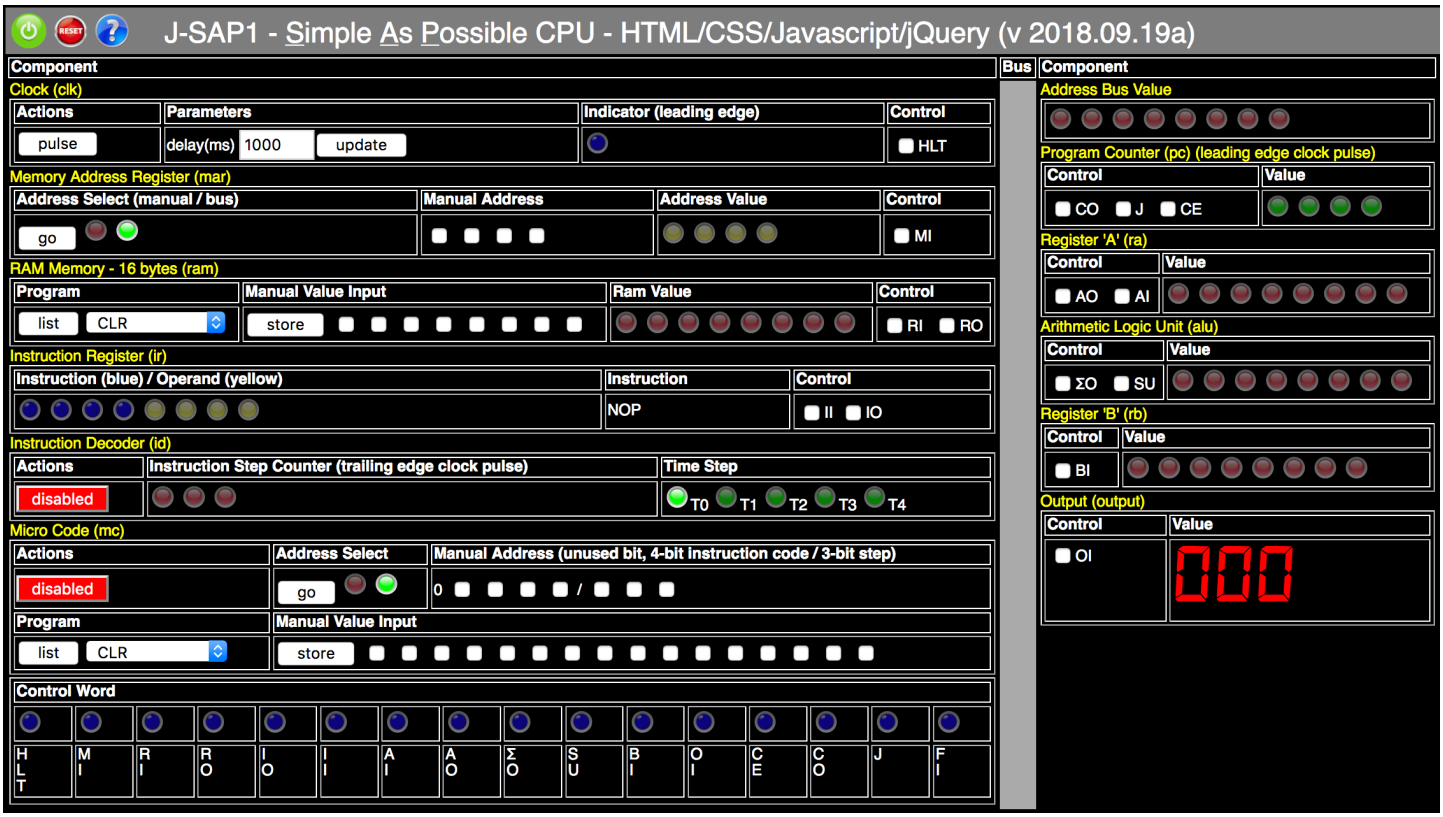
12bit.com

Presents…

J-SAP1 CPU – JavaScript: Simple As Possible CPU



Learn the basics of how a CPU works

Table of Contents

Learning Objectives

- Learn the main components of the CPU by interacting with the J-SAP1 simulator
- Understand that a CPU is type of automatic calculator
- Use J-SAP1 to manually calculate a simple addition problem
- Break this addition problem down into steps
- Document these steps and develop an algorithm
- Learn about Microcode, which is the internal language you never get to see.
- Develop 5 Assembler instructions to create re-usable steps
- Learn about the Fetch Cycle and how to integrate that in the Microcode
- Develop the final version of the Microcode and Program
- Run this program on the simulator and get 42!!

