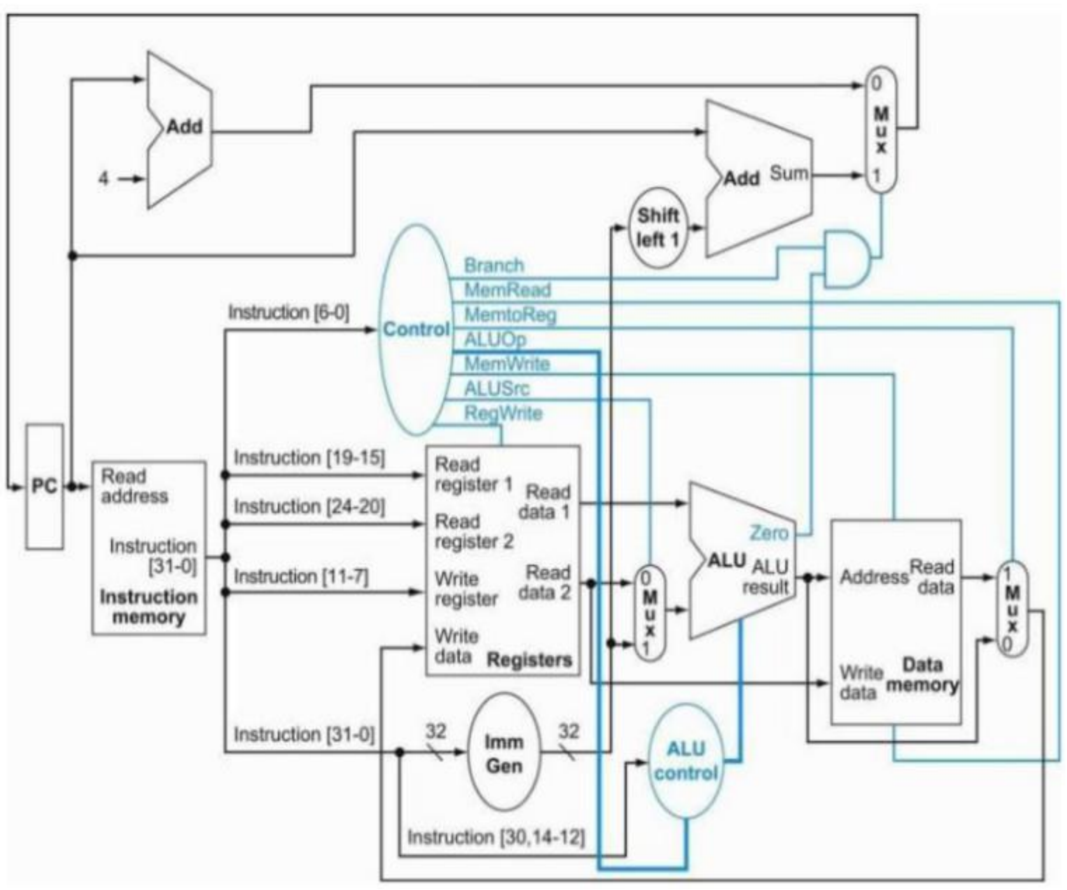
Single Cycle RISC V Processor



1. Implemented a single cycle RISC-V processor using the architecture shown in figure which supports the following instructions – add, sub, and, or, sll, slt,sltu, xor, srl, sra, addi, slti, sltui, xori, ori, andi,slli, srli, srai, sw, lw, beq.

