Exercise 7.39

Suppose the ARM pipelined processor is divided into 10 stages of 400 ps each, including sequencing overhead. Assume the instruction mix of Example 7.7. Also assume that 50% of the loads are immediately followed by an instruction that uses the result, requiring six stalls, and that 30% of the branches are mispredicted. The target address of a branch instruction is not computed until the end of the second stage. Calculate the average CPI and execution time of computing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor.

CPI = 0.25(1+0.5*6) + 0.1(1) + 0.13(1+0.3*1)+0.52(1) = 1.789 ≈ **1.8**
Execution Time = (100 x 10^9 instructions)(1.789 cycles/instruction)(400 x10^{-12} s/cycle) = 71.56 s ≈ **72 s**

Exercise 7.40

SystemVerilog

```
// ARM pipelined processor
module testbench();

  logic        clk;
  logic        reset;

  logic [31:0] WriteData, DataAdr;
  logic        MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, MemWrite);

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

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

  // check results
  always @(negedge clk)
    begin
      if(MemWrite) begin
        if(DataAdr === 100 & WriteData === 7) begin
          $display("Simulation succeeded");
```

```
          $stop;
        end else if (DataAdr !== 96) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule

module top(input  logic        clk, reset,
           output logic [31:0] WriteDataM, DataAdrM,
           output logic        MemWriteM);

  logic [31:0] PCF, InstrF, ReadDataM;

  // instantiate processor and memories
  arm arm(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
          WriteDataM, ReadDataM);
  imem imem(PCF, InstrF);
  dmem dmem(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
endmodule

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

  logic [31:0] RAM[2097151:0];

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

  assign rd = RAM[a[22:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[22:2]] <= wd;
endmodule

module imem(input  logic [31:0] a,
            output logic [31:0] rd);

  logic [31:0] RAM[2097151:0];

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

  assign rd = RAM[a[22:2]]; // word aligned
endmodule

module arm(input  logic        clk, reset,
           output logic [31:0] PCF,
           input  logic [31:0] InstrF,
           output logic        MemWriteM,
           output logic [31:0] ALUOutM, WriteDataM,
           input  logic [31:0] ReadDataM);
```

```
  logic [1:0]  RegSrcD, ImmSrcD, ALUControlE;
  logic        ALUSrcE, BranchTakenE, MemtoRegW, PCSrcW, RegWriteW;
  logic [3:0]  ALUFlagsE;
  logic [31:0] InstrD;
  logic        RegWriteM, MemtoRegE, PCWrPendingF;
  logic [1:0]  ForwardAE, ForwardBE;
  logic        StallF, StallD, FlushD, FlushE;
  logic        Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W,
Match_12D_E;

  controller c(clk, reset, InstrD[31:12], ALUFlagsE,
               RegSrcD, ImmSrcD,
               ALUSrcE, BranchTakenE, ALUControlE,
               MemWriteM,
               MemtoRegW, PCSrcW, RegWriteW,
               RegWriteM, MemtoRegE, PCWrPendingF,
               FlushE);
  datapath dp(clk, reset,
              RegSrcD, ImmSrcD,
              ALUSrcE, BranchTakenE, ALUControlE,
              MemtoRegW, PCSrcW, RegWriteW,
              PCF, InstrF, InstrD,
              ALUOutM, WriteDataM, ReadDataM,
              ALUFlagsE,
              Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W,
Match_12D_E,
              ForwardAE, ForwardBE, StallF, StallD, FlushD);
  hazard h(clk, reset, Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W,
Match_12D_E,
           RegWriteM, RegWriteW, BranchTakenE, MemtoRegE,
           PCWrPendingF, PCSrcW,
           ForwardAE, ForwardBE,
           StallF, StallD, FlushD, FlushE);

endmodule

module controller(input  logic         clk, reset,
                  input  logic [31:12] InstrD,
                  input  logic [3:0]   ALUFlagsE,
                  output logic [1:0]   RegSrcD, ImmSrcD,
                  output logic         ALUSrcE, BranchTakenE,
                  output logic [1:0]   ALUControlE,
                  output logic         MemWriteM,
                  output logic         MemtoRegW, PCSrcW, RegWriteW,
                  // hazard interface
                  output logic         RegWriteM, MemtoRegE,
                  output logic         PCWrPendingF,
                  input  logic         FlushE);

  logic [9:0] controlsD;
  logic       CondExE, ALUOpD;
  logic [1:0] ALUControlD;
  logic       ALUSrcD;
```

```
  logic       MemtoRegD, MemtoRegM;
  logic       RegWriteD, RegWriteE, RegWriteGatedE;
  logic       MemWriteD, MemWriteE, MemWriteGatedE;
  logic       BranchD, BranchE;
  logic [1:0] FlagWriteD, FlagWriteE;
  logic       PCSrcD, PCSrcE, PCSrcM;
  logic [3:0] FlagsE, FlagsNextE, CondE;

  // Decode stage

  always_comb
      casex(InstrD[27:26])
        2'b00: if (InstrD[25]) controlsD = 10'b0000101001; // DP imm
               else            controlsD = 10'b0000001001; // DP reg
        2'b01: if (InstrD[20]) controlsD = 10'b0001111000; // LDR
               else            controlsD = 10'b1001110100; // STR
        2'b10:                 controlsD = 10'b0110100010; // B
        default:               controlsD = 10'bx;          //
unimplemented
      endcase

  assign {RegSrcD, ImmSrcD, ALUSrcD, MemtoRegD,
          RegWriteD, MemWriteD, BranchD, ALUOpD} = controlsD;

  always_comb
    if (ALUOpD) begin                 // which Data-processing Instr?
      case(InstrD[24:21])
          4'b0100: ALUControlD = 2'b00; // ADD
          4'b0010: ALUControlD = 2'b01; // SUB
        4'b0000: ALUControlD = 2'b10; // AND
          4'b1100: ALUControlD = 2'b11; // ORR
          default: ALUControlD = 2'bx;  // unimplemented
      endcase
      FlagWriteD[1]   = InstrD[20];   // update N and Z Flags if S bit is
set
      FlagWriteD[0]   = InstrD[20] & (ALUControlD == 2'b00 | ALUControlD
== 2'b01);
    end else begin
      ALUControlD     = 2'b00;        // perform addition for non-
dataprocessing instr
      FlagWriteD      = 2'b00;        // don't update Flags
    end

  assign PCSrcD       = (((InstrD[15:12] == 4'b1111) & RegWriteD) |
BranchD);

  // Execute stage
  floprc #(7) flushedregsE(clk, reset, FlushE,
                           {FlagWriteD, BranchD, MemWriteD, RegWriteD,
PCSrcD, MemtoRegD},
                           {FlagWriteE, BranchE, MemWriteE, RegWriteE,
PCSrcE, MemtoRegE});
  flopr #(3)  regsE(clk, reset,
                    {ALUSrcD, ALUControlD},
```

```
                    {ALUSrcE, ALUControlE});

  flopr  #(4) condregE(clk, reset, InstrD[31:28], CondE);
  flopr  #(4) flagsreg(clk, reset, FlagsNextE, FlagsE);

  // write and Branch controls are conditional
