Birzeit University
Faculty of Engineering and Technology Department of Electrical and Computer Engineering
Computer Architecture - ENCS4370

Project NO.2

Prepared by: Qutaiba Olayyan 1190760 , Baraa Fatony 1180566 , Diana Allan 1180665
Instructor: Dr. Aziz Qaroush

BIRZEIT
July 2, 2023

1 ABSTRACT

In this project a multicycle 32-bit processor was designed and tested with HDL language (Verilog). A RISC processor with 32 32-bit general purpose registers and a 32-bit program counter register was implemented, supporting 32 bits instructions with four different formats (R-type, I-type, J-type, and S-type) and different addressing modes. A five-stage multicycle-datapath and its control logic were implemented completely from scratch. All details implemented and encountered in the process of implementing this datapath along with its logic will be illustrated in details and supported by various test cases.

Contents:

2 THEORY

In this section, the datapath which is a critical component in building any computer is described briefly as well as an explanation of what is meant by a multicycle architecture.

2.1 Datapath

The datapath consists of several key components, including registers, arithmetic and logic units (ALUs), multiplexers, and buses. Registers are temporary storage units used to hold data during processing. The ALU performs arithmetic and logical operations, such as addition, subtraction, AND, OR, etc. Multiplexers are used to select inputs from multiple sources and direct them to the appropriate destination. Buses are the communication pathways that transfer data between different components of the datapath.
The datapath operates in conjunction with the control unit, which coordinates the execution of instructions and manages the flow of data within the processor. The control unit generates control signals that determine which operations are performed by the datapath and when they occur.

2.2 Multicycle Architecture

Multicycle architecture is a computer architecture design approach that divides the execution of instructions into multiple clock cycles. Each clock cycle performs a specific stage of instruction execution, allowing for more efficient utilization of hardware resources and improved performance. It breaks down complex instructions into smaller steps, enabling concurrent execution of multiple operations. This approach involves stages such as instruction fetch, instruction decode , instruction execute , memory access and write back. Multicycle architecture balances hardware resource usage and can reduce design complexity and cost, although it introduces additional control logic.

3 PROJECT BACKGROUND

This project aims to implement and test a 32-bit multicycle processor. It was built starting by building the basic components followed by connecting these different components together . Our design is characterised by a file register with 32 general purpose registers, each with 32-bit. It also has a 32-bit PC register. The instruction set architecture supports four different format types as follows, in which each instruction is 32-bit obtained as a full word from the instruction memory .

This processor has a stack called control stack which saves the return addresses , also it has another special purpose register called Stack pointer (SP), to point to the top of the control stack. SP holds the address of the empty element on the top of the stack. For simplicity, you can assume a separate on-chip memory for the stack, and the initial value of SP is zero.

Instruction Types and Formats

As mentioned above, this ISA has four instruction formats, namely, R-type, I-type, J-type, and S-type. These four types have the following common fields:

1. **2-bit instruction type** (00: R-Type, 01: J-Type, 10: I-type, 11: S-type)

2. **5-bit function**, to determine the specific operation of the instruction

3. **Stop bit**, which is the least significant bit of each instruction binary format, and it is used to mark the end of a function code block. In other words, if the value of this stop bit is “1”, this means that this instruction is the last instruction of the function, and hence the execution control should return to the return address which is stored on the top of the control stack.

Instruction Formats as following :

1. R-type Format
5-bit Rs1: first source register
5-bit Rd: destination register
5-bit Rs2: second source register
9-bit unused

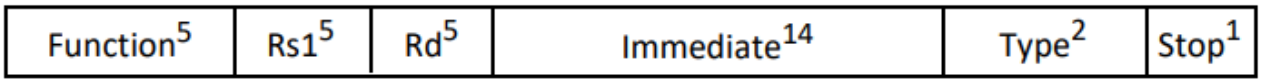
Function^5	Rs1^5	Rd^5	Rs2^5	Unused^9	Type^2	Stop^1

2. I-Type (Immediate Type) Format

5-bit Rs1: first source register

5-bit Rd: destination register

14-bit immediate: unsigned for logic instructions, and signed otherwise



3. J-Type (Jump Type) Format

24-bit signed immediate: jump offset

4. S-Type (Shift Type) Format

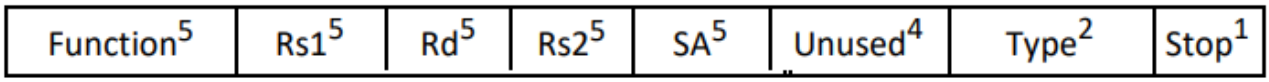
5-bit Rs1: first source register

5-bit Rd: destination register

5-bit Rs2: second source register. This register stores the shift amount in case the shift amount is variable and it is calculated at runtime

5-bit SA: the constant shift amount.

4-bit unused



Instructions' Encoding:

For our datapath, the following instructions are being supported as the encoding entered in Table 1.

Table 1: Instruction Encoding

No.	Instr	Meaning	Function Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	00000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	00001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	00010
4	CMP	zero-signal = Reg(Rs) < Reg(Rs2)	00011
I-Type Instructions			
5	ANDI	Reg(Rd) = Reg(Rs1) & $Immediate^{14}$	00000
6	ADDI	Reg(Rd) = Reg(Rs1) + $Immediate^{14}$	00001
7	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{14})	00010
8	SW	Mem(Reg(Rs1) + Imm^{14}) = Reg(Rd)	00011
9	BEQ	Branch if (Reg(Rs1) == Reg(Rd))	00100
J-Type Instructions			
10	J	PC = PC + $Immediate^{24}$	00000
11	JAL	PC = PC + $Immediate^{24}$ Stack.Push (PC + 4)	00001
S-Type Instructions			
12	SLL	Reg(Rd) = Reg(Rs1) << SA^5	00000
13	SLR	Reg(Rd) = Reg(Rs1) >> SA^5	00001
14	SLLV	Reg(Rd) = Reg(Rs1) << Reg(Rs2)	00010
15	SLRV	Reg(Rd) = Reg(Rs1) >> Reg(Rs2)	00011

4 DESIGN AND IMPLEMENTATION

Each developed datapath component as well as all required control units will be described in depth in this section.

4.1 Combinational Elements

This section will include illustrations of the ALU, comparator, multiplexers and extenders .

4.1.1 ALU

This unit has two 32-bit inputs as well as signal to decide which operation of ALU will select, and the 32-bit result , zero (it will be one if subtraction result equal zero) , CMP_zero (it will be one if A<B) , carry and negative (it will be one if the result is negative) comes out of it as outputs.

According to the instructions table provided, five separate ALU operations are required to carry out each of the supported instructions. Which are:

1. Anding: implementing simply with an AND gate, accepting two 32-bit inputs.

2. Addition: implemented with adder , accepting two inputs each 32-bit.

3. Subtraction: implemented with subtractor , accepting two inputs each 32-bit .

4. Right Shifting: implemented with shifter, one input is the value to be shifted and the other input is the amount of shifting .

5. Left Shifting: implemented with shifter, one input is the value to be shifted and the other input is the amount of shifting .

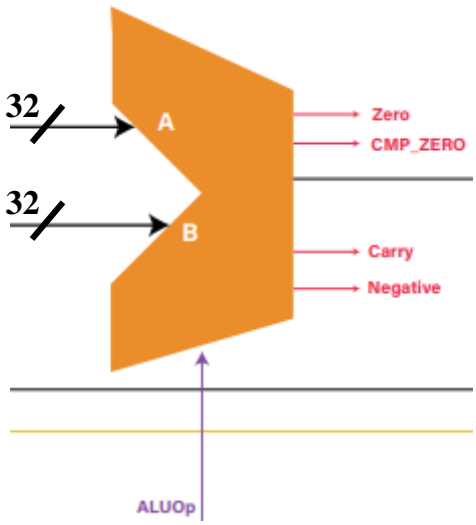


Figure 1: ALU Unit

4.1.2 Extender_5

This block was implemented to extend the 5-bit immediate, obtained from the S-type instruction, into a 32-bit number by using zero extension since all the S-Type instructions require a zero extension.

Figure 2: Extender_5 Unit

4.1.3 Extender_14

This block was implemented to extend the 14-bit immediate, obtained from the I-type instruction, into a 32-bit number with the capability for doing the extension with the sign extension or the zero extension based on the ExtOP signal, which is derived from the control unit.

