How to Build an 8-Bit Computer

by spel3o on May 1, 2012

Table of Contents

Author:spel3o 8 Bit Spaghetti
All of my life I have been interested in learning the way things work. It was always hard for me to use something and just accept that it works without taking it apart and seeing what makes it tick. Due to this electronics has become a very big part of my life and a very enjoyable part as well. Recently I gained a lot of curiosity about the operation of the computer that we know and love today.

Intro: How to Build an 8-Bit Computer

Building an 8-bit TTL computer sounds like a daunting and complicated task, or at least it did to me when i started out on my journey to understand the architecture of a basic CPU. When it comes down to it, a CPU is fairly simple in operation once you learn the fundamentals behind all of its processes. This project is intended to help anyone interested in building their own computer and gaining the wonderful knowledge that comes along with the process. Don't be afraid to try, you can only learn.

This project will start off by describing the basics of electronics. After that, the fundamentals of binary and boolean logic will be described. Lastly we will then move onto the function of the various parts of a simple-as-possible computer (with a few modifications) as described in Malvino's text Digital Computer Electronics. This means that the end product of this Instructable will be a computer that you can program with a unique instruction set. This project also leaves many of the design aspects of the computer up to you and serves as a guide for building your own computer. This is because there are many ways to approach this project. If you already have a sound understanding of boolean logic and the workings of binary feel free to skip to the meat of the project. I hope that you all enjoy and get something out of a build like this, I know that I sure did.

For this project you will need:

1.) A power supply
2.) Breadboards + lots of wires
3.) LED's for output
4.) Various logic IC's (discussed later)
5.) Free time
6.) A willingness to mess up and learn from mistakes
7.) A lot of patience

Optional (but very useful):

1.) Oscilloscope
2.) Digital multimeter
3.) EEPROM programmer
4.) Sonic screwdriver

Useful Links for a Project Like This:

Digital Computer Electronics: http://www.amazon.com/Digital-computer-electronics-Albert-Malvino/dp/007039861
TTL Cookbook: http://www.amazon.com/TTL-Cookbook-Understanding-Transistor-Transistor-Integrated/dp/B0049UUV38

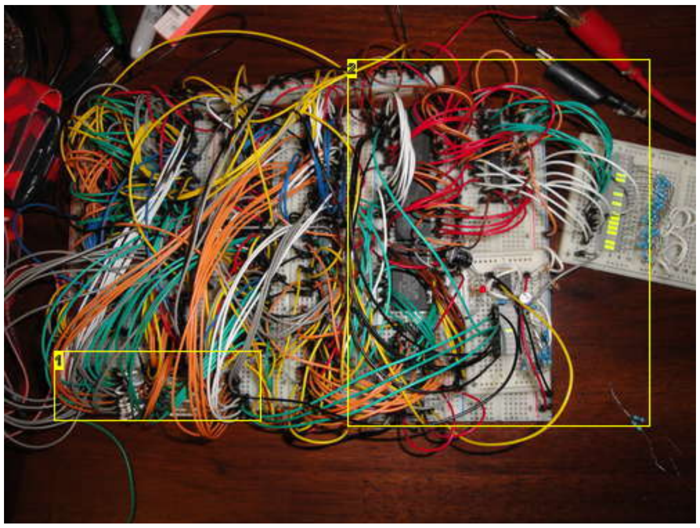


Image Notes
1. All of these wires connect to a common bus on my computer.
2. This is the control matrix for my computer.

Step 1: What Is a Computer?

This may seem like a very simplistic question that does not need answering when, in fact, it is a question that many people do not know the true answer to. Computers have existed a lot longer than the transistor in mechanical and theoretical form. The actual definition of a computer was thought up by a very intelligent individual by the name of Alan Turing. He described a machine that was termed the Turing Machine. Every computer that we use today, from the computer or cell phone that you are reading this on to supercomputers all can be classified as a Turing Machine at their most simplistic level.

What is a Turing Machine? A Turing Machine consists of 4 parts: the tape, head, table and state register. To visualize the operation of such a machine you first have to imagine a film strip spanning infinitely in each direction. Now imagine that each cell of this film strip can contain only one of a defined set of symbols (like an alphabet). For this example let us imagine that each cell can only contain either a "0" or a"1". These cells can be rewritten an infinite amount of time but retain their information indefinitely until they are changed again. The part of the Turing Machine known as the head can write symbols to the cells as well as either increment or decrement its position on the film strip by a given integer (whole number) of cells. The next part is the table which holds a given set of instructions for the head to execute such as "move right 4 cells" and "set cell to 1". The fourth and final part of a Turing Machine is its state register whose purpose is to hold the current state of the machine. The state includes the instruction as well as the current data on the tape.

That is how simple the operation of a computer is. When your computer operates, it is actually operating as a turing machine. It processes data held on your computer by a given set of instructions and algorithms. The computer described in this Instructable is a very simplistic model of a computer, but it still operates as one that you can program with a set of instructions that it will follow and execute.

Useful Links:

Wikipedia on Turing Machines: http://en.wikipedia.org/wiki/Turing_machine

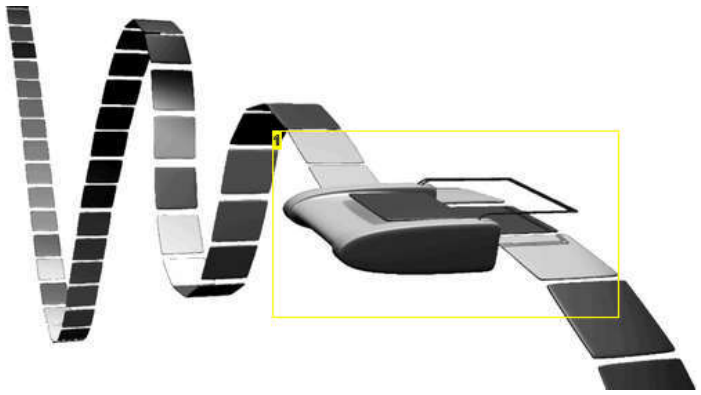


Image Notes
1. An artist's rendition of a Turing Machine from Wikipedia.org

Step 2: An Introduction to Electronics

Before building an 8-bit computer, it is extremely useful to have a grasp on the elemental properties of electricity and analog circuitry. There are parts on the computer you will build will need analog components. There are many electronics self teaching guides available for a minimal cost that provide a crash-course in electrical engineering. I personally found Electronics Self Teaching Guide by Harry Kybet and Earl Boysen to be a wonderful book for tackling the world of analog electronics.

Electronics Self Teaching Guide: http://www.amazon.com/Electronics-Self-Teaching-Guide-Teaching-Guides/dp/0470289619/

Common Components:

Resistor - Limits current, measured in ohms.

Capacitor - Stores current, can either be polar or non-polar (polar meaning that it must be placed in the correct direction to work). Measured in farads.

Diode - Only allows current to flow in one direction, breaks down at a certain voltage when placed in the wrong direction.

Transistor - A current gate that is controlled by a third pin that acts as a mediator. There are many types of transistors, but here we will be talking about the BJT (bipolar junction transistor) which comes in two types: NPN and PNP.

Current, voltage and resistance go hand-in-hand in a circuit. The relation between the three can be expressed with Ohm's law: V = IR. In other words, Voltage equals the current in amperes multiplied by the resistance in ohms. Ohm's law is one of the most important formulas in electronics and it is well worth knowing off of the top of your head.

To apply Ohm's law you need to know the resistance of a circuit. To find the value of a resistor you have to use its color code. The resistor color code is based upon the visible spectrum and can be memorized in many different fashions. For those who don't care to memorize it, there is a plethora of tools that exist to help you find the correct value for your resistor. To calculate total resistance in a circuit you need two formulas for two different configurations of resistors: series and parallel. In series one resistor follows the other one, whereas in parallel they work alongside each other. In series the formula is very simple:

Resistors in Series: R(total) = R(1) + R(2) + . . . + R(N)

Meaning that you just have to add up the values of the resistors.

Resistors in Parallel: R(total) = 1/{ 1/R(1) + 1/R(2) + . . . + 1/R(N) }

A good tool to find resistance from color code: http://www.csgnetwork.com/resistcolcalc.html

It is easier to understand the formula for resistors in parallel if you think of the resistors as working together like two people working together on a project. The same formula is used for word problems where you are given the rate at which two person operate and you must find out how fast their project will be completed if the work together.

To find how much current is supplied to a given component with a given resistance value you would simply plug in the resistance and voltage values into Ohm's law and solve for I. For instance:

A light is in a circuit and and two 1K (one thousand ohm) resistors are placed in front of it in parallel. With a power supply of 9 volts, how much current is supplied to the light?
1.) Calculate R(total):
R(total) = 1/(1/1000 + 1/1000) = 1/(2/2000) = 2000/2 = 1000 ohms
2.) Calculate current using Ohm's law:
9 = I * 1000
I = 9/1000 = .009 A = 9 mA (milliamps)

You can also arrange resistors in a circuit to regulate voltage. This is called a voltage divider and involves two resistors in series. The voltage output of the two resistors is at their junction. For a better idea, look at the picture that I have attached. In this arrangement the formula for voltage output is:

V(out) = V(source) * R(2)/{ R(1) + R(2) }

Capacitors will be useful in your computer with the construction of the clock. The clock is simply a circuit that turns on and off at a constant rate. Just like resistors, capacitors have two formulas for finding the total value for both series and parallel configurations.

Series: C(total) = 1/{ 1/C(1) + 1/C(2) + . . . + 1/C(N) }

Parallel: C(total) = C(1) + C(2) + . . . + C(N)

The rate at which a capacitor charges depends upon the resistance of the circuit before (or after if you are discharging) the capacitor as well as its capacitance. The charging of a capacitor is measured in time constants. It takes 5 time constants to fully charge or discharge a capacitor. The formula for finding the time constant of a capacitor in seconds is:

T(constant) = Resistance * Capacitance

Diodes are simple in operation and come in handy when building a TTL computer. They only allow current to flow in one direction. When they are placed in the correct direction they are what is called forward-biased. When they are reversed they break down at a certain voltage. When a diode is working against the current it is reverse-biased.

