
# Digital Design - Course Project Report

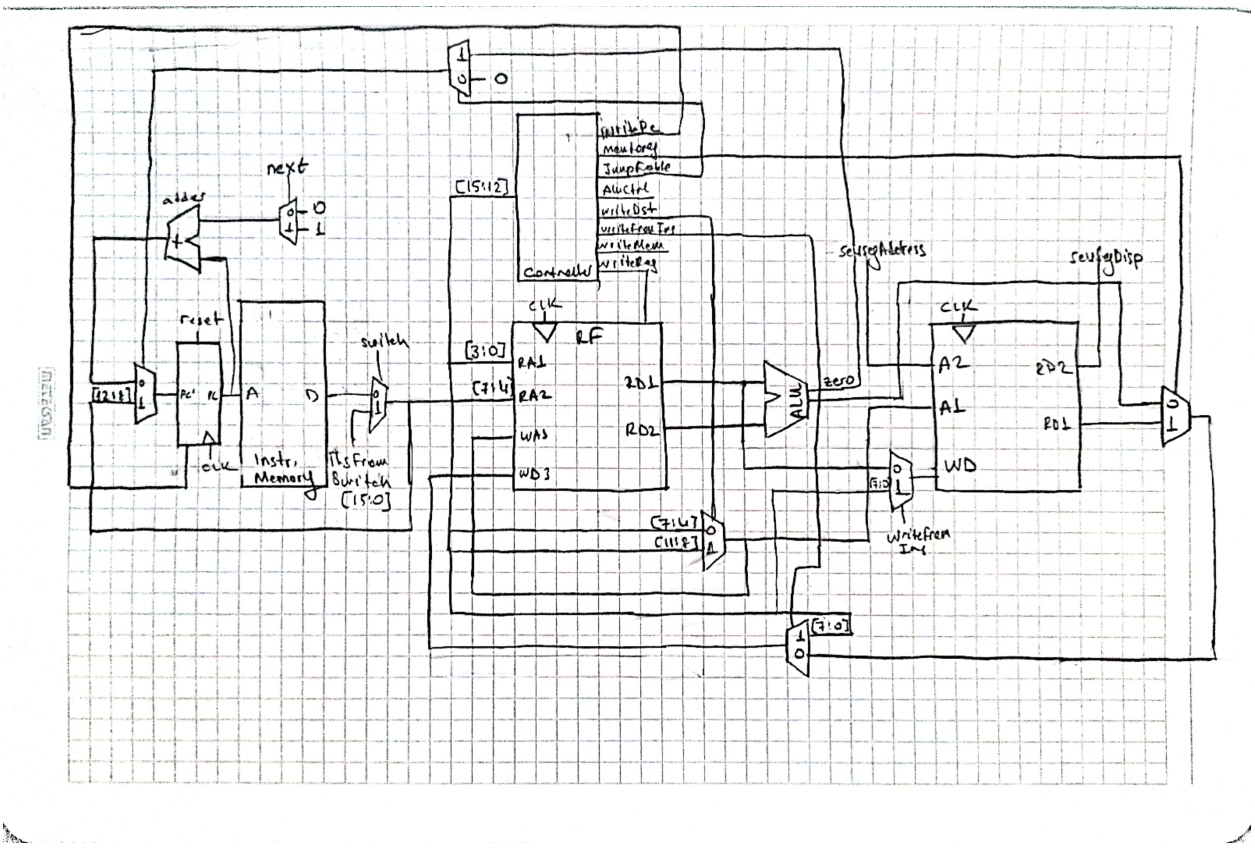
CS 223 - Section 4

Yağız YAŞAR

21902951

December 25, 2020

1) Datapath




Since there will be 4 different designs for partioning the 16 bits instructions which are, first, data writing to the register and data memory from the other memory array, and addition, second, data writing to the register and data memory from the given data, third, branch, and, fourth, stopping the execution.

First, xxx_x_xxxx_xxxx_xxxx
Second, xxx_x_xxxx_xxxxxxxx
Third, xxx_xxxxx_xxxx_xxxx
Fourth, xxx_xxxxxxxxxxxx

In my datapath, I am using 8 different multiplexers in order to arrange these partionings, required inputs for program counter, register file, data memory and instructions.

