Computer Architecture - COMP206

Dual Core CPU and Assembler Design

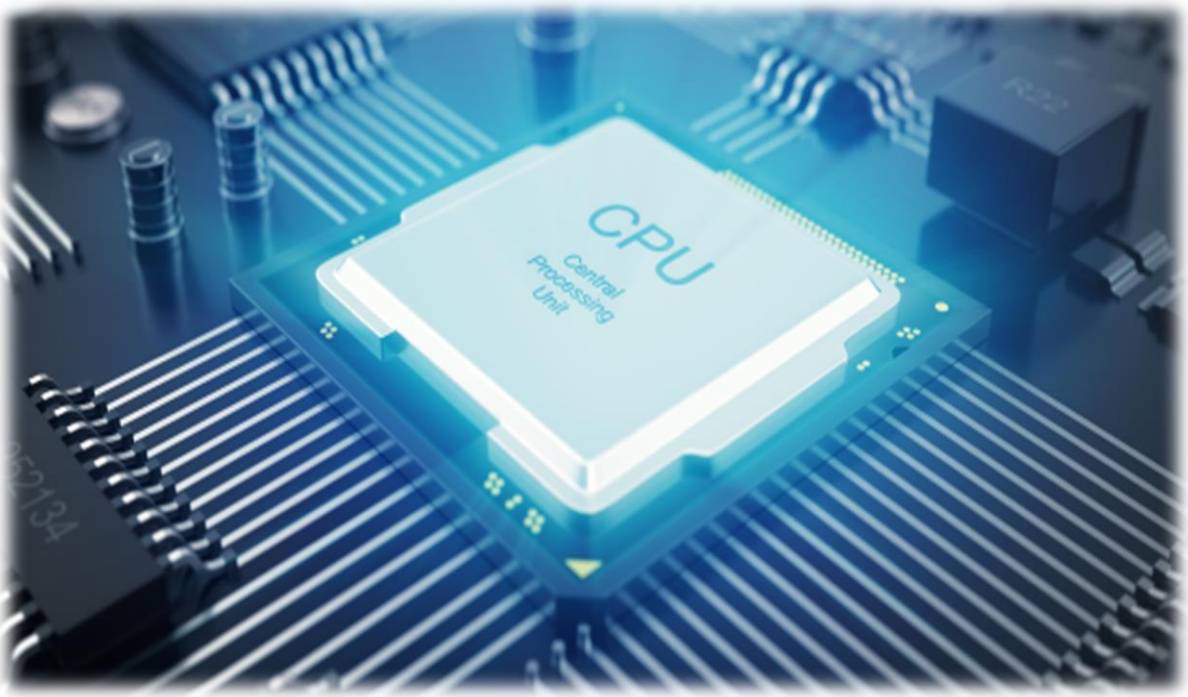
Muhammed Rahmetullah Kartal – 041701008

Ramazan Yetişmiş – 041701013

Instructor: Şuayb Ş. Arslan

Institution: MEF University

Date: May 24, 2019



Introduction

In Project firstly, we read the Project description carefully and then begin to work. Firstly, we implemented our 8-bit ALU. Then, we organized the registers, RAM and other stuff to decide the control units. After that, we implemented our control logic unit due to our main core controls bits. We faced lots of difficulties such that, we have 2 separate data in and 2 separate data out, we spent a lot of time on dealing with this problem, actually that was the majör problem after handling that everything was easy.

Objective

This Project aims 3 things:

First of all implementing our single core design which will have a datapath and a control unit that process 8-bit instructions based on the ISA specification, and ALU operations over a common 8-bit bus. The next phase is the Dual Core design, we doubled our Core and by the help of the CMU, we can control them as one complete core. The final phase was the implementing our own assembler with using High-level Language we fort his step we used Java.

Arithmetic Logic Unit (ALU)

