Digital Arithmetic and VHDL

Michael Wirthlin

March 10, 2014

1 Introduction

One of the most important set of operations performed in digital logic are basic arithmetic operations. Simple operations such as addition (including increment which is an addition with +1), subtraction, multiplication, and division are performed in most digital circuits. Because digital arithmetic is such an important part of digital systems, it is essential to have a firm understanding of digital arithmetic operations. The textbook we are using ([1]) does not provide a thorough discussion of digital arithmetic as the text assumes the reader has a strong, working knowledge of these concepts. The reader may want to refer to other books for more details on the fundamental concepts of binary arithmetic [2]. This reading supplement was written to provide a review of important digital arithmetic concepts and to supplement the textbook with important new concepts that may have not been covered in your previous digital hardware courses.

This reading supplement also discusses important concepts regarding digital arithmetic and how it relates to hardware description languages (HDL) like VHDL. VHDL imposes a number of rules and assumptions in regards to digital arithmetic that must be understood to properly implement digital arithmetic operations. These rules and concepts are not included in the textbook and will be described here.

This supplement will review and introduce the following concepts:

- Binary Representations,
- Binary Arithmetic Operations,
- Hardware Implementations of Binary Arithmetic,
- The Timing of Arithmetic Circuits, and
- Generating Binary Arithmetic circuits with VHDL.

2 Binary Representations and Arithmetic

A binary signal can represent one of two values: 0 or 1. A set of discrete binary signals can be combined together to represent numerical values. If N binary signals are grouped together into a multi-bit signal, 2^N different numerical values can be represented. The N-bit binary value can be interpreted in a number of different ways depending upon the "number representation" that is chosen. Two binary representations will be presented here: the unsigned binary representation and the two's complement representation. Other binary representations that are sometimes used are the "ones complement" and the "sign-magnitude" representations. Neither of these representations are supported directly in VHDL and thus will not be discussed in this document.

2.1 Unsigned Binary Representation

The unsigned binary representation can represent zero and positive numbers (i.e., much like the VHDL "Natural" type). If N binary signals are grouped together into a multi-bit signal named A and the individual bits of the signal A are labeled a_i (with the least significant bit indexed as 0 and the most significant bit indexed as a_{N-1}), the unsigned integer value of this N-bit signal can be specified as follows:

$$Value = \sum_{i=0}^{N-1} a_i 2^i. \tag{1}$$

For example, the eight-bit binary number "01101101" represents the integer number 109 as follows:

$$\begin{aligned} Value &= 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 0 + 64 + 32 + 0 + 8 + 4 + 0 + 1 \\ &= 109 \end{aligned}$$

The range of unsigned integer values that can be represented by N bits is $0 \leq value \leq 2^N - 1$. For example, 8 bits can represent the numbers $0 \leq value \leq 2^8 - 1$ or $0 \leq value \leq 255$. Additional bits can be used to represent larger unsigned integer numbers.

2.2 Two's Complement Representation

There are a number of number representations that represent binary signals as signed numbers. One of the most common is a representation called "Two's Complement Representation". The value of a binary number represented in two's complement format is as follows:

$$Value = (-1)a_{(N-1)} 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i. \tag{2}$$

This representation obtained its name because of the "two's complement" operation that accompanies this representation (this operation will be described in more detail below).

An important advantage of the two's complement representation is that it can be used to represent negative numbers. As seen in Equation 2, a negative value can be represented when the most significant bit (a_{N-1}) is '1'. For example, the eight-bit binary number "11101101" represents the integer number -19 as follows:

$$\begin{aligned} Value &= (-1)(1) \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= -128 + 64 + 32 + 0 + 8 + 4 + 0 + 1 \\ &= -128 + 109 \\ &= -19 \end{aligned}$$

Positive numbers are represented when the most significant bit is '0'. For example, the eight-bit binary number "00010011" represents the integer number +19 as follows:

$$\begin{aligned} Value &= (-1)(0) \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 0 + 0 + 0 + 16 + 0 + 0 + 2 + 1 \\ &= +19 \end{aligned}$$

The range of integer values that can be represented by N bits with the two's complement representation is $-2^{N-1} \leq value \leq 2^{N-1} - 1$. For example, 8 bits can represent the numbers $-128 \leq value \leq +127$.

The existence of more than one numerical representation system suggest that the same binary value can be represented as two different arithmetic numbers. The two representations described here can be visualized in the form of a number wheel as seen in Figure 1. Sixteen four-bit binary numbers are shown on the inside of the number circle. The numerical representation of each of these sixteen values is shown on the outside of the number circle – the unsigned representation is shown closest to the circle and the signed representation is shown on the outside of the unsigned representation. Note that some binary numbers have two different numerical representations. For example, the binary number "1100" is represented as the number +11 in the unsigned binary format and -5 in the two's complement representation.

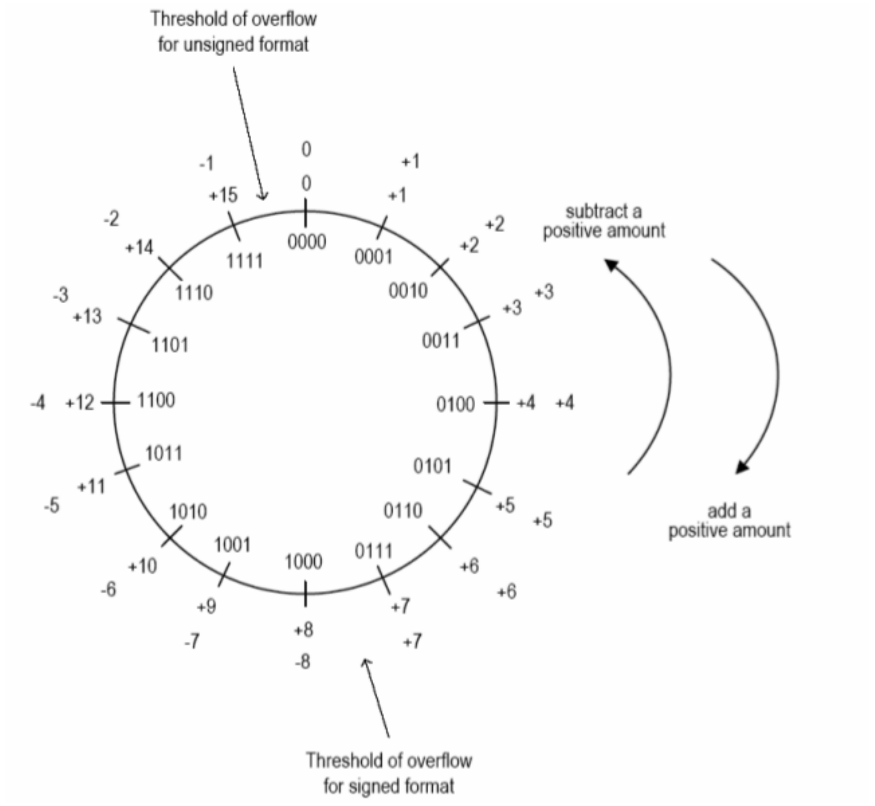


Figure 1: Binary Number Wheel Representation (from Chu, Figure 7.6).

2.2.1 Two's Complement

Each two's complement number has a corresponding "complement", called the Two's Complement[1], and is defined as the complement with respect to 2^N:

$$\text{Two's Complement(A)} = A^* = 2^N - A \tag{3}$$

When you perform this operation for a negative number, the resulting number will be greater than 2^N. Since there are only N bits in the number, the resulting complement is performed using modulo 2^N arithmetic (i.e., $A^* = (2^N - A)\, mod\, 2^N$).

The two's complement operation behaves much like the arithmetic negate operation – this operation is used to change the sign of positive numbers to negative and negative numbers to positive while keeping the absolute value the same. The two's complement of a two's complement number is computed by taking the "complement" of a number with respect to 2^N as follows:

$$\text{Two's Complement(A)} = 2^N - A = 2^N - \left[(-1)a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i\right] \tag{4}$$

For example, the two's complement of the number "11101101" (-19) is

$$\begin{aligned}\text{Two's Complement(11101101)} &= 2^N - \left[(-1)a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i\right] \\ &= 2^8 - (-2^7 + 2^6 + 2^5 + 2^3 + 2^2 + 2^0)\end{aligned}$$

[1]The term "two's complement" can be a bit confusing because it can refer to both a number format and an operation.

$$
\begin{aligned}
&= 256 - (-128 + 64 + 32 + 8 + 4 + 1) \\
&= 256 - (-19) \\
&= 256 + 19 \\
&= +19 \text{ (i.e., 275 modulo 256)}
\end{aligned}
$$

A common "trick" for easily performing the two's complement operation on a number is the "invert and add one" rule. This method involves first inverting each bit of the two's complement number (often called the "one's complement"). After inverting each bit, add '1' to the resulting number using unsigned, binary arithmetic. The following example demonstrators the "invert and add one" rule for the two's complement number "11101101":

$$
\begin{aligned}
\text{invert}(11101101) + 1 &= 00010010 + 1 \\
&= 00010011 \\
&= 2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19
\end{aligned}
$$

The mathematical derivation of this "invert and one" method is shown below:

$$
\begin{aligned}
A^* &= 2^N - A \\
&= \left(\sum_{i=0}^{N-1} 2^i + 1 \right) - A \\
&= \left(\sum_{i=0}^{N-1} 2^i + 1 \right) - \left((-1) a_{(N-1)} 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i \right) \\
&= a_{(N-1)} 2^{N-1} + \left(\sum_{i=0}^{N-1} 2^i + 1 - \sum_{i=0}^{N-2} a_i 2^i \right) \\
&= a_{(N-1)} 2^{N-1} + \sum_{i=0}^{N-2} (2^i - a_i 2^i) + 2^{N-1} + 1 \\
&= a_{(N-1)} 2^{N-1} + 2^{N-1} + \sum_{i=0}^{N-2} (1 - a_i) 2^i + 1 \\
&= \left(1 - (-1) a_{(N-1)} \right) 2^{N-1} + \sum_{i=0}^{N-2} (1 - a_i) 2^i + 1
\end{aligned}
$$

Note that each bit, a_i, is inverted (1-a_i) and one is added to the result.

2.2.2 Sign Extension

When performing arithmetic operations, it is sometimes necessary to increase the numerical range of a binary number. The numerical range of a binary number can be increased by adding additional bits to the number. When using the unsigned binary representation, additional range can be provided to a binary number by adding additional '0' bits in the most significant bit positions. For example, the number 19 can be represented in binary with six bits as follows: "010011". The same number can be represented in binary with eight bits by adding two additional '0' digits in the most significant bit positions as follows: "00010011".

The range of two's complement numbers can also be increased by adding additional bits in the most significant bit positions. In this representation, however, the value of the new digits must all be the same as the most significant digit of the original two's complement number. This process of adding additional sign digits in the most-significant bit position for two's complement nubmers is called "Sign Extension".

For example, -19 is represented with six bits in two's complement representation as follows: "101101". This number can be represented with eight bits by adding two additional '1's in the most significant bit positions as follows: "11101101".

2.3 Addition

Addition is a very common operation with binary numbers and most digital circuits contain binary adders to perform this addition. Addition with unsigned binary numbers is relatively straight forward and operates the same as with base 10 numbers. Two binary numbers are lined up into columns and addition occurs column by column. A sum is produced for each column and a carry can be generated for the next column (a carry occurs when both digits of a column are '1'). Figure 2(a) demonstrates the addition of the number 01100101_2 (101_{10}) with the number 01010111_2 (8_{10}). The marks above each column represent the carry from one column to the next.

$$\begin{array}{rl} \scriptstyle 0\,1\,0\,0\,0\,1\,1\,1 & \\ 01100101_2 & (65_{16}, 101_{10}) \\ +\,01010111_2 & (57_{16}, 87_{10}) \\ \hline 11011100_2 & (BC_{16}, 188_{10}) \end{array}$$

(a) Binary Addition Example.

$$\begin{array}{rl} \scriptstyle 0\,0\,0\,0\,1\,1\,0\,0 & \\ 01100101_2 & (65_{16}, \ 101_{10}) \\ +\quad 1110_2 & (E_{16}, \ 14_{10}) \\ \hline & \end{array}$$

(b) Adding Numbers with Different Sizes.

$$\begin{array}{rl} \scriptstyle 0\,0\,0\,0\,1\,1\,0\,0 & \\ 01100101_2 & (65_{16}, \ 101_{10}) \\ +\,00001110_2 & (E_{16}, \ 14_{10}) \\ \hline 01110011_2 & (73_{16}, 1155_{10}) \end{array}$$

(c) Padding with Zeros.

