8-Bit CPU Ground Up Construction using Open-Source Circuit Simulation Software

v1.0 by eddiewastaken - 2021

Introduction

TTL (transistor-transistor logic) chips were first developed in the 1960's. The most popular of these types of chips were the 7400 Series, which is a family of logic gates, memory chips, flip-flops, counters and other integrated circuit chips which were used in early electronic CPUs

and PCs. The 7400 series are still used today, usually for breadboard-prototyping and for educational purposes.

Another frequent use of these chips is for building 'Homebrew CPUs'. These are low-level, DIY built CPUs, often limited in their functionality but normally built as an educational project for the individual. An example of a popular Homebrew CPU build is Ben Eater's, which this project aims to build on in an educational sense; by breaking each subsystem, logic unit and TTL chip down to its pure discrete logic implementation.

The full Logisim Evolution project and associated files can be found at my GitHub repository for this project, in case you get stuck.

Assumptions

This write-up tries to remain accessible to as many people and hobbyists as possible, but does make some assumptions about a passing basic knowledge of the following principles:

- Truth tables
- Primitive Logic Gates (AND, OR, XOR, NOT..)
- Binary

If these topics are new to you, some great places to start are:

- https://www.mathsisfun.com/binary-number-system.html (Binary)
- https://www.codeconquest.com/tutorials/binary/ (Binary)
- https://learnabout-electronics.org/Digital/dig21.php (Logic Gates & Truth Tables)
- https://www.elprocus.com/basic-logic-gates-with-truth-tables/ (Logic Gates & Truth Tables)

You'll also need a computer able to execute the Logisim Evolution binaries, ideally a 64-Bit system running Debian, macOS or Windows.

Materials/Methods

Simulating Electronic Circuits

To make the design and build as accessible, efficient and organised as possible, a digital logic simulator was chosen over a physical breadboard build. These simulators often have a built-in library of commonly used logic gates, RAM, ROM and counter chips, I/O components, and transistors amongst others. These components can be 'drag and dropped' into a graphical editing window, which represents the overall circuit or subcircuit. Components can be joined with 'wires', and these overall systems then simulated and interacted with directly inside the software itself. This allows speedy prototyping, editing, debugging, and analysis of circuits, and removes the hindrances of a physical construction using IC chips and wires, such as 'floating voltages' causing an incorrect voltage representation of a binary 1 or 0 along a wire, short circuits across components or loose connections. If desired, the simulated design schematics

can be used as ‘blueprints’ for a physical implementation, at whichever level of abstraction given the IC chips, equipment and components available to the individual.

Software

Various different suites of software exist which facilitate the design, building and simulation of digital electronic circuits. There is no one size fits all solution for any project - each has its benefits, drawbacks and level of expertise required to make full use of its features. Several solutions exist as a web browser application, such as EasyEDA, AutoDesk Circuits and PartSim - amongst others. The decision was made to use an offline executable rather than a web-based solution, to take full advantage of the local computing power available to drive the (host machine) CPU intensive subcircuit builds and simulations required for this project. As the project aims to be used as an educational tool for readers looking to take a deeper dive into digital logic, an educational program which Further Education students would be somewhat familiar with was deemed appropriate. Among the options in this class, Logisim Evolution was the clear front runner - a standalone, open source, actively maintained free to use program with a broad built-in component library - and widely used amongst Further and Higher educational institutions across the world. Features such as VHDL ([Very High Speed Integrated Circuits] Hardware Description Language) which is used to programmatically define and simulate often high level and complex systems and subsystems, and FPGA (Field Programmable Gate Array) board flashing support weren’t required - the project is ideally completely free of any abstraction and aims to build from discrete logic wherever feasibly possible to create a deep understanding of the overall operation of the subcircuits and resulting CPU, and the simulation and I/O itself is handled by the program directly.

Plan/Design/Build

Architecture

There are two main architectures used when designing a CPU, known as ‘Von Neumann’ and ‘Harvard’. The key difference is the Harvard architecture proposes separate memory addresses for instructions and data, while the Von Neumann design uses the same memory for both. The drawback of using separate memory spaces for data and instructions is the complexity of the control unit required for two buses, which increases the amount of subcircuits required in the system. The separate memory spaces allow instruction and data fetches to occur simultaneously, therefore significantly increasing execution speed of instructions - however, as efficiency isn’t a key focus of this project, and the subcircuits are designed to be as barebones as possible, the simpler Von Neumann architecture was deemed appropriate.

The CPU Shared Bus[fig:1]

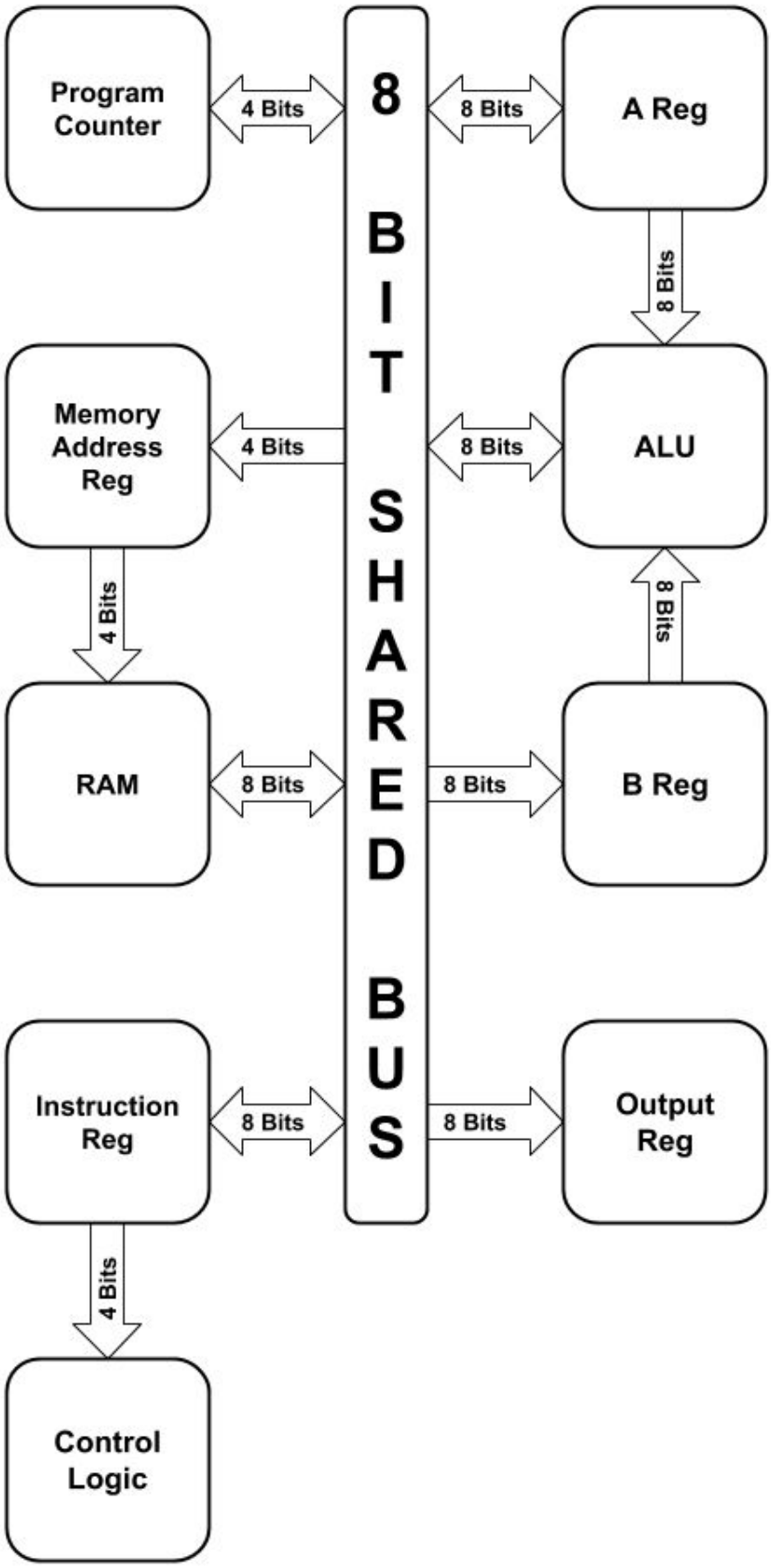

Program Counter
4 Bits
8 BIT SHARED BUS
8 Bits
A Reg
8 Bits
Memory Address Reg
4 Bits
8 Bits
ALU
4 Bits
8 Bits
RAM
8 Bits
8 Bits
B Reg
Instruction Reg
8 Bits
8 Bits
Output Reg
4 Bits
Control Logic

Figure 1 [fig:1] shows the data flow, direction and bit width between the subcircuits of the system, and the shared bus. The upper 4 bits of the bus are used for opcodes (the 'leftmost' bits), and the lower 4 (the 'rightmost' bits) for immediate values/operands. These full 8-bit words are programmed and stored in RAM in the order to be executed.

Registers

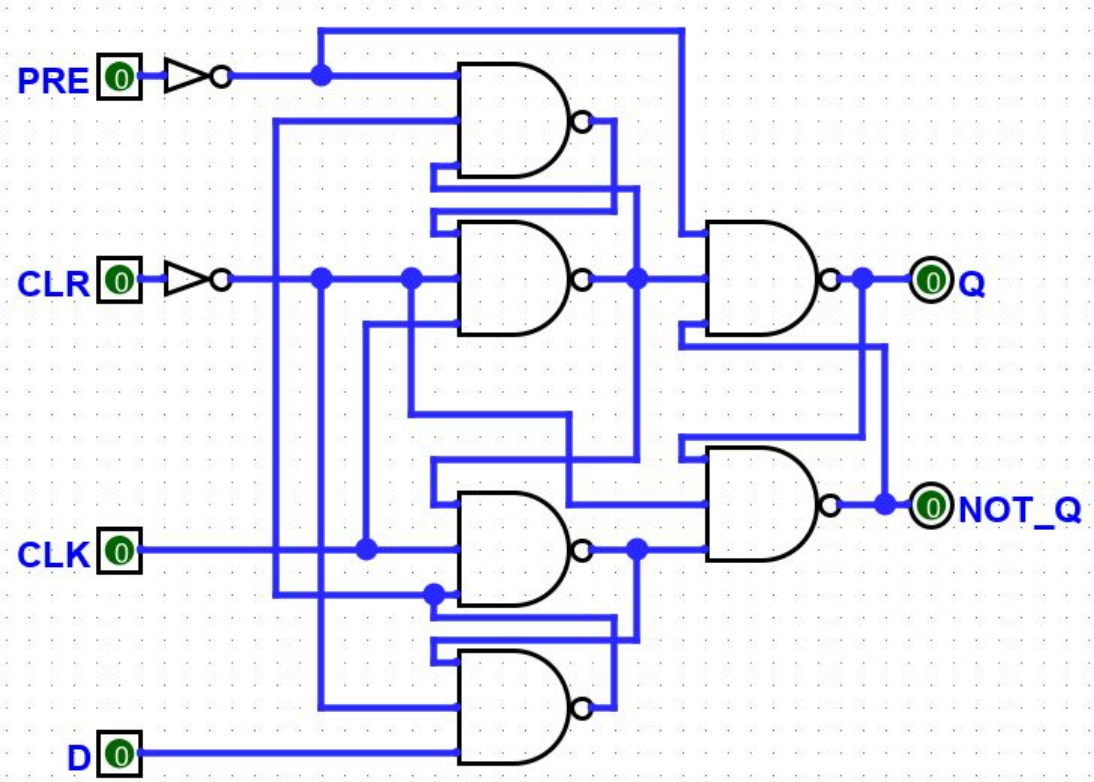
D Type Flip Flop

The D Type Flip Flop is the backbone of the registers in the CPU. Each Flip Flop can store a single bit of data in response to a clock pulse, which makes them perfect as storage for temporary values, such as the A and B registers. For the purposes of the register, the data present at D needs to appear at Q at the positive edge of a clock pulse (at the CLK) input. The required truth table for the operation is shown below, where ↑ represents the rising edge of a clock pulse:

Inputs				***Outputs***	
PRE	**CLR**	**CLK**	**D**	**Q**	**NOT Q**
0	0	↑	1	1	0
0	0	↑	0	0	1
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	1	1

Although the Clear input was needed for the operation of the circuit, specifically to clear register values, the Preset was not. However, following the educational angle of the build, the Preset input was included to mimic the 74LS74 IC - 'Dual D-Type Positive-Edge -Triggered Flip-Flops With Preset And Clear'. The discrete implementation of this circuit using NAND gates is demonstrated in Figure 2 [fig:2], with an added Clear/Reset as described in the truth table.

D Type Flip Flop with Preset and Clear[fig:2]



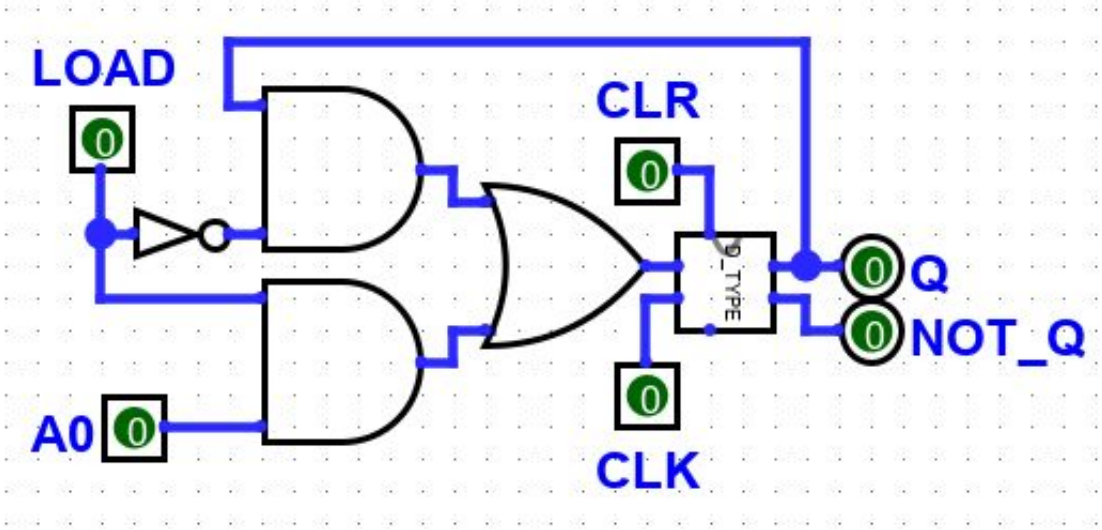
Single Bit Register

From the implementation of the D Type Flip Flop, the next step can be taken to design a 'single bit register'. This adds the functionality of an 'optional store' signal - 'LOAD' in this case. This allows conditional operation of the register, so it can be activated by the control logic as and when required. The target truth table for this is as follows:

Inputs			***Outputs***		
CLR	**CLK**	**LOAD**	**A0**	**Q**	**NOT Q**
1	X	X	X	0	1
0	↑	0	0	0	1
0	↑	0	1	0	1
0	↑	1	X	A0	NOT A0

The implementation in simulation to achieve this functionality (using the D Type Flip Flop from the previous section) is shown in Figure 3 [fig:3]. Note, the Preset pin was left disconnected as specified in the previous section.

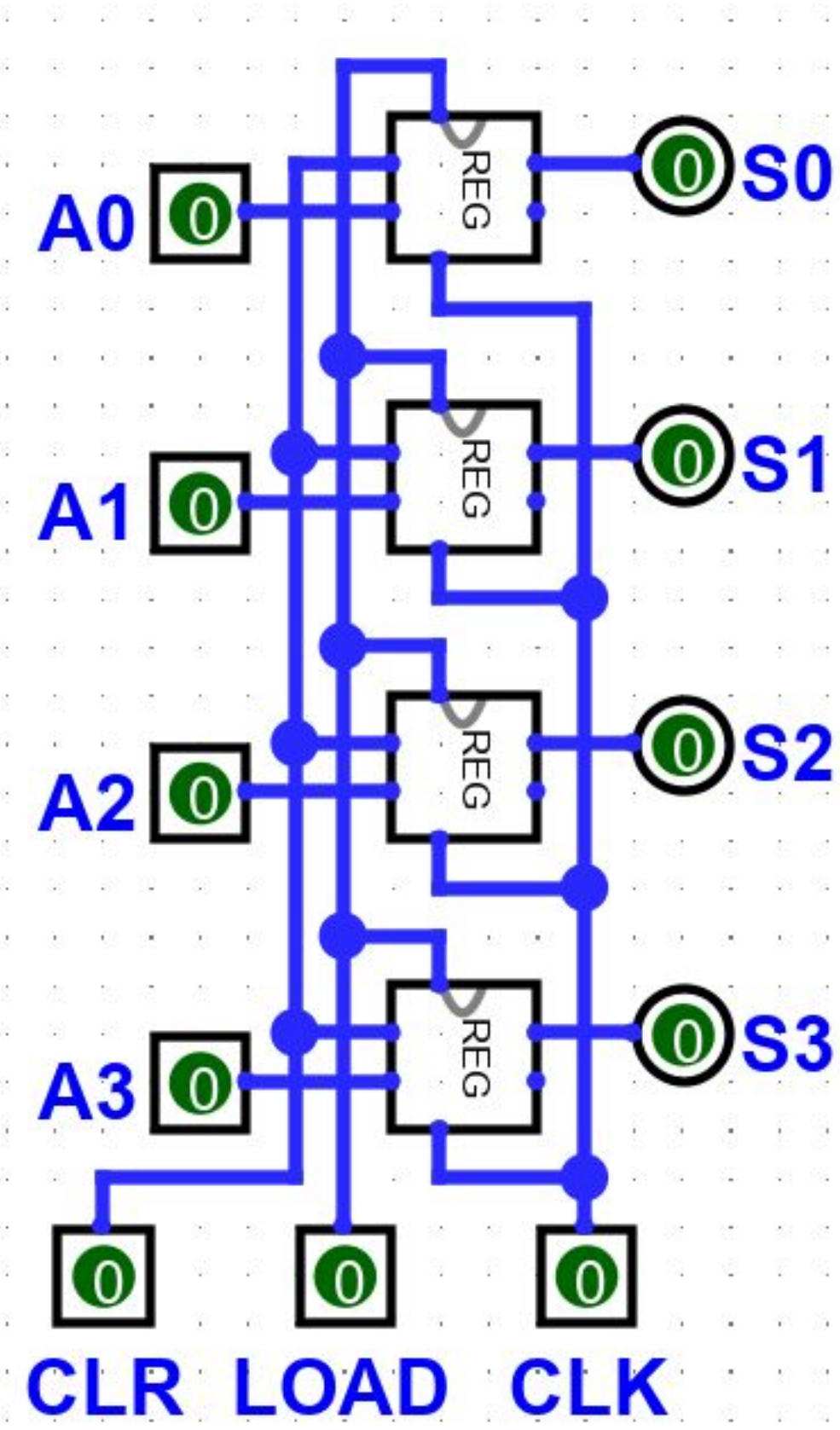
Single Bit Register[fig:3]



Four Bit Register

Next, a four bit register was designed, essentially daisy-chaining the One Bit Register subcircuit to achieve a four bit wide input and output, shown in Figure 4 [fig:4].