So, starting from the left, the program counter arranges instruction memory's input, 5-bit, which helps us to access to the ROM's values which is hard coded in to the instruction memory.

Program counter's input is arranged by 2 multiplexers and 1 adder. One of the mux's determine the input of the PC directly, if it jumpEnable is active, the next PC will be instruction[12:8], otherwise if the button next is pushed it will add one to the current PC value and next PC will be equal to PC value.

Instruction memory has hard coded instructions in itself and it has 32x16 bits of array. Output is determined by input from PC and multiplexer. If switch button is pressed, instruction will be taken from the switches.

Register file has 6 input ports, 2 output ports. 6 input ports contain readAddress1, readAddress2, writeAddress3, writeData3, clk and regWrite, on the other side output ports are readData1 and 2. ReadAddress 1 and 2, clk and regWrite are controlled by instructions. WriteAddress3 and writeData3 are both controlled by instructions and multiplexers, their range of array for the instructions can change depending on these multiplexers.

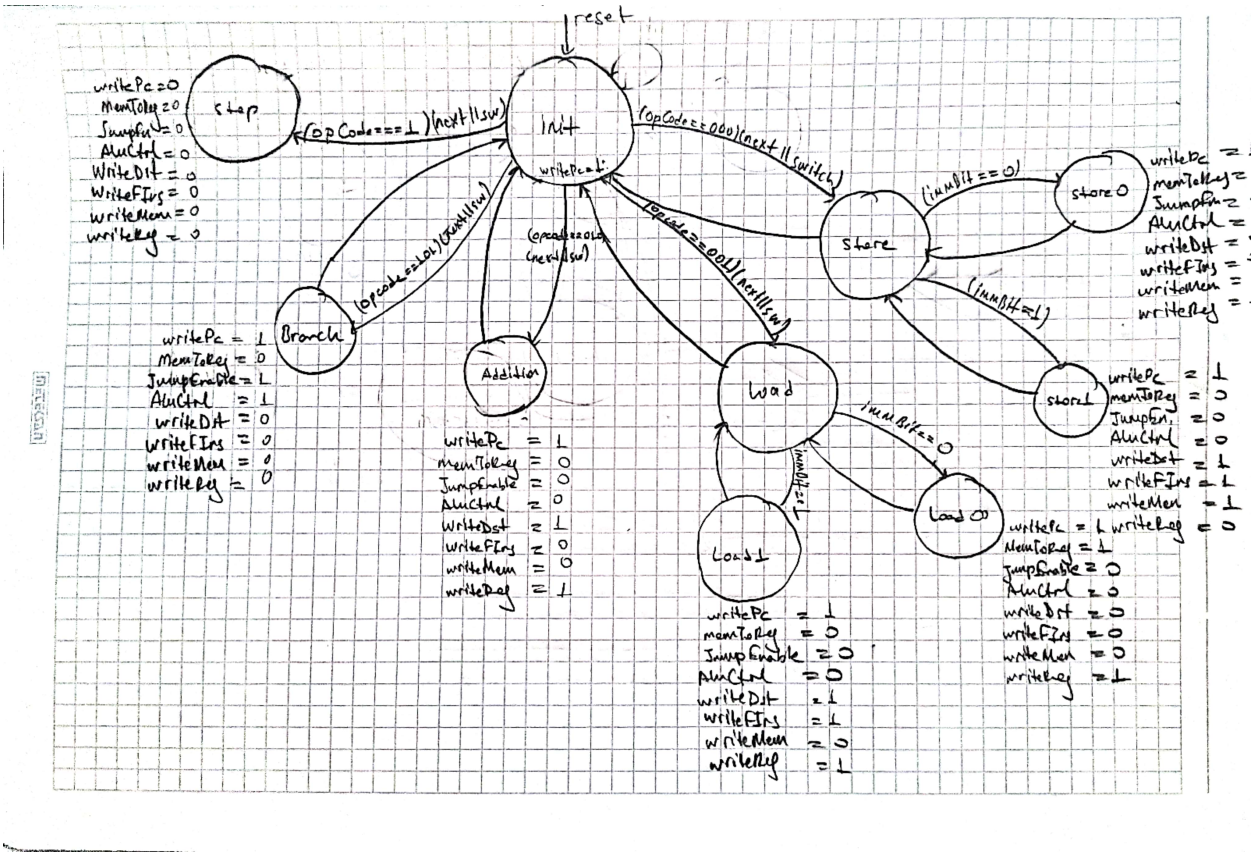
ALU gets its input ports from readData1 and readData2 which comes out from the register file. It has 2 outputs and 2 states. One of the state is "branch if equal". Zero output means that sources are equal and it means that branches are equal. Other state is "addition". Zero output loses its meaning, and ALU result will be get written to the register file depending on the multiplexers.

