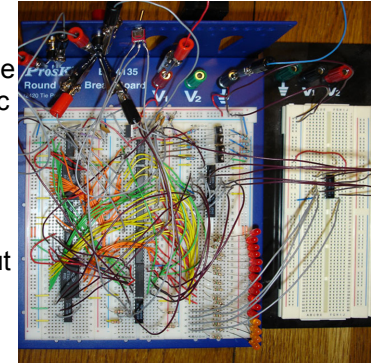


1-Building a Computer from the Ground Up

Many years ago I was taking a digital logic class in college, learning about the primitive circuits that were at the heart of a computer's operation. The class was clear enough and the subject interesting enough that I really wanted to put what I had learned into action. I envisioned designing and building a computer from those simple circuits into a completed (though slow and simple, by modern standards) computer. I decided to build it using "virtual circuits" inside a computer. It would be a simulated computer. I had two reasons for doing this.

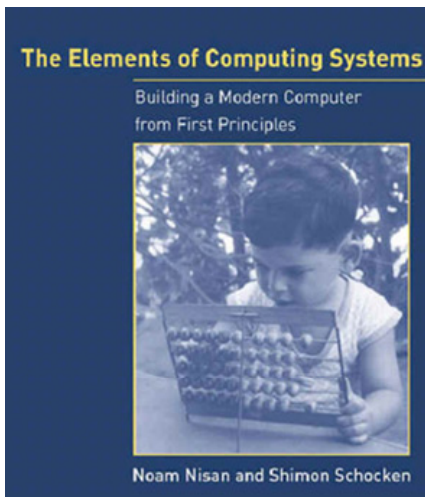
1) I am not an electrical engineer and wanted to focus on the computer's own internal logic. I did not also want to have to learn engineering on the side, just for what amounted to an (involved) hobby. So this would allow me to concentrate my efforts on how the logic components worked together, not on figuring out power levels and resistors and transistors and so forth. A cheat, I know.

2) Even more importantly, building a computer was going to take thousands and tens of thousands of switches. While I wanted to build this thing myself, I didn't want to purchase premade chips with my switches already made into logic components and ready for me to wire up. (Though that would still be a lot of hard work.) At the same time I did want to finish this thing before the end of the next millennium. Just to build 1K of memory, for example, was going to require 7168 components called gates. I didn't want to make hundreds of breadboard circuits that looked like **this!**



So doing it virtually inside a computer seemed like the best situation. I'd still get to design it from the ground up, but could focus on what I wanted to focus on. And because it was simulated, once I designed a component I could just make copies and hook them together however I wanted.

Of course, nothing is that simple. Components like multiplexors, demultiplexors, were easy enough. I even implemented NOR-based flip-flops and eventually a 1K addressable memory module. But, eventually real life caught up with me- a move, a new job, and other things. That, and I had hit a roadblock in the increasing number of problems cropping up with the component simulations and their signal timing. I had to focus on other things for a long while.



Fast forward a number of years. I found this book, [The Elements of Computing Systems: Building a Modern Computer from First Principles](#). It was exactly what I needed. In the book, you actually do build a computer in exactly the manner I had wanted (and for much of the same reasons.) The authors don't tell you how to do it, how things should connect together, or anything like that. Instead, they give the specification for specific components and it is up to you to come up with them. They describe how those components should work together, their behavior for specific circumstances, and it is you who comes up with a design that does exactly that. In short, I would do the work I wanted to do, but with some help knowing what I was going to need next or how things should work.

This is the result of my work. After I designed and tested my computer, I wanted to document what I had done, the design decisions and the things I had learned. From there, I just found myself imagining I was explaining to someone else exactly how to design a computer the way I did, the thought processes and the choices I made.

This project is, in many ways, like building a combustion engine from scratch. Your engine will not rival or even equal those in vehicles today. But making one yourself is still an experience. You find yourself looking at early models, the ones those first engineers designed, and find yourself nodding your head as you now understand the decisions they made. You also marvel at what they were able to accomplish in those early days.

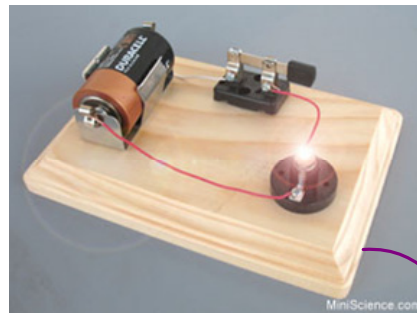
If you would like that experience yourself, or to try any of this out on your own, the authors have a website, www.nand2tetris.org, where you can find the book, assistance, and all the software tools you would need to build the computer yourself.

It has been a great experience. Enjoy!

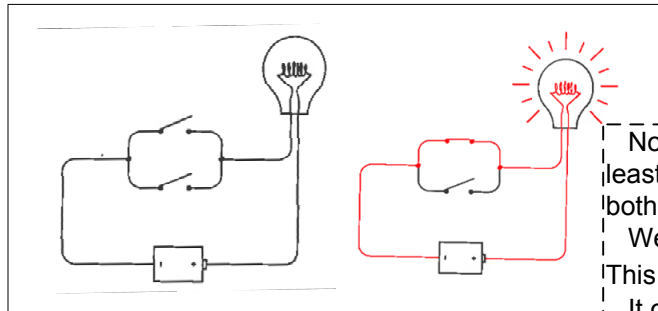
2-Digital Logic Primitive Circuits

We start with a simple switch circuit. This circuit consists of a light bulb, a battery, wires, and a switch. When the switch is closed, the connection to the battery is complete and electricity can flow into the light bulb. The light comes on. When the switch is opened, the circuit is broken and the light goes off. Very simple. But that switch is why a computer can do what it does. On and off can also be called true and false, 1 and 0, closed and open.

Copyright 2013. Ian Ohlander. All Rights Reserved



Switch Circuit



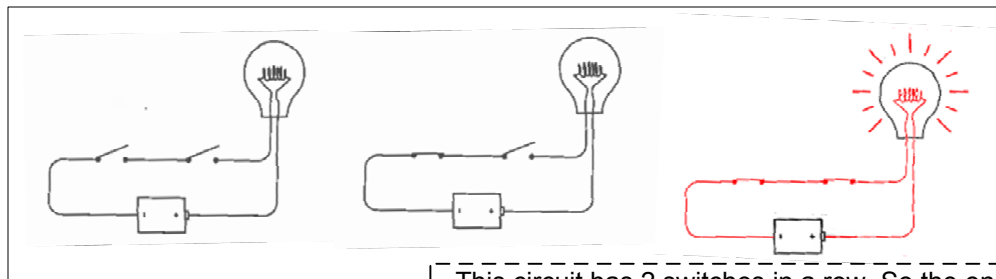
Notice that the circuit is complete as long as at least one switch is on. If the 1st switch OR the 2nd OR both are on, then the light is on.

We used of the word OR describing this circuit.

This is an **OR Circuit**.

It outputs light (or on, true, or 1) when either switch or both is on (or true, on, or 1). It doesn't output light (off, false, or 0) only when both switches are off (or false, 0).

We can also take these two switches and put them in series with each other.

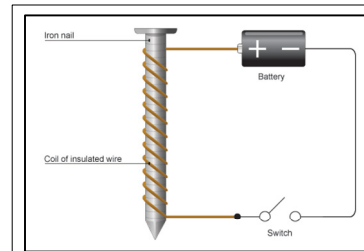


This circuit has 2 switches in a row. So the only way the circuit is complete and the light can be on (true, 1) is if BOTH switches are on (true, 1).

The 1st switch AND the 2nd switch have to be on (true, 1) for the light to be on (true, 1).

We call this an **AND Circuit**.

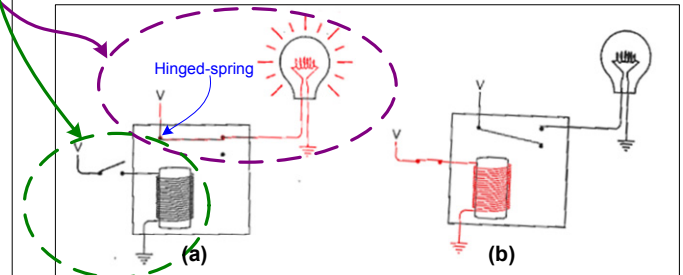
All of these circuits are called GATES. We have an **AND gate**, **OR gate**, and **NOT gate**. We constructed them using parts and technology that has been around since the 1800s to illustrate how basic they are.



A final circuit we can create makes use of an iron rod wrapped with a wire and connected to a battery. When this occurs, the iron rod becomes a magnet. When the circuit is broken and electricity is

no longer flowing through the wire, the iron rod stops being magnetic. We can call this an electromagnet.

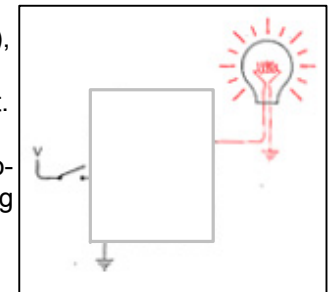
We can use this electromagnet in our final circuit.



We'll first put in a single circuit, as seen in the photograph of the **switch circuit**. But instead of using a normal switch, we'll put one that is on a **hinged-spring** that keeps it closed. Thus the light is always on.

Then we'll put an **electromagnet** underneath the spring-hinged switch, as in the diagram. We'll hide the switch circuit and expose to a user only the switch controlling the electromagnet. To a user, it will look like this.

When we close the switch **(b)**, the electromagnet pulls the latch open, breaking the circuit. The light turns off. When we open the switch **(a)**, the electromagnet shuts off and the spring pulls the switch circuit closed and the light turns on.



For a user, what has happened is that closing the circuit (setting it to 1, on, or true) outputs an off (0, false). But when the user opens the circuit (sets it to 0, off, or false) the circuit outputs an on (closed, 1, true).

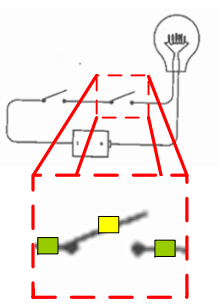
We call this a **NOT circuit**. It outputs the opposite of its input. This is also called *negation*.

3-Digital Logic Primitive Elements

When digital engineers want to design a digital circuit that performs some function, they don't concern themselves yet with how it physically works (at first). This is because the actual components can be implemented in numerous ways.

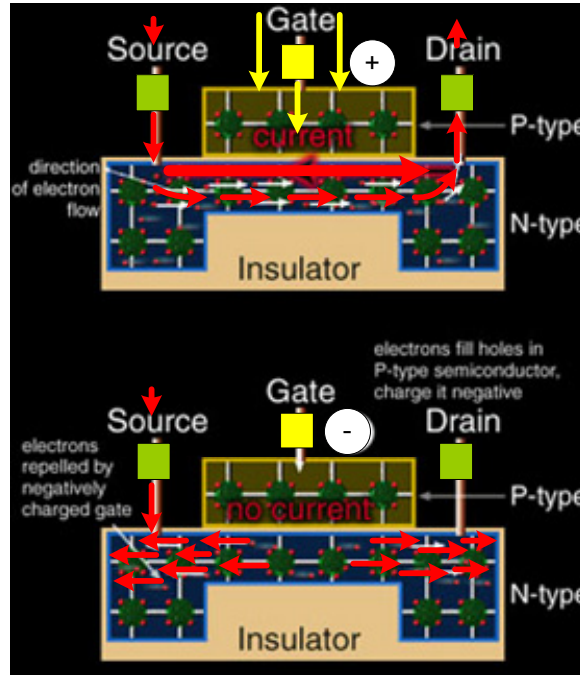
What matters to the designer is their behavior.

As we saw in the previous page, logic gates can be made using magnets and switches.



Here, we see the AND gate, with a focus on one of the switches controlling it.

But switches can also be implemented on silicon chips by exploiting the chemical properties of silicon, as well as silicon mixed (or doped) with other elements. Here is one example of how to do this.



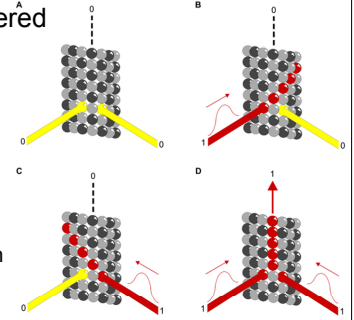
This switch has its two ends (or terminals, *drain* and *source*) connected to the N-type doped sections of silicon. Above the N-type doped section is a terminal called a *Gate* of P-type silicon. If we let the *Gate*'s charge be negative, electrons are repelled from flowing from source to drain (bottom). But when positive voltage is applied to the *Gate* terminal, it stops repelling (or blocking) the flow from source to drain. The two terminals are then connected, closing the circuit, and current flows.

This *non-mechanical* switch can be wired up into logic gates in the same way as mechanical switches.

(excellent video of another type of silicon switch: http://www.youtube.com/watch?v=IcrBqCFLHIY&list=PLkahZjV5wKe_dajngssVLffaCh2gbq55_)

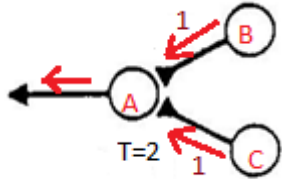
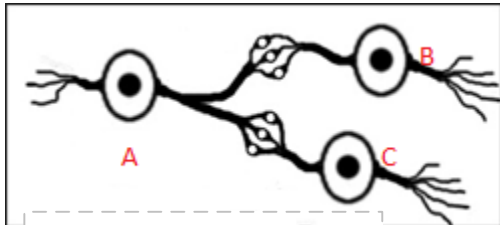
It has even been recently discovered that neurons themselves may contain protein structures inside them that behave as logic components like AND gates.

(<http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1002421>)



The inputs are at the bottom. Only when both are on (true, 1) does the structure output an on (true, 1).

Logic gates can also be made from a literal or simulated network of neuron (nerve) cells arranged in a particular way to create the very same behavior.



Neurons transmit a signal if the voltage sum of the incoming signals is greater than a certain threshold (T).

Here, Cell A is configured so that it will not fire a charge down its axon unless it receives a total incoming charge of at least 2. This only occurs if both inputs are at least 1. Thus, this neural network behaves as an AND gate.

With all of these possible implementations, it becomes clear that the designer doesn't care *how* the logic components work.

As far as he is concerned, they are **abstracted** away as black boxes that can be connected together in specific ways.

Instead, he uses symbols for them. The chart (*below*) shows the symbols for many different gates and their behavior.

Name	Graphic Symbol	Algebraic Function	Truth Table		
			A	B	F
AND		$F = A \cdot B$ or $F = AB$	0	0	0
			0	1	0
			1	0	0
			1	1	1
OR		$F = A + B$	0	0	0
			0	1	1
			1	0	1
			1	1	1
NOT		$F = \bar{A}$ or $F = A'$	0	1	
			1	0	
NAND		$F = (\overline{AB})$	0	0	1
			0	1	1
			1	0	1
			1	1	0
NOR		$F = \overline{(A + B)}$	0	0	1
			0	1	0
			1	0	0
			1	1	0

4-Basic Digital Designs

We can now take basic logic components and hook them up in useful ways.

For example, let's say we want to create a circuit that tests to see if both inputs are opposites- that is, if A is 1 and B is 0 or if A is 0 and B is 1. This could be useful, as we'll see later. To begin with, we can create a small table (called a truth table) to show the behavior we want.

	A	B	Output
	0	0	0
(a)	0	1	1
(b)	1	0	1
(c)	1	1	0

So to summarize, when **A=0 AND B=1 we output a 1.** Similarly, when **A=1 AND B=0 we output a 1.** We can write that together as the statement:

$$(A=0 \text{ AND } B=1) \text{ OR } (A=1 \text{ AND } B=0)=1$$

It looks like we could almost draw that with just logic symbols of AND and OR if we could figure out how to write A=0 and B=0.

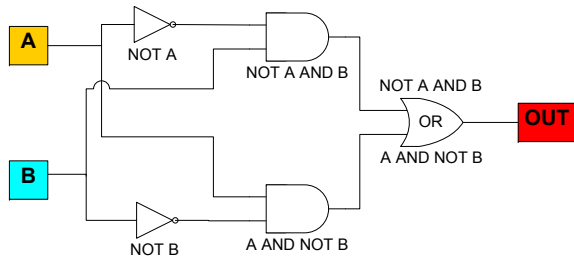
Well, what we're saying is A=0 is true and B=0 is true. "True" just means 1, right? So we are saying A=0 is 1 and B=0 is 1. A=0 outputs a 1 and B=0 outputs a 1.

Looking at our logic symbols on the previous page, we see a logic component that does just that: a **NOT gate**. When A=0 the output is 1. So we can substitute A=0 and B=0 with NOT A and NOT B and it will mean the same thing.

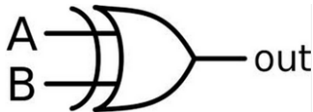
So we can write out our original statement as:

$$(\text{NOT } A \text{ AND } B) \text{ OR } (A \text{ AND } \text{NOT } B)=1$$

If we now use logic symbols instead of letters, we get the following circuit diagram.



This circuit does exactly what we want. It is called an **Exclusive-OR**, or **XOR GATE** since it ORs mutually exclusive inputs.



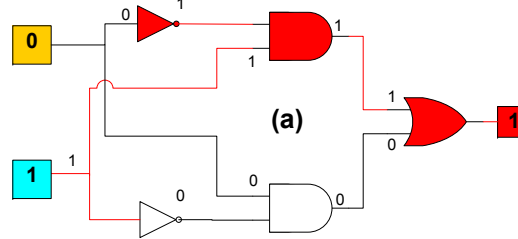
We can see it works by looking at our possible inputs.

Copyright 2013. Ian Ohlander. All Rights Reserved

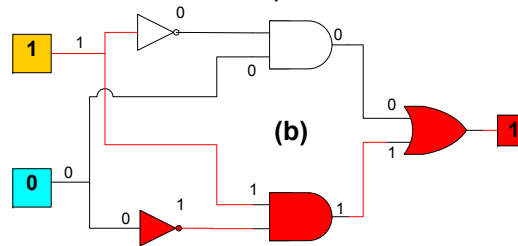
(a) If you set A=0 and B=1, then the top AND gate gets the inputs NOT A (which is 1) and B (which is also 1). 1 AND 1 is 1.

On the other hand the bottom AND gate gets an input of A=0 and NOT B (which is 0). 0 AND 0 is 0.

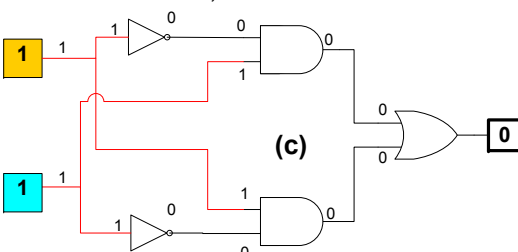
The OR gate thus gets inputs 1 and 0. An OR outputs a 1 if either is 1, so this circuit outputs a 1, or true, for when A=0 and B=1.



(b) The same thing (though opposite) happens when A=1 and B=0. It outputs a 1, or true.



(c) BUT, when A=1 and B=1, the top AND gets NOT A (0) as input and thus will output a 0. The bottom AND gets NOT B (0) as input, also outputting a 0. The OR gate thus takes inputs of 0 and 0. 0 OR 0 is 0, false.



The same thing happens when A=0 and B=0. The output of both AND gates is 0 and so the OR gate outputs 0.

At this point, we notice some interesting functionality of the AND and OR gates.

Note that when one of the inputs on an AND gate is 0, then it doesn't matter what the other one is. The output will be 0. In example (c), both AND gates outputted a 0 because one of the inputs was 0.

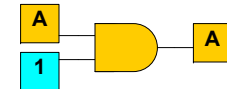
This is such an important thing to keep in mind that we want to write it down (where we use x for AND).

$$A \times 0 = 0$$

That looks pretty familiar. In multiplication, any number multiplied by 0 is 0. In Boolean algebra (the math used to describe digital logic) we use x for AND and that is partially the reason. The other reason is this:

$$A \times 1 = A$$

In Boolean algebra that is also true.



This makes sense. If A=0 then 0 x 1=0. If A=1 then 1 x 1=1. Whatever A is is outputted from the AND gate, provided the other input is 1.

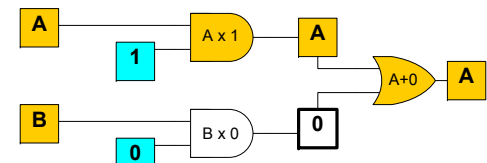
Thus, if we have a couple of AND gates and want to only *activate one of them*, all we need to do is AND that one with a 1 and the others with 0's. This is useful in conjunction with the properties of an OR gate.

Notice that the OR gate acts like a funnel. It will output A OR B. In Boolean algebra, the OR function is represented by +.

$$A + 0 = A$$

If OR has 2 inputs, A and 0, it will output A (as in examples (a) and (b)).

If used with an AND, where one AND gate has been activated and one deactivated, it can be used to *route a signal*.

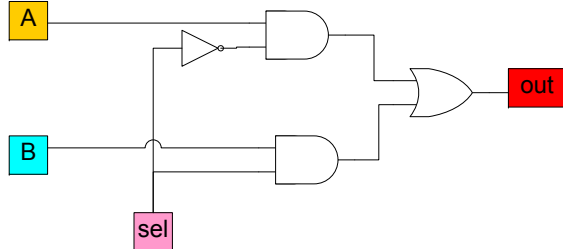


If we put a 1 on the top AND gate and 0 on the bottom then A is outputted.

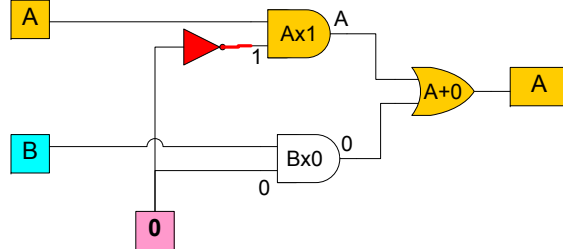
5-Advanced Digital Design

The routing circuit we created on the previous page works pretty well. But you have to remember to manually turn on or off each AND gate to control the routing. It would be nice if we could automatically select whatever gate we wanted without having to manually change each one.

We can do this by introducing a new input, *select (sel)*.

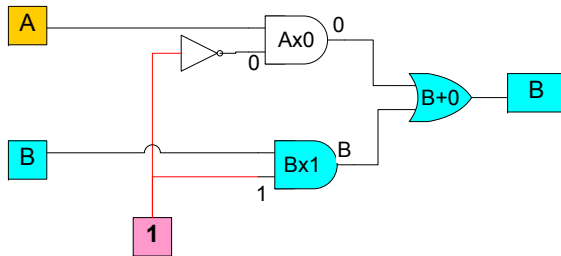


This circuit uses the routing circuit but adds a NOT gate. The NOT gate functions in the same way as the one on the XOR gate.

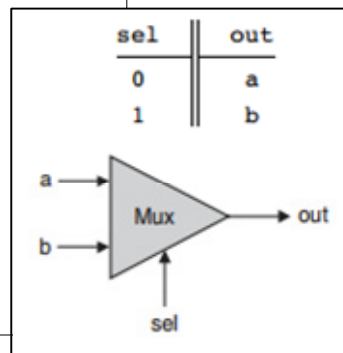


When $select=0$, the top AND gate is activated (by the NOT $select=1$) and the bottom AND gate is deactivated ($Bx0=0$). The OR gate receives $A+0$, which we recall means just A.

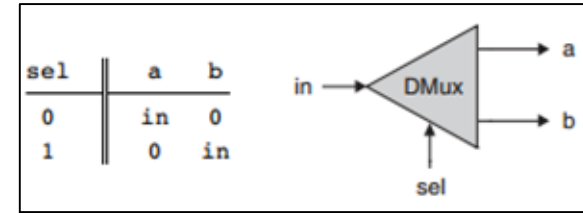
But when $select=1$, then the top AND is set to $A \times 0 (=0)$ and is deactivated. The bottom gate, though is $B \times 1 (=B)$. OR receives as inputs $0+B (=B)$ and thus B is the output.



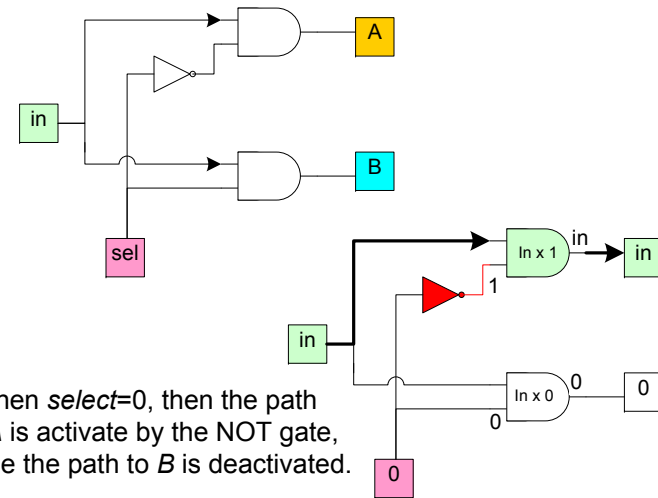
This kind of circuit is called a Multiplexor (Mux) and can be represented by this symbol.



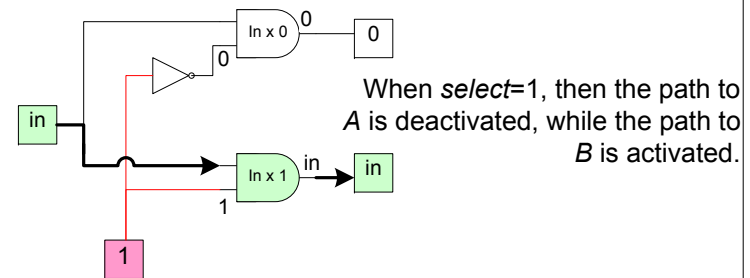
This circuit can be reversed so that a single input can be routed into 2 possible directions. The symbol (and function) of this circuit is:



It seems to work very similarly (though opposite) to the Multiplexor. But actually it is a bit simpler. We don't need to funnel outputs, so we can throw away the OR gate. Instead, we'll have one input and two outputs (a, b) and we'll just use *select* to activate the AND gates for the path out we want.



When $select=0$, then the path to A is activate by the NOT gate, while the path to B is deactivated.



When $select=1$, then the path to A is deactivated, while the path to B is activated.

