RV32I

A RISC-V BASED 32-BIT CPU

Team Members

1. Bibhav Jha(Pulchowk)
2. Biplob Giri (Pulchowk)
3. Shishir Dahal (Pulchowk)
4. Aashutosh Sah (Pulchowk)

AGENDA

01 **Introduction**

About the RISC-V

02 **Methodology**

Black Box Designs
Sub modules
Code Snippets

03 **Results**

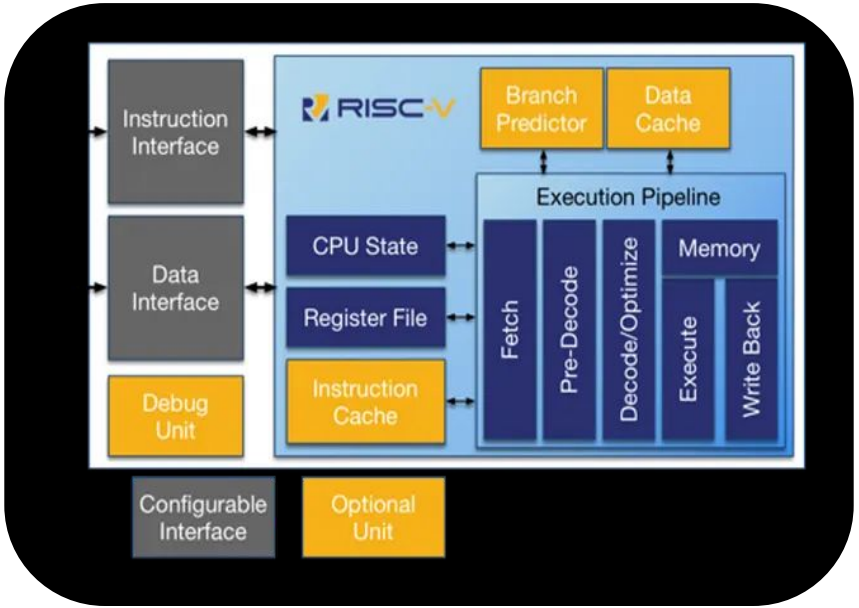
Simulation
Waveforms
Goals

01

Introduction

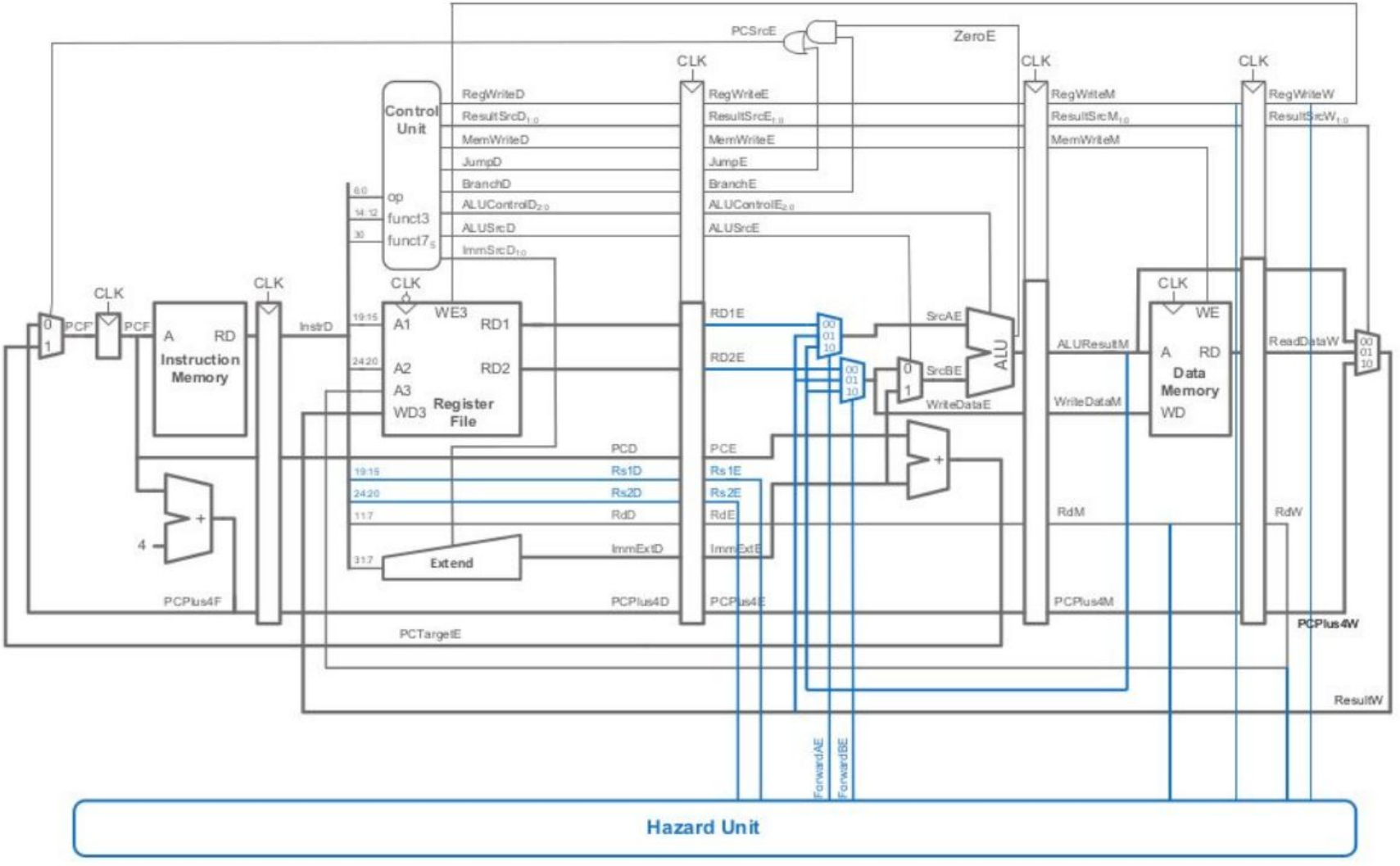
About RISC-V

RISC-V is an open-source and extensible instruction set architecture (ISA) based on the Reduced Instruction Set Computer (RISC) principles. It is modular, scalable, and supports multiple extensions, making it suitable for applications ranging from embedded systems to high-performance computing. Developed at UC Berkeley, RISC-V is free from licensing fees, encouraging innovation and adoption in academia and industry.



02

Methodology


