OCR Computing H446: Component 3

Noah Stride

Bournemouth School

Centre Number: 55119

Candidate Number: 2129

# Table of Contents

Contents

[Table of Contents 2](#_Toc1411856)

[Contents 2](#_Toc1411857)

[Section 1: Analysis 5](#_Toc1411858)

[Problem Identification 5](#_Toc1411859)

[Complexity of the Simulation 5](#_Toc1411860)

[Educational Requirement 5](#_Toc1411861)

[Controlled Environment 5](#_Toc1411862)

[Ease of Visualization 6](#_Toc1411863)

[Existing Infrastructure 6](#_Toc1411864)

[Stakeholders 7](#_Toc1411865)

[School Management 7](#_Toc1411866)

[Teaching Staff 8](#_Toc1411867)

[Students 9](#_Toc1411868)

[Existing Solutions 10](#_Toc1411869)

[ArmSim 10](#_Toc1411870)

[Little Man Computer 13](#_Toc1411871)

[Assembly Training Program (ATP) 16](#_Toc1411872)

[Limitations 19](#_Toc1411873)

[Direct execution of compiled code 19](#_Toc1411874)

[Ability to run on a wide variety of platforms 19](#_Toc1411875)

[The entirety of an existing instruction set 19](#_Toc1411876)

[Alternative architectures 19](#_Toc1411877)

[Software and Hardware Requirements 20](#_Toc1411878)

[Software 20](#_Toc1411879)

[Hardware 21](#_Toc1411880)

[Features and Requirements 22](#_Toc1411881)

[Essential Features 22](#_Toc1411882)

[Requirements 23](#_Toc1411883)

[Success Criteria 24](#_Toc1411884)

[Section 2: Design 25](#_Toc1411885)

[Decomposing the Problem 25](#_Toc1411886)

[Designing the Processor Class 25](#_Toc1411887)

[Designing the Register Class 30](#_Toc1411888)

[Designing the ALU Class 31](#_Toc1411889)

[Designing the MemoryBank Class 33](#_Toc1411890)

[Section 3: Development 37](#_Toc1411891)

[Wire Framing the Classes 37](#_Toc1411892)

[Testing 44](#_Toc1411893)

[Bug Fixes 44](#_Toc1411894)

[Producing the rough UI 47](#_Toc1411895)

[Testing 49](#_Toc1411896)

[Bug Fixes 50](#_Toc1411897)

[Producing the Memory Bank UI 52](#_Toc1411898)

[Testing 57](#_Toc1411899)

[Bug Fixes 58](#_Toc1411900)

[Producing the Memory Bank Model 59](#_Toc1411901)

[Testing 61](#_Toc1411902)

[Bug Fixes 62](#_Toc1411903)

[Adding Instructions 63](#_Toc1411904)

[Testing 69](#_Toc1411905)

[References 75](#_Toc1411906)

# Section 1: Analysis

All solutions begin with an adequate analysis of the problem. This sets a framework for the solution to be built within, with clear expectations. This helps prevent resource consuming ‘feature creep’ that can lead to missed deadlines and the eventual derailment of a project. The analysis will begin by focusing on the problem itself, before considering the stakeholders and other solutions to the problem. Finally, a rigid specification will be set for the solution.

## Problem Identification

The chosen problem is that of simulating a computer processor and operation for the purposes of teaching computer science. The software should be appropriate for a variety of educational institutions and be considered on par or above the quality level of competing software. The problem is almost exclusively solvable using computational methods, and is amenable to being solved in this way for the following reasons:

### Complexity of the Simulation

A computer processor involves many complicated aspects and this makes it ideal for simulation as a piece of software. Working through a simulation by hand would be incredibly time consuming especially when considering the great speed at which modern processors operate. To simulate even a second of a modern processor would take years, if not decades. By simulating the processor using a modern computer, often capable of running at speeds beyond a hundred million operations a second, we are able to run it at speeds that will allow the user to test basic programs.

### Educational Requirement

The idea of the program is for it to be used to help teach and give people an understanding of a processor. It would be impossible for somebody to manually simulate a processor without having the prerequisite knowledge. By making it an interactive simulation, students will be able to try different things and visualize the various cycles within a processor. Processors can seem to be quite an abstract idea, so being able to interact with a simplification will enable people to grasp a deeper understanding.

### Controlled Environment

A processor is a relatively simple model and can be considered a closed environment. This is the opposite of more complicated systems such as weather and chemical reactions that are often subject to external influencing factors that are difficult to take account for. A tiny inaccuracy in a weather forecast will simply be compounded as the simulation runs. Because a processor is simple, and a closed environment, the simulation will remain accurate, making it an ideal thing to simulate using a computer.

### Ease of Visualization

A modern computer is equipped with graphics capabilities and this means it is possible to render images that will accurately represent the model. This graphics would simply and clearly explain the workings of a processor. This will also make the learning more engaging, as many people tend to be more interested by a visual explanation than a verbal one.

### Existing Infrastructure

Almost all computer science lessons are taught in classrooms with access to computers, which makes this problem incredibly amenable to being solved using a computational solution. This means the software will be able to be conveniently accessed by its key users, and that the staff will not have to book the room in order to use the software. Students may be able to use the software at home to expand their own understanding and revise.

## Stakeholders

Stakeholders remain one of the most important considerations in the development of software. Ultimately, if the software is not appropriate for its stakeholders then the project is a failure. It is important to note that, in many cases, it is impossible to please all stakeholders involved in a project, as many often hold conflicting requirements.

I have decided to primarily target educational institutions because the requirements of a technical professional will contrast too greatly with those of a student. Complexity of the processor must be abstracted to an understandable level, but by doing this valuable information to a professional is lost. Simulating a modern x86 processor is also beyond the scope of my own personal understanding, as the instruction set has grown significantly more complicated from its conception.

### School Management

The bureaucratic element of many organizations is easily ignored when it comes to the production of software, as they will not be final end-users of the software. However, software must still meet many requirements produced by the management of any organization in order for it to be adopted.

|  |  |
| --- | --- |
| User Requirements | Justification |
| Affordable | Software must be deemed economically viable for an organization to adopt it. This does not only cover the cost of purchasing software, but also support agreements and the time as well as financial cost of training the users. |
| Secure | Software must not present an additional risk to the system, intentionally or unintentionally. Vulnerabilities continue to be misused in software in order to gain a foot hold or escalate an attack, and it is the responsibility of the software developer to ensure this is not possible. This is not overly applicable to my solution because the software will not interact with any networks, and hence not present an attack vector. As my solution will be written in a high level language, it is unlikely the software will be vulnerable to memory attacks used for privilege escalation. |
| Compatible | Software will be more convenient for adoption by an organization if it is compatible with existing systems, both technically and bureaucratically. Software must be adapted to integrate with any hierarchies or procedures that are ingrained within an organization, as this means that integration will be a simpler, and cheaper, process. For our solution, this means the application should be of adequate depth to easily meet the teaching requirements of common Computer Science |

### Teaching Staff

Teachers represent the first of the user groups. The software must be apt to their needs, as they will spend the most time interacting with the software as it is used year on year with different classes. Software that does not adequately fulfil their requirements will become disused, with more appropriate solutions being found.

Mr. Albanozzo is a computer science teacher at Bournemouth School. He first suggested the project because he was frustrated with the limitations proposed by the Little Man Computer currently used to teach about processors and felt that a better version could be made.

|  |  |
| --- | --- |
| User Requirements | Justification |
| Simplicity and Demonstrability | Software must be simple to use and demonstrate. Software that is self-explanatory is easier to teach the use of, and reduces questions raised by students, allowing more time to teach the course content. Demonstrable software ensures that the UI is clearly visible and clear when shown on a projector and that actions are easily reproducible. |
| Accurate | The software must accurately represent the content of most computer science courses. If the software is misleading it may cause a lack of understanding of the course, which may be complicated to fix. On the other hand, if the software is accurate, it will further their knowledge and allow them to gain a deeper understanding of computer processor architecture. |
| Fast and Stable | The software must be responsive, load quickly, and not experience crashes or data corruption. Slow software holds up the learning process and crashes lead to the loss of work, both of which may require the intervention of the teaching staff to resolve, wasting valuable time. |

### Students

Students act as the primary target for the use of the software. They mostly share requirements with teaching staff, including the requirements for software to be fast and stable, for it to be compatible with their course, and for it to be simple enough to be quickly grasped. There are a few unique requirements.

Emilio is a run-of-the-mill computer science student at Bournemouth School. Whilst he performs well in most areas of the course, he struggles most with processors and assembly. He feels that Little Man Computer does not currently make many concepts needed at A-Level clear. He thinks he would be more easily able to understand the concepts if he had something to visualize how the data moved around the processor.

|  |  |
| --- | --- |
| User Requirements | Justification |
| Adaptable | Students have a varied range of abilities, the software should be able to offer multiple levels of difficulty to continue to keep stronger students engaged. Many pieces of software aimed at educational institutions fail to cater for all levels of student, and either are seen as too complex by weaker students and too simple by stronger students. |
| Clear | The student must be able to clearly see how data and instructions are moved around the system and how the important registers interact with each other. This will enable them to get a better understanding and help them to more easily remember the concepts as they can see them visualized. |

## Existing Solutions

Existing solutions provide great inspiration as to the required features of an application if it is to compete within the same market. Weaknesses of existing applications can be used to bolster your own and to improve the user experience significantly. However, it should be noted that not all pre-existing solutions may seek to solve the problem in the same way, with some focusing on alternative user groups

### ArmSim

ArmSim is a realistic simulation of an ARM RISC processor designed for testing assembly code designed for ARM processors before it is used on physical hardware. It provides a rather gray UI and provides a simple way for a programmer to view the running memory of the simulated processor. The UI is not particularly intuitive and the software is mainly aimed at professional programmers who will quickly become accustomed to their tools.



ArmSim does not provide an interface for writing code and instead allows a programmer to use an existing IDE they own to write the assembly and compile it to machine code before importing it. This is far more suitable for a professional who will already have an established workflow with their software and may have to use a certain IDE because of rules set by their management. It acts purely as a simulation of the processor, rather than trying to also act as an IDE.

It provides access to all of the registers that are found within an ARM processor as well as providing the ability to browse through the entire memory space, with a quick jump to an address feature. The stack is also provided as a separate sidebar to allow its contents to be better visualized compared to just browsing the memory locations. It provides the ability to step into and step through a program, or to run it as fast as the processor can. Data can be shown in the registers in decimal, hexadecimal or binary, depending on how comfortable the user is with each form.



The registers are demonstrated on the left side, with modified registers shown in red to clarify what has changed to the developer.

The simulation provides a one to one in-depth replica of an Arm processor, with the entire instruction set implemented with a fine level of detail. Software can be directly compiled to this instruction set from various language and can then be ran step by step through ArmSim. Whilst this level of detail and accuracy is essential to a seasoned programmer, it is unnecessary and perplexing for a student. Instruction sets and processors are far more complicated than the abstraction used by A-Level courses and ArmSim is likely to further confuse rather than help students.

ArmSim is provided as a compiled executable file. It is natively designed to run on Windows x86 64 or 32bit processors and the source code is not open. There has been some documented success at running it using the WINE (WINE Is Not an Emulator) platform on GNU/Linux and MacOSX.

|  |  |
| --- | --- |
| Feature | Justification |
| Fast simulation | ArmSim’s ability to simulate an ARM processor at close to real speeds is useful for developers wanting to test their programs properly. It allows more complex algorithms to be designed. |
| Simple access to memory | ArmSim provides an easy way to browse the memory and this is useful for students and developers trying to understand their program and debug any issues with their algorithms as they can see the changes to the memory clearly. |
| Detailed Simulation | ArmSim provides a comprehensively detailed simulation of an ARM processor. A balance will have to be struck between simplicity and complexity for my simulation as it will primarily aimed at as a teaching tool. |
| Display of values in multiple bases | ArmSim provides the user the ability to view values in decimal, hex and binary and this is beneficial as it will allow users to familiarize themselves with the common bases used within the CompSci world. This is also beneficial as some values only make sense in certain bases (i.e Instructions are clearer in binary where you can see the divide between Opcode and Operand) |
| Console window to provide feedback | The console window acts a simple way for the program to inform the developer of issues, as well as providing the output of the processor to the user. It is less intrusive than pop-up messages and can be scrolled back through to view historical changes. |

### Little Man Computer

Little Man Computer (LMC) is a CPU simulation designed for teaching about assembly and the workings of a processor. It can run within the browser and provides a graphical interface for interaction. Unlike ArmSim it provides a basic editing interface where a simplified assembly instruction set can be written and then compiled into machine code.

LMC provides a variety of features such as the ability to change the simulation speed. This is useful for people wanting to see more about the workings of a processor since they can slow it down and see it step through each part of the Fetch Decode Execute (FDE) cycle. Others may want to speed it up so they can see their code that they have written running more fluidly. There is quite a low limit to this speed and it can be frustrating when you have written a more complicated algorithm (for example to calculate prime numbers) and it takes a long time to execute.

The LMC provides an incredibly simplified model, with only a few core registers such as the PC, IR and MAR. The memory itself has only 99 addresses and each of these is restricted to an integer from 0 to 999. The instruction set is cut down to a few basic instructions and there is only a single memory addressing mode that the user is not even made aware of.



LMC further abstracts the processor by showing all numbers and commands as base ten integers rather than showing them as their binary equivalents. This may make it easier for a newer student to understand, as at first binary instructions may seem intimidating; however this does obscure the fact that an instruction stored in memory is composed of two distinct parts, the operator and operand.

Whilst LMC does simulate only a handful of registers, they are all clearly shown within their own part of the UI. This makes it simple to see how the values of the registers change as the FDE cycle completes, something that you may be examined on at GCSE and A-Level.

The fact that LMC can be embedded within the browser has advantages as well as limitations. One of the prime advantages is portability. Browsers act as a layer of compatibility and this means that LMC can run across different operating systems as well as processor instruction sets. This also means that the application does not need to be pre-installed, only a relatively modern browser is needed, and for the most part this is already present in most schools. The biggest limitation of this is processing speed. Modern browsers and the JavaScript engine add lots of overhead to execution and this restricts how fast it is able to simulate the processor. JavaScript is also a problematic platform considering its varied implementation across not only versions of browsers, but browsers themselves. This makes it hard to predict how it will behave when deployed across a wider user-base.

|  |  |
| --- | --- |
| Feature | Justification |
| Appealing Interface | An appealing interface makes the program more attractive to those learning to program, whilst seasoned developers are more used to more plain design. An appealing interface will also be easy to work with and not frustrate users. |
| Ability to adjust speed | The ability to adjust speed is useful for almost all users, as at lower speeds it is similar to step-through debugging. At higher speeds, the user can examine the overall effectiveness of the algorithm, especially more complicated ones. |
| Simplicity | LMC simplifies the architecture of a processor to a point where it is understandable, however my program must go further in ensuring that it also provides enough depth, which LMC does not. |
| Simple Assembly Editor | LMC provides a useful interface for writing the assembly, unlike ArmSim, and this allows the user to quickly make changes and retranslate the code rather than having to switch out to another application and reload the code. This editor could be taken further to help the user. |
| Values of registers clearly displayed | The values of each register, and how they connect to each other, are clearly displayed (for example the accumulator is show connected to the ALU) and this helps users get a better understanding of how a processor works. |

### Assembly Training Program (ATP)

Assembly Training Program is an old windows application designed to teach assembly at higher levels. The application does not run on newer versions of Windows, however it comes with comprehensive documentation that outlines the specifics of the simulated processor and I will instead examine this.

The ATP processor is known as the BEP/16 which is a hypothetical 16 bit processor based roughly on the popular Intel 8086 microprocessor platform, the processor from which the x86 instruction set still in use today derives from. This makes it still an accurate representation of how a modern processor works, as even though major changes have been made over the years, the base instruction set has remained primarily the same.

The BEP/16 provides 10 general purpose registers as well as a full 16 bit Program Counter, Stack Pointer, Flag Register and Overflow register. The 16 bit nature of the Program Counter limits the total RAM size to at most 64K addressable bytes, however the processor also offers another 16K bytes of stack, both of which are accessed in 2 byte words. This configuration is incredibly flexible, with the general purpose registers allowing more complicated programs to be written by students. The amount of RAM is more than enough for the purposes of most students.

The flag register provides multiple useful flags. This includes the carry bit, zero flag, sign flag as well as an overflow flag. Each flag takes up a single bit of space and are updated as calculations are carried out to reflect the current state of the processor. For example, the zero flag will be set if the result of an operation in the ALU is 0, and be reset back to false if another operation takes place that produces a different result.

The 16 bit nature of the processor provides more than enough resources (65,535 integer values) for most calculations, with some arguing that it could be enough to actually overcomplicate matters. However, the next natural step down would be an 8 bit processor, which would be incredibly restrictive in only being able to hand unsigned integers of 0-255. A 32 bit processor would be greatly in excess of the requirements of the users and introduce new challenges in terms of the UI, as more data would have to be displayed.

The instructions are provided in the typical MODE-OPERATION-OPERAND format, with 2 bits for the memory mode, 6 bits for the operation (providing up to 64 possible operations), 4 and 4 bits for selecting the registers used in the operations with the remaining 8 bits used for the operand. This format is more complicated than that used in the Little Man Computer but provides much more flexibility to the programmer, certainly making it more appropriate for universities and sixth-forms as opposed to GCSE level.

Unlike the Little Man Computer, ATP provides access to a variety of addressing modes. This is useful as at A-Level addressing modes are taught as part of the specification, yet are missing from a common teaching tool. The BEP/16 offers 7 addressing modes, with not all of these available for all instructions. Direct, Indexed and Indirect are exclusively reserved for the STR and LDR register-RAM transfer commands.

ATP also provides a rudimentary assembler with preprocessing features. This allows the user to write assembler rather than the direct machine code that is executed by the simulated processor. The assembler is little more than rudimentary, providing a few simple #DEF operations allowing constant values to be easily set and reused within the program. Registers may also be provided an alias. This seems like a useful feature, even for new programmers, as it will allow them to write code that is more self-explanatory and easily understood by others.

Software Interrupt Routines (SWI) are also provided by the BEP/16 infrastructure and act as a way for the programmer to interact with an user. Calling an SWI triggers a context switch, with all the registers being pushed onto the stack bar R0 which is used for communication between the program and the interrupt routine. Four software interrupts are provided: getInt, putInt, getChar, putChar. These are a particularly useful part of the simulation that could have just been abstracted, however instead they have been implemented properly and this means that the user can get an understanding of how the software interrupt process works, something that is required at A-Level.

|  |  |
| --- | --- |
| Feature | Justification |
| Powerful instruction set | Allows the user to create more than just rudimentary programs and instead create more fully functioning programs. |
| Implementation of ISRs | Interrupt Service Routines are not defined in Little Man Computer but are required by the A-Level specification, the context-switching process could be clearly demonstrated. ISRs are also a useful way of the processor serving the requirement for input and output. |
| Appropriate Simplicity | Assembly Training Program strikes a careful balance in ensuring that the user is not overwhelmed with an entirely complete instruction set, instead only the core, most important, parts are implemented. |
| Multiple addressing modes | ATP provides multiple addressing modes to the user, something that is mentioned at A-Level, and this allows the user to write more efficient programs with, for example, indexed addressing. |
| Assembler with directives | The assembler within ATP supports useful directives for the developer. These speed up the programming process and allows constant values to easily be changed across the application. |

## Limitations

No project is without a scope and it is always important to have realistic expectations of what an application will be able to within the deadline and budget. The program will target the education market and I have drafted several relevant limitations to the project.

