جامعة الفيوم
Fayoum University

Computer Engineering

(Single cycle processor phase 1)

Supervisors:

Dr. Gihan Nagiub

TA. Jihad

Done by:

Mina Moawad Ibrahim (EECE)

Abanob Evraam (EECE)

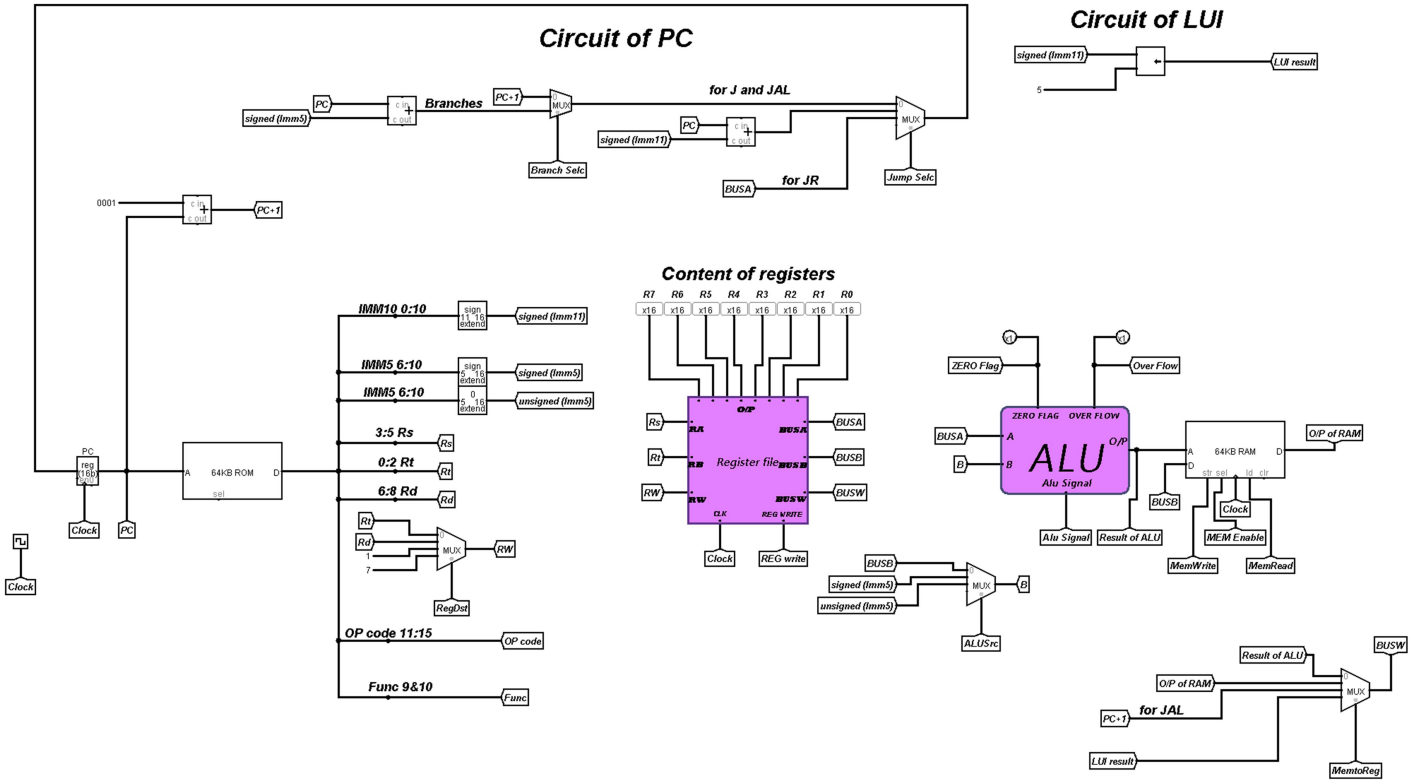
Mark Amgad (EECE)

Content:

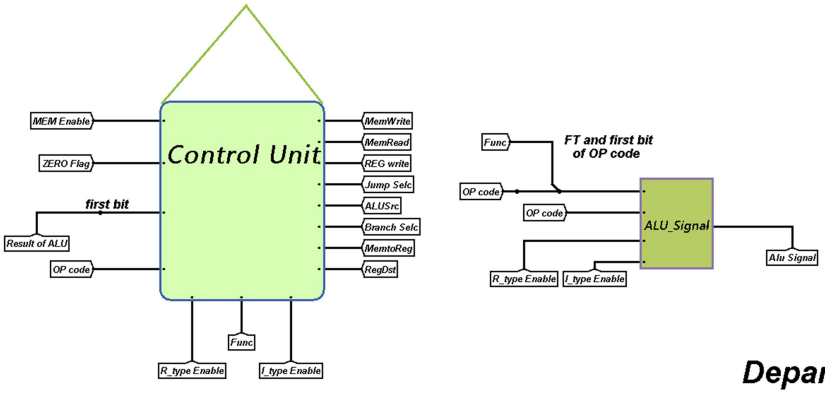
Introduction:

In this research we concerned to implement a 16-bit length MIPS single cycle processor which contains seven 16-bit general purpose registers (R1-R7). Also, R0 is hardwired to zero and cannot be written. There are only three instruction format which are R-type, I-type, and J-type. Each instruction is only 16 bits.

A single cycle processor is a processor that carries one instruction in a single cycle clock. An instruction is fetched from the memory, then decoded, and executed to store the result in a single cycle clock cycle. This is a simple model of MIPS processor in the terms of the hardware requirements. It has a poor data throughput (limited number of instructions).



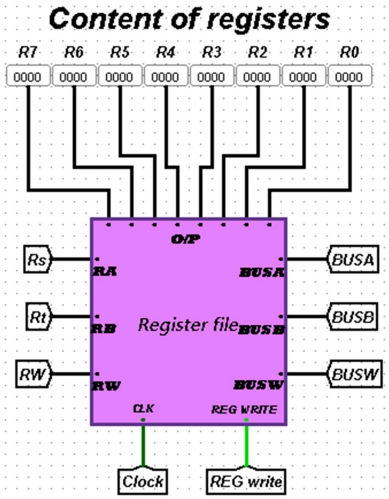
Circuit of Control Unit And Alu_Signal



BY : Abanob Evram
Mina Moawad
Mark Amgad

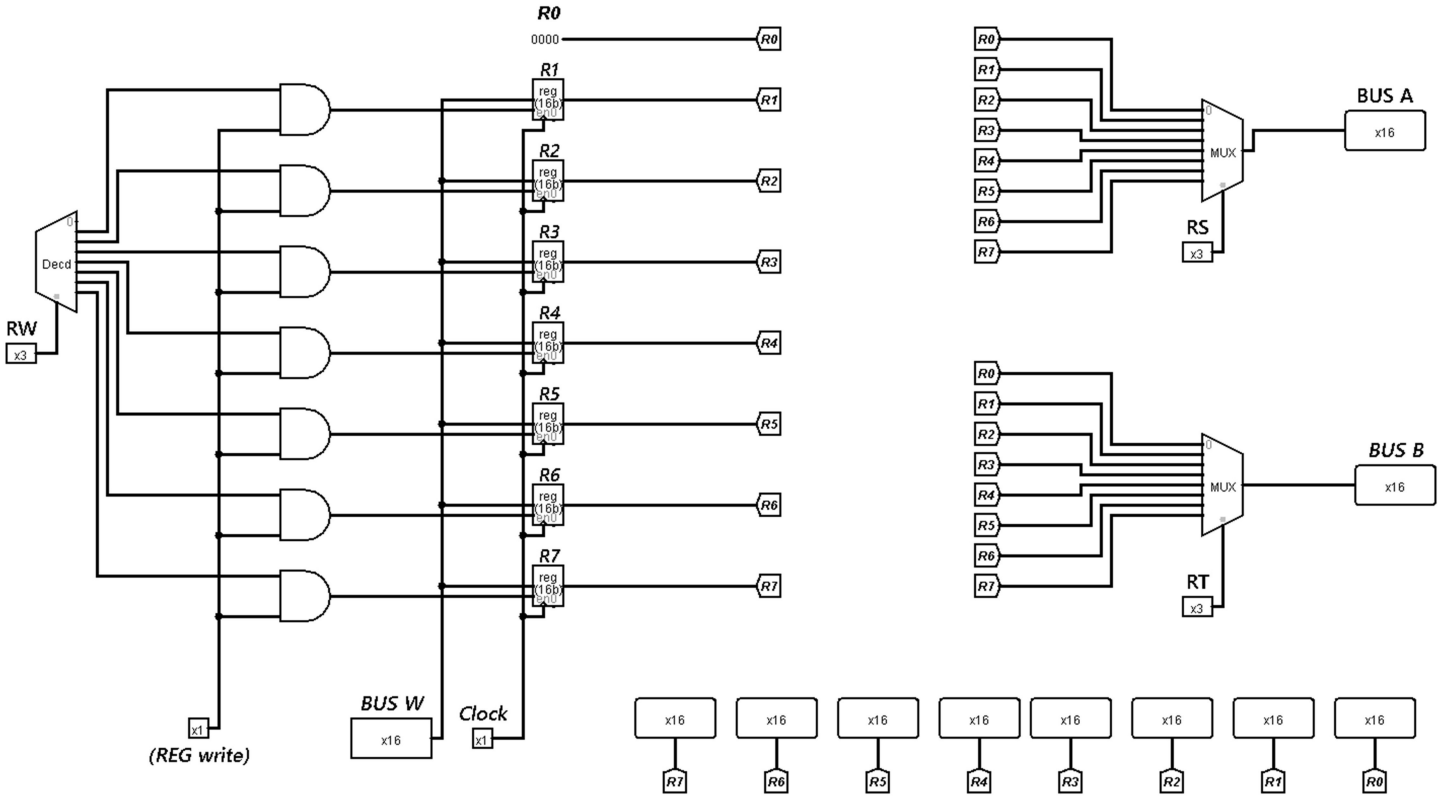
Department of electronics and communication

Register file:



Mainly, a register file is a set of 16-bit processor registers (8 registers) in the central processing unit (CPU). It contains seven 16-bit registers (R1-R7) with two read and one write port. In addition to the R0 register which is hardwired to zero. These registers are made of D flip-flops.

Our implementation relies on one decoder to choose in which register we prefer to write. Also, we use 7 AND gates for input enabled for all registers. Moreover, two multiplexors of 3 selectors to decide which register its data will be exhibited on the BUS A. Since BUS A represents the data in the register RS and BUS B represents the data in RT.



➢ALU:

It is a 16-bit arithmetic and logical unit used to perform all the required operations mentioned. We have about 16 different operations which can be chosen through the ALU MUX (ALU signal). In our ALU, there are two inputs (A), (B), and a MUX with 4 bits selector (ALU signal) in order to choose which operation is required to be performed.

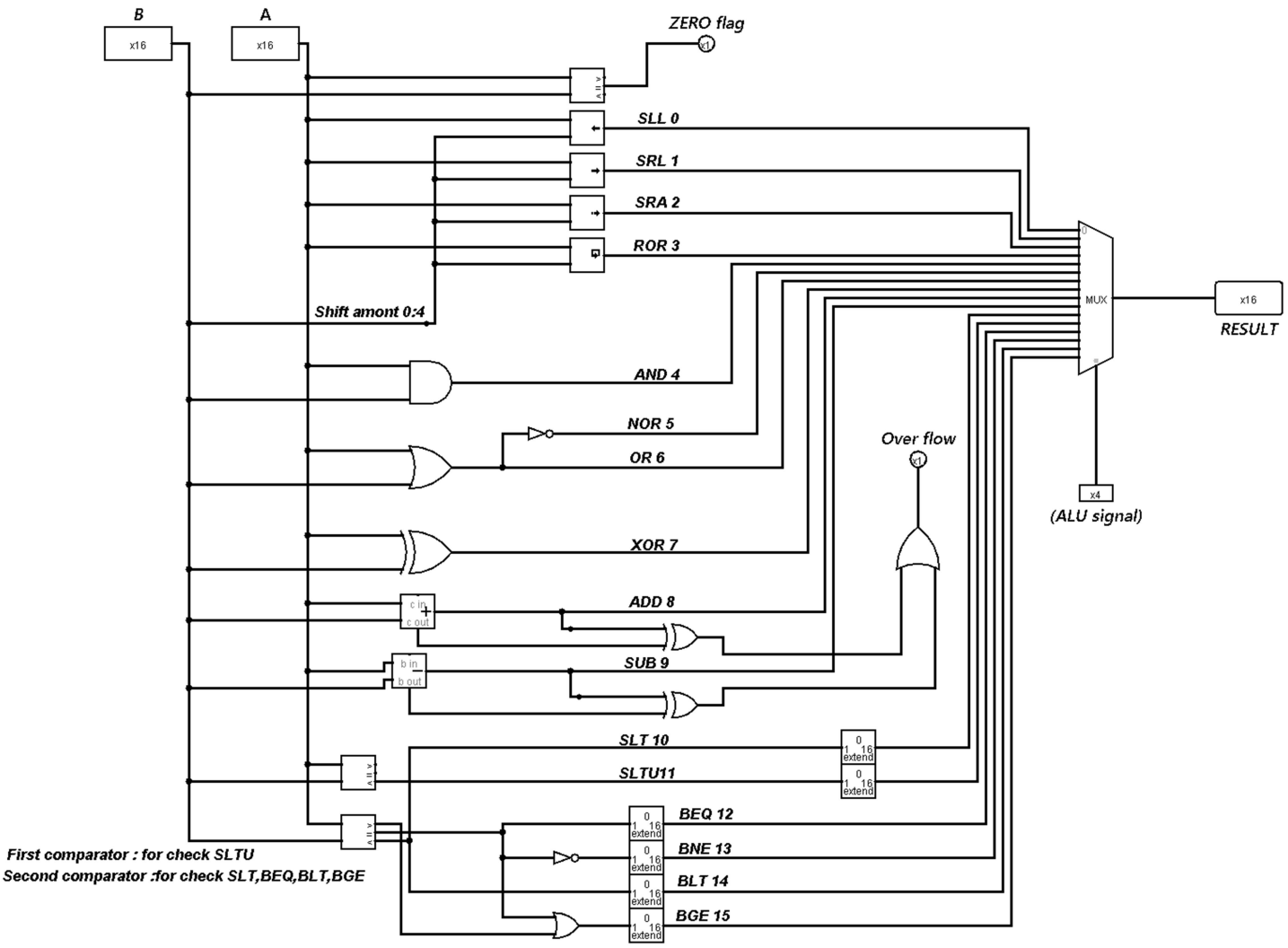
We used a comparator to check the zero flag. According to the shifting functions, input B is used as the shift amount so, we used a splitter to control the number of bits which will be considered as the shift amount. We also checked the over flow in addition and subtraction operations by Xoring the most significant bit of the output with the carry out. If the most significant bit of the output and the carry out are the same so there is no overflow, if not, so the overflow signal is activated.

