COAL Lab # 07

Designing Microarchitecture-II

Name: Muhammad Usman

Class: BSCS 3C1

Reg No: cs211208

Lab Task:

Literature Review:

Load word (lw) is an I-type instruction. The LW instruction loads data from the data memory through a specified address, with a possible offset, to the destination register. The name I-type is short for immediate-type. I-type instructions use two register operands and one immediate operand. Figure below shows the I-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rd, funct3 and imm. The first three fields, op, rs1, and rd, are like those of R-type instructions. The imm field holds the 12-bit immediate. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the two fields rs1, rd, and imm. rs1 and imm are always used as source operands. rd is used as a destination.

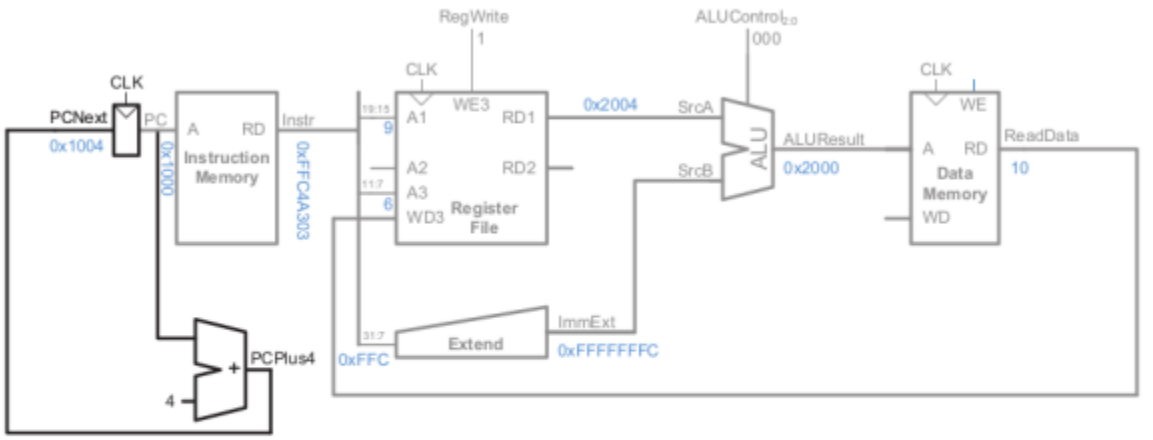
Task:

Data Path for Load Instruction

1.Write Verilog code for the sign extend and adder block. Make sure to name the input and output ports correctly with the correct number of bits. Attach the code.

2.Write a complete data path for Load instruction in Verilog by instantiating all the module blocks. Attach the code.

Block Diagram:



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Verilog Code:

Single_Cycle.v

```
`include "./PC/design.v"
`include "./Instruction-memory/design.v"
`include "./register-file/design.v"
`include "./ControlUnit/control_unit.v"
`include "./sign-extension/design.v"
`include "./ALU/design.v"
`include "./data-memory/design.v"
`include "./adder/design.v"
`include "./ControlUnit/DecoderModules/main_decoder.v"
`include "./ControlUnit/DecoderModules/alu_decoder.v"


module Single_Cycle(clk,reset);
    input clk,reset;

    // Interim Wire Declaration
        wire [31:0] PC_w;
```

```
        wire [31:0] Instruction;
        wire [31:0] RD1;
        wire [31:0] Extended;
        wire [31:0]ALUResult;
        wire RegWrite;
        wire [31:0] RD;
        wire [31:0] NextIns;
        wire [2:0]ALUControl;


    // Module Initialization
    program_counter program_counter (                                   // fetch cycle
==>
                                        .clk(clk),
                                        .reset(reset),
                                        .PC(PC_w),
                                        .PCNext(NextIns)
                                        );

    instruction_memory instruction_memory (
                                            .reset(reset),
                                            .A(PC_w),
                                            .RD(Instruction)
                                            );                          // <==

    control_unit control_unit (                                         // Decoding ==>
                                .zero(),
                                .op(Instruction[6:0]),
                                .func3(Instruction[14:12]),
                                .func7(),
                                .PCSrc(),
                                .RegWrite(RegWrite),
                                .ALUSrc(),
                                .MemWrite(),
                                .ResultSrc(),
                                .ImmSrc(),
                                .ALUControl(ALUControl)
                                );

    register_file register_file (
                                    .A1(Instruction[19:15]),
                                    .A2(),
                                    .A3(Instruction[11:7]),
                                    .clk(clk),
                                    .reset(reset),
```

```
                                .RD1(RD1),
                                .RD2(),
                                .WD3(RD),
                                .WE3(RegWrite)
                                );

    sign_extension sign_extension (
                                .Imm(Instruction[31:20]),
                                .ImmExt(Extended)
                                );                                  // <==

    Flags_ALU Flags_ALU (
                        .A(RD1),
                        .B(Extended),
                        .ctrl(ALUControl),
                        .Result(ALUResult),
                        .Z(),
                        .N(),
                        .C(),
                        .V()
                        );

    data_memory data_memory (
                                .clk(clk),
                                .A(ALUResult),
                                .WD(),
                                .WE(1'b0),
                                .RD(RD),
                                .reset(reset)
                            );

    Adder Adder(
                .Inp1(PC_w),
                .Inp2(32'd4),
                .Sum(NextIns)
                );

endmodule
```

