City University of Hong Kong

Department of Electronic Engineering

EE 2000 – Lab Manual 2

A 4-bit Full-Adder

Course Leader: CHIN Lip Ket

Objectives:

- Learn the modular design flow.
- Implement a **1-bit** Full-Adder using VHDL.
- Implement a **4-bit** Full-Adder using VHDL.

There are 5 checkpoints in total.

For each checkpoint, please take notes/photos/screenshots for your report.

Please show Checkpoints 1, 2 and 4 to the lab tutor or demonstrator for marking.

Experiment: 4-bit full-adder implementation

In this experiment, we first implement a 1-bit full adder, then a 4-bit full adder will be built on it in a modular way.

1. Implement a 1-bit full adder

i. Create the VHDL source file "full_adder_1bit.vhd" to a new project. You should write the code to generate o_S and o_Cout, i.e. sum and carry output of 1-bit full adder. We always keep the source file name same as the entity name. In VHDL, all the input/output ports should be defined in ENTITY. For example, in this program, the input ports are i_A, i_B and i_Cin and the output ports are o_S and o_Cout. The truth table of 1-bit full adder is

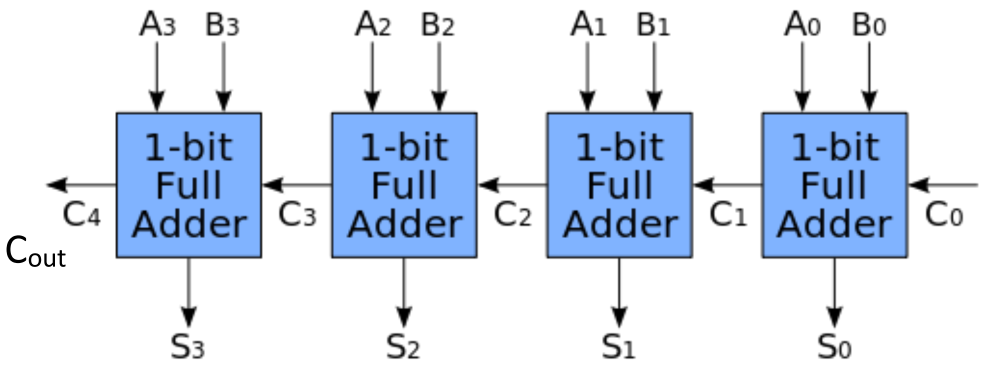
Input			**Output**	
i_Cin	**i_A**	**i_B**	**o_S**	**o_Cout**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Please try to figure out logic relations between input and output and write your code with the following template to generate a 1-bit full adder.

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

Entity full_adder_1bit is

   Port (
      i_A: in STD_LOGIC;
      i_B: in STD_LOGIC;
      i_Cin: in STD_LOGIC;
      o_S: out STD_LOGIC;
      o_Cout: out STD_LOGIC);
End full_adder_1bit;

Architecture Behavioral of full_adder_1bit is
Begin
   --Add Your own code here
End Behavioral;
```

2. Implement a 4-bit full adder

Here is the basic structure of a 4-bit full adder. We can see the 4-bit full adder consists of four 1-bit full adder units. Thus, in our implementation, we need to instantiate four full_adder_1bit units.



i. **Create** the VHDL source file full_adder_4bits.vhd for the same project.

ii. In this part, the previous 1-bit full adder is used and multiple instantiations are rearranged as a 4-bit full adder.

Try to complete the following code to implement a 4-bit full adder. Before we write the code, we should have a clear view of the connection between different modules.

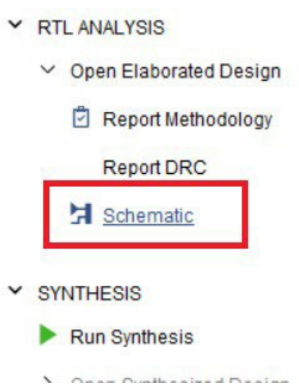
```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

Entity full_adder_4bits is

   Port (
      i_A: in STD_LOGIC_VECTOR (3 DOWNTO 0);
      i_B: in STD_LOGIC_VECTOR (3 DOWNTO 0);
      i_Ci: in STD_LOGIC;
      o_S: out STD_LOGIC_VECTOR (3 DOWNTO 0);
      o_Cout: out STD_LOGIC
      );
end full_adder_4bits;

