Digital Systems Design

Engr 378

Lab 6

Single-Cycle 32-bit MIPS32 CPU Core

Date:
5/11/21

Grade:

By:
Aaron Luong and Richard Couto

Problem Analysis

In this lab, we are creating a module that will take care of all the arithmetic that a CPU has to do. After we finish this, we place this module with other modules that have been provided to form a MIPS32 Core. We then translate pseudocode code to MIPS32 code and with help of a translator, translate the MIPS code to hex values for the FPGA's memory to understand.

Hardware Design

For the one module that we had to write, the hardware design consisted of three inputs and two outputs. In this module, there is one add gate, one or gate, one adder, and one comparer module. The comparer module consists of one add gate and one not gate. The not gate is connected to the first value of the inequality which is srcA in this case and is then connected to the and gate. The other value which is srcB in this case is connected directed to the and gate. The result of this comparison is the output of the and gate.

There also needed to be an 8x3 mux since the control signal has three bits. The inputs of the mux are connected to the respective separate operations and the rest is connected to srcA since it does not matter what these outputs are. The output of the mux is then connected to a nor gate. The nor gate is then connected to the zero output and the result of the mux is connected to the aluRslt output.

Verilog Modeling

For this lab, the only Verilog code that was written was to the ALU module since everything else had been written and was provided beforehand.

ALU:

```
module alu(srcA, srcB, aluCtrl, aluRslt, zero);

    input          [31:0]  srcA, srcB;
    input          [ 2:0]  aluCtrl;
    output reg     [31:0]  aluRslt;
    output reg             zero;

    always@(*)begin
        if(aluCtrl==3'b000)begin
            aluRslt = srcA & srcB;
        end
        else if(aluCtrl==3'b001)begin
            aluRslt = srcA | srcB;
        end
```

```
                else if(aluCtrl==3'b010)begin
                        aluRslt = srcA + srcB;
                end
                else if(aluCtrl==3'b110)begin
                        aluRslt = srcA + (~srcB + 32'b00000000000000000000000000000001);
                end
                else if(aluCtrl==3'b111)begin
                        aluRslt = srcA < srcB;
                end
                else begin
                        aluRslt = srcA;
                end
                if(aluRslt == 0)begin
                        zero = 1;
                end
                else begin
                        zero = 0;
                end
        end
endmodule
```

The point of the only module written is to determine which operation to carry out when a specific control signal is sent to the ALU. When the signal 000 is sent the result will be a bitwise and of two thirty-two bit numbers known as srcA and srcB. When the signal is 001 then the result will be the bitwise or of srcA and srcB and when the signal is 010, then the two numbers are added together. When the signal is 110 then srcA is subtracted from srcB and when the signal is 111, srcA is compared to srcB and if scrA is smaller then the result will be one, otherwise, the result will be zero. Also when the result of the output is zero then the zero output will become high and will stay low otherwise.

Results/ Verification

ModelSim:

```
module ALUTest();
        reg [31:0] srcA, srcB;
        reg [2:0] aluCtrl;
        wire [31:0] aluRslt;
        wire zero;
        parameter time_out = 100;
        initial $monitor (srcA, srcB, aluCtrl, aluRslt, zero);
        always begin
                #0 srcA=32'b00001010101010101010101000101010;
srcB=32'b00000100010100111000110100010100;
                #5 aluCtrl=3'b000;
                #5 aluCtrl=3'b001;
```

```
            #5 aluCtrl=3'b010;
            #5 aluCtrl=3'b110;
            #5 aluCtrl=3'b111;
            #5 aluCtrl=3'b101;
        end
        alu test (srcA, srcB, aluCtrl, aluRslt, zero);
endmodule
```

The ModelSim code is similar to all the other ModelSim code that has been written for this class with the only changes being the values of the input being changes as well as what the inputs and outputs are.

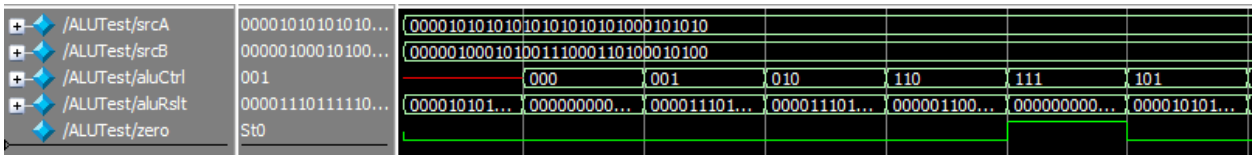


Fig1: Results of ModelSim Simulation

For the results of the ModelSim simulation, the ALU module works as intended with every possible type of input signal. The zero output also goes high when the resulting output of the ALU module is equal to zero. When the input signal is 000 the two 32-bit numbers are bitwise and together and when the input signal is 001 the two 32-bit numbers are bit-wise or together. When the input signal is 010, the two 32-bit numbers are added together and when the input signal is 110 the two 32-bit numbers are subtracted from each other. When the input signal is 111, the first 32-bit number is compared to the other 32-bit number and if the first one is smaller then the output is one and goes to zero otherwise.

FPGA:

Algorithm 1 Hex:

0x20090001
0x200A0000
0x200B0000
0x114B0001 or 0x08000001

Algorithm 1 Assembly:

```
addi t1 zero 0x1;
dest:
addi t2 zero  0x0;
addi t3 zero 0x0;
beq t2 t3 dest; or j dest;
```

Algorithm 2 Hex:

0x200C0001
0x20090002
0x01896824

0x01897025
0x01897820
0x0189C022
0x200A0000
0x200B0000
0x114B0006 or 0x08000006

Algorithm 2 Assembly:
addi t4 zero 0x1;
addi t1 zero 0x2;
and t5 t4 t1;
or t6 t5 t1;
add t7 t4 t1;
sub t8 t4 t1;
dest:
addi t2 zero 0x0;
addi t3 zero 0x0;
beq t2 t3 dest; or j dest;

Algorithm 3 Hex:
0x200C0001
0xADEC0000
0x8DE90000
0x200A0000
0x200B0000
0x114B0003 or 0x08000003

Algorithm 3 Assembly:
addi t4 zero 0x1;
sw t4 (t7);
lw t1 (t7);
dest:
addi t2 zero 0x0;
addi t3 zero 0x0;
beq t2 t3 dest; or j dest;

These hex values are the MIPS conversion of the assembly code given in the lab and these hex values are the hex values that are going to be inserted into the assembly memory of the FPGA. To make these values back into assembly code, you can convert the hex values in a MIPS code convertor. With these hex values, we can see if the entire program including the parts that were already written is working as intended. The code to interface with the FPGA was also provided so this wasn't written. For the assembly code written, (t7) is a random memory address location chosen to store values for algorithm 3 and can be any random place in memory.