Figure 2: Examples of Addition between Two Unsigned Binary Numbers.

When adding two unsigned binary numbers with N bits it is possible to generate a number that is too large to represent with the N bits. This is called "overflow" and is demonstrated in Figure 3(a) by adding 101_{10} with 215_{10} using $N = 8$ bits. The result should be 316_{10} but this number requires at least $N = 9$ bits to represent. Overflow can be handled in several ways. In some cases, overflow does not matter and the result of the addition will wrap around the number wheel using mod 2^N arithmetic (i.e., result$= (A + B) \bmod 2^N$). This wrap around is demonstrated in Figure 3(b). In this example, the result is $(101 + 215) \bmod 256 = 60$.

$$\begin{array}{rl} \scriptstyle 1\,1\,0\,0\,0\,1\,1\,1 & \\ 01100101_2 & (65_{16}, 101_{10}) \\ +\,11010111_2 & (D7_{16}, 215_{10}) \\ \hline 00111100_2 & \end{array}$$

(a) Unsigned Addition Overflow.

$$\begin{array}{rl} \scriptstyle 1\,1\,0\,0\,0\,1\,1\,1 & \\ 01100101_2 & (65_{16}, 101_{10}) \\ +\,11010111_2 & (D7_{16}, 215_{10}) \\ \hline 00111100_2 & (3C_{16}, 60_{10}) \end{array}$$

(b) Overflow Wrap-Around.

$$\begin{array}{rl} \scriptstyle 1\,1\,0\,0\,0\,1\,1\,1 & \\ 01100101_2 & (65_{16}, 101_{10}) \\ +\,11010111_2 & (D7_{16}, 215_{10}) \\ \hline 100111100_2 & (13C_{16}, 316_{10}) \end{array}$$

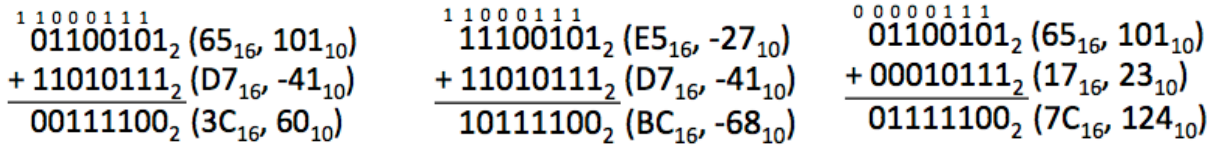
(c) Representing Full Result with Additional Bit.

Figure 3: Examples of Addition between Two Unsigned Binary Numbers.

In other cases, overflow is a problem and the larger, correct result needs to be represented. In these cases, an additional bit must be used to represent the full value. The use of an additional bit is sometimes called "bit-growth" For unsigned binary addition, the value of this bit can be obtained directly from the last carry in the computation as shown in Figure 3(c). In this example, the last carry is brought down and used as the most significant bit of the new 9-bit result.

Addition with two's complement numbers follows the same process as addition between signed numbers: two digits in the same column are added to generate the sum and the carry propagates to the next digit column. The primary difference is that the binary numbers are interpreted as signed, two's complement numbers. Figure 4 provides three examples of two's complement addition (positive+negative, negative+negative, and positive+positive).

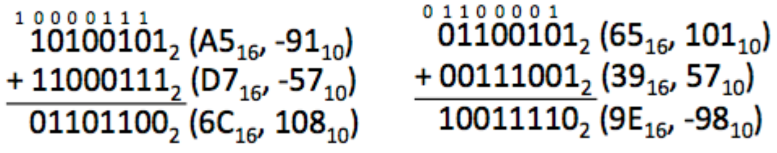
When adding two's complement numbers containing N bits it is possible to generate a number that cannot be represented with N bits. As described in Section 2.2, the range of numbers that can be represented in two's complement form with N bits is $-2^{N-1} \leq value \leq 2^{N-1} - 1$. When adding two positive numbers, the result may be greater than $2^{N-1} - 1$ resulting in a overflow. When adding two negative numbers, the result may be less than -2^{N-1} resulting in an underflow. Figure 5 demonstrates two examples in which overflow and underflow occur. In Figure 5(b) two negative numbers are added (-91 and -57) with an anticipated result of -148. However, -148 cannot be represented with $N = 8$ bits. The result is an underflow positive result of 108. A similar situation occurs when adding +101 with +57 (see Figure 5(b)). Note that overflow will not occur when adding a negative number with a positive number – overflow or underflow only occur when adding two numbers with the same sign.



$$\begin{array}{rl} {\scriptstyle 1\,1\,0\,0\,0\,1\,1\,1} & \\ 01100101_2 & (65_{16}, 101_{10}) \\ +\,11010111_2 & (D7_{16}, -41_{10}) \\ \hline 00111100_2 & (3C_{16}, 60_{10}) \end{array} \quad \begin{array}{rl} {\scriptstyle 1\,1\,0\,0\,0\,1\,1\,1} & \\ 11100101_2 & (E5_{16}, -27_{10}) \\ +\,11010111_2 & (D7_{16}, -41_{10}) \\ \hline 10111100_2 & (BC_{16}, -68_{10}) \end{array} \quad \begin{array}{rl} {\scriptstyle 0\,0\,0\,0\,0\,1\,1\,1} & \\ 01100101_2 & (65_{16}, 101_{10}) \\ +\,00010111_2 & (17_{16}, 23_{10}) \\ \hline 01111100_2 & (7C_{16}, 124_{10}) \end{array}$$

(a) Positive+Negative. (b) Negative+Negative. (c) Positive+Positive.

Figure 4: Examples of Two's Compliment Signed Addition.

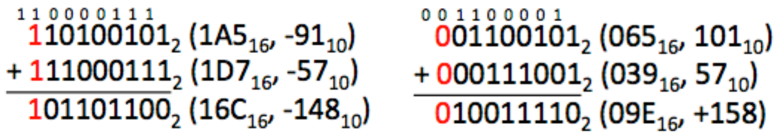


$$\begin{array}{rl} {\scriptstyle 1\,0\,0\,0\,0\,1\,1\,1} & \\ 10100101_2 & (A5_{16}, -91_{10}) \\ +\,11000111_2 & (D7_{16}, -57_{10}) \\ \hline 01101100_2 & (6C_{16}, 108_{10}) \end{array} \quad \begin{array}{rl} {\scriptstyle 0\,1\,1\,0\,0\,0\,0\,1} & \\ 01100101_2 & (65_{16}, 101_{10}) \\ +\,00111001_2 & (39_{16}, 57_{10}) \\ \hline 10011110_2 & (9E_{16}, -98_{10}) \end{array}$$

(a) Negative+Negative. (b) Positive+Positive.

Figure 5: Two's Compliment Signed Addition Overflow.

The problem of overflow and underflow can be resolved by using more bits to represent the operands of the addition. Unlike unsigned addition, however, the last carry out signal of a two's complement addition cannot be used to generate the extra bit of an overflow result. Instead, the two operands must be sign extended *before* performing the addition operation. The examples in Figure 6 demonstrate how overflow and underflow can be prevented by sign extending both operands before performing the addition.



$$\begin{array}{rl} {\scriptstyle 1\,1\,0\,0\,0\,0\,1\,1\,1} & \\ 110100101_2 & (1A5_{16}, -91_{10}) \\ +\,111000111_2 & (1D7_{16}, -57_{10}) \\ \hline 101101100_2 & (16C_{16}, -148_{10}) \end{array} \quad \begin{array}{rl} {\scriptstyle 0\,0\,1\,1\,0\,0\,0\,0\,1} & \\ 001100101_2 & (065_{16}, 101_{10}) \\ +\,000111001_2 & (039_{16}, 57_{10}) \\ \hline 010011110_2 & (09E_{16}, +158) \end{array}$$

(a) Sign Extension of Negative Numbers. (b) Sign Extension of Positive Numbers.

Figure 6: Sign Extension of Two's Compliment Numbers to Prevent Overflow and Underflow.

2.4 Subtraction

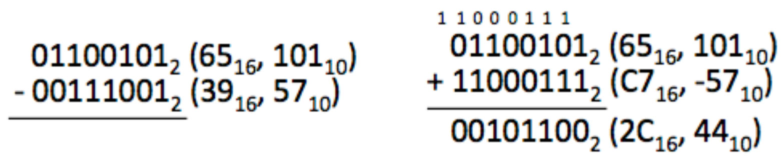
Subtraction is a very similar operation as addition and in most digital systems, subtraction is performed using addition. Specifically, subtraction is performed by computing the two's complement of the second operand (called the "subtrahend") and adding it to the first operand (called the "minuend"). This is shown arithmetically as follows:

$$\begin{aligned} S &= A - B \\ &= A + (-B) \\ &= A + B^{*} \end{aligned}$$

Figure 7 demonstrates this process by subtracting 57 from 101. In Figure 7(b), the value +57 (00111001) is replaced by the value -57 (11000111) and the subtract operator is replaced by an addition operator. Conventional binary arithmetic is performed to compute the result.

2.5 Multiplication

The multiplication operator is defined as $A \times B = P$ where A is the multiplicand, B is the multiplier, and P is the product. In general, if the multiplicand has N_a bits and the multiplier has N_b bits, the product will have $N_p = N_a + N_b$ bits. Unsigned binary multiplication operates the same way as decimal multiplication. A


1 1 0 0 0 1 1 1

$01100101_2\ (65_{16}, 101_{10})$

$-\ 00111001_2\ (39_{16}, 57_{10})$

$01100101_2\ (65_{16}, 101_{10})$

$+\ 11000111_2\ (C7_{16}, -57_{10})$

$00101100_2\ (2C_{16}, 44_{10})$

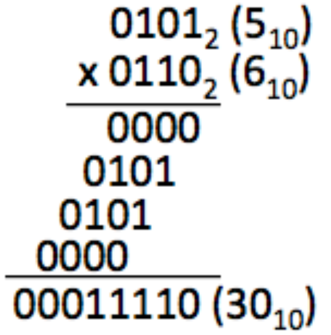
(a) Subtracting 57 from 101. (b) Two's Complement and Add.

Figure 7: Subtraction by Computing Two's Complement and Adding.

partial product is generated for each digit of the multiplier and the partial products are summed to generate the final product. Each partial product, P_i, is computed by multiplying each digit of the multiplier, b_i, with the multiplicand, A, and shifting the result by i binary positions, or $P_i = A \times b_i \times 2^i$. This is shown arithmetically as follows:

$$\begin{aligned} P &= A \times B && (5) \\ &= A \times \left(\sum_{i=0}^{N_b-1} b_i 2^i \right) && (6) \\ &= \sum_{i=0}^{N_b-1} (b_i \times A \times 2^i) = \sum_{i=0}^{N_b-1} P_i && (7) \\ & && (8) \end{aligned}$$

Consider the multiplication of $A = 0101_2$ (5) with $B = 0110_2$ (6). The first partial product, P_0, is generated by multiplying A with b_0 and shifting the partial product zero places (i.e., 2^0). This results in $P_0 = 0101 \times 0 \times 2^0 = 0$. The second partial product, P_1, is generated by multiplying A with b_1 and shifting the partial product one place (i.e., 2^1), or $P_1 = 0101 \times 1 \times 2^1 = 01010$. The other partial products are computed in the same manner ($P_2 = 010100$ and $P_3 = 0$). The finalproduct is computed by summing the partial products: $P = P_0 + P_1 + P_2 + P_3 = 0 + 01010 + 010100 + 0 = 11110$ or 30_{10}. The process of multiplying using partial products and addition shown in Figure 8.


