Code:

<u>Machine Code:</u>

```
20080001

20090001

01098020

01285022

210b0000

00000000

120a0003

014b5020

08100006

00000000

20080007

20090007
```

Assembly Code:

```
addi $t0, $0, 1         # t0 = 1
addi $t1, $0, 1         # t1 = 1

add $s0, $t0, $t1       # s0 = t0 + t1 = 2
sub $t2, $t1, $t0       # t2 = t1 - t0 = 0

addi $t3, $t0, 0  # t3 = t0 + (-1) = 1
nop

label:
beq $s0, $t2, end # checks if SUM = 2
add $t2, $t2, $t3 # counter, where t2 + t3 = SUM + 1

j label                       # if not equal, keep going through the counter
nop

end:
addi $t0, $0, 7         # t0 = 0 + 7 = 7
addi $t1, $0, 7         # t1 = 0 + 7 = 7
```

Verilog Code:

top.v

```
//--------------------------------------------------
// top.v
//--------------------------------------------------

module top_final(input         clk, reset,
                 //input  [31:0] readdata1, readdata2, //readdata is
ReadDataM
                 output [31:0] writedata1, writedata2, dataadr1, dataadr2,
//writedata is actually WriteDataM, and dataadr is the result of the ALU
                 output        MemWriteM1, MemWriteM2,
                         output [31:0] ResultW1, ResultW2); //This way the
testbench tests for MemWrite at the memory stage

  wire [31:0] pc;
  wire [63:0] instr;
  wire [31:0] readdata1, readdata2;

  // instantiate processor and memories
  mipssingle mipssingle(clk, reset, pc, instr, dataadr1, dataadr2,
                        writedata1, writedata2, readdata1, readdata2,
                                    MemWriteM1, MemWriteM2, ResultW1,
ResultW2);

  //Instruction Memory
  imem imem(pc[7:2], instr);

  //Data Memory
  dmem dmem(clk, MemWriteM1, MemWriteM2, dataadr1, dataadr2, writedata1,
writedata2, //Changed memwrite to MemWriteM
            readdata1, readdata2, reset);
endmodule
```

mipssingle.v

```
module mipssingle(input         clk, reset,
                  output [31:0] pcF,
                  input  [63:0] instrF,
                  output [31:0] ALUOutE1, ALUOutE2, WriteDataM1, WriteDataM2,
                  input  [31:0] readdataM1, readdataM2,
                          output          MemWriteM1, MemWriteM2,
                          output [31:0] ResultW1, ResultW2);
                          //Added MemWriteM1/MemWriteM2 for testbench
checking

  //Wires for 1st Path
  wire         memtoregD1, memwriteD1;
  wire         alusrcD1;
  wire         regdstD1;
  wire         regwriteD1, jump, pcsrcD1, zero;
  wire [3:0]   alucontrolD1;
  wire               EqualD1; // Output for the 'equals' module when solving
for 'PCSrcD'
  wire               branchD1;
  wire [31:0] instrD1;

  //Wires for 2nd Path
  wire         memtoregD2, memwriteD2;
  wire         alusrcD2;
  wire         regdstD2;
  wire         regwriteD2, pcsrcD2;
  wire [3:0]   alucontrolD2;
  wire               EqualD2; // Output for the 'equals' module when solving
for 'PCSrcD'
  wire               branchD2;
  wire [31:0] instrD2;

  wire temp;
  assign temp = 1'b0;

//CONTROLLER UNIT 1 INSTANTIATION
 controller1 c1(instrD1[31:26], instrD1[5:0], zero, //changed instr to
instrd1
               memtoregD1, memwriteD1, pcsrcD1,
               alusrcD1, regdstD1, regwriteD1, jump,
               alucontrolD1, EqualD1, branchD1); //EqualD1 and branch need to
go in here now

//CONTROLLER UNIT 2 INSTANTIATION
 controller2 c2(instrD2[31:26], instrD2[5:0], zero, //changed instr to
instrd1
               memtoregD2, memwriteD2, pcsrcD2,
               alusrcD2, regdstD2, regwriteD2, temp,
               alucontrolD2, EqualD2, branchD2); //EqualD1 and branch need to
go in here now
```

```
 datapath dp(clk, reset, memtoregD1, memtoregD2, pcsrcD1, pcsrcD2,
              alusrcD1, alusrcD2, regdstD1, regdstD2, regwriteD1, regwriteD2,
                    jump, alucontrolD1, alucontrolD2,
              zero, pcF, instrF[63:32], instrF[31:0],
              ALUOutE1, ALUOutE2, WriteDataM1, WriteDataM2, readdataM1,
readdataM2,
                    EqualD1, EqualD2, memwriteD1, memwriteD2, branchD1,
branchD2,