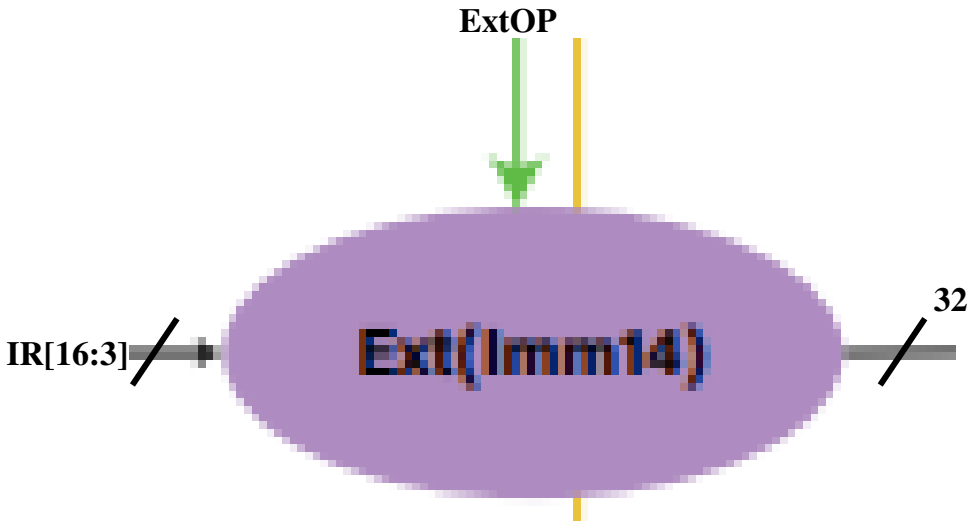


Figure 3: Extender_14 Unit

4.1.4 Extender_24

This block was implemented to extend the 24-bit number, obtained from the J-Type instruction, into a 32-bit number by using the sign extension since all the J-Type instructions require a sign extension.

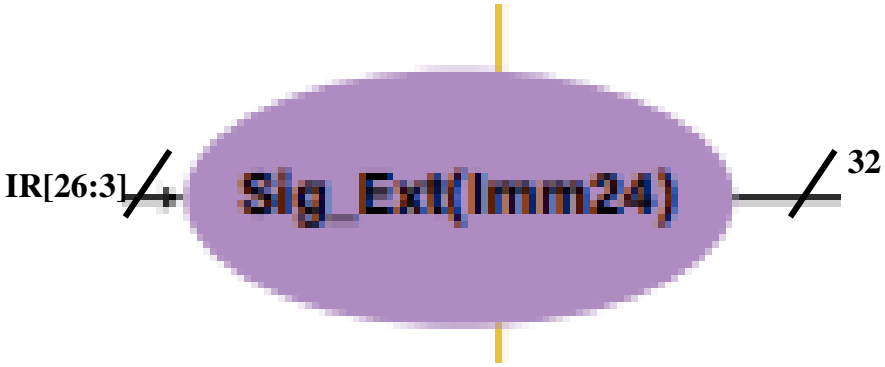


Figure 4: Extender_24 Unit

4.1.5 PC_Multiplexer

This multiplexer chooses the correct result for the current instruction in the fetch stage; so either it takes one of four choices (normal next address , jump target address , branch target address , return address after JAL instruction), so the selection signal for this mux is PCSrc signal from control unit.

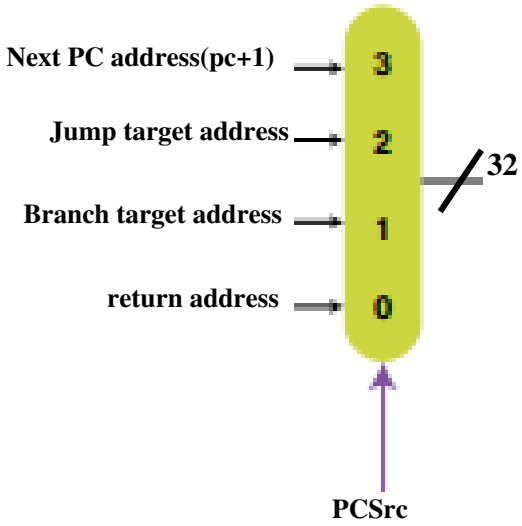


Figure 5: PC_Multiplexer

4.1.6 Source_Register Multilplexer

This multiplexer chooses the correct second register to read from registers file in the decoding stage especially in BEQ and SW instructions; so either it takes Rs2 or Rd, so the selection signal for this mux is RegSrc signal.

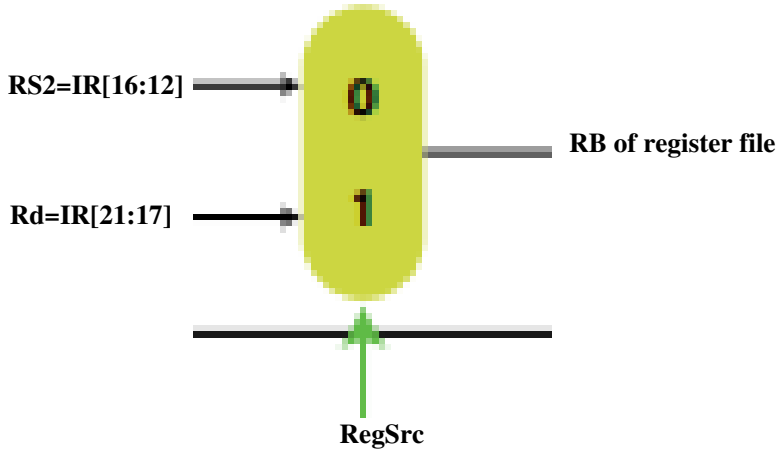


Figure 6: Source_Register Multilplexer

4.1.7 Second input of ALU Multilplexer

This multiplexer chooses the second input for the ALU in the execution stage; so either it takes immediate after extending in I-type instruction or immediate after extending in S-type instruction or the content of second register in R-type instruction , so the selection signal for this mux is ALUSrc signal.

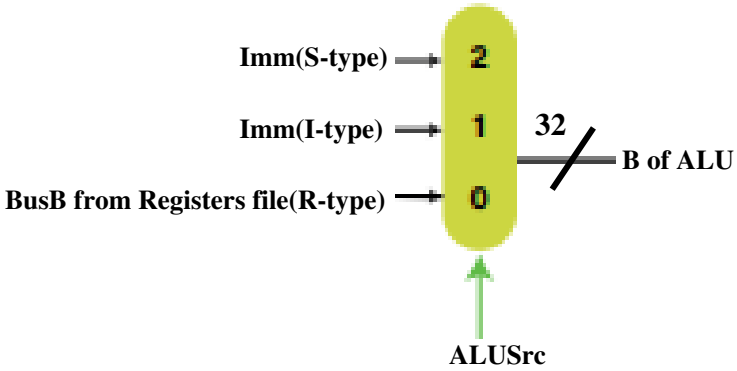


Figure 7: Second input of ALU Multilplexer

4.1.8 MemoryResult_ALUResult Multiplexer

This multiplexer chooses the final result that need to be written into the Register File which can either be the ALU result or the memory result, and the selection signal for this multiplexer is the WBData signal.

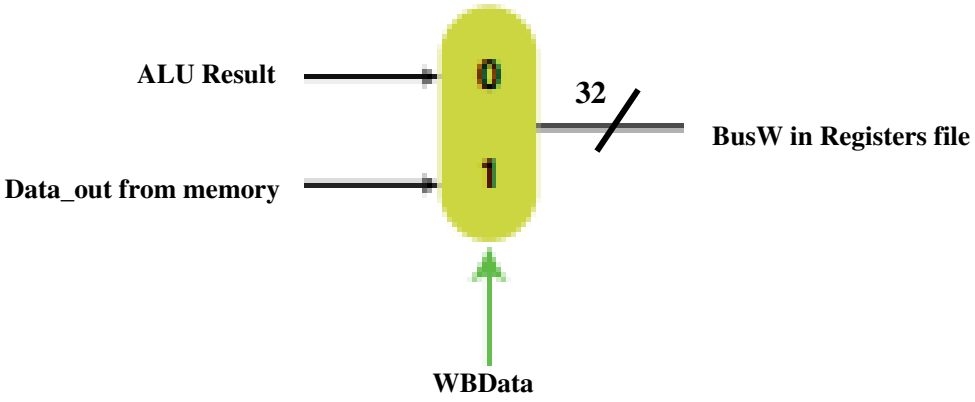


Figure 8: MemoryResult_ALUResult Multiplexer

4.1.9 Jump target address Adder

This adder calculate the target address that we want to jump to it in J-type instruction , this address equal previous pc added to it immediate of J-type instruction after extending .

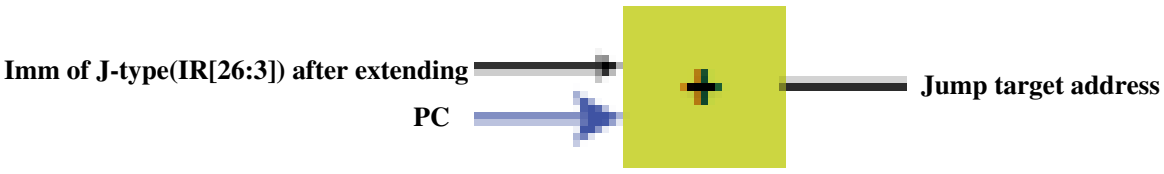


Figure 9: Jump address Adder

4.1.10 Branch target address Adder

This adder calculate the target address that we want to go to it in Branch instruction , this address equal pc+1 added to it immediate of I-type instruction after extending .

