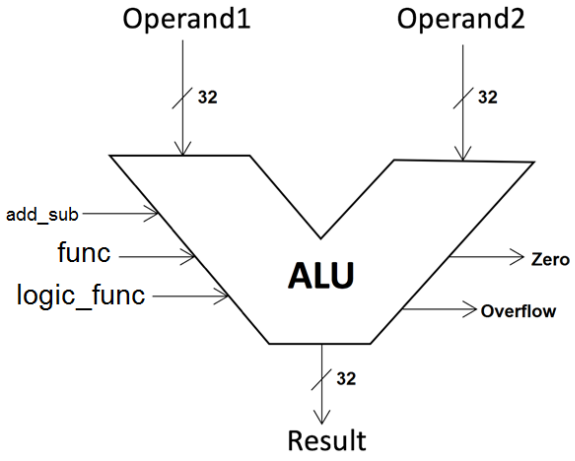
COEN 316 – Computer Architecture and Design **Jafar Abbas**
LAB 1 – Arithmetic and Logic Unit 26346650
Fall 2017 – Friday 13 (Spooky!)

Objectives

The purpose of this lab is to design, synthesize, simulate and test a 32-bit ALU which handles addition, substraction, logic operations, load upper immediate, set less than, zero and overflow for on signed integers. The ALU will be built upon and used as part of a more complex CPU design in future labs.



Procedure

This lab follows the procedure outlined in Parts I, II, III, and IV of the tutorial "Digital Logic Simulation and Synthesis Using Modelsim, Precision RTL, and Xilinx ISE". The end-goal is to simulate, synthesize and download to the Xilinx FPGA board the 32-bit ALU described below in the LAB 1 manual.

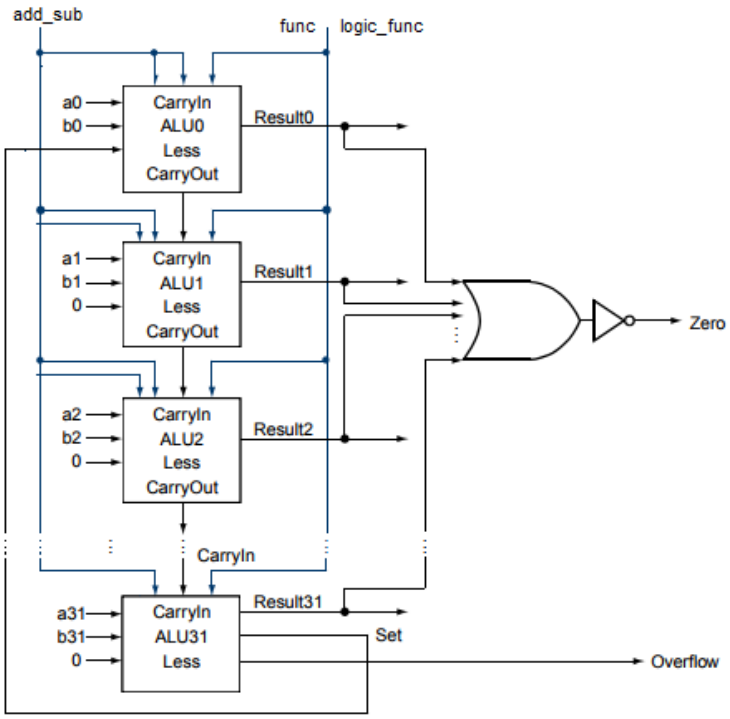
Requirements

1) Implement the ALU using any style of VHDL
2) Support signed integer overflow and arithmetic
3) Single-cycle implementation
4) Satisfy the following "truth" table

`func`	`output`	Comments
00	y	used to implement the `lui` instruction
01	"000...MSB" of adder_subttract	used for the `slt` (set less than zero) instruction
10	output of adder_subtract	`x+y` if `add_sub` = '0' `x-y` if `add_sub` = '1'
11	output of logic unit	x AND y if `logic_func` = "00 x OR y if `logic_func` = "01" x XOR y if `logic_func` = "10" x NOR y if `logic_func` = "11"

Results

My design:



I followed the **structural approach** (with port maps) to build my ALU for the following reasons:

Testing-> It is easier to verify correctness of a 1 bit ALU which makes debugging easier if the design is broken into components (isolates errors between 1bit ALU logic, MSB ALU logic, connections)

Reusability-> 1 bit ALUs can be used to build N bit ALU by just combining in multiples, the 1bit ALU can also be used easily as part of another system

Customization-> Bit-level functionality can be made custom for certain bits by just using a different architecture for certain bits

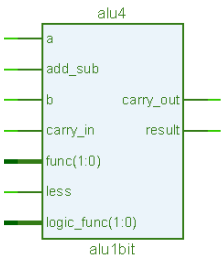
Optimization-> it is easier to optimize 1 small component than to try to optimize the whole thing at the same time

Synthesizer "decoupling"-> The design is closer to "reality" which makes the final implementation less dependent on the synthesizer used

Adaptability-> The 1 bit ALU can later be modified to implement its function in a different way but overall the same behavior will be seen at the 32bit level

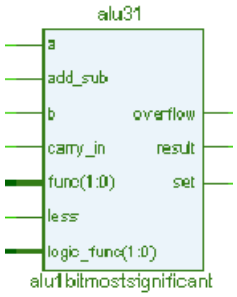
Furthermore, since I was not creating anything new I found this approach more intuitive as it implements a well-established design we learned in class. Certainly there are arguments in favor of different approaches as well.