SingleCycleCPU:

```
module SingleCycleCPU (
    input clk,           // Clock signal
    input start          // Start signal (reset control)
);
    wire [31:0] pc, next_pc, instruction, readData1, readData2, aluResult, imm, writeData, aluIn2,
o,readDataMem;
    wire [3:0] aluctl;
    wire aluSrc, branch, memRead, memtoReg, memWrite, regWrite, zero;
    wire [1:0] ALUOp;

    // Program Counter (PC)
    PC pcModule (
        .clk(clk),
        .rst(~start),
        .pc_i(next_pc),   // Next PC address (either PC+4 or branch address)
        .pc_o(pc)         // Current PC address
    );

    // Instruction Memory to fetch the instruction
    InstructionMemory instrMem (
        .readAddr(pc),
        .inst(instruction)   // Fetched instruction
    );

    // Control Unit that generates control signals based on the opcode
    Control controlUnit (
        .opcode(instruction[6:0]),  // Opcode from instruction
        .branch(branch),
        .memRead(memRead),
        .memtoReg(memtoReg),
        .ALUOp(ALUOp),
        .memWrite(memWrite),
        .ALUSrc(aluSrc),
        .regWrite(regWrite)
    );
```

```
// Register File for reading/writing registers
Register regFile (
    .clk(clk),
    .rst(~start),
    .regWrite(regWrite),
    .readReg1(instruction[19:15]),  // rs1 field
    .readReg2(instruction[24:20]),  // rs2 field
    .writeReg(instruction[11:7]),   // rd field
    .writeData(writeData),          // Data to write to rd
    .readData1(readData1),          // Data from rs1
    .readData2(readData2)           // Data from rs2
);

// Immediate Generator
ImmGen immGen (
    .inst(instruction),  // Instruction input
    .imm(imm)            // Immediate output
);

// ALU Control for determining the ALU operation
ALUCtrl aluControl (
    .ALUOp(ALUOp),
    .funct7(instruction[30]),       // funct7 from instruction
    .funct3(instruction[14:12]),   // funct3 from instruction
    .ALUCtl(aluctl)                 // ALU control output
);

// Mux to select between register and immediate value for ALU operand B
Mux2to1 aluSrcMux (
    .sel(aluSrc),
    .s0(readData2),  // Register value (rs2)
    .s1(imm),        // Immediate value
    .out(aluIn2)     // ALU input B
);

// ALU execution
ALU alu (
    .ALUCtl(aluctl),  // ALU control signal
    .A(readData1),    // ALU input A (rs1)
    .B(aluIn2),       // ALU input B (either rs2 or immediate)
    .ALUOut(aluResult), // ALU result
    .zero(zero)       // Zero flag for branch decision
);
```

```
    // Data Memory for memory access (load/store)
    DataMemory dataMemory (
        .rst(~start),
        .clk(clk),
        .memWrite(memWrite),      // Memory write enable
        .memRead(memRead),        // Memory read enable
        .address(aluResult),      // Memory address (ALU result)
        .writeData(readData2),    // Data to write (rs2)
        .readData(readDataMem)    // Data read from memory
    );

    // Mux to select between ALU result and data from memory for register write-back
    Mux2to1 wbMux (
        .sel(memtoReg),
        .s0(aluResult),     // ALU result
        .s1(readDataMem),   // Data read from memory
        .out(writeData)     // Data to write back to the register
    );

    // PC calculation: next PC is either PC+4 or branch target
    wire [31:0] pcPlus4, branchAddr;
    Adder pcAdder (
        .a(pc),
        .b(32'd4),          // Increment PC by 4
        .sum(pcPlus4)       // PC + 4
    );

    // Shift left by 1 for branch address calculation
ShiftLeftOne m_ShiftLeftOne(
            .i(imm),
            .o(o)
);

    // Branch address calculation
    Adder branchAdder (
        .a(pc),
        .b(o),
        .sum(branchAddr)    // Branch target address
    );

    // Mux to select next PC: either PC+4 or branch address
    Mux2to1 branchMux (
        .sel(branch & zero),  // Take the branch if branch is true and zero flag is set
        .s0(pcPlus4),          // PC + 4
        .s1(branchAddr),       // Branch target address
        .out(next_pc)          // Next PC address
    );

endmodule
```

//Submodules:
//ALU:

```
module ALU (
    input [3:0] ALUCtl,
    input [31:0] A,B,
    output reg [31:0] ALUOut,
    output zero
);
    // Zero is set if ALUOut is zero
    assign zero = (ALUOut == 0);

    always @(*) begin
        case (ALUCtl)
            4'b0000: ALUOut = A & B;          // AND
            4'b0001: ALUOut = A | B;          // OR
            4'b0010: ALUOut = A + B;          // ADD
            4'b0110: ALUOut = A - B;          // SUB
            4'b0011: ALUOut = A ^ B;          // XOR
            4'b1000: ALUOut = A << B[4:0]; // SLL / SLLI (Shift Left Logical)
            4'b1001: ALUOut = A >> B[4:0]; // SRL / SRLI (Shift Right Logical)
            4'b1010: ALUOut = $signed(A) >>> B[4:0]; // SRA / SRAI (Shift Right Arithmetic)
            4'b0111: ALUOut = ($signed(A) < $signed(B)) ? 32'b1 : 32'b0;  // SLT / SLTI (signed
comparison)
            4'b1011: ALUOut = (A < B) ? 32'b1 : 32'b0;  // SLTU / SLTIU (unsigned comparison)
            4'b1100: ALUOut = ~(A | B);      // NOR
            default: ALUOut = 32'b0;         // Default case
        endcase
    end

endmodule
```

