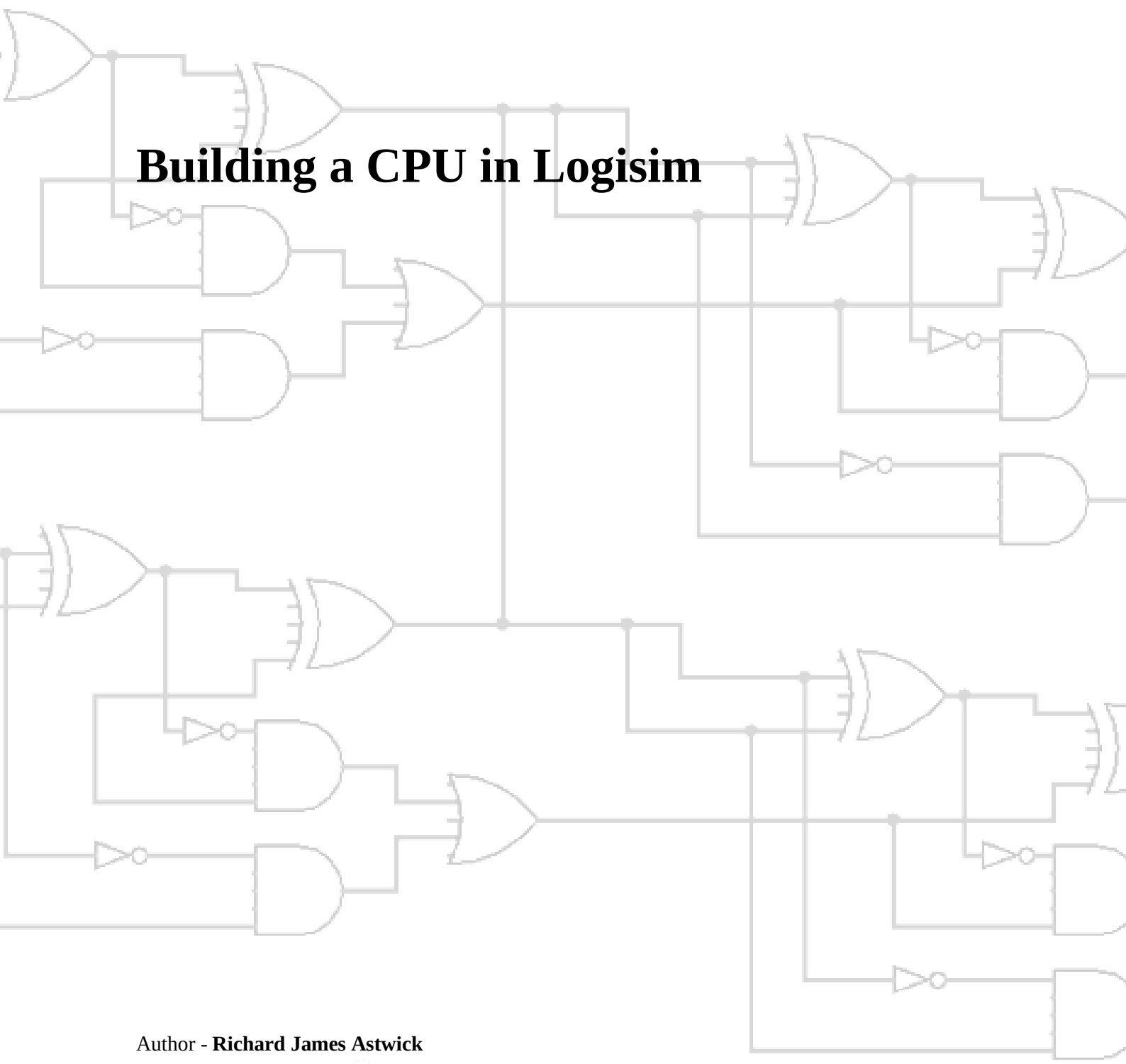


Building a CPU in Logisim

Author - **Richard James Astwick**

Module Code - **Com3600**

Supervisor - **Dirk Sudholt**

6/05/2015

This report is submitted in partial fulfilment of the requirement for the degree of Bachelors of Computer Science by Richard James Astwick

Signed Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Richard James Astwick

Signature:

Date: 06/05/2015

Abstract

The aim of this project is to demonstrate how every CPU, no matter how complex, can be designed from just a few basic building blocks. To show this a CPU was designed and created, using circuit design software called Logisim, capable of carrying out a number of operations. The CPU was constructed almost completely from elementary gates (AND, OR, NOT) and an entire custom assembly language was designed and built specifically for this unit. This report summarises research about how a CPU works and what components it contains in a number of structured sections. The software itself, Logisim, is explored and analysed. The design process is documented and design decisions justified. The project has been successfully implemented and a YouTube video presenting the CPU can be found at https://youtu.be/fFgeF9XXA0w.

Acknowledgements

I would like to thank Oxford University Press for giving me permission to use some of the images from A.Clements Principles of Computer Hardware.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

Term	Abbreviation	Definition
Central processing unit	CPU	Piece of computer hardware which carries out instructions of a computer program
Program counter	PC	Register that stores the memory address of the next instruction to be executed
Memory address register	MAR	Register that stores the memory address of the current instruction to be executed
Register transfer language	RTL	An alternative representation of assembly language
Memory buffer register (memory data register)	MBR (MDR)	Register to hold data once it has been read from memory
Instruction register	IR	Register to hold the current instruction being executed
Operation code	Op-code	Binary string that represents the operation of an instruction
Operand		An entity on which an operation is performed. Can be a memory address
Control unit	CU	Circuit that sets the control signals. Firstly to fetch an instruction from memory and then execute it.
Data register		Register to store data during a calculation
Arithmetic and logic unit	ALU	A circuit which performs all arithmetic and logical operations inside the CPU
Condition code register	CCR	Stores the value of the ALU's status flags after an operation has occurred
Instruction set architecture	ISA	A specification defining the composition of instructions the CPU can execute
Complex instruction set computer	CISC	A CPU design in which instructions can be executed on the contents of memory locations and are performed as several low-level operations
Reduced instruction set computer	RISC	A CPU design in which instructions can only be executed on contents of registers, allow it to achieve one instruction per clock cycle
Full adder	FA	Circuit to add two individual bits and a carry in to generate a sum bit and a carry out bit
Half adder	HA	Circuit to add two individual bits generating a sum and a carry out bit
Random access memory	RAM	Memory that can be read to or read from
Read only memory	ROM	Memory that can only be read from
Sequencer (timing signal generator)		Circuit to generate the timing signals used in the control unit
Binary coded decimal	BCD	Decimal representation of a binary number in which a designated number of bits are used to encode each digit

Chapter 1: Introduction

1.1 - Background

Students at the University of Sheffield have the opportunity to study a first year course entitled Com1006 Devices and Networks. Some of the objectives of this course are to introduce students to the idea of computer arithmetic and logical gates and how this is implemented in a computer. The course touches briefly on the structure of a computer's central processing unit (CPU) and how information flows in a CPU. It aims to make students interested in computer architecture and possibly pursue a career in it. These ideas are the basis for this project.

1.2 – Project Description

The project proposed by Dirk Sudholt is to design and create a CPU in a piece of open-source logic simulator software called Logisim which is often used for teaching. The CPU needs to be capable of carrying out a number of logical operations on 8-bit words such as logical AND, OR, NOT and XOR; as well as a number of arithmetic operations such as arithmetic addition, subtraction and multiplication. It also needs to be capable of simulating basic memory and the CPU should also support at least two different addressing modes. A requirement for the CPU is for it to be constructed from only elementary gates such as AND, OR and NOT. This is to help convey the idea that computers themselves are made up of nothing more than elementary gates and simple operations.

The purpose of this project is to give students taking the Devices and Networks module an idea of what can potentially be accomplished from learning the skills and knowledge contained in the module. The students will hopefully be inspired by this and will continue to be interested in the topic after they have finished their course. The simulation will be shown as an educational tool to the students at some point during the module either as a part of a demonstration or as extra material to view. Alongside the CPU simulation in Logisim, the dissertation report will be available for students to read. The aim of the report is to further aid the students understanding of how the CPU works. For this to be beneficial for education the report has been written in a structured educational manner that can be easily followed by the first year students.

1.3 – Analysis of Constraints

For this project, a strong understanding of the structure of the CPU is required. Understanding how the CPU works and carries out operations is fundamental to the project. Thorough research needs to be carried out to develop this understanding.

When designing the CPU each part is designed separately. For example the arithmetic and logic unit and the memory were designed at different stages. This allows complications to arise when implementing all parts together in Logisim; some parts may not work correctly with others. Testing also needs to be considered at an early stage. Logisim offers some premade input and output features that can be used to test if the CPU was working correctly. Some test programs have been designed to run using a newly developed machine language. A testing plan will be created in the design stage of the project like in any software development. Testing has the potential to expose issues with the CPU and so some components may need to be re-evaluated and therefore may not match the original component designs.

Another problem that needs to be addressed is the environment in which the CPU is being designed. It is important that the CPU is created in Logisim as students studying the Com1006 module learn how to use Logisim in lab classes and so will be able to relate to the project. This means that an expert understanding of how Logisim works and what tools it offers is needed to successfully complete a complex simulation of this nature. This problem can be overcome by spending time practicing building circuits in Logisim and experimenting with all the features that the software offers.

There are also many guides and tutorials, such as Carl Burch's website (Logisim's creator), to help users understand how to use Logisim.

Logisim offers a number of premade modules. However for the project, only basic elements (AND, OR, NOT) were used to construct these modules in a custom manner. The aim of the project is to make as much of the CPU out of these basic logic gates as possible. It is challenging to complete the CPU at the most basic level and some flexibility may be needed to complete the project without using some of Logisim's premade modules.

When writing this dissertation report it is important to keep in mind the project's educational theme and aim. The purpose of this report is to help students understand in detail how the CPU is designed and how each part works. It is important for this report to be well structured so that it is easy to follow and understand so as not to confuse readers who do not have much prior knowledge of the subjects discussed.

1.4 – Achievements

As will be explained in detail in chapters 6 and 7, the achievements of the project can be summarised as follows. A CPU was successfully implemented in Logisim meeting the specification defined in section 1.2. Designing a CPU is a complex task and the fact that this CPU was made just from elementary gates makes this achievement even greater. The CPU runs instructions based on a carefully designed custom instruction set architecture which was designed to be compact to encode a large amount of behaviour in relatively small number of instruction bits. As a result the CPU can demonstrate good functionality while remaining small.

1.5 – Overview

In the upcoming chapter we will look at background information revolving around the CPU such as information flow inside the CPU, how the CPU is structured, how it carries out tasks and how to implement the CPU concept into a working piece of hardware. Along with CPU design we will look at instruction set architecture, what it is, how to form instructions for the CPU to execute and different addressing modes that the CPU can implement. Finally in the last section of the chapter we shall touch briefly on the Logisim software being used to simulate the CPU and discuss its benefits and drawbacks.

Chapter 3 contains a list of requirements for the project and an in depth analysis of each of these requirements in turn. It also contains a section about how the project will be evaluated and tested once completed.

Chapter 4 describes the design process that was undertaken in order to create the working CPU and discusses why specific designs for circuits were chosen above others. Designs for individual circuits will be looked at in detail and the instruction set architecture is also defined. The design chapter also contains some test programs that the CPU should be capable of executing.

The implementation of the project is contained in chapter 5. Key areas where the implementation differs from the design are highlighted and reasons for the changes are discussed. The last section of chapter 5 highlights the testing procedure that was undertaken in order to check that the CPU was functioning correctly.

Chapter 6 describes the results of the project and evaluates the implementation against the set of requirements defined in chapter 3. Further developments that could be made to the project are also contained in chapter 6. Finally chapter 7 concludes the report with a summary of how successful the project was and lessons learnt from undertaking the project. The appendices contain complete control signal tables and circuit diagrams for all circuits created during the project.

Chapter 2: Background

To begin this project we need to look at the research that will help us to fully understand how the CPU works. It is beneficial to break down the CPU into individual parts to develop a deeper understanding of how these parts work together to complete the role of the CPU.

We will take a close look at Instruction Set Architecture, what a machine language is and how to design custom machine language programs for the CPU to run, which addressing modes to use and how this affects the design of the CPU itself.

Finally we will look at the Logisim software and discuss its benefits, why it is being used and what functionality it has. We will also discuss and critic other software of a similar nature.

2.1 – Structure of the CPU

The CPU is a complex piece of computer hardware. Therefore breaking it down into separate parts or modules will improve understanding of how it works rather than looking at the CPU as a whole. However it is important to understand what the CPU's role in a computer is before we can understand how it works.

The CPU can be thought of as the brain of the computer. Input is interpreted and processed by the CPU and an output is produced which controls the computer as a whole. The CPU is the most important part of a computer and is complex in design. This reflects the difficulty of the project and shows how much of a challenge it is. It also shows why the project was initially proposed.

A central processing unit is a piece of hardware inside a computer that "carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output operations of the system." [1] The CPU retrieves instructions called a program from memory. These instructions are interpreted by the CPU and are then executed. This is a brief high level overview of how the CPU functions and we will now look at specific parts to understand its internal workings.

2.1.1 – Information Flow

There are three types of information flow in a computer. Firstly "an address represents the location of a data element within memory." [2] This is important as the CPU will need to read the instruction from the computer's memory before it can start the execution process. Addresses flow from one part of the CPU to another by travelling along address paths. The next type of information flow is data. "Data comprises the instructions, constants and variables that are stored in memory and registers." [2] Data paths will be used if the instruction requires some type of arithmetic or logical calculation. Data to be used in the calculation will flow along data paths inside the CPU. Finally the last type of information flow is control. "Control paths comprise the signals that trigger events, provide clocks and control the flow of data and addresses throughout the computer." [2]

Instructions are carried out in a two stage process, called the fetch-execute cycle, where firstly the "instruction is read from memory and decoded by the control unit. This is followed by an execution phase in which the control unit generates all the signals necessary to execute the instruction." [2] We will now look at the fetch phase of the fetch – execute cycle.

2.1.2 – Fetch Phase

For the CPU to execute an instruction the instruction must be brought to the CPU from the computer's memory. To do this the program counter (PC) is used. "The program counter contains the address of the next instruction in memory to be executed." [2] This means that the program counter points to the next instruction so if `[PC] = 10` then the next instruction to execute is held in memory location 10.

The start of the fetch cycle is when the contents of the program counter moves to the memory address register (MAR). After this has happened the program counter's contents is incremented by a value and moved back to the program counter. This means that at the end of the operation the program counter points to the next instruction while the current instruction is being executed. From this we can see that the memory address register contains a copy of what the program counter previously contained.

Data needs to be fetched from the computer's memory for use in the instruction; the memory address register allows us to do this. The memory address register "holds the address of the location in memory in to which data is being written in a write cycle or from which data is being read in a read cycle." [2] After the data or instruction have been fetched from memory the "contents of the memory location specified by the memory address register are read from the memory and transferred to the memory buffer register (MBR)." [2] This is a memory read cycle. Clements chooses to represent the data transfer in register transfer language (RTL) as [MBR] ← [[MAR]] with [[MAR]] meaning "the contents of the memory whose address is given by the contents of the MAR." [2] The memory buffer register, sometimes referred to as memory data register (MDR), is a "temporary holding place for data received from memory in a read cycle, or for data to be transferred to memory in a write cycle." [2] The data stored in the memory buffer register is now a bit pattern of the instruction to be performed.

After being stored in the memory buffer register the instruction is then sent to the instruction register (IR). The instruction register holds the instruction while it is being decoded, "which is split into two fields, operation code (op-code) and operand field." [3] The operation code tells the CPU what operation is to be carried out whereas the operand field contains the address of data to be used when carrying out the instruction. The op-code from the instruction register is given to the control unit (CU). The control unit emits a series of control signals that are produced with a stream of clock pulses that are used to control the computer. For example the control unit has many roles such as "moving the contents of the program counter to the memory address register, executing a read cycle and moving the memory buffer register contents to the instruction register." [2] This concludes the fetch phase of the fetch-execute cycle.

2.1.3 – Execute Phase

We have seen how during the fetch phase the instruction is fetched from memory and decoded by the control unit. We shall now look at the "execute phase in which the control unit generates all the signals to execute the instruction." [2]

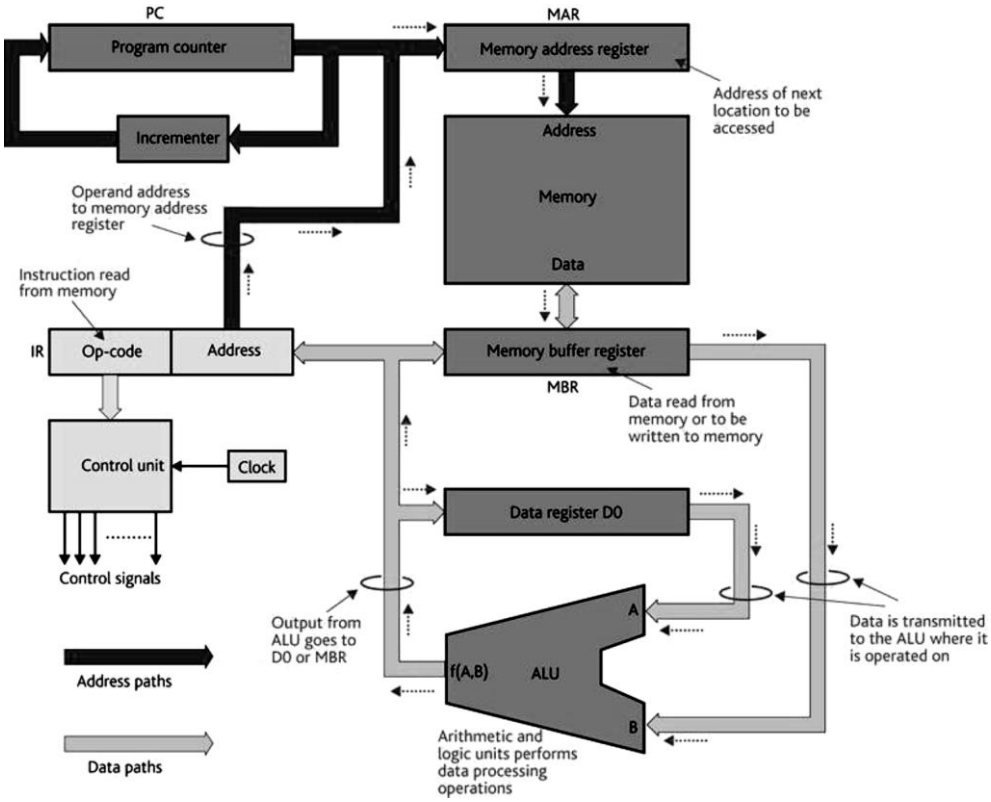


Figure 2.1 CPU's address paths and data paths [3]

Figure 2.1 shows an illustration of the structure of the CPU including its data paths and address paths. From this diagram we can clearly see each component of the CPU as described previously and how they interact together along address and data paths. We shall now analyse the two components that have not been explained, the data register (called D0 in this case) and the arithmetic and logic unit (ALU).

The data register "holds temporary results during a calculation." [3] This register is important because without intermediate values being stored here they would have to be stored in main memory. This creates inefficiency because "main memory is slower than accessing a register because the technology used for the main memory is slower and cheaper than that used for a register." [4]

The arithmetic and logic unit carries out all calculations in the CPU. As can be seen from figure 2.1 the arithmetic and logic unit can take input from the memory buffer register or the data register. This means that "arithmetic and logical operations are applied either to the contents of the data register or memory buffer register alone, or to the contents of the data register and the contents of the memory buffer register." [2] Again as can be seen from figure 2.1 the output of the arithmetic and logic unit is sent back into the data register as input or to the memory buffer register.

Operation	Type
Addition	Arithmetic
Subtraction	Arithmetic
Multiplication	Arithmetic
Division	Arithmetic
AND	Logical
OR	Logical
NOT	Logical
XOR	Logical

The arithmetic and logic unit carries out a number of operations of two types, as the name suggests, arithmetic and logical operations. The difference between arithmetic operations and logical operations is that logical operations do not generate a carry bit.

Table 2.1 Common arithmetic and logical operations [2]

To see how the CPU executes an instruction in practice we can examine the execution of `ADD P, D0` in RTL, which adds the data specified at address P and the contents of D0 with the result being written back into D0.

FETCH	[MAR]	←	[PC]
	[PC]	←	[PC] + 4
	[MBR]	←	[[MAR]]
	[IR]	←	[MBR]
	CU	←	[IR(op-code)]
ADD	[MAR]	←	[IR(address)]
	[MBR]	←	[[MAR]]
	ALU	←	[MBR], ALU ← [D0]
	[D0]	←	ALU

Table 2.2 Execution of `ADD P, D0` in RTL [3]

As described in the previous section during the instruction fetch phase the contents of the program counter are transferred to the memory address register; the contents of the program counter are then incremented by 4. The memory address register now contains the previous copy of the program counter. "A memory read cycle is performed that reads the instruction from an address pointed to by the memory address register" [3] and is transferred to the memory buffer register. The instruction is then transferred to the instruction register where it is split into two fields, op-code and address. The op-code is copied to the control unit and used to carry out the instruction. The execution phase is where the addition is performed. The operand address field is sent to the memory address register where it is used to read the data from memory that is needed in the addition. The data is then sent to the memory buffer register. Next the addition is carried out. We can see that two operations are carried out concurrently and the result is moved from the arithmetic and logic unit into the data register D0. Instructions such as [MAR] ← [PC] are known as microinstructions.

Now that we know how the CPU executes instructions in a sequential manner we can now look at branching instructions. "A branch instruction modifies the flow of control and causes the program to continue execution at the target address specified by the branch." [5] There are two types of branch instructions, unconditional and conditional. An example of unconditional branching is a simple instruction which causes the program to jump to the instruction at the address specified. A computer needs to be able to execute a test and branch on the outcome of that test, this is conditional branching. An example of a conditional branch instruction is branch on equal (`BEQ`) in which a branch to a predefined set of instructions occurs if the last result was zero as opposed to continuing down the list of instructions in memory sequentially.

The machine in figure 2.1 is not capable of performing branched computation. Some additions need to be made in order to achieve this. A condition code register (CCR) is used to record the state of the ALU's various flags; carry, negative, zero and overflow. If a conditional branch instruction is being executed it will look at the state of the condition code register and the "control unit will then either force the CPU to execute the next instruction in series or to branch another instruction somewhere in the program." [2] In order for the control unit to check the state of the condition code register a path between them is needed. The final addition is a path between the operand field, in the instruction register, and the program counter. This allows the program counter to know which instruction is to come next as a result of the branching condition.

The CPU needs to be able to execute instructions where one of the operands is a literal value. A literal value is where the operand is the actual value being used for example the number 4 as opposed to the operand being a memory or register address. To execute such instructions the CPU needs a path

"between the operand field of the instruction register and the data register and ALU." [2] This allows the literal value to be stored in the data register or used directly as part of a calculation in the ALU.

2.1.4 – Abstract to Implementation

Now that we have seen how a theoretical abstract CPU works we can now look at how a basic CPU can be implemented. We will now see how we can create a set of machine-level instructions, from a binary string, capable of executing an instruction such as `ADD P, D0`.

There are two ways of accomplishing this. Firstly, "logic could be created that directly transforms instructions into control signals." [3] This is known as a random logic control unit. Another way is to construct a micro-programmed control unit which is a type of computer that creates control signals from a machine level instruction, input as a program.

The random logic control unit uses hard-wired gates and flip-flops to create its control signals. When designing a random logic control unit the instruction first needs to be defined in register transfer language. The instruction then needs to be written as a set of micro-instructions also in register transfer language. Once this has been achieved we can create the logic elements needed to execute each micro-instruction. The arrangement of gates in the construction of a random logic control unit varies greatly from computer to computer. This means that the "random logic control unit is dedicated to a specific CPU and cannot be easily modified." [2] On the other hand the same micro-programmed control unit can be easily modified to suit many different computers. This is a clear advantage however hard-wired units are usually faster. Micro-programmed units are "difficult to handle for system software and compilers." [3]

We shall now look at the implementation of a basic CPU in figure 2.2 that allows us to see what control signals need to be produced.

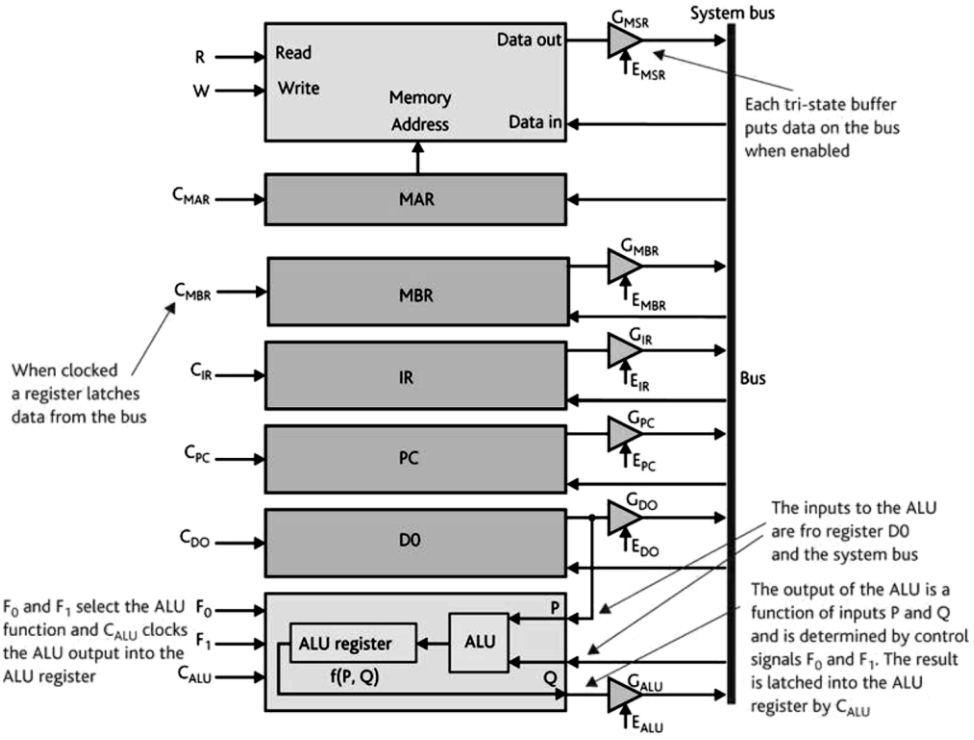


Figure 2.2 Single-bus CPU [3]

As we can see this CPU connects all registers and memory together with a single system bus. This "structure allows the memory to transfer data directly to or from any register" [2] therefore does not have to always pass through the memory buffer register. The output from the memory address register

is connected to the memory's address input. This is possible because the memory does not receive an address as input from another source, allowing the memory to receive the address of the location to be accessed straight from the memory address register and also removing the need for bus control circuits. The memory is connected to the system bus via two data paths.

When data is being sent out to the system bus during a read cycle, the memory control input R is asserted and data is sent to the system bus via tristate buffer G_{MSR}. During a write cycle the memory control input W is asserted and data is transferred into memory from the system bus. As figure 2.2 shows, all the other components of the CPU such as memory buffer register, program counter, etc. are each connected to the system bus in the same way as the memory. When a register needs to put data onto the system bus its tristate buffer is enabled. "Conversely data is copied into a register from the bus by clocking the register." [2]

The arithmetic and logic unit has two inputs; P which is from the D0 data register and Q which is from the bus. The arithmetic and logic unit contains its own internal register that is clocked by C_{ALU} allowing the output to be put on the system bus by enabling tristate gate G_{ALU}. "The arithmetic and logic unit is controlled by a two-bit code F_1 and F_0," [2] which determine the operation to be carried out.

Table 2.3 Decoding ALU control code [2]

F_1	F_0	Function
0	0	Add P to Q
0	1	Subtract Q
1	0	Increment Q
1	1	Decrement Q

Table 2.4 Instruction set for CPU in figure 2.2 [2]

Op-code	Mnemonic	Operation
000	`LOAD N`	[D0] ← [N]
001	`STORE N`	[N] ← [D0]
010	`ADD N`	[D0] ← [D0] + [N]
011	`SUB N`	[D0] ← [D0] – [N]
100	`INC N`	[N] ← [N] + 1
101	`DEC N`	[N] ← [N] – 1
110	`BRA N`	[PC] ← [N]
111	`BEQ N`	IF Z = 1 THEN [PC] ← [N]

Table 2.4 shows a list of very basic 3-bit instructions that the CPU can perform. We can now define each of the instructions in register transfer language. From these we can create a list of micro-operations for each instruction.

As an example of this we shall look at the implementation of the `LOAD N` instruction [D0] ← [N]. To start we need to express the instruction in register transfer language. Therefore [D0] ← [N] would be defined as [MAR] ← [IR], which means "copy target address N from IR to MAR" [3], and [D0] ← [[MAR]] which means "copy data at target address to data register." [3] Once this has been achieved we can see which control signals are needed to implement each step, which we just defined, in the case of the simple CPU in figure 2.2.

[MAR] ← [IR] $E_{IR} = 1, C_{MAR} = 1$

[D0] ← [[MAR]] $R = 1, E_{MSR} = 1, C_{D0} = 1$ [3]

As we can see, to send the operand address from the instruction register to the memory address register E_{IR} is enabled which activates the tristate gate G_{IR} allowing the operand address to be put on the system bus. Then this is clocked into the memory address register using C_{MAR}. Now R is used to read from memory. The data is read from memory to the system bus via the tristate gate G_{MSR} by enabling E_{MSR}. The data is then clocked into the data register D0 using clock C_{D0}.