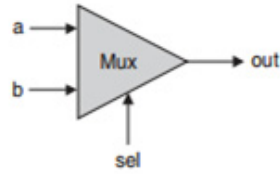
This circuit is called a Demultiplexor (DeMux), since it does the opposite of a Multiplexor (Mux).

Muxes and DeMuxes can be extended to handle more than 2 inputs or outputs. They can also be extended to handle more than 1 bit numbers as inputs or outputs

6-Larger Multiplexors

Copyright 2013. Ian Ohlander. All Rights Reserved

As we saw, Multiplexors take in 2 streams of bits and will output whichever stream is selected, a or b. This is a 2:1 Mux.



We can extend this to 4, 8, 16, etc Muxes by using more than one Mux and another extra selection bits to activate the output.

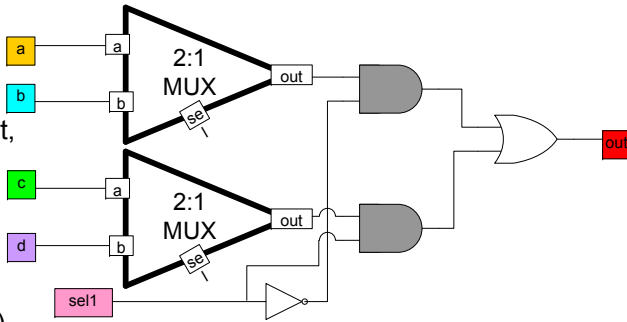
To see how this works, let create a 4:1 MUX. We will need two 2:1 Muxes. Now, we will need to select 4 possible outputs.

sel1	sel0	out
0	0	a
0	1	b
1	0	c
1	1	d

Remember that the select bit only gave us **two** choices: 0 or 1. So for 4 choices, we need 2 numbers (called bits). We can see this works by listing all the choices we get from two bits in the chart. We also add in what we'd like them to select. This chart gives us a clue as to how to make this work.

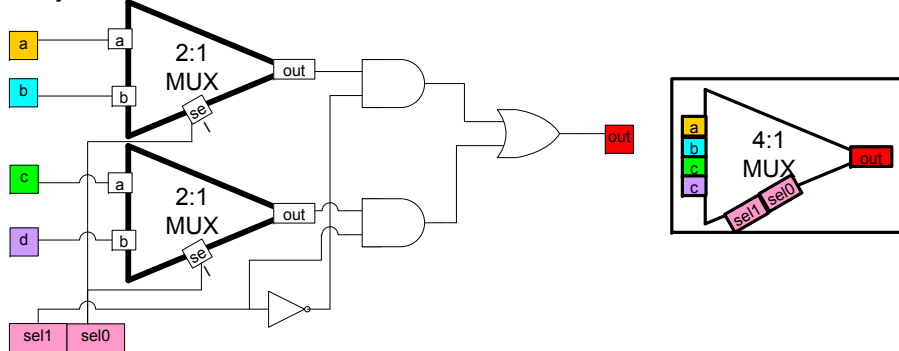
We start by taking our 2:1 Muxes and stacking them.

We then hook up all 4 inputs. Looking at the chart, when *select-bit-1* ($sel1$)=0, the output should be **a** or **b** (from the top Mux). But when $sel1=1$, then the output should be either **c** or **d** (from the bottom Mux).

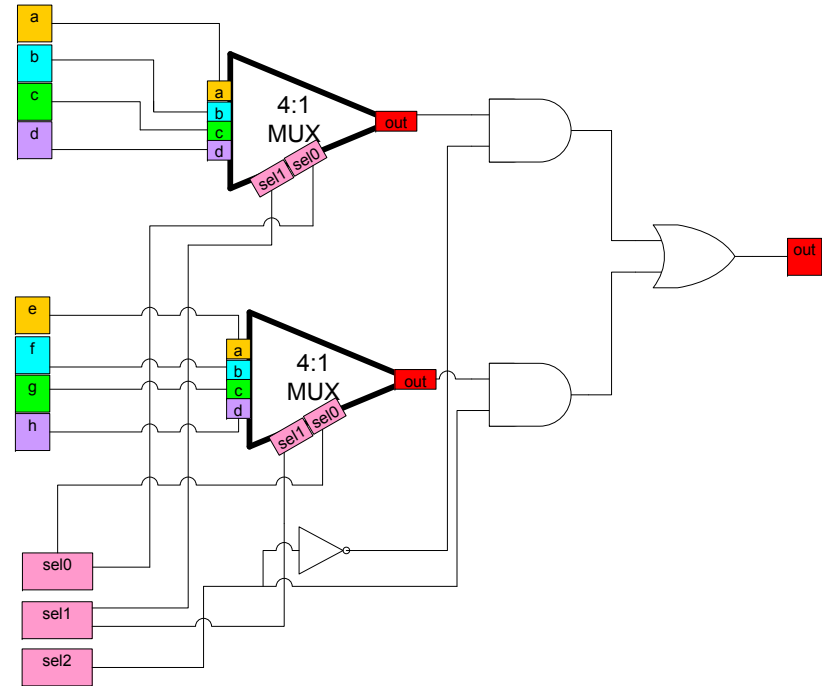


That should ring a bell. If we use two **AND** gates that we can activate depending on $sel1$, and then OR the result, we can select between the top and bottom mux based on $sel1$.

If we look at the chart again, we see that when $sel1=0$, $sel0$ can be used to select between **a** and **b**. Similarly, when $sel1=1$, $sel0$ selects between **c** and **d**. We can depict this 4:1 MUX with this symbol.

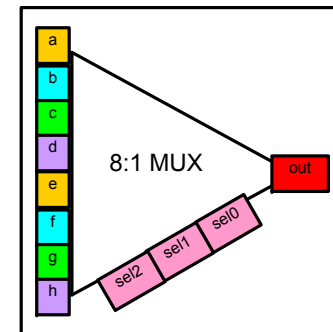


We can use this same method to create an 8:1 MUX. All we need to do is replace the 2:1 Muxes with 4:1 Muxes, connect in 8 inputs (**a-h**), add another select bit ($sel2$) to activate the top or bottom Mux output, and route the remaining 2 selection bits ($sel0$ and $sel1$) into the 4:1 Muxes.



This works exactly the same way as the 4:1 Mux. $sel0$ and $sel1$ select **a-d** from the top Mux and **e-h** from the bottom Mux. $sel2$ then activates output of the top or bottom Mux.

We can represent this 8:1 Mux with this symbol.

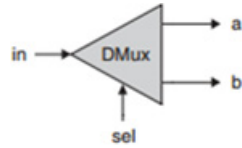


This same technique can be used to make any $2^n:1$ Mux. Just add two base Muxes ($2^{n-1}:1$), connect in all but the first select bit, use the first select bit to activate the outputs of the top or bottom, and then OR the output.

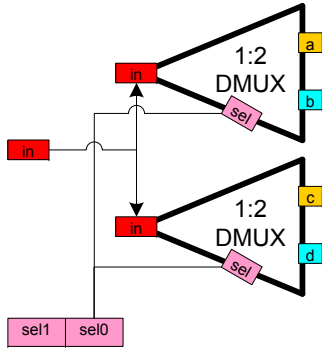
7-Larger Demultiplexors

Copyright 2013. Ian Ohlander. All Rights Reserved

We remember that a Demultiplexor is the opposite of a multiplexor. It takes in 1 signal and will route it 2 possible ways, depending on the *sel* signal. The symbol of it makes this behavior clear.



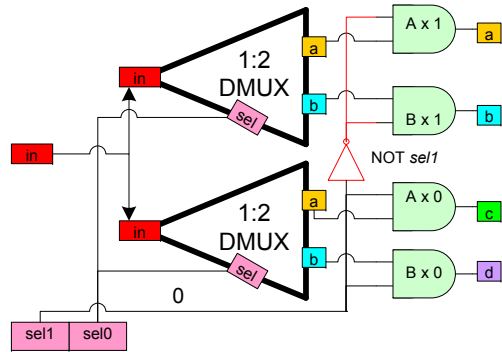
However, sometimes we need to route a signal more than simply 2 possible directions. We can make a 1:4 Demux to handle this in the same way that we made a 4:1 Mux. First we stack our 1:2 Demuxes and connecting in our input.



Since we have 4 possible outputs, we will need 2 select-bits (*sel1*, *sel0*). As we did with the larger muxes, we'll route the *sel0* bits to both demuxes. Thus, *sel0* will allow us to select **a** and **c**, or **b** and **d**.

That just leaves us to figure out how to use *sel1* to select between the top and bottom demuxes. When we made the larger muxes, we used *sel1* to activate the outputs of either the top or the bottom muxes using AND gates. We can do the same thing here, but with a slight difference.

We could put an AND gate on each demux's outputs (**a-d**). For the top 2 ANDs, we'd route in **NOT sel1** while for the bottom ANDs we route in *sel1*. Thus, when *sel1*=0, the top two outputs are activated (and *sel0* will then select between **a** and **b**) (see below). When *sel1*=1, the bottom two AND gates will be activated, with *sel0* then selecting **c** or **d**.



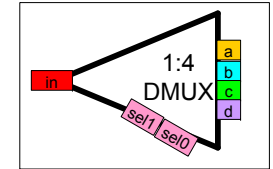
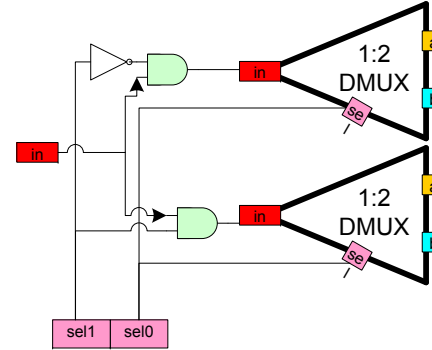
We can see that the top 2 AND gates do the same thing, as do the bottom 2 AND gates. They just activate the output of either the top demux (**a,b**) or the bottom demux (**c,d**). The circuit behaves exactly as the chart says it should.

But while this works, we are using a lot of AND gates, one for each output (**a-d**). A 1:4 demux will use 4 AND gates. A larger demux, like 1:8, will then use 8 AND gates. Extra AND gates mean higher costs, less physical space available, more power consumption and higher heat output. Let's fix this.

<i>sel1</i>	<i>sel0</i>	A	B	C	D
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

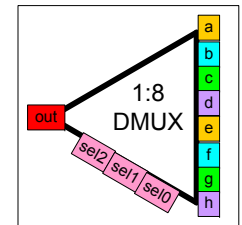
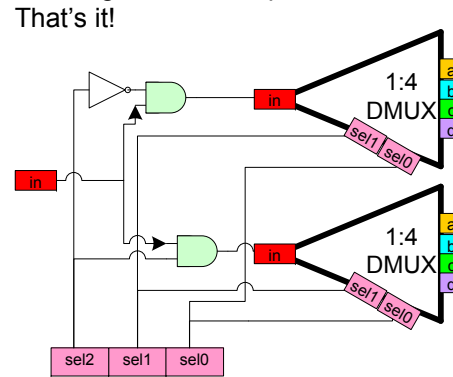
Top DMux (rows 1-2)
Bottom DMux (rows 3-4)

To get around the AND gate problem, we do something else. We can deactivate the top or bottom demux by deactivating the INPUTS instead. They both have a single input. If we put an AND gate there, along with *sel1*, we can control which demux will output the input and which will just output 0's.



Notice that by moving the AND gates to control the **inputs** of the 1:2 demuxes, we still are able to control which demux is activated, but we just cut the number of AND gates in half.

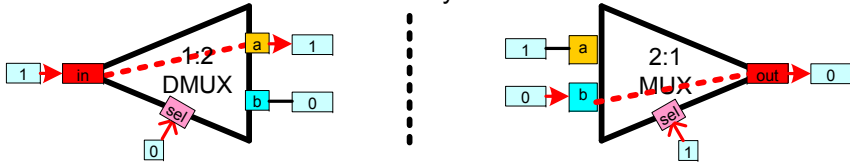
The beauty of this is that we can repeat it exactly for any 1:2ⁿ demux. For example, to create a 1:8 demux (one input going out 8 possible directions) we replace the 1:2 demuxes with 1:4 demuxes. We add another selection bit (*sel2*). We take *sel0* and *sel1* and route them into both 1:4 demuxes and put *sel2* into the AND gates at the inputs of the 1:4 demuxes.



If we had put the AND gates at the outputs of the demuxes, we would have had to use 8 AND gates. This method required only 2.

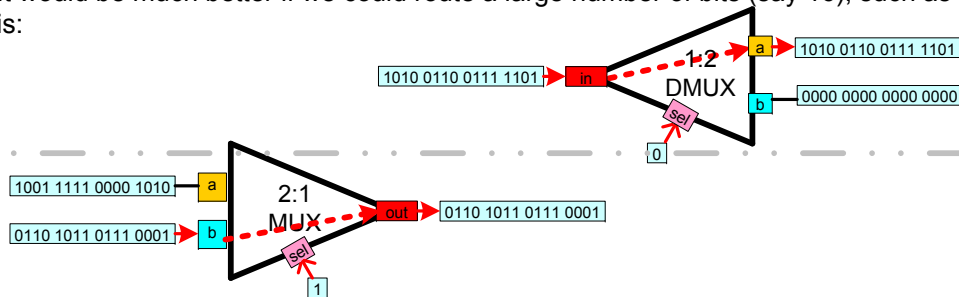
8-Multi-Bit Multiplexers and Demultiplexers

Up to now, we have been able to create multiplexers and demultiplexers that allow us to route bits in different directions. The symbols of them makes this clear.



But one problem we have is that this flow of information, or data, is only 1 bit. The input of a demultiplexer is either a 0 or a 1, while the multi-inputs of a multiplexer are also ultimately one bit that is either 0 or 1.

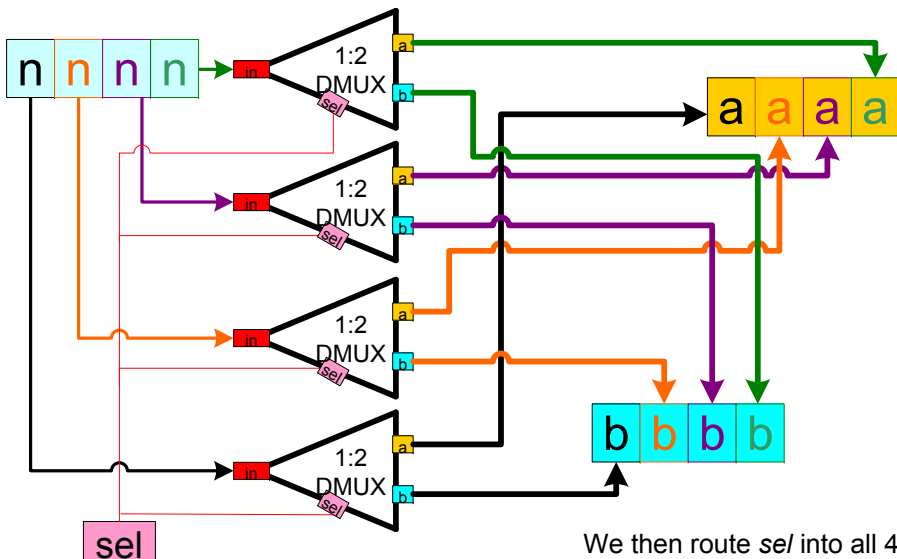
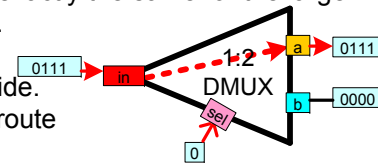
It would be much better if we could route a large number of bits (say 16), such as this:



The good news is that doing this is actually quite easy. We'll do it with 2:1/1:2 Muxes/ Demuxes for data 4 bits wide. But the method used is exactly the same for the larger Muxes/Demuxes and for data any number of bits wide.

Let's say we want to make a 4 bit 1:2 Dmux like this:

Notice that at it's root, our input of 0111 is just 4 bits wide. So that means we'll just need 4 1:2 Dmuxes and then route the appropriate input/output bits into the Dmuxes.

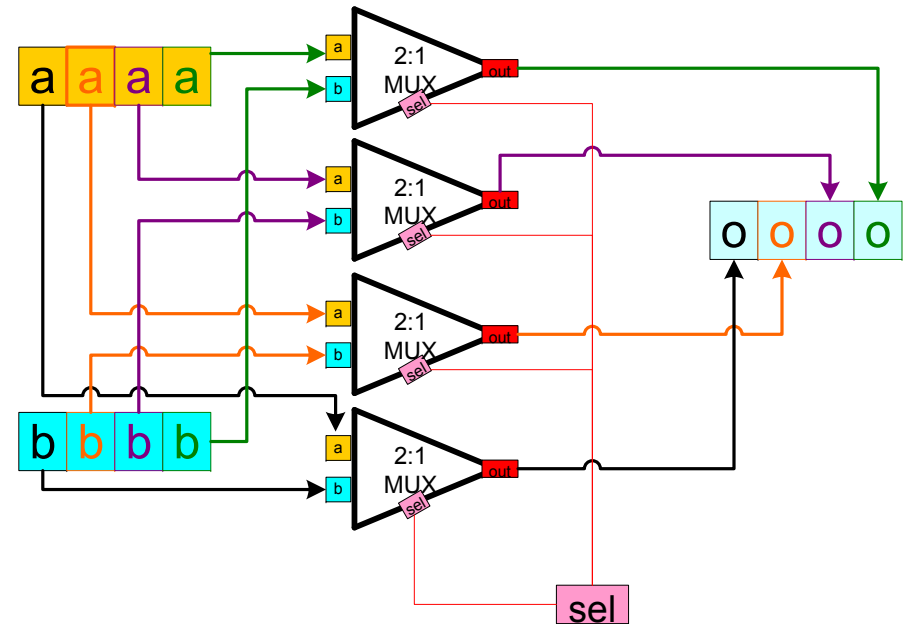


We then route *sel* into all 4 Dmuxes.

That's all there is to it. The 1st Dmux takes care of the 1st input bit, the 2nd takes care of the 2nd input bit, and so on. Output bits for *a* are connected one at a time and the same with *b*. Since we want the *a* or *b* output of each Dmux at the same time, the same *sel* bit is used on all the Dmuxes. Thus, when *sel*=0, the 4 *a*-output bits receive the *n*-input bits. When *sel*=1, the 4 *b*-output bits receive the *n*-input bits.

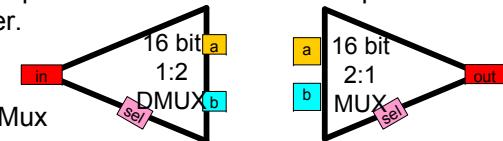
If we wanted to handle more output channels, we just add more Dmuxes and hook it up. And if we wanted this to be *n*-bit wide 1:4 or 1:8 Dmuxes, we would just use 1:4 or 1:8 Dmuxes as the core Dmuxes for each bit, We'd have to set up the output bits (*a-d*, or *a-h*), as well as route the selection bits (*sel-n*) into each Dmux. But all of that is easy enough.

And it works just the same for Muxes.

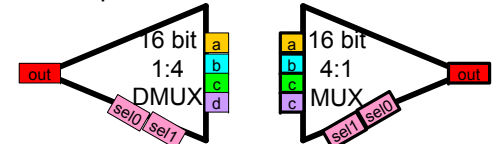


This should be pretty self-explanatory. The inputs on each Mux handles 1 bit. The *selection* bit sends the input out either *a* or *b*. The *outputs* of each Mux is arranged in the correct order.

Here are our 2 examples of 16 bit 1:2 Dmux and 2:1 Mux



Just as with Demuxes, to make an *n*-bit 4:1 or 8:1 Mux, we use one 4:1 or 8:1 Mux for each individual bit. We connect up the *selection* bits to each Mux, and then we capture the output.



Again, here are examples of a 16 bit 1:4 Dmux and 4:1 Mux

9-Decimal and Binary Numbers

Usually, the word binary scares people off. But it is just another number system—a way to represent numbers. Now it may surprise people when they hear there are different ways to represent numbers, so we're going to look at a simple example.

"We will meet up at 1400 hours."

That may seem familiar. We call it military time. It means "2pm". Why are there 2 different ways to say 2pm? Because our day lasts 24 hours. In normal everyday use, we use a base-12 number system for time. A day begins at midnight and, every 60 minutes, we increment the hour. 10am. 11am. 12pm, But then right after 12:59:59pm, we restart our count as 1:00:00pm. So the number system refers to how many digits it will allow before it "rolls over," in this case 12, and then the count restarts. But military time uses a 24 hour a day number system. So once 12:59:59 pm hits, the hour count continues to increment, becoming 13:00:00 ("13 hundred hours"). No AM or PM. Just a continual increase in hours until 23:59:59, at which time the count finally "rolls over" or restarts at 00:00:00 ("zero-hundred hours").

We don't go to 10 (or 100) before restarting the count. We go to 12 or 24. With minutes and seconds, we go to 60. We have 60 seconds=1 minute and 60 minutes=1 hour. We may remember a quarter was \$.25. But when counting time, a quarter hour was 15 minutes.

We don't have 10 or 100 day months. We have 28-31 day months, making 12 months for the entire year, at which time the count of days (it's the 31st day of the month) and the count of months (it's the 12th month, December) restarts, while the year (2013) increments. We do the same with conversions between *ounces*, *quarts*, and *gallons*, or *inches*, *feet*, *yards*, and *miles*.

The point is, we using different number systems all the time, and though conversion can seem annoying, we are used to them. We know that it's 12 inches to a foot and 5280 feet is 1 mile and 4 quarts to a gallon.

All these number systems of 10, 12, 24, 30, 360 and so on, are remnants of different cultures and how they counted. We get our time counting from the Babylonians and our linear measurement from the British.

But except for special counting, like time or measurement, in general most of the world uses a base-10 number system. Remember that refers to how many numbers you count before you "roll over" and restart. That means that we count 10 digits before we increment the next position.

We have only the digits 0-9 and their position, which we can use to represent any number.

For example: **1,679,935,500.00049**

BILLIONS			MILLIONS			THOUSANDS			ONES	DECIMALS							
hundred billions	ten billions	billions	hundred millions	ten millions	millions	hundred thousands	ten thousands	thousands	hundreds	tens	ones	tenths	hundredths	thousandths	ten thousandths	hundred thousandths	millionths
		1	6	7	9	9	3	5	5	0	0	.	0	0	0	4	9

So if we say 233, we mean:

two-hundred thirty-three

Writing that out more clearly, we are saying:

$$(2 \times 100) + (3 \times 10) + (3 \times 1)$$

Each position stands for 1, 10, 100, 1000, etc. While we don't consciously do this anymore, it's how we learned it in school. For example, this (*right*) comes out of 2nd grade math book.

Hundreds 2	Tens 3	Ones 3

Because values of each position (tens, hundreds, etc) are powers of 10, we say that it is a base-10 number system. We can make this clearer by writing this:

$$(2 \times 10^2) + (3 \times 10^1) + (3 \times 10^0)$$

Again, while that looks complex, it is just writing down mathematically what you do in your head all the time. We only have 10 digits (0-9), at which point we "roll over" to 0, while incrementing the next largest column. Look at what happens in the **one's column** when we add and exceed 9.

	10	1	
	2	9	
+		1	
	2+0	9+1	
	2-1	0	= 30

But while 10 seems special to us because we have 10 fingers and 10 toes and thus seems natural, there is really nothing unique about it. As we considered earlier, other cultures throughout history have used (and even we use) different number systems.

As we have seen, digital logic relies on just 2 possible binary digits (or bits): 0 and 1. So instead of going all the way up to 9 before "rolling over" to the next position value, binary numbers only go from 0 to 1 before rolling up. So how do we represent numbers with just 2 bits?

...	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
	16	8	4	2	1

The same way we did with 10. Each column will represent a power of 2. 0 or 1 will go into each column, representing how much of that power of 2.

2 ³	2 ²	2 ¹	2 ⁰
8	4	2	1
1	1	0	1

For example, **1101** in binary means:

Which we can break down to

$$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) =$$

$$(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) =$$

$$8+4+0+1 =$$

$$13$$

Any decimal (base-10) number can be represented as a binary (base-2) number, just as 1 foot is also be 12 inches. The number, or distance, is the same. **What we call it is the only difference.**

10-Decimal and Binary Addition Copyright 2013. Ian Ohlander. All Rights Reserved

Computers compute. They do all kinds of math. But at their core, they are simple *adding* machines. **Every other mathematical function they do derives from simple addition.** Multiplication, division, exponentiation, integration, linear algebra, vector calculus- for a computer, it all comes down to a hardware component that can add and software using complicated algorithms, or recipes.

Our next step then will be to build an adder. Since computers use binary numbers, we need to add in binary. To do this, let's review how we add using our normal decimal numbers.

Since each column in our numbers simply represents a power of ten- 1, 10, 100, 1000, etc)- we just add up each column. In this example, we are adding 4+8. Both the 4 and 8 are in the one's column. Their sum is 12. But the **1** in **12** is actually 10, so we need to put that **1** in the ten's column (we call that a carry). The ten's column already has a 0+0, so the sum in the ten's column is actually **1+0+0**. The total sum becomes 12.

10s	1s
0	4
+0	8
1	2

That seems like it was made unnecessarily complicated, but the fact is, that is what we learned in 2nd grade. We just do it automatically now. Our point, though, is to look at the actual process, because we do the exact same thing in binary addition. We add each column and if there is a carry we add it to the next column.

Here we add 1+0. The 1 in the one's column adds to the 0, leaving a 1 in the one's column. We had zeroes in the two's column, and 0+0=0. But that's not very useful. Let's add 1+1. So that means we have two 1's in the one's column. Remember, in binary, we don't have a digit for 2 (just as above, we didn't have a digit for 12.) So that means we put a 0 in the one's column and *carry* a **1** into the next column, the **two's column**. That makes sense though. *We are saying 1+1=2*. And we just put a **1** in the **two's column** and a 0 on the **one's column**, meaning one 2 and 0 ones:

2s	1s
0	1
+0	0
0	1

2s	1s
0	1
+0	1
1	0

$$(1 \times 2) + (0 \times 1) = 2 + 0 = 2.$$

It's just the same as decimal addition.

