
Faculty of Engineering and Technology.

Department of Electrical and Computer Engineering.

ENCS4370- Computer Architecture

# Project 2- Multicycle Datapath design and Verification

---

Prepared by : Noura khdour 1212072

Rasha Daoud 1210382

Nadia Thaer 1210021

Section :2

# I-Abstract

This report covers the design and verification of a simple pipelined RISC processor implemented in Verilog. The processor, which uses a 16-bit instruction and word size, includes eight 16-bit general-purpose registers and a 16-bit program counter (PC) and 16-bit RR (Return Register) to store the return address during function calls. It supports three main instruction types: R-type, I-type and J-type and employs single data and instruction memory with Word-addressable memory. The design process involved writing RTL code for each instruction, constructing the data path, and determining the necessary control signals. Boolean equations for control signals were developed, followed by writing and testing the Verilog code. Verification was done using a detailed testbench and simulation tools, ensuring the processor's functional correctness and confirming the successful execution of all instruction types within the pipelined architecture.

# Table of content

# Table of Figures

## Table of Tables

## II-Theory

### 1-RISC and CISC

RISC stands for **Reduced Instruction Set Computer Processor**, a microprocessor architecture with a simple collection and highly customized set of instructions. It is built to minimize the instruction execution time by optimizing and limiting the number of instructions. It means each instruction cycle requires only one clock cycle, and each cycle contains three parameters: fetch, decode and execute.[1]

The CISC Stands for **Complex Instruction Set Computer**, developed by the Intel. It has a large collection of complex instructions that range from simple to very complex and specialized in the assembly language level, which takes a long time to execute the instructions. So, CISC approaches reducing the number of instruction on each program and ignoring the number of cycles per instruction.[1]

### 2- What is Multiple Cycle Datapath?

Multiple Cycle Datapath divides the entire instruction into multiple parts, each of which occurs in a serial fashion in a single clock cycle. Reusability of functional units occurs in various clock cycles, which in turn cuts the usage of the hardware and shortens the cycle of the clock. This design optimizes utilization of the hardware and can result in more efficient execution of instructions

The multi-cycle data path operates through the following stages:

### 1- Fetch Instruction

The instruction stored in memory is fetched into the control unit of the CPU. This is done by supplying the memory with the address of the instruction, enabling the CPU to retrieve the required data.

### 2- Decode (Interpret Instruction)

The control unit decodes the fetched instruction to determine the sequence of operations required for execution. This step involves interpreting the instruction's opcode and operands to prepare for the subsequent actions.

### 3- Execute

The processor performs the operations specified by the instruction. This may involve arithmetic or logical operations, control flow changes, or other computations.

**4- Memory Access**

If the instruction requires reading from or writing to memory, this step accesses the memory. For example, a load instruction would read data from memory, while a store instruction would write data to memory.

**5- Register Write**

The results of the executed instruction are written back to the appropriate register. This step ensures that the outcomes of computations or data retrievals are stored for future use.[2]

## 3- Advantages of Multi-Cycle Datapath

1-Reduced Hardware Requirements: One of the important benefits is that the hardware that is needed is minimized; for instance, functional units such as ALUs or memory modules can be used in different instructions in different clock cycles.

2-Lower Power Consumption: With reduced functional units active at a time, multiple-cycle data paths tend to save power, which is beneficial in power-conscious applications like embedded systems.

3-Flexibility in Instruction Execution: Multi-cycle data paths are able to effectively compute the execution part of expensive instructions procedures that require sequential control by carrying out their work in stages.[2]

## 4- Disadvantages of Multi-Cycle Datapath

1-Variable Clock Cycle Count: A cycle can encompass any number of instructions in a machine, and, therefore, the timing of an instruction is difficult to predetermine; and even if the cycle time is efficient, what may happen is that one instruction may take many cycles to execute, and therefore, it could take many cycles to execute an instruction.

2-Slower Throughput: Due to the fact that it is only one instruction at a time, the throughput is much lower comparatively to pipeline designs, particularly when there are simpler stem instructions that do not require many cycles.[2]

## III-Register-Transfer Level (RTL)

Register Transfer Language (RTL) is a low-level language that is used to describe the functioning of a digital circuit and, more specifically, the transfer of information between registers. It provides how data moves from one register to the other and how data is processed within the digital system. Through RTL, there is a capability of creating abstraction levels where high-level design descriptions can be created and easily linked to low-level hardware implementation in designing, simulating, as well as synthesizing digital circuits.

**1-For R-type instruction**

AND Rd, Rs, Rt Reg(Rd) = Reg(Rs) & Reg(Rt)

ADD Rd, Rs, Rt Reg(Rd) = Reg(Rs) + Reg(Rt)

SUB Rd, Rs, Rt Reg(Rd) = Reg(Rs) - Reg(Rt)

SLL Rd, Rs, Rt Reg(Rd) = Reg(Rs) << Reg(Rt)

SRL Rd, Rs, Rt Reg(Rd) = Reg(Rs) >> Reg(Rt)

**Fetch instruction**: Instruction ← MEM[PC]

**Fetch operands**: data1 ← Reg(rs), data2 ← Reg(rt)

**Execute operation**: ALU_result ← func(data1, data2)

**Write ALU result**: Reg(rd) ← ALU_result

**Next PC address**: PC ← PC + 1

---

**2-For I-type instruction**

**A-**

ANDI Rt, Rs, Imm Reg(Rt) = Reg(Rs) & Imm (u)

ADDI Rt, Rs, Imm Reg(Rt) = Reg(Rs) + Imm(s)

**Fetch instruction**: Instruction ← MEM[PC]

**Fetch operands**: data1 ← Reg(rs), data2 ← Extend(imm6)

**Execute operation**: ALU_result ← op(data1, data2)

**Write ALU result:** Reg(rt) ← ALU_result

**Next PC address**: PC ← PC + 1

---

**B-** LW Rt, Imm(Rs) Reg(Rt) = Mem(Reg(Rs) + Imm)

**Fetch instruction**: Instruction ← MEM[PC]

**Fetch base register**: base ← Reg(rs)

**Calculate address**: address ← base + sign_extend(imm6)

**Read memory**: data ← MEM[address]

**Write register Rt**: Reg(rt) ← data

**Next PC address**: PC ← PC + 1

**C-** SW Rt, Imm(Rs) Mem(Reg(Rs) + Imm) = Reg(Rt)

**Fetch instruction**: Instruction ← MEM[PC]

**Fetch registers**: base ← Reg(rs), data ← Reg(rt)

**Calculate address**: address ← base + sign_extend(imm6)

**Write memory**: MEM[address] ← data

**Next PC address**: PC ← PC + 1

---

**D**- BEQ Rs, Rt, Imm if (Reg(Rs) == Reg(Rt)) Next PC = branch target else Next PC = PC + 1

**Fetch instruction**: Instruction ← MEM[PC]

**Fetch operands**: data1 ← Reg(rs), data2 ← Reg(rt)

**Equality**: zero ← subtract(data1, data2)

**Branch:** if (zero) PC ← Current PC + sign extended immediate

else PC ← PC + 1

---

**E**- BNE Rs, Rt, Imm if (Reg(Rs) != Reg(Rt)) Next PC = branch target else Next PC = PC + 1

**Fetch instruction**: Instruction ← MEM[PC]

**Fetch operands**: data1 ← Reg(rs), data2 ← Reg(rt)

**Equality**: zero ← subtract(data1, data2)

**Branch:** if (not zero) PC ← Current PC + sign extended immediate

else PC ← PC + 1

---

**F**- FOR Rs, Rt

**Fetch instruction**: Instruction ← MEM[PC]

**Fetch operands**: data1 ← Reg(rs), data2 ← Reg(rt)

**Equality**: data2 ← subtract( data2, one)

**Next PC address**: if (not zero) PC ← data1

else PC ← PC + 1

**3-For J-type instruction**

**A-** JMP Offset Next PC = jump target

**Fetch instruction:** Instruction ← MEM[PC]

**Target PC address:** target ← PC[15:9]||9-bit offset

**Jump:** PC ← target

---

**B-** CALL Offset Next PC = jump target PC + 1 is stored on the RR

**Fetch instruction:** Instruction ← MEM[PC]

**Target PC address:** target ← PC[15:9] || 9-bit offset

**Jump:** PC ← target

---

**C-** RET  Next PC = value of the RR

**Fetch instruction:** Instruction ← MEM[PC]

**Target PC address:** target ←reg( RR)

**Jump:** PC ← target

## IV- Datapath Implementation

before to implementing the data path, it was essential to comprehend the format of each instruction. The table below shows the structure for each type of instruction

| Bit/type | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **R-type** | op | op | op | op | rd | rd | rd | rs | rs | rs | rt | rt | rt | f | f | f |
| **I-type** | op | op | op | op | rs | rs | rs | rt | rt | rt | im | im | im | im | im | Im |
| **J-type** | op | op | op | op | off | off | off | off | off | off | off | off | off | f | f | f |