Why did I make this? The Inspiration

I watched a series of YouTube videos by Ben Eater where he builds a breadboard CPU based on the SAP-1 model. He does such a fantastic job explaining everything and I was amazed at how much I learned just by watching. Even though the expense of building it is not that bad, it takes up a lot of room. With all the other projects that are in flight in my space, I just did not want to do something that took up too much room. So, I decided to build an emulation of what I had seen.

In those videos, Ben builds a CPU with individual components on a series of breadboards. Along the way, you gain an understanding of each part of the CPU, what is does and how it works with all the other parts. In creating a JavaScript version, I gained a deeper understanding by applying what I had seen. I did not have to emulate the individual chips or components, but I did model each high-level component he built into a separate JavaScript file.

I would encourage you to check out his YouTube channel and watch all his videos. This document will help you use this emulator standalone, but it is also my hope that you can use this simulator in conjunction with his videos if you do not have the resources to build a real version or maybe you don't want to. Being able to be an 'active' learner (by using the emulator to play along) will benefit you as you will gain a deeper understanding of the material because you are doing something with the information you given.

https://www.youtube.com/channel/UCS0N5baNlQWJCUrhCEo8WlA

https://eater.net

What Is J-SAP1?

Even the first 4-bit CPU (Intel 4004) released commercially back in 1971 was quite sophisticated. So even starting there is probably too much. Fortunately, there is a simplified set of concepts that we can use to learn CPU basics. This emulation is based on the SAP1 model presented in Albert Paul Malvino's book, Digital Computer Electronics. (Chapter 10)

https://www.amazon.com/Digital-Computer-Electronics-Albert-Malvino/dp/0070399018/

SAP1 is a minimum set of requirements that has just enough functionality to be considered a 'computer' or 'CPU'. It can only add and subtract and do one type of logical decision (This is actually SAP2 functionality, but Ben integrates it in the end to

make his CPU ‘Turing Complete’. (SAP1 lacks conditional branching) It is bare minimum to facilitate learning. So, it is complete enough for our purposes.

https://en.wikipedia.org/wiki/Turing_completeness

The ‘J’ stands for JavaScript. This is implemented in JavaScript so the simulator can be available to just about anyone on the web with only a web browser. It is a bit too UI intensive for phone size screens (you could zoom and pan), but it should work in just about any modern browser that supports jQuery. There is even a way to ‘cache’ this web application on your device so you can access it off line much like a native app. See appendix for that info.

There is an SAP2 and SAP3 as well. They expand on these basic concepts and each add additional complexity until you reach a fully functional CPU. Maybe that topic can be a part II to this learning exercise.

The humble pocket calculator

Let's think about adding two numbers together. How about 28 & 14? We can probably do that in our head. It's 42.

Now, do it on a cheap pocket calculator.

1) Type 28
2) Press '+'
3) Type 14
4) Press '='
5) Display now says 42

Now, adding two numbers together in your head, or on this calculator is easy enough, we don't need a computer for that. But what if there were 10 numbers, then a calculator would be nice. What about 100 numbers, we could use a calculator, but would probably make mistakes. What if the numbers we wanted to add (or whatever) were in the millions. I think we would need a computer for that, right? And that is why we have CPUs and computers. We want to automate the calculation of numbers and manipulation of data to save us time, money, and effort. It's all about making things convenient for humans.

We will be 'teaching' our CPU how to perform this simple addition problem. Along the way, we will be learning the main parts of a CPU and how they come together.

This CPU is veeeeeerrrrrrryyyyy slow. It's that way on purpose. We slow it way down so we can understand how it does simple tasks step by step. So, it may seem like it is not very useful. But remember, modern CPUs can perform billions of instructions per second. This speed is where the true magic exists. The super-computers of just a few years ago, now fit in the palms of our hands. What happens inside a CPU step by step is actually very simple and it is those steps on which we are going to focus.

Binary – Computers count differently than we do

This emulation utilizes 'LEDs' which in turn are representing binary numbers. So, you really need to have a good understanding of what binary is. Most likely, you are most familiar with the base 10 number system. Normally, we count from 0 to 9 in the 1's position, then continue by placing a 1 in the 10's position when we count one more than 9. (that is, we 'carry the 1') We continue counting in the ones position from 0 to 9 until we get to twenty. A 2 in the 10's position means we really have 2 sets of ten.

With binary numbers, there is only 0 and 1 so we have to 'carry' much more often. Let's count to 3 in binary:

0 = 0b00
1 = 0b01
2 = 0b10 (oops, already had to carry since 1 is as high as we go)
3 = 0b11

NOTE: the '0b' before the two digits indicates this is a binary number. (this comes from the way JavaScript represents binary numbers. If you look into the code, that is what you will see)

Why are we using binary numbers to represent numbers here? The main reason is that the 0s and 1s represent either on or off. That is, the state of a switch. CPUs are really, when simplified, just switches. So, it is a good way to get your head around what's going on.