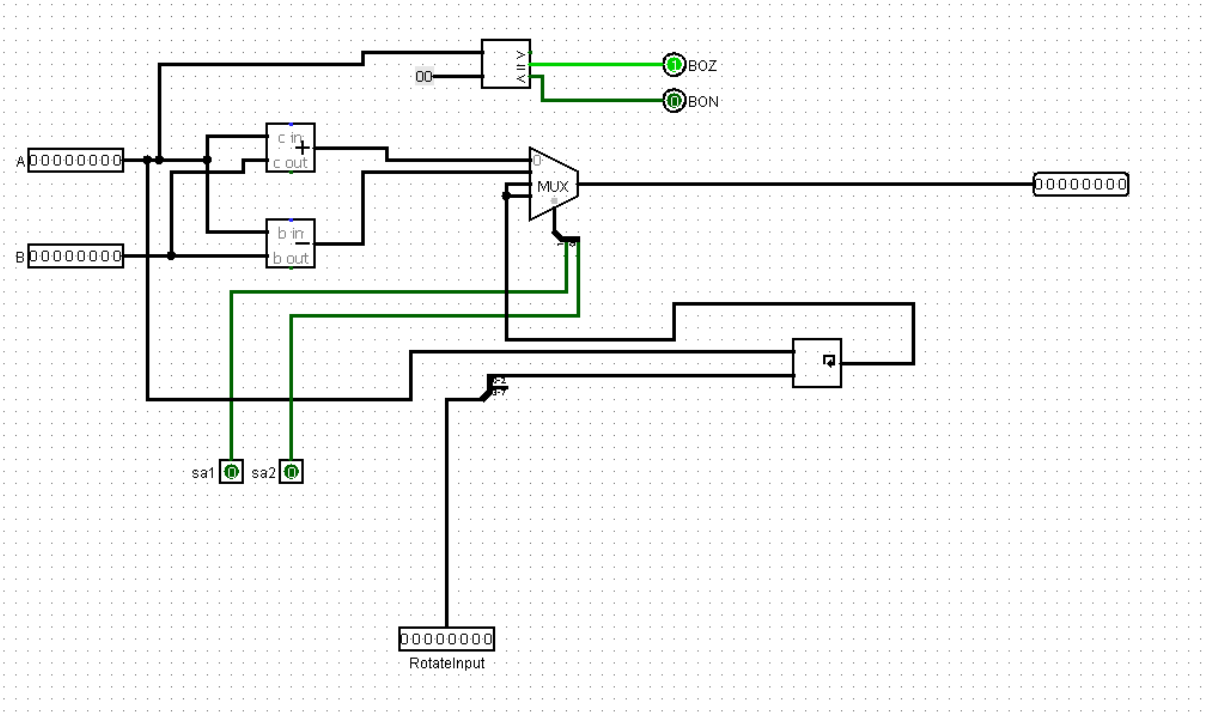


Figure 1-ALU Design

Our ALU performs the whole instruction which is given by the instruction sets. We used 1 adder, 1 substructure,1 comparator, 1 shifter and lastly 1 shifter. The ALU takes 5 inputs and gives 3 outputs. Sa1 and Sa2 control the operations, A, B and C are the 8-bit inputs which are going to be used in the operations such as add, sub, left shift.BZ controls the branch on zero and BN controls the branch on negative. Overall by the help of the ALU, we are able to perform the instructşons correctly.

Control Unit:

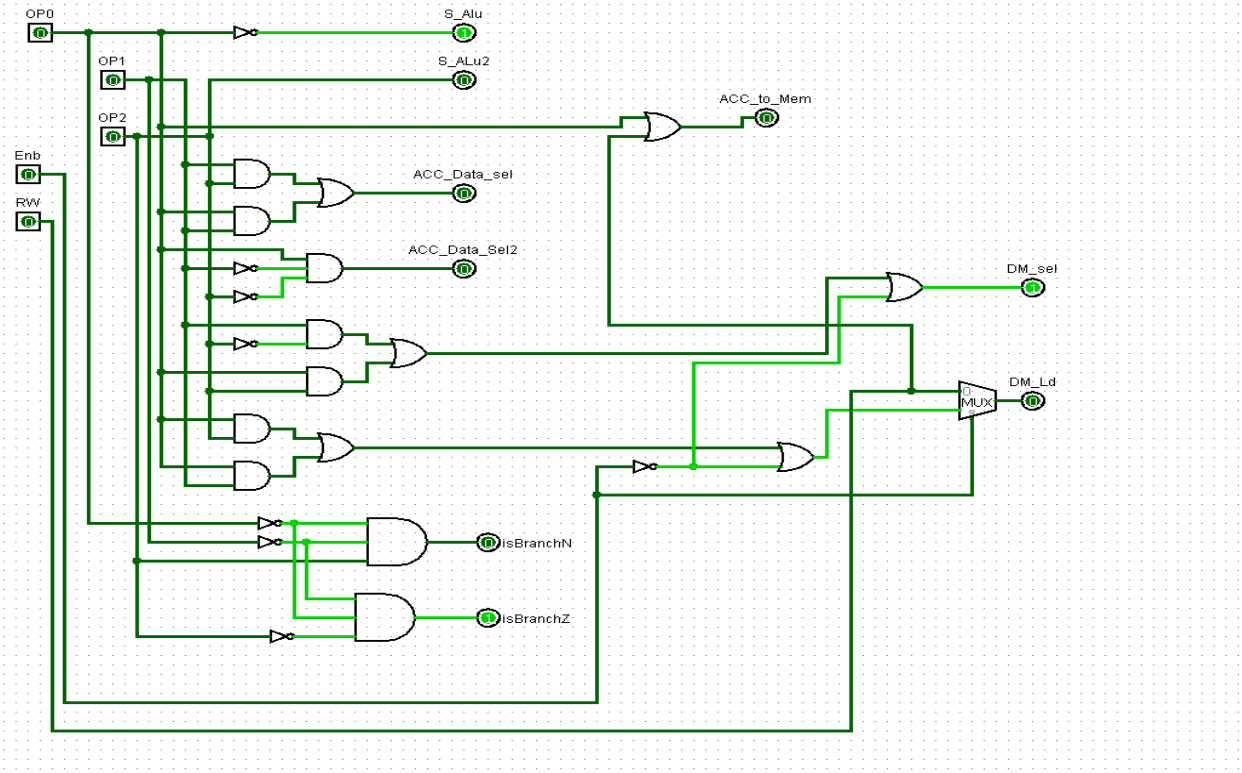


Figure 2 Control Unit

In order to implement the Control Unit in this circuit, we used the table below.

OP1	OP2	OP3	SA1	SA2	ACC_to_Mem	ACC_Data1	ACC_Data2	DM_Sel	DM_LD
0	0	0	x	x	x	x	x	0	0
0	0	1	x	x	x	x	x	0	0
0	1	0	x	x	0	0	0	1	0
0	1	1	1	x	x	1	0	0	0
1	0	0	x	x	x	0	1	0	0
1	0	1	x	x	x	0	0	x	1
1	1	0	0	0	1	1	0	1	1
1	1	1	0	1	1	1	0	1	1

Figure 3 Truth Table of the inputs and outputs

After this table, we add more selection bits because we faced some difficulties such as implementing the Move instruction, we added extra 3 bits, enable1, enable2 and read. Read determines, the core which the data will be taken.enable1 gives us chance to cease the Core and enable2 helps us to stall the clock 1 time after each move operation. By the help of these extra bits, we can process the operations simultaneously.

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 at 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 at that. ALU has a specific circuit to do that calculation. After all, we are incrementing PC.

Circuit

We can see the whole single core in Figure 4. It has an EnableCore input which is for MV operation if there is an MV operation it disables the core, also EnS is for that job too. We have one more input for reading or writing 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.