### Direct execution of compiled code

Whilst ArmSim provides the ability to run native code that has been compiled by GCC and other similar industry compilers, our application will not. There is limited use of this feature to students and it would require implementing the entirety of a comprehensive instruction set (e.g x86\_64 with 981 base instructions and 3683 variations of those [1] ), which is simply unrealistic within the deadline set.

### Ability to run on a wide variety of platforms

For my application I will most likely be using a .net based language and these executables do not natively run on operating systems other than Windows. Whilst it is possible to port them to compile with Mono (A .net alternative for GNU/Linux), this is simply not possible within the deadline set for the project.

### The entirety of an existing instruction set

Most of an existing instruction set will not be used by students day to day and to implement it within the timeframe set for the project would be unrealistic. Instead I will try to make my own instruction set in the style of x86 to allow people to get the gist of existing instruction sets

### Alternative architectures

My software will simulate a Von Neumann-esque processor. It would be too time consuming to implement multiple different architectures and would provide little advantage as students are not required to understand the exact workings of other architectures.

## Software and Hardware Requirements

It is reasonable to set out expectations for the kind of hardware and software required to be able to run my application. Choosing the frameworks and language to be used is often difficult, as no language is capable of dealing with all problems and choosing the wrong one can lead to more time being spent trying to work around a problem.

### Software

I have decided to choose C# with the .Net Framework as the language to use for my application development. Whilst it is not my favorite or most comfortable language, it is well suited to the problem for a variety of reasons:

#### Targeted Desktop Development

C# and the .Net Framework (Visual C#) was originally targeted at desktop development and provides a wide variety of useful classes for visual components (buttons, text boxes, panels etc) with a unified styling. This will allow me to produce a professional looking application quickly.

#### Fast

Once compiled C# is much faster than interpreted languages such as Python. This means it will be able to simulate more processor cycles each second, giving the user more flexibility. Of course it is not as fast as C++ and other lower level languages, but this is at the cost of usability when trying to rapidly prototype an application.

#### Type Safe

C# provides strict static typing as well as tight access control and this makes it an incredibly type safe language. This helps the developer, especially in larger projects, produce an overall more stable and predictable program that is not vulnerable to unexpected issues caused by dynamic types and casting. Stability is an incredibly important aspect of any commercial application, especially those where a user could lose work if the application crashes.

#### Comprehensive Tooling

Microsoft provides a comprehensive set of tools through its Visual Studio IDE that includes advanced debugging with stepping and watching, static code analysis and a WYSIWYG editor. This allows the application to be rapidly produced with easy results, which is important considering the short deadline on the project.

### Hardware

Hardware specification is something of a dark art as it can be hard to predict exactly how an application will perform until you have benchmarked it. However, rough guidelines can be drawn up from what we would expect from the average C# .Net application. Fortunately, most schools provide computer science labs with higher specification machines so that they can run virtual machines etc and this provides us more room to work within.

Minimum requirements:

* Dual Core processor at 2.6 Ghz
* 4 Gigabytes of RAM
* Free disk space of 10GB
* Integrated graphics capability

Recommended requirements:

* Quad core processor at 3 Ghz
* 8 Gigabytes of RAM

## Features and Requirements

### Essential Features

|  |  |
| --- | --- |
| Feature | Justification |
| Instruction set with enough depth for complex programs | My program must be able to allow students to challenge their own abilities by designing more complicated programs, software such as LMC simply does not have the depth required to allow students to stretch themselves adequately. |
| Simple and easily grasped user interface | The user interface must be quick to understand for new users. The interface should be accessible to a wide range of understandings and not confuse new users by presenting too much information. |
| The ability to modify the speed of the simulation | The user must be able to change the speed of the simulation in order to switch between wanting to understand the Fetch-Decode-Execute cycle on the microscopic level to wanting to write more complicated algorithms. |
| Accurate simulation of a processor according to the A-Level and GCSE specifications. | The processor should be designed to demonstrate the entirety of a processor according the A-Level specification with adequate depth to allow students to push themselves. There is no need for the processor to perfectly simulate an existing, more complicated, architecture. |
| A selection of example programs | Example programs for common algorithms should be included to demonstrate the abilities of the simulation as well as to help teach the user how to use the provided assembly. It would also be useful for users to learn how many of the core algorithms to Computer Science work. |
| Ability to save and restore the state of the processor | The user should be able to save the processor and the code currently stored in it and then restore this at a later date. |

|  |  |
| --- | --- |
| Requirement | Justification |
| Sliding scale for adjusting the speed of the simulation. | A sliding scale is easy for a user to adjust, and precise values are not required. It is intuitive and simple to understand, rather than requiring the user to enter a specific clock speed to use. |
| Visualization of the register contents of the PC, MAR, CIR, MDR, ACC | These registers are an important part of the specification and understanding of how a processor works, therefore they should be visually distinct from the general-purpose registers. This makes the interface easy to understand. |
| Example Bubble Sort and Binary Search | Bubble sort and Binary Search are of significant complexity to aptly demonstrate the abilities of the processor and instruction set. They are also key algorithms included in the Computer Science course, so users may already be familiar with them. |
| The editor must highlight unknown mnemonic, as well as cases where the mnemonic is not followed by the correct amount of parameters. | This simple analysis of the code written by the user will sufficiently catch a majority of mistakes made and assist them in writing correct assembly first time. It will not be too difficult to implement. |
| TO BE EXPANDED |  |

### Requirements

### Success Criteria

|  |  |
| --- | --- |
| Success Criteria | Justification |
| Sliding scale should correctly adjust the speed of the simulation from 1Hz to at least 100Hz | 1 Hz to 100 Hz provides an adequate range of test values to ensure that the simulation correctly scales to the different speeds. There shouldn’t be much need from users to run it faster. |
| The contents of the PC, MAR, CIR, MDR, ACC should be displayed accurately and real-time, ensuring the view and model in sync. | It is important that elements of the interface accurately represent the state of the model being simulated in the back-end. |
| The contents of the PC, MAR, CIR, MDR, ACC should be adjustable via the interface and this adjustment should have the correct effect on the simulation. | Bubble sort and Binary Search are of significant complexity to aptly demonstrate the abilities of the processor and instruction set. They are also key algorithms included in the Computer Science course, so users may already be familiar with them. |
| TO BE EXPANDED |  |

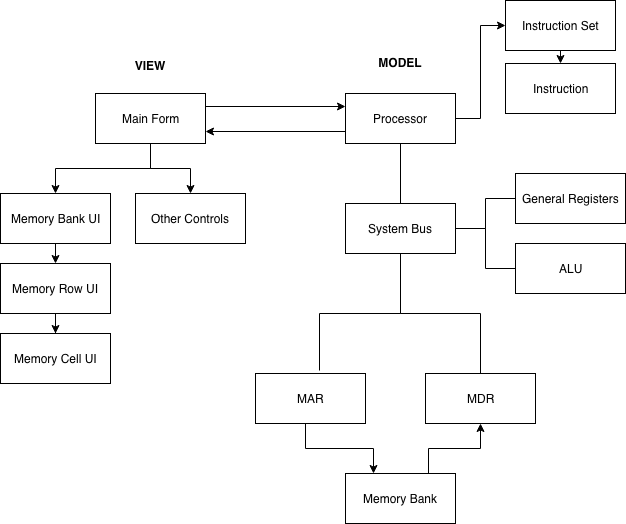
# Section 2: Design

Designing a solution before developing it is incredibly important to ensure that development remains a smooth and efficient experience and that nothing important is excluded. A well-planned program is far less likely to include bugs and unexpected behavior

## Decomposing the Problem

From the highest level of decomposition my application will be divided into two parts: the view and the model. This architecture means that both are individually testable and it can make forms of automated testing like unit-tests easier to design. These two halves will then be divided further into smaller individually testable classes.

By keeping a strong separation of concerns, it is clear where each behavior of the application is expected to reside. In larger projects with multiple developers, this style of splitting the view and the model is common, with certain developers being responsible for their half of the program. Below is a diagram that demonstrates roughly how each component (or class) will likely interact, with greater details to be included later.



### Designing the Processor Class

One of the most complicated parts of the application is the processor class. I decided to break the processor down into several independent classes that could then centrally accessed as parts of processor. This style of parent-child entity relationship is incredibly useful as it means that only a single reference needs to passed to each of the children entities in order to allow them to access the other sibling entities. This kind of access is expected to occur rather a lot considering the architecture of the processor.

#### Properties and Methods

The following table shows the expected properties, methods etc of the Processor class as well as an explanation and a reason for the access protection.

|  |  |  |
| --- | --- | --- |
| Type | Name | Explanation |
| Private InstructionSet | InstructionSet | This special class will contain all of the instructions it is possible for the processor to execute.  It is private as other objects will not need to access it. |
| Public Register | CIR | This will contain a reference to the register that will contain the currently executed instruction for access by the “control unit” functionality within the processor.  It is public as it needs to be read by the UI. |
| Public Register | PC | This will contain a reference to the register that will contain the memory location of the instruction that will next be fetched and executed. It will be manipulated as part of Branching instructions.  It is public as it needs to be read by the UI. |
| Public Register | MDR | This will contain a reference to the register that will contain the data that is being written to memory or that has just been read from memory.  It is public as it needs to be read by the UI. |
| Public Register | MAR | This will contain a reference to the register that will contain the memory address that is being read from or being written to.  It is public as it needs to be read by the UI. |
| Public Register | ALU | This will contain a reference to the register that will contain the results of mathematics completed in the Accumulator. This is a special register as it can be manipulated by the Accumulator without using the system bus.  It is public as it needs to be read by the UI. |
| Private Accumulator | ACC | This will contain a reference to the Accumulator, which will complete any arithmetic within the processor.  This can be private as it does not need to be read by the UI. |
|  |  |  |
| Public Array<Register> | RegisterBank | This array contains a reference to all of the general-purpose registers as well as any of the special registers that we want the user to be able to access. An array is used as we know that it will only contain objects of a specified class and we do not expect the length to change.  It is public as this needs to be accessed by the UI. |
| Public Boolean | Fetching | This public Boolean shows whether the processor is currently in the fetch or execute part of the cycle. Decode has been omitted as a matter of simplicity.  It is public because this needs to be accessed by the UI. |
| Public Event | ClockRising | This event will allow the various sub-components of the processor to know when the correct time is to write to the bus.  It is public because the components need to be able to hook onto it. |
| Public Event | ClockFalling | This event will allow the various sub-components of the processor to know when the correct time to read from the bus and perform computations is.  It is public because the components need to be able to hook onto it. |
| Public Event | UpdateUI | This event will be used to notify the UI components that they need to update their values based on the model as any computation for that clock cycle has been completed.  This is public as the UI components need to hook onto it. |
| Public Void Method | Clock() | The Clock() method is used to iterate the processor through a single clock cycle. This will be called by a timer, or when a user manually clicks a button to cycle through the |

#### Algorithms

There are two primarily algorithms held within the Processor class. Clock() which forms the basis of the workings of the entire processor and calls the fetch routine or the fetched instruction.

CLOCK:

IF PROCESSOR IS FETCHING THEN

EXECUTE FETCH ROUTINE

ELSE

EXECUTE SELECTED INSTRUCTION.

IF INSTRUCTION IS COMPLETE, FETCHING = TRUE

END

TRIGGER CLOCK RISING EVENT

TRIGGER CLOCK FALLING EVENT

TRIGGER UI UPDATE

This algorithm remains simple as the more complicated logic is moved away into the instructions themselves. This makes debugging easier as it means it is more likely a problem with the individual instruction than the dispatching logic.

The other algorithm is the fetching instruction:

FETCH:

SET MAR TO PROGRAM COUNTER

INCREMENT PROGRAM COUNTER

TELL MEMORY TO READ TWO BYTES

READ MDR INTO CIR

SET MAR TO PROGRAM COUNTER

INCREMENT PROGRAM COUNTER

TELL MEMORY TO READ TWO BYTES

READ MDR INTO CIR SECOND WORD

As this is written in micro-code, it can only consist of simple processor operations such as register read/writes and ALU arithmetic.

#### Validation

No validation will occur within the processor class, as validation will only occur in the view part of the application.

#### Test Data

With just the code for the processor written, very little testing will be possible. The only test that conclusively proves that the processor works correctly will be allowing it tick over and ensuring that the program counter cleanly increments by 2 every other processor cycle. Further testing can be completed once the instructions are complete, as testing these will test the processor class itself.

### Designing the Register Class

The register class is the base class for all types of registers in the processor with a variety of lengths and purposes, from the generic user-addressable single word registers to the double word special current instruction register.

#### Properties and Methods

The following table shows the expected properties, methods etc of the register class as well as an explanation and a reason for the access protection.

|  |  |  |
| --- | --- | --- |
| Type | Name | Explanation |
| Private Bus | \_bus | This variable will reference the bus that the register’s data input and output ports are connected to.  It is private as other objects will not need to access it. |
| Private Array<Byte> | \_value | This byte array contains the data held by the register. It can be accessed through a getter and setter, which means that the data array itself is private but can be accessed in a controlled manner. |
| Private Processor | \_proc | This variable will reference the processor that the register is connected to.  It private as no other objects will need to access it. |
| Private Void Method | ClockRisingHandler | This handler is called when the ClockRising event is propagated. |
| Private Void Method | ClockFallingHandler | This handler is called when the ClockFalling event is propagated. |
| Public Void Method | SetFlags | This method is called by other objects in order to control the operation of the Register. Hence it needs to be public. |

#### Algorithms

There are two interesting algorithms in the register.

CLOCK RISING HANDLER:

IF REGISTER READ FLAG IS TRUE:

IF REGISTER TWO BYTE FLAG IS TRUE:

WRITE 2 BYTES TO THE BUS

ELSE:

WRITE 1 BYTE TO THE BUS

CLOCK FALLING HANDLER:

IF REGISTER WRITE FLAG IS TRUE:

IF REGISTER TWO BYTE FLAG IS TRUE:

READ 2 BYTES FROM BUS

ELSE:

READ 1 BYTE FROM BUS

Both algorithms make use of the control flag system that will be set by the instructions in order to control the register. These flags will exist as part of an enum setup for flag operation. This keeps the code concise and readable.

#### Validation

No validation will occur within the register class, as validation will only occur in the view part of the application.

#### Test Data

With just the code for the processor and registers written, very little testing will be possible. Further testing of the registers will be completed when the instructions are tested, as if the register does not behave correctly, it will become obvious when the instruction consequently misbehaves.

### Designing the ALU Class

The ALU class is designed specifically for the instantiation of a single object, and therefore we do not need to worry so much about ensuring it remains generic and flexible.

#### Properties and Methods

The following table shows the expected properties, methods etc of the ALU class as well as an explanation and a reason for the access protection.

|  |  |  |
| --- | --- | --- |
| Type | Name | Explanation |
| Private Bus | \_bus | This variable will reference the bus that the ALU’s data input and output ports are connected to.  It is private as other objects will not need to access it. |
| Private Register | \_acc | This variable will reference the special register that the results of the ALUs calculations are stored in. This is known as the accumulator.  It is private as other objects will not need to access it. |
| Private Processor | \_proc | This variable will reference the processor that the ALU is connected to.  It private as no other objects will need to access it. |
| Private Void Method | ClockFallingHandler | This handler is called when the ClockFalling event is propagated. In this case it will perform the calculations and output the result into the accumulator. |
| Public Void Method | SetMode | This method is called by other objects in order to control the operation of the ALU. Hence it needs to be public. |

#### Algorithms

The ALU is much simpler and has one main algorithm which is called on the clock fall event.

CLOCK FALLING HANDLER:

VAR busValue IS CONVERT BUS TO INTEGER

VAR accValue IS CONVERT ACCUMULATOR TO INTEGER

VAR val IS 0

SWITCH ALU MODE:

CASE Increment THEN val IS busValue + 1

CASE IncrementWord THEN val is busValue + 2

CASE Add THEN val IS busValue + accValue

CASE Subtract THEN val IS accValue – busValue

ACCUMULATOR SET TO val AS BYTE ARRAY

This algorithm forms the main functionality of the ALU component. Due to the use of an enum for aluMode, it is possible to use a simple and semantically clear switch statement to decide what operation needs to be completed on that tick. Again, we see the flexible flag system being used to allow the rest of the processor to control the individual unit.

#### Validation

No validation will occur within the ALU class, as validation will only occur in the view part of the application.

#### Test Data

With just the code for the processor and registers and ALU written, very little testing will be possible. Further testing of the ALU will be completed when the instructions are tested, as if the ALU does not behave correctly, it will become obvious when the instruction consequently misbehaves. In particular, attention should be aimed towards the load cycle, ADD and SUB instructions. Some attempts should be made to make the ALU behave in an unexpected way.

### Designing the Memory Bank class

Much like the ALU, the memory bank will only be instantiated once. It would perhaps make sense to have it as a static class, or to make use of a singleton structure, however, this is most likely overkill on a smaller project where there is no added benefit to a singleton beyond cleanliness of code. This single instantiation means less care needs to be given to the idea of building a class as generic as possible.

#### Properties and Methods

The following table shows the expected properties, methods etc of the MemoryBank class as well as an explanation and a reason for the access protection.

|  |  |  |
| --- | --- | --- |
| Type | Name | Explanation |
| Private Register | \_mar | This variable will reference the special register that stores the address that the MemoryBank should read/write into.  It is private as other objects will not need to access it through the MemoryBank. |
| Private Register | \_mdr | This variable will reference the special register that stores the data should be written to memory or the data that has been retrieved from memory.  It is private as other objects will not need to access it through the MemoryBank. |
| Private Processor | \_proc | This variable will reference the processor that the MemoryBank is connected to in the child-parent hierarchy.  It private as no other objects will need to access it. |
| Private Void Method | ClockFallingHandler | This handler is called when the ClockFalling event is propagated. In this case it will perform a memory read/write ready for the rising edge of the next clock cycle. |
| Public Void Method | SetFlag | This method is called by other objects in order to control the operation of the MemoryBank. Hence it needs to be public. It adjusts a flag based enumerable value stored in the object. |

#### Algorithms

The MemoryBank also operates only on the Clock Falling handler and hence operates simply through a single algorithm that is similar to the ALU. However, the MemoryBank had added complexity as it has to deal with four potential circumstances: Read, Write, Read 2 Bytes, Write 2 Bytes. These are indicated using a Read/Write flag and a One Byte/Two byte flag.

CLOCK FALLING HANDLER:

VAR busValue IS CONVERT BUS TO INTEGER

VAR accValue IS CONVERT ACCUMULATOR TO INTEGER

VAR val IS 0

SWITCH ALU MODE:

CASE Increment THEN val IS busValue + 1

CASE IncrementWord THEN val is busValue + 2

CASE Add THEN val IS busValue + accValue

CASE Subtract THEN val IS accValue – busValue

ACCUMULATOR SET TO val AS BYTE ARRAY

#### Validation

No validation will occur within the ALU class, as validation will only occur in the view part of the application.