  conditional Cond(CondE, FlagsE, ALUFlagsE, FlagWriteE, CondExE,
FlagsNextE);
  assign BranchTakenE    = BranchE & CondExE;
  assign RegWriteGatedE  = RegWriteE & CondExE;
  assign MemWriteGatedE  = MemWriteE & CondExE;
  assign PCSrcGatedE     = PCSrcE & CondExE;

  // Memory stage
  flopr #(4) regsM(clk, reset,
                   {MemWriteGatedE, MemtoRegE, RegWriteGatedE,
PCSrcGatedE},
                   {MemWriteM, MemtoRegM, RegWriteM, PCSrcM});

  // Writeback stage
  flopr #(3) regsW(clk, reset,
                   {MemtoRegM, RegWriteM, PCSrcM},
                   {MemtoRegW, RegWriteW, PCSrcW});

  // Hazard Prediction
  assign PCWrPendingF = PCSrcD | PCSrcE | PCSrcM;

endmodule

module conditional(input  logic [3:0] Cond,
                   input  logic [3:0] Flags,
                   input  logic [3:0] ALUFlags,
                   input  logic [1:0] FlagsWrite,
                   output logic       CondEx,
                   output logic [3:0] FlagsNext);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
      4'b0110: CondEx = overflow;         // VS
      4'b0111: CondEx = ~overflow;        // VC
      4'b1000: CondEx = carry & ~zero;    // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
```

```
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // Always
      default: CondEx = 1'bx;             // undefined
    endcase

  assign FlagsNext[3:2] = (FlagsWrite[1] & CondEx) ? ALUFlags[3:2] :
Flags[3:2];
  assign FlagsNext[1:0] = (FlagsWrite[0] & CondEx) ? ALUFlags[1:0] :
Flags[1:0];
endmodule

module datapath(input  logic        clk, reset,
                input  logic [1:0]  RegSrcD, ImmSrcD,
                input  logic        ALUSrcE, BranchTakenE,
                input  logic [1:0]  ALUControlE,
                input  logic        MemtoRegW, PCSrcW, RegWriteW,
                output logic [31:0] PCF,
                input  logic [31:0] InstrF,
                output logic [31:0] InstrD,
                output logic [31:0] ALUOutM, WriteDataM,
                input  logic [31:0] ReadDataM,
                output logic [3:0]  ALUFlagsE,
                // hazard logic
                output logic        Match_1E_M, Match_1E_W, Match_2E_M,
Match_2E_W, Match_12D_E,
                input  logic [1:0]  ForwardAE, ForwardBE,
                input  logic        StallF, StallD, FlushD);


  logic [31:0] PCPlus4F, PCnext1F, PCnextF;
  logic [31:0] ExtImmD, rd1D, rd2D, PCPlus8D;
  logic [31:0] rd1E, rd2E, ExtImmE, SrcAE, SrcBE, WriteDataE, ALUResultE;
  logic [31:0] ReadDataW, ALUOutW, ResultW;
  logic [3:0]  RA1D, RA2D, RA1E, RA2E, WA3E, WA3M, WA3W;
  logic        Match_1D_E, Match_2D_E;

  // Fetch stage
  mux2 #(32) pcnextmux(PCPlus4F, ResultW, PCSrcW, PCnext1F);
  mux2 #(32) branchmux(PCnext1F, ALUResultE, BranchTakenE, PCnextF);
  flopenr #(32) pcreg(clk, reset, ~StallF, PCnextF, PCF);
  adder #(32) pcadd(PCF, 32'h4, PCPlus4F);

  // Decode Stage
  assign PCPlus8D = PCPlus4F; // skip register
  flopenrc #(32) instrreg(clk, reset, ~StallD, FlushD, InstrF, InstrD);
  mux2 #(4)   ra1mux(InstrD[19:16], 4'b1111, RegSrcD[0], RA1D);
  mux2 #(4)   ra2mux(InstrD[3:0], InstrD[15:12], RegSrcD[1], RA2D);
  regfile     rf(clk, RegWriteW, RA1D, RA2D,
                 WA3W, ResultW, PCPlus8D,
                 rd1D, rd2D);
  extend      ext(InstrD[23:0], ImmSrcD, ExtImmD);
```

```
  // Execute Stage
  flopr #(32) rd1reg(clk, reset, rd1D, rd1E);
  flopr #(32) rd2reg(clk, reset, rd2D, rd2E);
  flopr #(32) immreg(clk, reset, ExtImmD, ExtImmE);
  flopr #(4)  wa3ereg(clk, reset, InstrD[15:12], WA3E);
  flopr #(4)  ra1reg(clk, reset, RA1D, RA1E);
  flopr #(4)  ra2reg(clk, reset, RA2D, RA2E);
  mux3 #(32)  byp1mux(rd1E, ResultW, ALUOutM, ForwardAE, SrcAE);
  mux3 #(32)  byp2mux(rd2E, ResultW, ALUOutM, ForwardBE, WriteDataE);
  mux2 #(32)  srcbmux(WriteDataE, ExtImmE, ALUSrcE, SrcBE);
  alu         alu(SrcAE, SrcBE, ALUControlE, ALUResultE, ALUFlagsE);

  // Memory Stage
  flopr #(32) aluresreg(clk, reset, ALUResultE, ALUOutM);
  flopr #(32) wdreg(clk, reset, WriteDataE, WriteDataM);
  flopr #(4)  wa3mreg(clk, reset, WA3E, WA3M);

  // Writeback Stage
  flopr #(32) aluoutreg(clk, reset, ALUOutM, ALUOutW);
  flopr #(32) rdreg(clk, reset, ReadDataM, ReadDataW);
  flopr #(4)  wa3wreg(clk, reset, WA3M, WA3W);
  mux2 #(32)  resmux(ALUOutW, ReadDataW, MemtoRegW, ResultW);

  // hazard comparison
  eqcmp #(4) m0(WA3M, RA1E, Match_1E_M);
  eqcmp #(4) m1(WA3W, RA1E, Match_1E_W);
  eqcmp #(4) m2(WA3M, RA2E, Match_2E_M);
  eqcmp #(4) m3(WA3W, RA2E, Match_2E_W);
  eqcmp #(4) m4a(WA3E, RA1D, Match_1D_E);
  eqcmp #(4) m4b(WA3E, RA2D, Match_2D_E);
  assign Match_12D_E = Match_1D_E | Match_2D_E;

endmodule

module hazard(input  logic       clk, reset,
              input  logic       Match_1E_M, Match_1E_W, Match_2E_M,
Match_2E_W, Match_12D_E,
              input  logic       RegWriteM, RegWriteW,
              input  logic       BranchTakenE, MemtoRegE,
              input  logic       PCWrPendingF, PCSrcW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic       StallF, StallD,
              output logic       FlushD, FlushE);

  logic ldrStallD;

  // forwarding logic
  always_comb begin
    if (Match_1E_M & RegWriteM)      ForwardAE = 2'b10;
    else if (Match_1E_W & RegWriteW) ForwardAE = 2'b01;
    else                             ForwardAE = 2'b00;

    if (Match_2E_M & RegWriteM)      ForwardBE = 2'b10;
    else if (Match_2E_W & RegWriteW) ForwardBE = 2'b01;
```

```
    else                              ForwardBE = 2'b00;
  end

  // stalls and flushes
  // Load RAW
  //   when an instruction reads a register loaded by the previous,
  //   stall in the decode stage until it is ready
  // Branch hazard
  //   When a branch is taken, flush the incorrectly fetched instrs
  //   from decode and execute stages
  // PC Write Hazard
  //   When the PC might be written, stall all following instructions
  //   by stalling the fetch and flushing the decode stage
  // when a stage stalls, stall all previous and flush next

  assign ldrStallD = Match_12D_E & MemtoRegE;

  assign StallD = ldrStallD;
  assign StallF = ldrStallD | PCWrPendingF;
  assign FlushE = ldrStallD | BranchTakenE;
  assign FlushD = PCWrPendingF | PCSrcW | BranchTakenE;

endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [3:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3, r15,
               output logic [31:0] rd1, rd2);

  logic [31:0] rf[14:0];

  // three ported register file
  // read two ports combinationally
  // write third port on falling edge of clock (midcycle)
  //   so that writes can be read on same cycle
  // register 15 reads PC+8 instead

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

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule

module extend(input  logic [23:0] Instr,
              input  logic [1:0]  ImmSrc,
