Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Computer Architecture (ENCS4370)

Project Report

Multi-Cycle RISC Processor

Prepared by: **ID :**

Salwa Fayyad	**1200430**
Sami Moqbel	**1200751**
Lama Abuthaher	**1201138**

Instructor : Dr. Aziz Qaroush

Section : 2

Date : 18-06-2024

Abstract

The goal of this project is to design and evaluate a simple multi-cycle RISC processor. This processor features 8 general-purpose registers, each 16 bits wide, and employs two separate memory units: one for instructions and another for data memory. The processor supports four different instruction types: R, I, J, and S types. The design and testing of the processor were implemented using Verilog HDL in a behavioral manner.

Table of Contents

Table Of Figure

Design and Implementation

1. Data Path

The data path design is for a multi-cycle processor which have five stages: Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (EX), Memory Access (MA) and Write Back (WB). For this processor, all components are controlled by a common synchronized clock and triggered by the positive edge. It was assembled using the following components:

- **Program Counter (PC)**

The program counter (PC) was used to store the address of the next instruction that would be fetched. Figure 1 shows a 16-bit PC register, a mux 4x1 that was used to choose the PC input according to the 2-bits PC source signal that generated by PC control, a plus 1 component to make the normal update on the PC and the wires between the components. The PC input have four choices:

- The normal PC (PC+1).
- The branch target address (BTA) which is (PC+sign immediate$_{16}$).
- The jump target address (JTA) which is (PC [15:12] || immediate$_{12}$).
- A PC for the RET instruction which is (PC = R7).

These choices were made to support for different instruction types.

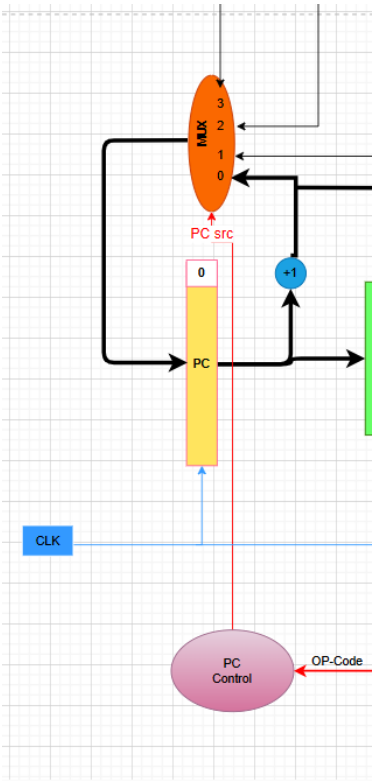

Figure 1:Program Counter (PC)

- **Instructions Memory**

The instruction memory component is a physical memory unit used to store instructions. The data path supports four instruction types (R-type, I-type, J-type, and S-type), each 16 bits in size. Figure 2 illustrates the instruction memory component, which has a single 16-bit address input and a 16-bit instruction output. Additionally, it includes three buffers (one for the PC and two for instructions) to store the results of the current stage (Instruction Fetch, IF) for use in the next stage (Instruction Decode, ID). These buffers are controlled by a common synchronous clock. At the end of this process, the instruction will be fetched.

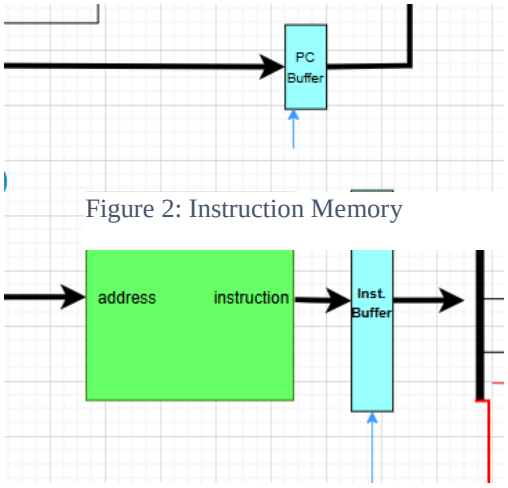

Figure 2: Instruction Memory

```
reg signed [15:0] Data_Memory[0:127]; // 16-bit address memory,
```

Figure 3: Instruction Memory Code-Part One

```
// -------------------- InstructionMemory --------------------------
task InstructionMemory(input [15:0] instAdd, output reg [15:0] instOut);

    instOut = Memory[instAdd];

    $display("Instruction = %b\n", instOut);
endtask
```

Figure 4: Instruction Memory Code-Part two

- ## Registers File

The registers file has 8-general purpose registers (R_0– R_7), each one has a 16-bit size. After the instruction decoded, it was used to read the operands and to write the result on a specific destination register. Figure 5 shows the registers file with eight inputs as following:

- **Rs1**: The first source which takes the address of the first operand.
- **Rs2**: The second source which takes the address of the second operand.
- **Rd**: The destination which takes the address of the destination register.
- **BusW**: The write data bus which takes the result in the WB stage and write it on the destination register when the RegW is one.
- **RegW**: A signal to enable the write process on the register file. When it is one, the write process allowed. It was generated by the main control.
- **Clk**: To make the registers file controlled by a common synchronous clock.

Additionally, the instruction memory component has two outputs: BusA and BusB. It also includes an extender to expand a 5-bit immediate value to 16 bits based on the **ExtOp** signal, which determines whether the extension process is signed or unsigned. A multiplexer, controlled by the **ExtImm** signal, selects between the 16-bit extended immediate or the original 8-bit immediate before extension. Moreover, an adder is incorporated to compute the Branch Target Address (BTA). As illustrated in the figure, buffers are used between stages to store the results of the current stage (Instruction Decode, ID) for use in the next stage (Execution, EX) and the previous stage (Instruction Fetch, IF).

