Chapter 7

Digital Arithmetic and Arithmetic Circuits

7.1 Digital Arithmetic
Signed/Unsigned Binary Numbers

- Signed Binary Number:
  - A binary number of fixed length whose sign (+/–) is represented by one bit (usually MSB) and its magnitude by the remaining bits.

    (最高位元為符號位元)

- Unsigned Binary Number:
  - A binary number of fixed length whose sign is not specified by a bit. All bits are magnitude and the sign is assumed +. **(沒有符號位元)**

Unsigned Binary Arithmetic

- **Sum:**
  - Result of an Addition Operation of two (or more) binary numbers.
- **Carry:**
  - A digit (or bit) that is carried over to the next most significant bit during an *n*-Bit addition operation.
- The carry bit is a 1 if the result was too large to be expressed in *n* bits.

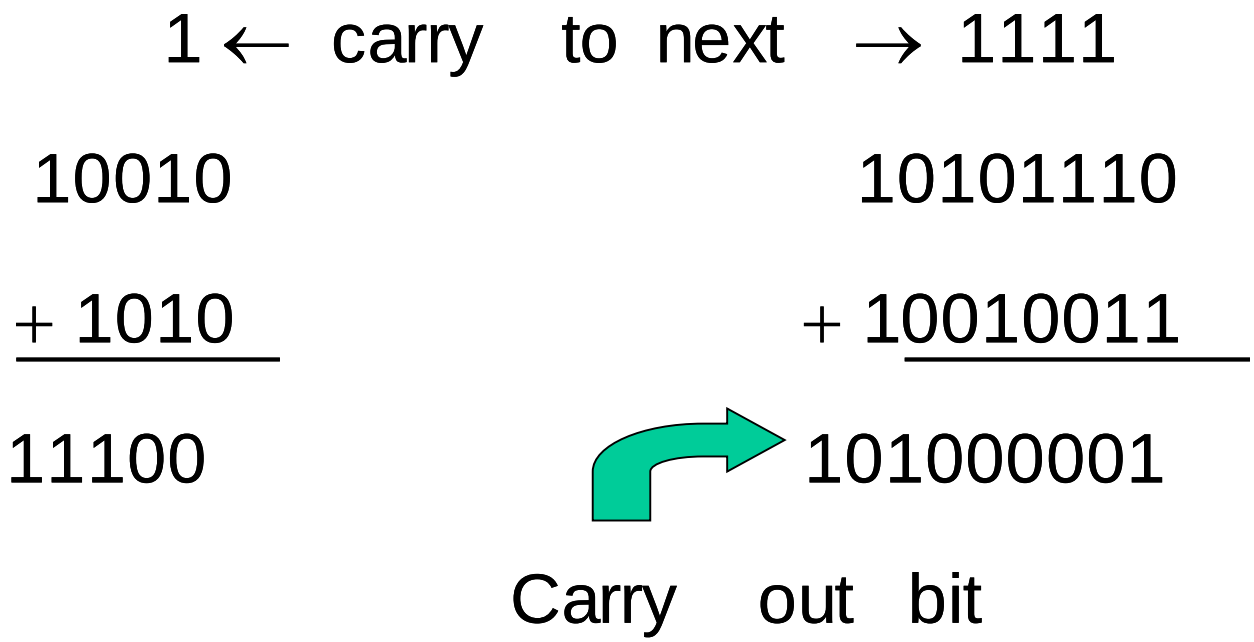
Basic Rules (Unsigned)

One-Bit Unsigned Addition

C_{in}		A		B		C_{out}	$\sum$
0	+	0	+	0	=	0	0
0	+	1	+	0	=	0	1
0	+	1	+	1	=	1	0
1	+	1	+	1	=	1	1

Binary Addition Examples

Example 7.1



Basic Subtraction

- Basic Subtraction of $x = a - b$, with a = minuend**(被減數),** b = subtrahend **(減數),** and x = difference or result**(差).**
- Requires a **Borrow Bit** if $a < b$.
- There are other forms of subtraction such as **2's Complement Addition(用2的補數的加法作減法運算)** used by microprocessors (such as in a PC).

Basic Subtraction Rules

$$0 - 0 = 0$$
$$1 - 0 = 1$$
$$1 - 1 = 0$$
$$10 - 1 = 1 \qquad (2_{10} - 1_{10} = 1_{10})$$

When subtracting a 1 from a 0: borrow 1 from the next most significant bit

Binary Subtraction with Borrow Examples

Example 7.3, 7.4

```
  1110          110(10)        Borrow    Stage
- 1001          100    1
                010    1

10000           0111(10)       Borrow    ripples   to  LSB
-   101           10    1
                0101    1
```

7.2 Representing Signed Binary Numbers

- Sign Bit **(符號位元)**:
  - A bit (usually the MSB) that indicates whether a number is positive (= 0) or negative (= 1).
- Magnitude Bits**(大小位元)**:
  - The bits of a signed binary number that tell us how large the number is (i.e., its magnitude).

Signed Binary Numbers – 1
(符號大小表示法)

- **True-Magnitude Form:**
  - A form of signed binary number whose magnitude is represented in true binary (not complements).

Signed Binary Numbers – 2

- **1's Complement:**
  - A form of signed binary in which negative numbers are created by complementing all bits.

- **2's Complement:**
  - A form of signed binary in which the negative numbers are created by complementing all the bits and adding a 1 (1's Complement + 1).

True-Magnitude Form

Example 7.5

Write the following numbers in 6-bit true-magnitude form:

+25 = 011001 (Note sign bit (MSB) Sign = 0)

–25 = 111001 (Same as +25 with sign = 1)

+12 = 001100

–12 = 101100

1's Complement Form

Example 7.6

Convert the following numbers to 8-bit 1's Complement form:

+57 = 00111001

–57 = 11000110 (All Bits Inverted)

+72 = 01001000

–72 = 10110111

2's Complement Form

Example 7.7

Convert the following numbers to 8-bit 2's Complement form:

```
 57 = 0011 1001      72 = 0100 1000
-57 = 1100 0110     -72 = 1011 0111
           +1                  +1
      1100 0111           1011 1000
```

7.3 Signed Binary Arithmetic
Signed Addition

Example 7.7

Add +30 and +75.

- Signed Addition Positive (Sign bit = 0)

$$+30 = 0001\ 1100$$

$$+75 = 0100\ 1011$$

$$0101\ 1001$$

- Similar to binary addition with a sign bit.

Sign bit

Subtraction Using 1's Complement Method

- Add the 1's Complement and then Carry.

$+80 = 0101\ 0000 \quad (+80) = 0101\ 0000$

$-65 = 0100\ 0001 \quad (+65) = 1011\ 1110 \quad$ (1's Comp 65)

$1\ 0000\ 1110$

$+\ 1$

$0000\ 1111$

- Uses an End-around carry addition method.

Subtraction Using 2's Complement Method

- Add 2's Complement to Minuend(被減數).

+ 80	= 0101	0000		0101	0000
+ 65	= 0100	0001		+ 1011	1111
– 65	= 1011	1110	+ 1	1 0000	1111
				⇑	
				Discard Carry Bit from Result	

Negative Sum or Difference

Example 7.9

Subtract $65_{10} - 80_{10}$ in 8-bit 2's complement form.

■ **Solution**

$+65_{10} =$ 01000001
$+80_{10} =$ 01010000
$-80_{10} =$ 10101111 (1's complement)
$+$ 1
10110000 (2's complement)

(+65) 01000001
+ (–80) + 10110000
11110001

Take the 2's complement of the difference to find the positive number with the same magnitude.

11110001 (–15)
00001110 (1's complement)
+ 1
00001111 (2's complement) (+15)

$00001111 = +15_{10}$. We generated this number by complementing 11110001. Thus, $11110001 = -15_{10}$.

Range of Signed Numbers -1

TABLE 7.1 4-Bit 2's Complement Numbers

Decimal	2's Complement
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
–1	1111
–2	1110
–3	1101
–4	1100
–5	1011
–6	1010
–7	1001
–8	1000

Range of Signed Numbers -2

- Range of Positive Numbers is 0 to $2^n - 1$ for a number with *n* magnitude bits.

- Range of Negative Numbers is -1 to -2^n for a number with *n* magnitude bits.

- 8-Bit Example:

8-Bit Number Range is $-2^n \leq x \leq +2^n - 1$

or -128 to $+127$

Example 7.10

Write the largest positive and negative numbers for an 8-bit signed number in 2's complement and decimal notation.

Solution

01111111 = + 127 (7 magnitude bits: $2^7 - 1 = 127$)

10000000 = – 128 (1 followed by seven 0s: $-2^7 = -128$)

Example 7.11

Write -16_{10}:

a. As an 8-bit 2's complement number
b. As a 5-bit 2's complement number

■ Solution

a. An 8-bit number has 7 magnitude bits and 1 sign bit.

$$
\begin{array}{rll}
+16 = & 00010000 & \\
-16 = & 11101111 & \text{(1's complement)} \\
+ & \underline{\qquad\quad 1} & \\
& 11110000 & \text{(2's complement)}
\end{array}
$$

b. A 5-bit number has 4 magnitude bits and 1 sign bit. Four magnitude bits are not enough to represent +16. However, a 1 followed by n 0s is equal to -2^n. For a 1 and four 0s, $-2^n = -2^4 = -16$. Thus, $10000 = -16_{10}$.

Sign Bit Overflow

- **Overflow:**
  - An erroneous(不正確的) carry into the sign bit of a signed binary number
  - Results from a sum or difference that is larger than can be represented by the magnitude bits.

- Results in a False Positive or False Negative Number.

Overflow in Positive Sums

- Addition of two 8-Bit Positive Numbers:

Sign bit ↓

+ 75	=		0100	1011
+ 96	=	+	0110	0000
			1010	1011

Result is Negative (False)

- Two positive numbers added with a result greater than the range of $2^7 - 1 = +127$ for 8-bit numbers causes an overflow.

Overflow in Negative Sums

- Addition of two 8-Bit Negative Numbers:

Sign bit

$$
\begin{array}{lcll}
-80 & = & 1011\ \ 0000 & \\
-65 & = & +\ 1011\ \ 1111 & \\
\hline
 & & 0110\ \ 1111 & \text{Result is Positive (False)}
\end{array}
$$

- Two Negative numbers were added to produce a False Positive Result due to overflowing the negative range of 8-bit numbers ($-2^7 \leq sum \leq 2^7 - 1$)

$$(-128 \leq sum \leq 127)$$

Example 7.12 -1

Which of the following sums will produce a sign bit overflow in 8-bit 2's complement notation? How can you tell?

a. $67_{10} + 33_{10}$
b. $67_{10} + 63_{10}$
c. $-96_{10} - 22_{10}$
d. $-96_{10} - 42_{10}$

Solution

A sign bit overflow is generated if the sum of two positive numbers appears to produce a negative result or the sum of two negative numbers appears to produce a positive result. In other words, overflow occurs if the operand sign bits are both 1 and the sum sign bit is 0 or vice versa. We know this will happen if an 8-bit sum is outside the range ($-128 \leq \text{sum} \leq +127$).

Example 7.12 -2

Solution

a.	(+67)	01000011	(no overflow;
	+(+33)	+00100001	sum of positive numbers
	(+100)	01100100	is positive.)
b.	(+67)	01000011	(Overflow; sum of
	+(+63)	00111111	positive numbers is negative.
	+(130)	10000010	Sum > +127; out of range.)

Example 7.12 -3

Solution

c.

```
(+96) =   01100000
(–96) =   10011111      (1's complement)
        +         1
          ----------
          10100000      (2's complement)

(+22) =   00010110
(–22)     11101001      (1's complement)
        +         1
          ----------
          11101010      (2's complement)

  (–96)   10100000
+(–22)    11101010
 (–118) 1 10001010
```

(Magnitude bits)
(Sign bit)
(Discard carry)

(No overflow; sum of two negative numbers is negative.)

Example 7.12 -4

■ **Solution**

d.

```
(+96) =   01100000
(–96) =   10011111      (1's complement)
        +         1
          10100000      (2's complement)

(+42) =   00101010
(–42)     11010101      (1's complement)
        +         1
          11010110      (2's complement)

(–96)     10100000
+(–42)    11010110
(–138)  1 01110110
```

(Magnitude bits)
(Sign bit)
(Discard carry)

(Overflow; sum of two negative numbers is positive. Sum < –128; out of range.)

7.4 Hexadecimal Addition

Hex Addition

Hex addition is very much like decimal addition, except that we must remember how to deal with the hex digits A to F. A few sums are helpful:

$$F + 1 = 10$$
$$F + F = 1E$$
$$F + F + 1 = 1F$$

The positional multipliers for the hexadecimal system are powers of 16. Thus, the most significant digit of the first sum is in the 16's column. The equivalent sum in decimal is:

$$15_{10} + 1_{10} = 16_{10} = 10H$$

The second sum is the largest possible sum of two hex digits; the carry to the next position is 1. This shows that the sum of two hex digits will never produce a carry larger than 1. The second sum can be calculated as follows:

$$\begin{aligned} FH + FH &= 15_{10} + 15_{10} \\ &= 30_{10} \\ &= 16_{10} + 14_{10} \\ &= 10H + EH \\ &= 1EH \end{aligned}$$

Hexadecimal Addition

- Example 7.13

Add 6B3H + A9CH.

