A Guide to Making an 8bit Breadboard Computer and the Basics of Computer Science

BY: PABLO LANZA

Table of Contents:

Introduction

Our modern everyday lives would be completely different, if not impossible without the invention of the computer. Just imagine having to go to the library looking through volumes of an encyclopedia, just to look up a simple piece of information that would've taken you less than a minute to google. And it's not just out laptops and PCs that are computers, everything modern, from our cars to our TVs and the ever-expanding internet of things, have some sort of processing computer that allow it to function properly.

Despite the omnipresence of computers in modern society most people live in ignorance regarding how they actually work. When asked the question of "how do computer work?" the majority of people would simply answer with an "I dunno". However, the truth is quite complicated, as there are billions of processes happening every instant making your computer display this very guide that you are reading.

In this guide I will hopefully clear up some of the mystery surrounding computers. And even though by the end of it you'll still be equally baffled as to how modern computers work, you will know the basic concepts behind computer science, and will grow a sense of respect towards the hundreds of engineers that made this marvel of modern technology possible.

Along with teaching you computer science, this guide will also show you how to build a computer from scratch. But don't get your hopes up, because while we will be building a computer in this project, don't expect the final product to be able to run Microsoft Word or your favorite game. No, this computer that we will be building can only perform very basic computational tasks like doing basic math, or displaying the Fibonacci numbers. These tasks could easily be programmed nowadays with only a few lines of code.

So, in that case, what's the point of this computer? Well, it is mainly a teaching tool used to show how a computer performs all the various tasks. By building it, you learn what each component in a computer does, how it works and interacts with the other components to output some data. Then, if you hunger for more knowledge you can scale up and learn how modern computers take the concepts taught in this guide, and inflate them to enormous proportions.

Lastly, don't be afraid of starting this project without any prior experience with electronics. By following this guide anyone with two hands, determination and patience, can build this 8bit breadboard computer. Through the guide, you will learn about computer science, and put the knowledge into practice by undertaking this project.

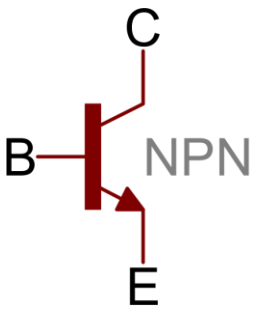
✦ ✦ ✦

The Very Basics

Transistors

As with all subjects, we must start from the beginning, or in the case of computer science from the smallest scales. Transistors are to a computer what cells are to an organism; they are microscopic devices that in unison to create a complex system. Nowadays transistors are smaller than 50 nanometers, that's smaller than the smallest thing visible using an optical microscope. Modern processors have billions of transistors inside and even our simple breadboard computer will have thousands of them (but we won't be using them separately).

So, what are these little transistors? Without getting into the physics of them, transistors are little switches that allow electricity to flow depending on an electrical input. Instead of activating the switch by hand, the transistor activates the switch with current. With the diagram on the right, if we have current flowing into B, then the transistor allows a larger current to flow from C to E. These transistors are essential for any computer as they can be connected to make much more complex and useful circuits. Of these circuits, logic gates are the most important.

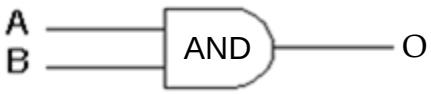


Logic Gates

Logic gates are circuits that take one or two binary inputs and return an output depending on what those two inputs are. This is called Boolean logic. These binary inputs are used throughout the whole guide and they will be referred to under different names (on/off, 1/0, HIGH/LOW) but they mean the same thing. The logic gates come in various flavors: AND, OR, NOT, NAND, NOR, XOR; each of these giving out a different output based on the combination of inputs. The output of the logic gate is dictated by series of transistors that are arranged to return the wanted output. Here we'll go over each type of resistor, explaining what they do:

<u>The AND Gate:</u>

The AND gate takes two inputs and if they're both turned on, it turns the output on as well. Otherwise, the output would remain off. Here is the symbol for the AND gate:

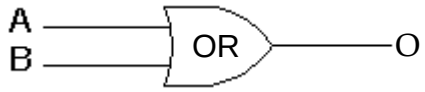


Input A	Input B	Output O
0	0	0
1	0	0
0	1	0
1	1	1

For this gate the output 'O' only turns on if current is flowing from both 'A' and 'B'. This can be expressed using the truth table on the, where a 1 represents current flowing and a 0 does not.

The OR Gate:

The OR gate takes two inputs and if one or both are turned on, the output turns on as well. If both inputs are off, then the output remains off. Here is the symbol for an OR gate:

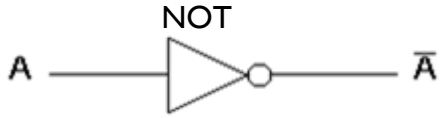


Input A	Input B	Output O
0	0	0
1	0	1
0	1	1
1	1	1

For this gate the output 'O' turns on if current is flowing from either 'A', 'B' or both.

The NOT Gate:

The NOT gate usually called an inverter only takes one input. This gate simply inverts the input, for example, if we have a 1 coming in, then we get a 0 as the output.
Here is the symbol for the NOT gate:



Input A	Output $\bar{A}$
0	1
1	0

In computer science a line above an input or output means the inverse or that it is activated when LOW. In this gate, if our input is A the output is the inverse of A, so $\bar{A}$.

The NOR and NAND Gates:

These two gates are less common as they are the combination of a AND or OR gate with a NOT gate. Simply put, they return the opposite of their original gate. The symbols for these two gates are the following:

Input A	Input B	Output $\bar{O}$
0	0	1
0	1	1
1	0	1
1	1	0

In this logic table we see that the outputs are the exact opposite as the ones for the AND gate. This gate outputs a 1 for every case except when the two inputs are on.

Input A	Input B	Output $\bar{O}$
0	0	1
0	1	0
1	0	0
1	1	0

The same case occurs for the NOR gate which is the opposite as the OR gate. In this case the output turns on only when neither inputs are on.

The XOR Gate:

The XOR gate, or exclusive OR turns on when only a single output is on. What this implies is that it outputs a 1 when one input or the other (but not both) are on. Here is the symbol for the XOR gate:

Input A	Input B	Output O
0	0	0
0	1	1
1	0	1
1	1	0

We can see that the logic table for the XOR is very similar to the OR gate's with the only difference being the last row where both inputs are on, so the gate doesn't turn on.

The Binary System

When we think of computers the images of 1s and 0s often come to mind. As we've seen in the logic gate section the 1s and 0s are used to represent the state (on or off) of a particular input or output. These 1s and 0s are also used to store any kind of computer data whether it be images, video, audio or text, all of which is achieved using the various computer components and a long sequence of 1s and 0s.

The binary system is a way of representing numbers using only two unique digits. This is helpful to have as a computer can treat these digits as little 'switches', representing a 0 when it's in one direction or a 1 when in the other direction. Computers use binary because the 'switches' can be easily controlled with current (using transistors).

So how does the binary system work? Binary is like any other system for representing numbers. To you, who has lived his life in a decimal centered world, the binary system may appear a little strange. To help us understand binary better, we often convert it to decimal, but keep in mind that the computer doesn't need to do that. Another advantage of using binary is that math is easier with binary and can be done with a series of logic gates (see ALU). As an example, let's take the number 27 in decimal and convert it to binary.

Decimal:

27

$2 \times 10^1 = 20 \quad + \quad 7 \times 10^0 =$

Binary:

11011

$1x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = 27$

$16 + 8 + 0 + 2 + 1 = 27$

As we can see in both systems, the individual digits are multiplied by some number and then added together. In decimal, these are the powers of ten while in binary these are the powers of two. In both cases we get the same answer, the difference comes in what values those numbers represent. The use of the numbers 1 and 0 can be confusing (10 in binary is 2 in decimal) so when using binary, it is equally as correct to represent the digits using ○ and ●.

✦ ✦ ✦

Component Overview

From transistors and logic gates we will now scale up to explain the different sections of the computer; this is computer architecture. The design of a computer: what it can do and its limitations, are all based on computer architecture and the specifications that are dictated by the computer's components. Inside a computer there are many components, these are the same basic ones that are found in the breadboard computer but they're repeated hundreds of times, are made orders of magnitude larger in terms of storage and memory and smaller in terms of physical space.

To make the components do work they each have their own control lines. When these lines are activated they let the component perform a certain action. For example, one of the control lines would be to output data from one component, so it would only be able to do so when the control line is activated. To run a program, the computer would activate the control lines in series, performing the instructions given.

Of the basic components the first is the clock. This is the computer's metronome and at each beat or tick of the clock an instruction is executed. Without this clock, there would be no order and multiple instructions would be happening at the same time, messing up data and the output of the program. However, this also means that the speed at which the computer runs, is set by the frequency of the clock. This is known as the clock speed which is usually expressed in gigahertz when talking about modern computers. Our computer's clock speed will be extremely slow compared to modern processors, but this won't matter because the programs that you write into this machine are very short (and if it's too fast you won't be able to see the steps involved in the computing process).

RAM is where the program, and any other data used by the program, is stored. Any computer program requires RAM to run with high end games and demanding applications requiring more than 8 gigabytes. In contrast, this computer will only be able to hold 16 bytes of memory (sixteen 8-digit binary numbers) but this could be expanded. This means that only 16 total instructions, including extra values to be used in the program, can be inputted into memory. While this is very little, it serves our purpose of seeing how computers work. It is important to know a computer's RAM is so that you can know what kind of programs it will be able to run.

Another component within the breadboard computer is the arithmetic logic unit, or ALU. In modern computers, these are incorporated within the processor. Here, is where all the math in the computer is done. In the case of this computer it simply adds or subtracts two binary numbers, but these operations can be repeated to perform multiplication and division through some clever Boolean logic. The values used to add or subtract are retrieved from the A and B registers in this computer.

Next, there are the registers. Again, modern computers have hundreds or even thousands of registers found inside the processor. Here, temporary data is stored to be sent to another component. For example, one of the registers will hold a value to then be sent to

the ALU which might then output this new value back into the same register. In our computer there are five total registers; these are:

A register - this is the first of two registers connected to the ALU. Data from memory is inputted in which can then later be added to the B register. The value from this register can then be outputted to the display or sent to RAM.

B register - this is the second register to be connected to the ALU. It serves the same function as the A register with the exception that the value stored in this register cannot be outputted.

Instruction register - as the name implies, this register holds the binary value of the current instruction. This value is then sent to the control logic to tell the rest of the components what to do to run the instruction.

Memory address register (MAR) – this register works together with the RAM to fetch the data stored in a certain location. In the breadboard register, the MAR holds 4bits of data which tell RAM what byte of data to retrieve. The more memory you have, the longer the addresses will need to be to accommodate said memory. The programs that will be inputted will be done so through the MAR and the RAM directly.

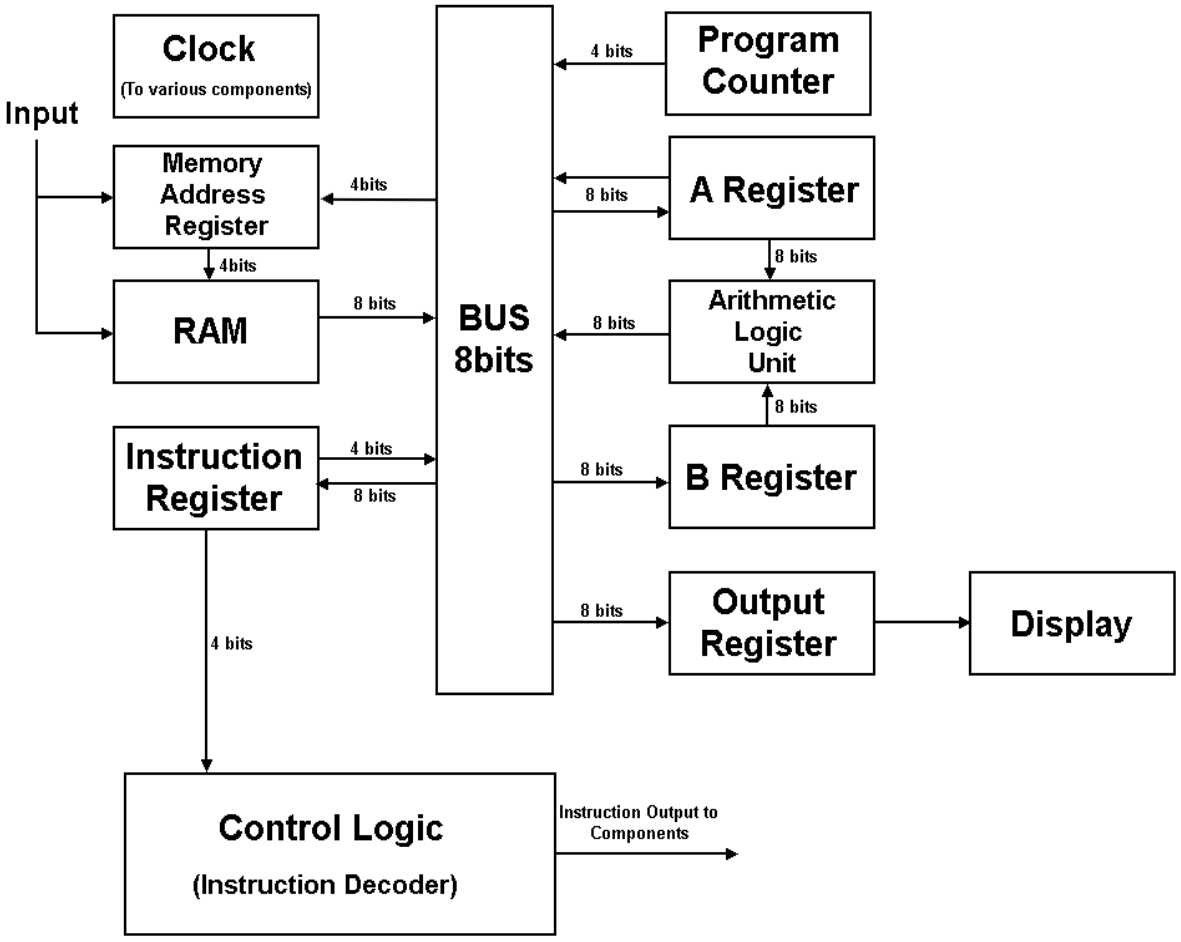
Output register – lastly, we have the output register. This register simply stores a value to then be outputted to the display. Since this register can only hold 8 bits of data in this computer, the computer can only output numbers up to 255 (eight 1s in binary).

The Program Counter is used to keep track of what instruction needs to be executed next. Since we have a maximum of 16 instructions due to the limitations in RAM our program counter counts in binary from 0000 to 1111. However, values can also be inputted into the program counter to skip forwards or backwards in the program to create loops.

The bus is the computer's highway as it transports data from one component to another. If we wanted to output a value from the ALU to then be stored in a register, the bus would be needed to transport the data. Instead of passing the data directly from one component to the other, it goes through the bus and then the second component picks it up and stores it. Having a bus is useful as it connects all the components together and allows all of them to access each other's data.

Finally, there is the control logic. This component takes the instruction given by the instruction register, and then tells the rest of the components what control lines to activate to complete that instruction. For example, if we needed to load a value from RAM into the A register the instruction decoder might say, "RAM, output data at address 0010 and put it onto the bus. A register, take the data from the bus and store it." Obviously, this wouldn't be done with words but instead through combinational logic, which takes all the possible inputs and gives the appropriate outputs.

Here is a basic diagram outlining how data is moved around the components.



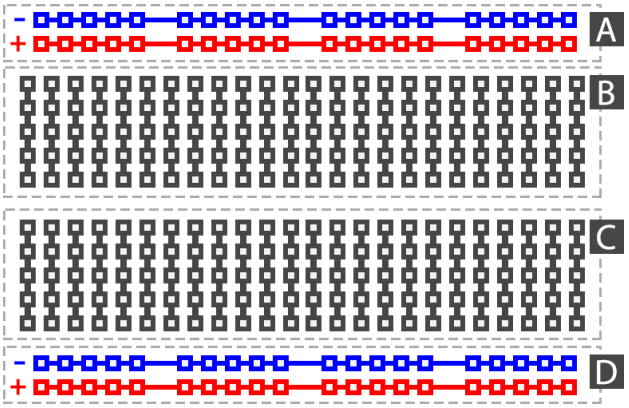
✦ ✦ ✦

Making the Computer

In the next few sections the guide will go one by one through every component in the computer and explain how it works and how to build your own. This type of computer is known as a 'breadboard' computer as it is built on breadboards. A breadboard is base in which you can connect various electronic circuits. They're most commonly used for prototyping and testing circuits out before soldering them. Breadboards are very useful since the only thing you need to build a circuit is the breadboard and the material for the circuit itself; no other tools or knowledge is needed. They're extremely safe, so no safety precautions are necessary except for the obvious "No water near electrical circuits".

However, there is a downside to these devices. While their safe and easy to use, the connections between wires can be finnicky, and for a large-scale project like this, it is essential to go over every connection carefully. Also, when hooking up all the wires, you often make a mistake and connect wires to their incorrect destinations. Most times nothing will happen, but if you mess up particularly badly a chip might heat up a lot or an LED could burn out.

The connections in the breadboard work as shown in the diagram. Sections A and D run horizontally, with every pin of each row making a connection. These rows are usually reserved for power and ground. Then, there is sections B and C running vertically. Notice that the columns are split in two, so no connections is made between the two sections. Here, the main circuit is built, where most devices are hooked up.



One last thing before getting on with the guide, is that when building the various components, many different chips are used. Some of the simpler chips won't be explained in detail, so to know what each pin does it is recommended that you look up the data sheet online.

✦ ✦ ✦

The Clock

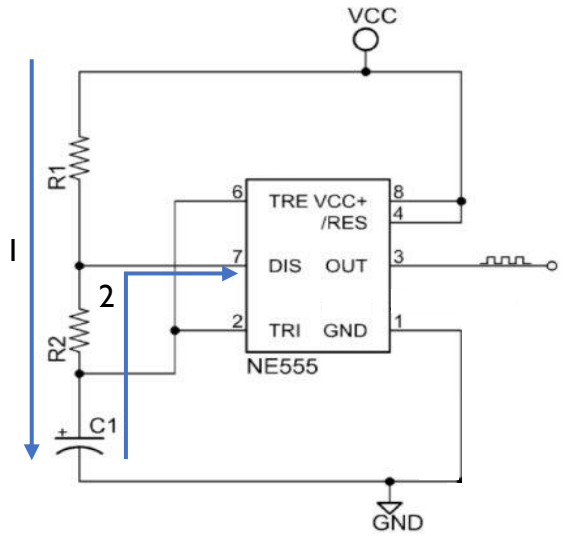
As was mentioned previously, the computer's clock keeps all the components in order, managing each instruction one by one. Without a clock, then the computer would get all sorts of errors: data would overlap on the bus or multiple instructions would be ran at the same time. Furthermore, since the clock ensures that all instructions happen step by step, then the speed at which the clock runs sets the pace for the rest of the computer.

For this computer we want the clock speed to be variable. Sometimes we would want to see what's going on, while other times we may just want to get the program's output, so, the clock speed should be able to change based on what the user wants. However, to truly see step by step what each component does then it would be best to just control the clock speed ourselves with the push of a button. In that case, the computer will have two clocks: an automatic one and a manual one.

