Ch. 6
Digital Arithmetic and Arithmetic Circuits

단원목차

Basic Digital Arithmetic

- 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
- 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 +.

Unsigned Binary Arithmetic

- Sum: Result of an Addition Operation of two (or more) binary numbers (operands).
- 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

$0 + 0 = \quad 0\ 0$

$1 + 0 = \quad 0\ 1$

$1 + 1 = \quad 1\ 0$

$1 + 1 + 1 = \quad 1\ 1$

CIN A B COUT SUM

Binary Addition Examples

- Binary Addition

1← Carry to next →1 1 1 1 1

1 0 0 1 0	1 0 1 0 1 1 1 0
+ 1 0 1 0	1 0 0 1 0 0 1 1
0 1 1 1 0 0	1 0 1 0 0 0 0 0 1

Carry Out Bit

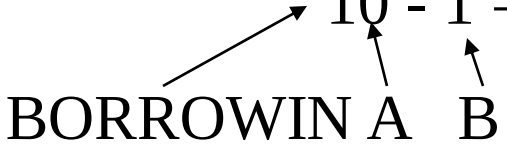
Basic Subtraction

- Basic Subtraction of X = A - B with A= Minuend, B= Subtrahend and X = Difference or result.
- Requires a Borrow Bit if A < B.
- There are other forms of subtraction such as 2s Complement Addition used by Microprocessors (such as in a PC).

Basic Subtraction Rules

- Binary 1 Bit Subtraction

0 - 0 =	0 0
1 - 0 =	0 1
1 - 1 =	0 0
10 - 1 =	0 1



Binary Subtraction Examples

- Binary Subtraction with Borrows ()

1110	110(10)	Borrow Stage
-1001	100 1	
	010 1	
10000	0111(10)	Borrow ripples to LSB
- 101	10 1	
	0101 1	

Signed Binary Numbers

- Sign Bit: A bit (usually the MSB) that indicates whether a number is positive(=0) or negative (=1).
- Magnitude Bits: The bits of a signed binary number that tell how large it is in value.
- True Magnitude Form: A form of signed binary whose magnitude bits are the TRUE binary form (not complements).

Signed Binary Numbers II

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

True Magnitude Form

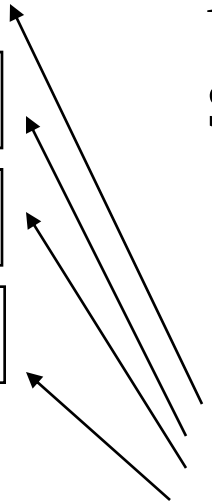
- 5 Bit Numbers Negative = S=1

+25 = 011001

-25 = 111001

-12 = 101100

+12 = 001100



Note sign bit = MSB = S = 0

Same as +25 with S=1

True magnitude

1s Complement Form

- 8 Bit 1s Complement (Negative = S = 1)

57 = 0011 1001

-57 = 1100 0110 All Bits Inverted

72 = 0100 1000

-72 = 1011 0111

2s Complement Form

- Used in MPU (PC) Arithmetic

57 = 0011 1001

-57 = 1100 0110

\+ 1

1011 1000

-72 = 1011 0111

\+ 1

1011 1000

Signed Binary Addition (8 Bit)

- Signed Addition Positive S = 0

+30 = 0001 1110

+75 = 0100 1011

+105= 0110 1001

Similar to Binary addition with a Sign bit

Subtraction with 1s Complement

- Add the 1s Complement and then Carry

+80 = 0101 0000 (+80) = 0101 0000

-65 => 0100 0001 (+65) = 1011 1110 (1s Comp 65)

10000 1110

\+ ⌊→ 1(end-around carry)

0000 1111 (+15)

Uses an End around carry addition method

2s Complement Subtraction

- Add 2s Complement to Minuend

+80 = 0101 0000 0101 0000

+65 = 0100 0001 +1011 1111

-65 = 1011 1110 +1 1 0000 1111

Discard Carry Bit from result

Negative Results

- If the True Magnitude Form is used for subtraction the results are incorrect. (refer to p228)
- If the result is from 1s or 2s Complement and the result is negative (S=1) the magnitude is found by taking the complement of the result.

Negative Result Example

- 2s Complement Negative Result (65-80)

+65 = 0100 0001 0100 0001

-80 = 1011 0000 (2s C.) +1011 0000

결과의 부호비트가 1이면 그것은 음수이고 2의 보수 형태이다.

1111 0001

Invert 0000 1110

Add 1 + 1

Final Result = -15 0000 1111 = 15(Neg.)

Range of Signed Numbers

- Range of Positive Numbers is 0 to $2^N - 1$ for an N Magnitude Bit Number.
- Range of Negative Numbers is -1 to -2^N for an N Magnitude Bit Number.
- 8 Bit Example

8 Bit Number Range is $-2^N <= X <= +2^N - 1$

or -128 to +127

Sign Bit Overflow

- Overflow: A erroneous carry into the sign bit of a signed binary number that 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.

False Negative Overflow

- 8 Bit Addition

+75 = 0100 1011

+96 = + 0110 0000

1010 1011 Result is Negative (False)

Two Positive Numbers added with a result greater than the range of +127 for eight bit numbers causes an overflow.

False Positive Overflow

- Addition of two 8 Bit Negative Numbers

-80 = 1011 0000

-65 = +1011 1111

0110 1111 Result is Positive(False)

Two Negative Numbers were added to produce a False Positive Result due to an overflow of negative range of 8 bit numbers (-128).

Hexadecimal Addition

- Similar to decimal addition with a range of digits of 0 to 9 and A to F.
- Examples

F + 1 = 10

F + F = 1E

BCD 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 0-9 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

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

Excess 3 Code

- A BCD Code formed by adding 3 (0011) to its true four bit binary value.
- Excess 3 is a self complementing code. If the bits of the Excess-3 digit are inverted, they yield the 9's complement of the decimal equivalent.
- Excess-3 code is useful for performing decimal arithmetic digitally.