Entity for alu1bit:




```
library IEEE;
use IEEE.std_logic_1164.all;

entity alu1bit is

port(
    -- input of 1bit alu
    a, b, less, carry_in : in std_logic;

    add_sub : in std_logic ; -- 0 = add , 1 = sub
    logic_func : in std_logic_vector(1 downto 0 ) ;  -- 00 = AND, 01 = OR , 10 = XOR , 11 = NOR
    func : in std_logic_vector(1 downto 0 ) ;   -- 00 = lui, 01 = setless , 10 = arith , 11 = logic

    -- output of 1bit alu
    result : out std_logic;
    carry_out : out std_logic
);

end alu1bit ;
```

Entity for alu1bitmostsignificant:




```
library IEEE;
use IEEE.std_logic_1164.all;

entity alu1bitmostsignificant is

port(
    -- input of 1bit alu most significant
    a, b, less, carry_in : in std_logic;

    add_sub : in std_logic ; -- 0 = add , 1 = sub
    logic_func : in std_logic_vector(1 downto 0 ) ;  -- 00 = AND, 01 = OR , 10 = XOR , 11 = NOR
    func : in std_logic_vector(1 downto 0 ) ;   -- 00 = lui, 01 = setless , 10 = arith , 11 = logic


    --output of 1bit alu most significant
    result : out std_logic;
    set : out std_logic;
    overflow: out std_logic
);

end alu1bitmostsignificant ;
```

Entity for alu32bit: (forgot to take blackbox picture)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity alu32bit is

port(

    -- input of 32bit alu
    a, b : in std_logic_vector(31 downto 0);    -- assuming SIGNED overflow logic, note:
                                                --possible to get overflow even if you don't have output carry bit
                                                --(so overflow logic should be done here or in MSB? by checking signs
    --a is x, b is y

    add_sub : in std_logic ; -- 0 = add , 1 = sub
    logic_func : in std_logic_vector(1 downto 0 ) ;   -- 00 = AND, 01 = OR , 10 = XOR , 11 = NOR
    func : in std_logic_vector(1 downto 0 ) ;    -- 00 = lui, 01 = setless , 10 = arith , 11 = logic

    -- output of 32bit alu
    result : out std_logic_vector(31 downto 0) ;
    overflow : out std_logic ;
    zero : out std_logic

);

end alu32bit ;
```

Architecture of alu1bit:

```
architecture lab1 of alu1bit is

begin

    main: process (a, b, less, carry_in, add_sub, logic_func, func)
    begin

        -- switch based on func
        case func is
            when "00" => --lui
                result <= b;
                carry_out <= '0'; --no carry

            when "01" => --setless

                result <= less;
                carry_out <= (a AND (NOT b)) OR (a AND carry_in) OR ((NOT b) AND carry_in);
                -- invert b
                --because we're doing substraction during setless,
                --the carry_in will be '1' for the LSB (see carry_in_hack)

            when "10" => --arith (1-bit, 2's complement based)
                         --(add_sub decides if we invert b,
                         --carry_in will be 0 for the LSB for addition and 1 during substraction)

                result <= a XOR (b XOR add_sub) XOR carry_in;
                carry_out <= (a AND (b XOR add_sub)) OR (a AND carry_in) OR ((b XOR add_sub) AND carry_in);
                -- (b XOR add_sub) means invert b if add_sub is 1 (i.e. doing sub);

            when "11" => --logic (bit-wise, carries don't exist)

                case logic_func is

                    when "00" => --and
                        result <= a AND b;
                        carry_out <= '0';

                    when "01" => --or
                        result <= a OR b;
                        carry_out <= '0';

                    when "10" => --xor
                        result <= a XOR b;
                        carry_out <= '0';

                    when "11" => --nor
                        result <= a NOR b;
                        carry_out <= '0';

                    when others =>
                        result <= '0';
                        carry_out <= '0';

                end case;

            when others =>
                result <= '0';
                carry_out <= '0';

        end case;

    end process;

end lab1;
```

Architecture of alu1bitmostsignificant:

The MSB is simply a alu1bit with added set and overflow logic:

```
architecture lab1 of alu1bitmostsignificant is

-- reuse alu1bit implementation
component alu1bit
    port(
        a, b, less, carry_in, add_sub: in std_logic;
        logic_func : in std_logic_vector(1 downto 0 ) ;
        func : in std_logic_vector(1 downto 0 ) ;
        result, carry_out : out std_logic
    );
    end component;

    --extending functionality of alu1bit
    signal last_carry: std_logic;
    signal temp_result: std_logic;

begin

    --MSB alu works like alu1bit... but has extra features
    MSB: alu1bit
    port map (a => a, b => b, less => less, carry_in => carry_in,
              add_sub => add_sub, logic_func => logic_func, func => func,
              result => temp_result, carry_out => last_carry);

    result <= temp_result;

    -- setlessthan bit logic
    set <= a XOR (NOT b) XOR carry_in; --the MSB from substraction result is "set" to "less" at the LSB

    msb_extra: process (carry_in, last_carry, func) --, a, b, add_sub, temp_result)
    begin
        -- overflow logic
        if (func = "10") then -- arith

            overflow <= carry_in XOR last_carry;

        else
            overflow <= '0';

        end if;

    end process;

