Binary Number Systems and Binary Arithmetic

A *SUMMARY* of what you learned in Chapter 5, Part I:
Binary decoders and 74138: an off-the-shelf 3-to-8 binary decoder
BCD code: upsides and downsides
Seven-segment code and seven-segment displays
BCD-to-seven-segment decoders
Binary encoders and Priority encoders
74148: an 8-to-3 priority encoder
Selectors or multiplexers (muxes)
Mux expansion in 2 dimensions: number of inputs (choices) and number of bits per input
Read-only-memories (ROMs)
Logic circuits as ROMs and vice versa
Muxs look like ROMs
How to use muxs to realize logic functions
74151: an 8-input 1-bit multiplexer
Constant inputs eat the hardware

Introduction

Binary number systems and binary arithmetic are introduced, and one type of adder/subtractor is designed using the systematic procedure developed in the previous chapters. Adders/subtractors are an essential component of microcomputers. We then learn how to use subtractors to design unsigned/signed comparators.

We also learn the concept of *non-systematic* design methodology. In this technique, a (complex) problem is decomposed into manageable (and maybe solved) problems. Then the solutions to these small problems are stitched together to reach a complete solution for the whole problem. Finally, two more codes, namely Gray and ASCII, are presented.

Unsigned Number System

In Chapter 1, the unsigned binary number system was introduced. In this system, we cannot represent both negative and positive numbers at the same time. In other words, all numbers that we use are *magnitudes*, and therefore assumed to have the *same* sign.

In Chapter 1, you learned that the binary system is also a *positional* system (similar to our daily used decimal system) as each digit has a weight determined by its *position* in the number. For example

$$101101 = 2^5 \times 1 + 2^4 \times 0 + 2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = (45)_{\text{ten}} \qquad (1)$$

The six bits (from left to right) comprising this binary number have a weight equal to 2^5, 2^4, 2^3, 2^2, 2^1, and 2^0, respectively, in which 5, 4, 3, 2, 1, and 0 represent the positions of the corresponding bits. The rightmost

Copyright © 2005-2022 N. Tabrizi

bit and the leftmost bit are called the least significant bit (LSb) and the most significant bit (MSb), respectively. The weighted sum in Equation (1) also shows how to convert a binary number to its decimal equivalent. In a binary-to-decimal conversion, if 0s (of the binary number) are fewer than 1s, it might be more convenient to obtain the decimal equivalent for the bit-wise complement of the binary number first, and then take it away from $2^n - 1$, where n is the length of the binary number. For example, consider N = 101101.

Bit-wise complement of N = 010010

$010010 = 2^5 \times 0 + 2^4 \times 1 + 2^3 \times 0 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 0 = (18)_{ten}$

$N = (2^6 - 1) - 18 = 63 - 18 = 45$

In Chapter 1, we also learned how to perform the opposite conversion through successive divisions by 2. The following example should refresh your memory:

Example 1. Convert $(235)_{ten}$ to binary.

The consecutive steps for this conversion are shown in Figure 1. The remainder from each step is circled, and the corresponding quotient is shown next to the remainder. In this algorithm, the quotient from each step becomes the dividend for the next step. When a quotient becomes 0, the conversion terminates. The first remainder is the LSb of the resulting binary number, and the following remainders comprise more significant bits in ascending order. Therefore, $(235)_{ten} = (11101011)_{two}$.

$$235 \div 2 = \textcircled{1}\ 117 \div 2 = \textcircled{1}\ 58 \div 2 = \textcircled{0}\ 29 \div 2 = \textcircled{1}\ 14 \div 2 = \textcircled{0}\ 7 \div 2 = \textcircled{1}\ 3 \div 2 = \textcircled{1}\ 1 \div 2 = \textcircled{1}\ 0$$

Figure 1. Decimal (235) to binary conversion using successive divisions-by-2 algorithm

Binary Addition

There are different algorithms to perform binary addition. In this section (and in this book), we study the *paper-and-pencil* method. This is the everyday algorithm that we use but with decimal numbers, i.e., numbers represented in base ten.

Example 2. Add in the 4-bit system: 1001 + 0011. (Each operand is 4 bits wide in the 4-bit system.)

The paper-and-pencil addition for this example is shown in Figure 2.

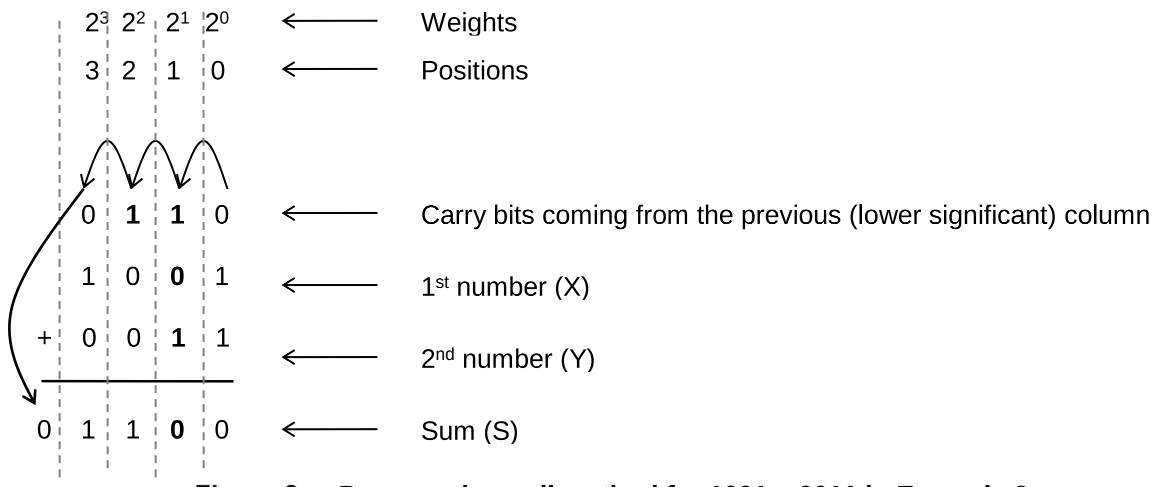


Figure 2. Paper-and-pencil method for 1001 + 0011 in Example 2

The same procedure is performed for each column. Starting with the least significant (or the right-most) column, three bits are added: Two bits from the two operands and one carry-in bit (or c_{in} for short) from the preceding (less) significant column. Since there is no column before the least significant column, the carry bit into this column is always 0. As a result of this addition, two output bits are generated: 1) a *sum* bit (or s for short) with the same weight as the corresponding column, so s is seated in the same column, and 2) a *carry-out* bit (or c_{out} for short), which has a weight twice as large as s. Therefore, c_{out} ripples to the next column with a matching weight, and becomes c_{in} for this column as shown in Figure 2. As an example, consider the second column from the right, which receives a c_{in} of 1 from the least significant column, adds this bit and the two operand bits, namely a 0 (from the 1st number) and a 1 (from the 2nd number), and generates an s equal to 0 and a c_{out} equal to 1.

The sum in the n-bit system (in which each operand is n bits wide) could be (n + 1) bits wide at the most. To prove this, note that the largest (n+1)-bit number is $2^{n+1} - 1$, and the largest n-bit number is $2^n - 1$. Therefore, $(2^n - 1) + (2^n - 1)$ would produce the largest sum in the n-bit system:

$(2^n - 1) + (2^n - 1) = 2 \times (2^n) - 2 = 2^{n+1} - 2 =$ largest (n+1)-bit number minus 1, so fits in (n + 1) bits.

In the result of an n-bit addition, the (n+1)th bit is called the *final carry bit* (or *carry bit* for short). The carry bit is the c_{out} from the most significant column of the addition. A carry bit could be either a 0 or a 1. If it is 0, we say *NO final carry* (or *NO carry* for short) is generated, or simply **no carry** happens. See Figure 2. Similarly, if the carry bit is 1, we say *carry* IS generated or simply **carry** occurs. See the next example.

Example 3. Add in the 4-bit system: 1011 + 1001

The paper-and-pencil addition is shown in Figure 3. Here, the carry bit is 1 signifying that the result is too large to fit in 4 bits, where 4 is the size of the addition system in this example; i.e., the result is *out of range*.

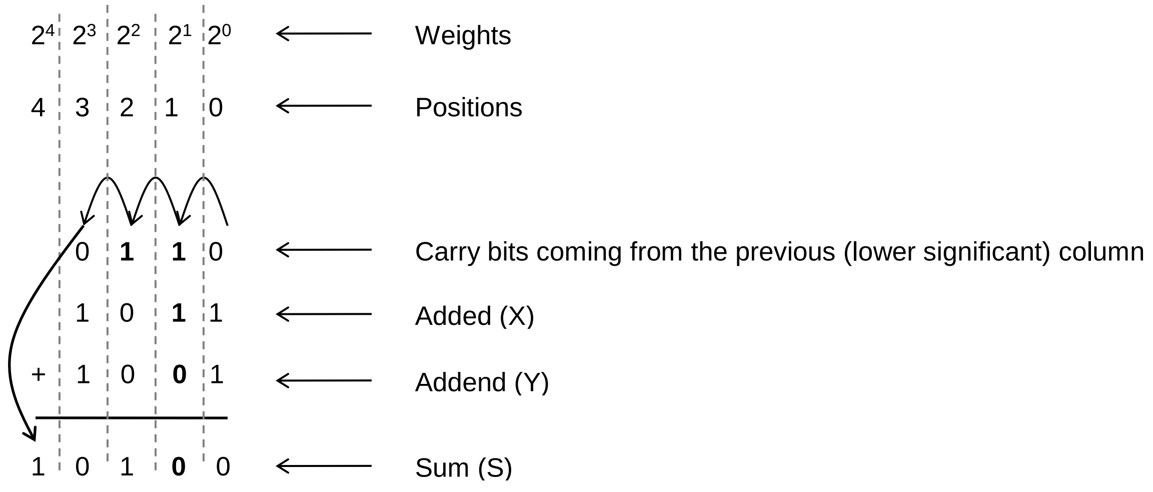


Figure 3. Paper-and-pencil method for 1011 + 1001 in Example 3

Example 4. Add in the 4-bit system: 1001 + 101

The second operand is only 3 bits wide. We first need to make it 4 bits wide by appending a 0 to the left:

101 = 0101

Now, addition 1001 + 0101 can be performed as usual. The result is 1110 with no final carry.

Definition: We may append as many 0s as we need to the left of a number. This is called ***zero extension***.

Adder Design

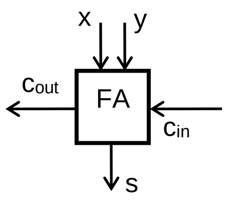
In this section, we implement the paper-and-pencil algorithm. The resulting hardware is called a *ripple carry adder*, or simply a *ripple adder*. As demonstrated in the above examples, a binary addition consists of some single-column additions, each of which can be translated into an eight-row truth table with three inputs (x, y, and c_{in}) and two outputs (s and c_{out}) as illustrated in Figure 4*a*. Figure 4*b* shows the minterm lists of s and c_{out}. The corresponding hardware is called a *full adder* or FA for short. The logic symbol of a FA is depicted in Figure 4*c*. n full adders can be cascaded to obtain an n-bit adder as shown in Figure 5*a* for $n = 4$. Figure 5*b* shows a logic symbol for this adder.

