CS223: Digital Design

Section 01

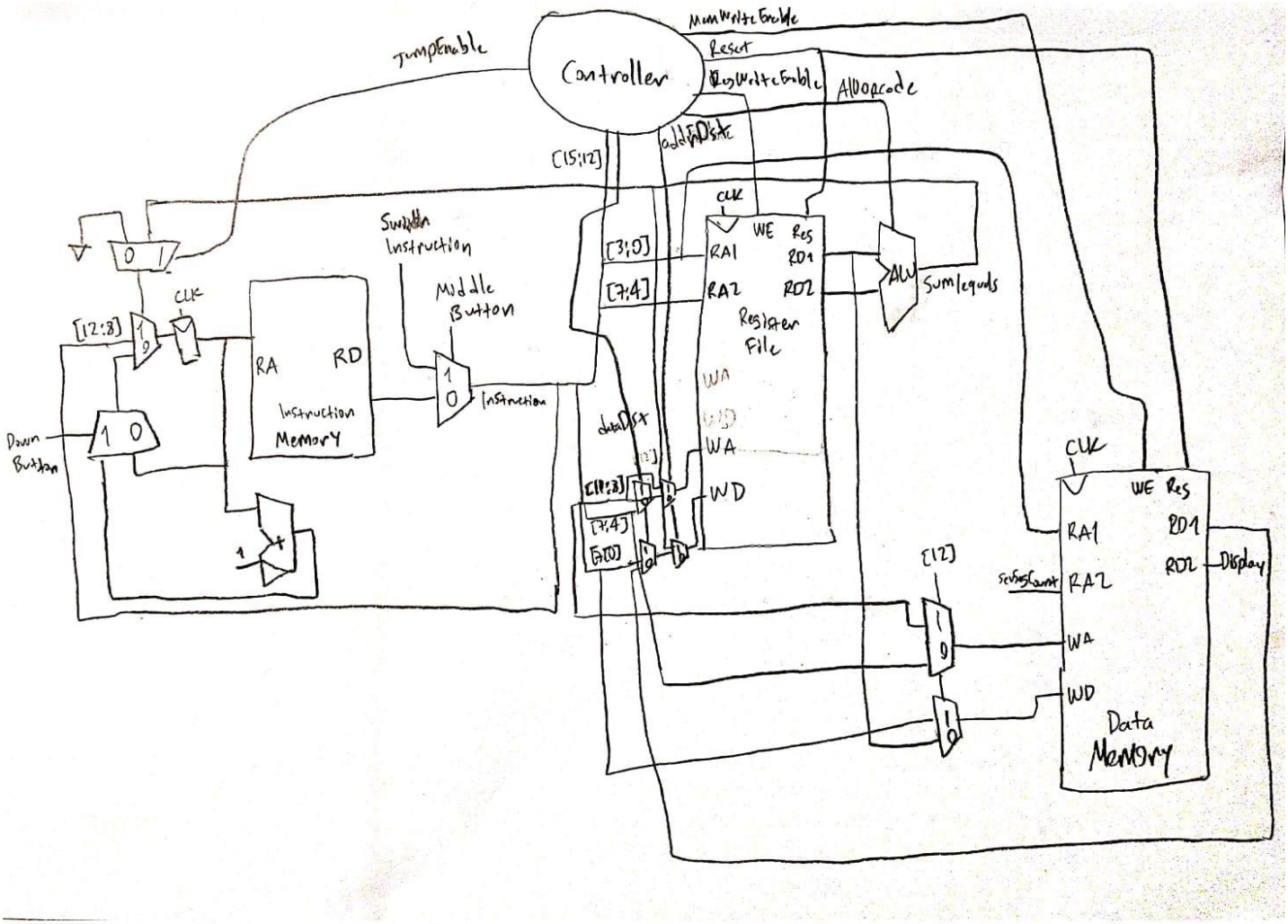
Project Report

Efe Beydoğan

21901548

25.12.2020

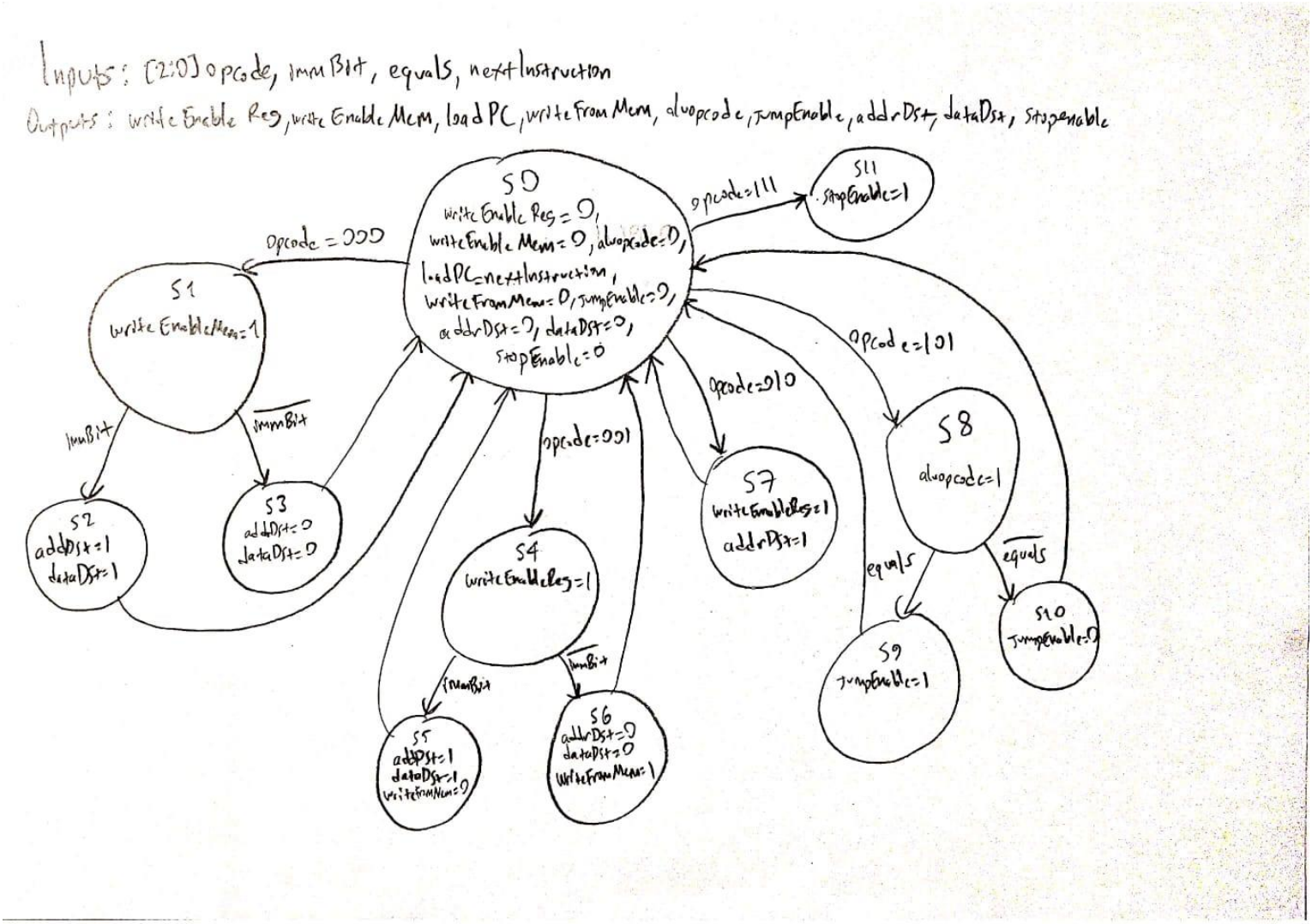
1) Block diagram of your controller/datapath and explanation:



CS Scanned with CamScanner

The datapath takes the control inputs from the controller and does the necessary functions accordingly. First, depending on the reset input the datapath resets the address counter to 0 and starts from the first instruction in the intstruction memory or depending on the stopping output and load inputs from controller, the datapath sets the new address to be taken from the instruction memory so the next instruction can be carried out. It looks at the jump signal to determine which address in the instruction memory to go to, it either increments the address counter by one or jumps to the specified address. Then, depending on signals from the controller the datapath decides which bits of the instruction (either [11:8] or [7:4]) to take as reading address for the register file. Also, depending on the signals it determines which data to write on the register file. Then, the datapath invokes an ALU module and passes the aluopcode to the ALU in order to either sum two numbers or see if the two numbers are equal for branch if equals operation. Finally, the datapath decides which addresses to use as reading addresses from the data memory and which data to write in the data memory.

