KUZEY KIBRIS KAMPÜSÜ
ODTÜ
METU
NORTHERN CYPRUS CAMPUS

TERM PROJECT #3 REPORT

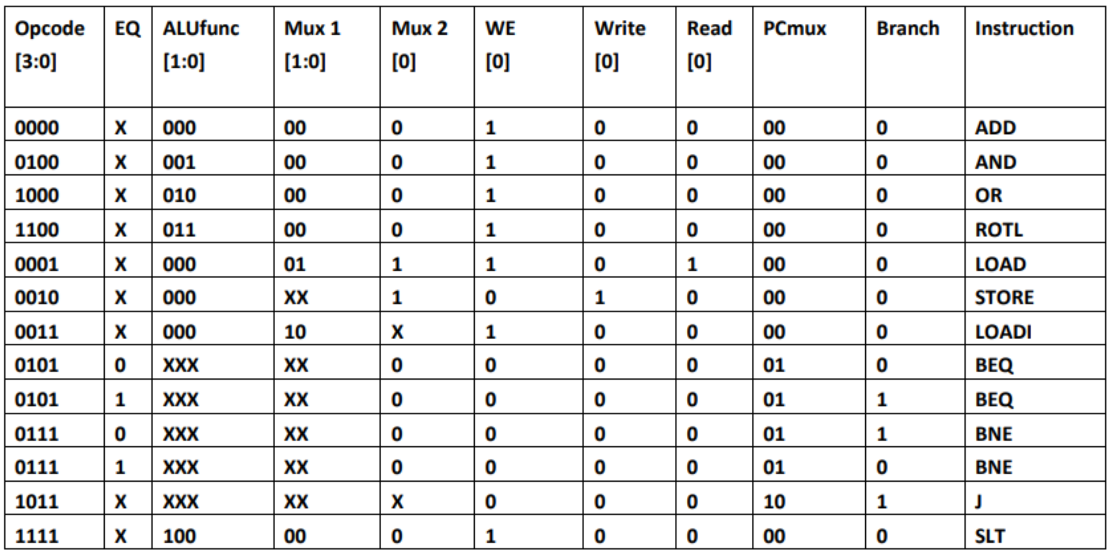
Berke Diler

2401503

The Control Unit:

The control unit was designed accoring to the truth table given below with Opcode and EQ are taken as inputs. The logic unit is meant to be the brain of the processor, in other words, it decides which operation is going to be done by each component by sending them signals.

Table 2.7: Control unit table



Opcode [3:0]	EQ	ALUfunc [1:0]	Mux 1 [1:0]	Mux 2 [0]	WE [0]	Write [0]	Read [0]	PCmux	Branch	Instruction
0000	X	000	00	0	1	0	0	00	0	ADD
0100	X	001	00	0	1	0	0	00	0	AND
1000	X	010	00	0	1	0	0	00	0	OR
1100	X	011	00	0	1	0	0	00	0	ROTL
0001	X	000	01	1	1	0	1	00	0	LOAD
0010	X	000	XX	1	0	1	0	00	0	STORE
0011	X	000	10	X	1	0	0	00	0	LOADI
0101	0	XXX	XX	0	0	0	0	01	0	BEQ
0101	1	XXX	XX	0	0	0	0	01	1	BEQ
0111	0	XXX	XX	0	0	0	0	01	1	BNE
0111	1	XXX	XX	0	0	0	0	01	0	BNE
1011	X	XXX	XX	X	0	0	0	10	1	J
1111	X	100	00	0	1	0	0	00	0	SLT

After using K-map each control bit equation is as follows:

Alufunc[0]=Opcode[2] . Not(Opcode[1])

Alufunc[1] = Opcode[3] . Not(Opcode[1])

Alufunc[2] = Opcode[2] . Opcode[0]

Mux1[0] = Not(Opcode[1]) . Opcode[0]

Mux1[1] = Not(Opcode[3]) . Opcode[1]

Mux2 = Not(Opcode[2]) . Opcode[0] + Not(Opcode[2]) . Opcode[1]

WE= Not(Opcode[1]) . Not(Opcode[0]) + Opcode[3] . Opcode[2] + Not(Opcode[3]) . Not(Opcode[2]) . Opcode[0]

