UNIVERSITY OF LIVERPOOL

ELEC373 ASSIGNMENT 3

MIPS Processor

Author:
Minghong Xu (201601082)

Assessor:
Prof. Jeremy Smith

Abstract

This report presents the results of ELEC373 assignment 3, which was divided into Part A and Part B. Part A focused on familiarise the MISP assembly language. Part B centered on adding three instructions to an existing MIPS CPU design. ASM chart, block diagram, Assembly code, Verilog Code, and test result were presented.

Declaration of academic integrity

I confirm that I have read and understood the University's definitions of plagiarism and collusion from the Code of Practice on Assessment. I confirm that I have neither committed plagiarism in the completion of this work nor have I colluded with any other party in the preparation and production of this work. The work presented here is my own and in my own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web). I understand the consequences of engaging in plagiarism and collusion as described in the Code of Practice on Assessment (Appendix L).

23rd April 2023

Contents

Perhaps a few more sections

List of Figures

1 Part A

Part A is to write a MIPS assembly program for displaying the lowest 8 digits of my student ID on the DE2 board seven segment display [1].

Below is a program in MIPS assembly displays 01601082 on the DE2 board seven segment display.

```
.text
.globl _main
_main:
        # Construct the HEX0_R reg addr in GPR 1
        lui $t1, 0xFFFF  # Upper half addr
        ori $t1, $t1, 0x2010  # Lower half addr

        # Clear the upper 16 bits of GPR 0
        lui $t0, 0x0000

        # Set the lower 16 bits of GPR 0
        ori $t0, $zero, 0x0024  # decimal 2
        sw $t0, 0x0000($t1)  # Store the decimal 0 into the HEX0_R reg

        ori $t0, $zero, 0x0000  # decimal 8
        sw $t0, 0x0004($t1)  # Store the decimal 1 into the HEX1_R reg

        ori $t0, $zero, 0x0040  # decimal 0
        sw $t0, 0x0008($t1)  # Store the decimal 6 into the HEX2_R reg

        ori $t0, $zero, 0x0079  # decimal 1
        sw $t0, 0x000C($t1)  # Store the decimal 0 into the HEX3_R reg

        ori $t0, $zero, 0x0040  # decimal 0
        sw $t0, 0x0010($t1)  # Store the decimal 1 into the HEX4_R reg

        ori $t0, $zero, 0x0002  # decimal 6
        sw $t0, 0x0014($t1)  # Store the decimal 0 into the HEX5_R reg

        ori $t0, $zero, 0x0079  # decimal 1
        sw $t0, 0x0018($t1)  # Store the decimal 8 into the HEX6_R reg

        ori $t0, $zero, 0x0040  # decimal 0
        sw $t0, 0x002C($t1)  # Store the decimal 2 into the HEX7_R reg
```

This instruction is not needed if you used the full of

The code above were work as expect, as shown in Figure 1

Add a block at the end like lab1: j lab1 to stop the programme running wild

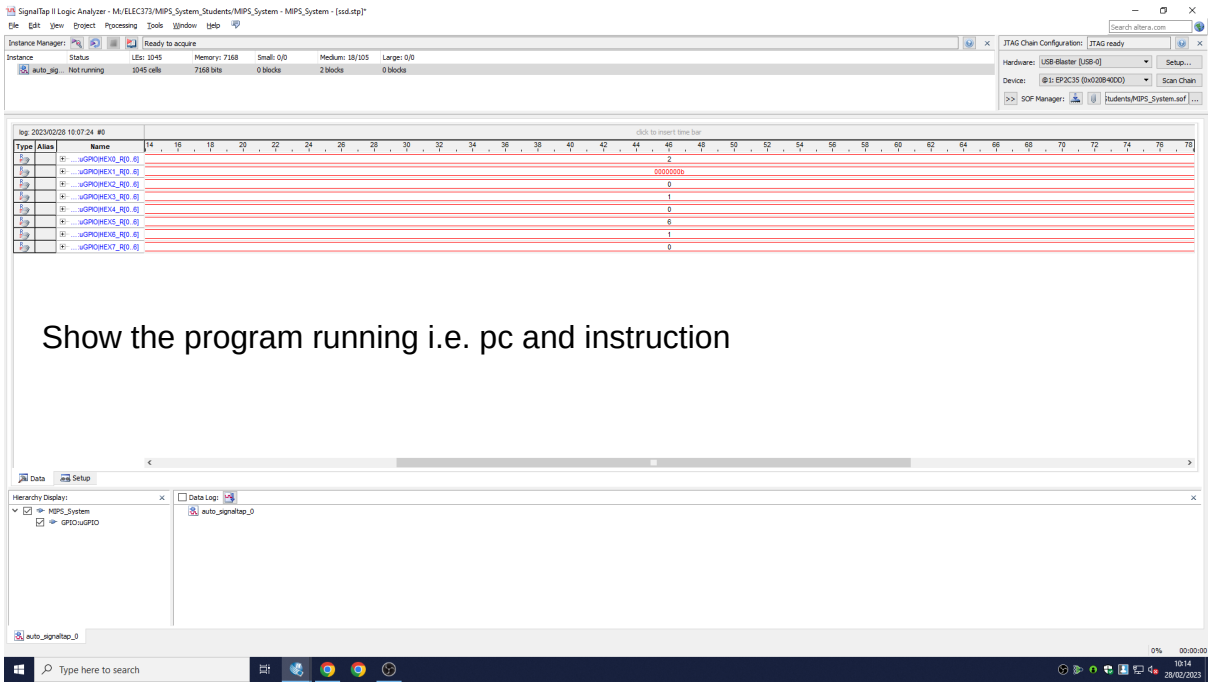


Figure 1: Result of Part A visualised by SignalTap. 0000000b is effectively 8 on the DE2 seven segment display.

1.0.1 Using data segment

The code below is an attempt to simplify by using a *.data* section. However, I didn't figure out how to use the dump of the *.data* with the given MIPS implementation. Figure 2 is the dumped data segment visualised in the *MARS* editor.

```
.data
.word
first_ld: 18      # 2
second_ld: 0      # 8
third_ld: 64      # 0
fourth_ld: 121    # 1
fifth_ld: 64      # 0
sixth_ld: 2       # 6
seventh_ld: 121   # 1
eighth_ld: 64     # 0
HEX7_R: 0xFFFF202C
HEX6_R: 0xFFFF2028
HEX5_R: 0xFFFF2024
HEX4_R: 0xFFFF2020
HEX3_R: 0xFFFF201C
HEX2_R: 0xFFFF2018
HEX1_R: 0xFFFF2014
HEX0_R: 0xFFFF2010

.text
.globl _main
_main:
        lw $t0, first_ld
        sw $t0, HEX7_R
```

```
25
26          lw $t0, second_ld
27          sw $t0, HEX6_R
28
29          lw $t0, third_ld
30          sw $t0, HEX5_R
31
32          lw $t0, fourth_ld
33          sw $t0, HEX4_R
34
35          lw $t0, fifth_ld
36          sw $t0, HEX3_R
37
38          lw $t0, sixth_ld
39          sw $t0, HEX2_R
40
41          lw $t0, seventh_ld
42          sw $t0, HEX1_R
43
44          lw $t0, eighth_ld
45          sw $t0, HEX0_R
```