A Transistor operates like a valve that is operated by current. A BJT has three pins: the collector, the emitter and the base. For sake of simplicity in this step I will describe a NPN transistor in which current flows from the collector to the emitter. The current applied at the base controls how much of the current flows from the collector to the emitter. Transistors are ideal for many applications due to their ability to amplify a signal. This is because the current applied at the base of the transistor can be considerably less than the current controlled. This gain in current is called the current gain of the transistor, or beta. The formula for beta is:

Beta = Current(Collector)/Current(Base)

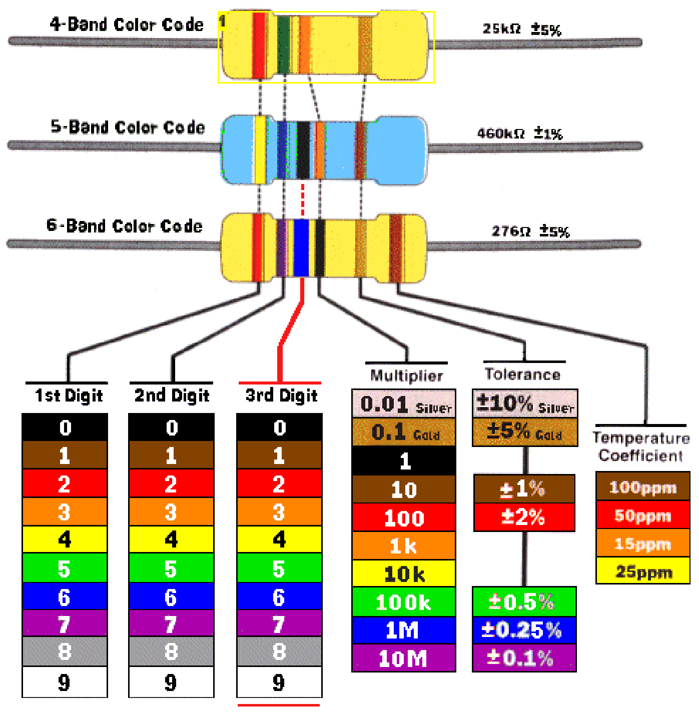
When a transistor is completely on it is said to be saturated. A boolean transistor is one that is either in its saturated or off state and never in between. This is the type of transistor that you will be dealing with mostly in digital electronics. Transistors form the logic gates needed for a computer to function. These will be described later.

Useful Links:

http://en.wikipedia.org/wiki/Resistor
http://en.wikipedia.org/wiki/Capacitor
http://en.wikipedia.org/wiki/Diode
http://en.wikipedia.org/wiki/Transistor

Image Notes
1. Series
2. Parallel

Image Notes
1. A typical capacitor. This one has a capacitance of 470 microfarads (millionths of a farad).


4-Band Color Code
25kΩ ±5%
5-Band Color Code
460kΩ ±1%
6-Band Color Code
276Ω ±5%
1

1st Digit	2nd Digit	3rd Digit	Multiplier	Tolerance	Temperature Coefficient
			0.01 Silver	±10% Silver	
			0.1 Gold	±5% Gold	
0	0	0	1		
1	1	1	10	±1%	100ppm
2	2	2	100	±2%	50ppm
3	3	3	1k		15ppm
4	4	4	10k		25ppm
5	5	5	100k	±0.5%	
6	6	6	1M	±0.25%	
7	7	7	10M	±0.1%	
8	8	8			
9	9	9			

Image Notes
1. The most common

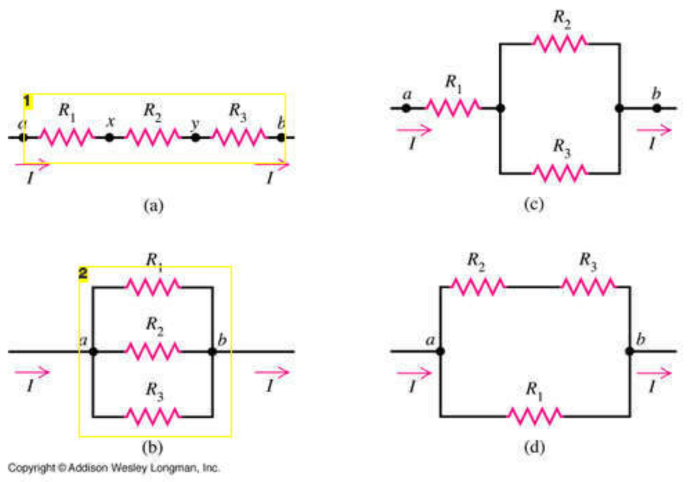


Image Notes
1. Series
2. Parallel

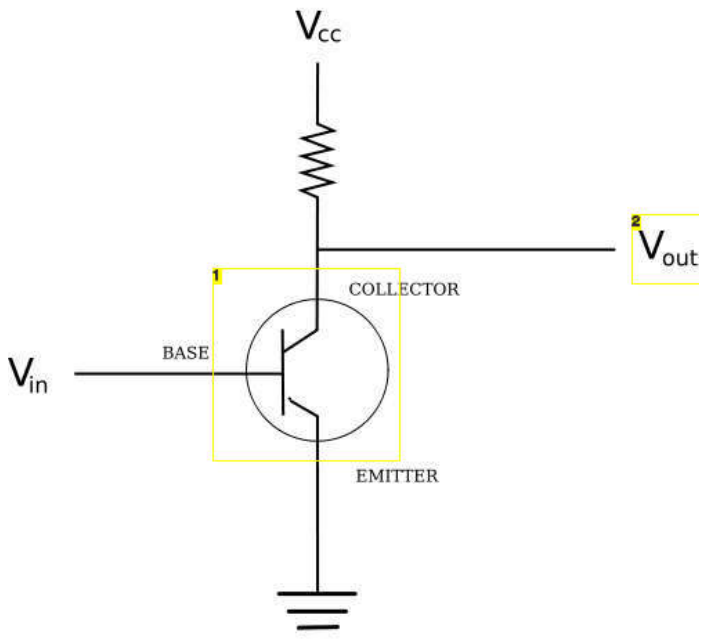


Image Notes
1. The symbol for a transistor usually has an arrow pointing in the direction of the current.
2. The voltage out in this case will be the inverse of the current flowing through the transistor. The current flowing through the transistor will divert current from the output.

Step 3: Binary Numbers

Today we are used to a worldwide numbering system that is based on the number ten. By that I mean that we have no numeral in our number system that is worth the value of ten and thus our number system is base ten.

Think of our number system as an odometer. An odometer counts from the lowest digit to the highest digit and then forces the next rotor in sequence to advance one place. For example:

0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0 9
1 0 <-- Carry to the next digit

Binary is base two, meaning that it only has two numerals and has no numeral for 2. Binary only has the numerals 0 and 1 or "off" and "on". To count in binary you simply apply the odometer technique:

0001b - 1
0010b - 2
0011b - 3
0100b - 4
0101b - 5
0110b - 6
0111b - 7
1000b - 8
etc . . .

There is another factor of our number system that makes it base ten; as we move higher in digits the weight of numerals increase by a power of ten. For example 1 = 10^0, 10 = 10^1, 100 = 10^2, etc . . . In binary, things of course are base two and as such, each successive numeral is another power of two. 1b = 1 = 2^1, 10b = 2 = 2^1, 100b = 4 = 2^2, etc . . .

To convert a decimal number to binary there is a simple trick known as double-dabble that makes the process a lot more easy:

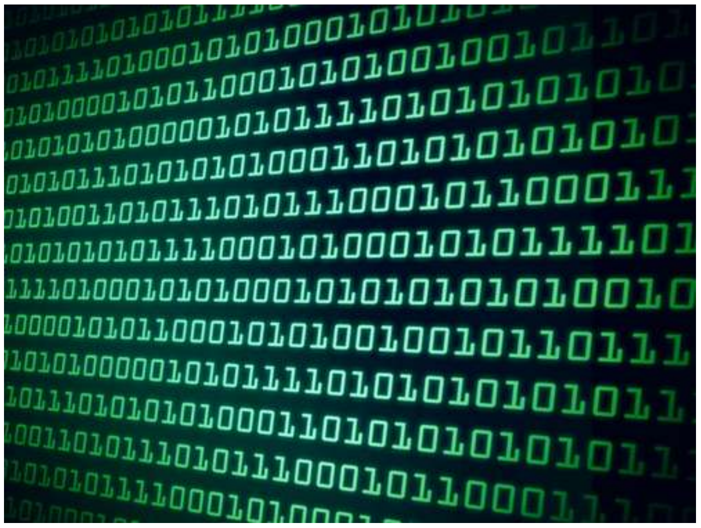
Say we want to convert 13 to a binary number, we start by dividing 13 by two and writing down the remainder. Then directly above it you write down the resulting number without the remainder (6 in this case) and divide that by two and write down the remainder above the previous one. You continue this process until you reach either a 1 or a 0. At the end you read from the top down to get the result.

1/2 = 0 R1 < Read from top to bottom. The result is 1101 or 2^0 + 0 + 2^2 + 2^3 = 1 + 0 + 4 + 8 = 13. This is called a binary word.
3/2 = 1 R1 <
6/2 = 3 R0 <
13/2 = 6 R1 <

Hexadecimal is used very often with binary. Hexadecimal is base 16 and contains the numerals 0-9 and a-f. One hexadecimal numeral is used to describe one nibble or four bits of data. A bit is a single 1 or 0 of binary. A nibble can count from 0 to 15 (0000 to 1111) before the next bit is in the next nibble. Two nibbles together is a byte or 8 bits. Since the first numeral is 2^0, the last numeral is weighted 2^7. Therefore a byte can be anywhere in the range from 0 to 255. To express the byte 00101110 (46 in decimal) in hexadecimal you would first separate the two nibbles into 0010 and 1110. The first nibble has a value of 2, and the second one has a value of E (or 14 in decimal). Therefore the byte 00101110 in hexadecimal would be 2E.