//ALUCtrl:

```
module ALUCtrl (
    input [1:0] ALUOp,
    input funct7,
    input [2:0] funct3,
    output reg [3:0] ALUCtl
);

always @(*) begin
    case(ALUOp)
        2'b00: ALUCtl = 4'b0010; // ADD for LW/SW
        2'b01: ALUCtl = 4'b0110; // SUB for BEQ
        2'b10: begin
            case({funct7, funct3})
                4'b0000: ALUCtl = 4'b0010; // ADD
                4'b1000: ALUCtl = 4'b0110; // SUB
                4'b0111: ALUCtl = 4'b0000; // AND
                4'b0110: ALUCtl = 4'b0001; // OR
                4'b0100: ALUCtl = 4'b0011;  // XOR
                4'b0001: ALUCtl = 4'b1000;  // SLL (funct7=0)
                4'b0101: begin              // SRL/SRA
                    if (funct7 == 1'b0)
                        ALUCtl = 4'b1001;   // SRL
                    else
                        ALUCtl = 4'b1010;   // SRA
                end
                4'b0010: ALUCtl = 4'b0111; // SLT
                4'b0011: ALUCtl = 4'b1011;  // SLTU (unsigned comparison)
                default: ALUCtl = 4'b0000; // Default
            endcase
        end
        2'b11: begin  // Immediate-type (I-type) instructions (like ADDI, ORI, etc.)
            case(funct3)
                3'b000: ALUCtl = 4'b0010;  // ADDI (addition with immediate)
                3'b111: ALUCtl = 4'b0000;  // ANDI (AND with immediate)
                3'b110: ALUCtl = 4'b0001;  // ORI  (OR with immediate)
                3'b100: ALUCtl = 4'b0011; //XORI
                 3'b001: ALUCtl = 4'b1000;  // SLLI
                3'b101: begin              // SRLI/SRAI
                    if (funct7 == 1'b0)
                        ALUCtl = 4'b1001;  // SRLI
                    else
                        ALUCtl = 4'b1010;  // SRAI
                end
                3'b010: ALUCtl = 4'b0111;  // SLTI (Set Less Than Immediate)
                3'b011: ALUCtl = 4'b1011;  // SLTIU (unsigned comparison)
                default: ALUCtl = 4'b0000; // Default case for unknown immediate instructions
            endcase
        end
        default: ALUCtl = 4'b0000; // Default case
    endcase
end


endmodule
```

//ImmGen:

```
module ImmGen#(parameter Width = 32) (
    input [Width-1:0] inst,
    output reg signed [Width-1:0] imm
);
    // ImmGen generate imm value based on opcode

    wire [6:0] opcode = inst[6:0];
    always @(*)
    begin
        case(opcode)
            7'b0010011: imm = {{20{inst[31]}}, inst[31:20]}; // I-type
            7'b0100011: imm = {{20{inst[31]}}, inst[31:25], inst[11:7]}; // S-type
            7'b1100011: imm = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0}; // B-type
            default: imm = 32'b0; // Default case
               endcase
    end

endmodule
```

//Control:

```
module Control (
    input [6:0] opcode,    // Opcode from the instruction
    output reg branch,      // Branch control signal
    output reg memRead,     // Memory read signal
    output reg memtoReg,    // Write data source (ALU result or memory)
    output reg [1:0] ALUOp, // ALU operation code
    output reg memWrite,    // Memory write signal
    output reg ALUSrc,      // ALU source (immediate or register)
    output reg regWrite      // Register write signal
);

always @(*) begin
    case(opcode)
        7'b0000011: begin // Load (LW)
```

```
            memRead = 1; memWrite = 0; regWrite = 1; ALUSrc = 1;
            branch = 0; memtoReg = 1; ALUOp = 2'b00;
        end
        7'b0100011: begin // Store (SW)
            memRead = 0; memWrite = 1; regWrite = 0; ALUSrc = 1;
            branch = 0; memtoReg = 0; ALUOp = 2'b00;
        end
        7'b1100011: begin // Branch (BEQ)
            memRead = 0; memWrite = 0; regWrite = 0; ALUSrc = 0;
            branch = 1; memtoReg = 0; ALUOp = 2'b01;
        end
        7'b0010011: begin // I-type ALU instructions (e.g., ADDI)
            memRead = 0; memWrite = 0; regWrite = 1; ALUSrc = 1;
            branch = 0; memtoReg = 0; ALUOp = 2'b11;
        end
        7'b0110011: begin // R-type
            memRead = 0; memWrite = 0; regWrite = 1; ALUSrc = 0;
            branch = 0; memtoReg = 0; ALUOp = 2'b10;
        end
        default: begin
            memRead = 0; memWrite = 0; regWrite = 0; ALUSrc = 0;
            branch = 0; memtoReg = 0; ALUOp = 2'b00;
        end
    endcase
end
endmodule
```