Excess 3 Examples

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

Gray Code

The Gray Code
is unweighted and not an arithmetic code; that is, there are no specific weights assigned to the bit positions. The important feature of the Gray code is that
It exhibits only a single bit change from one code number to the next.
→ Shaft position encoder

4-bit Gray code

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

Gray Code Conversion

Binary-to-Gray code conversion

1. MSB in the Gray code is the same as MSB in the binary number.
2. Going from left to right, add each adjacent pair of binary code bits to get the next Gray code bit. Discard carries.

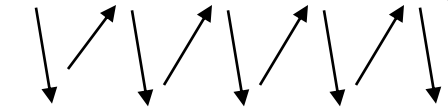
Ex) 1 0 1 1 0 (Binary)

1 1 1 0 1 (Gray)

Gray-to-Binary code conversion

1. MSB in the Binary code is the same as MSB in the Gray code.
2. Add each binary code bit generated to the Gray code bit in the next adjacent position. Discard carries.

Ex) 1 1 0 1 1 (Gray)



1 0 0 1 0 (Binary)

ASCII Code

- American Standard Code for Information Interchange.
- A seven bit alphanumeric code used to represent text letters, numerals, punctuation, and special controls.
- An expanded 8 bit form is becoming more widespread.
- Refer to Table 6.5

Binary Adders

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

Basic HA Addition

- Binary Two Bit Addition Rules

0 + 0 = 00

0 + 1 = 01

1 + 1 = 10

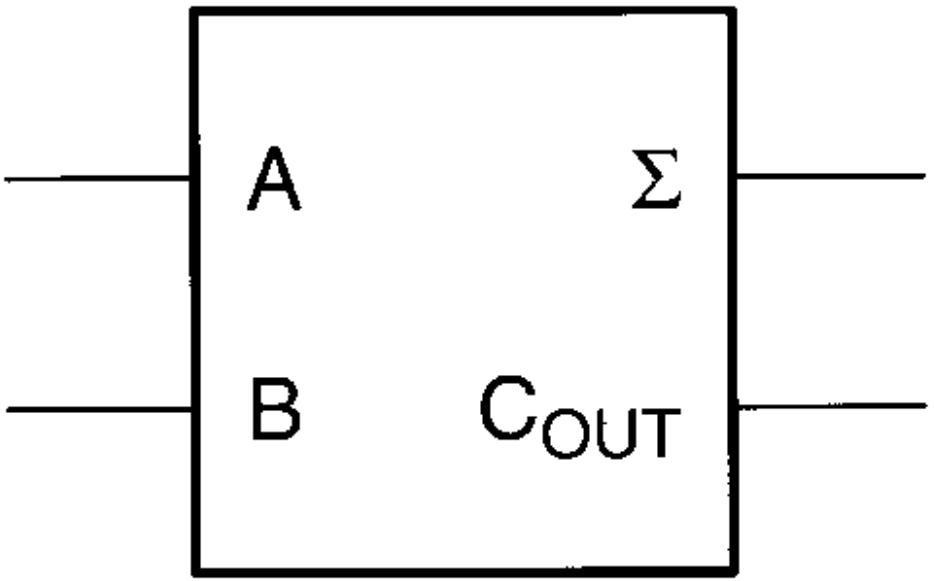
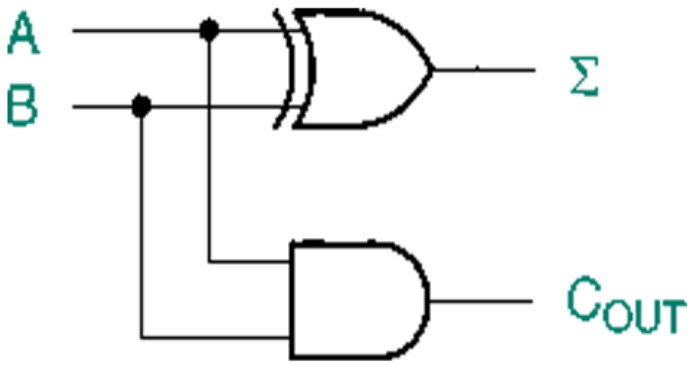


Figure 6.1
Half Adder

HA Circuit

- Basic Equations S = A xor B, C = A and B where S = Sum and C = Carry.
- Truth Table for HA Block