                    MemWriteM1, MemWriteM2, instrD1, instrD2, ResultW1,
ResultW2); //added EqualD1, memwrite, branch, MemWriteM1, and instrd1

endmodule

//CONTROLLER UNIT 1
module controller1(input  [5:0] op, funct,
                   input        zero,
                   output       memtoreg, memwrite,
                   output       pcsrc,
                           output       alusrc,
                   output       regdst,
                           output       regwrite,
                   output       jump,
                   output [3:0] alucontrol,  // 4 bits for SLL
                           input              EqualD1,
                           output       branch);

  wire [1:0] aluop;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = (branch & EqualD1); //For branching
endmodule

//CONTROLLER UNIT 2
module controller2(input  [5:0] op, funct,
                   input        zero,
                   output       memtoreg, memwrite,
                   output       pcsrc,
                           output       alusrc,
                   output       regdst,
                           output       regwrite,
                   output       jump,
                   output [3:0] alucontrol,  // 4 bits for SLL
                           input              EqualD2,
                           output       branch);

  wire [1:0] aluop;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = (branch & EqualD2); //For branching
```

```
endmodule


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

  reg [8:0] controls;

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

  always @(*)
    case(op)
      6'b000000: controls <= 9'b110000010; //Rtype
      6'b100011: controls <= 9'b101001000; //LW
      6'b101011: controls <= 9'b001010000; //SW
      6'b000100: controls <= 9'b000100001; //BEQ
      6'b001000: controls <= 9'b101000000; //ADDI
      6'b000010: controls <= 9'b000000100; //J
      default:   controls <= 9'bxxxxxxxxx; //???
    endcase
endmodule

module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [3:0] alucontrol); // 4-bits for SLL

  always @(*)
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add
      2'b01: alucontrol <= 4'b1010;  // sub
        2'b11: alucontrol <= 4'b1011;  // slt
      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0010; // ADD
          6'b100010: alucontrol <= 4'b1010; // SUB
          6'b100100: alucontrol <= 4'b0000; // AND
          6'b100101: alucontrol <= 4'b0001; // OR
          6'b101010: alucontrol <= 4'b1011; // SLT
              //6'b000010: alucontrol <= 4'b0101; // MUL
              //6'b011010: alucontrol <= 4'b0110; // DIV
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule

module datapath(input         clk, reset,
                input         memtoregD1, memtoregD2, pcsrcD1, pcsrcD2,
                input         alusrcD1, alusrcD2, //AluSrcD
                        input         regdstD1, regdstD2, //RegDstD
```

```
                 input          regwriteD1, regwriteD2, jump, //RegWriteD
                 input  [3:0]  alucontrolD1, alucontrolD2, //ALUControlD
                 output         zero, //trash wire (not used)
                 output [31:0] pcF,
                 input  [31:0] instrF1, //1st Instruction
                        input  [31:0] instrF2, //2nd Instruction
                 output [31:0] ALUOutM1, ALUOutM2, WriteDataM1, WriteDataM2,
                 input  [31:0] ReadDataM1, ReadDataM2,
                        output              EqualD1, EqualD2,
                        input         memwriteD1, memwriteD2,
                        input               BranchD1, BranchD2,
                        output              MemWriteM1, MemWriteM2,
                        output [31:0] instrD1,
                        output [31:0] instrD2,
                        output [31:0] ResultW1, ResultW2);

  wire [31:0] pcnext, pcnextbr, pcplus8F, pcbranch, pcplus4F, pcplus4D;
  wire [31:0] signimmD1, signimmD2, signimmshD1, signimmshD2;
  wire [31:0] srcaD1, srcaD2, srcbD1, srcbD2;
  //wire [31:0] ResultW1, ResultW2;