Data memory controls the seven segment display and values to put in register. It has 4 input ports including clk alongside with address1, address2 and writeData, also 2 output ports readData2 and

readData1. WriteData is taken from the switches by hand or register file's readData1. Address1 is also similar to the writeAddress3 of the register file. ReadData2 controls the seven segment display on the Basys3 and readData1 will be written to the register file if the controller is on the right state.

There are total of 8 multiplexers to arrange inputs more precisely for the controller.

2) Controller



(opCode = instruction[15:13], immediateBit = instruction[12], nextIns = debouncerD, switch = debouncerC )

Controller has 8 states including init.

Store0 and Store1 states stores data into the data memory, when the opCode is 000 and next or switch buttons are pushed. Yet, when immediateBit is 0, it will load the data from the register file's given address on the instruction. Also, when the immediateBit is 1, it will load the data to given address from the switches by hand.

Load0 and Load1 states loads the data into the register file, when the opCode is 001 and next or switch buttons are pushed. Like the Store0 and Store1, they rather take the data from the other memory array, data memory, or they take the data from the switches on the Basys3.

Addition state sums up the sources which are from readData1 and readData2, and comes from register file, at ALU.

Branch state determines if the sources are equal, if they are equal, the instruction address will jump to the address in the instruction for the design.

Stop execution halts the whole process, it is needed to reset the state to be back normal.

3) SystemVerilog Codes - Design

Note: I did not include seven segment and debounce since I used the given codes on the Moodle, but I included them in the zip file to demonstrate on Zoom.