Figure 6.2



A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

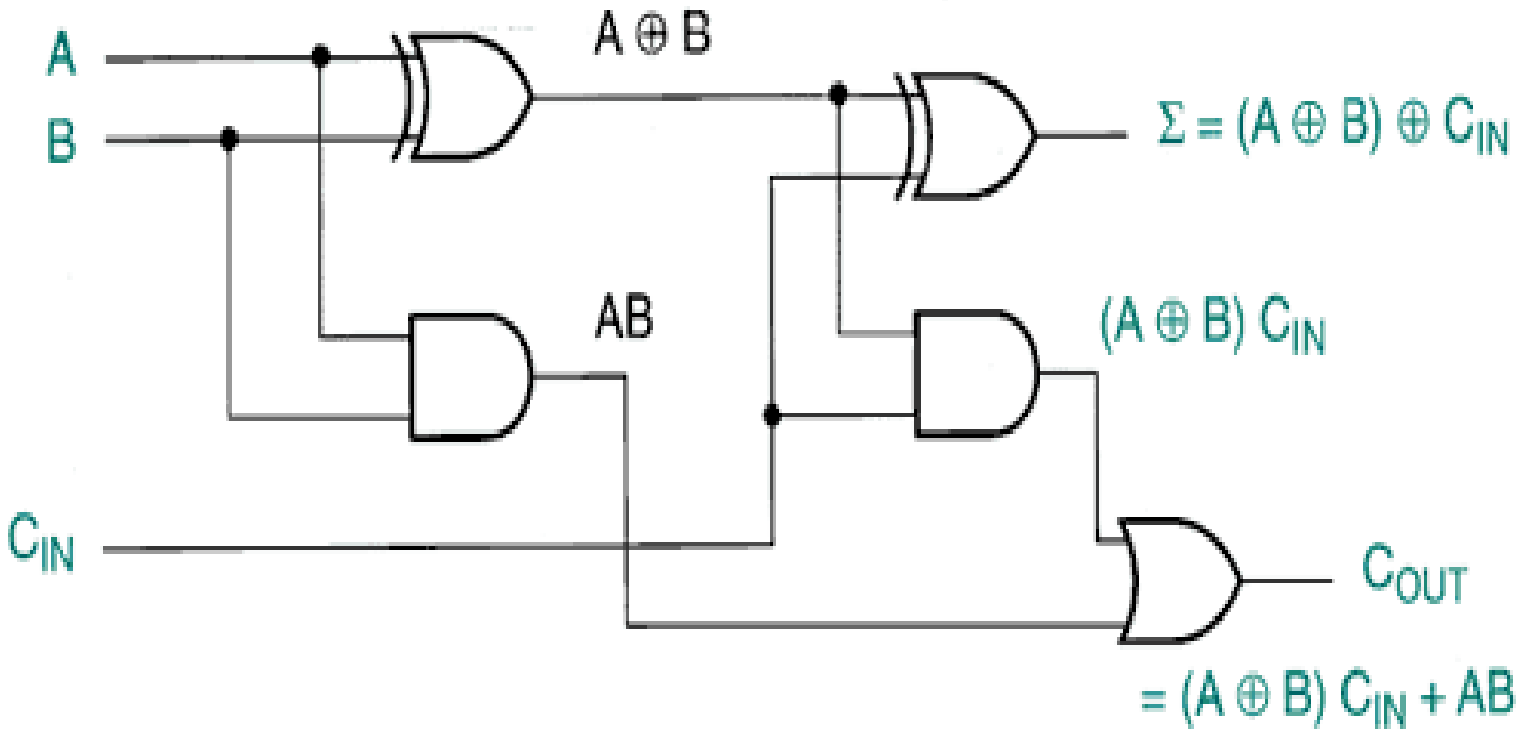
Full Adder Basics

- Adds a CIN input to the HA block.
- Equations are modified as follows.
- C = ((A xor B) and CIN) or (A and B).
- S = A xor B xor CIN.
- A FA can be made from two HA blocks and an OR Gate.

FA Circuit

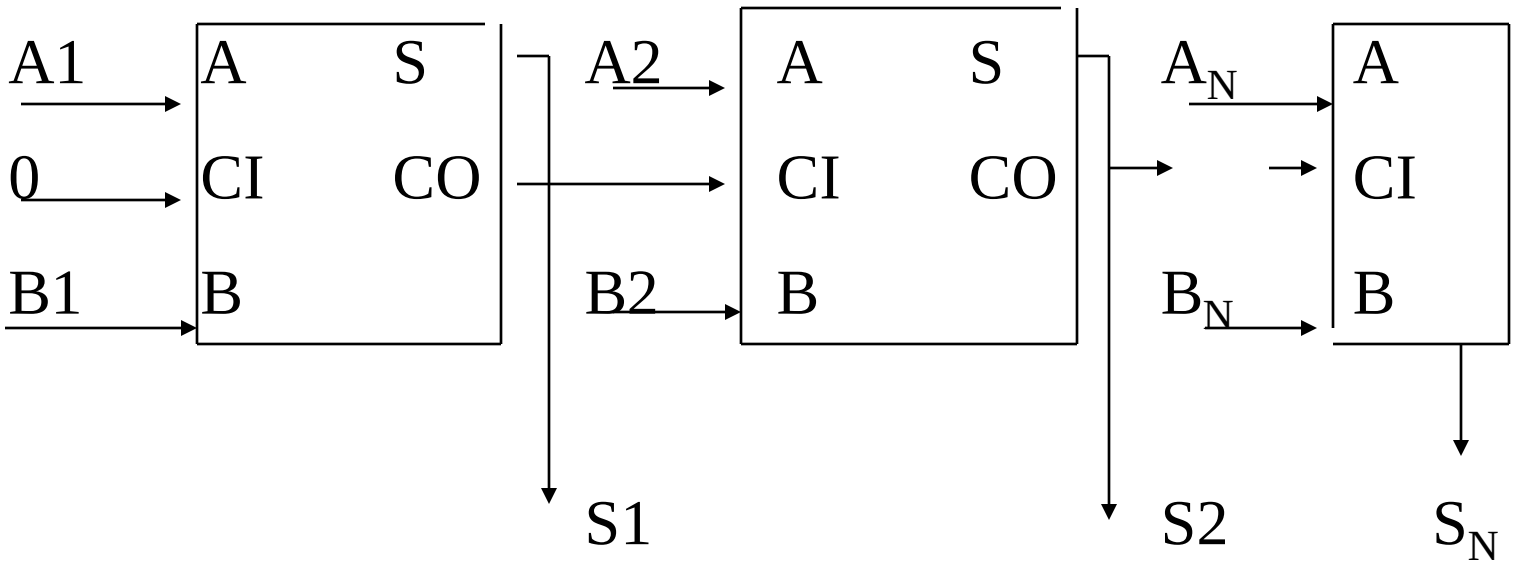
- Two HA Full Adder Circuit.

Figure 6.6



Parallel Adders

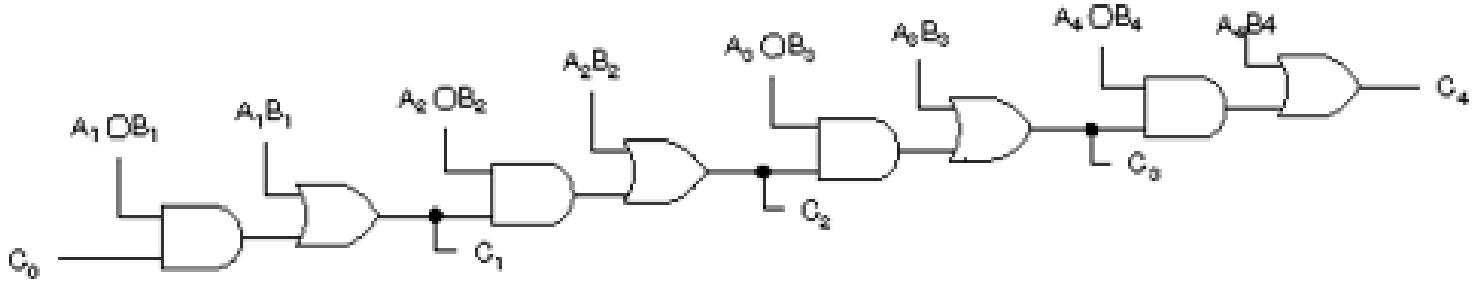
- N-Bit Multiple Adder (FA Stages)



Ripple Carry I

- In the N-Bit Parallel Adder (FA Stages) the Carryout is generated by the last stage (FAN).
- This is called a Ripple Carry Adder because the final carryout (Last Stage) is based on a ripple through each stage by CIN at the LSB Stage.