Row	x y c_{in}	c_{out}	s
0	0 0 0	0	0
1	0 0 1	0	1
2	0 1 0	0	1
3	0 1 1	1	0
4	1 0 0	0	1
5	1 0 1	1	0
6	1 1 0	1	0
7	1 1 1	1	1

(a)

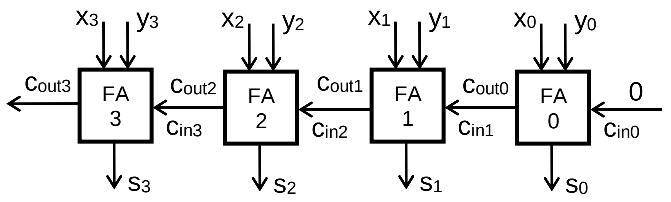
$$c_{out}(x, y, c_{in}) = \Sigma\,(3, 5, 6, 7)$$
$$s(x, y, c_{in}) = \Sigma\,(1, 2, 4, 7)$$

(b)

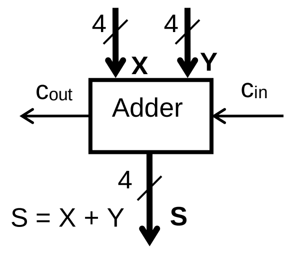


(c)

Figure 4. Full adder: (a) truth table, (b) on-set minterm lists, (c) logic symbol



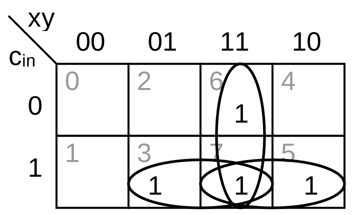
(a)



(b)

Figure 5. Four-bit ripple adder: (a) logic diagram, (b) symbol

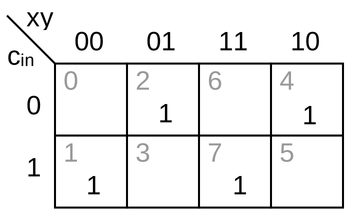
The K-maps and resulting SOP expressions for c_{out} and s are illustrated in Figure 6*a* and Figure 6*b*, respectively.



c_{in} \ xy	00	01	11	10
0	0	2	6: 1	4
1	1	3: 1	7: 1	5: 1

c_{out} (a b c_{in}) = x . y + y . c_{in} + c_{in} . x

(a)



c_{in} \ xy	00	01	11	10
0	0	2: 1	6	4: 1
1	1: 1	3	7: 1	5

$s(x\ y\ c_{in}) = x \,.\, y' \,.\, c'_{in} + x' \,.\, y \,.\, c'_{in} + x' \,.\, y' \,.\, c_{in} + x \,.\, y \,.\, c_{in}$

(b)

Figure 6. K-maps and logic expressions for (a) c_{out}, (b) s

c_{out} is the 3-bit majority function and can be minimized as shown in Figure 6*a*. Output s, however, cannot be simplified. It is pulled up only if an odd number of inputs are high as shown in Figure 6*b*. Remember from Chapter 2 that the output of an n-input XOR gate goes up only if the gate has an odd number of inputs at logic 1. Therefore, s may be realized with a 3-input XOR gate:

$$s = x \oplus y \oplus c_{in}$$

Binary Subtraction

We may use the paper-and-pencil subtraction algorithm to perform subtraction. To implement this algorithm, we first need to design a full subtractor, and then cascade n full subtractions, where n is the size of subtraction operands also called *subtractor size*. Another algorithm called *addition-based subtraction* and used in microprocessor design is more cost-effective when we need both addition and subtraction. In this technique, we modify the binary adder as demonstrated below:

Addition-based Subtraction

With some minor modifications, the adder (we designed above) is converted into a subtractor, such that the resulting hardware may be used for addition as well as subtraction (but not simultaneously of course!). In other words, subtraction is now carried out through addition as elaborated in this section.

In the n-bit binary system, $X - Y$ may be written as:

$$X - Y = X - Y + 2^n - 2^n = X + (2^n - Y) - \mathbf{2^n}$$

or

$$X + (2^n - Y) = X - Y + \mathbf{2^n} \qquad (2)$$

According to Equation (2), $X + (2^n - Y)$ consists of the required difference, ($X - Y$), plus a redundant term, $\mathbf{2^n}$. Therefore, if the following two problems are resolved, we may do addition $X + (2^n - Y)$ to perform subtraction $X - Y$.

1- The term ($2^n - Y$) on the left side of Equation (2) still requires a subtraction, but we don't want to do subtraction!
2- A redundant term, $\mathbf{2^n}$, exists on the right side of (2), so we have to get rid of that.

Resolve the first issue: It can be shown that subtraction $2^n - Y$ may be performed using the following subtraction-free algorithm:

$$2^n - Y = (\text{bit-wise complement of } Y) + 1 \qquad (3)$$

Definition: To get the bit-wise complement of Y, simply flip each bit of Y.

This means that instead of subtracting Y from 2^n mathematically, we may obtain the bit-wise complement of Y, plus 1. ($2^n - Y$) is called the *2's complement* of Y.

The above algorithm, (3), may be reworded as follows:

To obtain the 2's complement of Y: Starting with the LSb of Y, copy all the bits up to and including the first 1, and invert the remaining bits. (3')

Note: When using (3'), Y = 0 is a special case to be addressed in Example 8.

Example 5. Obtain $2^4 - 1001$

According to (3): $2^4 - 1001 = 0110 + 1 = 0111$

According to (3'): $2^4 - 1001 = 0111$

So, the first concern mentioned above is ruled out as we can perform $2^n - Y$ with no subtraction. More specifically

$$X + (\text{bit-wise complement of } Y) + 1 = X - Y + \mathbf{2^n} \qquad (4)$$

Resolve the second issue: "How can we get rid of the redundant term, $\mathbf{2^n}$, on the right of Equation (4)?" $\mathbf{2^n}$ is a 1 followed by n 0s, and this makes it straightforward to locate and ignore this redundant term, hence resolving the second issue. Look at the following example:

Example 6. In the 4-bit system ($n = 4$) use addition to do subtraction $X - Y = 1010 - 0101$.

Instead of subtraction $X - Y = 1010 - 0101$, we perform the following addition based on Equation (4):

$$X + (\text{bit-wise complement of } Y) + 1 = 1010 + 1010 + 1 = \mathbf{1}\ 0101 \qquad (5)$$

According to Equation (4), the sum, **1** 0101, is the result of the intended subtraction (1010 – 0101), plus the redundant term, $\mathbf{2^4}$, which is a **1** followed by four 0s: 10000. In other words, the fifth bit of the sum, the bold **1**, is the redundant term, which is the final carry generated by the addition, and therefore, can easily be identified and ignored. What is left (0101) is the correct subtraction result.

Let us see what is signified if a final carry is NOT generated when the left side of (4) is calculated. If $X < Y$ or $X - Y < 0$, then $X - Y + \mathbf{2^n}$, the right side of (4), becomes less than $\mathbf{2^n}$. This means that no final carry is generated in the addition on the left side of Equation (4). **So, no final carry means an impossible subtraction ($X < Y$).**

In summary, take the following steps to perform $X - Y$ in the n-bit unsigned system:

- Do zero extension to make the operands n bits wide if they are not.
- Perform $X + (\text{bit-wise complement of } Y) + 1$: left side of (4).
  - "Final carry" means "no borrow", i.e., subtraction ($X - Y$) is possible, or Y is NOT greater than X. Ignore the final carry; what is left is the correct subtraction result.
  - No "final carry" means "borrow", i.e., subtraction ($X - Y$) is impossible, or Y is greater than X. In other words, when "borrow" happens, the correct difference is not representable in the unsigned number system.

Example 7. Use addition to perform subtraction $X - Y = 1010 - 1101$

Instead of the subtraction 1010 – 1101, we perform the following addition according to Equation (4):

$$X + (\text{bit-wise complement of } Y) + 1 = 1010 + 0010 + 1 = \mathbf{0}\ 1101$$

No final carry is produced in this addition. Therefore, a final borrow occurs in the corresponding subtraction, indicating $X < Y$, an impossible subtraction.

There is a subtle point in addition-based subtraction when the 2nd number is 0, and Algorithm (3') is used to obtain the 2's complement of 0 as shown in the following example.

Example 8. Use addition to perform subtraction $X - Y = 0010 - 0000$

There is no 1 in Y. So, Algorithm (3') will generate 0000 as the 2's complement of Y. When this 0000 is added to 0010, the addition result would be 0010, which is the correct subtraction result. However, the addition does not produce a final carry, and this erroneously indicates that borrow is generated for the corresponding subtraction. To avoid this problem, we use Algorithm (3), which does generate carry if Y = 0. This algorithm is what we will use to design a subtractor in the next section.

$$0010 + 1111 + 1 = \mathbf{1}\ 0000 \text{ (carry is generated)}$$

Addition-based Subtractor Design

The addition-based subtraction algorithm described by Equation (4) has directly been translated into hardware in Figure 7 for $n = 4$, where n is the subtractor size. As shown in this figure, a four-bit ripple adder receives the "first 4-bit number", $X = x_3x_2x_1x_0$, and the bit-wise complement of the "second 4-bit number", $Y' = y'_3y'_2y'_1y'_0$, and produces the four-bit output $D = d_3d_2d_1d_0$. Additionally, a 1 is injected into the C_{in} of the least significant slice. So, this 4-bit adder performs the following addition:

$D = X + (\text{bit-wise complement of } Y) + 1$, which is exactly the left-side expression of Equation (4).

Consider c_{out3} (the carry-out from the most significant FA) in Figure 7. As explained above, in addition-based subtraction, a 1 on c_{out3} implies no final borrow for the subtraction. Similarly, a 0 on c_{out3} indicates a final borrow. That is why in Figure 7 c_{out3} is called *~borrow-out* or ~b_{out} for short. In other words, c_{out3} is now an active-low output representing the (final) borrow-out bit of this subtractor.

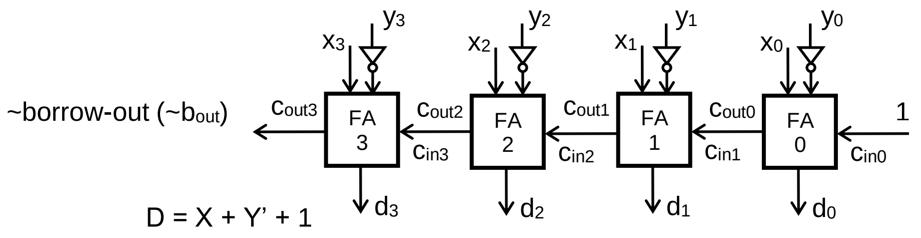


Figure 7. Adder-based subtractor

We may combine the four-bit adder shown in Figure 5a with the four-bit subtractor developed in Figure 7 to obtain an adder/subtractor unit depicted in Figure 8*a*. In this figure, the two operands are called $X = x_3x_2x_1x_0$, and $Y = y_3y_2y_1y_0$ but the result has two different names as explained shortly. (See also the logic symbol for this unit shown in Figure 8*b*.) This arithmetic unit has two operation modes, namely, add and subtract, and a mode-select control input called ~add/sub. When ~add/sub = 0, the add mode is selected, hence the output is called $S = s_3s_2s_1s_0$. Otherwise, if ~add/sub = 1, the subtract mode is entered and the output is called $D = d_3d_2d_1d_0$. As illustrated in Figure 8*a*, in the path of the second operand, Y, there are four programmable inverters (XOR gates) which are controlled by the ~add/sub line. This control line also drives the c_{in} of the least significant FA. When ~add/sub = 0 (the add mode), the XOR gates are in the non-inverting mode, and $c_{in0} = 0$ (see Figure 8*a*), hence the resulting configuration becomes a four-bit ripple adder shown in Figure 5*a*. However, when ~add/sub = 1 (the subtract mode), the XOR gates are converted to four inverters, such that the second operand of each FA becomes the complement of the corresponding bit in Y as illustrated in Figure 7. Additionally, a 1 is applied to the c_{in} input of the least significant FA because now ~add/sub = 1. These are exactly what we need to implement the addition-based subtraction algorithm modeled by Equation (4). c_{out3} from FA3 plays two different roles in the two operation modes: In the add mode, c_{out3} is the final carry bit. However, in the subtract mode, a 0 on c_{out3} indicates borrow or an impossible subtraction ($X < Y$). Otherwise, the subtraction is possible ($X \geq Y$). This is why we call this output c_{out}/~b_{out}, signifying that it is the (final) active-high carry-out bit in the add mode and the (final) active-low borrow-out bit in the subtract mode.