$0101_2\ (5_{10})$

$\times\ 0110_2\ (6_{10})$

0000

0101

0101

0000

$00011110\ (30_{10})$

Figure 8: Binary Multiplication Example.

2.5.1 Signed-Unsigned Multiplication

Binary multiplication can also be performed with signed, two's complement representations. However, the method for multiplication is slightly different for signed multiplicands and multipliers. The first example will consider a *signed* multiplicand (A) and an *unsigned* multiplier (B). The process of multiplication is the same: partial products are created for each digit of the multiplier and the partial products are added to produce the final result. The key difference when using a signed number for the multiplicand is that the partial products must be sign extended before adding them together. Figure 9 demonstrates an example of this with an unsigned multiplier ($A = 6$) and a signed multiplicand ($B = -3$).

```
       1101_2 (-3_10)
     x 0110_2 (6_10)
  ------------
  00000000
  1111101
  111101
  00000
  ------------
  11101110 (-18_10)
```

Figure 9: Multiplication Example with Signed Multiplicand and Unsigned Multiplier

2.5.2 Unsigned-Signed Multiplication

The second example to consider is performing a multiplication with an *unsigned* multiplicand (A) and a *signed* multiplier (B). In this case, the last partial product corresponding to the most significant bit of B must be computed differently. As shown in Equation 2, the most significant bit of a two's complement number is interpreted as $(-1)b_{(N_b-1)}2^{(N_b-1)}$. The last partial product, $P_{(N_b-1)}$, is computed by multiplying A by (-1) times A as follows:

$$\begin{aligned} P &= A \times B \\ &= A \times \left((-1)b_{(N_b-1)}2^{(N_b-1)} + \sum_{i=0}^{N_b-2} b_i 2^i \right) \\ &= \left((-1)b_{(N_b-1)} \times A \times 2^{(N_b-1)} \right) + \sum_{i=0}^{N_b-2} (b_i \times A \times 2^i) P_i \end{aligned}$$

The multiplication is the same as the unsigned-unsigned case except for the final parital product. If b_{N_b-1} is not zero, the partial product $P_{(N_b-1)}$ is obtained by taking the two's complement of A (i.e., multiplying by -1). This is demonstrated in in Figure 10 in the multiplication of $A = 3$ and $B = -5$. Note that the first partial products are based on A ($P_0 = 0011$ and $P_1 = 00110$) and the last partial product, P_3, is based on $-A$ (two's complement of A).

```
        0011_2 (3_10)
      x 1011_2 (-5_10)
  ------------
        0011
       0011
      0000
     1101
  ------------
     1110001 (-15_10)
```

Figure 10: Multiplication Example with Unsigned Multiplicand and Signed Multiplier

2.5.3 Signed-Signed Multiplication

Signed-signed multiplication where both operands are signed can be performed by applying both techniques at the same time. First, the partial products are sign extended to support a signed multiplicand (A) and second, the two's complement of A is used in the final partial product if $b_{N_b-1} = 1$. Figure 11 demonstrates the multiplication of two signed numbers using both of these techniques.

$$\begin{array}{l}
1101_2\ (-3_{10})\\
\times\ 1010_2\ (-6_{10})\\ \hline
00000000\\
1111101\\
000000\\
00011\\ \hline
00010010\ (+18_{10})
\end{array}$$

Figure 11: Multiplication Example with Signed Multiplicand and Signed Multiplier

2.6 Exercises

1. Determine the range of values that can be represented with $N = 10$ bits using (a) an unsigned binary representation, and (b) a signed, two's complement representation.
2. Represent the following numbers as a two's complement numbers using $N = 8$ bits.
   (a) -91
   (b) +113
   (c) -1
   (d) +127
3. Compute the two's complement of each of the numbers in the previous problem.
4. Represent -91 in two's complement representation with $N = 10$ bits and prove that both the $N = 8$ and $N = 10$ binary numbers represent the same decimal value.
5. Convert the following two numbers into $N = 8$ bit two's complement numbers: -83 and +47. Add these two numbers using binary arithmetic and convert the resulting two's complement number to binary prove that your result is what you expect.
6. Subtract the number +34 from the number -71 using a $N = 8$ bit two's complement representation. Show your work.
7. Demonstrate unsigned-unsigned multiplication in binary by multiplying the following two numbers using a 6-bit binary representation: 37 $\times$ 19.
8. Demonstrate signed-unsigned multiplication (signed multiplicand, unsigned multiplier) in binary by multiplying the following two numbers using a 6-bit binary representation: -28 $\times$ 13.
9. Demonstrate unsigned-signed multiplication (unsigned multiplicand, signed multiplier) in binary by multiplying the following two numbers using a 6-bit binary representation: 13 $\times$ -28.
10. Demonstrate signed-signed multiplication in binary by multiplying the following two numbers using a 6-bit binary representation: -13 $\times$ -28.

3 Arithmetic Circuits

Since digital arithmeteic is such an important part of digital systems, it is helpful to understand how these circuits are constructed using logic gates. Understanding the logical structure of these circuits will aid in understanding the size and timing of these arithmetic circuits. This section will desribe the basic arithmetic circuit structures.

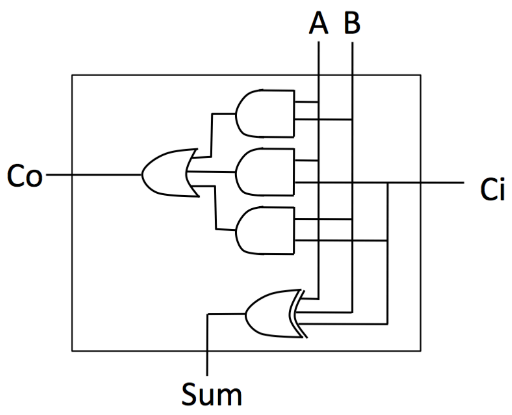
3.1 Full Adder

The first circuit structure to consider is the single-bit "full adder" circuit. The "full adder" is a fundamental arithmetic circuit primitive that is used in most arithmetic circuits. The full adder performs the addition operation between two single-bit values, A and B. Larger, multi-bit addition operations can be created by cascading multiple single-bit full adder primitives. The full adder primitive has three inputs: A, B, and CI. CI is the "carry-in" and allows additional full adders to be casceded. The full adder has two outputs: the sum, S, and carry out (CO). The truth table for the full adder is shown in Table 1.

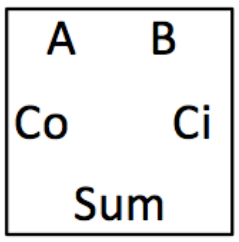
A	B	CI	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1: Truth Table for Single-Bit Full Adder

The logic for each output of the full adder can be minimized as follows: `Sum = A xor B xor CI` and `CO = (A and B) or (A and C) or (B and C)`. A schematic representation of this logic is shown in Figure 12(a). A symbol representing this logic is shown in Figure 12(b). The symbol will be used in place of the schematic when representing larger circuits composed of multiple full adder cells.



(a) Gate-Level Schematic of Full Adder.



(b) Full Adder Cell Symbol.

Figure 12: Full Adder Primitive.

The combinational delay of the full adder circuit is based on the timing of the logic gates used to build the circuit. For the full adder implemented in Figure 12(a), the combinational delay of both outputs is as follows:

$$t_{co} = t_{and2} + t_{or3} \tag{9}$$

$$t_{sum} = t_{xor3} \tag{10}$$

3.2 Multi-Bit Addition

Multi-bit adders can be created by cascading the multiple single-bit full adder of Figure 12. Multiple stages are cascaded by connecting the carry out of one stage to the carry in of the next stage as shown by the three-bit adder in Figure 13. This type of adder is known as a "carry-ripple" adder since the carry signal ripples from the least significant digit to the most significant digit. In this figure, the three-bit binary number A ($a_2a_1a_0$) is added to the three-bit binary number B ($b_2b_1b_0$) to produce a three-bit binary number S ($s_2s_1s_0$) and a carry out (CO_2).

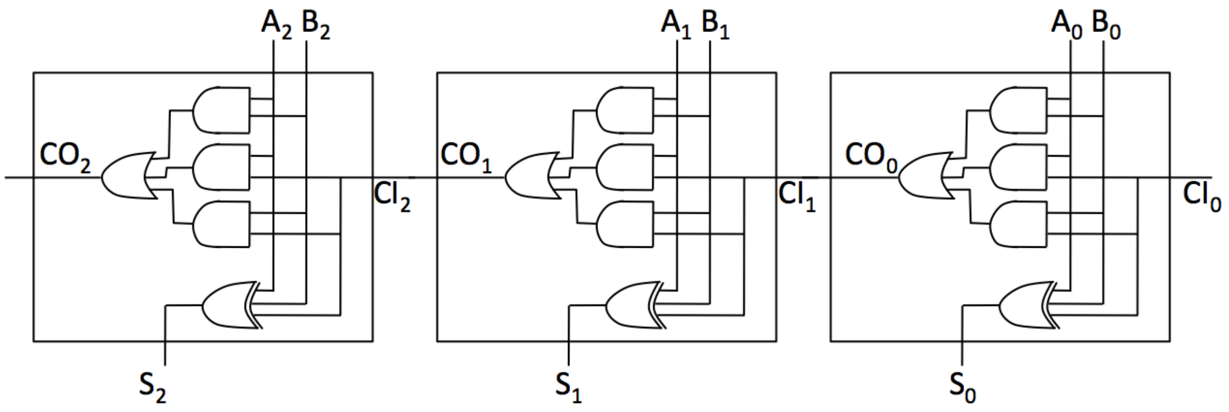


Figure 13: Three-Bit Adder.

The example in Figure 14 demonstrates the addition of the value `A=011` (3_{10}) with `B=100` (4_{10}). The result of this addition is `S=111` (7_{10}).

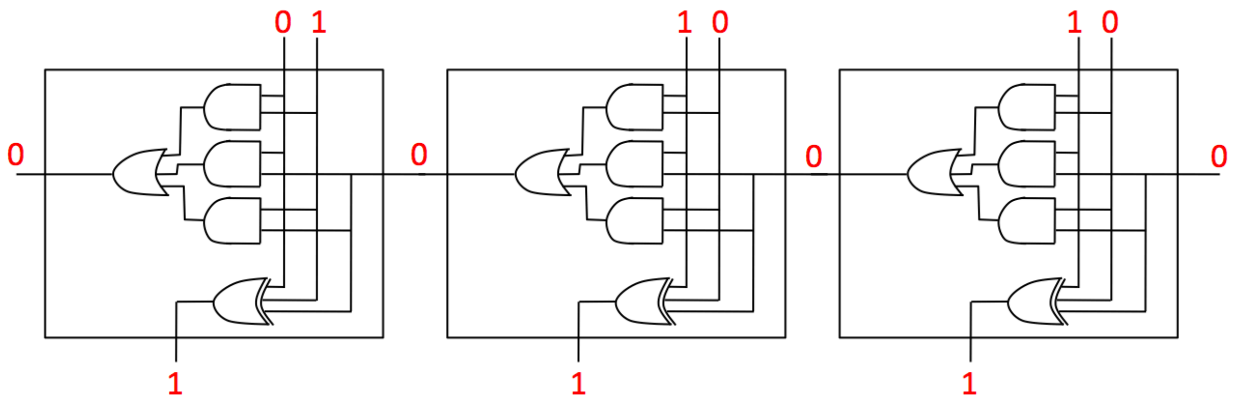


Figure 14: Annotated Three-Bit Adder - Adding $011 + 100$.

As described in Section 2.3, the addition of two N-bit numbers may generate a result that requries one more bit (bit-growth). For unsigned addition, this additional bit can be generated by the carry out output of the last stage. The use of the carry out to generate the extra addition bit is shown in Figure 15.