PCSrcE
ZeroE
CLK
Control Unit
RegWriteD
ResultSrcD1:0
MemWriteD
JumpD
BranchD
ALUControlD2:0
ALUSrcD
ImmSrcD1:0
op
funct3
funct7
RegWriteE
ResultSrcE1:0
MemWriteE
JumpE
BranchE
ALUControlE2:0
ALUSrcE
RegWriteM
ResultSrcM1:0
MemWriteM
RegWriteW
ResultSrcW1:0
PCF'
PCF
Instruction Memory
InstrD
WE3
A1
A2
A3
WD3
RD1
RD2
Register File
RD1E
RD2E
SrcAE
SrcBE
WriteDataE
ALU
ALUResultM
WriteDataM
WE
Data Memory
WD
ReadDataW
PCD
PCE
Rs1D
Rs1E
Rs2D
Rs2E
RdD
RdE
RdM
RdW
ImmExtD
ImmExtE
Extend
PCPlus4F
PCPlus4D
PCPlus4E
PCPlus4M
PCPlus4W
PCTargetE
ResultW
ForwardAE
ForwardBE
Hazard Unit

Block Designs

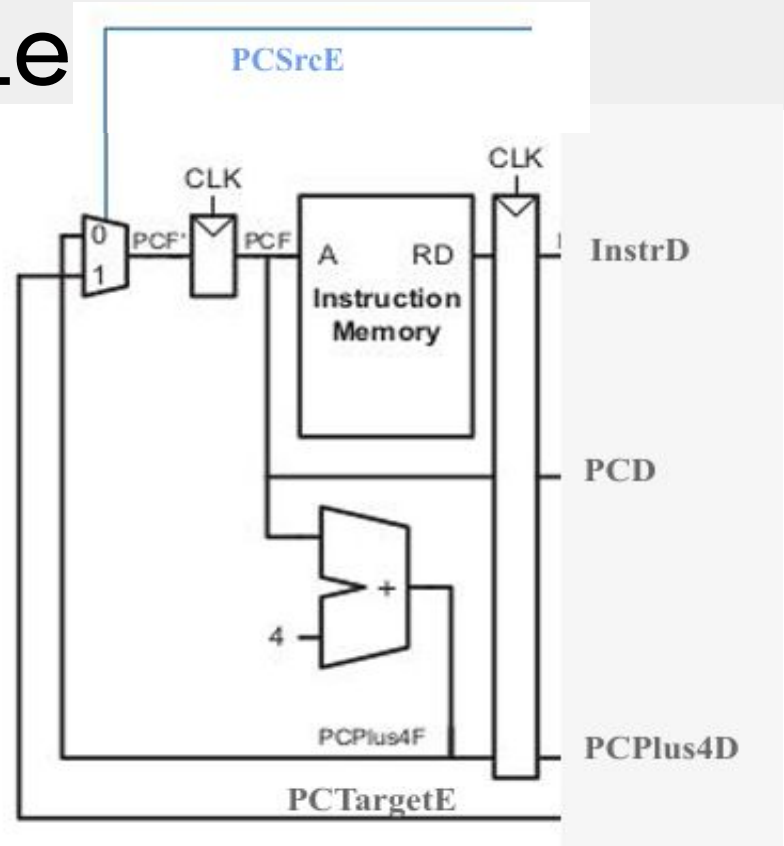
1. Fetch Cycle
2. Decode Cycle
3. Execute Cycle
4. Memory Cycle
5. Writeback Cycle
6. Hazard Unit

Block Designs
1. Fetch Cycle

Fetch Cycle of this particular designed system is used to read an instruction from instruction memory.

