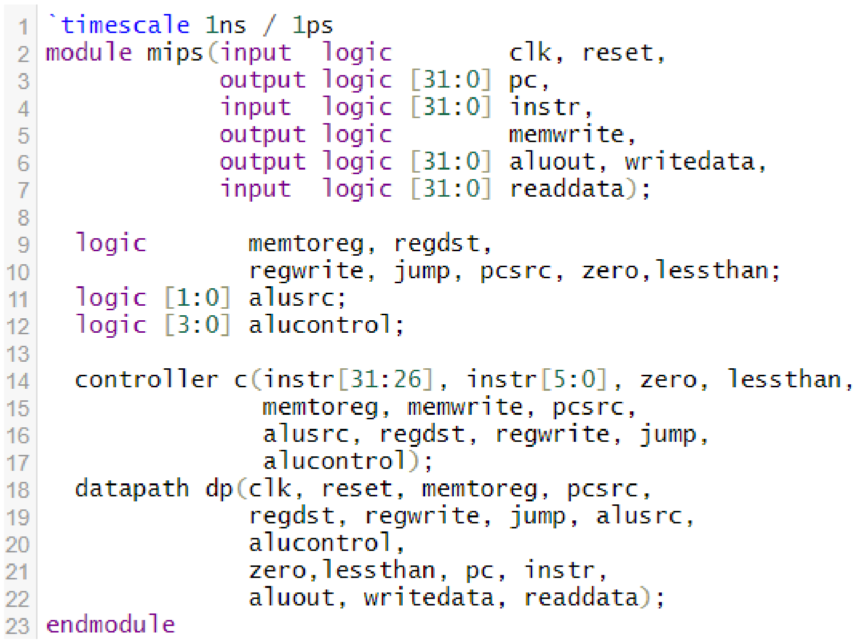
Hans Gabriel H. De Castro
2019-03184

CS 21-12
Project Documentation

## Changes in the files

### mips.sv



```
 1 `timescale 1ns / 1ps
 2 module mips(input  logic        clk, reset,
 3             output logic [31:0] pc,
 4             input  logic [31:0] instr,
 5             output logic        memwrite,
 6             output logic [31:0] aluout, writedata,
 7             input  logic [31:0] readdata);
 8
 9   logic       memtoreg, regdst,
10               regwrite, jump, pcsrc, zero,lessthan;
11   logic [1:0] alusrc;
12   logic [3:0] alucontrol;
13
14   controller c(instr[31:26], instr[5:0], zero, lessthan,
15                memtoreg, memwrite, pcsrc,
16                alusrc, regdst, regwrite, jump,
17                alucontrol);
18   datapath dp(clk, reset, memtoreg, pcsrc,
19               regdst, regwrite, jump, alusrc,
20               alucontrol,
21               zero,lessthan, pc, instr,
22               aluout, writedata, readdata);
23 endmodule
```

On line 10, we see that there is an additional 1-bit logic named less than that will serve as an input to controller and an output in datapath. In line 11 we can see that alusrc was expanded to 2 bits. This is because the mux for SrcB of the ALU is expanded to 4 inputs to accommodate inputs rd2 or writedata from regfile rf, signimm from signextender se, shiftedforlui from shiftleft luishifter, and zeroextended from zeroext zeroextender. These will be explained in the succeeding parts. In line 12, we see that alucontrol was expanded to 4 bits. This is to accommodate the additional instructions for this project.

controller.sv

```
1  `timescale 1ns / 1ps
2  module controller(input  logic [5:0] op, funct,
3                    input  logic       zero, lessthan,
4                    output logic       memtoreg, memwrite,
5                    output logic       pcsrc,
6                    output logic [1:0] alusrc,
7                    output logic       regdst, regwrite,
8                    output logic       jump,
9                    output logic [3:0] alucontrol);
10
11   logic [2:0] aluop;
12   logic       branch, blt;
13
14   maindec md(op, memtoreg, memwrite, branch,blt,
15              alusrc, regdst, regwrite, jump, aluop);
16   aludec  ad(funct, aluop, alucontrol);
17
18   assign pcsrc = (branch & zero)|(blt & lessthan);
19 endmodule
```

On line 3, we see the addition of another input signal, lessthan. We can also see in line 9 that alucontrol was expanded to 4 bits. On line 11, we see that aluop was expanded to 3 bits. On line 12 we see a new signal named blt. We see on line 18 will only become 1 if the branch signal is 1 and the zero signal from the alu is 1 or if the new blt signal is 1 and the new lessthan signal from the alu is 1.

maindec.sv

```
1  `timescale 1ns / 1ps
2  module maindec(input  logic [5:0] op,
3                 output logic       memtoreg, memwrite,
4                 output logic       branch, blt,
5                 output logic [1:0] alusrc,
6                 output logic       regdst, regwrite,
7                 output logic       jump,
8                 output logic [2:0] aluop);
9
10   logic [11:0] controls;
11
12   assign {regwrite, regdst, alusrc, branch, blt, memwrite,
13           memtoreg, jump, aluop} = controls;
14
15   always_comb
16     case(op)
17       6'b000000: controls <= 12'b110000000010; // RTYPE
18       6'b100011: controls <= 12'b100100010000; // LW
19       6'b101011: controls <= 12'b0x01001x0000; // SW
20       6'b000100: controls <= 12'b0x00100x0001; // BEQ
21       6'b001000: controls <= 12'b100100000000; // ADDI
22       6'b000010: controls <= 12'b0xxxx00x1xxx; // J
23       6'b001111: controls <= 12'b101000000100; // lui
24       6'b010001: controls <= 12'b101100000100; // li
25       6'b011111: controls <= 12'b0x00010x0011; // blt
26       default:   controls <= 12'bxxxxxxxxxxxx; // illegal op
27     endcase
28 endmodule
```

We see on line 4 that we have a new output signal called ‘blt’. We can see in line 5 that alusrc was expanded to 2 bits. On line 8, we see that aluop was expanded to 3 bits.

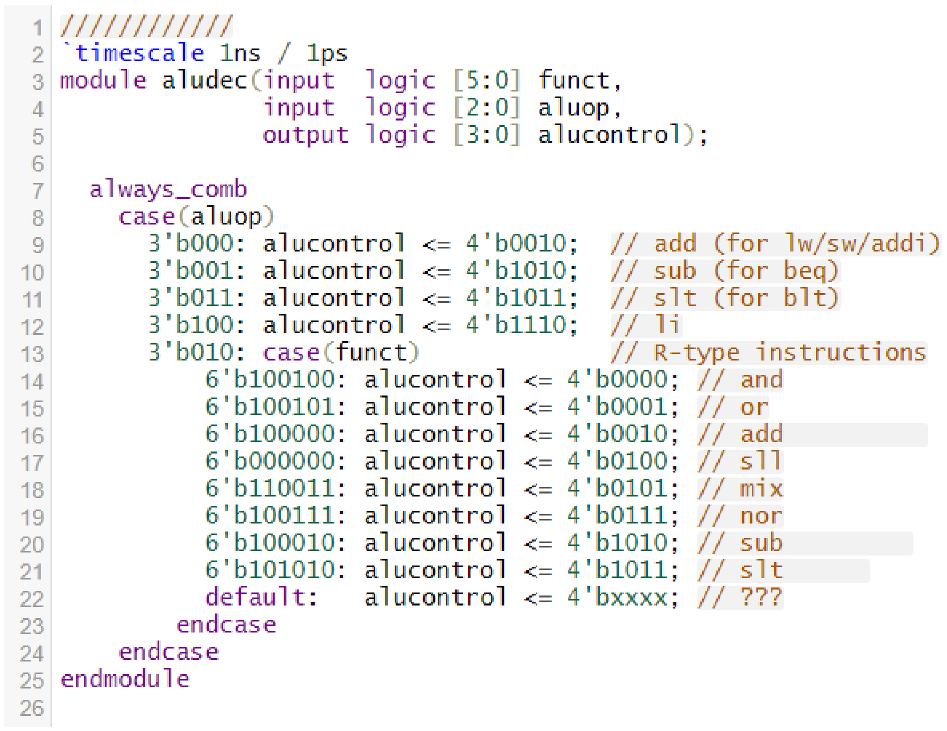
Because of the expansion of alusrc and aluop, and the addition of blt, controls is now 12 bits wide as seen on line 10 and 17-26.