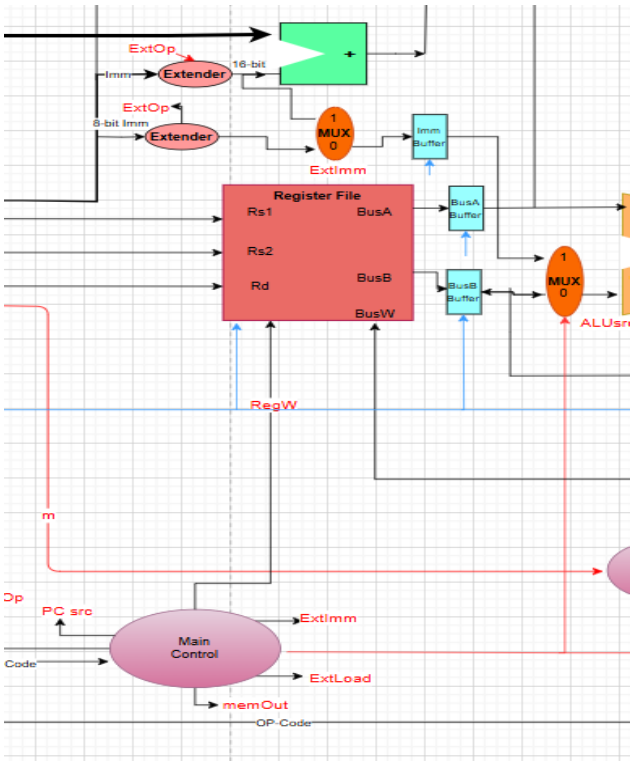

Figure 5: Registers File and the Components of the ID Stage

```
// -------------------- Register Read -------------------------
task RegsiterFileRead(input [2:0] Rs1, Rs2, output signed [15:0] BusA, BusB);
    BusA = Registers[Rs1];

    if (IR[15:12] >= 4'b1000 && IR[15:12] <= 4'b1011) // Branch instructions
        begin
            if(!IR[11])// if  mode = 0 then Compare With RS2
                begin
                    BusB = Registers[Rs2];
                end
            else // Compare With R0
                begin
                    BusB = Registers[0];
                end
        end
    else
        begin BusB = Registers[Rs2]; end


endtask
```

Figure 6: Register Read Code

```
// -------------------- Register Write -------------------------
task RegsiterFileWrite(input [2:0] Rd,  input RegW, input signed [15:0] BusW);

    if(RegW)
        Registers[Rd] = BusW;

    if(RegW)
        $display("The register R%0d updated new value = %2d\n", Rd, Registers[Rd]);
endtask
```

Figure 7: Register Write Code

- **Sign Extender**

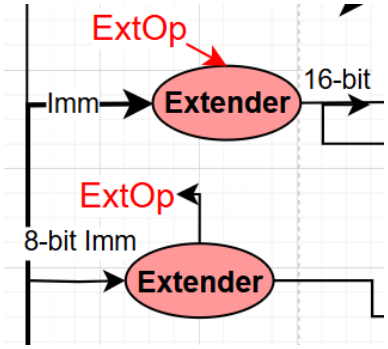

Figure 8; Sign_Extender

The extender's role is to expand an immediate number, either 8-bit or 5-bit, into a 16-bit number. There are two extenders: one extends an 8-bit input to 16 bits, and the other extends a 5-bit input to 16 bits. Each extender has two inputs: the immediate value to be extended and a signal specifying whether the extension is signed or unsigned. Both extenders have a single 16-bit output, which is the extended number.

```
///----------------------5 -> 16-bit EXTENDER---------------
task Extender1(input [4:0]imm, input ext_op, output signed [15:0]outExt);
    if(ext_op)
        begin
            if(imm[4]==1'b1)
                outExt={11'b11111111111,imm}; // the sign extended is ones
            else
                outExt={11'b0,imm};
        end
    else
        outExt={11'b0,imm};

endtask

///----------------------8 -> 16-bit EXTENDER---------------
task Extender2(input [7:0]imm, input ext_op, output signed [15:0]outExt);
    if(ext_op)
        begin
            if(imm[7]==1'b1)
                outExt={8'b11111111,imm}; // the sign extended is ones
            else
                outExt={8'b0,imm};
        end
    else
        outExt={8'b0,imm};

endtask
```

Figure 9: Sign_Extender implementation

- ## Arithmetic & Logical Unit (ALU)

The ALU integrated into this processor supports three primary operations: addition, subtraction, and bitwise AND. These operations cater to a wide range of instructions. The ALU has two 16-bit inputs, which are the operands, and a 2-bit input specifying the operation to be executed (add, and, sub), generated by the control unit. The first operand always comes from Bus A in the register file. The second operand can originate from:

- Bus B: used when processing R-type instructions, where the second operand resides in a register.
- Extended immediate: used for I-type instructions, where the second operand is derived from an 8-bit or 5-bit immediate value stored in the instruction register and extended to 16 bits (considering signed extension).

The selection between Bus B and Extended immediate is determined by an input signal called **ALUsrc**, which controls a 2x1 MUX to choose the appropriate operand.

The ALU in this processor produces a 16-bit output representing the result of the operation, along with three 1-bit flags: zero, negative, and overflow. These flags are crucial for handling all four branch instructions (BEQ, BNE, BGT, BLT).

- BEQ and BNE: These instructions utilize the zero flag to determine the condition. BEQ branches if the operands are equal (zero flag set), while BNE branches if they are not equal (zero flag clear).
- BGT and BLT: For these instructions, the negative and overflow flags are used. Despite the instruction description stating that BGT branches if (Reg(Rd) > Reg(Rs1)), the ALU's first operand is Reg[Rs1]. Therefore, BGT checks if the

first operand is less than the second operand, and BLT checks if the first operand is greater than the second operand.

These adjustments ensure that the branch instructions correctly interpret the results from the ALU operations within the processor architecture.

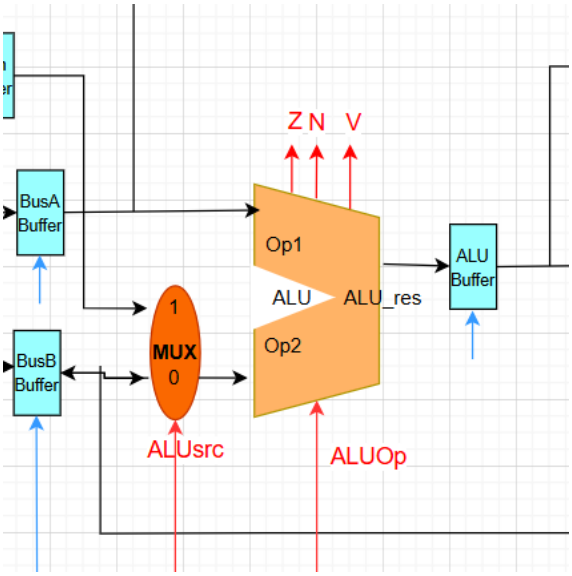

Figure 10: ALU Diagram in Execution Stage

```
task ALU (input [1:0] ALUOp, input signed [15:0] op1, input signed [15:0] op2, output reg signed [15:0]result, output reg z, n, v);

    // compute result
    case(ALUOp)
        `AND: begin result = op1 & op2; end
        `ADD: begin result = op1 + op2; end
        `SUB: begin result = op1 - op2; end
    endcase

    // set flags
    z = result == 16'd0;
    n = result < 0;
    v = (op1[15] != op2[15]) && (result[15] != op1[15]);

    $display("op1= %2d, op2=%2d\n", op1, op2);
    $display("ALU result= %2d, z=%b, n=%b, v=%b\n", result, z, n, v);
endtask
```

Figure 11: ALU Implementation

- ## Adder for BTA

Since the implementation of this processor does not support the instruction to use the same component in the data path more than once, an additional adder was added to the data path to calculate the branch target adders, while the ALU is handling the condition checking of the branch instruction.