This seems to be less efficient than your previous code it seems to be

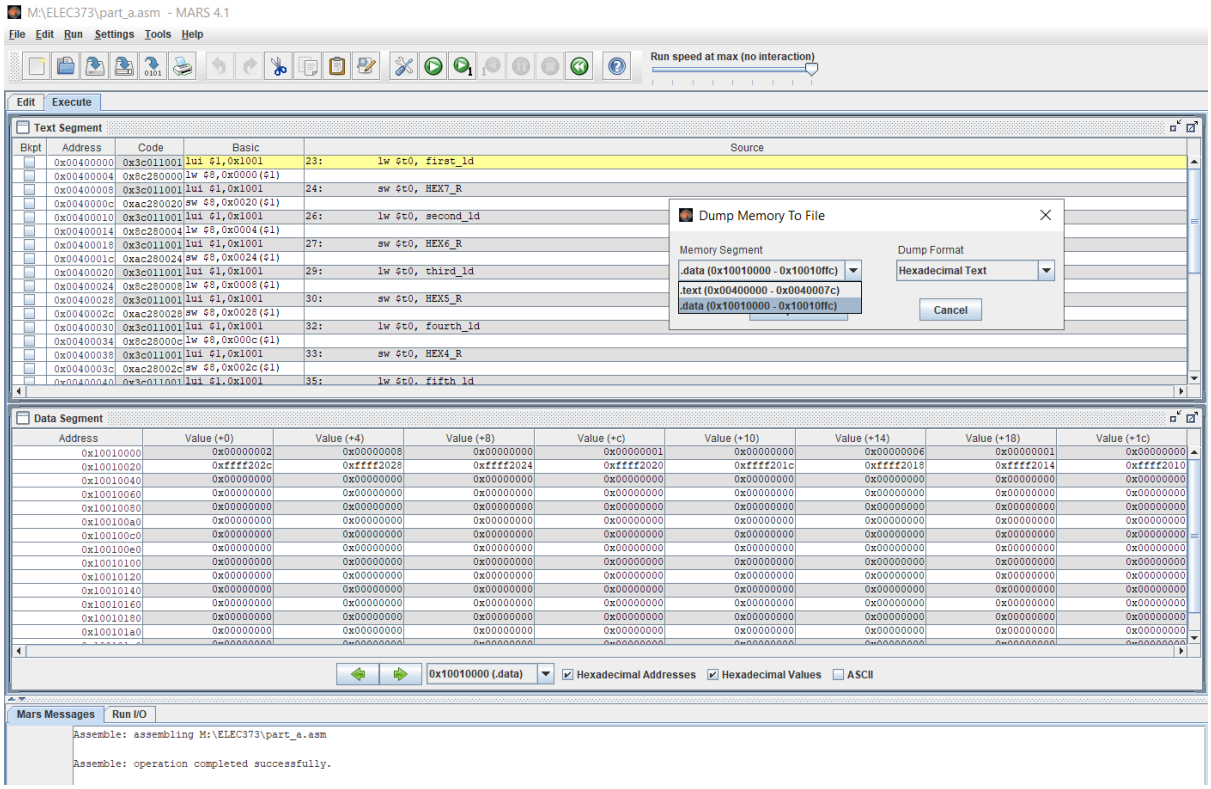


Figure 2: Dump *.data* section to data segment using the *MARS* editor.

Any photo of the number on the board.

2 Part B

Part B is to add three instructions to the existing MIPS CPU design and write tests for them [1].

Figure 3, 4, and 5 are the ASM charts for ALU, four inputs multiplexer, and sign extension, respectively. Figure 6 is the block diagram for the *lb* instruction implementation.

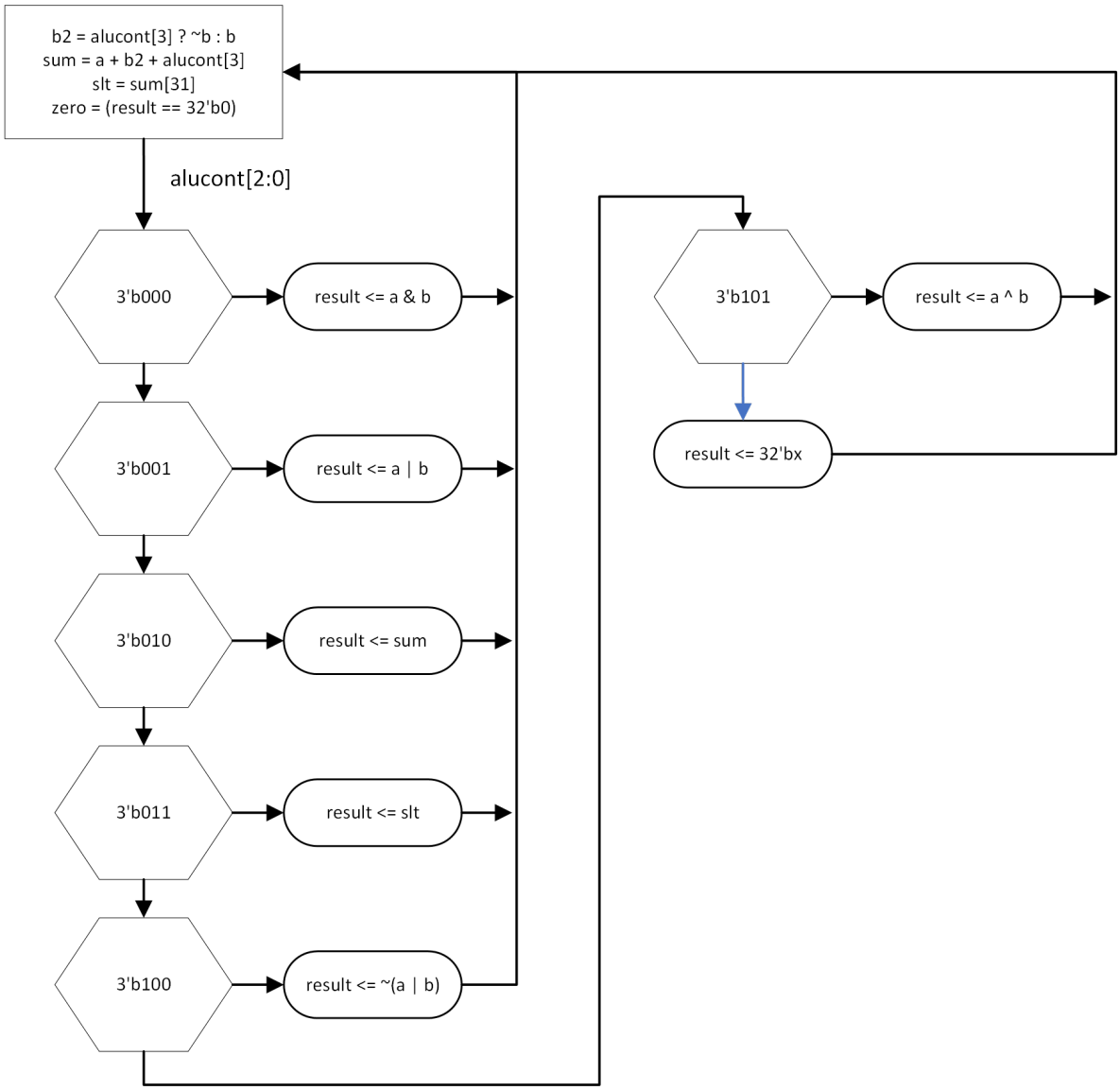


Figure 3: ASM chart for the ALU module.

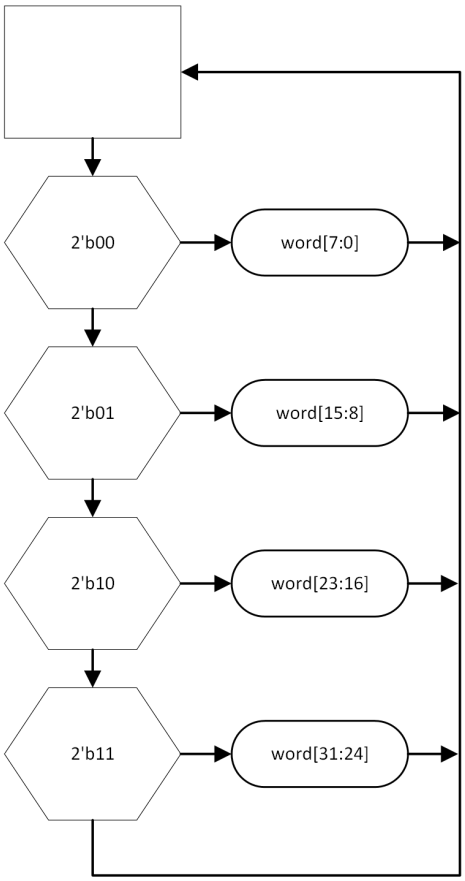


Figure 4: ASM chart for the four inputs multiplexer module.

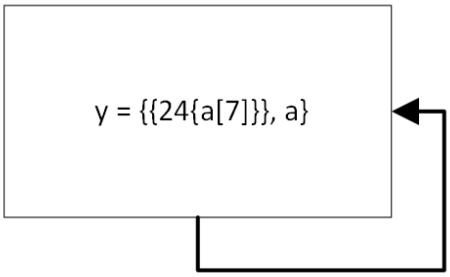


Figure 5: ASM chart for the sign extension module.

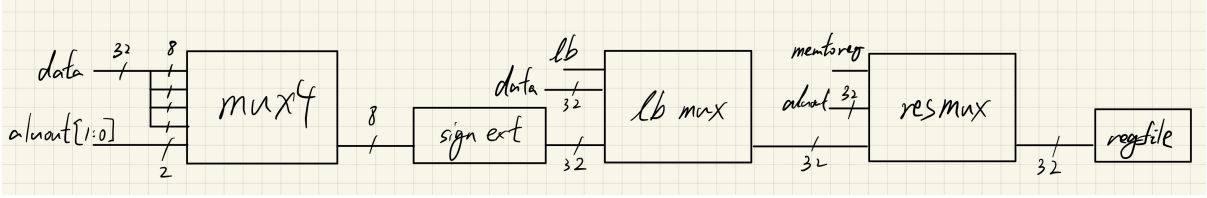


Figure 6: Block diagram of the *lb* implementation design.

