		(post-indexing, register offset)									

Exercise 7.7

She should work on the memory. t_{mem} = (200/2) ps = 100 ps
From Equation 7.3, the new cycle time is:

T_{c1} = 40 + 2(100) + 70 + 100 + 120 + 2(25) + 60 = **640 ps**

Exercise 7.9

SystemVerilog

```
// ex7.9 solutions
//
// single-cycle ARM processor
// additional instructions: TST, LSL, CMN, ADC

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 === 20 & WriteData === 2) begin
          $display("Simulation succeeded");
          $stop;
        end else begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule

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
  arm arm(clk, reset, PC, Instr, MemWrite, DataAdr,
          WriteData, ReadData);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

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

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

  assign rd = RAM[a[31:2]]; // word aligned

  always_ff @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule

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

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

  initial
      $readmemh("ex7.9_memfile.dat",RAM);

  assign rd = RAM[a[31:2]]; // word aligned
endmodule

module arm(input  logic        clk, reset,
           output logic [31:0] PC,
           input  logic [31:0] Instr,
           output logic        MemWrite,
           output logic [31:0] ALUResult, WriteData,
           input  logic [31:0] ReadData);

  logic [3:0] ALUFlags;
  logic       RegWrite,
              ALUSrc, MemtoReg, PCSrc;
  logic [1:0] RegSrc, ImmSrc;
  logic [2:0] ALUControl;  // ADC
  logic       carry;  // ADC
  logic       Shift;  // LSL

  controller c(clk, reset, Instr[31:12], ALUFlags,
               RegSrc, RegWrite, ImmSrc,
               ALUSrc, ALUControl,
               MemWrite, MemtoReg, PCSrc,
               carry,  // ADC
               Shift); // LSL
  datapath dp(clk, reset,
              RegSrc, RegWrite, ImmSrc,
              ALUSrc, ALUControl,
              MemtoReg, PCSrc,
              ALUFlags, PC, Instr,
              ALUResult, WriteData, ReadData,
              carry,  // ADC
              Shift); // LSL
endmodule

module controller(input  logic         clk, reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic [1:0]   RegSrc,
                  output logic         RegWrite,
                  output logic [1:0]   ImmSrc,
                  output logic         ALUSrc,
                  output logic [2:0]   ALUControl, // ADC
                  output logic         MemWrite, MemtoReg,
                  output logic         PCSrc,
                  output logic         carry,      // ADC
                  output logic         Shift);     // LSL

  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;
  logic       NoWrite;  // TST, CMN

  decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
              FlagW, PCS, RegW, MemW,
              MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl,
              NoWrite,  // TST, CMN
              Shift);   // LSL
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite,
               carry,      // ADC
               NoWrite);   // TST, CMN
endmodule

module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
             output logic       PCS, RegW, MemW,
             output logic       MemtoReg, ALUSrc,
             output logic [1:0] ImmSrc, RegSrc,
             output logic [2:0] ALUControl, // ADC
             output logic       NoWrite,    // TST, CMN
             output logic       Shift);     // LSL
logic [9:0] controls;
logic       Branch, ALUOp;

// Main Decoder

always_comb
   case(Op)
                           // Data processing immediate
     2'b00: if (Funct[5])  controls = 10'b0000101001;
                           // Data processing register
            else           controls = 10'b0000001001;
                           // LDR
     2'b01: if (Funct[0])  controls = 10'b0001111000;
                           // STR
            else           controls = 10'b1001110100;
                           // B
     2'b10:                controls = 10'b0110100010;
                           // Unimplemented
     default:              controls = 10'bx;
   endcase

assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
        RegW, MemW, Branch, ALUOp} = controls;

// ALU Decoder
always_comb
  if (ALUOp) begin                     // which DP Instr?
    case(Funct[4:1])
      4'b0100: begin                   // ADD
                 ALUControl = 3'b000;
                 NoWrite = 1'b0;
                 Shift = 1'b0;
               end
      4'b0010: begin                   // SUB
                 ALUControl = 3'b001;
                 NoWrite = 1'b0;
                 Shift = 1'b0;
               end
      4'b0000: begin                   // AND
                 ALUControl = 3'b010;
                 NoWrite = 1'b0;
                 Shift = 1'b0;
               end
      4'b1100: begin                   // OR
                 ALUControl = 3'b011;
                 NoWrite = 1'b0;
                 Shift = 1'b0;
               end
        4'b1000: begin                      // TST
                   ALUControl = 3'b010;
                   NoWrite = 1'b1;
                   Shift = 1'b0;
                 end
        4'b1101: begin                      // LSL
                   ALUControl = 3'b000;