sign_extension.v

```
module sign_extension(Imm,ImmExt);
```

```
    input [11:0] Imm;
    output [31:0] ImmExt;

    assign ImmExt = {{20{Imm[11]}},Imm};

endmodule
```

register_file.v

```
module register_file(A1,A2,A3,clk,reset,RD1,RD2,WD3,WE3);

    input[4:0] A1,A2,A3;
    input clk,reset,WE3;
    input[31:0] WD3;
    output [31:0]RD1,RD2;
    reg[31:0] register[31:0];

    initial begin
        register[0] <= 32'h00000000;
        register[1] <= 32'h00000000;
        register[2] <= 32'h00000000;
        register[3] <= 32'h00000000;
        register[4] <= 32'h00000000;
        register[5] <= 32'h00000000;
        register[6] <= 32'h00000000;
        register[7] <= 32'h00000000;
        register[8] <= 32'h00000000;
        register[9] <= 32'h00000000;
        register[10] <= 32'h00000000;
        register[11] <= 32'h00000000;
        register[12] <= 32'h00000000;
        register[13] <= 32'h00000000;
        register[14] <= 32'h00000000;
        register[15] <= 32'h00000000;
    end

    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
```

program_counter.v

```
module program_counter(PCNext,clk,reset,PC);

    input[31:0] PCNext;
    output reg[31:0] PC;
    input clk,reset;


    always @(posedge clk) begin
      if(reset == 1'b1) begin
        PC <=32'h00000000;
      end
      else begin
        PC <= PCNext;
      end
    end

endmodule
```

data_memory.v

```
module data_memory(clk,A,WD,WE,RD,reset);

    input[31:0] A,WD;
    input WE,clk,reset;
    output [31:0] RD;

    reg[31:0] memory [1023:0];

    initial begin
        memory[0] <= 32'h00000005;
        memory[1] <= 32'h00000007;
        memory[2] <= 32'h00000009;
        memory[3] <= 32'h0000000A;
    end
```

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

    always@(posedge clk) begin
        if(WE == 1'b1) begin
            memory[A] <=  WD;
        end
    end

endmodule
```

ALU.v

```
module Flags_ALU(A, B, ctrl, Result, Z, N, C, V);

  // inputs
  input [31:0] A, B;
  input [2:0] ctrl;

  // outputs
  output [31:0] Result;
  // Flags for Zero, Negative, Carry and Overflow respectively.
  output Z,N,C,V;

  // interim wires
  wire [31:0] A_and_B, A_or_B, B_not, A_sum_B;
  wire [31:0] S1;
  wire [31:0] not_Result;
  wire Cout, xor_A_Sum, xnor_A_B_ctrl0, ctrl1_not;

  // Logic Designing
  // And
  assign A_and_B = A & B;
  // Or
  assign A_or_B = A | B;
  // Not
  assign B_not = ~B;
  // 2x1 Mux for addition or subtraction
  assign S1 = (ctrl[0] == 1'b1) ? B_not : B;
  // Addition / Subtraction
  assign {Cout, A_sum_B} = A + S1 + ctrl[0];
  // Result output through 4x1 Mux
  assign Result = (ctrl[1:0] == 2'b00) ? A_sum_B :
```

```
                (ctrl[1:0] == 2'b01) ? A_sum_B :
                (ctrl[1:0] == 2'b10) ? A_and_B : A_or_B;

