CS223 Laboratory Project:

Single-cycle processor

N. ONUR VURAL, 21902330, SEC3

PROJECT REPORT

1) Block diagram of your controller/datapath and explanation. (also added as pdf)

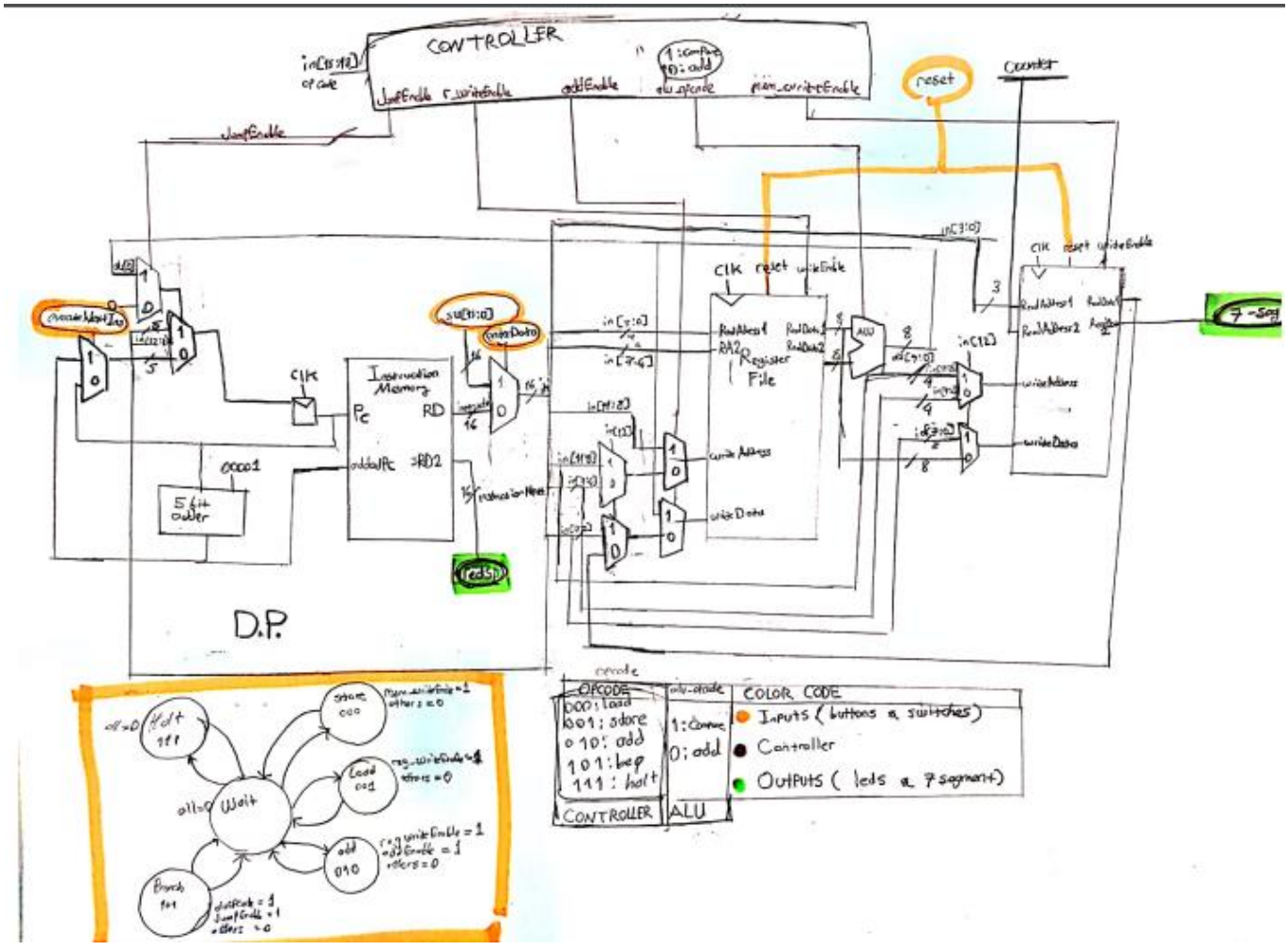

Figure 1: Datapath & Controller Combined Diagram

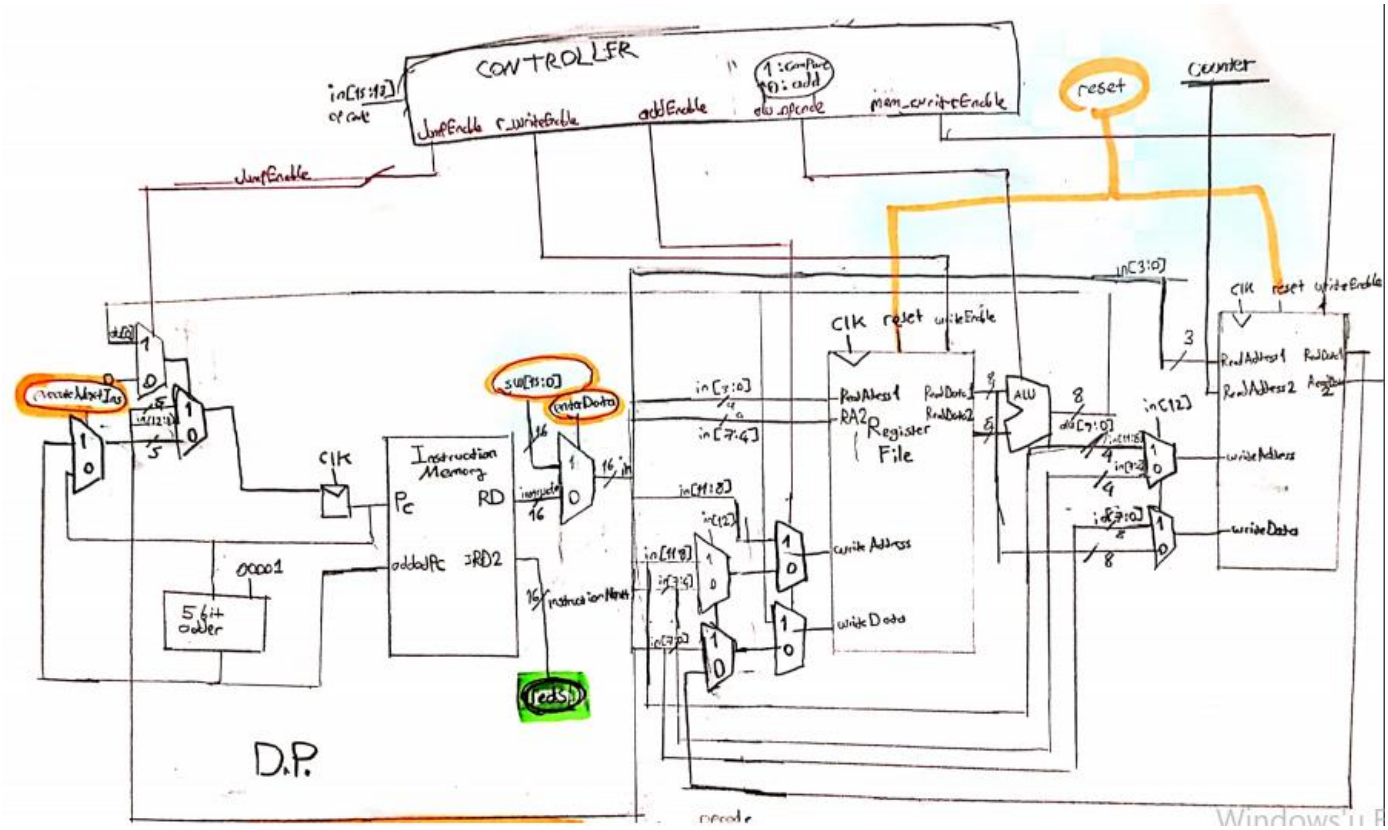

Figure 2: Datapath diagram and components

As seen on the diagram, the datapath consists of four important components, namely: Instruction Memory, Register File, ALU unit and Data Memory. As Instruction Memory shows the the characteristics of a read only memory it stores the instructions harcoded and thenat every rising edge of the clock sends the instruction corresponding to pc value where pc is incremented at every stage. As a next step the enter Data button combined with a 16 bit mux is essential to choose from input coming from the Instruction Memory and