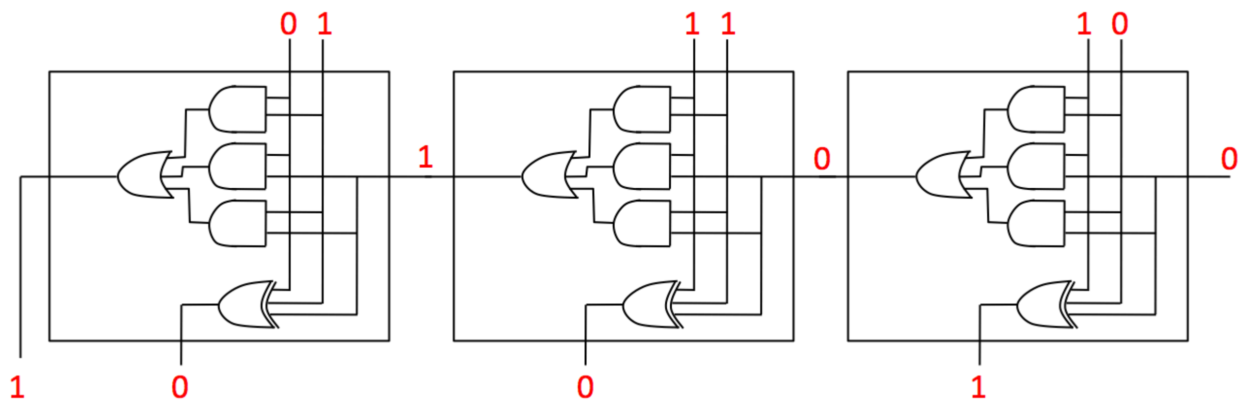


Figure 15: Using the Carry Out to Generate Additional bit.

For two's complement signed addition, the carry out of the last stage cannot be used to generate the extra bit. To generate the extra bit to accomodate bit growth, an additional adder stage must be added. Figure 16 demonstrates the use of an additional adder stage to produce a seven-bit result from two six-bit inputs. The six-bit inputs are sign-extended to create two seven-bit inputs. By sign extending the two addition operands, the inputs to this extra stage are the most significant bit (bit 5) of the input operands.

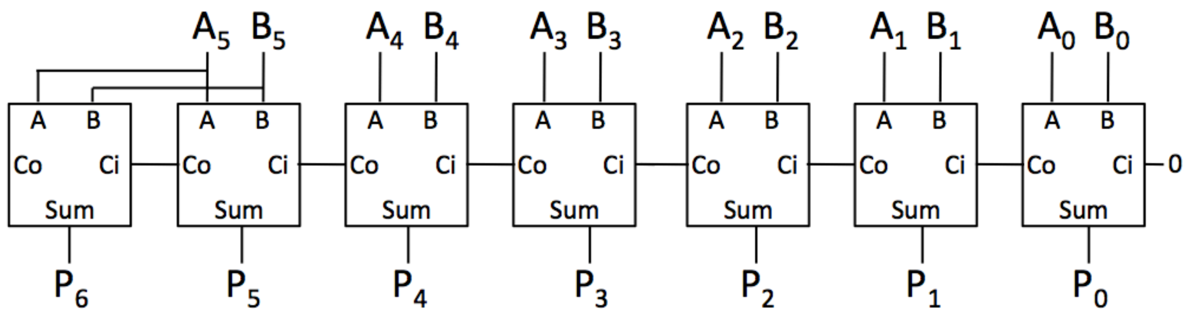


Figure 16: Additional Adder Stage to Support Growth of Two's Complement Addition.

The combinational delay of each signal of the three-bit adder is shown in Figure 17. The combinational delay of the first stage of the adder (t_{s_0} and t_{co_0} is the same as the combinational delay of a single full adder (see Equations 9 and 10):

$$\begin{aligned} t_{s_0} &= t_{sum} = t_{xor3} \\ t_{co_0} &= t_{co} = t_{and2} + t_{or3} \end{aligned}$$

Since the carry out signal of the first full adder stage is an input to the second full adder stage, the combinational delay of the signals in the second stage will be delayed by the carry out signal of the first stage as follows:

$$\begin{aligned} t_{s_1} &= t_{co_0} + t_{sum} = t_{and2} + t_{or3} + t_{xor3} \\ t_{co_1} &= t_{co_0} + t_{co} = 2 \times (t_{and2} + t_{or3}) \end{aligned}$$

The third stage must also wait for the carry out signal of the previous stage. The combinational delay of the third stage is as follows:

$$\begin{aligned} t_{s_2} &= t_{co_1} + t_{sum} = 2 \times (t_{and2} + t_{or3}) + t_{xor3} \\ t_{co_2} &= t_{co_1} + t_{co} = 3 \times (t_{and2} + t_{or3}) \end{aligned}$$

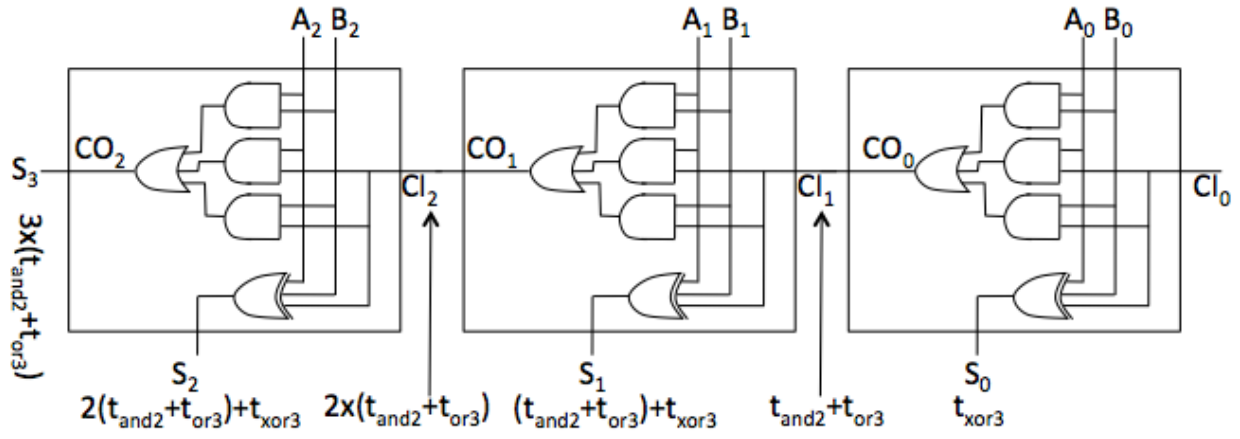


Figure 17: Timing of Three-Bit Adder.

The overall combinational delay of this three-bit adder is determined by the slow path in the logic network. There are two paths that could qualify as the slowest path: CO_2 (S_3) or S_2. The actual slowest path will depend on the relative timing of the xor3, and2, and or3 gates.

$$\begin{aligned} Delay &= Max(t_{s_2}, t_{co_2}) \\ &= Max(2 \times (t_{and2} + t_{or3}) + t_{xor3}, 3 \times (t_{and2} + t_{or3})) \\ &= 2 \times (t_{and2} + t_{or3}) + Max(t_{xor3}, t_{and2} + t_{or3}) \end{aligned}$$

An arbitrarily large ripple-carry adder can be created by cascading as many full adder stages as possible. Figure 18 demonstrates the cascading of N full adder stages to create an N-bit binary adder. The timing of the N-bit binary adder can be found by adding N stages of the carry signal to the sum and carry out signals as follows:

$$t_{s_{N-1}} = (N-1) \times (t_{and2} + t_{or3}) + t_{xor3} \tag{11}$$

$$t_{co_{N-1}} = N \times (t_{and2} + t_{or3}) \tag{12}$$

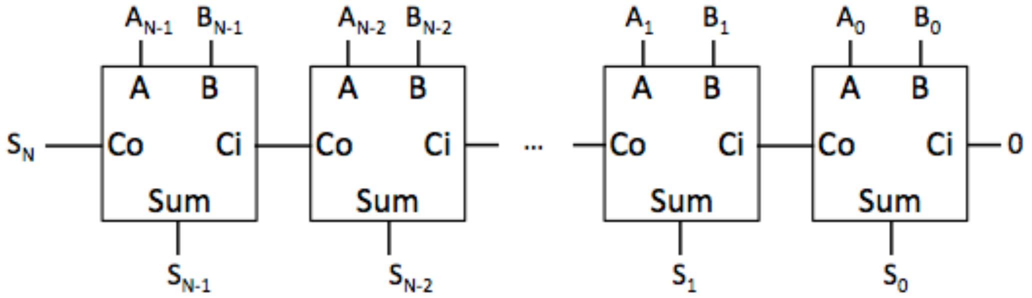


Figure 18: N-Bit Adder.

If $t_{and2} = 2ns$, $t_{or3} = 3ns$, and $t_{xor3} = 4ns$, the combinational delay of a $N = 32$ bit adder can be computed as follows:

$$\begin{aligned} t_{s_{31}} &= (N-1) \times (t_{and2} + t_{or3}) + t_{xor3} \\ &= 31 \times (2ns + 3ns) + 4ns \end{aligned}$$

$$\begin{aligned} &= 159ns \\ t_{s_{32}} = t_{co_{31}} &= N \times (t_{and2} + t_{or3}) \\ &= 32 \times (2ns + 3ns) \\ &= 160ns \end{aligned}$$

The longest combinational path is for S_{32} for a total of 160 ns.

3.3 Subtraction

As described in Section 2.4, subtraction in binary systems is often performed through addition between the first operand and the two's complement of the second operand. The unsigned addition circuit of Figure 13 can be modified to take the two's complement of the B operand. Figure 19 shows a modified adder circuit that performs this two's complement operation before performing the addition. In this circuit, each bit of the B operand is inverted and '1' is added to the operand by applying a '1' to the carry in input of first stage of the adder. Performing this two's complement operation will add an additional delay through the inverters (t_{inv}).

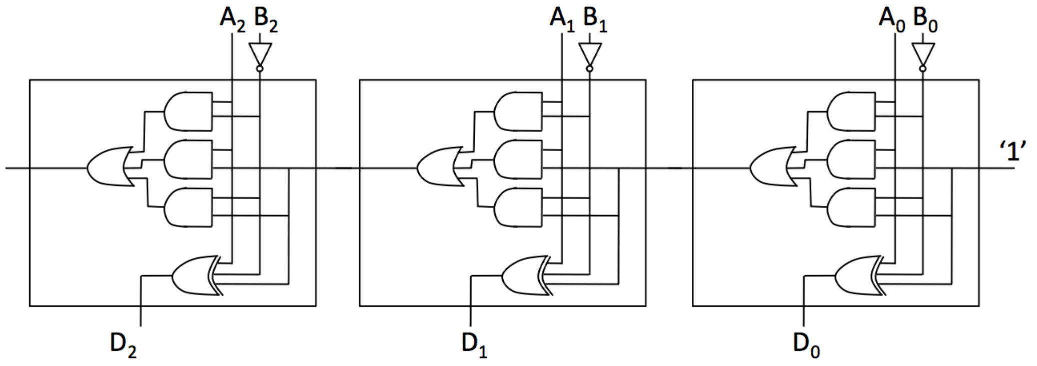


Figure 19: Performing Subtraction by Computing Two's Complement of the B Operand.

Because the standard two's complement adder can be used to perform both addition and subtraction, a dual-purpose adder/subtractor can easily be created from a single two's complement adder. Figure 20 demonstrates a adder/subtractor unit that contains an additional input: `sub`. This signal drives the "carry in" input of the multi-stage adder. When the `sub` signal is '0', no carry is applied to the adder and normal addition is performed. When `sub` is '1', the carry input is '1' for performing the two's complement of B. The `sub` signal is also used to 'invert' the signal B. When `sub` is '0', the signal B will pass through the xor gates. When `sub` is '1', B will be inverted.

3.4 Multiplication

Circuits that perform binary multiplication are very common in digital systems. There are many different ways of implementing multiplication with digital circuits. This section will describe the structure of basic combinataional multipliers to demonstrate the principles of binary multiplication. This will help give obtain the fundamentals necessary for understanding how other styles of multiplication operate. In addition to the basic structure of multiplication, this section will discuss the timing of the combinational multiplier and discuss several ways of pipelining the multiplication operation.

