CS223 - Laboratory Project

Single-Cycle Processor

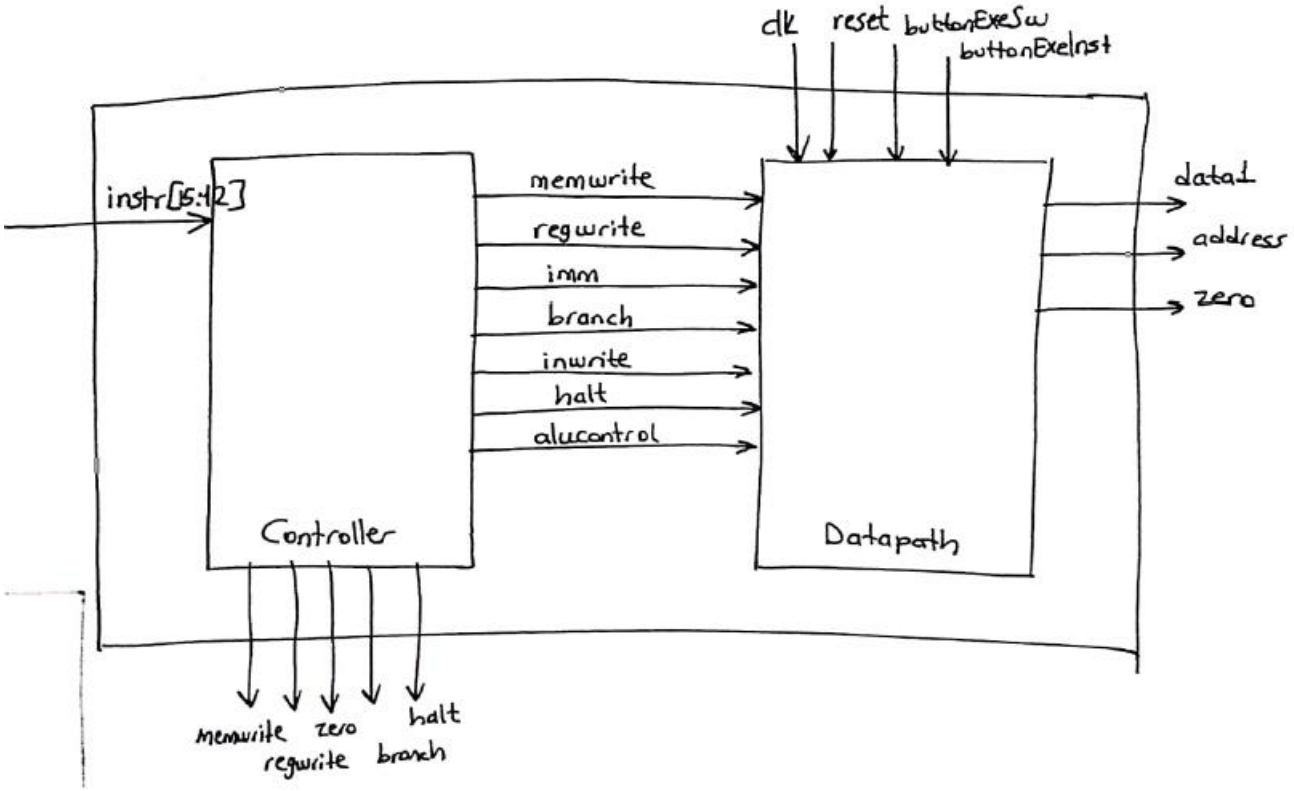
Ahmed Salih Cezayir

21802918

Section-4

25/12/2020

Blok Diagram of Controller/Datapath

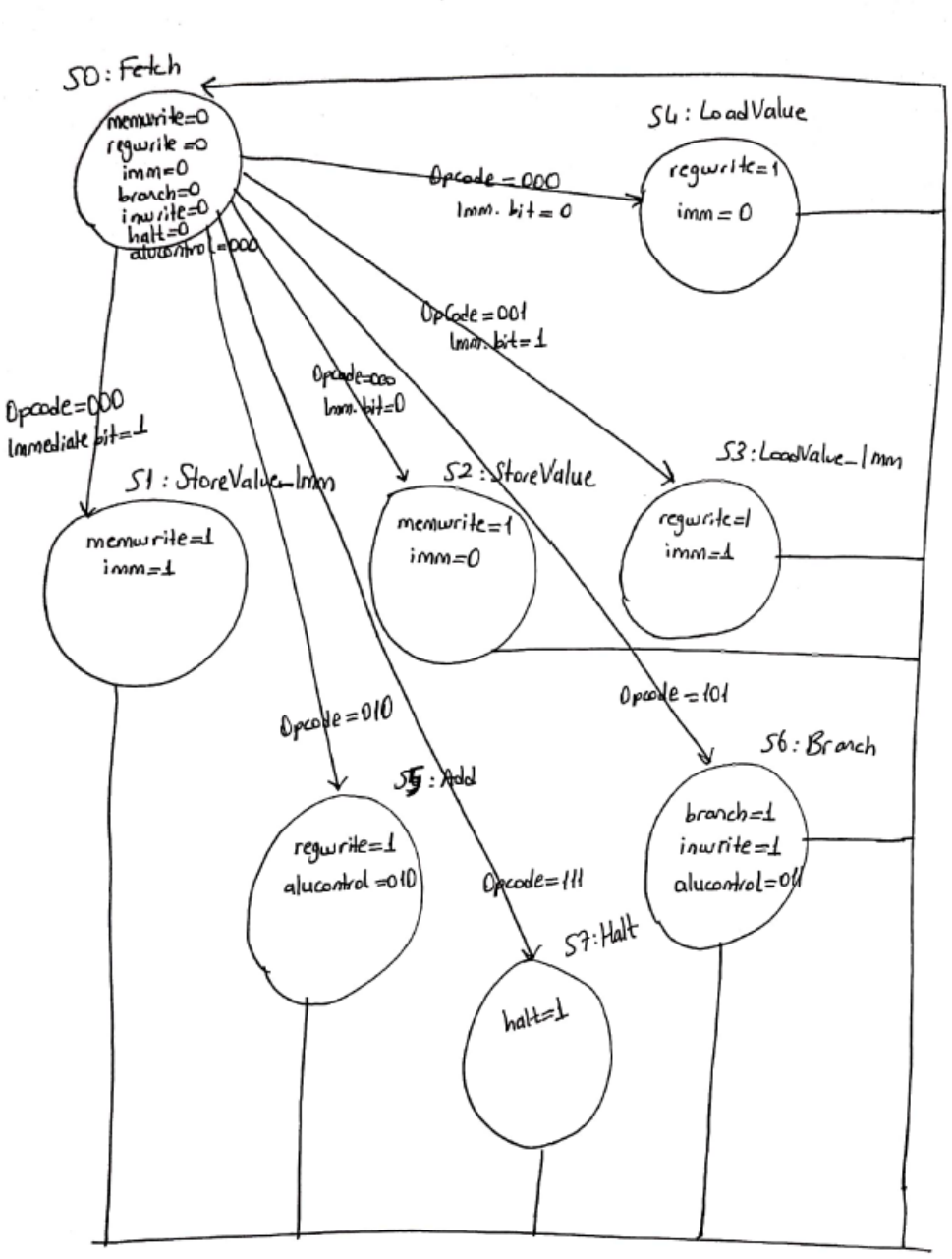


For input, controller only takes opcode and immediate bit part of the instruction. Then by using decoders, controller produces outputs. If the immediate bit of the opcode is 1, then controller sets imm to 1, If the immediate bit of the opcode is 0, it sets imm to 0. If the opcode is 000, the controller sets the memwrite to 1. If it is 001, the controller sets the regwrite to 1. If the opcode is 010, the controller sets the alucontrol to 010, regwrite to 1. If the opcode is 101, then the controller sets branch and inwrite to 1. If the opcode is 111, the controller sets halt to 1. Then all of these outputs are sent to datapath. Alucontrol parameter is used in the ALU located in the datapath, whereas regwrite is used as writeEnable parameter for register. Memwrite parameter is sent to topModule to be used as writeEnable for data memory.

In the datapath, there is a register file, an ALU and several 2:1 multiplexer. These components are used to create outputs such as address, data1 and zero. The data1

parameter is used when user is loading data to memory from a register. This data1 represents the data coming from register file. The address parameter also holds the address where the memory data should be stored. The output zero comes from the ALU and it is 1 if the result of the operation in the ALU is 0. It is used in the topModule to branch and jump in instruction module.

State Diagram of the Controller



At the start, the system is in the Fetch state where all the outputs are 0. Then depending on the opcode, system changes its state. If the opcode is 000 and immediate bit is 1, system goes to S1 state where it stores value in memory. In this state memwrite and imm is equal to 1. If the opcode is 000 and immediate bit is 0, system goes to S2 state where it stores value in memory by taking data from register. In this state memwrite is equal to 1, whereas imm is 0. If the opcode is 001 and the immediate bit is 1, system goes to S3 state where it loads data to registers. In this state both regwrite and imm is 1. If the opcode is 001 and the immediate bit is 0, system goes to S4 state where it loads data to registers by taking it from data memory. In this state regwrite is 1 while imm is 0. If opcode is 010 the system goes to S5 state. In this state, two data are taken from two registers and they are added and stored in another register. In this state, regwrite is 1 and alucode is 010. This alucode is for addition. If the opcode is 101, system goes to S6 where it compares 2 data from 2 registers. If both data are equal, a jump occur in the instruction memory. In this state, branch and inwrite are 1 while alucontrol is 011. This alucontrol is used for subtraction. After each of these steps end, they go back to S0.

SystemVerilog Codes for Single Cycle Processor

SevenSegmentDisplay

```
module SevSeg_4digit(
 input clk,
 input [3:0] in3, in2, in1, in0, //user inputs for each digit (hexadecimal value)
 output [6:0]seg, logic dp, // just connect them to FPGA pins (individual LEDs).
 output [3:0] an   // just connect them to FPGA pins (enable vector for 4 digits active low)
 );
// divide system clock (100Mhz for Basys3) by 2^N using a counter, which allows us to multiplex at lower speed
localparam N = 18;
logic [N-1:0] count = {N{1'b0}}; //initial value
```

```
always@ (posedge clk)
            count <= count + 1;
logic [4:0]digit_val; // 7-bit register to hold the current data on output
logic [3:0]digit_en;  //register for the 4 bit enable
always@ (*)
 begin
 digit_en = 4'b1111; //default
 digit_val = in0; //default
  case(count[N-1:N-2]) //using only the 2 MSB's of the counter

   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
```

Debouncer

```
module debounce(input logic clk, input logic button,output logic pulse );


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
```

TopModule

```
module top(input logic clk,
        input logic [4:0] buttons,
        input logic [15:0] switch,
        output logic [6:0] seg,
        output logic dp,
        output logic [3:0] an,
        output logic [15:0] inst);


   logic memwrite, regwrite, writeEnable, zero, branch, halt, writeEnable2;
   logic [3:0] address;
   logic [7:0] data1;
   logic [3:0] readAddress2 = 4'b0000;
   logic [5:0] curr = 5'b0;
   logic [7:0] readData1_M;
   logic [7:0] readData2_M;
   logic [15:0] finalInst;
   logic buttonPrev, buttonNext, buttonReset, buttonExeSw, buttonExeInst, buttonReset2;


   debounce b4(clk, buttons[4], buttonExeSw);
   debounce b3(clk, buttons[3], buttonReset);
   debounce b2(clk, buttons[2], buttonPrev);
   debounce b1(clk, buttons[1], buttonExeInst);
   debounce b0(clk, buttons[0], buttonNext);


   mux2#(16) instmux(switch, inst, buttonExeSw, finalInst);
   AND3 andGate1(memwrite, buttonExeSw, buttonExeInst, writeEnable);
   mux2 haltmux(0, writeEnable, halt, writeEnable2);
   mux2 haltmux2(0, buttonReset, halt, buttonReset2);


   datamem mem(clk, writeEnable2, buttonReset2, finalInst[3:0], readAddress2, address, data1, readData1_M,
readData2_M);
   mips mips(clk, buttonReset2, buttonExeSw, buttonExeInst, finalInst, readData1_M, data1, memwrite, regwrite, address,
zero, branch, halt);
   instmem im(curr, inst, nextInst);


   SevSeg_4digit sevSeg(clk, readAddress2, 4'b1111, readData2_M[7:4], readData2_M[3:0], seg, dp, an);
```

```
    always_ff@(posedge clk)
    begin
        if (buttonPrev)
            readAddress2 <= readAddress2 - 1;
        else if (buttonNext)
            readAddress2 <= readAddress2 + 1;
        else if (buttonExeInst == 1 && branch != 1)
                curr <= curr + 1;

        if (branch == 1 && zero == 1)
            if (buttonExeSw)
                curr <= inst[12:8];
            else if (buttonExeInst)
                curr <= inst[12:8];

        if (branch == 1 && zero == 0)
            if (buttonExeSw)
                curr <= curr + 1;
            else if (buttonExeInst)
                curr <= curr + 1;
    end
endmodule
```

Mips

```
module mips(input logic clk, reset, buttonExeSw, buttonExeInst,
            input logic [15:0] instr,
            input logic [7:0] readData1_M,
            output logic [7:0] data1,
            output logic memwrite, regwrite,
            output logic [3:0] address,
            output logic zero, branch, halt);
    logic imm, inwrite, needalu;
    logic [2:0] alucontrol;
    controller c(instr[15:12], memwrite, regwrite, imm, branch, inwrite, halt, needalu, alucontrol);
    datapath d(clk, reset, buttonExeSw, buttonExeInst, memwrite, regwrite, imm, branch, inwrite, halt, needalu,
readData1_M, instr, alucontrol, data1, address, zero);
```

```
endmodule
```

Memory Data

```
module datamem(input logic clk, writeEnable, reset,
               input logic [3:0] readAddress1, readAddress2, writeAddress,
               input logic [7:0] writeData,
               output logic [7:0] readData1, readData2);
  logic [7:0] dMem [15:0] = '{8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0};
  always_ff@(posedge clk)
    if(writeEnable)
      dMem[writeAddress] <= writeData;
    else if(reset)
      begin
        for (int i = 0; i < 16; i++)
          dMem[i][7:0] <= 8'b00000000;
      end
  assign readData1 = dMem[readAddress1];
  assign readData2 = dMem[readAddress2];
endmodule
```

Instruction Memory

```
module instmem(input logic [31:0] address,
               output logic [15:0] readData,
               output logic [15:0] nextReadData);
  logic [15:0] mem [31:0];
  assign mem[0] = 16'b001_0_0000_0000_0000;
  assign mem[1] = 16'b001_0_0000_0001_0001;
  assign mem[2] = 16'b001_1_0010_00000000;
  assign mem[3] = 16'b001_1_0011_00000000;
  assign mem[4] = 16'b001_1_0100_00000001;
  assign mem[5] = 16'b101_01001_0010_0001;
  assign mem[6] = 16'b010_0_0011_0011_0000;
  assign mem[7] = 16'b010_0_0010_0010_0100;
  assign mem[8] = 16'b101_00101_0000_0000;
```

```
    assign mem[9] = 16'b000_0_0000_0011_0011;
    assign mem[10] = 16'b0;
    assign mem[11] = 16'b0;
    assign mem[12] = 16'b0;
    assign mem[13] = 16'b0;
    assign mem[14] = 16'b0;
    assign mem[15] = 16'b0;
    assign mem[16] = 16'b0;
    assign mem[17] = 16'b0;
    assign mem[18] = 16'b0;
    assign mem[19] = 16'b0;
    assign mem[20] = 16'b0;
    assign mem[21] = 16'b0;
    assign mem[22] = 16'b0;
    assign mem[23] = 16'b0;
    assign mem[24] = 16'b0;
    assign mem[25] = 16'b0;
    assign mem[26] = 16'b0;
    assign mem[27] = 16'b0;
    assign mem[28] = 16'b0;
    assign mem[29] = 16'b0;
    assign mem[30] = 16'b0;
    assign mem[31] = 16'b0;

    assign readData = mem[address];
    assign nextReadData = mem[address + 1];
endmodule
```

Datapath

```
module datapath(input logic clk, reset, buttonExeSw, buttonExeInst,
                input logic memwrite, regwrite, imm, branch, inwrite, halt, needalu,
                input logic [7:0] readData1_M,
                input logic [15:0] instr,
                input logic [2:0] alucontrol,
                output logic [7:0] data1,
                output logic [3:0] address,
```

```
            output logic zero);

    logic [3:0] readAdd1_R, readAdd2_R, readAdd1_M, readAdd2_M, addressFinal;
    logic [7:0] readData1_R, readData2_R, readData2_M, data2, data, result;
    logic writeEnable2, writeEnable3;

    mux2#(4) addressmux(instr[11:8], instr[7:4], imm, address);
    mux2 datamux1(instr[7:0], readData1_R, imm, data1);
    mux2 datamux2(instr[7:0], readData1_M, imm, data2);
    mux2 datamux3(result, data2, needalu, data);
    mux2 datamux4(instr[11:8], address, needalu, addressFinal);
    mux2 haltmux(0, writeEnable2, halt, writeEnable3);
    AND3 andGate2(regwrite, buttonExeSw, buttonExeInst, writeEnable2);

    regfile rf(clk, writeEnable3, reset, instr[3:0], instr[7:4], addressFinal, data, readData1_R, readData2_R);
    alu alu1(readData1_R, readData2_R, alucontrol, result, zero);
endmodule
```

Controller

```
module controller(input logic [3:0] opcode,
                  output logic memwrite, regwrite, imm, branch, inwrite, halt,
                  output logic needalu,
                  output logic[2:0] alucontrol);

    logic [1:0] aluop;

    maindec md(opcode, memwrite, regwrite, imm, branch, inwrite, halt, needalu, aluop);
    aludec ad(aluop, needalu, alucontrol);

endmodule
```

Main Decoder

```
module maindec(input logic [3:0] op,
               output logic memwrite, regwrite, imm, branch, inwrite, halt,
               output logic needalu,
               output logic[1:0] aluop);
```

```
  logic [8:0] controls;

  assign {memwrite, regwrite, imm, branch, inwrite, halt, needalu, aluop} = controls;
  always_comb
    case(op)
      4'b0000: controls <= 9'b100000000; // Store value 0
      4'b0001: controls <= 9'b101000000; // Store value 1
      4'b0010: controls <= 9'b010000000; //Load value 0
      4'b0011: controls <= 9'b011000000; //Load value 1
      4'b0100: controls <= 9'b010000100; //Add
      4'b0101: controls <= 9'b010000100; //Add
      4'b1010: controls <= 9'b000110101; //Branch
      4'b1011: controls <= 9'b000110101; //Branch
      4'b1110: controls <= 9'b000001000; //Halt
      4'b1111: controls <= 9'b000001000; //Halt
      default: controls <= 9'bxxxxxxxx; //Illegal op
    endcase
endmodule
```

Alu Decoder

```
module aludec(input logic [1:0] aluop,
              input logic needalu,
              output logic [2:0] alucontrol);
  always_comb
    if (needalu == 0)
      alucontrol = 3'bxxx;
    else
    begin
      case(aluop)
        2'b00: alucontrol = 3'b010;
        2'b01: alucontrol = 3'b011;
        default: alucontrol = 3'bxxx;
      endcase
    end
endmodule
```

Register File

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

  logic [7:0] rFile [15:0] = '{8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0,8'b0};

  always_ff@(posedge clk)
    if(writeEnable)
      rFile[writeAddress] <= writeData;
    else if (reset)
      begin
        for (int i = 0; i < 16; i++)
          rFile[i][7:0] <= 8'b00000000;
      end

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

Mux2:1

```
module mux2#(parameter WIDTH = 8)
        (input logic [WIDTH-1:0] d1, d0,
         input logic s,
         output logic [WIDTH-1:0]y);

  assign y = s ? d1 : d0;
endmodule
```

AND3

```
module AND3(input logic a, b, c,
        output logic result);

  assign result = (a & b) | (a & c);
```

```
endmodule
```

ALU

```
module alu(input logic [3:0] a, b,
           input logic [2:0] aluControl,
           output logic [7:0] result,
           output logic zero);
  always_comb
  begin
    case(aluControl)
      3'b000: result = a & b; // A AND B
      3'b001: result = a | b; // A OR B
      3'b010: result = a + b; // A + B
      3'b011: result = a - b; // A - B
      3'b100: result = a < b;
      default: result = 8'bxxxx;
    endcase
  end
  assign zero = (result == 8'b0);
endmodule
```