  /**************************************************************/
  // Extra wires created for Pipeline/Hazard-Unit
  /**************************************************************/

  wire MemtoRegE1, MemWriteE1, RegWriteE1, MemtoRegM1, RegWriteM1,
RegWriteW1, StallF, StallD1, StallD2, FlushE1, FlushE2, ForwardAD1,
ForwardBD1, ALUSrcE1, RegDstE1, MemtoRegW1;
  wire MemtoRegE2, MemWriteE2, RegWriteE2, MemtoRegM2, RegWriteM2,
RegWriteW2, ForwardAD2, ForwardBD2, ALUSrcE2, RegDstE2, MemtoRegW2;
  wire [4:0] RsD1, RtD1, WriteRegE1, WriteRegM1, WriteRegW1, RsE1, RtE1,
RdE1, shamt1;
  wire [4:0] RsD2, RtD2, WriteRegE2, WriteRegM2, WriteRegW2, RsE2, RtE2,
RdE2, shamt2;
  wire [2:0] ForwardAE1, ForwardBE1;
  wire [2:0] ForwardAE2, ForwardBE2;
  wire [31:0] srcaeqD1, srcbeqD1, SrcAE1, SrcAEM1, SrcBE1, SrcBEMM1,
SignImmE1, WriteDataE1, ALUOutE1, ReadDataW1, ALUOutW1, pcplus8D;
  wire [31:0] srcaeqD2, srcbeqD2, SrcAE2, SrcAEM2, SrcBE2, SrcBEMM2,
SignImmE2, WriteDataE2, ALUOutE2, ReadDataW2, ALUOutW2;
  wire [3:0]  ALUControlE1;
  wire [3:0]  ALUControlE2;

  /**************************************************************/

  /**************************************************************/
  // Hazard Unit
  /**************************************************************/

  hazunit hu(BranchD1, BranchD2, MemtoRegE1, MemtoRegE2, RegWriteE1,
RegWriteE2, MemtoRegM1, MemtoRegM2,
                        RegWriteM1, RegWriteM2, RegWriteW1, RegWriteW2,
                        RsD1, RsD2, RtD1, RtD2,
                        RsE1, RsE2, RtE1, RtE2,
                        WriteRegE1, WriteRegE2, WriteRegM1, WriteRegM2,
WriteRegW1, WriteRegW2,
                        StallF, StallD1, StallD2, FlushE1, FlushE2,
                        ForwardAD1, ForwardAD2, ForwardBD1, ForwardBD2,
                        ForwardAE1, ForwardBE1, ForwardAE2, ForwardBE2);
//Hazard Unit instantiation

  /**************************************************************/


  /**************************************************************/
  // Pipeline Registers & MUX's & Equalizers added
  /**************************************************************/

  //Pipeline register in between Fetch and Decode
  mux2  #(32) eqda1(srcaD1, ALUOutM1, //ForwardAD1 MUX in Decode stage
                           ForwardAD1, srcaeqD1);
  mux2  #(32) eqdb1(srcbD1, ALUOutM1, //ForwardBD1 MUX in Decode stage
                           ForwardBD1, srcbeqD1);
  equals      eq1(srcaeqD1, srcbeqD1, //Checks to see if srcad1 and srcb are
equal to determine branching
                         EqualD1);

  mux2  #(32) eqda2(srcaD2, ALUOutM2, //ForwardAD1 MUX in Decode stage
                           ForwardAD2, srcaeqD2);
  mux2  #(32) eqdb2(srcbD2, ALUOutM2, //ForwardBD1 MUX in Decode stage
                           ForwardBD2, srcbeqD2);
  equals      eq2(srcaeqD2, srcbeqD2, //Checks to see if srcad1 and srcb are
equal to determine branching
                         EqualD2);


  //FIRST LANE
  regf      /*#(64)*/ rfe(clk, (pcsrcD1 | jump), /*(pcsrcD2 | jump),*/ reset,
~StallD1/*enable*/, instrF1, pcplus8F, //inputs into register f
                                instrD1, pcplus8D); //outputs out of register
f

  //Pipeline register in between Decode and Execute
  regd      /*#(120)*/rd(clk, FlushE1, reset, regwriteD1, memtoregD1,
memwriteD1, alucontrolD1, alusrcD1, regdstD1, srcaD1, srcbD1,
                               instrD1[25:21], instrD1[20:16],
instrD1[15:11], signimmD1, instrD1[10:6], //inputs into register d
                               RegWriteE1, MemtoRegE1, MemWriteE1,
ALUControlE1, ALUSrcE1, RegDstE1, SrcAE1, SrcBE1,
                               RsE1, RtE1, RdE1, SignImmE1, shamt1);
//outputs out of register d

  //Pipeline register in between Execute and Memory
  rege      /*#(71)*/re(clk, reset, RegWriteE1, MemtoRegE1, MemWriteE1,
ALUOutE1, WriteDataE1, WriteRegE1, //inputs into register e
                              RegWriteM1, MemtoRegM1, MemWriteM1, ALUOutM1,
WriteDataM1, WriteRegM1); //output out of register e