Four Bit Register[fig:4]

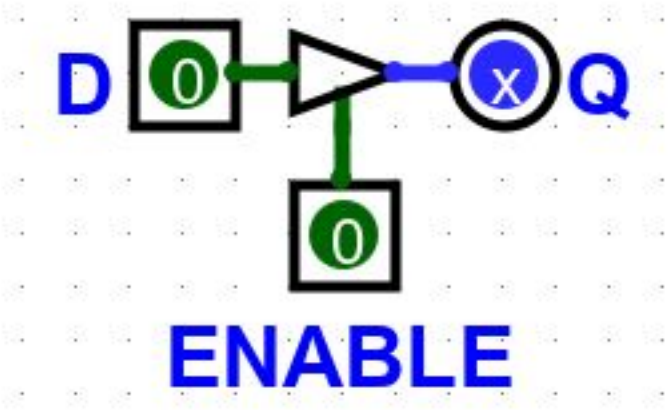

A0
A1
A2
A3
REG
REG
REG
REG
S0
S1
S2
S3
CLR
LOAD
CLK

The truth table remains identical to the one bit register, however the single bit input of A0 now maps to the four bit input A0-A3, and the single bit output Q now maps to S0-S3. The complement of this output is not required for the CPU circuit, so each NOT Q pin has been left disconnected to any outputs.

Buffers

The outputs of the registers cannot be directly wired to the common CPU data bus, as otherwise each subsystem's outputs would conflict along the same wires. To interface each subsystem with the bus, a buffer needs to be used. A tri-state buffer is used for this purpose. This component acts as a 'gate', allowing the output of, or input to, a subsystem to access the shared bus. The tri-state buffer is shown in Figure 5 [fig:5].

Tri-State Buffer[fig:5]



And the truth table of its operation:

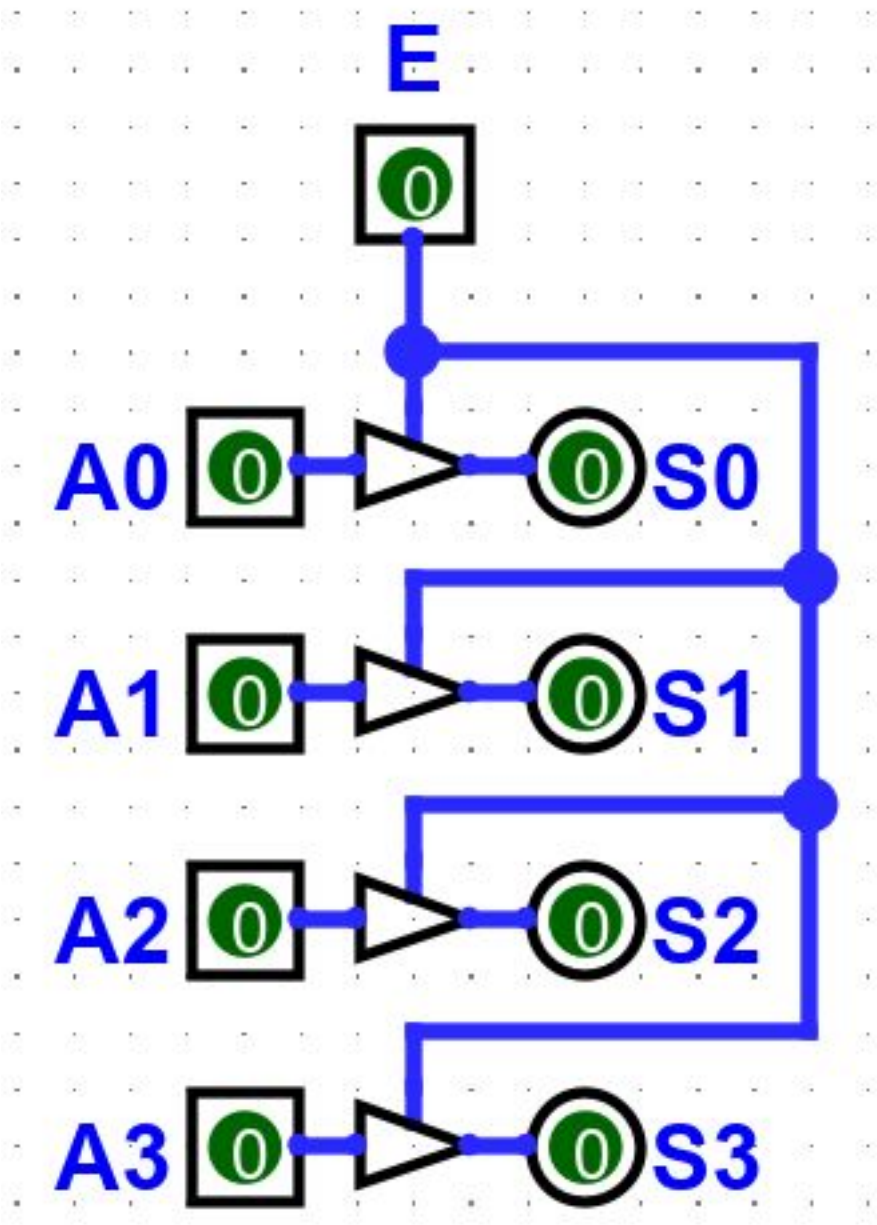
Inputs		***Outputs***
D	**ENABLE**	**Q**
X	0	High Impedance
0	1	0
1	1	1

As shown in the truth table, the tri-state has an extra state other than 0 or 1 (as the name implies) - this state is known as high impedance. This output essentially electronically isolates the output from the shared data bus, so no output whatsoever is passed onto it by the tri-state buffer. This solves the conflict issue with data being passed from the registers (or other subcircuits) onto the bus simultaneously.

N Bit Buffer

Using the tri-state buffers, a four bit buffer was simply designed by daisy-chaining them together, shown in Figure 6 [fig:6].

Four Bit Tri-State Buffer[fig:6]



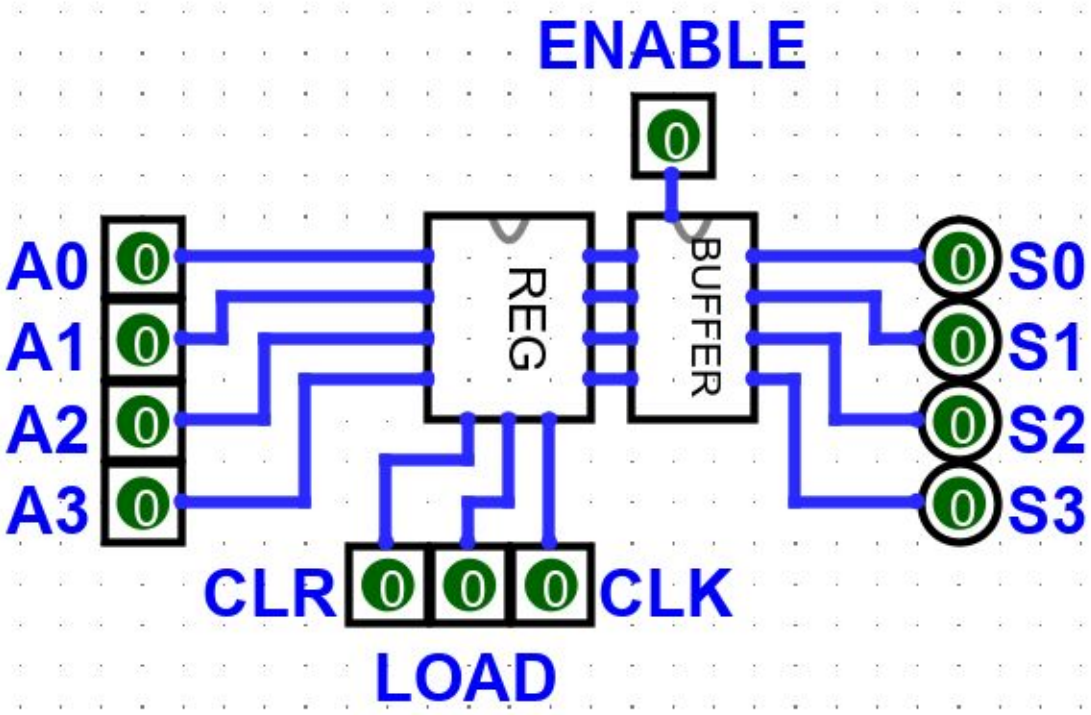
The reason for the four bit implementation instead of expanding straight to eight bit, is the aim of emulating select IC chips which could then be used to build this CPU in real life. This was kept in mind throughout the design process, and was applied to each subcircuit where possible.

Eight Bit A/B Registers

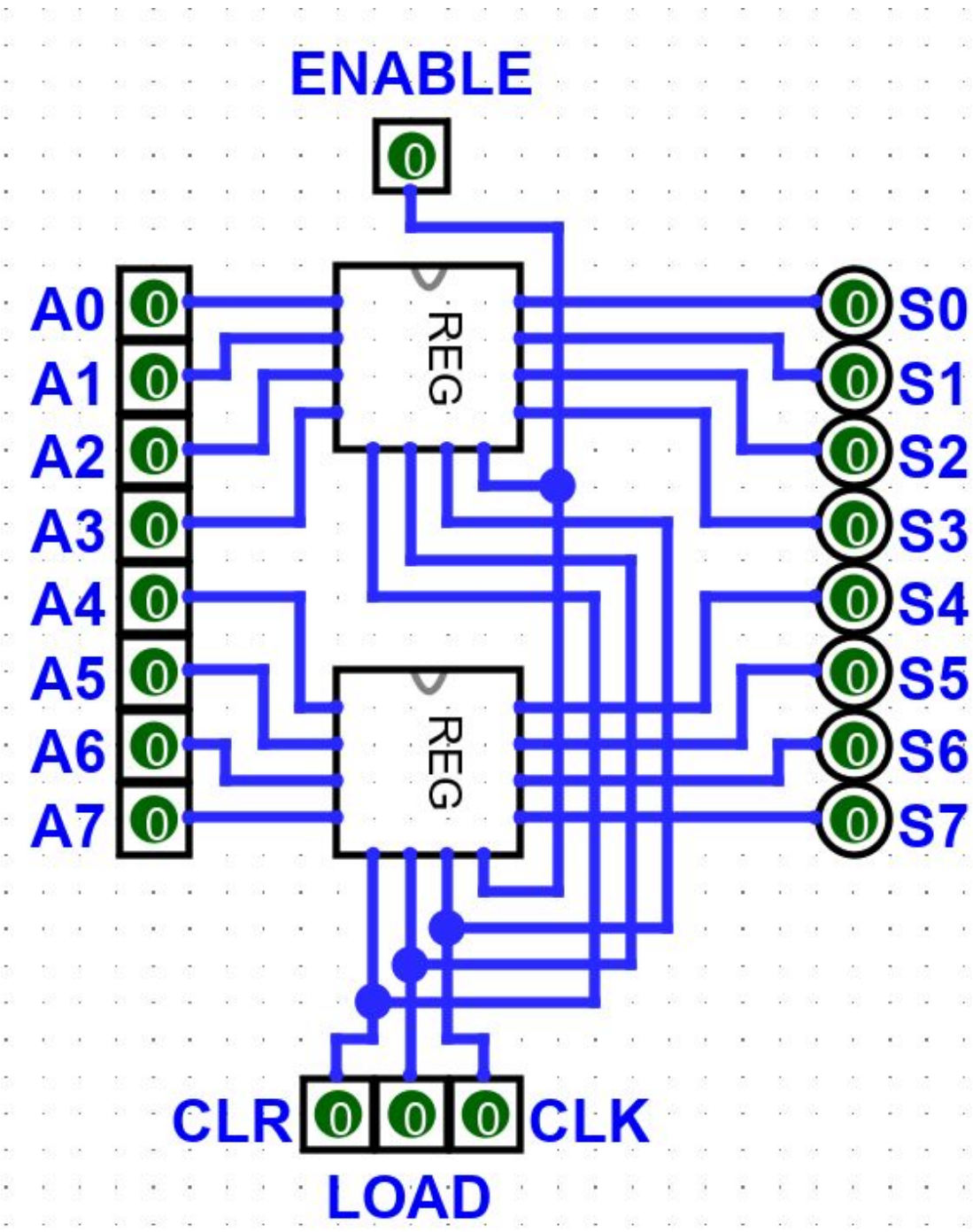
Combining the four bit register with the four bit buffer gives a four bit buffered register:

This subcircuit emulates the 74LS173 IC chip (4-Bit D-Type Registers with 3-State Outputs). Two of these circuits were then daisy-chained to make an eight bit implementation - the 4-Bit circuit is shown in Figure 7 [fig:7], and the daisy-chained 8-bit implementation in Figure 8 [fig:8].

Four Bit Buffered Register[fig:7]



Eight Bit Buffered Register[fig:8]



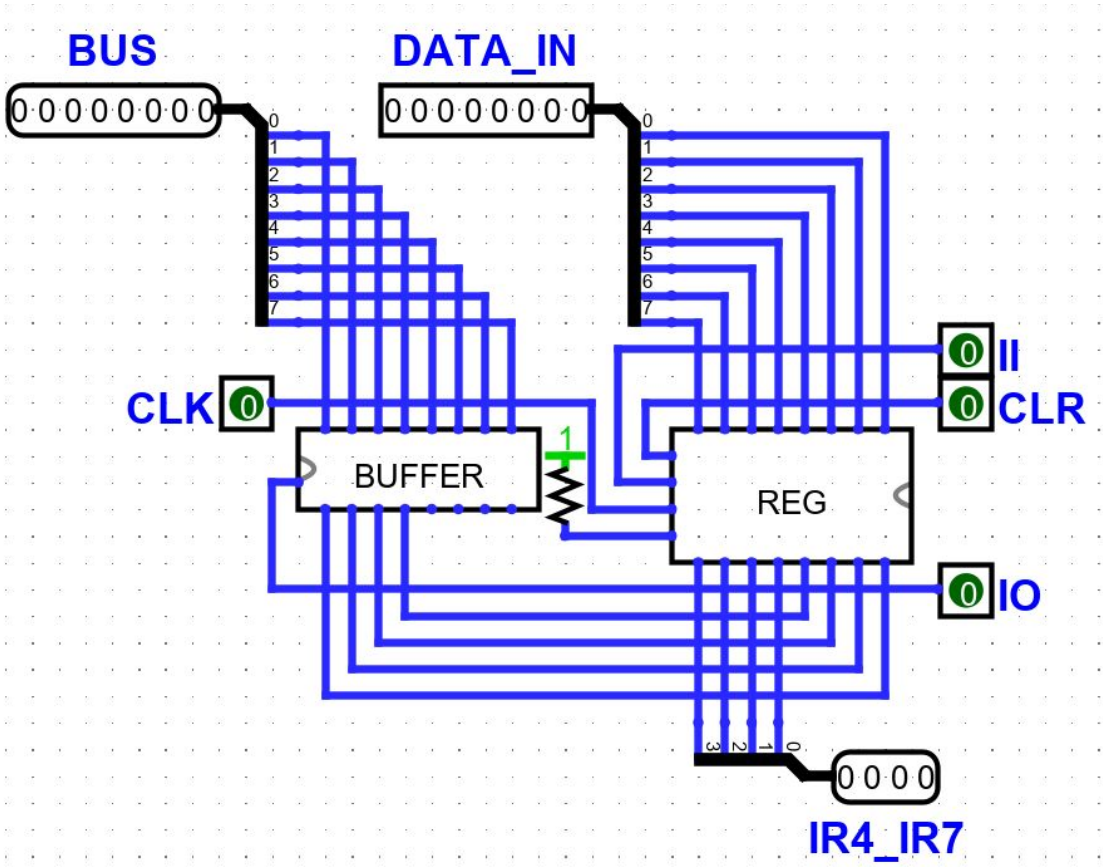
This was designed to be used as the A and B register in the CPU.

Four Bit Instruction Register

The instruction register needs to take its input from the upper 4 bits of the bus, i.e. the ‘opcode’, which will be defined later. These 4 bits are fed directly into the control logic when II is active, resulting in the relevant commands being sent around the system given the opcode

from the Instruction register. To achieve this, a simple 8-bit register and 8-bit buffer can be connected in a ‘loop’, shown in Figure 9 [fig:9].