Main decoder truth table

| | regwrite | regdst | alusrc | branch | blt | memwrite | memtoreg | jump | aluop |
|---|---|---|---|---|---|---|---|---|---|
| R-type | 1 | 1 | 00 | 0 | 0 | 0 | 0 | 0 | 010 |
| LW | 1 | 0 | 01 | 0 | 0 | 0 | 1 | 0 | 000 |
| SW | 0 | X | 01 | 0 | 0 | 1 | X | 0 | 000 |
| BEQ | 0 | X | 00 | 1 | 0 | 0 | X | 0 | 001 |
| ADDI | 1 | 0 | 01 | 0 | 0 | 0 | 0 | 0 | 000 |
| J | 0 | X | XX | X | 0 | 0 | X | 1 | XXX |
| LUI | 1 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 100 |
| LI | 1 | 0 | 11 | 0 | 0 | 0 | 0 | 0 | 100 |
| BLT | 0 | X | 00 | 0 | 1 | 0 | X | 0 | 011 |

The new controls seen in maindec.sv are summarized in the table above. The values above are better explained in the succeeding files.

aludec.sv



```
1  ////////////
2  `timescale 1ns / 1ps
3  module aludec(input  logic [5:0] funct,
4                input  logic [2:0] aluop,
5                output logic [3:0] alucontrol);
6
7    always_comb
8      case(aluop)
9        3'b000: alucontrol <= 4'b0010;  // add (for lw/sw/addi)
10       3'b001: alucontrol <= 4'b1010;  // sub (for beq)
11       3'b011: alucontrol <= 4'b1011;  // slt (for blt)
12       3'b100: alucontrol <= 4'b1110;  // li
13       3'b010: case(funct)             // R-type instructions
14           6'b100100: alucontrol <= 4'b0000; // and
15           6'b100101: alucontrol <= 4'b0001; // or
16           6'b100000: alucontrol <= 4'b0010; // add
17           6'b000000: alucontrol <= 4'b0100; // sll
18           6'b110011: alucontrol <= 4'b0101; // mix
19           6'b100111: alucontrol <= 4'b0111; // nor
20           6'b100010: alucontrol <= 4'b1010; // sub
21           6'b101010: alucontrol <= 4'b1011; // slt
22           default:   alucontrol <= 4'bxxxx; // ???
23         endcase
24     endcase
25 endmodule
26
```

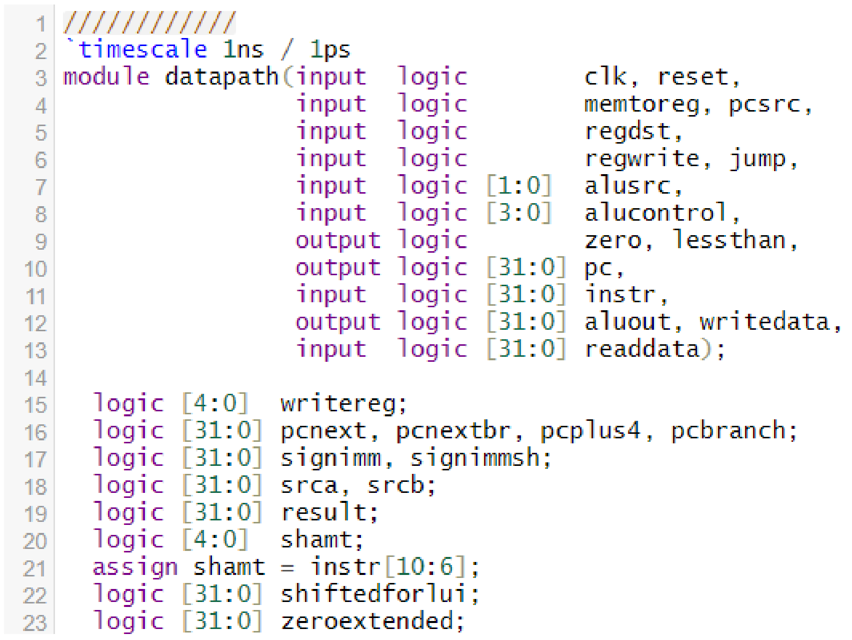
We see in line 4 that aluop was expanded to 3 bits and in line 5 we also see that alucontrol was expanded to 5 bits. The changes are summarized by the truth table below.

Truth Table for aludec.sv

| aluop | funct | alucontrol | instruction |
|---|---|---|---|
| 000 | XXXXXX | 0010 | lw/sw/addi |
| 001 | XXXXXX | 1010 | beq |
| 011 | XXXXXX | 1011 | blt |
| 100 | XXXXXX | 1110 | li |
| 010 | 100100 | 0000 | and |
| 010 | 100101 | 0001 | or |
| 010 | 100000 | 0010 | add |
| 010 | 000000 | 0100 | sll |
| 010 | 110011 | 0101 | mix |
| 010 | 100111 | 0111 | nor |
| 010 | 100010 | 1010 | sub |
| 010 | 101010 | 1011 | slt |

We can see above that the alucontrol[3] for sub and slt are 1. This is to make sure that the design of the alu is still similar to the one originally made in Lab Exercise 12. Notice that the alucontrol for blt is 1011. This matches the one for slt because in the alu, we are going to perform slt on the two operands in blt, and we will branch if and only if the result of the slt is 0b1.

datapath.sv



```
////////////
`timescale 1ns / 1ps
module datapath(input  logic        clk, reset,
                input  logic        memtoreg, pcsrc,
                input  logic        regdst,
                input  logic        regwrite, jump,
                input  logic [1:0]  alusrc,
                input  logic [3:0]  alucontrol,
                output logic        zero, lessthan,
                output logic [31:0] pc,
                input  logic [31:0] instr,
                output logic [31:0] aluout, writedata,
                input  logic [31:0] readdata);

  logic [4:0]  writereg;
  logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  logic [31:0] signimm, signimmsh;
  logic [31:0] srca, srcb;
  logic [31:0] result;
  logic [4:0]  shamt;
  assign shamt = instr[10:6];
  logic [31:0] shiftedforlui;
  logic [31:0] zeroextended;
```

```
25    flopr #(32) pcreg(clk, reset, pcnext, pc);
26    adder #(32) pcadd1(pc, 32'b100, 'b0, pcplus4); //So we adjust this to use the
   more complex adder; wmt-modification
27    sl2         immsh(signimm, signimmsh);
28    adder #(32) pcadd2(pcplus4, signimmsh, 'b0, pcbranch); //See comment above
29    mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
30    mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
31                      instr[25:0], 2'b00}, jump, pcnext);
32
33    // register file logic
34    regfile     rf(clk, regwrite, instr[25:21], instr[20:16],
35                   writereg, result, srca, writedata);
36    mux2 #(5)   wrmux(instr[20:16], instr[15:11],
37                      regdst, writereg);
38    mux2 #(32)  resmux(aluout, readdata, memtoreg, result);
39    signext     se(instr[15:0], signimm);
40
41    // ALU logic
42    zeroext     zeroxtender(instr[15:0],zeroextended);
43    shiftleft   luishifter(instr[15:0],shiftedforlui);
44    mux4 #(32)  srcbmux(writedata, signimm, shiftedforlui,zeroextended, alusrc,
   srcb);
45    alu         alu(srca, srcb, alucontrol,shamt, aluout, zero, lessthan);
46  endmodule
```

We can see in line 7 that alusrc was expanded to 2 bits and on line 8, we also see alucontrol expanded to 4 bits. On line 9, we see that there is an additional 1-bit output named less than. On line 20, we see a shamt, a new 5-bit value. We can see on line 21, that we assign to shamt bits 10:6 of instr. On line 22, we see the addition of a 32-bit value labeled shiftedforlui. On line 23, we also see another 32-bit logic variable named zeroextended.On line 42, we see a new module named zeroxtender of the type zeroext. It takes in 1 input, the last 16 bits of instr and outputs a 32-bit value named zeroextended. On line  43 we see another module named luishifter of the type shiftleft. It takes in 1 input, the last 16 bits of instr and outputs a 32 bit value named shiftedforlui. On line 44 we see that mux2 was replaced with mux4. Scrbmux now takes in 5 inputs from originally 3 and still has one output srcb. On line 45, we see that alu has an additional input called shamt and an additional output called lessthan.