To add *nor*, *xori*, and *lb* to the existing design, the following modifications were made.

```
--- MIPS_System_Students\MIPS_CPU\mipsparts.v        2013-02-08 11:49:24.000000000 +0100
+++ MIPS_System_Students\MIPS_CPU\mipsparts.v        2023-04-20 16:22:16.000000000 +0100
@@ -18,28 +18,31 @@
   assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
   assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
 endmodule


 module alu(input      [31:0] a, b,
-           input      [2:0]  alucont,
+           input      [3:0]  alucont,
            output reg [31:0] result,
            output            zero);

   wire [31:0] b2, sum, slt;

-  assign b2 = alucont[2] ? ~b:b;
-  assign sum = a + b2 + alucont[2];
+  assign b2 = alucont[3] ? ~b : b;
+  assign sum = a + b2 + alucont[3];
   assign slt = sum[31];

   always@(*)
-    case(alucont[1:0])
-      2'b00: result <= a & b;
-      2'b01: result <= a | b;
-      2'b10: result <= sum;
-      2'b11: result <= slt;
+    case(alucont[2:0])
+      3'b000: result <= a & b;
+      3'b001: result <= a | b;
+      3'b010: result <= sum;
+      3'b011: result <= slt;
+      3'b100: result <= ~(a | b); // NOR
+      3'b101: result <= a ^ b; // XOR
+      default: result <= 32'bx; // FIXME: I don't know whether this is a sensible default
     endcase

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

 endmodule

@@ -121,6 +124,48 @@
               input              s,
               output [WIDTH-1:0] y);

   assign y = s ? d1 : d0;

 endmodule
+
+
```

```
+
+module mux4 #(
+    parameter WIDTH = 8
+)(
+    input  [WIDTH-1:0] d0, d1, d2, d3,
+    input  [1:0] s,
+    output reg [WIDTH-1:0] y
+);
+
+always @(d0, d1, d2, d3, s)
+    case (s)
+       2'b00: y <= d0;
+       2'b01: y <= d1;
+       2'b10: y <= d2;
+       2'b11: y <= d3;
+    endcase
+
+endmodule
+
+
+
+module sign_ext(
+    input  [7:0] a,
+    output [31:0] y
+);
+
+    assign y = {{24{a[7]}}, a};
+
+endmodule
+
+
+
+module zero_ext(
+    input  [7:0] a,
+    output [31:0] y
+);
+
+    assign y = {24'b0, a};
+
+endmodule
```

Did you need zero_ext?

```
--- MIPS_System_Students\MIPS_CPU\mips.v         2013-02-08 11:48:52.000000000 +0100
+++ MIPS_System_Students\MIPS_CPU\mips.v         2023-04-20 16:23:12.000000000 +0100
@@ -19,14 +19,14 @@
              output [31:0] memaddr,
              output [31:0] memwritedata,
              input  [31:0] memreaddata);

   wire        signext, shiftl16, memtoreg, branch;
   wire        pcsrc, zero;
-  wire        alusrc, regdst, regwrite, jump;
-  wire [2:0]  alucontrol;
+  wire        alusrc, regdst, regwrite, jump, lb;
+  wire [3:0]  alucontrol;

   // Instantiate Controller
   controller c(.op         (instr[31:26]),
```

```
17                  .funct      (instr[5:0]),
18                  .zero       (zero),
19                  .signext    (signext),
20  @@ -35,12 +35,13 @@
21                  .memwrite   (memwrite),
22                  .pcsrc      (pcsrc),
23                  .alusrc     (alusrc),
24                  .regdst     (regdst),
25                  .regwrite   (regwrite),
26                  .jump       (jump),
27  +               .lb         (lb),
28                  .alucontrol (alucontrol));
29
30     // Instantiate Datapath
31     datapath dp( .clk        (clk),
32                  .reset      (reset),
33                  .signext    (signext),
34  @@ -48,12 +49,13 @@
35                  .memtoreg   (memtoreg),
36                  .pcsrc      (pcsrc),
37                  .alusrc     (alusrc),
38                  .regdst     (regdst),
39                  .regwrite   (regwrite),
40                  .jump       (jump),
41  +               .lb         (lb),
42                  .alucontrol (alucontrol),
43                  .zero       (zero),
44                  .pc         (pc),
45                  .instr      (instr),
46                  .aluout     (memaddr),
47                  .writedata  (memwritedata),
48  @@ -67,27 +69,29 @@
49                     output       signext,
50                     output       shiftl16,
51                     output       memtoreg, memwrite,
52                     output       pcsrc, alusrc,
53                     output       regdst, regwrite,
54                     output       jump,
55  -                  output [2:0] alucontrol);
56  +                  output       lb,
57  +                  output [3:0] alucontrol);
58
59  -  wire [1:0] aluop;
60  +  wire [2:0] aluop;
61     wire       branch;
62
63     maindec md( .op    (op),
64                 .signext   (signext),
65                 .shiftl16  (shiftl16),
66                 .memtoreg  (memtoreg),
67                 .memwrite  (memwrite),
68                 .branch    (branch),
69                 .alusrc    (alusrc),
70                 .regdst    (regdst),
71                 .regwrite  (regwrite),
72                 .jump      (jump),
73  +              .lb        (lb),
74                 .aluop     (aluop));
```

```
75
76     aludec  ad( .funct      (funct),
77                 .aluop      (aluop),
78                 .alucontrol (alucontrol));
79
80  @@ -100,77 +104,84 @@
81                  output       signext,
82                  output       shiftl16,
83                  output       memtoreg, memwrite,
84                  output       branch, alusrc,
85                  output       regdst, regwrite,
86                  output       jump,
87  -               output [1:0] aluop);
88  +               output       lb,
89  +               output [2:0] aluop);
90
91  -  reg [10:0] controls;
92  +  reg [12:0] controls;
93
94  -  assign {signext, shiftl16, regwrite, regdst,
95  -          alusrc, branch, memwrite,
96  -          memtoreg, jump, aluop} = controls;
97  +  assign {signext, shiftl16, regwrite, regdst, alusrc, branch, memwrite, memtoreg,
     ↪  jump, lb, aluop} = controls;
98
99     always @(*)
100 -     case(op)
101 -       6'b000000: controls <= 11'b00110000011; // Rtype
102 -       6'b100011: controls <= 11'b10101001000; // LW
103 -       6'b101011: controls <= 11'b10001010000; // SW
104 -       6'b000100: controls <= 11'b10000100001; // BEQ
105 +     case(op)                                    // aluop
106 +       6'b000000: controls <= 13'b0_0_1_1_0_0_0_0_0_111; // Rtype
107 +       6'b100011: controls <= 13'b1_0_1_0_1_0_0_1_0_000; // LW
108 +       6'b100000: controls <= 13'b1_0_1_0_1_0_0_1_0_1_000; // LB  signext regwrite
     ↪  alusrc memtoreg
109 +       6'b101011: controls <= 13'b1_0_0_0_1_0_1_0_0_0_000; // SW
110 +       6'b000100: controls <= 13'b1_0_0_0_0_1_0_0_0_0_001; // BEQ
111         6'b001000,
112 -       6'b001001: controls <= 11'b10101000000; // ADDI, ADDIU: only difference is
     ↪  exception
113 -       6'b001101: controls <= 11'b00101000010; // ORI
114 -       6'b001111: controls <= 11'b01101000000; // LUI
115 -       6'b000010: controls <= 11'b00000000100; // J
116 -       default:   controls <= 11'bxxxxxxxxxxx; // ???
117 +       6'b001001: controls <= 13'b1_0_1_0_1_0_0_0_0_0_000; // ADDI, ADDIU: only
     ↪  difference is exception
118 +       6'b001101: controls <= 13'b0_0_1_0_1_0_0_0_0_0_010; // ORI
119 +       6'b001110: controls <= 13'b0_0_1_0_1_0_0_0_0_0_011; // XORI
120 +       6'b001111: controls <= 13'b0_1_1_0_1_0_0_0_0_0_000; // LUI
121 +       6'b000010: controls <= 13'b0_0_0_0_0_0_0_0_1_0_000; // J
122 +       default:   controls <= 13'bx_x_x_x_x_x_x_x_x_x_xxx; // ???
123       endcase
124
125  endmodule
126
127  module aludec(input      [5:0] funct,
128  -             input      [1:0] aluop,
```

```
129 -               output reg [2:0] alucontrol);
130 +               input      [2:0] aluop,
131 +               output reg [3:0] alucontrol);
132
133     always @(*)
134       case(aluop)
135 -       2'b00: alucontrol <= 3'b010;  // add
136 -       2'b01: alucontrol <= 3'b110;  // sub
137 -       2'b10: alucontrol <= 3'b001;  // or
138 -       default: case(funct)          // RTYPE
139 +       3'b000: alucontrol <= 4'b0_010;  // add
140 +       3'b001: alucontrol <= 4'b1_010;  // sub
141 +       3'b010: alucontrol <= 4'b0_001;  // or
142 +       3'b011: alucontrol <= 4'b0_101;  // xori
143 +       default: case(funct) // RTYPE
144             6'b100000,
145 -           6'b100001: alucontrol <= 3'b010; // ADD, ADDU: only difference is exception
146 +           6'b100001: alucontrol <= 4'b0_010; // ADD, ADDU: only difference is
    ↪  exception
147             6'b100010,
148 -           6'b100011: alucontrol <= 3'b110; // SUB, SUBU: only difference is exception
149 -           6'b100100: alucontrol <= 3'b000; // AND
150 -           6'b100101: alucontrol <= 3'b001; // OR
151 -           6'b101010: alucontrol <= 3'b111; // SLT
152 -           default:   alucontrol <= 3'bxxx; // ???
153 -         endcase
154 +           6'b100011: alucontrol <= 4'b1_010; // SUB, SUBU: only difference is
    ↪  exception
155 +           6'b100100: alucontrol <= 4'b0_000; // AND
156 +           6'b100101: alucontrol <= 4'b0_001; // OR
157 +           6'b101010: alucontrol <= 4'b1_011; // SLT
158 +           6'b100111: alucontrol <= 4'b0_100; // NOR
159 +           default:   alucontrol <= 4'bx_xxx; // ???
160 +       endcase
161       endcase
162  endmodule
163
164  module datapath(input         clk, reset,
165                  input         signext,
166                  input         shiftl16,
167                  input         memtoreg, pcsrc,
168                  input         alusrc, regdst,
169                  input         regwrite, jump,
170 -                input  [2:0]  alucontrol,
171 +                input         lb,
172 +                input  [3:0]  alucontrol,
173                  output        zero,
174                  output [31:0] pc,
175                  input  [31:0] instr,
176                  output [31:0] aluout, writedata,
177                  input  [31:0] readdata);
178
179     wire [4:0]  writereg;
180     wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
181     wire [31:0] signimm, signimmsh, shiftedimm;
182     wire [31:0] srca, srcb;
183     wire [31:0] result;
184     wire        shift;
```

```
185 +  wire [31:0] lbres;
186 +  wire [31:0] seres;
187 +  wire [7:0] byteres;
188
189    // next PC logic
190    flopr #(32) pcreg (.clk   (clk),
191                       .reset (reset),
192                       .d     (pcnext),
193                       .q     (pc));
194 @@ -207,32 +218,65 @@
195                      .rd2     (writedata));
196
197    mux2 #(5)   wrmux(.d0  (instr[20:16]),
198                      .d1  (instr[15:11]),
199                      .s   (regdst),
200                      .y   (writereg));
201
202 +  mux4 bytemux(
203 +      .d0 (readdata[7:0]),
204 +      .d1 (readdata[15:8]),
205 +      .d2 (readdata[23:16]),
206 +      .d3 (readdata[31:24]),
207 +      .s  (aluout[1:0]),
208 +      .y  (byteres)
209 +  );
210 +
211 +  sign_ext  se(
212 +      .a (byteres),
213 +      .y (seres)
214 +  );
215 +
216 +  mux2 #(32)  lbmux(
217 +      .d0 (readdata),
218 +      .d1 (seres),
219 +      .s  (lb),
220 +      .y  (lbres)
221 +  );
222 +
223 -  mux2 #(32)  resmux(.d0 (aluout),
224 -                     .d1 (readdata),
225 -                     .s  (memtoreg),
226 -                     .y  (result));
227 +  mux2 #(32)  resmux(
228 +      .d0 (aluout),
229 +      .d1 (lbres),
230 +      .s  (memtoreg),
231 +      .y  (result)
232 +  );
233
234 -  sign_zero_ext  sze(.a       (instr[15:0]),
235 +  sign_zero_ext  sze(.a       (instr[15:0]),  // filled the left 32 bits with the
    ↪  sign bit
236                      .signext (signext),
237                      .y       (signimm[31:0]));
238
239    shift_left_16 sl16(.a         (signimm[31:0]),
240                       .shiftl16  (shiftl16),
241                       .y         (shiftedimm[31:0]));
```

```
242
243     // ALU logic
244 -   mux2 #(32)  srcbmux(.d0 (writedata),
245 -                       .d1 (shiftedimm[31:0]),
246 -                       .s  (alusrc),
247 -                       .y  (srcb));
248 +   mux2 #(32)  srcbmux(.d0 (writedata),  // data from a reg
249 +                       .d1 (shiftedimm[31:0]), // stands for immediate/offset
250 +                       .s  (alusrc),  // if `alusrc` is 1, select `d1`
251 +                       .y  (srcb));  // d0 or d1
252
253     alu         alu( .a       (srca),
254                      .b       (srcb),
255                      .alucont (alucontrol),
256                      .result  (aluout),
257                      .zero    (zero));
258  endmodule
```

