Single Cycle RV-32I Processor

RISC-V

By Mostafa Mohamed Farhan

Introduction

In this project, you are required to implement a 32-bit single-cycle microarchitecture RISC-V processor based on Harvard Architecture. The single cycle microarchitecture executes an entire instruction in one cycle. In other words, instruction fetch, instruction decode, execute, write back, and program counter update occurs within a single clock cycle.

Objective

Referring to figure one, you are required to write the RTL Verilog files for all sub modules of the RISC-V processor (e.g. Register File, Instruction Memory, etc.). Then, implementing the top module of the RISC-V processor. Finally, you will configure this processor on Cyclone® IV FPGA device.

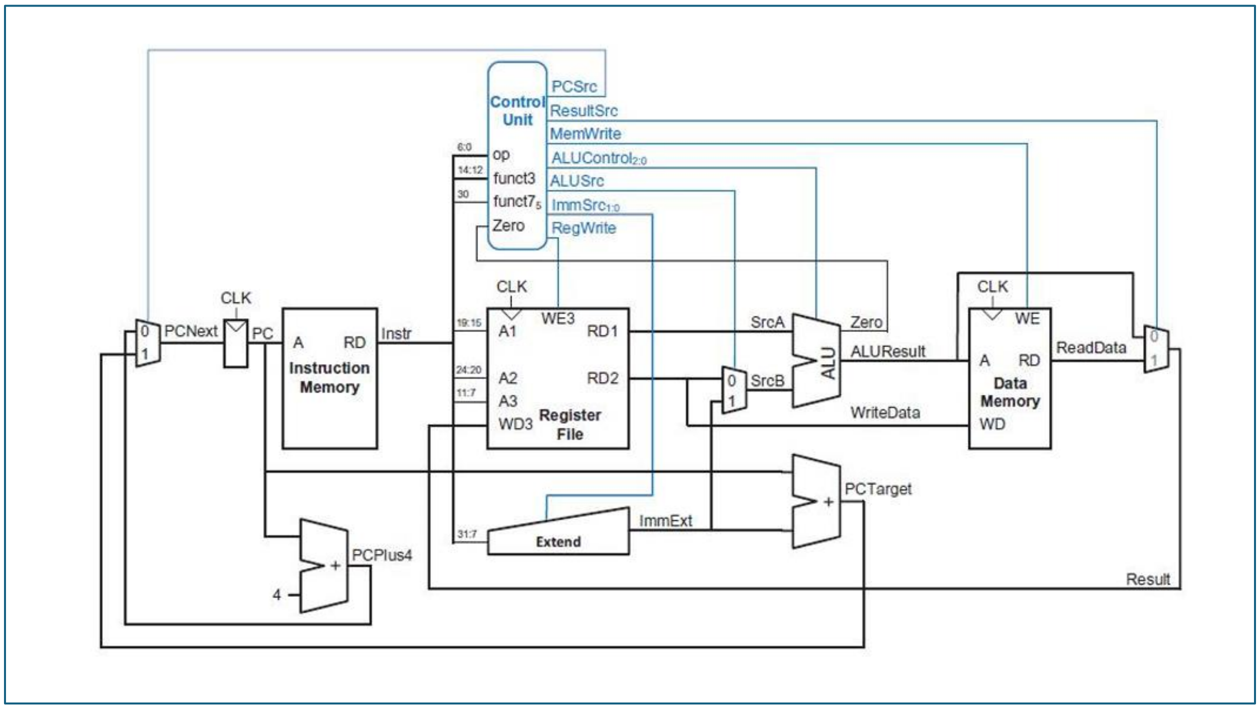
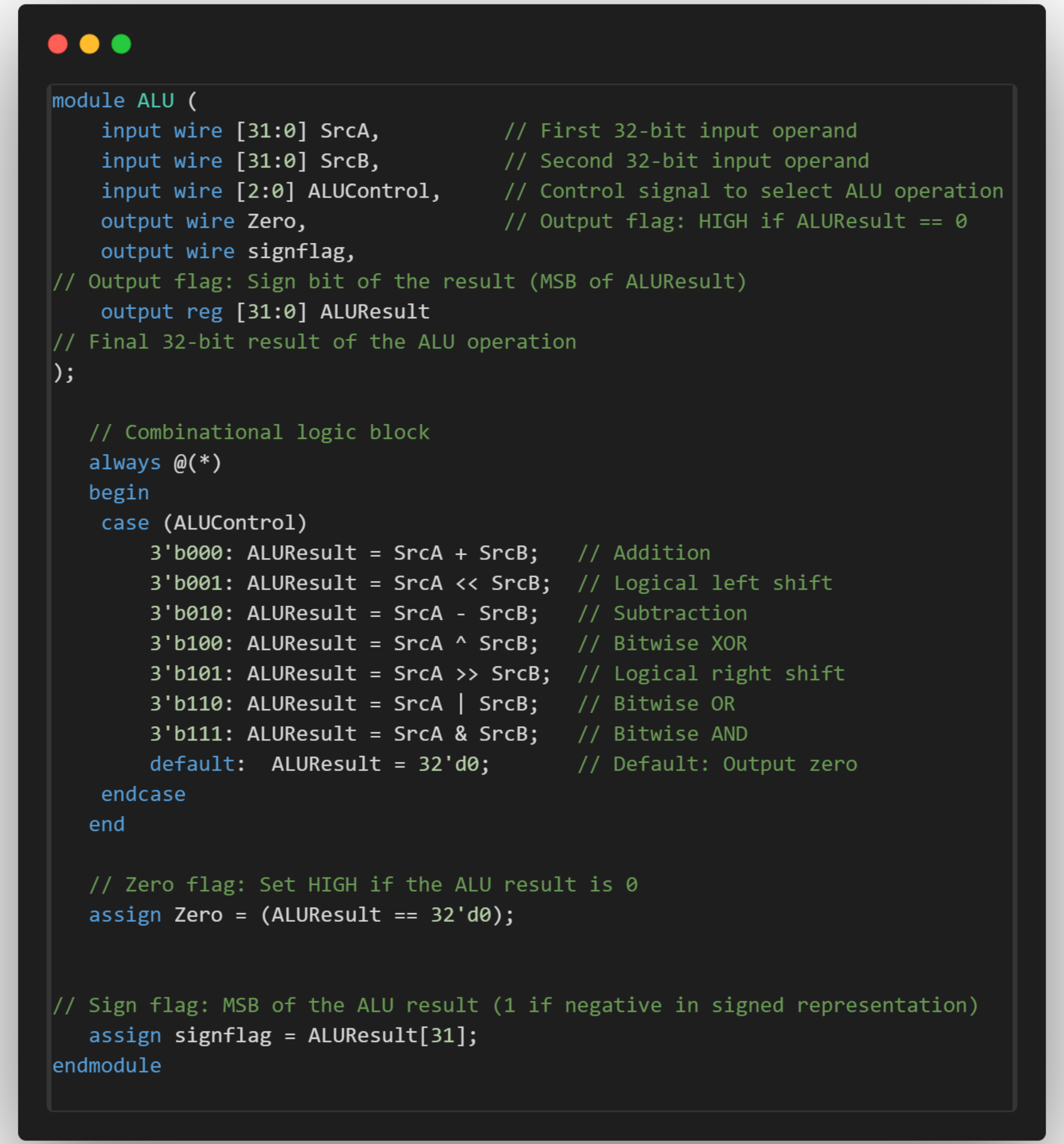


