Single Cycle Processor Design

Overview:

In the first phase of our project, we designed a simple 16-bit MIPS- like processor with seven 16-bit general purpose registers: R1 through R7. R0 is hardwired to zero and cannot be written. There is also one special-purpose 12-bit register, which is the program counter (PC). All instructions are 16 bits and there are three instruction formats: R-type, I-type, and J-type.

Through teamwork and after carefully analyzing the instruction set, we successfully designed a data path that can implement all the required instructions.

Main components:

PC Counter:

A special purpose 12-bit register containing the address of the instruction to be fetched.

Instruction Memory:

Read only memory contains the instructions of the program to be executed.

Register File:

It contains eight 16-bit general purpose register through R1 to R7 with R0 is hardwired to zero and cannot be written. Having two read ports to select the registers to be read and one write port to select which register to write and write enable bin to determine whether to write or not. Also a Data-In port having the data to be written.

Data Memory:

A RAM can be accessed with address to read or write data determined by two enable bins, one for read and another for write. Also Data In port having the data to be written.

ALU Unit:

It is considered the mastermind in the processor. It is where all the Arithmetic and Logic instructions get executed. And have multiplexers with some control signals as the selectors choosing which operation to be done.

Main Control Unit:

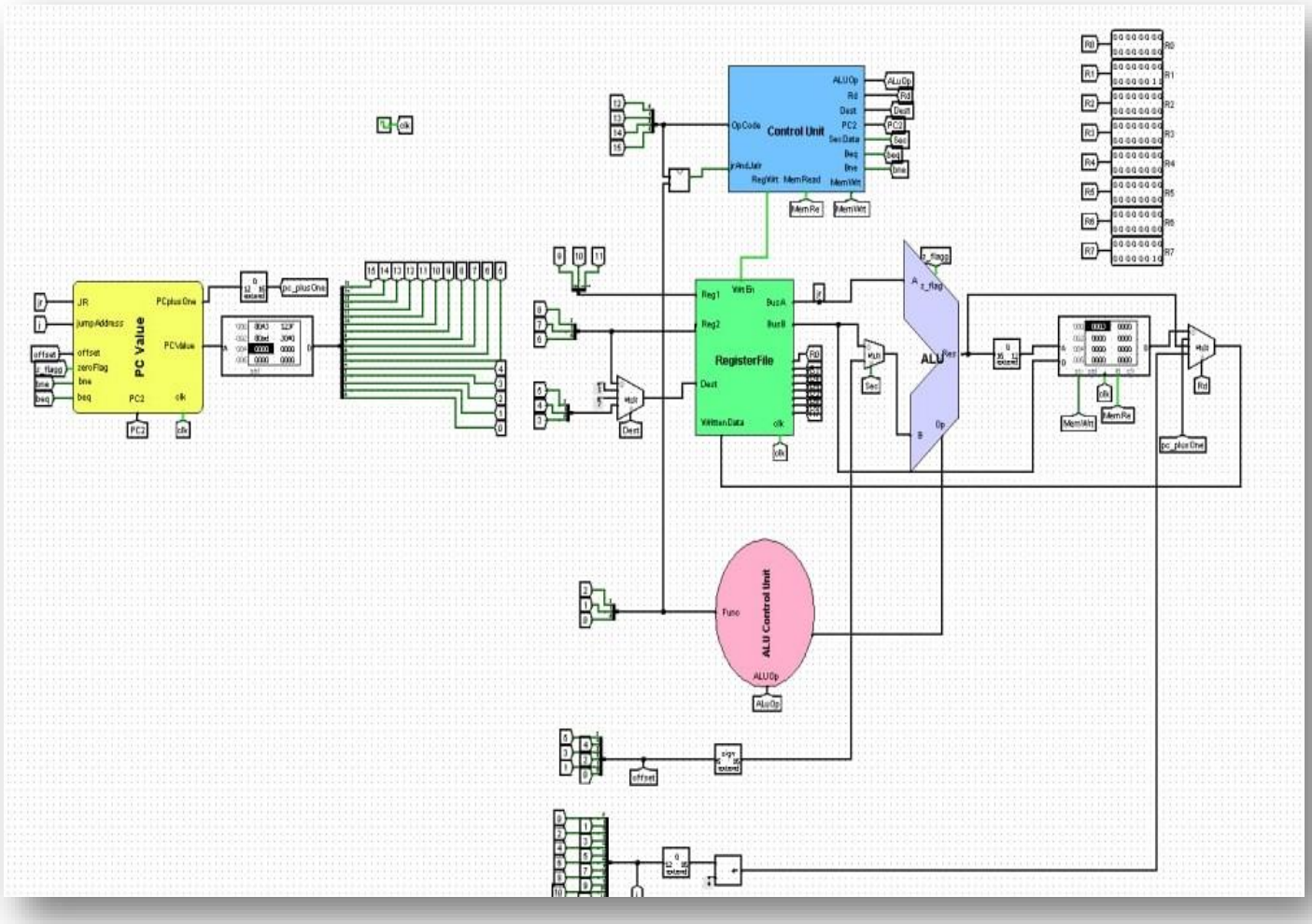
A combinational circuit generates the control signals for the whole processor to tell each part to do in which instruction given the OP Code of the instruction.

ALU Control Unit:

A circuit generates the control signals of the ALU given control signals from the main control unit and the function code from instruction.

Single Cycle

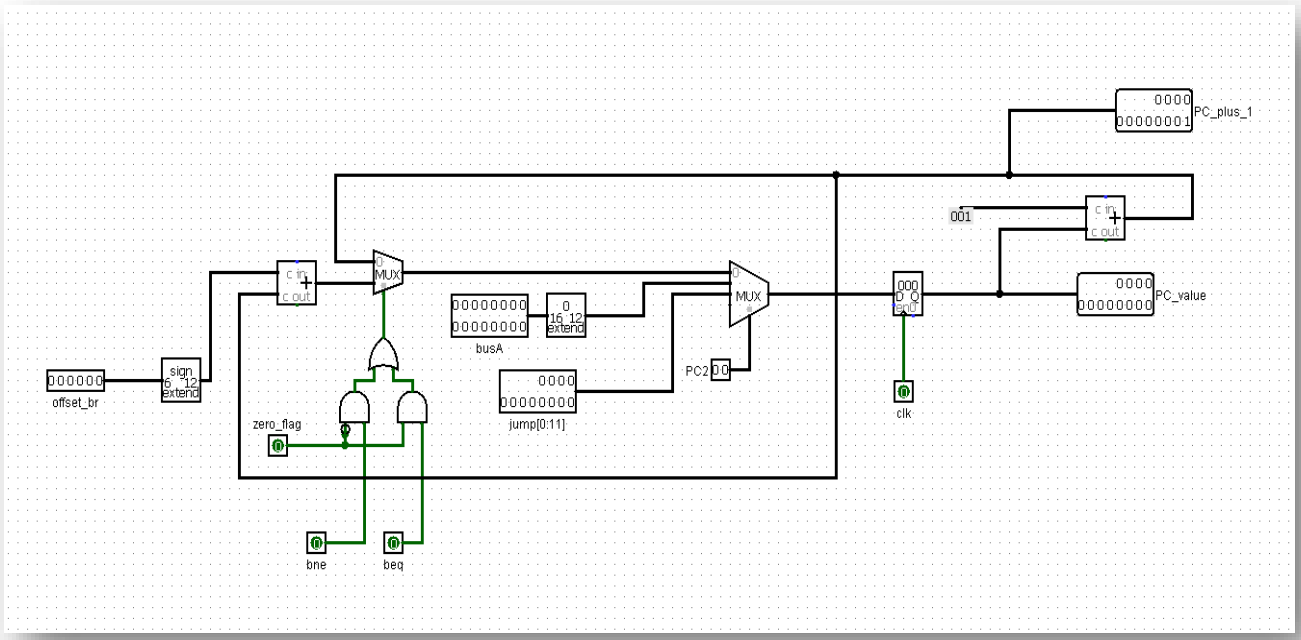
Outlook of the single cycle:



PC Counter Circuit

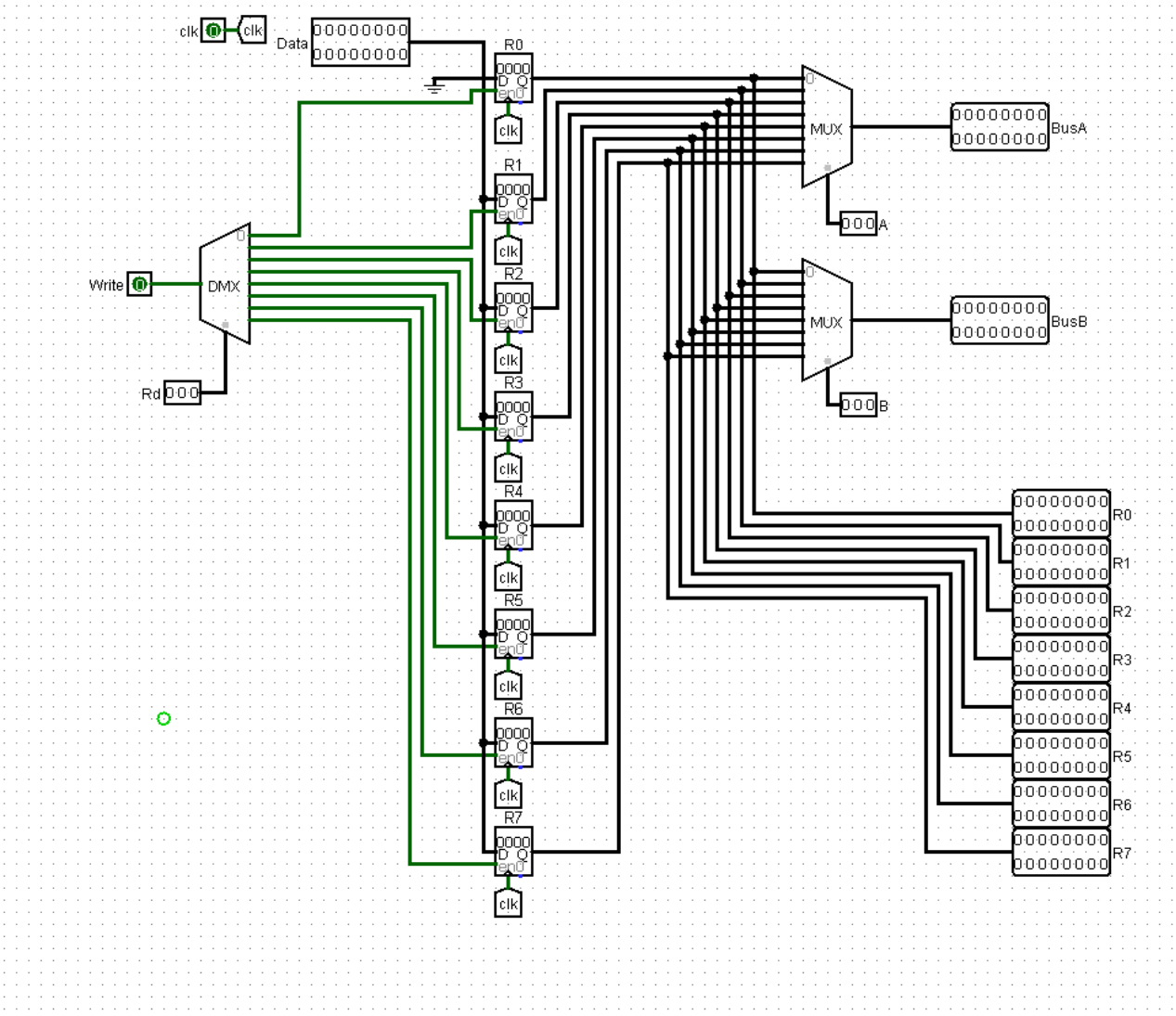
There are 4 different choices for PC input :

- PC+1: Fetching the next instruction if there are no Flow Control instructions or decision making.
- Branch Target address: PC = PC + 1 + sign-extend (6-bit immediate).
- Jump Target address: PC = 12-bit immediate (For J and JAL instructions).
- The lower 12-bit of Reg(Rs).