3.4.1 Unsigned-Unsigned Multiplication

As shown in Equation 8, binary multiplication involves the summation of partial products. Each partial product is computed by multipying a single bit of the multiplier, b_i by the multiplicand, A, and shifting the result by 2^i where i is the corresponding bit position of b_i: $P_i = A \times b_i \times 2^i$. Computing the partial products is relatively simple: each bit of the multiplicand, A, is ANDed with one bit of the multiplier, B. Figure 21(a) demonstrates a

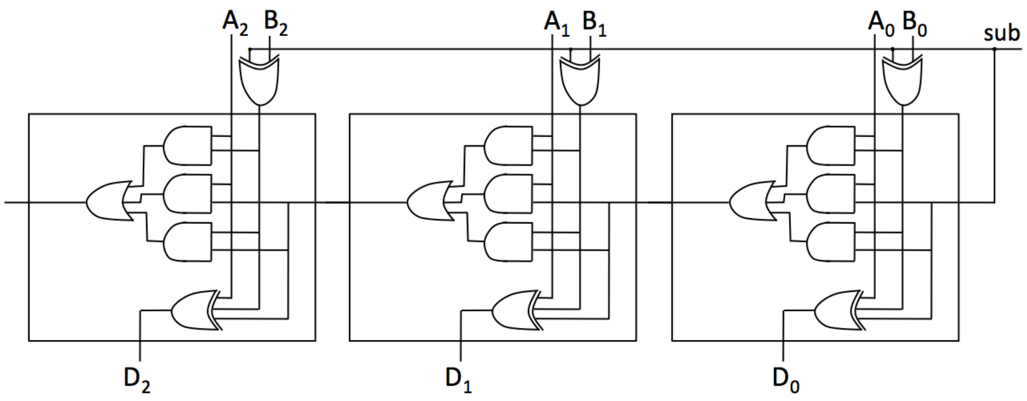
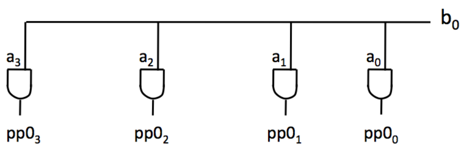
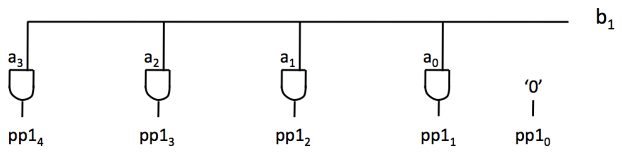


Figure 20: Dual Two's Complement Adder/Subtractor.

circuit that generates partial product P_0 for a four-bit multiplicand. Each bit of A is anded with b_0 to produce a four-bit partial product. If b_0 is '0', then the partial product is zero. If b_0 is '1', then the partial product is equal to A.



(a) Partial Product P_0



(b) Partial Product P_1

Figure 21: Example Partial Products.

Figure 21(b) demonstrates a circuit that generates the partial product P_1. This partial proudct is created by ANDing each bit of A with b_1. Unlike P_0, the partial product is shifted by one bit (2^1) and the least significant bit of P_0 is assigned '0'. The other two partial products, P_2 and P_3, are created in a similar manner.

Once the partial products have been computed, the final product can be found by adding all of the partial products (see Equation 8). Figure 22 demonstrates how the final product can be computed by summing the partial products with a set of adders. The first adder adds the first two partial products. The second adder adds the sum of the first two partial products to the third partial product. The final adder adds the fourth partial product to generate the final product.

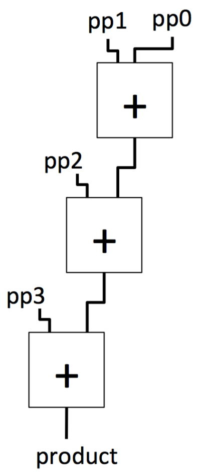


Figure 22: Binary Number Wheel Representation (from Chu, Figure 7.6).

The circuit structure of a 4x4 unsigned multiplier is shown in Figure 23. Four sets of AND gates are used to create the four partial partial products. Three sets of 4-bit adders are used to sum the partial products to create the final sum. The first adder adds the 4-bit P_0 partial product to the 5-bit P_1 partial product to create a 6-bit sum. The second adder adds the 6-bit P_2 partial product to the sum to create a 7-bit result. The final adder adds the last partial product to create an 8-bit result.

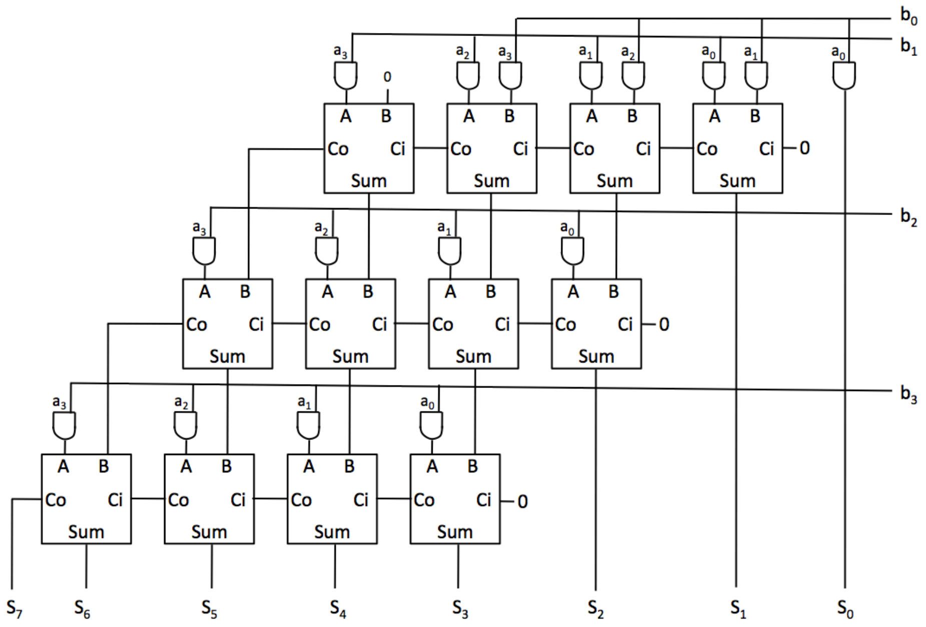


Figure 23: Unsigned Multiplier (4x4).

The multi-stage addition network seen in Figure 23 suggests that the timing of multiplication will be larger than that of addition. To perform a single multiplication operation, the inputs must pass through 2-input AND gates and three stages of adders. The structural multiplier in Figure 23 is repeated with timing annotations in Figure 24. This diagram calculates the delay of the multiplier through each individual adder cell using the paramters: t_{sum} (delay of single-bit sum), t_{co} (delay of single-bit carry-out signal), and t_{and2} (delay of two-input AND gate).

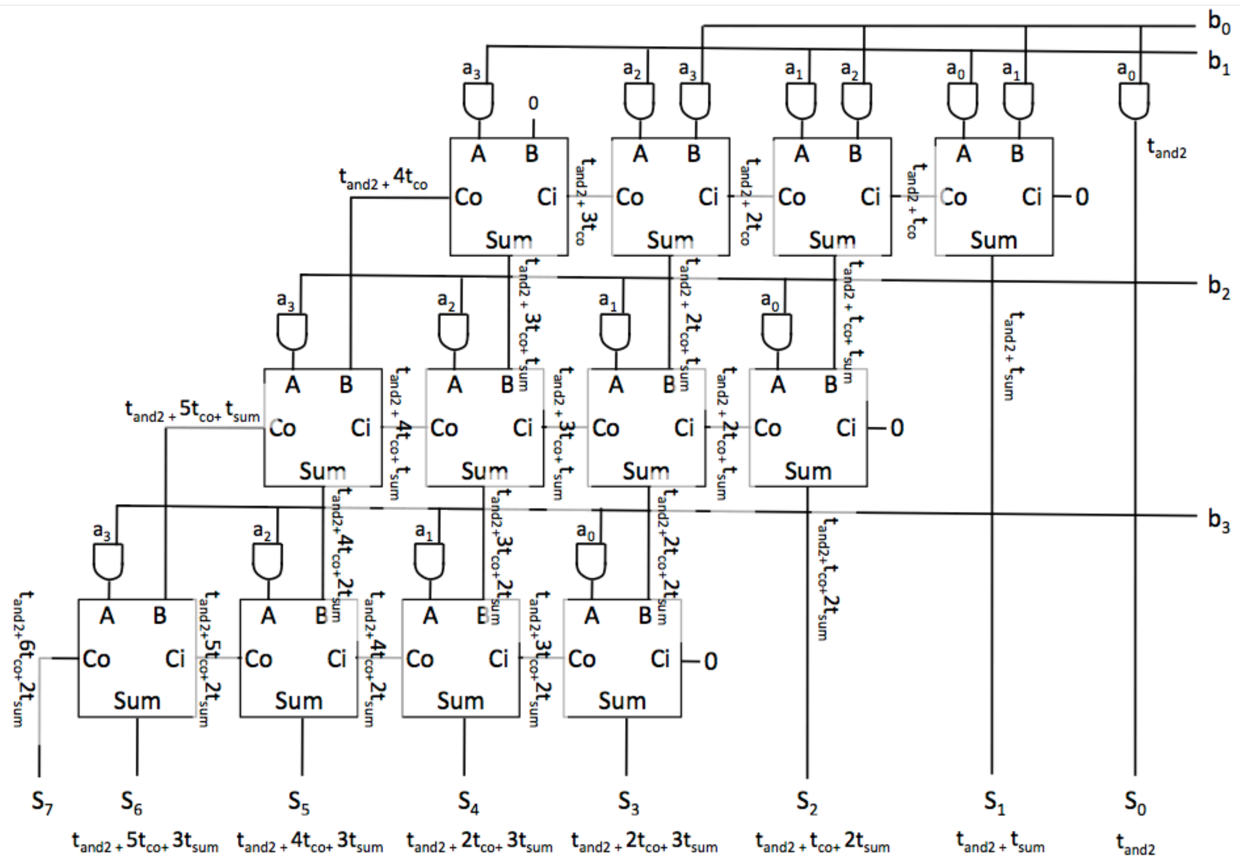


Figure 24: Unsigned Multiplier (4x4) Timing.

Initially, all four partial products are computing in parallel with a delay of t_{and2}. Next, the multiplication result propagates to the left along the carry path and down along the sum path. Every time the signal propagates to the left through a carry, an additional t_{co} is added to the delay. Every time the signal propagates down through a sum, an additional t_{sum} is added to the delay. The signals continue to propagate until reaching the two most significant bits: Bit 6 and Bit 7. Bit 7 has a delay of $t_{and2} + 6t_{co} + 2t_{sum}$ and Bit 6 has a delay of $t_{and2} + 5t_{co} + 3t_{sum}$. The combinational delay of this multiplier is thus Max($t_{and2} + 6t_{co} + 2t_{sum}$,$t_{and2} + 5t_{co} + 3t_{sum}$).

3.4.2 Signed-Unsigned Multiplication

As described in Section 2.5.1, the multiplication operation can use a signed multiplicand operator by *sign extending* the partial products. For the 4×4 bit multpilier, the first partial product, P_0, must be sign extended to five bits so it can be added to the second partial product, P_1. When two five-bit numbers are added, it is possible with signed addition to generate an overflow or underflow. To prevent this from happening, both partial products must be signed extended to *six* bits. This sign extension and the corresponding adder is shown in Figure 25. Note that there are *five* adder cells for this partial product adder instead of four adder cells used in the unsigned-unsigned multiplier. This additional adder cell is required to handle the overflow condition of signed two's complement numbers.