In our emulation, we are dealing with binary numbers that are 4-bit and 8-bit. 4-bit numbers can represent value between 0 and 15. 8-bit numbers can represent values between 0 and 255.

We represent a number 4-bit binary number like 0b0000. The 'b', of course, stands for binary. An 8-bit number looks like 0b00000000.

- 0b0000 is 0
- 0b00001 is 12
- 0b00010 is 3
- 0b1111 is 15

Pretty much the same with 8-bit numbers, we can just count higher.

- 0b00000000 is 0
- 0b00000001 is 1
- 0b00000010 is 2
- 0b00000011 is 3
- 0b11111111 is 255

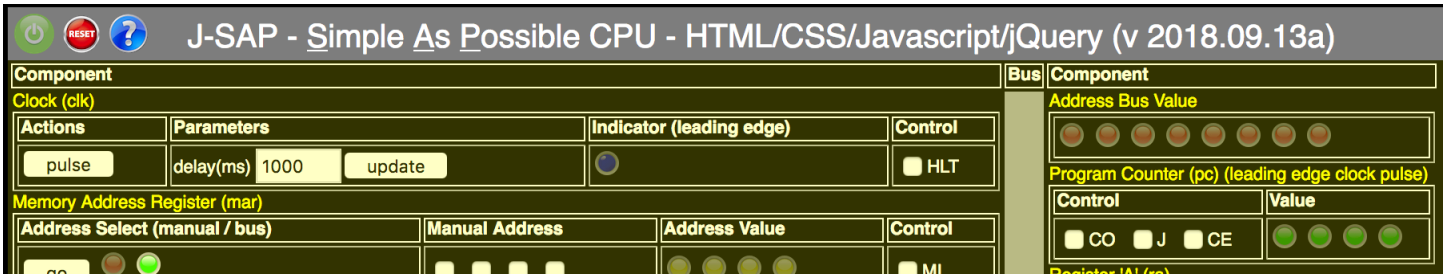
If this still does not make sense, or you want a more complete tutorial, check this out…

https://www.mathsisfun.com/binary-number-system.html

J-SAP1 – Load and power on

Go ahead and access the JavaScript SAP-1 emulator at the following address.

http://learn8bit.com/emulators/js-sap1/index.html



There is a lot going on here. Don't panic, we will look at each component in turn and you will gain an understanding as we explore what makes our basic CPU work. First off, you will notice that there is a yellow tint over the emulator. This indicates that the power is off. Nothing will work until you click the power button.

0) Go ahead and to that now. The yellow tint goes away and you will notice the clock LED (blue) will start flashing once per second. So, let's talk about the first component, the clock.

What time is it? Meet the clock

This clock does not keep the time of day, but is more like a metronome. In music, a metronome keeps the beat and allows one to play music at a steady pace. Our clock, like a real metronome can be sped up or slowed down. You will notice that under parameters, there is a 'delay' (ms) which defines the 'Clock Period'. When starting up, the default value is 1000 ms, which is 1 second.

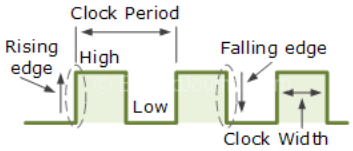


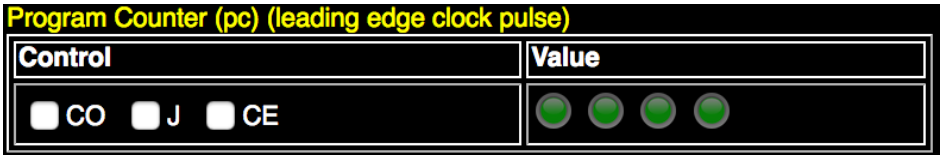
Image credit - https://embedjournal.com/need-for-clock/

In Ben's real electronic version, there are parts of the circuit that perform actions on the 'rising edge' of the clock signal. Other components work on the 'falling' (or 'trailing') edge of the clock 'pulse'. In our simulation when the LED is on, that is the 'leading edge'. When it is off, that is the 'trailing edge'.

The 'HLT' (short for halt, or stop the CPU) checkbox is what we will call a 'control line'. That is, it is a point of control for a particular component. In this case, checking HLT will disable the clock from automatically pulsing (end blinking the LED) every second. When it is halted, we can pulse the clock manually, which lets us step the CPU at our own pace. This feature is very helpful.

0) Go ahead and play with these features and get comfortable adjusting the clock's speed and its ability to be pulsed at your discretion. When you feel comfortable, move on to the next section.

Counting Clocks. Say hello to the program counter.



When a CPU powers up, it wants to immediately start executing code. That's is sole purpose in life. Where does it find the code to execute? In memory. Memory can be a big place (or not in our case), but none-the-less, there needs to be a way for it to know. That's where the Program counter comes in. It keeps track of which instruction in our program we need to execute. This program is stored in RAM (more about that later).