Inputs and outputs for adder are:

- 16-bit input with the value of the PC.
- 16-bit input immediate value after being sign extended.
- 16-bit output of the branch target address that feeds in the 4x1 MUX of PC source.

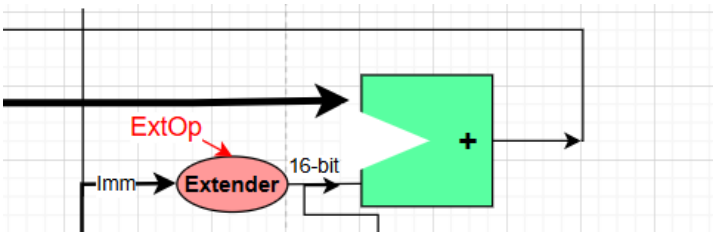

Figure 12: Adder for BTA in Decode Stage

```
if (zeroFlag == 1'b1)
    PC = PC + extendedImm; //branch taken --> PC = BTA
else
    PC = PC + 1;
```

Figure 13: Adder for BTA Implementation

- **Data Memory**

Data is stored within the data memory, which consists of 127 words, each 16 bits wide. This memory unit operates with the following inputs:

- Address (16-bit): Specifies the location in memory to access.
- Data_in (16-bit): Data to be written into the memory at the specified address, used during store or call instructions.
- MemR (signal): Signal indicating a read operation from memory, generated by the Main Control unit.
- MemW (signal): Signal indicating a write operation to memory, generated by the Main Control unit.

The memory provides a single 16-bit output, which holds the data read from memory when a read operation is performed.

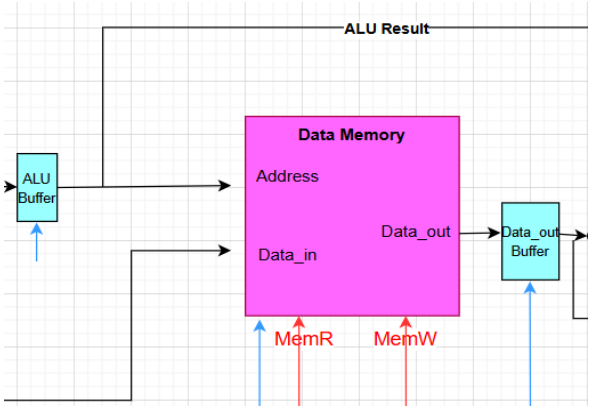

Figure 14: Data Memory in Memory Stage

```
// variable to save address used in memory, data in and data out from memory
reg signed [15:0] Data_Memory[0:127]; // 16-bit address memory, also this size is temporary and can be changed

reg signed [7:0] Memaddress; // ( SHOULD BE CHECKED )
reg signed [15:0] dataIn;
output reg signed [15:0] dataOut;
output reg signed [15:0] WBData; // Data be written to Rigster file (last MUX)

// -------------------- Data Memory --------------------------
task dataMem(input [15:0] Address, input signed [15:0] Data_in, input MemR, MemW, output signed [15:0] Data_out);

    if(IR[15:12] == 4'b0101 || IR[15:12] == 4'b0111 ) // LW and SW
        begin
            if(MemR==1'b1 && MemW==1'b0)
                Data_out=Data_Memory[Address];
            else if(MemR==1'b0 && MemW==1'b1)
                Data_Memory[Address]=Data_in;
        end
    else //Lbu , LBs
        begin
            fullData = Data_Memory[Address];
            byteData = fullData[7:0];
            byteExtender(byteData, ExtLoad, Data_out);
        end

endtask
```

Figure 15: Data memory implementation.

- **Byte Extender**

The byte extender handles byte extension operations. It includes a 2x1 multiplexer to choose between sign-extending (LBs) or zero-extending (LBu) a byte after a memory read. This ensures the processor correctly interprets and extends byte data for different instructions.

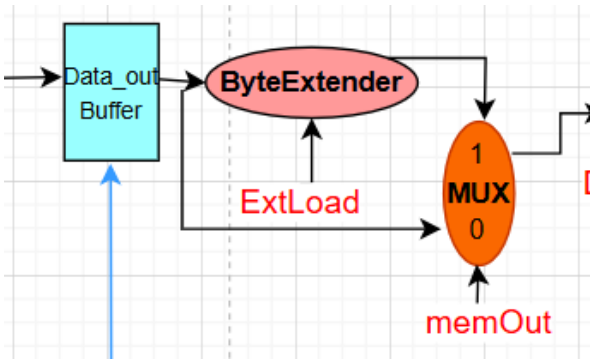

Figure 16: Byte Extender

```
    reg signed [15:0] fullData;
    reg signed [8:0] byteData;

task byteExtender(input [7:0] byteOut, input ExtLoad, output signed [15:0] loadOut);

    if(ExtLoad) // LBs
        begin
            if(byteOut[7]==1'b1)
                loadOut={8'b11111111,byteOut}; // the sign extended is ones
            else
                loadOut={8'b0,byteOut};
        end
    else // LBu
        loadOut={8'b0,byteOut};
endtask
```

Figure 17: Byte Extender implementation.

- **Write Back Stage**

In this stage, the output of the ALU or the output of the memory is stored back, and written on the register file, the choice between ALU output and memory output is done by using a 2x1 Mux that selects based on a signal called **DataWB**, if the data written back is from memory or ALU output. For the data to be written on the register file, the control signal **RegW** must be set ('1').

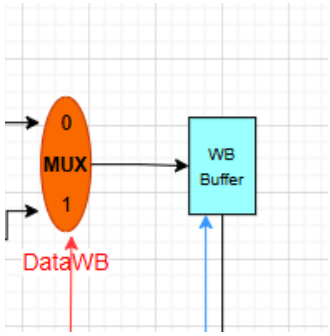

Figure 18: Write Back Stage.

2. Control Signals

We generate control signals from three different units: the main control unit, the PC control unit, and the ALU control unit. Dividing the control signals among these units simplifies understanding and enhances efficiency in building and testing each signal. Control signals are essential for activating the appropriate functional units for each instruction. For example, in R-type instructions, the write-back data comes from the ALU, whereas in load instructions, it comes from the memory. In summary, control signals are crucial for the correct operation of our processor, helping it distinguish between different instructions.

- **PC Control**

This control selects the value of the PC as follows: PC + 1 for normal execution, BTA when a branch is taken, and JTA for jump or call instructions. For branches, the ALU performs a subtraction, and the decision is based on the output flags.

- BEQ and BNE: Use the zero flag. BEQ is taken when Z == 1, and BNE is taken when Z == 0.
- BGT and BLT: To determine if Rd > Rs1, we check if Rs1 < Rd by reversing the operands without extra hardware. This keeps Rs1 as the first ALU operand and Rd as the second. For BGT, the branch is taken if N != V, and for BLT, if (Z == 0 && N == V).

This mechanism ensures the ALU's first operand is always Rs1, and the second is Rd, maintaining consistency in branch condition evaluation.