Write = Opcode[1] . Not(Opcode[0])

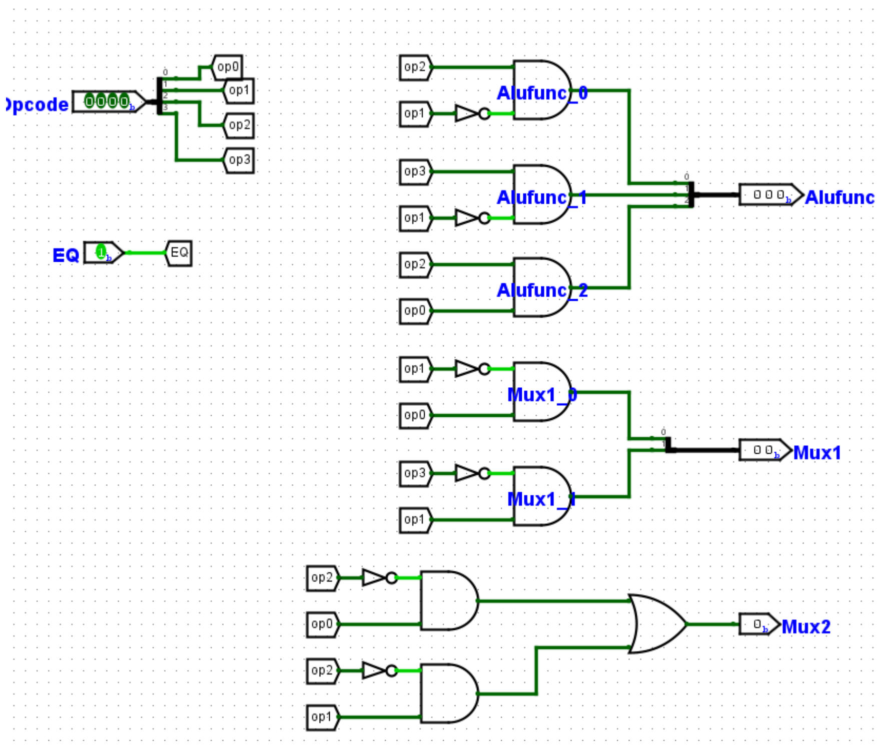
Read = Not(Opcode[2]) . Not(Opcode[1]) . Opcode[0]

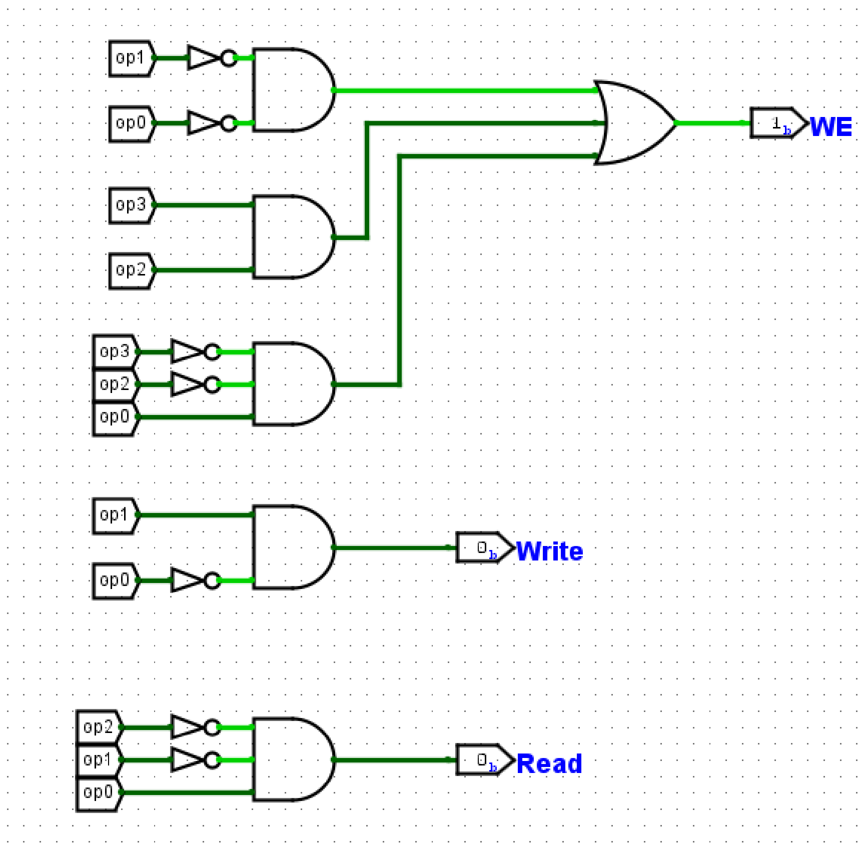
Pcmux[0] = Not(Opcode[3]) . Opcode[2] . Opcode[0]

