Digital System Design

Lecture 5
Arithmetic Circuits

- **Reading Assignment:**
  - **Brown, “Fundamentals of Digital Logic with VHDL, pp. 249 - 302,**
- **Learning Objective:**
  - **Present the design and timing considerations of circuits to perform basic arithmetic operations such as addition, subtraction, multiplication, and division**

Addition and Subtraction

Full Adder

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

(a) Truth table

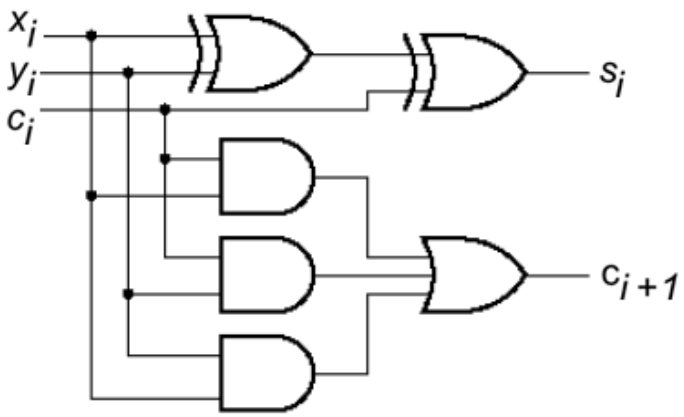
c_i \ x_iy_i	00	01	11	10
0		1		1
1	1		1	

$s_i = x_i \oplus y_i \oplus c_i$

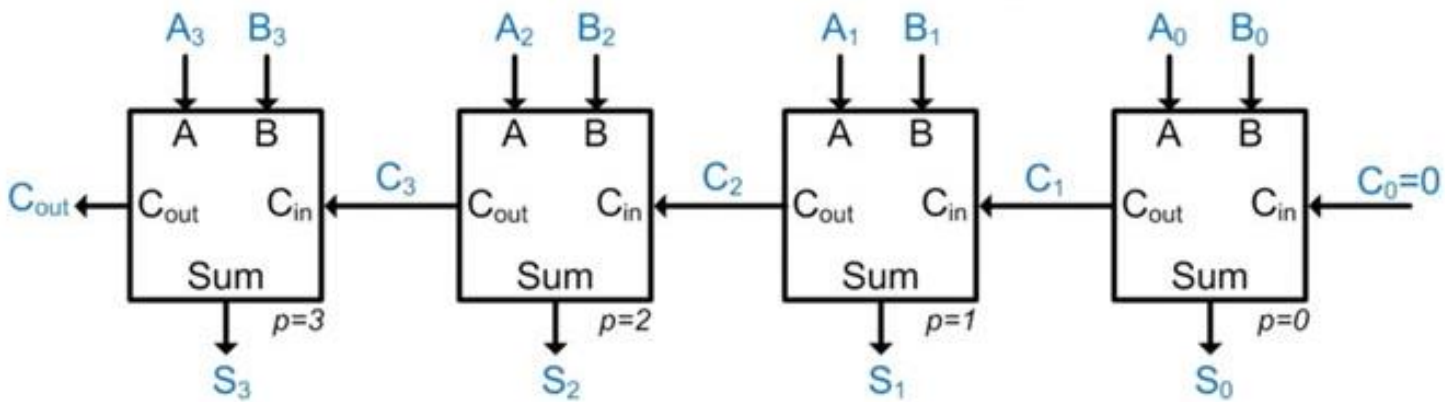
c_i \ x_iy_i	00	01	11	10
0			1	
1		1	1	1

$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$

(b) Karnaugh maps



Design of a 4-bit Ripple Carry Adder



- Assume each FA has Δ t delay =>

  The worst-case delay of a ripple carry adder is: $4\,\Delta t$

  (critical path delay)

- Detecting Overflow=>

  For unsigned 4-bit number: Overflow = C_{out} (C_4)

  For unsigned n-bit numbers: Overflow = C_n

  For signed 4-bit number: Overflow = $C_3C_{out}' + C_3'C_{out}$

  $= C_3 \oplus C_{out}$

  For signed n-bit numbers: Overflow = $C_{n-1} \oplus C_n$

Detecting Signed Number Overflow

Overflow occurs when:

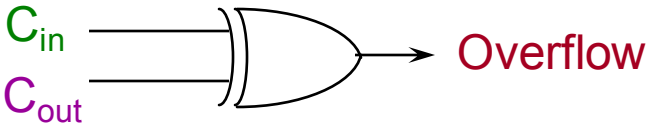
We add two positive numbers and obtain a negative

We add two negative numbers and obtain a positive

Looking at the sign bit (MSB):

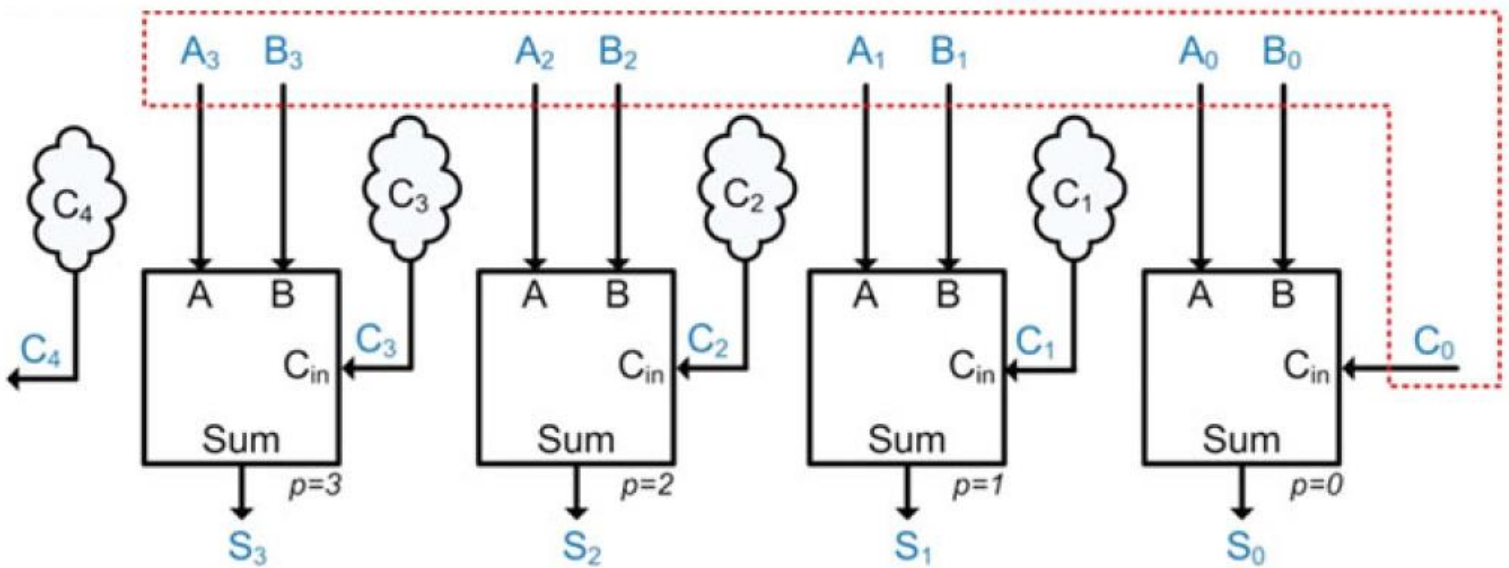
C_{out} 0, C_{in} 0	C_{out} 0, C_{in} 1	C_{out} 1, C_{in} 0	C_{out} 1, C_{in} 1	C_{out} 0, C_{in} 0	C_{out} 1, C_{in} 1
+ 0	+ 0	- 1	- 1	- 1	- 1
+ + 0	+ + 0	- + 1	- + 1	+ + 0	+ + 0
+ 0	- 1	+ 0	- 1	- 1	+ 0
No overflow	Overflow	Overflow	No Overflow	No Overflow	No Overflow

Overflow when $carry_{in}$ to sign bit does not equal $carry_{out}$



Carry Look Ahead Adder

- In order to address the potentially significant delay of a ripple carry adder, a carry look ahead (CLA) adder was created.
- In this approach, additional circuitry is included that produces the intermediate carry in signals immediately, <u>instead of waiting for them to be created by the preceding full adder stage.</u>



Carry Look Ahead Adder (Cont.)