Instruction	Operations (RTL)	R	W	C_{MAR}	C_{MBR}	C_{PC}	C_{IR}	C_{D0}	C_{ALU}	E_{MSR}	E_{MBR}	E_{IR}	E_{PC}	E_{D0}	E_{ALU}	F_1	F_0
LOAD	[MAR] ← [IR]	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0
	[D0] ← [[MAR]]	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0

Table 2.5 Display of micro-operations for LOAD instruction [2]

Table 2.5 displays which control actions are required to execute the LOAD N instruction in the form of two 16-bit binary strings, 0010000000100000 and 1000001010000000.

To execute an instruction firstly we need to decode the 3-bit op-code to determine which of the eight operations in table 2.4 is to be executed. This is done by using an instruction decoder circuit. The decoder works by setting one of the eight outputs to an active condition. Therefore if "the op-code corresponding to ADD (010) is loaded the ADD line from the AND gate array is asserted high" [2] while the other outputs will remain low. Once the control unit knows which instruction needs to be executed it now needs to carry out the microinstructions that correspond to the instruction. This is achieved by constructing a circuit called a sequencer that creates a source of signals to trigger each microinstruction. The timing pulse output by the sequencer can be seen in figure 2.3.

The next step is to "combine the signals from the instruction decoder with the timing signals from the sequencer to generate the actual control signals." [2] An example of this could be a similar structure to figure 2.4, in which nine vertical lines are used in the decoder (only two shown) where one line represents the fetch phase while the other eight lines represent the eight instructions that can be executed. "Only one of the vertical lines will be in a logical one state." [3] The timing signals from the sequencer are also input into the columns. "As the timing signals are generated the outputs of the AND gates enabled by the current instruction synthesize the control signals required to implement the random logic control unit." [2] Therefore the result of each AND gate actually triggers the microinstruction. The output of the AND gates is fed into OR gates that combine the outputs as a microinstruction may be activated by more than one control signal. Finally to implement a way of switching from fetch phase to execute phase we can use a RS flip-flop, which acts as a latch that can be set or reset.

Figure 2.3 Timing outputs from sequencer [3]

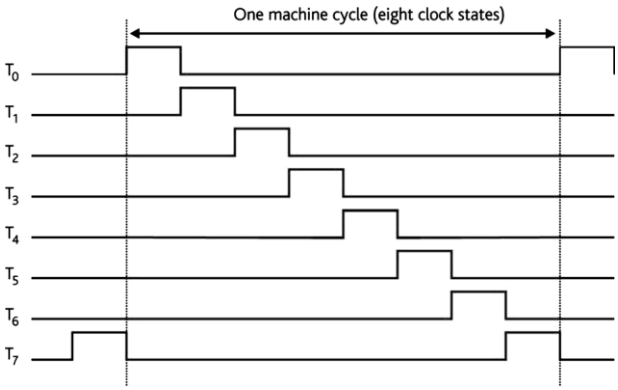
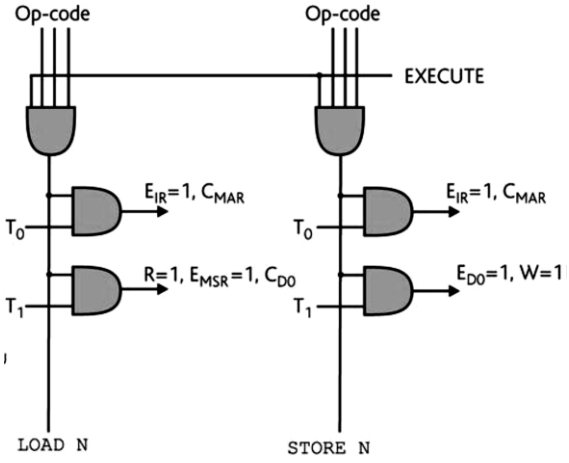


Figure 2.4 Circuit to combine control signals [3]



2.2 – Instruction Set Architecture

"An instruction set architecture (ISA) is an abstract model of a computer that describes what it does rather than how it does it." [6] It gives commands to the processor telling it how to function and comprises of addressing modes, native data types, instruction registers, memory architecture, interrupt and exception handling, and external I/O. "The instruction set architecture forms a boundary between the hardware and software." [6]

Figure 2.5 shows the different levels at which a computer can be viewed. In this section we will look at the machine level and micro-program levels.

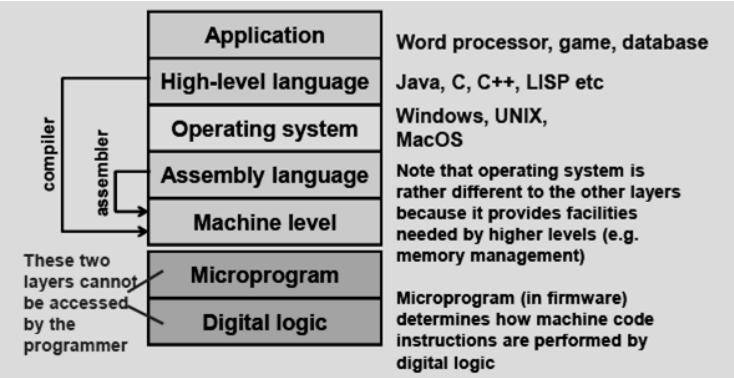


Figure 2.5 Machine levels and architectural layers [6]

In the previous section we looked at the structure of the CPU in detail. When designing a CPU the main goal is speed of execution. There are some influencing factors that affect this such as the "design of the instruction set, design of memory organisation and pipelining." [6] There are two types of design approaches for processors, complex instruction set computer (CISC) and reduced instruction set computer (RISC). CISC processors have large, irregular instruction sets. They can "perform operations directly on the contents of memory locations." [5] RISC processors can only perform operations on the contents of registers. The 'reduced' in reduced instruction set computer does not mean that the instruction set is smaller than CISC processors; it represents the one instruction per clock cycle that the RISC processor can achieve. The CISC processor may require many clock cycles per instruction. The RISC can achieve this because it uses pipelining to overlap consecutive instructions so the next instruction begins execution before the current instruction has been completed. "A simple relationship between the bit pattern of an instruction and what the instruction does" [5] is necessary for this.

2.2.1 – Instructions

Computer instructions are performed sequentially. This takes place unless an instruction deliberately changes the control flow or an exception occurs. Instruction structure varies depending on the machine so even if two instructions accomplish the same thing they will appear different on each machine.

Instructions are classified by what they accomplish and the number of operands they take. The three basic types of instruction are data movement, data processing and flow control. A data movement instruction "copies data from one location to another." [5] Data processing instructions operate on data and flow control instructions "alter the order in which instructions are executed." [6]

Instructions can take different numbers of operands three, two, one or zero. Three-address instructions such as `ADD P, Q, R` can be written in the form "operation source1, source2, destination" [5] where operation is the nature of the instruction being applied. Source1 and source2 are the locations in memory of the first and second operand respectively. The destination is memory location at which the result of the operation is saved. From the example `ADD P, Q, R` we can see that `ADD` is the operation that is being carried out on the contents of memory location `P` and memory location `Q`. The result of the addition is to be saved in memory location `R`. Theoretically `P`, `Q` and `R` could all be memory addresses containing 32 bits but this would make instructions too long as $3 \times 32 = 96$ and with an assumed 16-bit operation code the total instruction would be 112 bits. The history of computer technology means that this is only theoretical because early computers were developed when memory was expensive and implementing a 112-bit instruction would not be cost effective. Therefore "a modern RISC processor allows you to specify three addresses in an instruction by providing three 5-

bit operand address fields." [5] This only allows you to select one of 32 different operands located in registers within the CPU. As on-chip registers are being used to hold the operands the time taken to access data is fast because registers are the fastest storage devices to access. "An instruction with three 32-bit operands requires 3 × 5 bits to specify the operands" [5] this allows an instruction of 32 bits to use the remaining 17 bits to specify the operation.

Two-address machines use two operands in their instructions. An example of a two operand instruction is `ADD P, Q` this can be interpreted as add the contents of memory location `P` to the contents of memory location `Q` and store the result of the addition in memory location `Q`. Using only two operands requires less storage and memory traffic, however one of the source operands is overwritten during the execution of the instruction. "Most computer instructions can't directly access two memory locations" [5] therefore the operands are usually either two registers or a register and a memory location.

One-address machines use only one operand in their instructions. "The second operand is a fixed register called an accumulator" [5], the accumulator does not need to be specified in the instruction itself. For example `ADD P` means add the contents of the memory location `P` to the contents of the accumulator and store the result in the accumulator. "Eight-bit machines such as Intel 8080 and Motorola 6800 use one-address architecture." [5] Using one-address architecture is a long-winded solution as data needs to be loaded into the accumulator, processed and the contents of the accumulator stored into another memory location to stop it being overwritten in the next cycle. An example 8-bit code fragment which implements the operation R = P + Q on the Motorola 6800 processor illustrates this point.

```
LDA P    ; load accumulator with P
ADD Q    ; add Q to accumulator
STA R    ; store accumulator in R
```
[5]

One-address machines are still used in applications such as toys because of their low-cost and low-performance nature.

Zero-address machines use stacks for all computation. Therefore no operand locations need to be specified. An example of a zero-address instruction is `ADD` which pops the top two items off the stack performs the addition and pushes the result onto the top of the stack. A result needs to be popped off the stack in order to store it in memory. "Stack machines are largely experimental with one exception the Java Virtual Machine (JVM)." [6]

2.2.2 – Addressing modes

The concept of addressing modes is important; we need to specify where the data, the computer performs operations on, comes from. The different variations of expressing the source of operands are called addressing modes.

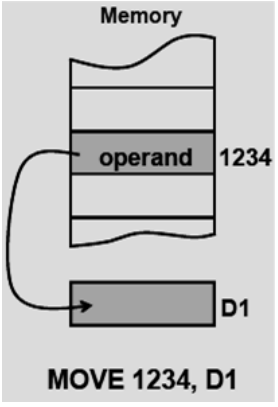


Figure 2.6 Example of absolute addressing [6]

Absolute addressing is where an operand is a location in memory or a register. For example in the instruction `ADD P, D0` absolute addressing is used because the "location of operand `P` is specified as a memory location." [5] Absolute addressing also includes specifying a register as an operand; this is sometimes referred to as register direct addressing.

Immediate addressing is where an operand is data itself rather than a memory location. An example of this is the instruction `ADD #4, D0` which in 68K means add the literal 4 to the contents of data register `D0` and store the result of the addition in `D0`. "Immediate addressing lets you specify a constant, rather than a variable." [5] The term 'immediate' is used because the operand does not have to be fetched from memory it is immediately available from the instruction.

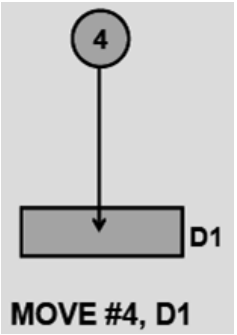
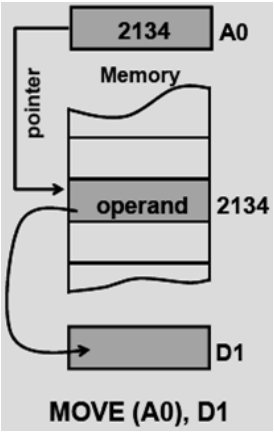


Figure 2.7 Example of immediate addressing [6]



Indirect addressing specifies a pointer to the actual operand. The instruction `MOVE (A0), D1` is an example of this. The contents of the `A0` register are read to find the address in memory of the operand to be used in the instruction. The operand is read at the address and is copied into the `D1` register. "This addressing mode requires three memory accesses" [5], first to read the instruction, second to read the register to obtain the address of the actual data and third to fetch the actual data from the location given by the pointer.

Figure 2.8 Example of indirect addressing [6]

2.3 – Logisim

We now need to examine the software that will be used to create the CPU. The software "Logisim is an educational tool for designing and simulating digital logic circuits." [7] Its simple nature means it is a useful tool to introduce students to the idea of logical circuit design and also allows them to easily experiment with their own circuits.

Logisim has a wide variety of useful features to allow users to easily design circuits. For example it offers a wide range of pre-built components such as arithmetic circuits, flip-flops and memory as well as all of the basic elementary logic gates that would be expected in circuit design software. This allows users to experiment with more complex circuits without designing every component themselves. Wires can be easily drawn by the user between these components. Logisim also offers a number of input and output components that allow users to test if their circuits work correctly. "Circuits can be saved into a file, exported as a GIF file or printed out from a printer," [7] which allows circuits to be easily transported between computers. The software allows circuits to be used as 'sub-circuits' of other circuits which means a neat hierarchical circuit design can be established. This is particularly useful for this project as each component of the CPU, such as the arithmetic and logical unit and the program counter, can be designed as sub-circuits allowing the top-level circuit to appear uncluttered making the circuit easier to design and understand. Logisim is free software and can "run on any machine supporting Java 5 or later." [7]

There are many other circuit simulation software packages available that are similar to Logisim such as LogicSim 3.0, Digital Works 2.0 and Multimedia Logic. We shall now discuss why Logisim has been chosen over these other programs. Firstly LogicSim 3.0 and Multimedia Logic do not have variable-input gates meaning they do not incorporate "gates that can accept more than two inputs." [8]

Furthermore Digital Works 2.0 and Multimedia Logic do not offer an indication of the value of the wires in the circuit whereas Logisim displays “a wire carrying low voltage with one colour and a wire carrying a high voltage with another colour.” [8] The main reason Logisim is more useful than other software of similar nature is because it is not limited to specific platforms whereas the other programs discussed above only run on Windows or Macintosh. Since Logisim does not have this limitation it means it is useful for students using different platforms in a laboratory class. This along with its ease of use is the reason why it is used in the Com1006 module. The CPU will be designed in Logisim as Com1006 students will be able to relate more closely to the project because of their previous experience with the software.

Chapter 3: Requirements and Analysis

As stated in the previous chapters the goal of the project is to create a CPU in Logisim which can be used as an educational tool to inspire future cohorts of Com1006 students. With this in mind a set of specific requirements for the project need to be clearly defined. For this section it is appropriate to define high level software and design requirements and then after, a more in depth list of requirements for operations that the CPU should be able to execute. These requirements are summarised in table form, where they have been kept brief, with an indication of their priority. They are then elaborated on and discussed in the Analysis of Requirements, section 3.2.

3.1 – Requirements

Priority key:

M – Mandatory

O – Optional

3.1.1 – Software and Design Requirements

Requirement	Description	Priority
Software	The CPU should be designed using the circuit design software Logisim	M
Complexity	The majority of the CPU components should be made out of elementary logic gates such as AND, OR, NOT and XOR.	M
Testing	A way of testing if the CPU is working correctly needs to be established	M
Memory	The CPU should contain a form of basic memory	M
Registers	The CPU should contain a number of data registers to store values	M
Addressing modes	The CPU should support at least two different addressing modes	M

3.1.2 – Arithmetic and Logical Operations Requirements

Requirement	Description	Priority
Addition operation	Can perform the arithmetic addition operation on two eight bit words	M
Subtraction operation	Can perform the arithmetic subtraction operation on two eight bit words	M
Multiplication operation	Can perform the arithmetic multiplication operation on two eight bit words	M
Division operation	Can perform the arithmetic division operation on two eight bit words	O
AND operation	Can perform the logical AND operation on two eight bit words	M
OR operation	Can perform the logical OR operation on two eight bit words	M
NOT operation	Can perform the logical NOT operation on an eight bit word	M
XOR operation	Can perform the logical XOR operation on two eight bit words	M
Shift right operation	Can perform the logical shift right operation on an eight bit word	O
Shift left operation	Can perform the logical shift left operation on an eight bit word	O
Branch instructions	The CPU should be able to execute conditional and unconditional branch instructions	M

3.2 – Analysis of Requirements

This section elaborates and analyses the requirements given in the previous section. Discussing the requirements in more detail and explaining how each of the requirements can be met.

3.2.1 – Software and Design Requirements Analysis

Software – As stated throughout the paper, the software that shall be used is Logisim. It is important to use Logisim as it is used by Com1006 students and will allow them to relate to the project.

Complexity – Logisim offers many pre-made components such as adders, dividers and many other components that could be used in the project. However one of the aims of the project is to create the CPU out of the elementary gates such as AND, OR, NOT and XOR. This means that the components should be made of the elementary gates instead of using Logisim's pre-made components. This may require some flexibility and will be discussed later in the design chapter.

Testing – To test if the CPU is working a form of user-interface will need to be created as part of the circuit. Logisim offers a screen component that can be written on to which could show the output of the CPU allowing users to test if this output is correct. Logisim also has a keyboard component that could be used to input test data into the CPU.

Memory – A basic memory should be implemented inside the CPU. Logisim does have memory components such as RAM but the memory should be built from elementary gates. It will need to be designed carefully because it will only have limited storage space.

Registers – A number of registers will be needed to store values while carrying out instructions. Logisim has pre-built registers which should not be used. Instead the registers could be made out of flip-flops. Logisim also provides pre-built flip-flops and a decision about whether to use these or create flip-flop circuits from elementary gates needs to be made. The number of registers also needs to be decided. A CPU with more registers would be more powerful however the more registers the CPU contains the more bits in the instruction will be needed to specify them. Circuitry needs to be designed to select which register data is being read from or written to while an instruction is being executed. The more registers the CPU contains the more complex this circuitry will be.

Addressing modes – The CPU should implement at least two addressing modes. There are three types of addressing modes therefore one can be an optional requirement. Absolute and immediate addressing are the two that should be implemented and indirect addressing is an optional requirement. To accommodate immediate addressing the instruction should be sufficiently long so that a literal value can be encoded into it when necessary. A method of choosing between addressing modes needs to be designed, this could be a simple bit designated for addressing mode selection or could be a special register code that only activates immediate addressing when necessary. This leads to the design of the CPU's instructions as their architecture will affect the design of the CPU itself.

The length of the instruction matters. The longer the instruction the more data can be encoded into the instruction, allowing for more instructions to be carried out and more addresses to be used. The number of bits designated to the op-code needs to be defined and each instruction needs to be encoded into the op-code. The number of bits used for source and destination registers needs to be considered, as the more bits used for this the more registers can be included. However too many may mean that the instruction cannot encode as much other information such as literals or designated addressing mode bits. Instructions need to be easy to follow and understand therefore should not be too long.

3.2.2 – Arithmetic and Logical Operations Requirements Analysis

Addition operation – The CPU should have an ALU that can carry out the arithmetic addition operation on two eight bit words. This can be done by implementing an adder into the ALU made out of elementary logic gates. Logisim has a pre-built adder component but this should not be used.

Subtraction operation – The CPU should have an ALU that can carry out the arithmetic subtraction operation on two eight bit words. This can be done by implementing a subtractor into the ALU made from elementary logic gates. Logisim has a pre-built subtractor component which should not be used.

Multiplication operation – The CPU should have an ALU that can carry out the arithmetic multiplication operation on two eight bit words. This can be done by implementing a multiplier into the ALU made from elementary logic gates. Logisim has a pre-built multiplier component which should not be used.

Division operation – The CPU should have an ALU that can carry out the arithmetic division operation on two eight bit words. This can be done by implementing a divider into the ALU made from elementary logic gates. As this feature is optional it may not be implemented, a compromise could be made where the Logisim pre-built component is used instead of designing a divider out of elementary gates which is a time consuming and complex task.

AND operation – The CPU should have an ALU capable of carrying out the logical AND operation on two eight bit words. This can be achieved by simply implementing an AND gate into the ALU which has two eight bit words as input.

OR operation - The CPU should have an ALU capable of carrying out the logical OR operation on two eight bit words. This can be achieved by simply implementing an OR gate into the ALU which has two eight bit words as input.

NOT operation - The CPU should have an ALU capable of carrying out the logical NOT operation on an eight bit word. This can be achieved by simply implementing a NOT gate into the ALU which has an eight bit word as input.

XOR operation – The CPU should have an ALU capable of carrying out the logical XOR operation on two eight bit words. This can be achieved by simply implementing an XOR gate into the ALU which has two eight bit words as input. This could also be implemented as an XOR circuit using AND, NOT and OR gates however doing this is unnecessary as it will clutter the ALU circuit; it is easier to use Logisim's pre-built XOR gate.

Shift right operation – The CPU should have an ALU capable of carrying out the logical shift right operation on an eight bit word. This can be achieved by creating a custom shift right component inside the ALU. This requirement is optional so it may not be implemented. Logisim has pre-built shift right component that could be used as a compromise instead of designing a custom component from elementary gates.

Shift left operation – The CPU should have an ALU capable of carrying out the logical shift left operation on an eight bit word. This can be achieved by creating a custom shift left component inside the ALU. This requirement is optional so it may not be implemented. Logisim has pre-built shift left component that could be used as a compromise instead of designing a custom component from elementary gates.

Branch instructions – The CPU should be capable of implementing a number of conditional and unconditional branch instructions. These should be encoded in the op-code which should be designed with sufficient space. The conditional branch instructions that should be implemented are; branch if

equal, not equal, carry set, carry clear, higher than and lower than or same. The unconditional branch instructions jump and trap should also be implemented.

3.3 – Evaluation and Testing

It is important to discuss how the success of the project can be evaluated in the early stages. There are some key criteria that will be used to evaluate the success of the project.

Firstly how well the project meets the requirements defined in the requirements table. The project should implement all mandatory requirements in order to be successful. Theoretically, in meeting the requirements, the CPU being designed should be Turing complete meaning it can implement any computer program which will be a great achievement in itself.

However meeting the requirements is the minimum for the evaluation of the project to be considered successful. Implementing some of the optional requirements will show ambition and will add new levels of complexity to the project. Optional increases in CPU efficiency using methods such as pipelining could also improve the CPU. In meeting the requirements the CPU needs to be made from elementary logic gates. It is important to keep this in mind when evaluating the CPU at the end of the project. If the majority of the CPU is made out of elementary gates it will be a far more impressive feat than using prebuilt Logisim modules.

Another method for evaluating the project will be the results of the tests carried out during the testing phase of implementation. A test table will be created in the design phase of the project which will list a number of tests that the CPU should pass. Tests will comprise of programs that should be input into the CPU and executed to create some output which can be evaluated. If the expected result in the test table is the same as the actual result from carrying out the test then the project can be deemed successful. If however results from testing are unexpected the CPU design may need to be altered in order to function correctly.

Finally and arguably the most important is meeting the purpose of project; to inspire future cohorts of Com1006 students. This criterion differs from the above because it is difficult to gauge the impact the project will have as an educational tool due to the subjective nature of student's perception. A video could be created and uploaded to YouTube that showcases the CPU and explains how it works. Feedback from this video could be used to help evaluate the project.

Chapter 4: Design

Now that we have examined in detail what components the CPU is comprised of and how it can be implemented in hardware. We shall now look at the proposed design of the CPU for this project. We shall start by creating an appropriate instruction set architecture that will determine how the CPU will be built.

4.1 – Instruction Set Design

The formats of the CPU's instructions are explained in figure 4.1. The instruction itself has a 16-bit length. The first 4 bits represent the operation code that is to be carried out by that specific instruction.

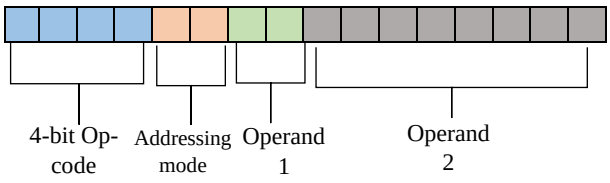


Figure 4.1 Format of instruction

4.1.1 – Op-code

A 4-bit op-code allows for a total number of 16 (2^4) instructions. The following table shows a list of the 16 operations the CPU is capable of executing, based on the Motorola 68k architecture. [9]

Op-code	Mnemonic	Operation
0000	MOVE	Move contents of one data register to another
0001	ADD	Perform addition on two operands
0010	SUB	Perform subtraction on two operands
0011	MULU	Perform multiplication on two operands
0100	AND	Perform AND on two operands
0101	OR	Perform OR on two operands
0110	NOT	Perform NOT on an operand
0111	XOR	Perform addition on two operands
1000	BEQ	Branch on equal (Zero = 1)
1001	BNE	Branch on not equal (Zero = 0)
1010	BHI	Branch on higher than (Carry = 0 and Zero = 0)
1011	BCS/BLO	Branch on carry set/ lower (Carry = 1)
1100	BCC/BHS	Branch on carry clear/ higher than or same (Carry = 0)
1101	BLS	Branch on lower than or same (Carry = 1 or Zero = 1)
1110	JUMP	Jump to instruction specified
1111	TRAP	Used to perform operating system calls

Table 4.1 Instruction set

Of note are the conditional branch instructions. The number representation used in the CPU is unsigned values, meaning if a number is represented by 8 bits it is in the range of 0-255. The branch instructions reflect this representation; there are only two status flags, carry and zero. If a signed representation was used we would have different branches that would be affected by the status of overflow, negative and zero flags. In addition to this, readers may notice that there is no compare operation (`CMP` in Motorola 68k) in order to see if a branch condition is met. For example in 68k the compare operation would be used on two numbers in order to set the status flags and then execute one of the branches based on the outcome, such as branch on higher (`BHI`) if the first operand is higher. The compare operation is not needed in the instruction set because it is the same as subtraction; the only difference being during compare the result is not saved. [10] Therefore the instruction set design can accommodate conditional branch instructions effectively.

4.1.2 – Addressing Mode

The next two bits in the instruction represent the addressing mode being used in the instruction. This could be absolute where a register or memory address is used as an operand or immediate where a literal is used in the instruction. Not only do these bits indicate the addressing mode but they also specify the order of operands being used. For example they distinguish between memory-to-register where instructions and register-to-memory instructions, this is shown in table 4.2.

Addressing mode	Encoded addressing sequence
00	Register to register
01	Immediate to register
10	Memory to register
11	Register to memory

Table 4.2 Addressing mode set

4.1.3 – Operand 1 and 2

Bits 8 and 9 in the instruction are the first operand to be used in the instruction. Two bits are used because operand 1 is always a register and the four registers are encoded using two bits as expected. The remaining eight bits are used to encode operand 2 which can be a register, memory address or immediate value; this will be defined by the addressing mode.

Usually in a two address machine the result of the operation would be stored in the location specified by the second operand in the instruction; this not always the case with this architecture because of the way addressing modes are encoded. In each instruction a register is used for at least one of the operands, so operand 1 is always a register. Where we have an instruction which uses a value from memory and a register value with the result being stored in the register, we cannot encode the memory address as operand 1 because it is used to encode the register in each instruction. Therefore the addressing mode (in this case 10) will encode which ways round the operands are being used.

4.1.4 – Example Test Programs

Now that we have an understanding of the instruction format we shall now look at some programs written in machine language which the CPU should be capable of executing. These programs will be used during the testing phase of the project in order to make sure the CPU functions correctly.

Instruction description	Instruction in assembly code	Instruction in machine code
Move the literal value 4 into data register r0	`MOVE #4, r0`	0000 01 00 00000100
Move the literal value 6 into data register r1	`MOVE #6, r1`	0000 01 01 00000110
Perform addition operation on contents of register r0 and r1, storing the result in r1	`ADD r0, r1`	0001 00 00 00000001
Use trap command to display result	`TRAP`	

Table 4.3 Test program to computer the summation of 4 and 6

A more complex program could be calculating the n^{th} number in the Fibonacci sequence.

Instruction description	Instruction in assembly code	Instruction in machine code
Move literal value 0 into data register r0	`MOVE #0, r0`	0000 01 00 00000000
Move literal value 1 into data register r1	`MOVE #1, r1`	0000 01 01 00000001
Move literal value n into data register r3	`MOVE #n, r3`	0000 01 11 n
Loop start – Move contents of r0 into r2	`MOVE r0, r2`	0000 00 00 00000010
Perform addition operation on contents of register r1 and r2, storing result in r2	`ADD r1, r2`	0001 00 01 00000010
Move contents of r1 to r0	`MOVE r1, r0`	0000 00 01 00000000
Move contents of r2 to r1	`MOVE r2, r1`	0000 00 10 00000001
Subtract the literal 1 from the value of r3 (loop counter) and store result in r3	`SUB #1,r3`	0010 01 11 00000001
Use branch on not equal operation to repeat loop until counter (r3) is 0	`BNE Loop start`	1001 00 00 Loop start
Use trap command to display result	`TRAP`	

Table 4.4 Test program to calculate nth Fibonacci number

Table 4.5 shows another test program which calculates the factorial of a given integer n.

Instruction description	Instruction in assembly code	Instruction in machine code
Move literal value n into data register r0	`MOVE #n, r0`	`0000 01 00 n`
Move literal value n into data register r1	`MOVE #n, r1`	`0000 01 01 n`
Subtract literal 1 from contents of r0 (n) storing result in r0	`SUB #1,r0`	`0010 01 00 00000001`
Loop start – Multiply the contents of r0 and r1 storing the result in r1	`MULU r0,r1`	`0011 00 00 00000001`
Subtract 1 from the contents of r0 storing result in r0	`SUB #1,r0`	`0010 01 00 00000001`
Use branch on not equal operation to repeat loop unit contents of r0 is 0	`BNE Loop start`	`1001 00 00 Loop start`
Use trap command to display result	`TRAP`	

Table 4.5 Test program to calculate n factorial

4.2 – CPU Design

The instruction set architecture has been established and it is now time to outline the design of the CPU itself. Following the structure of chapter 2 we shall examine the design of each individual component of the CPU and then construct a method for interaction between each component to create the overall CPU design.

4.2.1 – ALU Design

We shall start by looking at the design of the ALU. As we know from chapter 2 this component performs all of the CPU's calculations. In principle this seems like one of the more complex pieces but is actually relatively simple.

The ALU takes one or two 8-bit words and performs calculations based on the op-code provided in the instruction. From the table 4.1 in the previous section we can see that all arithmetic and logic instructions contain a 0 as their first bit. This allows us to ignore the first bit and choose the operation required using the remaining three bits.

