**Single Core**

**Instructions:**

On a single core design, we are implementing 8 main instructions:

* Branch On Zero (BZ)
* Branch On Negative (BN)
* Store (ST)
* Rotate Left (RL)
* Load Immediate (LI)
* Load from Memory (LM)
* Add (AD)
* Subtract (SB)

***Branch On Zero(BZ)***

On BZ instruction we need to look at the inside of Accumulator (ACC) firstly if it is zero we can increment the program counter by the given fetch address’ least significant 5 bits.

We have an output named BZ on ALU if inside of ACC is 0 it always gives “1” output.

We have another output on Control Unit named isBZ if the output of that is “1”, that means we need to do Branch on Zero.

With and gate we can get “1” output if both conditions satisfy each other.

***Branch On Negative(BN)***

On BZ instruction we need to look at the inside of Accumulator (ACC) firstly if it is negative we can increment the program counter by the given fetch address’ least significant 5 bits.

We have an output named BN on ALU if inside of ACC is negative it always gives “1” output.

We have another output on Control Unit named isBN if the output of that is “1”, that means we need to do Branch on Negative.

With and gate we can get “1” output if both conditions satisfy each other.

***Store(ST)***

On ST instruction we need to store the ACC’s value into determined place in RAM.

Least significant 5 bits of the fetch address is the determined place in RAM.

For the store instruction, we need to enable “sel” input of RAM, disable the “ld” input of RAM, disable ACC\_to\_MEM input of Control Unit on our design and also ALU is not important for this instruction so we don’t need to use it. After all we need to increment Program Counter(PC) 1.

***Rotate Left(RL)***

On RL instruction we are rotating the bits of ACC data to left by looking the least significant 3 bits of fetch address.

That instruction is useful for multiplication, it is like multiplying by Powers of 2.

For that instruction we are using just the ALU and ALU have a specific circuit to do that.

After that we need to increment PC 1.

***Load Immediate(LI)***

On LI instruction we are adding the least significant 5 bits of fetch address into ACC.

For that instruction, we don’t need to use memory and ALU. That means we just need to select the input data of ACC(it needs to come directly from fetch address).

After that, we need to increment PC 1

***Load From Memory(LM)***

On LM instruction we are getting specific data from RAM and putting it to ACC.

So for that, we need to use RAM, but for reading not writing so both “ld” and “sel” inputs will be 1.

After that, we need to increment PC 1.

***Add(AD)***

On AD instruction we are adding specific data from RAM to ACC’s value and store it into ACC.

To do that, we need to use RAM and ALU, so we are determining the Control Unit outputs by looking that. ALU has a specific circuit to do that calculation.

After all we are incrementing PC .

***Sub(SB)***

On SB instruction we are subtracting specific data from RAM to ACC’s value and store it into ACC.

To do that, we need to use RAM and ALU, so we are determining the Control Unit outputs by looking that. ALU has a specific circuit to do that calculation.

After all we are incrementing PC .

**Circuit**

We can see the whole single core at Figure 1. It has an EnableCore input which is for MV operation if there is a MV operation it disables the core, also EnS is for that job too. We have one more input for Read or Write on move operation if it is 1 that means we are creating a MoveDataOut output, it goes to MoveDataIn of other core. MoveAddress input is for selecting to write or read position from RAM.