Inst	Z	N	V	PCSrc
BGT	X	1	0	1
BGT	X	0	1	1
BGT	X	0	0	0
BGT	X	1	1	0
BGTZ	X	1	0	1
BGTZ	X	0	1	1
BGTZ	X	0	0	0
BGTZ	X	1	1	0
BLT	0	0	0	1
BLT	0	1	1	1
BLT	1	X	X	0
BLTZ	0	0	0	1
BLTZ	0	1	1	1
BLTZ	1	X	X	0
BEQ	1	X	X	1
BEQ	0	X	X	0
BEQZ	1	X	X	1
BEQZ	0	X	X	0
BNE	0	X	X	1 (T)
BNE	1	X	X	0 (NT)
BNEZ	0	X	X	1 (T)
BNEZ	1	X	X	0 (NT)
JMP	X	X	X	2
CALL	X	X	X	2
RET	X	X	X	3
Others	X	X	X	0

If (OpCode == JMP || OpCode == CALL) **PCsrc** = 2; (Jump Target Address)
Else if (OPCode == RET) **PCsrc** = 3;
Else if ((OPCode == BGT && N != V) || (OPCode == BLT && Z == 0 && N == V) || (OPCode == BEQ && Z == 1) || (OPCode == BNE && Z == 0)) **PCsrc** = 1; (Branch Target Address)
Else **PCsrc** = 0; (next PC)

Figure 19: PC Truth Table with Boolean equation.

- **Main Control**

This unit is crucial for most instructions as it generates essential signals used in the decode, execute, memory, and write-back stages. Key signals include:

- **ExtOp**: Determines whether the immediate is extended as signed or unsigned.
- **ExtImm**: Decides if the second immediate is extended from 5 bits to 16 bits or from 8 bits to 16 bits.
- **ExtLoad**: Decides if a byte should be sign-extended (LBs) or zero-extended (LBu).
- **RegW**: Enables writing the value from BusW to Rd.
- **ALUsrc**: Determines if the ALU's second operand is a register or an extended immediate.
- **MemR**: Enables reading from memory.
- **MemW**: Enables writing to memory.
- **DataWB**: Decides if the data being written back is from memory output or ALU result.

These signals ensure proper operation and data flow throughout the instruction processing stages.

Inst	Extop	ALUSRC	REGW	MEMR	MEMW	DATAWB	ExtImm	ExtLoad	memOut
AND	**X**	0	1	0	0	0	**X**	**X**	**X**
ADD	**X**	0	1	0	0	0	**X**	**X**	**X**
SUB	**X**	0	1	0	0	0	**X**	**X**	**X**
ADDI	1	1	1	0	0	0	0	**X**	**X**
ANDI	0	1	1	0	0	0	0	**X**	**X**
LW	1	1	1	1	0	1	0	**X**	0
LBu	1	1	1	1	0	1	0	0	1
LBs	1	1	1	1	0	1	0	1	1
SW	1	1	0	0	1	**X**	0	**X**	**X**
BGT	1	0	0	0	0	**X**	0	**X**	**X**
BGTZ	1	0	0	0	0	**X**	0	**X**	**X**
BLT	1	0	0	0	0	**X**	0	**X**	**X**
BLTZ	1	0	0	0	0	**X**	0	**X**	**X**
BEQ	1	0	0	0	0	**X**	0	**X**	**X**
BEQZ	1	0	0	0	0	**X**	0	**X**	**X**
BNE	1	0	0	0	0	**X**	0	**X**	**X**
BNEZ	1	0	0	0	0	**X**	0	**X**	**X**
JMP	**X**	**X**	0	0	0	**X**	**X**	**X**	**X**
CALL	**X**	**X**	1	0	0	**X**	**X**	**X**	**X**
RET	**X**	0	0	0	0	**X**	**X**	**X**	**X**
Sv	1	1	0	0	1	**X**	1	**X**	**X**

ExtOp = ADDI + LW + Lbu + LBs + SW + BGT + BGTZ + BLT + BLTZ + BEQ + BEQZ + BNE + BNEZ + Sv

ALUSrc = ADDI + ANDI + LW + Lbu + LBs + SW + Sv

REGW = AND + ADD + SUB +ADDI + ANDI + LW + Lbu + LBs

MEMR = LW + Lbu + Lbs

MW = SW + Sv

DataWB = LW + Lbu + Lbs

ExtImm = Sv

ExtILoad = Lbs

MemOut = Lbu + Lbs

Figure 20: Main control truth table with boolean equations

- ## ALU Control

This unit decides which operation the ALU should perform: logical AND, arithmetic addition, or arithmetic subtraction. These three operations are encoded using 2 bits:

- AND = 00

- ADD = 01

- SUB = 10

Each instruction is analyzed to determine the required ALU operation, and the appropriate signal is assigned according to this encoding, as shown in the following truth table.

Inst	ALUOP	Operation
AND	0	AND
ADD	1	ADD
SUB	2	SUB
ADDI	1	ADD
ANDI	2	ADD
LW	1	ADD
Lbu	1	ADD
LBs	1	ADD
SW	1	ADD
BGT	2	SUB
BGTZ	2	SUB
BLT	2	SUB
BLTZ	2	SUB
BEQ	2	SUB
BEQZ	2	SUB
BNE	2	SUB
BNEZ	2	SUB
JMP	X	X
CALL	X	X
RET	X	X
Sv	X	X