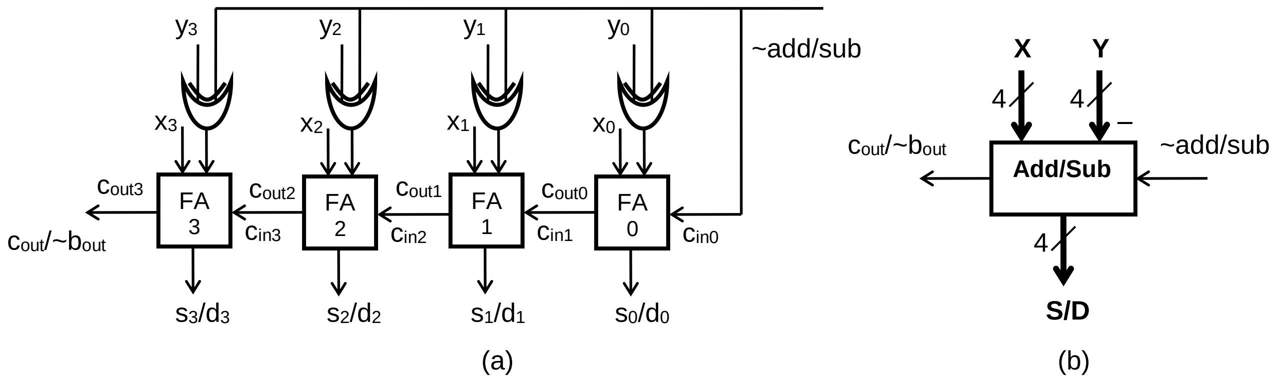


Figure 8. Adder/subtractor unit: (a) logic diagram, (b) symbol

Special Case: Incrementer/Decrementer

In the 4-bit adder/subtractor of Figure 8, let us assume that operand Y equals *constant* n. Such a 4-bit adder/subtractor is called a 4-bit "incrementer/decrementer by n". Let us abbreviate it to "Inc/Dec". A frequently used value for n, the increment, is '1'. In this book, we will assume that n equals '1' unless otherwise specified. For a wider Inc/Dec, we need a wider adder/subtractor. Figure 9*a* illustrates the logic diagram of a 4-bit Inc/Dec. A logic symbol for this circuit is shown in Figure 9*b*. Look at the changes made to Figure 8 to reach Figure 9. Input X has not changed. Input Y in Figure 8 is replaced with constant "0001" in Figure 9, where a more meaningful name, ~inc/dec, is assigned to control input ~add/sub in Figure 8. The new name means that a 0 applied to this input will take the Inc/Dec block to the *increment* mode (or Inc mode for short), where a 1 is added to input X. The Inc/Dec block will be taken to the *decrement* mode (or Dec mode for short) should the input ~inc/dec be tied to a logic 1. In the decrement mode, a 1 is taken away from input X. Label c_{out}/~b_{out} and its meaning do not change. Label S/D in Figure 8 changes to R = $r_3r_2r_1r_0$ in Figure 9 for ease of typing. The intermediate carry bits remain unchanged. The operation of the circuit illustrated in Figure 9 may be summarized as follows:

If ~inc/dec = 0 (Inc mode) then R = X + 1 and c_{out}/~b_{out} = carry-out for this addition.

If ~inc/dec = 1 (Dec mode) then R = X – 1 and c_{out}/~b_{out} = borrow-out for this subtraction.

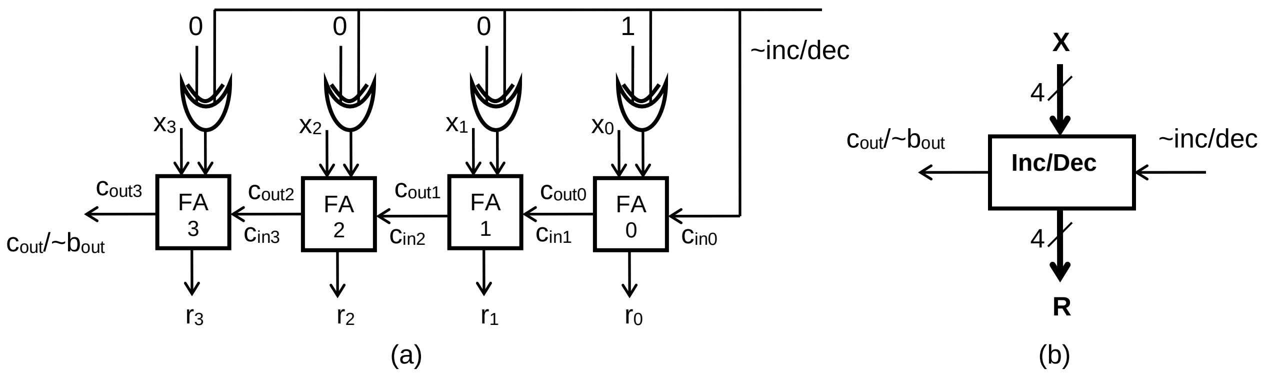


Figure 9. Inc/Dec: (a) logic circuit, (b) symbol

Question: What is the result if X = 1111 and ~inc/dec = 0?

Answer: R = 1111 + 1 = 0000

Question: What is the result if X = 0000 and ~inc/dec = 1?

Answer: R = 0000 – 1 = 1111

The rest of this section continues in Appendix A. You may skip the appendix on first reading.

Half Adders

When we talk about full adders, a question that may arise is "Do we have *non*-full adders as well?" The answer is "Yes, we also have *half adders*". A half adder (or HA for short) adds two single bits, x and y, (instead of three bits in a FA), and produces two output bits, c_{out} and s, as shown in the truth table of Figure 10*a*. Figure 10*b* shows the minterm lists of c_{out} and s, and Figure 10*c* illustrates a logic symbol for a HA. Half adders are used less frequently than FAs. A HA can replace FA0 in Figure 5*a* because the c_{in} of the least significant stage of an adder is always 0. In other words, this stage has only two inputs, x_0 and y_0.

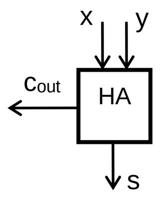
Row	x y	c_{out}	s
0	0 0	0	0
1	0 1	0	1
2	1 0	0	1
3	1 1	1	0

(a)

$s(x, y) = \Sigma\ (1, 2)$

$c_{out}(x, y) = \Sigma\ (3)$

(b)



(c)

Figure 10. Half adder (a) truth table, (b) symbol

Exercise: Show that instead of four FAs, four HAs can also be used to design an incrementor. Can you further simplify HA0?

Signed Number Systems

The number system that we have considered so far is unsigned. In this section, three different signed number systems, including the 2's complement system, are introduced to represent negative numbers and positive numbers together, and then addition/subtraction in the 2's complement system is addressed.

Sign-and-Magnitude Number System

The signed number system that we use every day is called *sign-and-magnitude* or S&M for short, in which a *sign* (either + or –) is added to the front of an unsigned number, the *magnitude*. For example, +2391 and -924 are two numbers in the decimal S&M system. The sign is a single bit as it may take either of two possible values. The same concept of sign-and-magnitude may be applied to binary numbers as well. For example, -110 (-6) and +101 (+5) are two binary numbers in the S&M system. Traditionally, a 1 or a 0 in the most significant bit position stands for the negative or positive sign, respectively. Therefore, 1110 = -110 = -6 and 0101 = +101 = +5. For the n-bit S&M number system, parameters P, R, B, Nmax, and Nmin are obtained as follows:

Number of bit patterns:	$P = 2^n$
Number of numbers:	$B = 2^n - 1$
Range of numbers:	$R = [-(2^{n-1} - 1) : +(2^{n-1} - 1)]$
Smallest number:	$Nmin = -(2^{n-1} - 1)$
Largest number:	$Nmax = 2^{n-1} - 1$

In this system, number 0 has two different representations, namely 1000, and 0000 for n = 4. This is why B, the number of representable numbers = P – 1.

Example 9. Obtain P, B, R, Nmin, and Nmax for the 4-bit S&M system:

Number of bit patterns:	P = 16
Number of numbers:	B = 15
Range of numbers:	R = [-7 : + 7]
Smallest number:	Nmin = -7
Largest number:	Nmax = +7

1's Complement Number System

In the 1's complement system, a positive number has the same representation it has in the S&M system. However, a negative number is the bit-wise complement or *1's complement* of the corresponding positive number. The MSb is still the sign bit. In this system, -6, for example, is represented as 1001 because +6 = 0110. If a number is 1's complemented twice, it is left unchanged. In the n-bit one's complement system, parameters P, R, B, Nmax, and Nmin (defined in Chapter 1) are obtained as follows:

Number of bit patterns:	$P = 2^n$
Number of numbers:	$B = 2^n – 1$
Range of numbers:	$R = [-(2^{n-1} – 1) : +(2^{n-1} – 1)]$
Smallest number:	$Nmin = -(2^{n-1} – 1)$
Largest number:	$Nmax = 2^{n-1} – 1$

Again, number 0 has two different representations in this system, namely 1111, and 0000 for $n = 4$. This is why $B = P – 1$.

Example 10. Obtain P, B, R, Nmin, and Nmax for the 4-bit 1's complement number system:

Number of bit patterns:	P = 16
Number of numbers:	B = 15
Range of numbers:	R = [-7 : + 7]
Smallest number:	Nmin = -7
Largest number:	Nmax = +7

2's Complement Number System

In the 2's complement system, a positive number has the same representation it has in the S&M or 1's complement system. However, negative numbers are obtained and represented according to algorithm (3) or (3') mentioned in the previous section. The MSb is still the sign bit. Algorithms (3) and (3') have been reworded and shown here as (6) and (6'), respectively, to be used in the 2's complement context:

– N (additive inverse of N) = (bit-wise complement of N) + 1 (6)

Or

To obtain the additive inverse of N, starting with the LSb, copy all the bits up to and including the first 1, and then complement all the remaining bits. (6')

-N is called the 2's complement of N. The 2's complement of 00…0 is always 00…0.

If a number is 2's complemented twice, it is left unchanged.

Example 11. Negate N = 0101 (+5) in the 4-bit 2's complement system:

Use (6):	-N = 1010 + 1 = 1011 (-5)
Use (6'):	-N = 1011 (-5)

Figure 11 shows all different 4-bit patterns and their interpretations in the 2's complement system.

In the n-bit 2's complement system, parameters P, R, B, Nmax, and Nmin are obtained as follows:

Number of bit patterns:	$P = 2^n$
Number of numbers:	$B = 2^n$
Range of numbers:	$R = [-2^{n-1} : +(2^{n-1} – 1)]$
Smallest number:	$Nmin = -2^{n-1}$
Largest number:	$Nmax = 2^{n-1} – 1$

4-bit number	Decimal equivalent	4-bit number	Decimal equivalent
0000	0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

Figure 11. Four-bit numbers in 2's complement system and their decimal equivalents

Example 12. Obtain P, B, R, Nmin, and Nmax for the 4-bit 2's complement system:

Number of bit patterns:	P = 16
Number of numbers:	B = 16
Range of numbers:	R = [-8 : +7]
Smallest number:	Nmin = -8
Largest number:	Nmax = +7

In the 2's complement system and unlike the first two signed systems, number 0 has only one representation in which all bits are 0s. This is why now B = P. The bit pattern saved this way (1000 in the 4-bit system) now becomes the most negative number (-8 in the 4-bit system) making the range of negative numbers greater than that of positive numbers by one. Therefore, be aware that the most negative number cannot be negated in the 2's complement system. For example, for n = 4, if 1000 (-8) was negated, the result would be 1000 which is still -8! This means that -8 cannot be negated because the correct result, -(-8) = +8, is out of range (see Example 12). In other words, *overflow* occurs when -8 is negated in the 4-bit 2's complement system. We will see more about overflow shortly.

We use the terminology *2's complement* for two different purposes: The 2's complement system, as explained above, and the 2's complement of a number, which means the additive inverse or opposite of a number in the 2's complement system.

Example 13. Interpret the binary bit pattern 101110 in the following 4-bit number systems.

Unsigned	S&M	1's complement	2's complement
101110 = 46	101110 = -01110 = -14	101110 = -010001= -17	101110 = -010010 = -18

Example 14. Represent decimal -50 in the 2's complement system. Use as few bits as possible.