switches. Accordingly the instruction is transferred to required destinations and the controller is used to arrange the data flow. In this respect the system manages to send the data to leds and the seven segment display as output.

2) Detailed state diagram of the controller and explanation.

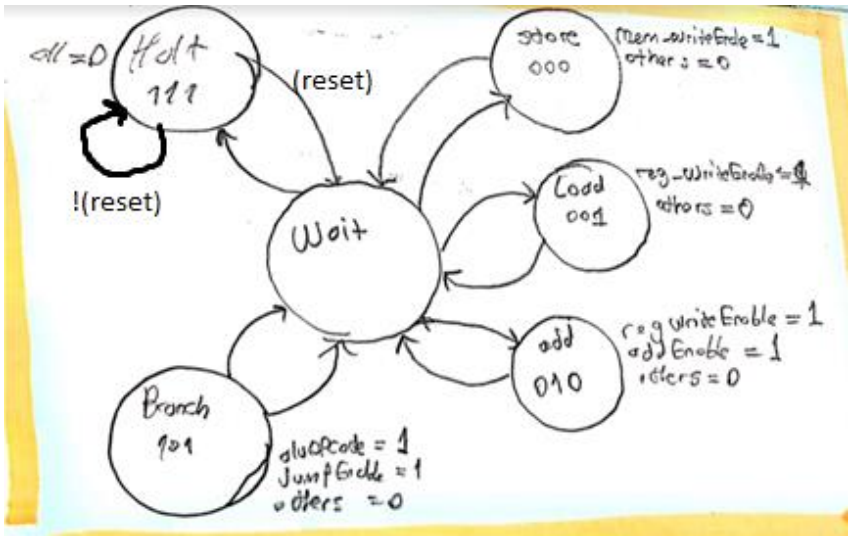

Figure 2: Controller state machine

The state machine uses the opcode, three most significant bits of the instruction , and according to the value it controls the enterence of data to the respective destination. The waiting state is important as after the process the system needs to zero all the signals. Otherwise the signals may stay open and the datapath may result in undesired outputs. As an example if add_Enable stays as 1 it will constantly add the values to the system until the button is pressed again ans therefore the seven segment will display a wrong value. Each state play a remarkable role to maket he datapath function properly. When we enter to store state, as it requires to write into Data Memory it sends the required signal (mem_writeEnable) to the mux and similarly when we enter to load state it makes it possible to write into register file (reg_writeEnable). The add state makes ALU make addition operation by sending add_Enable signal and sends the sum value back to the register file by making reg_writeEnable signal open. Branch state arranges aluOpcode to zero so that the ALU makes a comparison and decides whether two data from are equal or not. In order to increase the pc at desired level it enables jumpEnable signal and with this the last bit of alu_output is passes into pc value by the help of multiple muxes. Finally the halt state keeps at the same state until reset button is pressed again.