end lab1;
```

Architecture of alu32bit:

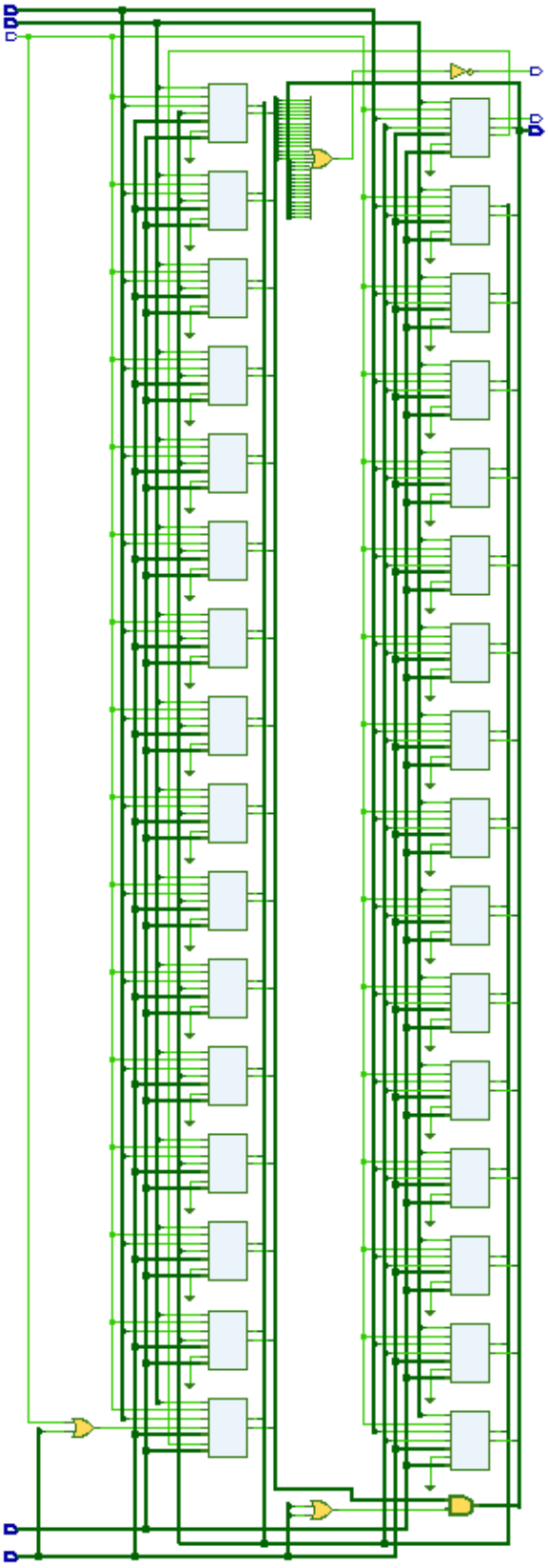
As you may imagine, a lot of port maps are incoming

```
architecture lab1 of alu32bit is

    constant logic_0: std_logic:= '0';
    signal carries: std_logic_vector(30 downto 0);      --vector holding carries of addition/substraction
    signal set_bit: std_logic:= '0';                    --taken from MSB and fed into LSB, if func is '01' (setless), it will be selected into result(0)
    signal temp_result: std_logic_vector(31 downto 0);  --holds results of all 1bit ALUs, used to compute Zero and LUI result masking
    signal carry_in_hack : std_logic := '0';            -- carry_in needs to be 1 for subs and for setless (01)

    component alu1bit
    port(
        a, b, less, carry_in, add_sub: in std_logic;
        logic_func : in std_logic_vector(1 downto 0 ) ;
        func : in std_logic_vector(1 downto 0 ) ;
        result, carry_out : out std_logic
    );
    end component;

    component alu1bitmostsignificant
    port(
        a, b, less, carry_in, add_sub: in std_logic;
        logic_func : in std_logic_vector(1 downto 0 ) ;
        func : in std_logic_vector(1 downto 0 ) ;
        result, set, overflow : out std_logic
    );
    end component;

    begin

        -- hacky, but carry_in needs to be 1 for subs and for setless (01)
        carry_in_hack <= add_sub OR func(0);

        -- if result is 000...000, zero is 1 else 0
        zero <= NOT( temp_result(0) OR temp_result(1) OR temp_result(2) OR temp_result(3)
                    OR temp_result(4) OR temp_result(5) OR temp_result(6) OR temp_result(7)
                    OR temp_result(8)OR temp_result(9) OR temp_result(10) OR temp_result(11)
                    OR temp_result(12) OR temp_result(13) OR temp_result(14) OR temp_result(15)
                    OR temp_result(16) OR temp_result(17) OR temp_result(18) OR temp_result(19)
                    OR temp_result(20) OR temp_result(21) OR temp_result(22) OR temp_result(23)
                    OR temp_result(24) OR temp_result(25) OR temp_result(26) OR temp_result(28)
                    OR temp_result(29) OR temp_result(30) OR temp_result(31));

        -- portmap each 1bit together to form 32bit alu

        alu0: alu1bit  --less is connected to the set_bit coming from MSB   --carry_in '1' when doing sub or slt
        port map (a => a(0), b => b(0), less => set_bit, carry_in => carry_in_hack, add_sub => add_sub,
                logic_func => logic_func, func => func, result => temp_result(0), carry_out => carries(0));