Figure 6.11



Ripple Carry II

- Each Stage will have a propagation delay on the CIN to COUT of one AND Gate and one OR Gate.
- A 4 Bit Ripple Carry Adder will then have a propagation delay on the final COUT of 4 X 2 = 8 Gates.
- A 32 Bit adder such as in a MPU in a PC could have a delay of 64 Gates.

Look Ahead Carry I

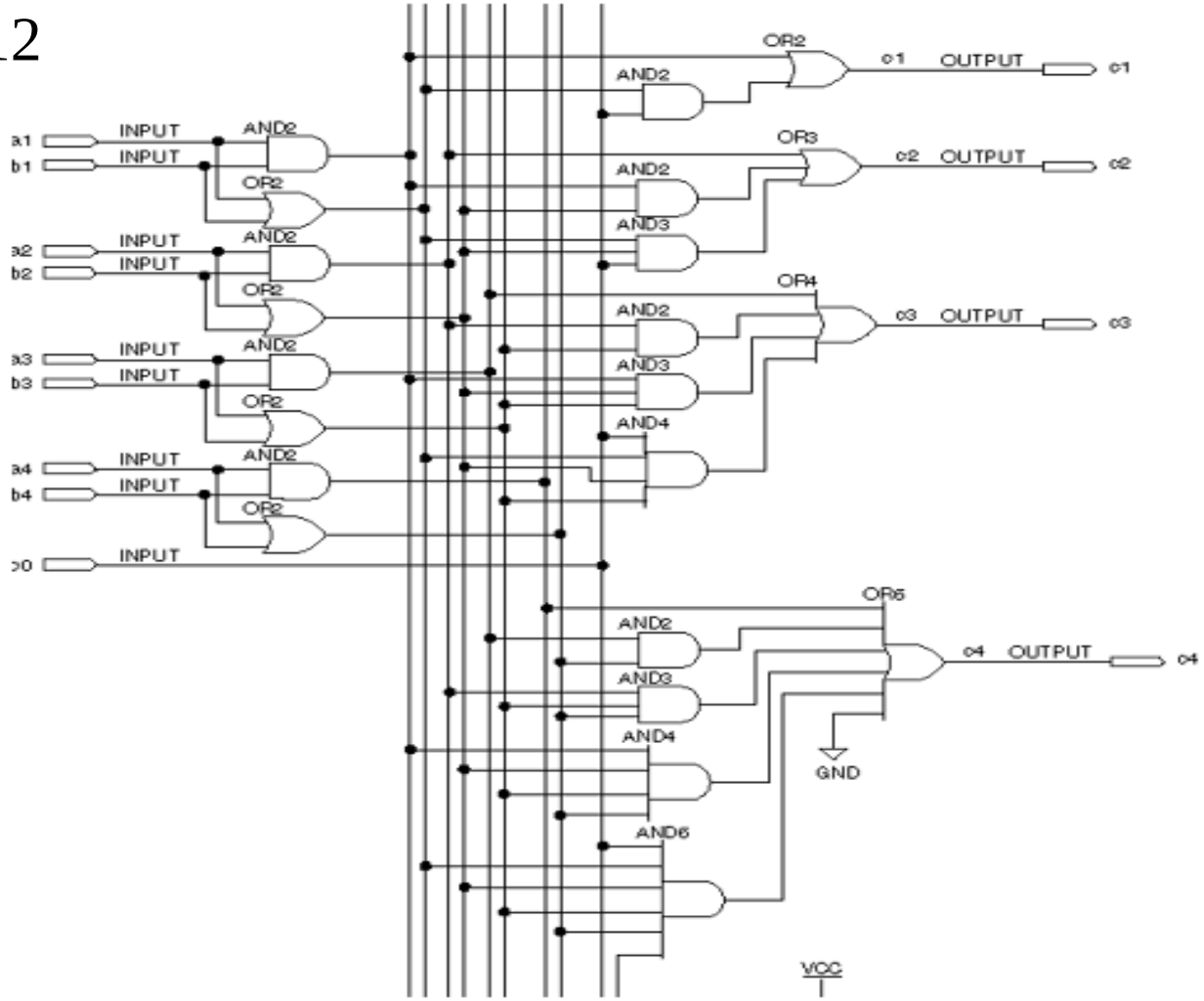
• Fast Carry or Look Ahead Carry: A combinational network that generates the final COUT directly from the operand bits (A1 to AN, B1 to BN). It is independent of the operations of each FA Stage(as the ripple carry is).

Look Ahead Carry II

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

4-bit Fast Carry Circuit

Fig. 6.12



Subtractor (2s Complement) I

- The concept of Subtraction using 2s Complement addition allows a Parallel FA to be used.
- This could be used in a MPU ALU (Arithmetic Logic Unit) for Subtraction.
- The subtract operation involves adding the inverse of the subtrahend to the minuend and the add a 1.