Table 1:Types format

The table indicates that the opcode occupies 4 bits [15:12] for all instruction types. In R-type instructions,the destination register (rd) is specified by bits [11:9], and the source registers are given by rs[8:6] and rt[5:3], with the remaining bits  for funct. For I-type instructions,rs is indicated by bits [11:9], rt by bits [8:6], and the immediate value by bits [5:0]. In J-type instructions, the offset value spans bits [11:3] and the funct is indicated by bits [2:0]

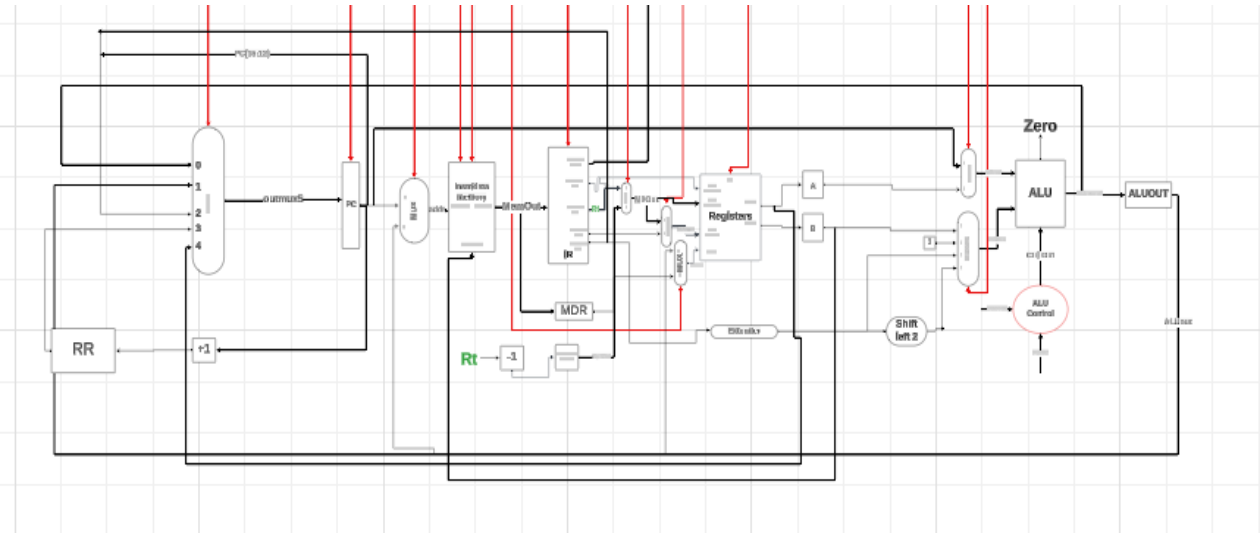
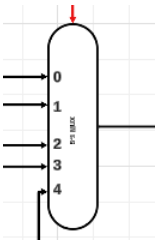



Figure 1:datapath implementation

**Detailed description of the data path**

This Datapath is divided into several stages, allowing it to function in a multi-cycle manner: Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory (MEMO), and Write Back (WB). Each stage is controlled by a specific enable signal (control signals will be discussed later).The data path comprises several components, including the PC register, instruction register, instruction memory/data, extender, memory data register , multiplexers (MUXes), ALU,Alu out, ,RR,specific register . Each of these components will be discussed in detail below.

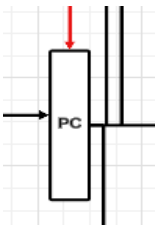
## PC MUX




This multiplexer (MUX) is designed to select the appropriate PC address based on the opcode. For instance, the default selection is PC+1, but if the opcode indicates the branch it will select 1,but if the opcode indicates the jump it will select 2, but if the opcode indicates the for loop it will select 4 and but if the opcode indicates the RET it will select 3

Figure 2:PC mux

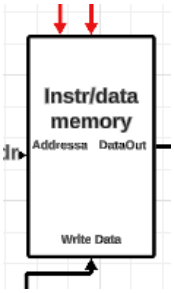
## Program Counter (PC)




The PC register holds the address of the next instruction to be fetched from instruction memory. It is updated at the beginning of each IF cycle to point to the next instruction address.

Figure 3:PC reg

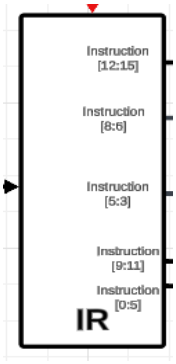
## Instruction memory/data




The purpose of instruction and data memory in a multicycle processor is to facilitate the fetching, decoding, and execution of instructions by storing both the program's instructions and the data it manipulates. The processor fetches an instruction from instruction memory using the Program Counter (PC), which specifies the address of the next instruction to execute. Also, it Stores the data needed by the program in case of load and store

Figure 4:instruction/data memory

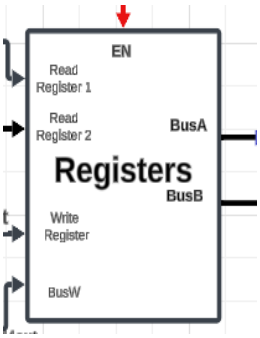
## The Instruction Register




The Instruction Register captures and holds the fetched instruction from the Instruction Memory during the Instruction Fetch (IF) stage. This register ensures that the instruction is stable and available for the next stage, Instruction Decode (ID). It acts as a buffer to prevent any changes in the fetched instruction before it is fully decoded and executed.After fetching the instruction, it will be divided based on the information

Figure 5:Instruction register

## Register File




The Register File consists of eight 16-bit general-purpose registers. It has two read buses (A and B) and one write bus, allowing simultaneous reading from two registers and writing to one register. A write enable signal, generated by the control unit, controls whether data is written to the register file during the Write Back (WB) stage.

Figure 6:register file

## Immediate Extender

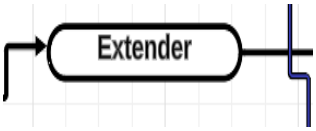



Figure 7:Extender

The Immediate Extender takes part [5:0] of the instruction and processes it based on the opcode. For I-type instructions, it operates on the least significant 6 bits. If the instruction is a logical operation (such as ANDI), it extends the immediate value with zeros from 6 bits to 16 bits. This component can be viewed as two separate extenders combined into : one for I-type unsigned, one for I-type signed.

