EXPERIMENT 6

Binary Adders

OBJECTIVES:

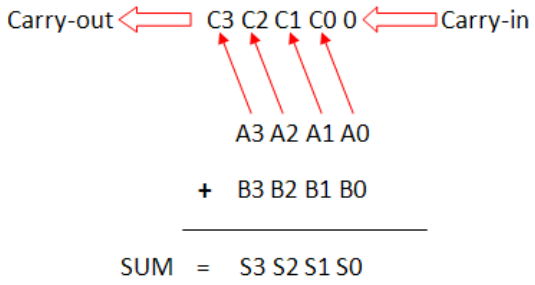
- Design a 1-bit full adder based on its truth table.
- Demonstrate modular design and hierarchy.
- Use Xilinx simulation tools to test combinational circuits.

MATERIALS:

- Xilinx Vivado software, student or professional edition V2018.2 or higher.
- IBM or compatible computer with Pentium III or higher, 128 M-byte RAM or more, and 8 G-byte Or larger hard drive.
- BASYS 3 Board.

DISCUSSION:

Addition and subtraction are two essential arithmetic functions performed by computers and other digital systems. It is therefore important to understand how to design a circuit to perform such functions. However, since subtraction is done by adding the 2s complement of a number, we will only need to design one adder circuit to perform both operations. An adder can be 1 or more bits. A 4-bit adder can add two 4-bit unsigned binary numbers. If larger binary numbers are to be added, an adder with more bits is needed. Let's observe what happens when adding two 4-bit binary numbers with pencil and paper:



The symbols [A3 A2 Al A0] and [B3 B2 B1 B0] represent addend and minuend, respectively. C0 is the carry bit generated by adding bits A0 and B0. C1 is the carry bit generated from the addition of C0, Al, and B1. C2 and C3 are generated in the same manner, with C3 the carry-out. The column containing A0 and B0 (the least significant bits of addend and minuend) allows for a carry-in from a previous addition, for this example we set it to 0. Each column adds three bits. The implementation of the above process in hardware called a full adder.

When we perform this addition, we will start from the least significant bit, and then push the process left one bit at a time. This means that a 1-bit full adder is the basic element of a 4-bit adder and four such elements are needed to construct a 4-bit adder.

The 1-Bit Full Adder

From the discussion above we know that a 1-bit full adder should have three inputs: carry input (Cin), addend (A), and minuend (B). We can determine the number of output bits by looking at any column in the addition process, say, the column containing C0, A1 and B1. Assume all three bits are 1. Then the result is 3 which, in binary, are 11. The sum requires two bits but S1 can be only one bit, so there must be a carry to the next column. Each column will produce a sum bit and a carry output to the next more significant bit position. So the circuit for the 1-bit full adder should have two outputs: sum bit (S) and carry output (Cout). Table 7.1 shows the truth table for the 1-bit full adder:

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Using Boolean algebra, we can derive the following two equations for the sum bit and the carry output bit:

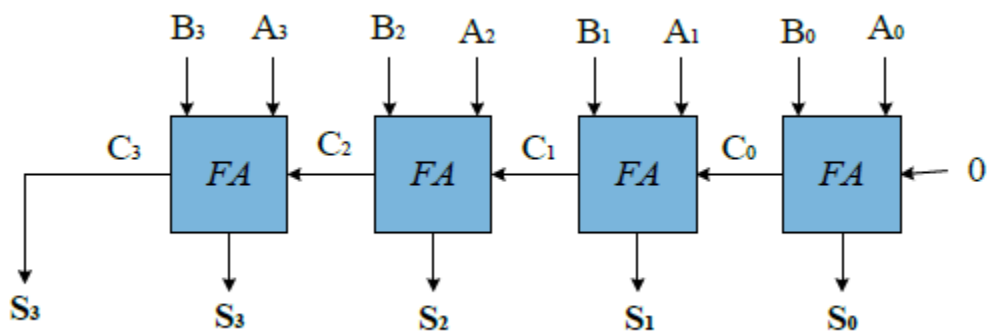
$$S = A \oplus B \oplus C$$

$$C_{out} = C_{in}(A + B) + AB$$

The above two equations can be implemented using a 3-input XOR gate, two 2-input AND gates, and two 2-input OR gates.

The 4-Bit Adder

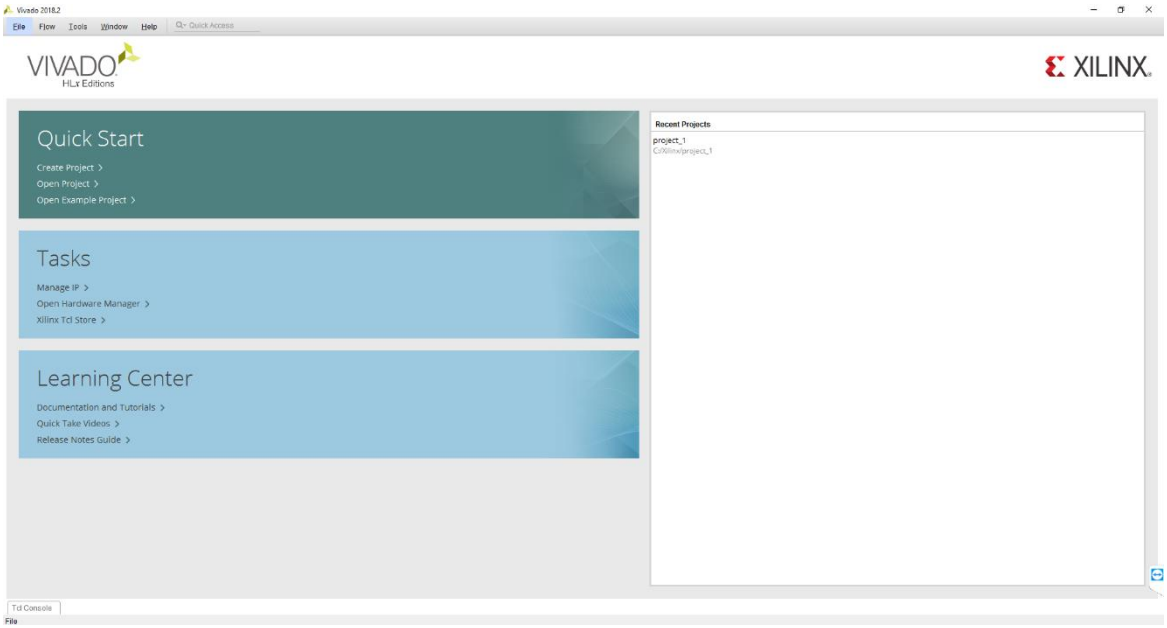
Once we have the 1-bit full-adder (FA), we can use it as a building block in any design that needs to do addition, such as the multi-bit adder in a CPU. In a multi-bit adder, the carry-in of the least significant bit (LSB) must be connected to 0 since there is no previous stage. The carry output from the LSB stage should feed into the second least significant bit. The carry output of second least significant stage feeds into the next more significant stage as carry input, and so on. The last carry output is the most significant bit of the sum. The 4-bit adder block diagram with interconnections between the FA modules is shown in figure below:



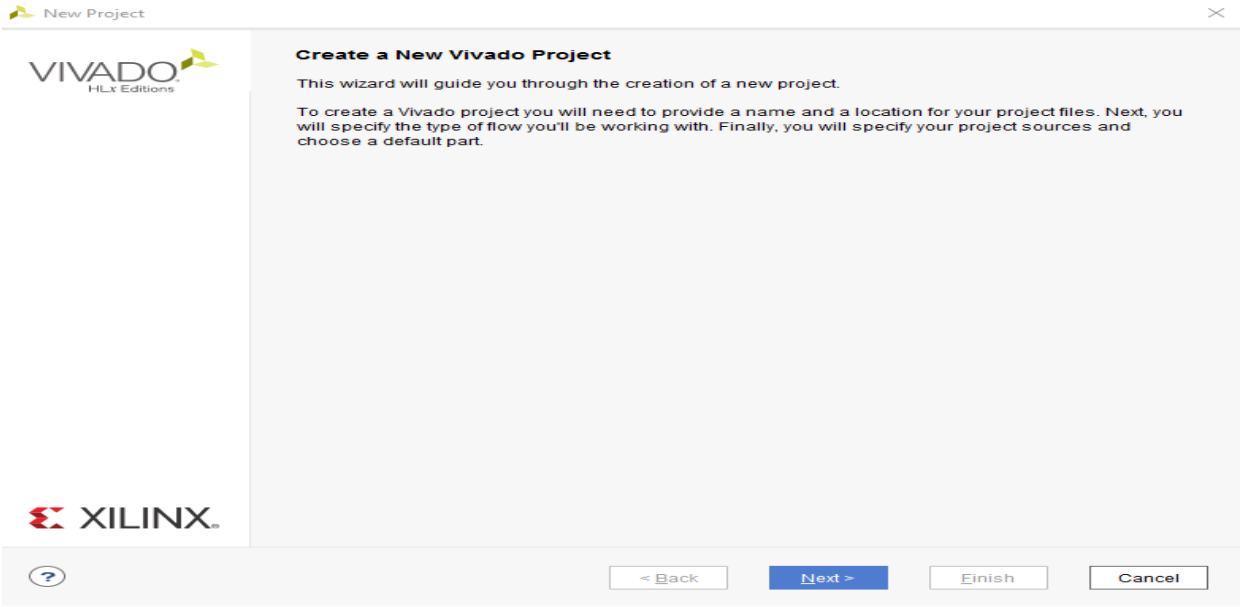
PROCEDURE:

Section I. The 1-Bit Full Adder

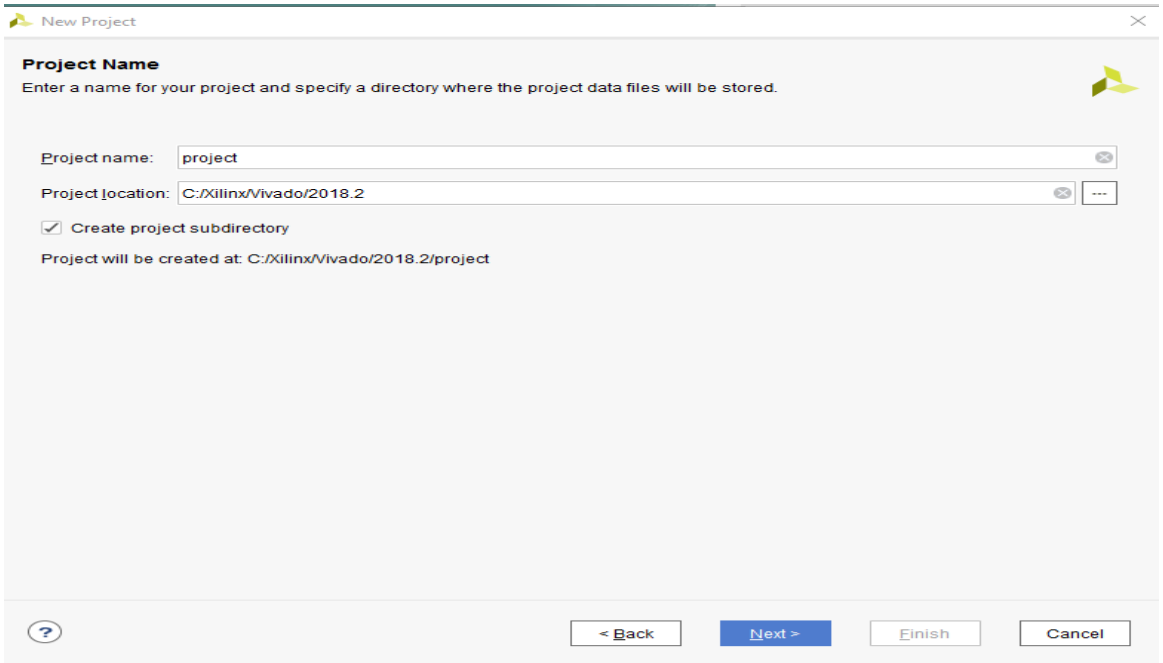
1. Open Xilinix Vivado.



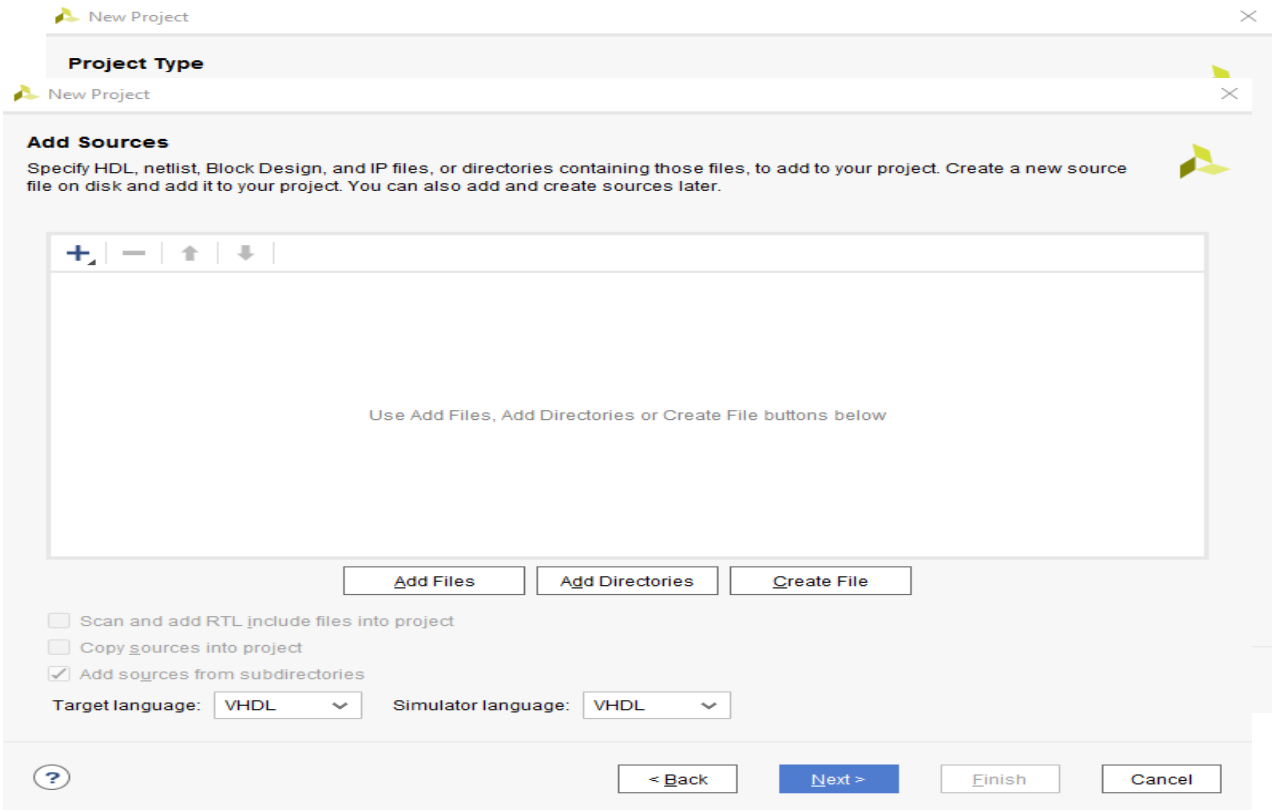
2. In the **Xilinx-Project Navigator** window, Quick start, **New Project**.



3. Name the project.



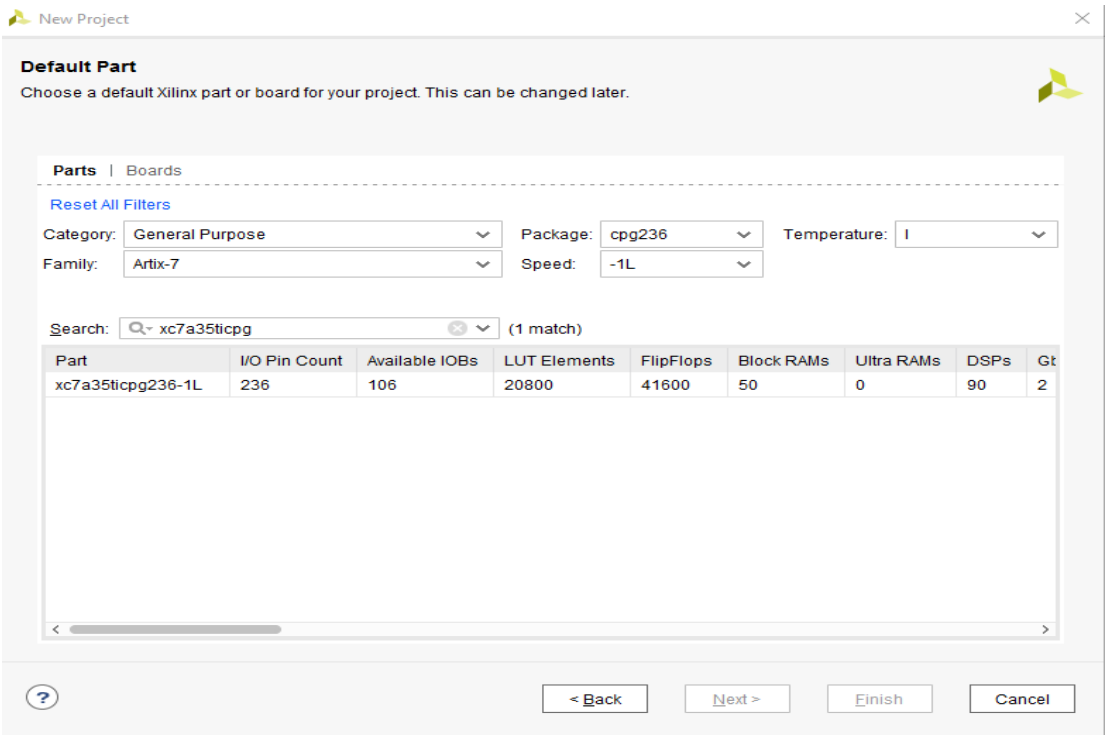
4. Choose "RTL Project" and check the "Do not specify sources at this time" as we will configure all the settings manually through the navigator from inside the project.
5. Select **New Source…** and the **New** window appears. In the New window, choose Schematic, type your file name (such as ***source_1***) in the File Name editor box, click

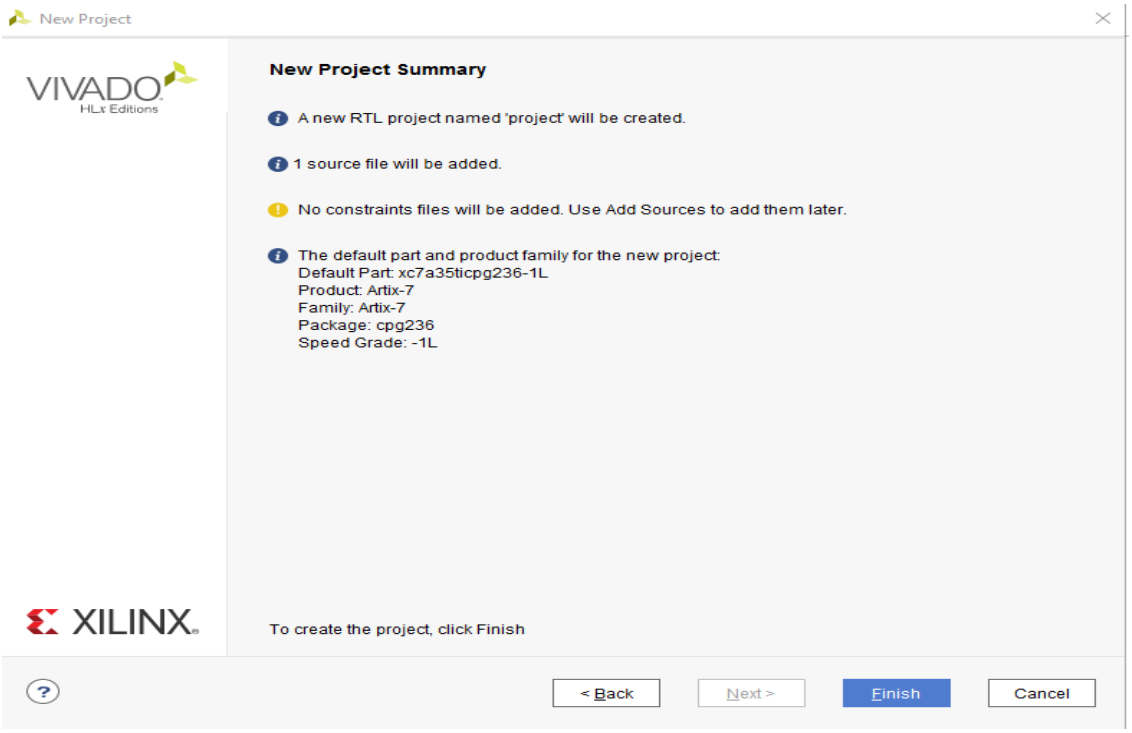


on OK, and then click on the Next button.

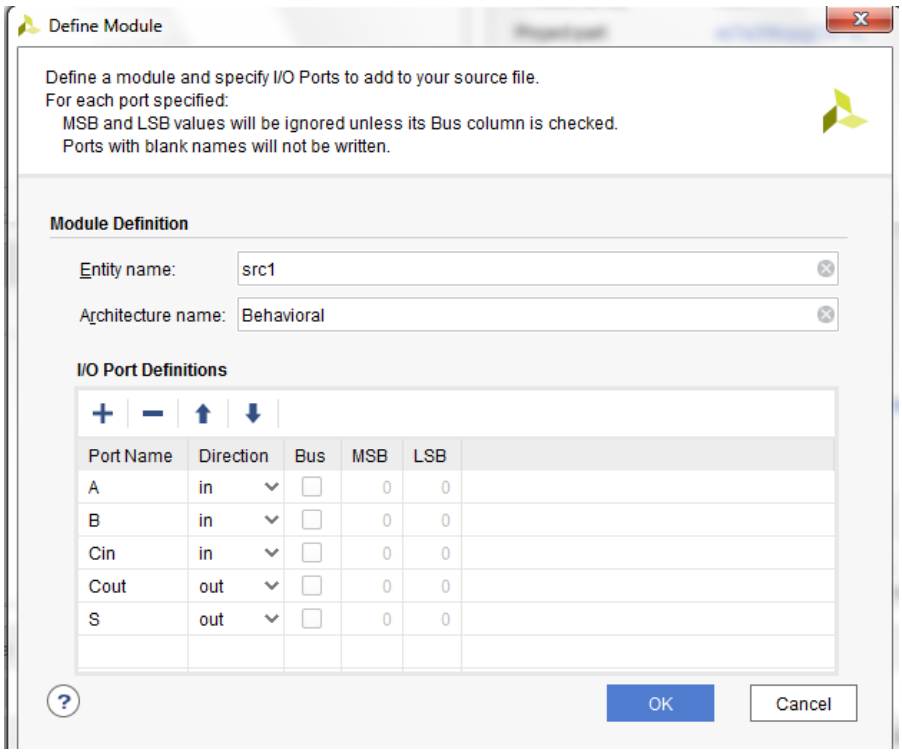
6. In the **Xilinx - Project Navigator** window, select the following
   - Category: "General Purpose"
   - Family: "Artix-7"
   - Package: "cpg236"
   - Speed: "-1"
   - Choose "xc7a35tcpg236-1" that corresponds to the board we are using.

Then Choose Finish.





7. The Define Module Window that will appear, we will choose the input and output labels for the gates under investigation in this experiment. In this experiment, we are investigating De Morgan's Theorem and we use 4 inputs to get 2 outputs. Under "Port Name", add "A", "B", and "Cin" as inputs and add "S", "Cout" as outputs and select OK.



8. In the "source_1.vhd" created file, type the gates equivalent VHDL code for the S and Cout between the "begin" and "end Behavioral" as follows and then save the file.

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity full_adder_vhdl_code is
5   Port ( A : in STD_LOGIC;
6   B : in STD_LOGIC;
7   Cin : in STD_LOGIC;
8   S : out STD_LOGIC;
9   Cout : out STD_LOGIC);
10  end full_adder_vhdl_code;
11
12  architecture gate_level of full_adder_vhdl_code is
13
14  begin
15
16  S <= A XOR B XOR Cin ;
17  Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B) ;
18
19  end gate_level;
```

9. Next, we need to add a constraint file with the".xdc" extension, as following:
Go to "Flow Navigator" and from "Project Manager" select "Add Sources" then "Add or create constraints". Next, choose "Create File" and enter the file name "lab_6" then "OK" followed by "Finish".

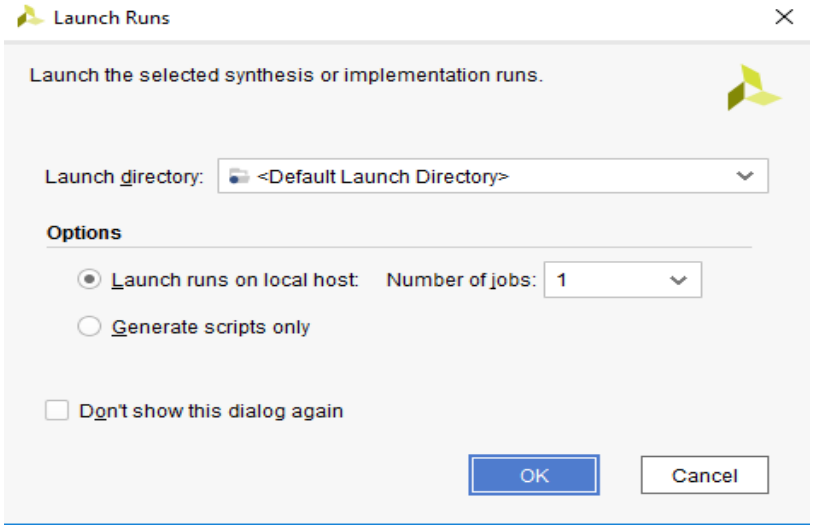
10. Then, we need to get a template xdc file that is going to be edited according to the different experiments. Google "basys 3 xdc file" and choose the "xilinix" link that appears (https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2015x/Basys3/Supporting%20Material/Basys3_Master.xdc). Copy the whole file and paste it into the "lab_2.xdc" that you have just created in the last step. Then uncomment and edit the input Switches and the output LEDs as in the next step.

11. Uncomment (by deleting the # sign) the ones you are going to use as following:

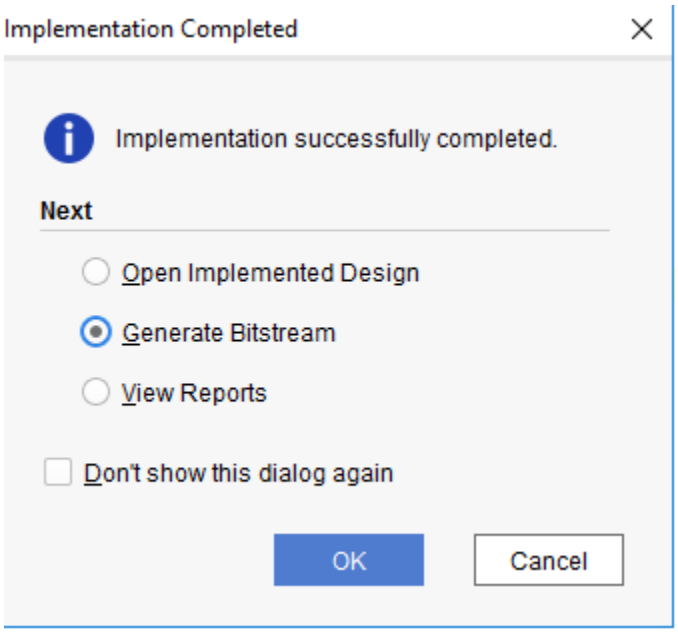
```
## Switches
set_property PACKAGE_PIN V17 [get_ports {A}]
    set_property IOSTANDARD LVCMOS33 [get_ports {a}]
set_property PACKAGE_PIN V16 [get_ports {B}]
    set_property IOSTANDARD LVCMOS33 [get_ports {B}]
set_property PACKAGE_PIN W16 [get_ports {Cin}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Cin}]
#set_property PACKAGE_PIN W17 [get_ports {sw[3]}]
```

```
## LEDs
set_property PACKAGE_PIN U16 [get_ports {S}]
    set_property IOSTANDARD LVCMOS33 [get_ports {S}]
set_property PACKAGE_PIN E19 [get_ports {Cout}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Cout}]
#set_property PACKAGE_PIN U19 [get_ports {led[2]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {led[2]}]
#set_property PACKAGE_PIN V19 [get_ports {led[3]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {led[3]}]
#set_property PACKAGE_PIN W18 [get_ports {led[4]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {led[4]}]
```

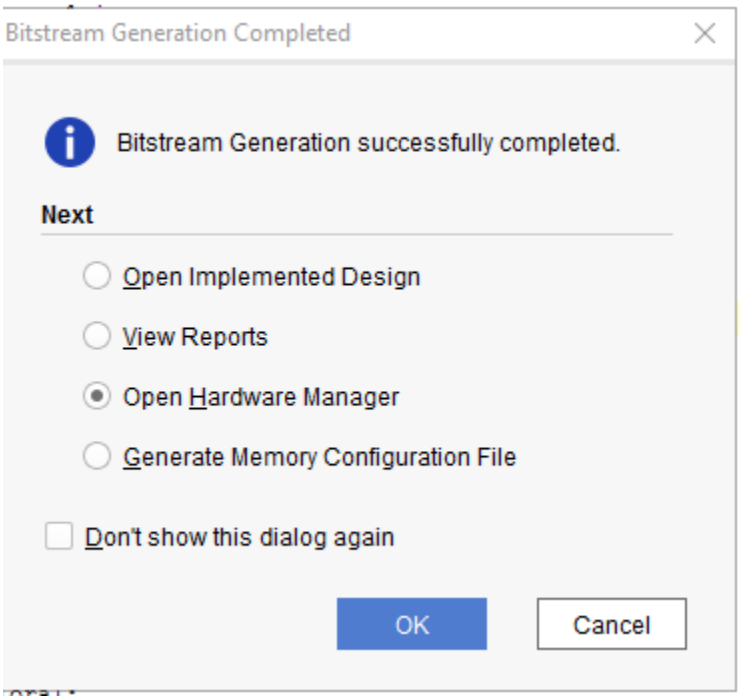
12. From the tool tab choose the play button and then "Run Implementation". Select "Number of jobs" =1 and then press OK.



13. The implementation errors window will appear if any or the successfully completed window. From this window select "Generate Bitstream" and then OK. This will make the software generate ".bin" file to be used in programing the hardware BAYAS 3.

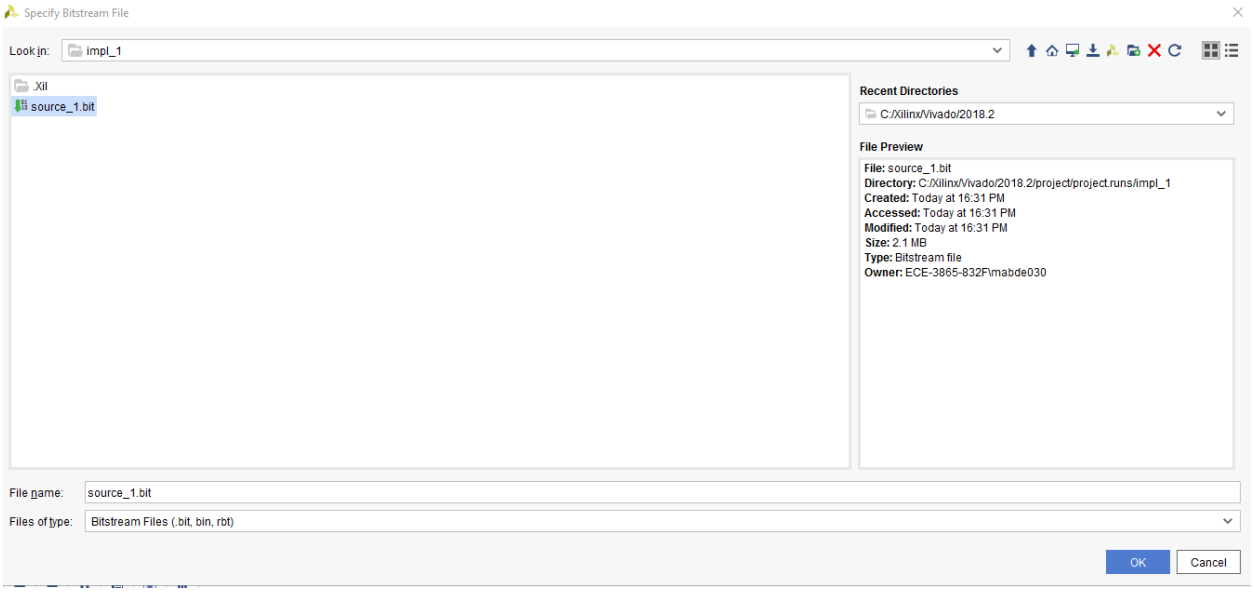


14. The next window will appear in which choose "Open Hardware Manger", connect the Hardware Kit to the USB port and then press OK.

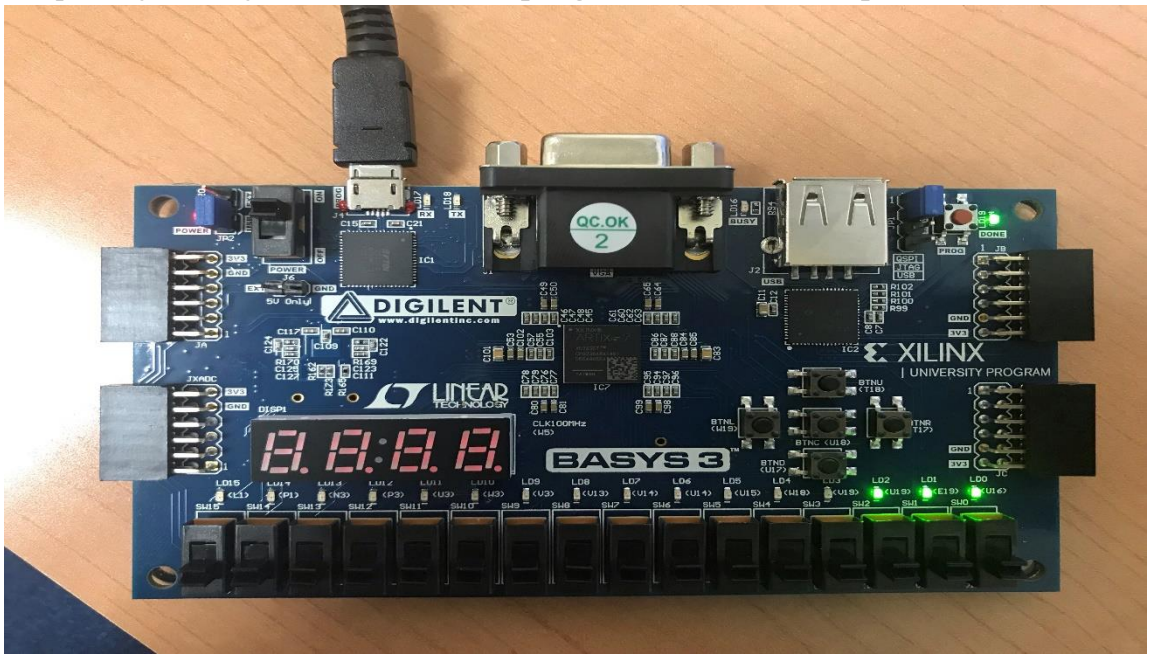


15. A green tab will appear in the top of the Vivado window, from which choose "open target" to program the hardware.

16. From the window appears, select the ".bin" file from the Project you create by browsing for the generated ".bit file" under the ".runs" folder and program the board then press OK.



17. Notice that the 7-segment on the hardware is counting up from 0 to 9 frequently until you download the program and it will stop.



18. Test the program on your board by going through all the input combinations and observing the two outputs. Fill the truth table.

Inputs			Outputs	
A	B	Cin	Cout	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

19. Are the two output the same? If they are, you have proved the Boolean distributive law. If not, figure it out.

20. Then you can use the simulation tools to verify the Boolean distributive law. For simulation, we need to create a simulation source file as following:

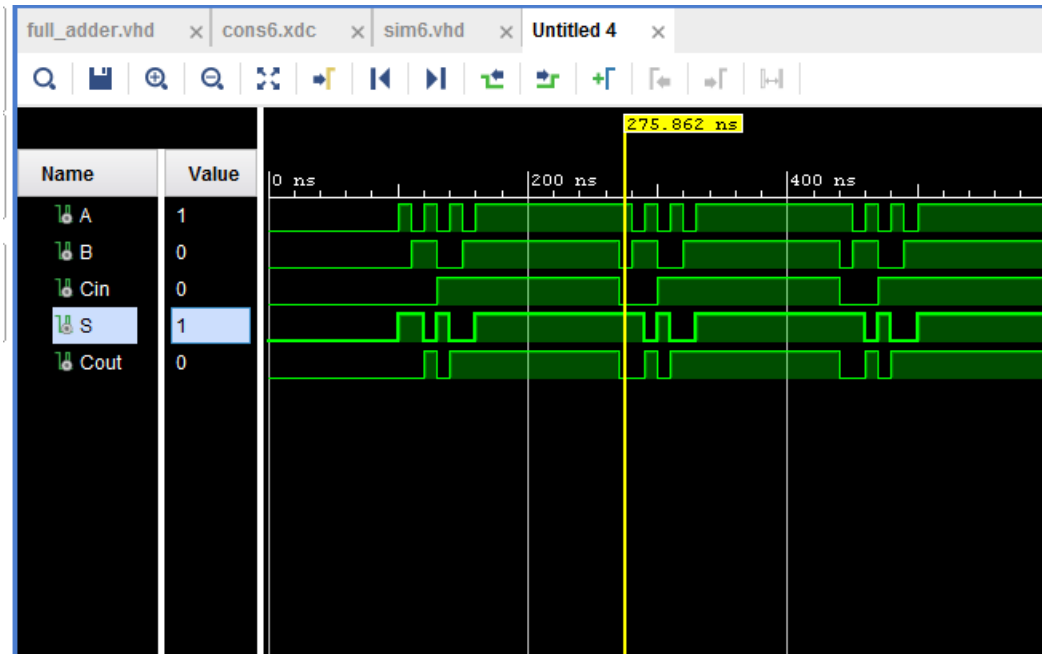
21. "Flow Navigator" ➔ "Project Manager" ➔ "Add Sources" ➔ "Add or create simulation sources" ➔ Name it "TB" (Test Bench) ➔ "VHDL" ➔ No need for switches and leds assignments as we will not be working on board. ➔ "OK".

22. After that, implement your simulation as similarly:

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Testbench_full_adder IS
END Testbench_full_adder;

ARCHITECTURE behavior OF Testbench_full_adder IS

 -- Component Declaration for the Unit Under Test (UUT)

 COMPONENT full_adder_vhdl_code
 PORT(
 A : IN std_logic;
 B : IN std_logic;
 Cin : IN std_logic;
 S : OUT std_logic;
 Cout : OUT std_logic
 );
 END COMPONENT;

 --Inputs
 signal A : std_logic := '0';
 signal B : std_logic := '0';
 signal Cin : std_logic := '0';

 --Outputs
 signal S : std_logic;
 signal Cout : std_logic;

BEGIN

 -- Instantiate the Unit Under Test (UUT)
 uut: full_adder_vhdl_code PORT MAP (
 A => A,
 B => B,
 Cin => Cin,
 S => S,
 Cout => Cout
 );

 -- Stimulus process
 stim_proc: process
 begin
 -- hold reset state for 100 ns.
 wait for 100 ns;

 -- insert stimulus here
 A <= '1';
 B <= '0';
 Cin <= '0';
 wait for 10 ns;

 A <= '0';
 B <= '1';
 Cin <= '0';
 wait for 10 ns;

 A <= '1';
 B <= '1';
 Cin <= '1';
 wait for 10 ns;

 end process;

END;
```

23. In the "initialization" section change the simulation variables according to your needs.

24. You should see similar output:



Section II. Building a 4-bit Adder Using Full-Adder (FA) Modules

In this part of the experiment, we will show how to perform modular design by building the 4-bit adder using four 1-bit full adder modules. We will start with making a symbol for a 1-bit full adder and add it as a module to the project library.

1. Create a new source file called four_bit_adder under the same project. Write the following code:

full_adder.vhd | cons6.xdc | sim6.vhd | four_bit_adder.vhd | four_bit_a

C:/Users/aayde001/project_6.2/project_6.2.srcs/sources_1/new/four_bit_adder.vhd

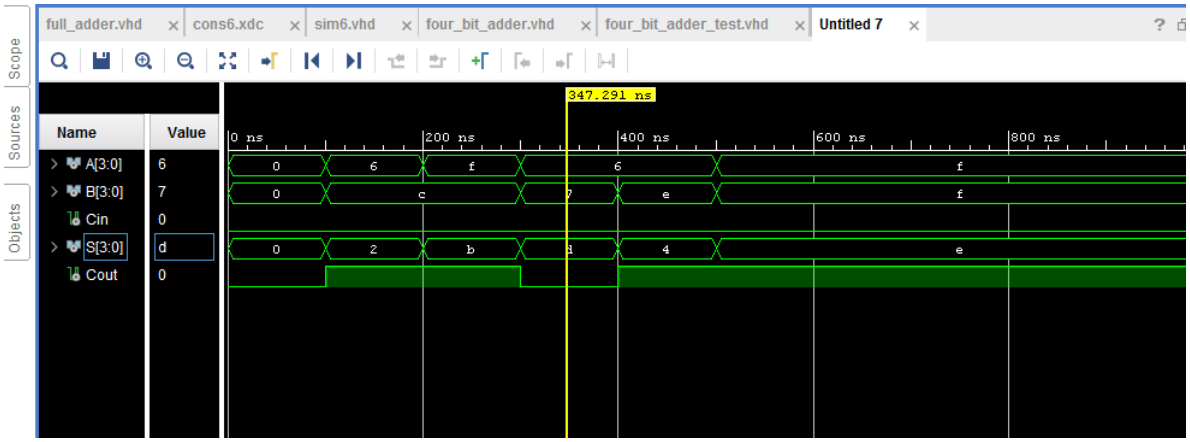
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Ripple_Adder is
Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
B : in STD_LOGIC_VECTOR (3 downto 0);
Cin : in STD_LOGIC;
S : out STD_LOGIC_VECTOR (3 downto 0);
Cout : out STD_LOGIC);
end Ripple_Adder;

architecture Behavioral of Ripple_Adder is

-- Full Adder VHDL Code Component Decalaration
component full_adder_vhdl_code
Port ( A : in STD_LOGIC;
B : in STD_LOGIC;
Cin : in STD_LOGIC;
S : out STD_LOGIC;
Cout : out STD_LOGIC);
end component;

-- Intermediate Carry declaration
signal c1,c2,c3: STD_LOGIC;

begin

-- Port Mapping Full Adder 4 times
FA1: full_adder_vhdl_code port map( A(0), B(0), Cin, S(0), c1);
FA2: full_adder_vhdl_code port map( A(1), B(1), c1, S(1), c2);
FA3: full_adder_vhdl_code port map( A(2), B(2), c2, S(2), c3);
FA4: full_adder_vhdl_code port map( A(3), B(3), c3, S(3), Cout);

end Behavioral;
```

2. Then create the simulation as you did before and write the following code for simulation.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Tb_Ripple_Adder IS
END Tb_Ripple_Adder;

ARCHITECTURE behavior OF Tb_Ripple_Adder IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT Ripple_Adder
PORT(
A : IN std_logic_vector(3 downto 0);
B : IN std_logic_vector(3 downto 0);
Cin : IN std_logic;
S : OUT std_logic_vector(3 downto 0);
Cout : OUT std_logic
);
END COMPONENT;

--Inputs
signal A : std_logic_vector(3 downto 0) := (others => '0');
signal B : std_logic_vector(3 downto 0) := (others => '0');
signal Cin : std_logic := '0';

--Outputs
signal S : std_logic_vector(3 downto 0);
signal Cout : std_logic;

BEGIN

-- Instantiate the Unit Under Test (UUT)
uut: Ripple_Adder PORT MAP (
A => A,
B => B,
Cin => Cin,
S => S,
Cout => Cout
);
```

```
40
41      -- Stimulus process
42      stim_proc: process
43      begin
44      -- hold reset state for 100 ns.
45      wait for 100 ns;
46      A <= "0110";
47      B <= "1100";
48
49      wait for 100 ns;
50      A <= "1111";
51      B <= "1100";
52
53      wait for 100 ns;
54      A <= "0110";
55      B <= "0111";
56
57      wait for 100 ns;
58      A <= "0110";
59      B <= "1110";
60
61      wait for 100 ns;
62      A <= "1111";
63      B <= "1111";
64
65      wait;
66
67      end process;
68
69      END;
70
```

3. Example output of simulation:



4. Fill the table accordingly by changing your initialization part of the code:

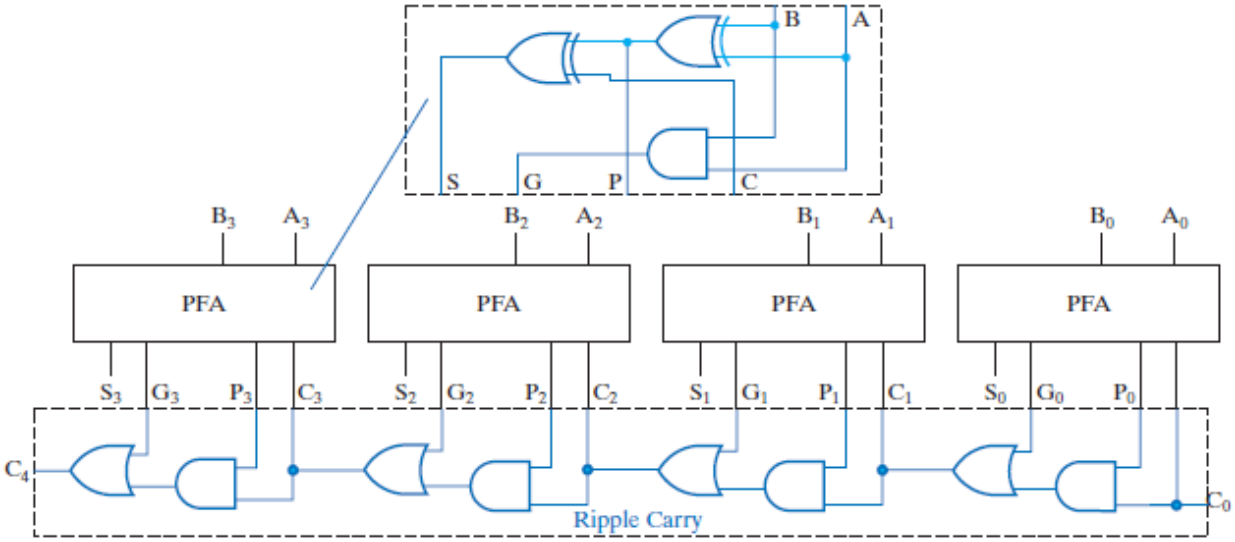
Switches(Input)										LEDs(Output)					
1	2	3	4	Decimal	5	6	7	8	Decimal	1	2	3	4	5	Decimal
A_3	A_2	A_1	A_0		B_3	B_2	B_1	B_0		C_{out}	S_3	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0						
0	0	0	1	1	0	0	0	0	0						
0	0	1	0	2	0	0	0	1	1						
0	0	1	1	3	0	0	1	0	2						
0	1	0	0	4	0	0	1	1	3						
0	1	0	1	5	0	0	1	1	3						
0	1	1	1	7	0	0	1	0	2						
1	0	0	0	8	0	0	1	0	2						
0	1	0	0	4	1	0	0	1	9						
0	1	0	1	5	1	1	0	0	12						
0	1	1	1	7	1	1	1	0	14						
1	0	1	0	10	0	1	1	1	7						
1	1	0	0	12	1	0	1	1	11						
1	1	1	1	15	1	1	1	1	15						

QUESTIONS

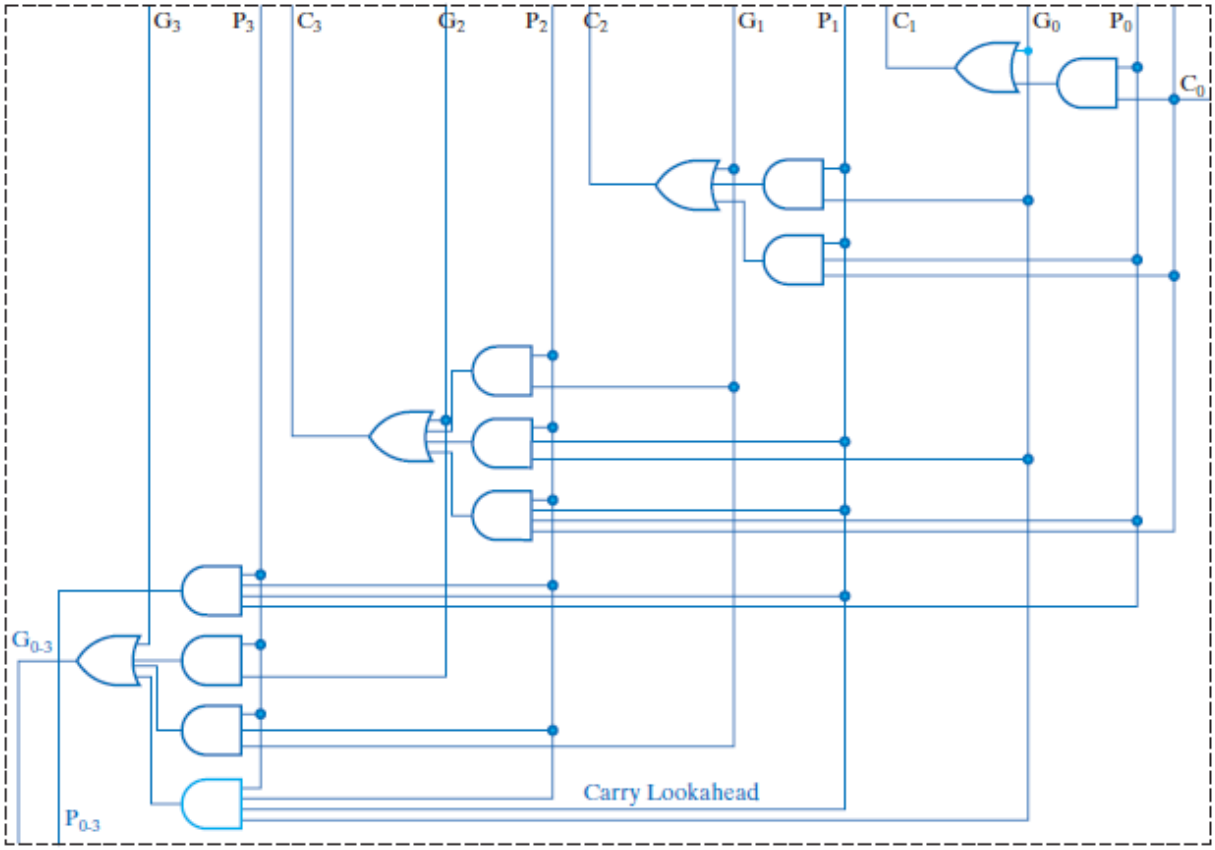
1. Find the ADD4 symbol in the symbol library. Draw a schematic diagram in the space provided below and show how to make an 8-bit adder using the ADD4s.

2. If you are asked to build and simulate the above design, what types of I/O buffers (symbols) would be convenient to use? How many sum bits would you expect to have?

3. What is a carry look-ahead adder? Why is it preferred over a regular adder? (Refer to your textbook and diagrams of Ripple Carry and Carry Lookahead in the following.)


B
A
S
G
P
C
B3
A3
B2
A2
B1
A1
B0
A0
PFA
PFA
PFA
PFA
S3
G3
P3
C3
S2
G2
P2
C2
S1
G1
P1
C1
S0
G0
P0
C4
C0
Ripple Carry

(a)


G3
P3
C3
G2
P2
C2
G1
P1
C1
G0
P0
C0
G0-3
P0-3
Carry Lookahead