        alu1: alu1bit  --less is '0' always except LSB
                       --carry_in of following 1bitALU is carry_out of previous one
        port map (a => a(1), b => b(1), less => logic_0, carry_in => carries(0),
                add_sub => add_sub, logic_func => logic_func, func => func, result => temp_result(1), carry_out => carries(1));
```

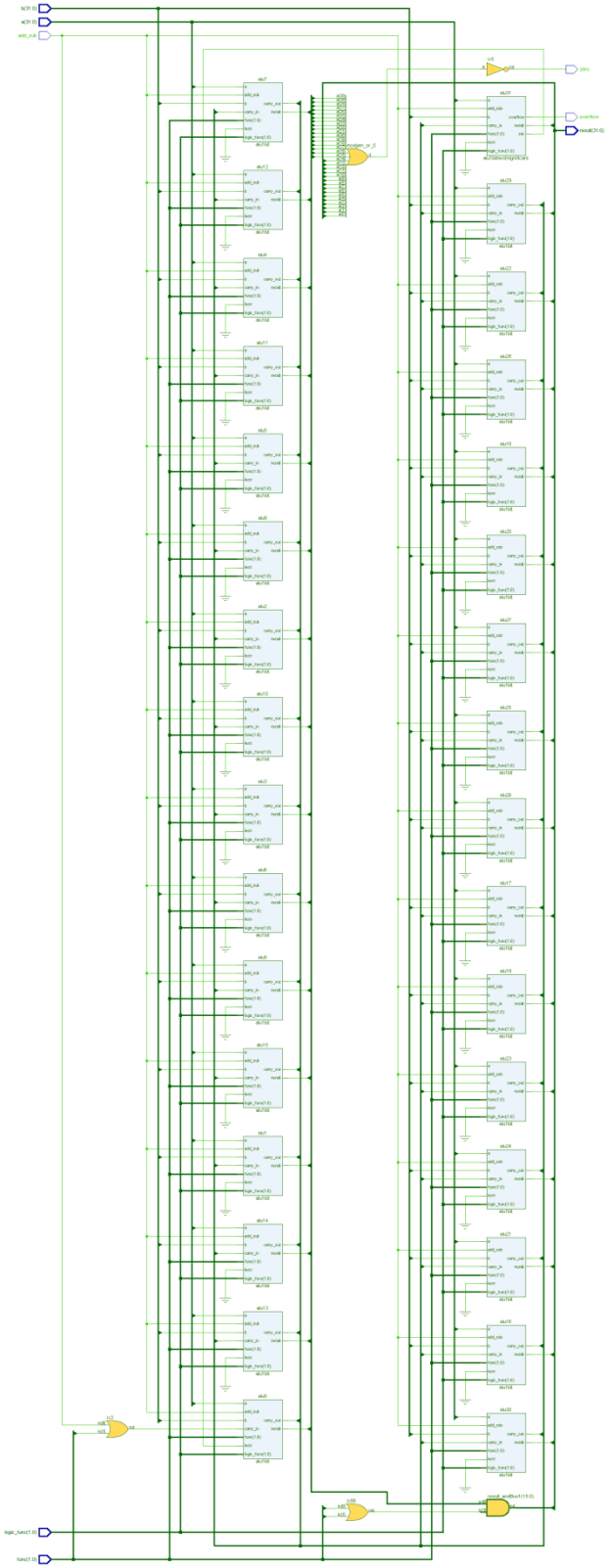
……..port maps of bits 2 to 29……..

```
        alu30: alu1bit
        port map (a => a(30), b => b(30), less => logic_0, carry_in => carries(29),
                add_sub => add_sub, logic_func => logic_func, func => func, result => temp_result(30), carry_out => carries(30));

        --overflow is decided at the MSB, the set bit is routed to the LSB alu and used during the setless than func
        alu31: alu1bitmostsignificant
        port map (a => a(31), b => b(31), less => logic_0, carry_in => carries(30),
                add_sub => add_sub, logic_func => logic_func, func => func, result => temp_result(31), overflow => overflow, set => set_bit);

        --special case
        alu32bitproc : process(func, temp_result)
        begin
            if (func = "00") then --lui
                result <= temp_result AND "11111111111111110000000000000000"; --mask lowest 16bits during LUI

            else
                result <= temp_result;
            end if;

        end process;

