Chapter 5

Spring 2010 Edition

Binary Number Systems and Binary Arithmetic

A *SUMMARY* of what you learned in Chapter 4:

How to draw K-maps.

Physical adjacency and logical adjacency in K-maps.

How to combine adjacent 1-cells to obtain onset E cells.

How to identify prime implicants and represent them as p-terms.

All p-terms in a minimal SOP must be prime implicants.

Phase one of logic minimization: obtain the set of all prime implicants.

Phase two of logic minimization: choose a minimal subset of the above set so that every single 1-cell will be covered by at least one prime implicant.

How to obtain essential prime implicants.

All essential prime implicants of the function to be minimized have to be included in the final minimal SOP.

All concepts mentioned above can be extended to 0-cells as well to eventually reach minimal POSs.

Incompletely specified functions and how to minimize them.

Introduction

The following is an outline of the manual and systematic design methodology of medium-size digital circuits, which was presented in the first four chapters:

- Begin with the problem description, usually in natural language, and translate it into a truth table.
- Translate the truth table into a K-map.
- Extract a minimal logic expression, and derive a logic circuit.
- Design a layout
- Wire up the physical circuit.
- Do verification/troubleshooting

Remember

- The last three stages were only practiced in the lab experiments.
- In the verification/troubleshooting stage we may backtrack to any stage in order to locate and fix possible errors.
- We also have switching algebra as a mathematical and powerful tool to utilize on different occasions. For example, if the problem description is in the form of a logic expression, we may need this tool first to manipulate the expression properly.

In this chapter binary number systems and binary arithmetic are introduced, and then one type of adder/subtractor is designed using the systematic procedure developed in the previous chapters. Adders/subtractors are used in almost all digital systems. And finally two more codes, namely Gray and ASCII, are presented.

Unsigned Numbers

In Chapter 1 the unsigned binary number system was introduced. You learned that, similar to our daily used decimal system, the binary number system is also a *positional* number system because each digit has a weight determined by its *position* in the number. For example

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

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. As you know, the rightmost bit and the leftmost bit are called the least significant bit (LSB) and the most significant bit (MSB), respectively. The weighed 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, as shown below for 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 (using the successive divisions-by-2 algorithm) are shown in Figure 1. The remainder from each step has been circled and then the corresponding quotient is shown next to the remainder. In this algorithm, the quotient from each step becomes the dividend for the following 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, $(235)_{ten} = (11101011)_{two}$. ◊

$235 \div 2 = (1)\ 117 \div 2 = (1)\ 58 \div 2 = (0)\ 29 \div 2 = (1)\ 14 \div 2 = (0)\ 7 \div 2 = (1)\ 3 \div 2 = (1)\ 1 \div 2 = (1)\ 0$

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

You have probably noticed how error-prone it is for human beings to work with long bit-patterns. To significantly relax this problem, radix 8 (octal) or radix 16 (hexadecimal) numbers are usually used as two shorthand representations for binary numbers, and with convenient conversion methods. What we need to convert a binary number into hexadecimal is, starting from the *righthand* side**,** to partition the binary number into groups of 4, and then convert each group to the equivalent hexadecimal digit. For binary-to-octal conversion we use the same procedure but now we split the number into groups of 3, and then convert each group to the equivalent octal digit. Octal digits are 0, 1, 2 … 7.

Example 2. Convert N = **10** 111 101 110 001 010 to octal.

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

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

In addition to the ten traditional digits, 0-9, the following six digits or symbols are also used in the hexadecimal system in order to reach 16 different digits required in this system.

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

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

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

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

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

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

Example 5. Convert hexadecimal N = 2 C 0 F D 4 B 8 E to binary.

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

Binary Addition

There are different algorithms to perform binary addition. In this section we study the traditional *paper-and-pencil* method or *ripple-carry* addition that we use in our daily math (but with decimal numbers).

Example 6. Add in the 4-bit system: 1001 + 0011. Note: in the 4-bit system each operand is 4 bits wide.

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

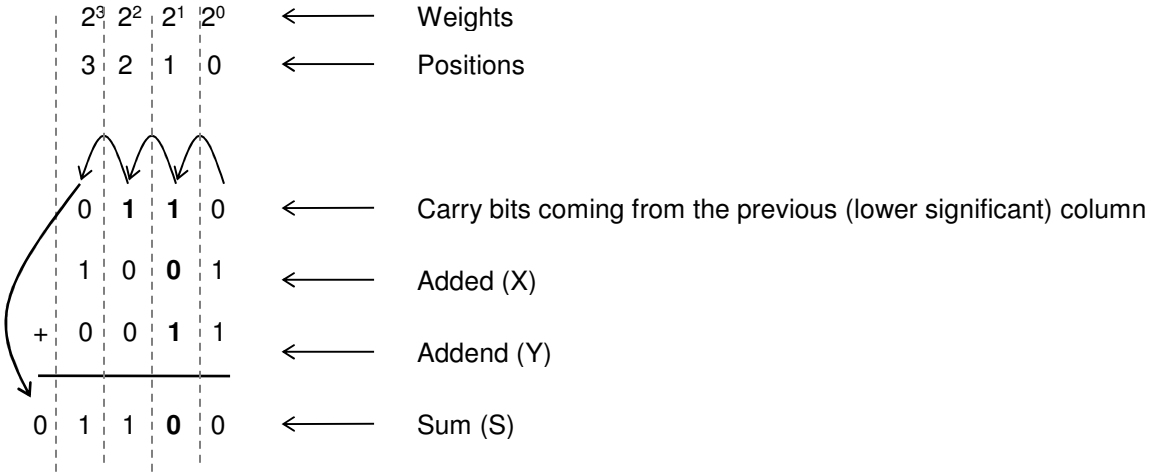


Figure 2. Paper-and-pencil method for 1001 + 0011 in Example 6

The same procedure is performed for each column: starting with the rightmost or least significant column, three bits (two bits belonging to the two operands and one carry-in bit (or c_{in} for short) from the preceding lower significant column) are added, and then two output bits are *generated*: 1) a *sum* bit (or s for short) with the same weight as the corresponding column (the *generating* 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 the weight of the generating column; therefore c_{out} ripples to the following column (the *consuming* column) with a

matching weight, and becomes c_{in} for this column, as shown in Figure 2. Since there is no column before the least significant column, the carry bit into this column is 0. As an example, consider the second column from the right, which receives a c_{in} of 1 from the least significant column, and then adds this bit and the two operand bits, namely a 0 (from the added) and a 1 (from the addend) together, and generates an s of 0 and a c_{out} of 1, as shown in Figure 2.

The addition result 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, notice 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 addition result in the n-bit system:

$(2^n - 1) + (2^n - 1) = 2 \times (2^n) - 2 = 2^{n+1} - 2$, which is the largest (n+1)-bit number minus 1.