//DataMemory:

```
module DataMemory(
            input rst,
            input clk,
            input memWrite,
            input memRead,
            input [31:0] address,
            input [31:0] writeData,
            output reg [31:0] readData
);
            // Do not modify this file!

            reg [7:0] data_memory [127:0];
            always @ (posedge clk) begin
                        if(rst) begin
                                    data_memory[0] <= 8'b0;
```

```
data_memory[1] <= 8'b0;
data_memory[2] <= 8'b0;
data_memory[3] <= 8'b0;
data_memory[4] <= 8'b0;
data_memory[5] <= 8'b0;
data_memory[6] <= 8'b0;
data_memory[7] <= 8'b0;
data_memory[8] <= 8'b0;
data_memory[9] <= 8'b0;
data_memory[10] <= 8'b0;
data_memory[11] <= 8'b0;
data_memory[12] <= 8'b0;
data_memory[13] <= 8'b0;
data_memory[14] <= 8'b0;
data_memory[15] <= 8'b0;
data_memory[16] <= 8'b0;
data_memory[17] <= 8'b0;
data_memory[18] <= 8'b0;
data_memory[19] <= 8'b0;
data_memory[20] <= 8'b0;
data_memory[21] <= 8'b0;
data_memory[22] <= 8'b0;
data_memory[23] <= 8'b0;
data_memory[24] <= 8'b0;
data_memory[25] <= 8'b0;
data_memory[26] <= 8'b0;
data_memory[27] <= 8'b0;
data_memory[28] <= 8'b0;
data_memory[29] <= 8'b0;
data_memory[30] <= 8'b0;
data_memory[31] <= 8'b0;
data_memory[32] <= 8'b0;
data_memory[33] <= 8'b0;
data_memory[34] <= 8'b0;
data_memory[35] <= 8'b0;
data_memory[36] <= 8'b0;
data_memory[37] <= 8'b0;
data_memory[38] <= 8'b0;
data_memory[39] <= 8'b0;
data_memory[40] <= 8'b0;
data_memory[41] <= 8'b0;
data_memory[42] <= 8'b0;
data_memory[43] <= 8'b0;
data_memory[44] <= 8'b0;
```

```
data_memory[45] <= 8'b0;
data_memory[46] <= 8'b0;
data_memory[47] <= 8'b0;
data_memory[48] <= 8'b0;
data_memory[49] <= 8'b0;
data_memory[50] <= 8'b0;
data_memory[51] <= 8'b0;
data_memory[52] <= 8'b0;
data_memory[53] <= 8'b0;
data_memory[54] <= 8'b0;
data_memory[55] <= 8'b0;
data_memory[56] <= 8'b0;
data_memory[57] <= 8'b0;
data_memory[58] <= 8'b0;
data_memory[59] <= 8'b0;
data_memory[60] <= 8'b0;
data_memory[61] <= 8'b0;
data_memory[62] <= 8'b0;
data_memory[63] <= 8'b0;
data_memory[64] <= 8'b0;
data_memory[65] <= 8'b0;
data_memory[66] <= 8'b0;
data_memory[67] <= 8'b0;
data_memory[68] <= 8'b0;
data_memory[69] <= 8'b0;
data_memory[70] <= 8'b0;
data_memory[71] <= 8'b0;
data_memory[72] <= 8'b0;
data_memory[73] <= 8'b0;
data_memory[74] <= 8'b0;
data_memory[75] <= 8'b0;
data_memory[76] <= 8'b0;
data_memory[77] <= 8'b0;
data_memory[78] <= 8'b0;
data_memory[79] <= 8'b0;
data_memory[80] <= 8'b0;
data_memory[81] <= 8'b0;
data_memory[82] <= 8'b0;
data_memory[83] <= 8'b0;
data_memory[84] <= 8'b0;
data_memory[85] <= 8'b0;
data_memory[86] <= 8'b0;
data_memory[87] <= 8'b0;
data_memory[88] <= 8'b0;
```

```
                data_memory[89] <= 8'b0;
                data_memory[90] <= 8'b0;
                data_memory[91] <= 8'b0;
                data_memory[92] <= 8'b0;
                data_memory[93] <= 8'b0;
                data_memory[94] <= 8'b0;
                data_memory[95] <= 8'b0;
                data_memory[96] <= 8'b0;
                data_memory[97] <= 8'b0;
                data_memory[98] <= 8'b0;
                data_memory[99] <= 8'b0;
                data_memory[100] <= 8'b0;
                data_memory[101] <= 8'b0;
                data_memory[102] <= 8'b0;
                data_memory[103] <= 8'b0;
                data_memory[104] <= 8'b0;
                data_memory[105] <= 8'b0;
                data_memory[106] <= 8'b0;
                data_memory[107] <= 8'b0;
                data_memory[108] <= 8'b0;
                data_memory[109] <= 8'b0;
                data_memory[110] <= 8'b0;
                data_memory[111] <= 8'b0;
                data_memory[112] <= 8'b0;
                data_memory[113] <= 8'b0;
                data_memory[114] <= 8'b0;
                data_memory[115] <= 8'b0;
                data_memory[116] <= 8'b0;
                data_memory[117] <= 8'b0;
                data_memory[118] <= 8'b0;
                data_memory[119] <= 8'b0;
                data_memory[120] <= 8'b0;
                data_memory[121] <= 8'b0;
                data_memory[122] <= 8'b0;
                data_memory[123] <= 8'b0;
                data_memory[124] <= 8'b0;
                data_memory[125] <= 8'b0;
                data_memory[126] <= 8'b0;
                data_memory[127] <= 8'b0;
        end
        else begin
                if(memWrite) begin
                        data_memory[address + 3] <= writeData[31:24];
                        data_memory[address + 2] <= writeData[23:16];
                        data_memory[address + 1] <= writeData[15:8];
                        data_memory[address]     <= writeData[7:0];
                    end

                    end
        end

        always @(*) begin
                if(memRead) begin
                        readData[31:24]  = data_memory[address + 3];
                        readData[23:16]  = data_memory[address + 2];
                        readData[15:8]   = data_memory[address + 1];
                        readData[7:0]    = data_memory[address];
                end
                else begin
                        readData         = 32'b0;
                end
        end

endmodule
```