Useful Links:

http://en.wikipedia.org/wiki/Binary_numeral_system
http://en.wikipedia.org/wiki/Hexadecimal



Step 4: Logic Gates

A computer consists of thousands of logic gates arranged to carry out certain functions. A logic gate is a component in digital electronics whose output depends on the state of its inputs. Most logic gates have two inputs and one output. You can think of logic gates as the decision-makers in digital electronics. The six main logic gates used in digital electronics are:

AND Gate: Output is high when all if its inputs are high.

OR Gate: Output is high when any of its inputs are high.

NOT Gate: Only has one input. Output is high when its input is low.

NAND Gate: Output is high unless all of its inputs are high.

NOR Gate: Output is high when none of its inputs are high.

XOR Gate: Output is high when an odd number of inputs are high.

Tri-State Buffer: A buffer that is controlled by a third logic signal.

It is important to mention now the difference between a high "1" signal and a low "0" signal. A high signal can either be a connection to positive voltage or it can be a floating input. A floating input is one that is not connected to any output. An example of a floating input would be one that is not connected at all or one that is connected to the output of a 3-state buffer that is not activated. A low signal is present when an input is at ground.

Logic gates can be fed into each other to produce almost any function imaginable. For instance, two NOR gates can be fed into each other to store one bit of data in a RS_NOR latch while power is supplied to the circuit.

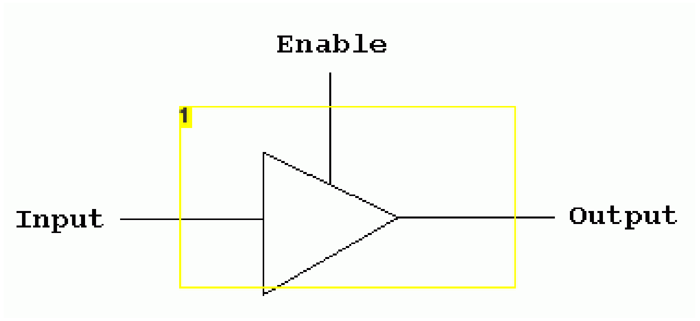


Image Notes
1. A Tri-State buffer is like a switch between the input and output that is controlled by an enable signal.

Image Notes
1. These are called truth tables.

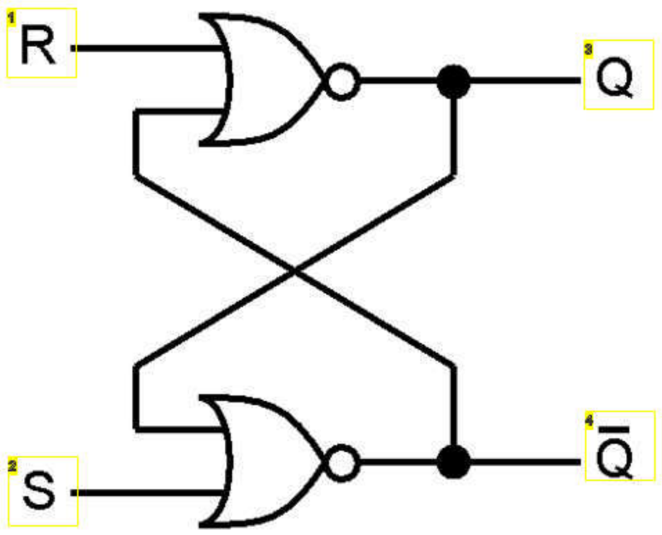


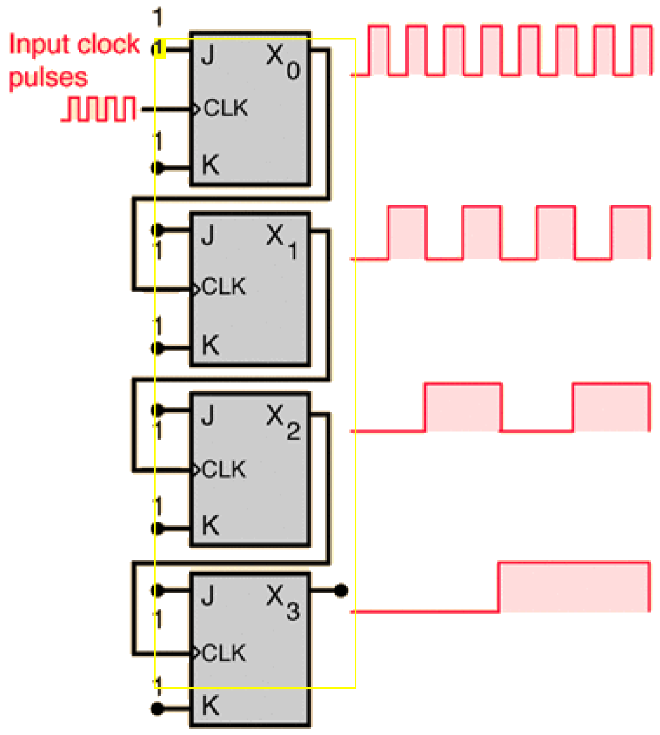
Image Notes
1. Reset (write 0)
2. Set (write 1)
3. Output
4. Opposite of output

Step 5: Binary Counting (The Program Counter)

One of the most essential parts to a computer is its program counter. The program counter provides the computer with the current address of the instruction to be executed. In order for the program counter to work, however, it needs to count in binary. To do this JK flip flops are used. A flip-flop is an arrangement of logic gates that stores one bit (like the RS_NOR latch described in the logic gates step). A JK flip-flop changes its state when its clock pulse input goes high and then low again (its J and K inputs also have to be high). In other words, whenever a JK flip flop gets the falling edge of a clock pulse its state changes from either a "0" to a "1" or from a "1" to a "0".

If you connect the output of one JK flip flop to another and cascade them the result is a binary counter that acts like an odometer. This is because as the first JK flip flop in the sequence goes high, and then low, it triggers the next one in the sequence. The clock's frequency (how many times it turns on and off a second) is halved with every successive addition of a JK flip flop. That is why a JK flip-flop is also called a divide-by-two circuit. The resulting pattern for four JK flip flops will be 0000, 0001, 0010, 0011, 0100, etc . . .

For the simple-as-possible computer described in this Instructable, however, there are a few more functions that you need in order to make the computer operational. In order for the computer to be able to restart its program it needs the ability to clear or set all of its outputs to zero. The program counter also needs the ability to load a binary word for the JMP op code which allows the computer to jump to a certain instruction. Logic gates are used to achieve this goal. Fortunately for us binary counters come in convenient chips with all of the functions that you need.



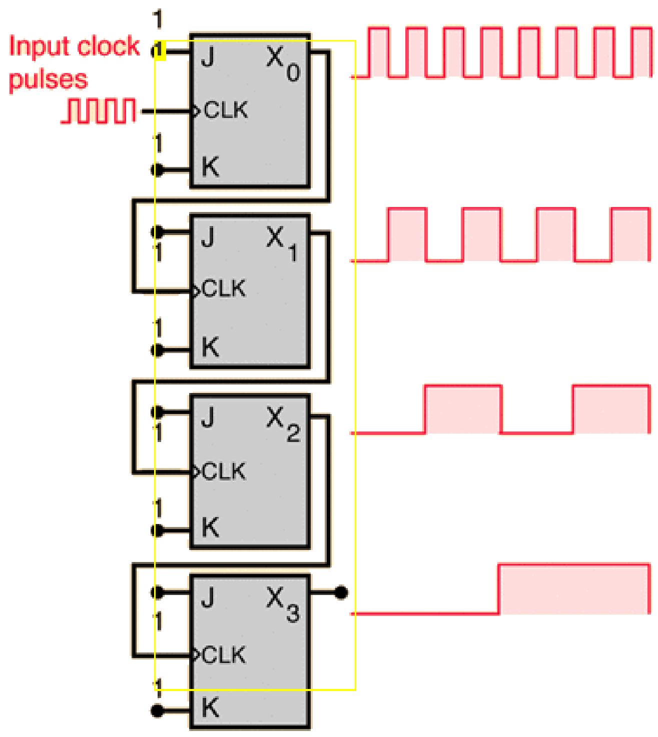


Image Notes
1. The 1's signify high inputs.

