

Lab 11: Single-Cycle Processor SOLUTIONS

Digital Design and Computer Architecture: RISC-V Edition (Harris & Harris, Elsevier © 2021)

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.
2. A marked up version of Figure 1 (single-cycle processor) showing the needed modifications (`lui` and `xor`)

Answer:

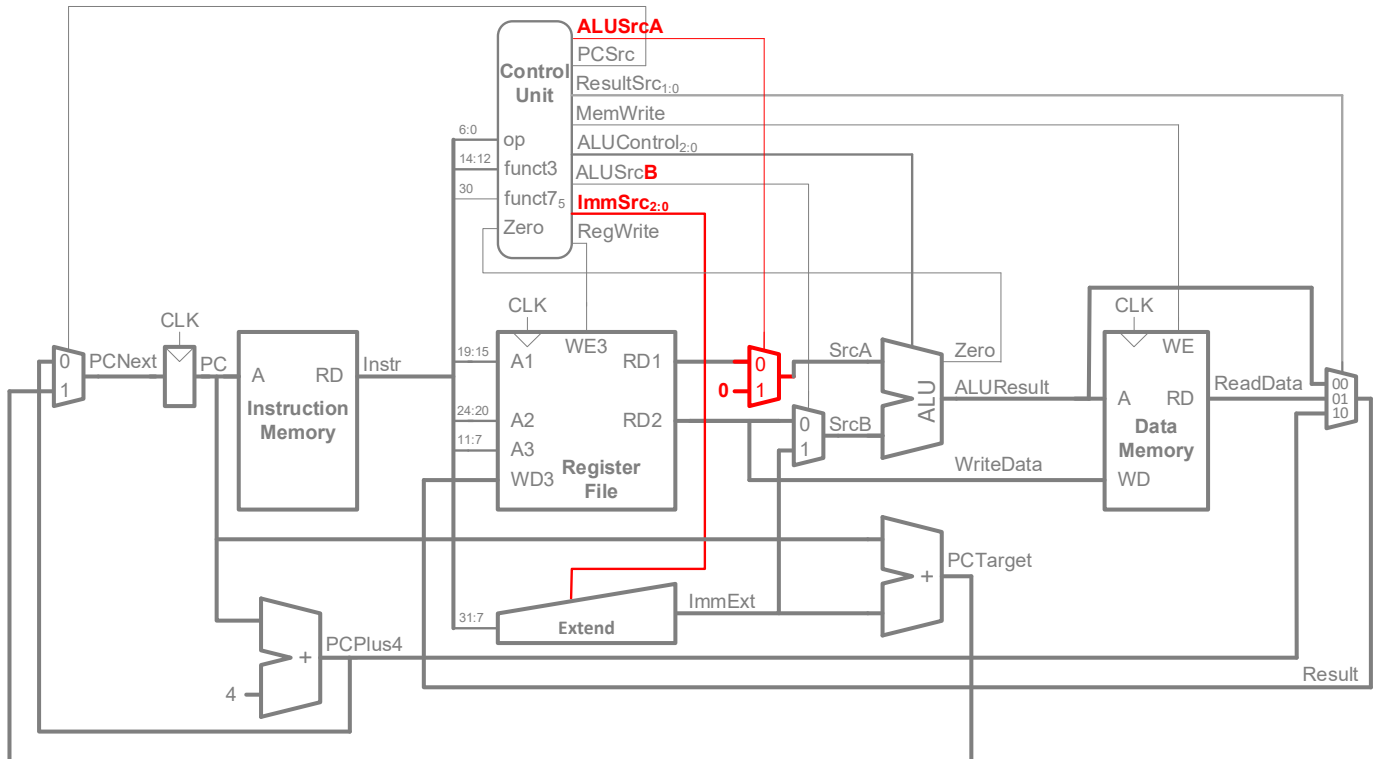


Figure 1: RISC-V single-cycle processor

3. A marked up version of Figures 2 and 3 (if modified) showing the needed modifications for the additional instructions (`lui` and `xor`)

Answer:

4. Amended Main Decoder, ALU Decoder, and ImmSrc truth tables to support `lui` and `xor`

Answer:

Table 1. Main Decoder Truth Table

Instruction	Opcode	RegWrite	ImmSrc	ALUSrcA	ALUSrcB	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	000	0	1	0	01	0	00	0
sw	0100011	0	001	0	1	1	xx	0	00	0
R-type	0110011	1	xxx	0	0	0	00	0	10	0
beq	1100011	0	010	0	0	0	xx	1	01	0
I-type ALU	0010011	1	000	0	0	0	00	0	10	0
jal	1101111	1	011	x	x	0	10	0	xx	1
lui	0110111	1	100	1	1	0	00	0	00	0

Table 2: ALU Decoder Truth Table to support `xor`

ALUOp	funct3	{op5, funct7 ₅ }	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add, addi
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt, slti
	100	x	100 (xor)	xor, xori
	110	x	011 (or)	or, ori
	111	x	010 (and)	and, andi

Table 3. ImmSrc encoding to support `lui`

ImmSrc	ImmExt	Type	Description
000	{{20{Instr[31]}}, Instr[31:20]}	I	12-bit signed immediate
001	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}	S	12-bit signed immediate
010	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}	B	13-bit signed immediate
011	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}	J	21-bit signed immediate
100	{Instr[31:12], 12'b0}	U	20-bit signed immediate

5. Amended SystemVerilog code to add support for `lui` and `xor`

Answer:

```

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 == 108 & WriteData == 32'hABCDE7D5) 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] WriteData, DataAdr,
            output logic MemWrite);

    logic [31:0] PC, Instr, ReadData;

    // instantiate processor and memories
    riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite, DataAdr,
                        WriteData, ReadData);
    imem imem(PC, Instr);
    dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule

module riscvsingle(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 ALUSrcA, ALUSrcB; // lui
    logic RegWrite, Jump, Zero;
    logic [1:0] ResultSrc;
    logic [2:0] ImmSrc; // lui
    logic [2:0] ALUControl;

    controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,
                ResultSrc, MemWrite, PCSrc,
                ALUSrcA, ALUSrcB, RegWrite, Jump,
                ImmSrc, ALUControl);
    datapath dp(clk, reset, ResultSrc, PCSrc,