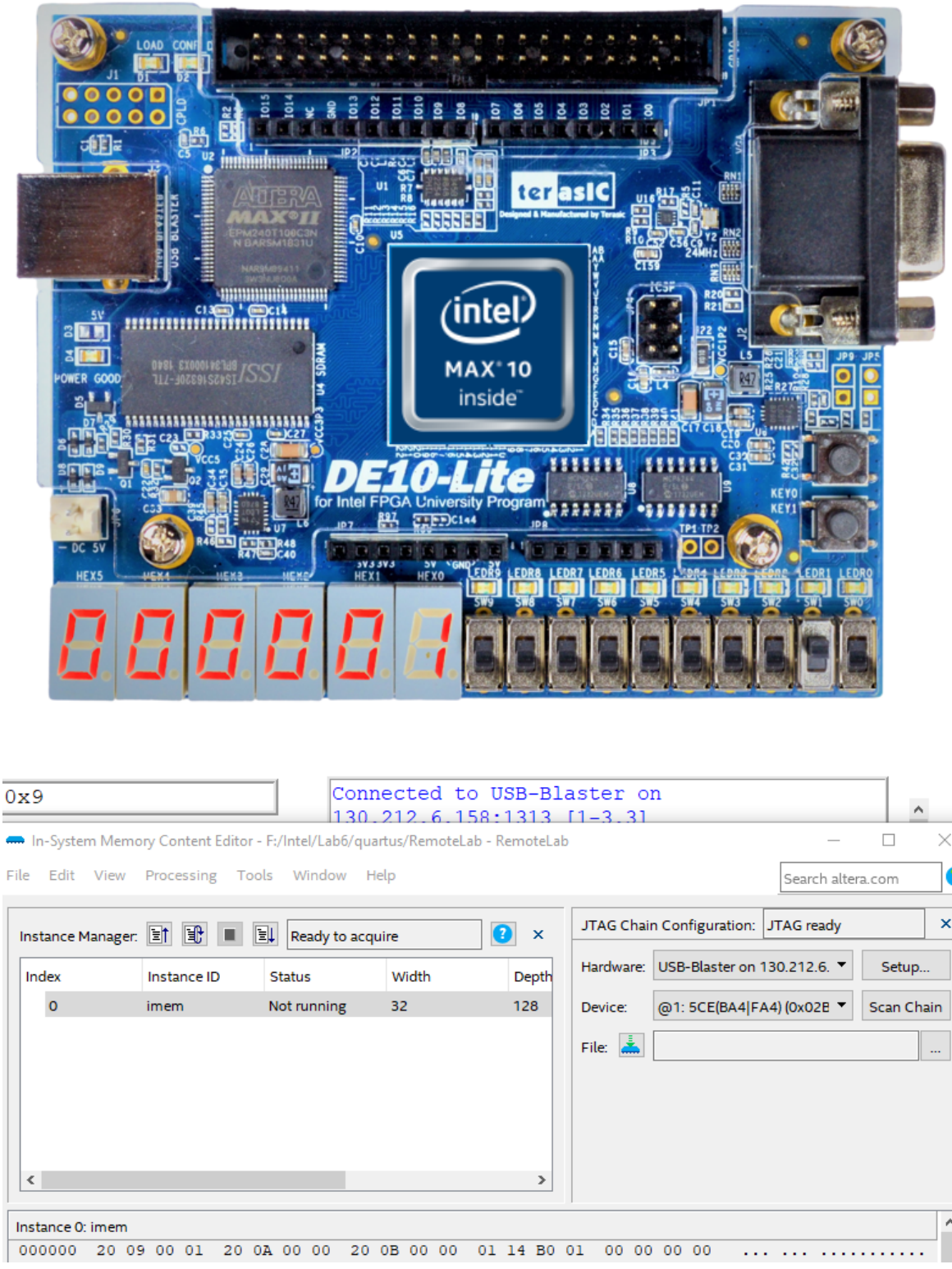


Fig2: Algorithm 1, Test of addi function (branch)

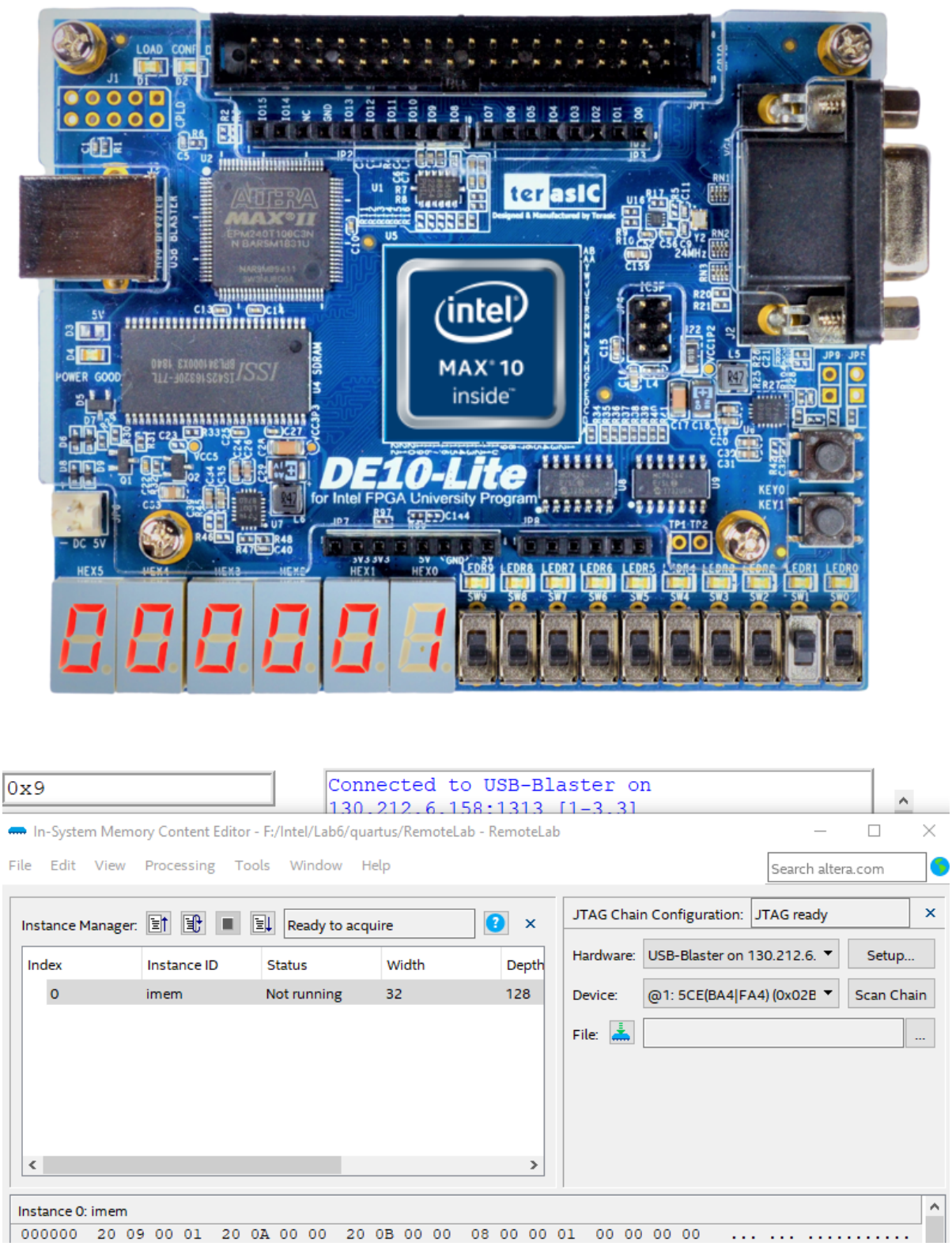


Fig3: Algorithm 1, Test of addi function (jump)