  //Pipeline register in between Memory and Write Back
  regm      /*#(71)*/rm(clk, reset, RegWriteM1, MemtoRegM1, ReadDataM1,
ALUOutM1, WriteRegM1, //inputs into register m
                              RegWriteW1, MemtoRegW1, ReadDataW1, ALUOutW1,
WriteRegW1); //outputs out of register m
```

```
 //SECOND LANE
  //Pipeline register in between Fetch and Decode
  regf      /*#(64)*/ rfe2(clk, (pcsrcD1 | jump), /*(pcsrcD2 | jump),*/
reset, ~StallD2/*enable*/, instrF2, pcplus4F, //inputs into register f
                                instrD2, pcplus4D); //outputs out of register
f

  //Pipeline register in between Decode and Execute
  regd      /*#(120)*/rd2(clk, FlushE2, reset, regwriteD2, memtoregD2,
memwriteD2, alucontrolD2, alusrcD2, regdstD2, srcaD2, srcbD2,
                               instrD2[25:21], instrD2[20:16],
instrD2[15:11], signimmD2, instrD2[10:6], //inputs into register d
                               RegWriteE2, MemtoRegE2, MemWriteE2,
ALUControlE2, ALUSrcE2, RegDstE2, SrcAE2, SrcBE2,
                               RsE2, RtE2, RdE2, SignImmE2, shamt2);
//outputs out of register d

  //Pipeline register in between Execute and Memory
  rege      /*#(71)*/re2(clk, reset, RegWriteE2, MemtoRegE2, MemWriteE2,
ALUOutE2, WriteDataE2, WriteRegE2, //inputs into register e
                              RegWriteM2, MemtoRegM2, MemWriteM2, ALUOutM2,
WriteDataM2, WriteRegM2); //output out of register e

  //Pipeline register in between Memory and Write Back
  regm      /*#(71)*/rm2(clk, reset, RegWriteM2, MemtoRegM2, ReadDataM2,
ALUOutM2, WriteRegM2, //inputs into register m
                              RegWriteW2, MemtoRegW2, ReadDataW2, ALUOutW2,
WriteRegW2); //outputs out of register m


  /***************************************************************/

  //The below is orginal code

  // next PC logic
  flopenr #(32) pcreg(clk, reset, ~StallF/*enable*/,pcnext, pcF);
  adder        pcadd1(pcF, 32'b1000, pcplus8F);
  adder        pcadd4(pcF, 32'b100, pcplus4F);
  sl2          immsh(signimmD1, signimmshD1);
  adder        pcadd2(pcplus4D, signimmshD1, pcbranch);
  mux2 #(32)   pcbrmux(pcplus8F, pcbranch, pcsrcD1,
                       pcnextbr);
  mux2 #(32)   pcmux(pcnextbr, {pcplus8F[31:28],
                     instrD1[25:0], 2'b00},
                     jump, pcnext);

  // register file logic
  regfile      rf(clk, RegWriteW1, RegWriteW2, instrD1[25:21], instrD2[25:21],
                  instrD1[20:16], instrD2[20:16], WriteRegW1, WriteRegW2,
                          ResultW1, ResultW2,
                          srcaD1, srcaD2, srcbD1, srcbD2);

  mux2 #(5)    wrmux1(RtE1, RdE1,
                      RegDstE1, WriteRegE1);
  mux2 #(5)    wrmux2(RtE2, RdE2,
                      RegDstE2, WriteRegE2);
  mux2 #(32)  resmux1(ALUOutW1, ReadDataW1,
                                MemtoRegW1, ResultW1);
  mux2 #(32)  resmux2(ALUOutW2, ReadDataW2,
                                MemtoRegW2, ResultW2);
  signext     se1(instrD1[15:0], signimmD1);
  signext     se2(instrD2[15:0], signimmD2);

  // ALU1 logic
  mux5 #(32)  srcaemux1(SrcAE1, ResultW1, ALUOutM1, ResultW2, ALUOutM2,
                                 ForwardAE1, SrcAEM1);
  mux5 #(32)  srcbemux1(SrcBE1, ResultW1, ALUOutM1, ResultW2, ALUOutM2,
                                 ForwardBE1, WriteDataE1);
  mux2 #(32)  srcbemmux1(WriteDataE1, SignImmE1,
                                    ALUSrcE1, SrcBEMM1);

  alu         alu1(SrcAEM1, SrcBEMM1, ALUControlE1, shamt1,  // SLL
                  ALUOutE1, zero);

  // ALU2 logic
  mux5 #(32)  srcaemux2(SrcAE2, ResultW2, ALUOutM2, ResultW1, ALUOutM1,
                                  ForwardAE2, SrcAEM2);
  mux5 #(32)  srcbemux2(SrcBE2, ResultW2, ALUOutM2, ResultW1, ALUOutM1,
                                  ForwardBE2, WriteDataE2);
  mux2 #(32)  srcbemmux2(WriteDataE2, SignImmE2,
                                     ALUSrcE2, SrcBEMM2);

  alu         alu2(SrcAEM2, SrcBEMM2, ALUControlE2, shamt2,  // SLL
                  ALUOutE2, zero);
Endmodule
```

mipsmem.v

```