The Automatic Clock

The way this clock works is by alternating the output voltage from high(on) to low(off) doing so in equal periods of time. If we were to graph this change in voltage we would see a series of square waves where the peak and trough of the wave is equal. As the voltage alternates from 0 volts to the clocks output voltage we get the following waves:

To generate these series of waves, the computer will take advantage of a chip which makes them automatically: the 555 timer. So, how does this timer take a continuous input voltage and turn it into a series of square waves? Here is a diagram of the circuit that the computer will use:



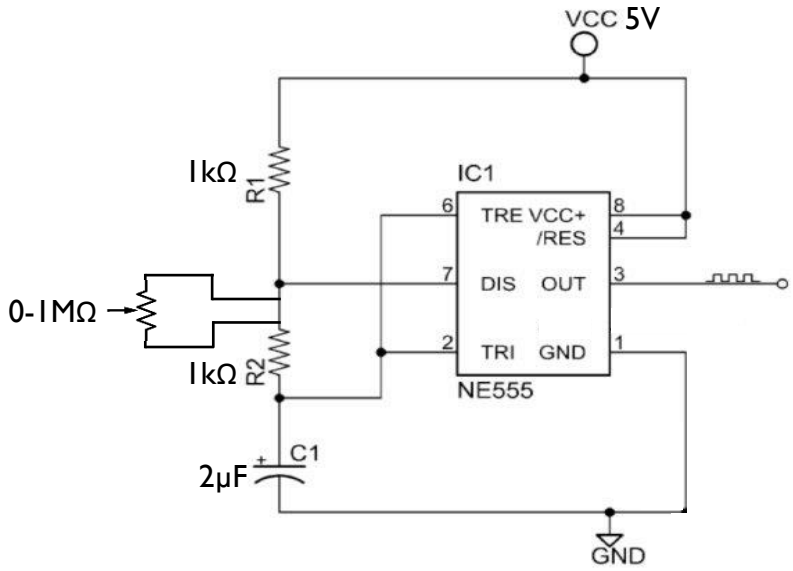
The first thing that happens once the chip is hooked up properly is, a low voltage that is less than a third of the supply (so in this case 1.67V) is applied for an instant causing the output pin to go high beginning the cycle. Inside the chip, there is an SR latch; this device simply holds the output which is HIGH due to the voltage applied previously.

Then, voltage (blue arrow 1) starts charging the capacitor C1 until it goes over the timer's threshold (pin 6) which is two thirds of the supply voltage: 3.33V. Once this happens the SR latch turns the output LOW and holds it. Apart from the regular output, inside the chip there is also an inverted output which goes HIGH at this point. This inverted output is connected to a discharge (pin 7) which discharges the capacitor C1 until it goes below 1.67V. Here we see the flow of electricity change as now it flows from the capacitor to the discharge (blue arrow 2). Once the capacitor discharges below 1.67V the capacitor is again charged up and the process is repeated, creating square waves as the output switches from HIGH to LOW.

Note that the output voltage isn't 5V volts because 555 timer limits it to 1.7V less than the input. Also, pin 4 is a reset pin that is activated when it is LOW, so it is set HIGH in the diagram.

The frequency between the changes in high to low is based on three factors, these are C1, R1 and R2. By varying C1's capacitance, it changes the amount of time that it takes it to charge up. The two resistors also limit the current passing which also slows down the rate at which C1 charges. Moreover, if you look at the diagram you can also see that when C1 discharges it must pass through R2 again, further increasing the time at which the capacitor discharges.

For our computer we want to be able to change the speed at which it runs, this means that we must have a way to change the clock's frequency. As we now know, by changing the R2's resistance we'll also see a change in the clock's frequency. Using a potentiometer which varies the resistance we can control the time which it takes for C1 to charge and discharge. Here is the previous diagram modified to show this addition:



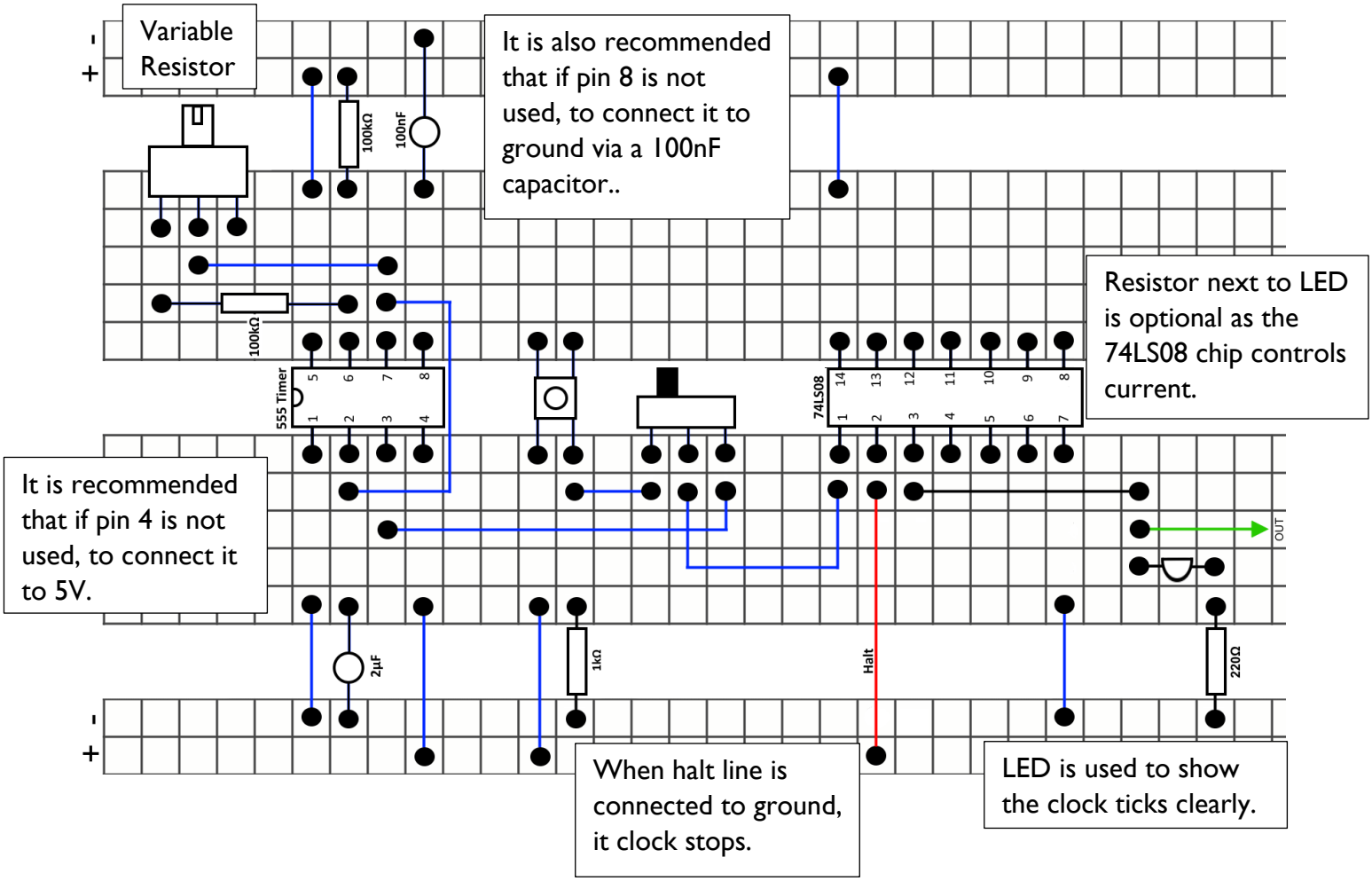
The Manual Clock

Next, the computer should also have a manual clock where you would step an instruction forward by pushing a button. This is much less complicated than the astable timer because it really is just connecting a button. The button works by letting electricity through when it is pushed down. If we were to connect the button to the clock's output, then the components would advance when the button is pressed.

Now we have two separate outputs for the clock, when we should only have one. Therefore, to switch between the two types of clock (automatic and manual) a two-state switch is used. This switch takes two inputs and based on its position it connects one of the inputs to the output.

Lastly, the clock needs a way to halt a program and stop the computer from running. To do this, the computer uses an AND gate through the 74LS08 chip. If the clock is on and the halt line is also on then the gate will let the output pass. Otherwise, if the halt line is brought LOW, then the AND gate won't let the clock's signal through and thus the clock will be stopped.

There is a minor problem when using the 74LS08, which is that when one of the gate's inputs isn't connected to either ground or power (so when the button is open), the AND gate would default to HIGH instead of the desired LOW. To fix this issue a pull-down resistor (10kΩ) is used with the button to allow electricity to flow when it is closed, and to connect to ground when it is open.



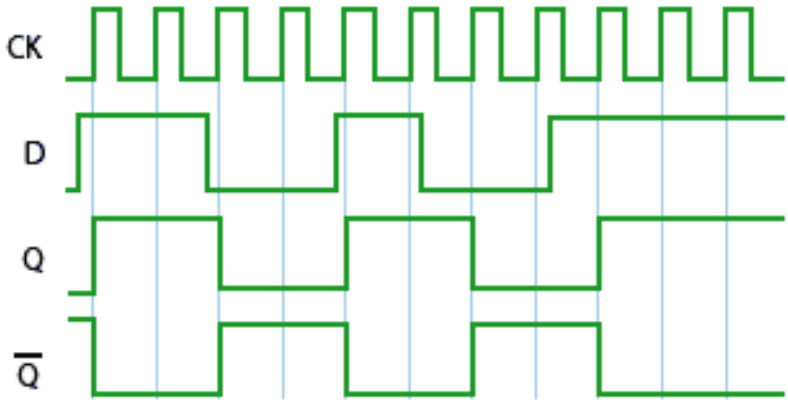
The Register

In any computer there are many registers, this one the five basic registers: the A and B registers to access the ALU; the instruction register to store the current instructions; the output register to hold values to be displayed; and the memory address register to access memory from RAM.

This component acts as fast memory that can be used to store values to be sent off to other parts of the CPU; in essence, registers are temporary data storages. In this computer, the largest register will only hold 8 bits (or a byte) of data, as this what most other components use. To be able to store a bit of data we need a device that can hold a value that is inputted when the clock ticks, and keep it there until another value replaces it. However, we also need a way to control when a value can be inputted, this is done through a control line. This so-far hypothetical device is called a D flip-flop.

The D flip-flop

D flip-flops hold a value (1 or 0) that is inputted through a data line when a different enable line is also activated. This little device also only allows data stored to be changed on the rising edge of the clock (so as it goes from low to high). This means that is the data line is changed in between clock cycles, the stored value won't change until the clock goes HIGH. Here is a graph showing how the flip-flop's stored value (Q) changes with the clock (CK) and input (D):



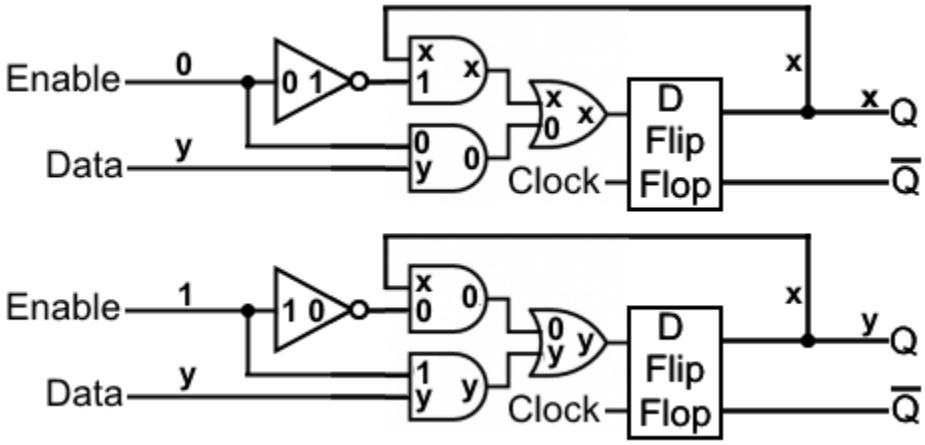
The clock oscillates between high and low.

D is the input going into the flip-flop.

Q is set to the same as D at every rise (marked with the blue lines)

Lastly, $\bar{Q}$ is always the opposite of Q

A D flip-flop isn't a full register though, because it at every clock pulse the data line would be set to whatever the data input is. So, an input enable line is needed that controls whether the value coming in should get loaded onto the flip flop. By creating a circuit with a D flip-flop and various logic gates we can create a register that updates its output on the rising edge of the clock, and only when the load enable line is on.

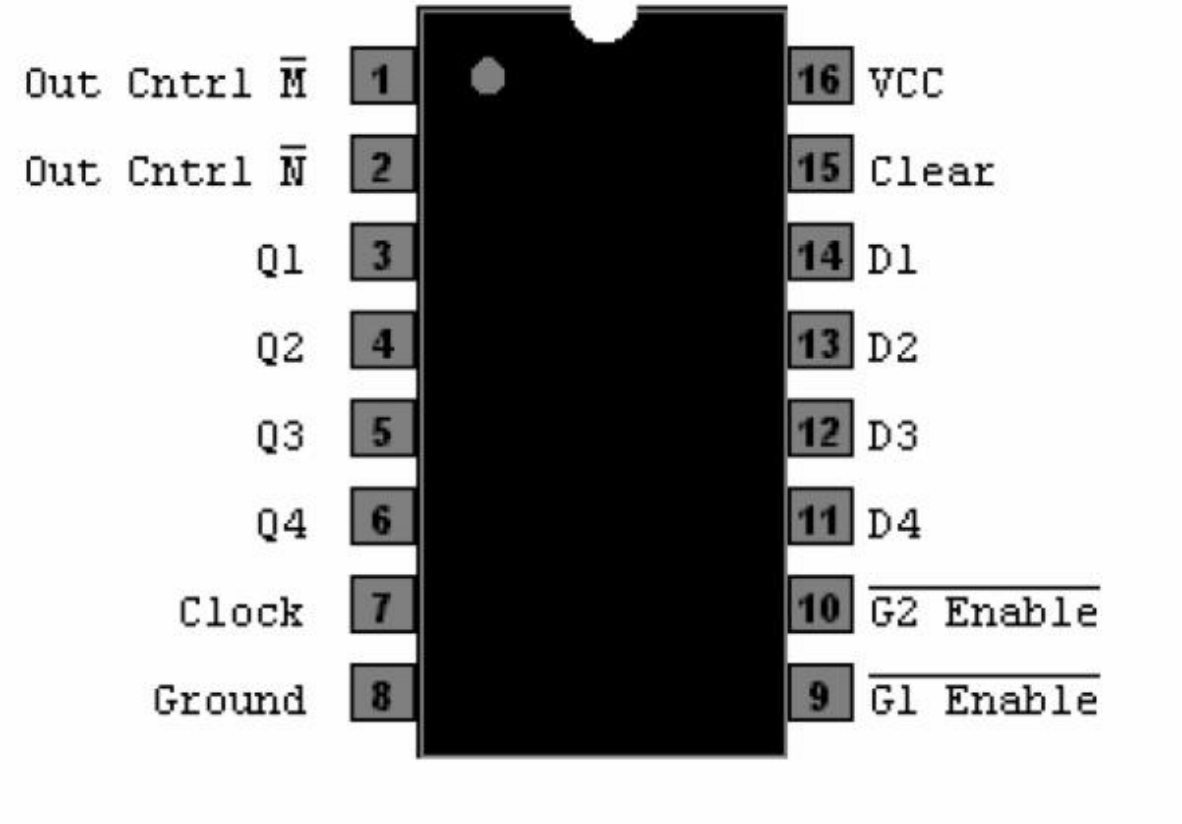


To show how this circuit works let's say that the enable line goes LOW: The bottom AND gate would output a 0 no matter what the input is as it's connected to the load. The top AND gate would also output whatever the previous output (Q) is. Thus, the OR gate would output whatever the top AND gate's result is because the bottom input is 0. Overall, there is no change to Q as that bit of data just gets cycled through.

If the enable line is a 1: The top AND gate would output a 0 no matter what Q is due to the not gate in front changing the 1 to a 0. Then the bottom AND gate returns whatever the input is. Next the OR gate outputs what its bottom input is, meaning that the input y has been transferred into Q.

This is the process that a 1 bit register goes through to output the correct value by setting its output the same as the data input when the load line is on. This 1 bit register is not very useful as most of our components deal with 8 bits at a time. To create an 8bit register we just link up these modules with the same load enable line and clock, creating 8 registers each with their own input and output.

To build an 8bit register, the computer will use the 74LS173 chip. This chip is simply a 4bit register so we'll have to use two of these in tandem. So here is how the chip works:



Pins 1 – 2. The output controls. We want to always have an output so that we can see what's inside the register at all times. Since these pins are inverted they should be connected to ground to be switched on.

Pins 3-6. The 4 data inputs. These will be connected to the bus and will be receiving data from it.

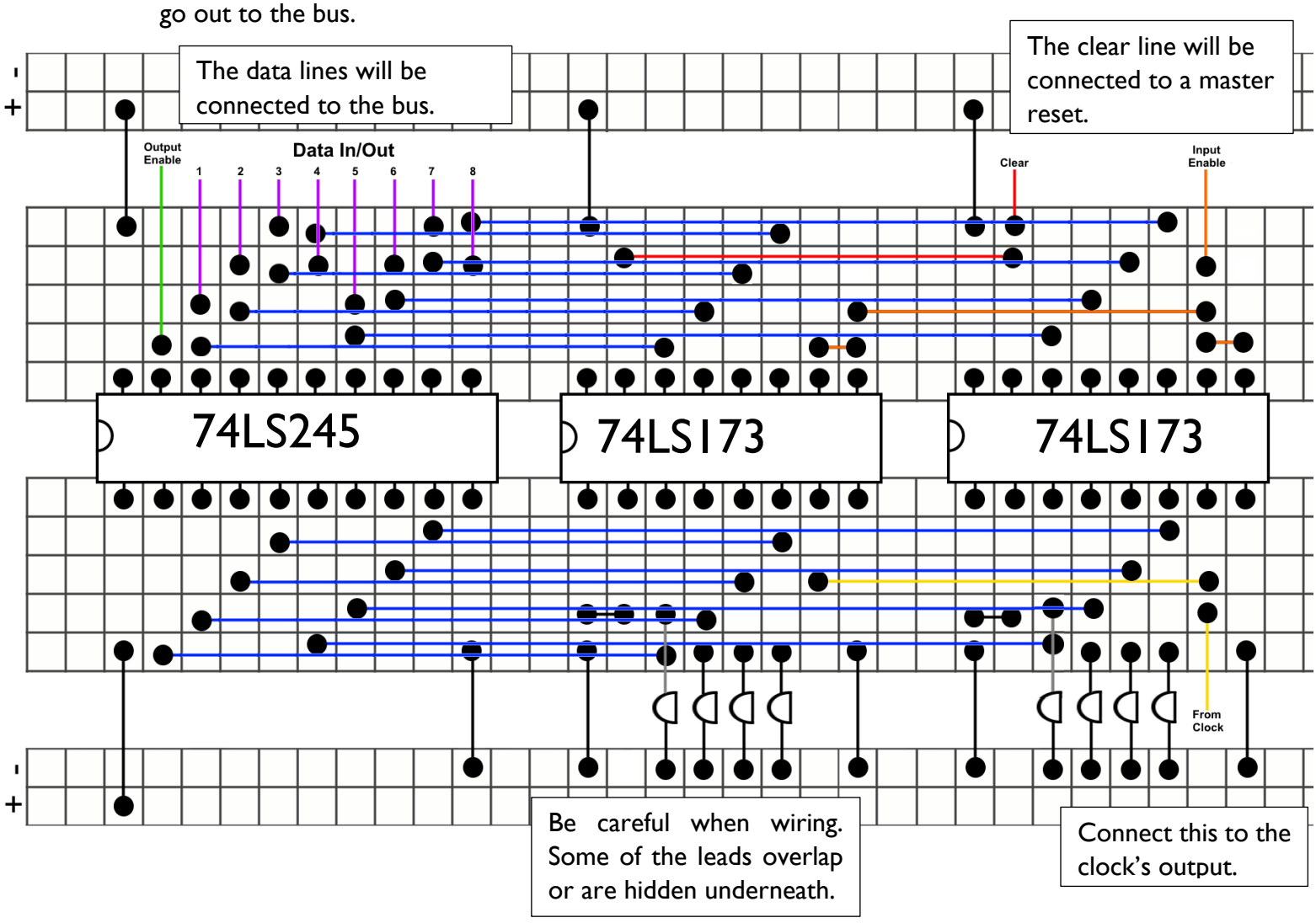
Pin 7. The clock input pin. This one will be hooked up to the clock's output to run the D flip-flops inside the chip.