Solution

Hex	Decimal Equivalents
6B3	(6) (11) (3)
+A9C	+ (10) (9) (12)
	(16) (20) (15)

For sums greater than 15, subtract 16 and carry 1 to the next position:

	Hex	Decimal Equivalents
(Carry) ———	11	(1) (1)
	6B3	(6) (11) (3)
	+ A9C	+ (10) (9) (12)
	114F	(1) (1) (4) (15)

Sum: 6B3H + A9CH = 114FH.

Hexadecimal Subtraction

- Example 7.14

Subtract 6B3H – 49CH.

■ Solution

Hex	Decimal Equivalent
6B3	(6) (11) (3)
– 49C	– (4) (9) (12)

To subtract the least significant digits, we must borrow 10H (16_{10}) from the previous position. This leaves the subtraction looking like this:

Hex	Decimal Equivalent
(Borrow) ——— 1	
6B3 ~~6A3~~	(6) (10) (16 + 3)
– 49C	– (4) (9) (12)
217	(2) (1) (7)

Example 7.15

Negate the hex number 15AC by calculating its 16's complement.

■ Solution

```
  FFFF
- 15AC
  EA53     (15's complement)
+    1
  EA54     (16's complement)
```

The original value, 15AC, can be restored by calculating the 16's complement of EA54. Try it.

Example 7.16

Subtract 8B63 – 55D7 using the complement method.

■ Solution

Find the 16's complement of 55D7.

```
   FFFF
 – 55D7
   AA28      (15's complement)
 +    1
   AA29      (16's complement)
```

Therefore, –55D7 = AA29.

```
    1
   8B63
 + AA29
 1 358C
```

(Discard carry)

Difference: 8B63 – 55D7 = 358C.

7.5 Numeric and Alphanumeric (字母與數字的) Codes

- **BCD Code (Binary-Coded Decimal):** A code used to represent each decimal digit of a number by a 4-Bit Binary Value.

- Valid Digits for 0 to 9 are 0000 to 1001.
  - The binary codes 1010 to 1111 are invalid

- Called an 8421 Code due to the decimal weight of each bit position.

BCD Examples

- Each digit is a 4-Bit Binary group:
- $(84)_{10}$ = 1000 0100
- $(4987)_{10}$ = 0100 1001 1000 0111_{BCD}

Decimal Digits and Their 8421 BCD Equivalents

Decimal Digit	BCD (8421)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Example 7.17

Write 4987_{10} in both binary and 8421 BCD.

■ Solution

The binary value of 4987_{10} can be calculated by repeated division by 2:

$$4987_{10} = 1\ 0011\ 0111\ 1011_2$$

The BCD digits are the binary values of each decimal digit, encoded separately. We can break bits into groups of 4 for easier reading. Note that the first and last BCD digits each have a leading zero to make them 4 bits long.

$$4987_{10} = 0100\ 1001\ 1000\ 0111_{BCD}$$

4 9 8 7

Excess-3 Code (超3碼)

- A BCD Code formed by adding 3 (0011) to its true 4-bit binary value.
- Inverting the bits of the Excess-3 digit yields 9's Complement of the decimal equivalent.

Excess-3 Code (超3碼)

TABLE 7.3 Decimal Digits and Their 8421 and Excess-3 Equivalents

Decimal Digit	8421	Excess-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

Excess-3 code is a type of BCD code that is generated by adding 11_2 (3_{10}) to the 8421 BCD codes. Table 7.3 shows the Excess-3 codes and their 8421 and decimal equivalents.

The advantage of this code is that it is **self-complementing.** If the bits of the Excess-3 digit are inverted, they yield the **9's complement** of the decimal equivalent.

Excess-3 Examples

- 3 = 0011 + 0011 = 0110 = 6 in Excess-3.
- 1 = 0001 + 0011 = 0100 = 4 in Excess-3.
- If we complement 1 = 1011 in Excess-3, this is the code for an 8.
  - 9's Complement of 1 = (9 – 1) = 8 (Self- Complement)

Gray Code (格雷碼)

- A binary code that progresses so that only one bit changes between two successive codes.

- Binary: $b_3b_2b_1b_0$

- Gray: $g_3g_2g_1g_0$

- Gray code bits can be defined as follows:

$$g_3 = b_3$$
$$g_2 = b_3 \oplus b_2$$
$$g_1 = b_2 \oplus b_1$$
$$g_0 = b_1 \oplus b_0$$

4-Bit Gray Code

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

ASCII Code -1

- **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange.

- A seven-bit alphanumeric(字母與數字的) code used to represent text letters, numerals(數的), punctuation(標點符號), and special controls.

ASCII Code -2

TABLE 7.5 ASCII Code

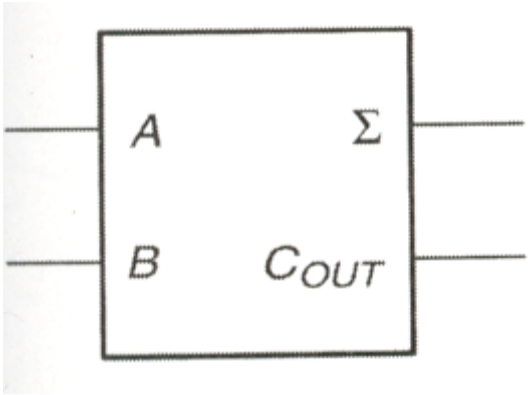
LSBs \ MSBs	000 (0)	001 (1)	010 (2)	011 (3)	100 (4)	101 (5)	110 (6)	111 (7)
0000 (0)	NUL	DLE	SP	0	@	P	`	p
0001 (1)	SOH	DC1	!	1	A	Q	a	q
0010 (2)	STX	DC2	"	2	B	R	b	r
0011 (3)	ETX	DC3	#	3	C	S	c	s
0100 (4)	EOT	DC4	$	4	D	T	d	t
0101 (5)	ENQ	NAK	%	5	E	U	e	u
0110 (6)	ACK	SYN	&	6	F	V	f	v
0111 (7)	BEL	ETB	'	7	G	W	g	w
1000 (8)	BS	CAN	(	8	H	X	h	x
1001 (9)	HT	EM	)	9	I	Y	i	y
1010 (A)	LF	SUB	*	:	J	Z	j	z
1011 (B)	VT	ESC	+	;	K	[	k	{
1100 (C)	FF	FS	,	<	L	\	l	|
1101 (D)	CR	GS	-	=	M	]	m	}
1110 (E)	SO	RS	.	>	N	^	n	~
1111 (F)	SI	US	/	?	O	—	o	DEL

7.6 Binary Adders and Subtractors

- **Half Adder (HA):** A circuit that will add two bits and produce a sum bit and a carry bit.

- **Full Adder (FA):** A circuit that will add a carry bit from another HA or FA and two operand bits to produce a sum bit and a carry bit.

Half Adder



Half Adder

There are only three possible sums of two 1-bit binary numbers:

$0 + 0 = 00$

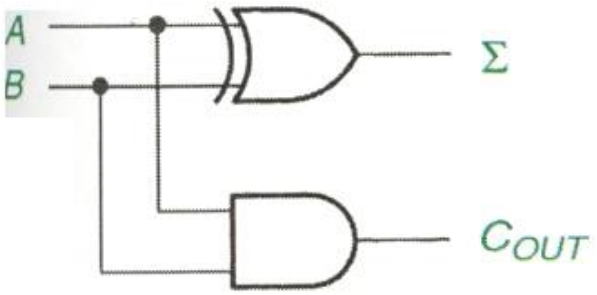
$0 + 1 = 01$

$1 + 1 = 10$

Half Adder Truth Table

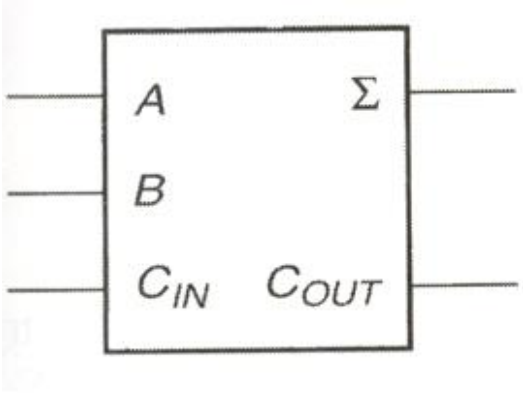
A	B	C_{OUT}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$C_{OUT} = AB$$
$$\Sigma = \bar{A}B + A\bar{B} = A \oplus B$$



Half Adder Circuit

Full Adder



Full Adder

A **full adder,** can add two 1-bit numbers *and* accept a carry bit from a previous adder stage.

$$0 + 0 + 0 = 00$$
$$0 + 0 + 1 = 01$$
$$0 + 1 + 1 = 10$$
$$1 + 1 + 1 = 11$$

Full Adder Truth Table

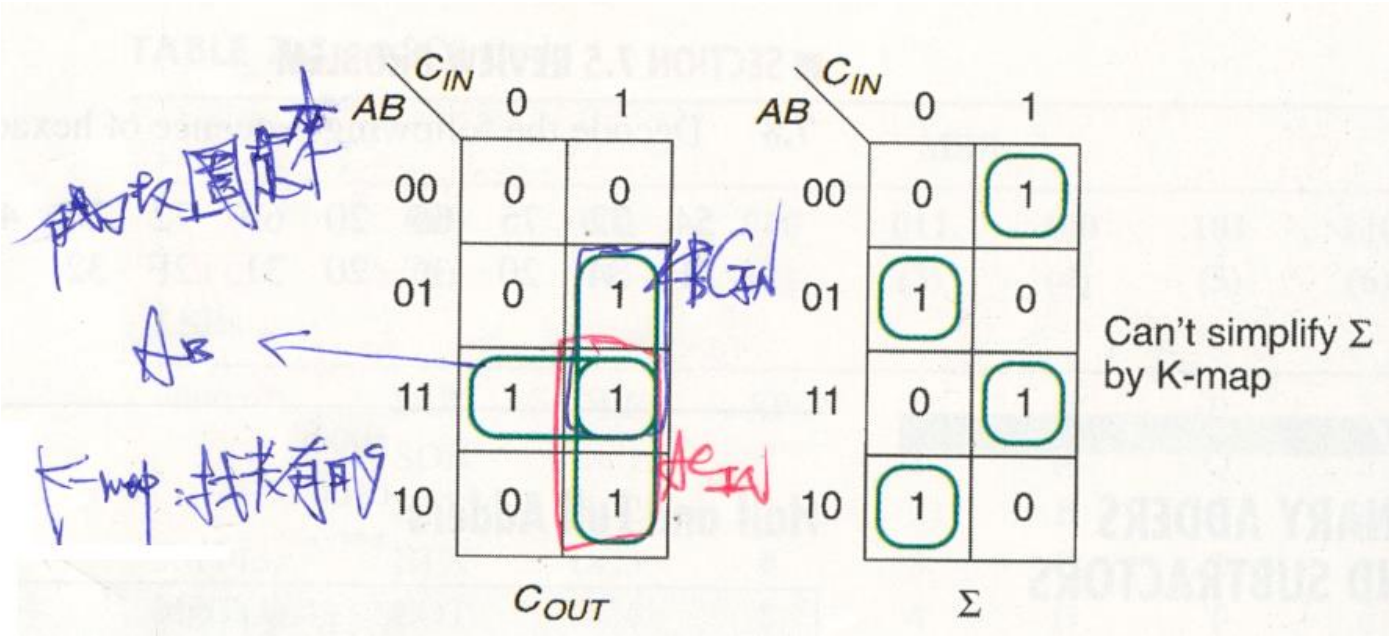
A	B	C_{IN}	C_{OUT}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The unsimplified Boolean expressions for the outputs are:

$$C_{OUT} = \bar{A}\,B\,C_{IN} + A\,\bar{B}\,C_{IN} + A\,B\,\bar{C}_{IN} + A\,B\,C_{IN}$$
$$\Sigma = \bar{A}\,\bar{B}\,C_{IN} + \bar{A}\,B\,\bar{C}_{IN} + A\,\bar{B}\,\bar{C}_{IN} + A\,B\,C_{IN}$$

There are a couple of ways to simplify these expressions.

Full Adder: Karnaugh Map Method -1



$$C_{OUT} = A\,B + A\,C_{IN} + B\,C_{IN}$$

The expression for Σ doesn't reduce at all.

logic circuits for Σ and C_{OUT}, don't give us much of a simplification.