## ALU

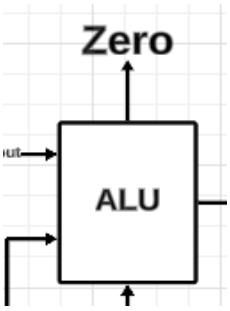



Figure 8:ALU

The Arithmetic Logic Unit (ALU) is a crucial component of a processor responsible for performing arithmetic and logic operations on the data.it include Arithmetic Operations like addition,subtrication,etc. Logical Operations like and,or,etc. Shift Operations like SLL,SRL,rtc. It has also Status Flags like zero flag which we need it for branch.it takes 3 inputs and one of these inputs is alu_ctrl that indicate the operation

## ALUout register

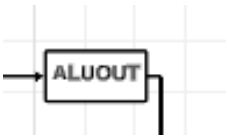



Figure 9:alu reg

The ALU Out Register is a vital component of a multicycle processor, enabling efficient execution of instructions by storing intermediate results and supporting the sequential nature of multicycle operations

## Memory data register

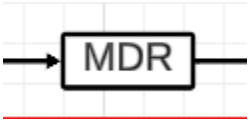



Figure 10:MDR

The Memory Data Register (MDR), also known as the Memory Buffer Register (MBR), is a crucial component in a multicycle processor. It temporarily holds data that is being transferred from the memory

## Return register

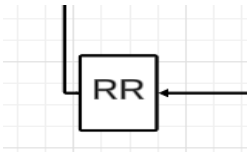
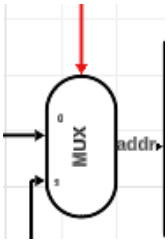



Figure 11:RR

The Retun register is 16-bit .it used to store the return address during function calls when the opcode is indicated by RET

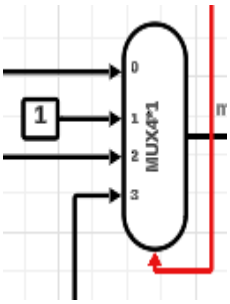
## PC-Memory MUX




This mux is set between the pc register and the memory.it choose 0 in the identical case and chose 2 for if the opcode is branch. It take signal control called IorD

Figure 12:Mux2_1

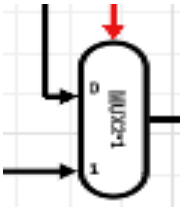
## Register file -Alu MUX4_1




This mux is set between RF and the ALU .It chose 1 for register B ,it choose 1 for identical case,it chose 2 when opcode is branch .It take a control signal called AlusrcB

Figure 13:mux4_1

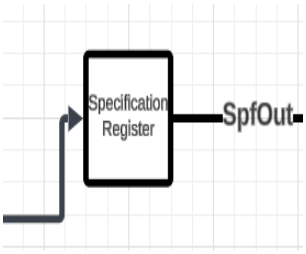
## Register file -Alu MUX2_1



This mux is set between RF and the ALU .It chose 1 for register A ,it choose 0 for current pc.It take a control signal called AlusrcA

Figure 14:Mux2_1

## Specific register




This register is used when the opcode is for loop it just used for store the value of Rt

Figure 15:Specific register

## V-Control signals design

After designing the data path and understanding the requirements for the control unit, we carefully determined the value of each control signal for every instruction. This detailed process was crucial for developing the Boolean expressions that define the control logic. By analyzing the requirements of each instruction, we specified the exact values needed for all control signals as shown in the tables below.

### A. Signal description

| Signal | Description |
|---|---|
| RegWr | Enables writing to a register (1: Write, 0: No write). |
| ALUSrcA | Selects ALU input A source (0: PC, 1: Register data). |
| ALUSrcB | Selects ALU input B source (00: Register data, 01: Constant 1, 10: Extender output, 11: Shift left 2 output). |
| MemRead | Enables reading from memory (1: Read, 0: No read). |
| MemWrite | Enables writing to memory (1: Write, 0: No write). |
| MemReg | Selects whether ALU result or memory data is written to a register (0: ALU result, 1: Memory data). |
| IRWrite | Enables writing to the instruction register (1: Write instruction, 0: No write). |
| RegDes | Chooses destination register (1: rd, 0: Output of a second mux: 0: rt, 1: rt - 1 for FOR instruction). |
| PCSource | Selects PC update source (00: ALU result, 01: Branch target, 10: Jump address, 11: PC+1, 100: BusA from the register). |
| PCWrCond | Conditional PC write for branches (1: Write if condition is met, combined with ZERO). |
| PCWr | Enables writing to the PC (1: Write PC, combined with PCWrCond using logic). |
| IorD | Selects the memory address source (0: PC for instruction fetch, 1: ALU output for data access). |
| NNR | Signal for selecting input of Read Register 2 (0: rt, 1: Memory data output). |
| Y (PC control) | Output from logic gates (AND for ZERO and PCWrCond; OR for result and PCWr). Controls PC update. |

Table 2:signal description:

**B-Control Signals values**

| Instruction | RegWr | ALUSrcA | ALUSrcB | MemRead | MemWrite | MemReg | IRWrite | RegDes | PCSource | PCWr | PCWrCond | IorD | NNR | zero |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AND | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 1 | 00 | 1 | 0 | 0 | 0 | X |
| ADD | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 1 | 00 | 1 | 0 | 0 | 0 | X |
| SUB | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 1 | 00 | 1 | 0 | 0 | 0 | X |
| SLL | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 1 | 00 | 1 | 0 | 0 | 0 | X |
| SRL | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 1 | 00 | 1 | 0 | 0 | 0 | X |
| ANDI | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 0 | 00 | 1 | 0 | 0 | 0 | X |
| ADDI | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 0 | 00 | 1 | 0 | 0 | 0 | X |
| LW | 1 | 1 | 00 | 1 | 0 | 1 | 0 | 0 | 00 | 1 | 0 | 1 | 0 | X |
| SW | 0 | 1 | 00 | 0 | 1 | x | 0 | X | 00 | 1 | 0 | 1 | 0 | X |
| BEQ | 0 | 0 | 11 | 0 | 0 | 0 | 0 | X | 01 | 1 | 1 | 0 | 0 | 1 |
| BNE | 0 | 0 | 11 | 0 | 0 | 0 | 0 | X | 01 | 1 | 1 | 0 | 0 | 0 |
| FOR | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 0 | 100 | 1 | 0 | 0 | 1 | X |
| JMP | x | X | X | 0 | 0 | 0 | 0 | x | 10 | 1 | 0 | 0 | 0 | X |
| CALL | x | X | X | 0 | 0 | 0 | 0 | x | 10 | 1 | 0 | 0 | 0 | X |
| RET | x | x | x | 0 | 0 | 0 | 0 | x | 11 | 1 | 0 | 0 | 0 | X |

Table 3:control signal values

**Boolean Equations for the main control's Signals**:

**RegWr** = ~(SW + BEQ + BNQ)

**ALUSrcB** (2 bits)

- Bit 1:

ALUSrcB[1] = BEQ + BNE

- Bit 0:

ALUSrcB[0] = 0 (constant)

**MemRead**

MemRead = LW

**MemWrite**

MemWrite = SW

**MemReg**

MemReg = LW

**IRWrite**

IRWrite = 0

**RegDes (2 bits)**

- Bit 1:

RegDes[1] = 0 (constant)

- Bit 0:

RegDes[0] = ~(BEQ + BNE + JMP + CALL + RET + SW)

**PCSource (3 bits)**

- Bit 2:

PCSource[2] = FOR

Bit 1:

PCSource[1] = JMP + CALL + RET

- Bit 0:

PCSource[0] = BEQ + BNE + RET

**PCWr**

PCWr = ~(SW + ANDI + ADDI + RET)