Architecture Behavioral of full_adder_4bits is
   SIGNAL ci: STD_LOGIC_VECTOR (3 DOWNTO 0);

   COMPONENT full_adder_1bit is
      Port (
        i_A: in STD_LOGIC;
        i_B: in STD_LOGIC;
        i_Cin: in STD_LOGIC;
        o_S: out STD_LOGIC;
        o_Cout: out STD_LOGIC
        );
   END COMPONENT;
begin

ci(0) <= i_Ci;
uut0: full_adder_1bit PORT MAP(i_A(0), i_B(0), ci(0), o_S(0), ci(1));
   --Write your own code here to instantiate other three units.

end Behavioral;
```

(Remarks: full_adder_1bit is the Component Entity and uut0 is a name of the component that you entitled.)

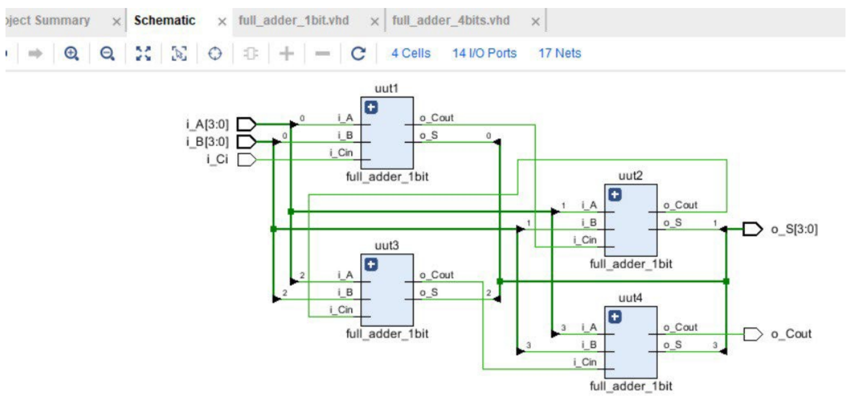
Checkpoint 1: implement "full_adder_1bit" and "full_adder_4bits" and check there are no syntax errors.

3. Perform RTL analysis on the source file

i. Expand the **Open Elaborated Design** entry under the **RTL Analysis** from the **Flow Navigator** panel and click on **Schematic**. And then click **OK**.



Your design will be elaborated and a logic view of the design is displayed as the following. We can check the design has four full_adder_1bit and they are correctly connected.



4. Simulate the Design using the XSim Simulator (Xilinx built-in simulator)

i. Create a Test Bench full_adder_tb.vhd to your project.

Your task is to add your own code to generate the desired waveform as follows.

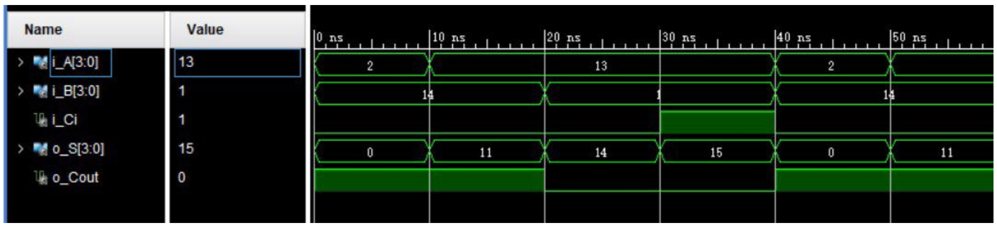
Initially: i_A and i_B are "0010" and "1110", respectively. i_Ci is '0'.

At time 10ns: change i_A to "1101".
At time 20ns: change i_B to "0001".
At time 30ns: change i_Ci to '1'.

Here is the Test Bench template for you:

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

Entity full_adder_tb is
end full_adder_tb;

Architecture Behavioral of full_adder_tb is
    COMPONENT full_adder_4bits is
        Port (
                    i_A: in STD_LOGIC_VECTOR (3 DOWNTO0);
                    i_B: in STD_LOGIC_VECTOR (3 DOWNTO0);
                    i_Ci: in STD_LOGIC;
                    o_S: out STD_LOGIC_VECTOR (3 DOWNTO 0);
                    o_Cout: out STD_LOGIC );
                    signal i_A: std_logic_vector (3 downto 0);
                    signal i_B: std_logic_vector (3 downto 0);
                    signal i_Ci: std_logic;
                    signal o_S: std_logic_vector (3 downto 0);
                    signal o_Cout: std_logic;
begin
    uut: full_adder_4bits PORT MAP (i_A, i_B, i_Ci, o_S, o_Cout);
    siggen: PROCESS
    begin

    --Add your code here

    end process siggen;
end Behavioral;
```