+50 = 0110010;
-50 = -0110010;
Use Algorithm 8': -0110010 = 1001110

Addition in 2's Complement System
The addition of two same-sign numbers in the S&M system is straightforward: The magnitude of the result is the sum of the magnitudes of the two operands. Additionally, the result has the same sign as the operands. However, the addition of two different-sign numbers needs more work: The smallest magnitude has to be

determined first and then a proper subtraction must be carried out accordingly. Finally, the sign of the largest magnitude has to be assigned to the result. This procedure is simplified in the 1's complement system. However, "unsigned" and "1's complement signed" additions are still different from each other. Additionally, and as mentioned before, this system has two different representations for number zero. These two problems are solved in the 2's complement system at the cost of some asymmetry (see Example 12) and also more difficult negation (see algorithm (6) or (6')). It can be shown that **if two numbers in the 2's complement system are added using an unsigned addition algorithm, the result would be valid in the 2's complement system, no matter what the signs of the numbers are**, i.e., we do not need a special adder to add two signed numbers in this system. In this addition, the sign bits are treated similar to other bits of the operands, which may seem unusual!

Example 15. Add the 4-bit signed numbers: 0011 (= +3) + 0100 (= +4)

As mentioned above, 2's complement additions can be performed using the paper-and-pencil addition algorithm:

$$\begin{array}{rrl}
 & \text{sign bit} & \\
 & 0\ 011 & = +011 = +3 \\
 & +\ 0\ 100 & = +100 = +4 \\ \hline
\text{No carry } \textcircled{0} & 0\ 111 & = +111 = +7
\end{array}$$

The sum generated by this algorithm is 0111, which is the correct result in the 4-bit 2's complement system.

Example 16. Add the 4-bit signed numbers: 0011 (= +3) + 1100 (= -4)

$$\begin{array}{rrl}
 & \text{sign bit} & \\
 & 0\ 011 & = +011 = +3 \\
 & +\ 1\ 100 & = -100 = -4 \\ \hline
\text{No carry } \textcircled{0} & 1\ 111 & = -001 = -1
\end{array}$$

The 4-bit sum is 1111, which is the correct result (-1) in the 4-bit 2's complement system.

Example 17. Add the 4-bit signed numbers: 1101 (= -3) + 1100 (= -4)

$$\begin{array}{rrl}
 & \text{sign bit} & \\
 & 1\ 101 & = -011 = -3 \\
 & +\ 1\ 100 & = -100 = -4 \\ \hline
\text{Carry } \textcircled{1} & 1\ 001 & = -111 = -7
\end{array}$$

In this addition, carry is generated. But **it can be shown that the final carry is not part of the sum and should be removed.** So, the 4-bit sum of this addition is **1**001, which is the correct result (-7) in the 4-bit 2's complement system. Notice that sum is always as wide as each of the operands. Moreover, the position of the sign bit is always fixed: it is the MSb of the result.

Example 18. Add the 4-bit signed numbers: 1101 (= -3) + 0100 (= +4)

$$\begin{array}{rlll}
 & \textbf{sign} & & \\
 & \textbf{bit} & & \\
 & 1\ 101 & = -011 = -3 \\
 + & 0\ 100 & = +100 = +4 \\
\hline
\text{Carry } \boxed{1} & 0\ \ 001 & = +001 = +1
\end{array}$$

Here, a final carry is again produced, but we ignore it as we did in Example 17. The 4-bit sum is **0**001, which is the correct result (+1) in the 4-bit 2's complement system.

Example 19. Add the 4-bit signed numbers: 0110 (= +6) + 0101 (= +5)

$$\begin{array}{rlll}
 & \textbf{sign} & & \\
 & \textbf{bit} & & \\
 & 0\ 110 & = +110 = +6 \\
 + & 0\ 101 & = +101 = +5 \\
\hline
\text{No carry } \boxed{0} & 1\ \ 011 & = -101 = -5
\end{array}$$

The 4-bit result of this addition (with no final carry) is **1**011 or -5_{ten}. In this example, two positive numbers, namely +6 and +5 are added but the result looks like a negative number, -5! What is wrong with this addition? The answer is: Since the correct sum of this addition is too large to fit in the designated number of bits, 4, the resulting sum (**1**011) looks negative. More specifically, 6 + 5 = +11, but from Example 12 we know that the largest number representable in the 4-bit 2's complement system is +7. Therefore, the correct result, +11, cannot be represented in four bits, and this means *overflow* has occurred.

Example 20. Add the 4-bit signed numbers: 1001 (= -7) + 1100 (= -4)

$$\begin{array}{rlll}
 & \textbf{sign} & & \\
 & \textbf{bit} & & \\
 & 1\ 001 & = -111 = -7 \\
 + & 1\ 100 & = -100 = -4 \\
\hline
\text{Carry } \boxed{1} & 0\ 101 & = +101 = +5
\end{array}$$

Here, a final carry is produced, but we ignore it as usual. Now the 4-bit result is **0**101 or $+5_{ten}$. In this example, two negative numbers, namely -7 and -4, are added but the result (**0**101) looks like a positive number, +5! The reason is that the correct sum is too negative to fit in the designated number of bits, 4. More specifically, (-7) + (-4) = -11, but the most negative number representable in the 4-bit 2's complement system is -8 (see Example 12). Therefore, the correct result cannot be represented in four bits, and this again means that overflow has happened.

Definition: We say *overflow* has occurred if the result of an operation does not fit in the designated number of bits. In other words, if the result of an operation falls outside the range of representable numbers, *overflow* has happened.

In a 4-bit unsigned addition, if a final carry occurs, the addition result becomes 5 bits wide, hence falling out of range, i.e., in an unsigned addition, a final carry means "overflow" and vice versa. This, however, is not the case with 2's complement additions as shown above. "Carry" and "overflow" are two different events, such that we may come across any of the four possible combinations of presence/absence of them:

neither carry nor overflow (see Example 15 and Example 16), carry, but no overflow, (see Example 17 and Example 18), no carry but overflow, (see Example 19), and finally both carry and overflow (see Example 20). In general, by "overflow" we mean "2's complement overflow" unless otherwise specified.

The sum of two different-sign operands will definitely not go beyond either of the two operands, and since the operands are valid numbers, the sum would also be valid. In other words, the addition of two different-sign operands will never produce overflow. So, in a 2's complement addition overflow may happen only if the two operands have the same sign.

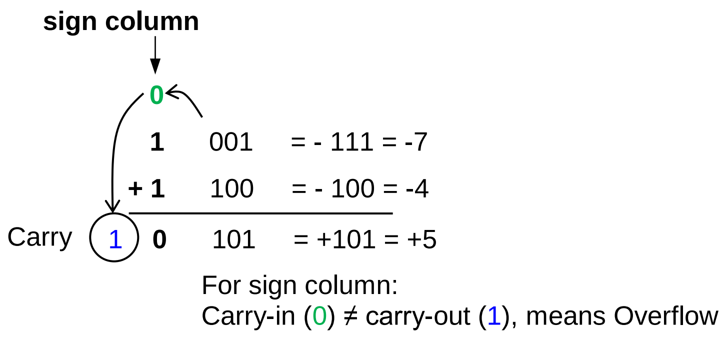
Lemma 1 shows the necessary and sufficient conditions to produce overflow in an addition:

Lemma 1. In 2's complement addition, overflow occurs if and only if 1) the two operands have the same sign, and 2) this sign is different from the sign of the sum.

It can be proved that Lemma 1 is equivalent to the following lemma:

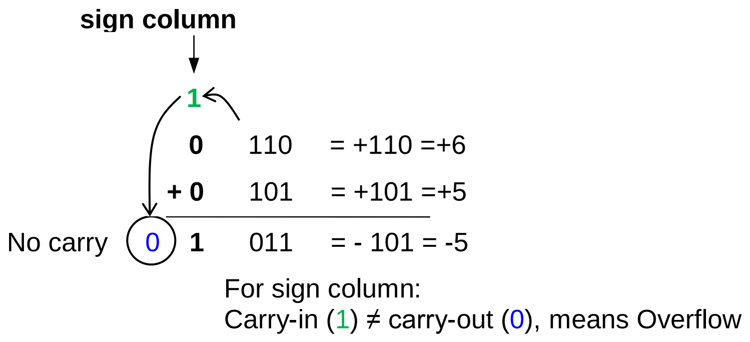
Lemma 2. In 2's complement addition, overflow occurs if and only if the carry bit arriving at the sign column is different from the carry bit leaving this column.

Example 21. Add the 4-bit signed numbers: 1001 (= -7) + 1100 (= -4)


sign column
0
1 001 = - 111 = -7
+ 1 100 = - 100 = -4
Carry 1 0 101 = +101 = +5
For sign column:
Carry-in (0) ≠ carry-out (1), means Overflow

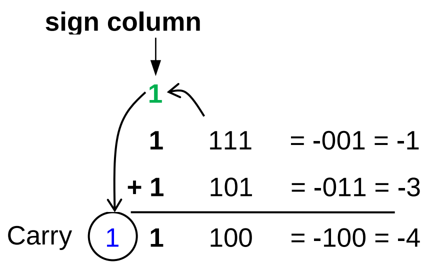
The carry bits arriving at and leaving the sign column are different (**0** ≠ **1**). Therefore, overflow occurs in this addition.

Example 22. Add the 4-bit signed numbers: 0110 (= +6) + 0101 (= +5)


sign column
1
0 110 = +110 =+6
+ 0 101 = +101 =+5
No carry 0 1 011 = - 101 = -5
For sign column:
Carry-in (1) ≠ carry-out (0), means Overflow

The carry bits arriving at and leaving the sign column are again different (1 ≠ 0). Therefore, overflow occurs in this addition as well.

Example 23. Add the 4-bit signed numbers: 1111 (= -1) + 1101 (= -3)



For sign column:
Carry-in (1) = carry-out (1), means NO Overflow

The carry bits arriving at and leaving the sign column are now the same. Therefore, no overflow occurs in this addition.

It can also be shown that:

1-The addition of two negative numbers always produces a final carry, no matter whether or not overflow occurs.

2-The addition of two positive numbers never produces a final carry, no matter whether or not overflow occurs.

3-The addition of two different-sign numbers may or may not produce a final carry.

Sign Extension
We may widen an unsigned number by zero extension as shown in Example 4. The same need may also frequently arise for signed numbers. However, the remedy could now be different.

Example 24. Add in the 4-bit 2's complement system: 1001 (= -7) + 011 (= 3)

Since the system is 4 bits wide, we need to make the second operand 4 bits wide as well. This is again carried out by appending a 0 to the left of this number because zero extension does not change the value of a positive number. Remember, the sign bit is always the most significant bit.

011 = 0011 => 1001 + 011 = 1001 + 0011 = 1100 = -4

So, positive numbers (similar to unsigned numbers) may be widened by zero extension.

Example 25. Add in the 4-bit 2's complement system: 1011 (= -5) + 110 (= -2)

The second operand is again three bits wide, hence has to be extended by one bit. However, this cannot be done by inserting a 0 because this 0 will convert the negative operand 110 (-2) to a positive number, namely 0110 = +6. It can be shown that *if 1s are appended to the left of a negative number, the value of the number will not change.* Therefore, in this example 110 is first converted to 1110 by appending a 1 to the left. Then the addition is performed as usual.

1011 + 110 = 1011 + 1110 = **1** 1001

Definition: To widen a number represented in the 2's complement system, we have to append as many copies of the sign bit as we need to the left of the number. This is called *sign extension*.

In summary, take the following steps to add two numbers in the n-bit 2's complement system:
- Do sign extension to make the operands n bits wide if they are not.
- Add the two operands using, say, the paper-and-pencil algorithm. In this addition, sign bits are treated similar to other bits. The sign bit of the result is the nth bit (and not the (n+1)th bit) from the right. Note

that for signed addition we use the same addition algorithm and therefore the same hardware that we used for the unsigned system.

- Using Lemma 1 or Lemma 2, check to see if overflow has occurred:
    - If it has, the result is invalid.
    - If has not, ignore the carry bit. The remaining n bits are the correct sum.

Two's Complement Subtraction

In a signed system, every subtraction may be rewritten in the form of addition because now different-sign numbers are representable: A – B = A + (-B). For example, instead of saying subtract 3 from -12, we could say add -3 to -12 (i.e., -12 – 3 = -12 + (-3)). Or, instead of saying subtract -5 from 14, we could say add 5 to 14 (i.e., 14 – (-5) = 14 + 5), and so on. In other words, if the "second number" is negated, the subtraction will be converted to an addition. Let us use Algorithm (6) for this negation:

$$X - Y = X + (\text{bit-wise complement of } Y) + 1 \qquad (7)$$

But this is exactly what did for unsigned subtraction as expressed in (4). So we use the same subtraction algorithm and therefore hardware that we used for the unsigned substation.