The program counter, PC, is a component that can be connected to the clock. This is accomplished by checking the 'CE' control line. 'CE' is short for Clock Enable. When this is activated, the counter will increment by 1 each time a clock signal (the leading edge of the clock) is received. In our case, you see 4 LEDs. This corresponds to a 4-bit number, which limits us to 16 memory addresses (0-15). When the number reaches 15, it is reset to 0. What does this mean? It means we are limited to having only 16 instructions in our program. That is OK because we do not need to calculate Pi to the 100th decimal. Simple addition is all we need for our understanding and that can be accomplished with what we have.

0) At this point, with the clock pulsing, enable 'CE' and observe watch the PC value increment in binary from 0 to 15 and then back to 0.
1) You might be wondering, what are the other to Control Lines ('CO' and 'J') do. CO stands for Counter Out and J is for Jump. We will talk about Jump much later. However, go ahead check CO and observe that the value associated with the PC is now showing in the bus as well.

So, it's a good time to talk about what the bus does, right?

Is there a seat on this bus?

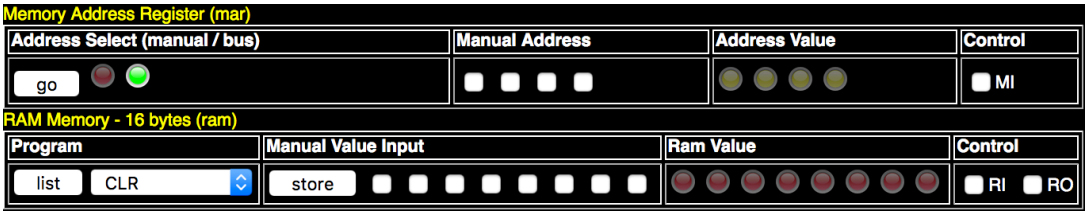
The easiest way to think about the bus is a highway for data. Ours has 8 lanes. (with 8 bits, it can represent values of 0 to 255) Components in our CPU can either read a value in from the bus or output a value to the bus. This is how different components of the CPU communicate with each other and share their values.

Earlier, when learning about the PC, we enabled CO which output the value to the bus. So, what can we do with that? If you recall, we mentioned that our programs were going to be programmed in RAM. RAM is another component that can interact with the bus. Let's move on to that now.

One important fact about the bus is that components Output to the bus immediately when their 'Out' control line is enabled. However, components read values from the bus only on a clock pulse. (leading edge). That fact is important in the transfer of information from one component to another. The clock pulse lets the receiving component know when the value is available for reading. We won't dive into the details now, just keep that in mind for a bit.

Up next, let's talk about RAM.

Clever, but forgetful – Learn about RAM



When discussing RAM, we will actually be talking about two related components. The Memory Address Register MAR and Random Access Memory RAM.

RAM is a place where values are remembered based on an address. Just like each house has a unique address on a street. Each house has a unique set of contents and people (its value).

The MAR, through enabling the MI (Memory In), takes the value on the bus and uses it to look up a value in RAM. Since RAM is empty now, only 0 values are stored at each address, so you will not see anything interesting.

0) Go ahead and check MI. You will see the Address Value will have the same value as the bus. So now, the PC counter is driving the MAR. And the MAR is telling RAM which value to display.

You should notice that there is an 'Address Select' button. This lets you manually choose a RAM address by specifying one in the 'Manual Address'. This is for useful in programming the RAM, which we will discuss shortly.

The RAM component is on the bottom of the picture below. It has 4 parts. The first is the program area which lets you select built in programs, we will not be using this feature yet. The 'Manual Value Input' allows to specify a value for the address from the MAR. The 'store' button commits a value you have specified with the check boxes. There are 8 check boxes which correspond to the 8 bits (1 byte) that each memory location can store. The 'RAM Value' indicates the value within the address. The RAM has two Control Lines; RI – RAM In (get and store from the bus) and RO – RAM Out (send value to bus based on the MAR address)

1) If the clock is still enabled, stop it by checking HLT. Let's program RAM by pressing 'go' on the 'Address Select', this will allow us to specify the RAM address. Ensure all checkboxes are NOT checked. This means we have specified address 0. In RAM's Manual Value Input area, check the boxes to correspond to the binary number 0b00011100 which is the number 28. To do this, check the boxes which corresponds with the bits that have a value of 1. (1 means 'on') This indicates we want to store a 28 in memory address 0. Go ahead and press 'store'.
2) Now, check the right most checkbox in MARs Manual Address area. This indicates that we want RAM address 1. When you do this, the value of 28 disappears from RAMs value. Why is that? Because 28 was stored in memory address 0 and we are now looking at memory address 1. Now, enter 0b00001110 into RAMs Manual Input Value. This indicates a value of 14 to be stored. Press the store button to save this value