Pins 9 – 10. The load enable pins. These pins allow the data to be written onto the chip. They will be connected together to a separate load line which will be used when setting up the computer's control logic. Since these pins are also inverted when this load line is low data can be written into the registers.

Pins 11-14. The output pins. They will output data from the registers into the bus through the 74LS245 chip.

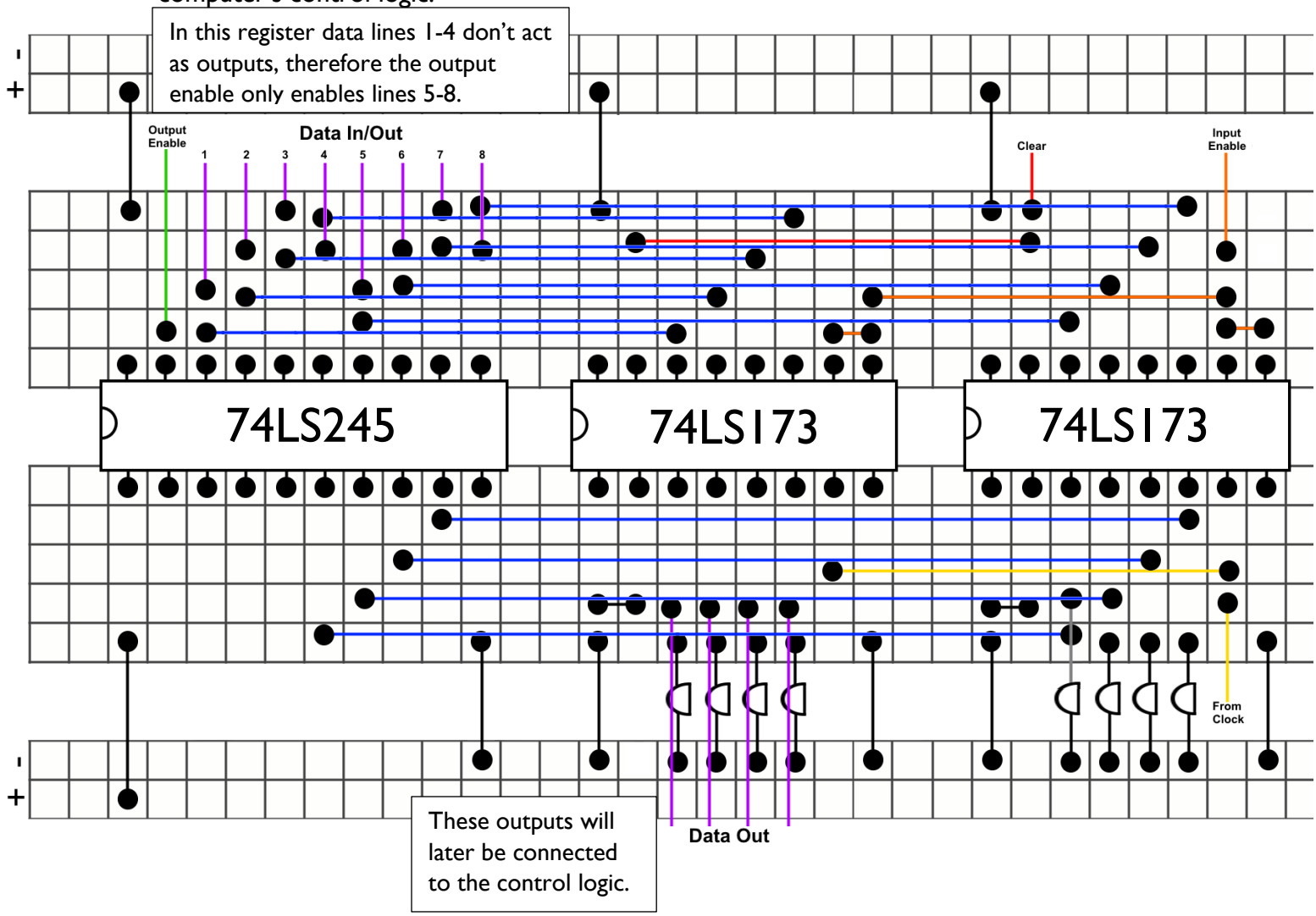
Pin 15. The clear pin. This pin erases all data stored in the pin. For now it will be connected to ground so that it doesn't clear the data when using the register, however, the computer will have a reset button which will be connected to this clear pin.

Lastly, to complete this circuit, one last chip is needed. As is mentioned above, the output enable lines will always be on to see the data inside the chip. This means that data will always be going onto the bus which could end up messing up the program. To prevent this from happening the 74LS245 (read the data sheet to what the pins do) is used, which only allows data to flow in one direction when an enable line is activated; this will be the registers' output enable control lines. So when this chip is not enabled, data from the registers won't go out to the bus.



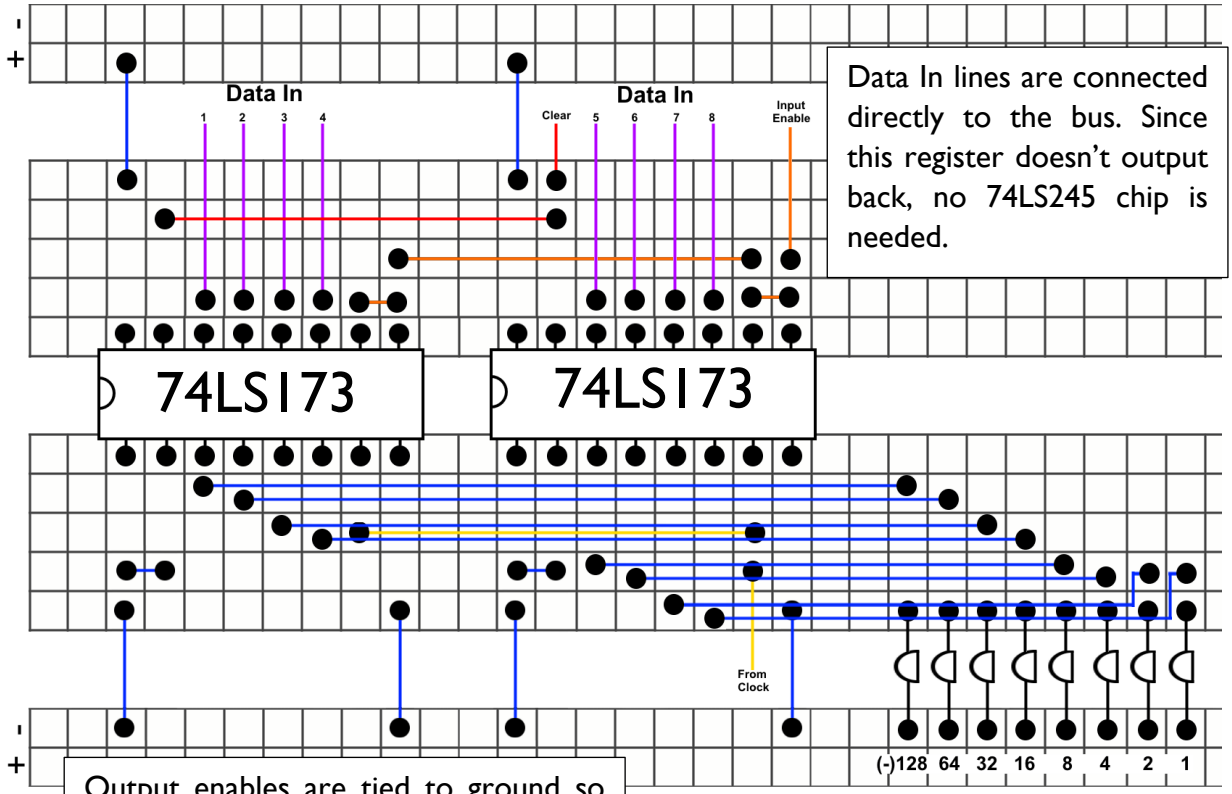
In the computer architecture overview section of this guide, I explained that there are 5 different registers. At this early stage, we can construct 4 of these 5 registers: The A register, the B register, the Instruction register and the output register. Both the A and B registers are identical to the one explained above. For now, they connect to the bus and input and output data from and to there.

However, the instruction register is slightly different, if we refer back to the original computer architecture diagram we can see that only 4 bits are output to the bus. The other four are being sent to the control logic module. This means that this register will have a slight difference, instead of connecting all 8 outputs to the bus we only connect the least significant (or right most) bits. The other 4 bits will later be used when constructing the computer's control logic.



Lastly, we have the output register. This register only takes inputs as it doesn't need to output any data back to the other components. Instead, this register is used to store the data that the user interacts with (e.g. the answer to a calculation). If the program inputted was to be the sum of two numbers, then the answer would be stored at this register and then shown in the display for the user.

`This computer's display sadly isn't very user friendly; there is no monitor or LCD display. To keep this project as simple yet as informative as possible, the computer will use 8 LEDs to represent an 8bit binary number. The display can be changed if you want to or even connected to a microcontroller to convert the outputs into a decimal number



Data In lines are connected directly to the bus. Since this register doesn't output back, no 74LS245 chip is needed.

Output enables are tied to ground so that it always outputs to the LEDs.

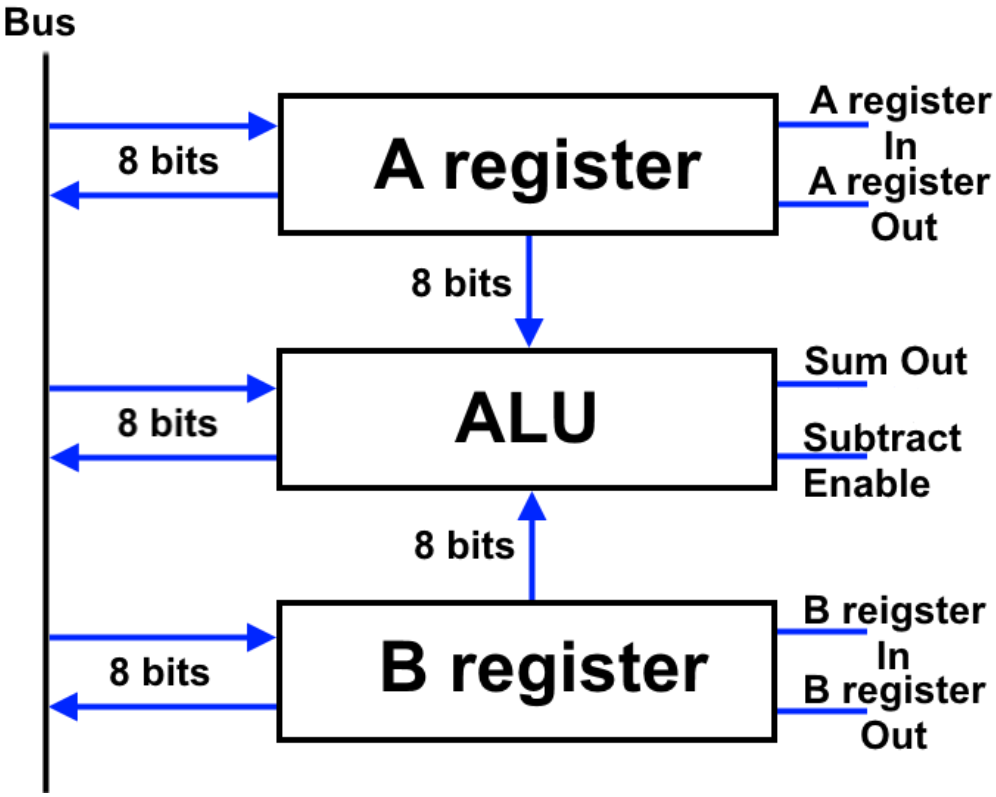
The "display". Each LED is labelled to show their value.

✦ ✦ ✦

The ALU

The ALU (Arithmetic Logic Unit) is the component in a computer that does all the math. Mostly, this component simply adds or subtracts two numbers, but these operations can be repeated to do more complex arithmetic. In modern computers, most ALUs are integrated within the processor and are capable of doing complex operations through clever programs.

For this component, we will use chips called adders which take two values and returns their sum. These two values will be coming in from directly from the A and B registers. Technically, the ALU will be adding the values in the registers all the time, but we won't be using the added value unless we want to. However, we can also set these chips to subtract two numbers through some clever use of binary logic. Therefore, we need our ALU to have a control signal that changes its mode from addition to subtraction. Lastly, we need another signal that enables the ALU's output as we also don't want to have it output data onto the bus all the time. With these parameters set, this is the basic design of the ALU:



Binary Arithmetic

Before building the ALU we should know how it works. As is mentioned above, the ALU makes use of chips called adders, so how does an adder take two 8bit values and add them together. The answer is of course through logic gates. First, let's start simple, how do you add two separate binary digits?

Well, it's quite simple: `0 + 0 = 0` These are all the operations with two 1-digit binary numbers

`0 + 1 = 1`

`1 + 0 = 1`

`1 + 1 = 0` Here the 1 is carried over to the next digit

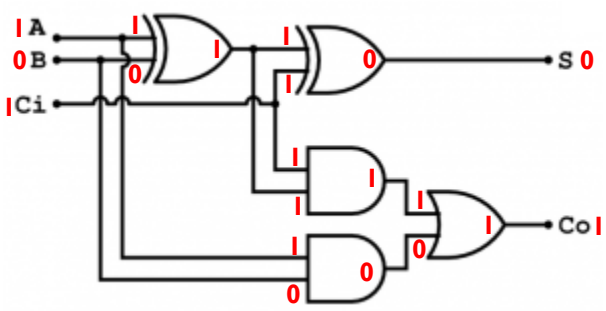
With this let's add two more complicated numbers:

```
  1 1
 111110  (30 in decimal)
+10101   (21 in decimal)
--------
110011   (51 in decimal)
```

Here we see the same operations as above, just repeated and using the numbers that are carried over

To figure out how to construct a circuit with logic gates that adds up two 1 digit binary numbers, it helps to see all the possible combinations of two bits that can be added:

Ci	A	B	Sum
0	0	0	0 0
0	0	1	0 1
0	1	0	0 1
0	1	1	1 0
1	0	0	0 1
1	0	1	1 0
1	1	0	1 0
1	1	1	1 1



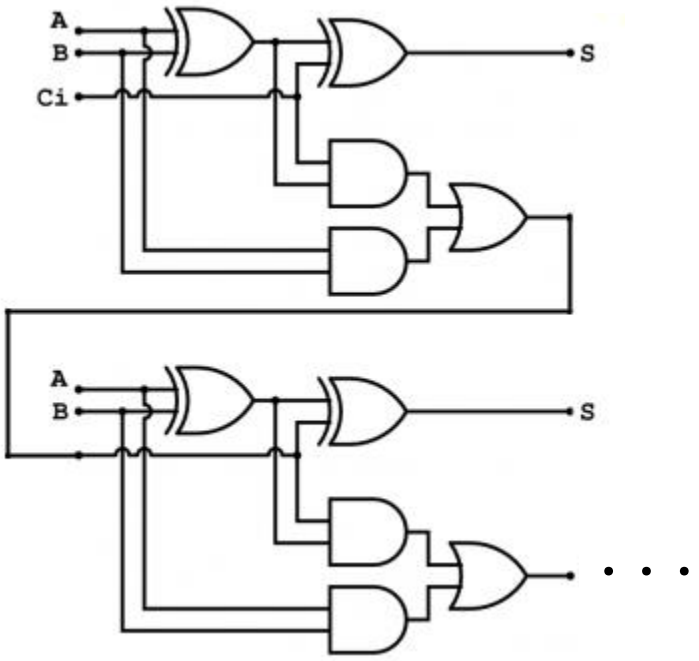
The above diagram shows the logic circuit of a full bit adder where: A and B are the two inputs; Ci is the carry coming in; S is the sum of the two inputs and is shown in the table as the first (right) digit of the sum. Co is the carry coming out and is shown in the table as the second (left) digit of the sum. Here's how this circuit works:

First, we need to look at this as two separate operations, one to find the sum digit and the other to find the carry out digit. For the first 4 cases, Ci is a 0, we can see that the sum digit here is the same as the XOR gate's logic table (the top-left XOR gate). Meanwhile, the output carry digit follows the same logic as an AND gate (the bottom most AND gate) so we connect one to the inputs.

The next 4 cases have the carry digit as a 1. Here, the sum digit is simply the inverse of the previous four digits. With XOR logic, if one input is a 1 then the other will end up being inverted as the output, otherwise the second input remains unchanged. So in this case, if the carry in is a 1, then whatever comes out of the first XOR gate will be inverted in the second XOR gate, giving us our desired results. The output for the carry digit gets a little more complicated. In our table we see that when A and B are the same, the carry digit is also the same; this is the same as in an AND gate, so we can use the one that was used for the sum digit. However, when A and B are not equal the carry digit is always 1; this logic

can be interpreted with first XOR gate used. We also need a second AND gate because we only want this output when the carry is a 1 as well. This is shown in the circuit diagram above with an example where the numbers in red is the data inputted processed to give the correct answer.

This diagram above shows a 1bit adder because our inputs are both just one bit. In our computer, we use 8bit values, so we need an 8bit adder. An 8bit adder can be created by linking up multiple of these 1bit adders, connecting the Co output to the next adder's Ci input like so:



Getting the ALU to Subtract

The 8bit adder isn't the complete ALU that we want as it also needs to be able to subtract two numbers. In math, subtraction is the same as addition with one of the numbers being negative, therefore we don't need a new circuit dedicated for subtraction. What we do need however, is a way of expressing negative numbers. The way that most computers do this is by taking the two's complement of a binary number.

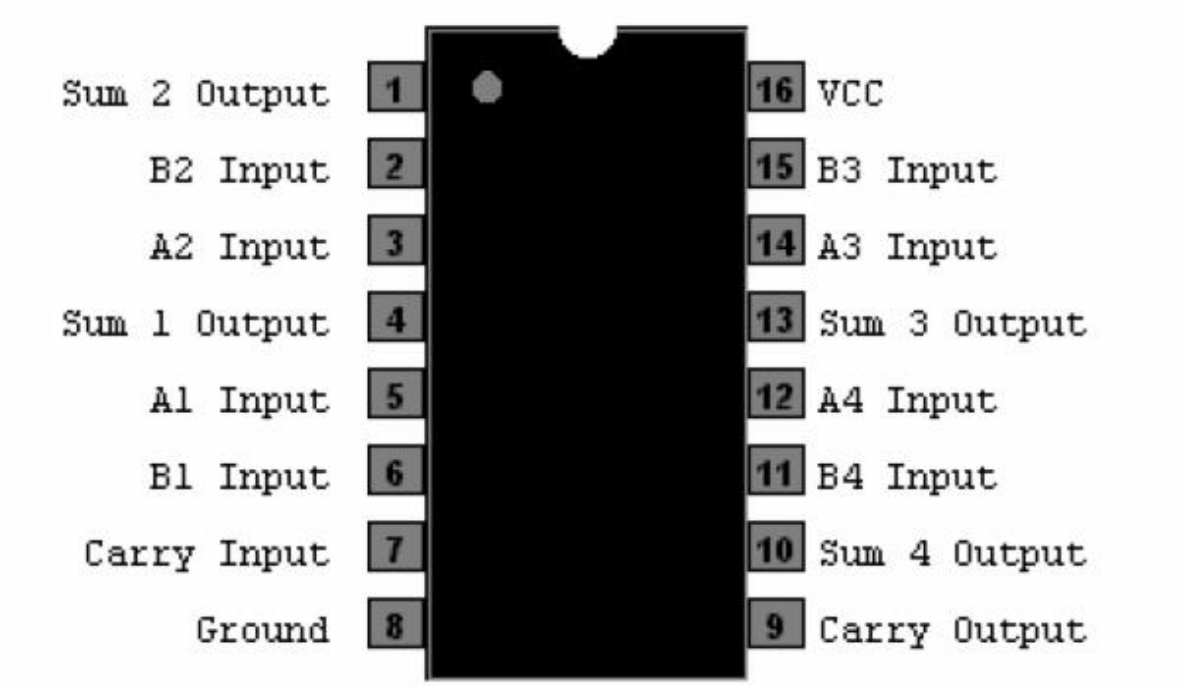
The two's complement of a number is it's inverse plus 1. For example: 0101 becomes 1010+1=1011. In two's complement the most significant bit acts as the sign bit meaning if it's a 1 then the number is negative and when it's a 0 the number is positive. The reason computers use this method for representing negative numbers is because it's the simplest method that works with the arithmetic explained above.