In the result of an n-bit addition, the $(n+1)^{th}$ bit is called *final carry bit* (or *carry bit* for short). The carry bit is in fact c_{out} from the most significant column of the addition. A carry bit could be either a 0 or a 1. If carry bit is 1, we say a *final carry* (or *carry* for short) has been generated; similarly, if carry bit is 0, we say no carry has been generated. For example, in the addition shown in Figure 2 the carry bit is 0; so, we say no carry has been generated in this addition. A similar distinction exists between *borrow bit* and *borrow* to be used in binary subtractions introduced in a later section.

Example 7. Add in the 4-bit system: 1011 + 1001

The paper-and-pencil addition for this example is shown in Figure 3. Here the carry bit is 1; in other words, a final carry is produced for the whole addition signifying that the result is too large to fit in the designated number of bits which is 4 in this example; i.e., the result is out of range.

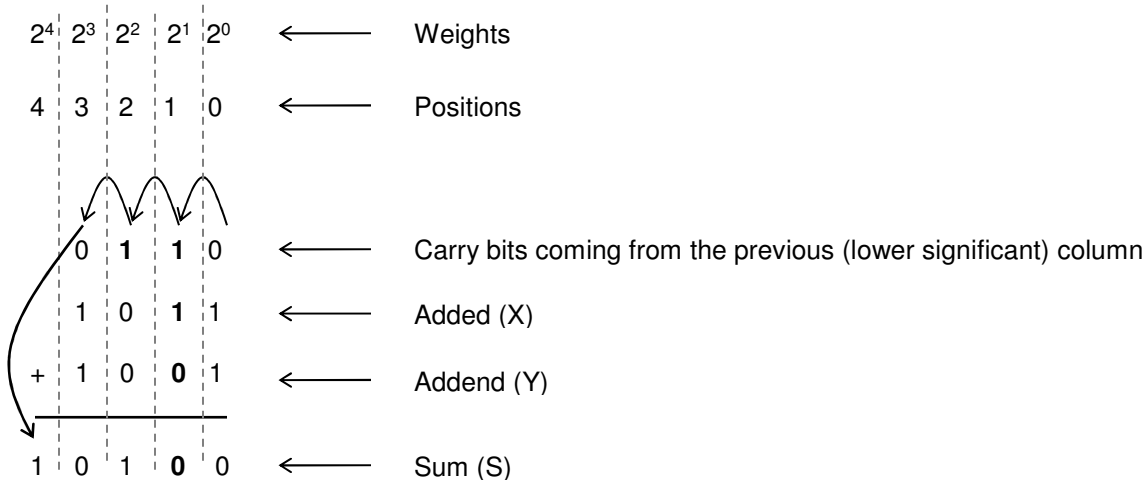


Figure 3. Paper-and-pencil method for 1011 + 1001 in Example 7

Example 8. Add in the 4-bit system: 1001 + 101

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

101 = 0101

Now addition 1001 + 0101 can be carried out as usual; the result is 1110 with no final carry.

In general, we may append as many 0s as we need to the left of an unsigned number. This is called *zero extension*.

Adder Design

As demonstrated in the above examples, a binary addition is made up of some single-column additions, each of which can be translated into an eight-row truth table with 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 logic block with three inputs and two outputs is called a *full adder* (or FA for short), and is shown in Figure 4*c*. A FA is in charge of performing one-column addition as described in the above examples. Now, n FAs can be cascaded to reach an n-bit adder, as shown in Figure 5*a* for n = 4. Figure 5*b* shows a logic symbol for this adder.

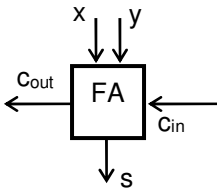
Row	x	y	c_{in}	c_{out}	s
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

(a)

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

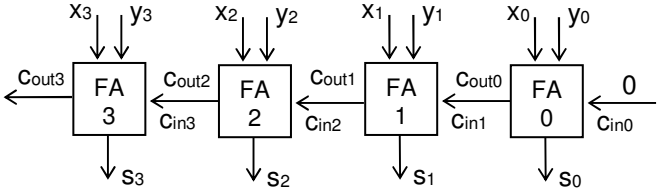
$s(x, y, c_{in}) = \Sigma\ (1, 2, 4, 7)$

(b)

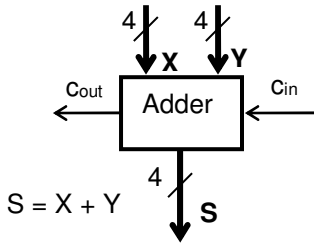


(c)

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



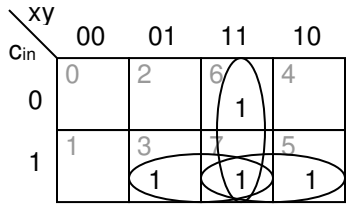
(a)



(b)

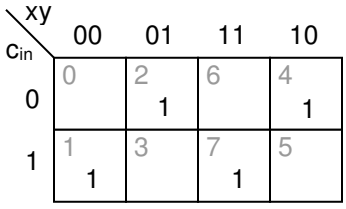
Figure 5. Four-bit ripple adder: (a) logic diagram, (b) symbol

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



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

(a)



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

Both functions should be familiar to us: c_{out} is the three-bit majority function which can be simplified as shown in Figure 6*a*. On the other hand, s cannot be simplified (see Figure 6*b*); it goes up whenever an

odd number of inputs are high and vice versa. And this is similar to the "hallway-light and 3-office problem" considered in Lab Exercise 2. Remember from Chapter 2 that the output of an n-input XOR gate goes up if the gate has an odd number of '1' inputs; otherwise, the output is pulled down. Therefore s can also be realized with a 3-input XOR gate:

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

Binary Subtraction: The *paper-and-pencil* (decimal) subtraction is the traditional method that we manually use in our daily math. In the following two examples binary subtraction is closely analyzed based on this algorithm:

Example 9. Subtract in the 4-bit unsigned system: 1001 – 10

The second operand is 2 bits wide. So, first we make it 4 bits wide (0010) by zero extension, similar to what we did in Example 8, and then apply the paper-and-pencil subtraction algorithm as shown in Figure 7. The same procedure is carried out for each individual column: the subtrahend bit is added to the borrow bit (borrow-in or b_{in} for short) coming from the previous column and then this sum is taken away from the minuend bit as summarized below:

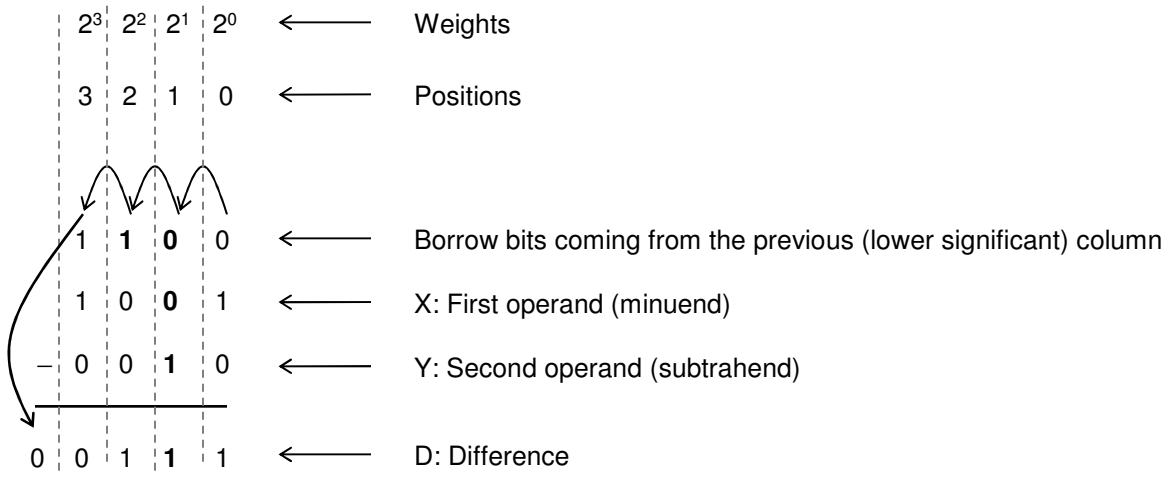


Figure 7. Paper-and-pencil method for 1001 – 0011 in Example 9

$$\text{difference bit} = \text{minuend bit} - (\text{subtrahend bit} + b_{in}) \quad (2)$$

Consider the first column in Figure 7 in which the minuend bit is 1 and the subtrahend bit is 0. Also the borrow bit arriving at this column is 0, as this is the least significant column. Therefore,

$$\text{difference bit} = 1 - (0 + 0) = 1$$

If the subtraction in Equation (2) is not doable, in other words, if the minuend bit is less than the sum of subtrahend bit and b_{in}, then the corresponding column will generate a borrow (out) bit (or b_{out} for short) of 1, signifying that this column has to borrow a 1 from the following more significant column, as shown by Equation (3):

$$\text{difference bit} = 2 + \text{minuend bit} - (\text{subtrahend bit} + b_{in}) \quad (3)$$

The 1 borrowed from the next more-significant column appears as a 2 in the current column as shown by Equation (3), because the weight of the next more-significant column is twice as much as the weight of the current column. Now, the next more-significant column receives a b_{in} of 1 from the current column. Consider the second least-significant column in Figure 7 in which the minuend bit, subtrahend bit and b_{in} are 0, 1 and 0, respectively. But now

minuend bit < (subtrahend bit + b_{in})

Therefore, for this column the difference and b_{out} are produced as follows:

difference bit = 2 + 0 – (1 + 0) = 1 and b_{out} = 1 ◊

In this example the final b_{out} generated by the most significant column is 0, which indicates that the minuend (1st operand) is not less than the subtrahend (2nd operand); in other words the subtraction is doable.

Example 10. Subtract in the 4-bit unsigned system: 0001 – 0111

The paper-and-pencil subtraction for this example is shown in Figure 8. Here, b_{out} from the most significant column is 1; in other words a final borrow is produced for the whole subtraction signifying that the minuend (1st operand) is less than the subtrahend (2nd operand), or the subtraction result is negative. However, the number system that we are considering here is unsigned, in which negative numbers cannot be represented.

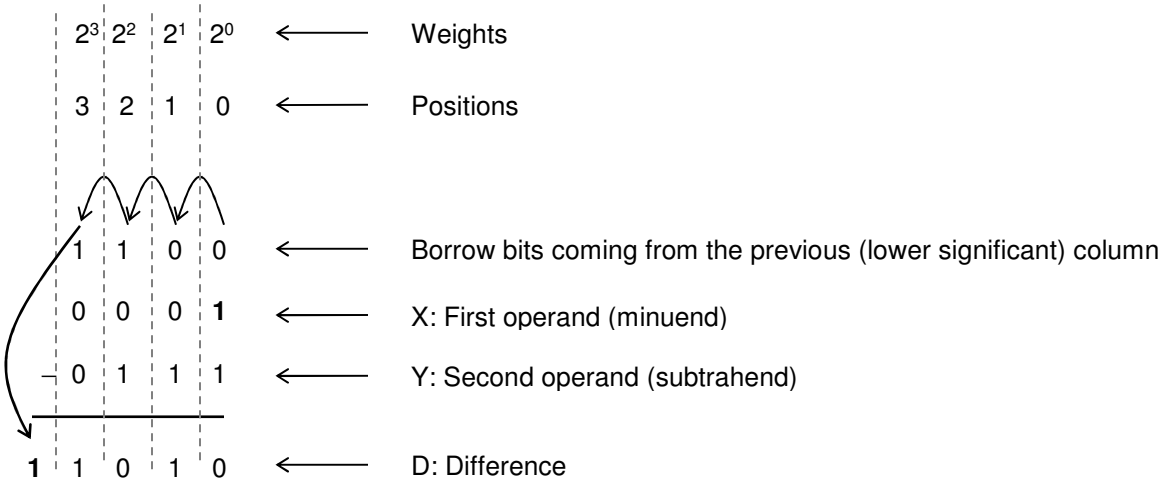


Figure 8. Paper-and-pencil method for 0001 – 0111 in Example 10

Subtractor Design

The algorithm mentioned above for each individual column can be translated into a truth table with three inputs (x, y and b_{in}) and two outputs (d and b_{out}), as shown in Figure 9*a*. Figure 9*b* shows the minterm lists of d and b_{out}. The logic block of this truth table is called a *full subtractor* or FS for short, which has 3

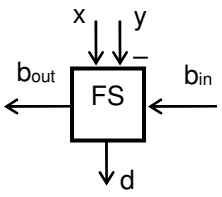
Row	x	y	b_{in}	b_{out}	d
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	1	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	0	0
6	1	1	0	0	0
7	1	1	1	1	1

(a)

$d(x\ y\ b_{in}) = \Sigma\ (1, 2, 4, 7)$

$b_{out}(x\ y\ b_{in}) = \Sigma\ (1, 2, 3, 7)$

(b)



(c)

Figure 9. Full subtractor: (a) truth table, (b) on-set minterm lists, (c) logic symbol

inputs and two outputs as shown in Figure 9*c*. (y, the second operand, has been signified with a minus sign.) So, a FS is in charge of performing one-column subtraction as described in the above examples. Now, n FSs can be cascaded to reach an n-bit subtractor, as shown in Figure 10*a* for n = 4. Figure 10*b* shows a logic symbol for this subtractor.