2) Detailed state diagram of the controller and explanation:



Scanned with CamScanner

The controller takes care of the control inputs for the whole datapath and makes sure the required functions are carried out. First, the controller sets all of its outputs to “0” in the S0 state. Then, depending on the opcode received from the instructrion, it proceeds to other states. If the opcode is 000, it moves to S1 and sets the output necessary for being able to write into the data memory “1”, and then depending on the immediate bit, it changes addrDst and dataDst so in the datapath, the places to receive the address and data can be determined. If the opcode is 001, the controller makes writeEnableReg “1”, so the datapath can write in the register file. The controller also sets the necessary outputs so the datapath can determine where to take the data and address to write in the register file. If the opcode is 010, the controller sets writeEnableReg to 1 so the output of the addition can be written in the register file. If the opcode is 101, the controller makes the aluopcode “1”, which means the ALU will check to see if two numbers are equal and as a result, the controller will determine the jumpEnable output for branch if equals operation. Finally, if the opcode is 111, the controller goes into the waiting state and doesn’t accept any more inputs until it is reset.

3) You are required to write SystemVerilog code for:

- The implementation of the single-cycle processor instruction set given in Table 1:

Instruction memory:

```
module instructionMem( input [4:0] address, output logic [15:0] instruction);
   always_comb
      case ( address)
         5'b00000: instruction = 16'b000_0_0000_0000_0000; // rf[0] = d[0]
         5'b00001: instruction = 16'b000_0_0000_0001_0001; // rf[1] to d[1]
         5'b00010: instruction = 16'b000_0_0000_0000_0100; // rf[4] to d[0]
         5'b00011: instruction = 16'b000_0_0000_0001_0101; // rf[5] to d[1]
         5'b00000: instruction = 16'b001_0_0000_0000_0000; // load from dm[0] to rf[0]
         5'b00001: instruction = 16'b001_0_0000_0001_0001; // load from dm[1] to rf[1]
         5'b00010: instruction = 16'b001_1_0010_00000000; // load 0 to rf[2]
         5'b00011: instruction = 16'b001_1_1111_00000000; // load 0 to rf[15]
         5'b00100: instruction = 16'b001_1_0100_00000001; // load 1 to rf[4]
         5'b00101: instruction = 16'b101_01001_0010_0001; // branch to IM[9] if rf[2] == rf[1]
         5'b00110: instruction = 16'b010_0_1111_1111_0000; // Add RF[15] += RF[0]
         5'b00111: instruction = 16'b010_0_0010_0010_0100; // Add RF[2] += RF[4]
         5'b01000: instruction = 16'b101_00101_0000_0000; // Branch to IM[5] if RF[0] == RF[0]
         5'b01001: instruction = 16'b000_0_1111_1111_1111; // Store RF[15] to DM[15]
         default: instruction = 16'b111_0_0000_0000_0000;
      endcase
endmodule
```

Data Memory:

```
module dataMem( input logic clk, reset, writeEnable, input logic [7:0] writeData, input logic [3:0] writeAddress,
            input logic [3:0] readAddress1, readAddress2,
            output logic [7:0] readData1, readData2);
   logic [7:0] memory [15:0]; // ram to hold 16 8 bit numbers


   initial begin
      for ( int i = 0; i < 16; i++) begin
         memory[i] = 8'b0;
      end
   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) begin
         memory[writeAddress] <= writeData;
      end


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

Register File:

```
module registerFile( input logic clk, reset, writeEnable, input logic [3:0] writeAddress, input logic [7:0] writeData,
                     input logic [3:0] readAddress1, readAddress2,
                     output logic [7:0] readData1, readData2);
  logic [7:0] registers [15:0];


  initial begin
    for ( int i = 0; i < 16; i++) begin
      registers[i] = 8'b0;
    end
  end


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


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

ALU:

```
module alu( input logic aluOP, input logic [7:0] number1, number2, output logic [7:0] sum, output logic equals);
  always_comb
    case ( aluOP)
      1'b1: begin if ( number1 == number2) equals = 1; sum = 8'b0; end
      1'b0: begin equals = 0; sum = number1 + number2; end
      default: begin equals = 0; sum = 8'b0; end
    endcase
endmodule
```

