Electrical & Computer Engineering Department
Computer Architecture

ENCS4370
Instructor: Dr. Aziz Qaroush

PROJECT.2

Bader Tawafsheh 1171214
Ahamd Fraij 1171503

Abstract

In this project, we are using Logisim to implement and test a pipelined RISC machine that processes simple 16-bit instructions that all given by the instructor.

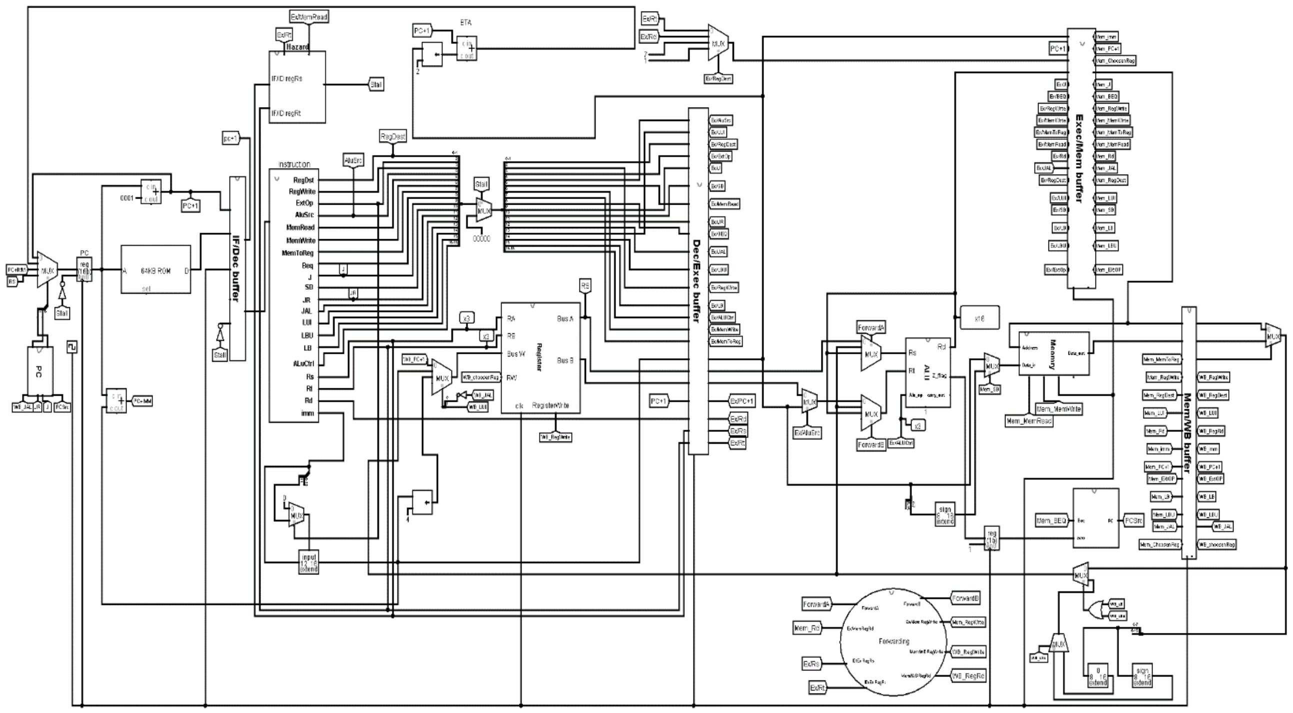
Table of Contents

Design and Implementation

Our design consists of four main blocks:

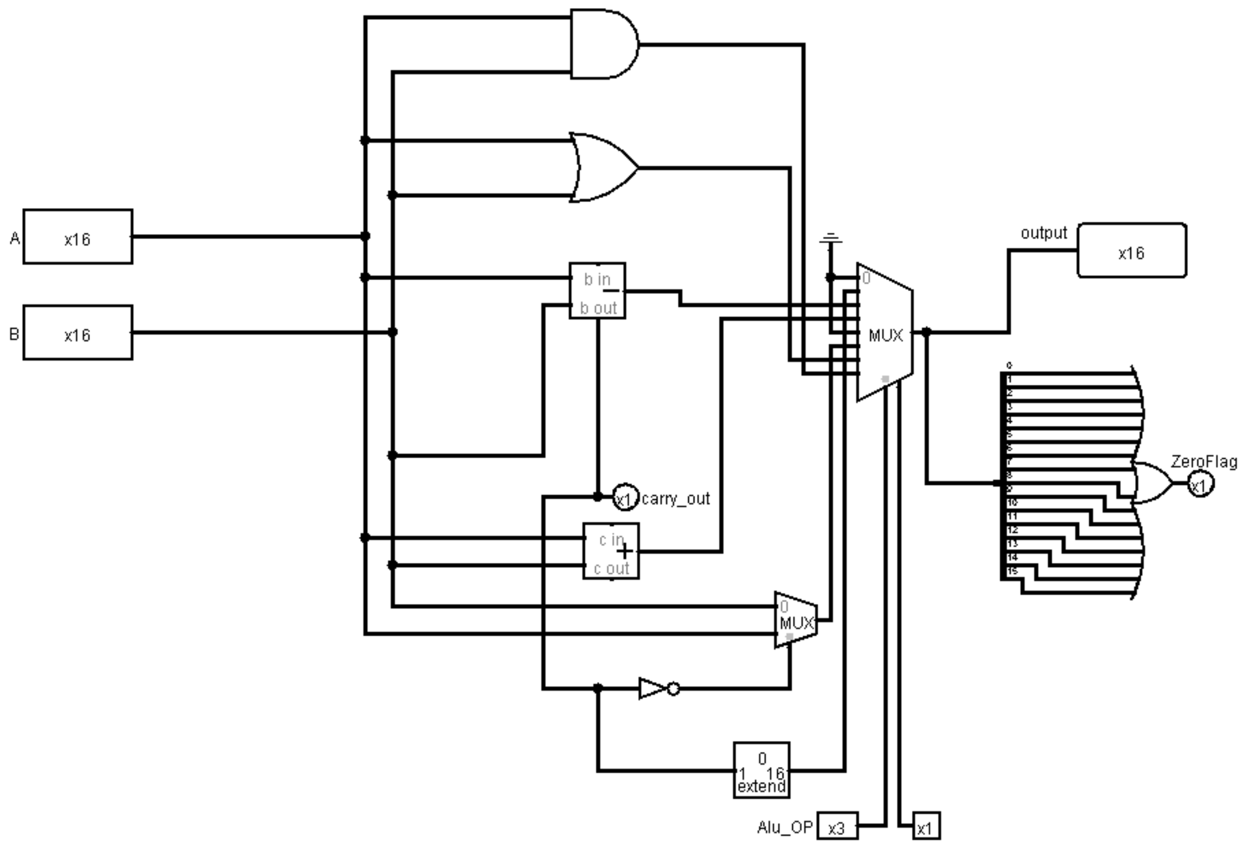
- Arithmetic and Logical Unit (ALU)
- Control Unit
- Register File
- Memory

Each one of these components will be used in the full Datapath along with the parts that are needed to enable pipelining and its hazard detection unit, forwarding unit, buffers between the stages and pc controllers.
The design will include five stages: Fetching, Decoding, ALU, Memory and the Write Back stage (Figure below show the whole system)



ALU

The ALU has six operations: AND, OR, CAS, SLT, ADD & SUB. It also has two flags: the SLT flag, and the carry_out flag. A MUX is used to multiplex the needed operation according to the 3-bit alu_op segment, with the alu_en bit, which enables the MUX therefore is used an enable for the ALU. Both the sources and the output are 16-bit registers.
The following shows our design of the ALU:



AND → just and between the 2 inputs A&B and the output is the input (111) in MUX

OR → just or between the 2 inputs A&B and the output in the input (110) in MUX

CAS → since CAS the MAX between the two inputs so just sub between them if the carry_out= 1 so i Mux 2x1 enable the input (1) = B and the output of mux 2x1 become input (101) in MUX and so on.

SLT → since SLT (Reg(Rs) < Reg(Rt)) so the same as CAS instruction if A>B then output and SLT flag 0 else 1 , if carry_out = 1 its mean A<B so take the carry flag and put them in extender to become 16 bit then the output of extender input of MUX in (001)

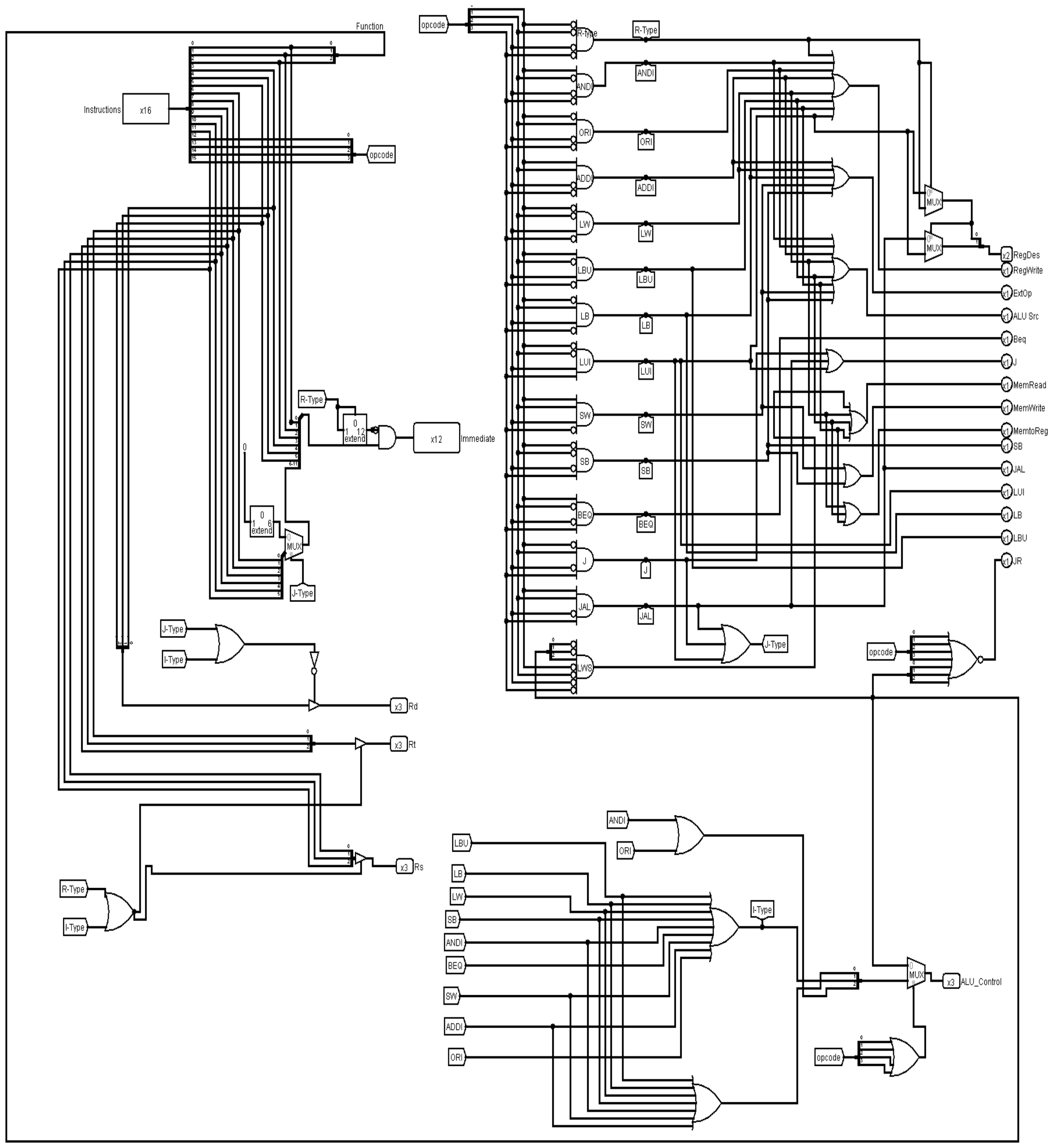
ADD→just summation the 16 bit of A and B and the output of Adder is the input (011) in MUX

SUB → just add subtractor and the output is the input of (010) in MUX .