- Must determine carry quickly.
  - $C_{i+1} = x_i y_i + x_i C_i + y_i C_i$
  - $C_{i+1} = x_i y_i + (x_i + y_i) C_i$
  - $C_{i+1} = g_i + p_i C_i$ where $g_i = x_i y_i$ (generate)
    $p_i = x_i + y_i$ (propagate)

- We can now write expressions for the subsequent carry terms as:

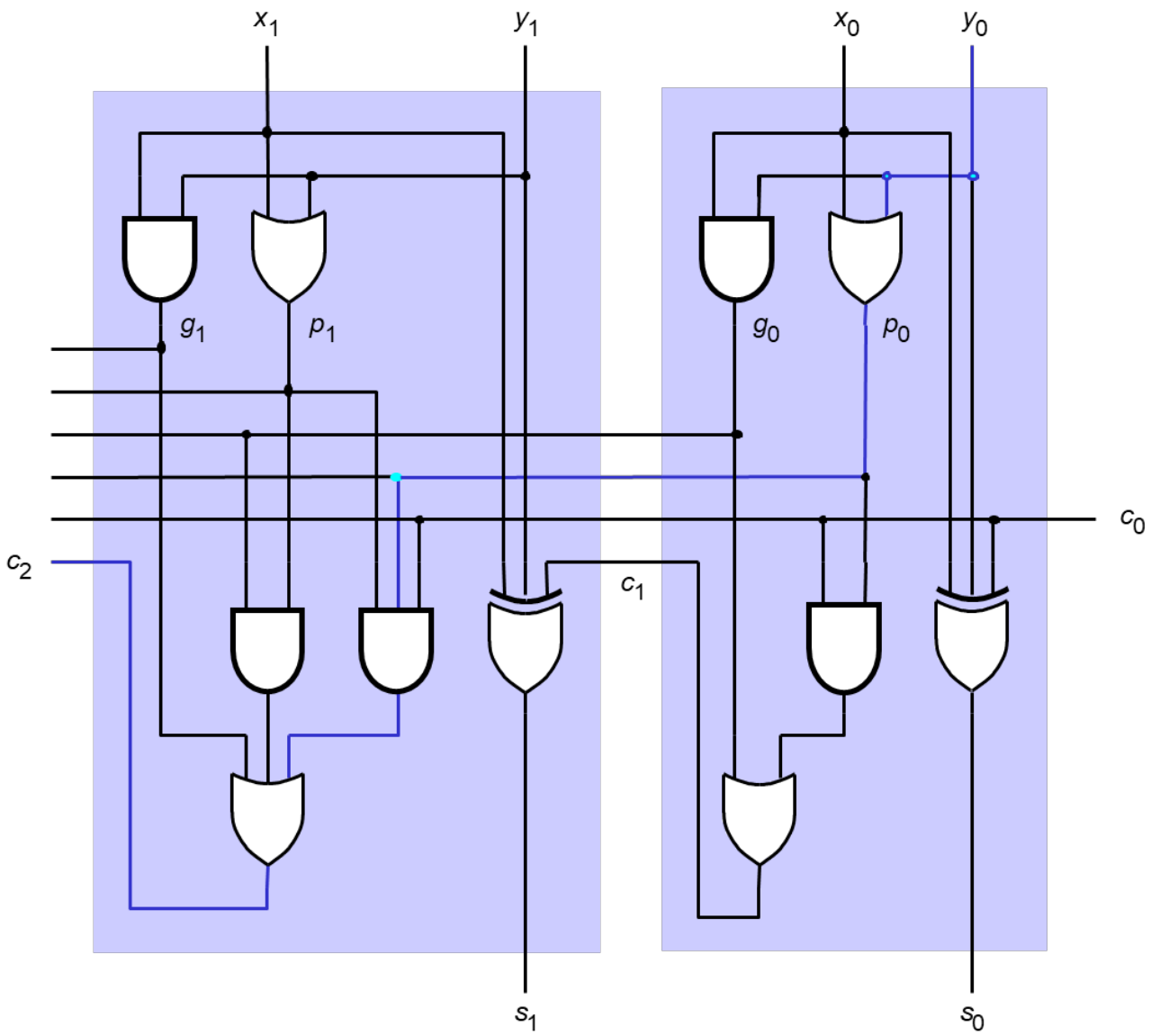
$$C_1 = g_0 + p_0 \cdot C_0$$

$$C_2 = g_1 + p_1 \cdot C_1$$
$$C_2 = g_1 + p_1 \cdot (g_0 + p_0 \cdot C_0)$$
$$C_2 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot C_0$$

$$C_3 = g_2 + p_2 \cdot C_2$$
$$C_3 = g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_0 \cdot p_1 \cdot C_0)$$
$$C_3 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot C_0$$

$$C_4 = g_3 + p_3 \cdot C_3$$
$$C_4 = g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot C_0)$$
$$C_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot C_0$$

The first two stages of a Carry Look Ahead Adder



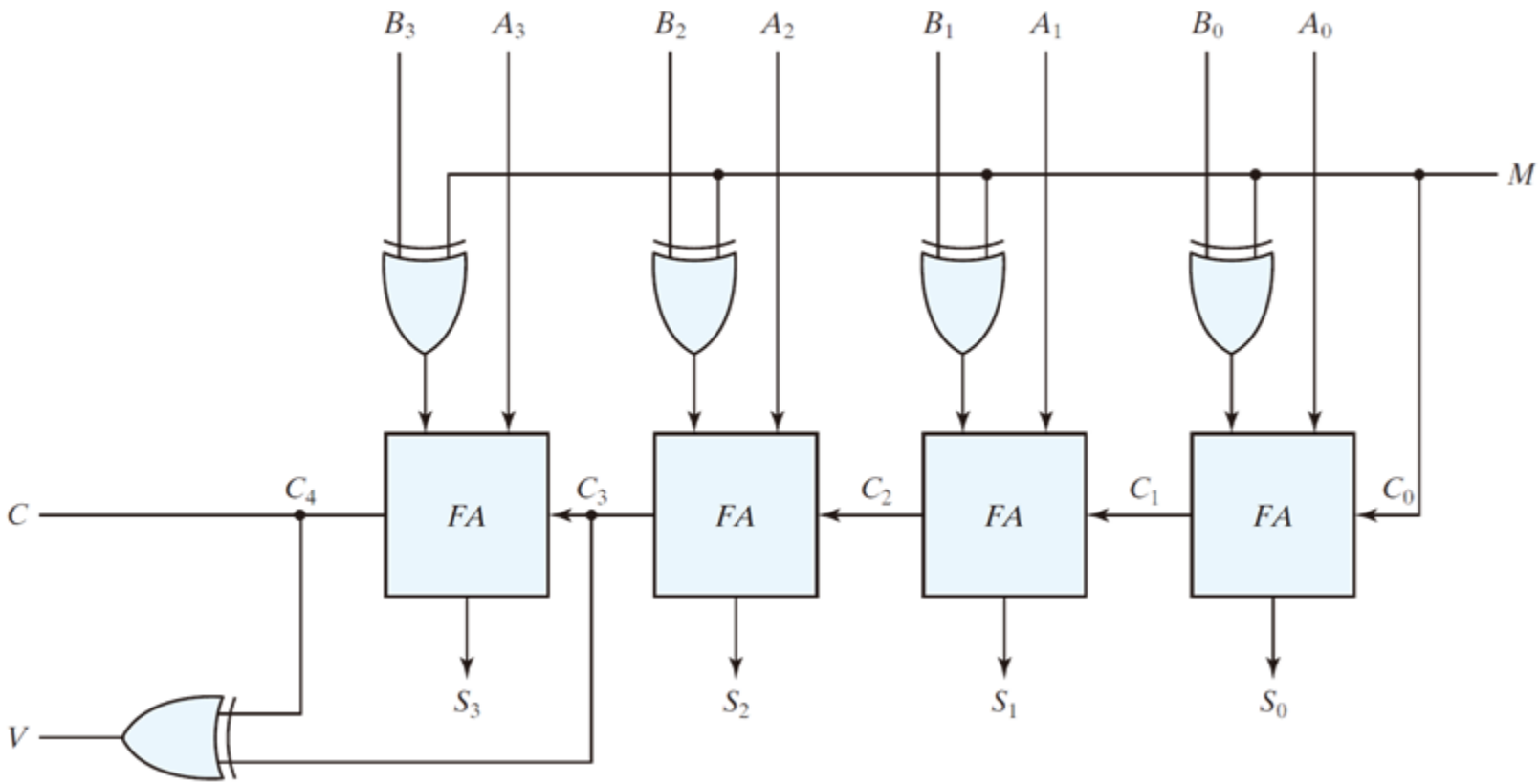
High Fan-in Issues

- $c_1 = g_0 + p_0 c_0$
- $c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$
- ...
- $c_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3$
  $+ p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1$
  $+ p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0$

- $c_8 = (g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4) + [(p_7 p_6 p_5 p_4)(g_3 +$
  $p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0) + (p_7 p_6 p_5 p_4)(p_3 p_2 p_1 p_0) c_0$

