Ihsan Doğramacı Bilkent University

CS 224

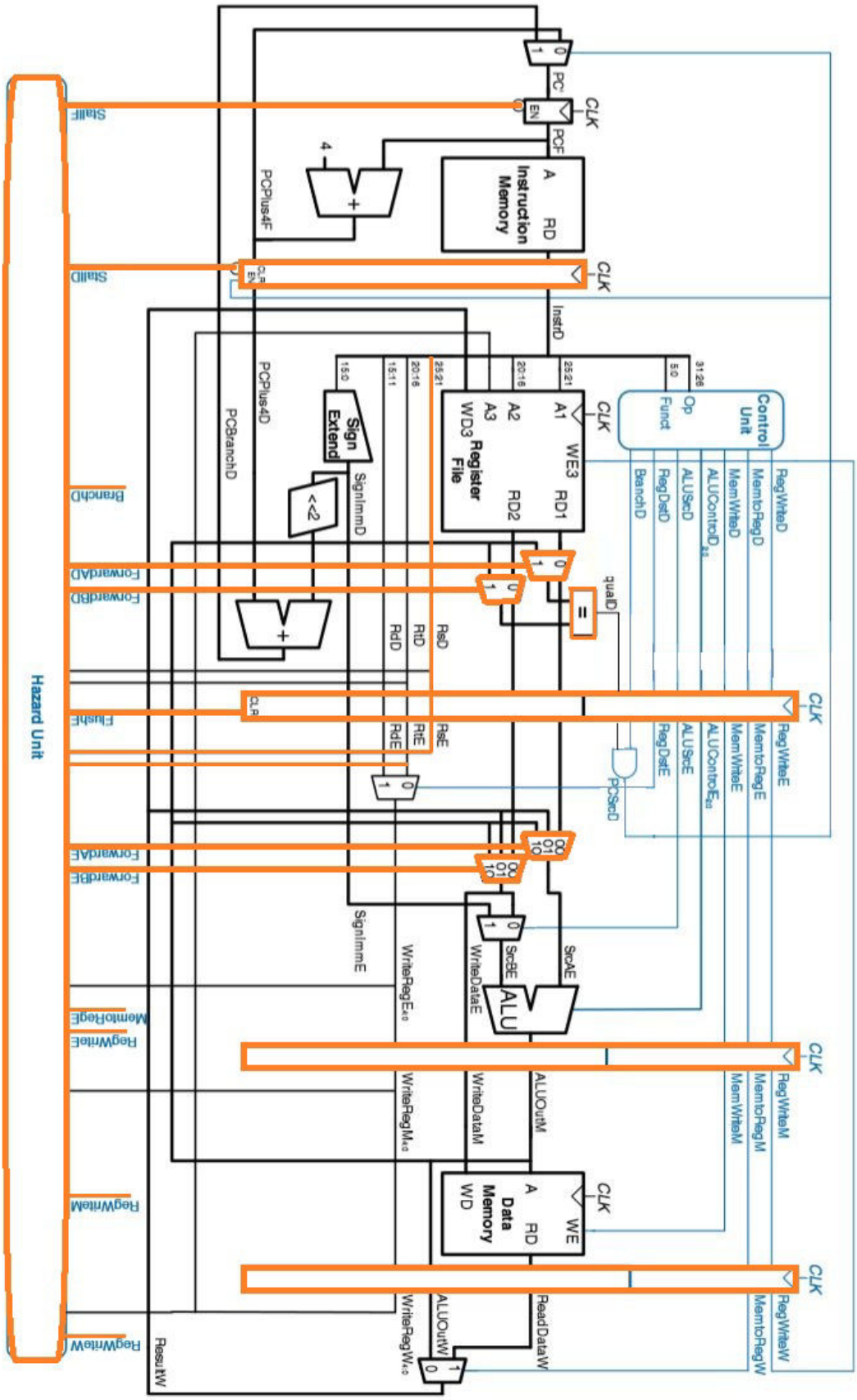
Computer Organization

Preliminary Design Report
Lab-4

Kerem Ayöz
Section 03 / 21501569

14/11/2017

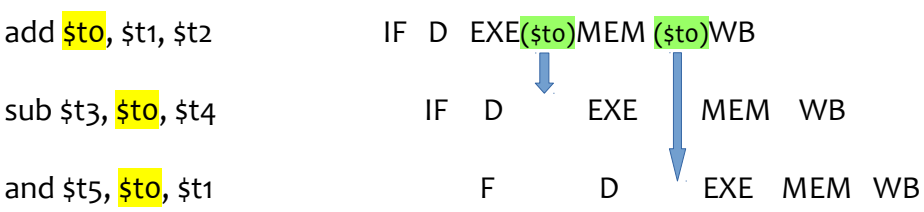
Part b


Hazard Unit
StallF
StallD
FlushE
ForwardAD
ForwardBD
ForwardAE
ForwardBE
MemtoRegE
RegWriteE
RegWriteM
RegWriteW
PCPlus4F
PCPlus4D
PCBranchD
BranchD
InstrD
Instruction Memory
Register File
Sign Extend
SignImmD
SignImmE
Control Unit
RegWriteD
MemtoRegD
MemWriteD
ALUControlD
ALUSrcD
RegDstD
BranchD
RegWriteE
MemtoRegE
MemWriteE
ALUControlE
ALUSrcE
RegDstE
PCSrcD
RsD
RtD
RdD
RsE
RtE
RdE
qualD
SrcAE
SrcBE
ALU
WriteDataE
WriteRegE
ALUOutM
WriteDataM
WriteRegM
Data Memory
ReadDataW
ALUOutW
WriteRegW
ResultW

Part c

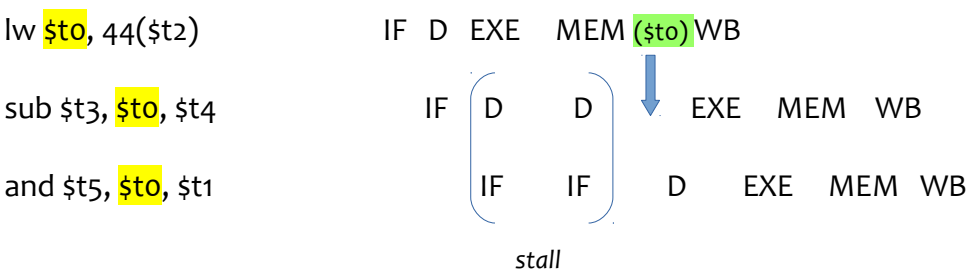
RAW(Read after Write) **Hazards**

1- Compute-use



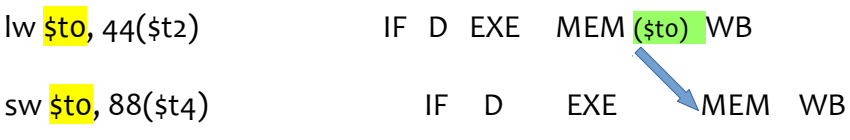
Here tha *sub* and *and* instructions try to read the value in $to before the add instruction's write operation on $to completed. This cause RAW hazard and could be solved by forwarding.

2- Load-use



In that type of situations, forwarding is not enough to solve hazard because the data that the sub and and instruction needs could not write before the time that sub and instructions are in the decode/fetch stage. Therefore next 2 instructions should be stalled and single forwarding to the sub instruction is enough to solve that type of hazard.

3- Load-store



In that situation sw need the data in the $to before it goes into the memory stage. However lw instruction completes the writing after the memory stage. Therefore single forwarding is enough for handling the hazard.

Control Hazards

1- J-Type Instructions

j *Target* IF D EXE MEM WB

sw $to, 88($t4) ~~IF D EXE MEM WB~~

.

.

Target: …

The jump instruction changes the PC and because the address calculation occurs in the decode stage, it should clear whatever the next instruction is. So flushing one instruction is enough for that hazard.

2- Branch Instruction

beq $t1, $to, *Target* IF D EXE MEM WB

sw $to, 88($t4) ~~IF D EXE MEM WB~~

addi $to, $t4, 102 ~~IF D EXE MEM WB~~

.