#### Test Data

With just the code for the processor and registers and ALU written, very little testing will be possible. Further testing of the MemoryBank will be completed when the instructions are tested, as if the MemoryBank does not behave correctly, it will become obvious when the instruction consequently misbehaves. In particular, attention should be aimed towards the direct memory mode function of all instructions as well as towards the more specific instructions LDA and STA.

### Designing the Main Form Class

The main form is how users will typically interact with the system. Whilst algorithmically it will remain simple, this class will be the entrypoint for a majority of the user facing testing, and hence special attention should be paid to the test table.

#### Properties and Methods

The following table shows the expected properties, methods etc of the MemoryBank class as well as an explanation and a reason for the access protection.

|  |  |  |
| --- | --- | --- |
| Type | Name | Explanation |
| Private Method | Update UI Handler | This method is setup as a hook onto the processors UpdateUI event. This signifies to the form that the view needs to be re-rendered in order to match the state of the model.  It is private as it does not need to be called by other objects. |
| Private Method | Clock Rate Value Changed Handler | The method is a hook onto the value changed event for the clock rate entry box. Here we can perform appropriate validation and update the timer mechanism so that the processor is actually affected.  It is private as it does not need to be called by other objects. |

#### Algorithms

The Update UI handler is not algorithmic in sense that no logic/conditions are applied to it. Therefore, there is no need to write pseudocode for it. The clock rate change handler does include some validation and therefore should be preplanned. The handler will be attached to a number scroller, so we don’t need to handle all scenarios as these are validated within the .net class instead. The only real interesting point shown within this pseudo-code is my preference for return based exits within specific conditions rather than using an if-else statement. This reduces nesting of code and resultant complexity.

CLOCK RATE VALUE CHANGED HANDLER:

SET lblClockrate’s Text TO STRING OF txtClockRate Value

IF txtClockRate Value IS 0 THEN

DISABLE TIMER

RETURN

END

ENABLE TIMER

SET TIMER INTERVAL TO (1/txtClockRate Value) \* 1000

#### Validation

The following table shows the validation that will be required for parts of the main form not covered else where

|  |  |  |
| --- | --- | --- |
| Element | Acceptable Range | Justification |
| txtClockRate | 0 to 100 inclusive | 0 allows the user to disable the auto-clocking feature. Values over 100 could place strain on the application and the user’s system, and pose no real educational value to the user. Nonsensical values could cause the system to crash without proper validation. |

#### Test Data

Since validation occurs, some testing is required to ensure that this meets expectations and behaves correctly.

|  |  |  |
| --- | --- | --- |
| Test | Expected Value or Behaviour | Justification of Test |
| Entering -1 into the txtClockRate | Field resets to 0. | Invalid values should be set to the closest valid value. This is intuitive to the user and prevents crashes. |
| Entering 50 into the txtClockRate | Field sets to 50 and clock ticks at 50hz. | Ensure that valid values are accepted to prevent a frustrating user experience. |
| Entering 0 into the txtClockRate | Fields sets to 0 and the timer is disabled. | Ensure that key part of functionality works correctly and validation allows correct range of values. |
| Entering 100 into the txtClockRate | Field sets to 100 and the clock ticks at 100hz. | Ensure that valid values at the extremities are correctly accepted. |
| Entering ‘azert|’ into the txtClockRate | Field remains at previous value or sets to 0. | Ensure that completely nonsense values do not crash the application. |

### Designing the Memory Bank Cell Component Class

The individual cell of memory represents another place where validation on user entered content is completed. The memory bank cell also acts as the final point of UI update propagation, as it sources the data directly from the memory bank when it is triggered to update. It also directly interacts with the memory bank in order to allow the user to set the value of a memory cell.

#### Properties and Methods

The following table shows the expected properties, methods etc of the MemoryBank Cell class as well as an explanation and a reason for the access protection.

|  |  |  |
| --- | --- | --- |
| Type | Name | Explanation |
| Public Method | Setup | This method is called as the very end of the propogation tree from MemoryBank Control down ton MemoryBank Row and finally MemoryBank cell. It signals the class to update its displayed value to match a changed value in the MemoryBank.  It is public as it needs to be called by other objects, specifically the MemoryBank Row. |
| Private Method | Value Changed Handler | The method is a hook onto the value changed event for the memory bank value entry. It performs appropriate validation before applying. The change.  It is private as it does not need to be called by other objects. |

#### Algorithms

Both algorithms included are relatively simple, although the second involves some element of validation that will need to be tested.

SETUP:

SET addressLbl Text TO MemoryAddress LEFT PADDED TO 3 LENGTH WITH 0

SET valueLbl Text TO MemoryAddress MemoryBank VALUE

VALUE CHANGED HANDLER:

IF (newValue LENGTH IS GREATER THAN 0) THEN

TRY

CAST newValue TO BYTES

CATCH

WARN USER

RESET valueBox Value

END TRY CATCH

END IF

#### Validation

The following table shows the validation that will be required for parts of the main form not covered else where

|  |  |  |
| --- | --- | --- |
| Element | Acceptable Range | Justification |
| txtValue | Any valid binary number up to a byte long. | Any memory location can store up to one byte of binary. Allowing the system to attempt other values would cause crashes or incoherent behavior. |

#### Test Data

Since validation occurs, some testing is required to ensure that this meets expectations and behaves correctly.

|  |  |  |
| --- | --- | --- |
| Test | Expected Value or Behaviour | Justification of Test |
| Entering 11110000 | Memory at address set to 11110000 | Correct values should be accepted without fault. |
| Entering 1111 | Field should be normalized to 00001111 due to left padding and the memory at address should be set to this. | Values that are valid but not normalized should be normalized to maintain consistency across the UI. |
| Entering 111111111 | Field resets to previous value. | Numbers that require over a byte should not be allowed as this will crash the system. |
| Entering ‘shibboleth’ | Field resets to previous value. | Ensure that completely nonsense values do not crash the application. |

### Designing the Instructions

The most algorithmically intensive section of the project was the instructions themselves. These would also require comprehensive testing.

#### Properties and Methods

The following table shows the expected properties, methods etc of the InstructionSet class as well as an explanation and a reason for the access protection.

|  |  |  |
| --- | --- | --- |
| Type | Name | Explanation |
| Public Array<Instructions> | Instructions | This array contains all possible instructions and permutations in a neat fashion, making the InstructionSet class a wrapper over the array.  It is public as it needs to be accessible to other objects such as the processor. |

#### Algorithms

Each instruction has its own specific algorithm, with variants for each memory mode.

NOP:

END

ADD:

SET ACCUMULATOR TO REGISTER A’s VALUE

IF MEMORY MODE IS DIRECT THEN

SET MAR TO VALUE FROM CIR SECOND WORD

SET MEMORY READ MODE TO FULL WORD

CLOCK CYCLE

READ MDR VALUE TO BUS

ELSE IF MEMORY MODE IS REGISTER THEN

READ REGISTER B’S VALUE TO BUS

ELS EIF MEMORY MODE IS IMMEDIATE THEN

READ CIR SECOND WORD TO BUS

END IF

SET ALU MODE TO ADD

CLOCK CYCLE

SET REGISTER A TO ACCUMULATOR VALUE

SUB:

(Same as above with ALU MODE as SUBTRACT)

BRA:

IF MEMORY MODE IS DIRECT THEN

SET MAR TO VALUE FROM CIR SEOCND WORD

SET MEMORY READ MODE TO FULL WORD

CLOCK CYCLE

SET PROGRAM COUNTER TO MDR VALUE

ELSE IF MEMORY MODE IS REGISTER THEN

SET PROGRAM COUNTER TO REGISTER B’s VALUE

ELSE IF MEMORY MODE IS IMMEDIATE THEN

SET PROGRAM COUNTER TO CIR SECOND WORD

END IF

BRZ:

(Same as above with IF REGISTER A’S VALUE IS 0 WRAPPING METHOD)

BRP:

(Same as BRA with IF REGISTER A’S VALUE IS GREATER OR EQUAL TO 0)

MOV:

IF MEMORY MODE IS REGISTER THEN

SET REGISTER A’s VALUE TO REGISTER B’s VALUE

ELSE IF MEMORY MODE IS IMMEDIATE THEN

SET REGISTER A’s VALUE TO CIR SECOND WORD VALUE

END IF

STA:

IF MEMORY MODE IS REGISTER THEN

SET MAR TO REGISTER B’s VALUE

ELSE IF MEMORY MODE IS IMMEDIATE THEN

SET MAR TO CIR SECOND WORD VALUE

END

SET MDR’s VALUE TO REGISTER A’s VALUE

SET MEMORY TO FULL WORD WRITE

LDA:

IF MEMORY MODE IS REGISTER THEN

SET MAR TO REGISTER B’s VALUE

ELSE IF MEMORY MODE IS IMMEDIATE THEN

SET MAR TO CIR SECOND WORD VALUE

END

SET MEMORY TO FULL WORD READ

CLOCK CYCLE

SET REGISTER A’s VALUE TO MDR’s VALUE

#### Validation

No validation occurs as part of the InstructionSet class.

#### Test Data

A wide range of tests will need to be completed as part of testing the instruction set.

|  |  |  |
| --- | --- | --- |
| Test | Expected Value or Behavior | Justification of Test |
| NOP | No changes to register or memory values. | Ensures there are no side effects of what should be a instruction with no effects. |
| ADD 1 TO REG 0 | Reg 0 should have a value of 1 | This tests register, ALU and instruction operation. |
| ADD 1 TO REG 0  Ran Twice | Reg 0 should have a value of 2 | This tests register, ALU and instruction operation. In particular it ensures register reading words correctly. |
| ADD MemoryLoc 16 (with value of 1) to REG 0  Ran Twice | Reg 0 should have a value of 2 | This tests register, ALU and instruction operation. In particular it ensures memory and register reads work correctly. |
| SUB MemoryLoc 16 (with value of 1) to REG 0  Ran Twice | Reg 0 should have a value of -2 | This test builds upon the previous one to ensure that subtraction works as expected. We do not need to run all the tests as the code is almost identical, simply running the most likely to fail is sufficient. |
| ADD 1 TO REG 0  Ran Twice  MOV REG 0 to REG 1 | Reg 1 should have the value of 2 | This test builds on previous testing to ensure that the MOV instruction functions correctly and reinforces trust in the ADD instructions and register operation. |
| MOV 256 to REG 0 | Reg 0 should have the value of 256 | This test ensures that the Immediate mode for the MOV instruction functions as expected. |
| MOV 256 to REG 0  MOV 16 to REG 1  STA REG 0 IN REG 1 | Memory location 16 should have the value of 256 stored. | This test reinforces trust in the MOV operations as well as testing the STA instruction. |
| MOV 256 to REG 0  STA REG 0 IN 16 | Memory location 16 should have the value of 256 stored. | This test reinforces trust in the MOV operations as well as testing the STA instruction in the Immediate memory mode. |
| (set the value of memory location 16 to 256)  LDA 16 TO REG 0 | Reg 0 should be set to the value 256 | This test ensures that the memory bank works correctly as well as registers and the LDA command in immediate memory mode. |
| (set the value of memory location 16 to 256)  MOV 16 TO REG 1  LDA REG 1 TO REG 0 | Reg 0 should be set to the value 256 | This test tests LDA operating in Register memory mode as well as reinforcing trust in the memory bank, registers and MOV command. |

# Section 3: Development

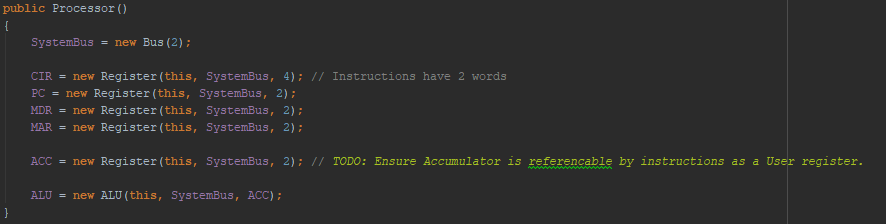
I decided to break my development down into several individually testable stages. At each point I would be able to review the functionality of the code I had just written, as well as retesting code that this code relied on to prevent regression.

At the end of each stage of development, following a brief description of the functionality of the code written, I will include the test tables detailing the test, expected result and actual result as well as any debugging and bug-fixing that occurred as a result of the testing.

## Wire Framing the Classes

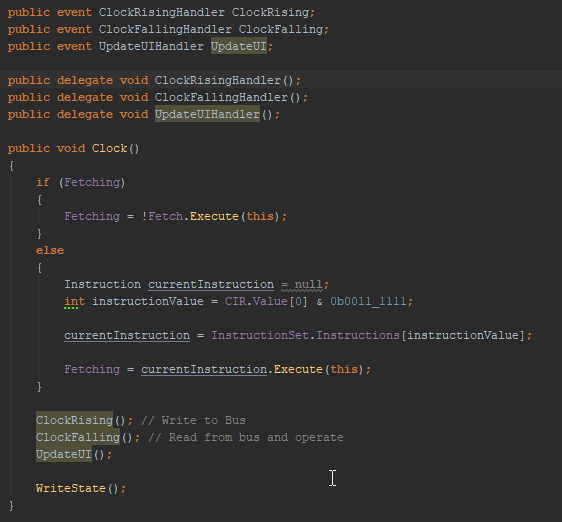
I started off development by creating a handful of classes and wire-framing their properties and methods. Whilst the code written would not reflect the final implementation of the classes, it would be enough to see how it would fit together and evaluate how effective the structure would be.

The first class I created was the processor class. This class would consist of other components and tie them together in a hierarchal structure. The processor class would act as the model that would eventually be interacted with and displayed through the view layer. By passing the processor to the sub-components, they would be able to easily access other components. This technique is clear within the constructor of the processor class:



Here you can see the instantiation of a variety of registers, linking them to the processor, system bus and specifying their width in bytes.

The processor works on an event-driven model, with each Clock tick triggering three events ClockRising, ClockFalling and UpdateUI. These events are listened to by the individual components which then act accordingly. I decided to use events as they provide a flexible way of triggering methods across a range of dissimilar objects.



Here the three events and their corresponding delegates can be seen. These are scoped as public because the other classes (the view and the components) also need to be able to hook onto these events.

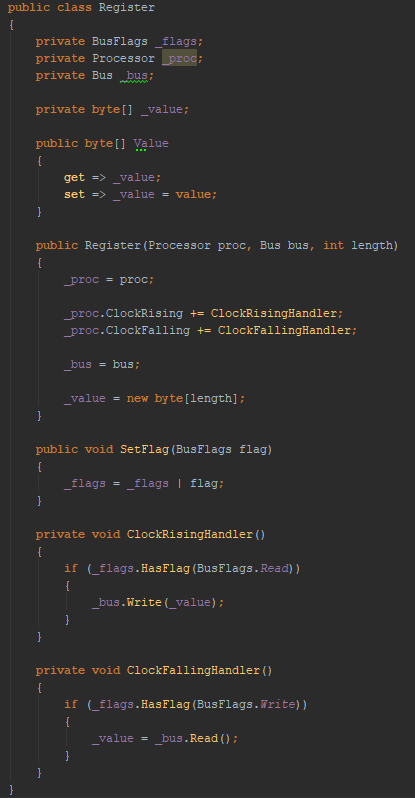
A rough implementation of the Clock method can also be seen. The Fetching flag simply refers to if the application is in the Fetching or Executing part of the cycle. The instruction class provides a simple Execute method which returns True or False whether or not the instruction has completed each of its micro-code steps, this will become clearer later in this section when the Instruction class is shown. Also notable is the inclusion of binary masking using the AND operator to select the operation code part of the instruction.

The final part of the processor class is the hard-encoded Fetch instruction. Whilst the instruction set will be inter-exchangeable, the Fetch instruction is not available to the user and instead used only internally by the processor. I drafted the fetch instruction based on the specification I processed for the EDP.



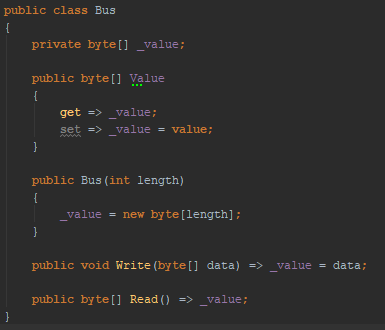
Each stage in the array represents a microcode instruction that sets the flags on each of the registers and components. The Fetch instruction is interesting because it has to retrieve two words from the memory. This means two memory operations are completed, one for the first word and one for the second. A rough comment is included for each step that shows the transfer of data between registers.

I then had to write the basic implementation for the registers. Like all other bus components they would be designed to write to the bus on the rising edge and read from the bus on the falling edge. This emulates an actual system where it takes time for the signal to physical propagate across the bus.



The class is relatively simple. During construction it hooks onto the rising and falling events and initializes the internal array. I decided that byte arrays would adequately fulfil the needs of the application as they can be indexed at each byte, just how actual memory would work. A flag enum is used for controlling the state that mimics the behavior of control lines in the actual processor.

The bus component is even simpler with the registers simply directly writing to it on the clocks rising edge. If more than a single register writes to it, then it will be overwritten and not merged as we might expect to see on actual physical hardware, although this is an edge-case as two registers should not simultaneously write to a bus.



As part of allowing further extensibility, there are two ways of writing or reading to the bus. The value can either be directly interacted with via a property or via two functions. The Write and Read functions may eventually provide more complicated logic for correctly handling a register with a shorter width writing to the bus.

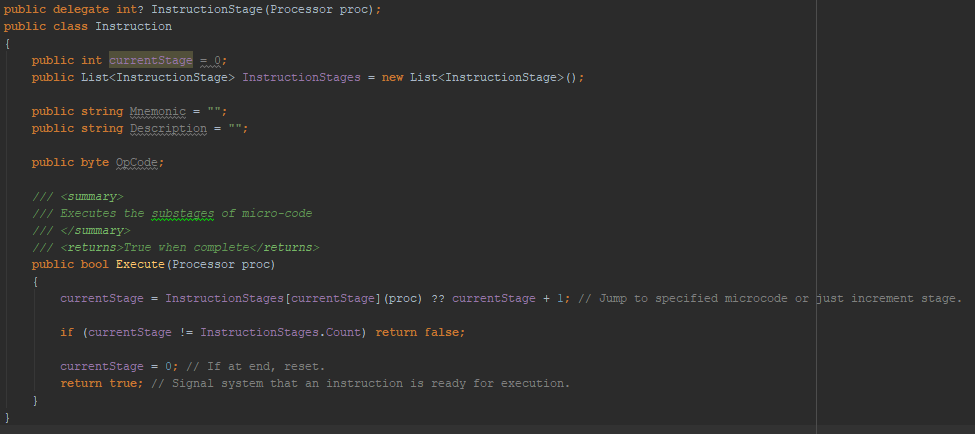
The final class needed as part of the processor components was going to be the ALU. This would need to interact with the bus as well as directly accessing the ACC. I decided to implement a basic add function, as well as two specialized ones that increment in word lengths. Similar specialist hard-coded functions can be found in modern processors, allowing the fetch-decode part of the cycle to take up less of the overall processor time.



The ALU acts similarly to the rest of the bus components, being driven by the two clock edges as well as being primarily configured via a set of flags. One primary difference is that it has a direct interaction with the accumulator without having to cross any busses. This is convenient and allows the processor to complete basic arithmetic in less cycles.