So far we have considered **1+0=1** and **1+1=10** (binary 2). 0+1 is exactly the same as 1+0, so that also is 1. And we know that 0+0=0. So we have: **0+0=0**, **1+0=1**, **0+1=0**, and **1+1=10**. Looks like we have covered all possibilities for adding in a single column.

We'll list them all here at *right*. We'll add **zeros** to show there was no carrying for almost all adding.

0	0	0	1
+0	+0	+1	+1
0	0	1	0

A + B	CARRY	SUM
0 + 0	0	0
0 + 1	0	1
1 + 0	0	1
1 + 1	1	0

Let's rearrange above into the chart *left*. We'll turn it sideways so it looks familiar. For the answer let's call the column on the right (the **one's column**) **Sum**. We'll call the column on the left (the **two's column**) **Carry**. And let's label our two digits we are adding as **A** and **B**.

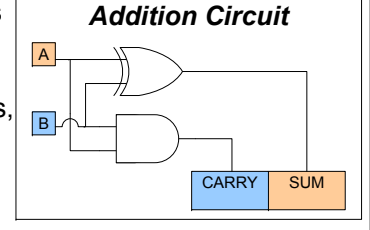
So, with our **A** and **B** inputs, we have two outputs: **Carry** and **Sum**. This looks like a truth table for some logic gates. Let's look at the **Carry** output first. Notice that it is *only* 1 when *both* **A** and **B** are 1. If we write that again, we can say **Carry**=1 when **A**=1 AND **B**=1. That **AND** jumps out at us. In binary addition, a **Carry** output is just an AND gate!

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

What about **Sum**? We notice that it is only 1 when either **A**=0 and **B**=1 OR **A**=1 and **B**=0. That may not look familiar, but we built a circuit a while ago that did that. It is called an **XOR** (Exclusive-OR gate).

If we compare the truth tables of **AND** and **XOR** next to **Carry** and **Sum** we see they are the same. So what does this mean? We can create a circuit that adds! The circuit doesn't know what it is doing. It is just performing logical operations. But **we** interpret its results as addition.



That's it! This circuit (right) adds two 1 bit numbers, **A** and **B**, and gives us the **Carry** and **Sum**.

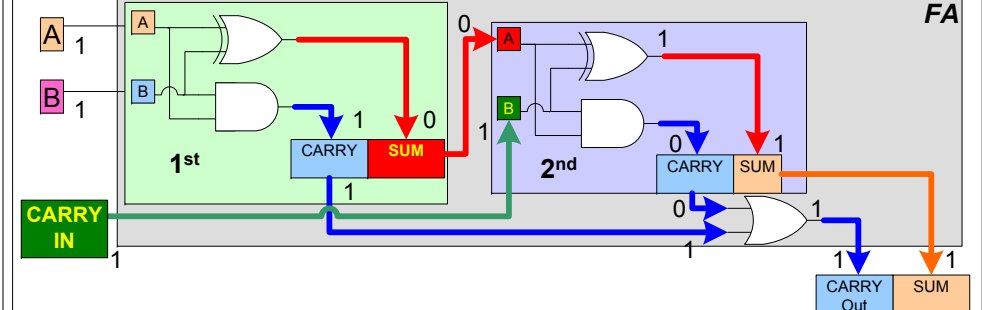
But we want to be able to add more than 1 bit numbers. So how do we add larger numbers like this?

Most of this would follow the rules we already know (0+1, 1+1, etc). But in the highlighted column, we see a **1+1+1**. There was already a carry in of 1 from the previous column and now the sum 1+1 made another. We know 1+1+1 = 3. How do we represent 3 in binary?

Well, 3 = 2+1, right? So we put a 1 in the **two's column** and a 1 in the **one's column**. Binary 3 is 11 (*left*).

Easy enough then. That's exactly how we deal with that kind of carry. We put a 1 in the column we are in, and then carry a 1 into the next column.

So how does that translate into an addition circuit? Well notice that we are actually doing 2 additions. We first add **A** and **B**. Then we take that sum and add it to our carry from the previous column. 2 additions means 2 of our adders. We'll call them 1st and 2nd. **A+B** will go into 1st. 1st's Sum will go into 2nd's **A** input. Our **Carry-in** from the previous column will go into 2nd's **B** input. 2nd's sum will be our total sum. If 1st generates a **Carry-out** OR 2nd generates one, then we have a **Carry-out**.



Logically that makes sense. Let's use our example problem of $1_{[Carry-in]} + 1_{[A]} + 1_{[B]}$. We first add $1_{[A]} + 1_{[B]}$. **Carry-out**_[1st]=1 and **Sum**_[1st]=0. $A_{[2nd]} = \text{Sum}_{[1st]} = 0$ and $B_{[2nd]} = \text{Carry-in} = 1$. **Carry-out**_[2nd]=0 and **Sum**_[2nd]=1.

$$\text{Carry-out}_{[FA]} = 1_{[Carry-in]} + 0_{[Carry-out 1st]} + 0_{[Carry-out 2nd]} = 1. \quad \text{Sum}_{[FA]} = 1_{[2nd]}$$

$$1_{[Carry-in]} + 1_{[A]} + 1_{[B]} = 11$$

It works. To add larger numbers, we can treat each Full Adder (FA) like a column. The 1st column will have a **Carry-in** of 0. The **Carry-out** of each FA will go into the **Carry-in** of the next FA.

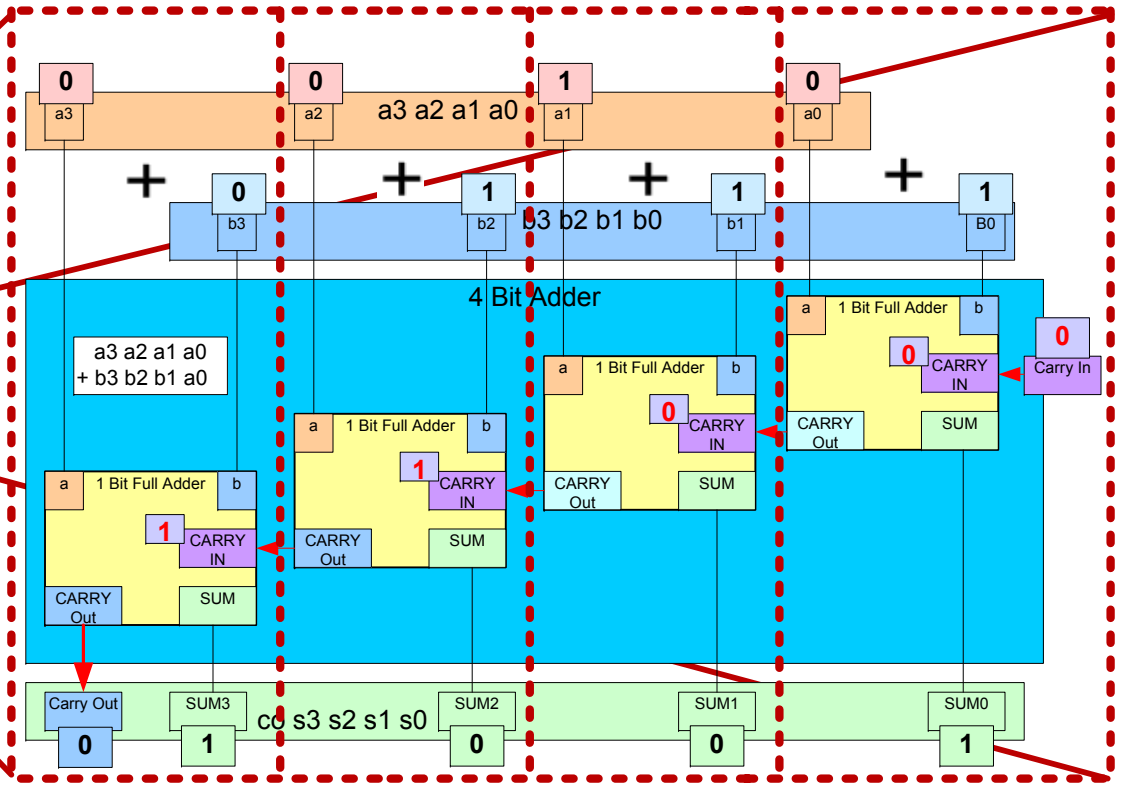
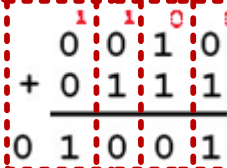
11-Advanced Adder Logic

We built a 1 bit adder that allows us to add two numbers, **A** and **B** along with a **Carry-In**. Now, in order to add larger numbers, we just create an array of these 1 bit full adders, one for each bit of the numbers we are adding. We can view each adder as an addition column that then passes any **Carry-Outs** on to the next column. The example addition of $0010 + 0111$ ($2+7 = 9$) can help us see this.

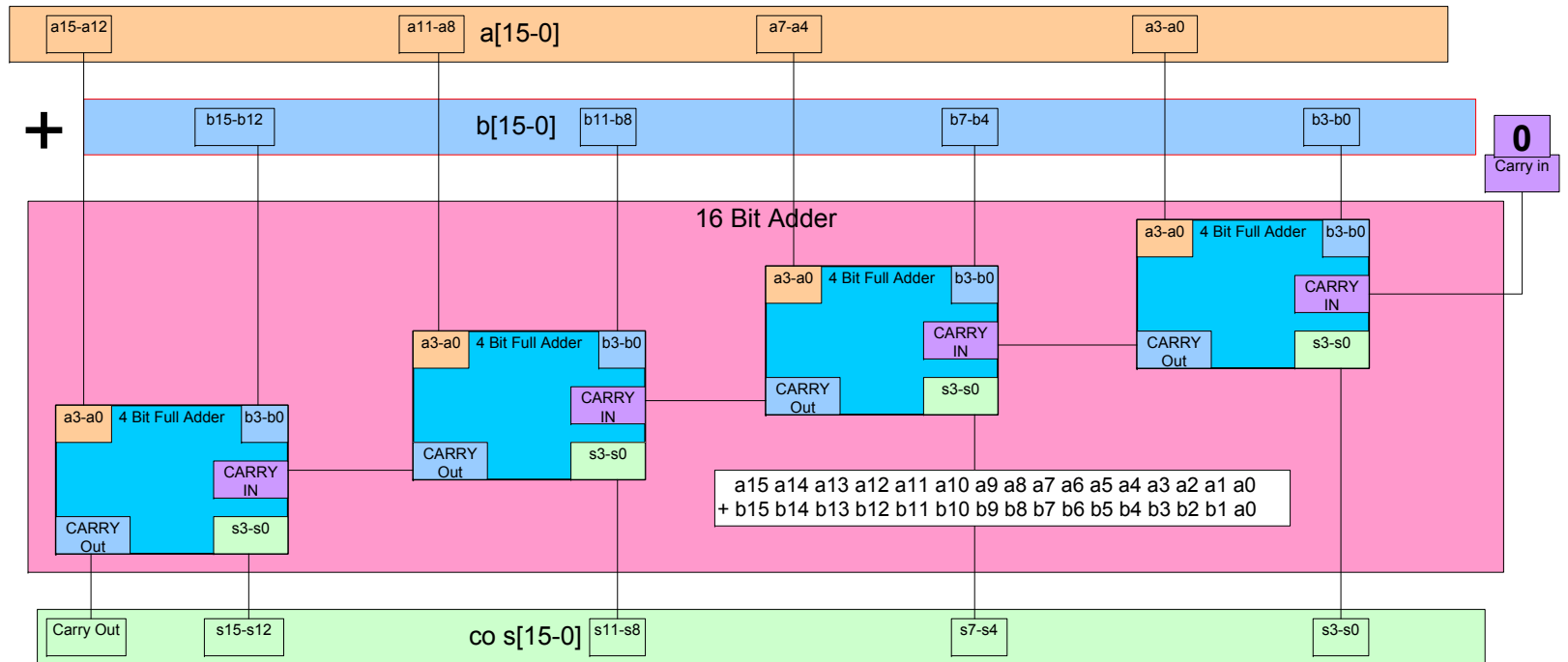
The adder for the one's column (furthest right) is the first one, so the **Carry-In** is set to 0. This makes sense since addition of the first digits in the one's column (whether Decimal or Binary) don't have anything to carry in.

Then, that adder's **Carry-Out** is routed into the next adder's **Carry-In**. We continue for as many columns we want to be able to add. Each adder gets 1 bit from the two numbers, **A** and **B**, we want to add, as well as the **Carry-In** from the previous adder (except for the first one, as we said.)

We can do this for 4 columns and make a 4-Bit Adder. We then can repeat the process with four 4-Bit adders, and make a 16-bit Adder. If we repeat that process, we can make Adders for any width of numbers we want to add. We just have to remember to put 0 in the first column's **Carry-In**.



Copyright 2013. Ian Ohlander. All Rights Reserved



12-Arithmetic Logic Unit- Part 1

Copyright 2013. Ian Ohlander. All Rights Reserved

So computers compute. We already said that. And an adder sits at the core of that. But it doesn't work alone. We've worked with AND gates a lot. And NOT gates. We can make 16-bit versions of these. A 16-bit AND gate will AND two 16-bit numbers together and output the result. A 16-bit NOT gate will NOT each bit of a 16-bit number and output the result.

```
0001 1011 0100 0110
1001 0010 1101 0101
0001 0010 0100 0100
```

(a) 16-bit AND

```
1101 1001 1101 1010
0010 0110 0010 0101
```

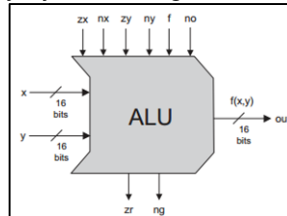
(b) 16-bit NOT

It may be hard to believe, but a 16-bit Adder, a 16-bit ANDer, and some 16-bit NOT gates, 16-bit 2:1 multiplexors and 16-bit 1:2 demultiplexors are all that is needed to create the core component of a CPU. The CPU is the Central Processing Unit of a computer, its brain. It is the unit responsible for executing each line of computer code.

Everything we do with computers (and that includes cell phones)- surfing the internet, watching YouTube videos, texting, Facebooking, editing pictures, swiping our fingers on the screen, moving the mouse pointer, listening to music- all occurs at it's deepest level as computer code. Computer code, or machine language, basically tells the CPU exactly what to do as well as where to put the results or data. That's it. Of course, there are layers and layers of code "objects" that make our sending an email or clicking a link seem so simple. But at its root, the only thing a computer ever does is follow basic instructions of *simple arithmetic* and *moving data around*.

And we have already created the pieces necessary to do the first part of that: simple arithmetic. Of course, it's not as simple as that. We want to create a component that will take in 2 numbers (**x** and **y**) and, depending on the instructions we give it, will add, subtract, AND and so on. So it will have to be able to adjust its output based on what we tell it we want. Surprisingly, though, we can do this fairly easily by controlling the flow of **x** and **y** using demultiplexors and multiplexors.

We will call this component an *Arithmetic Logic Unit (ALU)*. It will allow us to input two 16-bit numbers (**x**, **y**) and six control bits. It will output the result (**out**), as well as two flags indicating whether **out** was zero (**zr**) or a negative number (**ng**). In the end, we want it to be a component that looks like the picture at the right.



The control bits instruct the ALU to perform 3 simple operations: add, AND, NOT. But when done in various combinations, the resulting **output function** can be much more complex. The control bits are:

zx zero x	nx NOT x	zy zero y	ny NOT y	f AND/add	no NOT output
--------------	-------------	--------------	-------------	--------------	------------------

These control bits are fairly explanatory. **zx** means zero x. When **zx**=1, **x** is set to 0000 0000 0000 0000, no matter what it initially was. **nx** means NOT **x**. This occurs when **nx**=1. Looking at the example (b) above, you can see that if **x** was 1101 1001 1101 1010 then NOTing it would turn it into 0010 0110 0010 0101. **zy** and **ny** do the same to **y** when they are 1. **f** means function. When **f**=0, we AND **x** and **y** (Example (a) shows what that looks like). When **f**=1 we add **x** and **y**. **no** means NOT **out**, just like **nx** and **ny** and only happens when **no**=1. When set to the control bits to 1 in various combinations, they instruct the ALU to perform many functions.

zx zero x	nx NOT x	zy zero y	ny NOT y	f AND/add	no NOT output	f(x, y)= output function
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	NOT x
1	1	0	0	0	1	NOT y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x AND y
0	1	0	1	0	1	x OR y

It may seem strange that little things like turning **x** or **y** into zeroes or NOTing them, coupled with operations like ANDing or adding them can let us perform all these functions. So let's look at just a couple of examples.

Let's say we want to add **x** and **y**. Looking at the appropriate row in the chart tells us that we need to set the control bits to the following.

zx zero x	nx NOT x	zy zero y	ny NOT y	f AND/add	no NOT output
0	0	0	0	1	0

That's pretty easy. We don't do anything to **x** or **y** or the output. The only control bit we set to 1 is **f**, which means we add.

Let's try subtraction. Let's assume that **x**=0000 0000 0000 1011 (11) and **y**=0000 0000 0000 0010 (2). We want the **output function** to be **x-y**. **out** should result in 0000 0000 0000 0101 (9), which is 11-2. The chart tells us the bits are:

zx zero x	nx NOT x	zy zero y	ny NOT y	f AND/add	no NOT output
0	1	0	0	1	1

We'll follow the step for each control bit.

nx =1	NOT x	Not x = 1111 1111 1111 0100
f = 1	out =NOT x + y	out = 1111 1111 1111 0100 + 0000 0000 0000 0010 1111 1111 1111 0110
no =1	NOT out	Not out =0000 0000 0000 1001

Notice that **out**=0000 0000 0000 1001 (9), which was we wanted. But why does that work? Why do the other control bit settings result in all those **output functions**?

It has to do with Boolean Algebra, the math that governs the logical operations AND, OR, and NOT. These logical operations can be manipulated in a similar way to regular algebra. So after we decide on what function we want (like **x-y**) we can algebraically work backward from that to figure out what our initial operations (AND, NOT, add) for **x,y**, **out** need to be. That's already done and is in the chart.

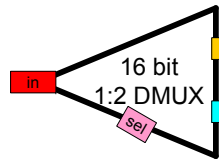
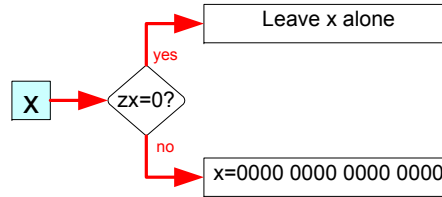
13-Arithmetic Logic Unit- Part 2

Now that we understand the behavior we want our ALU to support, let's figure out how to use the components we've already created to make that happen. Here are our control bits again, for reference.

zx zero x	nx NOT x	zy zero y	ny NOT y	f AND/add	no NOT output
---------------------	--------------------	---------------------	--------------------	---------------------	-------------------------

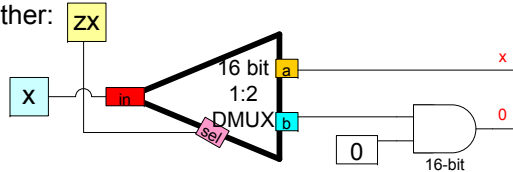
Let's just take the first control bit and figure out what that means. When **zx=0**, we need to leave **x** alone. It needs to stay the same. But when **zx=1**, we want to turn whatever **x** was into 0.

We can draw that out to help us visualize it better. The diagram makes this pretty clear. But it also looks sort of familiar, conceptually. We have an incoming line, **x**, and depending on the value of **zx**, we have two things happen to **x**. Either it is left alone or it is modified.



A 16-bit Demultiplexer happens to do this. We have 1 input (**in**), 1 selection bit (**sel**), and two output channels (**a,b**). We can put **x** into **in**, **zx** into **sel**. Output **a** can be the channel out to where we leave **x** alone. Output **b** can then be the channel out where we turn **x** into 0000 0000 0000 0000.

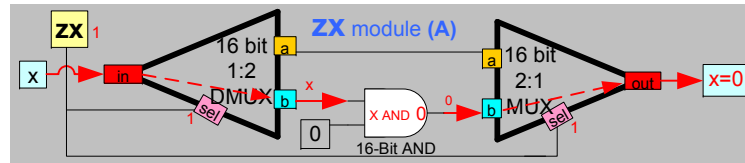
In order to turn **x** into all zeroes, we can use an AND gate. Recall that, if you put a 0 into an AND gate, then it will output 0 no matter what its other input is. Putting all this together:



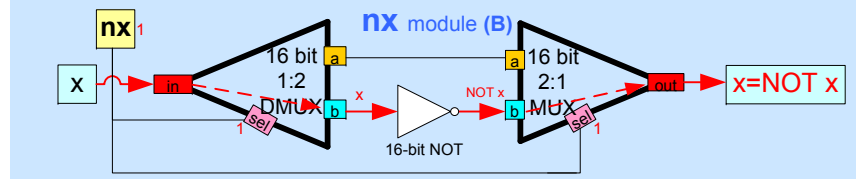
This works. When **zx=0**, the **x** input goes out output **a**. When **zx=1**, **x** input goes out channel **b**, where it is then ANDed with 0. **x** thus becomes 0.

Note that now, though, we have 2 **x** channels (**x,0**). That is a problem. There are still 5 more control bits left. Two of them (**nx, f**), still need to act on **x**. It doesn't make sense to try to have to **NOT x** and **AND/add** on two **x** channels, especially since only one of those channels is active at any given time. So it would be nice to select just the line that we want and discard the other.

We can do this with a 16-bit Multiplexer. We will reuse **zx**, this time to allow out only the channel we want (**x** unchanged, or **x=0**). We'll set **zx = 1** to see it work.

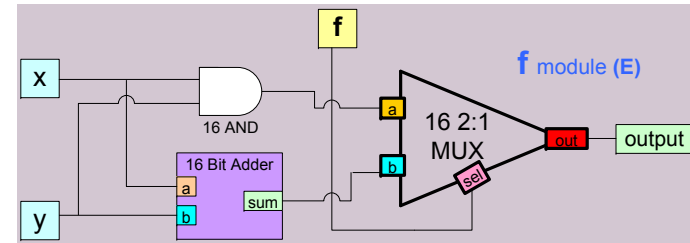


We can do exactly the same for the control bit **nx** (NOT x), using a 16-bit NOT gate.



At one point, we are going to be connecting those modules (**A** and **B**) together. The **x**-output of **A** will go into the **x**-input of **B**. And as you may guess, we can implement **zy** and **ny** in exactly the same way. (The diagrams are literally the same, except that wherever the **x** appears, replace it with a **y**. We will call them modules **C** and **D**.)

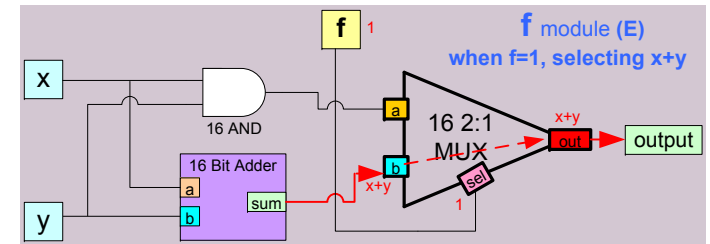
So let's look at what we need to do to implement the **f** control bit function. When **f=0**, we want **x AND y** and when **f=1**, we want **x + y**. So we know we will need a 16-bit AND gate as well as a 16-bit Adder. Both of these need to take **x** and **y** as inputs. Since both of them are taking **x** and **y** as inputs, all **f** needs to control is whether to pass through the output of the 16-AND gate or the 16-Bit Adder. We can do that with a 16-bit Multiplexor and **f** into **sel**.



As we can see, **f** selects between the outputs of the 16-Bit AND gate and the 16-bit Adder.

Below, we see an example when **f=1**.

Finally, **no** will take **output** and leave it alone if it is 0. But if **no=1** it will select a Demux-Multiplexor combination module (**F**) (exactly the same as modules **A-D**, so we will not diagram it) to NOT **output**.



There are two other things we need from this ALU. We need 2 *Status Flags*. One flag, **ng**, will tell us **if output is negative**. We don't need to go into an extended discussion about how negative numbers are represented in binary. The explanation is rather involved. Instead, we will just accept that the **Most Significant Bit (MSB)** of a binary number is the sign. The **MSB** is the bit furthest left of a binary number.

For example, in the number **1000**, the **MSB** is in the 4th column from the right, and is **1**. When **MSB=1**, the number is negative. When **MSB=0**, the number is positive. So we just set **ng=MSB** and we are done the negative indicator flag.

The other flag, **zr**, will tell us **if output is zero**. We can do this by taking all the bits of **output** and put them into a NOR gate. A NOR gate only outputs a 1 when all it's inputs are 0. If **output=0**, then **NOR output=1**. We can set **zr=NOR output**.