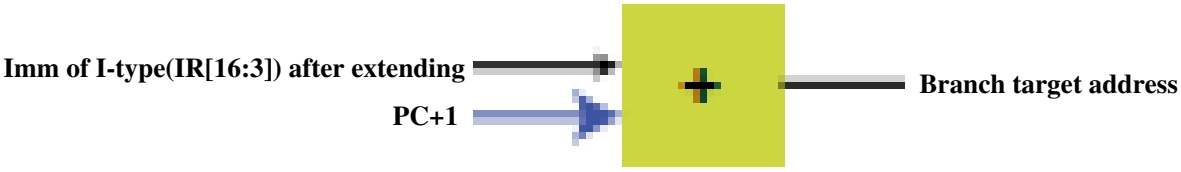


Figure 10: Branch target address Adder

4.2 Storage Elements

4.2.1 Data Memory

The data memory is of size 4GBX32, with a Data interface of type Separate load and store ports, so the interface of the memory has the following buses and signals:

1. Address : this port is specified for the address bus, which is of 32-bit.

2. Data_in : this port is specified for the data to be written into the memory, which is also of 32-bit.

3. Data_out : this port is specified for the data read from the memory, which is also of 32-bit.

4. clk : to synchronize the operations and ensure reliable data transfer between the memory and other components of a processor.

5. MemRd: read signal is necessary for memory because it indicates when data needs to be retrieved from the memory. It serves as a control signal that triggers the memory circuitry to fetch the requested data from a specific memory location.

5. MemWr: write signal is necessary for memory systems as it indicates when data needs to be stored or updated in the memory. It serves as a control signal that triggers the memory circuitry to store the incoming data at a specific memory location.

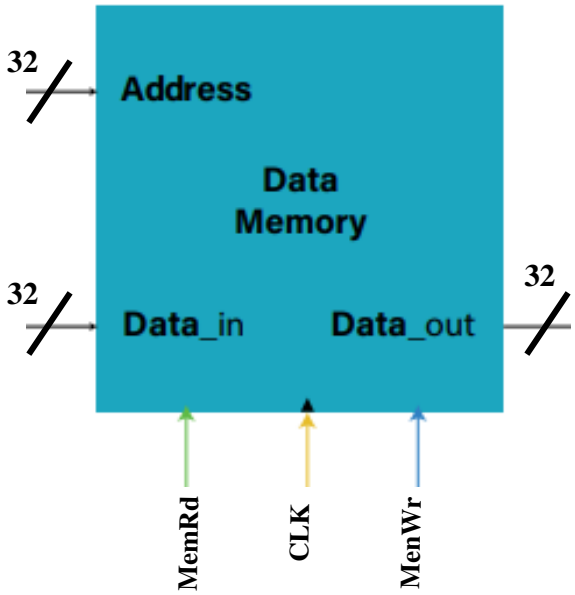


Figure 11: Data Memory

4.2.2 Instruction Memory

The instruction memory is of size 4GBX32, an input for the 32-bit address of instruction and an output is the instruction fetched from the specified address. The reading operation is done asynchronously whenever and address is ready.

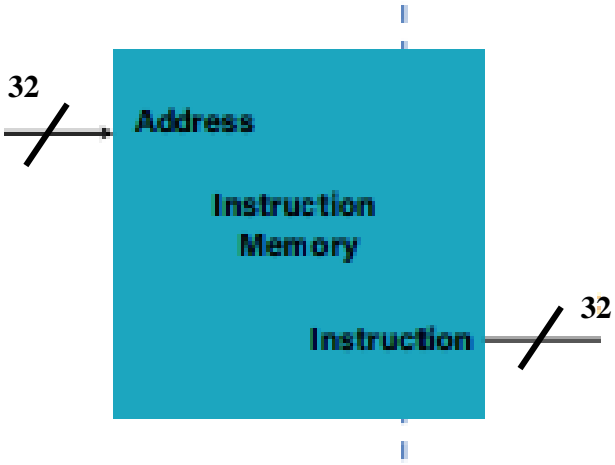


Figure 12: Instruction Memory

4.2.3 Program Counter

The program counter (PC) is a 32-bit register in a computer processor that holds the memory address of the next instruction to be fetched and executed. It determines the sequential order of instructions and plays a crucial role in the execution of programs.

This register has PCWr signal , it decide writing next address on PC or not , that depends on the number cycles needed by the processor in the previous instruction .

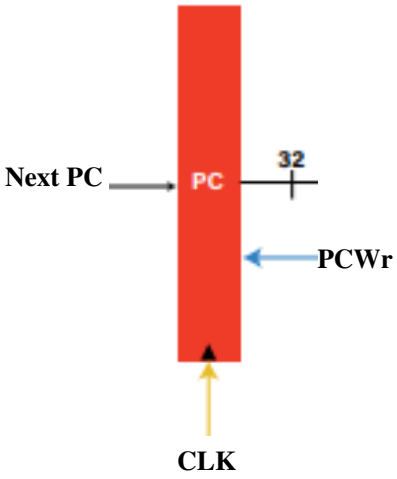


Figure 13: Program Counter

4.2.4 Control Stack

This component saves the return addresses(ra) that equal pc+1 when program has JAL instruction by push it on the top of the stack and increase stack pointer by one to point on the empty element in the top of stack, when the processor arrive the end of code block after detecting one on stop bit , control stack will pop return address to the PC as the address of next instruction and decrease stack pointer by one to point on the empty element in the top of stack .

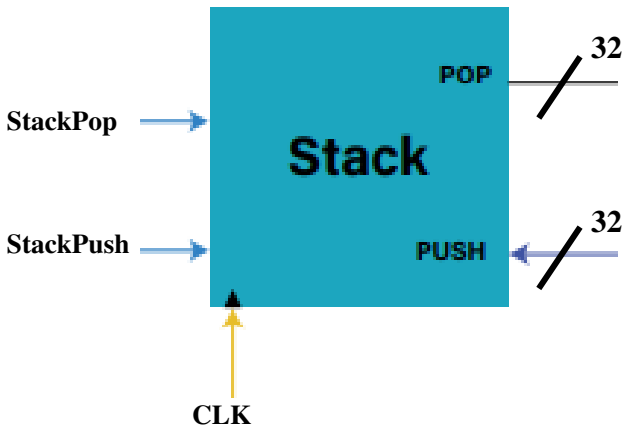


Figure 14: Control Stack

4.2.5 Buffers Between Stages

For multicycle purpose, we need to have a buffer between each two stages for managing data flow, coordinating operations, and enabling efficient resource sharing. They contribute to the smooth and efficient execution of instructions within the processor, so we have 4 buffers which are IF/ID, ID/EX, EX/MEM and MEM/WB , all of them are triggered at the rising edge of the clock. The following illustrate each buffer:

1. IF/ID: it has three buffers which they are : Instruction register (IR) , current program counter (pc) and next program counter (pc+1) , all of these were obtained from the Fetched instruction .

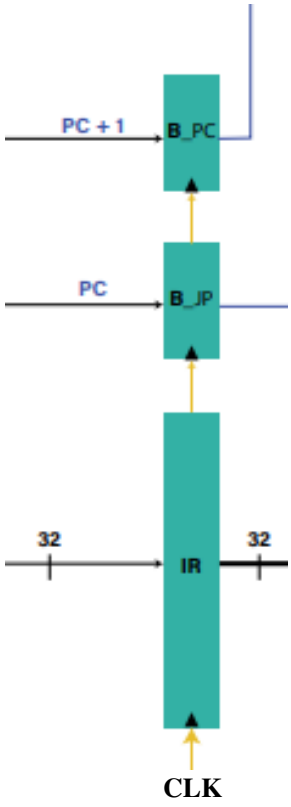


Figure 15: IF/ID Buffer

2. ID/EX: we have 7 buffers which are: BUSA and BUSB of 32-bit representing the values of Reg(RS1) and Reg(RS2 OR Rd) obtained from the register file , also we have imm_SType which it is obtained from the Extender_5 and imm_IType which it is obtained from the Extender_14 , also we have 32-bit jump target address , 32-bit branch target address and return address from control stack .

3. EX/MEM: We have 2 buffers which they are BusB of registers file and ALUResult.

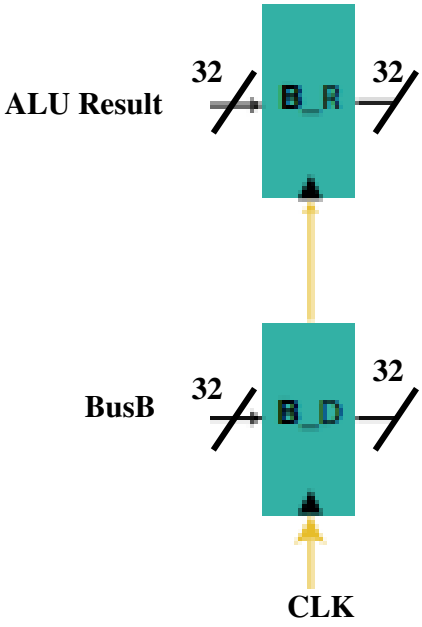


Figure 16: EX/MEM Buffer

4. MEM/WB: we have one buffer which it is the data that will write back to the register file , it will be either ALU result or Data_out from memory , it will be the output of MemoryResult_ALUResult Multiplexer.

