AA}</Expected>
587       <Found>[RSP-0x8]={AA, AA, AA, AA, AA, AA, AA, AA}</Found>
588     </SubCheckpointResult>
589   </CheckpointResult>
590 </TestcaseResult>
591 <TestcaseResult name="retcall" result="Passed">
592   <CheckpointResult Tag="Win">
593     <SubCheckpointResult result="Passed">
594       <Expected>$RAX=0xA0</Expected>
595       <Found>$RAX=0xA0</Found>
596     </SubCheckpointResult>
597     <SubCheckpointResult result="Passed">
598       <Expected>ZPF</Expected>
599       <Found>ZPF</Found>
600     </SubCheckpointResult>
601   </CheckpointResult>
602 </TestcaseResult>
603 <TestcaseResult name="rotate" result="Passed">
604   <CheckpointResult Tag="rol rm, cl">
605     <SubCheckpointResult result="Passed">
```

```
606     <Expected>$RAX=0x4444444333322114</Expected>
607     <Found>$RAX=0x4444444333322114</Found>
608   </SubCheckpointResult>
609   <SubCheckpointResult result="Passed">
610     <Expected></Expected>
611     <Found></Found>
612   </SubCheckpointResult>
613 </CheckpointResult>
614 <CheckpointResult Tag="rol dw rm, imm">
615   <SubCheckpointResult result="Passed">
616     <Expected>$RBX=0x22113333</Expected>
617     <Found>$RBX=0x22113333</Found>
618   </SubCheckpointResult>
619   <SubCheckpointResult result="Passed">
620     <Expected>CF</Expected>
621     <Found>CF</Found>
622   </SubCheckpointResult>
623 </CheckpointResult>
624 <CheckpointResult Tag="rol rm, 1">
625   <SubCheckpointResult result="Passed">
626     <Expected>[RSP]={22, 44}</Expected>
627     <Found>[RSP]={22, 44}</Found>
628   </SubCheckpointResult>
629   <SubCheckpointResult result="Passed">
630     <Expected></Expected>
631     <Found></Found>
632   </SubCheckpointResult>
633 </CheckpointResult>
634 <CheckpointResult Tag="rol full circle upper b rm, cl">
635   <SubCheckpointResult result="Passed">
636     <Expected>$DX=0x2200</Expected>
637     <Found>$DX=0x2200</Found>
638   </SubCheckpointResult>
639   <SubCheckpointResult result="Passed">
640     <Expected></Expected>
641     <Found></Found>
642   </SubCheckpointResult>
643 </CheckpointResult>
644 <CheckpointResult Tag="rcl rm, cl">
645   <SubCheckpointResult result="Passed">
646     <Expected>$RAX=0x444444433332211A</Expected>
647     <Found>$RAX=0x444444433332211A</Found>
648   </SubCheckpointResult>
649 </CheckpointResult>
650 <CheckpointResult Tag="rcl dw rm, imm">
651   <SubCheckpointResult result="Passed">
652     <Expected>$RBX=0x22111999</Expected>
653     <Found>$RBX=0x22111999</Found>
654   </SubCheckpointResult>
655   <SubCheckpointResult result="Passed">
656     <Expected>CF</Expected>
657     <Found>CF</Found>
658   </SubCheckpointResult>
659 </CheckpointResult>
660 <CheckpointResult Tag="rcl rm, 1(CF SET BY PREV)">
661   <SubCheckpointResult result="Passed">
662     <Expected>[RSP]={23, 44}</Expected>
663     <Found>[RSP]={23, 44}</Found>
664   </SubCheckpointResult>
665   <SubCheckpointResult result="Passed">
666     <Expected></Expected>
667     <Found></Found>
668   </SubCheckpointResult>
669 </CheckpointResult>
670 <CheckpointResult Tag="rcl full circle upper b rm, cl">
671   <SubCheckpointResult result="Passed">
672     <Expected>$DX=0x2100</Expected>
673     <Found>$DX=0x2100</Found>
674   </SubCheckpointResult>
675   <SubCheckpointResult result="Passed">
676     <Expected></Expected>
677     <Found></Found>
678   </SubCheckpointResult>
679 </CheckpointResult>
680 <CheckpointResult Tag="ror rm, cl">
```

```
681 <SubCheckpointResult result="Passed">
682   <Expected>$RAX=0x1444444443333221</Expected>
683   <Found>$RAX=0x1444444443333221</Found>
684 </SubCheckpointResult>
685 <SubCheckpointResult result="Passed">
686   <Expected></Expected>
687   <Found></Found>
688 </SubCheckpointResult>
689 </CheckpointResult>
690 <CheckpointResult Tag="ror dw rm, imm">
691   <SubCheckpointResult result="Passed">
692     <Expected>$RBX=0x22113333</Expected>
693     <Found>$RBX=0x22113333</Found>
694   </SubCheckpointResult>
695 </CheckpointResult>
696 <CheckpointResult Tag="ror rm, 1">
697   <SubCheckpointResult result="Passed">
698     <Expected>[RSP]={8, 91}</Expected>
699     <Found>[RSP]={8, 91}</Found>
700   </SubCheckpointResult>
701   <SubCheckpointResult result="Passed">
702     <Expected>CF</Expected>
703     <Found>CF</Found>
704   </SubCheckpointResult>
705 </CheckpointResult>
706 <CheckpointResult Tag="ror full circle upper b rm, cl">
707   <SubCheckpointResult result="Passed">
708     <Expected>$DX=0x2200</Expected>
709     <Found>$DX=0x2200</Found>
710   </SubCheckpointResult>
711   <SubCheckpointResult result="Passed">
712     <Expected></Expected>
713     <Found></Found>
714   </SubCheckpointResult>
715 </CheckpointResult>
716 <CheckpointResult Tag="rcr rm, cl">
717   <SubCheckpointResult result="Passed">
718     <Expected>$RAX=0x3444444443333221</Expected>
719     <Found>$RAX=0x3444444443333221</Found>
720   </SubCheckpointResult>
721 </CheckpointResult>
722 <CheckpointResult Tag="rcr dw rm, imm">
723   <SubCheckpointResult result="Passed">
724     <Expected>$RBX=0x44223333</Expected>
725     <Found>$RBX=0x44223333</Found>
726   </SubCheckpointResult>
727 </CheckpointResult>
728 <CheckpointResult Tag="rcr rm, 1">
729   <SubCheckpointResult result="Passed">
730     <Expected>[RSP]={8, 11}</Expected>
731     <Found>[RSP]={8, 11}</Found>
732   </SubCheckpointResult>
733   <SubCheckpointResult result="Passed">
734     <Expected>CF</Expected>
735     <Found>CF</Found>
736   </SubCheckpointResult>
737 </CheckpointResult>
738 <CheckpointResult Tag="rcr upper b rm, cl(CF SET BY PREV)">
739   <SubCheckpointResult result="Passed">
740     <Expected>$DX=0x5200</Expected>
741     <Found>$DX=0x5200</Found>
742   </SubCheckpointResult>
743   <SubCheckpointResult result="Passed">
744     <Expected></Expected>
745     <Found></Found>
746   </SubCheckpointResult>
747 </CheckpointResult>
748 </TestcaseResult>
749 <TestcaseResult name="sbbsub" result="Passed">
750   <CheckpointResult Tag="A Imm">
751     <SubCheckpointResult result="Passed">
752       <Expected>$EAX=0x111256A8</Expected>
753       <Found>$EAX=0x111256A8</Found>
754     </SubCheckpointResult>
755   </CheckpointResult>
```

```
756 <CheckpointResult Tag="A_Imm_sxt_dw">
757   <SubCheckpointResult result="Passed">
758     <Expected>$RAX=0xF1F1F1F212222223</Expected>
759     <Found>$RAX=0xF1F1F1F212222223</Found>
760   </SubCheckpointResult>
761   <SubCheckpointResult result="Passed">
762     <Expected>CFSFAF</Expected>
763     <Found>CFSFAF</Found>
764   </SubCheckpointResult>
765 </CheckpointResult>
766 <CheckpointResult Tag="Reg_Imm">
767   <SubCheckpointResult result="Passed">
768     <Expected>$EBX=0x111256A8</Expected>
769     <Found>$EBX=0x111256A8</Found>
770   </SubCheckpointResult>
771 </CheckpointResult>
772 <CheckpointResult Tag="Reg_Imm_sxt_dw">
773   <SubCheckpointResult result="Passed">
774     <Expected>$RBX=0xF1F1F1F212222223</Expected>
775     <Found>$RBX=0xF1F1F1F212222223</Found>
776   </SubCheckpointResult>
777   <SubCheckpointResult result="Passed">
778     <Expected>CFSFAF</Expected>
779     <Found>CFSFAF</Found>
780   </SubCheckpointResult>
781 </CheckpointResult>
782 <CheckpointResult Tag="Imm_sxt_b">
783   <SubCheckpointResult result="Passed">
784     <Expected>$RAX=0x33332360</Expected>
785     <Found>$RAX=0x33332360</Found>
786   </SubCheckpointResult>
787   <SubCheckpointResult result="Passed">
788     <Expected>CFPF</Expected>
789     <Found>CFPF</Found>
790   </SubCheckpointResult>
791 </CheckpointResult>
792 <CheckpointResult Tag="Reg_to_rm">
793   <SubCheckpointResult result="Passed">
794     <Expected>$RAX=0x7777777866675689</Expected>
795     <Found>$RAX=0x7777777866675689</Found>
796   </SubCheckpointResult>
797   <SubCheckpointResult result="Passed">
798     <Expected>$RBX=0x888888886664422</Expected>
799     <Found>$RBX=0x888888886664422</Found>
800   </SubCheckpointResult>
801   <SubCheckpointResult result="Passed">
802     <Expected>[RSP]={50, B8, 86, 88, 87, 88, 88, 88}</Expected>
803     <Found>[RSP]={50, B8, 86, 88, 87, 88, 88, 88}</Found>
804   </SubCheckpointResult>
805   <SubCheckpointResult result="Passed">
806     <Expected>SFPF</Expected>
807     <Found>SFPF</Found>
808   </SubCheckpointResult>
809 </CheckpointResult>
810 </TestcaseResult>
811 <TestcaseResult name="setbyte" result="Passed">
812   <CheckpointResult Tag="Win">
813     <SubCheckpointResult result="Passed">
814       <Expected>[RSP]={1, 0}</Expected>
815       <Found>[RSP]={1, 0}</Found>
816     </SubCheckpointResult>
817   </CheckpointResult>
818 </TestcaseResult>
819 <TestcaseResult name="sib" result="Passed">
820   <CheckpointResult Tag="SIB">
821     <SubCheckpointResult result="Passed">
822       <Expected>$AX=0x100</Expected>
823       <Found>$AX=0x100</Found>
824     </SubCheckpointResult>
825     <SubCheckpointResult result="Passed">
826       <Expected>$BL=0xFF</Expected>
827       <Found>$BL=0xFF</Found>
828     </SubCheckpointResult>
829     <SubCheckpointResult result="Passed">
830       <Expected>$CL=0xFF</Expected>
```

```
831 <Found>$CL=0xFF</Found>
832 </SubCheckpointResult>
833 <SubCheckpointResult result="Passed">
834   <Expected>[0x150]={FF}</Expected>
835   <Found>[0x150]={FF}</Found>
836 </SubCheckpointResult>
837 <SubCheckpointResult result="Passed">
838   <Expected>[0x200]={FF}</Expected>
839   <Found>[0x200]={FF}</Found>
840 </SubCheckpointResult>
841 <SubCheckpointResult result="Passed">
842   <Expected>[0x300]={FF}</Expected>
843   <Found>[0x300]={FF}</Found>
844 </SubCheckpointResult>
845 <SubCheckpointResult result="Passed">
846   <Expected>[0x20000400]={FF}</Expected>
847   <Found>[0x20000400]={FF}</Found>
848 </SubCheckpointResult>
849 <SubCheckpointResult result="Passed">
850   <Expected>[0x8FF]={FF}</Expected>
851   <Found>[0x8FF]={FF}</Found>
852 </SubCheckpointResult>
853 <SubCheckpointResult result="Passed">
854   <Expected>[0x2FF]={FF}</Expected>
855   <Found>[0x2FF]={FF}</Found>
856 </SubCheckpointResult>
857 <SubCheckpointResult result="Passed">
858   <Expected>[RBP+0x210]={FF}</Expected>
859   <Found>[RBP+0x210]={FF}</Found>
860 </SubCheckpointResult>
861 </CheckpointResult>
862 </TestcaseResult>
863 <TestcaseResult name="string operations" result="Passed">
864   <CheckpointResult Tag="rep movs/repz cmps">
865     <SubCheckpointResult result="Passed">
866       <Expected>$AX=0x1FF</Expected>
867       <Found>$AX=0x1FF</Found>
868     </SubCheckpointResult>
869     <SubCheckpointResult result="Passed">
870       <Expected>$ECX=0x0</Expected>
871       <Found>$ECX=0x0</Found>
872     </SubCheckpointResult>
873     <SubCheckpointResult result="Passed">
874       <Expected>[RSP]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 3F, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 7E, 7F, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 8A, 8B, 8C, 8D, 8E, 8F, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 9A, 9B, 9C, 9D, BD, BE, BF, CO, C1, C2, C3, C4, C5, C6, C7, C8, C9, CA, CB, CC, CD, CE, CF, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, DA, DB, DC, FC, FD, FE, FF}</Expected>
875       <Found>[RSP]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 5E, 7F, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 8A, 8B, 8C, 8D, 8E, 8F, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 9A, 9B, 9C, 9D, BE, BF, CO, C1, C2, C3, C4, C5, C6, C7, C8, C9, CA, CB, CC, CD, CE, CF, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, DA, DB, DC, FD, FE, FF}</Found>
876     </SubCheckpointResult>
877     <SubCheckpointResult result="Passed">
878       <Expected>[RSP+0x200]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 3D, 3E, 3F, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 7C, 7D, 7E, 7F, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 8A, 8B, 8C, 8D, 8E, 8F, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 9A, 9B, 9C, 9D, BB, BC, BD, BE, BF, CO, C1, C2, C3, C4, C5, C6, C7, C8, C9, CA, CB, CC, CD, CE, CF, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, DA, DB, DC, FA, FB, FC, FD, FE, FF}</Expected>
879       <Found>[RSP+0x200]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 3E, 3F, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 7D, 7E, 7F, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 8A, 8B, 8C, 8D, 8E, 8F, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 9A, 9B, 9C, 9D, BC, BD, BE, BF, CO, C1, C2, C3, C4, C5, C6, C7, C8, C9, CA, CB, CC, CD, CE, CF, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, DA, DB, DC, FA, FB, FC, FD, FE, FF}</Found>
880     </SubCheckpointResult>
881     <SubCheckpointResult result="Passed">
882       <Expected>[RSP+0x300]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 3D, 3E, 3F, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 7C, 7D, 7E, 7F, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 8A, 8B, 8C, 8D, 8E, 8F, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 9A, 9B, 9C, 9D, BB, BC, BD, BE, BF, CO, C1, C2, C3, C4, C5, C6, C7, C8, C9, CA, CB, CC, CD, CE, CF, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, DA, DB, DC, FA, FB, FC, FD, FE, FF}</Expected>
883       <Found>[RSP+0x300]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 3E, 3F, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 5A, 5B, 5C, 5D, 7D, 7E, 7F, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 8A, 8B, 8C, 8D, 8E, 8F, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 9A, 9B, 9C, 9D, BC, BD, BE, BF, CO, C1, C2, C3, C4, C5, C6, C7, C8, C9, CA, CB, CC, CD, CE, CF, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9, DA, DB, DC, FA, FB, FC, FD, FE, FF}</Found>
```

```

884     </SubCheckpointResult>
885     <SubCheckpointResult result="Passed">
886         <Expected>ZPF</Expected>
887         <Found>ZPF</Found>
888     </SubCheckpointResult>
889   </CheckpointResult>
890   <CheckpointResult Tag="repnz cmps">
891     <SubCheckpointResult result="Passed">
892         <Expected>$EAX=0x64636261</Expected>
893         <Found>$EAX=0x64636261</Found>
894     </SubCheckpointResult>
895     <SubCheckpointResult result="Passed">
896         <Expected>[RDI-0x4]={61, 62, 63, 64, 69, 6A, 6B, 6C}</Expected>
897         <Found>[RDI-0x4]={61, 62, 63, 64, 69, 6A, 6B, 6C}</Found>
898     </SubCheckpointResult>
899     <SubCheckpointResult result="Passed">
900         <Expected>[RSI]={61, 62, 63, 64, 69, 6A, 6B, 6C}</Expected>
901         <Found>[RSI]={61, 62, 63, 64, 69, 6A, 6B, 6C}</Found>
902     </SubCheckpointResult>
903     <SubCheckpointResult result="Passed">
904         <Expected>ZPF</Expected>
905         <Found>ZPF</Found>
906     </SubCheckpointResult>
907   </CheckpointResult>
908 </TestcaseResult>
909 <TestcaseResult name="xchg" result="Passed">
910   <CheckpointResult Tag="Win">
911     <SubCheckpointResult result="Passed">
912         <Expected>$RAX=0x33332211</Expected>
913         <Found>$RAX=0x33332211</Found>
914     </SubCheckpointResult>
915     <SubCheckpointResult result="Passed">
916         <Expected>$BX=0x2211</Expected>
917         <Found>$BX=0x2211</Found>
918     </SubCheckpointResult>
919     <SubCheckpointResult result="Passed">
920         <Expected>$CX=0x0</Expected>
921         <Found>$CX=0x0</Found>
922     </SubCheckpointResult>
923     <SubCheckpointResult result="Passed">
924         <Expected>$RDX=0x4444444433334444</Expected>
925         <Found>$RDX=0x4444444433334444</Found>
926     </SubCheckpointResult>
927     <SubCheckpointResult result="Passed">
928         <Expected>[RSP]={0}</Expected>
929         <Found>[RSP]={0}</Found>
930     </SubCheckpointResult>
931   </CheckpointResult>
932 </TestcaseResult>
933 </all>

```

Annotation

The TestHandler was used to execute all the testcases and output them into an XML file showing the results. As shown in the figures above, each testcase has passed.

Question