                   NoWrite = 1'b0;
                   Shift = 1'b1;
                 end
        4'b1011: begin                      // CMN
                   ALUControl = 3'b000;
                   NoWrite = 1'b1;
                   Shift = 1'b0;
                 end
        4'b0101: begin                      // ADC
                   ALUControl = 3'b100;
                   NoWrite = 1'b0;
                   Shift = 1'b0;
                 end
        default: begin                      // unimplemented
                   ALUControl = 3'bx;
                   NoWrite = 1'bx;
                   Shift = 1'bx;
                 end
      endcase

      // update flags if S bit is set
      // (C & V only updated for arith instructions)
      FlagW[1]      = Funct[0]; // FlagW[1] = S-bit
      // FlagW[0] = S-bit & (ADD | SUB)
      FlagW[0]      = Funct[0] &
        (ALUControl[1:0] == 2'b00 | ALUControl[1:0] == 2'b01);


    end else begin
      ALUControl = 3'b000; // add for non-DP instructions
      FlagW      = 2'b00;  // don't update Flags
      NoWrite    = 1'b0;
      Shift      = 1'b0;
    end


  // PC Logic
  assign PCS  = ((Rd == 4'b1111) & RegW) | Branch;
endmodule

module condlogic(input  logic       clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic       PCS, RegW, MemW,
                 output logic       PCSrc, RegWrite, MemWrite,
                 output logic       carry,    // ADC
                 input  logic       NoWrite); // TST, CMN

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  // write controls are conditional
  condcheck cc(Cond, Flags, CondEx);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondEx & ~NoWrite; // TST, CMN
  assign MemWrite  = MemW  & CondEx;
  assign PCSrc     = PCS   & CondEx;

  assign carry     = Flags[1];  // ADC
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  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
endmodule

module datapath(input  logic        clk, reset,
                input  logic [1:0]  RegSrc,
                input  logic        RegWrite,
                 input  logic [1:0]  ImmSrc,
                 input  logic        ALUSrc,
                 input  logic [2:0]  ALUControl,      // ADC
                 input  logic        MemtoReg,
                 input  logic        PCSrc,
                 output logic [3:0]  ALUFlags,
                 output logic [31:0] PC,
                 input  logic [31:0] Instr,
                 output logic [31:0] ALUResultOut,    // LSL
                 output logic [31:0] WriteData,
                 input  logic [31:0] ReadData,
                 input  logic        carry,           // ADC
                 input  logic        Shift);          // LSL

  logic [31:0] PCNext, PCPlus4, PCPlus8;
  logic [31:0] ExtImm, SrcA, SrcB, Result;
  logic [3:0]  RA1, RA2;
  logic [31:0] srcBshifted, ALUResult;  // LSL

  // next PC logic
  mux2 #(32)  pcmux(PCPlus4, Result, PCSrc, PCNext);
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder #(32) pcadd1(PC, 32'b100, PCPlus4);
  adder #(32) pcadd2(PCPlus4, 32'b100, PCPlus8);

  // register file logic
  mux2 #(4)   ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
  mux2 #(4)   ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
  regfile     rf(clk, RegWrite, RA1, RA2,
                 Instr[15:12], Result, PCPlus8,
                 SrcA, WriteData);
  mux2 #(32)  resmux(ALUResultOut, ReadData, MemtoReg, Result);
  extend      ext(Instr[23:0], ImmSrc, ExtImm);

  // ALU logic
  shifter     sh(WriteData, Instr[11:7], Instr[6:5], srcBshifted); // LSL
  mux2 #(32)  srcbmux(srcBshifted, ExtImm, ALUSrc, SrcB);   // LSL
  alu         alu(SrcA, SrcB, ALUControl,
                  ALUResult, ALUFlags,
                  carry);  // ADC
  mux2 #(32)  aluresultmux(ALUResult, SrcB, Shift, ALUResultOut);   // LSL

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 rising edge of clock
  // register 15 reads PC+8 instead

  always_ff @(posedge 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)
               // 8-bit unsigned immediate
      2'b00:   ExtImm = {24'b0, Instr[7:0]};
               // 12-bit unsigned immediate
      2'b01:   ExtImm = {20'b0, Instr[11:0]};
               // 24-bit two's complement shifted branch
      2'b10:   ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
      default: ExtImm = 32'bx; // undefined
    endcase
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 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 alu(input  logic [31:0] a, b,
           input  logic [2:0]  ALUControl,                // ADC
           output logic [31:0] Result,
           output logic [3:0]  ALUFlags,
           input  logic        carry);                    // ADC


  logic        neg, zero, carryout, overflow;
  logic [31:0] condinvb;
  logic [32:0] sum;
  logic        carryin;                                   // ADC

  assign carryin = ALUControl[2] ? carry : ALUControl[0]; // ADC
  assign condinvb = ALUControl[0] ? ~b : b;
  assign sum = a + condinvb + carryin;                    // ADC

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

// shifter needed for LSL
module shifter(input  logic [31:0] a,
               input  logic [ 4:0] shamt,
               input  logic [ 1:0] shtype,
               output logic [31:0] y);

  always_comb
    case (shtype)
      2'b00:   y = a << shamt;
      default: y = a;
    endcase
endmodule
```

VHDL