# ELEC0028 Advanced Digital Design Coursework

## Part 1 SystemVerilog Description and Simulation of the RISC-V Microarchitecture

### Task 1.1 Specifying control signals

Table 1 Table of RISC-V control signals

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Inst** | **RegWrite** | **ImmSrc** | **ALUSrc** | **ALUControl** | **MemWrite** | **ResultSrc** | **PCSrc** |
| **add** | **1** | **XXX** | **0** | **X0X10** | **0** | **01** | **00** |
| **sub** | **1** | **XXX** | **0** | **X1X10** | **0** | **01** | **00** |
| **or** | **1** | **XXX** | **0** | **XX111** | **0** | **01** | **00** |
| **and** | **1** | **XXX** | **0** | **XX011** | **0** | **01** | **00** |
| **sll** | **1** | **XXX** | **0** | **0XX00** | **0** | **01** | **00** |
| **srl** | **1** | **XXX** | **0** | **1XX00** | **0** | **01** | **00** |
| **slt** | **1** | **XXX** | **0** | **X1X01** | **0** | **01** | **00** |
| **addi** | **1** | **000** | **1** | **X0X10** | **0** | **01** | **00** |
| **ori** | **1** | **000** | **1** | **XX111** | **0** | **01** | **00** |
| **andi** | **1** | **000** | **1** | **XX011** | **0** | **01** | **00** |
| **slli** | **1** | **000** | **1** | **0XX00** | **0** | **01** | **00** |
| **srli** | **1** | **000** | **1** | **1XX00** | **0** | **01** | **00** |
| **slti** | **1** | **000** | **1** | **X1X01** | **0** | **01** | **00** |
| **lw** | **1** | **000** | **1** | **X0X10** | **0** | **10** | **00** |
| **sw** | **0** | **001** | **1** | **X0X10** | **1** | **XX** | **00** |
| **beq** | **0** | **010** | **0** | **X1X10** | **0** | **XX** | **{0, Zero}** |
| **bne** | **0** | **010** | **0** | **X1X10** | **0** | **XX** | **{0, ~Zero}** |
| **blt** | **0** | **010** | **0** | **X1X10** | **0** | **XX** | **{0, Negative}** |
| **bge** | **0** | **010** | **0** | **X1X10** | **0** | **XX** | **{0, ~Negative}** |
| **jal** | **1** | **100** | **X** | **XXXXX** | **0** | **11** | **01** |
| **jalr** | **1** | **000** | **1** | **X0X10** | **0** | **11** | **10** |
| **lui** | **1** | **011** | **X** | **XXXXX** | **0** | **00** | **00** |

### Task 1.2 Writing SystemVerilog descriptions of the sub-blocks

Program\_counter.sv code

module program\_counter (output logic [31:0] PC, PCPlus4,

                        input logic [31:0] PCTarget, ALUResult,

                        input logic [1:0] PCSrc,

                        input logic Reset, CLK);

// Enter your code here

    assign PCPlus4 = PC + 4;

    // Synchronous PC update with reset

    always\_ff @(posedge CLK or posedge Reset) begin

        if (Reset) begin

            PC <= 32'b0;  // Reset PC to 0 (start executing from 0x00000000)

        end else begin

            case (PCSrc)

                2'b00: PC <= PCPlus4;   // Sequential execution (PC + 4)

                2'b01: PC <= PCTarget;  // Branch target address

                2'b10: PC <= ALUResult; // JALR computed target

                default: PC <= PCPlus4; // Default case: fall back to sequential

            endcase

        end

    end

endmodule

Control\_unit.sv

module control\_unit (output logic [1:0] PCSrc,

                    output logic [1:0] ResultSrc,

                    output logic MemWrite, ALUSrc, RegWrite,

                    output logic [4:0] ALUControl,

                    output logic [2:0] ImmSrc,

                    input logic [31:0] Instr,

                    input logic Zero, Negative);

// Enter your code here

    logic [6:0] opcode;

    logic [2:0] funct3;

    logic [6:0] funct7;

    assign opcode = Instr[6:0];   // Opcode:

    assign funct3 = Instr[14:12]; // Funct3:

    assign funct7 = Instr[31:25]; // Funct7: R

    // \*\*1. Immsrc\*\*

    always\_comb begin

        case (opcode)

            7'b0000011: ImmSrc = 3'b000; // I (LW)

            7'b0010011: ImmSrc = 3'b000; // I (Arithmetic)

            7'b1100111: ImmSrc = 3'b000; // I (JALR)

            7'b0100011: ImmSrc = 3'b001; // S (SW)

            7'b1100011: ImmSrc = 3'b010; // B (Branch)

            7'b0110111: ImmSrc = 3'b011; // U (LUI)

            7'b1101111: ImmSrc = 3'b100; // J (JAL)

            default: ImmSrc = 3'b000;

        endcase

    end

    // \*\*2. PCSrc\*\*

    always\_comb begin

        case (opcode)

            7'b1100011: // (BEQ, BNE, BLT, BGE)

                PCSrc = (funct3 == 3'b000 && Zero)  ? 2'b01 :  // BEQ

                        (funct3 == 3'b001 && !Zero) ? 2'b01 :  // BNE

                        (funct3 == 3'b100 && Negative) ? 2'b01 : // BLT

                        (funct3 == 3'b101 && !Negative) ? 2'b01 : // BGE

                        2'b00;