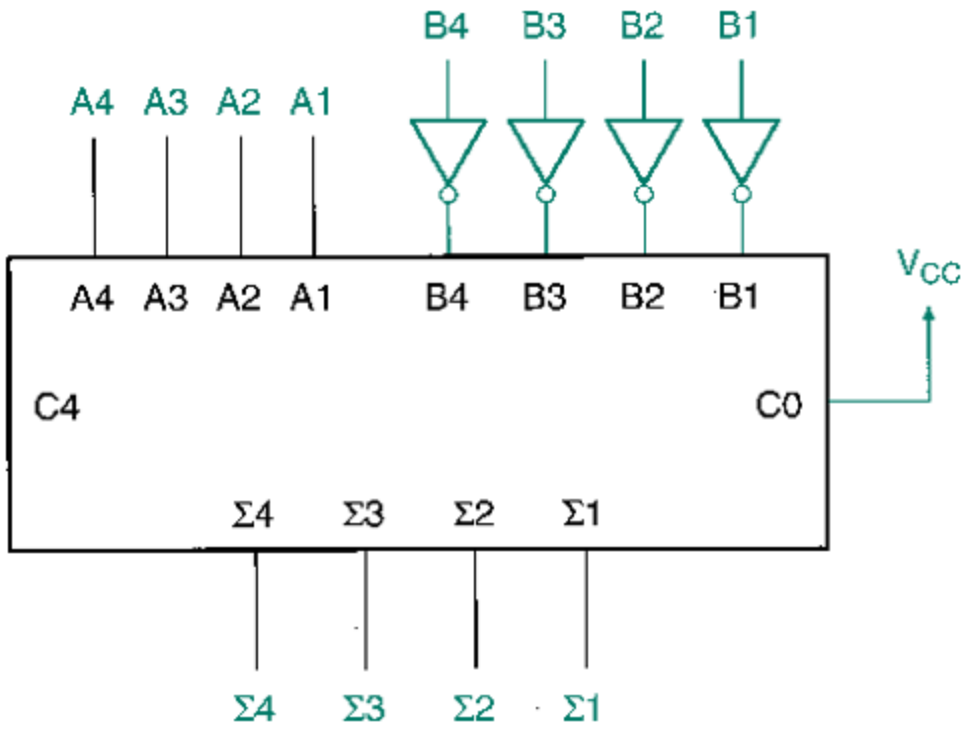
2s Complement Subtractor

Ex) 0101-0011
2's comp. Of 0011
0011
1100(1's)
+1
1101

0101(+5)
+ 1101(-3)
1 0010(+2)

↳ discard carry

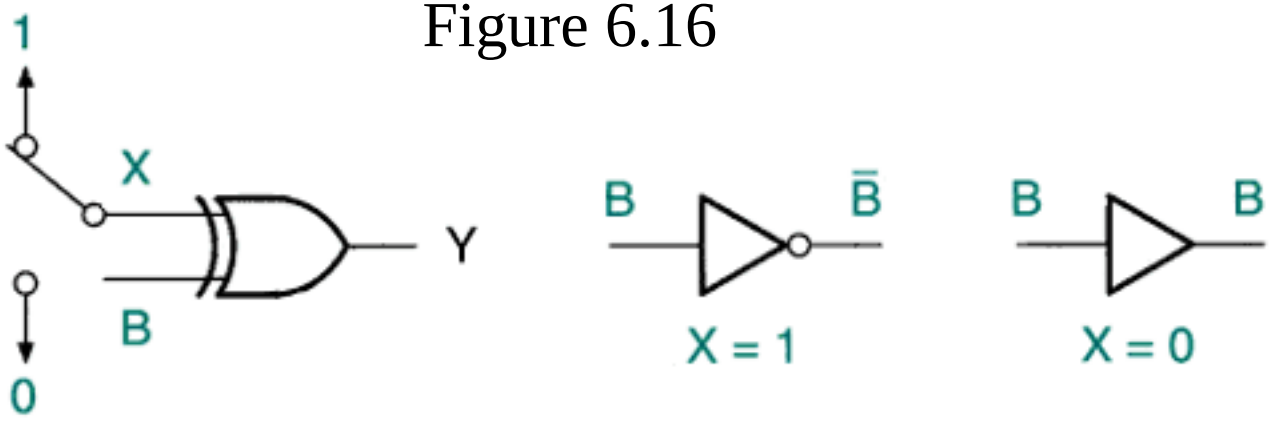
Fig. 6.14



Subtractor (2s Complement) II

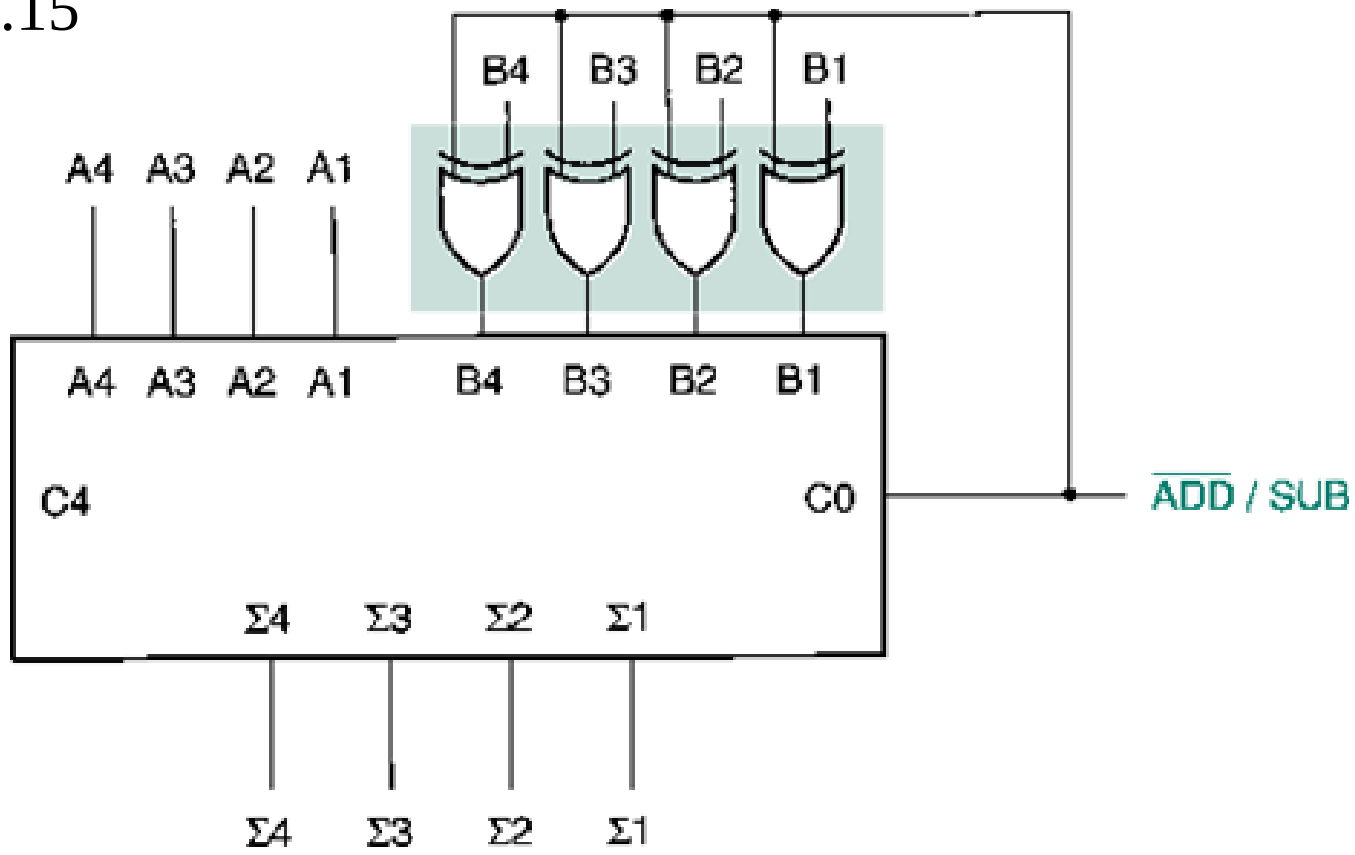
- Difference = A – B = A + !B + 1.
- This operation can be done in a parallel N-Bit FA by Inverting (B1 to BN) and connecting CIN at the LSB Stage to +5V.
- The circuit can be modified to allow either the ADD or SUBTRACT Operation to be performed.