Binary Subtractor

- A - B = A + (2's complement of B)
- 4-bit Adder/Subtractor
  - M=0, A+B;
  - M=1, A+B'+1



A Full-Adder (DataFlow)

```
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 ;
```

A four-bit Adder (Structure)

```
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 ;
```

Declaration of a Package

```
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 ;
```

Using a package for the four-bit adder

```
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 ;
```

A four-bit adder defined using multibit signals

```
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 DOWNTO 0) ;
                S           : OUT       STD_LOGIC_VECTOR(3 DOWNTO 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 ;
```

A four-bit Carry Look Ahead Adder (Structure)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity cla_4bit is
  port (A, B   : in std_logic_vector(3 downto 0);
        Sum    : out std_logic_vector(3 downto 0);
        Cout   : out std_logic);
end cla_4bit ;

architecture cla_4bit_arch of cla_4bit is

   component full_adder
     port (A, B, Cin   : in  std_logic;
           Sum, p, g   : out std_logic);
   end component;

   signal C0, C1, C2, C3 : std_logic;
   signal p, g           : std_logic_vector(3 downto 0);

 begin

   C0   <= '0';
   C1   <= g(0) or (p(0) and C0)
   C2   <= g(1) or (p(1) and C1)
   C3   <= g(2) or (p(2) and C2)
   Cout <= g(3) or (p(3) and C3)

   A0 : full_adder port map (A(0), B(0), C0, Sum(0), p(0), g(0));
   A1 : full_adder port map (A(1), B(1), C1, Sum(1), p(1), g(1));
   A2 : full_adder port map (A(2), B(2), C2, Sum(2), p(2), g(2));
   A3 : full_adder port map (A(3), B(3), C3, Sum(3), p(3), g(3));

end cla_4bit_arch ;
```

Arithmetic Packages

- ***std_logic_signed*** **package defines signed arithmetic for std_logic type.**
- ***std_logic_unsigned*** **package defines unsigned arithmetic for std_logic type.**
- **These are built on top of the package** ***std_logic_arith*****.**

- ***numeric_std*** ***package can completely replace the above three packages*** **(more popular nowaday).**

https://www.itread01.com/content/1541679484.html

Binary Arithmetic

- When performing binary arithmetic, the results of arithmetic operations and comparisons vary greatly depending on whether the binary number is unsigned or signed.
  - Example1: “1001” means

    *std_logic_vector : 4 binary bits

    *unsigned number : 9

    *signed number : -7 （2’s complement representation）

  - Example2: (A = “0000”, B =“1111”)

    if (A < B) then -- This condition is TRUE if A and B are UNSIGNED

    if (A < B) then -- This condition is FALSE if A and B are SIGNED

A 16-bit unsigned adder (Behavior)

裡面沒有 +, −, *, / 運算

要引用該 Package，compiler 才能成功編譯

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

ENTITY uadder16 IS
        PORT (    X, Y        : IN        STD_LOGIC_VECTOR(15 DOWNTO 0) ;
                  S           : OUT       STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END uadder16 ;

ARCHITECTURE Behavior OF uadder16 IS
BEGIN
        S <= X + Y ;
END Behavior ;
```

A 16-bit signed adder (Behavior)

```
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 ;
```

A 16-bit unsigned adder (Behavior with Arith package)

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

ENTITY uadder16 IS
        PORT (      X, Y        : IN        UNSIGNED (15 DOWNTO 0) ;
                    S           : OUT       UNSIGNED(15 DOWNTO 0) ) ;
END uadder16 ;

ARCHITECTURE Behavior OF uadder16 IS
BEGIN
        S <= X + Y ;
END Behavior ;
```

A 16-bit signed adder (Behavior with Arith package)

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

ENTITY adder16 IS
        PORT (    X, Y        : IN        SIGNED (15 DOWNTO 0) ;
                  S           : OUT       SIGNED(15 DOWNTO 0) ) ;
END uadder16 ;

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

A 16-bit signed adder (Behavior with NUMERIC_STD package)

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

ENTITY adder16 IS
        PORT (    X, Y        : IN        SIGNED (15 DOWNTO 0) ;
                  S           : OUT       SIGNED(15 DOWNTO 0) ) ;
END uadder16 ;

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

VHDL's Predefined Operators

Predefined operators for VHDL's integer and boolean types.

integer Operators		*boolean* Operators	
+	addition	and	AND
-	subtraction	or	OR
*	multiplication	nand	NAND
/	division	nor	NOR
mod	modulo division	xor	Exclusive OR
rem	modulo remainder	xnor	Exclusive NOR
abs	absolute value	not	complementation
**	exponentiation		

A 16-bit adder using Built-in INTEGER signals

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

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 ;
```

A 16-bit unsigned adder with carry and overflow

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

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

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

A 16-bit signed adder with carry and overflow - I

```
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 DOWNTO 0) ;
            S                : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0) ;
            Cout, Overflow   : OUT   STD_LOGIC ) ;
END adder16 ;

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

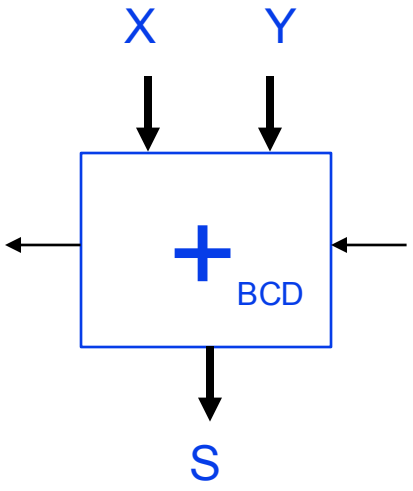
A 16-bit signed adder with carry and overflow - II

```
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 DOWNTO 0) ;
                S                   : OUT       SIGNED(15 DOWNTO 0) ;
                Cout, Overflow      : OUT       STD_LOGIC ) ;
END adder16 ;

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

BCD Adder

- Add two BCD's
  - 9 inputs: two BCD's and one carry-in
  - 5 outputs: one BCD and one carry-out

- Design approaches
  - A truth table with 2^9 entries
  - use binary full Adders
    - the sum <= 9 + 9 + 1 = 19
    - binary to BCD



BCD Adder

Table 4.5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

0的熱度 don't care

十位數

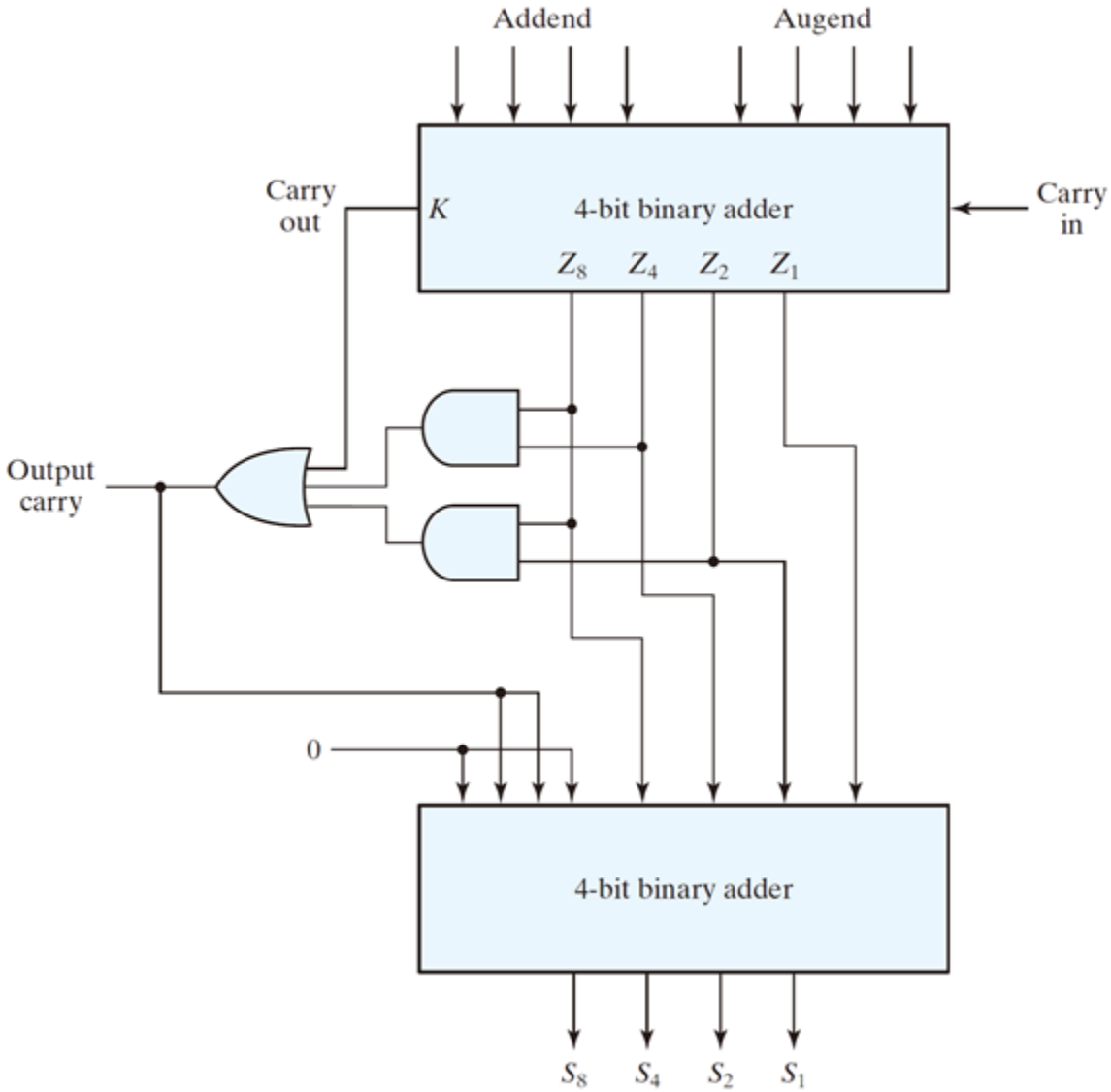
?