            7'b1101111: PCSrc = 2'b01; // JAL

            7'b1100111: PCSrc = 2'b10; // JALR

            default: PCSrc = 2'b00;

        endcase

    end

    // \*\*3. ResultSrc\*\*

    always\_comb begin

        case (opcode)

            7'b0000011: ResultSrc = 2'b10; // LW

            7'b1101111: ResultSrc = 2'b11; // JAL

            7'b1100111: ResultSrc = 2'b11; // JALR (PC+4)

            7'b0110111: ResultSrc = 2'b00; // LUI

            7'b0010011,

            7'b0110011: ResultSrc = 2'b01; // I & R

            default: ResultSrc = 2'b00;

        endcase

    end

    // \*\*4. ALUSrc\*\*

    assign ALUSrc = (opcode == 7'b0010011 ||  // I (Arithmetic)

                     opcode == 7'b0000011 ||  // I (LW)

                     opcode == 7'b0100011 ||  // S (SW)

                     opcode == 7'b1100111);   // I (JALR)

    // \*\*5. MemWrite\*\*

    assign MemWrite = (opcode == 7'b0100011); // SW

    // \*\*6. RegWrite\*\*

    assign RegWrite = (opcode == 7'b0110011 ||  // R

                       opcode == 7'b0010011 ||  // I

                       opcode == 7'b0000011 ||  // LW

                       opcode == 7'b1101111 ||  // JAL

                       opcode == 7'b1100111 ||  // JALR

                       opcode == 7'b0110111);   // LUI

    // \*\*7. ALUControl\*\*

    always\_comb begin

        case (opcode)

            7'b0110011: begin // \*\*R

                case ({funct7, funct3})

                    10'b0000000000: ALUControl = 5'b00110; // ADD

                    10'b0100000000: ALUControl = 5'b01010; // SUB

                    10'b0000000110: ALUControl = 5'b00111; // OR

                    10'b0000000111: ALUControl = 5'b00011; // AND

                    10'b0000000001: ALUControl = 5'b00000; // SLL

                    10'b0000000101: ALUControl = 5'b10000; // SRL

                    10'b0000000010: ALUControl = 5'b01001; // SLT

                    default: ALUControl = 5'b00000; //

                endcase

            end

            7'b0010011: begin // \*\*I

                case (funct3)

                    3'b000: ALUControl = 5'b00110; // ADDI

                    3'b110: ALUControl = 5'b00111; // ORI

                    3'b111: ALUControl = 5'b00011; // ANDI

                    3'b001: ALUControl = 5'b00000; // SLLI

                    3'b101: ALUControl = 5'b10000; // SRLI

                    3'b010: ALUControl = 5'b01001; // SLTI

                    default: ALUControl = 5'b00000;

                endcase

            end

            7'b0000011, // \*\*LW\*\*

            7'b0100011, // \*\*SW\*\*

            7'b1100111: // \*\*JALR\*\*

                ALUControl = 5'b00110;

            7'b1100011: begin // \*\*Branch\*\*

                case (funct3)

                    3'b000, // BEQ

                    3'b001, // BNE

                    3'b100, // BLT

                    3'b101: // BGE

                        ALUControl = 5'b01010;

                    default: ALUControl = 5'b00000;

                endcase

            end

            7'b1101111: ALUControl = 5'b00000; // JAL

            7'b0110111: ALUControl = 5'b00000; // LUI

            default: ALUControl = 5'b00000;

        endcase

    end

endmodule

Extend.sv code:

module extend (output logic [31:0] ImmExt,

                input logic [31:0] Instr,

                input logic [2:0] ImmSrc);

// Enter your code here

    always\_comb begin

        case (ImmSrc)

            3'b000: // I-type (e.g., LW, JALR, arithmetic immediates)

                ImmExt = {{20{Instr[31]}}, Instr[31:20]};

            3'b001: // S-type (e.g., SW)

                ImmExt = {{20{Instr[31]}}, Instr[31:25], Instr[11:7]};

            3'b010: // B-type (e.g., BEQ, BNE, BLT, BGE)

                ImmExt = {{19{Instr[31]}}, Instr[31], Instr[7], Instr[30:25], Instr[11:8], 1'b0};

            3'b011: // U-type (e.g., LUI, AUIPC)

                ImmExt = {Instr[31:12], 12'b0};

            3'b100: // J-type (e.g., JAL)

                ImmExt = {{11{Instr[31]}}, Instr[31], Instr[19:12], Instr[20], Instr[30:21], 1'b0};

            default:

                ImmExt = 32'b0;

        endcase

    end

endmodule

alu.sv code

module alu (output logic signed [31:0] ALUResult,

            output logic Zero, Negative,

            input logic signed [31:0] SrcA, SrcB,

            input logic [4:0] ALUControl);

// Enter your code here

    always\_comb begin

        case (ALUControl)

            5'b10110, 5'b10010, 5'b00010, 5'b00110: ALUResult = SrcA + SrcB;

            5'b11110, 5'b11010, 5'b01010, 5'b01110: ALUResult = SrcA - SrcB;

            5'b00111, 5'b01111, 5'b10111, 5'b11111: ALUResult = SrcA | SrcB;

