Birzeit University

Computer Systems Engineering

COMPUTER ARCHITECTURE

ENCS4370

Pipelined Processor

Prepared by :

Basil Mari 1191027

Islam Jihad 1191375

Yousra Sheikh Qasim 1192131

Instructor : Ayman Hroub

Date : June 10, 2022

Section : 3

1 - Abstract:

In this project, we have designed a 24-bit pipelined processor and the tool that was used is Logisim simulator.

The RISC processor has 3 different types that support 24 bits instructions (R-type, I-type and J-type) with different addressing modes, also eight 24-bit general-purpose registers: R0 through R7 and 24-bits program counter register were implemented.

The five stages of the pipelined processor were implemented and tested successfully.

Table of Contents

Table of figures

Table of tables

Theory:

The data path is a basic component if we want to build any computer.

Datapath:

Is a set of functional units that carry out data processing operations. Data paths, with a control unit, make up the CPU (central processing unit) of a computer system. A larger data path can also be created by joining more than one together using multiplexers.

Pipelining Architecture:

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages relate to one another to form a pipe like structure. Instructions enter from one end and exit from another end. Pipelining increases the overall instruction throughput.

Project background:

The aim of this project is to implement and design a 24-bit pipelined processor. The five stages were constructed like the pipeline presented in the class lectures.

Pipeline registers between stages were added. The control logic to detect data dependencies among instructions was designed and the forwarding logic was implemented.

For branch and jump instructions, the delay to one cycle only was reduced. The pipeline for one clock cycle after a jump or a taken branch instruction was stalled.

The data path was designed to support the 3 different types of the instructions (R-type, I-type and J-type).

No.	Instr	Meaning	Encoding			
R-Type Instructions						
0	AND	Reg(Rd) = Reg(Rs) & Reg(Rt)	Op = 00000	Rs	Rt	Rd
1	CAS	Reg(Rd) = Max[Reg(Rs) , Reg(Rt)]	Op = 00001	Rs	Rt	Rd
2	Lws	Reg(Rd) = Mem[Reg(Rs) + Reg(Rt)]	Op = 00010	Rs	Rt	Rd
3	ADD	Reg(Rd) = Reg(Rs) + Reg(Rt)	Op = 00011	Rs	Rt	Rd
4	SUB	Reg(Rd) = Reg(Rs) – Reg(Rt)	Op = 00100	Rs	Rt	Rd
5	CMP	zero-flag = Reg(Rs) < Reg(Rt)	Op = 00101	Rs	Rt	0000
6	JR	PC = Reg(Rs)	Op = 00110	Rs	0000	0000

Table 1 Instruction Encoding R-Type

I-Type Instructions						
7	ANDI	Reg(Rt) = Reg(Rs) & Immediate^{10}	Op = 00111	Rs	Rt	Immediate^{10}
8	ADDI	Reg(Rt) = Reg(Rs) + Immediate^{10}	Op = 01000	Rs	Rt	Immediate^{10}
9	Lw	Reg(Rt) = Mem(Reg(Rs) + Imm^{10})	Op = 01001	Rs	Rt	Immediate^{10}
10	Sw	Mem(Reg(Rs) + Imm^{10}) = Reg(Rt)	Op = 01010	Rs	Rt	Immediate^{10}
11	BEQ	Branch if (Reg(Rs) == Reg(Rt))	Op = 01011	Rs	Rt	Immediate^{10}
J-Type Instructions						
12	J	PC = PC + Immediate^{17}	Op = 01100	Immediate^{17}		
13	JAL	R7 = PC + 1, PC = PC + Immediate^{17}	Op = 01101	Immediate^{17}		
14	LUI	R1 = Immediate^{17} << 4	Op = 01110	Immediate^{17}		

Table 2 Instruction Encoding I-type and J-Tpe

Design and Implementation

Components

Register file

Register File consists of 8 × 24-bit registers

BusA and BusB are 24-bit output buses for reading and writing two registers, respectively. BusW is a 24-bit input bus for writing a register when RegWrite is 1.

In a cycle, two registers are read, and one is written.

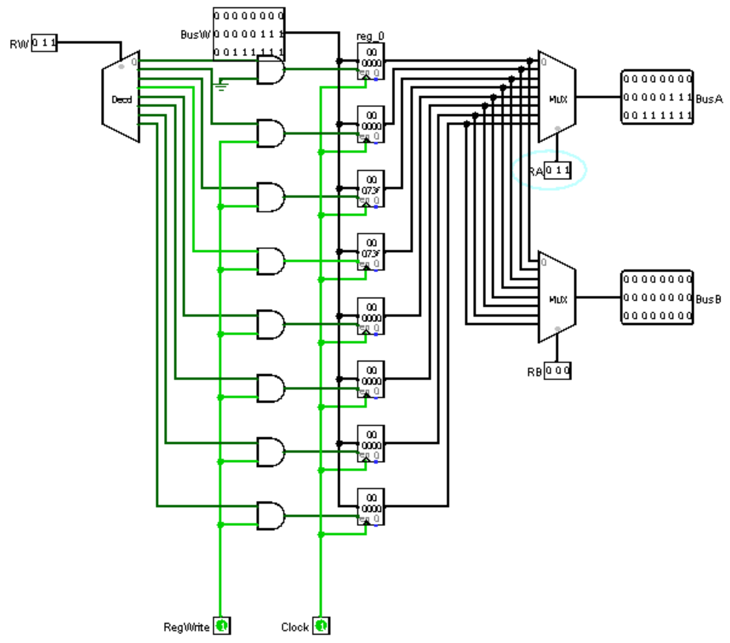


Figure 1 Register file

Instruction Memory

Because Datapath does not write instructions and instead acts as combinational logic for reads, instruction memory just needs to allow read access. After the access time, the address chooses the instruction.

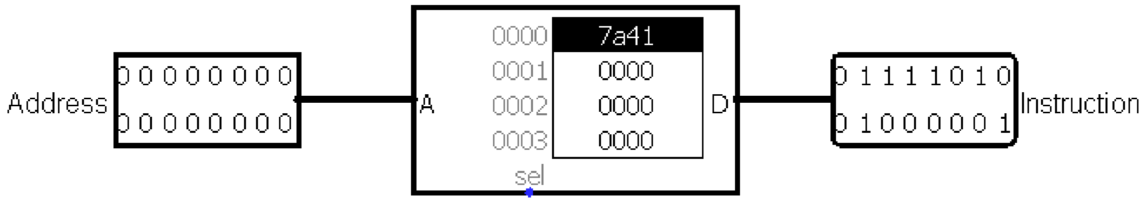


Figure 2 Instruction Memory

ALU

An arithmetic logic unit (ALU) is a digital combinational circuit that performs arithmetic and bitwise operations on integer binary integers in computing. A floating-point unit, on the other hand, works with floating-point numbers. In our CPU it will get 2 inputs of 24 bits and one output of 24 bits with ALU control signal to choose which operation to perform and some flags like ZERO flag.

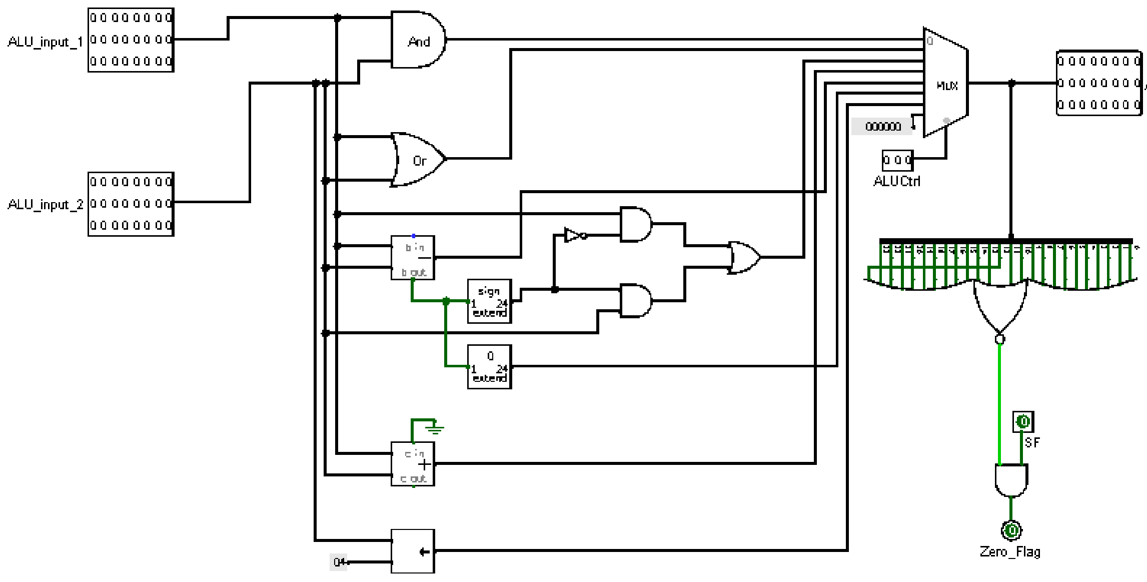
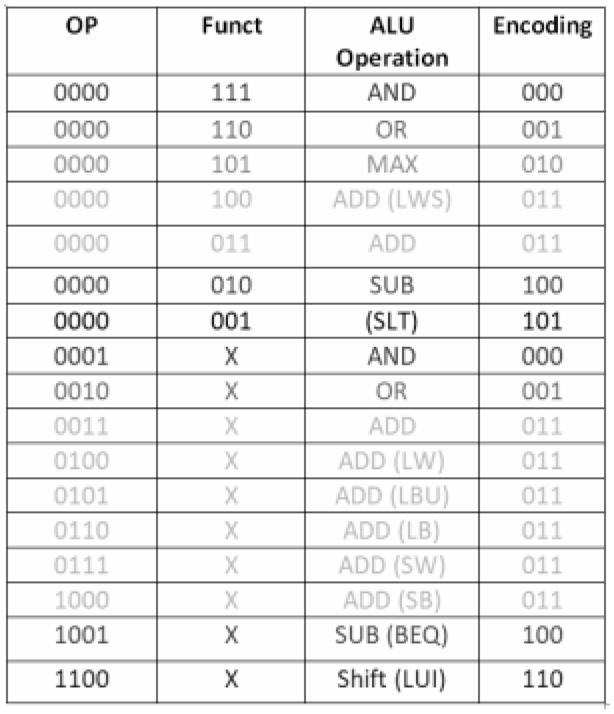


Figure 3 ALU



OP	Funct	ALU Operation	Encoding
0000	111	AND	000
0000	110	OR	001
0000	101	MAX	010
0000	100	ADD (LWS)	011
0000	011	ADD	011
0000	010	SUB	100
0000	001	(SLT)	101
0001	X	AND	000
0010	X	OR	001
0011	X	ADD	011
0100	X	ADD (LW)	011
0101	X	ADD (LBU)	011
0110	X	ADD (LB)	011
0111	X	ADD (SW)	011
1000	X	ADD (SB)	011
1001	X	SUB (BEQ)	100
1100	X	Shift (LUI)	110

Table 3 ALU truth table

Following are the logic equations for the ALU operations :

1. AND = OP(0000).Funct(111) + OP(0001).

2. OR = OP(0000).Funct(110) + Op(0010).

3. MAX = OP(0000).Funct(101)

4. ADD = OP(0000).Funct(100) + OP(0000).Funct(011) + OP(0011) + OP(0100) + OP(0110) + OP(0111) + OP(1000).

5. SUB = OP(0000).Funct(010) + OP(1001)

6. SLT = OP(0000).Funct(001)

7. Shift = OP(1100)

Next PC

It’ s a circuit to determine and calculate the next program counter for the processer, and making aware of some conditions like Branch if not equal or jump conditions.