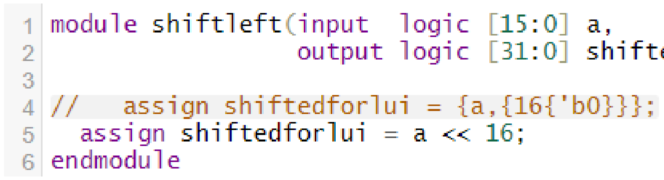
zeroext.sv

```
module zeroext(input  logic [15:0] a,
               output logic [31:0] y);

  assign y = {16'h0000, a};
endmodule
```

What this new module does is that it attaches 16 0’s on the left of the 16-bit input a and outputs it as y. This module was particularly created for the pseudo-instruction li. That is why the alusrc for li = 0b11. Looking back at srcbmux in datapath.sv, we can see that it is 4th in the inputs (00 – 11).

shiftleft.sv

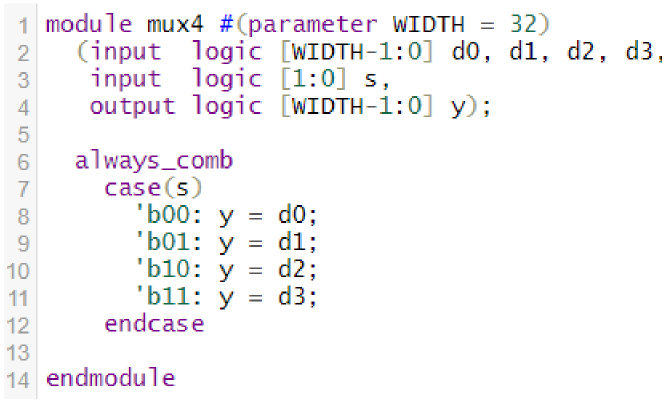


```
module shiftleft(input  logic [15:0] a,
                 output logic [31:0] shiftedforlui);

//   assign shiftedforlui = {a,{16{'b0}}};
  assign shiftedforlui = a << 16;
endmodule
```

What this new module does is that it shifts the 16-bit input a to the left and outputs it as y. This module was specifically created for the lui instruction.

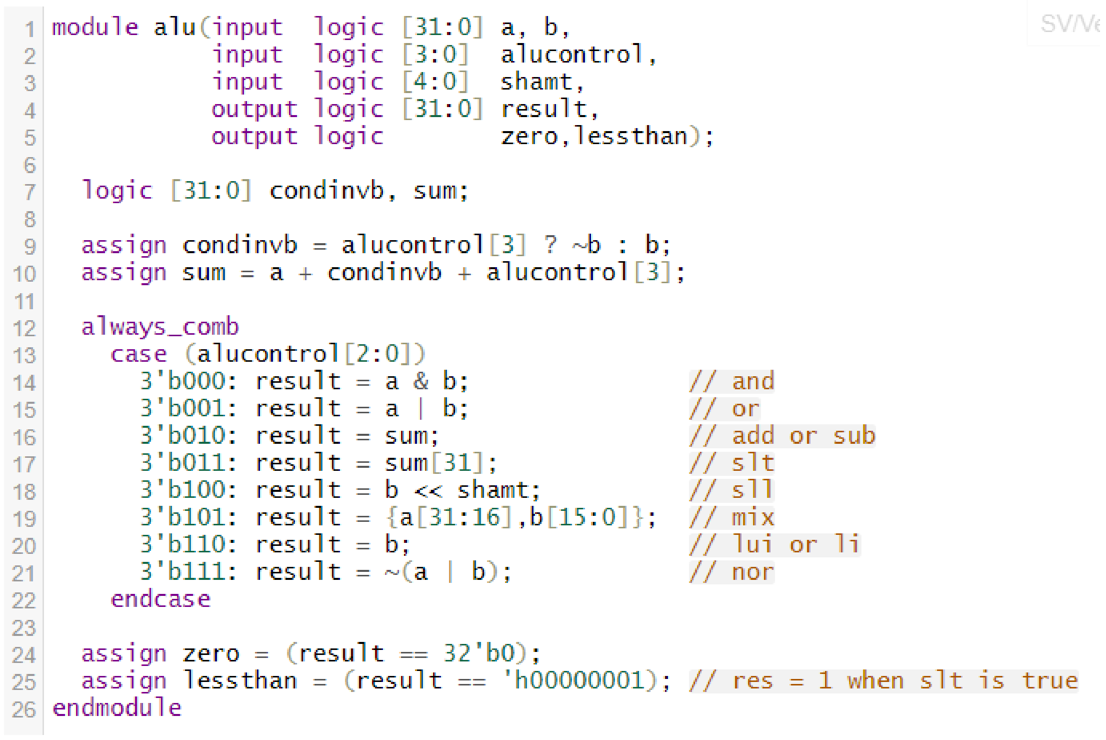
mux4.sv



```
1  module mux4 #(parameter WIDTH = 32)
2    (input  logic [WIDTH-1:0] d0, d1, d2, d3,
3     input  logic [1:0] s,
4     output logic [WIDTH-1:0] y);
5
6    always_comb
7      case(s)
8        'b00: y = d0;
9        'b01: y = d1;
10       'b10: y = d2;
11       'b11: y = d3;
12     endcase
13
14 endmodule
```

What this new module does is just replacing mux2.sv because this is now a 4-1 multiplexer. We can see on line 2 that there are now 4 inputs. And we can see on line 3 that the select input is now 2 bits wide. If we look back at srcbmux in datapath.sv we can see that if s = 0b00, then the output of the mux will be writedata or rd2 from the register file. If s = 0b01, then the output of the mux will be signimm, the output of the se the signextender. If s = 0b10, then the output of the mux will be shiftedforlui, the output of the new module shiftleft for the lui instruction. That is why in maindec.sv, the alusrc for lui is 0b10. If s = 0b11, then the ouput of the mux will be zeroextended, the outpuf of the new module zeroext. That is why in maindec.sv, the alusrc for li is 0b11.

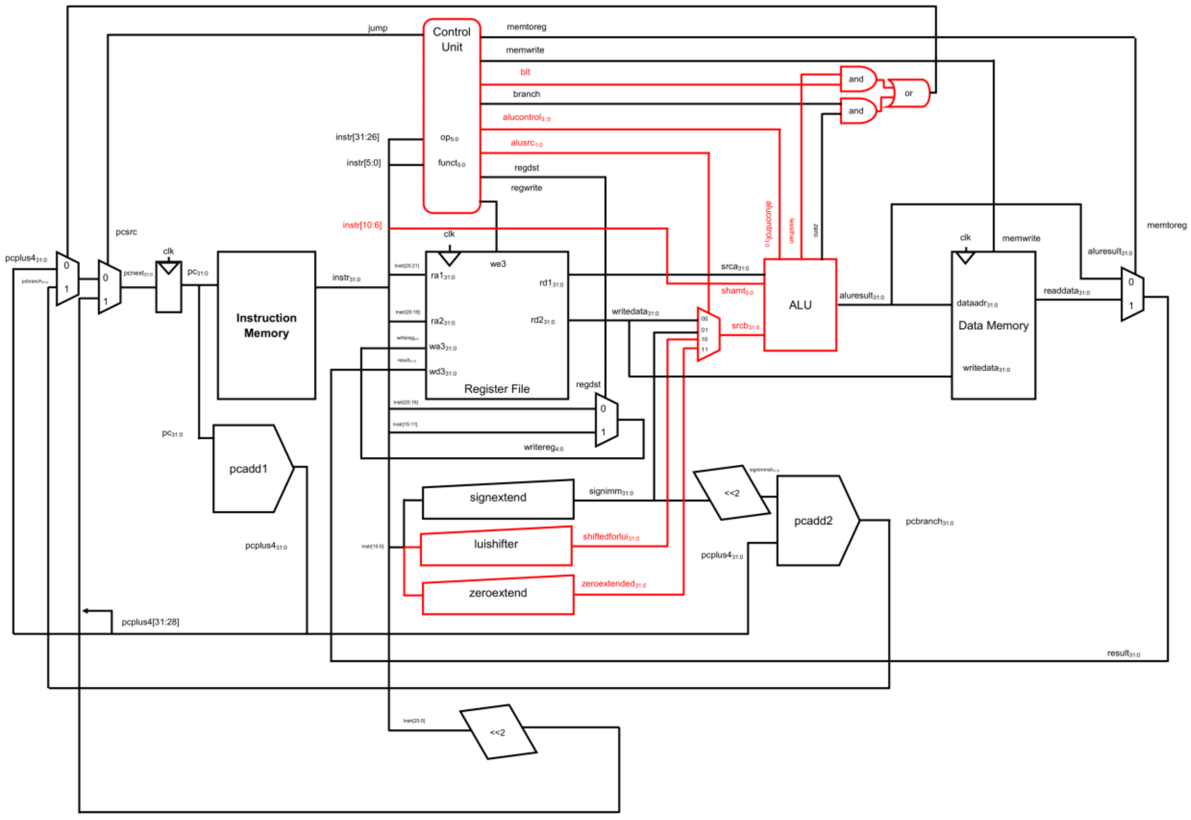
alu.sv



```
 1 module alu(input  logic [31:0] a, b,
 2            input  logic [3:0]  alucontrol,
 3            input  logic [4:0]  shamt,
 4            output logic [31:0] result,
 5            output logic        zero,lessthan);
 6
 7   logic [31:0] condinvb, sum;
 8
 9   assign condinvb = alucontrol[3] ? ~b : b;
10   assign sum = a + condinvb + alucontrol[3];
11
12   always_comb
13     case (alucontrol[2:0])
14       3'b000: result = a & b;              // and
15       3'b001: result = a | b;              // or
16       3'b010: result = sum;                // add or sub
17       3'b011: result = sum[31];            // slt
18       3'b100: result = b << shamt;         // sll
19       3'b101: result = {a[31:16],b[15:0]}; // mix
20       3'b110: result = b;                  // lui or li
21       3'b111: result = ~(a | b);           // nor
22     endcase
23
24   assign zero = (result == 32'b0);
25   assign lessthan = (result == 'h00000001); // res = 1 when slt is true
26 endmodule
```