Figure 6.16



2s Complement Adder/Subtractor

Fig. 6.15



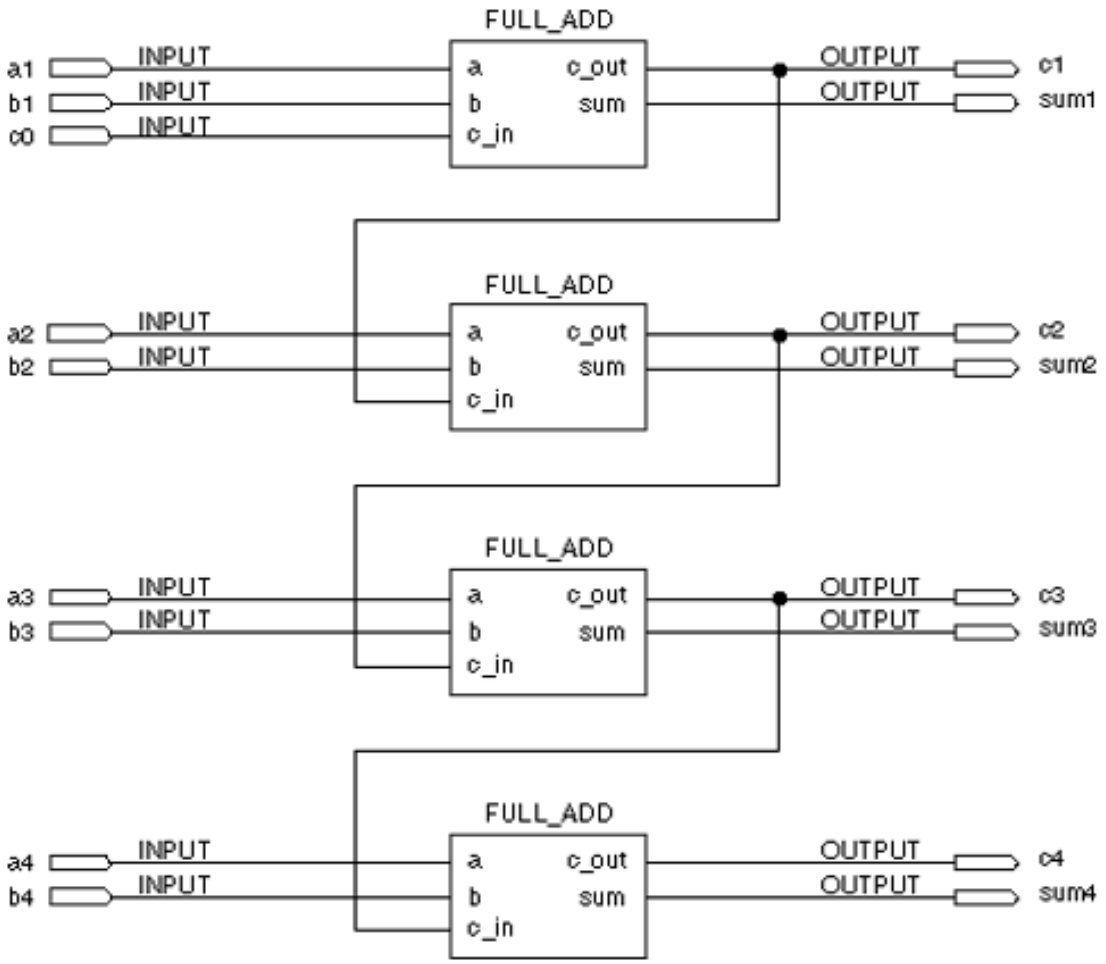
Structured VHDL Design

- Hierarchy: A group of design entities associated in a series of levels (the hierarchy) in which complete designs form portions or subsections of the upper design.
- Component: A complete VHDL Design Entity that can be used as part of a higher level file in a hierarchical design.

Structured VHDL Design

- Port: An Input or Output of a VHDL design entity or component.
- Component Declaration Statement: A statement that defines the I/O Port Names of a component 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.
- The following slide will illustrate a Full Adder VHDL Design using a Single FA Component.