              output logic [31:0] ExtImm);

  always_comb
    case(ImmSrc)
      2'b00:   ExtImm = {24'b0, Instr[7:0]};  // 8-bit unsigned immediate
      2'b01:   ExtImm = {20'b0, Instr[11:0]}; // 12-bit unsigned immediate
      2'b10:   ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00}; // Branch
```

```
      default: ExtImm = 32'bx; // undefined
    endcase
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [1:0]  ALUControl,
           output logic [31:0] Result,
           output logic [3:0]  Flags);

  logic        neg, zero, carry, overflow;
  logic [31:0] condinvb;
  logic [32:0] sum;

  assign condinvb = ALUControl[0] ? ~b : b;
  assign sum = a + condinvb + ALUControl[0];

  always_comb
    casex (ALUControl[1:0])
      2'b0?: Result = sum;
      2'b10: Result = a & b;
      2'b11: Result = a | b;
    endcase

  assign neg      = Result[31];
  assign zero     = (Result == 32'b0);
  assign carry    = (ALUControl[1] == 1'b0) & sum[32];
  assign overflow = (ALUControl[1] == 1'b0) & ~(a[31] ^ b[31] ^
ALUControl[0]) &
                                              (a[31] ^ sum[31]);
  assign Flags = {neg, zero, carry, overflow};
endmodule

module adder #(parameter WIDTH=8)
              (input  logic [WIDTH-1:0] a, b,
               output logic [WIDTH-1:0] y);

  assign y = a + b;
endmodule

module flopenr #(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

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

module flopenrc #(parameter WIDTH = 8)
                 (input  logic             clk, reset, en, clear,
                  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)
      if (clear) q <= 0;
      else       q <= d;
endmodule

module floprc #(parameter WIDTH = 8)
               (input  logic             clk, reset, clear,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

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

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

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

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

module eqcmp #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] a, b,
              output logic             y);

  assign y = (a == b);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:           in  STD_LOGIC;
         WriteDataM, DataAdrM: out STD_LOGIC_VECTOR(31 downto 0);
         MemWriteM:            out STD_LOGIC);
  end component;
  signal WriteData, DataAdr:     STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, MemWrite:   STD_LOGIC;
begin

  -- instantiate device to be tested
  dut: top port map(clk, reset, WriteData, DataAdr, MemWrite);

  -- Generate clock with 10 ns period
  process begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;

  -- Generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 7 gets written to address 84
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and MemWrite = '1') then
      if (to_integer(DataAdr) = 100 and
          to_integer(WriteData) = 7) then
        report "NO ERRORS: Simulation succeeded" severity failure;
      elsif (DataAdr /= 96) then
        report "Simulation failed" severity failure;
      end if;
    end if;
  end process;
end;


library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD_UNSIGNED.all;
```

```
entity top is -- top-level design for testing
  port(clk, reset:           in     STD_LOGIC;
       WriteDataM, DataAdrM: buffer STD_LOGIC_VECTOR(31 downto 0);
       MemWriteM:            buffer STD_LOGIC);
end;

architecture test of top is
  component arm
    port(clk, reset:          in  STD_LOGIC;
         PCF:                 out STD_LOGIC_VECTOR(31 downto 0);
         InstrF:              in  STD_LOGIC_VECTOR(31 downto 0);
         MemWriteM:           out STD_LOGIC;
         ALUOutM, WriteDataM: out STD_LOGIC_VECTOR(31 downto 0);
         ReadDataM:           in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
    port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
         rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
    port(clk, we:  in  STD_LOGIC;
         a, wd:    in  STD_LOGIC_VECTOR(31 downto 0);
         rd:       out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal PCF, InstrF, ReadDataM: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  i_arm: arm port map(clk, reset, PCF, InstrF, MemWriteM, DataAdrM,
                      WriteDataM, ReadDataM);
  i_imem: imem port map(PCF, InstrF);
  i_dmem: dmem port map(clk, MemWriteM, DataAdrM, WriteDataM, ReadDataM);
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity imem is -- instruction memory
  port(a:  in  STD_LOGIC_VECTOR(31 downto 0);
       rd: out STD_LOGIC_VECTOR(31 downto 0));
end;
architecture behave of imem is -- instruction memory
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable i, index, result: integer;
    type ramtype is array (63 downto 0) of
                    STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(i) := (others => '0');
```

```
    end loop;
    index := 0;
    FILE_OPEN(mem_file, "memfile.dat", READ_MODE);
    while not endfile(mem_file) loop
      readline(mem_file, L);
      result := 0;
      for i in 1 to 8 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
            result := character'pos(ch) - character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
           result := character'pos(ch) - character'pos('a')+10;
        elsif 'A' <= ch and ch <= 'F' then
           result := character'pos(ch) - character'pos('A')+10;
        else report "Format error on line " & integer'image(index)
             severity error;
        end if;
        mem(index)(35-i*4 downto 32-i*4) :=
          to_std_logic_vector(result,4);
      end loop;
      index := index + 1;
    end loop;

    -- read memory
    loop
      rd <= mem(to_integer(a(7 downto 2)));
      wait on a;
    end loop;
  end process;
end;

library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity dmem is -- data memory
  port(clk, we:  in STD_LOGIC;
       a, wd:    in STD_LOGIC_VECTOR(31 downto 0);
       rd:       out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array (63 downto 0) of
                    STD_LOGIC_VECTOR(31 downto 0);
    variable mem: ramtype;
  begin -- read or write memory
    loop
      if clk'event and clk = '1' then
          if (we = '1') then
            mem(to_integer(a(7 downto 2))) := wd;
          end if;
      end if;
      rd <= mem(to_integer(a(7 downto 2)));
```

```
      wait on clk, a;
    end loop;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity arm is -- pipelined processor
  port(clk, reset:          in  STD_LOGIC;
       PCF:                 out STD_LOGIC_VECTOR(31 downto 0);
       InstrF:              in  STD_LOGIC_VECTOR(31 downto 0);
       MemWriteM:           out STD_LOGIC;
       ALUOutM, WriteDataM: out STD_LOGIC_VECTOR(31 downto 0);