Example 26. Subtract the 4-bit signed numbers: 0111 (= +7) – 0011 (= +3)

Use (7) to perform subtraction:

```
Sign column
                1
             + 0  111   first number
             + 1  100   bit-wise complement of second number
             ----------
Carry   1      0  100   = +4
```

Carry occurs in this addition. **But it can be proved that the carry is not part of the result**. Remove it; the remaining bits (0100) are the correct subtraction result. This is similar to what we did for signed additions in the previous section. Also, no overflow is generated in this subtraction according to the following lemma:

Lemma 1'. In 2's complement subtraction modeled by (7), overflow occurs if and only if 1) the two operands have different signs, and 2) the sign of the result is different from that of the *first number*.

Example 27. Subtract in the 4-bit 2's complement system: 1100 (= -4) – 101 (= -3)

The "second number" is 3 bits wide, so first do sign extension by 1 bit: 101 = 1101.

The subtraction has two same-sign operands. So, there is no overflow according to Lemma 1'. Use (7) to carry out the subtraction:

```
Sign column
                1
             + 1  100   first number
             + 0  010   bit-wise complement of second number
             ----------
No carry  0    1  111   = -1
```

We could also perform the above addition in two separate steps:

The "1st number" = 1100

The 2's complement of the "2nd number" = 0010 + 1 = 0011

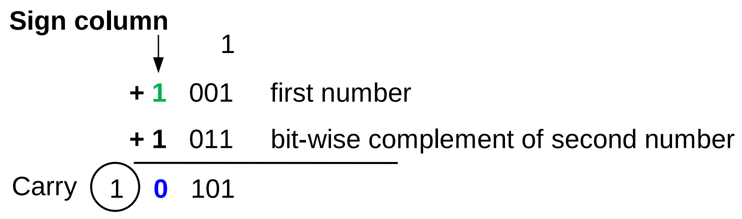
Add them up: 1100 + 0011 = 0 1111

Ignore the carry bit; what is left is the correct difference: 1111 = **-**1.

Let us look at the subtraction operands 1100 and 1101 as unsigned numbers: Since 1100 < 1101, borrow happens in this subtraction.

Example 28. Subtract the 4-bit signed numbers: 1001 (= **-**7) – 0100 (= +4)

Both operands are already 4 bits wide. So, no sign extension is required. Use (7) to perform the subtraction.



Different-sign subtraction operands:
Sign of 1st number (1) ≠ sign of difference (0), means Overflow

We could also perform the above addition in two separate steps:

The “1st number” = 1001

The 2’s complement of the “2nd number” = 1011 + 1 = 1100

Add them up:

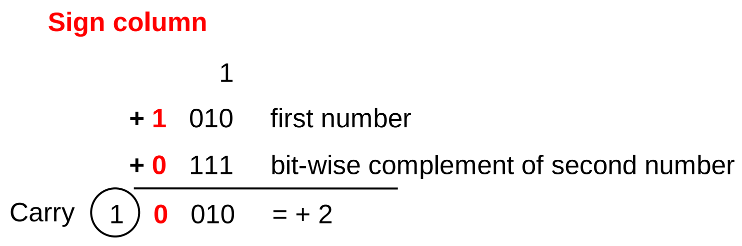
1001 + 1100 = 1 0101

The subtraction has two different-sign operands. The first operand and the result have different signs (1 ≠ 0). So, overflow happens according to Lemma 1’.

Let us look at the subtraction operands 1001 and 0100 as unsigned numbers: Since 1001 > 0100, borrow does not happen in this subtraction.

Example 29. Subtract the 4-bit signed numbers: 1010 (= -6) – 1000 (= -8)

Both operands are already 4 bits wide. Therefore, no sign extension is required. The subtraction has two same-sign operands. So, there is no overflow according to Lemma 1’. Use (7) to perform the subtraction:



Let us perform the above addition in separate steps:

The “1st number” = 1010

The 2’s complement of the “2nd number” = 0111 + 1 = 1000 (Special case, Overflow!)

In this complementing step overflow occurs, but we can ignore it!

Add them:

1010 + 1000 = 1 0010 = +2. (Ignore the carry bit.)

Let us look at the subtraction operands 1010 and 1000 as unsigned numbers: Since 1010 > 1000, no borrow is generated in this subtraction.

Lemma 3 is the extended version of Lemma 2. Lemma 3 covers subtraction as well:

Lemma 3. In 2's complement subtraction modeled by Equation (7), and also in 2's complement addition, overflow occurs if and only if the carry bit arriving at the sign column is different from the carry bit leaving this column.

Equation (7) is shown here again for ease of reference:

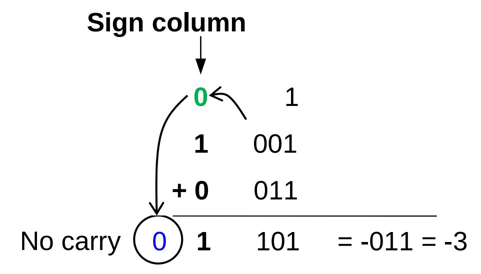
$$X - Y = X + (\text{bit-wise complement of } Y) + 1 \qquad (7)$$

In summary, take the following steps to perform ($X - Y$) in the n-bit 2's complement system:
- Do sign extension to make the operands n bits wide if they are not.
- Perform X + (bit-wise complement of Y) + 1.
- Use Lemma 1' or Lemma 3 to see if overflow has occurred:
  - If it has, the subtraction result is not representable in n bits.
  - If it has not, ignore the carry bit. The remaining n bits are the correct subtraction result.

Example 30. Subtract the 4-bit signed numbers: 1001 (= -7) – 100 (= -4)

First, let us do sign extension to make the second operand 4 bits wide as well and then use Algorithm (7).

The second operand: 100 = 1100



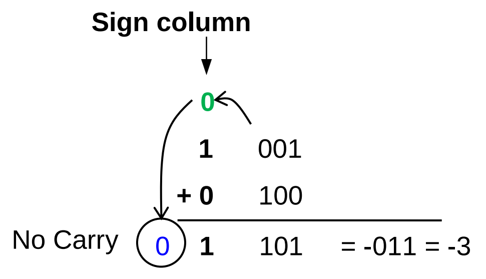
For sign column:
Carry-in (0) = carry-out (0), means No Overflow

We may also perform the above addition in separate steps: (See the special case in Example 32.)

The "1st number" = 1001

The 2's complement of the "2nd number" = 0011 + 1 = 0100

Add them up:



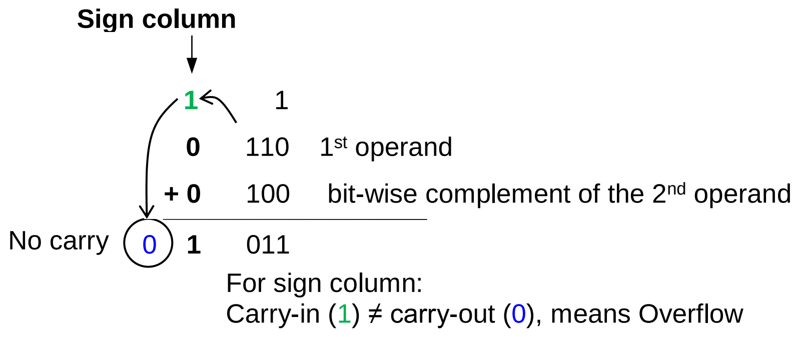
For sign column:
Carry-in (0) = carry-out (0), means No Overflow

The carry bits arriving at and leaving the sign column are the same. Therefore, this subtraction is overflow-free according to Lemma 3.

Let us look at the subtraction operands 1001 and 1100 as unsigned numbers: Since 1001 < 1100, borrow occurs in this subtraction.

Example 31. Subtract the 4-bit signed numbers: 0110 (= +6) – 1011 (= -5)

The operands are already 4 bits wide. Use Algorithm (7):

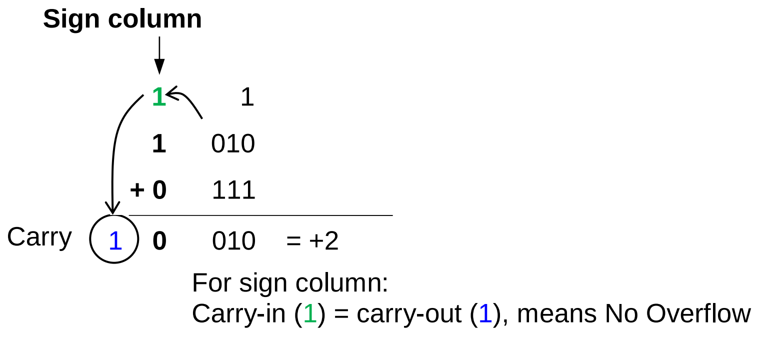

Sign column
1 1
0 110 1st operand
+ 0 100 bit-wise complement of the 2nd operand
No carry 0 1 011
For sign column:
Carry-in (1) ≠ carry-out (0), means Overflow

The carry bits arriving at and leaving the sign column are different. Therefore, overflow occurs according to Lemma 3.

Let us look at the subtraction operands 0110 and 1011 as unsigned numbers: Since 0110 < 1101, borrow is generated in this subtraction.

Example 32. (Special case) Subtract the 4-bit signed numbers: 1010 (= -6) – 1000 (= -8)

The operands are already 4 bits wide. Use Algorithm (7):



The carry bits arriving at and leaving the sign column are the same. Therefore, there is no overflow according to Lemma 3.

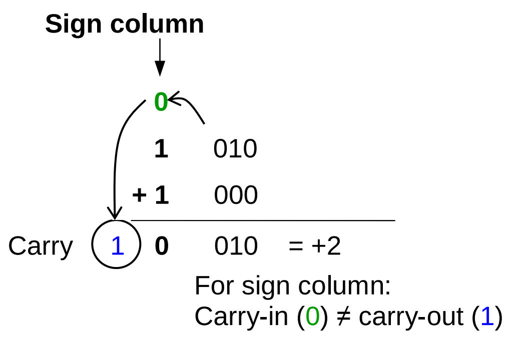
Let us perform the above addition in separate steps:

The “1st number” = 1010

The 2’s complement of the “2nd number” = 0111 + 1 = 1000 (Overflow!)

Overflow occurs in this complementing step. This will be problematic in the next step:

Add them:



The carry bits arriving at and leaving the sign column are different. However, there is no overflow as the two subtraction operands have the same sign!

Conclusion: (Special case) When the second subtraction operand is the most negative number, perform the two additions in Algorithm (7) together if you want to use Lemma 3. This is what our arithmetic unit carries out in the following section:

Adder/Subtractor Design for Signed Numbers

As we saw above, the same addition/subtraction algorithms used for unsigned numbers can also be used for signed numbers in the 2's complement system. And this is the power of 2's complement system! Therefore, the same unsigned arithmetic unit developed in Figure 8*a* will properly work for signed additions/subtractions as well.

Overflow Detection

The arithmetic unit developed in Figure 8*a* has been redrawn in Figure 12 for ease of reference. The necessary and sufficient condition for overflow to occur is specified by Lemma 3. According to this lemma, if cout3 ≠ cin3 in Figure 12, overflow has happened for the addition or subtraction and vice versa. Therefore, overflow can be detected through an XOR gate as shown in Figure 12.