Checkpoint 2: Generate your simulated waveform screen as below.



Name	Value
i_A[3:0]	13
i_B[3:0]	1
i_Ci	1
o_S[3:0]	15
o_Cout	0

Checkpoint 3: Generate a simulated waveform to add the first digit of your student ID with the second digit of the ID for the first 10 ns. Then add the third digit with the fourth digit for the next 10 ns. Finally, add the fifth and sixth digits for the next 10 ns (add a carry in this case).

5. I/O constraints

As introduced in Lab 1, we can add I/O constraints by creating the .xdc file.

To assign the I/O pin to different variables, we should check the **Reference guide of Basys 3 Board** to know the user I/O resources and how the peripheral connects with FPGA. In this experiment, we will use the on-board resources: switches, buttons, and LEDs.

The pins of the FPGA chip connected with the above peripheral are shown below. You may search the information below from the Google search engine. (Hints: Keywords – "Basys3 constraint file")

Table 12 – Button Switch Connections	
Signal Name	**Pin Assignment**
Center Button	**U18**
Upper Button	**T18**
Left Button	**W19**
Right Button	**T17**
Down Button	**U17**

Table 13 – DIP Switch Connections	
Signal Name	**Pin Assignment**
SW0	**V17**
SW1	**V16**
SW2	**W16**
SW3	**W17**
SW4	**W15**
SW5	**V15**
SW6	**W14**
SW7	**W13**

Table 14 - LED Connections	
Signal Name	**Pin Assignment**
LD0	**U16**
LD1	**E19**
LD2	**U19**
LD3	**V19**
LD4	**W18**
LD5	**U15**
LD6	**U14**
LD7	**V14**
LD8	**V13**
LD9	**V3**

In this experiment, we use SW0-SW3 as input i_A, SW4-SW7 as input i_B, push button btnC as input i_Ci, LD0-LD3 as o_S, and LD4 as o_Cout.

Basys 3 provides a .xdc file to help us perform I/O constraints when assigning pins. You can download **Basys3 Master XDC** from CANVAS and modify the file. For example, we want to use SW0-SW3 as input i_A. We can go to line 12 and change from

```
# Switches
set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
        set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
#set_property PACKAGE_PIN W16 [get_ports {sw[2]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {sw[2]}]
#set_property PACKAGE_PIN W17 [get_ports {sw[3]}]
        #set_property IOSTANDARD LVCMOS33 [get_ports {sw[3]}]
```

