Faculty of Engineering

Computer Engineering

Single Cycle & Pipeline Processor Project

Instructor: DR. Gihan Naghib
presented by:

Ahmed Attia Awis
Mohammed Ali Aiaty
Mohammed Khaled Shaban
Moaz Mahmoud Ahmed

Introduction:

we have implemented our Processor into two Phases:

1- Single cycle processor

2- Pipelined

phase 1: Single Cycle Processor:

❖ *Design and Implementation*

Is a processor that carries out one instruction in a single Clock cycle.

And to design it we need to design 4 main blocks. The blocks are:

• Register File

• ALU

• Next Program counter

• Control unit

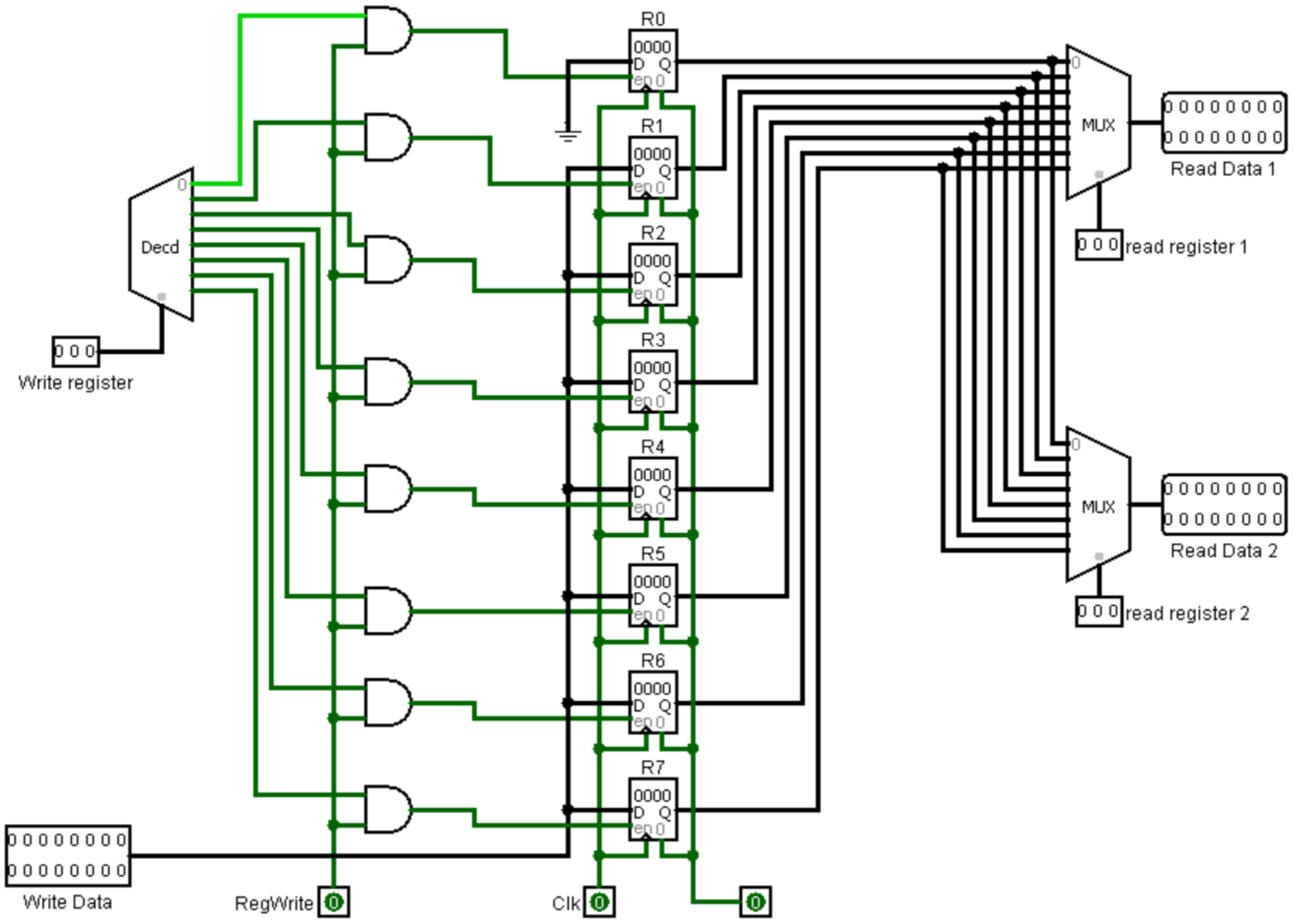
For Reliability and to make the design Process easier we have divided our main Control unit into three units which produce the whole control signals for the variant units of the Processor.. The three units are :

- The path control unit
- ALU control unit
- Branch control unit

1. Register File:

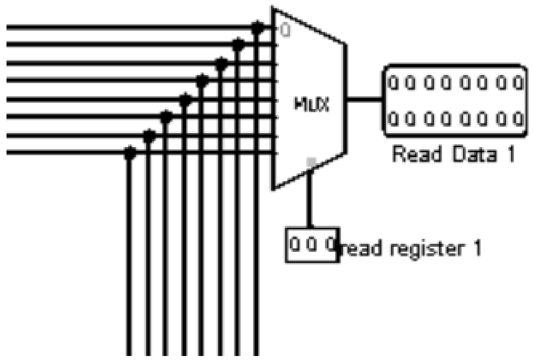
This Block contains 8 16-bit registers from R_0 to R_7 and R_0 is hardwired to 0.

Register file was designed by basic way as we combined eight registers on two access paths for reading the registers, the access for each path was implemented was a one multiplexer which has its selectors controlled by the input address for the register file.
For Writing process, the data to be written is available on each register but the target register is enabled by a MUX whose selectors is attached to the input write address.

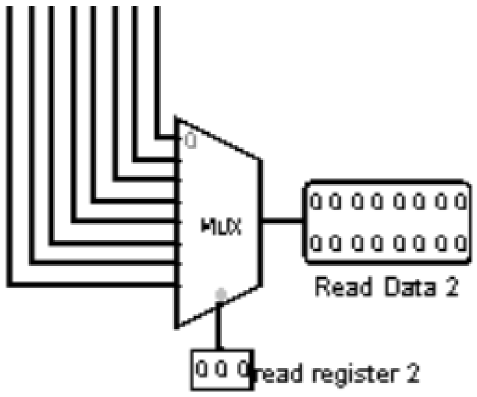


To manage we need 2 multiplexers:

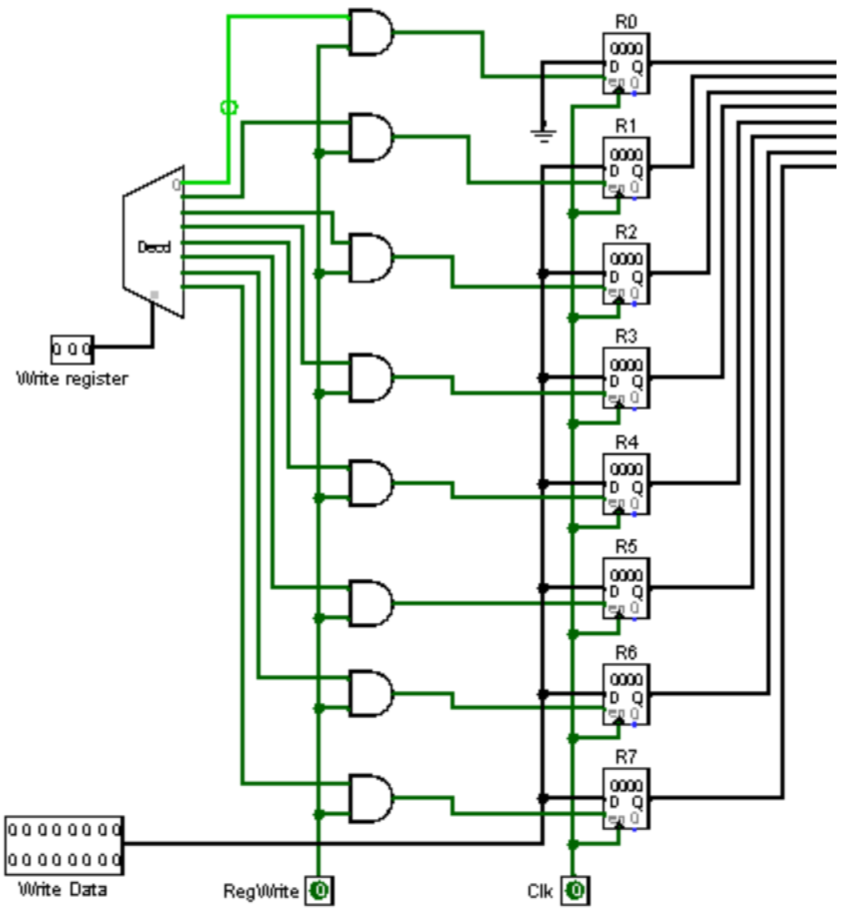
i. The first multiplexer with 3-bit selector to read the value of R_S. So it takes the bits (5-7) from the instruction as the first multiplexer's selector as shown.