For ease of development, I decided to cast the byte-arrays to Int16s for the purposes of arithmetic. This meant I did not need to write the underlying logic of the ALU, as this would have been time consuming for little additional benefit.

The final classes I wire-framed were the abstractions for the Instruction and Instruction Set.

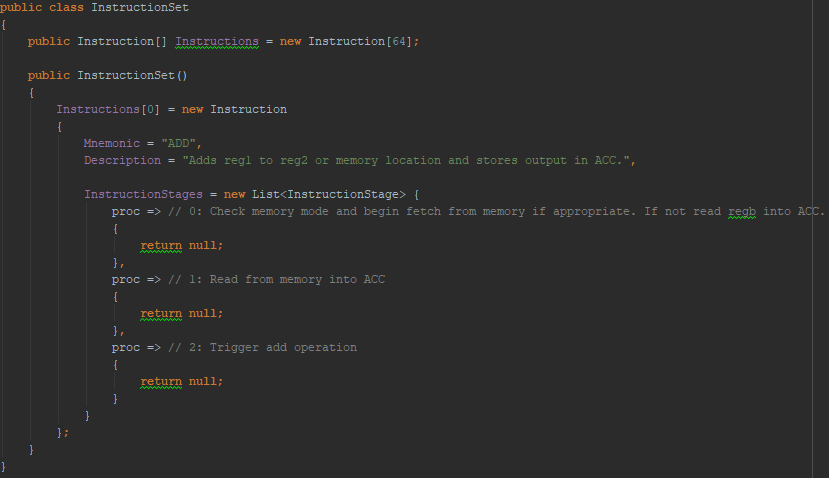


The instruction class represents a single assembly instruction. It is used both by the processor for processing the instruction via the microcode steps provided in InstructionStages and will also later be used by the assembler when converting from mnemonics to bytecode.

The execute function simply steps over each of the microcode steps and returns a true signal to show that the entirety of the microcode for that function has been completed. If it returns false, the processor should continue to execute that instruction until all of the microcode has been executed. Returning true will usually force the processor to return back to the fetching state.

The instruction set class acts as a wrapper around an array. It contains each of the possible instructions. The advantage of making this a class is that we are able to offer multiple instruction sets in the future that can easily be swapped out in both the assembler and the processor. This may allow an instruction set that mimics RISC like behavior and an instruction set that mimics more CISC like behavior with more comprehensive instructions.

Alternatively this could have been implemented with a static class, but this would not have offered the ability to change out the instruction sets so easily. I chose an array as the datatype as the instruction set will not grow in size during execution and the array type allows single operation reads to any element of the array.



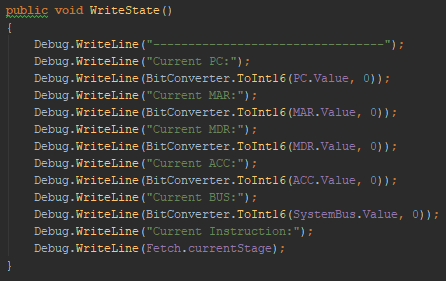
### Testing

The scope of the testing as this stage is limited as the application is very simple, but it is useful to test the initial stages to ensure they are correctly implemented before building upon them.

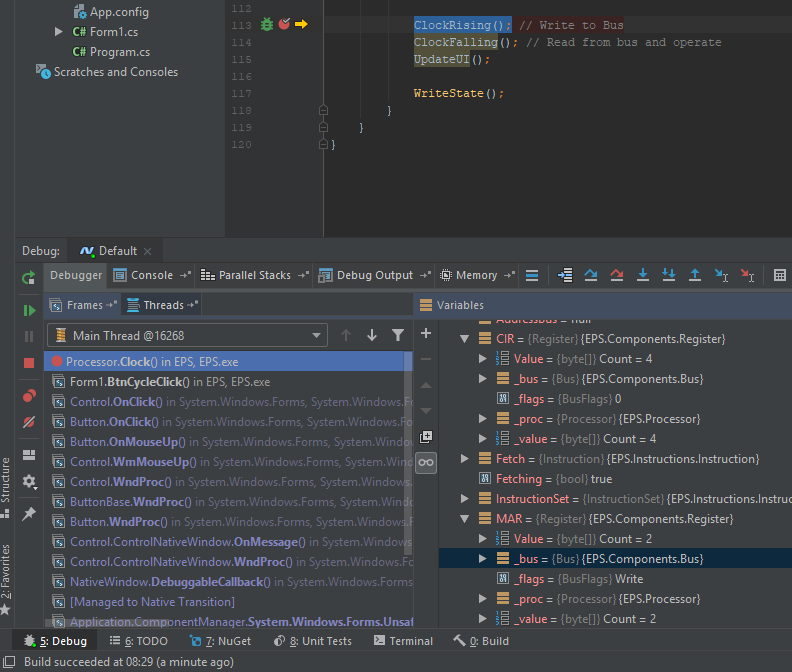
|  |  |
| --- | --- |
| Test | Expected Result |
| Allow the application to tick through the fetch cycle repeatedly | We expect to see the 5 distinct stages of the fetch cycle being correctly executed. This can be seen with the contents of the registers. For example, we expect at the end of each cycle for the Program Counter to have incremented by 4. |

### Bug Fixes

During testing I noticed some unexpected behavior. The Program Counter was setting itself within the cycle to various numbers and seemed to be stuck at a certain point. It looked as if the central control of the processor was entirely flawed, as several of the registers seemed to be doing their own thing, not heeding the flags being sent out by each stage of the processor. I decided to implement a basic debug routine to help show me what was happening in each of the registers to work out why it was not working.



Having implemented this I realized that the registers were successfully completing the first few iterations before being caught in some weird loop of simply repeating the same action. I realized that in order to debug this further I would need to make use of the built in runtime debugger within Rider, my preferred .net IDE. Runtime Debugging is very effective compared to simply printing values as you can more carefully step through and view all of the values within the scope, in some systems you may even modify the value of the variables. I placed a breakpoint inside of the main loop between the clocks so it would be possible to see the flags of each of the registers at the end of each application of the instruction.



It quickly became apparent using the runtime debugger that the register flags were not being reset and were instead accumulating. This meant that the first few iterations of the system would behave as expected before it would begin to diverge from the expected values. This prompted me to check if I had implemented a reset mechanism for the flags, and I realized that I had not. The problem was shortly resolved by setting the flags on all of the components of the processor to 0 at the end of each cycle.

I then decided to redo my tests, this time focusing on the expected changes in the registers. I calculated the expected value of each register based on the operations that should occur in that phase of the cycle, and compared this to the actual running application to verify that the entirety of the processor was working correctly.

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Register Operation | Expected Value | Actual Value |
| 0 | PC + 2-> ACC PC -> MAR | 2  0 | 2  0 |
| 1 | MDR -> CIR | 0 | 0 |
| 2 | ACC -> MAR  ACC + 2 -> ACC | 2  4 | 2  4 |
| 3 | MDR -> CIR[1] | 0 | 0 |
| 4 | ACC -> PC | 4 | 4 |

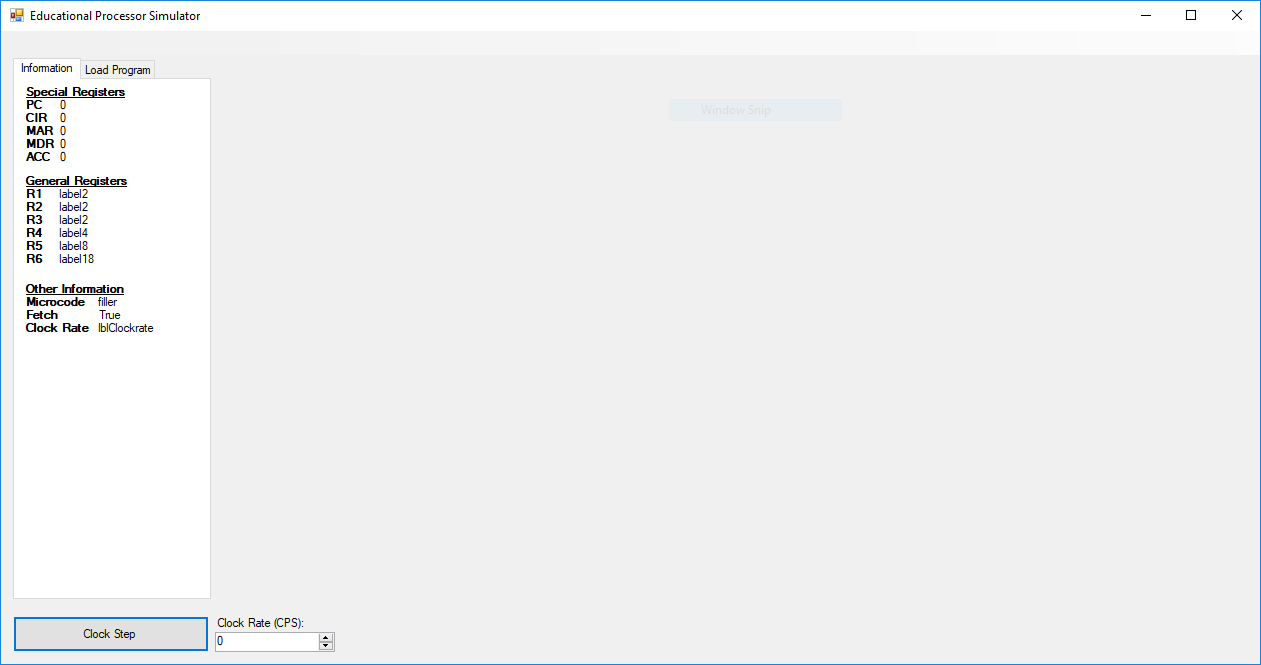
The tests seemed to indicate that the processor was now correctly orchestrating the registers for the purposes of loading a new instruction to execute. This meant I would then be able to move onto producing a rough UI that would show the state of the processor.

## Producing the rough UI

By the end of this task there should be a rough UI that will be able to control the processor and display the memory. Also as part of this part of the development, I will work on the memory and interfacing that to the MAR and MDR.

Whilst I primarily write code in Jetbrains Rider, for this task I switched to Visual Studio as this provides a helpful WYSIWYG editor that allows much more rapid development and prototyping for UI.

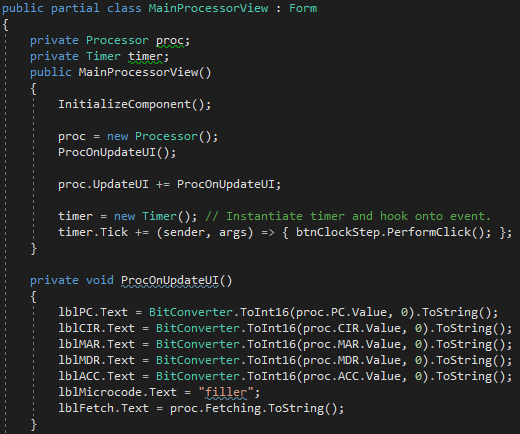
I first drew up a version that was purely UI and was not bound to any data. I could present this to the potential user base to see what they thought of it. My first prototype did not include the memory display as I will still thinking of several different ways of implementing this. It needed to be flexible and concise as it would have to show or be capable of showing what is quite a large data set. So, I left a blank area that would eventually be filled with my prototype for showing the memory.



It follows the general convention of application layouts with a space left for a toolbar at the top and a tabbed display on the left hand side. This allows more information to be included in a view without cluttering it, as you only show information relevant to the user’s experience at that time. I started off with two tabs, one for displaying information about the processors current state and another that will eventually hold the controls for loading a program and data into the memory.

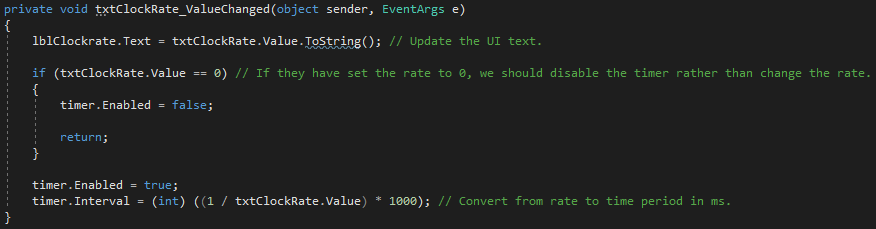
Controls such as the speed and manually cycling the clock are displayed in a space at the bottom. This is because I expect the user to want to always be able to easily access these and placing them within a menu structure would overcomplicate the experiences and slow down the user when they are trying to complete a task.

Hooking these into the actual data model was quite easy due to the modular design. I was able to extend the clock system to include a third and final stage “UpdateUI” which triggers the application to update the view to show the now changed data of the model after the rise and fall of the processor. I may want to change this behavior eventually so that it shows the Rise and the Fall separately to clearly demonstrate to an end-user how this specific functionality of the processor works.



Here we see that the Form instantiates a new processor object and then hooks into the UpdateUI event to trigger the ProcOnUpdateUI method. This goes through each of the UI elements and sets them to the value from the processor. As this function continues to grow, it may make sense to investigate only updating specific parts of the UI, or rate-limiting the updates. If the processor is running at 100hz, it may make more sense to only update the UI every 10 frames otherwise it will put a strain on the host machine which may become slow and unresponsive.

The Form also drives the processor using the built-in .net Windows Form Timer. I decided to use their implementation of the Timer as I doubted that I would require anything unusual or specific that would warrant writing my own Timer system. The Slider on the UI then hooks into this to change the interval of the Timer.



The logic also includes a simple test to make sure we disable the Timer when they have selected 0 Hz and then return out of the method. If we didn’t then a DivideByZeroException would be thrown when we try to complete the math on the last line.

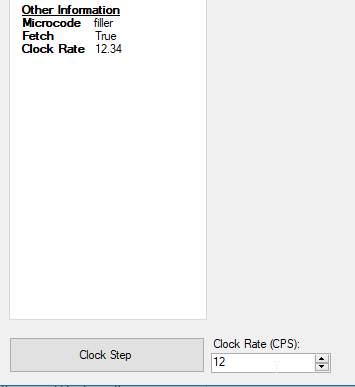
### Testing

With the core functionality of the model already tested, we now need to make sure that the same values are reflected within the UI. There are also a few more elements to be tested including the clock step button, and the clock rate entry.

|  |  |  |
| --- | --- | --- |
| Test | Expected Result | Actual Result |
| Pressing “Clock Step” and checking results via debug | This should cause the application to iterate through one clock cycle. In this case we should see the various registers behaving as they did in “Wire Framing the Classes” testing. This means when the fetch instruction completes (5 clock cycles), we should see the PC displaying 4. | When clicked 5 times (the length of the fetch instruction) we see the PC increment a full 2 words to display 4. |
|  |  |  |
| Pressing “Clock Step” and checking the results via the UI | The UI should match the data that is shown within the debug tool as tested above. | The view correctly matches the data within the model and shows 4 as the value for Program Counter. |
| Setting the Clock Rate to 0 | The Clock Rate displayed on the UI should be 0, and it should not clock the processor. | The processor does nothing as expected. |
| Setting the Clock Rate to 1 | The processor should slowly (1 per second) clock itself and this should be visible by the changing values. | The values change as expected, without direct interaction from the user. |
| Setting the Clock Rate to 100 | The processor should increment the registers as expected and remain responsive. | The processor increments as expected without interaction from the user and host machine CPU usage increase minorly but the application remains responsive. |
| Setting the Clock Rate to 666 | The field should, as configured, cap itself to a maximum value of 100 and reset to 100 after any unacceptable values. | The field immediately changes back to 100 after being set above this limit. |
| Setting the Clock Rate to 12.34 | The Clock Rate should round down to 12 and the UI should show this. | **The Clock Rate sets itself to 12.34 but the Clock Rate entry box rounds down to show 12. This means the UI is inconsistent in handling Decimal Places!** |

### Bug Fixes

Whilst most of the testing was successful, it did discover a inconsistency in the way the UI displays information. The selection box for the Clock Rate allows a user to enter a decimal value, and holds this value but by default only displays it as an integer. This means that whilst the value was actually 12.34, the user thought it had been rounded down to 12.



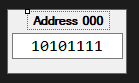
Resolving this was as simple as changing the ValueChanged method to begin by using Math.Round to set the value. This meant it was not only rounded in the data, but also correctly rounded in both the sections on the UI where it is shown

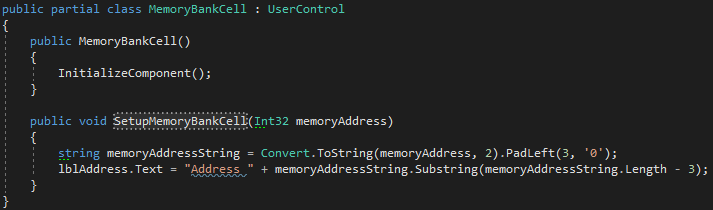
## Producing the Memory Bank UI

I felt that I would need to create the memory UI separately from the rest of the UI components because it had sufficient complexity and design requirements that it would need separate, specific and detailed testing.

I knew that this part of the UI would require lots of repeating elements and hence in the spirit of DRY (Don’t Repeat Yourself) I decided to break it down into three elements, so that code could be reused effectively. First, I would develop the individual cells that will display a byte or single address of memory, then the row that will contain those and finally the overall MemoryBank container that will set up the rows and organize their layout.

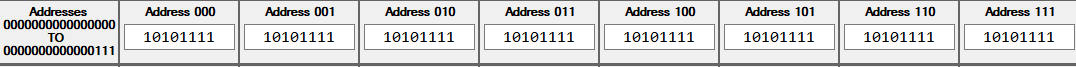
The MemoryBankCell is a relatively simple unit that contains a text box, for now filled with dummy data to show what it could look like, as well as a label that indicates the address of that byte.



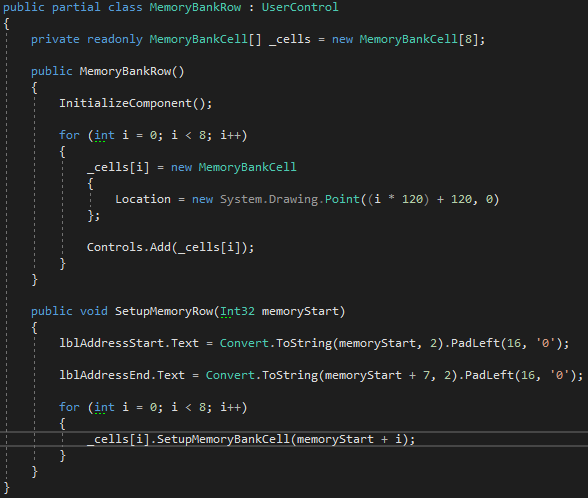


The code for this element is relatively simple however there is one interesting point. I was presented with two possible options: setting up the memory cell within the constructor, or providing this as a separate function to the constructor. The advantage of placing the setup in another function is that it is then possible to change the memory address represented by the cell on the fly, if it had remained in the constructor then our application would have to tear down the object before creating and rendering the new one.