Control unit

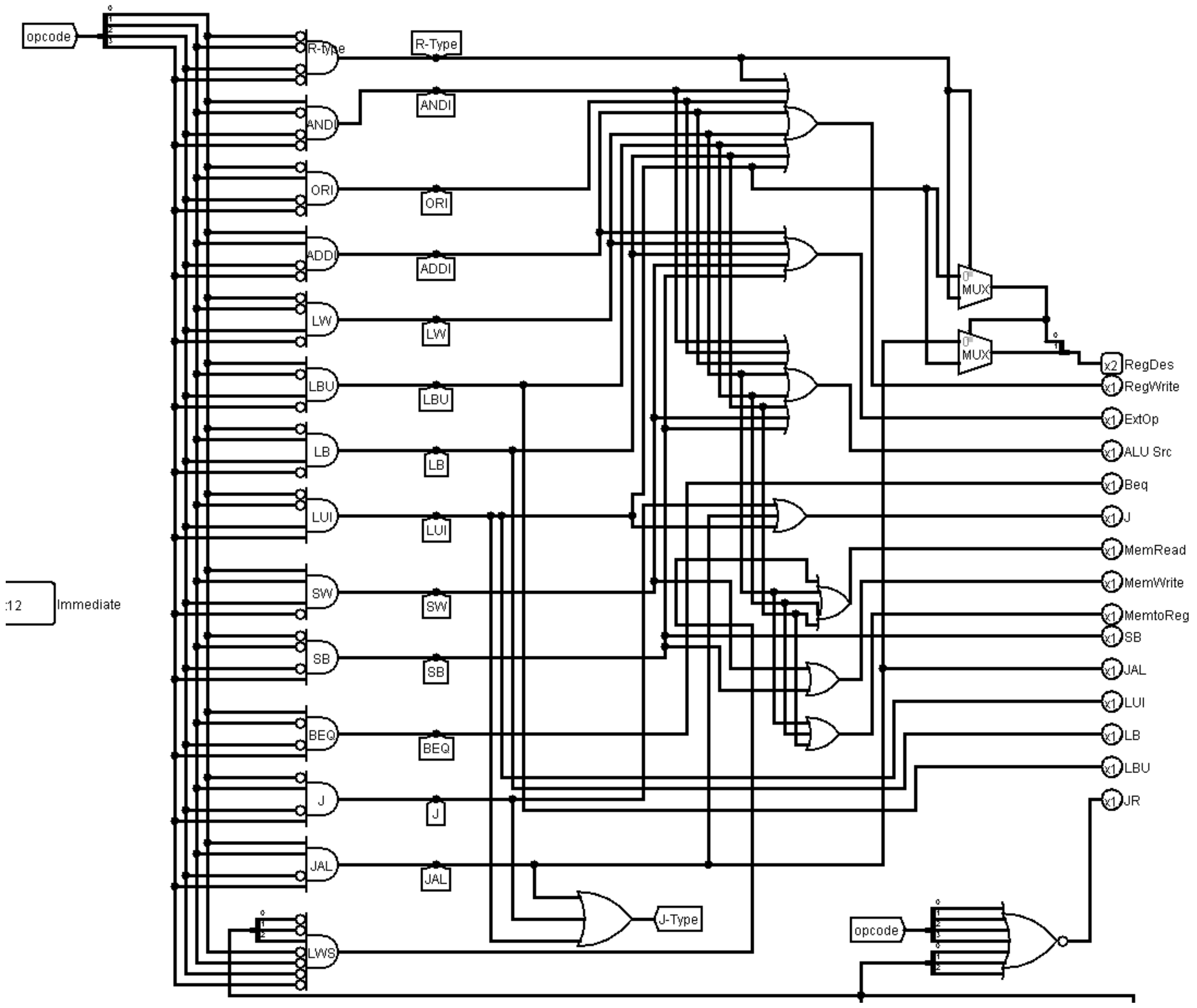
Op	RegDst	RegWrite	ExtOp	ALU Src	Beq	J	MemRead	MemWrite	MemtoReg	Sb
AND	1-Rd	1	X	0-Bus B	0	0	0	0	0	0
OR	1-Rd	1	X	0-Bus B	0	0	0	0	0	0
CAS	1-Rd	1	X	0-Bus B	0	0	0	0	0	0
LWS	1-Rd	1	X	0-Bus B	0	0	0	0	0	0
ADD	1-Rd	1	X	0-Bus B	0	0	0	0	0	0
SUB	1-Rd	1	X	0-Bus B	0	0	0	0	0	0
SLT	1-Rd	1	X	0-Bus B	0	0	0	0	0	0
JR	1-Rd	X	X	0-Bus B	0	0	0	0	0	0
ANDI	0-Rt	1	0	1	0	0	0	0	0	0
ORI	0	1	0	1	0	0	0	0	0	0
ADDI	0	1	1	1	0	0	0	0	0	0
LW	0	1	1	1	0	0	1	0	1	0
LBU	0	1	0	1	0	0	1	0	1	0
LB	0	1	1	1	0	0	1	0	1	0
SW	X	0	1	1	0	0	0	1	X	0
SB	X	0	1	1	0	0	0	1	x	1
BEQ	X	0	X	0	1	0	0	0	X	0
J	X	0	X	X	0	1	0	0	X	0
JAL	10-R7	0	x	x	0	1	0	0	x	0
LUI	11-R1	1	X	x	0	1	0	0	0	0

The whole block of control units implemented as follow:



About the Block:

At first implement control unit based on the 4 bit-opcode of instructions that given as follow:



Notice that we have a many of And gate having 4 bit input and 1 output and it implements and working depending of the OPCODE, let take an example about this (instructions R-type) Opcode = 000 so the first 4 bit of and gate inverted, Another example, (JAL) Opcode = 1011 so bit #3 was inverted and so on.

How to enable opcodes?

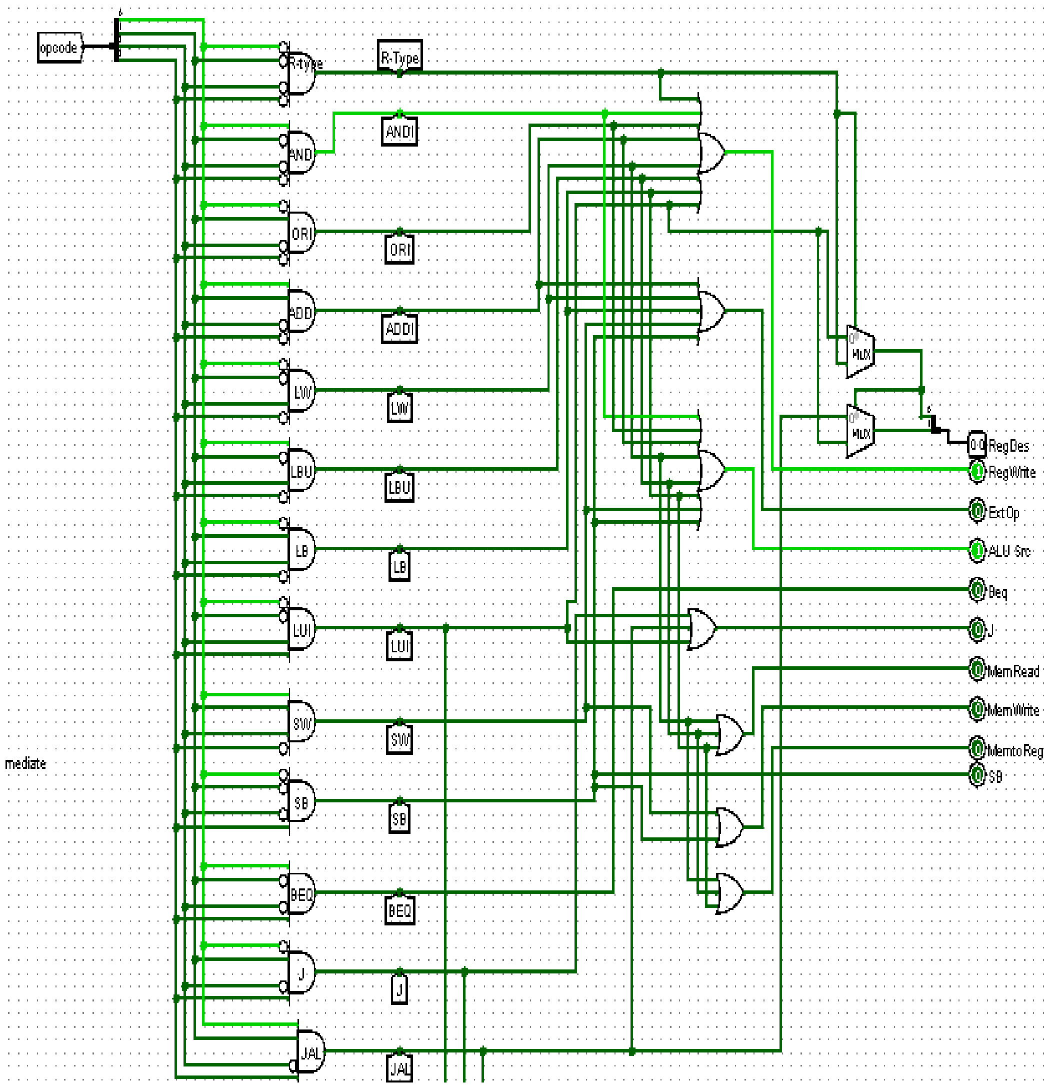
The main idea about it the truth table of control unit let see an example to discuss it and the other opcodes with the same step was implemented.

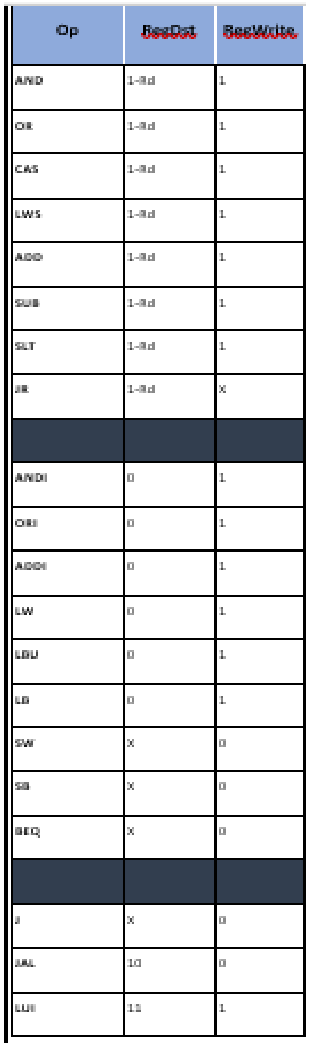
<u>RegWrite</u>

Writing of destination register. output of 9 bit OR gate it became 1 for all R-type instructions and in ANDI, ORI, ADDI, LW, LBU, LB and LUI, like in the table below

Now an example <u>of Applying ANDI instruction in control unit</u>

When the opcode 0001 (ANDI) just RegWrite and AluSrc enable (table at page 6)





Op	RegDst	RegWrite
AND	1-Rd	1
OR	1-Rd	1
CAS	1-Rd	1
LWS	1-Rd	1
ADD	1-Rd	1
SUB	1-Rd	1
SLT	1-Rd	1
JR	1-Rd	X
ANDI	0	1
ORI	0	1
ADDI	0	1
LW	0	1
LBU	0	1
LB	0	1
SW	X	0
SB	X	0
BEQ	X	0
J	X	0
JAL	10	0
LUI	11	1

How to enable ALUCtrl?

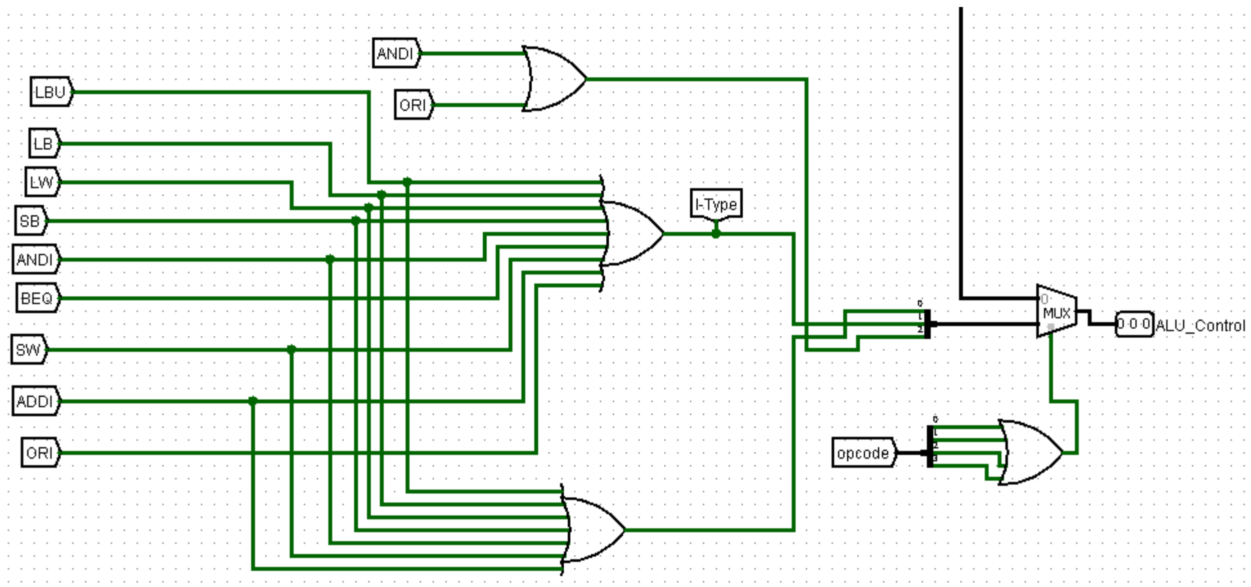


Figure above shows the ALU CTRL depends on the table below (this table like table in slide 51 of the course).