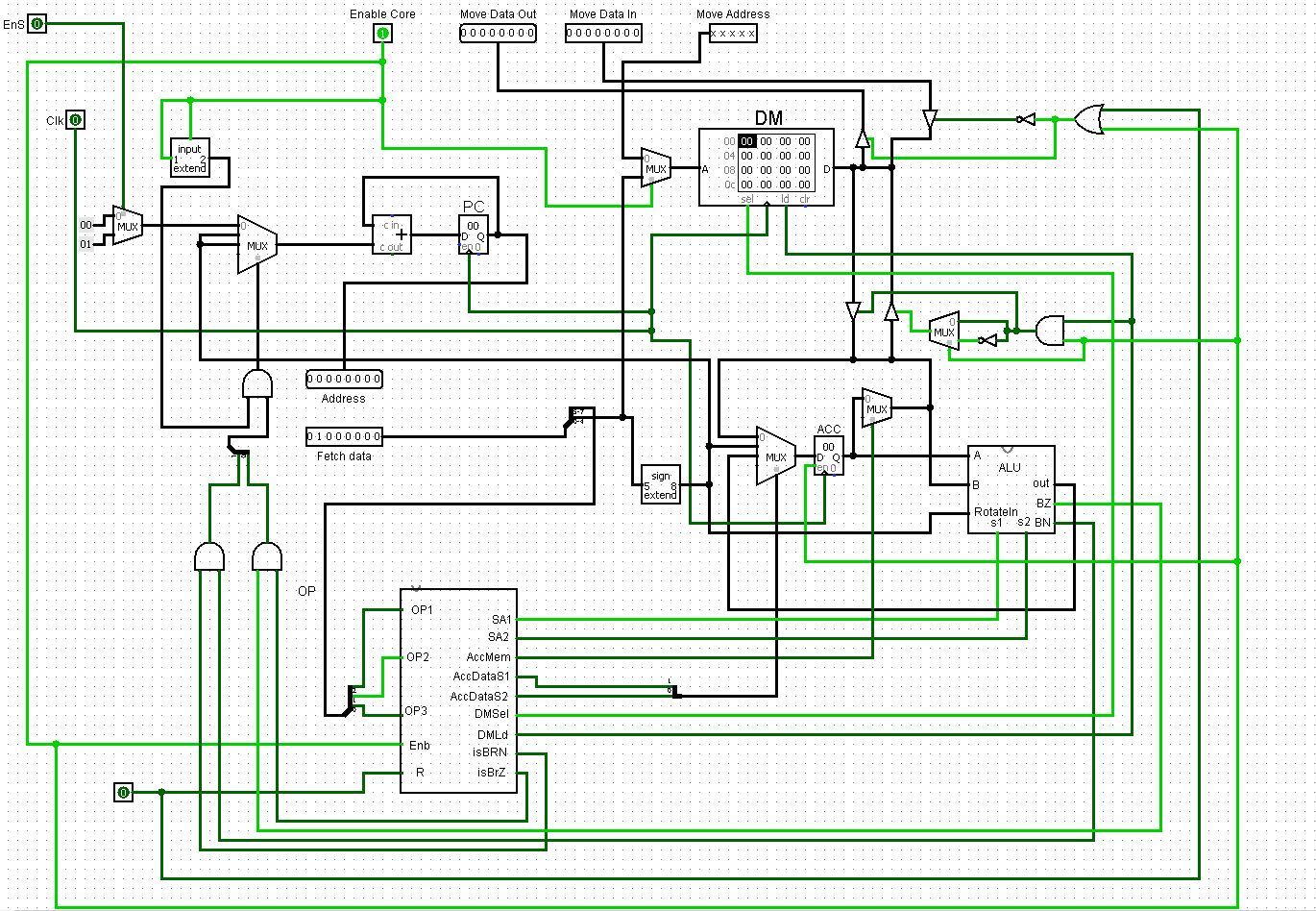


Figure 1: Whole Single Core

At the Figure 2 we can see the program counter implementation, with the and gates on the bottom of Figure 2 we can determine is it a Branch operation or not If it is branch with a MUX we are selecting the right incrementation. By the way if there is a move operation we are not incrementing the program counter so for that we used another MUX on the left of Figure 2. After incrementing the counter it prepares and writes the address to Address output.