Next I built the MemoryBankRow. This would wrap the cells in a row and provide to them their individual memory addresses. Eventually the main wrapper will pass the start address to each row. This tree of dependency means that it is incredibly easy to change the section of memory being displayed on the screen, as a change can propagate from the wrapper, to the row, to the child cells simply by invoking a single function on the wrapper.

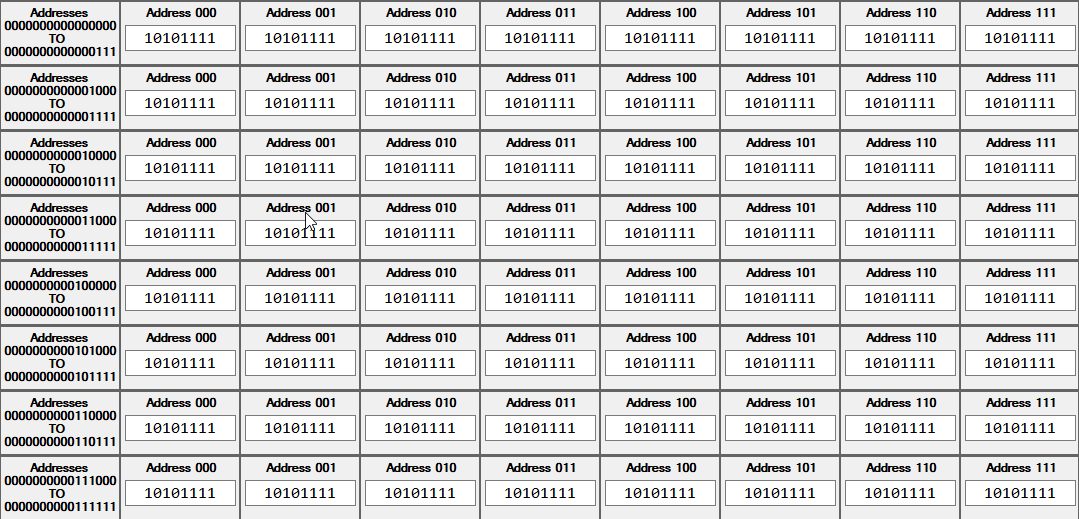


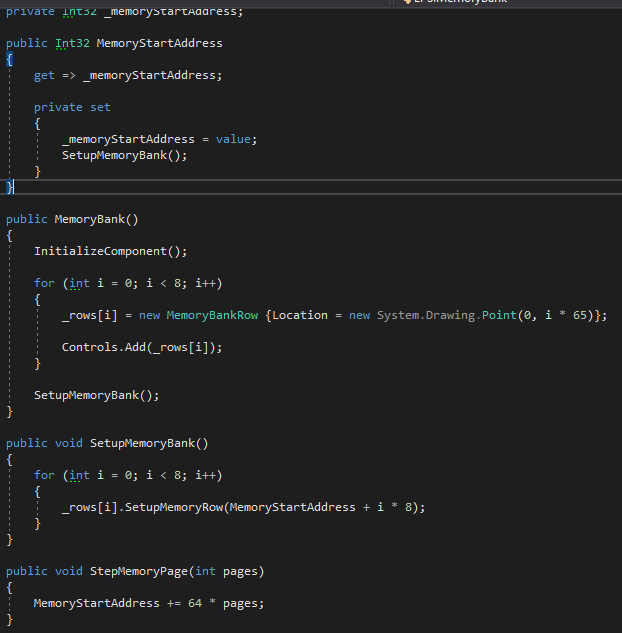
For this early prototype, I decided to use binary addresses and at a later date add in support for switching to a hexadecimal display. Hexadecimal would make it possible to display the entire address (16 bits) instead of just displaying the last three bits in the context of the row. This is because a single hexadecimal digit can be used to represent four bits, meaning only four digits could be used to represent the entire sixteen bits.



Again we see a relatively simple structure behind the component, as it only has to instantiate the eight cells and be able to pass information down to them as well as formatting the address range it represents to show as 16 bits. I use fixed length arrays to hold the MemoryBankCells as I can be sure that there will always be 8 cells and by using arrays access to data is faster and the data itself takes up less space than other data structures such as Linked Lists.

Finally we have the container that will hold the rows. It will also control the pagination of the memory bank so the user can switch through each page of 8 rows of 8 cells (64 memory addresses). Whilst the buttons for this interaction will be outside of the class, they will be able to interact with it through public methods. This is an example of encapsulation as objects outside of the MemoryBank will not be able to directly access the private properties such as the memory address being targeted, instead these private properties will be manipulated through public methods such as nextMemoryPage() which can have additional logic to ensure that only valid data is selected, for example, ensuring it is not possible to go past the end of the possible memory address range.

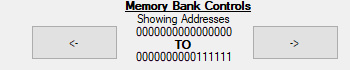




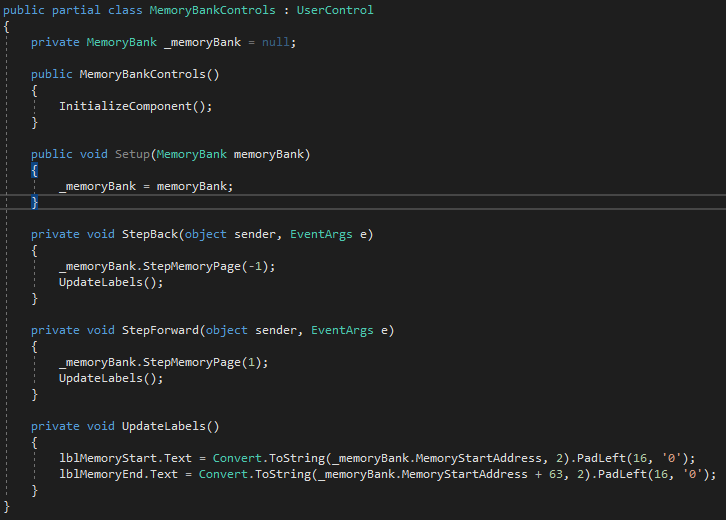
Again, we can see this this component simply builds upon the previous components and hence maintaining its simplicity. The code demonstrates the use of a getter/setter in order to maintain encapsulation as well as to trigger another function when data does change. The function is this example will propagate the change down to the rows, which will propagate it down to the cells.

The StepMemoryPage public method will be used by controls outside of the memory display to interact with it.

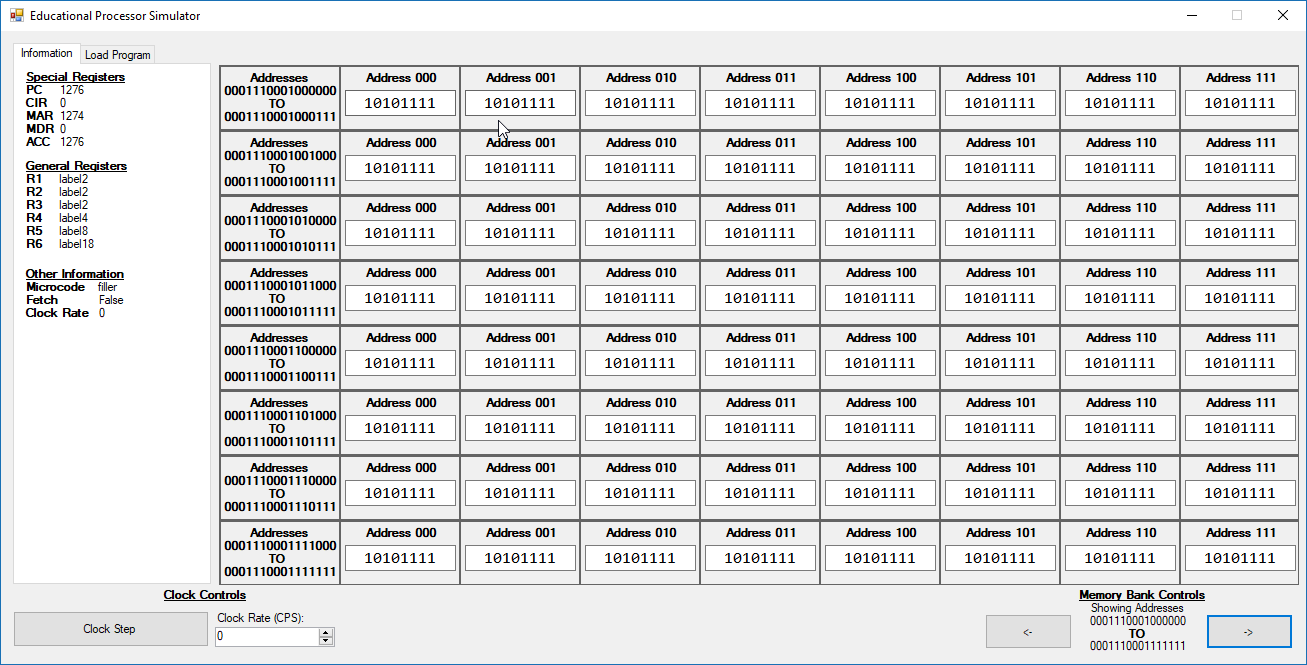
The final step before I could complete more testing was to link up some controls within the interface.



I created this unit as part of its own control because this would keep the code separate making it cleaner and more maintainable, this also meant that it would not have special or particular access to the MemoryBank, instead only interacting only through the public methods available.



With all of the components complete, the UI for the main window now looks as follows:

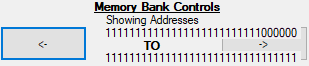


### Testing

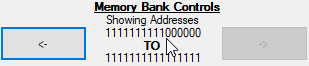
Whilst the Memory Bank UI has not yet been tied up to the data model, it is still possible to check that the address labels and interaction buttons work as expected.

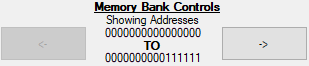
|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| The original state of the UI is correct before any user interaction. | At start, the memory bank should show addresses 0000\_0000\_0000\_0000 to 0000\_0000\_0011\_1111 with 8 unique addresses presented on each row. | The address ranges of the entire memory bank as well as the individual rows match what is expected. |
| Clicking the next page button from the original state. | The range of the memory bank should change to 0000\_0000\_0100\_0000 to 0000\_0000\_0111\_1111 and the address range of each row should fall within this. | The memory bank correctly shifts to the next page of 64 addresses. |
| Clicking the next page button again. | The range of the memory bank should change to 0000\_0000\_1000\_0000 to 0000\_0000\_1011\_1111 and the address range of each row should fall within this. | The memory bank correctly shifts to the next page of 64 addresses. |
| Clicking the previous page button. | The range of the memory bank should change to 0000\_0000\_0100\_0000 to 0000\_0000\_0111\_1111 and the address range of each row should fall within this. | The memory bank correctly shifts to the previous page of 64 memory addresses. |
| Clicking the previous page button from the original state | The user is not able to move the memory bank back another page. | **The memory address goes into a negative value.** |

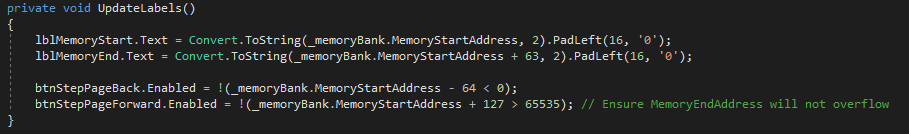
### Bug Fixes



As seen in the screenshot here, we have failed the test suite. This is definitely not a good user experience and some data validation needs to be implemented to ensure it is not possible for a user to scroll back before the acceptable range of memory addresses. To resolve this some basic data validation needed to be added. In my case it meant disabling the button to make it clear to the user that interacting with it would not produce any results. This is preferable to simply having the button do nothing as this might be confusing to a potential user.







Demonstrated is the MemoryBankControls at either end of the two extremes, where it would be able to cause it to go into the negative, or cause the memory address to overflow. We can now see that a visual indicator (the greying out of the control) clearly shows the user that the control cannot be interacted with. This was implemented simply by considering the memory address each time it has been incremented and whether or not it would go into the negative if the page was moved back and forth. Now that this has been completed, the test suite now passes and I can move onto producing the memory bank model.

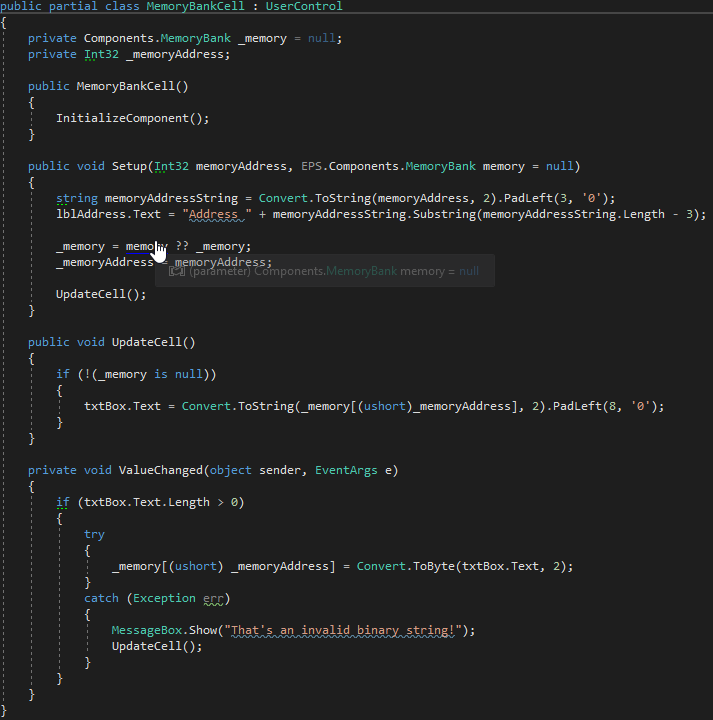
## Producing the Memory Bank Model

As previously explained, I decided to break this stage separately from producing the UI as both are independently testable.

The Memory Bank will emulate the behavior of RAM, reading the address supplied by the MAR and placing this data inside of the MDR. I have decided for the purposes of this processor for the latency of memory read and writes to be instant and occur within a single clock edge in the same way that the registers behave. This abstracts away some of the complexity typically involved with reading and writing from memory that exists outside of the processor itself.



Similar to the Registers, we see the component acts mainly on the falling clock signal and uses a set of flags to control its operation. One interesting thing to note is the inclusion of an ‘indexer’. An indexer is a feature of the C# language that allows the developer to specify getters and setters for when the Object based on a class is indexed. This allows the developer to create custom data types similar to arrays or dictionaries, and in this case other components will be able to access the memory via MemoryBank[address] as will be demonstrated later. This further shows the flexibility and power of OOP.



Here we see the binding of the individual MemoryBankCell to the MemoryBank model. We can see that a try catch is used as the default behavior of the Convert.ToByte is to throw an exception if the string entered is either too long to fit into that height or will not convert to a byte cleanly. If the user does enter an invalid string, we show a message to let them know and then reset it to the value of the cell from before they attempted to change it.

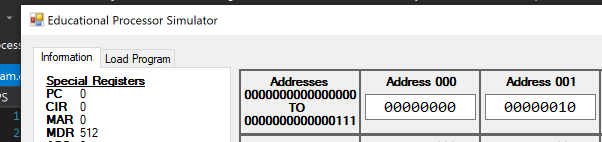
### Testing

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Cells should display correct initial value. | Cells should display 00000000 on initial load. | https://ss.noahstride.co.uk/screens/2019/01/EPS_2019-01-07_21-50-18.png  **Correct Result** |
|  |  |  |
| Entering a valid binary byte into a cell. | The user should be able to enter a valid binary byte and this value should persist when the user clicks out of the cell. | https://ss.noahstride.co.uk/screens/2019/01/EPS_2019-01-07_22-00-27.png  **Correct Result** |
| Entering an invalid binary byte into a cell | Upon entering a character that will not convert properly to a binary byte, the application should display a warning message to the user, and reset the cell when closed. | https://ss.noahstride.co.uk/screens/2019/01/EPS_2019-01-27_13-50-56.png  **Correct Result** |

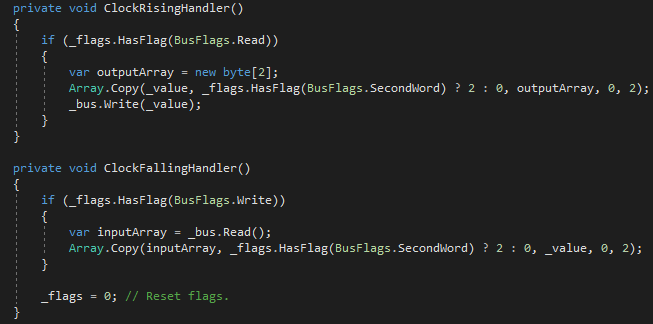
Because of the modular design, it is now possible to test the interaction between the memory bank module and the rest of the processor and to ensure that everything works together correctly.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Entering a value into a cell, and allowing the MAR to select this address | The MDR should show the value of the cell as 2. | A value of 512 is shown.  **FAIL** |
|  |  |  |

### Bug Fixes



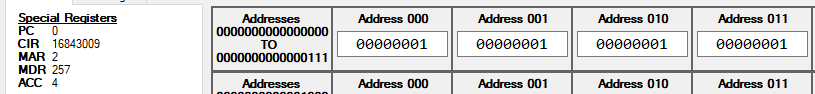
This result was rather un-expected, however, it quickly began to make sense why it was behaving this way. As part of the load cycle we expect the processor to load in 32 bits of memory (2 full words) into the CIR, here we instead see two things going wrong. I used my IDEs run-time debugger to step through the cycle to confirm my suspicions and upon inspecting my source I realized that the second word loading/writing of the CIR was not fully implemented and this meant we were only pulling in 16 bits of the expected 32 bits. This was quickly rectified within the ClockRising and ClockFalling handlers:



I made use of the ternary expression (?) here to ensure that the code remained short and concise and to enable maximum effective re-use of code.

The second issue I noticed as a result of this test was a mismatch in the byte orders. I was expecting the processor to operate in Big-Endian order and this means that the largest byte in a value comes first, this is common in older processors, but has been phased out in preference of Little-Endian order. The built-in C# converter assumes that byte arrays will be in Big-Endian order. This means that the values of the numbers are scrambled compared to what a user might be expecting, I decided to leave this for now as it made more sense to leave it displaying this way, if that is how the processor is actually handling the value.

With these resolved, the CIR now shows the correct value.

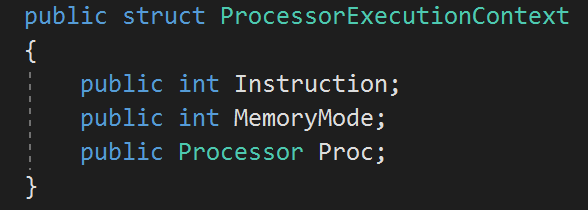


## Adding Instructions

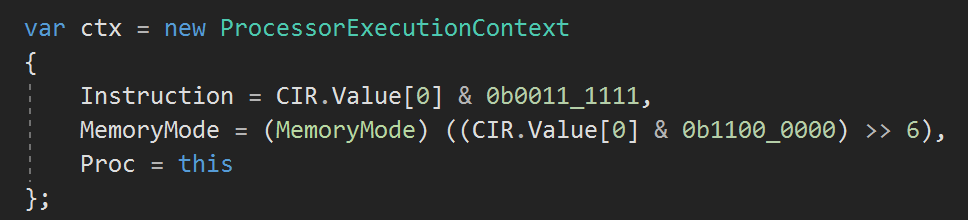
With the basis of the processor complete, it was now possible to implement several instructions for the user to use.