3) You are required to write SystemVerilog code for:

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

```
module controller(
 input logic [2:0] opcode,
 input logic reset, next_button, switchButton, clk,
 output logic mem_writeEnable, reg_writeEnable, jumpEnable, addEnable, alu_opcode
```

```
);

always_comb
begin
    if (next_button == 1'b1 | switchButton == 1'b1 ) begin
    case(opcode)
     3'b000:
     begin
     mem_writeEnable <= 1'b1;
     reg_writeEnable <= 1'b0;
     jumpEnable <= 1'b0;
     addEnable <= 1'b0;
     alu_opcode <= 1'b0;
     end
     3'b001:
     begin
     mem_writeEnable <= 1'b0;
     reg_writeEnable <= 1'b1;
     jumpEnable <= 1'b0;
     addEnable <= 1'b0;
     alu_opcode <= 1'b0;
     end
     3'b010:
     begin
     mem_writeEnable <= 1'b0;
     reg_writeEnable <= 1'b1;
     jumpEnable <= 1'b0;
     addEnable <= 1'b1;
     alu_opcode <= 1'b0;
     end
     3'b101:
```

```
  begin
  mem_writeEnable <= 1'b0;
  reg_writeEnable <= 1'b0;
  jumpEnable <= 1'b1;
  addEnable <= 1'b0;
  alu_opcode <= 1'b1;
  end
  3'b111:
  begin
  mem_writeEnable <= 1'b0;
  reg_writeEnable <= 1'b0;
  jumpEnable <= 1'b0;
  addEnable <= 1'b0;
  alu_opcode <= 1'b0;
  end
  default: begin
  mem_writeEnable <= 1'b0;
  reg_writeEnable <= 1'b0;
  jumpEnable <= 1'b0;
  addEnable <= 1'b0;
  alu_opcode <= 1'b0;
  end
  endcase
end
else if (reset)begin
mem_writeEnable <= 1'b0;
reg_writeEnable <= 1'b0;
jumpEnable <= 1'b0;
addEnable <= 1'b0;
alu_opcode <= 1'b0;
end
```

```
    else begin
      mem_writeEnable <= 1'b0;
      reg_writeEnable <= 1'b0;
      jumpEnable <= 1'b0;
      addEnable <= 1'b0;
      alu_opcode <= 1'b0;
    end
    end
endmodule
```

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

```
module user_Interface(
input logic clk, resetO, displayPrevO, displayNextO, enterDataO,  executeNextInstructionO,
input logic [15:0] sw,
output logic [15:0] nextInstruction,
output logic [6:0]seg, output logic dp, // just connect them to FPGA pins (individual LEDs).
output logic [3:0] an );



logic reset, displayPrev, displayNext, enterData,  executeNextInstruction; //input buttons to be updated


logic jumpContr;
//INTERNAL NODES AND FUNCTIONING DEVICES
//for instruciton
logic [15:0] instruction, instructionNext;
logic[15:0] in;

//controller
logic mem_writeEnable, reg_writeEnable, jumpEnable, addEnable, alu_opcode;
```

```
//regFile
logic [3:0] r_writeAddress1, r_writeAddress2;
logic [7:0] r_writeData1, r_writeData2;
logic [7:0] r_readData1, r_readData2;

//for mem
 logic [3:0] writeAddress, readAddress1, readAddress2;
 logic [7:0] writeData, readData1, readData2;

//ALU
logic [7:0] alu_output;

logic[4:0] pc, nextpc, addedpc;
logic[4:0] p_pc;



debounce db1( clk, resetO, reset);
debounce db2( clk, displayPrevO, displayPrev);
debounce db3( clk, displayNextO, displayNext);
debounce db4( clk, enterDataO, enterData);
debounce db5( clk, executeNextInstructionO, executeNextInstruction);

always_ff @ (posedge clk)
begin
 if (displayPrev)
 begin
    if (readAddress2 == 4'b0000)
       readAddress2 <= 4'b1111;
    else
       readAddress2 <= readAddress2 - 1;
```

```
 end
else if(displayNext)
begin
    if (readAddress2 ==  4'b1111)
      readAddress2 <= 4'b0000;
    else
      readAddress2 <= readAddress2 + 1;
 end

 if (reset)
 begin
   pc <= 5'b0;
   readAddress2 <= 4'b0;
 end
 else
 begin
 pc <= nextpc;
 end
 nextInstruction <= instructionNext;
// else if(enterData)
// begin
// nextInstruction <= instruction;
// end
end
//CONTROLLING
mux2_1 mPc1(alu_output[0], 1'b0, jumpEnable, jumpContr);
mux5bit_2_1 mPc2(in[12:8], p_pc, jumpContr, nextpc);
adder ad(pc, 5'b00001, addedpc);
mux5bit_2_1 mm(addedpc, pc, executeNextInstruction, p_pc);

//1)INITIALIZE CONTROLLER SYSTEM
```

```
controller cn( in[15:13], reset, executeNextInstruction, enterData, clk, mem_writeEnable,
reg_writeEnable, jumpEnable, addEnable, alu_opcode);

//2)INITIALIZE IM
instructionMemory ROM(pc, addedpc, instruction, instructionNext);

mux16bit_2_1 insMux(sw, instruction, enterData, in );

//3)INITIALIZE RF
mux4bit_2_1 r_adresMux1(in[11:8], in[7:4], in[12], r_writeAddress1 );
mux4bit_2_1 r_adresMux2(in[11:8], r_writeAddress1, addEnable, r_writeAddress2);

mux8bit_2_1 r_DataMux1(in[7:0], readData1, in[12], r_writeData1);
mux8bit_2_1 r_DataMux2(alu_output, r_writeData1, addEnable, r_writeData2);

registerFile rf(clk, reg_writeEnable, reset, r_writeAddress2, in[3:0] , in[7:4] , r_writeData2,
r_readData1, r_readData2  );

//ALU SETUP
ALU add_compare(r_readData1, r_readData2, alu_opcode, alu_output);

//4)INITIALIZE DM
mux4bit_2_1 dm_mux1(in[11:8], in[7:4], in[12], writeAddress);
mux8bit_2_1 dm_mux2(in[7:0], r_readData1, in[12], writeData);

dataMemory dM(clk, mem_writeEnable, reset, writeAddress, in[3:0], readAddress2, writeData,
readData1, readData2);

//OUTPUT ss-----
SevSeg_4digit sS(clk, readAddress2, 4'b0000, readData2[7:4], readData2[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):

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

// So if rf[0] = 4, rf[1] = 5, rf[4] = 2 rf[5] = 3, then rf[15] should be rf[4]*rf[5] = 6.