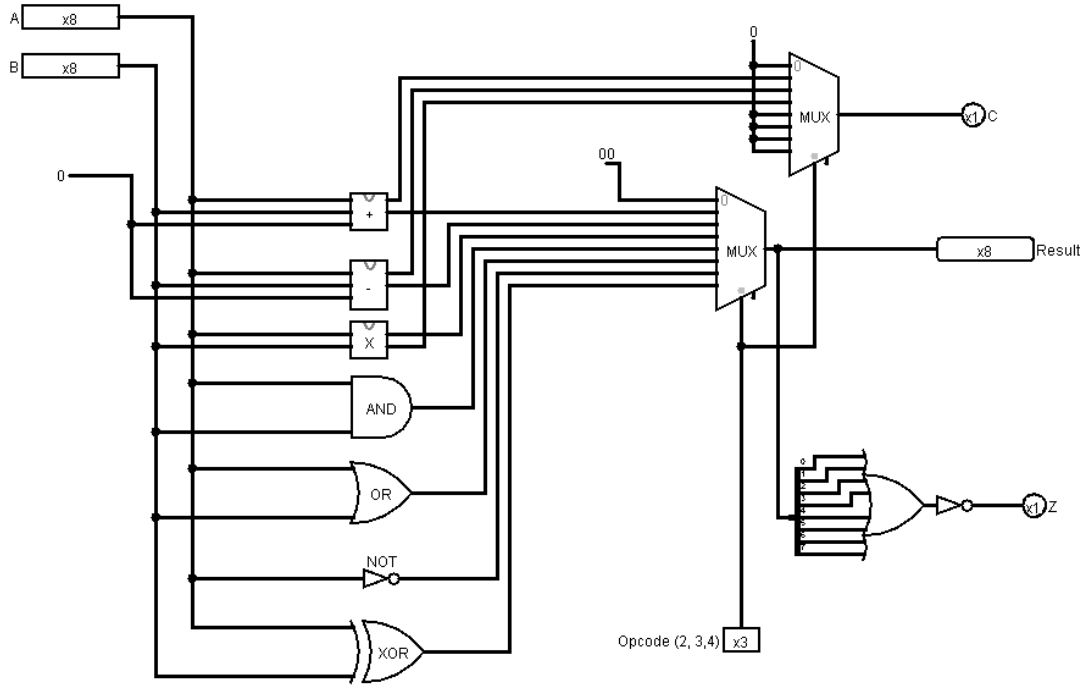


Figure 4.2 Design of ALU

Figure 4.2 shows a proposed design for the ALU. As you can see it contains all of the operations outlined in the requirements that the ALU should be capable of performing. The two 8-bit words are input into the circuit and each operation is performed. The output, Result, is then determined by the Opcode input which is used as the selection bits for the multiplexer. Note that the 0 input for the multiplexer is not a valid arithmetic or logical operation (see instruction set from table 4.1) so therefore the constant 0 is being used to stop the circuit from causing an error.

As well as the result from the operation the ALU has two other outputs. These are the status flags Carry out and Zero which allow conditional instructions to be executed. The carry flag is simply the carry out from the addition, subtraction and multiplication operations. A multiplexer is used to determine which is being used again taking the op-code as the selection bits. The multiplexer only chooses between three inputs so we could have used the middle part of the op-code to determine if addition, subtraction or multiplication is used as their op-codes are 001, 010 and 011 respectively. However this will not work in practice because other op-codes have the same middle bits which will

mean the carry out flag will be set for operations that are not only arithmetic. The solution is to have a similar multiplexer to the operation selection and use the carry out from the arithmetic components as input bit 1, 2 and 3. The rest of the inputs are zero constants.

The zero flag is calculated by computing the logical OR of all the result bits and negating the result. A bit splitter is used to separate the individual result bits, they are each input into an OR gate. “If all bits are zero, the OR is zero and the inversion is true.” [11]

The ALU as a whole has been designed but this project requires all components to be made from elementary gates. This means the pre-built Logisim components that perform the calculations such as the adder, subtractor and multiplier need to be designed individually from elementary gates.

4.2.2 – Adder

To create an adder we take two 8-bit word inputs and a 1-bit carry input. Perform addition of each bit of each word in turn with the carry output from each operation being used as the input carry for the next bit addition. This is known as a ripple carry adder as the carry ripples across the circuit.

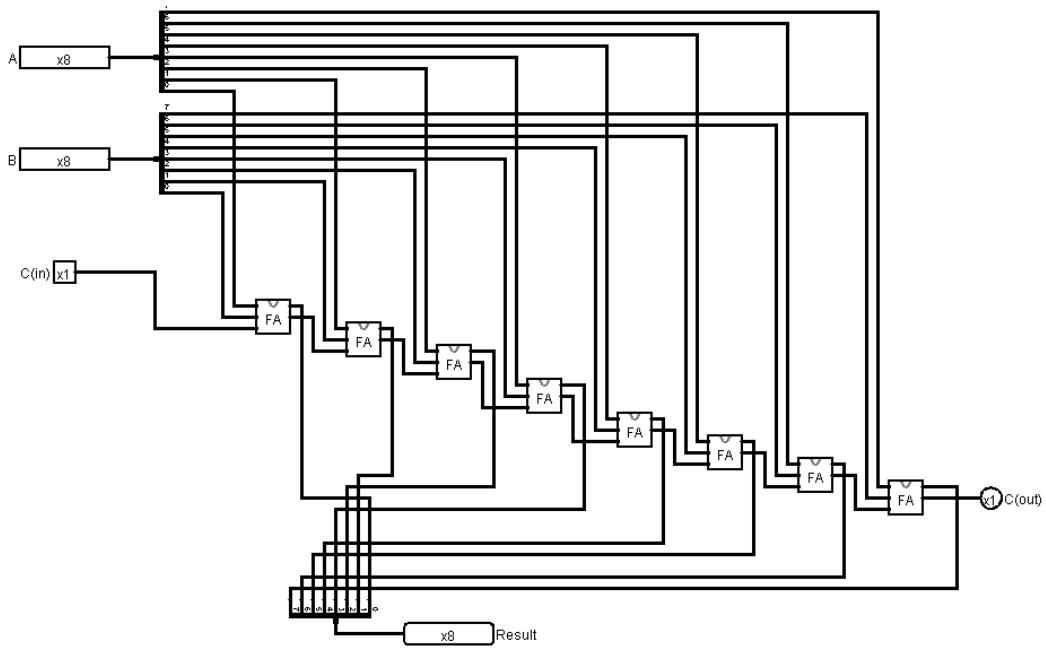


Figure 4.3 8-bit adder circuit

As we can see from figure 4.3 the two 8-bit inputs A and B are split and individual bits are summed together using full adders (FA). These allow the addition of two bits and a carry-in bit with the summation and carry-out as output. To understand how this circuit works we need to look at how full adders are constructed.

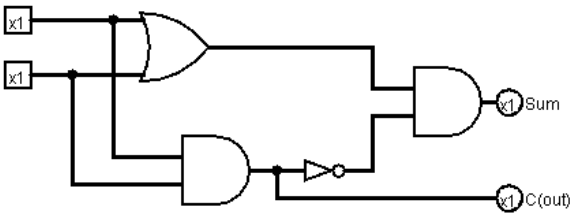


Figure 4.4 Half adder circuit

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

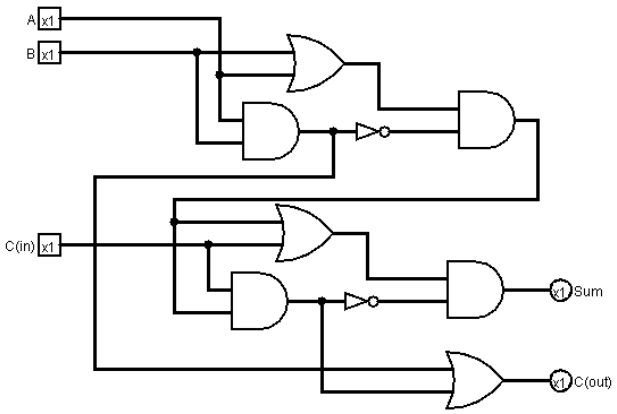


Figure 4.5 Full adder circuit and truth table

The half adder, seen in figure 4.4, is a basic circuit that allows the addition of two bits generating a summation and a carry out. It is not very useful because it does not include the carry input bit we need. However two of these can be combined to create a full adder (figure 4.5). This essentially uses "the first half adder to add the input bits A and B and then uses the second half adder to add the carry in bit to the result" [12]; generating a summation and a carry out.

We can include eight full adders into the 8-bit adder circuit to add each individual bit of the two words and use the carry throughout the circuit. In the end the summations are combined back into an 8-bit result and the final carry out from the last full adder is used as the carry out for the operation.

Ripple carry adders have a large circuit depth. This could lead to slow performance. If the circuit performs operations too slowly then the circuit may need to be redesigned using a carry look-ahead adder which will be more complex but will have faster performance.

4.2.3 – Subtractor

The subtractor circuit works in a similar way to the adder. Two 8-bit words are split into individual bits, subtracted from one another with a carry in rippling through each full subtractor circuit.

A	B	C_{in}	Result	C_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

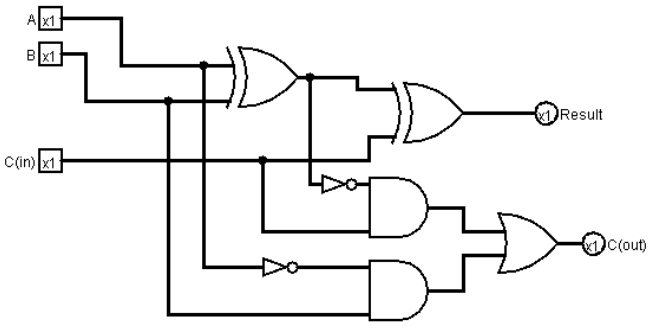


Figure 4.6 Full subtractor circuit with truth table

As with the full adder circuit the full subtractor combines two half subtractors made from XOR, NOT and AND gates. The first half subtractor is used to subtract the two inputs B from A and the second subtracts the carry from the result.

Eight of these circuits can be combined to create an eight bit subtractor. The results from each full subtractor are combined to create the final 8-bit result and the carry out from the last full subtractor circuit is used as the final carry out bit.

4.2.4 – Multiplier

The multiplication sub-circuit outputs the result of the multiplication of two 8-bit words. It works using the Wallace tree algorithm [13] in which the partial product bits are calculated by taking the AND of each bit in the two words separately. The results of the partial products are added using full and half adder sub-circuits to create the final result. The design of figure 4.7 was adapted from a design posted on planetminecraft.com. [14]

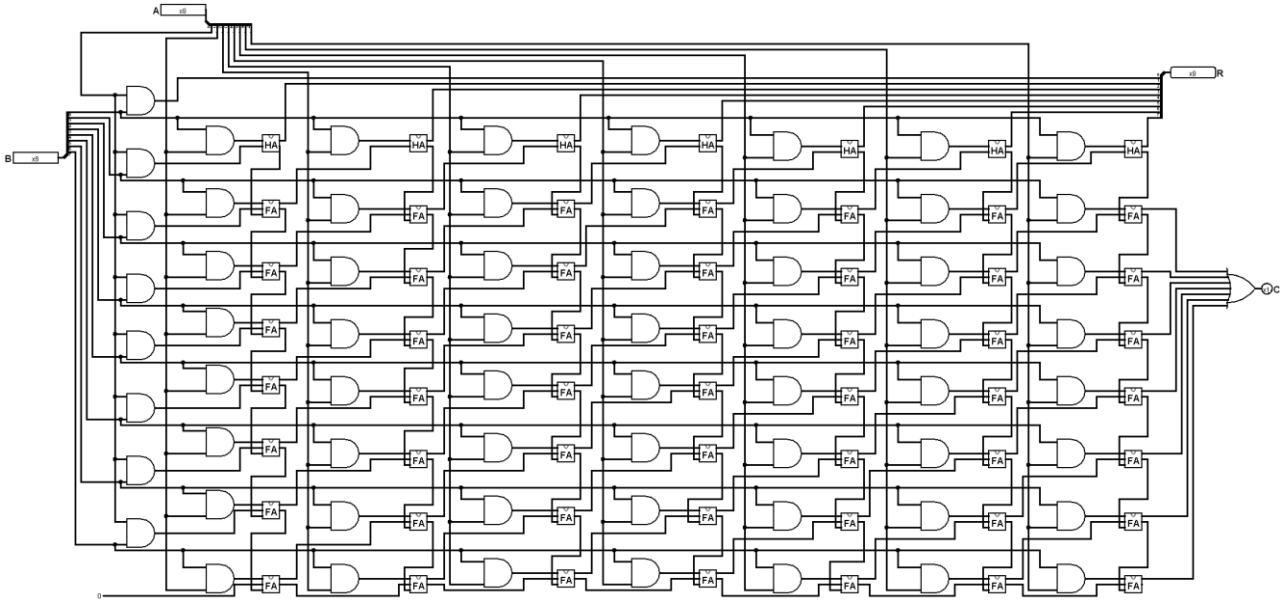


Figure 4.7 An 8-bit multiplier design

4.2.5 – Logical Operations

The logical operations are very simple to design as they are simply logical gates that Logisim provides. The AND, OR and XOR logical operations take two 8-bit inputs A and B and perform the corresponding logical operation on them. The NOT operation, due to its nature, only takes one input. They each run into the selection multiplexer and are selected by the op-code in the same way as the arithmetic operations. Note that these gates are in elementary gates apart from the XOR gate. This could be broken down into a sub-circuit containing only elementary gates however this is a trivial matter as readers will be familiar with how the XOR operation works.

4.2.6 – Registers

A fundamental part of the CPU is storing information in registers. There are several types of register for example data registers to hold data during a calculation, address registers to hold memory addresses and special registers such as the program counter which holds the location in memory of the next instruction to be executed. Despite there being different registers to store different values they are all fundamentally the same in design, just the number of bits stored may vary.

To understand how to design a register we need to look at how to store an individual bit. The answer is to use a D flip-flop. Figure 4.8 shows the basic circuit for a D flip-flop.

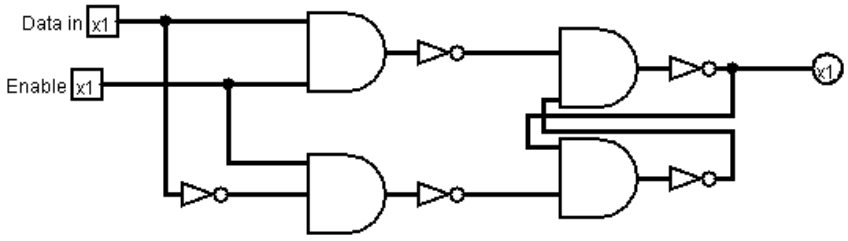


Figure 4.8 D flip-flop

The D flip-flop takes an input bit and a clock signal (write enable). When the clock signal is enabled the input bit will be written into the flip-flop. "It records the state of the D input and holds it constant until it's clocked." [15] Therefore the flip-flop will not override the value being stored until another clock pulse is sent.

From the D flip-flop it is relatively simple to see how a register can be formed. To store a word we create a circuit with multiple D flip-flops to store each bit in the word separately. These flip-flops will all use the same clock signal so the register value will remain constant until the next clock signal is input.

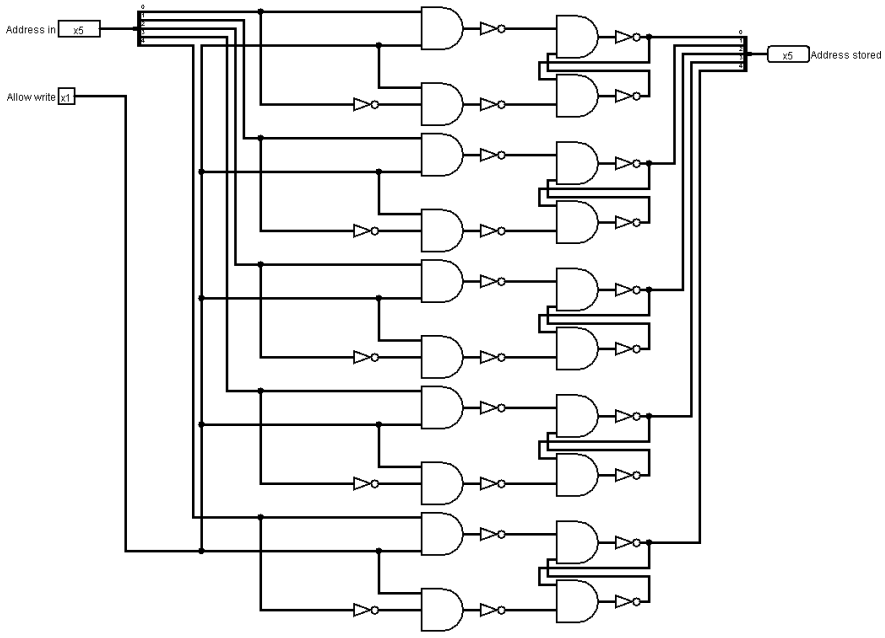


Figure 4.9 5-bit Register

As you can see, from the 5-bit register in figure 4.8, the input is split into individual bits and these are each stored in five D flip-flops. Each of the flip-flops is clocked using the same enable bit meaning that each bit is clocked into its flip-flop at the same time; therefore the whole word is clocked into the register in one clock pulse. The bits from the flip-flops are combined to create the output which is the word currently being stored in the register.

Figure 4.8 is a 5-bit register which could store a memory address (depending on size of memory) but any number of flip-flops can be combined in this way to create registers capable of storing different length words.

For this project we will need multiple registers which can store different word lengths. As defined in the instruction architecture section two bits will be designated to encode data registers which can be

used to store intermediate data values when carrying out operations. Since two bits are being used the CPU will have four data registers. Each should be capable of storing 8-bit word lengths because the ALU has been designed to only input and output 8-bit values.

The CPU will also need a memory address register, instruction register and program counter. The memory address register is used to store the memory address that is to be looked up in memory when running a program. This means its word length is determined by the number of addresses in memory. If the memory is able to store 64 words then the memory address register should be 6-bit. This allows all 64 addresses to be encoded.

The instruction register is used to store instructions, once fetched from memory, so that they can be decoded. This means for our CPU the instruction register should be 16-bit because the instructions that the CPU will be executing have a 16-bit length.

The program counter is a little more complex than a standard register. As we know from chapter 2 the program counter stores the memory address of the next instruction to be executed. This means the contents need to be incremented after each instruction has been fetched from memory. This could be done by using the ALU to add a constant to the address after the fetch phase which is simple because the ALU has already been designed and is capable of this but will require multiple clock cycles to perform. An alternative to this is to include an incrementor component inside the program counter circuit itself that will perform the addition without the address having to pass through the ALU. This should save several clock cycles because the address from the program counter will not have to be put onto a bus, go through the ALU and be clocked back into the program counter.

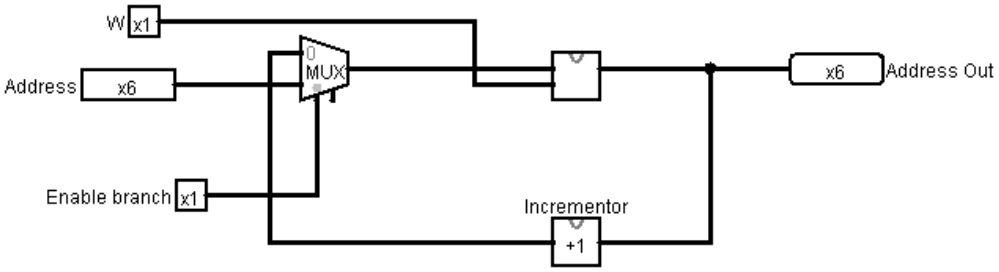


Figure 4.10 Program counter with incrementor

The program counter circuit is made up of a 6-bit register which will hold the address linked to an incrementor circuit which will increase the contents of the register by one. Other designs may increment by different values because the instructions in memory may span over multiple addresses due to their length. In this design each 16-bit memory line holds a full instruction therefore incrementing by one gives the memory address of the next instruction.

The incrementor is an adapted version of the adder from the ALU. Instead of two inputs A and B it has input A and adds the constant one to the value of A. To use the incrementor the register of the program counter is clocked by the input W.

Due to the need to accommodate branch instructions the program counter also needs to be capable of jumping to any instruction in memory. To include this feature two other inputs are included in the program counter. The address field specifies which address is being branched to and the enable branch bit is used to toggle between the incrementor path being taken and a branch being taken. The branch address is clocked into the register in the same way as if the address was going through the incrementor.

4.2.7 – Memory

The memory unit is a fundamental part of the CPU. This is the circuit which stores all of the programs that the CPU will execute. In essence it is a large register, meaning it follows a similar structure and is made from a combined number of D flip-flops. Since instructions being stored in memory are 16-bit, each memory address needs to be able to store 16 bits. We know how to store a bit using a D flip-flop so combining 16 of these to store an instruction is simple.

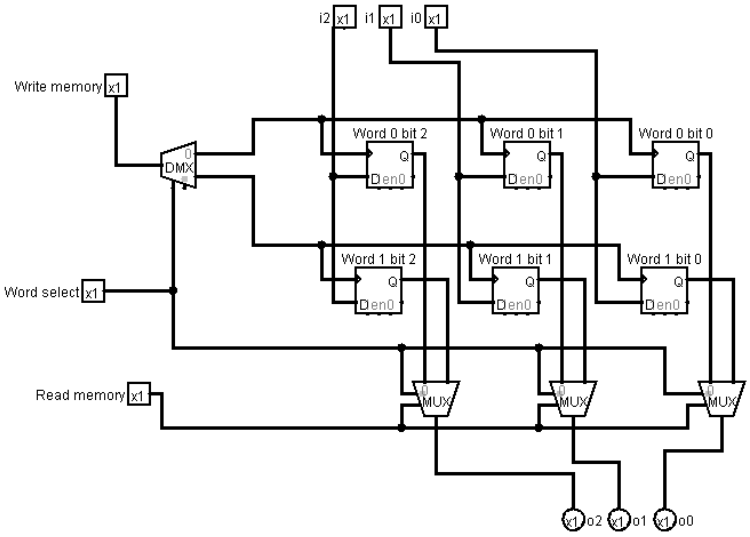


Figure 4.11 Basic memory circuit [16]

Figure 4.11 depicts a typical structure for a memory circuit. A series of D flip-flops are used to store a word, in this case only a 3-bit word. This memory unit is capable of storing two 3-bit words. A write memory bit is used to clock new data into a word through a demultiplexer. The word select bit is used as the selection bit for the demultiplexer. This allows data to be written into only one word per clock cycle. The read memory bit is used to enable and disable the series of multiplexers which read the output of the circuit. The same word select bit as before is used to select which word wants to be read from memory and the read memory bit allows the read to be output. This is a memory unit on a very small scale; the storage capacity for the project needs to be much larger in order to store multiple programs.

The proposed design contains 64 address locations allowing for a maximum of 64 instructions to be stored at one time in memory. This means addresses will need to be 6-bit to encode all 64 locations. The input for the circuit will be a memory address, 16-bit data-in, a read enable bit and a write enable bit. The memory address input will need to be decoded using a series of decoder circuits or demultiplexers. The output from the decoded address will be used to enable one of the address lines. In order to write to an address line the data must be clocked into the flip-flops using the write enable input. To read the contents of an address line the read enable bit needs to be selected to enable a series of multiplexers which will select which memory address line is being read from.

Note that the size of memory depends on the number and size of programs being stored. 64 words seems like a sufficient amount to run several programs but may need increasing if during implementation memory becomes full. The memory circuit will also be large and this could lead to slow runtime in Logisim. The size of memory may need to be decreased if the loading time for the CPU is deemed too long to be acceptable.

Memory population in Logisim is also an issue. Since the memory will be built from elementary gates data cannot be loaded via a text file as with the pre-built Logisim memory unit. This means that all of the instructions and data stored will need to be input manually. This is a laborious and time consuming process and furthermore when the Logisim program is closed the data will be lost. An

alternative to writing the data in manually on each start-up is using a ROM component to store the instructions and populate the memory unit. Since this is not in the scope of the project the pre-built ROM component could be used.

4.2.8 – Component Interaction

Now that each individual element has been designed we shall look at the connections between them. In chapter 2 the idea of the random logic control unit was discussed. Along with this a diagram for a single bus CPU. This is a simple idea which is easy to understand therefore it is a good solution for this project. To recap, each component is connected to a single system bus via tristate buffers. A control unit is used to decipher instructions and set the signals that are used to control the buffers, thus controlling data flow onto the bus. The control unit also selects which component the data on the bus is being clocked into.

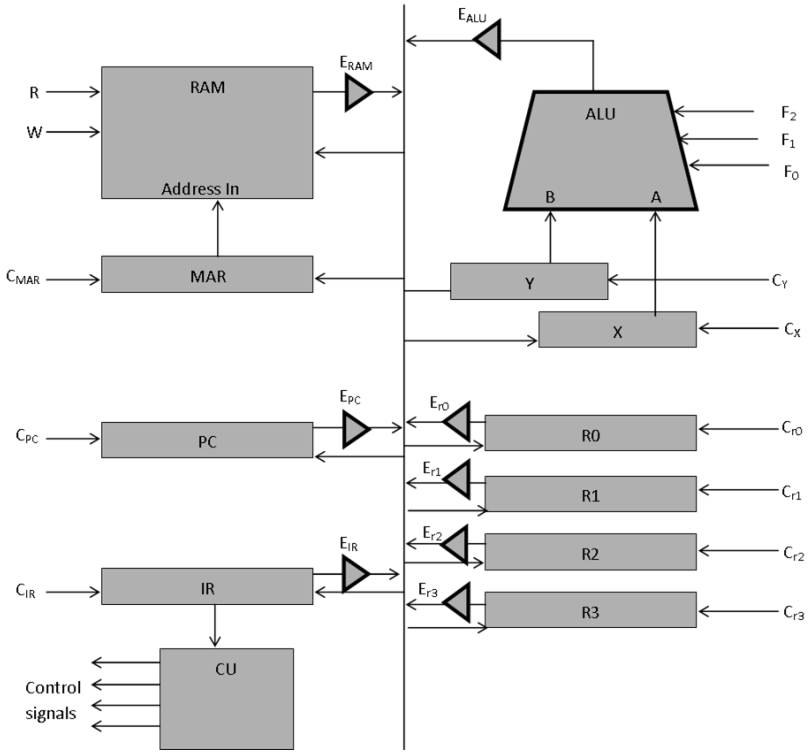


Figure 4.12 Initial design of single bus CPU

There are multiple ways of designing a single bus CPU. Figure 4.12 shows an initial design similar to the one seen in chapter 2. The structure works as expected with instructions being fetched from memory using the address in the program counter and stored in the instruction register. Instructions are decoded using the control unit which then sets appropriate control signals for that particular instruction in order to execute it. Control signals include tristate buffer enables, clock enables for each component and also the ALU op-code (F_0, F_1 and F_2). Once the instruction has been completed the program counter will point to the next instruction in memory and the process will start again.

This structure contains four general purpose data registers but despite this two intermediate registers are also used. The intermediate registers store intermediate values used in an ALU calculation before the calculation takes place. This is necessary due to the fact that only one piece of data can be on the bus at a given time. Therefore during an instruction, such as `ADD R0, R1`, simply enabling the contents of R0 and R1 onto the bus will cause a conflict. Instead, to perform this operation firstly the contents of R0 and R1 will need to be moved to intermediate registers X and Y respectively. Once this has been completed the op-code for the ALU will be set and the ALU will be enabled allowing the contents of the intermediate registers through the ALU and the result of the operation back onto the

bus. After this the result can be clocked into a register or memory location. This is a limitation as operands need to be moved one at a time into intermediate registers before the operation takes place taking up two clock cycles where ideally this would take one.

To accept immediate addressing the contents of the instruction register can be written onto the bus by setting the E_{IR} bit for the tristate gate. This however, only allows the last 8 bits onto the bus which represent the immediate value. This needs to be clocked into an intermediate register if an operation is being performed on it. The process of addressing to or form memory is similar to this. A memory address will be written onto the bus from the instruction register. However since memory addresses are 6-bit and immediate values are 8-bit the first two bits of the address will be 0. In order to cope with this the MAR needs to be able to accept 8-bit addresses and then split the bits so that only the 6 relevant bits are sent to the RAM. If the read bit (R) for memory is enabled then address stored in the MAR will be read from and can be sent onto bus by enabling the memory buffer bit E_{RAM}.

This design should be capable of meeting the requirements however it has an issue. Since the whole instruction is sent to the control unit, from the instruction register, it will need to decode every possible combination of source and destination operands for each instruction. For example to decode the `MOVE` operation the control unit will need to include circuits to move from register 0 to 1, 0 to 2, 0 to 3, etc. This means 12 individual circuits will need to be built to decode which combination pair of the four registers is being used. This does not include the other addressing modes immediate to register, memory to register or register to memory which only add to the issue. If the CPU was implemented in this way the control unit circuit would be very complex and very large.

An alternative to this design is to include a register file, which contains the four data registers, with separate buses leading directly to the ALU (figure 4.13). It works by sending the source register and the destination register, if needed, directly from the instruction register into the register file. This means the control unit does not need to decode every case for an instruction because the operands have already been decoded outside of the control unit. This eliminates the problem of complexity and allows for a more general approach to instruction decoding. Using the previous example; decoding the `MOVE` operation now only needs cases for register to register, immediate to register, memory to register and register to memory as opposed to every possible case operands for each addressing mode. This is the same for every instruction; each will have four different variations depending on addressing mode. This will be explained in more detail in the control unit design part of this chapter.