Controller:

```
module controller( input logic [2:0] opcode, input logic immBit, equals, nextInstruction,
            output logic writeEnableReg, writeEnableMem, loadPC, writeFromMem,
            output logic aluopcode, jumpEnable,
            output logic addrDst, dataDst,
            output logic stopEnable);

  always_comb
    case ( opcode)
      3'b000: begin
        writeEnableMem = 1;
        writeEnableReg = 0;
        aluopcode = 0;
        jumpEnable = 0;
        stopEnable = 0;
        writeFromMem = 0;
        if ( immBit) begin
```

```
        addrDst = 1;
        dataDst = 1;
    end
    else begin
        addrDst = 0;
        dataDst = 0;
    end
end
3'b001: begin
    writeEnableMem = 0;
    writeEnableReg = 1;
    aluopcode = 0;
    jumpEnable = 0;
    stopEnable = 0;
    if ( immBit) begin
        addrDst = 1;
        dataDst = 1;
        writeFromMem = 0;
    end
    else begin
        addrDst = 0;
        dataDst = 0;
        writeFromMem = 1;
    end
end
3'b010: begin
    writeEnableReg = 1;
```

```
    aluopcode = 0;
    writeEnableMem = 0;
    jumpEnable = 0;
    addrDst = 1;
    dataDst = 0;
    stopEnable = 0;
    writeFromMem = 0;
end
3'b101: begin
    writeEnableReg = 0;
    aluopcode = 1;
    writeEnableMem = 0;
    writeFromMem = 0;
    if ( equals)
        jumpEnable = 1;
    else
        jumpEnable = 0;

    addrDst = 0;
    dataDst = 0;
    stopEnable = 0;
end
3'b111: begin
    writeEnableReg = 0;
    aluopcode = 0;
    writeEnableMem = 0;
    jumpEnable = 0;
```

```
            addrDst = 0;
            dataDst = 0;
            stopEnable = 1;
            writeFromMem = 0;
        end
        default: begin
            writeEnableReg = 0;
            aluopcode = 0;
            writeEnableMem = 0;
            jumpEnable = 0;
            addrDst = 0;
            dataDst = 0;
            stopEnable = 0;
            writeFromMem = 0;
        end
    endcase

  assign loadPC = nextInstruction;
endmodule
```

Datapath:

```
module datapath( logic clk, reset,
            input logic [15:0] instruction,
            input logic [7:0] memReadData,
            input logic writeEnableReg, writeEnableMem, aluopcode, jumpEnable, stopEnable, loadPC,
writeFromMem,
            input logic addrDst, dataDst,
            output logic equals,
```

```
        output logic [3:0] memAddress,
        output logic [7:0] memWriteData,
        output logic [4:0] insAddress,
        output logic [3:0] memWriteAddress);

  logic [4:0] nextAddress;
  logic [3:0] regAddr;
  logic [7:0] regWriteData, regRead1, regRead2;
  logic [7:0] sum;

  // next instruction
  always_ff @( posedge clk, posedge reset)
    if (reset)
      insAddress <= 5'b0;
    else if ( ( loadPC && ~stopEnable) || jumpEnable) insAddress <= nextAddress;

  assign nextAddress = jumpEnable ? instruction[12:8] : (insAddress + 1);

  // register file
  registerFile register( clk, reset, writeEnableReg, regAddr, regWriteData, instruction[7:4], instruction[3:0], regRead1, regRead2);

  assign regAddr = addrDst ? instruction[11:8] : instruction[7:4];
  assign regWriteData = dataDst ? instruction[7:0] : ( writeFromMem ? memReadData : sum);

  // ALU
  alu ALU( aluopcode, regRead1, regRead2, sum, equals);
```

```
    // memory
    assign memAddress = instruction[3:0];
    assign memWriteData = dataDst ? instruction[7:0] : regRead2;

    assign memWriteAddress = addrDst ?  instruction[11:8] : instruction[7:4];

endmodule
```