//------------------------------------------------
// mipsmem.v
//------------------------------------------------

module dmem(input         clk, we1, we2,
            input  [31:0] a1, a2, wd1, wd2,
            output [31:0] rd1, rd2,
                  input                reset);

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

  assign rd1 = RAM[a1[31:2]]; // word aligned for address 1
  assign rd2 = RAM[a2[31:2]]; // word aligned for address 2

  always @(posedge clk)
  begin
    if (we1)
      RAM[a1[31:2]] <= wd1;
      if (we2)
      RAM[a2[31:2]] <= wd2;
  end
endmodule


module imem(input  [5:0] a,
            output [63:0] rd); //Expanded from 31:0 to 63:0

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

  initial
    begin
      $readmemh("memfile.dat",RAM);
    end

  assign rd = {RAM[a], RAM[a+1]}; // word aligned to take TWO instructions at
once
endmodule
```

mipsparts.v

```
//------------------------------------------------
// mipsparts.v
//------------------------------------------------


module regfile(input         clk,
               input         we3,
                     input          we4,
               input   [4:0]  ra1, ra3, ra2, ra4, wa3, wa4,
               input   [31:0] wd3, wd4,
               output  [31:0] rd1, rd3, rd2, rd4);

  reg [31:0] rf[31:0];

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

  always @(negedge clk)
  begin
    if (we3) rf[wa3] <= wd3;
      if (we4) rf[wa4] <= wd4;
  end

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
  assign rd3 = (ra3 != 0) ? rf[ra3] : 0;
  assign rd4 = (ra4 != 0) ? rf[ra4] : 0;
endmodule

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

  assign y = a + b;
endmodule

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

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

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

  assign y = {{16{a[15]}}, a};
endmodule

module flopr #(parameter WIDTH = 8)
              (input                  clk, reset,
               input      [WIDTH-1:0] d,
               output reg [WIDTH-1:0] q);
  always @(posedge clk)
    if (reset) q <= 0;
    else       q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
                (input                  clk, reset,
                 input                  en,
                 input      [WIDTH-1:0] d,
                 output reg [WIDTH-1:0] q);

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

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

  assign y = s ? d1 : d0;
endmodule


// mux3 needed for LUI
module mux3 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1, d2,
              input  [1:0]       s,
              output [WIDTH-1:0] y);

  assign #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule


// mux5 needed for Superscalar forwarding
module mux5 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1, d2, d3, d4,
              input  [2:0]       s,
              output [WIDTH-1:0] y);

  assign #1 y = s[2] ? d4 : (s[1] & s[0]) ? d3 : s[1] ? d2 : s[0] ? d1 : d0;
  //if select = 100, then out d4; else if select = 011, then out d3; else if
select = 10, then out d2; else if select = 1, then out d1; else if select =
0, then out d0
endmodule

/**************************************************************/
// Pipeline Registers
/**************************************************************/

//Pipeline register in between Fetch and Decode
module regf (input                      clk, clr, reset, en,
                   input            [31:0]      instrF1, pcplus4f,
                   output reg [31:0]      instrD1, pcplus4d);
  always @(posedge clk) //Everything stays synchronous to clock
    if (en) //So long as there's an enable (no StallF), continue through
register operations
    if (clr || reset) begin   //If clear or reset, set all outputs as 0
            instrD1 <= 0;
            pcplus4d <= 0;
      end
    else begin  //If no reset, let values go through as clock is high
            instrD1 <= instrF1;
            pcplus4d <= pcplus4f;
      end
endmodule

