City University of Hong Kong

Department of Electronic Engineering

EE 2000 – Lab Manual 2

A 4-bit Full-Adder

Course Leader: Dr. WONG Steve Hang

Objectives:

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

There are four checkpoints in Page 4, 6 and 8 For each checkpoint, please raise your hand and demonstrate to the course tutor.

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 code to generate o_S and o_Cout, i.e. sum and carry out of 1-bit full adder. As a convention, 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

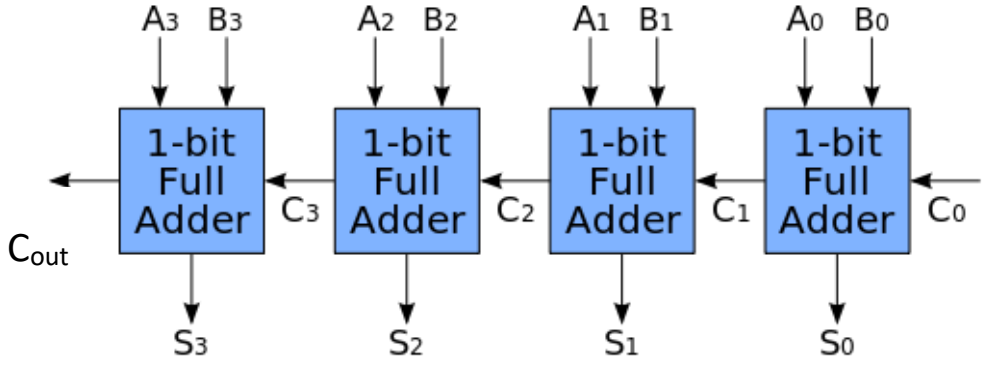
Try to figure out logic relations between input and output and write your own code with following template to generate a 1-bit full adder.

```
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 a 4-bit full adder consists four 1-bit full adder. So in our implementation, we need to instantiate four full_adder_1bit units.



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

ii. In this part, 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.

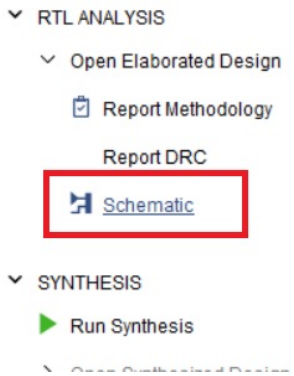
```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

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;
```

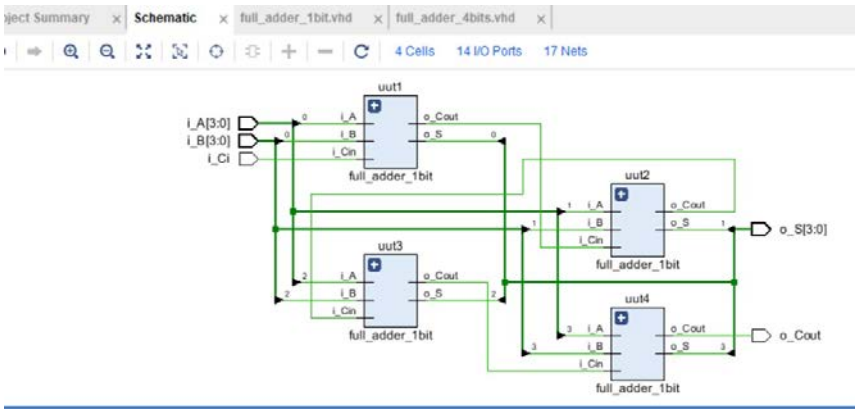
Checkpoint 1: implement “full_adder_1bit” and “full_adder_4bits” and check there is no syntax errors.

3. **Perform RTL analysis on the source file**

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



Your design will be elaborated and a logic view of the design is displayed like 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.**Add** a testbench full_adder_tb.vhd to the project.

Here we provide a testbench template and your task is to add your own code to generate desired waveform

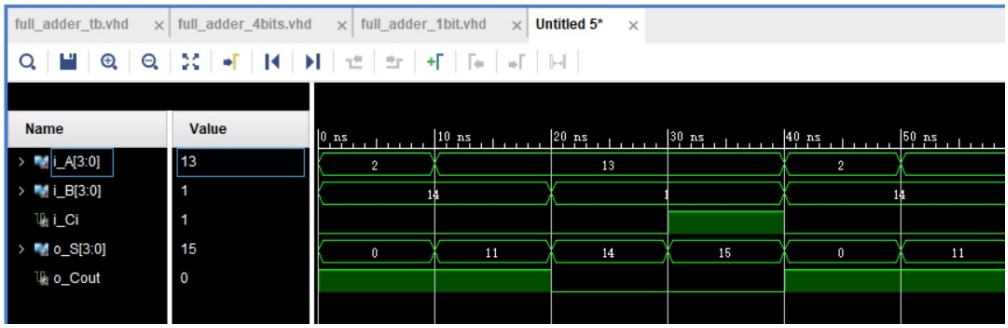
Initially: i_A and i_B are “0010” and “1110”, respectively. i_Ci is’0’ At time 10ns: change i_A to “1101”. At time 20ns: change i_B to “0001”. At time 30ns: change i_Ci to ‘1’.

This is a 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 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 COMPONENT;
   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;
```

ii.Run Simulation as Lab 1. We can get the waveform as follows.



Checkpoint 2: Generate your simulated waveform screen as above.

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

5. I/O constraints

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

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

The pins of FPGA chip connected with the above peripheral are shown as below. This information is included in Section 2.7 of "ZedBoard Hardware User's Guide" on the website http://zedboard.org/support/documentation/1521 .

Table 12 - Push Button Connections

Signal Name	Subsection	Zynq pin
BTNU	PL	T18
BTNR	PL	R18
BTND	PL	R16
BTNC	PL	P16
BTNL	PL	N15
PB1	PS	D13 (MIO 50)
PB2	PS	C10 (MIO 51)

Table 13 - DIP Switch Connections

Signal Name	Zynq pin
SW0	F22
SW1	G22
SW2	H22
SW3	F21
SW4	H19
SW5	H18
SW6	H17
SW7	M15

Table 14 - LED Connections

Signal Name	Subsection	Zynq pin
LD0	PL	T22
LD1	PL	T21
LD2	PL	U22
LD3	PL	U21
LD4	PL	V22
LD5	PL	W22
LD6	PL	U19
LD7	PL	U14
LD9	PS	D5 (MIO7)

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

To accelerate the pin assignment, ZedBoard provide the xdc file to help user perform

I/O constraints. We can go to website http://zedboard.org/support/documentation/1521 and download **ZedBoard Master XDC Rev C/D v3**. We just need to modify it a bit. For example, we want to use SW0-SW3 as input I_a. We can go to line 237 and modify

```
set_property PACKAGE_PIN F22 [get_ports {SW0}];  # "SW0"
set_property PACKAGE_PIN G22 [get_ports {SW1}];  # "SW1"
set_property PACKAGE_PIN H22 [get_ports {SW2}];  # "SW2"
set_property PACKAGE_PIN F21 [get_ports {SW3}];  # "SW3"
```

to

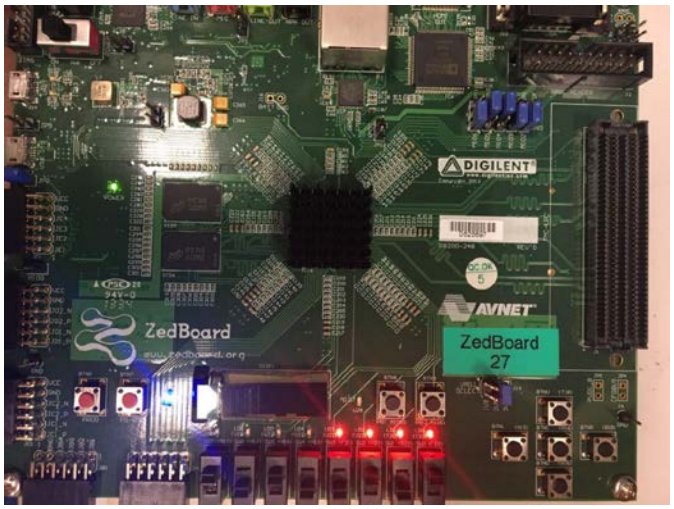
```
set_property PACKAGE_PIN F22 [get_ports { i_A[0] }];
set_property PACKAGE_PIN G22 [get_ports { i_A[1] }];
set_property PACKAGE_PIN H22 [get_ports { i_A[2] }];
set_property PACKAGE_PIN F21 [get_ports { i_A[3] }];
```

Similarly modify the port name for SW4-SW7, LD0-LD4 and BTNL. We don't need to modify the IOSTANDARD.

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

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

ii. Then, we **program** the bitstream file to the ZedBoard. After that, you can verify the design by setting different input. By switching SW4-SW7 and SW0-SW3, we can change the input operand A and B respectively. Then we can observe the result indicated by the LEDs, i.e. LD0-LD4.



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

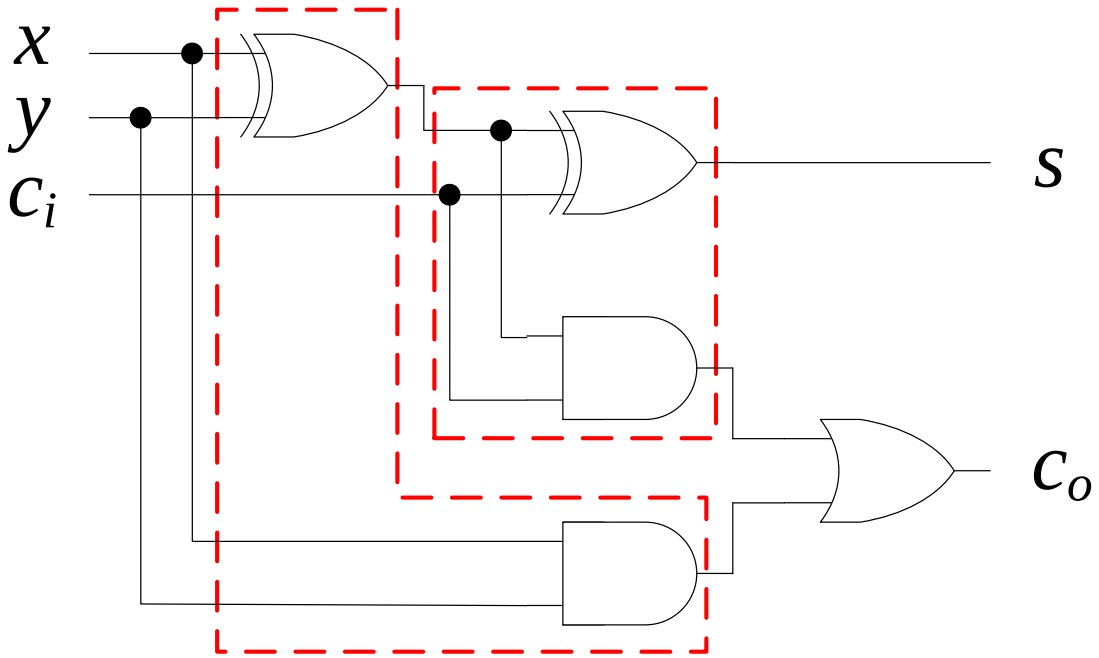
Supplementary materials:

1-bit Full-Adder (FA):

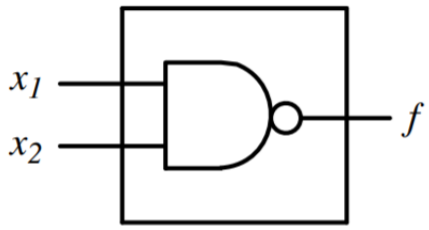
x y c_i	s c_o
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

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

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



Basic Logic Gates using VHDL:




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

Alternative: `f <= x1 nand x2;`