```
public Dictionary<string, bool> GetFlags()
{
    // Create a copy of the flags. ShallowCopy() is much better performance than DeepCopy(). As
    // FlagSet is a struct, there is no reason not to. Use ShallowCopy() where possible.
    FlagSet VMFlags = Handle.ShallowCopy().Flags;

    // Return string representations of each flag state.
    return new Dictionary<string, bool>()
    {
        {"Carry", VMFlags.Carry      == FlagState.ON},
        {"Parity", VMFlags.Parity    == FlagState.ON},
        {"Auxiliary", VMFlags.Auxiliary == FlagState.ON},
        {"Zero", VMFlags.Zero       == FlagState.ON},
        {"Sign", VMFlags.Sign        == FlagState.ON},
        {"Overflow", VMFlags.Overflow == FlagState.ON},
        {"Direction", VMFlags.Direction == FlagState.ON},
    };
}
```

1

2

3

4

5

Question

```
public FlagSet Overlap(FlagSet input) => new FlagSet()
{
    // Return a new flag set based on $this and $input.
    // If a flag in $input is FlagState.UNDEFINED, the value of that flag in $this is used instead(which could also be FlagState.UNDEFINED)
    // Otherwise, that flag in the returned FlagSet is equal to the same flag in $input.
    // For example,
    // $input.Carry == FlagState.UNDEFINED
    // $this.Carry == FlagState.ON
    // New Carry = FlagState.ON
    // Another example,
    // $input.Overflow == FlagState.ON
    // $this.Overflow == FlagState.OFF
    // New Carry = FlagState.ON
    Carry = input.Carry == FlagState.UNDEFINED ? Carry : input.Carry,
    Auxiliary = input.Auxiliary == FlagState.UNDEFINED ? Auxiliary : input.Auxiliary,
    Overflow = input.Overflow == FlagState.UNDEFINED ? Overflow : input.Overflow,
    Zero = input.Zero == FlagState.UNDEFINED ? Zero : input.Zero,
    Sign = input.Sign == FlagState.UNDEFINED ? Sign : input.Sign,
    Parity = input.Parity == FlagState.UNDEFINED ? Parity : input.Parity,
    Direction = input.Direction == FlagState.UNDEFINED ? Direction : input.Direction,
    Interrupt = input.Interrupt == FlagState.UNDEFINED ? Interrupt : input.Interrupt
};
```

- 1 2 3 4 5
-

Question

```
// An easy way to fetch a register given its information.
// Returns a byte[] of length $cap consisting of the bottom $cap bytes of $Registers[$stable+$register].
// Access to upper byte registers such as AH can be done artificially by fetching AX then using $AX[1] as the upper byte.
get
{
    byte[] Output = new byte[(int)cap];
    for (int i = 0; i < Output.Length; i++)
    {
        Output[i] = Registers[(int)register + (int)table][i];
    }
    return Output;
}
set
{
    // If the length of the input byte array is greater than the capacity, data would be lost. Something wrong here would most likely be on my part.
    if (value.Length > ((int)cap))
    {
        throw new Exception("RegisterGroup.cs Attempt to overflow register in base class");
    }
    // Overwrite the bytes in $Registers[] with $value[]
    for (int i = 0; i < value.Length; i++)
    {
        Registers[(int)register + (int)table][i] = value[i];
    }
}
```

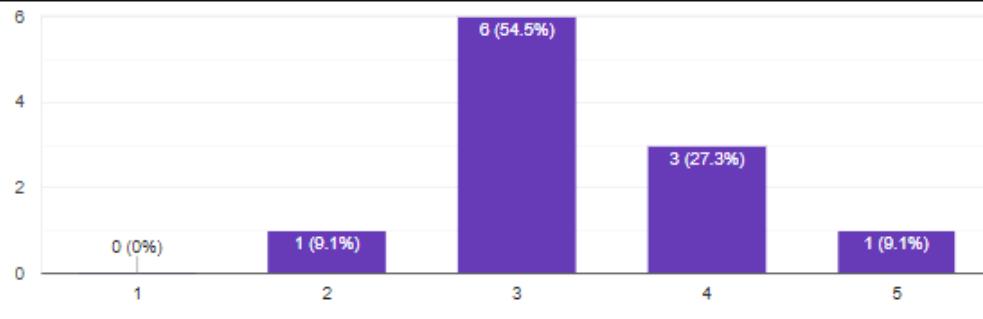
- 1 2 3 4 5
-

Question

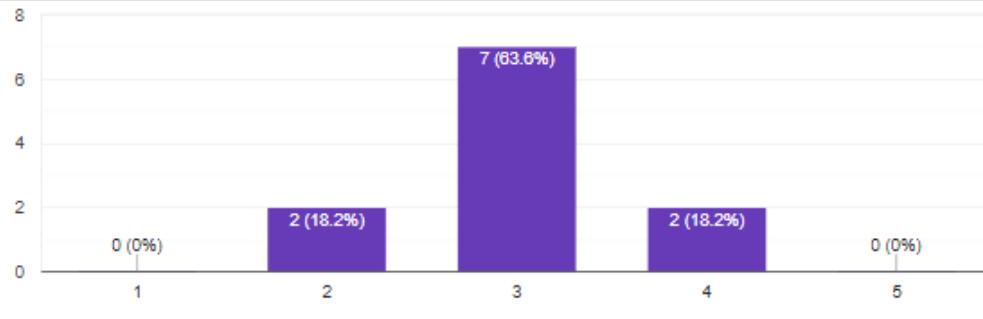
*** Linear scale

```
void Set(long address, byte value)
{
    // This is private because it does not affect AddressTable. This could be dangerous to an outside class, use the indexer or SetRange().
    // The address map doesn't need to be filled with 0s at initialisation. That would be massive waste of space. So instead, addresses are added to the dictionary as they are given values.
    // If the address is already in $AddressDict, its value is changed.
    // Setting a 0 byte to a location of the address is not allowed. This means that an address can be removed if a 0 byte is assigned to it, or never added to the address table at all.
    if (!AddressDict.ContainsKey(address))
    {
        if (value == 0x00)
        {
            AddressDict.Remove(address);
        }
        else
        {
            AddressDict[address] = value;
        }
    }
    else if (value != 0x00)
    {
        AddressDict[address] = value;
    }
}
```

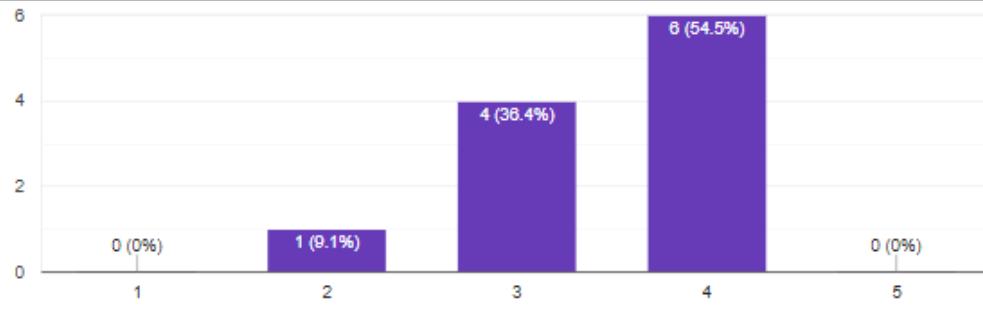
11 responses



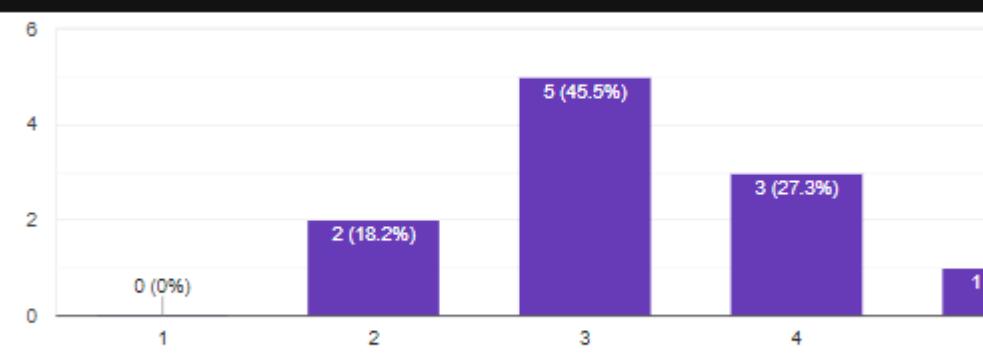
11 responses



11 responses



11 responses



Annotation

Classmates were given time to read a code extract and asked afterwards on a scale of 1 to 5(5 being positive) how much they could understand from the code. The above bar charts show the results of each question.

OO code review

Class: Hypervisor base

```
public abstract class HypervisorBase
{
    protected Handle Handle { get; private set; }
    public string HandleName { get => Handle.HandleName; }
    public int HandleID { get => Handle.HandleID; }
    public delegate void OnRunDelegate(Status input);
    public delegate void OnFlashDelegate(Context input);
    public event OnRunDelegate RunComplete = (input) => { };
    public event OnFlashDelegate Flash = (context) => { };
    public HypervisorBase(string inputName, Context inputContext, HandleParameters handleParameters = HandleParameters.NONE)
    {
        // Automatically create a new handle for the derived class. The derived class should rarely have to worry about handles,
        // but for the scope of advanced usage in the future, it is left as protected. I don't wish to restrict the modular
        // capabilities of the program.
        Handle = new Handle(inputName, inputContext, handleParameters);
    }
    protected virtual void OnFlash(Context input)
    {
        // To be implemented by derived classes.
    }
}
```

"There is good use of OO methods in this class, such as using virtual methods. This would allow classes to override inherited methods as required, so this class is definitely open for extension. This class also makes use of defined data-types(Context, MemorySpace), so it would be unlikely that this class needs to change since future classes will also be using these abstractions."

"This class is a good example of the open/closed principle. You have made use of events, which would allow each inheritor class to perform a different action when a condition is met, so functionality can be extended. Also, the class seems to perform very general procedures, such as Run(), FlashMemory(). These wouldn't need to change in the future since they clearly fulfill their purpose, so would have no reason to change, hence would not need to be modified."

Class: Opcode base class

```
[Flags]
public enum OpcodeSettings
{
    // Default behaviour
    NONE = 0,
    // Force byte capacity on the opcode
    BYTEMODE = 1,
    // Nothing explicitly in this class, but can be tested for by derived classes.
    SIGNED = 2,
}
```

```
public abstract class Opcode : IMyOpcode
{
    private RegisterCapacity _capacity;
    protected RegisterCapacity Capacity{...}
    public readonly OpcodeSettings Settings;
    private readonly string Mnemonic;
    private readonly IMyDecoded Input;
    public Opcode(string opcodeMnemonic, IMyDecoded input, OpcodeSettings settings = OpcodeSettings.NONE, RegisterCapacity capacity = RegisterCapacity.NONE)
    {
        Mnemonic = opcodeMnemonic;
        Input = input;
        Settings = settings;

        // If no capacity was provided, use the SetRegCap() method to try and automatically infer it. This
        // work with some opcodes, but not others. See SetRegCap().
        Capacity = (opcodeCapacity == RegisterCapacity.NONE) ? SetRegCap() : opcodeCapacity;
    }
    // Fetch all operands
    protected List<byte[]> Fetch() => Input.Fetch();
    // Discriminative set, use the operands in the order that would be expected
    protected void Set(byte[] data) => Input.Set(data);
    // Indiscriminative set (see documentation under "Indiscriminative operations")
    protected void Set(byte[] data, int operandIndex) => ((DecodedCompound)Input).Set(data, operandIndex);
    // Inheritable execute method that is defined by all derived classes (hence abstract keyword)
    public abstract void Execute();
    public virtual List<string> Disassemble()
    {
        // Function purpose: Associate the mnemonic of the opcode with the
        // disassembly of the operands, forming a complete disassembled instruction
        // to be pipelined into the disassembly modules

        // Start with the mnemonic
        List<string> Output = new List<string>() { Mnemonic };

        // Add the disassembly of the input
        Output.AddRange(Input.Disassemble());

        return Output;
    }
}
```

"This class has clearly used some aspects of the open/closed principle, for example, the IMyDecoded interface is used as an input for some functions instead of using specific classes, so it would always be possible to extend the behaviour of the class for new input types in this respect. However, the OpcodeSettings enum will need to be modified in the future when new settings are added."

Interface segregation principle review

"There is clear use of the ISP in the opcode classes. This has meant that the string operations, which had particular methods required such as AdjustDI, could be separated from the opcode base class, such that every opcode that isn't a string operation does not have to depend on those methods. Likewise, methods such as "Set" in the base opcode class are not necessary for the string operation class as they handle the process differently.

There are surprisingly many cases where further use of ISP would be nonsensical/useless in the code, however in some places there is room for improvement of ISP usage in this code. For example, there could have been more usage in the hypervisor interface layers instead of a base class. This would reduce dependency on the functions provided in the class if one day they are deprecated and instead, an interface could be used that specifies the minimum requirements for a base class. This would allow more base classes to be implemented in the future."

Annotation

I asked an OO programmer I know if he could look through some of the project code and write a short review. This will be used to evaluate the use of ISP in the project for the ISP success criterion.

Dependency inversion principle review

Scope: ControlUnit

"I don't know specifically about the dependency inversion principle, however, your code in the ControlUnit class was very logically structured and had a clear process of operation. Despite some understandable workarounds, the execute method behaves very similar to how a real life process would think, at least algorithmically and without lower level details irrelevant to the code. As you suggested, I did not need to read any other code files to understand the class, although found myself doing so out of curiosity afterwards."

Scope: ModRM

"This class represented a systematic process of determining the properties of a ModRM byte. It was a very reasonable attempt of decoding the table of parameters(following an if-else chain structure). However, I think this class should have been more dependent on the abstraction than it is currently. For example, the DecodedSIB variable is a nullable object. Although it is private, it is still strange to see this as it could cause errors and exceptions if not handled properly, which may not be known to a future maintainer. I think this should at least be signposted in the code or the approach re-evaluated.

Annotation

I asked an assembly programmer I know to look at some of the code without looking at any other classes(exclusively a given class, one at a time). I then asked him to write a short paragraph about how he could understand the code and any potential issues he had with it.

Call stack analysis

The screenshot shows a debugger interface with several panes:

- File Debug ?** (top left)
- Registers** pane (right side): Shows registers RIP, RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14 with their current values.
- Disassembly** pane (left side): Shows assembly code starting at address 0x0000000000000000. The code includes MOV EAX, 0x7FFFFFFF, ADD EAX, 0x01, NOP, MOV EAX, 0x80000000, ADD EAX, 0x80000001, NOP, MOV EAX, 0xF000, MOV EBX, 0x0, ADD AX, 0x1000, ADC BX, 0x0, and NOP.
- Memory** pane (bottom left): Shows memory dump with columns 0-15 and rows 0-3. The first row contains B8 FF FF 7F 83 C0 01 90 B8 00 00 00 80 05 01.
- Flags** pane (bottom right): Shows flags Carry, Parity, Auxiliary, Zero, Sign, Overflow, Direction with their current values.
- Step Run Reset** buttons (bottom right): Buttons for program control.

An example program was loaded for testing purposes.