Modules to be integrated:

a. PC MUX
b. Program Counter
c. Adder
d. Instruction Memory



1. a. PC Mux

The mux used here is a general mux that uses selector bit to choose between two inputs. In our case PCSrcE provides the selector pin data and the inputs are PCTargetE and PCPlus4F

```
module mux (a,b,s,c);

    input [31:0]a,b;
    input s;
    output [31:0]c;

    assign c = (~s) ? a : b ;

endmodule

// ============================
// Multiplexer for 3 Inputs
// ============================
module mux_3_by_1 (
    input [31:0] a, b, c,      // Inputs: three 32-bit data lines
    input [1:0] s,             // Select signal (2 bits)
    output [31:0] d            // Output: Selected data line
);

    assign d = (s == 2'b00) ? a :
               (s == 2'b01) ? b :
               (s == 2'b10) ? c : 32'h00000000; // Default case

endmodule
```

1. b. Program Counter

Program Counter(PC) module simply sets the current PC to next.

```
// ============================
// PC Module
// ============================
module PC (
    input clk, rst,             // Clock and reset signals
    input [31:0] PC_next,       // Next program counter value
    output reg [31:0] PC        // Current program counter
);
    always @(posedge clk) begin
        if (rst)
            PC <= PC_next;      // Update PC on clock edge
        else
            PC <= 32'h00000000; // Reset PC to zero
    end
endmodule
```

1. c. Adder

PC Next or the another value of PC is updated normally by adding 4 to the previous PC. This particular module is responsible for that action.

adder.v

```
// ============================
// PC Adder Module
// ============================
module pc_adder (
    input [31:0] a, b,          // Inputs to the adder
    output [31:0] c             // Sum output
);
    assign c = a + b;           // Perform addition
endmodule
```

1. d. Instruction Memory

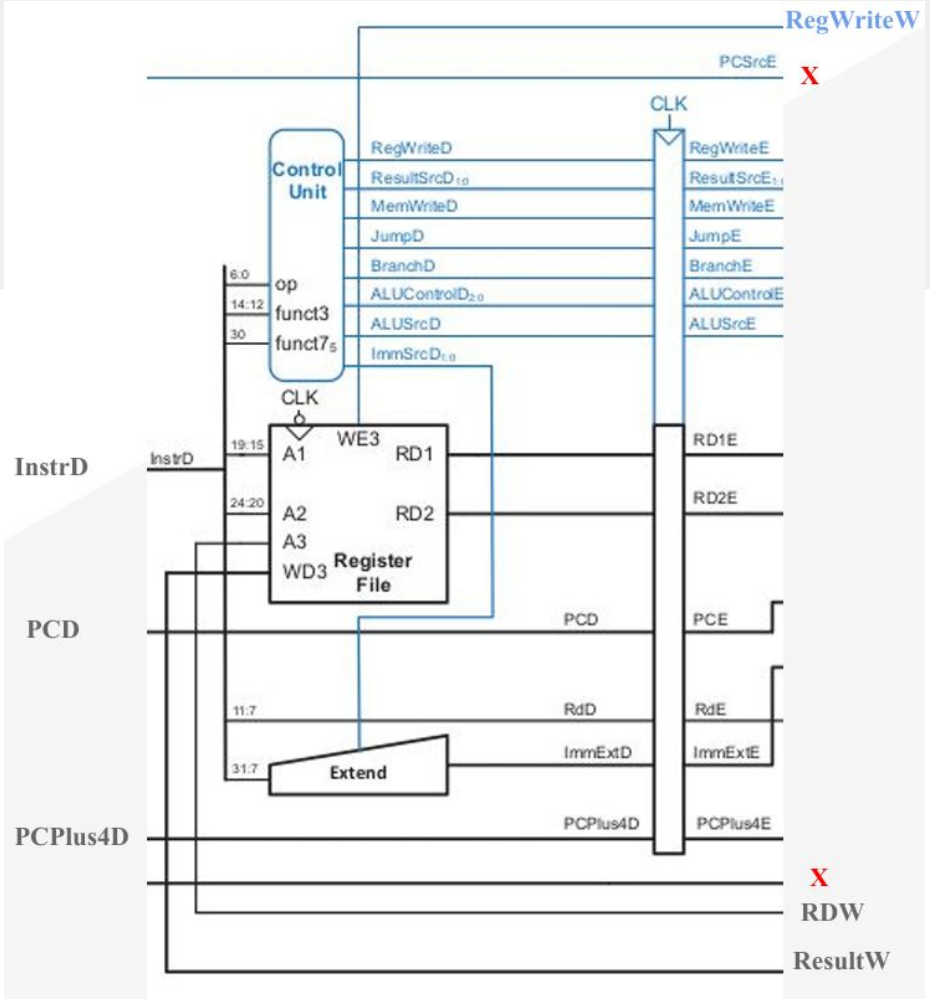
Instruction Memory reads the instruction from a .hex file and sends the output file from that read instruction as RD.

```
// ============================
// Instruction Memory Module
// ============================
module instruction_memory (
    input rst,                  // Reset signal
    input [31:0] A,             // Address to fetch instruction
    output [31:0] RD            // Instruction read
);
    reg [31:0] memory [1023:0]; // Instruction memory array

    assign RD = (rst == 1'b0) ? 32'h00000000 : memory[A[31:2]];

    initial begin
        $readmemh("memfile.hex", memory); // Load instructions from
    end
endmodule
```