       ReadDataM:           in  STD_LOGIC_VECTOR(31 downto 0));
end;

architecture struct of arm is
  component controller
    port(clk, reset:        in  STD_LOGIC;
         InstrD:            in  STD_LOGIC_VECTOR(31 downto 12);
         ALUFlagsE:         in  STD_LOGIC_VECTOR(3 downto 0);
         RegSrcD, ImmSrcD:  out STD_LOGIC_VECTOR(1 downto 0);
         ALUSrcE:           out STD_LOGIC;
         BranchTakenE:      out STD_LOGIC;
         ALUControlE:       out STD_LOGIC_VECTOR(1 downto 0);
         MemWriteM:         out STD_LOGIC;
         MemtoRegW:         out STD_LOGIC;
         PCSrcW:            out STD_LOGIC;
         RegWriteW:         out STD_LOGIC;
         -- hazard interface
         RegWriteM:         out STD_LOGIC;
         MemtoRegE:         out STD_LOGIC;
         PCWrPendingF:      out STD_LOGIC;
         FlushE:            in  STD_LOGIC);
  end component;
  component datapath
    port(clk, reset:        in  STD_LOGIC;
         RegSrcD, ImmSrcD:  in  STD_LOGIC_VECTOR(1 downto 0);
         ALUSrcE:           in  STD_LOGIC;
         BranchTakenE:      in  STD_LOGIC;
         ALUControlE:       in  STD_LOGIC_VECTOR(1 downto 0);
         MemtoRegW:         in  STD_LOGIC;
         PCSrcW:            in  STD_LOGIC;
         RegWriteW:         in  STD_LOGIC;
         PCF:               out STD_LOGIC_VECTOR(31 downto 0);
         InstrF:            in  STD_LOGIC_VECTOR(31 downto 0);
         InstrD:            out STD_LOGIC_VECTOR(31 downto 0);
         ALUOutM:           out STD_LOGIC_VECTOR(31 downto 0);
         WriteDataM:        out STD_LOGIC_VECTOR(31 downto 0);
         ReadDataM:         in  STD_LOGIC_VECTOR(31 downto 0);
         ALUFlagsE:         out STD_LOGIC_VECTOR(3 downto 0);
         -- hazard logic
         Match_1E_M:        out STD_LOGIC;
         Match_1E_W:        out STD_LOGIC;
         Match_2E_M:        out STD_LOGIC;
```

```
        Match_2E_W:        out STD_LOGIC;
        Match_12D_E:       out STD_LOGIC;
        ForwardAE:         in  STD_LOGIC_VECTOR(1 downto 0);
        ForwardBE:         in  STD_LOGIC_VECTOR(1 downto 0);
        StallF:            in  STD_LOGIC;
        StallD:            in  STD_LOGIC;
        FlushD:            in  STD_LOGIC);
  end component;
  component hazard
    port(clk, reset:       in  STD_LOGIC;
        Match_1E_M:        in  STD_LOGIC;
        Match_1E_W:        in  STD_LOGIC;
        Match_2E_M:        in  STD_LOGIC;
        Match_2E_W:        in  STD_LOGIC;
        Match_12D_E:       in  STD_LOGIC;
        RegWriteM:         in  STD_LOGIC;
        RegWriteW:         in  STD_LOGIC;
        BranchTakenE:      in  STD_LOGIC;
        MemtoRegE:         in  STD_LOGIC;
        PCWrPendingF:      in  STD_LOGIC;
        PCSrcW:            in  STD_LOGIC;
        ForwardAE:         out STD_LOGIC_VECTOR(1 downto 0);
        ForwardBE:         out STD_LOGIC_VECTOR(1 downto 0);
        StallF, StallD:    out STD_LOGIC;
        FlushD, FlushE:    out STD_LOGIC);
  end component;
  signal RegSrcD, ImmSrcD, ALUControlE: STD_LOGIC_VECTOR(1 downto 0);
  signal ALUSrcE, BranchTakenE, MemtoRegW, PCSrcW, RegWriteW: STD_LOGIC;
  signal ALUFlagsE: STD_LOGIC_VECTOR(3 downto 0);
  signal InstrD: STD_LOGIC_VECTOR(31 downto 0);
  signal RegWriteM, MemtoRegE, PCWrPendingF: STD_LOGIC;
  signal ForwardAE, ForwardBE: STD_LOGIC_VECTOR(1 downto 0);
  signal StallF, StallD, FlushD, FlushE: STD_LOGIC;
  signal Match_1E_M, Match_1E_W, Match_2E_M, Match_2E_W, Match_12D_E:
STD_LOGIC;

begin
  c: controller port map(clk, reset, InstrD(31 downto 12), ALUFlagsE,
                         RegSrcD, ImmSrcD,
                         ALUSrcE, BranchTakenE, ALUControlE,
                         MemWriteM,
                         MemtoRegW, PCSrcW, RegWriteW,
                         RegWriteM, MemtoRegE, PCWrPendingF,
                         FlushE);

  dp: datapath port map(clk, reset,
                        RegSrcD, ImmSrcD,
                        ALUSrcE, BranchTakenE, ALUControlE,
                        MemtoRegW, PCSrcW, RegWriteW,
                        PCF, InstrF, InstrD,
                        ALUOutM, WriteDataM, ReadDataM,
                        ALUFlagsE,
                        Match_1E_M, Match_1E_W, Match_2E_M,
                        Match_2E_W, Match_12D_E,
```

```
                        ForwardAE, ForwardBE, StallF, StallD, FlushD);
  h: hazard port map(clk, reset, Match_1E_M, Match_1E_W,
                     Match_2E_M, Match_2E_W, Match_12D_E,
                     RegWriteM, RegWriteW, BranchTakenE, MemtoRegE,
                     PCWrPendingF, PCSrcW,
                     ForwardAE, ForwardBE,
                     StallF, StallD, FlushD, FlushE);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- pipelined control decoder
  port(clk, reset:        in  STD_LOGIC;
       InstrD:            in  STD_LOGIC_VECTOR(31 downto 12);
       ALUFlagsE:         in  STD_LOGIC_VECTOR(3 downto 0);
       RegSrcD, ImmSrcD:  out STD_LOGIC_VECTOR(1 downto 0);
       ALUSrcE:           out STD_LOGIC;
       BranchTakenE:      out STD_LOGIC;
       ALUControlE:       out STD_LOGIC_VECTOR(1 downto 0);
       MemWriteM:         out STD_LOGIC;
       MemtoRegW:         out STD_LOGIC;
       PCSrcW:            out STD_LOGIC;
       RegWriteW:         out STD_LOGIC;
       -- hazard interface
       RegWriteM:         out STD_LOGIC;
       MemtoRegE:         out STD_LOGIC;
       PCWrPendingF:      out STD_LOGIC;
       FlushE:            in  STD_LOGIC);
end;
architecture synth of controller is
  component flopr generic(width: integer);
    port(clk, reset:      in  STD_LOGIC;
         d:               in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:               out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component floprc generic(width: integer);
    port(clk, reset, clear: in  STD_LOGIC;
         d:                 in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:                 out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component conditional
    port(Cond:              in  STD_LOGIC_VECTOR(3 downto 0);
         Flags:             in  STD_LOGIC_VECTOR(3 downto 0);
         ALUFlags:          in  STD_LOGIC_VECTOR(3 downto 0);
         FlagsWrite:        in  STD_LOGIC_VECTOR(1 downto 0);
         CondEx:            out STD_LOGIC;
         FlagsNext:         out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal controlsD: STD_LOGIC_VECTOR(9 downto 0);
  signal CondExE, ALUOpD: STD_LOGIC;
  signal ALUControlD: STD_LOGIC_VECTOR(1 downto 0);
  signal ALUSrcD: STD_LOGIC;
  signal MemtoRegD, MemtoRegM: STD_LOGIC;
  signal RegWriteD, RegWriteE, RegWriteGatedE: STD_LOGIC;
  signal MemWriteD, MemWriteE, MemWriteGatedE: STD_LOGIC;
```

```
  signal BranchD, BranchE: STD_LOGIC;
  signal FlagWriteD, FlagWriteE: STD_LOGIC_VECTOR(1 downto 0);