**IorD**

IorD = LW + SW

**NNR**

NNR = FOR

**Zero**

Zero = BEQ

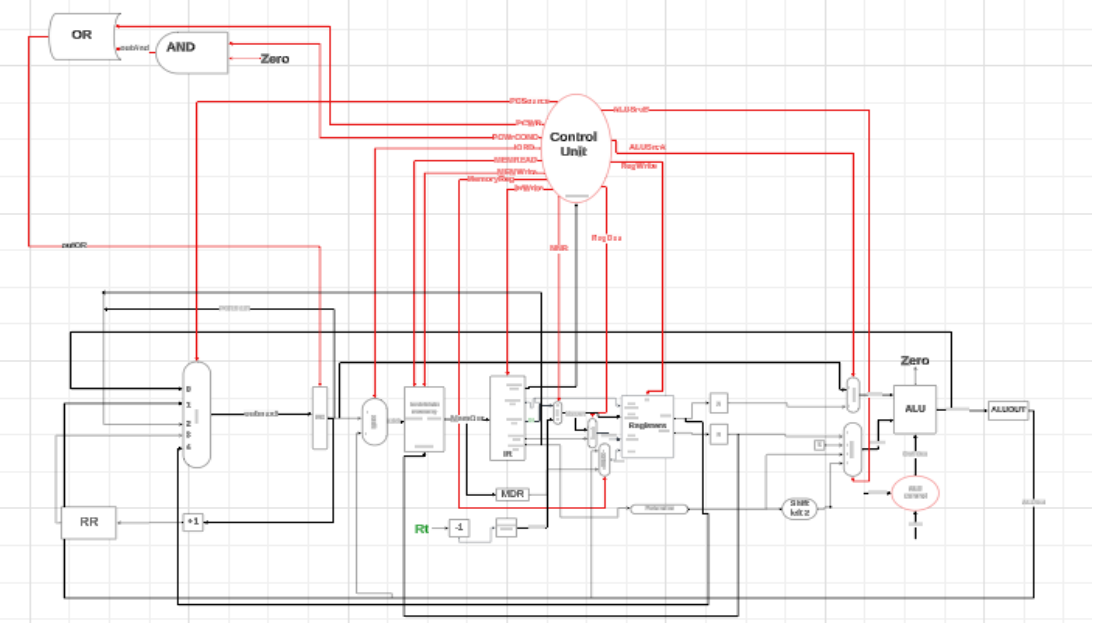



Figure 16: Datapath with control signal

## VI-State Diagram

Before we start writing the Verilog code for the processor, it's essential to draw the state diagram for each instruction. This diagram provides a clear, visual representation of the processor's control flow, making it easier to understand the sequence of operations and how different instructions traverse through various states. The state diagram serves as a crucial reference throughout the development process, aiding in debugging and verification. By comparing the expected flow of operations as depicted in the diagram with the actual behavior of the Verilog code, we can ensure accuracy and correctness. Each instruction type, such as R-type, load/store, branch, and jump, follows a specific path through these stages. For instance, R-type instructions perform an ALU operation between two registers in the EXE (execution) stage and write the result back to the register file in the WB (write-back) stage. Load instructions calculate the memory address in the EXE stage, fetch data from memory in the MEMO (memory) stage, and write it to a register in the WB stage. Similarly, store instructions compute the memory address in the EXE stage and write data to memory in the MEMO stage. Branch instructions evaluate conditions in the EXE stage and modify the program counter (PC) based on the result, while jump instructions directly update the PC.

Understanding this helps us determine the next stage for each instruction and identify which components to enable or disable at each step. This structured approach ensures that each instruction is processed correctly and efficiently, maintaining the overall integrity and performance of the processor.

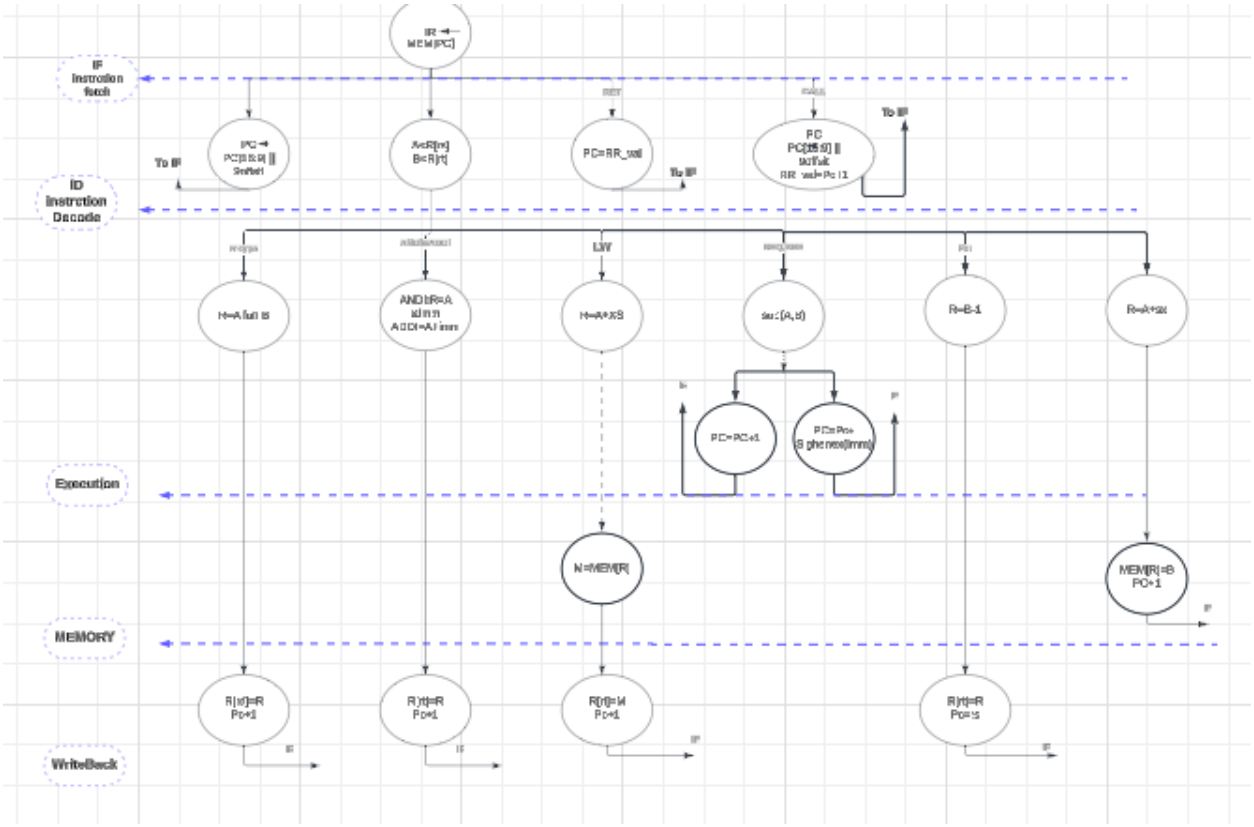



Figure 17:FSM

## VII- Simulation and Testing

First of all, we've created a separate code module each block in data path, then we connected all the modules with processor module.

### 1-Register File module

```
    module registers(
    input clk,
    input rst,            // Asynchronous reset
    input wire en,        // Enable signal for writing  (reg write)
    input wire [2:0] RA,  // Read Address A
    input wire [2:0] RB,  // Read Address B
    input wire [2:0] RW,  // Write Address
    input wire [15:0] BUSW,  // Write Data
    output reg [15:0] BUSA,  // Read Data A
    output reg [15:0] BUSB   // Read Data B
);

    // 8 registers of 16 bits
    reg [15:0] register [7:0];
    integer i;

    // Asynchronous reset (clear registers at start)
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            for (i = 0; i < 8; i = i + 1) begin
                register[i] <= 16'd0;
            end
        end else if (en && (RW != 3'b000)) begin  // Ensure no write to register 0
            register[RW] <= BUSW;  // Write to register if enabled
        end
    end

    // Read operations (synchronously with the clock)
    always @(posedge clk) begin
        BUSA <= register[RA];  // Output data for read address A
        BUSB <= register[RB];  // Output data for read address B
    end
endmodule
```

Figure 18:RF module

Register File module have clock, enable_write, RA,RB,RW, BusW as inputs and BusA, BusB as outputs. Also there is a registers array that represents the number of registers (8 registers (16 bit) as required ). The EN input controls the ID stage, if the EN is set, then the reg file module will be enabled and we can take values for BusA and BusB through registers array[RA] or [RB]. Also if the enable_write is set which is we have to write at destination register so the registers_array[RW] will take BusW value.The registers array values initiated with zero.