```

```

        ALUSrcA, ALUSrcB, RegWrite,
        ImmSrc, ALUControl,
        Zero, PC, Instr,
        ALUResult, WriteData, ReadData);
endmodule

module controller(input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic      funct7b5,
                  input  logic      Zero,
                  output logic [1:0] ResultSrc,
                  output logic      MemWrite,
                  output logic      PCSrc,
                  output logic      ALUSrcA, ALUSrcB, // lui
                  output logic      RegWrite, Jump,
                  output logic [2:0] ImmSrc,          // lui: 3 bits
                  output logic [2:0] ALUControl);

    logic [1:0] ALUOp;
    logic      Branch;

    maindec md(op, ResultSrc, MemWrite, Branch,
               ALUSrcA, ALUSrcB, RegWrite, Jump, ImmSrc, ALUOp);
    aludec ad(op[5], funct3, funct7b5, ALUOp, ALUControl);

    assign PCSrc = Branch & Zero | Jump;
endmodule

module maindec(input  logic [6:0] op,
               output logic [1:0] ResultSrc,
               output logic      MemWrite,
               output logic      Branch,
               output logic      ALUSrcA, ALUSrcB, // lui
               output logic      RegWrite, Jump,
               output logic [2:0] ImmSrc,    // lui: 3 bits
               output logic [1:0] ALUOp);

    logic [12:0] controls;

    assign {RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite,
           ResultSrc, Branch, ALUOp, Jump} = controls;

    always_comb
    case(op)
        // RegWrite ImmSrc ALUSrcA ALUSrcB MemWrite ResultSrc Branch ALUOp Jump
        7'b0000011: controls = 13'b1_000_0_1_0_01_0_00_0; // lw
        7'b0100011: controls = 13'b0_001_0_1_1_00_0_00_0; // sw
        7'b0110011: controls = 13'b1_xxx_0_0_0_00_0_10_0; // R-type
        7'b1100011: controls = 13'b0_010_0_0_0_00_1_01_0; // beq
        7'b0010011: controls = 13'b1_000_0_1_0_00_0_10_0; // I-type ALU
        7'b1101111: controls = 13'b1_011_0_1_0_10_0_00_1; // jal
        7'b0110111: controls = 13'b1_100_1_1_0_00_0_00_0; // lui
        default:    controls = 13'bx_xxx_x_x_x_xx_x_xx_x; // non-implemented
    endcase
endmodule

module aludec(input  logic      opb5,
```

```

        input logic [2:0] funct3,
        input logic      funct7b5,
        input logic [1:0] ALUOp,
        output logic [2:0] ALUControl);

logic RtypeSub;
assign RtypeSub = funct7b5 & opb5; // TRUE for R-type subtract instruction

always_comb
case(ALUOp)
2'b00:          ALUControl = 3'b000; // addition
2'b01:          ALUControl = 3'b001; // subtraction
default: case(funct3) // R-type or I-type ALU
3'b000: if (RtypeSub)
        ALUControl = 3'b001; // sub
        else
        ALUControl = 3'b000; // add, addi
3'b010: ALUControl = 3'b101; // slt, slti
3'b110: ALUControl = 3'b011; // or, ori
3'b111: ALUControl = 3'b010; // and, andi
3'b100: ALUControl = 3'b100; // xor, xori
        default: ALUControl = 3'bxxx; // ???
endcase
endcase
endmodule

module datapath(input logic      clk, reset,
input logic [1:0] ResultSrc,
input logic      PCSrc,
input logic      ALUSrcA, ALUSrcB, // lui
input logic      RegWrite,
input logic [2:0] ImmSrc, // lui: 3-bit
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 [31:0] PCNext, PCPlus4, PCTarget;
logic [31:0] ImmExt;
logic [31:0] SrcA, SrcB;
logic [31:0] A; // lui
logic [31:0] Result;

// next PC logic
flop # (32) pcreg(clk, reset, PCNext, PC);
adder      pcadd4(PC, 32'd4, PCPlus4);
adder      pcaddbranch(PC, ImmExt, PCTarget);
mux2 # (32) pcmux(PCPlus4, PCTarget, PCSrc, PCNext);

// register file logic
regfile    rf(clk, RegWrite, Instr[19:15], Instr[24:20],
              Instr[11:7], Result, A, WriteData);
extend     ext(Instr[31:7], ImmSrc, ImmExt);

// ALU logic

```

```

    mux2 #(32) srcamux(A, 32'b0, ALUSrcA, SrcA); // lui
    mux2 #(32) srcbmux(WriteData, ImmExt, ALUSrcB, SrcB);
    alu      alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
    mux3 #(32) resultmux(ALUResult, ReadData, PCPlus4, ResultSrc, Result);
endmodule

module regfile(input logic clk,
               input logic we3,
               input logic [4:0] a1, a2, a3,
               input logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

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

    // three ported register file
    // read two ports combinationaly (A1/RD1, A2/RD2)
    // write third port on rising edge of clock (A3/WD3/WE3)
    // register 0 hardwired to 0

    always_ff @(posedge clk)
        if (we3) rf[a3] <= wd3;

    assign rd1 = (a1 != 0) ? rf[a1] : 0;
    assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule

module adder(input [31:0] a, b,
             output [31:0] y);

    assign y = a + b;
endmodule

module extend(input logic [31:7] instr,
               input logic [2:0] immsrc, // lui
               output logic [31:0] immext);

    always_comb
        case(immsrc)
            // I-type
            3'b000: immext = {{20{instr[31]}}, instr[31:20]};
            // S-type (stores)
            3'b001: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
            // B-type (branches)
            3'b010: immext = {{20{instr[31]}}, instr[7], instr[30:25],
instr[11:8], 1'b0};
            // J-type (jal)
            3'b011: immext = {{12{instr[31]}}, instr[19:12], instr[20],
instr[30:21], 1'b0};
            // U-type (lui)
            3'b100: immext = {instr[31:12], 12'b0};
            default: immext = 32'bx; // undefined
        endcase
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 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 imem(input  logic [31:0] a,
            output logic [31:0] rd);

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

    initial
        $readmemh("riscvtest.txt",RAM);

    assign rd = RAM[a[31:2]]; // word aligned
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 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;
    logic        v;           // overflow
    logic        isAddSub;     // true when is add or subtract operation

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

