CALIFORNIA STATE UNIVERSITY LONG BEACH

CECS 341 - Computer Architecture and Organization

Lab Assignment 3

RISC-V
Single Cycle Processor
(continue)

Professor: *Maryam S. Hosseini*
Team members:
Nikki Valerie Benitez
Fiona Le
Thanh Nguyen
Marie Payad
Due date: *11/27/2020*

I. Data Memory:

1. Introduction:

Data memory stores 128 data each with 32 bits (128 x 32). This block has two tasks, and requires 4 inputs and produces 1 output in total.

- Read data from memory:
  - Inputs:
    - An address that needs to be read (9 bits, addr) - when MemRead is set, this address is the reading address.
    - The read enable signal (MemRead)
  - Output:
    - The data from the address line (read_data)
- Write data into memory:
  - Inputs:
    - An address that needs to be written (9 bits, addr) - when MemWrite is set, this address is the writing address.
    - The data need to be written into the memory (write_data)
    - The write enable signal (MemWrite)

2. Procedure:

➢ We defined a 128 x 32 2D array called ***Memory*** to store 128 data each with 32 bits.

➢ We used @(posedge MemWrite) to check every time that the MemWrite signal is set, then write the data into the memory.

➢ We also checked that if a MemRead signal is set, read the data from the address line, and write it into the output variable read_data.

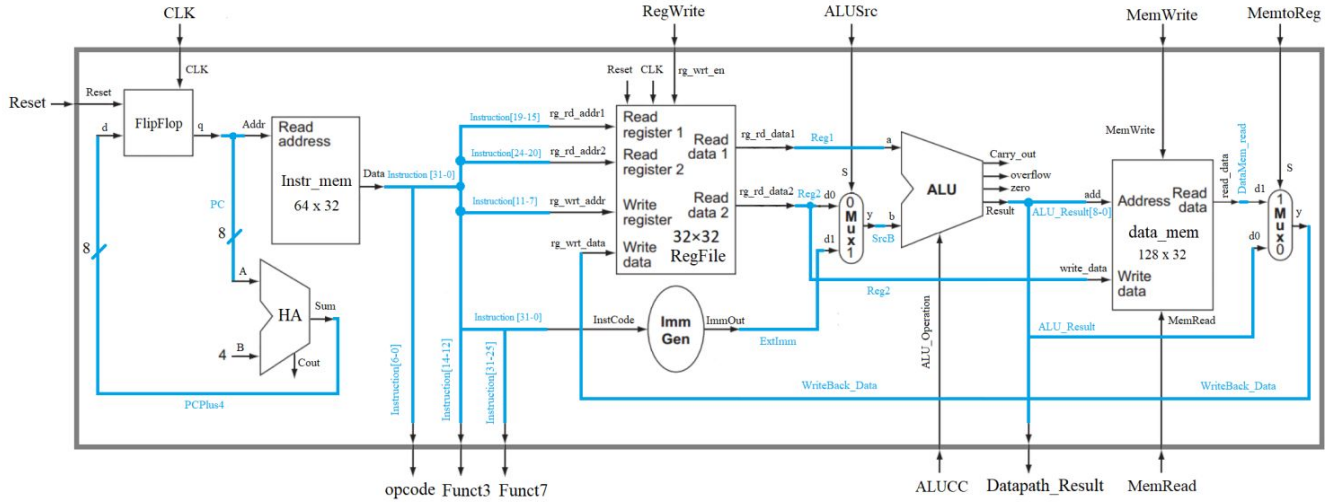
3. Appendix:

```
`timescale 1ns / 1ps

module DataMem(
    MemRead,
    MemWrite,
    addr,
    write_data,
    read_data
    );

    // Define I/O ports
    input          MemRead;    // Control signal for memory read
    input          MemWrite;   // Control signal for memory write
    input   [8:0]  addr;       // Input Address - 9 LSB bits of the ALU output
    input  [31:0]  write_data; // Data that needs to be written into the address
```

```
    output [31:0] read_data;  // Contents of memory location at Address

    // Describe data_mem behaviour
    // Memory Initialization
    reg [31:0] Memory[127:0]; // 128 x 32 memory block

    // Memory Write
    always @(posedge MemWrite) // write routine, creates flip-flops, Active
high MemWrite
        begin
            Memory[addr] <= write_data; // Word written at location addr
        end

    // Memory Read
    assign read_data = (MemRead) ? Memory[addr] : 32'd0;

endmodule
```

II. Datapath:

1. Introduction:



Datapath is the part of the processor that contains the hardware necessary to perform operations required by the processor.