Processor:

```
module processor( input logic clk, reset, nextInstruction,
                  input logic [2:0] opcode,
                  input logic immBit,
                  input logic [15:0] instruction,
                  input logic [7:0] memReadData,
                  output logic [7:0] memWriteData,
                  output logic [3:0] memAddress,
                  output logic [4:0] insAddress,
                  output logic [3:0] memWriteAddress,
                  output logic stopEnable,
                  output logic writeEnableMem);

    logic equals, writeEnableReg, loadPC, writeFromMem, jumpEnable, addrDst, dataDst, aluopcode;
    controller controls( opcode, immBit, equals, nextInstruction,
                         writeEnableReg, writeEnableMem, loadPC, writeFromMem,
                         aluopcode, jumpEnable,
                         addrDst, dataDst,
                         stopEnable);
```

```
   datapath path( clk, reset,
                  instruction,
                  memReadData,
                  writeEnableReg, writeEnableMem, aluopcode, jumpEnable, stopEnable, loadPC, writeFromMem,
                  addrDst, dataDst,
                  equals,
                  memAddress,
                  memWriteData,
                  insAddress,
                  memWriteAddress);
endmodule
```

- Push button, switch and seven-segment display interfaces in order to interact with the processor:

```
module TopModule( input logic clk, input logic [4:0] buttons, input logic [15:0] switchInstruction, output logic [15:0] memInst,
                  output logic [6:0] seg, output logic dp, output logic [3:0] an);


   logic prevMem, nextMem, reset, executeSwitch, executeMem;
   logic [4:0] insAddress;
   logic [3:0] addressCounter, memAddress;
   logic [15:0] instruction;
   logic writeEnable;
   logic [7:0] memReadData, memWriteData, sevSegData;
   logic [3:0] memWriteAddress;
   logic stopEnable;
```

```
debouncer prev( clk, buttons[4], prevMem);
debouncer next( clk, buttons[3], nextMem);
debouncer res( clk, buttons[2], reset);
debouncer switch( clk, buttons[1], executeSwitch);
debouncer mem( clk, buttons[0], executeMem);



always_ff @( posedge clk, posedge reset)
begin
   if (reset) begin
      instruction <= 16'b110_0000000000000;
      addressCounter <= 0;
   end
   else if ( nextMem) begin
      addressCounter <= addressCounter + 1;
   end
   else if ( prevMem) begin
      addressCounter <= addressCounter - 1;
   end
   else if ( !stopEnable) begin
      if ( executeSwitch) begin
         instruction <= switchInstruction;
      end
      else if ( executeMem) begin
         instruction <= memInst;
      end
      else begin
```

```
            instruction <= 16'b110_0000000000000;
        end
    end
end

instructionMem instMem( insAddress, memInst);
processor process( clk, reset, executeMem, instruction[15:13], instruction[12], instruction, memReadData, memWriteData, memAddress,
                   insAddress, memWriteAddress, stopEnable, writeEnable);
dataMem datamemory( clk, reset, writeEnable, memWriteData, memWriteAddress, memAddress,
                  addressCounter, memReadData, sevSegData);
sevSegDisplay display( clk, addressCounter, 0, sevSegData[7:4], sevSegData[3:0], seg, dp, an);
endmodule
```

- For the following code, workout the code in single-cycle processor instruction set and show that it works in your processor (don't care overflows and operate with only unsigned data):

$$rf[15] = rf[rf[0]] * rf[rf[1]];$$

The instruction set for this operation:

rf[0] = 4 -> 001_1_0000_0000100

rf[1] = 5 -> 001_1_0001_00000101

rf[4] = 2 -> 001_1_0100_00000010

rf[5] = 3 -> 001_1_0101_00000011

to put rf[0] and rf[1] to data memory:

rf[0] to data memory -> 000_0_xxxx_0000_0000 d[0] = 4

rf[1] to data memory -> 000_0_xxxx_0001_0001 d[1] = 5

rf[4] to data memory -> 000_0_xxxx_0000_0100 d[0] = 2

rf[5] to data memory -> 000_0_xxxx_0001_0101 d[1] = 3

IM[0] Load value from DM[0] to RF[0] 001_0_0000_0000_0000

IM[1] Load value from DM[1] to RF[1] 001_0_0000_0001_0001

IM[2] Load 0 immediately to RF[2] 001_1_0010_00000000

IM[3] Load 0 immediately to RF[15] 001_1_1111_00000000

IM[4] Load 1 immediately to RF[4] 001_1_0100_00000001

IM[5] Branch to IM[9] if RF[2] == RF[1] 101_01001_0010_0001

IM[6] Add RF[15] += RF[0] 010_0_1111_1111_0000

IM[7] Add RF[2] += RF[4] 010_0_0010_0010_0100

IM[8] Branch to IM[5] if RF[0] == RF[0] 101_00101_0000_0000

IM[9] Store RF[15] to DM[15] 000_0_xxxx_1111_1111