Furthermore, we used two other comparators; the first one is an unsigned comparator to check the set less than unsigned (SLTU). The second one is to check SLT and all branches. The result of the branches is one bit so we used an extended to extend its value.

The operations are:

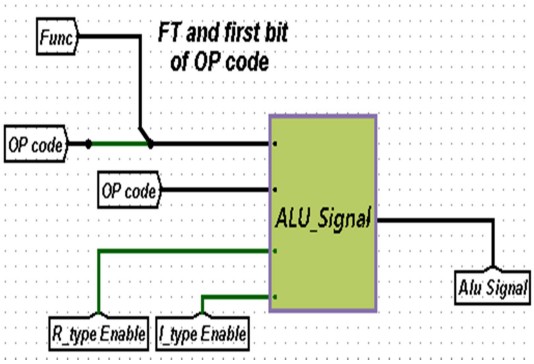
0	SLL	8	ADD
1	SLR	9	SUB
2	SRA	10	SLT
3	ROR	11	SLTU
4	AND	12	BEQ
5	NOR	13	BNE
6	OR	14	BLT
7	XOR	15	BGE

The following figure shows the implementation of the operations:

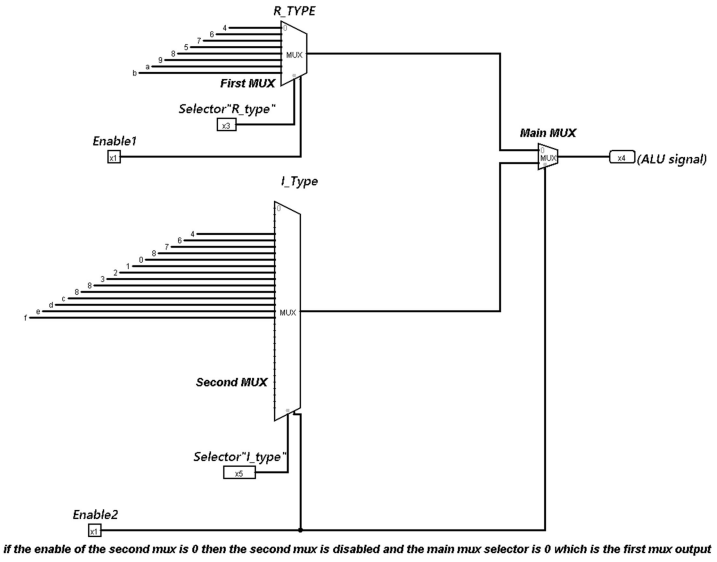


➢ALU Control:

The ALU Control unit receives two input signals which are the ALU-OP from the main control unit and 2-bit function (9&10) from the output of the instruction memory. These two input signals identify which operation will be performed and determine whether the instruction is R-type or I-type. Since the OP code of some R-type instruction is the same so the 2 bit function is necessary to clarify which function is desired. While in I-type, all the operations have different OP code. Thus; we used two MUXs to know which operation will be activated and both MUXs have an enable signal to activate one and deactivate the other.



The role of the Main MUX in the figure is to differentiate which MUX will be used. If the enable 1 signal is 0 and enable 2 signal is 1 then the Main Mux selects the I-type multiplexer. Finally the result is represented in 4 bits on the ALU signal which will be used in the ALU to identify specifically which operation will be performed.

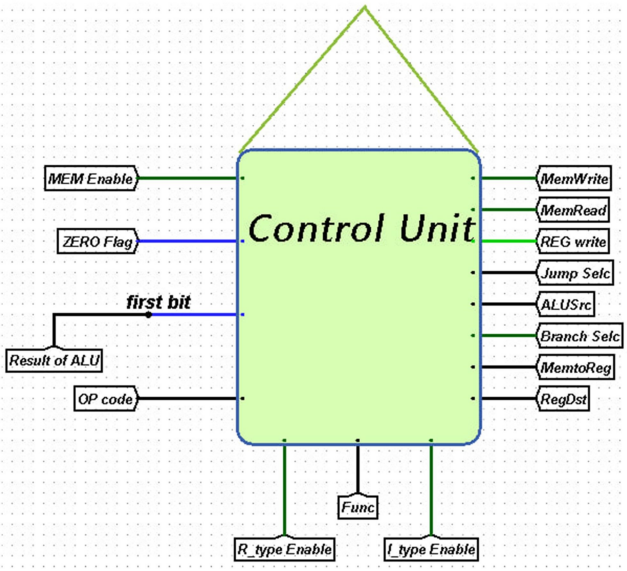


The following shows the truth table of the ALU signal:

Operation	O/P	Operation	O/P	Operation	O/P
AND	4	Andl	4	SW	8
OR	6	ORI	6	BEQ	c
XOR	7	XORI	7	BNE	d
Nor	5	Addl	8	BLT	e
Add	8	SLL	0	BGE	f
Sub	9	SRL	1	LUI	x
SLT	a	SRA	2	J	x
SLTU	b	ROR	3	JAL	x
JR	x	LW	8		

Control Unit:

The control unit is meant to be the brain of the whole processor as it controls every single sub-device in the processor. Each device in the processor has signals these signals are defined in the control unit. This control unit contains only one input which is the OP-Code. Actually, we used a decoder with the OP-code signal (5 bits) which identifies each operation according to its op-code. The first to outputs are ored together and passed to the R-type because the R-format is identified by OP Code 0 or 1. Op-code 2 is R-format also but a special case since it is a jump instruction.



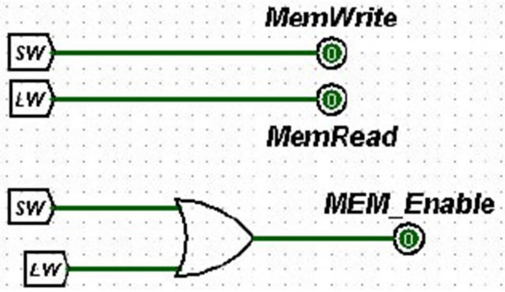
Control unit signal equations:

1-Memory (RAM) signals:

Memory write = SW

Memory read = LW

Memory Enable = SW + LW



2- RegDst:

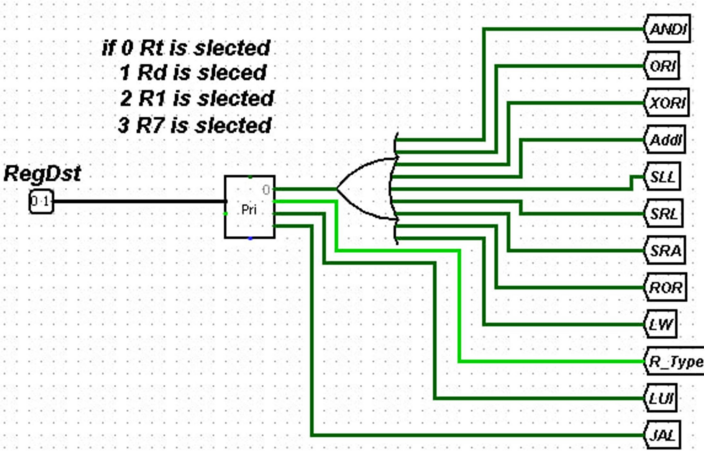
Signal which is used as a selector to select which Register will be accessed in the register file (destination register).

RegDst (00) = ANDI + ORI + XORI + ADDI + SLL + SRL + SRA + ROR + LW ➔ RT

RegDst (01) = R- TYPE ➔RD

RegDst (10) = LUI ➔ R1

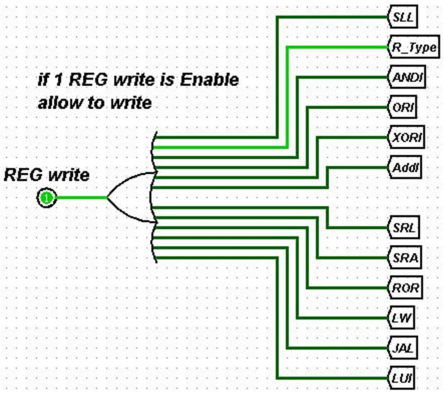
RegDst (11) = JAL ➔ R7



3-RegWrite:

A signal enables writing in the register file if it is 1.

RegWrite = ANDI + ORI + XORI + ADDI + SLL + SRL + SRA + ROR + LW + JAL + LUI + R_TYPE.



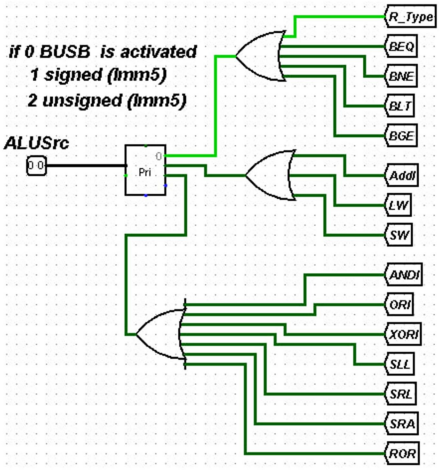
4-ALUSrc:

A signal enables us to choose to between BUSB, signed Imm5, and unsigned Imm5.

ALUSrc (00) = R-type + BEQ + BNE + BLT + BGE

ALUSrc (01) = ADDI + LW + SW

ALUSrc (10) = ANDI + ORI + XORI + SLL + SRL + SRA + ROR



5-Memory to register:

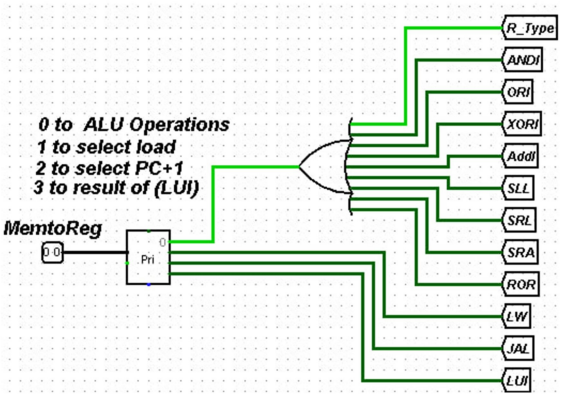
A signal used to select what will return to BUSW (ALU operation or value from memory or JAL or LUI)

MemtoReg (00) = ANDI + ORI + XORI + ADDI + SLL + SRL + SRA+ ROR + R_TYE

MemtoReg (01) = LW

MemtoReg (10) = JAL

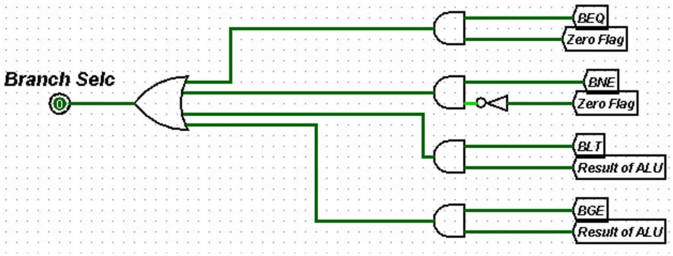
MemtoReg (11) = LUI



6-Branch selector:

This signal is to branch to a certain label with a specific address.

Branch selc = (BEQ * ZERO FLAG) + (BNE * $\overline{\text{ZERO FLAG}}$) + (BLT * ALU RESULT) + (BGE* ALUR ESULT).



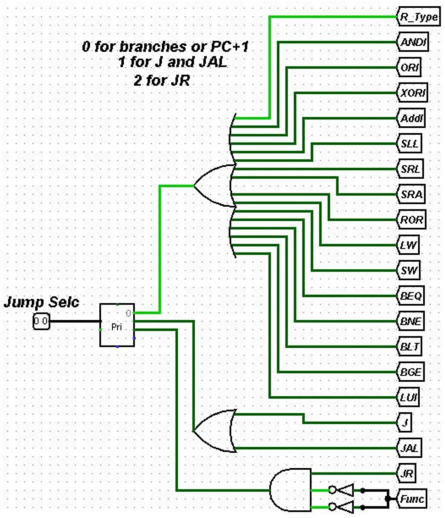
7-JUMP SELECTOR:

A signal used to select whether you branch or jump.

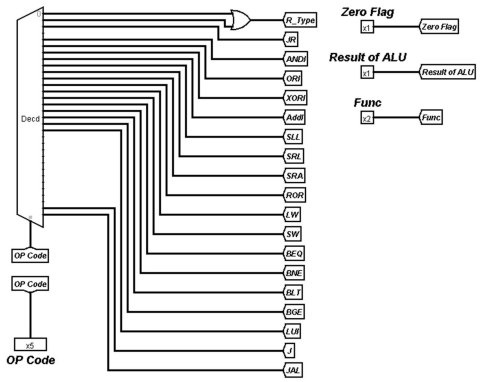
Jump selc (00) = ANDI + ORI + XORI + ADDI + SLL + SRL + SRA+ ROR + R_TYE + BEQ + BNE + BLT + BGE + LUI + LW + SW

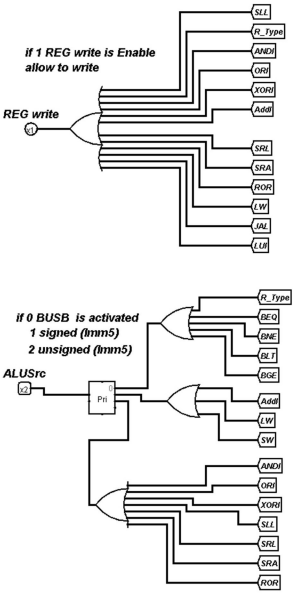
Jump selc (01) = J + JAL

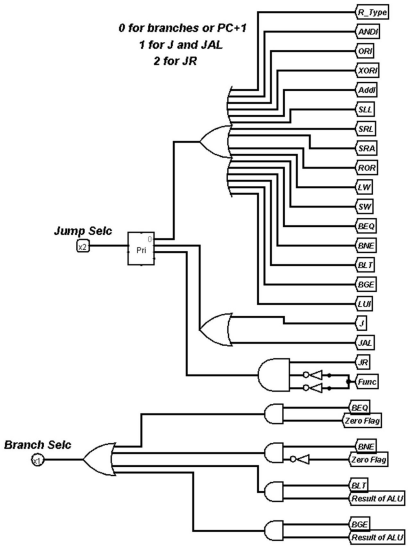
Jump selc (10) = JR * $\overline{\text{FUNC}}$

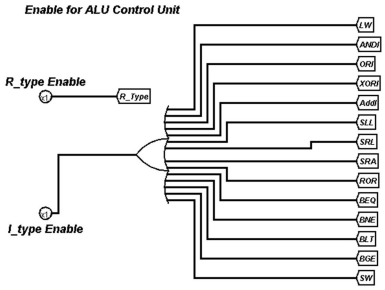


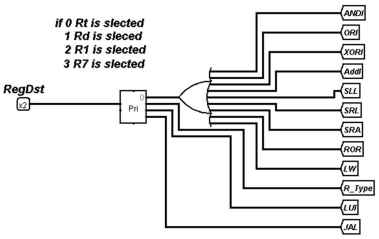
❖ *The whole design of the control unit:*

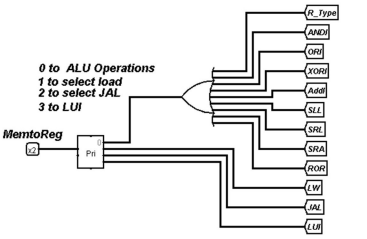












Each design in the in the processor is illustrated using these signals in the control unit. So, we used (AND) and (OR) gates to implement these signals. We also tend to use some encoders to get the signals needed to be passed to other devices to resume its processing. All of these signals own a truth table which simply shows how and when it works.

