Faculty of Engineering & Technology
Computer Systems Engineering Department

Semester, 2020/2021
Project -2-
Computer Architecture - ENCS4370

Prepared by:
Raghad Qadah 1170142
Leena Bany Odeh 1172351

Date: 12/6/2021

Abstract:

In this project, we are using Logisim to implement and test a pipelined RISC machine that processes 16-bit RISC processor with eight 16-bit general-purpose registers: R0 through R7. All instructions are 16 bits. There are three instruction formats (R-type, I-type, and J-type) and five addressing modes. The processor has a memory with word size equal 16-bit and a total size of 2^{16} bytes.

Table of Contents:

1. 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 data path along with
The parts that are needed to enable pipelining and its hazard detection unit, forwarding unit, and the buffers between the stages.

The design will include five stages: Fetching, Decoding, ALU, and Memory and
The Write Back stage.

1.1ALU:

The ALU has seven operations: AND, OR, CAS, LWS, ADD, SUB & SLT. It also has two flags: the zero 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:

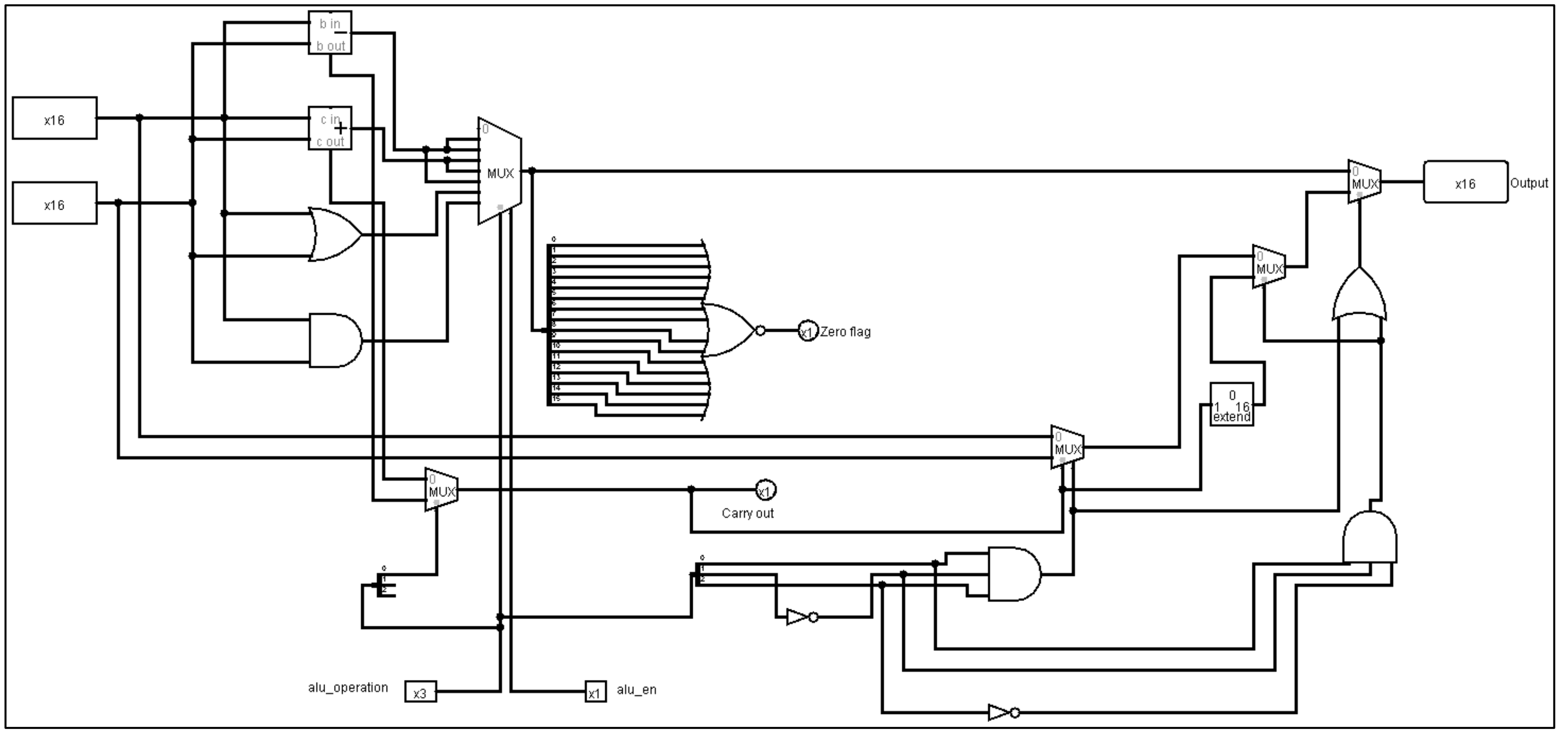


Figure 1: Arithmetic and Logic Unit (ALU)

The design procedures of the (ALU):

We specify the operations required that that the ALU circuit have to perform. Make a special combinational circuit for each operation and used a Multiplexer to set up the (ALU), where the selectors of this Multiplexer will be the (ALU) operation code bit that come from the main control unit (AluOp) , and the inputs of this multiplexer are the results of the combinational circuits we have been built under the constraint that each combinational circuit’s output should enter in the corresponding location to its (ALU) operation code.

1.2. Control Unit:

For the control unit, we built the Truth Table of all the necessary signals in the systems, and from it built the Boolean functions that use the Instruction bits as input and produce the needed signals. We then implemented the Boolean functions using the logic gates found in Logisim. Following are the truth table, the logic circuit of the Control unit and Boolean functions.