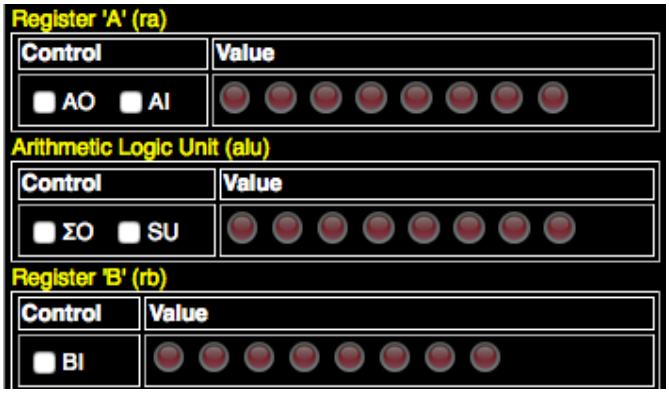
NOTE: For future reference, we will refer to the right most bit as bit 0 and the left most bit as bit 7 (or greater). This applies to LED value indicators and value input check boxes.

3) You can toggle bit-0 of MAR's Manual Address to see the two values you have now stored in RAM. Ensure a value of 28 is associated with memory address 0 (0b0000) and a value of 14 is associated with address 1 (0b0001). Can you see how our 'switch' metaphor works now?
4) Now, we will re-engage the MAR with the bus value. Ensure that MAR is getting its address from the bus by clicking 'go'. (The green LED should be lit to indicate this) Ensure MI is checked. Uncheck HLT from the clock.

5) When the PC has a value of 0 and 1, we should see our corresponding values displayed in RAM. Any other value for PC will show 0, which means no LEDs will be lit. This is expected. Each time the PC rolls around to 0 & 1, we will see our values.

What you are seeing now is how the CPU can retrieve values in sequence from RAM. This is how programs are executed. Our instructions for the CPU are stored values sequenced in order within the RAM. There are many more things to learn before we can make that happen. However, we need to talk about what a program actually is and decide what we want it to do at a high level before programming our CPU. Let's return to our addition problem of adding 28 + 14 = 42.

Getting the CPU to add two numbers



First off, we need to have a quick discussion about something called 'latching'. To latch means to capture something. In this case latching refers to a CPU component getting a value from the bus. This happens only if that components 'In' control line is enabled and a clock pulse has occurred. That is, the value from the bus does not immediately go into the component when enabled. This is a critical idea to understand. The CPU needs the ability to control when values are brought into a component. It needs time for some component to Output its value to the bus and a time for another component to Input or latch that value. This is what subsequent clock cycles are for. It should be noted that when a component's Output is enabled it goes immediately to the bus. But all Inputs are latched only on the raising edge of a clock pulse. This will be demonstrated shortly.

We now need to get our minds around the Arithmetic Logic Unit (ALU). This component is the 'calculator' within the CPU. In the case of SAP-1, it can only add and subtract. Where does it get the two values to add or subtract? From registers A & B. What are registers?

Registers are a special type of memory, not too different from RAM. Only in this casd, there is a single 8-bit value. It will store a value until overwritten by another. In most cases, registers have an In and Out control line that can be enabled. In our case, that is true for register A, but notice that B only has an In. That is due to the limited nature of our CPU and is not needed in our examples.

Notice that the ALU component has a ΣO and SU. That funny Greek symbol is Sigma (The math symbol for Summation is the Greek symbol Sigma, BTW) represents the letter 'S'. Ben uses this notation in his video, so I kept it if you are using this emulator with his videos. The ΣO, or 'Summation Out' causes the value within the ALU to go to the bus. The SU is the control line that tells the ALU to SUbtract versus add.

So, let's get this thing to do something useful for a change. Let's do 28+14=42.

0) Go ahead and press the reset button at the top to ensure we are starting at a known state. Pressing reset WILL NOT erase RAM so your values will not be lost. If you press the power button (or refresh your browser) however, all is lost. That is why RAM 'forgets'. So now, the clock should be blinking once per second. Press the HLT button as we will be pulsing the clock manually.
1) Enable MAR's MI Control Line, Enable the PC's CO Control Line, Pulse the clock. This causes MAR to latch the address on the bus which is 0.
2) Disable MAR's MI, Disable PC's CO, Enable PC's CE, Enable RAM's RO, Enable Register A's AI. Register A latched the value from RAM which is 28. The PC incremented since its CE was enabled. Notice the ALU now has a value of 28 since Register A is 28 and Register B is 0.
3) Disable RAMS's RO, Disable Register A's AI, Disable PC's CE, Enable MAR's MI, Enable PC's CO. Pulse the clock. Now the MAR is instructing RAM to show the value for memory address 1 which is 14.