//Pipeline register in between Decode and Execute
module regd (input                     clk, clr, reset, regwriteD1,
                                             memtoregD1, memwriteD1,
                   input      [3:0]  alucontrolD1,
                   input                     alusrcD1, regdstD1,
                   input            [31:0] srcaD1, srcbD1,
                   input            [4:0]  rsdD1, rtdD1, rddD1,
                   input            [31:0] signimmD1,
                   input            [4:0]  aluinstrD1,
                   output reg          RegWriteE1, MemtoRegE1,
                                             MemWriteE1,
                   output reg [3:0]  ALUControlE1,
                   output     reg            ALUSrcE1, RegDstE1,
                   output reg [31:0] SrcAE1, SrcBE1,
                   output reg [4:0]  RsE1, RtE1, RdE1,
                   output reg [31:0] SignImmE1,
                   output reg [4:0]  shamt1);

  always @(posedge clk) //Everything stays synchronous to clock
      if (clr || reset) begin   //If clear or reset, set all outputs as 0
            RegWriteE1 <= 0;
            MemtoRegE1 <= 0;
            MemWriteE1 <= 0;
            ALUControlE1 <= 0;
            ALUSrcE1 <= 0;
            RegDstE1 <= 0;
            SrcAE1 <= 0;
            SrcBE1 <= 0;
            RsE1 <= 0;
            RtE1 <= 0;
            RdE1 <= 0;
            SignImmE1 <= 0;
            shamt1 <= 0;
      end
    else begin   //If no reset, let values go through as clock is high
            RegWriteE1 <= regwriteD1;
            MemtoRegE1 <= memtoregD1;
            MemWriteE1 <= memwriteD1;
            ALUControlE1 <= alucontrolD1;
            ALUSrcE1 <= alusrcD1;
            RegDstE1 <= regdstD1;
            SrcAE1 <= srcaD1;
            SrcBE1 <= srcbD1;
            RsE1 <= rsdD1;
```

```verilog
            RtE1 <= rtdD1;
            RdE1 <= rddD1;
            SignImmE1 <= signimmD1;
            shamt1 <= aluinstrD1;
      end
endmodule

//Pipeline register in between Execute and Memory
module rege (input                     clk, reset, RegWriteE1, MemtoRegE1,
MemWriteE1,
                   input           [31:0] ALUOutE1, WriteDataE1,
                   input           [4:0]  WriteRegE1,
                   output     reg             RegWriteM1, MemtoRegM1,
MemWriteM1,
                   output reg [31:0] ALUOutM1, WriteDataM1,
                   output reg [4:0]  WriteRegM1);

  always @(posedge clk) //Everything stays synchronous to clock
    if (reset)
      begin   //If reset is high, set all outputs 0
            RegWriteM1 <= 0;
            MemtoRegM1 <= 0;
            MemWriteM1 <= 0;
            ALUOutM1 <= 0;
            WriteDataM1 <= 0;
            WriteRegM1 <= 0;
      end
      else begin //If no reset, let values go through as clock is high
            RegWriteM1 <= RegWriteE1;
            MemtoRegM1 <= MemtoRegE1;
            MemWriteM1 <= MemWriteE1;
            ALUOutM1 <= ALUOutE1;
            WriteDataM1 <= WriteDataE1;
            WriteRegM1 <= WriteRegE1;
      end
endmodule

//Pipeline register in between Memory and Write Back
module regm (input                     clk, reset, RegWriteM1,
                                          MemtoRegM1,
                   input           [31:0] ReadDataM1, ALUOutM1,
                   input           [4:0]  WriteRegM1,
                   output     reg             RegWriteW1, MemtoRegW1,
                   output reg [31:0] ReadDataW1, ALUOutW1,
                   output reg [4:0]  WriteRegW1);

  always @(posedge clk) //Everything stays synchronous to clock
    if (reset) //When reset is high, give all outputs 0
      begin
            RegWriteW1 <= 0;
            MemtoRegW1 <= 0;
            ReadDataW1 <= 0;
            ALUOutW1 <= 0;
            WriteRegW1 <= 0;
      end
      else //If no reset, let values go through as clock is high
      begin
```

```
            RegWriteW1 <= RegWriteM1;
            MemtoRegW1 <= MemtoRegM1;
            ReadDataW1 <= ReadDataM1;