ALU Control Boolean Equations

AND = 00 --> AND + ANDI

ADD = 01 --> ADD + ADDI + LW + LBu + LBs + SW

SUB = 10 --> SUB + BGT + BGTZ + BLT + BLTZ + BEQ + BEQZ + BNE + BNEZ

Figure 21: ALU Truth table

ALU Control Boolean Equations

AND = 00 → AND + ANDI

ADD = 01 → ADD + ADDI + LW + LBu + LBs + SW

SUB = 10 → SUB + BGT + BGTZ + BLT + BLTZ + BEQ + BEQZ + BNE + BNEZ

3. Datapath Block Diagram

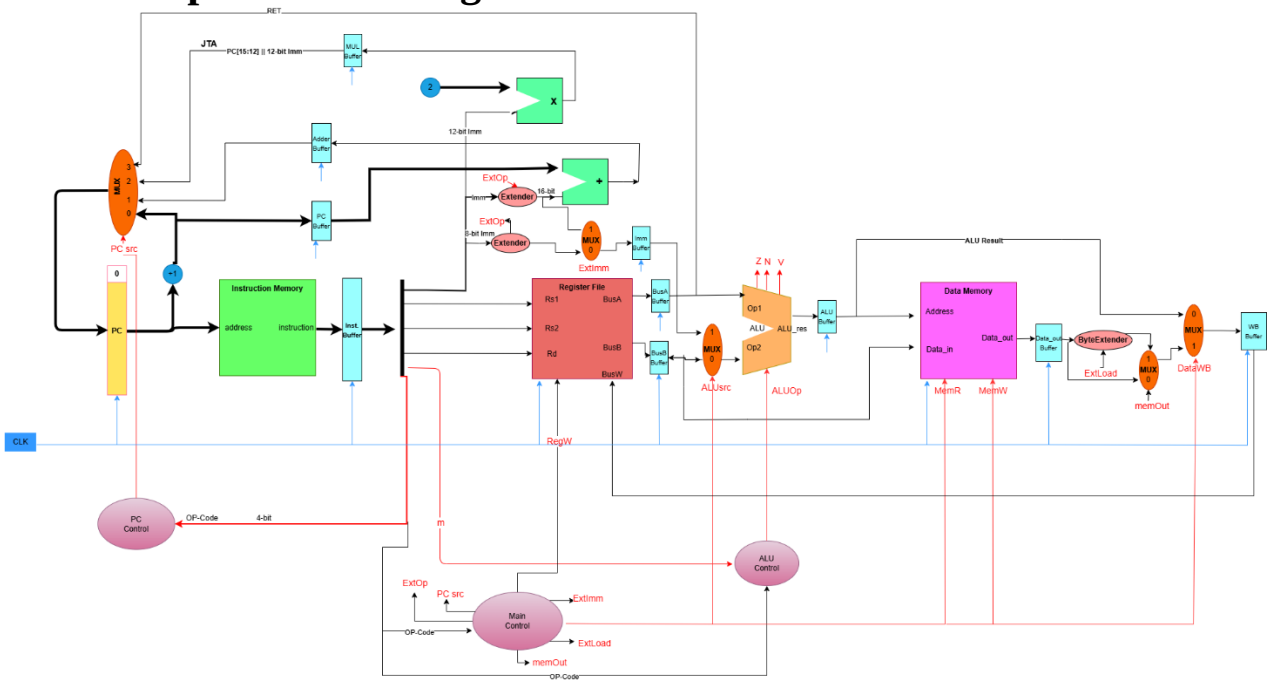

Figure 22: Datapath Block Diagram

We construct this datapath iteratively, starting with the functional units needed for R-type instructions, followed by those for I-type, J-type, and S-type instructions. To determine the required functional units for each type, we use register transfer language (RTL). This approach also helps us identify the number of cycles needed for each instruction to complete its operation.

4. Multi-cycle Processor Finite Stat Machine (FSM)

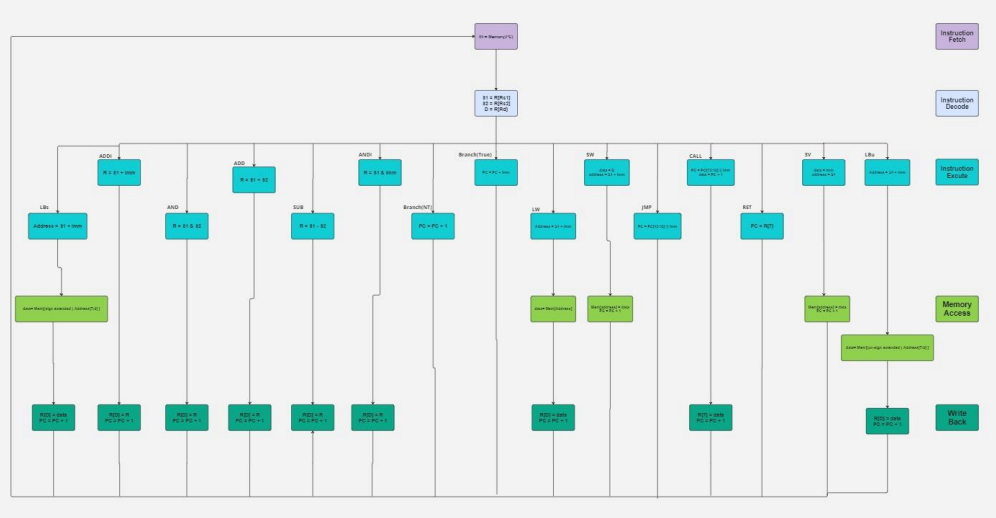

Figure 23.FSM

As we studied in the course to construct the multi-cycle, we need finite state machine because each instruction needs different cycles with different behavior. After this we implement our processor using Verilog code and then test each component and each instruction then check the results as indicated in the following section.

Simulation and Testing

To test multiple programs, we save the instructions in the instruction memory and store any necessary data in the data memory. Then, we run the code and examine the output results using both waveforms and console display information at each stage.

First, we define the registers that we used in the instructions:

Registers[0] = 16'd0; // MUST BE SET TO 0

Registers[1] = 16'd1;

Registers[2] = 16'd1;

Registers[5] = 16'd0;

Registers[6] = 16'd10;

- ## Test Program [1] – R-Type

R-Type (Register Type)

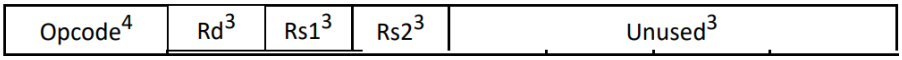

Figure 24: R-Type format

R-type and needs only 4 cycles → IF, ID, Ex, WB (no need to memory is this instruction). After this the change is only on R3.

These instructions perform the following operations:

1. **AND R-Type:** `R[3] = R[1] & R[2]`

   Memory[0] = 16'b0000011001010000; // AND R-Type (R[3] = R[1] & R[2])

```
Memory[0] = 16'b0000011001010000; // AND R-Type (R[3] = R[1] & R[2])

**********MultiCycle Procceser**********

----------Fetch Cycle----------

PC=    0
Instruction = 0000011001010000

R0 =      0, R1 =      1, R2 =      1

R3 =      0, R4 =      0, R7 =      0

M1 =      x, M4 =     13


----------Decode Cycle----------

Signals:
 RegW = 1,  ALUsrc = 0, ExtOp = x,   MemR = 0, MemW = 0, DataWB = 0, memOut = x, ExtLoad = x, ExtImm = x

Rs1 = 1, Rs2 = 2, BusA =      1,   BusB =      1


----------Execute Cycle----------

op1=  1, op2= 1

ALU result=  1, z=0, n=0, v=0


----------Write Back Cycle----------

The register R3 updated new value =  1
```

Figure 25: AND R-Type Result

2. **ADD R-Type:** R[3] = R[1] + R[2]
   Memory[1] = 16'b0001011001010000; // ADD R-Type (R[3] = R[1] + R[2])

