Chapter 12: Arithmetic Circuits

This chapter presents the design and timing considerations of circuits to perform basic arithmetic operations including addition, subtraction, multiplication, and division. A discussion is also presented on how to model arithmetic circuits in VHDL. The goal of this chapter is to provide an understanding of the basic principles of binary arithmetic circuits.

Learning Outcomes—After completing this chapter, you will be able to:

12.1 Design a binary adder using both the classical digital design approach and the modern HDL-based approach.
12.2 Design a binary subtractor using both the classical digital design approach and the modern HDL-based approach.
12.3 Design a binary multiplier using both the classical digital design approach and the modern HDL-based approach.
12.4 Design a binary divider using both the classical digital design approach and the modern HDL-based approach.

12.1 Addition

Binary addition is performed in a similar manner to performing decimal addition by hand. The addition begins in the least significant position of the number ($p = 0$). The addition produces the sum for this position. In the event that this positional sum cannot be represented by a single symbol, then the higher-order symbol is *carried* to the subsequent position ($p = 1$). The addition in the next higher position must include the number that was carried in from the lower positional sum. This process continues until all of the symbols in the number have been operated on. The final positional sum can also produce a carry, which needs to be accounted for in a separate system.

Designing a binary adder involves creating a combinational logic circuit to perform the positional additions. Since a combinational logic circuit can only produce a scalar output, circuitry is needed to produce the sum and the carry at each position. The binary adder size is predetermined and fixed prior to implementing the logic (i.e., an n-bit adder). Both inputs to the adder must adhere to the fixed size, regardless of their value. Smaller numbers simply contain leading zeros in their higher-order positions. For an *n*-bit adder, the largest sum that can be produced will require *n* + 1 bits. To illustrate this, consider a 4-bit adder. The largest numbers that the adder will operate on are $1111_2 + 1111_2$. (or $15_{10} + 15_{10}$). The result of this addition is 11110_2 (or 30_{10}). Notice that the largest sum produced fits within 5 bits, or *n* + 1. When constructing an adder circuit, the sum is always recorded using n-bits with a separate carry out bit. In our 4-bit example, the sum would be expressed as "1110" with a carry out. The carry out bit can be used in multiple word additions, used as part of the number when being decoded for a display, or simply discarded as in the case when using two's complement numbers.

12.1.1 Half Adders

When creating an adder, it is desirable to design incremental subsystems that can be reused. This reduces design effort and minimizes troubleshooting complexity. The most basic component in the adder is called a *half adder*. This circuit computes the sum and carry out on two input arguments. The reason it is called a half adder instead of a full adder is because it does not accommodate a *carry in* during the computation, thus it does not provide all of the necessary functionality required for the positional adder.

Example 12.1 shows the design of a half adder. Notice that two combinational logic circuits are required in order to produce the sum (the XOR gate) and the carry out (the AND gate). These two gates are in parallel to each other, thus the delay through the half adder is due to only one level of logic.

Example: Design of a Half Adder

Recall in binary addition, the output consists of a <u>sum</u> and a <u>carry bit</u>.

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline \underline{0} \end{array} \leftarrow \text{Sum} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \text{Carry} \rightarrow \begin{array}{r} 1 \\ +\ 1 \\ \hline \underline{1}\,0 \end{array}$$

We can build a simple circuit called a "Half Adder" to compute these outputs.

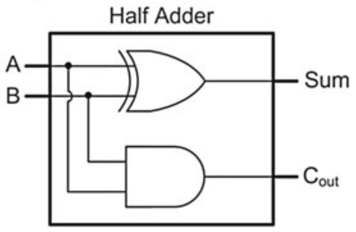
A	B	C_{out}	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$\rightarrow$

$\text{Sum} = A \oplus B$

$C_{out} = A \cdot B$

$\rightarrow$



Example 12.1
Design of a half adder

12.1.2 Full Adders

A full adder is a circuit that still produces a sum and carry out, but considers three inputs in the computations (A, B, and C_{in}). Example 12.2 shows the design of a full adder.

Example: Design of a Full Adder

In order to create multi-bit adders, a circuit is needed that also includes a "Carry In" bit.

The sum of position 1 needs to include the "Carry Out" from the sum of position 0. The sum of position 1 must include this carry, which is reffered to as the "Carry In" bit.

$$\begin{array}{r} {\scriptstyle 1} \\ 0\ 1 \\ +\ 0\ 1 \\ \hline 1\ 0 \end{array}$$

This circuit is called a "Full Adder".

C_{in}	A	B	C_{out}	Sum
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

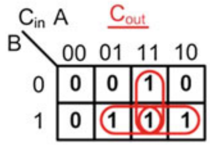
$\rightarrow$

<u>Sum</u>

B \ C_{in} A	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$\rightarrow$ $\text{Sum} = A \oplus B \oplus C_{in}$

<u>C_{out}</u>



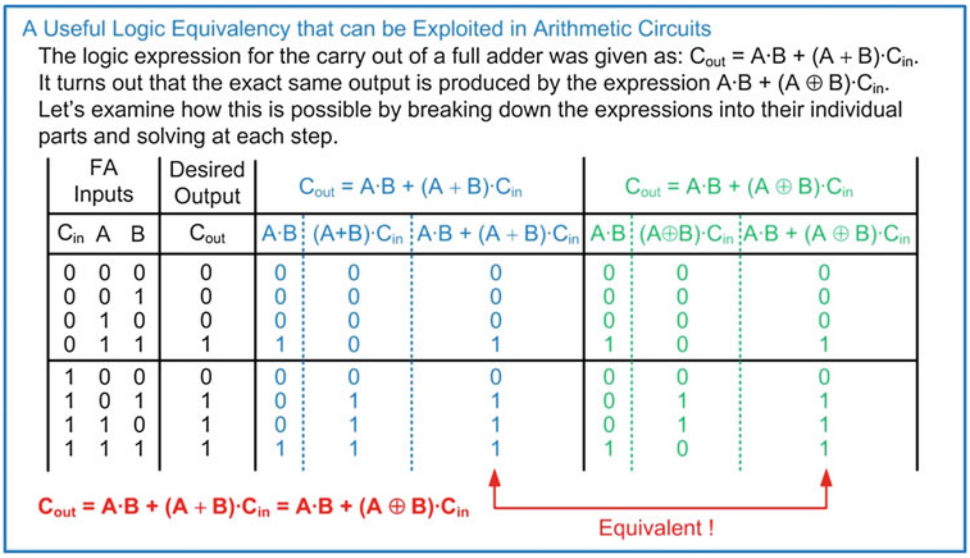
B \ C_{in} A	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$\rightarrow$

$$\begin{aligned} C_{out} &= A \cdot C_{in} + A \cdot B + B \cdot C_{in} \\ &= A \cdot B + (A+B) \cdot C_{in} \end{aligned}$$

Example 12.2
Design of a full adder

As mentioned before, it is desirable to reuse design components as we construct more complex systems. One such design reuse approach is to create a full adder using two half adders. This is straightforward for the sum output since the logic is simply two cascaded XOR gates (Sum $=$ A$\oplus$B$\oplus$Cin). The carry out is not as straightforward. Notice that the expression for Cout derived in Example 12.2 contains the term (A + B). If this term could be manipulated to use an XOR gate instead, it would allow the full adder to take advantage of existing circuitry in the system. Figure 12.1 shows a derivation of an equivalency that allows (A + B) to be replaced with (A$\oplus$B) in the Cout logic expression.



A Useful Logic Equivalency that can be Exploited in Arithmetic Circuits

The logic expression for the carry out of a full adder was given as: $C_{out} = A \cdot B + (A + B) \cdot C_{in}$. It turns out that the exact same output is produced by the expression $A \cdot B + (A \oplus B) \cdot C_{in}$. Let's examine how this is possible by breaking down the expressions into their individual parts and solving at each step.

FA Inputs			Desired Output	$C_{out} = A \cdot B + (A + B) \cdot C_{in}$			$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$		
C_{in}	A	B	C_{out}	$A \cdot B$	$(A+B) \cdot C_{in}$	$A \cdot B + (A + B) \cdot C_{in}$	$A \cdot B$	$(A \oplus B) \cdot C_{in}$	$A \cdot B + (A \oplus B) \cdot C_{in}$
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	1	1	1	1	0	1	1	0	1
1	0	0	0	0	0	0	0	0	0
1	0	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1	0	1

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

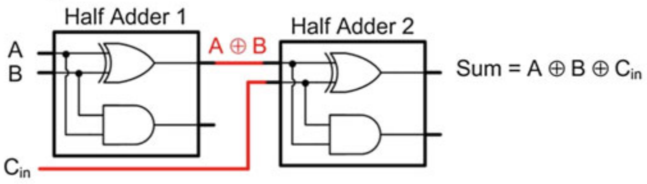
Equivalent !

Fig. 12.1
A useful logic equivalency that can be exploited in arithmetic circuits

The ability to implement the carry out logic using the expression $C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$ allows us to implement a full adder with two half adders and the addition of a single OR gate. Example 12.3 shows this approach. In this new configuration, the sum is produced in two levels of logic while the carry out is produced in three levels of logic.

Example – Design of a Full Adder Out of Two Half Adders

It is often desirable to create a full adder out of two half adders in order to re-use existing design components. The "Sum" of the full adder can be created by using two cascaded XOR gates provided by the half adders.



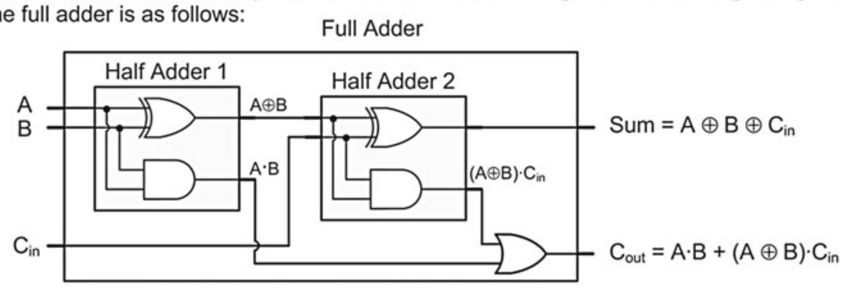
The expression for the "Carry Out" of the full adder is:

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

or

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

Notice that the carry out of Half Adder 1 produces the A·B term in this expression. Also notice that the carry out of Half Adder 2 produces the $(A \oplus B) \cdot C_{in}$ term. The only remaining logic needed to create the carry out of the full adder is an OR gate. The final logic diagram for the full adder is as follows:



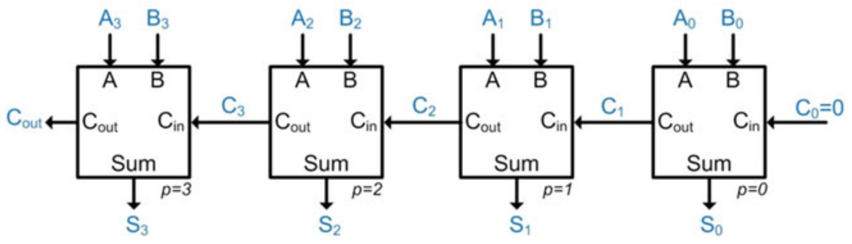
Example 12.3
Design of a full adder out of half adders

12.1.3 Ripple Carry Adder (RCA)

The full adder can now be used in the creation of multi-bit adders. The simplest architecture exploiting the full adder is called a *ripple carry adder* (RCA). In this approach, full adders are used to create the sum and carry out of each bit position. The carry out of each full adder is used as the carry in for the next higher position. Since each subsequent full adder needs to wait for the carry to be produced by the preceding stage, the carry is said to *ripple* through the circuit, thus giving this approach its name. Example 12.4 shows how to design a 4-bit ripple carry adder using a chain of full adders. Notice that the carry in for the full adder in position 0 is tied to a logic 0. The 0 input has no impact on the result of the sum but enables a full adder to be used in the 0th position.

Example: Design of a 4-Bit Ripple Carry Adder (RCA)

Full adders can be cascaded together to form a multi-bit adder. The symbols are typically drawn in the following fashion to mirror a positional number system.



The sum of position 1 cannot complete until it receives the carry in (C_1) from the sum in position 0. The position 2 sum cannot complete until it receives the carry in (C_2) from the sum in position 1, etc. In this way, the carry "ripples" through the circuit from right to left. This configuration is known as a Ripple Carry Adder (RCA).