.

Target: …

As like the jump instruction, next instruction should be cleared but this time if this operation is needed if the branch is taken. Since BTA calculation is done at execute stage, the instructions at decode and fetch stage should be cleared thus 2 flushes needed to handle that type of hazards.

Part d

RAW(Read after Write) Hazards

1- Compute-use

Logic equation

if ((rsE != 0) AND (rsE = WriteRegM) AND RegWriteM) then
ForwardAE = 10
else if ((rsE != 0) AND (rsE = WriteRegW) AND RegWriteW) then
ForwardAE = 01
else ForwardAE 00

2- Load-use

Logic equation

lwstall = ((rsD rtE) OR (rtD rtE)) AND MemtoRegE
StallF = StallD = FlushE = lwstall

3- Load-store

Logic equation

if ((rsE ! 0) AND (rsE WriteRegM) AND RegWriteM) then
ForwardAE = 10
else if ((rsE ! 0) AND (rsE WriteRegW) AND RegWriteW) then
ForwardAE = 01
else ForwardAE = 00

Control Hazards

1- J-Type Instructions

Logic equation

if (JumpD == 1) then
FlushE = 1
else
FlushE = 0

2- Branch Instruction

Logic equation

ForwardAD (rsD ! 0) AND (rsD WriteRegM) AND RegWriteM
ForwardBD (rtD ! 0) AND (rtD WriteRegM) AND RegWriteM

branchstall
BranchD AND RegWriteE AND (WriteRegE rsD OR WriteRegE rtD)
OR
BranchD AND MemtoRegM AND (WriteRegM rsD OR WriteRegM rtD)

StallF StallD FlushE lwstall OR branchstall

Part e

Modules that need to be changed

- Pipeline registers for Fetch, Decode, Execute and Memory stages are added. Parametrized registers are used for every input.
- Equality comparator module added to the decode stage for branch decision.
- Hazard unit and forwarding muxes added.
- 3to1 mux module added to execute stage for forwarding.
- Branch decision AND gate moved to execute stage with PcSrc Signal.
- Zero output removed from ALU.
- Many wires added to support new modules and connections.
- Many wires were renamed to represent stages.

Changes to controller

- Many wires were renamed to represent stages.

