**COAL Lab # 08**

**Designing Microarchitecture-III**

**Name:** Muhammad Usman

**Class:** BSCS 3C1

**Reg No:** cs211208

**Literature Review:**

Designing microarchitecture of 32-bit microprocessor which can perform different functionalities for computer. In this lab we have extended the functionality of 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) and storing data from registers to memory, That is Store word (sw).

Store instructions are S-type instructions. The store word instruction, sw, copies data from a register to memory. The register is not changed. The memory address is specified using a base/register pair. The name S-type is short for Store-type. S-type instructions use two register operands and one immediate operand. The figure below shows the S-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rs2, funct3, and imm. The first four fields, op, rs1, rs2, and funct3 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 three fields rs1, rs2, and imm. rs and imm are always used as source operands. rs2 is used as another source for store instructions.

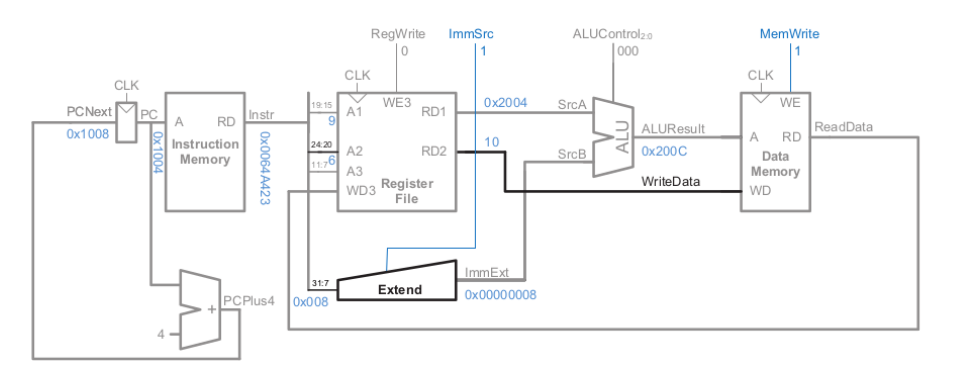
Like I-type instructions, S-type also has 12-bit immediate. The immediate in store instructions are split into two halves: the lower bits are stored in Instr[11:7] bit and the upper bits are stored in Instr[31:25]. First, we make the 12-bit immediate by concatenating the upper and lower part of immediate and then sign extend it for further use.

**Lab Task:**

**Task:**

1. Write a complete data path for Store Instruction in Verilog by instantiating all the module blocks. Attach the code.

**Block Diagram:**

****

**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 MemWrite;

        wire [31:0] RD;

        wire [31:0] RD2;

        wire [31:0] NextIns;

        wire [1:0] ImmSrc;

        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(MemWrite),

                                .ResultSrc(),

                                .ImmSrc(ImmSrc),

                                .ALUControl(ALUControl)

                                );

    register\_file register\_file (

                                    .A1(Instruction[19:15]),

                                    .A2(Instruction[24:20]),

                                    .A3(Instruction[11:7]),

                                    .clk(clk),

                                    .reset(reset),

                                    .RD1(RD1),

                                    .RD2(RD2),

                                    .WD3(RD),

                                    .WE3(RegWrite)

                                    );

    sign\_extension sign\_extension (

                                    .Imm(Instruction),

                                    .ImmExt(Extended),

                                    .ImmSrc(ImmSrc)

                                    );                                  // <==

    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(RD2),

                                .WE(MemWrite),

                                .RD(RD),

                                .reset(reset)

                            );

    Adder Adder(

                .Inp1(PC\_w),

                .Inp2(32'd4),

                .Sum(NextIns)

                );

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

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

      // sw x5,0x4(x0)

      // sw x6, 0x8(x0)

      // sw x7, 0xC(x0)

      mem[0] <= 32'h00502223;

      mem[1] <= 32'h00602423;

      mem[2] <= 32'h00702623;

    end

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[5] <= 32'h00000100;

        register[6] <= 32'h00000200;

        register[7] <= 32'h00000300;

        register[10] <= 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

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

adder.v

module Adder(Inp1,Inp2,Sum);

    input [31:0] Inp1,Inp2;

    output [31:0]Sum;

    assign Sum = Inp1+Inp2;

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

sign\_extension.v

module sign\_extension(Imm,ImmSrc,ImmExt);

    input [31:0] Imm;

    input[1:0] ImmSrc;

    output [31:0] ImmExt;

    assign ImmExt = (ImmSrc == 2'b00) ? {{20{Imm[31]}},Imm[31:20]} : {{20{Imm[31]}},Imm[31:25],Imm[11:7]};

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'h00000000;

    end

    assign RD = (WE == 1'b0) ? memory[A] : 32'd0;

    always@(posedge clk) begin

        if(WE == 1'b1) begin

            memory[A] <=  WD;

        end

    end

endmodule

**GTK WAVE:**