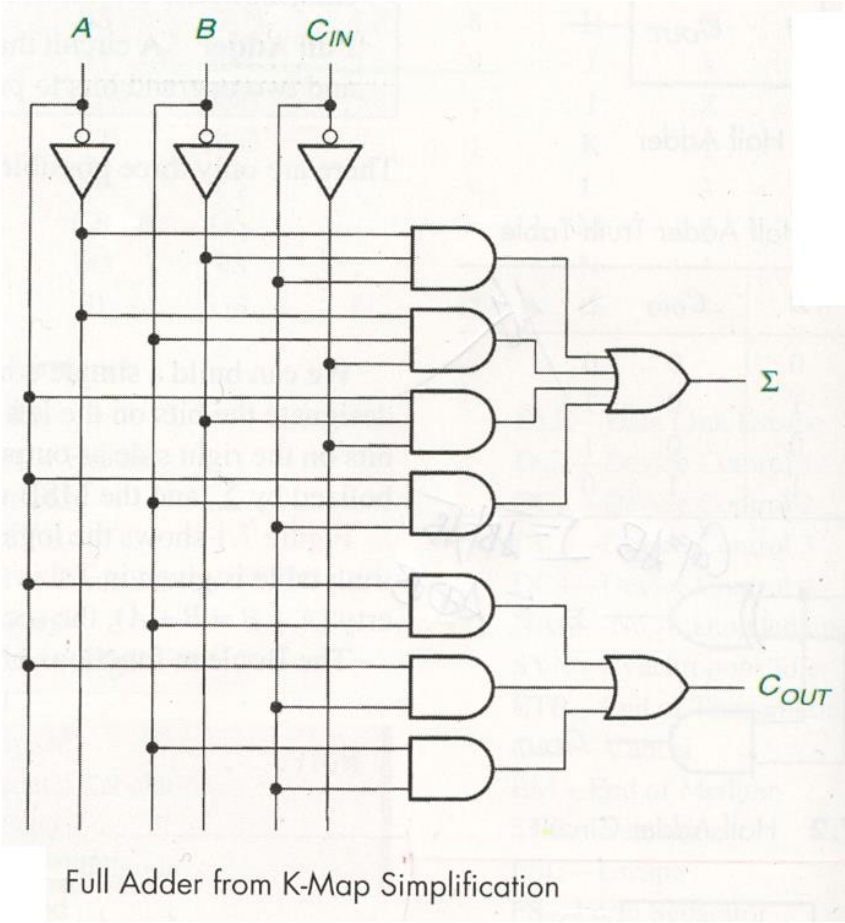
Full Adder: Karnaugh Map Method -2

$$C_{OUT} = \bar{A}\,B\,C_{IN} + A\,\bar{B}\,C_{IN} + A\,B\,\bar{C}_{IN} + A\,B\,C_{IN}$$
$$\Sigma = \bar{A}\,\bar{B}\,C_{IN} + \bar{A}\,B\,\bar{C}_{IN} + A\,\bar{B}\,\bar{C}_{IN} + A\,B\,C_{IN}$$

K-Map

$$C_{OUT} = A\,B + A\,C_{IN} + B\,C_{IN}$$

The expression for Σ doesn't reduce at all.



Full Adder from K-Map Simplification

Full Adder: Boolean Algebra Method -1

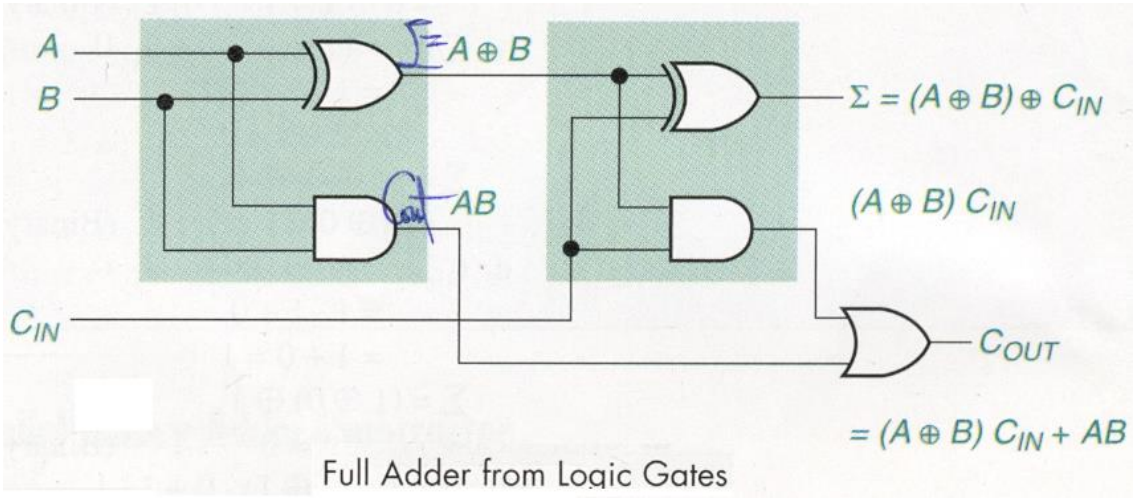
$$
\begin{aligned}
C_{OUT} &= \bar{A}\,B\,C_{IN} + A\,\bar{B}\,C_{IN} + A\,B\,\bar{C}_{IN} + A\,B\,C_{IN} \\
&= (\bar{A}\,B + A\,\bar{B})C_{IN} + A\,B\,(\bar{C}_{IN} + C_{IN}) \\
&= (A \oplus B)\,C_{IN} + A\,B
\end{aligned}
$$

$$
\begin{aligned}
\sum &= (\bar{A}\,\bar{B} + A,B)\,C_{IN} + (\bar{A}\,B + A\,\bar{B})\,\bar{C}_{IN} \\
&= (\overline{A \oplus B})\,C_{IN} + (A \oplus B)\,\bar{C}_{IN} \qquad \text{Let } x = A \oplus B \\
&= \bar{x}\,C_{IN} + x\,\bar{C}_{IN} \\
&= x \oplus C_{IN} \\
&= (A \oplus B) \oplus C_{IN}
\end{aligned}
$$

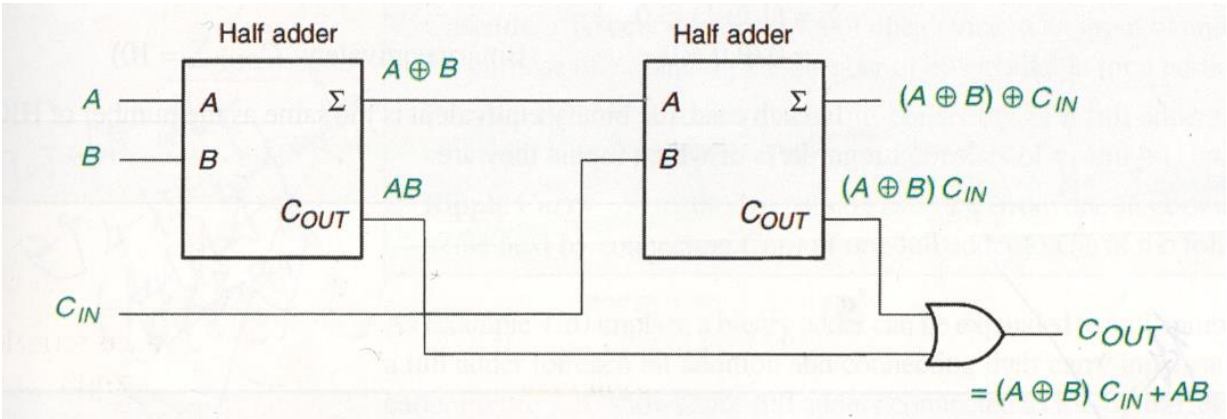
$$
\begin{aligned}
C_{OUT} &= (A \oplus B)\,C_{IN} + A\,B \\
\sum &= (A \oplus B) \oplus C_{IN}
\end{aligned}
$$

Full Adder: Boolean Algebra Method -2

$$C_{OUT} = (A \oplus B)\, C_{IN} + A\, B$$
$$\Sigma = (A \oplus B) \oplus C_{IN}$$



Full Adder from Logic Gates



Full Adder from Two Half Adders

Example 7.19 -1

Evaluate the Boolean expression for Σ and C_{OUT} of the full adder in Figure 7.8 for the following input values. What is the binary value of the outputs in each case?

a. $A = 0, B = 0, C_{IN} = 1$
b. $A = 1, B = 0, C_{IN} = 0$
c. $A = 1, B = 0, C_{IN} = 1$
d. $A = 1, B = 1, C_{IN} = 0$

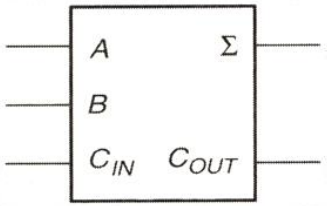


FIGURE 7.8 Example 7.19: Full Adder

The output of a full adder for any set of inputs is simply given by $C_{OUT}\,\Sigma = A + B + C_{IN}$. For each of the stated sets of inputs:

a. $C_{OUT}\,\Sigma = A + B + C_{IN} = 0 + 0 + 1 = 01$
b. $C_{OUT}\,\Sigma = A + B + C_{IN} = 1 + 0 + 0 = 01$
c. $C_{OUT}\,\Sigma = A + B + C_{IN} = 1 + 0 + 1 = 10$
d. $C_{OUT}\,\Sigma = A + B + C_{IN} = 1 + 1 + 0 = 10$

Example 7.19 -2

We can verify each of these sums algebraically by plugging the specified inputs into the full adder Boolean equations:

$$C_{OUT} = (A \oplus B)\, C_{IN} + A\, B$$
$$\Sigma = (A \oplus B) \oplus C_{IN}$$

a. $C_{OUT} = (0 \oplus 0) \cdot 1 + 0 \cdot 0$
$= 0 \cdot 1 + 0$
$= 0 + 0 = 0$
$\Sigma = (0 \oplus 0) \oplus 1$
$= 0 \oplus 1 = 1$ (Binary equivalent: $C_{OUT}\,\Sigma = 01$)

b. $C_{OUT} = (1 \oplus 0) \cdot 0 + 1 \cdot 0$
$= 1 \cdot 0 + 0$
$= 0 + 0 = 0$
$\Sigma = (1 \oplus 0) \oplus 0$
$= 1 \oplus 0 = 1$ (Binary equivalent: $C_{OUT}\,\Sigma = 01$)

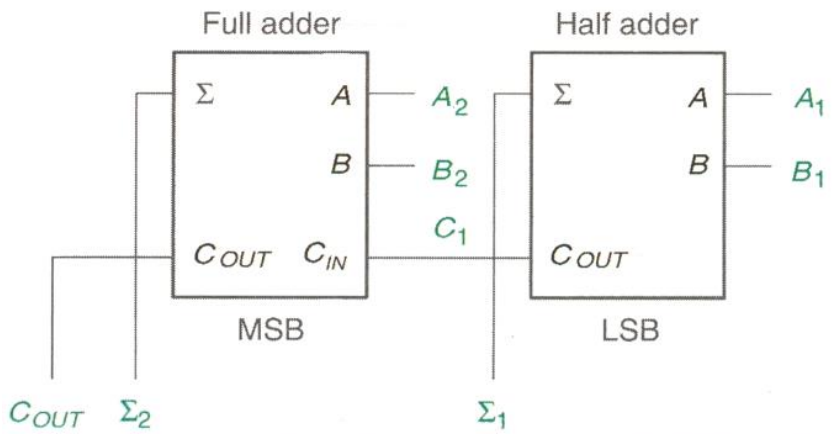
c. $C_{OUT} = (1 \oplus 0) \cdot 1 + 1 \cdot 0$
$= 1 \cdot 1 + 0$
$= 1 + 0 = 1$
$\Sigma = (1 \oplus 0) \oplus 1$
$= 1 \oplus 1 = 0$ (Binary equivalent: $C_{OUT}\,\Sigma = 10$)

d. $C_{OUT} = (1 \oplus 1) \cdot 0 + 1 \cdot 1$
$= 0 \cdot 0 + 1$
$= 0 + 1 = 1$
$\Sigma = (1 \oplus 1) \oplus 0$
$= 0 \oplus 0 = 0$ (Binary equivalent: $C_{OUT}\,\Sigma = 10$)

Example 7.20

Combine a half adder and a full adder to make a circuit that will add two 2-bit numbers. Check that the circuit will work by adding the following numbers and writing the binary equivalents of the inputs and outputs:

a. $A_2\,A_1 = 01,\ B_2\,B_1 = 01$
b. $A_2\,A_1 = 11,\ B_2\,B_1 = 10$



2-Bit Adder

Sums

a. $01 + 01 = 010$

$A_1 = 1, B_1 = 1 \Rightarrow C_1 = 1, \Sigma_1 = 0$

$A_2 = 0, B_2 = 0, C_1 = 1 \Rightarrow C_2 = 0, \Sigma_2 = 1$

(Binary equivalent: $A_2\,A_1 + B_2\,B_1 = C_2\,\Sigma_2\,\Sigma_1 = 010$)

b. $11 + 10 = 101$

$A_1 = 1, B_1 = 0 \Rightarrow C_1 = 0, \Sigma_1 = 1$

$A_2 = 1, B_2 = 1, C_1 = 0 \Rightarrow C_2 = 1, \Sigma_2 = 0$

(Binary equivalent: $A_2\,A_1 + B_2\,B_1 = C_2\,\Sigma_2\Sigma_1 = 101$)

Parallel Binary Adders/Subtractor

- Parallel Binary Adder

→ A circuit, consisting of *n* full adders, that will add *n*-bit binary numbers.

