Experiment 4: Combinational Circuit 4

Prateek Garg
20D070060
EE-214, WEL, IIT Bombay
September 22nd, 2021

Overview of the experiment:

Main Objectives:

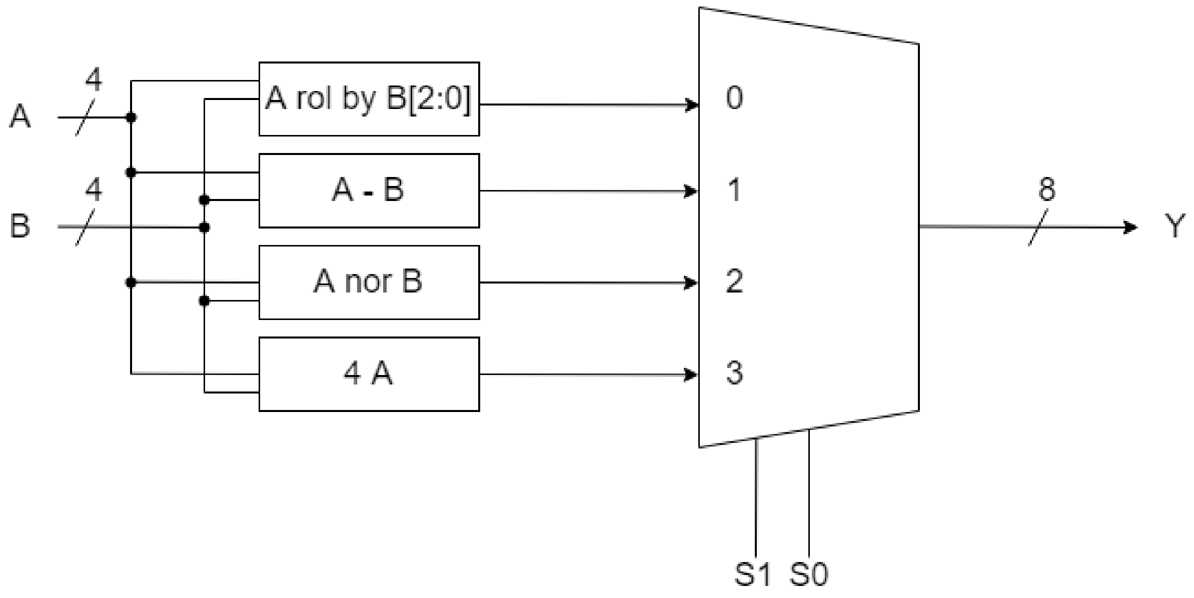
- Describe a given ALU circuit using VHDL which perform the following functions based on select lines:
    A. Rotate left A by B[2:0] number of bits
    B. Performs A-B Operation
    C. Performs A nor B Operation
    D. Produces output as 4*A

- Simulate the design using the generic testbench in ModelSim.
- Test the correctness of description using Scanchain.

The design was described in VHDL using the software Quartus Prime. Behavioral and Dataflow modelling for this experiment. The design was then simulated using ModelSim checked against every possible testcase. Then the design was simulated on Krypton board and checked against every possible test case using Scanchain.

Approach to the experiment:

Design:



We implement each of the function blocks separately and pass A and B to each block. We then select the output of a particular block to be passed to final output by using a 4x1 Multiplexer where select lines are mapped to the function as:

S1 S0	ALU Output
0 0	Rotate left A by B[2:0] number of bits
0 1	Performs A-B Operation
1 0	Performs A nor B Operation
1 1	Produces output as 4*A

We then implement the functions.

Rotate left A by B[2:0] number of bits:
We first extend 4 bit A to 8 bit value by adding zeroes and map ith bit of B to 2^i left shift in A. If a given bit of the three is 1 then we shift A by 2^i else we don't. Thus doing this in successive stages we achieve a shift value of B in decimal.

Performs A-B Operation:
We first extend 4 bit A and B to 8 bit values by adding zeroes. Then,we essentially invert the bits of B and calculate A+B' over 8 bit inputs keeping the track of carry in each successive iteration of loop.