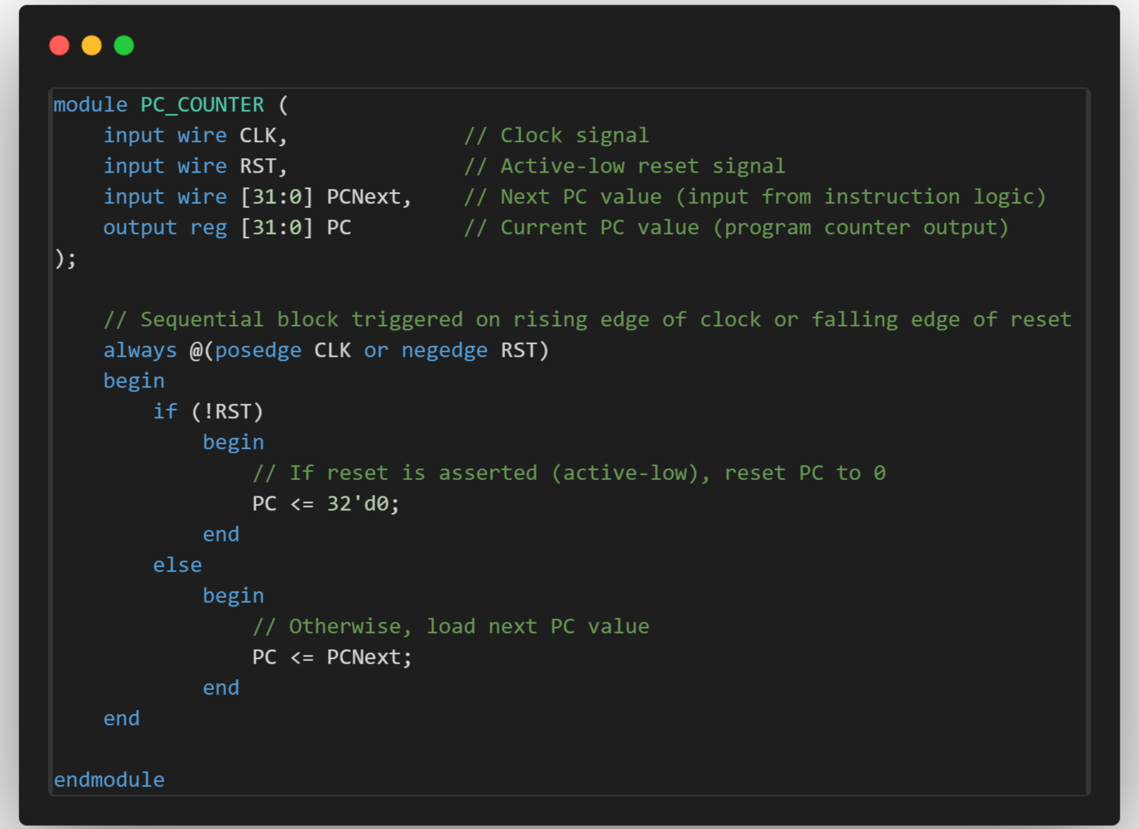
Figure 1 Complete single-cycle RISC-V processor

ALU Module:



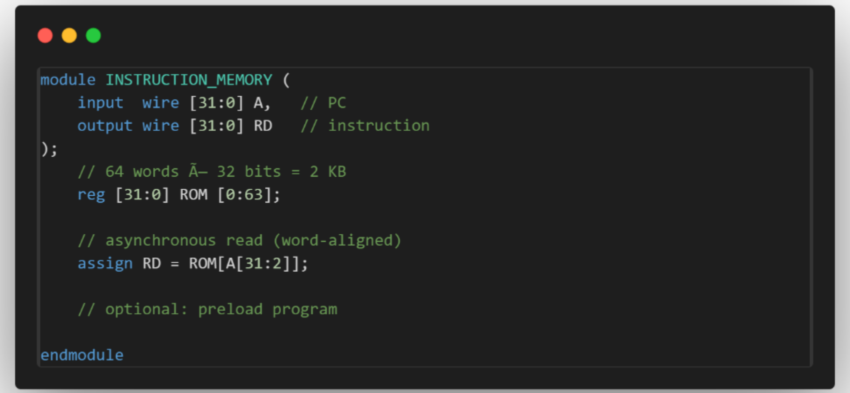
```
module ALU (
    input wire [31:0] SrcA,          // First 32-bit input operand
    input wire [31:0] SrcB,          // Second 32-bit input operand
    input wire [2:0] ALUControl,     // Control signal to select ALU operation
    output wire Zero,                // Output flag: HIGH if ALUResult == 0
    output wire signflag,
// Output flag: Sign bit of the result (MSB of ALUResult)
    output reg [31:0] ALUResult
// Final 32-bit result of the ALU operation
);

   // Combinational logic block
   always @(*)
   begin
    case (ALUControl)
        3'b000: ALUResult = SrcA + SrcB;   // Addition
        3'b001: ALUResult = SrcA << SrcB;  // Logical left shift
        3'b010: ALUResult = SrcA - SrcB;   // Subtraction
        3'b100: ALUResult = SrcA ^ SrcB;   // Bitwise XOR
        3'b101: ALUResult = SrcA >> SrcB;  // Logical right shift
        3'b110: ALUResult = SrcA | SrcB;   // Bitwise OR
        3'b111: ALUResult = SrcA & SrcB;   // Bitwise AND
        default:  ALUResult = 32'd0;       // Default: Output zero
    endcase
   end

   // Zero flag: Set HIGH if the ALU result is 0
   assign Zero = (ALUResult == 32'd0);


// Sign flag: MSB of the ALU result (1 if negative in signed representation)
   assign signflag = ALUResult[31];
endmodule
```

PC Counter Module:



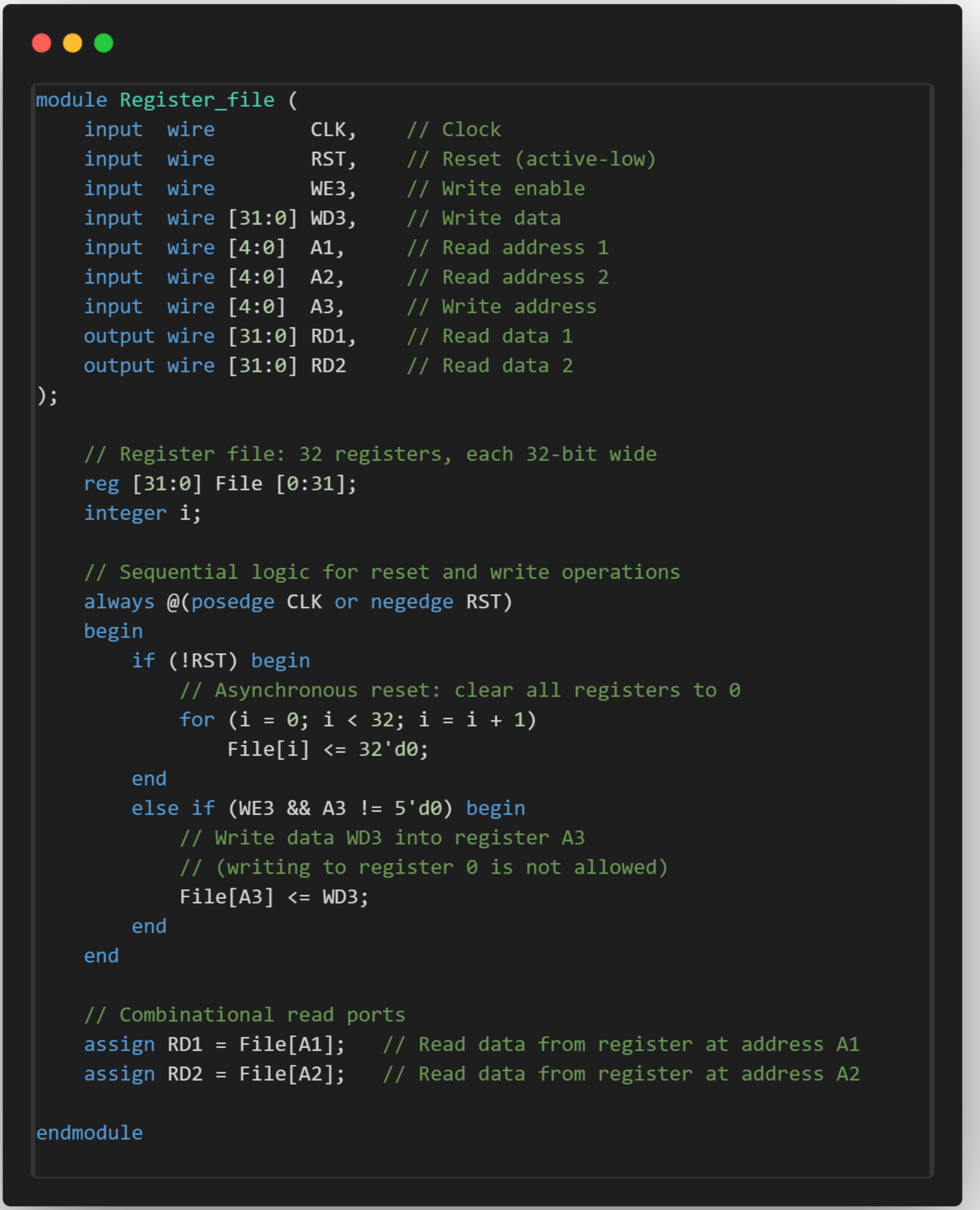
```verilog
module PC_COUNTER (
    input wire CLK,              // Clock signal
    input wire RST,              // Active-low reset signal
    input wire [31:0] PCNext,    // Next PC value (input from instruction logic)
    output reg [31:0] PC         // Current PC value (program counter output)
);

    // Sequential block triggered on rising edge of clock or falling edge of reset
    always @(posedge CLK or negedge RST)
    begin
        if (!RST)
            begin
                // If reset is asserted (active-low), reset PC to 0
                PC <= 32'd0;
            end
        else
            begin
                // Otherwise, load next PC value
                PC <= PCNext;
            end
    end

endmodule
```