To better show how this works, let's say we wanted to subtract 11 from 25, in decimal this would be 25 +(-11) = 14. Now if we were to do this is binary it would be 011001 + the two's complement of 01011 where the first digit is the signed bit. To do this calculation we first need to convert 11 into its two's complement, so 01011 becomes 10100+1=10101.

Then we do the normal binary addition so to give us 11001 + 10101 = 101110. If we get rid of the last carry bit, we end up with 01110 which is 14 in binary.

Building the ALU

For the ALU, the computer will not use separate AND, XOR and OR gate, instead it will use the 74LS283 chip which holds a 4bit adder. Two of these chips will be linked to get the full 8bit adder.

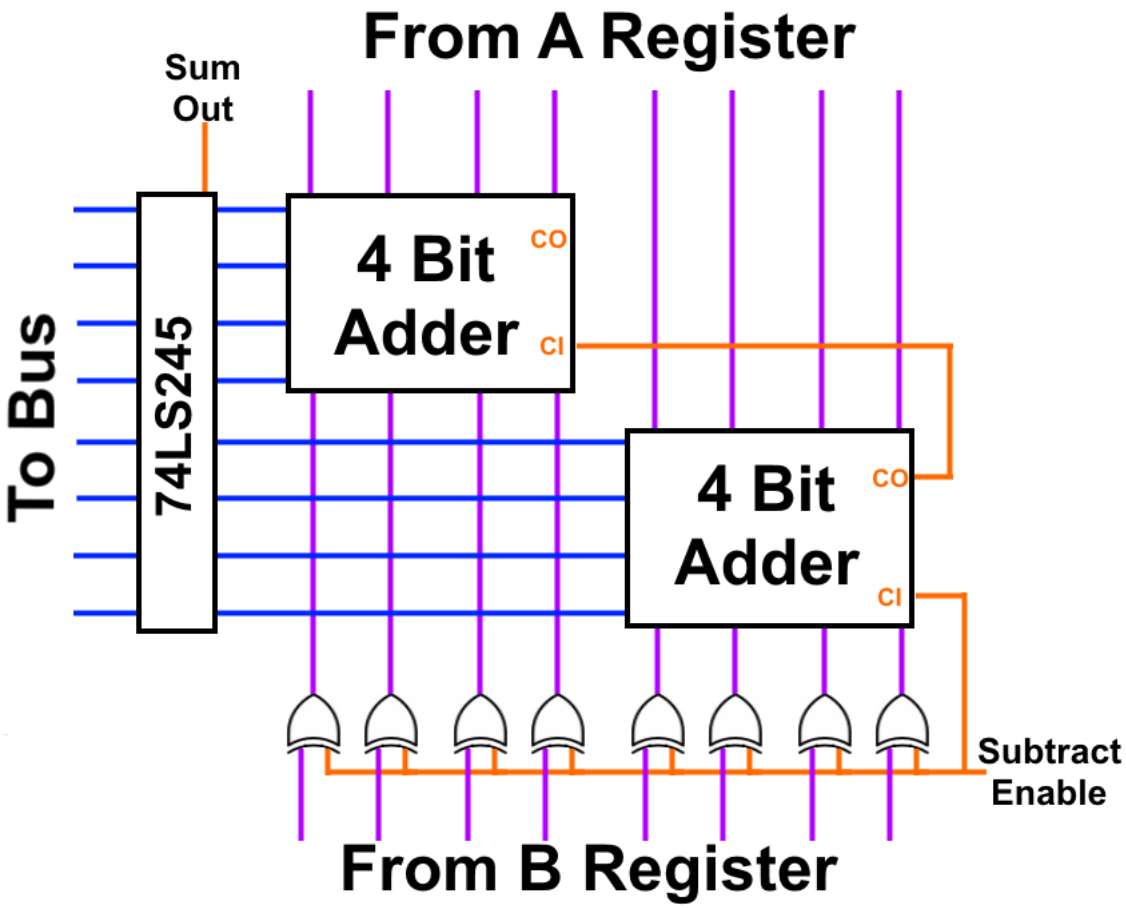


Pins 2,3,5,6,11,12,14,15. The input pins. As you can see on the schematic they come in pairs as they each add up to their respective output pins (e.g. A1 + B1 = S1). These pairs of pins will be connected to the values coming in from the A and B registers.

Pins 1,4,10,13. The output pins. They are the sums of the input pins. For example, pins 5 and 6 are the inputs that add up to the output at pin 4. The output pins will be connected to the bus through a 74LS245 chip which was also used to buffer the register's outputs.

Pin 7. The carry input. Here is where the carry from the first chip will enter into the second chip. This pin will also be used when subtracting two numbers because since we need the two's complement, the ALU would add 1 to the beginning of every subtraction through the carry in.

Finally, there is pin 9 which is the carry out pin. This pin is only used to transfer the carry bit onto the second adder chip.

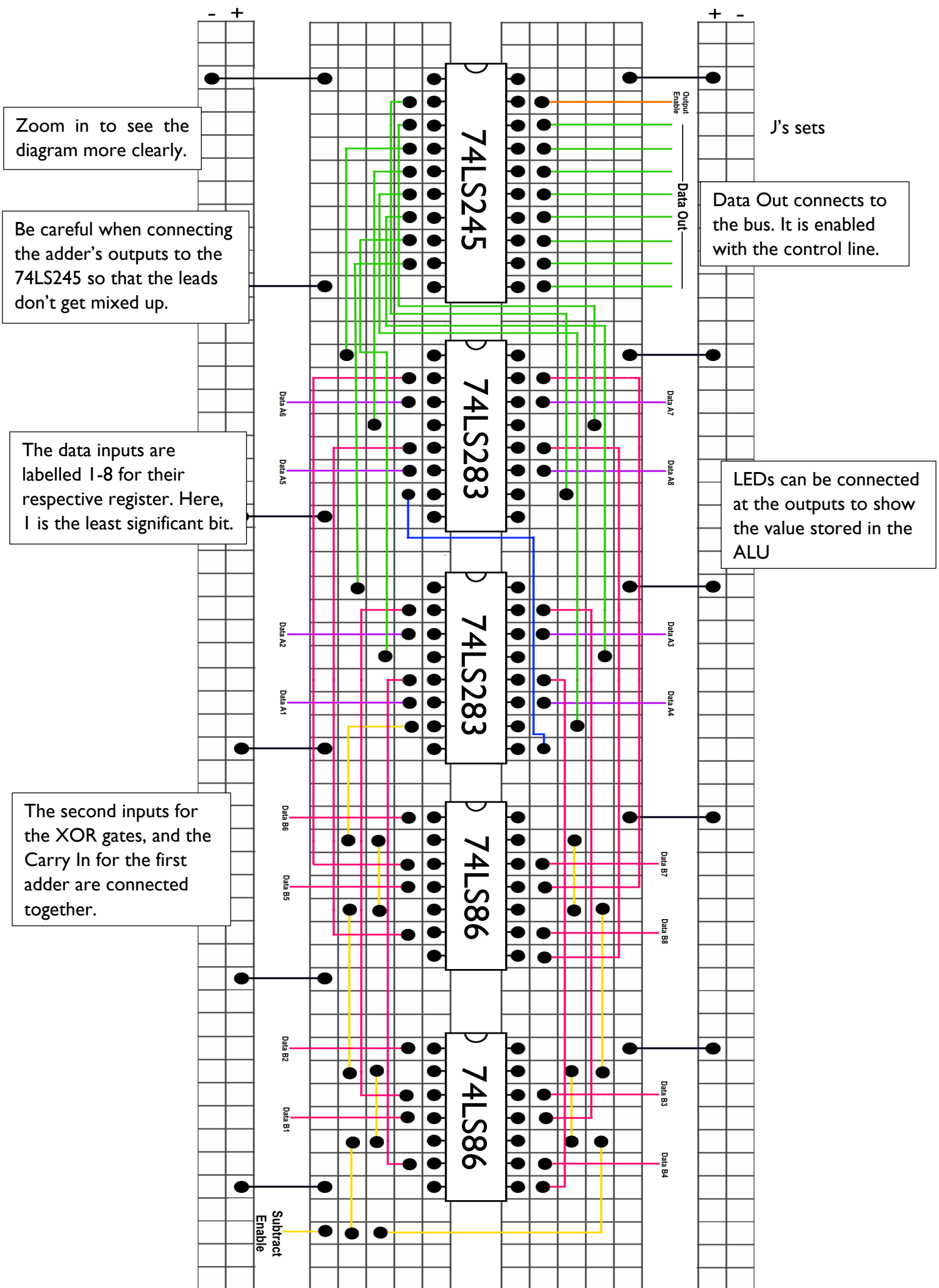


Using this chip, a basic diagram of the ALU can be drawn. In this diagram, the XOR gates are used to convert the B register's outputs into two's complements when the subtraction control signal is on. To show why XOR gates are used we need to look at the gate's logic table:

Input A	Input B	Output O
0	0	0
0	1	1
1	0	1
1	1	0

Say input A was the subtraction control signal and input B was data coming from the B register. If input A was off, then the output would remain the same as input B. However, is input A was on then the output would be the inverse of input B. So, the XOR gate gives us the inverse of the data coming from the B register only when the subtraction signal is on since it´s hooked up to all the gates. Otherwise, the B register data remains unchanged.

This subtraction signal is also connected to the first adder's carry input. This is done to add 1 to the value only when this signal is on, giving us the two's complement; an inverted number + 1 from the carry in. The XOR gates that will be used in this module can be found in the 74LS86. This chip includes 4 XOR gates, so 2 chips will be needed in total. One last thing to note is the 74LS245. This chip buffers the ALUs outputs, giving the computer control over when the ALU can output data.


Zoom in to see the diagram more clearly.
Be careful when connecting the adder's outputs to the 74LS245 so that the leads don't get mixed up.
The data inputs are labelled 1-8 for their respective register. Here, 1 is the least significant bit.
The second inputs for the XOR gates, and the Carry In for the first adder are connected together.
J's sets
Data Out connects to the bus. It is enabled with the control line.
LEDs can be connected at the outputs to show the value stored in the ALU
74LS245
74LS283
74LS283
74LS86
74LS86
Output Enable
Data Out
Data A6
Data A7
Data A5
Data A8
Data A2
Data A3
Data A1
Data A4
Data B6
Data B7
Data B5
Data B8
Data B2
Data B3
Data B1
Data B4
Subtract Enable

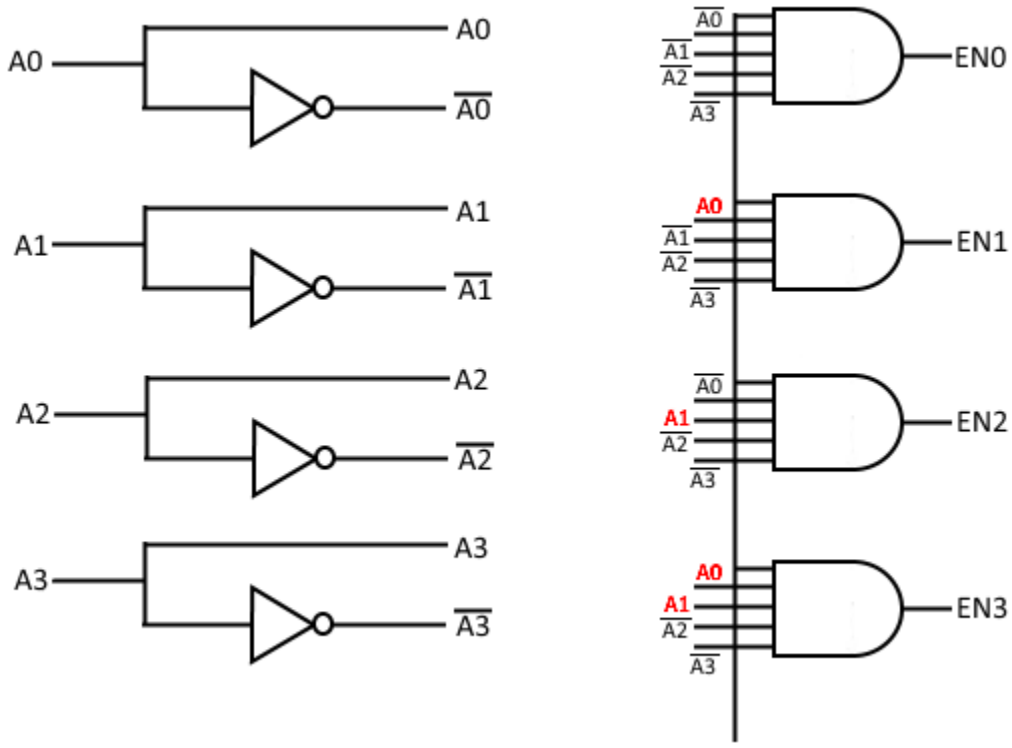
The RAM Module

A computer's Random-Access Memory is where all the data that is to be used in the execution of a program (including the program itself) is stored. Furthermore, while the program is running, new values can be saved into RAM to be used in operations later on in the program. While most computers nowadays have gigabytes of RAM, this one will only have 16 bytes in total. In modern computers, having more RAM allows your computer to run demanding programs more efficiently.

RAM works by storing bits of data in long arrays, each row being multiple bits long, which in the case of this computer they will be 8bit binary numbers. RAM access each line of memory by having input and output enable lines connected to each row instead of connecting them to each individual bit. This means that a computer can read or write data one row at a time.

For the breadboard computer, there are 16 rows of memory, each holding 8 bits, or a byte of data. Each set of 8 bits is called a word and they have an enable and write control line. This means that each word of RAM is like its own register, however, the main difference is that they're accessed differently. Every word in RAM has its own address, a 4bit number (4 bits because there are 16 bytes of RAM) through which the enable and write lines are accessed. The RAM module will have a series of inputs which tells itself to look in a particular address and retrieve information from it, these are the address lines.

To take a 4bit number and enable its corresponding memory address, an address decoder is needed. The following diagram shows the basic principle of the address decoder:

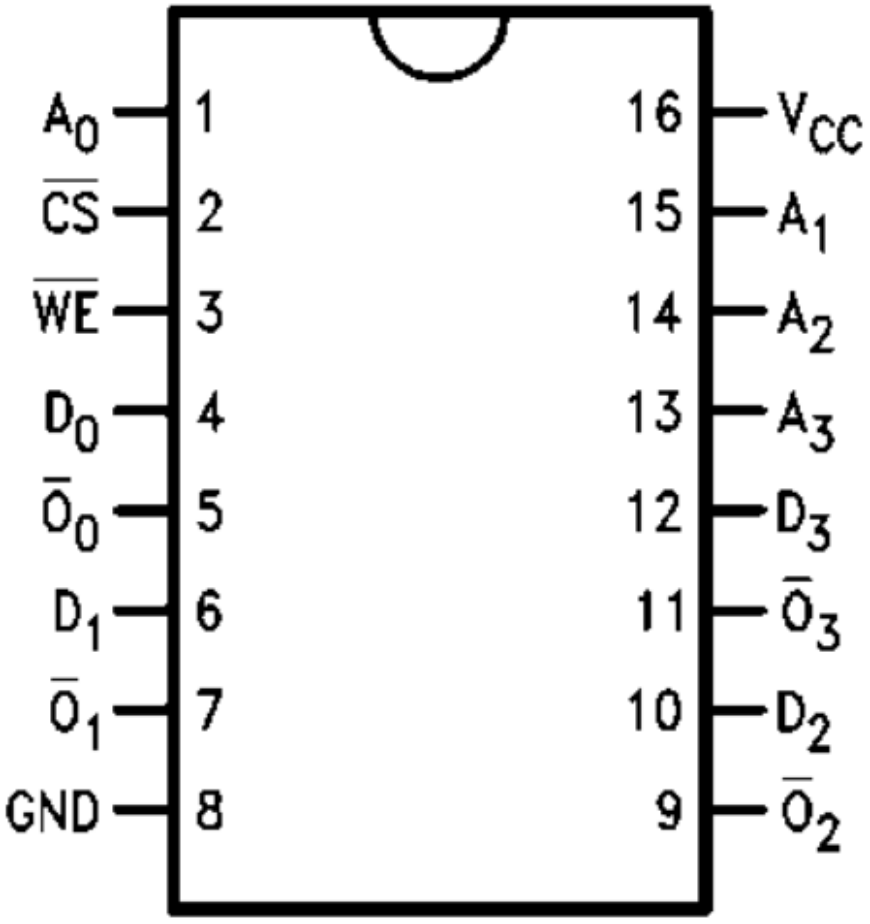


On the left, there are the 4 address lines, these are turned into two signals, the original and the inverse. On the right, there is a series of AND gates with multiple inputs. These AND gates allow the enable signal of an address to turn on if all its inputs are on. These inputs are the same as the address, just interpreted as inputs to electrical signals. For example, if we wanted to access the data at address 1100 first, the address lines would be set to 1100. This would cause A0, A1, $\overline{A2}$ and $\overline{A3}$ to go high. Only one out of all the AND gates would turn on, the one at address 1100 since those are its AND gate's parameters. On this diagram there is a fifth input into the AND gates, this would be a master enable line for all the memory addresses and would be kept high at all times. Lastly, this diagram only shows 4 addresses but of course, the full RAM module would have an address decoder for all 16 words of data.

Instead of constructing a RAM module out of individual parts, it's easier and more efficient to simply use an existing chip that contains the storage and address decoder. This way space is reduced, and mistakes are minimized, plus who wants to build 8 separate registers each connected to four and gates. So to do this, the chip that will be used is the 74LS189.

The 74LS189

This chip is holds 64bits of RAM, which you may have noticed is not enough for 16 bytes of data. This means that we'll simply have to use two of these chips, both accessed through the same address lines.



Pins 1, 13-15. The address line pins. On the diagram above A0 is the least significant bit while A3 is the most significant bit. This is where the address to access each word of memory is inputted through. This will later be connected to the Memory Address Register which will keep track of where to get the data from RAM.

Pins 4, 6, 10, 12. The data input pins. From these 4 bits of data are inputted to be stored in RAM. Data will be able to be inputted before and while the program is running. A series of DIP switches will be connected to these pins to program data beforehand. While the program is running, the inputs will come from the bus.

