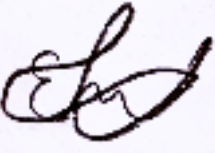


COMPUTER ARCHITECTURE

COMP 206


Project: Dual-Core Computer Design



Ertuğrul
Yılmaz



İlknur ATALAY



Yasemin
İzel
Çangal

ERTUĞRUL YILMAZ - 041701030

YASEMİN İZEL ÇANGAL - 041701021

İLKNUR ATALAY - 041701026

Introduction

First, we understood exactly what we should do in the project and we thought of a solution. Firstly, we design the datapath by drawing the appropriate truth tables. Then, we implement our 16 bit ALU. The ALU is the mathematical brain of a computer. It is a digital circuit that provides arithmetic and logic operation. After successfully performing these two operations, we design a single-core and dual-core. Then, we connect two cores and design CMU. While designing the CMU, we took the figure and table attention given to us. After that, we wrote the appropriate java codes for a single-core test and dual-core test. Our project successfully the tasks requested of us.

Objective

First of all, we designed our 16 bit ALU and datapath. ALU is a digital circuit used to perform arithmetic and logic operations. Our ALU successfully implemented the instructions requested from us. We used 1 adder, 1 subtractor, 1 multiplier, 1 divider, 1 negator, 2 shifters (1 logical shift right, 1 logical shift left) and we used or, and, not, xor gate. There are 2 inputs from ACC and Ram to ALU. These are 16 bits.

Arithmetic Logic Unit (ALU)