```

> debugger.Emulator.Opcodes.Mov.Execute() Processing layer
debugger.Emulator.ControlUnit.Execute(bool step = false) Processing layer
debugger.Emulator.ControlUnit.Handle.Run(bool step = false) Intermediary layer
debugger.Hypervisor.HypervisorBase.Run(bool Step = false) Intermediary layer
debugger.Hypervisor.HypervisorBase.RunAsync.AnonymousMethod_0() Intermediary layer
System.Threading.Tasks.Task<debugger.Emulator.Status> .InnerInvoke()
System.Threading.Tasks.Task.Execute()

```

Figure 1

```

debugger.Hypervisor.HypervisorBase.RunAsync(bool Step = false) Intermediary layer
debugger.MainForm.VMContinue(bool Step = false) Interface layer
debugger.MainForm.VMContinue_ButtonEvent(object sender = {Text = "Run"}, System.EventArgs e = {X = 0x00000010 Y = 0x0000000d Button = Left}) Interface layer
System.Windows.Forms.Control.OnClick(System.EventArgs e)

```

Figure 2

(As there the program is multithreaded, it follows that figure 2 was the first to be executed(from bottom to top))
This is the call stack in the mov opcode(which is the first instruction to be executed). This is the lowest the tree goes on the way down.The interface layer in this figure cannot be seen as it created a separate thread, and is instead in figure 2. There is no way to display them on the way up in visual studio, so have been listed manually.

1. (lowest) Mov.Execute()
2. debugger.Emulator.ControlUnit.Execute(bool step = false)
3. debugger.Emulator.ControlUnit.Handle.Run(bool step = false)
4. debugger.Hypervisor.HypervisorBase.Run(bool Step = false)
5. debugger.Hypervisor.HypervisorBase.RunAsync.AnonymousMethod_0()

Note that this is the same as the way down, but still shows that it is met both ways.

Adding breakpoints test

The figure consists of three vertically stacked screenshots of the Visual Studio Disassembly window. Each screenshot shows assembly code with memory addresses on the left and assembly instructions on the right. Red annotations are overlaid on each screenshot:

- Top Screenshot:** Shows a breakpoint at address 0x000000000000000E. A red arrow points from the text "Breakpoint" to the breakpoint marker in the assembly window.
- Middle Screenshot:** Shows the assembly code again. A red arrow points from the text "'Run' pressed" to the instruction at address 0x000000000000000E, which is labeled "+RIP".
- Bottom Screenshot:** Shows the assembly code after a click. A red arrow points from the text "Clicked again to remove" to the instruction at address 0x000000000000000E, which is now labeled "-RIP".

```

Disassembly
0x0000000000000000 <RIP      MOV EAX, 0xFFFFFFFF
0x0000000000000005              ADD EAX, 0x01
0x0000000000000008              NOP
0x0000000000000009              MOV EAX, 0x80000000
0x000000000000000E              ADD EAX, 0x80000001 Breakpoint

Disassembly
0x0000000000000000      MOV EAX, 0xFFFFFFFF
0x0000000000000005      ADD EAX, 0x01
0x0000000000000008      NOP
0x0000000000000009      MOV EAX, 0x80000000
0x000000000000000E +RIP    ADD EAX, 0x80000001 "Run" pressed

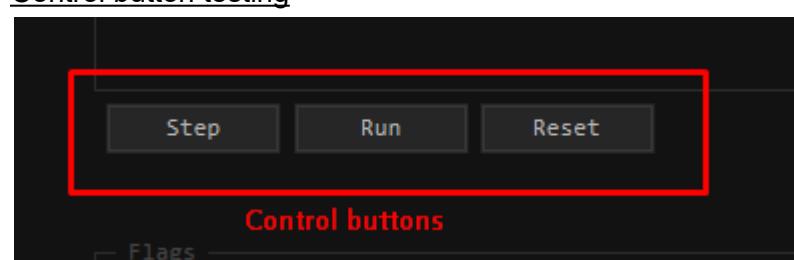
Disassembly
0x0000000000000000      MOV EAX, 0xFFFFFFFF
0x0000000000000005      ADD EAX, 0x01
0x0000000000000008      NOP
0x0000000000000009      MOV EAX, 0x80000000
0x000000000000000E -RIP   ADD EAX, 0x80000001 Clicked again to remove

```

Annotation

In example code, a breakpoint was added then run was pressed to execute between the breakpoint. This test shows that the process of inserting the breakpoint and “breaking” before the instruction at the breakpoint is executed.

Control button testing



Disassembly

```
0x0000000000000000 ←RIP      MOV EAX, 0xFFFFFFFF
0x0000000000000005          ADD EAX, 0x01      Initial state
0x0000000000000008          NOP
0x0000000000000009          MOV EAX, 0x80000000
0x000000000000000E          ADD EAX, 0x80000001
0x0000000000000013          NOP
0x0000000000000014          MOV EAX, 0xF000
0x0000000000000019          MOV EBX, 0x0
0x000000000000001E          ADD AX, 0x1000
0x0000000000000022          ADC BX, 0x0
0x0000000000000026          NOP
```

Initial state

Disassembly

```
0x0000000000000000          MOV EAX, 0xFFFFFFFF
0x0000000000000005+RIP     ADD EAX, 0x01      Step pressed
0x0000000000000008          NOP
0x0000000000000009          MOV EAX, 0x80000000
0x000000000000000E          ADD EAX, 0x80000001
0x0000000000000013          NOP
0x0000000000000014          MOV EAX, 0xF000
0x0000000000000019          MOV EBX, 0x0
0x000000000000001E          ADD AX, 0x1000
0x0000000000000022          ADC BX, 0x0
0x0000000000000026          NOP
```

Step pressed

Disassembly

```
0x0000000000000000          MOV EAX, 0xFFFFFFFF
0x0000000000000005          ADD EAX, 0x01
0x0000000000000008          NOP
0x0000000000000009          MOV EAX, 0x80000000
0x000000000000000E          ADD EAX, 0x80000001      Run pressed
0x0000000000000013          NOP
0x0000000000000014          MOV EAX, 0xF000
0x0000000000000019          MOV EBX, 0x0
0x000000000000001E          ADD AX, 0x1000
0x0000000000000022          ADC BX, 0x0
0x0000000000000026          NOP
```

(No RIP marker since all instructions have been executed, RIP points to next instruction to be executed)

```

Disassembly
0x0000000000000000 +RIP      MOV EAX, 0x7FFFFFFF
0x0000000000000005          ADD EAX, 0x01
0x0000000000000008          NOP
0x0000000000000009          MOV EAX, 0x80000000
0x000000000000000E          ADD EAX, 0x80000001
0x0000000000000013          NOP
0x0000000000000014          MOV EAX, 0xF000
0x0000000000000019          MOV EBX, 0x0
0x000000000000001E          ADD AX, 0x1000
0x0000000000000022          ADC BX, 0x0
0x0000000000000026          NOP

```

Reset pressed

Annotation

Above shows figures where the behaviour of each of the control buttons has been tested on example code. As shown, step executes one instruction, run executes all instructions(final breakpoint was removed), reset took the program back to the initial state.

Register resizing tests

Registers	
RIP	: 0x0000000000000009
RAX	: 0x0000000080000000
RCX	: 0x0000000000000000
RDX	: 0x0000000000000000
RBX	: 0x0000000000000000
RSP	: 0x0000000000800000
RBP	: 0x0000000000800000
RSI	: 0x0000000000000000
RDI	: 0x0000000000000000
R8	: 0x0000000000000000
R9	: 0x0000000000000000
R10	: 0x0000000000000000
R11	: 0x0000000000000000
R12	: 0x0000000000000000
R13	: 0x0000000000000000
R14	: 0x0000000000000000
Registers	
RIP	: 0x0000000000000009
AX	: 0x0000000000000000
CX	: 0x0000000000000000
DX	: 0x0000000000000000
BX	: 0x0000000000000000
SP	: 0x0000000000000000
BP	: 0x0000000000000000
SI	: 0x0000000000000000
DI	: 0x0000000000000000

Annotation

After double clicking the control, it turned to DWORD registers, then another double click turned it to show WORD registers. As shown in the first picture, RAX has 80 in upper DWORD byte. This is kept in the first double click showing the DWORD registers only. The upper QWORD bytes have been de-emphasised. When double clicking again, the bytes are gone entirely. Also, the empty bits have been de-emphasised slightly compared to bytes with significant bits(including significant zeros). The 0 in "10" is significant but the 0 in "011" is not). This is all as it was designed to do.

Robustness testing

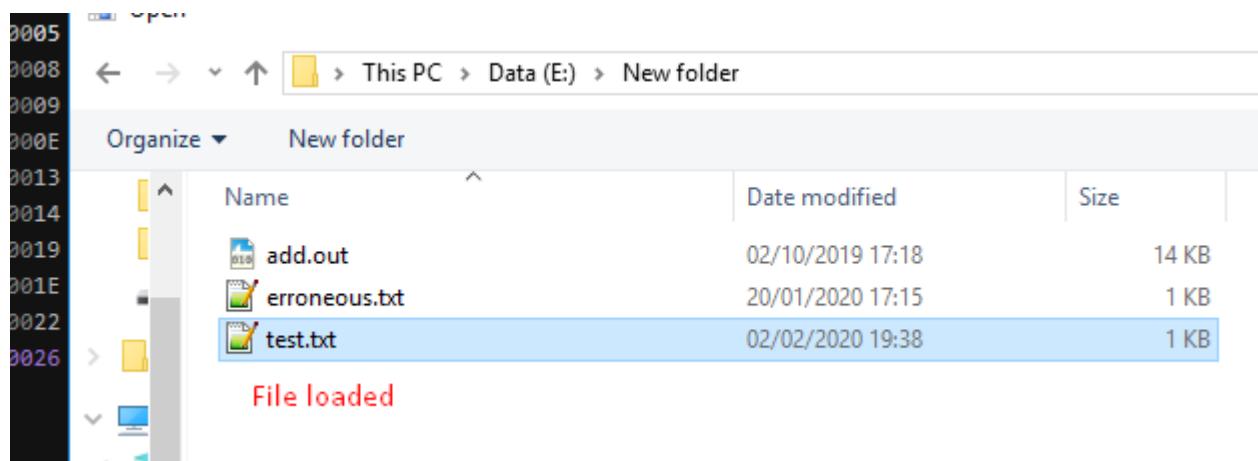
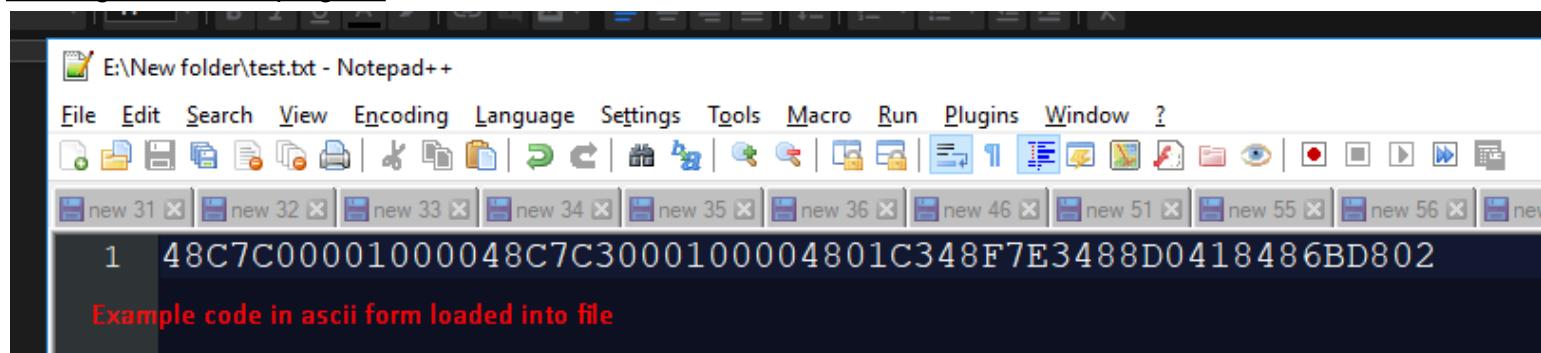
To test the robustness of this feature, clicks were spammed to try and cause an error. This would be the most likely way to cause an error as it is the only user input to this feature.

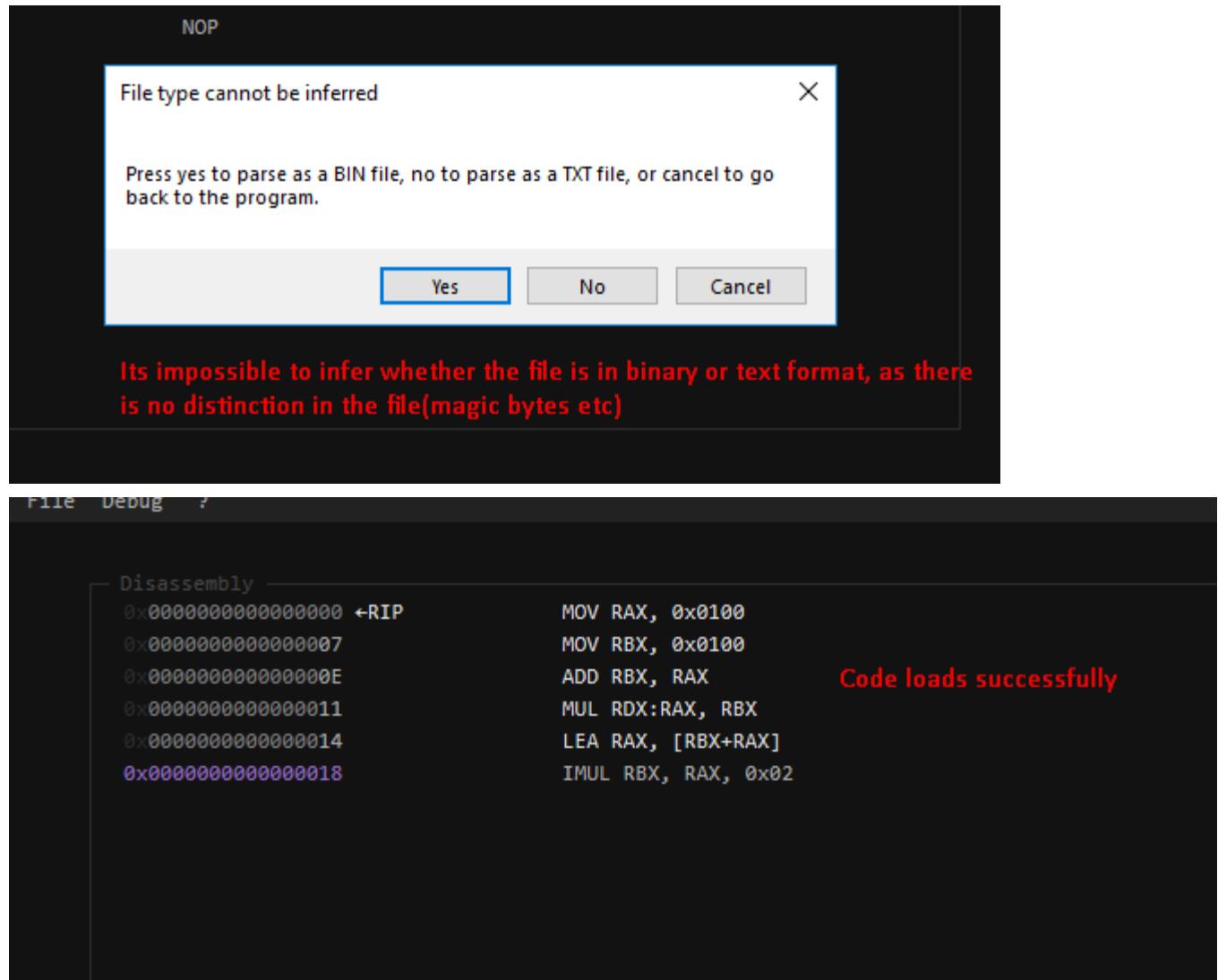
The image shows two side-by-side windows from a debugger or memory dump tool. Both windows are titled "Registers" and display a list of memory addresses and their corresponding values. The left window has a header "Registers" and lists addresses R8L through R14L. The right window also has a header "Registers" and lists addresses RIP, AL, CL, DL, BL, AH, CH, DH, BH, R8L, R9L, R10L, R11L, R12L, R13L, R14L, RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, and R14. All values are shown as 0x0000000000000000, indicating a consistent memory dump across both views.

Address	Value
RIP	0x000000000000000E
AL	0x0000000000000000
CL	0x0000000000000000
DL	0x0000000000000000
BL	0x0000000000000000
AH	0x0000000000000000
CH	0x0000000000000000
DH	0x0000000000000000
BH	0x0000000000000000
R8L	0x0000000000000000
R9L	0x0000000000000000
R10L	0x0000000000000000
R11L	0x0000000000000000
R12L	0x0000000000000000
R13L	0x0000000000000000
R14L	0x0000000000000000
RAX	0x0000000080000000
RCX	0x0000000000000000
RDX	0x0000000000000000
RBX	0x0000000000000000
RSP	0x0000000000800000
RBP	0x0000000000800000
RSI	0x0000000000000000
RDI	0x0000000000000000
R8	0x0000000000000000
R9	0x0000000000000000
R10	0x0000000000000000
R11	0x0000000000000000
R12	0x0000000000000000
R13	0x0000000000000000
R14	0x0000000000000000

Double clicking once bytes are displayed loops back around to qwords as designed, so this feature is robust against user input.

Loading text file into program

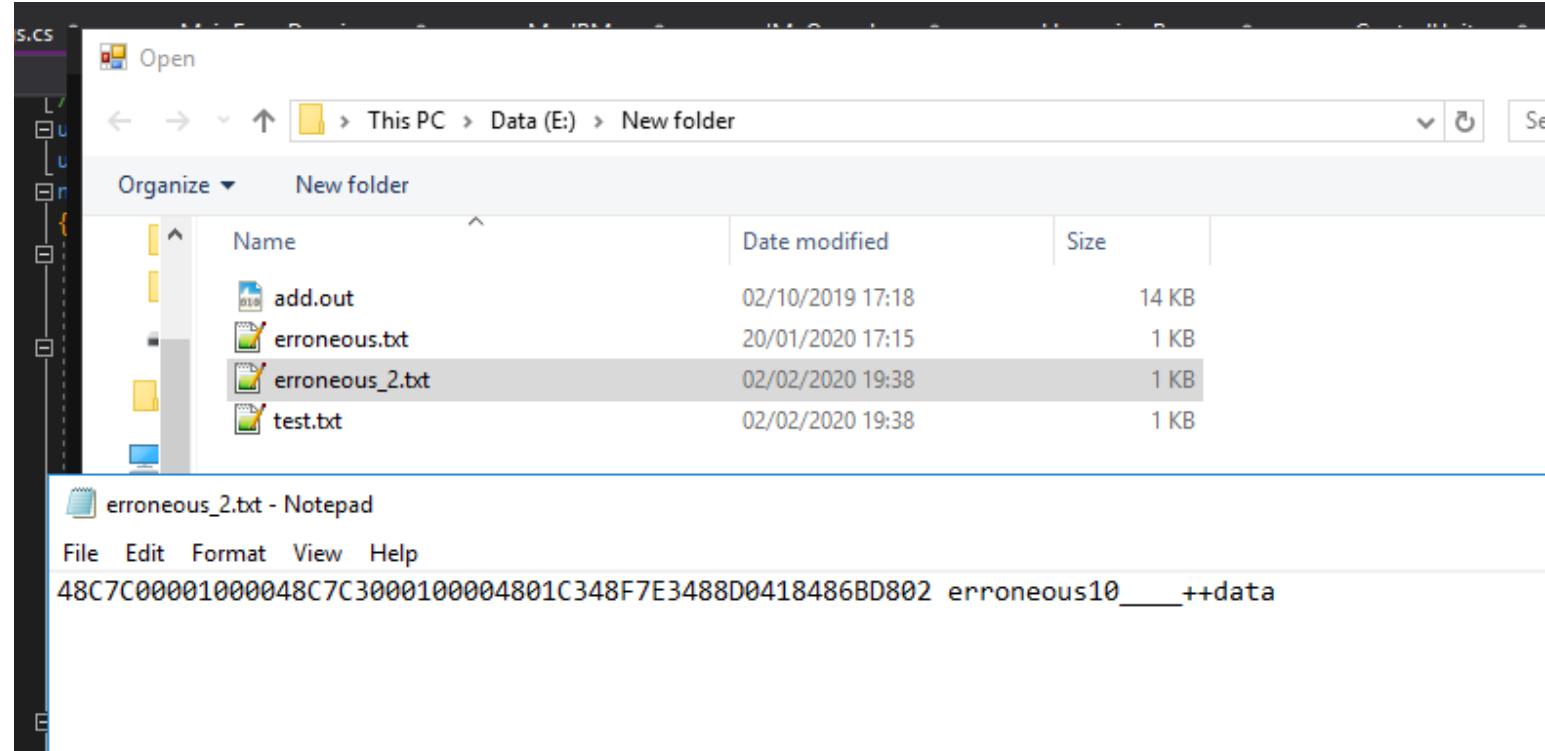




Annotation

The TXT file part of the program allows a text file to be interpreted as a binary file. This allows the user to quickly input their code without having to compile and link a binary.

Robustness testing



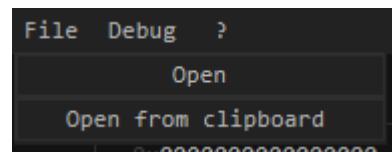
```