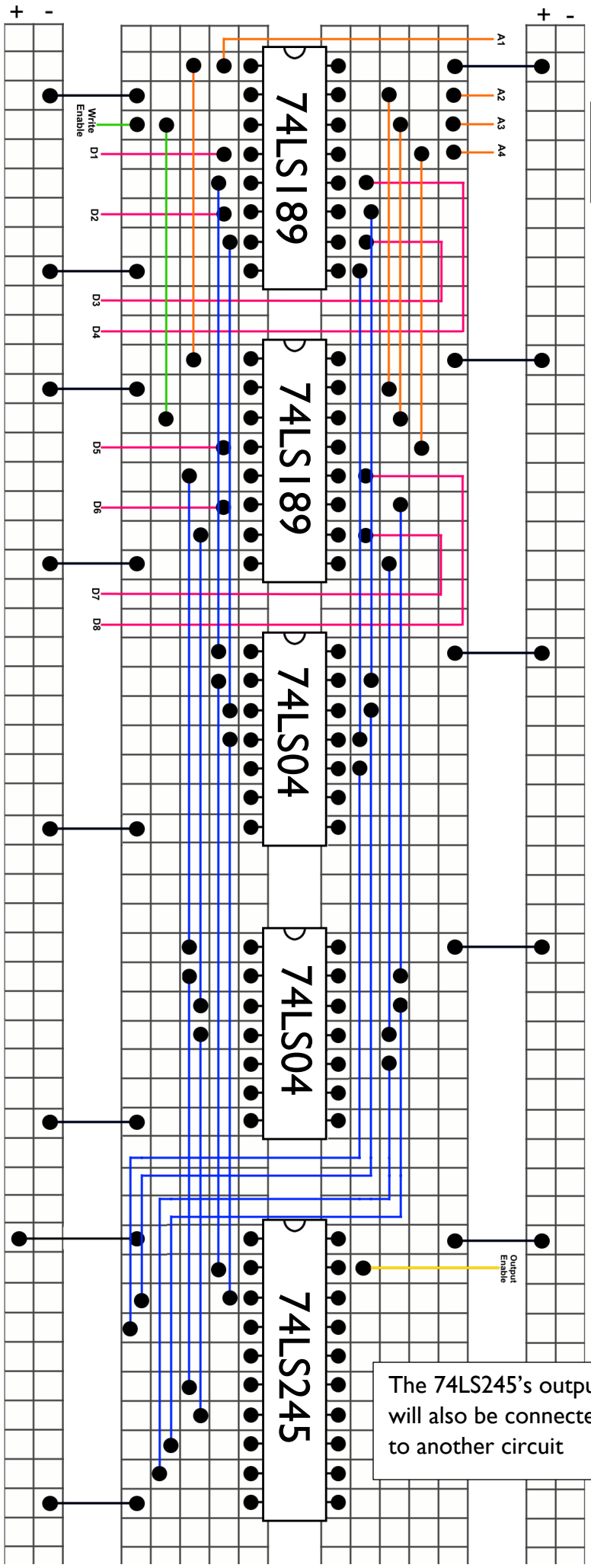
Pins 5, 7, 9, 11. The output pins. These pins from both chips used will be connected to the bus. These pins output the inverse of the data stored, as seen on the diagram with the line over the output. To invert it back to normal, NOT gates (or inverters) will be placed after these pins before the signals go on the bus.

Pin 2. The chip select. This pin acts line an output enable, which allows the data at the selected address to be outputted. Like in the previous modules, we want this to be on at all times, so we can see what's stored in memory. This pin activates when it is low, so it will be connected to ground. Since this chip is always outputting data, we need a way to control when it goes out to the bus. For this, the 74LS245 buffer is used.

Pin 3. The write enable. This pin allows data to be written into the address selected of memory. This pin will only be on when inputting data and will therefore act as a control signal for RAM's output enable. Like the previous pin, this one also activates when low.

Write Enable line will be connected to another circuit

D1-8 are the data inputs where 1 is the most significant bits. They will be connected to another circuit



A1-4 are the address lines. They will be connected to the MAR

The 74LS245's outputs will also be connected to another circuit

When using the 74LS189 chip we need to keep in mind that all the outputs are inverted within the chip. This means that we need to reinvert them back to their normal values using an inverter. To invert the output signals, we need to connect the outputs of the RAM chips to the inputs of the 74LS04 chips†; keep in mind that we are using 2 of both types of chip.

With the now proper output signals, we need to connect them to the bus. But again, like in the previous modules we need to make sure that data being output from multiple modules doesn't end up getting mixed up. For that, we use the octal bus transceiver which controls our outputs so that they're only one when we need them to be (link to appendix). Therefore, we connect the outputs from the 74LS04 chips to the inputs of 74LS245 chips. This chip will also hold our output enable line, as pin 19 of the 74LS245 acts as the enable.

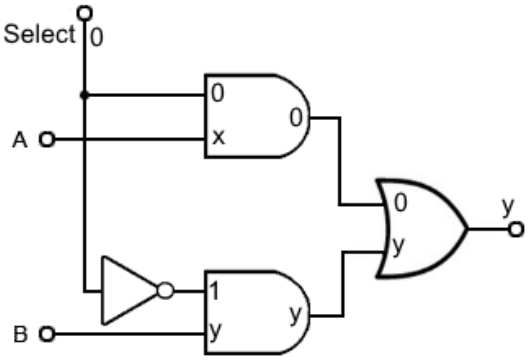
Next, we need both chips to be set to the same address when accessing data, so we tie the address line pins of the two chips together. We also want the two chips to write data at the same time, so pins 3 from both chips will be connected together and then to a separate control line that will only be on when writing the programs. Lastly, the data inputs for the RAM chips will be directly connected to the input switches.

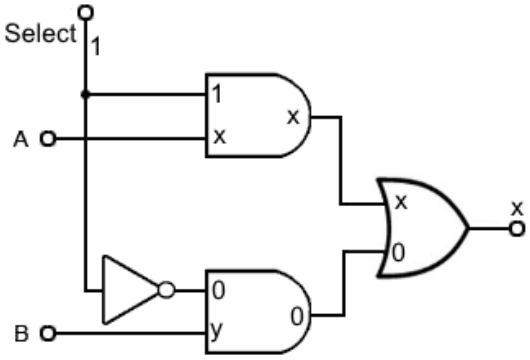
† If you want to skip this step you can use the 74LS219 chip which doesn't invert its outputs, however these chips are much more difficult to find.

The Memory Address Register

Together with the actual RAM, the computer needs a memory address register so that RAM know which address to output data. This computer accesses RAM at two separate times, before and during the execution of the program. When writing the program beforehand, we want to manually control which address the data that we're inputting is going into. However, during the execution of a program we want to read or write data from an address automatically. This means we need two ways of accessing data from the MAR: one through user inputs and the other from the bus. The user input will be done through 4bit dip switches while the address coming from the bus will be stored in a 4bit register.

To switch between two different inputs, we need a data selector. What this does is it takes two different inputs and based on a control line it will output one of the two inputs. This is done with the following series of logic gates:





A	B	S	O
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1

These logic gates output whatever input A is if the select (S) line is a 1. On the other hand, if the S is a 0 then this circuit outputs whatever input B is. The truth table on the right shows this. The way this works is that when S is a 1 then whatever is going into B's and gate will be a 0, therefore, the OR gate's output will equal A. The reverse is true when S is 0; A's AND gate outputs a 0, meaning that the final output must equal whatever B is.

The computer will utilize the 74LS157 chips which holds four of these data selectors, one for each bit in the memory address. With this chip each bit of the dip switches will be connected to their respective A input on the 74LS157, while the outputs of the 4bit register will be connected to their corresponding B inputs on the 2-to-1-line selector chip.

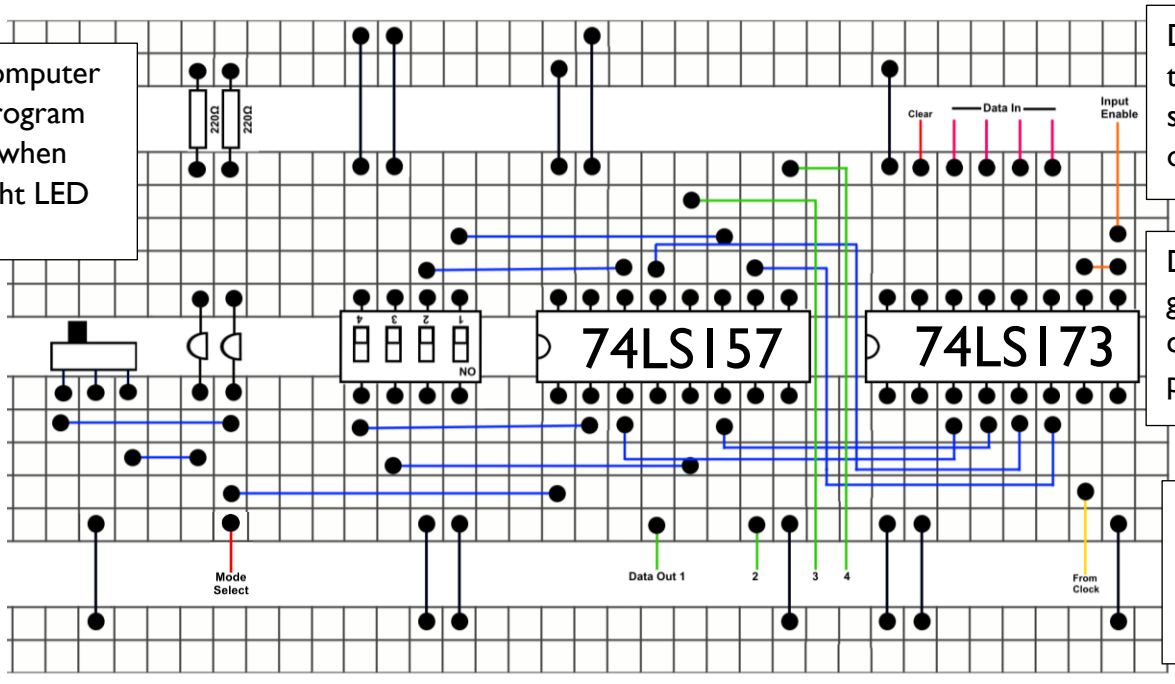
Since the MAR uses a 4 bit register we need to use the 74LS173 chip. We previously used these chips when building the A, B and instruction registers. For this module the chip will be used in much the same way: The out enable pins will always be turned on (ground), the input pins will be connected to the bus, the clock pin will be connected to the clock signal, the load enable pins to a separate control signal and the clear will be off (ground). The only difference, is that this time the outputs will be connected to the selector chip (74LS157).

To connect the dip switches to the 74LS157 chip, we connect one side of the switches to ground and the other side to the A inputs of the chip. The reason as to why we connect one side of the switches to ground is because when there is no input into the selector chip, it defaults to a 1. So therefore, if the switch is off, then the 74LS157's input will be a 1 and if the switch is on then the input will be a 0 as it's connected to ground.

Now, to actually change between the dip switches and the 4bit register we use a double throw switch where the first pin of the switch is connected to pin 1 of the 74LS157 to select between the inputs. And the second pin is connected to ground. If the switch is on (away from the switches first pin) then the selector chip will output its B inputs and vice versa.

To finish the memory address, register the only thing we have left to do is to connect the 74LS157's outputs to the RAM's address lines.

The computer is in program mode when the right LED is on



Data In lines go to the least significant bits on the bus

Data Out lines go to the RAM chips' address pins

The Mode Select line will be used in the next circuit

Back to RAM

The RAM chip will also be written with information from two locations: user input and the bus. Like for the MAR we will also need to use the 74LS157 chip two change between the two inputs. Since RAM uses 8bit words of data, we will need to use two of these chips in tandem.

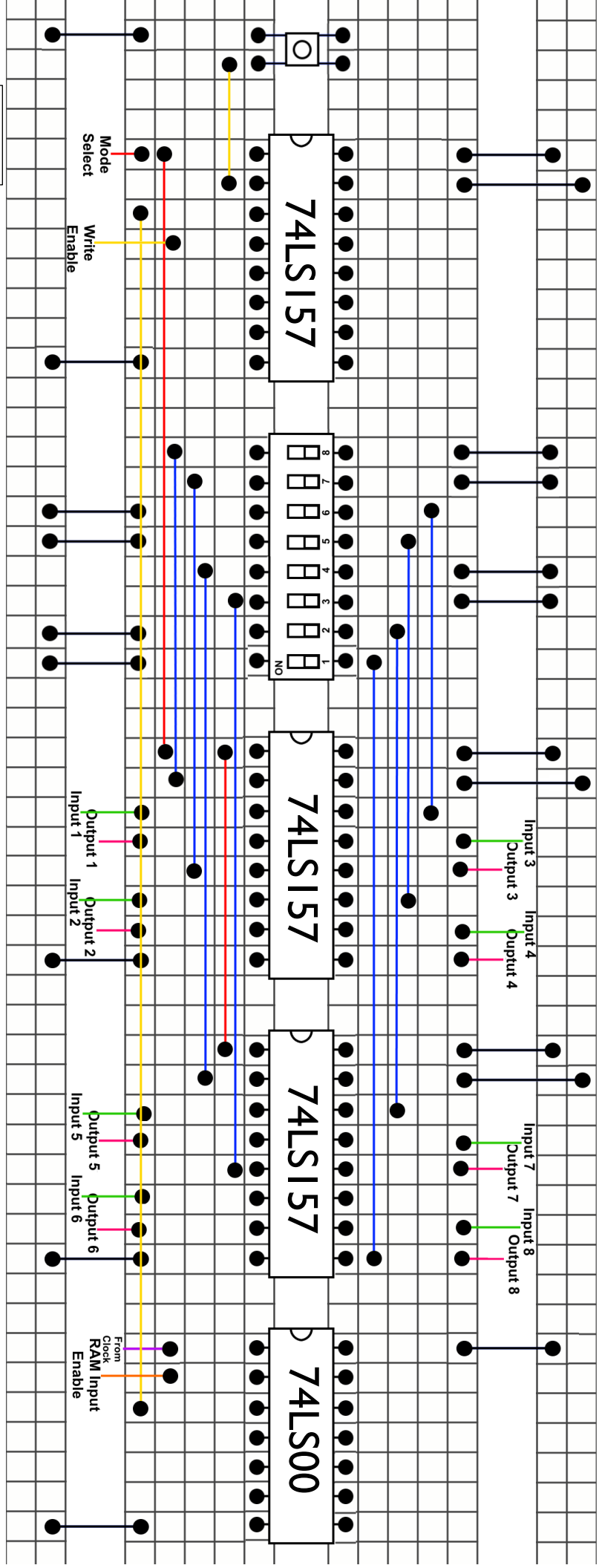
Connecting these chips is quite similar as with MAR: the output pins of the 74LS157 go to the data pins of RAM. We also use dip switches for the user input and these are connected in the same way, one side to ground the other to the A inputs. Lastly, we have the B inputs, these are connected to bus. Like in previous registers we will take the data from the bus using the same breadboard rows as the octal transceiver to reduce the space used.

Next, we need a way to toggle between the two to write into memory. Once again, the 74LS157 allows us to select when the write enable is on or off based on either the user input or the bus input. This time we only need one of the four selectors where the output goes into RAM's write enable line. First, when writing data manually what we will need to do is set the memory address using the MAR dip switches, then the actual data using the RAM dip switches and then use a touch button that writes the data in. In our 74LS157, the A input will come from this button where one side of the button is connected to ground and the other to the input.

Getting the B input for this selector is slightly more complicated. When writing data into memory from the bus we will have to use a control signal since we can't manually do it while the program is running. However, we don't want data to be written at the same time as the control signal goes high. Instead, we need data to be written on the upcoming clock pulse So what we do, is take the combination of the clock signal and RAM's write enable signal using a NAND gate. We use a NAND gate because the write enable pin on the RAM chips is inverted, this isn't an issue with the manual input because we connect the button to ground.

Inside the 74LS00 there are four NAND gates from which we'll only use one. The first input of the gate is from the control signal, the second input comes directly from the clock and the output goes into input B of the last 74LS157 chip that we've used.

Lastly, the select pins for the three 74LS157s that we've used now are also connected to the same double throw switch used in the MAR.



Mode Select line comes from the previous MAR circuit

This line goes to the Write Enable pin on the RAM chips

The Output lines go to the RAM chips' data inputs

The Input lines come from the BUS. To save space, connect them to the 74LS245's outputs from the first circuit in this module.

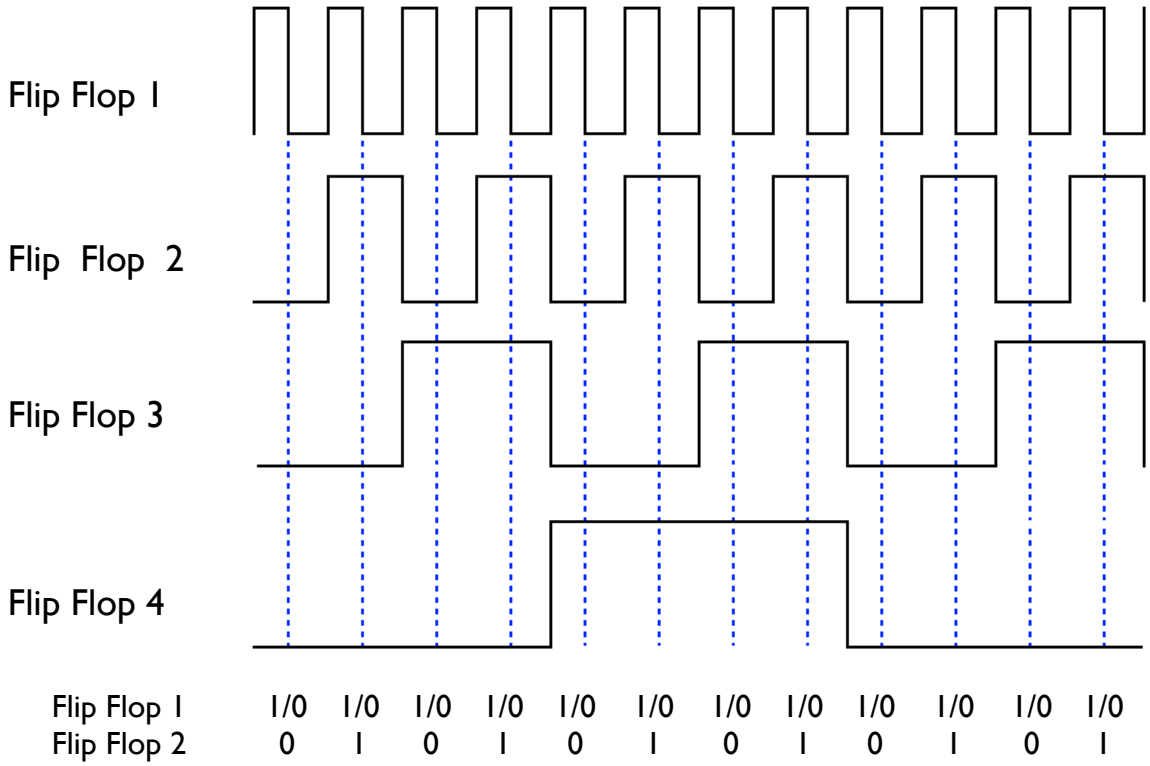
The Program Counter

As we saw with the previous section, instructions are stored in RAM within separate addresses. When running a program, we want the computer to execute the program instructions in an orderly manner; first instruction 0, then 1, then 2 and so on. This is the program counter's job, to keep track of which instruction comes next in the program. However, we don't want to always be running instructions in a consecutive order. Sometimes we may want to jump backwards once it reaches a certain instruction to create a loop. Therefore, the program counter apart from just counting in binary, should also have an 4bit input which would be transferred to the MAR to access a previous instruction from RAM.

The Binary Counter

As is evident in the name, the program counter needs to be able to count numbers one by one. This program counter will only need to count from 0-15 in binary; this range is the same as the number of addresses in RAM.

The way the binary counter works is with a series of JK flip flops hooked up to each other. JK flip-flops are little logic gate circuits with two inputs. One property that JK flip-flops have is that if both inputs are HIGH, then the output will switch on and off at every clock pulse, another way of saying this is the output being HIGH every other clock pulse. If we were to then connect this output to the clock of a second JK flip-flop while keeping both inputs HIGH so that it toggles the output, then the output of the second flip flop would change once every 2 clock pulses. If we do this two more times we can see how a 4bit binary counter has been created.