- This module has ***8*** inputs, some of them (RegWrite, ALUSrc, MemWrite, MemtoReg, MemRead, ALUCC) are control signals which tell the datapath what needs to be done. The other inputs are clock and reset signal.
- It also produces ***4*** outputs: The Instruction from memory (includes opcode, funct7, and funct3) and the result of ALU (called Datapath_Result)

2. Procedure:

- ➢ We used all the submodules (Flip Flop, HA, Instruction Memory, Register File, Immediate Generator, ALU, MUX - two instantiations, Data Memory) as components.
- ➢ We used wire to connect all the components to complete the Datapath.
  - First, the instruction execution starts by using the program counter (Flip Flop, HA) to supply the instruction address to the instruction memory.
  - Then based on the Instruction from Instruction Memory, we go over the Register File to get data from memory, and the Immediate Generator to determine which operation needs to execute.
  - If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register (ALU_Result is the output WriteBack_Data after MUX). If the operation is load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the register file (ALU goes into Data Memory as the address line).

3. **Result:**

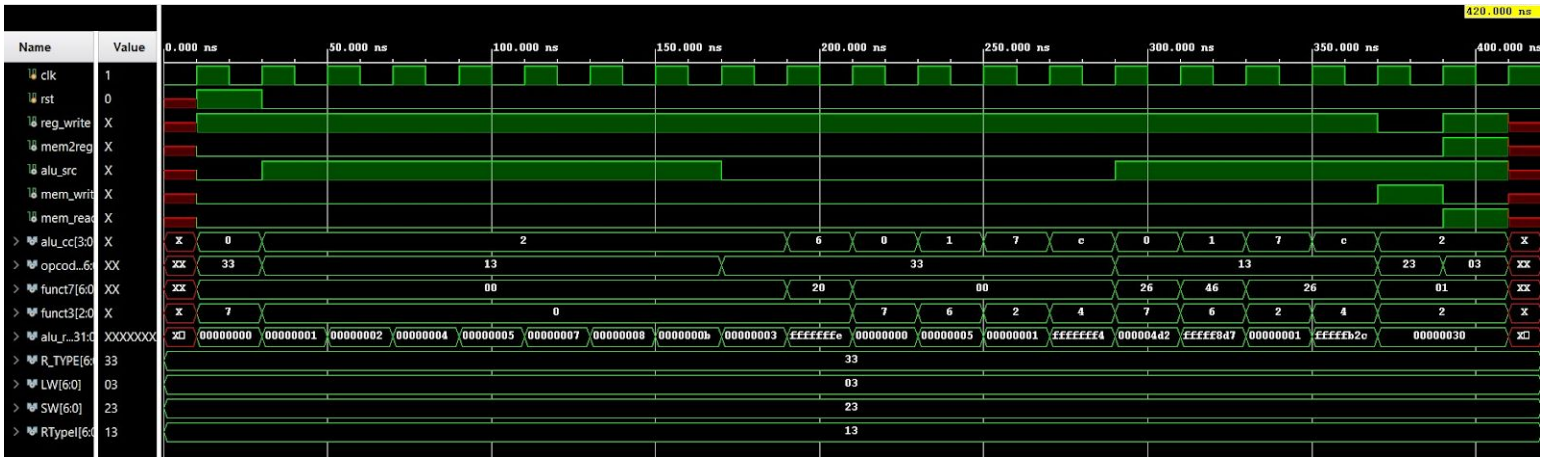
- Testbench:

```
`timescale 1ns / 1ps

module dp_tb_top();

    /** Clock & reset **/
    reg clk, rst;
    always
        begin
        #10;
        clk = ~clk;
        end

    initial
        begin
        clk = 0;
        @(posedge clk);
        rst = 1;
        @(posedge clk);
        rst = 0;
        end

    /** DUT Instantiation **/
    wire        reg_write;
    wire        mem2reg;
    wire        alu_src;
    wire        mem_write;
```

```
    wire        mem_read;
    wire  [3:0] alu_cc;
    wire  [6:0] opcode;
    wire  [6:0] funct7;
    wire  [2:0] funct3;
    wire [31:0] alu_result;


    data_path dp_inst(
        .clk(clk),
        .reset(rst),
        .reg_write(reg_write),
        .mem2reg(mem2reg),
        .alu_src(alu_src),
        .mem_write(mem_write),
        .mem_read(mem_read),