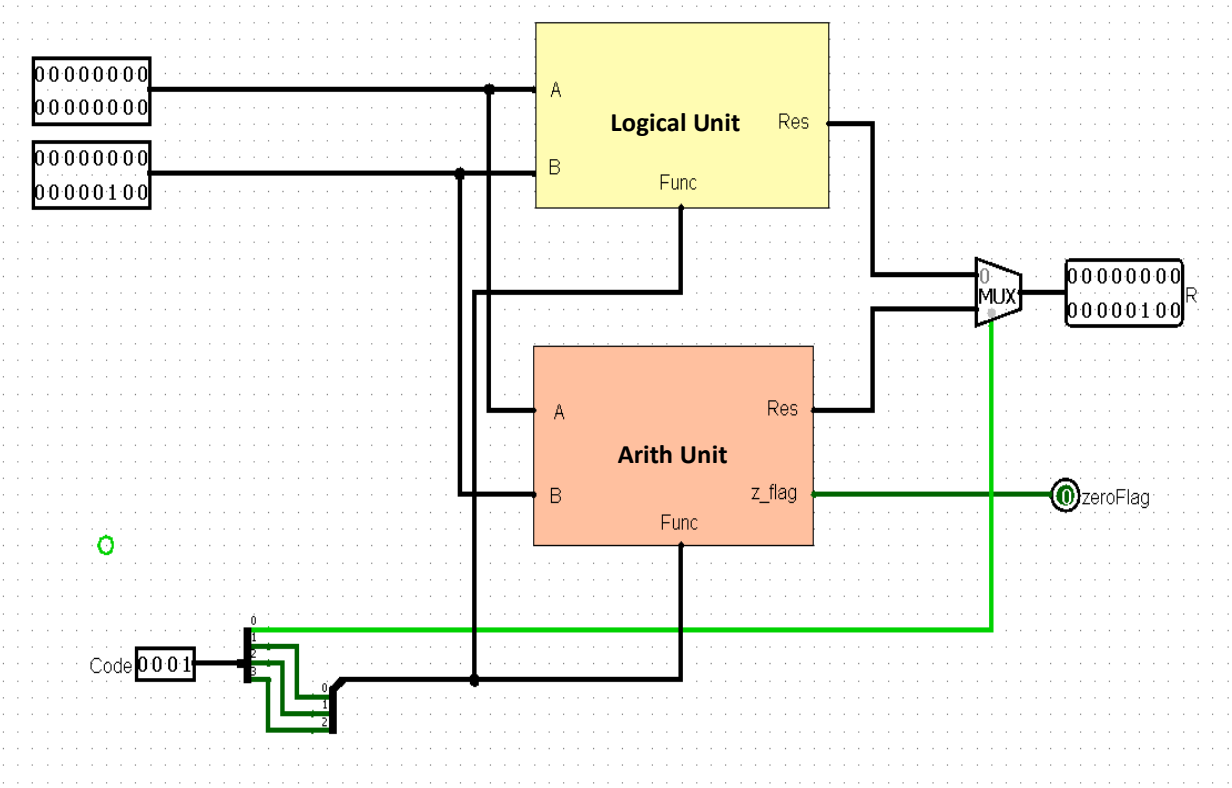
Register File Circuit

The block contains eight 16-bit Registers with R0 hardwired to zero and two read ports selects which registers to read using multiplexers and the ports as the selectors. A write enable decide whether to write or not .In addition, one write port is decoded to enable writing the destination register.


clk
clk
Data
R0
R1
R2
R3
R4
R5
R6
R7
MUX
MUX
BusA
BusB
A
B
Write
DMX
Rd
R0
R1
R2
R3
R4
R5
R6
R7

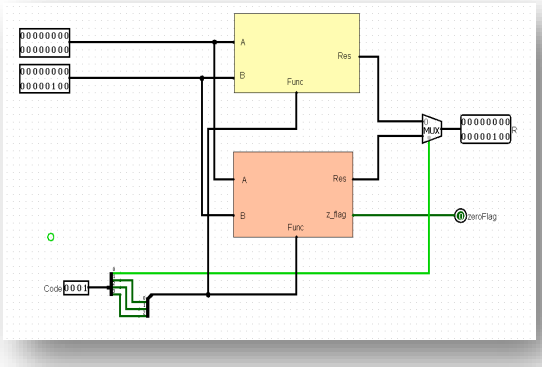
ALU Unit

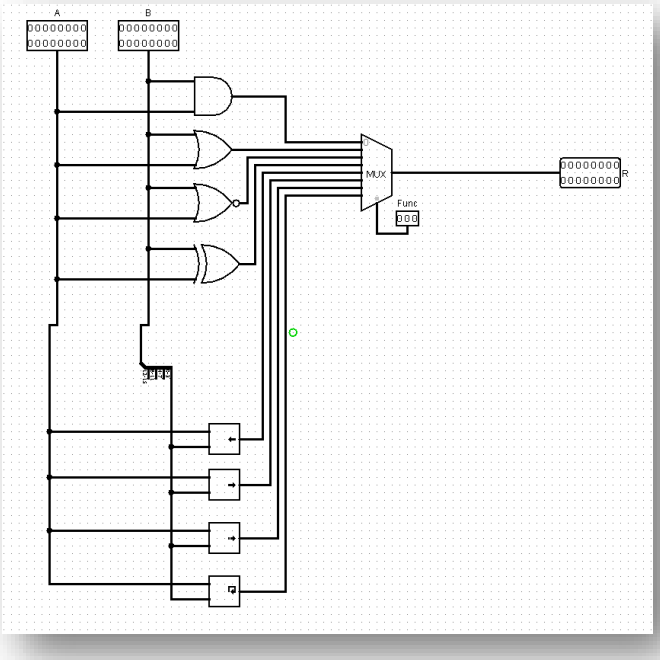
We separated the Logic Unit from the Arithmetic Unit and used a multiplexer to select whether to pass the logic or arithmetic result by using selectors generated by the ALU control unit. We also provided a Zero Flag which gives 1 if the two inputs (A&B) are equal. A Zero Flag is the key in the branching operations.



Logic Unit

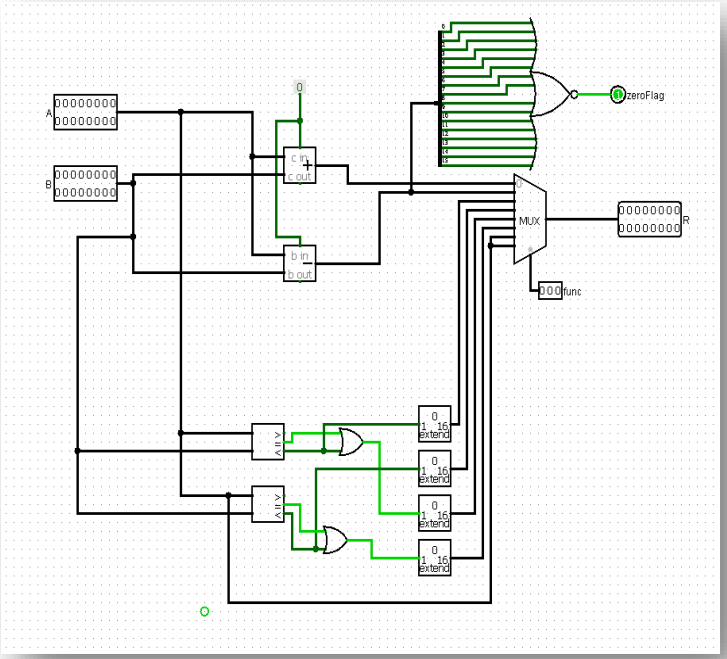
It Can do the following operations: (And, Or, Nor, Xor, Sll, Srl, Sra, Rol). Having a multiplexer selects the operation to pass using the func as the selector.





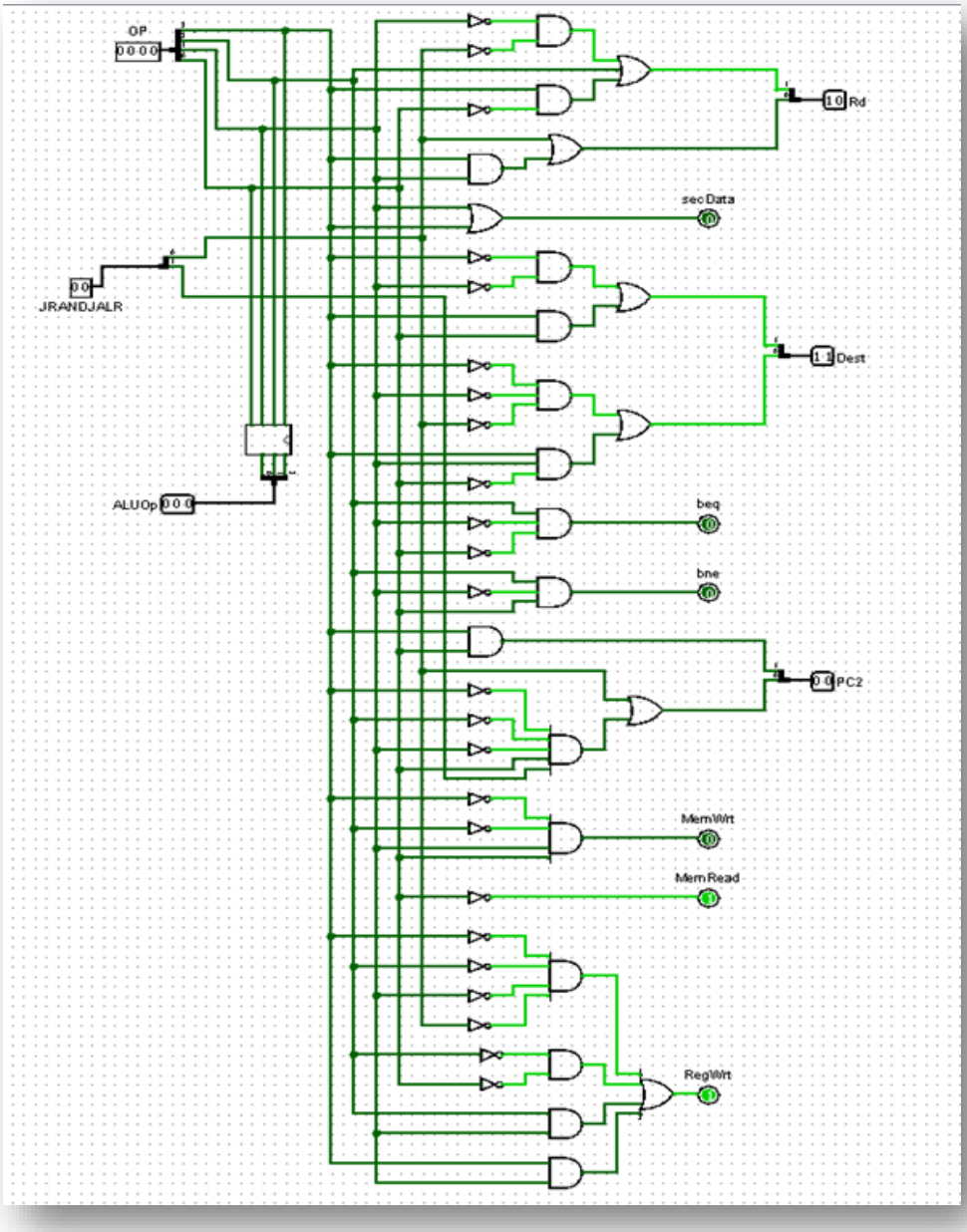
Arithmetic Unit

It can do the following operations (Add, Sub, Slt, Sltu, Slte, Slteu). Having a multiplexer selects the operation to pass using the func as the selector



Main Control Unit

This circuit is designed to receive two inputs, the op code of the instruction and a flag for jump register “JR” and jump and link “JALR” instructions to differentiate them from other R-Type instructions.
On the other hand, it is designed to generate signals to almost all components of the processor indicating which buses and paths instruction and data should take.


