Jyresa Mae M. Amboang
Frances Nicole S. Gigataras
Crishelle A. Agopitac
Erica Dianne Q. Canillo
Laboratory Submitted: June 13, 2022
Laboratory Hours: 504 hours

Final Project Part I
Multicycle ARM Processor (Part I)

Main FSM output table

State (Name)	NextPC	Branch	MemW	RegW	IRWrite	AdrSrc	$ResultSrc_{1:0}$	$ALUSrcA_{1:0}$	$ALUSrcB_{1:0}$	ALUOp	FSM Control Word
0 (Fetch)	1	0	0	0	1	0	10	0	1	10	0	0x114C
1 (Decode)	0	0	0	0	0	0	10	0	1	10	0	0x004C
2 (MemAdr)	0	0	0	0	0	0	00	0	0	01	0	0x0002
3 (MemRead)	0	0	0	0	0	1	00	0	0	00	0	0x0080
4 (MemWB)	0	0	0	1	0	0	01	0	0	00	0	0x0220
5 (MemWrite)	0	0	1	0	0	1	00	0	0	00	0	0x0480
6 (ExecuteR)	0	0	0	0	0	0	00	0	0	00	1	0x0001
7 (ExecuteI)	0	0	0	0	0	0	00	0	0	01	1	0x0003
8 (ALUWB)	0	0	0	1	0	0	00	0	0	00	0	0x0200
9 (Branch)	0	1	0	0	0	0	10	1	0	01	0	0x0852

Table 1. **Main FSM output**

SystemVerilog Code for Datapath

```
module datapath(input  logic        clk, reset,
                output logic [31:0] Adr, WriteData,
                input  logic [31:0] ReadData,
                output logic [31:0] Instr,
                output logic [3:0]  ALUFlags,
                input  logic        PCWrite, RegWrite,
                input  logic        IRWrite,
                input  logic        AdrSrc,
                input  logic [1:0]  RegSrc,
                input  logic [1:0]  ALUSrcA, ALUSrcB, ResultSrc,
                input  logic [1:0]  ImmSrc, ALUControl);

  logic [31:0] PCNext, PC;
  logic [31:0] ExtImm, SrcA, SrcB, Result;
  logic [31:0] Data, RD1, RD2, A, ALUResult, ALUOut;
  logic [3:0]  RA1, RA2;

  // next PC logic
  flopenr #(32) pcreg(clk, reset, PCWrite, Result, PC);

  // memory logic
  mux2 #(32)    adrmux(PC, ALUOut, AdrSrc, Adr);
  flopenr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
  flopr   #(32) datareg(clk, reset, ReadData, Data);

  // 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, Result,
                 RD1, RD2);
  flopr #(32) srcareg(clk, reset, RD1, A);
  flopr #(32) wdreg(clk, reset, RD2, WriteData);
  extend      ext(Instr[23:0], ImmSrc, ExtImm);

  // ALU logic
  mux3 #(32)  srcamux(A, PC, ALUOut, ALUSrcA, SrcA);
  mux3 #(32)  srcbmux(WriteData, ExtImm, 32'd4, ALUSrcB, SrcB);
  alu         alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
  flopr #(32) aluoutreg(clk, reset, ALUResult, ALUOut);
  mux3 #(32)  resmux(ALUOut, Data, ALUResult, ResultSrc, Result);
endmodule
```