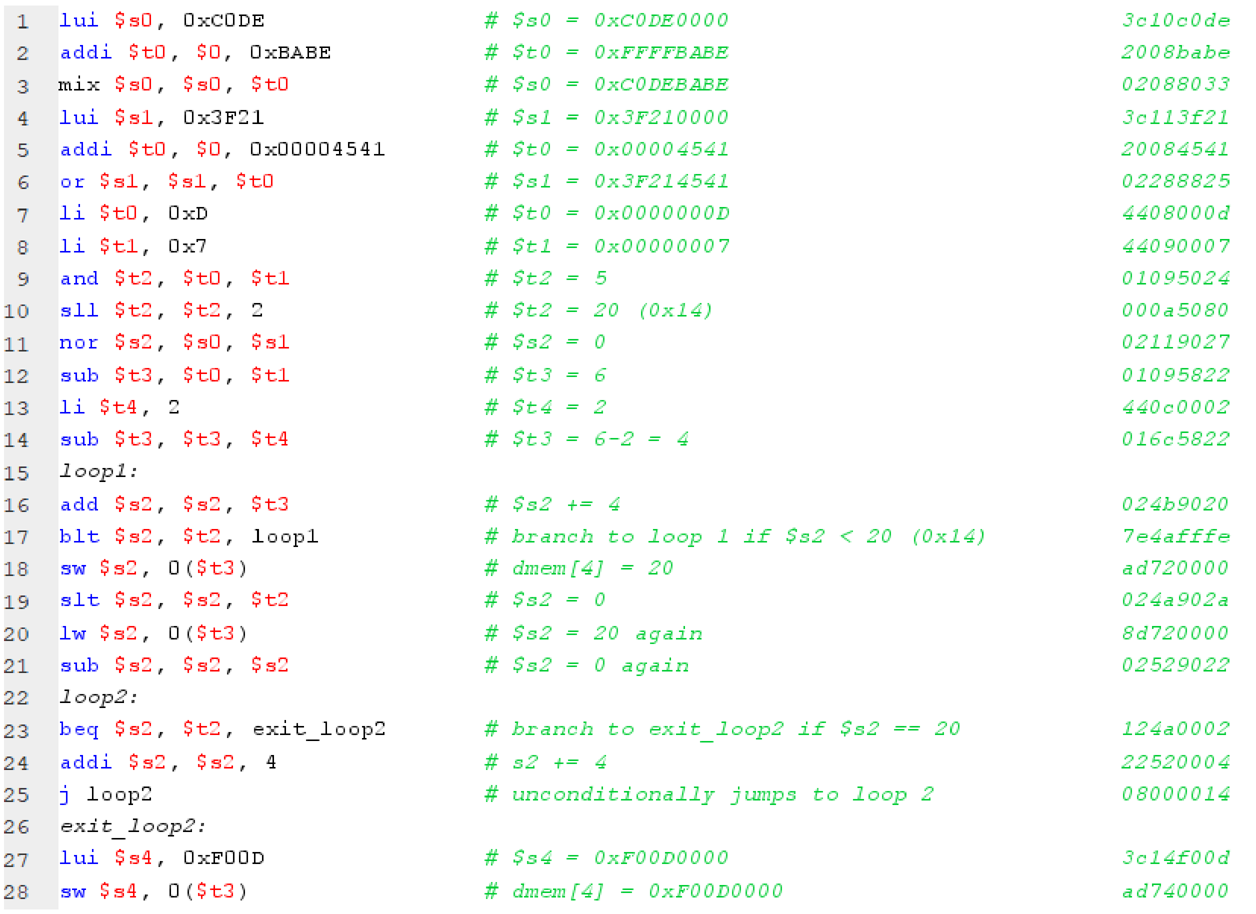
In line 2 we can see that alucontrol was expanded to 4 bits. On line 4 we see the addition of a new 5-bit input, the shamt. On line 5 we can see the addition of a new signal output called lessthan. On lines 9 and 10 we can see that the only change was that the msb of alucontrol is now bit 3 because alucontrol was expanded to 4 bits. On line 13, we can see that the case being checked are now the bits 2:0 of alucontrol instead of 1:0 when alucontrol was only 3 bits wide. The consequence of this is that we now have more options for instructions in the alu. Note that the alucontrol[1:0] for and, or, add, sub, and slt is still the same. A 0 was just added on the leftmost bit. We can see for alucontrol[2:0] = 0b100 that to perform sll, we shift b by bits determined by the shamt. B for sll comes from the regfile as sll is an r-type instruction.We see on line 19 the operation for the custom instruction ‘mix’. What it does is basically concatenating the first 16 bits of the first operand, a with the last 16 bits of the second operand, b. This is an r-type instruction so we expect b to be from the regfile. On line 20, we just return b because for these two instructions, ‘lui’ and ‘li’, the operations is already done by the new modules zeroext and shiftleft. It’s just a matter of which output from the 4-1 multiplexer b of the alu will get. On line 21 we see the operation performed for ‘nor’. It is just a simple not on the or of a and b.On line 25 we see that lessthan is 0b1 if the result of slt is 1, otherwise it’s 0. Remember that the alucontrol for blt is the alucontrol for slt in maindec.

Schematic diagram of modified Single-Cycle MIPS processor



The diagram above summarizes the changes in the single-cycle MIPS processor. The components, wires, and labels colored red are the ones that are either new or modified. We see that the control unit is red because we know that maindec.sv and aludec.sv were modified as explained earlier. We see the new output called ‘blt’ going to an and gate with the new ‘lessthan’ signal from the alu. The new and gate and or gate are responsible for generating the correct pcsrc value to determine whether the instruction should fetch pcbranch or not. Next, we see that alucontrol is now 4 bits wide and that is why the alu is highlighted red because a lot were modified inside as seen in the alu.sv section. Next, we see the consequence of expanding alusrc to 2 bits. srcbmux is now a 4 to 1 multiplexer. That was done to accommodate 2 more possible values for srcb of the alu. The first addition is shiftedforlui, the output of the new module called luishifter. This module is responsible for generating the correct output for the lui instruction. The second addition is zeroextended, the output of the new module called zerextend. This module is responsible for generating the correct output for the li instruction. Next, we see a new input to alu called the shamt which is instr[10:6]. This is necessary in performing sll inside the alu.