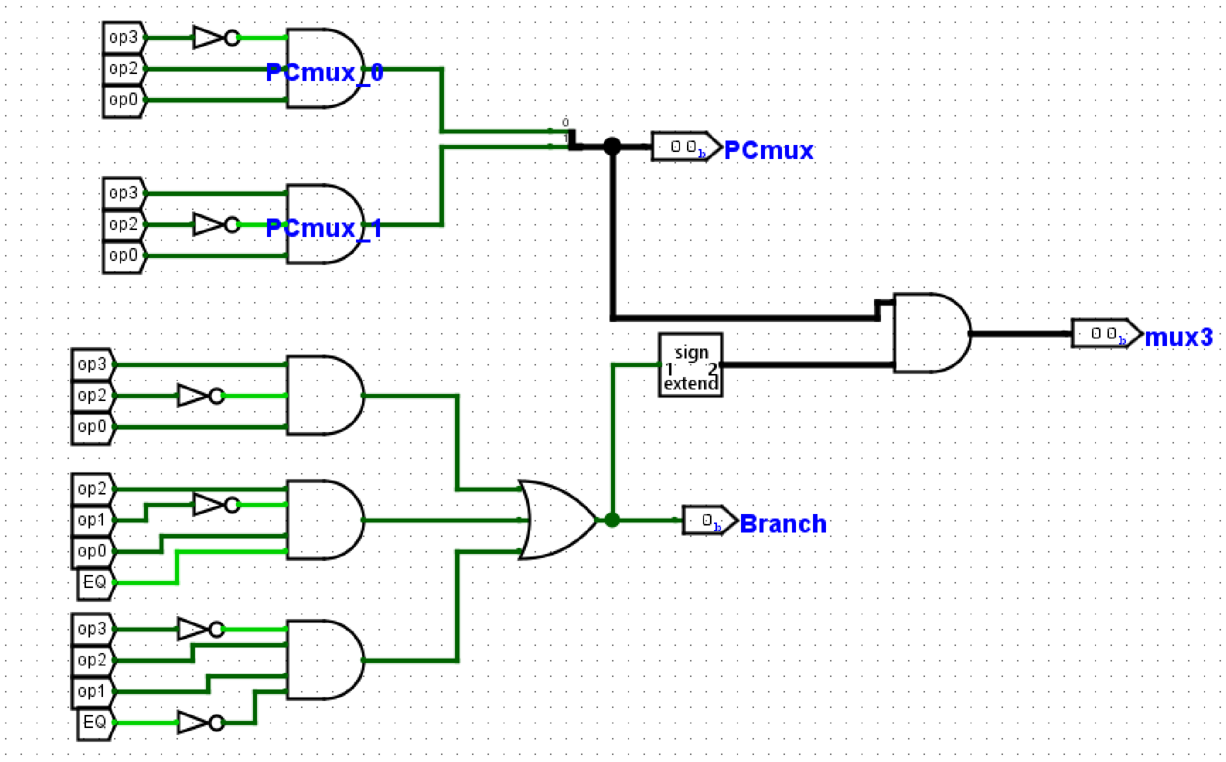
Pcmux[1] = Opcode[3] . Not(Opcode[2]) . Opcode[0]

Branch= Opcode[3] . Not(Opcode[2]) . Opcode[0] + Opcode[2] . Not(Opcode[1]) . Opcode[0] . EQ + Not(Opcode[3]) . Opcode[2] . Opcode[1] . Not(EQ)

Screenshots of the resulting control unit after the equations are implemented as logic circuitry:

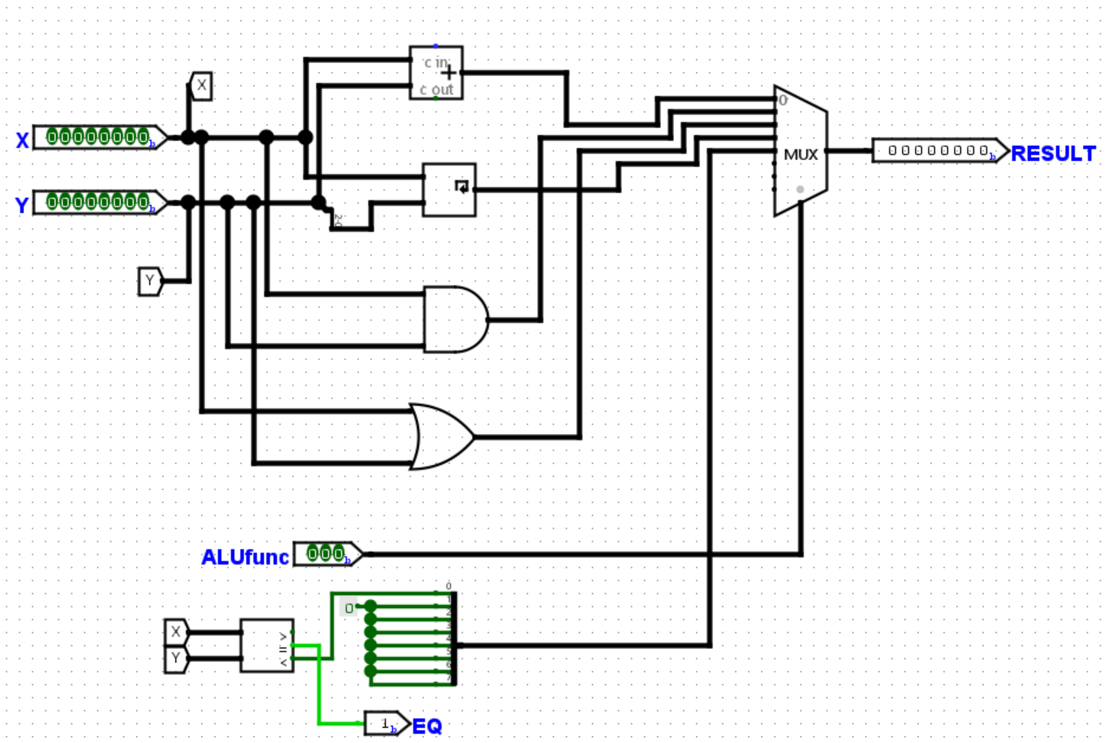






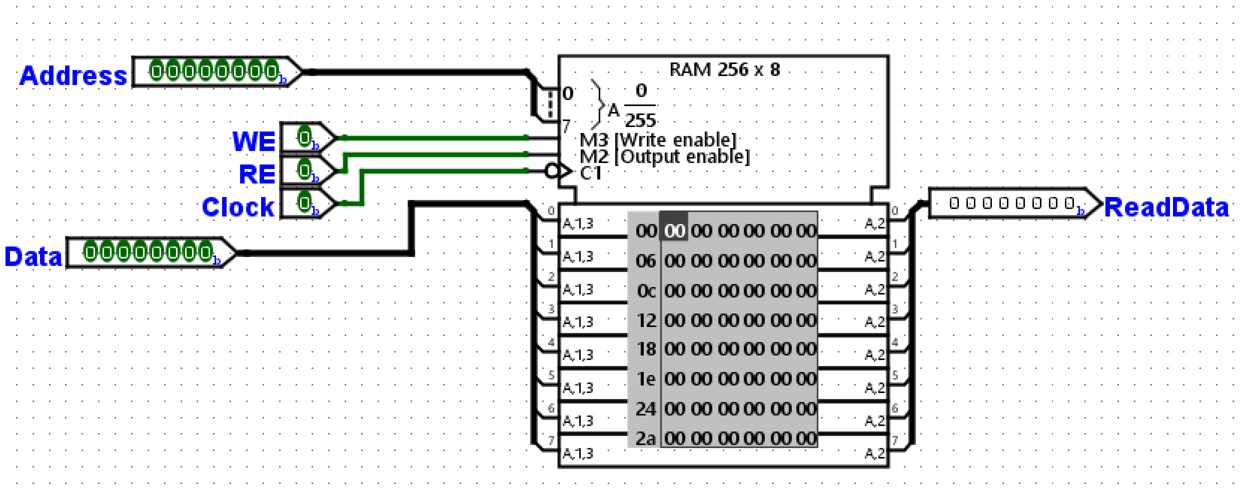
The ALU:

The ALU of the processor is responsible from every arithmetic/logic operation done, it decides which operation's result will be the output with the Multiplexer that has its control as the 'ALUfunc' which is a 3bit input sent from the control unit. In addition to the arithmetic/logic operations, there is also an output signal 'EQ' which is sent to control unit to tell if two values are equal or not, a comparator was used to implement the EQ signal, and also to implement the SLT operation.



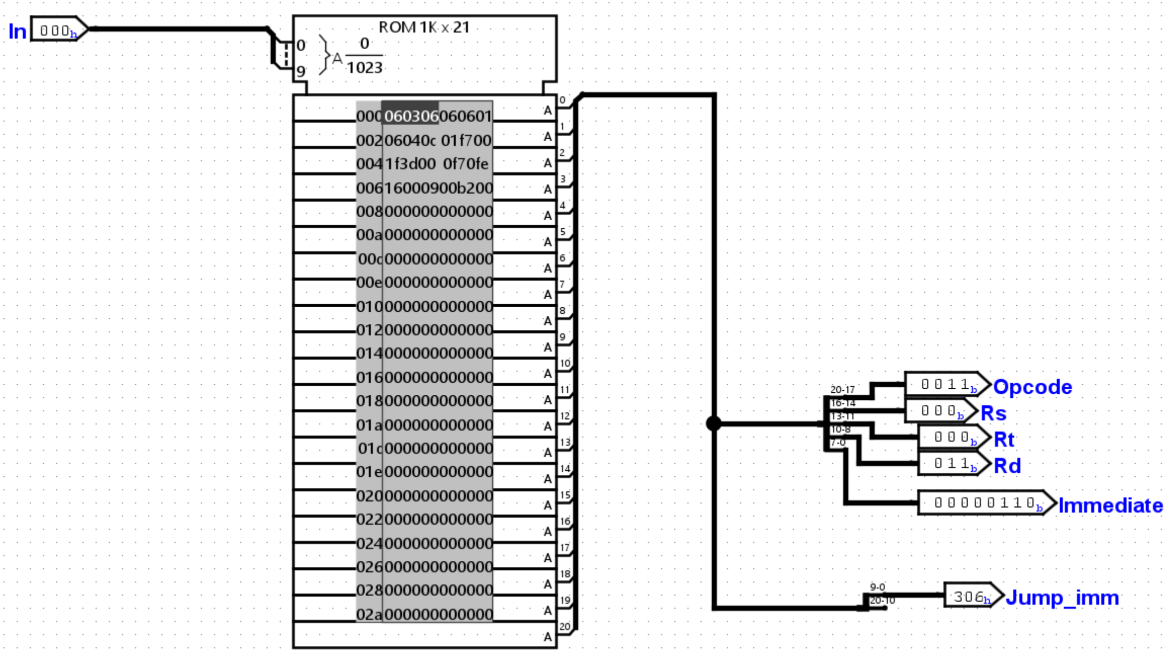
The Data Memory:

Data Memory is a part of the CPU that is responsible for storing values to the RAM using WE signal so that these values can be fetched using RE signal. The Data Memory gets the data to store from the ‘Data’ input and the address to store or read the value from the ‘Address’ input. When RE signal is 1, we get the output from the ‘ReadData’.