Four Bit Instruction Register[fig:9]



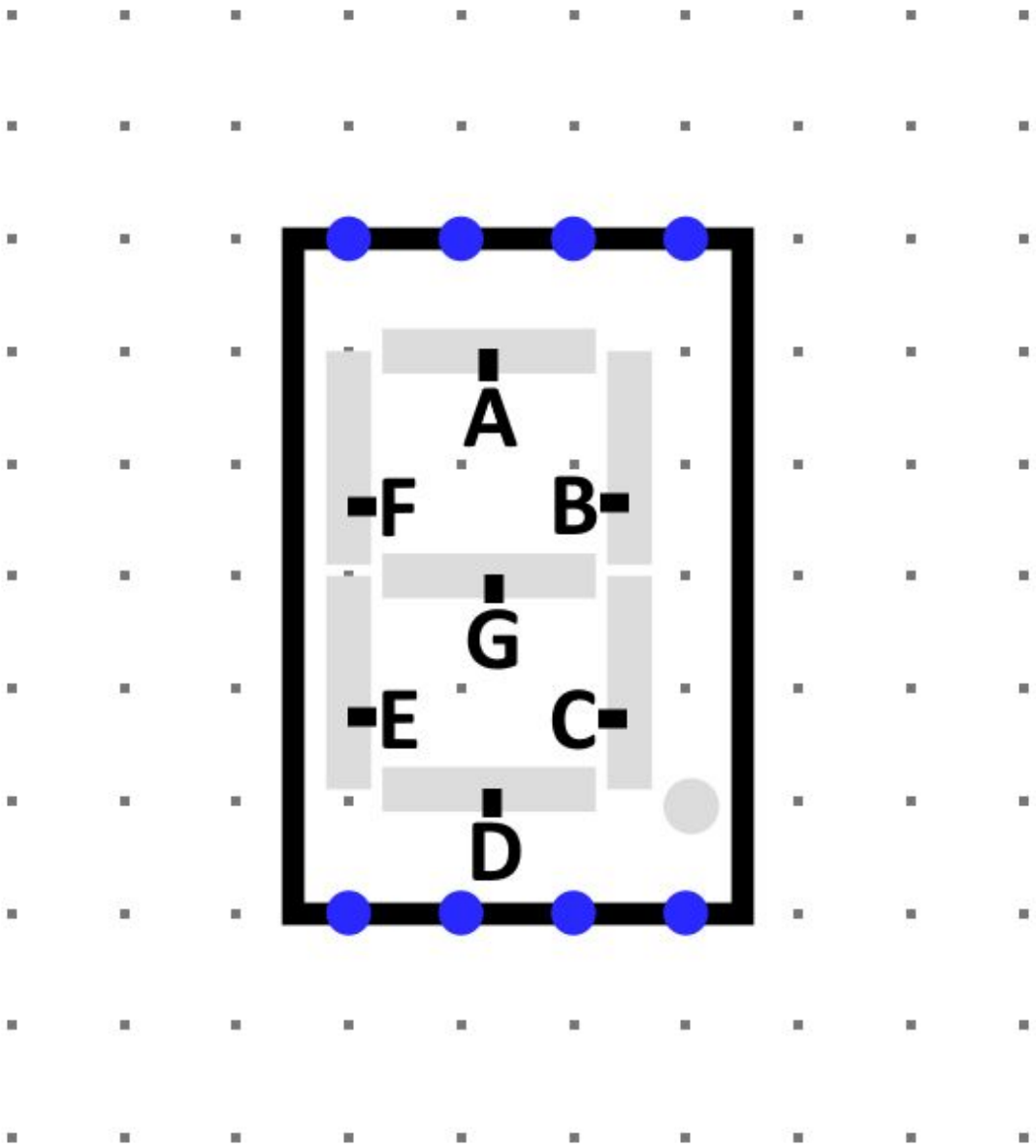
The upper 4-bits are extracted to be output across IR4_IR7, and the lower 4 allowed to pass back onto the bus when given the IO signal.

Output Register

The output register simply takes input from the bus when activated, and always displays its stored data across its output pins. It would be beneficial to have this register output a decimal value, rather than a binary value, as this would make reading the output of the current program much easier for the user, and makes the results of programs more immediately useful.

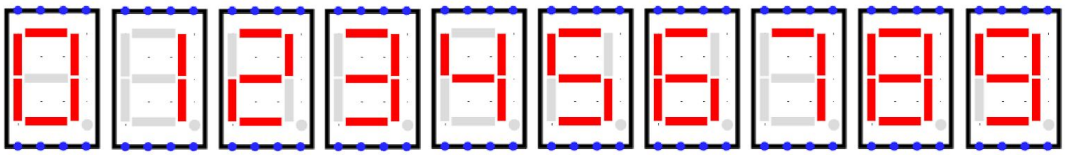
7-segment displays can be used for this purpose, which are 7 separate LEDs, known as ‘segments’, arranged around each other. There is also sometimes an additional LED for a decimal point if required, with each segment represented by a letter A through G. as shown in Figure 10 [fig:10].

7-Segment Display Segments[fig:10]



This allows the digits 0-9 to be represented, by activating certain LEDs as shown in Figure 11 [fig:11].

Digits on 7-Segment Displays[fig:11]



The separate LEDs allow a truth table to be constructed, mapping binary values to the required segments to display that value.

In order to 'split' the decimal value into each digit for the three individual displays, each decimal digit needs 4 bits to represent 0-9, (0000 to 1001). This is known as Binary Coded Decimal, or BCD for short. An example:

```
123 in decimal = 0111 1011 in binary
1 in decimal = 0001 in binary
2 in decimal = 0010 in binary
3 in decimal = 0011 in binary
```

Hence:

```
123 in decimal = 0001 0010 0011 in BCD
```

As the output register is 8-bits, the maximum value that can be represented in decimal is 255 - so 3 seven segment displays are required. 4-bits per digit means 12 BCD bits are required for the output.

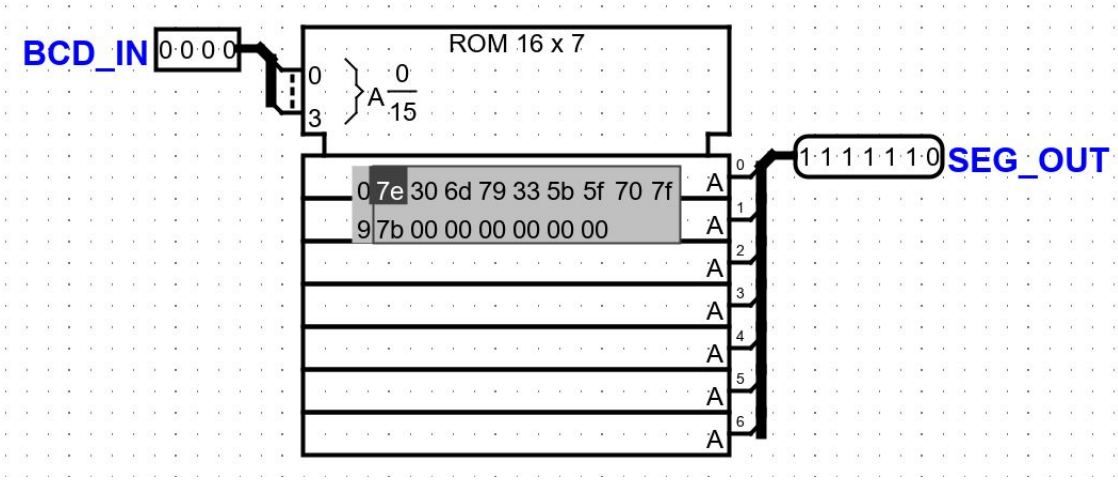
Each decimal digit i.e. 4-bits BCD output may then be converted to 0-9 on the display by a simple ROM.

The truth table for the binary to BCD conversion is 256 rows long, hence 8 address inputs, with each location outputting 4 bits to each of the 3 displays, hence a 12 bit width at each location. Each 4-bits group can then be fed into a ROM with 4 address inputs, and 7 bit outputs for driving each display:

BCD Input				***7 Segment Outputs***						
A	**B**	**C**	**D**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

As each value in ROM in Logisim Evolution is represented in hexadecimal, the resulting 7 bit binary word across outputs '**a**' through '**g**' were converted to hex, to be sequentially programmed in at each ROM address as shown in Figure 12 [fig:12].

Single Digit 7-Segment Display Driver[fig:12]

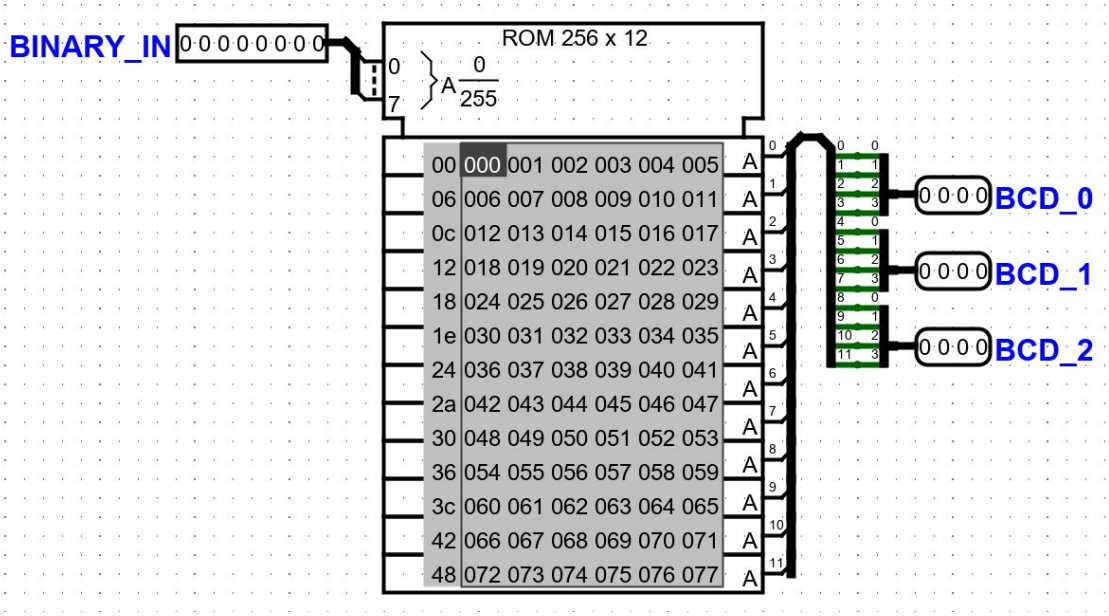


The binary to BCD ROM is larger, requiring 256 memory locations of 12 bits each to store the BCD representations of each 8-bit binary input. A handy coincidence is that as hexadecimal is base 16, a single hex digit can represent 16 different values - 0 to 15 in decimal, 0 to F in hex. Four binary bits also represent 16 values, 0000 to 1111. So for each four bit BCD value, the hex representation will actually be the same as the decimal value. For example:

```
255 in decimal
= 1111 1111 in binary
= 0010 0101 0101 in BCD
= 255 in hex
```

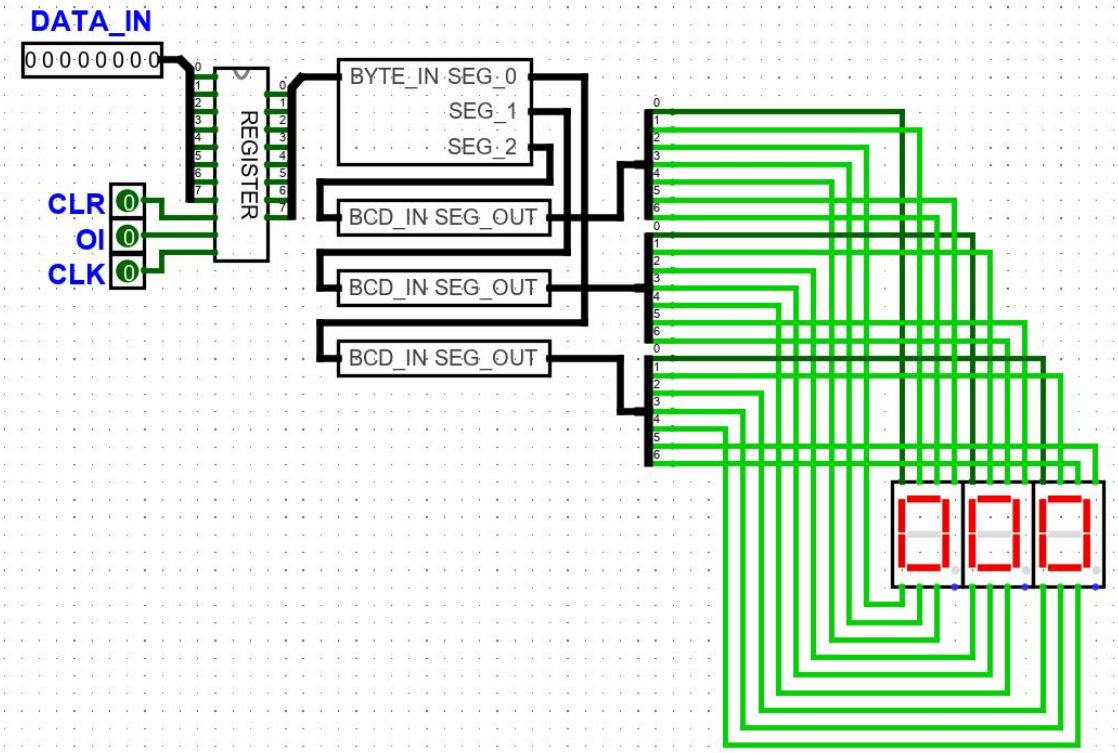
So each hex value in memory can simply be loaded with the decimal value wanting to be represented, as shown in Figure 13 [fig:13].

3 Digit Binary to BCD[fig:13]



And combining them both to convert the output register's output to a three digit decimal value, as shown in Figure 14 [fig:14].

8-Bit Binary to three 7-segment displays[fig:14]



ALU

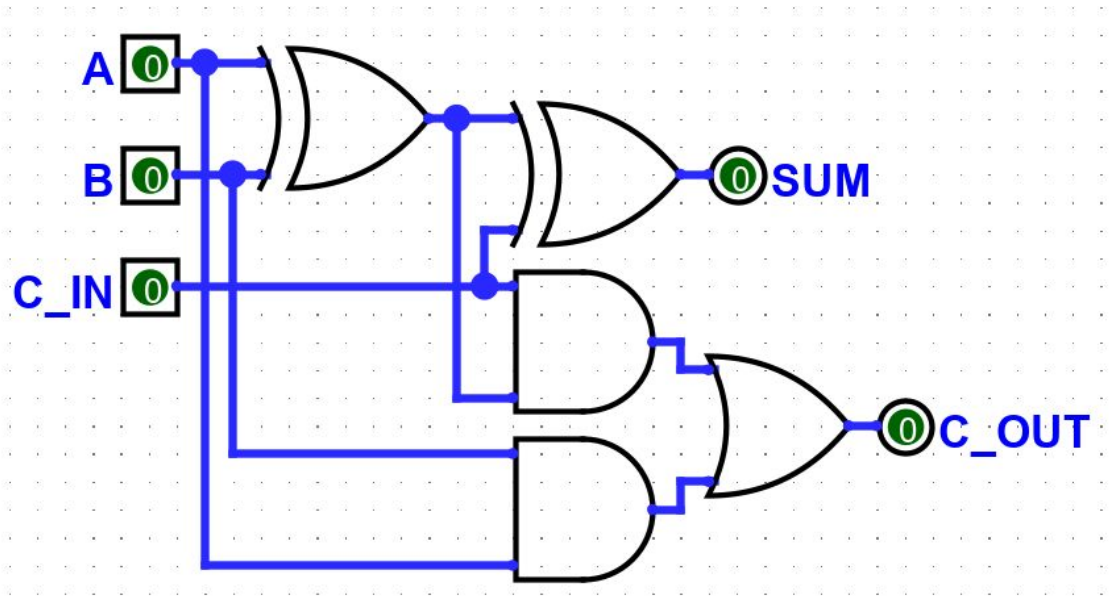
Single Bit Full Adder

The heart of the ALU is the full adder circuit. This ideal operation is to take two single binary bits and an optional ‘carry in’ input, and compute the sum and ‘carry out’ if present. The desired truth table looks like this:

Inputs			*Outputs*	
A	**B**	**Carry In**	**Sum**	**Carry 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

The optional ‘carry in’ input and ‘carry out’ outputs can be connected together if daisy chaining multiple instances of these adders, as the ‘carry’ will propagate through the n-bit wide input or result. The resulting circuit is shown in Figure 15 [fig:15].

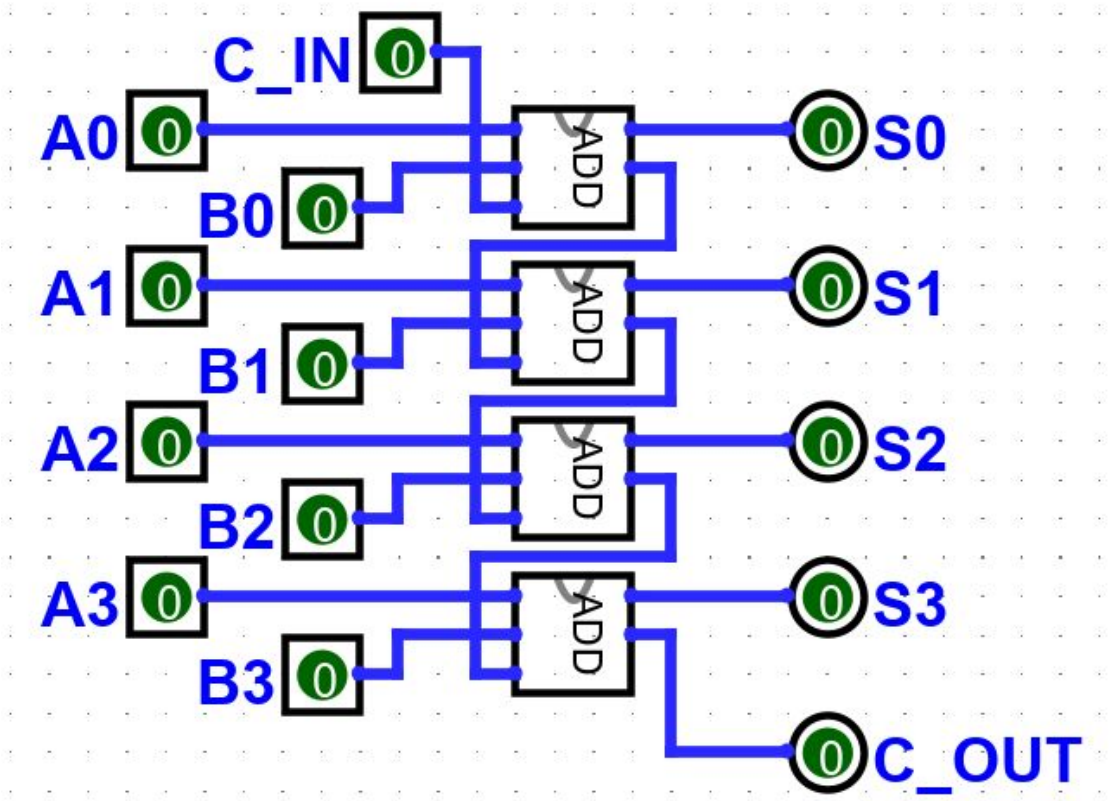
1-Bit Adder[fig:15]



Four Bit Full Adder

The previous circuit was then daisy chained to create a full adder of arbitrary length, to add two N bit binary numbers at a time. Shown in Figure 16 [fig:16] is a Four Bit Full Adder.