Figure 6.13 (4-bit parallel adder with ripple carry)



Full Adder VHDL

- Basic Full Adder (1 Bit Add plus carry)

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

Hierarchical VHDL Design Flow I

1. A separate component file for a full adder(full_adder.vhd), saved in a folder where the complier 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 component 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);
```

Hierarchical VHDL Design Flow II

```
BEGIN
      component instantiation(s);
      other statements;
END arch_name;

-- component declaration statement template
COMPONENT __component_name
      GENERIC(__parameter_name : string := __default_value;
            __parameter_name : integer := __default_value);
      PORT(
            __input_name, __input_name                  : IN      STD_LOGIC;
            __bidir_name, __bidir_name                  : INOUT
            STD_LOGIC;
            __output_name, __output_name      : OUT    STD_LOGIC);
END COMPONENT;
```

Hierarchical VHDL Design Flow III

```
-- component instantiation statement template
__instance_name: __component_name
      GENERIC MAP (__parameter_name => __parameter_value ,
                __parameter_name => __parameter_value)
      PORT MAP (__component_port => __connect_port,
               __component_port => __connect_port);
```

Four Bit Parallel Adder

Entity

- Four Bit Parallel Adder IO Pins

```
ENTITY add4par IS
        PORT(  C0      :IN BIT;
               A, B    :IN BIT_VECTOR(4 downto 1);
               C4      : OUT BIT;
               SUM     :OUT BIT_VECTOR(4 downto 1));
END add4par;
ARCHITECTURE adder OF add4par IS
```

4 Bit Parallel Adder Component (1 Bit FA)

- FA Component used for Parallel Adder

```
-- component declaration statement
COMPONENT full-add        -- Previous FA Design File
PORT( a, b, c_in          :IN BIT;
      c_out, sum          :OUT BIT);
END COMPONENT
      SIGNAL   c   :BIT_VECTOR(3 downto 1)
-- Internal signal used for intermediate carries
```

4 Bit Parallel Adder

- This design uses the 1 Bit FA as a Component to create the 4 Bit Parallel Adder.
- The basic Adder Component is mapped four times uses a Component Instantiation such as adder1, adder2, etc.
- The connections are set as a Port Map for each instance of the component.

4 Bit Adder Structured Architecture I

- Mapping Type of Architecture

```
BEGIN
-- 4 component instantiation statements
adder1: full_add   -- This defines the first component
        PORT MAP(a => A(1), b=> B(1), c_in => C0,
                            c_out => C(1), sum=> SUM(1));
adder2: full_add   -- This defines the second component
        PORT MAP(a => A(2), b=> B(2), c_in => C(1),
                            c_out => C(2), sum=> SUM(2)):
adder3: full_add   -- This defines the third component
        PORT MAP(a => A(3), b=> B(3), c_in => C(2),
                            c_out => C(3), sum=> SUM(3));
```

4 Bit Adder Structured Architecture II

- Remaining part of the Architecture

```
adder4: full_add   -- This defines the fourth component
        PORT MAP(a => A(4), b=> B(4), c_in => C(3),
                 c_out => C4, sum=> SUM(4));
END adder;
-- another component instantiation statement
adder1 : full_add PORT MAP (a(1), b(1), c0, c(1), sum(1));
adder2 : full_add PORT MAP (a(2), b(2), c(1), c(2), sum(2));
adder3 : full_add PORT MAP (a(3), b(3), c(2), c(3), sum(3));
adder4 : full_add PORT MAP (a(4), b(4), c(3), c4, sum(4));
-- use the correct order originally defined
```

Other Structures

- The preceding example directly mapped the 4 FA Components.
- Another approach would be to use a Repetitive Loop in VHDL called a Generate Statement.
- This is similar to a FOR Loop with an index of I.
- Both types are still ripple carry adders.

Generate Statement I

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

-- GENERATE statement template

__generate_label:

FOR __index_variable IN __range GENERATE

    __statement;

    __statement;

END GENERATE;

-- 4-bit adder VHDL code using generate statement

