

微算機系統 Fall 2020

Microprocessor Systems

Instructor : Yen-Lin Chen(陳彥霖), Ph.D. Professor
Dept. Computer Science and Information Engineering
National Taipei University of Technology

微算機系統—使用VHDL設計

第2單元 數值表示法與數學電路

NUMERICAL REPRESENTATION AND
MATHEMATICAL CIRCUITS

介紹

- 本章將討論執行數學運算的邏輯電路。
- 將解釋如何對數值作加法、減法和乘法。
- 將說明如何撰寫VHDL程式碼以描述數學電路。
- 這些電路提供很好的平台，說明VHDL在複雜邏輯電路有各種用途而且功能強大。
- 設計數學電路的觀念可以應用在許多其他電路。

一、無符號整數 Unsigned Integers

- 只有正值的數值稱為無符號(unsigned)，也有負值的數值稱為帶正負號(signed)。
- 通常來說，十進位整數是由N個十進位數字組成

$$D = d_{n-1}d_{n-2} \cdots d_1d_0$$

- 代表下列值

$$V(D) = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

- 稱為位置數值表示法(position number representation)。

Unsigned Integers(cont'd)

- 在二進位數值系統中，也用同樣的位置數值表示法

$$B = b_{n-1}b_{n-2} \cdots b_1b_0$$

- 表示整數值為

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 2^1 + b_0 2^0$$

- 例如，二進位數值1101代表下列值

$$V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

- 因為對不同的基數而言特定數字的意義不同，我們將基數寫在**下標**以說明可能造成混淆的情況。因此若指定1101為二進位數值，則寫成 $(1101)_2$ 。計算先前的式子可以得到 $V = 8 + 4 + 1 = 13$ 。

Unsigned Integers(cont'd)

- 因此

$$(1101)_2 = 13_{10}$$

- 注意整數的範圍可以二進位數值表示，與使用的位元個數有關。
- 在二進位數值中，最右邊的位元通常稱為**最低有效位元 (least-significant bit, LSB)**。
- 無符號整數的最左邊位元，為2的最高次方，稱為**最高有效位元 (most-significant bit, MSB)**。
- 數位系統中通常把數個位元視為一組。一組四個位元稱為**四位字節 (nibble)**，一組八個位元稱為**位元組 (byte)**。

二、十進位和二進位系統之間的轉換

Conversion between Decimal and Binary Systems

- 假設十進位數值 $D = d_{k-1} \cdots d_1 d_0$ ，其值為 V ，

- 將其轉換成二進位數值 $B = b_{n-1} \cdots b_2 b_1 b_0$ 。因此

$$V = b_{n-1} \times 2^{n-1} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0$$

- 若將 V 除以 2，得到

$$\frac{V}{2} = b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2^1 + b_1 + \frac{b_0}{2}$$

- 新的商數為 $b_{n-1} \times 2^{n-3} + \cdots + b_2$

三、八進位與十六進位表示法 Octal Representations

- 位置數值表示法可以用在任何基數。若基數為 r ，則下列數值

$$K = k_{n-1}k_{n-2} \cdots k_1k_0$$

- 的值為

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i$$

- 以基數8表示的數值稱為八進位(octal)數值，以基數16表示的數值稱為十六進位(hexadecimal)數值。

轉換 $(857)_{10}$

				餘數	
$857 \div 2$	$=$	428		1	LSB
$428 \div 2$	$=$	214		0	
$214 \div 2$	$=$	107		0	
$107 \div 2$	$=$	53		1	
$53 \div 2$	$=$	26		1	
$26 \div 2$	$=$	13		0	
$13 \div 2$	$=$	6		1	
$6 \div 2$	$=$	3		0	
$3 \div 2$	$=$	1		1	
$1 \div 2$	$=$	0		1	MSB

結果為 $(1101011001)_2$

圖2.1 從十進位轉換成二進位

表2.1 不同系統中的數值

十進位	二進位	八進位	十六進位
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

Octal and Hexadecimal Representations (cont'd)

- 電腦的主要數值系統為二進位。使用八進位和十六進位系統的原因是它們可以作為二進位數值的速記法。
- 一個八進位數字代表三個位元。因此將二進位數值從LSB開始以三個位元為一組，替換成對應的八位元數字，就可以轉換成八進位數值。例如，101011010111之轉換如下

101	011	010	111
5	3	2	7

表示 $(101011010111)_2 = (5327)_8$ 。若位元個數不是三的倍數，則在MSB的左邊加0。例如， $(010111011)_2 = (273)_8$

Octal and Hexadecimal Representations (cont'd)

- 從八進位轉換成二進位是很直接的，只要將每個八進位數字替換成代表相同值的三個位元。
- 同樣地，十六進位數字可以用四個位元表示。例如，16位元數值可以用十六進位數字表示，

$$(1010111100100101)_2 = (AF25)_{16}$$

- 若位元個數不是四的倍數，則在MSB的左邊加0。例如， $(01101101000)_2 = (368)_{16}$ 。

Octal and Hexadecimal Representations (cont'd)

- 從十六進位轉換成二進位是很直接的，只要將每個十六進位數字替換成代表相同值的四個位元。
- 現代電腦用的二進位數值通常為32或64位元。
- 寫成n個二進位（有時稱為位元向量），這些數值對人來說很難處理，表示成8或16個數字的十六進位數值較為容易處理。

四、無符號數的加法

Addition of Unsigned Numbers

- 二進位加法和十進位加法相同，除了個別數字是0或1之外。兩個一位元數值的加法有四個可能的組合，如圖2.2所示。
- 最右邊位元稱為**總和 S (sum)**。當兩個相加的位元都是1時會進位產生最左邊位元，稱為**進位 C (carry)**。

x	0	0	1	1
$+y$	$+0$	$+1$	$+0$	$+1$
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
$c\ s$	0 0	0 1	0 1	1 0

進位 ↑ ↑ 總和

圖2.2 四種可能的情況

四、無符號數的加法

Addition of Unsigned Numbers

- 加法運算定義在圖2.3中部分的真值表。
- 兩位元加法的電路稱為**半加器(half-adder)**。
- 必須對每個位元位置做成對位元的加法，加法運算也許會包含位置I-1的**進位輸入(carry-in)**。

x y		進位 c	總和 s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

圖2.3 真值表

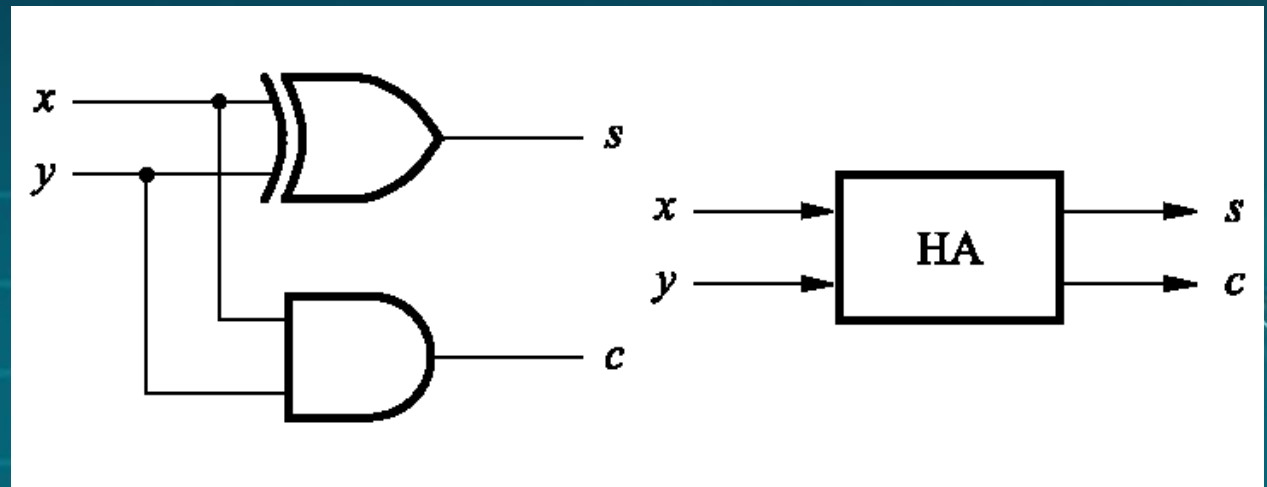


圖2.4 半加器電路圖

$X = x_4x_3x_2x_1x_0$	0 1 1 1 1	$(15)_{10}$	
$+ Y = y_4y_3y_2y_1y_0$	0 1 0 1 0	$(10)_{10}$	
	<hr/>		
	1 1 1 0		← 產生的進位
	<hr/>		
$S = s_4s_3s_2s_1s_0$	1 1 0 0 1	$(25)_{10}$	

圖2.5 加法的例子

四、無符號數的加法

Addition of Unsigned Numbers (cont'd)

- 全加器 (full-adder) 電路若 X_i 、 Y_i 和 C_i 的和為 2 或 3，則進位輸出 (carry-out) C_{i+1} 為 1。

c_i	x_i	y_i	c_{i+1}	s_i
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

圖2.6 全加器真值表

$c_i \backslash x_i y_i$				
	00	01	11	10
0		1		1
1	1		1	

$$s_i = x_i \oplus y_i \oplus c_i$$

$c_i \backslash x_i y_i$				
	00	01	11	10
0			1	
1		1	1	1

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

圖2.7 卡諾圖

四、無符號數的加法

Addition of Unsigned Numbers (cont'd)

- 對進位輸出函數而言，最佳化乘積總和實現為

$$C_{i+1} = x_i y_i + x_i C_i + y_i C_i$$

- 對 S_i 函數而言，乘積總和實現為

$$S_i = \bar{x}_i y_i \bar{C}_i + x_i \bar{y}_i \bar{C}_i + \bar{x}_i \bar{y}_i C_i + x_i y_i C_i$$

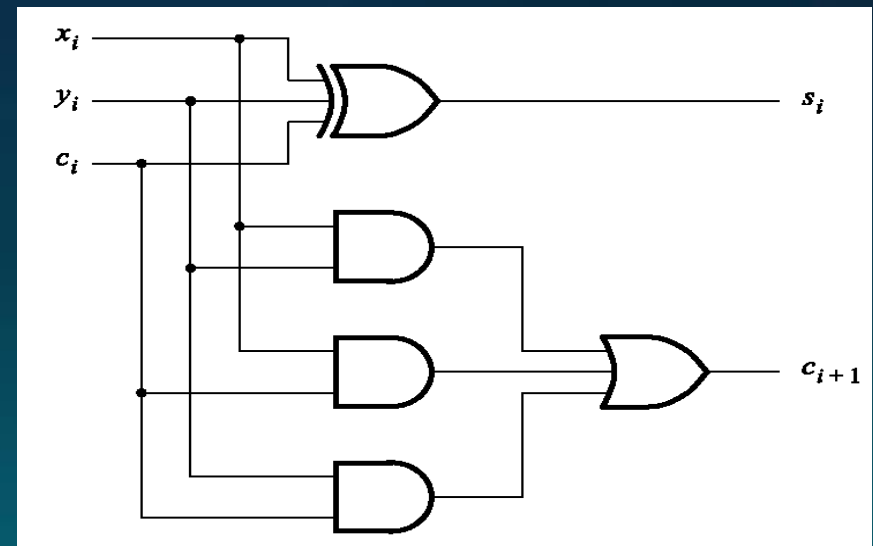


圖2.8 全加器電路圖

- 實作此函數更吸引人的方式是用XOR邏輯閘，接下來將說明之。

使用XOR邏輯閘 Use of XOR Gates

- 兩個變數的XOR函數定義為 $x_1 \oplus x_2 = \bar{x}_1x_2 + x_1\bar{x}_2$
- 之前總和位元的式子可以改成只用XOR運算，如下

$$s_i = x_i \oplus y_i \oplus c_i$$

- XOR運算具有結合性，因此可以寫成

$$\begin{aligned} s_i &= (\bar{x}_iy_i + x_i\bar{y}_i)\bar{c}_i + (\bar{x}_i\bar{y}_i + x_iy_i)c_i \\ &= (x_i \oplus y_i)\bar{c}_i + \overline{(x_i \oplus y_i)}c_i \\ &= (x_i \oplus y_i) \oplus c_i \end{aligned}$$

- 因此可以用一個三輸入XOR邏輯閘來實現 S_i 。

使用XOR邏輯閘 Use of XOR Gates(cont'd)

- 觀察發現XOR沒有可以合併成更大乘積項的小項，從圖2.7卡諾圖中函數 s_i 的棋盤狀可以明顯看出來。實作圖2.6真值表的邏輯電路在圖2.8。此電路稱為**全加器(full-adder)**。
- 根據XOR的定義 $x_i \oplus y_i = \bar{x}_i y_i + x_i \bar{y}_i$ 可以清楚地看出來。將 x 視為控制輸入。若 $x=0$ ，則輸出為 y 值。
- 在圖2.6真值表的上半部， c_i 等於0，總和函數 s_i 為 x_i 與 y_i 的XOR。在真值表的下半部，等於1，為上半部的互補。
- 這個運算相當常用，稱為**XNOR**。我們通常用特殊符號來表示XNOR運算，換句話說， $x \odot y = \overline{x \oplus y}$
- XNOR有時候也稱為**重和運算**，因為當輸入值相同時輸出為1。