Test code

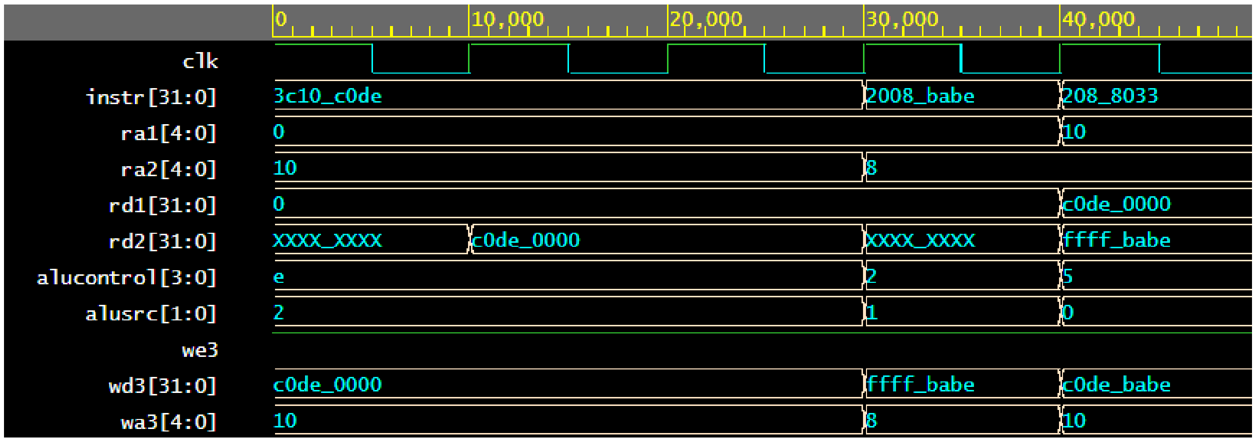


```
1  lui $s0, 0xC0DE                 # $s0 = 0xC0DE0000                                3c10c0de
2  addi $t0, $0, 0xBABE            # $t0 = 0xFFFFBABE                                2008babe
3  mix $s0, $s0, $t0               # $s0 = 0xC0DEBABE                                02088033
4  lui $s1, 0x3F21                 # $s1 = 0x3F210000                                3c113f21
5  addi $t0, $0, 0x00004541        # $t0 = 0x00004541                                20084541
6  or $s1, $s1, $t0                # $s1 = 0x3F214541                                02288825
7  li $t0, 0xD                     # $t0 = 0x0000000D                                4408000d
8  li $t1, 0x7                     # $t1 = 0x00000007                                44090007
9  and $t2, $t0, $t1               # $t2 = 5                                         01095024
10 sll $t2, $t2, 2                 # $t2 = 20 (0x14)                                 000a5080
11 nor $s2, $s0, $s1               # $s2 = 0                                         02119027
12 sub $t3, $t0, $t1               # $t3 = 6                                         01095822
13 li $t4, 2                       # $t4 = 2                                         440c0002
14 sub $t3, $t3, $t4               # $t3 = 6-2 = 4                                   016c5822
15 loop1:
16 add $s2, $s2, $t3               # $s2 += 4                                        024b9020
17 blt $s2, $t2, loop1             # branch to loop 1 if $s2 < 20 (0x14)             7e4afffe
18 sw $s2, 0($t3)                  # dmem[4] = 20                                    ad720000
19 slt $s2, $s2, $t2               # $s2 = 0                                         024a902a
20 lw $s2, 0($t3)                  # $s2 = 20 again                                  8d720000
21 sub $s2, $s2, $s2               # $s2 = 0 again                                   02529022
22 loop2:
23 beq $s2, $t2, exit_loop2        # branch to exit_loop2 if $s2 == 20               124a0002
24 addi $s2, $s2, 4                # s2 += 4                                         22520004
25 j loop2                         # unconditionally jumps to loop 2                 08000014
26 exit_loop2:
27 lui $s4, 0xF00D                 # $s4 = 0xF00D0000                                3c14f00d
28 sw $s4, 0($t3)                  # dmem[4] = 0xF00D0000                            ad740000
```

The code on the top left is the test instruction created for the purpose of showcasing the modified processor's capability of running all instructions both the pre-included ones from lab exercise 12 and the new ones required by the project. The image on the right shows the machine code translation of the MIPS code on the left. Lines 1-3 just stores 0xC0DEBABE into $s0 or $16 or $0x10. The purpose here is to showcase that the modified processor is capable of performing lui, addi, and mix properly. We should expect 0xC0DEBABE in the wd3[31:0] of instruction 0x2088033 in the waveforms. Lines 4-6 just stores 0x3F214541 into $s1 or $16 or $0x11. The purpose of this is to showcase the capability of the modified processor to perform 'or' properly. We should expect to see 0x3F214541 on the wd3[31:0] of instruction 0x02288825. Lines 7 and 8 shows the processor's capability to perform li properly. We should expect to see 0xD on the wd3[31:0] of instruction 0x4408000D and 0x7 on 0x44090007. Line 9 showcases the processor's ability to perform 'and'. We should expect to see 0x5 on the wd3[31:0] of instruction 0x01095024. Line 10 showcases the processor's ability to perform sll. We should expect to see a value of 2 in the shamt and 20 or 0x14 in the wd3[31:0] of instruction 0x000a5080. Line 11 showcases the processor's ability to perform bitwise nor. We should expect to see 0x0 on the wd3[31:0] of instruction 0x02119027. Line 12 showcases the processor's ability to perform sub. We should expect to see 0x6 on the wd3[31:0] of instruction 0x01095822. Line 13 is just dedicated for loading 2 into register $t4. We should expect to see 0x2 on the wd3[31:0] of instruction 0x440c0002. Line 14 performs sub and we should expect to see 0x4 on the wd3[31:0] of instruction 0x016c5822. Line 16 showcases the processor's capability to perform add. It is the increment part of the loop in the program and we expect to see instruction 0x024b9020 in the waveforms 5 times. Line 17 showcases the processor's capability to perform blt. We should expect to see branch signal set to low while blt and lessthan are set to high in the first 4 appearances of instruction 0x7e4afffe. On the fifth appearance of this instruction, lessthan should be set to low so we do not branch anymore. Line 18 shows the the processor's capability to perform sw. We expect to see 0x4 on the a[31:0] line, 0x14 on the wd[31:0] line, and memwrite set to high on instruction 0xad720000. Line 19 showcases the processor's ability to perform slt. We should expect to see 0x0 on the wd3[31:0] line of instruction 0x024a902a. Line 20 showcases the processor's capability to perform lw. We expect to see 0x4 on the a[31:0] line, 0x14 on the wd3[31:0] line, and memwrite set to low on instruction 0x8d720000. Line 21 again is showcasing the processor's ability to perform sub and we should expect 0x0 on the wd3[31:0] line on instruction 0x02529022. Line 23 showcases the processor's ability to perform beq. On the first 5 appearances of instruction 0x124a0002, we should expect to see the branch signal and zero signal set to low, only on the 6th appearance will those two signals be set to high. Since the branch of beq is still on line 27, we expect to see instructions 0x22520004 and 0x08000014 appear 5 times. Line 24 is just the increment portion of the loop. Line 25 on the other hand showcases the processor's ability to perform the jump operation. We expect to see the jump signal set to high here. After the end of the loop, Line 27 just sets $s4 to 0XF00D0000 and we should expect to see 0xF00D0000 on the wd3[31:0] line. Line 28 showcases again the processor's capability to perform sw and we expect to see 0x4 on the a[31:0] line, 0xF00D0000 on the wd[31:0] line, and memwrite set to high on instruction 0xad740000.

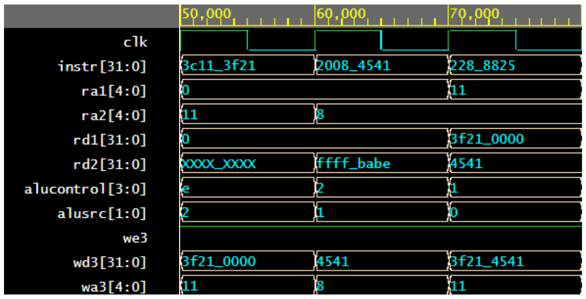
Waveforms of the testcode

For lines 1-3:



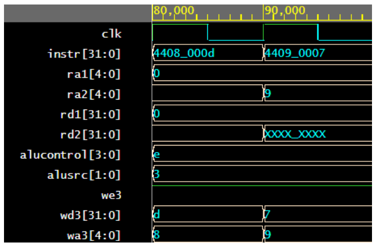
We can clearly see in 0x3c10c0de that the lui worked in shifting 0x0000c0de 16 bits to the left because the value to be written into $0x10 or $16 or $s0 is 0xc0de0000. We can also see in 0x2008babe that addi worked because wd3 = 0xffffbabe. We can see in 0x02088033 that mix worked because wd3 = 0xc0debabe (rd1 = 0xc0de0000 and rd2 = 0xffffbabe).