OP
0 0 0 0
JRANDJALR
0 0
ALUOp
0 0 0
1 0 Rd
secData
0
1 1 Dest
beq
0
bne
0
0 0 PC2
MemWrt
0
MemRead
0
RegWrt
0

The main control unit circuit is fully implemented by Logisim software after designing the data path and listing signals data as follows:

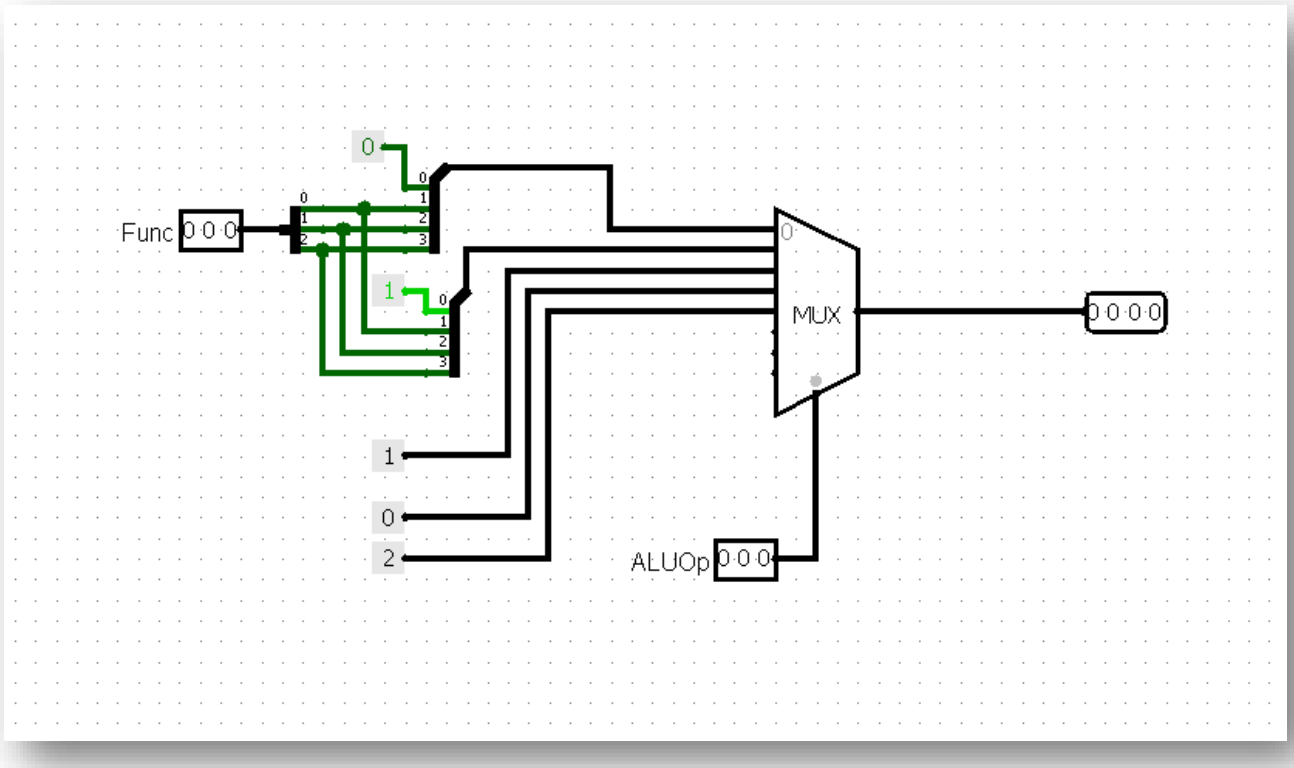
Instruction	Op	Rd data	SecData	Dest	PC1		PC2	MemWrt	MemRead	RegWrt	ALU Op
					beq	bne					
AND	0000	2	0	3	0	0	0	0	x	1	0
OR	0000										0
NOR	0000										0
XOR	0000										0
SLL	0000										0
SRL	0000										0
SRA	0000										0
ROL	0000										0
ADD	0001										1
SUB	0001										1
SLT	0001										1
SLTU	0001										1
SLTE	0001										1
SLTEU	0001										1
JR	0001	1					1			0	1
JALR	0001									1	1
LW	0010	0	1	0	0	0	0	0	1	1	2
SW	0011	x	1	0	0	0	0	1	0	0	2
ANDI	0110	2	1	0	0	0	0	0	x	1	3
ORI	0111	2	1	0	0	0	0	0	x	1	4
ADDI	1000	2	1	0	0	0	0	0	x	1	2
BEQ	0100	x	0	x	1	0	0	0	x	0	x
BNE	0101	x	0	x	0	1	0	0	x	0	x
BLT	1101	x	0	x	0	1	0	0	x	0	x
BGT	1111	x	0	x	0	1	0	0	x	0	x
BLTZ	1100	x	0	x	0	1	0	0	x	0	x
BLEZ	1100	x	0	x	0	1	0	0	x	0	x
BGTZ	1110	x	0	x	0	1	0	0	x	0	x
BGEZ	1110	x	0	x	0	1	0	0	x	0	x
J	1001	x	x	x	x	x	2	0	x	0	x
JAL	1011	1	x	2	x	x	2	0	x	1	x
LUI	1010	3	x	1	0	0	0	0	x	1	x

ALU Control Unit

It generates the control signals of the ALU. This circuit is designed to receive two signals, the function code [3 least significant bits of R-Type instruction] and a signal from main control unit works as a selector. Previously, we indicated that ALU is divided into two separated units [logical/arithmetic], they almost run on the same function codes which is 3-bits, we designed the ALU CU to generates 4-bits signal, where the least significant bit indicating to produce the result of logical unit [0] or the arithmetic one [1] at the function code [the other 3-bits]

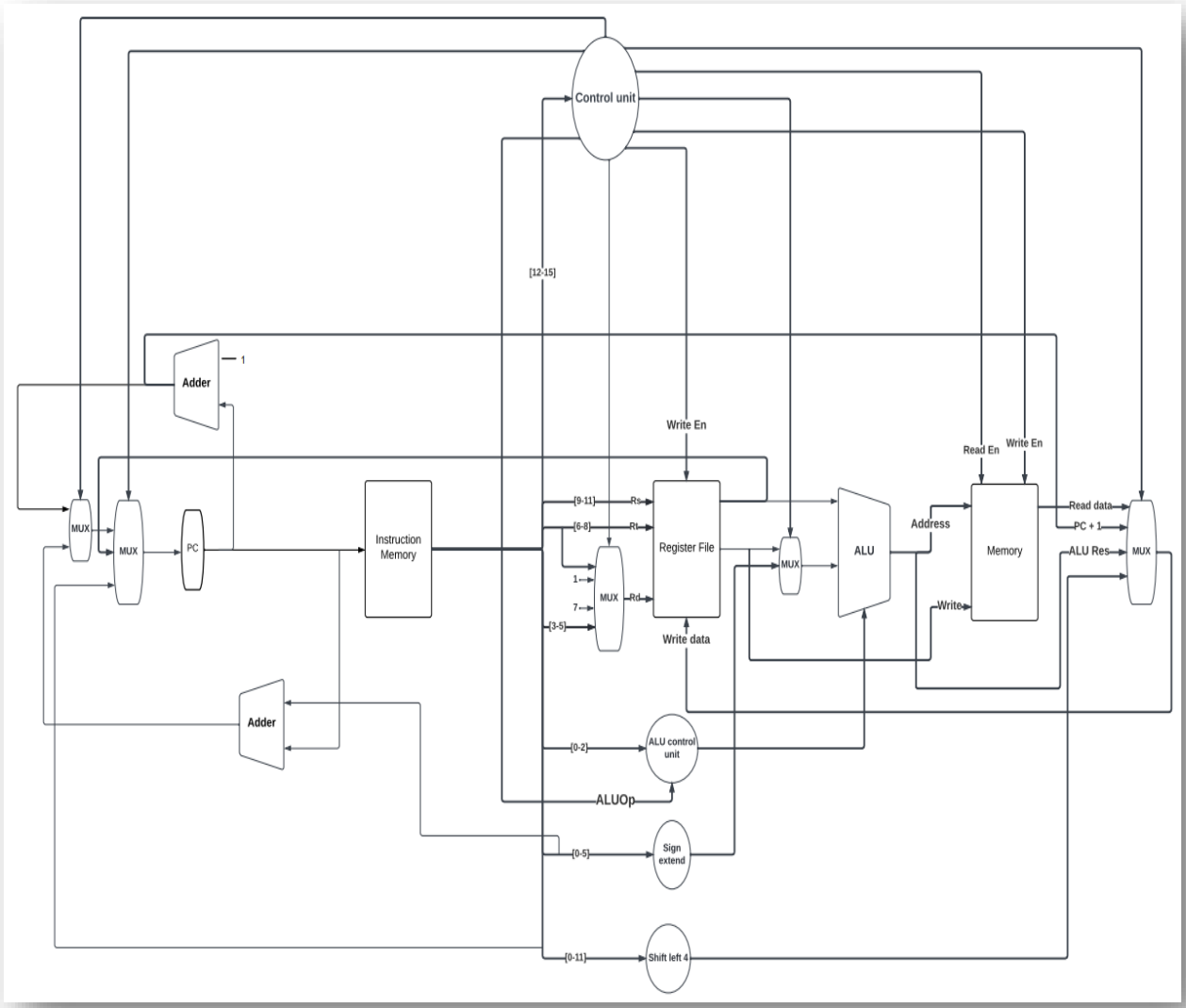
The other signals on MUX are passing when the instructions are one of

[ANDI, OR, ADDI, LW or SW].



Data Path:

We started designing our processor by carrying out the following diagram for the data path.

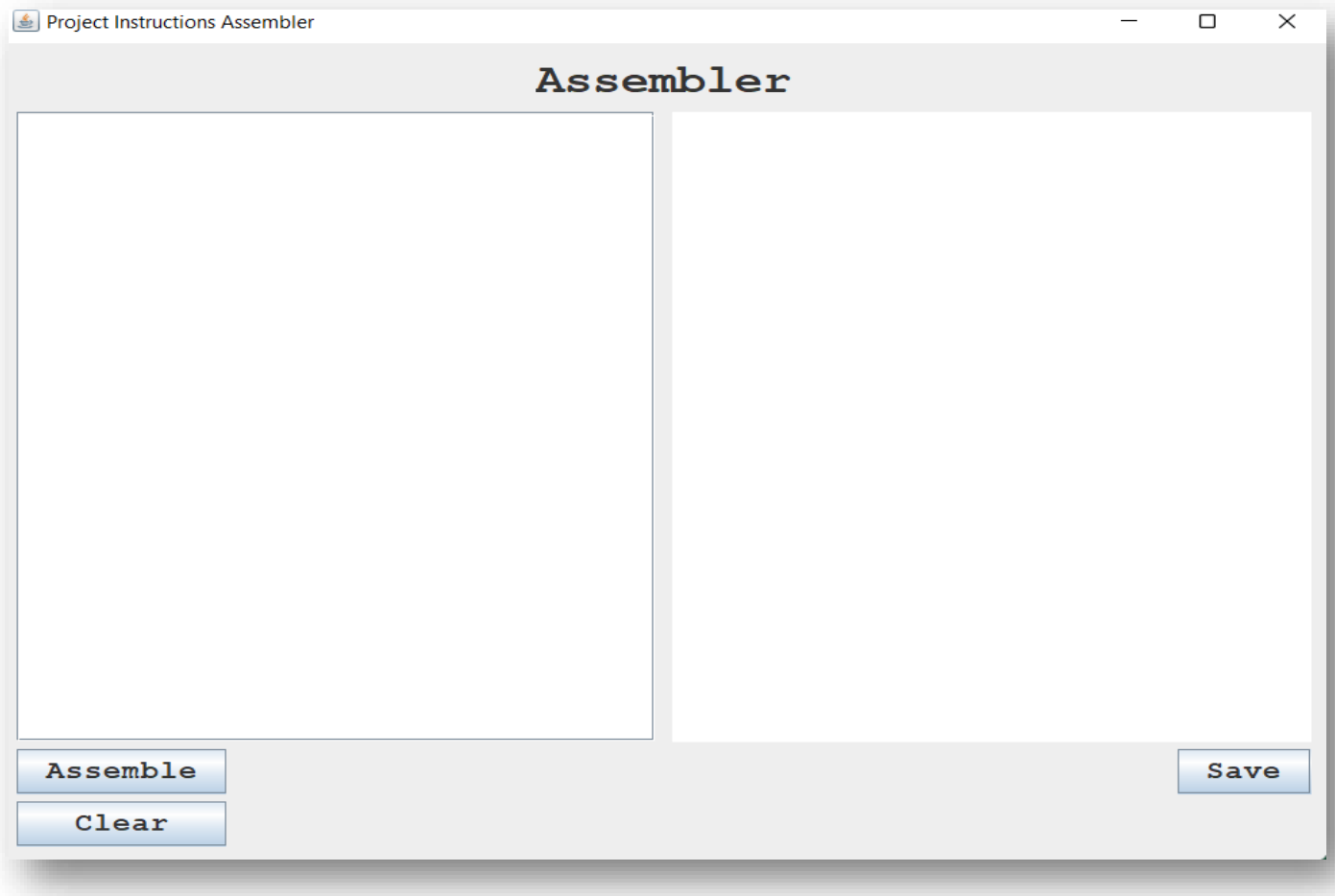

Control unit
[12-15]
Adder
1
Write En
Read En
Write En
MUX
MUX
PC
Instruction Memory
[9-11]
Rs
[6-8]
Rt
1
7
[3-5]
MUX
Rd
Register File
Write data
MUX
ALU
Address
Memory
Write
Read data
PC + 1
ALU Res
MUX
Adder
[0-2]
ALU control unit
ALUOp
[0-5]
Sign extend
[0-11]
Shift left 4

ISA Compiler (Bonus work)

Since each instruction will be converted to machine code and send to processor circuit to be executed. Moreover, we need to translate the binary code representing the instruction to hexadecimal code due to simulate it using logisim that required the data to be loaded in hexadecimal format.