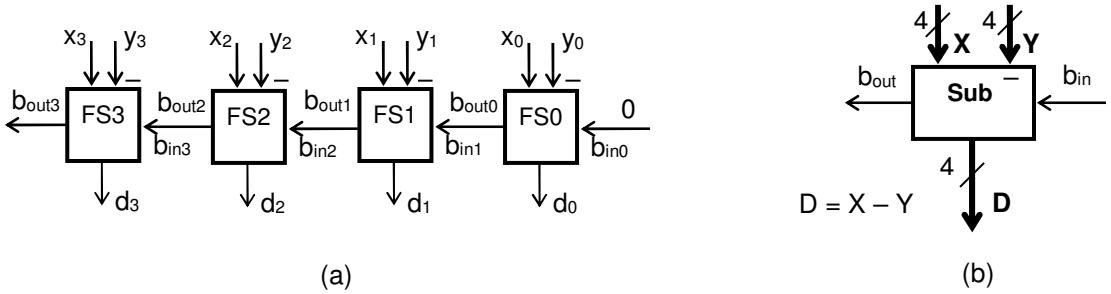


Figure 10. Four-bit ripple subtractor: (a) logic diagram, (b) symbol

Addition-based Subtraction

In order to save hardware, an adder is usually converted into a subtractor with some minor modifications, such that the resulting hardware could be used for addition or subtraction (but not simultaneously of course!); in other words, subtraction is now carried out by addition as elaborated on in this section. The corresponding hardware will be developed in the next section.

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

$$X - Y = X - Y + 2^n - 2^n = X + (2^n - Y) - 2^n$$

A simple manipulation yields

$$X + (2^n - Y) = X - Y + 2^n \qquad (4)$$

According to Equation (4) the result of addition $X + (2^n - Y)$ has the required subtraction result (i.e., $X - Y$), plus a redundant term, 2^n. Therefore, if the following two problems are handled properly, then the addition on the left of (4) has in fact replaced the required subtraction.

1- The term $(2^n - Y)$ on the left of Equation (4) still requires a subtraction, but we don't want to do subtraction.
2- A redundant term, 2^n, exists on the right side of (4), so we have to get rid of that.

It can easily be shown that the result of subtraction $2^n - Y$ is obtained using the following simple algorithm, which does not require subtraction:

$2^n - Y$ = (bit-wise complement of Y) + 1 (5)

This means that instead of subtracting Y form 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 to obtain the 2's complement of Y may be reworded as follows:

Starting from the righthand side of Y, copy all the bits until and including the first 1, and then complement the remaining bits. (5')

Note: When using (5'), subtrahend = 0 is a special case to be addressed in Example 14.

Example 11. Obtain $2^4 - 1001$

According to (5): $2^4 - 1001 = 0110 + 1 = 0111$

According to (5'): $2^4 - 1001 = 0111$

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

X + (bit-wise complement of Y) + 1 = $X - Y + 2^n$ (6)

Now let's approach the second issue: "how can we get rid of the redundant term, 2^n, on the right of Equation (6)?" 2^n is a 1 followed by n 0s, and this makes it straightforward to locate and ignore this redundant term, hence solve the second problem. See the following example for details.

Example 12. In the 4-bit system ($n = 4$) use addition to do the subtraction $X - Y = 1010 - 0101$.

Instead of the subtraction $X - Y = 1010 - 0101$, we carry out the corresponding addition based on Equation (6):

X + (bit-wise complement of Y) + 1 = 1010 + 1010 + 1 = **1** 0101 (7)

According to Equation (6) the addition result, **1** 0101, is the result of the intended subtraction (i.e. $1010 - 0101$), plus the redundant term, 2^4, which is a '1' followed by four 0s: **1**0000. In other words, the fifth bit of the addition result, i.e., the bold **1** on the right side of Equation (7) is the redundant term. This **1** is in fact the final carry generated by the addition on the left side of Equation (6) which can easily be identified and ignored without an explicit subtraction as we saw in this example. ◊

Let's see what is signified if a final carry is not generated when the addition on the left of Equation (6) is performed. In this equation if $X < Y$ or $X - Y < 0$ (i.e., the subtraction $X - Y$ is not doable, hence a final borrow occurs in this subtraction), then $X - Y + 2^n$ becomes less than 2^n; and this means that no final carry is generated in the addition on the left side of Equation (6).

In summary, if in the addition X + (bit-wise complement of Y) + 1 a final carry occurs, then the corresponding subtraction ($X - Y$) is successful (i.e., Y is NOT greater than X). Ignore the final carry; what is left is the correct subtraction result. However, no final carry in this addition indicates that Y is greater than X, i.e., a final borrow occurs in the corresponding subtraction, $X - Y$. In other words, the subtraction result is negative, hence not representable in an unsigned number system, as shown in the following example.

Example 13. Use addition to do the subtraction $X - Y = 1010 - 1101$

Instead of the subtraction $1010 - 1101$, we perform an addition according to Equation (6):

X + (bit-wise complement of Y) + 1 = 1010 + 0010 + 1 = **0** 1101

No final carry has been produced in this addition; therefore, a final borrow occurs in the corresponding subtraction, indicating that $X < Y$. ◊

There is a subtle point in addition-based subtraction when the subtrahend is 0, and Algorithm (5') is used to obtain the two's complement of 0, as shown in the following example:

Example 14. Use addition to do the subtraction $X - Y = 0010 - 0000$

There is no 1 in Y. So, Algorithm (5') will generate 0000 as the 2's complement of Y. When this 0000 is added to 0010 (the minuend), 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 a borrow has been generated for the corresponding subtraction. To avoid this problem we use Algorithm (5) which does generate a carry if $Y = 0$. This algorithm is exactly what we take into consideration in developing hardware for subtraction in the next section.

(5) (bit-wise complement of Y) + 1 = 1111 + 1 = **1** 0000 (carry bit generated in this stage) ◊

Here is a summary to perform $X - Y$, n-bit addition-based unsigned subtraction:

1- If the operands are not already n bits wide, do zero extension to make them n bits wide.

2- Perform X + (bit-wise complement of Y) + 1

(You may also use Algorithm (5') to obtain (bit-wise complement of Y) + 1 if $Y \neq 0$)

3- The result of step 2 is $X - Y + 2^n$. (see Equation 6)

4- No final carry in step 2 indicates a *final borrow*, or a greater subtrahend than the minuend; hence an impossible subtraction.

5- But if a final carry is produced in step 2, **ignore** the carry; the n LSBs of the addition result obtained in step 2 comprise the correct subtraction result.

Addition-based Subtractor Design

The addition-based subtraction algorithm described by Equation 6 has directly been translated into hardware in Figure 11 for n = 4, where n is the subtractor size. As shown in this figure, a four-bit ripple carry adder receives the four-bit minuend $X = x_3x_2x_1x_0$, and the bit-wise complement of the 4-bit subtrahend $Y' = y'_3y'_2y'_1y'_0$, and produces the four-bit output $D = d_3d_2d_1d_0$. Additionally, a '1' is injected to the C_{in} of the least significant FA. Therefore, this 4-bit adder performs the following addition:

D = X + (bit-wise complement of Y) + 1.