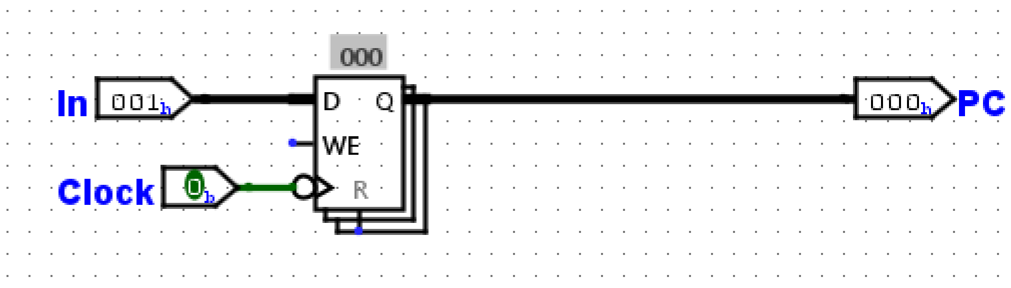
The Instruction Memory:

Instruction Memory is the part that is responsible of storing the instructions for the CPU which is built using a ROM, in each clock cycle the Instruction Memory decides which instruction to take according to the value that it recieves from the Program Counter (PC), in this specific project, each cycle the program counter increases by 1. So in every clock cycle, we take the next instruction. But if we have the jump function implemented in the CPU, then the instruction memory would fetch the jumped branch, or if we had the BEQ or BNE functions, instruction memory would take desired instruction.



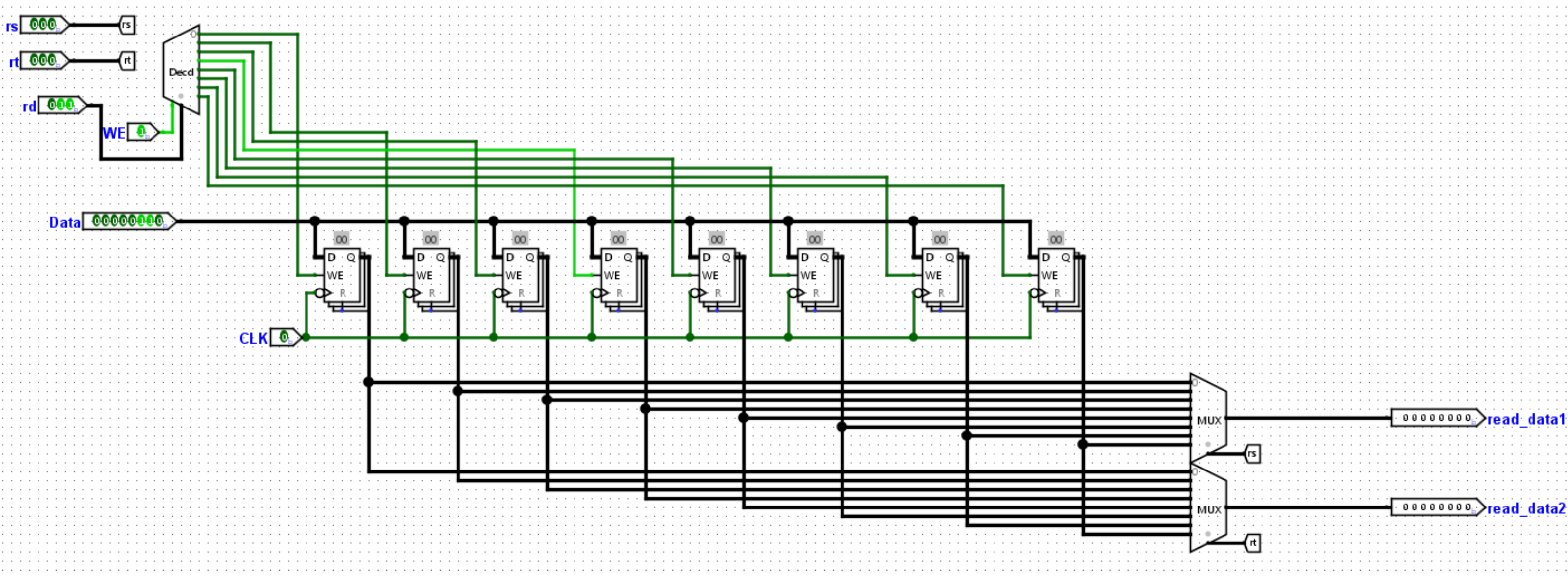
The Program Counter:

The program counter is a component in the CPU which ‘tells’ the Instruction Memory which instruction to take from the ROM each clock cycle. In the previous part of the project, we were incrementing the Program Counter by 1 in each clock cycle. But in this project, since we have J,BNE and BEQ functions in our processor, the PC gets input from the output of a Multiplexer which decides which address is going to be taken.