  signal PCSrcD, PCSrcE, PCSrcM: STD_LOGIC;
  signal FlagsE, FlagsNextE, CondE: STD_LOGIC_VECTOR(3 downto 0);
  signal Funct: STD_LOGIC_VECTOR(5 downto 0);
  signal Rd: STD_LOGIC_VECTOR(3 downto 0);
  signal PCSrcGatedE: STD_LOGIC;
  signal FlushedValsEnext, FlushedValsE: STD_LOGIC_VECTOR(6 downto 0);
  signal ValsEnext, ValsE: STD_LOGIC_VECTOR(2 downto 0);
  signal ValsMnext, ValsM: STD_LOGIC_VECTOR(3 downto 0);
  signal ValsWnext, ValsW: STD_LOGIC_VECTOR(2 downto 0);
begin
  -- Decode stage
  -- Main Decoder
  process(all) begin
    case InstrD(27 downto 26) is
      when "00" =>   controlsD <= "0000101001" when InstrD(25) -- DP imm
                             else "0000001001";                -- DP reg
      when "01" =>   controlsD <= "0001111000" when InstrD(20) -- LDR
                             else "1001110100";                -- STR
      when "10" =>   controlsD <= "0110100010";                -- B
      when others => controlsD <= "----------";                --
unimplemented
    end case;
  end process;

  (RegSrcD, ImmSrcD, ALUSrcD, MemtoRegD,
   RegWriteD, MemWriteD, BranchD, ALUOpD) <= controlsD;

  -- ALU Decoder
  Funct <= InstrD(25 downto 20);
  Rd <= InstrD(15 downto 12);
  process(all) begin
    if (ALUOpD) then
      case Funct(4 downto 1) is
        when "0100" => ALUControlD <= "00"; -- ADD
        when "0010" => ALUControlD <= "01"; -- SUB
        when "0000" => ALUControlD <= "10"; -- AND
        when "1100" => ALUControlD <= "11"; -- ORR
        when others => ALUControlD <= "--"; -- unimplemented
      end case;
      FlagWriteD(1) <= Funct(0);
      FlagWriteD(0) <= Funct(0) and (not ALUControlD(1));
    else
      ALUControlD <= "00";
      FlagWriteD  <= "00";
    end if;
  end process;

  PCSrcD <= ((and Rd) and RegWriteD) or BranchD;

-- Execute stage
  FlushedValsEnext <= (FlagWriteD, BranchD, MemWriteD, RegWriteD,
              PCSrcD, MemtoRegD);
```

```
  ValsEnext <= (ALUSrcD, ALUControlD);
  flushedregsE: floprc generic map (7)
    port map(clk, reset, FlushE, FlushedValsEnext, FlushedValsE);
  regsE: flopr generic map (3)
    port map(clk, reset, ValsEnext, ValsE);
  condregE: flopr generic map (4)
    port map(clk, reset, InstrD(31 downto 28), CondE);
  flagsreg: flopr generic map (4)
    port map(clk, reset, FlagsNextE, FlagsE);

  (FlagWriteE, BranchE, MemWriteE, RegWriteE, PCSrcE, MemtoRegE) <=
FlushedValsE;
  (ALUSrcE, ALUControlE) <= ValsE;

  -- write and Branch controls are conditional
  Cond: conditional port map(CondE, FlagsE, ALUFlagsE, FlagWriteE,
CondExE, FlagsNextE);
  BranchTakenE    <= BranchE and CondExE;
  RegWriteGatedE  <= RegWriteE and CondExE;
  MemWriteGatedE  <= MemWriteE and CondExE;
  PCSrcGatedE     <= PCSrcE and CondExE;

  -- Memory stage
  ValsMnext <= (MemWriteGatedE, MemtoRegE, RegWriteGatedE, PCSrcGatedE);
  regsM: flopr generic map (4)
    port map(clk, reset, ValsMnext, ValsM);
  (MemWriteM, MemtoRegM, RegWriteM, PCSrcM) <= ValsM;

  -- Writeback stage
  ValsWnext <= (MemtoRegM, RegWriteM, PCSrcM);
  regsW: flopr generic map (3)
    port map(clk, reset, ValsWnext, ValsW);
  (MemtoRegW, RegWriteW, PCSrcW) <= ValsW;

  -- Hazard Prediction
  PCWrPendingF <= PCSrcD or PCSrcE or PCSrcM;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity conditional is
  port(Cond:           in  STD_LOGIC_VECTOR(3 downto 0);
       Flags:          in  STD_LOGIC_VECTOR(3 downto 0);
       ALUFlags:       in  STD_LOGIC_VECTOR(3 downto 0);
       FlagsWrite:     in  STD_LOGIC_VECTOR(1 downto 0);
       CondEx:         out STD_LOGIC;
       FlagsNext:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture behave of conditional is
  signal neg, zero, carry, overflow, ge: STD_LOGIC;
begin
  (neg, zero, carry, overflow) <= Flags;
  ge <= (neg xnor overflow);
```

```
  process(all) begin -- Condition checking
    case Cond is
      when "0000" => CondEx <= zero;
      when "0001" => CondEx <= not zero;
      when "0010" => CondEx <= carry;
      when "0011" => CondEx <= not carry;
      when "0100" => CondEx <= neg;
      when "0101" => CondEx <= not neg;
      when "0110" => CondEx <= overflow;
      when "0111" => CondEx <= not overflow;
      when "1000" => CondEx <= carry and (not zero);
      when "1001" => CondEx <= not(carry and (not zero));
      when "1010" => CondEx <= ge;
      when "1011" => CondEx <= not ge;
      when "1100" => CondEx <= (not zero) and ge;
      when "1101" => CondEx <= not ((not zero) and ge);
      when "1110" => CondEx <= '1';
      when others => CondEx <= '-';
    end case;
  end process;

  FlagsNext(3 downto 2) <= ALUFlags(3 downto 2) when (FlagsWrite(1) and
CondEx) else   Flags(3 downto 2);
  FlagsNext(1 downto 0) <= ALUFlags(1 downto 0) when (FlagsWrite(0) and
CondEx) else   Flags(1 downto 0);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity datapath is
    port(clk, reset:        in  STD_LOGIC;
         RegSrcD, ImmSrcD:  in  STD_LOGIC_VECTOR(1 downto 0);
         ALUSrcE:           in  STD_LOGIC;
         BranchTakenE:      in  STD_LOGIC;
         ALUControlE:       in  STD_LOGIC_VECTOR(1 downto 0);
         MemtoRegW:         in  STD_LOGIC;
         PCSrcW:            in  STD_LOGIC;
         RegWriteW:         in  STD_LOGIC;
         PCF:               out STD_LOGIC_VECTOR(31 downto 0);
         InstrF:            in  STD_LOGIC_VECTOR(31 downto 0);
         InstrD:            out STD_LOGIC_VECTOR(31 downto 0);
         ALUOutM:           out STD_LOGIC_VECTOR(31 downto 0);
         WriteDataM:        out STD_LOGIC_VECTOR(31 downto 0);
         ReadDataM:         in  STD_LOGIC_VECTOR(31 downto 0);
         ALUFlagsE:         out STD_LOGIC_VECTOR(3 downto 0);
         -- hazard logic
         Match_1E_M:        out STD_LOGIC;
         Match_1E_W:        out STD_LOGIC;
         Match_2E_M:        out STD_LOGIC;
         Match_2E_W:        out STD_LOGIC;
         Match_12D_E:       out STD_LOGIC;
         ForwardAE:         in  STD_LOGIC_VECTOR(1 downto 0);
         ForwardBE:         in  STD_LOGIC_VECTOR(1 downto 0);
         StallF:            in  STD_LOGIC;
         StallD:            in  STD_LOGIC;
```

```
        FlushD:            in  STD_LOGIC);
end;
architecture struct of datapath is
  component alu
    port(a, b:       in  STD_LOGIC_VECTOR(31 downto 0);
         ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
         Result:     buffer STD_LOGIC_VECTOR(31 downto 0);
         ALUFlags:      out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  component regfile
    port(clk:           in  STD_LOGIC;
         we3:           in  STD_LOGIC;
         ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
         wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
         rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
         y:    out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component extend