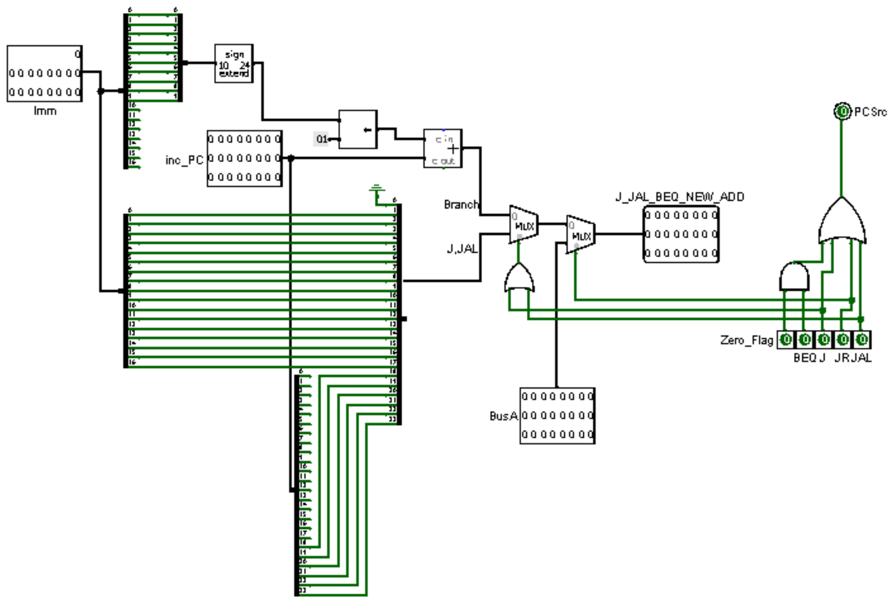


Figure 4 Next PC

Data Memory

Data memory where data is stored. It’ s used for load and store

MemRead: enables output on Data_out

Address selects the word to put on Data_out

MemWrite: enables writing of Data_in

Address selects the memory word to be written

The Clock synchronizes the write operation

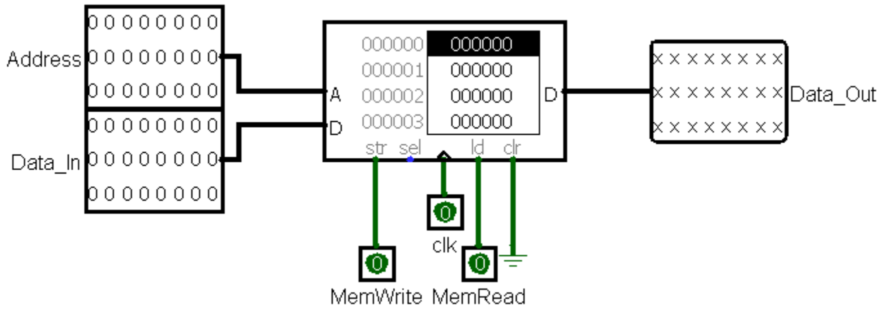


Figure 5 Data Memory

INC PC

By employing a D-Flip-Flop that transfers the value at its input to its output at the rising edge of the clock, this component is responsible for incrementing the PC by 2 at each clock cycle. A wire from its output is labeled (PCToInstMem) as an output of this component, and another wire from its output is linked to an adder that adds 2 to the input, yielding IncPC, which contains the address of the next instruction to be fetched. The INC PC circuit is shown below, which has been implemented in Lgisim.

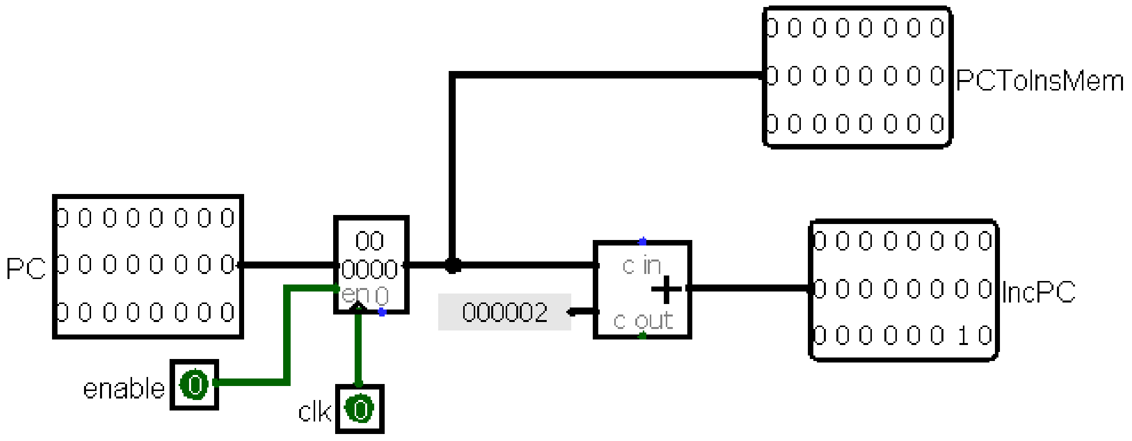


Figure 6 INC PC

IF/ID Buffer

We require a buffer between each two stages for pipelining purposes, thus we have four buffers: IF/ID, ID/EX, EX/MEM, and EM/WB. All of them are implemented in the same way, with registers for each value to be put in the buffer, and they're all triggered on the rising edge of the clock. The signals at each buffer are illustrated below:

IF/ID: There are seven signals: funct, Rd, Rt, Rs, all of which are 3-bits, OP, which is 4-bits, imm_10, which is 10-bits, and im_17, which is 17-bits, all of which were obtained by utilizing splitters from the Fetched instruction. The final signal is the IncPC, which is a 24-bit signal that is acquired from the INCPC resulting by the INC PC component.

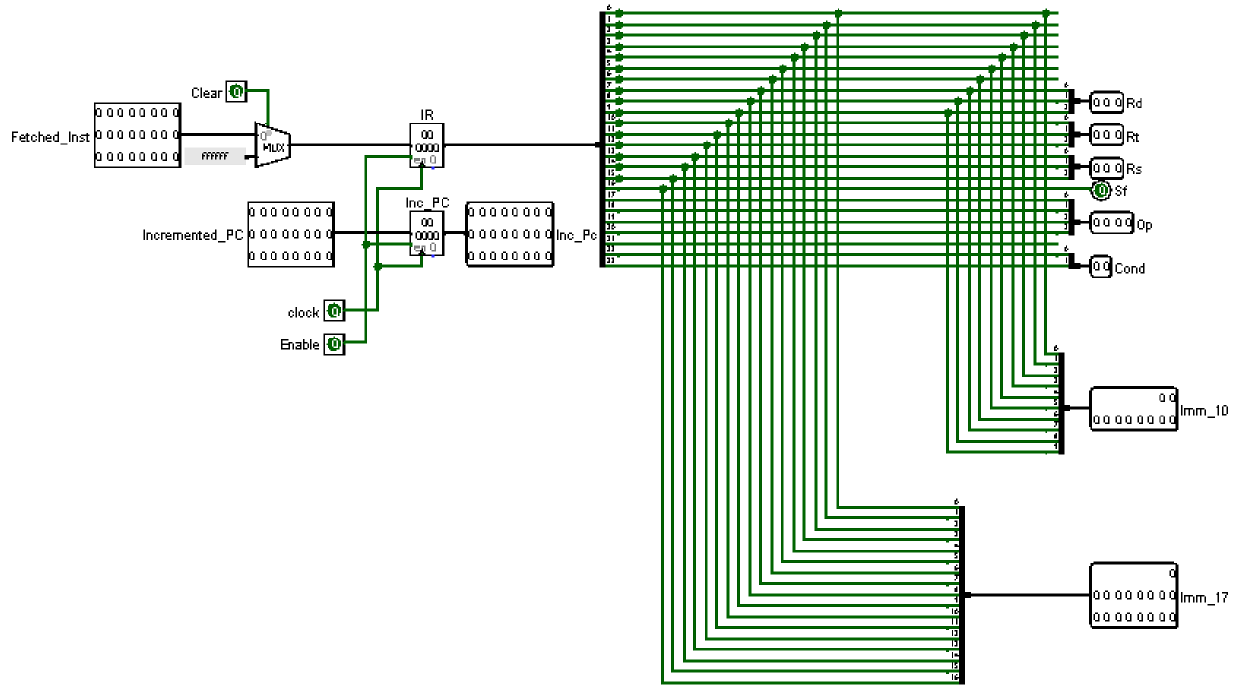


Figure 7 IF/ID Buffer

Extender 17

This unit take 17 bits as input and extend them to 24 bits, and they are signed

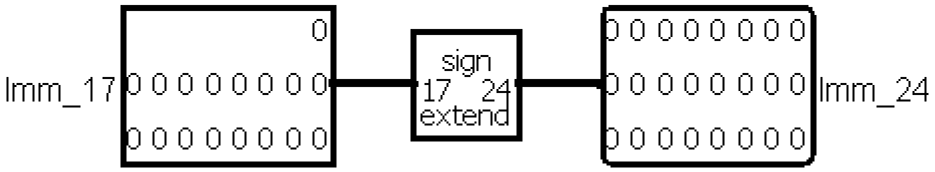


Figure 8 Extender 17

Extender 10

This unit take 10 bits as input and extend them to 24 bits, whether they are signed or unsigned. And the mux is to choose between signed or not.