I decided that for the processor to be able to display basic functionality I would need to implement at least ADD, SUB, LDA, STA, MOV and BRA, BRZ and BRP.

To begin, I needed to introduce a context struct that would describe the state of the processor so that each stage of the execution of an instruction has access to the information it needs to complete. Whilst a simple structure, it allows a much cleaner approach to providing general information to the steps of execution without polluting an existing namespace.

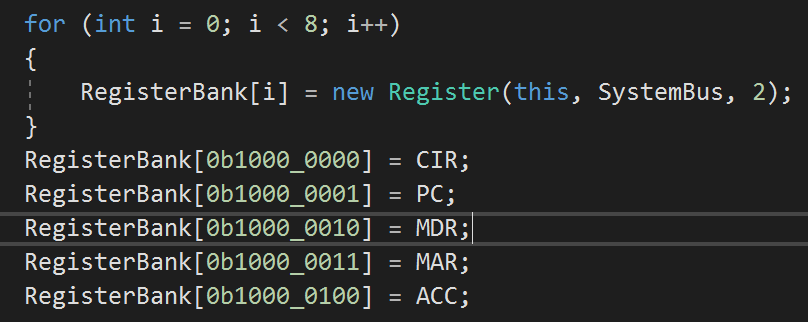


These values are then filled within the execution cycle before being passed to each step of execution. Also demonstrated here is a binary bitwise AND followed by right shift in order to convert the full instruction to just the memory mode. This is cast to the appropriate enum to keep remove



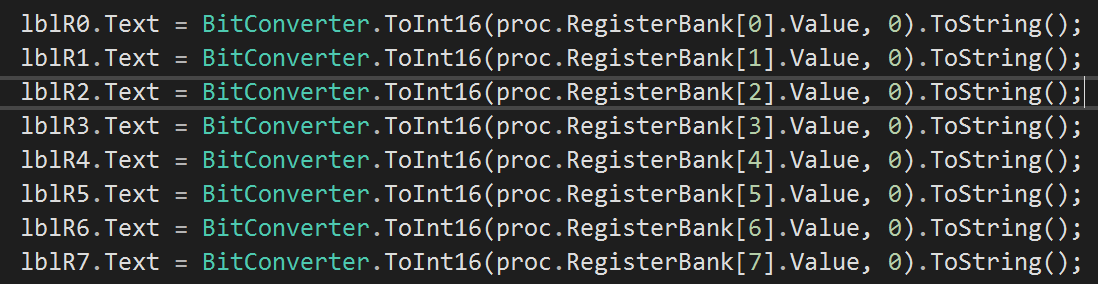
As well as producing the struct, I then needed to add the context where appropriate to replace direct calls to proc.

I also needed to implement the general registers, as well as a way of appropriately accessing them. I decided that a small 8-bit address space would allow the general registers as well as the special registers to be accessed by users’ applications programmatically.

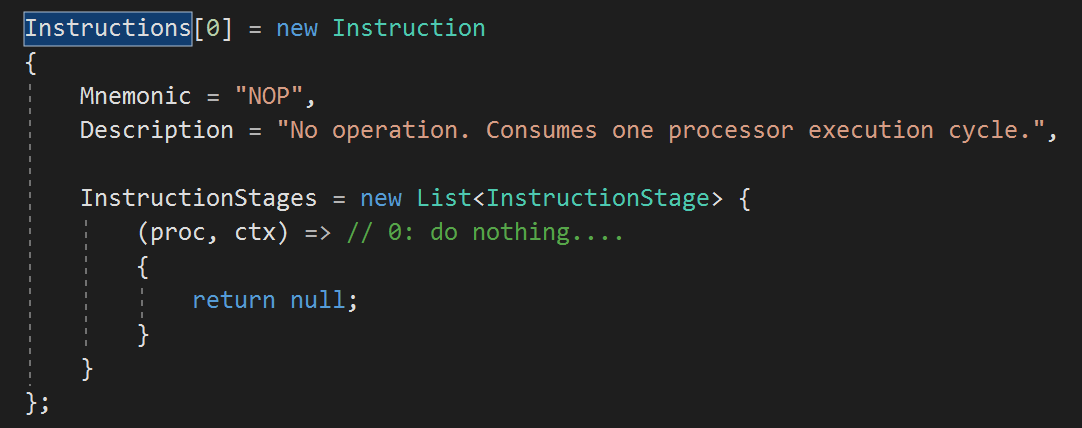


By adding these registers to a bank (in this case represented by an array) the user would be able to specify any of these registers (the general and special) when writing their machine code. I felt an array was a suitable data-type to choose since it would always have a fixed length. As I knew it was a fixed length I was then able to use the first bit to represent whether or not the program wanted to access a special or general register.

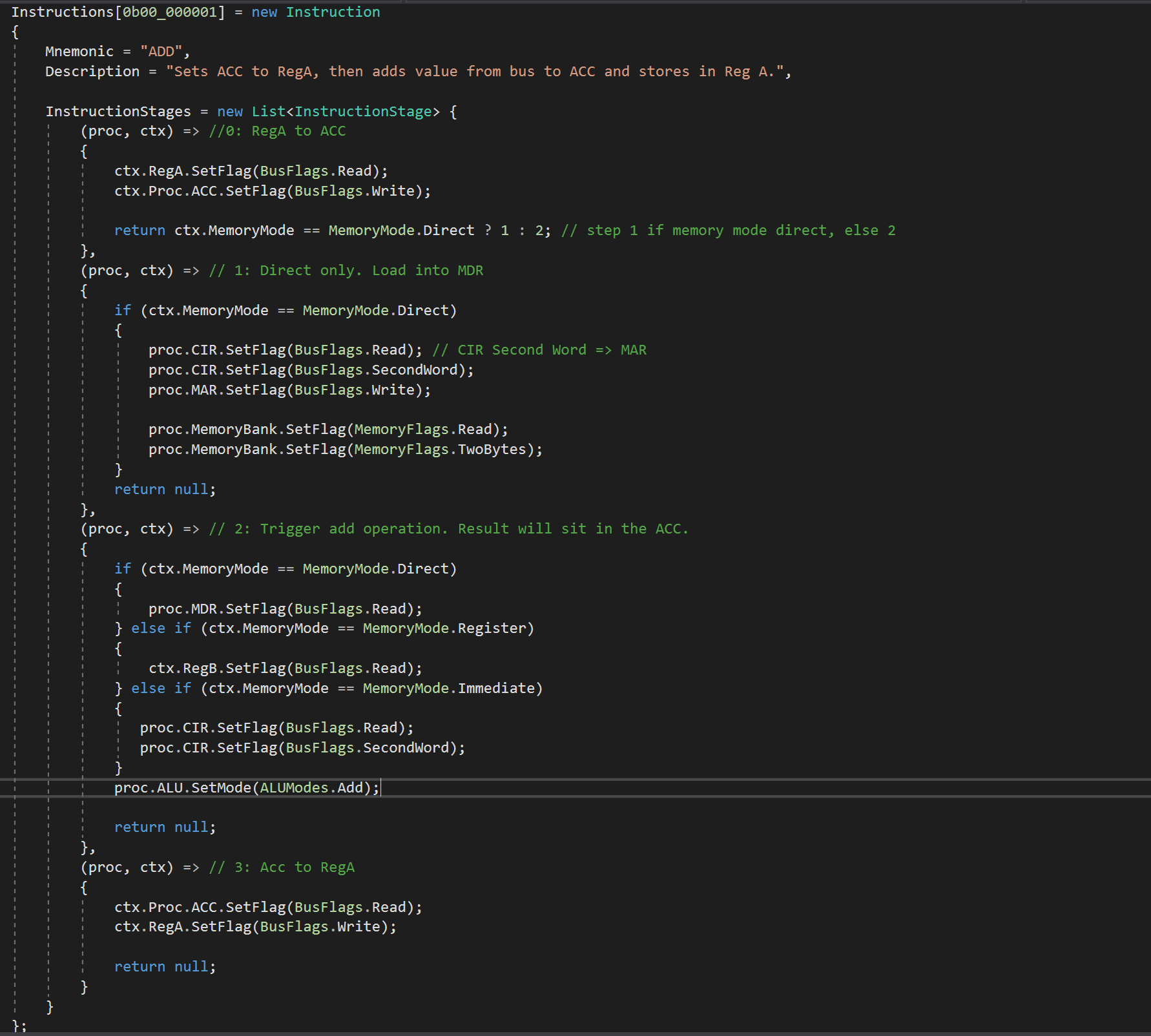
In addition to adding the data side, I also needed to add these registers to the view presented to the user. This was relatively simplistic due to the existing framework for mapping values I had created.



The first instruction I wrote was NOP (No Operation) as this was the simplest. Despite being simple, it is still an incredibly useful instruction in certain circumstances such as where it may be preferable for the processor to wait several cycles before completing an action.



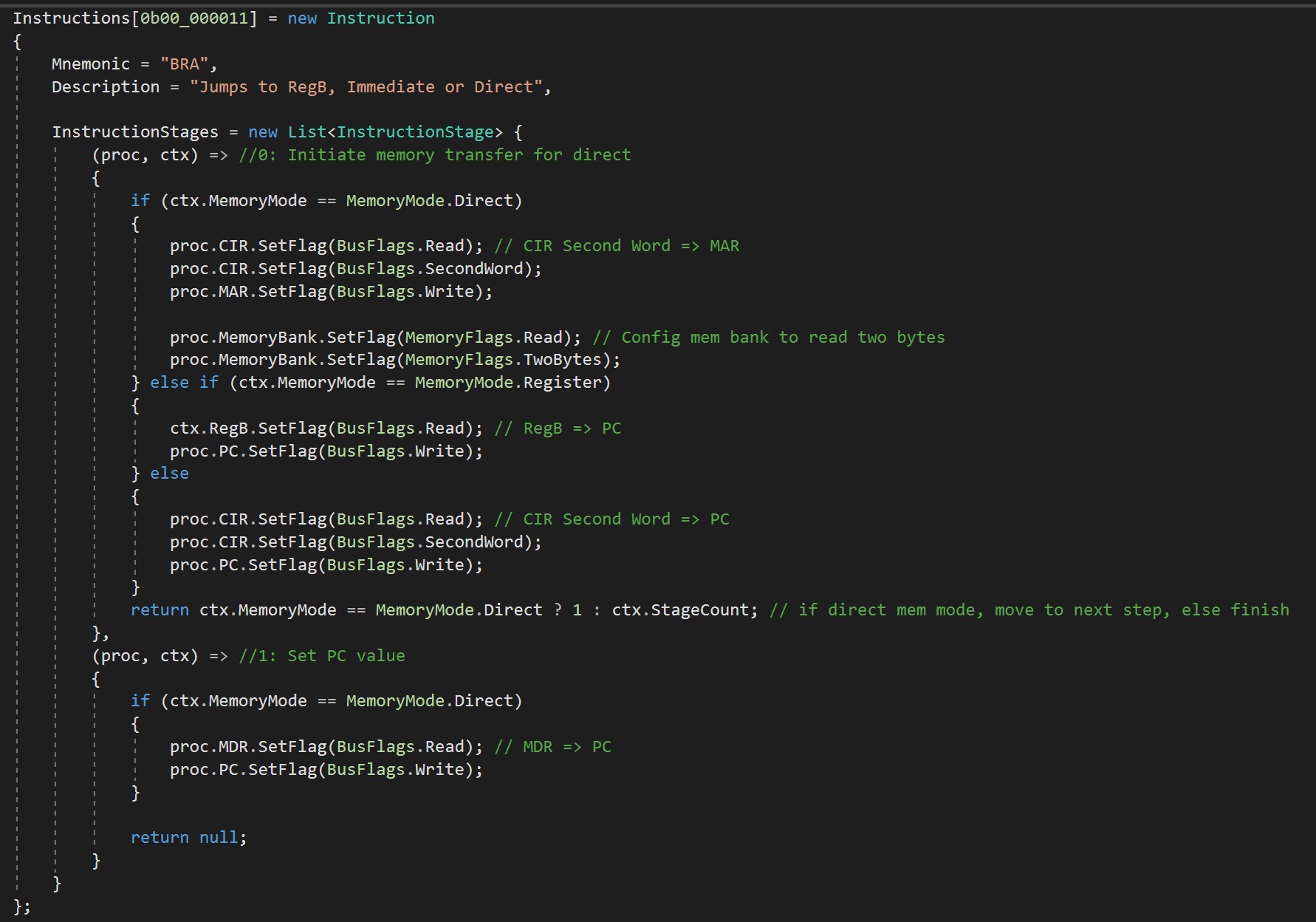
The next instruction I needed to implement was the Add instruction. It would need to be able to handle three distinct scenarios: direct, indirect and register read.



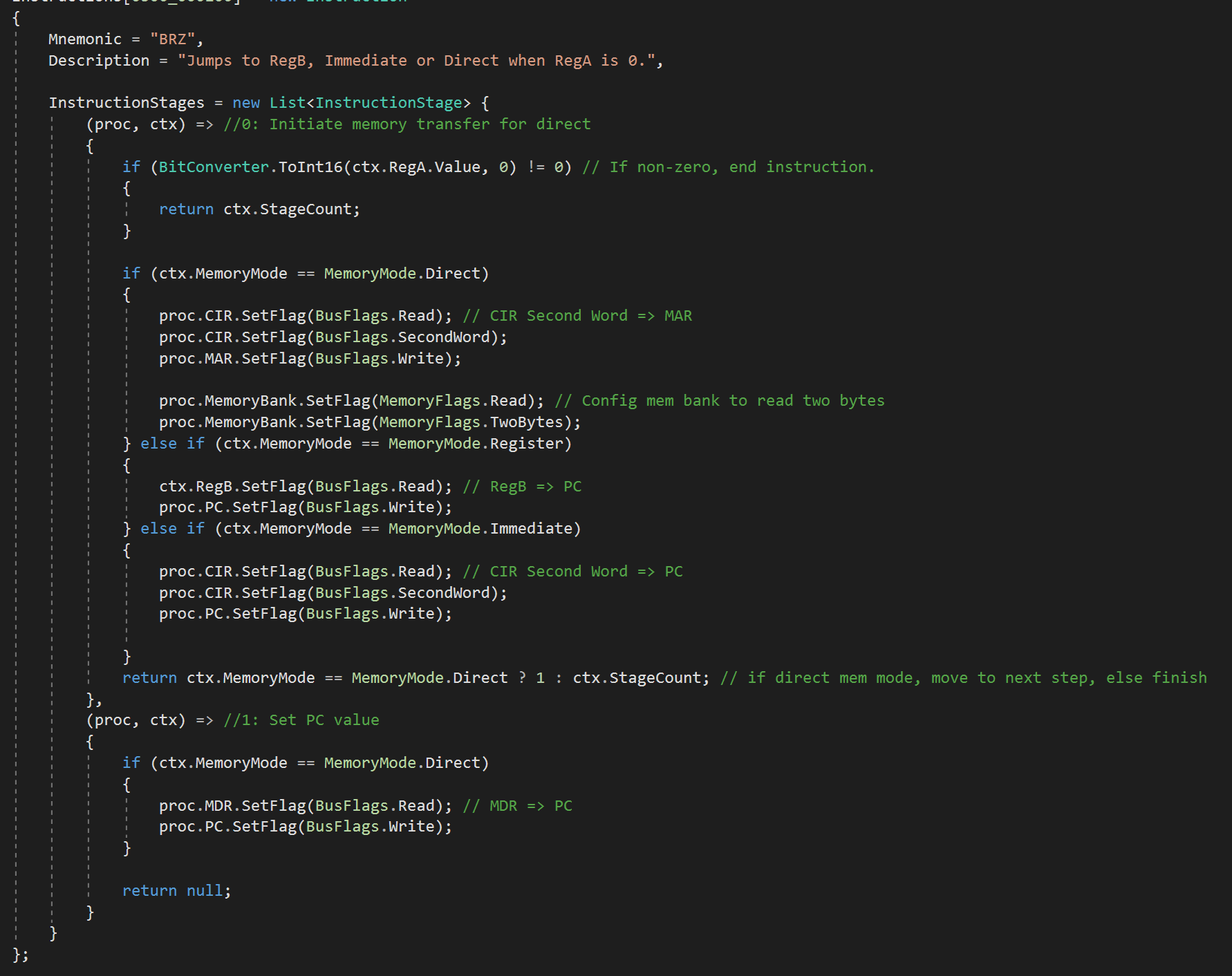
The add instruction completes within 4 cycles and has a relatively simplistic design making use of the micro-code framework I had previously developed. I also commented each stage of the instruction to make clear to myself and other developers the purpose of that step, since it is not always immediately clear what it intends to do. Even with the handling of the three memory modes, the code remains relatively concise due to the structure provided by the framework.

The next instruction I developed was the SUB instruction as this was just a small modification required to the ALU class itself as well as the just copying and making a small change to the addition instruction.

Then the next instruction to develop was BRA. BRA allows the user to tell the processor to jump to a specific memory location by setting the PC to that location. Theoretically this could have been completed with just a MOV instruction, but it makes more semantic sense to the user for it to be a separate command, and it will also be a foundation for BRZ and BRP. We can see that once again, the micro-code for each instruction becomes quite lengthy when we consider all three of the possible memory modes.