→ The output consists of *n* sum bits and a carry bit.

→ C_{OUT} of one full adder is connected to C_{IN} of the next full adder.

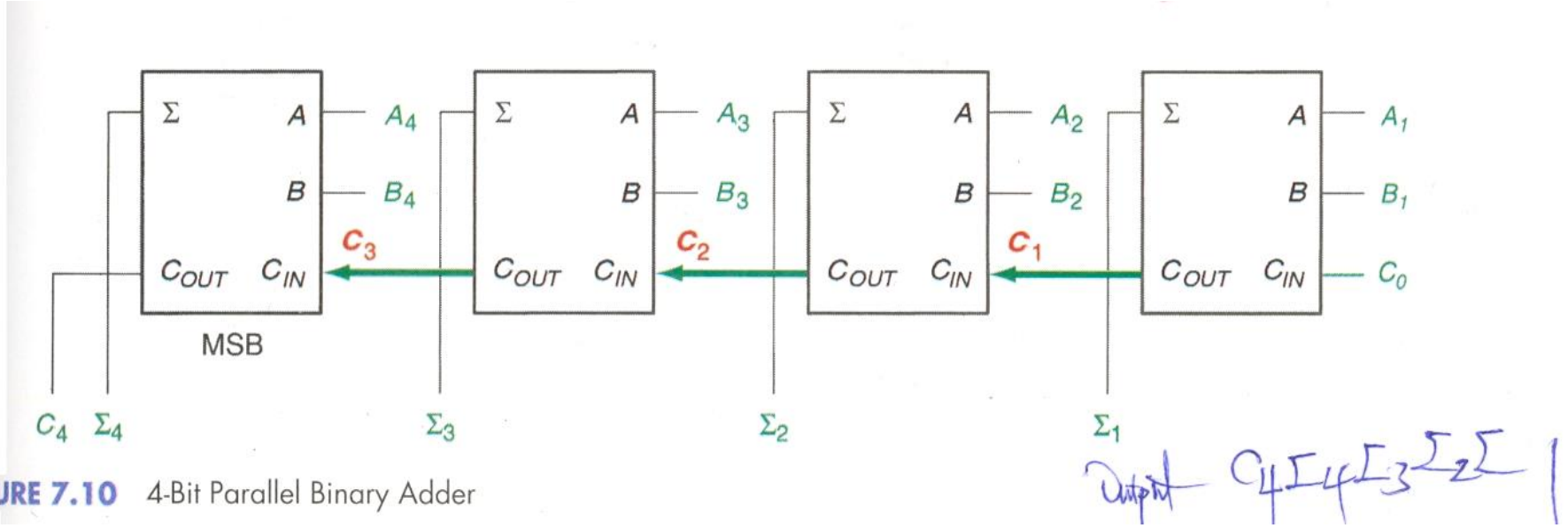


FIGURE 7.10 4-Bit Parallel Binary Adder

Example 7.21

Verify the summing operation of the circuit in Figure 7.10 by calculating the output for the following sets of inputs:

a. $A_4\,A_3\,A_2\,A_1 = 0101,\ B_4\,B_3\,B_2\,B_1 = 1001$
b. $A_4\,A_3\,A_2\,A_1 = 1111,\ B_4\,B_3\,B_2\,B_1 = 0001$

Solution

At each stage, $A + B + C_{IN} = C_{OUT}\,\Sigma$.

a. $0101 + 1001 = 1110$
$(5_{10} + 9_{10} = 14_{10})$
$A_1 = 1, B_1 = 1, C_0 = 0;\ C_1 = 1, \Sigma_1 = 0$
$A_2 = 0, B_2 = 0, C_1 = 1;\ C_2 = 0, \Sigma_2 = 1$
$A_3 = 1, B_3 = 0, C_2 = 0;\ C_3 = 0, \Sigma_3 = 1$
$A_4 = 0, B_4 = 1, C_3 = 0;\ C_4 = 0, \Sigma_4 = 1$
(Binary equivalent: $C_4\,\Sigma_4\,\Sigma_3\,\Sigma_2\,\Sigma_1 = 01110$)

b. $1111 + 0001 = 10000$
$(15_{10} + 1_{10} = 16_{10})$
$A_1 = 1, B_1 = 1, C_0 = 0;\ C_1 = 1, \Sigma_1 = 0$
$A_2 = 1, B_2 = 0, C_1 = 1;\ C_2 = 1, \Sigma_2 = 0$
$A_3 = 1, B_3 = 0, C_2 = 1;\ C_3 = 1, \Sigma_3 = 0$
$A_4 = 1, B_4 = 0, C_3 = 1;\ C_4 = 1, \Sigma_4 = 0$
(Binary equivalent: $C_4\,\Sigma_4\,\Sigma_3\,\Sigma_2\,\Sigma_1 = 10000$)

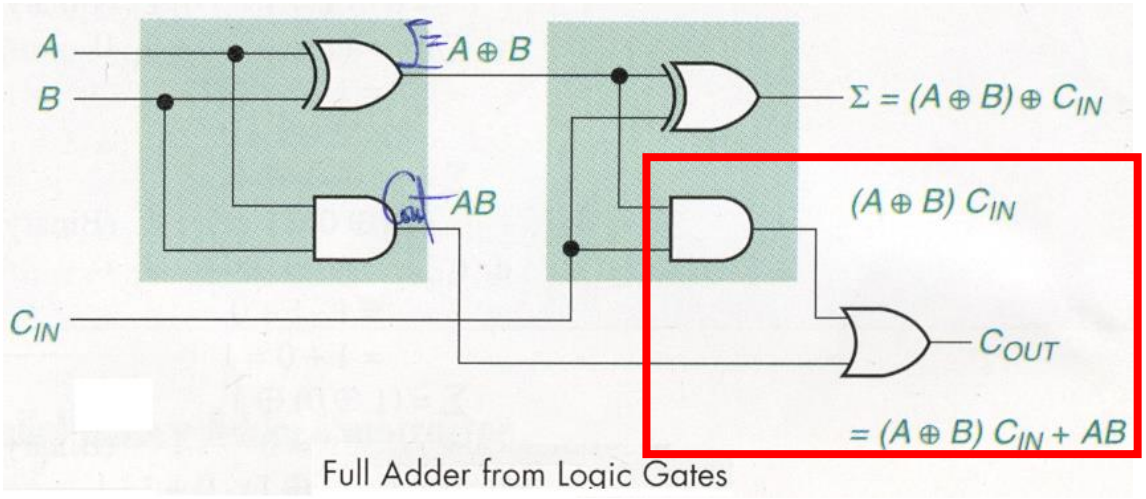
Ripple Carry – 1

- In the *n*-Bit Parallel Adder (FA Stages) the Carryout is generated by the last stage.

- This is called a Ripple Carry Adder because the final carryout (Last Stage) is based on a ripple through each stage by C_{IN} at the LSB Stage.

Ripple Carry – 2

- Each Stage will have a propagation delay on the C_{IN} to C_{OUT} of one AND Gate and one OR Gate.
- A 4-Bit Ripple Carry Adder will then have a propagation delay on the final C_{OUT} of $4 \times 2 = 8$ Gates.
- A 32-Bit adder such as in an MPU in a PC could have a delay of 64 Gates.



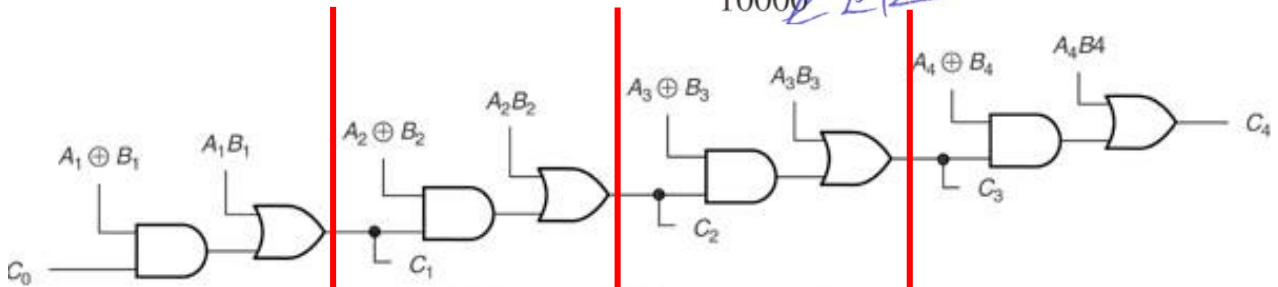
Full Adder from Logic Gates

Ripple Carry – 3

$1111 + 0001 = 10000$
$(15_{10} + 1_{10} = 16_{10})$
$A_1 = 1, B_1 = 1, C_0 = 0;\ C_1 = 1, \Sigma_1 = 0$
$A_2 = 1, B_2 = 0, C_1 = 1;\ C_2 = 1, \Sigma_2 = 0$
$A_3 = 1, B_3 = 0, C_2 = 1;\ C_3 = 1, \Sigma_3 = 0$
$A_4 = 1, B_4 = 0, C_3 = 1;\ C_4 = 1, \Sigma_4 = 0$
(Binary equivalent: $C_4\ \Sigma_4\ \Sigma_3\ \Sigma_2\ \Sigma_1 = 10000$)

Examine the sum (1111 + 0001 = 10000). For a parallel adder having a ripple carry, the output goes through the following series of changes as the carry bit propagates through the circuit:

$C_4\ \Sigma_4\ \Sigma_3\ \Sigma_2\ \Sigma_1 = 01111$
01110
01100
01000
10000



4-Bit Ripple Carry Chain

Fast Carry – 1

- **Fast Carry** or **Look-Ahead Carry**:
  - A combinational network that generates the final C_{OUT} directly from the operand bits (A_1 to A_n, B_1 to B_n).
  - It is independent of the operations of each FA Stage (as the ripple carry is).

- Fast Carry has a small propagation delay compared to the ripple carry.

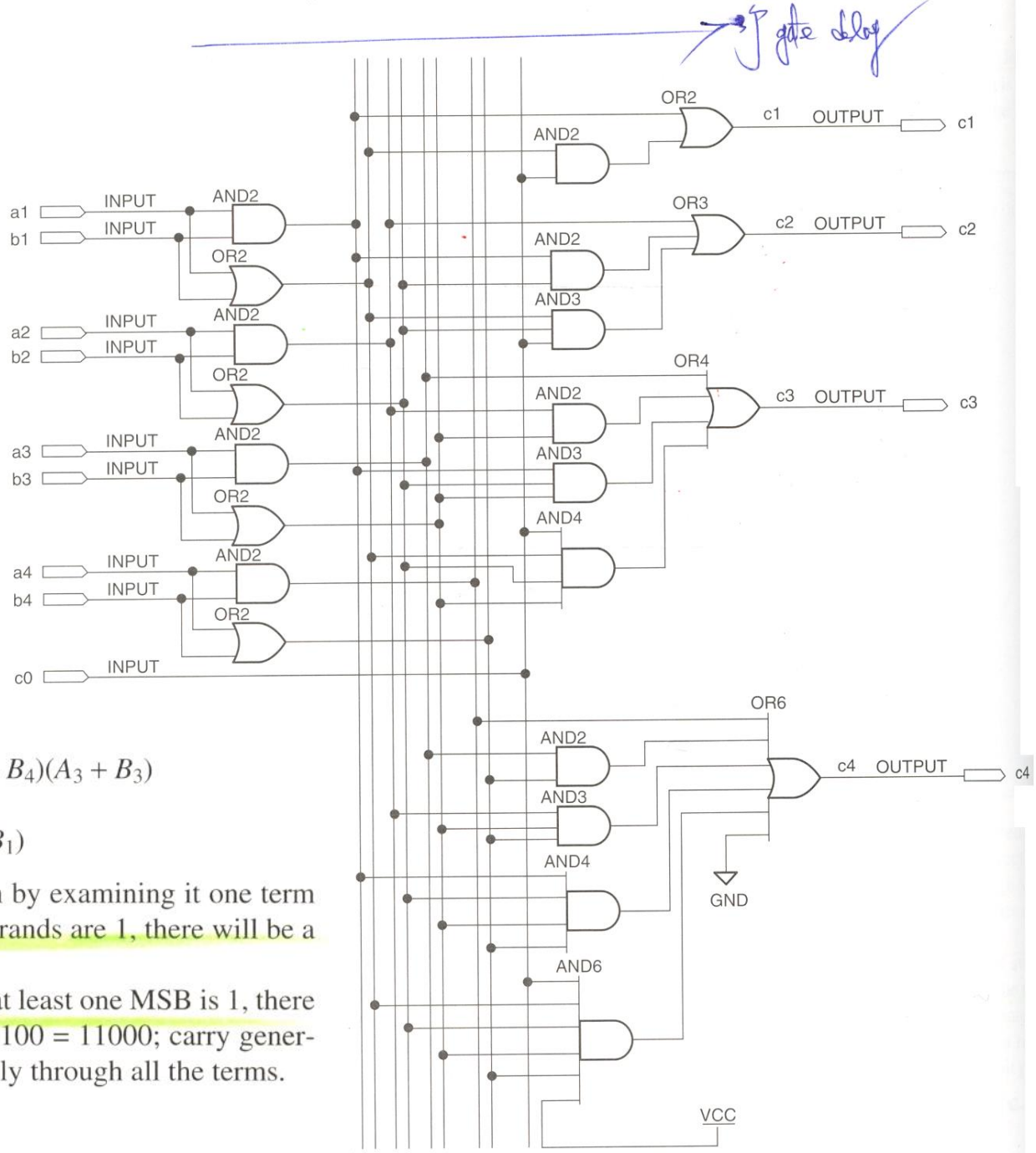
- The fast carry delay is 3 Gates for a 4-Bit Adder compared to 8 for the Ripple Carry.