## 2-ALU MODULE

```
// implement the alu unit
module ALU (
input [15:0] ALu_in1,Alu_in2,
output reg [15:0] Alu_out,
output reg zero,
input [2:0]Alu_control
);

always @(*)
    case(Alu_control[2:0])
        3'b000:Alu_out<= ALu_in1 & Alu_in2;
        3'b001:Alu_out<= ALu_in1 + Alu_in2;
        3'b010:Alu_out<= ALu_in1 - Alu_in2;
        3'b011:Alu_out<= ALu_in1 << Alu_in2;
        3'b100:Alu_out<= ALu_in1 >> Alu_in2;
        3'b101: Alu_out<= ALu_in1 | Alu_in2;
        default: Alu_out <= 16'b0;              // Default case

      endcase
  assign zero = (Alu_control == 3'b010) ? (Alu_out == 16'b0) : 1'b0;


     endmodule
```

Figure 19:ALU module

The module describes an ALU capable of performing several operations (AND, OR, addition, subtraction, shifts) based on the Alu_control signal. The zero flag is used to signal when a subtraction result is zero. This kind of ALU is common in CPU designs for executing basic arithmetic and logic instructions.ALu_in1 and Alu_in2 (16-bit inputs): These are the two operands that the ALU will perform operations on. Alu_out (16-bit output): This is the result of the operation performed by the ALU. zero (1-bit output): This indicates whether the result of a subtraction operation is zero.Alu_control (3-bit input): This control signal determines which operation the ALU will perform.

## 3-ALU Control module

```
module Alu_control(
    output reg[2:0] ALU_Ctrl,
    input [2:0] funct,
    input[1:0] Alu_op);

    parameter     // from alu.v
    AND =3'b000,
    ADD =3'b001,
    SUB=3'b010 ,
    SLL=3'b011 ,
    SRL=3'b100;

       always @(*) begin

           case (Alu_op)
           2'b10: begin  // R-type instruction (opcode == 000000)
            case (funct)
              3'b000:ALU_Ctrl=AND;
              3'b001:ALU_Ctrl=ADD;
              3'b010:ALU_Ctrl=SUB;
              3'b11:ALU_Ctrl=SLL;
              3'b100:ALU_Ctrl=SRL;
             default: ALU_Ctrl = 3'bxxx; // Undefined

            endcase
           end
           // I -type instruction

           2'b11:ALU_Ctrl=AND;
           2'b00:ALU_Ctrl=ADD;
           2'b01:ALU_Ctrl=SUB;
       default: ALU_Ctrl = 3'bxxx; // Undefined
           endcase
       end
       endmodule
```

Figure 20:ALU control module

This Alu_control module acts as a decoder that takes Alu_op and funct as inputs to generate the appropriate control signals (ALU_Ctrl) for the ALU. It supports both R-type and I-type instructions by using Alu_op to differentiate instruction types and funct for specifying operations within R-type instructions.

ALU_Ctrl (3-bit output): This controls the specific operation the ALU will perform.

funct (3-bit input): This comes from the instruction and specifies the exact operation for R-type instructions.

Alu_op (2-bit input): This determines the type of instruction (R-type or I-type) and influences the operation selection.

## 4-Instruction registers module

```
module IR (
    input wire [15:0] instruction,
    output reg [3:0] opcode,
    output reg [2:0] rdt,
    output reg [2:0] rs,funct,
    output reg [5:0] imm ,
    output reg [8:0] Jump_Imm  ,

    input clk,
    input rst,
    input IRWrite
);
       always @(posedge clk or posedge rst)
       begin
       if (rst)begin
           opcode <= 4'b0;
           rdt <= 3'b0;
           rs <=3'b0;
           imm <=6'b0;
           Jump_Imm <=9'b0;
           funct<=3'b0;
       end
       |
   else if (IRWrite)  begin
    opcode <=instruction[15:12]; // Extract opcode
    rdt <= instruction[11:9];      // 1 sourse
    rs <= instruction[8:6];      //  2 sour or dest
    imm <= instruction[5:0];      // Extract immediate valu
    Jump_Imm <=instruction[11:3];
    funct<=instruction[2:0];
end
end
endmodule
```

Figure 21:IR module

The IR module is a key component in a processor's datapath. It captures and holds the current instruction, extracting various fields (like opcode, rdt, rs, imm, Jump_Imm, funct) that are used by other parts of the processor (like the ALU, control unit, and register file). This design supports both immediate and jump-type instructions, making it versatile for various instruction formats.

**Inputs:**instruction (16-bit input): The full instruction fetched from memory.clk (clock input): Synchronizes the operation.rst (reset input): Resets all outputs to zero when asserted.IRWrite (input): Enables writing to the instruction register when asserted.

**Outputs:**opcode (4-bit output): Holds the opcode extracted from the instruction.rdt (3-bit output): Typically holds a destination register or source register identifier.rs (3-bit output): Typically holds a source register identifier.funct (3-bit output): Specifies the function for R-type instructions.imm (6-bit output): Holds an immediate value for immediate instructions.Jump_Imm (9-bit output): Holds the jump immediate value for jump instructions.

## 5-PC register module

```
module pc_register (
    input clk,
    input reset,
    input PCWrite,

    input [15:0] next_pc,
    output reg [15:0] pc

);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 16'b0; // Initialize PC to 0 on reset
        end
            else if (PCWrite)
 begin
            pc <= next_pc;
        end
    end

endmodule
```

Figure 22:PC reg module

The pc_register module is responsible for maintaining and updating the program counter, which holds the address of the next instruction to be executed. It includes reset functionality to initialize the PC and a control signal (PCWrite) to manage when the PC should be updated. This module is essential for guiding the flow of program execution in a processor.

## 6-MDR module

```
module reg_memory_data(
        input clk,
        input rst,
        input [15:0] dataIn,
        output reg [15:0] dataOut
    );


    always @(posedge clk)
        begin
        if (rst)
            begin
            dataOut <= 16'hFFFF;
            end
        else
            begin
            dataOut <= dataIn;
            end
        end

endmodule
```

The reg_memory_data module is a simple register that holds a 16-bit value. It updates its output (dataOut) based on the input (dataIn) on each rising edge of the clock (clk), unless reset (rst) is asserted, in which case it sets dataOut to 16'hFFFF. This module is useful for temporarily storing data in a synchronous system, with a reset capability to initialize or clear the register.

*Figure 23:MDR module*

## 7-Signed extender module

```
module Extender(
    input[5:0] extender_in,

    output  reg [15:0] extender_out );

    always @* begin
        if ( extender_in[5])
            extender_out = {10'h3FF, extender_in};
        else
            extender_out = {10'h000,extender_in};
    end

endmodule
```

Figure 24:signed extender module

The Extender module performs sign extension on a 6-bit input to produce a 16-bit output. It checks the MSB of the input to determine the sign and extends the input accordingly:If the input is negative (MSB is 1), it fills the higher bits with 1s.If the input is non-negative (MSB is 0), it fills the higher bits with 0s. This is commonly used in processors to handle immediate values and maintain correct sign information when performing operations on values of different bit-widths.