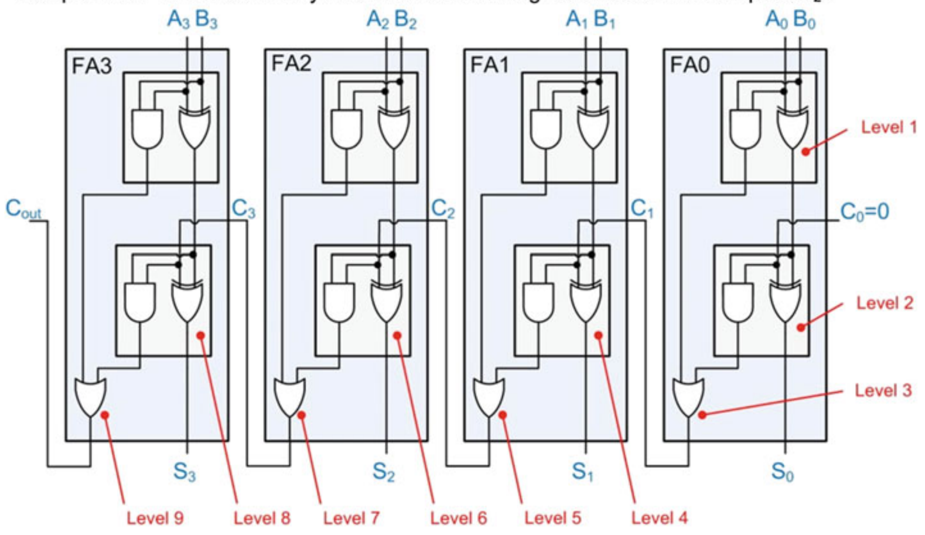
Example 12.4
Design of a 4-Bit Ripple Carry Adder (RCA)

While the ripple carry adder provides a simple architecture based on design reuse, its delay can become considerable when scaling to larger inputs sizes (e.g., $n = 32$ or $n = 64$). A simple analysis of the timing can be stated such that if the time for a full adder to complete its positional sum is t_{FA}, then the time for an n-bit ripple carry adder to complete its computation is $t_{RCA} = n \cdot t_{FA}$.

If we examine the RCA in more detail, we can break down the delay in terms of the levels of logic necessary for the computation. Example 12.5 shows the timing analysis of the 4-bit RCA. This analysis determines the number of logic levels in the adder. The actual gate delays can then be plugged in to find the final delay. The inputs to the adder are A, B, and C_{in} and are always assumed to update at the same time. The first full adder requires two levels of logic to produce its sum and three levels to produce its carry out. Since the timing of a circuit is always stated as its worst case delay, we say that the first full adder takes three levels of logic. When the carry (C_1) ripples to the next full adder (FA1), it must propagate through two additional levels of logic in order to produce C_2. Notice that the first half adder in FA1 only depends on A_1 and B_1, thus it is able to perform this computation immediately. This half adder can be considered as first-level logic. More importantly, it means that when the carry in arrives (C_1), only two additional levels of logic are needed, not three. The levels of logic for the RCA can be expressed as $3 + 2 \cdot (n - 1)$. If each level of logic has a delay of t_{gate}, then a more accurate expression for the RCA delay is $t_{RCA} = (3 + 2 \cdot (n - 1)) \cdot t_{gate}$.

Example: Timing Analysis of a Ripple Carry Adder

The carry ripples through the adder chain. The first full adder (FA0) takes three levels of logic to complete. When C_1 reaches the FA1, its first half adder has already completed its computation. This means only two more levels of logic are needed to compute C_2.



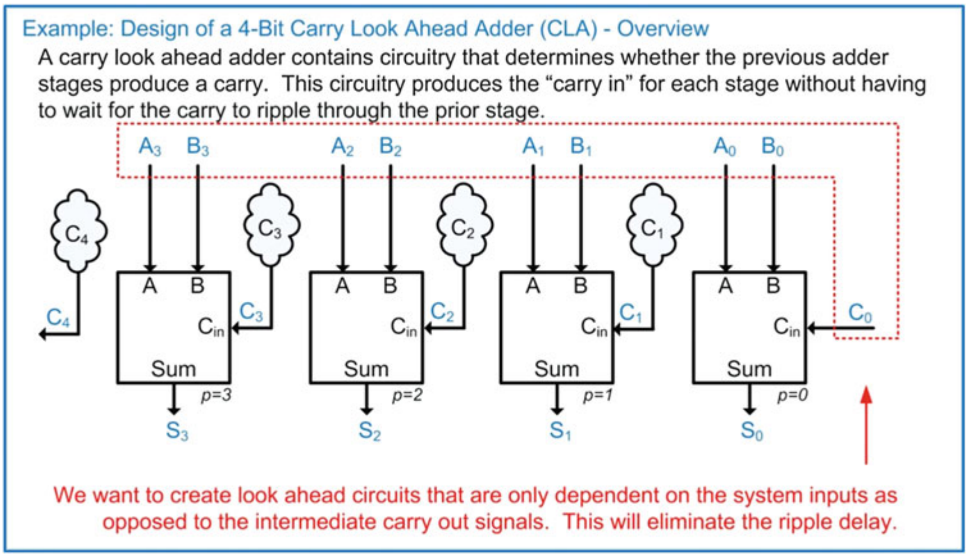
The number of logic levels for this adder is given as:

$$\text{Levels of Logic} = 3 + 2\cdot(n-1)$$

Example 12.5
Timing analysis of a 4-bit ripple carry adder

12.1.4 Carry Look Ahead Adder (CLA)

In order to address the potentially significant delay of a ripple carry adder, a *carry look ahead* (CLA) adder was created. In this approach, additional circuitry is included that produces the intermediate carry in signals immediately instead of waiting for them to be created by the preceding full adder stage. This allows the adder to complete in a fixed amount of time instead of one that scales with the number of bits in the adder. Example 12.6 shows an overview of the design approach for a CLA.



Example: Design of a 4-Bit Carry Look Ahead Adder (CLA) - Overview

A carry look ahead adder contains circuitry that determines whether the previous adder stages produce a carry. This circuitry produces the "carry in" for each stage without having to wait for the carry to ripple through the prior stage.

We want to create look ahead circuits that are only dependent on the system inputs as opposed to the intermediate carry out signals. This will eliminate the ripple delay.

Example 12.6
Design of a 4-Bit Carry Look Ahead Adder (CLA)—Overview

For the CLA architecture to be effective, the look ahead circuitry needs to be dependent only on the system inputs A, B, and C_{in} (i.e., C_0). A secondary characteristic of the CLA is that it should exploit as much design reuse as possible. In order to examine the design reuse aspects of a multi-bit adder, the concepts of carry **generation** (g) and **propagation** (p) are used. A full adder is said to *generate* a carry if its inputs A and B result in $C_{out} = 1$ when $C_{in} = 0$. A full adder is said to *propagate* a carry if its inputs A and B result in $C_{out} = 1$ when $C_{in} = 1$. These simple statements can be used to derive logic expressions for each stage of the adder that can take advantage of existing logic terms from prior stages. Example 12.7 shows the derivation of these terms and how algebraic substitutions can be exploited to create look ahead circuitry for each full adder that is only dependent on the system inputs. In these derivations, the variable *i* is used to represent position since *p* is used to represent the propagate term.

Example: Design of a 4-Bit Carry Look Ahead Adder (CLA) – Algebraic Formation

The look ahead circuitry considers whether the prior adder stages create a carry by considering two conditions: 1) whether a stage will **generate** (*g*) a carry; and 2) whether the stage will **propagate** (*p*) a carry. Let's look at the truth table for a full adder.

C_{in}	A	B	C_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

For the input codes where C_{in}=0, the full adder "generates" a new carry when A=1 and B=1. This behavior can be described with the expression: $g = A \cdot B$

For the input codes where C_{in}=1, the full adder "propagates" the incoming carry when either A=1 or B=1. This behavior can be described with the expression: $p = A+B$

The entire expression for the carry out can be written as:

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

Let's see how this can be used to our advantage in a multiple bit adder. Recall that for any arbitrary adder position, the generate, propagate, and carry out terms are:

$$g_i = A_i \cdot B_i$$
$$p_i = A_i + B_i$$
$$C_{i+1} = g_i + p_i \cdot C_i$$

Note: We'll use the subscript "i" to denote position since we're using "p" for *propagate*.

We can now write expressions for the subsequent carry terms as:

$$C_1 = g_0 + p_0 \cdot C_0$$

The C_1 expression only depends on the inputs A, B, and C_0.

$$C_2 = g_1 + p_1 \cdot C_1$$
$$C_2 = g_1 + p_1 \cdot (g_0 + p_0 \cdot C_0)$$
$$C_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot C_0$$

For C_2, we can plug in the expression for C_1 to create an expression that only depends on A, B, and C_0...

$$C_3 = g_2 + p_2 \cdot C_2$$
$$C_3 = g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_0 \cdot p_1 \cdot C_0)$$
$$C_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot C_0$$

and again for C_3…

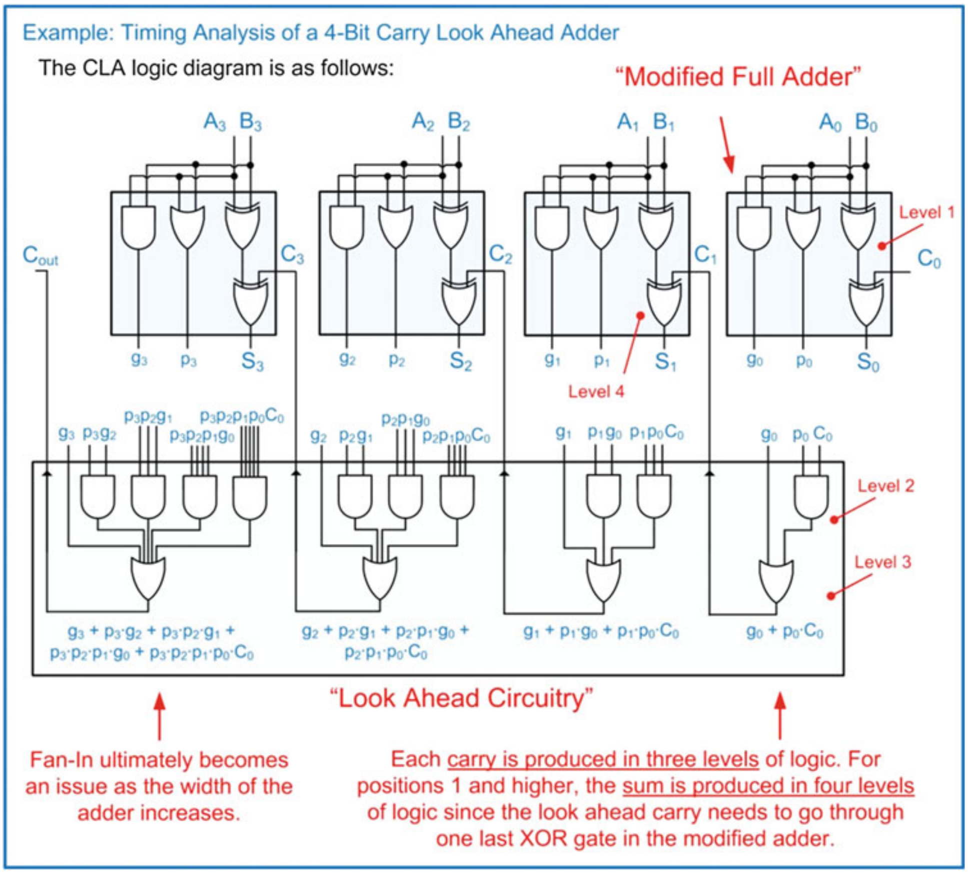
$$C_4 = g_3 + p_3 \cdot C_3$$
$$C_4 = g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot C_0)$$
$$C_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot C_0$$

and again for C_4…

All of these expressions only depend on the inputs A, B, and C_0. Also notice that each expression is in a 2-level sum of products form.

Example 12.7
Design of a 4-Bit Carry Look Ahead Adder (CLA)—algebraic formation

Example 12.8 shows a timing analysis of the 4-bit carry look ahead adder. Notice that the full adders are modified to add the logic for the generate and propagate bits in addition to removing the unnecessary gates associated with creating the carry out.


Example: Timing Analysis of a 4-Bit Carry Look Ahead Adder
The CLA logic diagram is as follows:
"Modified Full Adder"
A3 B3
A2 B2
A1 B1
A0 B0
Level 1
Cout
C3
C2
C1
C0
g3 p3 S3
g2 p2 S2
g1 p1 S1
g0 p0 S0
Level 4
g3 p3g2 p3p2g1 p3p2p1g0 p3p2p1p0C0
g2 p2g1 p2p1g0 p2p1p0C0
g1 p1g0 p1p0C0
g0 p0 C0
Level 2
Level 3
g3 + p3·g2 + p3·p2·g1 + p3·p2·p1·g0 + p3·p2·p1·p0·C0
g2 + p2·g1 + p2·p1·g0 + p2·p1·p0·C0
g1 + p1·g0 + p1·p0·C0
g0 + p0·C0
"Look Ahead Circuitry"
Fan-In ultimately becomes an issue as the width of the adder increases.
Each carry is produced in three levels of logic. For positions 1 and higher, the sum is produced in four levels of logic since the look ahead carry needs to go through one last XOR gate in the modified adder.