Fast Carry – 2

$$C_4 = A_4 B_4 + A_3 B_3 (A_4 + B_4) + A_2 B_2 (A_4 + B_4)(A_3 + B_3)$$
$$+ A_1 B_1 (A_4 + B_4)(A_3 + B_3)(A_2 + B_2)$$
$$+ C_0 (A_4 + B_4)(A_3 + B_3)(A_2 + B_2)(A_1 + B_1)$$

We can make some intuitive sense of this expression by examining it one term at one time. The first term says if the MSBs of both operands are 1, there will be a carry (e.g., 1000 + 1000 = 10000; carry generated).

The second term says if both second bits are 1 AND at least one MSB is 1, there will be a carry (e.g., 0100 + 1100 = 10000, or 1100 + 1100 = 11000; carry generated in either case). This pattern can be followed logically through all the terms.



4-Bit Fast Carry Circuit

Fast Carry – 3

$$C_n = A_n B_n + C_{n-1}(A_n + B_n)$$

The algebraic expressions for the remaining carry bits are:

$$C_1 = A_1 B_1 + C_0 (A_1 + B_1)$$
$$C_2 = A_2 B_2 + A_1 B_1 (A_2 + B_2) + C_0 (A_2 + B_2)(A_1 + B_1)$$
$$C_3 = A_3 B_3 + A_2 B_2 (A_3 + B_3) + A_1 B_1 (A_3 + B_3)(A_2 + B_2) + C_0 (A_3 + B_3)(A_2 + B_2)(A_1 + B_1)$$

Using VHDL Components to Implement a Parallel Adder

Hierarchy A group of design entities associated in a series of levels or layers in which complete designs form portions of another, more general design entity. The more general design is considered to be the higher level of the hierarchy.

Component A complete VHDL design entity that can be used as a part of a higher-level file in a hierarchical design.

Component Declaration Statement A statement that defines the input and output port names of a component used in a VHDL design entity.

Component Instantiation Statement A statement that maps port names of a VHDL component to the port names, internal signals, or variables of a higher-level VHDL design entity.

Port An input or output of a VHDL design entity or component.

Instantiate To use an instance of a component.

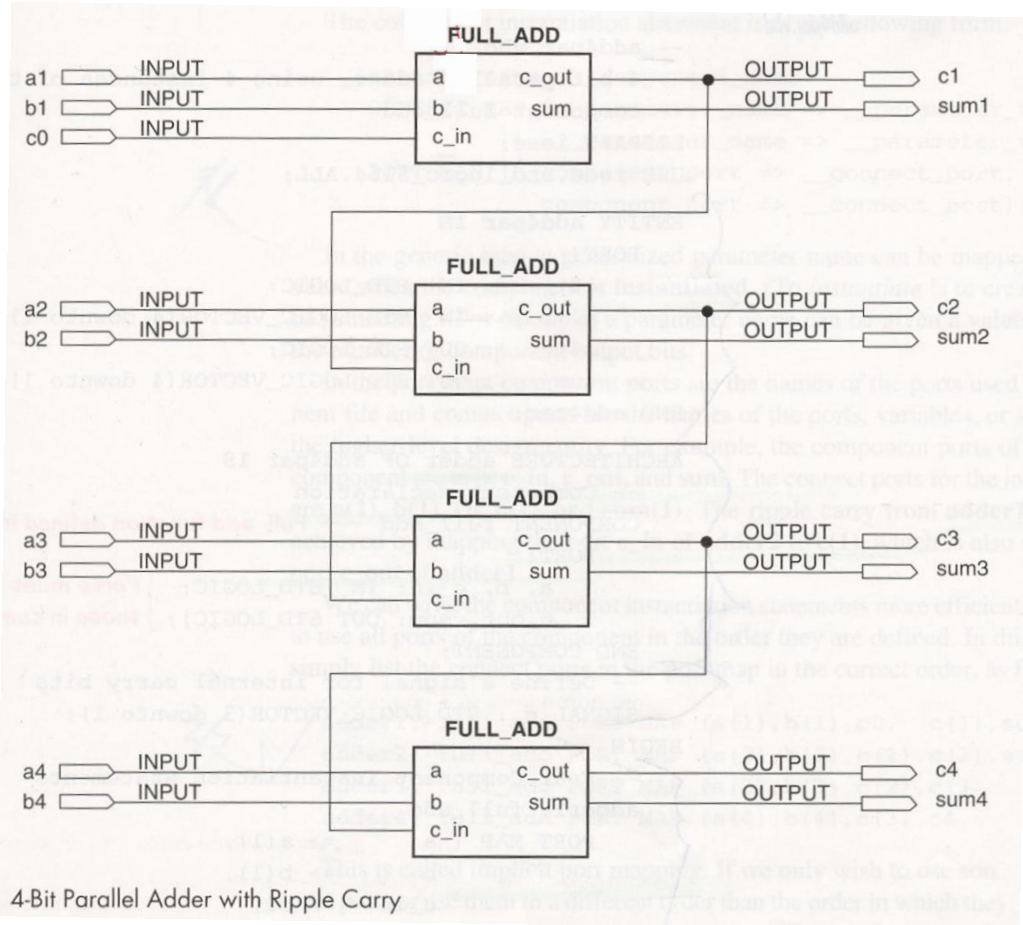
VHDL: 4-Bit Parallel Adder with Ripple Carry

$$C_{OUT} = (A \oplus B)\, C_{IN} + A\, B$$
$$\Sigma = (A \oplus B) \oplus C_{IN}$$

```
-- full_add.vhd
-- Full adder: adds two bits, a and b, plus input carry
-- to yield sum bit and output carry.
LIBRARY ieee;
USE ieee.std_logic_1164. ALL;

ENTITY full_add IS
PORT (
    a, b, c_in : IN  STD_LOGIC;
    c_out, sum : OUT STD_LOGIC);
END full_add;

ARCHITECTURE adder OF full_add IS
BEGIN
   c_out <= ((a xor b) and c_in) or (a and b) ;
   sum   <= (a xor b) xor c_in;
END adder;
```



4-Bit Parallel Adder with Ripple Carry

VHDL: 4-Bit Parallel Adder with Ripple Carry Hierarchical Design -1

1. A separate component file for a full adder **(full_add.vhd),** saved in a folder where the compiler can find it (i.e., on a library path)
2. A **component declaration statement** in the top-level file of the design hierarchy
3. A **component instantiation statement** for each instance of the full adder component

The general form of a design entity using components is:

```
ENTITY entity_name IS
   PORT ( input and output definitions);
END entity_name;

ARCHITECTURE arch_name OF entity_name IS
   component declaration(s);
   signal declaration(s);
BEGIN
   Component instantiation(s);
   Other statements;
END arch_name;
```

VHDL: 4-Bit Parallel Adder with Ripple Carry Hierarchical Design -2

The VHDL file for a 4-bit parallel adder using full adder components is shown next.

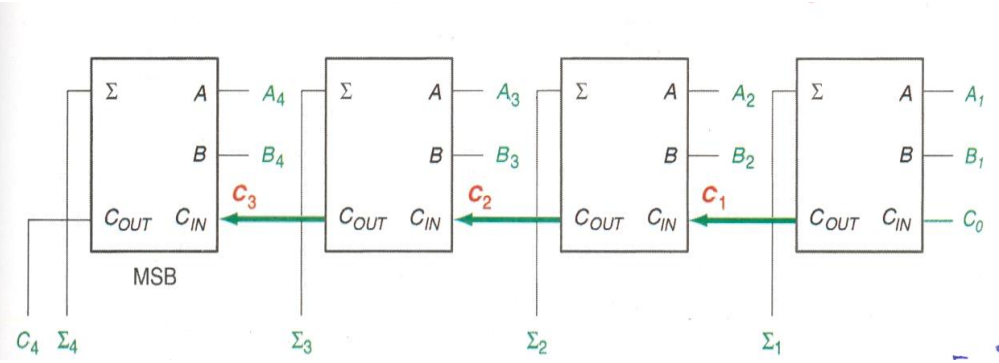
```
-- add4par.vhd
-- 4-bit parallel adder, using 4 instances of the
-- component full_add
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY add4par IS
   PORT(
      c0      : IN  STD_LOGIC;
      a, b    : IN  STD_LOGIC_VECTOR(4 downto 1);
      c4      : OUT STD_LOGIC;
      sum     : OUT STD_LOGIC_VECTOR(4 downto 1));
END add4par;

ARCHITECTURE adder OF add4par IS
   -- Component declaration
   COMPONENT full_add
      PORT(
         a, b, c_in: IN  STD_LOGIC;
         c_out, sum: OUT STD_LOGIC);
   END COMPONENT;
   -- Define a signal for internal carry bits
   SIGNAL c : STD_LOGIC_VECTOR(3 downto 1);
```

Full_add function defined in separate file.

Ports must be the same as those in component file.



VHDL: 4-Bit Parallel Adder with Ripple Carry Hierarchical Design -3

```
BEGIN
   -- Four Component Instantiation Statements
   adder1: full_add
      PORT MAP ( a      => a(1),
                 b      => b(1),
                 c_in   => c0,
                 c_out  => c(1),
                 sum    => sum (1));
   adder2: full_add
      PORT MAP ( a      => a(2),
                 b      => b(2),
                 c_in   => c(1),
                 c_out  => c(2),
                 sum    => sum(2));
   adder3: full_add
      PORT MAP ( a      => a(3),
                 b      => b(3),
                 c_in   => c(2),
                 c_out  => c(3),
                 sum    => sum (3));
   adder4: full_add
      PORT MAP ( a      => a(4),
                 b      => b(4),
                 c_in   => c(3),
                 c_out  => c4,
                 sum    => sum (4));
END adder;
```

Signal c(1) connects carry output of adder 1 to carry input of adder 2.

Ports of each full_add component are mapped explicitly.

User port and signal names

Component port names

VHDL: 4-Bit Parallel Adder
Generate Statement -1

GENERATE Statement A VHDL construct that is used to create repetitive portions of hardware.

```
-- add4gen.vhd
-- 4-bit parallel adder, using a generate statement and
-- components
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY add4gen IS
   PORT(
      c0     : IN  STD_LOGIC;
      a, b   : IN  STD_LOGIC_VECTOR(4 downto 1);
      c4     : OUT STD_LOGIC;
      sum    : OUT STD_LOGIC_VECTOR(4 downto 1));
END add4gen;
```

VHDL: 4-Bit Parallel Adder
Generate Statement -2

```
ARCHITECTURE adder OF add4gen IS
   -- Component declaration
   COMPONENT full_add
      PORT(
         a, b, c_in : IN  STD_LOGIC;
         c_out, sum : OUT STD_LOGIC);
   END COMPONENT;
   -- Define a signal for internal carry bits
   SIGNAL c : STD_LOGIC_VECTOR (4 downto 0);
BEGIN
   c(0)   <=  c0;
   adders:
   FOR i IN 1 to 4 GENERATE
      adder: full_add PORT MAP (a(i), b(i),
                c(i-1), c(i), sum(i));
   END GENERATE;
   c4 <= c(4);
END adder;
```

Input port c0 mapped to internal signal c(0).

Implicit port mapping

Output port c4 mapped to internal signal c(4).