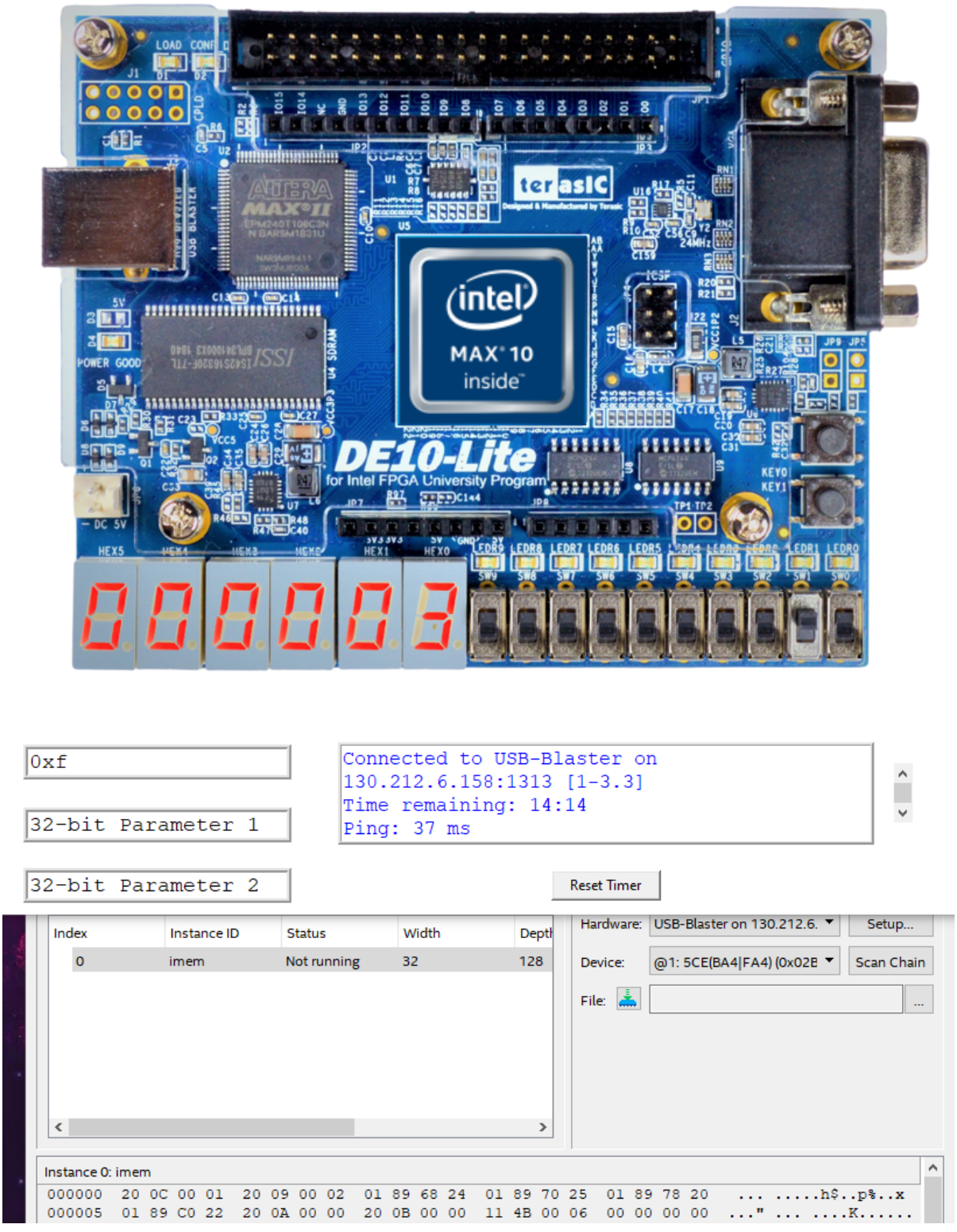


Fig4: Algorithm 2, test of add function (branch)

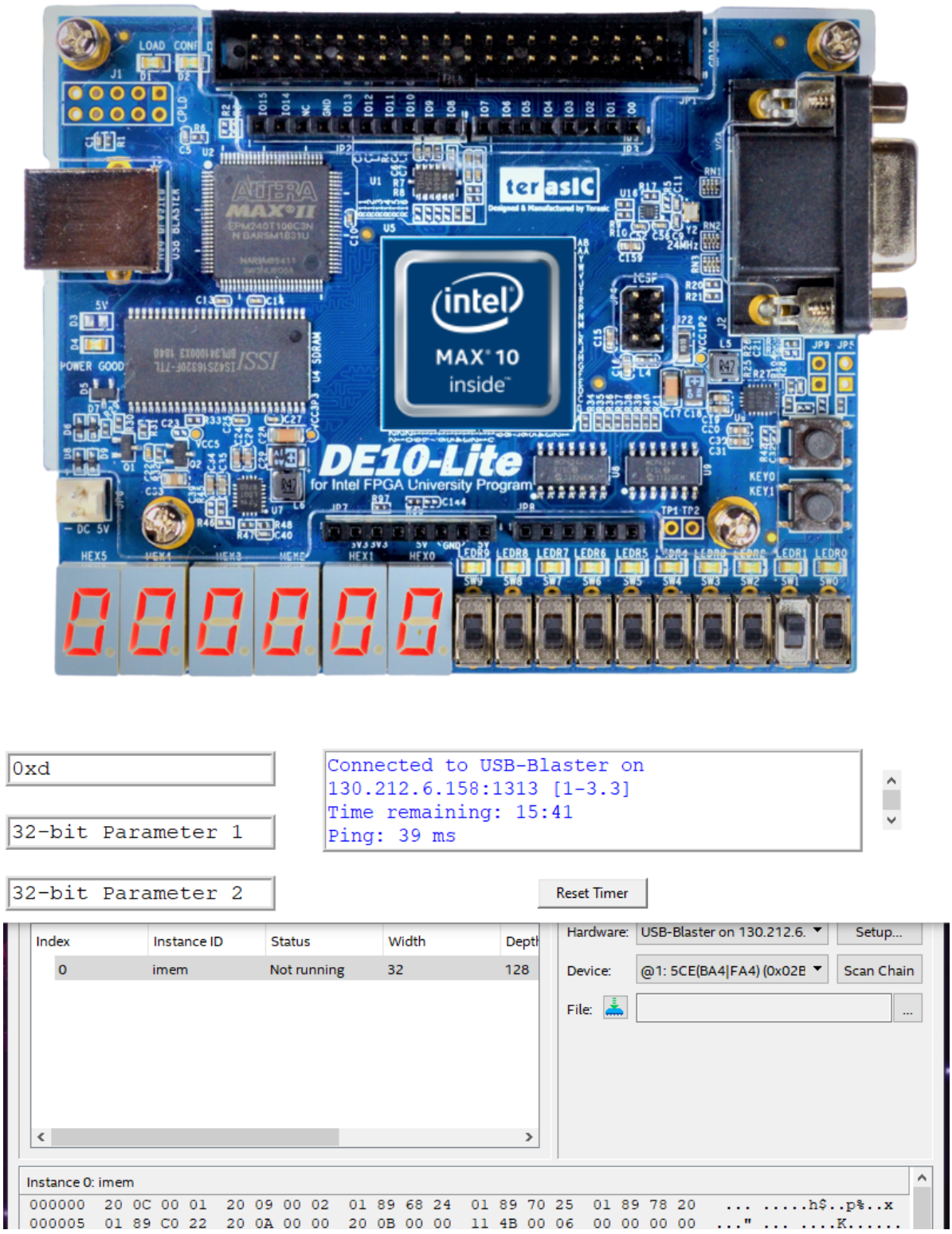


Fig5: Algorithm 2, test of and function (branch)

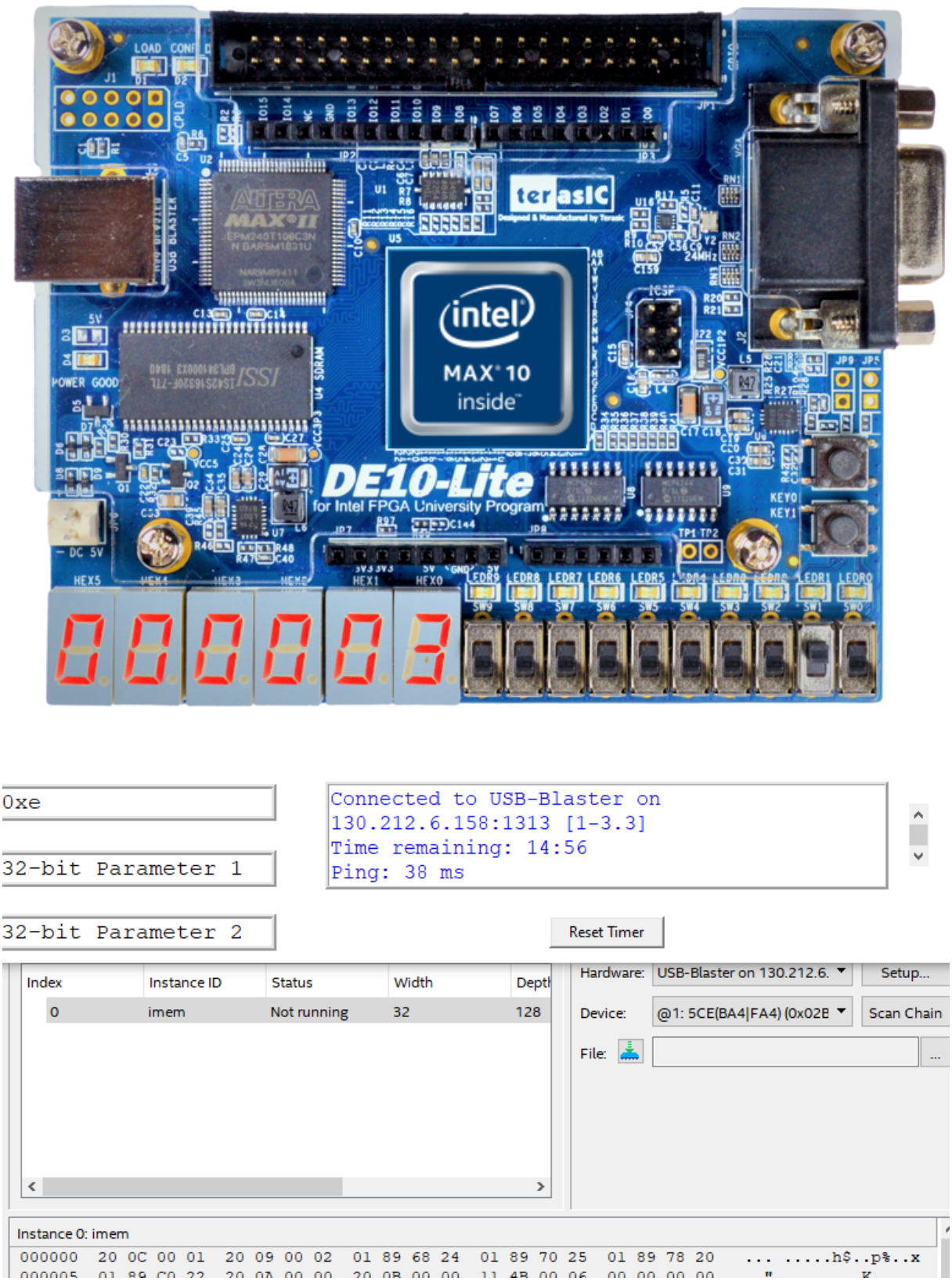


Fig6: Algorithm 2, test of or function (branch)