Example 12.8
Timing analysis of a 4-bit carry look ahead adder

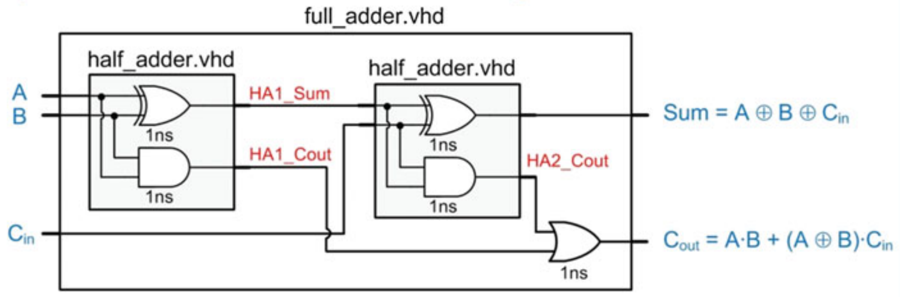
The 4-bit CLA can produce the sum in four levels of logic as long as fan-in specifications are met. As the CLA width increases, the look ahead circuitry will become fan-in limited and additional stages will be required to address the fan-in. Regardless, the CLA has considerably less delay than a RCA as the width of the adder is increased.

12.1.5 Adders in VHDL

12.1.5.1 Structural Model of a Ripple Carry Adder in VHDL

A structural model of a ripple carry adder is useful to visualize the propagation delay of the circuit in addition to the impact of the carry rippling through the chain. Example 12.9 shows the structural model for a full adder in VHDL consisting of two half adders. The full adder is created by instantiating two versions of the half adder as components. In this example, all gates are modeled with a delay of 1ns.

Example: Structural Model of a Full Adder in VHDL Using Two Half Adders



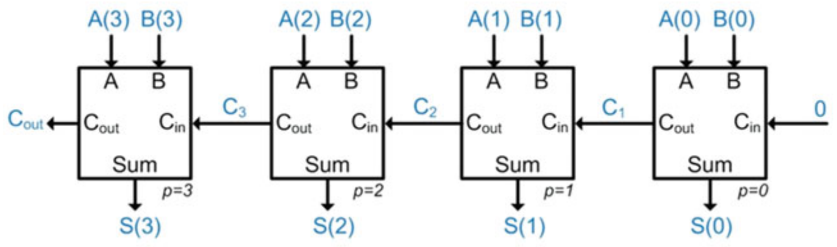
```
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
  port (A, B       : in  std_logic;
        Sum, Cout : out std_logic);
end entity;

architecture half_adder_arch of half_adder is

 begin

   Sum  <= A xor B after 1 ns;
   Cout <= A and B after 1 ns;

end architecture;
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
  port (A, B, Cin  : in  std_logic;
        Sum, Cout  : out std_logic);
end entity;

architecture full_adder_arch of full_adder is

   component half_adder
     port (A, B       : in  std_logic;
           Sum, Cout  : out std_logic);
   end component;

   signal HA1_Sum, HA1_Cout, HA2_Cout : std_logic;

 begin

   HA1 : half_adder port map (A, B, HA1_Sum, HA1_Cout);
   HA2 : half_adder port map (HA1_Sum, Cin, Sum, HA2_Cout);

   Cout <= HA1_Cout or HA2_Cout after 1 ns;

end architecture;
```

Example 12.9
Structural model of a full adder in VHDL using two half adders

Example 12.10 shows the structural model of a 4-bit ripple carry adder in VHDL. The RCA is created by instantiating four full adders. Notice that a logic 0 can be directly inserted into the port map of the first full adder to model the behavior of $C_0 = 0$.

Example: Structural Model of a 4-Bit Ripple Carry Adder in VHDL



```
library IEEE;
use IEEE.std_logic_1164.all;

entity rca_4bit is
  port (A, B   : in  std_logic_vector(3 downto 0);
        Sum    : out std_logic_vector(3 downto 0);
        Cout   : out std_logic);
end entity;

architecture rca_4bit_arch of rca_4bit is

   component full_adder
     port (A, B, Cin   : in  std_logic;
           Sum, Cout   : out std_logic);
   end component;

   signal C1, C2, C3 : std_logic;

 begin

   A0 : full_adder port map (A(0), B(0), '0', Sum(0), C1);
   A1 : full_adder port map (A(1), B(1), C1,  Sum(1), C2);
   A2 : full_adder port map (A(2), B(2), C2,  Sum(2), C3);
   A3 : full_adder port map (A(3), B(3), C3,  Sum(3), Cout);

end architecture;
```

Example 12.10
Structural model of a 4-bit ripple carry adder in VHDL

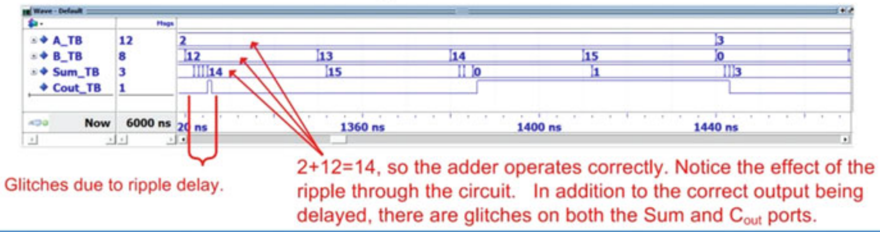
When creating arithmetic circuitry, testing under all input conditions is necessary to verify functionality. Testing under each and every input condition can require a large number of input conditions. To test an n-bit adder under each and every numeric input condition will take $(2^n)^2$ test vectors. For our simple 4-bit adder example, this equates to 256 input patterns. The large number of input patterns precludes the use of manual signal assignments in the test bench to stimulate the circuit. One approach to generating the input test patterns is to use nested for loops. Example 12.11 shows a test bench that uses two nested for loops to generate the 256 unique input conditions for the 4-bit ripple carry adder. Note that the loop variables *i* and *j* are automatically created when the loops are declared. Since the loop variables are defined as integers, type conversions are required prior to driving the values into the RCA. The simulation waveform illustrates how the ripple carry adder has a noticeable delay before the output sum is produced. During the time the carry is rippling through the adder chain, glitches can appear on each of the sum bits in addition to the carry out signal. The values in this waveform are displayed as unsigned decimal symbols to make the results easier to interpret.

Example: VHDL Test Bench for a 4-Bit Ripple Carry Adder Using Nested For Loops

Nested for loops can be used in order to generate an exhaustive set of test vectors to stimulate the adder.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity rca_4bit_TB is
end entity;

architecture rca_4bit_TB_arch of rca_4bit_TB is

   component rca_4bit
     port (A, B   : in  std_logic_vector(3 downto 0);
           Sum    : out std_logic_vector(3 downto 0);
           Cout   : out std_logic);
   end component;

   signal A_TB, B_TB, Sum_TB   : std_logic_vector(3 downto 0);
   signal Cout_TB              : std_logic;


 begin

   DUT : rca_4bit port map (A_TB, B_TB, Sum_TB, Cout_TB);

   STIM : process
     begin

      for i in 0 to 15 loop
         for j in 0 to 15 loop
           A_TB <= std_logic_vector(to_unsigned(i,4));
           B_TB <= std_logic_vector(to_unsigned(j,4));
           wait for 30 ns;
         end loop;
      end loop;

   end process;

end architecture;
```

The simulation waveform for the ripple carry adder is as follows. The numbers are shown in unsigned decimal format for readability.



Glitches due to ripple delay.

2+12=14, so the adder operates correctly. Notice the effect of the ripple through the circuit. In addition to the correct output being delayed, there are glitches on both the Sum and C_{out} ports.

Example 12.11
VHDL test bench for a 4-bit ripple carry adder using nested for loops

12.1.5.2 Structural Model of a Carry Look Ahead Adder in VHDL

A carry look ahead adder can also be modeled using a combination of concurrent signal assignments with logical operators and modified full adder components. Example 12.12 shows a structural model for a 4-bit CLA in VHDL. In this example, the gate delay is modeled using a constant (tgate) of 1ns. The delay due to multiple levels of logic is entered manually to simplify the model. The two cascaded XOR gates in the modified full adder are modeled using a single signal assignment with 2*tgate of delay.

Example: Structural Model of a 4-Bit Carry Look Ahead Adder in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mod_full_adder is
  port (A, B, Cin  : in std_logic;
        Sum, p, g  : out std_logic);
end entity;

architecture mod_full_adder_arch of mod_full_adder is

   constant tgate : time := 1 ns;

 begin

   Sum   <= (A xor B xor Cin) after 2*tgate;
   p     <= (A or B)          after 1*tgate;
   g     <= (A and B)         after 1*tgate;

end architecture;
```

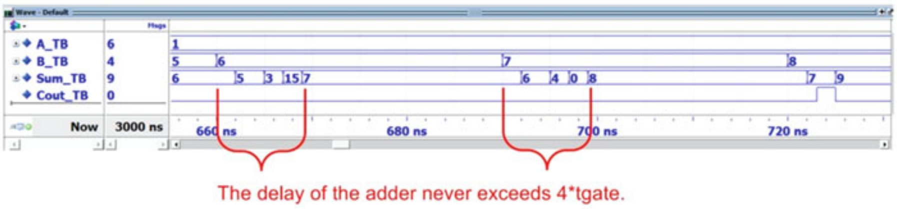
```
library IEEE;
use IEEE.std_logic_1164.all;

entity cla_4bit is
  port (A, B  : in std_logic_vector(3 downto 0);
        Sum   : out std_logic_vector(3 downto 0);
        Cout  : out std_logic);
end entity;

architecture cla_4bit_arch of cla_4bit is

   constant tgate : time := 1 ns;

   component mod_full_adder
     port (A, B, Cin  : in  std_logic;
           Sum, p, g  : out std_logic);
   end component;

   signal C0, C1, C2, C3 : std_logic;
   signal p, g           : std_logic_vector(3 downto 0);

 begin

   C0   <= '0';
   C1   <= g(0) or (p(0) and C0) after 2*tgate;
   C2   <= g(1) or (p(1) and C1) after 2*tgate;
   C3   <= g(2) or (p(2) and C2) after 2*tgate;
   Cout <= g(3) or (p(3) and C3) after 2*tgate;

   A0 : mod_full_adder port map (A(0), B(0), C0, Sum(0), p(0), g(0));
   A1 : mod_full_adder port map (A(1), B(1), C1, Sum(1), p(1), g(1));
   A2 : mod_full_adder port map (A(2), B(2), C2, Sum(2), p(2), g(2));
   A3 : mod_full_adder port map (A(3), B(3), C3, Sum(3), p(3), g(3));

end architecture;
```

Example 12.12
Structural model of a 4-bit carry look ahead adder in VHDL

Example 12.13 shows the simulation waveform for the 4-bit carry look ahead adder. The outputs still have intermediate transitions while the combinational logic is computing the results; however, the overall delay of the adder is bound to $< 4*t_{gate}$.

Example: 4-Bit Carry Look Ahead Adder - Simulation Waveform

The following simulation waveform illustrates that there are still glitches on the outputs while the logic computes the sum and carry out. The CLA architecture bounds the overall delay of the chain.



Example 12.13
4-Bit carry look ahead adder—simulation waveform

12.1.5.3 Behavior Model of an Adder Using UNSIGNED Data Types

VHDL also supports adder models at a higher level of abstraction using the “+” operator. While this operator is supported for the type integer in the std_logic_1164 package, modeling adders using integers can be onerous due to the multiple levels of casting, range checking, and manual handling of carry out. A simpler approach to modeling adder behavior is to use the types unsigned/signed and the “+” operator provided in the numeric_std package. Temporary signals or variables of these types are required to model the adder behavior with the “+” sign. Also, type casting is still required when assigning the values back to the output ports. One advantage of this approach is that range checking is eliminated because rollover is automatically handled with these types.

Example 12.14 shows the behavioral model for a 4-bit adder in VHDL. In this model, a 5-bit unsigned vector is created (Sum_uns). The two inputs, A and B, are concatenated with a leading zero in order to facilitate assigning the sum to this 5-bit vector. The advantage of this approach is that the carry out of the adder is automatically included in the sum as the highest position bit. Since A and B are of type std_logic_vector, they must be converted to unsigned before the addition with the “+” operator can take place. The concatenation, type conversion, and addition can all take place in a single assignment.

Example:

```
Sum_uns <= unsigned(('0' & A)) + unsigned(('0' & B));
```

The 5-bit vector Sum_uns now contains the 4-bit sum and carry out. The final step is to assign the separate components of this vector to the output ports of the system. The 4-bit sum portion requires a type conversion back to std_logic_vector before it can be assigned to the output port *Sum*. Since the *Cout* port is a scalar, an unsigned signal can be assigned to it directly without the need for a conversion.