Flip Flop 3	0	0	1	1	0	0	1	1	0	0	1	1
Flip Flop 4	0	0	0	0	1	1	1	1	0	0	0	0

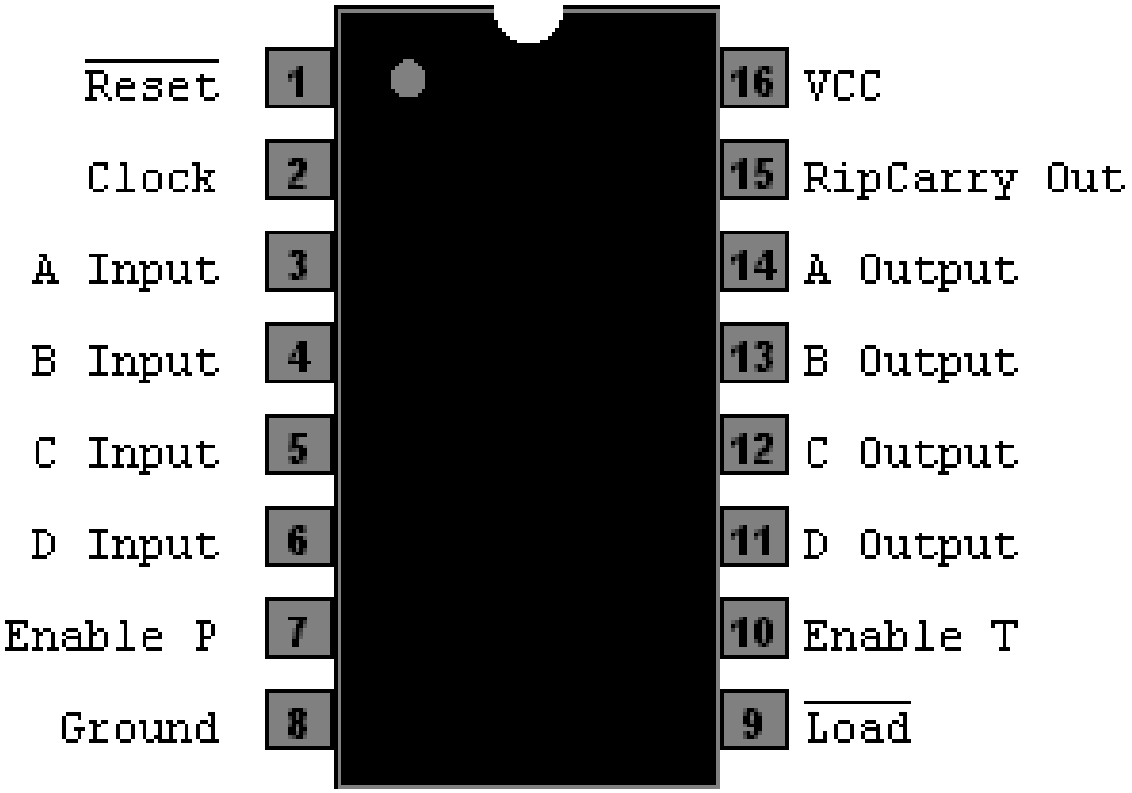
In the above diagram you can see that as the flip flops' outputs cycle, they alternate with a lesser frequency. As we go left to right in the diagram, and look at the edges of the square waves, we see that the flip-flops are counting in binary, with HIGH being a 1 and LOW a 0. In the table this is also shown. Note that flip flop #1 also changes between 1 and 0 but it isn't shown in the table for clarity.

The Program Counter's Design

To have a functional program counter that meets the computer's needs, three control signals are needed. The first control signal is the program counter's output line. This would output which instruction number that is to be executed into the 4 least significant bits of the bus. Next, to be able to input the instruction number that the program wants to jump back to, a jump line is needed. This jump line would act as an input enable which would put the 4 least significant bits from the bus to the program counter. Lastly, a count enable line is required. This line prevents the program counter from counting up at every clock tick. Each program instruction requires more than one clock tick, so we only want to increment the program once every instruction cycle.

The 74LS161

The central chip to this module is the 74LS161, which is a binary counter with some extra features that we will take advantage of to make the previously discussed control signals.



Pin 1. The reset pin. Self-explanatory, this pin resets whatever value is being stored in the chip. It will be connected to power for the most part so that it isn't activated, however, it will eventually be connected to a master reset switch.

Pin 2. The clock pin. This pin is the input of the clock to make the counter run properly. It will be connected to the clock's output.

Pins 3-6. The input pins. These pins will enable the program counter to be able to jump to a certain number. Whatever is being inputted at these pins gets stored into the program counter which will then continue counting from that value. These pins will be connected to bus's least significant bits.

Pins 7-10. The count enable pins. For the count enable control signal we will use these two pins to control when the counter should be incrementing in value. For our purposes, these two pins will be hooked up together and then two a separate control line.

Pin 9. The load enable. This pin allows the data coming in from pins 3-6 to be stored into the counter. From this pin we will connect the jump control line which will be connected to ground when we want it enabled and vice versa.

Pins 11-14. The output pins. Since the chip doesn't have any output enables and we need to control when data is coming out of the bus, the outputs will have to go through the buffer chip (74LS245).

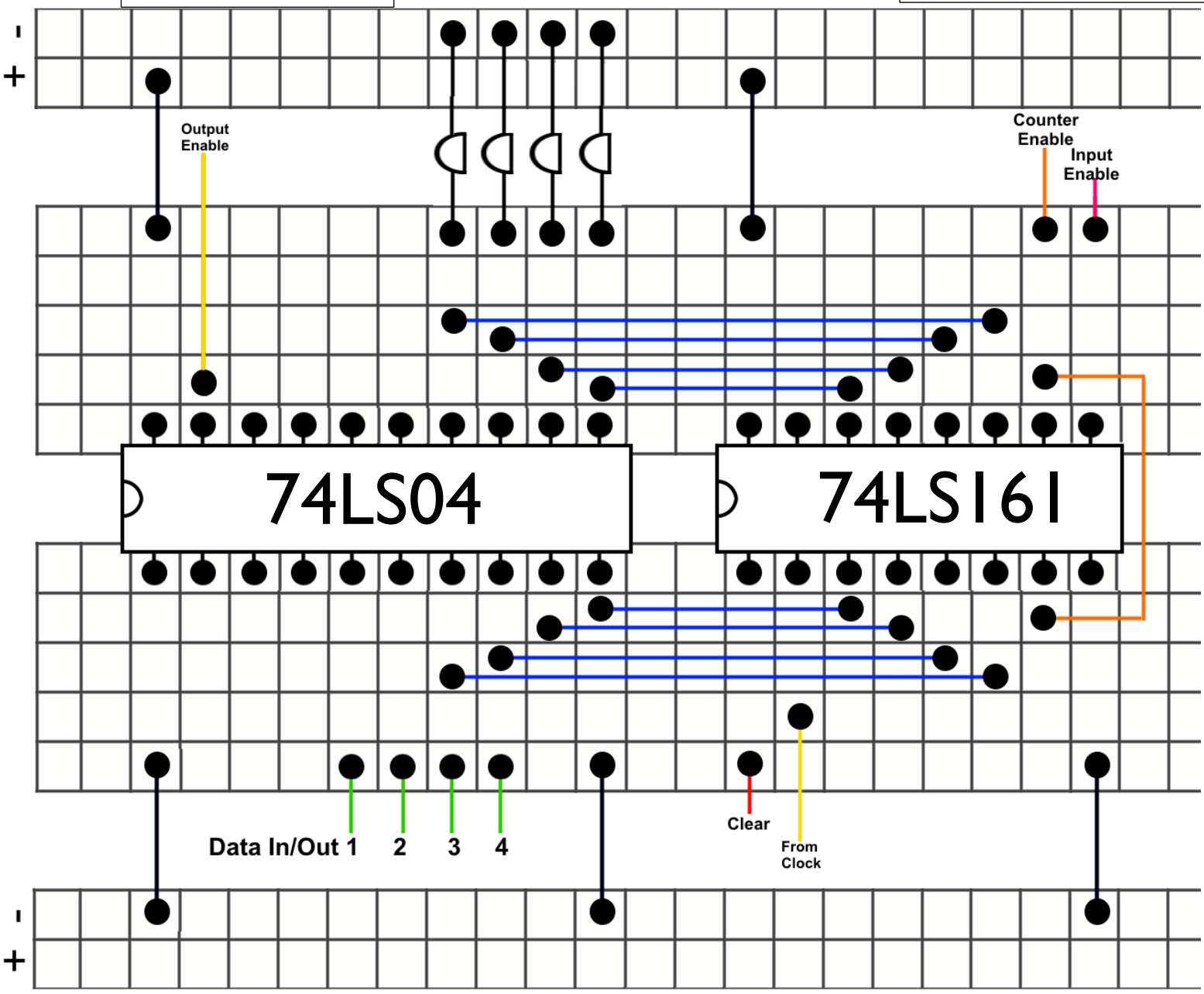
Pin 15. This pin would be used to connect multiple counters in series. Since we won't need it, it will be connected to ground.

To build the circuit connect the 74LS161's pins as described above. For the output pins the 74LS245 chip is needed. As mentioned in the beginning, refer to the chip's data sheet to see how the chips work. This 74LS245's inputs will come from the binary counter and the outputs will go to the bus. Pin 19 which is the output enable on the chip will also act as our last control line: the counter's output enable. If we were to output the counter's content to the bus then we would need pin 19 to be LOW (connected to ground); the opposite is true if we don't want the output enabled.

The Output Enable line outputs the data stored in the program counter.

The Counter Enable line simply increments the program counter.

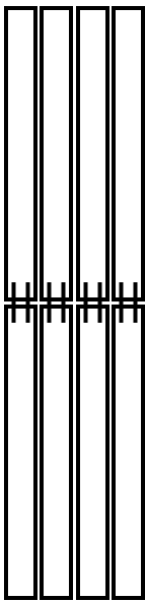
The Input Enable line acts as the jump signal, inputting whatever is on the bus to the program counter



The Data inputs and output lines will be connected to the bus's least significant bits

The Bus

When building the previous modules, most of the inputs and outputs have been connected to and from the bus. Through the bus, data from one module can be transferred to another. This computer's bus will transfer 8 bits of data at a time since that's how many bits most of our modules deal with. However, the bus should also allow for the possibility of only transferring 4 bits of data as that's what's used in the program counter and memory address register.



Building the bus is quite different to the other modules because no chips are being used. In fact, this is probably the simplest module to build but is equally important in the computer. Since this computer uses breadboards, the power rails on the edges of them work well to transfer data up and down the components. By laying four of these side by side, we end up with 8 rails which transfer the 8 bits of data. Then, to extend the length of the bus to reach all the components, we simply have two of these power rail bunches and connect them together using some leads.

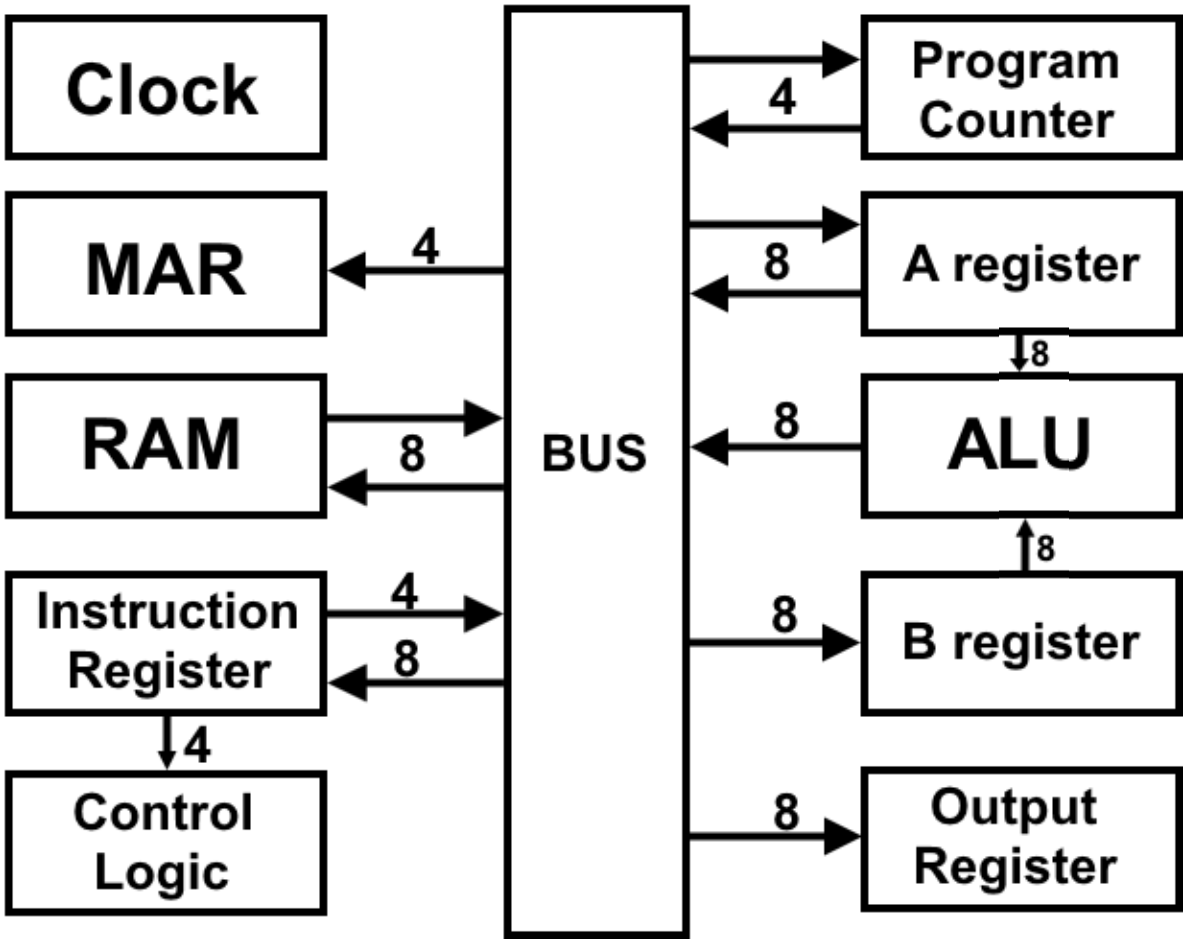
Next, all the data lines of the bus need to be connected to a pull-down resistor so that they're not left floating when only using 4 of the 8 bits. This can be done by connecting 10kΩ resistors at the top of the bus lines to the adjacent boards and then to ground. Furthermore, to visibly show what is currently on the bus, a series of LEDs can also be connected. This could be done in multiple ways:

1. Have the positive end of the LED connected to the bus and the negative end to a separate power rail on top of the bus. From here, connect them to ground.

2. Using leads, connect the data lines to some free space in a nearby board. Here, hook up some LEDs with the leads connected to the positive ends and the negative ends to ground.

3. Connect the positive ends of the LEDs to the data lines and then solder the negative ends together. This would then be connected to ground. Keep in mind that this method requires a soldering iron.

With the bus done, all that is left is to connect all the components to it. When doing this make sure not to cross any of the inputs or outputs. The program counter and MAR should be connected to the least significant bits (the right most) of the bus.


Clock
MAR
RAM
Instruction Register
Control Logic
BUS
Program Counter
A register
ALU
B register
Output Register
4
4
8
8
8
8
8
8
4
8
8
8

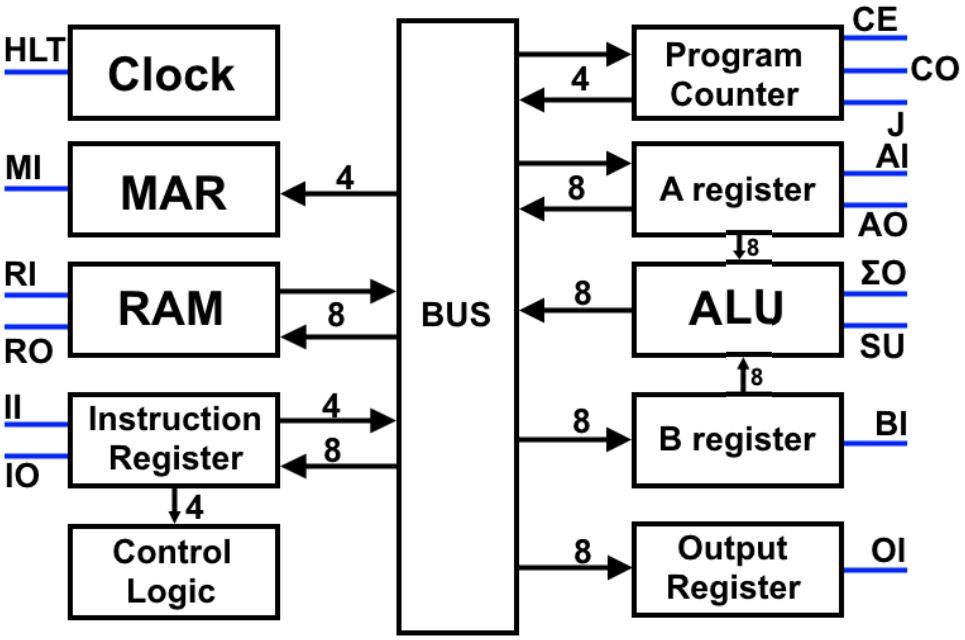
✦ ✦ ✦

The Control Logic

Lastly, there is the control logic: the brains of the computer. Here, all the modules are told what to do and when to do it so that a program is run. In a computer, the control logic is the module that enables the program inputted by the user, to run the way it's supposed to. By connecting the modules' control lines to the control logic and then manipulating them based on certain patterns of data, the computer takes series of 1s and 0s to run complex programs.

An overview of the control signals:

For the computer to be able to perform these instructions, it must do so by setting the various control signals on and off. In total, for all the modules in this computer there are 15 control signals. These are shown in the following diagram:



MI – Inputs the 4 least significant bits from bus to the MAR
RI – Inputs 8 bits from bus to RAM
RO – Outputs 8 bits from RAM to bus
II – Inputs 8 bits from bus to the instruction register
IO – Outputs the 4 least significant bits from the instruction register to bus
AI – Inputs 8 bits from bus to the A register
AO – Outputs 8 bits from the A register to bus
BI – Inputs 8 bits from bus to the A register
OI – Inputs 8 bits from bus to the output register
ΣO – Outputs 8 bits from the ALU to bus
SU – Sets the ALU to subtract the A and B registers
CE – Enables the program counter, essentially incrementing it
CO – Outputs the 4 least significant bits from the program counter to bus
J – Inputs the 4 least significant bits from bus the program counter, thus jumping to a different instruction

It is better for this computer to have all the control signals close together so that they are more easily accessed. Therefore, the computer will have a breadboard in which all the signals can be accessed through and also visualized via LEDs. However, there is a slight issue: some of the control signals are activated when they're set to LOW. It is simpler for us to deal with the control signals when they're all activated when HIGH. This means that we will have to run those control signals through an inverter so that they activate when HIGH.

When connecting the control signals to the breadboard, do so in this order so as to not mix them up later on: MI|RI|RO|II|IO|AI|AO|BI|OI|ΣO|SU|CE|CO|J|HLT.

The signals that are highlighted need to be inverted.

The Instruction Set

Before building this module or even learning how it works, we first need to figure out what are the instructions that the computer can perform and how will it do so using the other modules' control lines. The list of instructions is the following:

NOP (0000) – The "no operation" instruction. This instruction doesn't do anything it simply skips to the next instruction in the program. However, the computer still needs to go through the process of fetching the instruction and advancing the program counter.

LDA (0001) – The "load A" instruction. This instruction takes a value from memory and transfers it to the A register. This instruction also requires a 4 bit parameter which is the address from which memory should be retrieved to then transfer it to the A register. If we had an instruction in a program that said LDA 10, the number 10 wouldn't be loaded into register A, instead whichever value was stored in address 10 would be loaded to the register.

ADD (0010) – The "add" instruction. This instruction takes the value stored in the A register and adds another value stored in RAM. Therefore, this instruction also requires a parameter, the address of the value that you're adding. When using this instruction, the computer should load the value in the wanted address to register B and then add the values together. This added value is then outputted to the A register.