五、分解的全加器

Decomposed Full-Adder

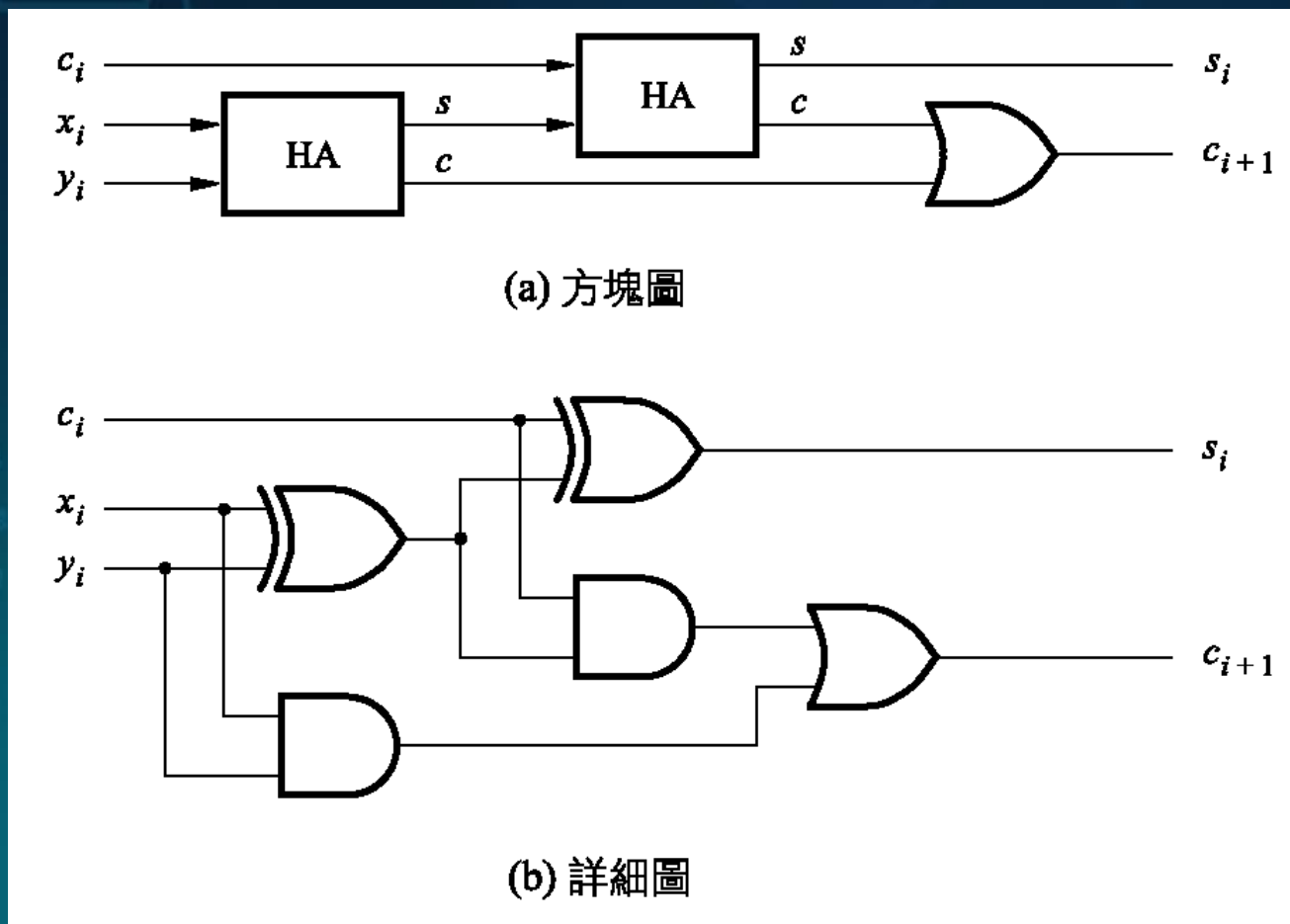


圖2.9 全加器電路的分解實作

六、漣波進位加法器 Ripple Carry Adder

- 每個位元位置使用全加器電路，連接如圖2.10所示。
- 當運算元 X 和 Y 都套用在加法器的輸入端，需要一點時間才會產生有效的輸出總和 S 。
- 因為進位訊號以類似漣波的方式穿過各級全加器，圖2.10的電路稱為漣波進位加法器(ripple-carry adder)。

六、漣波進位加法器 Ripple Carry Adder (cont'd)

- 根據圖2.10，我們通常用此圖形符號來代表加法器，字母 x_i 、 y_i 、 s_i 和 c_i 代表輸入和輸出。

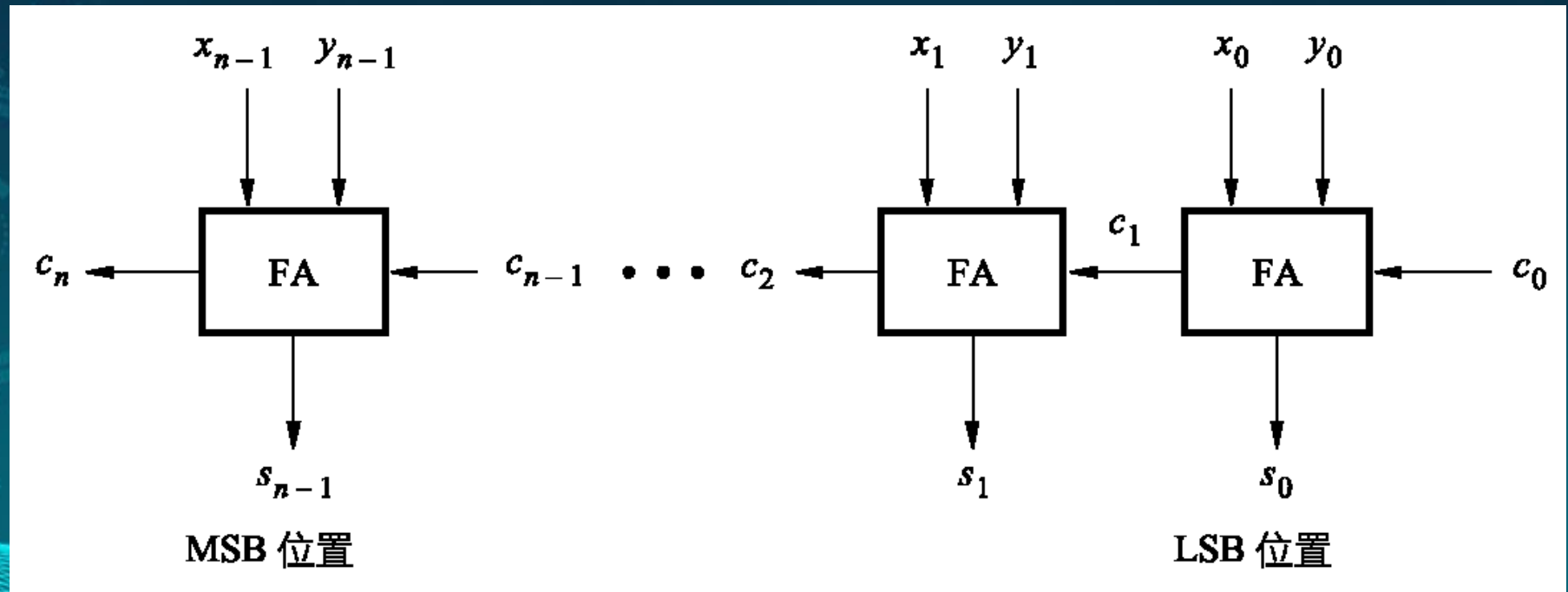
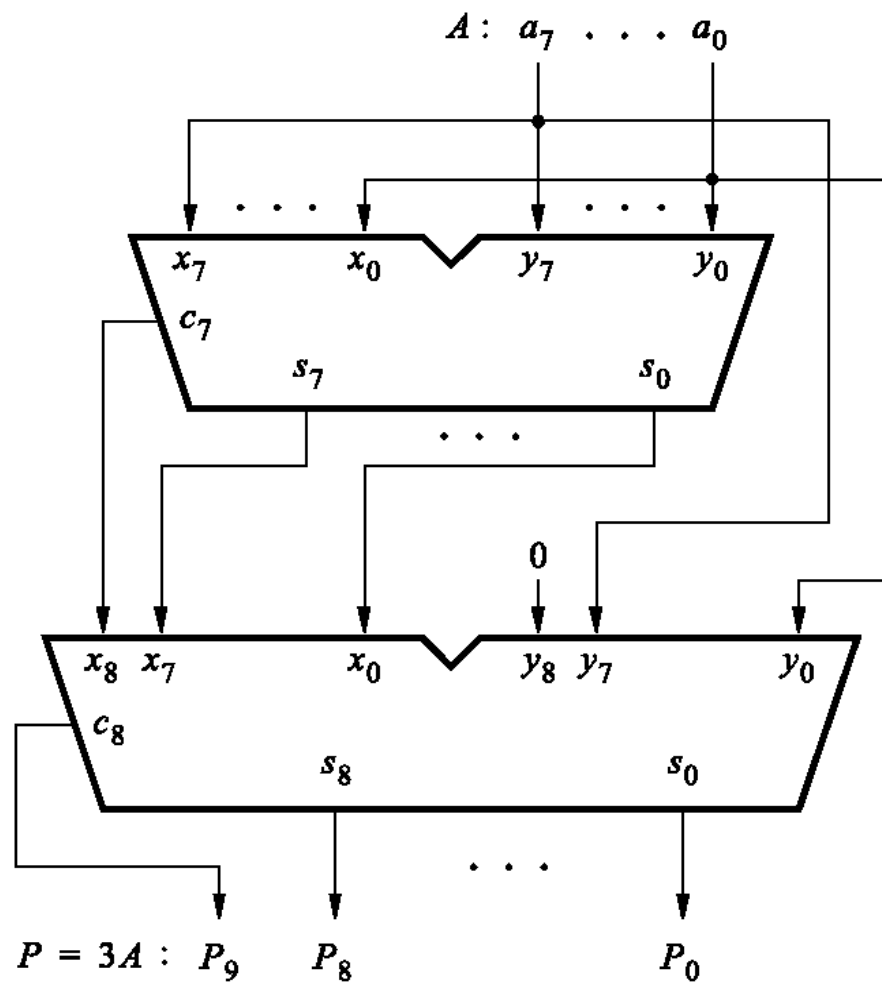


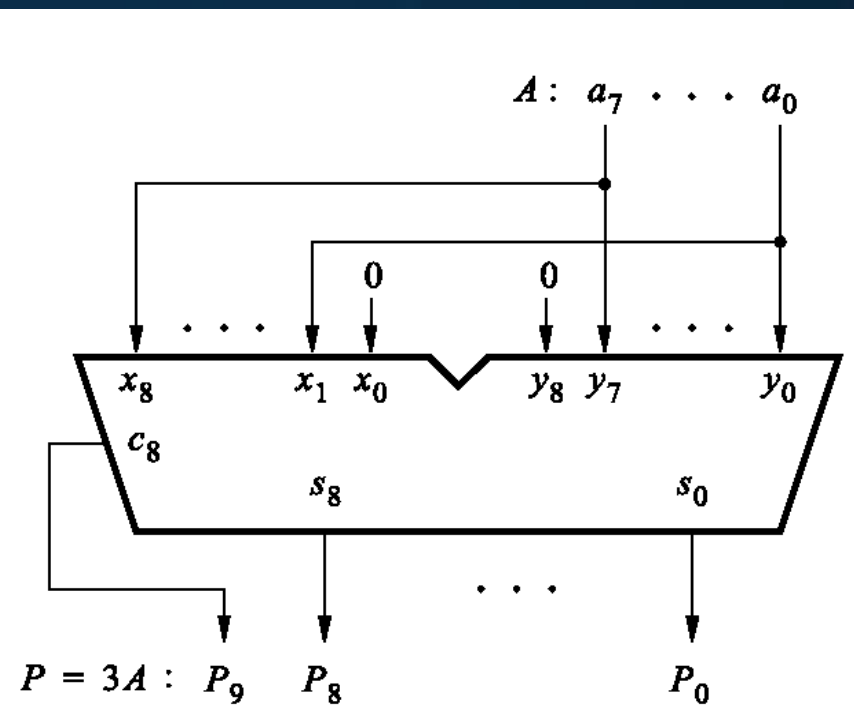
圖2.10 n 位元漣波進位加法器

七、設計範例 Design Example

- 有個簡單的方法可以設計此電路，用兩個漣波進位加法器將三個數字A相加，如圖2.11(a)所示。
- 第一個加法器產生 $A+A=2A$ 。其結果為八個總和位元和來自MSB的進位。第二個加法器產生 $2A+A=3A$ 。
- 這必須是九位元加法器，才能處理第一級產生的九位元 $2A$ 。因為輸入 y_i 只被八位元 A 驅動，所以第九個輸入 y_8 連接到0。
- 只需要一個漣波進位加法器來實作 $3A$ ，如圖2.11(b)所示。