Disassembly
0x0000000000000000 +RIP      MOV RAX, 0x0100  Best effort attempt at
0x0000000000000007          MOV RBX, 0x0100  parsing erroneous data,
0x000000000000000E          ADD RBX, RAX  as any data is admissible in
0x00000000000000011         MUL RDX:RAX, RBX  assembly, it is impossible to
0x00000000000000014         LEA RAX, [RBX+RAX] determine what is erroneous
0x00000000000000018         IMUL RBX, RAX, 0x02
0x0000000000000001C         BAD INSTRUCTION
0x0000000000000001D         ADC DL, BL
0x0000000000000001F         BAD INSTRUCTION

```

Erroneous data was entered into a text file and then loaded into the program. Aside from the erroneous data appended, the instructions are the same as in the previous test, so can be seen for comparison. The program still attempts to parse the code as designed. This is because at assembly level, instructions could be data or data could be instructions, so it is possible that the input file also contained data that would be read by their code, so would not be appropriate to throw a validation error. Therefore, this feature is robust, there was no error caused by the input.

Loading from clipboard



```

File  Debug  ?
Open
Open from clipboard

File  Debug  ?

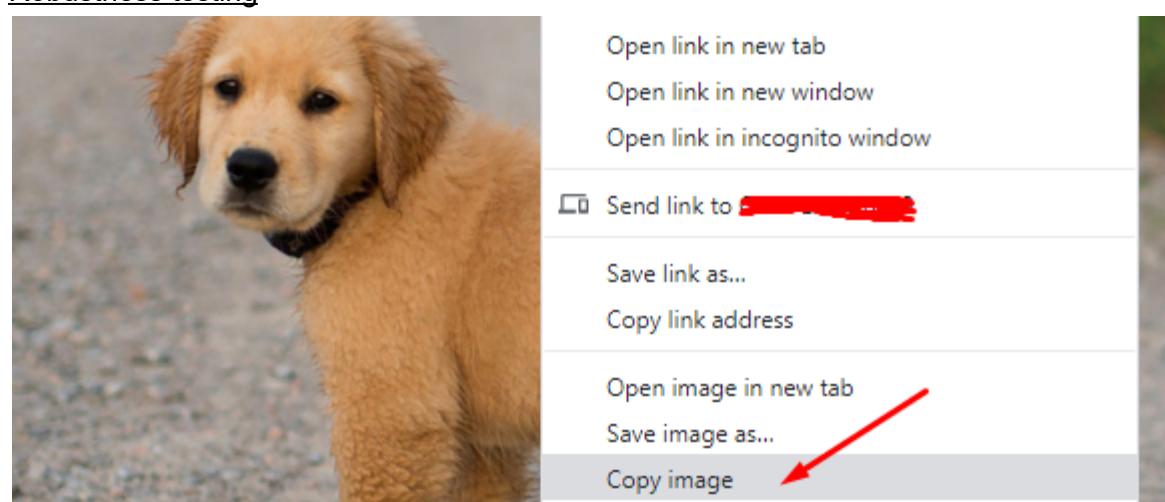
Disassembly
0x0000000000000000 +RIP      MOV RAX, 0x0100
0x0000000000000007          MOV RBX, 0x0100  Loaded
0x000000000000000E          ADD RBX, RAX  from clipboard
0x00000000000000011         MUL RDX:RAX, RBX
0x00000000000000014         LEA RAX, [RBX+RAX]
0x00000000000000018         IMUL RBX, RAX, 0x02

```

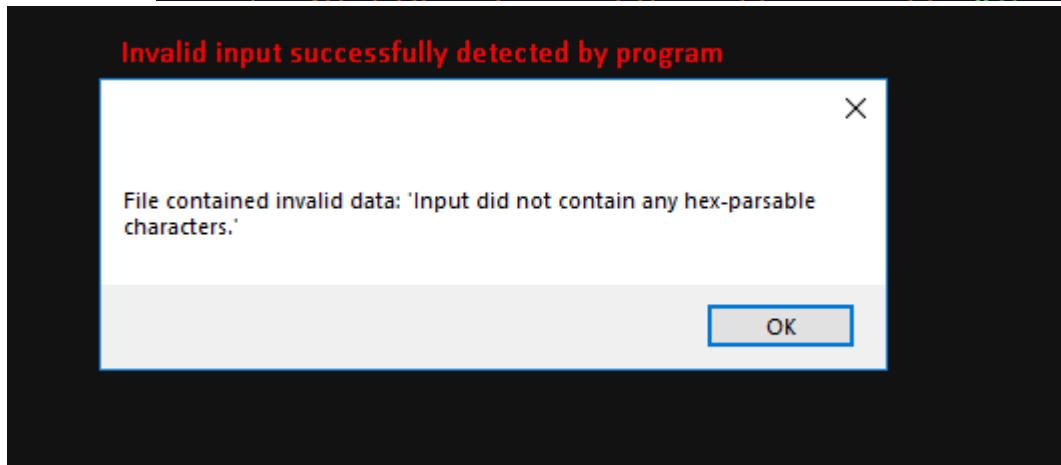
Annotation

ASCII bytes containing instructions were copied into the clipboard then loaded into the program.

Robustness testing



```
<?xml version="1.0" encoding="utf-8"?>
<all result="Passed">
  <TestcaseResult name="adcadd" result="Passed">
    <CheckpointResult Tag="Imm A">
      <SubCheckpointResult result="Passed">
        <Expected>$AL=0x10</Expected>
        <Found>$AL=0x10</Found>
      </SubCheckpointResult>
      <SubCheckpointResult result="Passed">
        <Expected>$AX=0x2010</Expected>
        <Found>$AX=0x2010</Found>
      </SubCheckpointResult>
    </CheckpointResult>
  </TestcaseResult>
</all>
```



As the picture contains windows metadata that cannot be parsed by the program at all, the input was blocked and a validation error was displayed. The program continued normally afterwards, hence was robust against erroneous input for this feature.

Evaluation

Success criteria breakdown

Outcome: Success

Criterion: The program should be able to accurately emulate a x86-64 processor from an outside perspective.

Figure from post development testing

How this has been met

The program uses the same procedures that a real processor would use and uses pseudocode and algorithms adapted from the official intel manual to ensure all behaviour is equivalent.

How this has been measured

The implementation of the TestHandler module has allowed a systematic and rigorous method of testing all parts of the code. This has allowed for all assembly features to be tested in one click. The test shows that each and every testcase has passed, hence all aspects of assembly in the program can be confirmed as functional and robust. As the testcase system was stated as the measure of success for this criterion, it can be said that this criterion has been met entirely.

Reflection

This criterion has allowed the project to have a purposeful and objective reason to be used. By emulating the x86-64 specification, which is most common in desktop computers, the stakeholders such as university students will have the most benefit from using the program as it is most likely to be the architecture they would learn about. It will also allow the project to be expanded on in the future for any more developments in the architecture itself.

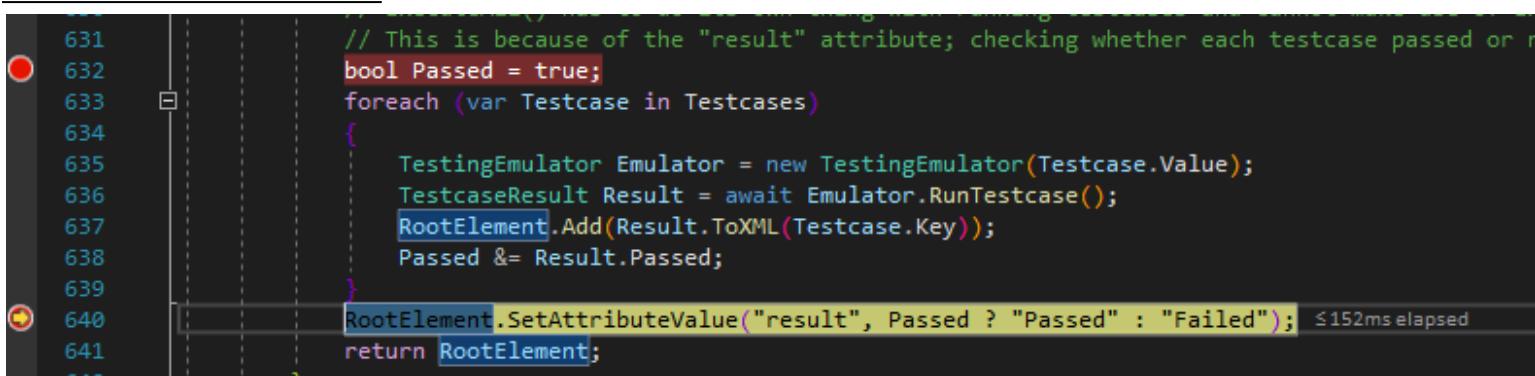
Further development plan

Due to the modular nature of the program and abstractions provided(such as the opcode api), maintaining this aspect will be very easy. However, there may be major changes required if there was a significant change, such as changing to a different architecture. This may operate in an entirely different way, such as AMD which has the instruction pointer point to the end of the current instruction rather than the start. This would require the ControlUnit class to be changed, which many other classes depend on. To improve this, instead of changing the ControlUnit class directly, a further layer of modularity could be added such that multiple types of ControlUnits can be created, such that the CU for x86-64 can be swapped out for the CU for AMD when an AMD file is loaded.

Criterion: The program should be able to execute instructions quickly

Outcome: Failure

How this has been measured



```
631 // This is because of the "result" attribute; checking whether each testcase passed or failed
632 bool Passed = true;
633 foreach (var Testcase in Testcases)
634 {
635     TestingEmulator Emulator = new TestingEmulator(Testcase.Value);
636     TestcaseResult Result = await Emulator.RunTestcase();
637    RootElement.Add(Result.ToXML(Testcase.Key));
638     Passed &= Result.Passed;
639 }
640 RootElement.SetAttributeValue("result", Passed ? "Passed" : "Failed"); ⏳152ms elapsed
641 return RootElement;
642 }
```

Figure from post development testing

This is an appropriate test to measure this criterion since there are a large amount of instructions from different files that will be executed. This shows that the execution took around 152 milliseconds. This criterion cannot be regarded as successful because the instructions executed on a processor complete on the order of microseconds. Large programs will execute slowly and the automated testing module neglected to benchmark the user computer as slower systems will experience severe latency, possibly rendering the program unusable for significant programs.

Reflection

This success criterion has been important and demonstrates the success of the project. Many GUI based programs can be very slow and crash often especially when performing complex tasks. By making use of extra threads and processor cores, this problem has been mitigated from the project. It could even be considered an upper bound since the breakpoint and other IDE related tasks slow down the program. This will allow the project to be more accessible to the stakeholders who own lower end computers such as laptops, which are very common for students at university or for programmers.

Further development plan

The use of threads has been embedded thoroughly into the program. The HypervisorBase class will automatically use threads to execute complex tasks on the CU without being told directly by the subclass. This means that when a new module is added to the code, it will automatically have performance optimised features. This will be important for maintainability as programmers will be able to focus on the essential features of the new code rather than have to spend a lot of time optimising it.

Criterion: The executable program and source code should be a great learning tool for the stakeholders.

Outcome: Partial success

How this has been measured

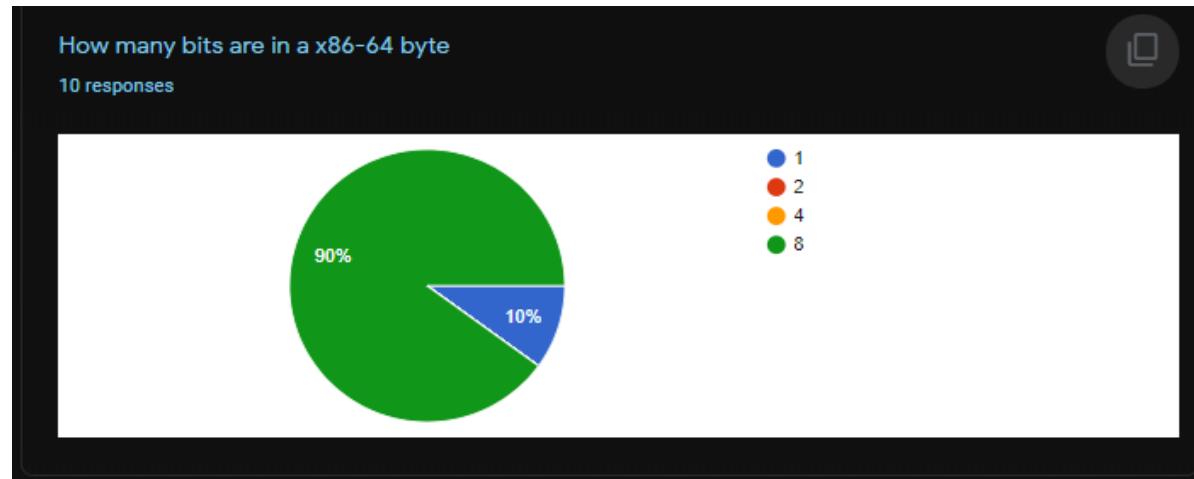


Figure from post development testing

I demonstrated the program to my classmates in lesson and gave them part of the code to read that had enough information in the comments to deduce the answers to this quiz. At this time, we had not covered any LMC or assembly in class, so it can be assumed that nobody knew any information prior. After giving them time to read, they took part in a short test on the content as planned in the analysis section. The average test score was 60% based on how questions got progressively harder. The scores were good which indicates some success, however, throughout the presentation I explained concepts and answered questions, suggesting that the program did not make these ideas self-evident, therefore this has been partially successful.

Further development plan

This can be improved in the future by having other maintainers review and progressively update the comments. This is because having multiple people review and develop and explanation will ensure that there are no assumptions or over-complications made in the explanation. A message will be left in the readme file of the repository to encourage potential maintainers and contributors to explain their modifications and updates. This will keep all of the code accessible rather than have the new parts of the code too complicated or explanations become deprecated. This may be hard for maintainers to do, for example if they don't speak english, or simply don't want to, and so could discourage them.

Reflection

I think this success criteria and its requirements have benefited the end project greatly. This is because it has allowed the program to effect a greater range of stakeholders as it not only has practical benefits, but also educational benefits. This will allow a stakeholder such as a student to read the source code and experiment with the program without having to have prior knowledge of assembly and no intentions to use the program to debug.

Criterion: Complex low level concepts will be abstracted into simple data structures and delegates

Outcome: Success

How this has been measured