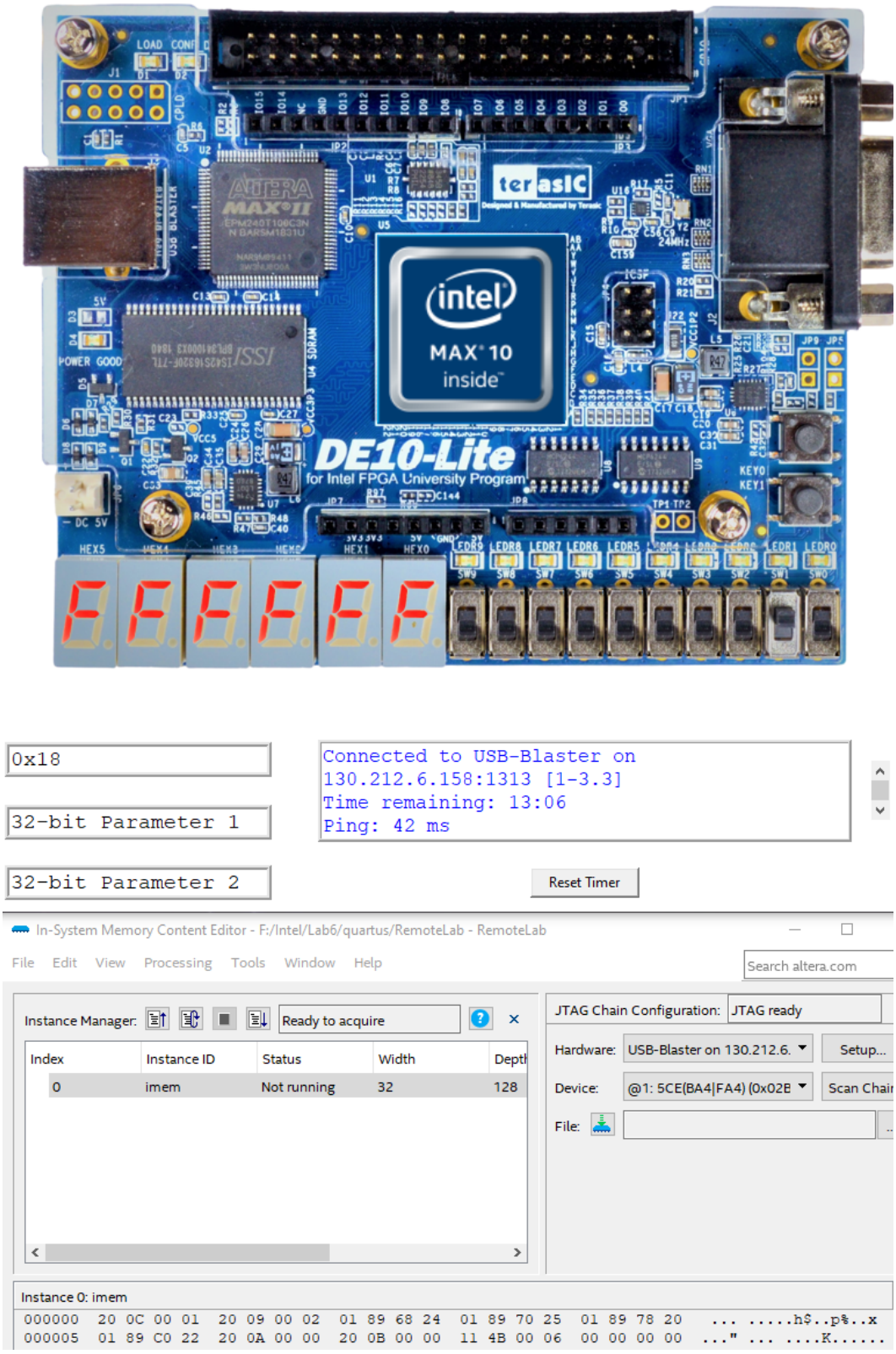
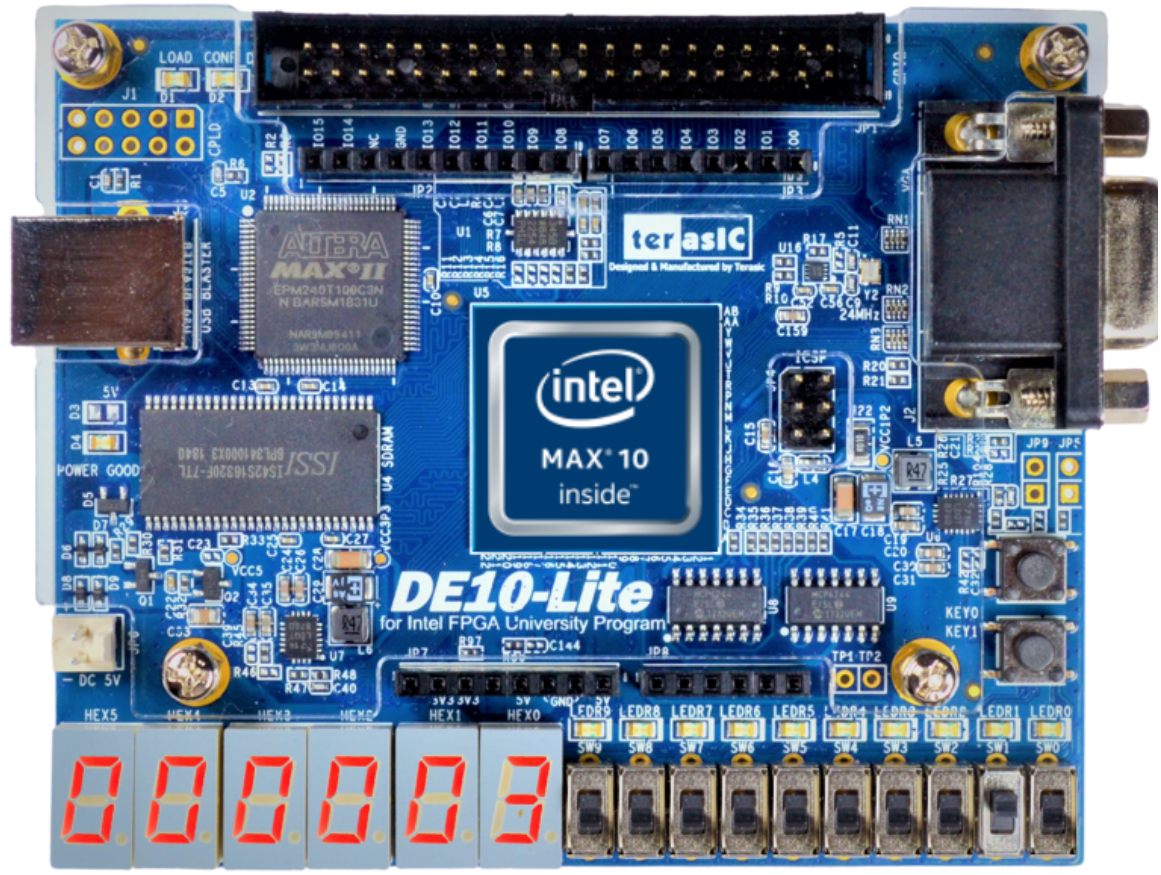


Fig7: Algorithm 2, test of subtracting function (branch)