//Register:

```
module Register (
    input clk,
    input rst,
    input regWrite,
    input [4:0] readReg1,
    input [4:0] readReg2,
    input [4:0] writeReg,
    input [31:0] writeData,
    output [31:0] readData1,
    output [31:0] readData2
);
    reg [31:0] regs [0:31];

// Do not modify this file!
    assign readData1 = (readReg1!=0)?regs[readReg1]:0;
    assign readData2 = (readReg2!=0)?regs[readReg2]:0;

    always @(posedge clk) begin
        if(rst) begin
            regs[0] <= 0; regs[1] <= 0; regs[2] <= 32'd128; regs[3] <= 0;
            regs[4] <= 0; regs[5] <= 0; regs[6] <= 0; regs[7] <= 0;
            regs[8] <= 0; regs[9] <= 0; regs[10] <= 0; regs[11] <= 0;
            regs[12] <= 0; regs[13] <= 0; regs[14] <= 0; regs[15] <= 0;
            regs[16] <= 0; regs[17] <= 0; regs[18] <= 0; regs[19] <= 0;
            regs[20] <= 0; regs[21] <= 0; regs[22] <= 0; regs[23] <= 0;
            regs[24] <= 0; regs[25] <= 0; regs[26] <= 0; regs[27] <= 0;
            regs[28] <= 0; regs[29] <= 0; regs[30] <= 0; regs[31] <= 0;
        end
        else if(regWrite)
            regs[writeReg] <= (writeReg == 0) ? 0 : writeData;
    end

endmodule
```

//Adder:

```
module Adder (
    input signed [31:0] a,
    input signed [31:0] b,
    output signed [31:0] sum
);
    // Adder computes sum = a + b
    // The module is useful for incrementing PC

 assign sum = a + b;

endmodule
```

//PC:

```
module PC (
    input clk,
    input rst,
    input [31:0] pc_i,
    output reg [31:0] pc_o
);

always @(posedge clk ) begin
            if (rst)
                        pc_o <=32'b0;
            else
                        pc_o <= pc_i;
end
endmodule
```

//InstructionMemory:

```
module InstructionMemory (
    input [31:0] readAddr,
    output [31:0] inst
);

    // Do not modify this file!

    reg [7:0] insts [127:0];

    assign inst = (readAddr >= 128) ? 32'b0 : {insts[readAddr], insts[readAddr + 1], insts[readAddr
+ 2], insts[readAddr + 3]};

    initial begin
//        insts[0] = 8'b0;  insts[1] = 8'b0;  insts[2] = 8'b0;  insts[3] = 8'b0;
//        insts[4] = 8'b0;  insts[5] = 8'b0;  insts[6] = 8'b0;  insts[7] = 8'b0;
//        insts[8] = 8'b0;  insts[9] = 8'b0;  insts[10] = 8'b0; insts[11] = 8'b0;
//        insts[12] = 8'b0; insts[13] = 8'b0; insts[14] = 8'b0; insts[15] = 8'b0;
//        insts[16] = 8'b0; insts[17] = 8'b0; insts[18] = 8'b0; insts[19] = 8'b0;
//        insts[20] = 8'b0; insts[21] = 8'b0; insts[22] = 8'b0; insts[23] = 8'b0;
//        insts[24] = 8'b0; insts[25] = 8'b0; insts[26] = 8'b0; insts[27] = 8'b0;
//        insts[28] = 8'b0; insts[29] = 8'b0; insts[30] = 8'b0; insts[31] = 8'b0;
        $readmemb("TEST_INSTRUCTIONS.dat", insts);
    end

endmodule
```

//ShiftLeftOne:

```
module ShiftLeftOne (
    input signed [31:0] i,
    output signed [31:0] o
);

   assign o = i << 1;

endmodule
```

//Mux2to1:

```
module Mux2to1 #(
    parameter size = 32
)
```

```
(
    input sel,
    input signed [size-1:0] s0,
    input signed [size-1:0] s1,
    output signed [size-1:0] out
);

assign out = sel ? s1 : s0;

endmodule
```