Op^6	Fun^6	AluOp^6	Encoding
R-type	ADD	ADD	011
R-type	SUB	SUB	010
R-type	AND	AND	111
R-type	OR	OR	110
R-type	SLT	SLT	001
ADDI	X	ADD	011
ANDI	X	AND	111
ORI	X	OR	110
LW	X	ADD	011
LBU	X	ADD	011
LB	X	ADD	011
SW	X	ADD	011
SB	X	ADD	011
BEQ	X	SUB	010
J-type	X	X	X

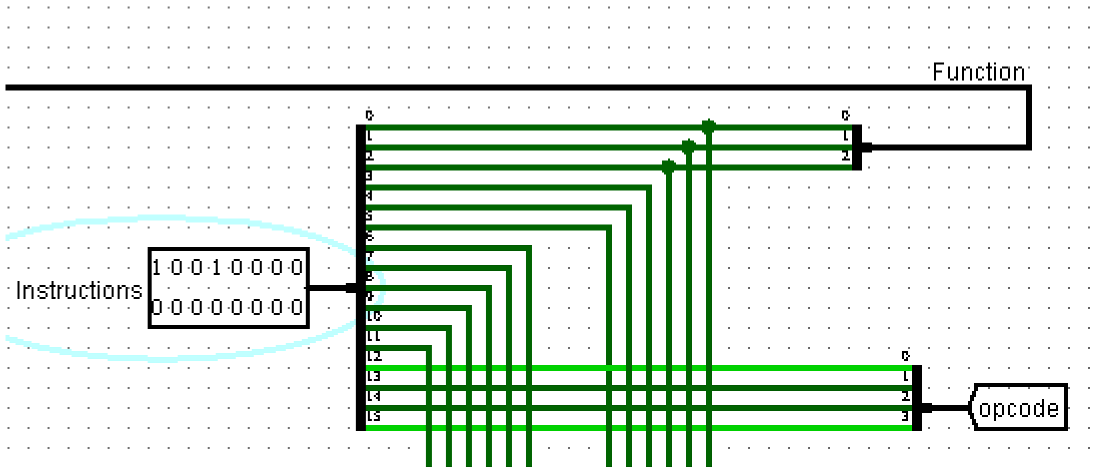
Figure have 3 or gate to enable I-type AlU CTRL and 1 or gate to enable R-type by his function of 3 bit ,

The first between ANDI and ORI because just 1-bit diff.

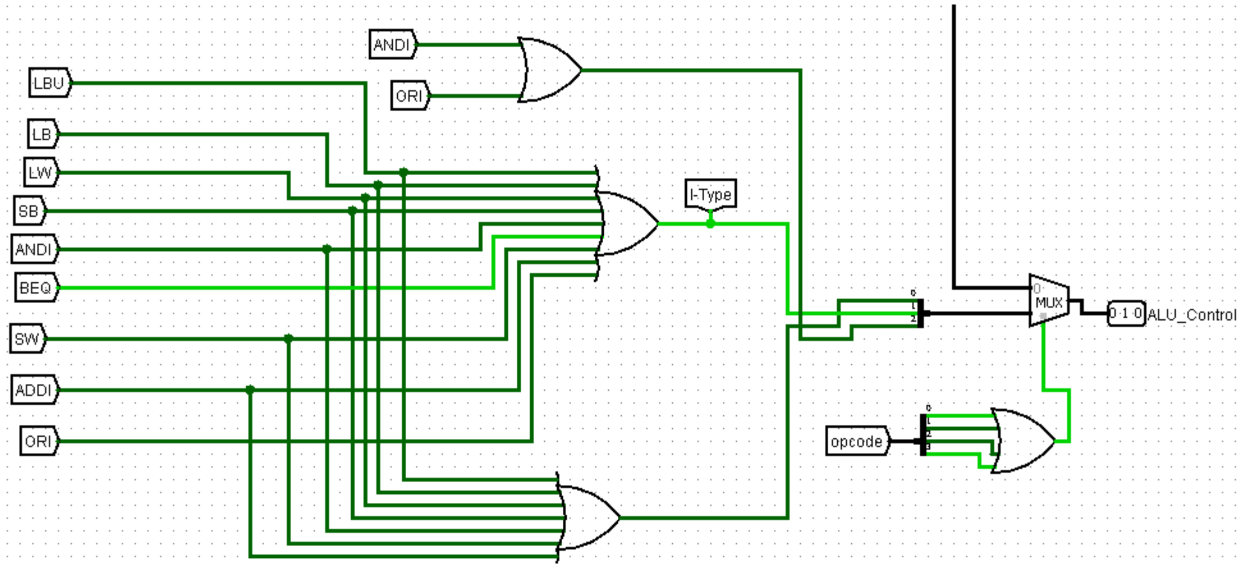
The second between all I-type instructions except ORI and BEQ because 2-bit diff.

Third between all I-type.

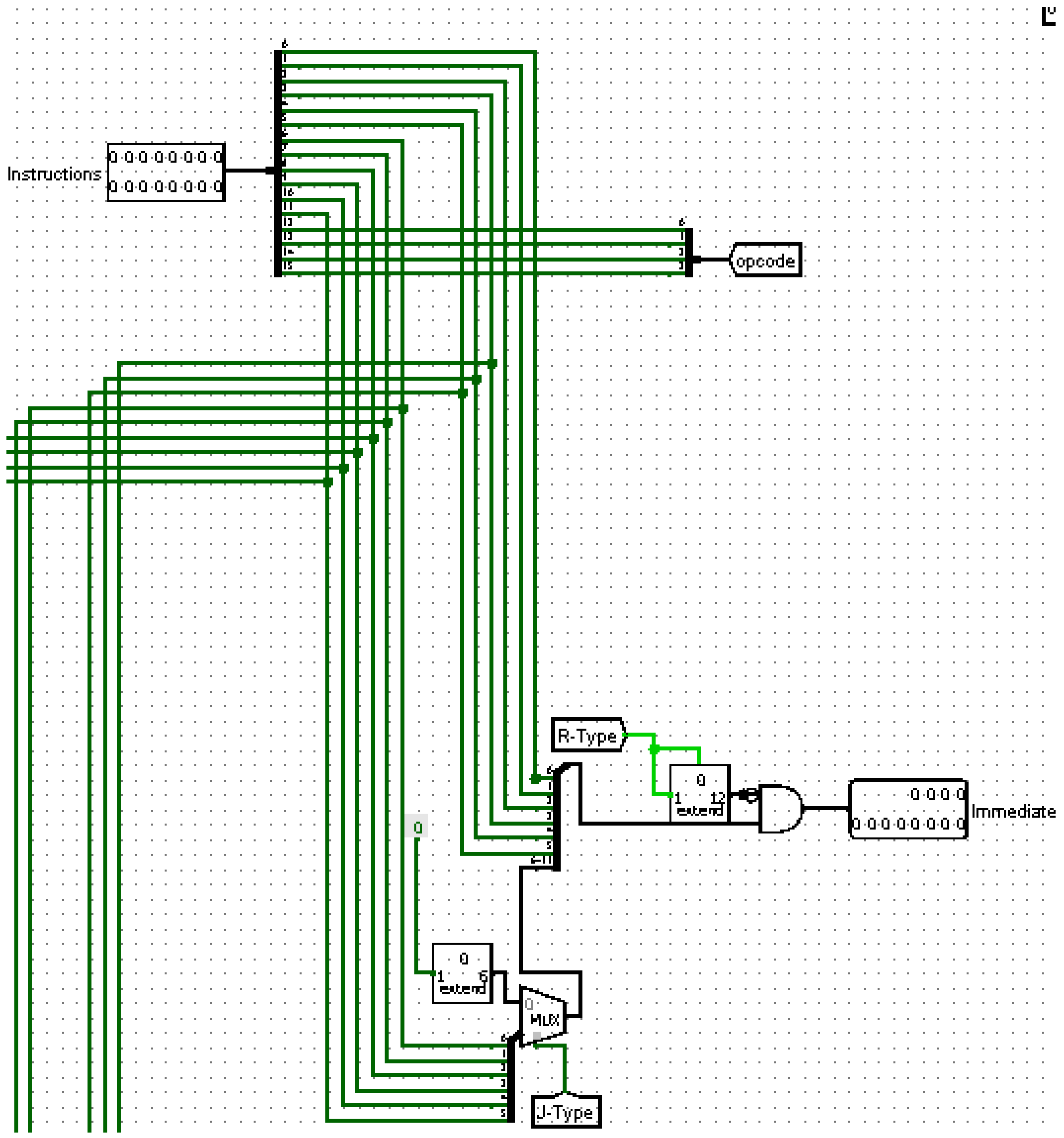
An example BEQ:



Since BEQ the Alu op = SUB so the encoding = 010 and this figure show it:



How to enable Immediate?

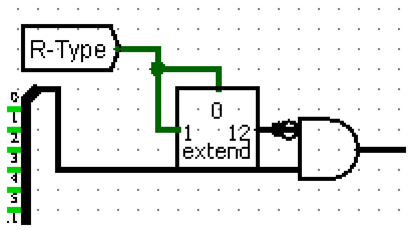


Notice that the first 4-bit in instruction for opcode and 12 bits immediate.

Explain it: we know the immediate for I-type and J-type and imm^{6} for I-type and imm^{12} for J-type

So if the opcode for j-type it means 12 bit immediate so take the 12 bit without any change but the problem in I-type because I want just 6 bit imm and the others I don't want them!

So, in I-type:

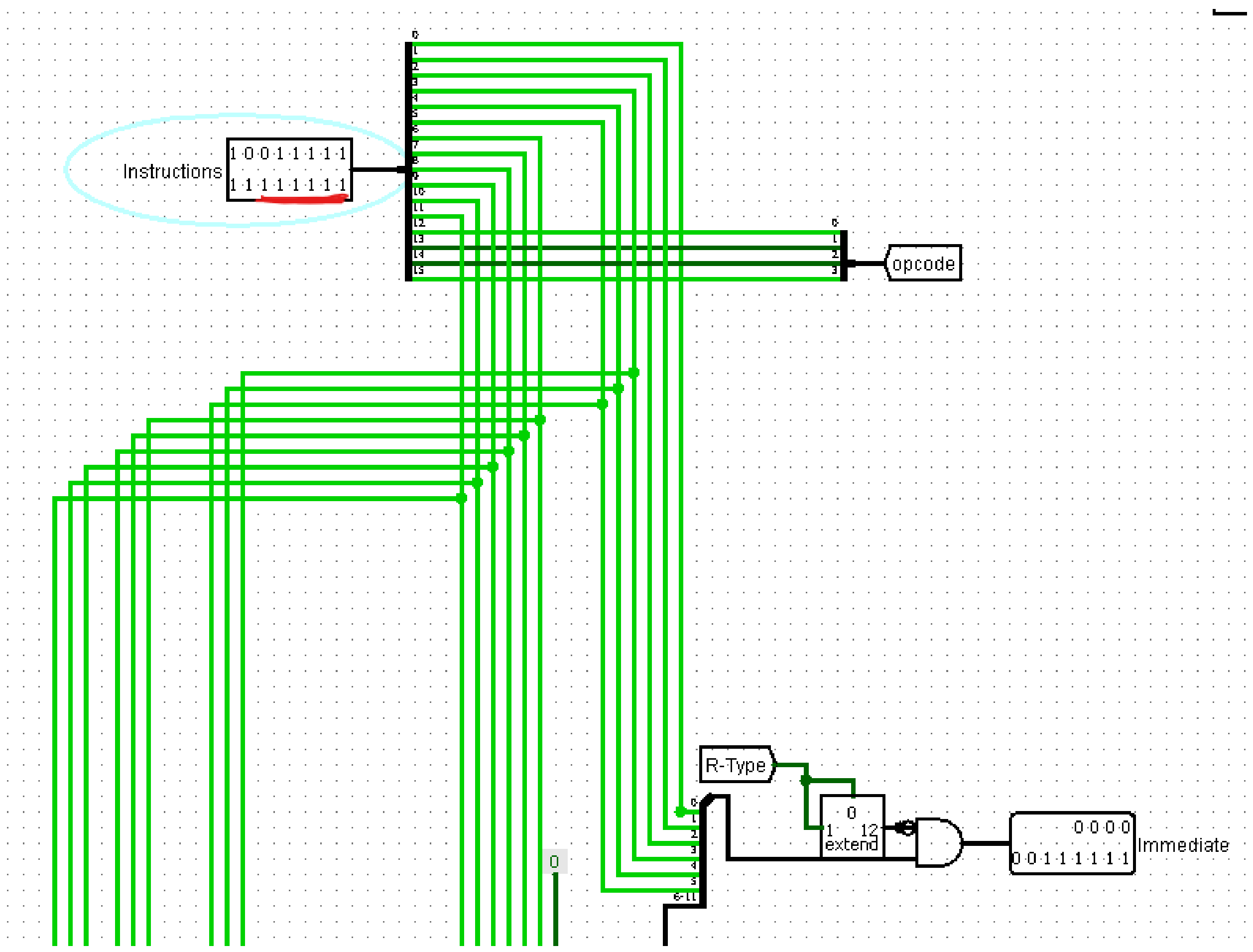


AND gate between the 6-bit of immediate and 12-bit

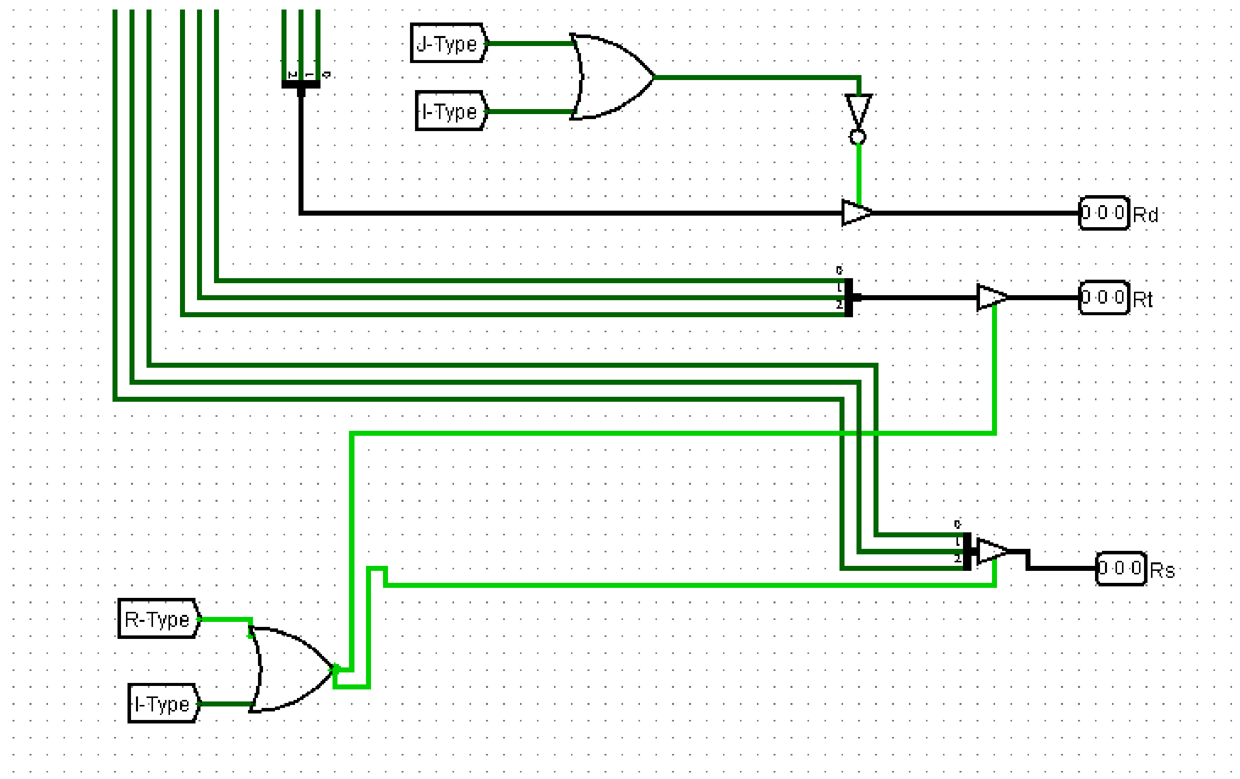
R-type because the R-type opcode = 0000 and

When extend it to 12 bit 000000000000 And with the

First 6-bit immediate it just shows 6-bit immediate; the figure below shows an example of BEQ with op code 1001



How to enable Rs,Rd,Rt?



At first, there is a format for the set of instruction

R-type format

4-bit opcode (Op), 3-bit register numbers (Rs, Rt, and Rd), and 3-bit function field (funct)

Op^4	Rs^3	Rt^3	Rd^3	$funct^3$

I-type format

4-bit opcode (Op), 3-bit register number (Rs and Rt), and 6-bit signed immediate constant

Op^4	Rs^3	Rt^3	$Immediate^6$

J-type format

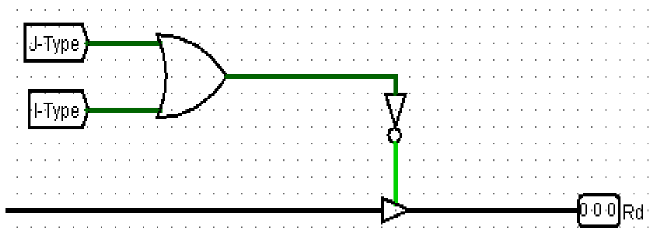
4-bit opcode (Op) and 12-bit immediate constant

Op^4	$Immediate^{12}$

Rs and Rt exist in R-type and in I-type but Rd just only in the R-type

<u>Explain:</u>

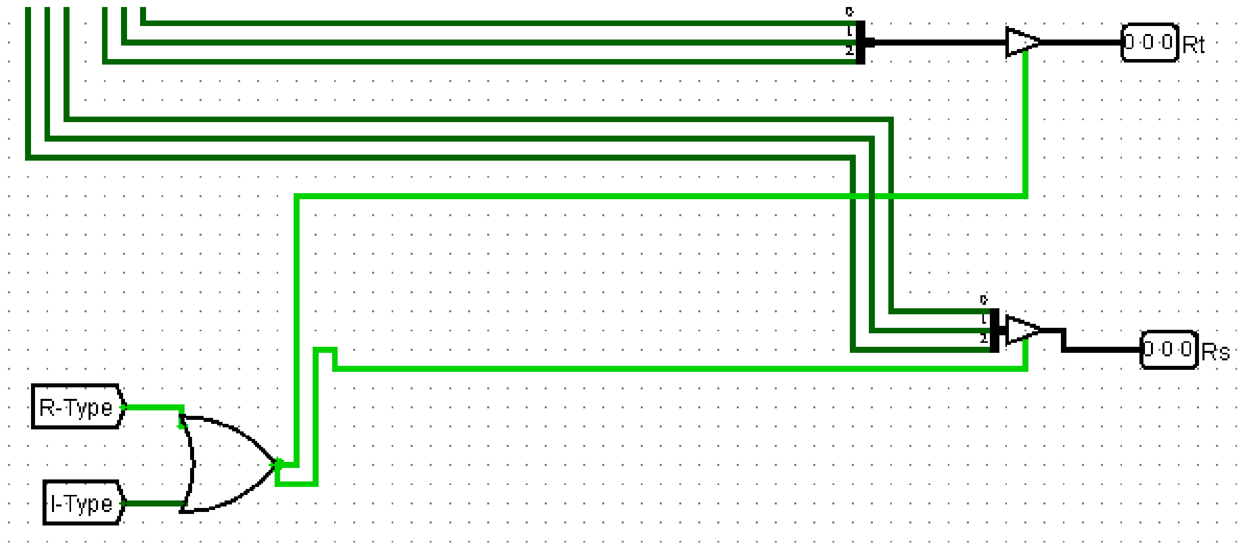
Rd→ first 3 bit for function and (0,1,2) and 3,4 and 5 for Rd in R-type so we put 2 buffers work like a switch that if the bit 3,4 and 5 so enable Rd other than that to disable



Rt→ in R-type and I-type the bits number is 6,7 and 8 so always enable it if in J-type or I-type since Rd doesn't enable if J-type or I-type

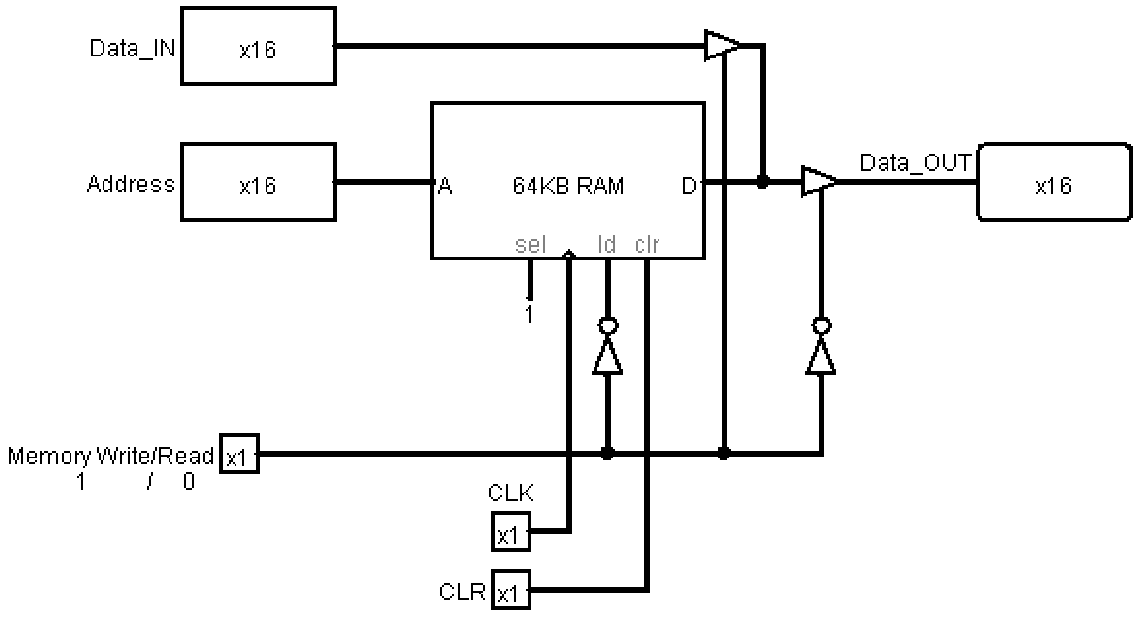
Rs→in R-type and I-type the bits number 9,10,11

The figure below explains them



Memory

For the memory, we used the built-in RAM module from Logisim, along with 2 tri-state buffers to control the input and output of the module, since the RAM units have only one port for both input and output. The unit is selected, ie. produces an output or takes input, by **ORing the mem_RD with the mem_WR .** mem_WR signal also controls the flow of data as well as the ld (load) control of the module.

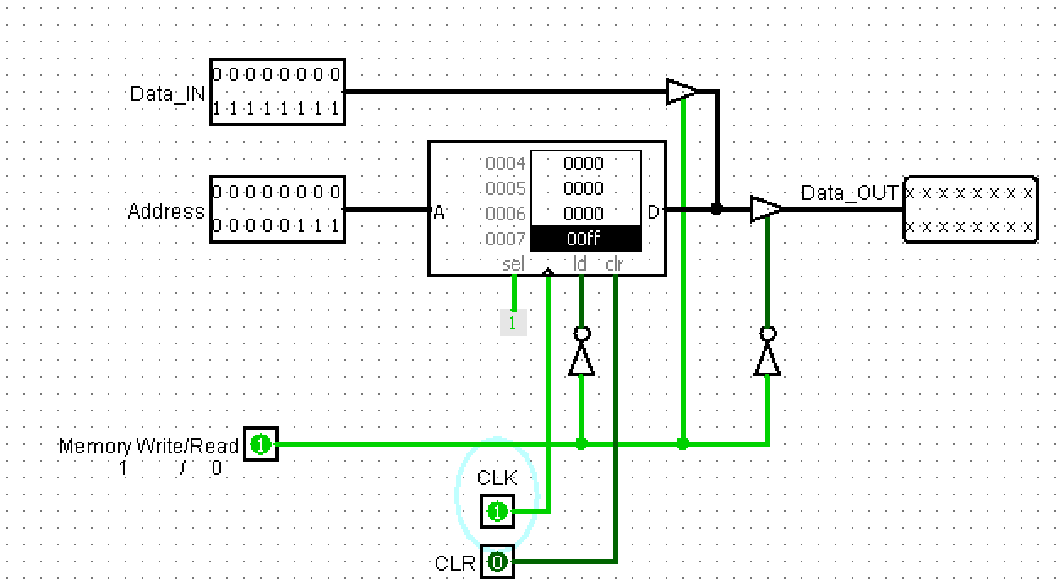


About the Block:

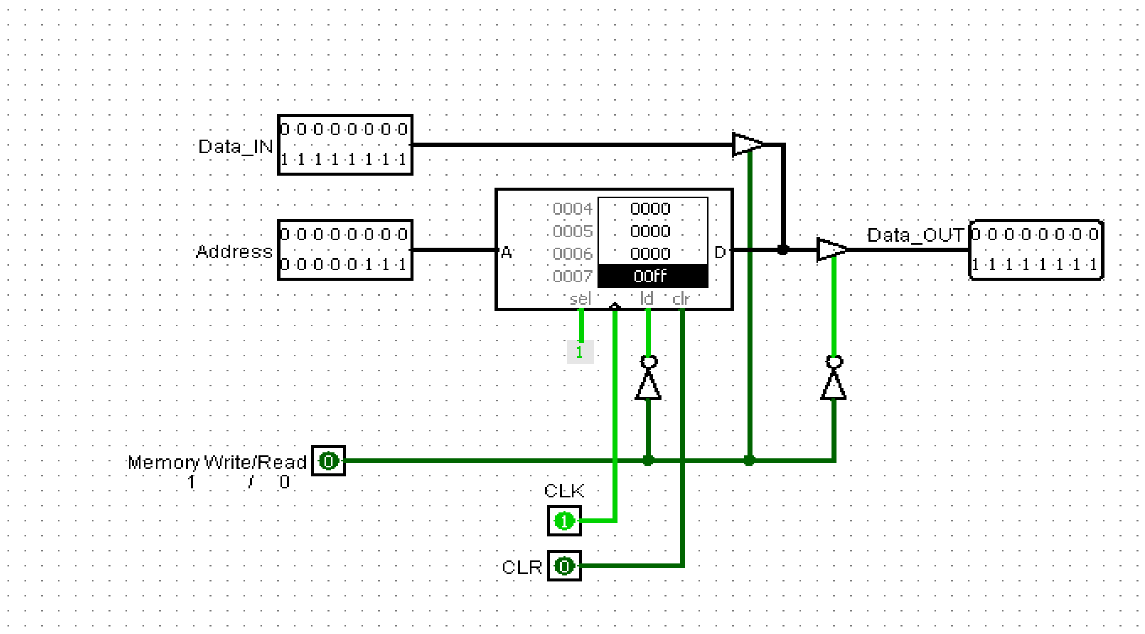
"A" chooses which address will be accessed (if any). "sel" essentially determines whether or not the RAM module is active (if "sel" is low, "D" is undefined). The clock input provides synchronization for memory writes. "ld" determines whether or not memory is being read or written. If "ld" is high, then "D" will be driven with the contents of memory at address "A". "clr" will instantly set all contents of memory to 0 if high. "D" acts as both data in and data out for this module. This means that you must use a controlled buffer on the input of "D" to prevent conflicts between data being driven in and the contents of memory