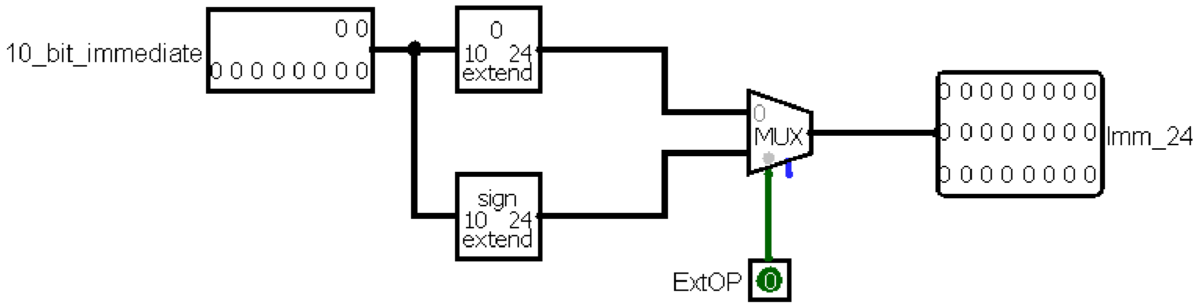


Figure 9 Extender 10

ID/EX Buffer

ID/EX: We have 20 signals, including BUSA and BUSB of 16 bits, which represent the values of Reg(RS) and Reg(Rt) obtained from either the register file or the forwarding units, as well as imm IType and imm JType of 16 bits, which are obtained from the Extender 6 and Extender 12, respectively, as well as Rt, Rd, Rs, OP, funct, and IncPC signals obtained from the first buffer

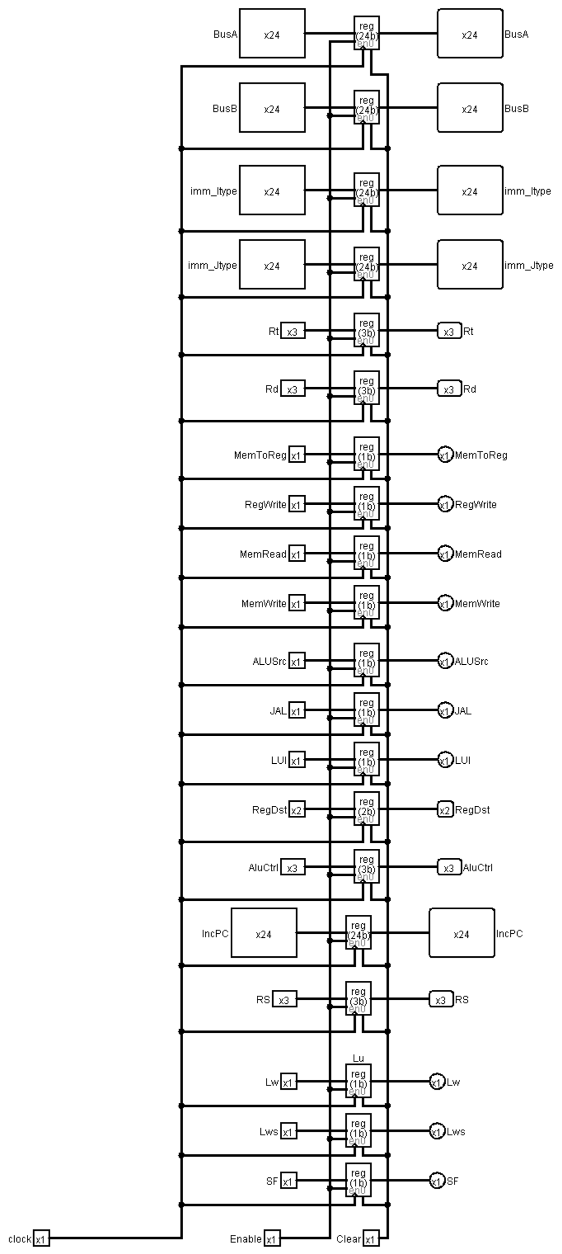


Figure 10 ID/EX Buffer

EX/MEM

EX/MEM: The following 11 signals are obtained from the second buffer: IncPC, BUSB, MemToReg, RegWrite, MemRead, MemWrite, Rt, Lb, LBu, as well as the ALUResult signal, which contains the address of the destination to write the result on, and the RW signal, which contains the address of the destination to write the result on.

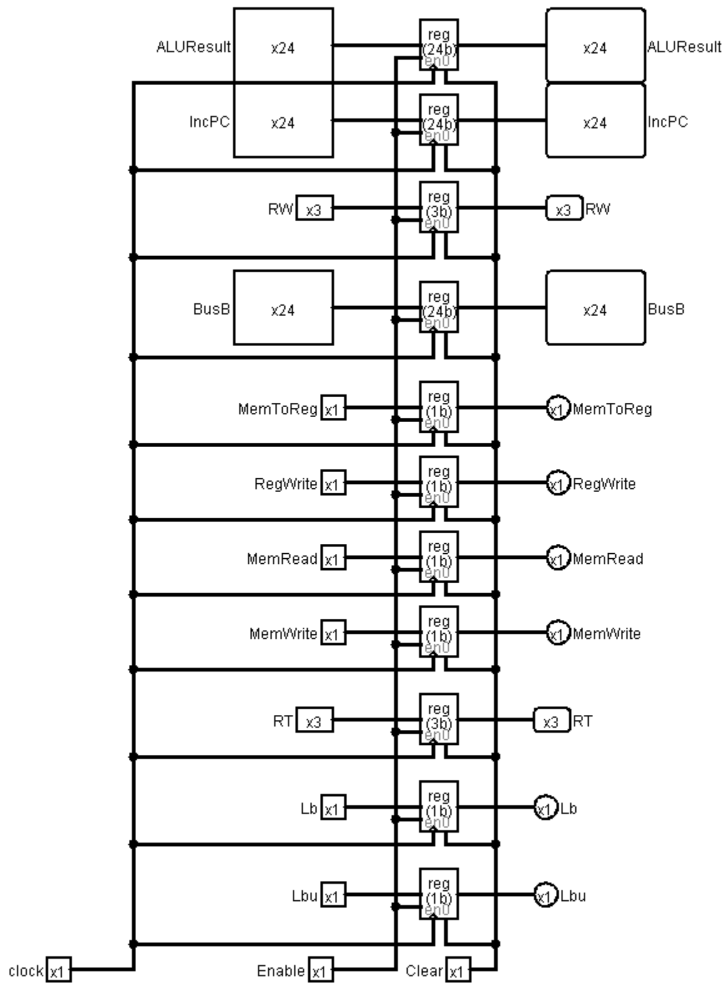


Figure 11 EX/MEM

MEM/WB

MEM/WB: The ALUResult, IncPC, MemToreg, RegWrite, and RW signals from the previous buffer, as well as the MemoryData signal from the Memory stage, are all retrieved from the previous buffer.

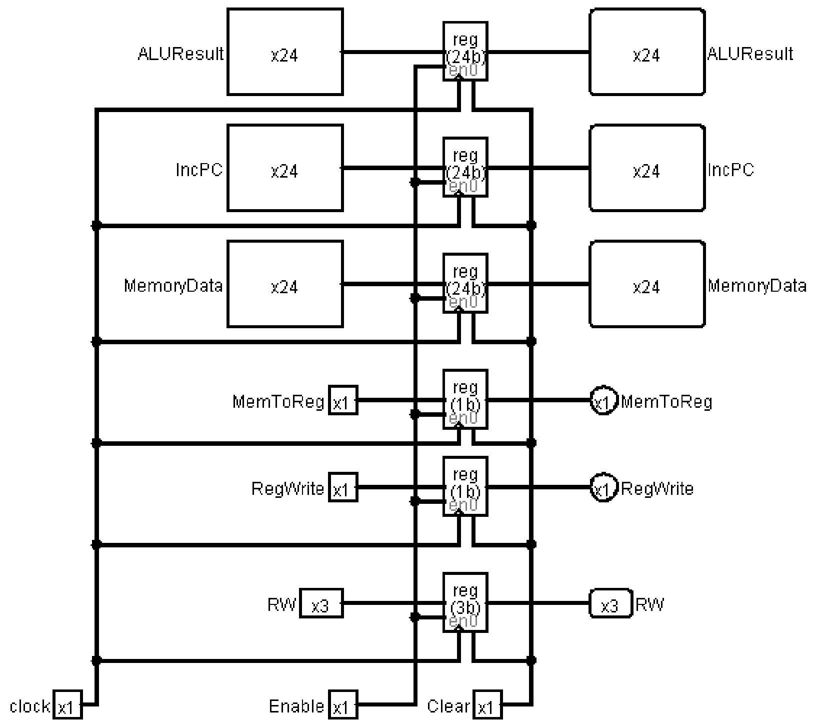


Figure 12 MEM/WB

Forwarding Units

All forwarded units that have been implemented should be indicated in this section. When a stall is required, a forwarding unit is used to optimize the pipelined route, since certain processes need results from previously unfinished actions. The following forwarding unit is used in our datapath:

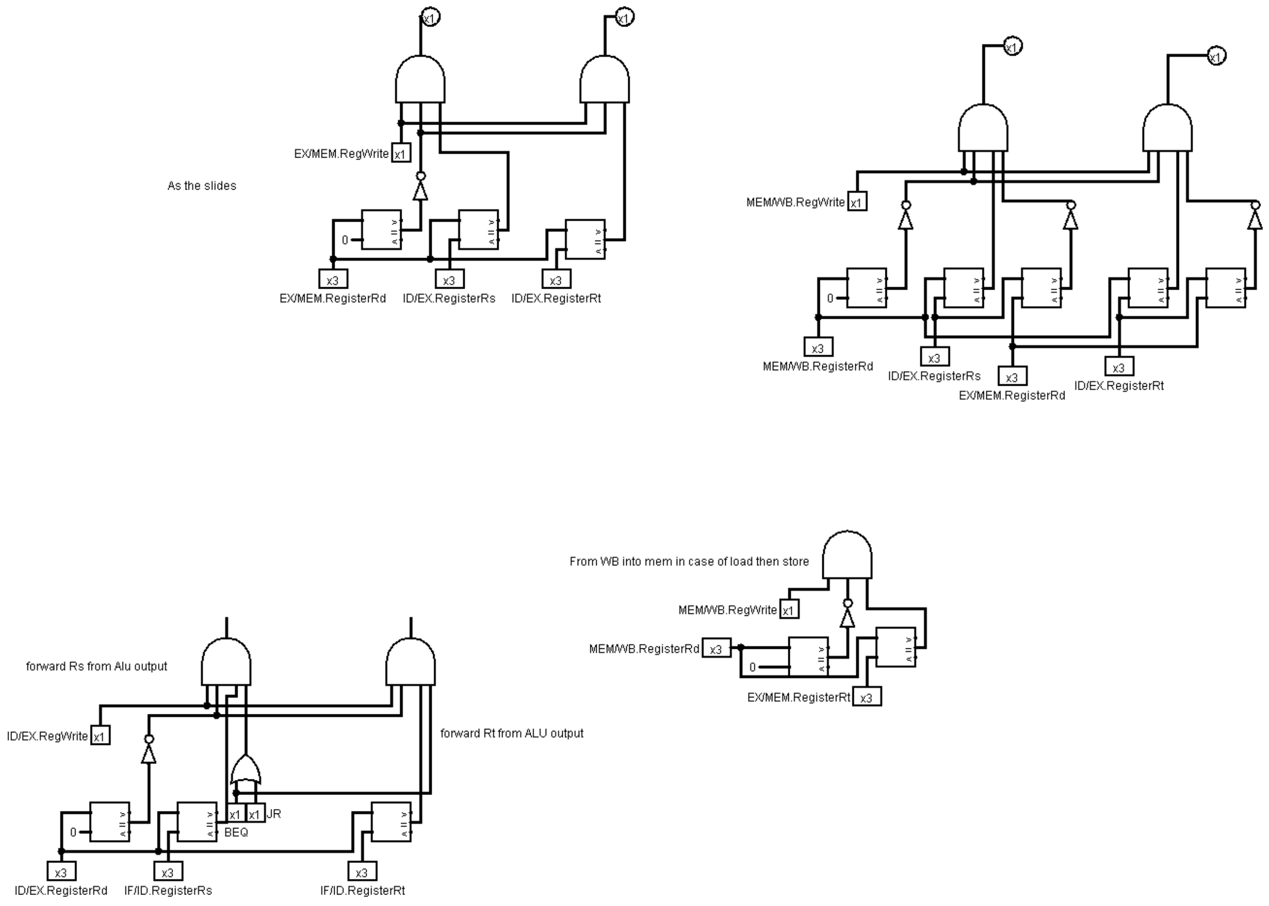


Figure 13 Forwarding Unit

Forwarding MEM to EX

Forwarding WB to EX

Forwarding WB to MEM

Forwarding EX to ID

Forwarding MEM to ID

The first sends the loaded value from memory in the memory stage to the decode stage when a read after write dependence between the instruction in the decode stage and the one in the memory stage is identified. This forwarding unit is mostly concerned with a load instruction, then any instruction, and finally an instruction that reads the loaded value. The following reasoning is used to determine the presence of a hazard:

```
If ( EX/MEM.RegWrite

and (EX/MEM.MEMRead)

and (EX/MEM.RegisterRd != 0)

and (ID/EX.RegisterRd != IF/ID.RegisterRs)

and (EX/Mem.RegisterRd = IF/ID.RegisterRs ))


then MEM_to_ID_BusA = 1

If ( EX/MEM.RegWrite

and (EX/MEM.MEMRead)

and (EX/MEM.RegisterRd != 0)

and (ID/EX.RegisterRd != IF/ID.RegisterRt)

and (EX/MEM.RegisterRd = IF/ID.RegisterRt))

then MEM_to_ID_BusB = 1
```

Following is the implemented circuit for the unit.

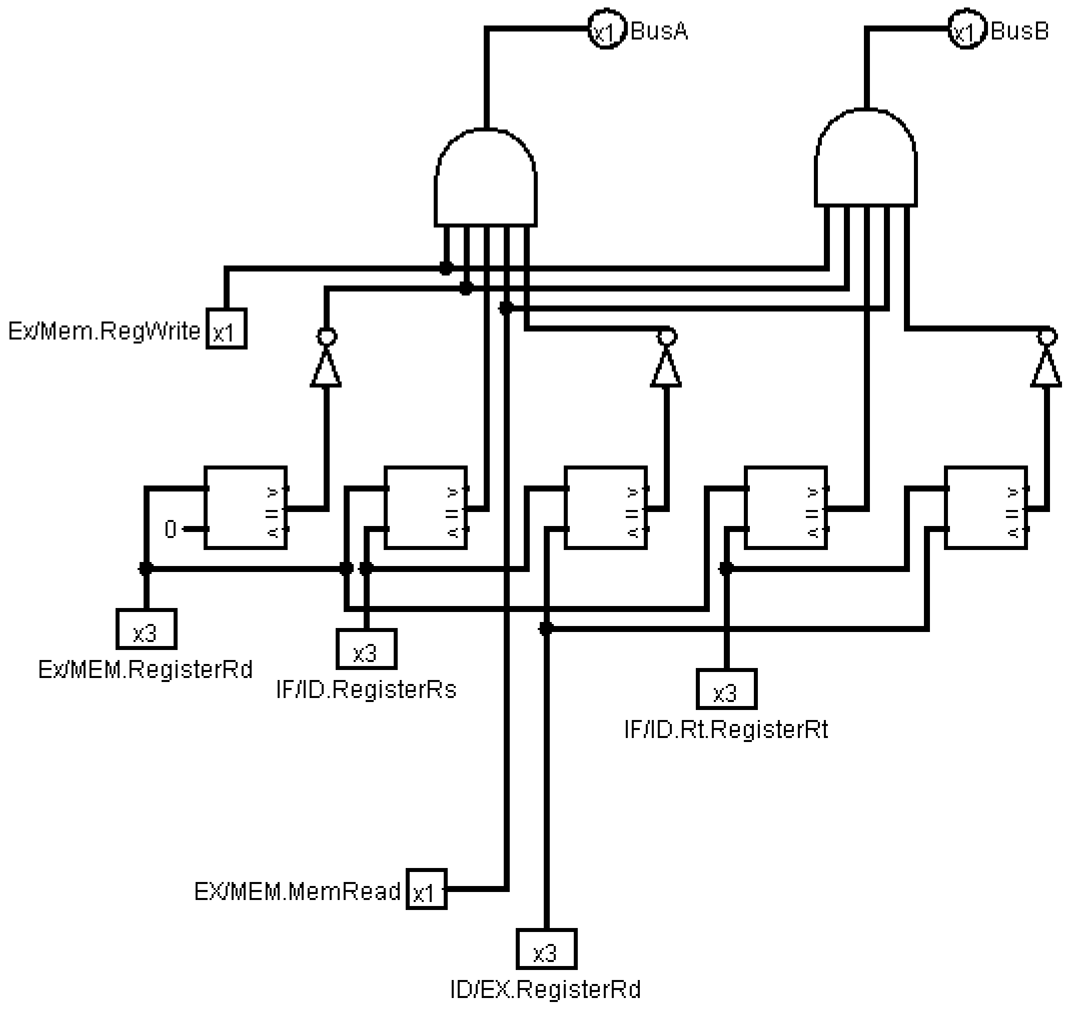


Figure 18 Forwarding MEM to ID

Hazard Detection

Hazard detection unit

This unit was created to determine when it was necessary to stall the pipeline for one cycle, i.e. when the forwarding fails to transmit the data at the appropriate moment, and by examining all of the instances, the forwarding fails and a stall cycle is required in the two circumstances below.

1. When a load instruction is followed by an instruction that is dependent on the data returned by the load instruction (R/W dependency), for example:

a. Load followed by an R-type or a Branch instruction that is dependent on the data returned by the load instruction (RS or Rt in the R-type or the Branch instruction need Rd from the load instruction).

b. Load followed by a store or JR instruction that uses the loaded data to determine the address of the required memory location in the case of a Store instruction, or to update the value of the PC in the case of a JR instruction (dependency between Rs in the store or JR instruction and Rt in the load instruction).

2. When the PC has to be updated due to J, JAL, JR, and BEQ commands.

How to detect the stall cycles:

1. The following logic is used to identify the stall cycle when a load instruction is followed by an instruction that relies on it (as shown above):
   ```
   If (ID/EX.MEMRead  and (ID/EX.RegisterRD != 0)
   and (ID/EX.RegisterRd = IF/ID.RegisterRs
   or (not IF/ID.MemWrite and ID/EX.RegisterRd = IF/ID.RegisterRt)))
   ```

2. In the event of a missprediction in BEQ, J, JAL, or JR, the following logic is used to identify the stall cycle:
   ```
   if( PCSrc and
   PC1 != PC2)
   ```
   where PC1 is the one generated from the Next_PC Component and the second is the IncPC.

What we need to do to stall the pipelining:

1. For the first type of stall (R/W dependency due to the Load instruction), we must disable the INC PC component and the IF/ID buffer using the signal generated by the detection logic shown above, as well as clear (flush) the second buffer for the first half cycle of the next cycle (after detecting the stall) by using a D-FlipFlop triggering at the falling edge and connecting its output to a and gate with the clock signal, so the resulted signal will be zero

2. For the second type of stall, we need to flush the first buffer (IF/ID buffer) when we detect the stall to prevent it from passing to the decode stage and processing the wrong instruction, and the way we control the stall in this case is by using a D-FlipFlop triggering at the falling edge of the clock to pass the signal generated from the detected logic of the second type, and anding its output with the not of the clock, so the flush for the first buffer is done in this way.

Note that flushing (clearing) means passing a nop instruction into the decoding stage (which includes setting all the bits in the instruction to be passed into the decoding stage) because clearing the buffer by resetting it results in a JR instruction and returns the program execution to the first instruction.

