E85: Digital Design and Computer Engineering
Problem Set 9

Hint: review Section 7.3 of the textbook about the single-cycle processor until you are comfortable about how it operates and how to add new instructions. Appendix B defines the instructions.

1) Suppose the RegWrite signal in a single-cycle RISC-V processor has a *stuck-at-1 fault* (i.e., the signal is always 1). Which instructions would malfunction, and why?

2) Modify the single-cycle RISC-V processor to implement the `blt` instruction. Mark up copies of the controller, main decoder, ALU decoder, and datapath (attached) to handle the new instruction as simply as possible. Name any control signals you need to add.

3) Modify the single-cycle RISC-V processor to implement the `sll` instruction. Mark up the Verilog (attached) to implement your changes as simply as possible.

4) Alyssa P. Hacker is a crack circuit designer. She offers to speed up one of the functional units in Table 7.7 by 50% (i.e., cut the delay in half) to improve the overall performance of the single-cycle processor. Which unit should she optimize, and by what percentage will the execution time improve?

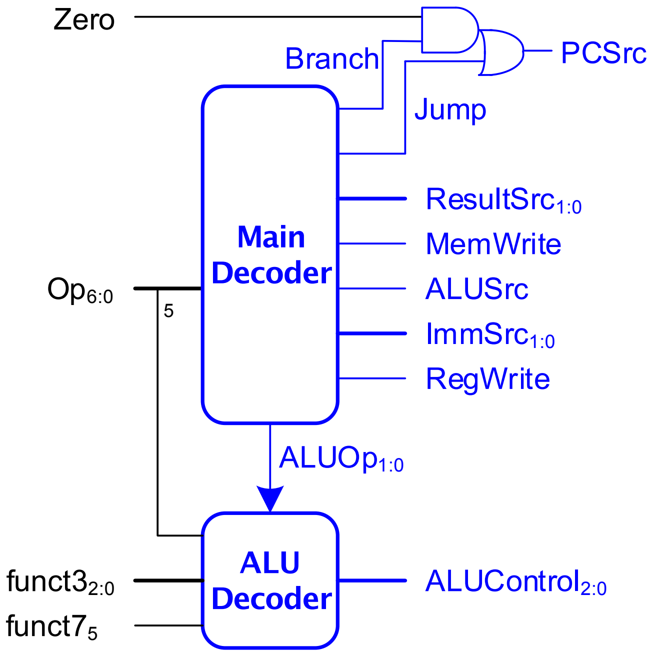
5) Impact on Society: RISC-V has gained tremendous attention in recent years. Explain the market forces that have caused this interest.

6) AI Question (Optional)
   This question must be solved by AI. Report what the AI produces, whether you believe it is accurate or a hallucination, and whether the solution is similar, better, or worse than what you would have done yourself in a reasonable amount of time.

   Modify the single-cycle RISC-V processor from Digital Design and Computer Architecture to implement a load halfword instruction.

How long did you spend on this problem set? This will not count toward your grade but will help calibrate the workload.