Example:

```
Sum <= std_logic_vector(Sum_uns(3 downto 0));
Cout <= Sum_uns(4);
```

Example: Behavioral Model of a 4-Bit Adder in VHDL

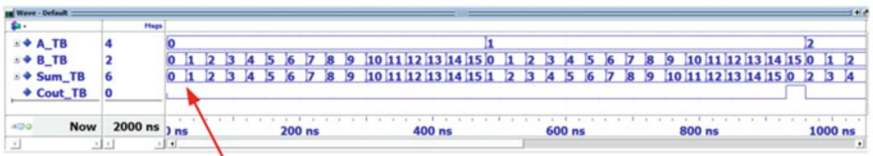
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity adder_4bit is
  port (A, B  : in  std_logic_vector(3 downto 0);
        Sum   : out std_logic_vector(3 downto 0);
        Cout  : out std_logic);
end entity;

architecture adder_4bit_arch of adder_4bit is

   signal Sum_uns : unsigned(4 downto 0);

 begin

   Sum_uns <= unsigned(('0' & A)) + unsigned(('0' & B));

   Sum  <= std_logic_vector(Sum_uns(3 downto 0));
   Cout <= Sum_uns(4);

end architecture;
```

A 5-bit unsigned signal is defined to hold the sum and carry.

Adding leading 0's to the inputs enables an assignment to "Sum_uns".

Converting the inputs to unsigned allows the "+" operator to be used.

Finally, the 5-bit vector is broken into its individual Sum and Cout parts.



Since no delay was included in the behavioral model, the outputs are produced instantaneously.

Example 12.14
Behavioral model of a 4-bit adder in VHDL

CONCEPT CHECK

CC12.1 Does a binary adder behave differently when it's operating on unsigned vs. two's complement numbers? Why or why not?

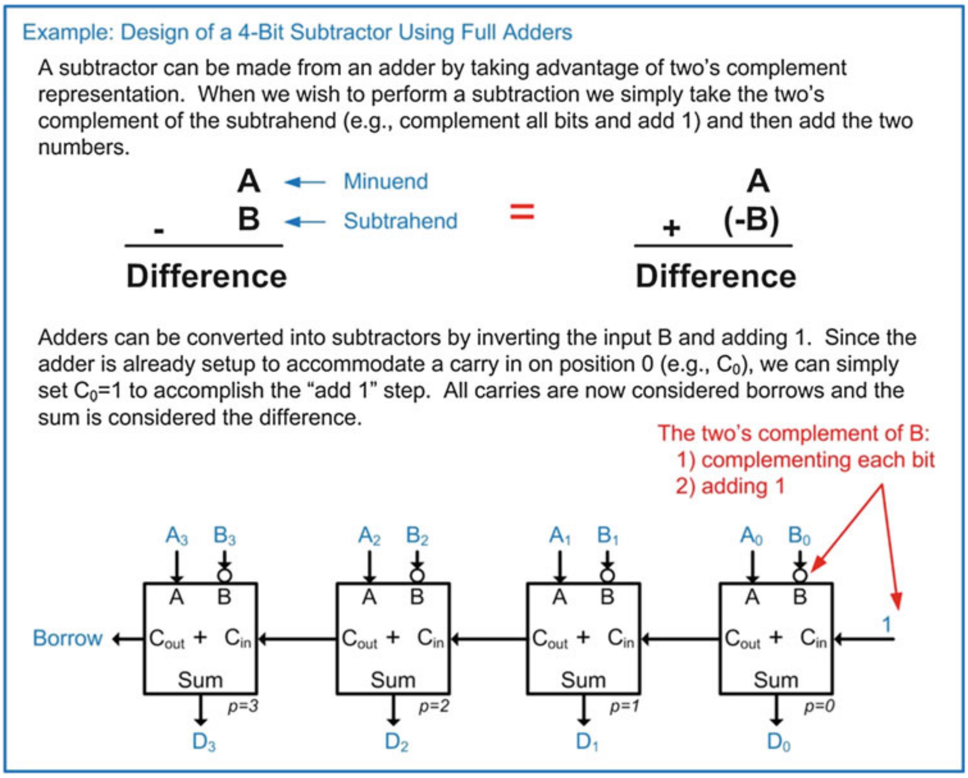
A) Yes. The adder needs to keep track of the sign bit, thus extra circuitry is needed.

B) No. The binary addition is identical. It is up to the designer to handle how the two's complement codes are interpreted and whether two's complement overflow occurred using a separate system.

12.2 Subtraction

Binary subtraction can be accomplished by building a dedicated circuit using a similar design approach as just described for adders. A more effective approach is to take advantage of two's complement representation in order to reuse existing adder circuitry. Recall that taking the two's complement of a number will produce an equivalent magnitude number, but with the opposite sign (i.e., positive to negative or negative to positive). This means that all that is required to create a subtractor from an adder is to first take the two's complement of the subtrahend input. Since the steps to take the two's complement of a number involve complementing each of the bits in the number and then adding

1, the logic required is relatively simple. Example 12.15 shows a 4-bit subtractor using full adders. The subtrahend B is inverted prior to entering the full adders. Also, the carry in bit C_0 is set to 1. This handles the “adding 1” step of the two’s complement. All of the carries in the circuit are now treated as *borrows* and the sum is now treated as the *difference*.



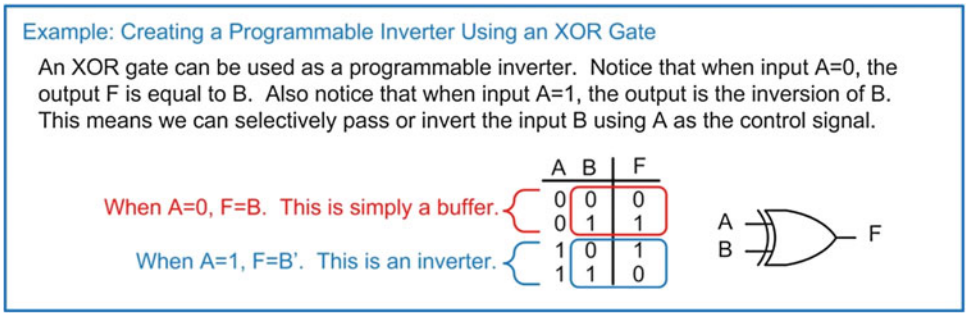
Example: Design of a 4-Bit Subtractor Using Full Adders

A subtractor can be made from an adder by taking advantage of two’s complement representation. When we wish to perform a subtraction we simply take the two’s complement of the subtrahend (e.g., complement all bits and add 1) and then add the two numbers.

Adders can be converted into subtractors by inverting the input B and adding 1. Since the adder is already setup to accommodate a carry in on position 0 (e.g., C_0), we can simply set C_0=1 to accomplish the “add 1” step. All carries are now considered borrows and the sum is considered the difference.

Example 12.15
Design of a 4-bit subtractor using full adders

A programmable adder/subtractor can be created with the use of a programmable inverter and a control signal. The control signal will selectively invert B and also change the C_0 bit between a 0 (for adding) and a 1 (for subtracting). Example 12.16 shows how an XOR gate can be used to create a programmable inverter for use in a programmable adder/subtractor circuit.



Example: Creating a Programmable Inverter Using an XOR Gate

An XOR gate can be used as a programmable inverter. Notice that when input A=0, the output F is equal to B. Also notice that when input A=1, the output is the inversion of B. This means we can selectively pass or invert the input B using A as the control signal.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

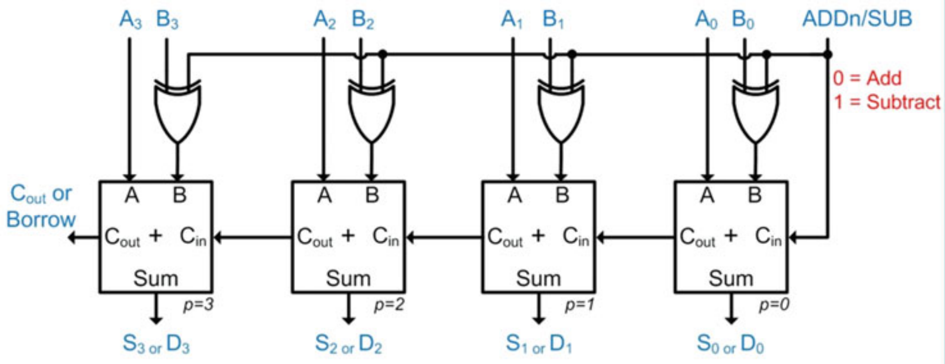
When A=0, F=B. This is simply a buffer.
When A=1, F=B’. This is an inverter.
A
B
F

Example 12.16
Creating a programmable inverter using an XOR Gate

We can now define a control signal called (ADDn/SUB) that will control whether the circuit performs addition or subtraction. Example 12.17 shows the architecture of a 4-bit programmable adder/subtractor. It should be noted that this programmability adds another level of logic to the circuit, thus increasing its delay. The programmable architecture in Example 12.17 is shown for a ripple carry adder; however, this approach works equally well for a carry look ahead adder architecture.

Example: Design of a 4-Bit Programmable Adder/Subtractor

The control signal "ADDn/SUB" is used to select whether the circuit performs addition (ADDn/SUB=0) or subtraction (ADDn/SUB=1). When in subtraction mode, the XOR gates invert the subtrahend B and add 1 to the first adder stage. These steps take the two's complement of B and allow an add operation to conduct subtraction.



Example 12.17
Design of a 4-bit programmable adder/subtractor

When using two's complement representation in arithmetic, care must be taken to monitor for two's complement overflow. Recall that when using two's complement representation, the number of bits of the numbers is fixed (e.g., 4-bits) and if a carry/borrow out is generated, it is ignored. This means that the Cout bit does not indicate whether two's complement overflow occurred. Instead, we must construct additional circuitry to monitor the arithmetic operations for overflow. Recall from Chap. 2 that two's complement overflow occurs in any of these situations:

- The sum of like signs results in an answer with opposite sign
  (i.e., Positive + Positive = Negative or Negative + Negative = Positive).
- The subtraction of a positive number from a negative number results in a positive number
  (i.e., Negative – Positive = Positive).
- The subtraction of a negative number from a positive number results in a negative number
  (i.e., Positive – Negative = Negative).

The construction of circuitry for these conditions is straightforward since the sign bit of all numbers involved in the operation indicates whether the number is positive or negative. The sign bits of the input arguments and the output are fed into combinational logic circuitry that will assert for any of the above conditions. These signals are then logically combined to create two's complement overflow signal.

CONCEPT CHECK

CC12.2 What modifications can be made to the programmable adder/subtractor architecture so that it can be used to take the 2's complement of a number?

A) Remove the input A.

B) Add an additional control signal that will cause the circuit to ignore A and just perform a complement on B and then add 1.

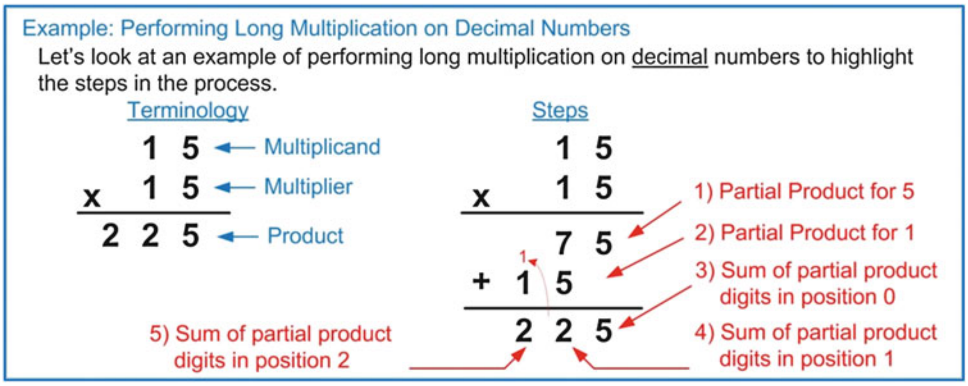
C) Add an additional 1 to the original number using an OR gate on Cin.