4-Bit Full Adder[fig:16]

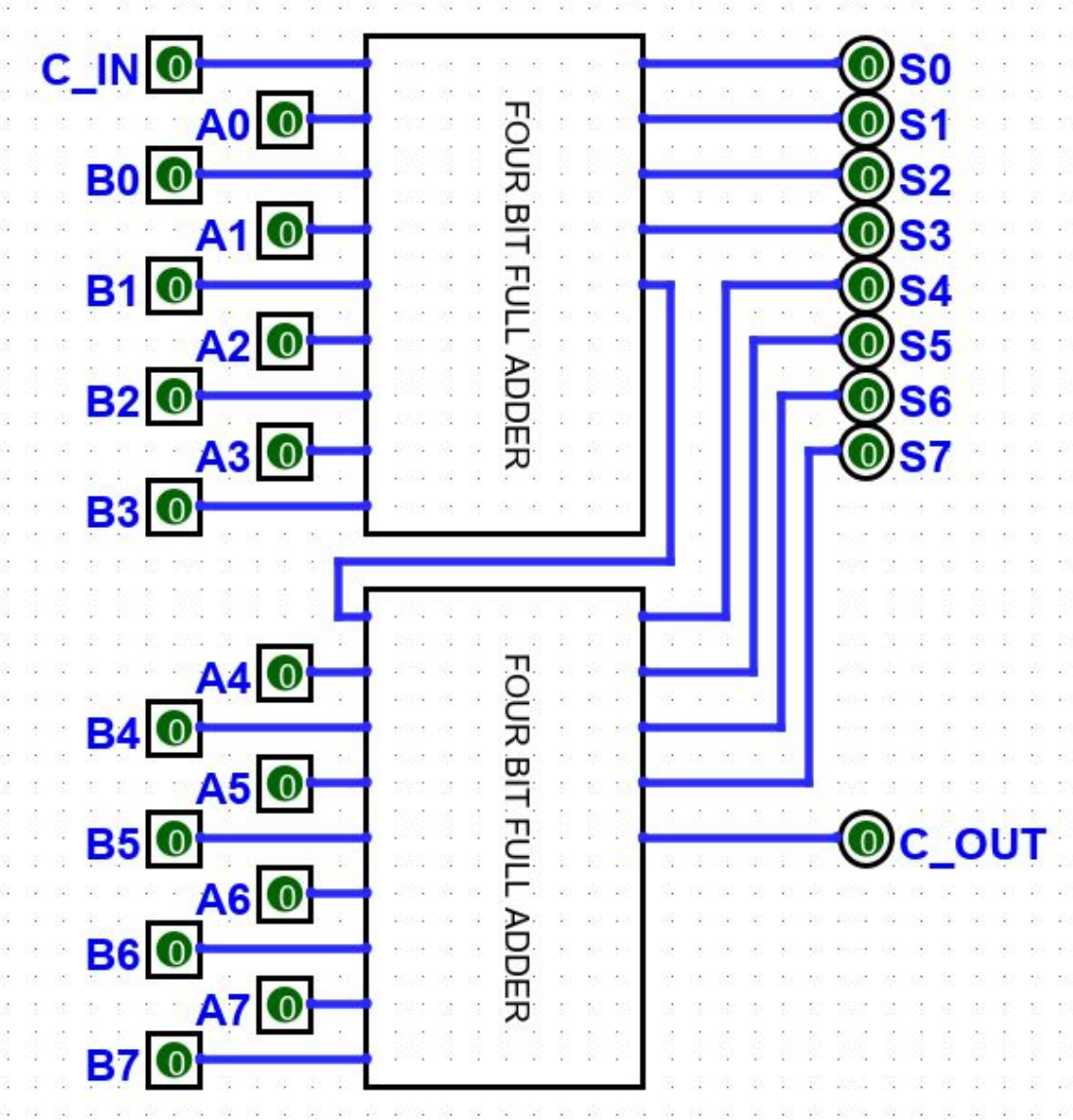


This circuit was designed to mimic the operation of the 74LS283 IC - '4-Bit Binary Full Adder with Fast Carry'. This means two of this IC or corresponding simulated circuit can again be connected to produce the eight bit implementation required for the CPU.

Eight Bit Full Adder

To achieve the eight bit implementation of the full adder, the previous circuit was daisy chained, with the carry out of the 'first' chip connected to the carry out of the 'second' chip - as shown in Figure 17 [fig:17].

8-Bit Full Adder[fig:17]



The operation remains identical to the one bit implementation, as essentially each A bit and B bit are computed in their pairs - the only difference is the carry will now propagate through each adder, and the input and output bit widths are eight wide instead of one.

Eight Bit Full Adder w/ Subtraction

A different way to think about subtraction is as the addition of a negative value. For example:

```
A - B = A + (-B)
```

This means the full adder can be used to subtract, if a negative representation of input values is allowed.

‘Two’s Complement’ is a system of representing negative (as well as positive) binary integers, by instead considering the most significant bit as a negative value while the rest of the bits remain as the standard positive representations. For example, the integer:

```
1001
```

In standard representation:

```
= (8 + 0 + 0 + 1) = 9
```

In Two’s Complement representation:

```
= (-8 + 0 + 0 + 1) = -7
```

Non-negative values remain the same in Two’s Complement, but as the MSB now represents a negative value, the range of values which can be represented in a given number of bits is ‘shifted’ to accommodate the negative values. For example, in a standard 8-bit representation:

```
Smallest possible value = 0000 0000 = 0
Largest possible value = 1111 1111 = 255
```

In 8-bit Two’s Complement:

```
Smallest possible value = 1000 0000 = -128
Largest possible value = 0111 1111 = 127
```

Conveniently, there is a very simple algorithm to negate a positive value represented in Two’s Complement. Each bit of the integer is inverted, and 1 is added to the result. A simple example:

```
0011 1110 = 62
```

Inverting the bits:

```
= 1100 0001
```

Adding 1:

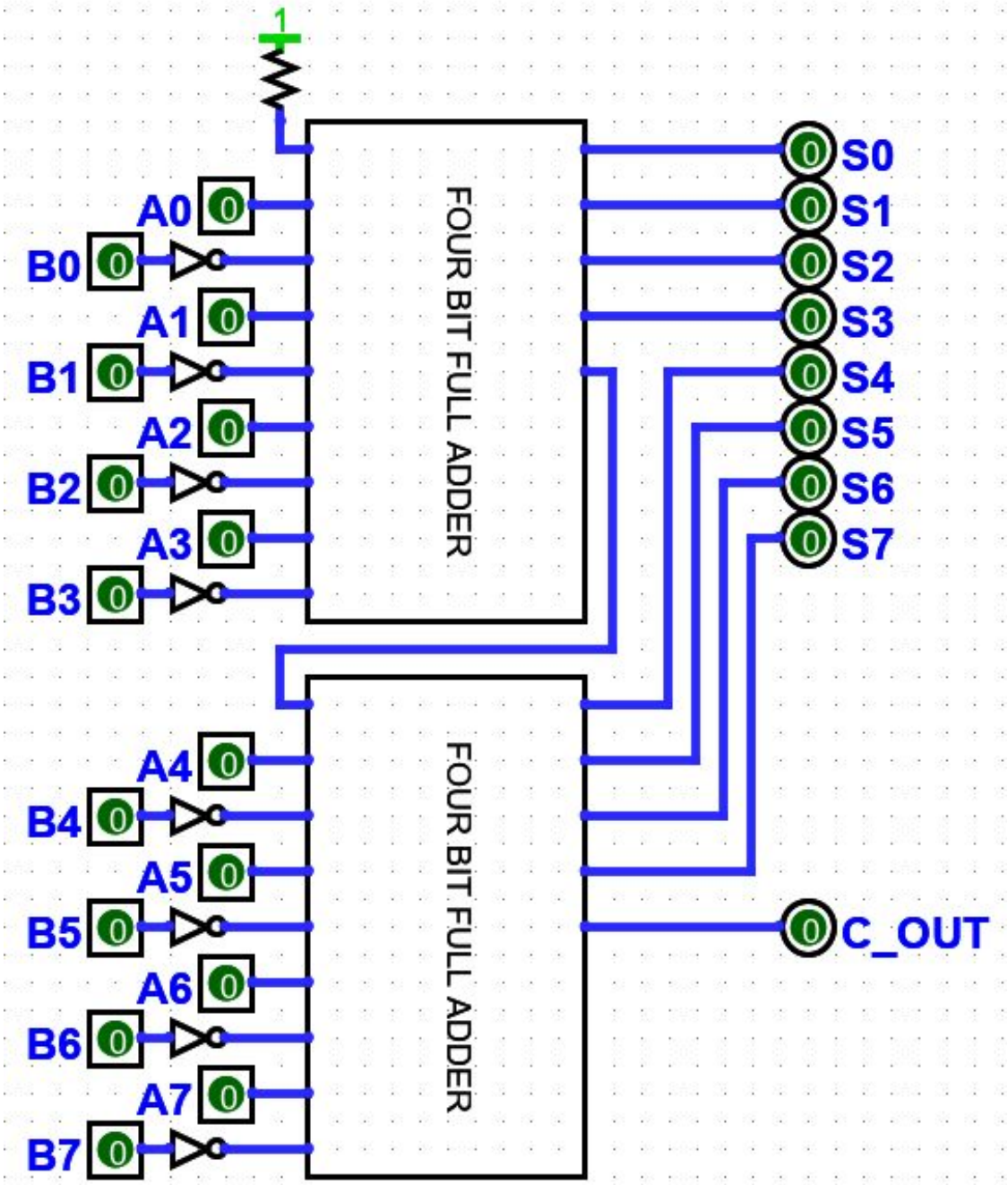
```
= 1100 0010
```

Conversion to decimal:

```
= (-128 + 64 + 2) = -62
```

This means the full adder can perform subtraction, if the addition is performed where one operand has been converted to a negative value. The inverting of the bits of one of the operands can easily be achieved using NOT gates, and the adding one was achieved by setting the ‘Carry In’ input of the Full Adder to High, shown in Figure 18 [fig:18].

8-Bit Full Adder performing Subtraction[fig:18]

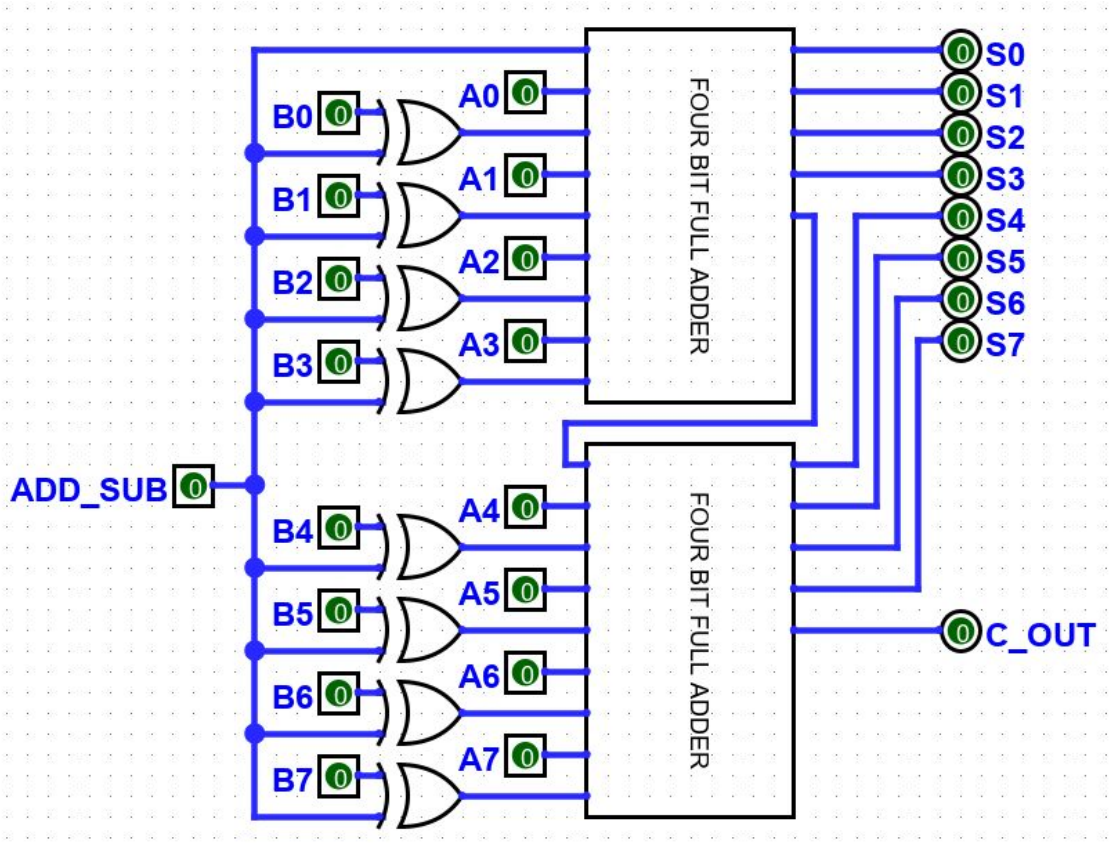


For the purposes of the CPU, it would've been much more convenient and efficient to have the Full Adder circuit perform both addition and subtraction, rather than having a seperate adder and subtractor. The 'Carry In' could be optionally activated by a simple toggle switch input, however the B input needed to be 'optionally inverted'. That is, when an 'invert' signal is given, the output is the inverse of the input. The truth table for the desired operation:

Inputs		*Outputs*
In	**Invert**	**Out**
0	0	0
0	1	1
1	0	1
1	1	0

This happens to be the exact same truth table as the 2-input XOR logic gate, where In and Invert are the inputs to the gate, and Out is the output. This meant one input to an XOR gate could be connected to a toggle switch, with each bit of the B operand connected to the other input, to 'optionally negate' each bit if the toggle switch was activated, shown in Figure 19 [fig:19].

8-Bit Full Adder with Optional Subtract[fig:19]



The toggle switch is also connected to the Carry In, to add the 1 required in the algorithm for negating the value. When ADD_SUB is high, the output of the subcircuit will be A + (-B) = A - B. When low, the output will be A + B.

ALU Flags

In order to facilitate ‘conditional instructions’ for the CPU, in which the program can ‘jump’ to other instructions based on a condition, some status signals need to be available about the current operation that the CPU can check against, then do something else if it’s been asked to. These are known as ‘flags’. The most common flags check the result of the ALU output - e.g. a Zero flag, which is a single bit set high when the result of the ALU’s operation is equal to zero. This, for example, can facilitate a loop inside a program, where a value is set in memory and 1 is subtracted N amount of times, where N is the desired amount of cycles of the loop. When the count reaches zero, i.e. the desired amount of loops have been executed, the program can check against the zero flag and jump elsewhere, or exit the loop. Allowing conditional ’jumps’ in the program’s execution allows the CPU to achieve ’Turing Completeness’, which means any computable problem may be solved by the CPU, given enough memory space for instructions ().

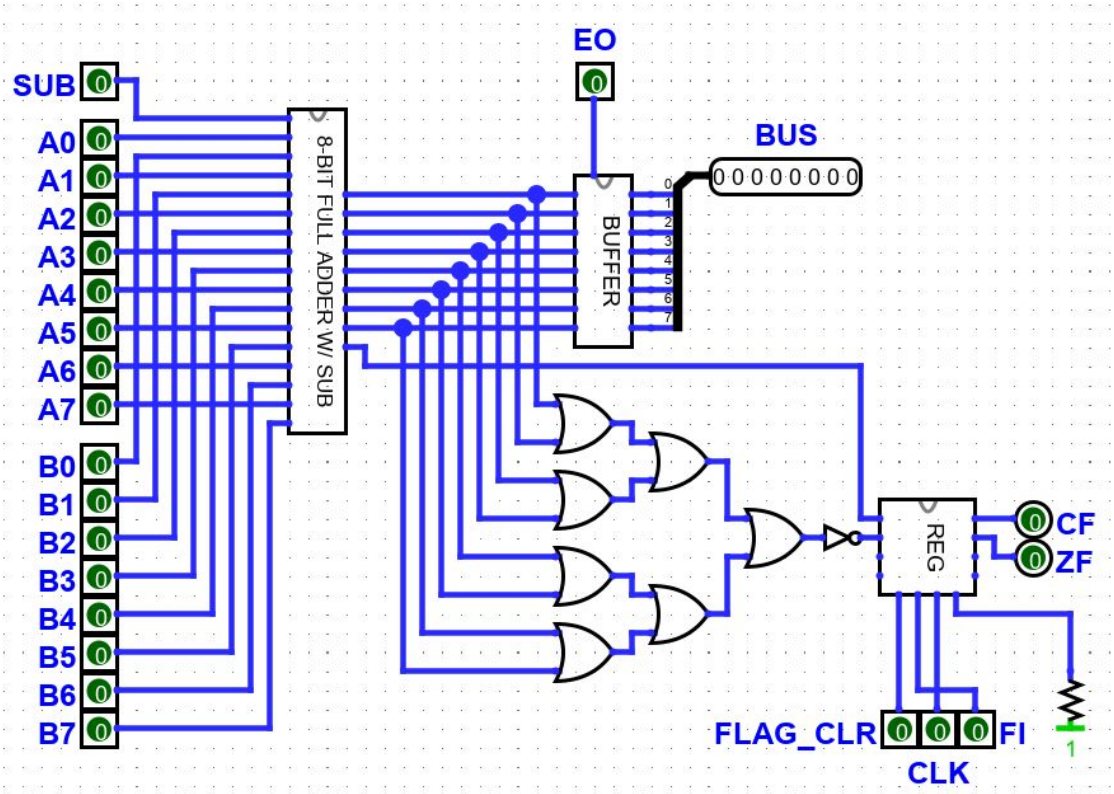
Another useful flag is the Carry flag, which indicates when the ALU result is too large to fit in its output bits. The flag bits need to be stored inside a register, so their values are remembered and can be checked against later in the execution of the program. The carry out of the ALU already provides an indication of whether the result is too large to fit in the output, so this can directly be used as the carry flag.

To check whether the output is zero, some logic needed to be implemented. Quite simply, if any of the bits are high, then the Zero flag is not set. Using OR gates to check if any of the output bits are high, then taking the inverse of the result, gives a single bit which is high only when the value is zero.