Performs A nor B Operation:
To get a bitwise NOR we perform NOR operation over bits of A and B over a loop.

Produces output as 4*A:
We use the fact that multiplying by 4 in the binary the number would shift by 2 places to the left. So we use the already constructed function to rotate left by passing an Argument B as "0100"

Design document and VHDL code:

Architecture:

```
entity alu_beh is
    generic(
        operand_width : integer:=4;
        sel_line : integer:=2
        );
    port (
        A:    in std_logic_vector(operand_width-1 downto 0);
        B:    in std_logic_vector(operand_width-1 downto 0);
        sel: in std_logic_vector(sel_line-1 downto 0);
        op: out std_logic_vector((operand_width*2)-1 downto 0)
    ) ;
end alu_beh;

architecture a1 of alu_beh is

    function sub(
        A: in std_logic_vector(operand_width-1 downto 0);
        B: in std_logic_vector(operand_width-1 downto 0)
        ) return std_logic_vector is
            -- declaring and initializing variables using aggregates
        variable diff : std_logic_vector(operand_width*2-1 downto 0):=
(others=>'0');
        variable inv_B : std_logic_vector(operand_width-1 downto 0):=
(others=>'0');
        variable carry : std_logic:= '1';
        variable as : std_logic:= '1';
          begin
                invert: for i in 0 to operand_width-1 loop
                    inv_B(i) := B(i) xor as;
                end loop invert;

                add: for i in 0 to operand_width*2-1 loop
                    if i < operand_width then
                        diff(i) := carry xor (A(i) xor inv_B(i));
                        carry := (A(i) and inv_B(i)) or (carry and
inv_B(i)) or(A(i) and carry);
                    else
                        diff(i) := carry xor as;
                    end if;
                end loop add;
        return diff;
    end sub; --Performs A-B Operation:

    function rolf(
        A: in std_logic_vector(operand_width-1 downto 0);
        B: in std_logic_vector(operand_width-1 downto 0)
        ) return std_logic_vector is
        variable shift : std_logic_vector((operand_width*2)-1 downto
0):= (others=>'0');
        variable tmp : std_logic := '0';
          begin
            shift(operand_width-1 downto 0):= A;

            shift_op : for i in 0 to operand_width-2 loop
                if B(i) = '1' then
                    shift_i: for j in 0 to 2**i-1 loop
                        tmp :=shift(operand_width*2-1);
                        single_shift: for k in 0 to operand_width*2-2
loop
                                shift(operand_width*2-1-k) :=
shift(operand_width*2-2-k);
                        end loop single_shift;
                        shift(0) := tmp;
                    end loop shift_i;
                end if;
            end loop shift_op;
        return shift;
    end rolf; --Rotate left A by B[2:0] number of bits:

    function my_nor(
        A: in std_logic_vector(operand_width-1 downto 0);
        B: in std_logic_vector(operand_width-1 downto 0)
        ) return std_logic_vector is
        variable outp: std_logic_vector(operand_width-1 downto 0):=
(others=>'0');
        begin
            oper: for i in 0 to operand_width-1 loop
                outp(i):= A(i) nor B(i);
            end loop oper;

        return outp;
    end my_nor; --Performs A nor B Operation:
begin
alu : process( A, B, sel )
begin
    if sel = "00" then
     op <= rolf(A,B);
     elsif sel = "01" then
     op <= sub(A,B);
     elsif sel = "10" then
     op <= "0000"&my_nor(A,B);
     else
     op <=  rolf(A,"0010");
     end if;
end process ; --alu
end a1 ; -- a1
```

RTL View:

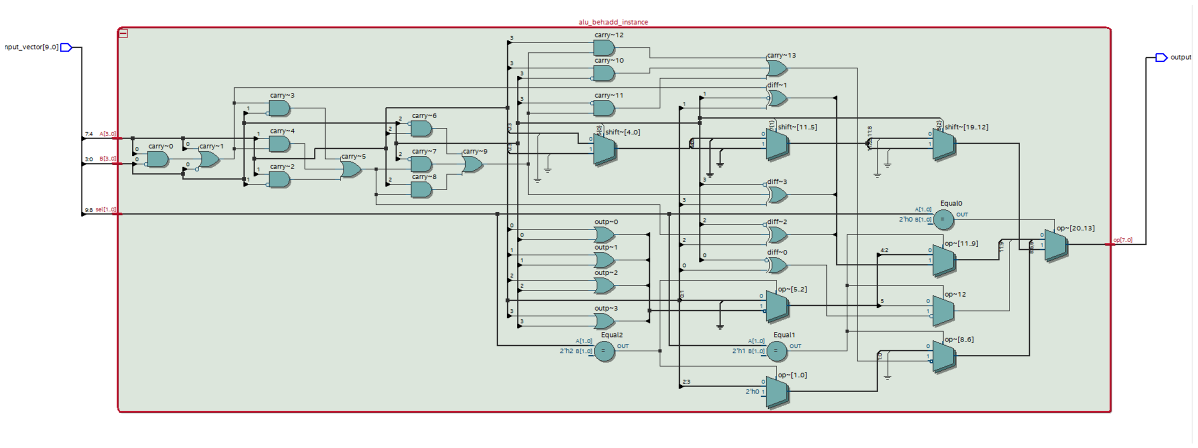


Fig: RTL View.

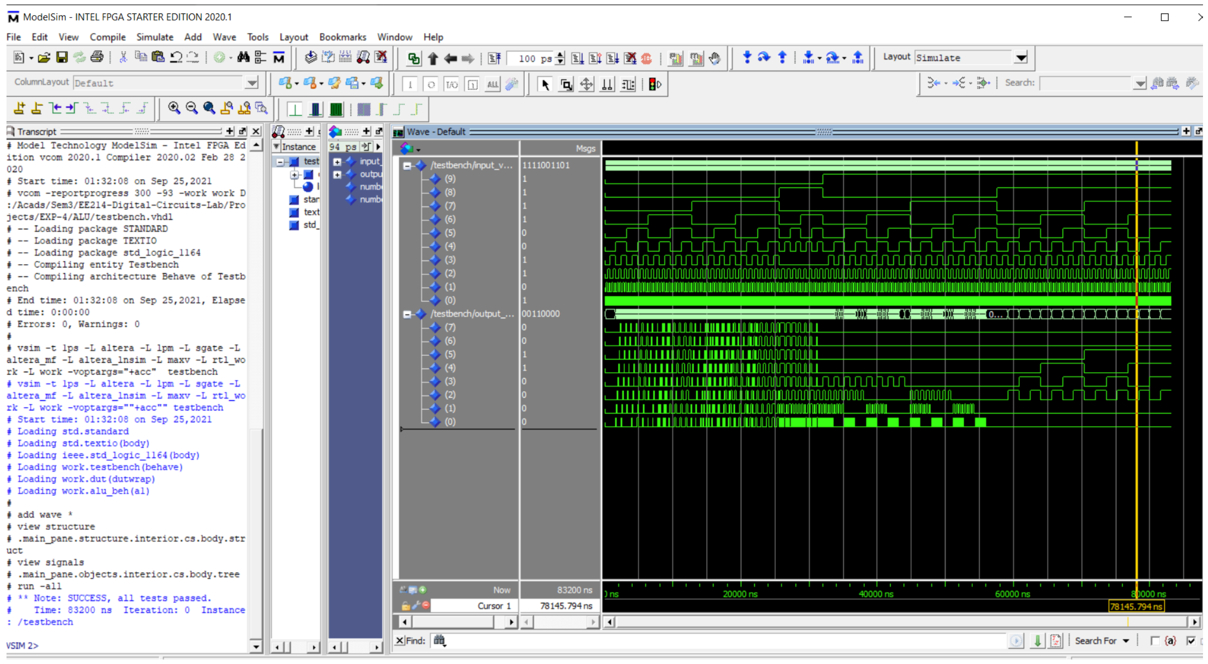
DUT Input/Output Format:

Port map:

```
        --input map
sel => input_vector( 9 downto 8),
A   => input_vector( 7 downto 4),
B   => input_vector( 3 downto 0),
        --output map
op  => output_vector
```

```
Some test cases from TRACEFILE:
TRACEFILE format:
 <S1 S0 A3 A2 A1 A0 B3 B2 B1 B0> <Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0> 11111111
                    1011111101 00000000 11111111
                    1011111110 00000000 11111111
                    1011111111 00000000 11111111
                    1100000000 00000000 11111111
                    1100000001 00000000 11111111
                    1100000010 00000000 11111111
                    1100000011 00000000 11111111
                    1100000100 00000000 11111111
                    1100000101 00000000 11111111
                    1100000110 00000000 11111111
                    1100000111 00000000 11111111
```

RTL Simulation:



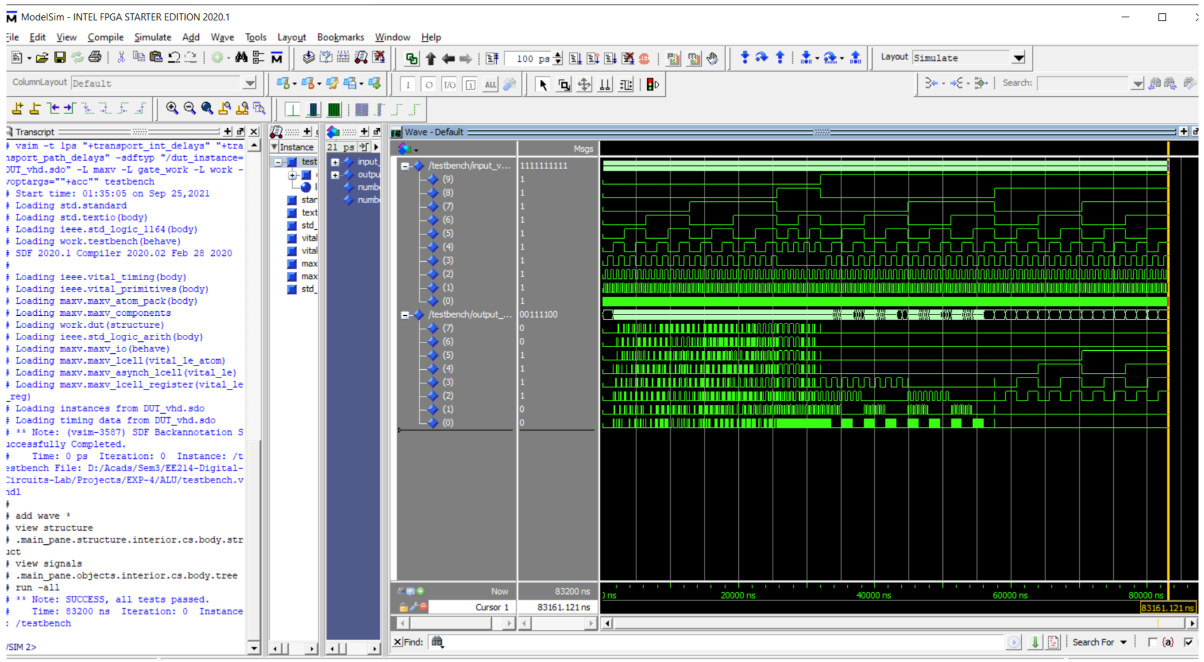
```
# Model Technology ModelSim - Intel FPGA Ed
ition vcom 2020.1 Compiler 2020.02 Feb 28 2
020
# Start time: 01:32:08 on Sep 25,2021
# vcom -reportprogress 300 -93 -work work D
:/Acads/Sem3/EE214-Digital-Circuits-Lab/Pro
jects/EXP-4/ALU/testbench.vhdl
# -- Loading package STANDARD
# -- Loading package TEXTIO
# -- Loading package std_logic_1164
# -- Compiling entity Testbench
# -- Compiling architecture Behave of Testb
ench
# End time: 01:32:08 on Sep 25,2021, Elapse
d time: 0:00:00
# Errors: 0, Warnings: 0
#
# vsim -t 1ps -L altera -L lpm -L sgate -L
altera_mf -L altera_lnsim -L maxv -L rtl_wo
rk -L work -voptargs="+acc"  testbench
# vsim -t 1ps -L altera -L lpm -L sgate -L
altera_mf -L altera_lnsim -L maxv -L rtl_wo
rk -L work -voptargs=""+acc"" testbench
# Start time: 01:32:08 on Sep 25,2021
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.testbench(behave)
# Loading work.dut(dutwrap)
# Loading work.alu_beh(al)
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.str
uct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ** Note: SUCCESS, all tests passed.
#    Time: 83200 ns  Iteration: 0  Instance
: /testbench

VSIM 2>
```

```
/testbench/input_v...   1111001101
  (9) 1
  (8) 1
  (7) 1
  (6) 1
  (5) 0
  (4) 0
  (3) 1
  (2) 1
  (1) 0
  (0) 1
/testbench/output_...   00110000
  (7) 0
  (6) 0
  (5) 1
  (4) 1
  (3) 0
  (2) 0
  (1) 0
  (0) 0
Now       83200 ns
Cursor 1  78145.794 ns
```