For lines 4-6:



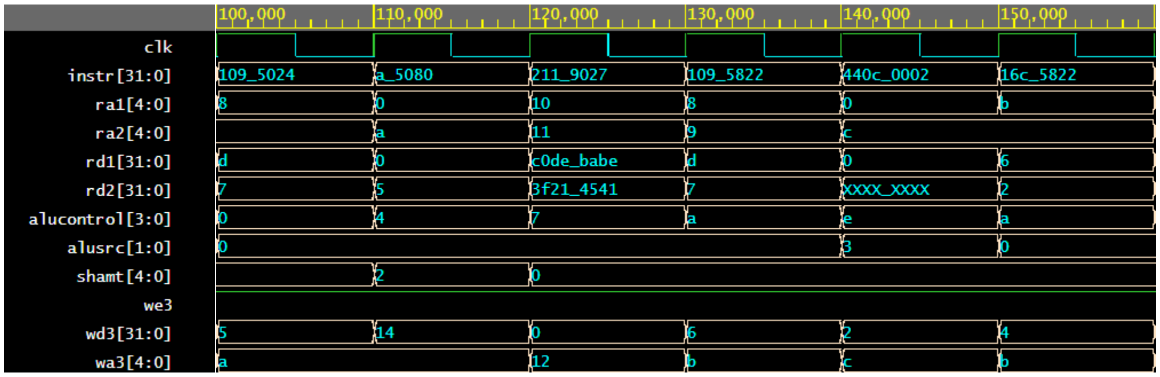
We can see in 0x3c113f21 that lui worked because we can clearly see in wd3 that it contains the value 0x3f210000. We know that addi worked in 0x20084541 because we can see that wd3 = 0x00004541 since our rd1 = 0. We can see in 0x02288825 that ‘or’ worked because we now have 0x3f214541 in wd3.

For lines 7 and 8:



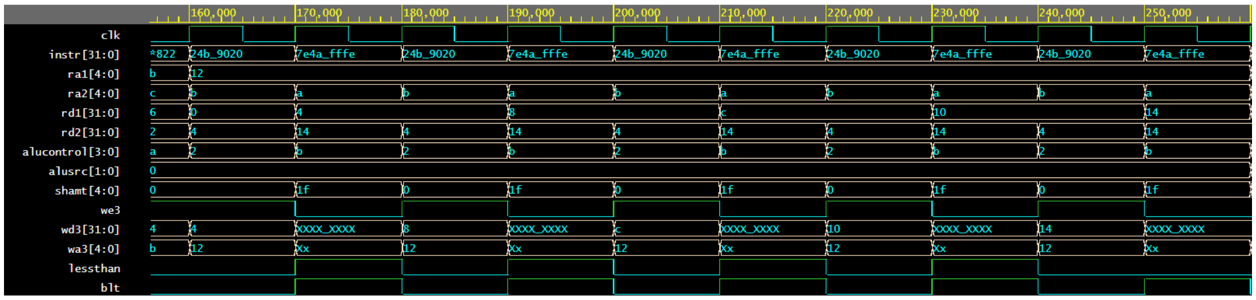
We can see that li worked in 0x4408000d and 0x44090007 because we see that d and 7 were zero extended instead as seen on wd3.

For lines 9-14:



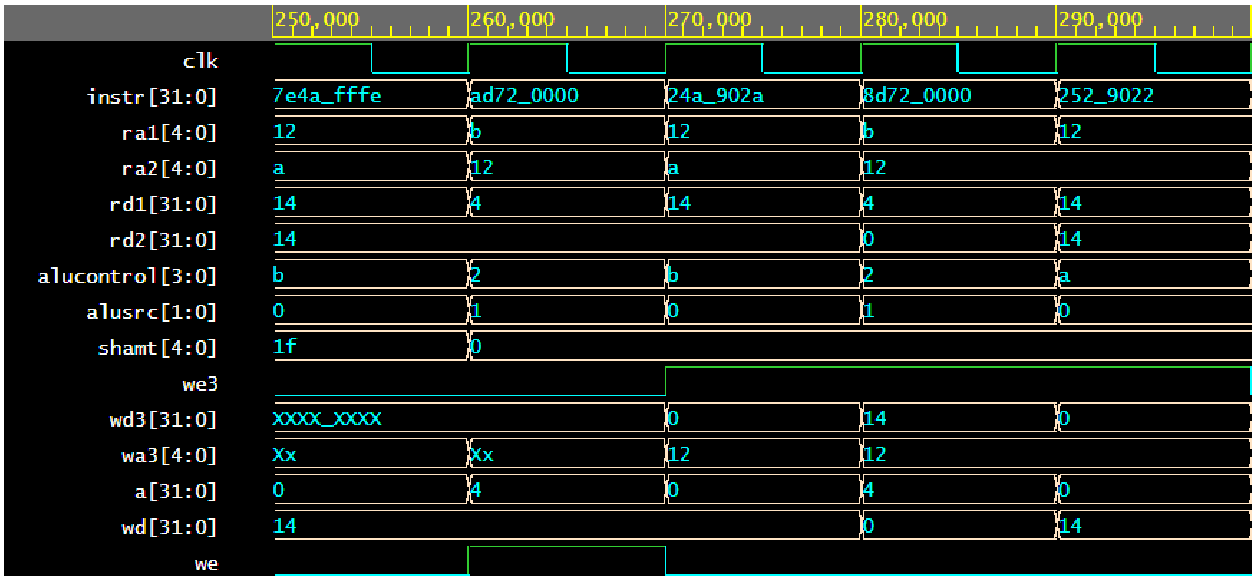
We can see in 0x1095024 that 'and' worked because the bitwise 'and' of 0xd and 0x7 is 5 and we can see 5 on wd3. We can see in 0x000a5080 that sll worked because the shifting 0x5 to the left by 2 is like multiplying 5 by 4 which is 20 or 0x14 and we can see 0x14 on wd3. We can see in 0x02119027 that nor worked because the bitwise nor of 0xc0debabe and 0x3f214541 is 0 and we can see 0 in wd3.This is because ~(0xc0debabe) = 0x3f214541 and performing nor on opposing bits would always result to 0. We can see in 0x01095822 that sub worked because the difference of 0xd (13) and 0x7 is 6 and we see 6 on wd3. We can see in 0x440c0002 that li worked because we can see that wd3 contains 0x2 (0x0002 was zero extended). We can see in 0x016c5822 that sub worked because the difference of 6 and 2 is 4 and we can see 4 in wd3.

For lines 16 and 17:



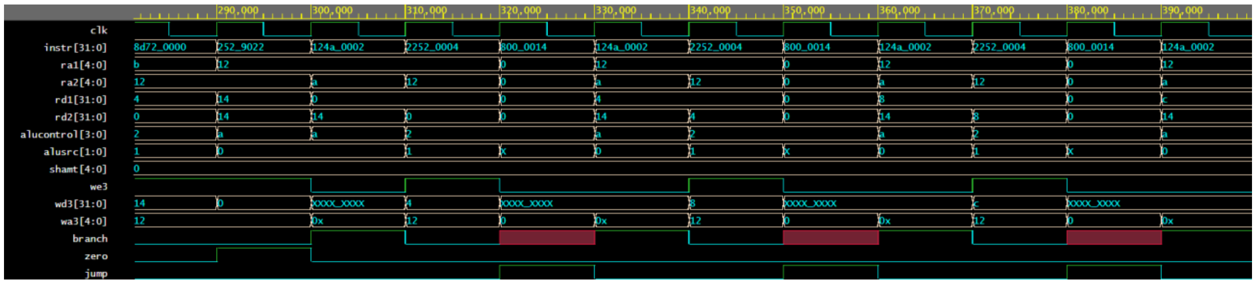
We know that blt worked because as discussed in the testcode part, we expected that in the 4 iterations of blt (0x7e4afffe), it will branch and we can see that lessthan and blt is set to high. But on the 5th iteration, we can see that lessthan is now set to low because 0x14 is no longer less than 0x14.