D) Set A to 0, put the number to be manipulated on B, and put the system into subtraction mode. The system will then complement the bits on B and then add 1, thus performing two's complement negation.

12.3 Multiplication

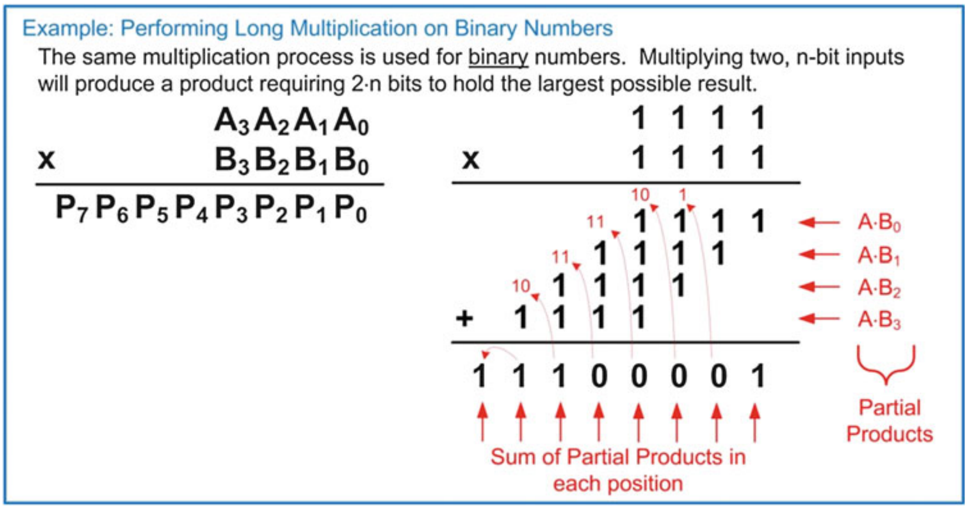
12.3.1 Unsigned Multiplication

Binary multiplication is performed in a similar manner to performing decimal multiplication by hand. Recall the process for long multiplication. First, the two numbers are placed vertically over one another with their least significant digits aligned. The upper number is called the *multiplicand* and the lower number is called the *multiplier*. Next, we multiply each individual digit within multiplier with the entire multiplicand, starting with the least position. The result of this interim multiplication is called the *partial product*. The partial product is recorded with its least significant digit aligned with the corresponding position of the multiplier digit. Finally, all partial products are summed to create the final product of the multiplication. This process is often called the *shift and add* approach. Example 12.18 shows the process for performing long multiplication on decimal numbers highlighting the individual steps.


Example: Performing Long Multiplication on Decimal Numbers
Let's look at an example of performing long multiplication on decimal numbers to highlight the steps in the process.
Terminology
15 Multiplicand
x 15 Multiplier
225 Product
Steps
15
x 15
75
+ 15
225
1
1) Partial Product for 5
2) Partial Product for 1
3) Sum of partial product digits in position 0
4) Sum of partial product digits in position 1
5) Sum of partial product digits in position 2

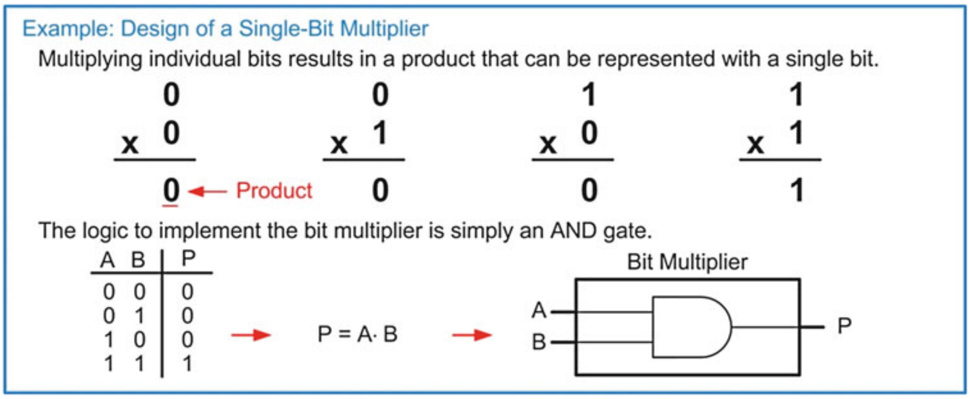
Example 12.18
Performing long multiplication on decimal numbers

Binary multiplication follows this same process. Example 12.19 shows the process for performing long multiplication on binary numbers. Note that the inputs represent the largest unsigned numbers possible using 4-bits, thus producing the largest possible product. The largest product will require 8-bits to be represented. This means that for any multiplication of n-bit inputs, the product will require 2·n bits for the result.


Example: Performing Long Multiplication on Binary Numbers
The same multiplication process is used for binary numbers. Multiplying two, n-bit inputs will produce a product requiring 2·n bits to hold the largest possible result.
$A_3A_2A_1A_0$
x $B_3B_2B_1B_0$
$P_7P_6P_5P_4P_3P_2P_1P_0$
1 1 1 1
x 1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
+ 1 1 1 1
1 1 1 0 0 0 0 1
10
1
11
11
10
$A·B_0$
$A·B_1$
$A·B_2$
$A·B_3$
Partial Products
Sum of Partial Products in each position

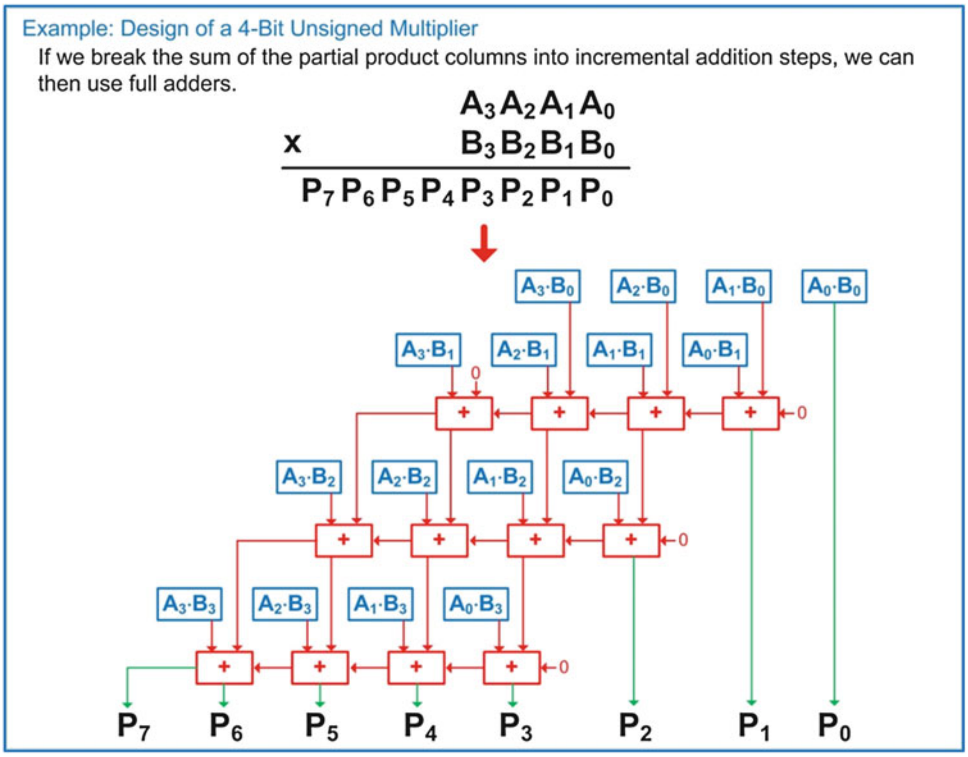
Example 12.19
Performing long multiplication on binary numbers

The first step in designing a binary multiplier is to create circuitry that can compute the product on individual bits. Example 12.20 shows the design of a single-bit multiplier.


Example: Design of a Single-Bit Multiplier
Multiplying individual bits results in a product that can be represented with a single bit.
0 x 0 = 0
Product
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
The logic to implement the bit multiplier is simply an AND gate.
A B | P
0 0 | 0
0 1 | 0
1 0 | 0
1 1 | 1
P = A· B
Bit Multiplier
A
B
P

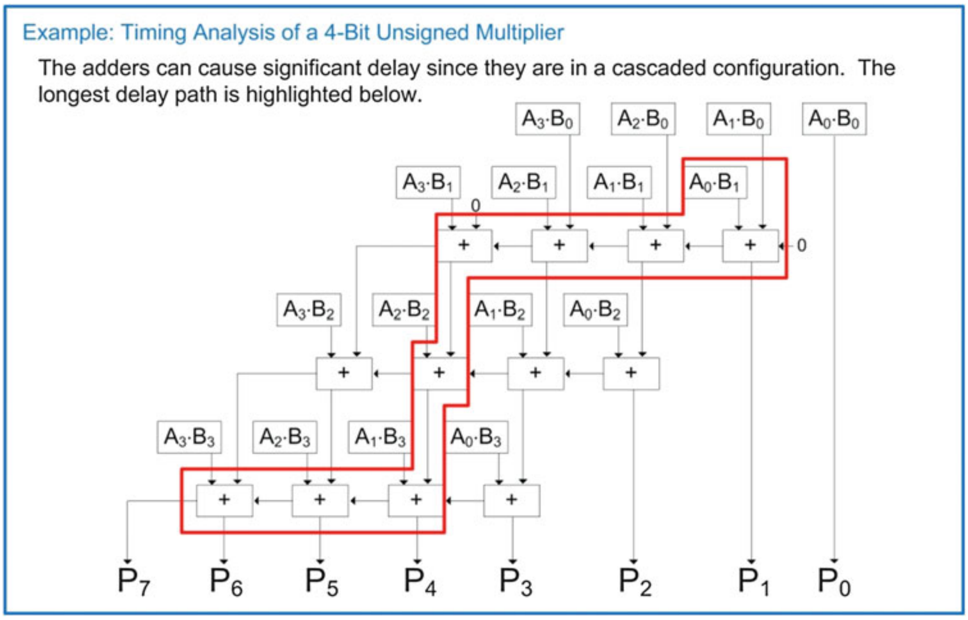
Example 12.20
Design of a single-bit multiplier

We can create all of the partial products in one level of logic by placing an AND gate between each bit pairing in the two input numbers. This will require n^2 AND gates. The next step involves creating adders that can perform the sum of the columns of bits within the partial products. This step is not as straightforward. Notice that in our 4-bit example in Example 12.19 that the number of input bits in the column addition can reach up to 6 (in position 3). It would be desirable to reuse the full adders previously created; however, the existing full adders could only accommodate 3 inputs (A, B, C_{in}). We can take advantage of the associative property of addition to form the final sum incrementally. Example 12.21 shows the architecture of this multiplier. This approach implements a shift and add process to compute the product and is known as a *combinational multiplier* because it is implemented using only combinational logic. Note that this multiplier only handles unsigned numbers.



Example 12.21
Design of a 4-bit unsigned multiplier

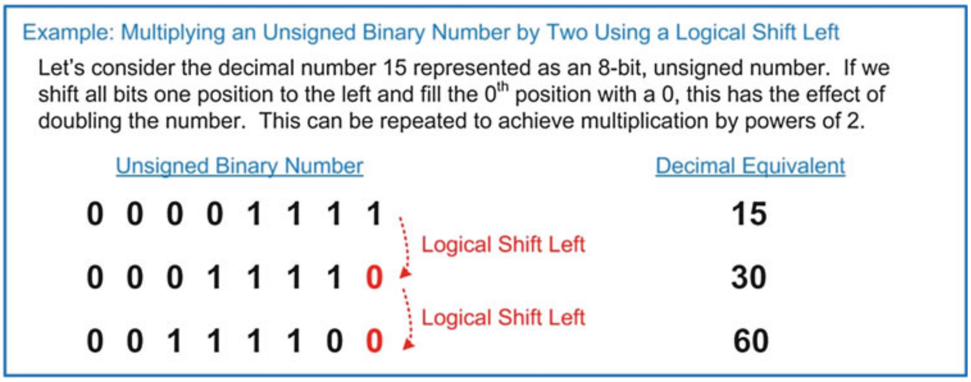
This multiplier can have a significant delay, which is caused by the cascaded full adders. Example 12.22 shows the timing analysis of the combinational multiplier highlighting the worst case path through the circuit.



Example 12.22
Timing analysis of a 4-bit unsigned multiplier

12.3.2 A Simple Circuit to Multiply by Powers of Two

In digital systems, a common operation is to multiply numbers by powers of two. For unsigned numbers, multiplying by two can be accomplished by performing a logical shift left. In this operation, all bits are moved to the next higher position (i.e., left) by one position and filling the 0th position with a zero. This has the effect of doubling the value of the number. This can be repeated to achieve higher powers of two. This process works as long as the resulting product fits within the number of bits available. Example 12.23 shows this procedure.


Example: Multiplying an Unsigned Binary Number by Two Using a Logical Shift Left

Let's consider the decimal number 15 represented as an 8-bit, unsigned number. If we shift all bits one position to the left and fill the 0th position with a 0, this has the effect of doubling the number. This can be repeated to achieve multiplication by powers of 2.

Unsigned Binary Number	Decimal Equivalent
0 0 0 0 1 1 1 1	15
0 0 0 1 1 1 1 0	30
0 0 1 1 1 1 0 0	60

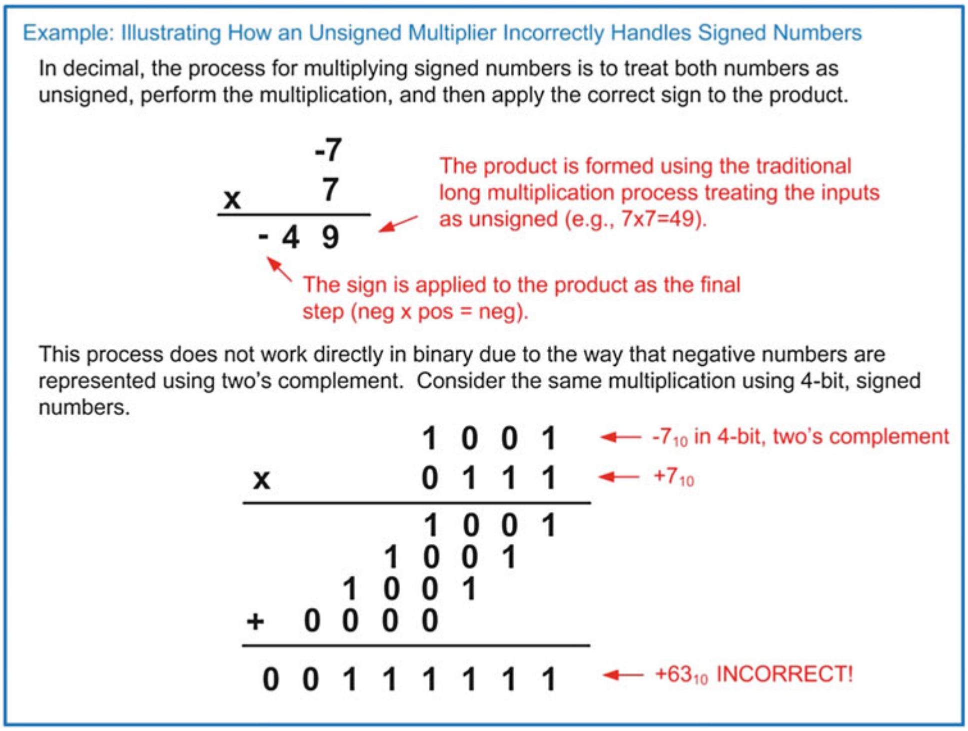
Example 12.23
Multiplying an unsigned binary number by two using a logical shift left

12.3.3 Signed Multiplication

When performing multiplication on signed numbers, it is desirable to reuse the unsigned multiplier in Example 12.21. Let's examine if this is possible. Recall in decimal multiplication that the inputs are multiplied together independent of their sign. The sign of the product is handled separately following these rules:

- A positive number times a positive number produces a positive number.
- A negative number times a negative number produces a positive number.
- A positive number times a negative number produces a negative number.

This process does not work properly in binary due to the way that negative numbers are represented with two's complement. Example 12.24 illustrates how an unsigned multiplier incorrectly handles signed numbers.



Example: Illustrating How an Unsigned Multiplier Incorrectly Handles Signed Numbers

In decimal, the process for multiplying signed numbers is to treat both numbers as unsigned, perform the multiplication, and then apply the correct sign to the product.