SUB (0011) – The "subtract" instruction. This instruction subtracts a value stored in RAM from the value stored in the A register. This value in RAM is accessed through another 4 bit parameter which points to the address of the wanted value and is then sent to the B register. The subtracted value is outputted from the ALU and into the A register.

STA (0100) – The "store A" instruction. This instruction stores a value from the A register into an address in RAM. This address is also accessed through a 4 bit parameter in the instruction.

LDI (0101) – The "load immediate" instruction. This instruction stores a 4 bit value into the A register without accessing RAM. The instruction holds a 4 bit parameter which is then directly transferred to the A register. Since this value is stored in the instruction itself it can only be 4 bits long.

ADI (0110) – The "add immediate" instruction. Similarly to the previous instruction, this instruction adds a value directly, without the need to access RAM. This works by sending a 4 bit value to the B register to then be added together. The sum of the two values is outputted to the A register.

SBI (0111) – The "subtract immediate". This instruction subtracts a 4 bit number from the A register and outputs it back to that register. Again, this 4 bit value comes from the instruction's parameter which is then loaded into the B register.

JMP (1000) – The "jump" instruction. This instruction sets the program counter to a different value which then causes the MAR to access the instruction set at the value which was set. Basically, this instruction jumps backwards or forwards to another instruction in RAM.

OUT (1110) – The "output" instruction. This instruction outputs a value. It takes the data stored in the A register and transfers it to the output register.

HLT (1111) – The "halt" instruction. This instruction stops the program to continue running. This is done by activating the halt line set into the clock module.

When programing code into the computer we would write the sequence of instructions into memory. However, we can't just write LDA 12 into memory, instead each instruction has its own binary "code name". You can see above that each instruction has its own 4 bit number, 0001 would be interpreted as the LDA instruction, turning on the necessary control signals to perform said instruction. The instructions' parameters, like the address in LDA, would be written after the "code name" to make a full 8bit binary number that can be written into memory. Using the same example, LDA 12 would be written into RAM as 0001 1100. This binary number can now be inputted with the dip switches that were set up in the RAM and MAR modules.

Microinstructions

Each instruction that the computer can perform, is done through microinstructions. These microinstructions set the control signals on and off. So through a series of microinstructions, the actual program instructions are performed.

Let's take the load A instruction. To perform this instruction, first, we output the program counter and input this value into the MAR; this tells the computer which instruction to retrieve. Then we output the value in RAM stored at that address and input it into the instruction register; this transfers the instruction to the instruction register so that it can then be evaluated by the control logic. After that, we increment the program counter by enabling it so that the next instruction can be fetched from the proper address. The next step is to output the instruction register and input the MAR. Note that the instruction register can only output 4 bits, these are the parameters that were mentioned when explaining the instructions. In this case the 4 bits set the address from which to fetch the data from RAM. Lastly, we output the value in RAM from the address that was set in the previous step, and input it into the A register.

This series of steps can be expressed like so:

CO|MI	Program Counter Out, MAR In
RO|II	RAM Out, Instruction Register In
CE	Program Counter Enable
IO|MI	Instruction Register Out, MAR In
RO|AI	RAM Out, A register In

The first three microinstructions are found in every instruction. They fetch the program from memory and advance the program counter. The last two instructions are unique to the LDA instruction. Now, we find out the unique microinstructions for the rest of the program instructions.

NOP

No unique microinstructions.

ADD

IO|MI	Instruction Register Out, MAR In
RO|BI	RAM Out, B register In
ΣO|AI	ALU Out, A register In

First, the memory address is set with the first microinstruction, then RAM is transferred to the B register, and finally the ALU's sum is outputted to the A register. There is no microinstruction telling the computer to add the numbers as the ALU does that automatically. What is needed however, is a microinstruction telling the ALU to output its value to the A register (shown in the third microinstruction)

SUB

IO|MI	Instruction Register Out, MAR In
RO|BI	RAM Out, B register In
SU|ΣO|AI	Subtract Enable, ALU Out, A register In

The process here is the same as above, the only difference is that we enable the "subtract" control line. Note that the last microinstruction enables three control lines at the same time. Since they don't interfere with each other it causes no problems.

STA

IO|MI	Instruction Register Out, MAR In
AO|RI	A register Out, RAM in

Here we see the only instance of the RAM In control line being used in this set of instructions. The first microinstruction sets the memory address to the one where data is stored. The second microinstruction transfers the data over.

LDI

IO|AI	Instruction Register Out, A register In

This instruction is very simple, the only thing required is the instructions parameter to be transferred to the A register.

ADI

IO|BI	Instruction Register Out, B register In
ΣO|AI	ALU Out, A register In

This instruction combines the ADD and LDI instructions where it loads the instructions parameter directly to the B register and then outputs the ALU's sum to the A register.

SBI

IO|BI	Instruction Register Out, B register In
SU|ΣO|AI	Subtract Enable, ALU Out, A register In,

Same as above with the exception of the subtract enable line.

JMP

IO|J	Instruction Register Out, Jump Enable

Another simple instruction where this time the computer outputs the instruction's parameter to the program counter so that the next instruction is fetched from a different memory address.

OUT

AO|OI	A Register Out, Output Register In

This instruction transfers the value in the A register to then be sent out to the output register where the user reads it.

HLT

HLT	Halt Enable

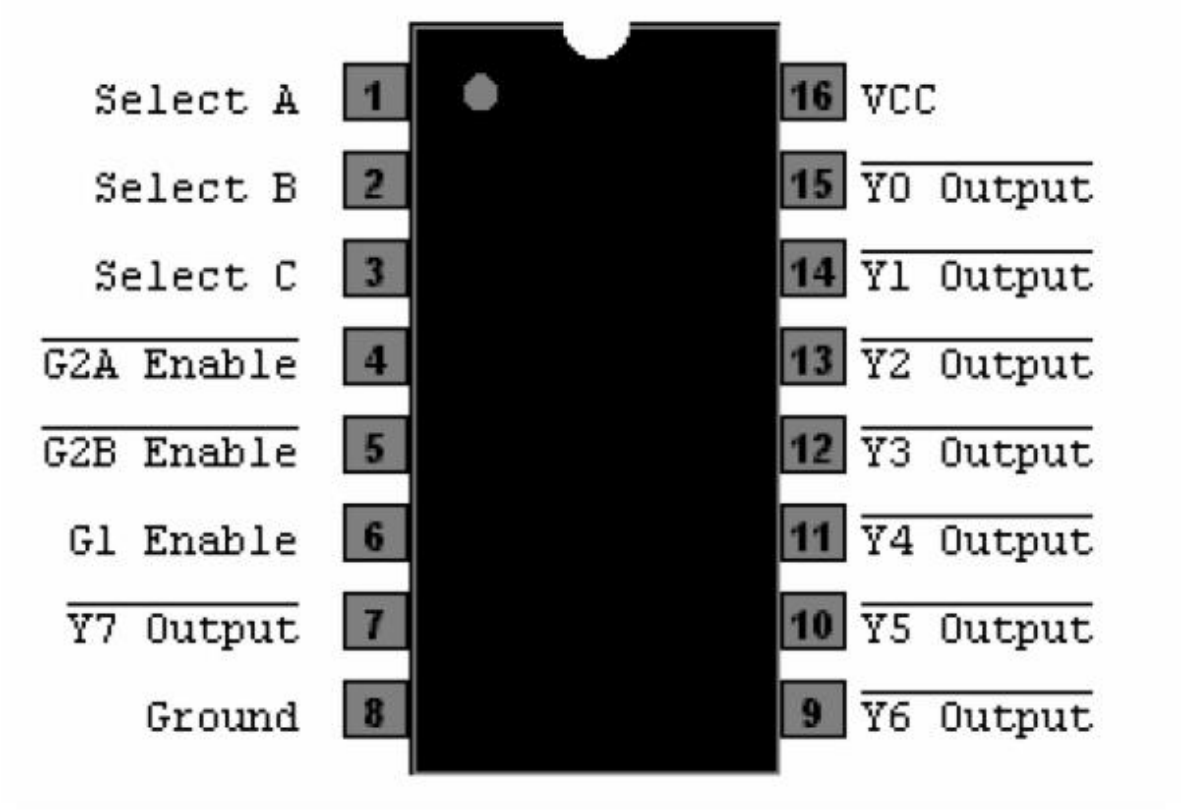
The only thing this instruction does is enable the halt line, so the clock stops and the program no longer runs.

The Microinstruction Counter

Like the program counter which keeps track of which instruction is being executed next, the control logic needs a counter that makes the microinstructions run one-by-one. This is where we use the 74LS161 chip which is the same binary counter that we used in the program counter. The 74LS161 counts from 0-15, this allows the instruction to have up to 16 microinstructions within it. However, this computer's instructions only need 6 at the most, so only the first three of the chip's outputs are used. When connecting this chip make sure to set the Load pin (we won't be loading values into the counter) and the Enable pins to HIGH.

For the programs to run smoothly the computer should set up the control lines in between clock cycles, this way the microinstruction will be set up before it's run by the computer. So it should enable the control lines and once the clock ticks the other modules should activate to run the microinstruction. This can be done by inverting the clock using the 74LS04 chip and have the output go into the counter's Clock pin.

(revise this paragraph) Instead of having the counter count in binary, it will be more useful to have it count as a series of outputs, where one turns off, then the next one turns off and so on. This can be done with a decoder chip which takes the binary numbers and turns off one of its outputs. For example: 101 activates the 5th output. By connecting this decoder to the binary counter, we get a series of outputs turning going LOW in series. The 74LS138 chip takes a 3bit binary number and turns on one of its 8 outputs.



Pins 1-3. The input pins. These pins are where the binary number is inputted. They are connected to the binary counter's first three outputs.

Pins 4-6. The enable pins. These pins enable various things on the chip. We want them to be all activated so set pins 4 and 5 to ground and pin 6 to power.

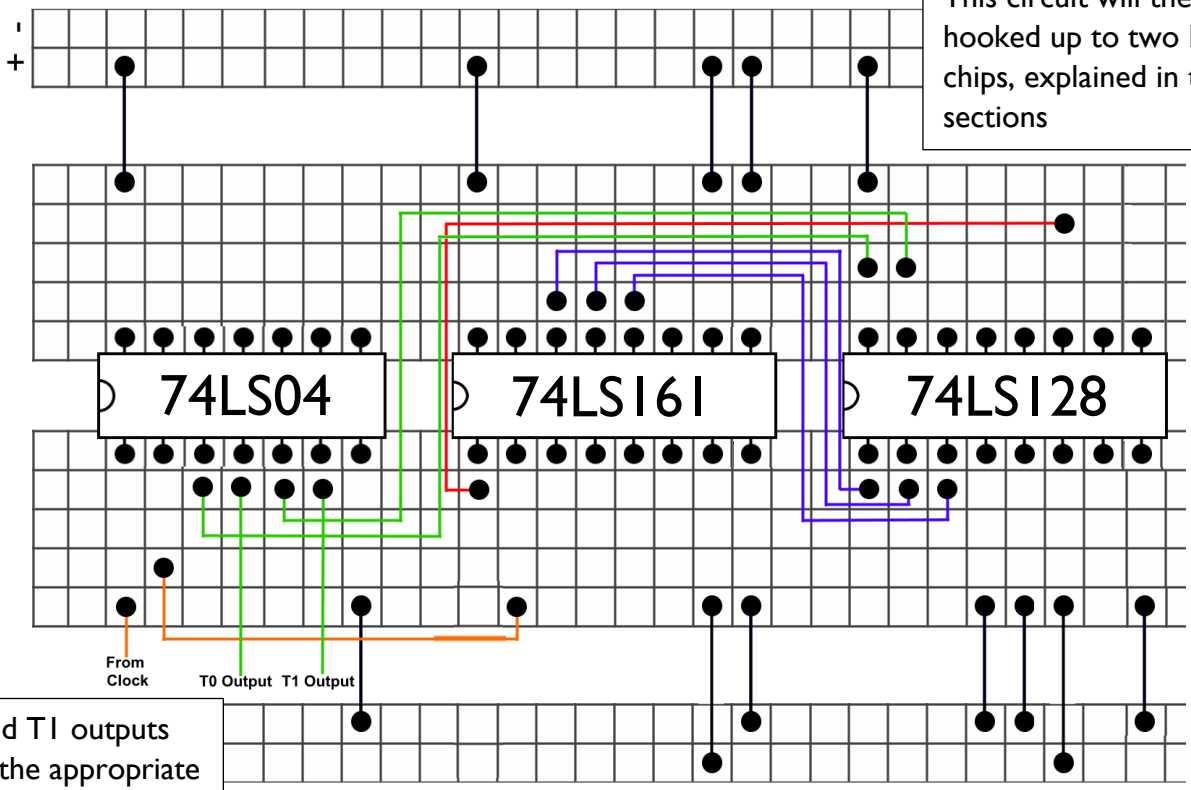
Pins 7, 9-15. The output pins. These are the chip's output after the binary number has been decoded.

With this chip in place you'd be able to see how the outputs turn off (they're inverted) one at a time from Y0 to Y1. These are 8 counts in total, but our instructions only take 6 microinstructions to complete at maximum. So to make the computer faster we can get rid of those last two steps by tying Y6 to the Clear pin on the binary counter chip. We can further optimize this process by combining the last two microinstructions of the fetch cycle like so:

Originally you had `RO|II, CE` but this can be shortened to be on the same microinstruction to become `RO|II|CE.` This saves us an extra tick allowing us to connect the counter's Clear pin to Y5 (pin 10). Now instructions that would take 6 clock ticks, would only take up 5, let's call the clock ticks T0 – T4.

The first two clock ticks of every instruction do the same thing, the T0 gets the memory address (`CO|MI`) and T1 inputs the data from RAM into the instruction register and advances the program counter (`RO|II|CE`). Therefore, we can tie T0 of the instruction cycle to always activate the CO and MI control signals. Similarly, we can also tie T1 to always activate its respective control signals.

The computer counts the instruction steps through the decoder chip. So the first step happens whenever T0 goes LOW since the output is inverted. This means that we now need to invert the T0 and T1 signals using the same 74LS04 inverter chip that we used to invert the clock signal. Then, the now inverted output can be directly hooked up to the proper control signals so that whenever T0 or T1 go off, the microinstructions for the fetch cycle are activated.



This circuit will then be hooked up to two EEPROM chips, explained in the next sections

The T0 and T1 outputs will go to the appropriate control lines

Combinational Logic Using Memory

With the microinstruction counter figured out, one issue still remains. How does the computer know that at a specific step in the instruction cycle it needs to turn on a control line based on the instruction given? For example, if we wanted to run the LDA instruction,

how would the computer process the instruction given to turn on the IO|MI control signals at the third step.

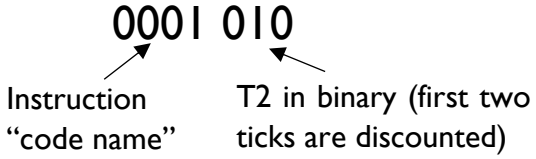
One way of doing this, would be to use really complicated circuits with logic gates, where all the possible combinations have to go through its own set of logic gates. This method however, would take up a lot of space and has a lot of room for error when wiring the circuit. Instead, this computer will achieve the same function through the use of memory.

In memory the computer would have stored all the outputs for every instruction, and you would access this data by setting the memory addresses to a combination of the instruction's "code name" and the step that the computer is running. Here's how this would work:

Say our instruction was to load address 12 from RAM to the A register → LDA 12. This is programmed into RAM as 0001 1100. The control logic only cares about the first four bits of the code as they tell the computer which instruction is needed.

The first microinstructions of the fetch cycle are automatically done above as they're already connected to the proper control lines, meaning that only the last three ticks are left. So in memory we would need three addresses for each instruction, which would then activate the proper control lines.

These addresses would be inputted as a combination of the instruction's "code name" and whether the computer is running T2, T3 or T4. The address for LDA at T2 would be:



At this address, a 16bit value would be stored. Each bit of this value, except the last one, represents one of the 15 control signals, therefore at the above address, a 1 would be stored for the IO and MI bits. The order in which the control signals will be represented as the bits, will be the same as the one laid out on the breadboard where all the control lines are connected: MI|RI|RO|II|IO|AI|AO|BI|OI|ΣO|SU|CE|CO|J|HLT.

Thus, at T2 for instruction LDA, 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 would be stored at address 0001 010. The following table is used to better show the addresses and their corresponding data since that long sequence of 1s and 0s doesn't really mean anything to our human eyes.

	Instruction				T2	T3	T4	MI	RI	RO	II	IO	AI	AO	BI	OI	ΣO	SU	CE	CO	J	HLT	X			
	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
NOP	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	0	0	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	MI	IO	
LDA	0	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	RO	AI	
	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	0	1	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	MI	IO	
ADD	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	RO	BI	
	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	ΣO	AI	
	0	0	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	MI	IO	
SUB	0	0	1	1	0	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	RO	BI	
	0	0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	ΣO	AI	SU
	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	MI	IO	
STA	0	1	0	0	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	RI	AO	
	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	IO	AI	
LDI	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	1	1	1	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	IO	BI	
ADI	0	1	1	1	0	1	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	ΣO	AI	
	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	IO	BI	
SBI	1	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	ΣO	AI	SU
	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	1	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	IO	J	
JMP	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	AO	OI	
OUT	1	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	HLT		
HLT	1	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			

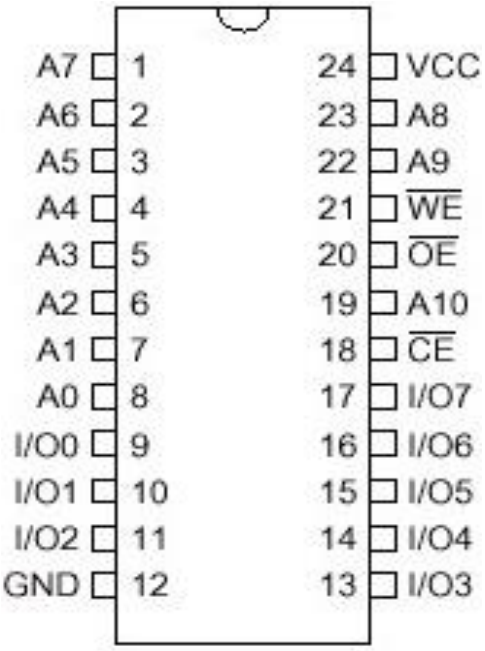
Memory Address **Memory Contents**

EEPROM

The type of memory that the computer will use to store all this data is called EEPROM, which stands for "Electrically Erasable Programable Read-Only Memory". EEPROM is read-only meaning that you can't write data into it while the program is running. ROM memory is the kind that you would find in video game cartridges or discs where the computer can only read data from it but can't change anything. The "Electrically Erasable Programable" part means that we can input data into it and then change it at a different time.

A 28C16 EEPROM chip will be used by this computer. This chip has more than enough memory to accommodate the 7bit addresses that are needed, but the data itself is stored in words that are only 8bits long whereas the computer needs 16-bit words to accommodate all the control signals. This means that we'll have to do the same thing as with the RAM module, use two chips that are accessed with the same address.

The 25C16 EEPROM



Pins 1-8, 22, 23, 19. The address pins. Through these pins the memory addresses will be inputted. Only pins 2-8 will be used as we only need 7 bits for the addresses.

Pins 9-11, 13-17. The input/output pins. These pins are used to program data into the chip and then retrieve that data once the computer is running. Whether the chip is inputting, or outputting is based on the output and write enable pins.

Pin 18. The chip enable pin. This pin allows itself to be used when LOW, so it will always be connected to ground.

Pin 20. The output enable pin. This pin allows data to be outputted through the I/O pins. To output data the pin should be LOW and the input data it should be HIGH. When hooked up to the computer it should stay connected to ground.