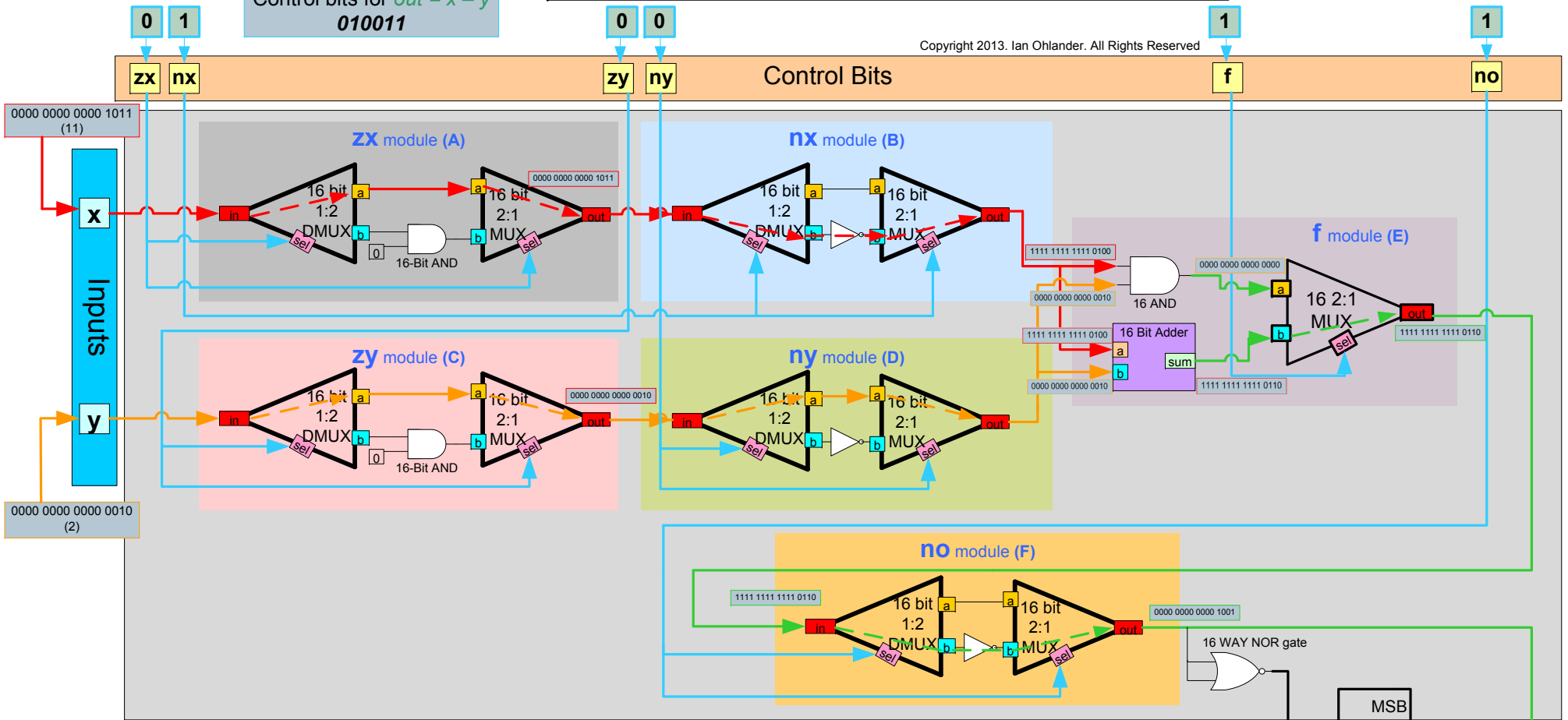
That's it! We now have an ALU that will perform all the logical operations specified by the control bits in the chart. And as we previously saw, various combinations of these control bits will perform numerous functions.

Now, we just put all our modules together.

14-Arithmetic Logic Unit (ALU)

Copyright 2013. Ian Ohlander. All Rights Reserved

Control bits for $out = x - y$
010011



Here is our completed ALU. (Let's ignore the arrows, as well as boxes edged in red, orange, blue and green.) We've taken all the modules we created (A-F) and put them together. We first link up our x -input modules (A output to B input) as well as our y -input modules (C output to D input). We then take both the x and y outs of B and D and route them into the function module E. E has both a 16-bit ANDer and a 16-Bit Adder, so x and y both go into them. Finally, E output goes into module F. Each module takes in a single control bit (zx, nx, zy, ny, f, no) that selects which path thru the module to allow.

We also then look at the NOR of all 16 bits of the output to see if it is zero and set the zr flag accordingly, as well as the **MSB** of the output to the ng flag.

Let's look back at our **example problem** we did earlier

$$x = 0000\ 0000\ 0000\ 1011\ (11)$$

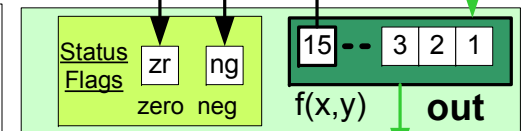
$$y = 0000\ 0000\ 0000\ 0010\ (2)$$

zx	nx	zy	ny	f	no
zero x	NOT x	zero y	NOT y	AND/add	NOT output
0	1	0	0	1	1

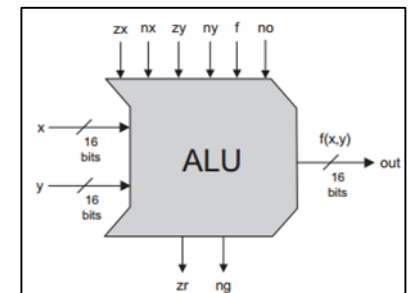
We set our **control bits**, based on our chart, to allow for $out = x - y$: 010011

We can follow the x and y inputs as they pass through each module using the arrows. The output of each module is also shown in color edged boxes.

Our output comes out as: 0000 0000 0000 1001 (9). $11 - 2 = 9$, so our ALU worked perfectly. We can do the same with any of the operations in our chart.

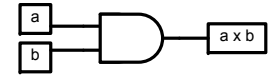


0000 0000 0000 1001 (9)

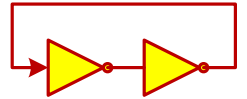


15-Memory

A computer needs to remember information. It needs to allow for retrieval and storage of that information. Doing this with the usual gates (AND, OR, etc) doesn't seem feasible. Up to now, a simple circuit takes in one or more inputs and, based on the way things are connected together and the rules of logic that govern each gate, outputs an answer. Look at the example of an AND gate (right). We have two inputs. Change the inputs and the output (A AND B) changes instantly. There is *no memory of any previous A AND B computations*.



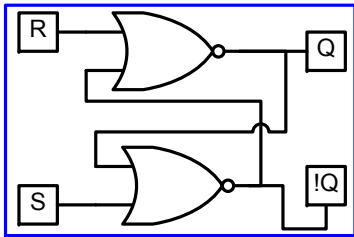
But we can get creative and try connecting an output back into an input. We can connect two NOT gates together - which amounts to not changing the input. NOT (NOT X) = X - and then loop the output of the last one back into the first one.



It doesn't seem like it does much. Sure, it "remembers" what it was holding. But there's no way to get any information into it. It's a start though. By looping back an output to an input we can get some kind of memory behavior.

Based on that concept, let's look at this circuit (left). This is similar to the loopback with NOT gates, but with some differences. We have 2 inputs, instead of 1: **S**(et) and **R**(eset). Each NOR gate outputs a value: **Q** and **NOT Q (!Q)**. We are using NOR gates because they have very specific behaviors. They only output a 1 when both $a=b=0$ (the opposite of an OR gate). To see how this helps us, we need to figure out what it does. Let's look at how this circuit behaves both when $Q=0$ (white) and $Q=1$ (red). To differentiate the NOR gates, we'll refer to the top as **NOR1** and the bottom as **NOR2**. To help us with this, we are going to be using our NOR truth table (below).

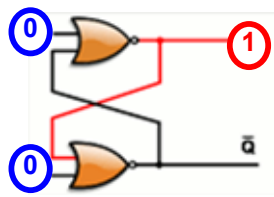
The output that we want to focus on is **Q**. **Q** is the *data-holding bit*.



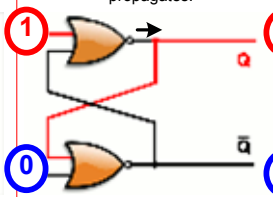
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

NOR Truth Table

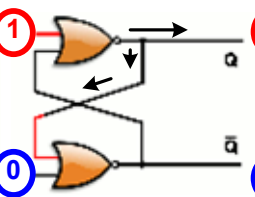
Right at this moment, $Q=1$. With no input on R or S, that doesn't change. Q holds a 1.



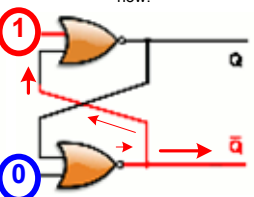
Notice we set $R=1$. Immediately, the change propagates.



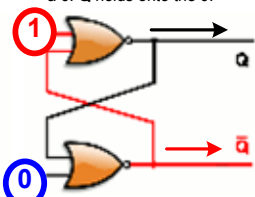
NOR1 outputs a 0, changing Q to 0. Q now holds 0.



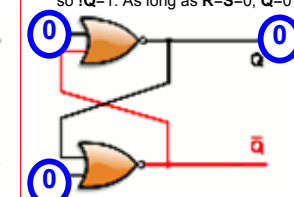
NOR2 has a 0's as input, so it outputs a 1. !Q holds a 1 now.



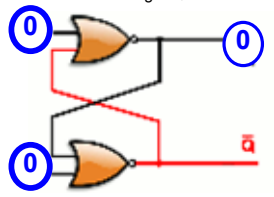
NOR1 now has 1's for its inputs. It continues to output a 0. Q holds onto the 0.



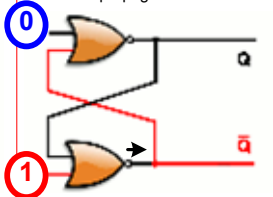
When $R=0$, NOR1's inputs are 0 and 1, so it continues to output 0. Q holds onto 0. NOR2's inputs are 0's so !Q=1. As long as $R=S=0$, $Q=0$



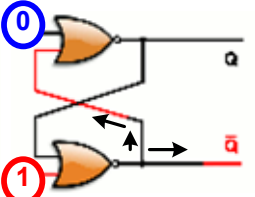
Right at this moment, $Q=0$. With no input on R or S, that doesn't change. Q holds a 0.



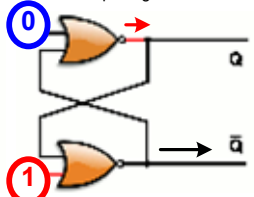
Notice we set $S=1$. Immediately, the change propagates.



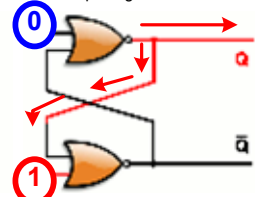
NOR2 now has 1 and 0 as inputs and outputs 0. !Q=0.



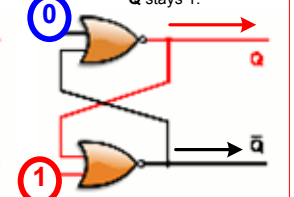
NOR1 has inputs of 0's, outputting 1.



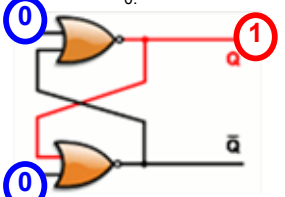
NOR1 has inputs of 0's, outputting 1. Q becomes 1.



NOR1 has inputs of 1's, outputting 1. !Q becomes 0. Q stays 1.



When $R=S=0$, NOR1 inputs are 0's. Q stays 1. NOR2 inputs stay 1 and 0. !Q stays 0.



To see this animated go to: http://en.wikipedia.org/wiki/File:R-S_mk2.gif

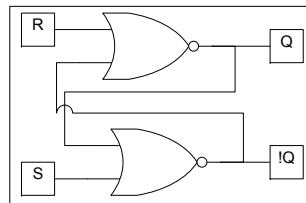
The bottom line of all this is that we now have a circuit that will hold onto a value, which we can read from Q. As long as S and R don't change, Q doesn't change. If Q was 0, then it stays 0. If Q was 1, it stays 1. But when we we put a 1 on R(eset), Q goes to 0. When R goes back to 0, Q stays at 0. When we put a 1 on S(et), Q goes to 1. When S goes back to 0, the Q stays at 1. (We never make both S and R equal one, since it causes Q to alternate between 0 and 1.) We now have a 1 bit memory circuit. We call this an **S-R Latch**, since it latches onto data.

16-Advanced Memory Latches

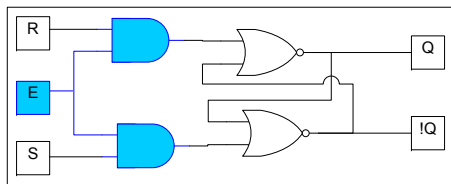
Copyright 2013. Ian Ohlander. All Rights Reserved

While this S-R latch is great at remembering one bit of information, there are some issues with this design.

Problem 1: We want to make sure that we only write to the memory when we *need* to update it. We want it to hold data and make that data available at any time. But we only want to write data when it *needs* to change. That way we don't accidentally overwrite it.

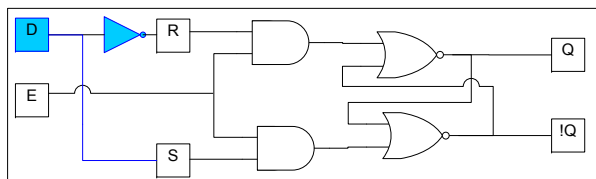


Solution: We can add 2 AND gates to enable inputs **S** and **R**. We'll call it **L**(oad)/**E**(nable). If it is 0, the **R** and **S** inputs are 0 and the data doesn't change. But if **E**(nable)/**L**oad is set to 1, then **R** and **S** inputs are active and will set or reset the SR latch.

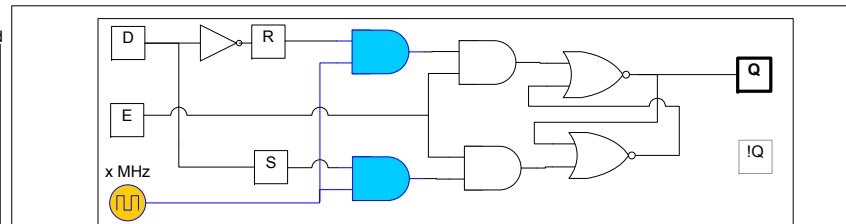


Problem 2: Setting $S=1$ and $R=1$ at the same time is too easy. And that will result in Q constantly flipping from 1 to 0 and back again, never latching onto anything. Plus, we have to deal with 2 inputs, complicating the interface it exposes to the outside world.

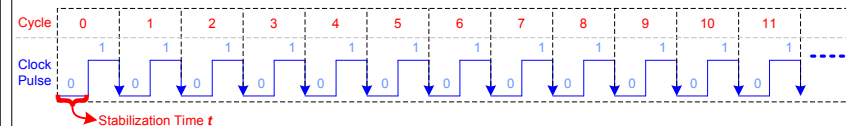
Solution: We can take a single input (**D**) and a NOT gate. NOT **D** goes into **R** and **D** goes into **S**. When we want to *write* we set **E**(nable)/**L**(oad)=1 and then we set **D**=1. The latch is loaded with 1. If we want to write a 0, we set **D**=0. If we want the latch to just hold the data, we set **E**(nable)/**L**(oad)=0.



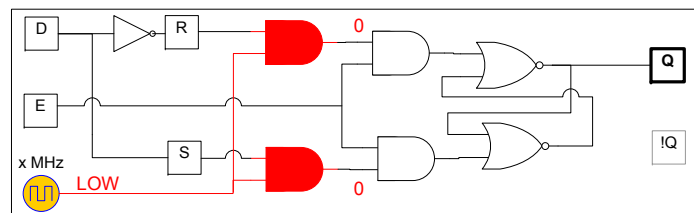
Problem 3: If you'll recall, when we set S or R to 1, there was a brief period where NOR1 and NOR2 propagated their new results out until the circuit stabilized. Because there is some "stabilization" time we need to wait for, we *only want to write at specific and known intervals*. We need to make sure the stabilization has occurred before we write again.



Solution: We can take care of the "stabilization" issue with this circuit. We simply add 2 more AND gates to the inputs of the SR latch to use a **clock pulse** with the inputs.



A clock pulse is an oscillating signal (alternating between 0 and 1). The signal stays low (0), for some time t and then goes high (1) for some time t . We need to make sure that t is long enough for the memory circuit to stabilize.



Thus, while the pulse is low (0) the **S** and **R** stay disabled at 0 for time t , (thus making no changes to **Q**) allowing the data to settle to a stable state. But the circuit can be read at any time. (This can pose it's own problem, as we may get bad data before the circuit settles down, but this is fixed by putting all memory circuits on the same clock. Thus they are never loading data from another memory latch before that latch has stabilized. They are in sync.)

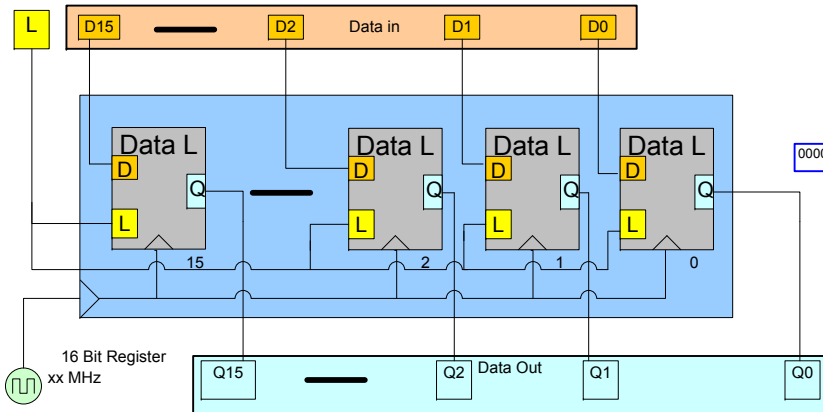
This **D**(ata) **l**atch will hold the data it is given for as long as we want. We can read it at any time. If we want to write to it, we set **L**(oad) to 1. The data will only be written in when the clock pulses. Since we are only interested in Q , we ignore IQ .

17-Memory Banks

Copyright 2013. Ian Ohlander. All Rights Reserved

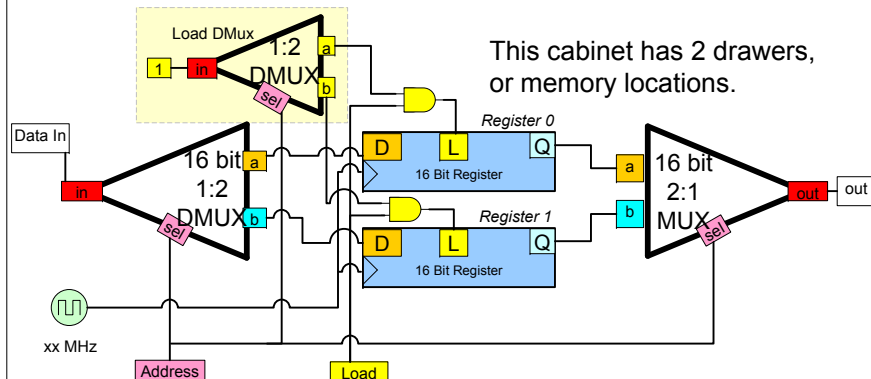
Once we have a 1-Bit Data latch, we can very easily create larger storage that handle more than 1 bit. We do this by stringing them together to each take in each bit of a larger number and wiring up the *loads* and *clocks* to the same incoming signal.

For example, to create a 16-Bit Data Register we route each of 16 data bits in into 16 D latches, set a common *L*(oad) input and clock input. The output will be what is stored in those 16 D latches.



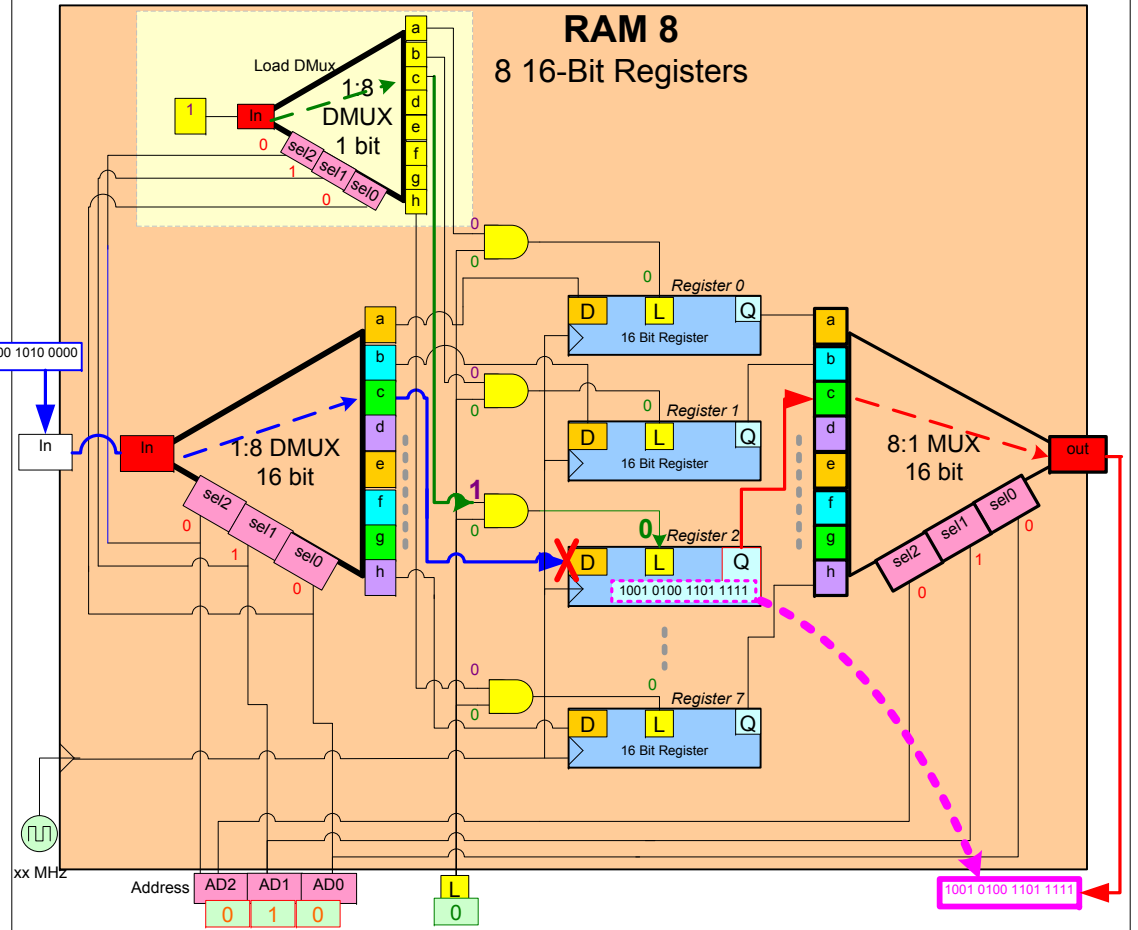
We can now read and write 16-bits of data (called a “word”) to and from what we call a register (*right*).

We can take this a step further. We want to create a bank of memory. We can visualize each register as a drawer in a cabinet. Each drawer is numbered (called an address) and can contain a 16-bit number. At any time, we want to be able to “open” and see what is inside one of those drawers using their address number. We also would like to be able to “open” and store numbers inside those drawers, using their address to choose which one. A Dmux-MUX combination module can do all this.



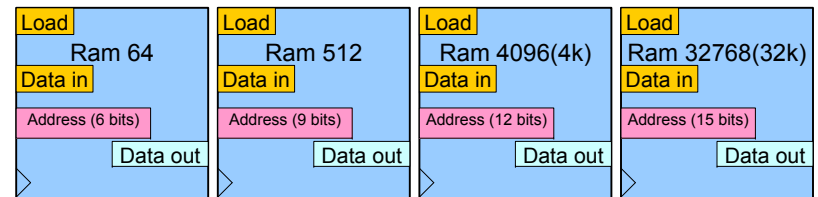
By putting in an **Address** (0 or 1) on our Mux, we can read from either register. If we want to write, we put the **Address** into the Dmux and set **Load**=1. A second “Load” Dmux (top) handles activating **Load** on *only* the selected register by ANDing it with the **Load** signal.

We can take that same design and use a 16-bit 1:8 Dmux/8:1 Mux to address a block of 8 16-bit registers with 8 **words**. We will use a 1-bit 1:8 Dmux as the “Load” Dmux. To address 8 possible registers, we will need to use 3-bit wide addresses to select from the Dmuxes and Mux. Other than that, the design is exactly the same as the 2 register memory module.



With the design above, we can input an address and read the date in that register on *out*. This example sets **Address**=010 (2). This activates the 3rd output on the MUX and the value inside *Register 2* (1001 0100 1101 1111) is passed out. Because **load**=0, none of the registers do any loading, despite the Dmux passing the input data into *Register 2*. If **load**=1 then 0000 0100 1010 0000 would have been loaded into *Register 2* (when the clock pulsed). Now have an addressable bank of 8 16-bit registers (RAM 8).

We can repeat this *by replacing the 8 individual 16 bit registers with our new RAM 8 module* (and adding 3 more bit address inputs to handle the new Mux/Dmuxes) to get a bank of 64 (8x8). We can do this again and again (replace 8 with 64, etc), always adding 3 more address bits, to get 512, 4k, 32k.



18-CPU Design- Overview

Copyright 2013. Ian Ohlander. All Rights Reserved

At this point, we have built digital components, like muxes and adders, an ALU, as well as memory to store information. We need to ask the question, do those things make a computer? What is a computer? Is a computer a calculator? It calculates. Our ALU calculates (in a very limited fashion), depending on the control-bits we pass it. Is that a computer?

No. A computer does more than calculate. Nor does it simply store information. A computer follows instructions. No more and no less. It does exactly what it is told, but very very quickly. These instructions basically make up 3 types:

- 1) Calculation
- 2) Reading to and writing from memory
- 3) Making decisions.

That last part- making decisions- is what keeps a computer from being a simple calculator. It will follow instructions and make decisions based on the results of those instructions, input from a user, the state of a mouse or any other piece of data.

Making decisions may sound overly complex, but we actually mean something very simple: *Testing for a specific condition and acting on it.* ("Does $x-y=0$?", If 'yes', do one thing. If 'no', do another.) That's it. That's all a computer does. That's all it ever does. Everything else we see a computer doing- displaying movies, letting us surf the internet, allowing us to write a letter- is simply the result of a computer doing those 3 things quickly over and over again in many different ways.

At its core, a computer's brain is the Central Processing Unit (**CPU**). It is made up of (not surprisingly) three major types of components:

- 1) ALU to perform simple calculations
- 2) Memory to store information
- 3) Instruction-following logic

We have already completed the first 2 of those elements. This last one, the component to allow a CPU to follow instructions, is the most complex and most powerful. When put all together, we will have a fully-fledged computer able to execute any program we care to write for it. (Of course, this computer will not be easy to use at all, yet. All we have been working on is its brain. The friendly human-machine interface requires an OS and would take much more work- including designing the software to make it as easy to interact with as possible. We won't go into any of that now.)

Before we delve into the CPU, let's first figure out what we mean by instructions. How does a computer follow instructions? We answer that by looking at an example involving regular human tasks.