How these Work?

when the Memory write/Read = 1 that's mean turn off load (ld) and turn on the buffer that comes from data in and turn off the buffer that going to Dat_out, briefly the data write in the memory like in the figure below

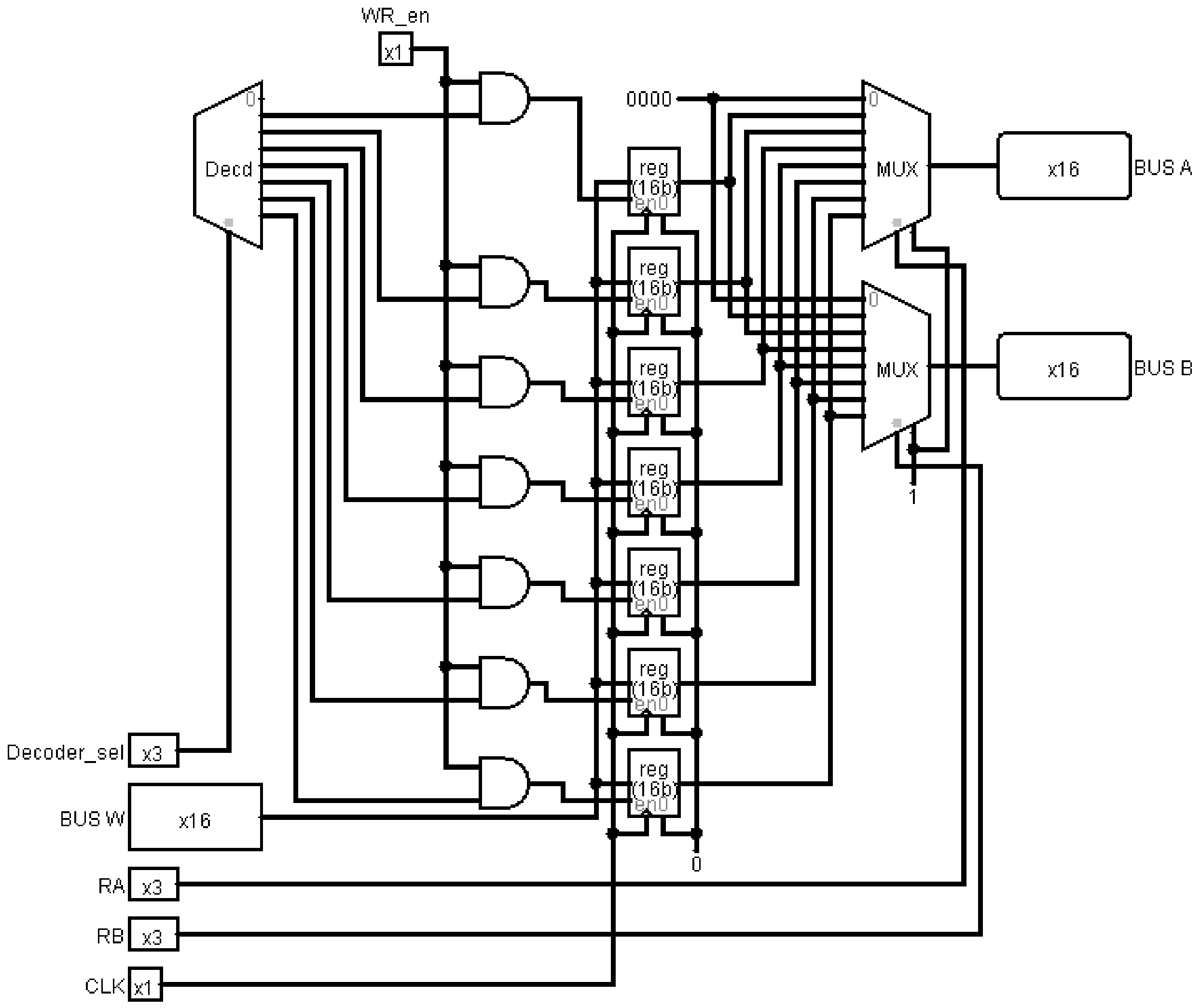


In the other hand, when the Memory write/Read = 0 that's mean turn on load (ld) and turn off the buffer that comes from data in and turn on the buffer that going to Dat_out, briefly the data write in the memory like in the figure below



Register

We approached the register file with a simple basic design (from the slides), controlling the output by multiplexing the register values and using a WR_en to enable the write of new data based on the address of the destination *and* the *reg_WR* control signal.

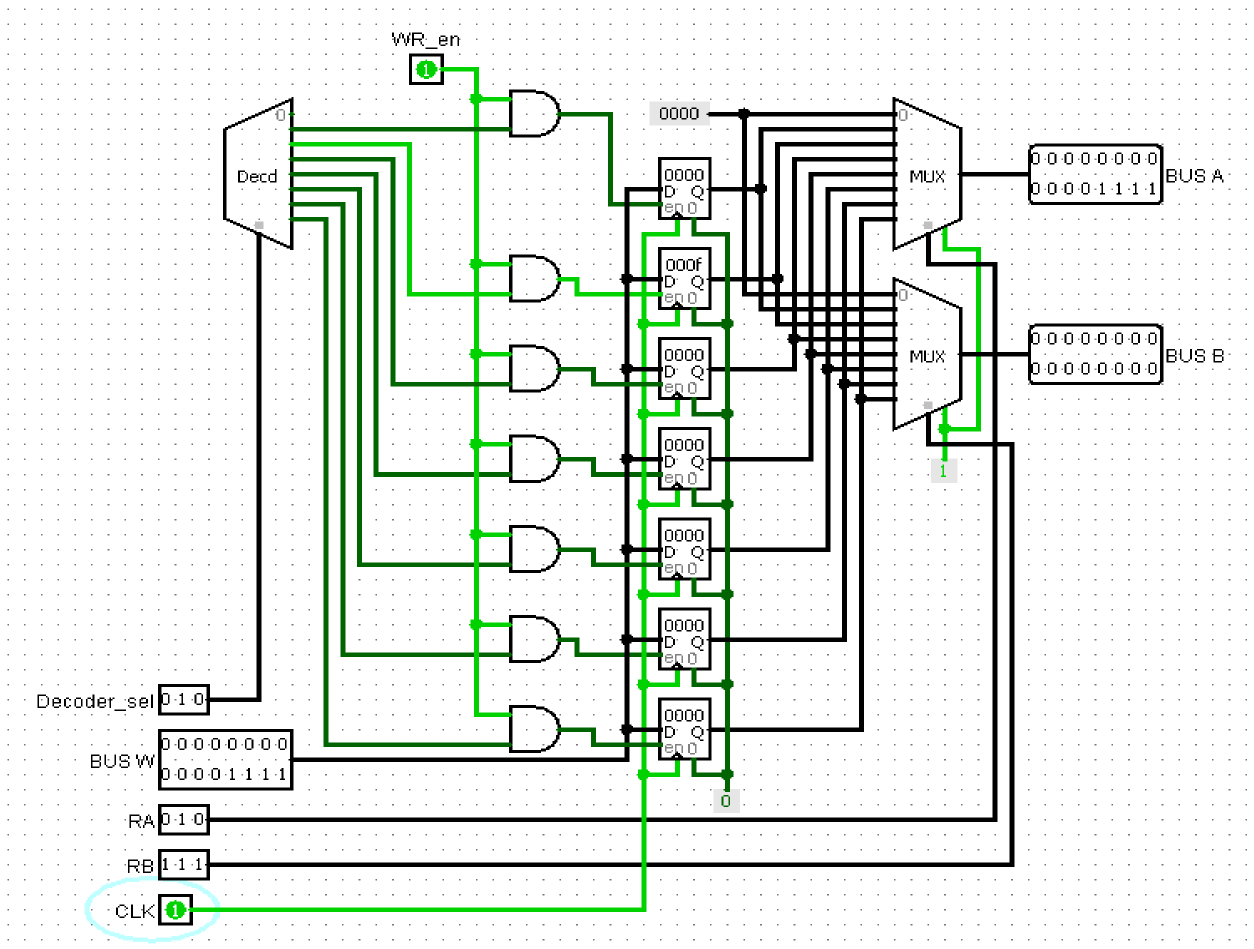


About the Block:

- Decoder_sel →to write data in register (R1-R7)
- BUS W→data input (data write in the register)
- RA→which means select register the BUS A that read from
- RB→ which means select register the BUS B that read from

How these Work?

Let take an example to explain it (show the figure below)



in the figure above in Decoder_sel (010) so write the data in register 2 and the BusW= 0xF since the RA select register 2 to read from it and the BUS Ato write the data in the output of the mux

and RB select the register 7 to read from it and the BUS B to write the data in the output of the mux

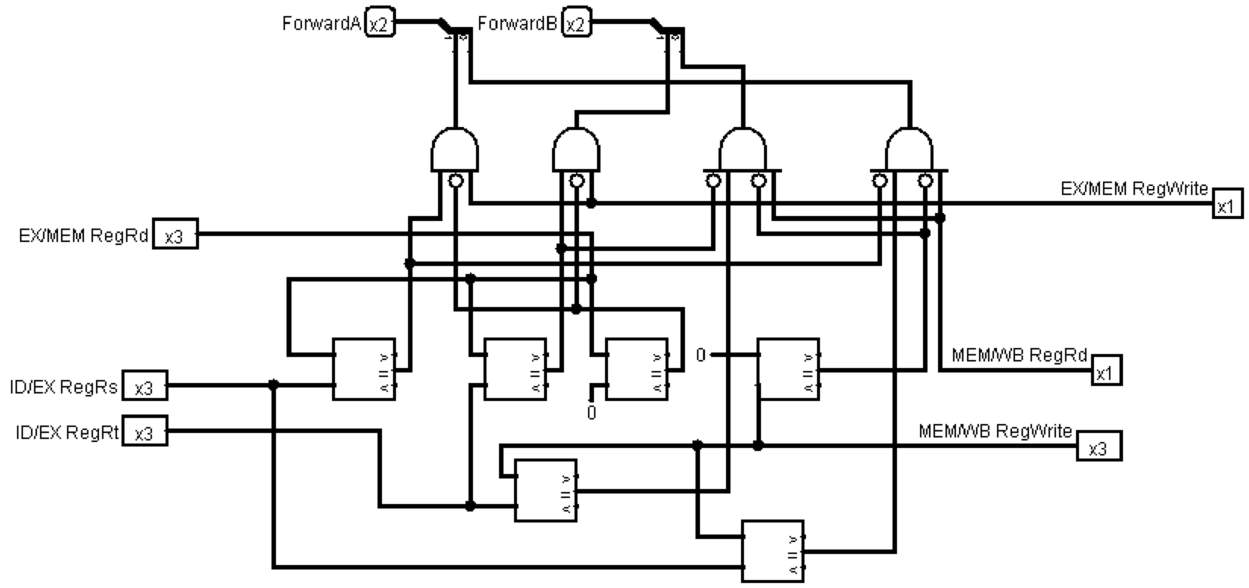
As we show BUS A write 0xf in binary because the RA select to read from register 2

and BUS B write 0x0 in binary because the RB select to read from register 7.

The Datapath also includes several blocks that are needed for pipelining like:

Forwarding Unit

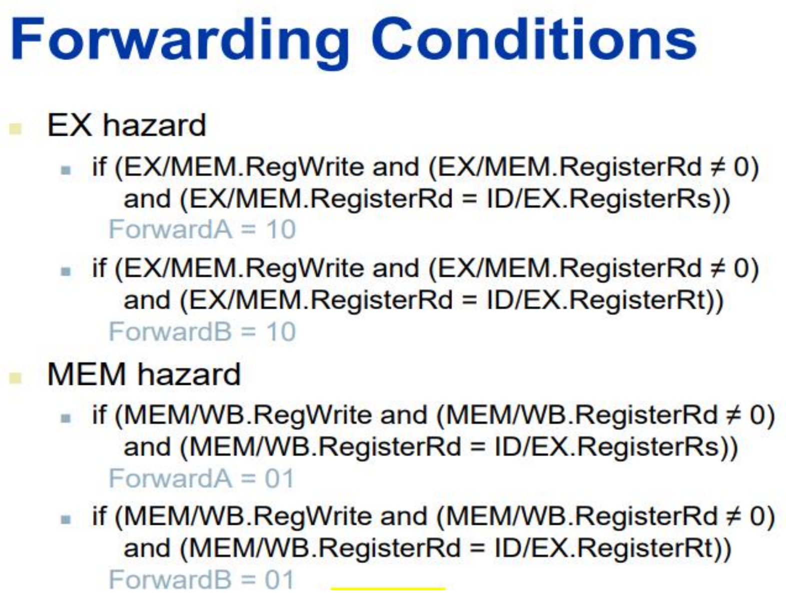
Implemented as follows:



How these Work?