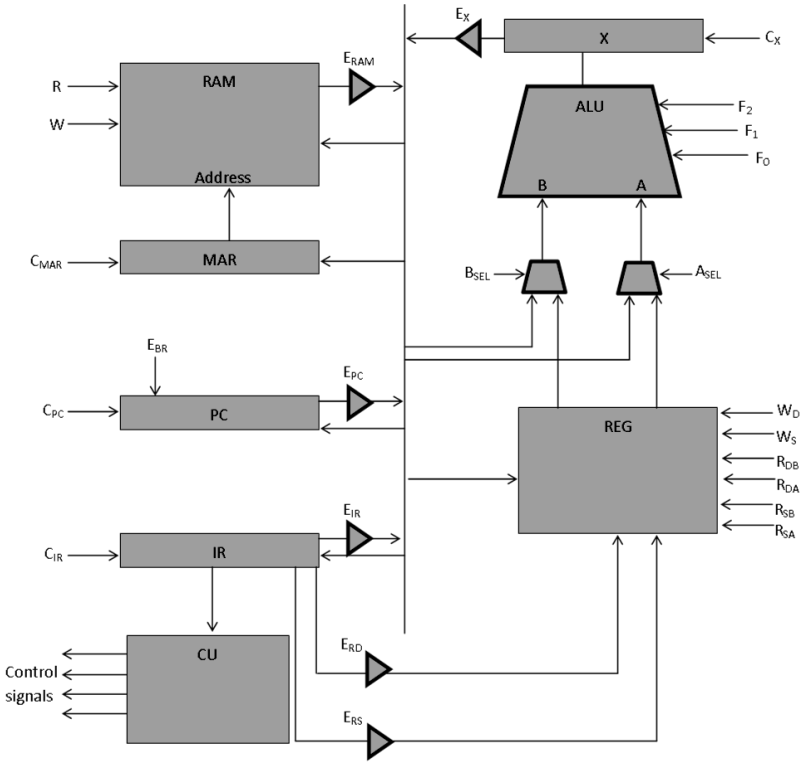


Figure 4.13 Final CPU design

The structure of figure 4.13 allows for the register file to be connected directly to each ALU data input. This is beneficial because it means that the two intermediate registers storing values before a calculation are unnecessary. We have already outlined the instruction set architecture for the CPU and as we know it has a one and a half machine structure. As at least one of the operands is always a register we can connect the register file directly to the ALU. This frees space on the main system bus allowing another operand such as an immediate value or value from memory to flow into the ALU at the same time as a register value. This lack of conflict means that no intermediate registers are needed before the operation takes place. Two multiplexers A and B are used to select which bus the operand has come from. These are both controlled by the control unit. The ALU still needs an intermediate register X to store the result of each operation because one of the operand values may be on the main system bus and so outputting the ALU result immediately will cause a conflict.

The register file itself has many inputs that are controlled by the control unit. Firstly two inputs from the instruction register which are the 2-bit source and destination register locations enabled by E_{RS} and E_{RD}. Each instruction will have a source register but only register to register instructions will enable the destination register. The R_{SA} and R_{SB} read the source register onto bus A and B respectively they are both 1-bit controls. This is the similar to R_{DA} and R_{DB} which read from the destination instead of source. Input bit W_S allows the source to be written to and W_D allows the destination to be written to.

Please note that source and destination are slightly ambiguous terms because in some instructions such as `ADD R0, R1` the source is R0 and the destination is R1 because the operation adds the contents of R0 and R1 and stores it in destination R1. However in other instructions such as `ADD #1, R0`, R0 is the source register because of the order in the instruction (operand 1 is always the source) but actually the result of the operation is being stored in R0 itself making it the destination too.

Some may suggest that the register file only needs to wire the source to A and the destination onto B with only one multiplexer for bus B to select between a register and immediate or memory value. This intuitively makes sense because the source register is used in every operation but destination is only used occasionally. To see why this will not work as intended we need to examine the subtraction operation in detail. `SUB R0, R1` subtracts the contents of R0 from R1 and stores the result in R1. R0

is the source register and R1 is the destination register. If we could only write the source to A and destination to B then R0 would be the A input to ALU and R1 the B input. This is fine for all other operations because they are commutative so the order does not matter. However for subtraction the order does matter; X – Y is not necessarily the same as Y – X. The ALU subtraction works by taking B from A and so would take R1 from R0 in the above case which is incorrect. Some may argue that for subtraction switch the operation inside the ALU so that it calculates B minus A. This would not work either because of other subtraction operations such as `SUB #1, R0` (takes 1 away from R0). In this case B would be selected from the main bus as it is an immediate value and A would be the source register R0 if the operation is swapped then A will be subtracted from B which is 1 subtract R0 which is the incorrect order. The only solution to this problem is to have the ability to write both the source and destination registers onto A or B and other values from the main bus onto A or B. This allows all possible orders and can easily be controlled by the control unit.

Apart from the register file this structure works in a similar way to the previous design, from figure 4.12, apart from the fact that the contents of registers cannot be moved to other registers in the same way. In figure 4.13 in order to move the contents of one register to another it must travel through the ALU. This is only a small draw back and may add an extra clock cycle if it needs to be stored in intermediate register X.

Branching is also considered in this structure. The program counter has a branch enable bit, E_{BR}, which allows it to jump to a specified address location when clocked. The address will be specified by the instruction and sent into the program counter along the main system bus from the instruction register.

The design in figure 4.13 is an improvement over 4.12; despite this they both suffer from the limitation of bus width. Logisim only allows for components to be connected to each other if they use data of the same bit length even if the wire is not active. For example in figure 4.13 instructions can be sent onto the bus from memory, these are 16-bit in length. The bus is also connected to the register file which takes data inputs that are 8-bit. This causes a conflict because the difference in data length means the components are not compatible even if the register file is not enabled. Addresses also travel along the bus but as they come from the instruction register they are 8-bit and the MAR will split the irrelevant initial two 0 filler bits from the memory address which is in fact 6-bit. A similar approach can be used to solve the above problem. In order to connect components with different data length inputs and outputs we need to fill each word with 0's before it is put onto the bus and then split the word down once inside another component. This is a relatively simple solution; an example of this would be sending an immediate value from the instruction register into the ALU for a calculation. The longest word length that travels along the bus is 16-bit (instructions). An immediate value is 8-bit and so we need to add 8 more 0's to the front or end of the immediate value before we put it onto the bus. The data will then be sent to the ALU which uses 8-bit words to perform calculations. Inside the ALU we will need to split the eight filler 0's from the actual 8-bit immediate value. Once this has been completed the calculation can take place as normal. This concept will need to be applied to all components in order to deal with the difference in word length.

4.2.9 – Control Unit

As mentioned in the previous section the control unit component controls all of the tristate enable and clock bits that dictate the flow of data inside the CPU. The control unit uses the fetch-execute cycle where it first sets the appropriate gates to fetch the instruction from memory and then goes on to decode and execute the instruction.

We briefly looked at the composition of the control unit in chapter 2, section 2.1.4, where we saw how instruction decoding is combined with timing signals in order to set signals at the appropriate times. Firstly we need to look at how timing signals can be generated inside the control unit. The

component which generates the timing signals is called a sequencer. It is comprised of a series of JK flip-flops which take a clock signal input and J, K inputs. Setting J and K to be constantly 1 allows the JK flip-flops output to be altered with each clock pulse. If the output of one flip-flop is used as the clock pulse to each subsequent flip-flop then a counter can be created by performing the AND of the outputs and the complements of the outputs from each of the JK flip-flops. The maximum number of timing signals that can be generated is equal to 2^N where N is the number of JK flip-flops in the series.

Using the design from figure 4.13, the control unit takes input from the instruction register and performs a decoding operation in order to determine which signals need to be set for the current instruction. Only the first 6 bits of the instruction need to be decoded; these are the 4-bit operation and the 2-bit addressing mode. If the CPU design of figure 4.12 was being used then the whole instruction would need to be decoded as we would have separate cases for each operand being used leading to a huge number of total cases. Since the design in figure 4.13 allows for a more concise control unit design there will only be 4 different cases for each instruction because the 2-bit addressing mode encodes 4 different cases. Note that there will only be 4 different cases for ALU operation instructions because branch instructions are not affected by the addressing mode. Decoding the op-code and addressing mode is relatively simple, in a similar way to the output of the JK flip-flops in the sequencer, the op-code bits are split and then each bit and its complement run into a series of AND gates. Each AND gate will only result in true if all inputs are true so, for example, to decode op-code 0000 the complement of all of those bits will run into one AND gate which will result in true if the op-code is ever 0000. For code 0001 the complements of the first 3 bits will run into an AND gate and the last bit itself will also run into the same AND gate; making the result true only when 0001 is the op-code. Using this system each op-code and addressing mode can be decoded and a unique AND gate will emit true for each instruction type.

Now that we have designed a decoder and sequencer we need to determine which of the control signals need to be set at which time for each instruction to be executed correctly. To do this we need to create a table of micro-instructions that break down each instruction into a series of steps that can be executed in one timing signal. This is a time consuming process as each instruction needs to be broken down into stages and then the control signals need to be determined for each stage. Table 4.6 shows a small portion of the instructions from the CPU's instruction set which have been broken down into operations in RTL and control actions deduced from these. This is only a small portion of the control signals and the complete set can be found in the appendix A.

Instruction	Operations (RTL)	Control actions																							
		R	W	C_{MAR}	C_{PC}	C_{IR}	C_X	W_S	W_D	R_{SA}	R_{SB}	R_{DA}	R_{DB}	E_{MEM}	E_{IR}	E_{PC}	E_{BR}	E_{RS}	E_{RD}	E_X	A	B	F_2	F_1	F_0
Fetch	Mar ← PC	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	IR ← [Mar]	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	PC +1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOVE 00	X ← Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
MOVE 01	Ri ← Imm	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
MOVE 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	Ri ← [Mar]	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
MOVE 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	[Mar] ←X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
ADD 00	X ← Ri + Rj	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	0	1
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
ADD 01	X ← Imm + Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
ADD 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [mar] + Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
ADD 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] + Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
	[Mar] ← X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Table 4.6 A section of the control signal table

Table 4.6 is important in designing the control unit because it allows us to see which control signals need to be set at which time in order to execute each instruction. Please note that the control signals are based on the figure 4.13 design and we shall now be considering this design in order to understand the table. We shall look at the action of the fetch instruction. This fetches the instruction from memory and sends it to the instruction register in order for it to be decoded. In the first step the contents of the program counter are sent into the MAR. This is done by enabling the program counter tristate gate (E_{PC}) and clocking the MAR (C_{MAR}) simultaneously. The next step fetches the contents of the memory address given by the MAR and transfers it into the instruction register. To do this we enable the read memory bit (R) and, simultaneously, enable the contents of memory onto the system bus (E_{MEM}) and at the same time clock the data from the bus into the instruction register (C_{IR}). The final part of the fetch phase is to clock the program counter (C_{PC}) this allows the incrementor to add 1 to the address stored in the program counter so it points to the next instruction in memory at the end of the fetch phase.

Now that the fetch phase has been completed it is time to perform the execute stage in which the instruction we just fetched from memory is executed. We decode the instruction's op-code and addressing mode as explained in section 4.2.9. This leaves us with multiple versions of the same instructions but with different addressing modes; these each need to be broken down into individual microinstructions because the control signals need to be set differently for each one. For example, Move 00 which moves data from one register to another requires two clock cycles whereas Move 01, moving an immediate value to a register, only takes one. Once the execute phase has finished the control unit moves back into the fetch phase and the process is repeated.

In order to combine the timing signals and decode signals with each microinstruction we used a series of AND gates that take one timing signal and output from the instruction decode. These AND gates flow into a series of OR gates (one for each control action) that set control signals for the control unit to output. For example if the Move 00 instruction was decoded, it has two microinstructions so therefore requires two AND gate rows. The first with time 0 and the next with time 1 (both would have the output from the decode AND). The output from the 'time 0 AND' would be run into the OR gate for C_X, R_{SA} and E_{RS} because these need to be set for the first instruction. Since one of the inputs into the OR gate will be true the output will also be true and so the control signal will be output. On the next clock cycle the sequencer will move to time 1 and the next AND will become true which will set the signals for the second microinstruction for Move 00.

This implementation requires us to be able to switch between the fetch and execute phase upon completion of instructions. To do this we can use an RS flip-flop to switch between phases.

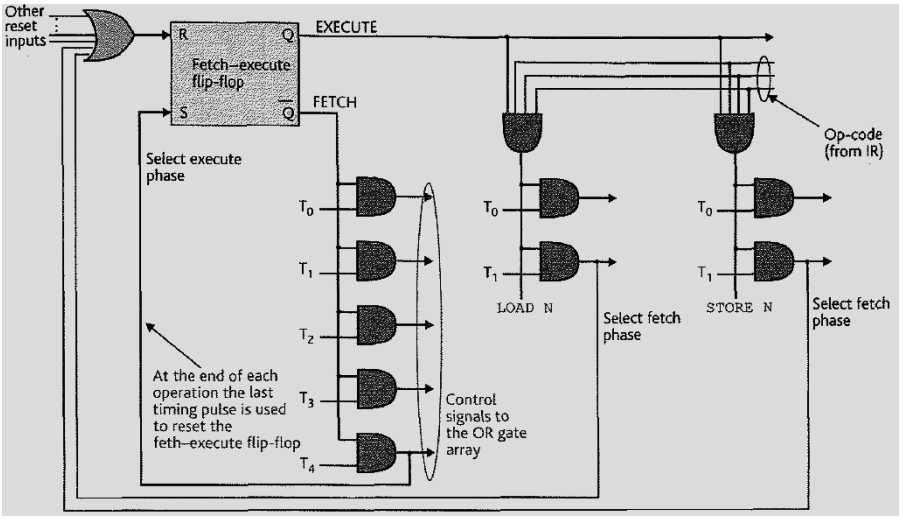


Figure 4.14 Fetch-execute flip-flop [2]

As we can see from figure 4.14 when the output of the RS flip-flop Q is 0 the control unit is in the fetch phase but while Q is 1 the execute phase is enabled. It also illustrates how the rows of AND gates are used to combine instruction decode to timing signals. An extra AND gate is used at the end of the fetch phase to set the RS flip-flop which will enable the execute phase to take place. This also resets the timing signal generator to time 0. With the execute phase enabled, instructions can be decoded and performed. The sequencer is also reset at the end of the execute phase. The RS flip-flop is reset at the same time to enable the fetch phase to recommence; this process is repeated.

Chapter 5: Implementation

In this chapter we will look into the implementation of the CPU; how the components interact with each other and how some of the concerns raised in the design chapter were overcome.

Due to the design nature of the project the design chapter and the implementation chapter will be relatively similar. In order to not to reiterate material previously covered, descriptions of how circuits work will be kept brief. Importantly, areas of design and implementation that differ will be highlighted and discussed in more detail.

Before we look at the implementation of the CPU it is worth noting that this project is only a simulation. Logisim has some limitations, involving infinite oscillations, which would not happen in real hardware because it cannot simulate the physical variations in signal times. [17] Logisim also has a bug where wires will carry unknown or error values despite them being connected correctly. [18] These issues meant that some compromises had to be made during the implementation to avoid Logisim's errors. The compromises will be discussed in this chapter.

The CPU has a modular design with separate Logisim files for each main component such as the control unit, memory, registers and ALU. These files are used as imports into the main CPU circuit. The files themselves are highly modular with circuits being broken down into sub-circuits such as the ALU including a sub-circuit for addition which contains multiple full adder sub-circuits. From preliminary testing this structure appears to have a much faster loading time compared to putting all circuits in one Logisim file.

5.1 – Implementation of Component Interaction

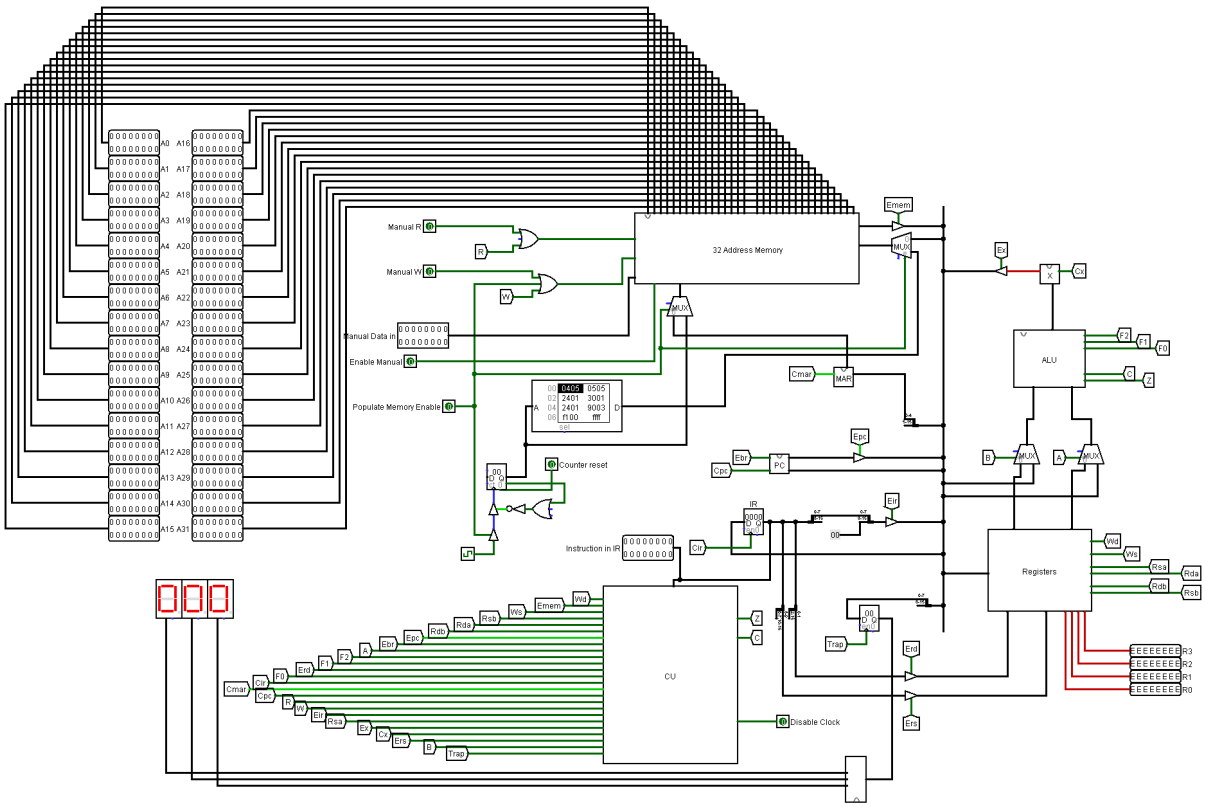


Figure 5.1 The complete CPU in Logisim

As you can see from figure 5.1, the implementation of the CPU closely follows the structure of figure 4.13 in the previous chapter. A single main system bus connects all components, with secondary

buses from the register file to the ALU. It also works in the same way with the program counter giving the address of the instruction to the MAR in order to fetch it from memory.

The instruction is clocked into the IR and then the control unit decodes the instruction and sets the signals for its execution. There are, however, a number of additions to the circuit that were not included in the design.

An issue described in the design chapter was the problem of connecting multiple components together that have different bit length inputs and outputs using the same bus. Logisim does not allow components with varying bit lengths to be connected along the same wire. To overcome this every piece of data that travels on the bus must be the same length as the largest piece of data, the 16-bit instruction. To make every piece of data 16-bit filler 0 bits were used in conjunction with bit splitters to add the 0's and take them off again once the data has moved to a new component. The bit splitters can be seen throughout the CPU. Note for components such as the ALU and Program counter there appears to be no bit splitter involved. This is because the bit splitter is contained inside the sub-circuit to improve the appearance of the main circuit.

Something to note is the use of Logisim's tunnel component throughout many of the circuits in the CPU. The tunnel component connects two tunnels, with the same name, as if there were a wire between them. This is very useful when creating large circuits as it saves time and makes the circuits easy to view and understand.

5.1.1 – ALU Implementation

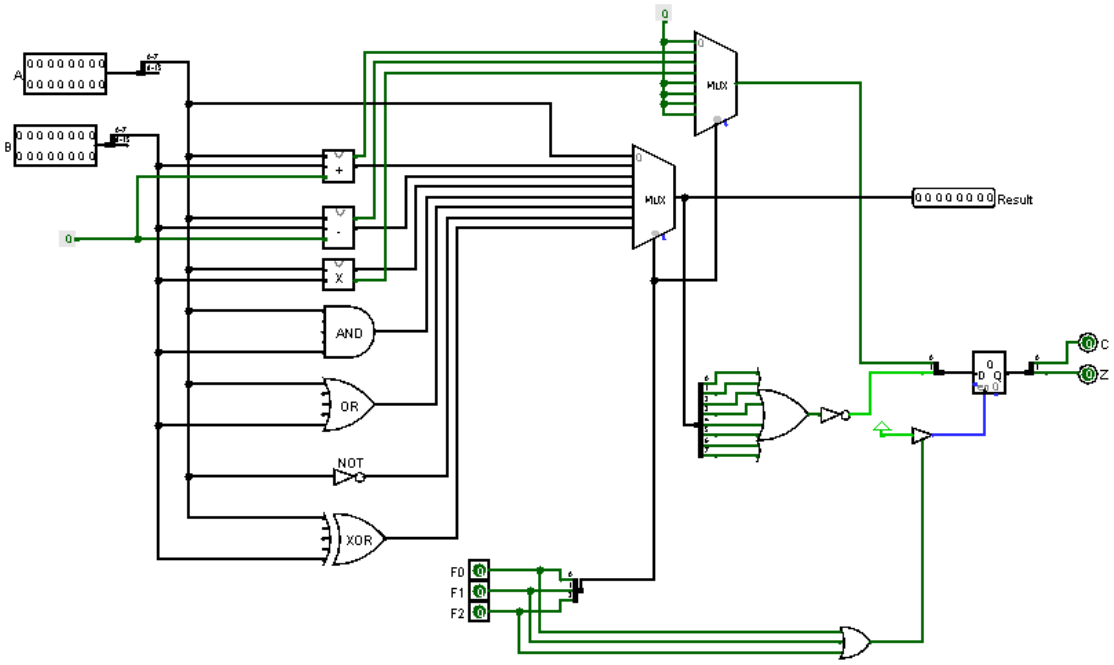


Figure 5.2 ALU circuit

The ALU is very similar to the one described in the design chapter. There are some differences however; firstly the ALU now takes two 16-bit inputs due to the component connection issue. Bit splitters are used to split off the filler 0's and so operations are still performed on 8-bit words. The operation is chosen using a multiplexer, with the op-code inputs (F0, F1 and F2) used as the selection bits. The result from the ALU does not need to be converted to a 16-bit output because it is connected to 8-bit intermediate register X. The filler 0's are added inside the register X as it is connected to the main system bus.

A 2-bit condition code register has been added to save the status flag values carry (C) and zero (Z). The values are clocked into the register by enabling a tristate gate. The register is only clocked when at least one of the op-code inputs is 1. If this was not the case, and the register was clocked every time an instruction goes through the ALU then the status flags may be overwritten when a `MOVE` command is executed which is undesirable behaviour as only arithmetic and logical operations should alter the status flags. The flags are stored as a 2-bit number but are split on output to be input into the control unit as individual bits.

5.1.2 – Arithmetic and Logical Operations Implementation

The arithmetic and logical operation components contained inside the ALU are identical to the circuits described in the design chapter. Therefore it is unnecessary to revisit them in this chapter. Complete diagrams of these circuits (and all circuits) can be found in appendix B.

5.1.3 – Register Implementation

This CPU includes many different kinds of registers. In figure 5.1 you may have noticed that the instruction register is the pre-built Logisim register component. This is because the custom instruction register produces an error value which breaks the system. The error is produced upon initialisation because the value of the register is unknown as no value has been clocked into it. This would be fine for other registers because the value can be clocked in at a later time when needed. However, because the instruction register links to the control unit the error value is sent through the control unit and stops the sequencer. If the sequencer is stopped then no instructions can be executed and the system fails before it has started. This does not meet the requirements of building the CPU from only elementary gates but since this is only one register and project demonstrates how other registers have been constructed it does not detract from the overall project. For completeness an instruction register circuit has been built as a demonstration despite it not being used in the CPU.

The 5-bit memory address register (MAR) and 8-bit intermediate register X are constructed from D flip-flops as described in the design chapter; these circuits can viewed in appendix B.

The program counter circuit, displayed in figure 5.3, follows the same structure as the program counter design.

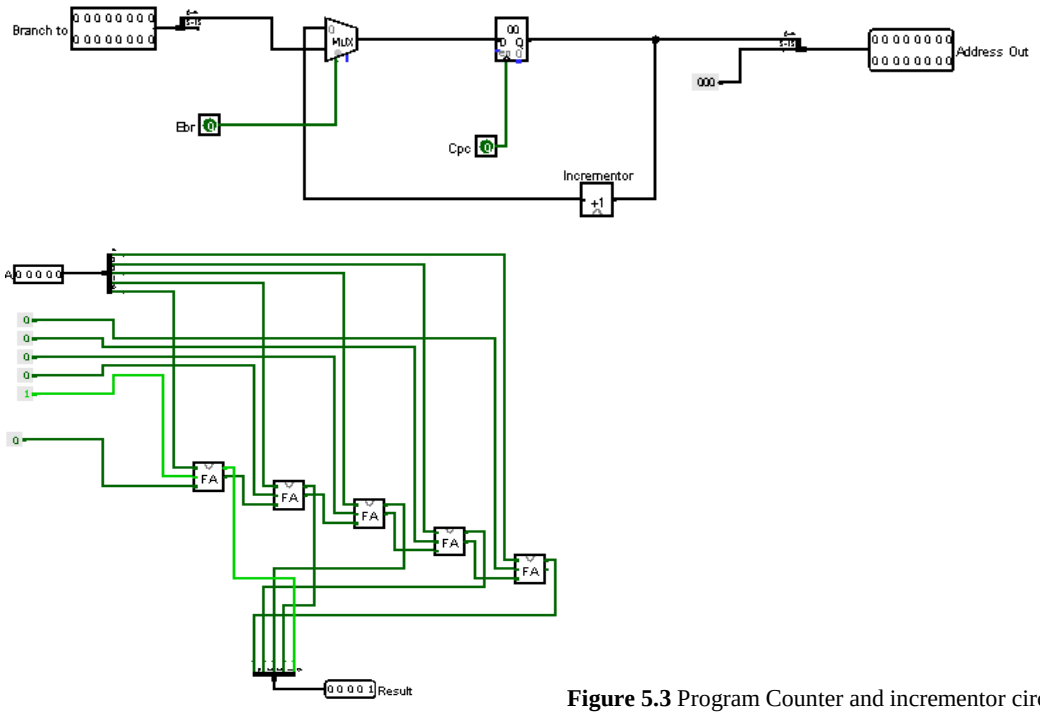


Figure 5.3 Program Counter and incrementor circuits

The incrementor circuit remains unchanged from the design, using a series of full adders to add the constant 1 to the input address, but the program counter register which stores the next address to be executed has been made from a prebuilt Logisim register instead of an elementary one. This is because using a register from elementary gates means that the register is set to trigger on a 'high level' meaning that when the clock is true the data is instantly clocked into the register. This is fine in usual cases however in this case if Cpc was set to 1 then the value would be constantly clocked in from the incrementor and this would cause an infinite loop. Logisim deals with infinite loops by stopping the system and displaying an "Oscillation Apparent" [19] error message. In order to solve this problem a prebuilt Logisim register needs to be used as the trigger value can be changed. Changing the register to trigger on an edge (either rising or falling), rather than on a high or low value, means that the register is only enabled when the clock changes from high to low or low to high. This stops the infinite loop and makes the system work as intended.

A large part of the design chapter was deciding how the general purpose registers should be organised. It was decided that a register file should be created that contains all four data registers.

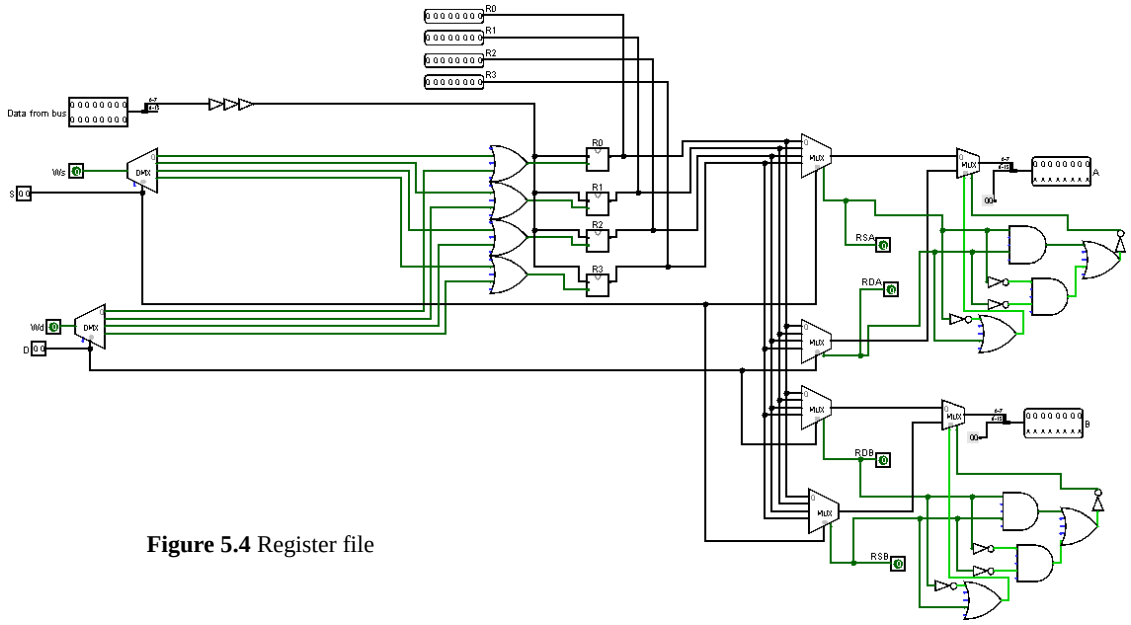


Figure 5.4 Register file

The four 8-bit data registers are made in the same way as every other register circuit. Two demultiplexers are used to choose which register is to be selected for a write operation using the source and destination register locations respectively as selector bits. As each register can be written to acting as the source or destination register; a series of OR gates are used to enable the write bit.