Problem 2: Controller

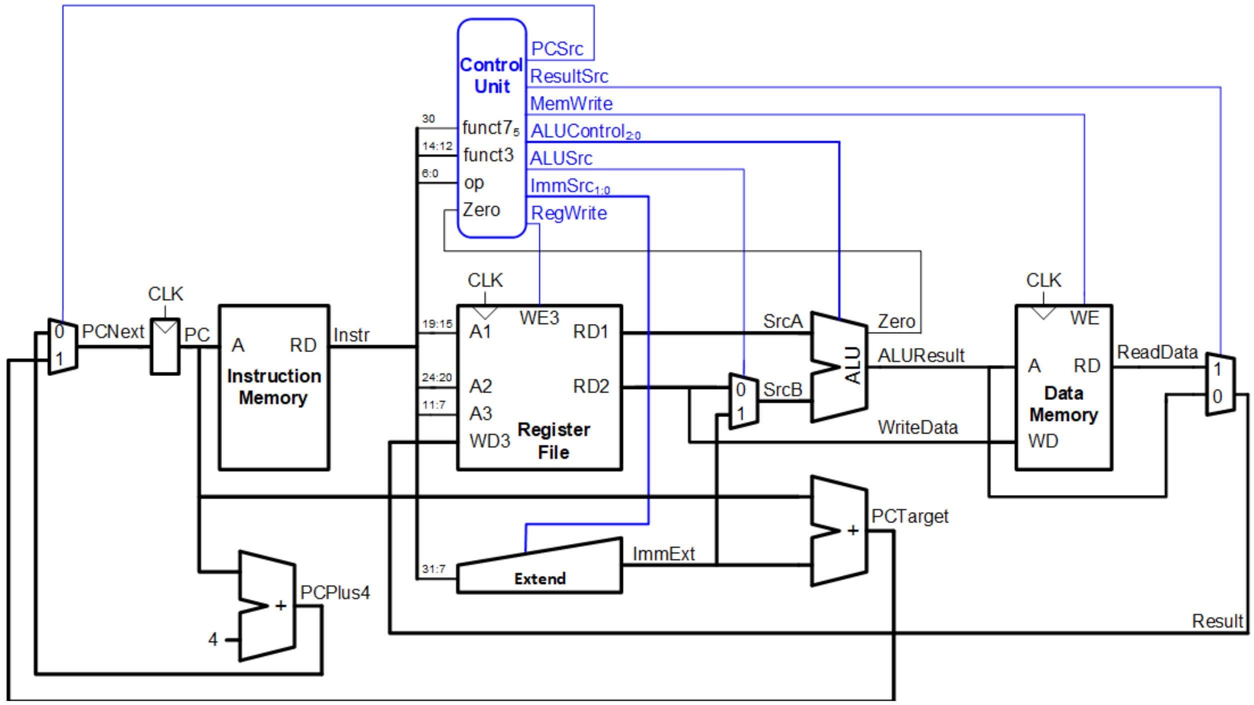


Problem 2 ALU Decoder

ALUOp	funct3	op_5, $funct7_5$	ALUControl	Instruction
00	XXX	XX	000 (add)	lw, sw
01	XXX	XX	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
10	000	11	001 (subtract)	sub
10	010	XX	101 (set less than)	slt
10	110	XX	011 (or)	or
10	111	XX	010 (and)	and

Problem 2 Main Decoder

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	00	1	0	10	0	00	0
sw	0100011	0	01	1	1	XX	0	00	0
R-type	0110011	1	XX	0	0	01	0	10	0
beq	1100011	0	10	0	0	XX	1	01	0
addi	0010011	1	00	0	0	01	0	10	0
jal	0111111	1	11	X	0	00	0	XX	1



Note: To support JAL, extend the Result mux to 3 inputs, as shown in Fig 7.15 but not important for this problem.