    port(Instr:  in  STD_LOGIC_VECTOR(23 downto 0);
         ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
         ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component flopr generic(width: integer);
    port(clk, reset: in  STD_LOGIC;
         d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flopenrc generic(width: integer);
    port(clk, reset, en, clear: in  STD_LOGIC;
         d:                     in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:                     out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component flopenr generic(width: integer);
    port(clk, reset, en: in  STD_LOGIC;
         d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:              out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux2 generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component mux3 generic(width: integer);
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:          in  STD_LOGIC_VECTOR(1 downto 0);
       y:          out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  component eqcmp generic(width: integer);
    port(a, b: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y:        out STD_LOGIC);
  end component;
```

```
  signal PCPlus4F, PCnext1F, PCnextF: STD_LOGIC_VECTOR(31 downto 0);
  signal ExtImmD, rd1D, rd2D, PCPlus8D: STD_LOGIC_VECTOR(31 downto 0);
  signal rd1E, rd2E, ExtImmE, SrcAE: STD_LOGIC_VECTOR(31 downto 0);
  signal SrcBE, WriteDataE, ALUResultE: STD_LOGIC_VECTOR(31 downto 0);
  signal ReadDataW, ALUOutW, ResultW: STD_LOGIC_VECTOR(31 downto 0);
  signal RA1D, RA2D, RA1E, RA2E: STD_LOGIC_VECTOR(3 downto 0);
  signal WA3E, WA3M, WA3W: STD_LOGIC_VECTOR(3 downto 0);
  signal Match_1D_E, Match_2D_E: STD_LOGIC;
  signal notStallF: STD_LOGIC;
begin
  -- Fetch stage
  notStallF <= (not StallF);
  pcnextmux: mux2 generic map (32)
    port map(PCPlus4F, ResultW, PCSrcW, PCnext1F);
  branchmux: mux2 generic map (32)
    port map(PCnext1F, ALUResultE, BranchTakenE, PCnextF);
  pcreg: flopenr generic map (32)
    port map(clk, reset, notStallF, PCnextF, PCF);
  pcadd: adder generic map (32)
    port map(PCF, 32D"4", PCPlus4F);

  -- Decode Stage
  PCPlus8D <= PCPlus4F; -- skip register
  instrreg: flopenrc generic map (32)
    port map(clk, reset, (not StallD), FlushD, InstrF, InstrD);
  ra1mux: mux2 generic map (4)
    port map(InstrD(19 downto 16), 4D"15", RegSrcD(0), RA1D);
  ra2mux: mux2 generic map (4)
    port map(InstrD(3 downto 0), InstrD(15 downto 12), RegSrcD(1), RA2D);
  rf: regfile
    port map(clk, RegWriteW, RA1D, RA2D,
             WA3W, ResultW, PCPlus8D,
             rd1D, rd2D);
  ext: extend
    port map(InstrD(23 downto 0), ImmSrcD, ExtImmD);

  -- Execute Stage
  rd1reg: flopr generic map (32)
    port map(clk, reset, rd1D, rd1E);
  rd2reg: flopr generic map (32)
    port map(clk, reset, rd2D, rd2E);
  immreg: flopr generic map (32)
    port map(clk, reset, ExtImmD, ExtImmE);
  wa3ereg: flopr generic map (4)
    port map(clk, reset, InstrD(15 downto 12), WA3E);
  ra1reg: flopr generic map (4)
    port map(clk, reset, RA1D, RA1E);
  ra2reg: flopr generic map (4)
    port map(clk, reset, RA2D, RA2E);
  byp1mux: mux3 generic map (32)
    port map(rd1E, ResultW, ALUOutM, ForwardAE, SrcAE);
  byp2mux: mux3 generic map (32)
    port map(rd2E, ResultW, ALUOutM, ForwardBE, WriteDataE);
```

```
  srcbmux: mux2 generic map (32)
    port map(WriteDataE, ExtImmE, ALUSrcE, SrcBE);
  i_alu: alu
    port map(SrcAE, SrcBE, ALUControlE, ALUResultE, ALUFlagsE);

  -- Memory Stage
  aluresreg: flopr generic map (32)
    port map(clk, reset, ALUResultE, ALUOutM);
  wdreg: flopr generic map (32)
    port map(clk, reset, WriteDataE, WriteDataM);
  wa3mreg: flopr generic map (4)
    port map(clk, reset, WA3E, WA3M);

  -- Writeback Stage
  aluoutreg: flopr generic map (32)
    port map(clk, reset, ALUOutM, ALUOutW);
  rdreg: flopr generic map (32)
    port map(clk, reset, ReadDataM, ReadDataW);
  wa3wreg: flopr generic map (4)
    port map(clk, reset, WA3M, WA3W);
  resmux: mux2 generic map (32)
    port map(ALUOutW, ReadDataW, MemtoRegW, ResultW);

  -- hazard comparison
  m0: eqcmp generic map (4)
    port map(WA3M, RA1E, Match_1E_M);
  m1: eqcmp generic map (4)
    port map(WA3W, RA1E, Match_1E_W);
  m2: eqcmp generic map (4)
    port map(WA3M, RA2E, Match_2E_M);
  m3: eqcmp generic map (4)
    port map(WA3W, RA2E, Match_2E_W);
  m4a: eqcmp generic map (4)
    port map(WA3E, RA1D, Match_1D_E);
  m4b: eqcmp generic map (4)
    port map(WA3E, RA2D, Match_2D_E);
  Match_12D_E <= Match_1D_E or Match_2D_E;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity hazard is
  port(clk, reset:        in  STD_LOGIC;
       Match_1E_M:        in  STD_LOGIC;
       Match_1E_W:        in  STD_LOGIC;
       Match_2E_M:        in  STD_LOGIC;
       Match_2E_W:        in  STD_LOGIC;
       Match_12D_E:       in  STD_LOGIC;
       RegWriteM:         in  STD_LOGIC;
       RegWriteW:         in  STD_LOGIC;
       BranchTakenE:      in  STD_LOGIC;
       MemtoRegE:         in  STD_LOGIC;
       PCWrPendingF:      in  STD_LOGIC;
       PCSrcW:            in  STD_LOGIC;
       ForwardAE:         out STD_LOGIC_VECTOR(1 downto 0);
```

```
       ForwardBE:         out STD_LOGIC_VECTOR(1 downto 0);
       StallF, StallD:    out STD_LOGIC;
       FlushD, FlushE:    out STD_LOGIC);
end;

architecture behave of hazard is
  signal ldrStallD: STD_LOGIC;
begin
  ForwardAE(1) <= '1' when (Match_1E_M and RegWriteM) else '0';
  ForwardAE(0) <= '1' when (Match_1E_W and RegWriteW and (not
ForwardAE(1))) else '0';

  ForwardBE(1) <= '1' when (Match_2E_M and RegWriteM) else '0';
  ForwardBE(0) <= '1' when (Match_2E_W and RegWriteW and (not
ForwardBE(1))) else '0';

  ldrStallD <= Match_12D_E and MemtoRegE;

  StallD <= ldrStallD;
  StallF <= ldrStallD or PCWrPendingF;
  FlushE <= ldrStallD or BranchTakenE;
  FlushD <= PCWrPendingF or PCSrcW or BranchTakenE;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity regfile is -- three-port register file
  port(clk:           in  STD_LOGIC;
       we3:           in  STD_LOGIC;
       ra1, ra2, wa3: in  STD_LOGIC_VECTOR(3 downto 0);
       wd3, r15:      in  STD_LOGIC_VECTOR(31 downto 0);
       rd1, rd2:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of regfile is
  type ramtype is array (31 downto 0) of
    STD_LOGIC_VECTOR(31 downto 0);
  signal mem: ramtype;
begin
  process(clk) begin
    if falling_edge(clk) then  -- write rf on negative edge of clock
       if we3 = '1' then mem(to_integer(wa3)) <= wd3;
       end if;
    end if;
  end process;
  process(all) begin
    if (to_integer(ra1) = 15) then rd1 <= r15;
    else rd1 <= mem(to_integer(ra1));
    end if;
    if (to_integer(ra2) = 15) then rd2 <= r15;
    else rd2 <= mem(to_integer(ra2));
    end if;
  end process;
end;
```