2. Decode Cycle

Decode cycle gets the instruction from fetch cycle and optimally decodes them to further be used in next stages.
Modules to be integrated:

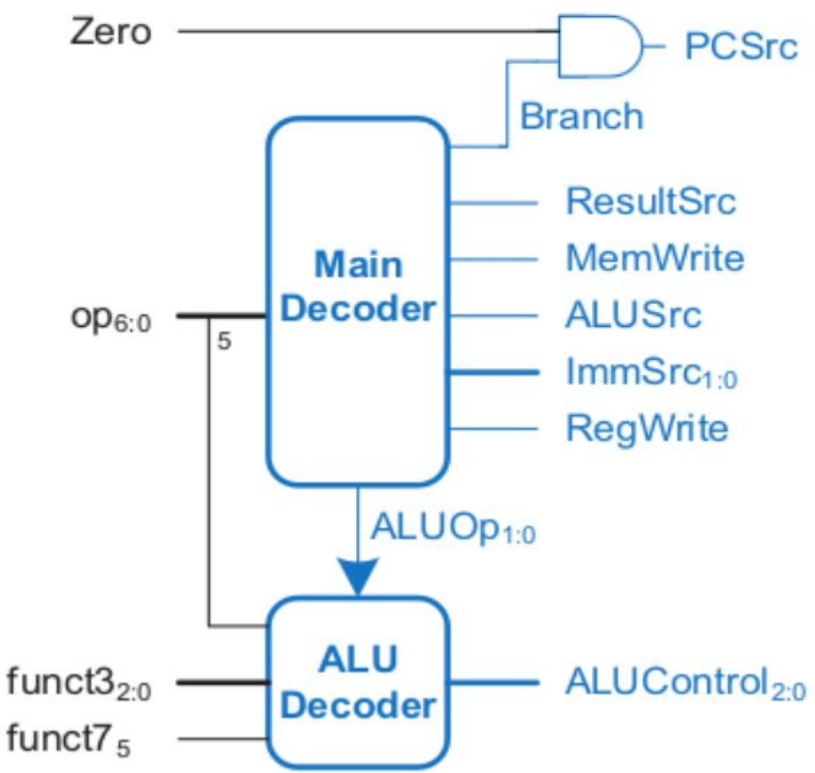
a. Control Unit
b. Extender
c. Register File



2. a. Control Unit

Control Unit is responsible for providing appropriate control signals to the respective areas where the operation required them to.
Modules to be integrated:

1. Main Decoder
2. ALU Decoder



2. a. i. Main Decoder

Main Decoder provides appropriate control signals for the system based on the opcodes read from fetch cycle. However control signals related to ALU are not governed by main decoder

```
// ============================
// Main Decoder Module
// ============================
module main_decoder (
    input [6:0] opcode,          // Opcode from instruction
    output RegWrite, MemWrite, ALUSrc, Branch, Jump, // Control signals
    output [1:0] ImmSrc, ResultSrc, ALUOp // Immediate, result source, and ALU operation
);
    parameter lw = 7'b0000011;     // Load word
    parameter sw = 7'b0100011;     // Store word
    parameter R_type = 7'b0110011; // R-type instructions
    parameter beq = 7'b1100011;    // Branch equal
    parameter jal = 7'b1101111;    // Jump and link
    parameter jalr = 7'b1100111;   // Jump and link register

    // Control signal assignments based on opcode
    assign RegWrite = ((opcode == lw) |
                       (opcode == R_type) |
                       (opcode == jal)  |
                       (opcode == jalr)) ? 1'b1 : 1'b0;

    assign MemWrite = (opcode == sw) ? 1'b1 : 1'b0;
    assign ALUSrc = ((opcode == lw) | (opcode == sw) | (opcode == jalr)) ? 1'b1 : 1'b0;
    assign Branch = (opcode == beq) ? 1'b1 : 1'b0;

    assign Jump = ((opcode == jal) | (opcode == jalr)) ? 1'b1 : 1'b0;

    assign ImmSrc = (opcode == sw) ? 2'b01 :   // S-type immediate
                    (opcode == beq) ? 2'b10 :  // B-type immediate
                    (opcode == jal) ? 2'b11 :  // J-type immediate
                    2'b00;                     // Default (I-type)

    assign ALUOp = ((opcode == lw) | (opcode == sw)) ? 2'b00 : // Load/store
                   (opcode == R_type) ? 2'b10 :               // ALU operations
                   (opcode == beq) ? 2'b01 :                  // Branch
                   2'b00;                                     // Default

    assign ResultSrc = (opcode == lw) ? 2'b01 :  // Memory result
                       ((opcode == jal) | (opcode == jalr)) ? 2'b11 : // PC+4 (Jump link)
                       2'b00;                    // Default (ALU result)
endmodule
```