Problem 3: Single-Cycle Processor Verilog

```
module riscv(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       alusrc, regwrite, jump;
  logic [1:0] memtoreg, immsrc;
  logic [2:0] alucontrol;

  controller c(instr[30], instr[14:12], instr[6:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regwrite, jump,
               immsrc, alucontrol);
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regwrite,
              immsrc, alucontrol,
              zero, pc, instr,
              aluresult, writedata, readdata);
endmodule

module controller(input  logic       f7b5,
                  input  logic [2:0] funct3,
                  input  logic [6:0] op,
                  input  logic       zero,
                  output logic [1:0] memtoreg,
                  output logic       memwrite,
                  output logic       pcsrc, alusrc,
                  output logic       regwrite, jump,
                  output logic [1:0] immsrc,
                  output logic [2:0] alucontrol);

  logic [1:0] aluop;
  logic       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regwrite, jump, immsrc, aluop);
  aludec  ad(f7b5, op[5], funct3, aluop, alucontrol);
  //f7b5 = function 7 bit 5 (for R-type instructions)

  assign pcsrc = branch & zero | jump;
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] memtoreg,
               output logic       memwrite,
               output logic       branch, alusrc,
               output logic       regwrite, jump,
               output logic [1:0] immsrc,
               output logic [1:0] aluop);

  logic [10:0] controls;

  assign {regwrite, immsrc, alusrc, branch, memwrite,
          memtoreg, jump, aluop} = controls;

  always_comb
    casez(op)
                // regwrite_immsrc_alusrc_branch_memwrite_memtoreg_jump_aluop
      7'b0110011: controls <= 11'b1_xx_0_0_0_00_0_10; // R-type data processing
      7'b0010011: controls <= 11'b1_00_1_0_0_00_0_10; // I-type data processing
      7'b0000011: controls <= 11'b1_00_1_0_0_01_0_00; // LW
      7'b0100011: controls <= 11'b0_01_1_0_1_00_0_00; // SW
      7'b1100011: controls <= 11'b0_10_0_1_0_00_0_01; // BEQ
      7'b1101111: controls <= 11'b1_11_0_0_0_10_1_00; // JAL
      default:    controls <= 11'bxxxxxxxxxxx; //???
    endcase
endmodule
```

```
module aludec(input  logic       f7b5, op5,
              input  logic [2:0] funct3,
              input  logic [1:0] aluop,
              output logic [2:0] alucontrol);

  logic addSubType;
  assign addSubType = f7b5 & op5;
  always_comb
    case(aluop)
      2'b00: alucontrol <= 3'b000;  // add
      2'b01: alucontrol <= 3'b001;  // sub
      default: case({addSubType, funct3})        // R- or I-type
          4'b0000: alucontrol <= 3'b000; // ADD
          4'b1000: alucontrol <= 3'b001; // SUB
          4'b0111: alucontrol <= 3'b010; // AND
          4'b0110: alucontrol <= 3'b011; // OR
          4'b0010: alucontrol <= 3'b101; // SLT
          default: alucontrol <= 3'bxxx; // ???
        endcase
    endcase
endmodule

module datapath(input  logic        clk, reset,
                input  logic [1:0]  memtoreg,
                input  logic        pcsrc, alusrc,
                input  logic        regwrite,
                input  logic [1:0]  immsrc,
                input  logic [2:0]  alucontrol,
                output logic        zero,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluresult, writedata,
                input  logic [31:0] readdata);

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcplus4, pcbranch;
  logic [31:0] immext;
  logic [31:0] srca, srcb;
  logic [31:0] result;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd4(pc, 32'd4, pcplus4);
  adder       pcaddbranch(pc, immext, pcbranch);
  mux2 #(32)  pcmux(pcplus4, pcbranch, pcsrc, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[19:15], instr[24:20],
                 instr[11:7], result, srca, writedata);
  immext      immextnd(instr[31:7], immsrc, immext);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, immext, alusrc, srcb);
  alu         alu(srca, srcb, alucontrol, aluresult, zero);
  mux3 #(32)  resmux(aluresult, readdata, pcplus4, memtoreg, result);
endmodule

module immext(input  logic [31:7] instr,
              input  logic [1:0]  immsrc,
              output logic [31:0] extimm);

  always_comb
    case(immsrc)
               // I-type (Data processing with immediate and loads)
      2'b00:   extimm = {{21{instr[31]}}, instr[30:20]};
               // S-type (Stores)
      2'b01:   extimm = {{21{instr[31]}}, instr[30:25], instr[11:7]};
               // B-type (Branches)
      2'b10:   extimm = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
               // U-type (Jumps)
      2'b11:   extimm = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
      default: extimm = 32'bx; // undefined
    endcase
```

```
endmodule

module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

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

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 0 hardwired to 0

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

  assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
  assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule

module alu(input  logic [31:0] a, b,
           input  logic [2:0]  alucontrol,
           output logic [31:0] result,
           output logic        zero);

  logic [31:0] condinvb, sum;

  assign condinvb = (alucontrol[1:0] == 2'b01) ? ~b : b; // for subtraction or slt
  assign sum = a + condinvb + alucontrol[2];

  always_comb
    case (alucontrol)
      3'b000: result = sum;          // addition
      3'b001: result = sum;          // subtraction
      3'b010: result = a & b;        // and
      3'b011: result = a | b;        // or
      3'b101: result = sum[31];      // slt
      default: result = 0;
  endcase

  assign zero = (result == 32'b0);
endmodule
```