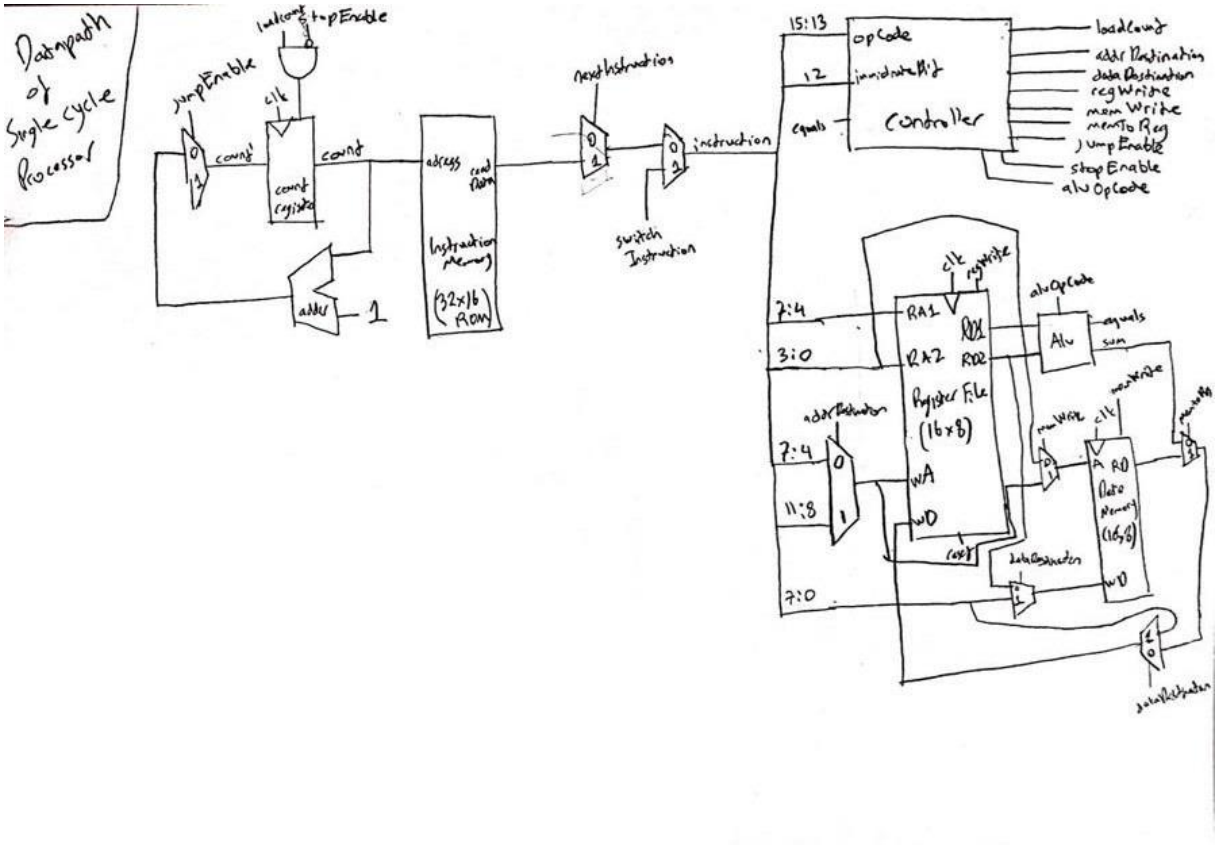
CS223
Laboratory
Project
Single-cycle
Processor