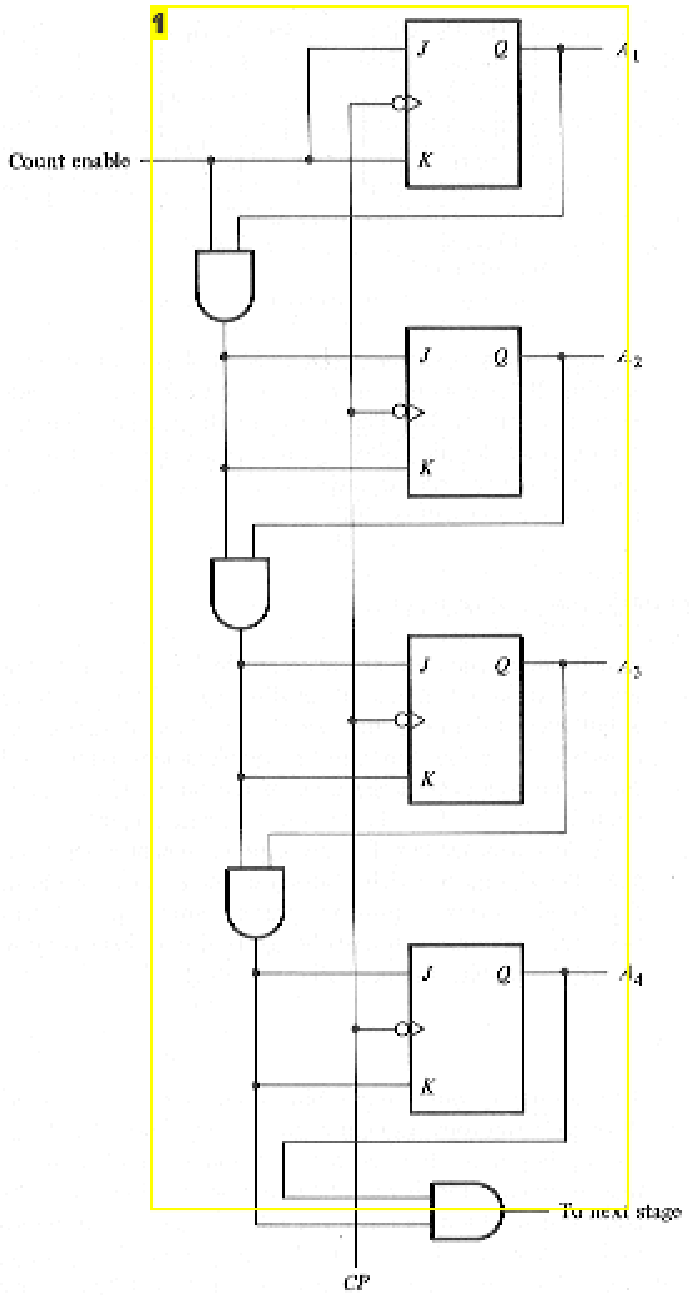


Image Notes
1. This counter utilizes a synchronous clock pulse, meaning that the clock pulse arrives at every flip-flop at once. Flip flops only change state in this case if the previous bit is high.

Step 6: Registers

Registers:

Registers could potentially be the most important part of a computer. A register temporarily stores a value during the operation of a computer. The 8-bit computer described in this Instructable has two registers attached to its ALU, a register to store the current instruction and a register for the output of the computer.

Depending on the chip, a register will have 2 or 3 control pins. The registers that we will be using have two control pins: output enable and input enable (both active when low). When the output enable pin is connected to ground the currently stored binary word is sent out across the output pins. When the input pin is connected to ground the binary word present on the input pins is loaded into the register.

An example of the use of a register on a computer is the accumulator on the ALU (arithmetic logic unit that performs mathematical operations). The accumulator is like the scratchpad for the computer that stores the output of the ALU. The accumulator is also the first input for the ALU. The B register is the second input. For an addition operation, the first value is loaded into the accumulator. After that the second value to be added to the first value is loaded into the B register. The outputs of the accumulator and B register are fused open and are constantly feeding into the ALU. The final step for addition is to transfer the output of the operation into the Accumulator.

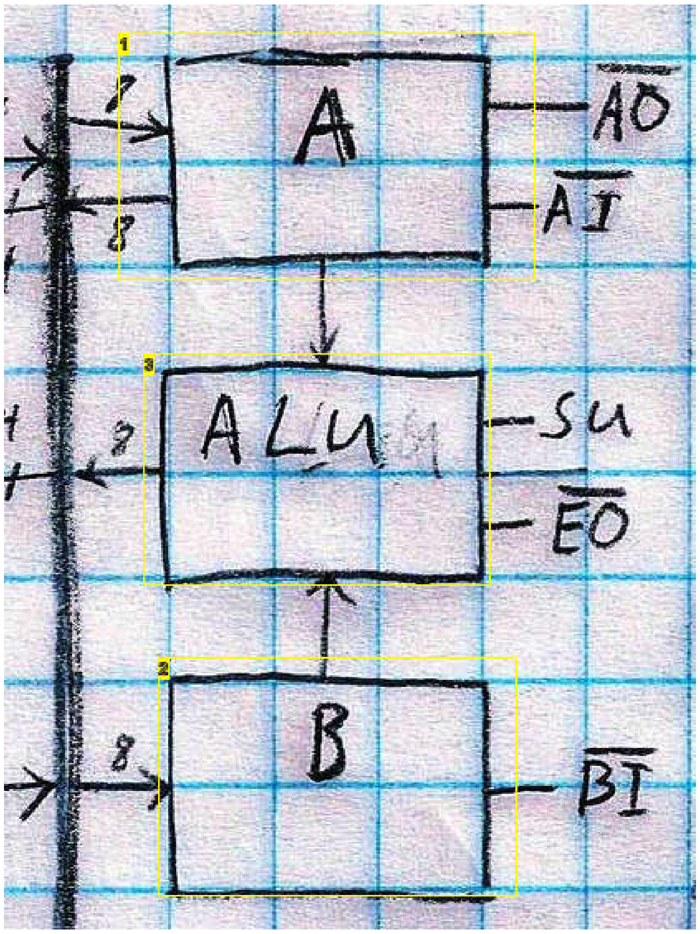


Image Notes
1. Accumulator (A register)
2. B register
3. Arithmetic logic unit

Step 7: The ALU

The ALU (arithmetic logic unit) of a computer is the part that executes mathematical operations. For the SAP computer it will only need to have two functions: addition and subtraction. Adding and subtracting in binary works very similarly to addition and subtraction in decimal terms, for example:
1<-- Carry 1 1 <-- Carry Bits
05 0101
+05+0101
10 1010

To add binary we need what is called a full-adder. A full-adder effectively adds one bit of binary to another with a carry in and carry out. The carry in of a full adder is like a third input for the addition process. They are used to chain multiple full-adders together. The carry out of a full-adder occurs when there is a pair of ones in the addition process. The carry out of a full adder is fed into the carry in to add multiple bits of binary. To construct a full adder you need two XOR gates, two AND gates and an OR gate.

To subtract binary we need to convert a number to its negative counterpart and add it to the number we are subtracting from. To do this we use what is called 2's compliment. To take the 2's compliment of a binary word you invert each bit (change every 0 to a 1 and every 1 to a 0) and add one.

5 = 0101, -5 = 1010+1 = 1011

Not used-->1 1
10 1010
+(-5)+1011
5 0101

To control the inversion of bits we use XOR gates with one normally low input. With one normally low input, the output is equivalent to the other input. When you set the control input high you invert the other input. If we couple this inversion with a bit sent to the carry in of the full adders a subtraction operation is the result.

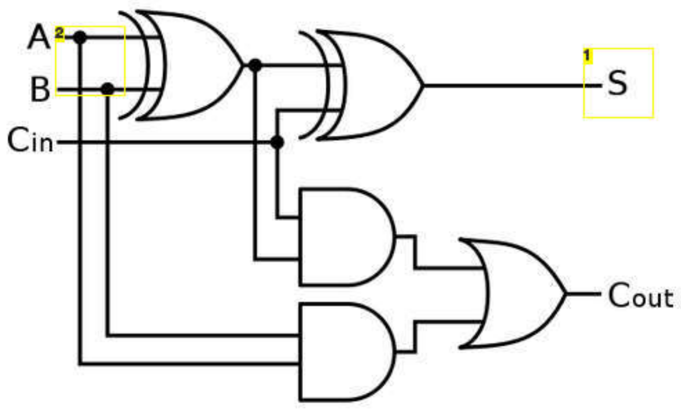


Image Notes
1. Sum
2. Wires only make contact when there is a dot

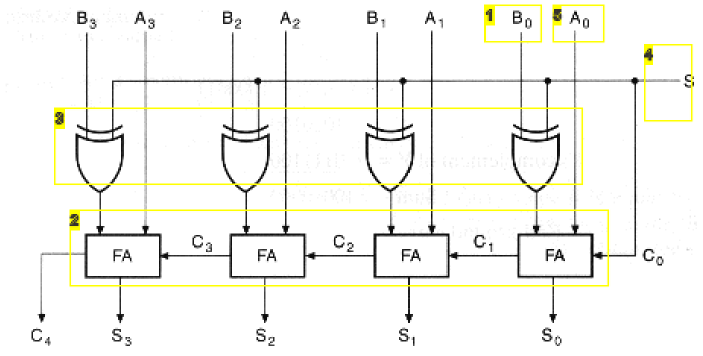


Image Notes
1. Input B
2. Full adders
3. Controlled inversion
4. High for a result of A-B, low for a result of A B
5. Input A

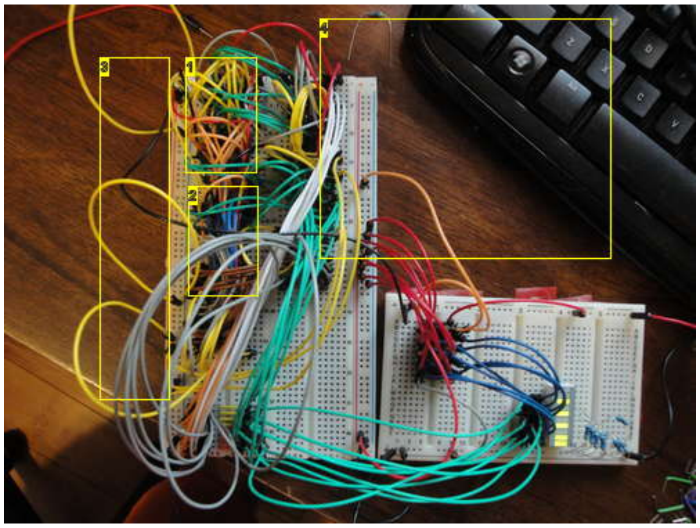


Image Notes
1. XOR chips
2. Full adders
3. Yellow control wires
4. The 8-bit ALU for my computer

Image Notes
1. These are called truth tables.