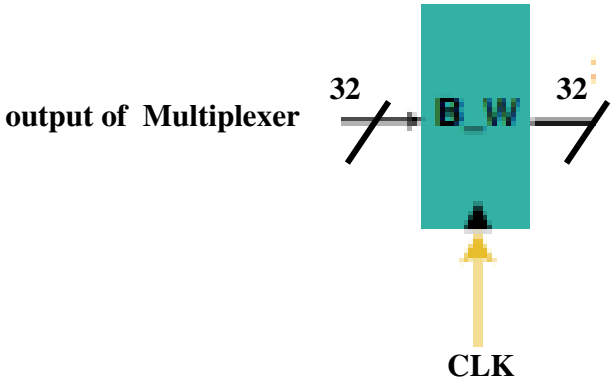


Figure 17: MEM/WB Buffer

4.2.6 Register File

Register file consists of 32 32-bit registers , register R0 equal zero and we can't write on it, and it is illustrated as the following:

1. BUSA And BUSB: 32-bit output buses for reading two registers.

2. BUSW: 32-bit input bus to write its data into the specified register when the needed conditions are applied .

3. RA: 5-bit input bus that select the register to be read on BUSA.

4. RB: 5-bit input bus that select the register to be read on BUSB.

5. RW: 5-bit input bus that select the register to loaded with the value in BUSW.

6. RegWr: 1-bit input that selects if the current action is read or write (1 means write, 0 means read).

7. CLK: The clock input is used only during write operation. During read, register file behaves as a combinational logic block RA or RB valid => BusA or BusB valid after access time.

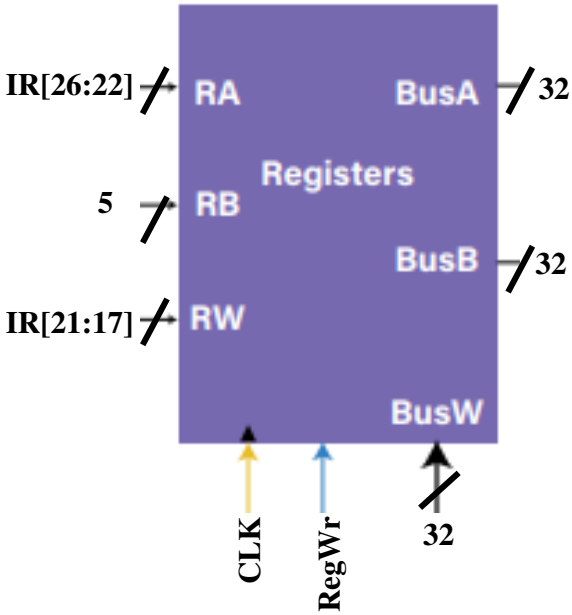


Figure 18: Register File Unit

4.3 Control Units

4.3.1 ALU_control Unit

Following is the ALU Control truth table

Table 2: ALU Control Truth Table

Type	Function	ALUOp	3bit Coding
R(00)	AND(00000)	AND	000
R(00)	ADD(00001)	ADD	001
R(00)	SUB(00010)	SUB	010
R(00)	CMP(00011)	SUB	010
J(01)	J(00000)	X	X
J(01)	JAL(00001)	X	X
I(10)	ANDI(00000)	AND	000
I(10)	ADDI(00001)	ADD	001
I(10)	LW(00010)	ADD	001
I(10)	SW(00011)	ADD	001
I(10)	BEQ(00100)	SUB	010
S(11)	SLL(00000)	SLL	011
S(11)	SLR(00001)	SLR	100
S(11)	SLLV(00010)	SLL	011
S(11)	SLRV(00011)	SLR	100

Following are the logic equations for the ALU operations:

1. AND = Type(00).Function(00000) + Type(10).Function(00000)
2. ADD = Type(00).Function(00001) + Type(10).Function(00001) + Type(10).Function(00010) + Type(10).Function(00011)
3. SUB = Type(00).Function(00010) + Type(00).Function(00011) + Type(10).Function(00100)
4. SLL = Type(11).Function(00000) + Type(11).Function(00010)
5. SLR = Type(11).Function(00001) + Type(11).Function(00011)

Following is ALU control unit interface that take Function and Type as input and generate ALUop that select the operation for ALU unit.

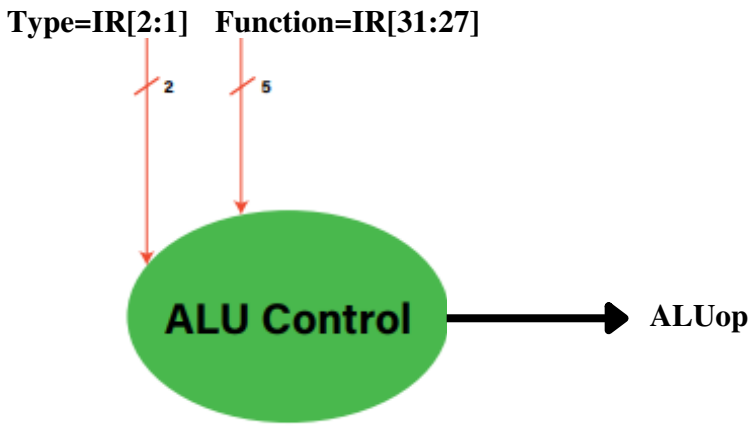


Figure 19: ALU Control Unit

4.3.2 Main_Control Unit

The Main_Control Unit, is a core unit in the implemented datapath, it is responsible for generating various signals responsible for the correct coordination of all used multiplexers, the inputs to different units and some of the work done by different units. It takes as input, the Type and function fields of the instruction . Following is the Control unit truth table indicating the generated signals.

Table 3: Main Control Unit Truth Table

Type	Function	RegSrc	ExtOp	ALUSrc	MemRd	WBData
R(00)	AND(00000)	0 = Rs2	X	0 = BusB	0	0 = ALU
R(00)	ADD(00001)	0 = Rs2	X	0 = BusB	0	0 = ALU
R(00)	SUB(00010)	0 = Rs2	X	0 = BusB	0	0 = ALU
R(00)	CMP(00011)	0 = Rs2	X	0 = BusB	0	X
J(01)	J(00000)	X	X	X	0	X
J(01)	JAL(00001)	X	X	X	0	X
I(10)	ANDI(00000)	X	0 = Zero	1 = Imm	0	0 = ALU
I(10)	ADDI(00001)	X	1 = Sign	1 = Imm	0	0 = ALU
I(10)	LW(00010)	X	1 = Sign	1 = Imm	1	1 = Mem
I(10)	SW(00011)	1 = Rd	1 = Sign	1 = Imm	0	X
I(10)	BEQ(00100)	1 = Rd	1 = Sign	0 = BusB	0	X
S(11)	SLL(00000)	X	X	2 = SA	0	0 = ALU
S(11)	SLR(00001)	X	X	2 = SA	0	0 = ALU
S(11)	SLLV(00010)	0 = Rs2	X	0 = BusB	0	0 = ALU
S(11)	SLRV(00011)	0 = Rs2	X	0 = BusB	0	0 = ALU

Following are the logic equations for the control all generated signals by the control unit.

RegSrc = SW + BEQ
ExtOp = LW + SW + BEQ
MemRd = LW
WBData = LW

IF (ANDI || ADDI || LW || SW):
        ALUSrc == 1
ELSE IF (SLL || SLR):
 ALUSrc == 2
ELSE:
        ALUSrc = 0

Following is Main_Control Unit interface that take Function and Type as input and generate RegSrc , ALUSrc , MemRd , WBData and ExtOp .