4) Disable PC's CO, Disable MAR's MI, Enable RAM's RO, Enable Register B's BI. Pulse the clock. The value of 14 was latched by register B and now the ALU is showing 42 because A has 28 and B has 14. 28+14=42, right?
5) Disable RAM's RO, Disable Register B's BI. Let's display the value by enabling Output's OI and enabling ALU's ΣO. Pulse the clock. You now see our result of 42.

NOTE: We did not formally discuss the Output component, but at this point, hopefully, it does not require one as it is pretty simple and what you know about the other components can be applied to this one as well.

Congratulations, you have now added these two numbers in what probably seems the most complicated way imaginable.

This procedure is really a baby step in our overall understanding. Its purpose is to get you interacting with the CPU and see how setting the control lines and then pulsing the clock can coordinate the movement of values from one component to another to perform some useful function. In this case, adding two numbers.

Instruction Fetch Cycle

In the previous section, we were using the PC as a way to retrieve the values we were adding. That was sort of convenient just for learning purposes so we could get you interacting with the CPU quickly without having to go through understanding how instructions really work. That's gonna take a little more time.

In reality, instructions are loaded from RAM and then executed. So, there is a need to retrieve and 'decode' the instruction. That is, figure out what we need to do, then do it. Getting the CPU ready to execute an instruction is called the Fetch Cycle. It is basically the overhead (prep work) needed for each instruction. Luckily, it is the same for every instruction. What are those steps?

So, let's do these steps manually to understand how the fetch cycle works. However, we need to load a single instruction into RAM before we execute the Fetch Cycle procedure. That will go into address 0. We also need to store 28 into RAM as well. Previously, we had put it in address 0. It needs to get out our way for now, so that's why we are putting it at almost the end of memory at address 14.

1) Cycle the power button so that RAM is cleared.
2) Using the RAM's manual input, store the value 0001 1110 into RAM address 0
    a. NOTE: This is our 'secret value' and the meaning of this value will be revealed shortly.
3) Using the RAM's manual input, enter the value of 28 (0b00011100)) into RAM address 14 (0b1110)
4) Press reset button (RAM will be preserved) to ensure everything has been initialized to a known state.
5) Enable HLT on the clock.

Fetch Cycle

0) We need MAR's address to reflect the value in the PC
    a. Enable PC's CO – Its value goes immediately to the bus
    b. Enable MAR's MI - to receive bus value on the clock pulse
    c. Pulse the clock - so that MI latches the value from bus
1) Put the value of the current instruction to the Instruction Register (IR)
    a. Disable all control lines from Step 1 (not HLT)
    b. Enable RAM's RO – Instruction value goes immediately to the bus
    c. Enable IR's II (Instruction In) – so that 'II' latches value from bus
    d. Enable PC's CE - so it will increment and point to the next instruction
    e. Pulse the clock – IR value is latched and PC's counter is incremented
    f. Disable all control lines except HLT

You should now see that the Instruction Register (IR) has taken in the value we stored in RAM's address 0 value. Before we get carried away with explanations, let's break and start another section to talk about the instruction register (IR)

Instruction Register

In the previous section, we referenced the Instruction Register (IR), but we have yet to discuss exactly what that is. Also, after manually performing the Fetch Cycle, we were left with something interesting in IR component. Let's discuss that now.

You can see that the value we put in RAM address 0 is now in the IR. The main purpose of the IR is two-fold. First, as part of the fetch cycle, it receives the instruction. As you can see, the LEDs that hold this value are two different colors and the description indicates that there is an Instruction in blue and an 'operand' in yellow. What is an operand? In this case it is an address. In our SAP-1 level example, we are using the first 4 bits for an optional address and the last 4 bits for the instruction. We will only have a few instructions, so 4-bits is enough for that. Also, since RAM only contains 16 bytes (address 0-15) 4 bits is enough for that too. All instructions do not need an operand, so when that is the case, the operand bits are 0000.

You may also notice that the IR has interpreted this value for us. (it is a little too clever for its own good, btw). The first 4 bits (yellow) indicate memory address 14 (0b1110) and the last 4 bits indicate a value of 1 (0b0001) which is LDA? What is LDA? It is going to be our first instruction in our addition program that we are working up to. You might have guessed that is stands for 'Load A'. Good guess.

Another important fact to know about the IR, is that it has an IO control line. This of course, outputs its value to the bus. However, for this component, it only outputs the first 4 bits, which is an address. This fact will prove useful because we will somehow need to access values in RAM that are not instructions, but actual values. In fact, in our final program, RAM address 14 will hold the value 28, which is the first number we will be adding in our program. And address 15 will hold the value 14. Another equally important fact is that the instruction portion of the value is passed to the Microcode component, which we will be covering in a bit.

Since we have completed a fetch cycle and the LDA command is ready to be executed, let's go ahead and manually go through the two remaining steps.