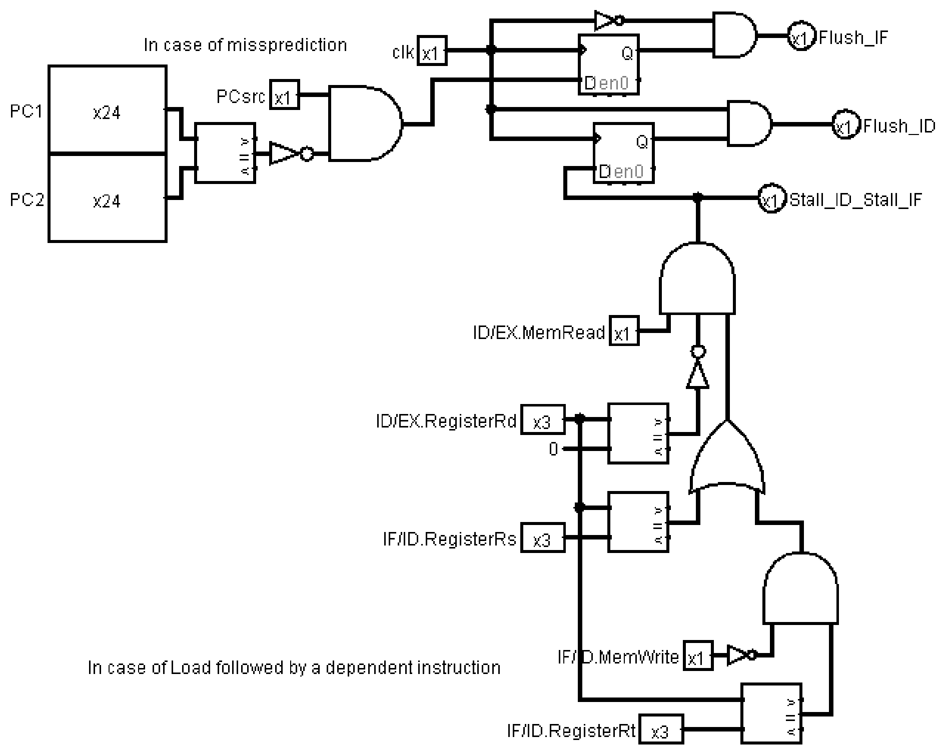


Figure 19 Hazard Detection

Control Unit

The Main Control Unit is a core unit in the implemented datapath, and it is in charge of generating various signals that ensure proper coordination of all employed multiplexers, inputs to various units, and some of the work performed by various units. It takes the instruction's opcode and function fields as input and then goes about doing its job. The generated signals are listed in the truth table for the Control unit.

Following is the implemented Control unit after having the truth table and the expression for each signal ready.

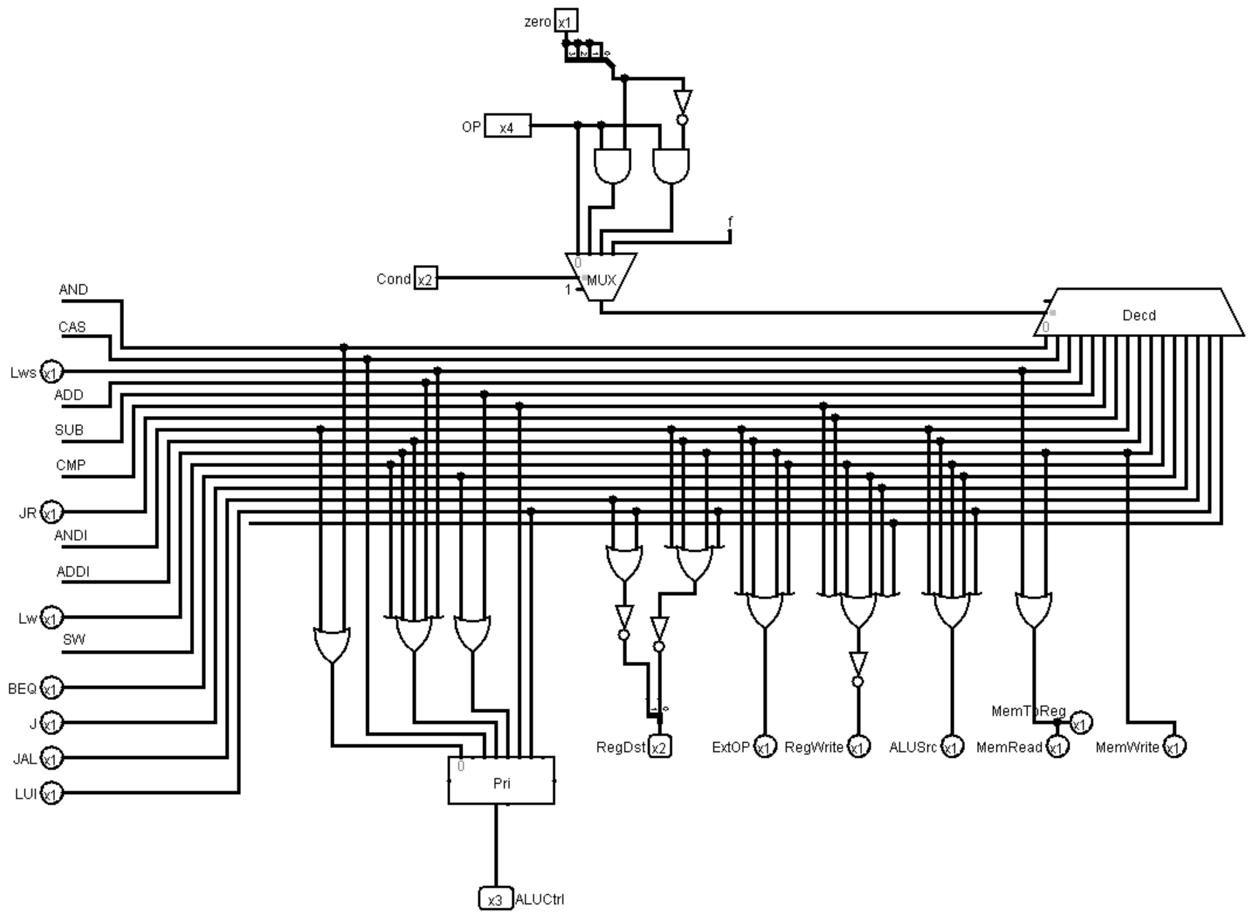


Figure 20 Control Unit

Control unit truth table:

OP		ALUctrl	Regdest	ExtOp	RegWrite	ALUsrc	Beq	J	JAL	JR	LUI	MR	MW
AND	0000	000	11	X	1	0	0	0	0	0	0	0	0
COS	0001	010	11	X	1	0	0	0	0	0	0	0	0
LWS	0010	011	11	X	1	0	0	0	0	0	0	1	0
ADD	0011	011	11	X	1	0	0	0	0	0	0	0	0
SUB	0100	100	11	X	1	0	0	0	0	0	0	0	0
CMP	0101	101	X	X	0	0	0	0	0	0	0	0	0
JR	0110	X	X	X	0	X	0	0	0	1	0	0	0
ANDI	0111	000	10	X	1	1	0	0	0	0	0	0	0
ADDI	1000	011	10	1	1	1	0	0	0	0	0	0	0
LW	1001	011	10	1	1	0	0	0	0	0	0	1	0
SW	1010	011	X	1	0	1	0	0	0	0	0	0	1
BEQ	1011	100	X	X	0	1	1	0	0	0	0	0	0
J	1100	X	X	X	0	X	0	1	0	0	0	0	0
JAL	1101	X	01	X	1	X	0	0	1	0	0	0	0
LUI	1110	110	00	X	1	1	0	0	0	0	0	0	0

Table 4 Control unit truth table

Expiration signal:

Regdest [0] = ¬(Jal x LUI)

[0] = (a`b`)+ bc`

And we calculated it for all other labels

BEQ = BEQ

J = J

JAL = JAL

LUI = LUI

LBU = LBU

LB = LB

Lws or Lw

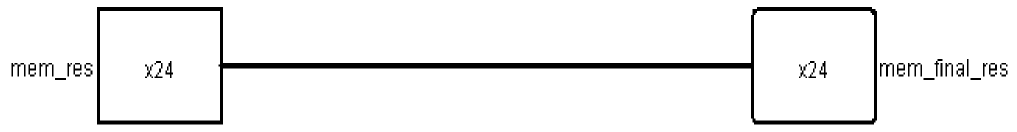


Figure 21 Lws or Lw

Pipeline Circuit

Pipelining is a method of executing many instructions at the same time. The pipeline is split into stages, each of which is joined to the next to create a pipe-like structure. Instructions come in from one end and leave from the other. Pipelining improves the total flow of instructions.

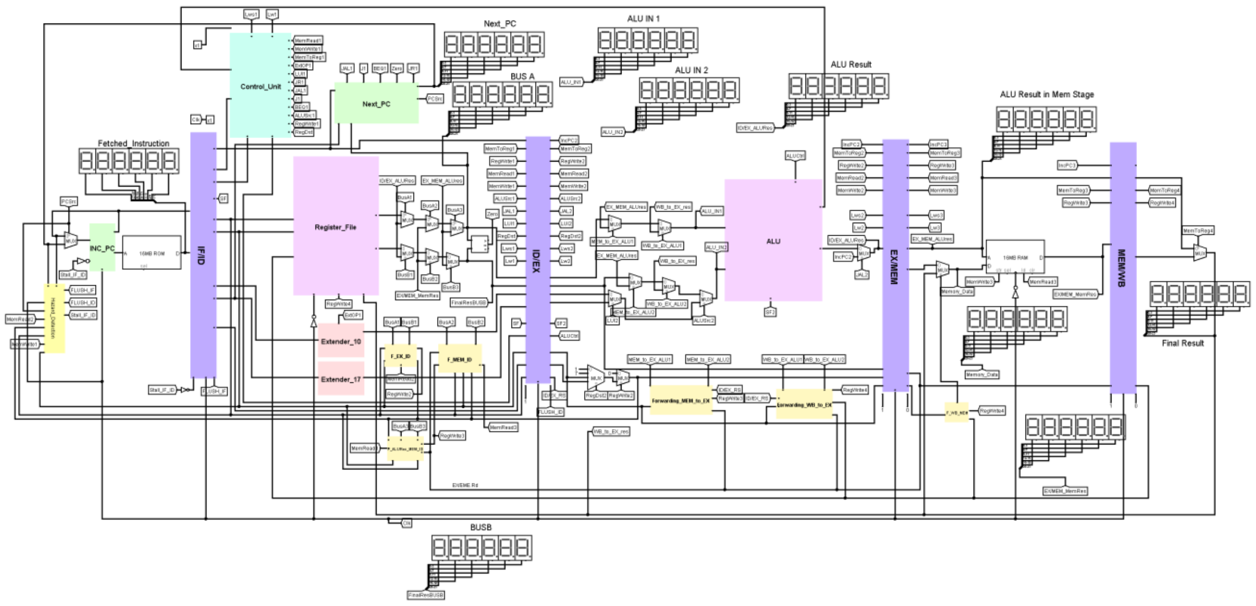


Figure 22 Pipeline Circuit

Simulation and Testing

We will test a case that contains many functions which are: addi, sub, beq, lw, J and the code is:

```
lw r1, 5

lw r2,1

loop:

      beq r2, 5, exit

      sub r1,r1,1

      addi r2,r2,1

      j loop

exit
```

And after translate it to Logisim we got this:

v2.0 raw

100405

100801

162002

082880

19fffd

So to follow the clocks of the machine we' ll look in every cycle.