Fig: RTL Simulation

Gate-level Simulation:



```
# vsim -t 1ps "+transport_int_delays" "+tra
nsport_path_delays" -sdftyp "/dut_instance=
DUT_vhd.sdo" -L maxv -L gate_work -L work -
voptargs=""+acc"" testbench
# Start time: 01:35:05 on Sep 25,2021
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading work.testbench(behave)
# SDF 2020.1 Compiler 2020.02 Feb 28 2020
#
# Loading ieee.vital_timing(body)
# Loading ieee.vital_primitives(body)
# Loading maxv.maxv_atom_pack(body)
# Loading maxv.maxv_components
# Loading work.dut(structure)
# Loading ieee.std_logic_arith(body)
# Loading maxv.maxv_io(behave)
# Loading maxv.maxv_lcell(vital_le_atom)
# Loading maxv.maxv_asynch_lcell(vital_le)
# Loading maxv.maxv_lcell_register(vital_le
_reg)
# Loading instances from DUT_vhd.sdo
# Loading timing data from DUT_vhd.sdo
# ** Note: (vsim-3587) SDF Backannotation S
uccessfully Completed.
#    Time: 0 ps  Iteration: 0  Instance: /t
estbench File: D:/Acads/Sem3/EE214-Digital-
Circuits-Lab/Projects/EXP-4/ALU/testbench.v
hdl
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.str
uct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ** Note: SUCCESS, all tests passed.
#    Time: 83200 ns  Iteration: 0  Instance
: /testbench

VSIM 2>
```

```
/testbench/input_v...   1111111111
  (9) 1
  (8) 1
  (7) 1
  (6) 1
  (5) 1
  (4) 1
  (3) 1
  (2) 1
  (1) 1
  (0) 1
/testbench/output_...   00111100
  (7) 0
  (6) 0
  (5) 1
  (4) 1
  (3) 1
  (2) 1
  (1) 0
  (0) 0
Now       83200 ns
Cursor 1  83161.121 ns
```

Fig: Gate Level Simulation

Observation:

Parts of the file out.txt generated by scanchain:

```
0000000000 00000000 Success
0000000001 00000000 Success
0000000010 00000000 Success
0000000011 00000000 Success
0000000100 00000000 Success
0000000101 00000000 Success
0000000110 00000000 Success
0000000111 00000000 Success
0000001000 00000000 Success
0000001001 00000000 Success
0000001010 00000000 Success
0000001011 00000000 Success
0000001100 00000000 Success
0000001101 00000000 Success
0000001110 00000000 Success
0000001111 00000000 Success
0000010000 00000001 Success
```

```
0000011111 10000000 Success
0000100000 00000010 Success
0000100001 00000100 Success
0000100010 00001000 Success
0000100011 00010000 Success
0000100100 00100000 Success
0000100101 01000000 Success
0000100110 10000000 Success
0000100111 00000001 Success
0000101000 00000010 Success
0000101001 00000100 Success
0000101010 00001000 Success
0000101011 00010000 Success
0000101100 00100000 Success
0000101101 01000000 Success
0000101110 10000000 Success
0000101111 00000001 Success
```

References:

1. J.F.Wakerly: Digital Design, Principles and Practices,4th Edition,Pearson Education, 2005