**COAL Lab # 05**

**Designing Microarchitecture-I**

**------------------------------------------------------**

**Name:** Saad Nisar Butt

**Reg. no:** cs211246

**Class:** BSCS 3C1

**------------------------------------------------------**

**Lab Task:**

**Literature Review:**

**State Elements**

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Here we are designing microarchitecture that shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.

**Program Counter**

The program counter is an ordinary 32-bit register. Its output, PC, points to the current instruction. Its input, PCNext, indicates the address of the next instruction.

**Instruction Memory**

The instruction memory has a single read port. It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.

**Register File**

The 32-element × 32-bit register file has two read ports and one write port. The read ports take 5-bit address inputs, A1 and A2, each specifying one of 2 = 32 registers as source 5 operands. They read the 32-bit register values onto read data outputs RD1 and RD2, respectively. The write port takes a 5-bit address input, A3; a 32-bit write data input, WD; a write enable input, WE3; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

**Data Memory**

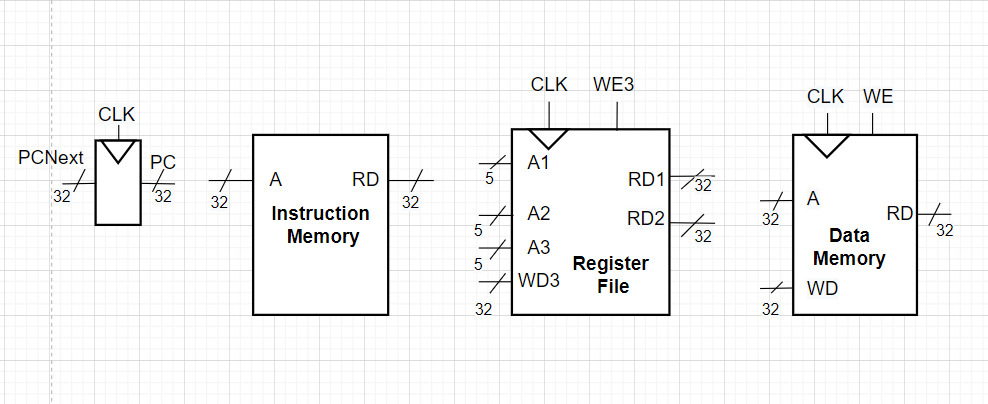
The data memory has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

**Task:**

1. Write Verilog code for each of the following state elements. Attach the code.

* Register File
* Data Memory
* Instruction Memory
* Program Counter

**Block Diagram:**

****

**Verilog Code:**

design.v (Program Counter)

module Program\_Counter(PCnext, clk, reset, PC);

    input [31:0] PCnext

    input clk, reset;

    output reg [31:0] PC;

    always @(posedge clk) begin

        if (reset == 1'b1) begin

            PC <= 32'h00000000;

        end

        else begin

            PC <= PCnext;

    end

endmodule

design.v (Instruction Memory)

module Instruction\_Memory(reset, A, RD);

    input reset;

    input [31:0] A;

    output [31:0] RD;

    reg [31:0] mem [1023:0]; // 8bit = byte, 16bit = half-word, 32bit = word

    assign RD = (reset == 1'b1) ? 32'h00000000 : mem[A[31:2]];

    initial begin

      mem[0] <= 32'h006248B3;

      mem[1] <= 32'h006248B3;

    end

endmodule

design.v (Register File)

module Register\_File(A1, A2, A3, WD3, clk, reset, WE3, RD1, RD2);

    input [4:0] A1, A2, A3; // A1->rs1 | A2->rs2 | A3->rd

    input clk, reset, WE3; // WE3 -> a key signal to let write or not

    input [31:0] WD3;

    output [3:0] RD1, RD2;

    reg [31:0] register[31:0];

    assign RD1 = (reset == 1'b1) ? 32'd0 : register[A1];

    assign RD2 = (reset == 1'b1) ? 32'd0 : register[A2];

    always @(negedge clk) begin

        if((WE3 == 1'b1) & (A3 != 5'h00)) begin

            register[A3] <= WD3;

        end

    end

endmodule

design.v (Data Memory)

module Data\_Memory(A, WD, clk, WE, RD);

    input [31:0] A, WD;

    input clk, WE;

    output [31:0] RD;

    reg [31:0] memory [1023:0];

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

    always @(posedge clk) begin

        if (WE == 1'b1) begin

            memory[A] <= WD;

        end

    end

endmodule

**---------------------------------------------**