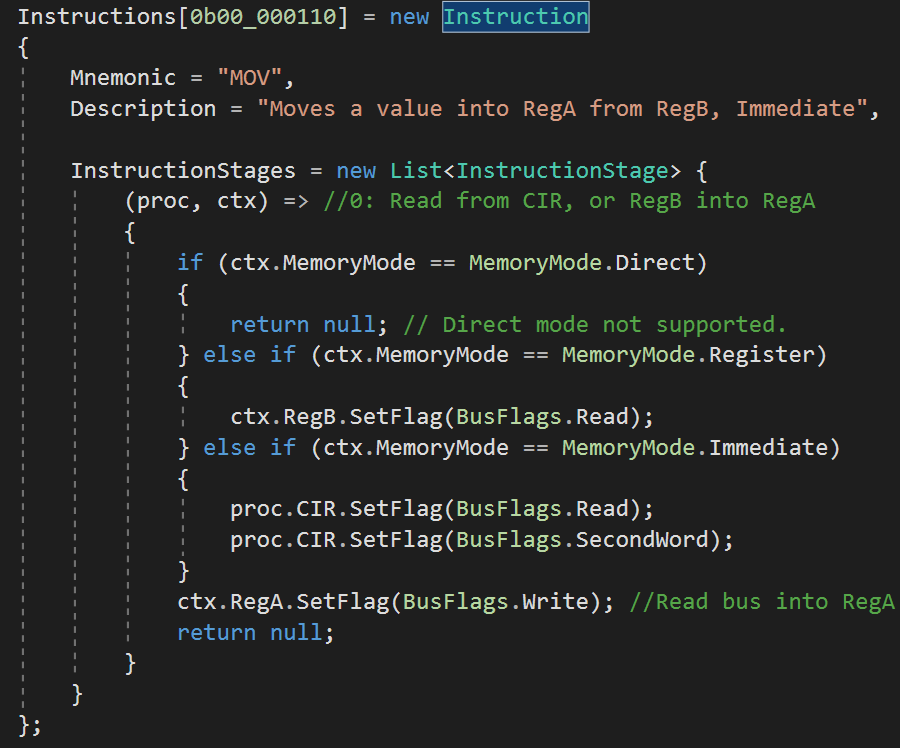


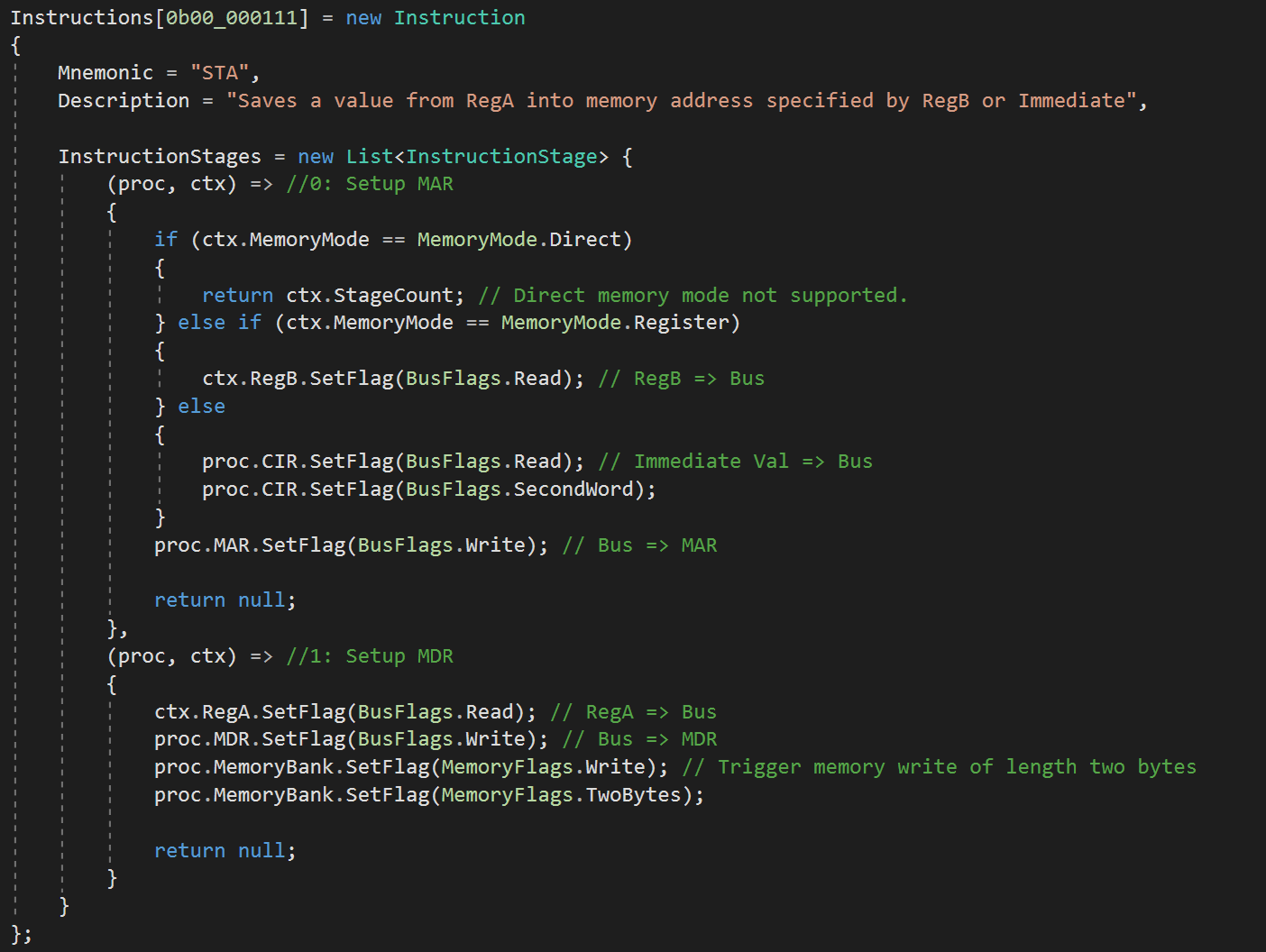
With BRA (Branch Always) correctly implemented it now made sense to introduce BRZ (Branch on Zero) and BRP (Branch on Zero or Positive). These would be incredibly similar to BRA, but with an adding conditional statement. These instructions will jump to a memory location specified in a memory location, an immediate or using a location in a RegB when RegA’s contents match the expected condition.



As we can see, the only real change between BRA and BRZ is the inclusion of a single condition at the start of the micro-code. The only difference between BRZ and BRP was changing the condition here to instead end the instruction if the value is less than zero.

To allow the user to use registers effectively instructions are needed that will allow data to be moved between them, as well as moved to the RAM. Hence three instructions will be introduced, MOV, STA, LDA. MOV will allow content to be moved between registers or data to be directly inserted into a register, LDA will allow something to be moved into RegA from RAM from an address specified in RegB or as an immediate value and STA will allow operations in the opposite direction. Whilst it would be possible to allow a memory location specified by a memory location to be read or stored into, I feel this would create additional un-needed complexity when this action is unlikely to occur regularly, and it is easy enough to work around this by using a register as an intermediate data store.





In the interests of brevity, I will not include a screenshot of the code for the LDA instruction as this essentially mirrors the STA instruction with the exception of the order of operation. Demonstrated in these screenshots once again in the Instruction class being used to provide a strong framework to work within, with the workings of each instruction remaining decoupled from the processor’s components.

### Testing

In order to correctly test this section, I need to test each of the instructions and their memory modes against the Register and Memory values I expect to see.

#### NOP

The NOP instruction should be relatively simple to test.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| 00000000 00000000 00000000 00000000 | There should be no changes to any registers and no writes to memory. | All registers remain at 0 and there are no changes to any memory.  **Correct Result** |

#### ADD

The ADD instruction will need to be tested in all three of the supported memory modes.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00000001 00000000 00000000 00000001 | Register 0 should be filled with the result of 0 + 256. Which is 256. | **Correct Result** |
| Same instruction ran again. | Register 0 should be filled with the result of R0 + 256. Here R0 has a value of 256 from the previous instruction so Register 0 should become 512. This tests that RegA is correctly read into the ACC. | **Correct Result** |
| Direct Mode  01000001 00000000  00001000 00000000  With 00000000 00000001 in memory position 8 | The value of 256 should be read from memory location 8 and added to 0 from Reg0 before being moved back into Reg0 to give it a value of 256. | **Correct Result** |
| Register Mode  10000001 00000000 00000001 00000000 | Register 0 should be filled with the result of 0 since Register 1 is empty and Register 0 are empty. | **Correct Result** |

#### SUB

The SUB instruction will need to be tested in only one of the memory modes, since it is identical to the ADD instruction with only one small change.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00000010 00000000 00000000 00000001 | Register 0 should be filled with the result of 0 - 256. Which is -256. | **Correct Result** |

#### BRA

BRA will be tested in two memory modes, Immediate and Direct. To see if the test has been successful, the contents of PC will be observed as this is indicative of a BRA operation.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00000011 00000000 00000000 00000001 | Program Counter should be set to 256. | **Correct Result** |
| Register Mode  10000011 00000000 00000000 00000000 | Program Counter should be set back to 0, as the value of R0 is 0. | **Correct Result** |
| Direct Mode  01000011 00000000 00001000 00000000  With 00000000 00000001 in memory location 8 | Program Counter should be set to 256. | **Correct Result** |

#### BRZ

BRZ will be tested in a single memory mode as it is functionally similar to BRA. Instead emphasis will be placed on testing that the conditional behaves correctly.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00000100 00000000 00000000 00000001 | Program Counter should be set to 256 as Reg0 is zero. | **Correct Result** |
| Immediate Mode  00000100 00000000 00000000 00000001  With 1 in Reg0 | Program value should not Jump to 256 and should continue to 12. | **Correct Result** |
| Immediate Mode  00000100 00000000 00000000 00000001  With -1 in Reg0 | Program value should not Jump to 256 and should continue to 12. | **Correct Result** |

#### BRP

BRP will be tested in a single memory mode as it is functionally similar to BRZ and BRA. Instead emphasis will be placed on testing that the conditional behaves correctly.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00000101 00000000 00000000 00000001 | Program Counter should be set to 256 as Reg0 is zero. | **Correct Result** |
| Immediate Mode  00000101 00000000 00000000 00000001  With 1 in Reg0 | Program value should Jump to 256. | **Correct Result** |
| Immediate Mode  00000101 00000000 00000000 00000001  With -1 in Reg0 | Program value should not Jump to 256 as -1 is not greater than or equal to 0. Instead it will continue to 12. | **Correct Result** |

#### MOV

MOV will need to be tested in both of its memory modes.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00000110 00000000 00000000 00000001 | Reg0 should be set to 256. | **Correct Result** |
| Register Mode  10000110 00000001 00000000 00000000  With 1 in Reg0 | Reg1 should be set to the value of Reg0, which is 256 from the results of the previous instruction. | **Correct Result** |

#### STA

STA will need to be tested in both of its memory modes.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00000111 00000000 00010000 00000000  With 256 in Reg0 | 256 should be inserted in memory location 16. | **Correct Result** |
| Register Mode  10000111 00000000 00000001 00000000  With 256 in Reg0 and 16 in RegB | 256 (from RegA) should be inserted in memory location 16 from RegB. | **Correct Result** |

#### LDA

LDA will need to be tested in both of its memory modes.

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Result** | **Actual Result** |
| Immediate Mode  00001000 00000000 00010000 00000000  With 256 in memory location 16 | Reg0 should be set to the value of memory location 16, which is 256. | **Correct Result** |
| Immediate Mode  00001000 00000000 00000001 00000000  With 256 in memory location 16 and 16 in Reg0 | Reg0 should be set to the value of the memory location in Reg0 which is 16, which itself contains a value of 256. | **Correct Result** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| MEM MODE | MEM  MODE | INSTRCT | INSTRCT | INSTRCT | INSTRCT | INSTRCT | INSTRCT |
| REGA | REGA | REGA | REGA | REGA | REGA | REGA | REGA |
| REGB OR MEM OR  DIRECT | REGB OR MEM OR  DIRECT | REGB OR MEM OR  DIRECT | REGB OR MEM OR  DIRECT | REGB OR MEM OR  DIRECT | REGB OR MEM OR  DIRECT | REGB OR MEM OR  DIRECT | REGB OR MEM OR  DIRECT |
| MEM OR  DIRECT | MEM OR  DIRECT | MEM OR  DIRECT | MEM OR  DIRECT | MEM OR  DIRECT | MEM OR  DIRECT | MEM OR  DIRECT | MEM OR  DIRECT |

2 memory bits, followed by 6 bits of instruction identifier. Followed by 8 bit address of register A, then 8 bits of Reg B, or 16 bits of Memory Address or 16 bits

## Final Code Listing

This is alternatively available as a github repository. Access can be granted by contact with the candidate.

### Form 1.cs

using System;

using System.Windows.Forms;

namespace EPS

{

public partial class Form1 : Form

{

private Processor proc;

private Timer timer;

public Form1()

{

InitializeComponent();

proc = new Processor(); // Create processor model.

ProcOnUpdateUI();

proc.UpdateUI += ProcOnUpdateUI;

timer = new Timer(); // Setup timer and hook up button

timer.Tick += (sender, args) => { btnClockStep.PerformClick(); };

memoryBankControls.Setup(memoryBank); // Hook memorybank up to proc

memoryBank.Setup(proc.MemoryBank);

}

private void ProcOnUpdateUI()

{

lblPC.Text = BitConverter.ToInt16(proc.PC.Value, 0).ToString();

lblCIR.Text = BitConverter.ToInt32(proc.CIR.Value, 0).ToString();

lblMAR.Text = BitConverter.ToInt16(proc.MAR.Value, 0).ToString();

lblMDR.Text = BitConverter.ToInt16(proc.MDR.Value, 0).ToString();

lblACC.Text = BitConverter.ToInt16(proc.ACC.Value, 0).ToString();

lblR0.Text = BitConverter.ToInt16(proc.RegisterBank[0].Value, 0).ToString();

lblR1.Text = BitConverter.ToInt16(proc.RegisterBank[1].Value, 0).ToString();

lblR2.Text = BitConverter.ToInt16(proc.RegisterBank[2].Value, 0).ToString();

lblR3.Text = BitConverter.ToInt16(proc.RegisterBank[3].Value, 0).ToString();

lblR4.Text = BitConverter.ToInt16(proc.RegisterBank[4].Value, 0).ToString();

lblR5.Text = BitConverter.ToInt16(proc.RegisterBank[5].Value, 0).ToString();

lblR6.Text = BitConverter.ToInt16(proc.RegisterBank[6].Value, 0).ToString();

lblR7.Text = BitConverter.ToInt16(proc.RegisterBank[7].Value, 0).ToString();

lblMicrocode.Text = "filler";

lblFetch.Text = proc.Fetching.ToString();

}

private void BtnCycleClick(object sender, System.EventArgs e)

{

proc.Clock();

}

private void txtClockRate\_ValueChanged(object sender, EventArgs e)

{

lblClockrate.Text = txtClockRate.Value.ToString();

if (txtClockRate.Value == 0) // If 0, disable timer.

{

timer.Enabled = false;

return;

}

timer.Enabled = true;

timer.Interval = (int) ((1 / txtClockRate.Value) \* 1000);

}

}

}

### Processor.cs

using System;

using System.Collections.Generic;

using EPS.Components;

using EPS.Instructions;

namespace EPS {

// Possible acceptable memory modes.

public enum MemoryMode

{

Immediate,

Direct,

Register

}

// Provided to instructions as context for the execution.

public struct ProcessorExecutionContext {

public int FullInstruction;

public int Opcode;

public MemoryMode MemoryMode;

public Processor Proc;

public int StageCount;

public Register RegA;

public Register RegB;

public short? Immediate; //16bit

public ushort? MemoryAddress; //16bit unsigned

}

public partial class Processor

{

public InstructionSet InstructionSet = new InstructionSet();

public readonly Bus SystemBus;

public readonly Register CIR;

public readonly Register PC;

public readonly Register MDR;

public readonly Register MAR;

public readonly Register ACC;

public readonly ALU ALU;

public readonly Components.MemoryBank MemoryBank;

public readonly Register[] RegisterBank = new Register[256];

public bool Fetching = true; // true for fetch, false for execute

private Instruction Fetch = new Instruction

{

InstructionStages = new List<InstructionStage> //Microcode for the Fetch instruction

{

(proc, ctx) => //0

{

proc.PC.SetFlag(BusFlags.Read); // PC -> MAR

proc.MAR.SetFlag(BusFlags.Write);

proc.ALU.SetMode(ALUModes.IncrementWord); // Increment value and store in ACC

proc.MemoryBank.SetFlag(MemoryFlags.Read);

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

return null;

},

(proc, ctx) => //1

{

proc.MDR.SetFlag(BusFlags.Read); // MDR -> CIR/ALU

proc.CIR.SetFlag(BusFlags.Write);

return null;

},

(proc, ctx) => //2

{

proc.ACC.SetFlag(BusFlags.Read); // ACC -> MAR

proc.MAR.SetFlag(BusFlags.Write);

proc.MemoryBank.SetFlag(MemoryFlags.Read);

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

proc.ALU.SetMode(ALUModes.IncrementWord); // Increment value and store in ACC

return null;

},

(proc, ctx) => //3

{

proc.MDR.SetFlag(BusFlags.Read); // MDR -> CIR Second word

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.CIR.SetFlag(BusFlags.Write);

return null;

},

(proc, ctx) => //4

{

proc.ACC.SetFlag(BusFlags.Read); // ACC -> PC. PC is now fully incremented 2 words.

proc.PC.SetFlag(BusFlags.Write);

return null;

}

}

};

public Processor()

{

SystemBus = new Bus(2);

CIR = new Register(this, SystemBus, 4); // Instructions have 2 words

PC = new Register(this, SystemBus, 2);

MDR = new Register(this, SystemBus, 2);

MAR = new Register(this, SystemBus, 2);

ACC = new Register(this, SystemBus, 2); // TODO: Ensure Accumulator is referencable by instructions as a User register.

ALU = new ALU(this, SystemBus, ACC);

for (int i = 0; i < 8; i++)

{

RegisterBank[i] = new Register(this, SystemBus, 2); // Generate general registers.

}

RegisterBank[0b1000\_0000] = CIR; // Set special registers.

RegisterBank[0b1000\_0001] = PC;

RegisterBank[0b1000\_0010] = MDR;

RegisterBank[0b1000\_0011] = MAR;

RegisterBank[0b1000\_0100] = ACC;

MemoryBank = new Components.MemoryBank(this, MDR, MAR);

}

// Setup events and delegates.

public event ClockRisingHandler ClockRising;

public event ClockFallingHandler ClockFalling;

public event UpdateUIHandler UpdateUI;

public delegate void ClockRisingHandler();

public delegate void ClockFallingHandler();

public delegate void UpdateUIHandler();

public void Clock()

{

var ctx = new ProcessorExecutionContext // Setup execution context

{

FullInstruction = CIR.Value[0],

Opcode = CIR.Value[0] & 0b0011\_1111,

MemoryMode = (MemoryMode) ((CIR.Value[0] & 0b1100\_0000) >> 6),

Proc = this,

RegA = RegisterBank[CIR.Value[1]]

};

// Setup execution context for specific memorymode.

if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB = RegisterBank[CIR.Value[2]];

} else if (ctx.MemoryMode == MemoryMode.Immediate)

{

ctx.Immediate = BitConverter.ToInt16(CIR.Value, 2);

} else if (ctx.MemoryMode == MemoryMode.Direct)

{

ctx.MemoryAddress = BitConverter.ToUInt16(CIR.Value, 2);

}

// Main execution logic.

if (Fetching)

{

Fetching = !Fetch.Execute(ctx);

}

else

{

var currentInstruction = InstructionSet.Instructions[ctx.Opcode]

?? InstructionSet.Instructions[0]; //Fallback to NOP if instruction does not exist.

Fetching = currentInstruction.Execute(ctx);

}

ClockRising(); // Write to Bus

ClockFalling(); // Read from bus and operate

MemoryBank.ClockFallingHandler();

UpdateUI();

WriteState();

}

}

}

### Register.cs

using System;

namespace EPS.Components

{

// Possible operation flags for register.

[Flags]

public enum BusFlags : byte

{

Read = 0b0001,

Write = 0b0010,

SecondWord = 0b0100

}

public class Register

{

private BusFlags \_flags;

private Processor \_proc;

private Bus \_bus;

// Setup internal value and getter/setter protection.

private byte[] \_value;

public byte[] Value

{

get => \_value;

set => \_value = value;

}

public Register(Processor proc, Bus bus, int length)

{

\_proc = proc;

\_proc.ClockRising += ClockRisingHandler; // Hook onto events.