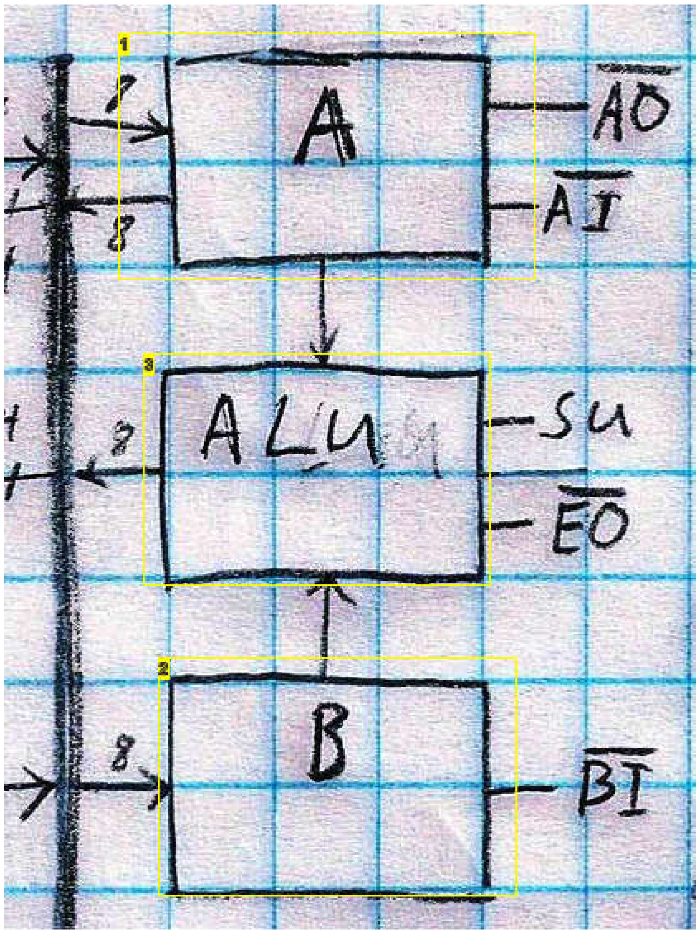


Image Notes
1. Accumulator (A register)
2. B register
3. Arithmetic logic unit

Step 8: Program Memory and RAM

The program memory of your computer will store the instructions to be executed. It will also act as RAM that can store values during the operation of the computer. The program memory consists of three main parts: the memory, the memory address register (MAR) and the multiplexer. The memory is a chip that has 16 bytes of storage. There is a four bit address that is fed into the memory that tells it what byte it should read or write. The MAR stores the current address for the byte to be read or written from the memory. It is constantly feeding into the memory chip unless the computer is in its programming state. A multiplexer allows you to choose between two inputs and output the given input. The multiplexer used in my computer allows you to select from two four bit inputs (the MAR and a manual input). When the computer is in its programming state the manual address is fed into the memory and allows you to program bytes into the computers memory at the address that you define.

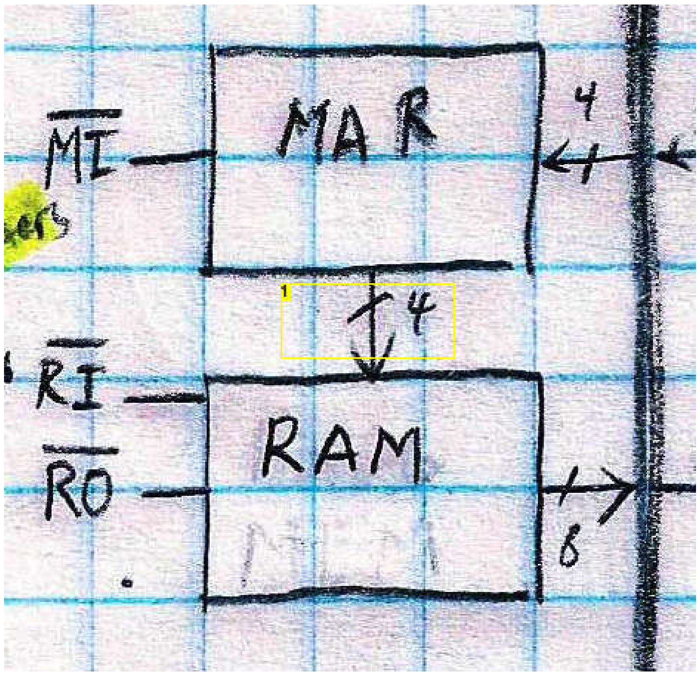


Image Notes
1. The multiplexer is not shown here.

Step 9: Instruction Register

The instruction register of a computer stores the current instruction as well as an address that the instruction will operate on. It is a very simple component with a very important purpose. During the operation of the computer, the content of a given address in memory is transfered into the instruction register. In my computer the leftmost fout bits are the OP code or current instruction to be carried out. The right four bits, or lowest four bits, tell the computer what address to use for the operation. The first four bits constantly feed the OP code into the control matrix which tells the computer what to do for a given instruction. The rightmost four bits feed back into the computer so that the address can be transferred into the MAR or program counter.

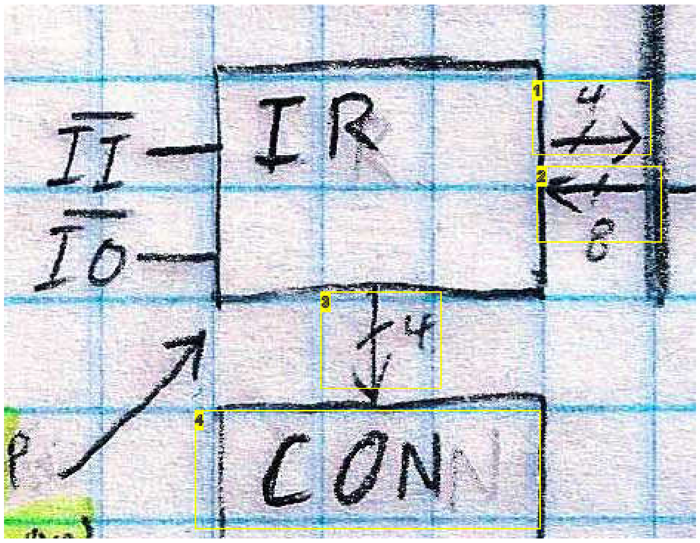


Image Notes
1. Lowest four bits
2. Byte input
3. Highest 4 bits (OP code)
4. Control matrix

Step 10: Output Register

If a computer were to just feed the output of the bus to the operator, the readout would make little to no sense. This is why there is an output register whose purpose is to store values meant for output. The output for your computer can either be simple LED's that display raw binary, or you could have a display that reads out actual numbers on seven-segment displays. It all depends how much work you want to put into your computer. For my computer I am using some IV-9 Russian Numitron tubes for the output of my computer coupled with an Arduino to do the conversion from binary to decimal.

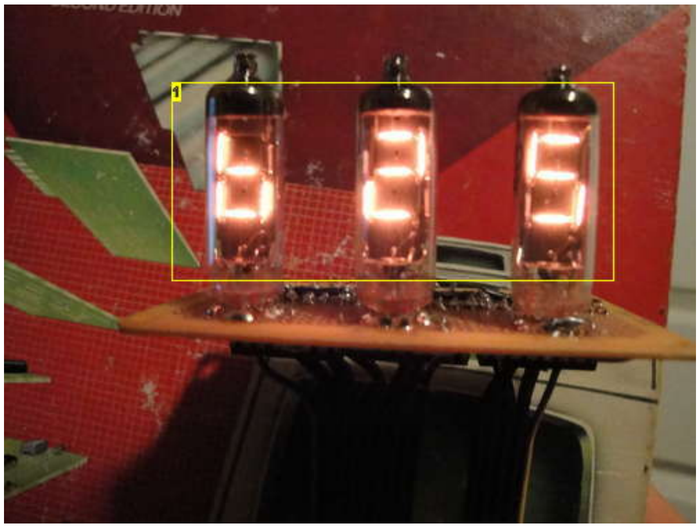


Image Notes
1. This is the output for my computer. Obviously this is a test considering that 625 is way outside of the 0-255 range for one byte.

Image Notes
1. Be creative with the display.
2. Output register

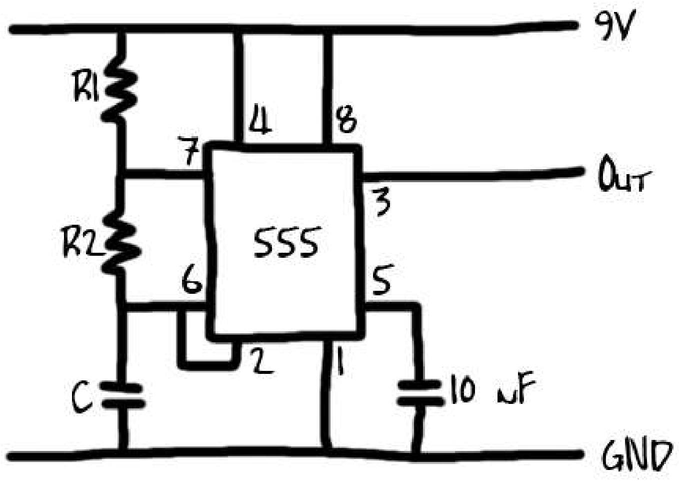
Step 11: Clock

Every part in the computer has to be completely synchronized in order to function correctly. In order to do this your computer needs a clock or a circuit that has an output that turns on and off at a constant rate. The easiest way to do this is to use a 555 timer. The 555 timer is an extremely popular timer that was invented in the era of the emergence of the computer that is stille extremely popular with hobbyists today. To build the 555 circuit you need to know how one operates.

The clock for your computer should be relatively slow at first. 1Hz, or one cycle per second, is a good starting value. This allows you to view the operation of your computer and check for any errors. The 555 chip needs two resistors and a capacitor for operation. The two resistors determine how long the high and low pulses are as well as the overall frequency. The capacitor changes the pulse length for both. If you do not have any experience with 555 timers I recommend experimenting with them.