The following spreadsheet shows the truth table of the signals in the control unit:

		RegDst	REG write	ALUSrc	MemWrite	MemRead	MEM Enable	MemtoReg	Branch Selc	Jump Selc
R_TYPE	AND	1	1	0	0	0	0	0	0	0
	OR	1	1	0	0	0	0	0	0	0
	XOR	1	1	0	0	0	0	0	0	0
	Nor	1	1	0	0	0	0	0	0	0
	Add	1	1	0	0	0	0	0	0	0
	Sub	1	1	0	0	0	0	0	0	0
	SLT	1	1	0	0	0	0	0	0	0
	SLTU	1	1	0	0	0	0	0	0	0
	JR	x	0	x	0	0	0	x	x	2
I_TYPE	AndI	0	1	2	0	0	0	0	0	0
	ORI	0	1	2	0	0	0	0	0	0
	XORI	0	1	2	0	0	0	0	0	0
	AddI	0	1	1	0	0	0	0	0	0
	SLL	0	1	2	0	0	0	0	0	0
	SRL	0	1	2	0	0	0	0	0	0
	SRA	0	1	2	0	0	0	0	0	0
	ROR	0	1	2	0	0	0	0	0	0
	LW	0	1	1	0	1	1	1	0	0
	SW	x	0	1	1	0	1	x	0	0
	BEQ	x	0	0	0	0	0	x	1	0
	BNE	x	0	0	0	0	0	x	1	0
	BLT	x	0	0	0	0	0	x	1	0
	BGE	x	0	0	0	0	0	x	1	0
J_TYPE	LUI	2	1	x	0	0	0	3	0	0
	J	x	0	x	0	0	0	x	x	1
	JAL	3	1	x	0	0	0	2	x	1

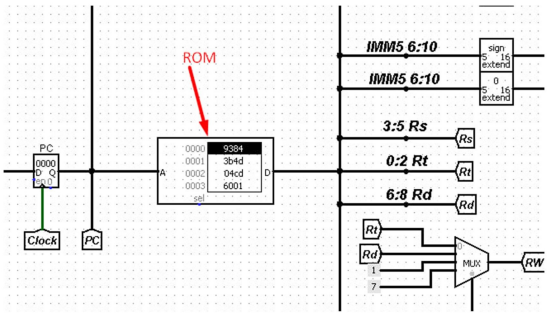
➢ Program Counter (Pc):

The program counter is a register in the computer processor which contains the location address of the instruction being executed. After the current instruction is fetched, the program counter is incremented by 1 since the program counter contains a word address. When the PC is incremented by 1, it points to the next instruction in the memory.

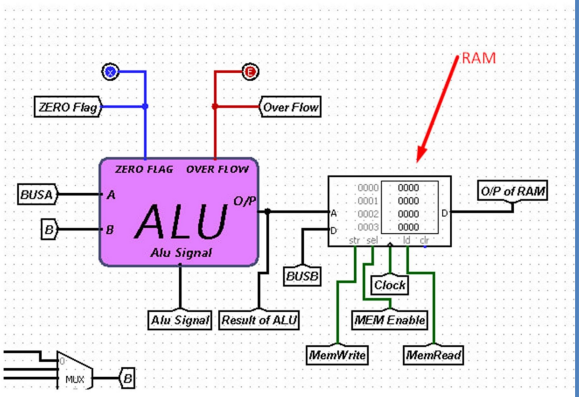
➢ Data path:

The data path is the lines (wires) that connect the components of the processor with each other and with the different signals. It contains the previous mentioned components. It also contains an instruction memory (ROM), data memory (RAM), and a program counter register. In addition, it has decoders, multiplexers, extenders, clock, and adders.

THE INSTRUCTION MEMORY (ROM): it is a 16-bit data width with 16-bit address width. The code or the instructions is loaded in the instruction memory. After that, the instruction is then read and passed to each device in the processor and the program is being executed.



THE DATA MEMORY (RAM): its data and address is the same as the ROM 16-bit width. It is used to read and write inside it based on the instruction. After the program is executed, if we store data in the memory it will appear in the data memory (RAM) block after the program is terminated.



THE CLOCK: the program starts executing at each positive rising edge of the clock.

➢ Testing:

The following code is testing most of the operations and functions which is implemented in the circuit of the MIPS single cycle processor.

The following table shows the code instruction, code in binary, conversion of code to hex, and the expected value for each instruction.

Instruction number	instruction	code in binary	code in hex	expected value
0000	Lui 0x384	1001001110000100	9384	R1=0x7080
0001	Addi $5, $1,13	0011101101001101	3B4D	R5=0x708d
0002	Xor $3, $1, $5	0000010011001101	04CD	R3=0x000d
0003	Lw $1, 0($0)	0110000000000001	6001	R1=0x0001
0004	Lw $2, 1($0)	0110000001000010	6042	R2=0x0001
0005	Lw $3, 2($0)	0110000010000011	6083	R3=0x000a
0006	Addi $4, $4, 10	0011101010100100	3AA4	R4=0x000a
0007	Sub $4, $4, $4	0000101100100100	0B24	R4=0x0000
0008	L2: Add $4, $2, $4	0000100100010100	0914	R4=0x0001 / R4 finally= 0x0037
0009	Slt $6, $2, $3	0000110110010011	0D93	R6=0x0001/R6 finally =0x0000
000a	Beq $6, $0, L1	0111000011110000	70F0	Branch after ten iterations
000b	Add $2, $1, $2	0000100010001010	088A	R2=0x0002/R2 finally =0x000a
000c	Beq $0, $0, L2	0111011100000000	7700	Branch to L2
000d	L1: Sw $4, 0($0)	0110100000000100	6804	Location 0 = 0x0037
000e	Jal Func	1111100000000100	F804	R7=0x000f & jump to func
000f	Sll $3, $2, 6	0100000110010011	4193	R3=0xc280
0010	ROR $6, $3, 3	0101100011011110	58DE	R6=0x1850
0011	beq $0,$0,-1	0111000000000000	7000	program is over, keep looping back to here
0012	Func: or $5, $2, $3	0000001101010011	0353	R5=0x000a
0013	Lw $1, 0($0)	0110000000000001	6001	R1=0x0037
0014	Lw $2, 5($1)	0110000101001010	614A	R2=0x430a
0015	Lw $3 ,6($1)	0110000110001011	618B	R3=0x7342
0016	And $4, $2, $3	0000000100010011	0113	R4=0x4302
0017	Sw $4, 0($0)	0110100000000100	6804	Location 0 = 0x4302
0018	Jr $7	0001000000111000	1038	PC =0x000f to SLL

➢ Following the code instructions:

First of all we save in the memory (RAM) some values in its addresses:

Memory starting from address 0 contains:

M[0]= 0001

M[1]=0001

M[2]=000a

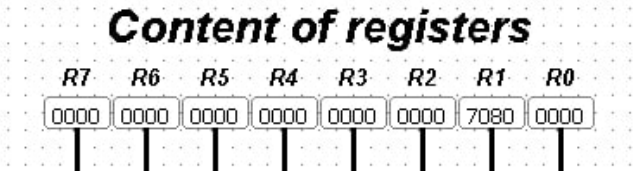
M[60]=430a

M[61]=7342

1) **Lui 0x384 :**

The Lui shifts the immediate value by 5 and save the result R1.

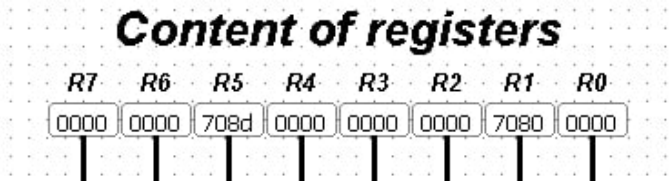
The expected value is R1 = 0x780



2) **Addi $5, $1, 13:**

We add 13 to R1 and save the result of the addition in R5.

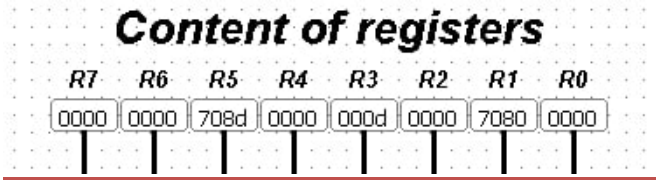
The expected value is R5 = 0x708D.



3) **XOR $3, $1, $5:**

Xoring R5 with R1 and save the result in R3.

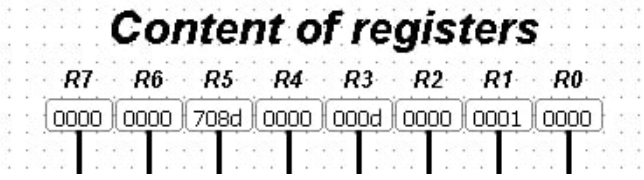
The expected value is R3 = 0x000d.



4) **Lw $1, 0($0):**

We load the value of the memory first location in R1.

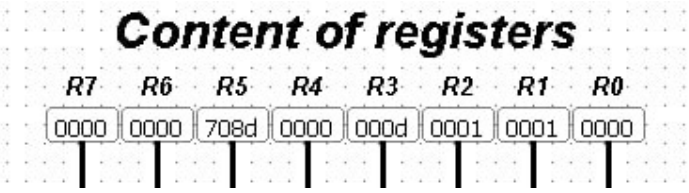
The expected value is R1 = 0x0001.



5) **Lw $2, 1($0):**

We load the value of the memory second location in R2.

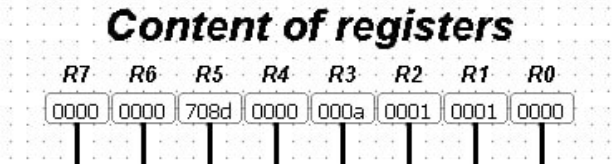
The expected value is R2 = 0x0001.



6) **Lw $3, 2($0):**

We load the value of the memory third location in R3.

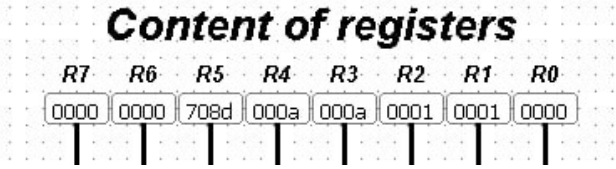
The expected value is R3 = 0x000a.



7) **Addi $4, $4, 10**:

We add 10 to the value saved in R4 and the whole result is saved again in R4.

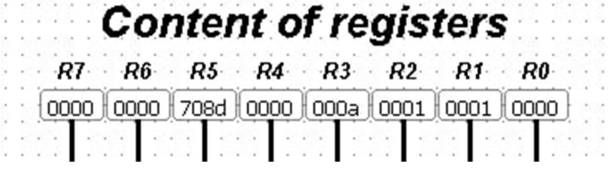
The expected value is R4 = 0x000a.



8) **sub $4, $4, $4:**

We subtract R4 from R4 and set the result of the subtraction in R4.

The expected value is R4 = 0x0000

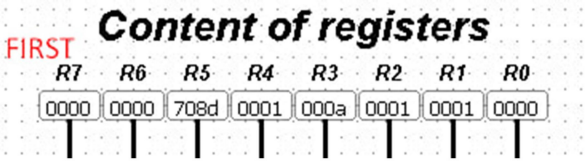


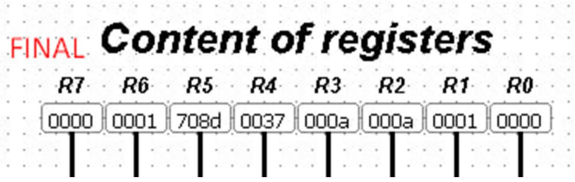
9) L2: ADD $4, $2, $4:

From here L2 starts with its first instruction. The instruction is to add R4 and R2 since the result is set to R4.

The expected value of the first iteration is R4 = 0x0001

The expected value of the last iteration is R4 = 0x0037



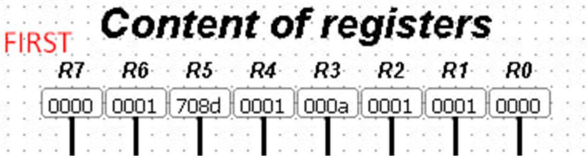


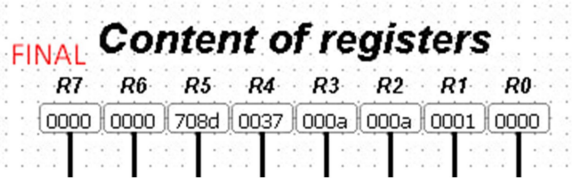
10) SLT $6, $2, $3:

Set less than instruction where we compare two register R2, R3 if R2 < R3 then the value set to R6 is 1 else the value of R6 is 0.

The expected value of the first iteration is R6 = 0x0001

This instruction will be satisfied till the last iteration where the two registers are the same then the expected value of the last iteration is R6 = 0x0000.

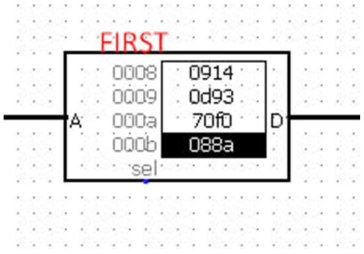


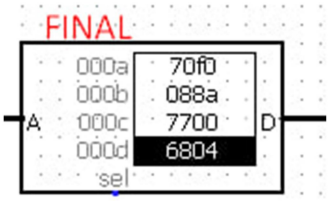


11) BEQ $6, $0, L1:

In this instruction we compare the two mentioned register if they match then we branch to L1. But if they don't match then we just move to the next instruction (pc+1).

The expected value is that the program counter will branch to L1 after 10 iterations where the branch comparator is satisfied.