end lab1;
```

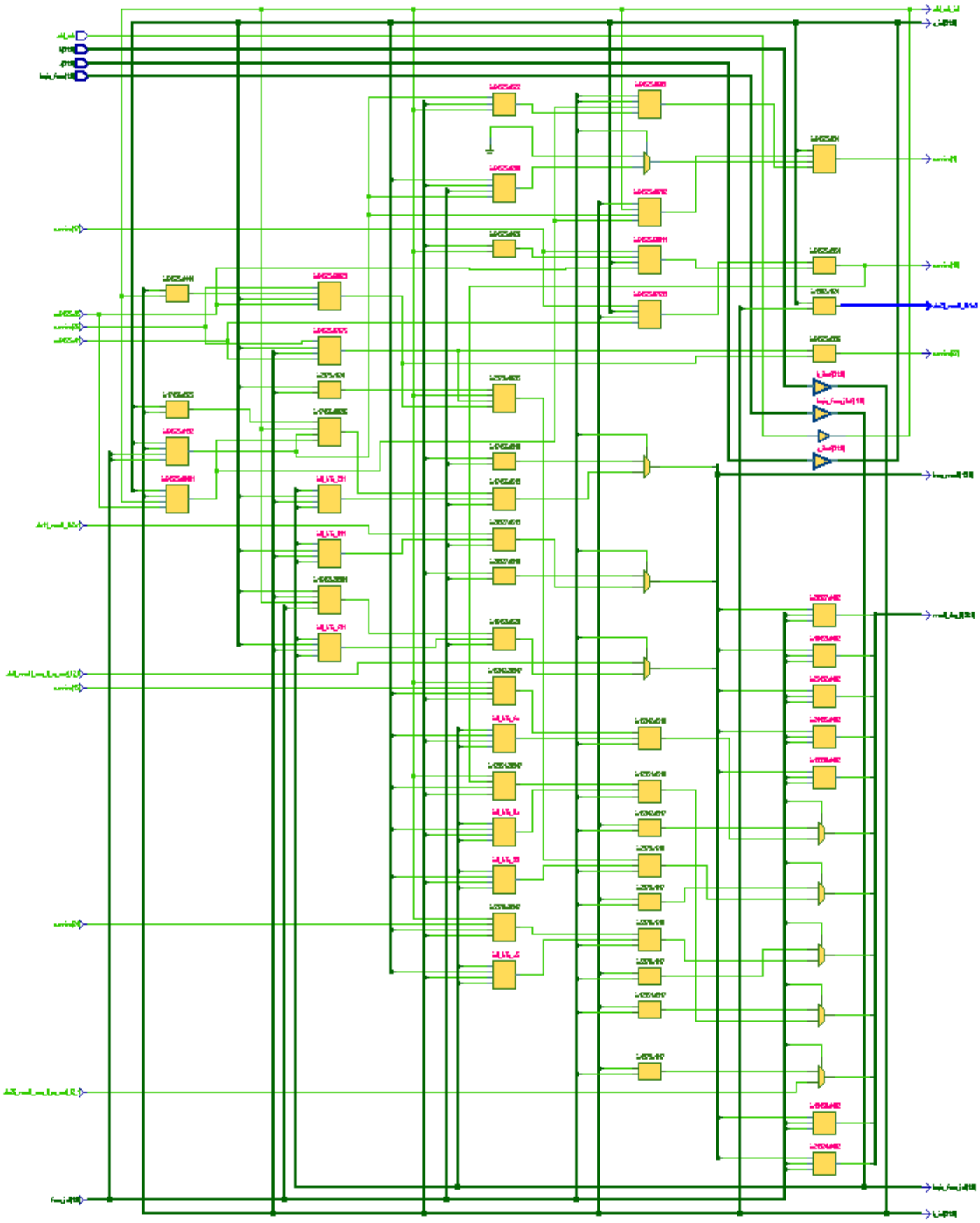
Synthesized circuit:



Another view of synthesized circuit:



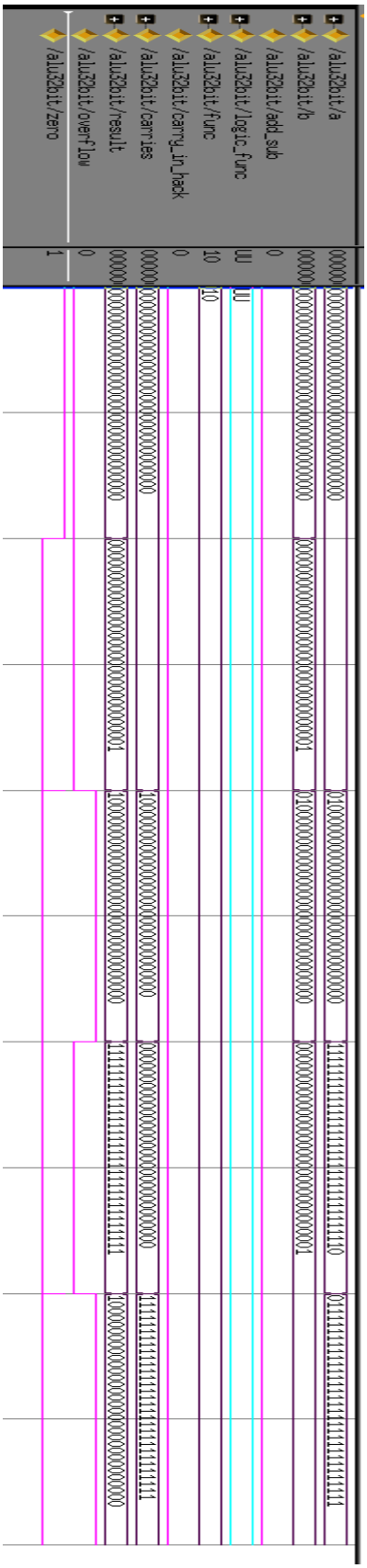
Technology mapped circuit:



Simulation Results

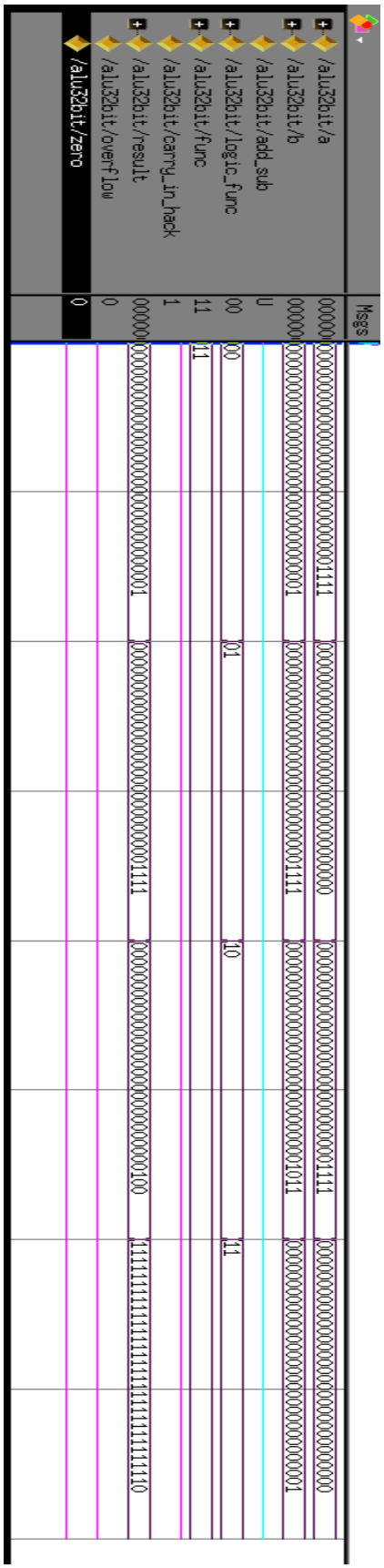
ADD

