<u>**THE FUNCTIONALITIES OF THE 8-BIT CPU**</u>

This piece is written to make clear all the functionalities of the 8-bit microprocessor we built. It's an 8-bit microprocessor because it has an 8-bit data bus. Let's begin with this;

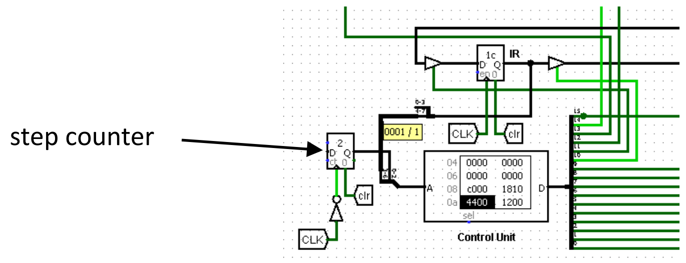
HOW IT ALL STARTS

Note that all the processes, especially the transfer of data from one memory component to the other are all controlled by the CU (control unit).

- The address of the first instruction is transferred from the PC (program counter) to the MAR (memory address register).
- The address in the MAR points to a particular location in the memory (usually the first instruction is placed in the first memory location-address 0000 in binary).
- The instruction that the address points to is moved from the memory to the IR (instruction register).
- The 8-bit instruction is made up of two parts the opcode and the operand.

7 – 4	3 – 0
opcode	operand
xxxx	xxxx

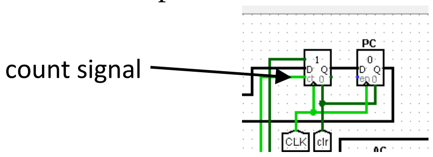
- The opcode is split from the instruction that's in the IR using a *splitter* and directed into the address port of the CU.



  - ○ The CU is just a ROM that holds the microcode in discrete memory locations. The opcode coming into the address port is joined to the values coming from the counter beside the CU, let's call it the *step counter*. The opcode together with the counter values point to a particular location in the ROM (CU). The value it points to is the microcode which implements the opcode.
  - ○ The instruction (opcode) is executed in steps. The steps are provided by the step counter. The step counter always counts up to the fixed maximum number of steps all the instructions will need to finish executing. It counts from 0 up to 4.
  - ○ As the step counter continues to count the CU moves on to the next microcode instruction for a particular opcode till the fixed number steps is reached then it resets to 0 and begins counting from there.
  - ○ The CU has been preprogrammed to carry out the fetch part of all the instructions which is common to all.
  - ○ For example, let's say a load (0001) instruction comes in into the CU. The 0001 is joined with the current step value of the counter which is 001, but at that point

step 001 from the counter has already been executed as part of the preprogrammed steps to fetch the instruction that was talked about. So the next step (010) is added to the 0001 and becomes 0001010. 0001010 which is 7 bits long enters the address port of the CU to be decoded. The 0001010 value should be thought of as an address which points to a memory location in the ROM. That location holds a particular microcode. The value comes out at the output where the value is split into its individual bits using a splitter. Each of these lines go to different components of the microprocessor– three-state buffer, registers, RAM, counters, the ALU, logic blocks, etc.

- After fetching an instruction into the IR the PC is incremented by 1. This is done by sending an on signal to the count port of the counter.



- If the instruction needs an operand for execution, then it is also split to be used. Usually the operand goes to the MAR to point the value you want to use because it is just an address. Other times, during a jump instruction the operand (address) is passed to the PC.

In a nutshell that's basically what the microprocessor goes through to execute the instructions found in the RAM. What's left with now is how the individual instructions are executed. Before we commence please note that the fetch part of the cycle is similar for all the instructions so it may not be covered in details again. You can refer to "HOW IT ALL STARTS" to understand it more.

LOAD X

The load instruction consists of two parts, the opcode and the address it going to work with. The opcode value is 0001 but the operand is whatever valid value (address) you specify. An example code will look like this '00010111' (0x17), where '0001' is the opcode for load and '0111' is the address/location in memory where the value you are going to load can be found. The load instruction simply lets you load data from the memory location specified by the operand into the accumulator. With that out of the way let's look at the steps in details.

- After the load instruction has being transferred to the IR, the opcode, 0001 is split from the instruction to be joined with the value from the step counter. For load the step counter value '010' which is the 3rd step after execution of the fetch is where the actual load execution starts. So the complete value becomes '0001010', which enters the address port of the CU.
- The CU holds a value called the microcode to execute the load instruction. The value found at address 0001010 in the CU looks like this '0100010000000000'. It's in binary and each bit represents a line. These lines go to separate components to control them.

The two 1's in the binary value actually go to two three-state buffers to allow data to flow from one spot to the other. And by the way '0' means off.

