Nursena Köprücü- 59843

Hasan Can Aslan- 60453

COMP303 TERM PROJECT FALL 2019

Single Cycle Processor Design

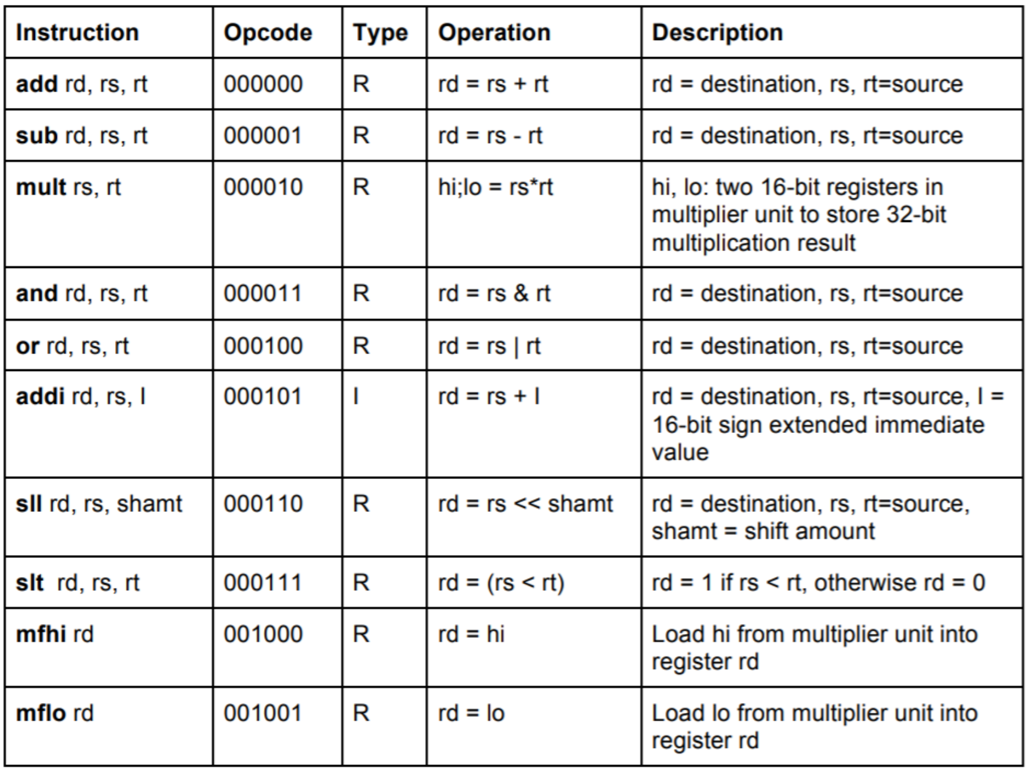
In this project, we have first designed a 16-bit single-cycle processor, implemented it by using Logisim, and tested the design whether it gives us the result as we expected.

I. The ALU Design

We have first designed and implemented our own Arithmetic Logic Unit (ALU) which supports 16-bit. However, we have 32-bit instructions as MIPS. So, the instruction formats should be as the following order:

- R-type instruction: 6-bit for opcode, 5-bit for rs, 5-bit for rt, 5-bit for rd, 5-bit for shamt, 6-bit for function
- I-type instruction: 6-bit for opcode, 5-bit for rs, 5-bit for rt, 16-bit for immediate
- J-type instruction: 6-bit for opcode, 26-bit for address