The principle of the *lb* implementation is complex. A typical form of the *lb* instruction is

```
lb rd, imm(addr)
```

ALU outputs *addr* + *imm* as the data address to the data memory for reading a data. On the other hand, the data memory ignores the two least significant value of the input data address:

```
ram2port_inst_data Inst_Data_Mem (
    .address_a   (inst_addr[12:2]),
    .address_b   (data_addr[12:2]),
```

Thus, we can use the last two bits of the immediate value, which is effectively the last two bits of the ALU output provided that the last two bits of the *addr* are both zero (remember *aluout* = *addr* + *imm*), for selecting the byte of the data stored at the address *addr*.

2.1 Testing

Below is the test code for *nor*, *xori*, and *lb*.

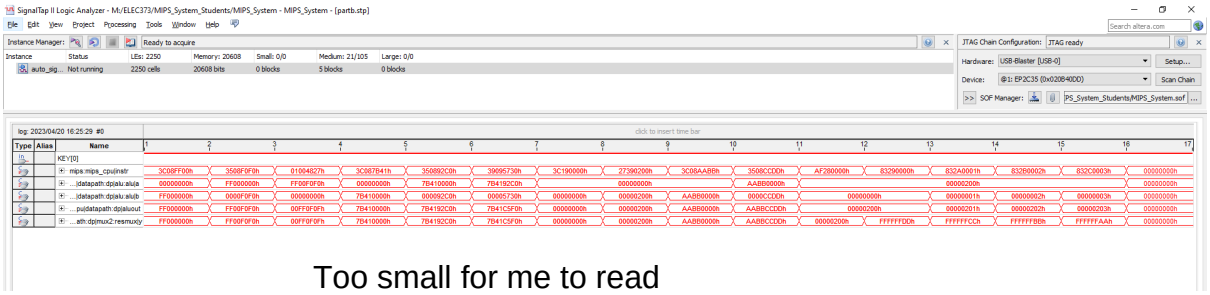
```
1  ### nor
2
3  # Test data 0xFF00 F0F0
4  lui $t0, 0xFF00
5  ori $t0, $t0, 0xF0F0
6
7  nor $t1, $t0, $zero  # Invert all bits; expect 0x00FF 0F0F
8
9
10 You've done 0xFF00F0F0 NOR 0x00000000 which will toggle all the bits but a NOT instruction wo
11 ### xori
```

```
12
13  # Test data 0x7B41 92C0
14  lui $t0, 0x7B41
15  ori $t0, $t0, 0x92C0
16
17  xori $t1, $t0, 0x5730  # expect GPR 1 contains 0x7B41 C5F0
18
19
20
21  ### lb
22
23  # Store test data 0xAABB CCDD to GPR 9
24  lui $t9, 0x0000
25  addiu $t9, $t9, 0x0200
26  lui $t0, 0xAABB
27  ori $t0, $t0, 0xCCDD
28  sw $t0, 0x0000($t9)
29
30  lb $t1, 0x0000($t9)
31  lb $t2, 0x0001($t9)
32  lb $t3, 0x0002($t9)
33  lb $t4, 0x0003($t9)
```

Why go for something that will take more time to check if it is working. Ju

Your test value was 0XAABBCCDD what this doesn't prove is whether th

Figure 7 is the overview of the test results for Part B. Figure 8, 9, and 10 are the test result for *nor*, *xori*, and *lb*, respectively.