Let's say we wanted to tell someone how to make breakfast of boiled eggs and toast. We could do it pretty simply with the following steps:

- 1) Boil egg
- 2) Toast bread

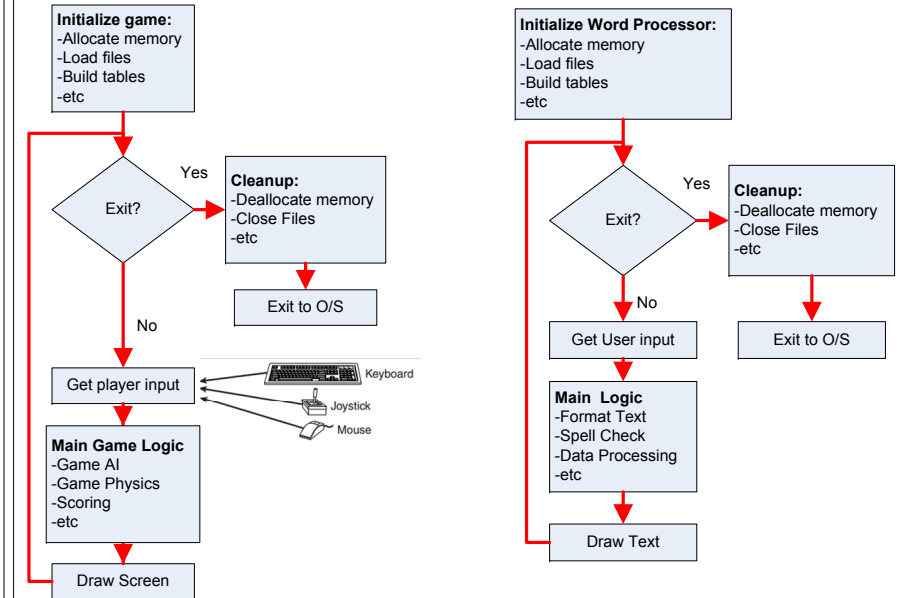
Of course, those are "high-level" steps. They each are made of more basic steps. If someone, for example, didn't know how to boil an egg or make toast, we'd have to give more explicit instructions. We could expand step 1 to be:

- 1a) Put water in a pot
- 1b) Turn on heat
- 1c) Put egg in pot
- 1d) Wait 5 minutes

Those steps would allow anyone to make boiled eggs (though of course we left out the taking of the eggs off the stove and then peeling the shells off of them). The same is true if we define how to "toast bread".

Those step-by-step instructions make up a "recipe" or "algorithm". Our steps can be "high-level" (boil egg) or they can be "low level" (get pot, put water in, etc).

Computers just follow directions. Because they are machines, there is no "common sense" (how to boil water). Each step must be explicitly defined. Computers just follow those instructions, or an algorithm, of which each individual step is made up of lower level algorithms. For example, a computer game or word processor algorithm may look like this.



Each one of those steps actually constitutes a number of lower-level algorithms. And those algorithms, in turn, are actually made up of still lower level algorithms. At its base, a computer, and more specifically, its brain the Central Processing Unit (CPU), is doing rudimentary calculations, reading from and writing to memory, and making decisions based on those calculations.

Even things like getting user input from a keyboard or mouse or drawing on the screen is done by writing to or reading from memory (using an ingenious technique called "memory mapping".)

So our goal will be to now design a CPU that will do those 3 basic operations. It will "read" a binary code that will make up one single instruction. It will then do what that instruction says and then go on to the next one.

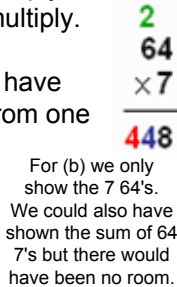
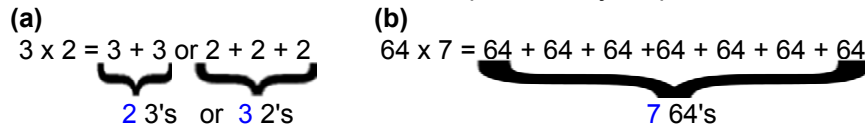
To do that, we need to determine exactly how this CPU should work- what it will need to read the instruction and follow it. We can do this by studying a very basic algorithm and seeing what we would need to follow it.

19-CPU Design- Algorithms and Operations

So let's look at a literal example of what we need a computer to be able to do. From there, we can determine what kind of functions, or operations, the CPU must be able to perform. Once we have determined exactly what the CPU should be able to do, we can then get to figuring out how to implement it.

Currently, our ALU only adds and subtracts. We are going to need it to multiply if it is going to be any use to us. So let us figure out how to teach a computer to multiply.

What does multiplication mean, anyway? (a) 3×2 ; (b) 64×7 . For (a) we have memorized that $3 \times 2 = 6$. For (b) we do long multiplication, carrying digits from one column to the next. But at its base, multiplication is just quick addition.



For (b) we only show the 7 64's. We could also have shown the sum of 64 7's but there would have been no room.

Because we have usually memorized the first 12 multiplication tables and the technique of long multiplication, we generally forget that all multiplication is, and ever has been, is repeated adding.

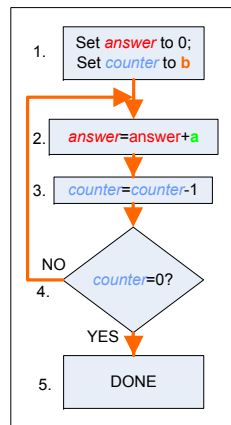
So if we had to direct a computer to multiply, what would we do? We could figure out and teach it the algorithm for long multiplication. In fact, that is the best way. It's a very efficient algorithm (which is why we do it!). But let's keep things easy. Instead, let's just tell the computer to multiply by repeatedly adding. If we have 2 numbers, **a** and **b**, we tell the computer to add **a**, **b** times. If we had **a** x 6, we'd tell the computer to add **a** six times. It's not efficient **at all**, especially for large numbers (imagine 82,123 x 120,125,634!) and so is called a naïve algorithm. But it's a start and easy enough for us to figure out.

So we have 2 numbers, **a** and **b**. We want to add **a** **b** times. If **b** was 85, we'd need to make sure that we added **a** exactly 85 times.. The best way would be to start a count of each addition of **a** (to some partial answer) and stop at **b** (85). Or, we would start at **b** (85) and count down: add **a**, **b=b-1**: 85-1=84; add **a**, 84-1=83... Of course, we don't want to change **a** or **b** themselves, so it would be best to copy **b** somewhere and use that as a counter. We can describe this algorithm with a flow chart.

Given: two numbers to multiply: **a**, **b**
 one number to count: **counter**
 product of **a** x **b**: **answer**

We can see the algorithm works when **a=2** and **b=3**.

- 1: **answer** = 0;
counter = **b** = 3
- 2: **answer** = **answer** + **a** = 0 + 2 = 2
- 3: **counter** = **counter** - 1 = 3 - 1 = 2
- 4: **counter** = 0? NO - we jump to step 2.
 2: **answer** = **answer** + **a** = 2 + 2 = 4
 3: **counter** = **counter** - 1 = 2 - 1 = 1
- 4: **counter** = 0? NO - we jump to step 2.
 2: **answer** = **answer** + **a** = 4 + 2 = 6
 3: **counter** = **counter** - 1 = 1 - 1 = 0
- 4: **counter** = 0? YES - we continue to step 5.
 5: **answer** = 6 (2 x 3)



This algorithm will work for any 2 numbers (**a,b**) whose product can be encoded in binary using 16 bits (the answer has to be between (-32767 and 32767). To do larger numbers we'd have to modify the algorithm. But for our purposes, it is sufficient, both in terms of utility (it multiplies 2 numbers) as well as to help us figure out what our CPU needs to do.

So looking at our algorithm, we note that the CPU will have to be able to do the following:

- I) read and write from memory locations **counter** and **answer** (1)
- II) Do a simple calculation (2,3)
- III) test the results of a calculation (4)
- IV) jump to another instruction based on that test (4)

The ability to do these things requires a few things.

a) all the instructions (steps in the algorithm) the CPU will be following need to be stored somewhere. We will call that our **Instruction Memory**.

b) I requires the CPU to interact with memory locations. Thus, we will need a block of **Memory (RAM)**. We will use addresses to read from and write to that memory.

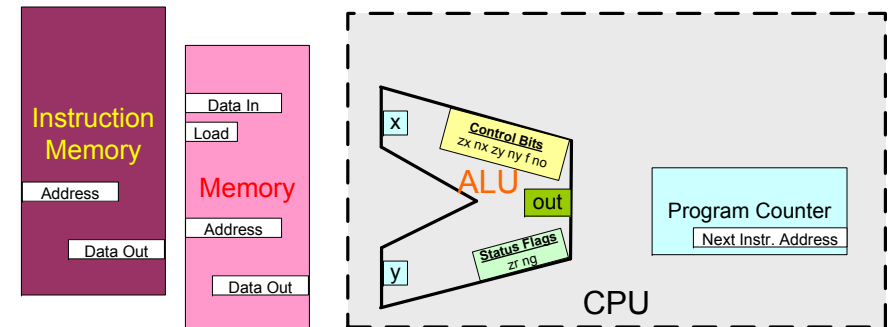
c) II can be done by the ALU. The ALU can perform simple calculations and its **status flags** will tell us things about the result.

d) III will require us to do some testing of those status flags.

e) IV requires that if the conditions are met (or are not met), the CPU will jump to another step in our Instruction Memory and continue executing instructions from that location. (We see this in the algorithm step 4, where we jump to step 2 if **counter** is not 0.)

To do that, we will need some kind of component to store the address of the **next instruction step**. We will call it the **Program Counter**. This allows us to jump by putting another instruction address in that Program Counter. This component will be the driving engine of our CPU. It is what will allow the CPU to perform its tasks, step by step. So we will spend some time on the motivation for this component and its design later.

Visually, lets lay out all the components we are going to need so far



Let's notice that the Instruction Memory doesn't take in any data. It is Read-Only Memory (called ROM). To load the computer with another program, we just "pop in" another ROM (think old Nintendo cartridges.) Our CPU has only 2 components for now (one just conceptual!). We will change that soon.

20-Computer Memory

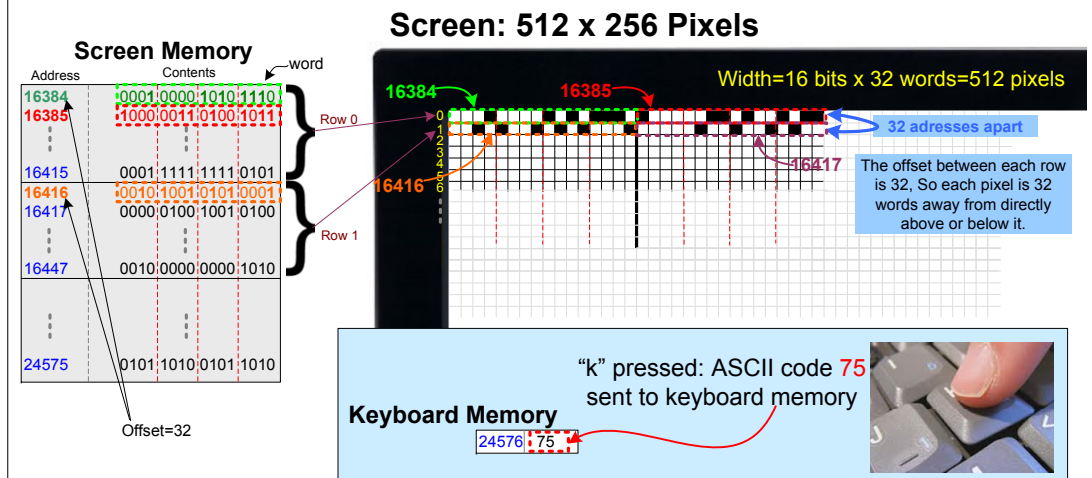
Copyright 2013. Ian Ohlander. All Rights Reserved

We can use the existing memory chips we made earlier to create the **Instruction Memory**. We'll use the 32K RAM chip with **Load** permanently set to 0. This makes this Read-Only Memory (ROM). Our instructions will be stored in that ROM and each instruction will be read by the CPU.

Now we need to create the computer's memory. We want it to be 24K. That means we'll need 15 bits to address it, since 15 bits can handle up to 32,768 addresses. (14 bits will only address 16,384 addresses, which is not enough). To get 24K, we will stack a 16K and an 8K module (and one more, which we will explain in a moment.)

It is critical that we realize that our memory will do **more** than simply store information and be a place for the computer to do work. Computers need to be able to interact with the outside world: accept input from a keyboard, a mouse, a stylus or even a finger on a touchscreen. They need to output images to a screen, a printer, or over a network card onto a network. And each interaction method should be simple and consistent across all devices. Reading input from a keyboard shouldn't depend on one method of communication while reading input from a mouse depend on another.

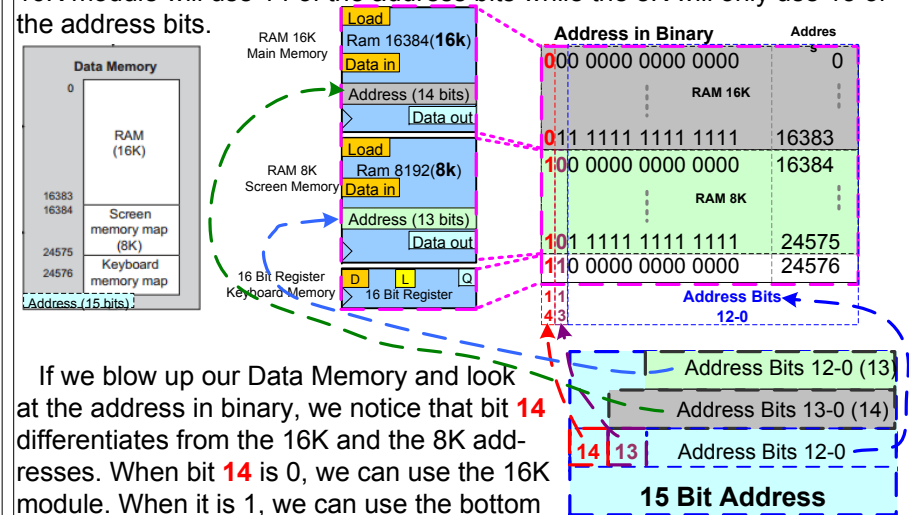
So computer designers came up with a strategy called "Memory Mapping". The idea is that each device- from display to mouse- is "given" a block of memory. Writing data to those memory locations then "sends" information to a device. So to draw on the screen, specific data is written to the screen memory. Reading data from those memory mapped locations allows the computer to receive input. The keyboard's memory map location contains a code for what key is being pressed.



But we need to keep in mind, this memory is *regular old memory*. There is nothing special about it. It is the **device** that has the responsibility to interact with the memory. It is the screen (its hardware and driver) that is scanning the computer's screen memory and updating its display many times a second to reflect what it found in that location. The computer is just storing information in that memory like it does in any other memory address. And the keyboard is constantly storing in the computer's memory the ASCII code of the key that was pressed. The computer checks that location when it receives an interrupt signal, to see what was put in there, just as it checks any other memory location when it is instructed to.

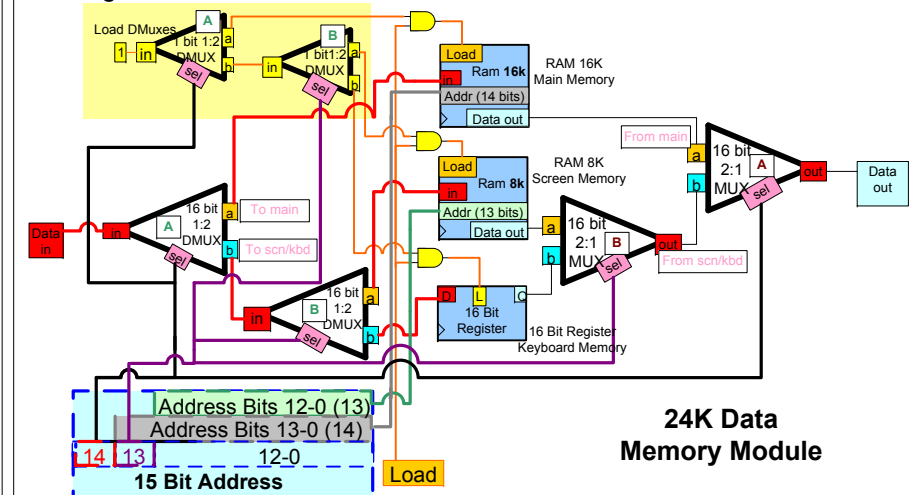
So we will use a single 16-bit register for the keyboard's memory along with 16K and 8K.

We are going to use 15 bits to address these 3 modules. We need to figure out how to do that so that we get data to/from the right module. The 16K module will use 14 of the address bits while the 8K will only use 13 of the address bits.



If we blow up our Data Memory and look at the address in binary, we notice that bit 14 differentiates from the 16K and the 8K addresses. When bit 14 is 0, we can use the 16K module. When it is 1, we can use the bottom 2 modules (the 8K and the 16 Bit register.) Then we look at bit 13 and we see that it can be used to select between the 8K and 16 Bit register.

We thus can use bit 14 to select between Main and Screen/Keyboard memory, and then bit 13 to select between Screen and Keyboard memory. That means 2 Dmuxes to handle data flow in and 2 Muxes to handle data flow out. They will address with bits 14 and 13 respectively. As we did with our original memory modules, we will use "load" Dmuxes to handle write-enabling our 3 modules.



Bit 13 selects between screen and the keyboard on the **B Dmux/MUX**. Bit 14 selects between main memory and screen/keyboard on the **A Dmux/MUX**. (For lack of room, we are not depicting the clock.)

21-CPU Preliminary Architecture

So we have our ALU for calculation and our memory modules, the Instruction ROM and Computer Memory, completed. All that is left is to design the CPU layout- its logic components that decode our CPU instructions from the 32K Instruction Memory ROM. Let's look again at our multiplication algorithm and the requirements that it gave us for the CPU. Those requirements will allow us to create the logic decoding components.

- I) read and write from memory locations *counter* and *answer*
- II) Do a simple calculation
- III) test the results of a calculation
- IV) jump to another instruction based on that test


Let's go through these one by one.

Requirements I & II: The CPU needs to interact with memory locations and perform calculations. The CPU will use our 24K memory as an external storage. But performing operations regularly on external memory is a time intensive operation. Most algorithms require a lot of temporary storage for intermediate calculations (like the counter). Though we are talking milliseconds, accessing computer memory is relatively slow. Only accessing a mechanical device like a hard drive, cd-rom, or getting information over a network takes longer. The best solution is for the CPU to get a copy of what it needs from memory, do operations on it in some local storage, and then put the result back in external memory.

So we are going to add two **16-Bit registers** to our CPU. If we look at our ALU, we note that it has 2 inputs: **x** and **y**. So one register for the **ALU-x-input** and one register for the **ALU-y-input** fits nicely. We will call them **D register** and **A register**. **D** will hold *data* and go into **ALU-x-input**. **A** will hold either *data* or a *memory address*. Why?

We need a way to get data **into** our CPU and memory. So we'll make **A** be a single data input point. Remember that **A** will hold either *data* or an *address* to our memory (such as the screen memory). So we want to be able to directly load **A** with either an actual value or address in memory.

But we **also** may need to *calculate* an address or value and store it in **A**. Think back to our screen memory explanation. Each row contains 32 addresses. If we want to draw a word in the 2nd row, we need to take the top left corner's address (16384) and then add 32 to get to the position of the 1st 16 pixels in the 2nd row (16416).

We could (somehow) put 32 in **D** and then 16384 in **A** and have the ALU add them together. We would then put the result back into **A**. Now **A** has the address of the screen where we want to draw. We could then write `0010 1001 0101 0001` (10577) at that address to draw this on the 2nd line of the top corner: 

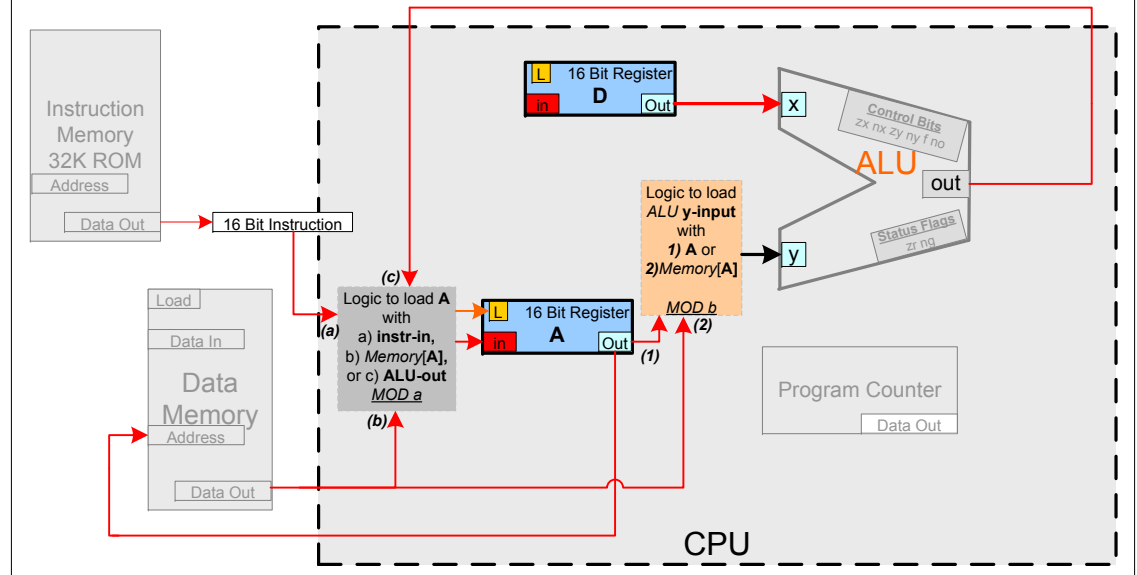
So **A** will be loaded from: **(a)** directly from our instruction, **(b)** from memory at the address stored in **A**, or **(c)** from the result of some ALU operation. How we can do that we don't yet know, so we will also leave that as a magic **grey** box for now.

Similarly, the **ALU y-input** should get input either directly **(1)** from the **A** register or **(2)** from the data stored in memory at the address in **A** (we'll write that as **Memory[A]**.) Let's say we want to know what key was typed. We'd put our keyboard memory address (24576) into **A**, send that to **memory-address-in**, and then put the resulting output of **Memory[24576]** into the **ALU-y-input**. It would now contain the code of the typed key. This would be just the first step as we might then test to see what letter was typed. We haven't yet decided on how to load **y-input** in both ways, so we will leave it an **orange** magic box for now.

Notice, too, that we haven't yet figured out how to get data into **D** (and from there into **x-input** on the ALU). So we have to do a little more planning to get this working.

But at this point, we can look at our 16 bit instruction more closely. So far, it needs to load the **A** register from the **instruction** itself, as well as from another source (**Memory[A]** and **ALU-out**). It also needs to load the **ALU y-input** from 2 possible sources (**A**, **Memory[A]**). And yet we only have 16 bits per instruction to do all that. 16 bits may sound like a lot, but is actually pretty limited. But we can be clever about our instructions.

We can use our 16 bit instruction in 2 different ways. The first loads the **A** register, allowing us to get data into the CPU. The second way would tell the CPU *what* to do (such as how to load the **ALU-y-input**). So each instruction would either be a data-entry instruction or a work instruction. Notice that because we are in binary, we can represent both choices using a single bit. We can (arbitrarily) decide to designate one of its bits as an *instruction-type* bit. If we set it to 0, then we are putting data into **A**- a data-entry instruction. If our instruction-type bit is 1, it is a CPU work instruction. With that in mind, we can define our instruction more clearly.

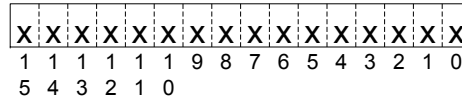


22-CPU Instructions

Copyright 2013. Ian Ohlander. All Rights Reserved

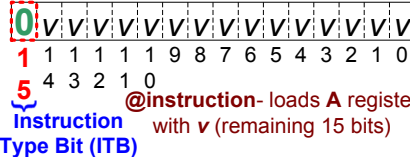
Our Instruction Memory will contain hundreds or thousands of instructions. But each instruction will just be a 16-bit number at its core. So let's design our instruction to do some real work. Recall that we decided we wanted the instruction to do 2 things (to meet Requirements I and II).

16 Bit Instruction



- 1) Read/Write data from/to memory (**A**, **D**, and **M[A]**)
- 2) Tell the CPU to do some operation

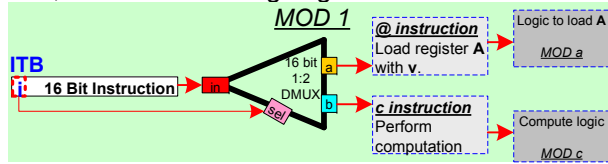
With these 2 options, we can designate the 15th bit, an **instruction-type bit (ITB)**, to indicate which job we want the instruction to do. When that bit is 0, it loads the **A** register. When it is 1, it performs some operation. We'll call the first type an "**@ instruction**". We can represent an **@ instruction** like this:



It tells the CPU to load the **A** register with the 15-bit number represented by **v**. (We'll worry about loading **D** and **M[A]** later.)

The other type of instruction performs an operation or computation. We'll call it a "**c-instruction**". The remaining 15 bits will specify the details of the operation.

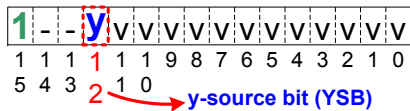
Given two possible options based on the **ITB**, we know we are going to use a 16-bit 1:2 Dmux, using **ITB** as the switch. We show all this in **MOD 1**.



Now, let's define how our **c-instruction** works. First, remember that the **c-instruction** tells the CPU to perform some computation. So immediately we know that it will contain instructions for the ALU.

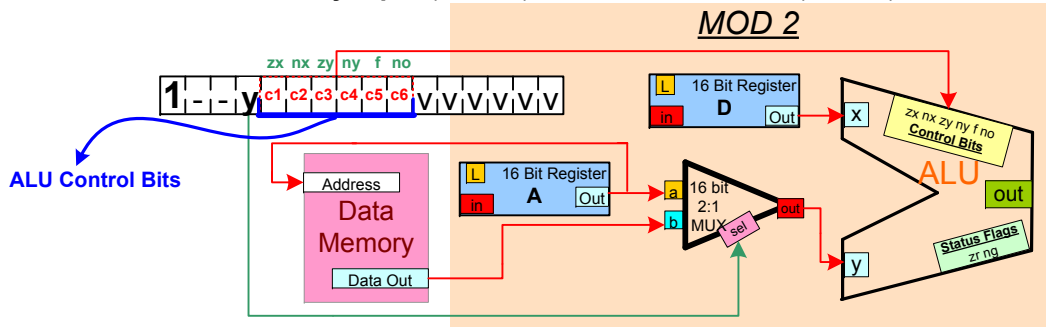
Also, remember the orange box (**MOD b**)? Our **ALU y-input** would take in either the **A** register or **Memory[A]**. This allowed us to perform computations on data we either just loaded into the **A** register or on **memory** at the address we loaded into **A**.