## 8-Main controller module

```
    module main_control(

   input [3:0] opcode,   // 6-bit opcode from instruction
   input [2:0] funct,
   input clk,reset,
   output reg RegWrite,
   output reg MemRead,
   output reg MemWrite,
   output reg MemToReg, // Control signal for writing memory data to registers
   output reg ALUSrcA,
   output reg RegDst,
   output reg ALUZeroCond,
   output reg [2:0]PCSrc,
   output reg IRWrite,
   output reg [1:0] ALUSrcB,
   output reg [1:0] ALUOp,  // ALU operation control (2 bits)
   output reg IorD,        // Control signal for memory address source
   output reg PCWriteCond,
   output reg NNR, // for loop
   output reg PCWrite

   );

   // States for the multicycle control
    parameter FETCH = 3'b000,
              DECODE = 3'b001,
              EXECUTE = 3'b010,
              MEMORY = 3'b011,
              WRITEBACK = 3'b100;

   reg [2:0] state, next_state;

// Control signals logic based on the state
    always @(*) begin
        // Default values for all control signals
        PCWrite = 0;
        MemRead = 0;
```

```
// Control signals logic based on the state
    always @(*) begin
        // Default values for all control signals
        PCWrite = 0;
        MemRead = 0;
        MemWrite = 0;
        PCWriteCond=0;
        IorD = 0;
        MemToReg = 0;
        IRWrite = 0;
        RegDst = 0;
        RegWrite = 0;
        ALUSrcA = 0;
        ALUSrcB = 2'b00;
        ALUOp = 2'b00;
        PCSrc = 3'b000;
        ALUZeroCond=0;

            case (state)
        FETCH: begin
            PCWrite = 1;
            IorD=0;
            MemRead=1;
            MemWrite =0;
            IRWrite =1;
            PCSrc=3'b000;
            ALUOp=2'b00;
            ALUSrcB=2'b01;
            ALUSrcA=0;
            RegWrite=0;


        end

        DECODE: begin

            ALUOp=2'b00;
            ALUSrcB=2'b11;
            ALUSrcA=0;

        end

        EXECUTE: begin
        case (opcode)
            4'b0000: begin // R-type instruction
                ALUOp=2'b10;
                ALUSrcB=2'b00;
                ALUSrcA=1;
```

```
            end

            4'b0101: begin // SW instruction
                MemWrite =1;
                IorD=1;
            end

        endcase
        end

          WRITEBACK: begin
        case (opcode)
        4'b0000: begin // R-type instruction
                MemToReg=0;
                RegWrite =1;
                RegDst =1;


        end

        4'b0100: begin // LW instruction
            RegWrite=1;
            MemToReg=1;
            RegDst=0;
        end
        endcase
    end
    endcase
end

// State transition logic
always @(posedge clk or posedge reset) begin
    if (reset)
        state <= FETCH;
    else
        state <= next_state;
end


    // Next state logic
always @(*) begin
    case (state)
        FETCH: next_state = DECODE;
        DECODE: next_state = (opcode == 4'b0100 || opcode == 4'b0101) ? MEMORY : EXECUTE;
        EXECUTE: next_state = (opcode == 4'b0100) ? MEMORY : WRITEBACK;
        MEMORY: next_state = WRITEBACK;
        WRITEBACK: next_state = FETCH;
        default: next_state = FETCH;
    endcase
end

endmodule
```

Figure 25:control unit module

This module orchestrates the control signals necessary for the multicycle execution of instructions. It uses a state machine to step through the stages of instruction fetch, decode, execute, memory access, and writeback. Each instruction type is handled by setting appropriate control signals to guide the data flow and operations within the processor. opcode (4-bit): Specifies the operation type of the instruction. funct (3-bit): Additional function information, especially for certain R-type instructions and jumps. clk (Clock): Synchronizes the state transitions. reset: Resets the state machine to the initial state. Outputs (Control Signals)**RegWrite**: Enables writing to a register.**MemRead**: Enables reading from memory.**MemWrite**: Enables writing to memory.**MemToReg**: Selects the data source for writing to the register (memory data or ALU result).**ALUSrcA**: Selects the source for the first ALU operand.**RegDst**: Selects the destination register for writing.**ALUZeroCond**: Condition for branch instructions based on ALU zero output.**PCSrc**: Selects the source for the next PC value.**IRWrite**: Enables writing to the instruction register.**ALUSrcB**: Selects the source for the second ALU operand.**ALUOp**: Controls the ALU operation.**IorD**: Selects the source for the memory address.**PCWriteCond**: Conditional PC write for branches.**NNR**: Custom control signal for a loop.**PCWrite**: Enables writing to the PC. **State Transitions:FETCH → DECODE DECODE → MEMORY** for load/store instructions or **EXECUTE** for others. **EXECUTE → MEMORY** for load instructions or **WRITEBACK** for others.**MEMORY → WRITEBACK**. **WRITEBACK → FETCH**

## 9-ALU out module

```
module reg_alu_out(
        input clk,
        input rst,
        input [15:0] ALUIn,
        output reg [15:0] ALUOut
    );

    always @(posedge clk)
        begin
        if (rst)
            begin
            ALUOut <= 16'd0;
            end
        else
            begin
            ALUOut <= ALUIn;
            end
        end

endmodule
```

Figure 26:Aluout module

The reg_alu_out module is a register that captures the ALU output on each clock cycle, providing a stable output (ALUOut) that can be reset asynchronously. This is crucial in sequential circuits where the result of an ALU operation needs to be stored and used in subsequent cycles. **Inputs:**clk (Clock): The module is synchronized with the clock signal, meaning it updates its output on the rising edge of the clock.rst (Reset): An asynchronous reset signal that, when asserted, resets the output (ALUOut) to zero.ALUIn (16-bit): The input value from the ALU that needs to be stored in the register. **Output:**ALUOut (16-bit): The output of the register, which holds the value from the ALU input (ALUIn).

## 10-Instruction /data memory module

```
module Memory (
    input wire clk,
    input wire [15:0] addrs,      // Address for instruction or data
    input wire [15:0] write_data,    // Data to be written to memory
    input wire mem_write,            // Control signal for memory write
    input wire mem_read,             // Control signal for memory read
    output reg [15:0] read_data,     // Data read from memory
    output reg [15:0] instruction    // Instruction read from memory
);

    reg [15:0] memory [255:0];       // 256 words of 16-bit memory

    // Initialize the instruction memory with a test program
    initial begin
        memory[0] = 16'b0010000000000001; // ADDI R0, R0, 1
        memory[1] = 16'b0101000000000001; // SW R1, R0, 2
        memory[2] = 16'b0000001000000001; // ADD R1, R0, R0
        memory[3] = 16'b0100010000000100; // LW R2, R0, 4
        memory[4] = 16'b0110001010000001; // BEQ R1, R2, 1
        memory[5] = 16'b0001000000011000; // JMP to address 3
        memory[6] = 16'b0001000000100001; // CALL to address 4
        memory[7] = 16'b0001000000000010; // RET
        memory[8] = 16'b0000010000100010; //sub R2,R0,R4
        memory[9] = 16'b0111001010000001; //BNQ R1,R2,1

    end

    // Fetch instruction and read/write data
    always @(posedge clk) begin
        if (mem_read) begin
            read_data <= memory[addrs];   // Read data from memory
        end
        if (mem_write) begin
            memory[addrs] <= write_data;  // Write data to memory
        end
        instruction <= memory[addrs];     // Fetch instruction from memory
    end
endmodule
```

Figure 27:memory module

This module simulates a memory unit in a processor that can store data and instructions, allowing both read and write operations. The module is useful for storing a sequence of instructions and performing memory operations like loading, storing, and branching in a processor simulation. **Inputs:clk**: The clock signal. Memory operations are synchronized with the rising edge of this clock.**addrs**: The 16-bit address input, which specifies the memory location for reading or writing.**write_data**: The 16-bit data to be written to memory, used when mem_write is asserted.**mem_write**: A control signal that indicates whether data should be written to memory. If high (1), the memory at the given address will be written with write_data.**mem_read**: A control signal that indicates whether data should be read from memory. If high (1), data will be read from the address specified by addrs.**Outputs:read_data**: The 16-bit data read from memory when mem_read is high. This data is stored in this output.**instruction**: A 16-bit output that provides the instruction read from memory at the specified address. This is always updated with the memory content at addrs.