Part g

Test program for new instructions

addi $v0 $zero 0x0005

addi $v1 $zero 0x000C

addi $a3 $v1 0xFFF7

or $a0 $a3 $v0

and $a1 $v1 $a0

add $a1 $a1 $a0

beq $a1 $a3 0x000A

slt $a0 $v1 $a0

beq $a0 $zero 0x0001

addi $a1 $zero 0x0000

slt $a0 $a3 $v0

add $a3 $a0 $a1

sub $a3 $a3 $v0

sw $a3 0x0044 $v1

lw $v0 0x0050 $zero

j 0x0000011

addi $v0 $zero 0x0001

sw $v0 0x0054 $zero

lui $t1, 12

addi $t0, $t0, 0x0054

addi $t8, $t8, $0

J 0x0000012

Part f

DATAPATH

```
module datapath (input  logic clk, reset, memtoreg, branch, alusrc, regdst,
                 input  logic regwrite, jump,memwrite,
                        input  logic[2:0]  alucontrol,
                        output logic[31:0] pc,
                    input  logic[31:0] instr,
                 output logic[31:0] aluout, writedata,
                    input  logic[31:0] readdata);

  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh, srca, srcb, result;
  //Pipe logic
  logic useless = 1'b1;
  logic stallD,stallF,pcplus4D,flushE, pcsrc,
                 pcsrcE,regwriteE,memtoregE,memwriteE,alusrcE,regdstE,
                 regwriteM,memtoregM,memwriteM,
                 regwriteW,memtoregW,
                 forwardAD,forwardBD;
  logic[31:0] instrD,source1E,source2E,signImmE,
              writeDataE,aluOutM,writeDataM,
              readDataW,aluOutW,
              srcaComp,srcbComp,
              srcAE,srcBE;
  logic[4:0] rsE,rtE,rdE,writeRegE,writeRegM,writeRegW;
  logic[2:0] alucontrolE;
  logic[1:0] forwardAE,forwardBE;

  //Branch Decision
  mux2 srcaMux(srca,aluOutM,forwardAD,srcaComp);
  mux2 srcbMux(srcb,aluOutM,forwardBD,srcbComp);
  assign pcsrc = branch & (srcaComp == srcbComp);

  // next PC logic
  flopr #(32) pcreg(clk, reset,~stallF, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4D, signimmsh, pcbranch);
  mux2 pcbrmux(pcplus4, pcbranch, pcsrcE,pcnextbr);
  mux2 pcmux(pcnextbr, {pcplus4[31:28],instr[25:0], 2'b00}, jump, pcnext);

  //Hazard Unit
  hazard h(rsD, rtD, rsE, rtE, writeRegE, writeRegM,
           writeRegW,regwriteE, regwriteM, regwriteW,
           memtoregE, memtoregM, branch,
           forwardAD, forwardBD, forwardAE,forwardBE,
           stallF, stallD, flushE);

//Pipe F-D
pipeRegisterFD fdreg (clk,reset,~stallD,(jump |
pcsrcE),instr,pcplus4,instrD,pcplus4D);
```

```
//Pipe D-E
pipeRegisterDE dereg(clk,reset,useless,flushE,srca,writedata,
            instr[25:21],instr[20:16],instr[15:11],
            signimm,source1E,source2E,rsE,rtE,rdE,signImmE,
            //Control Signals
            pcsrc,regwrite,memtoreg,memwrite,alucontrol,regdst,
            pcsrcE,regwriteE,memtoregE,memwriteE,alucontrolE,alusrcE,regdstE);

//Pipe E-M
pipeRegisterEM emreg(clk,reset,useless,~useless,aluOut,
            writeDataE,writeRegE,
            aluOutM,writeDataM,writeRegM,
            //Control Signals
            regwriteE,memtoregE,memwriteE,
            regwriteM,memtoregM,memwriteM);

//Pipe M-W
pipeRegisterMW mwreg(clk,reset,useless,~useless,
                readdata, aluOutM,writeRegM,
            readDataW,aluOutW,writeRegW,
           //Control Signals
           regwriteM,memtoregM,
           regwriteW,memtoregW);

// register file logic
   regfile     rf (clk, regwriteW, instrD[25:21], instrD[20:16], writeRegW,
                result, srca, writedata);
   mux3 alu1(source1E,result,aluOutM,forwardAE,srcAE);
   mux3 alu2(source2E,result,aluOutM,forwardBE,writeDataE);
   mux2p #(5)    wrmux (rtE, rdE, regdstE, writeregE);
   mux2p #(32)  resmux (aluOutW, readDataW, memtoregW, result);
   signext         se (instr[15:0], signimm);
  // ALU logic
   mux2p #(32)  srcbmux (writeDataE, signImmE, alusrcE, srcBE);
   alu         alu (srcAE, srcBE, alucontrolE, aluout);

endmodule

//2-to-1 MUX
module mux2(input  logic[31:0] d0, d1,
          input  logic s,
          output logic[31:0] y);

   assign y = s ? d1 : d0;
endmodule

//3-yo-1 MUX
module mux3(input logic [31:0] d0,d1,d2,
          input logic [1:0] S,
          output logic [31:0] Y
          );
```

```
assign Y = S[1] ? (S[0] ? 32'bX : d2)
 : (S[0] ? d1 : d0);

endmodule
```