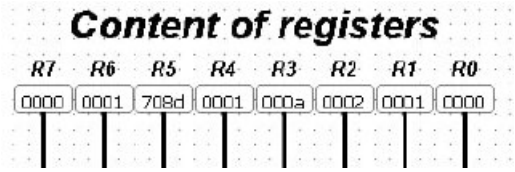


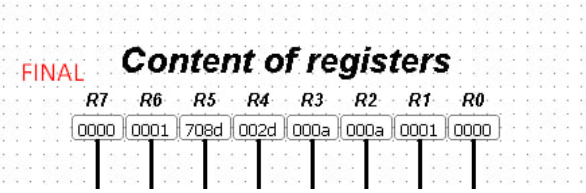
12) ADD $2, $1, $2:

Add instruction where we add R1, R2 and we put their result in R2.

The expected value of the first iteration is R2 = 0x0002

The expected value of the final iteration is R2 = 0x000a





13) BEQ $0, $0, L2:

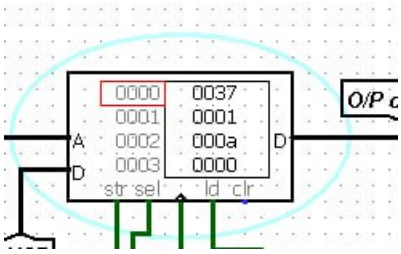
We compare between the two mentioned registers where it is matched the condition so it will branch to L2.

Then the program counter will branch to location 8 in the ROM.

14) SW $4, 0($0):

Store instruction is to save the value of R4 in the first location of the memory (RAM).

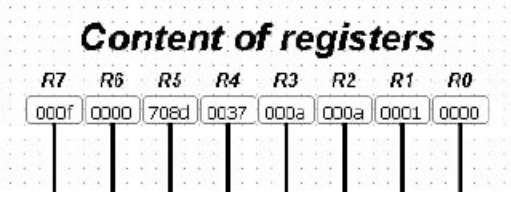
The expected output is location zero is equal to 0x0037.



15) JAL FUNC:

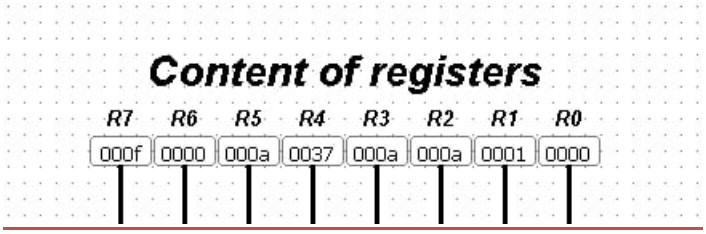
This operation is to jump to FUNC and save in R7 the value of pc+1.

The expected value is R7 = 0x000f.



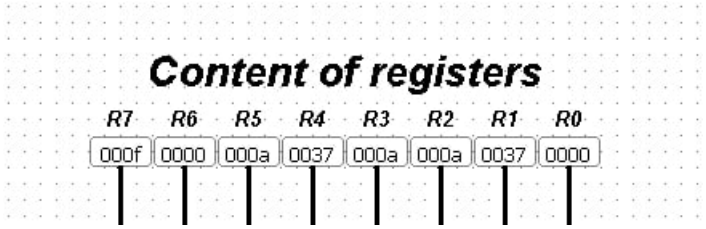
16) OR $5, $2, $3:

The expected value of the output is R5 = 0x000a.



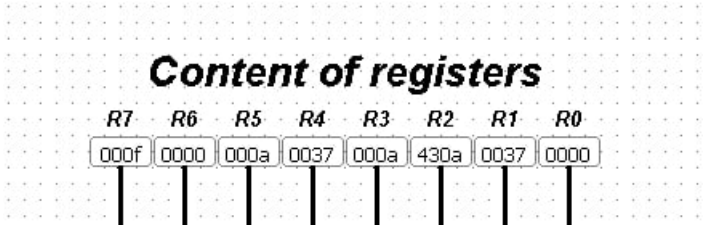
17) Lw $1, 0($0):

The expected value is R1 = 0x0037.



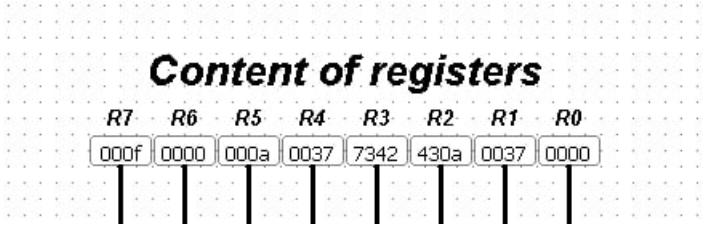
18) Lw $2, 5($0):

The expected output is R2 = 0x430a.



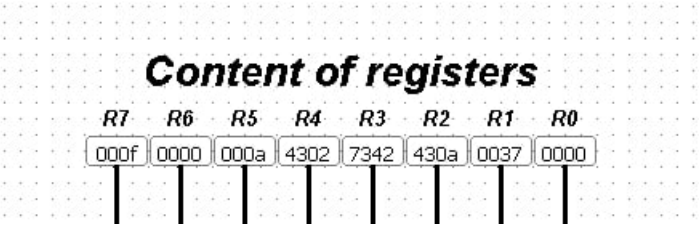
19) Lw $3, 6($0):

The expected output is R3 = 0x7342.



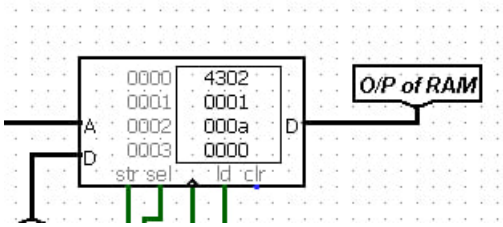
20) AND $4, $2, $3:

The output is R4 = 0x4302.



21) SW $4, 0($0):

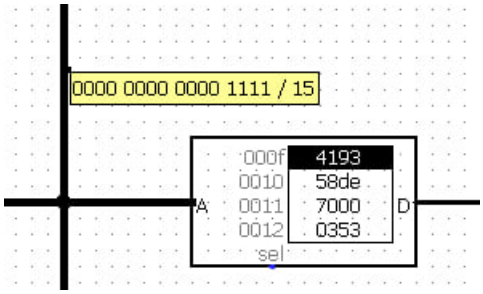
The expected value is that location zero = 0x4302.



22) JR R7:

We set the program counter to the value of R7.

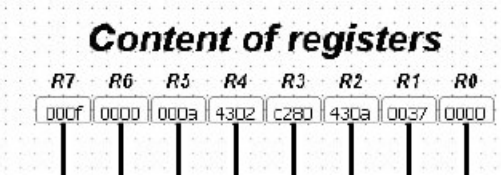
The expected output is pc = 0x000f.



23) SLL $3, $2,6

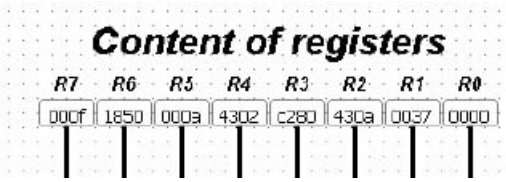
Here we shift the value placed in R2 by (2^6) and we save the result in R3.

The expected value is R3 = 0xc280.



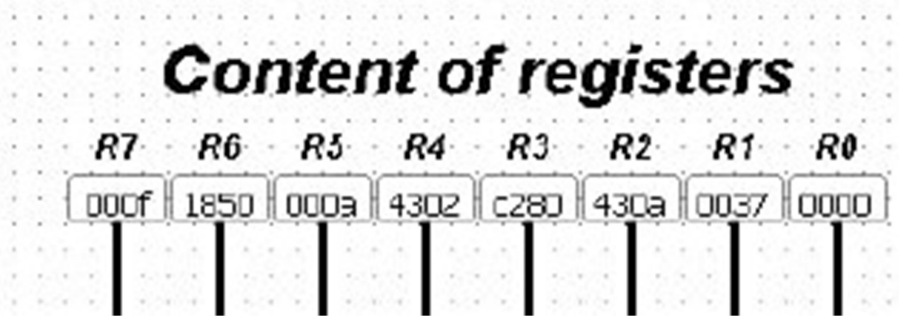
24) ROR $6, $3, 3:

The expected value R6 = 0x1850.



25) BEQ $0, $0,-1 :

Comparing the two register, they are the same so, the program stuck in an infinity loop. The program is over and it will keep looping here.



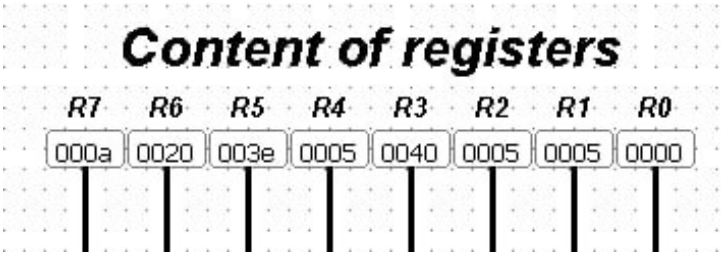
****

❖ Second code for testing:

The following code creates an array of five elements in the memory. In each element it saves the multiple of 2. After that a summation is done over the elements in each location in the memory. The summation is calculated and the result is saved in R5. Expected result R5 = 2+4+8+16+32= 62

Instruction number	instruction	code in binary	code in hex	expected value
0000	AddI $1,$0,5	0011100101000001	3941	R1=5
0001	XOR $2,$2,$2	0000010010010010	0492	R2=0
0002	AddI $3 ,$0 ,2	0011100010000011	3883	R3= 2
0003	Loop1 : SW $3,0($2)	0110100000010011	6813	Location 0 = 2
0004	SLL $3, $3,1	0100000001011011	405b	R3=2*2=4
0005	AddI $1,$1,-1	0011111111001001	3fc9	R1-- / finally R1=0
0006	AddI $2,$2,1	0011100001010010	3852	R2++
0007	BNE $1,$0,Loop1	0111111100001000	7f08	Branch after five iterations
0008	AddI $1,$0,5	0011100101000001	3941	R1=5
0009	JAL Sum	1111100000000011	F803	R7=PC+1 and jump to sum
000a	Add $5,$5,$0	0000100101101000	0968	R5 =0x003e=62
000b	Finish : BEQ $0,$0,Finish	0111000000000000	7000	program is over, keep looping back to here
000c	Sum : LW $6,0($4)	0110000000100110	6026	save the value from MEM to R6
000d	Add $5,$5,$6	0000100101101110	096e	R5+=R6
000e	AddI $4,$4,1	0011100001100100	3864	R4++
000f	BLT $4,$1,Sum	1000011101100001	8761	Not Branch after five iterations
0010	Jr $7	0001000000111000	1038	jump to instruction num 000a

The expected values of the whole registers are:



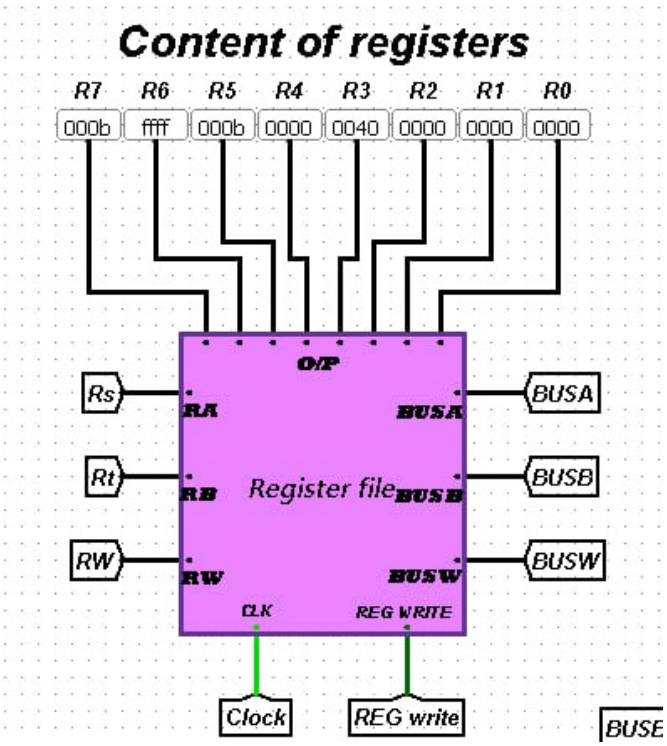
****

❖ *Third code for testing:*

The following code examines various operations in the MIPS processor. Mainly, it initiates some registers with certain values then a loop is implemented using BLT since the branch satisfies the conditions. The program terminates the branch when R2 is not less than R1 and then jumps to exit.

Instruction number	instruction	code in binary	code in hex	expected value
0000	addi $1, $1, 5	0011100101001001	3949	R1 =0X0005
0001	addi $2, $2, 0	0011100000010010	3812	R2=0X0000
0002	addi $3, $3, 2	0011100010011011	389B	R3=0X0002
0003	ori $5, $5, 11	0010101011101101	2AED	R5=0X000b
0004	sw $5, 0($3)	0110100000011101	681D	location 3 = 0x000b
0005	L1:addi $1, $1, -1	0011111111001001	3FC9	R1--/finally R1=0
0006	sll $3, $3, 1	0100000001011011	405B	R3=0x0004/ finally R3=0x0040
0007	lw $6 , 0($3)	0110000000011110	601E	R6=0x0000/finally R6=0x0000
0008	nor $6, $6, $6	0000011110110110	07B6	R6=0xffff/finally R6=0xffff
0009	blt $2, $1, L1	1000011100010001	8711	not branch after five iteratiors
000a	jal exit	1111100000000001	F801	R7=0x000b(R7)
000b	addi $0 , $0 , 0	0011100000000000	3800	INFINITY LOOP
000c	jr $7	0001000000111000	1038	

The expected output values of the whole registers are:



➢ Team work:

Mina➔ ALU signal, Control unit, documentation, main design, test code.

Abanob ➔ main design (data path), test code, documentation, control unit, ALU.

Mark ➔ register file, converting instruction to hex code, ALU, test code.

➢ Plan for completing subtasks of the project:

First (F2F) meeting ➔ we designed the ALU and the register file.

Second (F2F) meeting ➔ we work on the idea and the implementation of the ALU signal.

Third (online) meeting ➔ we were concerned about the Data path and the main design.

Fourth (F2F) meeting ➔ we designed the main circuit, all signals, and the program counter.

Fifth (online) meeting ➔ we designed the control unit.

Sixth (online) meeting ➔ testing the code and make sure all operations are working.

Seventh (F2F) meeting ➔ we worked on the test code and recorded the illustration of our project.

Eighth (online) meeting ➔ reviewing the whole project and finishing the report.