(a) 原始方法

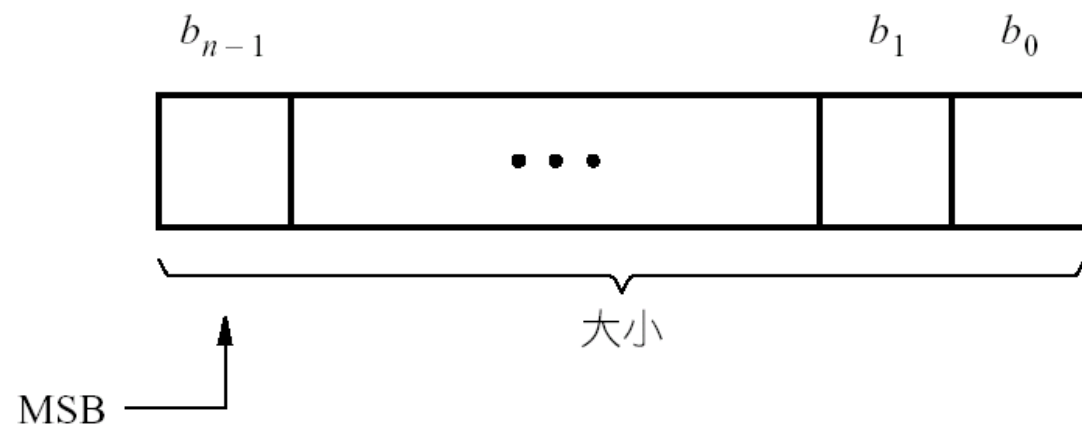


(b) 有效率的設計

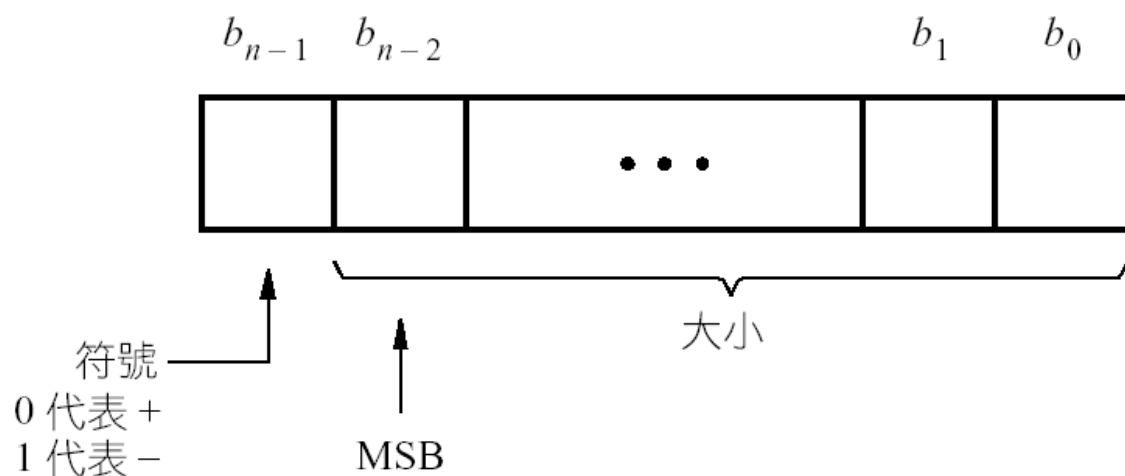
圖2.11 八位元無符號數乘以3的電路

八、帶正負號數 Signed Numbers

- 在十進位系統中，數值的符號以**MSB左邊的+或-**符號表示。
- 在二進位系統中，數值的符號是以**最左邊位元**表示。**正數的最左邊位元為0**，**負數的最左邊位元為1**。
- 因此，帶正負號數的最左邊位元代表其**符號(sign)**，剩下的 $n-1$ 位元代表其**大小(magnitude)**，如圖2.12所示。特別注意MSB位置造成的差異。
- 對無符號數而言，**所有的位元都代表數值的大小**，因此所有 n 個位元在定義大小時都是**有效的(significant)**，所以MSB為最左邊的位元 b_{n-1} 。
- 對帶正負號數而言有 $n-1$ 個有效的位元，所以MSB的位元位置為 b_{n-2} 。



(a) 無符號數



(b) 帶正負號數

圖2.12 整數表示法的格式

九、負數 Negative Numbers

- 負數可以三種不同方式表示：
- 符號與大小
- 一的補數
- 二的補數

符號與大小表示法

Sign-and-Magnitude Representation

- 符號可以區別此數為正或負。這方法稱為符號與大小(sign-and-magnitude)數值表示法。
- 同樣的方法也可以用在二進位數值，符號位元對正數和負數分別為0和1。
- 例如，若使用四位元數值，則 $+5 = 0101$ 且 $-5 = 1101$ 。
- 因為和十進位符號與大小數值很類似，這種表示法很容易理解。

一的補數表示法

1's Complement Representation

- 在補數系統中，負數的定義是根據正數的減法運算。我們考慮兩種二進位數值的方法：一的補數和二的補數。
- 在一的補數 (1's complement) 方法中，欲得到 n 位元負數 K ，可用 $2^n - 1$ 減去等效正數 P ；也就是 $K = (2^n - 1) - P$ 。

二的補數表示法

2's Complement Representation

- 在二的補數方法中，欲得到負數 K ，可用 2^n 減去等效正數 P ；也就是 $K=2^n-P$ 。
- 若 K_1 為 P 的一的補數，且 K_2 為 P 的二的補數，則

$$K_1 = (2^n - 1) - P$$

$$K_2 = 2^n - P$$

- 因此 $K_2 = K_1 + 1$ 。

求出二的補數的規則

Rules for Finding 2's Complements

- 表2.2說明四位元帶正負號數表示法中所有16個四位元型樣(pattern)。
- 使用二的補數表示法， n 位元數值 $B = b_{n-1}b_{n-2} \cdots b_1b_0$ 代表下列值

$$V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

- 因此最大負數 $100 \cdots 00$ 之值為 -2^{n-1} 。
- 最大正數 $011 \cdots 11$ 值為 $2^{n-1} - 1$ 。

表2.2 四位元帶正負號整數的說明。

$b_3b_2b_1b_0$	符號與大小	1的補數	2的補數
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

十、加法和減法

Addition and Subtraction

- 為了評估不同數值表示法的適用性，有必要研究如何用於數學運算——特別是加法和減法。
- 我們用的是四位元數值，由一個符號位元和三個有效(significant)位元組成。
- 因此這些數值必須夠小，使其總和大小可以三位元表示，也就是總和不能超過7。
- 正數的加法對三種數值表示法來說都相同。

符號與大小加法

Sign-and-Magnitude Addition

- 若兩個運算元符號都相同，則符號與大小加法很簡單。將大小相加，所得總和冠上運算元的符號。
- 兩個運算元的符號相反，這項工作就變得更複雜。
 - 必須用較大的數值減去較小的數值，這表示需要比較和相減的邏輯電路。
 - 我們將簡要說明可以執行減法但不需要這些電路。因此，符號與大小表示法不適用於電腦。

一的補數加法


1's Complement Addition

- 圖2.13顯示兩個數值相加，根據不同的符號組合，有四種情況。
- 如圖左半部所示， $5 + 2 = 7$ 和 $(-5) + 2 = (-3)$ 的計算是很直觀的，運算元的簡單加法就可以得到正確結果。
- 另外兩種情況就不是如此，計算 $5 + (-2) = 3$ 產生位元向量10010。在符號位元的位置有進位輸出，將其加到LSB位置以校正結果，如圖2.13所示。

$$\begin{array}{r}
 (+5) \quad 0101 \\
 + (+2) \quad +0010 \\
 \hline
 (+7) \quad 0111
 \end{array}$$


$$\begin{array}{r}
 (-5) \quad 1010 \\
 + (+2) \quad +0010 \\
 \hline
 (-3) \quad 1100
 \end{array}$$

$$\begin{array}{r}
 (+5) \quad 0101 \\
 + (-2) \quad +1101 \\
 \hline
 (+3) \quad 10010
 \end{array}$$



$$\begin{array}{r}
 0011
 \end{array}$$

$$\begin{array}{r}
 (-5) \quad 1010 \\
 + (-2) \quad +1101 \\
 \hline
 (-7) \quad 10111
 \end{array}$$



$$\begin{array}{r}
 1000
 \end{array}$$

圖2.13 一的補數加法的例子

二的補數加法

2's Complement Addition

- 圖2.14顯示如何以二的補數執行加法。

$\begin{array}{r} (+5) \\ + (+2) \\ \hline (+7) \end{array}$	$\begin{array}{r} 0101 \\ + 0010 \\ \hline 0111 \end{array}$	$\begin{array}{r} (-5) \\ + (+2) \\ \hline (-3) \end{array}$	$\begin{array}{r} 1011 \\ + 0010 \\ \hline 1101 \end{array}$
$\begin{array}{r} (+5) \\ + (-2) \\ \hline (+3) \end{array}$	$\begin{array}{r} 0101 \\ + 1110 \\ \hline 10011 \end{array}$ <div>↑ 忽略</div>	$\begin{array}{r} (-5) \\ + (-2) \\ \hline (-7) \end{array}$	$\begin{array}{r} 1011 \\ + 1110 \\ \hline 11001 \end{array}$ <div>↑ 忽略</div>

圖2.14 二的補數加法的例子

二的補數減法

2's Complement Subtraction

- 圖2.15顯示如何以二的補數執行減法。

$\begin{array}{r} (+5) \\ - (+2) \\ \hline (+3) \end{array}$	$\begin{array}{r} 0101 \\ - 0010 \\ \hline \end{array}$	\Rightarrow	$\begin{array}{r} 0101 \\ + 1110 \\ \hline 10011 \end{array}$	$\begin{array}{r} (+5) \\ - (-2) \\ \hline (+7) \end{array}$	$\begin{array}{r} 0101 \\ - 1110 \\ \hline \end{array}$	\Rightarrow	$\begin{array}{r} 0101 \\ + 0010 \\ \hline 0111 \end{array}$
			\uparrow 忽略				
$\begin{array}{r} (-5) \\ - (+2) \\ \hline (-7) \end{array}$	$\begin{array}{r} 1011 \\ - 0010 \\ \hline \end{array}$	\Rightarrow	$\begin{array}{r} 1011 \\ + 1110 \\ \hline 11001 \end{array}$	$\begin{array}{r} (-5) \\ - (-2) \\ \hline (-3) \end{array}$	$\begin{array}{r} 1011 \\ - 1110 \\ \hline \end{array}$	\Rightarrow	$\begin{array}{r} 1011 \\ + 0010 \\ \hline 1101 \end{array}$
			\uparrow 忽略				

圖2.15 二的補數減法的例子

- 我們以圖形視覺化輔助圖2.14和2.15的加法與減法，將所有可能的四位元型樣都放在16模數(modulo-16)的圓圈上，如圖2.16所示。

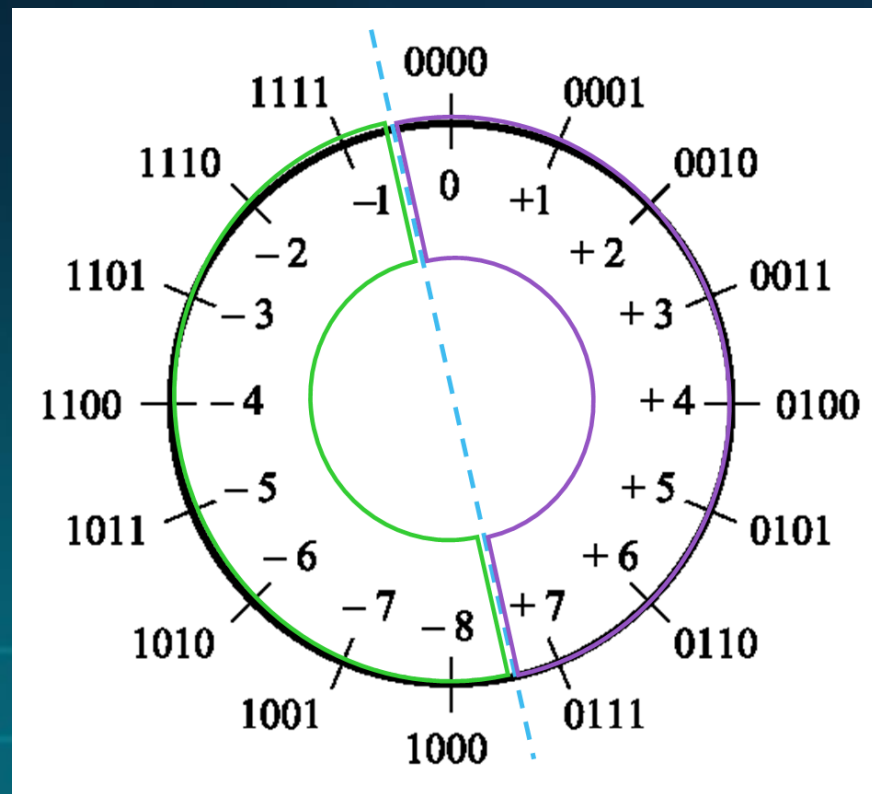


圖2.16 四位元二的補數的圖形說明

十一、加法器和減法器單元

Adder and Subtractor Unit

- 執行加法和減法唯一的不同是，減法必須用一個運算元為二的補數。
- 欲在LSB位置加1，只需設定進位輸入位元 c_0 為1。
- 合併的加法器／減法器單元是設計邏輯電路中重要觀念的極佳範例。
 1. 設計電路越有彈性越好，並且要盡量共用電路完成許多工作。
 2. 這方法可以縮小實作電路所需的邏輯閘個數，因此大大地減低接線複雜度。

十二、算術溢位 Arithmetic Overflow

- 若以 n 位元表示帶正負號數，則結果必須在範圍 -2^{n-1} 到 $2^{n-1}-1$ 。若結果不在這範圍內，則稱為發生算術溢位 (arithmetic overflow)。
- 大小為7和2的二的補數，相加的四種情況如圖2.17所示。
- 圖中顯示當進位輸出有不同值時會發生溢位，當有相同值時則產生正確的總和。

- 範圍 -2^{n-1} 到 $2^{n-1}-1 = -8$ 到 7

$$\begin{array}{r}
 (+7) \quad 0111 \\
 + (+2) \quad + 0010 \\
 \hline
 (+9) \quad 1001
 \end{array}$$

$c_4 = 0$ 溢位
 $c_3 = 1$

$$\begin{array}{r}
 (-7) \quad 1001 \\
 + (+2) \quad + 0010 \\
 \hline
 (-5) \quad 1011
 \end{array}$$

$c_4 = 0$ 正確
 $c_3 = 0$

$$\begin{array}{r}
 (+7) \quad 0111 \\
 + (-2) \quad + 1110 \\
 \hline
 (+5) \quad 10101
 \end{array}$$