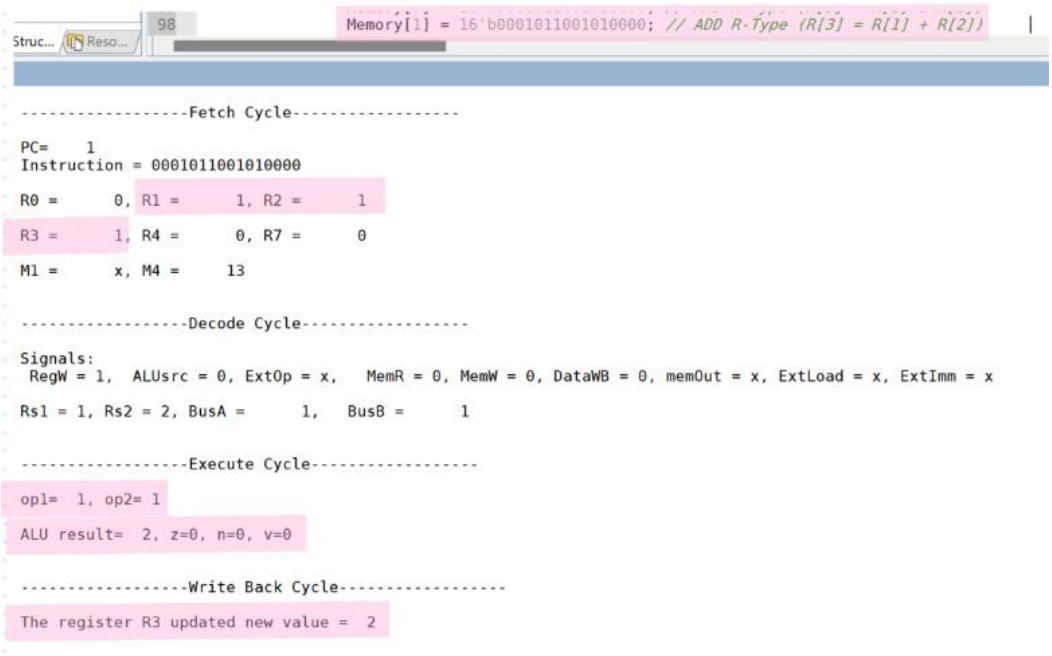

Figure 26: ADD R-Type result

3. **SUB R-Type:** `R[3] = R[1] - R[2]`
   Memory[2] = 16'b0010011001010000; // SUB R-Type (R[3] = R[1] - R[2])

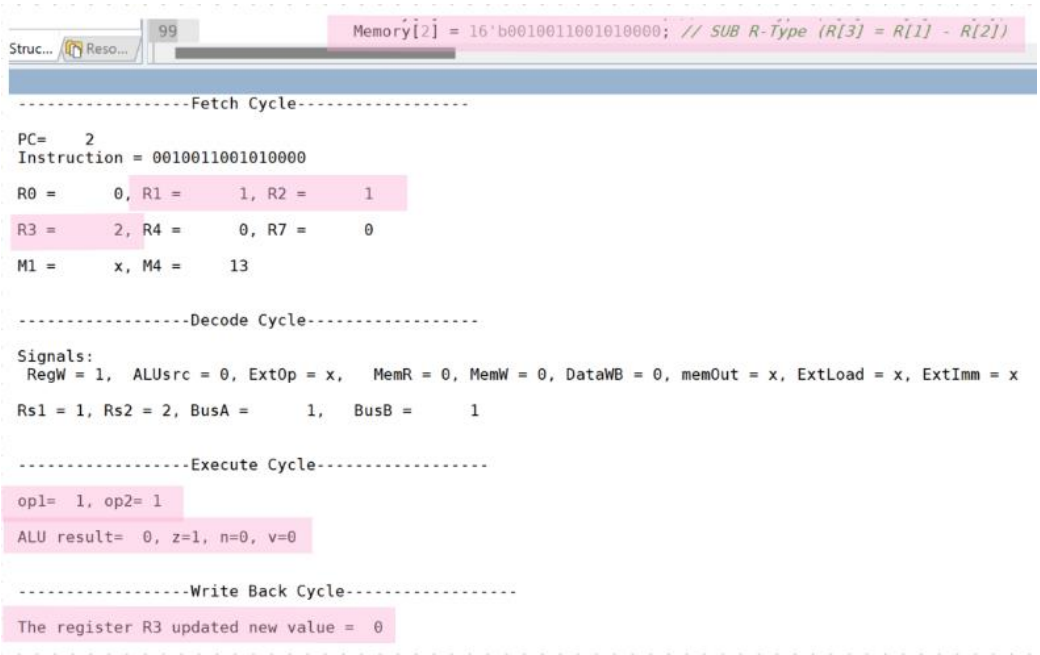

Figure 27: SUB R-Type result

- **Test Program [2] – I-Type**

I-Type (Immediate Type)

Opcode[4]	m[1]	Rd[3]	Rs1[3]	Immediate[5]

Figure 28: I-Type format

These instructions perform the following operations:

1. **ANDI I-Type:** (R[3] = R[1] & Imm)
   Memory[3] = 16'b0100001100100001; // ANDI I-Type (R[3] = R[1] & 1)

```
------------------Fetch Cycle------------------

PC=    3
Instruction = 0100001100100001

R0 =      0, R1 =      1, R2 =      1

R3 =      0, R4 =      0, R7 =      0

M1 =      x, M4 =     13


------------------Decode Cycle------------------

Signals:
 RegW = 1,  ALUsrc = 1, ExtOp = 0,   MemR = 0, MemW = 0, DataWB = 0, memOut = x, ExtLoad = x, ExtImm = 0

Rs1 = 3, Rs2 = 1, BusA =      1,   BusB =      1


------------------Execute Cycle------------------

op1=  1, op2= 1

ALU result=  1, z=0, n=0, v=0


------------------Write Back Cycle------------------

The register R3 updated new value =  1
```

Figure 29: ANDI I-Type result

2. **ADDI I-Type:** (R[3] = R[1] + Imm)
   Memory[4] = 16'b0011001100100001; // ADDI I-Type (R[3] = R[1] + 1)

```
PC=    4
Instruction = 0011001100100001

R0 =      0, R1 =      1, R2 =      1

R3 =      1, R4 =      0, R7 =      0

M1 =      x, M4 =     13


------------------Decode Cycle------------------

Signals:
 RegW = 1,  ALUsrc = 1, ExtOp = 1,   MemR = 0, MemW = 0, DataWB = 0, memOut = x, ExtLoad = x, ExtImm = 0

Rs1 = 3, Rs2 = 1, BusA =      1,   BusB =      1


------------------Execute Cycle------------------

op1=  1, op2= 1

ALU result=  2, z=0, n=0, v=0


------------------Write Back Cycle------------------

The register R3 updated new value =  2
```

Figure 30: ADDI I-Type result

3. **LW I-Type** : (R[4] = Mem(R[1] + Imm)
   Memory[5] = 16'b0101010000100001; // LW I-Type (R[4] = Mem(R[1] + 1)
   In this instruction, the value of Reg[4] will be the memory[2] which is =9.
   The load instruction need 5 cycles → IF, ID, Ex, Mem,WB