this design was implemented as the Detect condition in Slides (page 140)

let take an example to show it

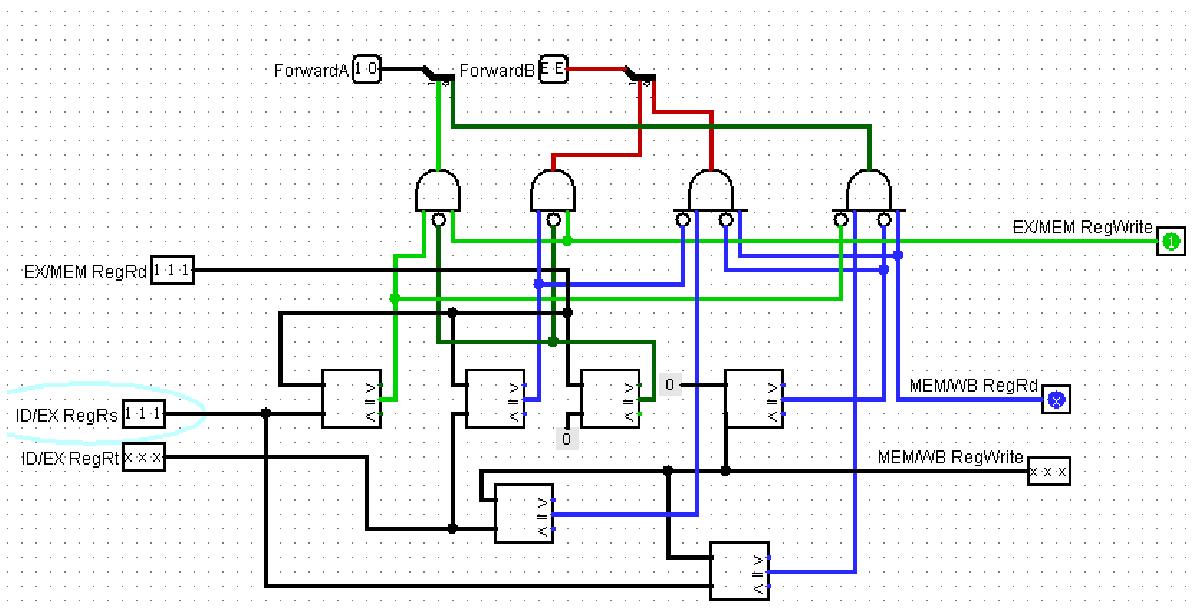


Forwarding Conditions

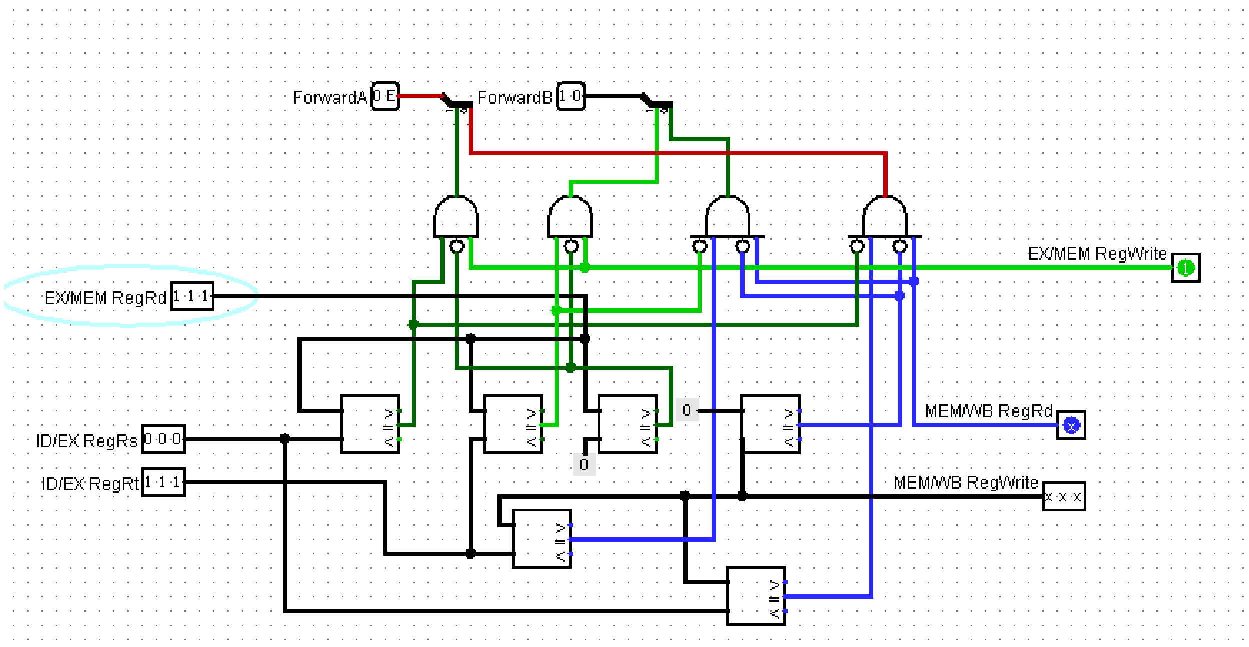
- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

At First let change Ex/Mem.regWrite = 1 and Ex/Mem.RegisterRd=!0 and Ex/mem.RegisterRd = ID/RX.registerRs and the result of ForwardA will Equal = 10

Show the figure below



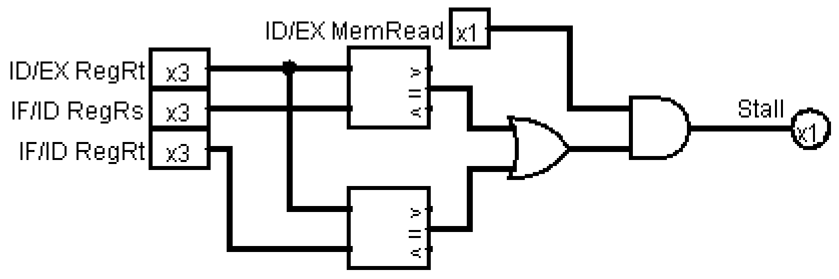
Same the condition but now if Ex/M.Rd = =ID/Ex.Rt also the same result



Hazard Detection

The data hazard unit is used to ensure the correctness of the result of the instructions execution flow. by creating artificial stalls that prevent an instruction from executing if its data isn't yet prepared.

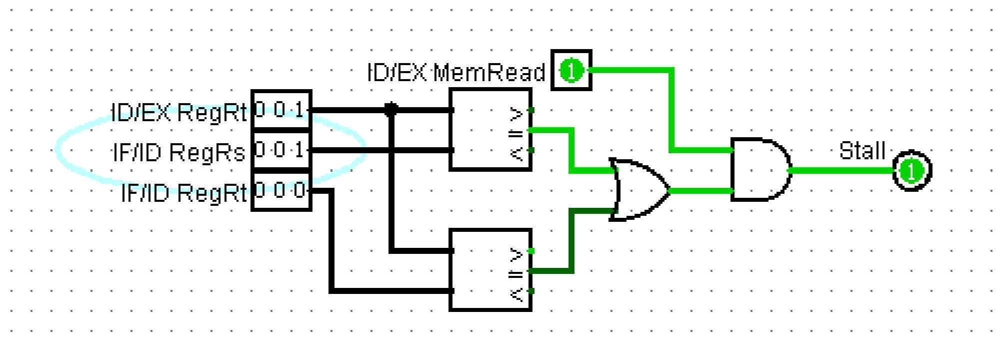
The following is our design for the data hazard unit:



How these Work?

- **Conditions for the hazard detection**
  - **If (ID/EX.MemRead and**
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    ((ID/EX.RegisterRt = IF/ID.RegisterRt)))
    stall the pipeline

Let apply the condition as doing in forwarding unit.



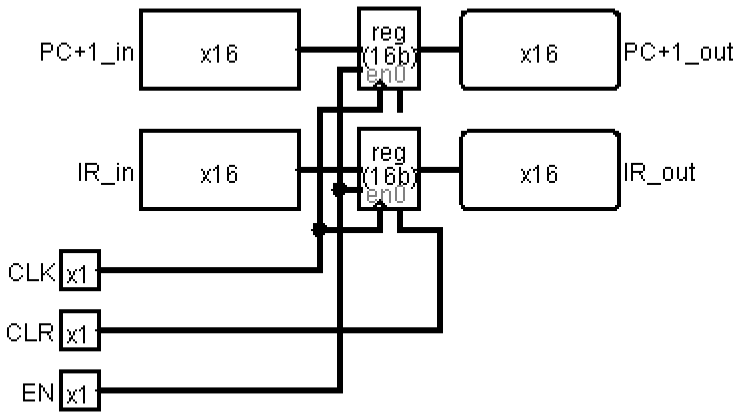
In this figure ID/EX.MemRead = 1 and one of the condition true **(ID/EX.RegisterRt = IF/ID.RegisterRs**)so there is a stall

Buffers

Four stages of falling-edge buffering registers also are implemented (Fetch, Decode, Execute, Memory). The reason these buffers are falling edge, in contrast to the rest of the sequential logic in the Datapath, is to allow data to be read between stages, this is essential in the second half of the cycle while writing in the first half of the cycle. and it's
implemented as follow:

1-Fetch/Decoded Buffer

The buffer implemented like in the figure below:



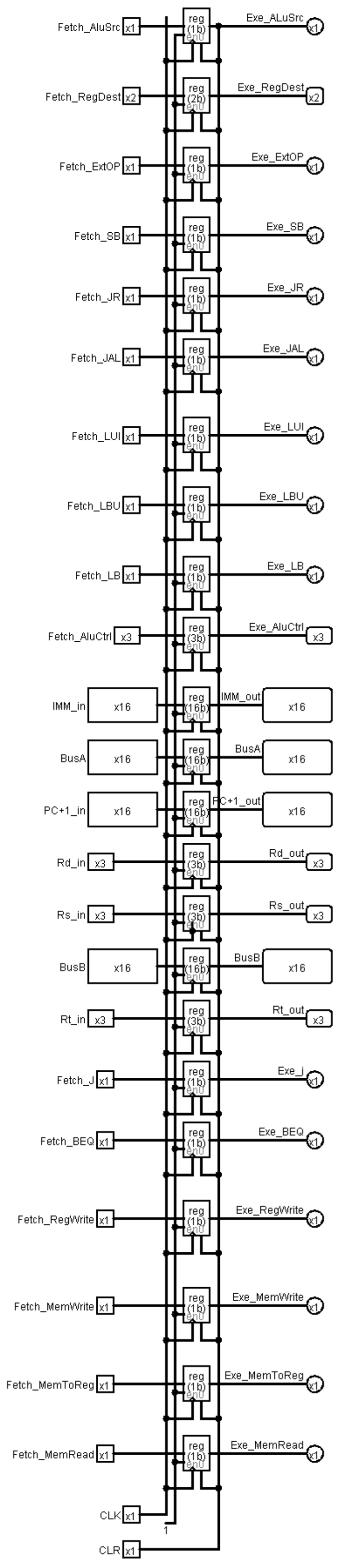
It to pass the instruction with incremented PC register from the fetching stage to the decoding stage.