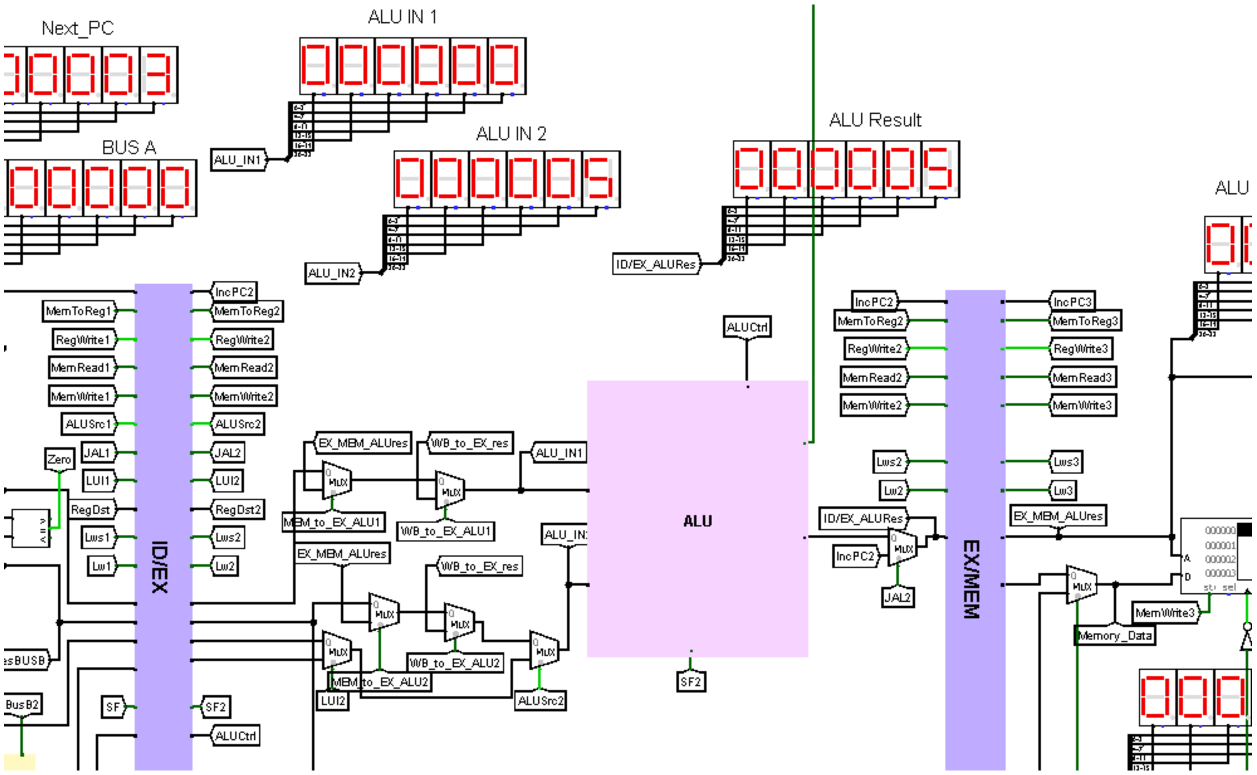


Figure 23 Testing 1

lw r1, 5

At first clock the first line has been fetched to R1 by integer 5 as seen.

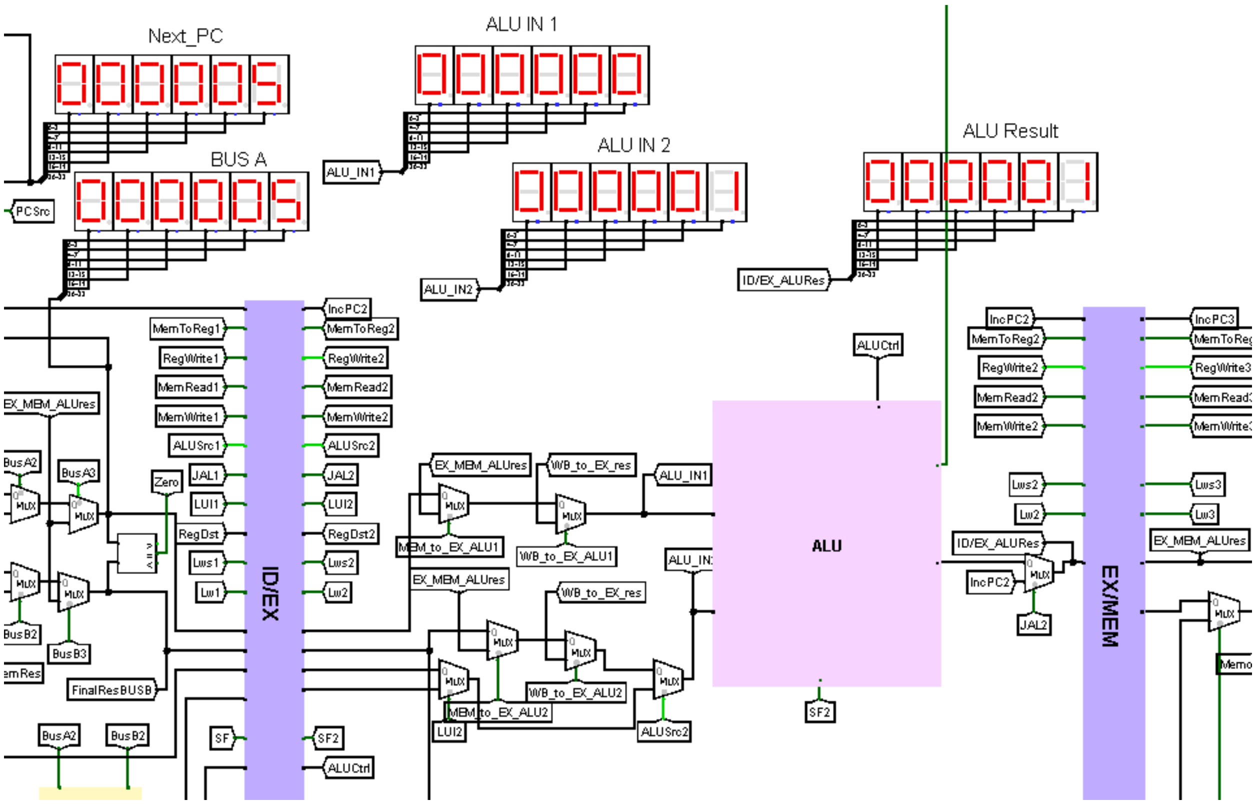


Figure 24 Testing 2

lw r2,1

At second clock the second line has been fetched to R2 by integer 1 as seen.

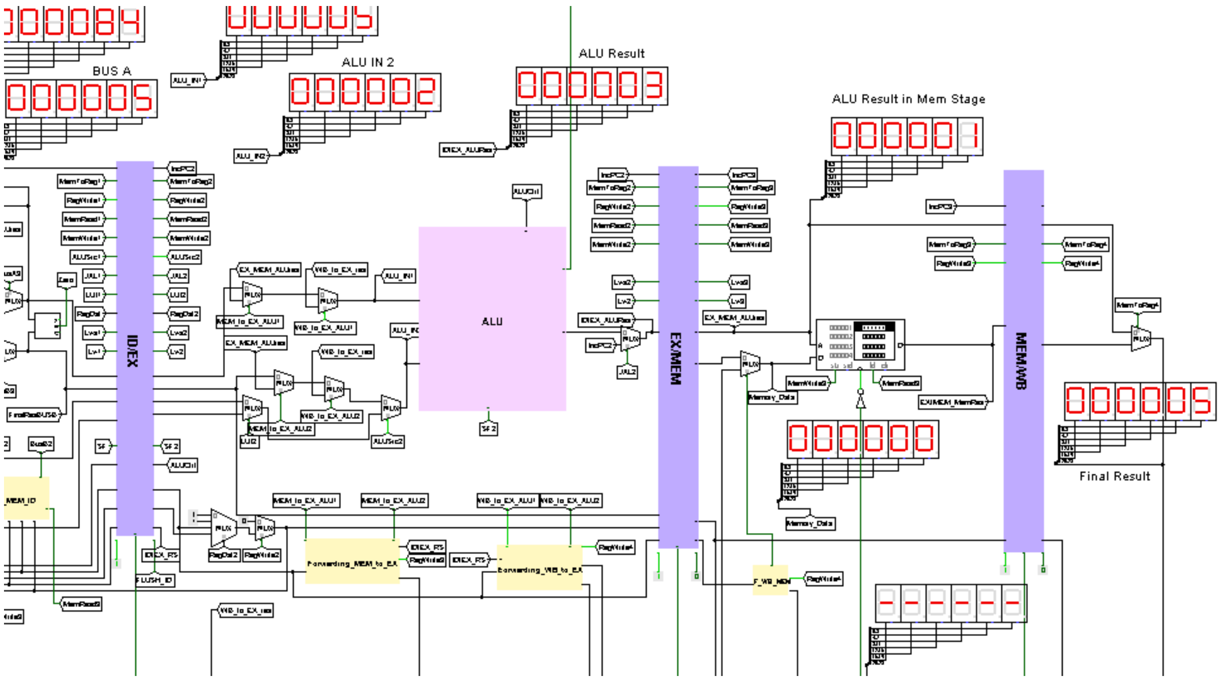


Figure 25 Testing 3

beq r2, 5, exit

In 3rd clock, branch instruction has been fetched and the previous 2 instruction are in execution and decode levels