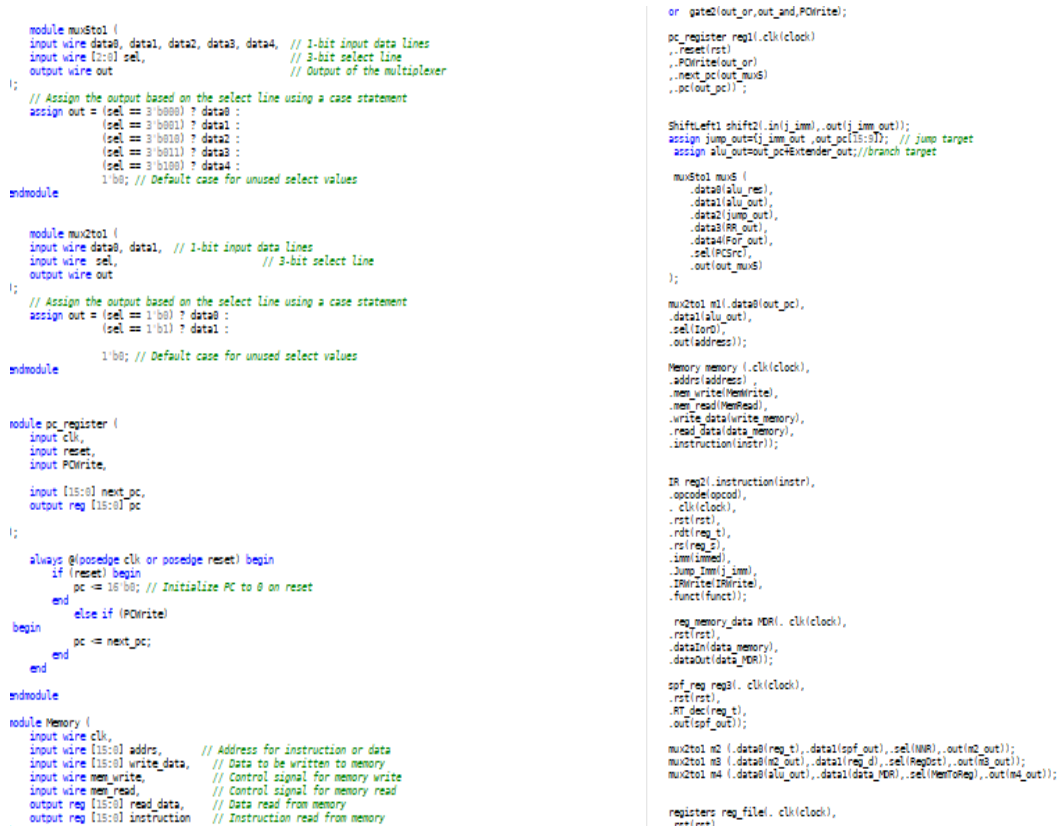
## 11-Processor module

```
module data_path(

        input clock,
        input rst,
        output [2:0] State
    );

    // control wires
    wire IRWrite;
    wire PCWrite;
    wire ALUZeroCond;
    wire PCWriteCond;
    wire IorD;
    wire MemWrite;
    wire MemToReg;
    wire ReadACond;
    wire RegWrite;
    wire ALUSrcA;
    wire NNR;
    wire  MemRead;
    wire RegDst;
    wire out_and;
    wire out_or;
    wire [1:0] ReadBCond;
    wire [2:0] PCSrc;
    wire [1:0] ALUOp;
    wire [1:0] ALUSrcB;
    wire [15:0] alu_res;
    wire [15:0] alu_out;
    wire [15:0] jump_out;
    wire [15:0] RR_out;
    wire [15:0] For_out;
    wire [15:0] out_mux5;
    wire[15:0] out_pc;
    wire [15:0] address;
    wire[15:0] write_memory;
    wire[15:0] data_memory;
    wire[15:0] instr;
    wire [3:0] opcod;
    wire [2:0]reg s,reg t,reg d,funct;
```

```
    module reg_alu_out(
        input clk,
        input rst,
        input [15:0] ALUIn,
        output reg [15:0] ALUOut
    );

    always @(posedge clk)
        begin
        if (rst)
            begin
            ALUOut <= 16'd0;
            end
        else
            begin
            ALUOut <= ALUIn;
            end
        end

endmodule


module mux5to1 (
    input wire data0, data1, data2, data3, data4,  // 1-bit input data lines
    input wire [2:0] sel,                          // 3-bit select line
    output wire out                                // Output of the multiplexer
);
    // Assign the output based on the select line using a case statement
    assign out = (sel == 3'b000) ? data0 :
                 (sel == 3'b001) ? data1 :
                 (sel == 3'b010) ? data2 :
                 (sel == 3'b011) ? data3 :
                 (sel == 3'b100) ? data4 :
                 1'b0; // Default case for unused select values
endmodule


    module mux2to1 (
    input wire data0, data1,  // 1-bit input data lines
    input wire  sel,                          // 3-bit select line
    output wire out
);
    // Assign the output based on the select line using a case statement
    assign out = (sel == 1'b0) ? data0 :
                 (sel == 1'b1) ? data1 :

                 1'b0; // Default case for unused select values
endmodule


module pc_register (
    input clk,
    input reset,
    input PCWrite,

    input [15:0] next_pc,
    output reg [15:0] pc

);
```

```
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 16'b0; // Initialize PC to 0 on reset
        end
            else if (PCWrite)
    begin
            pc <= next_pc;
        end
    end

endmodule

module Memory (
    input wire clk,
    input wire [15:0] addrs,      // Address for instruction or data
    input wire [15:0] write_data,    // Data to be written to memory
    input wire mem_write,            // Control signal for memory write
    input wire mem_read,             // Control signal for memory read
    output reg [15:0] read_data,     // Data read from memory
    output reg [15:0] instruction    // Instruction read from memory
);

    reg [15:0] memory [255:0];       // 256 words of 16-bit memory

    // Initialize the instruction memory with a test program
    initial begin
        memory[0] = 16'b0010000000000001; // ADDI R0, R0, 1
        memory[1] = 16'b0101000000000001; // SW R1, R0, 2
        memory[2] = 16'b0000001000000001; // ADD R1, R0, R0
        memory[3] = 16'b0100010000000100; // LW R2, R0, 4
        memory[4] = 16'b0111000101000001; // BEQ R1, R2, 1
        memory[5] = 16'b0001000000011000; // JMP to address 3
        memory[6] = 16'b0001000000100001; // CALL to address 4
        memory[7] = 16'b0000100000000010; // RET
        memory[8] = 16'b0000010000100010; //sub R2,R0,R4
        memory[9] = 16'b0111001010000001; //BNQ R1,R2,1

    end

    // Fetch instruction and read/write data
    always @(posedge clk) begin
        if (mem_read) begin
            read_data <= memory[addrs];   // Read data from memory
        end
        if (mem_write) begin
            memory[addrs] <= write_data;  // Write data to memory
        end
        instruction <= memory[addrs];     // Fetch instruction from memory
    end
endmodule
// Instruction Register (IR) Module
module IR (
    input wire [15:0] instruction,
    output reg [3:0] opcode,
    output reg [2:0] rdt,
    output reg [2:0] rs,funct,
    output reg [5:0] imm ,
    output reg [8:0] Jump_Imm  ,

    input clk,
```

```
    module mux5to1 (
    input wire data0, data1, data2, data3, data4,  // 1-bit input data lines
    input wire [2:0] sel,                          // 3-bit select line
    output wire out                                // Output of the multiplexer
);
    // Assign the output based on the select line using a case statement
    assign out = (sel == 3'b000) ? data0 :
                 (sel == 3'b001) ? data1 :
                 (sel == 3'b010) ? data2 :
                 (sel == 3'b011) ? data3 :
                 (sel == 3'b100) ? data4 :
                 1'b0; // Default case for unused select values
endmodule


    module mux2to1 (
    input wire data0, data1,  // 1-bit input data lines
    input wire  sel,                          // 3-bit select line
    output wire out
);
    // Assign the output based on the select line using a case statement
    assign out = (sel == 1'b0) ? data0 :
                 (sel == 1'b1) ? data1 :

                 1'b0; // Default case for unused select values
endmodule


module pc_register (
    input clk,
    input reset,
    input PCWrite,

    input [15:0] next_pc,
    output reg [15:0] pc

);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 16'b0; // Initialize PC to 0 on reset
        end
            else if (PCWrite)
    begin
            pc <= next_pc;
        end
    end

endmodule

module Memory (
```



```
module mux5to1 (
    input wire data0, data1, data2, data3, data4,  // 1-bit input data lines
    input wire [2:0] sel,                          // 3-bit select line
    output wire out                                // Output of the multiplexer
);
    // Assign the output based on the select line using a case statement
    assign out = (sel == 3'b000) ? data0 :
                 (sel == 3'b001) ? data1 :
                 (sel == 3'b010) ? data2 :
                 (sel == 3'b011) ? data3 :
                 (sel == 3'b100) ? data4 :
                 1'b0; // Default case for unused select values
endmodule


    module mux2to1 (
    input wire data0, data1,  // 1-bit input data lines
    input wire  sel,                       // 3-bit select line
    output wire out
);
    // Assign the output based on the select line using a case statement
    assign out = (sel == 1'b0) ? data0 :
                 (sel == 1'b1) ? data1 :

                 1'b0; // Default case for unused select values
endmodule



module pc_register (
    input clk,
    input reset,
    input PCWrite,

    input [15:0] next_pc,
    output reg [15:0] pc

);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            pc <= 16'b0; // Initialize PC to 0 on reset
        end
            else if (PCWrite)
begin
            pc <= next_pc;
        end
    end

endmodule

module Memory (
    input wire clk,
    input wire [15:0] addrs,      // Address for instruction or data
    input wire [15:0] write_data,   // Data to be written to memory
    input wire mem_write,           // Control signal for memory write
    input wire mem_read,            // Control signal for memory read
    output reg [15:0] read_data,    // Data read from memory
    output reg [15:0] instruction   // Instruction read from memory
```

```
or  gate2(out_or,out_and,PCWrite);

pc_register reg1(.clk(clock)
,.reset(rst)
,.PCWrite(out_or)
,.next_pc(out_mux5)
,.pc(out_pc)) ;


ShiftLeft1 shift2(.in(j_imm),.out(j_imm_out));
assign jump_out={j_imm_out ,out_pc[15:13]};  // jump target
 assign alu_out=out_pcHExtender_out;//branch target

 mux5to1 mux5 (
    .data0(alu_res),
    .data1(alu_out),
    .data2(jump_out),
    .data3(RR_out),
    .data4(For_out),
    .sel(PCSrc),
    .out(out_mux5)
);

mux2to1 m1(.data0(out_pc),
.data1(alu_out),
.sel(Ior0),
.out(address));

Memory memory (.clk(clock),
.addrs(address) ,
.mem_write(MemWrite),
.mem_read(MemRead),
.write_data(write_memory),
.read_data(data_memory),
.instruction(instr));


IR reg2(.instruction(instr),
.opcode(opcod),
. clk(clock),
.rst(rst),
.rdt(reg_t),
.rs(reg_s),
.imm(immed),
.Jump_Imm(j_imm),
.IRWrite(IRWrite),
.funct(funct));

 reg_memory_data MDR(. clk(clock),
.rst(rst),
.dataIn(data_memory),
.dataOut(data_MDR));

spf_reg reg3(. clk(clock),
.rst(rst),
.RT_dec(reg_t),
.out(spf_out));

mux2to1 m2 (.data0(reg_t),.data1(spf_out),.sel(NWR),.out(m2_out));
mux2to1 m3 (.data0(m2_out),.data1(reg_d),.sel(RegDst),.out(m3_out));
mux2to1 m4 (.data0(alu_out),.data1(data_MDR),.sel(MemToReg),.out(m4_out));


registers reg_file(. clk(clock),
```