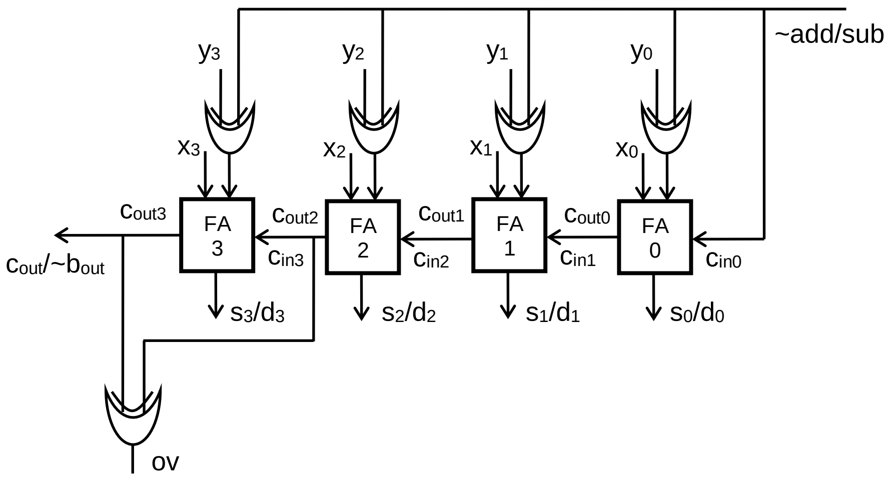


Figure 12. Overflow detection using an XOR gate

Comparators

An n-bit unsigned comparator compares two n-bit inputs, A and B, as two unsigned numbers and determines whether A < B, A > B, or A = B, by asserting the appropriate output bit. Figure 13*a* shows a logic symbol for such a comparator. Have you seen a comparator in this chapter? The answer is: almost yes! A subtractor is a comparator as well! When the difference of two operands (A – B) generates borrow, then B > A, otherwise A > B unless the difference is 0, which signifies A = B as shown in Figure 13*b*. Another technique for comparator design was introduced in Lab Exercise 4.

An interesting question is what happens if we apply two signed numbers to an unsigned comparator? For positive numbers, the comparator of course generates the right response. This is true for negative numbers as well. For example, consider 1001 (= -7) and 1100 (= -4). As two signed numbers, -4 is greater than -7. If these two numbers are interpreted as two unsigned numbers, then 1100 (12) is still greater than 1001 (9). So, two same-sign numbers are compared correctly by an unsigned comparator. This, however, is not the case with two different-sign numbers. Consider 1100 (-4) and 0101 (= 5). As two signed numbers 0101 > 1100, however, an unsigned comparator believes that 0101 < 1100. This problem can be solved by swapping the sign bits of the two numbers: now 0101 becomes 1101, and 1100 becomes 0100. With this change, an unsigned comparator will make the right decision! Swapping the sign bits does not affect same-sign numbers as they have anyway identical signs. Therefore, an unsigned comparator can be converted to a signed one by swapping the sign bits of the two numbers as illustrated in Figure 14.

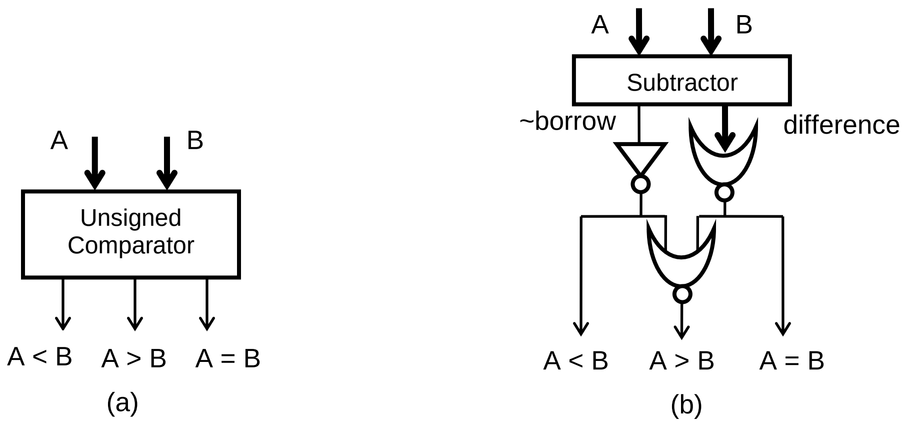


Figure 13. An unsigned comparator: (a) logic symbol, (b) subtractor-based logic circuit

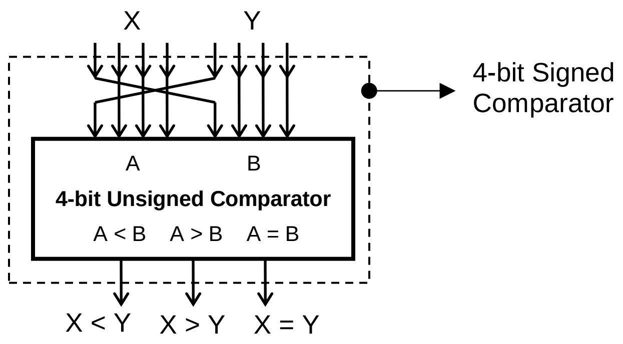


Figure 14. Signed comparison using an unsigned comparator

The above comparators can determine whether one number is equal to, less than, or greater than a second number. Let us call them *full* comparators as opposed to *partial* comparators, which are only able to determine whether or not two numbers are equal. The single cell of such a comparator, an XNOR gate, was introduced in Chapter 2. Figure 15 shows a 4-bit partial comparator with 2 x 4-bit inputs, A (= a_3 a_2 a_1 a_0), B(= b_3 b_2 b_1 b_0), and one active-high output, AEQB. The output is asserted only if A = B, otherwise, it would be deasserted. Partial comparators are of course superior in terms of the number of gates and propagation delay time.

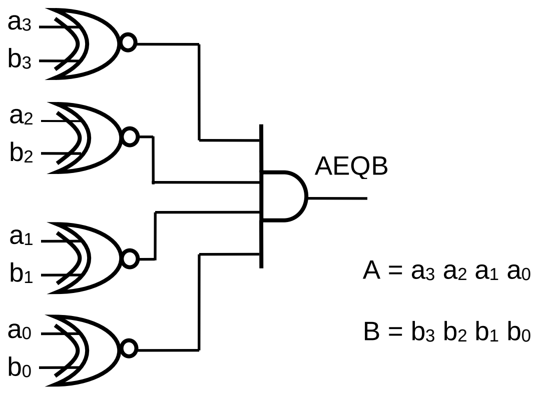


Figure 15. A four-bit partial comparator

A 4-bit partial comparator shown in Figure 15 represents a good example of non-systematic design. In this design methodology, a (complex) problem is decomposed into some smaller, hence easily solvable and/or already solved problems. Then the solutions to these small problems are stitched together to reach a complete solution for the whole problem. It goes without saying that the designer has to be creative to efficiently approach a problem non-systematically. For the above comparator, we would need a 256-row truth table should we want to use the systematic design procedure we learned in previous chapters. However, it is not easy to handle such a table, let alone a 2^{64}-row truth table that we would have needed if we had decided to apply the systematic design procedure to a 32-bit comparator! In Figure 15, we decomposed the 4-bit comparator into 4 single-bit comparators (with which we are already familiar) and then merged the four outputs through a 4-input AND gate to generate the final output. The same approach also works for a larger number of inputs.

Similar to adders and subtractors, comparators are also a must for microprocessors. If you do not know why, pay close attention to the rest of this page:

Consider the “while” and “if-then-else” constructs in the C language:

- While loop

```
while (A < B)
{
        body of loop
}
```

- If-then-else statement

```
if ( A < B)
{
    body for condition true
}
else
{
    body for condition false
}
```

The underlying microprocessor that executes these statements (after they are translated into the machine language) has to be able to compare two numbers, A and B; otherwise, we will not be able to create while (or other types of) loops or if-then-else statements, hence writing computer programs!

Octal and Hexadecimal Numbers

You have probably noticed how error-prone it is to work with long bit patterns. To significantly relax this problem, instead of base 2 (binary) we usually use base 8 (octal) or base 16 (hexadecimal) representations (or shorthand notations). Since each of these two bases is a power of 2, binary representations can conveniently be converted into octal or hexadecimal and vice versa.

Example 33. Convert N = **10** 111 101 110 001 010 to octal.

Starting with the *rightmost* bit, partition the binary number into groups of 3, and then convert each group into the equivalent *octal* digit. Octal digits are 0, 1, 2 … 7, which are the first eight decimal digits.

The length of N, 17, is not a multiple of 3. More specifically, the most significant group of bits, 10, is only 2 (not 3) bits wide in this example.

N = 10 111 101 110 001 010 = $2\ 7\ 5\ 6\ 1\ 2_{\text{octal}}$

In addition to the ten traditional digits, 0-9, the following six digits or symbols are also used in the hexadecimal system for which we need 16 different digits:

A: decimal 10 (= binary 1010)

B: decimal 11 (= binary 1011)

C: decimal 12 (= binary 1100)

D: decimal 13 (= binary 1101)

E: decimal 14 (= binary 1110)

F: decimal 15 (= binary 1111).

Example 34. Convert N = **110** 1110 1010 1001 1111 0111 1101 1100 to hexadecimal.

Starting with the *rightmost* bit, partition the binary number into groups of 4, and then convert each group into the equivalent *hexadecimal* digit. Hexadecimal digits are 0, 1, 2 … 9, A, B, C, D, E, and F.

The length of N, 31, is not a multiple of 4. More specifically, the most significant group of bits, 110, is only 3 (not 4) bits wide in this example.

N = 110 1110 1010 1001 1111 0111 1101 1100 = $6\ E\ A\ 9\ F\ 7\ D\ C_{\text{hexadecimal}}$

The reverse conversion is also straightforward: We need to replace each octal or hexadecimal digit with the equivalent bit pattern as shown in the following examples:

Example 35. Convert octal N = 1 3 0 0 7 3 2 4 5 6 1 6 to binary:

N = $(1\ 3\ 0\ 0\ 7\ 3\ 2\ 4\ 5\ 6\ 1\ 6)_{\text{octal}}$ = 1 011 000 000 111 011 010 100 101 110 001 110

Example 36. Convert hexadecimal N = 2 C 0 F D 4 B 8 E into binary.

N = $(2\ C\ 0\ F\ D\ 4\ B\ 8\ E)_{\text{hexadecimal}}$ = 10 1100 0000 1111 1101 0100 1011 1000 1110

Two More Codes

0s and 1s might be concatenated under different rules for different purposes. When a rule is applied, the resulting set of strings of 0s and 1s is called a *code*. We have seen different codes so far, such as BCD, one-out-of-m, and 7-segment. In this section, we take a closer look at Gray code, and then a new code called ASCII is introduced.

Gray Code

Two consecutive binary numbers may differ in more than one bit, such as 0111 (5) and 1000 (6), which differ in four bits. This may have some undesirable side effects such as the one to be elaborated on in

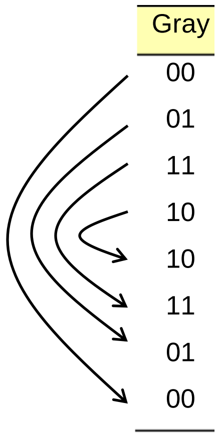
Chapter 9. Unlike the binary code, in Gray code, every two consecutive bit patterns (or consecutive *code words*) differ from each other in exactly one bit. Gray code may be generated recursively as follows:

- 0 and 1 are the two consecutive single-bit Gray code words. For these two code words n equals 1, where n is the number of bits in each code word. In general, the number of n-bit Gray code words is 2^n.
- Starting with $n = 1$, list 2^n x n-bit Gray code words followed by 2^n x n-bit Gray code words but in *reverse* order. This results in 2^{n+1} x n-bit code words with two instances per code word.
- Append a 0 to the left of each code word in the upper half of the above list. Append a 1 to the left of each code word in the lower half of the above list. The resulting list is the (n+1)-bit Gray code.

Example 37. Figure 16*a* shows two-bit Gray code words. These are the already-generated code words in this example. Let us use the above algorithm to obtain 3-bit Gray code words. In Figure 16*b*, the already-generated code words are followed by the same code words but in reverse order. Then, the first four code words (upper half) each receive a leading 0, and the last four code words (the lower half) each receive a leading 1 to eventually generate a 3-bit Gray code as shown in Figure 16*c*.

Decimal	Gray
0	00
1	01
2	11
3	10

(a)



(b)

Decimal	Gray
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

(c)

Figure 16. Recursive generation of a 3-bit Gray code: (a) 2-bit Gray code, (b) 2-bit code followed by reversed-order 2-bit code, (c) 3-bit Gray code

Figure 17 shows 4-bit Gray code words and their binary and decimal equivalents.