            5'b00011, 5'b01011, 5'b10011, 5'b11011: ALUResult = SrcA & SrcB;

            5'b00000, 5'b00100, 5'b01000, 5'b01100: ALUResult = SrcA << SrcB[4:0];

            5'b10000, 5'b10100, 5'b11000, 5'b11100: ALUResult = SrcA >> SrcB[4:0];

            5'b11001, 5'b11101, 5'b01101, 5'b01001: ALUResult = (SrcA < SrcB) ? 32'b1 : 32'b0;

        endcase

        Zero = (ALUResult == 32'b0);

        Negative = ALUResult[31];

    end

endmodule

always @(\*) begin is used first for alu module, after reinstalling for the latest version of Verilog, always\_comb begin works as well.

Their corresponding testbench code is added in the appendix.

### Task 1.3 Simulations of sub-blocks using testbenches

Program\_counter\_tb text results:

A screen shot of a computer

AI-generated content may be incorrect.

Program\_counter\_tb graphical results:

A screenshot of a computer

AI-generated content may be incorrect.

The PC will be written to 0x00000000 each time when the Reset is signal is pulled high. When the CLK signal is positive edged and the Reset is 0, the PC will be written depending on the value of PCSrc, which is followed by the microarchitecture provided: 00 for PCPlus4, 01 for PCTarget and 10 for ALUResult. This works well.

Control\_unit\_tb text results:

A screen shot of a computer

AI-generated content may be incorrect.

Control\_unit\_tb graphical results:

A screenshot of a computer

AI-generated content may be incorrect.

This control unit will be input a Instr from the instruction\_memory file, which converts the machine codes in the txt file into a series of instructions. It will divide the Instr into several parts followed by the Instruction format and subset provided. Combined with the control signal I have written in the first task, this unit is basically linking all the control signals with the Instr, such as the opcode will be used to determine the instruction type and generate the ImmSrc. The control\_unit\_tb result is the same as the control signal table. This works well.

Alu\_tb text results:

A computer screen with many lines and numbers

AI-generated content may be incorrect.

Alu\_tb graphical results:

A screenshot of a computer

AI-generated content may be incorrect.

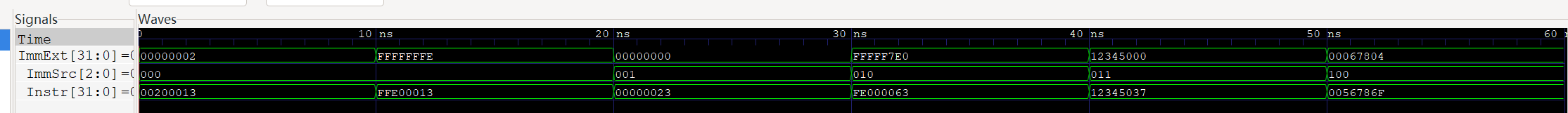
This alu module is using the ALUControl signal generated from the control unit to achieve the actual function, such as add is adding the values of two registers and put this into the ALUResult signal. The arithmetic operation is two’s complement as required. Though the negative is pulled high even for the OR operation, but it will not cause problem as we set the condition for the branch for both Negative and Func3. Therefore, this module works well.

Expand\_tb text results:

A computer screen with white text

AI-generated content may be incorrect.

Expand\_tb graphical results:



This module simply extracts the ImmExt based on the Instr and Immsrc as the same as the Table 3 provided. It works well.

### Task 1.4 Completing top-level RISC-V SystemVerilog description

Risc\_v. sv code:

`include "instruction\_memory.sv"

`include "reg\_file.sv"

`include "extend.sv"

`include "alu.sv"

`include "data\_memory\_and\_io.sv"

`include "program\_counter.sv"

`include "control\_unit.sv"

module risc\_v(output logic [31:0] CPUOut,

                input logic [31:0] CPUIn,

                input logic Reset, CLK);

logic [31:0] Instr, WD3, RD1, RD2, SrcA, SrcB, ALUResult, WD, RD, Result, ImmExt, PCTarget, PCNext, PC, PCPlus4;

logic [4:0]  A1, A2, A3;

logic        MemWrite, ALUSrc, RegWrite;

logic        Zero, Negative;

logic [4:0]  ALUControl;

logic [2:0]  ImmSrc;

logic [1:0]  ResultSrc;

logic [1:0]  PCSrc;

// Enter your code here

assign PCTarget = PC + ImmExt;

program\_counter program\_counter\_inst (

    .PC(PC),

    .PCPlus4(PCPlus4),

    .PCTarget(PCTarget),

    .ALUResult(ALUResult),

    .PCSrc(PCSrc),

    .Reset(Reset),

    .CLK(CLK)

);

instruction\_memory instruction\_memory\_inst (

    .PC(PC),

    .Instr(Instr)

);

control\_unit control\_unit\_inst (

    .PCSrc(PCSrc),

    .ResultSrc(ResultSrc),

    .MemWrite(MemWrite),

    .ALUSrc(ALUSrc),

    .RegWrite(RegWrite),

    .ALUControl(ALUControl),

    .ImmSrc(ImmSrc),

    .Instr(Instr),

    .Zero(Zero),

    .Negative(Negative)