//Test Bench file:

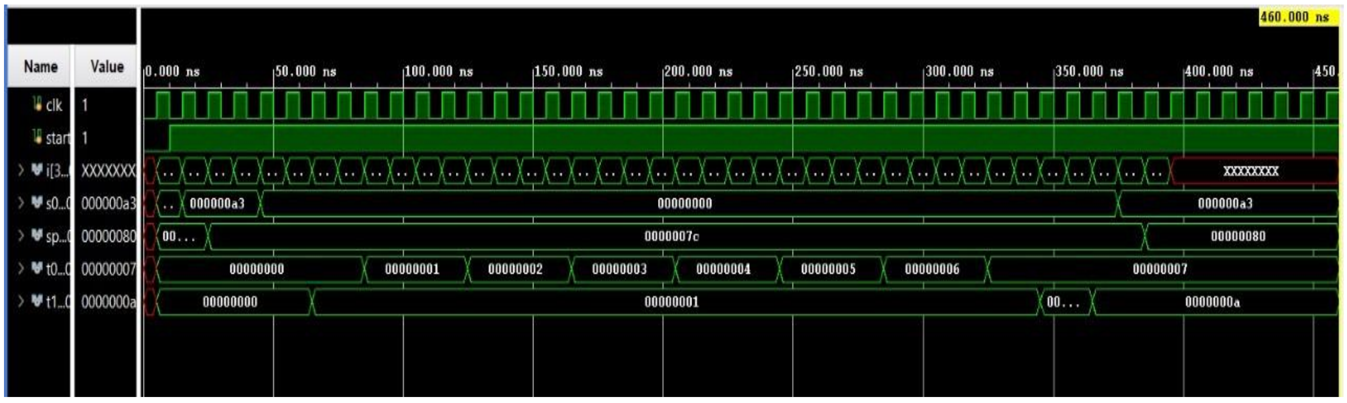
```
module tb_riscv_sc;
//cpu testbench

reg clk;
reg start;

SingleCycleCPU uut(clk, start);

wire [31:0] i;
assign i = uut.instruction;

wire [31:0] s0, sp, t0, t1;

    // Accessing the registers directly from the SingleCycleCPU
    assign s0 = uut.regFile.regs[8];  // s0 is register x8
    assign sp = uut.regFile.regs[2];  // sp is register x2
    assign t0 = uut.regFile.regs[5];  // t0 is register x5
    assign t1 = uut.regFile.regs[6];  // t1 is register x6


initial
        forever #5 clk = ~clk;

initial begin
        clk = 0;
        start = 0;
        #10 start = 1;

        #450 $finish;

end
```

```
endmodule
```

Test Instructions:

```
addi s0, zero, 163
addi sp, sp, -4
sw s0, 0(sp)
add s0, zero, zero
addi t0, zero, 0
LOOP:
  slti t1, t0, 7
  beq t1, zero, EXIT
  addi t0, t0, 1
  beq zero, zero, LOOP
EXIT:
   ori t1, t1, 10
   lw s0, 0(sp)
   addi sp, sp, 4
```

Results:



Here, t0 is incremented from 0 to 7. t1 is zero until ori instruction is implemented, after that t1 updates as 0xa.

2. Tested using the assembly level code using Ripes to get the machine language code.

We are testing the Q1 of Assignment 2 which sorts the array of 10 numbers in ascending order and indicates the smallest and the largest number.

Modified code:

```
addi t0, zero, 10
addi sp, sp, -40
lw a0, 0(sp) //max
lw a1, 0(sp) //min
add a4, sp, zero
loop:
  lw t1 , 0(a4)
  addi a4, a4, 4
  slt s2 , t1, a0
  beq s2, zero, maxup
continue1:
  slt s3 , a1, t1
  beq s3, zero, minup
continue2:
  addi t0, t0, -1
  beq t0, zero, exit
  beq zero, zero, loop
maxup:
  add a0, t1, zero
  beq zero, zero, continue1
minup:
  add a1, t1, zero
  beq zero, zero, continue2

exit:
  addi sp, sp, -4
  sw a1, 0(sp)
  addi sp, sp, -4
  sw a0, 0(sp)
```