ii. The second multiplexer with 3-bit selector to read the value of second outpu



We also need a decoder with 3-bit selector to write the input to R_d, so it takes the bits(8-10) from the instruction as the demultiplexer's selector as shown.



The output of the decoder is directed to AND gates to let the Wr_enable control signal to decide when to write the register file.

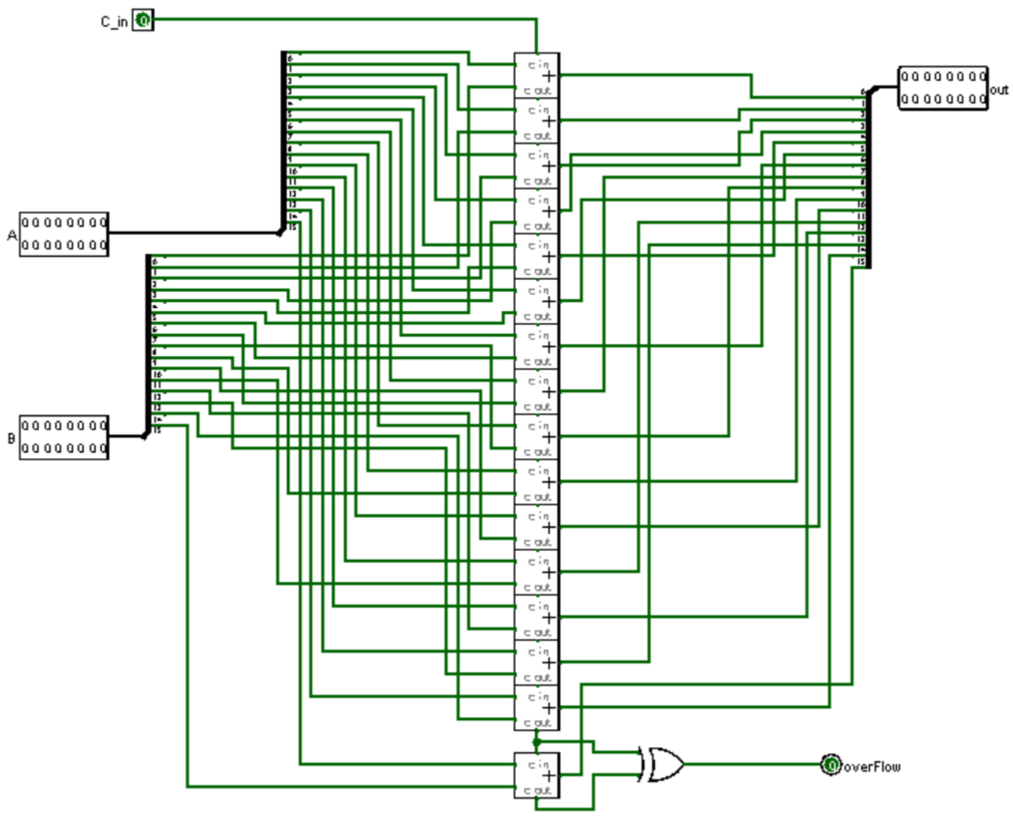
2. ALU

This block contains 2 blocks:

- Arithmetic unit
- logic unit
- Shifter & Rotator

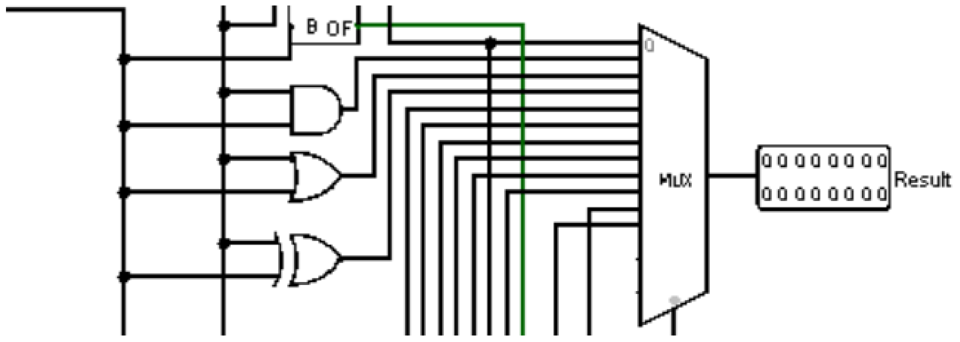
i. Arithmetic unit

For Arithmetic operations we designed an adder which its input has two-sixteen-bit data buses, one of the input bus can be inverted two implement the subtraction operation.



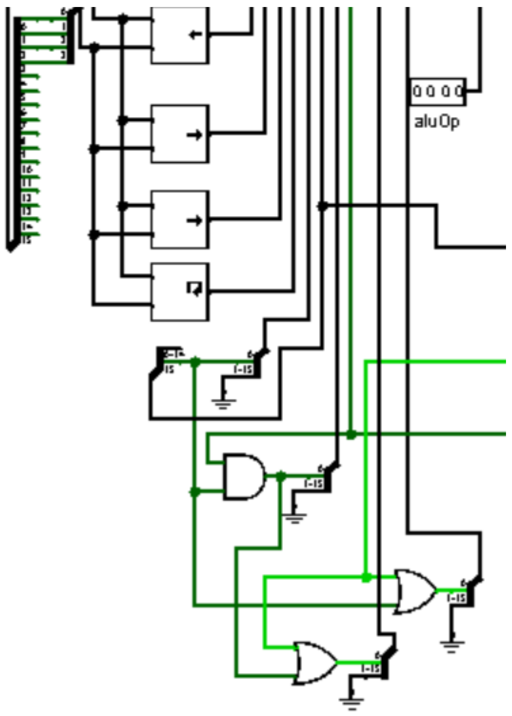
ii. logic unit

For logical operations we combined the basic logical Gates (AND , OR , XOR) on a multiplexer with the arithmetic unit, other logical operation like NOR & NAND can be performed by invers the inputs and perform AND operation or OR (for NAND).