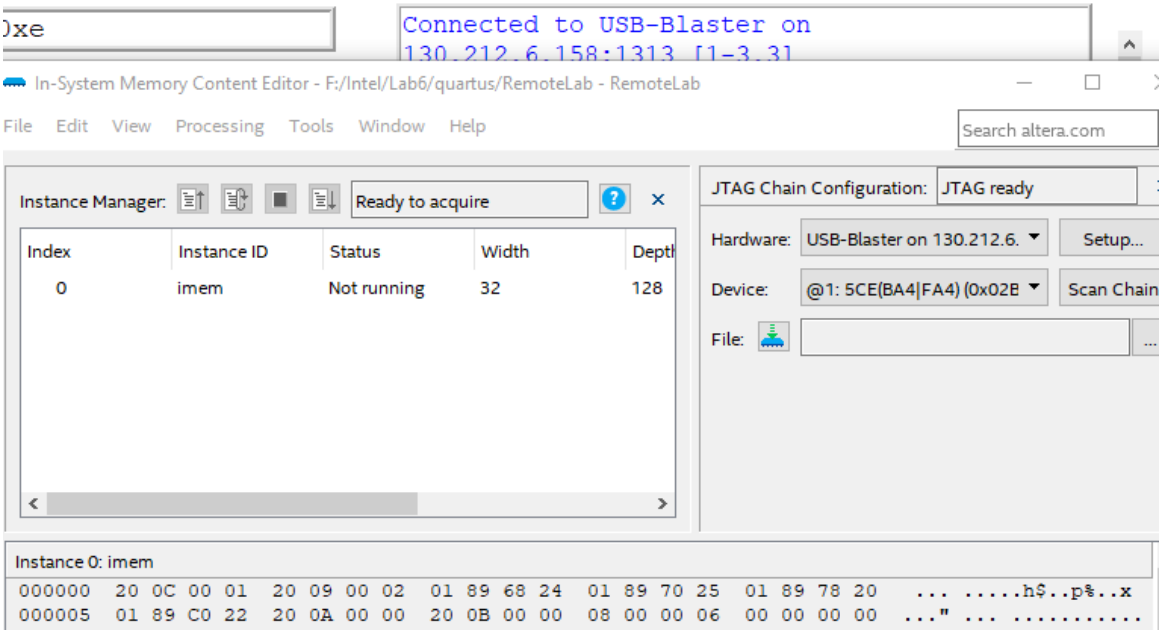


Fig8: Algorithm 2, test of or function (jump)

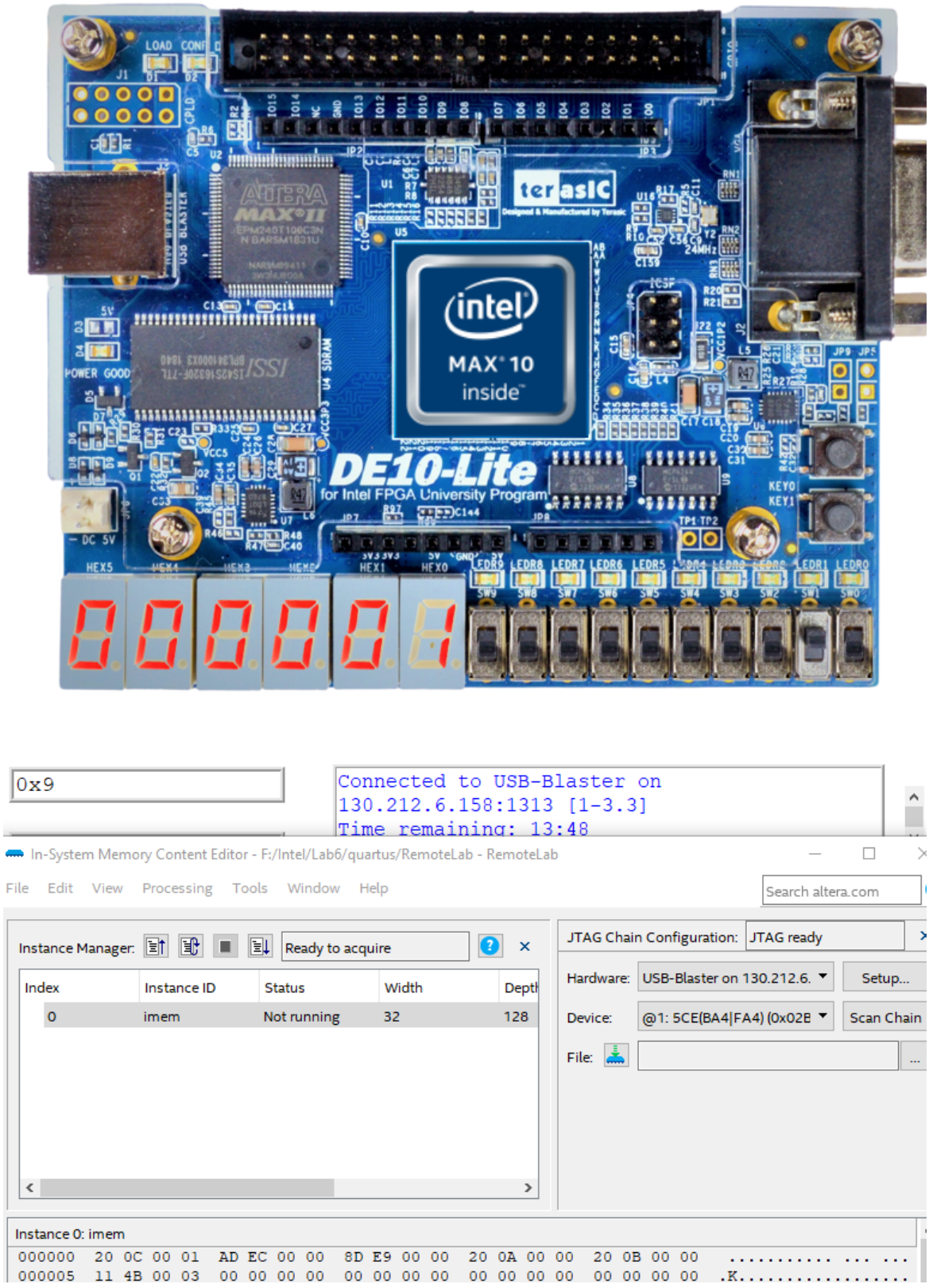


Fig9: Algorithm 3, test of load function (branch)