BCD Adder

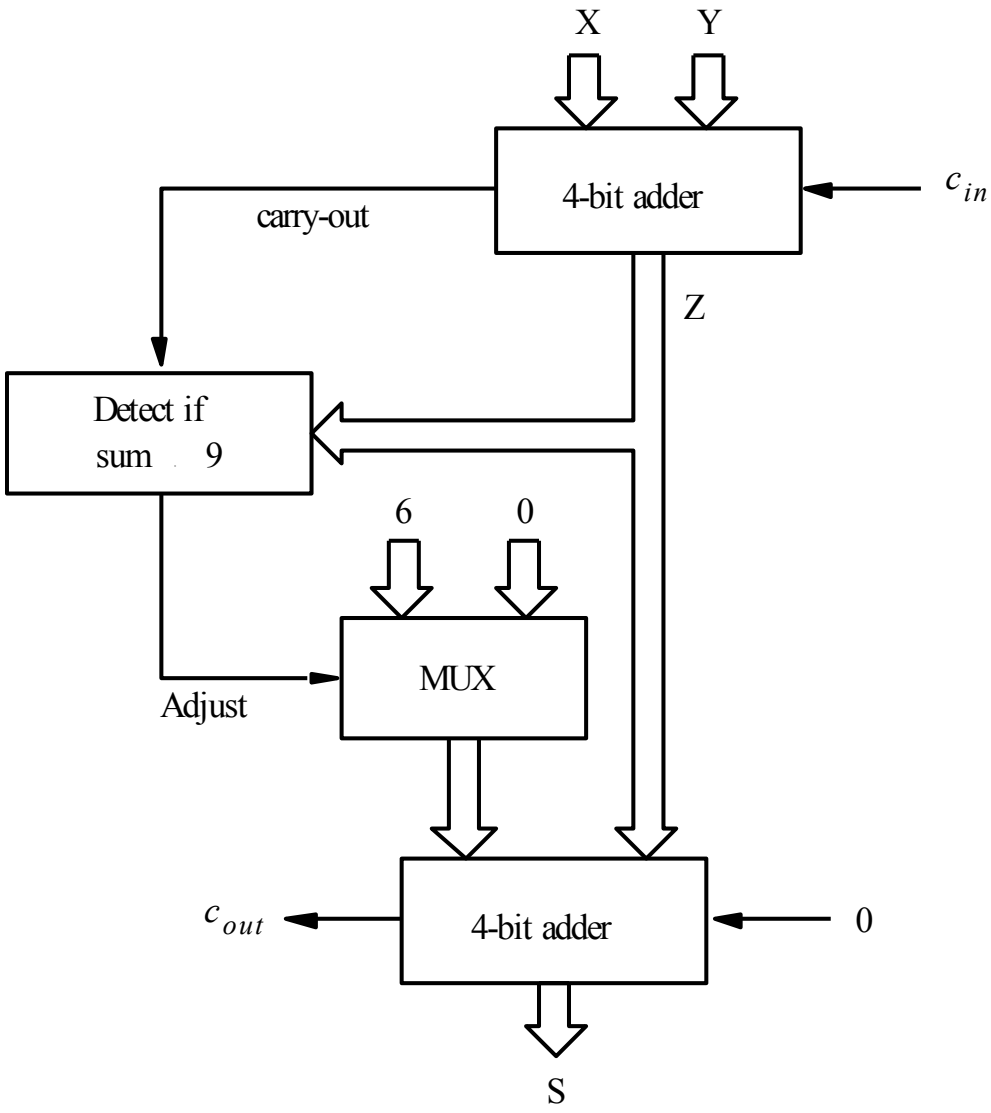
- Modifications are needed if the sum > 9
  - C = 1
    - K = 1
    - $Z_8Z_4 = 1$
    - $Z_8Z_2 = 1$
  - modification: +6

$$C = K + Z_8Z_4 + Z_8Z_2$$

BCD Adder


Addend
Augend
Carry
out
K
4-bit binary adder
Carry
in
Z_8 Z_4 Z_2 Z_1
Output
carry
0
4-bit binary adder
S_8 S_4 S_2 S_1

Block diagram for a one-digit BCD adder



A one-digit BCD adder (Behavior)

```
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 ;
```

Function and Procedure

型別轉換(Type Conversion)

- VHDL是一種資料型別檢查非常嚴格的語言，不同的資料型別彼此之間不能直接作直接設定敘述或運算 ;因此我們必需採用型別轉換的函式將其資料型別轉換成完全相同的型別才能進一步作運算處理。

TYPE	Value	Origin
std_ulogic	'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	std_logic_1164
std_ulogic_vector	array of std_ulogic	std_logic_1164
std_logic	resolved std_ulogic	std_logic_1164
std_logic_vector	array of std_logic	std_logic_1164
unsigned	array of std_logic	numeric_std, std_logic_arith
signed	array of std_logic	numeric_std, std_logic_arith
boolean	true, false	standard
character	191 / 256 characters	standard
string	array of character	standard
integer	$-(2^{31}-1)$ to $(2^{31}-1)$	standard
real	-1.0E38 to 1.0E38	standard
time	1 fs to 1 hr	standard

Packages for Numeric Operations

- **numeric_std** **-- IEEE standard**
  - Defines types signed, unsigned
  - Defines arithmetic, comparison, and logic operators for these types
- **std_logic_arith** **-- Synopsys,** a defacto industry standard
  - Defines types signed, unsigned
  - Defines arithmetic, and comparison operators for these types
- **std_logic_unsigned** **-- Synopsys,** a defacto industry standard
  - Defines arithmetic and comparison operators for std_logic_vector

Recommendation, if you use Synopsys Packages:

Use std_logic_arith for numeric operations

Use std_logic_unsigned only for counters and testbenches

函式(Functions)應用-型別轉換(Type Conversion)

- Altera Quartus II在其IEEE Library裡提供有***std_logic_arith*** Package，已經內含下面幾種格式轉換函式:

  - conv_integer： 將integer、unsigned、signed或std_ulogic轉換成整數(integer)。

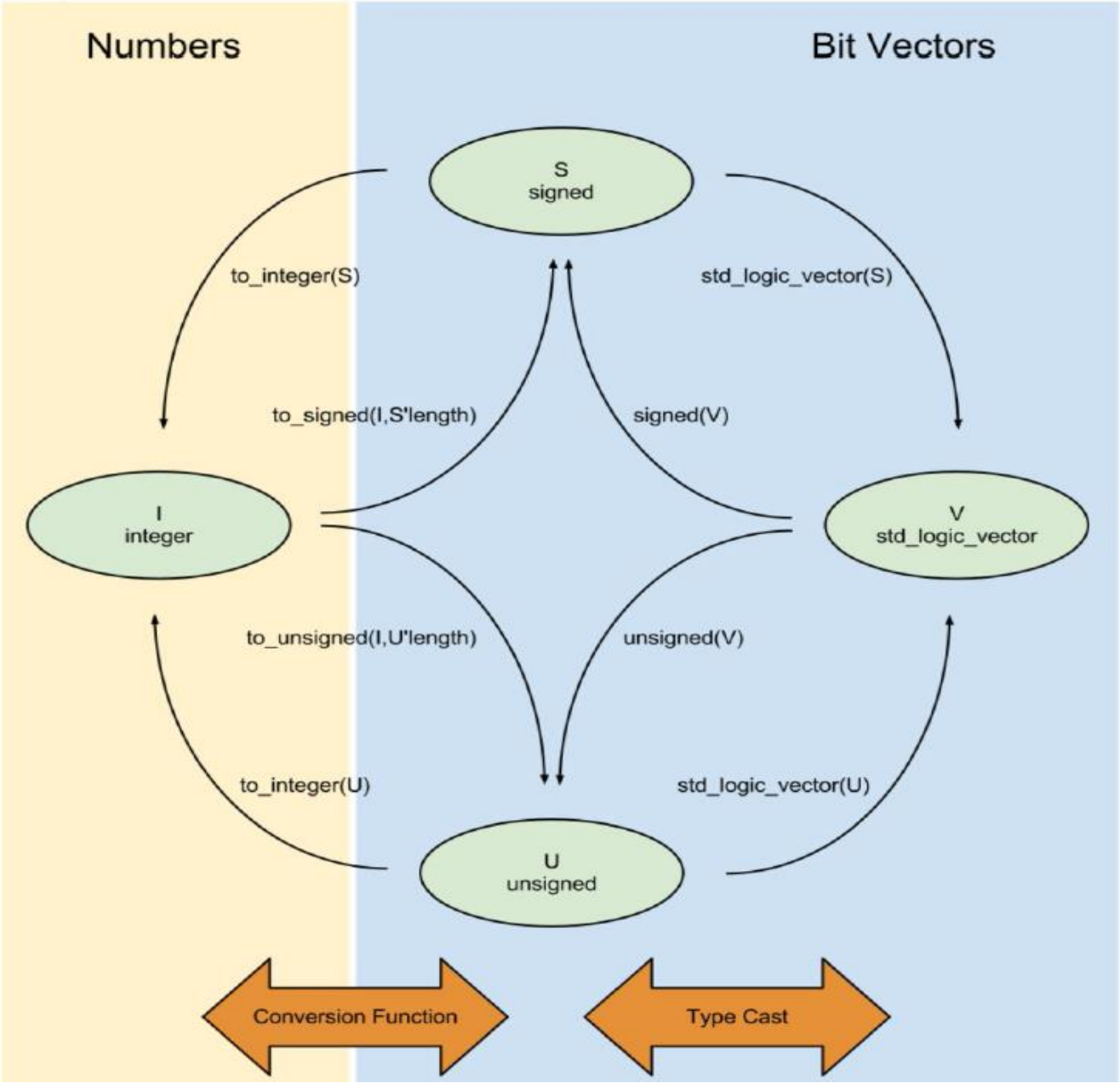
  - conv_unsigned： 將 integer、unsigned、signed 或 std_ulogic 轉 換 成 Unsigned數值。

  - conv_std_logic_vector： 將integer、unsigned、signed或std_ulogic轉換成std_logic_vector。

➢ **Examples of VHDL Conversions**

https://www.nandland.com/vhdl/tips/tip-convert-numeric-std-logic-vector-to-integer.html

Use numeric_std to convert between common VHDL types



Type Casting

Some VHDL built-in type casting operations are commonly used with the *numeric_std* package.

```
--signal definitions
signal slv : std_logic_vector(7 downto 0);
signal sgn : signed(7 downto 0);
signal usgn : unsigned(7 downto 0);

--FROM std_logic_vector TO signed/unsigned
sgn <= signed(slv);
usgn <= unsigned(slv);

-- FROM signed/unsigned TO std_logic_vector
svl <= std_logic_vector(sgn);
svl <= std_logic_vector(usgn);
```

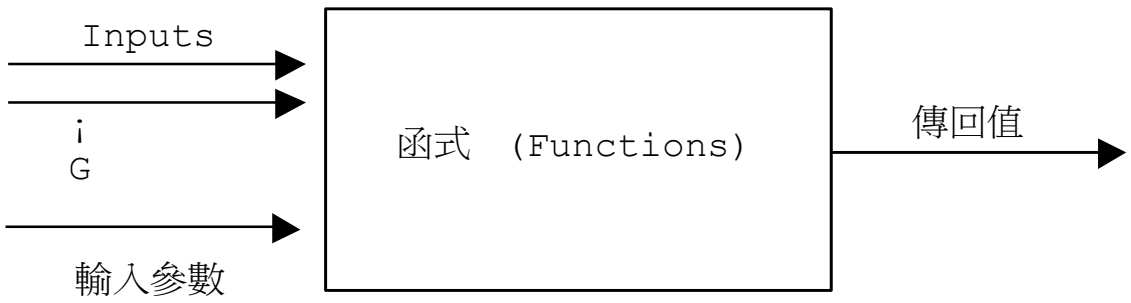
Conversion Functions

Functions are used to move between signed and unsigned types and the integer type.

(Note: These conversion functions are in numeric_std package.)

```
--signal definitions
signal i : integer;
signal sgn : signed(7 downto 0);
signal usgn : unsigned(7 downto 0);

--FROM integer TO signed/unsigned

sgn <= to_signed(i,8);
usgn <= to_unsigned(i,8);

-- FROM signed/unsigned TO integer

i <= to_integer(sgn);
i <= to_integer(usgn);
```

函式(Functions) 的語法



函式(Functions)的宣告語法如下:

Function 函式名稱 (輸入參數: 資料型別) **Return** 輸出參數型別;

函式(Functions)的主體內容語法如下:

Function 函式名稱 (輸入參數: 資料型別) **Return** 輸出參數型別 **IS**
    函式內的區域變數(Local Variable)宣告區
Begin
    函式內的主體內容區
    Return 輸出之參數名稱;
END 函式名稱;

函式(Functions) 的呼叫

```
ARCHITECTURE a OF carry_pack IS
BEGIN
 Process(a,b,c)
    Begin
        Cout<=carry(a,b,c);
 End process;
END a;
```

```
Function carry (bit1,bit2,bit3:
in std_logic)
```

函式(Functions)-Example1

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

PACKAGE my_package IS
    function carry (bit1,bit2,bit3: std_logic) return std_logic;
END my_package;
```

- When a procedure or function is declared in a package, its body (the algorithm part) must be placed in the “package body”:

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

package body my_package is
    function carry (bit1,bit2,bit3: std_logic) return std_logic IS
            variable result:std_logic;
    begin
            result:=(bit1 and bit2) or (bit1 and bit3) or (bit2 and bit3);
            return result;
   end ;
end my_package;
```

函式(Functions)-Example1 (續)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;        --使用目前工作目錄下的my_package套件

ENTITY  carry_pack is
PORT( a,b,c : IN std_logic;
      Cout   : OUT std_logic);
END carry_pack;

ARCHITECTURE a OF carry_pack IS
BEGIN
 Process(a,b,c)
 Begin
        Cout<=carry(a,b,c);
 End process;
END a;
```

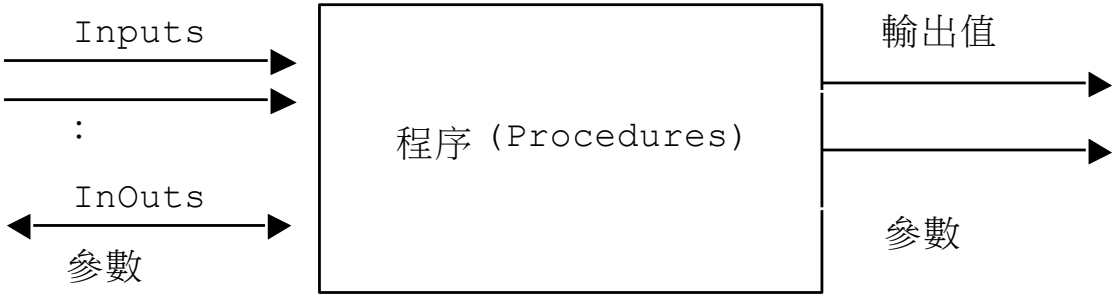
函式(Functions)-Example2

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

entity proc is
port (  s       :in integer range 0 to 15;
        A       :in integer range 0 to 31;
        B       :in integer range 0 to 31;
      Result :out integer range 0 to 63);
end proc;

Architecture a of proc IS

Function Sum( A  :in integer range 0 to 15;          --兩整數相加函式
              B  :in integer range 0 to 15)
  Return Integer IS
Begin
  Return A+B;
End Sum;

Function Diff( A  :in integer range 0 to 15;          --兩整數相減函式
               B  :in integer range 0 to 15)
  Return Integer IS
Begin
  Return A-B;
End Diff;
```