ModelSim Simulation (clk, reset, PC, Instr, state, and ALUResult)
SystemVerilog of the Controller unit

```
module controller(input  logic         clk,
                  input  logic         reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic         PCWrite,
                  output logic         MemWrite,
                  output logic         RegWrite,
                  output logic         IRWrite,
                  output logic         AdrSrc,
                  output logic [1:0]   RegSrc,
                  output logic [1:0]   ALUSrcA,
                  output logic [1:0]   ALUSrcB,
                  output logic [1:0]   ResultSrc,
                  output logic [1:0]   ImmSrc,
                  output logic [1:0]   ALUControl);

  logic [1:0] FlagW;
  logic       PCS, NextPC, RegW, MemW;

  decode dec(clk, reset, Instr[27:26], Instr[25:20], Instr[15:12],
             FlagW, PCS, NextPC, RegW, MemW,
             IRWrite, AdrSrc, ResultSrc,
             ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, NextPC, RegW, MemW,
               PCWrite, RegWrite, MemWrite);
endmodule

module decode(input  logic       clk, reset,
              input  logic [1:0] Op,
              input  logic [5:0] Funct,
              input  logic [3:0] Rd,
              output logic [1:0] FlagW,
              output logic       PCS, NextPC, RegW, MemW,
```

```
        output logic       IRWrite, AdrSrc,
        output logic [1:0] ResultSrc, ALUSrcA, ALUSrcB,
        output logic [1:0] ImmSrc, RegSrc, ALUControl);

logic       Branch, ALUOp;

// Main FSM
mainfsm fsm(clk, reset, Op, Funct,
        IRWrite, AdrSrc,
        ALUSrcA, ALUSrcB, ResultSrc,
        NextPC, RegW, MemW, Branch, ALUOp);

// ADD CODE BELOW
// Add code for the ALU Decoder and PC Logic.
// Remember, you may reuse code from previous labs.
// ALU Decoder
always_comb
  if (ALUOp) begin                // which Data-processing Instr?
    case(Funct[4:1])
                    4'b0100: ALUControl = 2'b00; // ADD
                    4'b0010: ALUControl = 2'b01; // SUB
          4'b0000: ALUControl = 2'b10; // AND
                    4'b1100: ALUControl = 2'b11; // ORR
                    default: ALUControl = 2'bx;  // unimplemented
    endcase
    FlagW[1]      = Funct[0]; // update N & Z flags if S bit is set
    FlagW[0]      = Funct[0] & (ALUControl == 2'b00 | ALUControl == 2'b01);
  end else begin
    ALUControl = 2'b00; // add for non data-processing instructions
    FlagW      = 2'b00; // don't update Flags
  end

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

// Add code for the Instruction Decoder (Instr Decoder) below.
// Recall that the input to Instr Decoder is Op, and the outputs are
// ImmSrc and RegSrc. We've completed the ImmSrc logic for you.

// Instr Decoder
assign ImmSrc    = Op;
assign RegSrc[0] = (Op == 2'b10); // read PC on Branch
  assign RegSrc[1] = (Op == 2'b01); // read Rd on STR

endmodule

module mainfsm(input  logic         clk,
               input  logic         reset,
               input  logic [1:0]   Op,
               input  logic [5:0]   Funct,
               output logic         IRWrite,
               output logic         AdrSrc,
               output logic [1:0]   ALUSrcA, ALUSrcB, ResultSrc,
               output logic         NextPC, RegW, MemW, Branch, ALUOp);

  typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMRD, MEMWB,
                            MEMWR, EXECUTER, EXECUTEI, ALUWB, BRANCH,
                                                       UNKNOWN}
statetype;

  statetype state, nextstate;
  logic [12:0] controls;

  // state register
  always @(posedge clk or posedge reset)
    if (reset) state <= FETCH;
    else state <= nextstate;

  // ADD CODE BELOW
  // Finish entering the next state logic below.  We've completed the
  // first two states, FETCH and DECODE, for you.

  // next state logic
  always_comb
    casex(state)
      FETCH:                    nextstate = DECODE;
      DECODE: case(Op)
                2'b00:
                  if (Funct[5])  nextstate = EXECUTEI;
                  else           nextstate = EXECUTER;
                2'b01:           nextstate = MEMADR;
                2'b10:           nextstate = BRANCH;
                default:         nextstate = UNKNOWN;
              endcase
```

```
    EXECUTER:                nextstate = ALUWB;
    EXECUTEI:                nextstate = ALUWB;
    MEMADR:
      if (Funct[0])          nextstate = MEMRD;
      else                   nextstate = MEMWR;
    MEMRD:                   nextstate = MEMWB;
    default:                 nextstate = FETCH;
  endcase

 // ADD CODE BELOW
 // Finish entering the output logic below.  We've entered the
 // output logic for the first two states, FETCH and DECODE, for you.

 // state-dependent output logic
 always_comb
  case(state)
    FETCH:      controls = 13'b10001_010_01100;
    DECODE:     controls = 13'b00000_010_01100;
    EXECUTER: controls = 13'b00000_000_00001;
    EXECUTEI:  controls = 13'b00000_000_00011;
    ALUWB:      controls = 13'b00010_000_00000;
    MEMADR:   controls = 13'b00000_000_00010;
    MEMWR:     controls = 13'b00100_100_00000;
    MEMRD:     controls = 13'b00000_100_00000;
    MEMWB:     controls = 13'b00010_001_00000;
    BRANCH:    controls = 13'b01000_010_10010;
    default:          controls = 13'bxxxxx_xxx_xxxxx;
  endcase

 assign {NextPC, Branch, MemW, RegW, IRWrite,
        AdrSrc, ResultSrc,
        ALUSrcA, ALUSrcB, ALUOp} = controls;
endmodule

// ADD CODE BELOW
// Add code for the condlogic and condcheck modules. Remember, you may
// reuse code from prior labs.
module condlogic(input  logic       clk, reset,
                 input  logic [3:0] Cond,
                 input  logic [3:0] ALUFlags,
                 input  logic [1:0] FlagW,
                 input  logic       PCS, NextPC, RegW, MemW,
                 output logic       PCWrite, RegWrite, MemWrite);

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

  // ADD CODE HERE
  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);
  flopr #(1)condreg(clk, reset, CondEx, CondExDelayed);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondExDelayed;
  assign MemWrite  = MemW  & CondExDelayed;
  assign PCWrite   = (PCS  & CondExDelayed) | NextPC;
endmodule

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

  // ADD CODE HERE
  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
```