Arda Önal
21903350
Section 01

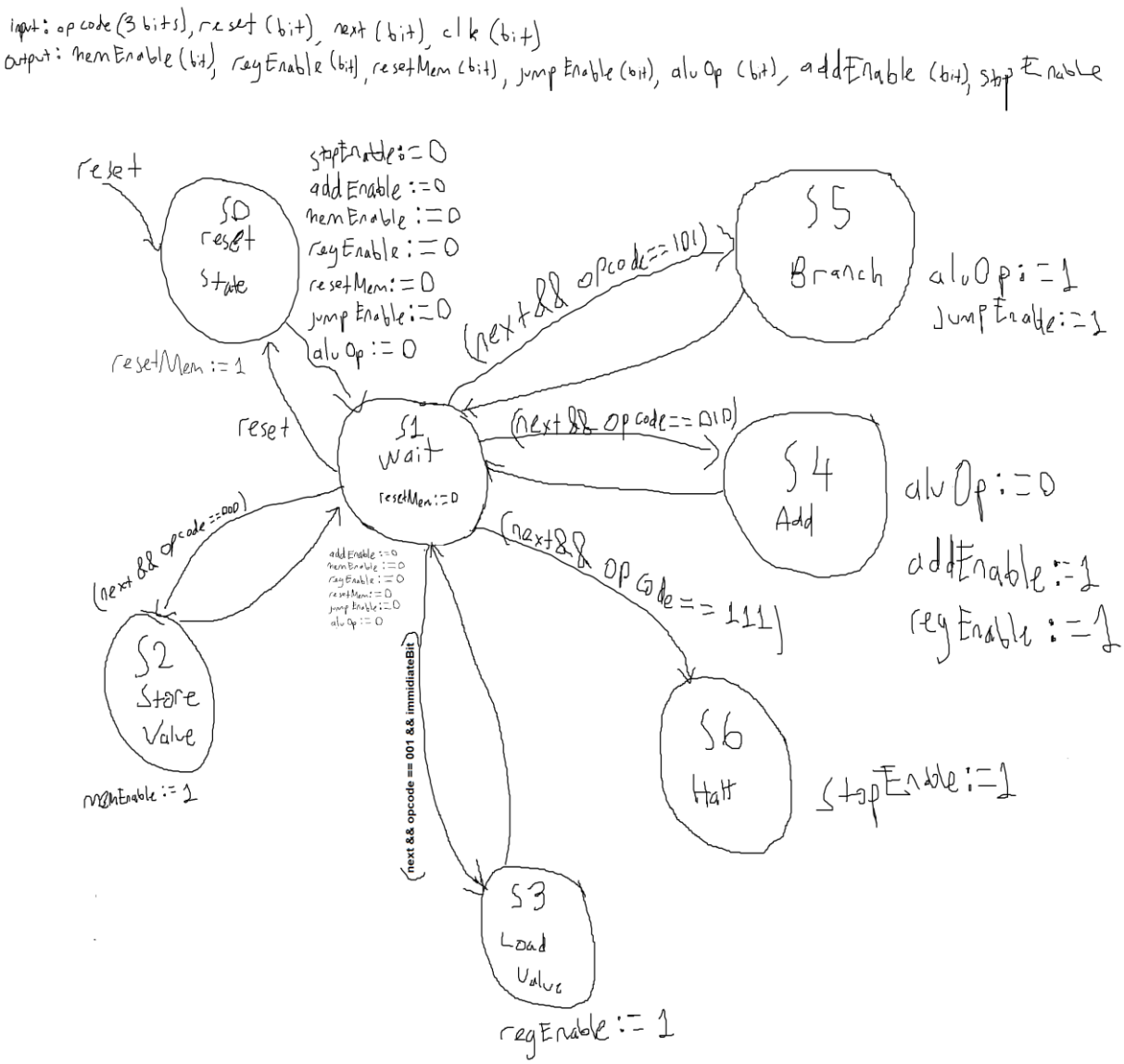
24.12.2020

1) Block diagram of controller/datapath and explanation:



This is the detailed schematic of the single cycle processor. Block Diagram of the Controller is drawn on the top right. It has three inputs which are opcode, immediateBit and equals where opcode is the [15:13] of the instruction, immidiateBit is the 12th bit of the instruction and equals determines whether it is going to generate the control input for branch if equals. The ouptus are the control inputs of the datapath. Datapath has a counter, an ALU, a ROM 32x16 Instruction Memory, a 16x8 RAM Register File and a 16x8 RAM Memory Module. The Instruction Memory holds instructions that multiplies two numbers in the Memory Module. Alu has two functions, it adds two values or checks if branch equals. All of these combined create our Single Cycle Processor.