Summary

Combining the flags register with the full adder, with an intermediary buffer to optionally isolate the ALU’s output from the bus, yields the finished ALU shown in Figure 20 [fig:20].

The completed ALU[fig:20]



RAM

2-to-4 Line Decoder

In order to have addressable RAM locations, a way to single out a specific register from a bank of registers was needed. For the addresses, a binary input is required, so a circuit was needed which could take a binary integer, and convert it to a single unique output chosen from a range of single outputs: each of these unique outputs would be connected to a single register.

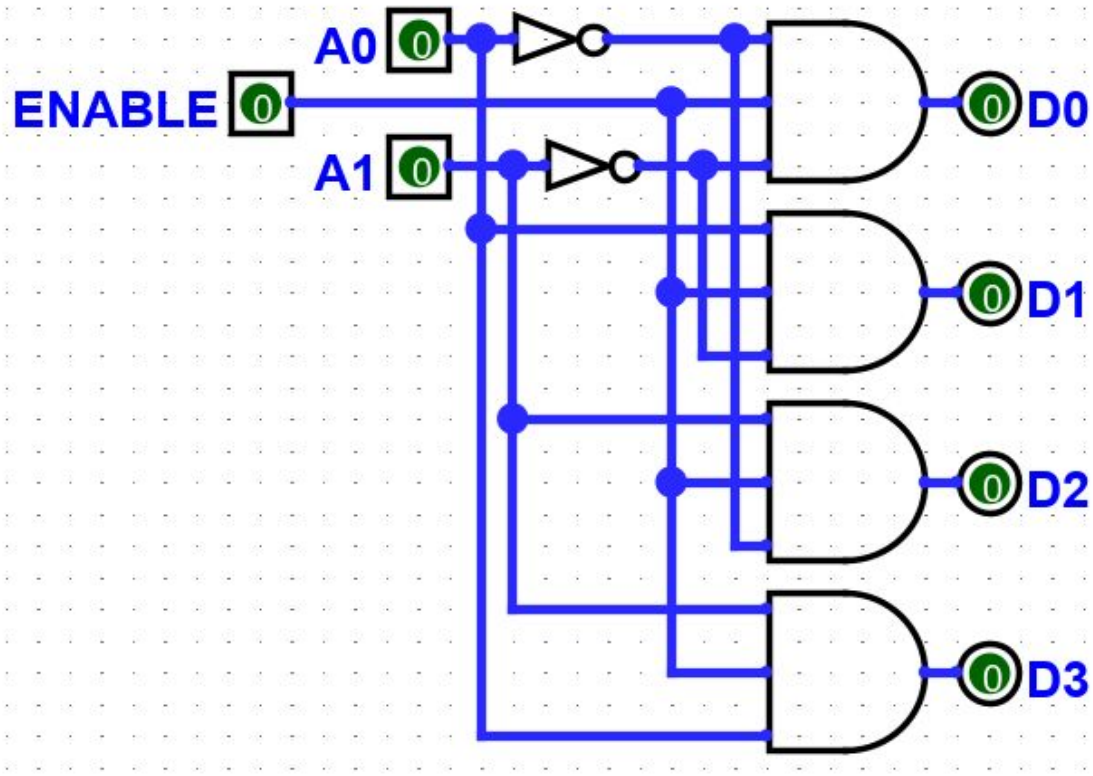
This way, all the registers could be fed the input data at once, and only the one which is being 'addressed' would store it. This eradicated the need for a huge input multiplexer routing the input data to a single register, then having a separate signal for the registers to 'store'. Instead, the input data is fed to all registers, and the line decoder triggers the 'store' signal for the addressed register.

This is the desired operation for a 2-bit integer to 4-line decoder:

Inputs			*Outputs*			
ENABLE	**A1**	**A0**	**D3**	**D2**	**D1**	**D0**
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Letting n equal the two-bit binary integer represented by A1 and A0, then the Dn output will be activated, provided ENABLE is high. The circuit implementation is shown in Figure 21 [fig:21].

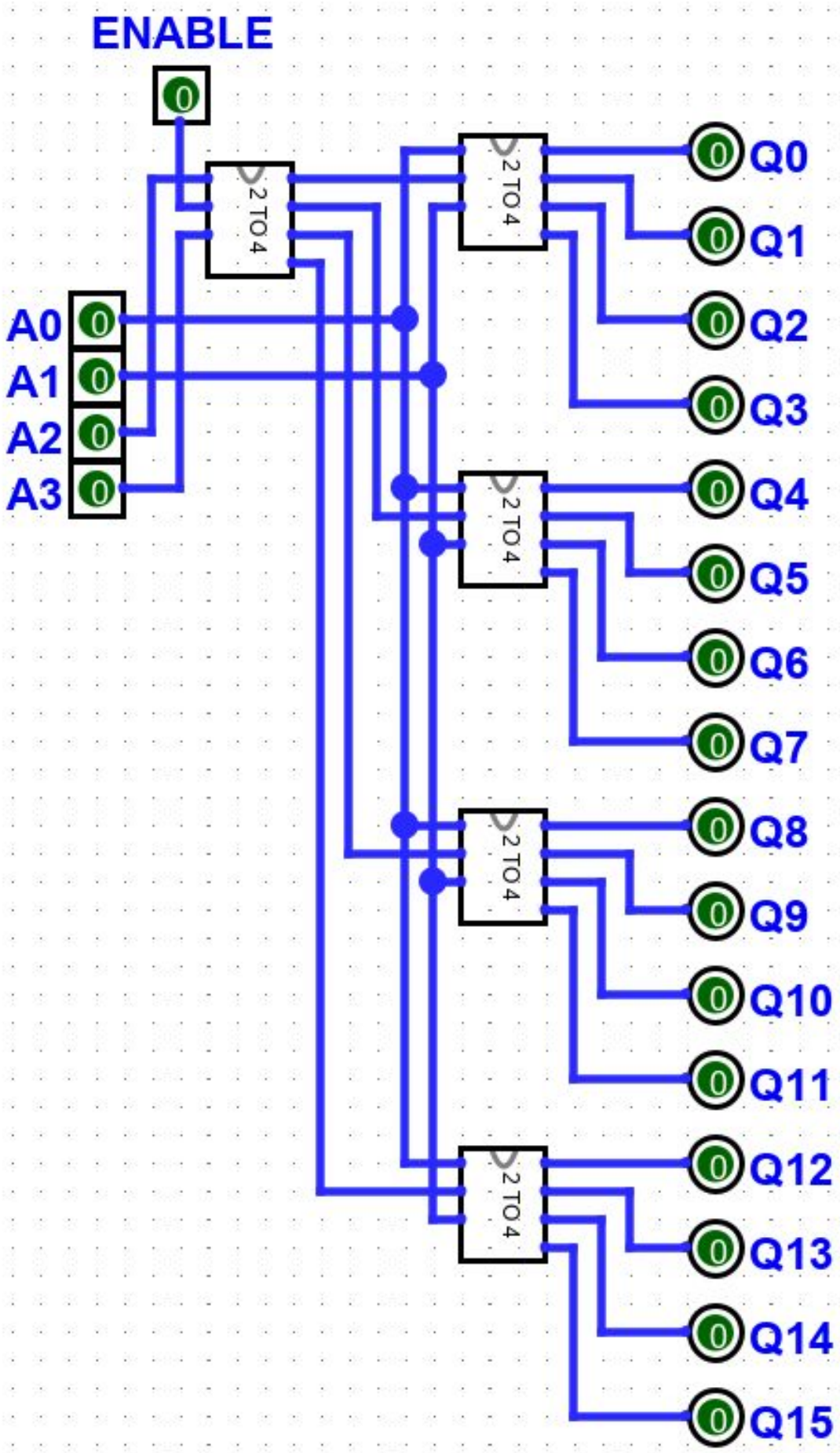
2-to-4 Line Decoder[fig:21]



4-to-16 Line Decoder

Expanding on the 2-to-4 decoder, it's trivial to daisy chain this decoder, and decoders in general, together to take a higher amount of inputs and produce a wider range of outputs. If each output of a single 2-to-4 line decoder is fed into the ENABLE of another 2-to-4 line decoder, then a 'nested' implementation can be achieved whereby each individual decoder is activated when required, shown in Figure 22 [fig:22].

4-to-16 Line Decoder[fig:22]


ENABLE
A0
A1
A2
A3
2 TO 4
2 TO 4
2 TO 4
2 TO 4
2 TO 4
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9
Q10
Q11
Q12
Q13
Q14
Q15

The four decoders on the right provide the 16 outputs, and the decoder on the left selects which of the four is 'allowed' to output.

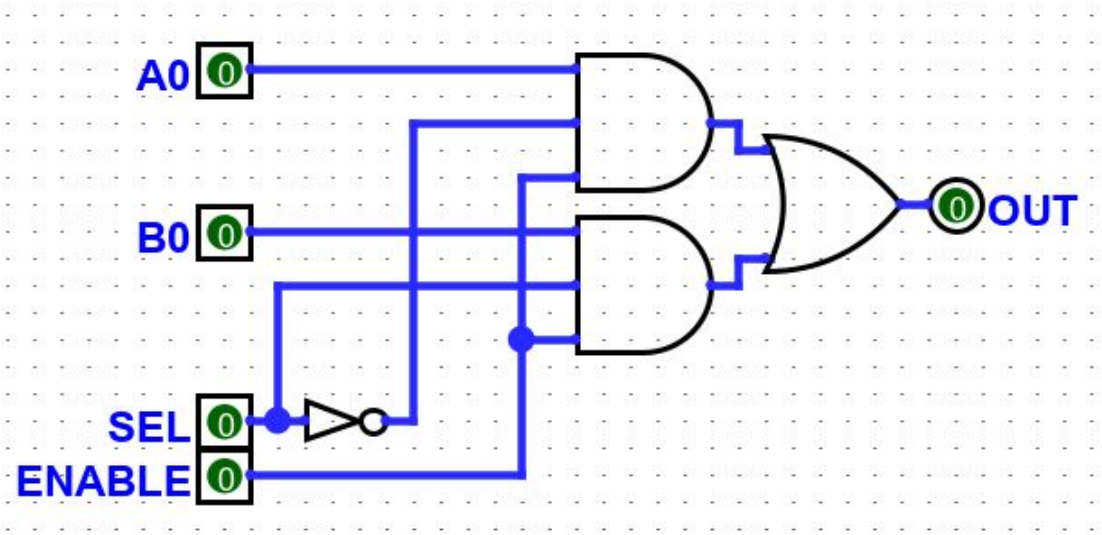
2-to-1 Multiplexer

With the addressing of the register bank handled by the decoder, a way to select an single register's output from a bank of registers was needed. This was handled by a multiplexer - also known as a 'data selector'. The desired operation is the selection, and consequent output, of a single input - given a range of inputs:

Inputs				***Outputs***
ENABLE	**A0**	**B0**	**SEL**	**OUT**
0	X	X	X	High Impedance
1	X	X	0	A0
1	X	X	1	B0

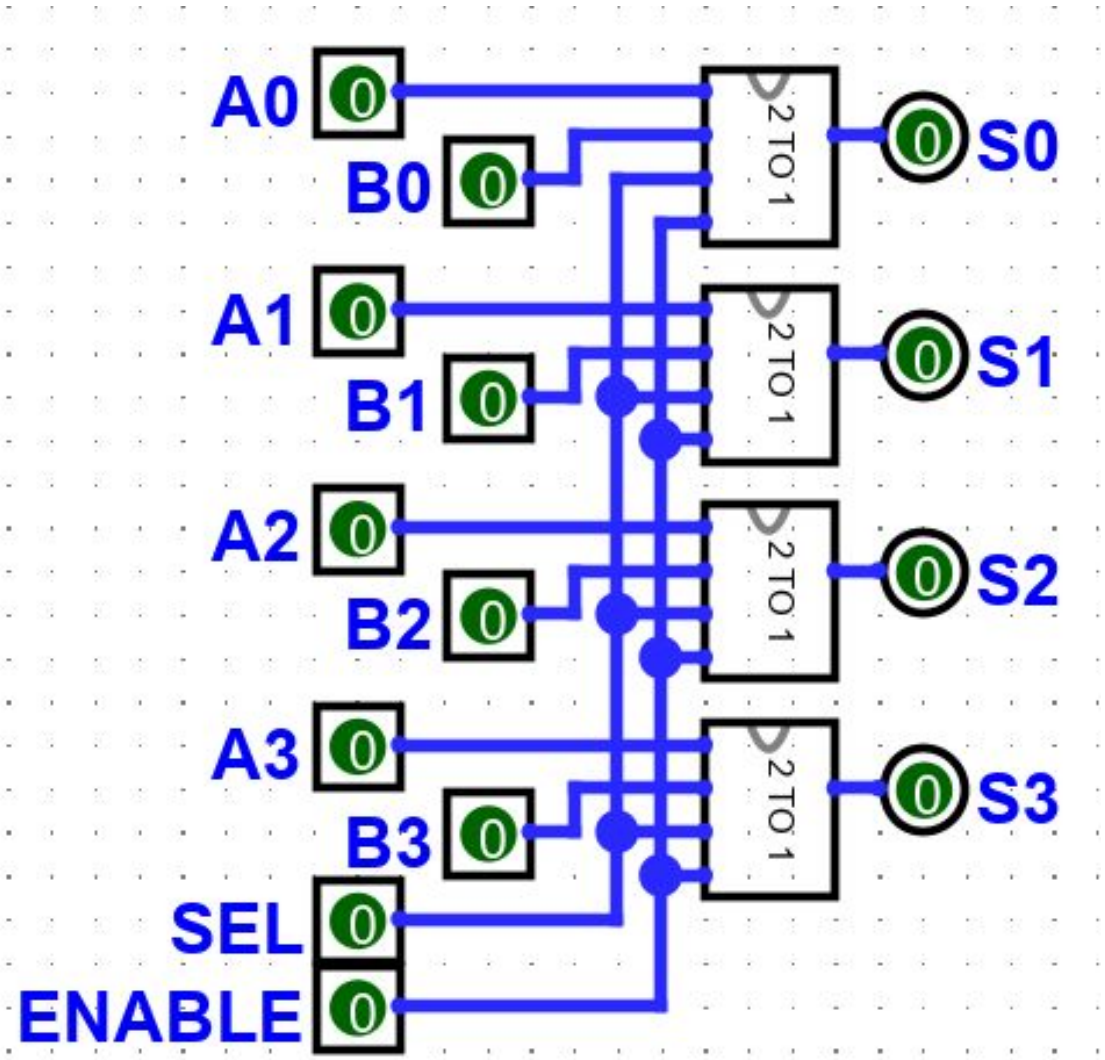
When ENABLE is high, and SEL is 0, the input A0 appears at the output. When ENABLE is high, and SEL is 1, the input B0 appears at the output. This allows an 'addressed' method of selecting the required output given the inputs - SEL = 0 gives one, and SEL = 1 gives the other. The circuit implementation is given in Figure 23 [fig:23].

2-to-1 Multiplexer[fig:23]



This implementation is configured as a one-bit wide, 2-to-1 multiplexer. In order to increase the bit width, several of these were stacked together to create a 2-to-1 multiplexer, potentially of an arbitrary length, as shown in Figure 24 [fig:24].

4-Bit 2-to-1 Multiplexer[fig:24]

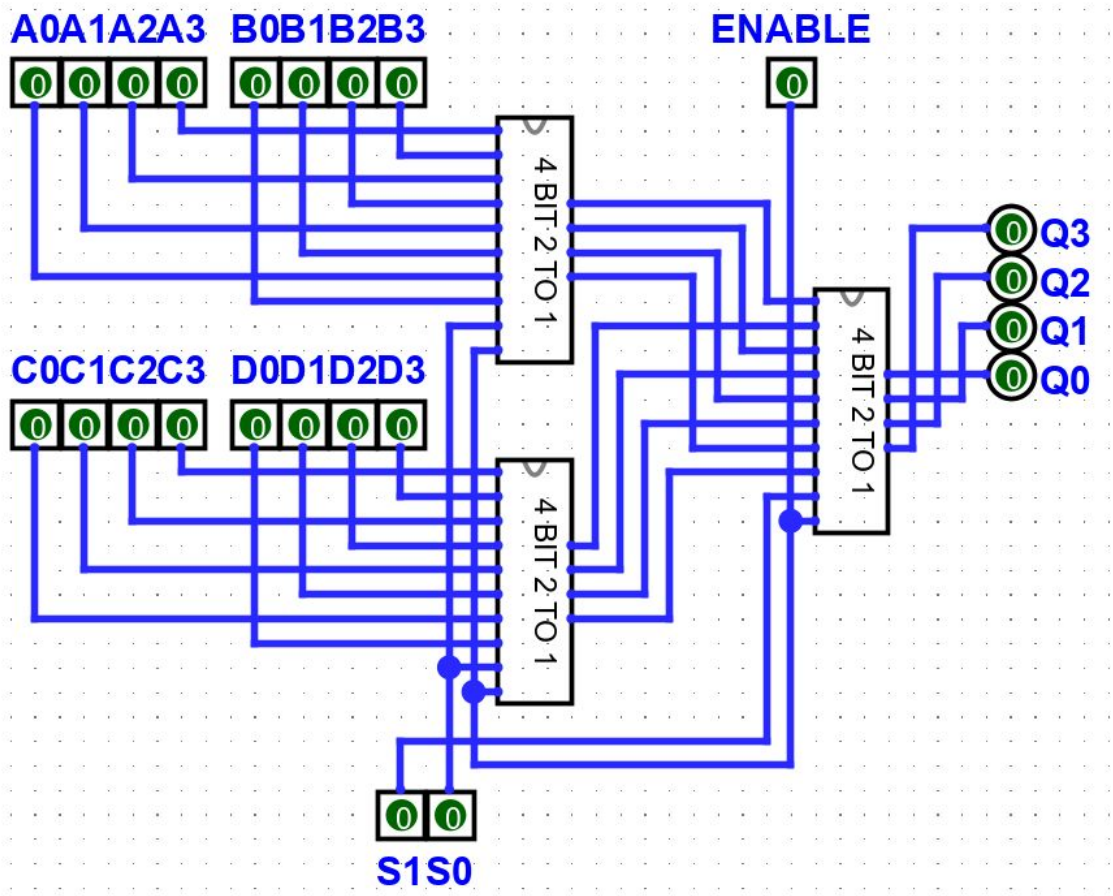


This results in a four-bit 2-to-1 multiplexer, which matches the four-bit word length of the RAM. This circuit intentionally mirrored the operation of the 74LS157 IC - ‘Quad 2-Input Multiplexer’.

