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
```