Figure 28:processor module

The processor uses a **finite state machine (FSM)** to control different stages of instruction processing. The FSM transitions through the following stages: **FETCH**: The processor fetches the instruction from memory. The Program Counter (PC) is updated, and the instruction is written to the instruction register. **DECODE**: The instruction is decoded, preparing the operands for the ALU operation. **EXECUTE**: The ALU performs the required operation using the operands from the registers or immediate values. **MEMORY**: Memory read/write operations occur if needed, and data is moved between memory and registers. **WRITEBACK**: The result from the ALU or memory is written back to the register file.

The control signals manage the operations for each stage, like reading/writing from/to memory, selecting ALU inputs, and determining where to store results. The FSM keeps track of the current state and moves to the next state based on the execution of the previous stage.

**Key Control Signals: RegWrite**: Controls whether data is written to a register. **MemRead** and **MemWrite**: Control memory read and write operations. **ALUSrcA** and **ALUSrcB**: Control the operands used by the ALU. **ALUOp**: Determines the ALU operation.**PCWrite**: Controls whether the PC is updated.

The FSM cycles through states based on the opcode and function code, enabling the corresponding actions for each instruction type in a multicycle processor.

## 12-Processor module testbench

```
module tb_datapath;
    // Inputs
    reg clk;
    reg rst;

    // Outputs
    wire [2:0] State;

    // Instantiate the Unit Under Test (UUT)
    data_path uut (
        .clock(clk),
        .rst(rst),
        .State(State)
    );

    // Clock generation: 10ns clock period
    always #5 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 1;
        // Reset sequence
        #10 rst = 0;
        // Test Case 1: Reset and fetch initial instruction
        #50;
        rst = 1;  // Assert reset
        #20;
        rst = 0;  // Release reset

        // Test Case 2: Verify PC register behavior
        // Simulate an incremented program counter and memory operations
        #100;

        // Test Case 3: Apply ALU and control signal stimulus
        // Further tests can be defined here to validate datapath behavior

        // Finish simulation
        #500 $finish;
    end

    // Monitor outputs for debug purposes
    initial begin
        $monitor("Time = %0t | State = %b", $time, State);
    end
endmodule
```

Figure 29:Test bench

Inputs: clk (Clock): This is the clock signal used to synchronize the operations of the datapath. rst (Reset): This is the reset signal that initializes the datapath to a known state.

Outputs: State: This 3-bit signal represents the current state of the datapath. It is monitored during simulation to check the processor's state transitions.

Components: data_path uut: This is the unit under test (UUT) where the actual datapath module is instantiated. The clock and rst signals are connected to the corresponding inputs of the datapath, and the State output is connected to the datapath's state.

This testbench is a basic starting point to test the datapath functionality of our processor. It provides:Clock generation,Reset and stimulus application and  Monitoring the state of the datapath during the simulation.

## ❖ Conclusion

This project required the design, implementation, and verification of a Verilog multicycle RISC processor. We succeeded in achieving the following goals with this project:

1. Design and Implementation: We created a five-stage multicycle processor that included stages for write-back, memory access, fetch, decode, and ALU. Carefully designed datapath and control path to support the provided ISA instructions. Important parts including memory, the ALU, and registers were combined to create a coherent CPU design. We carefully considered our design options to guarantee the accuracy and performance of the processor.

2. Control Signals: We created the control signals required to run the CPU. In order to guarantee correct instruction execution, truth tables and Boolean equations were developed.

3-Teamwork: The project was a cooperative involving all team members who contributed to the design, implementation, and report writing. Every phase of the project was fully covered by all the group team.

4-Documentation and Reporting: Althorough report that included control signals, microarchitecture schematics, FSM,and design process documentation was put together. This report offers a thorough description of the project's progress and results.

# References

**[1]-** https://www.javatpoint.com/risc-vs-cisc

**[2]-** https://www.geeksforgeeks.org/differences-between-single-cycle-and-multiple-cycle-datapath/

**[3]-** https://www.eecg.utoronto.ca/~moshovos/ECE352-2022/l20-multicycle.html

THE END