);

assign A1 = Instr[19:15]; // rs1

assign A2 = Instr[24:20]; // rs2

assign A3 = Instr[11:7];  // rd

reg\_file reg\_file\_inst (

    .RD1(RD1),

    .RD2(RD2),

    .WD3(WD3),

    .A1(A1),

    .A2(A2),

    .A3(A3),

    .WE3(RegWrite),

    .CLK(CLK)

);

extend extend\_inst (

    .Instr(Instr),

    .ImmSrc(ImmSrc),

    .ImmExt(ImmExt)

);

assign SrcA = RD1;

assign SrcB = (ALUSrc) ? ImmExt : RD2; // Select SrcB based on ALUSrc control

alu alu\_unit (

    .ALUResult(ALUResult),

    .Zero(Zero),

    .Negative(Negative),

    .SrcA(SrcA),

    .SrcB(SrcB),

    .ALUControl(ALUControl)

);

data\_memory\_and\_io data\_mem (

    .RD(RD),

    .CPUOut(CPUOut),

    .A(ALUResult),

    .WD(RD2),

    .CPUIn(CPUIn),

    .WE(MemWrite),

    .CLK(CLK)

);

always\_comb begin

    case (ResultSrc)

        2'b00: Result = ImmExt;

        2'b01: Result = ALUResult;

        2'b10: Result = RD;

        2'b11: Result = PCPlus4;

        default: Result = 32'b0;

    endcase

end

assign WD3 = Result;

endmodule

Risc\_v\_tb.sv code:

`timescale 1ns/1ps

`include "risc\_v.sv"

module risc\_v\_tb;

logic [31:0] CPUOut, CPUIn;

logic Reset, CLK;

risc\_v dut(CPUOut, CPUIn, Reset, CLK);

initial begin // Generate clock signal with 20 ns period

CLK = 0;

forever #10 CLK = ~CLK;

end

initial begin // Apply stimulus

// Enter your code here

$dumpfile("risc\_v\_tb.vcd");

$dumpvars(0, risc\_v\_tb);

Reset = 1;

CPUIn = 32'b0;

// Step 2: Release reset

#20 Reset = 0;

$display("CPU Reset Done");

// Step 3: Run CPU for several clock cycles

repeat (50) begin

    #20;

    CPUIn = 0;

end

// Step 4: End simulation

$display("Simulation complete.");

$finish; // This system tasks ends the simulation

end

always @ (negedge CLK)

$display ("t = %3d, CPUIn = %d, CPUOut = %d, Reset = %b, PCSrc = %b PC = %d, PCTarget = %h, ImmExt = %h, Instr = %h, ALUResult = %d", $time, CPUIn, CPUOut, Reset, dut.PCSrc, dut.PC, dut.PCTarget, dut.ImmExt, dut.Instr, dut.ALUResult);

endmodule

Risc\_v\_tb results:

A screen shot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The machine code is input from the program.txt file, after disassembling the machine code into assembly code, we know that:

addi x1, x0, 50 Load the value 50 into register x1

lui x10, 0x80000 Load upper immediate to x10 (sets x10 to 0x80000000)

lw x2, -4(x10) Load a word from memory at address (x10 - 4) = 0x7FFFFFFC into x2

sub x3, x1, x2 Subtract x2 from x1 and store the result in x3

sw x3, -4(x10) Store x3 back into memory at address (x10 - 4) = 0x7FFFFFFC

beq x0, x0, 0 Infinite loop (branch to itself)

In the risc\_v\_tb file, the CPUIn is set to 0, therefore x2 is always 0. From the data\_memory\_and\_io file, it is clear that 0x7FFFFFFC is a specific address instead of normal memory address, it is related to CPUOut. Hence, sw x3, -4(x10), will store the value of x3 register to CPUOut in the following positive clock edge. The result from the testbench code is aligned with the assembly code.

### Task 1.5 Running a test program

A screenshot of a computer

AI-generated content may be incorrect.

Risc\_v\_tb simulation result:

A screen shot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

This program test is basically test each control signals along with the ALUResult respectively. All of the simulated output are performed as expected with the comparison of RISCV emulator. Though, there is a few points might cause confusion: firstly, immediates like 0x800 in our simulation is different with the emulator as the instruction extend table provided for operating addi function is different. The ImmExt extracted becomes 0xFFFFF800 instead of 0x00000800. Besides, though the ALUResult in the control module is defined as signed, it is defined as unsigned in the risc\_v module, therefore the output of the ALUResult is unsigned.

Apart from the above two harmless difference, other functions performs very well. The PC increments in the same way as the emulator, increasing by 4 after each instruction. Arithmetic operations are correct, such as add x3, x1, x2 → ALUResult = 4. Jump and branch instructions correctly modify PCTarget in the simulation.

At final state, the result from RISC-V emulator is that the code would jump to finish at line 40 (machine code: 0100006f), which is exactly aligned with out simulation result no ALU. Similar to the last task, it is clear that at line 38, the assembly code is addi x3, zero, 0x789, which stores 0x789 to x3. Then the line 39 sw x3, 0x0(x31), it again stores the value of x3 to CPUOut, we can see that in our simulation result, the CPUOut is written 1929 in decimal in the next positive clock edge.