Too small for me to read

Figure 7: Test results of Part B visualised by SignalTap.

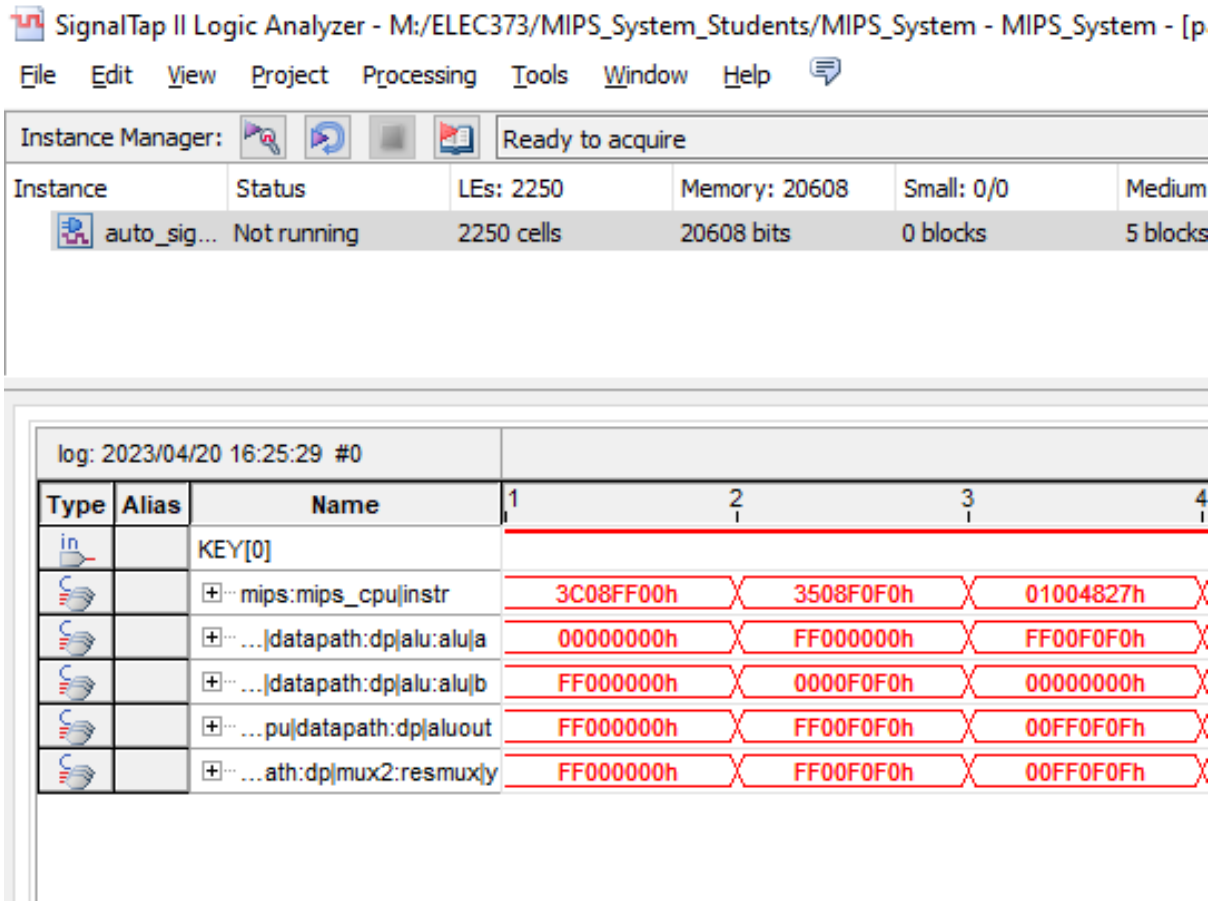


Figure 8: Expected test result of *nor*.

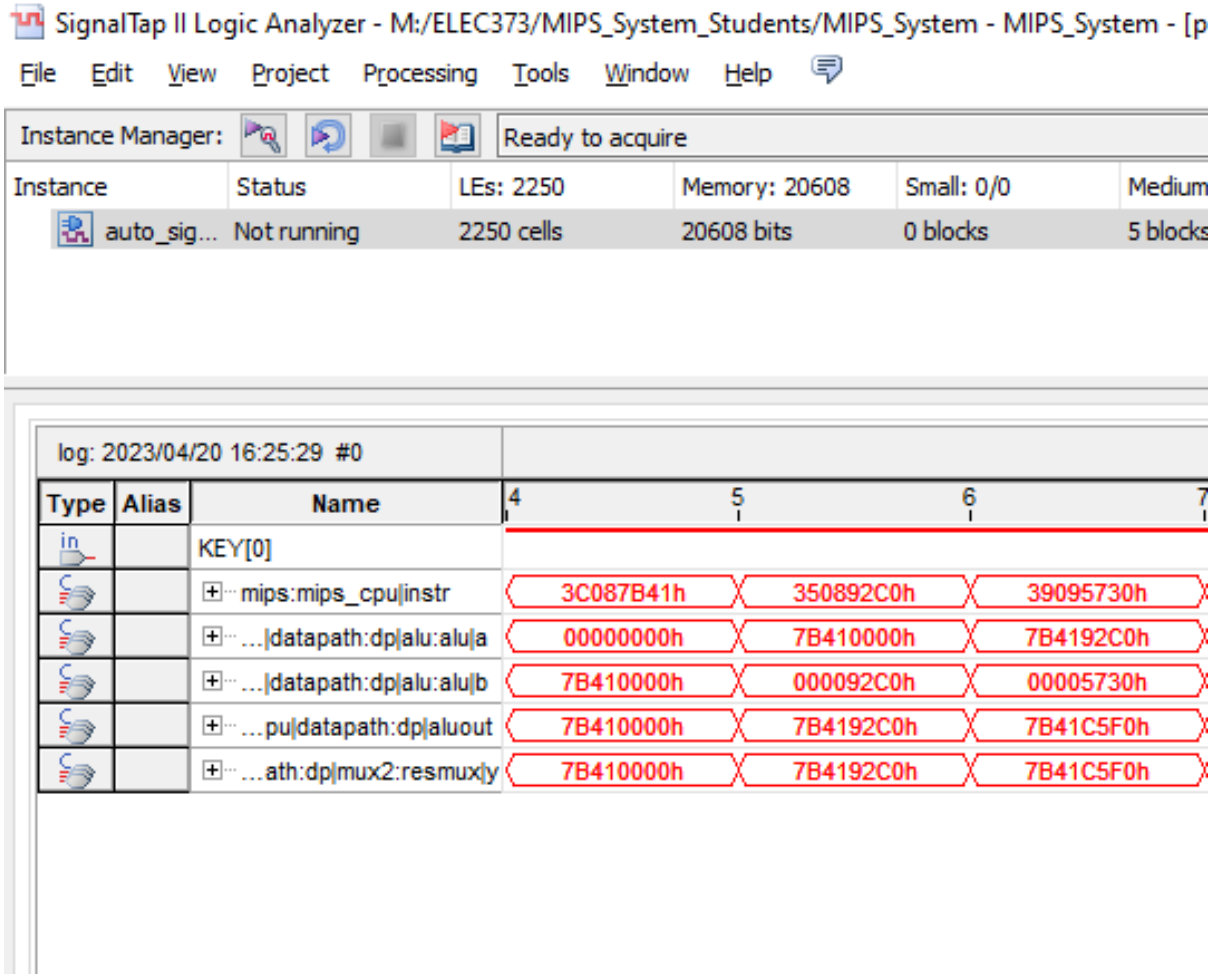


Figure 9: Expected test result of *xori*.