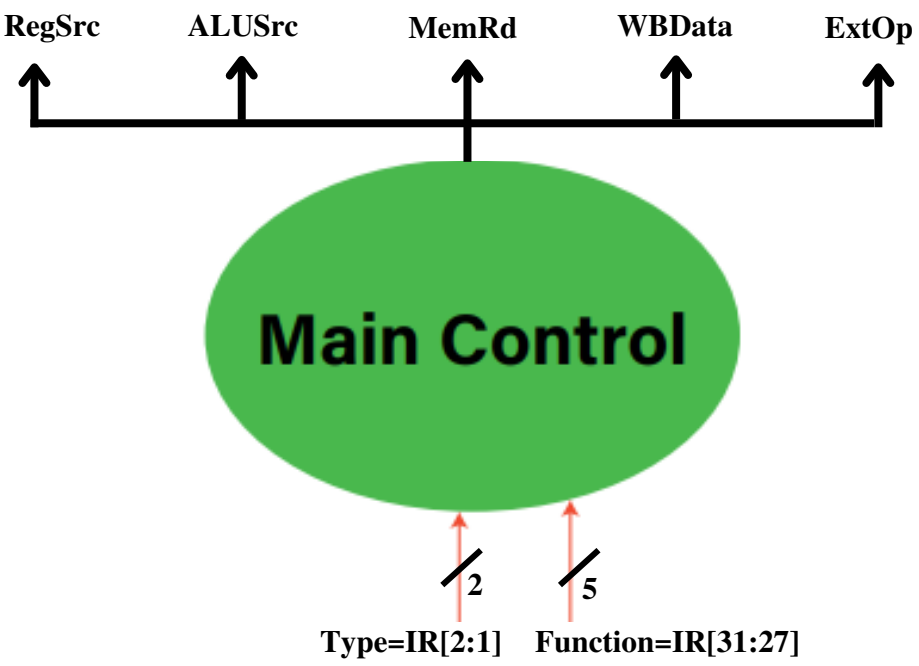


Figure 20: Main_Control Unit

4.3.3 PC_Control Unit

The PC_Control Unit, is a unit in the implemented datapath that decide which program counter will go to PC register , it is responsible for generating PCSrc signal . It takes as input Type , function fields , stop bit of the instruction and zero flag from ALU . Following is the PC Control unit truth table indicating the generated signal.

Table 4: PC Control Unit Truth Table

Type	Function	Zero	Stop	PcSrc
R(00)	AND(00000)	X	0	3
R(00)	ADD(00001)	X	0	3
R(00)	SUB(00010)	X	0	3
R(00)	CMP(00011)	X	0	3
J(01)	J(00000)	X	0	2
J(01)	JAL(00001)	X	0	2
I(10)	ANDI(00000)	X	0	3
I(10)	ADDI(00001)	X	0	3
I(10)	LW(00010)	X	0	3
I(10)	SW(00011)	X	0	3
I(10)	BEQ(00100)	1	0	1
S(11)	SLL(00000)	X	0	3
S(11)	SLR(00001)	X	0	3
S(11)	SLLV(00010)	X	0	3
S(11)	SLRV(00011)	X	0	3
X	X	X	1	0

Following is the logic that take by PC control unit to generate PCSrc signal .

```
IF (Stop = 1):
        PcSrc = 0
ELSE IF (BEQ && ZERO):
        PcSrc = 1
ELSE IF ((J || JAL) :
        PcSrc = 2
ELSE:
        PcSrc = 3
```

Following is PC_Control Unit interface that take Function , Type , stop bit and zero flag as input and generate PcSrc signal .

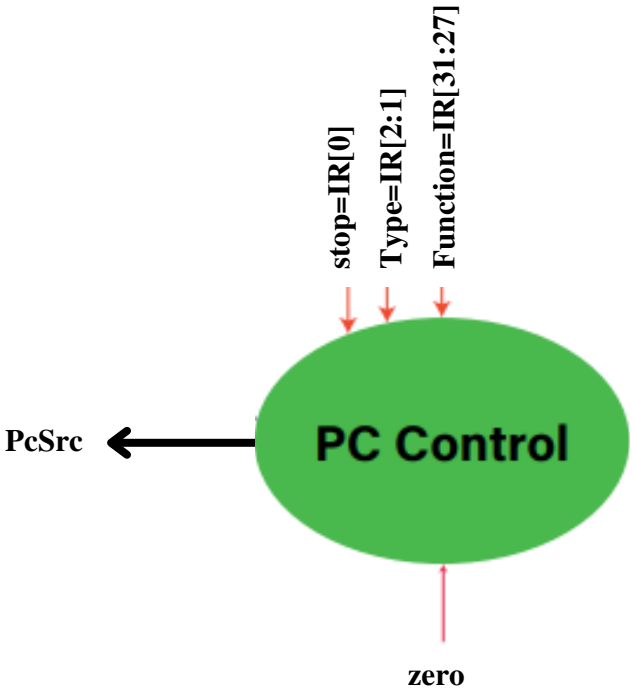


Figure 21: PC_Control Unit

4.3.4 Write_Control Unit

The Write_Control Unit, is a core unit in the implemented datapath, it is responsible for generating various signals responsible for writing in different storage elements. It takes as input Type , function fields and stop bit of the instruction . Following is the Write control unit truth table indicating the generated signals.

Table 5: Write Control Unit Truth Table

Type	Function	StackPush	StackPop	PcWr after cycles	RegWr	MemWr
R(00)	AND(00000)	0	Stop	3 + 1	1	0
R(00)	ADD(00001)	0	Stop	3 + 1	1	0
R(00)	SUB(00010)	0	Stop	3 + 1	1	0
R(00)	CMP(00011)	0	Stop	1 + 1	0	0
J(01)	J(00000)	0	Stop	0 + 1	0	0
J(01)	JAL(00001)	1	Stop	0 + 1	0	0
I(10)	ANDI(00000)	0	Stop	3 + 1	1	0
I(10)	ADDI(00001)	0	Stop	3 + 1	1	0
I(10)	LW(00010)	0	Stop	3 + 1	1	0
I(10)	SW(00011)	0	Stop	2 + 1	0	1
I(10)	BEQ(00100)	0	Stop	1 + 1	0	0
S(11)	SLL(00000)	0	Stop	3 + 1	1	0
S(11)	SLR(00001)	0	Stop	3 + 1	1	0
S(11)	SLLV(00010)	0	Stop	3 + 1	1	0
S(11)	SLRV(00011)	0	Stop	3 + 1	1	0

We can see from previous table that PcWr signal for all instructions contain two numbers , first one is number of cycles that specific instruction need after decode stage and second one is number of cycles that PC register to write new address for new instruction on it .

Following is Write_Control Unit interface that take Function , Type and stop bit as input and generate RegWr , MemWr , PcWr , StackPush and StackPop signals .