```
using debugger.Util;
namespace debugger.Emulator.Opcodes
{
    public class Double : Opcode
    {
        public Double(DecodedTypes.IMyDecoded input, OpcodeSettings settings = OpcodeSettings.NONE) : base("DBL", input, settings)
        {
        }
        public override void Execute()
        {
            byte[] Result;
            FlagSet ResultFlags = Bitwise.Multiply(Fetch()[0], new byte[] { 2 }, false, 8, out Result);
            ControlUnit.SetFlags(ResultFlags);
            Set(Result);
        }
    }
}
```

An extract from post development testing

I tasked some of my classmates in a lesson to try to create their own opcode using the libraries included in the project. The figure shows one result. This opcode was designed to double the input. As shown in the image, the classmate had been able to use Set(), SetFlags(), and Bitwise.Multiply, which are 3 different functions from three different libraries. This shows that in around 30 minutes, they were able to understand the libraries enough to be able to use them in their own module. Therefore, it can be said that this success criterion has been met to a significant extent as they were successfully able to incorporate the abstractions into their code.

Reflection

This success criterion has ensured that future maintenance and reuse of program modules will be effective and as simple as possible. It will allow maintainers in the future to be able to quickly write their own modules and makes the project attractive to other maintainers since they won't have a hard time understanding the code and libraries.

Further development plan

This criterion has laid the foundations for future development. Improvements made in the future will be able to function from the apis provided and only needed to be expanded where needed as the core functionality is already implemented. This will allow for quicker development and more code can be reused throughout the program since the common subroutines such as multiply and add have already been coded.

Criterion: User interface must be intuitive.

Outcome: Partial success

To measure how intuitive the UI is, I used black box testing with a few of my classmates in lesson. I intentionally asked people who had not been asked to test my program before to ensure fair results. This test took place in the post development testing section, but will be discussed further here.

Usability testing

A classmate was used to test the usability of this feature. They were first briefed on what was meant by "Step" and "Run". It is fair to assume that the stakeholders described in the analysis section will already understand what is meant by these terms, and it is not hard to figure out by pressing them. The classmate was asked to first run the instructions using the Run button, then execute them a second time by stepping through without restarting the program(they had to figure out that they would need to use the reset button)

1. User opens program
2. User opens file(irrelevant to this test)
3. User presses run
4. User immediately pressed reset then stepped through again

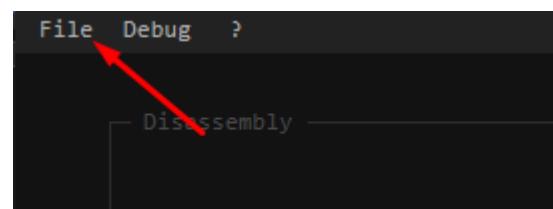
This shows that the "User interface must be intuitive." success criterion has been met as the user was able to figure out on their own how to use the interface.

An extract from post development testing

This test extract shows the positive impacts that are a result of how intuitive the user interface is. A user entirely new to the program was able to perform a task that involved interactions with the majority of core functionality. This can only be explained by having an intuitive UI, however, they did not figure this out immediately and took some thinking. This can be explained as the menu bar at the top of the window not following the conventional windows aero style, therefore this criterion is only partially successful.

Reflection

This success criterion has positive impacts on the stakeholders discussed in the analysis section. The UI took advantage of UI concepts that users will already know, for example, the file toolbar,



Since this is present in almost any program and used to open files, the testers in the post development testing were quickly able to catch on without any explanation. This will be very useful to stakeholders such as students who have not used assembly before, as they will not be familiar with how the program works. By keeping the interface simple, they will still be able to see and understand the effects of complex tasks, which will help them understand the process.

Further development plan

As the groundwork for the UI has already been set up, it will be very simple to expand to accommodate new features. For example, to add another header to the toolbar. This will make it easy for maintainers to continue keeping the UI as intuitive and as smooth as possible rather than making significant changes.

Criterion: User interface must not be excessive

Outcome: Success

How this has been met

The interface has been clearly split into 5 main different sections including the toolbar that each have their purpose written(covered by red in test).

How this has been measured

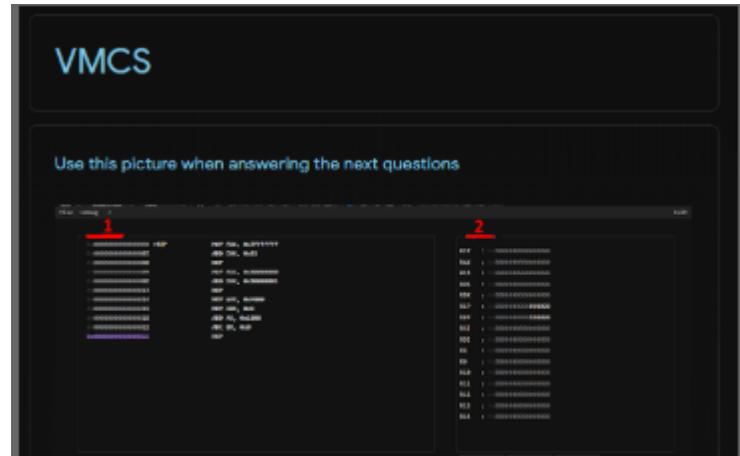


Figure from UI intuition test from post development testing

As planned in the analysis section, to test this, classmates were given a picture of the interface and asked to describe what each element of the interface displayed. This success criterion can be considered met entirely as there was an overwhelming majority of correct answers. Since one answer wrong would logically lead to another being wrong(since 4 options were presented), the 20% on each chart is not representative. Since each person tested had not seen too much of the UI in the latest version(a few had been used to test and review the alpha development versions), it can be said that the information displayed was not excessive. This is because they were able to quickly understand what each section displayed provided raw information displayed as the titles of each box were hidden.

Further development plan

In the future, the maintenance of this feature/criterion may be hard to keep up. This is because as new features are added over time, there will eventually be a point where there are too many features for the interface. This could be addressed by having two separate versions, one that has the basic features so can still meet the needs of stakeholders, and a more feature-heavy version for experienced users.

Criterion: User interface interaction will be kept minimal

Outcome: Successful

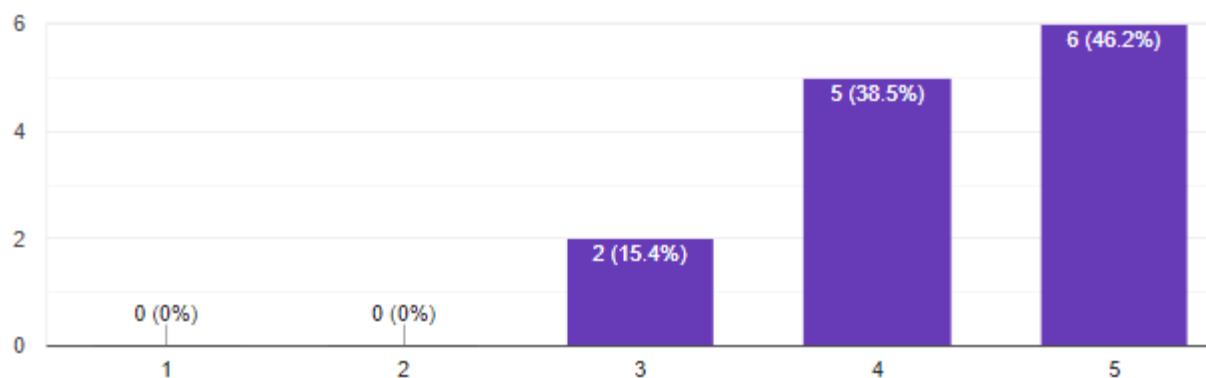
How this has been met

All of the user-active controls(buttons, toolbars) have been clearly placed and labelled in the interface. The dropdown menus are only used to hide large amounts of data(such as the dropdown displaying every testcase) where necessary.

How this has been measured

Rating

13 responses



Good positioning of GUIs. I like how the name of each section is discretely put around the border, clear to see when viewing the first time but not obstructive once you know what you're doing.

Looks great. Glad you were able to reduce the amount of information compared to the alternative software showcased earlier, such as removing empty lines from the memory viewer. The borders also add a really nice touch.

Figures from post development testing

In the testing to inform evaluation section, a test was conducted to survey the opinions of stakeholders about the UI. This shows that the stakeholders are happy with the state of the interface. This has further been shown by a selection of comments that specifically state that they thought the UI was minimal. The feedback is overwhelmingly positive, so this criterion is considered successful.

Further development plan

In the future, this will be maintained by ensuring future maintainers are aware of the principles and success criteria of this project. As suggested previously, it may be possible to create a separate version that has more advanced features, such that a non-minimalistic UI would not be a drawback to experienced users. There is still plenty of room acceptable to be used in the toolbar, where dropdowns containing new functions can be added.

Criterion: Core features will be easily accessible through the UI

Outcome: Successful

How this has been met

For core features, buttons have been placed on the main GUI form to allow easy use, such as stepping through a program, which would require the step button to be pressed many times, which is easier than having to select it through a dropdown.

How this has been measured

Throughout the testing to inform evaluation section, each test performed by a stakeholder has the number of clicks required to perform the relevant actions have been recorded. For example,

Usability testing

A classmate was instructed to “execute the mov testcase” with no further intervention. The events in the video are detailed as follows,

1. User opens program
2. User looks around for a few seconds then clicks on help button
3. User reads “execute a testcase” section of help
4. User goes to debug and executes mov testcase by search
(2 clicks)

Usability testing of feature

A classmate/stakeholder was asked to perform this task without any further intervention(a competent python programmer, no assembly knowledge). The events of the video recording are outlined here.

1. User opens program
2. User clicked file then open

3. User opened the file

(Once in program, 1 click to get to “open”)

Figures from post development testing

This evidence shows that the clicks required are very low. The target was two or less, which has clearly been met. There are some other tests that suggest otherwise, for example,

Usability testing

A classmate was asked to load code into the program, given the same website open(the assembly code was already entered by myself). They were not explicitly told about the clipboard feature.

1. User opens program
2. User hovers over file and immediately sees open from clipboard
3. User presses it(without code copied)
4. User goes back to window and copies the code
5. User presses open from clipboard again

4 clicks

Figure from post development testing

This is because the user clicked on the wrong button. This can be explained by the fact that they were new to the program, so did not fully understand the UI. As the initial targets were met, this criterion has been successful.

Further development plan

This can be improved in the future by providing more help and documentation features for users. For example, in the evidence, it was shown that a user made more clicks than optimal to use a core feature. This would not have happened if the user had been given a manual that explained where each function was, as they would have known what they needed to click. This could be continuously updated throughout future development to show how new features can be used.

Criterion: User interface will follow the Material.io material dark theme design specification

Outcome: Successful

How this has been met

The specifications from the official material.io website have been used to design the user interface(as shown in the design section). This includes the concept of layers, where less relevant material(such as the names of each interface element in the borders), has been dimmed using darker colours.

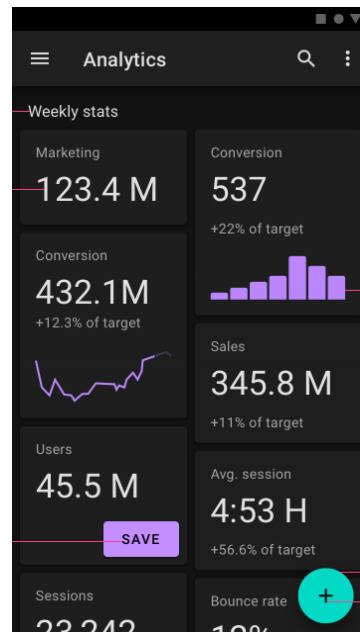
How this has been measured

In the testing to inform evaluation, users were asked about their thoughts on the UI. One included,

A great UI. Information is clearly sorted in a logical order and the buttons are placed in very natural positions. The dark theme makes the program very presentable and creates a good atmosphere. Features have clearly been highlighted appropriately, such as colouring breakpoints and dimming 00 bytes.

This shows that the stakeholders did like the material dark theme as there were particular lines, “the dark theme” that show they were specifically talking about it without any prompt. Moreover, in the entirety of the survey, there were no negative opinions on it, therefore the measurability of the criterion, which was that “It would be considered unsuccessful if users say that they would prefer a more serious style.”, has clearly been met.

Further development plan



It will be simple to maintain this criterion in the future. This is because the foundations such as the theme engine and custom controls have already been made. This means that when a new window/control needs to be added, it will be even easier to inherit from existing classes than to make a new class(because of the libraries). This criterion could be met further by introducing more of the concepts of material dark design. For example the example in the figure has rounded edges for the form controls, but in VMCS, the form controls were rectangular. This could be implemented by creating a new method in the base classes for controls that paints every control with a rounded border, so would apply to every control due to inheritance

Criterion: Single responsibility principle: Every module, class, and function will be designed to fit a specific requirement. A class should only have one reason to change[2]

Outcome: Partially successful

How this has been met

The project was split into four independent layers of abstraction: the interface layer, processing layer, intermediary layer, and utility library. Each of these layers has a specific task, for example, the processing layer handles all the emulation of assembly, whereas the interface layer handles user input and displaying information. This meant that the classes could be split categorically and reused appropriately. For example, the bitwise library has many functions that perform operations on byte arrays as opposed to integer values. These were only relevant to the processing layer, such that the library only had to be imported in the processing layer classes, such that the modules in other layers need not depend on the bitwise library as it is irrelevant to them.

How this has been measured

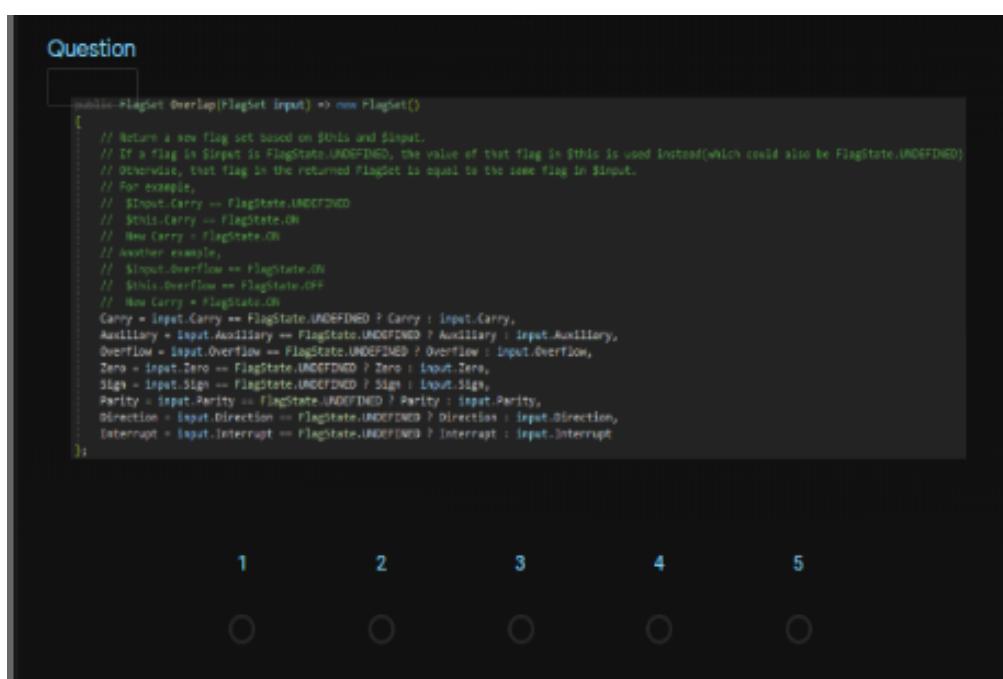


Figure from post development testing

Classmates were given independent extracts of the code and asked how much of each they could understand. This allowed me to see how easy it was to understand certain modules without reading the rest of the code as a module under the single responsibility principle should not be performing tasks outside of the ask it is designed to perform. The mean score given was a 3, which was "OK". This is as expected since it is very hard to understand code in a potentially new language from a new project in a short amount of time. It can still be considered a good score, but due to the aims of VMCS, it will be improved in the future. Therefore since the mean score was below desirable, but not terrible, this criterion can be considered partially met.

Future development plan

This will be improved in the future by ensuring there is enough context in the comments to understand the code. For example, in the extracts given in the survey, there were many places where C#-specific syntax was used but not properly explained. For example, the get-set accessors. These would have been entirely new to a python programmer, so they would not have understood that part of the code. This would be a problem to stakeholders who new another high level language, but not C#. This can be gradually improved through maintenance and frequent stakeholder review as it was hard to tell throughout development whether each code section met this criterion since

there was a large amount of code and a small amount of stakeholders that I had access to. This could also be improved in the future by looking for stakeholders online and performing black box testing, such as posting in forums and asking what people there thought of the code. This would allow for more accurate results when tested statistically as if I were to keep using my classmates for tests, they would eventually become too familiar with the code.

Criterion: Open/Closed principle. Software entities should be open for extension but closed for modification[3].

Outcome: Partially successful

How this has been met

Base classes have been inherited throughout the code to provide the core function of that particular object, then also allows the subclass the create/override its own methods to provide unique functionality. For example, the HypervisorBase class handles the creation of a handle and context for a class. This is a common procedure required by all hypervisors, such that all hypervisors inherit from this base class and use its functionality(open for extension), such that new features/hypervisors can be implemented without having to change the code in the base class that other classes depend on(closed for modification).

How this has been measured

Class: Hypervisor base