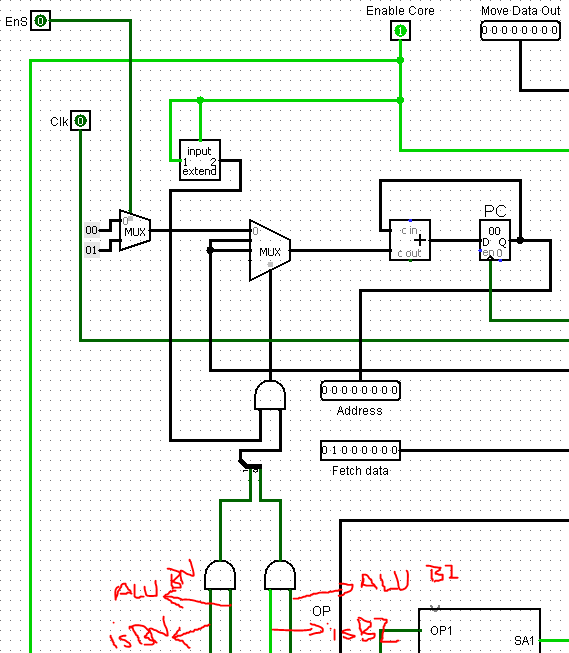


Figure 2: Program Counter implementation

You can see the related circuit to ALU and ACC operations on the Figure 3 below. Each operation on ALU gets the fetch input’s least significant 5 bit and sign extends it because ALU and ACC works on 8 bit. If ACC needs a data from Data Memory(RAM) it goes to leftmost MUX’s first input, if ACC needs sign extended version of address it goes to second input, if ACC needs the output of ALU it goes the third input (this operations are controlled by Control Unit). Also, we can select to forward our ACC’s output to RAM by the middle MUX.

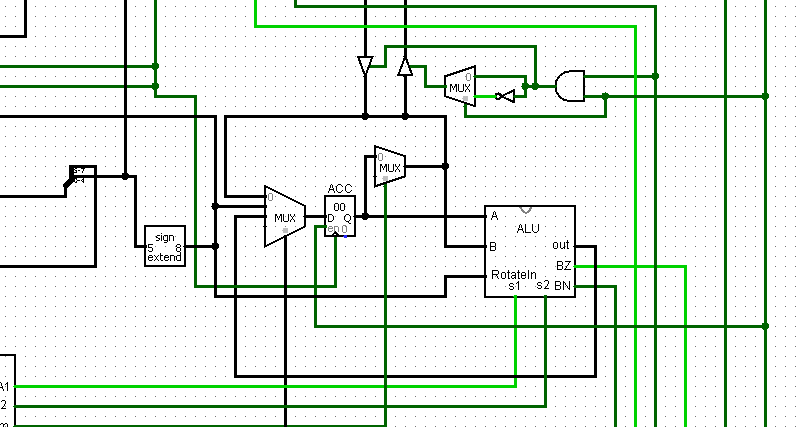


Figure : ALU and ACC operations

On the Figure 4 usage of RAM is implemented. On that figure we can get inputs from other core or from ALU and ACC, by using buffers and MUXes we determined is the input coming or going through, if the input is coming to RAM we are looking for is the input coming from other core or ACC. This selection is implemented by looking is core enabled, if enabled that means input coming from ACC else that means it is coming from other core. On the other situation if we need to read the input we are going to disable all other buffers and muxes.

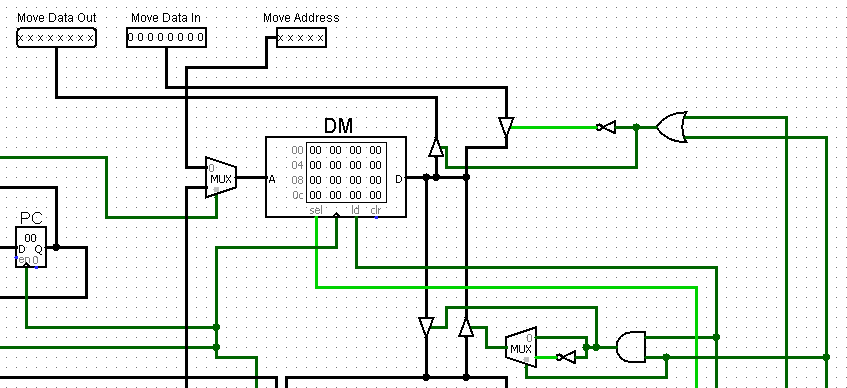


Figure : Implementation of Data Memory(RAM) Management

**ASSEMBLER**

In our Assembler design we used Java programming language, our assembler supports different operations as you can see in Table 1 and Table 2