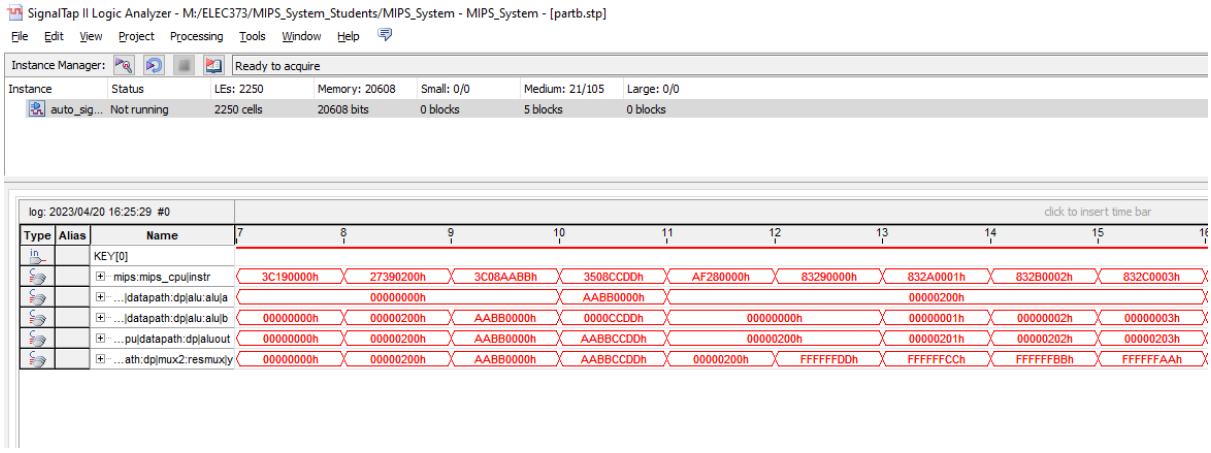


Figure 10: Expected test result of *lb*.

2.1.1 Code coverage

High test coverage could help to ensure modifications don't affect other functionalities. There are numerous coverage criteria such as function coverage and condition coverage. For this task, we want to have high function coverage to ensure the modification doesn't affect the other instruction implementation. The test code below covers all the implemented instructions.

```
# lui ori
lui $s0, 0x0000
ori $s0, $s0, 0x0200

# sw lw

lui $s1, 0xAABB
ori $s1, $s1, 0xCCDD

sw $s1, 0x0000($s0)
lw $t0, 0x0000($s0)

# lb

lb $t0, 0x0000($s0)
lb $t0, 0x0001($s0)
lb $t0, 0x0002($s0)
lb $t0, 0x0003($s0)

# nor

lui $s1, 0xFF00
ori $s1, $s1, 0xF0F0  # Test data 0xFF00 F0F0

nor $t0, $s1, $zero  # Invert all bits; expect 0x00FF 0F0F

# xori

lui $s1, 0x7B41
ori $s1, $s1, 0x92C0  # Test data 0x7B41 92C0

xori $t0, $s1, 0x5730  # Expect 0x7B41 C5F0

# and or

lui $s1, 0xF0F0
ori $s1, $s1, 0xF0F0

lui $s2, 0x0F0F
ori $s2, $s2, 0x0F0F

and $t0, $s1, $s2  # Expect 0x0000 0000
or $t0, $s1, $s2   # Expect 0xFFFF FFFF

# slt

lui $s1, 0x6666
ori $s1, $s1, 0x6666
```

```
49
50  lui $s2, 0x7777
51  ori $s2, $s2, 0x7777
52
53  slt $t0, $s1, $s2  # Expect being set to 1
54
55  # addi, addiu, add, addu
56
57  lui $s1, 0x7FFF
58  ori $s1, $s1, 0xFFFF  # Max positive int in 2's complement
59
60  addi $t0, $s1, 0x0001  # Expect 0x8000 0000 with overflow
61  addiu $t0, $s1, 0x0001  # Expect 0x8000 0000 without overflow
62
63  lui $s2, 0x0000
64  ori $s2, $s2, 0x0001
65
66  add $t0, $s1, $s2  # Expect 0x8000 0000 with overflow
67  addu $t0, $s1, $s2  # Expect 0x8000 0000 without overflow
68
69  # sub, subu
70
71  lui $s1, 0x8000
72  ori $s1, $s1, 0x0000  # Min negative int in 2's complement
73
74  lui $s2, 0x0000
75  ori $s2, $s2, 0x0001
76
77  sub $t0, $s1, $s2  # Expect 0x7FFF FFFF with overflow
78  subu $t0, $s1, $s2  # Expect 0x7FFF FFFF without overflow
79
80  # beq
81
82  lui $s1, 0x6666
83  ori $s1, $s1, 0x6666  # Min negative int in 2's complement
84
85  lui $s2, 0x6666
86  ori $s2, $s2, 0x6666
87
88  beq $s1, $s2, GOTO
89  add $t0, $t0, $t0  # Random instr
90  GOTO:
91
92  # j
93
94  j End
95  add $t0, $t0, $t0  # Random instr
96  End:
```

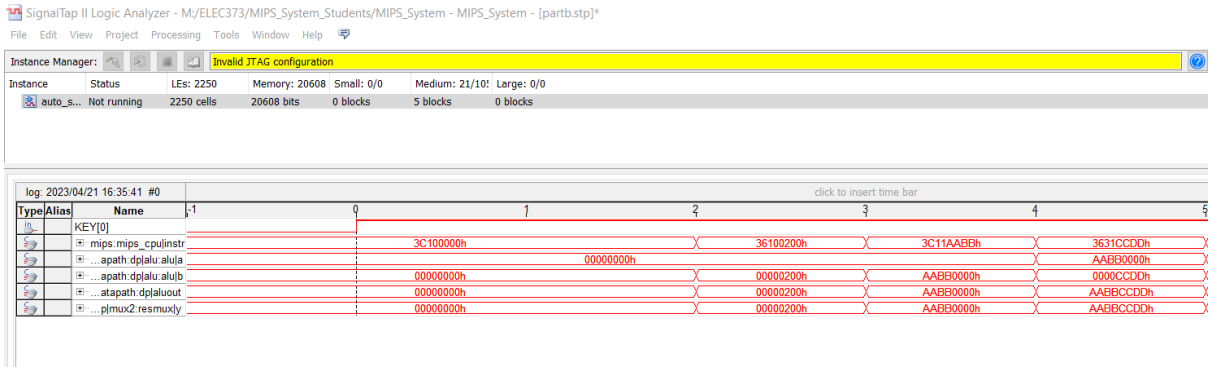


Figure 11: Store 0xAABB CCDD to address 0x000 0200 with *lui*, *ori*, *sw*, and *lw*.

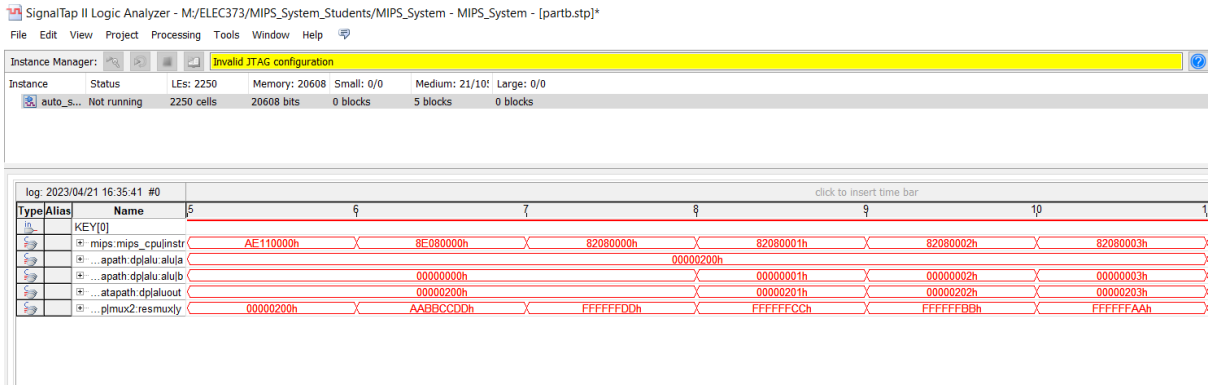


Figure 12: Pick byte DD, CC, BB, and AA from 0xAABB CCDD with sign extension.