```
public abstract class HypervisorBase
{
    protected Handle Handle { get; private set; }
    public string HandleName { get => Handle.HandleName; }
    public int HandleID { get => Handle.HandleID; }
    public delegate void OnRunDelegate(Status input);
    public delegate void OnFlashDelegate(Context input);
    public event OnRunDelegate RunComplete = (input) => { };
    public event OnFlashDelegate Flash = (context) => { };
    public HypervisorBase(string inputName, Context inputContext, HandleParameters handleParameters = HandleParameters.NONE)
    {
        // Automatically create a new handle for the derived class. The derived class should rarely have to worry about handles,
        // but for the scope of advanced usage in the future, it is left as protected. I don't wish to restrict the modular
        // capabilities of the program.
        Handle = new Handle(inputName, inputContext, handleParameters);
    }
    protected virtual void OnFlash(Context input)
    {
        // To avoid it in need .NET sometime to let the derived class to the
    }
}
```

"There is good use of OO methods in this class, such as using virtual methods. This would allow classes to override inherited methods as required, so this class is definitely open for extension. This class also makes use of defined data-types(Context, MemorySpace), so it would be unlikely that this class needs to change since future classes will also be using these abstractions."

"This class is a good example of the open/closed principle. You have made use of events, which would allow each inheritor class to perform a different action when a condition is met, so functionality can be extended. Also, the class seems to perform very general procedures, such as Run(), FlashMemory(). These wouldn't need to change in the future since they clearly fulfill their purpose, so would have no reason to change, hence would not need to be modified."

Class: Opcode base class

```
[Flags]
public enum OpcodeSettings
{
    // Default behaviour
    NONE = 0,
    // Force byte capacity on the opcode
    BYTEMODE = 1,
    // Nothing explicitly in this class, but can be tested for by derived classes.
    SIGNED = 2,
}
```

```
public abstract class Opcode : IMyOpcode
{
    private RegisterCapacity _capacity;
    protected RegisterCapacity Capacity...;
    public readonly OpcodeSettings Settings;
    private readonly string Mnemonic;
    private readonly IMyDecoded Input;
    public Opcode(string opcodeMnemonic, IMyDecoded input, OpcodeSettings settings = OpcodeSettings.NONE, F
    {
        Mnemonic = opcodeMnemonic;
        Input = input;
    }
}
```

"This class has clearly used some aspects of the open/closed principle, for example, the IMyDecoded interface is used as an input for some functions instead of using specific classes, so it would always be possible to extend the behaviour of the class for new input types in this respect. However, the OpcodeSettings enum will need to be modified in the future when new settings are added."

Figures from post development testing

Peer review was conducted using experienced OO programmers to evaluate the code. Feedback was given in written form since a more in-depth explanation and justification was needed than a survey like for previous criteria. This criterion is partially met. This is because for the first class, there was highly positive feedback, however for the second class, the feedback was critical. This suggests that there are ways this criterion can be met further in the future.

Further development plan

This criterion can be improved upon in further development. This can be done by restructuring the opcode class to have classes to represent each characteristic of the opcode class. This would be a large amount of work, so it appropriate to delegate among maintainers. This would involve having a class for each opcode setting instead of an enum such that a new class could always inherit from a base "opcode setting" class and work with the opcode rather than having to modify the enum entries in the code, hence code would not be open for modification and therefore meet the success criterion fully.

Criterion Liskov substitution principle. Subtype Requirement: Let $f(x)$ be a property provable about objects of type T. Then $f(y)$ should be true for objects y of type S where S is a subtype of T.[4]

Outcome: Partially successful

How this has been met

Interfaces have been used to allow any class that meets the requirements of the interface to be able to be used in a function that accepts that interface as a parameter.

```
public interface IMyOpcode
{
    public void Execute();
    public List<string> Disassemble();
}
```

```
OpcodeCaller CurrentCaller;
IMyOpcode CurrentOpcode;
if (OpcodeTable[OpcodeWidth].TryGetValue(Fetched, out CurrentCaller) && (CurrentOpcode = CurrentCaller()) != null)
{
    // If disassembling, whether the instruction is executed or not is not of importance(so long as the opcode class is written along with convention),
    // Conversely, if executing, whether the instruction is disassembled or not doesn't matter. Together, this check speeds up the program a lot.
    if ((CurrentHandle.HandleSettings | HandleParameters.DISASSEMBLE) == CurrentHandle.HandleSettings)
    {
        DisassemblyBuffer.Add(
            new DisassembledLine(CurrentOpcode.Disassemble(),
                CurrentContext.Breakpoints.Contains(InstructionPointer) ? AddressInfo.BREAKPOINT : AddressInfo.NONE
                , Start_RIP
                , InstructionPointer - Start_RIP));
    }
    else
    {
        CurrentOpcode.Execute();
    }
}
```

Extract from code

For example, IMyOpcode is the interface used by all opcodes. This allows any opcode, including opcodes that will be developed in the future, to have execute and disassemble called on them, which are the two functions required by the processing layer. This would be the property provable of every subtype.

How this has been measured

This criterion has been measured by the use of polymorphism in the program. There are three main modules that would expect to be expanded in number of classes throughout the development. These are the CustomControls, Hypervisors and Opcodes. As shown previously the IMyOpcode interface is used for opcodes. For hypervisors, the hypervisor base class has been used.

```
public HypervisorBase(string inputName, Context inputContext, HandleParameters handleParameters = HandleParameters.NONE)
{
    // Automatically create a new handle for the derived class. The derived class should rarely have to worry about handles,
    // but for the scope of advanced usage in the future, it is left as protected. I don't wish to restrict the modular
    // capabilities of the program.
    Handle = new Handle(inputName, inputContext, handleParameters);
}
```

Extract from code

This was appropriate since the hypervisors needed more of a rigorous definition than an interface in the procedures which are implicitly performed by hypervisor base such as creating the handle. This still an implementation of the LSP

as there are many attributes and provable properties that can be used by other classes by interacting with the HypervisorBase class rather than the subclass, such as the HandleID and HandleName. For custom controls, the IMyCustomControl interface was used.

```
public interface IMyCustomControl
{
    Layer DrawingLayer { get; set; }
    Emphasis TextEmphasis { get; set; }
}
```

Extract from code

This gave properties that were used by the theming engine to apply the theme consistently to each control. Each control is able to be processed by the theme engine so long as it has the properties, hence can be drawn accordingly. This would allow for any class that implements this interface to be drawn in the same way. Since the main parts of the object code that would be applicable to the LSP have been used, this criterion can be said to be partially met.

Future development plan

This criterion can be improved on in the future by making more of the code work through an interface rather than through explicit references to attributes. For example, the ControlUnit could become an interface that has a required method for executing, then this would allow any class that implements said interface to constitute as a control unit, allowing the CU module to be swapped out easier in the future, which would further allow for different architectures to be emulated.

Criterion: Interface segregation principle. Clients should not be forced to depend upon interfaces that they do not use.[5]

Outcome: Partially successful

How this has been met

Large classes have been broken down into subclasses with more specific purposes. For example, at the beginning of the iterative development, there existed only one base class for opcodes. Then in the developer preview version, the base class was split into two, one for general opcodes and one for specific opcodes. This allowed for more specific methods relevant to the string operation opcodes to be used, moreover these methods also did not need to be depended on by other opcodes that would not need them.

How this has been measured

"There is clear use of the ISP in the opcode classes. This has meant that the string operations, which had particular methods required such as AdjustDI, could be separated from the opcode base class, such that every opcode that isn't a string operation does not have to depend on those methods. Likewise, methods such as "Set" in the base opcode class are not necessary for the string operation class as they handle the process differently.

There are surprisingly many cases where further use of ISP would be nonsensical/useless in the code, however in some places there is room for improvement of ISP usage in this code. For example, there could have been more usage in the hypervisor interface layers instead of a base class. This would reduce dependency on the functions provided in the class if one day they are deprecated and instead, an interface could be used that specifies the minimum requirements for a base class. This would allow more base classes to be implemented in the future."

Figure from post development testing

I asked an OO programmer I know if he could look through some of the project code and write a short review. The feedback was overall positive, however there were some areas that could be improved. This suggests that the success criterion has been partially met.

Further development plan

This success criterion can be met fully in further development. As the essential features are functional and robust, it would be appropriate to take time to reevaluate the code structure. As suggested in the feedback, more modularity could be introduced by creating skeleton interfaces for some of the core modules such that they can be replaced in the future. This could be done by having an interface that specifies which methods and attributes will be necessary for any class in the future, such as really basic things like "Run" for the hypervisor base. This will prevent the interfaces from ever becoming outdated as the program develops. This would meet the criterion further since new modules in the future will not be dependent on crowded interfaces(e.g. many required methods that are no longer necessary in future code), rather a bare interface.

Criterion: Dependency inversion principle, " High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces). Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions." [2]

Outcome: Partially successful

How this has been met

Abstractions have been used as part of the intermediary layer to simplify complex assembly tasks. For example, the memory space emulates primary memory used to store program data, however appears like a dictionary to the caller. This has allowed the processing layer(details) to depend on the intermediary layer(abstractions) such that the interface layer can be updated freely without being affected by any changes to the lower level modules.

How this has been measured

Scope: ControlUnit

"I don't know specifically about the dependency inversion principle, however, your code in the ControlUnit class was very logically structured and had a clear process of operation. Despite some understandable workarounds, the execute method behaves very similar to how a real life process would think, at least algorithmically and without lower level details irrelevant to the code. As you suggested, I did not need to read any other code files to understand the class, although found myself doing so out of curiosity afterwards."

Scope: ModRM

"This class represented a systematic process of determining the properties of a ModRM byte. It was a very reasonable attempt of decoding the table of parameters(following an if-else chain structure). However, I think this class should have been more dependent on the abstraction than it is currently. For example, the DecodedSIB variable is a nullable object. Although it is private, it is still strange to see this as it could cause errors and exceptions if not handled properly, which may not be known to a future maintainer. I think this should at least be signposted in the code or the approach re-evaluated."

Figure from post development testing

I asked an assembly programmer I know to look at some of the code without looking at any other classes(exclusively a given class, one at a time). I then asked him to write a short paragraph about how he could understand the code and any potential issues he had with it. The feedback is mostly positive. It definitely suggests that for the control unit, the dependency inversion principle has been met entirely. However, as said, there are some areas that need to be improved for the criterion to be met fully. As the reviewer understood the code without having to read the code for the high level code or abstractions, abstractions clearly served their purpose as part of the dependency inversion principle in order to simplify the code. Therefore it can be said that this success criterion has been partially met.

Further development plan

This criterion can be met further by making some changes to the code. As suggested by the reviewer, the dependency inversion principle has entirely been met in some respects(such as not having dependencies between high and low level modules), but the usage of the abstractions into which the details are dependent can be improved. This could be done by looking particularly at the ModRM class and seeing where the issues suggested were, maybe asking other programmers for input as a second opinion. I would also reach out the the reviewer again and ask them about other parts of the code to see if they have similar issues. The ModRM class reviewed was old, it has been part of the project for a long time, so is due a code-refactor to improve in places where the abstractions it depends on were updated, but the usage in the ModRM class was not, which would allow for more opportunities for better usage of the abstractions.

Criterion: KISS principle(Keep it stupid simple)

Outcome: Partially successful

How this has been met

```
public static FlagSet Add(byte[] input1, byte[] input2, out byte[] Result, bool carry = false)
{
    // Validate that the two strings are both the same length.
    if(input1.Length != input2.Length)
    {
        throw new Exception("Invalid input array sizes; they are not equal length");
    }
    // The two inputs must be the same length.
    Result = new byte[input1.Length];

    // Instead of using built-in methods, I use my own algorithms compatible with byte arrays increased performance
    // on critical operations that are frequently used. http://prntscr.com/ojwfs2
    for (int i = 0; i < Result.Length; i++)
    {
        // Declare a sum integer, which must be an integer because I anticipate values > 0xFF
        // , which are handled using carries shortly.
        // If there was a carry on the previous byte in the array
        // (or operation even, if the CF was set), add that to the sum.
        // Consider 9 + 1, the least significant digit overflows to 0
        // whilst the 1 carries to the next. It is impossible for the carry
        // to represent a value greater than one in the next column(e.g 9+9=18) in addition.
        int sum = input1[i] + input2[i] + (carry ? 1 : 0);
        Result[i] = (byte)(sum % 256);
        carry = sum > 255;
    }
}
```

Extract from code

Code has been heavily commented, describing the actions performed by important lines and also explaining their purpose. This helps with unavoidably complex lines. In general the simplest route has been taken for code. For example, instead of using integer types, byte arrays have been used as a universal data type throughout the code. This is simpler as memory is very type-independent and has many sizes, there would otherwise be a large amount of casting between types and separate subroutines for each size(e.g. short, int, long, as they are not interchangeable without a cast).

How this has been measured

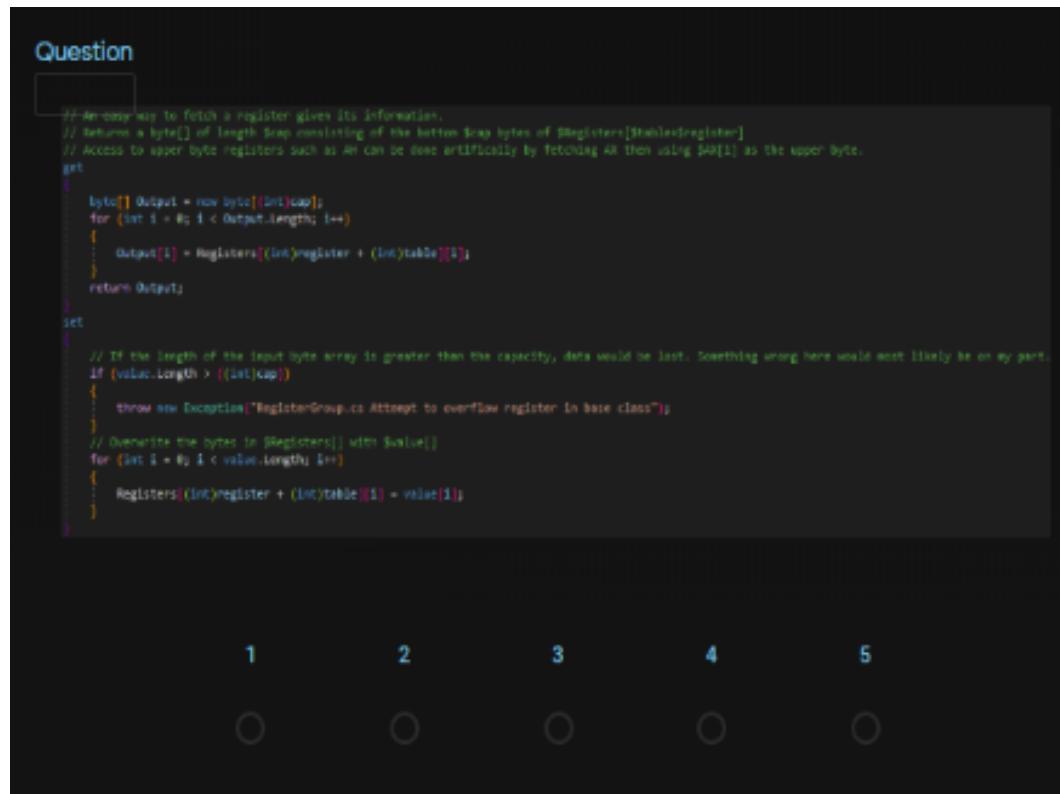


Figure from post-development testing

In a test from the testing to inform evaluation section, classmates were given extracts of code to read, then asked to give a score from 1-5 based on how much they could understand. The average score was 3. This is acceptable as it is unrealistic to expect intermediate programmers given an entirely new piece of code in a language they have perhaps not even seen before. Therefore it can be said that this success criterion has been partially met.

Further development plan

This success criterion can be met further in the future. This could involve a more complete documentation. This would mean that users who read the code starting from an arbitrary file would have something to refer to, rather than following the order in which things tend to be explained(such as reading an opcode class before reading the opcode base class, they would not understand the inherited functions). Another approach would be to have further peer review giving in-detail justification. This would allow specific changes to the comments to be made to fix the problem directly, as it is unclear from the test how to improve, only that it needs to be improved further.

Criterion: Calling trees will be pipelined through layers without backflow where possible

Outcome: Successful

How this has been met

1. (In interface layer)User presses run button
2. (In intermediary layer)Hypervisor performs preconditional actions then executes Run
3. (In intermediary layer)Handle calls CU.Execute()
4. (In processor layer)CU sends out instructions then stores and executes them

Example pipelining process in code

The project has been divided into three separate layers: the intermediary layer, processing layer, and interface layer. In this context, the library is standalone as it is acceptable to have each layer accessing the library regardless of pipeline, as that is its purpose. The processing layer and interface layer communicate through the intermediary layer through abstractions.

How this has been measured

The screenshot shows two call stacks. The top stack is for the 'Processing layer' and includes frames like 'debugger.Emulator.Opcodes.Mov.Execute()' and 'System.Threading.Tasks.Task<debugger.Emulator.Status>.InnerInvoke()'. The bottom stack is for the 'Interface layer' and includes frames like 'debugger.Hypervisor.HypervisorBase.RunAsync(bool Step = false)' and 'System.Windows.Forms.Control.OnClick(System.EventArgs e)'. Red annotations highlight the 'Processing layer' and 'Interface layer' names next to their respective stack frames.

Figures from post development testing

A test was conducted to monitor the call stack for executing an example program. This figure shows the call stack for the mov opcode, however for any other opcode, the only difference would have been in the final call, "Mov.Execute()". The figures show a descent from the interface layer, to pre-processing layers in the intermediary layer, then finally to the processing layer. This shows that the success criterion has been met entirely. There was no need for direct communication between the processing layer and interface layer, hence the abstractions served their purpose.

Further development plan

This success criterion can be improved and maintained in the future. This would be through the expansion of abstractions to accommodate new features in either the processing layer or interface layer. This would remove any need for direct calls in the future. As the criterion has already been met, it is only necessary to maintain it. This could be done by communicating the intentions with any future maintainers.

Evaluation of usability features

Feature: Adding breakpoints

The screenshots show the debugger's assembly view. The first screenshot shows a breakpoint being set at address 0x000000000000000E. A red arrow points to the instruction 'ADD EAX, 0x80000001' with the label 'Breakpoint'. The second screenshot shows the 'Run' button being pressed, with a red arrow pointing to the instruction 'ADD EAX, 0x80000001' and the label '"Run" pressed'. The third screenshot shows the breakpoint being removed by clicking again, with a red arrow pointing to the instruction 'ADD EAX, 0x80000001' and the label 'Clicked again to remove'.

Evidence from post development testing

Verdict: Partial success

Justification

This can be considered a partial success because it meets the aims set out in the design section, but has room for improvement. As shown in the evidence, a breakpoint could be inserted by clicking and then is displayed visually by

the pink colour. This has made it stand out significantly, it would be very clear to a new user which lines are breakpoints and which are not. If the user did not know what a breakpoint was, after pressing run a few times it would become clear quickly. This shows that the usability aspect of this feature has been successful. Moreover, the evidence shows that the usability feature also functions properly.

Further development plan

This usability feature can be more successful through further development. This would include having a separate button, possibly in a dropdown, to add a breakpoint by entering its address into a textbox. This would allow users to quickly add a breakpoint if they know the line. This would be most noticeable when the user has a large program, and would otherwise have to scroll around to find the line they want.

Feature: Control buttons