PIPELINE REGISTERS

```
//Pipe register between fetch and decode
module pipeRegisterFD
            (input logic clk, reset,en, clear,
                  input logic[31:0] instrF,
             input logic[31:0] pcplus4F,
             output logic[31:0] instrD,
             output logic[31:0] pcplus4D);

  always_ff@(posedge clk, posedge reset)
    if (reset | clear)  begin instrD <= 0; pcplus4D <= 0; end
    else if (en)        begin instrD <= instrF; pcplus4D <= pcplus4F; end
endmodule

//Pipe register between decode and execute
module pipeRegisterDE
            (input logic clk, reset,en, clear,
                  input logic[31:0] source1D,source2D,
             input logic [4:0] rsD,rtD,rdD,
             input logic[31:0] signImmD,
             output logic[31:0] source1E,source2E,
             output logic[4:0] rsE,rtE,rdE,
             output logic[31:0] signImmE,
             //Control Signals
            input logic pcsrcD,
            input logic regwriteD,memtoregD,memwriteD,
            input logic[2:0] alucontrolD,
            input logic alusrcD, regdstD,
            output logic pcsrcE,
            output logic regwriteE,memtoregE,memwriteE,
            output logic[2:0] alucontrolE,
            output logic alusrcE, regdstE);

  always_ff@(posedge clk, posedge reset)
    if (reset | clear) begin
       source1E <= 0; source2E <= 0;
       rsE <= 0; rtE <= 0; rdE <= 0;
       signImmE <= 0;
       //Signals
       pcsrcE <= 0; regwriteE <= 0;
       memtoregE <= 0; memwriteE <= 0;  alucontrolE <= 0;
       alusrcE <= 0; regdstE <= 0;
       end
    else if (en) begin
       source1E <= source1D; source2E <= source2D;
       rsE <= rsD; rtE <= rtD; rdE <= rdD;
```

```
        signImmE <= signImmD;
        //Signals
        pcsrcE <= pcsrcD; regwriteE <= regwriteD;
        memtoregE <= memtoregD; memwriteE <= memwriteD;  alucontrolE <=
alucontrolD;
        alusrcE <= alusrcD; regdstE <= regdstD;
        end
endmodule

//Pipe register between execute and memory
module pipeRegisterEM
            (input logic clk, reset,en, clear,
                 input logic[31:0] aluOutE,writeDataE,
             input logic [4:0] writeRegE,
             output logic[31:0] aluOutM,writeDataM,
             output logic[4:0] writeRegM,
             //Control Signals
             input logic regwriteE,memtoregE,memwriteE,
             output logic regwriteM,memtoregM,memwriteM);

  always_ff@(posedge clk, posedge reset)
    if (reset | clear) begin
        aluOutM <= 0; writeDataM <= 0;
        writeRegM <= 0;
        //Signals
        regwriteM <= 0; memtoregM <= 0; memwriteM <= 0;
        end
    else if (en) begin
        aluOutM <= aluOutE; writeDataM <= writeDataE;
        writeRegM <= writeRegE;
        //Signals
        regwriteM <= regwriteE; memtoregM <= memtoregE; memwriteM <=
memwriteE;
        end
endmodule

//Pipe register between memory and writeback
module pipeRegisterMW
            (input logic clk, reset,en, clear,
                 input logic[31:0] readDataM,
             input logic[31:0] aluOutM,
             input logic [4:0] writeRegM,
             output logic[31:0] readDataW,
             output logic[31:0] aluOutW,
             output logic[4:0] writeRegW,
            //Control Signals
            input logic regwriteM,memtoregM,
            output logic regwriteW,memtoregW);

  always_ff@(posedge clk, posedge reset)
    if (reset | clear) begin
        readDataW <= 0; aluOutW <= 0;
```

```
        writeRegW <= 0;
        //Signals
        regwriteW <= 0; memtoregW <= 0;
        end
    else if (en) begin
        readDataW <= readDataM; aluOutW <= aluOutM;
        writeRegW <= writeRegM;
        //Signals
        regwriteW <= regwriteM; memtoregW <= memtoregM;
        end
endmodule
```