Our team developed a compiler to do this job in order to simplify our work.

The way of using our compiler is so simple, you just write your code and then click assemble and it will translate it into hexadecimal code and save the file.



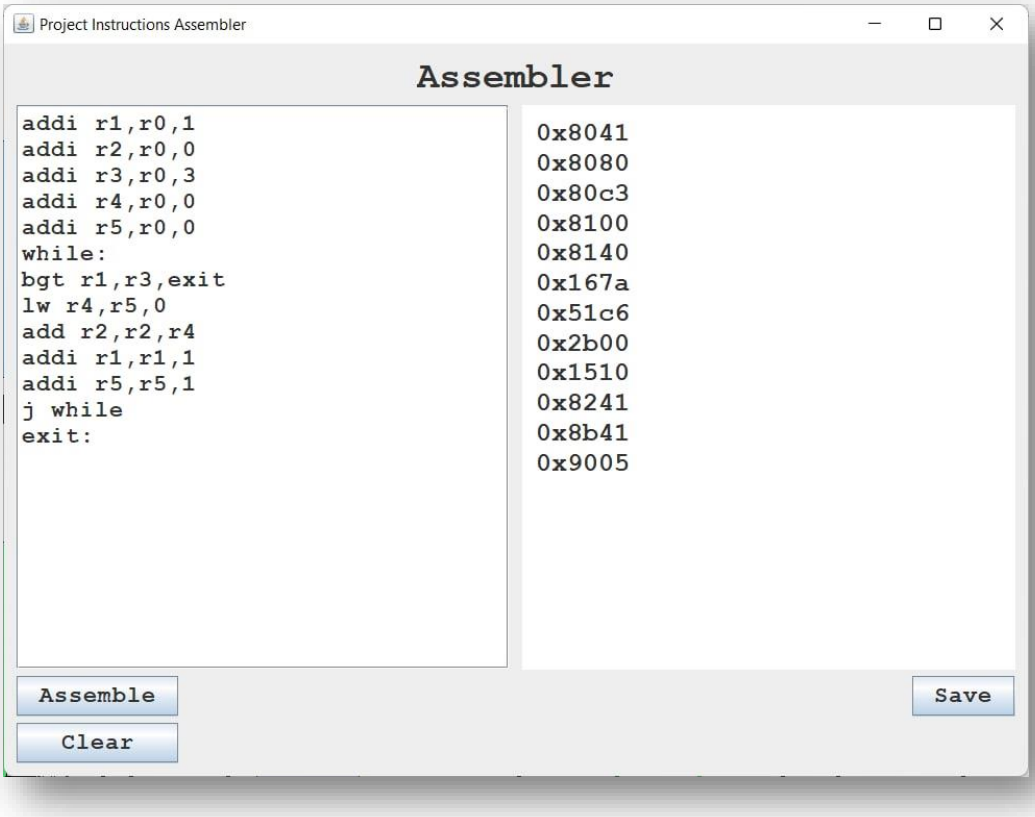
Test Programs:

Program 1 (Adding the elements of array) :

Java code:

```
int arr [] = {2,4,6};

int size =3;

int sum = 0;

For(int i=1; i<=3; i++){sum += arr[i];}
```

MIPS Code:



Expected values:

R1 = 4

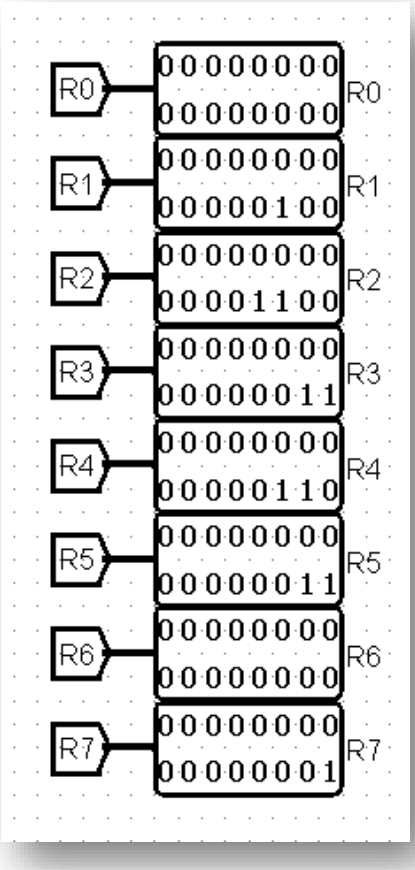
R2 = 12 (Final Result)

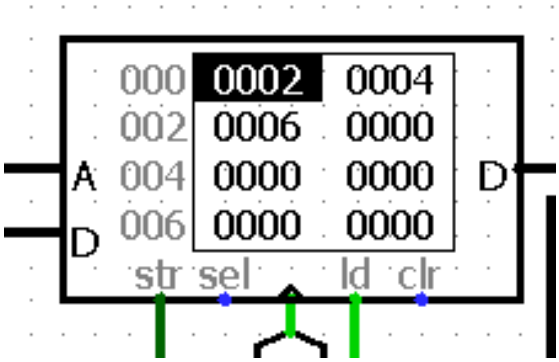
R3 = 3 (size of array)

R4 = 6 (last element of the array)

R5 = 3 (counter)

Register File after implementing the program:



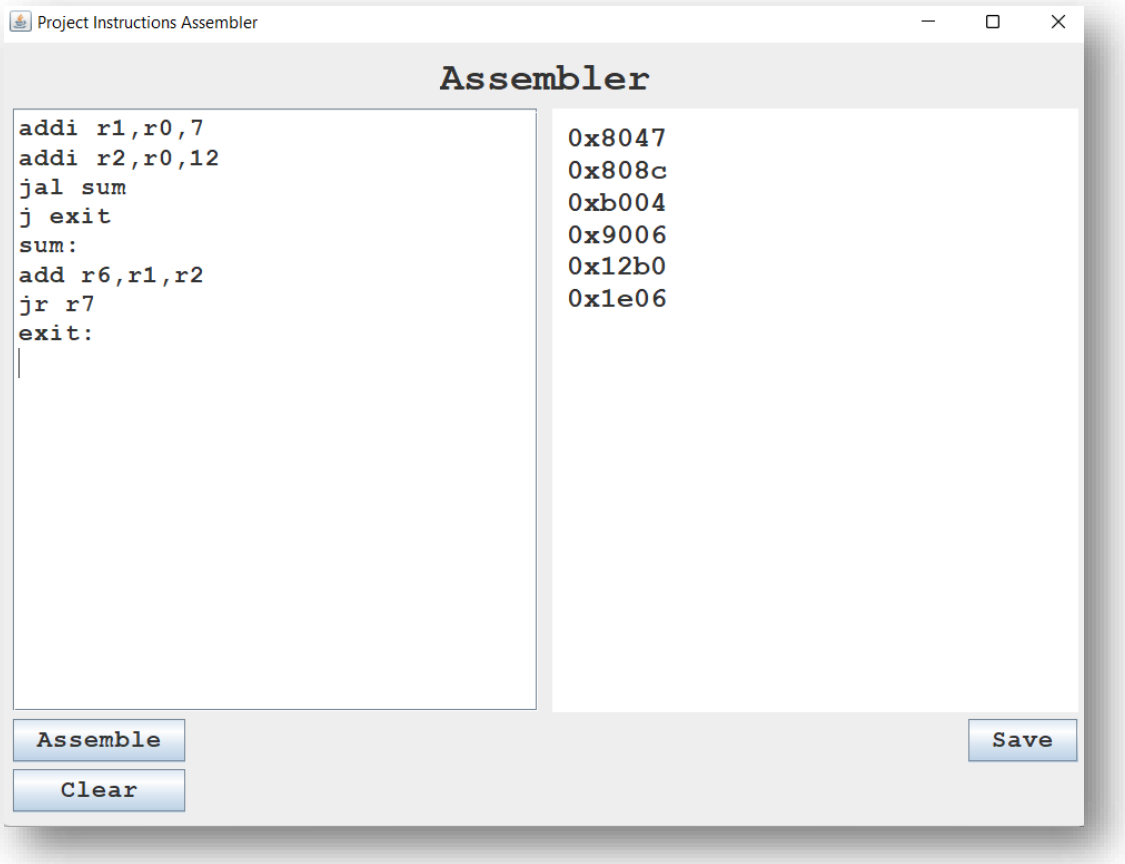


Program 2 (Function to sum two integers):

Java Code:

```
 int a = 7;

int b = 12;

int c = sum(a,b);

public static int sum(int a , int b){

return a+b;}
```

MIPS Code:



Expected values:

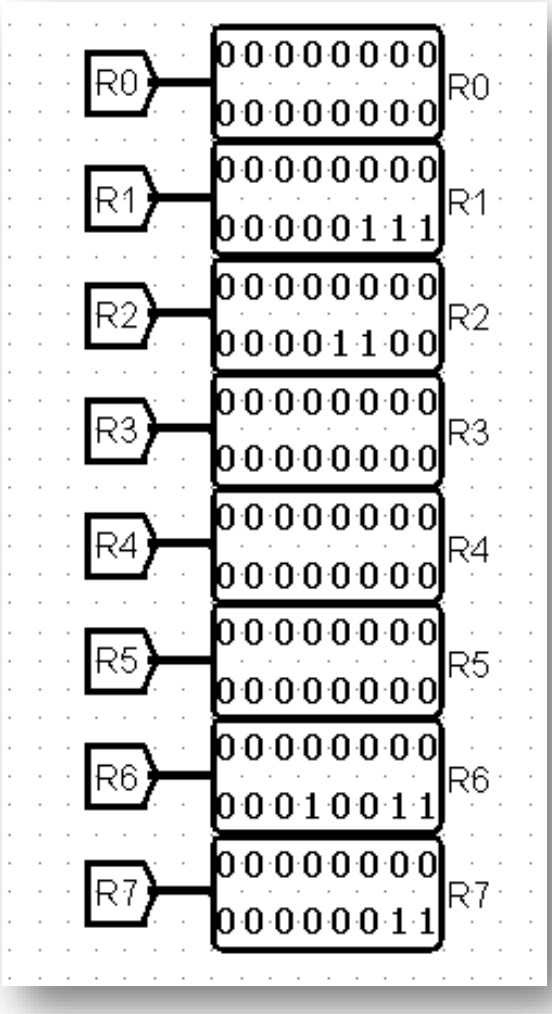
R1 = 7

R2 = 12

R7 = 3

R6 = 19

Register File after implementing the program:

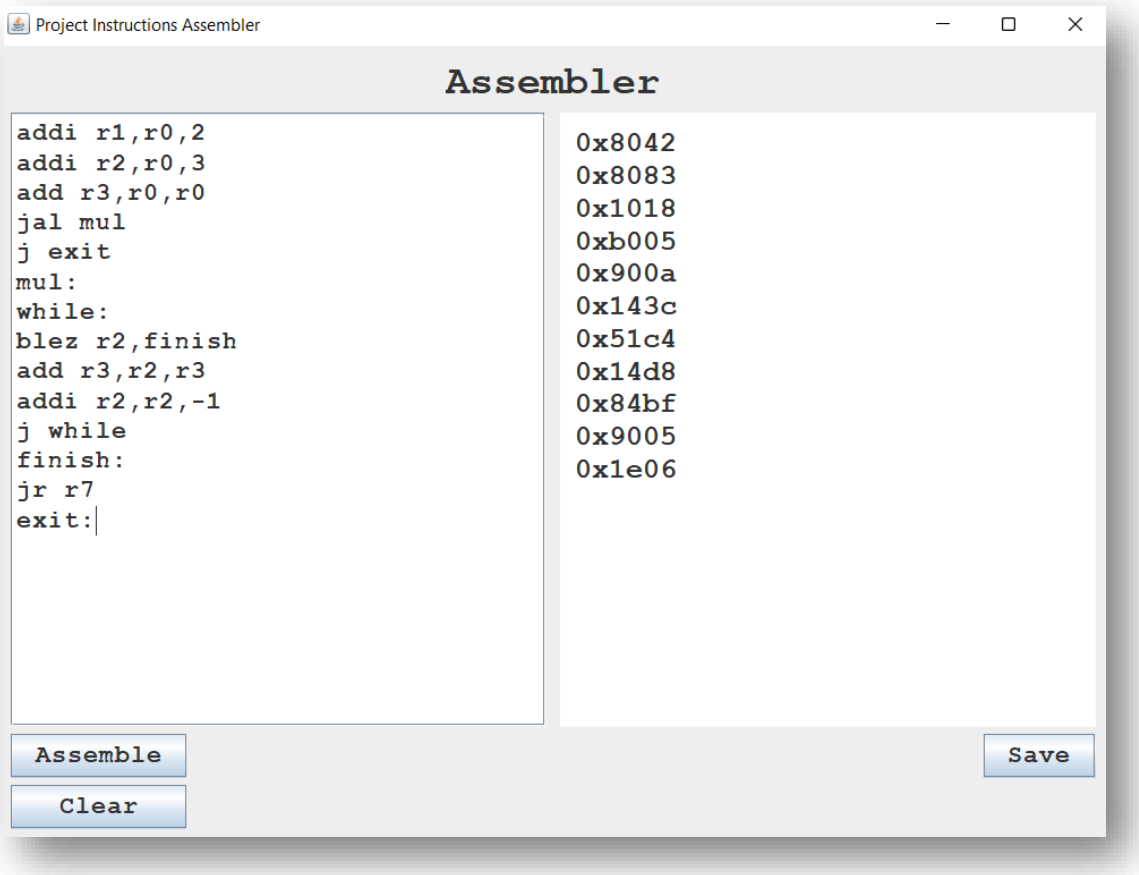


Program 3 (Function to multiply two integers) :

Java Code:

```
int a = 2; int b = 3;

int c = mul(a,b);

public static int mul (int a , int b){

int result=0;

while (b>0){ result+=a;

b--;} return result;}
```

MIPS Code:



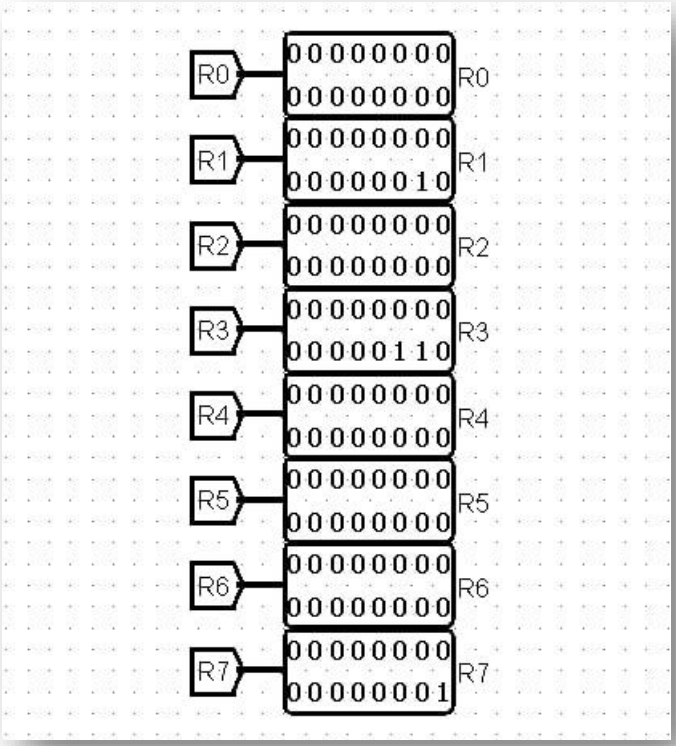
Expected values:

R1 = 2

R2 = 0

R3 = 6

Register File after implementing the program:

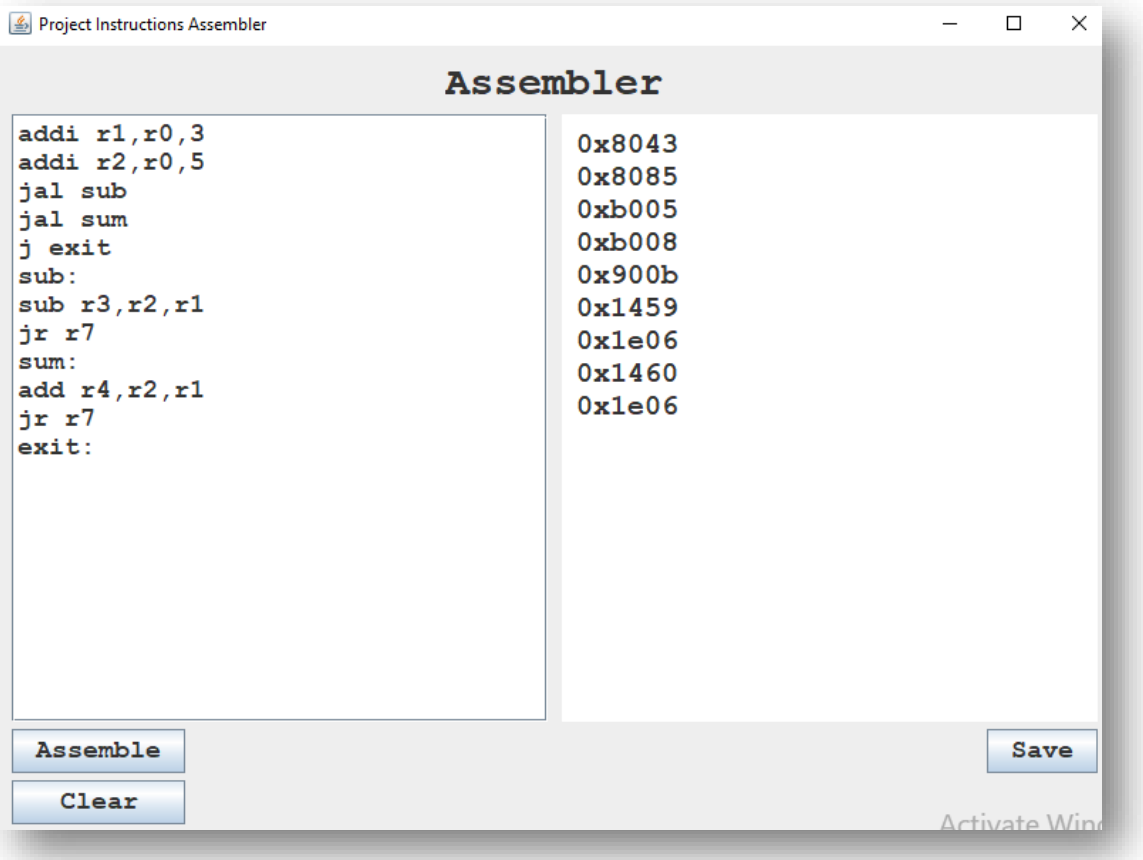


Program 4 (Calling two functions):

Java Code:

```
int a = 3;

int b = 5;

int c = sum(a,b);

int d = sub(b,a);

public static int sum(int a, int b){ return a+b;}

public static int sub(int a, int b){ return a-b;}
```

MIPS Code:



Expected values:

R1 = 3

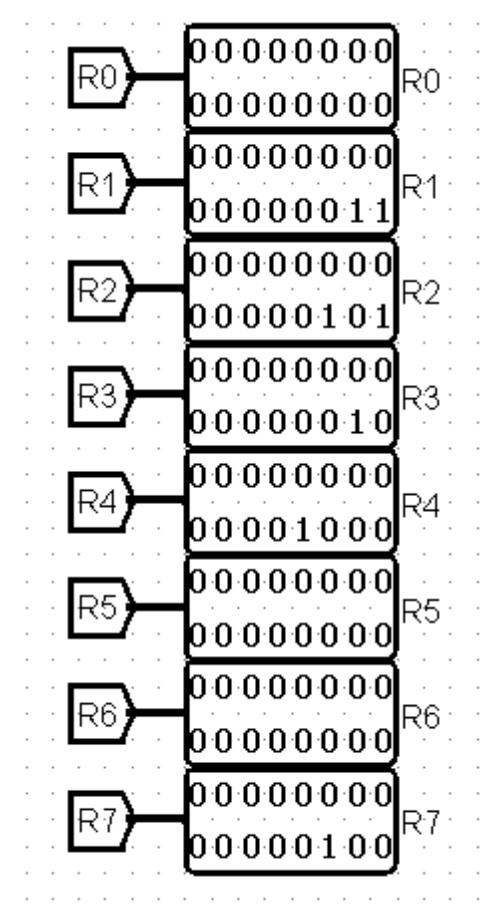
R2 = 5

R3 = 2

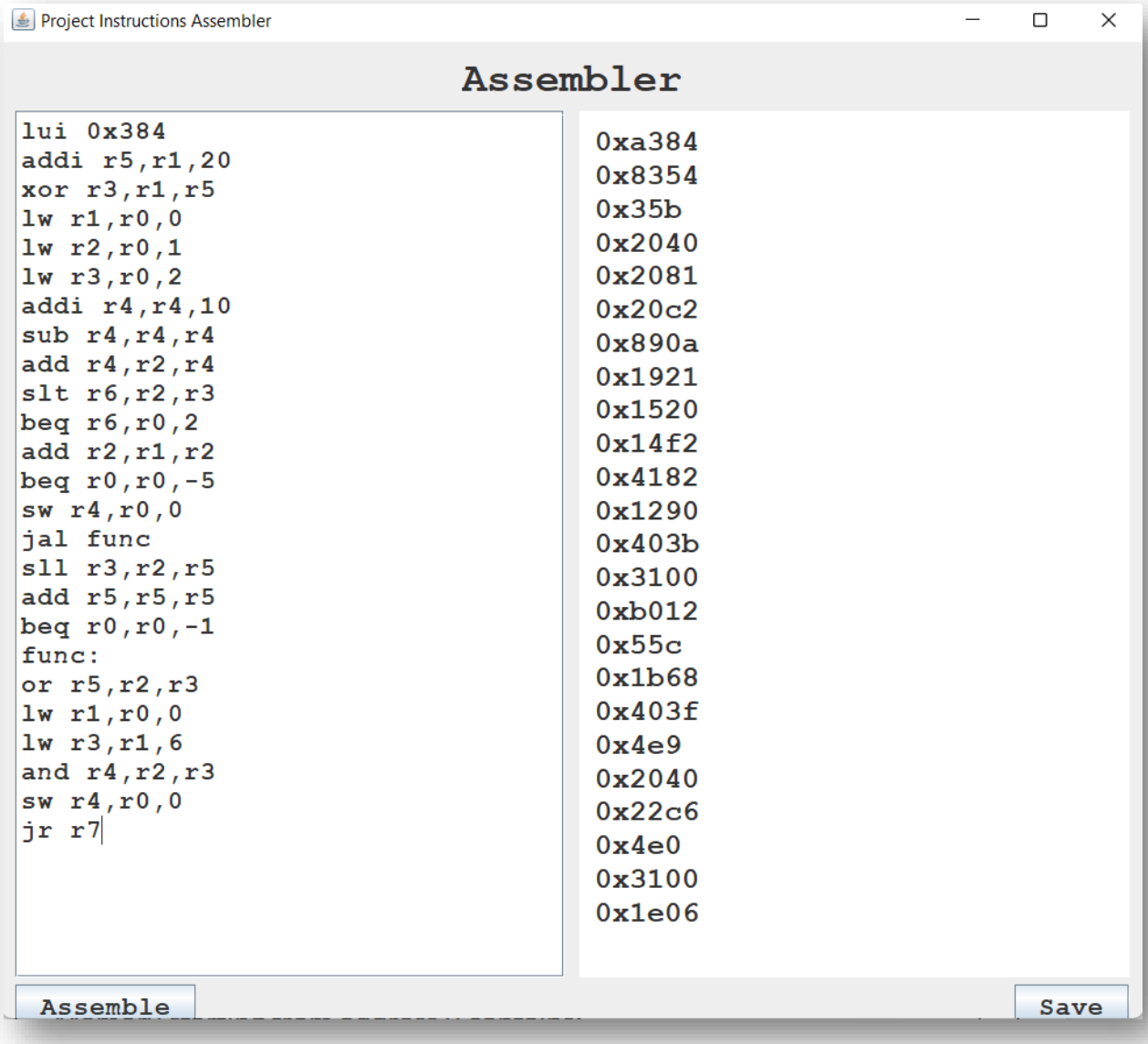
R4 = 8

R7 = 4

Register File after implementing the program:



Test Code:



Expected values:

R1 = 0x37

R2 = 0x430a

R3 = 0x2800

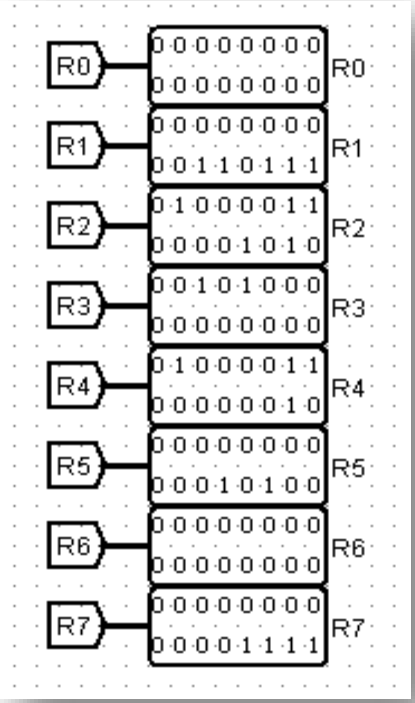
R4 = 0x4320

R5 = 0x14

R6 = 0x0

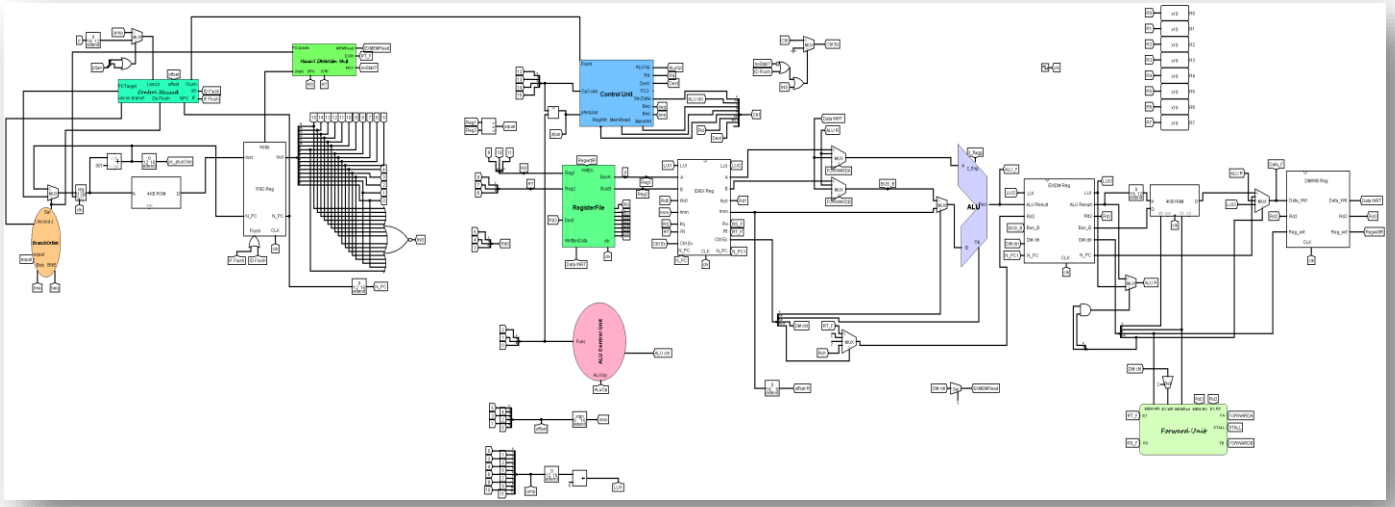
R7 = 0xf

Register File after implementing the program:



Pipeline Process

In The pipelining process, we worked mainly on increasing the performance by decreasing the total number of clock cycles taken to execute a specific program of instructions achieving Throughput.

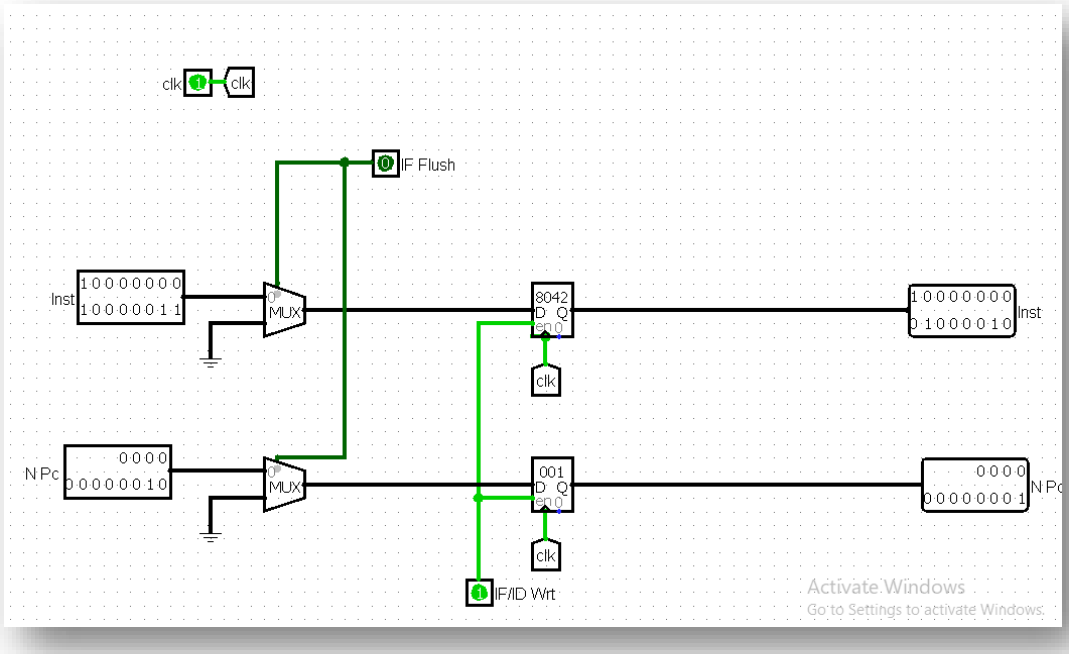


Firstly, we installed “Base Pipeline Registers“_on Our Design of the single cycle_ that store particular information about current instruction while next instruction is being passed simultaneously.

Thereby, We applied four Base Registers as follows:

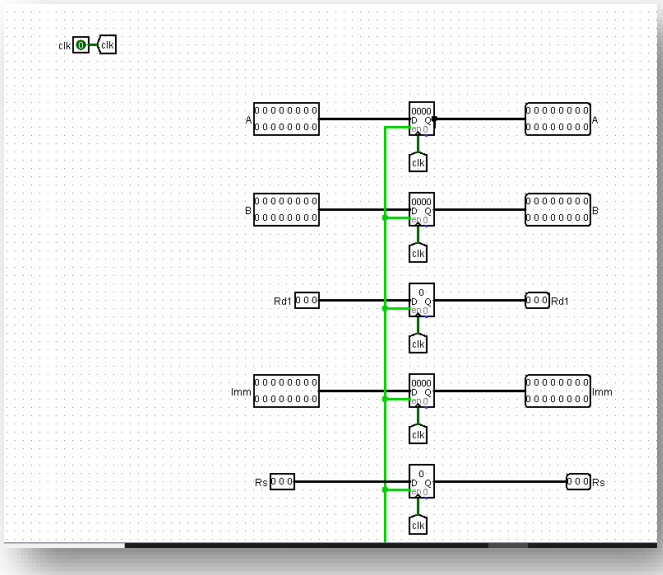
IF/ID REGISTER

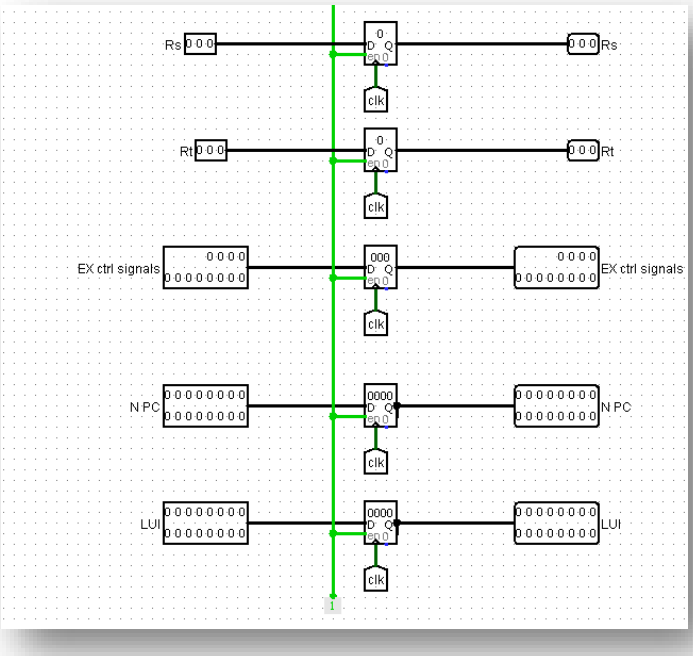
In Which, we stored the 16_bit Instruction and the 12_bit Next PC till the next clock cycle.