```
   -7
x   7
-----
 - 4 9
```

The product is formed using the traditional long multiplication process treating the inputs as unsigned (e.g., 7x7=49).

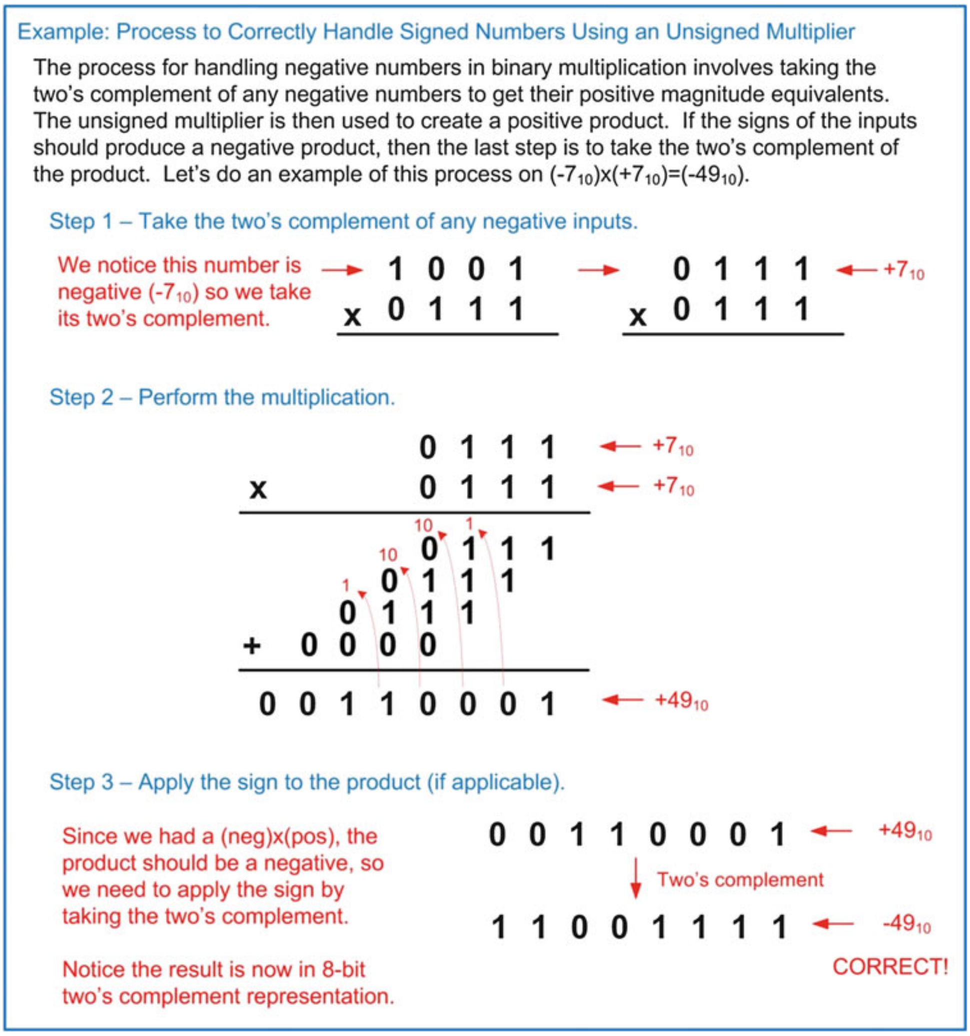
The sign is applied to the product as the final step (neg x pos = neg).

This process does not work directly in binary due to the way that negative numbers are represented using two's complement. Consider the same multiplication using 4-bit, signed numbers.