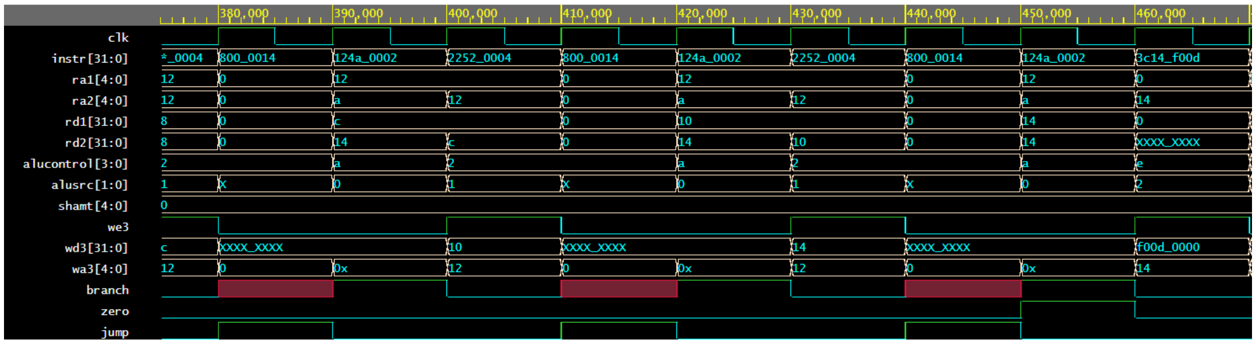
For lines 18-21:



We can see in 0xad720000 that sw worked because wd[31:0] contains 0x14 and the a[31:0] is 4 while we is set to high so we can write to dmem. We know that slt worked because we can see in 0x024a902a that wd3 = 0 because 0x14 is not less than 0x14. We can see in 0x8d720000 that lw worked because we3 is set to high meaning we can write to regfile and wd3 = 0x14, the value contained by address 0x4. We can see in 0x02529022 that sub worked because the difference of 0x14 and 0x14 is 0 and we can see 0 in wd3.

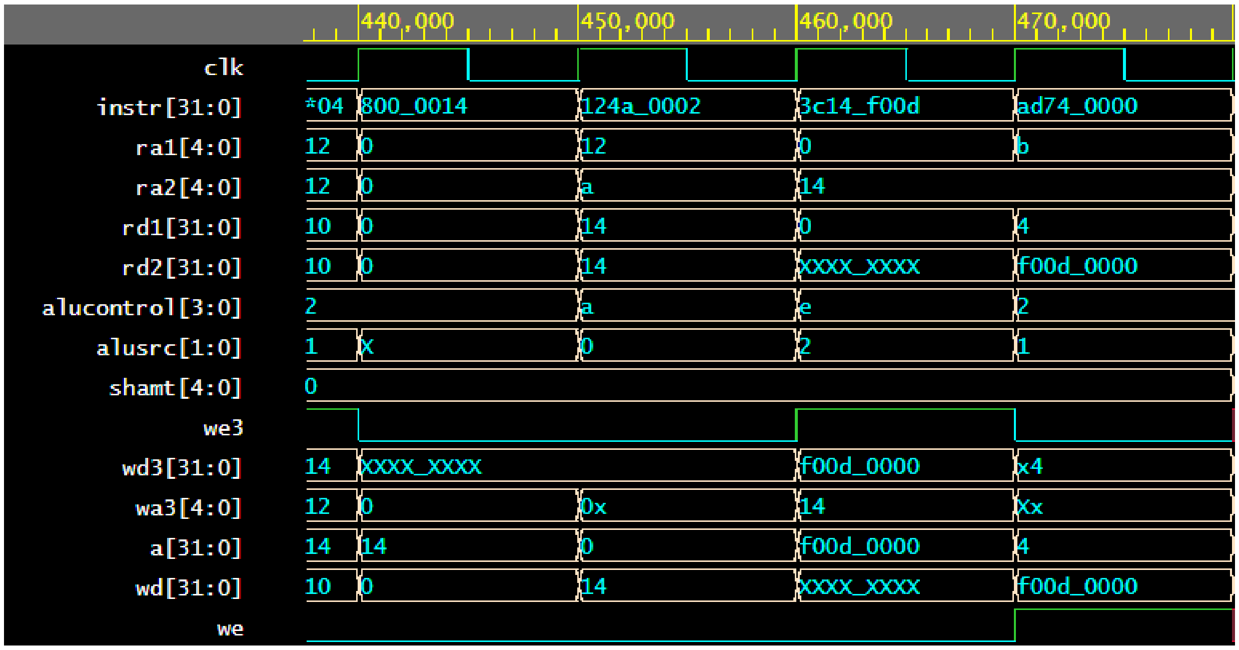
For lines 23-25:





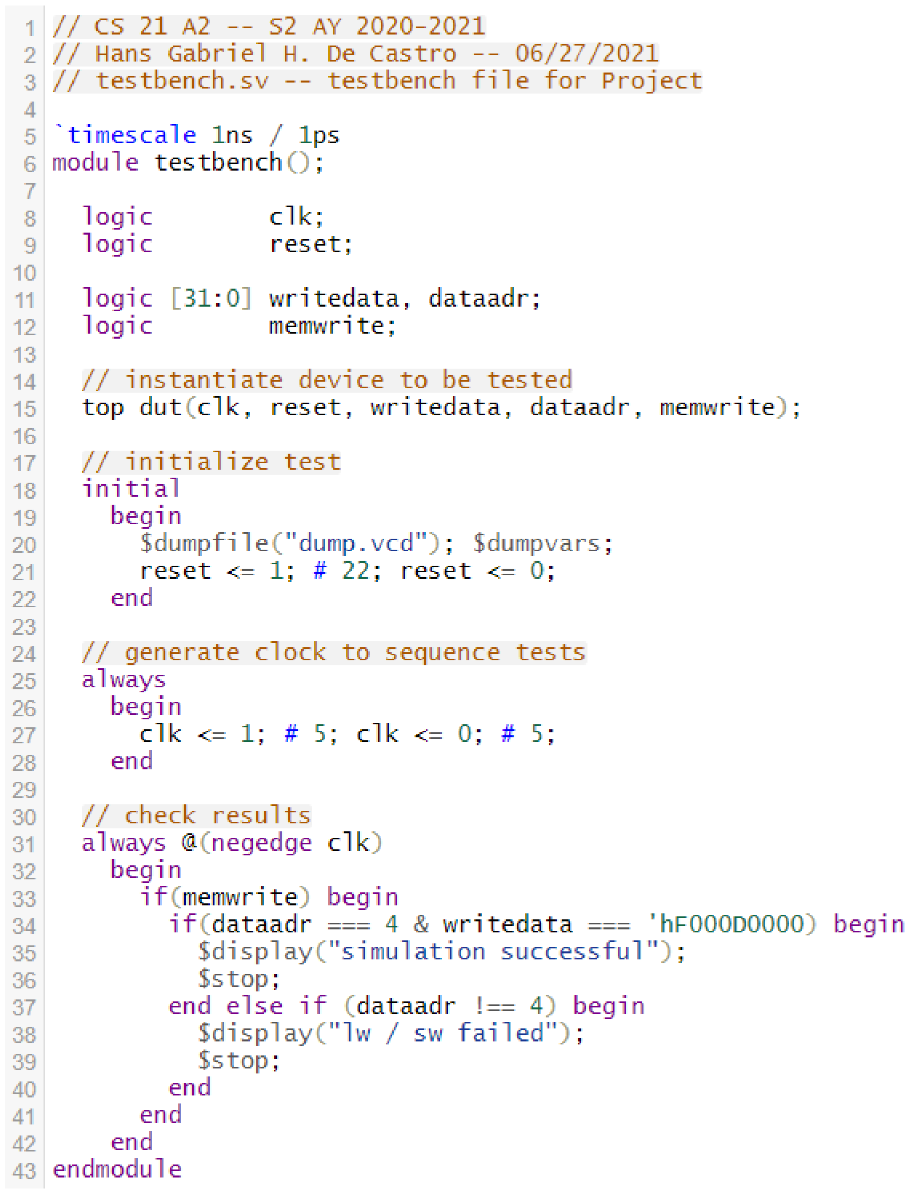
We can see here that beq (0x124a0002) worked because beq ran 6 times and it in the first 5 times, zero was set to low hence, we did not branched to the branch target address. We know that in the 5th iteration, the branching worked because as we see that the zero signal is set to high, the next instruction is 0x3c14f00d which is in line 27. We also know that addi worked because we can see in wd3 that it was increasing from 0x0 to 0x14 in increments of 4. We also know that j worked because we were able to repeatedly go to instruction 0x124a0002 (blt) and we see that the jump signal is set to high there.