Control Unit truth table:

	RegDest	RegW	ALUSrc	Beq	J	Jal	MemW	MemR	MemToReg	RegS	Ex,Lb	Lw,Lb	Jr
AND	1	1	0	0	0	0	0	0	0	X	X	X	0
OR	1	1	0	0	0	0	0	0	0	X	X	X	0
CAS	1	1	0	0	0	0	0	0	0	X	X	X	0
LWS	1	1	0	0	0	0	0	1	1	X	X	X	0
ADD	1	1	0	0	0	0	0	0	0	X	X	X	0
SUB	1	1	0	0	0	0	0	0	0	X	X	X	0
SLT	1	1	0	0	0	0	0	0	0	X	X	X	0
JR	X	X	X	0	0	0	0	0	0	X	X	X	1
ANDI	0	1	1	0	0	0	0	0	0	X	X	X	0
ORI	0	1	1	0	0	0	0	0	0	X	X	X	0
ADDI	0	1	1	0	0	0	0	0	0	X	X	X	0
LW	0	1	1	0	0	0	0	1	1	X	X	0	0
LBU	0	1	1	0	0	0	0	1	1	X	0	1	0
LB	0	1	1	0	0	0	0	1	1	X	1	1	0
SW	X	0	1	0	0	0	1	0	0	0	X	X	0
SB	X	0	1	0	0	0	1	0	0	1	X	X	0
BEQ	X	0	0	1	0	0	0	0	0	X	X	X	0
J	X	0	X	0	1	0	0	0	0	X	X	X	0
JAL	X	0	X	0	0	1	0	0	0	X	X	X	0
LUI	X	0	X	0	0	0	0	0	0	X	X	X	0

*RegDest→ Rd=1, Rt=0

*ALUSrc→BusB=0

* Ex, Lb→ex=1

*Lw, Lb→Lw=0, Lb=1

Control unit Boolean functions:

RegDest=Rtype

RegW= ~ (SW+SB+BEQ+J+JAL+LUI)

ALUSrc=~ (Rtype+BEQ)

BEQ=BEQ

J=J

JAL=JAL

JR=JR

MemW= SW+SB

MemR=LW+LBU+LB+LWS

MemToReg= LW+LBU+LB+LWS

Ex, Lb=LB

LW, Lb=LB+LBU

RegS=SB

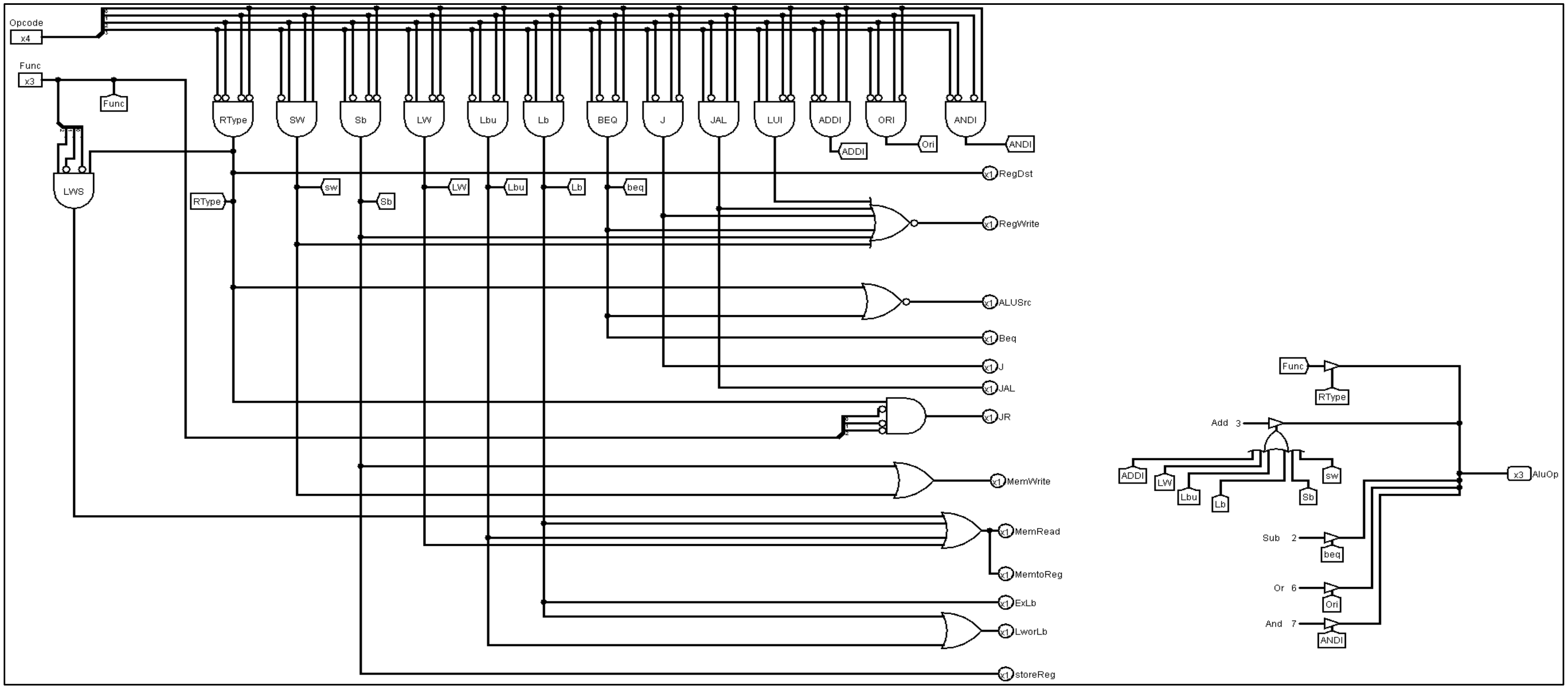


Figure 2: control unit

1.3. Memory:

We have used two types of memory in our design as follow:

1- Data Memory

For the memory, we used the built-in RAM module from Logisim, along with MemRead-MEM with the MemWrite-MEM signals. MemWrite-MEM signal also controls the flow of data as well as the ld (load) control of the module.

The processor will have separate instruction and data memories with 216 words each. Each word is 16 bits or 2 bytes. Memory is word addressable. Only words (not bytes) can be read and written to memory, and each address is a word address. This will simplify the processor implementation. The PC contains a word address (not a byte address). Therefore, it is sufficient to increment the PC by one to point to the next instruction in memory.