And this is exactly the left-side expression of Equation (6).

Consider c_{out3} (carry-out bit from the most significant FA) in Figure 11. As explained on page 9, in the addition-based subtraction, c_{out3}= 1 implies no final borrow for the corresponding subtraction. And similarly, c_{out3}= 0 indicates that a final borrow has been generated. That is why c_{out3} (in Figure 11) is called *~borrow-out* or $\sim b_{out}$ for short. In other words, c_{out3} is now an active-low signal representing the (final) borrow-out bit of this subtractor. ◊

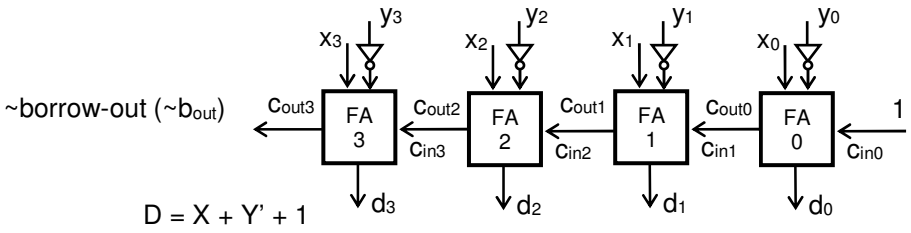


Figure 11. Adder-based subtractor

We may combine the four-bit adder shown in Figure 5a with the four-bit subtractor developed in Figure 11 to obtain an adder/subtractor unit depicted in Figure 12*a*, in which 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 a logic symbol for this unit shown in Figure 12*b*.) This arithmetic unit has two operation modes, namely add and subtract, and a mode-select control signal 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 12*a*, in the path of the second operand, Y, there are four programmable inverters (i.e., four XOR gates) which are controlled by the ~add/sub line. This control line also drives 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 12*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; so that the second operand of each FA becomes the complement of the corresponding bit in subtrahend Y, as illustrated in Figure 11. Additionally, a '1' is applied to the c_{in} input of the least significant FA because now ~add/sub = 1; and these are exactly what are required in order to implement the addition-based subtraction algorithm modeled by Equation (6). c_{out3} from FA3 plays two different roles in these two operation modes: in the

add mode c_{out3} is the final carry bit in the corresponding four-bit addition, i.e., if $c_{out3} = 1$, then a final carry has occurred, otherwise there is no final carry. However, c_{out3} has a different interpretation in the subtract mode: now if $c_{out3} = 0$, then a final borrow has occurred in the corresponding subtraction, otherwise there is no final borrow. And that is why we call this output $c_{out}/\sim b_{out}$, 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.

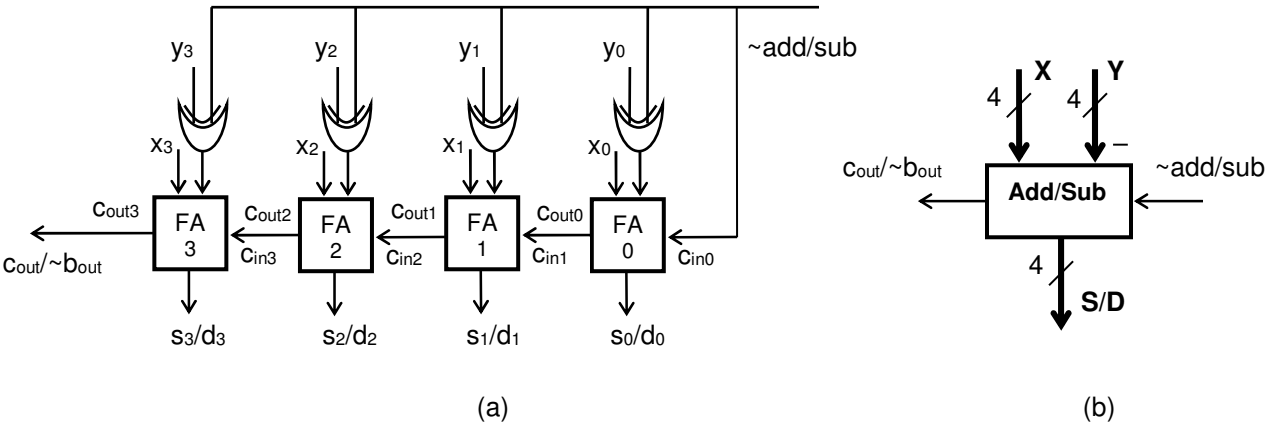


Figure 12. Adder/subtractor unit: (a) logic circuit, (b) symbol

Half Adders and Half Subtractors

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 input 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 13*a*. Figure 13*b* shows the minterm lists of c_{out} and s, and Figure 13*c* illustrates a logic symbol for a HA. HAs are used less frequently than FAs. As an example, a HA can replace FA0 in Figure 5*a* because 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.

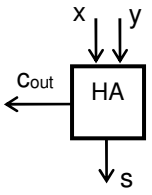
Row	x	y	c_{out}	s
0	0	0	0	0
1	0	1	0	1
2	1	0	0	1
3	1	1	1	0

(a)

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

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

(b)



(c)

Figure 13. Half adder (a) truth table, (b) symbol

A half subtractor (or HS for short) can be defined in a similar way.

Signed Number Systems

The number system that we have considered so far is unsigned, in which the smallest possible number to represent is 0. In this section three different signed number systems, including the 2's complement system, are introduced to represent negative numbers as well as positive numbers, 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 the *sign-and-magnitude* system 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 in fact 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 (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: $Nmax = 2^{n-1} - 1$

In this system number 0 has two different representations, namely 1000, and 0000 for n = 4; that is why B = P – 1.

Example 15. Obtain P, B, R, Nmin and Nmax for the 4-bit S&M 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

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 in the 1's complement system is the bit-wise complement or *1's complement* of the corresponding positive number, while the MSB is still the sign bit. For example, -6 = 1001 in the 4-bit 1's complement system 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; that is why B = P – 1.

Example 16. 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 (5) or (5') mentioned in the previous section, while the MSB remains the sign bit. Algorithms (5) and (5') have been reworded and shown here as (8) and (8'), respectively, to be used in the 2's complement system:

– N (additive inverse of N) = (bit-wise complement of N) + 1 (8)

Or

To obtain the additive inverse of N, starting with the LSB of N, copy all bits until and including the first 1, and then complement all of the remaining bits. (8')

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

Example 17. Negate N = 0110 (+6) in the 4-bit 2's complement system:

Use (8): -N = 1001 + 1 = 1010 (-6)

Use (8'): -N = 1010 (-6)

Figure 14 shows all different 4-bit patterns and their interpretations in the 2's complement system.