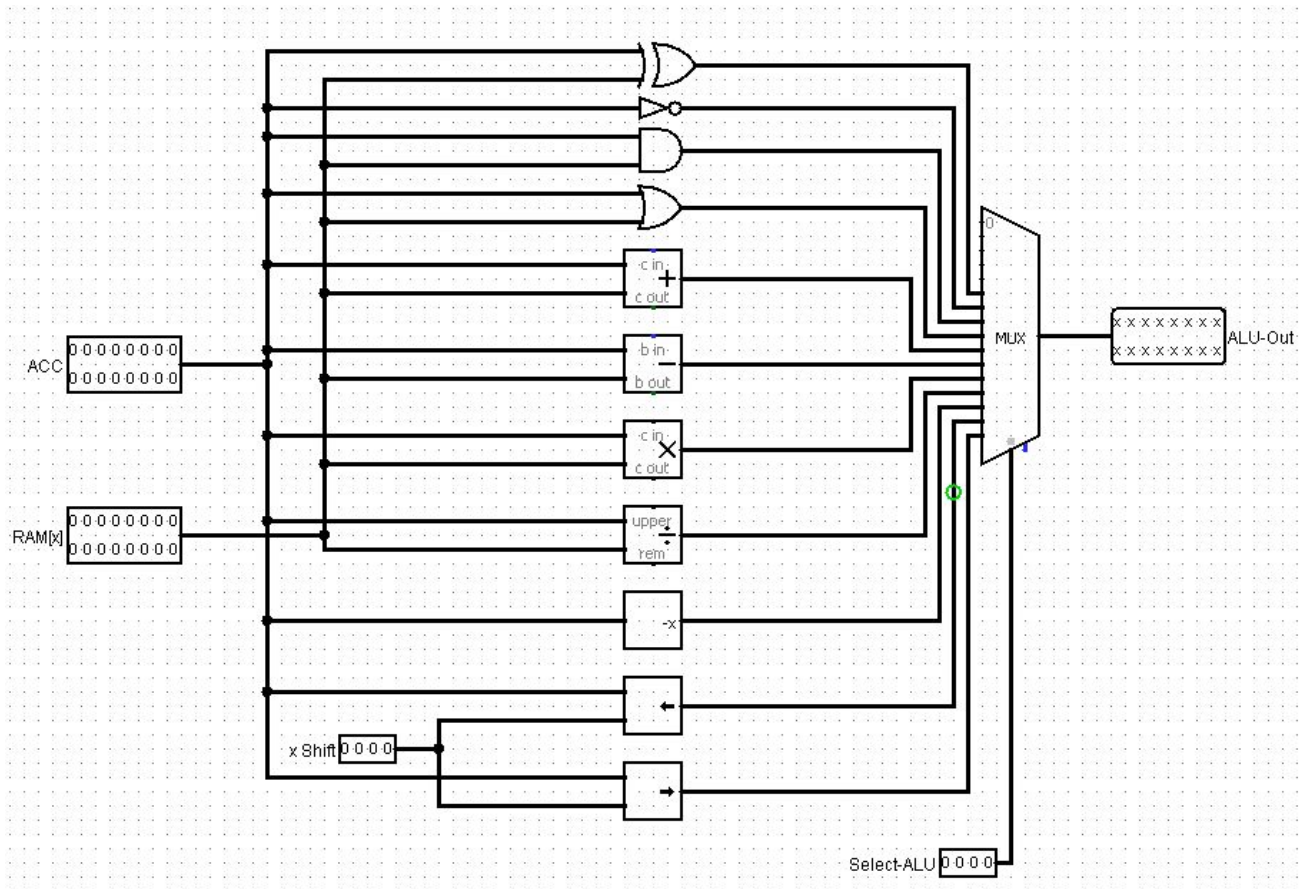


Figure 1: ALU

In the Arithmetic Logic Unit, we had two inputs for an ALU operation. First operand is ACC value and the second operand is RAM[x] value. Our select signals for ALU is same as opcodes because we got 11 different arithmetic or logic operations for this project. For eleven operation we need 4 bits to select because with 3 bits we could just select 8 different operation that is the why we need 4 bit control signals for the ALU. We are selecting the ALU operation which we need by using a 16x4 multiplexer.

Control Unit:

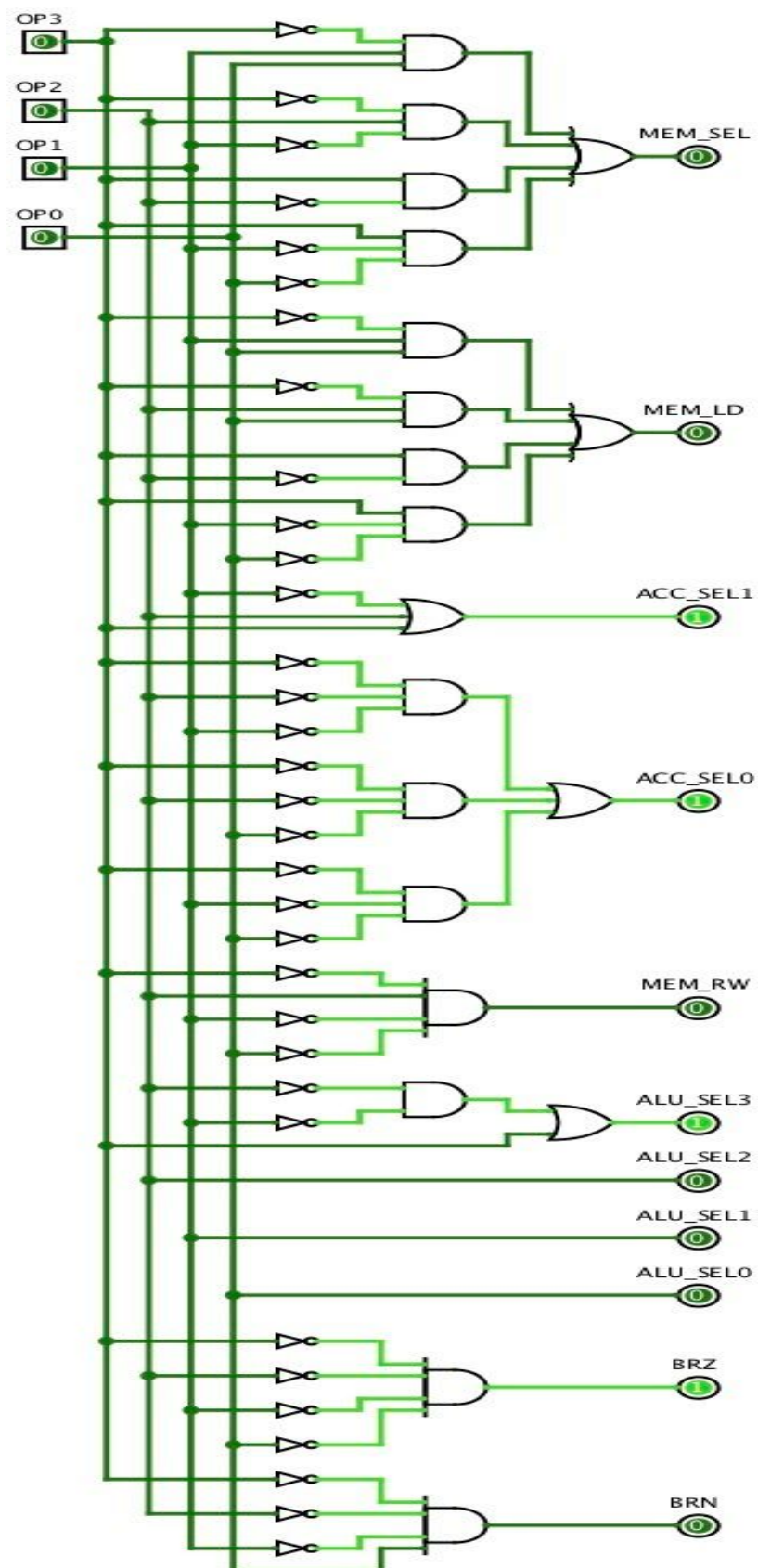


Figure 2 : Control Unit

In the control unit first we specified what will be our control signals. After specifying the control signals. We draw a truth table according to the project ISA. Control unit has 4 inputs OP0, OP1, OP2, OP3 and the outputs are MEM_SEL, MEM_LD, ACC_SEL1-0, MEM_RW, ALU_SEL3-0, BRZ and BRN.

MEM_SEL : Decides RAM will work or not.

MEM_LD : Decides RAM is going to make a read operation or write operation.

ACC_SEL1-0 : Decides which value is going to write to ACC.

MEM_RW : Decides is there write operation for the RAM.

ALU_SEL3-0 : Decides which ALU operation will be given as output.

BRZ : Decides if instruction is a Branch on Zero operation.

BRN : Decides if instruction is a Branch on Negative operation.

We considered this truth table when designing the control unit. We have 4 input and 11 output.

OP3	OP2	OP1	OP0	MEM_SEL	MEM_LD	ACC_SEL1	ACC_SEL0	MEM_RW	ALU_SEL3	ALU_SEL2	ALU_SEL1	ALU_SEL0	BRZ	BRN
0	0	0	0	0	0	1	1	0	1	0	0	0	1	0
0	0	0	1	0	0	1	1	0	1	0	0	1	0	1
0	0	1	0	0	0	0	1	0	0	0	1	0	0	0
0	0	1	1	1	1	0	0	0	0	0	1	1	0	0
0	1	0	0	1	0	1	1	1	0	1	0	0	0	0
0	1	0	1	1	1	1	0	0	0	1	0	1	0	0
0	1	1	0	0	0	1	0	0	0	1	1	0	0	0
0	1	1	1	1	1	1	0	0	0	1	1	1	0	0
1	0	0	0	1	1	1	0	0	1	0	0	0	0	0
1	0	0	1	1	1	1	0	0	1	0	0	1	0	0
1	0	1	0	1	1	1	0	0	1	0	1	0	0	0
1	0	1	1	1	1	1	0	0	1	0	1	1	0	0
1	1	0	0	1	1	1	0	0	1	1	0	0	0	0
1	1	0	1	0	0	1	0	0	1	1	0	1	0	0
1	1	1	0	0	0	1	0	0	1	1	1	0	0	0
1	1	1	1	0	0	1	0	0	1	1	1	0	0	0
1	1	1	1	0	0	1	0	0	1	1	1	1	0	0

Figure 3 :Truth Table for Control Unit

CMU:

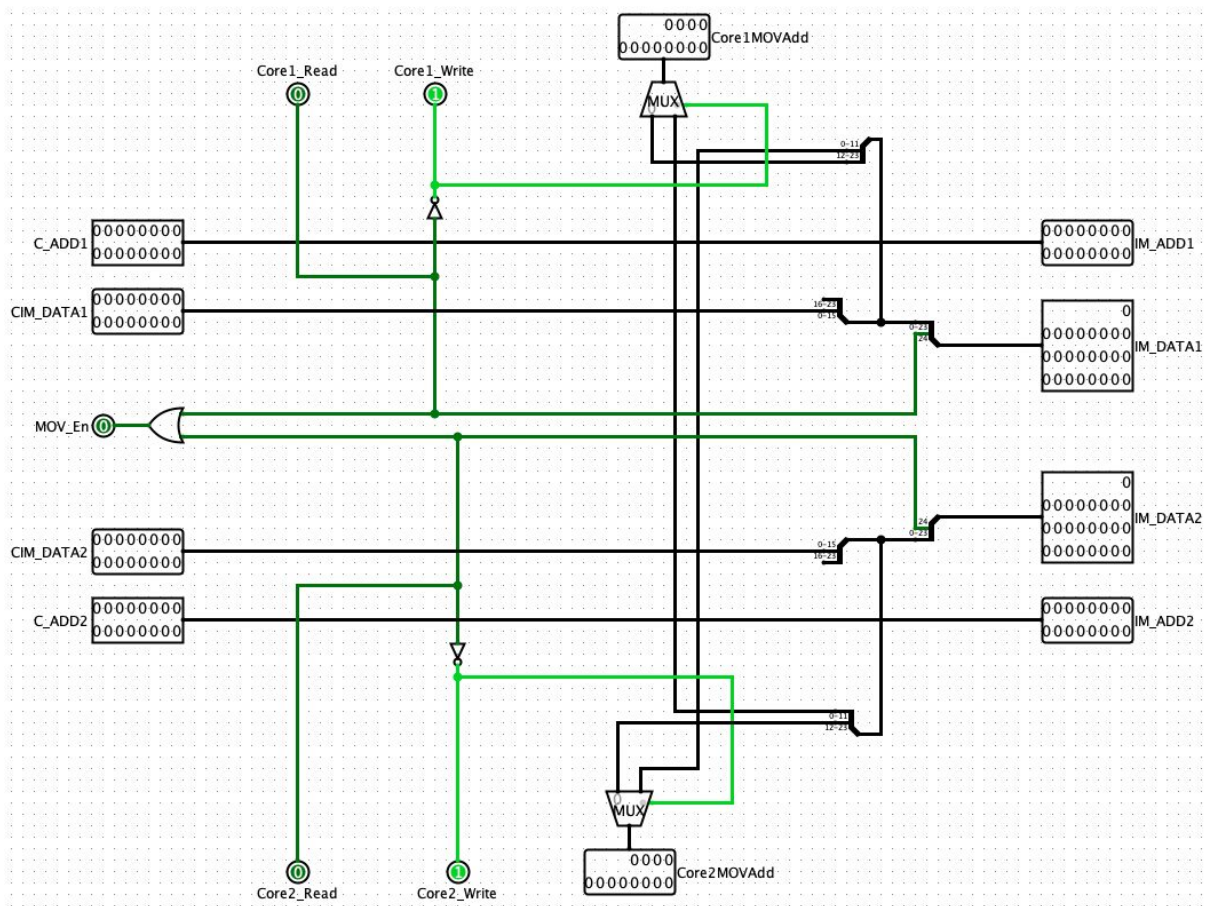


Figure 4 : CMU

In CMU we are taking instructions as 25 bits long. First we are deciding whether the coming instruction is a MOV or not. If it is a MOV operation we are giving the signals which core is going read and which core is going to write. With these signals we are deciding which core's program counter is going to be incremented and which is going to be the same. If it is not a MOV operation we are giving the instructions as normal 16 bits to the cores. It is taking the addresses from cores and giving the datas to cores. Also giving to enable signal for both cores if there is a MOV operation.

CORE:

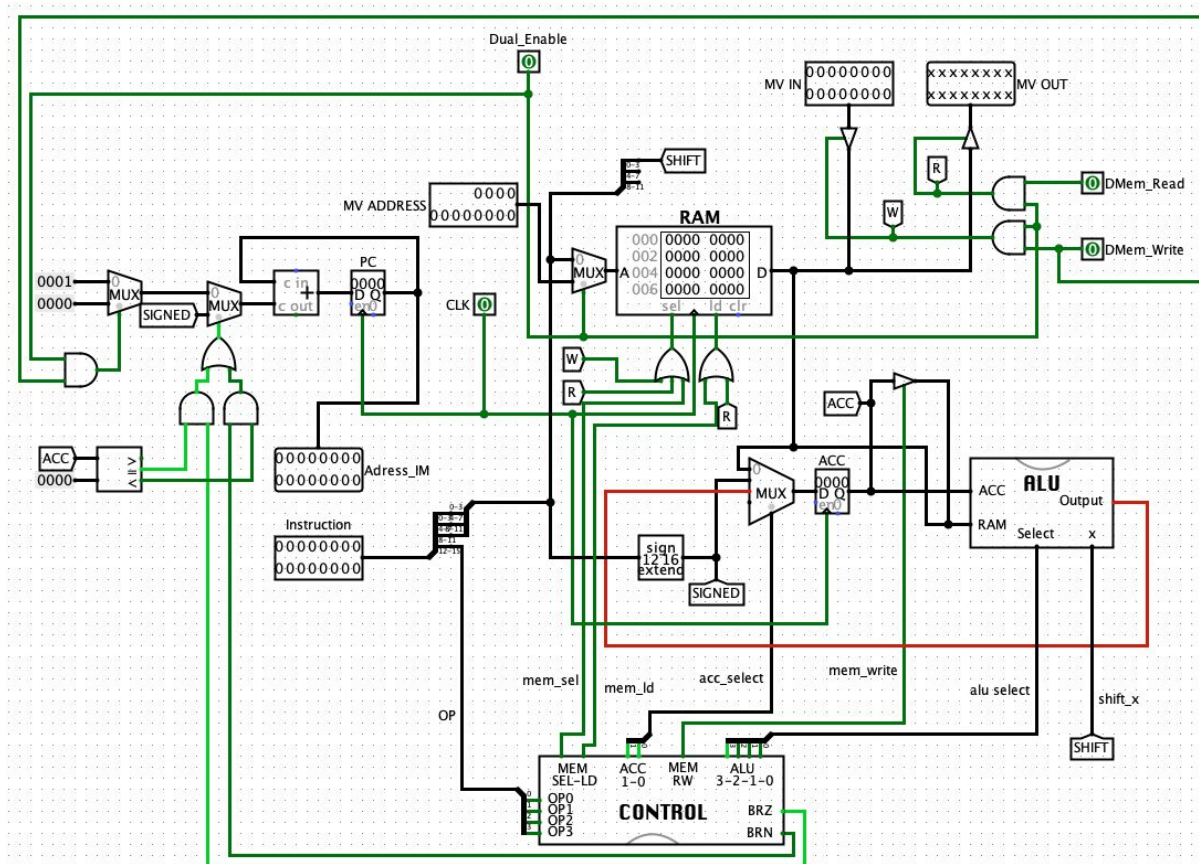


Figure 5 : Core

That is our core architecture, we have a multiplexer at top left corner that is for deciding if there is a branch and adding the signed(x) to the program counter if branch condition is true. In the middle left there is a comparator which checks the branch conditions. The top middle multiplexer is deciding if there is dual core MOV operation and it's select is Dual_Enable which is coming from the CMU. The middle right multiplexer decides which value will be written the ACC and it's select is coming from the control unit. At the top right corner you can see the addresses for a dual core operation. For a dual core operation, if the current core is making read from RAM we are incrementing it's program counter by using the signals DMem_Read at top right and using the multiplexer at top left corner.

DUAL CORE:

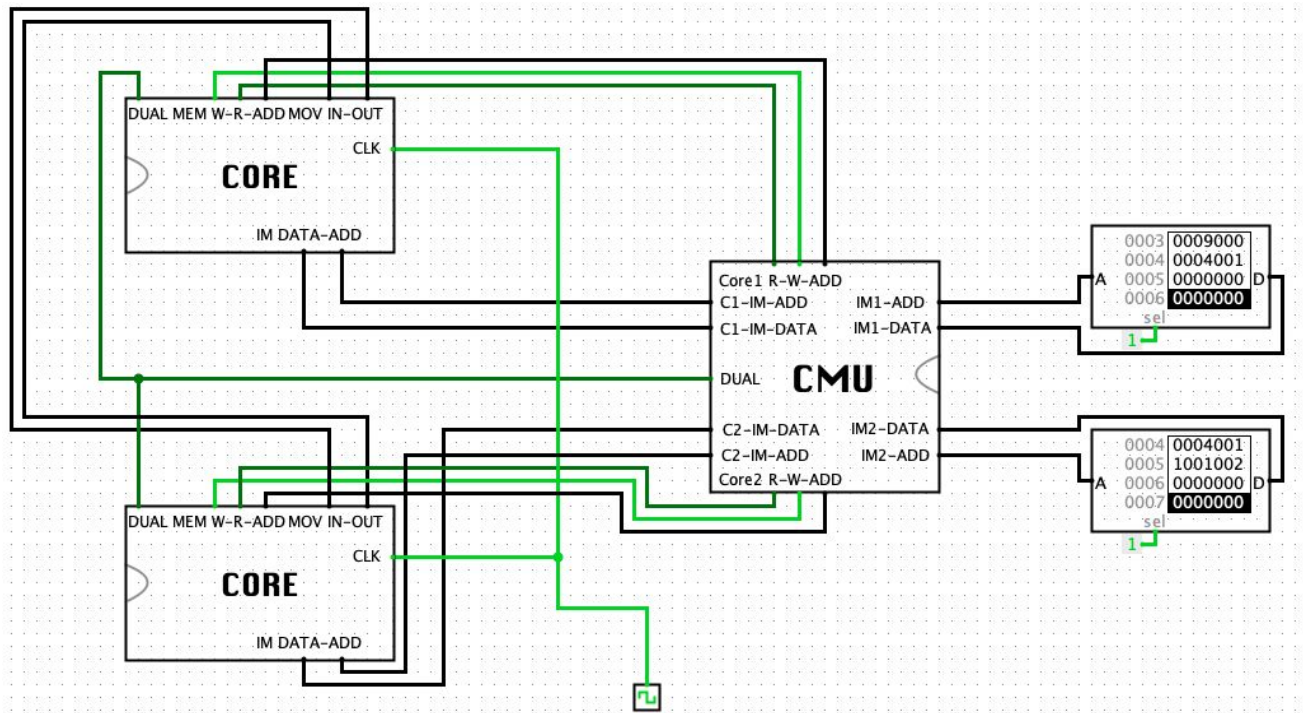


Figure 6 : Dual Core CPU

In this figure you can clearly see the connection between two cores with the help of the Core Management Unit(CMU). Both ROMs are connected to CMU. The Core Management Unit is like a filter for instructions it decides is dual core operation or not for an instruction. Both cores are getting correct information from the CMU and sending the addresses that they want to CMU.

ASSEMBLER

We wrote our Assembler in Java. Our assemblers can perform many different operations. You can see these operations in the tables below. Assembler part of the project was about taking file input from a txt which includes instructions. These instructions have opcodes which are in binary forms as 4 bits. Also, while reading the file, some eliminations have been done like comments etc. During these eliminations, immediate value and number of dual core, which tells which dual core is this, gathered in variables. Then, functions are called to convert these values into hex forms. Finally, the results are gathered in arraylist(s) and shown in output files.

Table 1: ISA for the cores.

opcode	menomic	name	operation
0000	BRZ x	Branch on Zero	if $ACC = 0$: $PC \leftarrow PC + \text{signed}(x)$
0001	BRN x	Branch on Negative	if $ACC < 0$: $PC \leftarrow PC + \text{signed}(x)$
0010	LDI x	Load Immediate	$ACC \leftarrow \text{signed}(x)$
0011	LDM x	Load from Memory	$ACC \leftarrow RAM[x]$
0100	STR x	Store	$RAM[x] \leftarrow ACC$
0101	XOR x	Bitwise XOR	$ACC \leftarrow ACC \wedge RAM[x]$
0110	NOT x	Bitwise NOT	$ACC \leftarrow !ACC$
0111	AND x	Bitwise AND	$ACC \leftarrow ACC \&\& RAM[x]$
1000	ORR x	Bitwise OR	$ACC \leftarrow ACC RAM[x]$
1001	ADD x	Add	$ACC \leftarrow ACC + RAM[x]$
1010	SUB x	Subtract	$ACC \leftarrow ACC - RAM[x]$
1011	MUL x	Multiply	$ACC \leftarrow ACC * RAM[x]$
1100	DIV x	Divide	$ACC \leftarrow ACC / RAM[x]$
1101	NEG x	Negate	$ACC \leftarrow -ACC$
1110	LSL x	Logical Shift Left	$ACC \leftarrow ACC$ (shift left by x times)
1111	LSR x	Logical Shift Right	$ACC \leftarrow ACC$ (shift right by x times)

Figure 7 : ISA for the cores

Table 2: Multicore CPU Instruction Table where $c \in \{0, 1, X\}$.

CMU	OP	menomic	name	operation
1	xxxx	MOV $c\ y\ z$	Move data to other core	$RAM(\text{other})[z] \leftarrow RAM(c)[y]$
0	0000	BRZ $c\ y$	Branch on Zero	if $ACC(c) = 0$: $PC(c) \leftarrow PC(c) + \text{singed}(y)$
0	0001	BRN $c\ y$	Branch on Negative	if $ACC(c) < 0$: $PC(c) \leftarrow PC(c) + \text{singed}(y)$
0	0010	LDI $c\ y$	Load Immediate	$ACC(c) \leftarrow \text{singed}(y)$
0	0011	LDM $c\ y$	Load from Memory	$ACC(c) \leftarrow RAM(c)[y]$
0	0100	STR $c\ y$	Store	$RAM(c)[y] \leftarrow ACC(c)$
0	0101	XOR $c\ y$	Bitwise XOR	$ACC(c) \leftarrow ACC(c) \wedge RAM(c)[y]$
0	0110	NOT $c\ y$	Bitwise NOT	$ACC(c) \leftarrow !ACC(c)$
0	0111	AND $c\ y$	Bitwise AND	$ACC(c) \leftarrow ACC(c) \&\& RAM(c)[y]$
0	1000	ORR $c\ y$	Bitwise OR	$ACC(c) \leftarrow ACC(c) \parallel RAM(c)[y]$
0	1001	ADD $c\ y$	Add	$ACC(c) \leftarrow ACC(c) + RAM(c)[y]$
0	1010	SUB $c\ y$	Subtract	$ACC(c) \leftarrow ACC(c) - RAM(c)[y]$
0	1011	MUL $c\ y$	Multiply	$ACC(c) \leftarrow ACC(c) * RAM(c)[y]$
0	1100	DIV $c\ y$	Divide	$ACC(c) \leftarrow ACC(c) / RAM(c)[y]$
0	1101	NEG $c\ y$	Negate	$ACC(c) \leftarrow -ACC(c)$
0	1110	LSL $c\ y$	Logical Shift Left	$ACC(c) \leftarrow ACC(c)$ (shift left by y times)
0	1111	LSR $c\ y$	Logical Shift Right	$ACC(c) \leftarrow ACC(c)$ (shift right by y times)