Our risc\_v code perfectly performs the arithmetic operation and stores the value into the exact register position as required. CPUOur perfectly respond to the sw function. The emulator shows the same function as our designed risc\_v program.

## Part 2 RISC-V Assembly Programming

### Task 2.1 A program to count the number of 1’s in an input byte

Assembly code:

# Initialize CPUIn and CPUOut

Lui x31, 0x80000

addi x31, x31, -4 # CPUOut and CPUIn

# Initialize parameters: a, sum, i

lw x5, 0(x31) # x5 = a = CPUIn

addi x6, x0, 0 # x6 = sum = 0

addi x7, x0, 0 # x7 = i = 0

addi x9, x0, 1 # Load constant 1

addi x10, x0, 8 # Load constant 8 (loop limit)

# Start

branch1:

and x8, x5, x9 # x8 used as temporary mask, check if LSB is 1

beq x8, x0, branch2

add x6, x6, x9 # sum += 1

branch2:

srli x5, x5, 1 # a = a >> 1 (shift right)

add x7, x7, x9 # i += 1

bne x7, x10, branch1

# Stop

sw x6, 0x0(x31) # CPUOut = sum

After inputting the assembly code into the emulator by setting the CPUIn to 0x000001F, the CPUOut is 5 in decimal as expected. As the register 6 stores the value of CPUOut, the assembly code works well.

RiscV emulator result:

A screenshot of a computer program

AI-generated content may be incorrect.

Then, we convert the assembly code into machine code for further simulation. It should be noticed that instead of using branch1 in the assembly code, it should be counted as an immediate number (steps \* 4).

Machine code:

80000fb7

ffcf8f93

000fa283

00000313

00000393

00100493

00800513

0092F433

00040463

00930333

0012d293

009383b3

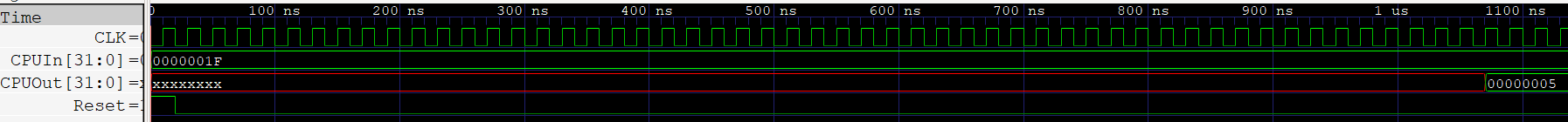
fe7516e3

006fa023

Here we save the machine code into a new text file and simulate the result.

A close up of a screen

AI-generated content may be incorrect.



It is clear that the simulation aligned with the expected output. The value of CPUIn is written from the testbench code into the register file, counts the number of 1’s in the least significant byte (CPUIn[7:0]), and outputs this value via CPUOut after fixed 8 repeated cycles.

Task 2.1 Testbench code:

`timescale 1ns/1ps

`include "risc\_v.sv"

module risc\_v\_tb;

logic [31:0] CPUOut, CPUIn;

logic Reset, CLK;

risc\_v dut(CPUOut, CPUIn, Reset, CLK);

initial begin // Generate clock signal with 20 ns period

CLK = 0;

forever #10 CLK = ~CLK;

end

initial begin // Apply stimulus

// Enter your code here

$dumpfile("risc\_v\_tb.vcd");

$dumpvars(0, risc\_v\_tb);

Reset = 1;

CPUIn = 32'h0000001F;

// Step 2: Release reset

#20 Reset = 0;

$display("CPU Reset Done");

// Step 3: Run CPU for several clock cycles

repeat (70) begin

    #20;

end

// Step 4: End simulation

$display("Simulation complete.");

$finish; // This system tasks ends the simulation

end

always @ (negedge CLK)

$display ("t = %3d, CPUIn = %b, CPUOut = %d, Reset = %b, PCSrc = %b PC = %d, PCTarget = %h, ImmExt = %h, Instr = %h, ALUResult = %d", $time, CPUIn, CPUOut, Reset, dut.PCSrc, dut.PC, dut.PCTarget, dut.ImmExt, dut.Instr, dut.ALUResult);

endmodule

### Task 2.2 A program to multiply two bytes using the shift and add method

Assmbly code:

# Initialize CPUIn and CPUOut

lui x31, 0x80000 # Load upper immediate (Base Address)

addi x31, x31, -4 # Adjust CPUOut and CPUIn memory location

# Load values from memory

lw x8, 0x0(x31) # Load A from CPUIn -> x8 (Multiplicand)

lw x9, 0x0(x31) # Load B from CPUIn -> x9 (Multiplier)

# Initialize variables

addi x10, x0, 0 # x10 = P (Product), initially 0

addi x11, x0, 0 # x11 = i (Loop Counter)

addi x12, x0, 1 # x12 = constant 1 (for bit checking)

addi x13, x0, 8 # x13 = constant 8 (loop limit)

# Start of multiplication loop

mul\_loop:

and x14, x9, x12 # Check if LSB of B (x9) is 1

beq x14, x0, skip\_add # If B[0] == 0, skip addition

add x10, x10, x8 # P = P + A

skip\_add:

slli x8, x8, 1 # A = A << 1 (Shift Left)

srli x9, x9, 1 # B = B >> 1 (Shift Right)

addi x11, x11, 1 # i += 1

bne x11, x13, mul\_loop # If i < 8, repeat loop

# Store result

sw x10, 0x0(x31) # Store result in CPUOut

Emultor result (3\*9=27)

A screenshot of a computer

AI-generated content may be incorrect.

machine code:

80000fb7

ffcf8f93

000fa403

000fa483

00000513

00000593

00100613

00800693

00c4f733

00070463

00850533

00141413

0014d493

00158593

fed594e3

00afa023

Riscv\_tb simulation result:

A screen shot of a computer screen

AI-generated content may be incorrect.