函式(Functions)-Example2 (續)

```
begin
 process(s)
  begin
      if (s=0) then
         Result<=Sum(A,B);                --s=0時呼叫兩整數相加函式
      elsif (s=1) then
         Result<=Diff(A,B);               --s=1時呼叫兩整數相減函式
      else
         Result<=63;                      --當s為0和1以外條件時輸出值為63
      end if;
 end process;
end a;
```

程序(Procedures) 的語法



- 程序(Procedures)的傳回值則可以不限於一個(與函式最大的不同)

- 在VHDL語言中，程序(Procedure)的宣告語法如下：

```
Procedure 程序名稱 ( Signal 訊號A:      資料型別;
                     Signal 訊號B:      資料型別;
                        .....
                     Signal 訊號E:      資料型別;
                     Signal 訊號M: OUT  資料型別;
                     Signal 訊號N: OUT  資料型別 ) IS
Begin
    程序的主體內容
END Procedure;
```

Example:四對一多工器

```
Library IEEE;
use IEEE.std_logic_1164.all;

entity proc_MUX41 is
port ( s:in std_logic_vector(1 downto 0);
       D0,D1,D2,D3: in std_logic;
       Y: out std_logic  );
end proc_MUX41;

architecture a of proc_MUX41 is

procedure MUX4x1 ( X: in std_logic_vector (1 downto 0);
                   A,B,C,D:in std_logic;
                   variable data: out std_logic ) is
 begin
   case X is
     when "00" => data := A;
     when "01" => data := B;
     when "10" =>  data := C;
     when others => data := D;
   end case;
 end MUX4x1;
```

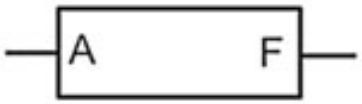
```
begin
  process
   variable     Z : std_logic;
  begin
      MUX4x1(S,D0,D1,D2,D3,Z);
      Y <= Z ;
  End process ;
end a;
```

Signal vs Variable

Sequential Signal Assignment Revisited

- Let's take a look at an example of how signals behave in a process

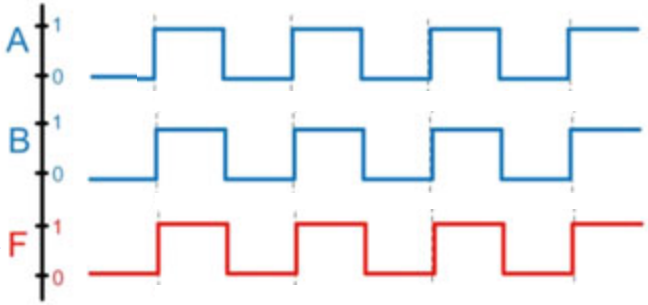
```
entity Ex is
  port (A  : in  bit;
        F  : out bit);
end entity;
```




```
architecture Ex_arch of Ex is

  signal B : bit;

begin

  Proc_Ex : process (A)
    begin
      B <= A;
      F <= not B;
  end process;

end architecture;
```

偵測A的改變(Trigger)

因為在此B還未被賦予A
F也被賦予A'(相當F並不知道A變動了)
∴導致F還是與A,B一樣的波形



- When A transitions from 0 to 1 with B=0 and F=0 initially. This transition triggers the process.
- The first signal assignment (B <= A) will cause B=1, but his assignment occurs only after the process ends.
- This means that when the second signal assignment is evaluated (F <= not B), it uses the initial value of B from when the process triggered (B = 0).
- When the process ends, A = 1, B = 1, and F = 1. Therefore, the behavior of this process will always result in A = B = F.

例如

A: 0 ⟶ 1

B <= A (此時 B still 0)

F <= not B (F still 舊 B)

Variable Assignment

- There are situations inside of processes in which it is desired for assignments to be made instantaneously instead of when the process suspends.
- For these situations, VHDL provides the concept of a *variable*.
- A variable has the following characteristics:
  - Variables only exist within a process.
  - Variables are defined in a process before the begin statement.
  - Once the process ends, variables are removed from the system. (This means that assignments to variables cannot be made by systems outside of the process.)
  - Assignments to variables are made using the := operator.
  - Assignments to variables are made instantaneously.

```
variable variable_name : <type> := <initial_value>;
```

Variable Assignment

V.S. P.52 的波形

- Let's reconsider previous example, but this time we'll use a variable in order to model the behavior where F is the complement of A.

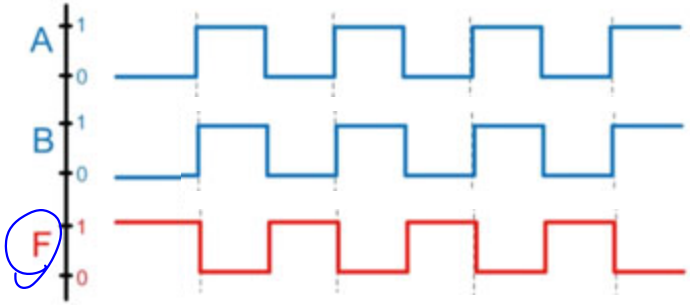
```
entity Ex is
  port (A  : in  bit;
        F  : out bit);
end entity;
```


```
architecture Ex_arch of Ex is

  signal B : bit;

begin

  Proc_Ex : process (A)
    variable temp : bit := '0';

    begin
      temp := A;
      B    <= temp;
      F    <= not temp;

  end process;

end architecture;
```

在此就直接賦值(若SIGNAL則要等離開process才賦值)

這邊就有了

temp = A
B = temp
F = not temp
= not A



A的反向

A Combinational Counting "1" Circuit

The intent of the code is to describe a combinational circuit that counts the number of bits in the three-bit signal X that are equal to 1.

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

ENTITY numbits IS
    PORT ( X      : IN      STD_LOGIC_VECTOR(1 TO 3) ;
           Count  : BUFFER  INTEGER RANGE 0 TO 3 ) ;
END numbits ;

ARCHITECTURE Behavior OF numbits IS
BEGIN
    PROCESS ( X )   -- count the number of bits in X with the value 1
    BEGIN
        Count <= 0 ;   -- the 0 with no quotes is a decimal number
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Count <= Count + 1 ;
            END IF ;
        END LOOP ;
    END PROCESS ;
END Behavior ;
```

↳ 可做 Input 也可 Output

↳ 若是過度宣告則會使 I/O 電路 Complexity 增加

要 Count 離開 Process 才會賦值 (但 Maybe 会 initial 0)
當作不影響

如果是 0 11

↳ 那只會做到 1 次

0 + 1 = 1
but still 沒賦值
0+1 = 1 → Ans

↳ 所以 Ans 会是 1

- Count is declared with the mode Buffer because it is used in the architecture body on both the left and right sides of an assignment operator.

A Combinational Counting “1” Circuit

- The code given in the figure is a legal VHDL code and can be compiled without generating any errors.
- However, it will not work as intended, and it does not represent a sensible logic circuit.

What’s wrong !

- Multiple assignment statements for the signal Count within the process.
  - ✓ Only the last of these assignments will have any effect.
- The statement “Count <= Count + ’1’ ;” describes a circuit with feedback.
  - ✓ Since the circuit is combinational, such feedback will result in oscillations and the circuit will not be stable.

A Correct Combinational Counting "1" Circuit

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

ENTITY Numbits IS
    PORT ( X      : IN    STD_LOGIC_VECTOR(1 TO 3) ;
           Count  : OUT   INTEGER RANGE 0 TO 3 ) ;
END Numbits ;

ARCHITECTURE Behavior OF Numbits IS
BEGIN
    PROCESS ( X )   -- count the number of bits in X equal to 1
        VARIABLE Tmp : INTEGER ;
    BEGIN
        Tmp := 0 ;
        FOR i IN 1 TO 3 LOOP
            IF X(i) = '1' THEN
                Tmp := Tmp + 1 ;
            END IF ;
        END LOOP ;
        Count <= Tmp ;
    END PROCESS ;
END Behavior ;
```

- Count is declared with the mode OUT.
- Variable Tmp is used instead of the signal Count inside the process, and the value of Tmp is assigned to Count at the end of the process.
- The := is called the variable assignment operator.
- Unlike <=, the assignment being scheduled until the end of the process. The variable assignment := takes place immediately.
- The variable does not represent a wire in a circuit, the FOR-LOOP need not be literally interpreted as a circuit with feedback.