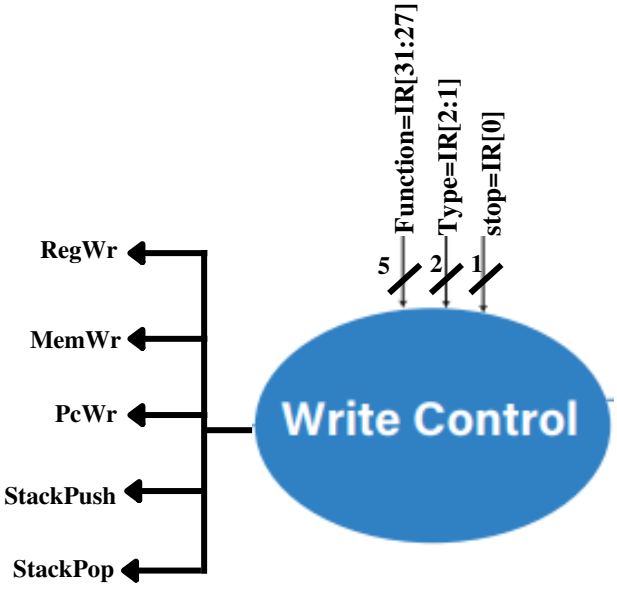


Figure 22: Write_Control Unit

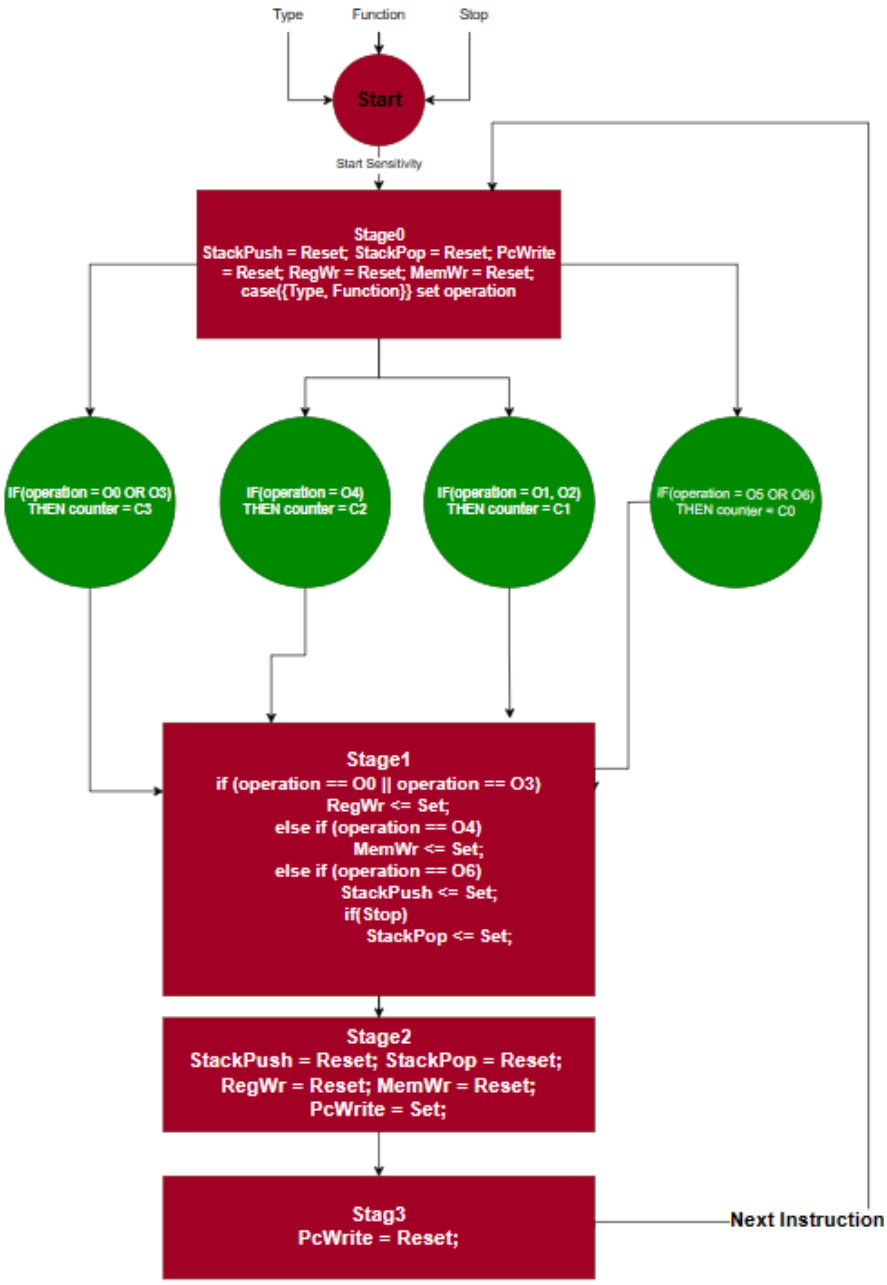


Figure 23: Finite state write control for multi cycle datapath

We can see from state diagram that instructions placed in groups , the decision is made about the number of cycles needed by the instruction based on the group to which the instruction belongs and set the control signals, as following :

1. O0 (AND, ADD, SUB, ANDI, ADDI, SLL, SLLV, SLR, SLRV)
2. O1 (CMP)
3. O2 (BEQ)
4. O3 (LW)
5. O4 (SW)
6. O5 (J)
7. O6 (JAL)

4.4 Final Datapath

The following figure is the final datapath after adding all compensational , storage and control units .

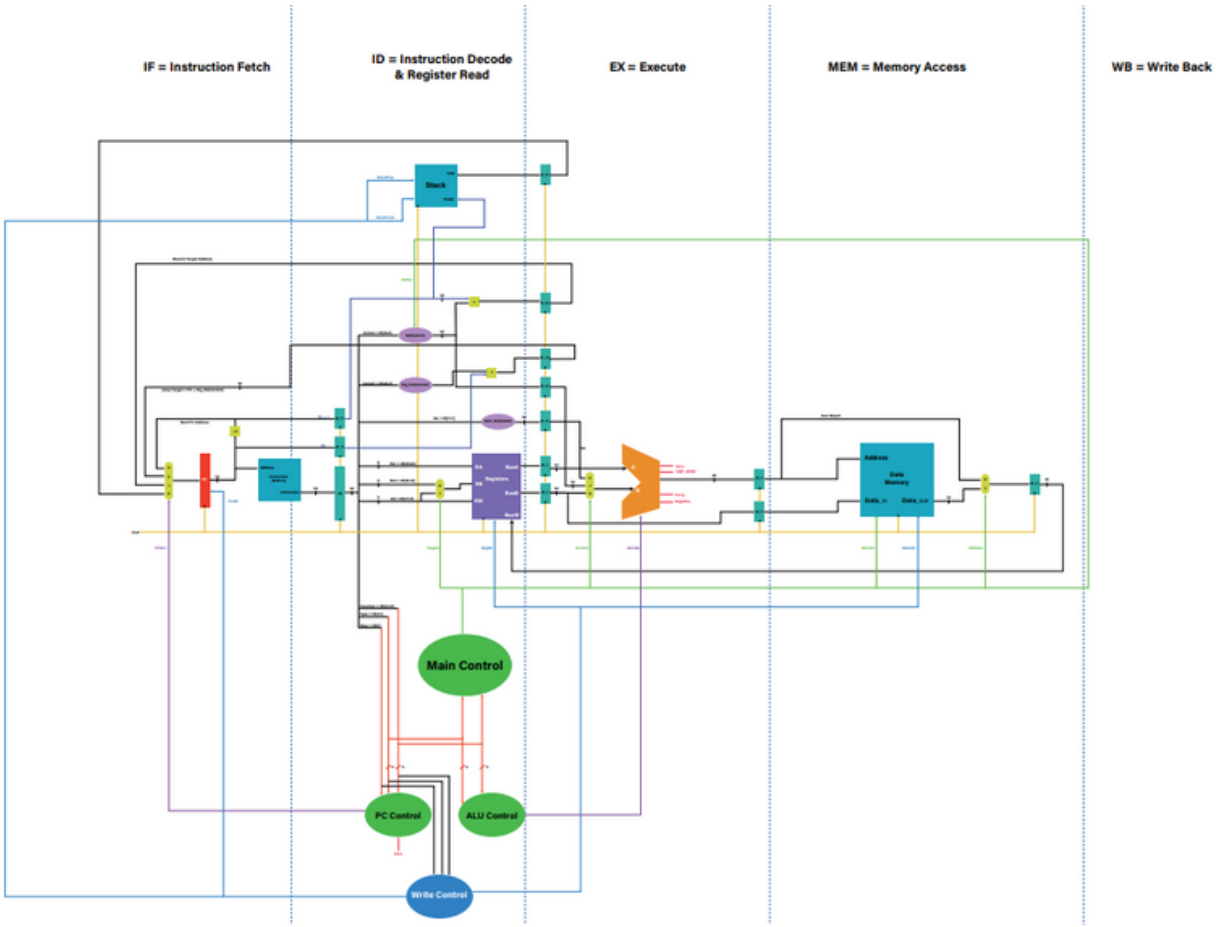


Figure 24: Final Datapath

5 SIMULATION AND TESTING

In this section, many test cases are provided to demonstrate our correct implementation of all components .

The register file was initialized with the following values:
1. cpu.register_file.Registers[0] = 32'h00000000;
2.cpu.register_file.Registers[1] = 32'h00000001;
3.cpu.register_file.Registers[2] = 32'h00000002;
4.cpu.register_file.Registers[3] = 32'h00000003;
5.cpu.register_file.Registers[4] = 32'h00000004;
6.cpu.register_file.Registers[5] = 32'h00000005;
7.cpu.register_file.Registers[6] = 32'h00000001;

The memory was initialized with the following values:
1.cpu.memory.Mem[0] = 32'h00000001;
2.cpu.memory.Mem[1] = 32'h00000005;
3.cpu.memory.Mem[2] = 32'h00000007;
4.cpu.memory.Mem[3] = 32'h00000004;
5.cpu.memory.Mem[4] = 32'h00000008;

5.1 ADD , ADDI , SUB and CMP

The following instructions were written in the instruction memory:

```
// instructions
cpu.instruction_memory.Mem[0] = 32'b00001_00010_00111_00011_000000000_00_0; // ADD R7, R2, R3    : R2 = 2, R3 = 3
cpu.instruction_memory.Mem[1] = 32'b00010_00111_00110_00011_000000000_00_0; // SUB R6, R7, R3    : R7 = 5, R3 = 3
cpu.instruction_memory.Mem[2] = 32'b00000_00110_01000_00000000001111_10_0; // ADDI R8, R6, 15   : R6 = 2
cpu.instruction_memory.Mem[3] = 32'b00011_00010_01111_00011_000000000_00_0; // CMP R15, R2, R3   : R2 = 2, R3 = 3
```

Figure 25: ADD , ANDI , SUB and CMP

The following figure show true simulation waves for ADD instruction ,
R7 = ALU_R = 5 = R2+R3 = 2+3 .