HAZARD UNIT

```
module hazard(input logic [4:0] rsD, rtD, rsE, rtE,
              input logic [4:0] writeregE, writeregM, writeregW,
              input logic regwriteE, regwriteM, regwriteW,
              input logic memtoregE, memtoregM, branchD,
              output logic forwardaD, forwardbD,
              output logic [1:0] forwardaE, forwardbE,
              output logic stallF, stallD, flushE);

logic lwstallD, branchstallD;

// forwarding sources to D stage (branch equality)
assign forwardaD = (rsD !=0 & rsD == writeregM & regwriteM);
assign forwardbD = (rtD !=0 & rtD == writeregM & regwriteM);

// forwarding sources to E stage (ALU)
always_comb
begin
    forwardaE = 2'b00; forwardbE = 2'b00;
    if (rsE != 0)
    if (rsE == writeregM & regwriteM)
    forwardaE = 2'b10;
    else if (rsE == writeregW & regwriteW)
    forwardaE = 2'b01;
    if (rtE != 0)
    if (rtE == writeregM & regwriteM)
    forwardbE = 2'b10;
    else if (rtE == writeregW & regwriteW)
    forwardbE = 2'b01;
end
// stalls
assign #1 lwstallD = memtoregE &
(rtE == rsD | rtE == rtD);
assign #1 branchstallD = branchD &
(regwriteE &
(writeregE == rsD | writeregE == rtD) |
memtoregM &
(writeregM == rsD | writeregM == rtD));
assign #1 stallD = lwstallD | branchstallD;
assign #1 stallF = stallD;
```

```
// stalling D stalls all previous stages
assign #1 flushE = stallD;
// stalling D flushes next stage
// Note: not necessary to stall D stage on store
// if source comes from load;
// instead, another bypass network could
// be added from W to M
endmodule
```

CONTROLLER

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

   logic [1:0] aluop;

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

   aludec  ad (funct, aluop, alucontrol);


endmodule
```

MIPS PIPELINED PROCESSOR

```
// 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, alusrc, regdst, regwrite, jump;
  logic [2:0]  alucontrol;

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

  datapath dp (clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump,
memwrite,
               alucontrol, pc, instr, aluout, writedata, readdata);

endmodule


module maindec (input logic[5:0] op,
                output logic memtoreg, memwrite, branch,
                output logic alusrc, regdst, regwrite, jump,
```

```
                    output logic[1:0] aluop );
   logic [8:0] controls;

   assign {regwrite, regdst, alusrc, branch, memwrite,
             memtoreg,  aluop, jump} = controls;
  always_comb
    case(op)
      6'b000000: controls <= 9'b110000100; // R-type
      6'b100011: controls <= 9'b101001000; // LW
      6'b101011: controls <= 9'b001010000; // SW
      6'b000100: controls <= 9'b000100010; // BEQ
      6'b001000: controls <= 9'b101000000; // ADDI
      6'b000010: controls <= 9'b000000001; // J
      default:   controls <= 9'bxxxxxxxxx; // illegal op
    endcase
endmodule
```

ALUDECODER

```
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)
      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
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, posedge we3)
     if (we3)
         rf [wa3] <= wd3;
  assign rd1 = (ra1 != 0) ? rf [ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ ra2] : 0;
endmodule
```

ALU

```
module alu(input  logic [31:0] A, B,
           input  logic [2:0]  F,
           output logic [31:0] Z);
              logic [31:0] Y;
              always_comb
                 case (F)
                 3'b000: Y = A & B;
                 3'b001: Y = A | B;
                 3'b010: Y = A + B;
                 3'b100: Y = A & ~B;
                 3'b101: Y = A | ~B;
                 3'b110: Y = A - B;
                 3'b111: Y = (A < B) + 32'b0000;
                 endcase
                 assign Z = Y;
endmodule
```

ADDER

```
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
```

D FLIP-FLOP

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

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

MUX 2-1

```
// paramaterized 2-to-1 MUX
module mux2p #(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
```

TOP

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

   logic [31:0] pc, instr, readdata;

   // 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
```

TEST BENCH

```
module testbench();
        reg clk;
        reg rst;
    reg [31:0] writeData;
    reg [31:0] dataAddress;
    reg memWrite;

        top proc(clk,rst,writeData,dataAddress,memWrite);

        always
                #5 clk = ~clk;
        initial
        begin
                // time = 0
                clk = 1'b0;
                // Reset CPU
                rst = 1'b1;
                // run 1st iteration to reset cpu, and load first instruction
                @(posedge clk);

                // set Rest to 0
                rst = 1'b0;

                // Run through 5 CPU cycles
     repeat(40)
                        @(posedge clk);

                $finish();
        end

endmodule
```