**********

```
`timescale 1ns / 1ps

module top_module(input logic clk,
                  input logic [4:0] buttons,
                  input logic [15:0] switches,
                  output logic [15:0] leds,
                  output logic [6:0] seg,
                  output logic dp,
                  output logic [3:0] an);

    logic [7:0] sevSegDisplay;
    logic [3:0] sevSegAddress;
    logic [15:0] instruction, nextInstruction;

    logic reset, dispPrev, switch, dispNext, nextInstructionButton;

    debounce resetButton(clk, buttons[0], reset);
    debounce dispPrevButton(clk, buttons[1], dispPrev);
    debounce switchButton(clk, buttons[2], switch);
    debounce dispNextButton(clk, buttons[3], dispNext);
    debounce nextInsButton(clk, buttons[4], nextInstructionButton);


    always_ff @(posedge clk) begin
        if (reset) begin
            sevSegAddress <= 0;
        end else if (dispNext) begin
            if (sevSegAddress == 4'b1111) begin
                sevSegAddress <= 4'b0000;
            end else begin
                sevSegAddress <= sevSegAddress + 1;
            end
        end else if (dispPrev) begin
            if (sevSegAddress == 4'b0000) begin
                sevSegAddress <= 4'b1111;
```

```
                end else begin
                        sevSegAddress <= sevSegAddress - 1;
                end
            end
        end


        datapath path(clk, reset, nextInstructionButton, switch, switches, sevSegAddress,
writePc, memToReg, jumpEnable, aluCtrl, writeDst, writeFromIns, writeMemory, writeReg,
                instruction, nextInstruction, sevSegDisplay);

        controller controllerUnit(clk, nextInstructionButton, switch, instruction[15:13],
instruction[12], writePc, memToReg, jumpEnable, aluCtrl, writeDst, writeFromIns,
writeMemory, writeReg);

        assign leds = instruction;

        SevSeg_4digit sevSeg(clk, sevSegAddress, 0, sevSegDisplay[7:4], sevSegDisplay[3:0],
seg, dp, an);
endmodule

**********************************************************************************
`timescale 1ns / 1ps

module datapath(input logic clk, reset, next, switch,
                        input logic [15:0] instructionFromSwitch, input logic [3:0]
sevSegAddress,
                        input logic writePc, memToReg, jumpEnable, aluCtrl, writeDst,
writeFromIns, writeMemory, writeReg,
                                output logic [15:0] instruction, nextInstruction, output logic
[7:0] sevSegDisplay);

        logic [3:0] writeAddressOfReg, addressOfMem;
        logic value, zero;
        logic [4:0] nextPc, pc;
        logic [7:0] readDataReg1, readDataReg2, writeDataToReg;
        logic [7:0] readDataMem, writeDataToMem, readDataMemMux;
        logic [7:0] aluResult;
        logic [15:0] instr, nextInstr;

    instruction_memory ROM(pc, instr, nextInstr);

    /*
```

```
    * Program Counter
    */
   always_comb begin
      if ( jumpEnable & zero ) begin
         nextPc = instruction[12:8];
      end
      else begin
         if (switch)
             nextPc = pc;
         else if (next)
             nextPc = pc + 1;
         else
            nextPc = pc;
      end
   end

   always_ff @( posedge clk ) begin
      if (reset) begin
         pc <= 0;
      end
      else if (writePc) begin
         pc <= nextPc;
      end
   end


   always_comb begin
      if (switch) begin
          instruction = instructionFromSwitch;
      end
      else begin
         instruction = instr;
      end
   end

         //register file
         mux_2to1 #4 writeAddressReg(instruction[7:4], instruction[11:8], writeDst,
writeAddressOfReg);
         mux_2to1 #8 writeDataReg(readDataMemMux, instruction[7:0], writeFromIns,
writeDataToReg);
         register_file regFile(clk, reset, writeMemory, instruction[3:0], instruction[7:4],
writeAddressOfReg, writeDataToReg, readDataReg1, readDataReg2);
```

```
	//alu
	alu alu(readDataReg1, readDataReg2, aluCtrl, aluResult, zero);

	//data memory
	mux_2to1 #8 writeDataMem(readDataReg1, instruction[7:0], writeFromIns,
writeDataToMem);
	mux_2to1 #4 writeAddressMem(instruction[7:4], instruction[11:8], writeDst,
addressOfMem);
	data_memory dataMem(clk, reset, writeMemory, addressOfMem, sevSegAddress,
writeDataToMem, readDataMem, sevSegDisplay);
	mux_2to1 #8 readDataToMux(aluResult, readDataMem, memToReg,
readDataMemMux);

endmodule

**********************************************************************
`timescale 1ns / 1ps

module controller( input logic clk, nextInstructionButton, switch, input logic [2:0] opCode, input
logic immediateBit,
                                    output logic writePc, memToReg, jumpEnable, aluCtrl,
writeDst, writeFromIns, writeMemory, writeReg);

	always_comb begin
	if (nextInstructionButton || switch) begin
		case(opCode)
		3'b000: begin //store
			if (immediateBit == 0) begin
				writePc <= 1;
				memToReg <= 0;
				jumpEnable <= 0;
				aluCtrl <= 0;
				writeDst <= 0;
				writeFromIns <= 0;
				writeMemory <= 1;
				writeReg <= 0;
			end else begin
				writePc <= 1;
				memToReg <= 0;
				jumpEnable <= 0;
				aluCtrl <= 0;
				writeDst <= 1;
				writeFromIns <= 1;