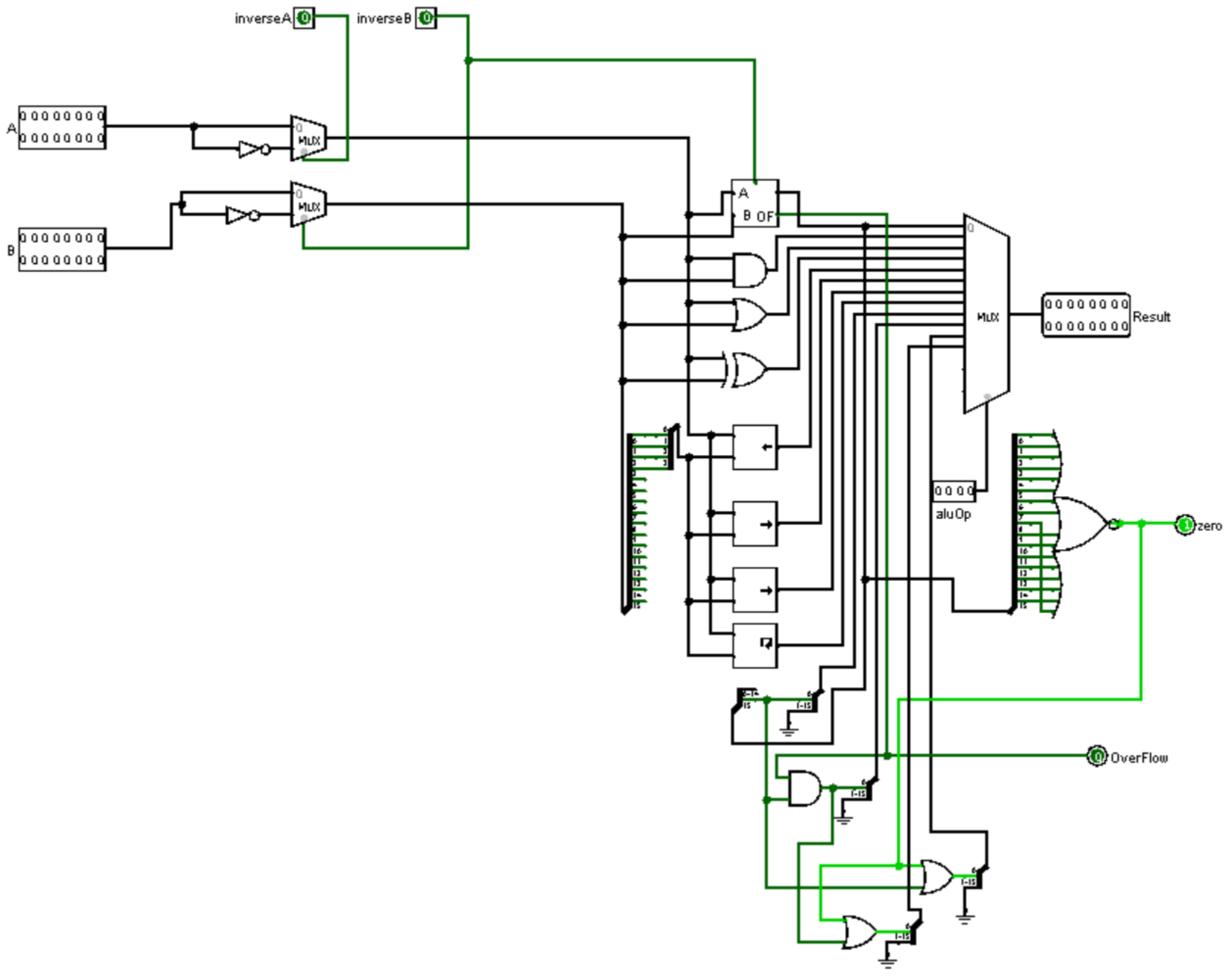


iii. Shifter & Rotator

For shifting and rotating we used shifters and rotators units which also MUXed with the logical gated and arithmetic unit.



The Full ALU Implementation.

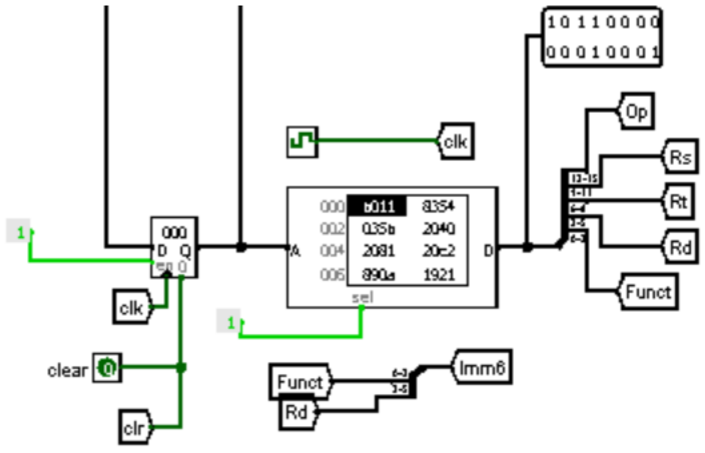


The Table for the MUX selectors aluOp

The Operation	AluOp	Inv_A&B
ADD	0000	00
AND	0001	00
OR	0010	00
NOR	0001	11
XOR	0011	00
SUB	0000	01
SLL	0100	00
SRL	0101	00
SRA	0110	00
ROL	0111	00
SLT	1000	01
SLTU	1001	01
SLTE	1010	01
SLTEU	1011	01

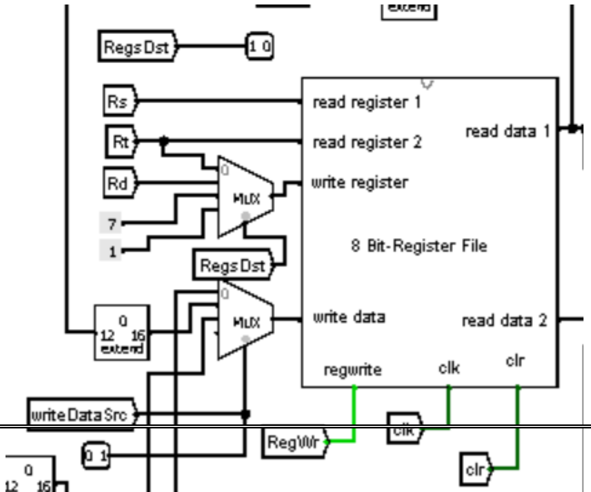
3. Data Path

- First we used a register for PC which is output attached the address of memory and the output of memory which carry the instruction are separated for the bits of the instructions combination(op code, Rt, Rs, Rd, function and Immediate value).



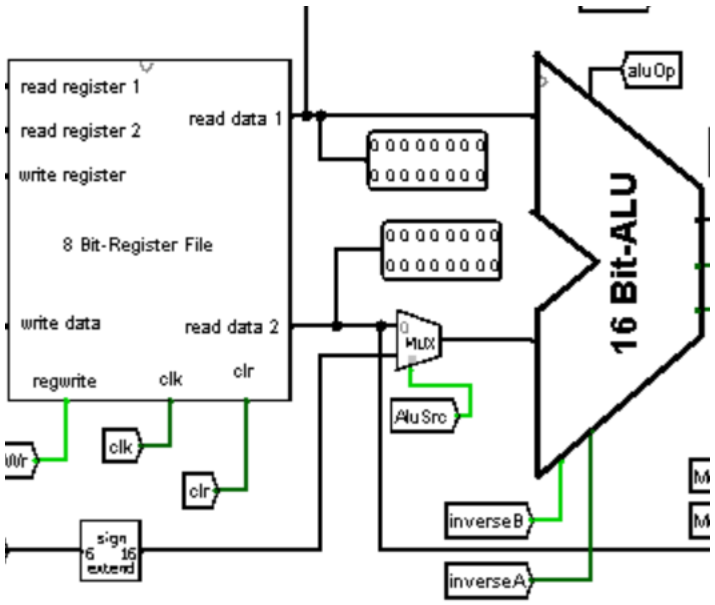
- Then, the op code directed to general control unit, Rs to the first input of the register file, Rt for the second, Rd is muxed with Rt and constant values of 7 & 1 to the write address of the register file.

- For the write data it is possible to come from the memory output or Alu or the PC + 1 content so we put them on a mux which its selectors decided by the

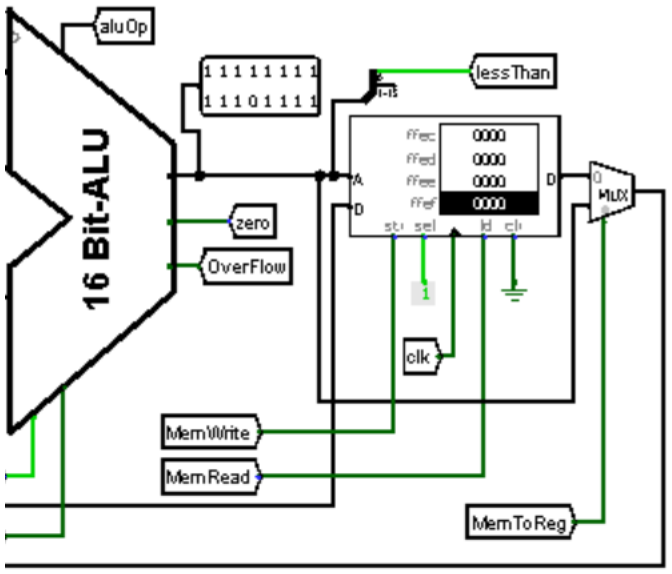


control unit.