2's Complement Subtractor – 1

For example, to find the difference 0101 – 0011 by 2's complement subtraction:

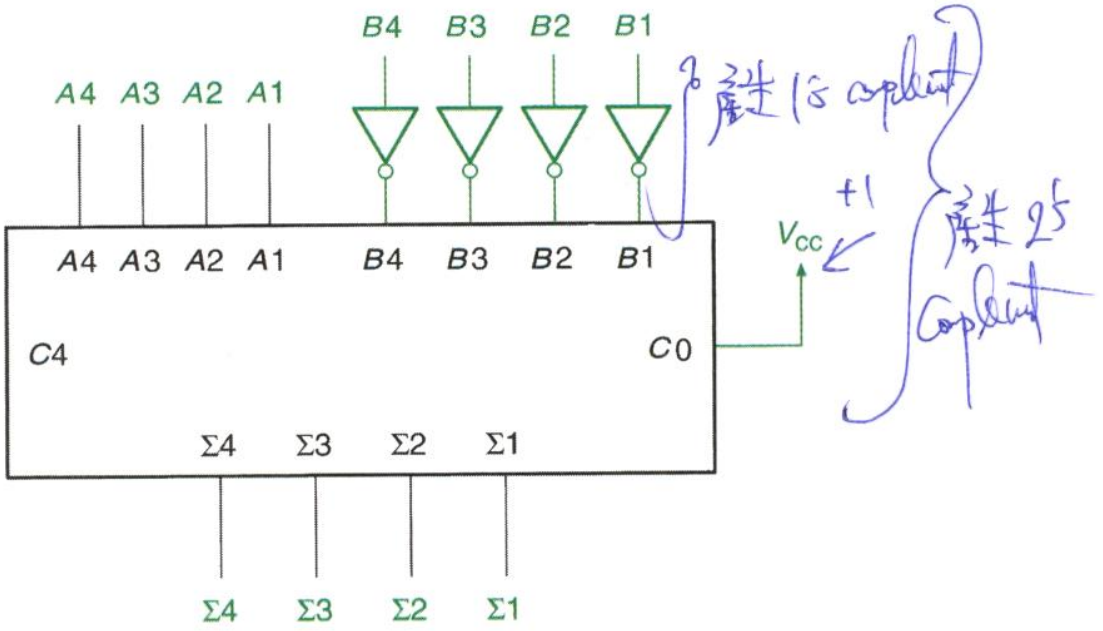
1. Find the 2's complement of 0011:

```
 0011
 1100      (1's complement)
  +1
 1101      (2's complement)
```

2. Add the 2's complement of the subtrahend to the minuend:

```
   0101     (+5)
 + 1101     (–3)
 1 0010     (+2)
(Discard carry)
```

2's Complement Subtractor – 2



2's Complement Subtractor

The four inverters generate the 1's complement of B. The parallel adder generates the 2's complement by adding the carry bit (held at logic 1) to the 1's complement at the B inputs. Algebraically, this is expressed as:

$$A - B = A + (-B) = A + \overline{B} + 1$$

where $\overline{B}$ is the 1's complement of B, and $(\overline{B} + 1)$ is the 2's complement of B.

Example 7.22

Verify the operation of the 2's complement subtractor in Figure 7.14 by subtracting:

a. 1001 – 0011 (unsigned)
b. 0100 – 0111 (signed)

Solution

Let $\overline{B}$ be the 1's complement of B.

a. Inverter inputs (B): 0011
Inverter outputs ($\overline{B}$): 1100
Sum ($A + \overline{B} + 1$):

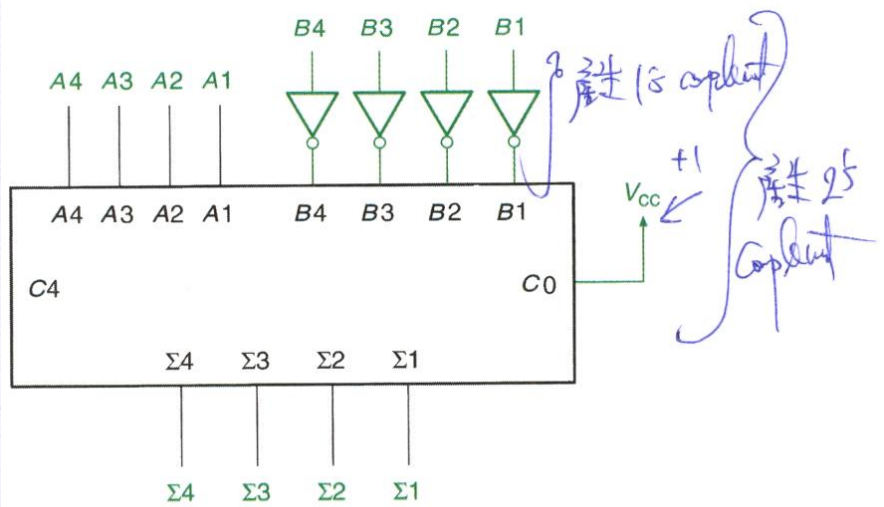
	1001	(9)
	1100	+ (–3)
+	1	
1	0110	(6)

(Discard carry)

b. Inverter inputs (B): 0111
Inverter outputs ($\overline{B}$): 1000
Sum ($A + \overline{B} + 1$):

	0100	(+4)
	1000	+ (–7)
+	1	
	1101	(–3)

Negative result: 1101 (–3)
1's complement of 1101: 0010
+ 1
2's complement of 1101: 0011 (+3)

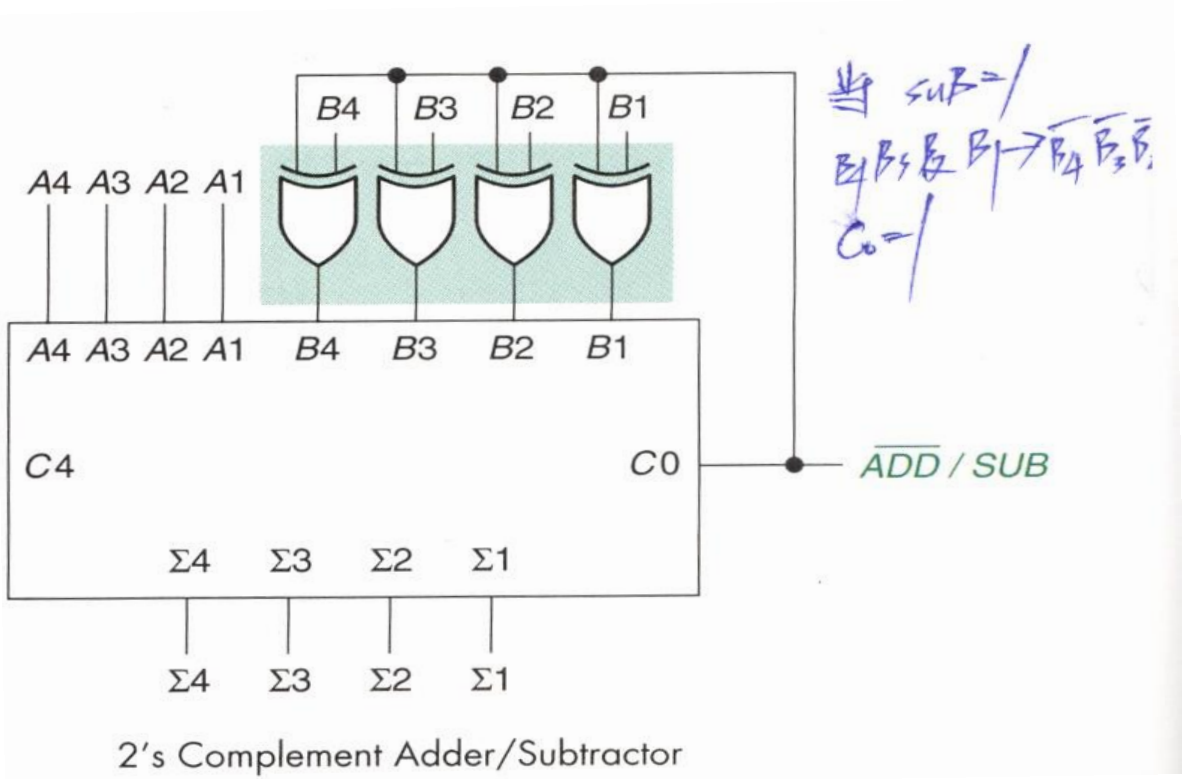


2's Complement Subtractor

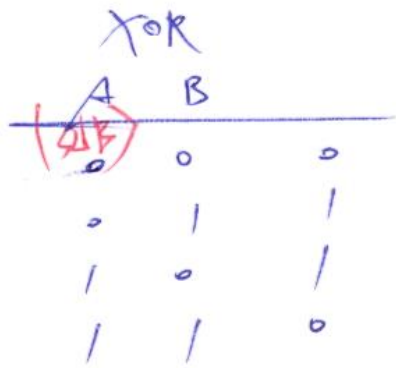
Figure 7.14

Parallel Binary Adder/Subtractor

a parallel binary adder configured as a programmable adde
subtractor. The Exclusive OR gates work as programmable inverters to pass B t
parallel adder in either true or complement form



2's Complement Adder/Subtractor



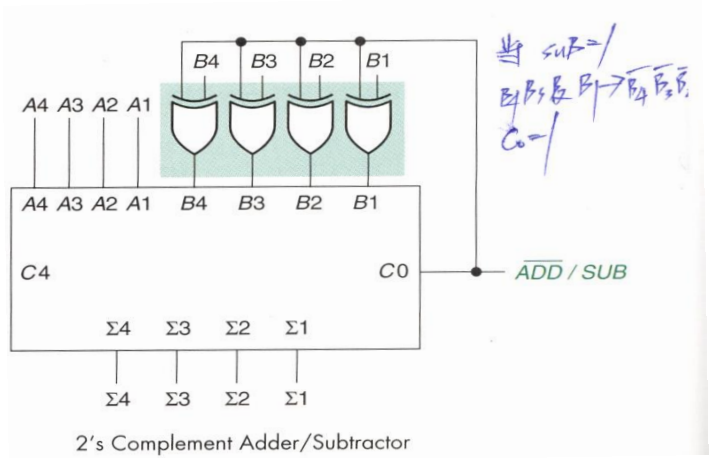
Example 7.23

Write a VHDL file to implement the 4-bit adder/subtractor shown in Figure 7.15. Write a set of simulation criteria to test the functionality of the adder/subtractor.

Solution

The VHDL file is as follows:

```
-- addsub4g.vhd
ENTITY addsub4g IS
   PORT (
       sub   : IN  BIT;
       a, b : IN  BIT_VECTOR (4 downto 1);
       c4    : OUT BIT;
       sum   : OUT BIT_VECTOR (4 downto 1));
END addsub4g;
```



2's Complement Adder/Subtractor

```
ARCHITECTURE adder OF addsub4g IS
   -- Component declaration
   COMPONENT full_add
       PORT (
           a, b, c_in : IN BIT;
           c_out, sum : OUT BIT);
   END COMPONENT;
   -- Define a signal for internal carry bits
   SIGNAL c       : BIT_VECTOR (4 downto 0);
   SIGNAL b_comp : BIT_VECTOR (4 downto 1);
```

```
BEGIN
   -- add/subtract select to carry input (sub=1 for subtract)
   c(0)   <= sub;
   adders:
   FOR i IN 1 to 4 GENERATE
      -- invert b for subtract (b(i) xor 1),
      -- do not invert for add (b(i) xor 0)
      b_comp(i) <= b(i) xor sub;
      adder: full_add PORT MAP (a(i), b_comp(i), c(i-1), c(i),
sum(i));
   END GENERATE;
   c4   <= c(4);
END adder;
```

Overflow Detection -1

- If the sign bits of both operands are the same and the sign bit of the sum is different from the operand sign bits, an overflow has occurred.

- Overflow is not possible if the sign bits of the operands are different from each other.

Overflow Detection -2

- Adding two 8-bit negative numbers:

80 H	1000 0000
+ 80 H	+ 1000 0000
100 H	10000 0000

(Sign bit overflow; $V = 1$)

- Adding two 8-bit positive numbers:

7FH	0111 1111
+ 01 H	0000 0001
80 H	1000 0000

(Sign bit overflow; $V = 1$)

Overflow Detection -3

An 8-bit parallel binary adder will add two signed binary numbers as follows:

$$\begin{array}{ll} S_A A_7 A_6 A_5 A_4 A_3 A_2 A_1 & (S_A = \text{Sign bit of } A) \\ S_B B_7 B_6 B_5 B_4 B_3 B_2 B_1 & (S_B = \text{Sign bit of } B) \\ \hline S_\Sigma \Sigma_7 \Sigma_6 \Sigma_5 \Sigma_4 \Sigma_3 \Sigma_2 \Sigma_1 & (S_\Sigma = \text{Sign bit of sum}) \end{array}$$

From our condition for overflow detection, we can make a truth table for an overflow variable, V, in terms of S_A, S_B, and S_Σ. Let us specify that $V = 1$ when there is an overflow condition. This condition occurs when $(S_A = S_B) \neq S_\Sigma$. Table 7.10 shows the truth table for the overflow detector function.