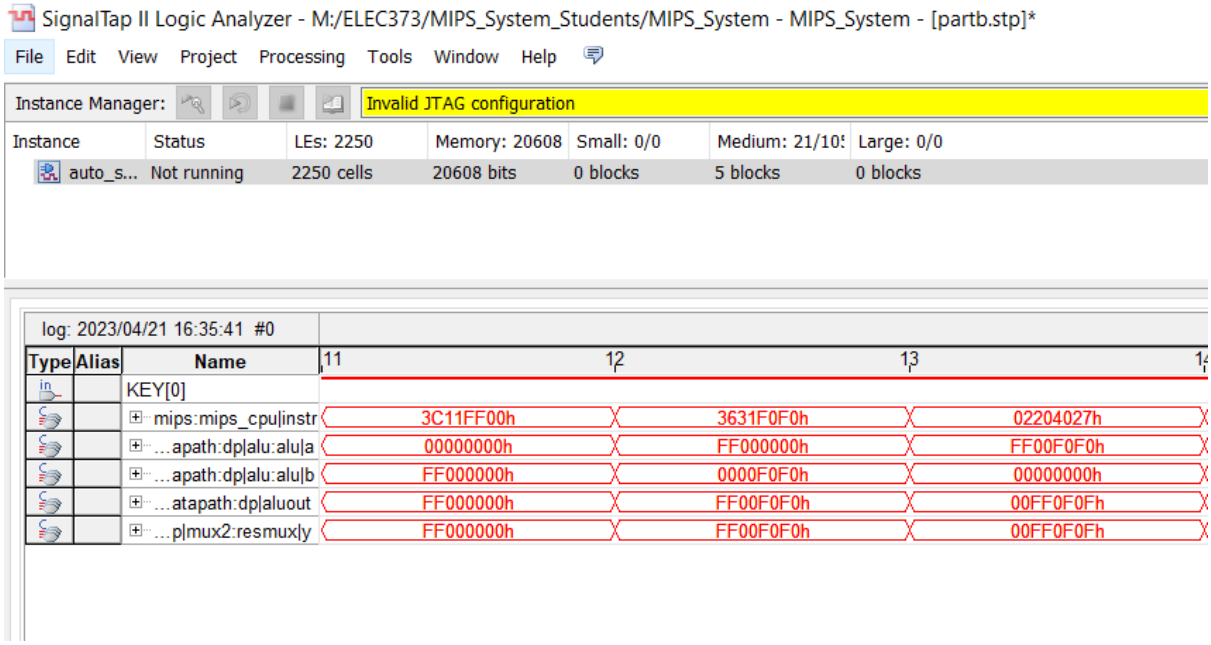


Figure 13: Invert all bit of 0xFF00 F0F0 to 00FF 0F0F by *nor* it with zeros.

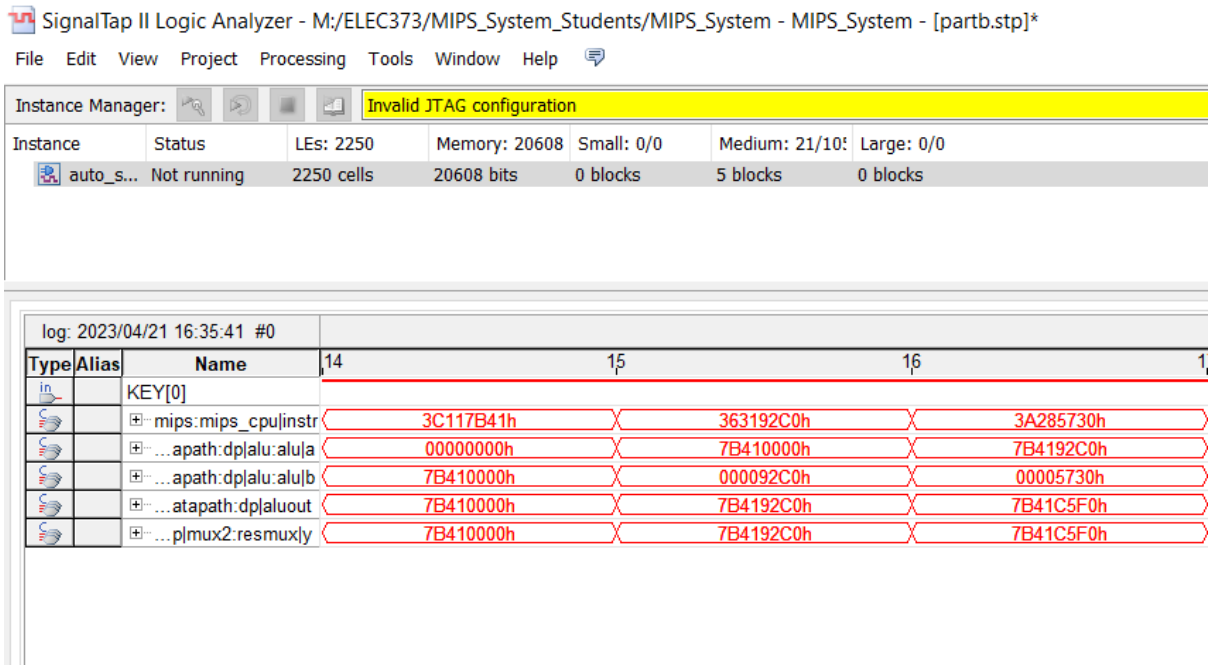


Figure 14: 0x7B41 92C0 *xori* 0x0000 5730 yields 0x7B41 C5F0.

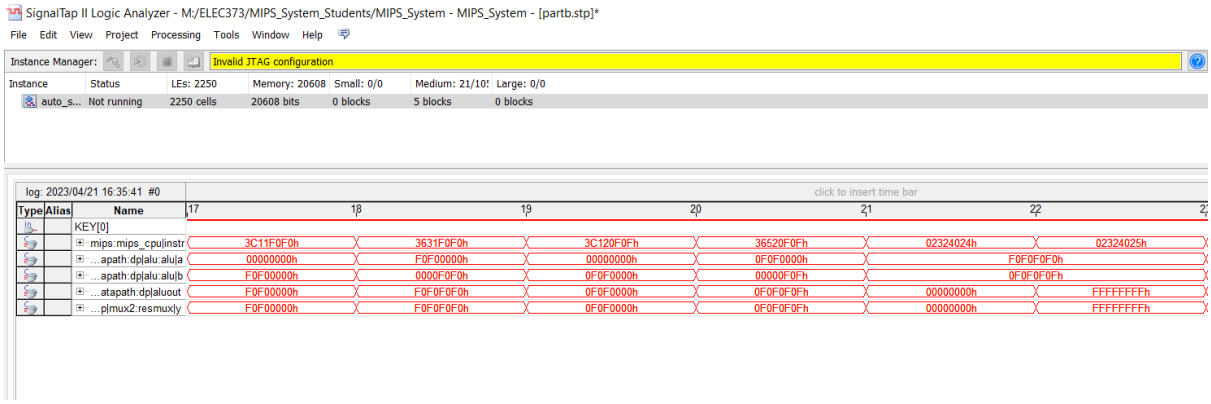


Figure 15: 0xF0F0 F0F0 *and* 0x0F0F 0F0F yields 0x0000 0000, *or* yields 0xFFFF FFFF.

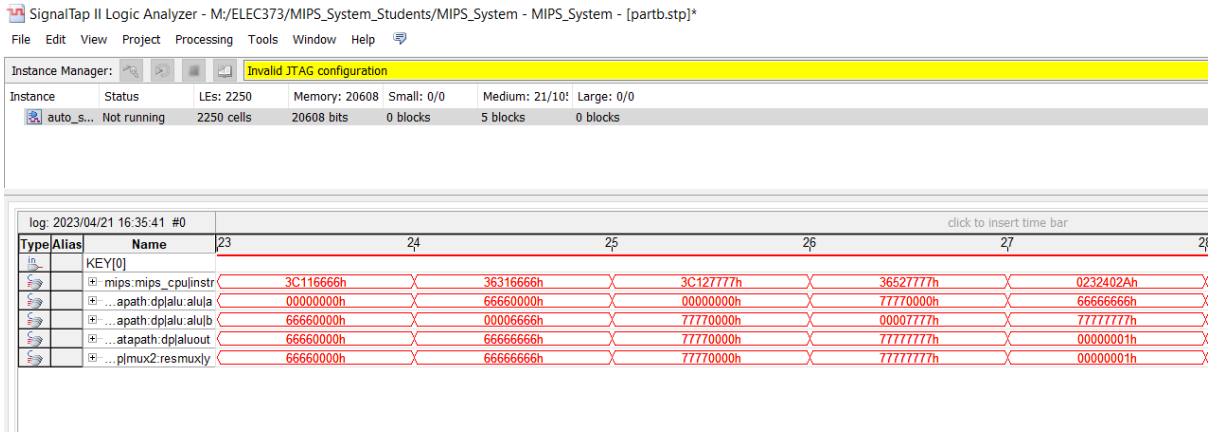


Figure 16: Set 1 because 0x6666 6666 less than 0x7777 7777.