  // Flags Outputs
  // for zero checking
  assign not_Result = ~Result;
  assign Z = &(not_Result);
  // for negative checking
  assign N = Result[31];
  //for carry checking
  assign ctrl1_not = (~ctrl[1]);
  assign C = ctrl1_not & Cout;
  // for overflow checking
  assign xor_A_Sum = A_sum_B[31] ^ A[31];
  assign xnor_A_B_ctrl0 = ~(A[31] ^ B[31] ^ ctrl[0]);
  assign V = ctrl1_not & xor_A_Sum & xnor_A_B_ctrl0;

endmodule
```

adder.v

```
module Adder(Inp1,Inp2,Sum);

    input [31:0] Inp1,Inp2;
    output [31:0]Sum;

    assign Sum = Inp1+Inp2;

endmodule
```

control_unit.v

```
module control_unit(zero, op, func3, func7, PCSrc, RegWrite, ALUSrc, MemWrite,
ResultSrc, ImmSrc, ALUControl);

    input zero, func7;
    input [6:0] op;
    input [2:0] func3;

    output PCSrc, RegWrite, ALUSrc, MemWrite, ResultSrc;
```

```
    output [1:0] ImmSrc;
    output [2:0] ALUControl;

    wire [1:0] ALUOp;
    wire op5, Branch;

    assign op5 = op[5];

    main_decoder main_dec (
        .op(op), .RegWrite(RegWrite), .ALUSrc(ALUSrc), .MemWrite(MemWrite),
        .ResultSrc(ResultSrc), .Branch(Branch), .ImmSrc(ImmSrc), .ALUOp(ALUOp)
    );

    alu_decoder alu_dec (
        .ALUOp(ALUOp), .func3(func3), .op5(op5), .func7_5(func7),
.ALUControl(ALUControl)
    );

    assign PCSrc = zero & Branch;

endmodule
```

alu_decoder.v

```
module alu_decoder(ALUOp, func3, op5, func7_5, ALUControl);
    input [1:0] ALUOp;
    input [2:0] func3;
    input op5, func7_5;
    wire [1:0] signal;

    output [2:0] ALUControl;

    assign signal = {op5, func7_5};

    assign ALUControl = (ALUOp == 2'b00) ? 3'b000 :
                        (ALUOp == 2'b01) ? 3'b001 :
                        ((ALUOp == 2'b10) & (func3 == 3'b000) & (signal == 2'b11)) ?
3'b001 :
                        ((ALUOp == 2'b10) & (func3 == 3'b000) & (signal != 2'b11)) ?
3'b000 :
                        ((ALUOp == 2'b10) & (func3 == 3'b010)) ? 3'b101 :
                        ((ALUOp == 2'b10) & (func3 == 3'b110)) ? 3'b011 :
                        ((ALUOp == 2'b10) & (func3 == 3'b011)) ? 3'b010 : 3'b000;
```

```
endmodule
```

main_decoder.v

```
module main_decoder(op, RegWrite, ALUSrc, MemWrite, ResultSrc, Branch, ImmSrc, ALUOp);

    input[6:0] op;
    output RegWrite, ALUSrc, MemWrite, ResultSrc, Branch;
    output [1:0] ImmSrc, ALUOp;

    assign RegWrite = ((op == 7'b0000011) | (op == 7'b0110011)) ? 1'b1 : 1'b0;
    assign ALUSrc = ((op == 7'b0000011) | (op == 7'b0100011)) ? 1'b1 : 1'b0;
    assign MemWrite = ((op == 7'b0100011)) ? 1'b1 : 1'b0;
    assign ResultSrc = ((op == 7'b0000011)) ? 1'b1 : 1'b0;
    assign Branch = ((op == 7'b1100011)) ? 1'b1 : 1'b0;

    assign ImmSrc = ((op == 7'b0100011)) ? 2'b01 : (op == 7'b1100011) ? 2'b10 : 2'b00;
    assign ALUOp = ((op == 7'b0110011)) ? 2'b10 : (op == 7'b1100011) ? 2'b01 : 2'b00;

endmodule
```

instruction_memory.v

```
module instruction_memory(reset,A,RD);

    input[31:0] A;
    input reset;
    output [31:0] RD;

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

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

    initial begin
      // mem[0] <= 32'h00000013;
      // mem[1] <= 32'h0002A203;
      // mem[2] <= 32'h8142A203;

      mem[0] <= 32'h00050513;
      mem[1] <= 32'h00052303;
      mem[2] <= 32'h00152283;
      mem[3] <= 32'h00252283;
    end
```

```
endmodule
```

Assembly code:

addi x10,x10,0

lw x6,0(x10)

lw x5,1(x10)

lw x5,2(x10)

testBench.v

```
module tb();

    reg clk=0,reset;

    Single_Cycle dut(
                        .clk(clk),
                        .reset(reset)
                    );

    always begin
        clk = ~clk;
        #50;
    end

    initial begin
        reset <= 1'b1;
        #100;
        reset <= 1'b0;
        #500;
        $finish;
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0);
    end
```

```
endmodule
```

Wave Forms: