22103277
Uygar Aras
CS 224-03
Lab 4 Preliminary Report

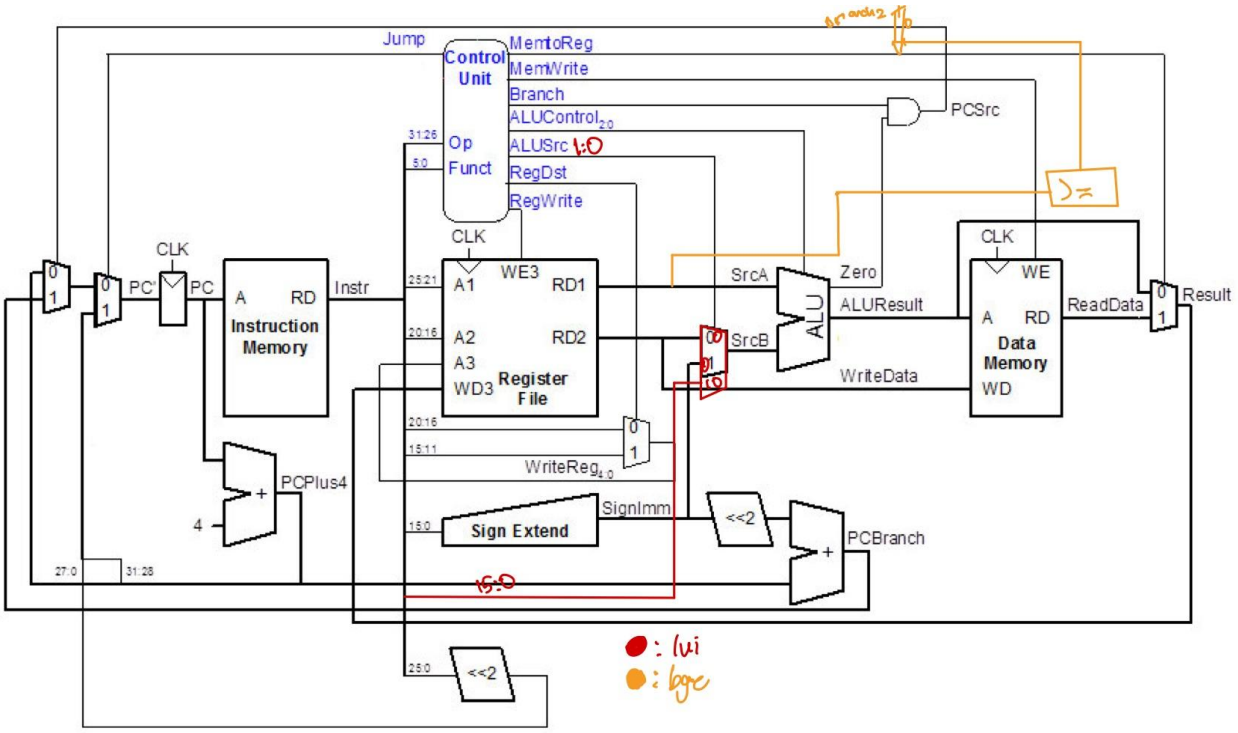
A. Hex Instructions and their equivalents

```
8'h00: instr = 32'h20020005;          //  ADDI $v0 $zero 0x0005
8'h04: instr = 32'h2003000c;          //  ADDI $v1 $zero 0x000C
8'h08: instr = 32'h2067fff7;    //  ADDI $a3 $v1 0xFFF7
8'h0c: instr = 32'h00e22025;          //  OR $a0 $a3 $v0
8'h10: instr = 32'h00642824;    //  AND $a1 $v1 $a0
8'h14: instr = 32'h00a42820;    //  ADD $a1 $a1 $a0
8'h18: instr = 32'h10a7000a;    //  BEQ $a1 $a3 0x000A
8'h1c: instr = 32'h0064202a;    //  SLT $a0 $v1 $a0
8'h20: instr = 32'h10800001;    //  BEQ $a0 $zero 0x0001
8'h24: instr = 32'h20050000;    //  ADDI $a1 $zero 0x0000
8'h28: instr = 32'h00e2202a;    //  SLT $a0 $a3 $v0
8'h2c: instr = 32'h00853820;    //  ADD $a3 $a0 $a1
8'h30: instr = 32'h00e23822;    //  SUB $a3 $a3 $v0
8'h34: instr = 32'hac670044;    //  SW $a3 0x0044 $v1
8'h38: instr = 32'h8c020050;    //  LW $v0 0x0050 $zero
8'h3c: instr = 32'h08000011;    //  J 0x0000011
8'h40: instr = 32'h20020001;    //  ADDI $v0 $zero 0x0001
8'h44: instr = 32'hac020054;    //  SW $v0 0x0054 $zero
8'h48: instr = 32'h08000012;  //  J 0x0000012
```

B. RTL Expression Set For new Instructions

bge:
IM[PC]
PC = PC + 4
R[rt] = immediate << 16
lui:
IM[PC]
PC = PC + 4
if (R[rs] >= R[rt]) then PC = PC + (offset << 2)

C. New Datapath



D. Alu Decoder Table

Instruction	Opcode (6 bits)	Funct (6 bits, for R-type)	ALU Operation (Control Lines)
ADD (R-type)	000000	100000	0010 (Add)
SUB (R-type)	000000	100010	0110 (Subtract)
AND (R-type)	000000	100100	0000 (And)
OR (R-type)	000000	100101	0001 (Or)
LUI (I-type)	001111	N/A	0011 (Load Upper Immediate)
BGE	000001	N/A	0110 (Subtract, for condition check)

E. Mips Model

```
main:
    # Test lui instruction
    lui $t0, 0x1234       # Load upper immediate
    sw $t0, result        # Store result
    lw $t1, result        # Load result for verification
    # Test bge instruction
    li $t2, 10
    li $t3, 5
    bge $t2, $t3, label1  # Branch if $t2 >= $t3
    li $t4, 1
    j endTest
label1:
    li $t4, 0             # Set success flag
endTest:
```

```
    add $t5, $t2, $t3     # Test add
    sub $t6, $t2, $t3     # Test sub
    and $t7, $t2, $t3     # Test and
    or $t8, $t2, $t3      # Test or
    li $v0, 10
    syscall
```

F. System Verilog Model

```
`timescale 1ns / 1ps
// Top level system including MIPS and memories


module top  (input   logic         clk, reset,
             output  logic[31:0] writedata, dataadr,
             output  logic[31:0] pc, instr, readdata,
             output  logic       memwrite);

  // instantiate processor and memories
  mips mips (clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
  imem imem (pc[7:2], instr);
  dmem dmem (clk, memwrite, dataadr, writedata, readdata);

endmodule



// External data memory used by MIPS single-cycle processor

module dmem (input  logic        clk, we,
             input  logic[31:0]  a, wd,
             output logic[31:0]  rd);

  logic  [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]];    // word-aligned  read (for lw)

  always_ff @(posedge clk)
    if (we)
      RAM[a[31:2]] <= wd;      // word-aligned write (for sw)

endmodule



// External instruction memory used by MIPS single-cycle
// processor. It models instruction memory as a stored-program
// ROM, with address as input, and instruction as output
```

```
module imem ( input logic [5:0] addr, output logic [31:0] instr);

// imem is modeled as a lookup table, a stored-program byte-addressable ROM
        always_comb
          case ({addr,2'b00})                   // word-aligned fetch
//                address                   instruction
//                -------             -----------
          8'h00: instr = 32'h20020005;          //  ADDI $v0 $zero 0x0005
          8'h04: instr = 32'h2003000c;          //  ADDI $v1 $zero 0x000C
          8'h08: instr = 32'h2067fff7;     //  ADDI $a3 $v1 0xFFF7
          8'h0c: instr = 32'h00e22025;          //  OR $a0 $a3 $v0
          8'h10: instr = 32'h00642824;    //  AND $a1 $v1 $a0
          8'h14: instr = 32'h00a42820;    //  ADD $a1 $a1 $a0
          8'h18: instr = 32'h10a7000a;    //  BEQ $a1 $a3 0x000A
          8'h1c: instr = 32'h0064202a;    //  SLT $a0 $v1 $a0
          8'h20: instr = 32'h10800001;    //  BEQ $a0 $zero 0x0001
          8'h24: instr = 32'h20050000;    //  ADDI $a1 $zero 0x0000
          8'h28: instr = 32'h00e2202a;    //  SLT $a0 $a3 $v0
          8'h2c: instr = 32'h00853820;    //  ADD $a3 $a0 $a1
          8'h30: instr = 32'h00e23822;    //  SUB $a3 $a3 $v0
          8'h34: instr = 32'hac670044;    //  SW $a3 0x0044 $v1
          8'h38: instr = 32'h8c020050;    //  LW $v0 0x0050 $zero
          8'h3c: instr = 32'h08000011;    //  J 0x0000011
          8'h40: instr = 32'h20020001;    //  ADDI $v0 $zero 0x0001
          8'h44: instr = 32'hac020054;    //  SW $v0 0x0054 $zero
          8'h48: instr = 32'h08000012;  //  J 0x0000012
              default:  instr = {32{1'bx}}; // unknown address
            endcase
endmodule


// single-cycle MIPS processor, with controller and datapath

module mips (input  logic        clk, reset,
             output logic[31:0]  pc,
             input  logic[31:0]  instr,
             output logic        memwrite,
             output logic[31:0]  aluout, writedata,
             input  logic[31:0]  readdata);

  logic        memtoreg, pcsrc, zero, alusrc, regdst, regwrite, jump, JumpReg;
  logic [2:0]  alucontrol;

  controller c (instr[31:26], instr[5:0], zero, memtoreg, memwrite, pcsrc,
                alusrc, regdst, regwrite, jump, JumpReg, alucontrol);

  datapath dp (clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump,
```

```
                    alucontrol, zero, pc, instr, aluout, writedata, readdata, JumpReg);

endmodule
module controller(input  logic[5:0] op, funct,
                  input  logic     zero,
                  output logic     memtoreg, memwrite,
                  output logic     pcsrc, alusrc,
                  output logic     regdst, regwrite,
                  output logic     jump,
                  output logic    JumpReg,
                  output logic[2:0] alucontrol);

   logic [1:0] aluop;
   logic       branch;

   maindec md (op, memtoreg, memwrite, branch, alusrc, regdst, regwrite,
                     jump, aluop, JumpReg);

   aludec  ad (funct, aluop, alucontrol);

   assign pcsrc = branch & zero;

endmodule

module maindec (input logic[5:0] op,
                      output logic memtoreg, memwrite, branch,
                      output logic alusrc, regdst, regwrite, jump,
                      output logic[1:0] aluop,
  output logic JumpReg );
   logic [9:0] controls;

   assign {regwrite, regdst, alusrc, branch, memwrite,
               memtoreg,  aluop, jump, JumpReg} = controls;

  always_comb
    case(op)
      6'b000000: controls <= 10'b1100001000; // R-type
      6'b000011: controls <= 10'b0000000001; // JR
      6'b100011: controls <= 10'b1010010000; // LW
      6'b101011: controls <= 10'b0010100000; // SW
      6'b000100: controls <= 10'b0001000100; // BEQ
      6'b001000: controls <= 10'b1010000000; // ADDI
      6'b000010: controls <= 10'b0000000010; // J
      6'b001110: controls <= 10'b1010001100; // XORI this will change
      default:   controls <= 10'bxxxxxxxxxx; // illegal op
    endcase
endmodule
```

```
module aludec (input    logic[5:0] funct,
               input    logic[1:0] aluop,
               output   logic[2:0] alucontrol);
  always_comb
    case(aluop)
      2'b00: alucontrol  = 3'b010;  // add  (for lw/sw/addi)
      2'b01: alucontrol  = 3'b110;  // sub   (for beq)
      2'b11: alucontrol  = 3'b011; //xor (for xori)
      default: case(funct)          // R-TYPE instructions
          6'b100000: alucontrol  = 3'b010; // ADD
          6'b100010: alucontrol  = 3'b110; // SUB
          6'b100100: alucontrol  = 3'b000; // AND
          6'b100101: alucontrol  = 3'b001; // OR
          6'b101010: alucontrol  = 3'b111; // SLT
          default:   alucontrol  = 3'bxxx; // ???
        endcase
    endcase
endmodule
//changes in the datapath are the most important ones
module datapath (input  logic clk, reset, memtoreg, pcsrc, alusrc, regdst,
                 input  logic regwrite, jump,
                        input  logic[2:0]  alucontrol,
                 output logic zero,
                        output logic[31:0] pc,
                    input  logic[31:0] instr,
                 output logic[31:0] aluout, writedata,
                    input  logic[31:0] readdata,
         input logic JumpReg);

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch, pcJumpReg;
  logic [31:0] signimm, signimmsh, srca, srcb, result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  regfile     rf (clk, regwrite, instr[25:21], instr[20:16], writereg,
                  result, srca, writedata);
  mux2 #(32)  pcJumpRegmux(pcnextbr, srca, JumpReg, pcJumpReg);
  mux2 #(32)  pcmux(pcJumpReg, {pcplus4[31:28],
                    instr[25:0], 2'b00}, jump, pcnext);

// register file logic
```

```
  mux2 #(5)    wrmux (instr[20:16], instr[15:11], regdst, writereg);
  mux2 #(32)  resmux (aluout, readdata, memtoreg, result);
  signext         se (instr[15:0], signimm);

 // ALU logic
  mux2 #(32)  srcbmux (writedata, signimm, alusrc, srcb);
  alu         alu (srca, srcb, alucontrol, aluout, zero);

endmodule


module regfile (input    logic clk, we3,
                input    logic[4:0]  ra1, ra2, wa3,
                input    logic[31:0] wd3,
                output   logic[31:0] rd1, rd2);

 logic [31:0] rf [31:0];

 // three ported register file: read two ports combinationally
 // write third port on rising edge of clock. Register0 hardwired to 0.

 always_ff@(posedge clk)
   if (we3)
      rf [wa3] <= wd3;

 assign rd1 = (ra1 != 0) ? rf [ra1] : 0;
 assign rd2 = (ra2 != 0) ? rf[ ra2] : 0;

endmodule


module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucont,
           output logic [31:0] result,
           output logic zero);

   always_comb
      case(alucont)
         3'b010: result = a + b;
               3'b011: result = a ^ b;
         3'b110: result = a - b;
         3'b000: result = a & b;
         3'b001: result = a | b;
         3'b111: result = (a < b) ? 1 : 0;
         default: result = {32{1'bx}};
      endcase

   assign zero = (result == 0) ? 1'b1 : 1'b0;
```

```
endmodule


module adder (input  logic[31:0] a, b,
              output logic[31:0] y);

     assign y = a + b;
endmodule

module sl2 (input  logic[31:0] a,
            output logic[31:0] y);

     assign y = {a[29:0], 2'b00}; // shifts left by 2
endmodule

module signext (input  logic[15:0] a,
                output logic[31:0] y);

  assign y = {{16{a[15]}}, a};    // sign-extends 16-bit a
endmodule

// parameterized register
module flopr #(parameter WIDTH = 8)
              (input logic clk, reset,
                   input logic[WIDTH-1:0] d,
               output logic[WIDTH-1:0] q);

  always_ff@(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule


// paramaterized 2-to-1 MUX
module mux2 #(parameter WIDTH = 8)
             (input  logic[WIDTH-1:0] d0, d1,
              input  logic s,
              output logic[WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule

`timescale 1ns / 1ps
//Test Bench
module testBench();
    logic clk, rst;
    logic [31:0] pc, instr, writedata, dataadr, readdata;
    logic memwrite;
```

```
    top test(clk, rst, writedata, dataadr, pc, instr, readdata, memwrite);
    initial begin
        clk = 0;
        rst = 1; #10;
        rst = 0;
    end
    always #5 clk <= ~clk;
endmodule

//////////////////////////////////////////////////////////////////////////////////
//
//   This module takes a slide switch or pushbutton input and
//   does the following:
//     --debounces it (ignoring any addtional changes for ~40 milliseconds)
//     --synchronizes it with the clock edges
//     --produces just one pulse, lasting for one clock period
//
//   Note that the 40 millisecond debounce time = 2000000 cycles of
//   50MHz clock (which has 20 nsec period)
//
//   Inputs/Outputs:
//   sw_input: the signal coming from the slide switch or pushbutton
//   CLK: the system clock on the BASYS3 board
//   clear: resets the pulse controller
//   clk_pulse: the synchronized debounced single-pulse output
//
//   Usage for CS224 Lab4-5:
//   - Give the BASYS3 clock and the push button signal as inputs
//   - You don't need to clear the controller
//   - Send the output pulse to your top module
//
//   For correct connections, carefully plan what should be in the .XDC file
//
//////////////////////////////////////////////////////////////////////////////////

module pulse_controller(
          input CLK, sw_input, clear,
          output reg clk_pulse );

           reg [2:0] state, nextstate;
           reg [27:0] CNT;
           wire cnt_zero;

          always @ (posedge CLK, posedge clear)
            if(clear)
                  state <=3'b000;
            else
                  state <= nextstate;
```

```
          always @ (sw_input, state, cnt_zero)
        case (state)
          3'b000: begin if (sw_input) nextstate = 3'b001;
                    else nextstate = 3'b000; clk_pulse = 0; end
          3'b001: begin nextstate = 3'b010; clk_pulse = 1; end
          3'b010: begin if (cnt_zero) nextstate = 3'b011;
                    else nextstate = 3'b010; clk_pulse = 1; end
          3'b011: begin if (sw_input) nextstate = 3'b011;
                    else nextstate = 3'b100; clk_pulse = 0; end
          3'b100: begin if (cnt_zero) nextstate = 3'b000;
                    else nextstate = 3'b100; clk_pulse = 0; end
         default: begin nextstate = 3'b000; clk_pulse = 0; end
        endcase

          always @(posedge CLK)
            case(state)
                  3'b001: CNT <= 100000000;
                  3'b010: CNT <= CNT-1;
                  3'b011: CNT <= 100000000;
                  3'b100: CNT <= CNT-1;
            endcase

//  reduction operator |CNT gives the OR of all bits in the CNT register
          assign cnt_zero = ~|CNT;

endmodule

//////////////////////////////////////////////////////
//
//  This module puts 4 hexadecimal values (from O to F) on the 4-digit 7-segment display
unit
//
//  Inputs/Outputs:
//  clk: the system clock on the BASYS3 board
//  in3, in2, in1, in0: the input hexadecimal values. in3(left), in2, in1, in0(right)
//  seg: the signals going to the segments of a digit.
//       seg[6] is CA for the a segment, seg[5] is CB for the b segment, etc
//  an:  anode, 4 bit enable signal, one bit for each digit
//       an[3] is the left-most digit, an[2] is the second-left-most, etc
//  dp:  digital point
//
//  Usage for CS224 Lab4-5:
//  - Give the system clock of BASYS3 and the hexadecimal values you want to display
as inputs.
//  - Send outputs to 7-segment display of BASYS3, using the .XDC file
//
//  Note: the an, seg and dp outputs are active-low, for the BASYS3 board
```

```
//
//  For correct connections, carefully plan what should be in the .XDC file
//
//////////////////////////////////////////////////////

module display_controller(

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'b0, 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

//Basys3 top module
```

```
module basys3Top(input logic clkButton, rstButton, clk,
                 output logic dp, memwrite,
                 output logic [3:0] an,
                 output logic [6:0] seg);
    logic pClk, pRst;
    logic[31:0] pc, instr, dataadr, writedata, readdata;
    pulse_controller pulseClk(clk, clkButtonn, 1'b0, pClk);
    pulse_controller pulseRst(clk, rstButton, 1'b0, pRst);

    display_controller sevenSegmentDisplay(clk, writedata[7:4], writedata[3:0],
dataadr[7:4], dataadr[3:0], seg, dp, an);
    top topModuleForMIPSlite(pClk, pRst, writedata, dataadr, pc, instr, readdata,
memwrite);

endmodule

module basysTest(input logic clkButton, rstButton, clk,
                 output logic dp, memwrite,
                 output logic [3:0] an,
                 output logic [6:0] seg);
    basys3Top finalTest (clkButton, rstButton, clk,dp, memwrite,an, seg);
endmodule
```