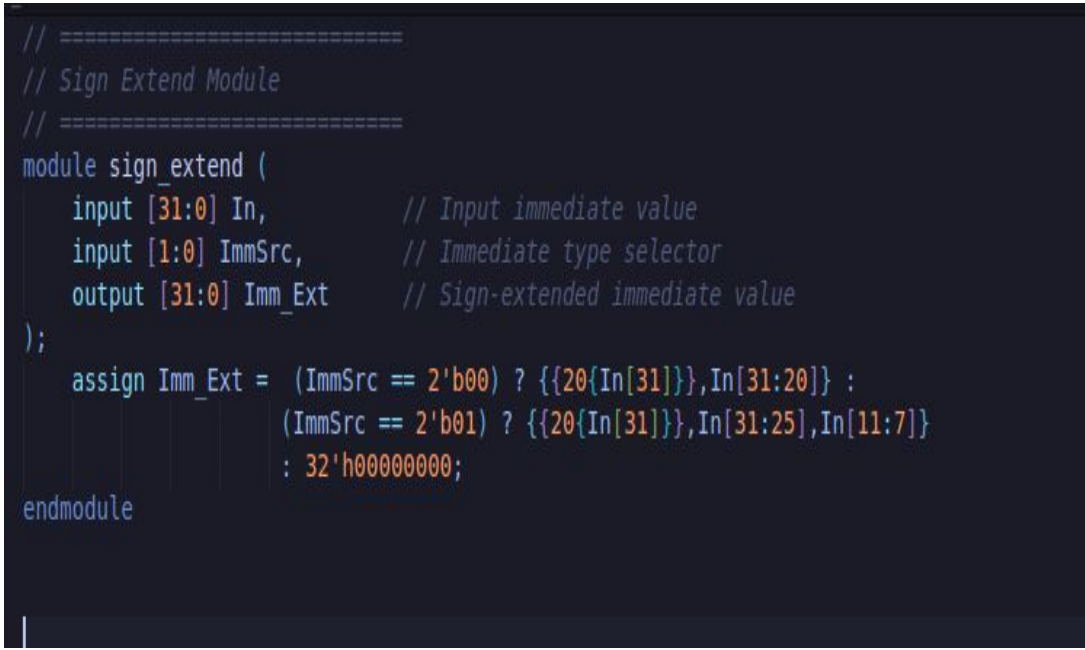
2. a. i. ALU Decoder

ALU Decoder on the other hand deals with the control signals related to ALU controls.

```
// ============================
// ALU Decoder Module
// ============================
module alu_decoder (
    input [1:0] ALUOp,          // ALU operation code from control unit
    input [6:0] funct7, opcode, // Function and opcode fields from instruction
    input [2:0] funct3,         // Function field from instruction
    output [2:0] ALUControl     // ALU control signals
);
    parameter add = 3'b000;    // Add operation
    parameter sub = 3'b001;    // Subtract operation
    parameter slt = 3'b101;    // Set less than
    parameter OR  = 3'b011;    // OR operation
    parameter AND = 3'b010;    // AND operation

    wire [1:0] concatenation;  // Concatenated signals for specific operations

    // Determine ALUControl based on ALUOp and instruction fields
    assign concatenation = {opcode[5], funct7[5]};
    assign ALUControl = (ALUOp == 2'b00) ? add : // lw, sw operations
                        (ALUOp == 2'b01) ? sub : // beq operation
                        ((ALUOp == 2'b10) & (funct3 == 3'b010)) ? slt : // slt operation
                        ((ALUOp == 2'b10) & (funct3 == 3'b110)) ? OR :  // OR operation
                        ((ALUOp == 2'b10) & (funct3 == 3'b111)) ? AND : // AND operation
                        ((ALUOp == 2'b10) & (funct3 == 3'b000) & (concatenation == 2'b11)) ? sub : // sub operation
                        ((ALUOp == 2'b10) & (funct3 == 3'b000) & (concatenation != 2'b11)) ? add : // add operation
                        3'b000; // Default
endmodule
```

2. b. Extender

Extender is responsible for extending a 12 bit immediate value to a 32 bit value for ease of operations. This is done because the alu and the other cycle demand results that are of 32 bit word length.



```
// ============================
// Sign Extend Module
// ============================
module sign_extend (
    input [31:0] In,          // Input immediate value
    input [1:0] ImmSrc,       // Immediate type selector
    output [31:0] Imm_Ext     // Sign-extended immediate value
);
    assign Imm_Ext =  (ImmSrc == 2'b00) ? {{20{In[31]}},In[31:20]} :
                      (ImmSrc == 2'b01) ? {{20{In[31]}},In[31:25],In[11:7]}
                      : 32'h00000000;
endmodule
```