```
                1 0 0 1   ← -7₁₀ in 4-bit, two's complement
x               0 1 1 1   ← +7₁₀
  -----------------------
                1 0 0 1
              1 0 0 1
            1 0 0 1
+         0 0 0 0
  -----------------------
    0 0 1 1 1 1 1 1       ← +63₁₀ INCORRECT!
```

Example 12.24
Illustrating how an unsigned multiplier incorrectly handles signed numbers

Instead of building a dedicated multiplier for signed numbers, we can add functionality to the unsigned multiplier previously presented to handle negative numbers. The process involves first identifying any negative numbers. If a negative number is present, the two's complement is taken on it to produce its equivalent magnitude, positive representation. The multiplication is then performed on the positive values. The final step is to apply the correct sign to the product. If the product should be negative due to one of the inputs being negative, the sign is applied by taking the two's complement on the final result. This creates a number that is now in 2·n two's complement format. Example 12.25 shows an illustration of the process to correctly handle signed numbers using an unsigned multiplier.



Example: Process to Correctly Handle Signed Numbers Using an Unsigned Multiplier

The process for handling negative numbers in binary multiplication involves taking the two's complement of any negative numbers to get their positive magnitude equivalents. The unsigned multiplier is then used to create a positive product. If the signs of the inputs should produce a negative product, then the last step is to take the two's complement of the product. Let's do an example of this process on $(-7_{10})\text{x}(+7_{10})=(-49_{10})$.

Step 1 – Take the two's complement of any negative inputs.

We notice this number is negative (-7_{10}) so we take its two's complement.

Step 2 – Perform the multiplication.

Step 3 – Apply the sign to the product (if applicable).

Since we had a (neg)x(pos), the product should be a negative, so we need to apply the sign by taking the two's complement.

Notice the result is now in 8-bit two's complement representation.

Example 12.25
Process to correctly handle signed numbers using an unsigned multiplier

CONCEPT CHECK

CC12.3 Will the AND gates used to compute the partial products in a binary multiplier ever experience an issue with fan-in as the size of the multiplier increases?

A) Yes. When the number of bits of the multiplier arguments exceed the fan-in specification of the AND gates used for the partial products, a fan-in issue has occurred.

B) No. The number of inputs of the AND gates performing the partial products will always be two, regardless of the size of the input arguments to the multiplier.

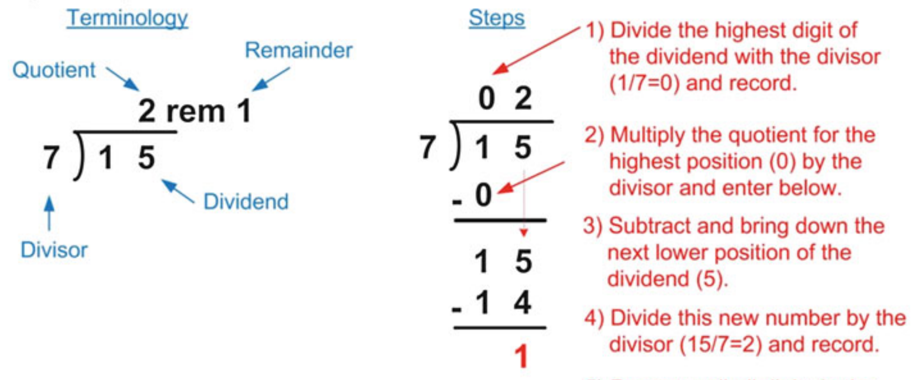
12.4 Division

12.4.1 Unsigned Division

There are a variety of methods to perform division, each with trade-offs between area, delay, and accuracy. To understand the general approach to building a divider circuit, let's focus on how a simple iterative divider can be built. Basic division yields a *quotient* and a *remainder*. The process begins by checking whether the *divisor* goes into the highest position digit in the *dividend*. The number of times this dividend digit can be divided is recorded as the highest position value of the quotient. Note that when performing division by hand, we typically skip over the condition when the result of these initial operations are zero, but when breaking down the process into steps that can be built with logic circuits, each step needs to be highlighted. The first quotient digit is then multiplied with the divisor and recorded below the original dividend. The next lower position digit of the dividend is brought down and joined with the product from the prior multiplication. This forms a new number to be divided by the divisor to create the next quotient value. This process is repeated until each of the quotient digits have been created. Any value that remains after the last subtraction is recorded as the remainder. Example 12.26 shows the long division process on decimal numbers highlight each incremental step.

Example: Performing Long Division on Decimal Numbers

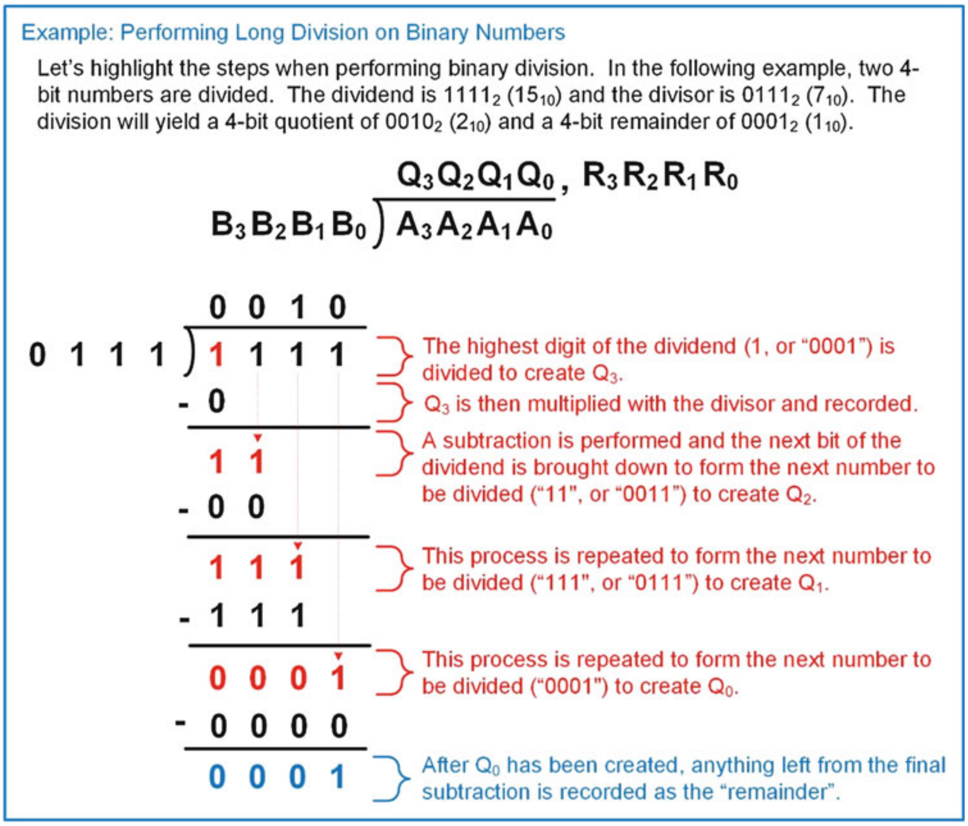
Let's look at an example of performing long division on decimal numbers to highlight the steps in the process.



1) Divide the highest digit of the dividend with the divisor (1/7=0) and record.
2) Multiply the quotient for the highest position (0) by the divisor and enter below.
3) Subtract and bring down the next lower position of the dividend (5).
4) Divide this new number by the divisor (15/7=2) and record.
5) Repeat until all digits in the dividend have been evaluated.
6) If anything remains, it is recorded as the "remainder".

Example 12.26
Performing long division on decimal numbers

Long division in binary follows this same process. Example 12.27 shows the long division process on two 4-bit, unsigned numbers. This division results in a 4-bit quotient and a 4-bit remainder.



Example: Performing Long Division on Binary Numbers

Let's highlight the steps when performing binary division. In the following example, two 4-bit numbers are divided. The dividend is 1111_2 (15_{10}) and the divisor is 0111_2 (7_{10}). The division will yield a 4-bit quotient of 0010_2 (2_{10}) and a 4-bit remainder of 0001_2 (1_{10}).

$$\begin{array}{r} Q_3Q_2Q_1Q_0,\ R_3R_2R_1R_0 \\ B_3B_2B_1B_0 \overline{)A_3A_2A_1A_0} \end{array}$$

The highest digit of the dividend (1, or "0001") is divided to create Q_3.

Q_3 is then multiplied with the divisor and recorded.

A subtraction is performed and the next bit of the dividend is brought down to form the next number to be divided ("11", or "0011") to create Q_2.

This process is repeated to form the next number to be divided ("111", or "0111") to create Q_1.

This process is repeated to form the next number to be divided ("0001") to create Q_0.

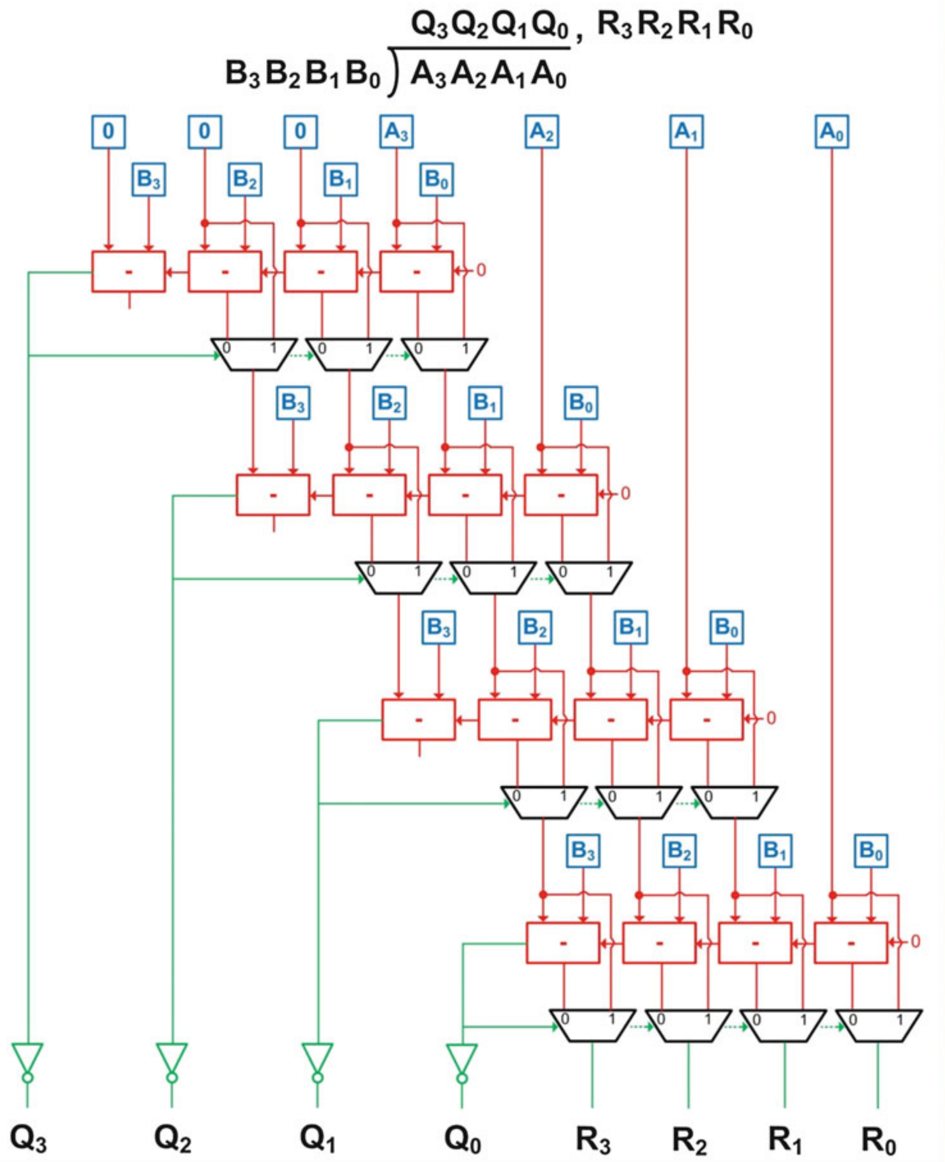
After Q_0 has been created, anything left from the final subtraction is recorded as the "remainder".

Example 12.27
Performing long multiplication on binary numbers

When building a divider circuit using combinational logic, we can accomplish the computation using a series of iterative subtractors. Performing division is equivalent to subtracting the divisor from the interim dividend. If the subtraction is positive, then the divisor went into the dividend and the quotient is a 1. If the subtraction yields a negative number, then the divisor did not go into the interim dividend and the quotient is 0. We can use the borrow out of a subtraction chain to provide the quotient. This has the advantage that the difference has already been calculated for the next subtraction. A multiplexer is used to select whether the difference is used in the next subtraction ($Q = 0$), or if the interim divisor is simply brought down ($Q = 1$). This inherently provides the functionality of the multiplication step in long division. Example 12.28 shows the architecture of a 4-bit, unsigned divider based on the iterative subtraction approach. Notice that when the borrow out of the 4-bit subtractor chain is a 0, it indicates that the subtraction yielded a positive number. This means that the divisor went into the interim dividend once. In this case, the quotient for this position is a 1. An inverter is required to produce the correct polarity of the quotient. The borrow out is also fed into the multiplexer stage as the select line to pass the difference to the next stage of subtractors. If the borrow out of the 4-bit subtractor chain is a 1, it indicates that the subtraction yielded a negative number. In this case, the quotient is a 0. This also means that the difference calculated is garbage and should not be used. The multiplexer stage instead selects the interim dividend as the input to the next stage of subtractors.

Example: Design of a 4-Bit Unsigned Divider Using a Series of Iterative Subtractors

The following architecture shows a combinational divider that uses a series of iterative subtractions to determine the quotient and remainder.

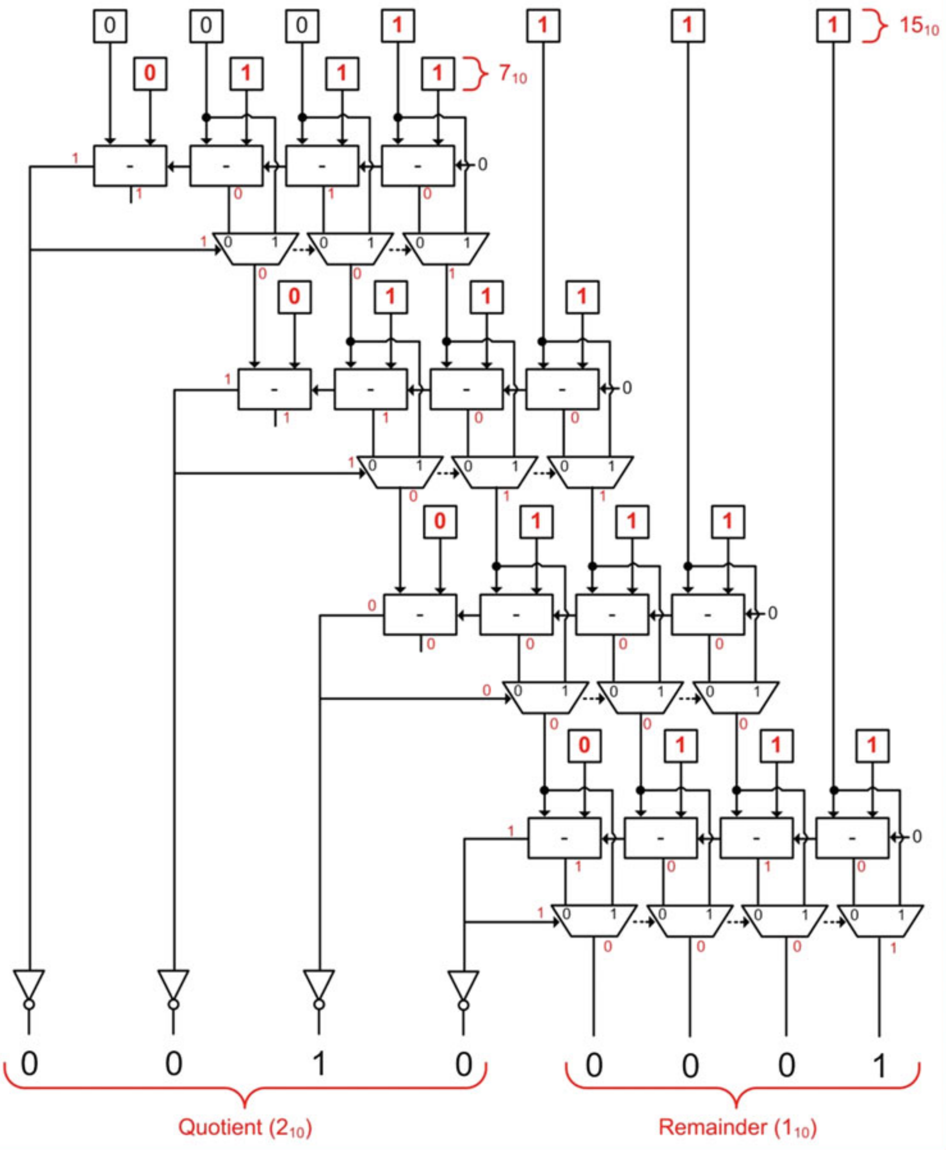


Example 12.28
Design of a 4-bit unsigned divider using a series of iterative subtractors

To illustrate how this architecture works, Example 12.29 walks through each step in the process where 1111_2 (15_{10}) is divided by 0111_2 (7_{10}). In this example, the calculations propagate through the logic stages from top to bottom in the diagram.

Example: Dividing 1111_2 (15_{10}) by 0111_2 (7_{10}) Using the Iterative Subtraction Architecture

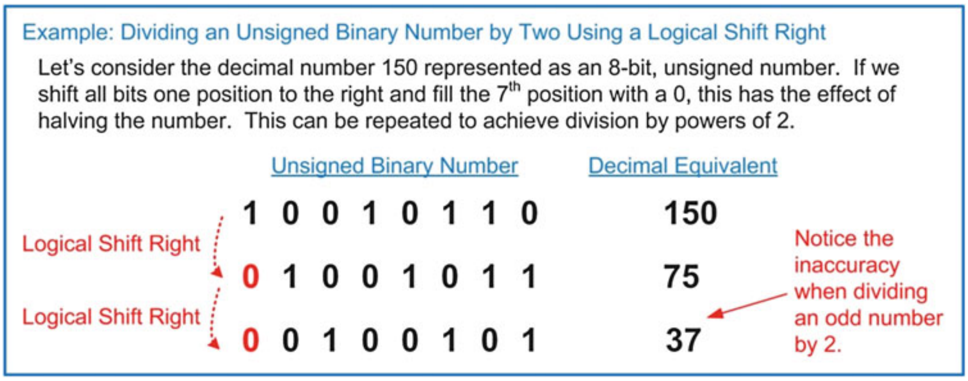
This division will yield a 4-bit quotient of 0010_2 (2_{10}) and a 4-bit remainder of 0001_2 (1_{10}).



Example 12.29
Dividing 1111_2 (15_{10}) by 0111_2 (7_{10}) using the iterative subtraction architecture

12.4.2 A Simple Circuit to Divide by Powers of Two

For unsigned numbers, dividing by two can be accomplished by performing a logical shift right. In this operation, all bits are moved to the next lower position (i.e., right) by one position and then filling the highest position with a zero. This has the effect of halving the value of the number. This can be repeated to achieve higher powers of two. This process works until no more ones exist in the number and the result is simply all zeros. Example 12.30 shows this process.



Example: Dividing an Unsigned Binary Number by Two Using a Logical Shift Right

Let's consider the decimal number 150 represented as an 8-bit, unsigned number. If we shift all bits one position to the right and fill the 7th position with a 0, this has the effect of halving the number. This can be repeated to achieve division by powers of 2.

	Unsigned Binary Number	Decimal Equivalent
	1 0 0 1 0 1 1 0	150
Logical Shift Right	0 1 0 0 1 0 1 1	75
Logical Shift Right	0 0 1 0 0 1 0 1	37

Example 12.30
Dividing an unsigned binary numbers by two using a logical shift right

12.4.3 Signed Division

When performing division on signed numbers, a similar strategy as in signed multiplication is used. The process involves first identifying any negative numbers. If a negative number is present, the two's complement is taken on it to produce its equivalent magnitude, positive representation. The division is then performed on the positive values. The final step is to apply the correct sign to the divisor and quotient. This is accomplished by taking the two's complement if a negative number is required. The rules governing the polarities of the quotient and remainders are:

- The quotient will be negative if the input signs are different (i.e., pos/neg or neg/pos).
- The remainder has the same sign as the dividend.

Concept Check

CC12.4 Could a shift register help reduce the complexity of a combinational divider circuit? How?

A) Yes. Instead of having redundant circuits holding the different shifted versions of the divisor, a shift register could be used to hold and shift the divisor after each subtraction.

B) No. A state machine would then be needed to control the divisor shifting, which would make the system even more complex.

Summary

❖ Binary arithmetic is accomplished using combinational logic circuitry. These circuits tend to be the largest circuits in a system and have the longest delay. Arithmetic circuits are often broken up into interim calculations in order to reduce the overall delay of the computation.

❖ A *ripple carry adder* performs addition by reusing lower-level components that each performs a small part of the computation. A full adder is made from two half adders and a ripple carry adder is made from a chain of full adders. This approach simplifies the design of the adder but leads to long delay times since the carry from each sum must ripple to the next higher position's addition before it can complete.

- A *carry look ahead adder* attempts to eliminate the linear dependence of delay on the number of bits that exists in a ripple carry adder. The carry look ahead adder contains dedicated circuitry that calculates the carry bits for each position of the addition. This leads to a more constant delay as the width of the adder increases.
- A binary multiplier can be created in a similar manner to the way multiplication is accomplished by hand using the *shift and add* approach. The partial products of the multiplication can be performed using 2-input AND gates. The sum of the partial products can have more inputs than the typical ripple carry adder can accommodate. To handle this, the additions are performed two bits at a time using a series of adders.
- Division can be accomplished using an iterative subtractor architecture.

Exercise Problems

Section 12.1: Addition

12.1.1 Give the total delay of the full adder shown in Fig. 12.2 if all gates have a delay of 1 ns.

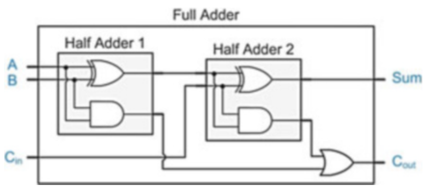


Fig. 12.2
Full Adder Timing Exercise

12.1.2 Give the total delay of the full adder shown in Fig. 12.2 if the XOR gates have delays of 5 ns while the AND and OR gates have delays of 1 ns.

12.1.3 Give the total delay of the 4-bit ripple carry adder shown in Fig. 12.3 if all gates have a delay of 2 ns.

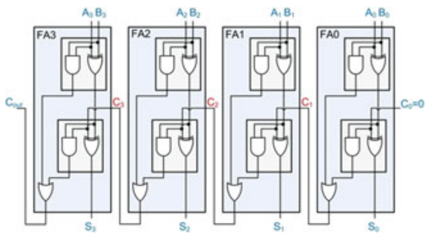


Fig. 12.3
4-Bit RCA Timing Exercise

12.1.4 Give the total delay of the 4-bit ripple carry adder shown in Fig. 12.3 if the XOR gates have delays of 10 ns while the AND and OR gates have delays of 2 ns.

12.1.5 Design a VHDL model for an 8-bit Ripple Carry Adder (RCA) using a structural design approach. This involves creating a half adder (half_adder.vhd), full adder (full_adder.vhd), and then finally a top-level adder (rca.vhd) by instantiating eight full adder components. Model the ripple delay by inserting 1ns of gate delay for the XOR, AND, and OR operators using a delayed signal assignment. The general topology and entity definition for the design are shown in Fig. 12.4. Create a test bench to exhaustively verify this design under all input conditions. The test bench should drive in different values every 30 ns in order to give sufficient time for the signals to ripple through the adder.