            ALUOutW1 <= ALUOutM1;
            WriteRegW1 <= WriteRegM1;
      end
endmodule

/**************************************************************/
// Equalizer
/**************************************************************/
//This is the arithmetic that will return whether or not srca and srcb are
equal. Then follow up on the branch instruction
module equals (input [31:0] A, B,
                     output Y);

      assign Y = ( A === B ) ? 1 : 0; //If srca is equal to srcb, then output
1 bit 1
endmodule
```

mipshaz.v

```
module hazunit (input          BranchD1, BranchD2, MemtoRegE1, MemtoRegE2,
RegWriteE1, RegWriteE2, MemtoRegM1, MemtoRegM2,
                        input        RegWriteM1, RegWriteM2, RegWriteW1,
RegWriteW2,
                        input  [4:0] RsD1, RsD2, RtD1, RtD2,
                        input  [4:0] RsE1, RsE2, RtE1, RtE2,
                        input  [4:0] WriteRegE1, WriteRegE2, WriteRegM1,
WriteRegM2, WriteRegW1, WriteRegW2,
                        output             StallF, StallD1, StallD2, FlushE1,
FlushE2,
                        output             ForwardAD1, ForwardAD2,
ForwardBD1, ForwardBD2,
                        output [2:0] ForwardAE1, ForwardBE1,
                        output [2:0] ForwardAE2, ForwardBE2);

      wire branchstall1, lwstall1; //Wires used for the stalls and flush
      wire branchstall2, lwstall2;

//Forwarding Logic
      assign ForwardAD1 = (RsD1 != 0) & (RsD1 == WriteRegM1) & RegWriteM1;
//Logic for ForwardAD to push forward values from memory to decode or just
pass through
      assign ForwardBD1 = (RtD1 != 0) & (RtD1 == WriteRegM1) & RegWriteM1;
//Logic for ForwardBD to push forward values from memory to decode or just
pass through
      assign ForwardAD2 = (RsD2 != 0) & (RsD2 == WriteRegM2) & RegWriteM2;
//Logic for ForwardAD to push forward values from memory to decode or just
pass through
      assign ForwardBD2 = (RtD2 != 0) & (RtD2 == WriteRegM2) & RegWriteM2;
//Logic for ForwardBD to push forward values from memory to decode or just
pass through
      //As long as it's an I-Type (because branching has to use two registers
RS and RT) AND the same register is about to be reused before placing the
value in RegFile AND we still need to write the previous value of an older
instruction into RegFile

      //lane 1
      assign ForwardAE1 = ((RsE1 != 0) & (RsE1 == WriteRegM1) & RegWriteM1) ?
3'b010 : ((RsE1 != 0) & (RsE1 == WriteRegW1) & RegWriteW1) ? 3'b001 : //2'b00
); //Logic for ForwardAE to push forward values from memory to execute or WB
to execute
                          ((RsE1 != 0) & (RsE1 == WriteRegM2) & RegWriteM2) ?
3'b100 : (((RsE1 != 0) & (RsE1 == WriteRegW2) & RegWriteW2) ? 3'b011 :
3'b000); //forwarding from lane 1 to lane 2
      assign ForwardBE1 = ((RtE1 != 0) & (RtE1 == WriteRegM1) & RegWriteM1) ?
3'b010 : ((RtE1 != 0) & (RtE1 == WriteRegW1) & RegWriteW1) ? 3'b001 : //2'b00
); //Logic for ForwardBE to push forward values from memory to execute or WB
to execute
                                    ((RtE1 != 0) & (RtE1 == WriteRegM2) &
RegWriteM2) ? 3'b100 : (((RtE1 != 0) & (RtE1 == WriteRegW2) & RegWriteW2) ?
3'b011 : 3'b000); //forwarding from lane 1 to lane 2
      //lane 2
      assign ForwardAE2 = ((RsE2 != 0) & (RsE2 == WriteRegM2) & RegWriteM2) ?
3'b010 : ((RsE2 != 0) & (RsE2 == WriteRegW2) & RegWriteW2) ? 3'b001 : //2'b00
```

```
); //Logic for ForwardAE to push forward values from memory to execute or WB
to execute
                           ((RsE2 != 0) & (RsE2 == WriteRegM1) & RegWriteM1) ?
3'b100 : (((RsE2 != 0) & (RsE2 == WriteRegW1) & RegWriteW1) ? 3'b011 :
3'b000); //forwarding from lane 1 to lane 2
      assign ForwardBE2 = ((RtE2 != 0) & (RtE2 == WriteRegM2) & RegWriteM2) ?
3'b010 : ((RtE2 != 0) & (RtE2 == WriteRegW2) & RegWriteW2) ? 3'b001 : //2'b00
); //Logic for ForwardBE to push forward values from memory to execute or WB
to execute
                                     ((RtE2 != 0) & (RtE2 == WriteRegM1) &
RegWriteM1) ? 3'b100 : (((RtE2 != 0) & (RtE2 == WriteRegW1) & RegWriteW1) ?