A full 4×4 signed-unsigned multiplier can be constructed using single-bit adder cells as shown in Figure 26. The primary difference between this multiplier and the unsigned multiplier of Figure 23 is the sign extension of the partial products and the use of an additional adder cell in each adder stage to prevent overflow of the signed addition. This circuit structure is capable of performing multplication on the example shown in Figure 9.

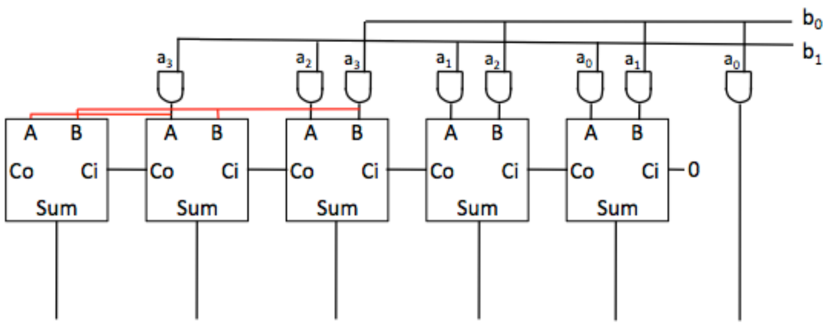


Figure 25: Sign Extending Partial Products.

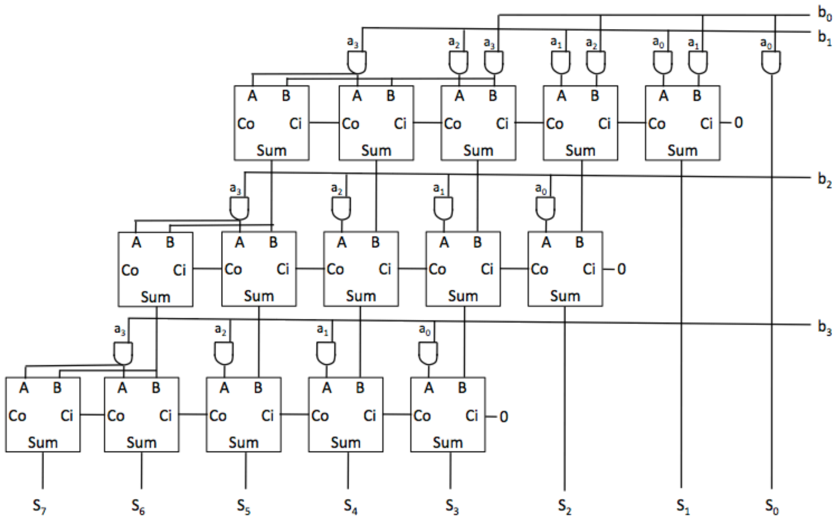


Figure 26: Signed-Unsigned Multiplier.

3.4.3 Unsigned-Signed Multiplication

As described in Section 2.5.2, the multiplication operation can use a signed multiplier operator by taking the two's complement of the final partial product, P_3. Taking the two's complement of the final partial product is the same as *subtracting* the final, non complemented partial product from the current sum. The final partial product adder can be replaced with a partial product subtractor (see Figure 19). The use of a subtractor for the final stage is shown in Figure 27.

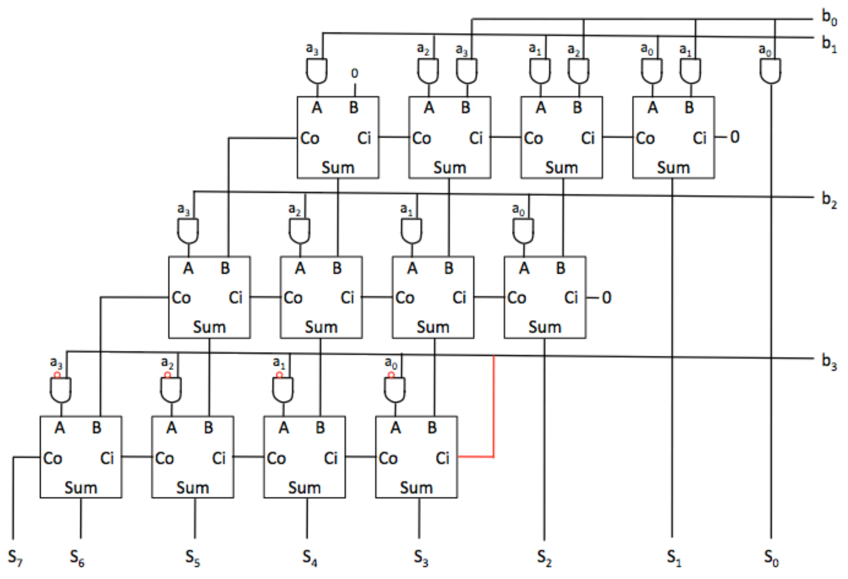


Figure 27: Unsigned-Signed Multiplier.

3.4.4 Signed-Signed Multiplication

A combined signed-signed multiplier may be created by combining both techniques described above: the partial products are sign extended to support a signed multiplicand and the final adder is replaced with a subtractor to take the two's complement of the final partial product.

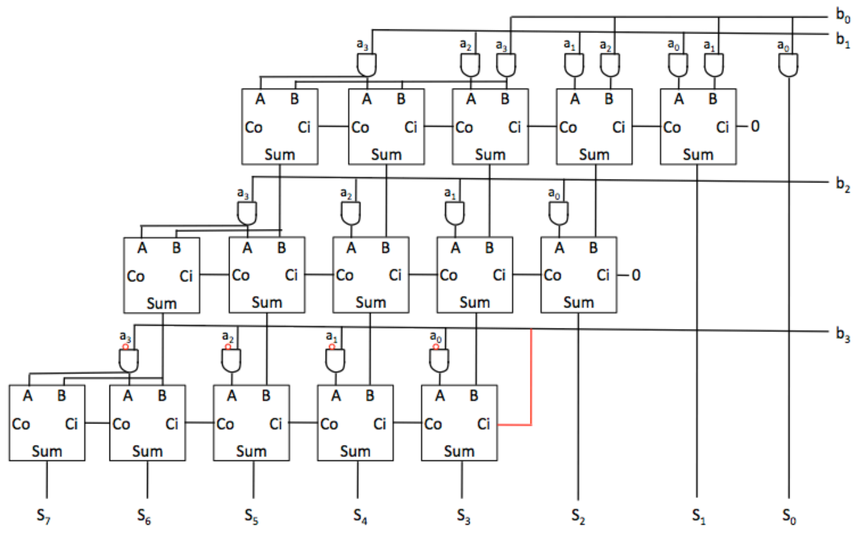


Figure 28: Signed-Signed Multiplier.

3.5 Exercises

1. Create a truth table for a single-bit "Subtractor" (see Table 1). Your inputs should be: A, B, and BI (Borrow In), and your outputs should be: D (difference) an BO (Borrow out). Create a gate-level representation of this single-bit subtractor (see Figure 12(a)).

2. Determine the combinational delay of a $N = 64$ bit adder where $t_{and2} = .5ns$, $t_{or3} = .75ns$, and $t_{xor3} = 1ns$.

3. Create a block diagram representation like Figure 16 to perform the addition of two 8-bit two's complement numbers. Make sure your adder will generate a 9-bit result to avoid overflow and underflow. Using the parameters from the previous problem, determine the combinational delay of this adder.

4. Determine the combinational delay of an N bit subtraction. Base your answer on the structure of Figure 19. Refer to Equation 12 as you determine your answer.

5. Draw a 4-bit (multiplicand)$\times$3-bit (multiplier) unsigned multiplier using the single-cell adder. Your diagram should look like Figure 23 but modifed to represent the different sized operands.

6. Determine the maximum combinational delay of the 4×4 multiplier of Figure 23 assuming $t_{and2} = 2ns$, $t_{co} = 3ns$, and $t_{sum} = 4ns$.

7. Determine the combinational delay of the signed-unsigned multiplier of Figure 26 symbollically using the variables t_{and2}, t_{co}, and t_{sum}.

8. Draw a 3-bit signed-signed multiplier using the single-cell adder. Your diagram should look like Figure 28 but with a different number of adder cells. Demonstrate your circuit operates correctly by showing your multiplier operating on the values $A = 110(-2)$ and $B = 100(-4)$.

9. Determine the combinational delay of the signed-signed multiplier of Figure 28 symbollically using the variables t_{inv}, t_{and2}, t_{co}, and t_{sum}.

4 Arithmetic Operations in VHDL

An important advantage of using hardware description languages (HDL) like VHDL is the ability to quickly generate complex circuits that include arithmetic operations. It is relatively easy to create arithmetic circuits by including statements with arithmeitc operators such as addition (+), subtraction (-), mulitplication (*), and even division (/).

4.1 Numerical Representations using VHDL Types

The unsigned binary and two's complement numeric representation of binary numbers can be represented in VHDL using the IEEE `numeric_std` package[2]. This package introduces new arithmetic types, operators for these types (including both arithmetic and relational operators), and conversion functions. Most VHDL synthesis tools support these types and generate valid digital arithmetic circuits when these types and their operators are used properly.

The two new types introduced in this package include`unsigned` and `signed`. Both types are defined as an array of `std_logic` data types:

```
type UNSIGNED is array (NATURAL range <>) of std_logic;
type SIGNED is array (NATURAL range <>) of std_logic;
```

Signals and ports defined using these types have a fixed range (i.e., a fixed number of bits) and support many arithmetic operations. VHDL synthesis tools are able to synthesize a variety of arithmetic circuits for signals and VHDL structures that use these types.

In addition to the arithmetic operators, the `unsigned` and `signed` types also define the standard logical operators used by the `std_logic_vector` type. These include `not`, `and`, `or`, `nand`, `nor`, `xor`, and `xnor`. Thus standard logical operations can be performed on signals of type `unsigned` and `signed` just as they are for `std_logic_vector`.

Although HDL synthesis provides the ability to generate complex arithmetic circuits, arithmetic operators in VHDL must be used with care. It is relatively easy to generate large, slow circuits with careless use of arithmetic in VHDL. Even worse, it is possible to create circuits that produce an unexpected result if you are not careful. This section will describe a number of rules and introduce several guidelines that you should follow when using arithmetic operations in your synthesizable VHDL code.

The first two rule involve the proper use of types:

Rule Only signals of type `unsigned` and `signed` can be used with arithmetic operations. Objects of type `std_logic_vector` cannot be used in arithmetic operations.

Rule Signals used in arithmetic expressions must be of the same type: `unsigned` with `unsigned`, `signed` with `signed`. Expressions mixing signals of type `unsigned` with `signed` are not allowed[3].

The type `std_logic_vector` defined in package `std_logic_1164` does not define arithmetic operators for `std_logic_1164`. All arithmetic must be done using the `unsigned` with `signed` types defined in `std_logic_arith`. Further, you cannot mix the `unsigned` and `signed` types in the same expression. The behavior of the arithmetic operations depend upon the type and all types must be the same to If arithmetic operations need to be performed between signals of type `unsigned` and `signed`, the appropriate casting functions can be used to insure all signals are interpreted as the same type.

Arithmetic circuits synthesized from hardware description languages can sometimes be very large. The size of these circuits is directly related to the size of the types used in the arithmetic operations. When larger sized operands are used, more hardware is allocated. The designer of the arithmetic circuit needs to be aware of the appropriate operand size and carefully control the size of the operands throughout the circuit. Each of the

[2]Note that there are several other arithmetic packages that have been created within VHDL that perform similar functions including (`std_logic_arith`, `std_logic_unsigned`, and `std_logic_signed`). These packages were created by HDL synthesis vendors to standardize numeric representations and functions within VHDL synthesis flows. Although these packages perform a similar function of the `numeric_std` pacakge, they will not be used or discussed in this manual.

[3]Note that this is specific for `numeric_std` package. There are other packages that do not impose this strict requirement.

following sections describe functions and operations whose corresponding synthesized circuits are directly affected by the size of the type used. Part of your job as a designer is to use as few bits as necessary to complete your arithmetic functions while meeting the specified requirements of the circuit.