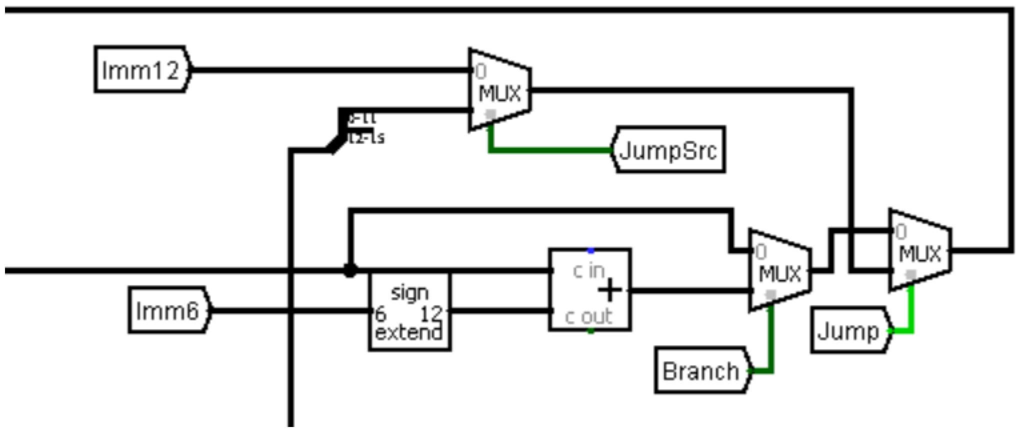
- The first output of the register file is connected to input A of the Alu, the input B could be coming from the register file or the extended 6 bits immediate value so we put them on a Mux controlled by control unit.



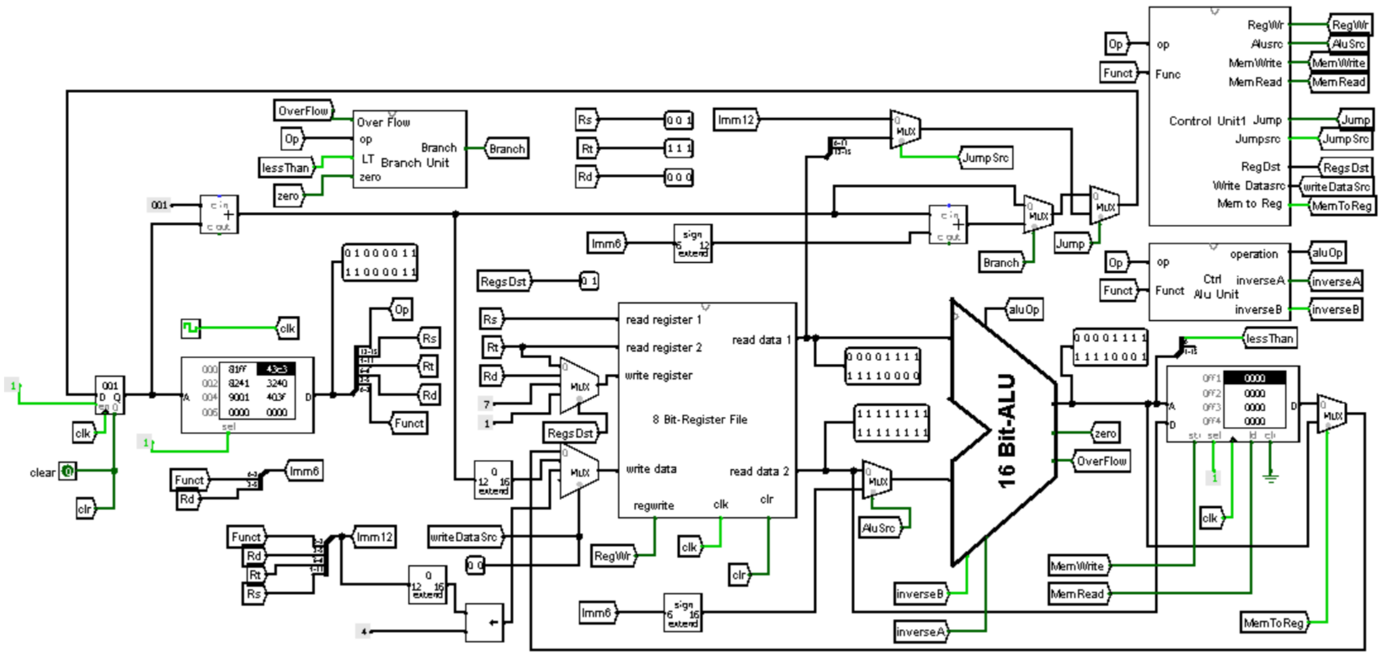
- The sixteen bit output of the Alu connected to the address input the data memory to be used in the store & load instruction and also directed back to the write data path of the register file, the decision up to the control unit.



- The updated the content of the PC register is calculated by several means (PC + 1 , 12-bits immediate value, PC + 1 + 6-bits immediate value for branches) so we used MUXs to decide which value to be written.



The Full Data Path.



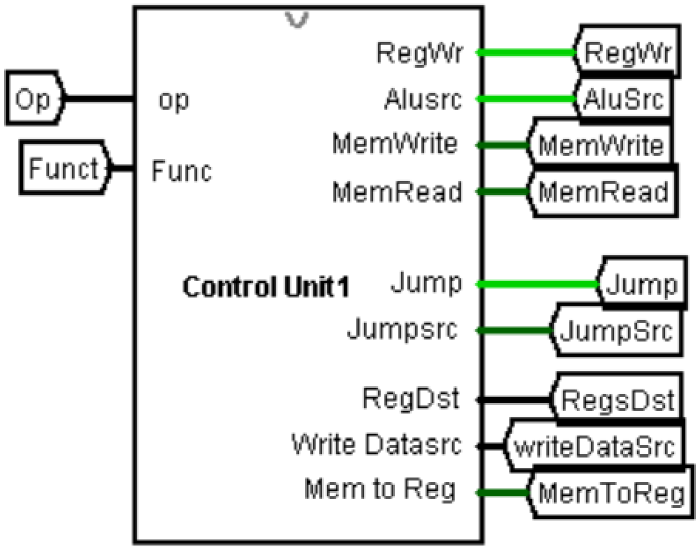
4.Control Unit

As we said before, to reduce the truth table of the combinational circuit of the control unit, we prefer to separate the condition which the unit decide to perform a branch or not onto a sub unit produce one control unit “Branch”. Also another unit to generate the aluop signals for the ALU unit depends on the type of the instruction.

We can explore each one with short details in the next sub sections.

I. General Control Unit

A control unit coordinates how data moves around a cpu. The control unit (CU) is a component of a computer's central processing unit (CPU) that directs operation of the processor. It tells the computer's memory, arithmetic/logic unit and input output devices how to respond to a program's instructions.
We can determine the type of the Instruction form the All bits.



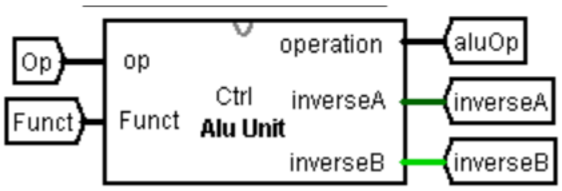
The truth Table of the combinational general control unit.

INST	REGWR	ALUSrc	MEMWrite	MEMRead	Jump	JumpSrc	RegDst	WrDSrc	MEMtoReg
LW	1	1	0	1	0	X	00	00	0
SW	0	1	1	0	0	X	XX	XX	X
ANDI	1	1	0	X	0	X	00	00	1
ORI	1	1	0	X	0	X	00	00	1
ADDI	1	1	0	X	0	X	00	00	1
ALL Branches	0	0	0	X	0	X	XX	XX	X
J	0	X	0	X	1	0	XX	XX	X
JAL	1	X	0	X	1	0	10	01	X
LUI	1	X	0	X	0	X	11	10	X

II. ALU Control Unit

ALU control unit decides which operation the ALU must perform for each instruction so it reads the op code and the function and generate the control signals for the Alu.

