EE126

Lab 1 Carry propagation adder

Welcome to ee126 lab1. In this lab, we will investigate carry propagation adders, as well as VHDL/Verilog programming. We will also design two types of 4-bit carry propagation adders and implement them on an FPGA device.

Before we start, it is a good idea to review the logic design of 1-bit full adders. (Check the appendix for the VHDL/Verilog code of a full-bit adder.)

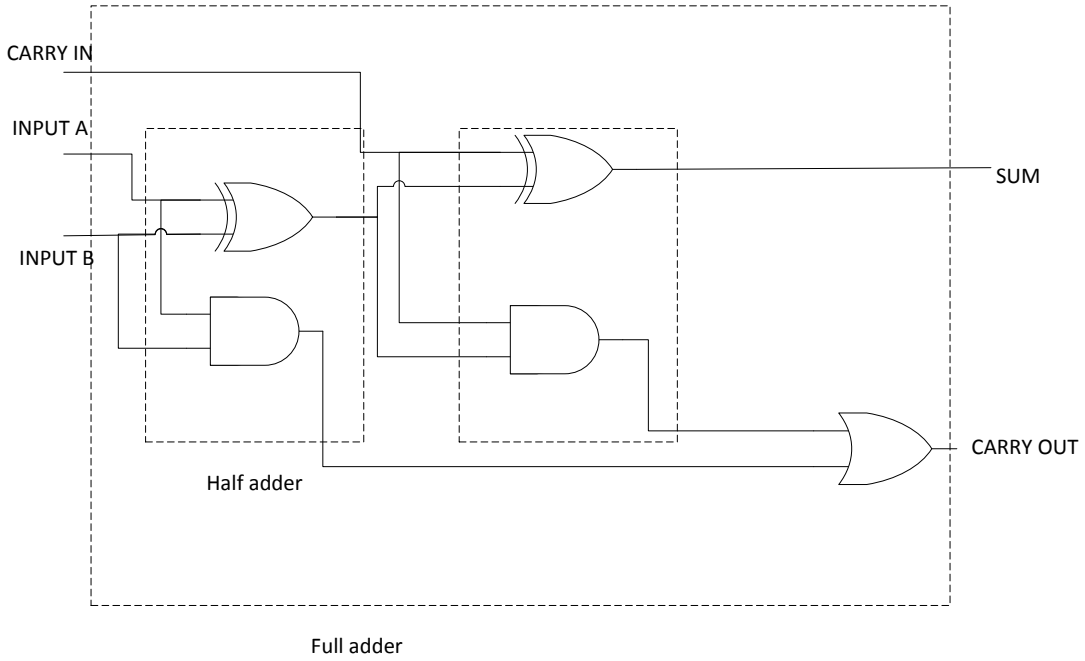
A full adder adds binary numbers and accounts for values carried in as well as out. A one-bit full adder adds three one-bit numbers, often written as A, B, and Cin; A and B are the operands, and Cin is a bit carried in from the previous less significant stage. The full-adder is usually a component in a cascade of adders, which add 8, 16, 32, etc. bit binary numbers. The circuit produces an unsigned two-bit output, output carry and sum typically represented by the signals Cout and S, where $sum = 2 \times C_{out} + S$.

A full adder can be implemented in many different ways such as with a custom transistor-level circuit or composed of other gates. The Boolean functions for the full adder in terms of exclusive-OR operations can be expressed as:

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$$

In this implementation, the final OR gate before the carry-out output may be replaced by an XOR gate without altering the resulting logic. The logic diagram for this multiple-level implementation consists of two half adders and an OR gate. It is shown below:



Q1. Write down the 1-bit full adder's truth table.

N-bit adders take inputs {AN, ⋯, A1}, {BN, ⋯, B1}, and carry-in Cin, and compute the sum {SN, ⋯, S1} and the carry-out of the most significant bit Cout. They are called carry-propagate adders (CPAs) because the carry into each bit can influence the carry into all subsequent bits.

It is easy to do a simple design in which the carry-out of one bit is simply connected as the carry-in to the next. This is called the carry-ripple adder, since each carry bit "ripples" to the next full adder.

Q2. Design a 4-bit carry-ripple adder using 4 one-bit full adders in VHDL/Verilog. Following restrictions apply:

- All the numbers are signed 4 bit numbers. Use 2's complement to represent the numbers.
- Use a one bit output overflow to indicate overflow in the addition.
- Use inputs cin and cout to indicate carry-in and carry-out.
- DO NOT use arithmetic operators in VHDL/Verilog. The adder should be implemented using only logic gates.

Q3. Implement your 4-bit carry-ripple adder on a FPGA using the following Pin assignment table.

Register	Pin	Board Component
a[0]	PIN_N25	SW[0]
a[1]	PIN_N26	SW[1]
a[2]	PIN_P25	SW[2]
a[3]	PIN_AE14	SW[3]
b[0]	PIN_AF14	SW[4]
b[1]	PIN_AD13	SW[5]
b[2]	PIN_AC13	SW[6]
b[3]	PIN_C13	SW[7]
cin	PIN_B13	SW[8]
sum[0]	PIN_AE23	LEDR[0]
sum[1]	PIN_AF23	LEDR[1]

sum[2]	PIN_AB21	LEDR[2]
sum[3]	PIN_AC22	LEDR[3]
cout	PIN_AE22	LEDG[0]
overflow	PIN_AF22	LEDG[1]

The layout of a ripple-carry adder is simple, which allows for fast design time; however, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder.