For lines 27 and 28:


440,000
450,000
460,000
470,000
clk
instr[31:0] *04 800_0014 124a_0002 3c14_f00d ad74_0000
ra1[4:0] 12 0 12 0 b
ra2[4:0] 12 0 a 14
rd1[31:0] 10 0 14 0 4
rd2[31:0] 10 0 14 XXXX_XXXX f00d_0000
alucontrol[3:0] 2 a e 2
alusrc[1:0] 1 X 0 2 1
shamt[4:0] 0
we3
wd3[31:0] 14 XXXX_XXXX f00d_0000 x4
wa3[4:0] 12 0 0x 14 Xx
a[31:0] 14 14 0 f00d_0000 4
wd[31:0] 10 0 14 XXXX_XXXX f00d_0000
we

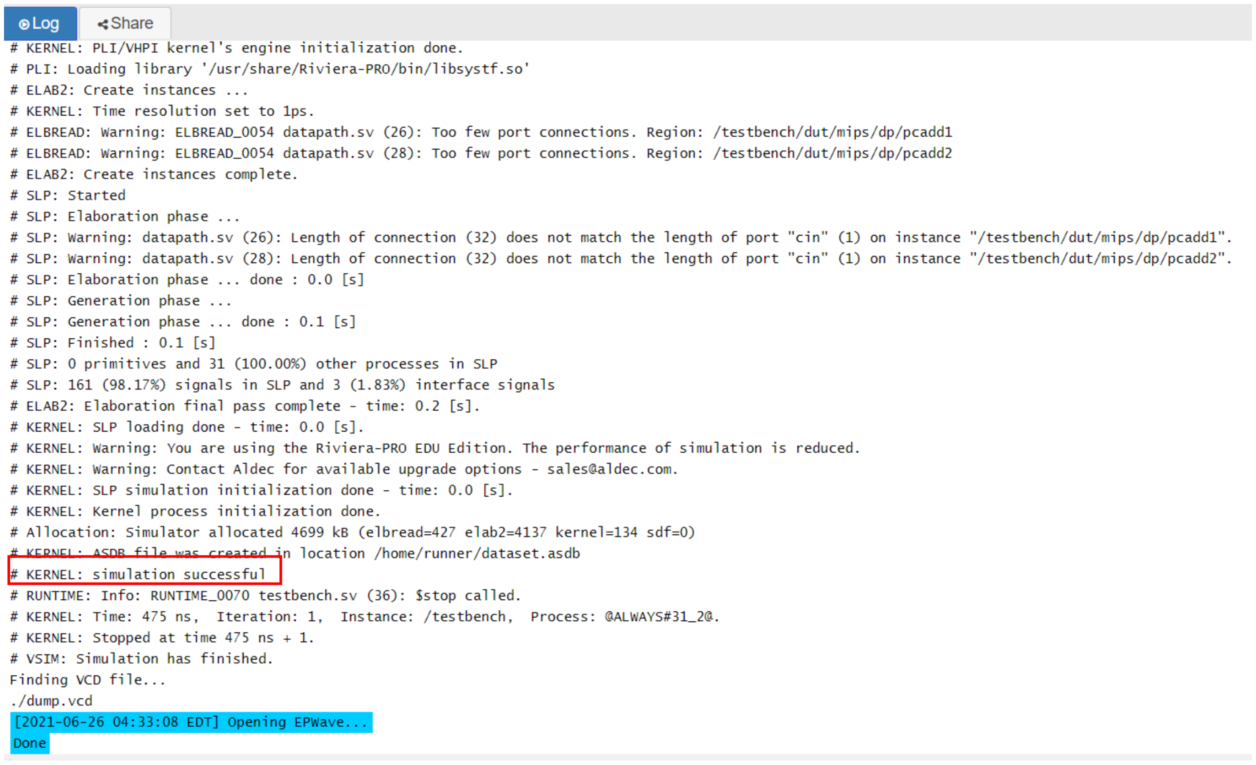
We know that lui worked in 0x3c14f00d because we can see that wd3 = 0xf00d0000 (shifted to the left by 16 bits). We can see in 0xad740000 that sw worked because we can see that we is set to high, wd = 0xf00d0000, and the address that we want to write to is 4.

testbench.sv



```
 1 // CS 21 A2 -- S2 AY 2020-2021
 2 // Hans Gabriel H. De Castro -- 06/27/2021
 3 // testbench.sv -- testbench file for Project
 4 
 5 `timescale 1ns / 1ps
 6 module testbench();
 7 
 8   logic        clk;
 9   logic        reset;
10 
11   logic [31:0] writedata, dataadr;
12   logic        memwrite;
13 
14   // instantiate device to be tested
15   top dut(clk, reset, writedata, dataadr, memwrite);
16 
17   // initialize test
18   initial
19     begin
20       $dumpfile("dump.vcd"); $dumpvars;
21       reset <= 1; # 22; reset <= 0;
22     end
23 
24   // generate clock to sequence tests
25   always
26     begin
27       clk <= 1; # 5; clk <= 0; # 5;
28     end
29 
30   // check results
31   always @(negedge clk)
32     begin
33       if(memwrite) begin
34         if(dataadr === 4 & writedata === 'hF000D0000) begin
35           $display("simulation successful");
36           $stop;
37         end else if (dataadr !== 4) begin
38           $display("lw / sw failed");
39           $stop;
40         end
41       end
42     end
43 endmodule
```

The code above is the testbench that was used in running the test code in the previous pages. The only portion that was modified here from the testbench file used in lab exercise 12 is the ‘check results’ portion. On line 34, dataadr is checked if it is equal to 4 and if writedata is equal to 0xF000D0000 because it is the last instruction that is supposed to be executed in the test code. The way we would check if there is something wrong in the execution of the code is if dataadr is not equal to 4 because nowhere in the test code will there be an instance that the address that is going to be accessed by lw or sw is not 4.



```
# KERNEL: PLI/VHPI kernel's engine initialization done.
# PLI: Loading library '/usr/share/Riviera-PRO/bin/libsystf.so'
# ELAB2: Create instances ...
# KERNEL: Time resolution set to 1ps.
# ELBREAD: Warning: ELBREAD_0054 datapath.sv (26): Too few port connections. Region: /testbench/dut/mips/dp/pcadd1
# ELBREAD: Warning: ELBREAD_0054 datapath.sv (28): Too few port connections. Region: /testbench/dut/mips/dp/pcadd2
# ELAB2: Create instances complete.
# SLP: Started
# SLP: Elaboration phase ...
# SLP: Warning: datapath.sv (26): Length of connection (32) does not match the length of port "cin" (1) on instance "/testbench/dut/mips/dp/pcadd1".
# SLP: Warning: datapath.sv (28): Length of connection (32) does not match the length of port "cin" (1) on instance "/testbench/dut/mips/dp/pcadd2".
# SLP: Elaboration phase ... done : 0.0 [s]
# SLP: Generation phase ...
# SLP: Generation phase ... done : 0.1 [s]
# SLP: Finished : 0.1 [s]
# SLP: 0 primitives and 31 (100.00%) other processes in SLP
# SLP: 161 (98.17%) signals in SLP and 3 (1.83%) interface signals
# ELAB2: Elaboration final pass complete - time: 0.2 [s].
# KERNEL: SLP loading done - time: 0.0 [s].
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 4699 kB (elbread=427 elab2=4137 kernel=134 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: simulation successful
# RUNTIME: Info: RUNTIME_0070 testbench.sv (36): $stop called.
# KERNEL: Time: 475 ns,  Iteration: 1,  Instance: /testbench,  Process: @ALWAYS#31_2@.
# KERNEL: Stopped at time 475 ns + 1.
# VSIM: Simulation has finished.
Finding VCD file...
./dump.vcd
[2021-06-26 04:33:08 EDT] Opening EPWave...
Done
```

We can verify above that the test code indeed ran successfully with the testbench because as highlighted in red above, we see that the testbench displayed “simulation successful” in the log.