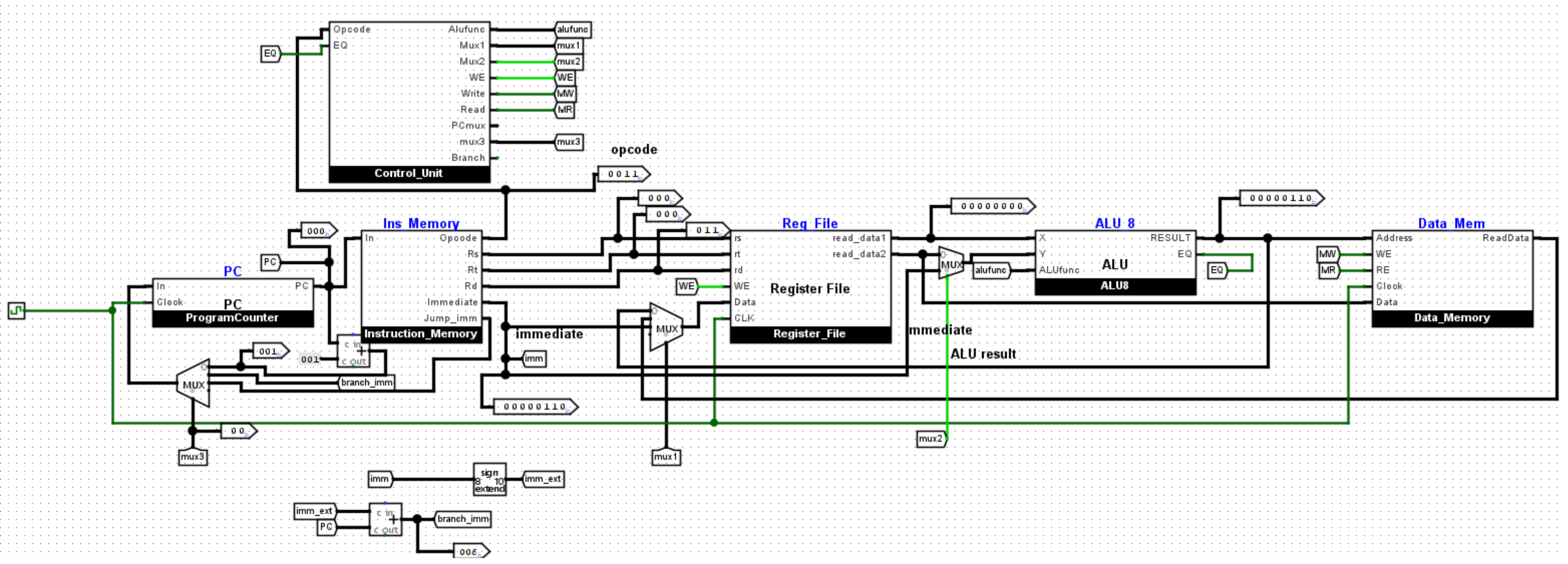
The Register File:

The register file is the component in the CPU that is responsible for getting the rs,rt,rd,WE,Data as inputs. And it either reads data from its registers or writes data to its registers according to to the inputs. If WE is not 1, then the register file reads from its registers and decides where to read the data from the rs and rt inputs which are connected to the output MUXes. If WE is 1, then the register file writes to the desired register which is decided by the input rd and the data that is written to the register is the Data input.



The Top Level Design of the CPU:

In the top level design, a mux ,which is controlled by mux3 output from the control unit, was added in order to decide which address is going to be fetched since J,BEQ and BNE functionalities are added.
In addition, the immediate value of the instruction is extended using an extender from 8 bits to 10 bits so that it could be added with PC and send to the mux.