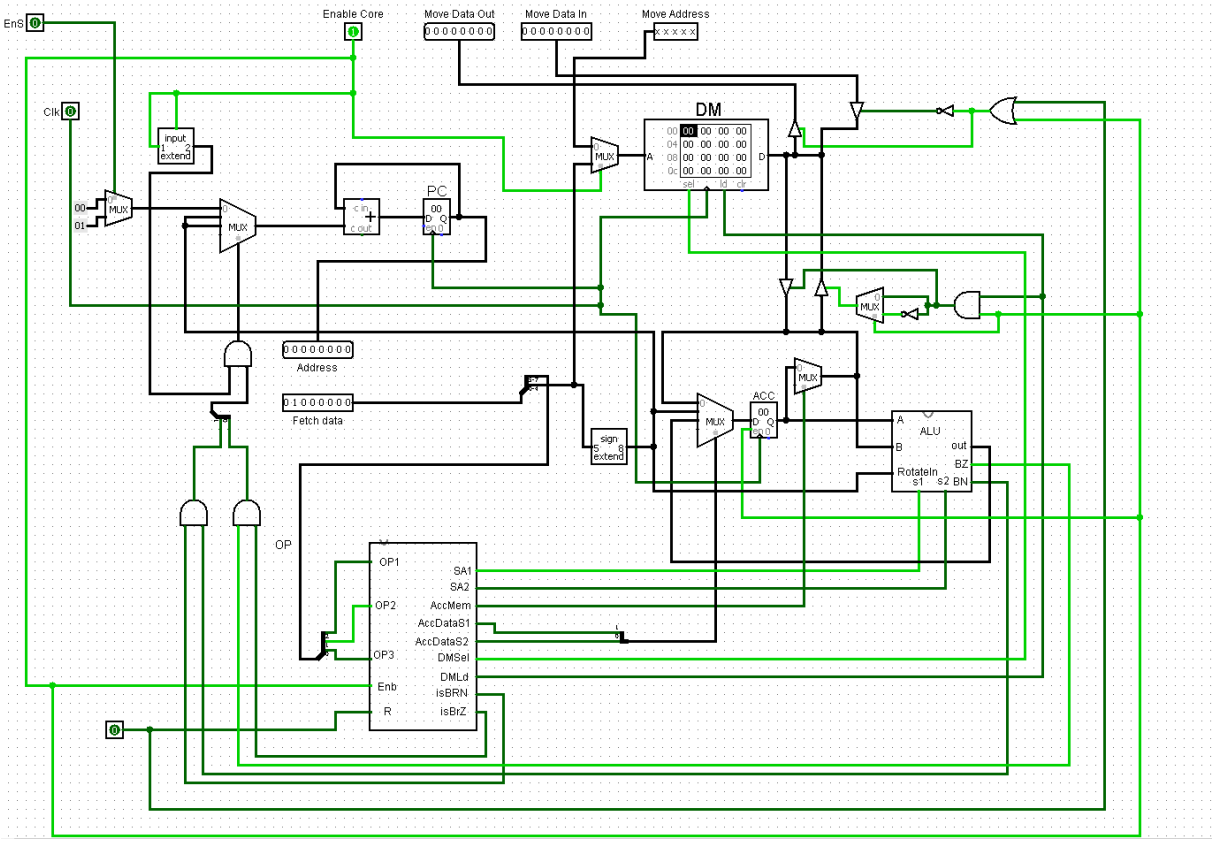


Figure 4: Whole Single Core

At the Figure 5 we can see the program counter implementation, with the and gates on the bottom of Figure 5 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 5. After incrementing the counter it prepares and writes the address to Address output.