4.2 Conversion Functions

Sometimes it is necessary to assign an Integer value to a `signed` or `unsigned` signal. However, it is not possible to directly assign integer values to `signed` or `unsigned` signals. Instead, one of the conversion functions shown in Table 2 must be shown. Because `signed` or `unsigned` signals have a finite number of bits, the number of bits must be specified in either conversion function. If a value is given as the first parameter to either of these functions that cannot be represented with the specified number of bits, an error will be generated (i.e., the function `to_unsigned(100,4)` will generate an error.

Function	Parameter 1	Parameter 2	Result
`to_unsigned(Natural,Natural)`	Value to Convert	Number of Bits	`unsigned`
`to_signed(Integer,Natural)`	Value to Convert	Number of Bits	`signed`

Table 2: Conversion Functions to `unsigned` and `signed`.

The following example demonstrates the use of both conversion functions:

```
signal u8 : unsigned(7 downto 0);
signal s8 : signed(7 downto 0);

u8 <= to_unsigned(100,8);   -- Ok
u8 <= 100;                  -- Not allowed
u8 <= to_unsigned(-16,8)    -- Can't represent negative numbers with unsigned
s8 <= -10;                  -- Not allowed
s8 <= to_signed(-100,8);    -- Ok
```

It is important to provide sufficient number of bits to adequately represent the given `Natural` or `Integer` number. If you do not provide sufficient number of to represent the number exactly, a warning will be generated by the VHDL simulator and it is likely that an error will be generated by the VHDL synthesis tool. The following two examples demonstrate using the conversion function with insufficient number of bits to represent the number.

```
s8 <= to_signed(-130,8);    -- Not enough bits to represent number
u8 <= to_unsigned(1000,8)   -- Not enough bits to represent number
```

In many cases it is necessary to assign a `signed` or `unsigned` number the value zero. This can be done without using the conversion functions by assigning every bit '0' as follows:

```
u8 <= (others => '0');
s8 <= (others => '0');
```

Since the `signed` and `unsigned` types are based on the `std_logic` type, the individual bits of a signal can be assigned any valid character supported by the `std_logic` type:

```
u8 <= "01100100";    -- unsigned "100"
s8 <= "10011100";    -- signed "-100"
```

Sometimes it is necessary to interpret a `signed` or `unsigned` signal as an `Integer` number. The conversion functions shown in Table 3 can be used to perform this type conversion.

Function	Parameter 1	Result
`to_integer(unsigned)`	`unsigned` to Convert	`Integer`
`to_integer(signed)`	`signed` to Convert	`Integer`

Table 3: Conversion Functions to `Integer`.

Function	Operand 1	Operand 2	Result
`resize`	`unsigned`	# bits (`Natural`)	`unsigned`
`resize`	`signed`	# bits (`Natural`)	`signed`

Table 4: Resize functions.

4.3 Signal Resizing

When a `unsigned` or `signed` signal is declared with a fixed range, the signal has a fixed precision. Sometimes it is necessary to increase or decrease the precision of a value. The `numeric_std` package provides the `resize` function for creating new signals with a different precision (i.e., more or less bits than the original signal. The `resize` function is defined for both `unsigned` and `signed` types as shown in Table 4.

The `resize` function operates differently for the `unsigned` or `signed` types. For the `unsigned` type, the `resize` function will "zero-pad" the most-significant bits when the resize function creates a new signal that has more bits than the original signal. The following example demonstrates this "zero-pad" behavior:

```
signal u8 : unsigned(7 downto 0) := "10011100";
signal u4 : unsigned(3 downto 0) := "1001";

-- the two statements produce the same result
u8 <= "0000" & u4;
u8 <= resize(u4,8);
```

When the resize signal creates a new signal that has *fewer* bits than the original signal, the `resize` function will truncate the signal as follows:

```
-- the two statements produce the same result
u4 <= u8(3 downto 0);
u4 <= resize(u8,4);
```

If the value of the original signal can be represented in the new, lower precision type, no information is lost in this resizing operation. For example, if the signal u8="00001100" (+12), the resulting signal `u4=''1100''` has the same result. However, since the original signal can represent a larger range of numbers, information can be lost when resizing to a smaller precision signal. For example, if the signal `u8=''01100100''` (+100), the resulting signal `u4=''0100''` or +4. Care must be taken when truncating signals to avoid inadvertently loosing information from the signal.

The `resize` function uses different rules when operating on `signed` signals. For the `signed` type, the `resize` function will "sign extend" the most-significant bits when the resize function creates a new signal that has more bits than the original signal. The following example demonstrates this sign extension behavior:

```
signal s8 : unsigned(7 downto 0) := "10011100";
signal s4 : unsigned(3 downto 0) := "1001";

-- the two statements produce the same result
s8 <= s4(3) & s4(3) & s4(3) & s4(3) & s4;
s8 <= resize(s4,8);
```

As described in Section 2.2.2, the most significant bits of the new signal are set to the most-significant bit (or sign bit) of the original signal. This preserves the sign and value of the original signal and provides additional precision.

When the resize signal creates a new signal that has *fewer* bits than the original signal, the `resize` function will truncate a `signed` signal as follows:

```
-- the two statements produce the same result
s4 <= s8(7) & s8(2 downto 0);
s4 <= resize(s8,4);
```

The new signal has the same sign as the original signal but fewer bits. If the value of the original signal falls within the range of the new signal, no information in this resizing operation will occur. For example, if the signal `s8=‘‘11111010’’` (-6), the signal resulting from the resize operation `s4=‘‘1010’’` or -6. Like the `unsigned` type, information can be lost when resizing to a smaller type. For example, if the signal `su8=‘‘10011100’’` (-100), the resulting signal `u4=‘‘1100’’` or -4.

4.4 Relational Operators

The `numeric_std` package defines new relational operators for the `unsigned` and `signed` types. These operators do not perform the same function as the equivalent relational operators defined for the `std_logic_vector` type. All relational operators return the type `boolean` and are summarized in Table 5.

Operation	Function
>	Greater Than
>=	Greater Than or Equal
<	Less Than
<	Less Than or Equal
=	Equal
/ =	Not Equal

Table 5: Relational operators for the `unsigned` and `signed` types.

Not all combinations of types can be used for these relational operators. Table 6 summarizes the acceptable combinations of types used for these operators.

Left Operand	Right Operand
`unsigned`	`unsigned`
`Natural`	`unsigned`
`unsigned`	`Natural`
`signed`	`signed`
`Integer`	`unsigned`
`unsigned`	`Integer`

Table 6: Acceptable combinations of types for `unsigned` or `signed` relational operations.

4.5 Addition (+)

The most common arithmetic operation performed is addition (+). The Addition operator is defined for both `unsigned` and `signed` types. The addition operation is defined for several combination of types as shown in Table 7. Note that `unsigned` and `signed` types cannot be mixed within an addition operation.

The following examples demonstrate both allowable and unallowable operations on `unsigned` and `signed` types.

```
signal ua,ub,uc : unsigned(7 downto 0);
signal sa,sb,sc : signed(7 downto 0);
```

Left Operand	Right Operand	Result
unsigned	unsigned	unsigned
unsigned	Natural	unsigned
Natural	unsigned	unsigned
signed	signed	signed
signed	Integer	signed
Integer	signed	signed

Table 7: Valid types for the Addition and Subtraction operation.

```
-- Allowed
ua <= ub + 5;
ua <= 5 + ub;
sa <= sb + -5;
sa <= -5 + sb;

-- Not allowed
ua <= 13 + 5;
ua <= ub + -5;
ua <= sa + 5;
sa <= 5 + -4;
sa <= ua + 3;
```

Although the operands for the addition operation must be of the same type, the two operands do not need to be the same size. The `numeric_std` package supports the ability to perform arithmetic operations on two signals of the same type with different sizes. For example, a signal of type `unsigned(3 downto 0)` can be added to a signal of type `unsigned(7 downto 0)`. When two signals with different sizes are added, the resulting signal type is the *larger* of the two operands. In the previous example, the type of the signal returned from the addition operation is `unsigned(7 downto 0)`. This feature avoids the need to cast signals to specific types when performing simple addition operations. However, the signal assigned to the addition operation *must* be the correct type. If the signal is not the size of the larger operand, a type error will occur. The following examples demonstrate both correct and incorrect use of adding two operands of different size.

```
signal u8a,u8b : unsigned(7 downto 0);
signal u5 : unsigned(4 downto 0);
signal u4 : unsigned(3 downto 0);
signal s8a,s8b : signed(7 downto 0);
signal s4 : signed(3 downto 0);

-- Correct
u8a <= u8b + u5;
s8a <= s4 + s8b;

-- Incorrect
u8a <= u5 + u4; -- resulting signal is unsigned(4 downto 0)
u8a <= s4 + 3; -- the resulting signal is signed(3 downto 0)
```

When `Natural` or `Integer` values are used as one of the two operands within an addition operation the value is automatically converted to the same type as the other operand and with the same size as the other operand. In the following example, the use of the `Natural` number `100` is added to a signal of type `unsigned(7 downto 0)`. The number 100 is internally converted to the same type as `u8b` as shown below:

```
signal u8a,u8b : unsigned(7 downto 0);
```

```
-- The two statements perform the same function
u8a <= u8b + to_unsigned(100,8);
u8a <= u8b + 100;
```

As described above, the size of the result of an addition operation is the size of the *larger* of the two addition operands. This method does not allow for the "bit-growth" that naturally occurs when performing addition. It is possible that an addition operation will generate a result that is too large to fit within the finite range of the result. The following demonstrates how two 8-bit `unsigned` numbers are added and generate a result that requires 9-bit to represent.

```
signal u8a : unsigned(7 downto 0) := "10011100"; -- +156
signal u8b : unsigned(7 downto 0);

-- Overflow (156+117=273 which cannot be represented with 8 bits)
u8b <= u8a + 117;
```

Although the accurate result of the addition operation cannot be represented by the signal, there is no error. The result of an addition operation where overflow occurs will be the actual full precision result truncated to the size of the largest operand. In the example shown above, `ub8` will be equal to "00001001" or 9 (i.e., the lower 8 bits of the correct result: "100001001"). This number is (156+117=273) mod 256 (i.e., mod 2^{N-1} where N is the number of bits used to represent the signal). Truncation occurs the same for both `unsigned` and `signed` numbers.

In many cases we do not care about this overflow and we allow overflow results to wrap around. For example, counters are often created with numeric types and the counter wraps around when it reaches the maximimum value. In the counter example below, the signal `cnt` continuously counts from 0 to 255 and then back to 0.

```
signal u8 : unsigned(7 downto 0) := (others => '0');

process(clk)
begin
  if clk'event and clk='1' then
    cnt <= cnt+1;
  end if;
end process;
```

In some cases, it is important to account for this overflow. This is often true when creating arithmetic or signal processing circuits where we do not want to lose the most-significant bits of an additoin operation. If you need to account for overflow, you need to create a new signal that is one bit larger than the two addition operands. In addition, you need to resize one of the two addition operations so that they are the same size as the result. The following example demonstrates how to preserve the overflow result for the addition of two `unsigned` numbers.

```
signal u8 : unsigned(7 downto 0) := "10011100"; -- +156
signal u9 : unsigned(8 downto 0);

-- No overflow
u9 <= resize(u8,9) + 117;
```

4.6 Subtraction

The `numeric_std` package also defines a subtraction ($-$) operator. The subtraction operator is defined for both `unsigned` and `signed` operands using the same rules as addition (see Table 7). Like addition, subtraction can mix `unsigned` operands with `Natural` numbers and `signed` operands can be used with `Integer` operands. In addition, the same rules for operand sizing apply for subtraction as for addition. Specifically, the resulting type of a subtraction operand is the same size as the *larger* of the two operands. The following example demonstrates both correct and incorrect use of the subraction operator.

```
signal u8a,u8b : unsigned(7 downto 0);
signal u5 : unsigned(4 downto 0);
signal u4 : unsigned(3 downto 0);
signal s8a,s8b : signed(7 downto 0);
signal s4 : signed(3 downto 0);

-- Correct
u8a <= u8b - u5;
u8a <= 25 - u8b;
s8a <= s4 - s8b;
s8a <= s8b - -100;

-- Incorrect
u8a <= u5 - s8a; -- Mixing unsigned and signed
u8a <= u5 - 4;   -- Resulting type does not match
s8a <= s4 - -3;  -- Resulting type does not match
```

Like addition, the subtraction may produce a result that cannot be represented by the resulting type. For the `unsigned` type it is possible that a subtraction will generate a negative number. Since negative numbers are not represented by the `unsigned` type, the result will wrap around to a large positive number. For the `signed` type it is possible that a subtraction operation will generate a negative number that cannot be represented in the given fixed precsion of the signal. The following example demonstrates underflow for a `signed` subtraction operation:

```
signal s8a : signed(7 downto 0) := "10000011"; -- -125
signal s8b : signed(7 downto 0);

-- Underflow
s8b <= s8a - 10;
```

The correct result should be -135 but this number cannot be represented with an 8-bit two's complement number. The actual result will be "01111001" (which is a truncated version of the full precision result of "101111001"). The correct result can be obtained by resizing one of the operands and assigning the result to a larger `signed` signal:

```
signal s8 : signed(7 downto 0) := "10000011"; -- -125
signal s9 : signed(8 downto 0);

-- No overflow
s9 <= resize(s8,9) - 10;
```

4.7 Multiplication (*)

The multiplication operator ($*$) is also supported within the `numeric_std` package. Multiplication circuits, however, are much larger than addition/subtraction circuits and great care should be used when using this operator. The multiplication operator is defined for both `unsigned` and `signed` operands using the same rules as addition and subtraction (see Table 7). Like addition, subtraction can mix `unsigned` operands with `Natural` numbers and `signed` operands can be used with `Integer` operands.