3'b011 : 3'b000); //forwarding from lane 1 to lane 2
      //default = 000, Grab from WB2 = 001, Grab from Mem2 = 010, Grab from
WB1 = 011, Grab from Mem1 = 100
      //assign ForwardBE1 = ((RtE1 != 0) & (RtE1 == WriteRegM1) & RegWriteM1)
? 2'b10 : ( ((RtE1 != 0) & (RtE1 == WriteRegW1) & RegWriteW1) ? 2'b01 : 2'b00
); //Logic for ForwardBE to push forward values from memory to execute or WB
to execute
      //assign ForwardAE2 = ((RsE2 != 0) & (RsE2 == WriteRegM2) & RegWriteM2)
? 2'b10 : ( ((RsE2 != 0) & (RsE2 == WriteRegW2) & RegWriteW2) ? 2'b01 : 2'b00
); //Logic for ForwardAE to push forward values from memory to execute or WB
to execute
      //assign ForwardBE2 = ((RtE2 != 0) & (RtE2 == WriteRegM2) & RegWriteM2)
? 2'b10 : ( ((RtE2 != 0) & (RtE2 == WriteRegW2) & RegWriteW2) ? 2'b01 : 2'b00
); //Logic for ForwardBE to push forward values from memory to execute or WB
to execute
      //IF rs and/or rt holds a value AND the older rs and rt was used in a
previous instruction (memory stage) AND the previous instruction is going to
write a value to the register file
      //THEN grabs the 'previous' instructions value
      //ELSE IF rs and/or rt holds a value AND the older rs and rt was used
in 2 previous instructions ago (write back stage) AND that previous
instruction is going to write a value to the register file
      //THEN grabs the value of the '2nd previous' instruction value
      //ELSE 'no forwarding'

//Stalling Logic
      assign branchstall1 = (BranchD1 & (RegWriteE1 & ((WriteRegE1 == RsD1) |
(WriteRegE1 == RtD1)) | MemtoRegM1 & ((WriteRegM1 == RsD1) | (WriteRegM1 ==
RtD1))));
      assign branchstall2 = (BranchD2 & (RegWriteE2 & ((WriteRegE2 == RsD2) |
(WriteRegE2 == RtD2)) | MemtoRegM2 & ((WriteRegM2 == RsD2) | (WriteRegM2 ==
RtD2)))); //Logic to check if branching
      assign lwstall1 = (((RsD1 == RtE1) | (RtD1 == RtE1)) & MemtoRegE1);
      assign lwstall2 = (((RsD2 == RtE2) | (RtD2 == RtE2)) & MemtoRegE2);
//Logic to check if load word
      //assign parallelstall = (RsE2 == WriteRegM1) | (RsE1 == WriteRegM2);

      assign StallF = lwstall1 | branchstall1 | lwstall2 | branchstall2;
//Logic for StallF1
      assign StallD1 = lwstall1 | branchstall1; //Logic for StallD
      assign StallD2 = lwstall2 | branchstall2 | branchstall1; //Logic for
StallD
      assign FlushE1 = lwstall1 | branchstall1; //Logic for FlushE
      assign FlushE2 = lwstall2 | branchstall2 | branchstall1; //Logic for
FlushE
endmodule
```

alu32.v

```
//------------------------------------------------
// alu32.v
//------------------------------------------------

`timescale 1ns / 1ps

module alu( input [31:0] A, B,
            input [3:0] F, input [4:0] shamt, // SLL
                        output reg [31:0] Y, output Zero);

      wire [31:0] S, Bout;
      wire ltez;

      assign Bout = F[3] ? ~B : B;
      assign S = A + Bout + F[3];  // SLL

      always @ ( * )
            case (F[2:0])
                  3'b000: Y <= A & Bout;
                  3'b001: Y <= A | Bout;
                  3'b010: Y <= S;
                  3'b011: Y <= S[31];
                  3'b100: Y <= (Bout << shamt);  // SLL
                  3'b101: Y <= A * B; // MUL
                  3'b110: Y <= A / B; // DIV
            endcase

      assign Zero = (Y == 32'b0);
      assign ltez = Zero | S[31];  // BLEZ

//    assign Overflow =  A[31]& Bout[31] & ~Y[31] |
//                                          ~A[31] & ~Bout[31] & Y[31];

endmodule
```

mipstest.v

```
//------------------------------------------------
// mipstest.v
//------------------------------------------------

module testbench();

  reg         clk;
  reg         reset;

  wire [31:0] dataadr1, dataadr2, writedata1, writedata2, readdata1,
readdata2, ResultW1, ResultW2;
  wire memwrite1, memwrite2;

  // instantiate device to be tested
  top_final dut(clk, reset, writedata1, writedata2,
                dataadr1, dataadr2, memwritem1, memwritem2, ResultW1,
ResultW2);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

endmodule
```