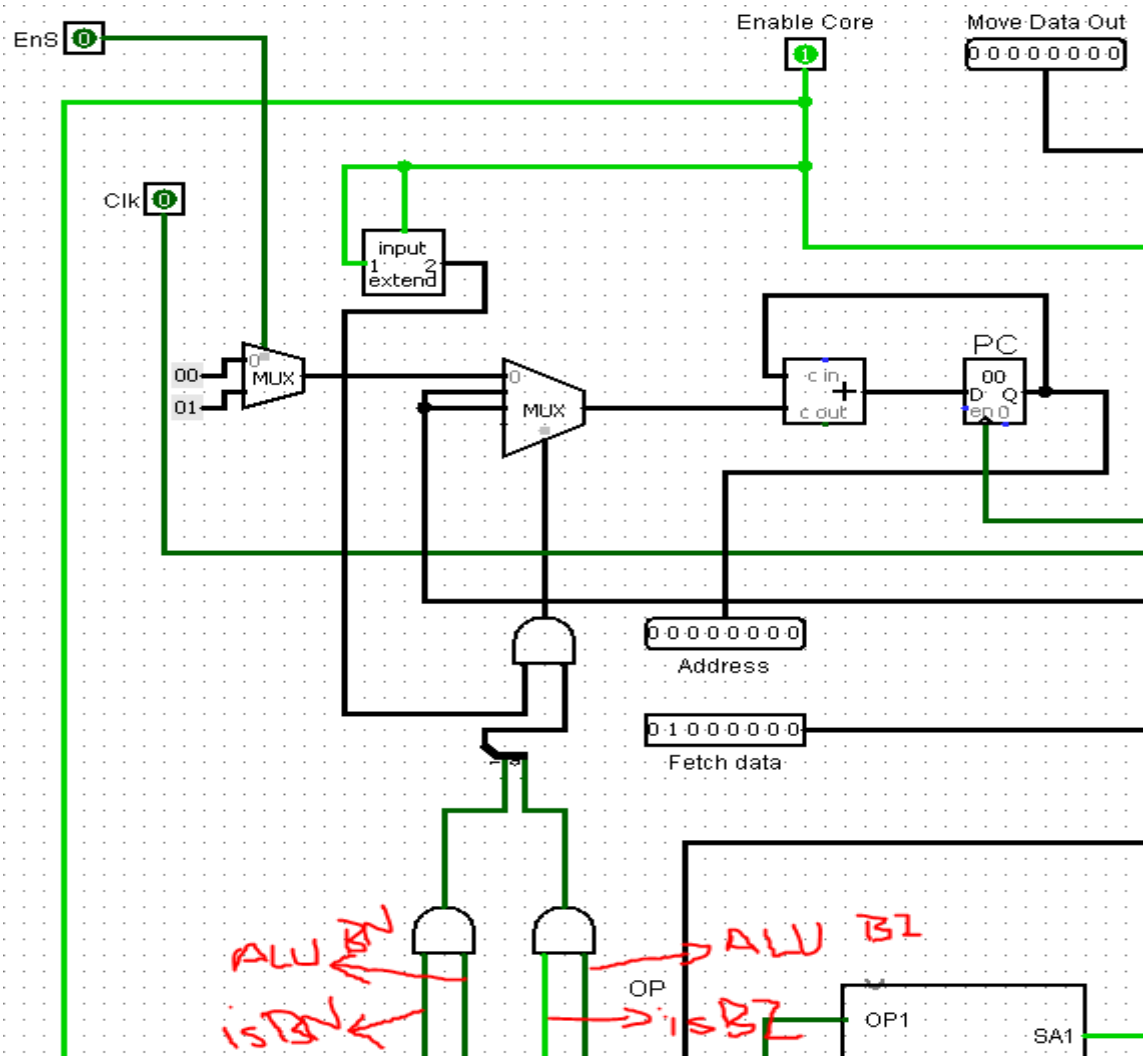


Figure 5: Program Counter implementation

You can see the related circuit to ALU and ACC operations in Figure 6 below. Each operation on ALU gets the fetch input's least significant 5 bit and sign extends it because of ALU and ACC work 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 the second input if ACC needs the output of ALU it goes the third input (these operations are controlled by Control Unit). Also, we can select to forward our ACC's output to RAM by the middle MUX.

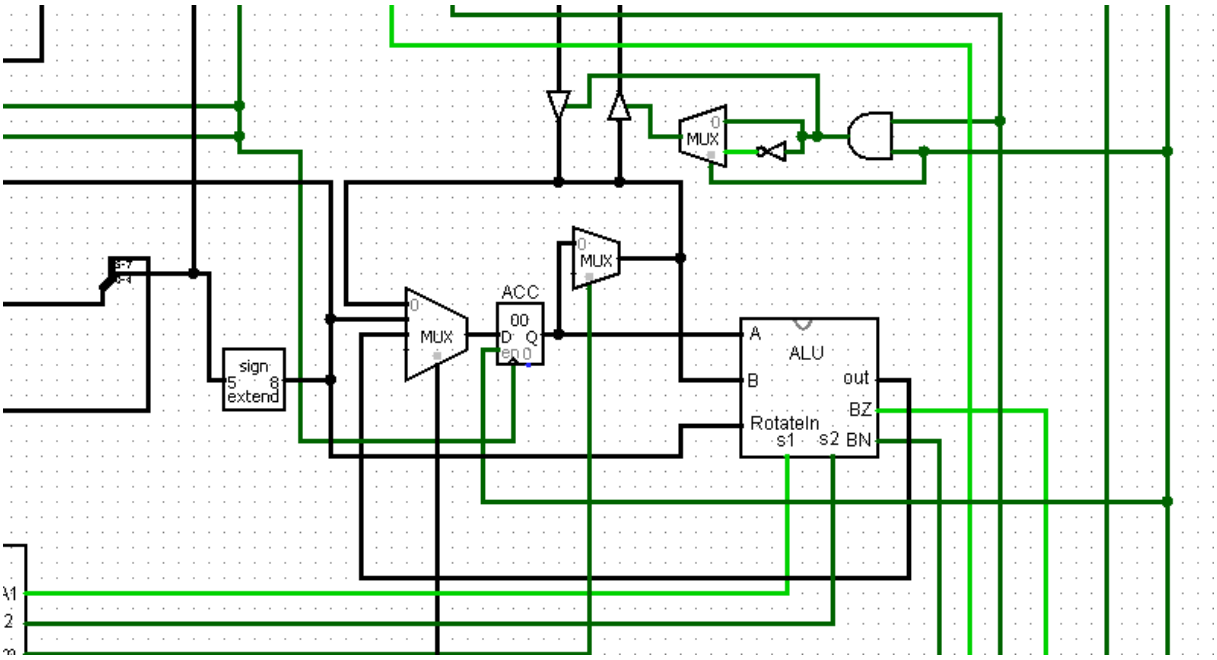


Figure 6: ALU and ACC operations

On Figure 7 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.