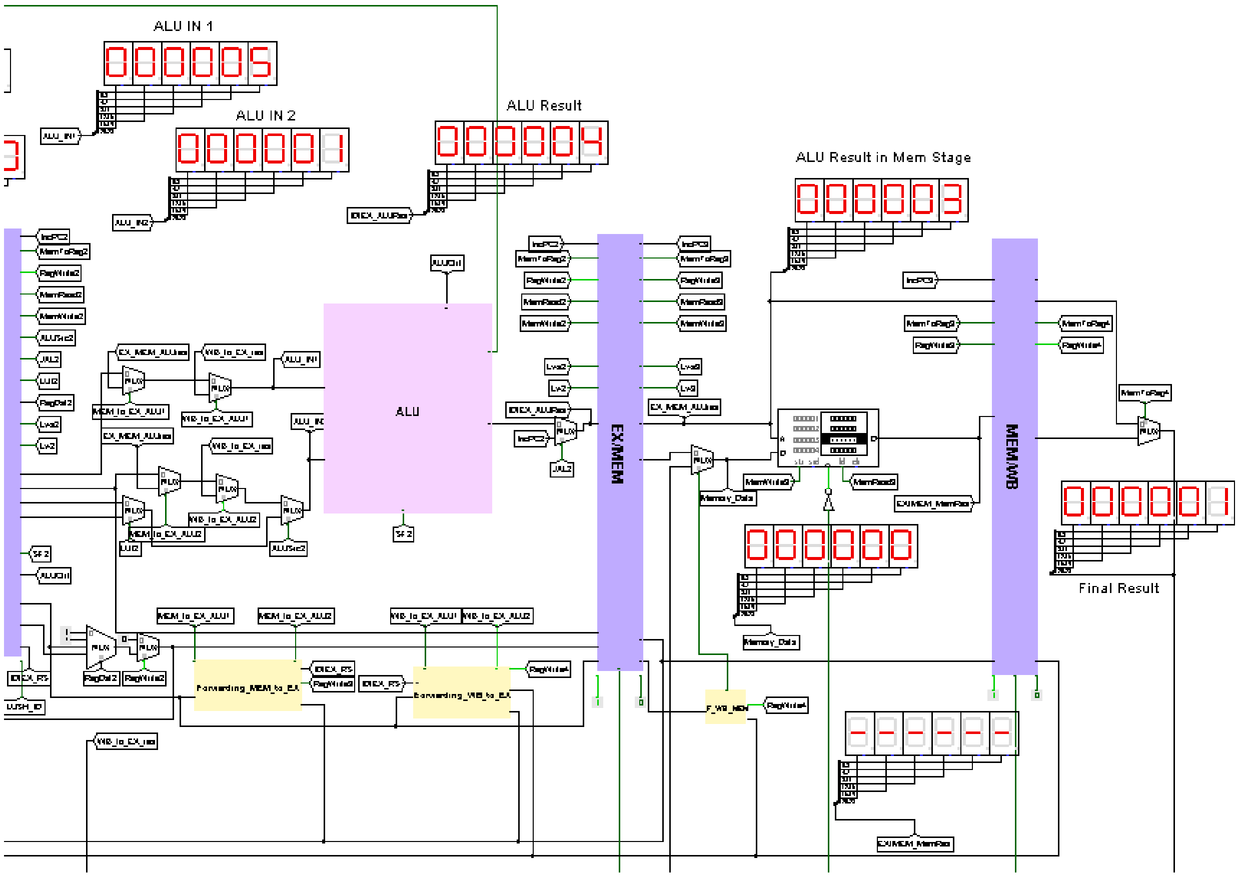


Figure 26 Testing 4

In the 4th clock the **sub r1,r1,1** instruction is being as seen 5 - 1 = 4 in ALU result when ALU first input = 5 and second input =1 and the operation is subtract.

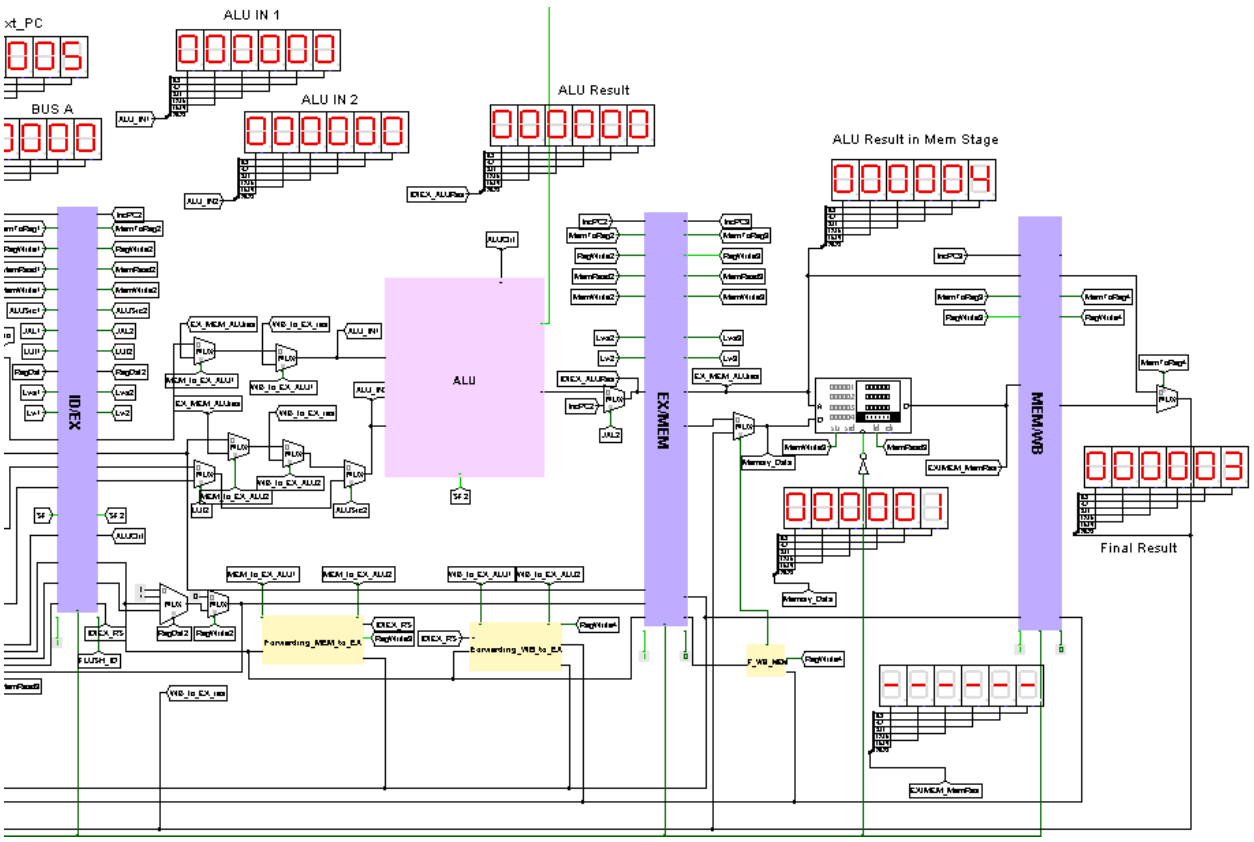


Figure 27 Testing 5

In 5th cycle the answer were sent to memory after it finished the ALU operation as seen

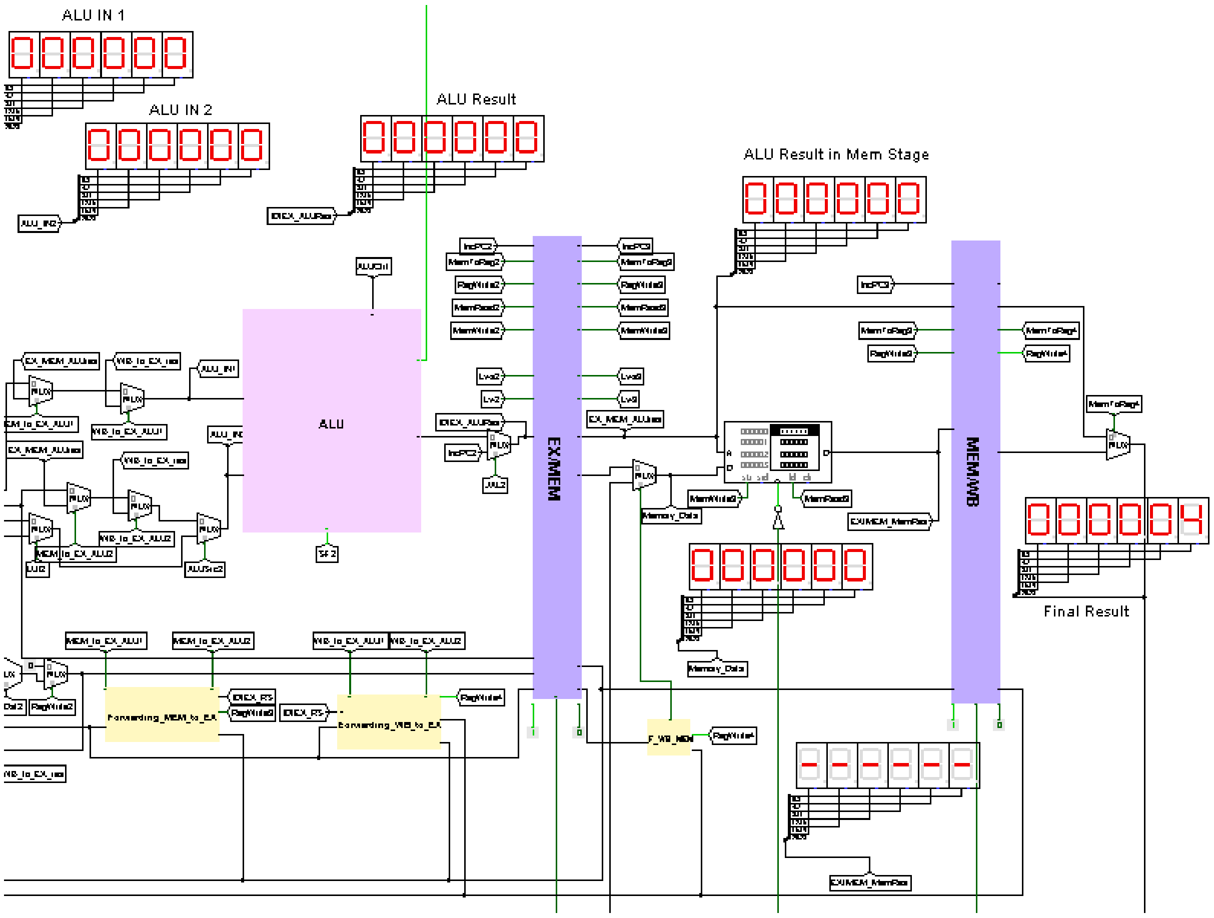


Figure 28 Testing 6

In 6th clock cycle the result has been transferred from memory to memory write back