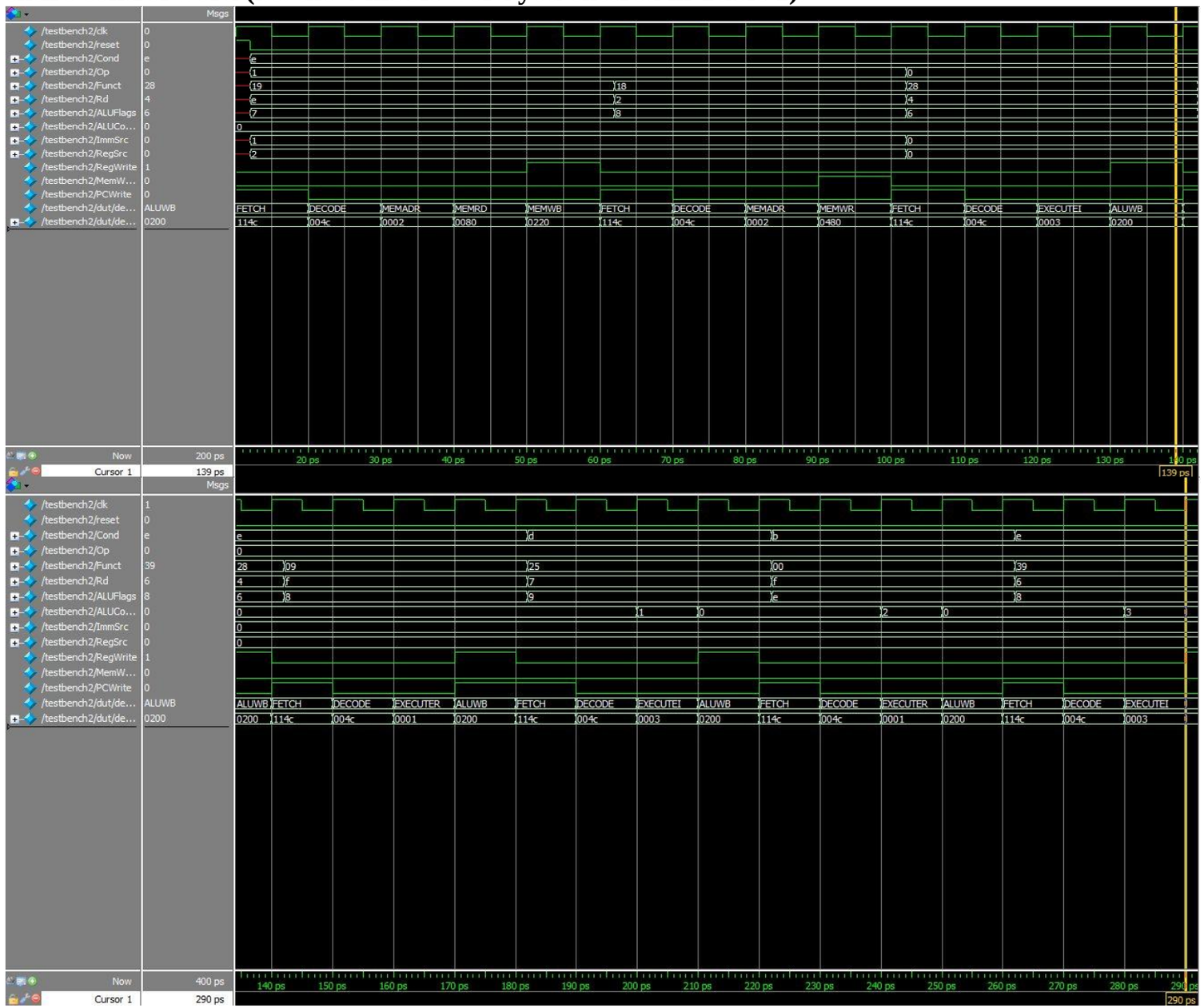
```
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // Always
      default: CondEx = 1'bx;             // undefined
    endcase
endmodule
```