The rules for determining the size of the product of a multiplication are different than the rules used in addition and subtraction. Like addition and subtraction, the multiplication operator can operate on two values of different sizes. Unlike addition and subtraction, the multiplication operation does not result in any loss of range. To preserve the full range of the result, the size of the signal assigned as the product of the multiplication must be the sum of the sizes of the two operations used in the multiplication. Consider the following example:

```
product <= operand1 * operand2;
```

In this example, the size of the signal `product` must be equal to the size the signal `operand1` *plus* the size of the signal `operand2` (i.e., `product'length = operand1'length + operand2'length`). The following examples demonstrate both correct and incorrect multiplication operations:

```
signal u8a,u8b : unsigned(7 downto 0);
signal u5 : unsigned(4 downto 0);
signal u16 : unsigned(15 downto 0);
signal s8a,s8b : signed(7 downto 0);
signal s16 : signed(15 downto 0);
signal s4 : unsigned(3 downto 0);

-- Correct
u16 <= u8a * u8b;
u16 <= u8a * 100;
s16 <= s8a * s8b;
s16 <= s8a * -58;

-- Incorrect
u16 <= u8a * u5; -- result should be unsigned(12 downto 0);
s16 <= s8a * s4; -- result should be unsigned(11 downto 0);
```

In many cases we do not want to keep the full precision of the multiplication operation. In these cases, you must still create a full precision signal and then use only a sub-range of the resulting full precision signal. The following example demonstrates how to save the top ten bits of the 16-bit product between two 8-bit operands:

```
signal s8 : signed(7 downto 0) := "01101011";
signal s16 : signed(15 downto 0);
signal prod10 : signed(9 downto 0);

s16 <= s8 * -58;
prod10 <= s16(15 downto 6);
```

Because multipliers can consume so much area, it is often necessary to manually build the multiplier from VHDL addition operators. Listing 7.34 from [1], copied below, is a VHDL code example for implementing a $8x8 = 16$ bit unsigned-unsigned multiplier. This code will generate a multiplier similar in style to the circuit shown in Figure 23. A unique signal is assigned for each partial product and the final product is computed by adding all of the partial products. Variations on this VHDL code can be made to implement a signed-unsigned, unsigned-signed, and signed-signed multiplier.

```
--=============================
-- Listing 7.34 adder-based multiplier
--=============================
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult8 is
   port(
      a, b: in std_logic_vector(7 downto 0);
      y: out std_logic_vector(15 downto 0)
   );
end mult8;

architecture comb1_arch of mult8 is
   constant WIDTH: integer:=8;
   signal au, bv0, bv1, bv2, bv3, bv4, bv5, bv6, bv7:
      unsigned(WIDTH-1 downto 0);
   signal p0,p1,p2,p3,p4,p5,p6,p7,prod:
      unsigned(2*WIDTH-1 downto 0);
begin
   au <= unsigned(a);
   bv0 <= (others=>b(0));
   bv1 <= (others=>b(1));
   bv2 <= (others=>b(2));
   bv3 <= (others=>b(3));
   bv4 <= (others=>b(4));
   bv5 <= (others=>b(5));
   bv6 <= (others=>b(6));
   bv7 <= (others=>b(7));
   p0 <="00000000" & (bv0 and au);
   p1 <="0000000" & (bv1 and au) & "0";
   p2 <="000000" & (bv2 and au) & "00";
   p3 <="00000" & (bv3 and au) & "000";
   p4 <="0000" & (bv4 and au) & "0000";
   p5 <="000" & (bv5 and au) & "00000";
   p6 <="00" & (bv6 and au) & "000000";
   p7 <="0" & (bv7 and au) & "0000000";
   prod <= ((p0+p1)+(p2+p3))+((p4+p5)+(p6+p7));
   y <= std_logic_vector(prod);
end comb1_arch;
```

4.7.1 Inferring Multipliers

FPGAs and standard-cell architectures sometimes provide dedicated multiplier blocks for high-speed, efficient multiplication. HDL synthesis tools attempt to map multiplication operations to these predefined blocks. For example, the Xilinx^{tm} Spartan-3E series FPGA contains dedicated *pipelined* unsigned 18x18=32-bit multipliers. To infer this multiplier, the size of the multiplier must match the size of the built-in multiplier and the timing of the pipelining must match the timing of the built-in multiplier. Details on how to properly infer such multipliers can be found in the corresponding synthesis tool documents [3]. The following example demonstrates one way to infer this multiplier within the Spartan-3E FPGA:

```
signal a,a_in,b,b_in : unsigned(17 downto 0);
signal mult_res,product : unsigned(35 downto 0);
signal pipe1, pipe2, pipe3 : unsigned(35 downto 0);

mult_res <= a_in * b_in;

process(clk)
begin
  if clk'event and clk='1' then
    a_in <= a;
    b_in <= b;
    pipe1 <= mult_res;
    pipe2 <= pipe1;
    pipe3 <= pipe2;
    product <= pipe3;
  end if;
end process;
```

4.8 Division (\)

Division (\) is also supported within the `numeric_std` package. Division, however, is not commonly supported by HDL synthesis tools and thus will not be described in much detail here. The division operator is defined for both `unsigned` and `signed` operands using the same rules as addition and subtraction (see Table 7). Like addition, subtraction can mix `unsigned` operands with `Natural` numbers and `signed` operands can be used with `Integer` operands. The size of the quotient (result of division) must be the same size as the dividend (first operand). The divisor (second operand) can be any size. The following examples demonstrate the proper types for division:

```
signal u8a,u8b : unsigned(7 downto 0);
signal u5 : unsigned(4 downto 0);
signal u16 : unsigned(15 downto 0);
signal s8a,s8b : signed(7 downto 0);
signal s16 : signed(15 downto 0);
signal s4 : unsigned(3 downto 0);

-- Correct
u8a <= u8b / u5;
s8a <= s8b / to_signed(-4,3);
```

4.9 Summary of VHDL Arithmetic Operation Rules

1. Only signals of type `unsigned` and `signed` can be used with arithmetic operations. Objects of type `std_logic_vector` cannot be used in arithmetic operations.
2. Only values of the same type can be used in the same expression
   (a) Objects of type `unsigned` can only be used in arithmetic expressions of type `unsigned`
   (b) Objects of type `signed` can only be used in arithmetic expressions of type `signed`
   (c) Objects of type `signed` cannot be used in the same expression as objects of type `unsigned`
3. Arithmetic operations *can* be performed between two objects of the same type that have *different* sizes.
4. When two objects of the same size are added (+) or subtracted (-), the size of the resulting object is the same size as the two operands.
   (a) There is no "bit-growth" to take care of overflow or underflow.
   (b) VHDL descriptions must handle "bit-growth" manually
5. When two objects of a different same size are added (+) or subtracted (-), the size of the resulting object is the same size as the *largest* of the two operands.
   (a) Again, there is no "bit-growth" to take care of overflow or underflow.VHDL descriptions must handle "bit-growth" manually
6. When two objects are multiplied (*), the size of the resulting object is the size of the sum of the operand sizes
   (a) There is no "bit-loss" with the multiplication operator
   (b) You may not need all of the bits – use a sub-range of the mulitplication result to ignore some of the bits of the computation

4.10 Exercises

1. Create the VHDL statements necessary for adding a 4-bit signed number to a 6-bit signed number and generate an 8-bit result.

2. Create the VHDL statements necessary to subtract a 8-bit unsigned number from a 4-bit signed number and generate a 9-bit result. Make sure your types are correct and there is no underflow in the result.

3. For each of the following statements: i) indicate whether the right side of the signal assignment statement is valid VHDL (i.e. will it compile without error), ii) if the right side of the signal assignment statement is valid, indicate the type of the result (i.e., the type of rx where x=1-13), and iii) determine the result of the statement (again, only if the right side of the statement is valid). Use the following VHDL signal declarations to respond to each of the statements below.

   ```
   signal u1 : unsigned(3 downto 0) := "0110";
   signal u2 : unsigned(3 downto 0) := "1101";
   signal u3 : unsigned(5 downto 0) := "110011";
   signal u4 : unsigned(5 downto 0) := "010100";
   signal s1 : signed(3 downto 0) := "0111";
   signal s2 : signed(3 downto 0) := "1100";
   signal s3 : signed(5 downto 0) := "111111";
   signal s4 : signed(5 downto 0) := "011000";
   ```

   (a) `r1 <= u1 + u2;`
   (b) `r2 <= u2 + u3;`
   (c) `r3 <= u1 - u4;`
   (d) `r4 <= u1 + s1;`
   (e) `r5 <= signed(u3) + s4;`
   (f) `r6 <= s1 - s2;`
   (g) `r7 <= s3 + s4;`
   (h) `r8 <= s2 - u2;`
   (i) `r9 <= s3 + s1;`
   (j) `r10 <= s4 - (signed(u2));`
   (k) `r11 <= u1 * u4;`
   (l) `r12 <= s3 * (-s2);`
   (m) `r13 <= s1 * s2;`

4. Text Problem 7.2 The "sign-magnitude" format (sometimes called the "ones complement" format) is a different format than the two's complement format. The "sign-magnitude" format uses the MSB for the sign (like the two's complement format) but the magnitude is the absolute value of the magnitude. For example, the number -6 is represented as "1110" in sign-magnitude format (MSB is a '1' indicating a negative number and the remaining bits, 110, represent the magnitude of the negative number). In two's complement format, -6 is represented as "1010". You are to create a combinational circuit that converts from the twos complement format to the sign-magnitude format (i.e., convert "1010" to "1110").

5. Create the VHDL code that performs a multiplication between the two 8-bit signed numbers and produce a 8-bit result that uses the *least* significant bits of the result. Make sure that your multiplier produces the correct sign.

6. Modify the VHDL in Listing 7.34 to create an $8 \times 8 = 16$ bit signed-signed multiplier (see Problem 7.16).

7. Modify the VHDL in Listing 7.34 to create a *pipelined* multiplier that provides a register between each stage of the partial product adders.

References

[1] Pong P. Chu. *RTL Hardware Design Using VHDL*. Wiley, 2006.

[2] Brent E. Nelson. *Designing Digital Systems*. Brigham Young University, 2005.

[3] Xilinx Corporation. *XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices*, December 2010. UG627 (v 12.4).