2-Decode/Execute Buffer

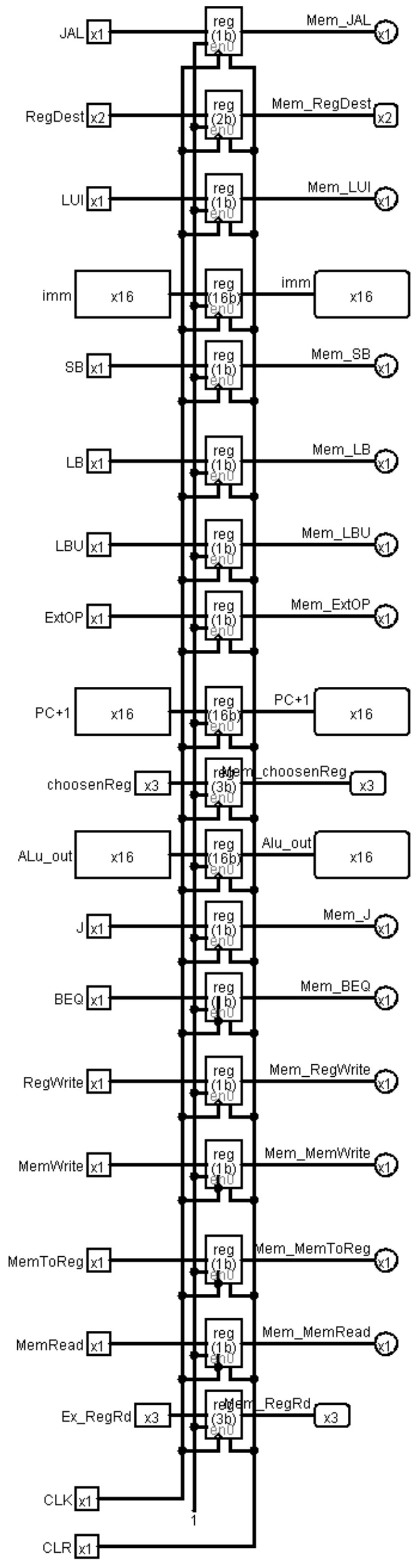
3-Execute/Memory

4-Memory/WB

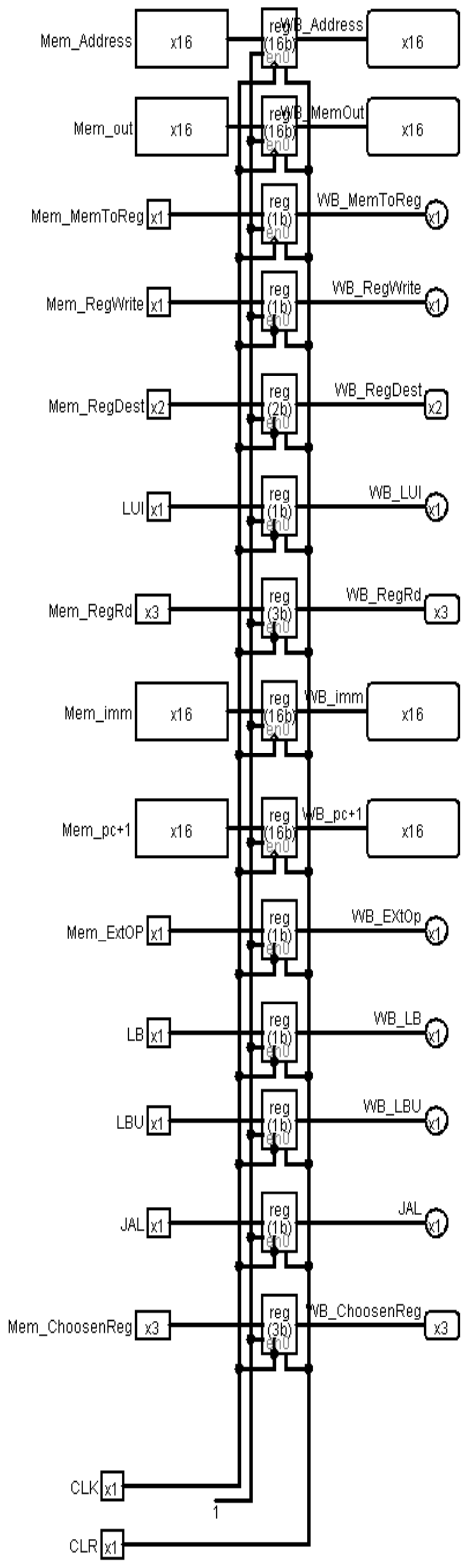
Decode/Execute



Execute/Memory

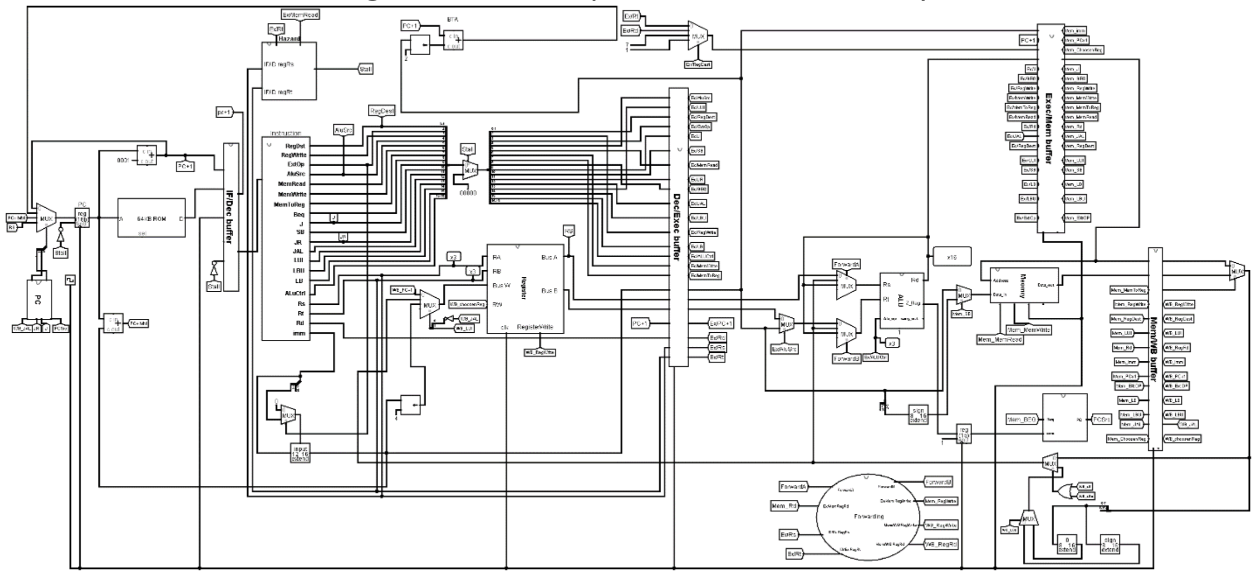


Memory/WB



Datapath

After that we connecting all described parts and the whole system became:

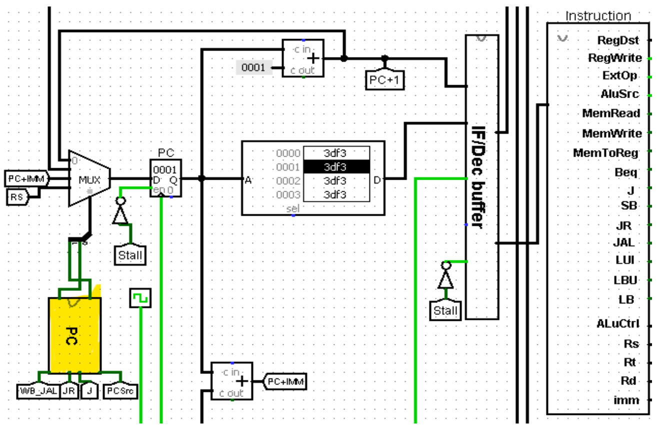


About this block and how to implement it?

At first, we will explain stage by stage to become easy to understand.

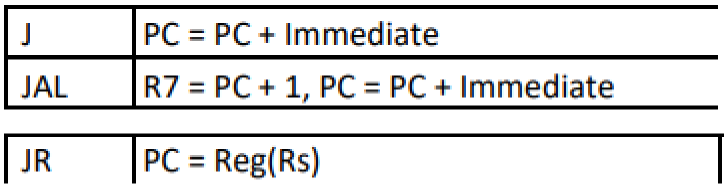
In this figure its start with implemented the **PC controller** as given in the table and in this project, there are different form of the PC so we need a Block (PC) this block work as a controller of PC

We implemented the block by this table



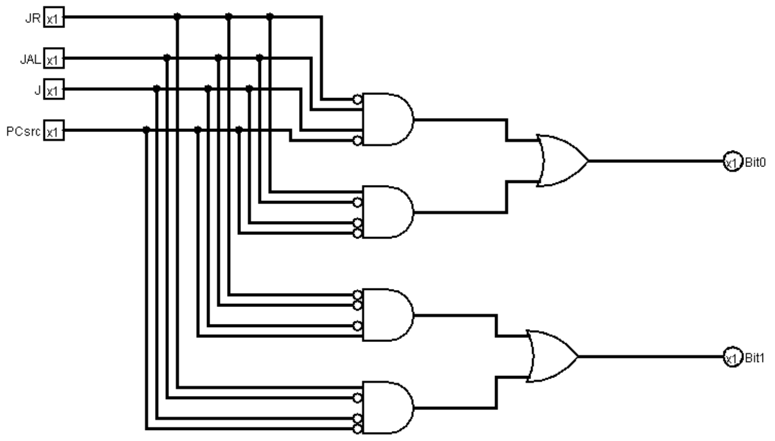
		JR	JAL	J	PCsrc	Bit 0	Bit 1
0	0	*0*	0	0	0	0	0
0	1	*0*	0	0	1	0	1
1	0	*0*	1	1	0	1	0
1	1	1	0	0	0	1	1

The mux has 4 input and 1 output and in the first input PC+1 in the second BTA then PC+IMM and Finally Rs this input taken from this given data

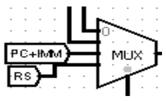


J	PC = PC + Immediate
JAL	R7 = PC + 1, PC = PC + Immediate
JR	PC = Reg(Rs)

This design was implemented to controller the pc by the table in above



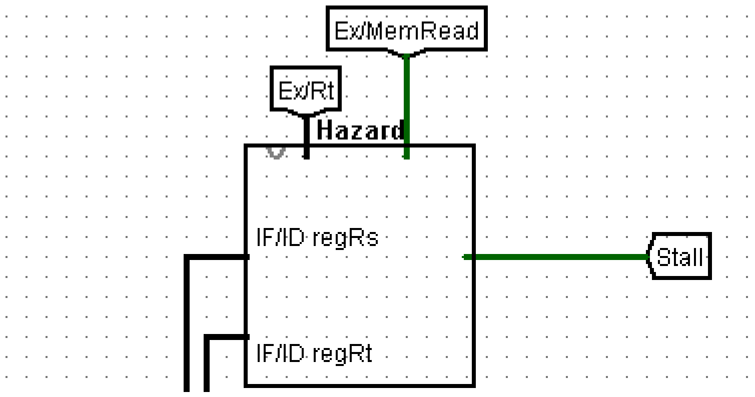
How this Figure implemented?

As we see in the mux  when the Bit 00 do not enable anything because it enables PC+1 as the default state and when it 01 enable PCsrc and when it 10 that's mean PC+IMM So enable J and JAL finally when 11 enable JR.

After we implemented the PC controller, we passed the instruction from fetch buffer to decode buffer

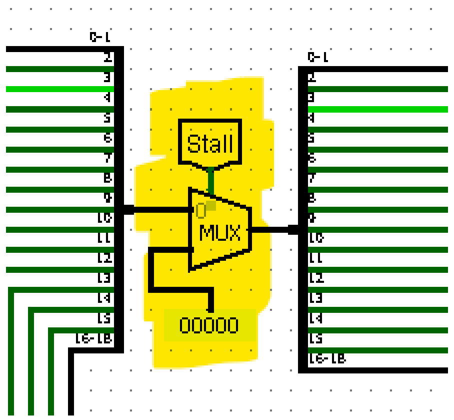
Hazard (STALL)

As we describe in the previous part of hazard it takes a 4 input and the output of hazard is the stall as in the figure below

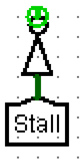


How it works?

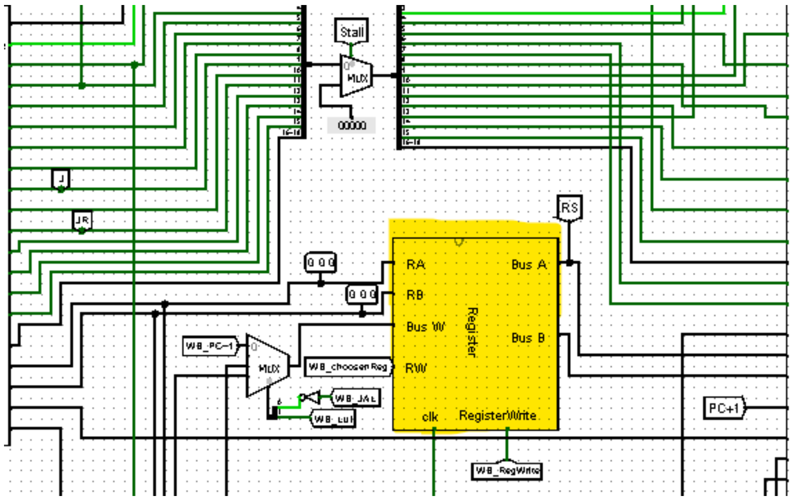
When the stall = 1 the control unit set to zero that doing by make a constant zero in the input 1 of mux as shown in the figure below



So, we can't Read/Write in Data Memory also disable IF/ID and the PC since there is a stall in the enable of both pc and the Fetch buffer and to disable it when the stall = 1 we connected stall with buffer to became zero to disable PC and IF/ID



Register File



AS in the figure above RA and RB are both connected with Rs and Rt from the IF/ID buffer and that as we explain in the Register file part Ra and Rb to choose which register we want to read to (BUS A or BUS B) and since the Bus W the input data and we have a 3 types of input data depends on the instructions that given:

Default → PC+1

JAL → R7 = PC + 1

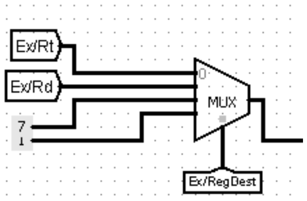
J → R1 = Immediate12 << 4

As we explain in the part of the register Rw in this block is a (Decoder_sel) so choose between 4 registers to which register write in and the 4 Register is:

- Rd for R-type
- Rt for I-type
- R7 for JAL
- R1 for LUI

} depend on RegDest (show table page 6)

The implementation of this mux

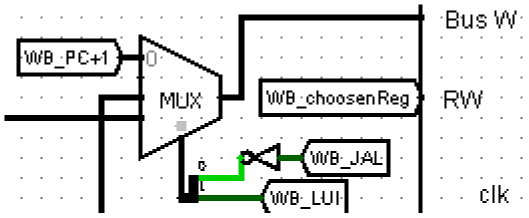


Since if the RegDest 00 will enable Rt for I-type instructions

And when it 01 that's for R-type and when 10 that enable R7 for Jal and when 11 that enable R1 for LUI instruction.