Figure 8 : Multicore CPU Instruction Table

QUESTION AND SOLUTION

4.1

In this question, we check the accuracy of our single core by using assembler and circuit.

Also, results are true as shown in figures. Here are our input and output files with results.

```

ret EQU 0 # This is the memory location
# where the result is stored .
LDI 5 # Places number to be squared into ACC ( assume 5 )
STR 1 # Store 5 into memory so it can be multiplied .
MUL 1
STR ret # Store result into memory location 0
LDI 0 # Stop the computer .
BRZ 0

```

Figure 9 : This code is input file for single core as in pdf

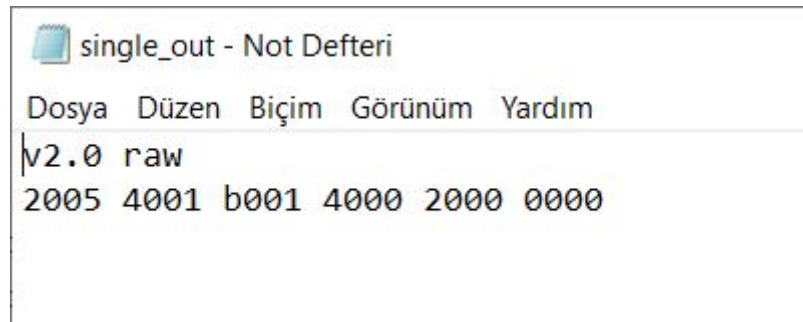


Figure 10 : This is output of single-core as in pdf

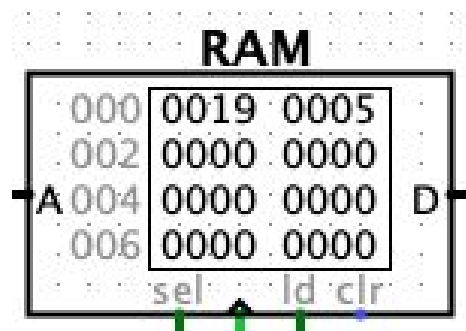


Figure 11 : This is the output of the circuit for 4.1

4.2

In this question, we check the accuracy of our dual core by using assembler and circuit. Also, results are true as shown in figures. Here are our input and output files with results.

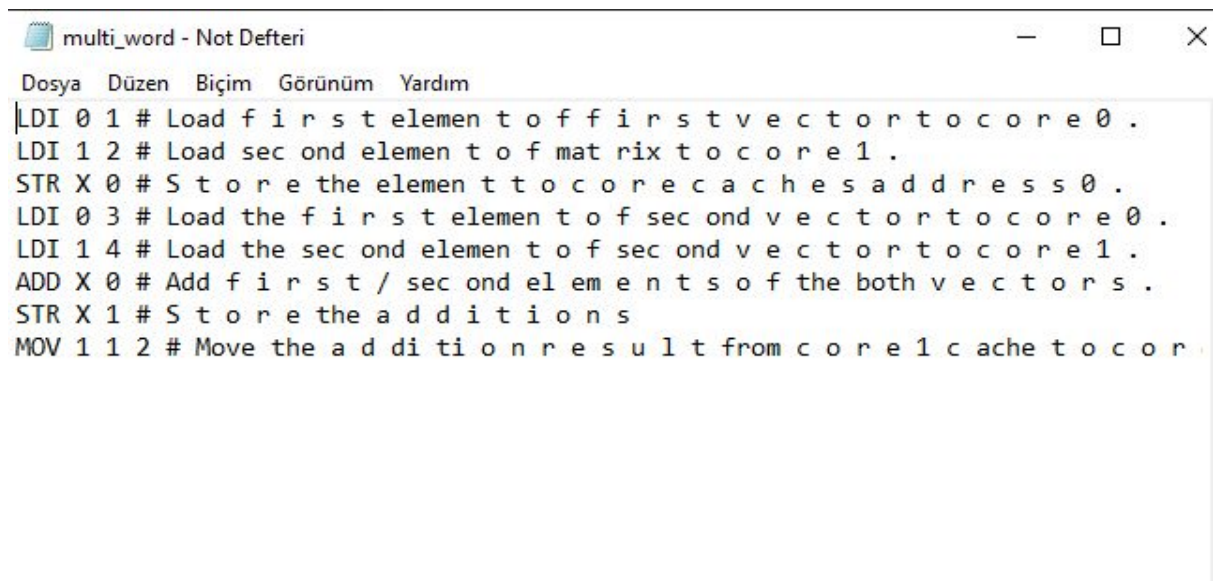


Figure 12 : This code input for multi-core

```
multi_rom0_out - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
# Core 0 ROM
v2.0 raw
2001 4000 2003 9000 4001
```