4-to-1 Multiplexer

There are 16 registers storing data in the RAM module, so the multiplexing range needed to accommodate this. The 2-to-1 multiplexer can be relatively easily extended to allow larger amounts of inputs, shown in Figure 25 [fig:25].

4-Bit 4-to-1 Multiplexer[fig:25]

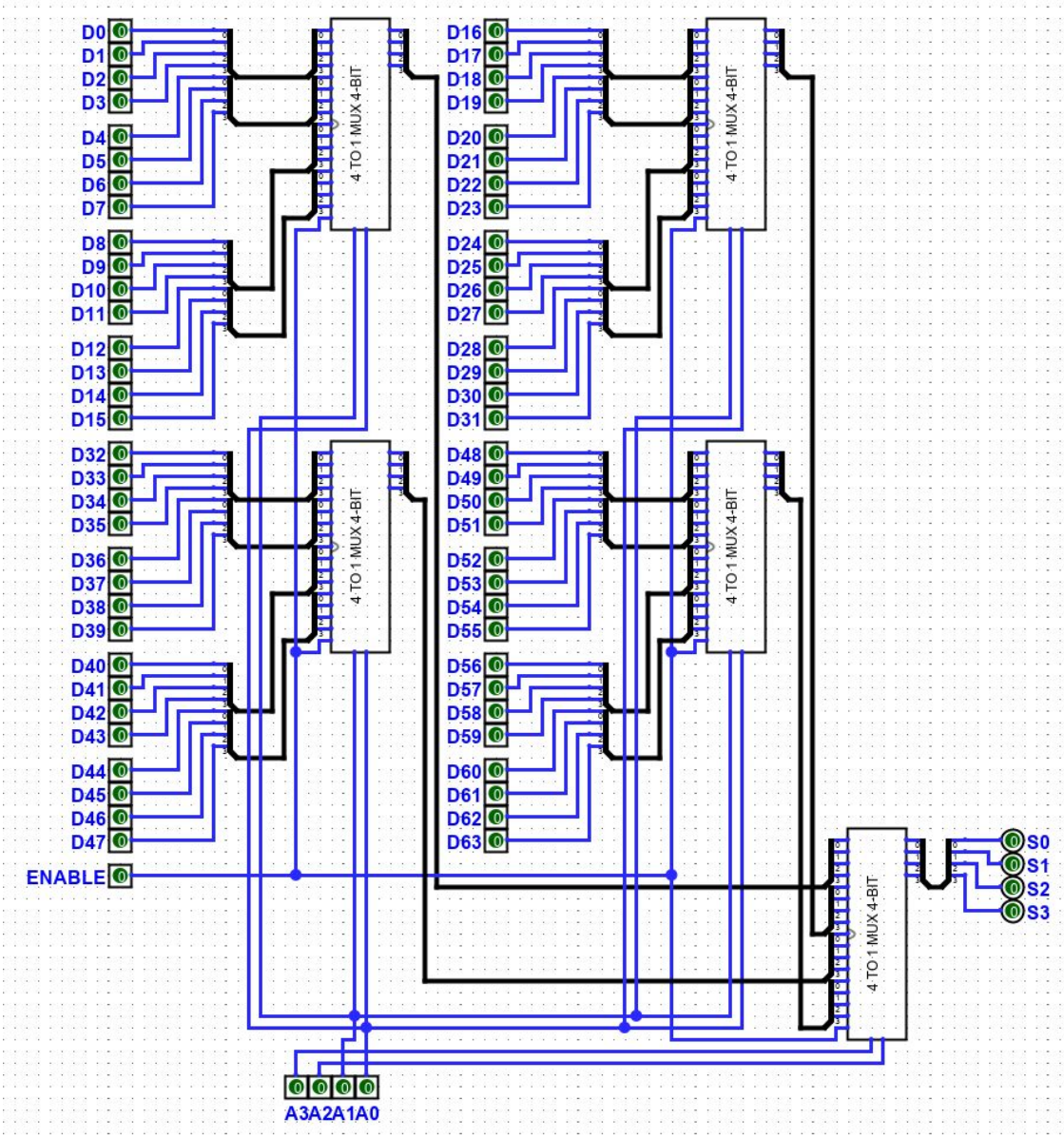


This is a 4-bit 4-to-1 multiplexer, built using the 4-bit 2-to-1 multiplexers shown by the previous circuit. The two 2-to-1 muxes on the left decide which of the A/B and C/D outputs will be selected respectively, and the mux on the right then chooses which of these two will appear at the output.

16-to-1 Multiplexer

Following on, a 4-bit 16-to-1 could now be constructed by 'stacking' the 4-to-1 implementation shown previously in Figure 25 [fig:25], with the result shown in Figure 26 [fig:26].

4-Bit 16-to-1 Multiplexer[fig:26]

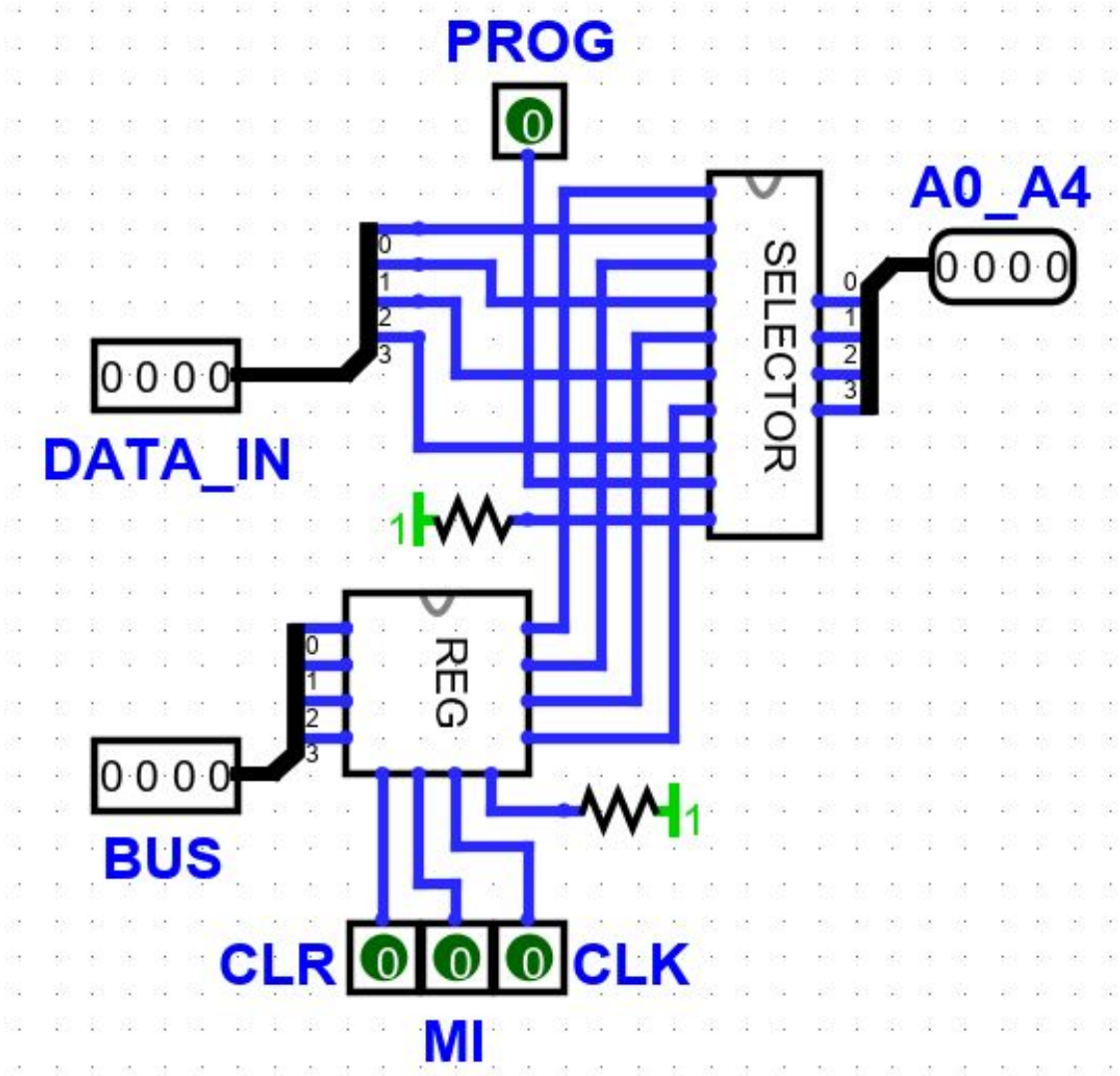


Although the new 'stacked' circuit shown in Figure 26 [fig:26] looks much more complicated than the 2-to-1 and 4-to-1 mux circuits, the principle remains identical. Each 4-bit 4-to-1 mux divides the amount of its 4-bit inputs by 4: so each outputs a single 4-bit value. Therefore, the four muxes arranged in a square shape output a single 4-bit value each. Then, each of these is passed to one more 4-bit 4-to-1 mux, which then decides which of the four values is to appear at the overall output. In other words, the upper two bits of the SELECT inputs (A3,A2) control which output each of the four initial muxes will all provide, and the lower two bits (A1,A0) control which of these four results will appear at the output mux.

Four Bit Memory Address Register

The Memory Address Register is a simple design, taking a 4-bit input from either the Program Counter via the bus (or the manual input DATA_IN when 'Prog' is high), and outputting a 4-bit address to RAM as shown in Figure 27 [fig:27].

4-Bit Memory Address Register[fig:27]



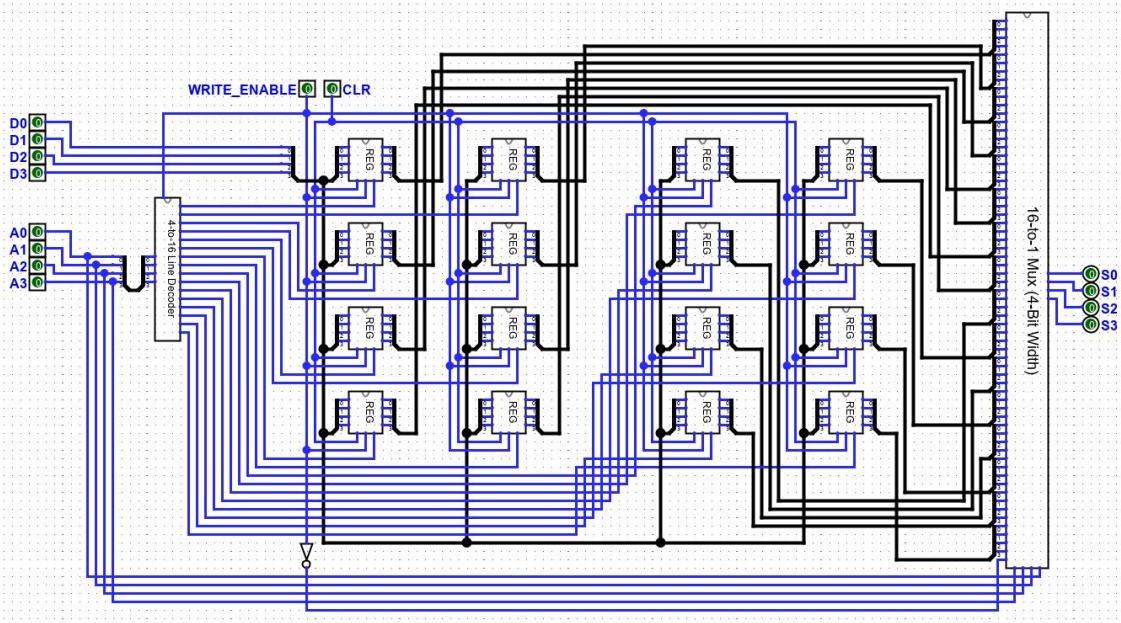
Sixteen Word Four Bit RAM

Using the line decoders and multiplexers designed, a RAM module could now be constructed. The line decoders can be given a binary 'address', and a corresponding single register can be 'activated' by the line decoder's single bit output. For a 4-to-16 line decoder, 4 address lines specify a single output chosen from 16 parallel outputs available - with one of these 16 outputs connected to each respective registers' 'LOAD ENABLE' pin. This allows 16 individual registers,

4-bits wide in this case, to act as ‘memory cells’ which may now be accessed for reading or writing, given an address.

The output of each of these registers, now configured to act as ‘addressed memory cells’, need to be selected between to give the RAM’s output. The 4-bit 16-to-1 multiplexer handles this. Each 4-bit register is fed into one of the sixteen 4-bit inputs of the multiplexer. The same address given to the line decoder can also be fed into the multiplexer, which yields a completed 16 word x 4-bit RAM module as shown in Figure 28 [fig:28].

16 x 4 Bit RAM[fig:28]

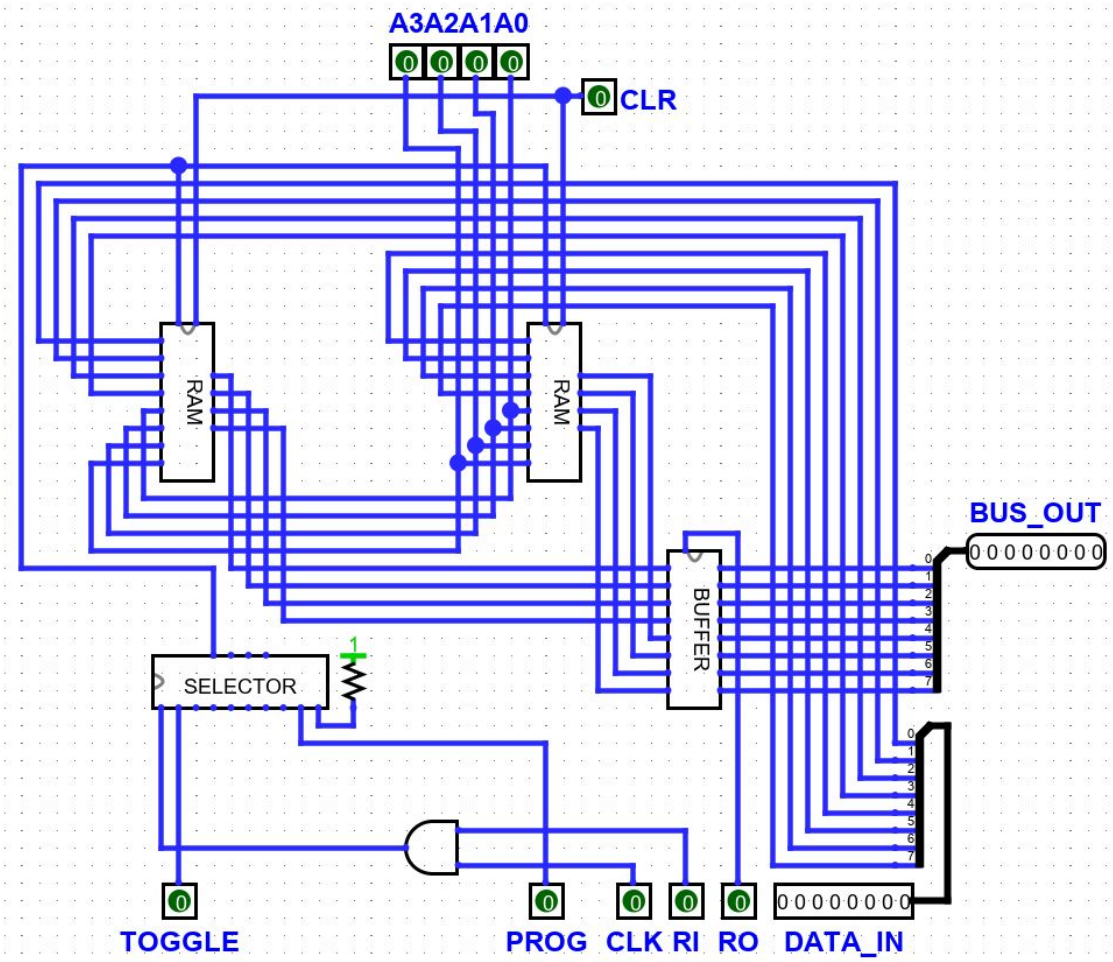


The WRITE ENABLE pin, when low, does two things. Firstly, it feeds a 0 into the 4-to-16 decoder’s ENABLE input, which prevents the registers storing any values given at D0-D4. Secondly, the 16-to-1 Mux is supplied with the opposite signal to the 4-to-16 decoder, using a single NOT gate, so the RAM isn’t allowed to take input and supply output at the same time - preventing data leaking to the shared bus when programming the RAM. This means by default the multiplexer is always emitting the value stored in the given address at A0-A4, and the 4-to-16 decoder is inactive.

Summary

The previously designed RAM module aims to mimic the 74189 - ‘64-Bit Random Access Memory with 3-STATE Outputs’ - hence only accommodating a single 4-bit value per address. For the 8-Bit CPU, each command, known as an ‘opcode’, occupies a 4-bit value. The other 4-bits of the 8-Bit Bus are used to pass a value, address or some other data for the opcode to act upon - the ‘operand’. This means the RAM, which is where the program to run is loaded, needs to hold at least 8-bits. To achieve this, two of the 74189-style chips can be daisy chained. The amount of locations - 16 - stays the same, but twice as many bits can be stored given two chips. The address inputs of both can be simply tied together as shown in Figure 29 [fig:29].

16 x 4-Bit RAM[fig:29]



Program Counter

Counting in binary is intrinsically based on powers of 2. This can be easily shown by the sequence of numbers between 0 and 7 in the binary system:

```
000
001
010
011
100
101
110
111
```

If each bit is treated as its own ‘column’ starting from the top value, the least significant bit (rightmost) ‘toggles’ its value between 0 and 1 every time the overall value increments by 1. The middle bit ‘toggles’ half as frequently as the least significant bit, and the most significant bit

‘toggles’ half as frequently as the middle bit. This chain continues with the more bits added, with each bit ‘toggling’ half as often.