PIPELINE (PHASE 2):

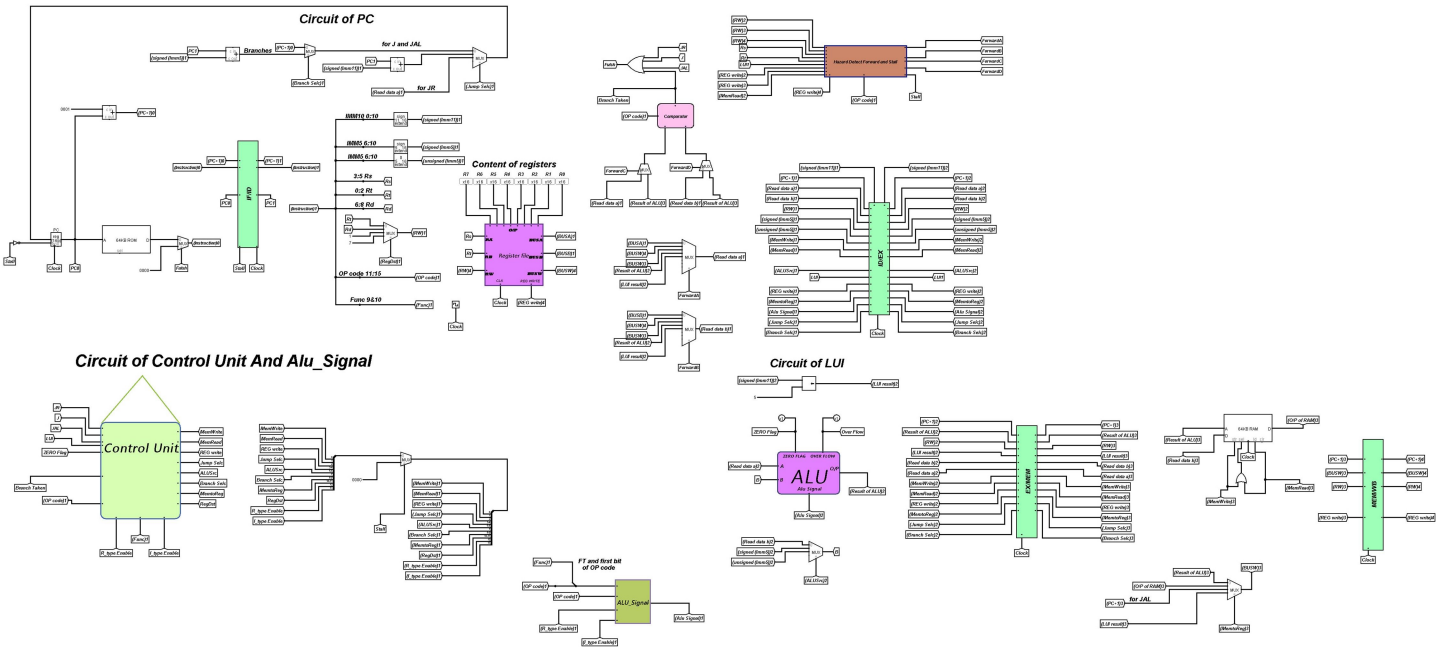
➢ Introduction:

The idea of pipelining is to allow multiple instructions to be executed at the same time. However; each instruction use different functional device in the data path. It mainly increases the performance of the processor as it reduces the execution time of the whole instructions (code).

All the main components are mostly the same, but we have some slight changes in the ALU device as we deleted the comparators for branches and we moved it to the comparator device. The check of branch is separated from the rest of operations.

We used four different pipeline register of different stages:

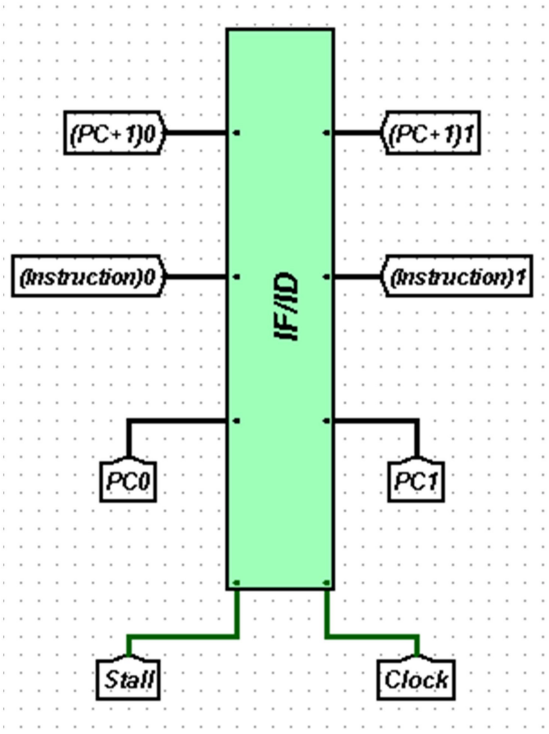
- IF/ID stage
- ID/EX stage
- EX/MEM stage
- MEM/WB stage

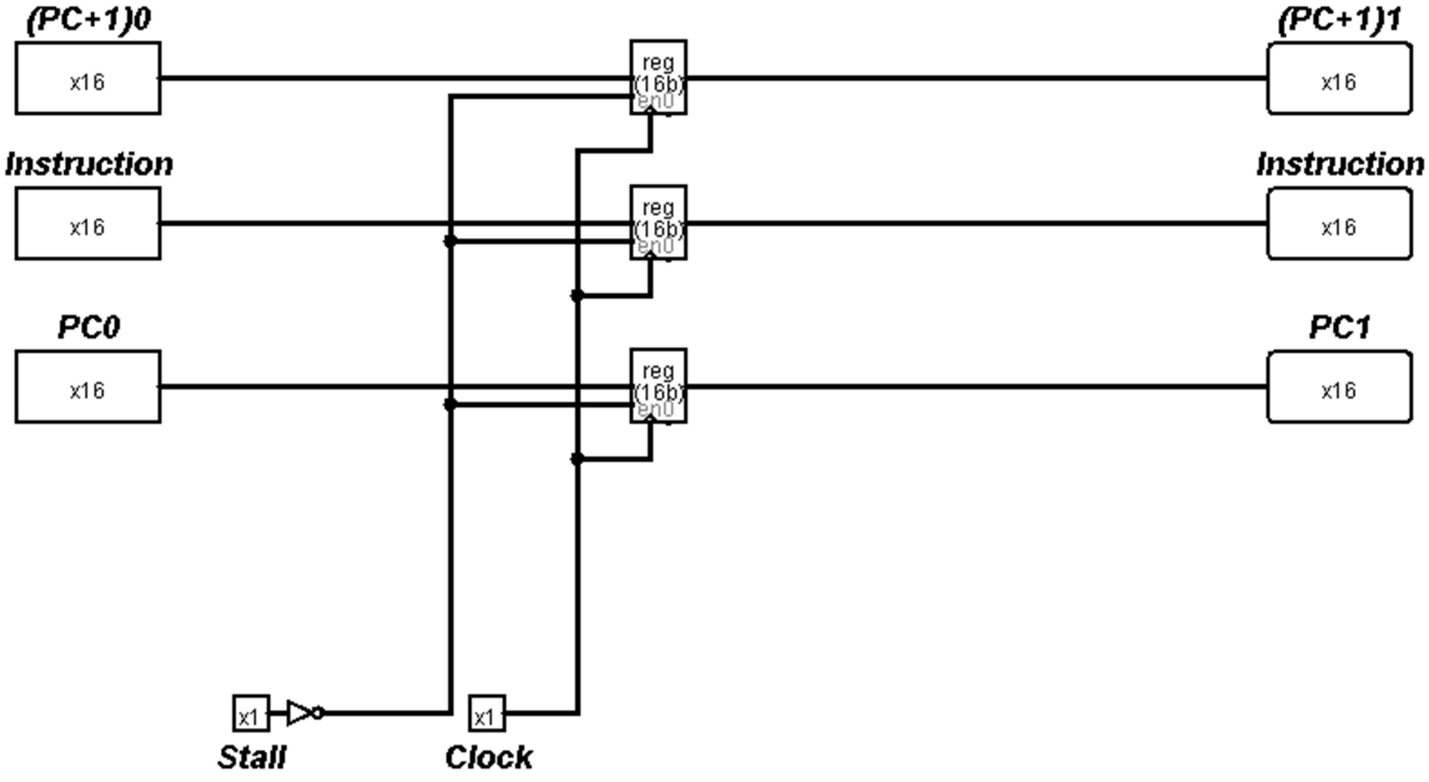


➢ Pipeline registers:

1) IF/ID stage:

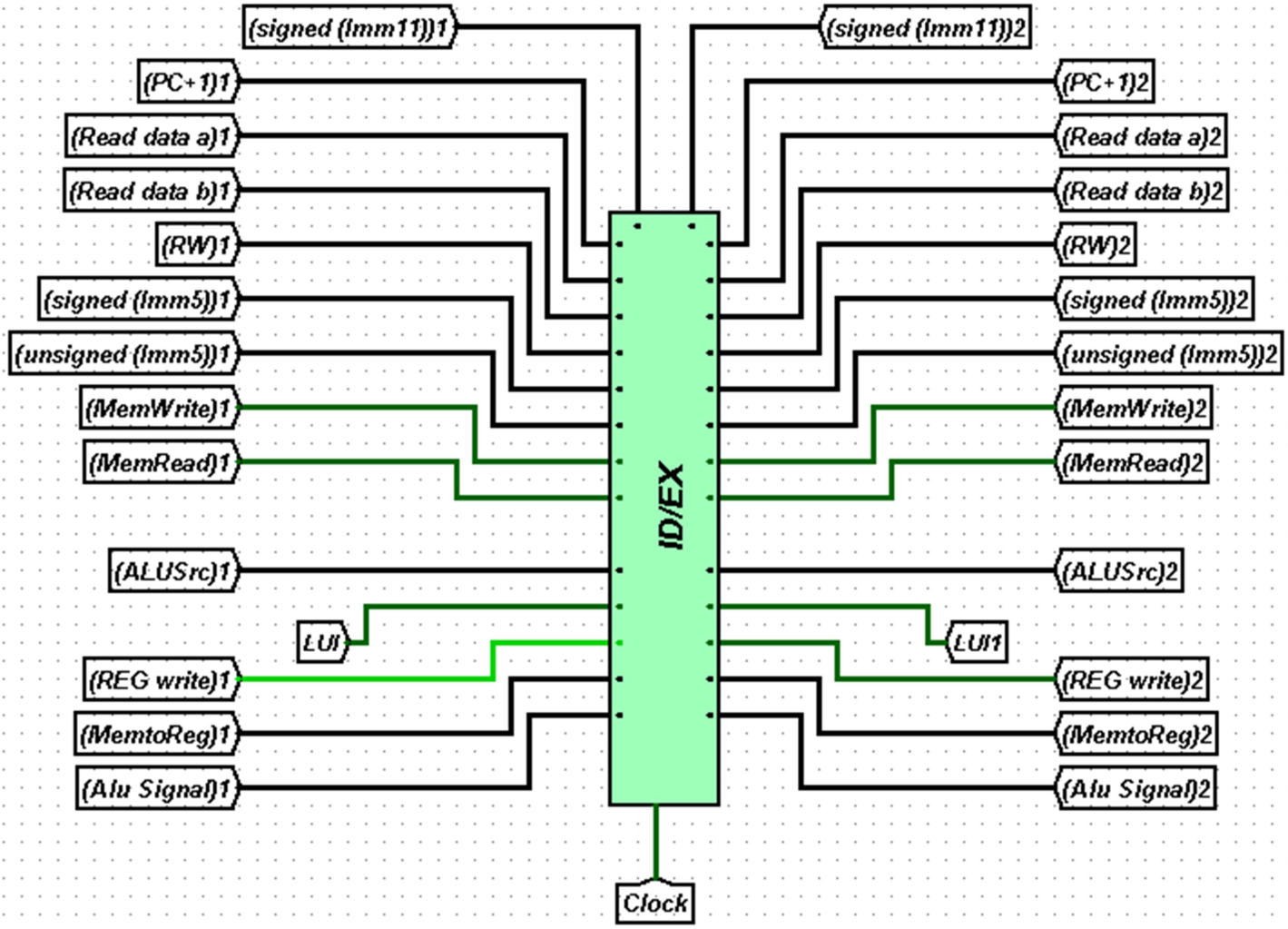
Instruction fetch and instruction decode is the first stage in pipeline registers in which the instruction is fetched form the instruction memory (ROM) and the bits of all the control signals are decoded. These instructions are saved in the IF/ID register. Also we have to save the current program counter (PC), and the next program counter (PC+1). Since, they will be used at a long term in the executing some of the operations. The stall signal is the enable of the IF/ID register.