```
Disassembly
0x00000000000000000000000000000000 +RIP      MOV EAX, 0x7FFFFFFF
0x00000000000000000000000000000005          ADD EAX, 0x01      Initial state
0x00000000000000000000000000000008          NOP
0x00000000000000000000000000000009          MOV EAX, 0x80000000
0x0000000000000000000000000000000E          ADD EAX, 0x80000001
0x00000000000000000000000000000013          NOP
0x00000000000000000000000000000014          MOV EAX, 0xF000
0x00000000000000000000000000000019          MOV EBX, 0x0
0x0000000000000000000000000000001E          ADD AX, 0x1000
0x00000000000000000000000000000022          ADC BX, 0x0
0x00000000000000000000000000000026          NOP
```

```
Disassembly
0x00000000000000000000000000000000      MOV EAX, 0x7FFFFFFF
0x00000000000000000000000000000005 +RIP    ADD EAX, 0x01      Step pressed
0x00000000000000000000000000000008          NOP
0x00000000000000000000000000000009          MOV EAX, 0x80000000
0x0000000000000000000000000000000E          ADD EAX, 0x80000001
0x00000000000000000000000000000013          NOP
0x00000000000000000000000000000014          MOV EAX, 0xF000
0x00000000000000000000000000000019          MOV EBX, 0x0
0x0000000000000000000000000000001E          ADD AX, 0x1000
0x00000000000000000000000000000022          ADC BX, 0x0
0x00000000000000000000000000000026          NOP
```

Part of evidence from post development testing

Verdict: Partial success

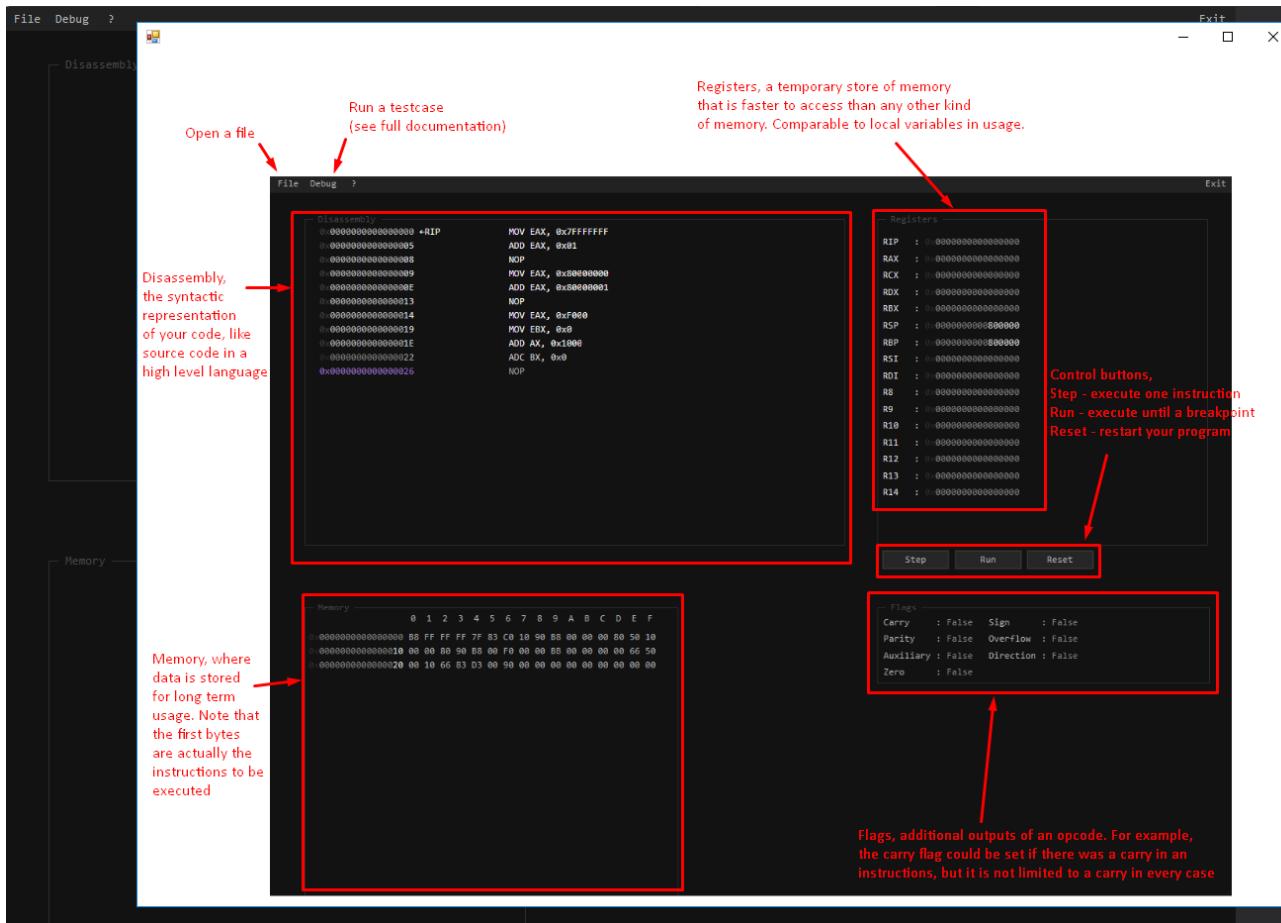
Justification

This usability feature has been partially successful as it meets the aims set out in the design, but could be improved. The step, run, and reset buttons have been implemented successfully as shown in the evidence; they function properly. They are also very usable, the buttons are named instead of having icons. This was shown in the analysis research section to be confusing for the stakeholders of VMCS. If stakeholders are unsure of what the button does, they could always google the text, but would struggle to google an icon.

Further development plan

In the future, this could be expanded by having extra control buttons that make the program more usable. For example, a jump button could allow certain instructions to be skipped. This is a feature present in most alternative software. Along with other operations, this would allow the user to have more control over the execution of instructions.

Feature: Help menu



Screenshot from program

Verdict: Failure

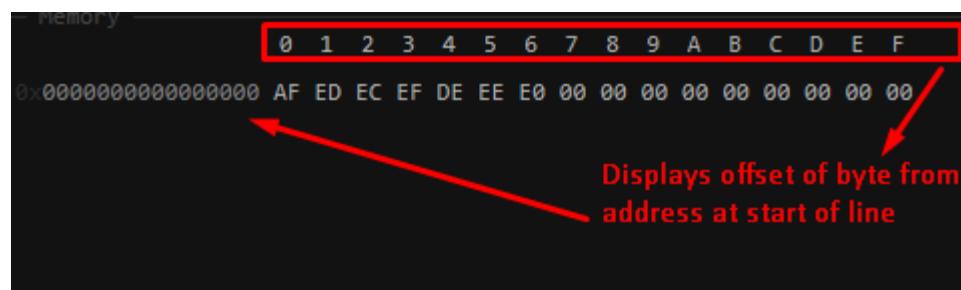
Justification

This usability feature has not been successful to the extents that were set out in the design stage. It has made, or has the potential to, the program somewhat more usable. This is because it is easily accessible from the "?" button in the interface, which has obvious implications. It also briefly describes the function of each element of the interface. This would be most useful to someone who already understands what each feature does out of context, just not where it can be found/exact implementation details in the context of VMCS, however, for a user who does not know, there are obvious keywords such as "Flags" used in the descriptions that could easily be googled. However, the help menu is a very surface level view of the program. It does not explain anything that could not be deduced and it does not introduce the user to new features that are not present in the other programs researched in the analysis.

Further development plan

In the future, this usability feature could be integrated more into each UI control. For example, having a specific help button in the disassembly window for disassembly. This would give more specific and in-depth help on a feature-by-feature basis. This would prevent any information overload and give a more relevant explanation for the part of the program the user is unsure about, rather than information about every part. It would also be overhauled using a graphics library. This would allow for a more guided tour of the program and interactive help to show the user how the program works as opposed to displaying a block of text. Unfortunately due to time constraints, it has not been possible to code a graphics library and learn the coding techniques required in graphics.

Feature: Column headers



Screenshot from program

Verdict: Failure

Justification

Through research in the analysis section, it was found that alternative softwares did not all have this feature, and as a result it could be complicated/easy to make an error when using their interface. While the feature may at first appear

very successful, in hindsight it has not addressed the problems that was found in the alternative software. It is hard, or at least not easy, to see where the columns line up with the bytes displayed. This means the users may have to trail down the screen with their fingers to find what they are looking for, all of which is inconvenient. However, to create such a grid or table format would require creating a custom windows control as currently a ListView is used. This would require reinventing the wheel to recreate the functions performed by a ListView as they and put them in the new custom control. It would also require rewriting all of the code for the sorting and managing of memory as the use of List objects is deeply rooted into other parts of the program. Solely due to time constraints, it has not been possible to code this in the release version.

Further development plan

This feature will be improved in the future by allowing the address of any byte to be copied, possibly by the use of a dropdown right click menu. This would mean that users do not have to write out the address, which could equally have as high a possibility of error, such as missing a zero.

Feature: Register resizing

Registers	
RIP	: 0x0000000000000009
RAX	: 0x0000000080000000
RCX	: 0x0000000000000000
RDX	: 0x0000000000000000
RBX	: 0x0000000000000000
RSP	: 0x0000000000800000
RBP	: 0x0000000000800000
RSI	: 0x0000000000000000
RDI	: 0x0000000000000000
R8	: 0x0000000000000000
R9	: 0x0000000000000000
R10	: 0x0000000000000000
R11	: 0x0000000000000000
R12	: 0x0000000000000000
R13	: 0x0000000000000000
R14	: 0x0000000000000000

Registers	
RIP	: 0x0000000000000009
AX	: 0x0000000000000000 Two double clicks
CX	: 0x0000000000000000
DX	: 0x0000000000000000
BX	: 0x0000000000000000
SP	: 0x0000000000000000 As designed, the bytes outside of word range have been omitted
BP	: 0x0000000000000000 entirely(to only
SI	: 0x0000000000000000 show relevant information)
DI	: 0x0000000000000000

Part of evidence from post development testing

Verdict: Success

Justification

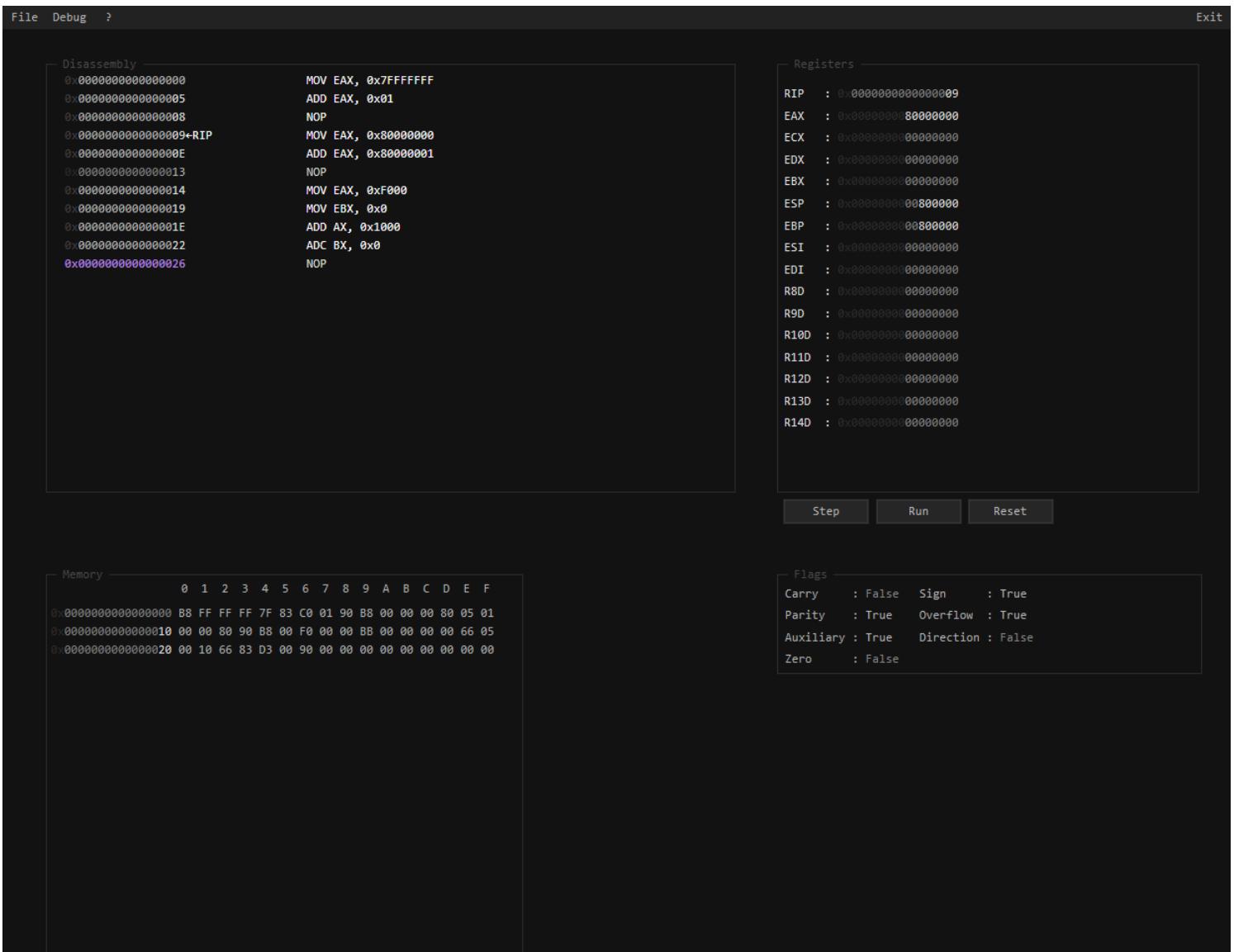
An example of this features success is that the emphasis of text has been implemented properly. The “0x” tag has been de-emphasised. This is because it is important to notate the values as hexadecimal, but does not need to be highly visible, as it would only be a distraction. Another example is the emphasis of significant bytes by making them white. This makes it clear what values are present in the registers, as it can be confusing seeing so many numbers in one area. It also clarifies an entire byte as two hexadecimal digits, as this may be otherwise hard to see on longer numbers. This can be seen on RIP in the figures. These factors reduce the amount of irrelevant information displayed to the user in general, whilst still leaving it accessible when it is needed. This would allow the user to use the program more effectively as there is less information overload, even though this is a very information heavy context(consider memory viewer, registers, disassembly in one interface. It has already been shown through research into alternative

software in the research section that information overload was a significant drawback that reduced accessibility and efficiency when using the program. The success in this feature is that this has been mitigated to significant extents in VMCS.

Further development plan

This feature can be more successful through further development. This would be by reducing the amount of irrelevant information further. For example, it is not necessary to display empty registers. When empty, they could be removed from the display entirely, or de-emphasised entirely like insignificant zeros currently. This would allow the user to work more efficiently as it will be faster to find the information they want. e.g. if they look at the memory viewer than look back, it could be annoying to have to look through many numbers to find the number they were using previously.

Feature: Material Dark concepts



Screenshot from program

The material dark design theme states 3 key principles.[7]

- Darken with grey: Use dark grey – rather than black – to express elevation and space in an environment with a wider range of depth.
- Color with accents: Apply limited color accents in dark theme UIs, so the majority of space is dedicated to dark surfaces.
- Enhance accessibility: Accommodate regular dark theme users (such as those with low vision), by meeting accessibility color contrast standards.

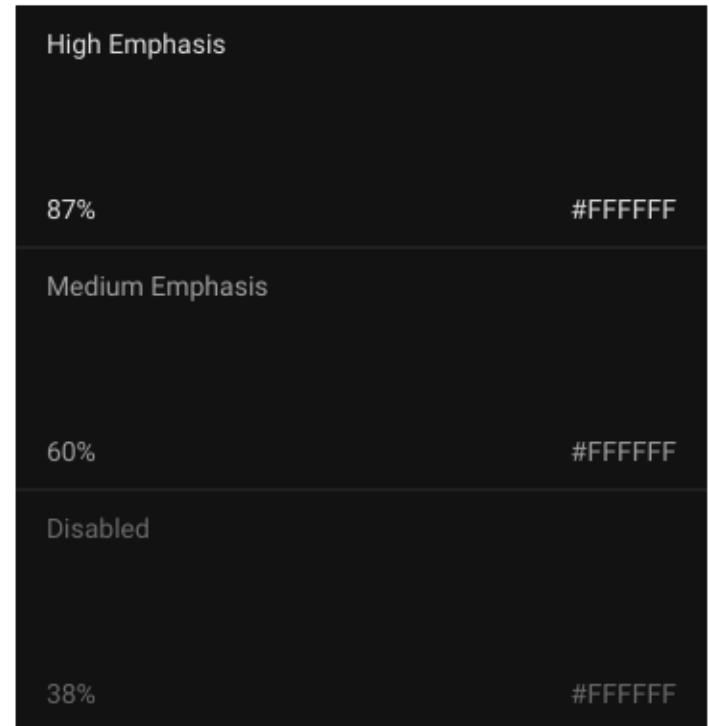
These principles will be followed throughout the theme engine design.

Design criteria

Light text on dark backgrounds

When light text appears on dark backgrounds (shown here as white on black) it should use the following opacity levels:

- High-emphasis text has an opacity of 87%
- Medium-emphasis text and hint text have opacities of 60%
- Disabled text has an opacity of 38%



High-emphasis, medium-emphasis and disabled text

<https://material.io/design/color/dark-theme.html#ui-application>