Instruction memory:



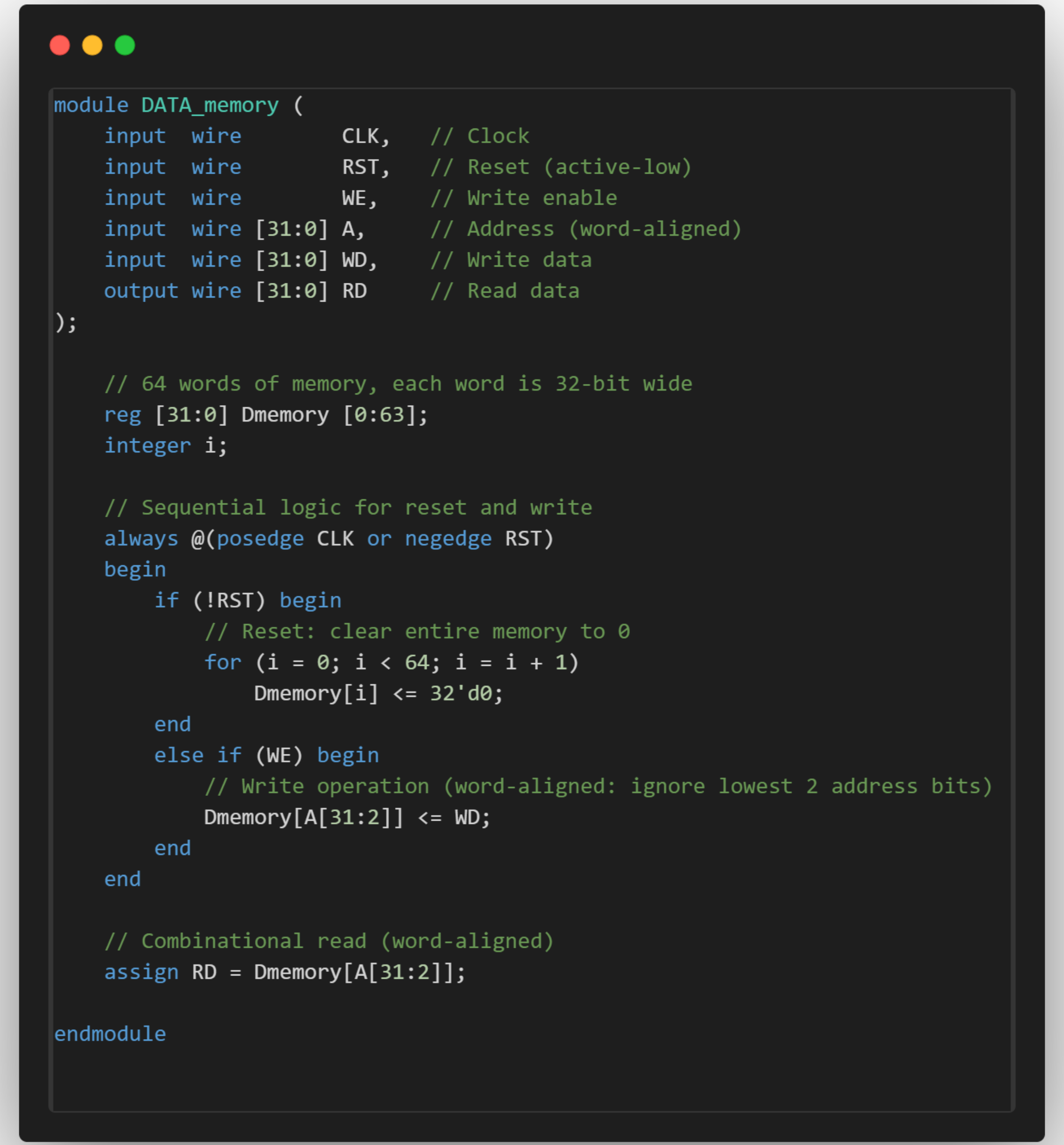
```verilog
module INSTRUCTION_MEMORY (
    input  wire [31:0] A,   // PC
    output wire [31:0] RD   // instruction
);
    // 64 words Ã— 32 bits = 2 KB
    reg [31:0] ROM [0:63];

    // asynchronous read (word-aligned)
    assign RD = ROM[A[31:2]];

    // optional: preload program

endmodule
```

Register File:



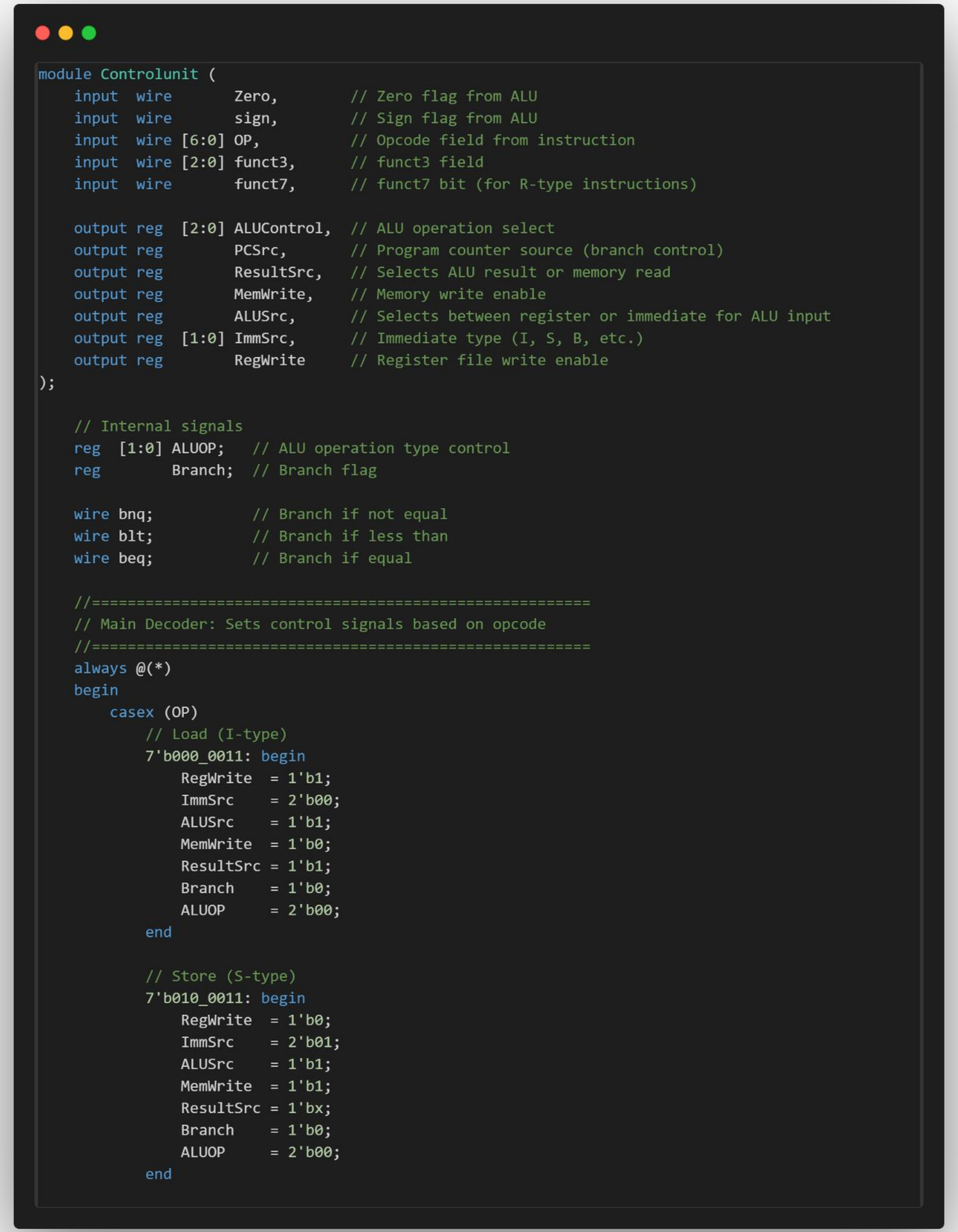
```verilog
module Register_file (
    input  wire        CLK,    // Clock
    input  wire        RST,    // Reset (active-low)
    input  wire        WE3,    // Write enable
    input  wire [31:0] WD3,    // Write data
    input  wire [4:0]  A1,     // Read address 1
    input  wire [4:0]  A2,     // Read address 2
    input  wire [4:0]  A3,     // Write address
    output wire [31:0] RD1,    // Read data 1
    output wire [31:0] RD2     // Read data 2
);

    // Register file: 32 registers, each 32-bit wide
    reg [31:0] File [0:31];
    integer i;

    // Sequential logic for reset and write operations
    always @(posedge CLK or negedge RST)
    begin
        if (!RST) begin
            // Asynchronous reset: clear all registers to 0
            for (i = 0; i < 32; i = i + 1)
                File[i] <= 32'd0;
        end
        else if (WE3 && A3 != 5'd0) begin
            // Write data WD3 into register A3
            // (writing to register 0 is not allowed)
            File[A3] <= WD3;
        end
    end

    // Combinational read ports
    assign RD1 = File[A1];   // Read data from register at address A1
    assign RD2 = File[A2];   // Read data from register at address A2

endmodule
```

Data Memory:

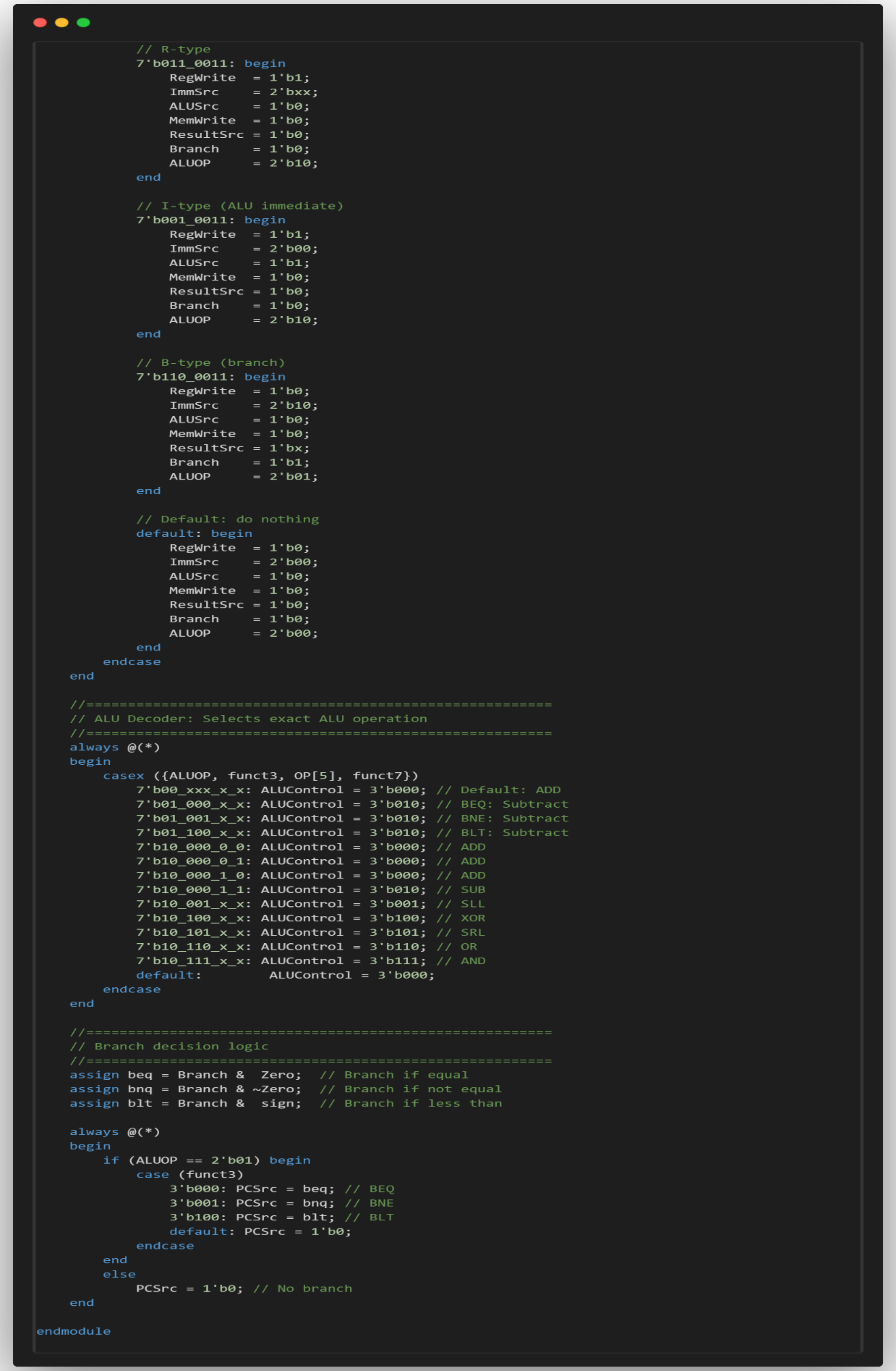


```verilog
module DATA_memory (
    input  wire        CLK,    // Clock
    input  wire        RST,    // Reset (active-low)
    input  wire        WE,     // Write enable
    input  wire [31:0] A,      // Address (word-aligned)
    input  wire [31:0] WD,     // Write data
    output wire [31:0] RD      // Read data
);

    // 64 words of memory, each word is 32-bit wide
    reg [31:0] Dmemory [0:63];
    integer i;

    // Sequential logic for reset and write
    always @(posedge CLK or negedge RST)
    begin
        if (!RST) begin
            // Reset: clear entire memory to 0
            for (i = 0; i < 64; i = i + 1)
                Dmemory[i] <= 32'd0;
        end
        else if (WE) begin
            // Write operation (word-aligned: ignore lowest 2 address bits)
            Dmemory[A[31:2]] <= WD;
        end
    end

    // Combinational read (word-aligned)
    assign RD = Dmemory[A[31:2]];

endmodule
```