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

In the n-bit 2'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: $B = 2^n$

Range of numbers: $R = [-2^{n-1} : +(2^{n-1} - 1)]$

Smallest number: $Nmin = -2^{n-1}$

Largest number: $Nmax = 2^{n-1} - 1$

Example 18. 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 systems, number 0 has only one representation in which all bits are 0s; that is why now B = P. The single bit pattern saved in this way (i.e., 1000 in the 4-bit system) now becomes the most negative number (i.e., -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 this system. For example, for n = 4, if 1000 (-8) was negated, then the result would be 1000 which is still -8! This means that -8 cannot be negated because the result, -(-8) = +8, is out of range (see Example 18). In other words an *overflow* occurs when -8 is negated in the 4-bit two's complement system. We will talk more about overflow shortly in this chapter.

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

Example 19. 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 20. Represent decimal -50 in the 2's complement format. 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 sign of the result is the same 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. And finally the sign of the largest magnitude has to be assigned to the result. This procedure is significantly simplified in the 1's complement system. However, in this system signed and unsigned additions are still different from each other. Additionally (and as mentioned before), this system has two different representations for number zero (which are, for example, 0000 and 1111 in the 4-bit 1's complement system). These two problems are solved in the 2's complement system at the cost of some asymmetry (see Example 18) and also more difficult negation (see algorithm (8) or (8')). 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 21. Add in the four-bit 2's complement system: 0011 (= +3) + 0100 (= +4)

As you know, 2's complement additions can still be carried out using the paper-and-pencil addition algorithm. We apply this algorithm to the above operands to obtain 0111, which is a valid result in the 4-bit 2's complement system.

sign bit	**0** 011	= + 011 = +3
+	**0** 100	= + 100 = +4
	0 111	= + 111 = +7

Example 22. Add in the four-bit 2's complement system: 0011 (= +3) + 1100 (= -4)

In this addition (with no final carry) the 4-bit result is 1111 which is a valid result (-1) in the 4-bit 2's complement system.

```
sign
bit
  0 011    = + 011 = +3
+ 1 100    =  - 100 = -4
  ---------------------
  1 111    =  - 001 = -1
```

Example 23. Add in the four-bit 2's complement system: 1101 (= -3) + 1100 (= -4)

In this example the addition results in a final carry. But **it can be proved that it is not part of the addition result and should be removed**; so the 4-bit result of this addition is 1001, which is the correct result (-7) in the 4-bit 2's complement system. Notice that the result is as wide as each of the operands. Moreover, the position of the sign bit is always fixed: it is the MSB of the result.

```
sign
bit
  1 101    = - 011 = -3
+ 1 100    = - 100 = -4
  ---------------------
(1) 1 001  =  - 111 = -7
 ↳ Carry out, ignore it
```

Example 24. Add in the four-bit 2's complement system: 1101 (= -3) + 0100 (= +4)

In this addition a final carry is again produced, but we ignore it as we did in Example 23. The 4-bit result is 0001 which is a valid result (+1) in the 4-bit 2's complement system.

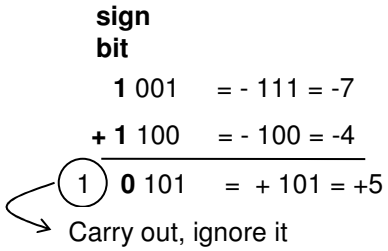
```
sign
bit
  1 101    =  - 011 = -3
+ 0 100    = + 100 = +4
  ---------------------
(1) 0 001  =  + 001 = +1
 ↳ Carry out, ignore it.
```

Example 25. Add in the four-bit 2's complement system: 0110 (= +6) + 0101 (= +5)

```
sign
bit
  0 110    = + 110 = +6
+ 0 101    = + 101 = +5
  ---------------------
  1 011    =  - 101 = -5
```

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 have been added but the result is a negative number, -5! What is wrong with this addition? The answer is since the correct result of this addition is too large to fit in the designated number of bits, 4, the (4-bit) result looks like a negative number. More specifically, 6 + 5 = +11, but from Example 18 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 by four bits, and this means that an *overflow* has occurred.

Example 26. Add in the four-bit 2's complement system: 1001 (= -7) + 1100 (= -4)



In this addition 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, have been added but the result looks like a **positive** number, +5! The reason is the absolute value of the correct result of this addition is again too large to fit in the designated number of bits, 4. More specifically, (-7) + (-4) = -11; but the most negative number representable in the four-bit 2's complement system is -8 (see Example 18); therefore the correct result cannot be represented in four bits, and this again means that an overflow has happened.

Definition: If the result of an operation does not fit into the designated number of bits, then an *overflow* has occurred.

In other words, if the result of an operation falls outside the range of the representable numbers in the corresponding number system, then an *overflow* has happened.

In a four-bit unsigned addition if a final carry occurs, then the addition result becomes 5 bits wide, hence falls out of range, i.e., in an unsigned addition a final carry means an 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 in the 2's complement system; so that we may come across any of the four possible combinations of presence/absence of these two events: neither carry nor overflow (see Example 21 and Example 22), carry, but no overflow, (see Example 23 and Example 24), no carry but overflow, (see Example 25), and finally both carry and overflow (see Example 26). In general, by an overflow we mean a 2's complement overflow unless otherwise specified.

The addition result of two different-sign operands will definitely not go beyond either of the two operands, and since the operands are valid numbers in the 2's complement system, the addition result would also be a valid number in this system. In other words, the addition of two different-sign operands will never produce an overflow; i.e., in a two's complement addition an overflow might happen only if the two operands have the same sign.

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

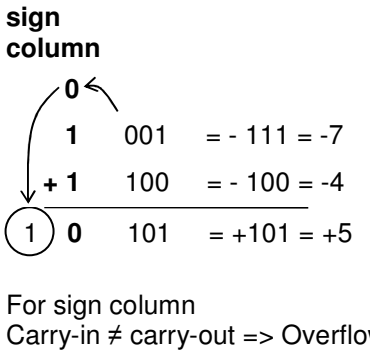
Lemma 1. In 2's complement addition an overflow occurs if and only if 1) the two operands have the same sign bit, and 2) this sign bit is different from the sign bit of the result.

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

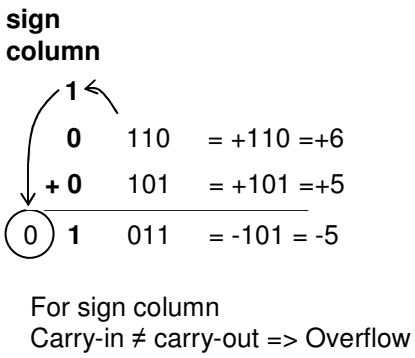
Lemma 2. In 2's complement addition an overflow occurs if and only if the carry bit arriving at the sign column is different from the carry bit leaving this column.