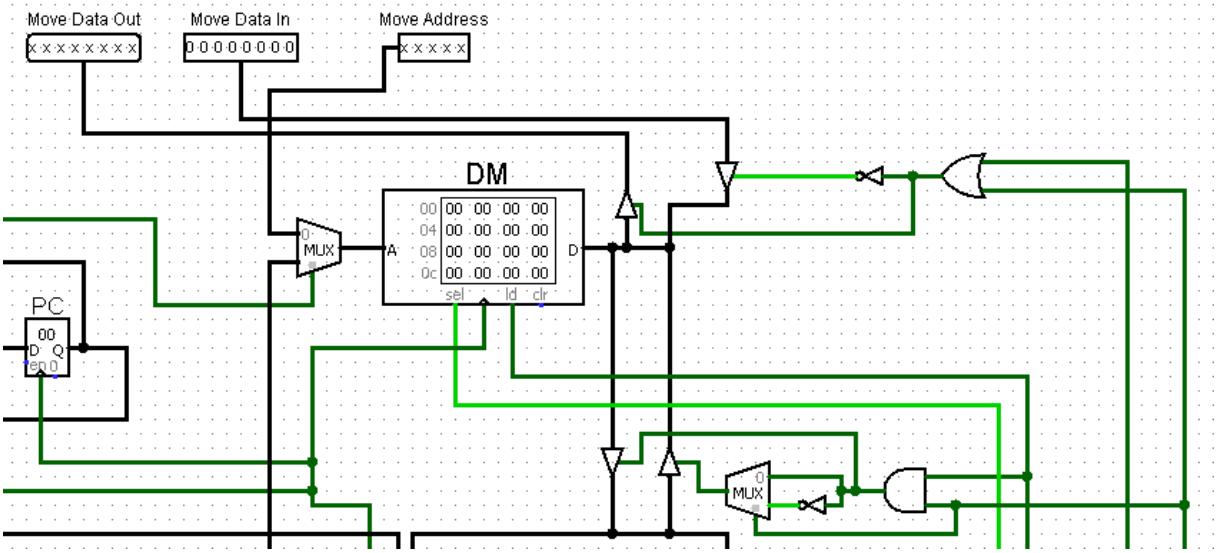


Figure 7: Implementation of Data Memory(RAM) Management

CMU:

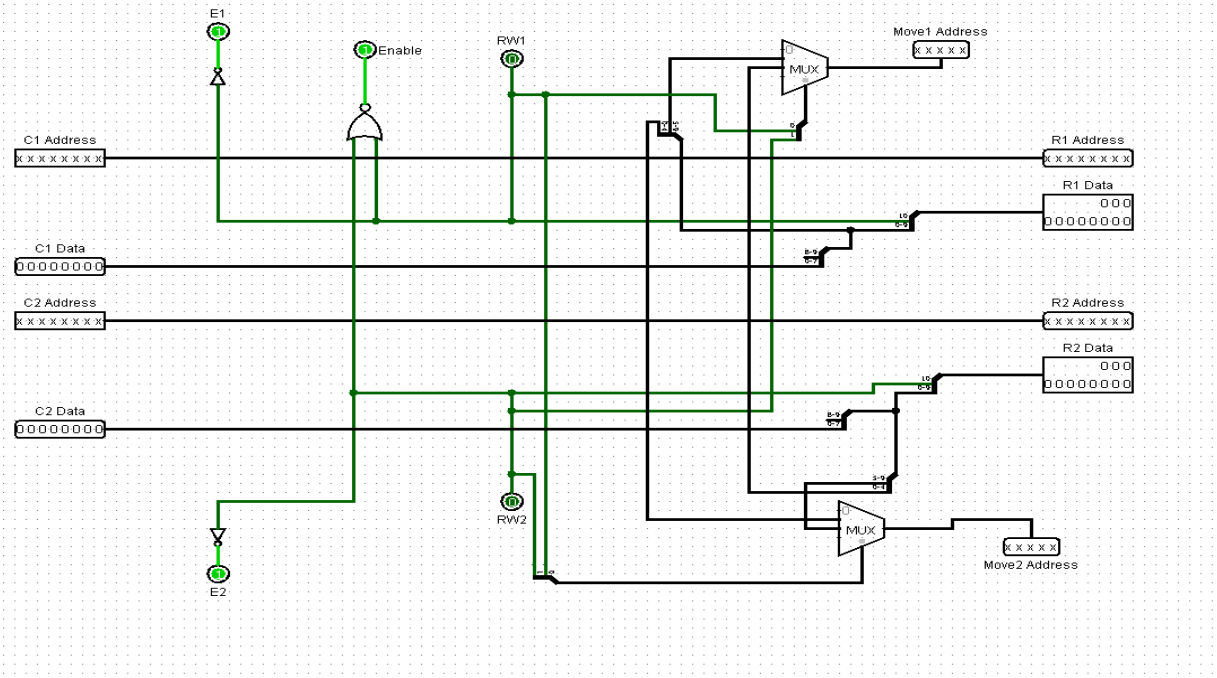


Figure 8 The CMU Unit implementation

To be able to implement the CMU unit, we used 2 MUXs that determines whether the Move Address is from ROM1 or ROM 2.To determine the Enable we looked the left-most bit of the given 11-bit instruction. Because if the first bit is 1 this means Move operation so the Cores have to cease not to perform the next instruction until the Move operation is done. The input Data which consists of 8-bits are directly given to the ROMs separately.

Dual core:

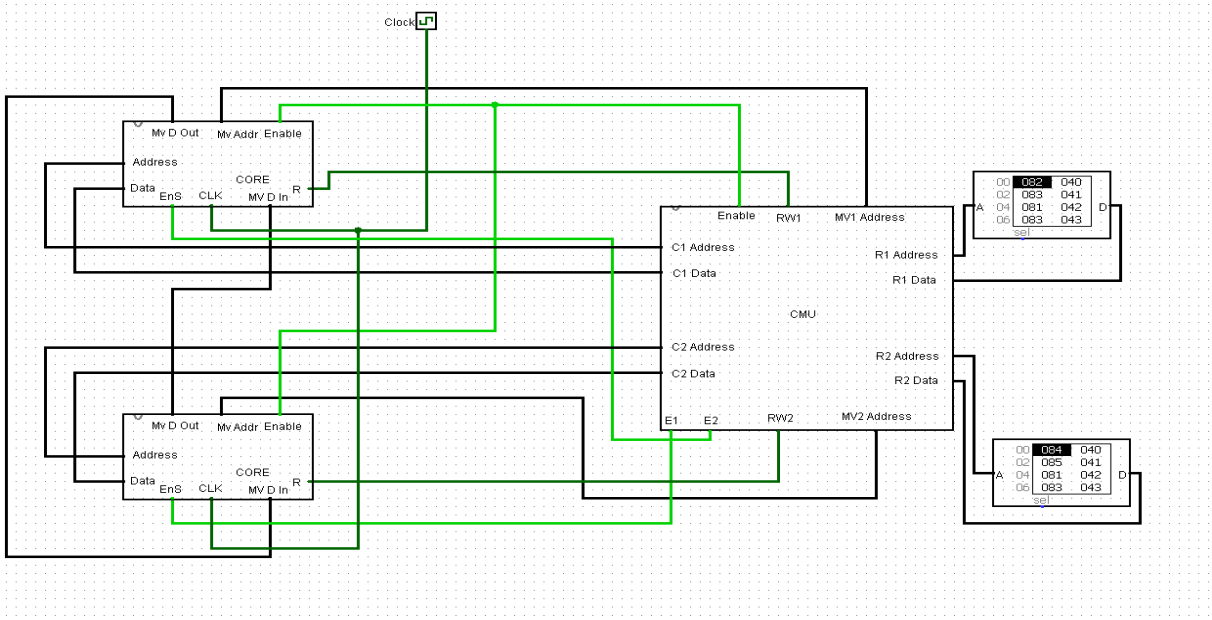


Figure 9 Dual Core Circuit

In figure 9, we used two cores and CMU which we implemented earlier, in addition to them we used 2 separate ROMs. The CMU unit controls the data bit flows simultaneously. The dual core performs the operations like this way: Firstly each ROM takes the address bits from the core and gives the instructions at the same time, then CMU unit takes the data bits and splits them to be able to decide which core is going to perform the operation. Unless there is a Move operation in one clock cycle the cores do separate operations. This cycle carries on until the stop command is given from the ROMs.

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

CMU	OP	menomic	name	operation
1xx	xxx	MV c y z	Move data to other core	RAM(other)[z] ← RAM(c)[y]
000	000	BZ c y	Branch on Zero	if ACC(c) = 0: PC(c) ← PC(c) + singed(y)
000	001	BN c y	Branch on Negative	if ACC(c) < 0: PC(c) ← PC(c) + singed(x)
000	010	ST c x	Store	RAM(c)[y] ← ACC(c)
000	011	RL c y	Rotate Left	ACC(c) ← ACC(c) (rot. lower 3 bits of y)
000	100	LI c y	Load Immediate	ACC(c) ← singed(y)
000	101	LM c y	Load from Memory	ACC(c) ← RAM(c)[y]
000	110	AD c y	Add	ACC(c) ← ACC(c) + RAM(c)[y]
000	111	SB c y	Subtract	ACC(c) ← ACC(c) - RAM(c)[y]

Table 1: Multi-Core Operations

opcode	menomic	name	operation
000	BZ x	Branch on Zero	if ACC = 0: PC ← PC + signed(x)
001	BN x	Branch on Negative	if ACC < 0: PC ← PC + signed(x)
010	ST x	Store	RAM[x] ← ACC
011	RL x	Rotate Left	ACC ← ACC (rot. lower 3 bits of x)
100	LI x	Load Immediate	ACC ← signed(x)
101	LM x	Load from Memory	ACC ← RAM[x]
110	AD x	Add	ACC ← ACC + RAM[x]
111	SB x	Subtract	ACC ← ACC - RAM[x]

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 the decimal format(we have just 2 cores).

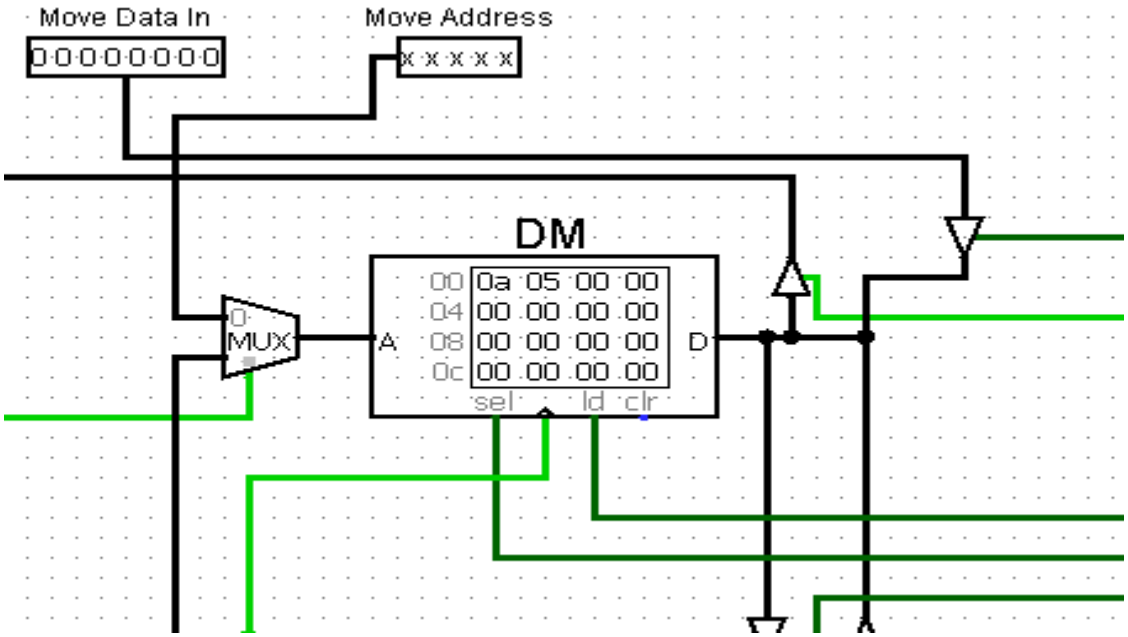
Questions&Solutions

Question 4.1:

```
Dosya  Düzen  Biçim  Görünüm  Yardım
ret EQ 0 #This is the memory location
LI 5 #Places number to be doubled into ACC
ST 1 # Store 5 into memory so it can be added .
AD 1 #Adds the memory location 1 to ACC contend.
ST ret #stores result into memory LOacation
LI 0 #Stop the Core .
BZ 0 #does not increment adress
```

Figure 10-The input Assembly File for 4.1

In this question, we wrote assembly code to our assembly and gave then output file to our ROM and perform the operations. As can be seen from figure X Single core operation done correctly. Firstly Loaded 5 to address 1 then added to 5, finally stored the solution(10 ->Hex=0a) to address zero.