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity adder is -- adder
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of adder is
begin
  y <= a + b;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity extend is
  port(Instr:  in  STD_LOGIC_VECTOR(23 downto 0);
       ImmSrc: in  STD_LOGIC_VECTOR(1 downto 0);
       ExtImm: out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of extend is
begin
  process(all) begin
    case ImmSrc is
      when "00"   => ExtImm <= (X"000000", Instr(7 downto 0));
      when "01"   => ExtImm <= (X"00000", Instr(11 downto 0));
      when "10"   => ExtImm <= (Instr(23), Instr(23), Instr(23),
        Instr(23), Instr(23), Instr(23), Instr(23 downto 0), "00");
      when others => ExtImm <= X"--------";
    end case;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopenr is -- flip-flop with enable and asynchronous reset
  generic(width: integer);
  port(clk, reset, en: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenr is
begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
        q <= d;
      end if;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
```

```
entity flopr is -- flip-flop with asynchronous reset
  generic(width: integer);
  port(clk, reset: in  STD_LOGIC;
       d:          in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset then  q <= (others => '0');
    elsif rising_edge(clk) then
      q <= d;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity floprc is -- flip-flop with asynchronous reset
                 -- and synchronous clear
  generic(width: integer);
  port(clk, reset, clear: in  STD_LOGIC;
       d:                 in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:                 out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of floprc is
begin
  process(clk, reset) begin
    if reset then   q <= (others => '0');
    elsif rising_edge(clk) then
      if clear then q <= (others => '0');
      else          q <= d;
      end if;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopenrc is -- flip-flop with enable and asynchronous reset,
synchronous clear
  generic(width: integer);
  port(clk, reset, en, clear: in  STD_LOGIC;
       d:                     in  STD_LOGIC_VECTOR(width-1 downto 0);
       q:                     out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture asynchronous of flopenrc is
begin
  process(clk, reset) begin
    if reset then q <= (others => '0');
    elsif rising_edge(clk) then
      if en then
        if clear then
```