Input Data: 11,9,7,6,11,2,15,5,10,12

We were only required to update the DataMemory module , rest all the modules remain same.
We are directly storing the elements in the data memory.

```
// DataMemory:
module DataMemory(
        input rst,
        input clk,
        input memWrite,
        input memRead,
        input [31:0] address,
        input [31:0] writeData,
        output reg [31:0] readData
);


        reg [7:0] data_memory [127:0];
        always @ (posedge clk) begin
                if(rst) begin
                        data_memory[0] <= 8'b0;
                        data_memory[1] <= 8'b0;
                        data_memory[2] <= 8'b0;
                        data_memory[3] <= 8'b0;
                        data_memory[4] <= 8'b0;
                        data_memory[5] <= 8'b0;
                        data_memory[6] <= 8'b0;
                        data_memory[7] <= 8'b0;
                        data_memory[8] <= 8'b0;
                        data_memory[9] <= 8'b0;
                        data_memory[10] <= 8'b0;
                        data_memory[11] <= 8'b0;
                        data_memory[12] <= 8'b0;
                        data_memory[13] <= 8'b0;
                        data_memory[14] <= 8'b0;
                        data_memory[15] <= 8'b0;
                        data_memory[16] <= 8'b0;
                        data_memory[17] <= 8'b0;
                        data_memory[18] <= 8'b0;
                        data_memory[19] <= 8'b0;
                        data_memory[20] <= 8'b0;
                        data_memory[21] <= 8'b0;
                        data_memory[22] <= 8'b0;
                        data_memory[23] <= 8'b0;
```

```
data_memory[24] <= 8'b0;
data_memory[25] <= 8'b0;
data_memory[26] <= 8'b0;
data_memory[27] <= 8'b0;
data_memory[28] <= 8'b0;
data_memory[29] <= 8'b0;
data_memory[30] <= 8'b0;
data_memory[31] <= 8'b0;
data_memory[32] <= 8'b0;
data_memory[33] <= 8'b0;
data_memory[34] <= 8'b0;
data_memory[35] <= 8'b0;
data_memory[36] <= 8'b0;
data_memory[37] <= 8'b0;
data_memory[38] <= 8'b0;
data_memory[39] <= 8'b0;
data_memory[40] <= 8'b0;
data_memory[41] <= 8'b0;
data_memory[42] <= 8'b0;
data_memory[43] <= 8'b0;
data_memory[44] <= 8'b0;
data_memory[45] <= 8'b0;
data_memory[46] <= 8'b0;
data_memory[47] <= 8'b0;
data_memory[48] <= 8'b0;
data_memory[49] <= 8'b0;
data_memory[50] <= 8'b0;
data_memory[51] <= 8'b0;
data_memory[52] <= 8'b0;
data_memory[53] <= 8'b0;
data_memory[54] <= 8'b0;
data_memory[55] <= 8'b0;
data_memory[56] <= 8'b0;
data_memory[57] <= 8'b0;
data_memory[58] <= 8'b0;
data_memory[59] <= 8'b0;
data_memory[60] <= 8'b0;
data_memory[61] <= 8'b0;
data_memory[62] <= 8'b0;
data_memory[63] <= 8'b0;
data_memory[64] <= 8'b0;
data_memory[65] <= 8'b0;
data_memory[66] <= 8'b0;
data_memory[67] <= 8'b0;
```