```

```
                writeMemory <= 1;
                writeReg <= 0;
        end
end
3'b001:begin // load
        if (immediateBit == 0) begin
                writePc <= 1;
                memToReg <= 1;
                jumpEnable <= 0;
                aluCtrl <= 0;
                writeDst <= 0;
                writeFromIns <= 0;
                writeMemory <= 0;
                writeReg <= 1;
        end else begin
                writePc <= 1;
                memToReg <= 0;
                jumpEnable <= 0;
                aluCtrl <= 0;
                writeDst <= 1;
                writeFromIns <= 1;
                writeMemory <= 0;
                writeReg <= 1;
        end
end
3'b010: begin //addition
        writePc <= 1;
        memToReg <= 0;
        jumpEnable <= 0;
        aluCtrl <= 0;
        writeDst <= 1;
        writeFromIns <= 0;
        writeMemory <= 0;
        writeReg <= 1;
end
3'b101: begin //branch if equals
        writePc <= 1;
        memToReg <= 0;
        jumpEnable <= 1;
        aluCtrl <= 1;
        writeDst <= 0;
        writeFromIns <= 0;
        writeMemory <= 0;
```

```
                writeReg <= 0;
            end
            3'b111: begin //stop exec
                writePc <= 0;
                memToReg <= 0;
                jumpEnable <= 0;
                aluCtrl <= 0;
                writeDst <= 0;
                writeFromIns <= 0;
                writeMemory <= 0;
                writeReg <= 0;
            end
            default: begin
                writePc <= 1;
                memToReg <= 0;
                jumpEnable <= 0;
                aluCtrl <= 0;
                writeDst <= 0;
                writeFromIns <= 0;
                writeMemory <= 0;
                writeReg <= 0;
            end
        endcase // opcode
        end else begin
          writePc <= 1;
                memToReg <= 0;
                jumpEnable <= 0;
                aluCtrl <= 0;
          writeDst <= 0;
                writeFromIns <= 0;
                writeMemory <= 0;
                writeReg <= 0;
        end
        end

endmodule
**********************************************************************************
`timescale 1ns / 1ps
module mux_2to1 #(parameter WIDTH = 8)(input logic [WIDTH-1:0] input1, input2,  input
logic control, output logic [WIDTH-1:0] output1);
        assign output1 = control ? input2 : input1;
endmodule
**********************************************************************************
```

```
`timescale 1ns / 1ps
module instruction_memory(input logic [4:0] address, output logic [15:0] instruction,
nextInstruction);
	logic [15:0] instructionMemory[31:0];


   assign instructionMemory[0] = 16'b0001000000000111;
   assign instructionMemory[1] = 16'b0001000100000010;
   assign instructionMemory[2] = 16'b0010000000000000;
   assign instructionMemory[3] = 16'b0010000000010001;
   assign instructionMemory[4] = 16'b0011001000000000;
   assign instructionMemory[5] = 16'b0011001100000000;
   assign instructionMemory[6] = 16'b0011010000000001;
   assign instructionMemory[7] = 16'b1010100100100001;
   assign instructionMemory[8] = 16'b0100001100110000;
   assign instructionMemory[9] = 16'b0100001000100100;
   assign instructionMemory[10] = 16'b1010010100000000;
   assign instructionMemory[11] = 16'b0000000000110011;
   assign instructionMemory[12] = 16'b1110000000000000;

	assign instruction = instructionMemory[address];
	assign nextInstruction = instructionMemory[address + 1];
endmodule
**************************************************************************
`timescale 1ns / 1ps
module register_file(input logic clk, reset, writeEnable, logic [3:0] readReg1, readReg2,
writeReg3, logic [7:0] writeData3,
	output logic [7:0] readData1, readData2);

	logic [7:0] register[15:0];

	always_ff @(posedge clk) begin
	   if (reset) begin
	     for (int i = 0; i < 16; i++) begin
	           register[i] <= 8'b0;
	     end
	   end  else if (writeEnable) begin
	               register[writeReg3] = writeData3;
	       end
	end

	assign readData1 = register[readReg1];
	assign readData2 = register[readReg2];
```

```
endmodule
**************************************************************************
`timescale 1ns / 1ps
module alu(input logic [7:0] a, b, input logic opcode, output logic [7:0] result, logic zero);

	always_comb begin
		case (opcode)
			1'b0: result <= a + b;
			1'b1:
				if (a == b) begin
					result <= 8'b0;
				end
				else begin
					result <= 8'b1;
				end
			default: result <= a + b;
		endcase
	end

	assign zero = (result == 8'b0);

endmodule
**************************************************************************
`timescale 1ns / 1ps
module alu(input logic [7:0] a, b, input logic opcode, output logic [7:0] result, logic zero);

	always_comb begin
		case (opcode)
			1'b0: result <= a + b;
			1'b1:
				if (a == b) begin
					result <= 8'b0;
				end
				else begin
					result <= 8'b1;
				end
			default: result <= a + b;
		endcase
	end

	assign zero = (result == 8'b0);
```

```
endmodule
**************************************************************************
```