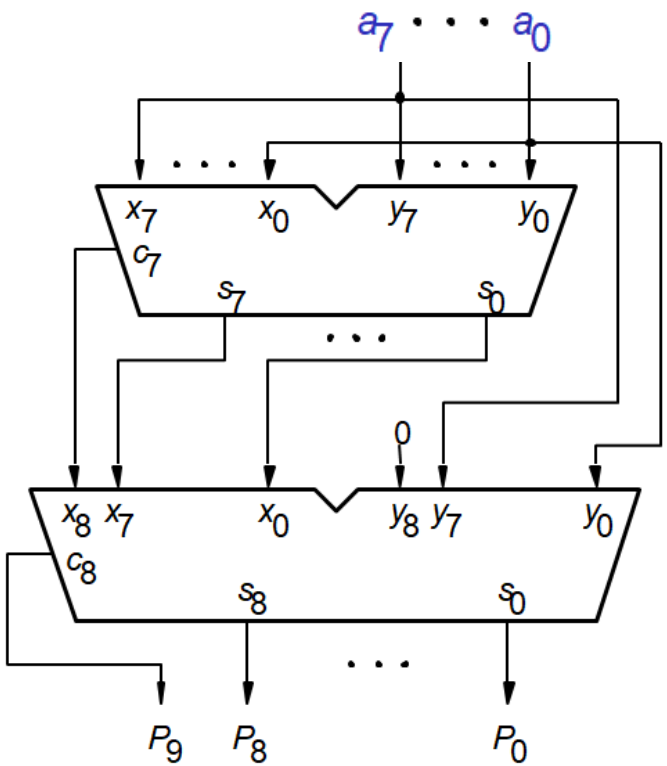
Multiplication and Division

Multiplication

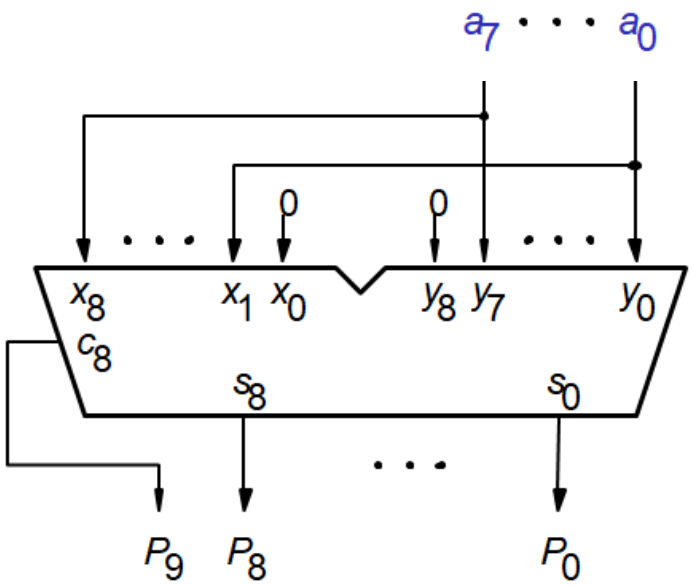
- Binary number can be multiplied by 2 shifting it one position to the left:
  - $A = a_{n-1}\, a_{n-2} \ldots a_1 a_0$
  - $2 \times A = a_{n-1}\, a_{n-2} \ldots a_1 a_0 0$
- Similarly, multiplying by 2^k can be done by shifting left by k bit positions.
- Right shifts divide by powers of 2.

Circuit that Multiplies an 8-bit Unsigned Number by 3

A: $a_7 \cdots a_0$ $\qquad$ $P = 3A$



(a) Naive approach



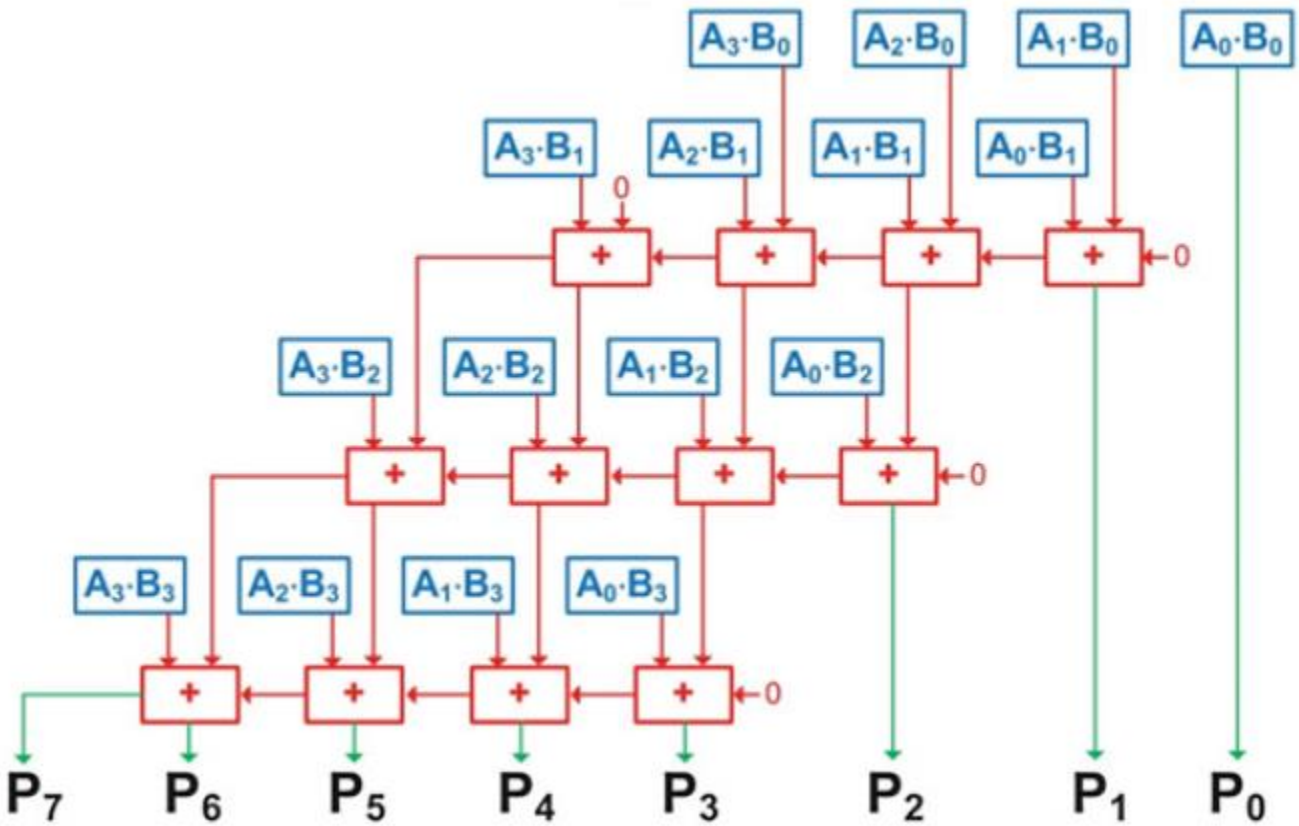
(b) Efficient design

Multiplication by hand

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	x 1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

4-bit Unsigned Multiplication in Hardware

$$
\begin{array}{r}
A_3 A_2 A_1 A_0 \\
\times \quad B_3 B_2 B_1 B_0 \\
\hline
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
\end{array}
$$



Signed Multiplication

- When performing multiplication on signed numbers, it is desirable to reuse the unsigned multiplier
- The process involves first identifying any negative numbers.
  - If a negative number is present, the two's complement is taken on it to produce its equivalent magnitude, positive representation.
  - The multiplication is then performed on the positive values.
  - The final step is to apply the correct sign to the product.
    - If the product should be negative due to one of the inputs being negative, the sign is applied by taking the two's complement on the final result.

Signed Multiplication

Step 1 – Take the two's complement of any negative inputs.

We notice this number is negative (-7_{10}) so we take its two's complement.

```
  1 0 0 1      0 1 1 1   ← +7₁₀
x 0 1 1 1    x 0 1 1 1
```

Step 2 – Perform the multiplication.

```
                 0 1 1 1   ← +7₁₀
x                0 1 1 1   ← +7₁₀
-----------------------
               10  1
             0 1 1 1
           10
           0 1 1 1
         1
         0 1 1 1
+  0 0 0 0
-----------------------
  0 0 1 1 0 0 0 1   ← +49₁₀
```

Step 3 – Apply the sign to the product (if applicable).

```
0 0 1 1 0 0 0 1   ← +49₁₀
        ↓ Two's complement
1 1 0 0 1 1 1 1   ← -49₁₀
```

Division

- Performing division is equivalent to subtracting the interim divisor from the dividend
  - If the subtraction is positive, then the divisor went into the dividend and the quotient is a 1.
  - If the subtraction yields a negative number, then the divisor did not go into the interim dividend and the quotient is 0.
- A multiplexer can be used to select whether the difference is used in the next subtraction (Q = 0), or if the interim divisor is simply brought down (Q = 1).