$c_4 = 1$ 正確
 $c_3 = 1$

$$\begin{array}{r}
 (-7) \quad 1001 \\
 + (-2) \quad + 1110 \\
 \hline
 (-9) \quad 10111
 \end{array}$$

$c_4 = 1$ 溢位
 $c_3 = 0$

圖2.17 決定溢位的例子

十二、算術溢位 Arithmetic Overflow (cont'd)

- 我們做個快速的確認，考慮圖2.14的例子，這些數值夠小使得任何情況都不會發生溢位。在圖上方的兩個例子，符號sign和MSB位置都有進位輸出0。在圖下方的兩個例子，符號和MSB位置都有進位輸出1。因此，圖2.14和2.17的例子，可以以下列方式偵測溢位的發生

$$\begin{aligned}\text{溢位} &= c_3 \bar{c}_4 + \bar{c}_3 c_4 \\ &= c_3 \oplus c_4\end{aligned}$$

- 對n位元數值而言 $\text{溢位} = c_3 \oplus c_4$

十三、效能考量 Performance Issues

- 常用來評估系統價值的指標為價格效能比值 (price/performance ratio) ↓ 愈好。
- 數值的加法和減法是常執行的基本運算。
- 這些運算執行的速度對電腦整體效能有很大的影響。
- 任何電路的速度都被經過電路的最長延遲所限制。
- 最長延遲通常稱為關鍵路徑延遲 (critical-path delay)，造成此延遲的路徑稱為關鍵路徑 (critical path)。

十四、快速加法器 Fast Adders

- 大型數位系統的效能與各種函數單元的電路速度有關。很明顯地，使用較快的電路可以得到較好的效能。
- 可以用較優秀(通常是較新的)技術以降低邏輯閘的延遲，以達成此目的。
- 但是也可以改變函數單元的整體架構來達成此目的，這通常會有更明顯的改善。

十五、進位預看加法器 Carry-Lookahead Adder

- 為了減少進位傳遞經過漣波進位加法器造成的延遲，我們對每一級加法器作快速評估，不管來自前一級的進位輸入是0或1。若可以在很短的時間內做出正確評估，則整個加法器的效能就會改善。

- 根據全加器卡諾圖，第 i 級的進位輸出函數為

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

- 若將此式因式分解，得到 $c_{i+1} = x_i y_i + (x_i + y_i) c_i$
- 可以寫成 $c_{i+1} = g_i + p_i c_i$

十五、進位預看加法器

Carry-Lookahead Adder (cont'd)

- 其中 $g_i = x_i y_i$
 $p_i = x_i + y_i$
- 當輸入 x_i 和 y_i 皆為1時函數 g_i 為1，與輸入到此級的進位 c_i 無關。此例中第 i 級保證產生進位輸出，所以 g 稱為產生(generate)函數。
- 當輸入 x_i 和 y_i 至少有一個為1時函數 p_i 為1。此例中若 $c_i=1$ 則產生進位輸出。這是因為進位輸入1傳遞經過第 i 級，因此 p_i 稱為傳遞(propagate)函數。
- 以第 $i-1$ 級擴充3.3式，得到

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

十五、進位預看加法器

Carry-Lookahead Adder (cont'd)

- 對其他級作相同的擴充，直到第0級，得到

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0$$

- 此式代表一個二級AND-OR電路，其 c_{i+1} 的評估非常快速。以此式為基礎的加法器稱為進位預看加法器(carry-lookahead adder)。
- 為了瞭解上式的物理意義，我們考慮以其建構快速加法器和漣波進位加法器的不同效應。
- 圖2.18為漣波進位加法器的前兩級，其進位輸出函數實作如前式。
- 圖2.18中關鍵路徑從輸入 x_0 和 y_0 到輸出 c_2 。它經過五個邏輯閘，以藍色表示。
- 圖2.19為進位預看加法器的前兩級，以上式實作進位輸出函數。因此

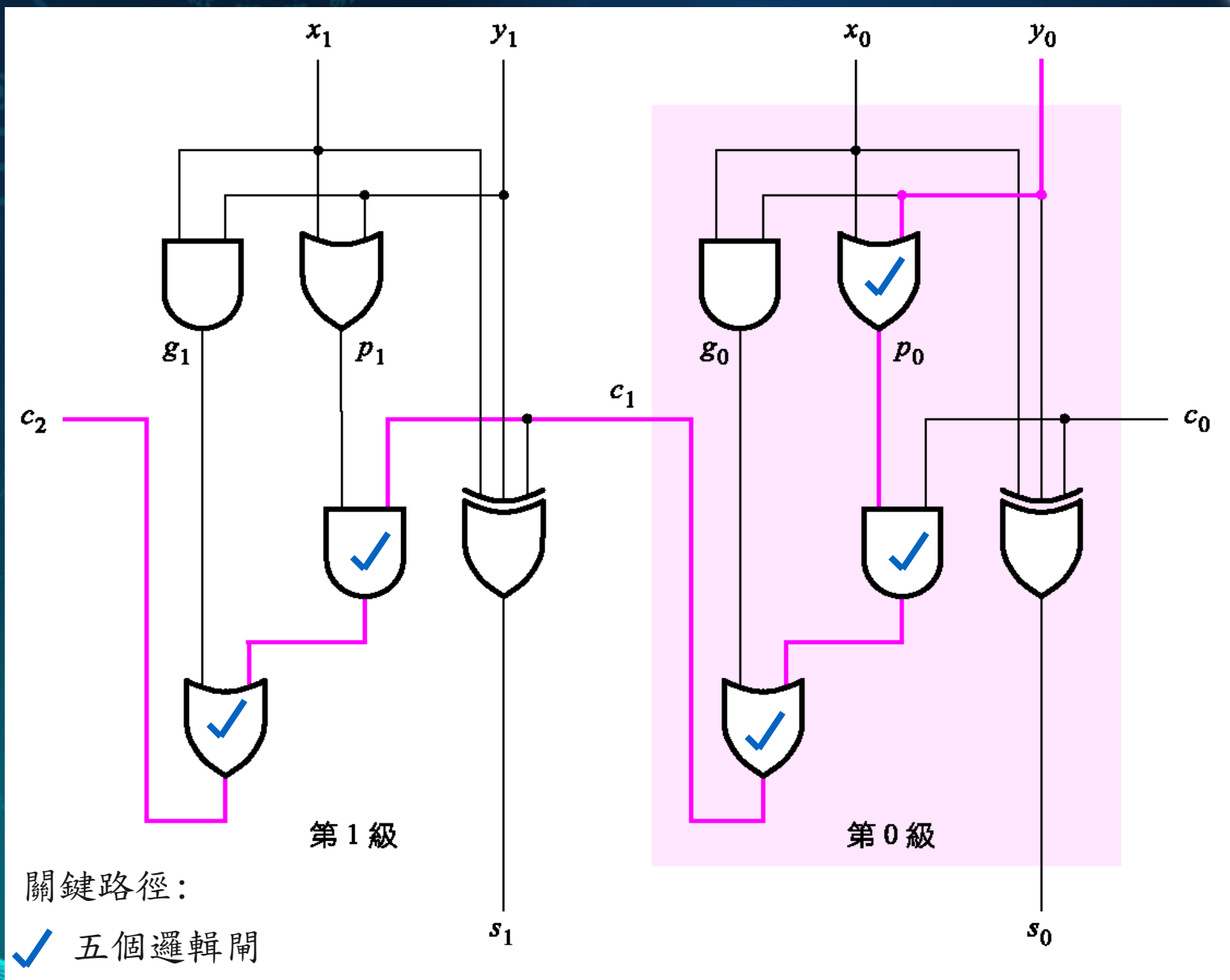


圖2.18 根據前式的漣波進位加法器

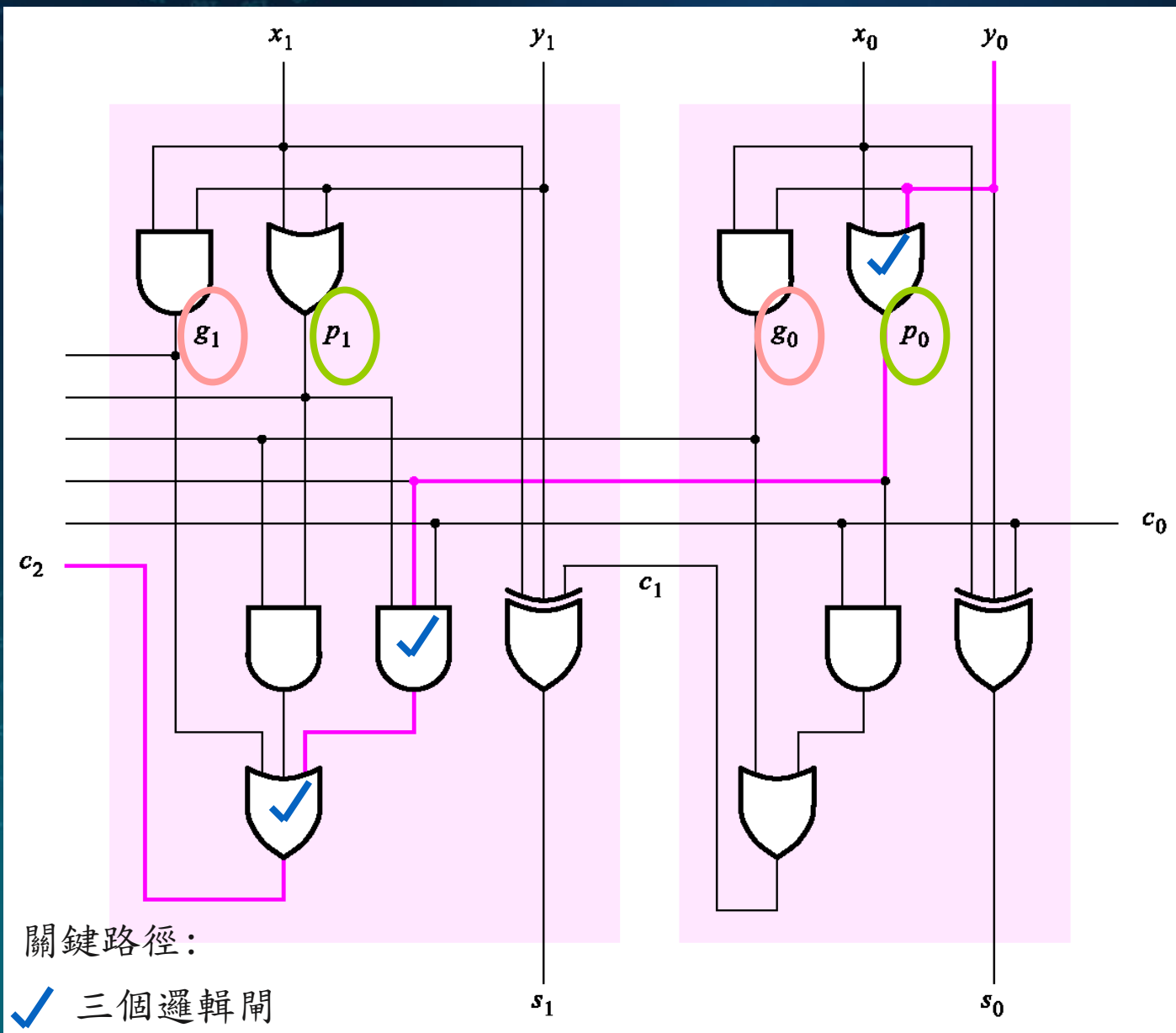


圖2.19 進位預看加法器的前兩級

十五、進位預看加法器

Carry-Lookahead Adder (cont'd)

- 隨著 n 變大， n 位元進位預看加法器的複雜度也隨著快速增加。為了降低其複雜度，我們可以階層性(hierarchical)方法設計大型加法器。
- 當每個區塊內進位預看時，進位漣波穿過各區塊間。此電路如圖2.20所示。

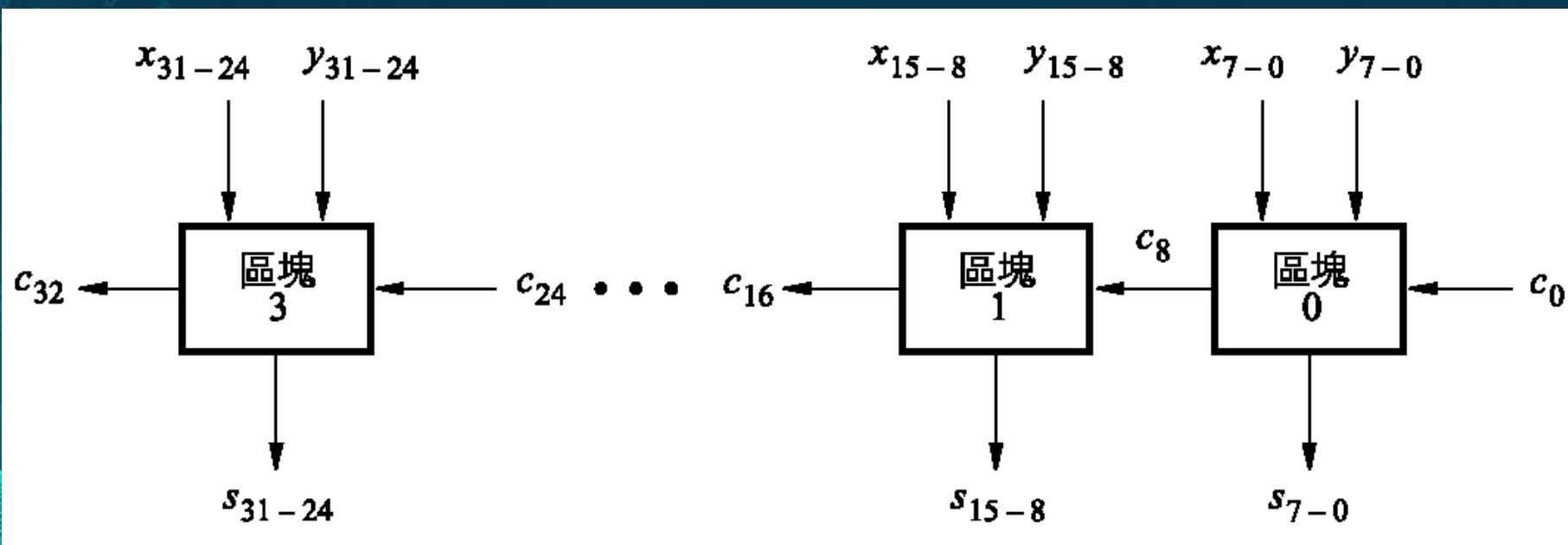


圖2.20 階層性進位預看加法器，各區塊間為漣波進位

十五、進位預看加法器

Carry-Lookahead Adder (cont'd)