Again, all we need to choose between the 2 possible sources is a single bit indicator. We'll call it **Y-Source Bit (YSB)**. So we can represent our **c-instruction**, so far, here:



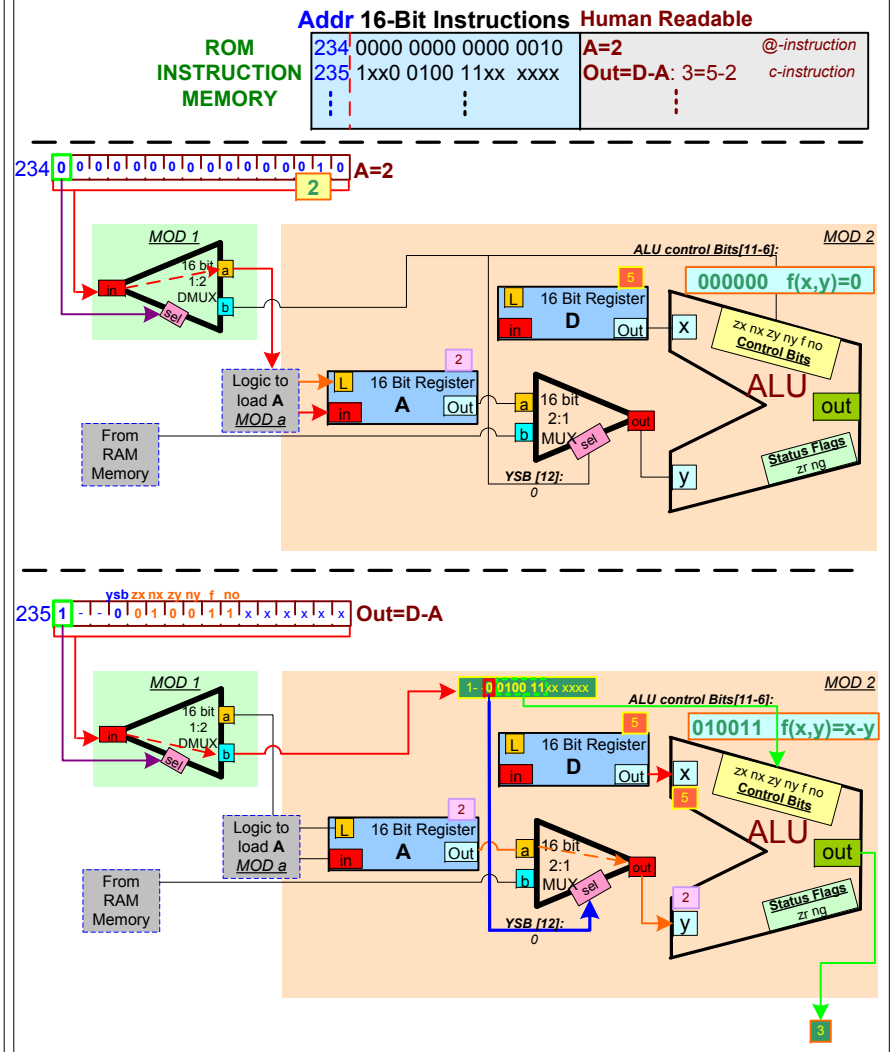
As with the **@-instruction**, we know we are going to use **YSB** to select, this time between 2 inputs. That means another 16-bit 2:1 MUX, which will take care of loading **ALU-y-input (MOD b)**.

Let's also add our 6 **ALU Control Bits**: **zx, nx, zy, ny, f, no** into our instruction. If we do all that, we can combine **MOD b** and **MOD c** into **MOD 2**, since we now can decode an instruction that tells us how to load the **ALU y-input (MOD b)** and what to calculate (**MOD c**).



Let's take a step back to assess what we have. Our 32K ROM module will contain hundreds and thousands of CPU instructions. Each instruction will reside in a sequential memory address. The CPU will start at the top of the instructions and then (using our as-yet unmade program counter and our clock pulse) work it's way down, responding to each coded instructions.

Let's visualize how this works so far. Let's assume that we have somehow loaded **D** with 5. And we are currently at address 234 and 235 of our ROM instruction memory (with a human readable version next to it). Let's see how what we have come up with so far interprets those 2 instructions at 234 and 235. The first instruction (234) loads **A** with 2. The second instruction (235) tells the ALU to calculate **D-A**.



23-CPU Instructions- Writing to Memory

Copyright 2013. Ian Ohlander. All Rights Reserved

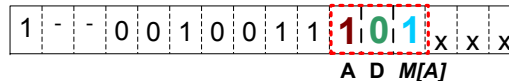
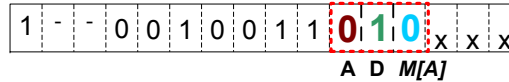
So at this point we can tackle the question of how our *c-instruction* can write the results of any computation it does to our 3 memory modules: **A**, **D**, and **Memory[A]**. If we look back at our instruction, we notice we still have 6 bits on the very end (bits 5-0). Since we have 3 memory locations, we can use 1 bit to load each of them. Thus, we can write to all, none, or some combination of, our 3 memory modules (*above*).

Let's look back at our example *c-instruction* in our memory ROM address 235 (right). We can replace bits 5-3 (xxx) with our destination information.

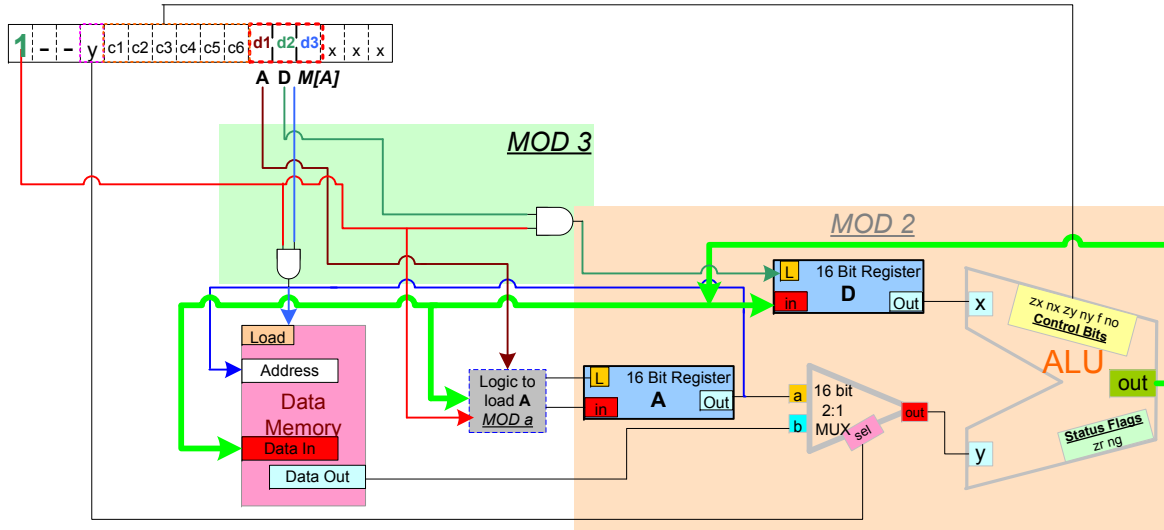
For example, we can load **D** with the result of that operation.
D = D-A:
 We could also load **M[A]** and **A** with that result.
M[A], A = D-A:

Addr 16-Bit Instructions Human Readable

235 1xx0 0100 11xx xxxxx Out=D-A: 5-2=3



Doing this is actually pretty straightforward. We are basically saying we want to load **A** and/or **D** and/or **M[A]** with the **ALU-Out**. So that means we route **ALU-Out** into **D Data-In** and **M[A] Data-In**. We also route **ALU-Out** into **MOD a** (our logic to load **A** register) since we haven't figured out how to do that yet.) Finally, we note that we are **ONLY** writing when it is a *c-instruction* (and thus a computation has been done). We have to make sure that our destination-bits [5-3] load each of those 3 memory modules (**A**, **D**, **M[A]**) *only* when **ITB=1**. (If we didn't do that, then if we had an *@-instruction* where **v** happened to have a 1 in bits [5-3] (as in the 1 in 0000 0000 0000 1000, for example) then one or more our memory modules would load (in that example, it would be **M[A]**, that would be overwritten, even though we were just loading **A** with the number 0000 0000 0000 1000.) We take care of this by ANDing **ITB** with each of those *destination-bits* (activating them) and routing that into those modules' **Load** inputs (**MOD 3** and **MOD a**, which we still don't know, yet).



This scheme will do everything we want. We've hidden away **MOD 1** and **A** register's output to **Memory[address]**, to make things more simple to see. When we have a *c-instruction*, the ALU will perform an operation and then will put the answer in up to all 3 locations: **A**, **D**, **M[A]**

And now this allows us to load the **D** register as well as write to memory (**M[A]**). Remember the steps of our multiplication algorithm. We wanted to be able to:

- I) Read/Write from memory locations (**A**, **D**, **M[A]**)
- II) Perform calculations.

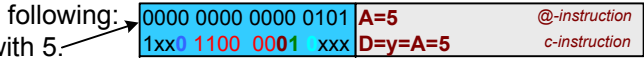
We've already taken care of #2. And we already could (sort of, missing **MOD a**) load the **A** register from an *@-instruction*. Now we can load it and **D** and **M[A]** from an ALU operation. How?

Look back at the ALU control-bit functions, specifically this:

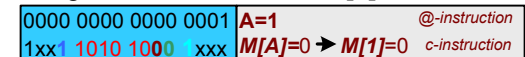
zx	nx	zy	ny	f	no	f(x, y)=
zero x	NOT x	zero y	NOT y	AND/add	NOT output	output function
1	1	0	0	0	0	y

Notice that we can output whatever is coming in on **ALU-y-input**. So if we have **A** or **M[A]** (selected by **YSB**) coming in on that input, we can output that to **D**.

For example, let's say we wanted to load **D** with 5 (0101). We could do the following:
 We load **A** with 5:
 Next, we set **YSB=0**, so that **ALU-y-input=A**. We then set our **ALU control bits** to 110000 to output **y-input**. We then set destination to **D** (d2=1).

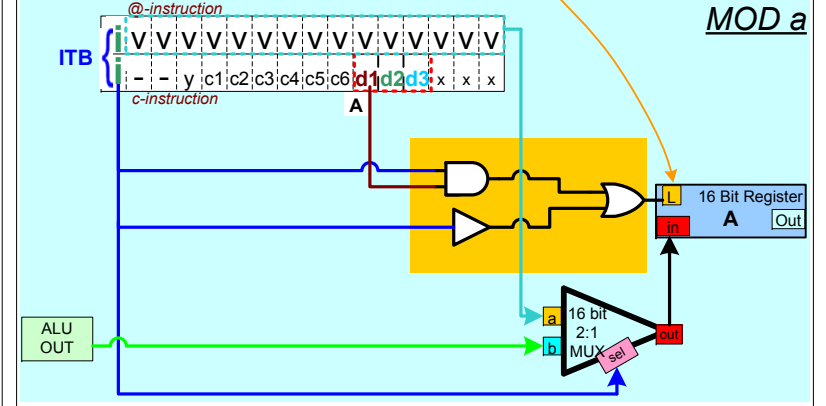


We can write data to **M[A]** the same way, using **d3**. In our multiplication algorithm (step 1), we wanted to set **answer=0**. Looking back at our *control-bits* chart we see the ALU can output 0 when they are 101010. **answer** refers to a memory location, so **YSB=1**. If we assign **answer** to refer to **M[1]**, we could do this:



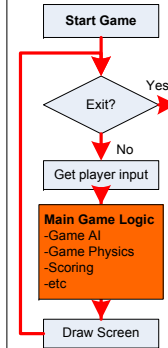
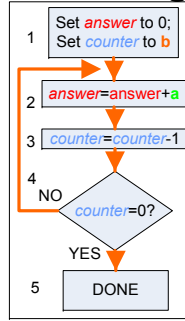
Finally, let's get our **MOD a** figured out, now that we have both sources for loading **A** register defined. We want to load **A** when we have an *@-instruction* (**ITB=0**) **OR** when we have a *c-instruction* (**ITB=1**) **AND** the **A destination bit** (**d1**, bit 5) is 1.

We can write that out as: **LOAD = NOT(ITB) + (ITB x d1)**
 We load **in** either **v** or **ALU-out** depending on **ITB** (using a 16 bit 2:1 MUX with **ITB** as selector.)



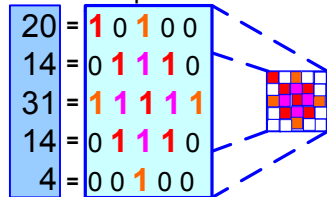
24-CPU Instructions-Power of Testing and Branching

Before we create our final module and **Program Counter**, it would be good to understand how critical it is that our CPU be able to test its computations and then branch to other program steps. If we look at our multiplication algorithm, we see that until the counter is 0, we keep repeating the addition. The flow the program is based on that test. When the counter finally reaches 0, then we stop and answer holds the product of $a \times b$. That means that if we are multiplying $32,001 \times 5,532$, we will repeat the addition of 32,001 5,532 times! The computer will obediently keep doing its test (does counter=0?) and if not, it will keep doing the addition. If the CPU didn't have that ability then it would not be able to multiply at all (or do anything other than add and subtract.)

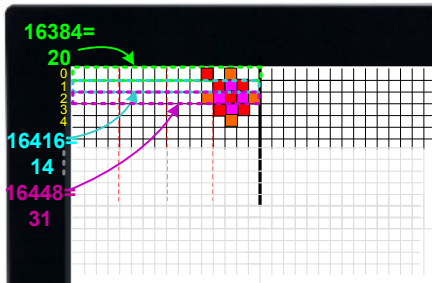


But let's leave the realm of mathematics (as critical as that is to a computer's function) and see how testing and branching make games possible. A basic flow of a game can be seen in a simple flowchart. In particular, notice **Main Game Logic | Scoring**. Let's say this is a game where you fire and if it hits a ship you get a point. How does the CPU know if your bomb hits the ship?

If you remember back to Screen Memory Maps, you'll remember that drawing on the screen is done by simply writing 1's or 0's into a certain location in memory. Each memory bit in addresses 16384-24575 corresponds to one specific pixel, or dot, on the screen. A 1 there makes it black, a 0 makes it white. So I can draw a bomb by first putting together the bit code for the image.



Then I write that bit code (20,14,31,14,4) into my screen memory.



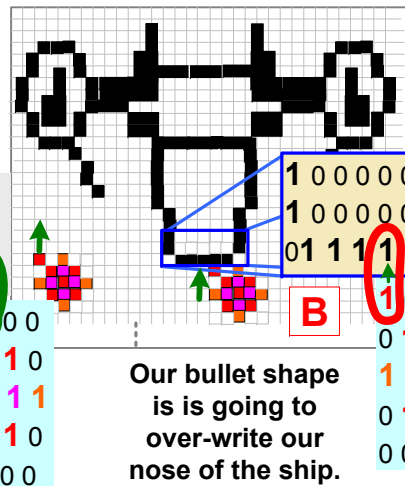
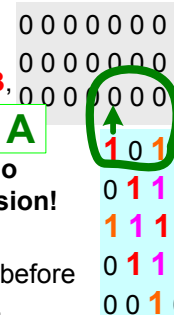
20 is stored in the top left row, address 16384. The next row of the image, **14**, goes into the memory location for the 2nd row, far left, which is $16384+32=16416$. The 3rd row of the image is **31** and we put it in $16416+32=16448$. And so on.

So let's saw we are drawing-erasing-drawing our bomb across the screen.

We have to calculate its new position. Once that is done, we can look at that location and see if something is already there. If so, we have a collision.

Look at **A**. Notice we have a 1 going into a 0. No collision. So $1 \text{ AND } 0 = 0$. But in **B**, we have a 1 going into a 1, which means a collision. $1 \text{ AND } 1 = 1$. That tells us that we just have to AND our moving shape, row by row, with the data that is already in the screen memory at that location.

This is a necessary function that we perform before we continue with the game animation and play.

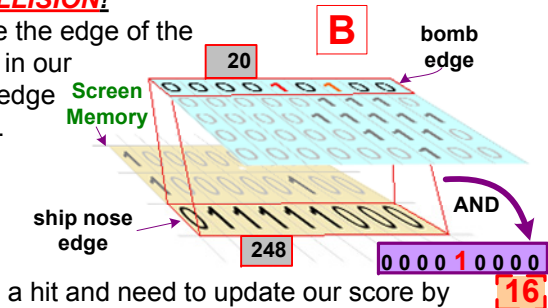


We can define this algorithm more explicitly like this. We define a counter we call i . Each 16-bit section of the Screen Memory is **Screen Memory [i]**. **Screen Memory[0]** would refer to the contents of 16384, the top left 16 bits of the screen. Adding 32 will drop us down one row, so **Screen Memory[32]** refers to 16416, etc. We define j to refer to each row of our shape. So **Shape[0]** in our bullet would be 20. **Shape[1]** would be 14. We only need to check the total rows of the Shape with what is already on the screen, so we also define a variable called **total** that contains the number of rows in the shape. (In our bomb, row **total=5**.) Then we **AND** our Shape bit with the contents of our Screen at i and the result tells us if there is collision.

$$\text{Collision} = \text{Screen Memory}[i] \text{ AND } \text{Shape}[j]$$

If any bit in **Screen Memory[i]** has a 1 that corresponds to a 1 in **Shape[j]**, then the result of ANDing those 2 numbers together will be a number. **It doesn't matter what the number is, but as long as it is NOT ZERO it means COLLISION!**

Look again at **B**. We see the edge of the ship's nose (bit code 248) in our **Screen Memory** and the edge of the bomb (bit code 20). Notice how we **AND** them together. The result is **16**. We have a **Collision!**



We can generalize this.

If **Collision ≠ 0**, we have a hit and need to update our score by 1 point. We also don't need to continue checking to see if a collision occurred, so we can end our collision check. From that point onward, our **Main Game Logic** would proceed onto its next step.

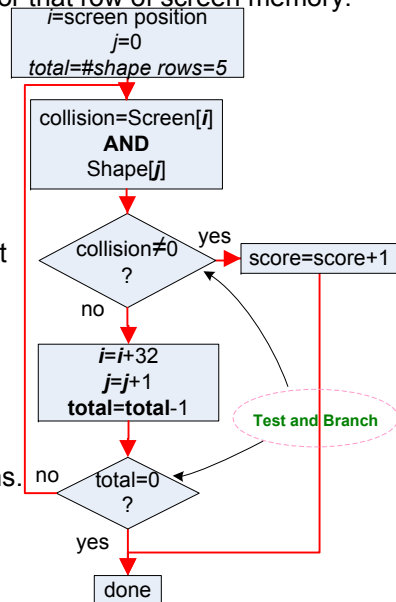
If **Collision = 0**, we have no collision for that row of screen memory. So we increase i and j and decrease the **total** number of rows to consider.

Then we repeat the whole process until all the rows of the shape are done (**total=0**).

From that point onward, our **Main Game Logic** would proceed onto its next step. It's not the best and most efficient algorithm. But it does the job.

The point of all this is to make clear how important it is that our CPU be able to **test** and then jump, or **branch**, to another place in our instructions.

Branching is what gives computers such versatility and power.



25-CPU Instructions- Testing and Branching

It is at this point that we come to the final 2 modules of our CPU to meet *Requirements III and IV*. First, as we saw, the ability to test for conditions and act on them is critical for our CPU. So we need a way to specify to our CPU what **ALU-out** conditions we are looking for. If those conditions are met then the CPU needs to be able to jump to other instructions. We will get to the jumping (and the **Program Counter (PC)**) in the last building project.

So how can we specify what sort of conditions to look for? In our two examples, we wanted to test whether or not **counter=0** and whether

Collision $\neq 0$. So we have **=** and **\neq** as possible tests. If we think about it, we could have also made our counter test this: $\text{counter} > 0$? Then we would loop as long as that was true. Once counter hit 0, we would be done. When we extend that, we can come up with *8 types of jump conditions (right)*:

Null	No Jump	
JGT	Jump if Greater Than 0	>
JEQ	Jump if Equal to 0	=
JGE	Jump if Greater than or Equal to 0	\geq
JLT	Jump if Less than 0	<
JNE	Jump if Not Equal to 0	\neq
JLE	Jump if Less than or Equal to 0	\leq
JMP	Jump no matter what	

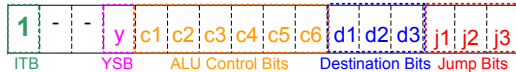
Notice that we have 3 different core conditions: **<**, **=**, **>**

But when we look at our ALU, we notice that we only have 2 flags: **ng** and **zr**. **ng** means less than zero (<0). **zr** means the ALU computation equals zero ($=0$). So we have 2 out of 3 of our core conditions. But if we think about it, we have our 3rd condition as well, that the **ALU out** is greater than zero (>0). How? Well, if we have a number and we know that it is NOT less than 0 and NOT equal to 0, then we **know it MUST BE** greater than 0. It must be a **positive** number. We can write and diagram this new flag, **ps**:

$$\text{ps} = \text{NOT}(\text{ng}) \text{ AND } \text{NOT}(\text{zr})$$

Now we have 3 flags to indicate what the ALU has output. So now we need to tell the CPU what conditions we are looking for. To do this we are going to use our CPU instruction. We've been able to use the *c-instruction* to tell the **ALU y-input** where it needs to come from (**A** or **M[A]**), indicate what operation the ALU should do (**control-bits**), and also where to store the result (**destination bits**). We can use our final 3 bits of the *c-instruction* to indicate the conditions necessary for a jump.

c-instruction



j1 will indicate **negative**, **j2** will be **zero**, and **j3** will be **positive**. So now we have 3 jump conditions (which we can group, like greater than or equal to) and 3 status flags (which can also be grouped). Now we just want to test if our jump-conditions match our ALU-out flags. We can do this by ANDING each possibility together. If our jump-conditions (**j-**) match our out flags, we jump. We can modify our chart above to reflect this requirement.

j1	j2	j3	Command	JUMP WHEN:
0	0	0	Null	$\text{ng} \times \text{j1} + \text{zr} \times \text{j2} + \text{pos} \times \text{j3}$
0	0	1	JGT >	$\text{pos} \times \text{j3}$
0	1	0	JEQ = 0	$\text{zr} \times \text{j2}$
0	1	1	JGE \geq	$\text{pos} \times \text{j3} + \text{zr} \times \text{j2}$
1	0	0	JLT <	$\text{ng} \times \text{j1}$
1	0	1	JNE $\neq 0$	$\text{ng} \times \text{j1} + \text{pos} \times \text{j3}$
1	1	0	JLE \leq	$\text{ng} \times \text{j1} + \text{zr} \times \text{j2}$
1	1	1	JUMP	$\text{ng} \times \text{j1} + \text{zr} \times \text{j2} + \text{pos} \times \text{j3}$

For example. If we want to jump when **ALU-out=greater than OR equal to 0**, we set our jump-bits (**j1,j2,j3**) to 011. **j2=1** and **j3=1**, so we need to see if the status of flags **zr** and **ps** matches. If **zr=1** OR **ps=1** then we know that **ALU-out is greater than OR equal to 0**. To see if they match the jump-bits, we just AND each term together: $\text{JUMP} = (\text{zr} \times \text{j2}) + (\text{ps} \times \text{j3})$

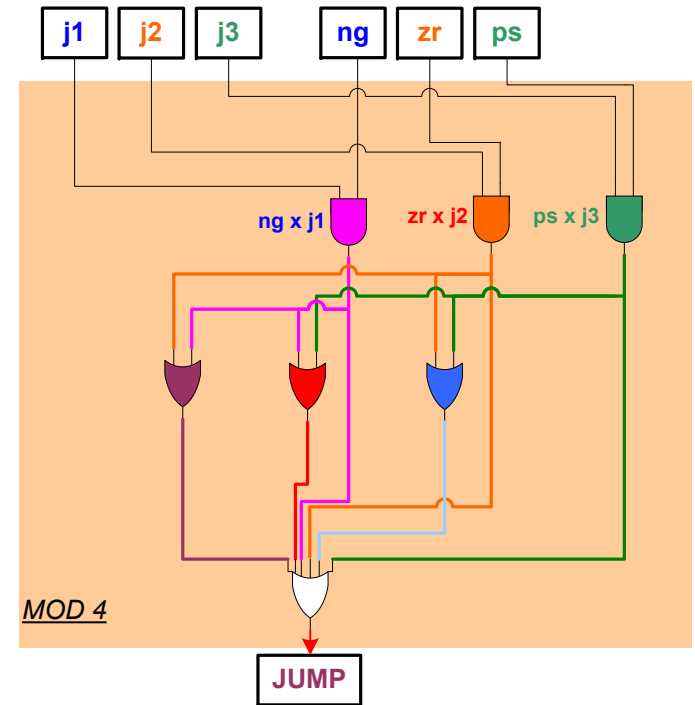
So to review, we have our **jump-bits** indicating what condition we want to jump. Then we have our **status flags**. All we need to do is test each one of 6 possibilities (the first and last one we'll talk about in a minute). If we want to jump on greater than 0, we take **pos** AND **j3**. If that is 1, then we **jump**. But if we want to jump on ALU-out being less than or equal to zero, we use the expression:

$$\text{JUMP} = (\text{ng} \text{ AND } \text{j1}) \text{ OR } (\text{zr} \text{ AND } \text{j2})$$

This works because if **ALU-out** is less than or equal to zero then either **ng** AND **j1** is 1 OR **zr** AND **j2** is 1. That output of 1 indicates that we have a jump.

But notice the first and last type of condition: **Null** and **Jump no matter what**. Most of the time, we do NOT want the CPU to jump after performing some operation. Think back to our multiplication algorithm's repeated addition: $\text{answer} = \text{answer} + a$. We weren't jumping after that calculation. Instead we went on to next step. So for that case, we set our **jump-bits** all to 0. Thus, every single flag/jump bit test will fail and *no jump* will occur. That is what we want. In the same way, if we want a jump no matter what, setting all jump-bits to 1 will ensure that at least 1 of our tests will succeed (**ALU-out** has to be negative, zero or positive) and a jump will occur.