```
          q <= (others => '0');
        else
          q <= d;
        end if;
      end if;
    end if;
  end process;
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
  generic(width: integer);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d1 when s else d0;
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux3 is -- three-input multiplexer
  generic(width: integer);
  port(d0, d1, d2: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:          in  STD_LOGIC_VECTOR(1 downto 0);
       y:          out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux3 is
begin
  process(all) begin
    case s is
      when "00"    => y <= d0;
      when "01"    => y <= d1;
      when "10"    => y <= d2;
      when others  => y <= d0;
    end case;
  end process;
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;
entity eqcmp is -- equality comparator
  generic(width: integer);
  port(a, b: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y:        out STD_LOGIC);
end;

architecture behave of eqcmp is
begin
  y <= '1'when a = b else '0';
```

```
end;


library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD_UNSIGNED.all;
entity alu is
  port(a, b:       in  STD_LOGIC_VECTOR(31 downto 0);
       ALUControl: in  STD_LOGIC_VECTOR(1 downto 0);
       Result:     buffer STD_LOGIC_VECTOR(31 downto 0);
       ALUFlags:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture behave of alu is
  signal condinvb: STD_LOGIC_VECTOR(31 downto 0);
  signal sum:      STD_LOGIC_VECTOR(32 downto 0);
  signal neg, zero, carry, overflow: STD_LOGIC;
begin
  condinvb <= not b when ALUControl(0) else b;
  sum <= ('0', a) + ('0', condinvb) + ALUControl(0);

  process(all) begin
    case? ALUControl(1 downto 0) is
      when "0-"   => result <= sum(31 downto 0);
      when "10"   => result <= a and b;
      when "11"   => result <= a or b;
      when others => result <= (others => '-');
    end case?;
  end process;

  neg      <= Result(31);
  zero     <= '1' when (Result = 0) else '0';
  carry    <= (not ALUControl(1)) and sum(32);
  overflow <= (not ALUControl(1)) and
              (not (a(31) xor b(31) xor ALUControl(0))) and
              (a(31) xor sum(31));
  ALUFlags    <= (neg, zero, carry, overflow);
end;
```

Exercise 7.41

SystemVerilog

```
module hazard(input  logic       clk, reset,
              input  logic       Match_1E_M, Match_1E_W, Match_2E_M,
              input  logic       Match_2E_W, Match_12D_E,
              input  logic       RegWriteM, RegWriteW,
              input  logic       BranchTakenE, MemtoRegE,
              input  logic       PCWrPendingF, PCSrcW,
              output logic [1:0] ForwardAE, ForwardBE,
              output logic       StallF, StallD,
              output logic       FlushD, FlushE);

  logic ldrStallD;
```

```
  // forwarding logic
  always_comb begin
    if (Match_1E_M & RegWriteM)      ForwardAE = 2'b10;
    else if (Match_1E_W & RegWriteW) ForwardAE = 2'b01;
    else                             ForwardAE = 2'b00;

    if (Match_2E_M & RegWriteM)      ForwardBE = 2'b10;
    else if (Match_2E_W & RegWriteW) ForwardBE = 2'b01;
    else                             ForwardBE = 2'b00;
  end

  // stalls and flushes
  // Load RAW
  //   when an instruction reads a register loaded by the previous,
  //   stall in the decode stage until it is ready
  // Branch hazard
  //   When a branch is taken, flush the incorrectly fetched instrs
  //   from decode and execute stages
  // PC Write Hazard
  //   When the PC might be written, stall all following instructions
  //   by stalling the fetch and flushing the decode stage
  // when a stage stalls, stall all previous and flush next

  assign ldrStallD = Match_12D_E & MemtoRegE;

  assign StallD = ldrStallD;
  assign StallF = ldrStallD | PCWrPendingF;
  assign FlushE = ldrStallD | BranchTakenE;
  assign FlushD = PCWrPendingF | PCSrcW | BranchTakenE;

endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity hazard is
  port(clk, reset:        in  STD_LOGIC;
       Match_1E_M:        in  STD_LOGIC;
       Match_1E_W:        in  STD_LOGIC;
       Match_2E_M:        in  STD_LOGIC;
       Match_2E_W:        in  STD_LOGIC;
       Match_12D_E:       in  STD_LOGIC;
       RegWriteM:         in  STD_LOGIC;
       RegWriteW:         in  STD_LOGIC;
       BranchTakenE:      in  STD_LOGIC;
       MemtoRegE:         in  STD_LOGIC;
       PCWrPendingF:      in  STD_LOGIC;
       PCSrcW:            in  STD_LOGIC;
       ForwardAE:         out STD_LOGIC_VECTOR(1 downto 0);
       ForwardBE:         out STD_LOGIC_VECTOR(1 downto 0);
       StallF, StallD:    out STD_LOGIC;
       FlushD, FlushE:    out STD_LOGIC);
end;
```

```
architecture behave of hazard is
  signal ldrStallD: STD_LOGIC;
begin
  ForwardAE(1) <= '1' when (Match_1E_M and RegWriteM) else '0';
  ForwardAE(0) <= '1' when (Match_1E_W and RegWriteW and (not
ForwardAE(1))) else '0';

  ForwardBE(1) <= '1' when (Match_2E_M and RegWriteM) else '0';
  ForwardBE(0) <= '1' when (Match_2E_W and RegWriteW and (not
ForwardBE(1))) else '0';

  ldrStallD <= Match_12D_E and MemtoRegE;

  StallD <= ldrStallD;
  StallF <= ldrStallD or PCWrPendingF;
  FlushE <= ldrStallD or BranchTakenE;
  FlushD <= PCWrPendingF or PCSrcW or BranchTakenE;
end;
```

Hazard Unit Schematic