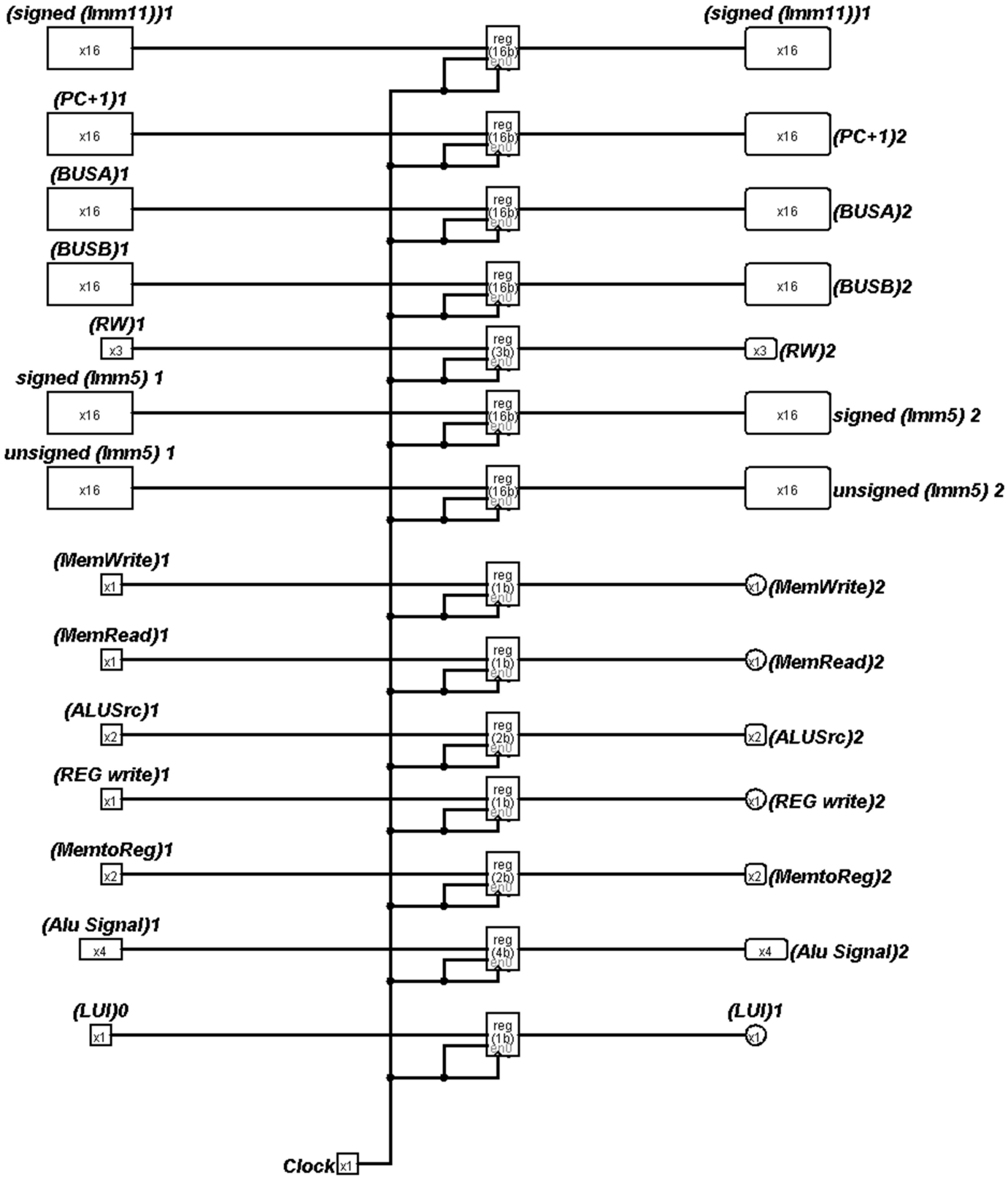




2) ID/EX stage:

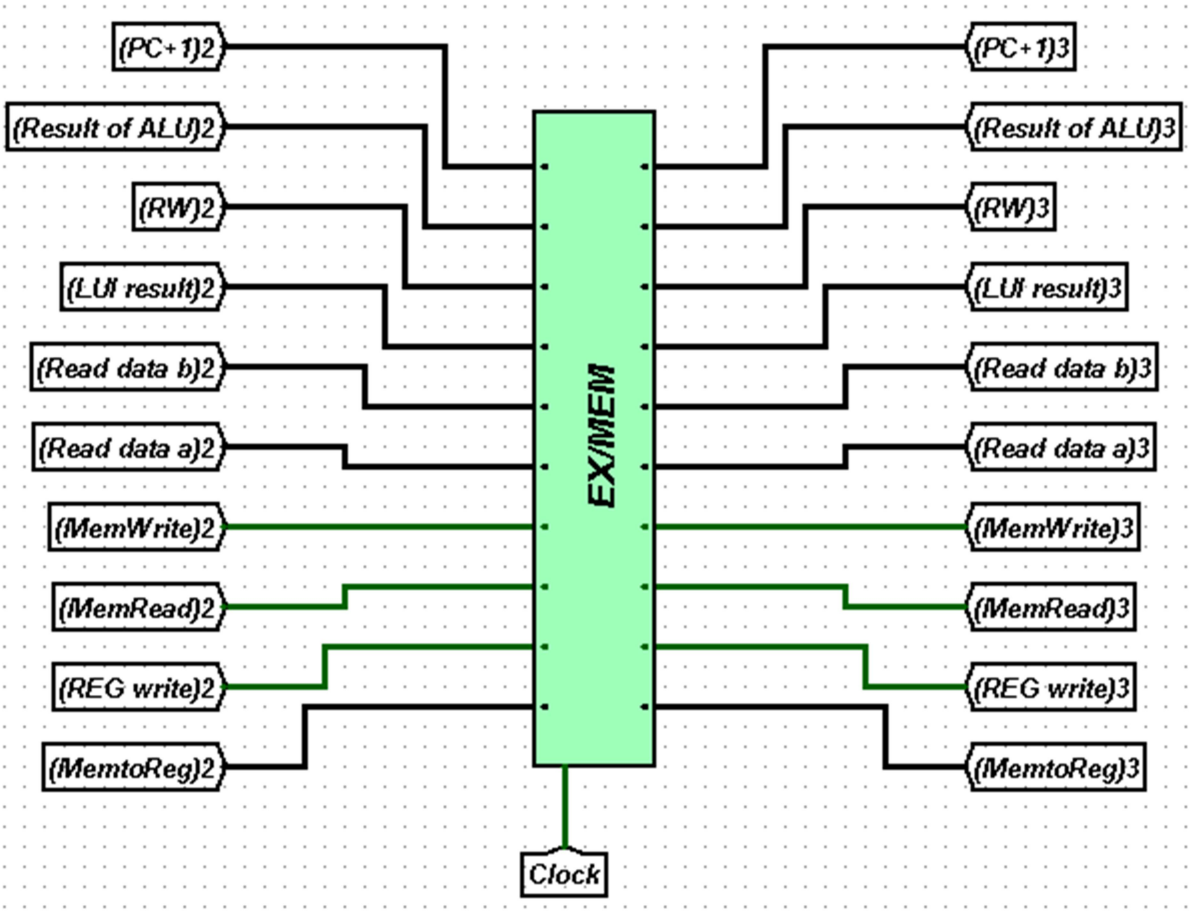
In this stage the instruction is executed, especially if this instruction is logical or arithmetic operations. The data that will be used in the ALU is saved in this register till it is executed in the next clock cycle. If this is a load or store instruction, the address is being executed. We are concerned to save all the signals and the required data that will be used at a long term in the execution of some operations because if we do not save it, the required data may be lost or changed on the next rising edge of the clock.



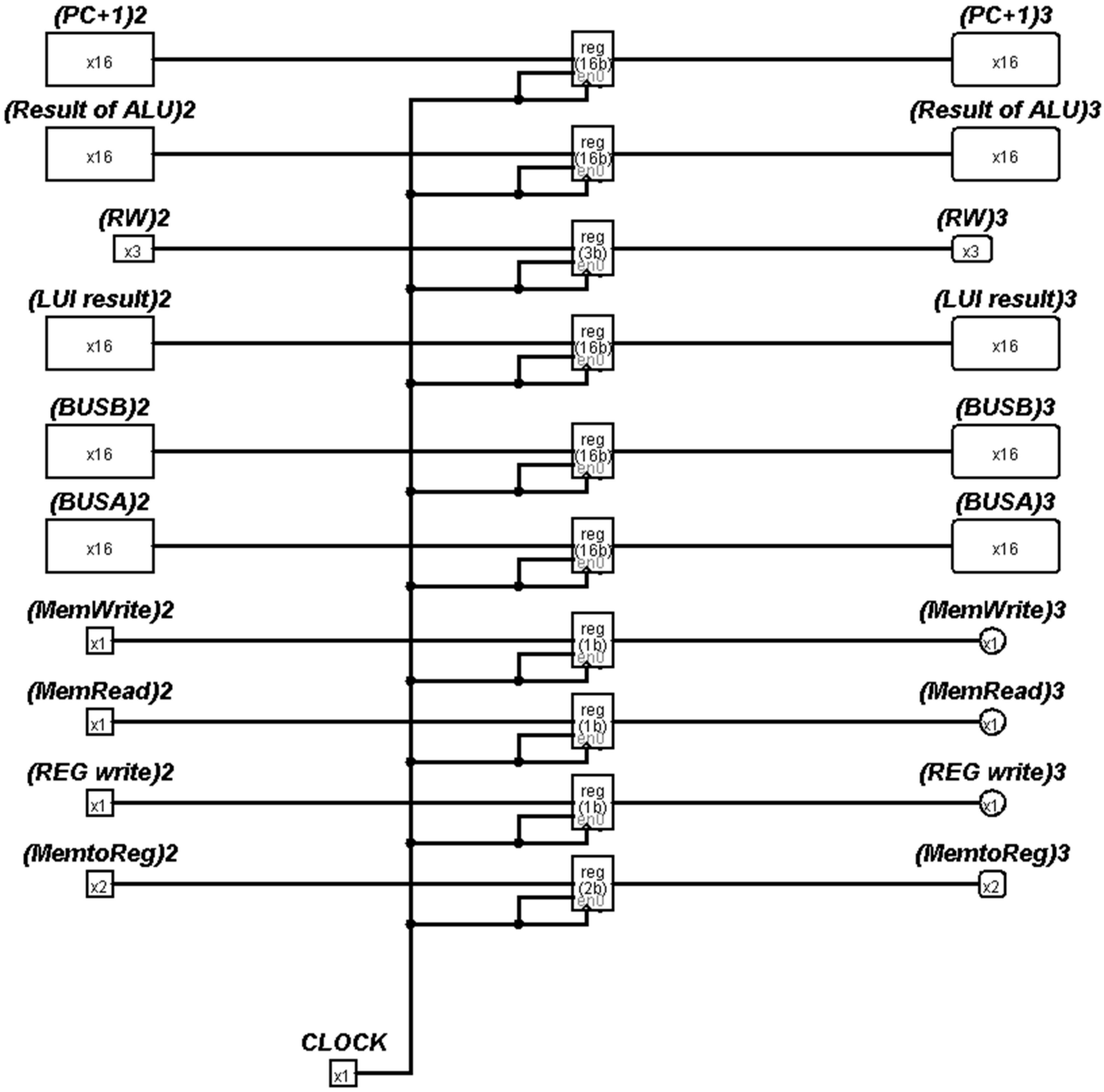

(signed (Imm11))1
x16
reg (16b)
x16
(signed (Imm11))1
(PC+1)1
x16
reg (16b)
x16
(PC+1)2
(BUSA)1
x16
reg (16b)
x16
(BUSA)2
(BUSB)1
x16
reg (16b)
x16
(BUSB)2
(RW)1
x3
reg (3b)
x3
(RW)2
signed (Imm5) 1
x16
reg (16b)
x16
signed (Imm5) 2
unsigned (Imm5) 1
x16
reg (16b)
x16
unsigned (Imm5) 2
(MemWrite)1
x1
reg (1b)
x1
(MemWrite)2
(MemRead)1
x1
reg (1b)
x1
(MemRead)2
(ALUSrc)1
x2
reg (2b)
x2
(ALUSrc)2
(REG write)1
x1
reg (1b)
x1
(REG write)2
(MemtoReg)1
x2
reg (2b)
x2
(MemtoReg)2
(Alu Signal)1
x4
reg (4b)
x4
(Alu Signal)2
(LUI)0
x1
reg (1b)
x1
(LUI)1
Clock
x1

3) EX/MEM stage:

Mainly this stage targets memory accessing. In which memory is enabled and we can read or write form the memory in LW or ST functions. Also we added all the signals and the selectors that will be used in the next rise of the clock cycle.

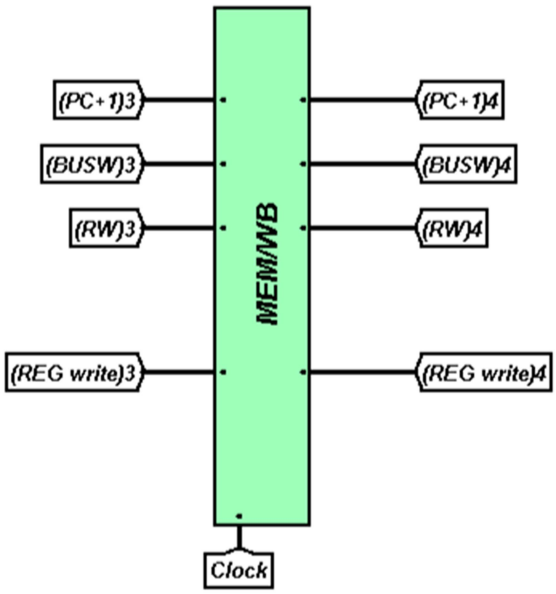


INTERNAL DESIGN OF THE EX/MEM STAGE:

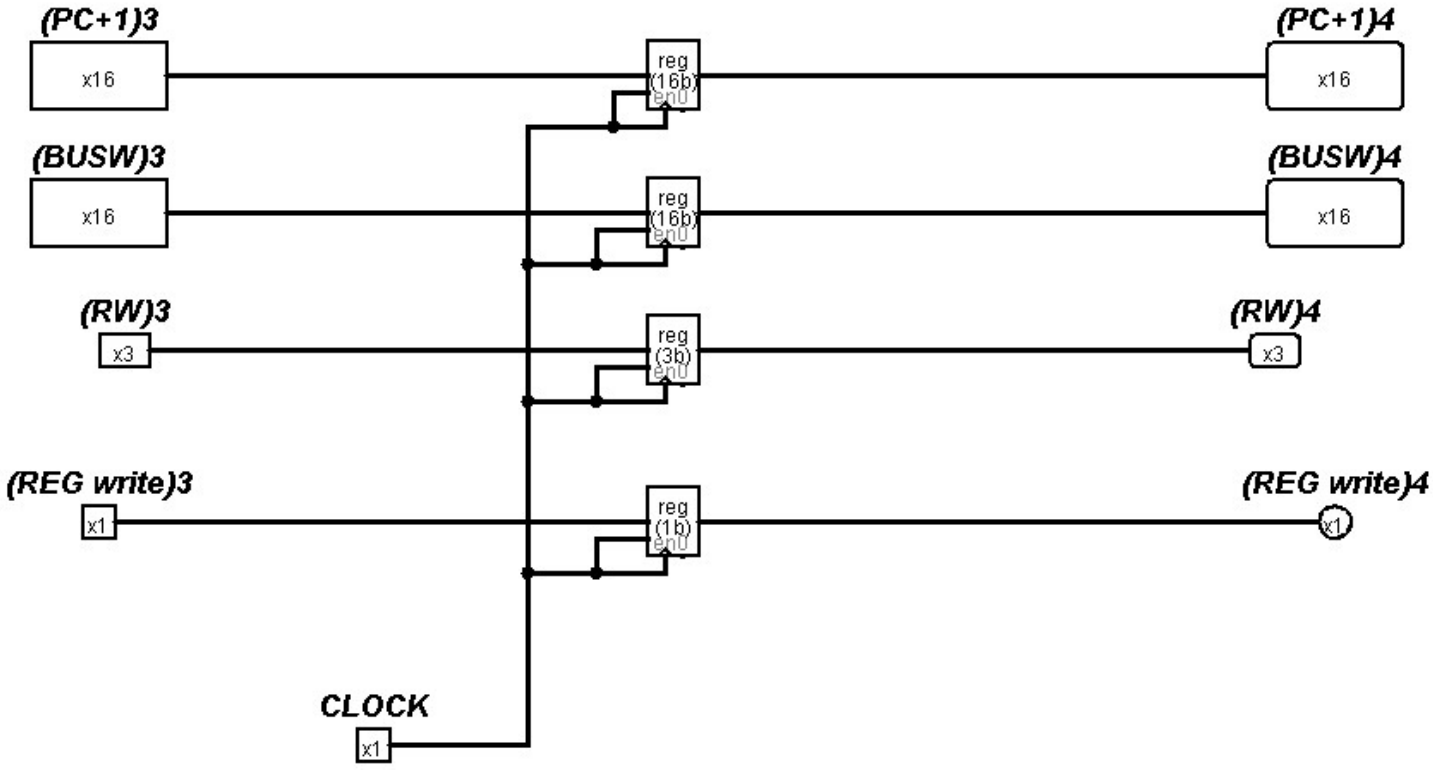


4) MEM/WB stage:

This is the write backstage in which the results of the operations are written in their destination registers. The destination register signal and the value that will be written inside the register are saved in that stage, so we can write in the desired register at the rising edge of the register.