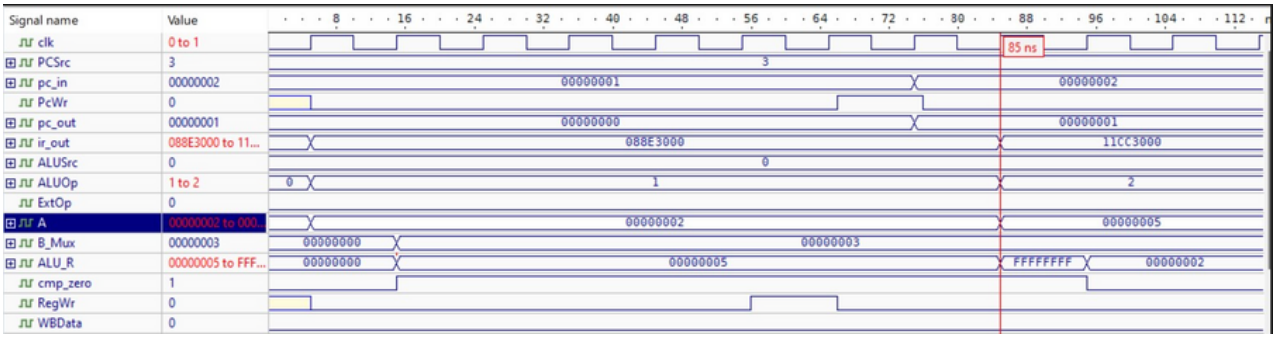


Figure 26: ADD simulation

The following figure show true simulation waves for SUB instruction ,
R6 = ALU_R = 2 = R7-R3 = 5-3 .

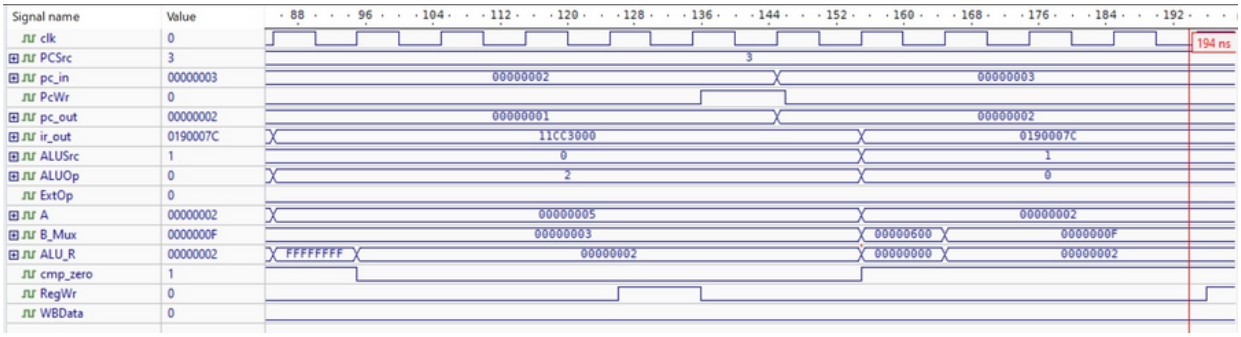


Figure 27: SUB simulation

The following figure show true simulation for the following instruction:

1. R8 = ALU_R = R6&F = 2&F =2
2. CMP instruction for R2=2 and R3=3 so CMP_zero =1 because A<B

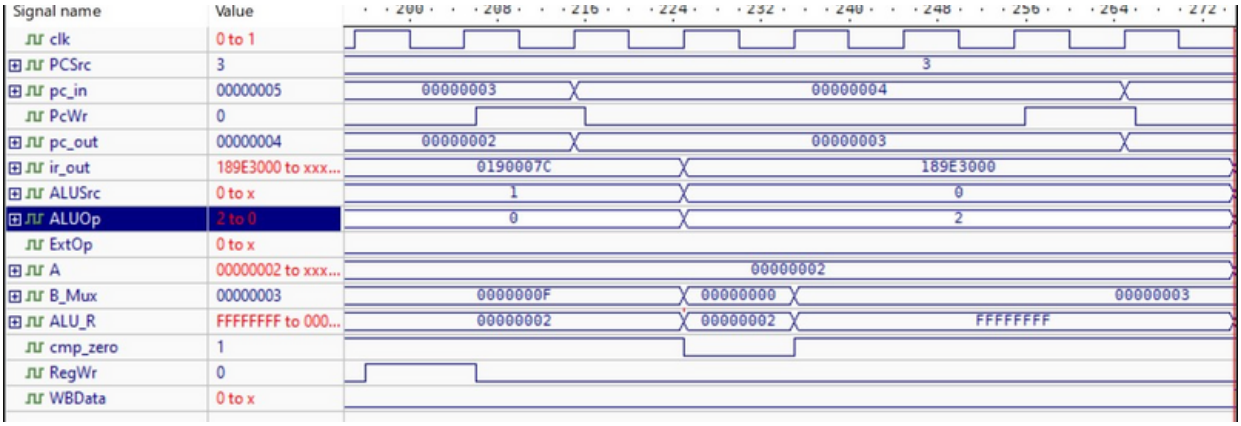


Figure 28: ANDI and CMP simulation

5.2 JAL , NOP and ADD

The following instructions were written in the instruction memory:

```
// JAL F
// NOP
// F: ADD R7, R2, R3 : STOP BIT
cpu.instruction_memory.Mem[0] = 32'b00001_00000000000000000000010_01_0; // JAL F
cpu.instruction_memory.Mem[1] = 32'b00000000000000000000000000000000;  // NOP
cpu.instruction_memory.Mem[2] = 32'b00001_00010_00111_00011_000000000_00_1; // ADD R7, R2, R3
```

Figure 29: JAL , NOP and ADD

The following figures show true simulation waves for JAL instruction , it pushed pc+1 to stack (StackPush signal = 1) , then it jumped to ADD as the following equation:
R7 = ALU_R = 5 = R2+R3 = 2+3 .
Finally , it detect stop = 1 so it popped pc+1 from stack (StackPop signal = 1) to jump to the return address (NOP instruction).