- 若不在各區塊間使用進位漣波方法，設計第二級進位預看加法器可以快速產生各區塊間的進位訊號，得到較快的電路。這種『階層性進位預看加法器』如圖2.21所示。

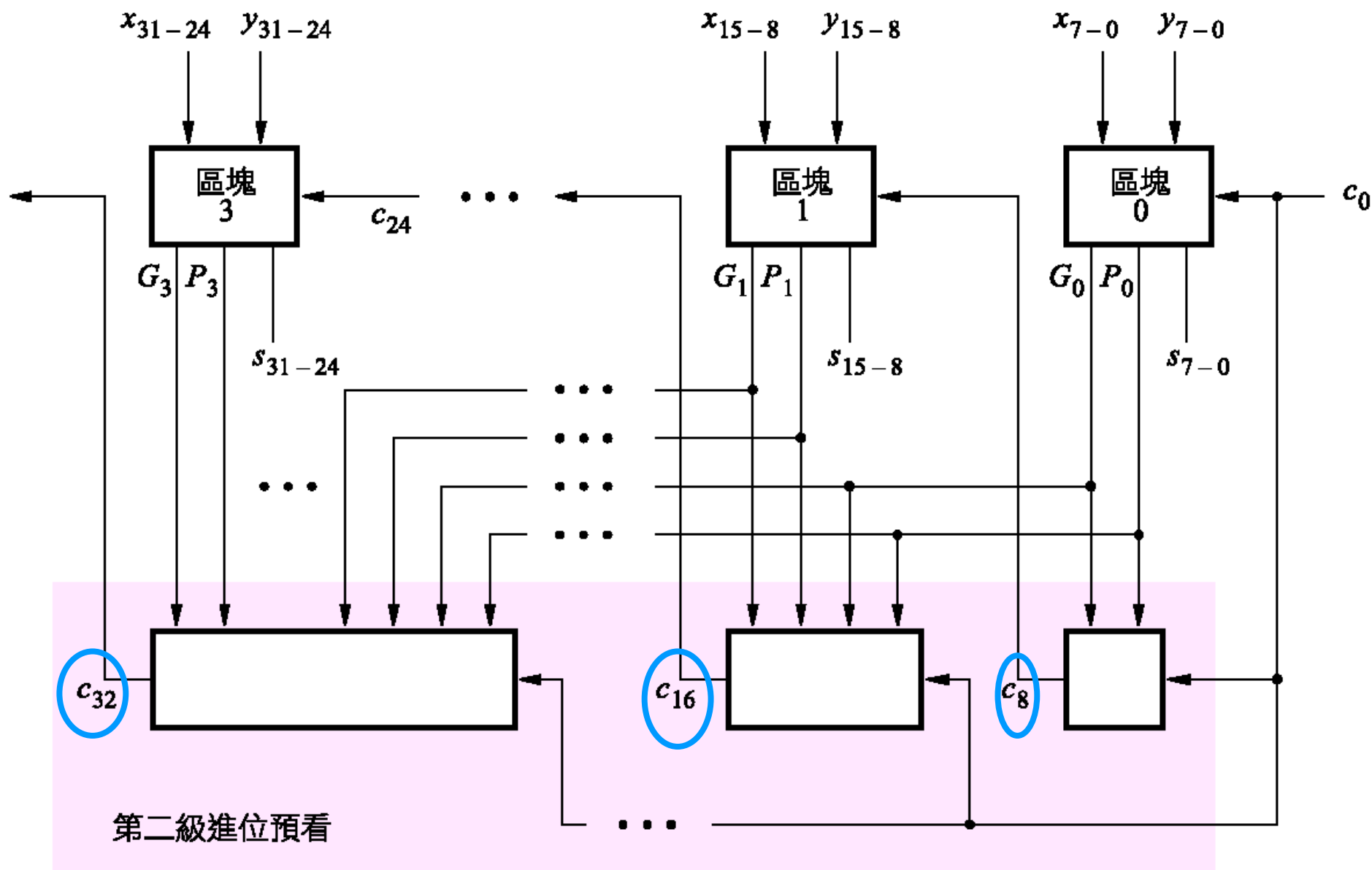


圖2.21 階層性進位預看加法器

十五、進位預看加法器 Carry-Lookahead Adder (cont'd)

- 為了得到區塊0的產生和傳遞訊號，我們檢視 c_8 的表示式

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\ + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

- 此式的最後一項表示，若所有八個傳遞函數皆為1，則進位輸入 c_0 被傳遞經過整個區塊。因此 $P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$

- 此式中剩下的項表示產生進位輸出的所有其他情況。因此

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \cdots + p_7p_6p_5p_4p_3p_2p_1g_0$$

- 在階層性加法器中 c_8 的表示式為 $c_8 = G_0 + P_0c_0$

十五、進位預看加法器

Carry-Lookahead Adder (cont'd)

- 對區塊1而言 G_1 和 P_1 的表示式和 G_0 和 P_0 的表示式相同，只是每個次標 i 換成了 $i+8$ 。 G_2 、 P_2 、 G_3 和 P_3 都可以相同方式得到。區塊1的進位輸出 c_{16} 的表示式為

$$\begin{aligned}c_{16} &= G_1 + P_1 c_8 \\ &= G_1 + P_1 G_0 + P_1 P_0 c_0\end{aligned}$$

- 類似地， c_{24} 和 c_{32} 的表示式為

$$\begin{aligned}c_{24} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0 \\ c_{32} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0\end{aligned}$$

- 以此方式，還需要兩個邏輯閘延遲以產生進位訊號 c_8 、 c_{16} 和 c_{24} ，而不需要產生 G_j 和 P_j 函數的時間。

技術考量

Technology Considerations

- 用來實作邏輯閘的技術將扇入限制在很少的輸入個數。因此必須將扇入限制的事實納入考量。為說明此問題，考慮前八個進位的表示式：

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

⋮

$$\begin{aligned} c_8 = & g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 \\ & + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0 \end{aligned}$$

- 為了滿足扇入限制， c_8 的式子可以重新寫成

$$\begin{aligned} c_8 = & (g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4) \\ & + [(p_7 p_6 p_5 p_4)(g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0)] \\ & + (p_7 p_6 p_5 p_4)(p_3 p_2 p_1 p_0) c_0 \end{aligned}$$

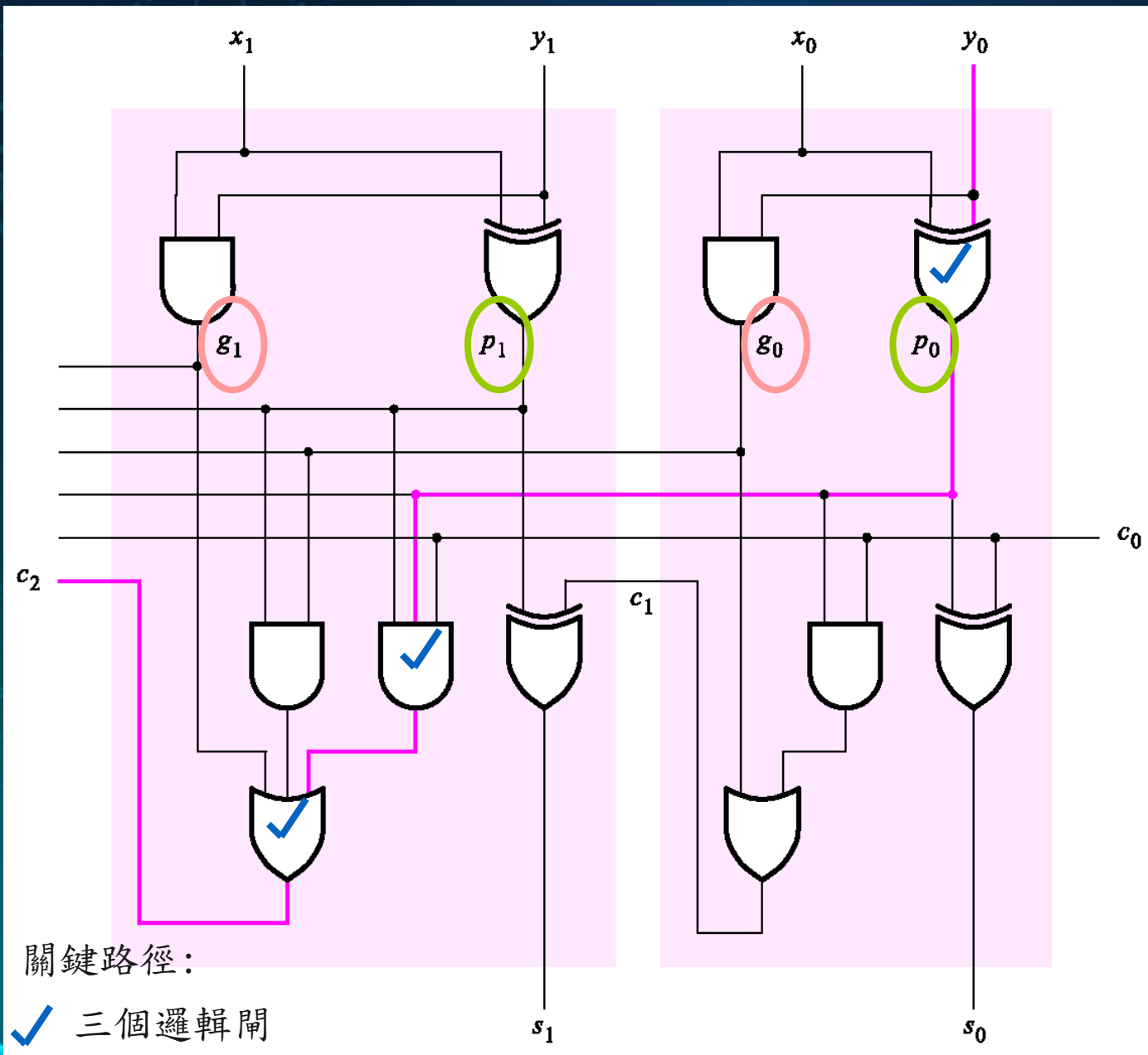
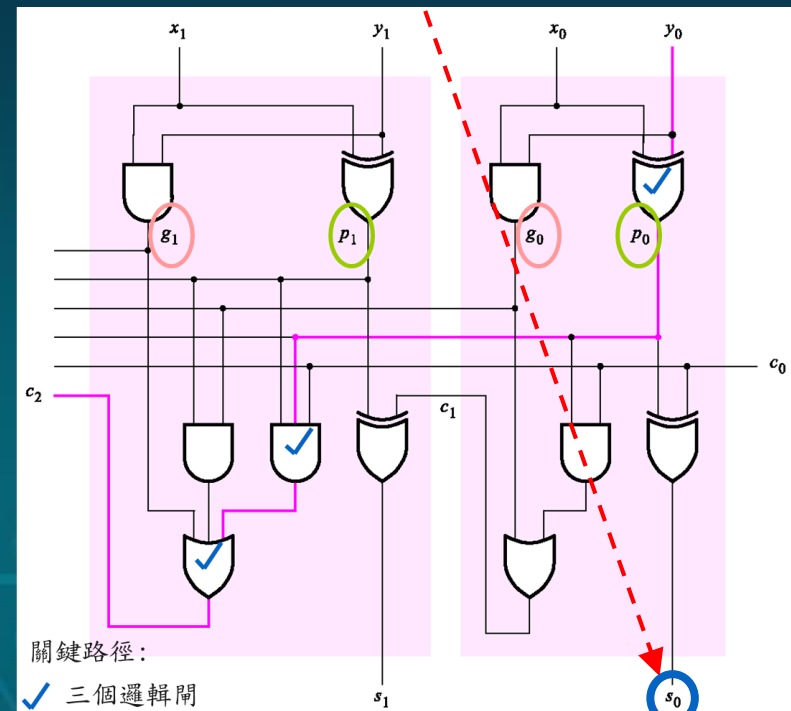
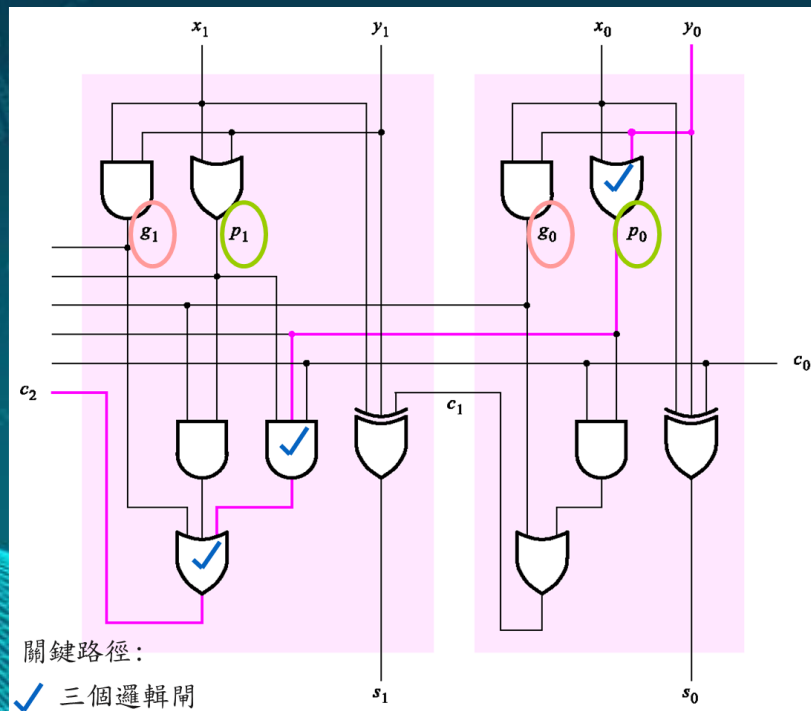