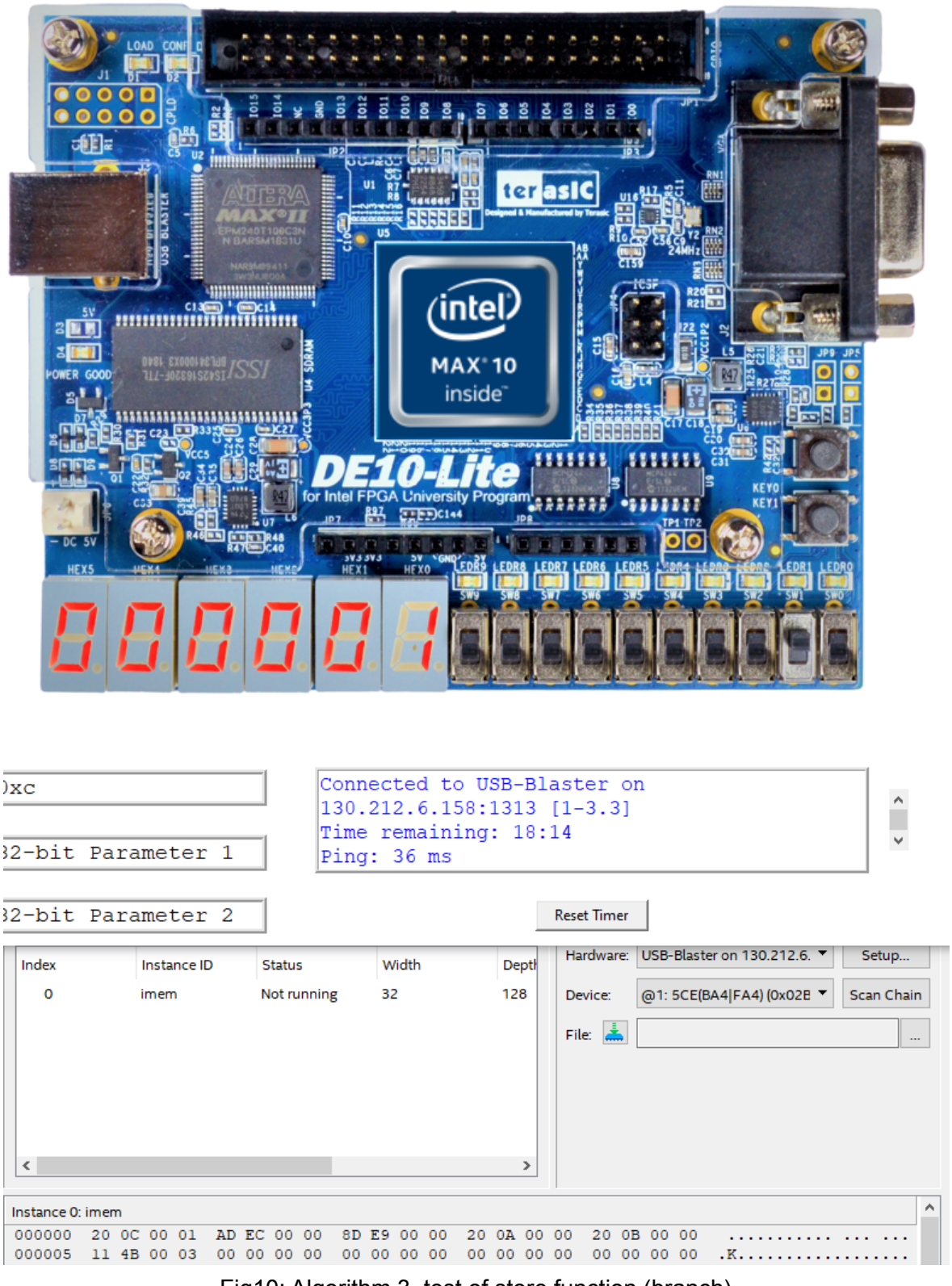


Fig10: Algorithm 3, test of store function (branch)

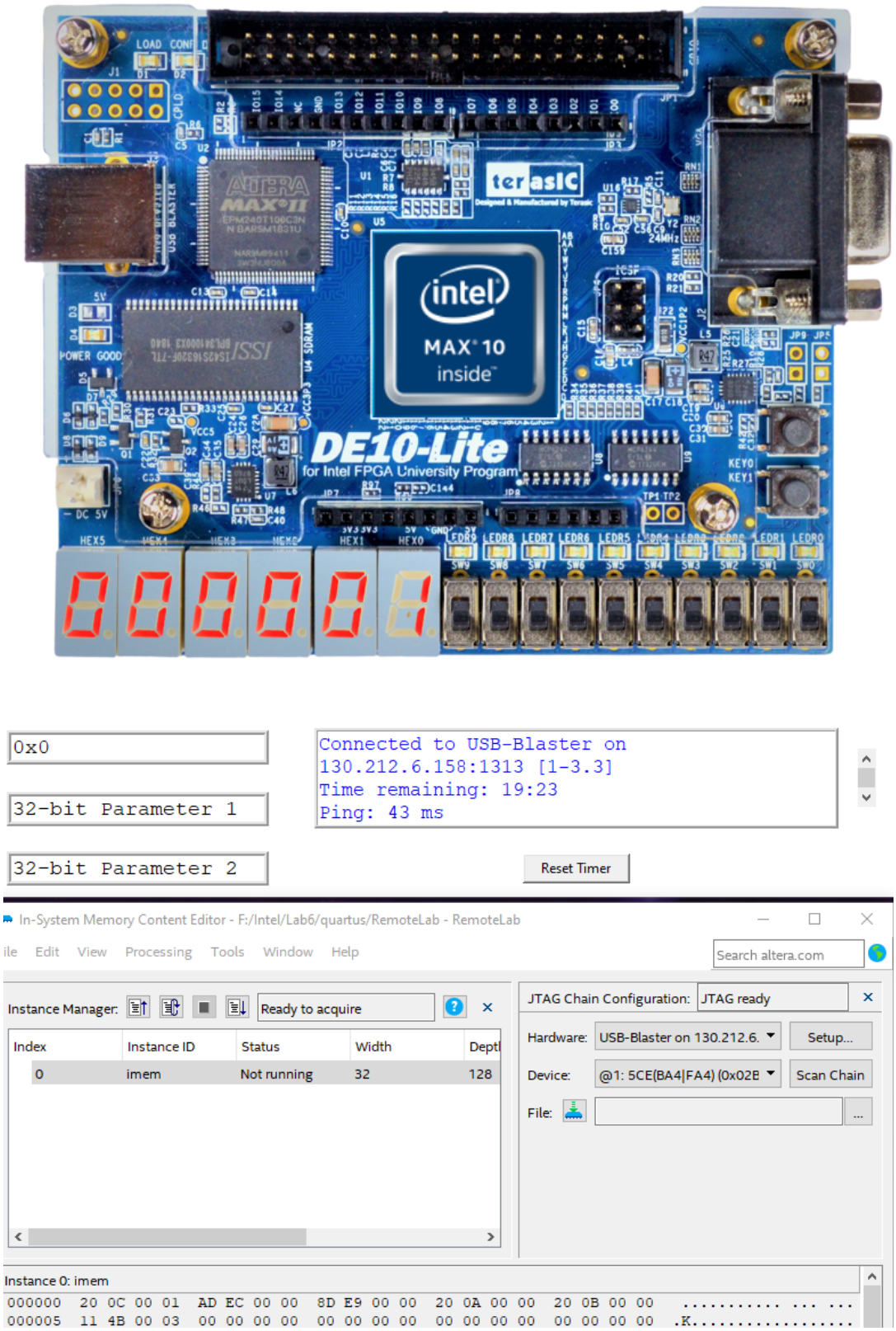


Fig11: Algorithm 3, memory location of $rw (branch)

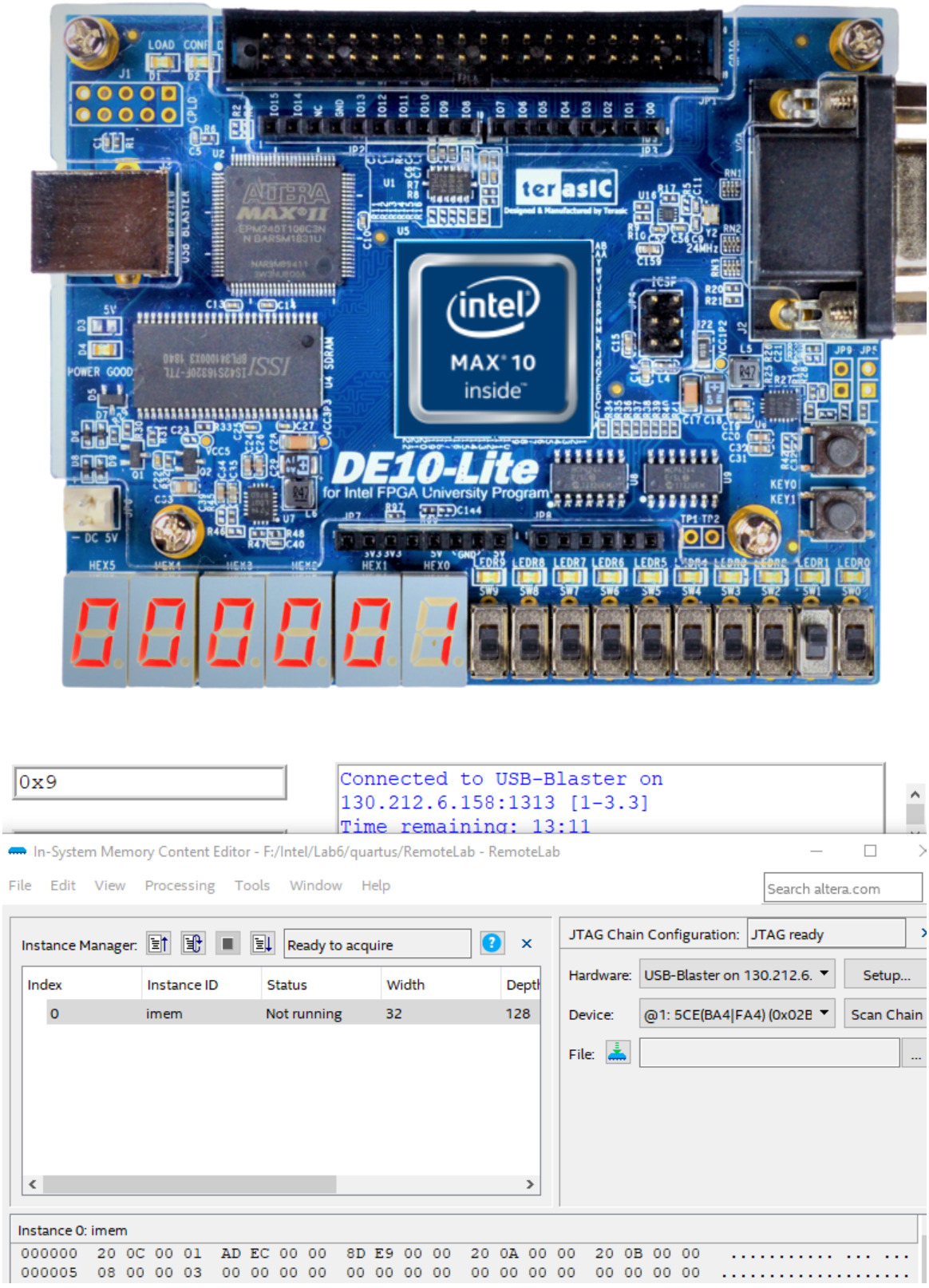


Fig12: Algorithm 3, test of store function (jump)