Useful Links:

http://en.wikipedia.org/wiki/555_timer_IC



Step 12: Architecture

This is the step where everything comes together. It is time to design the architecture of your computer. The architecture is how the registers and different components of your computer are organized. The design aspect is completely up to you, although it helps to keep a goal in mind (what you want your computer to do) and a model to go off of. If you want to design your computer after mine it is completely fine. I modified the architecture of the SAP-1 found in Digital Computer Electronics for my 8-bit computer.

One design aspect to always keep in mind is how data is transferred between the various components of your computer. The most common method is to have a common "bus" for all of the data on the computer. The inputs and outputs of the registers, ALU, program counter and RAM all are connected to the computer's bus. The wires are arranged in order from least significant bit (1) to highest significant bit (128).

Any and all outputs that are connected to the bus have to be completely disconnected while inactive or else they would merge with each other and result in erroneous output. To do this we use Tri-state buffers to control the output of certain elements that output by default like the accumulator, ALU and actual input for the programming of the computer.

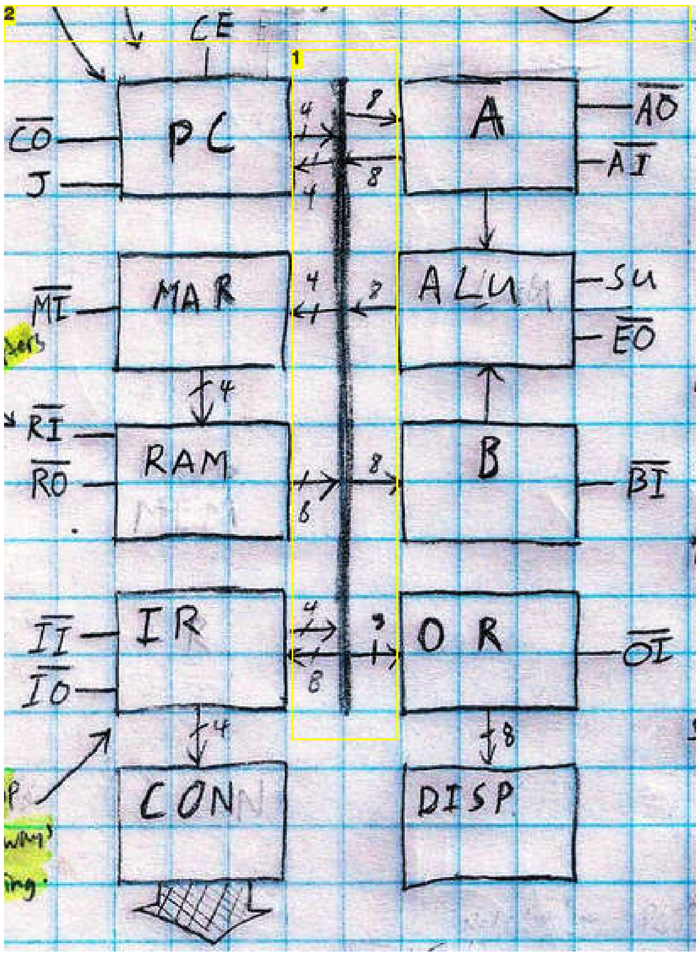


Image Notes
1. This is the common bus. The numbers indicate the bus width, or how many bits are carried on each bus.
2. This is the architecture for my computer. All of the lines going off to the side are control pins.

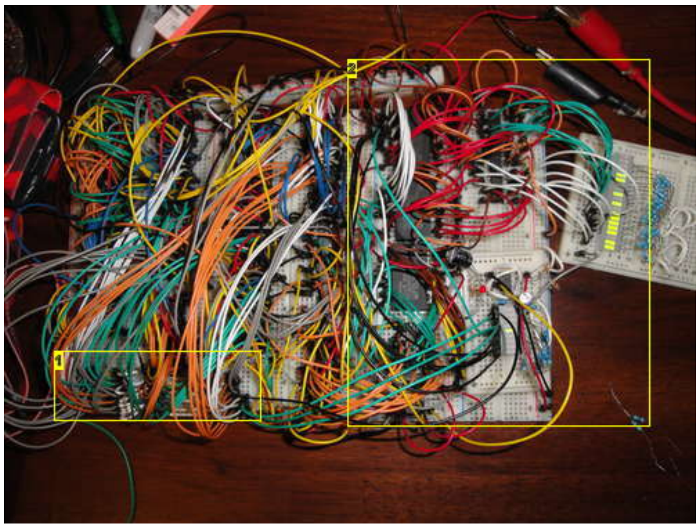


Image Notes
1. All of these wires connect to a common bus on my computer.
2. This is the control matrix for my computer.

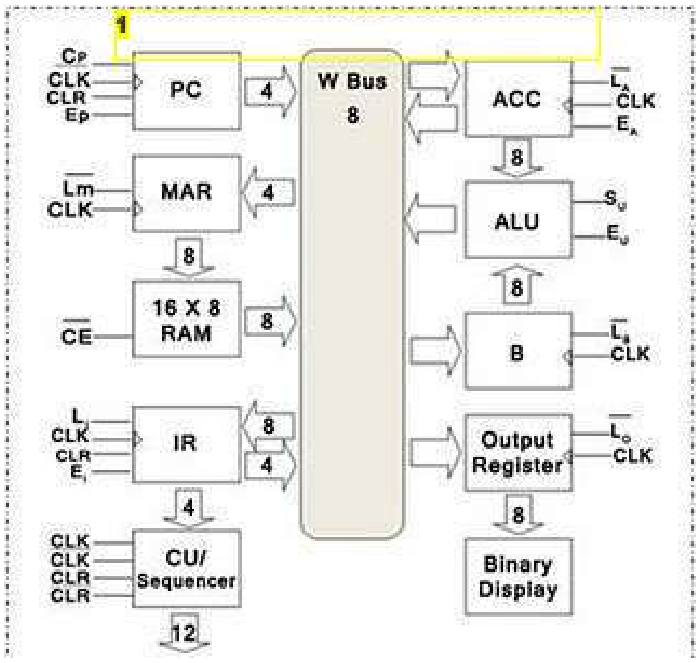


Image Notes
1. This is the SAP-1 architecture found in Digital Computer Electronics

Step 13: Control Matrix

The control matrix of a computer tells each individual part when to take input and output its value. There are multiple states of each operation in a computer. These states are triggered by a type of counter called a ring counter. A ring counter only has one bit high at a time and cycles through its outputs consecutively. For instance, if a ring counter has 4 outputs it will first have its first output active. At the next clock pulse it will set its second output high (and the first low). The next clock pulse will advance the bit one output higher and so on. These stages are called T states. The computer in this Instructable uses 6 T states for the operation of one command. The first three T states are what is called the fetch cycle in which the current instruction is fetched and placed into the instruction register. The program counter is also incremented by one. The second set of three T states depends on what OP code is fed into the control matrix from the instruction register. The T states are as follows:

T1: The contents of the program counter are transferred into the memory address register. (Address State)
T 2: The program counter is incremented by one. (Increment State)
T3: The addressed byte in the program memory is transfered into the instruction register. (Memory State)
T4: Dependent on what command is being executed.
T5: Dependent on what command is being executed.
T6: Dependent on what command is being executed.

There are two primary ways to create a control matrix: using discrete logic and using ROM's. The ROM method is the easiest and most efficient. Using discrete logic involves designing a massive logic schematic that will output the correct control words for your computer based on an OP code input. ROM stands for read-only-memory. There are several types of ROM's that you can consider for use in your build. For my computer I originally used EEPROM (electronically erasable programmable ROM) but then shifted to NVRAM (non-volatile random access memory) after the EEPROM chips failed to write. I do not recommend NVRAM as it is meant for random access memory and not permanent storage. EEPROM is the most efficient solution in my opinion.

The control matrix will have three ROM chips each having at least 32 addresses of 8 bit storage (as well as the timing and counting elements). The binary word that is sent out from the control matrix is called the control ROM and contains all of the control bits for every component of your computer. You want to be sure to organize the control bits and know their order. For no operation you want a control word that renders every part of the computer inactive (except the clock of course). The control word for the computer described in this Instructable is 16 bits in length and is stored in two of the control ROM chips. The first three addresses of the control ROM chips hold the control words for the fetch cycle. The rest of the addresses on the chip hold the control words in pairs of three for each OP code. The third ROM chip holds the memory location for the start of the control word sequence for each OP code and is addressed by the OP code itself. For instance, in my computer if you give the control the OP code 0110 it will output binary 21, which is the address of the start of the JMP command. There is an 8-bit counter in between the OP ROM and the control ROM's that counts from 0-2 (first three T states) then on the third T state loads the address outputted by the OP ROM and counts from that position until the T1 state clears the counter again. The ring and binary counter for the control matrix are controlled by an inversion of the clock pulse so that control words are present when the rising clock pulse goes to the elements of the computer. The entire process in order is as follows:

1.) T1 state clears the counter to 0, the control word stored at 0 is sent out
2.) The clock goes high and the address state takes place
3.) The clock goes low and in turn the control counter increments and control word 1 is sent out
4.) The clock goes high and the increment cycle takes place
5.) The clock goes low and the control counter increments to 2, control word 2 is sent out
6.) The clock goes high and the memory state takes place and the OP code arrives at the instruction register, T3 is also active which means on the next low clock pulse the OP control address will be loaded
7.) The clock goes low and loads the counter with the address for the first of the three control words for the given OP code
8.) T4, T5 and T6 execute the OP code
9.) T1 resets the counter, the process continues until a HLT OP is received. The HLT command stops the clock.