The Test Part:

The assembly code to be tested on the CPU:

LI R3,6
LI R6,1
LI R4,12
L1:
ADD R7,R7,R6
SLT R5,R4,R7
BNE R6,R5,L1
J END
ADD R2,R2,R6
END:

Assembly code converted to machine code format then to hexadecimal line by line:
LI R3,6:
Machine Code Format: 0011 0000 0001 1000 0011 0 #loading number 6 to register 3
Hexadecimal: $(60306)_{16}$

LI R6,1:
Machine Code Format: 0011 0000 0011 0000 0000 1 #loading number 1 to register 6
Hexadecimal: $(60601)_{16}$

LI R4,12:
Machine Code Format: 0011 0000 0010 0000 0110 0 #loading number 12 to register 4
Hexadecimal: $(6040C)_{16}$

ADD R7,R7,R6:
Machine Code Format: 0000 1111 1011 1000 0000 0 #adding the values inside registers 6 and 7, then storing the result in register 7
Hexadecimal: $(1F700)_{16}$

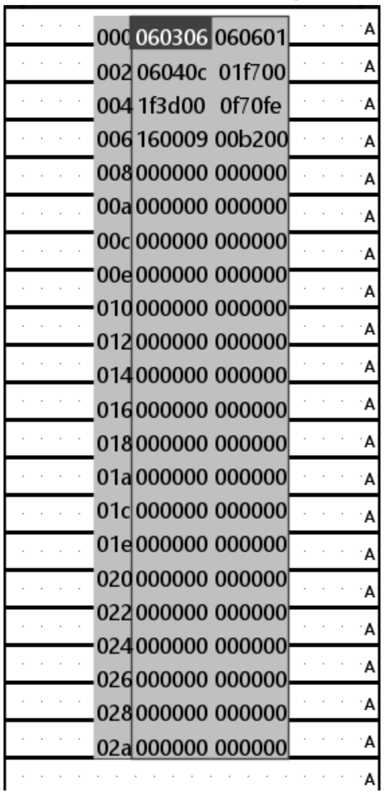
SLT R5,R4,R7:
Machine Code Format: 1111 1001 1110 1000 0000 0 #if(R4<R7), then R5 is set to 1, else R5 is set to 0
Hexadecimal: $(1F3D00)_{16}$

BNE R6,R5,L1:
Machine Code Format: 0111 1011 1000 0111 1111 0 #if(R6!=R5),then PC=PC+IMM(-2) -2 since the branch is 2 instructions before, else PC=PC+1
Hexadecimal: $(F70FE)_{16}$

J END:
Machine Code Format: 1011 0000 0000 0000 0100 1 #jump to target branch
Hexadecimal: $(160009)_{16}$

ADD R2,R2,R6:
Machine Code Format: 0000 0101 1001 0000 0000 0 #add R2 and R6, then store the value in R2
Hexadecimal: $(B200)_{16}$

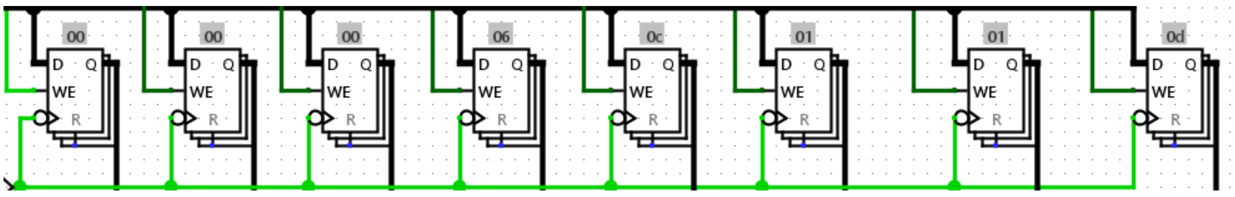
The instructions then inserted on instruction memory to be later fetched by it:



```
000 060306 060601
002 06040c 01f700
004 1f3d00 0f70fe
006 160009 00b200
008 000000 000000
00a 000000 000000
00c 000000 000000
00e 000000 000000
010 000000 000000
012 000000 000000
014 000000 000000
016 000000 000000
018 000000 000000
01a 000000 000000
01c 000000 000000
01e 000000 000000
020 000000 000000
022 000000 000000
024 000000 000000
026 000000 000000
028 000000 000000
02a 000000 000000
```

The test code iterates thorugh the loop until the value of R7 is bigger than the value of R4.

The register file after running the test:


00
00
00
06
0c
01
01
0d