Example 27. Add in the 4-bit 2's complement system: 1001 (= -7) + 1100 (= -4)

The carry bits arriving at and leaving the sign column are different; therefore, an overflow occurs in this addition.

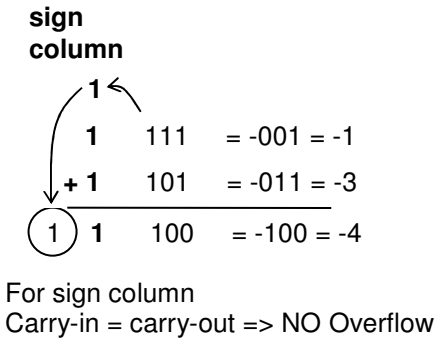


Example 28. Add in the 4-bit 2's complement system: 0110 (= +6) + 0101 (= +5)



The carry bits arriving at and leaving the sign column are again different; therefore, an overflow occurs in this addition as well.

Example 29. Add in the 4-bit 2's complement system: 1111 (= -1) + 1101 (= -3)



The carry bits arriving at and leaving the sign bit 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 an overflow occurs.

2-The addition of two positive numbers never produces a final carry, no matter whether or not an 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 8 and Example 9. The same need may also frequently arise for signed numbers; however the remedy could now be different.

Example 30. Add in the four-bit 2's complement system: 1001 (= -7) + 011 (= 3)

Since the system is four bits wide, we need to widen the second operand by one bit. 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 that 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 31. Add in the four-bit 2's complement binary system: 1011 (= -5) + 110 (= -2)

The second operand is again three bits wide, hence has to be extended by one bit. However, now this cannot be done by inserting a 0, because this 0 will convert the negative operand 110 (-3) 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 ◊

The observations in the last two examples may be summarized as follows:

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

In summary, here is the procedure to add two numbers in the n-bit 2's complement system:

- If the operands are not already n bits wide, do sign extension to make them n bits wide.
- Add the two operands using, say, the paper-and-pencil algorithm. In this addition sign bits are treated similar to other bits of the operands.
- The sign bit of the result is the n^{th} bit (and not the $(n+1)^{th}$ bit) from the right.
- Check to see whether or not an overflow has occurred, as explained above. If there is an overflow, the result is invalid.
- If there is no overflow, ignore the carry bit. The remaining n bits are the correct result of the addition.

Two's Complement Subtraction
In a signed system every subtraction may be reworded into an addition because now negative numbers are representable as well: 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 subtrahend is negated, the subtraction will be converted to an addition; therefore in the 2's complement system we may use Algorithm 8 and write:

X – Y = X + (bit-wise complement of Y) + 1 (9)

Example 32. Subtract in the 4-bit 2's complement system: 0111 (= +7) – 0011 (= +3)

Negate the second operand, and then add it to the first operand:

7 – 3 = 7 + (-3) = +4

Or

0111 – 0011 = 0111 + (-0011)

To negate 0011 apply (8) or (8')

-0011 = 1101

Therefore, 0111 – 0011 = 0111 + 1101 = **1** 0100 (= +4)

A carry has occurred in this addition. **But it can be proved that the carry is not part of the result**. Remove it; the remaining bits (0100) comprise the correct subtraction result. This is similar to what we did for the signed additions in the previous section. Also, no overflow is generated in this subtraction because no overflow occurs in the corresponding addition. There is an exception to this rule (to be elaborated on in Example 35) if the subtrahend is the most negative number and it is negated by applying 8'.

In addition-based subtractions the overflow detection method mentioned above may be reworded as follows:

Lemma 1'. In 2's complement subtraction modeled by (9), an overflow occurs if and only if 1) the two operands have different sign bits, and 2) the sign of the result is different from that of the minuend.

Example 33. Subtract in the 4-bit 2's complement system: 1100 (= -4) – 101 (= -3)

The subtrahend is 3 bits wide, so do sign extension by 1 bit: 101 = 1101

Negate the subtrahend: -1101 = 0011

Add the minuend to the negated subtrahend: 1100 + 0011 = 1111. No carry is produced.

The subtraction operands have the same sign, so according to Lemma 1' there is no overflow in this subtraction; i.e., 1111 is the correct subtraction result (-1).

Example 34. Subtract in the 4-bit 2's complement system: 1001 (= -7) – 0100 (= +4)

Both operands are already 4 bits wide; so no sign extension is required.

Negate the subtrahend: -0100 = 1100

Add the minuend to the negated subtrahend: 1001 + 1100 = 1 0101

Consider the above addition: both operands are negative but the result (0101) looks positive, so an overflow has happened in this subtraction. Or we could say since the subtraction operands have different signs and the subtraction result (0101) looks positive while the minuend (1001) is negative, according to Lemma 1' an overflow has occurred in this subtraction.

Example 35. Subtract in the 4-bit 2's complement system: 1010 (= -6) – 1000(= -8)

There is a subtle point in this example: the subtrahend, -8, is the most negative number in the 4-bit 2's complement system, hence cannot be negated, however in this and similar subtractions we may ignore this overflow!

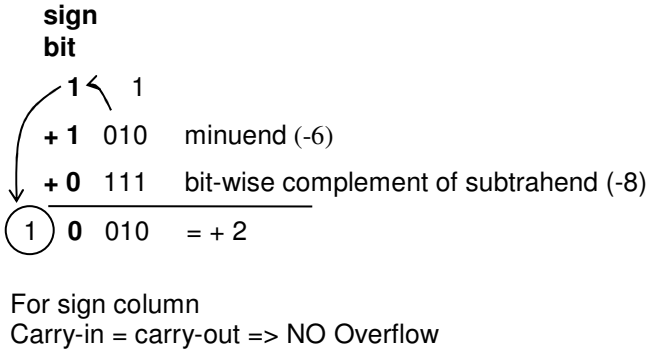
Negate the subtrahend: -1000 = 1000: overflow, but ignore it!

Add the minuend to the negated subtrahend: 1010 + 1000 = 1 0010 (10)

Since the subtraction operands have the same sign, according to Lemma 1' there is no overflow.

Ignore the carry bit; 0010 (+2) is the correct result.

You need to pay close attention to this example to interpret its result properly. The addition in (10) has seemingly produced an overflow (because the two operands look negative, while the result looks positive), although the corresponding subtraction is overflow-free. In order to resolve this discrepancy we need to perform the two additions in Equation (9) together when the subtrahend is the most negative number as shown below for the subtraction in Example 35. This concurrent addition is applicable to any subtraction.