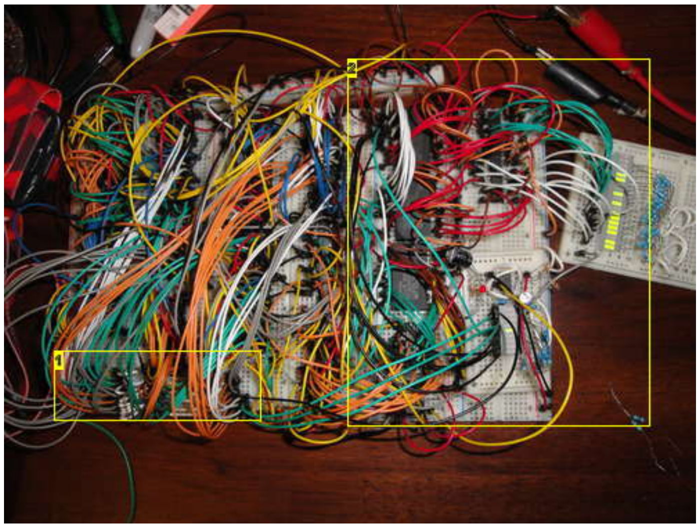


Image Notes
1. All of these wires connect to a common bus on my computer.
2. This is the control matrix for my computer.

Step 14: Microprogramming

Now is the part where you decide what commands you want your computer to be capable of. I gave my computer 6 unique operations that would give it the basic programming functions that I would need. The commands that you will program into your computer are what is called Assembly language. Assembly is one of the earliest programming languages and can still be used on computers today. Commands in the language include loading the accumulator, adding, moving, outputting and storing variables. Each command has its own 4-bit OP code in this 8-bit computer. The commands that I chose for my computer are:

NOP: No operation. (0000)
LDA: Load the accumulator with the value at this address. (0001)
ADD: Add the value at the specified address to the value in the accumulator. (0010)
SUB: Subtract the value at the specified address from the value in the accumulator. (0011)
STO: Store the accumulator's contents at the specified address. (0100)
OUT: Store the accumulator's contents in the output register so that the operator can see it. (0101)
JMP: Jump to a certain instruction in memory at the specified address. (0110)
HLT: Stop the operation of the computer. (0111)

To determine what control words need to be sent out for each OP you need to know what bits have to be active during each T state. For my computer I organized the bits as follows (an underline denotes an active-low bit):

CE CO J MI RO II IO OI BI EO SU AI AO RI HLT X

CE - Count Enable (enables the program counter's clock input)
CO - Clock out enable
J - Jump enable
MI - MAR Input
RO - Program memory in
II - Instruction register in
IO - Instruction register out
OI - Output register in
BI - B register in
EO - ALU output enable
SU - Subtract
AI - Accumulator in
AO - Accumulator output enable
RI - Program memory in
HLT - Halt
X - Not used

Here are what bits should be active for each T state for a given instruction as well as the address that they should be in the control ROM:

Fetch:
0: CO, MI - The program counter outputs into the MAR
1: CE - The counter is enabled for the next clock pulse
2: RO, II - The addressed byte is outputted from RAM into the instruction register

NOP:
3: X
4: X
5: X

LDA:
6: IO, MI - The address in the instruction register is transfered to the MAR (lowest four bits)
7: RO, AI - The addressed byte is outputted from memory into the accumulator
8: X

ADD:
9: IO, MI - The address in the instruction register is transfered to the MAR (lowest four bits)

10: RO, BI - The addressed byte is outputted from memory into the accumulator
11: EO, AI - The sum of the accumulator and the B register is loaded into the accumulator

SUB:
12: IO, MI - The address in the instruction register is transfered to the MAR (lowest four bits)
13: RO, BI - The addressed byte is outputted from memory into the accumulator
14: AI, SU, EO - The difference of the accumulator and the B register is loaded into the accumulator

STO:
15: IO, MI - The address in the instruction register is transfered to the MAR (lowest four bits)
16: AO, RO, RI - The accumulator outputs into the program memory at the addressed location (RO and RI have to be active for a write on the chip that I used)
17: X

OUT:
18: OI, AO - The accumulator outputs into the output register
19: X
20: X

JMP:
21: J, IO - The instruction register loads the program counter with its lowest four bits
22: X
23: X

HLT:
24: HLT - A halt signal is sent to the clock
25: X
26: X

Your OP ROM contains multiples of three at each memory location. This is of course because each cycle takes three execution states. Therefore the addressed data for your OP ROM will be:
0 - 3
1 - 6
2 - 9
3 - 12
4 - 15
5 - 18
6 - 21
7 - 24

To program your choice of chip you have many different options. You could buy an EEPROM and EPROM programmer, but they usually cost a considerable amount of money. I built a breadboard programmer for my ROM that is operated by moving wires around and controlling the write and read enable pins by push buttons. Later I simplified the process and designed an Arduino programmer for my NVRAM specifically. I'll attach the code as it can be easily modified to program almost any parallel memory chip that you would use for this project.

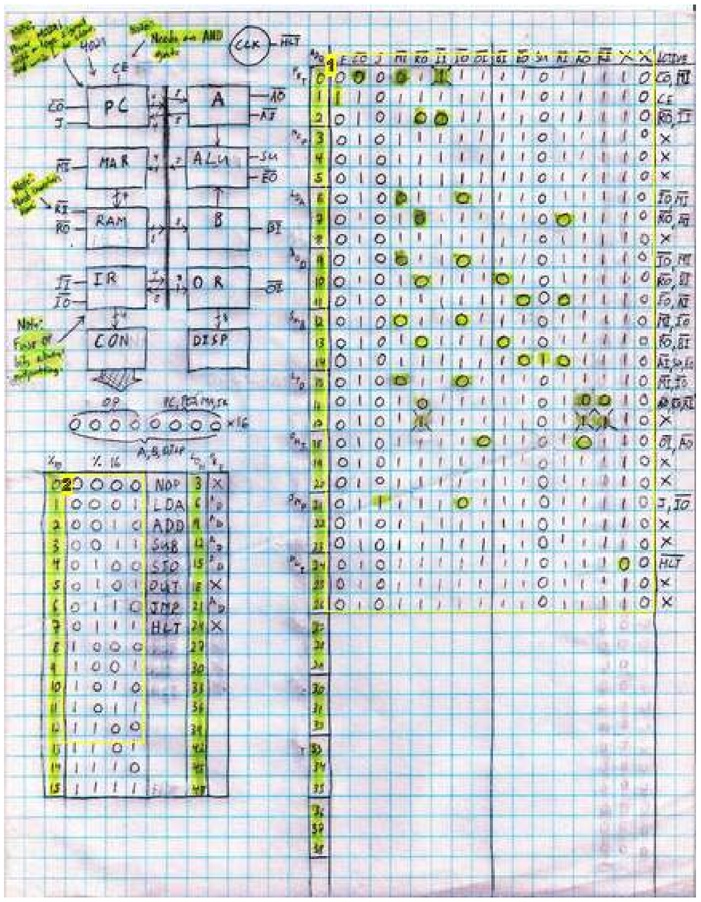


Image Notes
1. This is my breadboard programmer. It uses 3 state buffers for input and output.

Image Notes

1. Control words
2. OP codes

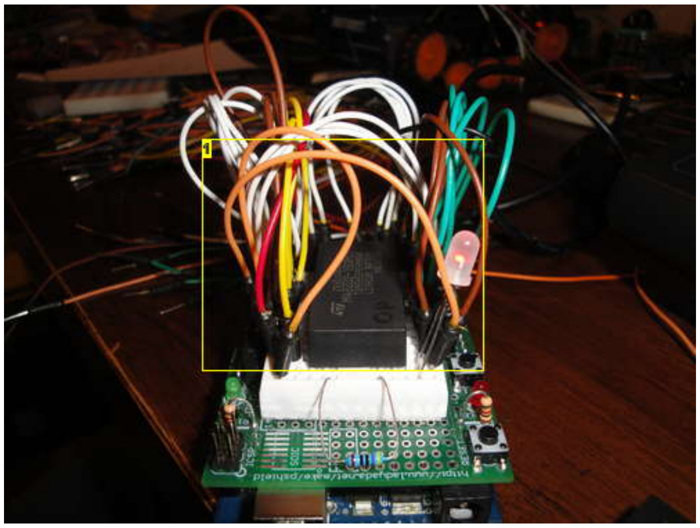


Image Notes
1. This method worked on and off for me. This is probably due to the abnormal voltage levels required for my NVRAM chips.

File Downloads

NVRAM_Programmer.pde (2 KB)
[NOTE: When saving, if you see .tmp as the file ext, rename it to 'NVRAM_Programmer.pde']

Step 15: Buying Parts

The great thing about building an 8-bit computer is that most parts will cost you less than a dollar a piece if you buy them from the correct place. I purchased 90% of my parts from Jameco Electronics and I have been completely satisfied with their services. The only parts I have really bought from anywhere else are the breadboards and breadboard wires (and the Numitron tubes). These can be found considerably cheaper on sites like Amazon. Always be sure to make sure the parts that you are ordering are the correct ones. Every part that you buy should have a datasheet available online that explains all of the functions and limitations of the item that you are buying. Make sure to keep these organized as you will be using many datasheets in the construction of your computer. To help you with your computer I will list the parts that I used for mine:

4-Bit Counter:
74161 - http://www.jameco.com/webapp/wcs/stores/servlet/ProductDisplay?freeText=74161&langId=-1&storeId=10001&productId=49664&search_type=jamecoall&catalogId=10001&ddkey=http:StoreCatalogDrillDownView

4-Bit Register (I use two for each 8-bit register):
74LS173 - http://www.jameco.com/webapp/wcs/stores/servlet/ProductDisplay?freeText=74LS173&langId=-1&storeId=10001&productId=46922&search_type=jamecoall&catalogId=10001&ddkey=http:StoreCatalogDrillDownView

2-1 Multiplexer:
74LS157 - http://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_46771_-1

16x8 RAM (output needs to be inverted):
74189 - http://www.jameco.com/webapp/wcs/stores/servlet/ProductDisplay?freeText=74189&langId=-1&storeId=10001&productId=49883&search_type=jamecoall&catalogId=10001&ddkey=http:StoreCatalogDrillDownView