ControlUnit:

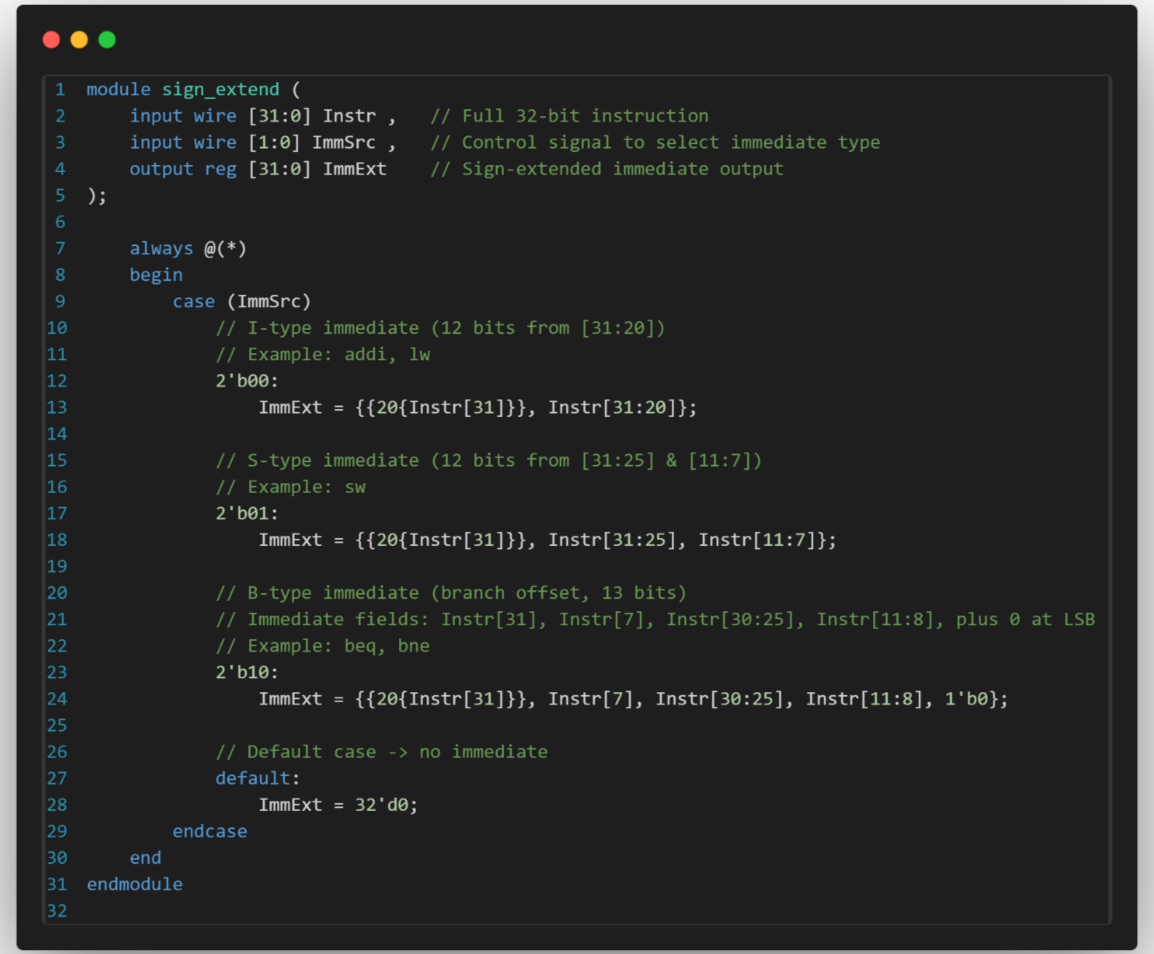


```
module Controlunit (
    input  wire       Zero,        // Zero flag from ALU
    input  wire       sign,        // Sign flag from ALU
    input  wire [6:0] OP,          // Opcode field from instruction
    input  wire [2:0] funct3,      // funct3 field
    input  wire       funct7,      // funct7 bit (for R-type instructions)

    output reg  [2:0] ALUControl,  // ALU operation select
    output reg        PCSrc,       // Program counter source (branch control)
    output reg        ResultSrc,   // Selects ALU result or memory read
    output reg        MemWrite,    // Memory write enable
    output reg        ALUSrc,      // Selects between register or immediate for ALU input
    output reg  [1:0] ImmSrc,      // Immediate type (I, S, B, etc.)
    output reg        RegWrite     // Register file write enable
);

    // Internal signals
    reg  [1:0] ALUOP;   // ALU operation type control
    reg        Branch;  // Branch flag

    wire bnq;           // Branch if not equal
    wire blt;           // Branch if less than
    wire beq;           // Branch if equal

    //========================================================
    // Main Decoder: Sets control signals based on opcode
    //========================================================
    always @(*)
    begin
        casex (OP)
            // Load (I-type)
            7'b000_0011: begin
                RegWrite  = 1'b1;
                ImmSrc    = 2'b00;
                ALUSrc    = 1'b1;
                MemWrite  = 1'b0;
                ResultSrc = 1'b1;
                Branch    = 1'b0;
                ALUOP     = 2'b00;
            end

            // Store (S-type)
            7'b010_0011: begin
                RegWrite  = 1'b0;
                ImmSrc    = 2'b01;
                ALUSrc    = 1'b1;
                MemWrite  = 1'b1;
                ResultSrc = 1'bx;
                Branch    = 1'b0;
                ALUOP     = 2'b00;
            end
```