TABLE 7.10 Overflow Detector Truth Table

S_A	S_B	S_Σ	V
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

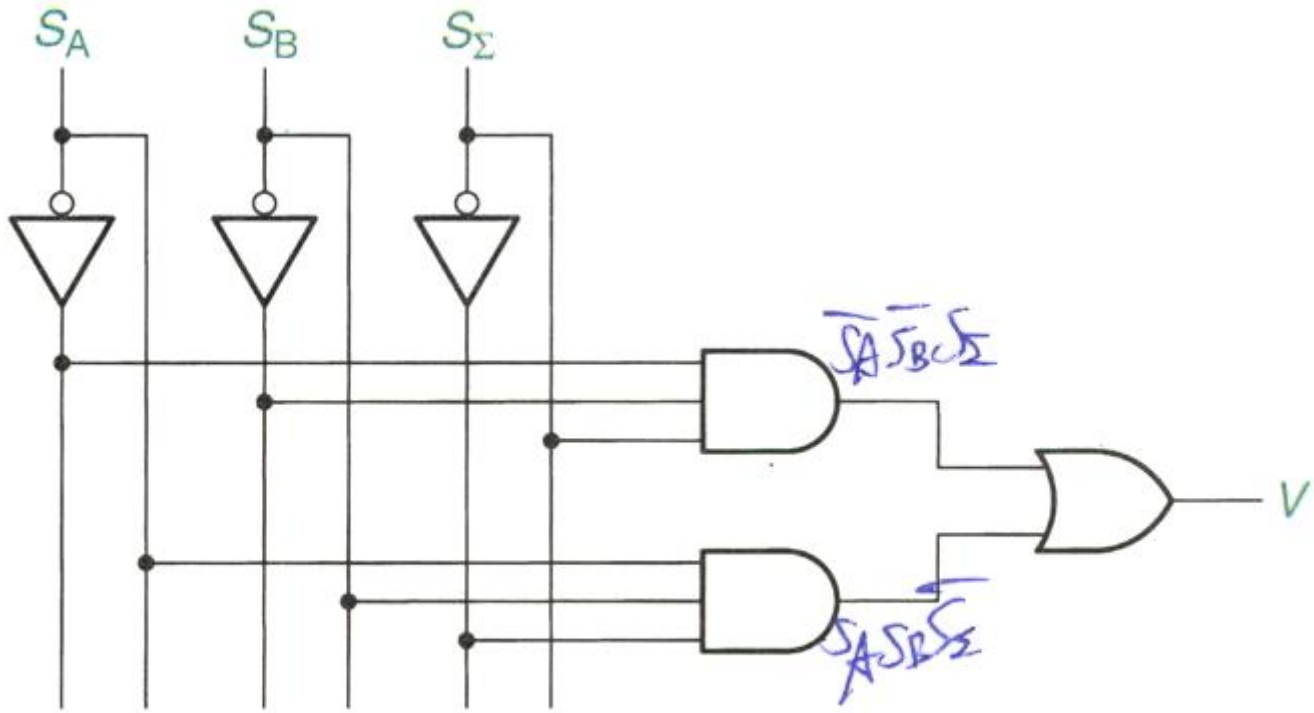
The SOP Boolean expression for the overflow detector is:

$$V = S_A S_B \overline{S}_\Sigma + \overline{S}_A \, \overline{S}_B \, S_\Sigma$$

Overflow Detection -4

The SOP Boolean expression for the overflow detector is:

$$V = S_A S_B \overline{S}_\Sigma + \overline{S}_A \, \overline{S}_B \, S_\Sigma$$



Overflow Detector

Example 7.24 -1

Combine two instances of the 4-bit adder/subtractor shown in Figure 7.15 and other logic to make an 8-bit adder/subtractor that includes a circuit to detect sign bit overflow.

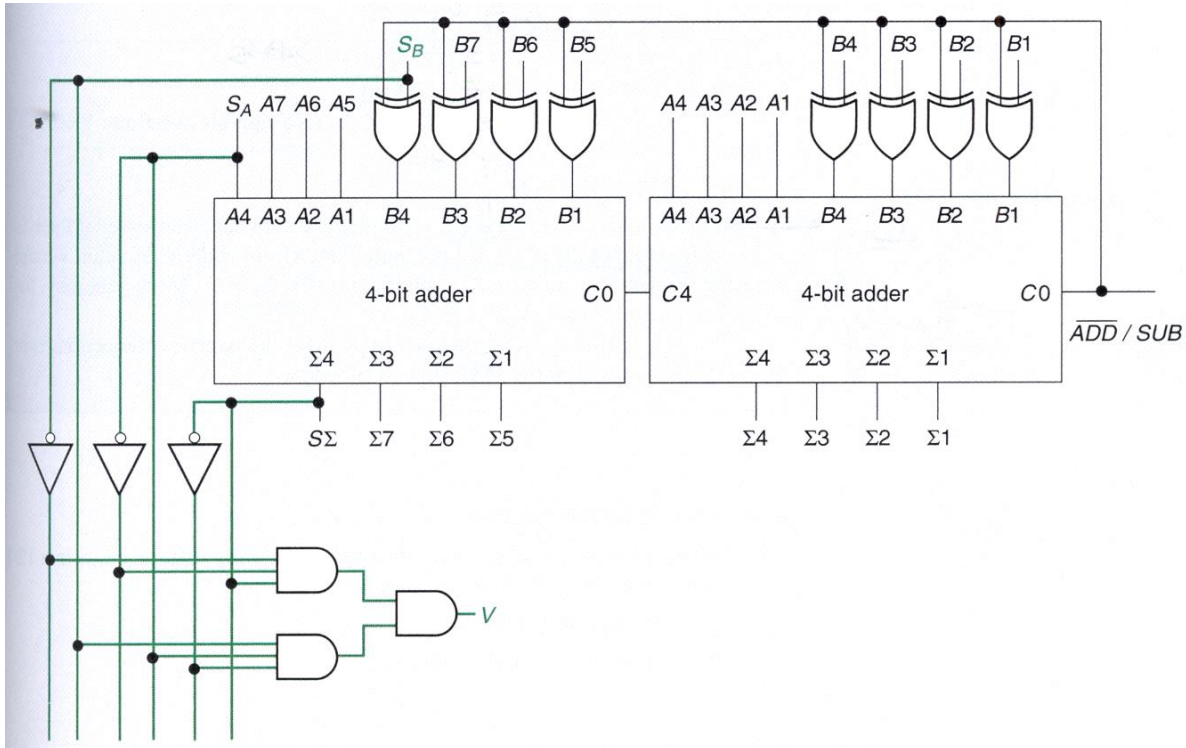


FIGURE 7.19 Example 7.24: 8-Bit Adder with Overflow Detector

Example 7.24 -2

A second method of overflow detection generates an overflow indication by examining the carry bits into and out of the MSB of a 2's complement adder/subtractor.

Consider the following 8-bit 2's complement sums. We will use our previous knowledge of overflow to see whether overflow occurs and then compare the carry bits into and out of the MSB.

a. 80H + 80H
b. 7FH + 01H
c. 7FH + 80H
d. 7FH + C0H

a. 80H = 10000000

```
  10000000
+ 10000000
 100000000    (Sign bit overflow; V = 1)
```

Carry into MSB = 0
Carry out of MSB = 1

Example 7.24 -3

b. 7FH = 01111111
01H = 00000001

```
 01111111
+00000001
 10000000
```

(Sign bit overflow; $V = 1$)

Carry into MSB = 1
Carry out of MSB = 0

c. 7FH = 01111111
80H = 10000000

```
 01111111
+10000000
 11111111
```

(No sign bit overflow; $V = 0$)

Carry into MSB = 0
Carry out of MSB = 0

d. 7FH = 01111111
C0H = 11000000

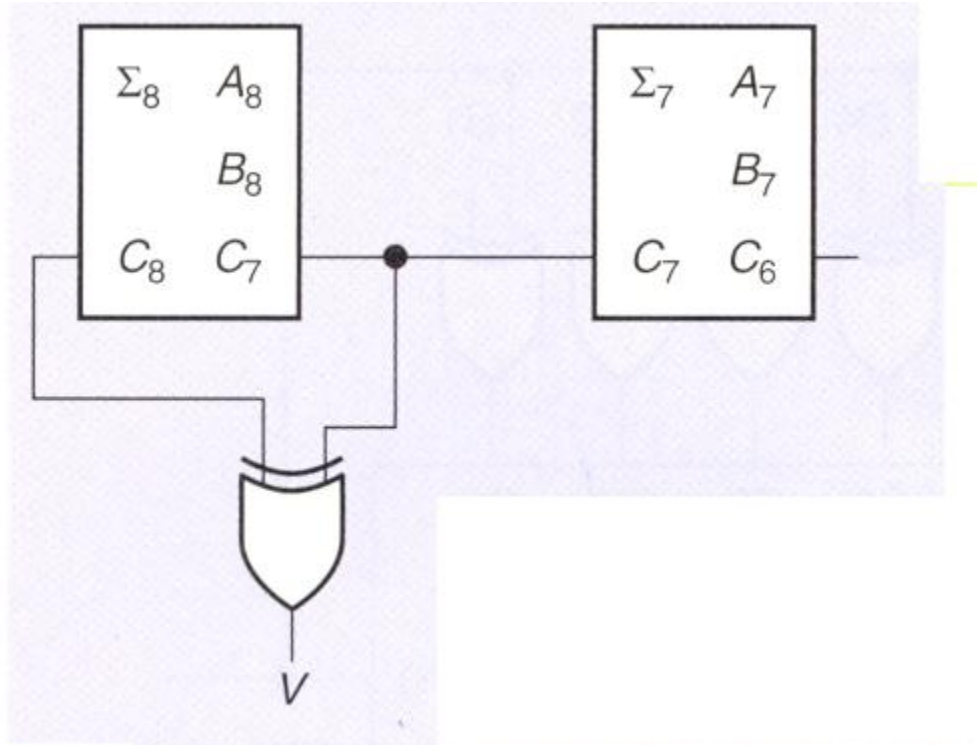
```
 01111111
+11000000
100111111
```

(No sign bit overflow; $V = 0$)

Carry into MSB = 1
Carry out of MSB = 1

These examples suggest that a 2's complement sum has overflowed if there is a carry into or out of the MSB, but not both. For an 8-bit adder/subtractor, we can write the Boolean equation for this condition as $V = C_8 \oplus C_7$. More generally, for an n-bit adder/subtractor, $V = C_n \oplus C_{n-1}$.

Example 7.24 -4



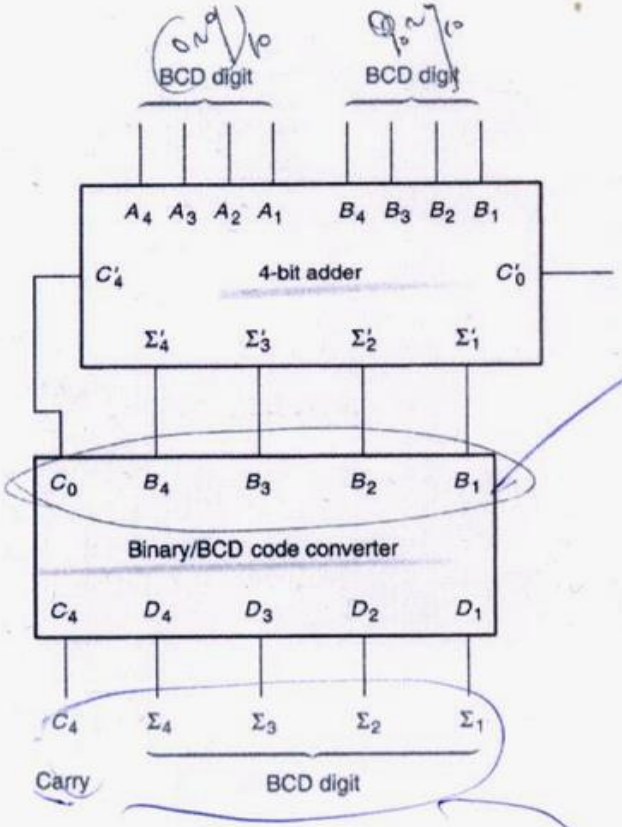
Over- flow Detector Using Internal Carry Bits

$$V = C_8 \oplus C_7$$

BCD Adder -1

- A Parallel Adder whose output sum is in groups of 4 bits, each representing a BCD (8421) Digit.
- Basic design is a 4-Bit Binary Parallel Adder to generate a 4-Bit Sum of $A + B$.
- Sum is input to the four-bit input of a Binary-to-BCD Code Converter.

BCD Adder -2



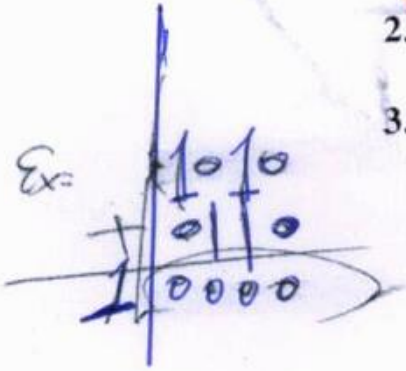
The MSD of the BCD sum is shown only as a carry bit, with leading zeros suppressed.

TABLE 7.11 Binary Sums of Two BCD Digits and a Carry Bit

Binary Sum ($A + B + C$)	Decimal	Corrected BCD (Carry + BCD)
00000	0	0 + 0000
00001	1	0 + 0001
00010	2	0 + 0010
00011	3	0 + 0011
00100	4	0 + 0100
00101	5	0 + 0101
00110	6	0 + 0110
00111	7	0 + 0111
01000	8	0 + 1000
01001	9	0 + 1001
01010	10	1 + 0000
01011	11	1 + 0001
01100	12	1 + 0010
01101	13	1 + 0011
01110	14	1 + 0100
01111	15	1 + 0101
10000	16	1 + 0110
10001	17	1 + 0111
10010	18	1 + 1000
10011	19	1 + 1001

BCD Adder -3

1. A carry output is generated if the binary sum is in the range 01010 ≤ sum ≤ 10011 (BCD equivalent: 1 0000 ≤ sum ≤ 1 1001).
2. If the binary sum is less than or equal to 01001, the output of the code converter is the same as the input.
3. If the sum is in the range 01010 ≤ sum ≤ 10011, the four LSBs of the input must be corrected to a BCD value. This can be done by adding 0110 to the four LSBs of the input and discarding any resulting carry. We add 0110_2 (6_{10}) because we must account for six unused codes.