2) Detailed state diagram of the controller and explanation:



The controller of the Single Cycle Processor is a FSM which produces the control signals of the other modules. It has 7 states which are reset state, wait state, store state, load state, add state, branch if equals state and the stop state. In the reset state, the reset signal is set to one and the other control signals are set to 0. In the wait state, control signals are set to zero. In the store state, it enables the memory modules writeEnable. In the load state, it enables the register files writeEnable. In the add state, alu opcode is generated accordingly, the registers files WriteEnable is set to one and addEnable control input is set to one. The branch state makes jumpEnable one and sets alu opcode accordingly. Lastly, the halt state is the stop execution step which stops the processor by setting the stopEnable control input to one.

3) SystemVerilog code of the Single Cycle Processor

singleCycleProcessor module:

```
module singleCycleProcessor(input logic clk, reset, nextInstruction, inputInstr,
output logic [4:0] outcnt,input logic [15:0] instrInMem, inputInstruction,
output logic memWriteEnable,output logic [3:0] addressMem,
output logic [7:0] writeMemData,input logic [7:0] readMemData);

   logic [15:0] instruction;
   logic equals, stopEnable,  data, regWriteEnable,addr, memToReg, jumpEnable, loadcnt;
   logic [1:0] aluOp;

   always_ff @( posedge clk, posedge reset) begin
      if ( reset)
         instruction <= 16'b011_0000000000000;
      else if ( ~stopEnable) begin
         if ( nextInstruction)
            instruction <= instrInMem;
         else if ( inputInstr)
            instruction <= inputInstruction;
         else
            instruction <= 16'b011_0000000000000;
      end
   end

   // Calling the controller module
   controller_module controller( clk, reset, nextInstruction, inputInstr, instruction[15:13],
instruction[12], equals, loadcnt,
                        addr, data, regWriteEnable, memWriteEnable, memToReg, jumpEnable,
stopEnable, aluOp);
   // Using the control outputs from the controller in the datapath
   datapath_module datapath( clk, reset, loadcnt, outcnt, instruction,
```

addr, data, regWriteEnable, memWriteEnable, memToReg, jumpEnable, stopEnable, aluOp,

equals, addressMem, writeMemData, readMemData);

endmodule

controller module:

```
module controller_module(input logic clk, reset, insMem, inputInstr,
input logic [2:0] opcode,input logic immediateBit, equals,output logic outcnt,
addr, data, regWriteEnable, memWriteEnable, memToReg, jumpEnable, stopEnable,output logic [1:0] aluOp);
    logic [8:0] controlOutputs;
    logic [8:0] tmpOut = 9'b0;
    assign {addr, data, regWriteEnable, memWriteEnable, memToReg, stopEnable, aluOp} = controlOutputs;


    always_comb
        case ( opcode)
            4'b010: controlOutputs = 9'b10100011; // add
            4'b101: controlOutputs = 9'b00000010; // branch if equals
            4'b111: controlOutputs = 9'b00000100; // stop (wait)
            default: case ( {opcode, immediateBit})
                4'b000_0: controlOutputs = 9'b00010000; // store from reg
                4'b000_1: controlOutputs = 9'b11010000; // store to mem
                4'b001_0: controlOutputs = 9'b00101000; // load from mem to reg
                4'b001_1: controlOutputs = 9'b11100000; // load to reg
                default: controlOutputs = 9'b00000000; // does nothing
            endcase
        endcase
    assign outcnt = insMem;
    assign jumpEnable = equals;
endmodule
```