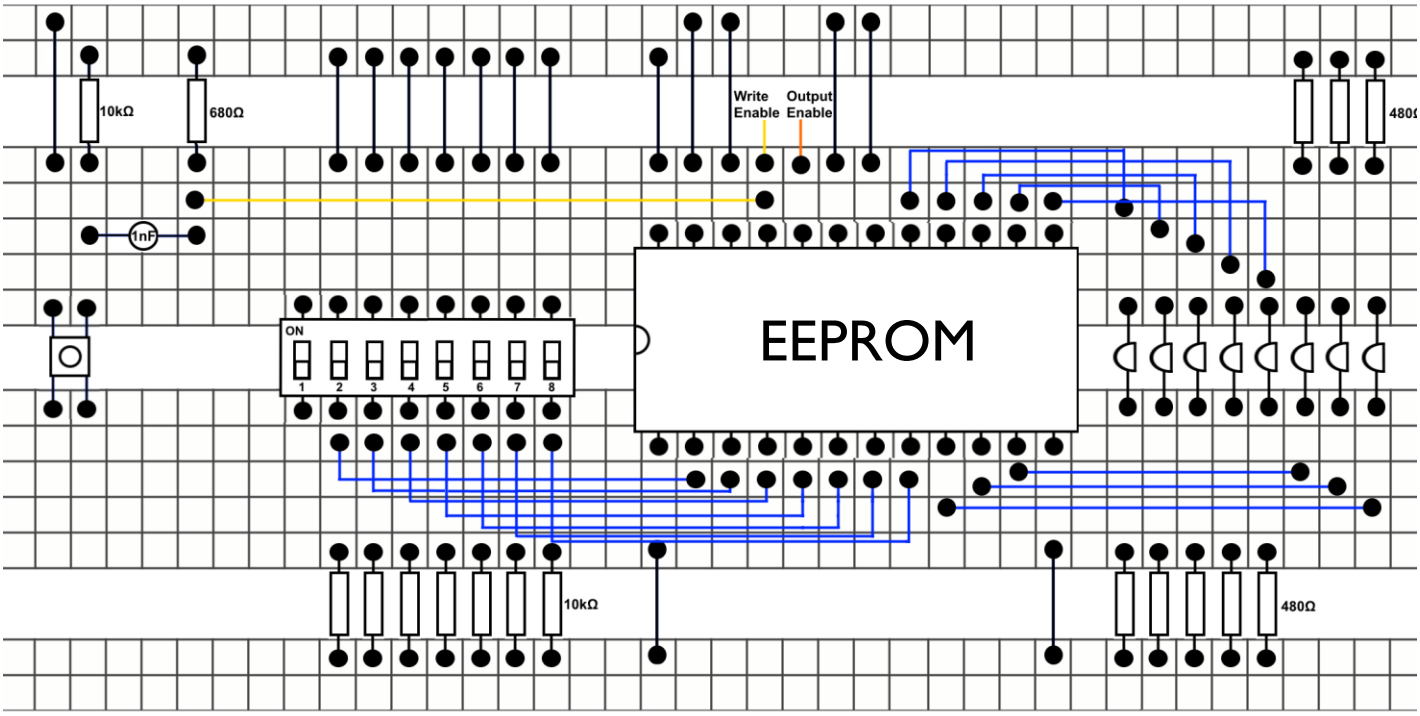
Pin 21. The write enable pin. When this pin is brought HIGH, the whatever is coming through the I/O pins is saved into the address selected. Once in the computer it will stay connected to power.

Programing the 25C16 EEPROM

Before connecting the chip to the computer itself, the data needs to be programed in. This will be done using dip switches to control the addresses and then jumper leads to input the data easily. When programing the chip, it helps to connect the I/O pins to LEDs so that you can see the data that you're inputting, but this step isn't required. Between the LEDs and the I/O pins hook up some jumper cables, these will be brought HIGH or LOW based on what we want to input.

Then, the address pins would be connected to the dip switches, where the ON side of the switches is connected to 5V and the OFF side to ground through a 1kΩ resistor(this way the data defaults to a 0). Since the computer only needs 7bit addresses, pins A7-10 can be connected to ground. Next, the chip enable pin should be connected to ground and the output enable to power. The output enable pin can be brought LOW to see the data stored in the chip but not while inputting data. Lastly, the write enable pin should be connected to a push button that would be pressed to store the data being inputted.

However, the chip's data sheet specifies that to input data, the write enable pin should only be LOW for between 100-1000 nanoseconds. Considering that 1000 nanoseconds is very little time to push a button, we can connect a 1nF capacitor and a 470Ω resistor in series in front of the push button with the lead connected to the write enable pin in between (this is shown more clearly in the diagram below). A second resistor is also needed to discharge the capacitor on its other side. This little circuit stops electricity from flowing through the capacitor for 470 nanoseconds.



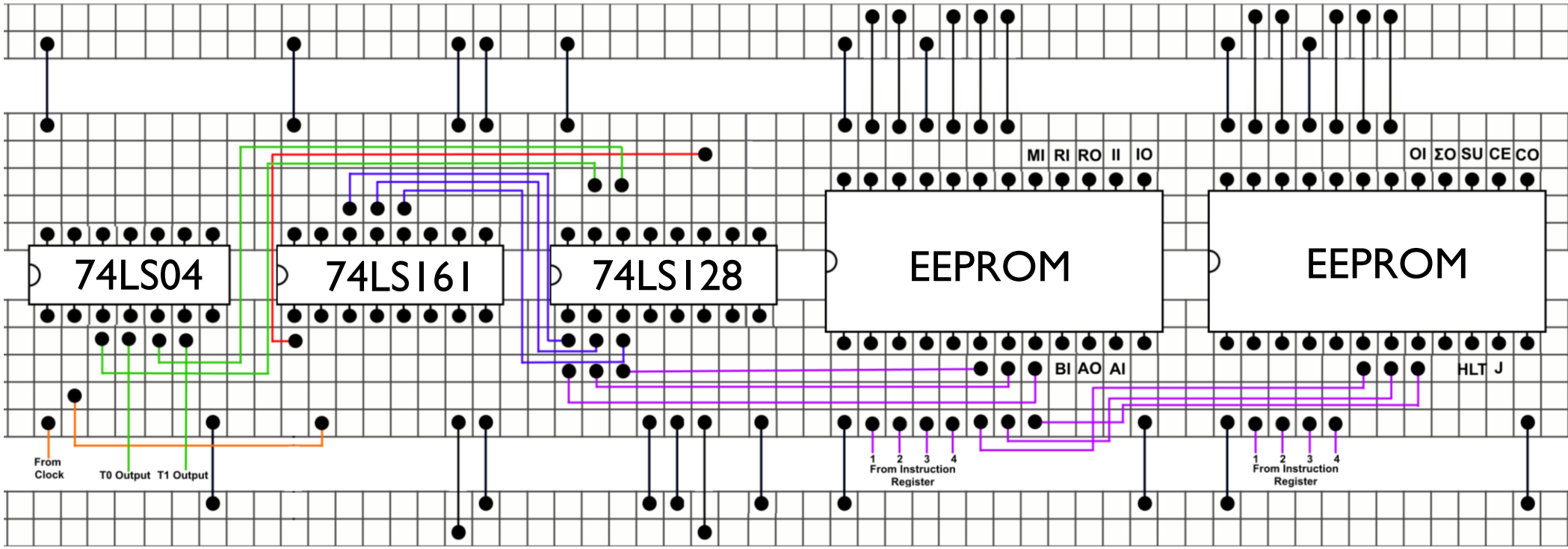
With this set up you can program the EEPROM by first selecting the address with the dip switches, then manually connecting the input leads to either ground or power based on the data required, and then pressing the push button. When connecting the I/O pins they should follow the same order that was laid out:

MI	RI	RO	II	IO	AI	AO	BI	OI	ΣO	SU	CE	CO	J	HLT	Gnd.
I/07	I/06	I/05	I/04	I/03	I/02	I/01	I/00	I/07	I/06	I/05	I/04	I/03	I/02	I/01	I/00
Chip 1								Chip 2							

Make sure to switch out the EEPROM chips to program the full 16 bits of data.

Connecting the EEPROMs

With both 25C16 chips programed, they can now be connected together with the rest of the computer. The A0-2 address pins are to be connected to the binary counter's outputs, the A3-6 address pins should be connected to the instruction register, and the other address pins should be connected to ground since they won't be changing. The I/O pins connect to the control lines following the order laid out above. Finally, the write enable pin goes to 5V, and the output and chip enables go to ground.


74LS04
74LS161
74LS128
EEPROM
EEPROM
MI RI RO II IO
BI AO AI
OI ΣO SU CE CO
HLT J
From Clock
T0 Output
T1 Output
1 2 3 4
From Instruction Register
1 2 3 4
From Instruction Register

Programing the Computer

Instructions are inputted into the computer through a series of dip switches connected to RAM. What the user would have to do is first set the computer to Write mode with the two-state switch in the RAM module, input the proper memory address and then input the instruction with the binary code. A simple program that could be inputted is the addition of two numbers.

Say we wanted to add 67 and 29:

Instruction	Memory Address	Data in RAM
LDA 4	0000	0001 0011
ADD 5	0001	0010 0100
OUT	0010	1110 0000
HLT	0011	1111 0000
Data	0100	01000011 (67)
Data	0110	00011101 (29)

The inputs for the memory address and the data are shown in the table above. We would have to manually set the dip switches ON or OFF for each instruction. Once the program is written, the computer would be set to Run mode and the clock would be started. The output would show 01100000 or 96 in decimal.

Now, let's try a more complicated program like outputting the Fibonacci sequence. This sequence is an infinite list of numbers where the next number is equal to the sum of the two previous ones: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34… A computer can create the Fibonacci sequence by having three variables, the first two adding to the next and then the values cycling between the variables like so:

x → y	y → z	z → x + y
0	1	1
1	1	2
1	2	3
2	3	5
3	5	8
5	8	13
8	13	21

The table shows how to output the Fibonnacci sequence, first x is set to 0 and y is set to 1. Then, z is set to the sum of x + y, x is set to y's value and y is set to z's value; this process is then looped. In our program we want the computer to output x's changing values as x is the only variable with the full sequence starting from 0.

Next, this process needs to be written with our computer's instructions and then converted to the binary code to be inputted into RAM.

	Instruction		Memory Address	Data in RAM
1.	LDA 10	Load x	0001	0001 1010
2.	OUT	Output x	0010	1110 0000
3.	ADD 11	Add y to x	0011	0010 1011
4.	STA 12	Store the sum to z	0100	0100 1100
5.	LDA 11	Load y	0101	0001 1011
6.	STA 10	Set x to y's value	0110	0100 1010
7.	LDA 12	Load z	0111	0001 1100
8.	STA 11	Set y to z's value	1000	0100 1011
9.	JMP 1	Jump to the beginning	1001	1000 0001
10.	Data for 'x'	Begins by setting x to 0	1010	0000 0000
11.	Data for 'y'	Begins by setting y to 1	1011	0000 0001
12.	Data for 'z'		1100	0000 0000

If we were to input this program into the computer, we would see the output show the Fibonacci numbers until reaching a number greater than 255 which is the most the output can show. This computer doesn't have a way of detecting whether the number is greater than 255 so the computer would keep going unless the clock is stopped.

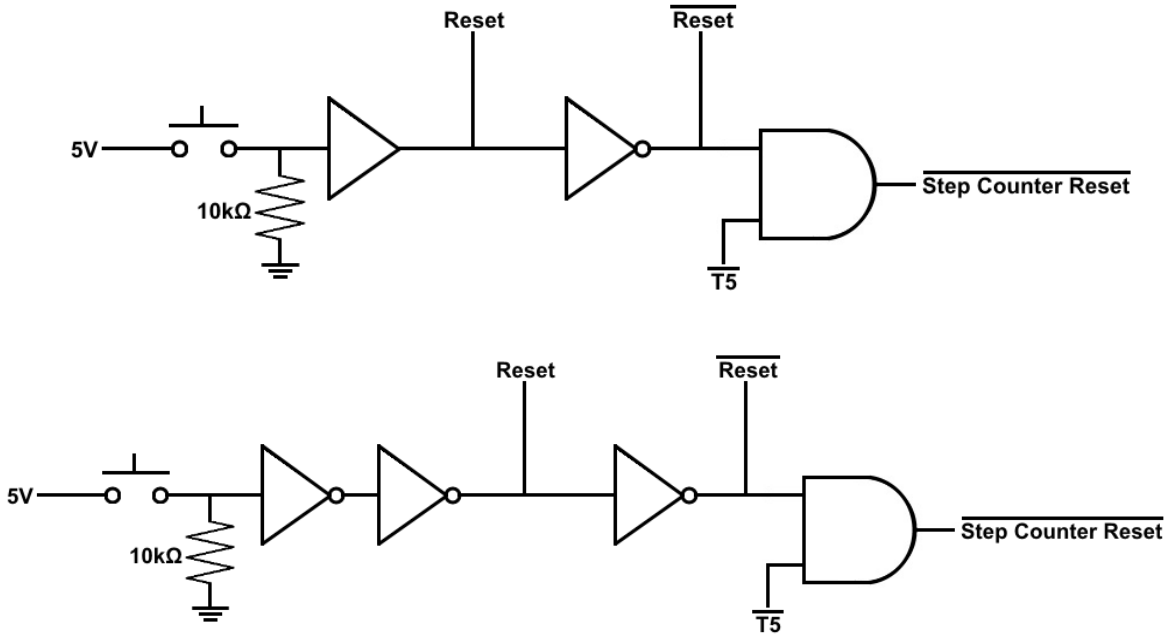
✦ ✦ ✦

The Finishing Touches

Reset Switch

At this point the computer is basically finished, you can program code in through RAM and run it, giving you the expected answer. However, to run that program again or run a different one, you would need to reset all of the components one at a time. This is really inconvenient so it's easier to have a master reset switch that clears all the modules. The way this will work is by having a push button that defaults to LOW so that when it's pressed a reset signal will go HIGH clearing all the registers.

While the reset switch may sound simple there are three different reset lines that the computer has. The first is a reset that clears the components when turned HIGH, the second acts the same way but clears other components when turned LOW (the program counter for example resets when brought LOW). The last reset line only clears the control logic's step counter since the binary counter is already connected to the T5 output. The following circuit shows how the reset line would be created:



The pull-down resistor defaults the basic reset line to LOW so that it isn't active all the time. The buffer (triangle without the circle) simply allows the button to turn on the Reset line which will be connected to the computer's components. The NOT gate inverts the signal so that it goes LOW when the button is pressed. Lastly, both the $\overline{\text{Reset}}$ and the $\overline{\text{T5}}$ lines default to HIGH, thus allowing current to pass through the AND gate. Only when either of the two signals go LOW does the AND gate output LOW, causing the Step Counter Reset to activate.

To make this circuit use up less chips, we can change the buffer to have two inverters in series. The three total inverters can be found in the 74LS04 chip used in the control logic

and the 74LS08 chip (AND gates) used for the clock can be taken advantage of to complete the circuit.

Make sure that:
$\overline{\text{Reset}}$ goes to MAR, A register, B register, Instruction Register, Output Register
$\overline{\text{Reset}}$ goes to Program counter.
$\overline{\text{Step Counter Reset}}$ goes to the reset pin (pin 1) on the control logic's binary counter.

Power Supply
For a power supply you can use anyone that outputs 5V. Most phone chargers output 5V but to use it you would have to strip the wire and solder the connecting so that they fit the breadboard. Dedicated breadboard power supplies also exist or even the power output of a micro controller can be used.

✦ ✦ ✦

Further Additions to the Computer

The computer that this guide shows how build can be used to do basic computations. To further improve this computer to make it more user friendly or improve its usefulness, there are various additions that can be made to the computer.

1. Instead of having a binary output, the computer's output register can be connected to a decimal segment display (the ones you would find in a cheap calculator). This addition would require combinational logic like the one used for the computer's control logic using the EEPROMs.
2. Another possible expansion is to have more instructions for the computer, which would allow for more complicated programs. One of these possible expansions is the conditional jump. This instruction would jump to another instruction if an event happened; for example, if the ALU returns a negative number, a 0 or a number greater than 255. The conditional jump is extremely useful and can be implemented quite easily.
3. By increasing the amount of RAM, longer programs could be stored in memory. 16 bytes of RAM is enough for basic calculations but not enough for complex programs. Along those lines, instead of the components only using 8bits of data, you could increase that number.
4. A more practical expansion is to remove the program counter enable control line so that it doesn't become part of the microinstruction. Instead of every instruction enabling the program counter, it could simply be incremented automatically at the end of every instruction.
5. If you found programing the EEPROM manually very tedious, then you could find a way of doing it using a microcontroller. There are even pre-built devices that are specifically made to program EEPROMs.
6. Apart from the previous example, there are many other ways of implementing modern microcontrollers like the Raspberry Pi or an Arduino. These devices could be used to input programs using a keyboard instead of the DIP switches. They could also be used to hook up the output to a monitor while converting the binary output to a more user friendly decimal based output.

✦ ✦ ✦

Materials

Item	Quantity	Modules
Breadboard	14-15	All
Wire	-	All
2-State Switch	2	Clock, RAM
Momentary Push Button	2	Clock RAM
4-position DIP switch	1	RAM, Control Logic
8-position DIP switch	1	RAM, Control Logic (can be switched out to program the EEPROM)
Various Color LEDs	-	All
220Ω Resistor	2	RAM
470Ω Resistor	15	Bus, Control Logic
1kΩ Resistor	3	Clock, RAM
10k Resistor	15	Bus, Control Logic
100kΩ Resistor	2	Clock
1MΩ Resistor	1	Clock
0-1MΩ Potentiometer	1	Clock
1nF Capacotor	1	Control Logic
10nF Capacitor	1	Clock, RAM
100nF capacitor	2	Clock
1µF Capacitor	2	Clock
555 Timer	1	Clock
74LS00 – NAND gates	2	RAM
74LS04 – Inverters	4	Clock, RAM, Control Logic
74LS08 – AND gates	3	Clock, Output
74LS86 – XOR gates	2	ALU
74LS138 – Binary Decoder	1	Control Logic
74LS157 – 2-to-1 line select	4	RAM
74LS161 – 4bit binary counter	4	Program Counter, Control Logic
74LS173 – 4bit register	7	RAM, Registers
74LS189 – 64bit RAM	2	RAM
74LS245 – Bus Transceiver	6	Registers, ALU, RAM, Program Counter
74LS283 – 4bit adder	2	ALU
28C16 EEPROM	2	Control Logic

This list of materials is not completely accurate as I made it before making the final design of the computer. Therefore, there may be some extra pieces or some that are missing.

Conclusion

By reading through this guide and building the 8bit breadboard computer, you will hopefully have learned the basics of computer science. Instead of computers being shrouded in mystery, you will now realize that they're machines which contain many smaller components that work elegantly to provide the user with their desire.

While it's true that most, if not all modern computers don't resemble the one made in this project, they do share many similarities. Think of this breadboard computer as being the bare-bones skeleton that is used to teach how the body holds itself up. Sure it can't perform most of the functions that we need today, but it's still a good teaching tool that shows you how changing a couple of switches can give you the answers to mathematical problems.

Through this computer you will have learnt the basics of computer architecture, what different components are needed to perform a task and how they're used to run a program. Many of the modules built in this project can be found in your everyday laptop, they're just bigger and more complicated.

You will also have learnt how computers execute programs. Even if you learn how to code nowadays, you probably wouldn't know that every instruction is made up of even smaller instructions, which manipulate various control lines, moving data around multiple components. The "programming language" used in this computer is called machine language and is the foundation for all modern programming languages. Languages like C have instructions that are then translated into machine language which the computer can understand.

There are many resources online that you can find that explore these computer science concepts at a deeper level. Ultimately, by reading this guide I hope that you've gained knowledge in this subject and have become interested to continue learning about it.