        .alu_cc(alu_cc),
        .opcode(opcode),
        .funct7(funct7),
        .funct3(funct3),
        .alu_result(alu_result)
        );

    /** Stimulus **/
    wire [6:0] R_TYPE, LW, SW, RTypeI;

    assign R_TYPE = 7'b0110011;
    assign LW     = 7'b0000011;
    assign SW     = 7'b0100011;
    assign RTypeI = 7'b0010011;


    assign alu_src   = (opcode == LW || opcode == SW || opcode == RTypeI);
    assign mem2reg   = (opcode == LW);
    assign reg_write = (opcode == R_TYPE || opcode == LW || opcode == RTypeI);
    assign mem_read  = (opcode == LW);
    assign mem_write = (opcode == SW);

    assign alu_cc = ((opcode == R_TYPE || opcode == RTypeI) // add
                  && (funct7 == 7'b0000000) && (funct3 == 3'b000)) ? 4'b0010 :
                    ((opcode == R_TYPE || opcode == RTypeI) // sub
                  && (funct7 == 7'b0100000)) ? 4'b0110 :
                    ((opcode == R_TYPE || opcode == RTypeI) // xor
                  && (funct7 == 7'b0000000) && (funct3 == 3'b100)) ? 4'b1100 :
                    ((opcode == R_TYPE || opcode == RTypeI) // or
                  && (funct7 == 7'b0000000) && (funct3 == 3'b110)) ? 4'b0001 :
                    ((opcode == R_TYPE || opcode == RTypeI) // and
                  && (funct7 == 7'b0000000) && (funct3 == 3'b111)) ? 4'b0000 :
                    ((opcode == R_TYPE || opcode == RTypeI) // slt
```

```
                    && (funct7 == 7'b0000000) && (funct3 == 3'b010)) ? 4'b0111 :
                      ((opcode == R_TYPE || opcode == RTypeI) // nori
                    && (funct3 == 3'b100)) ? 4'b1100 :
                      ((opcode == R_TYPE || opcode == RTypeI) // ori
                    && (funct3 == 3'b110)) ? 4'b0001 :
                      ((opcode == R_TYPE || opcode == RTypeI) // addi
                    && (funct3 == 3'b010)) ? 4'b0111 :
                      ((opcode == LW || opcode == SW) // load or store
                    && (funct3 == 3'b010))? 4'b0010 : 0;

      initial
        begin
        #420;
        $finish;
        end
endmodule
```

- Waveform:



4. Appendix:

```
`timescale 1ns / 1ps

`include "ALU.v"
`include "FlipFlop.v"
`include "Instr_mem.v"
`include "RegFile.v"
`include "Imm_Gen.v"
`include "Mux.v"
`include "Data_mem.v"

module data_path #(
    parameter PC_W       = 8,  // Program Counter
    parameter INS_W      = 32, // Instruction Width
    parameter RF_ADDRESS = 5,  // Register File Address
```

```
    parameter DATA_W     = 32, // Data Write Data
    parameter DM_ADDRES  = 9,  // Data Memory Address
    parameter ALU_CC_W   = 4   // ALU Control Code Width
    )(
    input                   clk,       // CLK in Datapath Figure
    input                   reset,     // Reset in Datapath Figure
    input                   reg_write, // RegWrite in Datapath Figure
    input                   mem2reg,   // MemtoReg in Datapath Figure
    input                   alu_src,   // ALUSrc in Datapath Figure
    input                   mem_write, // MemWrite in Datapath Figure
    input                   mem_read,  // MemRead in Datapath Figure
    input  [ALU_CC_W - 1:0] alu_cc,    // ALUCC in Datapath Figure
    output            [6:0] opcode,    // opcode in Datapath Figure
    output            [6:0] funct7,    // Funct7 in Datapath Figure
    output            [2:0] funct3,    // Funct3 in Datapath Figure
    output   [DATA_W - 1:0] alu_result // Datapath_Result in Datapath Figure
    );

    // Define Datapath Wires
    wire  [7:0] PC;
    wire  [7:0] PCPlus4;
    wire [31:0] Instruction;
    wire [31:0] Reg1;
    wire [31:0] Reg2;
    wire [31:0] ExtImm;
    wire [31:0] WriteBack_Data;
    wire [31:0] SrcB;
    wire [31:0] ALU_Result;
    wire [31:0] DataMem_read;
    wire        Carry_out, overflow, zero;

    // Assign relationships
    // Controller
    assign opcode = Instruction[6:0];
    assign funct3 = Instruction[14:12];
    assign funct7 = Instruction[31:25];

    // HA
    assign PCPlus4 = PC + 8'd4;

    // Flip FLop
    FlipFlop flip_flop(
        // inputs
        .clk(clk),     // 1 bit
        .reset(reset), // 1 bit
        .d(PCPlus4),   // 8 bits
        // output
        .q(PC)         // 8 bits
        );
```

```
// Instruction Memory
InstMem instruction_memory(
    // input
    .addr(PC),                // 8 bits
    // output
    .instruction(Instruction) // 32 bits
);

// Register File
RegFile register_file(
    // inputs
    .clk(clk),                          // 1 bit
    .reset(reset),                      // 1 bit
    .rg_wrt_en(reg_write),              // 1 bit
    .rg_wrt_addr(Instruction[11:7]),    // 5 bits
    .rg_rd_addr1(Instruction[19:15]),   // 5 bits
    .rg_rd_addr2(Instruction[24:20]),   // 5 bits
    .rg_wrt_data(WriteBack_Data),       // 32 bits
    // outputs
    .rg_rd_data1(Reg1),                 // 32 bits
    .rg_rd_data2(Reg2)                  // 32 bits
    );

// Mux for ALU
MUX21 mux_src(
    // inputs
    .D1(Reg2),   // 32 bits
    .D2(ExtImm), // 32 bits
    .S(alu_src), // 1 bit
    // output
    .Y(SrcB)     // 32 bits
    );

// Sign Extend
Imm_Gen immediate_generator(
    // input
    .InstCode(Instruction), // 32 bits
    // output
    .ImmOut(ExtImm)         // 32 bits
    );

// ALU
alu_32 ALU(
    // inputs
    .A_in(Reg1),            // 32 bits
    .B_in(SrcB),            // 32 bits
    .ALU_Sel(alu_cc),       // 4 bits
    // outputs
```

```
        .ALU_Out(ALU_Result),  // 32 bits
        .Carry_Out(Carry_out), // 1 bit
        .Zero(zero),           // 1 bit
        .Overflow(overflow)    // 1 bit
        );

     assign alu_result = ALU_Result;

    // Mux for Data memory
    MUX21 mux_memory_to_register(
        // inputs
        .D1(ALU_Result),   // 32 bits
        .D2(DataMem_read), // 32 bits
        .S(mem2reg),       // 1 bit
        // output
        .Y(WriteBack_Data) // 32 bits
        );

    // Data Memory
    DataMem data_memory(
        // inputs
        .MemRead(mem_read),      // 1 bit
        .MemWrite(mem_write),    // 1 bit
        .addr(ALU_Result[8:0]),  // 9 bits
        .write_data(Reg2),       // 32 bits
        // output
        .read_data(DataMem_read) // 32 bits
        );
endmodule
```