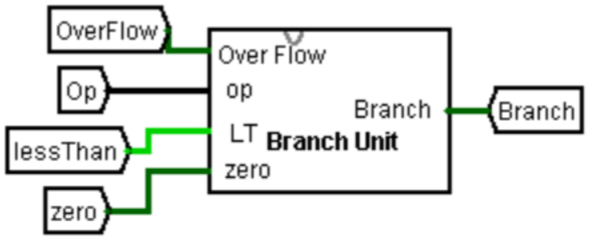
It also designed by building the truth table same to the general one.



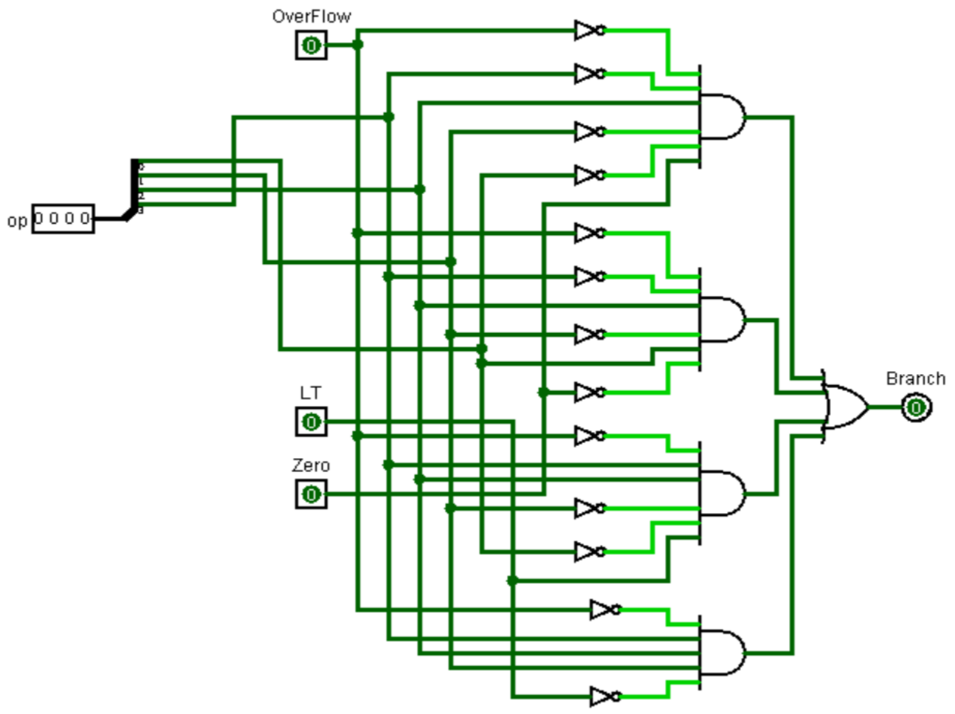
III. Branch Control Unit

For the several condition of the branching instructions must have been to dedicate a separated unit for the unconditional branch.

It takes the zero signal & the less than signal to decide whether to branch or not up to the instruction op code and generate one signal branch.



Implementation of Branch Control



❖ Testing

The above code is to test the processor. third column is to show if the expected results Matches the processor results or not

LUI 0x384	0Xa384	Yes
ADDI r5, r1, 20	0x8354	Yes
XOR r3, r1 , r5	0x035b	Yes
LW R1, 0(R0)	0x2040	Yes
LW R2, 1(R0)	0x2081	Yes
LW R3, 2(R0)	0x20c2	Yes
ADDI R4, R4, 10	0x890a	Yes
Sub R4,R4,R4	0x1921	Yes
Add R4,R2,R4	0x1520	Yes
SLT R6,R2,R3	0x14f2	Yes
BEQ R6,R0,2	0x4c02	Yes
ADD R2,R1,R2	0x1290	Yes
BEQ R0,R0,-5	0x403b	Yes
SW R4,0(R0)	0x3100	Yes
JAL func	0xb012	Yes
SLL R3,R2,R5	0x055c	Yes
ADD R5,R5,R5	0x1b68	Yes
BEQ R0,R0,-1	0x403f	Yes
Code will end here … but for testing SW and JR instructions we will suppose that BEQ R0,R0,-1 not exist		
func: OR R5,R2,R3	0x04e9	Yes
LW R1,0(R0)	0x2040	Yes
LW R2,5(R1)	0x2285	Infinite loop
LW R3,6(R1)	0x22c6	Yes
AND R4,R2,R3	0x04e0	Yes
SW R4,0(R0)	0x3100	
JR R7	0x1e06	

- Output registers

  R0 = 0X0000

  R1=0X0037

  R2=0X430a

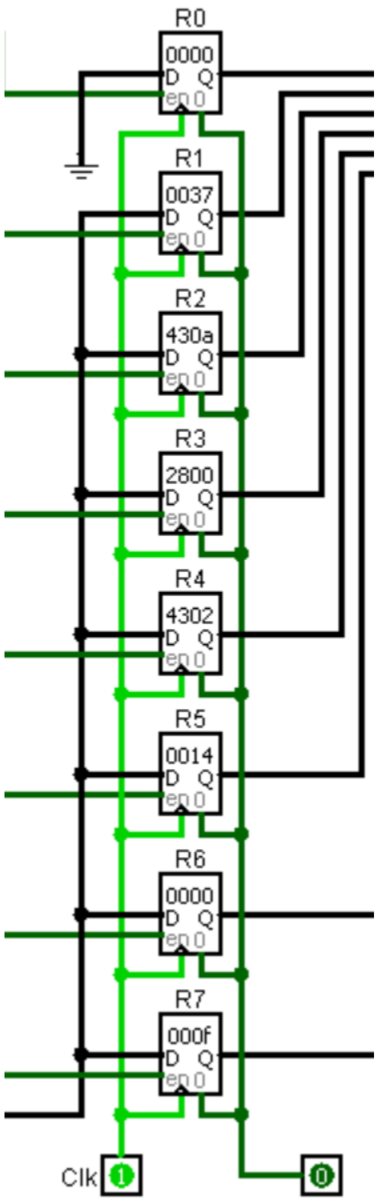
  R3=0X2800

  R4=0X4302

  R5=0X0014

  R6=0X0000

  R7=0X000f



phase 2 : Pipelining the single cycle processor

we started by separating each stage from other stages by inserting some registers files called (IF/ID , ID/EX , EX/MEM , MEM/WB) , between each stage.

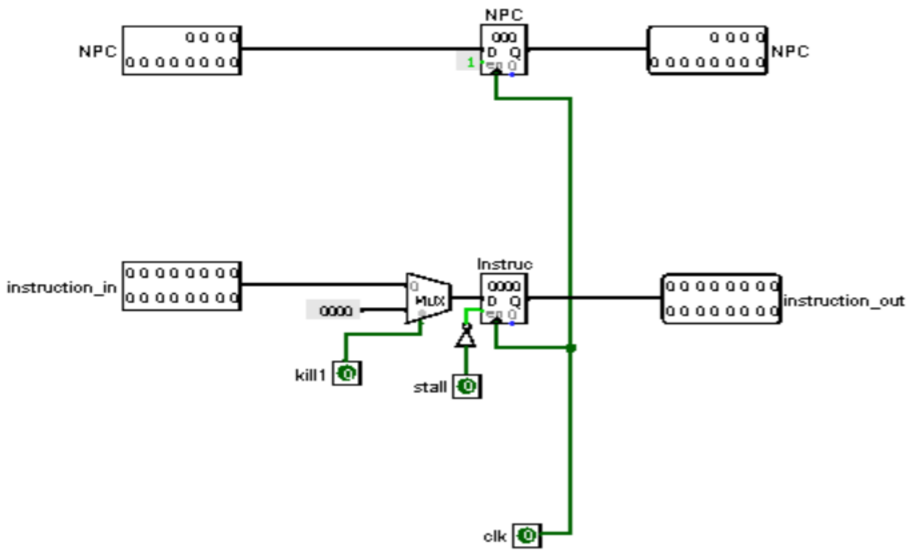
each register file contains inner registers to pass the needed data for a specific stage. Data is Passed from a stage to another on each raise in clock cycle.