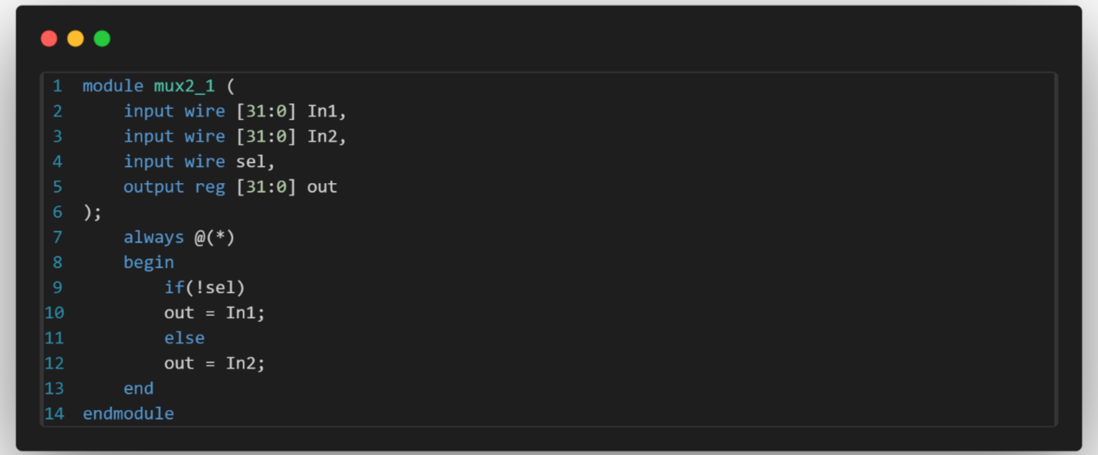
```verilog
            // R-type
            7'b011_0011: begin
                RegWrite  = 1'b1;
                ImmSrc    = 2'bxx;
                ALUSrc    = 1'b0;
                MemWrite  = 1'b0;
                ResultSrc = 1'b0;
                Branch    = 1'b0;
                ALUOP     = 2'b10;
            end

            // I-type (ALU immediate)
            7'b001_0011: begin
                RegWrite  = 1'b1;
                ImmSrc    = 2'b00;
                ALUSrc    = 1'b1;
                MemWrite  = 1'b0;
                ResultSrc = 1'b0;
                Branch    = 1'b0;
                ALUOP     = 2'b10;
            end

            // B-type (branch)
            7'b110_0011: begin
                RegWrite  = 1'b0;
                ImmSrc    = 2'b10;
                ALUSrc    = 1'b0;
                MemWrite  = 1'b0;
                ResultSrc = 1'bx;
                Branch    = 1'b1;
                ALUOP     = 2'b01;
            end

            // Default: do nothing
            default: begin
                RegWrite  = 1'b0;
                ImmSrc    = 2'b00;
                ALUSrc    = 1'b0;
                MemWrite  = 1'b0;
                ResultSrc = 1'b0;
                Branch    = 1'b0;
                ALUOP     = 2'b00;
            end
        endcase
    end

    //========================================================
    // ALU Decoder: Selects exact ALU operation
    //========================================================
    always @(*)
    begin
        casex ({ALUOP, funct3, OP[5], funct7})
            7'b00_xxx_x_x: ALUControl = 3'b000; // Default: ADD
            7'b01_000_x_x: ALUControl = 3'b010; // BEQ: Subtract
            7'b01_001_x_x: ALUControl = 3'b010; // BNE: Subtract
            7'b01_100_x_x: ALUControl = 3'b010; // BLT: Subtract
            7'b10_000_0_0: ALUControl = 3'b000; // ADD
            7'b10_000_0_1: ALUControl = 3'b000; // ADD
            7'b10_000_1_0: ALUControl = 3'b000; // ADD
            7'b10_000_1_1: ALUControl = 3'b010; // SUB
            7'b10_001_x_x: ALUControl = 3'b001; // SLL
            7'b10_100_x_x: ALUControl = 3'b100; // XOR
            7'b10_101_x_x: ALUControl = 3'b101; // SRL
            7'b10_110_x_x: ALUControl = 3'b110; // OR
            7'b10_111_x_x: ALUControl = 3'b111; // AND
            default:        ALUControl = 3'b000;
        endcase
    end

    //========================================================
    // Branch decision logic
    //========================================================
    assign beq = Branch &  Zero;  // Branch if equal
    assign bnq = Branch & ~Zero;  // Branch if not equal
    assign blt = Branch &  sign;  // Branch if less than

    always @(*)
    begin
        if (ALUOP == 2'b01) begin
            case (funct3)
                3'b000: PCSrc = beq; // BEQ
                3'b001: PCSrc = bnq; // BNE
                3'b100: PCSrc = blt; // BLT
                default: PCSrc = 1'b0;
            endcase
        end
        else
            PCSrc = 1'b0; // No branch
    end

endmodule
```

Signed Extend:



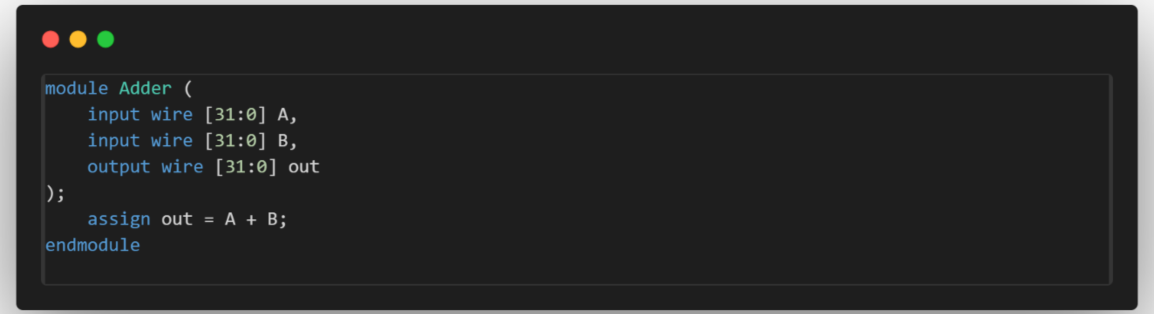
```
module sign_extend (
    input wire [31:0] Instr ,    // Full 32-bit instruction
    input wire [1:0] ImmSrc ,    // Control signal to select immediate type
    output reg [31:0] ImmExt     // Sign-extended immediate output
);

    always @(*)
    begin
        case (ImmSrc)
            // I-type immediate (12 bits from [31:20])
            // Example: addi, lw
            2'b00:
                ImmExt = {{20{Instr[31]}}, Instr[31:20]};

            // S-type immediate (12 bits from [31:25] & [11:7])
            // Example: sw
            2'b01:
                ImmExt = {{20{Instr[31]}}, Instr[31:25], Instr[11:7]};

            // B-type immediate (branch offset, 13 bits)
            // Immediate fields: Instr[31], Instr[7], Instr[30:25], Instr[11:8], plus 0 at LSB
            // Example: beq, bne
            2'b10:
                ImmExt = {{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0};

            // Default case -> no immediate
            default:
                ImmExt = 32'd0;
        endcase
    end
endmodule

```

Mux:



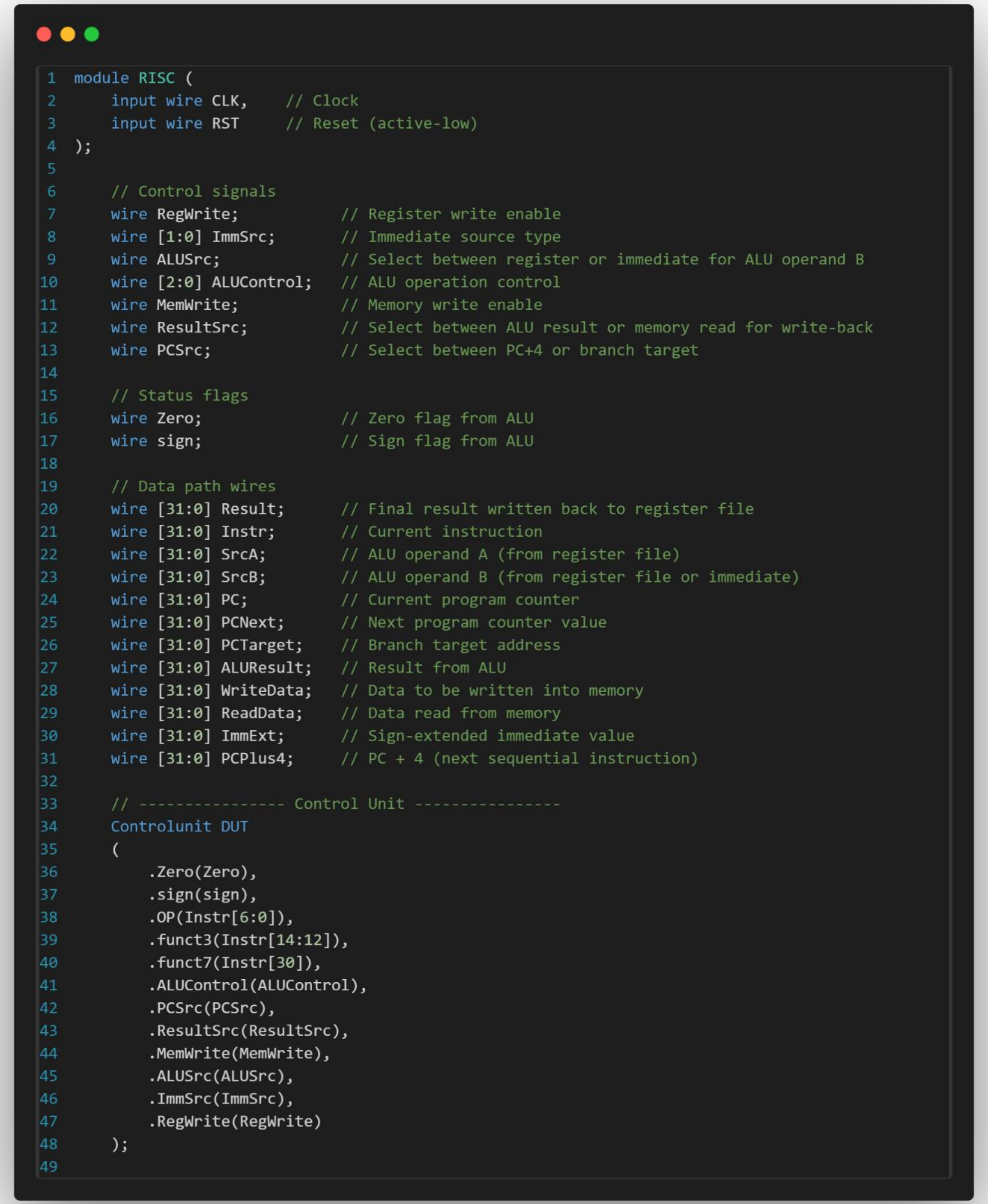
```
module mux2_1 (
    input wire [31:0] In1,
    input wire [31:0] In2,
    input wire sel,
    output reg [31:0] out
);
    always @(*)
    begin
        if(!sel)
        out = In1;
        else
        out = In2;
    end
endmodule
```