圖2.22 進位預看加法器的另一種設計

- 圖2.19和2.22電路需要相同個數的邏輯閘。三輸入XOR函數可以CPLD中的巨集實現，使用乘積總和表示式

$$s_i = x_i \bar{y}_i \bar{c}_i + \bar{x}_i y_i \bar{c}_i + \bar{x}_i \bar{y}_i c_i + x_i y_i c_i$$

- 因為這個巨集允許四個乘積項的實作。
- 進位預看加法器是眾所周知的觀念。有些標準晶片就實作進位預看加法器電路的一部份，稱為進位預看產生器 (carry-lookahead generator)。



十六、以CAD工具設計算術電路

Design of Arithmetic Circuits Using CAD Tools

- 本節將說明如何使用**CAD工具**設計算術電路。
- 有兩種不同的設計方法：**圖示擷取**和**VHDL程式碼**。
- 這裡只討論**VHDL程式碼**設計方法。

十七、以VHDL設計算術電路

Design Arithmetic Circuits Using VHDL

- 建立 n 位元加法器很明顯的方法，是畫出包含 n 個全加器的階層性電路圖。這方法也可以用VHDL完成，首先建立一個全加器的VHDL模組，然後定義較高階的模組，使用 n 個全加器實體(instance)。
- 以VHDL指定此電路的一種方法為使用邏輯閘階層基本元件，如圖2.23所示。

```
LIBRARY ieee ;  
USE ieee.std_logic_1164.all ;
```

```
ENTITY fulladd IS  
    PORT (    Cin, x, y    : IN        STD_LOGIC ;  
           s, Cout : OUT    STD_LOGIC ) ;  
END fulladd ;
```

```
ARCHITECTURE LogicFunc OF fulladd IS  
BEGIN  
    s <= x XOR y XOR Cin ;  
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;  
END LogicFunc ;
```

圖2.23 全加器的VHDL程式碼

- VHDL 允許將這些敘述合併在單一敘述中，如圖 2.24 所示。

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT (    Cin           : IN          STD_LOGIC ;
             x3, x2, x1, x0 : IN          STD_LOGIC ;
             y3, y2, y1, y0 : IN          STD_LOGIC ;
             s3, s2, s1, s0 : OUT         STD_LOGIC ;
             Cout           : OUT         STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT (    Cin, x, y : IN          STD_LOGIC ;
                 s, Cout   : OUT         STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

圖 2.24 四位元加法器的 VHDL 程式碼

十七、以VHDL設計算術電路

Design Arithmetic Circuits Using VHDL

- 另一種使用函數表示式的方法如圖2.25所示。

```
LIBRARY ieee ;  
USE ieee.std_logic_1164.all ;  
  
PACKAGE fulladd_package IS  
  COMPONENT fulladd  
    PORT ( Cin, x, y : IN STD_LOGIC ;  
          s, Cout   : OUT  STD_LOGIC ) ;  
  END COMPONENT ;  
END fulladd_package ;
```

圖2.25 包裝的宣告

- XOR運算以^符號表示。同樣的，我們可以將兩個連續指定敘述合併成單一敘述，如圖2.26所示。

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;

ENTITY adder4 IS
    PORT (    Cin           : IN      STD_LOGIC ;
             x3, x2, x1, x0  : IN      STD_LOGIC ;
             y3, y2, y1, y0  : IN      STD_LOGIC ;
             s3, s2, s1, s0  : OUT     STD_LOGIC ;
             Cout            : OUT     STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

圖2.26 另一種指定四位元加法器的方法

十七、以VHDL設計算術電路

Design Arithmetic Circuits Using VHDL (cont'd)

- 圖2.27的四位元加法器是由四個實體(instantiation)敘述所描述。
- 接下來是實體名稱(instance name)，可以是任何合法的VHDL名稱。實體名稱必須是唯一的。
- 加法器的最低有效級(least-significant stage)稱為第0級(stage 0)，最高有效級(most-significant stage)為第3級(stage 3)。
- 可以將fulladd模組包含在和adder4模組相同的VHDL程式碼檔案，如圖2.27所示，但也可以包含另一個檔案。

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd package.all ;

ENTITY adder4 IS
    PORT ( Cin      : IN      STD_LOGIC ;
           X, Y : IN  STD_LOGIC_VECTOR(3 DOWNT0 0) ;
           S       : OUT     STD_LOGIC_VECTOR(3 DOWNT0 0) ;
           Cout    : OUT     STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
    stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
    stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
    stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

圖2.27 以多位元訊號定義四位元加法器

十九、算術指定敘述

(直接使用算術運算子'+', 不用自行設計加法器)

- 若定義下列訊號

SIGNAL X,Y,S :STD_LOGIC_VECTOR(15 DOWNTO 0);

則下列算術指定敘述代表16位元加法器，如圖2.28。

$$S \leq X + Y;$$

- 除了+運算子用於加法之外，VHDL還提供其他算術運算子，列在附錄A的表A.1。
- 邏輯合成演算法可以對不同的目的產生不同的電路，例如成本或速度最佳化。
- 在VHDL中，&運算子稱為串接(concatenate)運算子。

Sum <= ('0' & X) + Y + Cin **'0' + X -> S : 16 + 1 = 17 Bits**

```
LIBRARY ieee ;  
USE ieee.std_logic_1164.all ;  
USE ieee.std_logic_signed.all ;
```

```
ENTITY adder16 IS  
    PORT ( X, Y      : IN      STD_LOGIC_VECTOR(15 DOWNTO 0) ;  
           S          : OUT     STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;  
END adder16 ;
```

```
ARCHITECTURE Behavior OF adder16 IS  
BEGIN  
    S <= X + Y ;  
END Behavior ;
```

圖2.28 16位元加法器的 VHDL 程式碼

十九、算術指定敘述 (直接使用算術運算子'+', 不用自行設計加法器)

- 圖2.29的程式碼使用 *std_logic_signed* 包裝，允許STD_LOGIC訊號用於算術運算子。
- 事實上*std_logic_signed*包裝使用另一個包裝*std_logic_arith*。此包裝定義兩個資料類型SIGNED和UNSIGNED，用於算術電路以處理帶正負數和無符號數。
- 圖2.29程式碼可以直接寫成使用 *std_logic_arith* 包裝，如圖2.30所示。
- 我們可以選擇使用*std_logic_signed*包裝和STD_LOGIC_VECTOR訊號，如圖2.29，或是使用*std_logic_arith*包裝和SIGNED訊號，如圖2.30。

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY adder16 IS
    PORT (    Cin           : IN      STD_LOGIC ;
            X, Y           : IN      STD_LOGIC_VECTOR(15 DOWNT0 0) ;
            S              : OUT     STD_LOGIC_VECTOR(15 DOWNT0 0) ;
            Cout, Overflow : OUT     STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : STD_LOGIC_VECTOR(16 DOWNT0 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin; '0' + X -> S : 16 +1 = 17 Bits
    S <= Sum(15 DOWNT0 0) ;
    Cout <= Sum(16) ;
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;

```

圖2.29 圖2.28的16位元加法器，加上進位和溢位訊號

```

LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY adder16 IS
    PORT ( Cin           : IN      STD_LOGIC ;
           X, Y          : IN      SIGNED(15 DOWNT0 0) ;
           S             : OUT     SIGNED(15 DOWNT0 0) ;
           Cout, Overflow : OUT     STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : SIGNED(16 DOWNT0 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(15 DOWNT0 0) ;
    Cout <= Sum(16) ;
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;

```

2.30 使用算術包裝