to

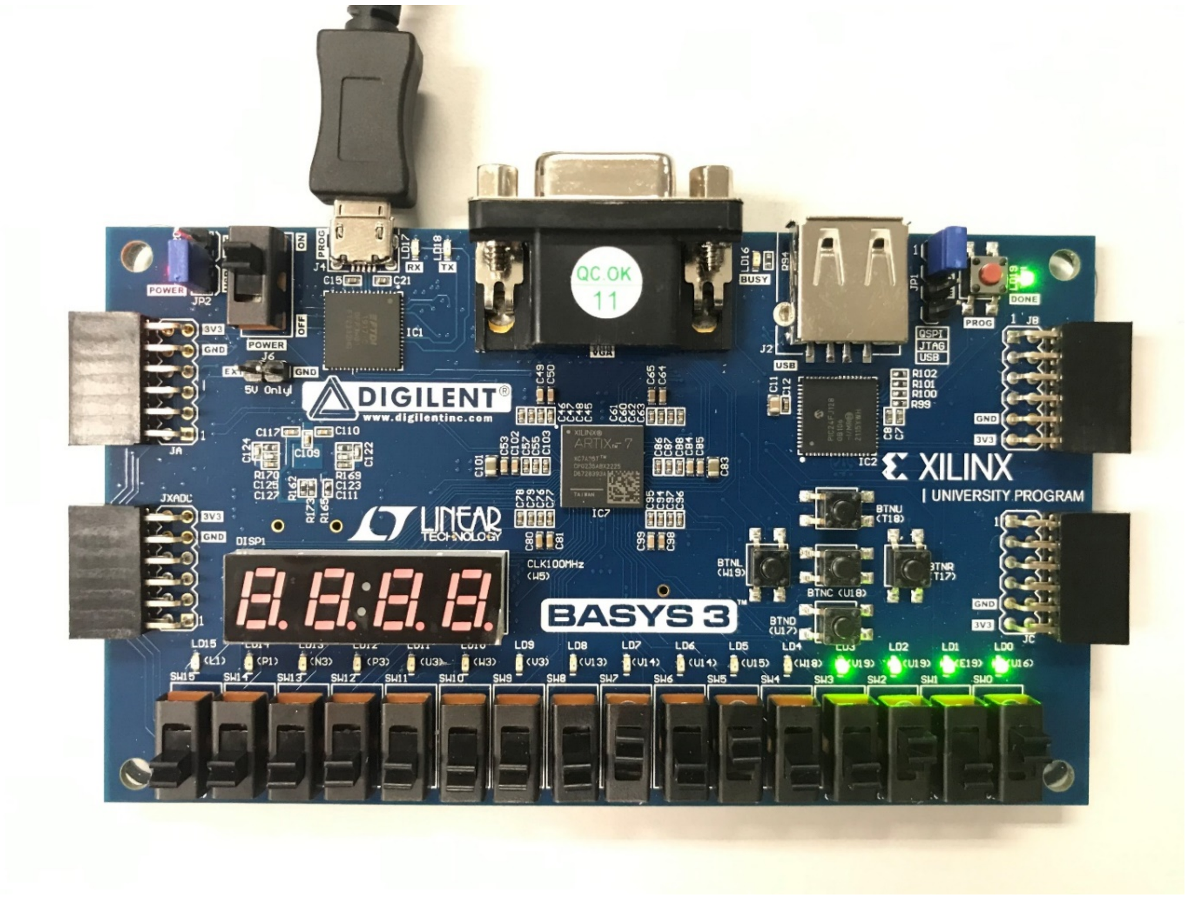
```
# Switches
set_property PACKAGE_PIN V17 [get_ports { i_A[0] }]
        set_property IOSTANDARD LVCMOS33 [get_ports { i_A[0] }]
set_property PACKAGE_PIN V16 [get_ports { i_A[1] }]
        set_property IOSTANDARD LVCMOS33 [get_ports { i_A[1] }]
set_property PACKAGE_PIN W16 [get_ports { i_A[2] }]
        set_property IOSTANDARD LVCMOS33 [get_ports { i_A[2] }]
set_property PACKAGE_PIN W17 [get_ports { i_A[3] }]
        set_property IOSTANDARD LVCMOS33 [get_ports { i_A[3] }]
```

Similarly, edit the port names for SW4-SW7, LD0-LD4 and BTNL.

6. **Synthesize, implement the design and generate a bitstream file.**

i. **Run synthesis, implementation and generate bitstream** as in Lab 1.

ii. Then, we **program** the Basys3 board by transferring the bitstream file to it. You can verify the design by switching on/off of switches on the board. Finally, we can observe the result indicated by the LEDs, i.e. LD0-LD4.



Checkpoint 4: Show your result to your demonstrator. LEDs should be illuminated correctly correspond to different input cases.

Checkpoint 5: Discuss in the lab report how to modify the VHDL source file to implement a 4-bit Ripple Borrow Subtractor.