Look at this 3-operand addition. The carry bit arriving at and the carry bit leaving the sign column are now the same, signifying no overflow for this subtraction. Lemma 3 is the extended version of Lemma 2 which states the necessary and sufficient condition for 2's complement addition or subtraction to produce an overflow:

Lemma 3. In 2's complement subtraction modeled by Equation (9), and also in 2's complement addition, an overflow occurs if and only if the carry bit arriving at the sign column is different from the carry bit leaving this column. Perform the two additions in (9) concurrently if the subtrahend is the most negative number. Equation (9) is shown here again for ease of reference:

$$X - Y = X + (\text{bit-wise complement of } Y) + 1 \qquad (9)$$

In summary, here is the procedure for n-bit subtraction in the 2's complement system:

1- If the operands are not already n bits wide, do sign extension to make them n bits wide.

2- Perform $X + (\text{bit-wise complement of } Y) + 1$

Carry out these two additions concurrently if Y is the most negative number.

3- If there is an overflow, then the subtraction result is not representable in n bits.

Note: The sign bit of the addition result is the n^{th} bit (and not the $(n+1)^{th}$ bit) from the right.

4- Otherwise, if there is no overflow, ignore the carry bit. The remaining n bits comprise the correct result of the subtraction.

Adder/Subtractor Design for Signed Numbers

As mentioned before, it can be shown that the same addition algorithm used for unsigned numbers can be used for signed numbers in the 2's complement system. Additionally, note that a signed subtraction can be converted to a signed addition by negating (2's complementing) the subtrahend as modeled by Equation (9). And this is exactly what is performed in the unsigned addition-based subtractor implemented in Figure 11. Therefore, the same unsigned arithmetic unit developed in Figure 12*a*, will properly work for signed additions/subtractions as well.

Overflow Detection

The arithmetic unit developed in Figure 12*a* has been repeated in Figure 15 for ease of reference. The necessary and sufficient condition for an overflow to occur has been specified by Lemma 3. According to this lemma and in a 4-bit adder/subtracttor, if cout3 ≠ cin3 (Figure 15), an overflow has happened for the corresponding addition or subtraction and vice versa. Therefore, overflow can be detected by an XOR gate as shown in Figure 15.

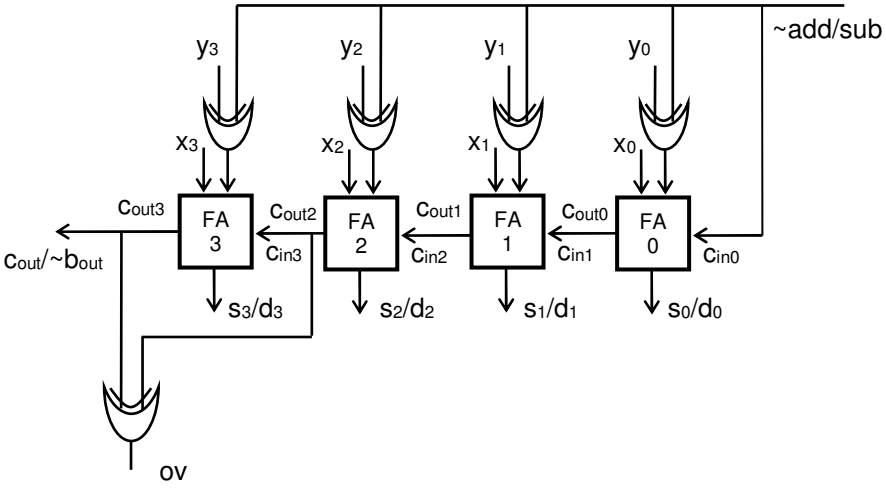


Figure 15. Overflow detection using an XOR gate

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*. For example, as we have seen in this chapter so far, we may put 0s and 1s together as radix 2 or straight binary numbers, which can mathematically be manipulated conveniently. In this section two more codes, namely Gray code and ASCII code, are introduced.

Gray Code

Two consecutive binary numbers may be different in more than one bit, such as 0111 (7) and 1000 (8), which differ in 4 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 only one bit. Gray code may be created recursively as follows:

- 0 and 1 are the two (consecutive) single-bit Gray code words. For these two code words $n = 1$, 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 the 2^n already-existing n-bit Gray code words in *reverse* order and following the already existing code words. This results in 2^{n+1} n-bit code words with two instances per code word.
- Append a 0 to the left of each old code word and append a 1 to the left of each new code word. This results in 2^{n+1} $(n+1)$-bit consecutive Gray code words.

Example 36. Figure 16*a* shows two-bit Gray code words. In Figure 16*b* these code words have been repeated in reverse order following the 2-bit Gray code words. Now the first four code words (the old ones) each receive a leading 0, and the last four code words (the new ones) each receive a leading 1 to eventually reach a 3-bit Gray code as shown in Figure 16*c*.

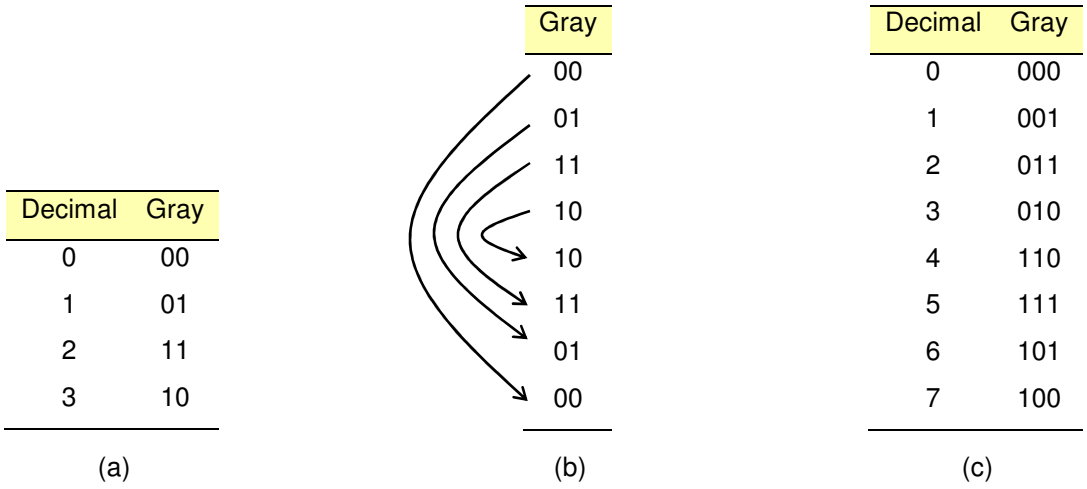


Figure 16. Recursive generation of a 3-bit Gray code: (a) 2-bit Gray code, (b) 2-bit code followed by reversed-order 2-bit code, (c) 3-bit Gray code

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.

Example 37. 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).

Character Codes

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 basic building blocks available to us to represent whatever we need in the digital world. This means that in order to digitally represent characters as well, we 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 character code 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 some 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 hexadecimal and 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 hexadecimal digits (7, 2, and 2) come from the top row, and the most-significant hexadecimal 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