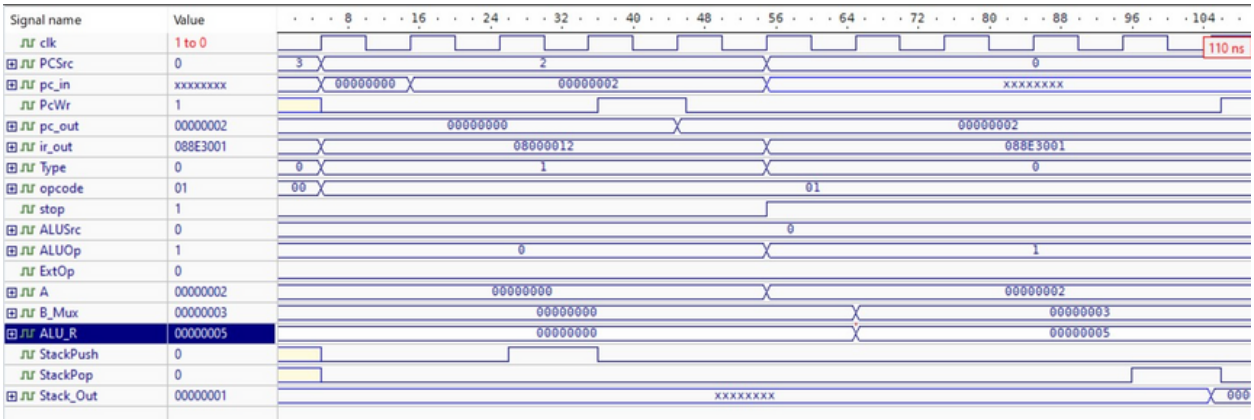


Figure 30: JAL , NOP and ADD simulation 1

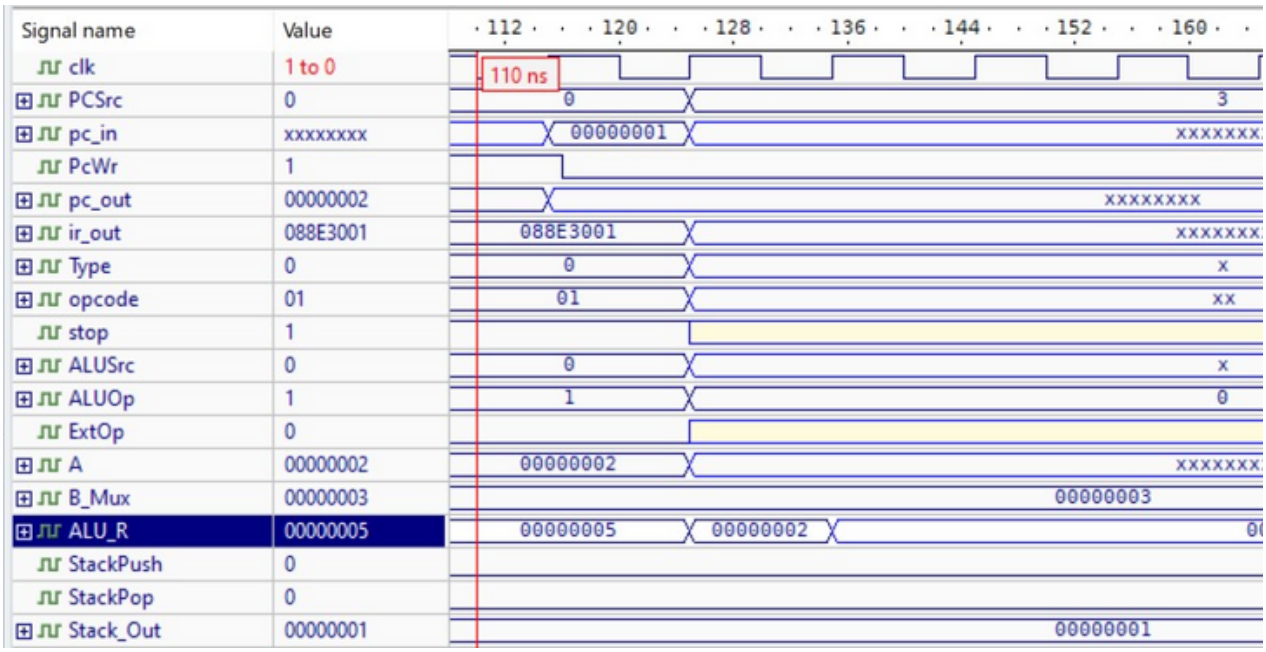


Figure 31: JAL , NOP and ADD simulation 2

5.3 SW and LW

The following instructions were written in the instruction memory:

```
// SW then LW
cpu.instruction_memory.Mem[0] = 32'b00011_00000_00011_00000000000100_10_0; // SW R3, 4(R0)   : R3 = 3
cpu.instruction_memory.Mem[1] = 32'b00010_00000_00101_00000000000100_10_0; // LW R5, 4(R0)   : 4(R0) = 3
```

Figure 32: SW and LW

The following figure show true simulation waves for SW and LW instructions as following :

1. ALU calculate the address of memory to store R3=3 on it so ALU_R= R0+4 = 0+4 = 4 then store the value of R3 on Memory[4] (MemWr=1).
2. ALU calculate the address of memory that we want to read from it so ALU_R= R0+4 = 0+4 = 4 then read the value of Memory[4] (MemRd=1 , Mem_Out=3) and load it on R5.

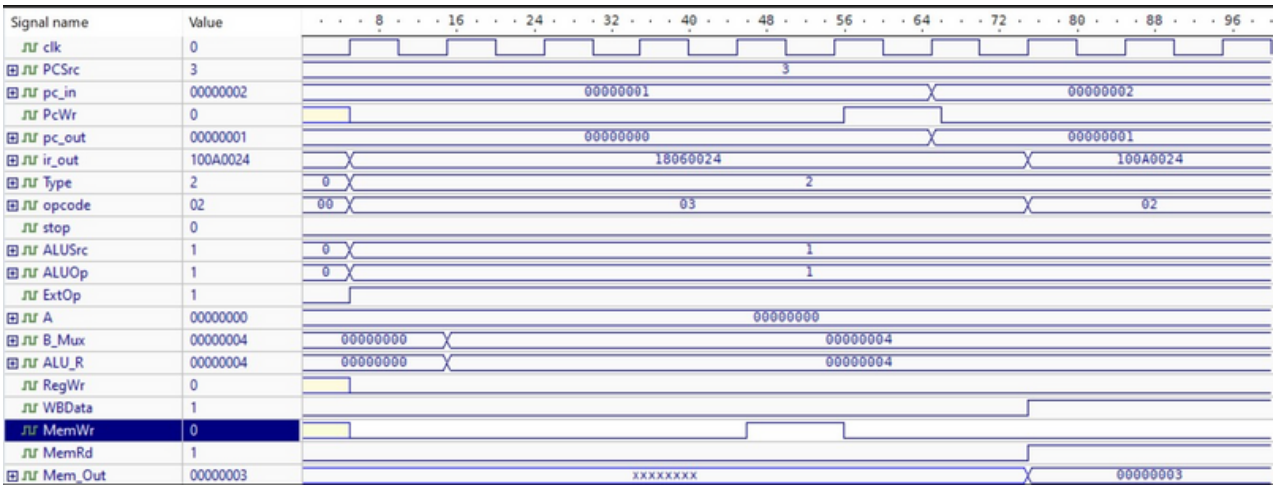


Figure 33: SW and LW simulation

5.4 BEQ

The following instructions were written in the instruction memory:

```
// BEQ
PC_OLD = 0, PC_1 = PC_OLD + 1 = 1, PC_BRANCH = PC_1 + SIGN(IMM14) = 1 + 4 = 5
cpu.instruction_memory.Mem[0] = 32'b00100_00001_00110_00000000000100_10_0; // BEQ R1, R6     : R1 = 1, R6 = 1
```

Figure 34: BEQ

The following figure show true simulation waves for BEQ instruction , R1=R6=1 so zero flag equal 1 , it will branch to the target address (pc_in=5).

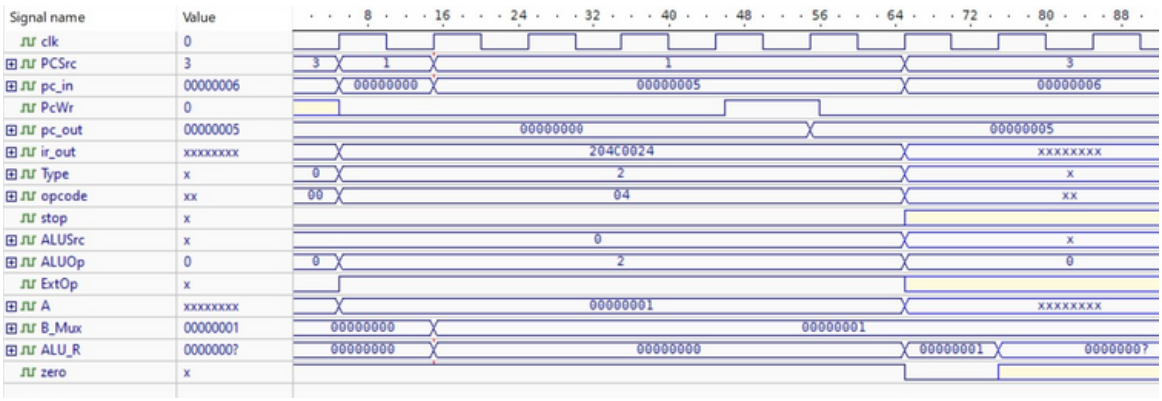


Figure 35: BEQ simulation

5.5 SLL and SLRV

The following instructions were written in the instruction memory:

```
// SLL and SLRV
cpu.instruction_memory.Mem[0] = 32'b00000_00010_00111_00000_00010_0000_11_0; // SLL R7, R2, 2   : R2 = 2, R7 = R2 << 2
cpu.instruction_memory.Mem[1] = 32'b00011_00111_01000_00010_00000_0000_11_0; // SLRV R8, R7, R2 : R7 = 8, R2 = 2, R8 = R7 >> R2
```

Figure 36: SLL and SLRV

The following figure show true simulation waves for SLL and SLRV instructions as following:

1. R2=2 =8'b00000010 it shifted by 2 to the left so R7=8'b00001000 = 4 (ALU_R=8).
2. R7=2 =8'b00001000 it shifted by R2=2 to the right so R8=8'b00000010 = 2 (ALU_R=2).

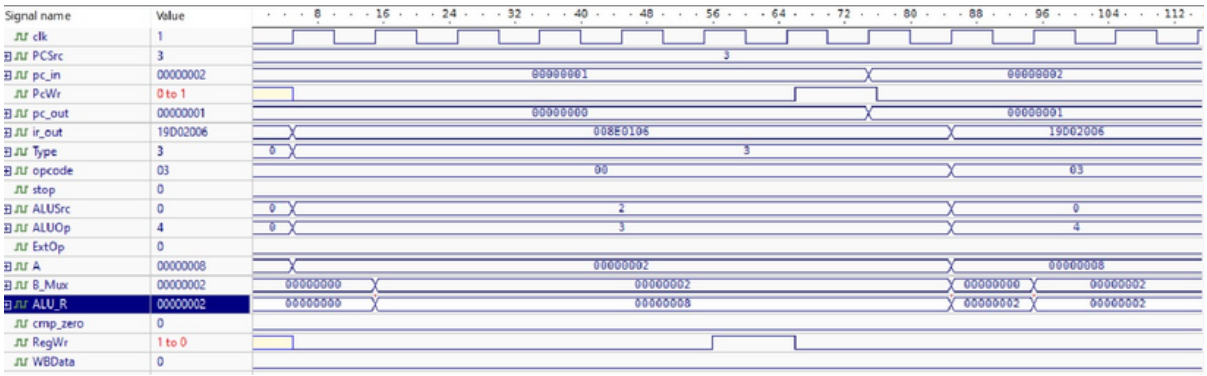


Figure 37: SLL and SLRV simulation

6 CONCLUSION AND FUTURE WORK

In this project we have learned a lot about the multicycle structure and the implementation of it. We have succeed in implementing a multicycle 32-bit processor using Verilog language with all the aims that were required in this project.
As a future plan, this project encourages us to dive more and more in computer architecture techniques and develop this structure to be pipeline structure to implement the forwarding technique and deal with hazard detection .