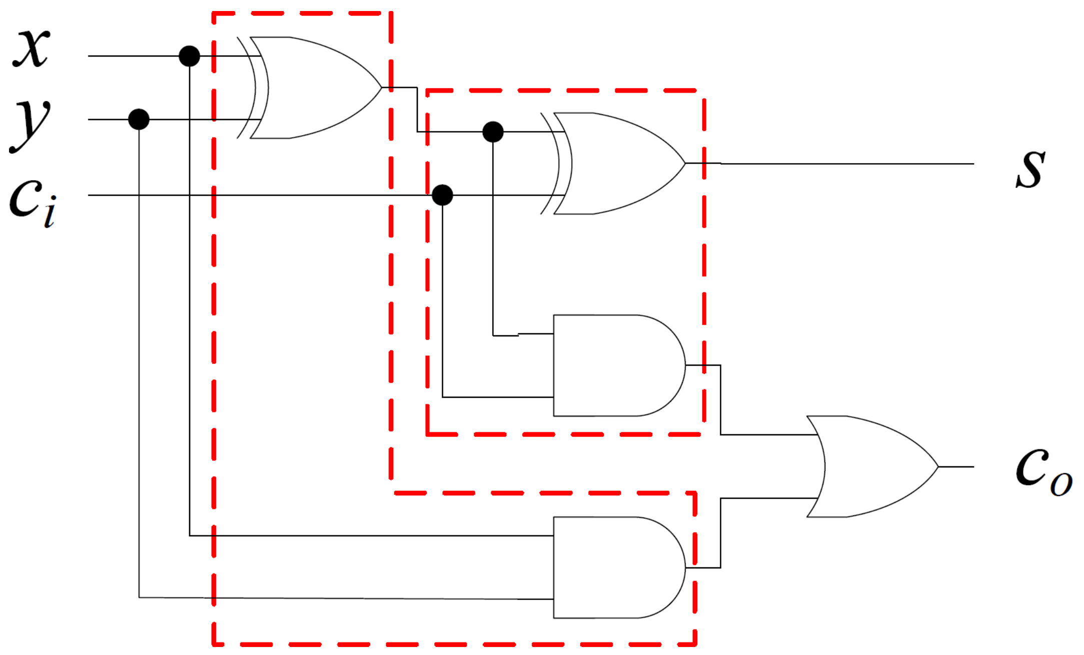
Supplementary materials:

1-bit Full-Adder (FA):

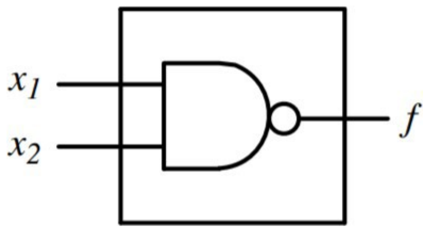
$x\ y\ c_i$	$s\quad c_o$
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

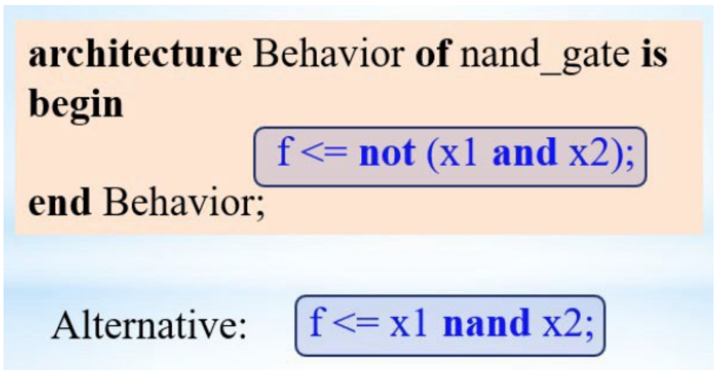
$$\begin{aligned} s &= x\,yc_i + xyc_i + xy\,c_i + xyc_i \\ &= x(\,yc_i + yc_i\,) + x(yc_i + y\,c_i\,) \\ &= x(\,y \oplus c_i\,) + x(\,y \oplus c_i\,) \\ &= x \oplus y \oplus c_i \end{aligned}$$

$$\begin{aligned} c_o &= yc_i + xc_i + xy \\ &= xy + x(\,y + y)c_i + y(x + x)c_i \\ &= xy + xyc_i + xyc_i + xyc_i \\ &= xy(1 + c_i) + (xy + xy)c_i \\ &= xy + (x \oplus y)c_i \end{aligned}$$



Basic Logic Gates using VHDL:





```
architecture Behavior of nand_gate is
begin
        f <= not (x1 and x2);
end Behavior;
```

Alternative: `f <= x1 nand x2;`