Executing the LDA Instruction – last two steps

2) We need to get the address from the LDA instruction on the bus and into the MAR
    a. Enable IR's IO – first four bits go onto the bus
    b. Enable MAR's MI – Will take bus's first 4 bits and set RAMs address to that value (14)
    c. Pulse the clock
3) We need to get the value of RAM address 14 into Register A
    a. Disable all control lines except HLT
    b. Enable RAM's RO – Put value of 1 on the bus
    c. Enable Register A's AI – Take in the value of 28 from the bus
    d. Pulse the clock

Congratulations, you have just executed your first full instruction on this CPU. Pretty tedious huh? Great news though, a CPU does not mind executing these tedious tasks billions of times per second.

You will notice that the Fetch Cycle started with '0)' and ended with '3)' when considering the fetch cycle along with the execution of the LDA instruction. These correspond one to one with the Time Steps that are part of the Instruction Decoder (ID).

Before we discuss the ID, let's introduce a shortcut notation for documenting these steps to make our work easier.

Executing the LDA instruction: (shorthand)

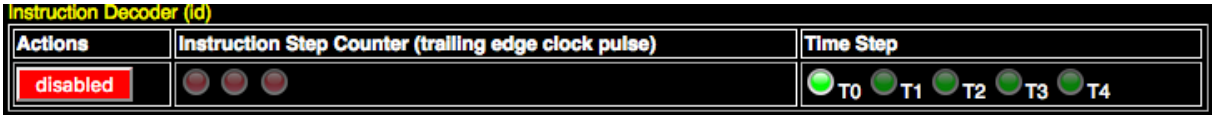
NOTES:

- Pulse the clock after each numbered step only (not Setup).
- Ensure only the control lines that have an 'X' are active for the given step. That is, uncheck control lines from previous step by default.
- Step 4 does not show any control lines set since the LDA instruction only needs two steps. Step 4 is considered a 'No Operation'. However, other instructions may need step 4.

Step	**Phase**		HLT	MI	IO	II	AI	CE	CO	RO
----	Setup		X							
Step 0	Fetch			X					X	
Step 1	Fetch					X		X		X
Step 2	Instruction			X	X					
Step 3	Instruction						X			X
Step 4	Instruction									

Ok, on to talking about the ID.

The Instruction Decoder (ID)



Up to this point, we have been executing steps on the CPU by hand. We have been loading values, setting control lines, and pulsing the clock manually. That was completely necessary so you can understand the steps that we will now automate.

You probably already realize that for the CPU to do its work, we need to set some control lines and pulse the clock. Then set a new set of control lines and pulse the clock again. Understanding the ID is the first step toward completely automating this process.

The ID is similar to the PC in that it takes the clock pulse and increments a counter. The big difference is that it acts on the trailing edge of the clock pulse. The reason for this is because it needs to help with the reset and setting of control lines BEFORE the next clock cycle. (leading edge)

Note: Even though you see 3-bits for the counter's display, we are only counting 5 steps (T0-T4). The counter resets back to zero after that. In our SAP-1 example, no instruction needs more than 3 steps to execute, so there is no need for more. Remember, that each instruction always requires the first 2 two steps to be the fetch cycle. The sum of those two sets of steps is 5.

How does the ID help? Our goal is to somehow automate the setting of the control lines in a defined sequence. To do this, we need to know two things. The first is what instruction we are executing and the second is where are we in the sequence of steps. The IR provided the value for the instruction (bits 4,5,6,7) and the ID provides 3 bits from its counter. With these two pieces of information, we can identify which set of control lines we want to. That is, these two pieces of information provides an address to look up control line settings for each instruction at the step level.

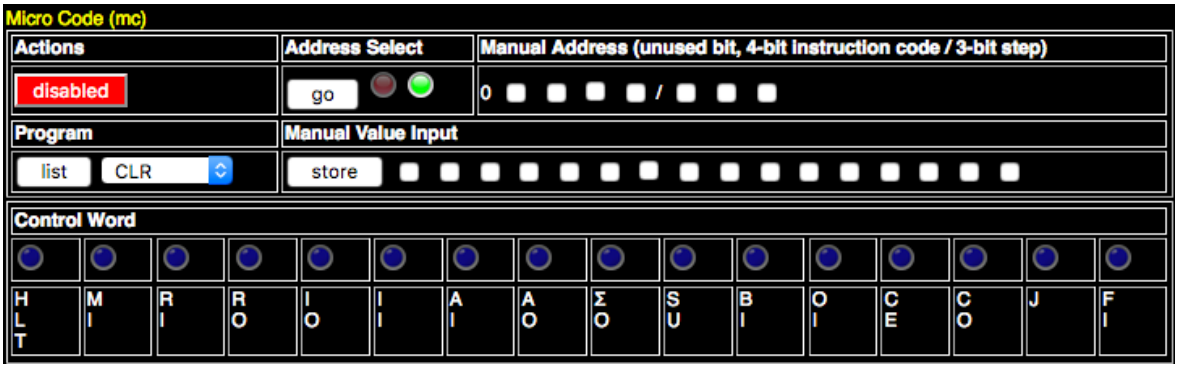
You might be thinking, "Why are there only 7 bits of information here? I thought our addresses are 8-bit?" You are correct. In building the address, bit 7 will always be 0. So, the format of this address will be:

0AAAABBB

Where bits 0,1,2 (BBB) is the counter from ID, bits 3,4,5,6 (AAAA) is the instruction from IR, and bit 7 is always 0 (zero)

0) Exercise, you can reset the CPU and enable the ID (Press red 'disabled' button to toggle enabled) and see how the counter is incremented. Re-execute the Fetch Cycle & LDA instruction. You will notice that the steps align to the T#.

Microcode (MC) - ROM



Earlier, we learned how to program the Ram. We stored and retrieved values based on address. ROM is similar in the fact that is stores values based on some address. The main difference is that the values in ROM do not go away when power is lost. (Power button will not erase our ROM, but a browser refresh will). These components have to be programmed, but they operate on a different set of principles, so their content is mostly permanent.

By default, our ROM is empty. There are a few microcode programs that you can 'load'. We will discuss those later on.

In our emulator, the ROM is not as obvious as it is in Ben's breadboard CPU. In reality, in most real CPUs, the microcode is hard-coded (literally etched) in the chip itself. Since Ben was building up a CPU from components, using actual ROM chips is the best way to emulate this. In the JavaScript we have emulated the ROM chips, but that is not important to our learning goals, so we will not get into those details.

We now know the Instruction Register (IR) and Instruction Decoder (ID) work together to form the address needed to retrieve the microcode. As you recall, with RAM, we have an address in which we referenced a stored value. An address is just a number that can have any meaning we give it. The only requirement is that each address is unique so we can use it to lookup a certain value.

What exactly do we want to store? We want to store the information that we put into our step/control line table in the previous sections. That is, we want to store a value which represents the settings for each control line. Conveniently, each control line is a switch which is either on or off. And we can easily store this information in binary format. That explains what we want to store, but how will that be formatted?

If you look at the image above, you will see something called a Control Word. A Control Word is just the set of all the control lines. You will notice that some of the control lines you have been working with are there along with others you have yet to interact with. All in all, there are 16 control lines in our SAP-1 emulation. Having 16 control lines means we have 16 switches or 16 bits. Ben used two separate ROM chips in his CPU since they can only store an 8-bit value each for each unique address and we need 16 bits. This is accomplished by programming each one with the same address, but each one gets 8-bits from our Control Word. ROM #0 gets bits 0-7 and Rom #1 gets bits 8-15. Don't worry too much about this as it is not exposed in the CPU UI. (It's just tie-in info if you are watching ben's videos too)

You will notice that the MC component has a similar interface for programming, just like the RAM component. Let's program just enough microcode to automate executing the LDA instruction.

0) Make sure the ID and MC are disabled
1) Reset the CPU and enable the HLT flag
2) Click the go button to set manual entry for the MC
3) Use the following table to know the address and value. Program these into the MC.

Step	**Address**		HLT	MI	RI	RO	IO	II	AI	AO	ΣO	SU	BI	OI	CE	CO	J	FI
Step 0	0b00001000			X												X		
Step 1	0b00001001					X		X							X			
Step 2	0b00001010			X			X											
Step 3	0b00001011					X			X									
Step 4	0b00001100																	

0) Make sure the ID and MC are disabled
1) Rest the CPU and enable the HLT flag
2) Enable the ID and MC
3) Pulse the clock 5 times slowly. Observe what happens after each pulse

Wait, nothing happened except the cycling from T0 – T4? What the heck? (I did know this would happen, muh ha ha ha)

Ok, when the CPU needs to get its very first instruction, there is no instruction value. It is zero. (In reality, there is an instruction for value 0000, it's called NOP for 'No Operation') However, we did not program a fetch cycle for this case. Enter the following. Then execute the LDA instruction.

Note: You can think of this as a sort of a 'boot strap' for getting the CPU started on it first Instruction. It also serves as the Microcode for the NOP instruction as well

Step	**Address**		HLT	MI	RI	RO	IO	II	AI	AO	ΣO	SU	BI	OI	CE	CO	J	FI
Step 0	0b00000000			X												X		
Step 1	0b00000001					X		X							X			
Step 2	0b00000010																	
Step 3	0b00000011																	
Step 4	0b00000100																	

When complete, re-run the LDA instruction. You should now see the value of 28 (0b00011100) in Register A.

What’s next?

We need to review and validate what we have so far before finishing the rest of the instructions…

More to come…