COAL Lab-07

Designing Microarchitecture-II

Name: Saad Nisar Butt

Reg. no: cs211246

Class: BSCS-3C-1

Literature Review:

Designing microarchitecture of 32-bit microprocessor which can perform different functionalities for computer brain. In this lab we have designed a complete single cycle core which can perform every instruction one by one. At this point the design is limited for loading data from an address from memory, that is Load word (lw).

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.

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	**rs1**	**func3**	**rd**	**op**
12 bits	5 bits	3 bits	5 bits	7 bits

I-type instructions have a 12-bit immediate field, but the immediates are used in 32-bit operations. For example, lw adds a 12-bit offset to a 32-bit base register. What should go in the upper half of the 32 bits? For positive immediates, the upper half should be all 0's, but for negative immediates, the upper half should be all 1's. This is called sign extension. An N-bit two's complement number is sign-extended to an M-bit number (M >N) by copying the sign bit (most significant bit) of the N-bit number into all of the upper bits of the M-bit number. Sign-extending a two's complement number does not change its value.

Lab Tasks

Task 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.

Verilog Code for Sign-Extended Block:

```
module Sign_Extension(ImmInst, ImmExt);

    input [11:0] ImmInst;
    output [31:0] ImmExt;

    assign ImmExt = {{20{ImmInst[11]}} , ImmInst};
endmodule
```

Verilog Code for Adder Block:

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

    // inputs
    input [31:0] Inp1, Inp2;
    // outputs
    output [31:0] Sum;
```

```
    //logic designing
    assign Sum = Inp1 + Inp2;

endmodule
```

Task 2:

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

Verilog Code

Single_Cycle.v

```
`include "./ProgramCounter/design.v"
`include "./InstructionMemory/design.v"
`include "./RegisterFile/design.v"
`include "./ControlUnit/Control_Unit.v"
`include "./ControlUnit/DecoderModules/main_decoder.v"
`include "./ControlUnit/DecoderModules/alu_decoder.v"
`include "./SignExtension/design.v"
`include "./ALU/design.v"
`include "./DataMemory/design.v"
`include "./Adder/design.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;

    wire [31:0] X,Y;

    // Module Instantiation
    Program_Counter Program_Counter (  // fetch cyle starts
```

```
        .PCNext(NextIns),
        .clk(clk),
        .reset(reset),
        .PC(PC_w)
    );

    Instruction_Memory Instruction_Memory (
        .reset(reset),
        .A(PC_w),
        .RD(Instruction)
    );                                  // fetch cyle ends

    Control_Unit Control_Unit (  // Decode cycle satrts
        .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]),
        .WD3(RD),
        .clk(clk),
        .reset(reset),
        .WE3(RegWrite),
        .RD1(RD1),
        .RD2()
    );

    Sign_Extension Sign_Extension (
        .ImmInst(Instruction[31:20]),
        .ImmExt(Extended)
    );                                // Decode cycle ends

    Flags_ALU ALU (
        .A(RD1),
```

```
        .B(Extended),
        .ctrl(ALUControl),
        .Result(ALUResult),
        .Z(),
        .N(),
        .C(),
        .V()
    );

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

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

endmodule
```

Program_Counter design.v

```
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
    end

endmodule
```

Instruction_Memory design.v

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

    input reset;
    input [31:0] A;

    output [31:0] RD;

    // Memory
    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'h00000013;
      mem[1] <= 32'h0002A203;
      mem[2] <= 32'h0042A203;
    end

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
```

Decoder_Modules

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
```

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
```

Register_File design.v

```
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 - Write
Enable
    input [31:0] WD3; // WD3 -> Write Data

    output [31:0] RD1, RD2;

    // 32 registers of 32-1bits
    reg [31:0] register[31:0];
    initial begin
      register[5] = 32'h00000000;
    end

    // reading from registers which are operands
    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
```

Sign_Extension design.v

```
module Sign_Extension(ImmInst, ImmExt);

    input [11:0] ImmInst;
```

```
    output [31:0] ImmExt;

    assign ImmExt = {{20{ImmInst[11]}} , ImmInst};
endmodule
```

ALU design.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
```

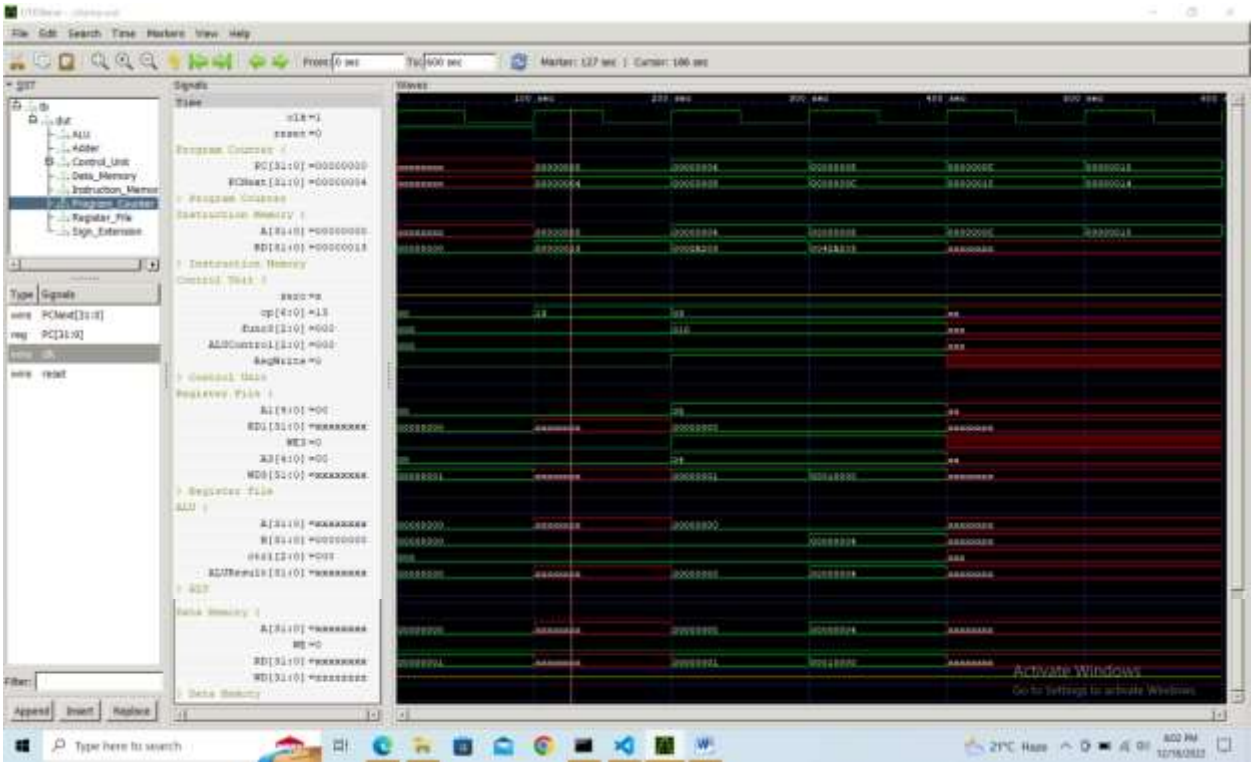
Data_Memory design.v

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

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

    output [31:0] RD;

    reg [31:0] memory [1023:0];
    initial begin
      memory[0] = 32'h00000001;
      memory[1] = 32'h00000010;
      memory[2] = 32'h00000100;
      memory[3] = 32'h00001000;
      memory[4] = 32'h00010000;
    end

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

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

endmodule
```

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
```

Waveforms



* _____ *