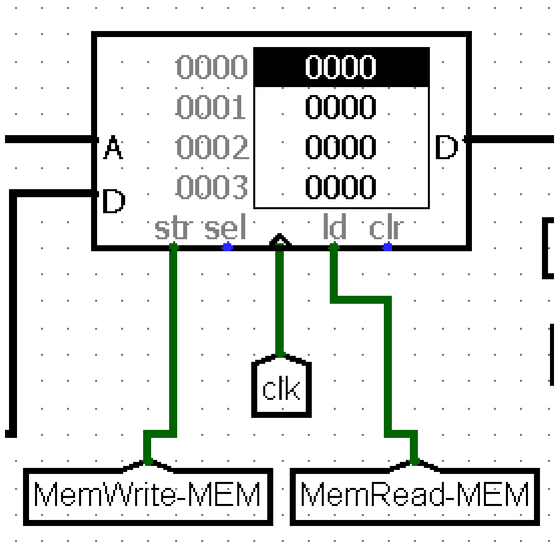
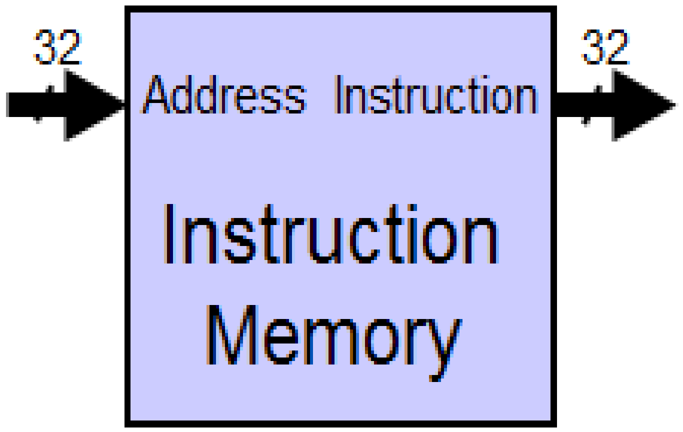


Figure 3: Data Memory

2- Instruction Memory

It represents the instruction memory that needs only provide read access, because Datapath does not write instruction, also it behaves like a combinational logic for read, address which is input from PC selects instruction after access time and then take them to the output of the instruction memory.



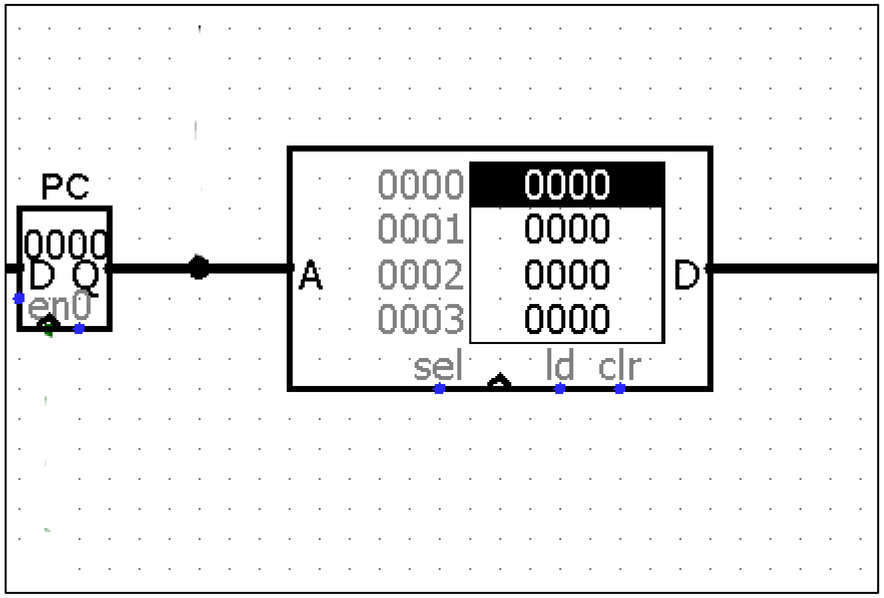


Figure 4: Instruction Memory

1.4. Register File:

We have implemented this Block by choosing the register you want to write it on using decoder, it takes the address that you want to write it on, then it enables only the register you called, RegWrite control signal must be equal to 1 for the AND gate, if RegWrite was equal to 1 , then the desired register will work and take the input from the control signal (BusW 16-bit), but if it was equal to 0 , then you can’t write to any register, instead of it can be read any register you want.

We used (Registers Block) to build Register File, Registers Block represent as an input for the MUXES, then the upper MUX choose the registers that it must pass to BusA(out1) according to the address in RA(reg1) control signal, same way in getting BusB(out2) according to the address in RB(reg2) control signal.

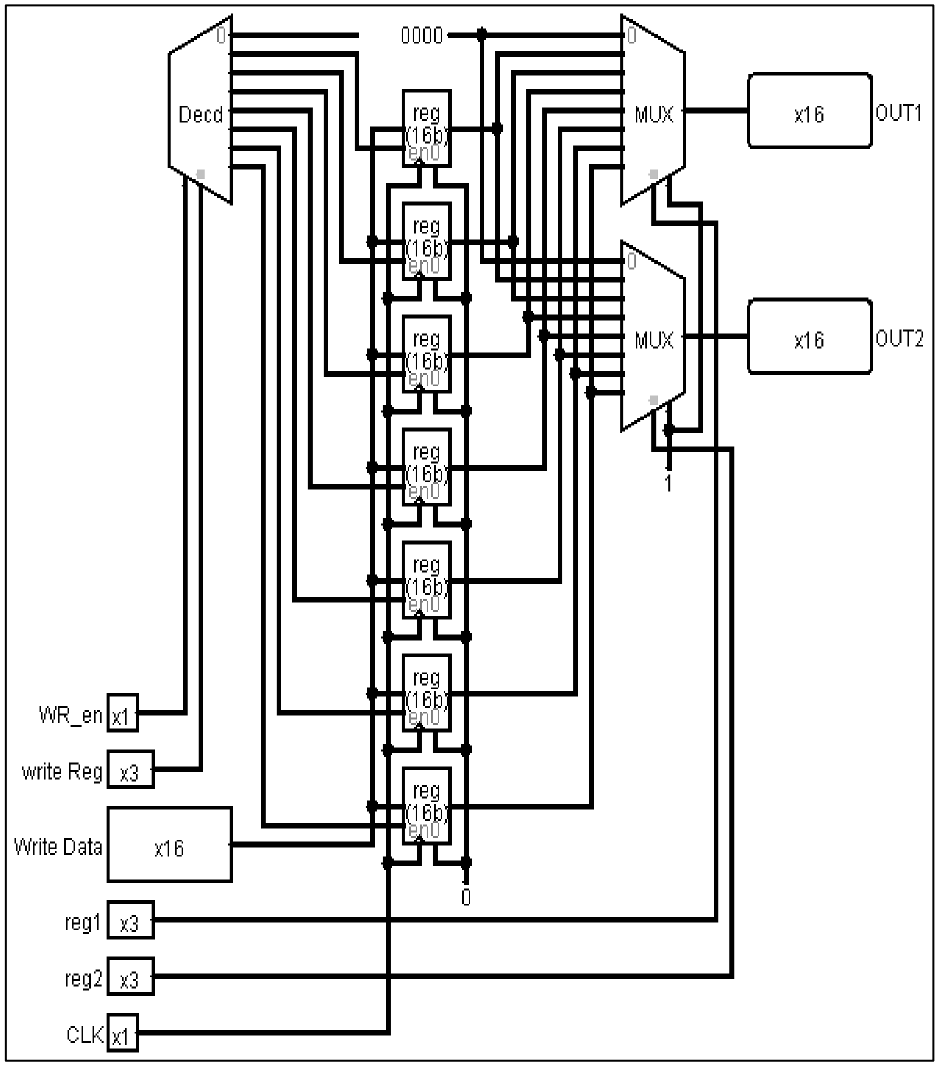


Figure 5: Register File

1.5. Fetching Stage:

The instruction is fetched here, from the instruction memory and then is fed to the decoding stage through the Fetch/Decode buffer.

- For instruction fetching, we need …
  - ❖ Program Counter (PC) register.
  - ❖ Instruction Memory.
  - ❖ Adder for incrementing PC.

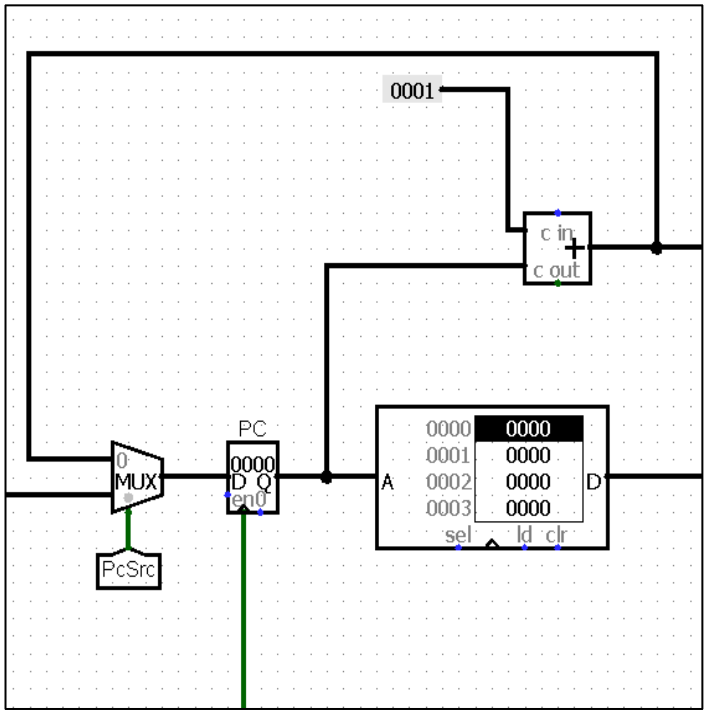


Figure 6: Fetching stage

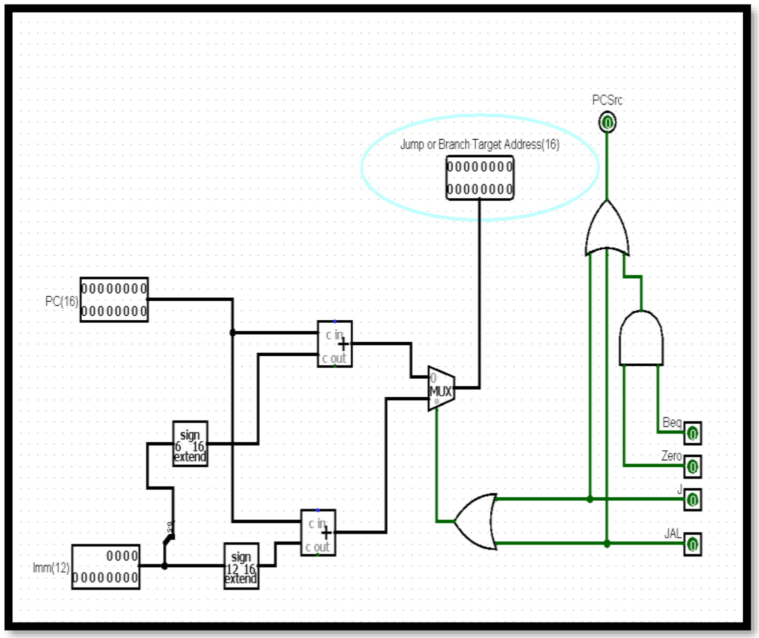


Figure7: next Pc unit

1.6. Decoding Stage:

IN this part, splitter decodes the fetched instruction to four format as follow:

- R_type format:

  - 4-bits Opcode (Op) [12:15]…...to control unit.
  - 3-bits destination register Rd [3:5]…..to the register file.
  - 3-bits source registers Rs [6:8]…...to the register file.
  - 3-bits source registers Rt [9:11]…......to the register file.
  - 3-bits function field [0:2]…....to (ALU).

- I_type format:

  - 4-bits Opcode (Op) [12:15]…....to control unit.
  - 3-bits destination register Rt [6:8]…to the register file.
  - 3-bits source register Rs [9:11]…...to the register file.
  - 6-bits immediate [0:5]…...to (ANDI, ORI ADDI, LW, LBU, LB, SW, SB, BEQ) target address adder.

- J_type format:

  - 4-bits [12:15] Opcode (Op)….to control unit
  - 12-bits immediate [0:11]… to (J, JAL, LUI). ALU target address adder.

The decoding stage will decode each instruction and pass its bit segments separately to the control unit to obtain flags, and then passes them to the next stage along with the registers addresses and the immediate data(if any).

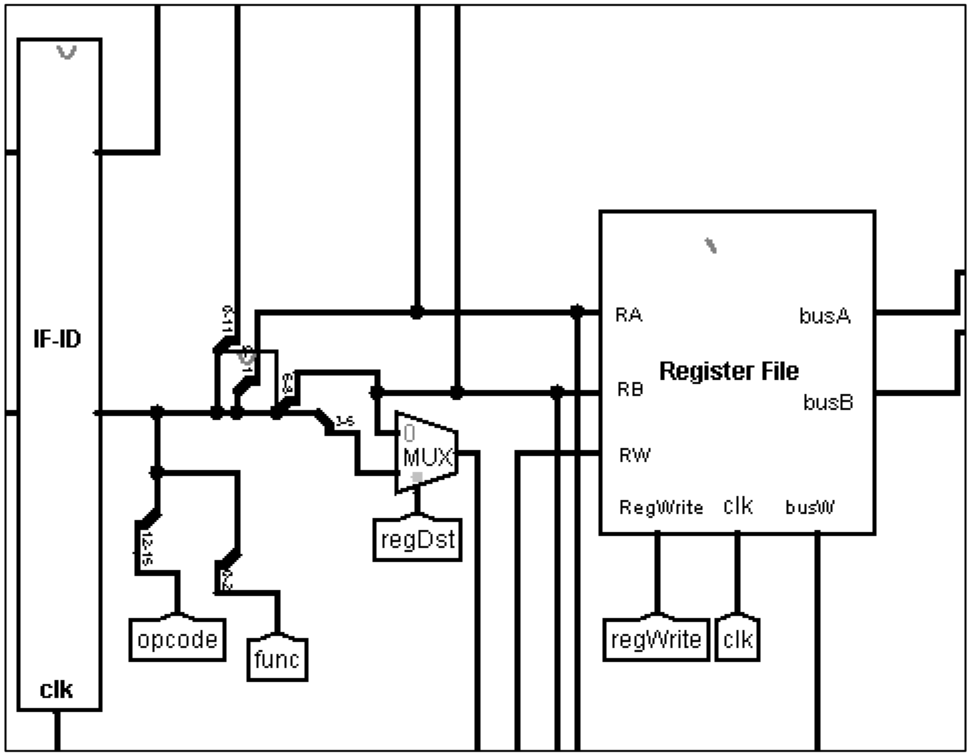


Figure 7: Decoding Stage.

1.7. Write Back Stage:

In this part, we select which data will be written in the register file through three multiplexers:

- ❖ Memory to register mux differs between data memory output and Alu results , its selector is the memory to register pin from control unit
- ❖ The second mux choose between load word or load byte if the instruction is LB or LBU the output of the first mux will be extended according to the signal exLb-WB.
- ❖ The third mux is to choose between R-Type, ALU I-type and load instructions.

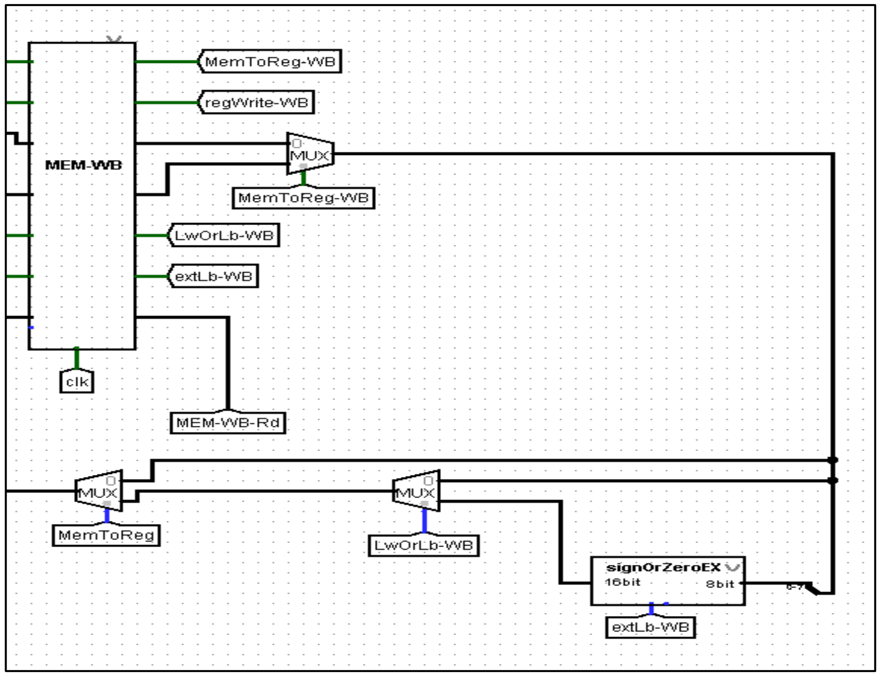


Figure 8: Write Back Stage.

2. Pipelining:

In the single cycle processor the CPI equal one that means, “The processor performs only one instruction each clock cycle”. The main problem of this technique is that there are many parts and devices may be stand free for long time period and this affects the performance of the processor.

We can exploit these circuits by using another technique, which is called “pipelining “.This technique improves the performance of the processor. The main difference between the single cycle processor and the Pipelined processor is that the instructions are overlapped through the different stages in the Datapath.

The only major advantage is performance improvement. By pipelining instructions, we can pump in more instructions in the processor and then we get a significant improvement in processor speed. This happens because we can execute parts of instructions in parallel to parts of other instructions.

For long sequences of instructions, this technique is extremely effective because in every step the processors different parts busy with an instruction. The only major disadvantage is complicity process in designing the processor. All instructions must enter sequence of five stages each of them needs one clock cycle so each instruction is completely performed after five clock cycles.

2.1. Data Hazard Detection Unit:

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

- **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

  - Check for load instruction
  - Check if the register to be loaded is part of the current instruction
  - If it is, stall the pipeline
  -

And the following is our design for the data hazard unit:

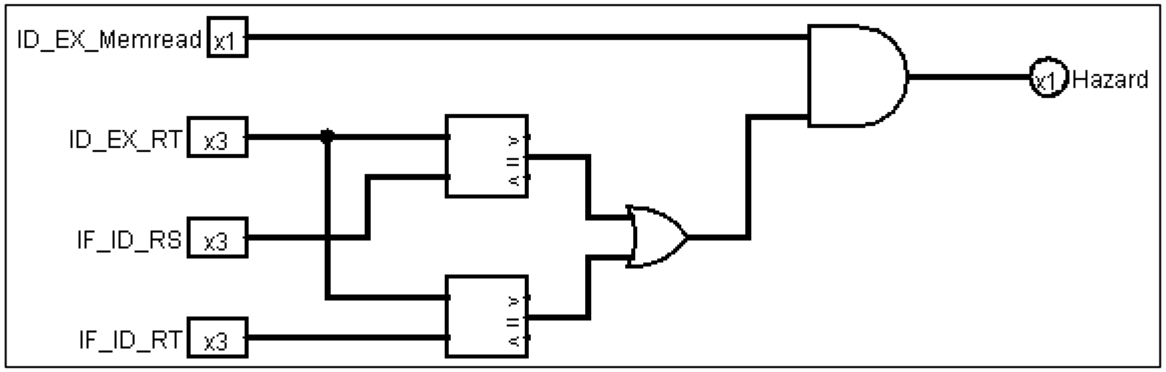


Figure 9: Hazard Detection Unit.

2.2. Forwarding Unit:

The reason for forwarding:

An instruction in the execution stage needs data (dependency) from an instruction in the memory access stage (EXE/MEM register) or write back stage (MEM/WB register).

Detection 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 (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)

    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

ForwardA = 01

– If (MEM/WB.RegWrite

and (MEM/WB.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)

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

ForwardB = 01

And the following is our design for the forwarding unit based on the above conditions:

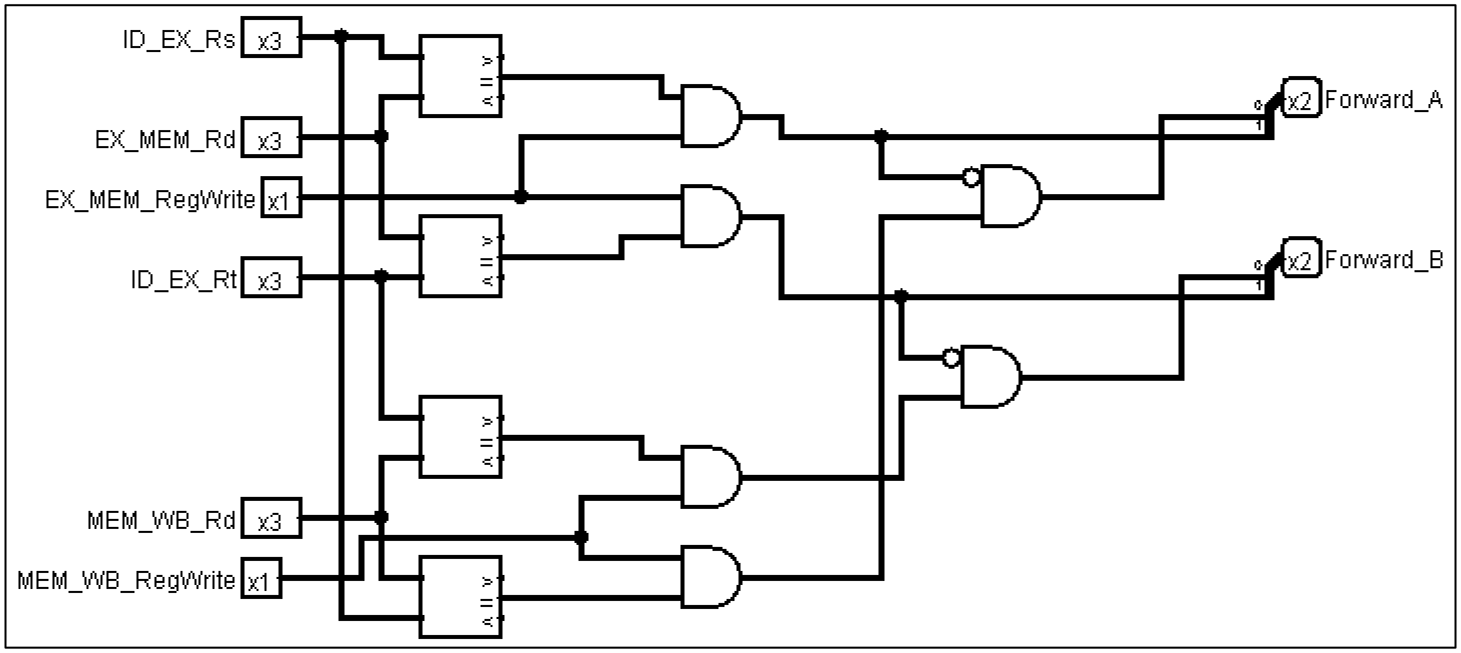


Figure 10: Forwarding Unit.

2.3. Buffers:

1. **Fetch/Decode buffer:**

   This buffer is implemented as follows:

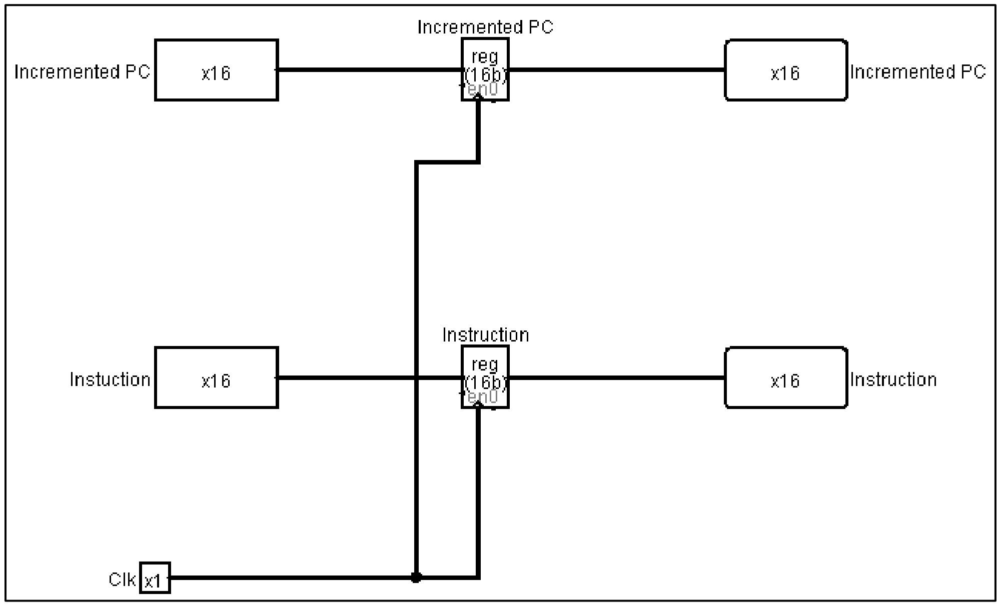
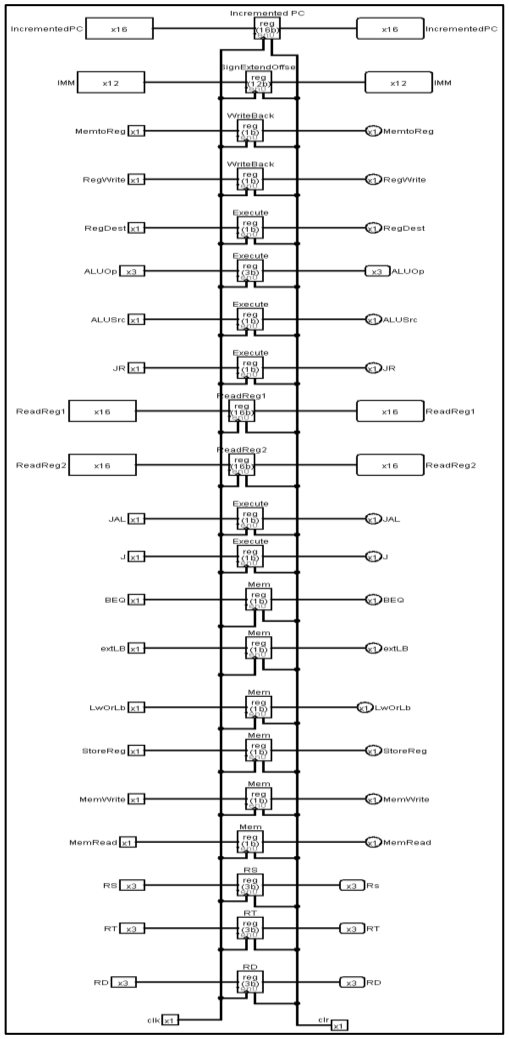


Figure 11: IF/ID Buffer.

Its purpose is pass the instruction along with the incremented PC register from the fetching stage to the decoding stage.

2. **Decode/Execute buffer:**

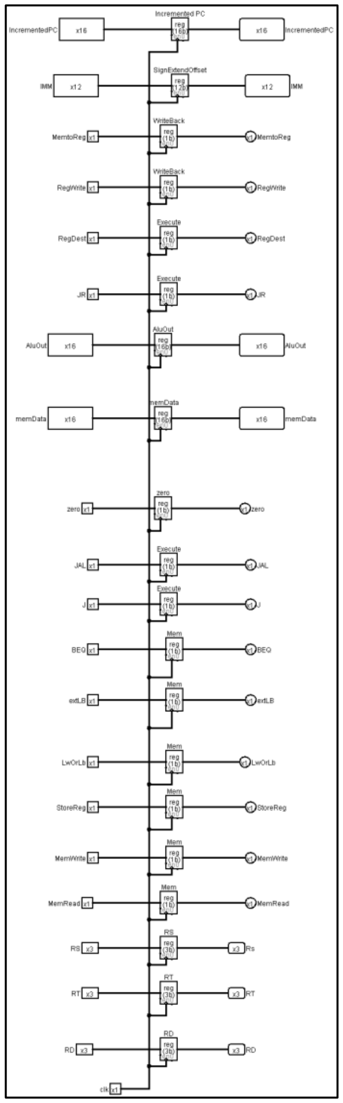
This buffer is implemented as follows:



This buffer is used after decoding the instruction is done. It buffers all segments, and control signals, for the instruction to be used.

Figure 12: ID/EX Buffer.

3. **Execute/memory buffer:**
   Implemented as follows:



This buffer is used execution and memory stage .

4. Memory/WB buffer

This is used right before the last stage (WB); it passes only the dst register address and the register WR signal. Implemented as follows:

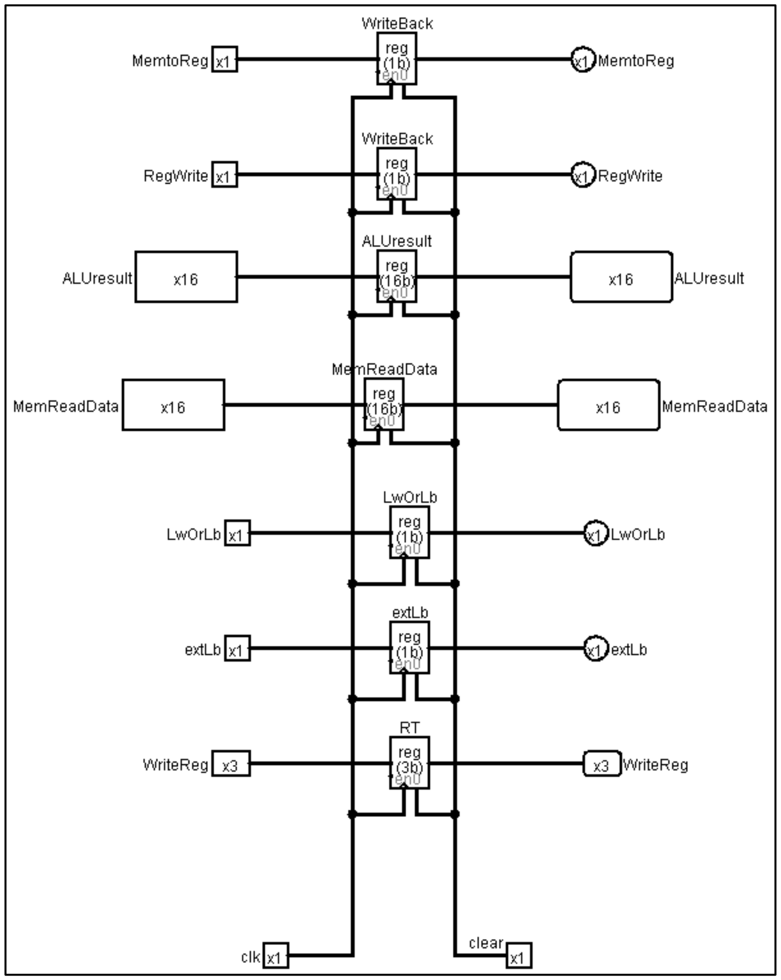


Figure 14: MEM/WB Buffer.

5. Full Datapath:

And so the full, final, datapath (connecting all the described parts) becomes:

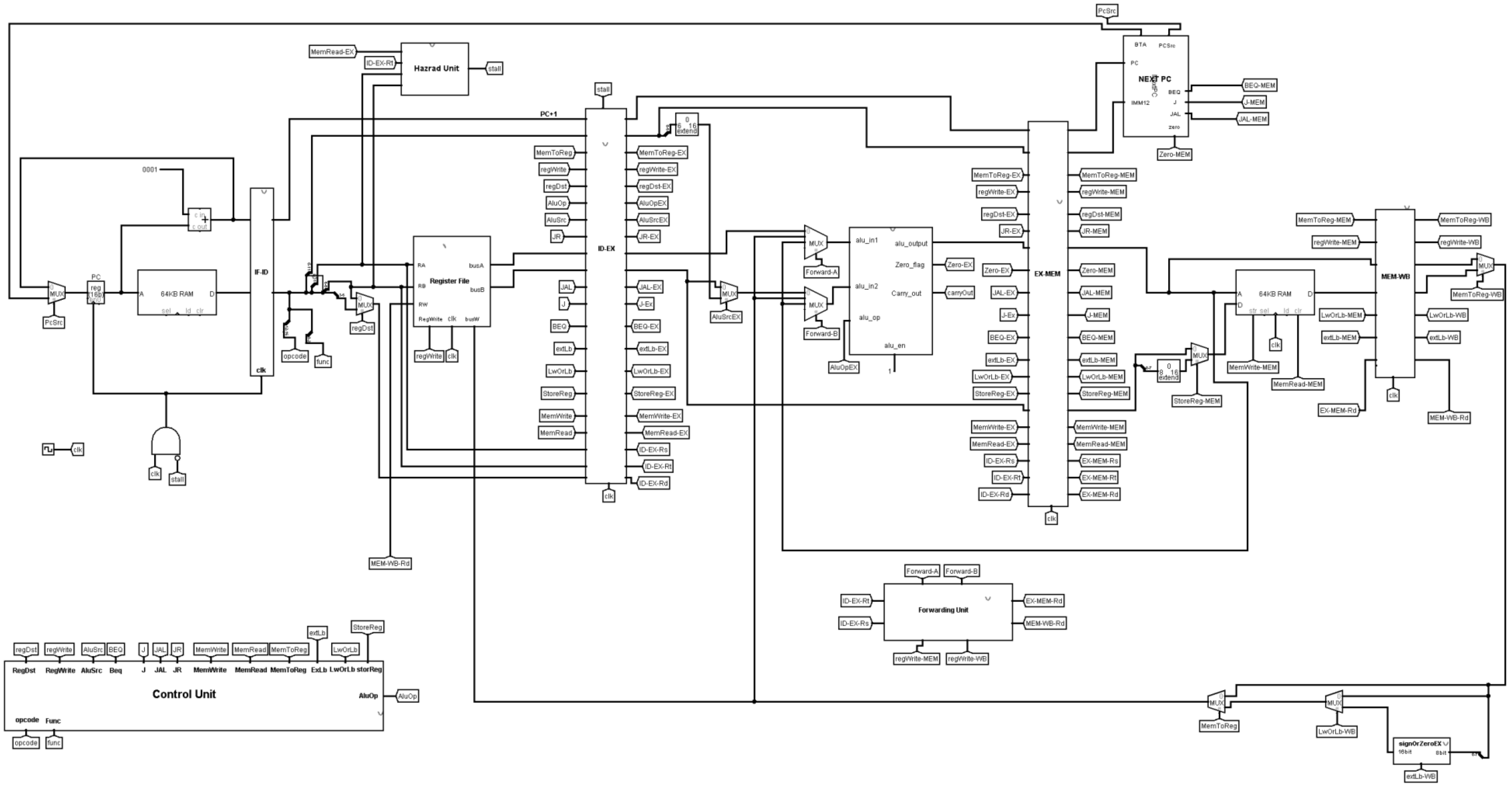


Figure 15: Final Data path

6. Simulation and Testing:

We tested the following instruction sequence after we loaded it into the program memory:

addi R1, R2, 1	**0x3441**
addi R2, R1,2	**0x3282**
CAS R3, R1, R2	**0x045D**

This code sequence is supposed to add number (1) and R2 and store result in register R1, then add it with number (2) and store the output in R2. Then it compare between R1 and R2 using CAS and load the max value in R3.

And other instruction used:

addi R2, R1, 1	**0x3282**
addi R1, R2,5	**0x3445**
sub R1, R1, R2	**0x028A**
Lb R3 , 0(R1)	**0x62C0**
LW R4 , 0(R2)	**0x4500**

This code sequence is supposed to add number (1) and R1 and store result in register R2, then add it with number (5) and store the output in R1. Then sub 2 from R1 and load byte from R1 with 0 offset to R3. and load byte from R2 with 0 offset to R4.

8. Conclusion:

The simulation has worked fine and it outputs in almost instruction is correct and the data in the right place in memory. The project was not hard, but it took large time, and it helped us to understand the pipelined processor in such a way that we cannot easily forget, and it can be considered as a good progress and a good project to help us to understand the processor by simply designing it.