A screen shot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

It is clear that the multiplication result is correct for 3\*4=12. The s0 and s1 is input via the CPUIn, which is written in the testbench code.

Task 2.2 Testbench code:

`timescale 1ns/1ps

`include "risc\_v.sv"

module risc\_v\_tb;

logic [31:0] CPUOut, CPUIn;

logic Reset, CLK;

risc\_v dut(CPUOut, CPUIn, Reset, CLK);

initial begin // Generate clock signal with 20 ns period

CLK = 0;

forever #10 CLK = ~CLK;

end

initial begin // Apply stimulus

// Enter your code here

$dumpfile("risc\_v\_tb.vcd");

$dumpvars(0, risc\_v\_tb);

Reset = 1;

CPUIn = 32'b0;

// Step 2: Release reset

#20 Reset = 0;

$display("CPU Reset Done");

#20 CPUIn = 32'b0;

#20 CPUIn = 32'h00000003;

#20 CPUIn = 32'h00000004;

// Step 3: Run CPU for several clock cycles

repeat (200) begin

    #20;

    CPUIn = 0;

end

// Step 4: End simulation

$display("Simulation complete.");

$finish; // This system tasks ends the simulation

end

always @ (negedge CLK)

$display ("t = %3d, CPUIn = %d, CPUOut = %d, Reset = %b, PCSrc = %b PC = %d, PCTarget = %h, ImmExt = %h, Instr = %h, ALUResult = %d", $time, CPUIn, CPUOut, Reset, dut.PCSrc, dut.PC, dut.PCTarget, dut.ImmExt, dut.Instr, dut.ALUResult);

endmodule

## Appendix for testbench code

### Alu\_tb

`timescale 1ns/1ps

`include "alu.sv"

module alu\_tb;

    // \*\*Inputs\*\*

    logic signed [31:0] SrcA, SrcB;

    logic [4:0] ALUControl;

    // \*\*Outputs\*\*

    logic signed [31:0] ALUResult;

    logic Zero, Negative;

    // \*\*Instantiate ALU module\*\*

    alu dut (

        .ALUResult(ALUResult),

        .Zero(Zero),

        .Negative(Negative),

        .SrcA(SrcA),

        .SrcB(SrcB),

        .ALUControl(ALUControl)

    );

    // \*\*Test sequence\*\*

    initial begin

        $dumpfile("alu\_tb.vcd");

        $dumpvars(0, alu\_tb);

        // \*\*Test ADD\*\*

        SrcA = 32'd10;

        SrcB = 32'd5;

        ALUControl = 5'b00010;

        #10;

        $display("ADD | SrcA = %0d, SrcB = %0d | ALUResult = %0d", SrcA, SrcB, ALUResult);

        // \*\*Test SUB\*\*

        SrcA = 32'd20;

        SrcB = 32'd5;

        ALUControl = 5'b11110;

        #10;

        $display("SUB | SrcA = %0d, SrcB = %0d | ALUResult = %0d", SrcA, SrcB, ALUResult);

        // \*\*Test OR\*\*

        SrcA = 32'h0F0F0F0F;

        SrcB = 32'hF0F0F0F0;

        ALUControl = 5'b00111;

        #10;

        $display("OR  | SrcA = %h, SrcB = %h | ALUResult = %h", SrcA, SrcB, ALUResult);

        // \*\*Test AND\*\*

        SrcA = 32'h0F0F0F0F;

        SrcB = 32'hF0F0F0F0;

        ALUControl = 5'b00011;

        #10;

        $display("AND | SrcA = %h, SrcB = %h | ALUResult = %h", SrcA, SrcB, ALUResult);

       // \*\*Test Logical Left Shift (SLL)\*\*

        SrcA = 32'h00000002;

        SrcB = 32'h00000002; // shift by 2

        ALUControl = 5'b00000;

        #10;

        $display("SLL | SrcA = %0d, SrcB = %0d | ALUResult = %0d (Expected: 8)", SrcA, SrcB, ALUResult);

        // \*\*Test Logical Right Shift (SRL)\*\*

        SrcA = 32'h00000080;

        SrcB = 32'h00000003; // shift by 3

        ALUControl = 5'b10000;

        #10;

        $display("SRL | SrcA = %0d, SrcB = %0d | ALUResult = %0d (Expected: 16)", SrcA, SrcB, ALUResult);

        // \*\*Test Zero flag\*\*

        SrcA = 32'd10;

        SrcB = 32'd10;

        ALUControl = 5'b11110; // 10 - 10 = 0

        #10;

        $display("ZERO TEST | SrcA = %0d, SrcB = %0d | ALUResult = %0d | Zero = %b", SrcA, SrcB, ALUResult, Zero);