4.2:

.1:

Figure 11-Ram Contents for 4.1(Single Core Operation)

Question 4.2:

4.2Second.asm - Not Defteri

Dosya Düzen Biçim Görünüm Yardım

```
LI 0 1 # Load first element of first vector t1 core 0
LI 1 2 # Load second element of first vector t1 core 1
ST X 0 #Store the element to core caches  address 0
LI 0 3 #Load the first element of second vector to core 0
LI 1 4 #Load the secondelement of second vector to core 1
AD X 0 #Add first/second elemnts of the both vectors
ST X 1 #Store the addition
MV 1 1 2 #Move the addition result from core 1 to core 0
```

Figure 12-The input assembly file of 4.2 (Dual core example)

We can easily explain the instructions by commenting on them. After giving these code to our assembly The Ram contents are matched with the solution, overall we can say that the dual-core performs the operations correctly.

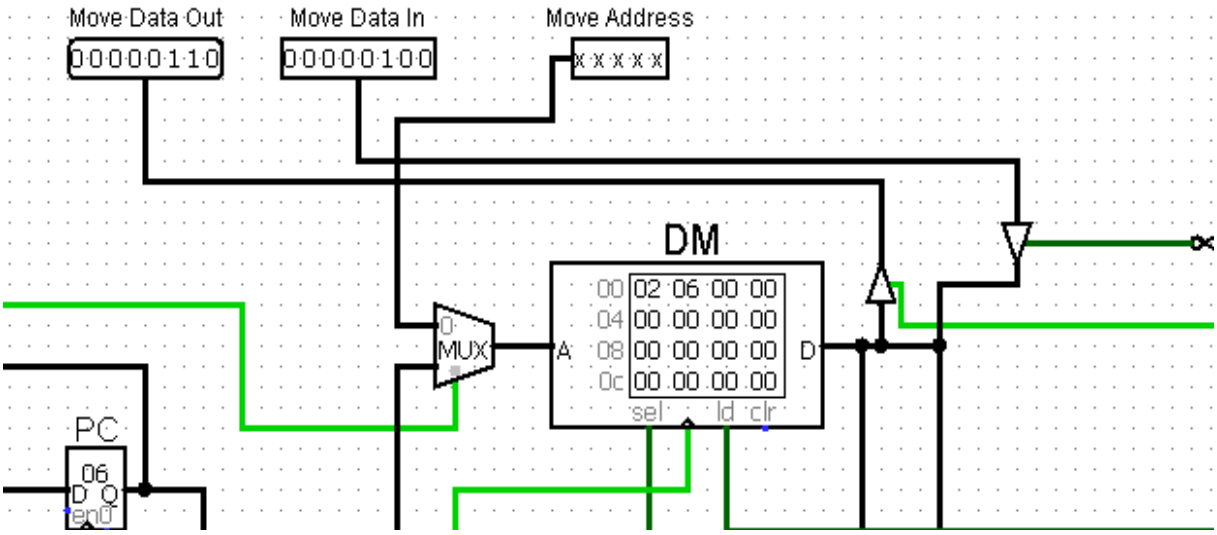


Figure 13: The Ram Content of 4.2 (Dual core example) of the first Core

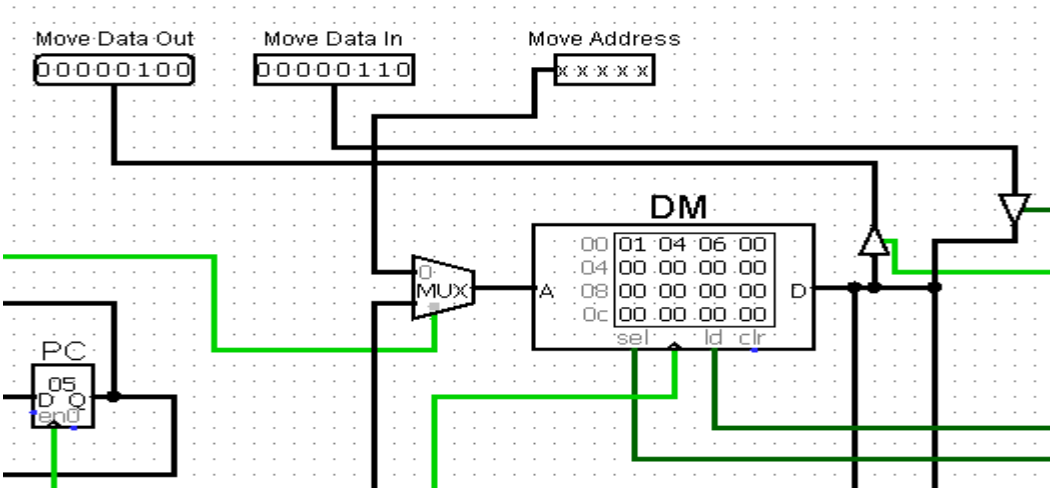


Figure 14: The Ram Content of 4.2 (Dual core example) of the Second Core

Question 5.1

```java
public class Test5p1 {
    public static void main(String[]args){
        System.out.println("Answer of 5.1 is: ");
        fivePointOne();
        System.out.println("Answer of 5.2 is:");
        fivePointTwo();
    }

    public static void fivePointOne(){
        int x = 5;
        int y = x + x;
        System.out.println("X is "+x);
        System.out.println("Y is "+y);
    }
    public static void fivePointTwo(){
        int x = 1;
        int y = 2;
        int z = x+3;
        int f = y+4;
        System.out.println("X is "+x);
        System.out.println("Y is "+y);
        System.out.println("Z is "+z);
        System.out.println("F is "+f);
    }
}
```