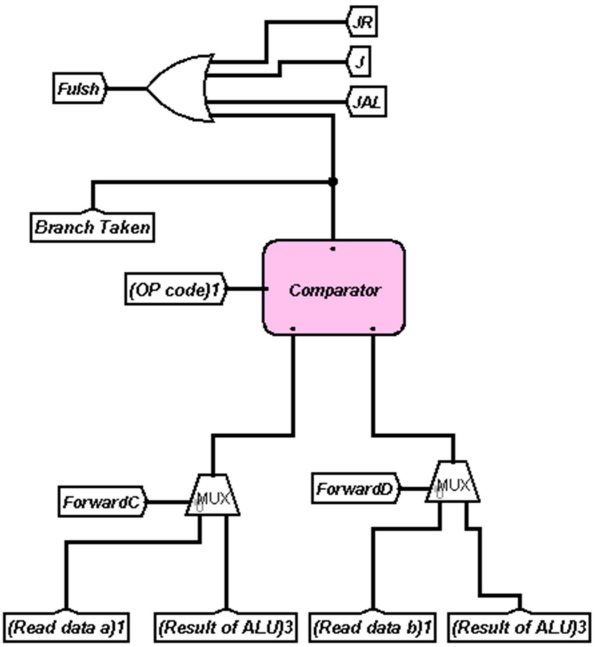


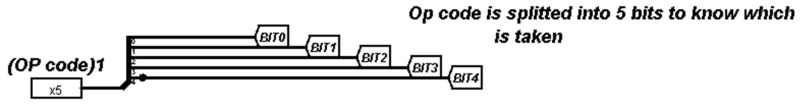
INTERNAL DESIGN OF THE MEM/WB STAGE:

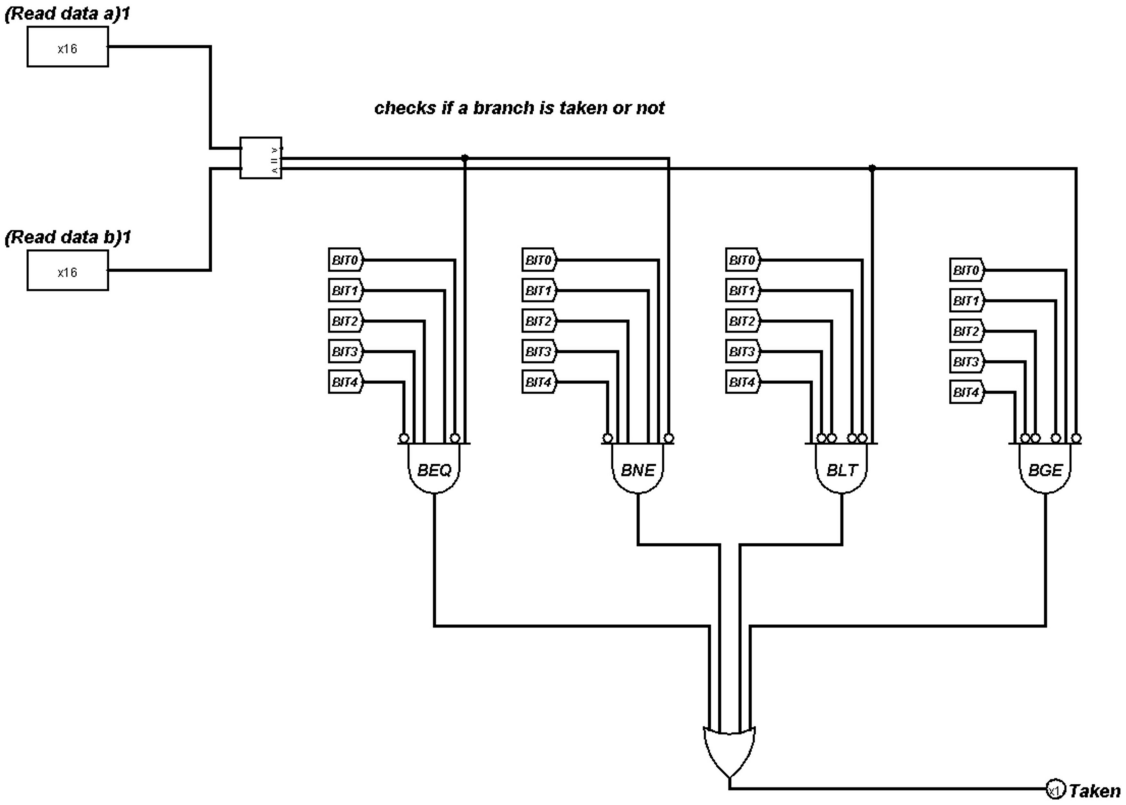


Comparator:

This device is implemented to check if there is a branch. The inputs of the comparator are the Opcode which is used to check which branch is activated. The other two inputs of the comparator are the outputs of the forwardC and forwardD multiplexers. These two MUXs choose between the data comes out of the forwardA and forwardB MUXs or the result of the ALU. The output of the comparator is the signal that deduces that the branch is taken.







➢ Hazard Detect Forward and Stall:

In this circuit we implement all the forward units and stall circuit.

ForwardA and ForwardB are implemented using a priority encoder in which the required signal is generated when the desired conditions are met. The inputs of those two forwards are result of LUI, result of ALU, data saved in memory, data written in memory, and no forward (signal=000).

For RAW detection hazards:

Else if ((Rs! = 0) and (Rs == Rw4) and (WB.RegWr)) ForwardA = 1 ➔ MEM/WB

Else if ((Rs! = 0) and (Rs == Rw3) and (MEM.RegWr)) ForwardA = 2 ➔ EX/MEM

If ((Rs! = 0) and (Rs == Rw2) and (EX.RegWr)) ForwardA = 3 ➔ ALU

If ((Rs! = 0) and (Rs == Rw2) and (EX.RegWr) and LUI) ForwardA = 4 ➔ result of LUI

Else ForwardA = 0 ➔ ELSE

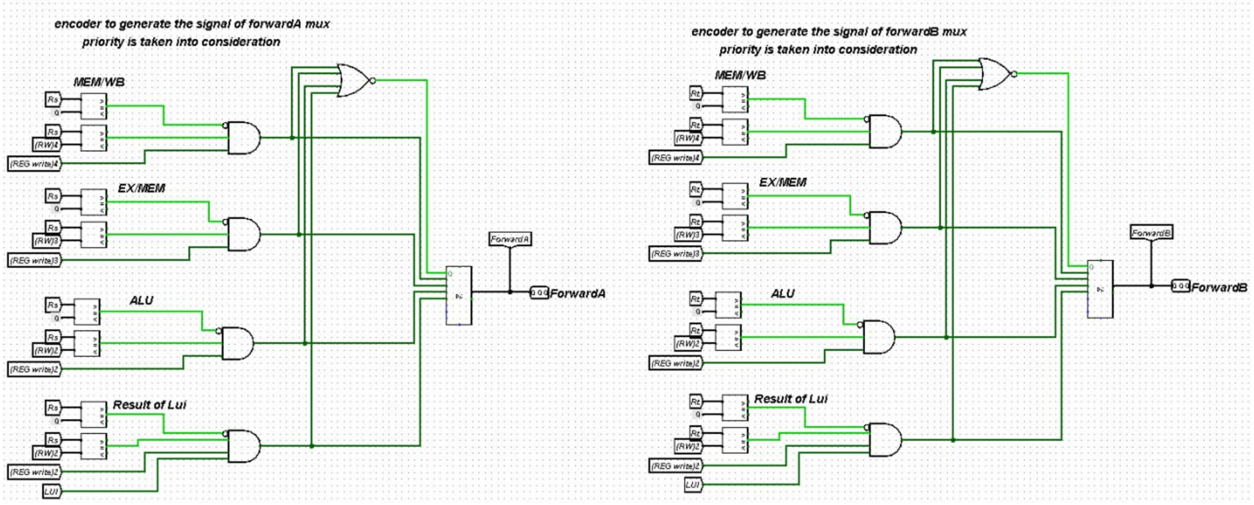
Else if ((Rt! = 0) and (Rt == Rw4) and (WB.RegWr)) ForwardB = 1 ➔ MEM/WB

Else if ((Rt! = 0) and (Rt == Rw3) and (MEM.RegWr)) ForwardB = 2 ➔ EX/MEM

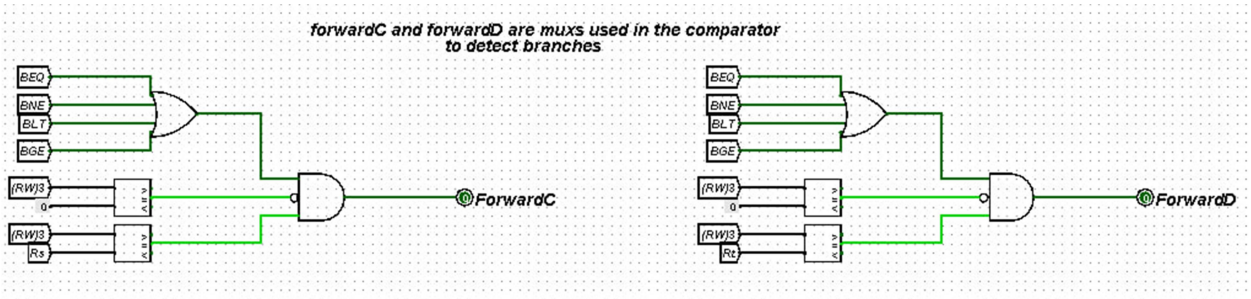
If ((Rt! = 0) and (Rt == Rw2) and (EX.RegWr)) ForwardB = 3 ➔ ALU

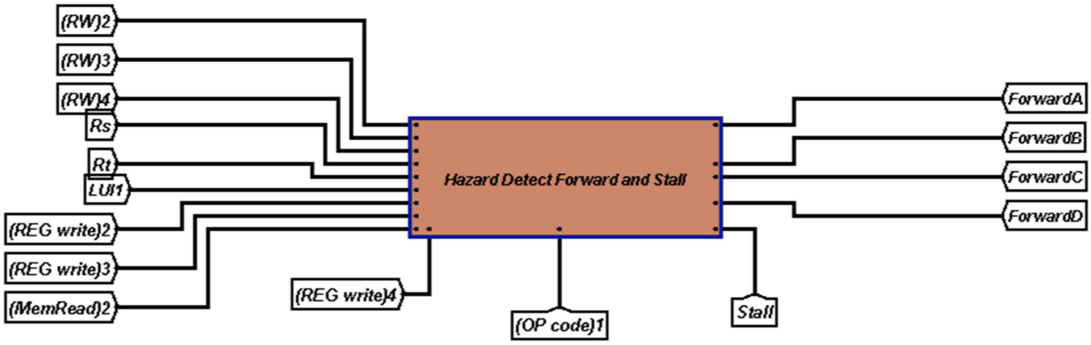
If ((Rt! = 0) and (Rt == Rw2) and (EX.RegWr) and LUI) ForwardB = 4 ➔ result of LUI

Else ForwardAB= 0 ➔ ELSE

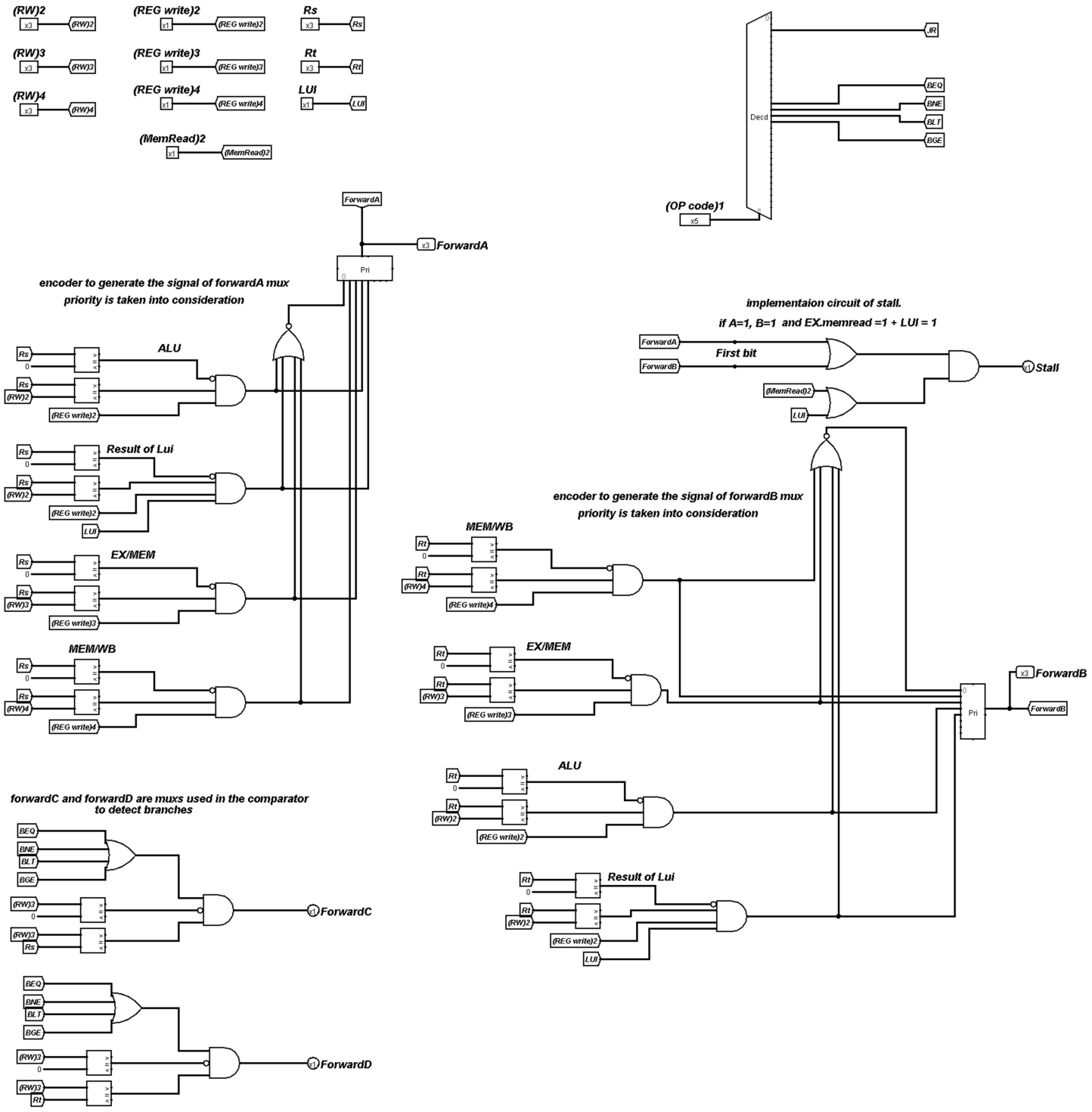


According to forwardC and ForwardD, they are used in the comparator for the detection of the branches.