From all the results gathered from the FPGA test, we can see that the assemble code that was translated works as intended and we can also see that the code as a whole including the code that was provided works together as well. Some results are not included above because the only difference between the two different hex codes and assemble code of each algorithm is whether the last line is a jump command or a branch command. Since this does not affect the results that were gathered to tell whether everything works or not, only one type of image for the jump command is included for each algorithm. Also for the test, not every line is included in the test, but rather the commands that were not present in the previous algorithm. This is because if the command works as intended with the algorithm that came before the current algorithm, it will work in the current algorithm.

Conclusion and Discussion

In conclusion, the lab works as intended with the added ALU module. This module is the only module that was added to the project since everything else had been provided. The only issues that we had in this lab were figuring out how to transform the pseudocode to assemble code and how to transform that assembly code to hex values to put into the FPGA. This translation took up most of the time of this lab. For other issues, the only other issue was related to the testbench inputs. Since you need to input a 32-bit number for both srcA and srcB forgetting to put in a number in the sequence created issues that made our group think that there was something wrong with the ALU module. However by adding another zero to one of the 32-bit numbers, making it 32-bits long fixed this issue.

Work Breakdown

The ALU module as well as the ModelSim code was written and modified respectfully by Aaron. The conversion of the assembly code was done by Ricky. The FPGA testing was a joint effort by both parties.