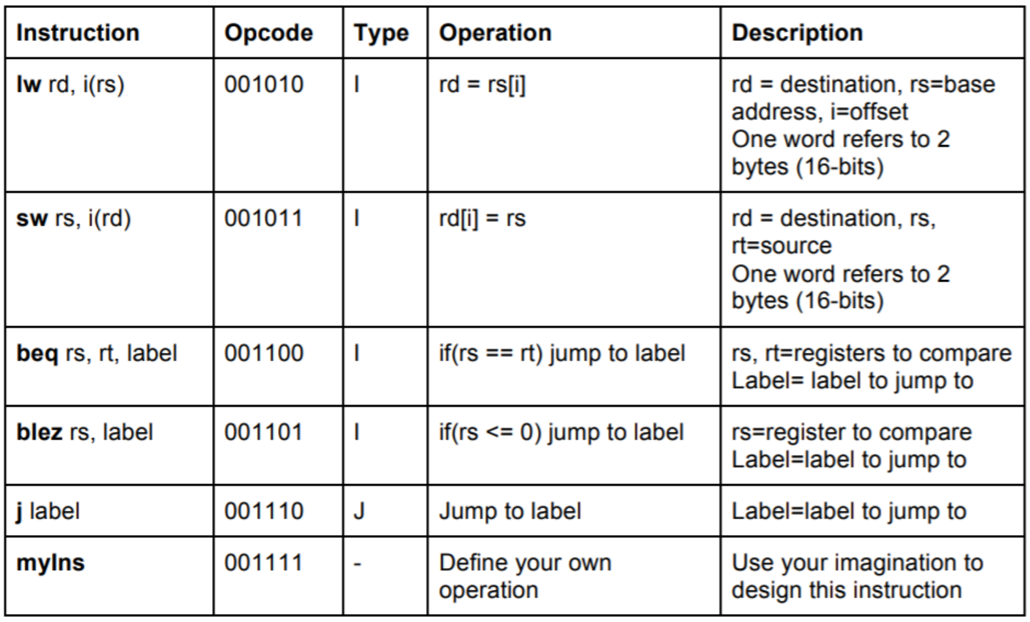
The ALU supports the instructions as listed in the table below:



Instruction	Opcode	Type	Operation	Description
add rd, rs, rt	000000	R	rd = rs + rt	rd = destination, rs, rt=source
sub rd, rs, rt	000001	R	rd = rs - rt	rd = destination, rs, rt=source
mult rs, rt	000010	R	hi;lo = rs*rt	hi, lo: two 16-bit registers in multiplier unit to store 32-bit multiplication result
and rd, rs, rt	000011	R	rd = rs & rt	rd = destination, rs, rt=source
or rd, rs, rt	000100	R	rd = rs | rt	rd = destination, rs, rt=source
addi rd, rs, I	000101	I	rd = rs + I	rd = destination, rs, rt=source, I = 16-bit sign extended immediate value
sll rd, rs, shamt	000110	R	rd = rs << shamt	rd = destination, rs, rt=source, shamt = shift amount
slt rd, rs, rt	000111	R	rd = (rs < rt)	rd = 1 if rs < rt, otherwise rd = 0
mfhi rd	001000	R	rd = hi	Load hi from multiplier unit into register rd
mflo rd	001001	R	rd = lo	Load lo from multiplier unit into register rd

Figure 1: ALU instructions

In addition to these, we have added the following load-store, branch and jump instructions as listed in the table below.



Instruction	Opcode	Type	Operation	Description
lw rd, i(rs)	001010	I	rd = rs[i]	rd = destination, rs=base address, i=offset One word refers to 2 bytes (16-bits)
sw rs, i(rd)	001011	I	rd[i] = rs	rd = destination, rs, rt=source One word refers to 2 bytes (16-bits)
beq rs, rt, label	001100	I	if(rs == rt) jump to label	rs, rt=registers to compare Label= label to jump to
blez rs, label	001101	I	if(rs <= 0) jump to label	rs=register to compare Label=label to jump to
j label	001110	J	Jump to label	Label=label to jump to
myIns	001111	-	Define your own operation	Use your imagination to design this instruction

Figure 2: Load-store, branch and jump instructions

For myIns that is shown in the table, we have implemented the instruction that computes the 1s complement of a given input. It is an I-type instruction and simply inverts all bits with its complements is bitwise. Here is the design for this instruction:

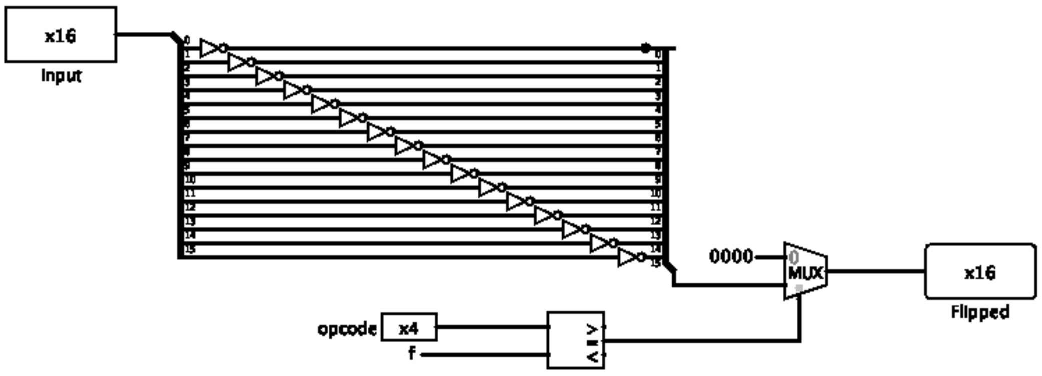


Figure 3: myIns Design

In addition, we have specified the carryout, zero, and overflow bits to be able to display them when it is necessary. In the end, we have implemented all these instructions, and the final ALU design is as the following:

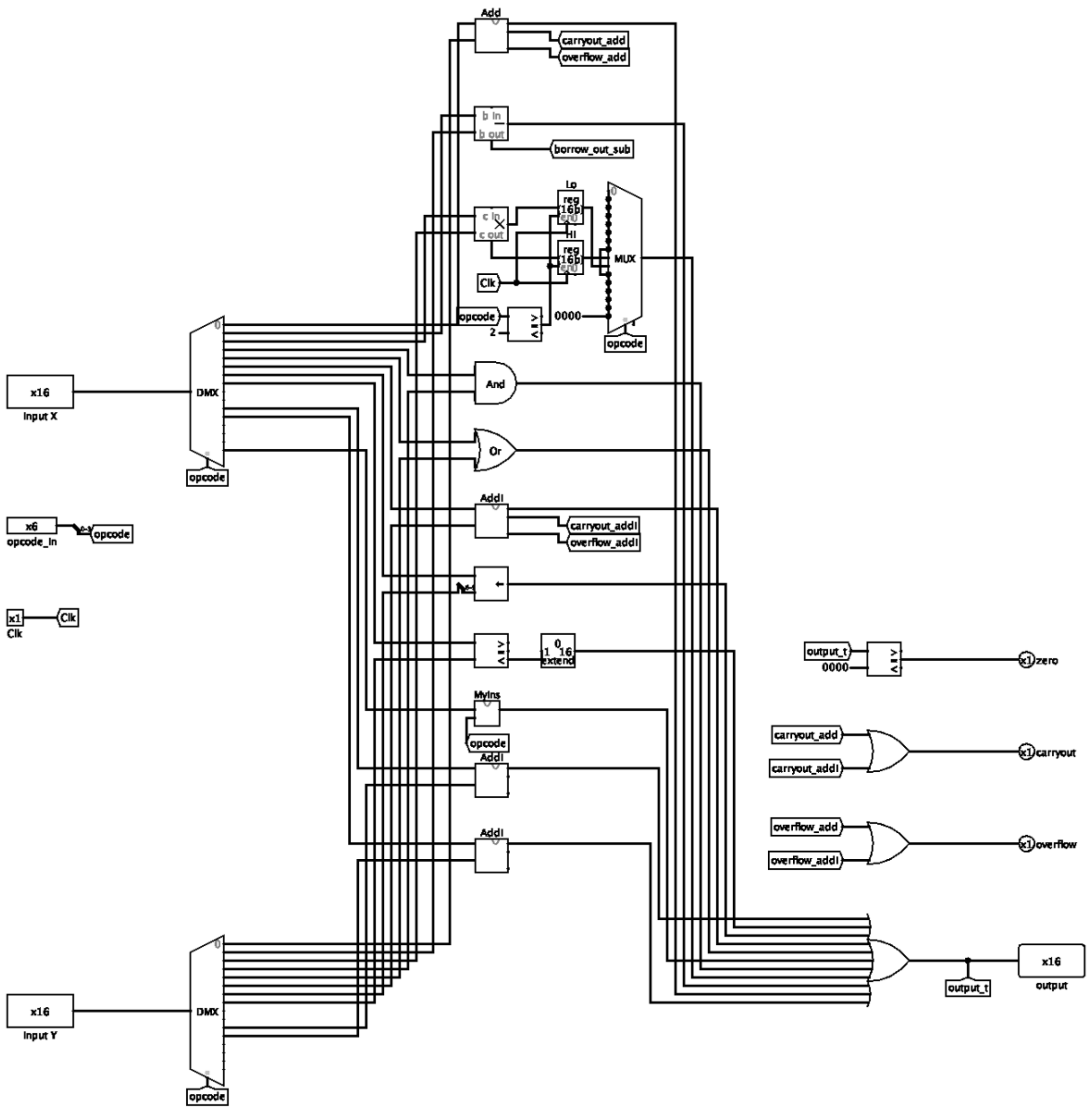


Figure 4: The ALU Design

II. Register File

We have designed our Register File by using 16 16-bit registers. It has two read register address ports, one write register address port, one write register data port, two read register output ports, a register write control signal, and a clock input.

In the end, it has the design as shown on the next page.