The logic for reading from registers is slightly more complicated because both the acting source and destination registers can be written to bus A or to bus B. Buses A and B lead into the ALU's A and B inputs respectively. They are appropriately named, so if a value is written onto bus A then it will be used as value A in the ALU and a value written onto bus B will be used as value B in the ALU (see figure 5.2).

Four multiplexers (arranged vertically to the right of the registers in figure 5.4) are used to select which of the four registers are being read to bus A and B. For example the top multiplexer selects using the source register as selection bits, allowing the selected source register onto bus A. The second multiplexer down selects uses the destination register as selection bits, allowing the selected destination register onto bus A. The two remaining multiplexers in the stack work in a similar way but instead read the destination and source onto bus B.

A pair of smaller multiplexers (right most side of figure 5.4 arranged vertically) are then used. One to select either the source onto A or the destination onto A, but not both as this would cause a conflict; and another to read the source to B or the destination to B, but again not both. Filler 0's are then added to result. This is necessary because even though the data doesn't travel directly on the main system bus, bus A and B are connected to the main bus via multiplexers to select between immediate and register values. As all inputs to multiplexers need to be the same bit length the register file output needs to be made up to 16 bits as immediate values have to travel along the main system bus and as discussed before anything on the main bus must be 16-bit.

Of note are the 3 buffer gates between the data input and the registers themselves. These appear to have no effect on the circuit and do not alter its behaviour. Despite this they have an effect on the CPU as a whole. The CPU itself is very fragile with data travelling extremely quickly around the circuit. During a preliminary piece of testing it was found that wrong values were being written into registers when transitioning between the execute phase of an instruction and the fetch phase of the next instruction. Therefore the 3 buffer gates were added to create a tiny delay in the data signal so that values cannot be accidentally written into the registers.

Initially the custom made registers did not work correctly when the register file circuit was introduced into the main CPU. The registers would produce error values when written in to which would break the system. To combat this, a solution was found which involved changing the pull behaviour of the pins used in the individual data register circuits. The pull behaviour changes the value of the pin on error input. Setting a pin to pull up means that on an error value the pin will be set to 1 and pull down means the value will be 0. Changing the data input and output to pull down and pull up respectively fixed the issue. This is worth mentioning because without this change the custom made registers will not work correctly.

5.1.4 – Memory Unit Implementation

It was stated, in the design chapter, that the memory unit would be capable of holding 64 individual 16-bit words. A memory unit matching this specification was created and successfully implemented as an individual circuit. However when imported into the main CPU circuit made the loading time extensive. The decision was made to adapt the memory component, converting it into a smaller 32-address unit. Having a smaller memory unit means less instructions and values can be stored which affects the length of programs that the CPU can run. Nevertheless 32 addresses are still enough to compute the majority of programs and the benefit to loading time is large enough for the trade-off in space to be overlooked. Creating the 64 address unit was a difficult and time consuming process and includes some interesting design features such as linked decoder circuits so will be included in appendix B for viewing purposes despite it not being used in the project.

Changing the memory to 32-address means the input to the circuit only needs to be 5 bits in length. This is opposed to the 64-address circuit's 6-bit address input. Since the address is input from the MAR, the MAR only needs to be a 5-bit register not 6. Changes also needed to be made to the instruction architecture itself, as values used in addressing modes involving memory are taken from the last 5 bits of the instruction, not the last 6. These changes were relatively minor and easy to implement.

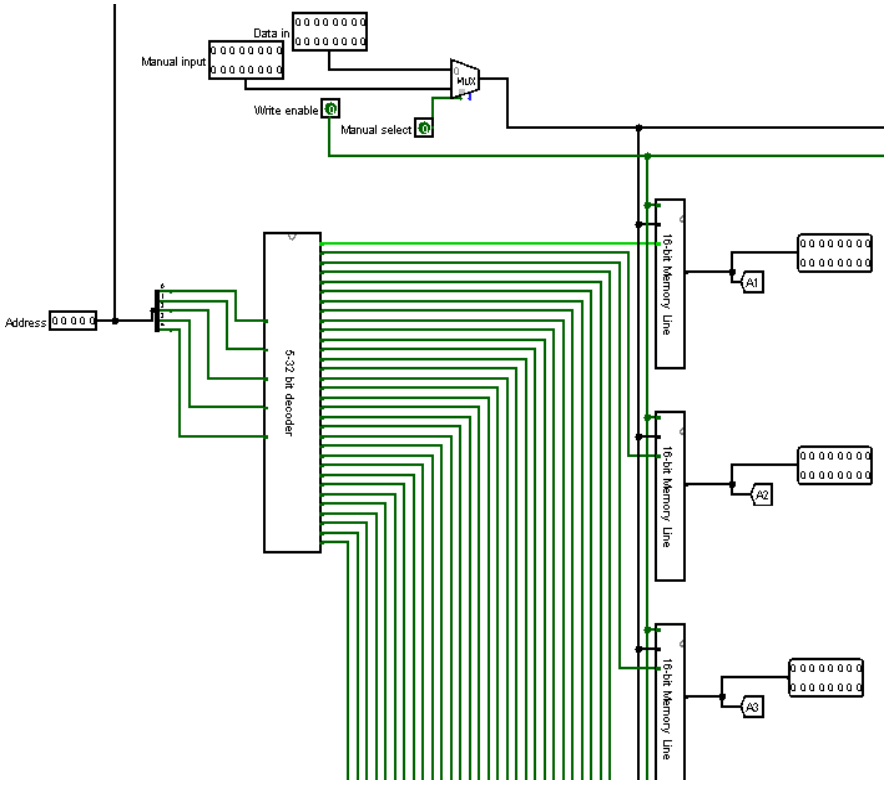


Figure 5.5 A section of the memory circuit

Figure 5.5 shows a part of the memory circuit as the complete circuit is too large to show in one diagram. As you can see a 5-bit address is input into the circuit and is sent through a 5 to 32 bit decoder to select one of the 32 memory lines. A prebuilt Logisim decoder could have been used here but this decoder was built from elementary gates to demonstrate how such a component can be constructed. The 16-bit instruction to be saved is also input and is connected to each of the 32 memory lines.

Each memory line is capable of holding a 16-bit word; naturally there are 32 memory lines, one for each memory address. The write bit, set by the control unit, is also input into the memory lines along with the designated decoder line.

The memory line sub-circuit itself consists of 16 D flip-flops. The 16-bit data coming into the memory line is split into its individual bits with each bit being input into one of the D flip-flops for storage. The flip-flops are enabled when both the write bit and the enable bit from the decoder are both true. If this is the case, the 16-bit data input is written into all the flip-flops at the same time. The output from the flip-flops are combined to output the 16-bit word being stored so that it can be read from memory later.

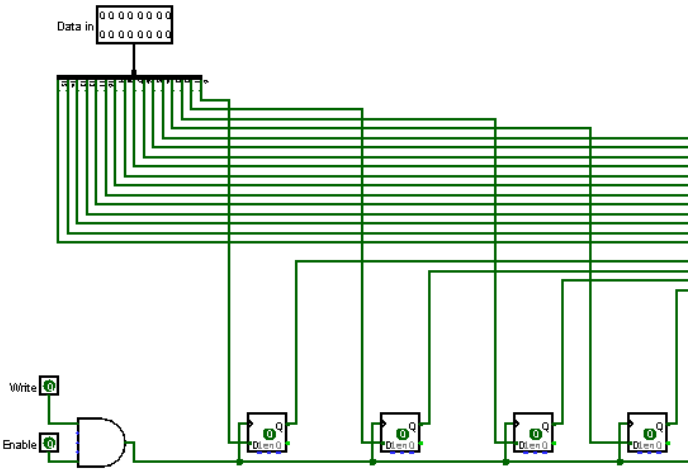


Figure 5.6 Part of the 16-bit memory line

From figure 5.6 we can see that pre-built Logisim D flip-flops are being used as opposed to elementary sub-circuits. This is due to a problem that occurred during some rudimentary testing. The custom flip-flops allowed an initial population of memory with data; however when data needed to be written into memory, inside the running of a program, an oscillation error occurred. Furthermore Logisim has a bug where, on occasion, the value of a wire is unknown and appears blue despite the circuit being correct. This would happen frequently in the memory circuit and cause the unit to work incorrectly as it would not know the value of some of the bits that were stored when executing a program. The only way to fix this issue is to turn off Logisim's simulation mode, reload the circuit and then re-enable the simulation. This was less than ideal and combined with the oscillation error meant the CPU could not execute all instructions correctly. As in the last case, this does not detract from the project as this oscillation error and unknown wire value bug are unavoidable due to Logisim's limitations. A D flip-flop elementary sub-circuit is still included in the memory file to demonstrate how it is constructed.

Reading from memory is simple, a multiplexer is used which has all 32 of the 16-bit words being stored input into it with the address used as the selector bits. The multiplexer is enabled by the read bit which is controlled by the control unit.

5.1.5 – Control Unit Implementation

The control unit circuit is the largest of all the circuits in the CPU; it controls the movement of data around the CPU by enabling other components corresponding tristate gates. The control unit takes input from the instruction register which will be holding the current instruction to be executed and decodes it. Then the control unit sets the appropriate gates at the correct times for the instruction to be completed. It will then revert to the fetch phase, in which a new instruction is fetched from memory and the process is repeated.

In order to set the tristate gates at a certain time a timing signal generator (sequencer) needs to be created that tells the control unit the current clock time status. This circuit is effectively a counter; counting up from clock time 0 up to a clock time limit that can be set in the hardware.

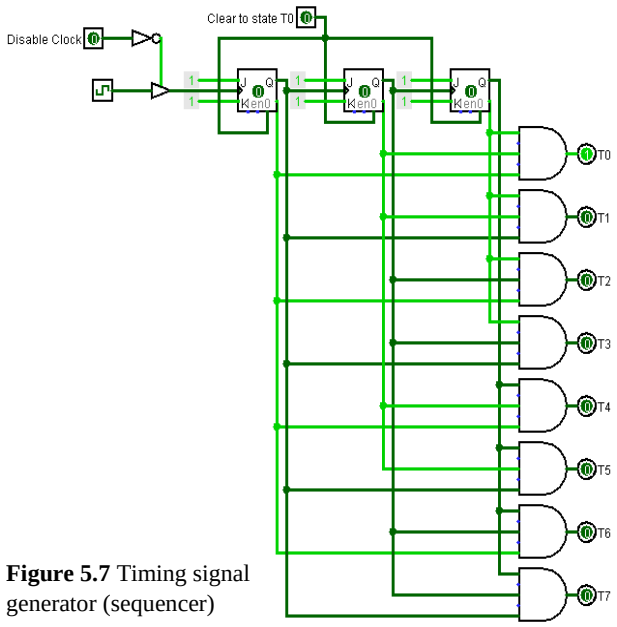


Figure 5.7 Timing signal generator (sequencer)

The sequencer circuit contains a clock which alternates between values 0 and 1. As this happens the series of JK flip-flops counts up by 1 each time the clock changes from 1 to 0 (falling edge). A series of AND gates are used to decode the values from the flip-flops into unique output timing signals. The total number of timing signals produced by a sequencer is 2^N where N is the number of JK flip-flops connected together. In figure 5.7, as 3 flip-flops are used, 8 timing signals can be produced before the sequencer resets to time 0. This is a more than adequate amount as the longest instruction the CPU can execute is completed in just 3 clock cycles. A reset bit is included to reset the sequencer back to time 0. This is important as the sequencer needs to be reset after each fetch and execution phase.

Pre-built Logisim JK flip-flops are being used because initially using elementary JK flip-flops caused an oscillation error inside the flip-flop itself as the flip-flop could not settle into a stable state due to its sequential nature. This issue was fixed by including a buffer in the circuit as this simulates the variation in signal time inside a physical circuit. [17] However even with this modification the circuit produced an error upon initialisation because the values in the JK flip-flops were unknown; this caused the control unit circuit to stop functioning and so pre-built Logisim flip-flops were resorted to. As in other cases the elementary JK flip-flop is included in the file despite it not being used.

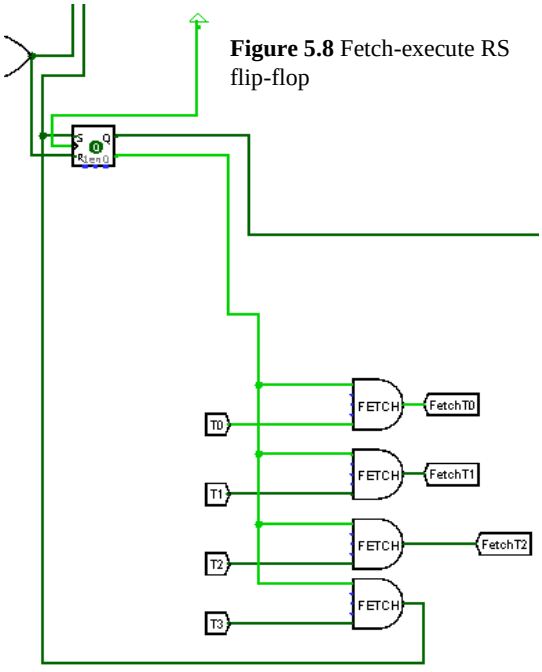


Figure 5.8 Fetch-execute RS flip-flop

We shall now look at how the sequencer's timing signals can be incorporated into the fetch-execute cycle described in the design chapter. Figure 5.8 shows the RS flip-flop that is used to switch between fetch and execute phases. This design follows closely to the one defined in the design chapter; with the RS flip-flop being set after the fetch phase has completed to enable the execution line from output Q. Once the instruction has been executed the RS flip-flop is reset which switches back to the fetch phase.

To decode an instruction a series of AND gates are used which take inputs from the op-code, addressing mode and execute enable bits. Each instruction in the architecture has 4 different AND gates to represent not only the instruction but also the addressing mode.

To execute an instruction a series of AND gates are used which take an input from the instruction selector logic and the timing signal generator. One AND gate is used for each clock cycle that the instruction requires to execute. The number of clock cycles for each instruction can be seen in the control signal tables discussed in the design chapter and found in appendix A. Please note that some instructions such as `FETCH` require more than the designated clock cycles listed in the tables. This is because some instructions need to reset on the next clock cycle to avoid incorrect data movement in the CPU. This is most apparent for operations that happen in one

clock cycle such as moving an immediate value to a register. Without an extra clock cycle, in which the sequencer and RS flip-flop are reset, the instruction would be executed and immediately reset. At the same time the first cycle of the fetch phase would begin; causing errors with data being clocked into incorrect locations.

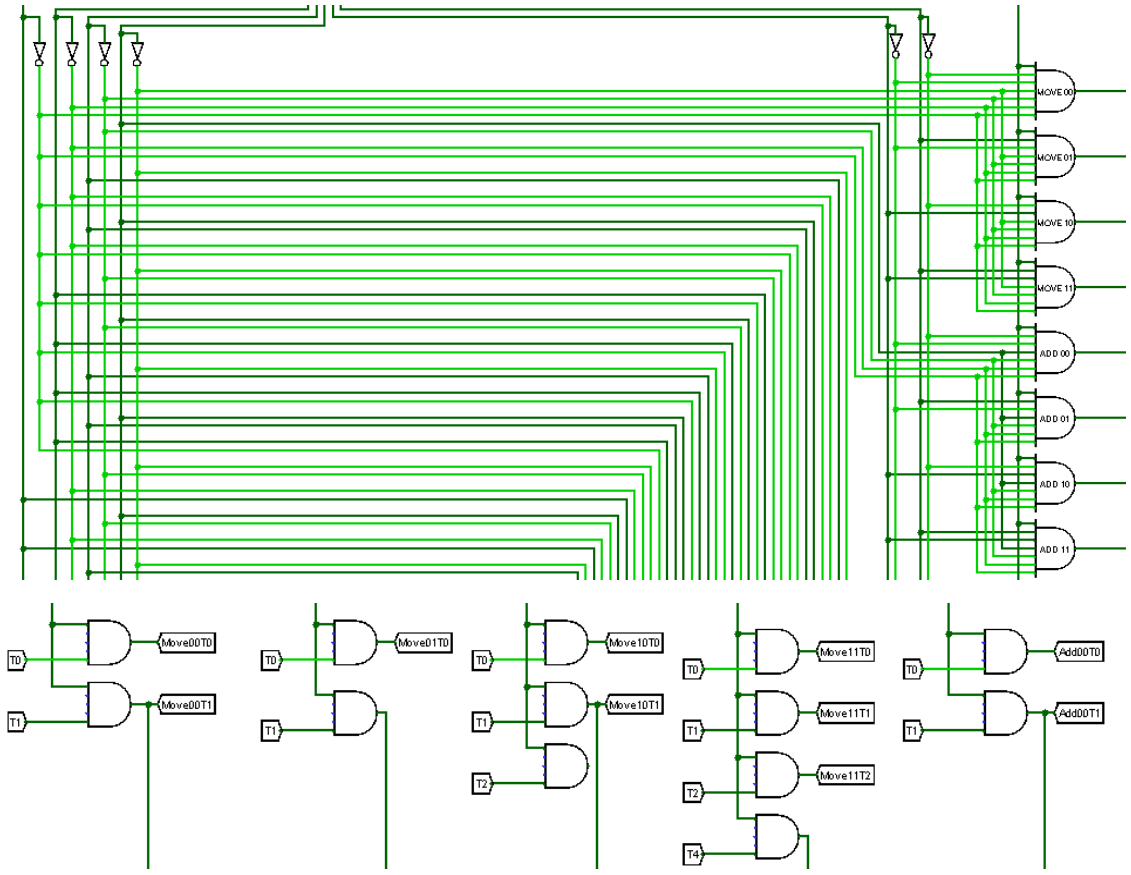


Figure 5.9 Part of the Control unit's circuit to decode and execute instructions

The series of AND gates, in the lower section of figure 5.9, link to an array of OR gates which are used to set the signals for the entire CPU circuit.

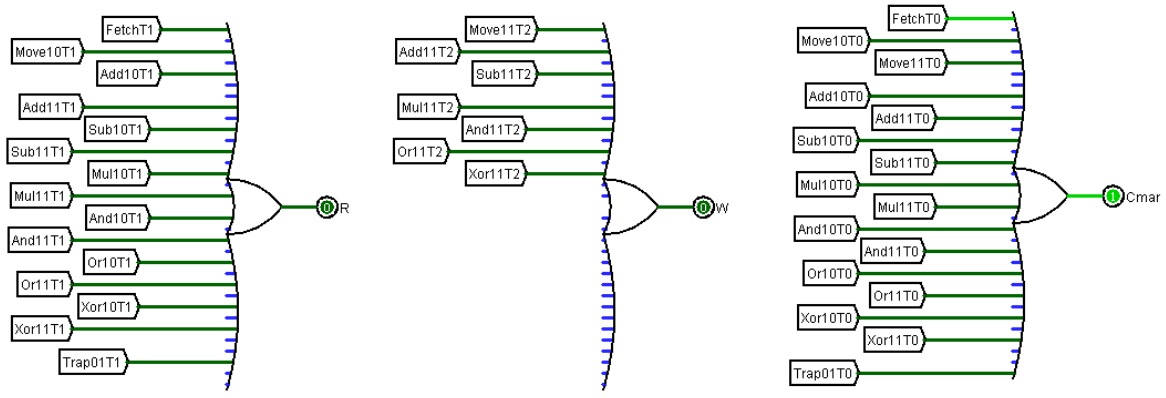


Figure 5.10 Part of the control signal OR gate array

The control signal tables are used to determine which AND outputs flow into which OR gates. For example we can see during the `FETCH` instruction at time 1 the 'FetchT1' AND output will be true which will set the R OR output to true in figure 5.10. There is an OR gate for every signal that needs to be set through the CPU. The sequencer is reset to time 0 at the end of each fetch and execute phase,

this makes sure that every instruction starts from time 0. If this was not the case, control signals may be set at the wrong times causing the CPU to execute instructions incorrectly.

In order to implement conditional branching instructions, the control unit takes status flag inputs from the ALU and uses these to decide if a branch condition is met. These wires are input into the instruction decoding AND gates, making the instruction selection true if the corresponding condition is met.

5.2 – Implementation of Additional Features

The implementation of the CPU works correctly and appears to meet the requirements specification. Nevertheless, some additions to the CPU were made to create a basic user interface allowing interaction with the circuit much easier. Many of the additions include Logisim components, since these are additional features and not in the main scope of the project the use of such components is acceptable.

5.2.1 – User Interface Features

The main CPU circuit contains many Logisim pins which read output values. Most notably, additions were made to the memory circuit to output the contents of every address. Similar pins were added so users can view the contents of each data register inside the register file circuit.

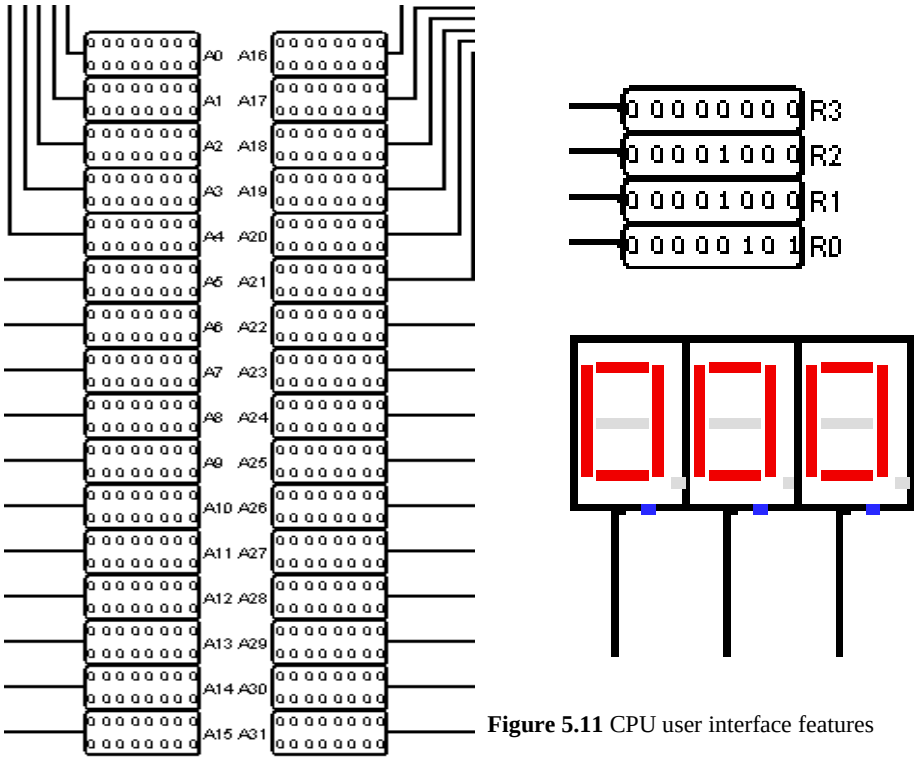


Figure 5.11 CPU user interface features

Two variations of `TRAP` functions were also implemented which are able to convert an 8-bit binary number, held in either a data register or in memory, into a decimal number displayed on Logisim's hex digit display. This is a very useful feature as users may not necessarily know if the result of a program is correct if it is only displayed in binary. In order to convert from an eight bit binary number into a binary coded decimal value (BCD) a circuit was made that implements the Double-dabble algorithm in which a conversion is made by shifting bits to the left and adding three if the value exceeds five. [20] The design of the circuit is adapted from a design by Dr John Loomis based on teaching materials by Professor Richard E. Haskell. [21] This circuit can be found in appendix B.

Another variation of the `TRAP` function was implemented to allow users to stop a program after it has been executed. This is a useful feature as if a program is not stopped the CPU will keep fetching from memory addresses that contain no information. This could override register results from the actual program. Therefore if the instruction containing 16 1's is fetched from memory the control unit's clock is disabled so no more can be fetched, thus ending the program.

5.2.2 – Memory Population Circuit

One of the issues discussed in the design chapter is the issue of memory storage and population. Inputting programs into memory is a time consuming process if the normal pin inputs are used as the user will need to select the individual value of each bit. Furthermore the program stored in memory will not save so the process will have to be repeated every time the user loads the CPU circuit. To alleviate the issue the Logisim ROM component is used in the main circuit. This lets users enter instructions into the ROM by typing each instruction into the allocated memory slot. A counter is then used to count from 0 to 31 in combination with the memory write enable bit. This means that the counter counts up the memory addresses in the ROM and loads their values into the memory unit circuit; stopping when it reaches 31. The ROM component also allows users to save and load programs into it so a previous program does not have to be rewritten each time the CPU file is opened.

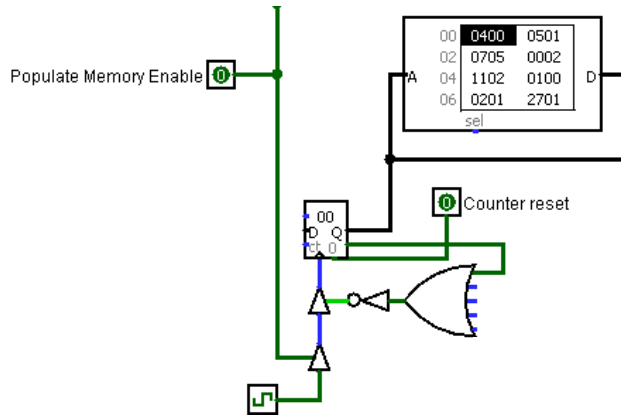


Figure 5.12 Memory population circuit

Using the ROM does have the disadvantage that the user must enter the instructions in hexadecimal format which is not very intuitive. This problem can be overcome by writing the instructions in binary first and then using a binary to hex converter which can be easily found via a web search. This issue is outweighed by the speed at which the memory circuit can be populated using this method. Furthermore many test programs will be already written and can be simply loaded into the ROM from file.

Manual pin inputs have also been included that enable users to write instructions into memory, or read from memory, for demonstration and testing purposes.

5.3 – Manual and Automated Testing

Despite the project having a theoretical nature using the Logisim ROM component allows for testing programs to be stored and executed in a similar way to unit tests in software development. Test programs were written to check that every instruction had the correct functionality. This not only involves tests for the 16 different instructions the CPU can perform but for every addressing mode variant of each instruction. Showing all test results would take up too much space, therefore we shall only examine two test programs. The complete set of test programs can be found in table 5.

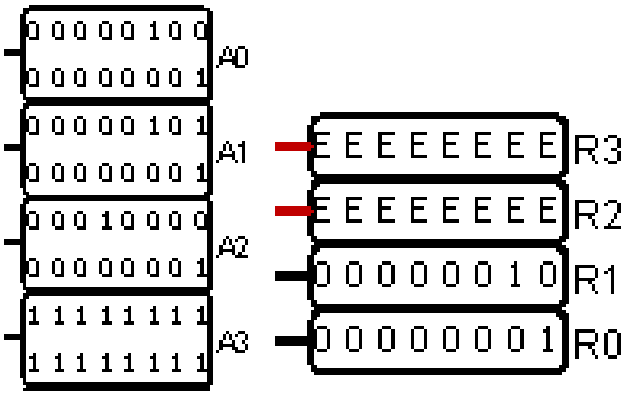


Figure 5.13 Result from the Add00 test program

Figure 5.13 shows the Add00 test program in memory. The program moves the literal value 1 into registers R0 and R1 (memory slot A0 and A1 respectively). An addition operation is performed on the two registers, with R1 as the destination (A2). As we can see after the program has been completed R1 contains the value 2 which shows the operation has been successful and the test has been passed.

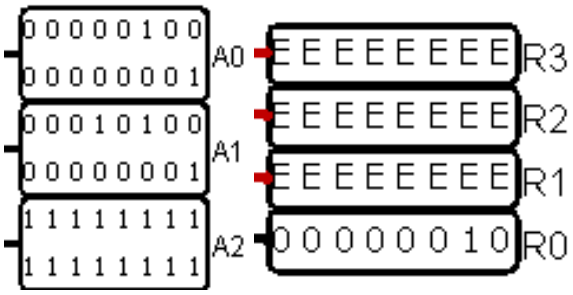
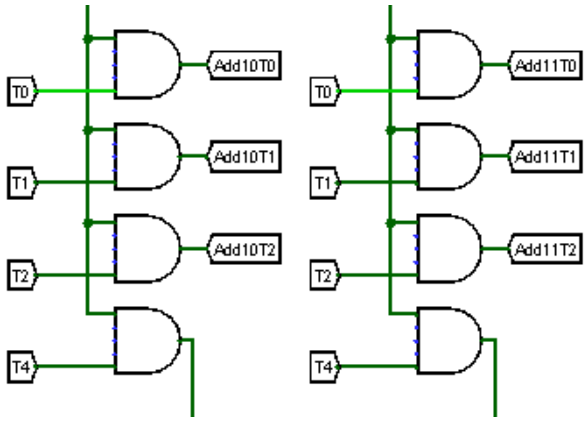


Figure 5.14 Result from the Add01 test program

Figure 5.14 shows the Add01 test program in memory. This program moves the immediate value 1 into register R0 (memory slot A0). An immediate value 1 is then added to the contents of register R0 with the result stored back in to R0 (A1). We can see that R0 contains the value 2 after the program has been executed. Therefore the test has been passed.

The results from this type of testing highlighted a large problem where the addressing mode 10 and 11 variants of instructions were functioning incorrectly. After the instructions were executed the program counter would jump an extra memory address meaning that instructions were being skipped during a program. The issue arose from an incorrect value being clocked into the program counter from the main bus while the operation was being performed. During the next fetch phase this caused instructions to be missed; overall meaning programs would give incorrect results. The solution to this problem was to add an extra clock cycle in which no tristate gates are set, similarly to instructions that are completed in one clock cycle as discussed in section 5.1.5. Having a cycle where no gates are set means that values cannot be accidentally written into the program counter inside the execute to fetch phase transition.