Decimal	Binary	Gray	Decimal	Binary	Gray
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Figure 17. Four-bit Gray code words, and their binary and decimal equivalents

A Gray code word may also be obtained from the corresponding binary code word as illustrated by the following rule. Bit 0 is the LSb, as usual.

- Append a 0 to the left of the binary number.
- If bits i and $i+1$ in the binary number are different, then bit i in the corresponding Gray code word will be 1, otherwise 0. **Note**: $0 \le i \le n - 1$, where n is the number of bits in a code word.

Example 38. Obtain a 4-bit Gray code word corresponding to binary 1100. (See row 12 in Figure 17)

Append a 0 to the left of binary 1100 to obtain binary 01100.

Let i = 0; bits 0 and 1 of binary 01100 both are 0s. So, bit 0 of the resulting Gray code word will be 0.

Let i = 1; bits 1 and 2 of binary 01100 are 0 and 1, respectively. So, bit 1 of the resulting Gray code word will be 1.

Let i = 2; bits 2 and 3 of binary 01100 are both 1s. So, bit 2 of the resulting Gray code word will be 0.

Let i = 3; bits 3 and 4 of binary 01100 are 1 and 0, respectively. So, bit 3 of the resulting Gray code word will be 1.

Therefore, the Gray code word corresponding to the binary number 1100 would be 1010 (see Figure 17).

ASCII

In addition to numbers, characters should also be represented in digital systems, or more specifically in computers. On the other hand, 0s and 1s are the only elements (building blocks) that may be put together to represent whatever we need in the digital world. This means that to digitally represent characters, we still need to assign a unique bit string or a *character code word* to each character. There are different character codes in the literature. The most widely used one is called ASCII, which stands for American Standard Code for Information Interchange. Each ASCII code word is 7 bits wide. So, in this code there are 128 (= 2^7) different code words assigned to 128 different characters including the numerals, uppercase, and lowercase alphabet, and nonprintable control characters as shown in Figure 18. The top row in this table shows the four LSbs of ASCII code words in hexadecimal. The three MSbs in the range of 0 to 7 are listed in the leftmost column. Therefore, to obtain the ASCII code word of a character we need to read the coordinates of that character in the ASCII table. For example, the ASCII code words for characters 7, b, and B are 37, 62, and 42, respectively, all in hexadecimal. The least-significant digits (7, 2, and 2) come from the top row, and the most-significant digits (3, 6, and 4) are taken from the leftmost column. We may also come across 8-bit extended ASCII in the literature. In addition to the 128 characters shown in Figure 18, the extended version covers special characters as well.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	**7**	8	9	:	;	<	=	>	?
4	@	A	**B**	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	**b**	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 18. ASCII table

Appendix A: Incrementers/Decrementers (Cont'd): (You may skip Appendix A on first reading.)

The logic diagram of Figure 9*a* will be simplified here:

Inc mode: First let us assume that ~inc/dec = 0. This means that the right input of each XOR gate in Figure 9*a* is tied to logic 0. Remember from Chapter 2 that a 2-input XOR gate with one input fixed at logic 0 is eliminated. Therefore, in the Inc mode, the Inc/Dec circuit can be simplified as illustrated in Figure 19. This circuit clearly shows that R = X + 1.

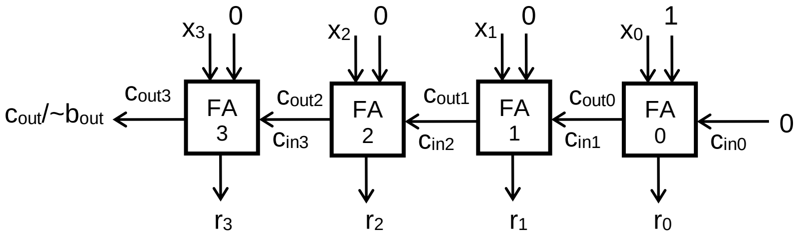

x_3 0 x_2 0 x_1 0 x_0 1
FA 3
FA 2
FA 1
FA 0
c_{out}/~b_{out}
c_{out3} c_{in3} c_{out2} c_{in2} c_{out1} c_{in1} c_{out0} c_{in0}
0
r_3 r_2 r_1 r_0

Figure 19. Inc/Dec in Inc mode

Dec mode: Now consider the second scenario, ~inc/dec = 1. This means that the right input of each XOR gate in Figure 9*a* is tied to logic 1. Remember from Chapter 2 that a 2-input XOR gate with one input fixed at logic 1 is functionally equivalent to an inverter. Therefore, in the Dec mode, the Inc/Dec circuit can be simplified as illustrated in Figure 20*a*. Note that the constant input bits are inverted and changed to "1110".

In Figure 20*a*, look at the least significant FA (FA0) and its three inputs, namely, x0, 0, and 1. According to the definition of a full adder, its operation result will not change if two arbitrary inputs of the full adder are swapped. Let us swap inputs 0 and 1 of FA0 as illustrated in Figure 20*b*.

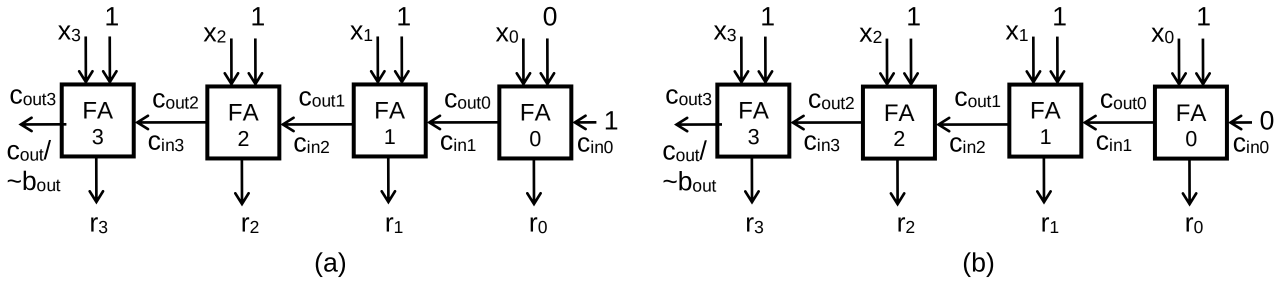

x_3 1 x_2 1 x_1 1 x_0 0
FA 3
FA 2
FA 1
FA 0
c_{out3} c_{in3} c_{out2} c_{in2} c_{out1} c_{in1} c_{out0} c_{in0}
1
c_{out}/ ~b_{out}
r_3 r_2 r_1 r_0
(a)
x_3 1 x_2 1 x_1 1 x_0 1
FA 3
FA 2
FA 1
FA 0
c_{out3} c_{in3} c_{out2} c_{in2} c_{out1} c_{in1} c_{out0} c_{in0}
0
c_{out}/ ~b_{out}
r_3 r_2 r_1 r_0
(b)

Figure 20. Two versions of Inc/Dec in Dec mode

Take a close look at the logic diagrams in the two operation modes illustrated in Figure 19 and Figure 20*b*. In both of them, c_{in0} = 0, and the upper-right input of FA0 is tied to 1. The upper-right inputs of FA1 through FA3 in Figure 19 are tied to 0, which is the same value applied to input ~inc/dec. The upper-right inputs of FA1 through FA3 in Figure 20*b* are tied to 1, which is again the same value applied to input ~inc/dec. Therefore, the two logic diagrams can be combined as illustrated in Figure 21.

We may easily verify the correctness of the design reached in Figure 21:

Based on what is seen in Figure 21 we may write:

R = X + 0001 if ~inc/dec = 0 (Inc mode)

R = X + 1111 if ~inc/dec = 1 (Dec mode)

The Inc mode is obviously correct. According to the addition-based subtraction rule, which you learned in this chapter, the addition "X + 1111" (in the Dec mode) generates the difference "X – 0001", which is what we need in the Dec mode.

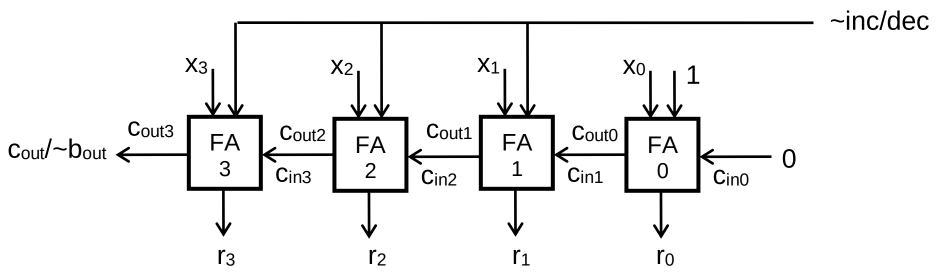
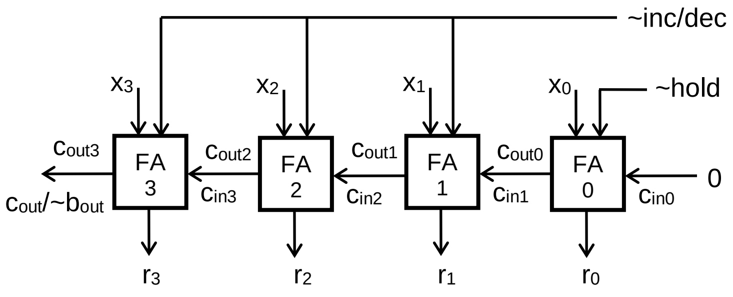


Figure 21. Simplified Inc/Dec

Modified Inc/Dec with HOLD Mode

The Inc/Dec of Figure 21 will be used to design a new circuit in Chapter 9. However, as you will see in Chapter 9, we usually need to add to Inc/Dec another feature called the *hold* mode. In this mode, the output holds the exact value of input X; in other words, R = X. The new feature is added to Inc/Dec and illustrated in Figure 22*a*, where the upper-right input of FA0 is now called *~hold*. The function table of the modified Inc/Dec circuit is shown in Figure 22*b*. As you see in Figure 22*a*, if both ~hold and ~inc/dec inputs are tied to logic 0, then output R will equal X + 0000, or R = X. Otherwise, with a '1' applied to input ~hold, the logic diagram of Figure 22*a* will change back to the original Inc/Dec illustrated in Figure 21. Figure 22*c* shows a logic symbol for the modified Inc/Dec.

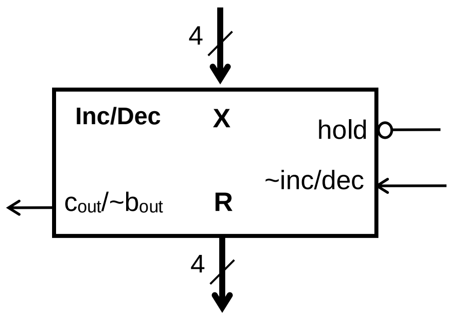
Exercise: In Figure 22*a*, what happens if inputs ~inc/dec and ~hold are tied to 1 and 0, respectively?



(a)

Mode	~inc/dec	~hold	R
Hold	0	0	X
Increment	0	1	X + 1
Decrement	1	1	X – 1

(b)



(c)

Figure 22. Inc/Dec with HOLD mode: (a) logic diagram, (b) function table, (c) logic symbol

Further Simplification of Inc/Dec

In this section, the Inc/Dec circuit of Figure 21 is further simplified. In this simplification, we will use the following two theorems, (I) and (II), which you proved in Homework Assignment 3:

Theorem (I): (A XOR B) . C + (A XOR D) . C' = A XOR (B . C + D . C')

Theorem (II): If the following two conditions are met:

If S = 1, then B = 0

If S = 0, then A = 0

Then it is always true to write:

$A . S + B . S' = A + B$

Back to Figure 21, let us first simplify FA0 and call the resulting circuit ID0, which stands for **Incrementer**/**Decrementer** cell number '**0**' or the least significant cell. We will then simplify the rest of the FAs to reach ID1, ID2, and ID3, which will be identical cells, fortunately.