```
SurfaceShades = new List<Color>()
{
    BaseUI.SurfaceColour,
    Color.FromArgb(220, BaseUI.SurfaceColour), // ~87% transparency of text.
    Color.FromArgb(153, BaseUI.SurfaceColour), // 60%
    Color.FromArgb(97, BaseUI.SurfaceColour), // ~38%
    Color.FromArgb(17, BaseUI.SurfaceColour) // 20% An extra was required for background
};
```

Code extract

Verdict: Partial success

Justification

This usability feature can be considered a partial success. This is because it has met most of the design criteria. For the first criterion, dark gray has been used to express elevation in all parts of the interface except the background, so has been partially met. For the second criterion, colour has been used to highlight breakpoints and the majority is dark, so both parts of this criterion have been met. For the third criterion, accessibility color contrasts have been met. This is certain because the same colours were used in the program as in the official design specification, as shown in the figures.

Further development plan

This usability feature can be made more successful in the future. This would include changing the background to comply more with the specification. This could be done by making it gray instead of black, allowing this first criterion to be met entirely.

Feature: Markdown engine

Disassembly	
0x0000000000000000 ←RIP	MOV EAX, 0xFFFFFFFF
0x0000000000000005	ADD EAX, 0x01
0x0000000000000008	NOP
0x0000000000000009	MOV EAX, 0x80000000
0x000000000000000E	ADD EAX, 0x80000001
0x0000000000000013	NOP
0x0000000000000014	MOV EAX, 0xF000
0x0000000000000019	MOV EBX, 0x0
0x000000000000001E	ADD AX, 0x1000
0x0000000000000022	ADC BX, 0x0
0x0000000000000026	NOP

Screenshot from program

Verdict: Partial success

Justification

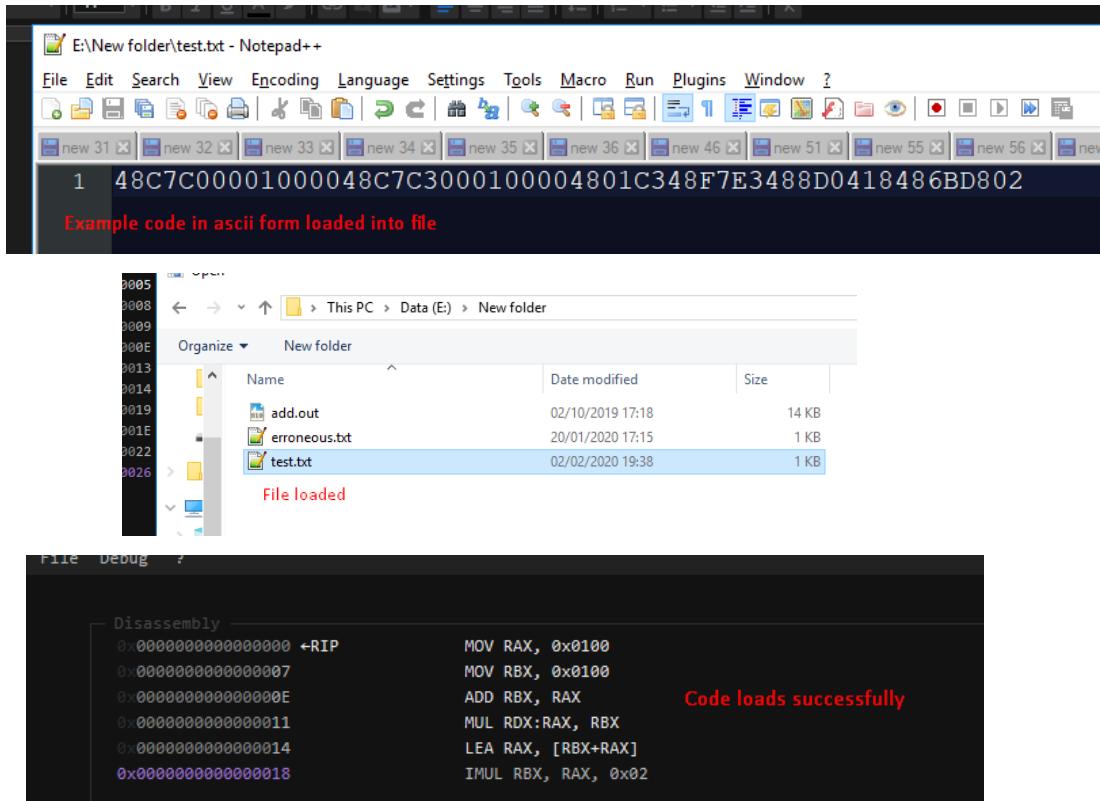
This usability feature has been partially successful. An example of success would be the highlighting of text in the interface. This significantly reduces the effects of information overload, which was identified as a drawback to alternative software. This is because the markdown engine allows irrelevant information, such as insignificant zeros, to be de-emphasised, and more relevant information, such as mnemonics, to be emphasised. However, there are some flaws that could be improved in the future, such as allowing more colours and different font types(e.g. bold, italic), as most markdown engines provide in order to add more customization to the user interface. This would also allow the code to be better reused in other projects.

Further development plan

This usability feature can be met further in the future by expanding the feature set. Currently the markdown engine only provides a narrow feature set tailored specifically to the current needs of the program. This could be expanded to allow for more advanced UI features in the future, such as bold text and resizing of text.

Feature: Loading of TXT files

Part of evidence from post development testing



Verdict: Partial success

Justification

This usability feature has been partially successful. This is because it has met some of its initial goals, but could have been developed further. For example, it is possible for the user to load code without having to compile and link it. This is done by converting the ascii bytes into the binary they represent e.g. "48" => 0x48 and validating in the process, then interpreting the input. This makes the program more usable as the user does not have to depend on other tools to compile and link code that may not be available to them, allowing them to make use of the program on limited memory or a restricted computer. There are websites on the internet that allow assembly code to be assembled for free.

Further development plan

This usability feature could be more successful in the future by allowing mnemonics to be loaded without having to be assembled. E.g. a text file containing "MOV EAX, 0x100". This would remove all dependence of the program from other programs; the user need not have internet access or other software installed. This would make the program more accessible to stakeholders as the stakeholders are generally new to assembly, so would not know which programs they need/how to compile or assemble code. This could be implemented by constructing a reverse process for the disassembler module, such that instead of converting IOpcodes objects into mnemonics, mnemonics would instead be converted into IOpcodes objects.

Feature: Loading instructions from clipboard

The screenshot shows a debugger interface with a menu bar (File, Debug, ?) and a submenu (Open, Open from clipboard). The assembly code window displays the following instructions:

0x0000000000000000	→RIP	MOV RAX, 0x0100
0x0000000000000007		MOV RBX, 0x0100
0x000000000000000E		ADD RBX, RAX
0x0000000000000011		MUL RDX:RAX, RBX
0x0000000000000014		LEA RAX, [RBX+RAX]
0x0000000000000018		IMUL RBX, RAX, 0x02

A red annotation "Loaded from clipboard" is placed next to the assembly code.

Part of evidence from post development testing

Verdict: Partial success

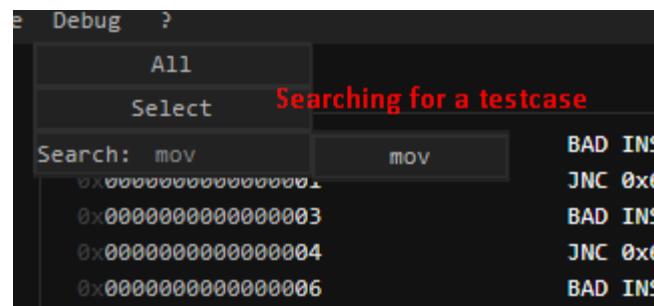
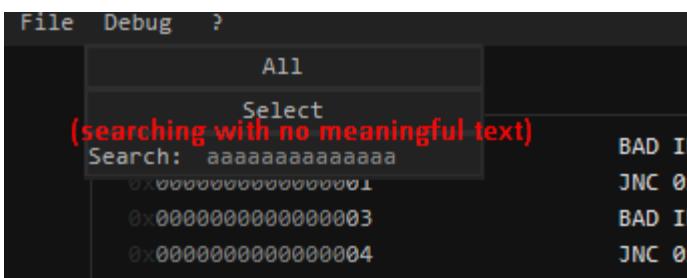
Justification

This feature was partially successful as it allowed code to be loaded easily. The user need only copy the instructions into their clipboard, such that they do not have to install any other tools to compile and link the code, or create a text file(however they may still wish to use a text file if they downloaded it). This allows them to quickly load any modifications to their code back into the program, so can use it more efficiently. This has not been met entirely as its usability could be improved. This could be through a help button, explaining what the user has to do to use this feature. It is not clear that the input has to be in text form.

Further development plan

A section will be added to the help menu that explains how to format the input correctly to be parsed by the “open from clipboard” feature. This could be simple, such as “Input must be in ascii format”, but will definitely make it more clear to the user.

Feature: Searching testcases



Part of evidence from post development testing

Verdict: Partial success

Justification

This feature has been partially successful. This is because it has met its usability and robustness requirements; the user can type into the box directly to find a testcase they are looking for quickly, which could take some time as there are currently many testcases in the program, and that number will grow in the future. It has also been shown to be robust in the testing to inform evaluation by entering arbitrary text to try and produce an error, which it did not. This could be improved by allowing a different search directory to be set, as it would not be intuitively clear to use the testcases folder where the executable is located; they may not have realised it exists.

Further development plan

This usability feature will be met further by allowing the user to configure the search directory and will be displayed separately on the dropdown menu. This will allow them to use their own directories or load a testcase downloaded from the internet quickly.

Limitations and maintenance issues

Limitation: The opcode map will only be covered to a reasonable scope

A large number of opcodes have been included in the final solution, enough to produce a reasonable program to allow genuine applications of the program. However, an advanced user would have little use for the program as they would require every opcode in the x86-64 specification to be at their disposal.

How it could be dealt with

It would be possible to overcome this limitation through further development as the apis and interfaces are already set up to implement new opcodes easily, all that is required is the extra work of writing and testing code.

Limitation: The solution will not include features of alternative software if the only purpose is that it is a useful feature. This would include extra control buttons to allow the user to have more control over the flow of execution, for example, jumping to addresses and editing memory. Due to time constraints, it has not been possible to implement these features. This is because there are so many small quality of life features present in modern debuggers that have accumulated over the long time they have been in development.

How it could be dealt with

This would be negated entirely through further development as newer versions would gradually introduce such features and the program would then include these features. It could also be a process of waiting for user feedback on what features they need.

Limitation: The solution will not include a built in assembler

It has not been possible to implement an assembly into the code due to the nature of how the program is structured. In its current state, it would require a backflow of data from the processing layer to the interface layer as user input would be translated directly into machine code. This would contradict the success criteria, so would require significant workaround to include.

How it could be dealt with

This would be no problem in further development as a program/library could be developed to run alongside VMCS and handle all of the disassembly then be integrated into the program such that the success criterion would still be met.

Limitation: The solution will not include methods of loading windows .exe executables.

There are many ways windows exe files can be loaded with discrete differences that would require a very rigorous approach. It would also require interaction with the native windows binaries and libraries. This would have been a significant amount of work for a feature that is very unlikely to be used by the stakeholders, they would always have the option of using an assembler to write their code.

How it could be dealt with

This is unlikely to be addressed in further development as there are more important issues to be addressed that will benefit the stakeholders. However, the code certainly has the ability to facilitate it. Modules allow the exe to be detected via magic bytes and loaded into a class, this is not a problem. The work required would be in creating a set

of rules to read the file headers into variables then interpret them, adjusting the necessary settings in the VM to accommodate any changes. As windows executables depend strongly on Win32.DLL libraries, a module would be required that allows multiple programs to execute at once, allowing the injection of independent libraries into a running process. As such, this would require a large team of highly skilled programmers, which is not possible due to budget constraints.

Maintenance issue: Language barrier in comments

All comments are written in english as I only speak english. This narrows down the accessibility of the program code as someone who does not speak english could neither read the comments and understand the code nor maintain the project, which are important parts of VMCS. There is also a lot of necessary technical terms used when describing the code, which may be easy for an english speaker, but hard for an english learner. It is unlikely that someone would translate the code for free due to the large amount of comments, so they would have to be paid. This is unlikely to happen since VMCS is an open-source led project, so has no budget.

Maintenance issue: Code uses .NET framework and requires windows

Most open source developers use GNU/Linux, which is not compatible with the .NET framework. This would mean that maintainers have to use windows to work on the program. This would seem like a lot of hassle for them especially if they do not have a windows computer. This would mean that less maintainers are likely to look at the code, so less issues will be found and reported. Since the project stakeholders are generally new to low level programming and not experienced with this type of software, there will be a lot of issues reported and requests made. This may be a lot to handle for one developer.