ID/EX REGISTER

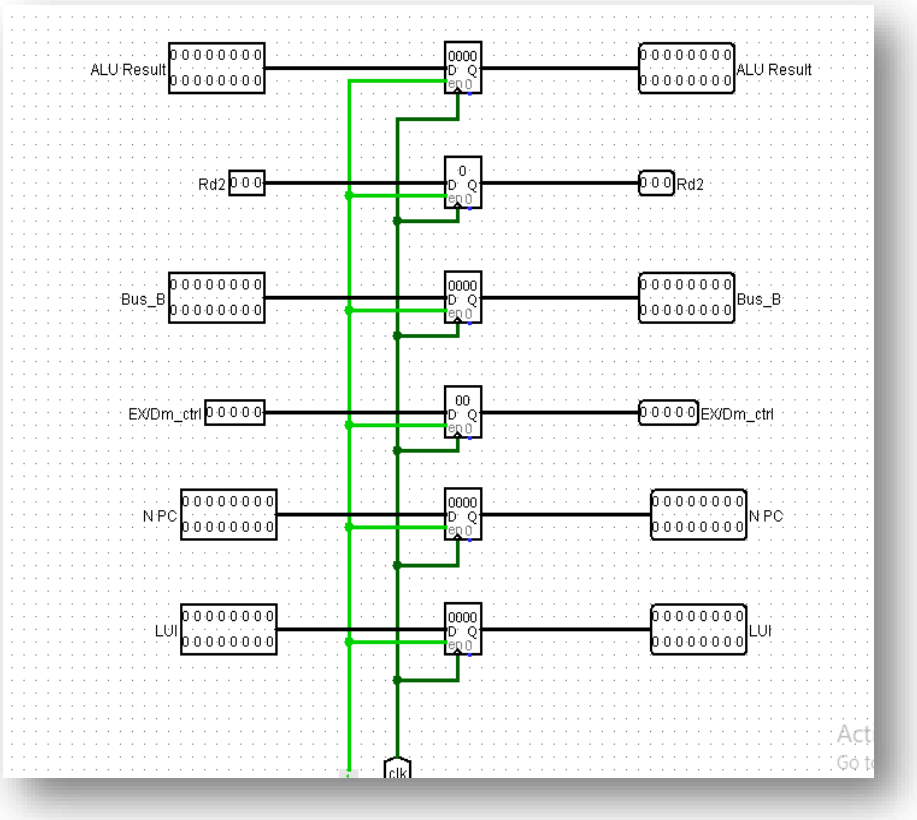
In Which, After Decoding The passed instruction from the previous register. We now store the data of [3-bit (Rs, Rt, Rd), 16_bit (BUS A, BUS B), Imm_6bit, 12_bit NPC, 16_bit of LUI instruction: to pass it over till the mux of WB just in that case, control signals obtained through control unit of which some signals are consumed and others are to be passed to successive stages].


clk
A
B
Rd1
Imm
Rs
clk


Rs
Rt
EX ctrl signals
N PC
LUI
clk

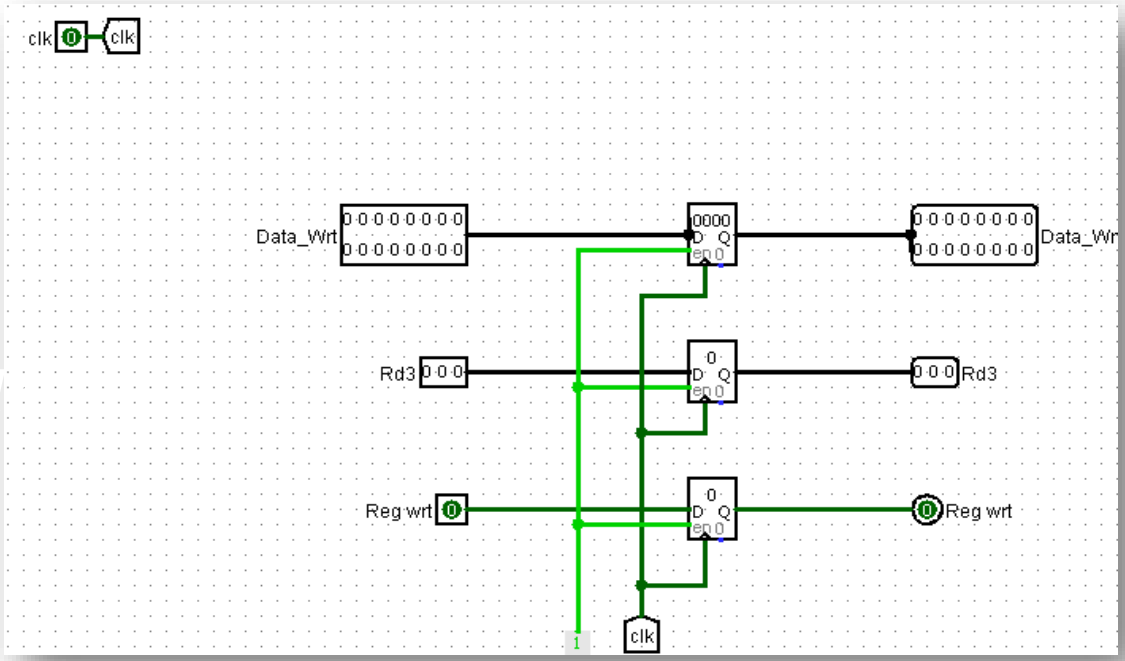
EX/MEM REGISTER

In Which, Storing important data coming from the Alu unit (16_bit ALU Result), besides the remaining control signals, 16_bit LUI, 12_bit N_PC, 3_bit Rd number, 16_bit BUS B to be passed over again to the next register.



MEM/WB REGISTER

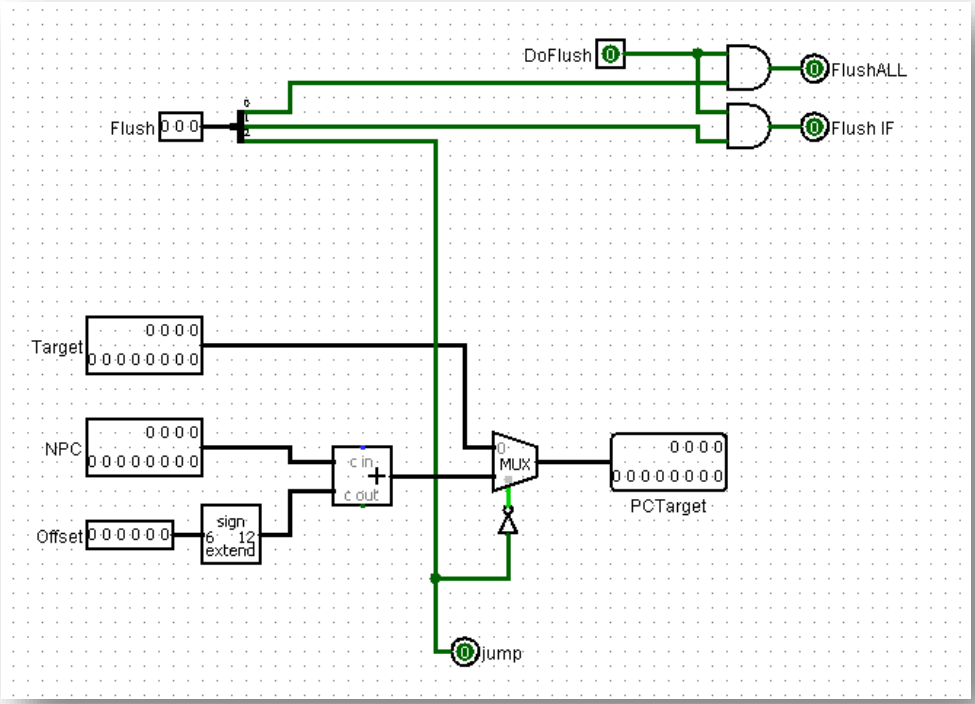
In which, we finally store the data to be written on the destination register in the RegFile and the control signal RegWr to enable or disable writing on the RegFile



Control Hazards Approach

In this circuit, we started to handle control Hazards of Branch and jump instructions that play a great role in changing the address of the next instruction to be executed and whether to flush any coming instruction or waiting some stall before flushing or not to flush at all.

According to our design we prepare the pc_target whether it is the 12_Imm bits of J instruction or Jr instruction[forwarded from Bus A] or it is The branch 6_Imm extended + N_pc and the selector signal is The third bit of the Flush input bits.

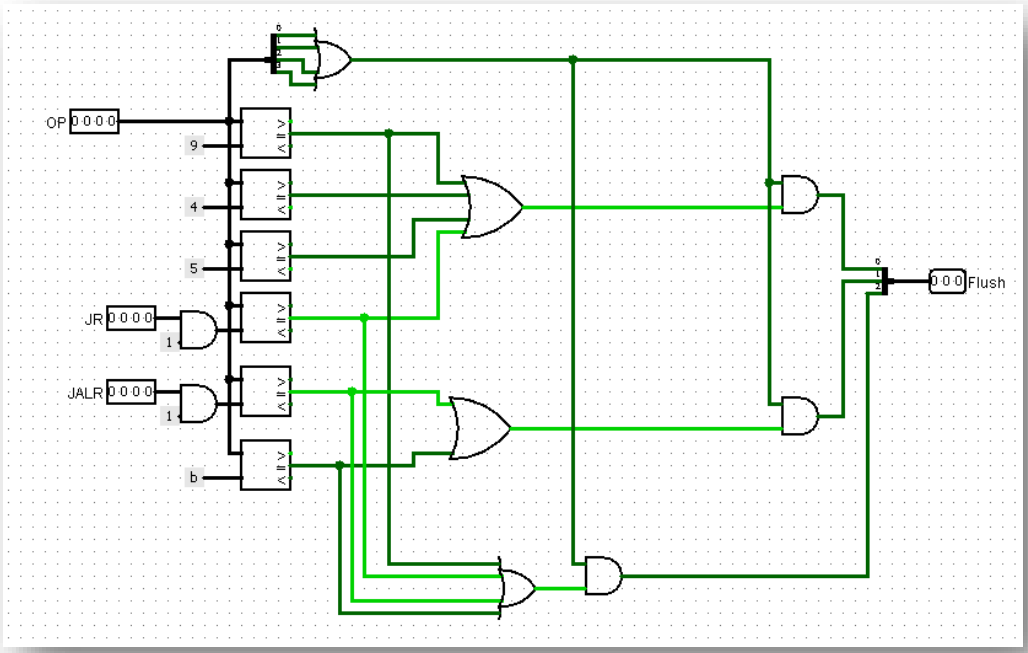


Flushing Unit

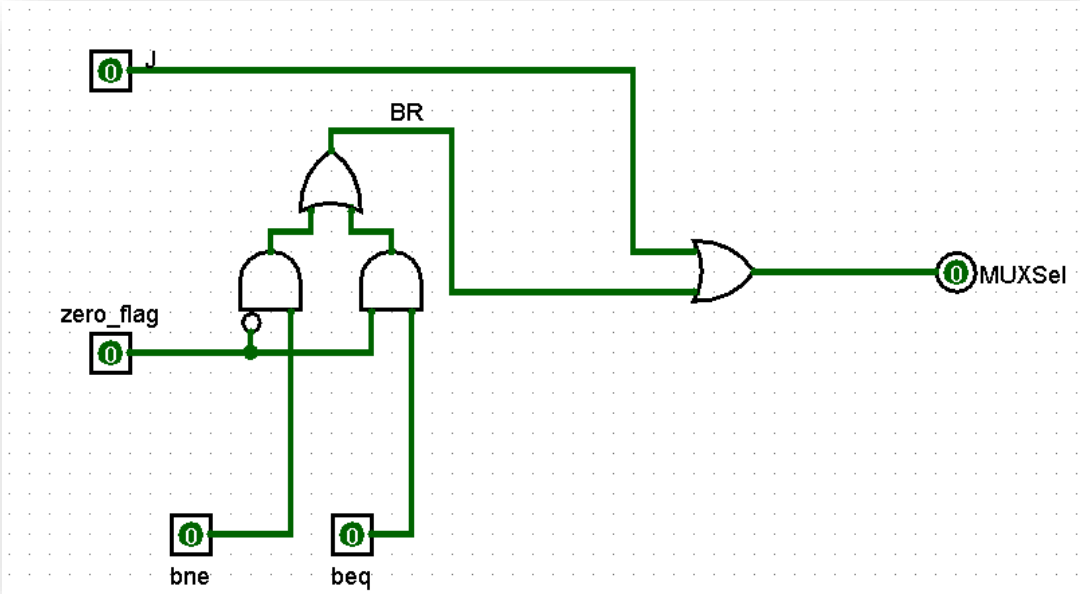
Which takes The OP_CODE as an input to check whether it's [J, BEQ, BNE, JR, JALR, JAL] instructions and takes some consecutive decisions afterwards.

It indicates if it's going to Flush both [Next instruction & the coming control signals of the current instruct] as in [J, BEQ, BNE, JR] instructions or to ONLY Flush the Next instruction and keep the control signals of the current instruction as it is that are used and needed afterwards like [JALR]: that needs to write The N_PC on R7 and [JAL] with a similar concept.

Besides detecting in The Third Bit whether it's an unconditional jump or a conditional Branch



Branch Or Not Unit



Forward Unit:

The Forwarding unit is a simple solution based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard.