```
#arith
#arith ADD
force add_sub 0
force func 10
force a 00000000000000000000000000000000
force b 00000000000000000000000000000000
run 2


#arith
#arith ADD
force add_sub 0
force func 10
force a 00000000000000000000000000000000
force b 00000000000000000000000000000001
run 2


#arith
#arith ADD with overflow
force add_sub 0
force func 10
force a 01000000000000000000000000000000
force b 01000000000000000000000000000000
run 2


#arith
#arith ADD
force add_sub 0
force func 10
force a 11111111111111111111111111111110
force b 00000000000000000000000000000001
run 2


#arith
#arith ADD with overflow
force add_sub 0
force func 10
force a 01111111111111111111111111111111
force b 00000000000000000000000000000001
run 2
```



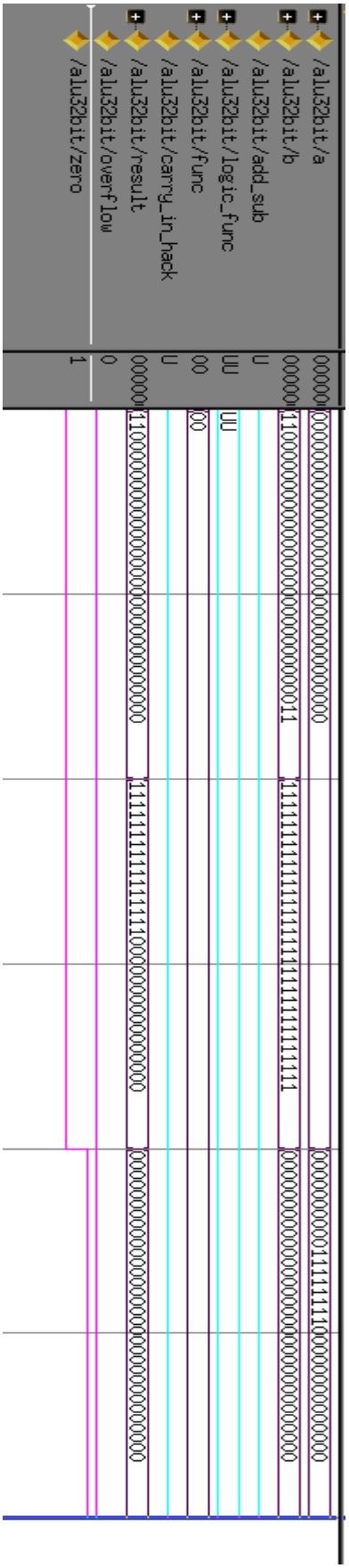
LOGIC

```
#logic
#logic AND
force func 11
force logic_func 00
force a 00000000000000000000000000001111
force b 00000000000000000000000000000001
run 2

#logic
#logic OR
force func 11
force logic_func 01
force a 00000000000000000000000000000000
force b 00000000000000000000000000001111
run 2

#logic
#logic XOR
force func 11
force logic_func 10
force a 00000000000000000000000000001111
force b 00000000000000000000000000001011
run 2

#logic
#logic NOR
force func 11
force logic_func 11
force a 00000000000000000000000000000000
force b 00000000000000000000000000000001
run 2
```



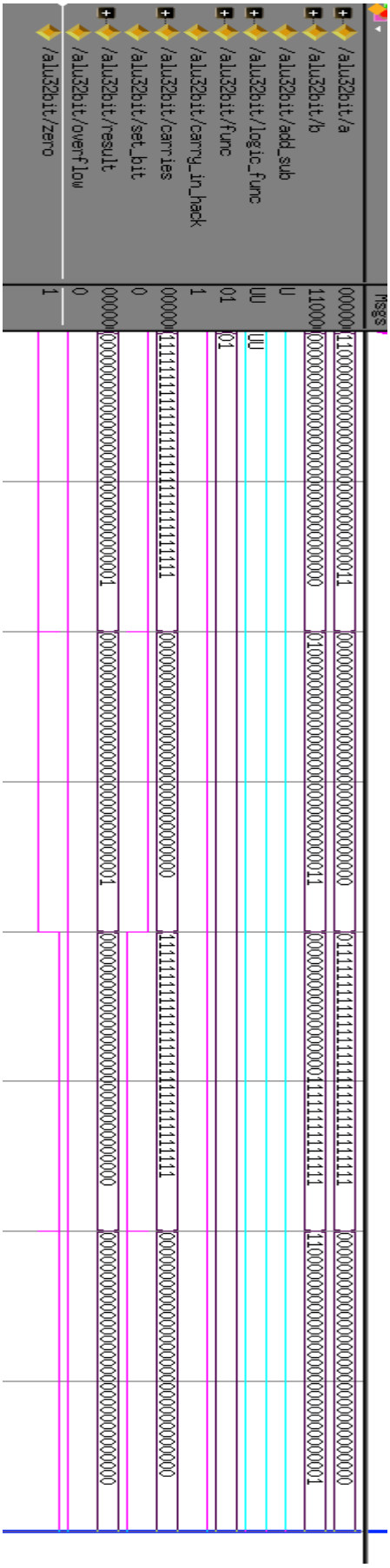
LUI

```
#lui
force func 00
force a 00000000000000000000000000000000
force b 11000000000000000000000000000011
run 2

#lui
force func 00
force a 00000000000000000000000000000000
force b 11111111111111111111111111111111
run 2