Figure 13 : This is output of multi-core file which is ROM0's

```
multi_rom1_out - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
# Core 1 ROM
v2.0 raw
2002 4000 2004 9000 4001 1001002
```

Figure 14 : This is output of multi-core file which is ROM1's

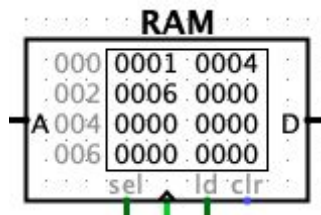


Figure 15 : This is the output of the circuit for 4.2

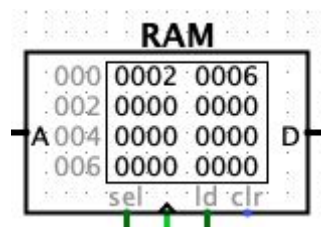


Figure 16 : This is the output of the circuit for 4.2

DEMONSTRATION

5.1 We did test cases during the presentation as below. It worked successfully.

```
test_case_1 - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
LDI 5
STR 0
LDI 4
STR 1
MUL 0
LSL 1
SUB 0
ADD 1
SUB 0
DIV 0
STR 2
```

Figure 17 : Test case 1 code as input

```
single_out - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
v2.0 raw
2005 4000 2004 4001 b000 e001 a000 9001 a000 c000 4002
```

Figure 18 : Test case1 code's output about single core

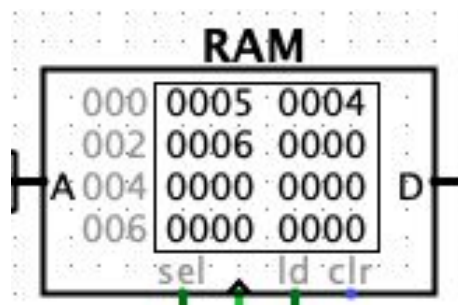


Figure 19: This is the result about test cases which we executed during presentation about single core

```
test_case_2 - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
LDI 0 1
LDI 1 2
STR X 0
LDI 0 3
LDI 1 4
ADD X 0
STR X 1
MOV 1 1 2
```

Figure 20 : Test case 2 code as input

```
multi_rom0_out - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
# Core 0 ROM
v2.0 raw
2001 4000 2003 9000 4001
```

Figure 21 : Test case 2 code's output about ROM0 about multi-core

```
multi_rom1_out - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
# Core 1 ROM
v2.0 raw
2002 4000 2004 9000 4001 1001002
```

Figure 22: Test case 2 code's output about ROM1 about multi-core

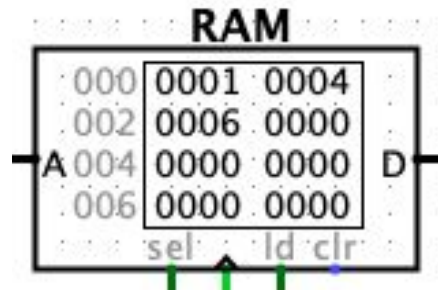


Figure 23: This is the result about test cases which we executed during presentation about multi core of ROM0

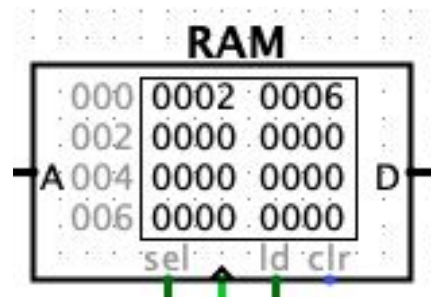


Figure 24: This is the result about test cases which we executed during presentation about multi core of ROM1

Lastly, we explained all answers which are assembler parts and circuit parts and all cases are working successfully for each part. Everyone had a great job and worked as a group during the project and actually, we didn't face any problem that much.

Labor Authentication

Single-core part done as a group. Yasemin and İlknur did the assembler part and Ertuğrul did the Dual-core and CMU part.