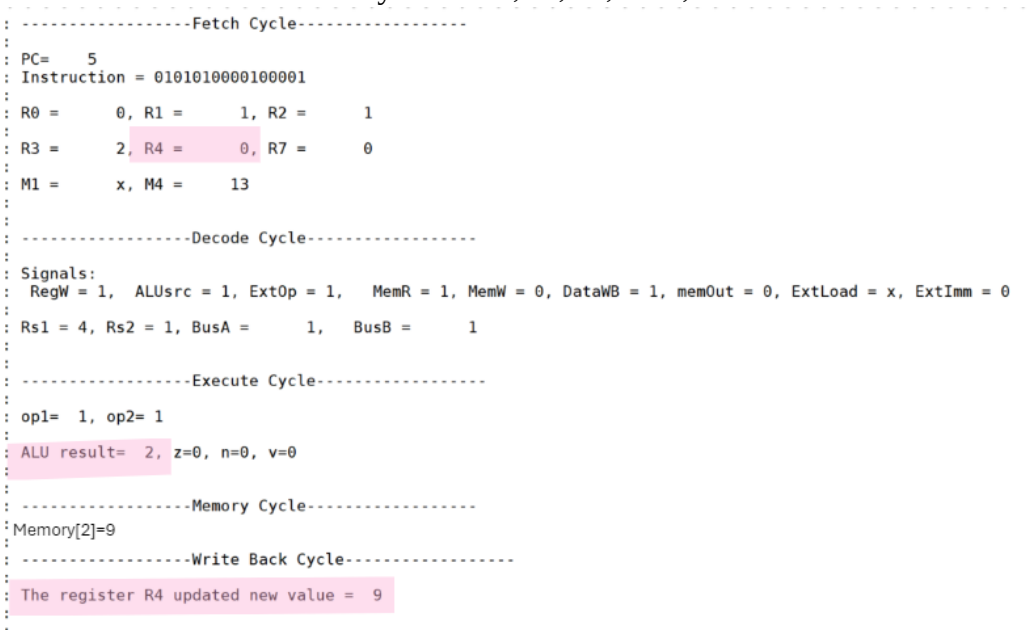

Figure 31: LW I-Type result

4. **LBu I-Type**: Reg(Rd) = Mem(Reg(Rs1) + Imm)
   Memory [6] = 16'b0110010000100010; // LBu I-Type (R [4] = Mem(R [1] + 2)
   In this instruction, the value of Reg[4] will be a zero extended byte of memory[3] since the mode bit is zero.
   Data_Memory[3] = 16'b0000000110000001;//// 385 in decimal
   Which is 129.
   The load instruction need 5 cycles → IF, ID, Ex, Mem,WB

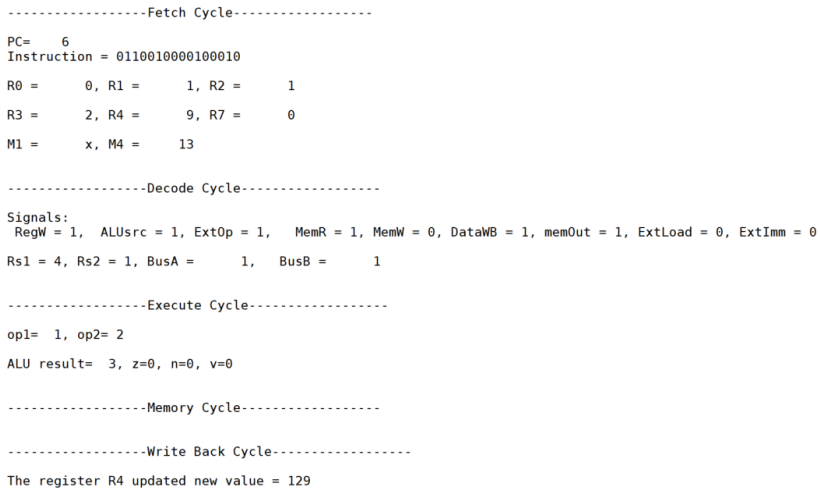

Figure 32- LBu I-Type result

5. **LBs I-Type** : Reg(Rd) = Mem(Reg(Rs1) + Imm)
   Memory [7] = 16'b0110110000100010; // LBs I-Type (R [4] = Mem(R[1] + 2)
   In this instruction, the value of Reg [4] will be a sign extended byte of memory [3] since the mode bit is one.
   Data_Memory [3] = 16'b0000000110000001; //// 385 in decimal
   Which is -127.
   The load instruction need 5 cycles → IF, ID, Ex, Mem, WB

```
------------------Fetch Cycle------------------

PC=    7
Instruction = 0110110000100010

R0 =      0, R1 =      1, R2 =      1

R3 =      2, R4 =    129, R7 =      0

M1 =      x, M4 =     13


------------------Decode Cycle------------------

Signals:
 RegW = 1,  ALUsrc = 1, ExtOp = 1,   MemR = 1, MemW = 0, DataWB = 1, memOut = 1, ExtLoad = 1, ExtImm = 0

Rs1 = 4, Rs2 = 1, BusA =      1,   BusB =      1


------------------Execute Cycle------------------

op1=  1, op2= 2

ALU result=  3, z=0, n=0, v=0


------------------Memory Cycle------------------


------------------Write Back Cycle------------------

The register R4 updated new value = -127
```

Figure 33:LBs I-Type result

6. **SW I-Type** : Mem(R[1] + 3) = R[3]

Memory[8] = 16'b0111001100100011

```
------------------Fetch Cycle------------------

PC=    8
Instruction = 0111001100100011

R0 =      0, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =      0

M1 =      5, M4 =     13


------------------Decode Cycle------------------

Signals:
 RegW = 0,  ALUsrc = 1, ExtOp = 1,   MemR = 0, MemW = 1, DataWB = 1, memOut = 1, ExtLoad = 1, ExtImm = 0

Rs1 = 3, Rs2 = 1, BusA =      2,   BusB =      1


------------------Execute Cycle------------------

op1=  1, op2= 3

ALU result=  4, z=0, n=0, v=0


------------------Memory Cycle------------------

ALUOut=      4, Memaddress=    4, dataIn =      2
```

Figure 34. SW I-Type result

The value inside R [3] will be stored in M4, the change can be noticed in pc 9.