        // \*\*Test Negative flag\*\*

        SrcA = 32'd5;

        SrcB = 32'd10;

        ALUControl = 5'b11110; // 5 - 10 = -5

        #10;

        $display("NEGATIVE TEST | SrcA = %0d, SrcB = %0d | ALUResult = %0d | Negative = %b", SrcA, SrcB, ALUResult, Negative);

        // \*\*End simulation\*\*

        #50;

        $finish;

    end

    // \*\*Monitor ALU computations\*\*

    initial begin

        $monitor("t = %3d | ALUControl = %b | SrcA = %0d (%h) | SrcB = %0d (%h) | ALUResult = %0d (%h) | Zero = %b | Negative = %b",

                  $time, ALUControl, SrcA, SrcA, SrcB, SrcB, ALUResult, ALUResult, Zero, Negative);

    end

endmodule

### Control\_unit\_tb

`timescale 1ns/1ps

`include "control\_unit.sv"

module control\_unit\_tb;

  // Inputs

  logic [31:0] Instr;

  logic Zero, Negative;

  // Outputs

  logic [1:0] PCSrc, ResultSrc;

  logic MemWrite, ALUSrc, RegWrite;

  logic [4:0] ALUControl;

  logic [2:0] ImmSrc;

  // Instantiate the DUT (Device Under Test)

  control\_unit dut (

    .PCSrc(PCSrc),

    .ResultSrc(ResultSrc),

    .MemWrite(MemWrite),

    .ALUSrc(ALUSrc),

    .RegWrite(RegWrite),

    .ALUControl(ALUControl),

    .ImmSrc(ImmSrc),

    .Instr(Instr),

    .Zero(Zero),

    .Negative(Negative)

  );

  // Test sequence

  initial begin

    $dumpfile("control\_unit\_tb.vcd");

    $dumpvars(0, control\_unit\_tb);

    // Test Case 1: R-type (ADD x5, x6, x7) - opcode: 0110011, funct3: 000, funct7: 0000000

    Instr = 32'b0000000\_00110\_00111\_000\_00101\_0110011;

    Zero = 0; Negative = 0; #10;

    $display("R-type (ADD) | PCSrc = %b, ALUControl = %b, RegWrite = %b", PCSrc, ALUControl, RegWrite);

    // Test Case 2: I-type (ADDI x5, x6, 10) - opcode: 0010011, funct3: 000

    Instr = 32'b000000000101\_00110\_000\_00101\_0010011;

    Zero = 0; Negative = 0; #10;

    $display("I-type (ADDI) | ALUSrc = %b, ALUControl = %b, RegWrite = %b", ALUSrc, ALUControl, RegWrite);

    // Test Case 3: Load Word (LW x5, 0(x6)) - opcode: 0000011, funct3: 010

    Instr = 32'b000000000000\_00110\_010\_00101\_0000011;

    Zero = 0; Negative = 0; #10;

    $display("LW | ResultSrc = %b, MemWrite = %b, RegWrite = %b", ResultSrc, MemWrite, RegWrite);

    // Test Case 4: Store Word (SW x5, 0(x6)) - opcode: 0100011, funct3: 010

    Instr = 32'b0000000\_00101\_00110\_010\_00000\_0100011;

    Zero = 0; Negative = 0; #10;

    $display("SW | MemWrite = %b, ALUSrc = %b", MemWrite, ALUSrc);

    // Test Case 5: Branch if Equal (BEQ x5, x6, offset) - opcode: 1100011, funct3: 000

    Instr = 32'b0000000\_00101\_00110\_000\_00000\_1100011;

    Zero = 1; Negative = 0; #10;

    $display("BEQ | PCSrc = %b", PCSrc);

    // Test Case 6: Branch if Less Than (BLT x5, x6, offset) - opcode: 1100011, funct3: 100

    Instr = 32'b0000000\_00101\_00110\_100\_00000\_1100011;

    Zero = 0; Negative = 1; #10;

    $display("BLT | PCSrc = %b", PCSrc);

    // Test Case 7: JAL (Jump and Link) - opcode: 1101111

    Instr = 32'b00000000000000000000\_00000\_1101111;

    Zero = 0; Negative = 0; #10;

    $display("JAL | PCSrc = %b, RegWrite = %b", PCSrc, RegWrite);

    // Test Case 8: JALR (Jump and Link Register) - opcode: 1100111

    Instr = 32'b000000000000\_00001\_000\_00000\_1100111;

    Zero = 0; Negative = 0; #10;

    $display("JALR | PCSrc = %b, RegWrite = %b", PCSrc, RegWrite);

    // Test Case 9: LUI (Load Upper Immediate) - opcode: 0110111

    Instr = 32'b00000000000000000000\_00000\_0110111;

    Zero = 0; Negative = 0; #10;

    $display("LUI | ResultSrc = %b, RegWrite = %b", ResultSrc, RegWrite);

    // End test

    #50;

    $finish;

  end

  // Monitor values

  initial begin

    $monitor("t = %3d | Instr = %b | PCSrc = %b | ResultSrc = %b | ALUSrc = %b | ALUControl = %b | RegWrite = %b | MemWrite = %b | ImmSrc = %b",