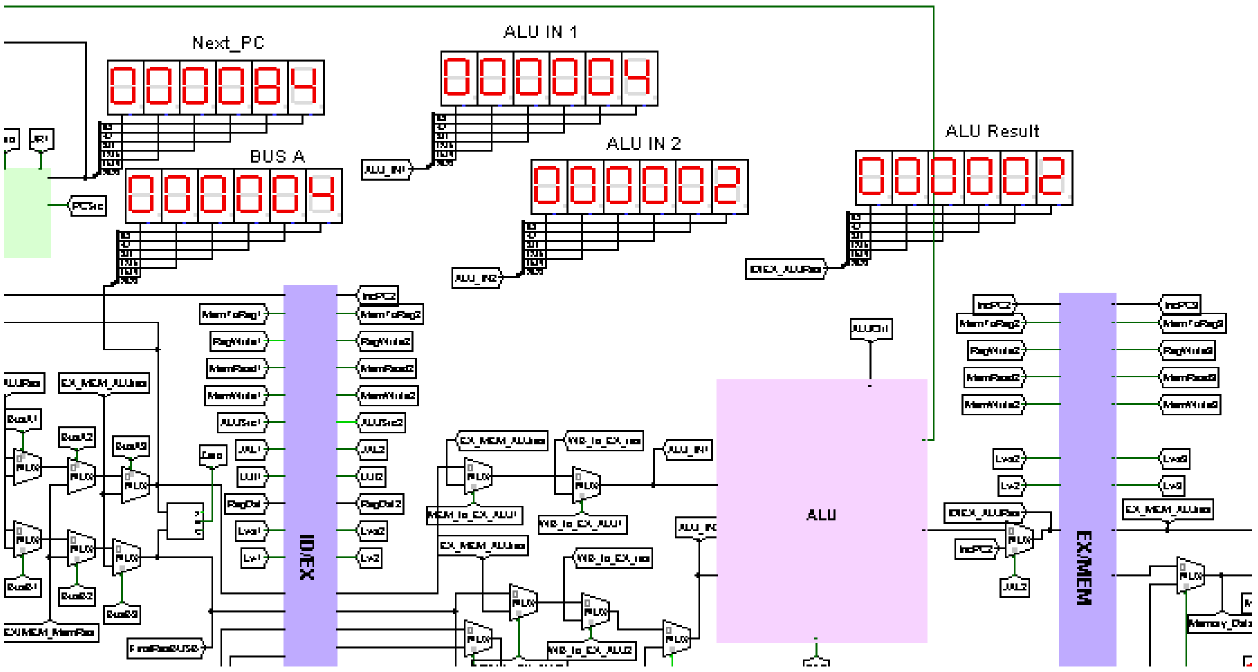


Figure 29 Testing 7

addi r2,r2,1

in this clock cycle it added 1 to R2 and put the result as 2 on the ALU result.

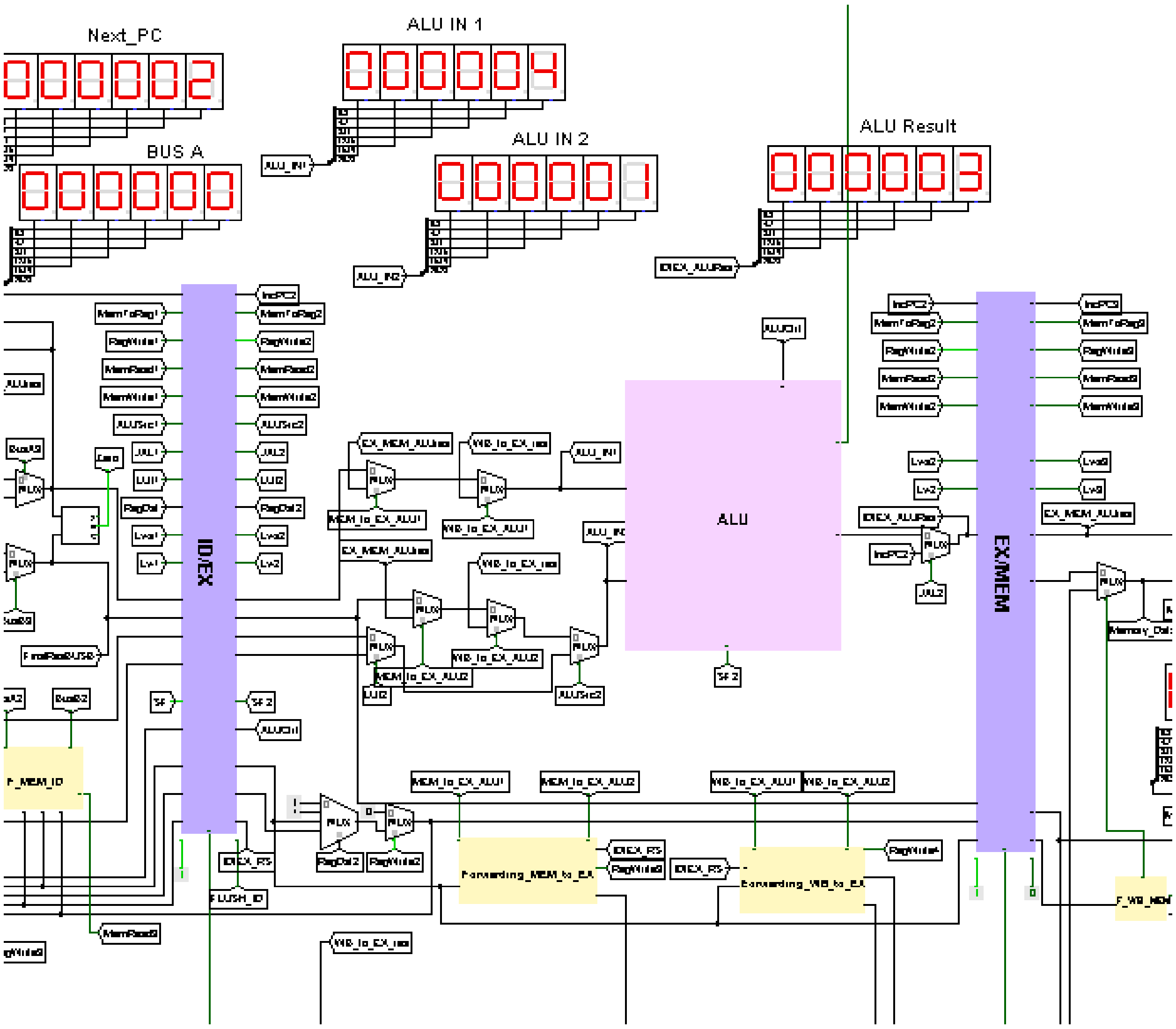


Figure 30 Testing 8

In the next clock cycle it repeated the subtract instruction as 4 - 1 = 3 and repeating this depends on the loop iterations.

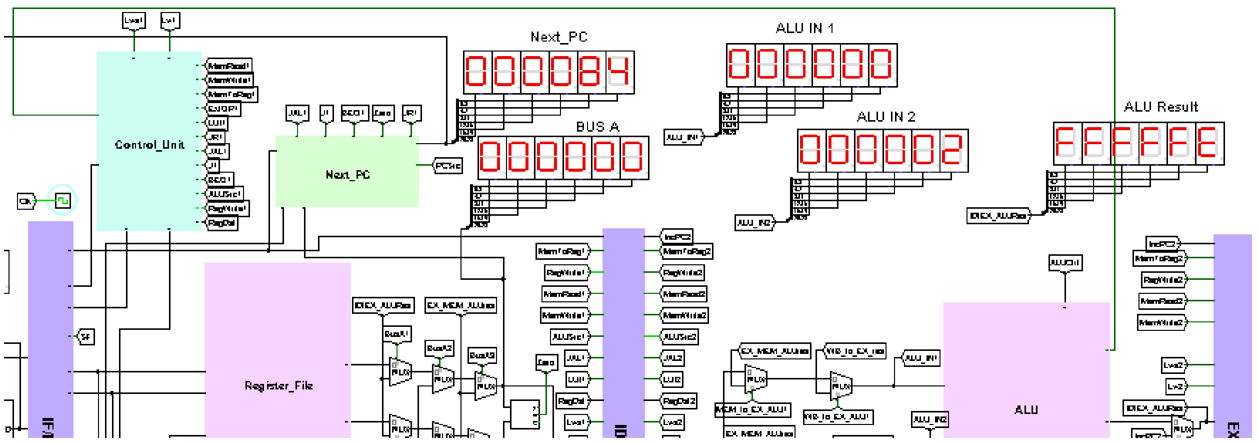


Figure 31 Testing 9

After finishing the loop the beq instruction will work and branch the loop and as seen the next PC is jumped to the label and not 4 bits as usual

Conclusion

We learnt a lot about pipelined structures and how to apply them in this project. We successfully implemented a pipelined 24-bit processor using Logisim simulator, achieving all the project's objectives, including implementing the forwarding technique and obtaining the fewest possible stall cycles, which is 1 stall cycle in the case of load followed by an instruction that relies on it (R/W) or a miss prediction in the case of a branch or jump instruction. As a long-term goal, this project motivates us

to learn more about computer architecture methodologies and how to improve the current designs.

References

Doctor slides and contacting him sometimes

geeks for geeks

stack overflow