```
ENTITY adder16 IS
    PORT ( X, Y      : IN      INTEGER RANGE -32768 TO 32767 ;
           S          : OUT     INTEGER RANGE -32768 TO 32767 );
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
BEGIN
    S <= X + Y ;
END Behavior ;
```

圖2.31 圖2.28的16位元加法器使用INTEGER訊號

二十、乘法 Multiplication

- 有效率地將 B 的所有位元往左移，也就是將 B 往左位移(shift)一個位元位置。
- 圖2.32顯示如何手動執行乘法，以四位元數值為例。從右到左檢視每個乘數位元。若位元為1，加上被乘數的位移版本以形成部分乘積(partial product)。
- 同樣的方法也可以用來設計乘法器電路。
- 這種循序方法會產生很慢的電路，主要是因為用單一八位元加法器執行產生部分乘積和最終乘積的所有加法。

		Multiplicand
		Multiplier
被乘數 M	(14)	1 1 1 0
乘數 Q	(11)	× 1 0 1 1

		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0

乘積 P	(154)	1 0 0 1 1 0 1 0

(a) 手動做乘法

被乘數 M	(11)	1 1 1 0
乘數 Q	(14)	× 1 0 1 1

部分乘積 0		1 1 1 0
Partial product		+ 1 1 1 0

部分乘積 1		1 0 1 0 1
		+ 0 0 0 0

部分乘積 2		0 1 0 1 0
		+ 1 1 1 0

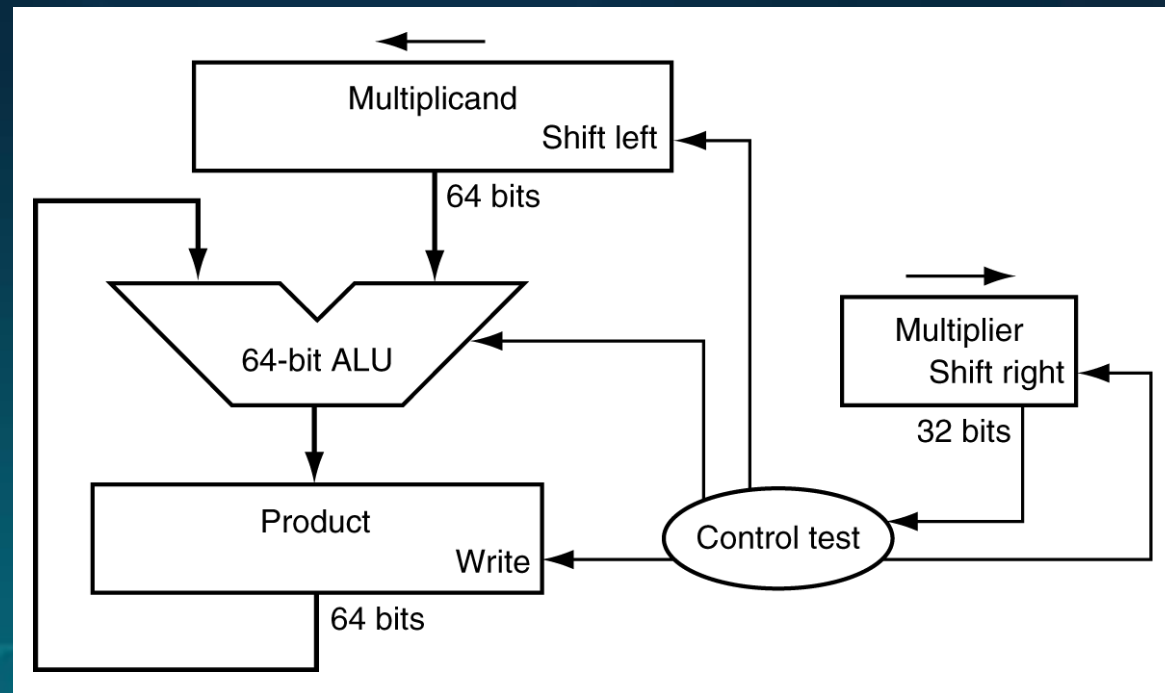
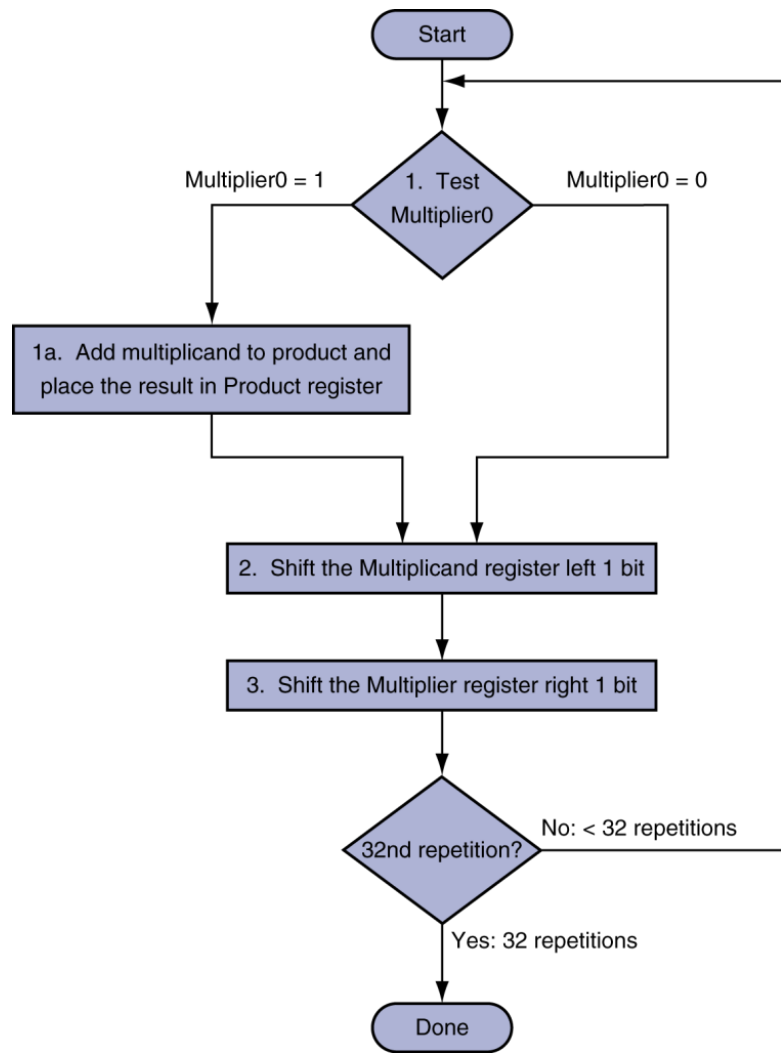
乘積 P	(154)	1 0 0 1 1 0 1 0

(b) 以硬體實作乘法

慢的單一八位元加法器

圖2.32 無符號數的乘法

Multiplication Hardware



Initially 0

二十一、無符號數的陣列乘法器 Array Multiplier for Unsigned Numbers

- 將 q_0 與 M 的每個位元作AND可以產生部分乘積0
- $PP0 = pp0_3 pp0_2 pp0_1 pp0_0$ 。因此

$$PP0 = m_3 q_0 \ m_2 q_0 \ m_1 q_0 \ m_0 q_0$$

- 將 q_1 與 M 作AND且加到 $PP0$ 可以產生部分乘積1， $PP1$ ，如下

$$\begin{array}{rcccccc} PP0: & 0 & pp0_3 & pp0_2 & pp0_1 & pp0_0 \\ & + m_3 q_1 & m_2 q_1 & m_1 q_1 & m_0 q_1 & 0 \\ \hline PP1: & pp1_4 & pp1_3 & pp1_2 & pp1_1 & pp1_0 \end{array}$$

- 類似地，將 q_2 與 M 作AND且加到 $PP1$ 可以產生部分乘積2， $PP2$ ，以此類推……

二十二、帶正負號數的乘法 Multiplication of Signed Numbers

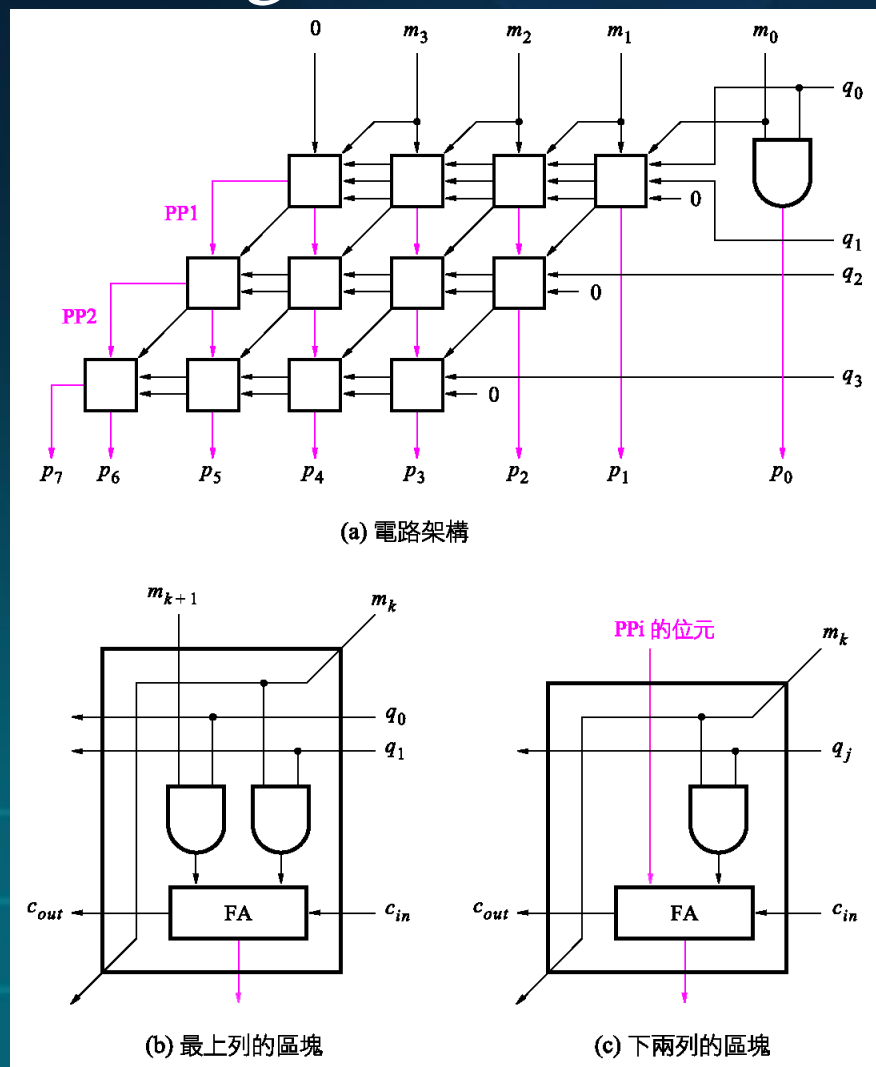


圖2.33 4×4乘法器電路

二十二、帶正負號數的乘法 Multiplication of Signed Numbers

被乘數 M	(+14)	0 1 1 1 0
乘數 Q	(+11)	× 0 1 0 1 1
部分乘積 0		0 0 0 1 1 1 0
		+ 0 0 1 1 1 0
部分乘積 1		0 0 1 0 1 0 1
		+ 0 0 0 0 0 0
部分乘積 2		0 0 0 1 0 1 0
		+ 0 0 1 1 1 0
部分乘積 3		0 0 1 0 0 1 1
		+ 0 0 0 0 0 0
乘積 P	(+154)	0 0 1 0 0 1 1 0 1 0

(a) 被乘數為正數

被乘數 M	(-14)	1 0 0 1 0
乘數 Q	(+11)	× 0 1 0 1 1
部分乘積 0		1 1 1 0 0 1 0
		+ 1 1 0 0 1 0
部分乘積 1		1 1 0 1 0 1 1
		+ 0 0 0 0 0 0
部分乘積 2		1 1 1 0 1 0 1
		+ 1 1 0 0 1 0
部分乘積 3		1 1 0 1 1 0 0
		+ 0 0 0 0 0 0
乘積 P	(-154)	1 1 0 1 1 0 0 1 1 0

(b) 被乘數為負數

圖2.34 帶正負號數的乘法

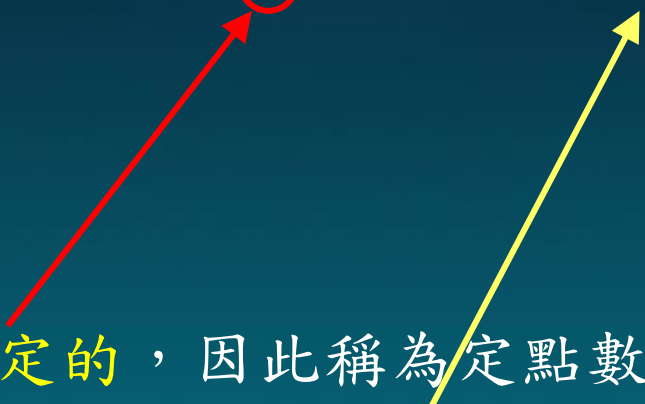


其他數值表示法

OTHER NUMBER REPRESENTATIONS

二十四、定點數 Fixed-Point Numbers

- 定點數(fixed-point number)由整數和小數部分組成。可以用位置數值表示法寫成

$$B = b_{n-1}b_{n-1} \cdots b_1b_0 \cdot b_{-1}b_{-2} \cdots b_{-k}$$


- 此數的值為
- $V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$
- 假設基數點的位置是固定的，因此稱為定點數。若沒有顯示基數點，則假設在最低有效數字的右邊，表示此數值為整數。

二十五、浮點數

Floating-Point Numbers

- 定點表示法需要許多有效位元，我們不用定點表示法，比較好的方式是用浮點數表示法，由包含有效數字的尾數(mantissa)和基數R的指數(exponent)表示。其格式為

$$\text{尾數} \times R^{\text{指數}}$$

- 尾數這數值通常是正規化的(normalized)，使得基數點位在第一個非零數字的右邊，例如 3.234×10^{43} 或 6.31×10^{-28} 。
- 二進位浮點表示法已經被電子電機工程師協會 (Institute of Electrical and Electronic Engineers, IEEE) 正規化[3]。此標準指定兩種尺寸的格式，如圖 3.32 所示。
 - 單精度 (single-precision) 32位元格式
 - 雙精度 (double-precision) 64位元格式

單精度浮點格式

Single-Precision Floating-Point Format

- IEEE標準指定指數為**超127(excess-127)格式**，而不是範圍從-128到127的八位元帶正負號數。此格式必須加127到真實指數值，使得

$$\text{指數} = E - 127$$

- 依此方式 E 就變成正數。
- 尾數以23位元表示**。IEEE標準需要正規化的尾數，表示**MSB永遠為1**。因此不必將此位元包含在尾數欄位。因此，**若 M 為尾數欄位的位元向量**，則尾數的實際值為 **$1.M$** ，為**24位元尾數**。因此，圖3.33a的浮點格式代表的數值為

$$\text{值} = \pm 1.M \times 2^{E-127}$$

- 尾數欄位的大小使數值表示法的精準度大約為7個十進位數字。指數欄位範圍從 **2^{-126} 到 2^{127}** ，對應到大約 **$10^{\pm 38}$** 。

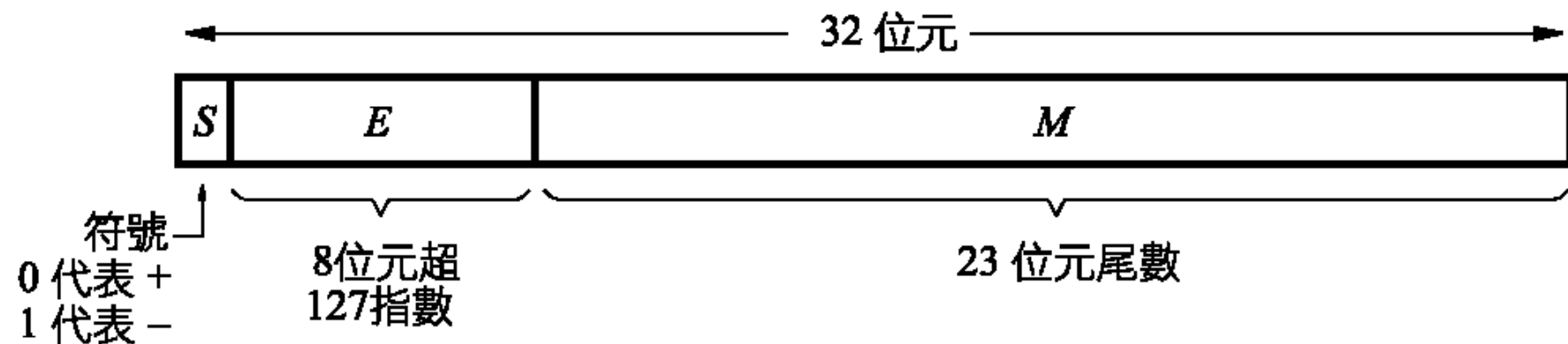
雙精度浮點格式

Double-Precision Floating-Point Format

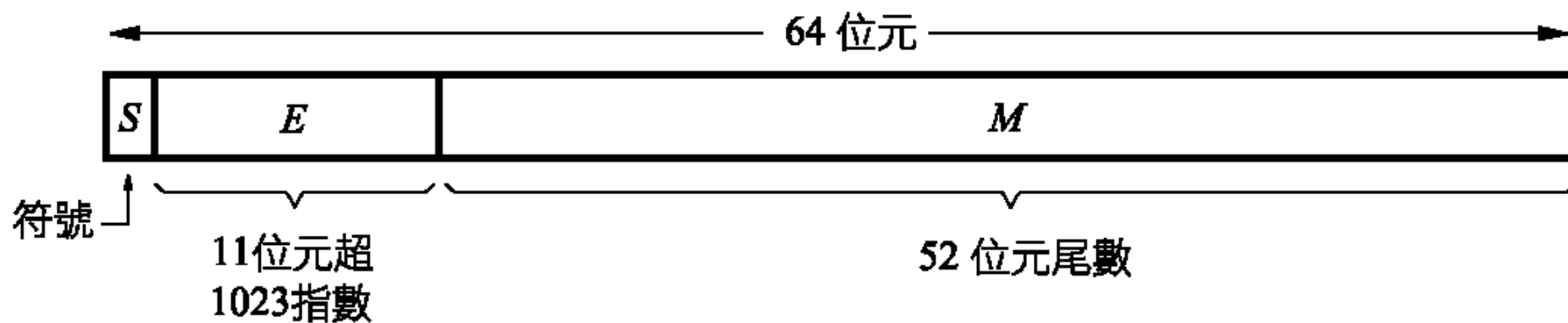
- 指數欄位有11位元，以超1023(excess-1023)格式指定指數，其中

$$\text{指數} = E - 1023$$

- E的範圍從0到2047，但是 $E = 0$ 和 $E = 2047$ 分別表示零和無限大。因此，一般的指數範圍從-1022到1023，以 E 表示為從1到2046。
- 尾數欄位有52位元。因為假設尾數為正規化的，其實際值為 $1.M$ 。因此，浮點數的值為
$$\text{值} = \pm 1.M \times 2^{E-1023}$$
此格式使數值表示法的精準度大約為16個十進位數字，範圍大約為 $10^{\pm 308}$ 。



(a) 單精度



(c) 雙精度

圖2.35 IEEE標準浮點格式

二十六、二進位編碼十進位表示法 Binary-Coded-Decimal Representation

- 在數位系統中，我們可以將十進位數值的每個數字都以二進位形式編碼。這稱為二進位編碼十進位(binary-coded-decimal, BCD)表示法。
- 它主要的優點是當數值資訊必須顯示在簡單數字導向螢幕時，這個格式很方便。其缺點為電腦執行算術運算的複雜度，以及浪費6個可用的編碼型樣。

表2.3 BCD 數字

Decimal digit	BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

.....6個可用的編碼型樣

BCD加法

BCD Addition

- 兩個BCD數字的加法總和可能會超過9，因此需要校正。
- 有兩種情況必須作校正，圖2.36說明這些情況：
 - 總和大於9但是使用四位元沒有進位輸出。
 - 總和大於15使得使用四位元產生進位輸出。
- 因此可以安排計算如下

$$Z=X+Y$$

若 ≤ 9 ，則進位輸出 = 0

若 > 9 ，則進位輸出 = 1

- 圖2.36的第二個例子顯示 $X+Y > 15$ 的情況。此例中Z的四個LSB代表數字1是錯的。

X	0 1 1 1	7
+ Y	+ 0 1 0 1	+ 5
<hr/>		
Z	1 1 0 0	12
	+ 0 1 1 0	
	<hr/>	
進位 →	1 0 0 1 0	
	<hr/>	
	S = 2	

X	1 0 0 0	8
+ Y	+ 1 0 0 1	+ 9
<hr/>		
Z	1 0 0 0 1	17
	+ 0 1 1 0	
	<hr/>	
進位 →	1 0 1 1 1	
	<hr/>	
	S = 7	

圖2.36 BCD數字的加法

BCD加法

BCD Addition (cont'd)

- 若我們想要手動產生實作圖2.37方塊圖的電路，而不是用VHDL，則可以用下列方法。為了定義 $Adjust$ 函數，我們觀察發現若中間總和超過9，則四位元加法器的進位輸出為1，或是如果，或者 z_2 或 z_1 (或兩者皆)為1。因此此函數的邏輯表示式為

$$Adjust = Carry-out + z_3(z_2 + z_1)$$

- 我們不是實作另一個完整四位元加法器以執行校正，而是以較簡單的電路，因為加常數6不需要完整的四位元加法器。

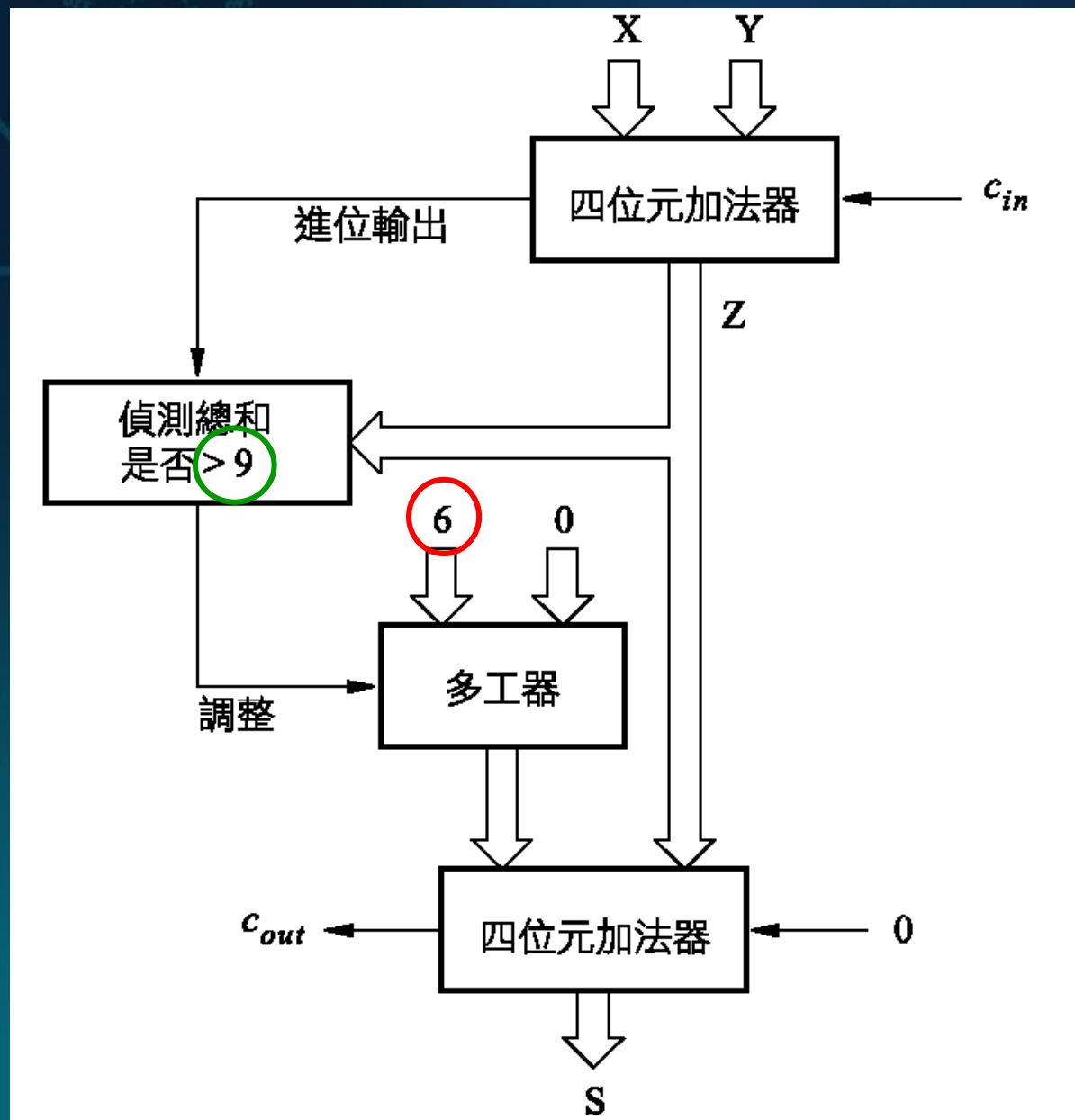


圖2.37 單一數字BCD加法器的方塊圖

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY BCD IS
    PORT ( X, Y      : IN      STD_LOGIC_VECTOR(3 DOWNTO 0) ;
          S          : OUT     STD_LOGIC_VECTOR(4 DOWNTO 0) ) ;
END BCD ;

ARCHITECTURE Behavior OF BCD IS
    SIGNAL Z : STD_LOGIC_VECTOR(4 DOWNTO 0) ;
    SIGNAL Adjust : STD_LOGIC ;
BEGIN
    Z <= ('0' & X) + Y ;
    Adjust <= '1' WHEN Z > 9 ELSE '0' ;
    S <= Z WHEN (Adjust = '0') ELSE Z + 6 ;
END Behavior ;
```