2. c. Register File

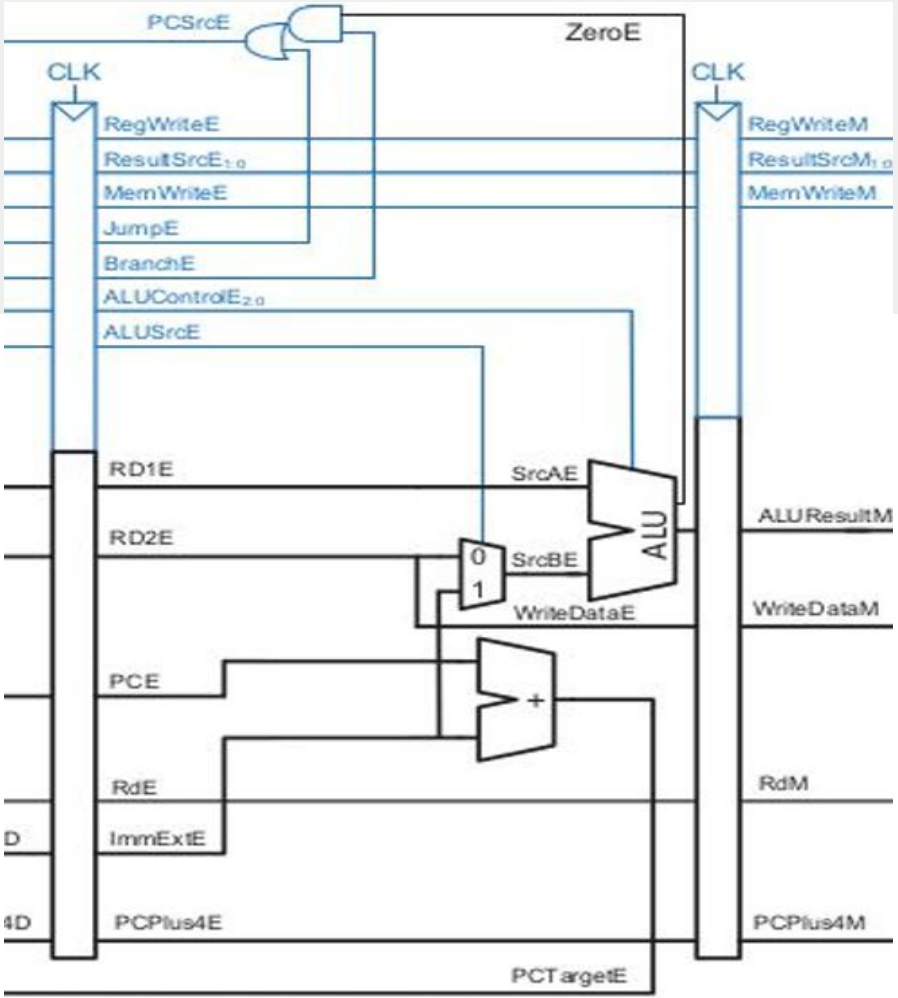
Register File reads the instruction from source registers that contain the data for processing and stores them into destination registers for future use.

```
// ============================
// Register File Module
// ============================
module registerFile (
    input clk, rst,             // Clock and reset signals
    input [4:0] rs1, rs2, rd,   // Register addresses
    input writeEnable,          // Write enable signal
    input [31:0] writeData,     // Data to write to register
    output [31:0] readData1, readData2 // Data read from registers
);
    reg [31:0] x [31:0];        // Register array
    always @(posedge clk) begin
        if ((rd != 5'h00) & writeEnable)
            x[rd] <= writeData;
    end

    assign readData1 = (rst==1'b1) ? x[rs1] : 32'h00000000;
    assign readData2 = (rst==1'b1) ? x[rs2] : 32'h00000000;

    initial begin
        x[0] = 32'h00000000;
    end
endmodule
```

3. Execution Cycle

Execution cycle is responsible for appropriate operations on the data input from the previous cycles.
Modules to be integrated:

a. Mux
b. ALU
c. Adder



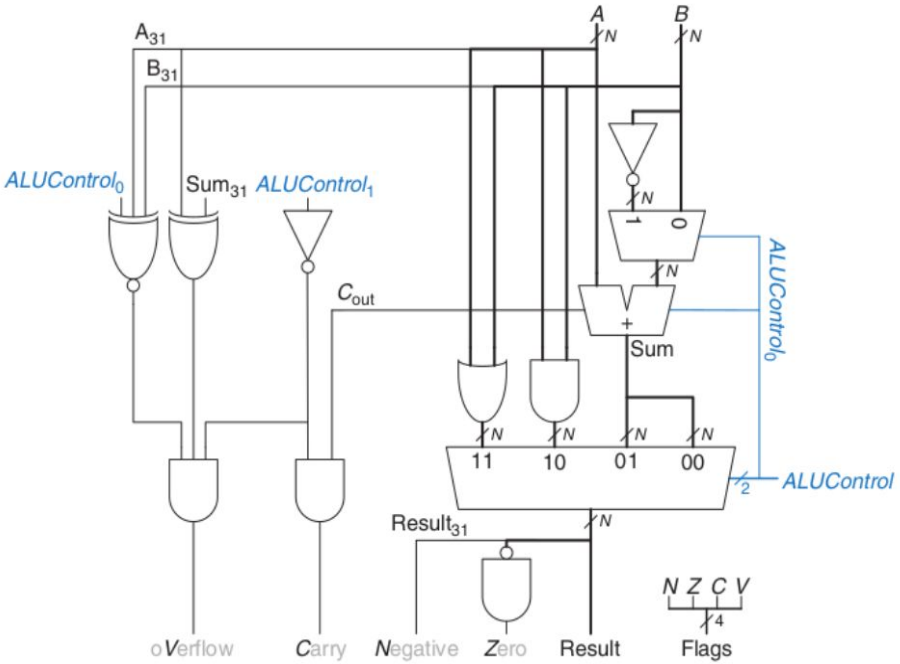
3. a. Mux

Mux in this cycle is used to differentiate between signals via immediate registers or the data registers.
Also in further hazard implementation a 3 X 1 mux is used to implement execution based on forwarding signals.

```
module mux (a,b,s,c);

    input [31:0]a,b;
    input s;
    output [31:0]c;

    assign c = (~s) ? a : b ;

endmodule

// ============================
// Multiplexer for 3 Inputs
// ============================
module mux_3_by_1 (
    input [31:0] a, b, c,      // Inputs: three 32-bit data lines
    input [1:0] s,             // Select signal (2 bits)
    output [31:0] d            // Output: Selected data line
);

    assign d = (s == 2'b00) ? a :
               (s == 2'b01) ? b :
               (s == 2'b10) ? c : 32'h00000000; // Default case

endmodule
```

3. b. ALU

ALU is used to perform general arithmetic calculation as well as provide the logical output for the provided 32 bit input signals.



3. c. Adder

Adder in this scenario is used to add the PC with the received immediate value (extended) for the next PC as PCTargetE, which is in fact selected in the fetch cycle as an option via a mux.

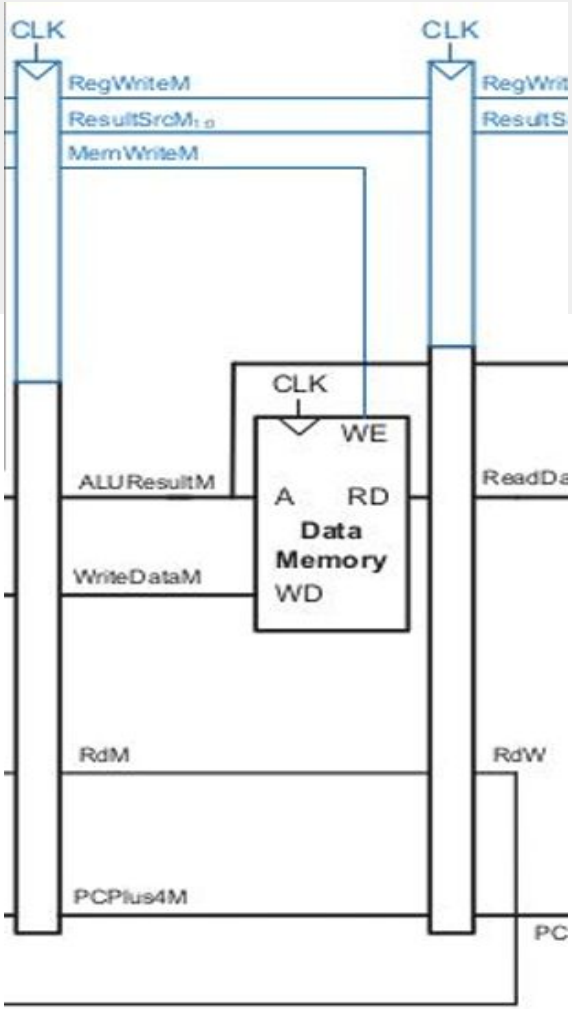
adder.v

```
// ============================
// PC Adder Module
// ============================
module pc_adder (
    input [31:0] a, b,          // Inputs to the adder
    output [31:0] c             // Sum output
);
    assign c = a + b;           // Perform addition
endmodule
```

4. Memory Cycle

Memory cycle is used to store the executed result into data memory for the next operation.
Modules to be integrated:

a. Data Memory



4. a. Data Memory

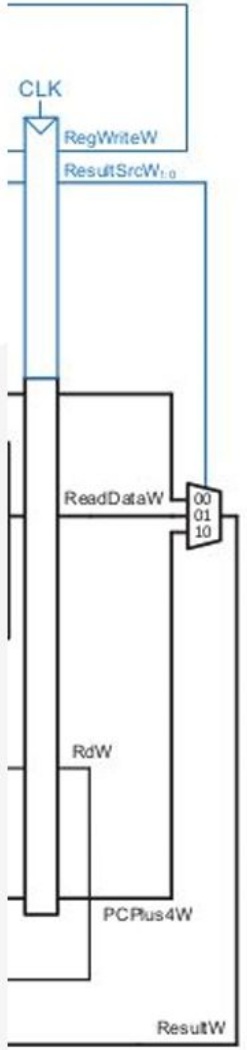
Data memory both saves an instruction into memory and outputs the instruction that is required for the operation given the input parameter provided, in this case alongside A.

```
// ============================
// Data Memory Module
// ============================
module data_memory (
    input [31:0] A, writeData,  // Address and data to write
    input clk, rst, writeEnable,// Clock, reset, and write enable signals
    output [31:0] RD            // Data read from memory
);
    reg [31:0] data_memory [1023:0]; // Memory array

    assign RD = (rst) ? data_memory[A] : 32'h00000000;

    always @(posedge clk) begin
        if (writeEnable)
            data_memory[A] <= writeData;
    end
endmodule
```