Since all instruction variants with addressing mode 10 and 11 take three clock cycles to complete, naturally, one would assume that adding an extra cycle using time four would fix the issue. However this is not the case as the fetch phase resets on time four so during the transition from fetch to execute the signal for time four is still set because the sequencer is falling edge triggered. If time four is still set at the start of the execute phase the RS flip-flop controlling the fetch-execute cycle is instantly reset, resulting in the instruction never been executed.



The final solution was to implement two extra clock cycles to these variations of instructions so that they are reset on time five (T4) as we can see from figure 5.15. This stops the conflict of incorrect values being loaded and also stops the issue of the instant reset from occurring. With this issue fixed all instructions passed their tests.

Figure 5.15 ADD 10 and 11 instructions with two extra clock cycles

Other tests carried out involved manually testing if instructions were being written into the correct memory locations using manual inputs.

Table 5.1 shows the programs that were used to test each instruction:

Instruction	Test Program	Results	Pass/Fail
Move00	0000 10 00 00000001 0000 00 00 00000001	Move #1 into R0 Move R0 into R1	Pass
Move01	0000 10 00 00000001	Move #1 into R0	Pass
Move10	0000 10 00 00011111	Move contents of address 31 into R0	Pass
Move11	0000 01 00 00000001 0000 11 00 00011111	Move #1 into R0 Move contents of R0 into address 31	Pass
Add00	0000 01 00 00000001 0000 01 01 00000001 0001 00 00 00000001	Move #1 into R0 Move #1 into R1 Add R0 to R1 (result #2 stored in R1)	Pass
Add01	0000 01 00 00000001 0001 01 00 00000001	Move #1 into R0 Add #1 to R0 (result #2 stored in R0)	Pass
Add10	0000 01 00 00000001 0001 10 00 00011111	Move #1 into R0 Add contents of address 31 (#1) to R0 (result #2 stored in R0)	Pass
Add11	0000 01 00 00000001 0001 11 00 00011111	Move #1 into R0 Add R0 to address 31 (#1) (result #2 stored in address 31)	Pass
Sub00	0000 01 00 00000010 0000 01 01 00000010 0010 00 00 00000001	Move #2 into R0 Move #2 into R1 Subtract R0 from R1 (result #0 stored in R1)	Pass
Sub01	0000 01 00 00000010 0010 01 00 00000010	Move #2 into R0 Subtract #2 from R0 (result #0 stored in R0)	Pass
Sub10	0000 01 00 00000010 0010 10 00 00011111	Move #2 into R0 Subtract contents of address 31 (#2) from R0 (result #0 stored in R0)	Pass
Sub11	0000 01 00 00000010 0010 11 00 00011111	Move #2 into R0 Subtract R0 from contents of address 31 (#2) (result #0 stored in address 31)	Pass
Mul00	0000 01 00 00000010 0000 01 01 00000010 0011 00 00 00000001	Move #2 into R0 Move #2 into R1 Multiply R0 by R1 (result #4 stored in R1)	Pass
Mul01	0000 01 00 00000010 0011 01 00 00000010	Move #2 into R0 Multiply R0 by #2 (result #4 stored in R0)	Pass
Mul10	0000 01 00 00000010 0011 10 00 00011111	Move #2 into R0 Multiply contents of address 31 (#2) by R0 (result #4 stored in R0)	Pass

Mul11	0000 01 00 00000010 0011 11 00 00011111	Move #2 into R0 Multiply contents of address 31 (#2) by R0 (result #4 stored in address 31)	Pass
And00	0000 01 00 00000001 0000 01 01 11111111 0100 00 00 00000001	Move 000000001 into R0 Move 11111111 into R1 And R0 with R1 (result 00000001 stored in R1)	Pass
And01	0000 01 00 01111111 0100 01 00 00000001	Move 11111111 into R0 And R0 with 00000001 (result 00000001 stored in R0)	Pass
And10	0000 01 00 11111111 0100 10 00 00011111	Move 111111111 into R0 And contents of address 31 (00000001) with R0 (result 00000001 stored in R0)	Pass
And11	0000 01 00 00000001 0100 11 00 00011111	Move 00000001 into R0 And contents of address 31 (11111111) with R0 (result 00000001 stored in address 31)	Pass
Or00	0000 01 00 00010000 0000 01 01 00000001 0101 00 00 00000001	Move 00010000 into R0 Move 00000001 into R1 Or R0 and R1 (result 00010001 stored in R1)	Pass
Or01	0000 01 00 00010000 0101 01 00 00000001	Move 00010000 into R0 Or 00000001 with R0 (result 00010001 stored in R0)	Pass
Or10	0000 01 00 00010000 0101 10 00 00011111	Move 00010000 into R0 Or contents of address 31 (00000001) with R0 (result 00010001 stored in R0)	Pass
Or11	0000 01 00 00010000 0101 11 00 00011111	Move 00010000 into R0 Or contents of address 31 (00000001) with R0 (result 00010001 stored in address 31)	Pass
Not	0000 01 00 00000000 0110 00 00 00000000	Move 00000000 into R0 Not contents of R0 (result 11111111 stored in R0)	Pass
Xor00	0000 01 00 00000001 0000 01 01 10000001 0111 00 00 00000001	Move 00000001 into R0 Move 10000001 into R1 Xor R0 with R1 (result 10000000 stored in R1)	Pass
Xor01	0000 01 00 00000001 0111 01 00 10000001	Move 00000001 into R0 Xor 10000001 with R0 (result 10000000 stored in R0)	Pass
Xor10	0000 01 00 00000001 0111 10 00 00011111	Move 00000001 into R0 Xor contents of address 31 (10000001) with R0 (result 10000000 stored in R0)	Pass
Xor11	0000 01 00 00000001 0111 11 00 00011111	Move 00000001 into R0 Xor contents of address 31 (10000001) with R0 (result 10000000 stored in address 31)	Pass
Beq	0000 01 01 00000001 0000 01 00 00000001 0010 01 00 00000001 1000 00 00 00011110 At address 30 – 1111 00 01 00000000	Move #1 into R1 Move #1 into R0 Sub #1 from R0 (result R0 = #0) Branch on equal to address 30 (branch taken as Zero flag is true) Trap R1 to screen (result screen displays 001)	Pass
Bne	0000 01 01 00000001 0000 01 00 00000001 0001 01 00 00000001 1001 00 00 00011110 At address 30 – 1111 00 01 00000000	Move #1 into R1 Move #1 into R0 Add #1 to R0 (result R0 = #2) Branch on not equal to address 30 (branch taken as Zero flag is false) Trap R1 to screen (result screen displays 001)	Pass
Bhi	0000 01 00 00000010 0000 01 01 00000001 0010 00 01 00000000 1010 00 00 00011110	Move #2 into R0 Move #1 into R1 Sub R1 from R0 (result 1 stored in R0) Branch on higher to address 30 (branch taken as R0 (#2)	Pass

	At address 30 – 1111 00 01 00000000	was higher than R1 (#1) so Carry and Zero flag both false Trap R1 to screen (result screen displays 001)	
Bcs/Blo	0000 01 00 00000001 0000 01 01 00000010 0010 00 01 00000000 1011 00 00 00011110 At address 30 – 1111 00 01 00000000	Move #1 into R0 Move #2 into R1 Sub R1 from R0 Branch on carry set to address 30 (branch taken as R0 is lower than R1 so Carry flag is true) Trap R1 to screen (result screen displays 002)	Pass
Bcc/Bhs	0000 01 00 00000001 0000 01 01 00000010 0010 00 00 00000001 1100 00 00 00011110 At address 30 – 1111 00 01 00000000	Move #1 into R0 Move #2 into R1 Sub R0 from R1 Branch on carry clear to address 30 (branch taken as R1 is higher than R0 so Carry flag is false) Trap R1 to screen (result screen displays 001)	Pass
Bls	0000 01 00 00000001 0000 01 01 00000010 0010 00 01 00000000 1101 00 00 00011110 At address 30 – 1111 00 01 00000000	Move #1 into R0 Move #2 into R1 Sub R1 from R0 Branch on lower than or same to address 30 (branch taken as R0 is lower than R1 so Carry flag is true) Trap R1 to screen (result screen displays 002)	Pass
Jump	0000 01 01 00000010 1110 00 00 00011110 At address 30 – 1111 00 01 00000000	Move #2 into R1 Jump to address 30 Trap R1 to screen (result screen displays 002)	Pass
Trap00	0000 01 00 00000001 1111 00 00 00000000	Move #1 into R0 Trap R0 to screen (result screen displays 001)	Pass
Trap01	0000 01 00 00000001 0000 11 00 00011111 1111 01 00 00011111	Move #1 into R0 Move R0 into address 31 (address 31 contains #1) Trap address 31 to screen (result screen displays 001)	Pass

Table 5.1 Table of test programs

Testing to see if individual instructions work correctly is important but it is just as important to test whether instructions can interact with each other to perform programs. To test this, programs that were written in the design chapter for calculating the n^{th} number in the Fibonacci sequence and calculating n factorial were used.

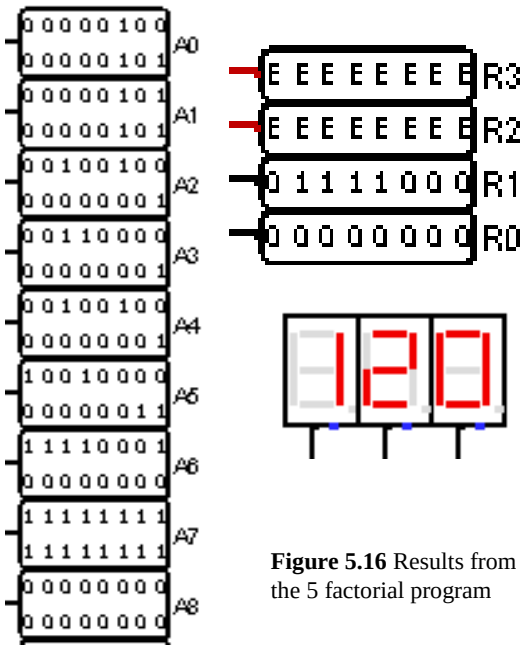


Figure 5.16 Results from the 5 factorial program

Figure 5.16 shows the n factorial program in memory which was defined in section 4.1.4 of the design chapter; with n being replaced by 5. We can see the result displayed as binary in the registers and as a decimal trapped to the CPU's screen. As we can see the result of the operation is correct. A similar test was also carried out using the Fibonacci program from the design chapter and this also passed. These tests show the CPU's instructions interact correctly with each other in order to perform programs written by users.

Chapter 6: Results and Discussion

The result of the project is that CPU has been successfully implemented in Logisim and meets the requirements. The majority of the CPU has been constructed from elementary gates and it can run programs based on a custom made assembly language. Every operation the CPU can execute has passed the tests and work as expected. All of the circuits can be found in appendix B as they are too large to show in this chapter. Therefore in this chapter we shall evaluate how well the CPU built matches the requirements specification in chapter 3 and discuss future improvements that could be made to the project.

6.1 – Meeting Requirements

In this section we will recall the requirements specified in chapter 3 and evaluate which have been met and where the system falls short.

6.1.1 – Software and Design Requirements Evaluation

Requirement	Description	Priority	Requirement met?
Software	The CPU should be designed using the circuit design software Logisim	M	Met
Complexity	The majority of the CPU components should be made out of elementary logic gates such as AND, OR, NOT and XOR.	M	Met
Testing	A way of testing if the CPU is working correctly needs to be established	M	Met
Memory	The CPU should contain a form of basic memory	M	Met
Registers	The CPU should contain a number of data registers to store values	M	Met
Addressing modes	The CPU should support at least two different addressing modes	M	Met

The CPU was created in Logisim as specified by the project outline. It is portable so can run on any machine that has Logisim installed. Programs can also be saved and transferred between machines.

Every component inside the CPU was constructed from elementary gates, however not all components could be integrated in the final CPU circuit due to Logisim oscillation errors or initialisation errors. As explained in the implementation chapter these components were replaced with their Logisim equivalent. These components were made from elementary gates and are still included in the files to show users how they are constructed, even though they were not used. Other Logisim components were used to enable quick population of memory and to create a basic user interface; these features are not in the scope of this requirement, as they are additional features, so therefore the requirement has been met.

To establish a method to test the CPU, the Logisim ROM component was employed to allow users to create, save and load their own programs based on the custom made architecture. This was used to test that all the instructions in the CPU work as expected. Results from operations can be displayed in binary or as a decimal using the Logisim screens.

The CPU contains a 32-address memory unit component which is capable of storing 32 individual 16-bit words. These could be 16-bit instructions or 8-bit operands used in calculations. The CPU contains four general purpose data registers, contained in a register file circuit, that can be used to store 8-bit values during a calculation.

The instruction set architecture allows for four different types of address interactions: register to register, immediate to register, memory to register and register to memory. This appears as though the CPU has four different addressing modes however strictly speaking register to register, memory to register and register to memory modes are all types of absolute (direct) addressing. Immediate to register mode is immediate addressing, meaning the CPU supports two addressing modes as specified in the requirements.

6.1.2 – Arithmetic and Logical Operations Requirements Evaluation

Requirement	Description	Priority	Requirement met?
Addition operation	Can perform the arithmetic addition operation on two eight bit words	M	Met
Subtraction operation	Can perform the arithmetic subtraction operation on two eight bit words	M	Met
Multiplication operation	Can perform the arithmetic multiplication operation on two eight bit words	M	Met
Division operation	Can perform the arithmetic division operation on two eight bit words	O	Not met
AND operation	Can perform the logical AND operation on two eight bit words	M	Met
OR operation	Can perform the logical OR operation on two eight bit words	M	Met
NOT operation	Can perform the logical NOT operation on an eight bit word	M	Met
XOR operation	Can perform the logical XOR operation on two eight bit words	M	Met
Shift right operation	Can perform the logical shift right operation on an eight bit word	O	Not met
Shift left operation	Can perform the logical shift left operation on an eight bit word	O	Not met
Branch instructions	The CPU should be able to execute conditional and unconditional branch instructions	M	Met

The ALU component contains subcircuits made from elementary gates that can perform addition, subtraction and multiplication on 8-bit words. It can also perform the logical operations AND, OR, NOT and XOR simply using logic gates.

The instruction set does not contain any room for any additional instructions. This means specific division and shift operations could not be implemented. This could be a piece of further work that could be undertaken where, if an extra op-code bit was added, components to execute division and shift operations could be put into the ALU. Despite the ALU not containing specific components to perform division or shift operations, overall the CPU can execute these operations in the form of programs. For example division can be performed by a series of subtractions. Shift operations can be performed by multiplying or dividing by 2^N where N is the number of shifts. These requirements were also deemed optional due to their complexity so partially meeting these requirements exceeds expectations.

The instruction set contains the `JUMP` operation which is an unconditional branch and a series of unsigned conditional branches that vary using the zero and carry flags. However the instruction set does not contain a direct `CMP` operation, as the Motorola 68k does, to compare two values and change status flags accordingly in order to make branch instructions such as branch on higher/ lower possible. In order to perform such branches correctly the subtraction operation can be used as `CMP` is simply

subtraction with the result not being stored; meaning the CPU can execute both conditional and unconditional branch instructions.

6.2 – Future Development

The CPU has been successfully implemented and matches almost all of the requirements. This is a great achievement as almost all circuitry was constructed from the basic elementary gates. With this in mind the CPU could be used as the basis for a number of follow up projects. Areas that could be developed further will be discussed in this section.

Firstly the instruction set architecture could be improved by increasing the length of instructions from 16-bit to possibly 32-bit. Making this change would allow for more bits being designated for each section defined in the current architecture. For example using more than four bits for the op-code would allow for more instructions to be coded, possibly instructions such as divide, shifts or compare could be added to increase the CPU's functionality. Leading on from more op-codes; the extra operations could be implemented into the CPU in the form of sub-circuits inside the ALU so that division can be done in one stage without writing a program to perform multiple subtractions.

The CPU could be developed to incorporate indirect addressing where instructions contain address pointers to the location of the operand to be used in the instruction. This would complement an increased instruction size because currently all of the designated addressing mode bits are used, adding another addressing mode requires more bits in the instruction.

Increasing the instruction size could also make the CPU more powerful by increasing the number of bits to encode registers; currently two bits are used for the register field which only allows for the CPU to contain four data registers. Increasing the number of bits used to encode registers means more registers can be added to the CPU. Adding more registers can increase functionality in programs by providing more places to store variables.

The size of memory could also be increased from 32-address as more bits could be used to encode memory addresses. It is worth noting, however, that currently the limitations in memory size are due to Logisim loading times and not the instruction length. Theoretically memory addresses could be 8-bit in the current implementation allowing for a 256-address unit.

The final benefit of having larger instructions is that literal operands can also be larger. Currently operands have a maximum value of 255. Along with this, the ALU could also be changed to perform operations on say 16-bit words as opposed to the current 8-bit; allowing for larger numbers in operations leading to increased functionality.

The design of the CPU could be altered to allow for extra buses. Currently the CPU has a main system bus and a dedicated register bus which is relatively simple. Adding extra buses, say from memory directly to the ALU, would allow for increased functionality as it would free up the main bus for other data during the execution of some instructions, saving clock cycles and execution time overall.

The architecture of the CPU could be altered to allow for signed representation of numbers. Currently the CPU supports unsigned binary integers but allowing for a signed representation as well would allow more complex programs to be executed. Implementing a decimal number representation could also improve a results precision a calculation.

Logisim offers user interface features such as a keyboard that could be integrated into the system for users to type instructions into memory. There is also a TTY screen that could be added to display ASCII characters output by programs. For example, programs such as 'Hello World' where letters are displayed as currently only numbers can be displayed as part of the CPU's basic user interface.

All of the above changes alter the performance of the CPU and increase its technical functionality. It is also important to consider that while these changes could be made, and would improve the CPU, they could also detract from the purpose of the project; to help educate and aid understanding. Some may argue that the more complex the CPU becomes the further away it moves from meeting its purpose. Currently the CPU is at a stage where there is balance between functionality, and simplicity. This balance allows people to understand how it works, while not being too basic. The future developments could be implemented by Com1006 students after reading this report as additional exercises to aid their understanding of circuitry after they have learnt some of the fundamental design ideas from the current implementation.

Chapter 7: Conclusions

The aim of the project was to demonstrate how every CPU, no matter how complex, can be constructed from a few basic elements. This goal has been successfully achieved and the results can be seen from the circuit diagrams in appendix B. The brief outlined in the in chapter 1 states than the CPU should be able to perform basic operations such as addition, subtraction, multiplication, AND, OR, NOT and XOR. Circuits were designed for each of these operations from just basic AND, OR and NOT gates. Originally division was included in the list of required operations but it was felt creating a component to perform division would be too complicated from elementary gates. With more time a division circuit could be implemented from basic gates. Even though this requirement was not met division can be accomplished in the form of a program. In fact, the CPU is Turing complete meaning it can execute any program. Some of the circuits were made from Logisim's prebuilt components. These were highlighted in the implementation chapter and the reasons why custom components couldn't be used were discussed. Although this is somewhat disappointing, as ideally all components should be custom made, only a very small portion of the CPU is not custom made which is a large achievement.

This achievement is made even greater because of the Logisim limitations that had to be overcome in order to implement some of the custom made circuits. Logisim has trouble in handling circuits with loops, such as flip-flops, and also has a bug in which wires carry unknown values, stopping circuits from functioning correctly. At the beginning of the project and in the initial stages of implementation these limitations seemed as though they would have a large impact on the project and would degrade the result. However the knowledge gained by practicing using Logisim and its features allowed the limitations to become less significant as the project progressed. An awareness of Logisim's features was developed over time. Altering some of the settings and design of circuits, as described in the implementation chapter, allowed the majority of these issues to be overcome.

The project outline stated that a basic memory should be implemented. Initially a 64 address memory unit was created but due to Logisim's limitations it was too large to be appropriate for the project. A smaller 32 address unit was created which was much better suited for the project as it improved loading time and also eliminated the Logisim unknown wire value bug. The current memory unit has enough space to store the majority of programs that a user could think of. The prebuilt Logisim ROM unit was included to allow users to save and load in programs from file which is a useful usability feature of the CPU.

The most challenging part of the project was combining all of the individual circuits that make up the CPU such as the memory unit, ALU and registers. The individual circuits were relatively easy to construct on their own but when combined with the control unit circuit caused many errors and conflicts. These errors were overcome by altering the design of the control unit which is arguably the most complex circuit in the CPU. Having only one main system bus for data to travel along meant that setting the correct signals at the correct times was extremely important to avoid conflicts. This meant a lot of time was spent creating the timing signal tables that can be seen in appendix A. Testing the control unit exposed the fragile nature of the CPU and showed how even the tiniest of delays in timing signals can be the difference between a successful implementation and a disastrous one.

A large part of the project was deciding on a suitable instruction set architecture to encode all of the operations and addressing modes necessary for the CPU to meet the desired requirements. The design of the instruction set affects the construction of the CPU itself. It was also difficult to encode all of the data needed using just a 16-bit instruction. The instruction is very compact with the number of bits being exploited fully; there is no wasted space in the instruction. Even though the instruction design is compact it is still reasonably simple and easy to understand making it ideal for this project from an

educational standpoint. It matches the balance between functionality and simplicity that this project strives for.

Deciding on a design for the CPU as a whole was challenging. It is important to remember that even though this project was designed to be simple but also perform well there are many alternative designs that could have been employed. In the design chapter we looked at an initial design that would have worked successfully but required a more complex control unit circuit to control the bus traffic. Another CPU representation could have multiple buses, giving it better performance but a higher complexity. There are countless more designs, each with their own benefits and drawbacks. It was difficult to develop an appropriate design that met the educational aspect of the project. The design chosen, again, strikes a good balance between performance and complexity. It is capable of executing many complex operations while being basic in design.

The CPU works correctly and meets the requirements specified in the initial project description. Numerous improvements could still be made as stated in the previous chapter. As suggested previously, after reading the report it should not be difficult for readers to build on the foundation that is the CPU in its current state. Undertaking such a project would show that the reader has fully understood the material covered in this project and is able to put the knowledge they have acquired to use, verifying the project as an educational success.

The most important thing that was learnt during the project was the knowledge of how to design custom circuits that work effectively. As the project progressed circuit design skills were learnt and with practice meant that circuits could be designed more efficiently. During the implementation stage of the project many circuits were changed after they had been created. Looking back at the circuits after gaining more knowledge allowed the author to see where unnecessary gates could be removed to make the circuits more efficient. Despite its limitations, Logisim has been very useful throughout the project and contains some handy features to aid circuit design. For example using Logisim's tunnel component helped save time in wiring circuits and improved the overall look of the CPU.

It is hoped that by reading this report and spending time learning how the CPU functions will set readers on a path to improve their own knowledge of circuitry and pursue an interest in computer hardware.

References

[1] – Wikipedia, 'Central Processing Unit'. (en.wikipedia.org/wiki/Central_processing_unit)

[2] – Clements, A. (2006). Chapter 7. Principles of computer hardware. Oxford University Press.

[3] – Dr Dirk Sudholt, Department of Computer Science, University of Sheffield, COM1006 Devices and Networks, The structure of the CPU lecture 11

[4] – Wikipedia, 'Accumulator (Computing)'. (en.wikipedia.org/wiki/Accumulator_(computing))

[5] – Clements, A. (2006). Chapter 5. Principles of computer hardware. Oxford University Press.

[6] – Dr Dirk Sudholt, Department of Computer Science, University of Sheffield, COM1006 Devices and Networks, Instruction set architecture lecture 9

[7] – Cburch.com, 'Logisim'. (www.cburch.com/logisim/)

[8] – Logisim: A graphical system for logic circuit design and simulation, Carl Burch, ACM Journal on Educational Resources in Computing (JERIC), Volume 2, Number 1, March, 2002

[9] – Motorola M68000 FAMILY PROGRAMMER'S REFERENCE MANUAL

[10] – Clements, A. (2006). Chapter 6. Principles of computer hardware. Oxford University Press.

[11] – Minnie.tuhs.org. 'No Title'. (http://minnie.tuhs.org/Programs/UcodeCPU/)

[12] – Dr Dirk Sudholt, Department of Computer Science, University of Sheffield, COM1006 Devices and Networks, Implementing Arithmetic lecture 7

[13] – Wallace, C. S. 'A Suggestion For A Fast Multiplier'. IEEE Transactions on Electronic Computers EC-13.1 (1964): 14-17.

[14] – www.pixelhivedesign.com, Pixel. 'Fast 8Bit Signed/Unsigned Multiplier Using Instant Carry Adders Minecraft Project'. Planetminecraft.com. (http://www.planetminecraft.com/project/fast-8bit-signedunsigned-multiplier-using-instant-carry-adders/)

[15] – Clements, A. (2006). Chapter 3. Principles of computer hardware. Oxford University Press.

[16] – Dr Dirk Sudholt, Department of Computer Science, University of Sheffield, COM1006 Devices and Networks, Tutorial sheet 7

[17] – Sourceforge.net, 'Logisim / Bugs / #16 Oscillation In Simple JK Flip Flop'. (http://sourceforge.net/p/circuit/bugs/16/#1f94)

[18] – Www-inst.eecs.berkeley.edu, 'Logisim Reference'. (http://www-inst.eecs.berkeley.edu/~cs61c/su13/proj3/logisim_guide/logisim.html)

[19] – Cburch.com, 'Oscillation Errors'. (http://www.cburch.com/logisim/docs/2.3.0/guide/prop/oscillate.html)

[20] – Haskell, Richard E., and Darrin M. Hanna. "What are the Fundamentals of Digital Design?" Proc. 2005 ASEE North Central Section Conference, Ohio Northern University, Ada, OH. 2005.