After we choose a RW (decoder_sel) **how to choose BUS W** (data input)**?**

Data input as we explain in the previous page have 3 types of instructions so at first, we implemented a mux that have 3 inputs as in the figure



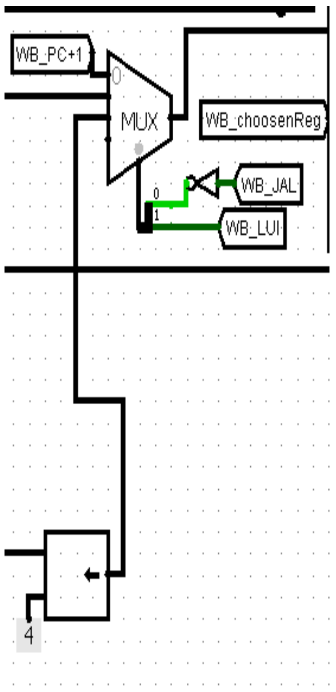
When we enable JAL, we want (R7 = PC+1) and when enable J (R1 =imm^{16}<<4)

explain the mux:

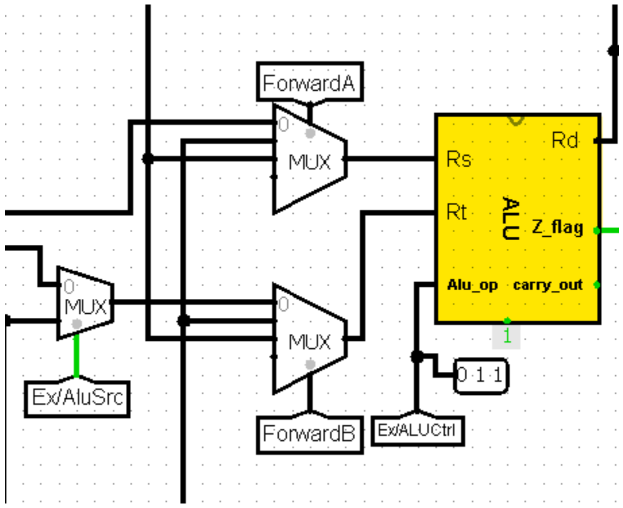
when we applied JAL instruction that's mean =1 and since there is an invertor in JAL so the Bit 0 of selection = 0 and since JAL applied LUI disable that's mean = 0

so, the Bit 1 also = 0 so the selection = 00 so the input 0 of mux will run so **PC+1** and that's for JAL

when applied LUI instruction that mean Bit 1 = 1 and JAL disable =0 and there is an invertor in JAL so became 1 so Bit 0 = 1, the selection =11 so input 3 of mux will run **imm^{16}<<4** like in the figure and the default when the both JAL and LUI 0's that mean the selection = 01 for other instruction will run(ADDI,ANDI,ORI,…..)



ALU

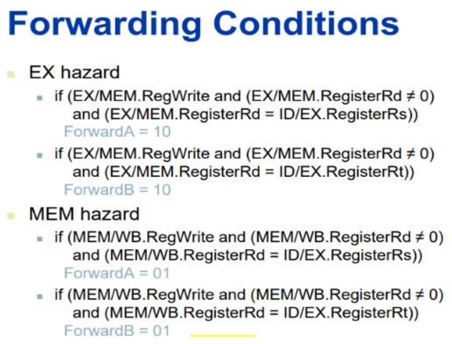


As shown in the figure and we explain this block in the previous part and its work successfully, it takes 2 input Rs and Rt of 16-bit and each this input takes the data depends if there is a forwarding or no so **how ALU works in Datapath?**

At first, we need to mux's for 2 data input

At default the input of Rs and Rt is output from Dec/Exec buffer Bus A and B respectively, so it's the <u>input 0</u> of the 2 mux's

When forward occurs that mean's as we explain in the forward part



Forwarding Conditions

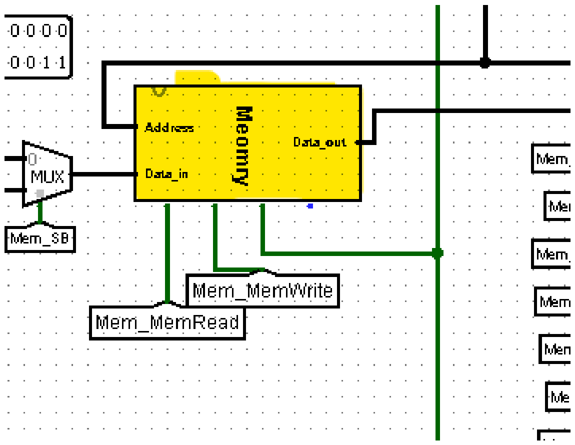
- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01

So, when <u>input 1</u> →is the WB_MEMoutput

And when <u>input 2</u> → is the ALU_output

And the second MUX work like the first mux 😊

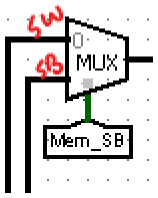
Memory



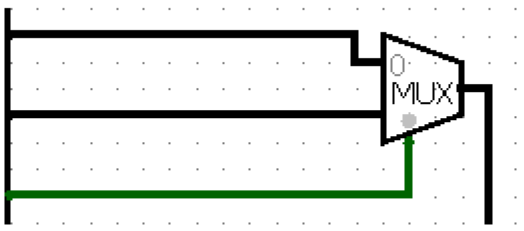
As we explain the part of memory it takes to input Address and Data input and the 16-bit address is the output of ALU after pass the Exec/Mem buffer and the second input is the 16-bit data input that the data will be written in it.

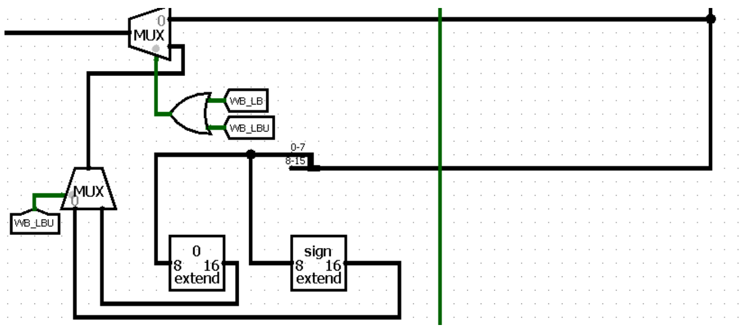
The instruction that’s write on the Memory

- SB Mem (Reg(Rs) + Imm6) = Reg(Rt [7:0])
  In this instruction just we want the first 8 bit from Rt and save it on the memory and we made it by a splitter that takes first 8 bit and put a mux if Mem_SB = 1 that’s mean we applied SB instruction so enable the input 1 and the data input is the 8 -bit of Rt store as Sign Extend on RAM
- SW Mem(Reg(Rs) + Imm6) = Reg(Rt)
  This instruction means that store Rt in RAM and this enable when SB = 0



Now how to enable LB and LBU in Datapath?

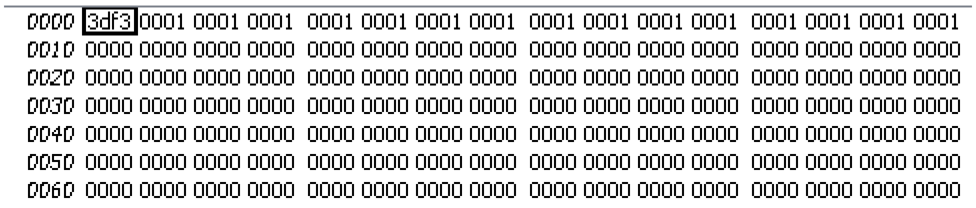
- LB Reg(Rt [7:0]) = Mem(Reg(Rs) + Imm6) Reg(Rt [15:8]) = sign bit
  This instruction mean take the input from the address memory (8-bit) and load to 8 bit Reg (Rt) look at the figure below
  

  the mux has 2 input the first is the address of the memory and the second DataOut of the memory and the selecter of the mux is MemToReg that's always 1 when the instruction is Load so the output is WB_MemOut and we want the first 8 bit of it so take it and pass them to sign extender
  

- LBU Reg(Rt [7:0]) = Mem(Reg(Rs) + Imm6) Reg(Rt [15:8]) = Zero bit
  Same as the LB instruction but zero extender instead of sign extender

Notice that the 2 extenders are connected with mux and the selector to mux LBU that's mean when LBU = 1 enable zero extender and when = 1 enable the sign extender for LB

Test



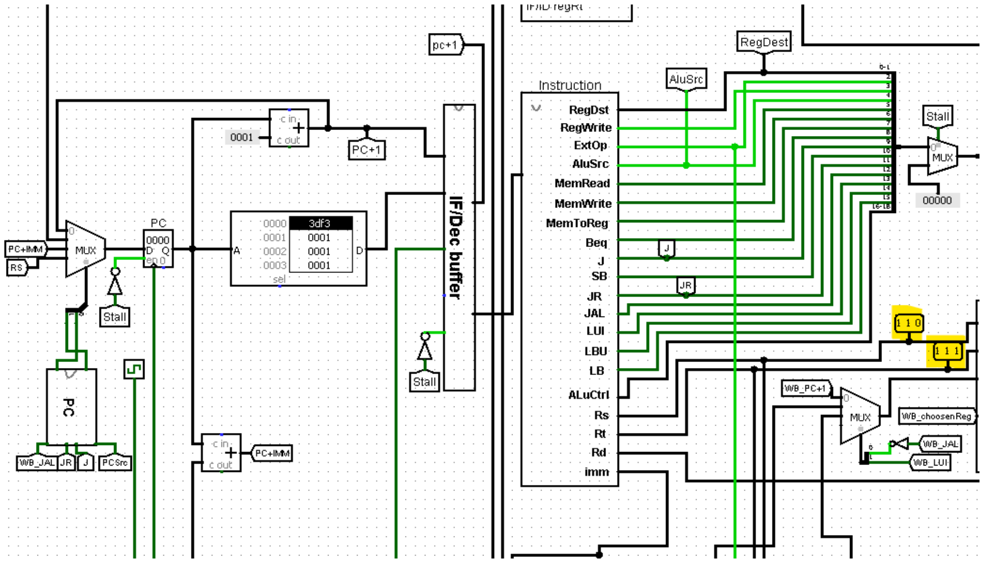
```
0000 3df3 0001 0001 0001  0001 0001 0001 0001  0001 0001 0001 0001  0001 0001 0001 0001
0010 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0020 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0030 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0040 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0050 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
0060 0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000  0000 0000 0000 0000
```

An example for test instruction 3df3 that's in binary **0011 1101 1111 0011**

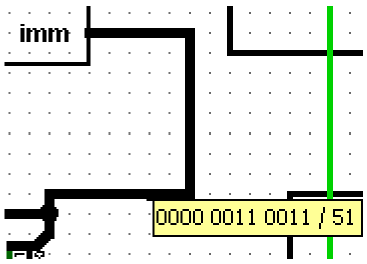
Opcode = 0011 (ADDI) and Rs = 110 and Rt = 111 and

the immediate 6-bit = 110011

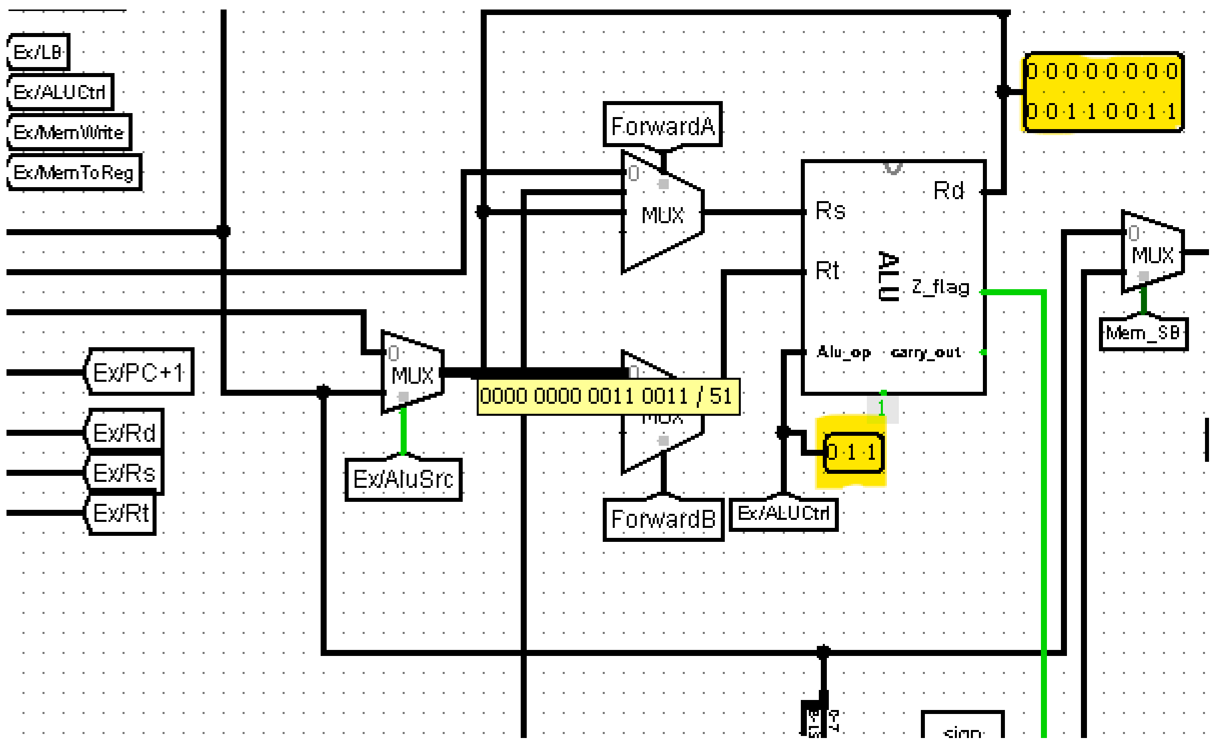
when click the first clock



and the imm6



The second clock



In ADDI the AluSrc= 1 so pass the immediate since ForwardB = 0 so the immediate is the Rt input and input for Rs = 0000000000000000 and ALUctrl for ADDI is 011

So, work as adder between immediate and 0 and the result is the immediate

References

[1]. Course slides

[2]. Exams format to implement some instructions

[3]. Arch website that’s help to design some blocks