- This code '0100010000000000' allows the IR value to be passed into the MAR (note that not the whole IR value but only the operand part). The operand is detached to enter into the MAR.
- After that's done the step counter increments to 011 making the whole value '0001011' which enters the CU to be decoded.
- The next value at address location 0001011 is '0001001000000000'. This comes out of the CU and the individual bits split to wherever they have to go to.
- The two 1's in the code go to 2 three-state buffers which allows the data at the address that was placed at the MAR to move from the RAM to the AC (accumulator). Remember the address in the MAR is the operand part of the instruction that was transferred from the IR.
- With that done the step counter increments for the last time because the maximum count value is set to '100', before it resets to 000.

And that is it, the load instruction is complete. The value at address location 0111 has been successfully put into the AC. Its notation looks something this;

MAR←X

AC←M[MAR]

X is the operand, **M** stands for memory (the RAM). And **MAR** is acting as the index.

I'm tired right now so what I will do is this, I will just tell you what the rest does and something small about how they do it or maybe not. But I trust that you can apply the concept used for the load to other similar instructions.

STORE X

The store instruction is in the form '0010xxxx', where 0010 is the opcode in binary and xxxx is the operand which is any valid address value you specify. The store instruction lets you transfer data from the AC into the memory location specified by the operand. Its notation is this,

MAR←X

M[MAR] ← AC

ADD X

The add instruction adds the value found at the memory location specified by the operand to the value found in the AC. It's in the form, '0011xxxx'.

What happens is this;

- The value the operand points to is moved into the register B, so that the add operation can be carried out.

- The appropriate lines have to be asserted in the ALU to carry out the add command.
- After the addition, the result is moved into the AC.
- And that’s all.

This applies to all the other operations that uses the ALU except NOT. This is its notation;

```
MAR←X
RegB← M[MAR]
AC←AC+RegB
```

SUBT X

It is in the form 0100xxxx. Notaton;

```
MAR←X
RegB← M[MAR]
AC←AC-RegB
```

MULT X

It is in the form 0101xxxx. Notation;

```
MAR←X
RegB← M[MAR]
AC←AC×RegB
```

DIV X

It is in the form 0110xxxx. Notation;

```
MAR←X
RegB← M[MAR]
AC←AC/RegB
```

OR X

It is in the form 1010xxxx. Notation;

```
MAR←X
RegB← M[MAR]
AC←AC | RegB
```

AND X

It is in the form 1011xxxx. Notation;

```
MAR←X
RegB← M[MAR]
AC←AC & RegB
```

NOT

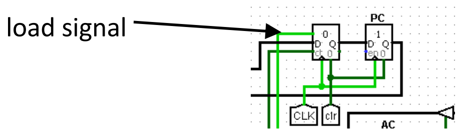
It is in the form 1100. Notice that this does not require any operand. What it does is just NOT the value in the AC and then put the result back into it. We don't need the register B in this case so the notation looks like this;

AC← ~AC

JUMP X

Jump instruction let you jump to a particular point in your code. It is very useful for looping. It is in the form, 0111xxxx. The operand specifies the address you want to jump to. The jump instruction is executed a differently.

- From the IR the operand (address) doesn't go into the MAR but rather it moves to the PC. Why? Because the PC holds the address of the next instruction.
- But first you have to send a signal to the counter which is part of the PC. This signal is the load signal. When the load signal is on the counter allows data to be input into the counter. Then the counter starts counting from that value upwards.



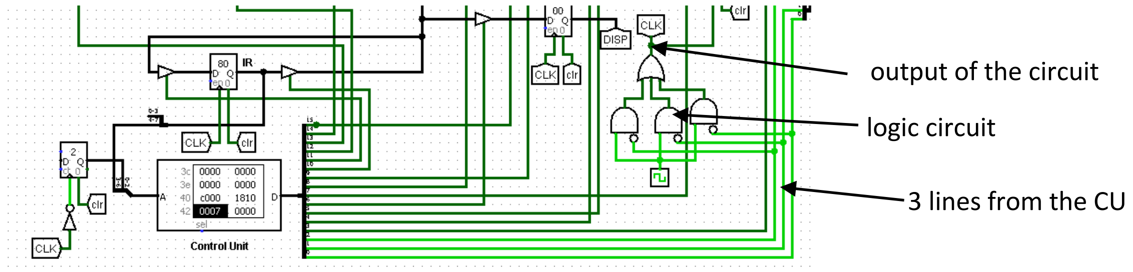
- The PC then puts its value into the MAR and the instruction you want to jumped to is fetched. Then the whole fetch-decode-execute cycle continues again.

Its notation is;

PC ← X

HALT

Halt simply halts the program. More specifically it makes the clock ticks ineffective. It is in the form, 1000. It doesn't need any operand to execute. There is a logic circuit in the microprocessor which takes inputs from three lines that come from the CU. This block also takes input from the clock. The specific microcode value sends three 'on' signals into the input which in effect makes the clock ticks ineffective thereby halting the whole microprocessor.



You can see from the figure above that even though the clock is on, it doesn't show at the output.

OUT

The out command allows the value in the AC to be transferred to the output register. This is important when you want the result of an arithmetic or logic operation to be displayed on a 7-segment display. It is in the form, 1001. It needs no operand. Notation is;

OutReg←AC