[21] – Johnloomis.org, 'Binary To BCD Converter'. (http://www.johnloomis.org/ece314/notes/devices/binary_to_BCD/bin_to_bcd.html)

Appendix A – Timing Signal Tables

Instruction	Operations (RTL)	Control actions																							
		R	W	C_{MAR}	C_{PC}	C_{IR}	C_X	W_S	W_D	R_{SA}	R_{SB}	R_{DA}	R_{DB}	E_{MEM}	E_{IR}	E_{PC}	E_{BR}	E_{RS}	E_{RD}	E_X	A	B	F_2	F_1	F_0
Fetch	Mar ← PC	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	IR ← [Mar]	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	PC +1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MOVE 00	X ← Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
MOVE 01	Ri ← Imm	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
MOVE 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	Ri ← [Mar]	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
MOVE 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	[Mar] ←X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
ADD 00	X ← Ri + Rj	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	0	1
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
ADD 01	X ← Imm + Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
ADD 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [mar] + Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
ADD 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] + Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
	[Mar] ← X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Instruction	Operations (RTL)	Control actions																						
		R	W	C_{MAR}	C_{PC}	C_{IR}	C_X	W_S	W_D	R_{SA}	R_{SB}	R_{DA}	R_{DB}	E_{MEM}	E_{IR}	E_{PC}	E_{BR}	E_{RS}	E_{RD}	E_X	A	B	F_2	F_1	F_0
SUB 00	X ← Rj - Ri	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0	1	0
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
SUB 01	X ← Ri-Imm	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	0	1	0
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
SUB 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← Ri-[Mar]	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
SUB 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar]-Ri	1	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	1	0
	[Mar] ← X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
MUL 00	X ← Ri * Rj	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	1
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
MUL 01	X ← Imm * Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	0	1	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
MUL 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] * Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
MUL 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] * Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	1
	[Mar] ← X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Instruction	Operations (RTL)	Control actions																						
		R	W	C_{MAR}	C_{PC}	C_{IR}	C_X	W_S	W_D	R_{SA}	R_{SB}	R_{DA}	R_{DB}	E_{MEM}	E_{IR}	E_{PC}	E_{BR}	E_{RS}	E_{RD}	E_X	A	B	F_2	F_1	F_0
AND 00	X ← Ri and Rj	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	0	0
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
AND 01	X ← Imm and Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
AND 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] and Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	0
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
AND 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] and Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	0
	[Mar] ← X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
OR 00	X ← Ri or Rj	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	0	1
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
OR 01	X ← Imm or Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	1	0	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
OR 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] or Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
OR 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] or Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	1
	[Mar] ← X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
NOT	X ← not Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0

Instruction	Operations (RTL)	Control actions																						
		R	W	C_{MAR}	C_{PC}	C_{IR}	C_X	W_S	W_D	R_{SA}	R_{SB}	R_{DA}	R_{DB}	E_{MEM}	E_{IR}	E_{PC}	E_{BR}	E_{RS}	E_{RD}	E_X	A	B	F_2	F_1	F_0
XOR 00	X ← Ri xor Rj	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1	1	1
	Rj ← X	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
XOR 01	X ← Imm xor Ri	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1	1	1	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
XOR 10	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] xor Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1	1
	Ri ← X	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
XOR 11	Mar ← IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	X ← [Mar] xor Ri	1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1	1
	[Mar] ← X	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
BEQ	If z=1 then PC ← IR	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
BNE	If z=0 then PC ← IR	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
BGT	If v=n and z=0 then PC ← IR	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
BLT	If v/=n then PC ← IR	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
BGE	If v=n then PC ← IR	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
BLE	If v/=n or z=1 then PC ← IR	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
JUMP	PC ← IR	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
TRAP 00	Trap	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
TRAP 01	Mar ←IR	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	Trap	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Appendix B – Complete Circuit Diagrams

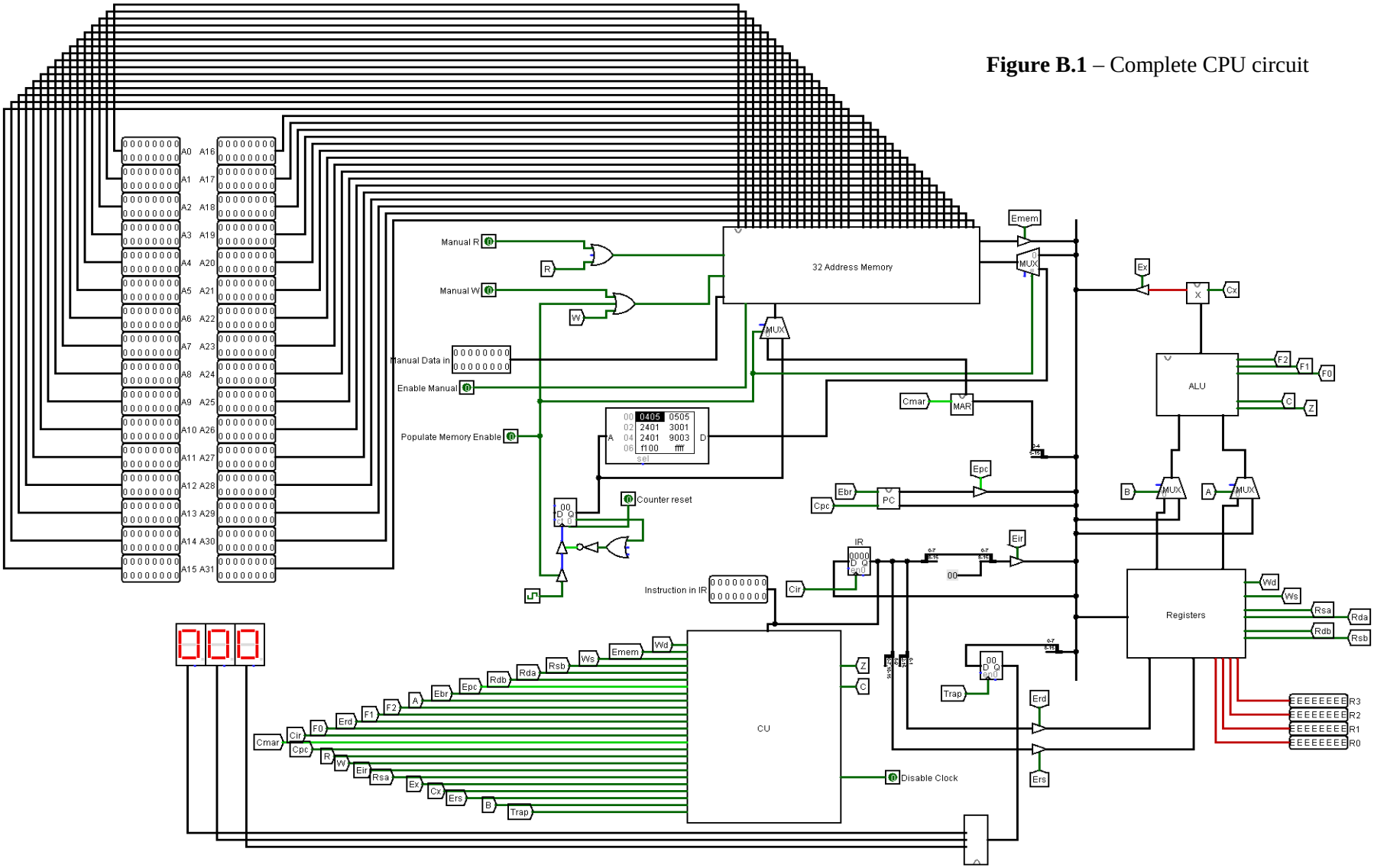


Figure B.1 – Complete CPU circuit

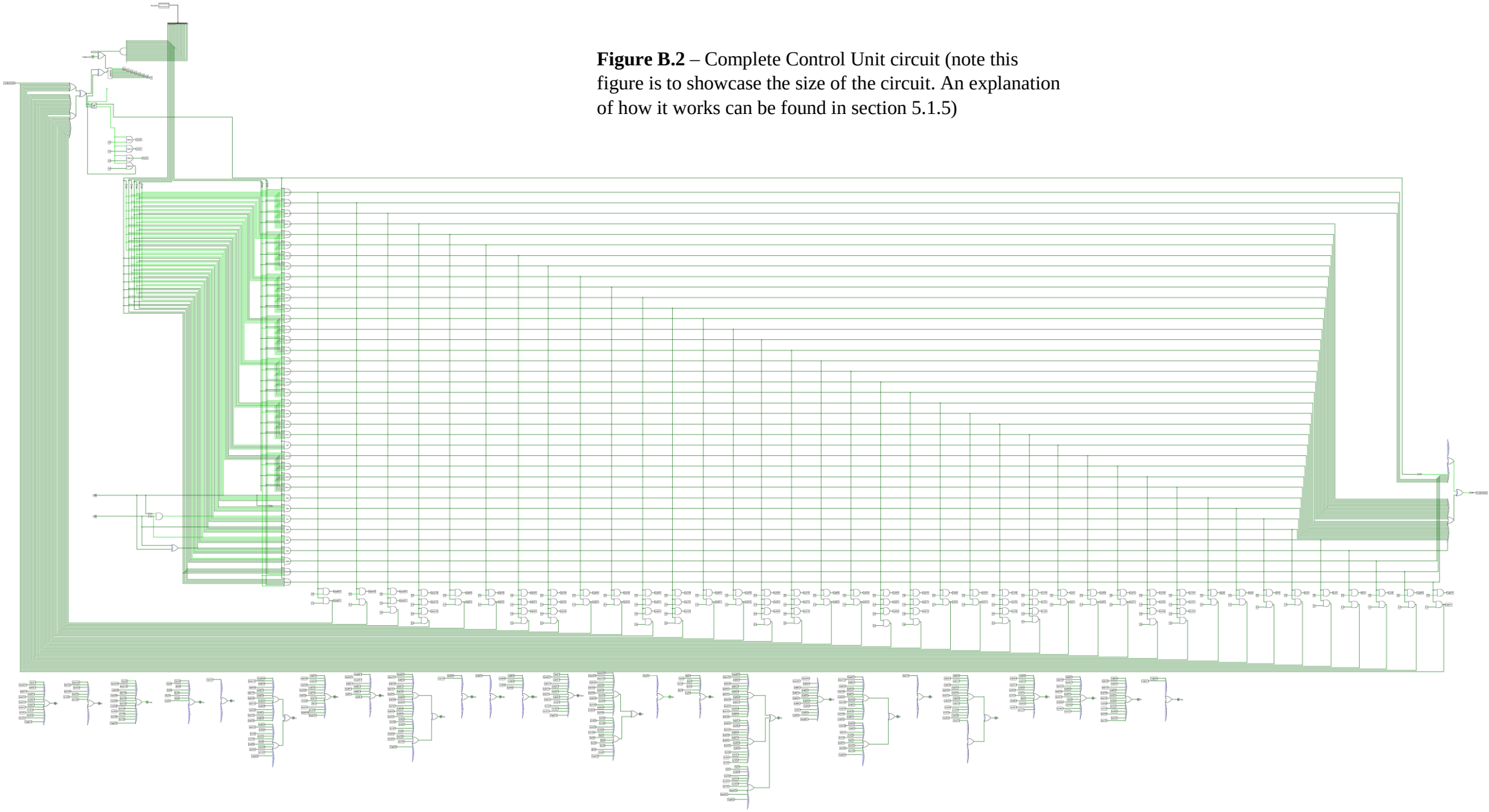


Figure B.2 – Complete Control Unit circuit (note this figure is to showcase the size of the circuit. An explanation of how it works can be found in section 5.1.5)

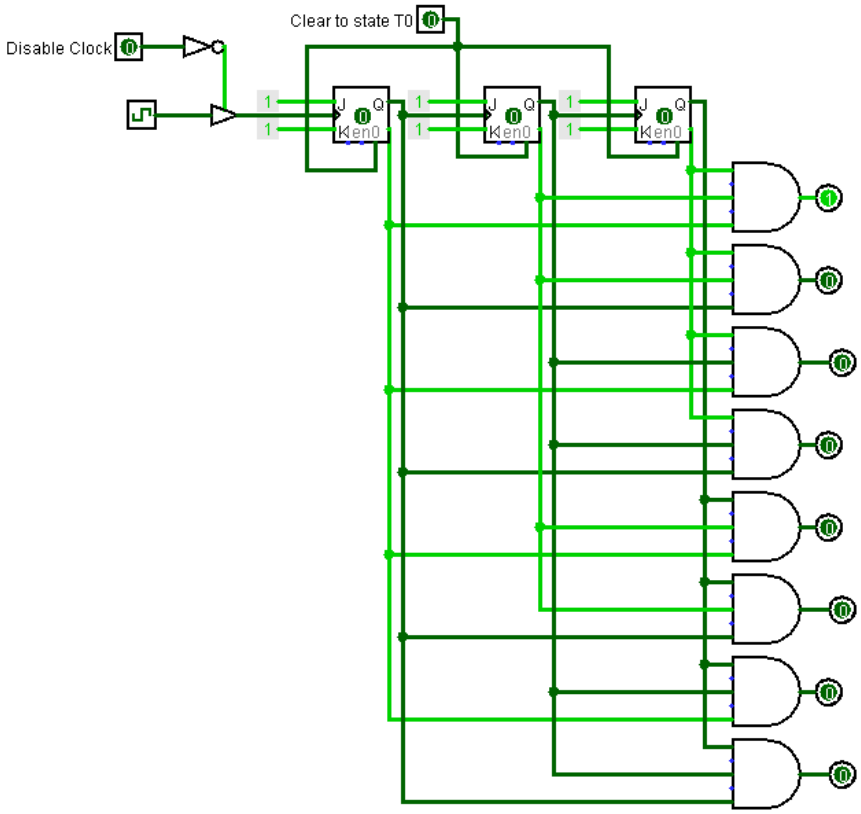


Figure B.3 – Sequencer circuit

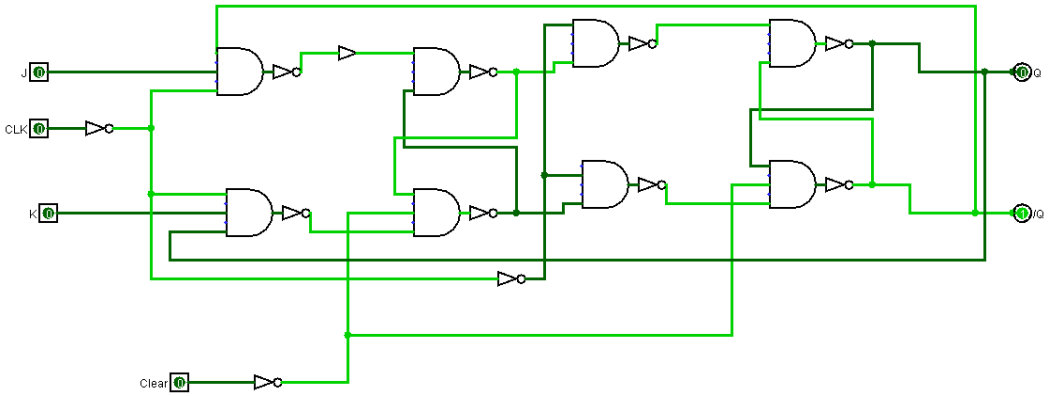


Figure B.4 – Master slave JK flip-flop

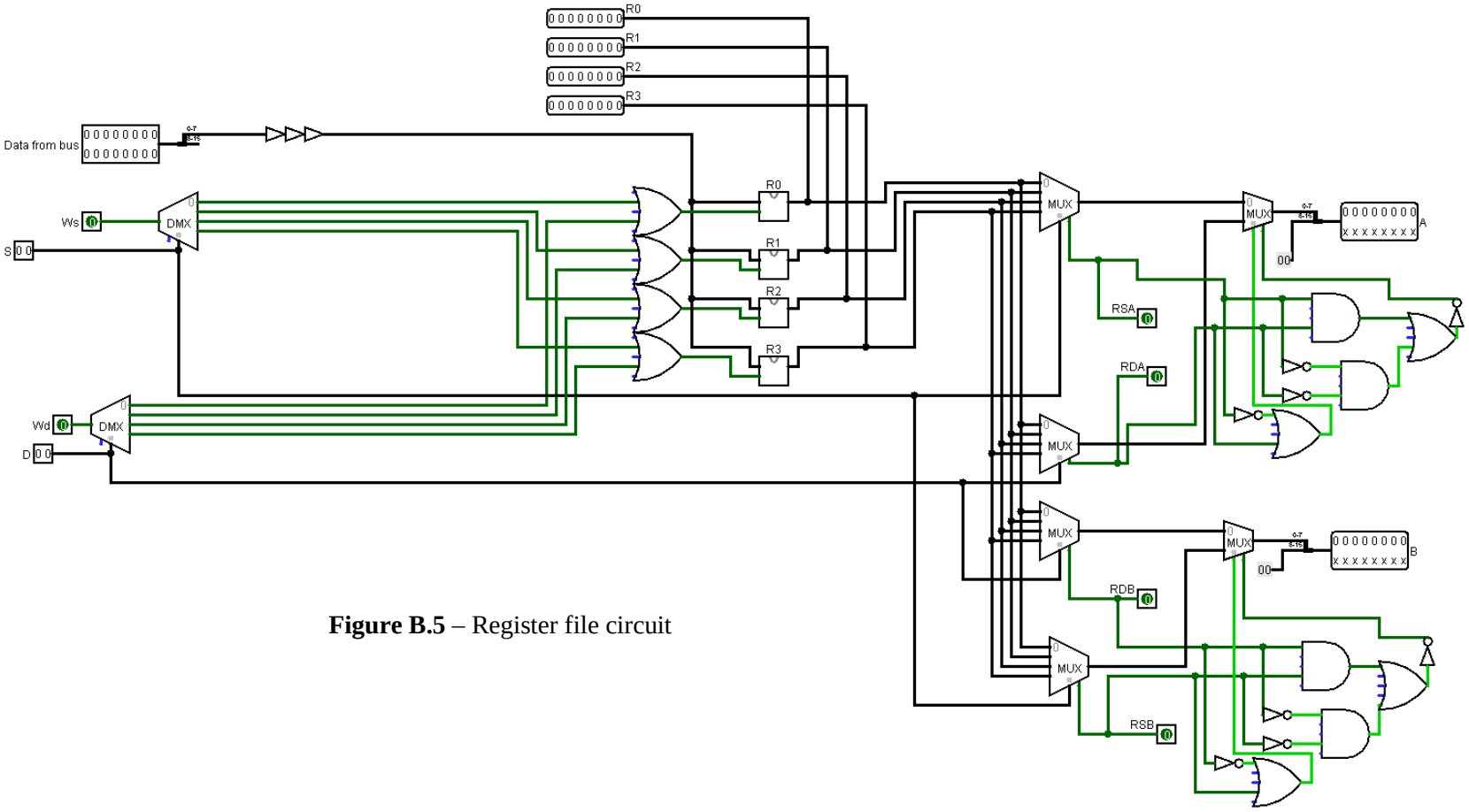


Figure B.5 – Register file circuit

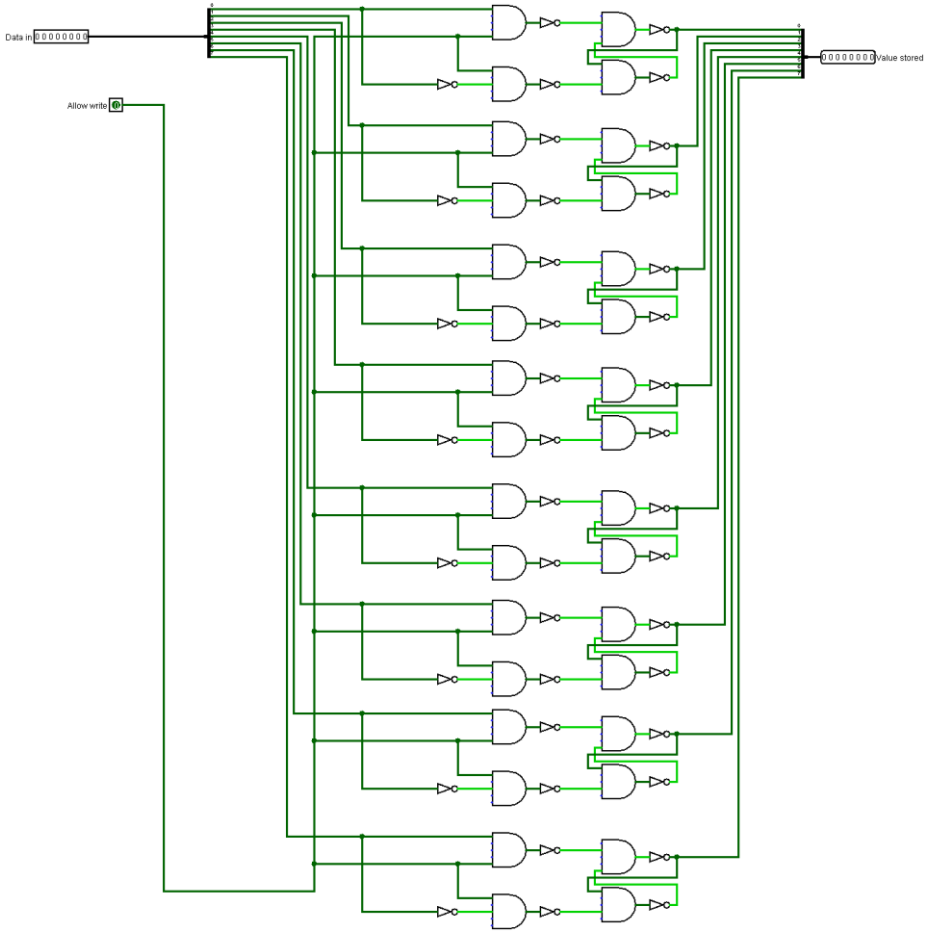


Figure B.6 – 8-bit data register circuit

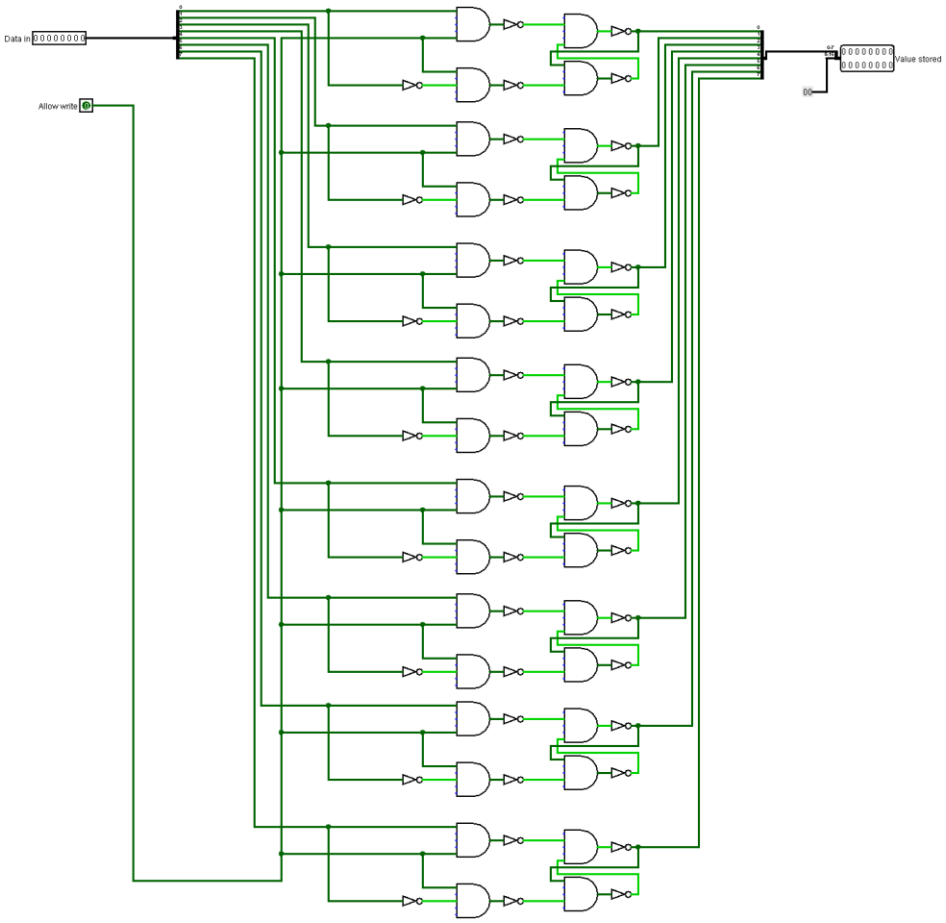


Figure B.7 – Intermediate register X

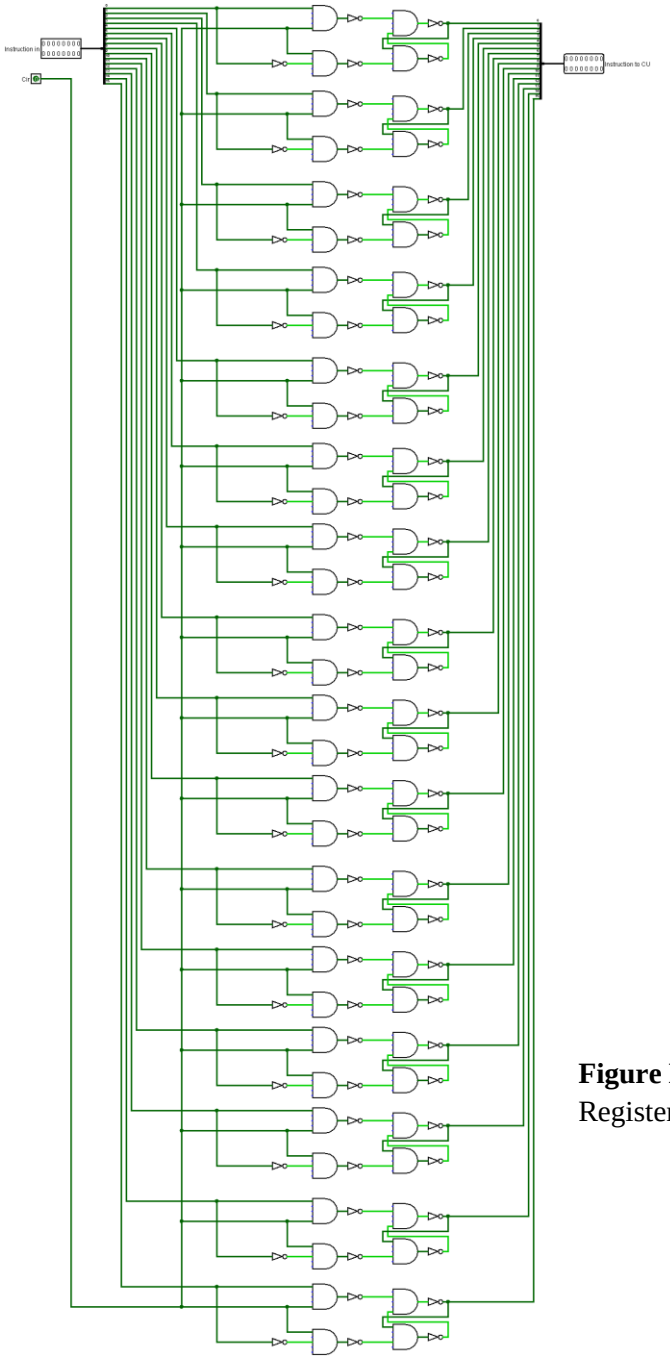


Figure B.8 – Instruction Register (IR)

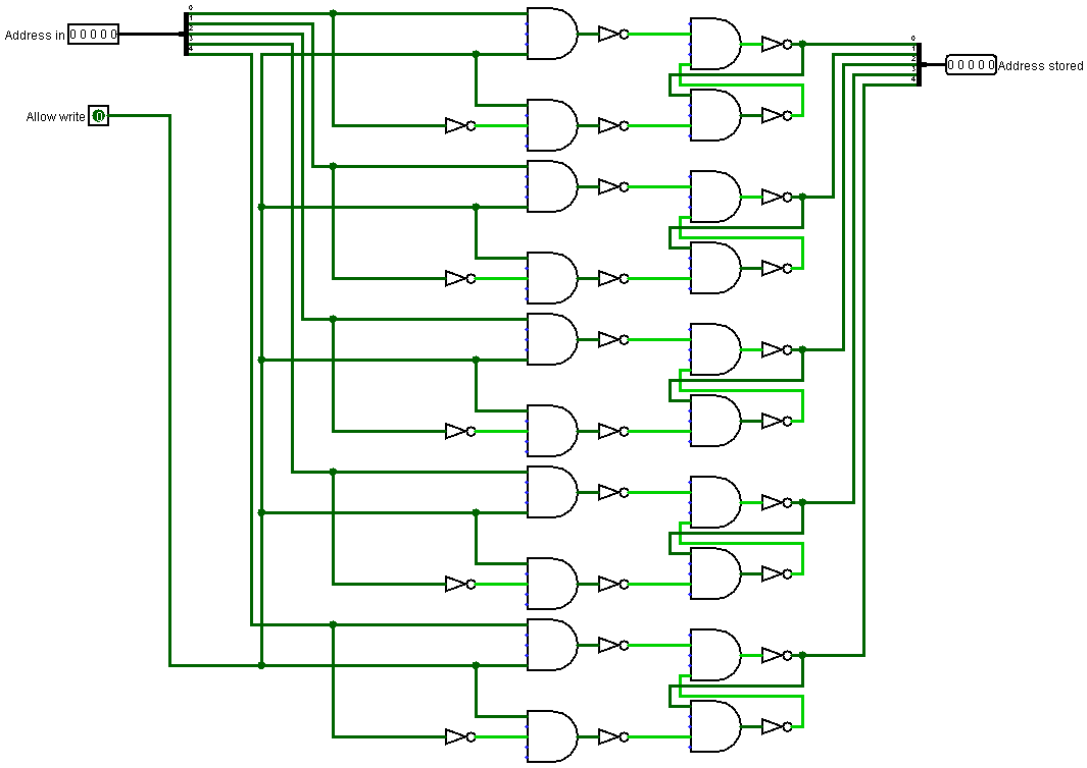


Figure B.9 – Memory Address Register (MAR)

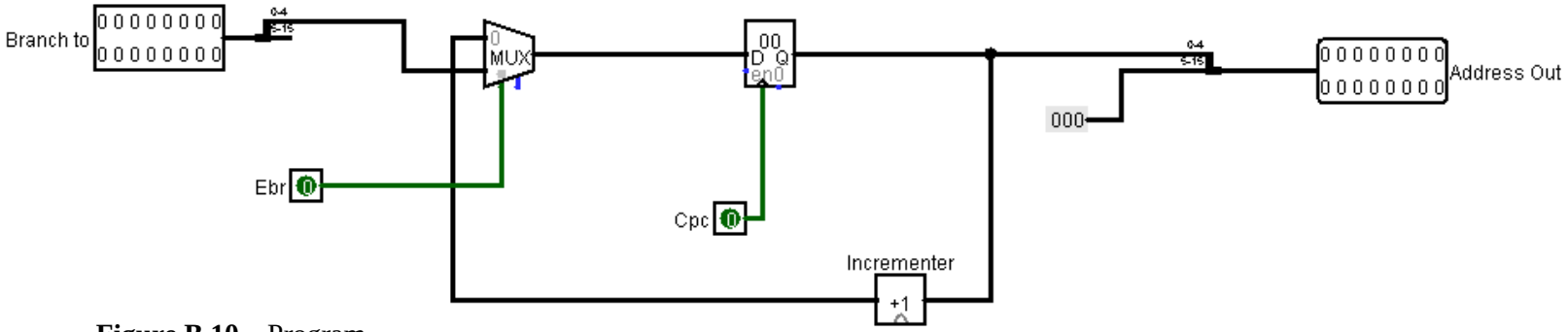


Figure B.10 – Program Counter (PC)