5. Writeback Cycle

Writeback Cycle provides the observed results back to the previous cycles in order for those cycles to continue the operation normally.
Modules to be integrated:

a. Mux(3X1)



5. a. Mux(3 x 1)

Mux here is used to select between 3 data in order to map them respectively into ResultW. The selector used in this case is ResultSrcW.

```
x.v
module mux (a,b,s,c);

    input [31:0]a,b;
    input s;
    output [31:0]c;

    assign c = (~s) ? a : b ;

endmodule

// ============================
// Multiplexer for 3 Inputs
// ============================
module mux_3_by_1 (
    input [31:0] a, b, c,      // Inputs: three 32-bit data lines
    input [1:0] s,             // Select signal (2 bits)
    output [31:0] d            // Output: Selected data line
);

    assign d = (s == 2'b00) ? a :
               (s == 2'b01) ? b :
               (s == 2'b10) ? c : 32'h00000000; // Default case

endmodule
```

Data Hazards

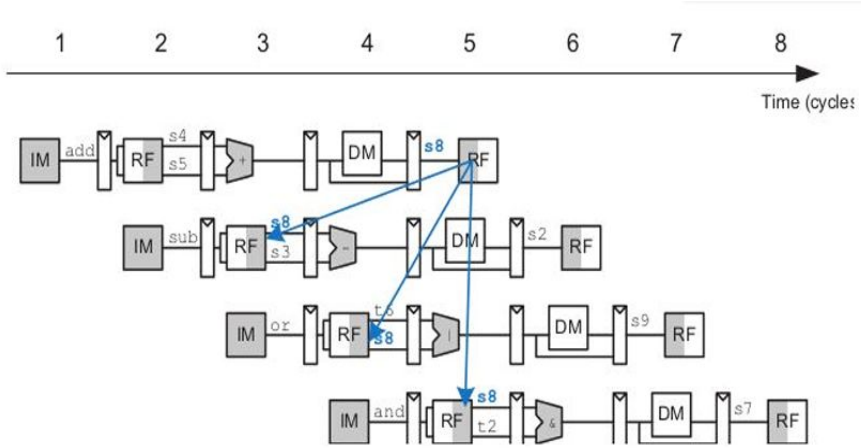
In spite of these cycles, a pipeline stage never actually performs well under heavy operations because of something called hazards. These hazards are categorized into:

1. Structural Hazards(Not dealt in this case)
2. Data Hazards

Data Hazards

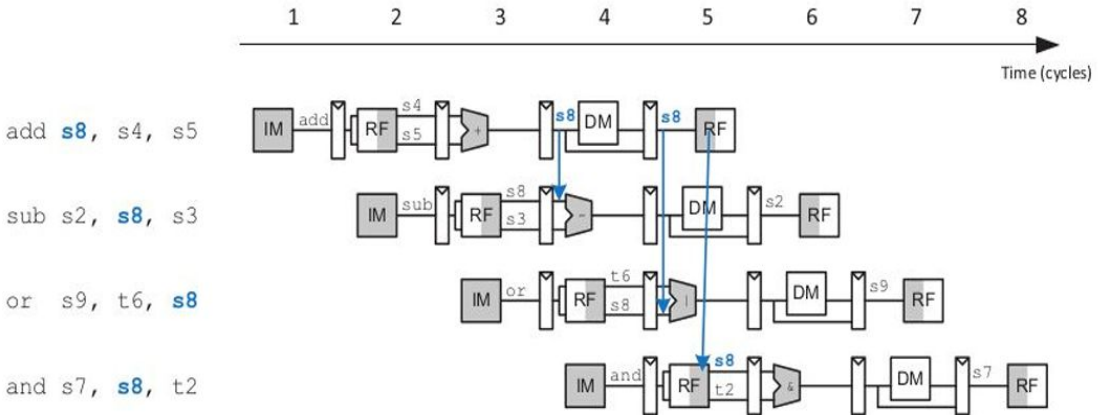
Data Hazards in pipelining occur when the data inside the register is used without operation on that register from a previous instruction
For example in the instruction alongside, s8 needs to be updated before we proceed to second instruction, however, since the second instruction is executed on 3rd clock cycle, we don't receive desired output.

```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



Solution to Data Hazard

Solution to data hazard, in this particular implementation is done via Forwarding. That is as soon as the cycle observed the value is calculated the value is passed directly on to the next instruction.



03

Results


clk
rst
WriteDataM
ResultW
ResultSrcW
ResultSrcM
ResultSrcE
RegWriteW
RegWriteM
RegWriteE
ReadDataW
RS2_E
RS1_E
RD_M
RD_E
RDW
RD2_E
RD1_E
PCTargetE
PCSrcE
PCPlus4W
PCPlus4M
PCPlus4E
PCPlus4D
PCE
PCD
MemWriteM
MemWriteE
InstrD
Imm_Ext_E
ForwardBE
ForwardAE
BranchE
ALU_ResultW
ALU_ResultM
ALUSrcE
ALUControlE

Further Works

This project although simulated is not yet upto mark, the project is yet to be implemented on a board, and further testing and benchmarks of the system can still be extended.

We are happy to receive any critics and review on regards to the implementation and overall improvement of this project.

THANK YOU

RV32I