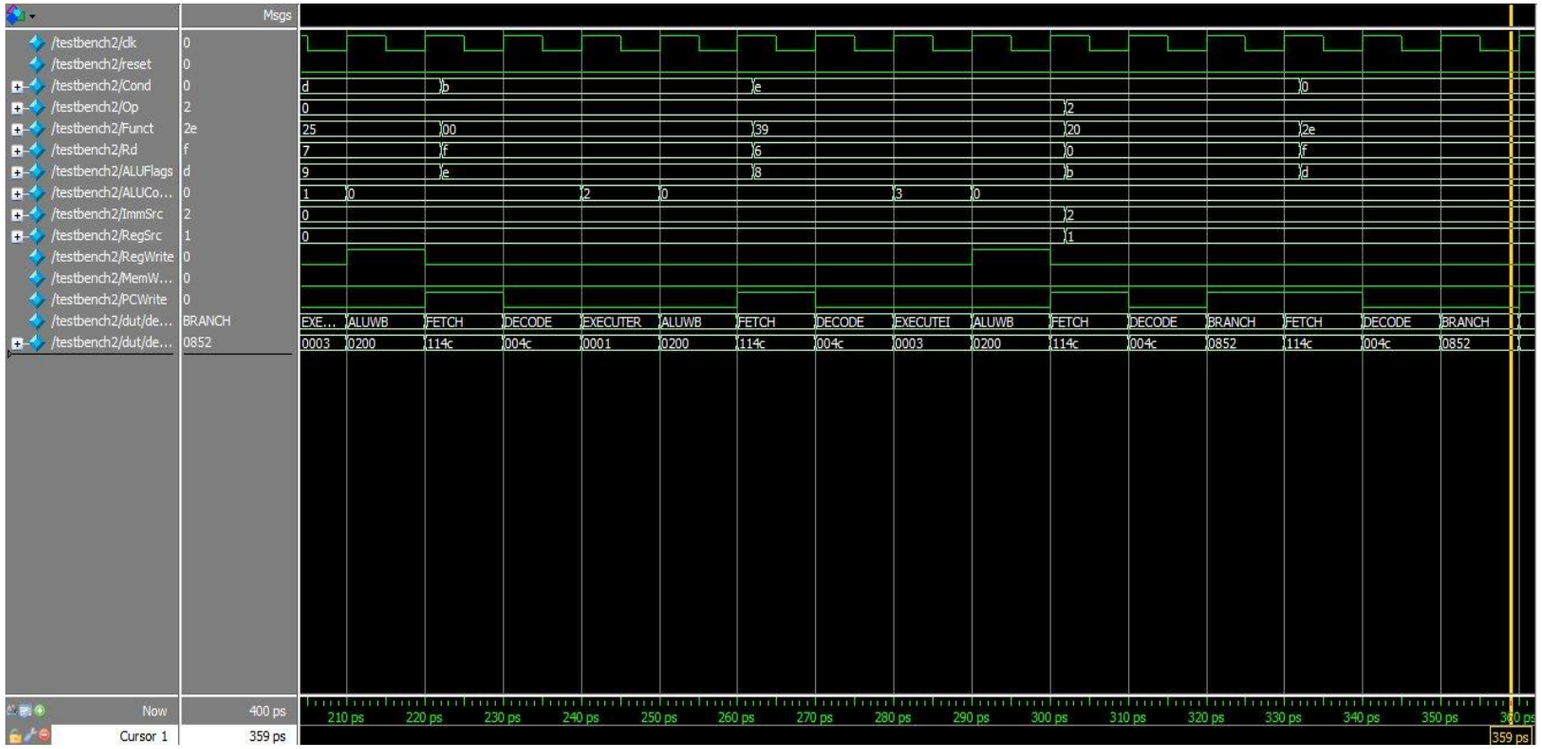
Testbench

```
module testbench2();

  // controller inputs
  logic        clk;
  logic        reset;
  logic [19:0] Instr;
  logic [3:0]  ALUFlags;

  // controller outputs
  logic        PCWrite, MemWrite, RegWrite, IRWrite, AdrSrc;
  logic [1:0]  RegSrc, ALUSrcA, ALUSrcB, ResultSrc, ImmSrc, ALUControl;

  // define Cmd fields for data-processing instructions
  parameter   ADD      = 4'b0100;
  parameter   SUB      = 4'b0010;
  parameter   AND                 = 4'b0000;
  parameter   ORR                 = 4'b1100;

  // Sub-fields of Instr[31:12]
  logic [3:0]  Cond;
  logic [1:0]  Op;
  logic [5:0]  Funct;
  logic [3:0]  Rd;

  assign Instr = {Cond, Op, Funct, 4'b0, Rd};

  // instantiate dut
  controller dut(clk, reset, Instr, ALUFlags,
                 PCWrite, MemWrite, RegWrite, IRWrite, AdrSrc,
                 RegSrc, ALUSrcA, ALUSrcB, ResultSrc, ImmSrc,
                 ALUControl);

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

// apply inputs; user must manually verify outputs
initial
  begin
    // reset controller
    reset = 1; #12; reset = 0;

    // test LDR
        assign Cond = 4'b1110; assign Op = 2'b01;
        assign Funct = 6'b011001; assign Rd = 8'b1110;
        assign ALUFlags = 4'b0111;
        #50;       // run 5 cycles

        // test STR
        assign Cond = 4'b1110; assign Op = 2'b01;
        assign Funct = 6'b011000; assign Rd = 8'b0010;
        assign ALUFlags = 4'b1000;
        #40;       // run 4 cycles

        // test ADD – immediate Src2
        assign Cond = 4'b1110; assign Op = 2'b00;
        assign Funct = {1'b1, ADD, 1'b0}; assign Rd = 8'b0100;
        assign ALUFlags = 4'b0110;
        # 40;      // run 4 cycles

        // test ADD – register Src2, set condition flags, Rd = PC
        assign Cond = 4'b1110; assign Op = 2'b00;
        assign Funct = {1'b0, ADD, 1'b1}; assign Rd = 8'b1111;
        assign ALUFlags = 4'b1000;
        # 40;      // run 4 cycles

        // test SUB – immediate Src2, conditional execution (LE),
        // set condition flags
```

```
        // should be executed
        assign Cond = 4'b1101; assign Op = 2'b00;
        assign Funct = {1'b1, SUB, 1'b1}; assign Rd = 8'b0111;
        assign ALUFlags = 4'b1001;
        # 40;      // run 4 cycles

        // test AND – register Src2, Rd = PC, cond. execution (LT)