To reduce the computation time, engineers devised faster ways to add two binary numbers by using carry-lookahead adders.

We begin by introducing two new functions from which we will construct the lookahead carry. These are called carry generate, written as Gi, and carry propagate, written as Pi. They are defined as

$$G_i = A_i \bullet B_i$$

$$P_i = A_i \oplus B_i$$

When Ai and Bi are both 1, a carry-out must be asserted, independently of the carry-in. Hence, we call the function a carry generate. If one of Ai and Bi is 1 while the other is 0, then the carry-out will be identical to the carry-in. In other words, when the XOR is true, we pass or propagate the carry across that stage. The sum and carry-out can be expressed in terms of the carry-generate and carry-propagate functions:

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$$

$$
\begin{aligned}
C_{i+1} &= A_i B_i + A_i C_i + B_i C_i \\
&= A_i B_i + C_i (A_i + B_i) \\
&= A_i B_i + C_i (A_i \oplus B_i) \\
&= G_i + C_i P_i
\end{aligned}
$$

When the carry-out is 1, either the carry is generated internally within the stage (G_i) or the carry-in is 1 (C_i) and it is propagated (P_i) through the stage. Expressed in terms of carry propagate and generate, we can rewrite the carry-out logic as follows:

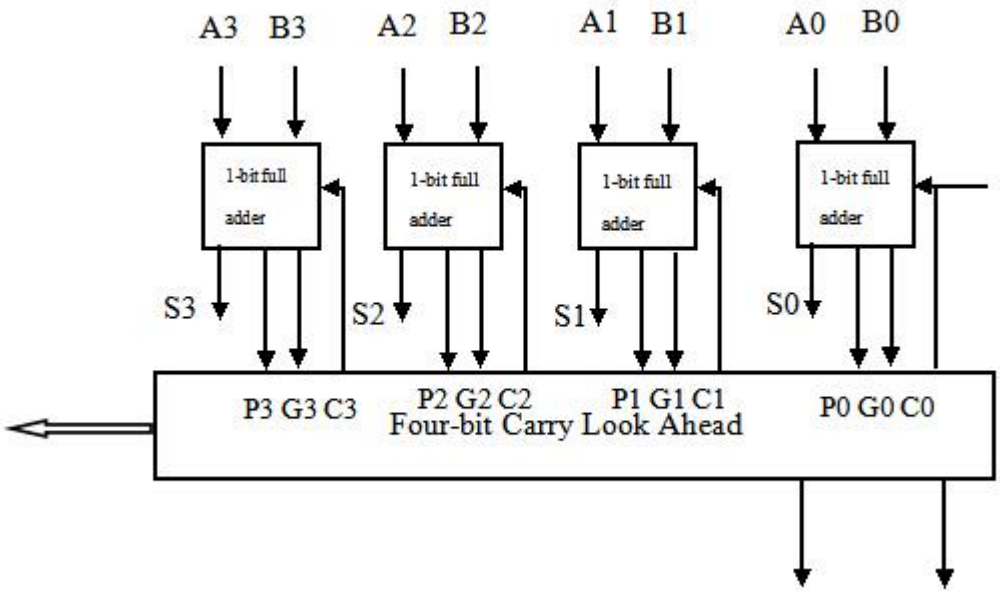
$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

The ith carry signal is the OR of i+1 product terms, the most complex of which has i+1 literals. This places a practical limit on the number of stages across which the carry lookahead logic can be computed. Four-stage lookahead circuits commonly are available in parts catalogs and cell libraries.A 4-bit carry-lookahead adder is given below.



Q4. Design a 4-bit carry-lookahead adder using VHDL/Verilog.

Q5. Compare the delay of carry-ripple adder and carry-lookahead adder. Explain the reason of shorter delay.

Appendix A: VHDL and Verilog Standard Formats

Standard Structure of a VHDL Design

```
entity entity_name is
        Port(signal0 : in std_logic;
                signal1 : out std_logic;
                …
                signaln : out std_logic_vector (3 downto 0));
end entity_name;
architecture Behavioral of entity_name is
-- component declarations
component comp_name is
        Port(a : in std_logic;
                …);
end component;
-- signal declarations
signal wire0, wire1 : std_logic;
-- main block
begin
-- behavioral and/or structural code here.
-- module instantiation
instance_name: comp_name
        port map(signal0, signal1, …);
-- logical operations
signal3 <= (signal4 and signal5) xor signal8;
end
```

Standard Structure of a Verilog Design

```
module module_name(signal0,
                                signal1,
                                … ,
                                signaln);
        // module signals
        input signal0;
        output [15:0] signal1;
        …
        output signaln;
        // internal registers
        reg register0;
        reg signal1;
        // internal signals
        wire wire0;
```

```
        wire wire1;
// behavioral and/or structural code here.
// module instantiation
module_name1 instance_name1 (signal0, signal1);
// logical operations
always @ (signal4 or signal5 or signal8)
begin
        signal3 <= (signal4 && signal5) ^^ signal8;
end
endmodule
```