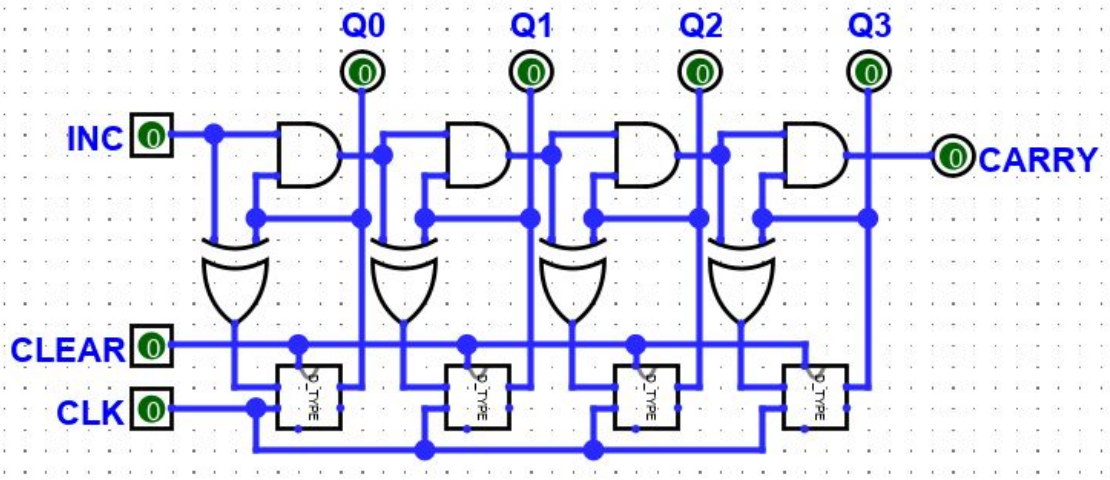
This is because each bit added in the most significant position represents an incrementally higher power of 2 - therefore is active only in the higher ‘half’ of the overall represented value. A component which can toggle its value between 0 and 1 based on its inputs and a clock signal would allow daisy chaining of such a component in order to count up from 0 to an arbitrary binary value.

Four Bit Synchronous Up Counter

To achieve the counting function, the ‘counter’ needed to increment each time a clock pulse is applied to it. Essentially, the clock pulses are counted, each rising edge raising the count by 1. D-Type Flip Flops, as already explored, ‘latch’ the value present at their D input to their Q output, on the rising edge of a clock pulse. Chaining these together in a specific way allows them to trigger each other, each one half one as often as the last, which produces a clean N-bit binary count output, where N equals the amount of D-Types used.

The context of this counter is to increment through the addresses in RAM, sequentially executing the CPU’s instructions. It’s desirable to add a ‘Count Enable’ input for the counter, so it only increments when required. The circuit implementation of this operation is shown in Figure 30 [fig:30].

4-Bit Synchronous Up Counter[fig:30]



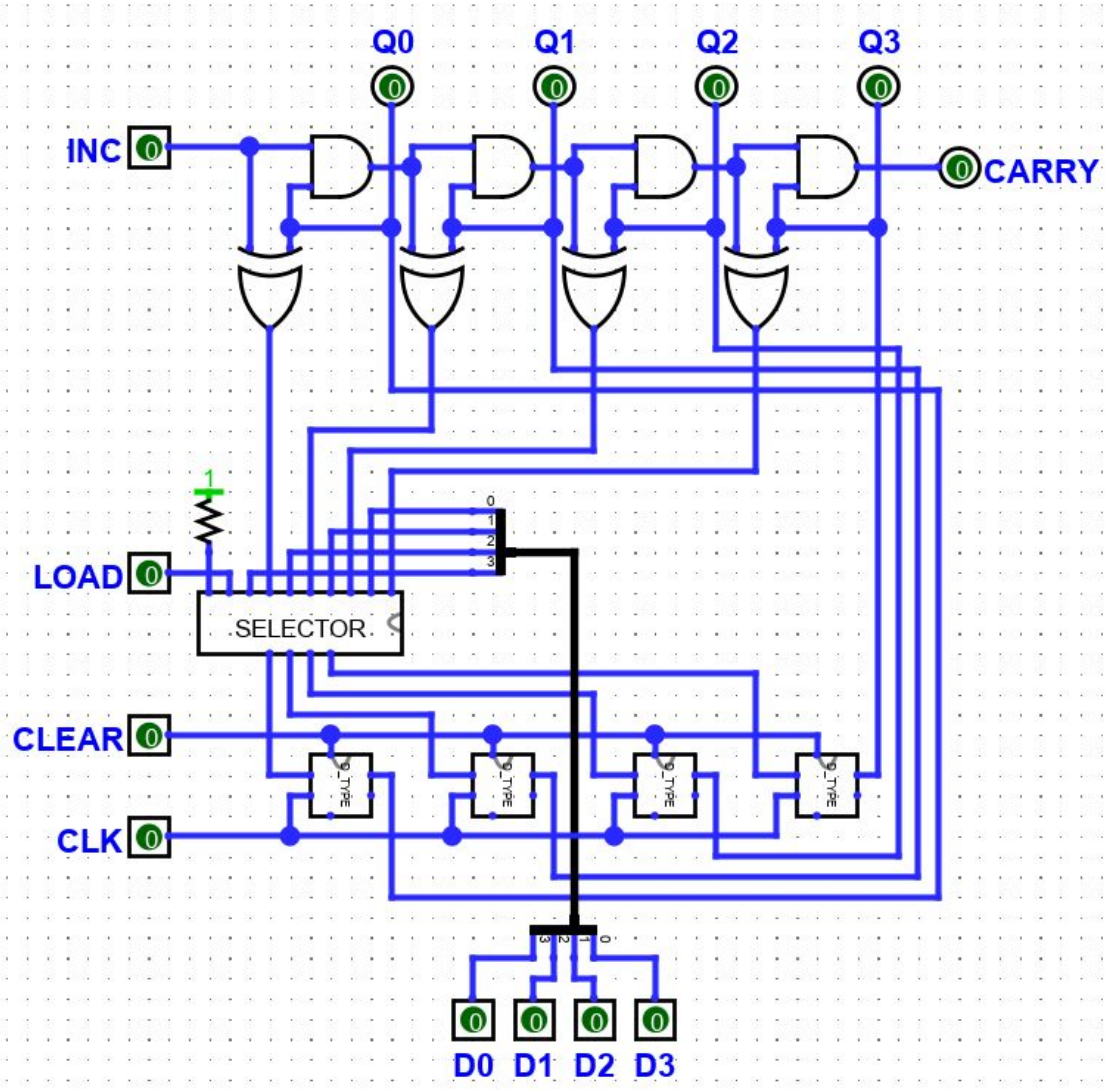
The common clock pulse ensures each flip-flop triggers at exactly the same time - i.e. synchronous as opposed to asynchronous This is important, because if each Clock was triggered by the output of the previous Flip Flop, there would be a ‘propagation delay’ between each output going high. This can cause problems in the CPU, as some numbers may be mistakenly read as others, and an incorrect address could be read from the Program Counter into RAM. The CLR are tied together for an asynchronous ‘reset’ back to 0 if needed. The AND gates and the XOR are configured so the INC toggle only allows the count to increase if pulled high. There

is an optional ‘CARRY’ output, which allows daisy-chaining of the counter to create implementations of N-bit width.

Four Bit Synchronous Up Counter with Load

Conditional ‘jumps’ back and forth between the otherwise sequential instructions in RAM will need to be implemented to allow loops in programs - so an option to load a predefined value into the counter i.e. address would be very useful, as shown in Figure 31 [fig:31].

4-Bit Synchronous Up Counter with Load[fig:31]

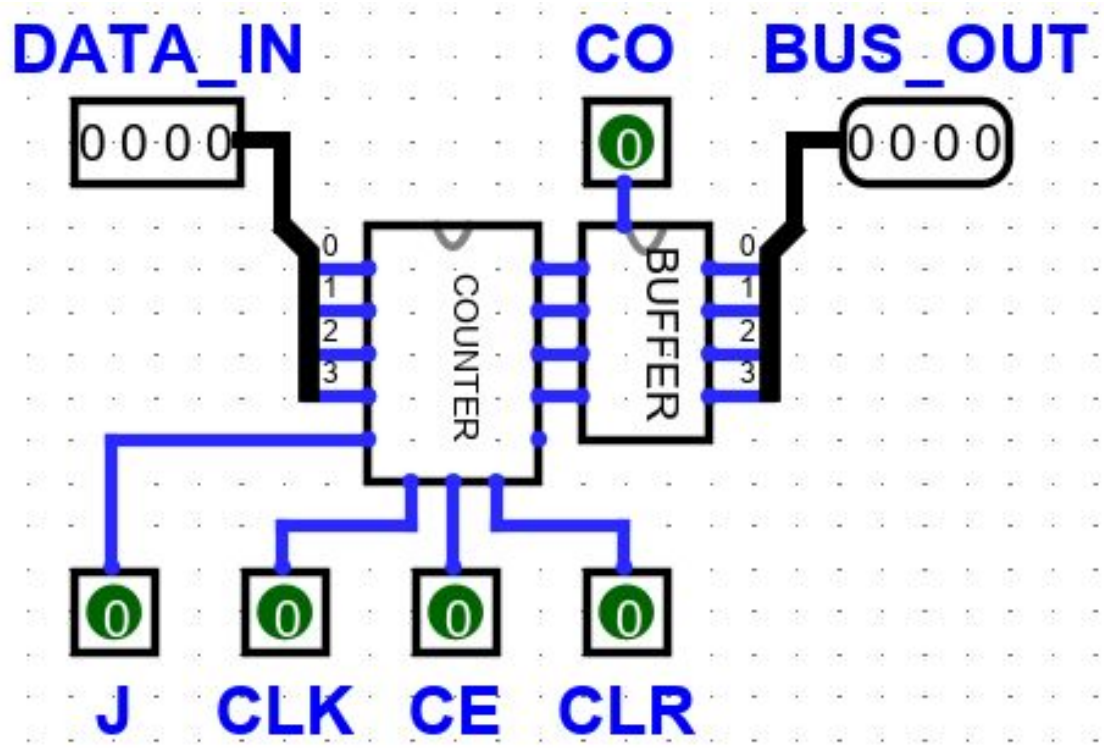


The circuit is identical, aside from a Four-Bit 2-to-1 multiplexer to select between the current count being fed into the D inputs, and the custom D0-D3 value to be loaded in. The ENABLE of the multiplexer is permanently pulled high so the output is always active, and the LOAD input is tied to the SELECT pin. The custom value will be latched to the output at the next rising clock pulse.

Summary

The counter simply needs to be run through a 4-bit buffer to isolate its output from the shared bus, as shown in Figure 32 [fig:32].

4-Bit Program Counter[fig:32]



The ‘LOAD’ pin is now named ‘J’, as the load function is used to jump to a certain value, and therefore RAM address, as described in the previous section. ‘CO’ is the buffer’s enable pin, and ‘CE’ stands for ‘Count Enable’, named ‘INC’ in the previous section. The ‘Carry’ output goes unused, as only 4 bits are required for the 16 RAM locations.

Control Logic

Microinstructions

Now all the functional units have been built, the CPU needs a way to be programmed. Each singular function in the CPU is known as a ‘microinstruction’, e.g. the ‘CE’ (Count Enable), ‘CO’ (Counter Output) and ‘J’ (Jump) functions described in the previous Program Counter section. Each microinstruction is activated by a high signal, and will perform its function on the next available rising edge of the common clock pulse.

Aside from manually enabling and disabling microinstructions in each subcircuit to facilitate a sequence of operations, the CPU can’t currently automate this process. This means a system needs to be constructed which can take a ‘command’ as an input, and then execute the required microinstruction(s) available to the CPU to complete the given command.

A full list of the available microinstructions constructed so far for this CPU:

Subcircuit	Microinstruction (by Pin Name)	Description
MAR	MI	Memory Address Register in
RAM	RI	RAM data in
	RO	RAM data out
IR	IO	Instruction Register out
	II	Instruction Register In
A Reg	AI	A Register in
	AO	A Register out
ALU	EO	ALU out
	SU	ALU subtract
B Reg	BI	B Register in
Out Reg	OI	Output Register in
PC	CE	Program Counter enable
	CO	Program Counter out
	J	Program Counter in (Jump)
Flags Reg	FI	Flags Register in

Opcodes & Operands

Given the list above, higher level instructions may now be formulated, made up of several of these microinstructions. The bus width of the CPU is 8-bits, therefore the amount of potential instructions is limited. As the RAM takes 4-bit addresses (by design), there needs to be at least 4 bits of the bus allocated to perform some operation on any given location in memory by the different subsystems of the CPU. This means the other 4 bits may be used for 16 unique ‘codes’ (0000 to 1111), each representing an instruction. These are known as ‘opcodes’, and span the upper 4 bits of each instruction. The lower 4 bits, which provide an ‘argument’ for the opcode, e.g. a memory address or immediate binary value, are known as the ‘operand’.

The Fetch-Decode-Execute Cycle

The main function of this (and most) CPUs is to continually take in an instruction, interpret and execute it, and repeat. This is known as the ‘fetch-decode-execute’ cycle. The ‘fetch’ references having the program counter output its current count to the memory address register, which then accesses that address in RAM and outputs its contents - the instruction - to the bus. The ‘decode’ stage happens at the control logic, which translates the opcode and operand into the signals required for each subcircuit to perform, in order to complete the current instruction. Finally, the microinstructions are sent from the control logic directly (not via the bus) to the relevant subcircuits to ‘execute’ the instruction. To keep the loop repeating (hence the ‘cycle’),

the program counter is simply incremented to sequentially access the next instruction in RAM, and the process is repeated. The microinstructions required for this cycle are as follows:

1. CO (Counter Out)
2. MI (Memory Address Register In)
3. RO (RAM Out)
4. II (Instruction Register In)
5. CE (Count Enable)

Note - the 'decode' step happens when the Instruction Register takes its Input, as the Control Logic will decode when the opcode is read directly from the Instruction Register. The 'execute' stage will then begin on the successive clock pulse(s). As several subcircuits are in use given this sequence of steps, and each microinstruction executes on the next available rising edge of the common clock pulse after being activated by the control logic, this means microinstructions, provided they are from different subcircuits, can be executed in 'parallel', i.e. on the same clock pulse:

1. CO, MI (Counter Out, Memory Address Register In)
2. RO, II, CE (RAM Out, Instruction Register In, Count Enable)

Therefore, this operation (fetch-decode) can be completed in two clock cycles.

Control Words & ROM

As mentioned, to achieve this behaviour, the CO, MI signals could be manually activated and the clock pulsed, then the RO, II and CE signals activated and the clock pulsed again. To formulate 'higher level' instructions by automatically executing these (or any) microinstructions given an opcode, a ROM can be used.

Given the 4-bit limitation, there are 16 unique opcodes available to define. Each of these can be represented by a location in ROM - specifically by the address. Each addressed location needs to be able to accommodate 16 bits - one for each microinstruction shown in the list above. This 16 bit value is known as a 'control word', and the CPU can execute one control word per clock pulse. Hence, the minimum required size of the ROM needs to be 16 locations * 16 bits per control word = 256 bits. However, this would only allow a single cycle of microinstructions per opcode i.e. one control word, and it has already been determined at least 2 control words are needed for the fetch and decode cycle.

It was decided at this point to define an 'instruction set' - that is, a list of opcodes, their desired overall function, and a list of sequential control words required to perform the function:

Opcode	Mnemonic	Desired Function
0000	NOP	No operation
0001	LDA	Load A reg with contents of given address
0010	ADD	Add contents of given address to A reg
0011	SUB	Subtract contents of given address from A reg
0100	STA	Store contents of A reg at given address
0101	LDI	Load A reg with given value
0110	JMP	Jump to given address
0111	JC	Jump to given address if Carry Flag is set
1000	JZ	Jump to given address if Zero Flag is set
1001	N/A	N/A
1010	N/A	N/A
1011	N/A	N/A
1100	N/A	N/A
1101	N/A	N/A
1110	OUT	Load Out reg with contents of A reg
1111	HLT	Halt the clock

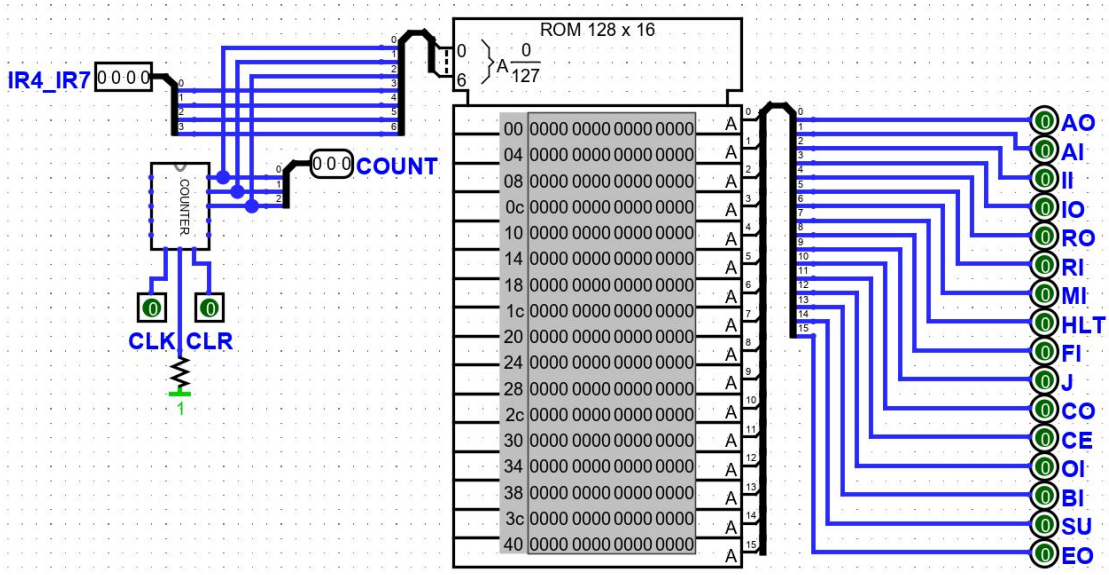
Note - Opcodes 0111 to 1101 remained unneeded and therefore unused for the overall function of this system. The relevant micro instructions can then be defined for each opcode, which allows the calculation of the maximum amount of cycles to accommodate all defined opcodes, simply by taking the length of the longest list of control words:

Cycle					Opcode	Mnemonic
1	**2**	**3**	**4**	**5**		
MI|CO	RO|II|CE	0	0	0	0000	NOP
MI|CO	RO|II|CE	IO|MI	RO|AI	0	0001	LDA
MI|CO	RO|II|CE	IO|MI	RO|BI	EO|AI|FI	0010	ADD
MI|CO	RO|II|CE	IO|MI	RO|BI	EO|AI|SU|FI	0011	SUB
MI|CO	RO|II|CE	IO|MI	AO|RI	0	0100	STA
MI|CO	RO|II|CE	IO|AI	0	0	0101	LDI
MI|CO	RO|II|CE	IO|J	0	0	0110	JMP
MI|CO	RO|II|CE	IO|J	0	0	0111	JC
MI|CO	RO|II|CE	IO|J	0	0	1000	JZ
MI|CO	RO|II|CE	0	0	0	1001	N/A
MI|CO	RO|II|CE	0	0	0	1010	N/A
MI|CO	RO|II|CE	0	0	0	1011	N/A
MI|CO	RO|II|CE	0	0	0	1100	N/A
MI|CO	RO|II|CE	0	0	0	1101	N/A
MI|CO	RO|II|CE	AO|OI	0	0	1110	OUT
MI|CO	RO|II|CE	HLT	0	0	1111	HLT

Programming the Control Logic

The highest amount of cycles required are for the ADD and SUB commands, which take five each. This means at least five control words need to be stored per opcode. Therefore, the ROM needs to be able to accomodate at minimum 5 control words * 16 bits per control word * 16 opcodes = 1,280 bits. Each ROM location can only store a single control word - so in order to facilitate several control words per opcode, the ROM was configured to act as a 'two dimensional array'. This can be achieved by feeding the opcode to the upper four address lines of the ROM, and using a counter to iterate over the required amount of lower bits 'inside' the 'outer' opcode address, shown in Figure 33 [fig:33].