We are going to show up all registers files with some light inside.

Pipeline Stage

- **IF / ID**

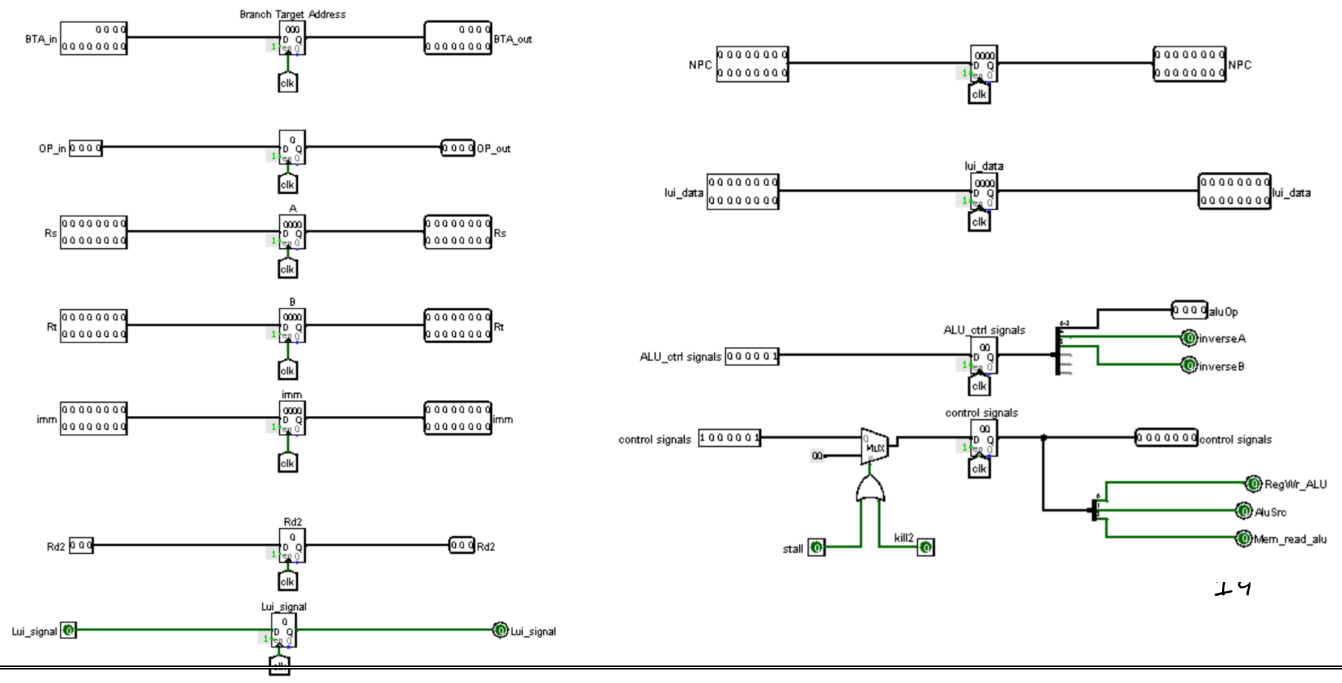
Contain two register one of them is 12-bit carry the NPC, another one which is 16-bit carry instructions.
We did stall for LW instruction and kill1 for branch and jump instructions.



- **ID / EX**

Contain eleven registers which are :

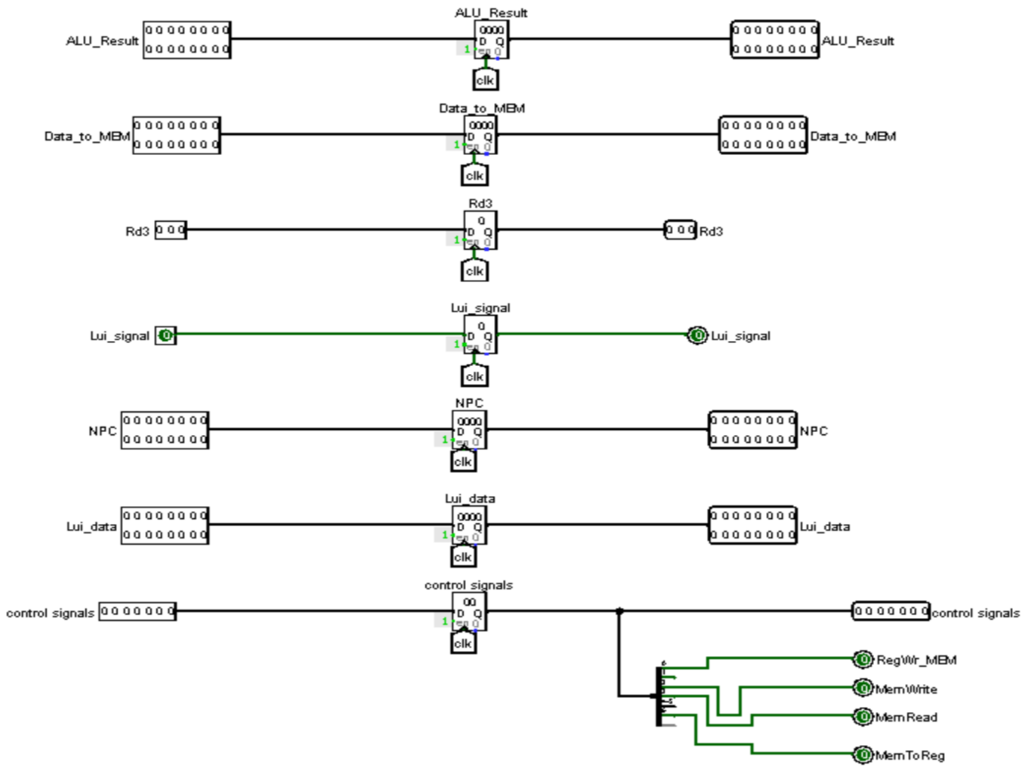
- Branch target address : which link pass the address of branching to the next stage .
- Register needed to pass the Op code that needed in the branch block.
- Register needed to pass input ‘A’ to the Alu.
- Register needed to pass input ‘B’ to the Alu.
- Register needed to pass the immediate value to the next stage.
- Register needed to pass the register destination.
- Register needed to pass the LUI signal to the next stage.
- Register needed to pass the next PC.
- Register needed to pass the LUI value .
- Register needed to pass the control signals of the ALU to the next stage.
- The last register pass the rest of controls signals.



• EX / MEM

There are seven registers needed in this stage:

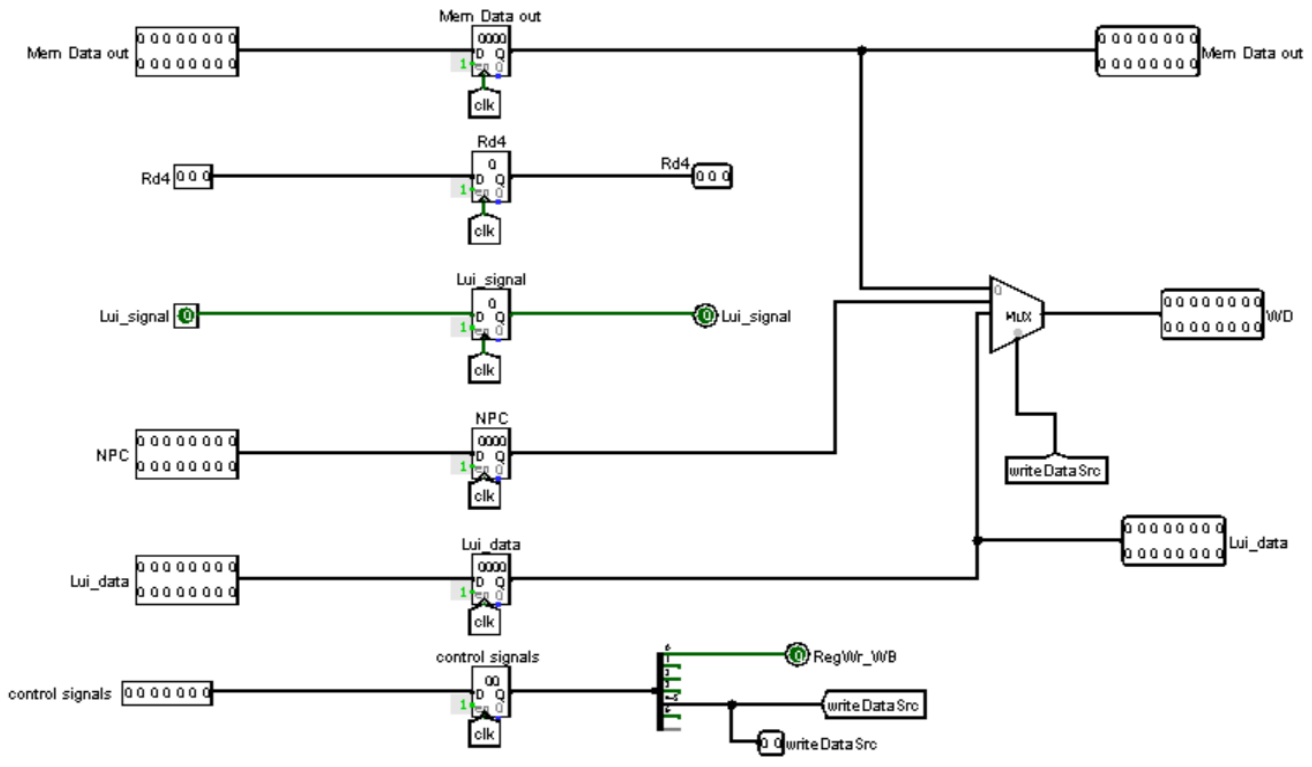
- Register needed to pass the ALU result to the next stage.
- Register needed to pass the data to the Memory.
- Register needed to pass register destination to the next stage.
- Register needed to pass LUI signal to the next stage.
- Register needed to pass the NPC.
- Register needed to pass the LUI data.
- The last one needed to pass the control signals.



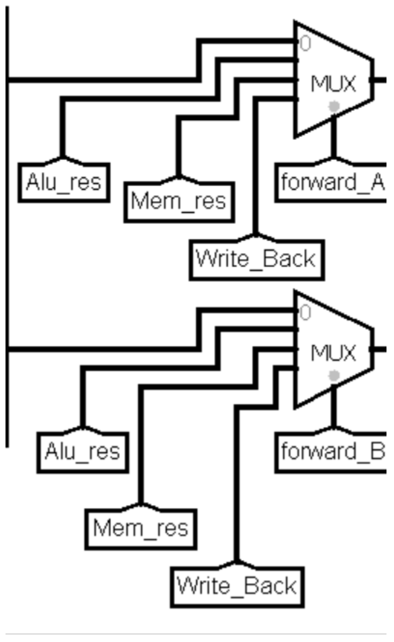
- **MEM / WB**

There are six registers needed in this stage :

- Register needed to pass the output of the memory.
- Register needed to pass the register destination.
- Register needed to pass the LUI signal.
- Register needed to pass the NPC.
- Register needed to pass the LUI data.
- Register needed to pass the control signals.
- A MUX needed to select the written data into the register file.



Forwarding

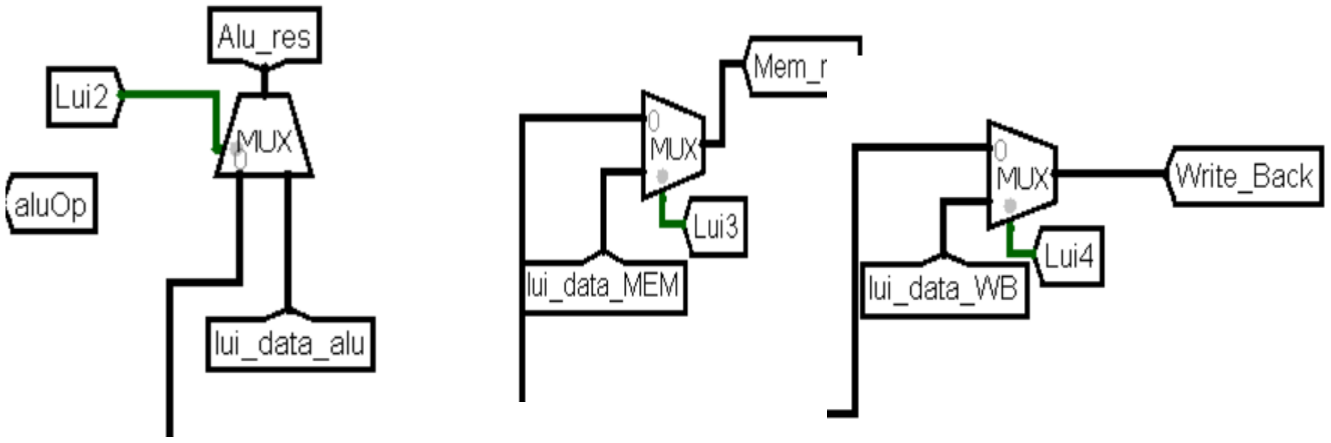


We put two MUXs to select the input to the Execution stage which have to select between data from:

- The result of the ALU.
- The end of the memory stage.
- The write-back stage.