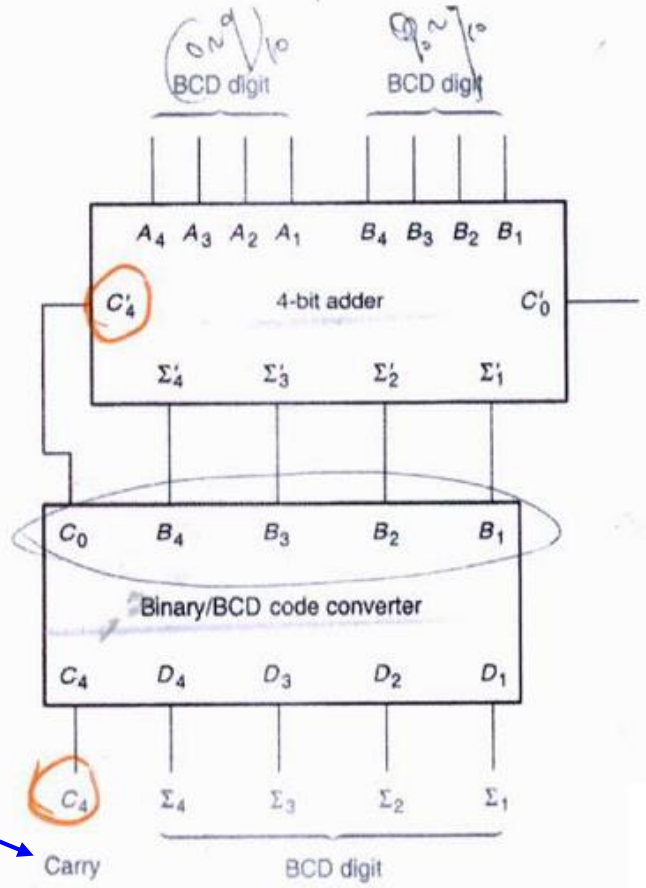


Carry Output -1

The carry output will be automatically 0 for any uncorrected sum from 00000 to 01001 and automatically 1 for any sum from 10000 to 10011. Thus, if the binary adder's carry output, which we will call C'_4, is 1, the BCD adder's carry output, C_4, will also be 1.

$$C_4 = \Sigma'_4\Sigma'_3\Sigma'_2\Sigma'_1 + C'_4 = C_0 B_4 B_3 B_2 B_1$$

uncorrected sum (A + B + C)	Decimal	Corrected BCD (Carry + BCD)
00000	0	0 + 0000
00001	1	0 + 0001
00010	2	0 + 0010
00011	3	0 + 0011
00100	4	0 + 0100
00101	5	0 + 0101
00110	6	0 + 0110
00111	7	0 + 0111
01000	8	0 + 1000
01001	9	0 + 1001
01010	10	1 + 0000
01011	11	1 + 0001
01100	12	1 + 0010
01101	13	1 + 0011
01110	14	1 + 0100
01111	15	1 + 0101
10000	16	1 + 0110
10001	17	1 + 0111
10010	18	1 + 1000
10011	19	1 + 1001



Carry Output -2

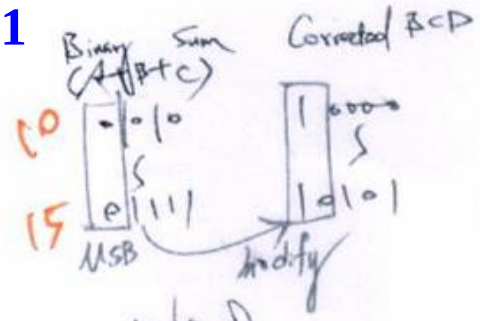
Any sum falling between these ranges, that is, between 01010 and 01111, must have its MSB modified. This modifying condition is a function, designated C_4'', of the binary adder's sum outputs when its carry output is 0.

uncorrected sum

Binary Sum ($A + B + C$)	Decimal	Corrected BCD (Carry + BCD)
00000	0	0 + 0000
00001	1	0 + 0001
00010	2	0 + 0010
00011	3	0 + 0011
00100	4	0 + 0100
00101	5	0 + 0101
00110	6	0 + 0110
00111	7	0 + 0111
01000	8	0 + 1000
01001	9	0 + 1001
01010	10	1 + 0000
01011	11	1 + 0001
01100	12	1 + 0010
01101	13	1 + 0011
01110	14	1 + 0100
01111	15	1 + 0101
10000	16	1 + 0110
10001	17	1 + 0111
10010	18	1 + 1000
10011	19	1 + 1001

$C_4'\Sigma_4\Sigma_3\Sigma_2\Sigma_1$

1



2

3

$\Sigma_3\Sigma_4$ \ $\Sigma_1\Sigma_2$	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$C_4' = \Sigma_4\Sigma_3 + \Sigma_4\Sigma_2$

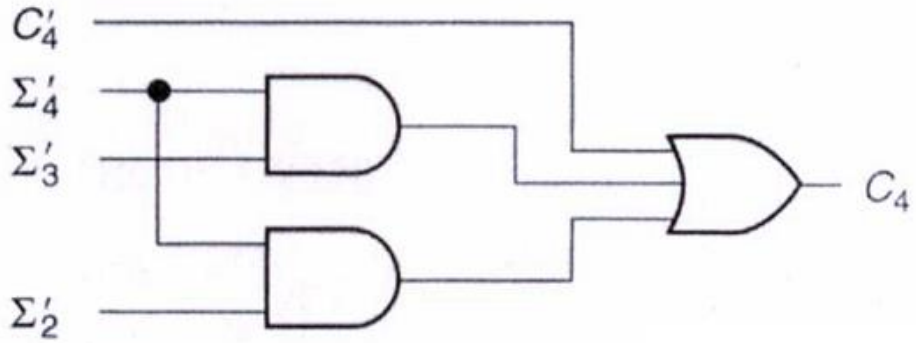
Carry Output -3

Boolean expression.

$$C''_4 = \Sigma'_4 \Sigma'_3 + \Sigma'_4 \Sigma'_2$$

The BCD carry output C_4 is given by:

$$C_4 = C'_4 + C''_4$$
$$= C'_4 + \Sigma'_4 \Sigma'_3 + \Sigma'_4 \Sigma'_2$$



BCD Carry Circuit

Sum Correction -1

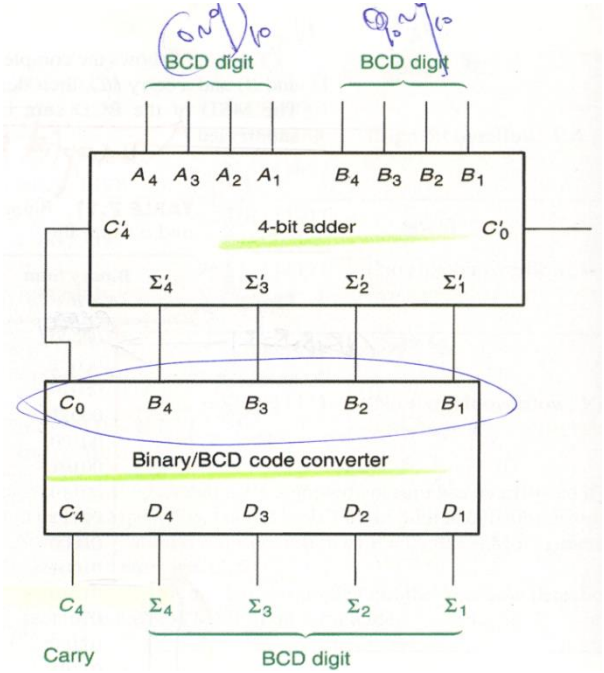
The four LSBs of the binary adder output need to be corrected if the sum is 01010 or greater and need not be corrected if the binary sum is 01001 or less. This condition

Binary Sum ($A + B + C$)	Decimal	Corrected BCD (Carry + BCD)
00000	0	0 + 0000
00001	1	0 + 0001
00010	2	0 + 0010
00011	3	0 + 0011
00100	4	0 + 0100
00101	5	0 + 0101
00110	6	0 + 0110
00111	7	0 + 0111
01000	8	0 + 1000
01001	9	0 + 1001
01010	10	1 + 0000
01011	11	1 + 0001
01100	12	1 + 0010
01101	13	1 + 0011
01110	14	1 + 0100
01111	15	1 + 0101
10000	16	1 + 0110
10001	17	1 + 0111
10010	18	1 + 1000
10011	19	1 + 1001

Sum Correction -2

is indicated by the BCD carry. Let us designate the binary sum outputs as $\Sigma'_4\ \Sigma'_3\ \Sigma'_2\ \Sigma'_1$ and the BCD sum outputs as $\Sigma_4\ \Sigma_3\ \Sigma_2\ \Sigma_1$.

$$\text{If } C_4 = 0,\ \Sigma_4\ \Sigma_3\ \Sigma_2\ \Sigma_1 = \Sigma'_4\ \Sigma'_3\ \Sigma'_2\ \Sigma'_1 + 0000;$$
$$\text{If } C_4 = 1,\ \Sigma_4\ \Sigma_3\ \Sigma_2\ \Sigma_1 = \Sigma'_4\ \Sigma'_3\ \Sigma'_2\ \Sigma'_1 + 0110$$

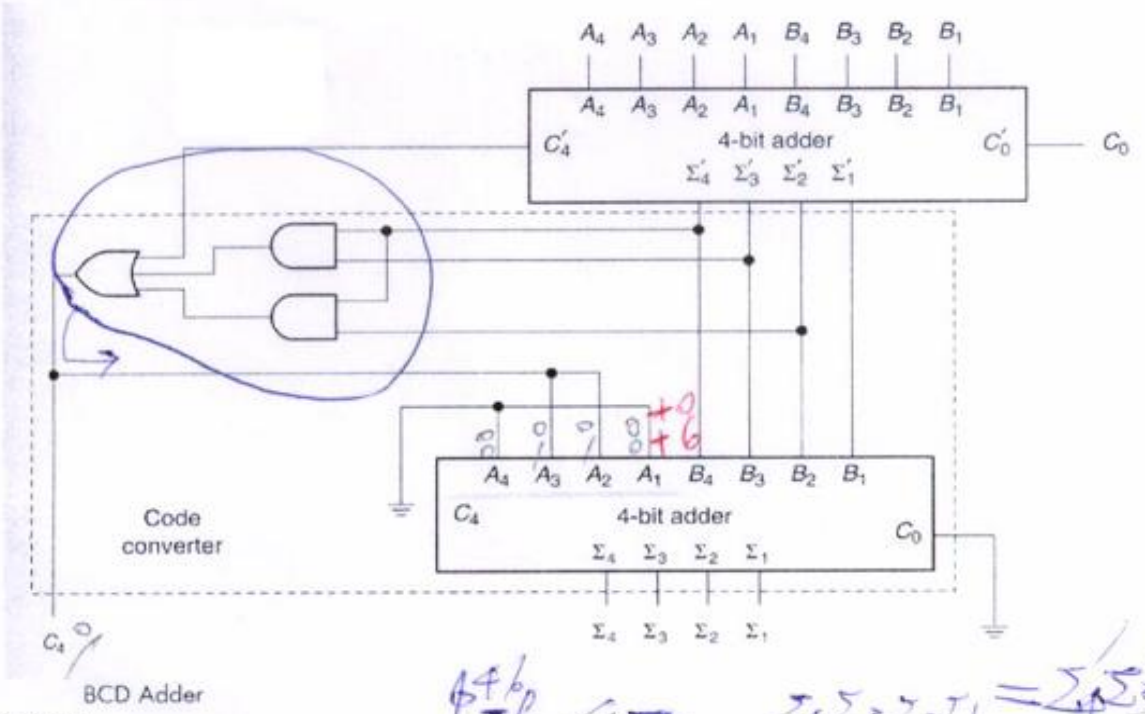


$C'_4\Sigma'_4\Sigma'_3\Sigma'_2\Sigma'_1$

Binary Sum ($A + B + C$)	Decimal	Corrected BCD (Carry + BCD)
00000	0	0 + 0000
00001	1	0 + 0001
00010	2	0 + 0010
00011	3	0 + 0011
00100	4	0 + 0100
00101	5	0 + 0101
00110	6	0 + 0110
00111	7	0 + 0111
01000	8	0 + 1000
01001	9	0 + 1001
01010	10	1 + 0000
01011	11	1 + 0001
01100	12	1 + 0010
01101	13	1 + 0011
01110	14	1 + 0100
01111	15	1 + 0101
10000	16	1 + 0110
10001	17	1 + 0111
10010	18	1 + 1000
10011	19	1 + 1001

Sum Correction -3

Figure 7.24 shows a BCD adder, complete with a binary adder, BCD carry, and sum correction. A second parallel adder is used for sum correction. The *B* inputs are the uncorrected binary sum inputs. The *A* inputs are either 0000 or 0110, depending on the value of the BCD carry.



HW