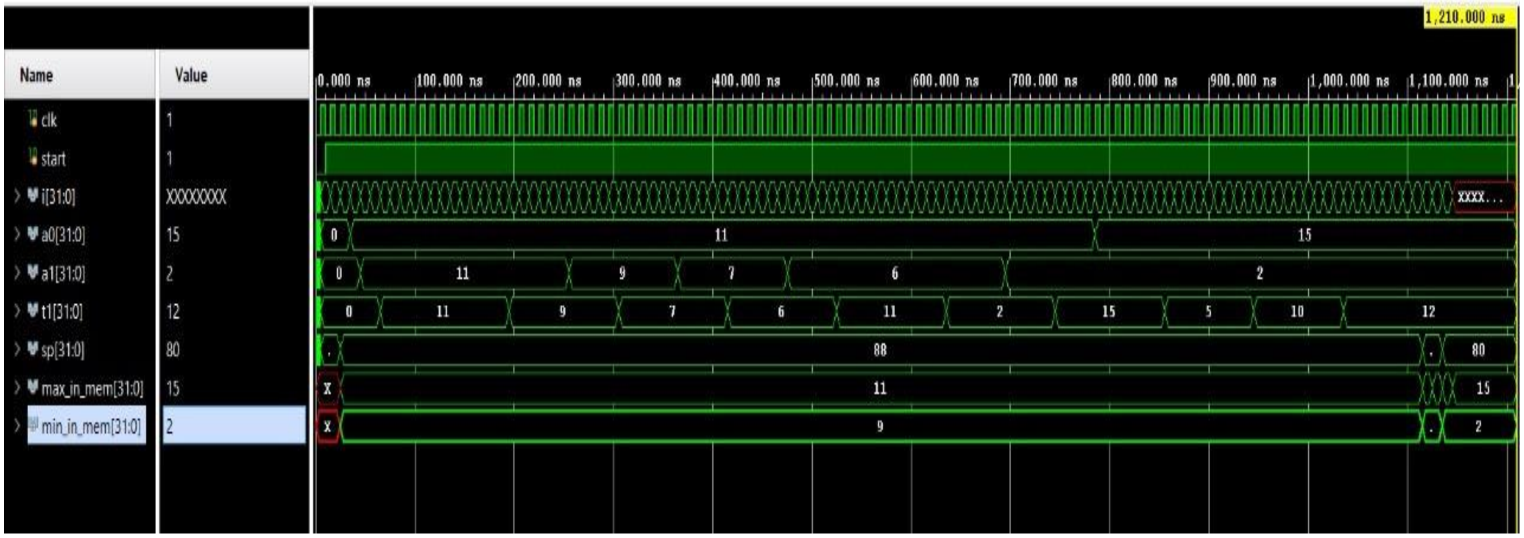
```
data_memory[68] <= 8'b0;
data_memory[69] <= 8'b0;
data_memory[70] <= 8'b0;
data_memory[71] <= 8'b0;
data_memory[72] <= 8'b0;
data_memory[73] <= 8'b0;
data_memory[74] <= 8'b0;
data_memory[75] <= 8'b0;
data_memory[76] <= 8'b0;
data_memory[77] <= 8'b0;
data_memory[78] <= 8'b0;
data_memory[79] <= 8'b0;
data_memory[80] <= 8'b0;
data_memory[81] <= 8'b0;
data_memory[82] <= 8'b0;
data_memory[83] <= 8'b0;
data_memory[84] <= 8'b0;
data_memory[85] <= 8'b0;
data_memory[86] <= 8'b0;
data_memory[87] <= 8'b0;
data_memory[88] <= 11;
data_memory[89] <= 8'b0;
data_memory[90] <= 8'b0;
data_memory[91] <= 8'b0;
data_memory[92] <= 9;
data_memory[93] <= 8'b0;
data_memory[94] <= 8'b0;
data_memory[95] <= 8'b0;
data_memory[96] <= 7;
data_memory[97] <= 8'b0;
data_memory[98] <= 8'b0;
data_memory[99] <= 8'b0;
data_memory[100] <= 6;
data_memory[101] <= 8'b0;
data_memory[102] <= 8'b0;
data_memory[103] <= 8'b0;
data_memory[104] <= 11;
data_memory[105] <= 8'b0;
data_memory[106] <= 8'b0;
data_memory[107] <= 8'b0;
data_memory[108] <= 2;
data_memory[109] <= 8'b0;
data_memory[110] <= 8'b0;
data_memory[111] <= 8'b0;
```

```
                data_memory[112] <= 15;
                data_memory[113] <= 8'b0;
                data_memory[114] <= 8'b0;
                data_memory[115] <= 8'b0;
                data_memory[116] <= 5;
                data_memory[117] <= 8'b0;
                data_memory[118] <= 0;
                data_memory[119] <= 0;
                data_memory[120] <= 10;
                data_memory[121] <= 0;
                data_memory[122] <= 0;
                data_memory[123] <= 0;
                data_memory[124] <= 12;
                data_memory[125] <= 0;
                data_memory[126] <= 0;
                data_memory[127] <= 0;
            end
            else begin
                if(memWrite) begin
                    data_memory[address + 3] <= writeData[31:24];
                    data_memory[address + 2] <= writeData[23:16];
                    data_memory[address + 1] <= writeData[15:8];
                    data_memory[address]     <= writeData[7:0];
                end

                end
        end

        always @(*) begin
            if(memRead) begin
                readData[31:24]  = data_memory[address + 3];
                readData[23:16]  = data_memory[address + 2];
                readData[15:8]   = data_memory[address + 1];
                readData[7:0]    = data_memory[address];
            end
            else begin
                readData         = 32'b0;
            end
        end

endmodule
```

Test Bench File:

```
module tb_riscv_sc;
//cpu testbench

reg clk;
reg start;

SingleCycleCPU uut(clk, start);

wire [31:0] i;
assign i = uut.instruction;

wire [31:0] a0, a1, t1, sp, max_in_mem, min_in_mem;

   // Accessing the registers directly from the SingleCycleCPU
   assign a0 = uut.regFile.regs[10];
   assign a1 = uut.regFile.regs[11];
   assign t1 = uut.regFile.regs[6];
   assign sp = uut.regFile.regs[2];
   assign max_in_mem =
{uut.dataMemory.data_memory[sp+3],uut.dataMemory.data_memory[sp+2],uut.dataMemory.dat
a_memory[sp+1],uut.dataMemory.data_memory[sp]};
   assign min_in_mem =
{uut.dataMemory.data_memory[sp+7],uut.dataMemory.data_memory[sp+6],uut.dataMemory.dat
a_memory[sp+5],uut.dataMemory.data_memory[sp+4]};


initial
        forever #5 clk = ~clk;

initial begin
        clk = 0;
        start = 0;
        #10 start = 1;

        #1200 $finish;

end

endmodule
```

RESULTS:



Here, max_in_mem has the maximum of all elements ie., 15
and min_in_mem has the minimum of all elements ie., 2.