The SW instruction need 4 cycles → IF, ID, Ex, Mem.

```
: ------------------Fetch Cycle------------------
:
: PC=     9
: Instruction = 1000001100100010
:
: R0 =       0, R1 =       1, R2 =       1
:
: R3 =       2, R4 =    -127, R7 =       0
:
: M1 =       5, M4 =       2
:
:
```

Figure 35. PC 9

7. **BGT I-Type** : R[3] > R[1]
   Memory [9] = 16'b1000001100100010;
   If Yes -> Pc = Pc + Imm else Pc = PC + 1.

```
------------------Fetch Cycle------------------

PC=    9
Instruction = 1000001100100010

R0 =      0, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =      0

M1 =      5, M4 =      2

------------------Decode Cycle------------------

Signals:
 RegW = 0,  ALUsrc = 0, ExtOp = 1,   MemR = 0, MemW = 0, DataWB = 1, memOut = 1, ExtLoad = 1, ExtImm = 0

Rs1 = 3, Rs2 = 1, BusA =      2,   BusB =      1

------------------Execute Cycle------------------

op1=  2, op2= 1

ALU result=  1, z=0, n=0, v=0

------------------Fetch Cycle------------------

PC=   11
Instruction = 1000101100100010

R0 =      0, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =      0
```

Figure 36. BGT I-Type result

At PC = 9, the `BGT` instruction checks if R3 > R1. With R3 = 2 and R1 = 1, the condition is true. Thus, the PC jumps to PC + Imm (11).

The BGT instruction need 3 cycles → IF, ID, Ex.

All the other branches will work the same for different conditions.

- **Test Program [3] – S-Type**

1. **Sv S-Type** : M[rs] = imm
   Memory[27] = 16'b1111001000000100

```
------------------Fetch Cycle------------------

PC=   27
Instruction = 1111001000000100

R0 =      2, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =     27

M1 =      5, M4 =      2


------------------Decode Cycle------------------

Signals:
 RegW = 0,  ALUsrc = 1, ExtOp = 1,   MemR = 0, MemW = 1, DataWB = 0, memOut = 1, ExtLoad = 1, ExtImm = 1

immS, extended S Imm = 0000000000000010
Rs1 = 0, Rs2 = 0, BusA =      2,   BusB =      2


------------------Execute Cycle------------------

op1=  2, op2= 2

ALU result=  2, z=0, n=0, v=0


------------------Memory Cycle------------------

ALUOut=      2, Memaddress=    2, dataIn =      2


------------------Fetch Cycle------------------

PC=   28
Instruction = 0000011001010000

R0 =      2, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =     27

M1 =      2, M4 =      2
```

Figure 37. SV S-Type

In pc =27

M[1] = 2

- **Test Program [4] – J-Type**

1. **JMP J-Type** : PC = {PC[15,12], Imm * 2 in Binary}
   Memory[22] = 16'b1100000000001101

```
------------------Fetch Cycle------------------

PC=   22
Instruction = 1100000000001101

R0 =      0, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =      0

M1 =      5, M4 =      2


------------------Decode Cycle------------------

Signals:
 RegW = 0,  ALUsrc = 0, ExtOp = 1,   MemR = 0, MemW = 0, DataWB = 1, memOut = 1, ExtLoad = 1, ExtImm = 0

Rs1 = 0, Rs2 = 1, BusA =      0,   BusB =      1


------------------Fetch Cycle------------------

PC=   26
```

Figure 38.JMP J-Type result

The next value of PC is 4 msb of PC with 12 imm *2, so for PC =22

It will jump to Imm*2 which is 26.

2. **CALL J-Type :** PC = {PC[15,12], Imm * 2 in Binary} , R[7] = PC + 1

```
------------------Fetch Cycle------------------

PC=   26
Instruction = 1101000000010001

R0 =      0, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =      0

M1 =      5, M4 =      2


------------------Decode Cycle------------------

Signals:
 RegW = 1,  ALUsrc = 0, ExtOp = 1,   MemR = 0, MemW = 0, DataWB = 1, memOut = 1, ExtLoad = 1, ExtImm = 0

Rs1 = 0, Rs2 = 2, BusA =      0,   BusB =      1


------------------Execute Cycle------------------

op1=  0, op2= 1

ALU result= -1, z=0, n=1, v=0


------------------Write Back Cycle------------------

The register R7 updated new value = 27


------------------Fetch Cycle------------------

PC=   34
```

Figure 39: CALL J-Type

So the value of PC=26, the next value of PC is 4 msb of PC with 12 imm *2 which is 34, and R7 value is 26+1 =27.

3. **RET J-Type** : PC= R[7]
   Memory [35] = 16'b1110000000000001
   This instruction will save the value of PC=R[7], which is 27 here.

```
------------------Fetch Cycle------------------

PC=   35
Instruction = 1110000000000001

R0 =      2, R1 =      1, R2 =      1

R3 =      2, R4 =   -127, R7 =     27

M1 =      5, M4 =      2


------------------Decode Cycle------------------

Signals:
 RegW = 0,  ALUsrc = 0, ExtOp = 1,   MemR = 0, MemW = 0, DataWB = 0, memOut = 1, ExtLoad = 1, ExtImm = 0

Rs1 = 0, Rs2 = 0, BusA =      2,   BusB =      2


------------------Execute Cycle------------------

op1=  2, op2= 2

ALU result=  4, z=0, n=0, v=0


------------------Fetch Cycle------------------

PC=   27
```

Figure 40. RET J-type result

Waveform

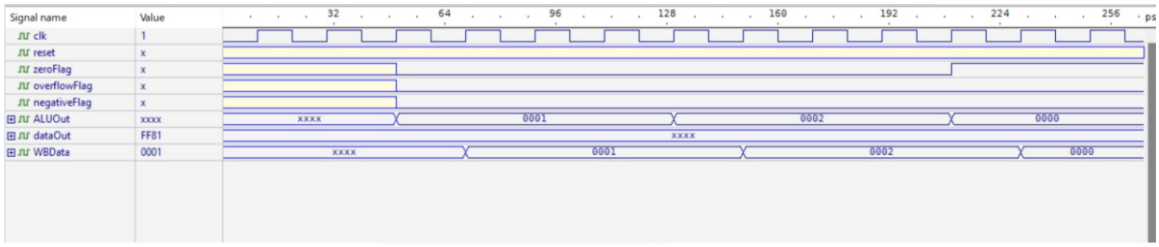

Figure 41. Waveform

The waveform is accepted and good because we see at each clock cycle positive edge new stage is operated. We put the result of ALU, Memory out and write back for our program.

Teamwork

We work together at each project stage from thinking to designing and finally testing. We change our wrong ideas. Also, we cooperate in writing the report and tracing and testing our programs code.

Conclusion

In this project we understand how to build processor and how the processor works. How to operate different instructions and see its different behavior and needs. To add, how to build processor Datapath and how to generate the controls needed for proper flow of data. Finally, we test our processor using different programs to see if there is any failure, detect it and solve it.

References

[1]: Course Slides

[2]: draw.io for drawing diagrams

[3]: miro.com for FSM

https://miro.com/welcomeonboard/a25SVndtTFEyaVNCVEZ4enJNY250YkpjMXZNd0xqeThPeVdrcWROQjFrYnFvdG11WGFhNXZENlFDcDBsSERCTXwzNDU4NzY0NTgxNTQyMTQzMDE1fDI=?share_link_id=229521274254

[4]: ArchProj - Google Drive