```
assign IM[0] = 16'b001_1_0000_00000100;

assign IM[1] = 16'b001_1_0001_00000101;

assign IM[2] = 16'b001_1_0100_00000010;

assign IM[3] = 16'b001_1_0101_00000011;

//1- You need to first put rf[0] and rf[1] to data memory and check the values on seven segment, where for this example you will see 4 and 5.

assign IM[4] = 16'b000_0_0000_0000_0000;

assign IM[5] = 16'b000_0_0000_0001_0001;

//2- Then you can store the values of register[4] and register[5] to data memory through switch instructions.

//--------------------SWITCH GÖSTERME ------------------------------------------------------

// 16'b000_0_xxxx_0100_0100; //store value of register[4]

// 16'b000_0_xxxx_0101_0101; //store value of register[5]

//--------------------SWITCH GÖSTERME ------------------------------------------------------

//3- You will have rf[4] and rf[5] values in data memory, and you can continue with the multiplication instructions in instruction memory.

assign IM[6] = 16'b001_1_0111_00000000; //Load 0 immediately to RF[7]

assign IM[7] = 16'b001_1_1000_00000000; //Load 0 immediately to RF[8]

assign IM[8] = 16'b001_1_1001_00000001; //Load 1 immediately to RF[9]


assign IM[9] = 16'b101_01101_0101_0111; //Branch to IM[13] if RF[5] == RF[7]

assign IM[10] = 16'b010_0_1000_1000_0100; //Add RF[8] += RF[4]

assign IM[11] = 16'b010_0_0111_0111_1001; //Add RF[7] += RF[9]


assign IM[12] = 16'b101_01001_0100_0100; //Branch to IM[7] if RF[4] == RF[4]


assign IM[13] = 16'b000_0_0000_1111_1000; //Store RF[8] to DM[15]
```