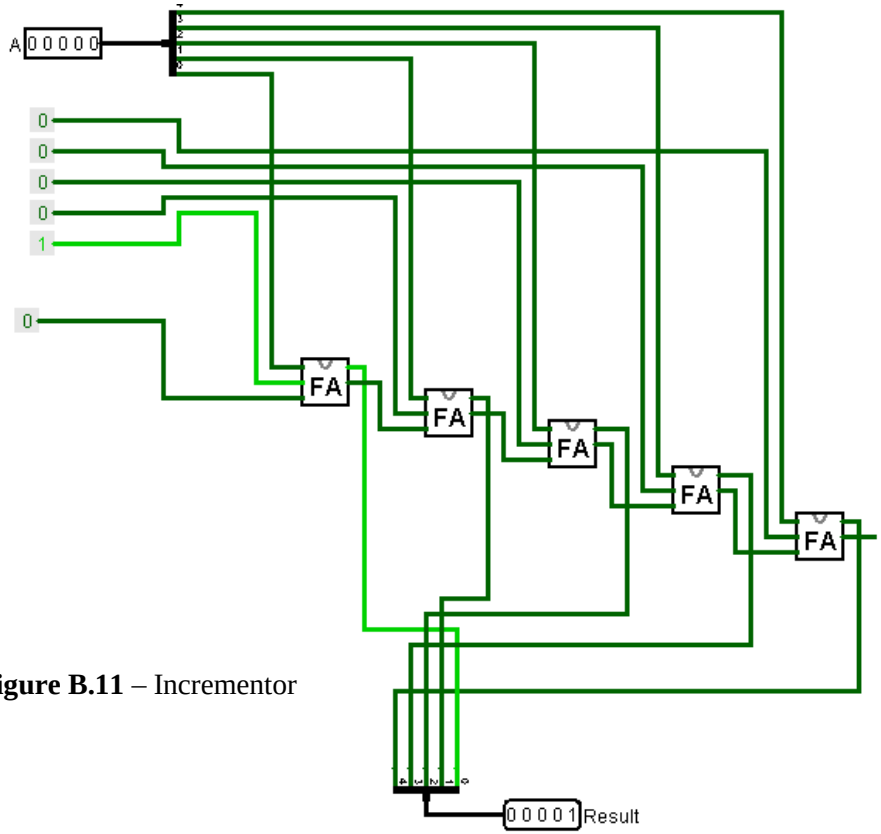
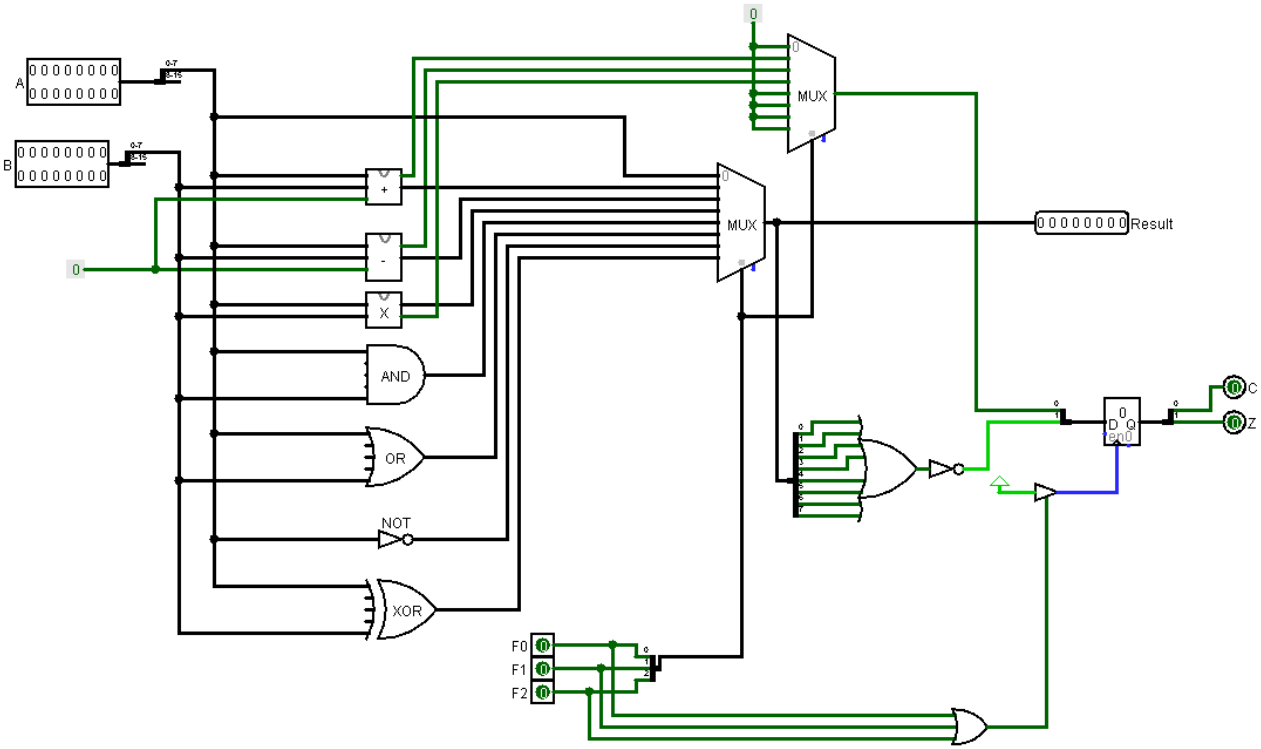


Figure B.11 – Incrementor

Figure B.12 – Arithmetic and logic unit (ALU)



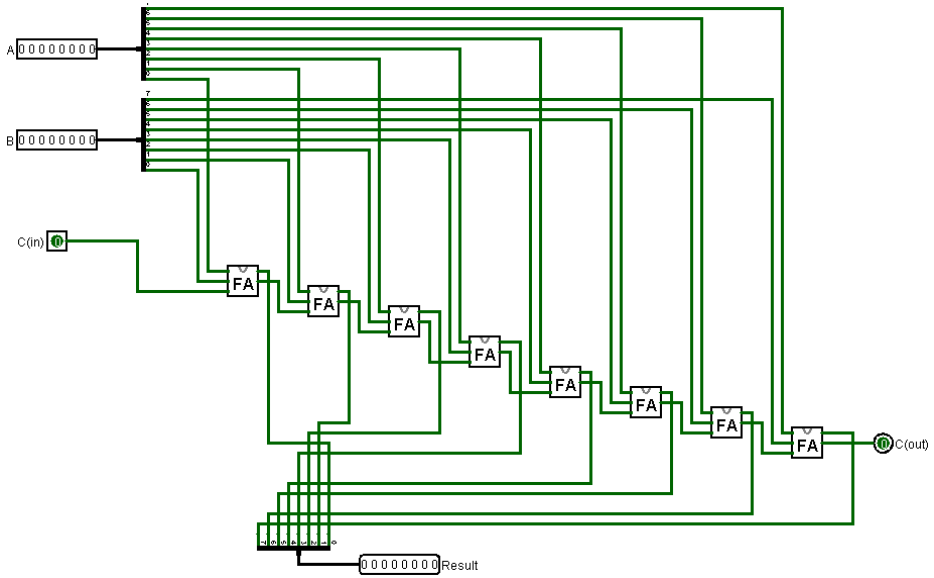


Figure B.13 – 8-bit Adder

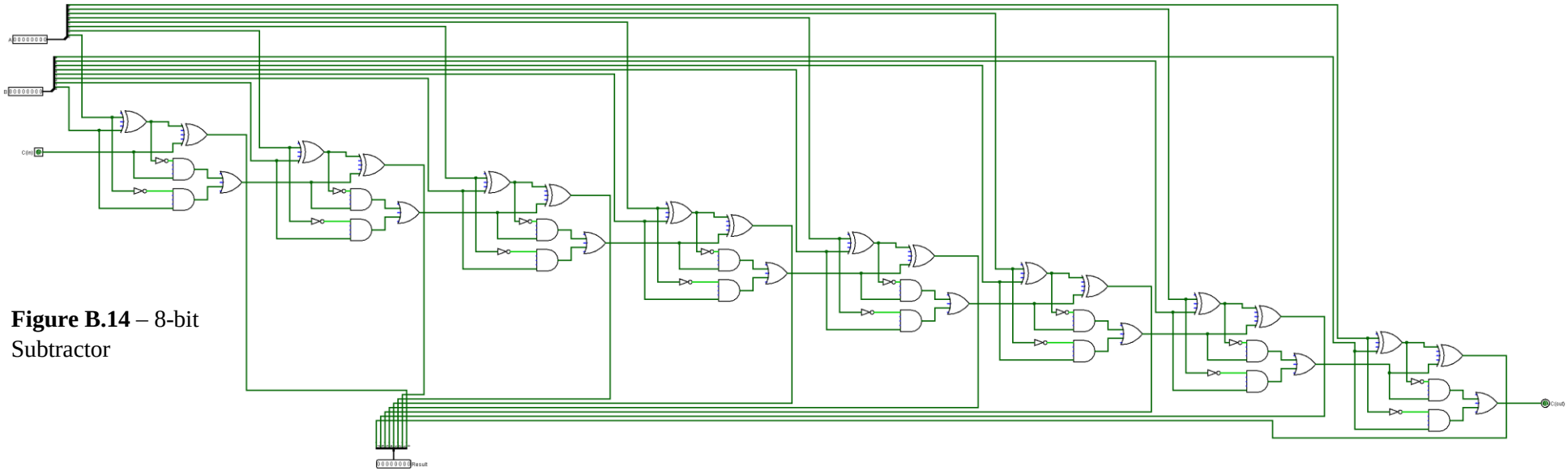


Figure B.14 – 8-bit Subtractor

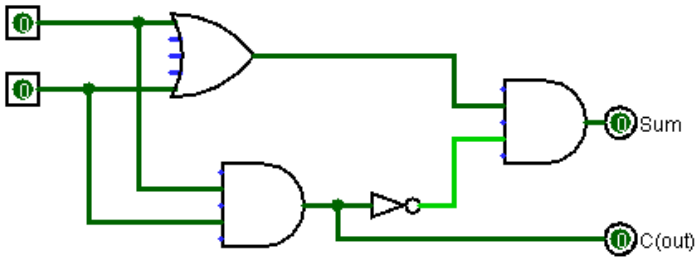


Figure B.15 – Half Adder

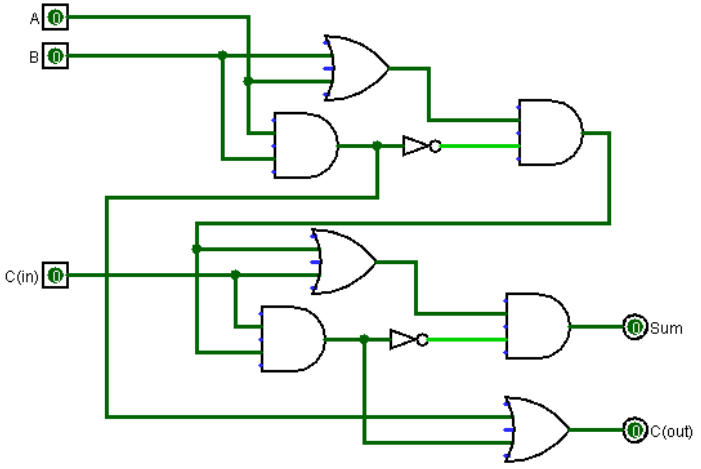


Figure B.16 – Full Adder

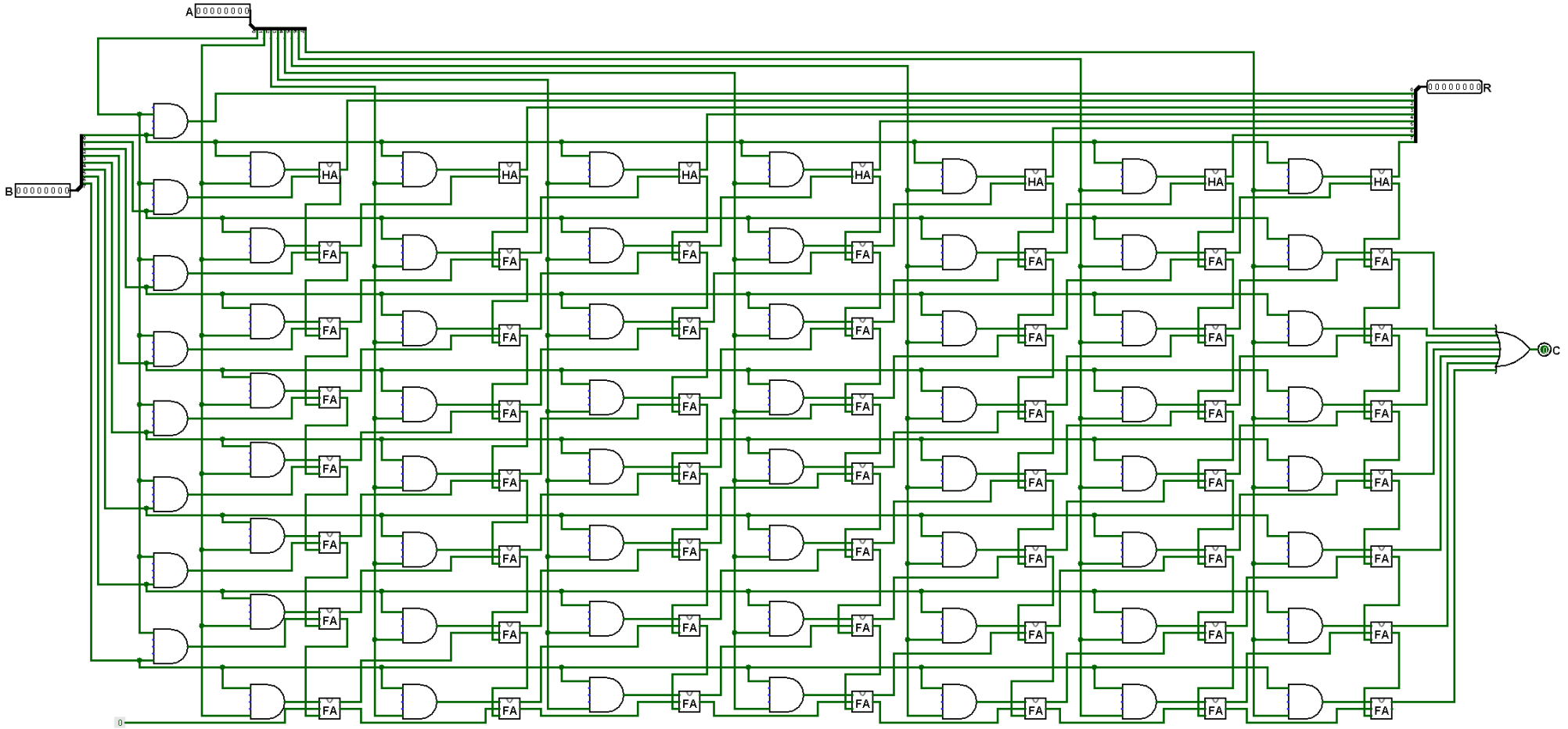


Figure B.17 – 8-bit Multiplier

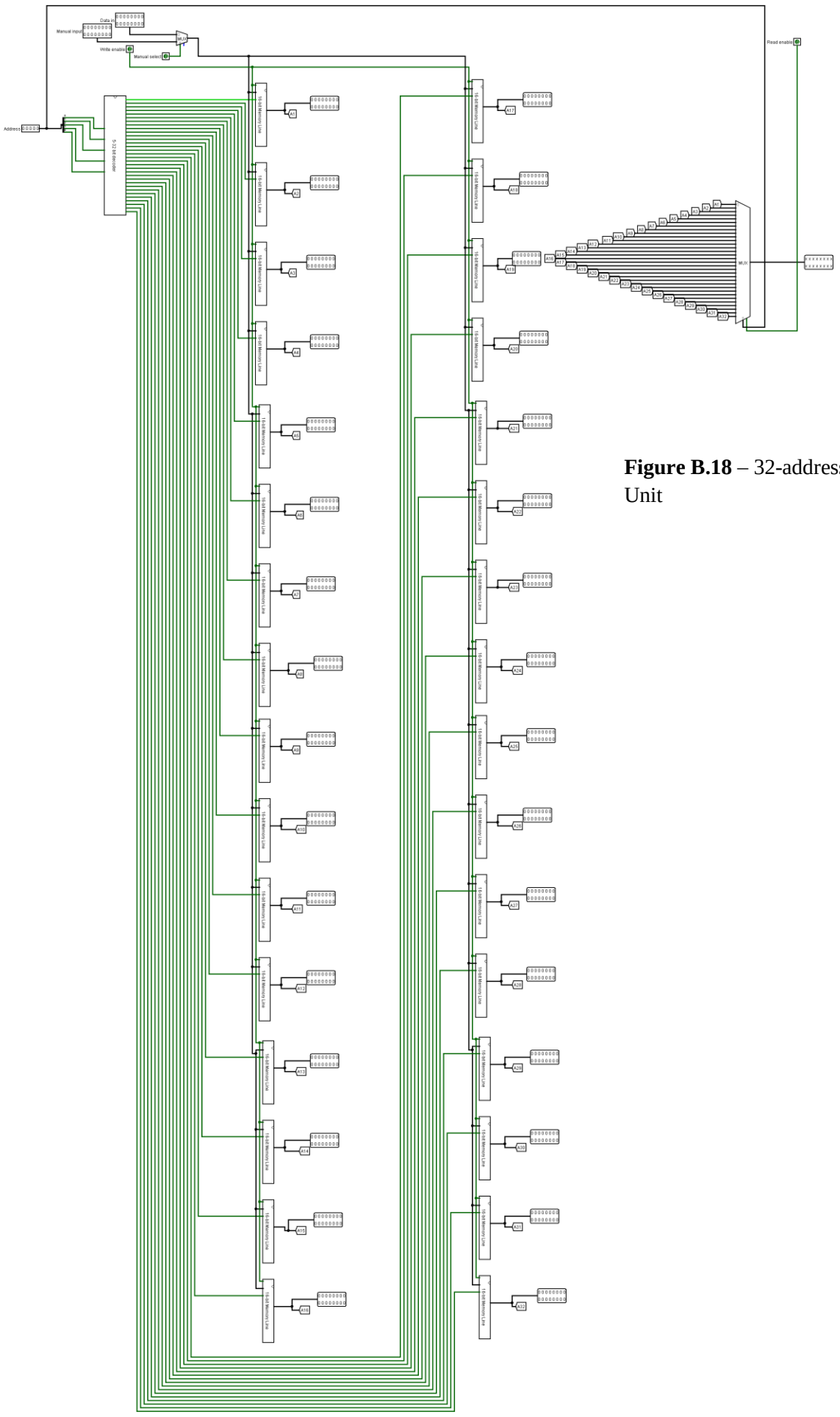


Figure B.18 – 32-address Memory Unit

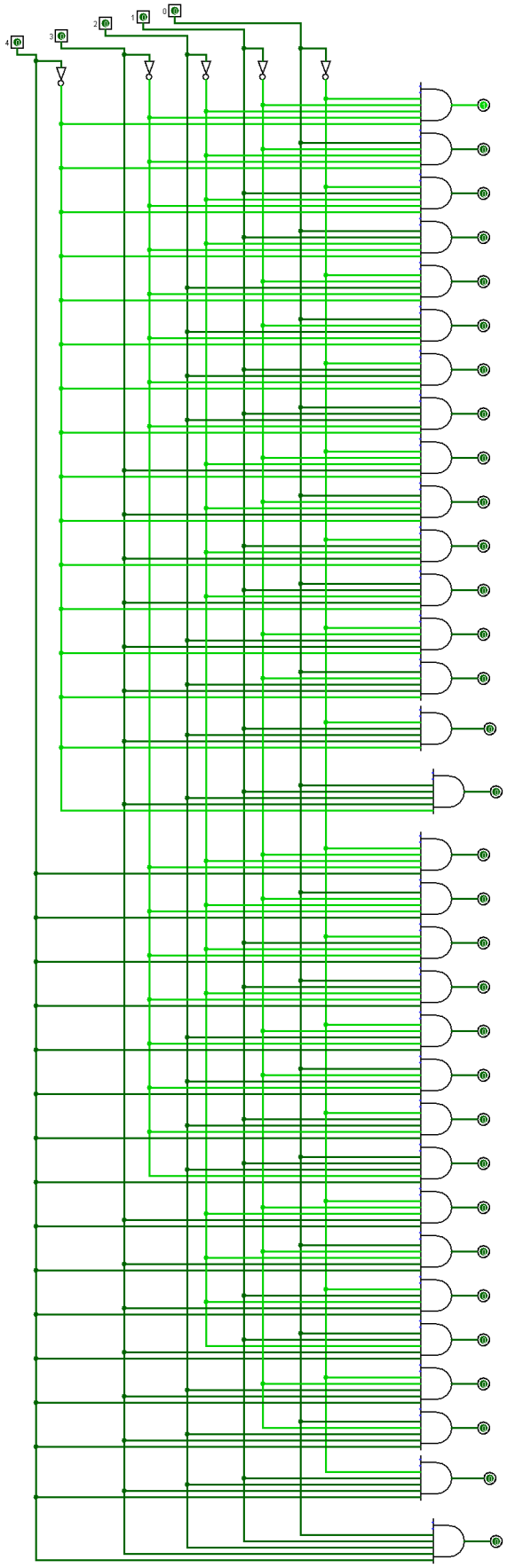
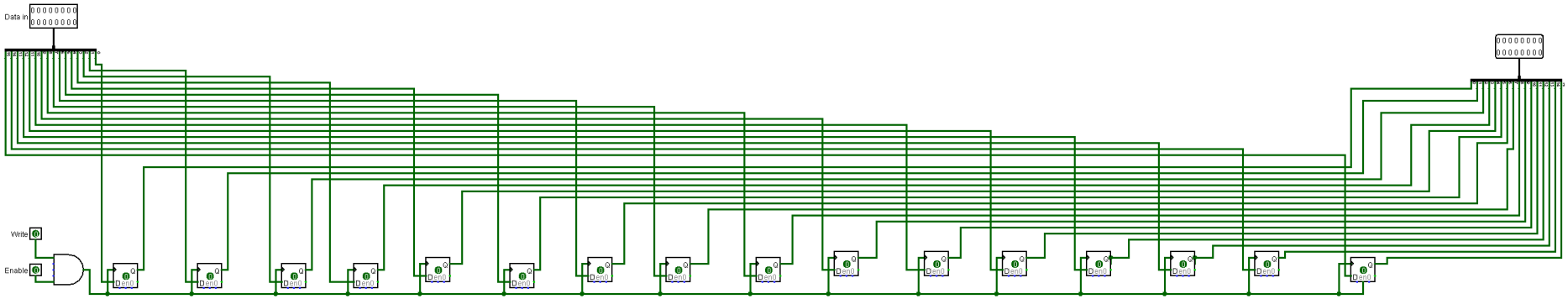


Figure B.19 – 5 to 32-bit Decoder

Figure B.20 – 16-bit Memory Line



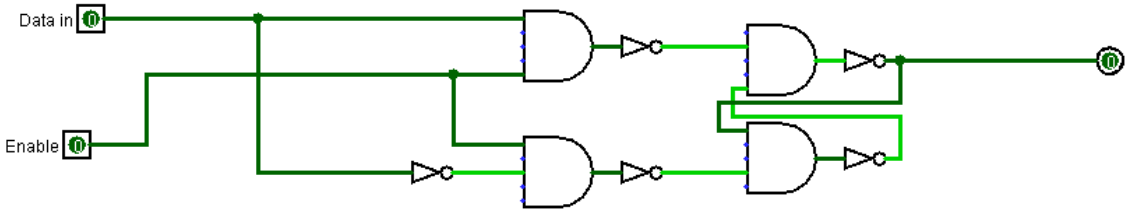


Figure B.21 – D flip-flop

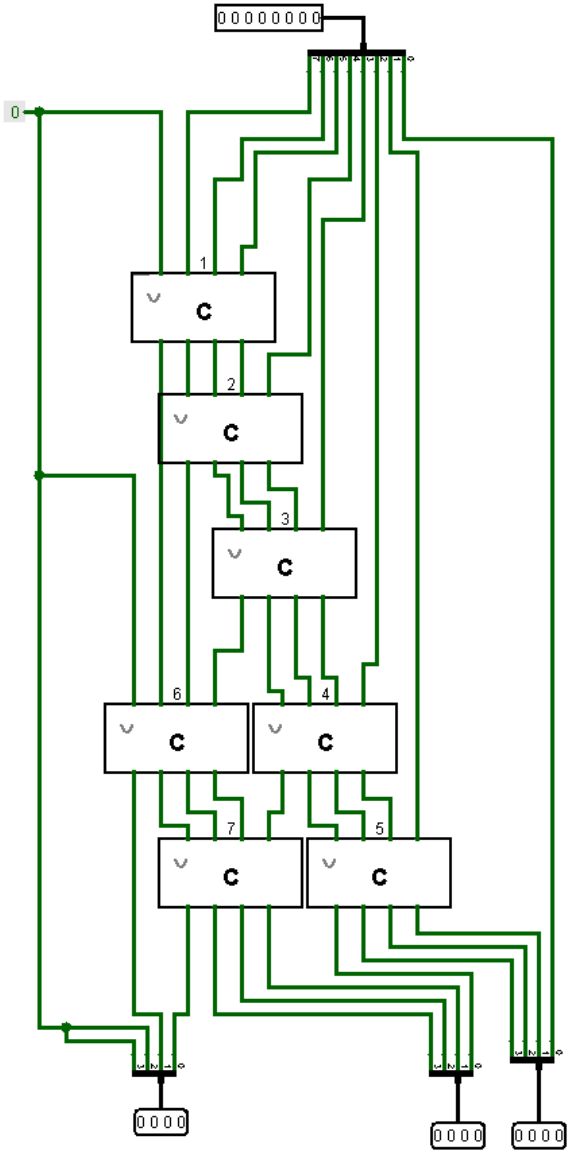


Figure B.22 – Binary to BCD Converter

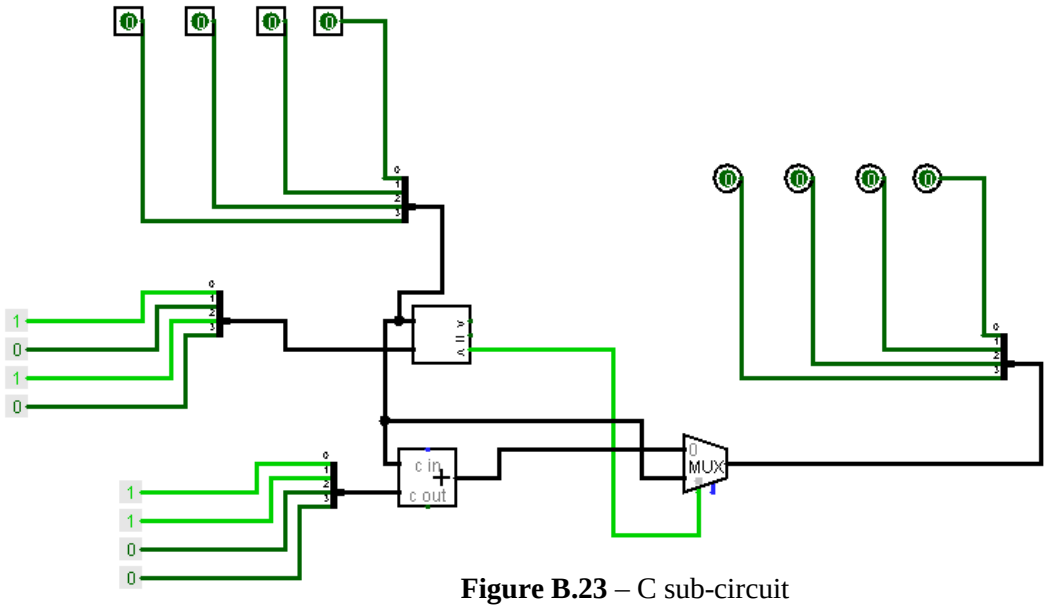


Figure B.23 – C sub-circuit (adds 3 if input is over 5)

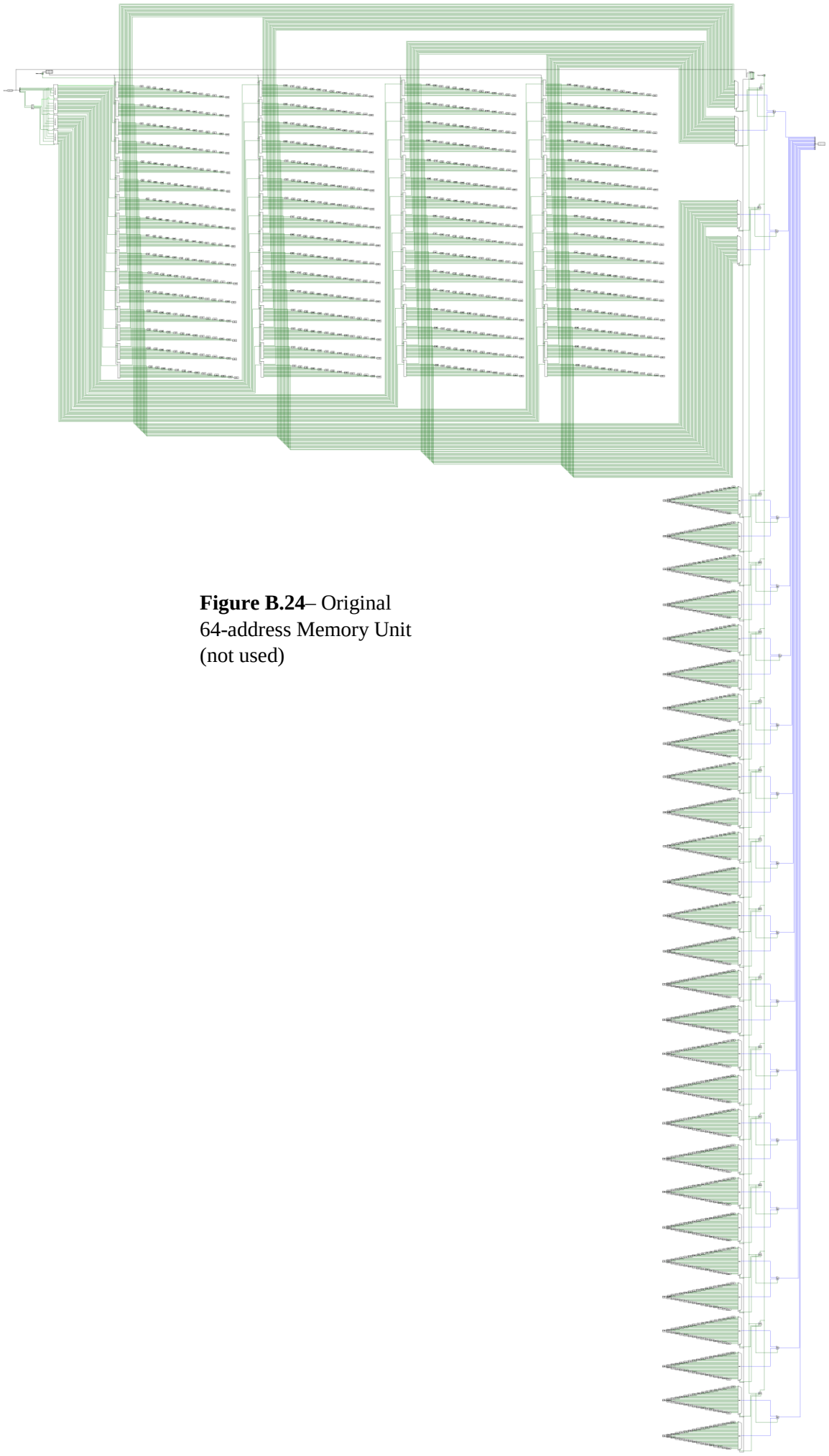


Figure B.24– Original 64-address Memory Unit (not used)