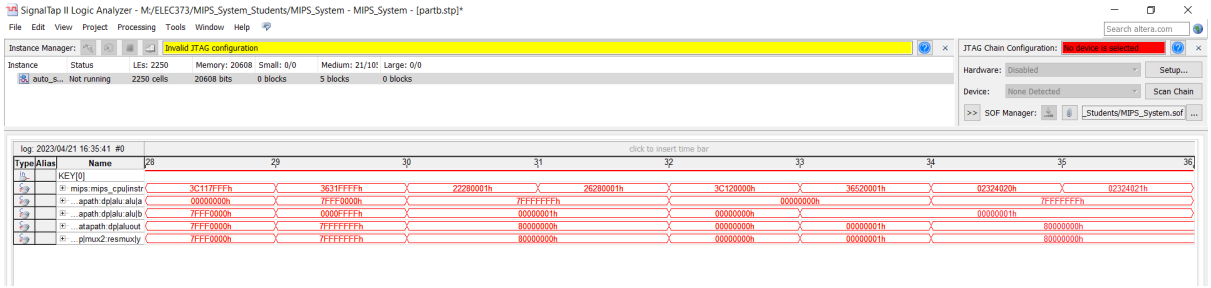


Figure 17: Expected test result of *addi*, *addiu*, *add*, and *addu*. The given MIPS implementation does not implement signalling overflow exception, so the overflow condition cannot be tested.

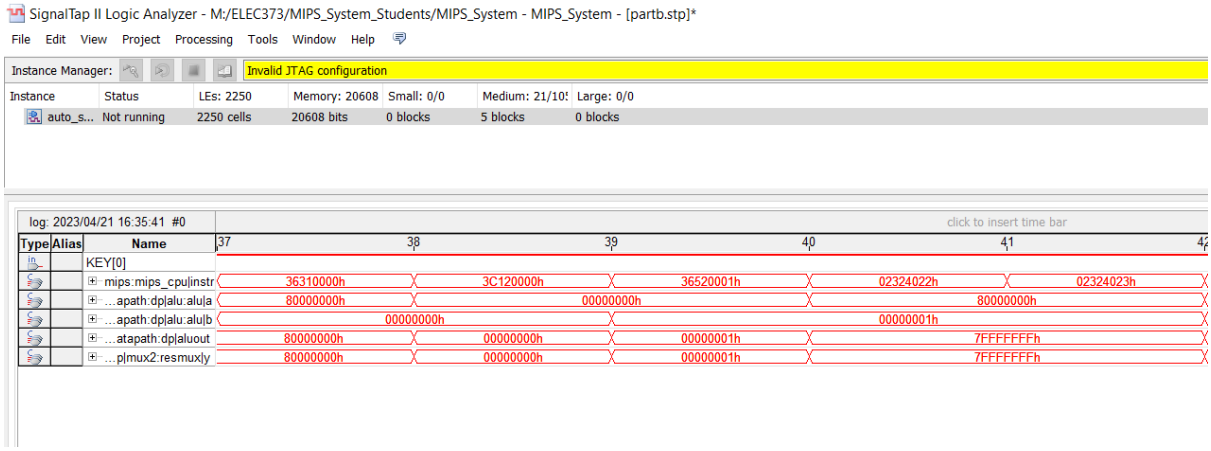


Figure 18: Expected test result of *sub* and *subu*. Again, the overflow condition cannot be tested.

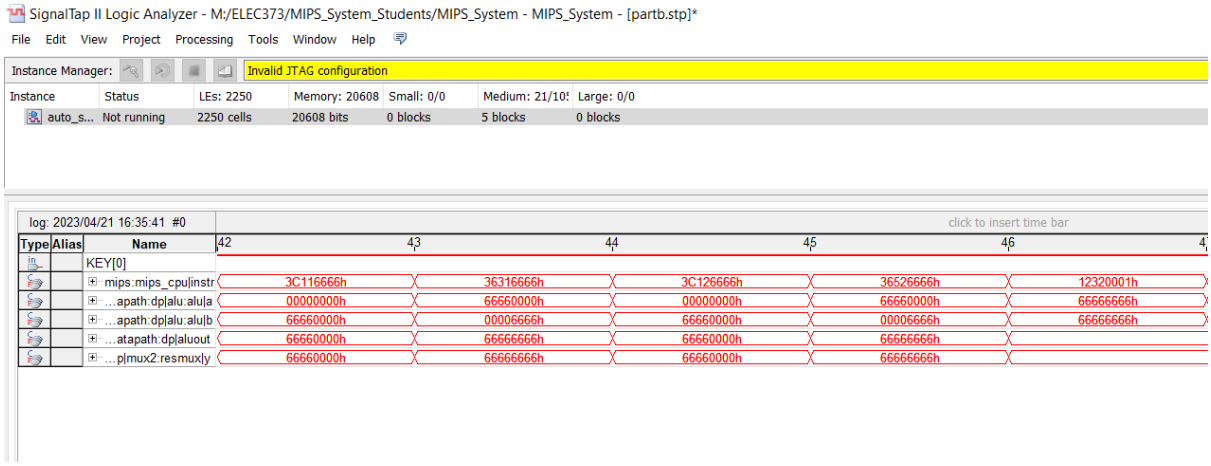


Figure 19: Two values were equal, branching to a label was executed, and skipped an instruction.

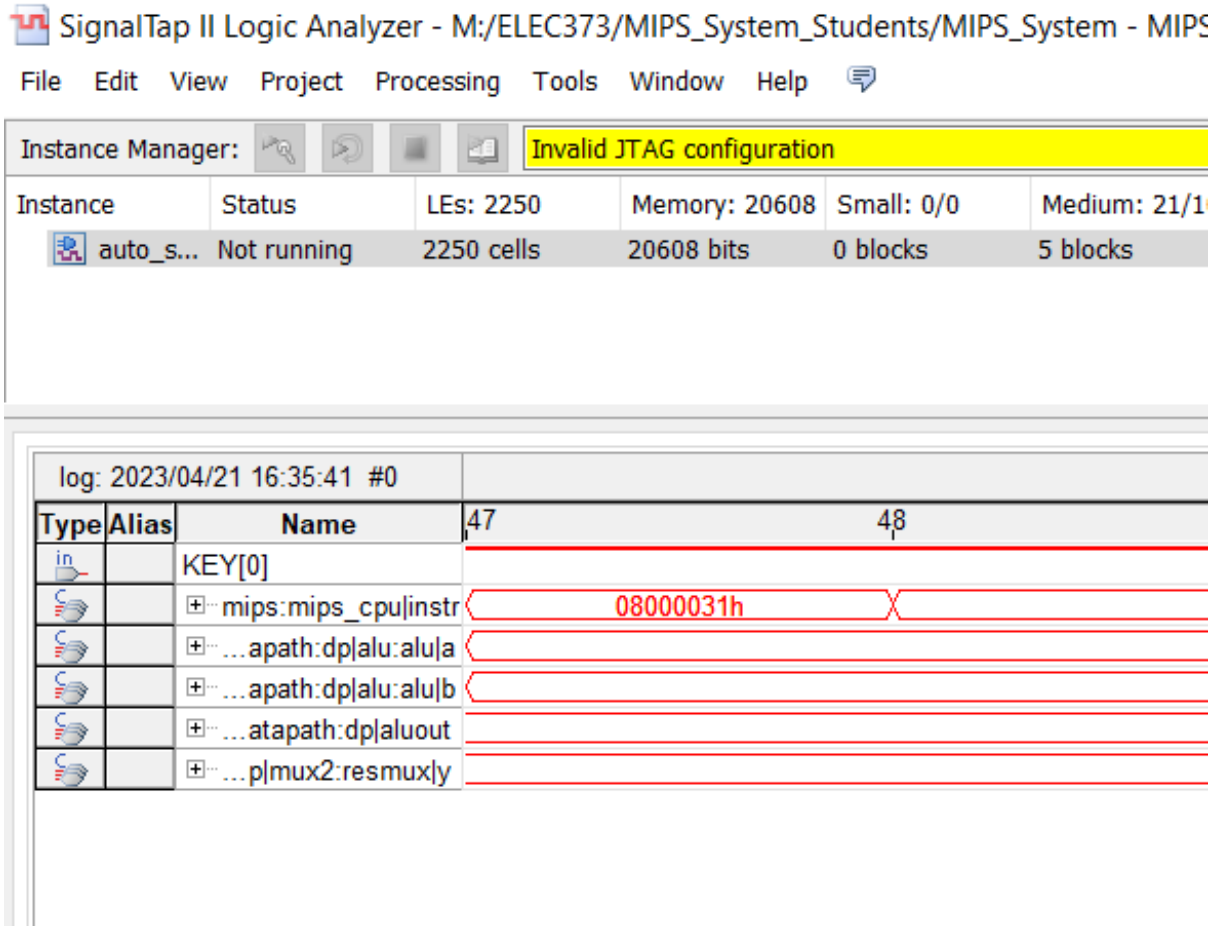


Figure 20: Jumped to a label and skipped an instruction.

References

[1] J. Smith, *ELEC 373 assignment 3 brief*, Mar. 2023.