```



```

assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                 ~alucontrol[1] & alucontrol[0];

always_comb
case (alucontrol)
  3'b000: result = sum;          // add
  3'b001: result = sum;          // subtract
  3'b010: result = a & b;        // and
  3'b011: result = a | b;        // or
  3'b100: result = a ^ b;        // xor
  3'b101: result = sum[31] ^ v; // slt
  3'b110: result = a << b[4:0]; // sll
  3'b111: result = a >> b[4:0]; // srl
  default: result = 32'bx;
endcase

assign zero = (result == 32'b0);
assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

endmodule

```

6. Modified test program that uses lui and xor

Answer:

Answers will vary – but the modified test program must fail if xor or lui doesn't work correctly.

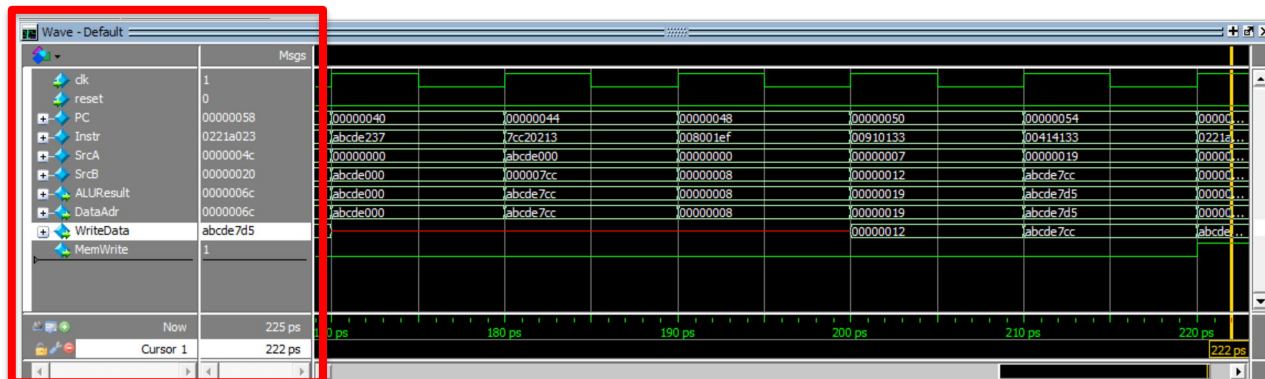
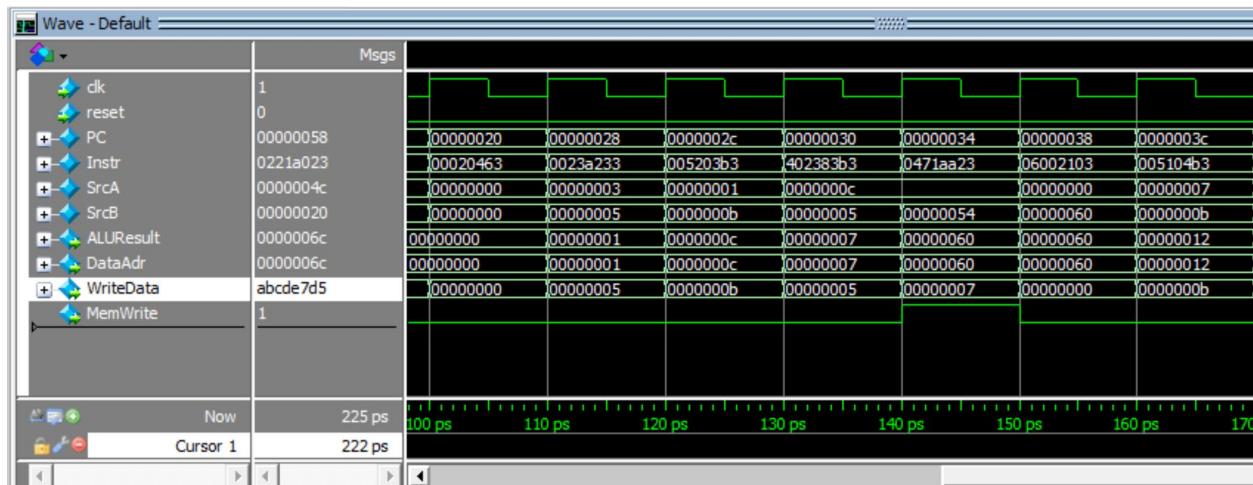
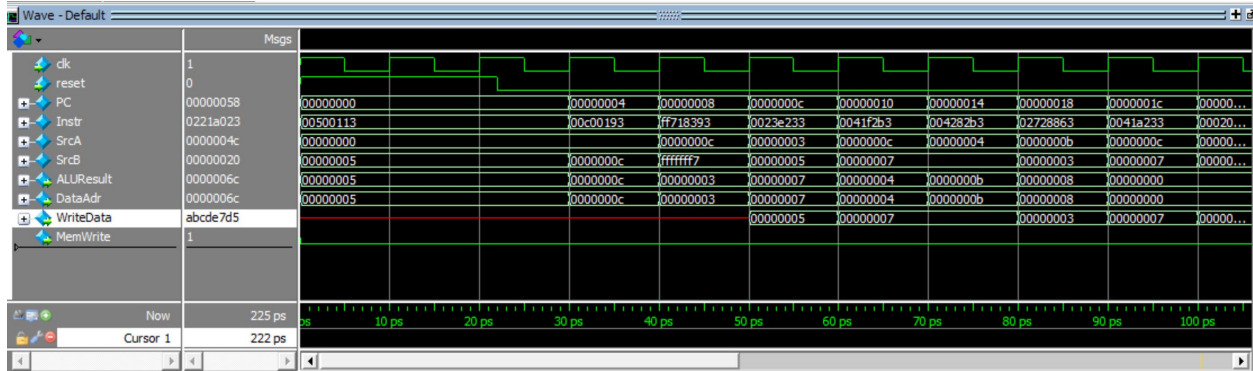
```

# If successful, it should write the value 0xABCDE7D5 to address 108
#
# RISC-V Assembly      Description      Address  Machine
main:  addi x2, x0, 5      # x2 = 5          0         00500113
      addi x3, x0, 12     # x3 = 12          4         00C00193
      addi x7, x3, -9     # x7 = (12 - 9) = 3      8         FF718393
      or x4, x7, x2       # x4 = (3 OR 5) = 7      C         0023E233
      and x5, x3, x4      # x5 = (12 AND 7) = 4    10        0041F2B3
      add x5, x5, x4      # x5 = (4 + 7) = 11    14        004282B3
      beq x5, x7, end     # shouldn't be taken  18        02728863