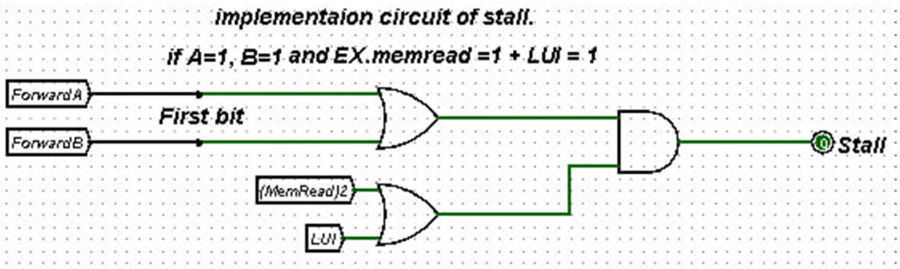


INTERNAL DESIGN OF HAZARD DETECT FORWARD AND STALL:



➢ Flush and stall circuits:

Stall circuit is activated in LW and LUI instruction. it is considered as a solution for Read after Write hazard. It delays an operation specific number of cycles till another operand is prepared in the register file to be used in the execution of the operation. When we stall the first pipeline IF/ID is frozen (do not change). The selectors of the MUXs signals are all zeros.



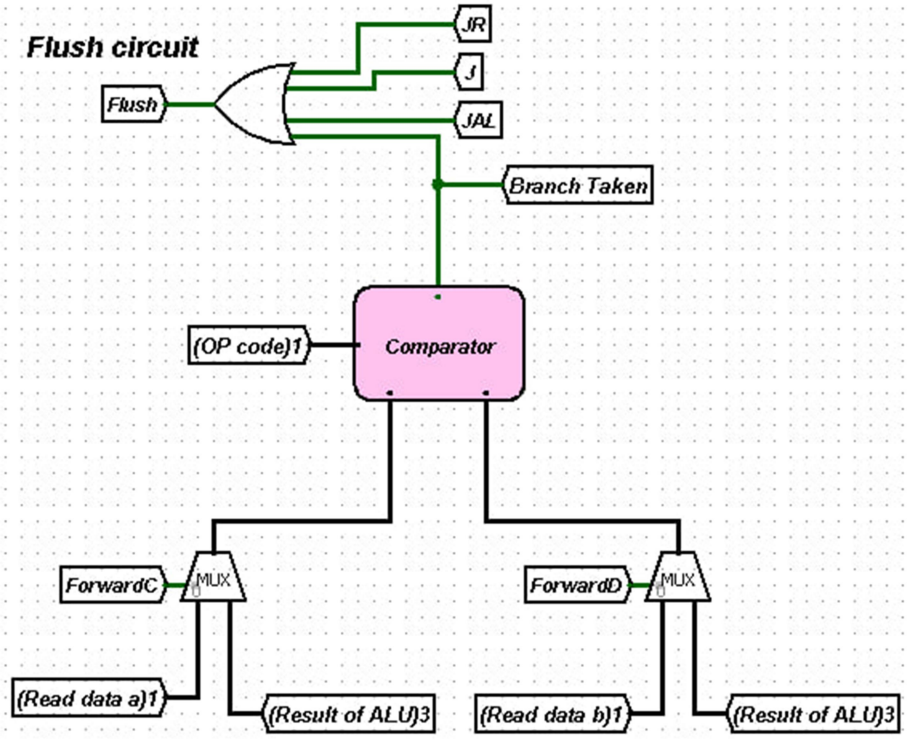
If ((EX.MemRd == 1) OR (LUI = 1)

(ForwardA==1 or ForwardB==1)) Stall // RAW Hazard

⇨ Flush circuit is to zero the instruction field of the IF/ID pipeline register. It means inserting a no operation.

It occurs in all types of Jump cases, and branches (if taken).

It takes place in the decode stage.



➢ Test codes:

- The following code is testing most of the operations and functions which is implemented in the circuit. It also examines the forwards, stalls and flushes.
- The following table shows the code instruction, code in binary, conversion of code to hex, and the expected value for each instruction.

Instruction number	instruction	code in binary	code in hex	expected value
0000	Lui 0x384	1001001110000100	9384	R1=0x7080
0001	Addi $5, $1,13	0011101101001101	3B4D	R5=0x708d
0002	Xor $3, $1, $5	0000010011001101	04CD	R3=0x000d
0003	Lw $1, 0($0)	0110000000000001	6001	R1=0x0001
0004	Lw $2, 1($0)	0110000001000010	6042	R2=0x0001
0005	Lw $3, 2($0)	0110000010000011	6083	R3=0x000a
0006	Addi $4, $4, 10	0011101010100100	3AA4	R4=0x000a
0007	Sub $4, $4, $4	0000101100100100	0B24	R4=0x0000
0008	L2: Add $4, $2, $4	0000100100010100	0914	R4=0x0001 / R4 finally= 0x0037
0009	Slt $6, $2, $3	0000110110010011	0D93	R6=0x0001/R6 finally =0x0000
000a	Beq $6, $0, L1	0111000011110000	70F0	Branch after ten iterations
000b	Add $2, $1, $2	0000100010001010	088A	R2=0x0002/R2 finally =0x000a
000c	Beq $0, $0, L2	0111011100000000	7700	Branch to L2
000d	L1: Sw $4, 0($0)	0110100000000100	6804	Location 0 = 0x0037
000e	Jal Func	1111100000000100	F804	R7=0x000f & jump to func
000f	Sll $3, $2, 6	0100000110010011	4193	R3=0xc280
0010	ROR $6, $3, 3	0101100011011110	58DE	R6=0x1850
0011	beq $0,$0,-1	0111000000000000	7000	program is over, keep looping back to here
0012	Func: or $5, $2, $3	0000001101010011	0353	R5=0x000a
0013	Lw $1, 0($0)	0110000000000001	6001	R1=0x0037
0014	Lw $2, 5($1)	0110000101001010	614A	R2=0x430a
0015	Lw $3 ,6($1)	0110000110001011	618B	R3=0x7342
0016	And $4, $2, $3	0000000100010011	0113	R4=0x4302
0017	Sw $4, 0($0)	0110100000000100	6804	Location 0 = 0x4302
0018	Jr $7	0001000000111000	1038	PC =0x000f to SLL

First of all we save these values in the data memory (RAM).

Memory starting from address 0 contains:

M[0]= 0001

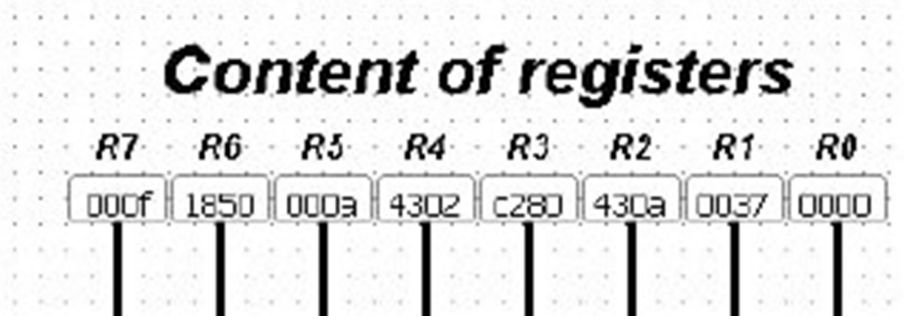
M[1]=0001

M[2]=000a

M[60]=430a

M[61]=7342

The expected result of the output of register:



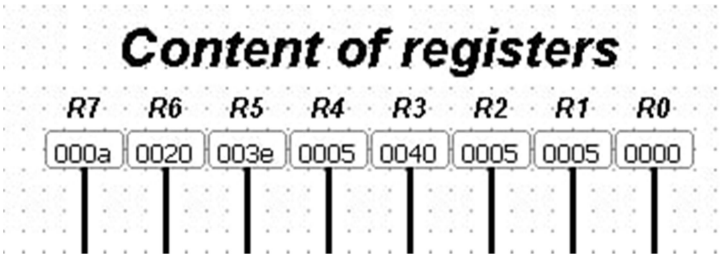
****

❖ Second code for testing:

The following code creates an array of five elements in the memory. In each element it saves the multiple of 2. After that a summation is done over the elements in each location in the memory. The summation is calculated and the result is saved in R5. Expected result R5 = 2+4+8+16+32= 62

Instruction number	instruction	code in binary	code in hex	expected value
0000	AddI $1,$0,5	0011100101000001	3941	R1=5
0001	XOR $2,$2,$2	0000010010010010	0492	R2=0
0002	AddI $3 ,$0 ,2	0011100010000011	3883	R3= 2
0003	Loop1 : SW $3,0($2)	0110100000010011	6813	Location 0 = 2
0004	SLL $3, $3,1	0100000001011011	405b	R3=2*2=4
0005	AddI $1,$1,-1	0011111111001001	3fc9	R1-- / finally R1=0
0006	AddI $2,$2,1	0011100001010010	3852	R2++
0007	BNE $1,$0,Loop1	0111111100001000	7f08	Branch after five iterations
0008	AddI $1,$0,5	0011100101000001	3941	R1=5
0009	JAL Sum	1111100000000011	F803	R7=PC+1 and jump to sum
000a	Add $5,$5,$0	0000100101101000	0968	R5 =0x003e=62
000b	Finish : BEQ $0,$0,Finish	0111000000000000	7000	program is over, keep looping back to here
000c	Sum : LW $6,0($4)	0110000000100110	6026	save the value from MEM to R6
000d	Add $5,$5,$6	0000100101101110	096e	R5+=R6
000e	AddI $4,$4,1	0011100001100100	3864	R4++
000f	BLT $4,$1,Sum	1000011101100001	8761	Not Branch after five iterations
0010	Jr $7	0001000000111000	1038	jump to instruction num 000a

The expected values of the whole registers are:



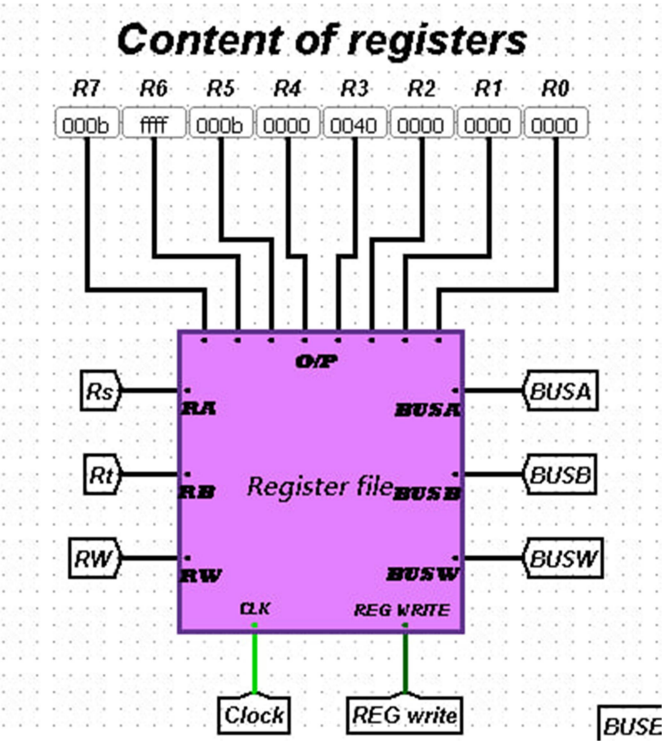
****

❖ Third code for testing:

The following code examines various operations in the MIPS processor. Mainly, it initiates some registers with certain values then a loop is implemented using BLT since the branch satisfies the conditions. The program terminates the branch when R2 is not less than R1 and then jumps to exit.

Instruction number	instruction	code in binary	code in hex	expected value
0000	addi $1, $1, 5	0011100101001001	3949	R1 =0X0005
0001	addi $2, $2, 0	0011100000010010	3812	R2=0X0000
0002	addi $3, $3, 2	0011100010011011	389B	R3=0X0002
0003	ori $5, $5, 11	0010101011101101	2AED	R5=0X000b
0004	sw $5, 0($3)	0110100000011101	681D	location 3 = 0x000b
0005	L1:addi $1, $1, -1	0011111111001001	3FC9	R1--/finally R1=0
0006	sll $3, $3, 1	0100000001011011	405B	R3=0x0004/ finally R3=0x0040
0007	lw $6 , 0($3)	0110000000011110	601E	R6=0x0000/finally R6=0x0000
0008	nor $6, $6, $6	0000011110110110	07B6	R6=0xffff/finally R6=0xffff
0009	blt $2, $1, L1	1000011100010001	8711	not branch after five iteratiors
000a	jal exit	1111100000000001	F801	R7=0x000b(R7)
000b	addi $0 , $0 , 0	0011100000000000	3800	INFINITY LOOP
000c	jr $7	0001000000111000	1038	

❖ **The expected output values of the whole registers are:**



➢ Team work:

Mina➔ detection hazard unit, pipeline registers, test code, documentation, and forward unit.

Abanob ➔ main design (data path), test code, documentation, pipeline registers, forward units.

Mark ➔ comparator, main design, forward unit, test code.

➢ Plan for completing subtasks of the project:

First (F2F) meeting ➔ we designed the four pipeline registers.

Second (F2F) meeting ➔ we work on the data path of the four pipeline registers.

Third (online) meeting ➔ we were concerned about comparator and hazard detection unit.

Fourth (F2F) meeting ➔ we designed the comparator circuit, detection hazard and forward units.

Fifth (online) meeting ➔ we finished the whole circuit.

Sixth (online) meeting ➔ testing the code and make sure all operations are working.

Eighth (online) meeting ➔ we worked on the test code and recorded the illustration of our project. We also finished the report.