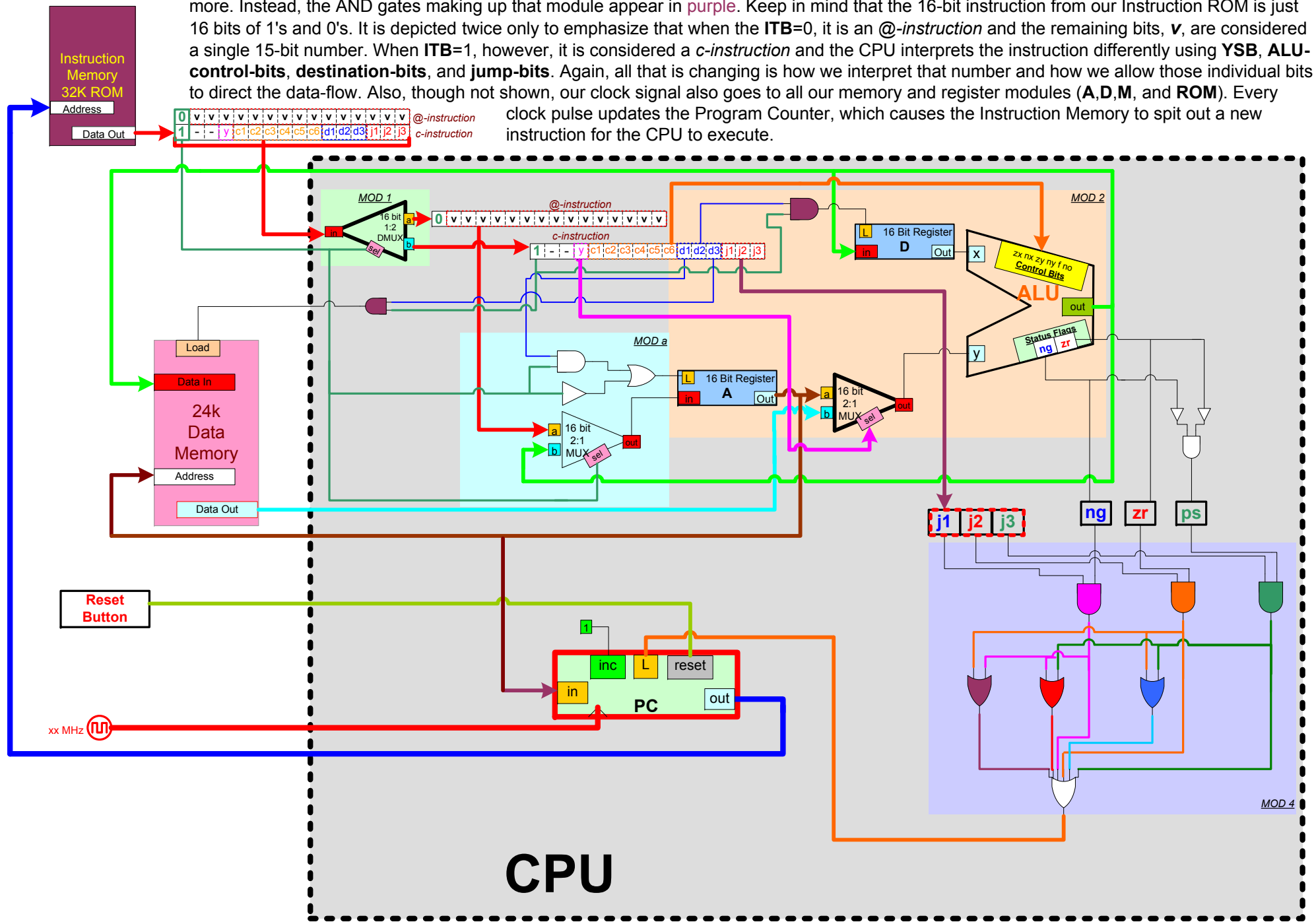
We can use the chart and create our test and jump circuit.



This circuit will take our jump-bits and see if they match our ALU status flags. If so, it will output a **JUMP=1**. Otherwise, **JUMP=0**.

26-CPU- Putting it all together

We are now finished with all the necessary modules and components of our CPU. We can put all of them together and our CPU is complete. A few notes, though. **MOD 3** does not appear labeled because it just consisted of the AND gates that took ITB and d2 and d3 as inputs. Displaying a colored box would only clutter up the diagram more. Instead, the AND gates making up that module appear in purple. Keep in mind that the 16-bit instruction from our Instruction ROM is just 16 bits of 1's and 0's. It is depicted twice only to emphasize that when the **ITB=0**, it is an **@-instruction** and the remaining bits, **v**, are considered a single 15-bit number. When **ITB=1**, however, it is considered a **c-instruction** and the CPU interprets the instruction differently using **YSB**, **ALU-control-bits**, **destination-bits**, and **jump-bits**. Again, all that is changing is how we interpret that number and how we allow those individual bits to direct the data-flow. Also, though not shown, our clock signal also goes to all our memory and register modules (**A,D,M**, and **ROM**). Every clock pulse updates the Program Counter, which causes the Instruction Memory to spit out a new instruction for the CPU to execute.



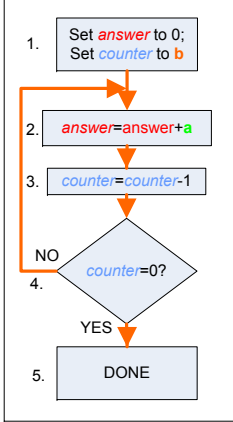
27-Making our first program

Copyright 2013. Ian Ohlander. All Rights Reserved

Now it's time to actually put this CPU to work. We'll return to our multiplication algorithm and proceed to convert it into code that will run on this CPU. First, we need to break down, in discrete steps, exactly how our CPU should carry out this algorithm in steps it can do. As we do this, you'll notice that we'll use the **D** register as a temporary storage, since **A** is going to be in constant use as a way to load data, memory addresses, and Program Counter addresses into our CPU.

(The // means a comment.)

Remember we have 2 numbers to multiply, **a** and **b**, an **answer** and **counter**



1. answer=0
counter=b
2. answer= answer+a
3. counter= counter-1
4. counter=0?
5. done

A=answer
M[A]=0

A=b
D=M[A] // D=b
A=counter
M[A]=D // counter=b

A=a
D=M[A] // D=a
A=answer
M[A]=M[A]+D // answer=answer+a

A=counter
M[A]=M[A]-1 // counter=counter-1

D=M[A] // D=counter
A=Step-2
D; JNE // If D≠0 then Jump to Step-2

A=Step-5
0; JMP // Jump to Step-5

We need to load in 2 pieces of data:
 1) the value of **b**
 2) The address of our **counter**.
 So we are going to be using **A** twice.
 We use **D** as a temporary storage and put **b** in **D**.
 Then we get the address to **counter** and put **b**'s value (stored in **D**) there.

Notice we again use D as a temporary storage for a. Then we add answer plus a and store that in answer.

The CPU cannot stop, so we put it in an infinite loop after we are done. Our answer is in answer.

Here, we put **counter** value (whose address is already in **A**) in **D**. We then load **A** with the program step we will need to jump to (in this case **Step-2**). Then we load **D** into the ALU. If **D=0**, then the program continues. If **D** is NOT 0, then we need to jump. The CPU needs to **Jump** only when **counter (D)** is **Not Equal** to 0 (**JNE**). Otherwise, it keeps going.

We'll number each line of our code. We'll also label our **Step-2** and **Step-5** as **LOOP** and **END**. In our code (lines 13 & 15) we then will load **A** register with the line address where we see **LOOP** and **END** (6 & 15). We also can set our memory locations. **a** will be at address 0, **b** will be at address 1, **answer** will be at address 2, and **counter** will be stored at address 16. Now we have a completed multiplication program written in human readable machine instructions. All the memory and instruction locations have been resolved into actual addresses. Now that we have our program, we need to convert it into the machine code our CPU can recognize. We will need the following information to do the conversion: our **instruction format** (for both @- or c-instruction) our **ALU control bit table**.

```

0 A=2
1 M[A]=0
2 A=1
3 D=M[A]
4 A=16
5 M[A]=D
6 (LOOP) A=0
7 D=M[A]
8 A=2
9 M[A]=M[A]+D
10 A=16
11 M[A]=M[A]-1
12 D=M[A]
13 A=6
14 D; JNE
15 (END) A=15
16 0;JMP
    
```

FORMAT 0 v v v v v v v v v v v v v v v v @-instruction

ALU Control Bits		zx	nx	zy	ny	f	no	f(x,y)=
zero x	NOT x	zero y	NOT y	AND/add	NOT output	output function		
1	0	1	0	1	0	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	-1
0	0	1	1	0	0	0	0	x
1	1	0	0	0	0	0	0	y
0	0	1	1	0	1	0	1	NOT x
1	1	0	0	0	1	0	1	NOT y
0	0	1	1	1	1	1	1	-x
1	1	0	1	1	1	1	1	-y
0	0	1	1	1	0	1	0	x+1
1	1	0	1	1	0	1	0	y+1
0	0	1	1	0	0	1	0	x-1
1	1	0	0	1	0	1	0	y-1
0	0	0	0	0	1	0	0	x+y
0	1	0	0	1	1	1	1	x-y
0	0	0	1	1	1	1	1	y-x
0	0	0	0	0	0	0	0	x AND y
0	1	0	1	0	1	1	1	x OR y

If we take instruction 0, **A=2**, we see that it is merely an @-instruction. So we set **ITB=0** and then **v=2** (binary 10).

0000 0000 0000 0010

Instruction 1 is **M[A]=0**. That is a c-instruction so **ITB=1**. Our **ALU control bits** need to output a 0, which they can do with **101010**. Our **YSB** doesn't matter since we are outputting 0, so we'll leave it at 0. Our **destination bits** are for **M[A]** only, so **d3=1**. And we don't jump, so all the **jump bits** are 0. So that means.

1110 1010 1000 1000

Instruction 2 is pretty straightforward, so let's jump to 9: **M[A]=M[A]+D**. It is a c-instruction so **ITB=1**. Our y source is M, so **YSB=1**. We are doing a **x+y** calculation, so **ALU bits** are: **000010**. We are writing to M, so **d3=1**. And our **jump-bits** are 0. Therefore:

1111 0000 1000 1000

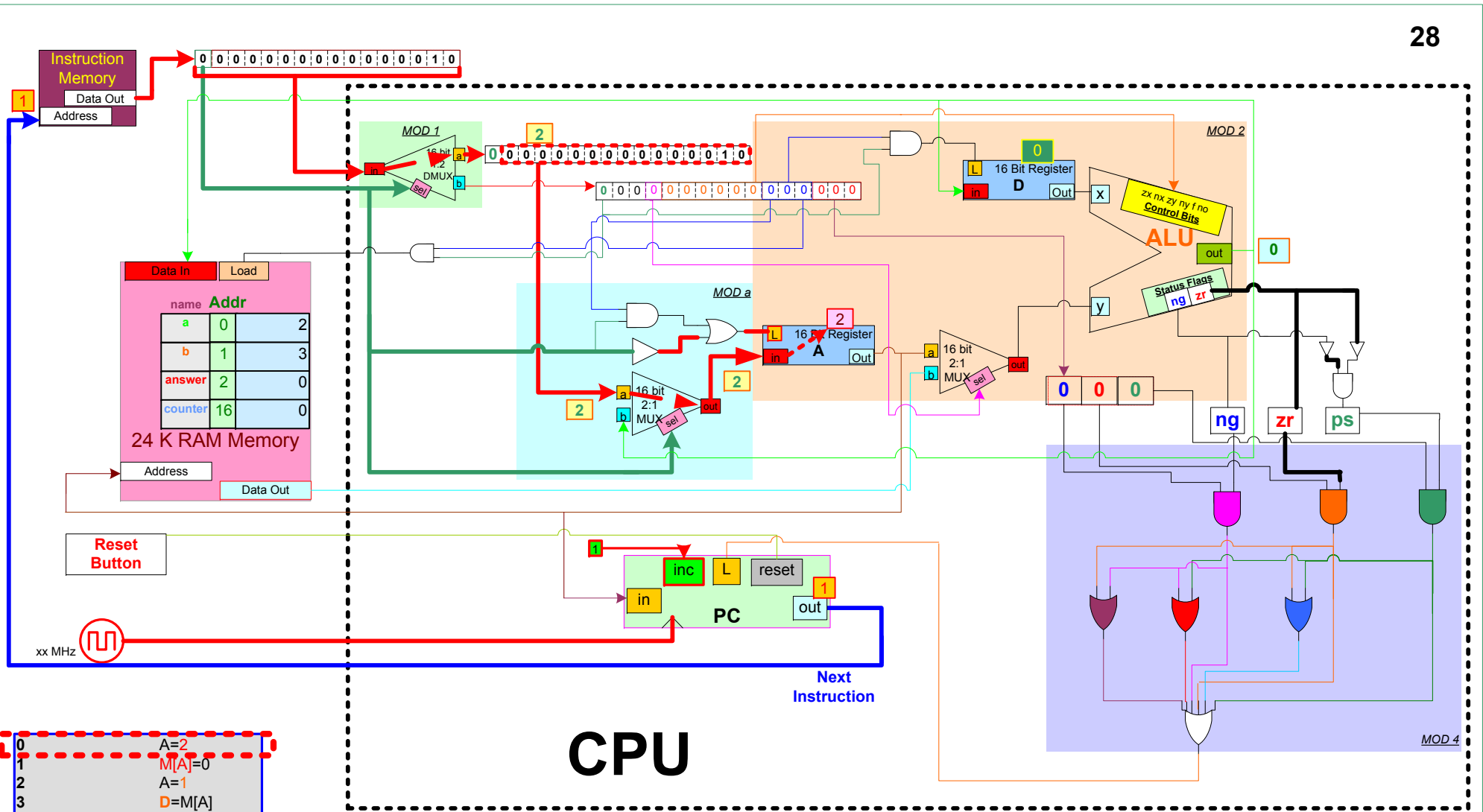
Let's look at an instruction with a jump, 14: **D; JNE**. This is a c-instruction, so **ITB=1**. We don't care about the **YSB**, so we'll leave it a 0. Our calculation requires we look at **D**, which is on the **x-input**. To output x, we need **ALU code**: **001100**. We aren't writing to memory, so **destination bits** are 0. But we do jump ONLY when the output is NOT 0. So we set **jump bits** to **101** (**neg** or **pos**, but not **zero**.) Thus:

1110 0011 0000 0101

That's tedious to do, however, so we can use a program (called an assembler) running on another computer, to do it for us. Here it is all done.

This machine language code will multiply any 2 numbers we put in RAM memory 0 and 1 and will place the result in RAM memory 2.

0	0000000000000010	9	1111000100010000
1	1110101010001000	10	0000000000001000
2	0000000000000001	11	1111110010001000
3	1111110000010000	12	1111110000010000
4	000000000010000	13	000000000000110
5	1110001100001000	14	1110001100000100
6	000000000000000	15	000000000001111
7	1111110000010000	16	1110101010000111
8	000000000000010		



CPU

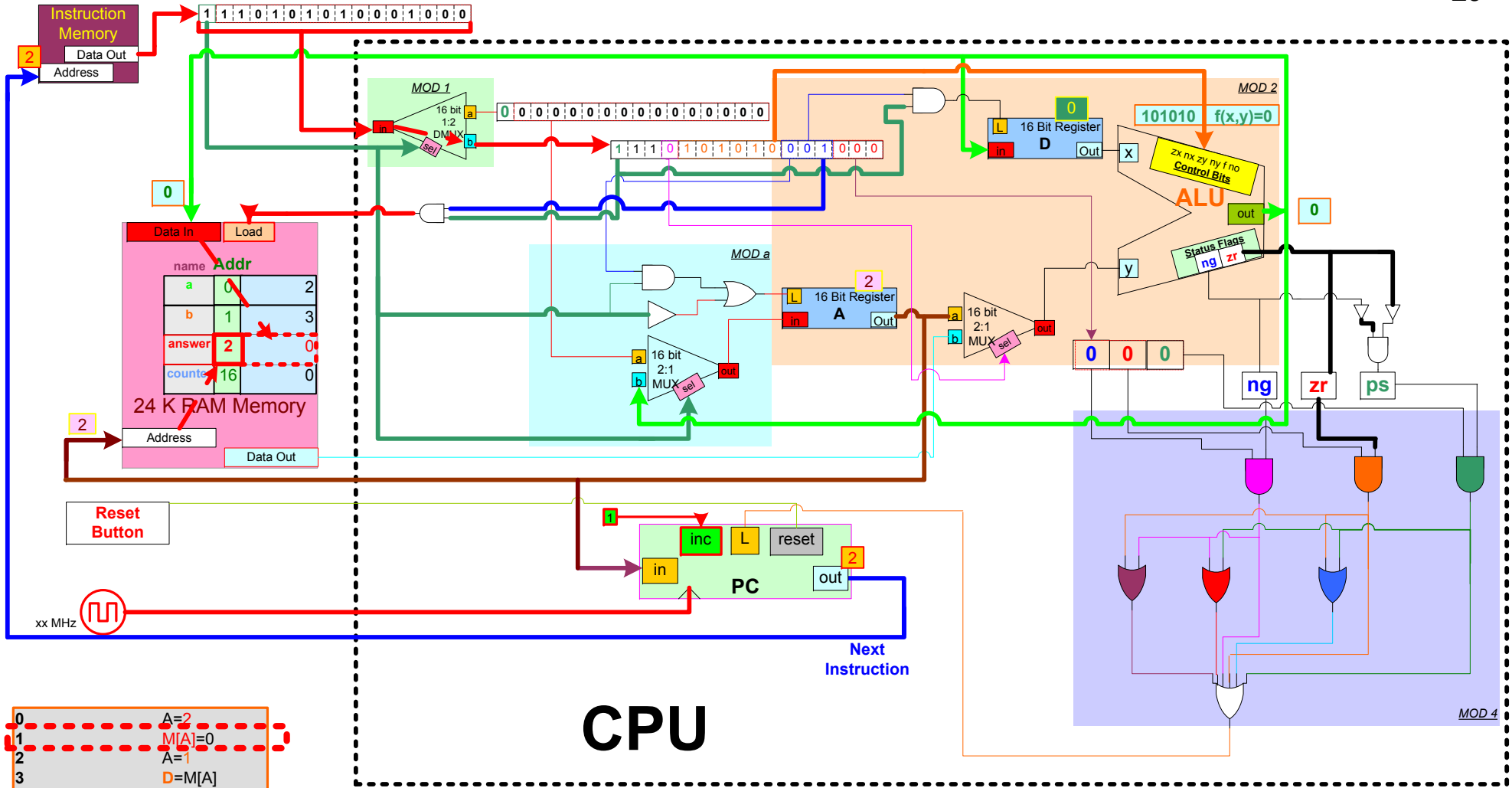
```

0 A=2
1 M[A]=0
2 A=1
3 D=M[A]
4 A=16
5 M[A]=D
6 (LOOP) A=0
7 D=M[A]
8 A=2
9 M[A]=M[A]+D
10 A=16
11 M[A]=M[A]-1
12 D=M[A]
13 A=6
14 D; JNE
15 (END) A=15
16 0; JMP
    
```

0	0000000000000010	9	1111000010001000
1	1110101010001000	10	0000000000001000
2	0000000000000001	11	1111110010001000
3	1111110000001000	12	1111110000001000
4	0000000000001000	13	0000000000000110
5	1110001100001000	14	1110001100000101
6	0000000000000000	15	0000000000001111
7	1111110000001000	16	1110101010000111
8	0000000000000010		

Let's see this CPU and program in action. We are going to look at a few of the instructions and see how things run. Imagine that, similar to how we'd take a cartridge and plug it into a Nintendo or Atari system, we plugged in a ROM with our multiplication program on it. We'll begin with address 0 in our Instruction Memory ROM. The numbers we want to multiply, **a** and **b** are loaded into memory address 0 and 1. The **answer** will be in memory address 2.

ROM INSTRUCTION MEMORY	Addr 16-Bit Instructions	Human Readable
0	0000000000000010	A=2 @-instruction



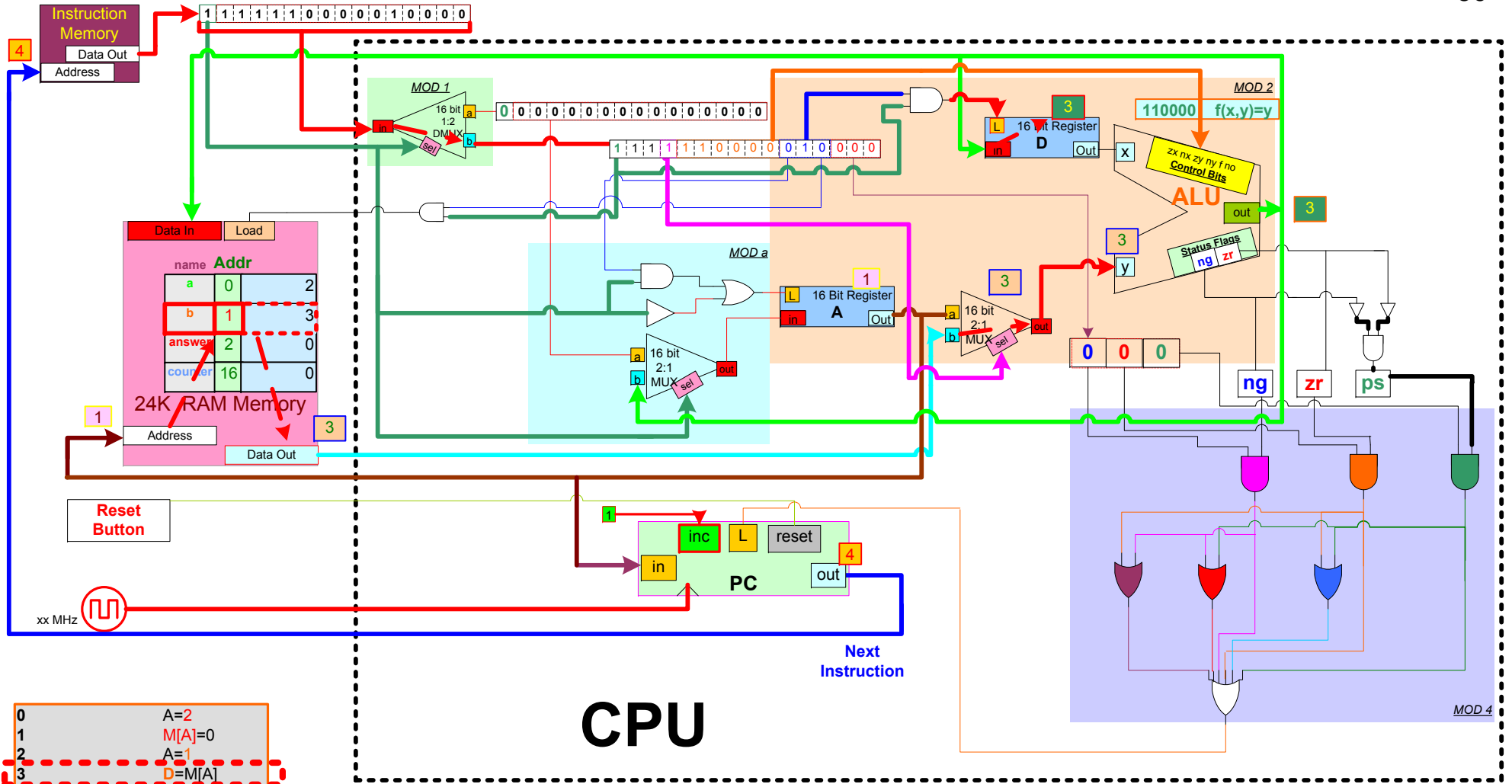
CPU

```

0 A=2
1 M[A]=0
2 A=1
3 D=M[A]
4 A=16
5 M[A]=D
6 (LOOP) A=0
7 D=M[A]
8 A=2
9 M[A]=M[A]+D
10 A=16
11 M[A]=M[A]-1
12 D=M[A]
13 A=6
14 D; JNE
15 (END) A=15
16 0; JMP
    
```

0	0000000000000010	9	1111000010001000
1	1110101010001000	10	0000000000010000
2	0000000000000001	11	1111110010001000
3	1111110000010000	12	1111110000010000
4	0000000000010000	13	0000000000000110
5	1110001100001000	14	1110001100000101
6	0000000000000000	15	0000000000001111
7	1111110000010000	16	1110101010000111
8	0000000000000010		

ROM INSTRUCTION MEMORY	Addr 16-Bit Instructions	Human Readable
1	1110101010001000	M[A]=0 <i>c-instruction</i>



CPU

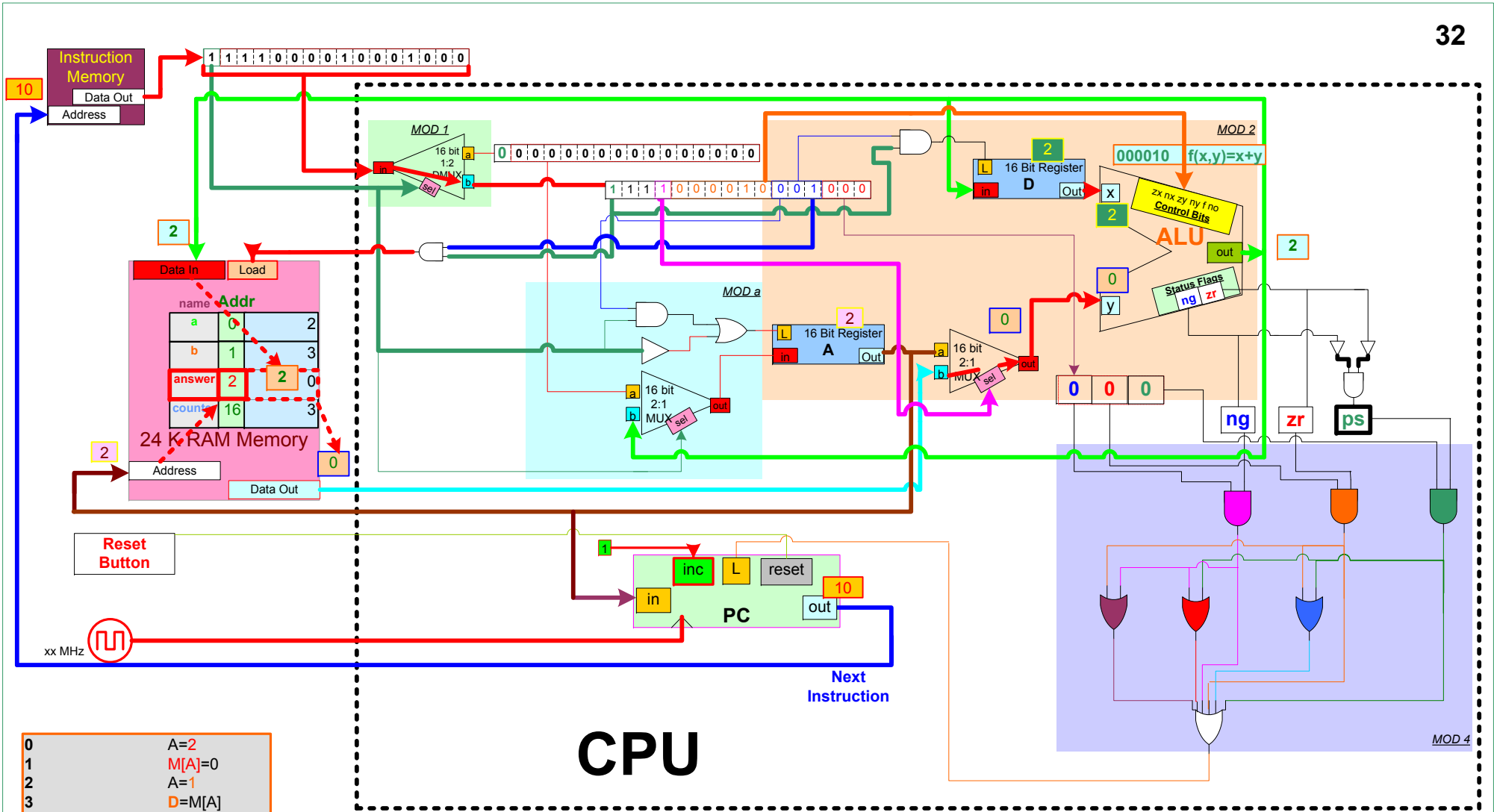
```

0 A=2
1 M[A]=0
2 A=1
3 D=M[A]
4 A=16
5 M[A]=D
6 (LOOP) A=0
7 D=M[A]
8 A=2
9 M[A]=M[A]+D
10 A=16
11 M[A]=M[A]-1
12 D=M[A]
13 A=6
14 D; JNE
15 (END) A=15
16 0; JMP
    
```

0	000000000000010	9	1111000010001000
1	1110101010001000	10	0000000000001000
2	0000000000000001	11	1111110010001000
3	1111110000001000	12	1111110000001000
4	0000000000001000	13	0000000000000110
5	1110001100001000	14	1110001100000101
6	0000000000000000	15	0000000000001111
7	1111110000001000	16	1110101010000111
8	000000000000010		

Instruction 2 is identical to Instruction 0. It simply loads **A** register with the Memory Address of **b**. Let's assume we've done that. **A** register contains the Memory Address of **b**, 1. Now let's go on to Instruction 3, which places the value **b** in **D** register.