datapath module:

```
module datapath_module(input logic clk, reset, loadProgramCount,
output logic [4:0] outputCount,input logic [15:0] instruction,input logic address, data,
regWriteEnable,
memWriteEnable, memToReg, jumpEnable, stopEnable,input logic [1:0] aluOp,output logic
equals,
output logic [3:0] addressMem,output logic [7:0] writeMemData,input logic [7:0]
readMemData);


    logic [4:0] countTmp;
    logic [7:0] regData, readReg1, readReg2;
    logic [7:0] sum;
    logic [3:0] regAddr;
    logic [31:0] rom;


    always_ff @( posedge clk, posedge reset)
        if (reset) outputCount <= 5'b0000;
        else if ( (loadProgramCount && ~stopEnable) || jumpEnable) outputCount <=
countTmp;


    assign countTmp = jumpEnable ? instruction[12:8] : (outputCount + 1);


    register_file registerfile( clk, reset, regWriteEnable, instruction[7:4], instruction[3:0],
regAddr, regData,
                readReg1, readReg2);
    assign regAddr = address ? instruction[11:8] : instruction[7:4];
    assign regData = data ? instruction[7:0] : ( memToReg ? readMemData : sum);



    alu ArithmeticLogicUnit( aluOp, readReg1, readReg2, sum, equals);
    assign addressMem = memWriteEnable ? (address ? instruction[11:8] : instruction[7:4]) :
instruction[3:0];
    assign writeMemData = data ? instruction[7:0] : readReg2;
```

endmodule

registerfile module:

```
module register_file(input logic clk, reset, writeEnable,
input logic [3:0] readAddress1, readAddress2, writeAddress, input logic [7:0] writeData,
output logic [7:0] readData1, readData2);


   logic [7:0] register [15:0]; // 16x8 register module
   // This module is the same as memory module but name register file??
   // initialize data to 0
   initial begin
      register[0] = 8'b0;
      register[1] = 8'b0;
      register[2] = 8'b0;
      register[3] = 8'b0;
      register[4] = 8'b0;
      register[5] = 8'b0;
      register[6] = 8'b0;
      register[7] = 8'b0;
      register[8] = 8'b0;
      register[9] = 8'b0;
      register[10] = 8'b0;
      register[11] = 8'b0;
      register[12] = 8'b0;
      register[13] = 8'b0;
      register[14] = 8'b0;
      register[15] = 8'b0;
   end


   always_ff @( posedge clk, posedge reset)
      if ( reset) begin
         for ( int i = 0; i < 16; i++) begin
```

```
            register[i] = 8'b0;
        end
    end
    else if ( writeEnable)
        register[writeAddress] <= writeData;


  assign readData1 = register[readAddress1];
  assign readData2 = register[readAddress2];
endmodule
module alu(input logic [1:0] aluOP, input logic [7:0] data1, data2, output logic [7:0] sum,
output logic equals);
  always_comb
    case ( aluOP)
      2'b10: // branch if data1 and data2 are equal
        begin
          if ( data1 == data2)
            equals = 1;
          sum = 8'b0;
        end
      2'b11: // add
        begin
          equals = 0;
          sum = data1 + data2;
        end
      default: begin equals = 0;  sum = 8'b0; end
    endcase
endmodule
```

instruction memory module:

```
module instruction_memory(input logic [4:0] address, output logic [15:0] instruction);
   // This is a Read Only Memory that does multipication which is taken from the document
   always_comb
         case ( address)
            5'b00000: instruction = 16'b001_0_0000_0000_0000;
            5'b00001: instruction = 16'b001_0_0000_0001_0001;
            5'b00010: instruction = 16'b001_1_0010_00000000;
            5'b00011: instruction = 16'b001_1_1111_00000000;
            5'b00100: instruction = 16'b001_1_0100_00000001;
            5'b00101: instruction = 16'b101_01001_0010_0001;
            5'b00110: instruction = 16'b010_0_1111_1111_0000;
            5'b00111: instruction = 16'b010_0_0010_0010_0100;
            5'b01000: instruction = 16'b101_00101_0000_0000;
            5'b01001: instruction = 16'b000_0_0000_1111_1111;
         default: instruction = 16'b111_0000000000000;
         endcase
endmodule
```