Adder:

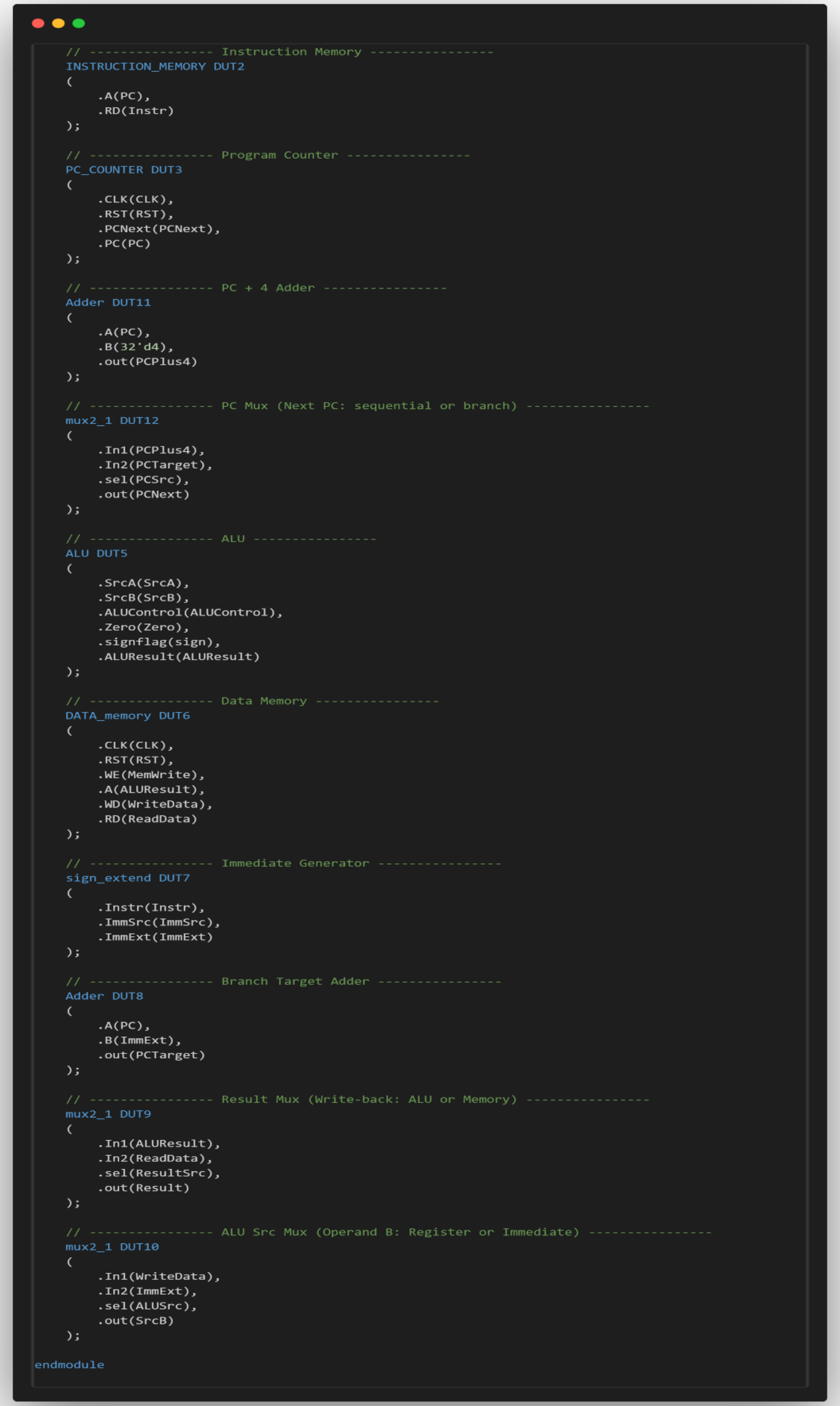


```
module Adder (
    input wire [31:0] A,
    input wire [31:0] B,
    output wire [31:0] out
);
    assign out = A + B;
endmodule
```

Risc Top Module:



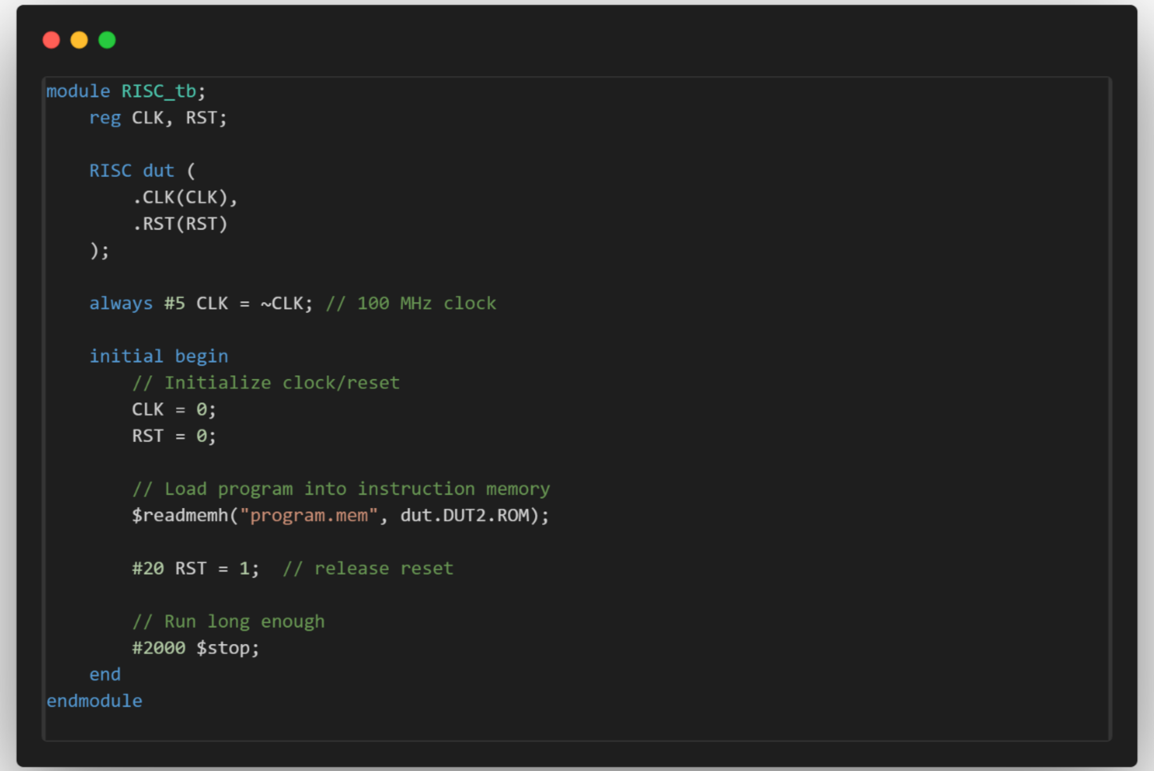
```verilog
module RISC (
    input wire CLK,    // Clock
    input wire RST     // Reset (active-low)
);

    // Control signals
    wire RegWrite;           // Register write enable
    wire [1:0] ImmSrc;       // Immediate source type
    wire ALUSrc;             // Select between register or immediate for ALU operand B
    wire [2:0] ALUControl;   // ALU operation control
    wire MemWrite;           // Memory write enable
    wire ResultSrc;          // Select between ALU result or memory read for write-back
    wire PCSrc;              // Select between PC+4 or branch target

    // Status flags
    wire Zero;               // Zero flag from ALU
    wire sign;               // Sign flag from ALU

    // Data path wires
    wire [31:0] Result;      // Final result written back to register file
    wire [31:0] Instr;       // Current instruction
    wire [31:0] SrcA;        // ALU operand A (from register file)
    wire [31:0] SrcB;        // ALU operand B (from register file or immediate)
    wire [31:0] PC;          // Current program counter
    wire [31:0] PCNext;      // Next program counter value
    wire [31:0] PCTarget;    // Branch target address
    wire [31:0] ALUResult;   // Result from ALU
    wire [31:0] WriteData;   // Data to be written into memory
    wire [31:0] ReadData;    // Data read from memory
    wire [31:0] ImmExt;      // Sign-extended immediate value
    wire [31:0] PCPlus4;     // PC + 4 (next sequential instruction)

    // ---------------- Control Unit ----------------
    Controlunit DUT
    (
        .Zero(Zero),
        .sign(sign),
        .OP(Instr[6:0]),
        .funct3(Instr[14:12]),
        .funct7(Instr[30]),
        .ALUControl(ALUControl),
        .PCSrc(PCSrc),
        .ResultSrc(ResultSrc),
        .MemWrite(MemWrite),
        .ALUSrc(ALUSrc),
        .ImmSrc(ImmSrc),
        .RegWrite(RegWrite)
    );