ROM INSTRUCTION MEMORY	Addr 16-Bit Instructions	Human Readable
3	1111110000010000	D=M[A] <i>c-instruction</i>



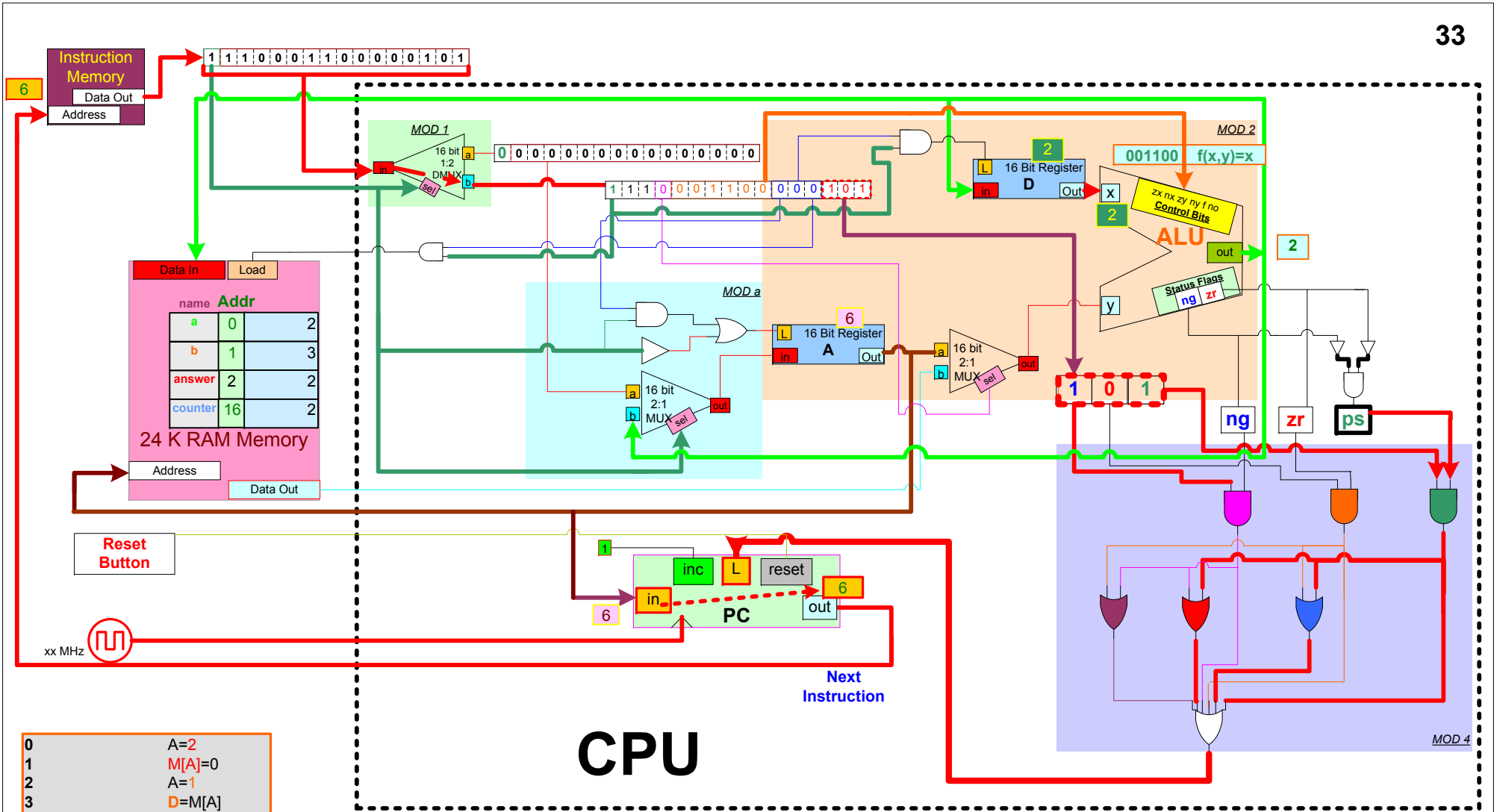
CPU

Let's jump to instruction 9. We will assume that instructions 6-8 have been completed. D contains the value of Memory Location 0, which is **a**. A now contains the address of our **answer**, 2. Our next instruction will add what is in **answer** to **a** and put the result back in **answer**.

0	A=2
1	M[A]=0
2	A=1
3	D=M[A]
4	A=16
5	M[A]=D
6	(LOOP) A=0
7	D=M[A]
8	A=2
9	M[A]=M[A]+D
10	A=16
11	M[A]=M[A]-1
12	D=M[A]
13	A=6
14	D; JNE
15	(END) A=15
16	0; JMP

0	000000000000010	9	1111000010001000
1	1110101010001000	10	0000000000001000
2	0000000000000001	11	1111110010001000
3	1111110000010000	12	1111110000010000
4	0000000000001000	13	0000000000000110
5	1110001100001000	14	1110001100000101
6	0000000000000000	15	0000000000001111
7	1111110000010000	16	1110101010000111
8	0000000000000010		

ROM INSTRUCTION MEMORY	Addr 16-Bit Instructions	Human Readable
9	1111000010001000	M[A]=M[A]+D c-instruction



CPU

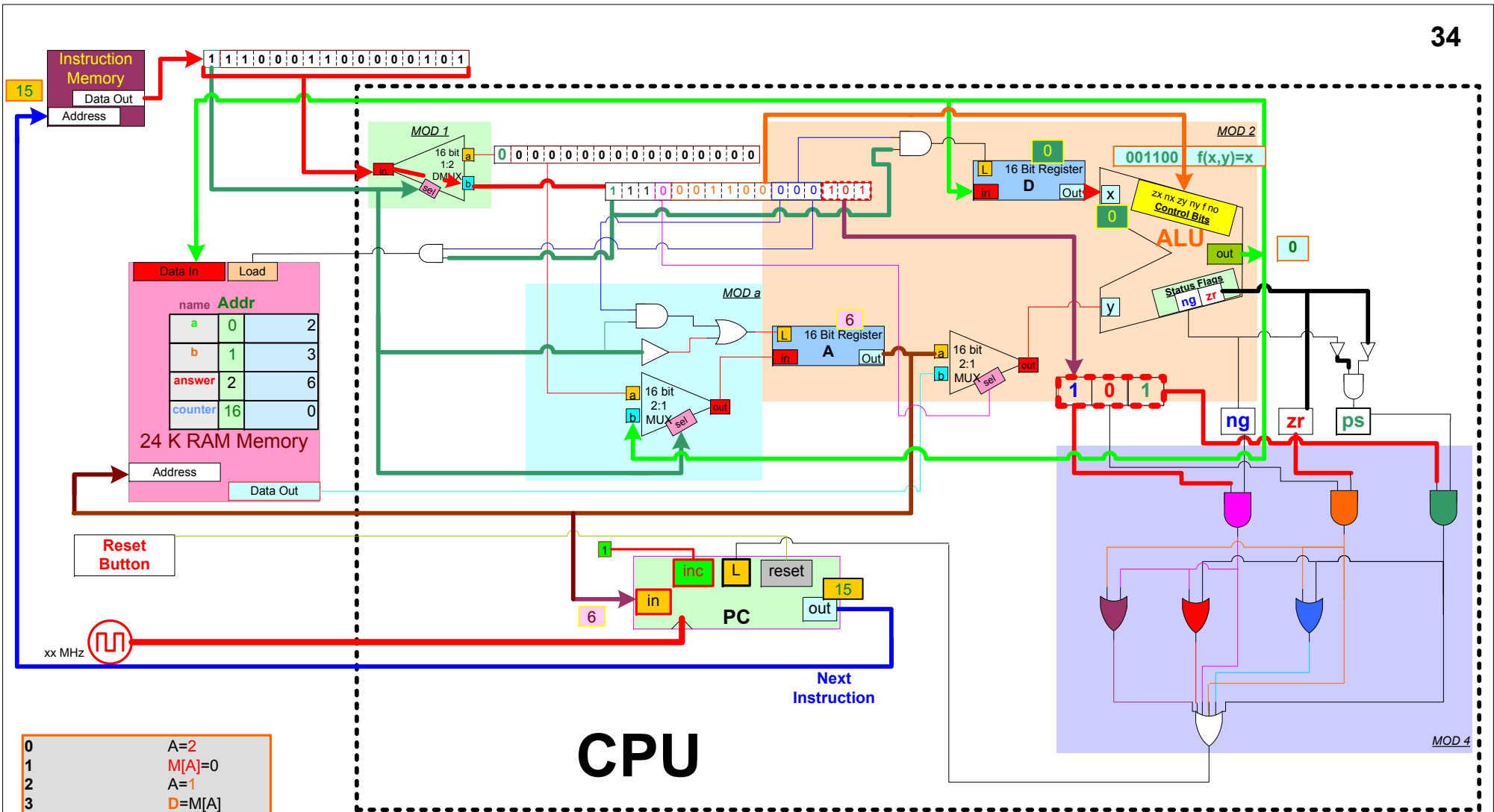
Let's jump to instruction 14, so we can see a jump occur. We assume instructions 11-13 have completed. **counter** (Memory Address 16) now contains 2. **D** contains of the contents of **counter**, also a 2. And we have loaded **A** with the **Instruction Address** we will need jump to if the jump conditions are met.

```

0 A=2
1 M[A]=0
2 A=1
3 D=M[A]
4 A=16
5 M[A]=D
6 (LOOP) A=0
7 D=M[A]
8 A=2
9 M[A]=M[A]+D
10 A=16
11 M[A]=M[A]-1
12 D=M[A]
13 A=6
14 D; JNE
15 (END) A=15
16 O; JMP
    
```

0	000000000000010	9	1111000010001000
1	1110101010001000	10	0000000000001000
2	0000000000000001	11	1111110010001000
3	1111110000001000	12	1111110000001000
4	0000000000001000	13	000000000000110
5	1110001100001000	14	1110001100000101
6	0000000000000000	15	0000000000001111
7	1111110000001000	16	1110101010000111
8	000000000000010		

ROM INSTRUCTION MEMORY	Addr 16-Bit Instructions	Human Readable
14	1110001100000101	D; JNE <i>c-instruction</i>



CPU

Let's again look at instruction 14. We assume instructions 11-13 have completed. Let's assume we have looped a few times. **answer** contains a 6 and **counter** (Memory Address 16) now contains 0. **D** contains the contents of **counter**, also a 0. We thus should expect that our jump conditions will fail (which it does. PC just increments 14 to 15 and that is the next instruction to execute.) **A** is loaded with the **Instruction Address** we would usually need.

```

0 A=2
1 M[A]=0
2 A=1
3 D=M[A]
4 A=16
5 M[A]=D
6 (LOOP) A=0
7 D=M[A]
8 A=2
9 M[A]=M[A]+D
10 A=16
11 M[A]=M[A]-1
12 D=M[A]
13 A=6
14 D; JNE
15 (END) A=15
16 O; JMP
    
```

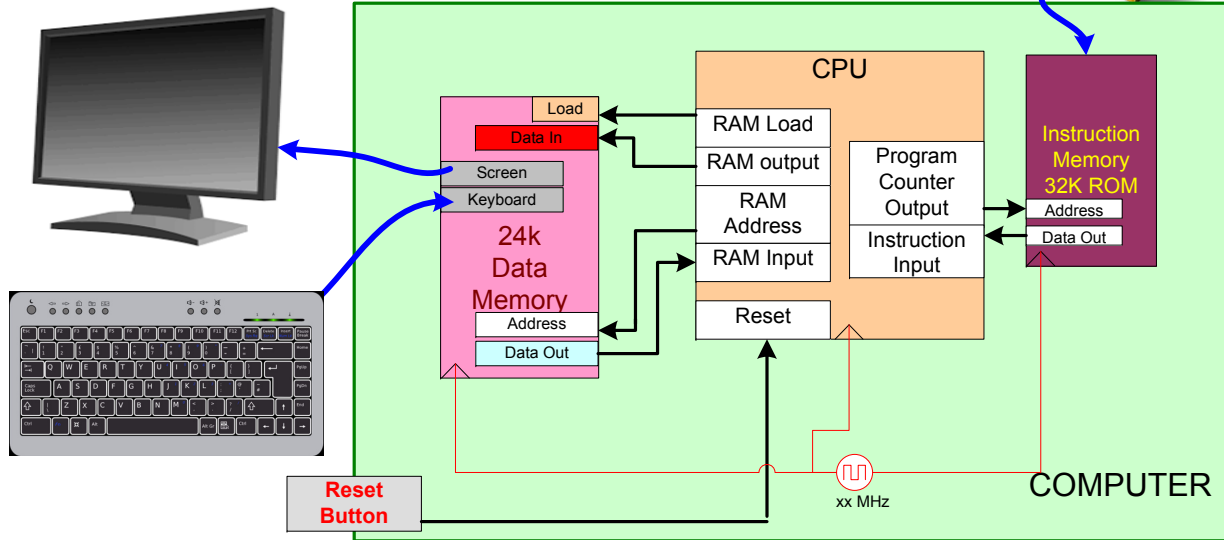
0	000000000000010	9	1111000100001000
1	1110101010001000	10	0000000000010000
2	0000000000000001	11	1111110010001000
3	1111110000010000	12	1111110000010000
4	0000000000010000	13	0000000000001110
5	1110001100001000	14	1110001100000101
6	0000000000000000	15	0000000000001111
7	1111110000010000	16	1110101010000111
8	000000000000010		

ROM INSTRUCTION MEMORY	Addr	16-Bit Instructions	Human Readable
	14	1110001100000101	D; JNE <i>c-instruction</i>

35-Wrapping it all together

Copyright 2013. Ian Ohlander. All Rights Reserved

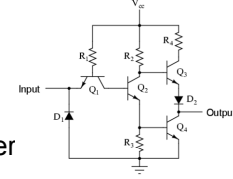
So now that we have our CPU built, we can treat it as a single unit. We will label our input/output lines for a reset button, the RAM Memory Address (for reading and writing from/to specific locations), RAM Memory Input, RAM Memory Output, RAM Memory Load, Instruction Input, Program Counter Output, and a Clock. We can then put our entire computer together, as well as a keyboard, screen, and ROM Cartridge (Instruction Memory).



At this point we can step back and observe a couple of things about our computer.

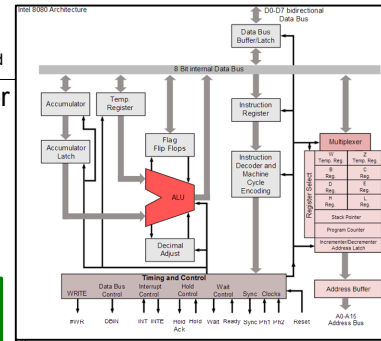
1) It is theoretical (which is how we wanted it. We wanted to focus on the logical design itself). To physically build this, you'd have to transform the schematics we've designed into actual electrical engineering plans. To illustrate the difference, here is an image of what a NOT gate schematic actually looks like implemented using transistors and resistors. (Physical implementation will also change component design to deal with other issues.)

Practical inverter (NOT) circuit



2) Compared to modern computers, it is slow, has little memory, and is not very easy to use. Of course, ease of use depends on the software written for it. Obviously, we would not want to have to interact with this machine by poking code into binary memory. We'd like something more user friendly. Modern computers are very easy to use because software has been written to take care of all the behind the scenes requirements. The software is called an Operating System (OS)- a large group of applications that take care of all the low level stuff. When you type in your word processor (WP), the CPU receives an *interrupt* notification. The OS reads the keyboard memory and gets your letter. It passes that letter on to the WP, which sees it's an actual keystroke (as opposed to a control command like ctrl-P) and tells the OS to draw it. The OS receives that request and looks up the image bit-code for that letter and pokes those bits into the appropriate screen memory. Other code in your word processor will spell check by comparison with an in-memory dictionary. If not, it places a squiggly red line underneath the word by asking the OS to draw it at a specific place on the screen. Clearly, to do all that will require a lot of speed, memory and software working together.

Our computer, despite have 24K memory and the ability to process 16-bits of information at a time, is very similar to the computers of the mid 70's. The computer that really jump-started personal computing was the Altair 8800. It could only handle 8-bits of information at a time and had 1K, 4K or 8K memory.



The 8800 used an Intel 8080 CPU chip (rough schematic *left*.) Its architecture is far more complicated than ours. We didn't do anything to optimize performance or data-flow. Normally, after each component is designed, it is studied and

redesigned to run faster and/or with fewer parts. Our 16-Bit Adder, would be replaced with a 16-Bit carry *look ahead* adder. Data latches have some flaws and would be replaced by data flip-flops. The existing memory banks have some unnecessary components, as does the ALU. And so on.

The Altair 8080 could be bought as a kit or fully assembled. Programs were entered using the switches on the front of the case and output was "read" using the LEDs. Programs and data could also be loaded using paper, or punch tape, if you could afford the reader. Otherwise you had to reenter the program yourself each time it started.



The Apple I, designed by Steve Wozniak, was much more user friendly than the Altair. From there, he engineered the Apple II which made a personal computer appealing to a

much larger audience. A tape recorder, and later a disk drive, allowed programs and personal data to be stored. It and the Commodore 64 were most responsible for setting off the personal computing revolution- and our modern world!



But at its core, all computers use the same principles as ours. Their advanced capabilities come from more memory, faster processors, better architecture and an advanced operating system.

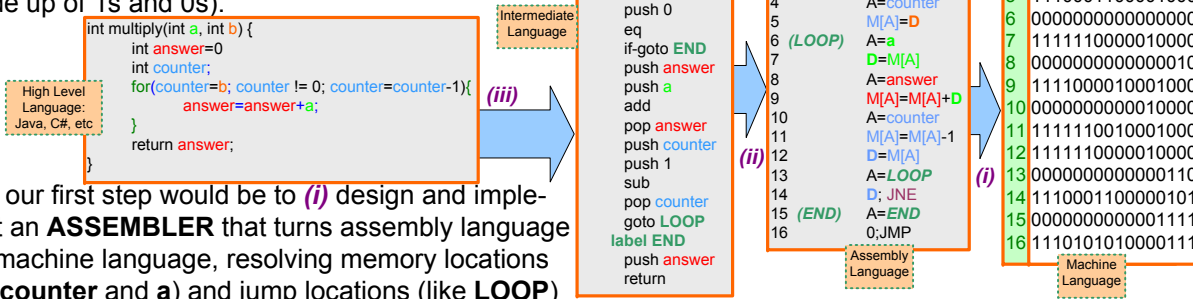
36-Where do we go from here?

Copyright 2013. Ian Ohlander. All Rights Reserved

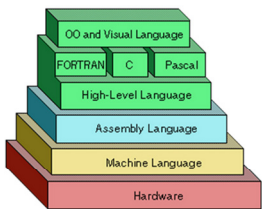
At this point, we are done. Our computer is complete. From here on out, any further development will be done by writing software for the CPU. But if you look at our multiplication program, you recall it wasn't very easy to write. We had to *directly* manipulate the registers and specific memory locations. It meant we had to know the CPU's architecture. If we designed all the necessary programs (drawing text on screen, responding to keyboard input, etc) that way, we'd have to make sure that they played nice together- not overwriting each other or any memory storage they each might be using. It would take a lot of work.

Computer designers therefore came up with "Higher Level" languages that allow the programmer to concentrate on the task at hand (creating the programs) without having to worry about the hardware details. In the same way that we treated our logic gates as black boxes that performed a function (however they worked), these high level languages allow the programmer to not worry about how the CPU will actually carry out its instructions. He does not have to worry about the memory locations of **counter** or **answer**; he doesn't need to make sure **A** is loaded with a jump address before doing a test, or what that address might be.

Instead, the programmer just writes his program in language more akin to human thought (with its own grammar and vocabulary) and a compiler program translates that into an intermediate language and then into assembly language. Then an assembler translates that into the CPU's machine language (made up of 1s and 0s).



So our first step would be to (i) design and implement an **ASSEMBLER** that turns assembly language into machine language, resolving memory locations (like **counter** and **a**) and jump locations (like **LOOP**)

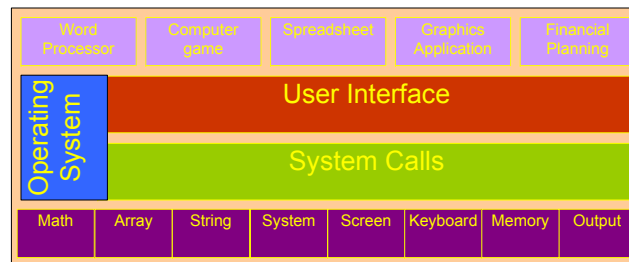


into actual numbers (which we did by hand earlier.) We'd write it in a high-level language on another computer. (ii) Then we would design a **COMPILER** to convert an intermediate language into that assembly language. (iii) Then we'd expand our compiler to convert the higher level language into that intermediate language. Now we can write software more easily. We would proceed to write our operating system in that easier higher-level language. Libraries of programs would need to be written to take care of things like printing characters to the screen,

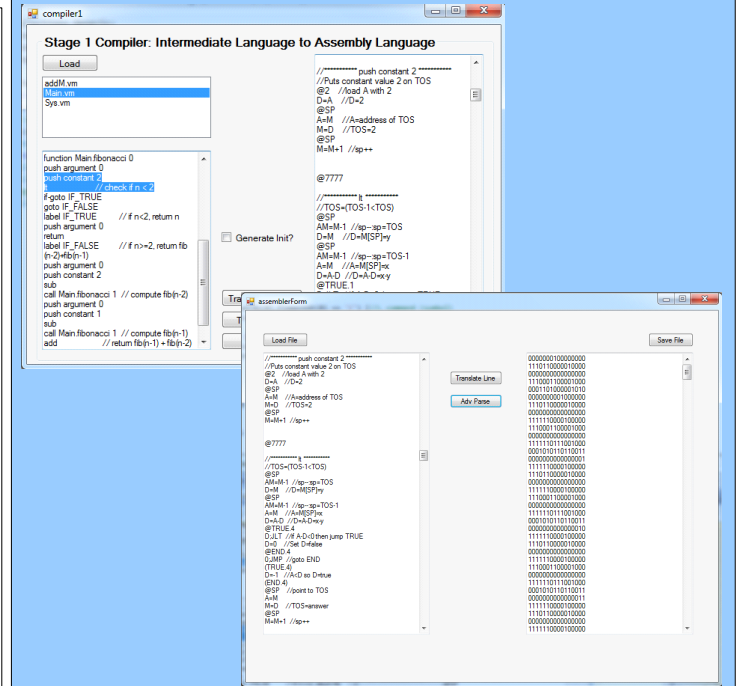
drawing primitive shapes (points, lines, circles) to any coordinate on the screen, perform mathematical functions, manipulate text, manage memory arrays, read input from the keyboard, and handle the execution of programs and any errors they generate.

But if we do all that, then we will have an environment that will let us create software to run on our computer.

So there it is, our completed computer. It's been a long journey, but it was worth it to build it with our own hands.



Screen shots of the Compiler and Assembler



Sample graphics program written for our CPU