data memory module:

```
module dataMemory(input logic clk, reset, writeEnable, input logic [3:0] readAddress1,
readAddress2,
input logic [7:0] writeData,output logic [7:0] readData1, readData2);


   logic [7:0] memory [15:0]; // 16x8 memory module


   // initialize data to 0
   initial begin
      memory[0] = 8'b0;
      memory[1] = 8'b0;
      memory[2] = 8'b0;
      memory[3] = 8'b0;
```

```
        memory[4] = 8'b0;
        memory[5] = 8'b0;
        memory[6] = 8'b0;
        memory[7] = 8'b0;
        memory[8] = 8'b0;
        memory[9] = 8'b0;
        memory[10] = 8'b0;
        memory[11] = 8'b0;
        memory[12] = 8'b0;
        memory[13] = 8'b0;
        memory[14] = 8'b0;
        memory[15] = 8'b0;
    end

    always_ff @( posedge clk, posedge reset)
        if ( reset) begin
            for ( int i = 0; i < 16; i++) begin
                memory[i] = 8'b0;
            end
        end
        else if ( writeEnable)
            memory[readAddress1] <= writeData;

    assign readData1 = memory[readAddress1];
    assign readData2 = memory[readAddress2];
endmodule
```

seven segment display module:

```
module SevenSegmentDisplay(
    input clk,
    input [3:0] in3, in2, in1, in0,
```

```
output [6:0]seg, logic dp,
output [3:0] an
);

localparam N = 18;
logic [N-1:0] count = {N{1'b0}};
always@ (posedge clk)
    count <= count + 1;



logic [4:0]digit_val;
logic [3:0]digit_en;

always@ (*)
 begin
 digit_en = 4'b1111;
 digit_val = in0;

  case(count[N-1:N-2])

   2'b00 :  //select first 7Seg.
    begin
     digit_val = {1'b0, in0};
     digit_en = 4'b1110;
    end

   2'b01:  //select second 7Seg.
    begin
     digit_val = {1'b0, in1};
     digit_en = 4'b1101;
```

```
    end

   2'b10:  //select third 7Seg.
    begin
     digit_val = {1'b1, in2};
     digit_en = 4'b1011;
    end

   2'b11:  //select forth 7Seg.
    begin
     digit_val = {1'b0, in3};
     digit_en = 4'b0111;
    end
  endcase
 end


//Convert digit number to LED vector. LEDs are active low.
logic [6:0] sseg_LEDs;
always @(*)
 begin
  sseg_LEDs = 7'b1111111; //default
  case( digit_val)
   5'd0 : sseg_LEDs = 7'b1000000; //to display 0
   5'd1 : sseg_LEDs = 7'b1111001; //to display 1
   5'd2 : sseg_LEDs = 7'b0100100; //to display 2
   5'd3 : sseg_LEDs = 7'b0110000; //to display 3
   5'd4 : sseg_LEDs = 7'b0011001; //to display 4
   5'd5 : sseg_LEDs = 7'b0010010; //to display 5
   5'd6 : sseg_LEDs = 7'b0000010; //to display 6
```

```
      5'd7 : sseg_LEDs = 7'b1111000; //to display 7
      5'd8 : sseg_LEDs = 7'b0000000; //to display 8
      5'd9 : sseg_LEDs = 7'b0010000; //to display 9
      5'd10: sseg_LEDs = 7'b0001000; //to display a
      5'd11: sseg_LEDs = 7'b0000011; //to display b
      5'd12: sseg_LEDs = 7'b1000110; //to display c
      5'd13: sseg_LEDs = 7'b0100001; //to display d
      5'd14: sseg_LEDs = 7'b0000110; //to display e
      5'd15: sseg_LEDs = 7'b0001110; //to display f
      5'd16: sseg_LEDs = 7'b0110111; //to display "="
      default : sseg_LEDs = 7'b0111111; //dash
    endcase
  end

  assign an = digit_en;
  assign seg = sseg_LEDs;
  assign dp = 1'b1; //turn dp off

endmodule
```