圖2.38 單一數字BCD加法器的VHDL程式碼

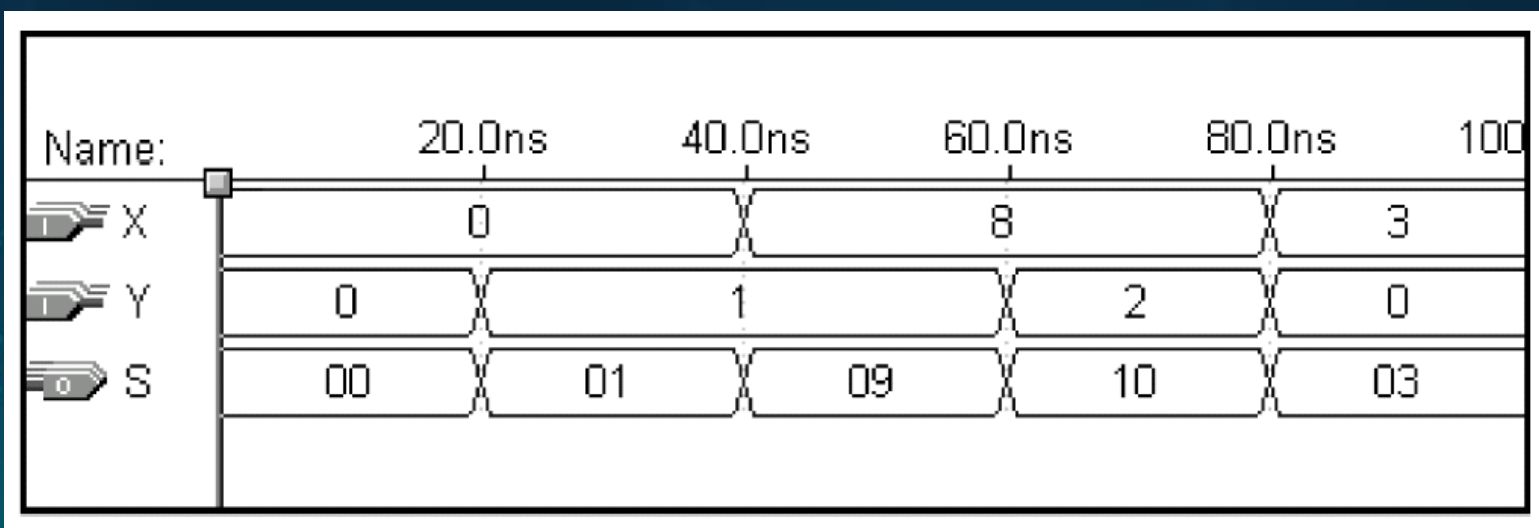


圖2.39 圖3.38中VHDL程式碼的功能性模擬

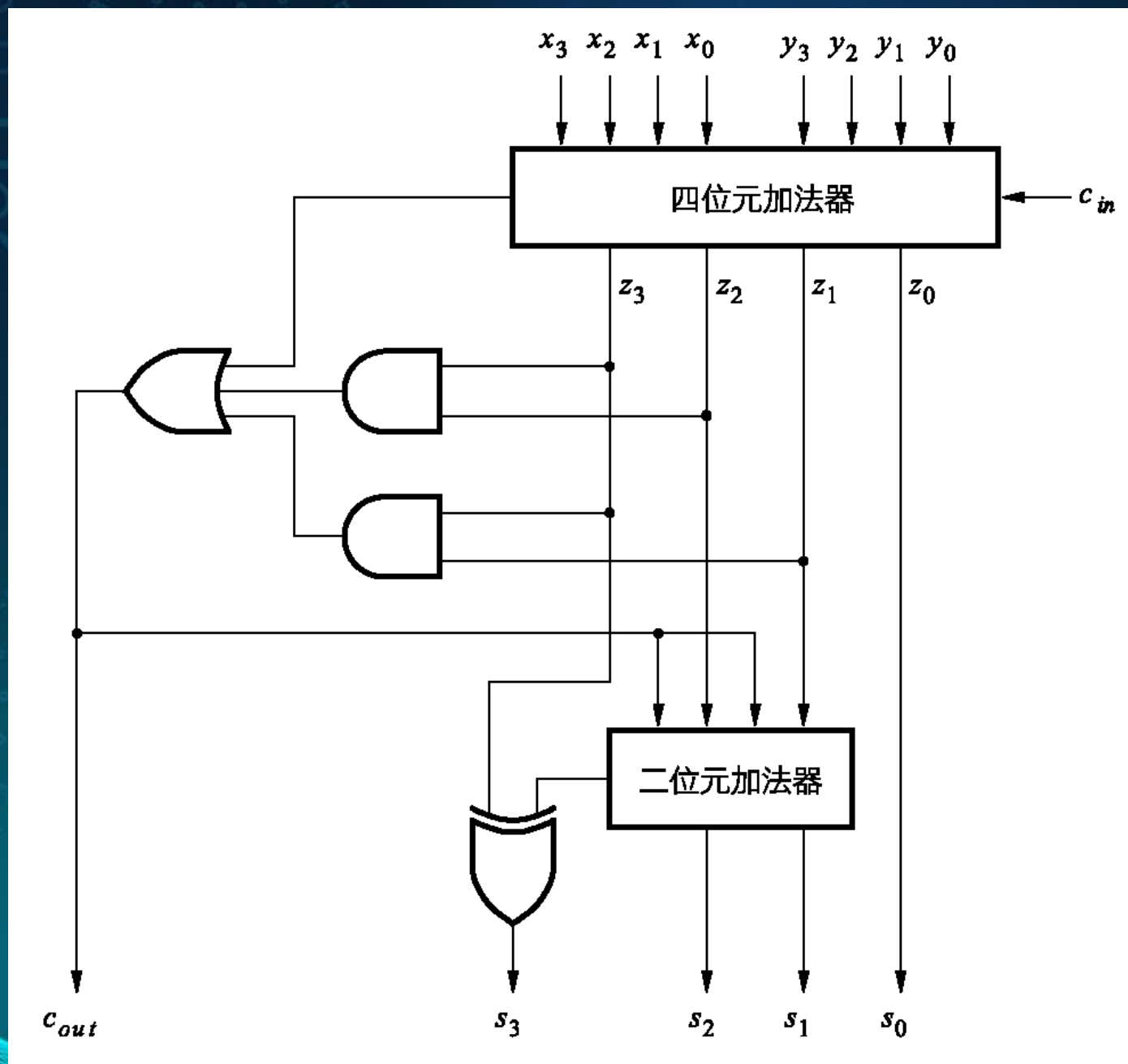


圖2.40 單一數字BCD加法器的電路