```
entity rca_8bit is
  port (A, B : in  std_logic_vector(7 downto 0);
        S    : out std_logic_vector(7 downto 0);
        Cout : out std_logic);
end entity;
```

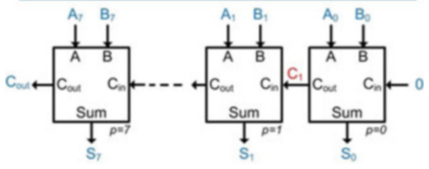


Fig. 12.4
4-Bit RCA Entity

12.1.6 Give the total delay of the 4-bit carry look ahead adder shown in Fig. 12.5 if all gates have a delay of 2 ns.

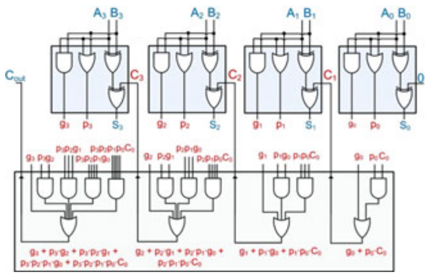


Fig. 12.5
4-Bit CLA Timing Exercise

12.1.7 Give the total delay of the 4-bit carry look ahead adder shown in Fig. 12.5 if the XOR gates have delays of 10ns while the AND and OR gates have delays of 2 ns.

12.1.8 Design a VHDL model for an 8-bit Carry Look Ahead Adder (cla.vhd). The model should instantiate eight modified full adders (mod_full_adder.vhd). The carry look ahead logic should be implemented using concurrent signal assignments with logical operators. Model each level of gate delay as 1ns using delayed signal assignments. The general topology and entity definition for the design are shown in Fig. 12.6. Create a test bench to exhaustively verify this design under all input conditions. The test bench should drive in different values every 30 ns in order to give sufficient time for the signals to propagate through the adder.

```
entity cla_8bit is
  port (A, B  : in  std_logic_vector(7 downto 0);
        S     : out std_logic_vector(7 downto 0);
        Cout  : out std_logic);
end entity;
```

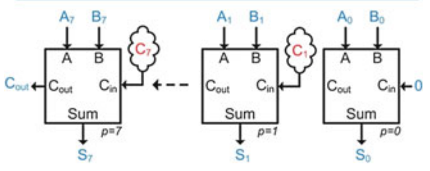


Fig. 12.6
4-Bit CLA Entity

Section 12.2: Subtraction

12.2.1 How is the programmable adder/subtractor architecture shown in Fig. 12.7 analogous to 2's complement arithmetic?

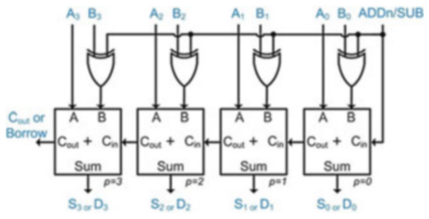


Fig. 12.7
Programmable Adder/Subtractor Block Diagram

12.2.2 Will the programmable adder/subtractor architecture shown in Fig. 12.7 work for negative numbers encoded using signed magnitude or 1's complement?

12.2.3 When calculating the delay of the programmable adder/subtractor architecture shown in Fig. 12.7 does the delay of the XOR gate that acts as the programmable inverter need to be considered?

12.2.4 Design a VHDL model for an 8-bit, programmable adder/subtractor. The design will have an input called "ADDn_SUB" that will control whether the system behaves as an adder (0) or as a subtractor (1). The design should operate on two's complement signed numbers. The result of the operation(s) will appear on the port called "Sum_Diff". The model should assert the output "Cout" when an addition creates a carry or when a subtraction creates a borrow. The circuit will also assert the output Vout when either operation results in two's complement overflow. The entity definition and block diagram for the system is shown in Fig. 12.8. Create a test bench to exhaustively verify this design under all input conditions.

```
entity add_n_sub_8bit is
  port (A, B      : in  std_logic_vector(7 downto 0);
        ADDn_SUB  : in  std_logic;
        Sum_Diff  : out std_logic_vector(7 downto 0);
        Cout_Bout : out std_logic;
        Vout      : out std_logic);
end entity;
```

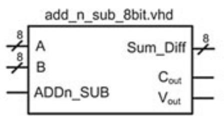


Fig. 12.8
Programmable Adder/Subtractor Entity

Section 12.3: Multiplication

12.3.1 Give the total delay of the 4-bit unsigned multiplier shown in Fig. 12.9 if all gates have a delay of 1ns. The addition is performed using a ripple carry adder.

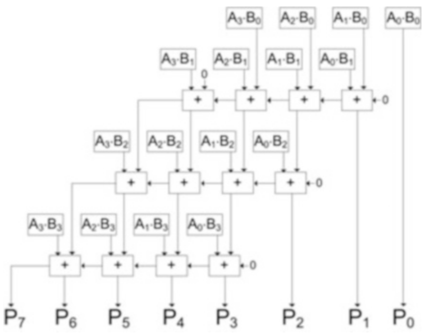


Fig. 12.9
4-Bit Unsigned Multiplier Block Diagram

12.3.2 For the 4-bit unsigned multiplier shown in Fig. 12.9, how many levels of logic does it take to compute all of the partial products?

12.3.3 For the 4-bit unsigned multiplier shown in Fig. 12.9, how many AND gates are needed to compute the partial products?

12.3.4 For the 4-bit unsigned multiplier shown in Fig. 12.9, how many total AND gates are used if the additions are implemented using full adders made of half adders?

12.3.5 Based on the architecture of a unsigned multiplier in Fig. 12.9, how many AND gates are needed to compute the partial products if the inputs are increased to 8-bits?

12.3.6 For an 8-bit multiplier, how many bits are needed to represent the product?

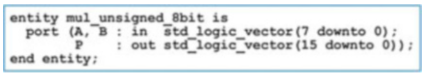
12.3.7 For an 8-bit *unsigned* multiplier, what is the largest value that the product can ever take on? Give your answer in decimal.

12.3.8 For an 8-bit *signed* multiplier, what is the largest value that the product can ever take on? Give your answer in decimal.

12.3.9 For an 8-bit *signed* multiplier, what is the smallest value that the product can ever take on? Give your answer in decimal.

12.3.10 What is the maximum number of times that a 4-bit unsigned multiplicand can be multiplied by two using the *logical shift left* approach before the product is too large to be represented by an 8-bit-product? Hint: The maximum number of times this operation can be performed corresponds to when the multiplicand starts at its lowest possible nonzero value (i.e., 1).

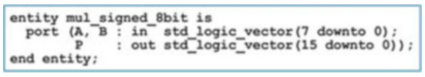
12.3.11 Design a VHDL model for an 8-bit unsigned multiplier using whatever modeling approach you wish. Create a test bench to exhaustively verify this design under all input conditions. The entity definition for this multiplier is given in Fig. 12.10. Hint: Consider converting the inputs into type integers and then performing the multiplication using the "*" operation. The result of this operation will need to be an internal signal also of type interger. The integer product can then be converted back to a 16-bit std_logic_vector. Make sure to apply a *range* to your internal integers.



```
entity mul_unsigned_8bit is
  port (A, B : in  std_logic_vector(7 downto 0);
        P    : out std_logic_vector(15 downto 0));
end entity;
```

Fig. 12.10
8-Bit Unsigned Multiplier Entity

12.3.12 Design a VHDL model for an 8-bit signed multiplier using whatever modeling approach you wish. Create a test bench to exhaustively verify this design under all input conditions. The entity definition for this multiplier is given in Fig. 12.11. Hint: Consider converting the inputs into type integers and then performing the multiplication using the "*" operation. The result of this operation will need to be an internal signal also of type interger. The integer product can then be converted back to a 16-bit std_logic_vector. Make sure to apply a *range* to your internal integers.



```
entity mul_signed_8bit is
  port (A, B : in  std_logic_vector(7 downto 0);
        P    : out std_logic_vector(15 downto 0));
end entity;
```

Fig. 12.11
8-Bit Signed Multiplier Entity

Section 12.4: Division

12.4.1 For a 4-bit divider, how many bits are needed for the quotient?

12.4.2 For a 4-bit divider, how many bits are needed for the remainder?

12.4.3 Explain the basic concept of the iterative-subtractor approach to division.

12.4.4 For the 4-bit divider shown in Example 12.28, estimate the total delay assuming all gates have a delay of 1 ns.