```



```verilog
    // ---------------- Instruction Memory ----------------
    INSTRUCTION_MEMORY DUT2
    (
        .A(PC),
        .RD(Instr)
    );

    // ---------------- Program Counter ----------------
    PC_COUNTER DUT3
    (
        .CLK(CLK),
        .RST(RST),
        .PCNext(PCNext),
        .PC(PC)
    );

    // ---------------- PC + 4 Adder ----------------
    Adder DUT11
    (
        .A(PC),
        .B(32'd4),
        .out(PCPlus4)
    );

    // ---------------- PC Mux (Next PC: sequential or branch) ----------------
    mux2_1 DUT12
    (
        .In1(PCPlus4),
        .In2(PCTarget),
        .sel(PCSrc),
        .out(PCNext)
    );

    // ---------------- ALU ----------------
    ALU DUT5
    (
        .SrcA(SrcA),
        .SrcB(SrcB),
        .ALUControl(ALUControl),
        .Zero(Zero),
        .signflag(sign),
        .ALUResult(ALUResult)
    );

    // ---------------- Data Memory ----------------
    DATA_memory DUT6
    (
        .CLK(CLK),
        .RST(RST),
        .WE(MemWrite),
        .A(ALUResult),
        .WD(WriteData),
        .RD(ReadData)
    );

    // ---------------- Immediate Generator ----------------
    sign_extend DUT7
    (
        .Instr(Instr),
        .ImmSrc(ImmSrc),
        .ImmExt(ImmExt)
    );

    // ---------------- Branch Target Adder ----------------
    Adder DUT8
    (
        .A(PC),
        .B(ImmExt),
        .out(PCTarget)
    );

    // ---------------- Result Mux (Write-back: ALU or Memory) ----------------
    mux2_1 DUT9
    (
        .In1(ALUResult),
        .In2(ReadData),
        .sel(ResultSrc),
        .out(Result)
    );

    // ---------------- ALU Src Mux (Operand B: Register or Immediate) ----------------
    mux2_1 DUT10
    (
        .In1(WriteData),
        .In2(ImmExt),
        .sel(ALUSrc),
        .out(SrcB)
    );

endmodule
```

Simple TB :

To check FIBONACCI series



```
module RISC_tb;
    reg CLK, RST;

    RISC dut (
        .CLK(CLK),
        .RST(RST)
    );

    always #5 CLK = ~CLK; // 100 MHz clock

    initial begin
        // Initialize clock/reset
        CLK = 0;
        RST = 0;

        // Load program into instruction memory
        $readmemh("program.mem", dut.DUT2.ROM);

        #20 RST = 1;  // release reset

        // Run long enough
        #2000 $stop;
    end
endmodule
```

Results:

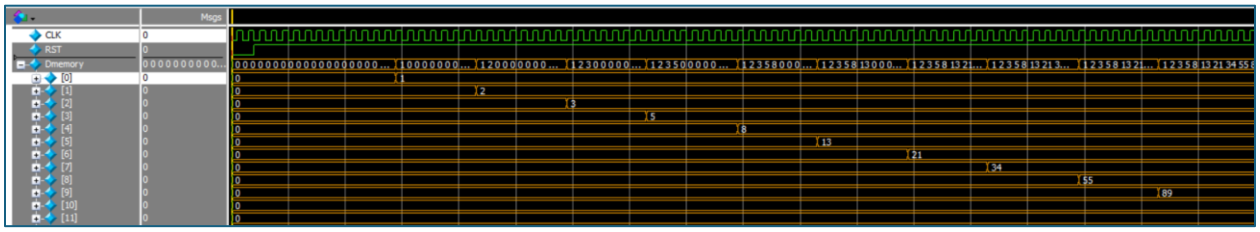


Figure2 Waveform

Elaborated Design:

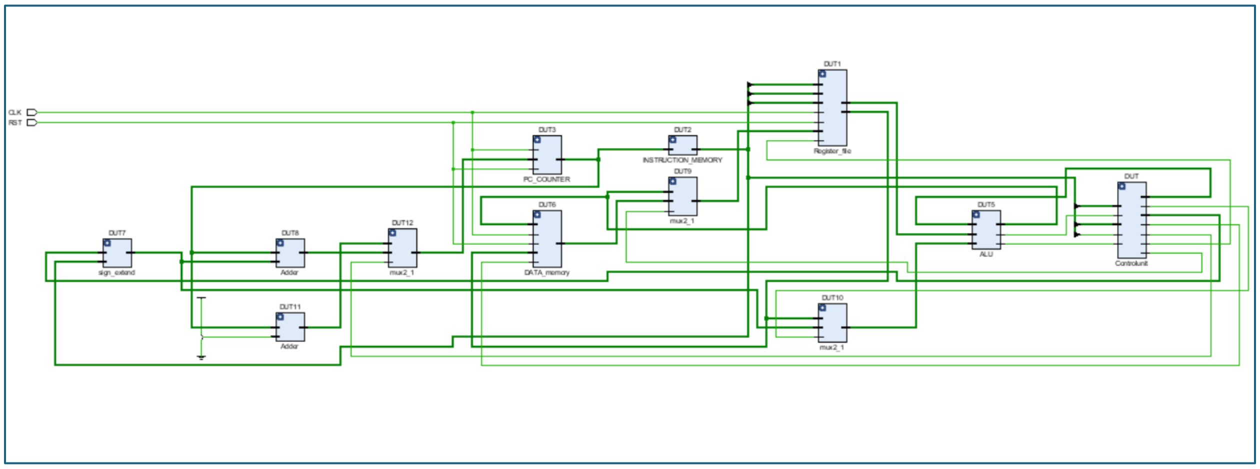


Figure 3 Elaborated Design