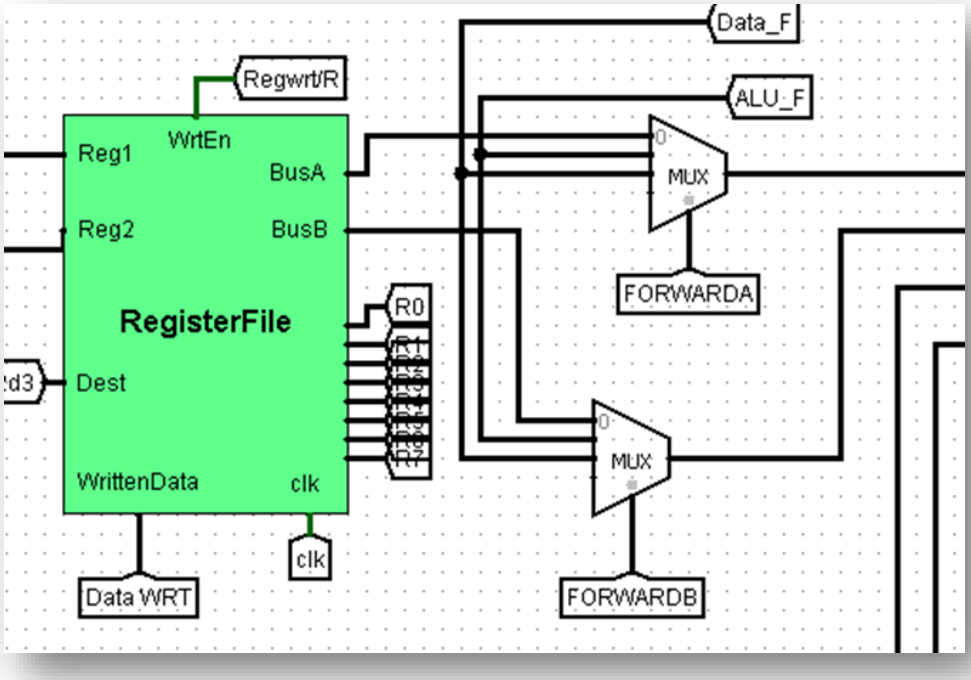
It selects the correct ALU inputs for the EX stage :

- *If there is no hazard, the ALU's operands will come from the register file*
- *If there is a hazard, the operands will come from either the EX/MEM or*

MEM/WB pipeline registers instead. The ALU sources will be selected by two new multiplexers, with control signals named ForwardA and ForwardB .

Forward A is the selector of the multiplexer that decides which input will enter as 1st operand

Forward B is the selector of the multiplexer that decides which input will enter as 2nd operand.



Case1(from previous)

- *if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)and (EX/MEM.RegisterRd = ID/EX.RegisterRs))*
  ForwardA=01 (Forward from EX/MEM pipe stage)

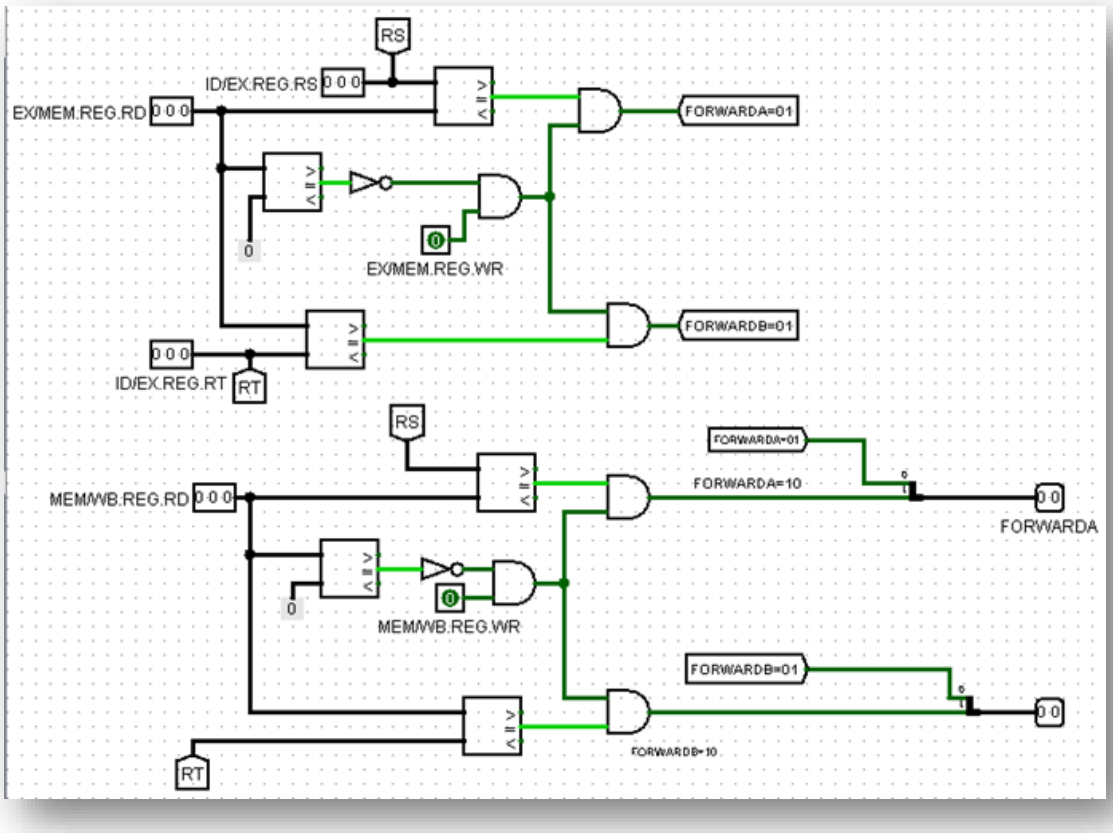
- *if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)and (EX/MEM.RegisterRd = ID/EX.RegisterRt))*
  ForwardB = 01 (Forward from EX/MEM pipe stage)

Case2(from previous previous

- *if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)and (MEM/WB.RegisterRd = ID/EX.RegisterRs))*
  ForwardA = 10 (Forward from MEM/WB pipe stage)

- *if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)and (MEM/WB.RegisterRd = ID/EX.RegisterRt))*
  ForwardB = 10 (Forward from MEM/WB pipe stage)

Forward unit



As we mentioned above if there's hazard I have two cases that from where I forward ..! so after detecting there's RAW hazard I need to know either it from previous or from previous previous stage

The mechanisim that forwarding unit work:

when EX/MEM.RegisterRd=ID/EX.RegisterRs

or EX/MEM.RegisterRd=ID/EX.RegisterRt i will forward from EX/MEM pipeline reg

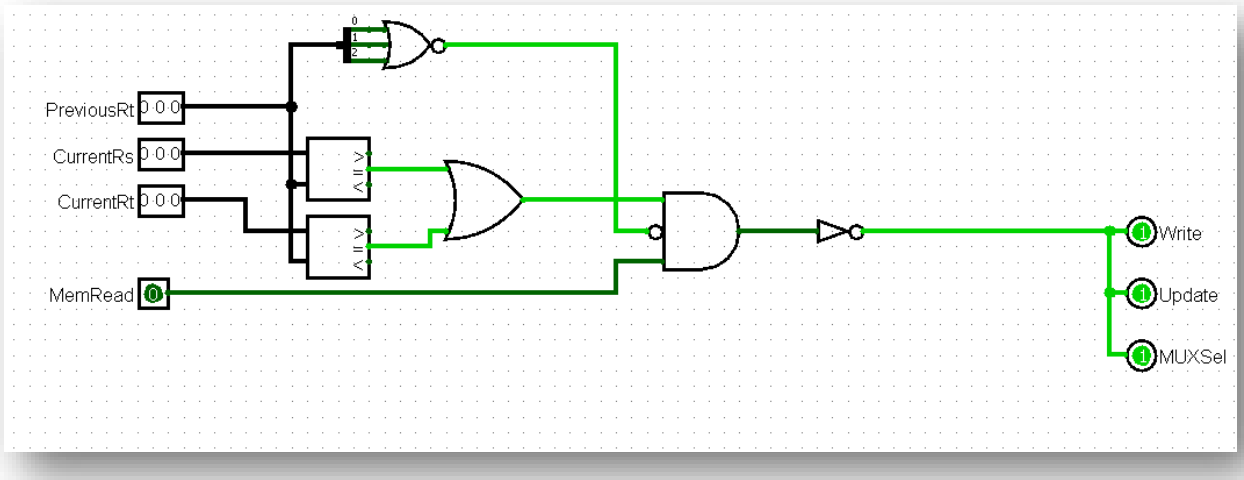
while when MEM/WB.RegisterRd=ID/EX.RegisterRs or MEM/WB.RegisterRd=ID/EX.RegisterRt i will forward from MEM/WB pipeline reg .

Unfortunately, not all data hazards can be forwarded

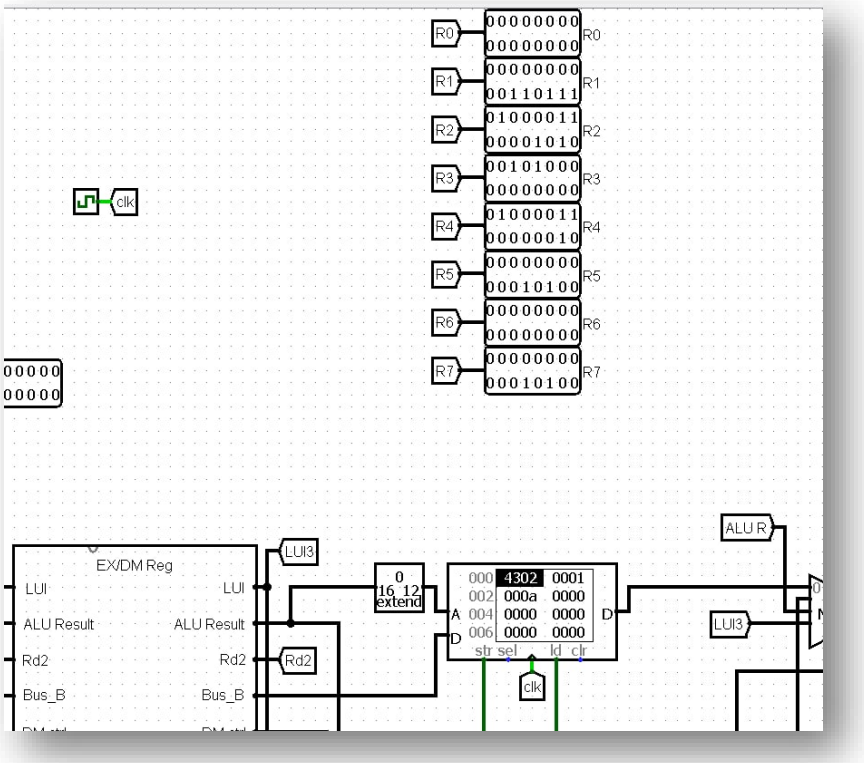
Load has a delay that cannot be eliminated by forwarding so I need to insert NOP into EX stage after a load instruction that delays the dependent instruction after load by one cycle

Condition for stalling the pipeline:

```
if((EX.MemRd== 1) // Detect Load in EX stage and
(ForwardA==1 or ForwardB==1)) Stall // RAWHazard
```



TEST CODE



Team work:

Student	*Departement*	*Work done -Single Cycle-*	*Work done -Pipeline-*
Mohammad Mustafa	**Computers and systems**	**-ISA Compiler -Instruction and Data memory -Main Control**	***-Control Hazard Detection Fully. - Testing***

		unit **-Data Path** **-Testing**	
Fatma Mohamed	**Computers and systems**	**-ALU** **-ALU control unit** **-Data Path** **-Testing**	***-DATAPATH*** ***-Testing***
Aya Yasser	**Computers and systems**	**-ALU** **-ALU control unit** **-Data Path** **- Testing** **-Report –Single Cycle section.**	***-DATAPATH*** ***-Testing*** ***- Report – Pipeline section.***
Mai Omar	**Communications**	**-Register File** **-PC Counter circuit** **-Data Path**	***-Forwarding Unit &Stall detection.*** ***-Testing*** ***-Report –Forward Unit Part.***