HDL Source Code

```
module alu(srcA, srcB, aluCtrl, aluRslt, zero);

    input        [31:0]  srcA, srcB;
    input        [ 2:0]  aluCtrl;
    output reg   [31:0]  aluRslt;
    output reg           zero;

    always@(*)begin
        if(aluCtrl==3'b000)begin
            aluRslt = srcA & srcB;
        end
        else if(aluCtrl==3'b001)begin
```

```
                aluRslt = srcA | srcB;
            end
            else if(aluCtrl==3'b010)begin
                aluRslt = srcA + srcB;
            end
            else if(aluCtrl==3'b110)begin
                aluRslt = srcA + (~srcB + 32'b00000000000000000000000000000001);
            end
            else if(aluCtrl==3'b111)begin
                aluRslt = srcA < srcB;
            end
            else begin
                aluRslt = srcA;
            end
            if(aluRslt == 0)begin
                zero = 1;
            end
            else begin
                zero = 0;
            end
        end
endmodule

module ALUTest();
        reg [31:0] srcA, srcB;
        reg [2:0] aluCtrl;
        wire [31:0] aluRslt;
        wire zero;
        parameter time_out = 100;
        initial $monitor (srcA, srcB, aluCtrl, aluRslt, zero);
        always begin
                #0 srcA=32'b00001010101010101010101000101010;
srcB=32'b00000100010100111000110100010100;
                #5 aluCtrl=3'b000;
                #5 aluCtrl=3'b001;
                #5 aluCtrl=3'b010;
                #5 aluCtrl=3'b110;
                #5 aluCtrl=3'b111;
                #5 aluCtrl=3'b101;
        end
        alu test (srcA, srcB, aluCtrl, aluRslt, zero);
endmodule

module rf(ad1, ad2, adw, wData, clk, we, rData1, rData2, RFQUERY, RFOUT);
```

```
        input       [ 4:0]  ad1, ad2, adw, RFQUERY;
        input               clk, we;
        input       [31:0]  wData;
        output      [31:0]  rData1, rData2;
        output reg  [31:0]  RFOUT;

        reg         [31:0]  mem [31:0];

        assign rData1 = (|ad1  )      ? mem[ad1]    : 32'b0;
        assign rData2 = (|ad2  )      ? mem[ad2]    : 32'b0;

        always@(posedge clk) begin

                RFOUT  <= (|RFQUERY)  ? mem[RFQUERY] : 32'b0;

                if(we==1'b1)

                        mem[adw] <= wData;

        end
endmodule

module pc(pcp, clk, pc);

        input       [31:0]  pcp;
        input                       clk;
        output reg  [31:0]  pc;

        always@(posedge clk)
                pc<=pcp;

endmodule

module mips(rst, clk, QUERY, QSEL, seg0, seg1,
                        seg2, seg3, seg4, seg5);

        parameter INSTMEMSIZE = 128;
        parameter DATAMEMSIZE = 128;

        input                           rst, clk, QSEL;

        input       [9:0]               QUERY;
```

```
        wire            [4:0]           RFQUERY;
        wire            [31:0]          RFOUT;

        wire            [9:0]           DMQUERY;
        wire            [31:0]          DMOUT;

        output  [6:0]           seg0, seg1, seg2, seg3,
                                                seg4, seg5;

        wire                                    regW, regDst, aluSrc, brnch, memW,
                                                mem2Reg, jmp, zero;

        wire            [2:0]           aluCtrl;

        wire            [4:0]           adwReg;

        wire            [31:0]          inst, pc, pcp, srcB, srcA, result,
                                                aluRslt, rd2, rData;

        assign RFQUERY    = QUERY[4:0];
        assign DMQUERY      = QUERY[9:0];

        //You have to create the module for this instantiation!

        hexOut                          ho0(            .RFOUT(RFOUT),

.DMOUT(DMOUT),
                                                                .seg0(seg0),
                                                                .seg1(seg1),
                                                                .seg2(seg2),
                                                                .seg3(seg3),
                                                                .seg4(seg4),
                                                                .seg5(seg5),

.QSEL(QSEL));

        //You have to create the module for this instantiation!


         alu                                    al0(            .srcA(srcA),
                                                                .srcB(srcB),

.aluCtrl(aluCtrl),
```

```
.aluRslt(aluRslt),
                                                                .zero(zero));

        //No need to touch the modules below

        ctrlogic                    cl0(        .op(inst[31:26]),

.func(inst[5:0]),
                                                                .regW(regW),

.regDst(regDst),

.aluSrc(aluSrc),
                                                                .brnch(brnch),

.memW(memW),

.mem2Reg(mem2Reg),

.aluCtrl(aluCtrl),
                                                                .jmp(jmp),
                                                                .zero(zero));

        dplogic                     dl0(    .inst(inst),
                                                                .pc(pc),
                                                                .pcp(pcp),
                                                                .brnch(brnch),
                                                                .jmp(jmp),

.adwReg(adwReg),
                                                                .rst(rst),

.regDst(regDst),

.aluSrc(aluSrc),
                                                                .rd2(rd2),
                                                                .srcB(srcB),

.mem2Reg(mem2Reg),
                                                                .result(result),

.aluRslt(aluRslt),
```

```
                                                                .rData(rData));

        INSTMEM                                         imem( .address(pcp[6:2]),
                                                                .clock(clk),
                                                                .q(inst) );


        dmem  #(DATAMEMSIZE)            dm0(    .ad(aluRslt),
                                                                .we(memW),
                                                                .clk(clk),
                                                                .wData(rd2),
                                                                .rData(rData),

.DMQUERY(DMQUERY),

.DMOUT(DMOUT));

        pc                                              pc0(            .pcp(pcp),
                                                                .clk(clk),
                                                                .pc(pc));

        rf                                              rf0(            .ad1(inst[25:21]),

.ad2(inst[20:16]),

.adw(adwReg),
                                                                .wData(result),
                                                                .clk(clk),
                                                                .we(regW),
                                                                .rData1(srcA),
                                                                .rData2(rd2),

.RFQUERY(RFQUERY),

.RFOUT(RFOUT));

endmodule

module          hexOut(                 RFOUT,
                                                        DMOUT,
                                                        seg0,
                                                        seg1,
                                                        seg2,
                                                        seg3,
```

```
                                                    seg4,
                                                    seg5,
                                                    QSEL);

        input                   QSEL;
        input [31:0]    RFOUT, DMOUT;

        output [6:0]    seg0, seg1, seg2, seg3,
                                            seg4, seg5;

        wire   [3:0]    segin0, segin1, segin2, segin3,
                                            segin4, segin5;


        assign {segin5, segin4,
                        segin3, segin2,
                        segin1, segin0} = QSEL ? RFOUT : DMOUT;

        seg7dec i0(segin0, seg0);
        seg7dec i1(segin1, seg1);
        seg7dec i2(segin2, seg2);
        seg7dec i3(segin3, seg3);
        seg7dec i4(segin4, seg4);
        seg7dec i5(segin5, seg5);

endmodule

module seg7dec(in, out);

        input           [3:0] in;
        output reg      [6:0] out;

        always@(*)

                case(in)

                        4'h0: out <= 7'b1000000;
                        4'h1: out <= 7'b1111001;
                        4'h2: out <= 7'b0100100;
                        4'h3: out <= 7'b0110000;
                        4'h4: out <= 7'b0011001;
                        4'h5: out <= 7'b0010010;
                        4'h6: out <= 7'b0000010;
                        4'h7: out <= 7'b1111000;
```

```
                4'h8: out <= 7'b0000000;
                4'h9: out <= 7'b0011000;
                4'hA: out <= 7'b0001000;
                4'hB: out <= 7'b0000011;
                4'hC: out <= 7'b1000110;
                4'hD: out <= 7'b0100001;
                4'hE: out <= 7'b0000110;
                4'hF: out <= 7'b0001110;

        endcase

endmodule

module dmem #(MEMSIZE = 128)(ad, we, clk, wData, rData, DMQUERY, DMOUT);

        input   [31:0]  ad, wData;
        input   [ 9:0]  DMQUERY;
        input                           clk, we;
        output          [31:0]  rData;
        output reg [31:0] DMOUT;

        reg             [31:0]  mem [(MEMSIZE-1):0];

        assign rData   = mem[{ 2'b0,     ad[31:2]}];

        always@(posedge clk) begin

                DMOUT  <= mem[{20'b0, DMQUERY[ 9:0]}];

                if(we == 1'b1)

                        mem[{2'b00,ad[31:2]}] <= wData;

        end

endmodule

module dplogic(         inst, pc, pcp, brnch, jmp, adwReg, rst,
                                regDst, aluSrc, rd2, srcB, mem2Reg, result,
                                aluRslt, rData);

        input   [31:0]  inst, pc, rd2, aluRslt, rData;
        input                           brnch, jmp, regDst,
                                                aluSrc, mem2Reg, rst;
```

```
        output  [ 4:0]   adwReg;
        output  [31:0]   pcp, srcB, result;
        wire       [31:0]          pcplus4, pca, pcb, seimm, bta, jta;

        //Logic to create PCP options
        assign pcplus4          =       pc + 32'h4;
        assign seimm            =       {{16{inst[15]}},inst[15:0]};
        assign bta                      =       {seimm[29:0], 2'b0} + pcplus4;
        assign jta                      =       {pcplus4[31:28], {inst[25:0], 2'b00}};
        //Decision for PCP
        MUX2X1                          m0(pcplus4, bta,        brnch,  pca);
        MUX2X1                          m1(pca,         jta,    jmp,    pcb);
        MUX2X1                          m3(pcb,         32'b0,  rst,    pcp);
        //Decision for ALU srcB operand
        MUX2X1                          m4(rd2,         seimm, aluSrc,          srcB);
        //Decision for result source
        MUX2X1                          m5(aluRslt, rData, mem2Reg, result);
        //Decision for register to write data to
        MUX2X1      #(5)    m6(inst[20:16], inst[15:11], regDst, adwReg);

endmodule

module MUX2X1 #(MUXWIDTH = 32)(a, b, sel, y);

        input   [(MUXWIDTH-1):0]    a, b;
        input                                       sel;
        output  [(MUXWIDTH-1):0]    y;

        assign y = (sel) ? b : a;

endmodule

module ctrlogic(op, func, regW, regDst, aluSrc, brnch, memW, mem2Reg, aluCtrl, jmp, zero);

        input           [5:0]   op, func;
        input                                   zero;
        output reg                          regW, regDst, aluSrc,
                                                    brnch, memW, mem2Reg, jmp;
        output          [2:0]           aluCtrl;
        reg                     [1:0]           aluOp;

        always@(*)begin

                case(op)
```

```
                        //R-Type operation
                        6'b000000 : {regW, regDst, aluSrc, brnch, memW, mem2Reg, aluOp, jmp}
                                                = 9'b1_1_0_0_0_0_10_0;
                        //LW operation
                        6'b100011 : {regW, regDst, aluSrc, brnch, memW, mem2Reg, aluOp, jmp}
                                                = 9'b1_0_1_0_0_1_00_0;
                        //SW operation
                        6'b101011 : {regW, regDst, aluSrc, brnch, memW, mem2Reg, aluOp, jmp}
                                                = 9'b0_X_1_0_1_X_00_0;
                        //BEQ operation
                        6'b000100 : {regW, regDst, aluSrc, brnch, memW, mem2Reg, aluOp, jmp}
                                                = {3'b0_X_0, zero, 5'b0_X_01_0};
                        //ADDI operation
                        6'b001000 : {regW, regDst, aluSrc, brnch, memW, mem2Reg, aluOp, jmp}
                                                = 9'b1_0_1_0_0_0_00_0;
                        //J operation
                        6'b000010 : {regW, regDst, aluSrc, brnch, memW, mem2Reg, aluOp, jmp}
                                                = 9'b0_X_X_X_0_X_XX_1;
                        //Undef state
                        default   :       {regW, regDst, aluSrc, brnch, memW, mem2Reg, aluOp,
jmp}
                                                = 9'bX_X_X_X_X_X_XX_X;
                endcase // op

        end

        aluDec  al0(aluOp, func, aluCtrl);

endmodule


module aluDec(aluOp, func, aluCtrl);

        input [1:0] aluOp;
        input [5:0]       func;

        output reg [2:0] aluCtrl;

        always@(*)begin

                casex({aluOp, func})

                        8'b00_XXXXXX : aluCtrl <= 3'b010;
                        8'bX1_XXXXXX : aluCtrl <= 3'b110;
```

```
                8'b1X_100000 : aluCtrl <= 3'b010;
                8'b1X_100010 : aluCtrl <= 3'b110;
                8'b1X_100100 : aluCtrl <= 3'b000;
                8'b1X_100101 : aluCtrl <= 3'b001;
                8'b10_101010 : aluCtrl <= 3'b111;
                default                 : aluCtrl <= 3'bXXX;

        endcase

    end

endmodule

// Include the top-level pin ip file
`include "../hdl/pin_ip.v"

module top( input CLK_50 );

    // Peripheral interconnect wires

        // Use these wires to feed outputs to LED widgets
        wire [9:0]      LEDR;
        // Use these wires as 8-bit active-low 7-segment
        // outputs, where the 8th bit is the decimal point
        wire [7:0]      HEX0, HEX1, HEX2,
                        HEX3, HEX4, HEX5;
        // Use these wires as slide switch inputs
        wire [9:0]      SW;
        // Use these wires as active low, push-button inputs
        wire [1:0]      KEY;
        // These function as programmable register inputs to a
        // design, useful for adjusting inputs using a keyboard
        wire [31:0] param1, param2, param3;

    // User instantiates design below

        // Turn off HEX display dot
        assign {        HEX0[7], HEX1[7], HEX2[7],
                        HEX3[7], HEX4[7], HEX5[7] } = {6{1'b1}};

        // Instantiate MIPS subset core
        mips    i0(     .rst(SW[0]),
                        .clk(CLK_50),
                        .QUERY(param1[9:0]),
```

```
                                        .QSEL(SW[1]),
                                        .seg0(HEX0[6:0]),
                                        .seg1(HEX1[6:0]),
                                        .seg2(HEX2[6:0]),
                                        .seg3(HEX3[6:0]),
                                        .seg4(HEX4[6:0]),
                                        .seg5(HEX5[6:0]));

    // IP to allow simple user design interfacing with developent kit

        pin_ip                  platform_designer_pin_ip(   .MAX10_CLK1_50(CLK_50),

            .LEDR(LEDR),

            .HEX0(HEX0),

            .HEX1(HEX1),

            .HEX2(HEX2),

            .HEX3(HEX3),

            .HEX4(HEX4),

            .HEX5(HEX5),

            .SW(SW),

            .KEY(KEY),

            .param1(param1),

            .param2(param2),

            .param3(param3));

endmodule
```