Unsigned Number Subtraction

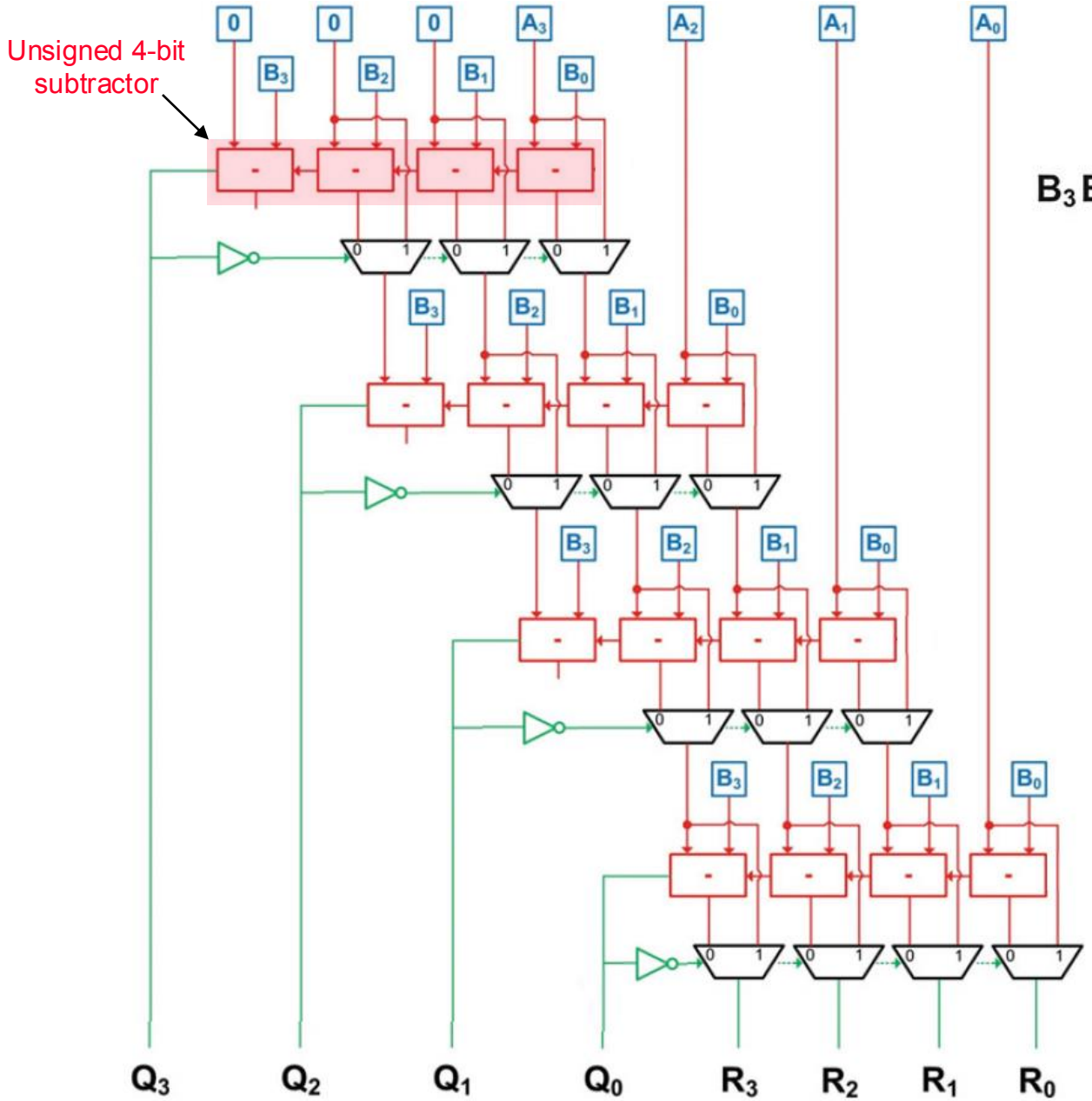
- Unsigned number **addition** is as simple as ordinary binary addition.
- However, the hardware implementation of borrow concept in subtraction is less efficient than the method that uses radix complements.

➢ The subtraction of two *n*-digit **unsigned numbers** $M - N$ in base r can be done as follows:

Algorithm

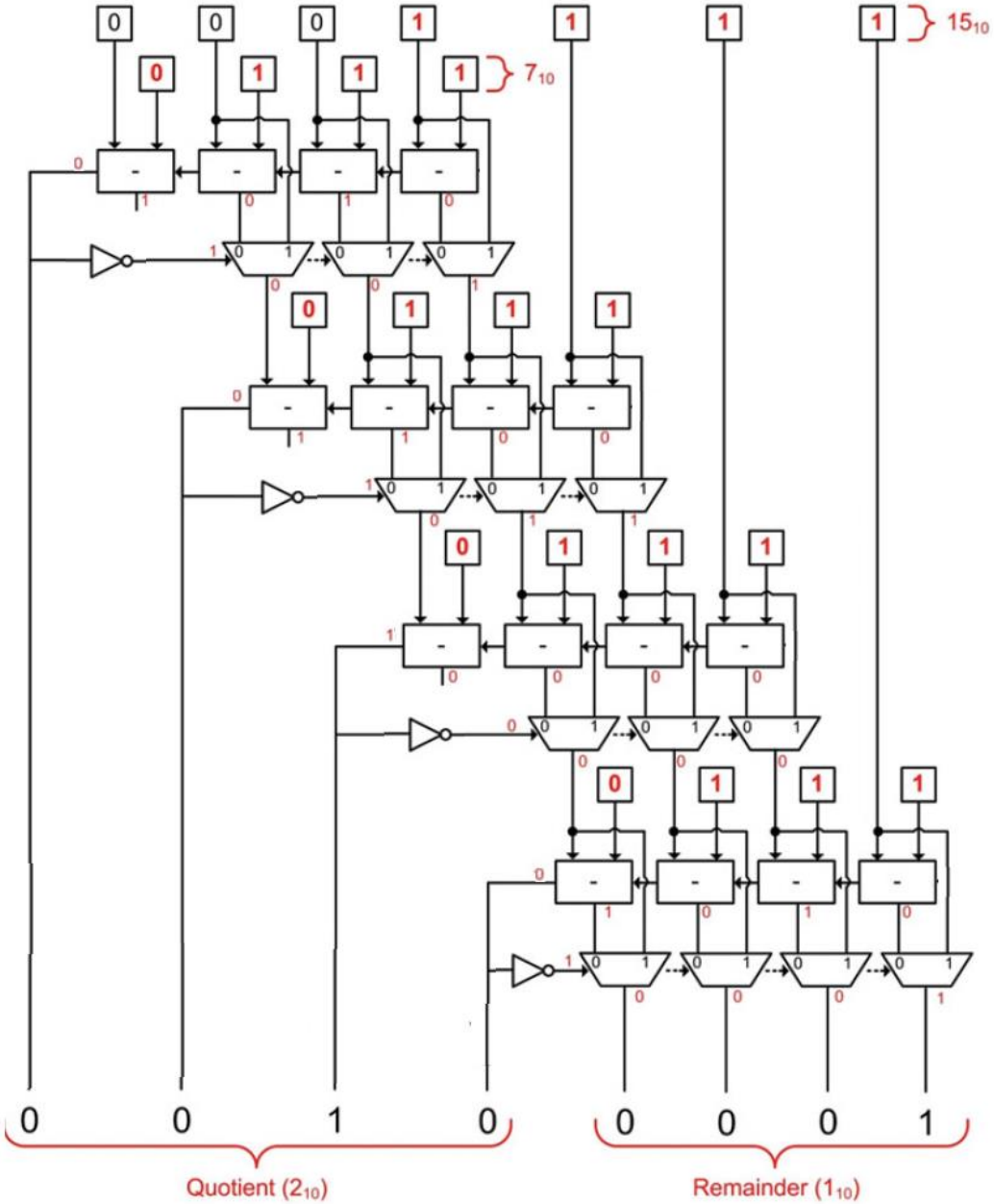
1. Add the minuend M to the r's complement of the subtrahend N. Mathematically, $M - N = M + (r^n - N) = M - N + r^n$.
2. If $M \geqq N$, the sum will produce and end carry r^n, which can be discarded; what is left is the result $M - N$.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

Design of an unsigned 4-bit divider using subtractors.



$$B_3B_2B_1B_0 \overline{\big)\, A_3A_2A_1A_0}^{\;\; Q_3Q_2Q_1Q_0}, \; R_3R_2R_1R_0$$

Design of an unsigned 4-bit divider using subtractors.



15 / 7 = 2 … 1

```
          0 0 1 0
0 1 1 1 ) 1 1 1 1
        - 0
          1 1
        - 0 0
          1 1 1
        - 1 1 1
          0 0 0 1
        - 0 0 0 0
          0 0 0 1
```