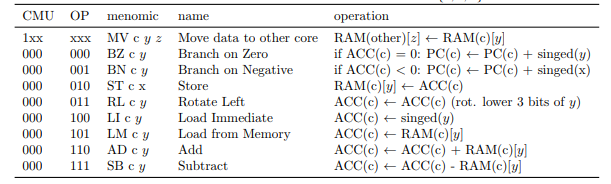


Table 1: Multi Core Operations

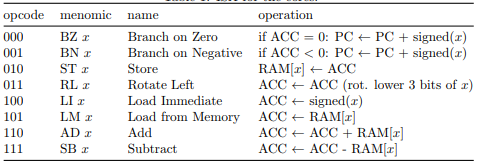


Table 2: Single Core Operations

And also there is another instruction for copying a value to a string like “hiro EQ 5” this instruction allows you to use “hiro” string instead of using “5”.

The assembler is supporting comments too.

The supported format in the single core operations: “instruction address #comment”

The supported format in the multi core operations: “instruction core address #comment”

The supported format in move operation “instruction core1 adressCore1 addressCore2 #comment”

The supported format in EQ instruction “string instruction value #comment”

All of the addresses must be in decimal format, instructions must be in string format, cores must be on decimal format(we have just 2 cores).

***Question 5.2***

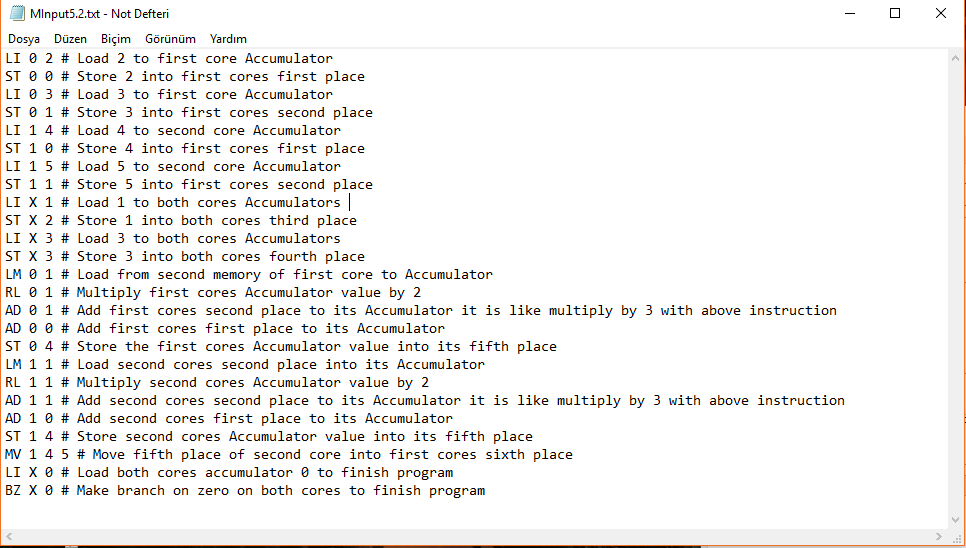


Figure : Input file of Question 5.2

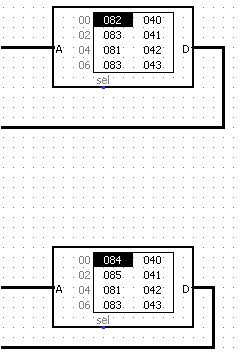
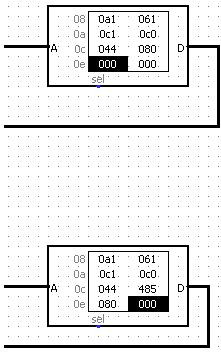


Figure : Beginning of Instructions



Şekil : End of Instructions

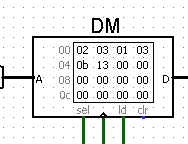


Figure : First cores RAM after instructions

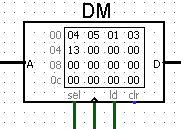


Figure : Second cores RAM after instructions