        // shouldn't be executed
        assign Cond = 4'b1011; assign Op = 2'b00;
        assign Funct = {1'b0, AND, 1'b0}; assign Rd = 8'b1111;
        assign ALUFlags = 4'b1110;
        # 40;      // run 4 cycles

        // test ORR – immediate Src2, set Flags[3:2] (i.e., N,Z)
        assign Cond = 4'b1110; assign Op = 2'b00;
        assign Funct = {1'b1, ORR, 1'b1}; assign Rd = 8'b0110;
        assign ALUFlags = 4'b1000;
        # 40;      // run 4 cycles

        // test B (taken)
        assign Cond = 4'b1110; assign Op = 2'b10;
        assign Funct = 6'b100000; assign Rd = 8'b0;
        assign ALUFlags = 4'b1011;
        # 30;      // run 3 cycles

        // test B (not taken) EQ condition mnemonic
        assign Cond = 4'b0000; assign Op = 2'b10;
        assign Funct = 6'b101110; assign Rd = 8'b1111;
        assign ALUFlags = 4'b1101;
        # 30;      // run 3 cycles
    end
endmodule
```

ModelSim Waveforms: CLK, Reset, Cond, OP, Funct, Rd, ALUFlags, ALUControl, ImmSrc, RegSrc, RegWrite,MemWrite, PCWrite, state, and the entire control word (i.e. the 4-nibble word you entered in Table 1).





CONCLUSION:

What did we learn or take away from performing this Final Project Part I?

This lab exercise will teach us essential information about digital hardware design. To explore the implication of processor design and function, we will construct a simplified ARM single-cycle processor in SystemVerilog, and the ALU and register file, among other submodules of the control and datapath units, will become more apparent to us. SystemVerilog coding improves our expertise with hardware description languages, letting us describe complicated hardware designs in a condensed form. In this lab, we will enhance encoding strategies to strengthen the implementation of the CPU, with a focus on increasing hardware design efficiency. We'll also build testbeds and write test scripts to ensure our designs are sound. By utilizing ModelSim to simulate the CPU, our understanding of digital circuit analysis and interpretation has been dramatically improved. In addition to learning about ARM architecture and processor design, we can improve our problem-solving, debugging, and communication skills in this laboratory activity. These abilities will be essential for any digital hardware engineering initiatives we may do in the future.