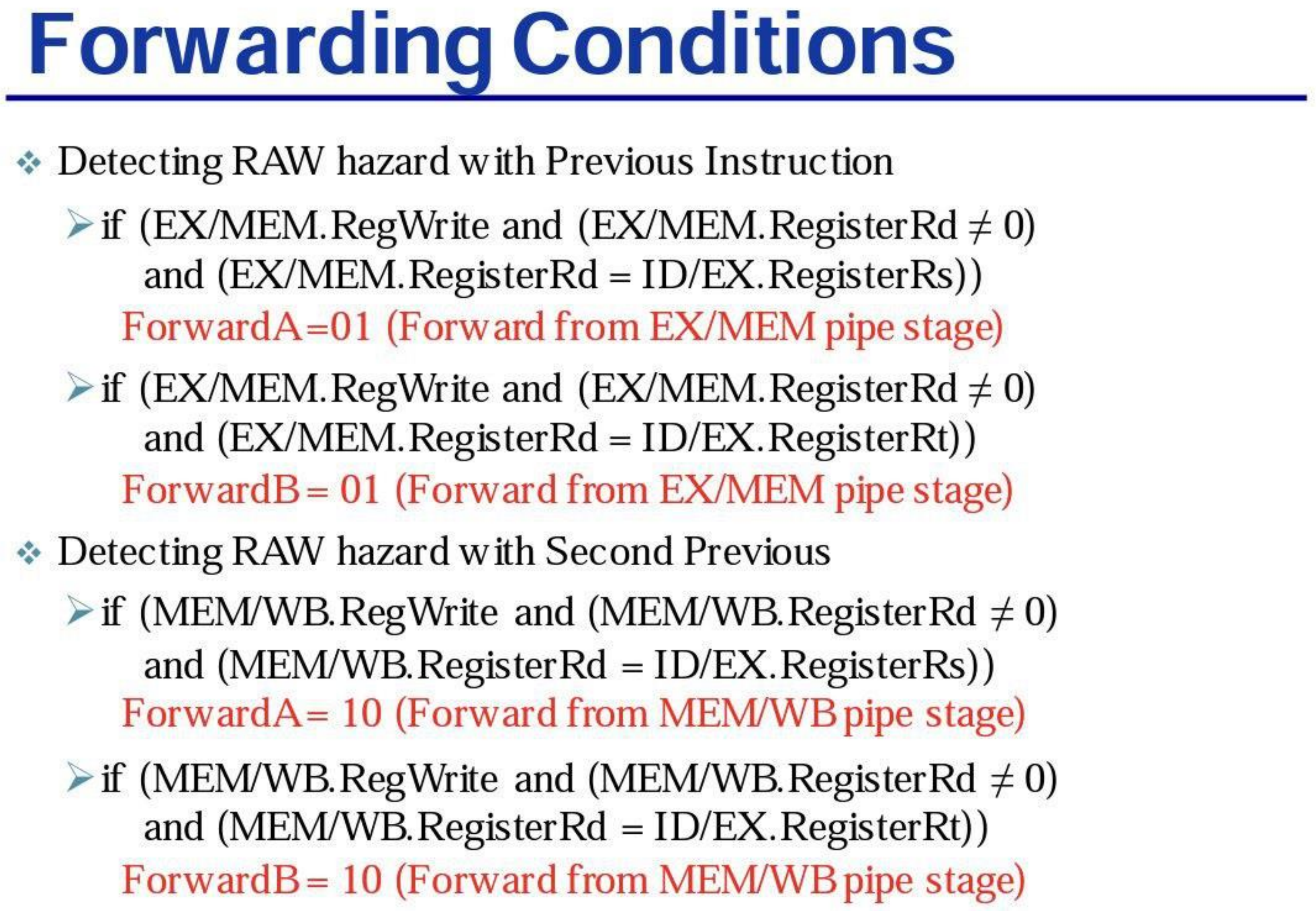
Forwarding problem

As we needed forwarding data of the LUI instruction we had to manage a successful forwarding from each stage, so we put an extra mux inside each stage to choose between the ordinary out data and the LUI data.



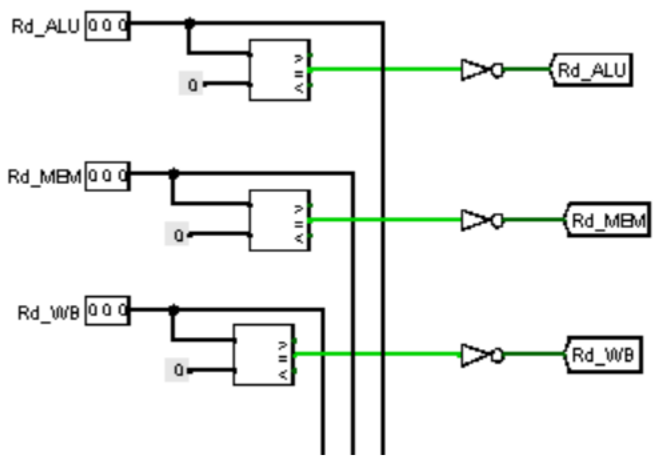
Forward unit:

There are three conditions that needed to consider a forward from a stage to another which are :



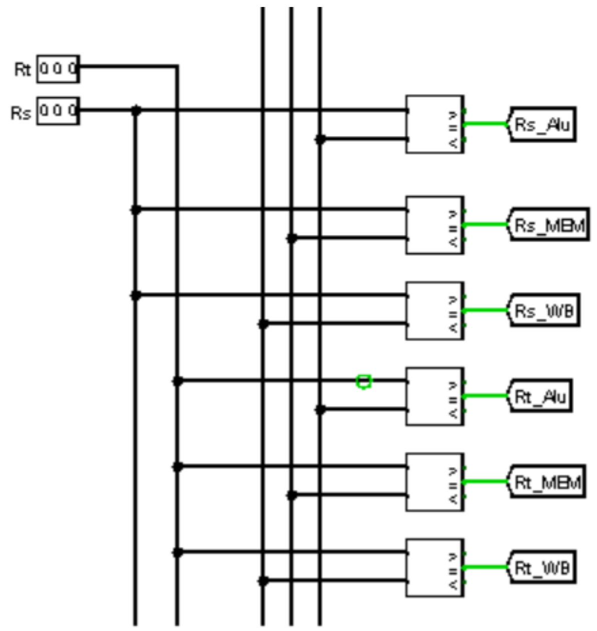
Forwarding Conditions

- Detecting RAW hazard with Previous Instruction
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq$ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA=01 (Forward from EX/MEM pipe stage)
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd $\neq$ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB= 01 (Forward from EX/MEM pipe stage)
- Detecting RAW hazard with Second Previous
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd $\neq$ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA= 10 (Forward from MEM/WB pipe stage)
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd $\neq$ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB= 10 (Forward from MEM/WB pipe stage)

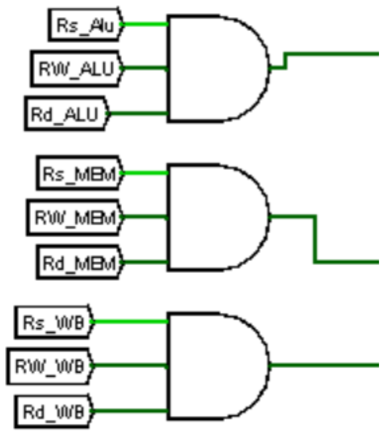


Making sure that Rd not equal zero:

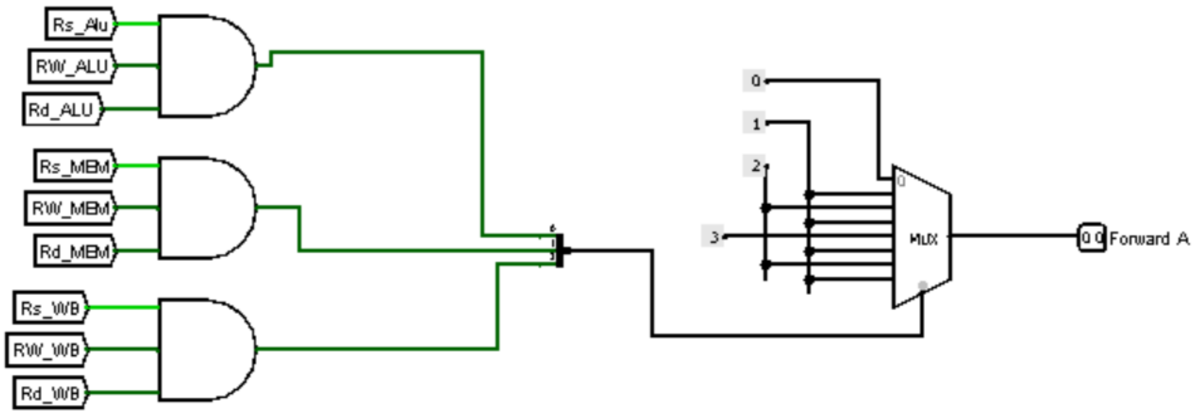
Comparing Rs or Rt with the different Rd from each stage:



Finally making sure that all three conditions are met:

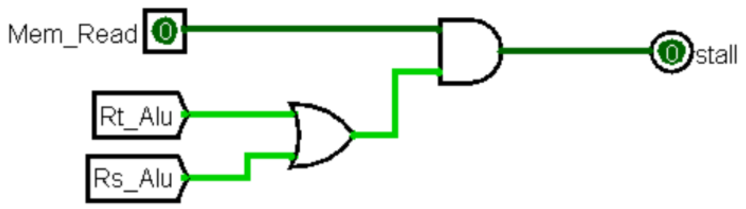


Choosing which stage to forward from:

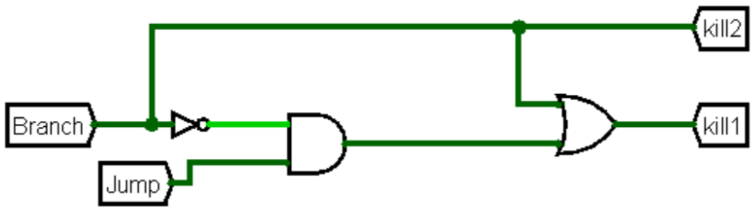


Stalling

We only need to stall when there is load instruction followed by a depended instruction so we make our check to get that dependence between the ID \ EX stages, and we do that for Rs and Rt addresses.



In branch instructions we need to kill two instructions, and in kill instructions we need to kill one instruction.



- As an addition we create a custom Assembler code to translate any instruction in our ISA into the right machine code applicable to our implementation. [customasm Web (hlorenzi.github.io)]