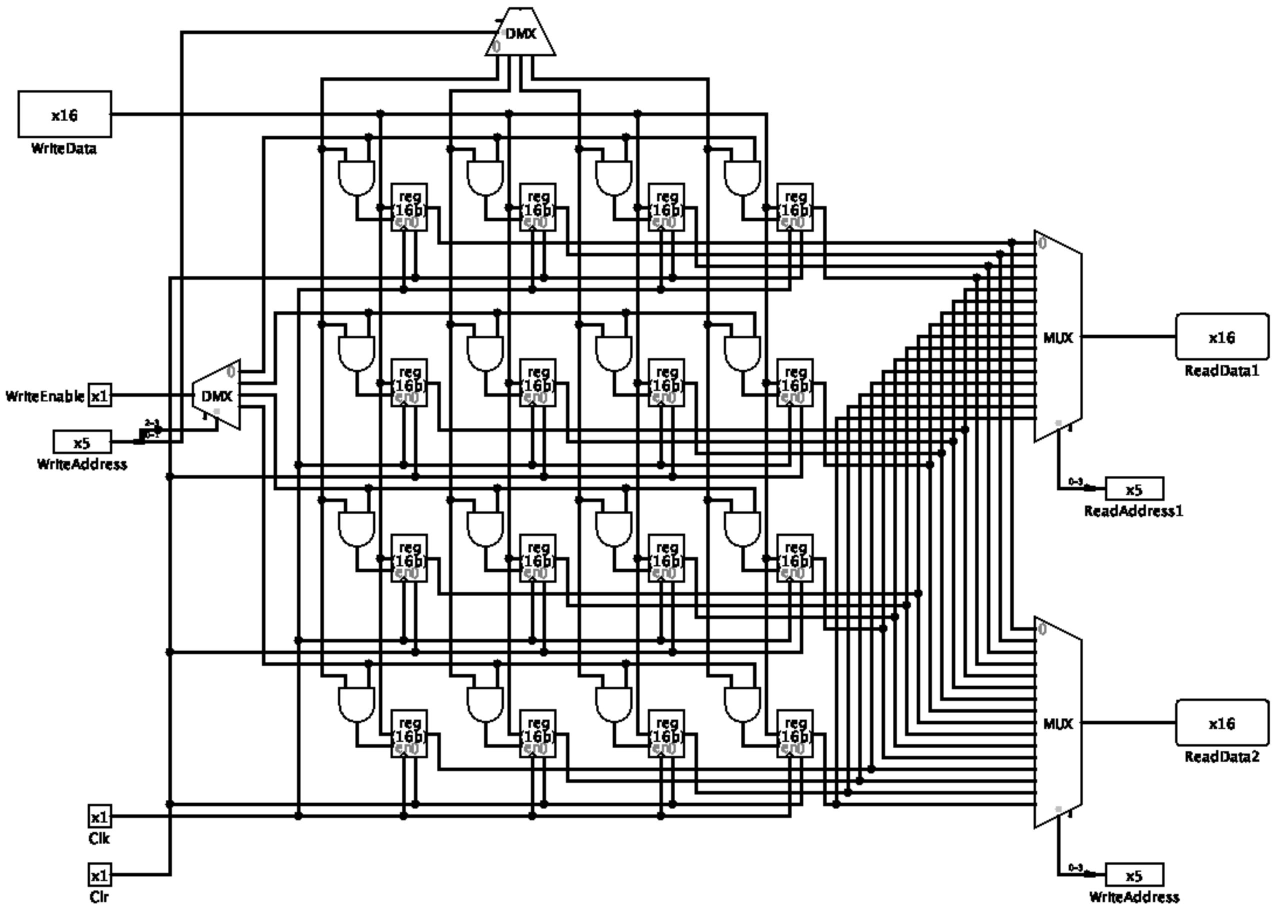


Figure 5: Register File Design

III. Control Unit

In the Control Unit, we have taken the opcode of a given instruction as an input; however, the higher two bits are redundant, since the lower four bits are enough to decide what the operation is. We have nine control signals such that RegDst, Branch, MemRead, MemtoReg, Blez, MemWrite, ALUSrc, RegWrite, and Jump. In the table below, we specify which control signal would be active for a given instruction type, and which one would not be active as 1 or 0, respectively. For example, when we have an addition operation, add, the Control Unit gives RegDst and RegWrite signals to do the instruction properly.

Ins. Type	**RegDst**	**Branch**	**Mem Read**	**Memto Reg**	**Blez**	**Mem Write**	**ALUSrc**	**Reg Write**	**Jump**
add	1	0	0	0	0	0	0	1	0
sub	1	0	0	0	0	0	0	1	0
mult	0	0	0	0	0	0	0	0	0
and	1	0	0	0	0	0	0	1	0
or	1	0	0	0	0	0	0	1	0
addi	0	0	0	0	0	0	1	1	0
sll	1	0	0	0	0	0	1	1	0
slt	1	0	0	0	0	0	0	1	0
mfhi	1	0	0	0	0	0	0	1	0
mflo	1	0	0	0	0	0	0	1	0
lw	0	0	1	1	0	0	1	1	0
sw	0	0	0	0	0	1	1	0	0
beq	0	1	0	0	0	0	0	0	0
blez	0	1	0	0	1	0	0	0	0
j	0	0	0	0	0	0	0	0	1
myIns	0	0	0	0	0	0	0	1	0

Figure 6: Table of Control Signals

At the end, the Control Unit design is as the following:

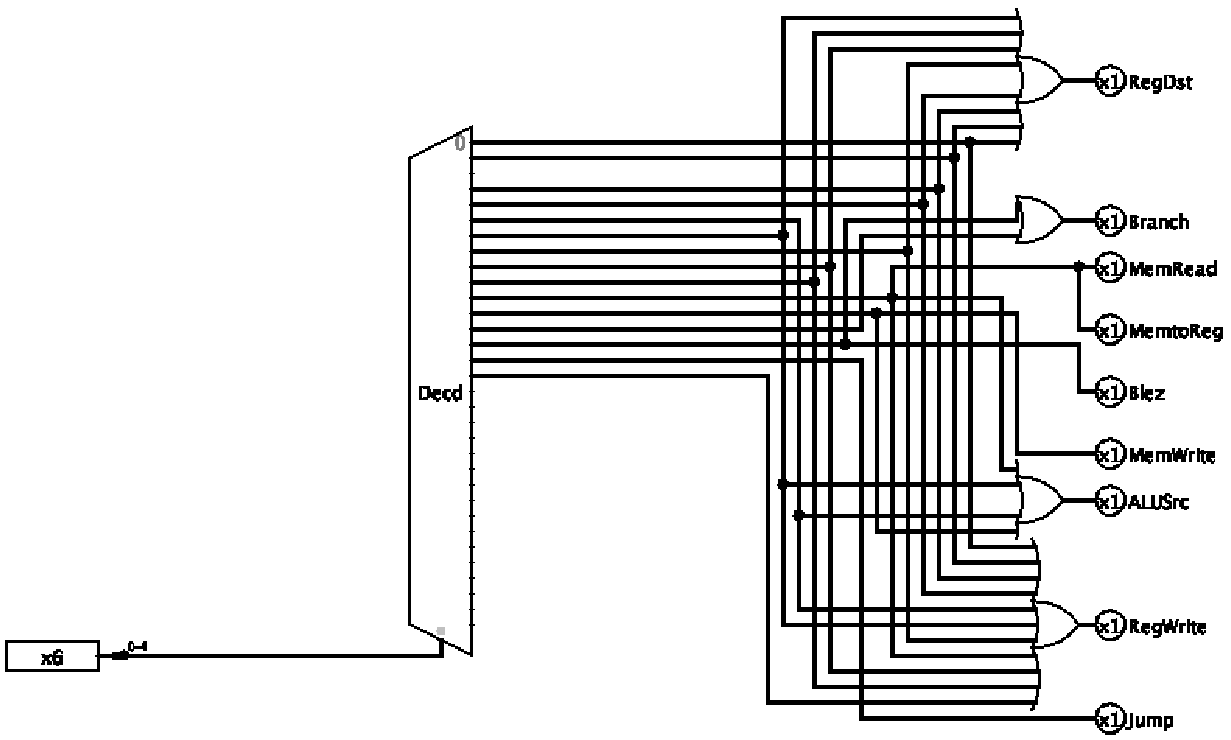


Figure 7: Control Unit Design

IV. CPU Design

In this part, we put the components that we have designed before like ALU and Register File all together to be able to implement the CPU design. For the memory part in the design, we have chosen to use a ROM for the instruction memory, but we have used a RAM for the data memory. Moreover, we give all the control signals into the correct destinations, since they are the key point for the process. The final CPU design is on the next page.