FA0 has two constant inputs, 1 and 0. If we plug these values into the full adder equations of Figure 6, the following simplified equations will be reached for the carry-out (Cout0) and sum (r0) of FA0 no matter what the operation mode is:

$r0 = x0'$ (8)
$Cout0 = Cin1 = x0$

As you will see shortly, however, the following cell, ID1, will need Cout0 (= x0) and also Cout0' (= x0') in the Inc and Dec modes, respectively. Additionally, it will be necessary to force Cout0 and Cout0' to logic 0 in the Dec and Inc modes, respectively. (Be patient! you will see the reason soon.) Therefore, let ID0 generate the following two carry outputs to be used by ID1, in addition to r0 produced in Equation (8):

$C^{inc}_{out0} = x0 . (\sim inc/dec)'$	(9)	To be used in the Inc mode: ~inc/dec = 0
$C'^{dec}_{out0} = x0' . \sim inc/dec$	(10)	To be used in the Dec mode: ~inc/dec = 1

The logic diagram of ID0 is illustrated in Figure 23 following Equations (8) through (10):

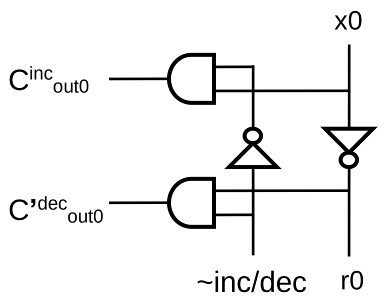


Figure 23. Logic diagram for ID0

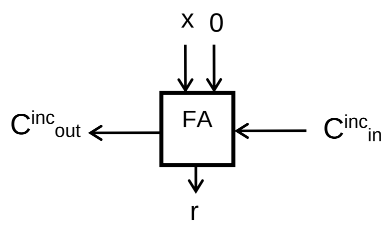
Each of the rest of the FAs in Figure 21 may be decomposed into two FAs for the two Inc and Dec modes. These two FAs along with their truth tables and logic equations are shown in Figure 24*a* and Figure 24*b*, respectively. Read them carefully, but do not worry! Soon, the two FAs will be combined and simplified as one ID cell.

Equations (11) and (12) may be merged as follows:

$r = (x \text{ XOR } C^{inc}_{in}) . \sim inc/dec' + (x \text{ XOR } C'^{dec}_{in}) . \sim inc/dec$ (15)

According to Theorem (I) on page 12, Equation (15) is simplified:

$r = x \text{ XOR } [(C^{inc}_{in} . \sim inc/dec') + (C'^{dec}_{in} . \sim inc/dec)]$ (16)

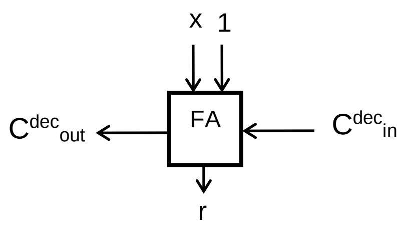


x	C^{inc}_{in}	C^{inc}_{out}	r
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$r = x \text{ XOR } C^{inc}_{in} \qquad (11)$$

$$C^{inc}_{out} = x \,.\, C^{inc}_{in} \qquad (13)$$

(a)



x	C^{dec}_{in}	C^{dec}_{out}	r
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	1

$$r = x \text{ XOR } C'^{dec}_{in} \qquad (12)$$

$$C^{dec}_{out} = (x' \,.\, C'^{dec}_{in})' \quad \text{or}$$

$$C'^{dec}_{out} = x' \,.\, C'^{dec}_{in} \qquad (14)$$

(b)

Figure 24. FA of Figure 21 in two operation modes (a) Inc mode, (b) Dec mode

We are going to combine the two FAs shown in Figure 24 to reach one ID cell. Then, we will be able to put as many ID cells in series as we need provided that we always use ID0 (designed in Figure 23) in the least significant position. To cascade IDs, the C^{inc}_{out} and C'^{dec}_{out} outputs of each ID cell must be tied to the C^{inc}_{in} and C'^{dec}_{in} inputs, respectively, of the following ID cell. Therefore, the logic equations for C^{inc}_{in1}, C^{inc}_{in2}, and C^{inc}_{in3} are derived as follows:

Since C^{inc}_{in1} will be tied to C^{inc}_{out0}, Equation (9) can be written as:

$$C^{inc}_{in1} = C^{inc}_{out0} = x0 \,.\, \sim inc/dec' \qquad (17)$$

According to Equation (13):

$$C^{inc}_{out1} = x1 \,.\, C^{inc}_{in1}$$

But C^{inc}_{out1} will directly be tied to C^{inc}_{in2}; therefore:

$$C^{inc}_{in2} = x1 \,.\, C^{inc}_{in1} \qquad (18)$$

Substitute for C^{inc}_{in1} from (17) into (18):

$$C^{inc}_{in2} = x1 \,.\, x0 \,.\, \sim inc/dec' \qquad (19)$$

According to Equation (13):

$$C^{inc}_{out2} = x2 \,.\, C^{inc}_{in2}$$

But C^{inc}_{out2} will directly be tied to C^{inc}_{in3}; therefore:

$$C^{inc}_{in3} = x2 \,.\, C^{inc}_{in2} \qquad (20)$$

Substitute for C^{inc}_{in2} from (19) into (20):

$$C^{inc}_{in3} = x2 \,.\, x1 \,.\, x0 \,.\, \sim inc/dec' \qquad (21)$$

The above analysis can be generalized to show that for an arbitrary ID, call it IDn, in an arbitrary-size Inc/Dec circuit, the C^{inc}_{in} input is generated by ANDing the ~inc/dec' input with all the less significant x inputs. In other words,

$$C^{inc}_{in-n} = xn - 1 . \dots x2 . x1 . x0 . \sim inc/dec' \qquad (22)$$

Similarly, it can be shown that

$$C'^{dec}_{in-n} = x'n - 1 . \dots x'2 . x'1 . x'0 . \sim inc/dec \qquad (23)$$

Equation (22) shows that in the Dec mode (~inc/dec = 1), the C^{inc}_{in} inputs of all IDs will be pulled down. Similarly, Equation (23) shows in the Inc mode (~inc/dec = 0), the C'^{dec}_{in} inputs of all IDs will be pulled down. Therefore and according to Theorem (II) on page 12, Equation (16) can be simplified:

$$r = x \text{ XOR } (C^{inc}_{in} + C'^{dec}_{in}) \qquad (24)$$

The logic diagram of an ID cell is illustrated in Figure 25*a* following Equations (13), (14), and (24). Figure 25*b* shows a logic symbol for ID.

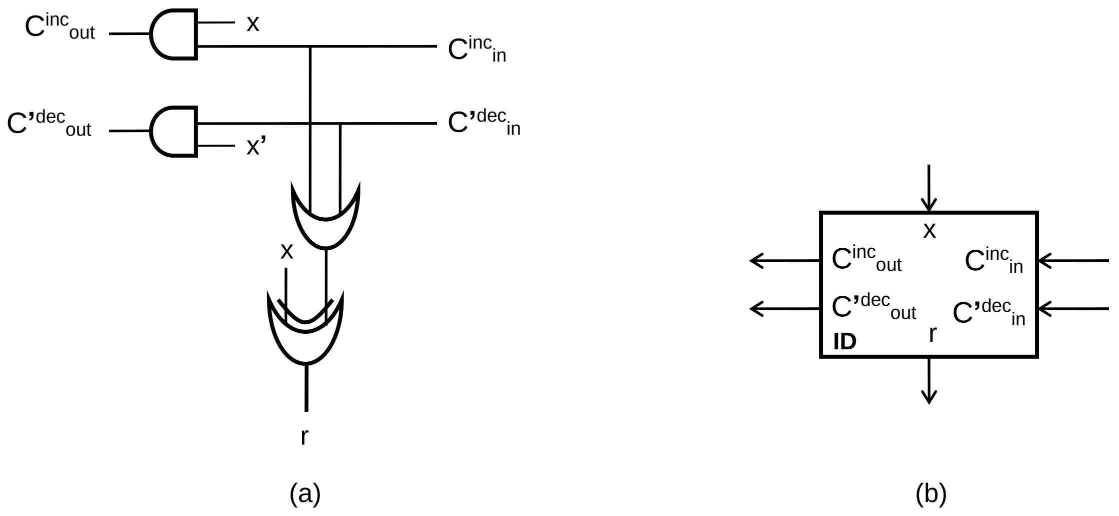


Figure 25. ID cell: (a) logic diagram, (b) logic symbol

A 4-bit Inc/Dec circuit is depicted in Figure 26. We may generalize this *repetitive* architecture to wider Inc/Dec circuits.

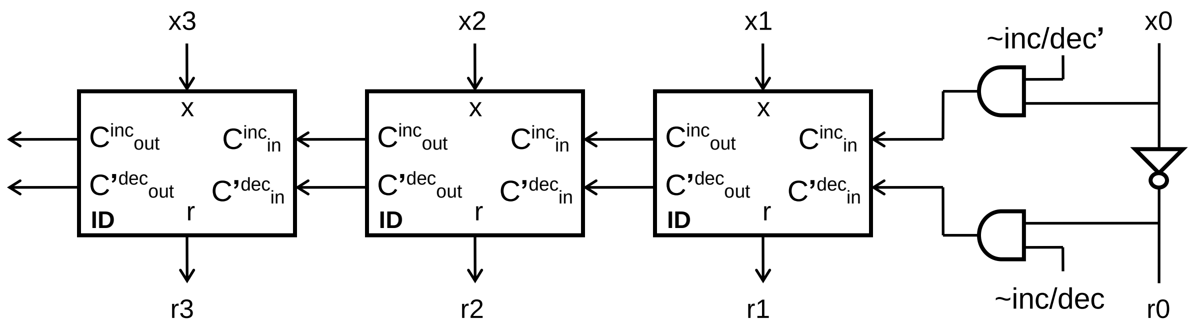


Figure 26. Four-bit Inc/Dec

Exercise: Add a *hold* input to the above Inc/Dec circuit. Remember that in the hold mode R equals X.

A different approach could also be used to reach the design of the ID cell shown in Figure 25*a*. The new approach is based on the following facts:

Special case: r0 = x'0 as shown in Equation (8).

In the Inc mode and for an arbitrary ID, the fact is:
r ≠ x if and only if all the x inputs of the less-significant IDs are at logic 1.

For example, if X = 1011, then R will be 1100:

r0 must always be the opposite of s0. Since s0 = 1, r0 becomes 0.

r1 ≠ x1, because x0 (the only less significant x input) equals 1. Since, x1 = 1, r1 becomes 0.

r2 ≠ x2, because x0 and x1 (all the less significant x inputs) equal 1. Since, x2 = 0, r2 becomes 1.

r3 = x3, because x2 (one of the less significant x inputs) is 0.

In the Dec mode and for an arbitrary ID, the fact is:
r ≠ x if and only if all the x inputs of the less significant IDs are at logic 0.

For example, if X = 1100, then R will be 1011:

r0 must always be the opposite of s0. Since s0 = 0, r0 becomes 1.

r1 ≠ x1, because x0 (the only less significant x input) equals 0. Since, x1 = 0, r1 becomes 1.

r2 ≠ x2, because x0 and x1 (all the less significant x inputs) equal 0. Since, x2 = 1, r2 becomes 0.

r3 = x3, because x2 (one of the less significant x inputs) is 1.

You can see more examples of these facts in the consecutive outputs of a 4-bit incrementer/decrementer:

Consecutive outputs for a 4-bit incrementer:
… 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 …

Consecutive outputs for a 4-bit decrementer:
… 0000, 1111, 1110, 1101, 1100, 1011, 1010, 1001, 1000, 0111, 0110, 0101, 0100, 0011, 0010, 0001, …

Fortunately, we already have two key signals for an arbitrary ID, call it cell number n:
C^{inc}_{in-n} generated in Equation (22) is pulled up if and only if all the less significant x inputs are at logic 1 provided that the Inc/Dec circuit is in the Inc mode.

C'^{dec}_{in-n} generated in Equation (23) is pulled up if and only if all the less significant x inputs are at logic 0 provided that the Inc/Dec circuit is in the Dec mode.

According to the above facts and in an arbitrary ID, a logic 1 at either of the C^{inc}_{in} or C'^{dec}_{in} inputs must make the r output equal to the opposite of the x input, otherwise, r must equal x. This behavior has been implemented using an OR gate and a programmable inverter (an XOR gate) in Figure 25*a*.