debounce module:

```
module debouncer(input logic clk, input logic button,output logic pulse);
  // Taken from cs223 moodle
  logic [24:0] timer;
  typedef enum logic [1:0]{S0,S1,S2,S3} states;
  states state, nextState;
  logic gotInput;

  always_ff@(posedge clk)
    begin
      state <= nextState;
```

```
        if(gotInput)
            timer <= 25000000;
        else
            timer <= timer - 1;
    end
  always_comb
    case(state)
        S0: if(button)
            begin //startTimer
                nextState = S1;
                gotInput = 1;
            end
            else begin nextState = S0; gotInput = 0; end
        S1: begin nextState = S2; gotInput = 0; end
        S2: begin nextState = S3; gotInput = 0; end
        S3: begin if(timer == 0) nextState = S0; else nextState = S3; gotInput = 0; end
        default: begin nextState = S0; gotInput = 0; end
        endcase

  assign pulse = ( state == S1 );
endmodule
```

top module:

```
module topModule(input logic clk,input logic [4:0] buttons,input logic [15:0]
inputInstruction,
output logic [15:0] nextInstruction,output logic [6:0] seg,output logic dp,
output logic [3:0] an);

  logic [4:0] count;
  logic memWrite;
  logic [3:0] addressMem;
  logic [3:0] addressSevSeg = 4'b0;
```

```
    logic [7:0] writeMemData, readMemData, readSevSegData;
    logic switchInstruction, reset, prevData, nextData, nextInstr; // buttons

    // set the button inputs and outputs
    debouncer inputInstrBtn( clk,  buttons[4], switchInstruction); // center
    debouncer resetBtn( clk,  buttons[3], reset);          // up
    debouncer prevDataBtn( clk, buttons[2], prevData);     // left
    debouncer nextDataBtn( clk,  buttons[1], nextData);     // right
    debouncer nextInstrBtn( clk,  buttons[0], nextInstr);   // down

    always_ff @( posedge clk, posedge reset)
    begin
        if ( reset)
            addressSevSeg <= 4'b0;
        else if ( prevData)
            addressSevSeg <= addressSevSeg - 1;
        else if ( nextData)
            addressSevSeg <= addressSevSeg + 1;
    end

    // create processor and memories
    singleCycleProcessor processor( clk, reset, nextInstr, switchInstruction, count,
nextInstruction, inputInstruction, memWrite, addressMem, writeMemData, readMemData);
    instruction_memory instMem( count, nextInstruction);
    dataMemory dataMem( clk, reset, memWrite, addressMem, addressSevSeg,
writeMemData, readMemData, readSevSegData);
    SevenSegmentDisplay display( clk, addressSevSeg, 4'b0, readSevSegData[7:4],
readSevSegData[3:0], seg, dp, an);
endmodule
```

The processor can do the following function:

rf[15] = rf[rf[0]] ∗ rf[rf[1]]

Lets say that rf[0] is 4 rf[1] is 5 which two random numbers. Then we want to put the values that we want to multiply into rf[4] and rf[5]. Lets say that we want to multiply 2x3, then we do rf[4] = 2 and rf[5] = 3. After that, the values in rf[4] and rf[5] are loaded to d[0] and d[1]. Here is the the instruction set to do these:

001_1_0000_00000100 rf[0] = 4

001_1_0001_00000101 rf[1] = 5

001_1_0100_00000010 rf[4] = 2

001_1_0101_00000011 rf[5] = 3

000_0_0000_0000_0100 d[0] = rf[4] = 2

000_0_0000_0001_0101 d[1] = rf[5] = 3

The rest is the multiplication code in the lab document but it uses rf[15] and d[15] instead of rf[3] and data[3]. Here is the instruction set for this:

001_0_0000_0000_0000

001_0_0000_0001_0001

001_1_0010_00000000

001_1_1111_00000000

001_1_0100_00000001

101_01001_0010_0001

010_0_1111_1111_0000

010_0_0010_0010_0100

101_00101_0000_0000

000_0_0000_1111_1111

111_0000000000000 (this is the execution stop code)

After it is done, we see 6 in the F of the data memory which proves that it can do

rf[15] = rf[rf[0]] ∗ rf[rf[1]].