              $time, Instr, PCSrc, ResultSrc, ALUSrc, ALUControl, RegWrite, MemWrite, ImmSrc);

  end

endmodule

### Extend\_tb

`timescale 1ns/1ps

`include "extend.sv"

module extend\_tb;

    // \*\*Inputs\*\*

    logic [31:0] Instr;

    logic [2:0] ImmSrc;

    // \*\*Outputs\*\*

    logic [31:0] ImmExt;

    // \*\*Instantiate Extend module\*\*

    extend dut (

        .ImmExt(ImmExt),

        .Instr(Instr),

        .ImmSrc(ImmSrc)

    );

    // \*\*Test sequence\*\*

    initial begin

        $dumpfile("extend\_tb.vcd");

        $dumpvars(0, extend\_tb);

        // \*\*Test I-type\*\*

        Instr = 32'h00200013;

        ImmSrc = 3'b000;

        #10;

        $display("I-type | Instr = %h | ImmExt = %h | Expected: 00000002", Instr, ImmExt);

        // \*\*Test I-type Negative\*\*

        Instr = 32'hFFE00013;

        ImmSrc = 3'b000;

        #10;

        $display("I-type (Negative) | Instr = %h | ImmExt = %h | Expected: FFFFFFFE", Instr, ImmExt);

        // \*\*Test S-type\*\*

        Instr = 32'h00000023;

        ImmSrc = 3'b001;

        #10;

        $display("S-type | Instr = %h | ImmExt = %h | Expected: 00000000", Instr, ImmExt);

        // \*\*Test B-type\*\*

        Instr = 32'hFE000063;

        ImmSrc = 3'b010;

        #10;

        $display("B-type | Instr = %h | ImmExt = %h | Expected: FFFFF7E0", Instr, ImmExt);

        // \*\*Test U-type\*\*

        Instr = 32'h12345037;

        ImmSrc = 3'b011;

        #10;

        $display("U-type | Instr = %h | ImmExt = %h | Expected: 12345000", Instr, ImmExt);

        // \*\*Test J-type\*\*

        Instr = 32'h0056786F;

        ImmSrc = 3'b100;

        #10;

        $display("J-type | Instr = %h | ImmExt = %h | Expected: 00067804", Instr, ImmExt);

        // \*\*End simulation\*\*

        $finish;

    end

    // \*\*Monitor all test cases\*\*

    initial begin

        $monitor("t = %3d | ImmSrc = %b | Instr = %b | ImmExt = %b",

                  $time, ImmSrc, Instr, ImmExt);

    end

endmodule

### Program\_counter\_tb

`timescale 1ns/1ps

`include "program\_counter.sv"

module program\_counter\_tb;

  // Inputs

  logic CLK;

  logic Reset;

  logic [1:0] PCSrc;

  logic [31:0] PCTarget, ALUResult;

  // Outputs

  logic [31:0] PC, PCPlus4;

  // Instantiate the `program\_counter` module

  program\_counter dut (

    .PC(PC),

    .PCPlus4(PCPlus4),

    .PCTarget(PCTarget),

    .ALUResult(ALUResult),

    .PCSrc(PCSrc),

    .Reset(Reset),

    .CLK(CLK)

  );

  // Clock generation (Toggle every 10ns, 20ns period)

  initial begin

    CLK = 0;

    forever #10 CLK = ~CLK;

  end

  // Stimulus

  initial begin

    $dumpfile("program\_counter\_tb.vcd");

    $dumpvars(0, program\_counter\_tb);

    // Initialize inputs

    Reset = 1; PCSrc = 2'b00;

    PCTarget = 32'h00000020;  // Example branch target address

    ALUResult = 32'h00000040; // Example JALR computed address

    // Apply Reset

    #20 Reset = 0;

    // Test vector 1: Sequential Execution (PC + 4)

    PCSrc = 2'b00; #20;

    $display("PC = %h, Expected = 4 | PCPlus4 = %h, Expected = 8", PC, PCPlus4);

    // Test vector 2: Branch Target (PCTarget)

    PCSrc = 2'b01; #20;

    $display("PC = %h, Expected = %h | PCPlus4 = %h (unchanged)", PC, PCTarget, PCPlus4);

    // Test vector 3: JALR (ALUResult)

    PCSrc = 2'b10; #20;

    $display("PC = %h, Expected = %h | PCPlus4 = %h (unchanged)", PC, ALUResult, PCPlus4);

    // Test vector 4: Reset

    Reset = 1; #20;

    Reset = 0;

    $display("PC = %h, Expected = 0 | PCPlus4 = 4", PC, PCPlus4);

    // Additional test cases

    #20 PCSrc = 2'b00; #20;

    #20 PCSrc = 2'b01; PCTarget = 32'h00000050; #20;

    #20 PCSrc = 2'b10; ALUResult = 32'h00000080; #20;

    #20 PCSrc = 2'b00; #20;

    // Allow some additional time for simulation

    #100;

    $finish;

  end

  // Monitor PC and other signals

  initial begin

    $monitor("t = %3d, CLK = %b, Reset = %b, PCSrc = %b, PCTarget = %h, ALUResult = %h, PC = %h, PCPlus4 = %h",

              $time, CLK, Reset, PCSrc, PCTarget, ALUResult, PC, PCPlus4);

  end

endmodule