Full Adders:
74LS283 - http://www.jameco.com/webapp/wcs/stores/servlet/ProductDisplay?freeText=74LS283&langId=-1&storeId=10001&productId=47423&search_type=all&catalogId=10001&ddkey=http:StoreCatalogDrillDownView

Tri-State Buffers:
74S244 - http://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_677020_-1

XOR Gates:
74LS86 - http://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_295751_-1

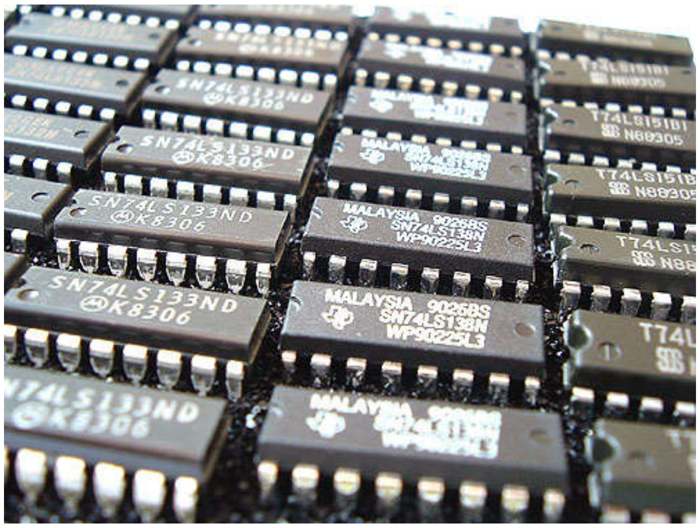
AND Gates:
74LS08 - http://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_295401_-1

NOR Gates:
74LS02 - http://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_283741_-1

Inverters:
74LS04 - http://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_283792_-1

Ring Counter:
CD4029 - http://www.jameco.com/webapp/wcs/stores/servlet/ProductDisplay?freeText=4029&langId=-1&storeId=10001&productId=12925&search_type=jamecoall&catalogId=10001&ddkey=http:StoreCatalogDrillDownView

JK Flip-Flops:
74LS10 - http://www.jameco.com/webapp/wcs/stores/servlet/Product_10001_10001_295427_-1



Step 16: Construction

Here is where the patience really comes in. I chose using a breadboard for the actual computer, but there are many other methods out there (such as wire-wrapping) that will work just as well. To make things a lot more simple I included a block diagram for the actual schematic of my computer. I did not however include part numbers or pin numbers. I believe that this will make things more simple and open for creativity. The 4-bit program counter output, MAR input and instruction register output are all connected to the four least significant bits of the computer's bus.

The second diagram shown is the control logic for the operation end of the computer. The controls are designed so that toggles can be an input for the computer. RS_NOR latches are placed in front of the toggle switches to debounce them. Toggle switches often have dirty connections that may bounce from an on to an off state and provide more pulses than you want. Adding a flip-flop to the output of a toggle eliminates the extra pulses. This would be extremely useful when using the manual clock option. You would not want to flip the switch and initiate 8 clock pulses. The read/write button writes the active input byte to the addressed memory. By changing the default control word input to RAM to two low RO and RI bits initiating a write cycle. The run/program switch changes which input is active on the memory address multiplexer. The JK flip flop after the 555 means that when the computer is run, it will not start on the middle of a clock pulse. A low HLT signal will stop the clock from passing on either the manual clock or 555. And finally, the run/clear switch is connected to all of the clear pins on the computer such as those on the registers and counters.

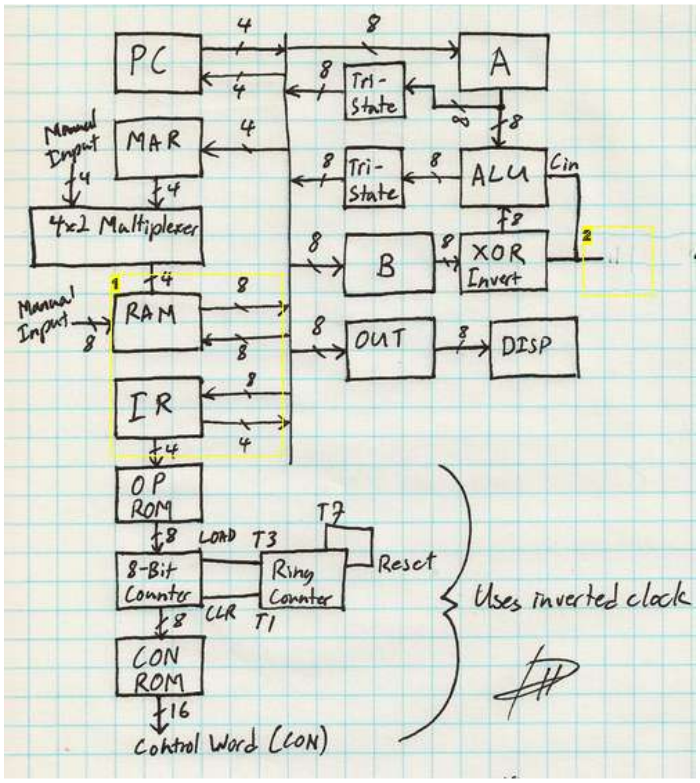


Image Notes
1. The RAM and register chips that I used have tri-state logic inside of them and do not need to be buffered like the output of the ALU or accumulator.
2. High for subtraction, low for addition.

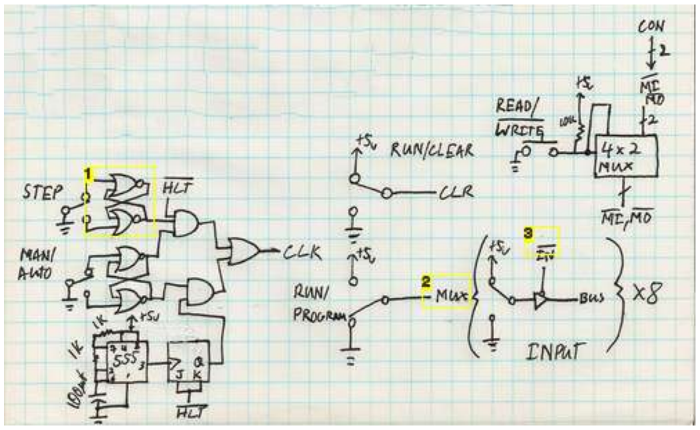


Image Notes
1. RS_NOR latches for debouncing
2. This is also connected to the IN control bit that enables the input for the computer.
3. Input control

Step 17: Programming

Now that the computer is done, it can be programmed to carry out instructions. To do this you first have to put the computer into its program setting by flipping the run/program toggle switch into the program position. After that you select addresses starting at 0 and going to 15 and insert the needed data for your program. For instance, to start with 5 and add 4 with every output the program would be as follows:

```
Address - Data:
0000 - 00010111 LDA 7: Load the accumulator with the value stored at memory address 7 (5)
0001 - 00101000 ADD 8: Add value stored at memory address 8 (4)
0010 - 01010000 OUT: Output the accumulator
0011 - 01100001 JMP 1: Jump to instruction 1
0100 - X
0101 - X
0110 - X
0111 - 00000101 5
1000 - 00000100 4
1001 - X
1010 - X
1011 - X
1100 - X
1101 - X
1110 - X
1111 - X
```

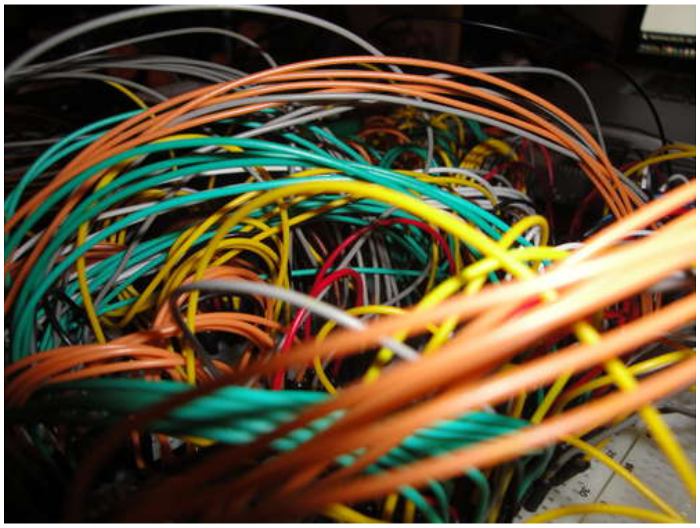
Image Notes
1. Big geek points if you know what this is

Step 18: Going Further

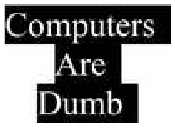
I hope you enjoyed this Instructable and, most of all, I hope that you got something out of it. You can consider all of this hard work an incredibly valuable learning experience that will give you a better understanding of electronics, computers and programming. In the end you will also have something very cool to show for all of your hard work as well.

After constructing your first 8-bit computer you can go further and add more functionality. The ALU used in this computer is very simplistic in operation and true ALU's today have a myriad of functions such as bit-shifting and logical comparisons. Another important aspect to move onto is conditional branching. Conditional branching means that an instruction depends on the current state of flags set by the ALU. These flags change as the accumulator's contents become negative or are equal to zero. This allows for a much more expansive possibility for application of your computer.

If you have any questions about this project feel free to comment on this Instructable, comment on my website at http://8bitspaghetti.com or shoot me an email at sudokyle@gmail.com . I wish you the best of luck with this project.



Related Instructables

					
Computers are Dumb! by msuzuki777	**Arduino: an easier way to work with seven segment displays** by spel3o	**Pocket Ardiuno kit.** by Bongmaster	**The 74HC164 Shift Register and your Arduino** by osgeld	**4 Bit Binary Calculator** by Teslaling	**Mechanical CPU Clock** by lelazary