Test5p1 › main()

Test5p1

```
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
Answer of 5.1 is:
X is 5
Y is 10
Answer of 5.2 is:
X is 1
Y is 2
Z is 4
F is 6

Process finished with exit code 0
```

Figure 15: High level codes of 5.1 and 5.2

Question 5.2

MInput5.2.txt - Not Defteri

Dosya Düzen Biçim Görünüm Yardım

```
LI 0 2 # Load 2 to first core Accumulator
ST 0 0 # Store 2 into first cores first place
LI 0 3 # Load 3 to first core Accumulator
ST 0 1 # Store 3 into first cores second place
LI 1 4 # Load 4 to second core Accumulator
ST 1 0 # Store 4 into first cores first place
LI 1 5 # Load 5 to second core Accumulator
ST 1 1 # Store 5 into first cores second place
LI X 1 # Load 1 to both cores Accumulators
ST X 2 # Store 1 into both cores third place
LI X 3 # Load 3 to both cores Accumulators
ST X 3 # Store 3 into both cores fourth place
LM 0 1 # Load from second memory of first core to Accumulator
RL 0 1 # Multiply first cores Accumulator value by 2
AD 0 1 # Add first cores second place to its Accumulator it is like multiply by 3 with above instruction
AD 0 0 # Add first cores first place to its Accumulator
ST 0 4 # Store the first cores Accumulator value into its fifth place
LM 1 1 # Load second cores second place into its Accumulator
RL 1 1 # Multiply second cores Accumulator value by 2
AD 1 1 # Add second cores second place to its Accumulator it is like multiply by 3 with above instruction
AD 1 0 # Add second cores first place to its Accumulator
ST 1 4 # Store second cores Accumulator value into its fifth place
MV 1 4 5 # Move fifth place of second core into first cores sixth place
LI X 0 # Load both cores accumulator 0 to finish program
BZ X 0 # Make branch on zero on both cores to finish program
```

Figure 16: Input file of Question 5.2

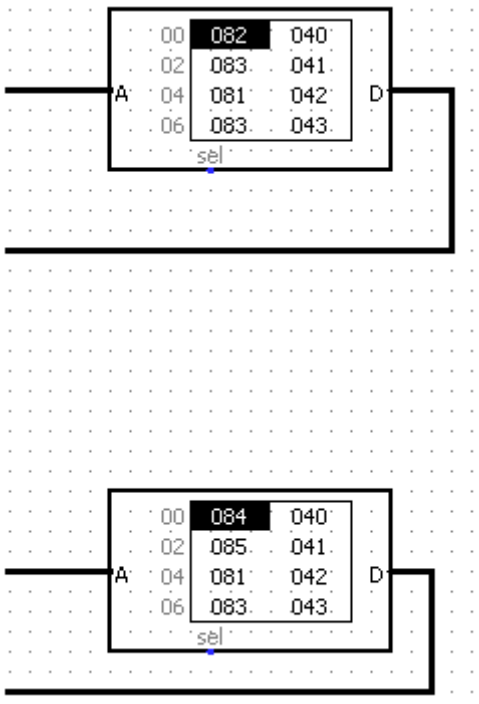


Figure17: Beginning of Instructions

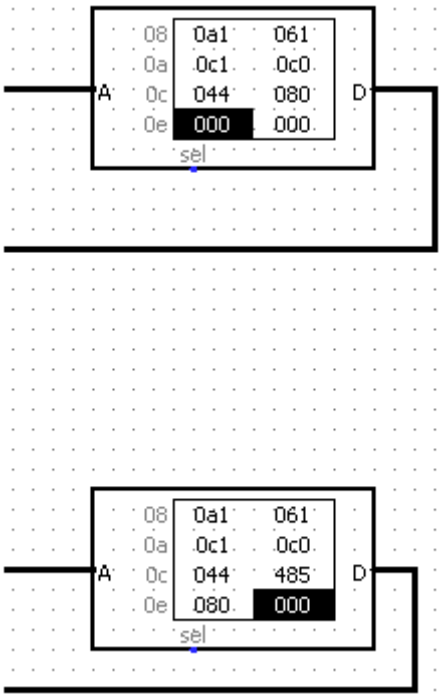


Figure 18: End of Instructions

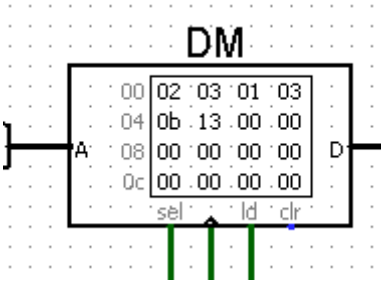


Figure 19: First cores RAM after instructions

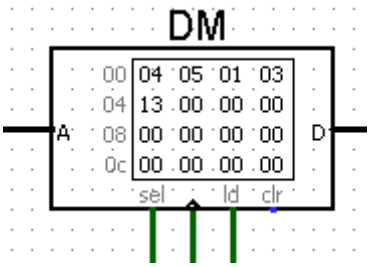


Figure 20: Second cores RAM after instructions

Contribution

As a contribution of this project we learned how to design a Dual Core CPU, what are the advantages of using a Dual Core CPU, how CMU works and the most importantly we learned that how we can implement our own microarchitecture.