      slt x4, x3, x4      # x4 = (12 < 7) = 0    1C        0041A233
      beq x4, x0, around  # should be taken    20        00020463
      addi x5, x0, 0      # shouldn't happen    24        00000293
around: slt x4, x7, x2     # x4 = (3 < 5) = 1     28        0023A233
      add x7, x4, x5      # x7 = (1 + 11) = 12   2C        005203B3
      sub x7, x7, x2      # x7 = (12 - 5) = 7    30        402383B3
      sw x7, 84(x3)       # [96] = 7          34        0471AA23
      lw x2, 96(x0)       # x2 = [96] = 7      38        06002103
      add x9, x2, x5      # x9 = (7 + 11) = 18  3C        005104B3
      lui x4, 0xABCDE     # ***x4 = 0xABCDE000 40        ABCDE237
      addi x4, x4, 0x7CC   # ***x4 = 0xABCDE7CC 44        7CC20213
      jal x3, end         # jump to end, x3 = 0x4C 48        008001EF
      addi x2, x0, 1      # shouldn't happen    4C        00100113
end:   add x2, x2, x9      # x2 = (7 + 18) = 25  50        00910133
      xor x2, x2, x4      # ***x2 = 0xABCDE7D5 54        00414133
      sw x2, 0x20(x3)     # mem[108] = 0xABCDE7D5 58        0221A023
done:  beq x2, x2, done   # infinite loop      5C        00210063

```

7. Simulation waveforms (in the order listed above: clk, reset, PC, Instr, SrcA, SrcB, ALUResult, DataAdr, WriteData, and MemWrite – all displayed in hexadecimal for ease of reading). Does your system pass your testbench? Circle or highlight the waves showing that the correct value is written to the correct address, and make sure it is legible.

Answer:



Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.