#lui
force func 00
force a 00000000001111111100000000000000
force b 00000000000000000000000000000000
run 2
```



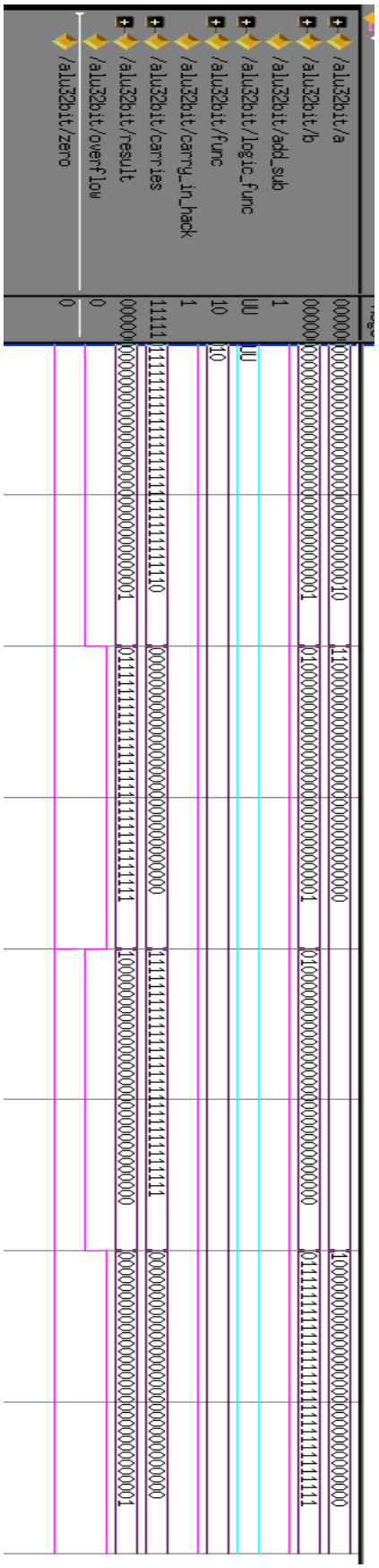
SLT

```
#setless true
force func 01
force a 11000000000000000000000000000011
force b 00000000000000000000000000000000
run 2

#setless true
force func 01
force a 00000000000000000000000000000000
force b 01000000000000000000000000000011
run 2

#setless false
force func 01
force a 01111111111111111111111111111111
force b 00000000000000000011111111111111
run 2

#setless false
force func 01
force a 00000000000000000000000000000000
force b 11000000000000000000000000000001
run 2

force func 01
force a 11111111111111111111111111111111
force b 11111111111111111111111111111111
run 2
```



SUB

```
#arith
#arith SUB
force add_sub 1
force func 10
force a 00000000000000000000000000000010
force b 00000000000000000000000000000001
run 2

#arith
#arith SUB with overflow
force add_sub 1
force func 10
force a 11000000000000000000000000000000
force b 01000000000000000000000000000001
run 2


#arith
#arith SUB
force add_sub 1
force func 10
force a 11000000000000000000000000000000
force b 01000000000000000000000000000000
run 2

#arith
#arith SUB with overflow
force add_sub 1
force func 10
force a 10000000000000000000000000000000
force b 01111111111111111111111111111111
run 2
```



Configuration file (I/O mapping):

```
NET a(2) LOC = N5;
NET a(1) LOC = L4;
NET a(0) LOC = N2;
NET b(2) LOC = R9;
NET b(1) LOC = M3;
NET b(0) LOC = P1;
NET add_sub LOC = N3;
NET func(1) LOC = AF9;
NET func(0) LOC = AF8;
NET logic_func(1) LOC = AD11;
NET logic_func(0) LOC = AC11;
NET zero LOC = AC4;
NET overflow LOC = AA5;
NET result(7) LOC = P2;
NET result(6) LOC = R7;
NET result(5) LOC = P4;
NET result(4) LOC = T2;
NET result(3) LOC = R5;
NET result(2) LOC = R3;
NET result(1) LOC = V1;
NET result(0) LOC = T6;
```

Human explanation of mapping:

a(31,30,29):	red_switch(dip) 1,2,3
b(31,30,29):	red_switch(dip) 4,5,6
add_sub:	red_switch 8
func(1,0):	blue_switch 3,2
logic_func(1,0):	blue_switch 1,0
zero:	small_led 0
overflow:	small_led 3
output(31,30,29,28):	red_led(left to right)
output(3,2,1,0):	green_led(left to right)

NOTES:

switch up = logic 0
switch down = logic 1
LEDs are active LOW (logic 0 = ON)

Precision.log file:

```
# Info: [9566]: Logging session transcript to file /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/precision.log
//  Precision RTL Synthesis  64-bit 2016.1.0.15 (Production Release) Wed Jun  8 09:35:56 PDT 2016
//
//  Copyright (c) Mentor Graphics Corporation, 1996-2016, All Rights Reserved.
//            Portions copyright 1991-2008 Compuware Corporation
//                      UNPUBLISHED, LICENSED SOFTWARE.
//           CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//         PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
//
//  Running on Linux j_abba@grace.encs.concordia.ca #1 SMP Tue Sep 12 10:10:26 CDT 2017 3.10.0-693.2.2.el7.x86_64 x86_64
//
//  Start time Fri Oct  6 17:44:11 2017
# ------------------------------------------------------
# Info: [9566]: Logging session transcript to file /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/precision.log
# COMMAND: new_project -name 32bitalu -folder /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV -createimpl_name 32bitalu_impl_1
# Info: [9574]: Input directory: /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV
# Info: [9569]: Moving session transcript to file /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/precision.log
# Info: [9555]: Created project /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu.psp in folder /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV.
# Info: [9531]: Created directory: /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu_impl_1.
# Info: [9554]: Created implementation 32bitalu_impl_1 in project /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu.psp.
# Info: [9575]: The Results Directory has been set to: /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu_impl_1/
# Info: [9566]: Logging project transcript to file /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu_impl_1/precision.log
# Info: [9566]: Logging suppressed messages transcript to file /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu_impl_1/precision.log.suppressed
# Info: [9550]: Activated implementation 32bitalu_impl_1 in project /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu.psp.
new_project -name 32bitalu -folder /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV -createimpl_name 32bitalu_impl_1
# COMMAND: add_input_file {../Code/alu32bit.vhd}
add_input_file {../Code/alu32bit.vhd}
# COMMAND: add_input_file {../Code/alu1bitmostsignificant.vhd ../Code/alu1bit.vhd}
add_input_file {../Code/alu1bitmostsignificant.vhd ../Code/alu1bit.vhd}
# COMMAND: setup_design -manufacturer Xilinx -family "VIRTEX-II Pro" -part 2VP30ff896 -speed -7
# Info: [15298]: Setting up the design to use synthesis library "xcv2p.syn"
# Info: [575]: The global max fanout is currently set to 10000 for Xilinx - VIRTEX-II Pro.
# Info: [15324]: Setting Part to: "2VP30ff896".
# Info: [15325]: Setting Process to: "7".
# Info: [7512]: The place and route tool for current technology is ISE.
setup_design -manufacturer Xilinx -family "VIRTEX-II Pro" -part 2VP30ff896 -speed -7
# COMMAND: setup_design -frequency 100 -max_fanout=10000
# Info: [575]: The global max fanout is currently set to 10000 for Xilinx - VIRTEX-II Pro.
setup_design -frequency 100 -max_fanout=10000
# COMMAND: compile
# Info: [3022]: Reading file: /CMC/tools/mentor/precision/Mgc_home/pkgs/psr/techlibs/xcv2p.syn.
# Info: [634]: Loading library initialization file /CMC/tools/mentor/precision/Mgc_home/pkgs/psr/userware/xilinx_rename.tcl
# Info: XILINX
# Info: [40000]: vhdlorder, Release 2016a.7
# Info: [40000]: Files sorted successfully.
# Info: [40000]: hdl-analyze, Release RTLC-Precision 2016a.7
# Info: [42502]: Analyzing input file "/nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/../Code/alu1bit.vhd" ...
# Info: [42502]: Analyzing input file "/nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/../Code/alu1bitmostsignificant.vhd" ...
# Info: [42502]: Analyzing input file "/nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/../Code/alu32bit.vhd" ...
# Warning: [43642]: "/nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/../Code/alu32bit.vhd", line 183:
# Design unit LAB1 is already present in library work. Overwriting old definition of this design unit.
# Info: [659]: Top module of the design is set to: lab1.
# Info: [657]: Current working directory: /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu_impl_1.
# Info: [40000]: RTLC-Driver, Release RTLC-Precision 2016a.7
# Info: [40000]: Last compiled on Jun  2 2016 06:11:46
# Info: [44512]: Initializing...
# Info: [44504]: Partitioning design ....
# Info: [40000]: RTLCompiler, Release RTLC-Precision 2016a.7
# Info: [40000]: Last compiled on Jun  2 2016 06:47:43
# Info: [44512]: Initializing...
# Info: [44522]: Root Module work.alu32bit(lab1)(configuration lab1): Pre-processing...
# Info: [44506]: Module work.alu1bit(lab1): Pre-processing...
# Info: [44506]: Module work.alu1bitmostsignificant(lab1): Pre-processing...
# Info: [44508]: Module work.alu1bit(lab1): Compiling...
# Info: [44508]: Module work.alu1bitmostsignificant(lab1): Compiling...
# Info: [44523]: Root Module work.alu32bit(lab1)(configuration lab1): Compiling...
# Info: [44842]: Compilation successfully completed.
# Info: [44856]: Total lines of RTL compiled: 2558.
# Info: [44835]: Total CPU time for compilation: 0.0 secs.
# Info: [44513]: Overall running time for compilation: 1.0 secs.
# Info: [657]: Current working directory: /nfs/home/j/j_abba/Modelsim/LAB1/FPGA_ADV/32bitalu_impl_1.
# Info: [15330]: Doing rtl optimizations.
# Info: [660]: Finished compiling design.
compile
```

I got one warning during compilation which I believe is minor since it was my second time compiling the warning was about overwriting an older version of the ‘LAB1’ (name of my architecture) design unit.

Conclusion

In conclusion, in this lab I successfully implemented a single-cycle 32-bit ALU using structural VHDL following a well-established design of cascading 1-bit ALUs and adding the overflow/set functionality to the MSB 1-bit ALU.

The code was then compiled and simulated using Modelsim which allowed confirming its functionality and correctness. Having done that an RTL schematic was generated to obtain a physical representation of the entity and using the precision log we were able to identify possible mistakes and errors.

After mapping the inputs and outputs to test the design on the FPGA, I noticed some weird behavior with the unmapped bits. I would have expected them to default to 'pull down' or 'pull up' but they seemed to have unexpected behavior (perhaps this was just my board).

Finally, my design approach led me to a solid understanding of ALUs and to appreciate the expressiveness of VHDL (i.e. if I hadn't used structural). In the report I presented some arguments as to the advantages of my approach but certainly there are advantages to take the "easier" route.

Regardless, in the future it will be easy to instantiate my 32-bit ALU and use it as part of a larger design of a CPU.