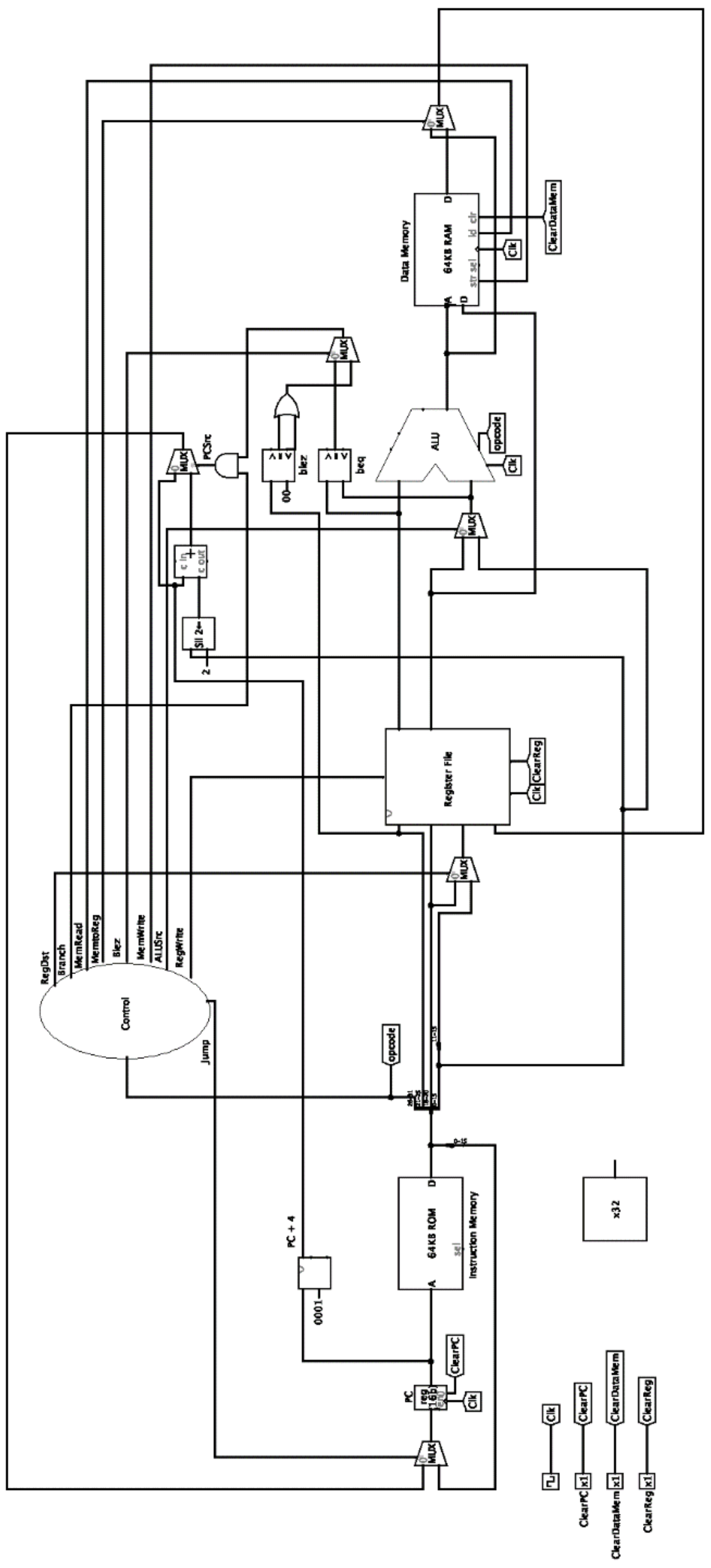


Figure 8: CPU Design

V. Testing the Processor

In this part, we have written an assembly code in a file named *test_program_asm.txt*. It contains the assembly instructions and its machine codes for each instruction, respectively. So, we can see which assembly instruction corresponds to which machine code. Here is the implementation:

```
# MAIN

# addi $0, $0, 0

0: 14000000;

# addi $2, $2, 0

4: 14420004;

# addi $5, $5, 0

8: 14A50000;

# addi $6, $6, 4

c: 14C60004;

# addi $7, $7, 8

10: 14E70008;

# beq $4, $5, 44 [exit-0x0000002c-4]

14: 30020012;

# lw $3, 0($5)

18: 28A30000;

# lw $4, 0($6)

1c: 28C40000;

# add $1, $3, $4

20: 08640800;

# sll $8, $1, 1

24: 18204040;

# sw $8, 0($7)

28: 2CE80000;

# addi $5, $5, 1

2c: 14A50001;

# addi $6, $6, 1

30: 14C60001;

# addi $7, $7, 1

34: 14E70001;

# addi $0, $0, 1

38: 14000001;

# j 14 [loop-0x0000000e-4]

3c: 38000006;

# DATA

0 : 00000005;

4 : 00000007;

8 : 0000fffe;

c : 00000028;

10: 00000041;

14: 0000ffe9;

18: 00000011;

1c: 00000400;

20: 0000008c;

24: 0000ffe0;

28: 0000001e;

2c: 00000850;
```

We have also written the corresponding machine instructions for the assembly program in the file named *test_program_code.txt.* It is used to load instructions into the instruction memory of the processor to test it.

We have also provided a separate file named *test_program.txt* that contains the data of our program.

VI. Conclusion

Finally, we have designed a 16-bit single-cycle processor and implemented it by using Logisim. We have also written the test files and checked whether it gives the expected result or not. We have first designed the ALU, and Register File, and use them to design a functional CPU. Test trace shows that our design works properly.