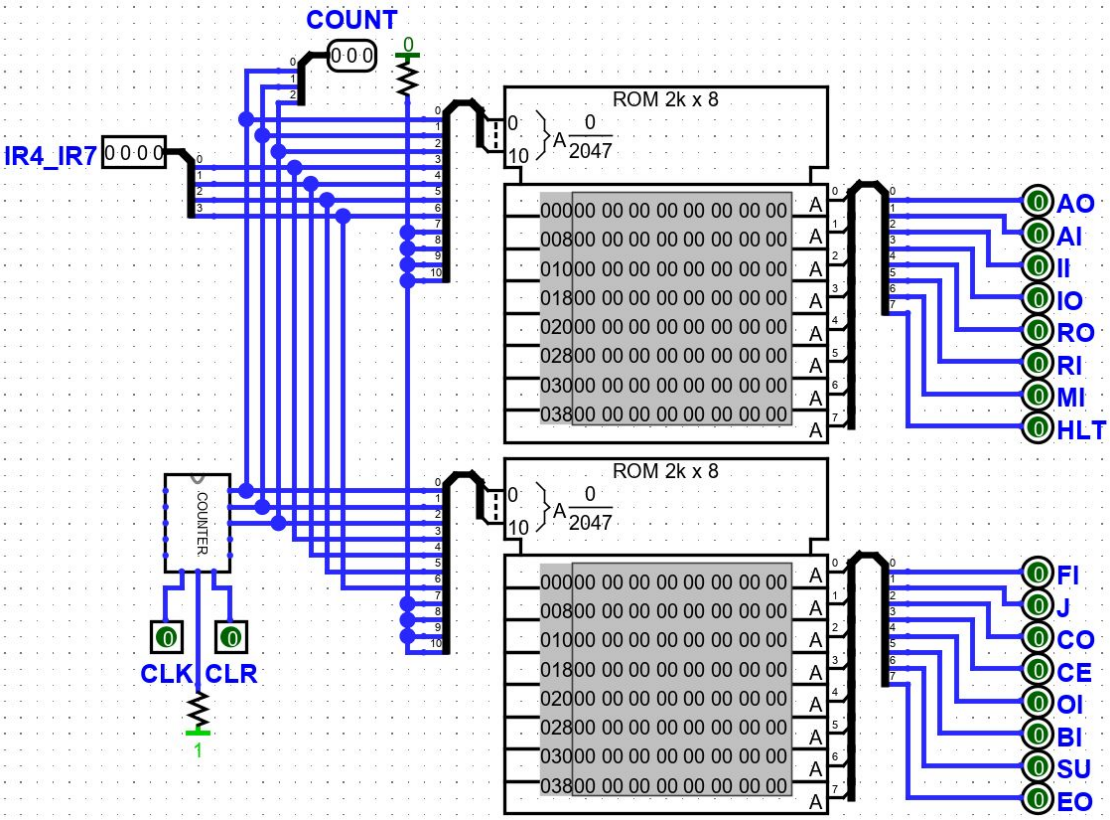
Control Logic ROM configuration[fig:33]



Using 3 lower bits to iterate over each location ‘inside’ the opcode, which is occupying the upper 4 bits, allows each opcode to effectively output up to 8 sequential control words. As the ROM is accessed asynchronously, the clock needn’t be pulsed to allow it to output. This means the clock can be fed instead into an up counter (in this case using the same synchronous 4-bit up counter designed for the Program Counter, which mirrors the operation of the 74LS161 IC chip - ’4-bit binary counter’), which will iterate over and output each control word for the opcode given via the Instruction Register input. The ‘COUNT’ output shows which control word is currently being executed.

For the purpose of aiming to use commonly available IC chips throughout this project, the actual implementation of the ROM was based on the 28C16 ROM IC - ‘16K (2K x 8) Parallel EEPROMs’. As each only has a data bit width of 8 rather than 16, two were used, as shown in Figure 34 [fig:34].

Control Logic ROM configuration using the 28C16 ICs[fig:34]



To avoid programming both identically and repeating data in order to produce the correct output, the control words will instead be split, and the upper and lower half separately written into the upper and lower ROM respectively. As a 4-bit opcode and 3-bit control word counter are all that's required for this system, the higher address bits are redundant.

The Control Logic ROM now needs to be programmed with the relevant control words for each opcode. Logisim Evolution allows a ROM file to be loaded directly into the ROM, with the following format: '*The first line identifies the file format used (currently, there is only one file format recognized). Subsequent values list the values in hexadecimal, starting from address 0; you can place several such values on the same line. If there are more memory locations than are identified in the file, Logisim Evolution will load 0 into the other memory locations*'. A Python program was written which can take the Carry and Zero flags as conditional inputs as well as the opcodes and their corresponding control words in binary, and produce a hexadecimal ROM file for Logisim Evolution, which can be found at my GitHub repository for this project.

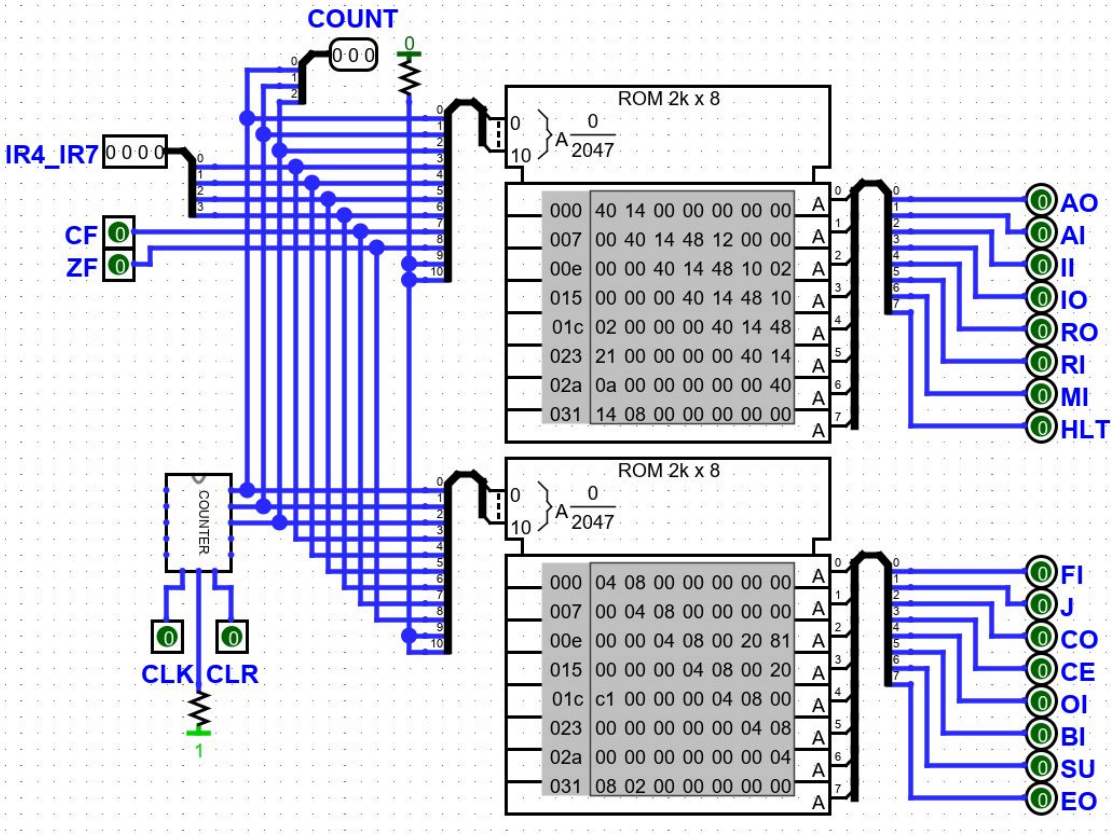
The control words are split between the ROMs in order to accommodate a 16 bit output, so two ROM files were produced of approximately the same size: 'upper' and 'lower'. As the Zero and Carry flags need to enable the ROM to act differently, i.e. jump if the flag conditions are met, there needs to be 4 copies of the ROM contents loaded in to accommodate each permutation of the two flags being set. For each, if CF = 1 then the JC (Jump if Carry) opcode was set to perform the control words for the 'Jump' command. The same was done for the JZ (Jump if

Zero) opcode, and for neither and both. The ‘nested’ address structure for the ROM input therefore was as follows:

```
[ZF][CF][OPCODE][STEP]
```

Where the Zero and Carry flag are a single bit each, the opcode itself is 4 bits and the ‘step’ or ‘count’ through each opcode’s control words is 3 bits, requiring a total of 9 input bits of the ROM as shown in Figure 35 [fig:35]:

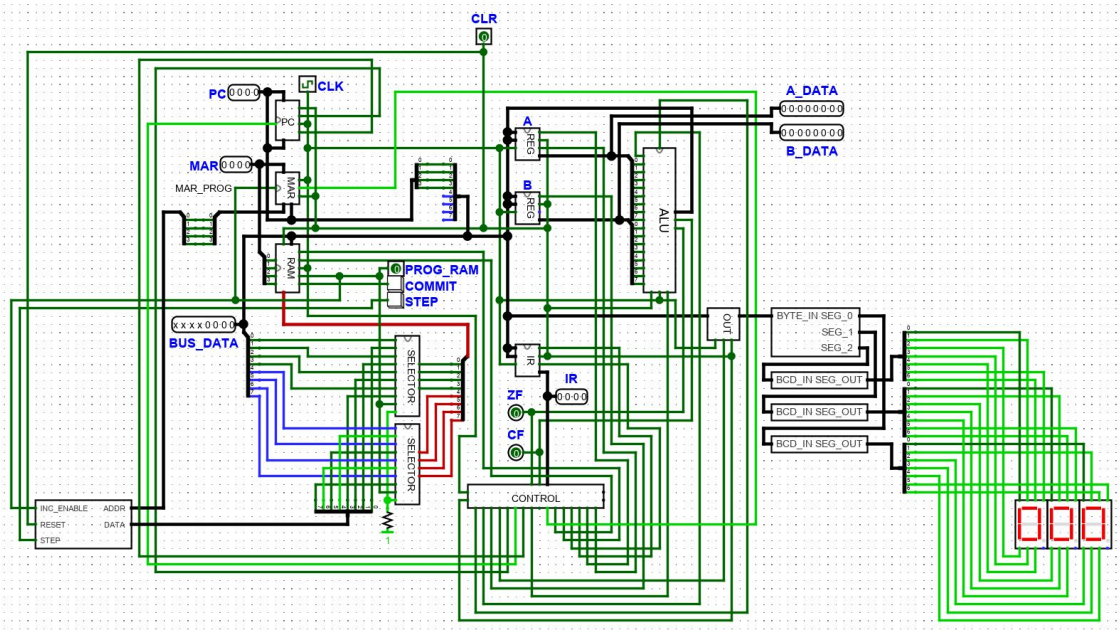
Programmed Control Logic[fig:35]



Running Programs

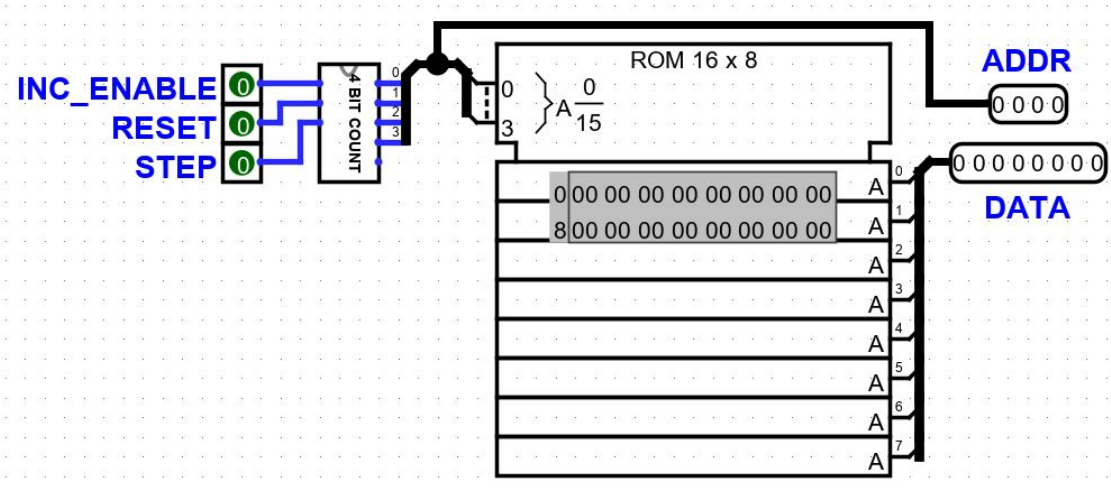
At this point, the control logic can be connected to all the inputs of each subsystem, and the common bus, clock and clear connected, shown in Figure 36 [fig:36].

The CPU[fig:36]



There are several additional outputs connected to view the current output from each subsystem. With the control ROM programmed with the relevant control words for each opcode, the RAM can finally be loaded with sequential opcodes and the clock started, where the fetch-decode-execute cycle will occur on each clock cycle, and the instructions executed. To facilitate loading programs into the RAM, a ‘RAM Programmer’ circuit was designed, as shown in Figure 37 [fig:37].

RAM Programmer[fig:37]



This is simply the Four-Bit Synchronous Up Counter used throughout the build attached to a 16 location x 8 bit ROM, to mirror the RAM dimensions. The counter steps through each location sequentially, and asynchronously emits the contained value when ’INC_ENABLE’ is held active and ’STEP’ is pulsed. As shown in xyzxyz, some extra inputs ’PROG_RAM’, ’COMMIT’ and ’STEP’ have been included in the CPU. The ’PROG_RAM’ input activates four things when high:

1. The RAM's 'PROG' pin, which will store the value given at the RAM's 'DATA_IN' input at the address given at the RAM's address input, when the RAM's 'TOGGLE' pin is pulsed.
2. The RAM Programmer's 'INC_ENABLE' pin, to prepare it to step through its memory contents and supply the RAM with instructions.
3. The MAR 's 'PROG' pin, to asynchronously load (and output) the value given to the MAR's 'DATA_IN' input from the RAM Programmer's 'ADDR' output.
4. The 'SEL' pin of both 4-bit 2-to-1 Multiplexers to switch the RAM's 'DATA_IN' from receiving its input from the BUS, to reading the 'DATA' output from the RAM Programmer.

Now, when the 'COMMIT' is manually pulsed which is directly wired to the RAM's TOGGLE pin, the value given the 'DATA' output from the RAM Programmer will be stored at the address given by the 'ADDR' output from the RAM Programmer, in the CPU's RAM. 'STEP' can then be pulsed to increment the RAM Programmer's next data value, and also increment the MAR's address which is being fed into the RAM. This allows fast 'writing' of programs from the RAM Programmer circuit into the discrete CPU RAM.

As Logisim Evolution supports loading ROM files into its ROM component, programs can be loaded into the ROM of the RAM Programmer, in order to then be programmed into the CPUs RAM for execution. The data flow can be described as such:

```
Sequential Binary Instructions > Hexadecimal Equivalents >
Logisim Evolution ROM File > RAM Programmer > CPU RAM
```

Another Python program was written to facilitate the parsing of binary instructions supplied in a .CSV file to a Logisim Evolution ROM File, which again can be found at my GitHub repository for this project (*genROM.py*). This small program was designed to be generalised with the view to open-sourcing, as many Logisim users use the built-in RAM rather than a discrete implementation. This built in RAM can be loaded with the exact same ROM file as the built in ROMs can.

Fibonacci

A program which can test all standard instructions, plus conditional jumps is a Fibonacci sequence generator. The program shown will generate the sequence up to the maximum value that the CPU can handle (233), as the 8-bit width cannot exceed 255. The program then loops and resets the sequence:

Address	Mnemonic	Instruction	Note
0x0	LDI 0X0	0101 0000	X=0
0x1	STA [0XD]	0100 1101	
0x2	OUT	1110 0000	Output X (0)
0x3	LDI 0X1	0101 0001	Y=1
0x4	STA [0XE]	0100 1110	
0x5	OUT	1110 0000	Output Y (Rest of the sequence)
0x6	ADC [0XD]	0010 1101	Z=X+Y
0x7	JC 0X0	0111 0000	
0x8	STA [0XF]	0100 1111	
0x9	LDA [0XE]	0001 1110	X=Y
0xA	STA [0XD]	0100 1101	
0xB	LDA [0XF]	0001 1111	Y=Z
0xC	JMP 0X4	0110 0100	

A pre-generated ROM file for this program can also be found at my GitHub repository for this project.