\_proc.ClockFalling += ClockFallingHandler;

\_bus = bus;

\_value = new byte[length];

}

public void SetFlag(BusFlags flag)

{

\_flags = \_flags | flag;

}

private void ClockRisingHandler()

{

if (\_flags.HasFlag(BusFlags.Read))

{

var outputArray = new byte[2];

Array.Copy(\_value, \_flags.HasFlag(BusFlags.SecondWord) ? 2 : 0, outputArray, 0, 2);

\_bus.Write(outputArray);

}

}

private void ClockFallingHandler()

{

if (\_flags.HasFlag(BusFlags.Write))

{

var inputArray = \_bus.Read();

Array.Copy(inputArray, 0, \_value, \_flags.HasFlag(BusFlags.SecondWord) ? 2 : 0, 2);

}

\_flags = 0; // Reset flags.

}

}

}

### ALU.cs

using System;

using System.Windows.Forms;

namespace EPS.Components

{

// Possible ALU operation modes.

public enum ALUModes: byte

{

None,

Increment,

IncrementWord,

Decrement,

Add,

Subtract,

And,

Or,

Xor

}

public class ALU

{

private Register \_acc; // Special reference to ACC. Can access without using main bus.

private Processor \_proc;

private Bus \_bus;

private ALUModes \_aluMode;

public ALU(Processor proc, Bus bus, Register acc)

{

\_acc = acc;

\_bus = bus;

\_proc = proc;

\_proc.ClockFalling += ClockFallingHandler;

}

public void SetMode(ALUModes mode)

{

\_aluMode = mode;

}

/// <summary>

/// Calculates answer and writes this into the Accumulator. Pulls data from bus hence falling.

/// </summary>

private void ClockFallingHandler()

{

if (\_aluMode != ALUModes.None) // save execution time in case of no op.

{

var busValue = BitConverter.ToInt16(\_bus.Read(), 0);

var accValue = BitConverter.ToInt16(\_acc.Value, 0);

switch (\_aluMode)

{

case ALUModes.Increment:

\_acc.Value = BitConverter.GetBytes(busValue += 1);

break;

case ALUModes.IncrementWord:

\_acc.Value = BitConverter.GetBytes(busValue += 2);

break;

case ALUModes.Add:

\_acc.Value = BitConverter.GetBytes(busValue + accValue);

break;

case ALUModes.Subtract:

\_acc.Value = BitConverter.GetBytes(accValue - busValue);

break;

}

}

\_aluMode = 0; // Reset flags.

}

}

}

### MemoryBank.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms.VisualStyles;

namespace EPS.Components

{

// Possible memory operation modes.

[Flags]

public enum MemoryFlags : byte

{

Read = 0b0001,

Write = 0b0010,

TwoBytes = 0b0100

}

public class MemoryBank

{

private readonly byte[] \_data = new byte[65535]; // Maximal addressable size in case of 16 bit system, with the minimal addressable unit being byte.

private MemoryFlags \_flags;

private readonly Register \_mdr;

private readonly Register \_mar;

public MemoryBank(Processor proc, Register mdr, Register mar)

{

\_mdr = mdr;

\_mar = mar;

}

public void SetFlag(MemoryFlags flag)

{

\_flags = \_flags | flag;

}

public void ClockFallingHandler()

{

UInt16 marAddress = BitConverter.ToUInt16(\_mar.Value, 0); // Cast to Uint from Byte for easier handling.

if (\_flags.HasFlag(MemoryFlags.Read))

{

\_mdr.Value[0] = \_data[marAddress];

if (\_flags.HasFlag(MemoryFlags.TwoBytes))

\_mdr.Value[1] = \_data[marAddress + 1];

else

\_mdr.Value[1] = 0; //Reset most significant byte. Don't want to copy previous value.

} else if (\_flags.HasFlag(MemoryFlags.Write))

{

\_data[marAddress] = \_mdr.Value[0];

if (\_flags.HasFlag(MemoryFlags.TwoBytes))

\_data[marAddress + 1] = \_mdr.Value[1];

}

\_flags = 0; // Reset flags.

}

// Allow memory bank to be indexed as if it was an array.

public byte this[UInt16 addr]

{

get { return \_data[addr]; }

set

{

\_data[addr] = value;

}

}

}

}

### Bus.cs

namespace EPS.Components

{

public class Bus

{

// Use getter setter for protection of internal value

private byte[] \_value;

public byte[] Value

{

get => \_value;

set => \_value = value;

}

public Bus(int length)

{

\_value = new byte[length];

}

public void Write(byte[] data) => \_value = data;

public byte[] Read() => \_value;

}

}

### Instruction.cs

using System.Collections.Generic;

namespace EPS.Instructions

{

public delegate int? InstructionStage(Processor proc, ProcessorExecutionContext ctx); // Setup delgate for InstructionStage anonynmous functions.

public class Instruction

{

public int currentStage = 0; // current stage of execution of instruction.

public List<InstructionStage> InstructionStages = new List<InstructionStage>(); // All stages of execution.

public string Mnemonic = ""; // Short code, i.e LDA

public string Description = ""; // Longer explanation for user.

public byte OpCode;

/// <summary>

/// Executes the substages of micro-code

/// </summary>

/// <returns>True when complete</returns>

public bool Execute(ProcessorExecutionContext ctx)

{

ctx.StageCount = InstructionStages.Count;

currentStage = InstructionStages[currentStage](ctx.Proc, ctx) ?? currentStage + 1; // Jump to specified microcode or just increment stage.

if (currentStage != InstructionStages.Count) return false;

currentStage = 0; // If at end, reset.

return true; // Signal system that an instruction is ready for execution.

}

}

}

### InstructionSet.cs

using System;

using System.Collections.Generic;

using EPS.Components;

namespace EPS.Instructions

{

public class InstructionSet

{

public Instruction[] Instructions = new Instruction[64];

public InstructionSet()

{

Instructions[0b00\_000000] = new Instruction

{

Mnemonic = "NOP",

Description = "No operation. Consumes one processor execution cycle.",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => // 0: do nothing....

{

return null;

}

}

};

Instructions[0b00\_000001] = new Instruction

{

Mnemonic = "ADD",

Description = "Sets ACC to RegA, then adds value from bus to ACC and stores in Reg A.",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: RegA to ACC

{

ctx.RegA.SetFlag(BusFlags.Read);

ctx.Proc.ACC.SetFlag(BusFlags.Write);

return ctx.MemoryMode == MemoryMode.Direct ? 1 : 2; // step 1 if memory mode direct, else 2

},

(proc, ctx) => // 1: Direct only. Load into MDR

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => MAR

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.MAR.SetFlag(BusFlags.Write);

proc.MemoryBank.SetFlag(MemoryFlags.Read);

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

}

return null;

},

(proc, ctx) => // 2: Trigger add operation. Result will sit in the ACC.

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.MDR.SetFlag(BusFlags.Read);

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read);

} else if (ctx.MemoryMode == MemoryMode.Immediate)

{

proc.CIR.SetFlag(BusFlags.Read);

proc.CIR.SetFlag(BusFlags.SecondWord);

}

proc.ALU.SetMode(ALUModes.Add);

return null;

},

(proc, ctx) => // 3: Acc to RegA

{

ctx.Proc.ACC.SetFlag(BusFlags.Read);

ctx.RegA.SetFlag(BusFlags.Write);

return null;

}

}

};

Instructions[0b00\_000010] = new Instruction

{

Mnemonic = "SUB",

Description = "Sets ACC to RegA, (ACC-Bus) and stores in Reg A.",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: RegA to ACC

{

ctx.RegA.SetFlag(BusFlags.Read);

ctx.Proc.ACC.SetFlag(BusFlags.Write);

return ctx.MemoryMode == MemoryMode.Direct ? 1 : 2;

},

(proc, ctx) => // 1: Direct only. Load into MDR

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => MAR

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.MAR.SetFlag(BusFlags.Write);

proc.MemoryBank.SetFlag(MemoryFlags.Read);

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

}

return null;

},

(proc, ctx) => // 2: Trigger sub operation. Result will sit in the ACC.

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.MDR.SetFlag(BusFlags.Read);

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read);

} else if (ctx.MemoryMode == MemoryMode.Immediate)

{

proc.CIR.SetFlag(BusFlags.Read);

proc.CIR.SetFlag(BusFlags.SecondWord);

}

proc.ALU.SetMode(ALUModes.Subtract);

return null;

},

(proc, ctx) => // 3: Acc to RegA

{

ctx.Proc.ACC.SetFlag(BusFlags.Read);

ctx.RegA.SetFlag(BusFlags.Write);

return null;

}

}

};

Instructions[0b00\_000011] = new Instruction

{

Mnemonic = "BRA",

Description = "Jumps to RegB, Immediate or Direct",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: Initiate memory transfer for direct

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => MAR

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.MAR.SetFlag(BusFlags.Write);

proc.MemoryBank.SetFlag(MemoryFlags.Read); // Config mem bank to read two bytes

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read); // RegB => PC

proc.PC.SetFlag(BusFlags.Write);

} else

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => PC

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.PC.SetFlag(BusFlags.Write);

}

return ctx.MemoryMode == MemoryMode.Direct ? 1 : ctx.StageCount; // if direct mem mode, move to next step, else finish

},

(proc, ctx) => //1: Set PC value

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.MDR.SetFlag(BusFlags.Read); // MDR => PC

proc.PC.SetFlag(BusFlags.Write);

}

return null;

}

}

};

Instructions[0b00\_000100] = new Instruction

{

Mnemonic = "BRZ",

Description = "Jumps to RegB, Immediate or Direct when RegA is 0.",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: Initiate memory transfer for direct

{

if (BitConverter.ToInt16(ctx.RegA.Value, 0) != 0) // If non-zero, end instruction.

{

return ctx.StageCount;

}

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => MAR

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.MAR.SetFlag(BusFlags.Write);

proc.MemoryBank.SetFlag(MemoryFlags.Read); // Config mem bank to read two bytes

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read); // RegB => PC

proc.PC.SetFlag(BusFlags.Write);

} else if (ctx.MemoryMode == MemoryMode.Immediate)

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => PC

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.PC.SetFlag(BusFlags.Write);

}

return ctx.MemoryMode == MemoryMode.Direct ? 1 : ctx.StageCount; // if direct mem mode, move to next step, else finish

},

(proc, ctx) => //1: Set PC value

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.MDR.SetFlag(BusFlags.Read); // MDR => PC

proc.PC.SetFlag(BusFlags.Write);

}

return null;

}

}

};

Instructions[0b00\_000101] = new Instruction

{

Mnemonic = "BRP",

Description = "Jumps to RegB, Immediate or Direct when RegA >= 0.",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: Initiate memory transfer for direct

{

if (BitConverter.ToInt16(ctx.RegA.Value, 0) < 0) // If less than zero, end instruction.

{

return ctx.StageCount;

}

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => MAR

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.MAR.SetFlag(BusFlags.Write);

proc.MemoryBank.SetFlag(MemoryFlags.Read); // Config mem bank to read two bytes

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read); // RegB => PC

proc.PC.SetFlag(BusFlags.Write);

} else if (ctx.MemoryMode == MemoryMode.Immediate)

{

proc.CIR.SetFlag(BusFlags.Read); // CIR Second Word => PC

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.PC.SetFlag(BusFlags.Write);

}

return ctx.MemoryMode == MemoryMode.Direct ? 1 : ctx.StageCount; // if direct mem mode, move to next step, else finish

},

(proc, ctx) => //1: Set PC value

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

proc.MDR.SetFlag(BusFlags.Read); // MDR => PC

proc.PC.SetFlag(BusFlags.Write);

}

return null;

}

}

};

Instructions[0b00\_000110] = new Instruction

{

Mnemonic = "MOV",

Description = "Moves a value into RegA from RegB, Immediate",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: Read from CIR, or RegB into RegA

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

return null; // Direct mode not supported.

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read);

} else if (ctx.MemoryMode == MemoryMode.Immediate)

{

proc.CIR.SetFlag(BusFlags.SecondWord);

proc.CIR.SetFlag(BusFlags.Read);

}

ctx.RegA.SetFlag(BusFlags.Write); //Read bus into RegA

return null;

}

}

};

Instructions[0b00\_000111] = new Instruction

{

Mnemonic = "STA",

Description = "Saves a value from RegA into memory address specified by RegB or Immediate",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: Setup MAR

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

return ctx.StageCount; // Direct memory mode not supported.

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read); // RegB => Bus

} else

{

proc.CIR.SetFlag(BusFlags.Read); // Immediate Val => Bus

proc.CIR.SetFlag(BusFlags.SecondWord);

}

proc.MAR.SetFlag(BusFlags.Write); // Bus => MAR

return null;

},

(proc, ctx) => //1: Setup MDR

{

ctx.RegA.SetFlag(BusFlags.Read); // RegA => Bus

proc.MDR.SetFlag(BusFlags.Write); // Bus => MDR

proc.MemoryBank.SetFlag(MemoryFlags.Write); // Trigger memory write of length two bytes

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

return null;

}

}

};

Instructions[0b00\_001000] = new Instruction

{

Mnemonic = "LDA",

Description = "Loads a value from Memory Address specified by RegB or Immediate",

InstructionStages = new List<InstructionStage> {

(proc, ctx) => //0: Setup MAR

{

if (ctx.MemoryMode == MemoryMode.Direct)

{

return ctx.StageCount; // Direct memory mode not supported.

} else if (ctx.MemoryMode == MemoryMode.Register)

{

ctx.RegB.SetFlag(BusFlags.Read); // RegB => Bus

} else

{

proc.CIR.SetFlag(BusFlags.Read); // Immediate Val => Bus

proc.CIR.SetFlag(BusFlags.SecondWord);

}

proc.MAR.SetFlag(BusFlags.Write); // Bus => MAR

proc.MemoryBank.SetFlag(MemoryFlags.Write); // Trigger memory read of length two bytes

proc.MemoryBank.SetFlag(MemoryFlags.TwoBytes);

return null;

},

(proc, ctx) => //1: Setup MDR

{

proc.MDR.SetFlag(BusFlags.Read); // MDR => Bus

ctx.RegA.SetFlag(BusFlags.Write); // Bus => RegA

return null;

}

}

};

}

}

}

### Memory Bank Control

using System;

using System.Windows.Forms;

namespace EPS

{

public partial class MemoryBank : UserControl

{

private readonly MemoryBankRow[] \_rows = new MemoryBankRow[8]; // 8 rows of 8 gives 64 bytes per page

private Int32 \_memoryStartAddress; // Address to start memorybank on.

public Int32 MemoryStartAddress

{

get => \_memoryStartAddress;

private set

{

\_memoryStartAddress = value;

Setup(); // Trigger setup on change of page. This setup propoagates down to the individual cell.

}

}

public MemoryBank()

{

InitializeComponent();

for (int i = 0; i < 8; i++)

{

\_rows[i] = new MemoryBankRow {Location = new System.Drawing.Point(0, i \* 65)}; // Setup rows and positions programatically.

Controls.Add(\_rows[i]);

}

Setup();

}

public void Setup(EPS.Components.MemoryBank memory = null)

{

for (int i = 0; i < 8; i++)

{

\_rows[i].Setup(MemoryStartAddress + i \* 8, memory); // Trigger setup propagation.

}

}

public void StepMemoryPage(int pages) // Helper method to ensure steps are equal 64 byte chunks.

{

MemoryStartAddress += 64 \* pages;

}

}

}

### Memory Bank Row Control

using System;

using System.Windows.Forms;

namespace EPS

{

public partial class MemoryBankRow : UserControl

{

private readonly MemoryBankCell[] \_cells = new MemoryBankCell[8]; // 7 cells of 1 byte = 8 bytes per row.

public MemoryBankRow()

{

InitializeComponent();

for (int i = 0; i < 8; i++)

{

\_cells[i] = new MemoryBankCell

{

Location = new System.Drawing.Point((i \* 120) + 120, 0) // Setup positions programtically

};

Controls.Add(\_cells[i]);

}

}

public void Setup(Int32 memoryStart, EPS.Components.MemoryBank memory = null)

{

lblAddressStart.Text = Convert.ToString(memoryStart, 2).PadLeft(16, '0'); // String formatting for binary style to 16 bits.

lblAddressEnd.Text = Convert.ToString(memoryStart + 7, 2).PadLeft(16, '0');

for (int i = 0; i < 8; i++)

{

\_cells[i].Setup(memoryStart + i, memory); // Propogate setup to cells.a

}

}

}

}

### Memory Bank Cell Control

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Drawing;

using System.Data;

using System.Linq;

using System.Net.Configuration;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace EPS

{

public partial class MemoryBankCell : UserControl

{

private Components.MemoryBank \_memory = null;

private Int32 \_memoryAddress;

public MemoryBankCell()

{

InitializeComponent();

}

public void Setup(Int32 memoryAddress, EPS.Components.MemoryBank memory = null) // Final stage of setup.

{

string memoryAddressString = Convert.ToString(memoryAddress, 2).PadLeft(3, '0'); // String format val

lblAddress.Text = "Address " + memoryAddressString.Substring(memoryAddressString.Length - 3); // Ensure length is correct

\_memory = memory ?? \_memory; // Handle memory being null.

\_memoryAddress = memoryAddress;

UpdateCell();

}

public void UpdateCell()

{

if (!(\_memory is null))

{

txtBox.Text = Convert.ToString(\_memory[(ushort)\_memoryAddress], 2).PadLeft(8, '0'); // String format to 8 bits.

}

}

private void ValueChanged(object sender, EventArgs e)

{

// Provide binary value validation

if (txtBox.Text.Length > 0)

{

try

{

\_memory[(ushort) \_memoryAddress] = Convert.ToByte(txtBox.Text, 2); // Update value

}

catch (Exception err) // Convert throws exception if it cannot cast.

{

MessageBox.Show("That's an invalid binary string!"); // Inform user.

UpdateCell(); // Reset to prev value

}

}

}

}

}

### Memory Bank Controls Control

using System;

using System.Windows.Forms;

namespace EPS

{

public partial class MemoryBankControls : UserControl

{

private MemoryBank \_memoryBank = null;

public MemoryBankControls()

{

InitializeComponent();

}

public void Setup(MemoryBank memoryBank)

{

\_memoryBank = memoryBank;

UpdateLabels();

}

private void StepBack(object sender, EventArgs e)

{

\_memoryBank.StepMemoryPage(-1); // Use page step helper to ensure pages remain equal size.

UpdateLabels();

}

private void StepForward(object sender, EventArgs e)

{

\_memoryBank.StepMemoryPage(1);

UpdateLabels();

}

private void UpdateLabels()

{

lblMemoryStart.Text = Convert.ToString(\_memoryBank.MemoryStartAddress, 2).PadLeft(16, '0'); // Binary formatting to 16 bit number even if shorter.

lblMemoryEnd.Text = Convert.ToString(\_memoryBank.MemoryStartAddress + 63, 2).PadLeft(16, '0');

btnStepPageBack.Enabled = !(\_memoryBank.MemoryStartAddress - 64 < 0); // Prevent user overflowing.

btnStepPageForward.Enabled = !(\_memoryBank.MemoryStartAddress + 127 > 65535); // Ensure MemoryEndAddress will not overflow

}

}

}

References

|  |  |
| --- | --- |
| [1] | S. Heule, "How Many x86-64 Instructions Are There Anyway?," 7 March 2016. [Online]. Available: https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/. |

During development, I made use of <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/> and <https://docs.microsoft.com/en-us/dotnet/api/> for information pertaining to the language and the .net library.