```
ENTITY add4gen IS
        PORT(  c0   : IN BIT;
               a, b  : IN BIT_VECTOR(4 downto 1);
               c4   : OUT BIT;
               sum : OUT BIT_VECTOR(4 downto 1));
END add4gen;
ARCHITECTURE adder OF add4gen IS
```

Generate Statement II

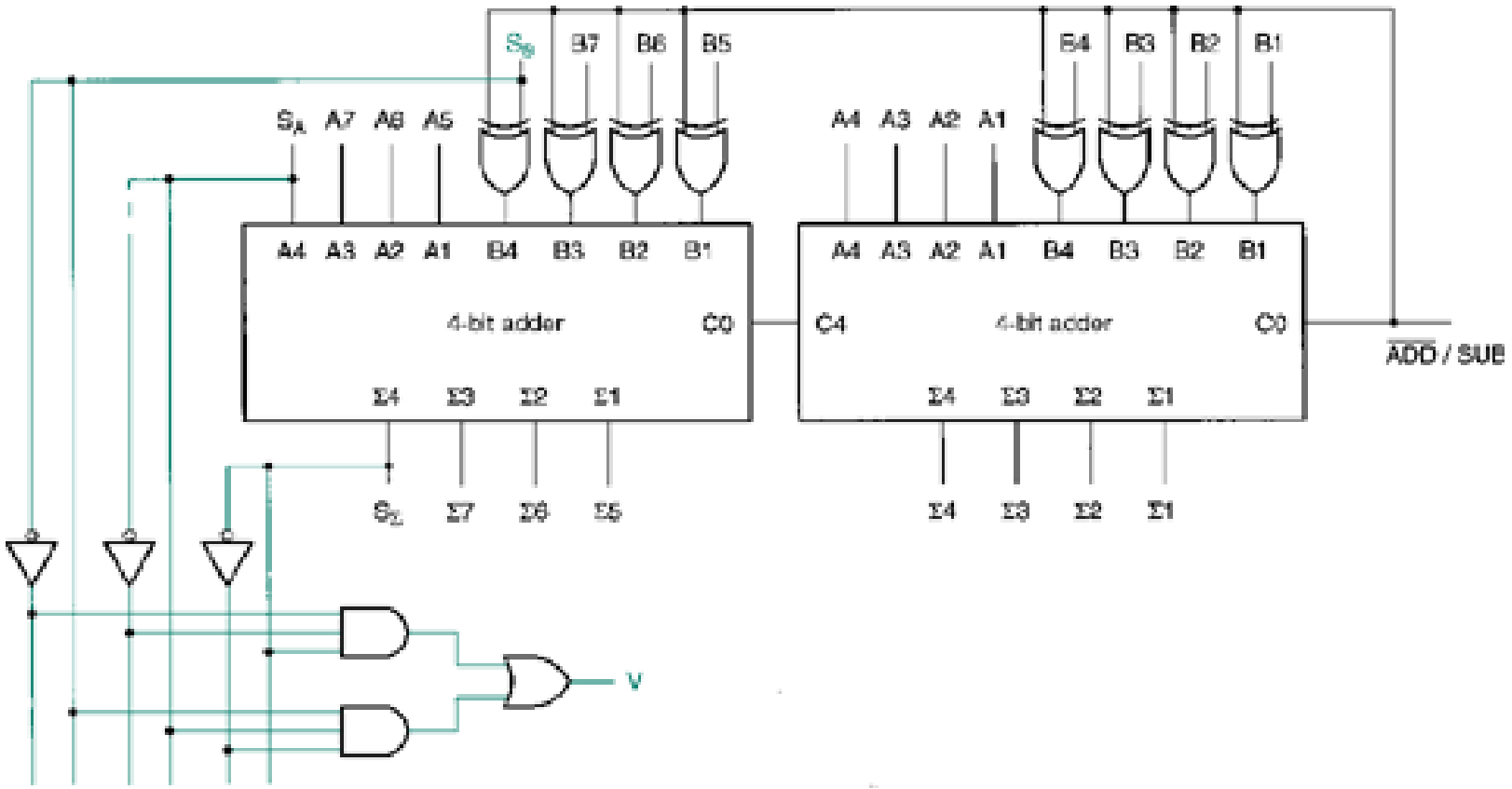
```
-- COMPONENT declaration
COMPONENT full_add  -- Previous FA Design File
            PORT( a, b, c_in          :IN BIT;
            c_out, sum    : OUT BIT);
END COMPONENT
            SIGNAL    c  : BIT_VECTOR(4 downto 0)
BEGIN
            c(0)  <=  c0;
            adders :
            FOR  i  IN  1 to 4 GENERATE  -- easily expandable by changing the range
               adder : full_add PORT MAP (a(i), b(i), c(i-1), c(i), sum(i);
            END GENERATE;
            c4 <= c(4);
END adder;
```

Overflow Detection

1. Check the sign-bits
   - 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.
   - refer to table 6.9

2. Check the carry-bits
   - $V = C_n$ XOR $C_{n-1,}$
   - If V=1, then overflow (Fig. 6.22)

Fig. 6.21 8-bit Adder with Overflow Detector



BCD Adder

- 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 BIN to BCD Code Converter.

Fig. 6.23 BCD Adder (1 1/2 Digit Output)

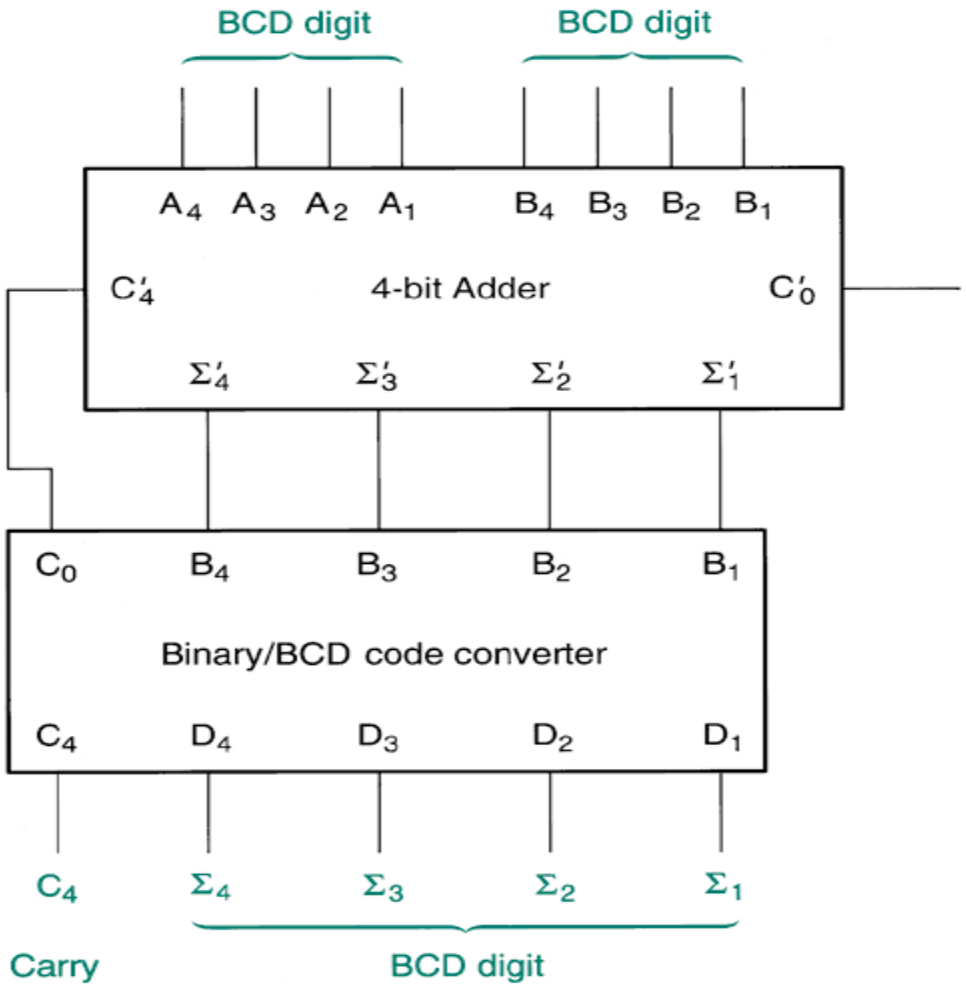
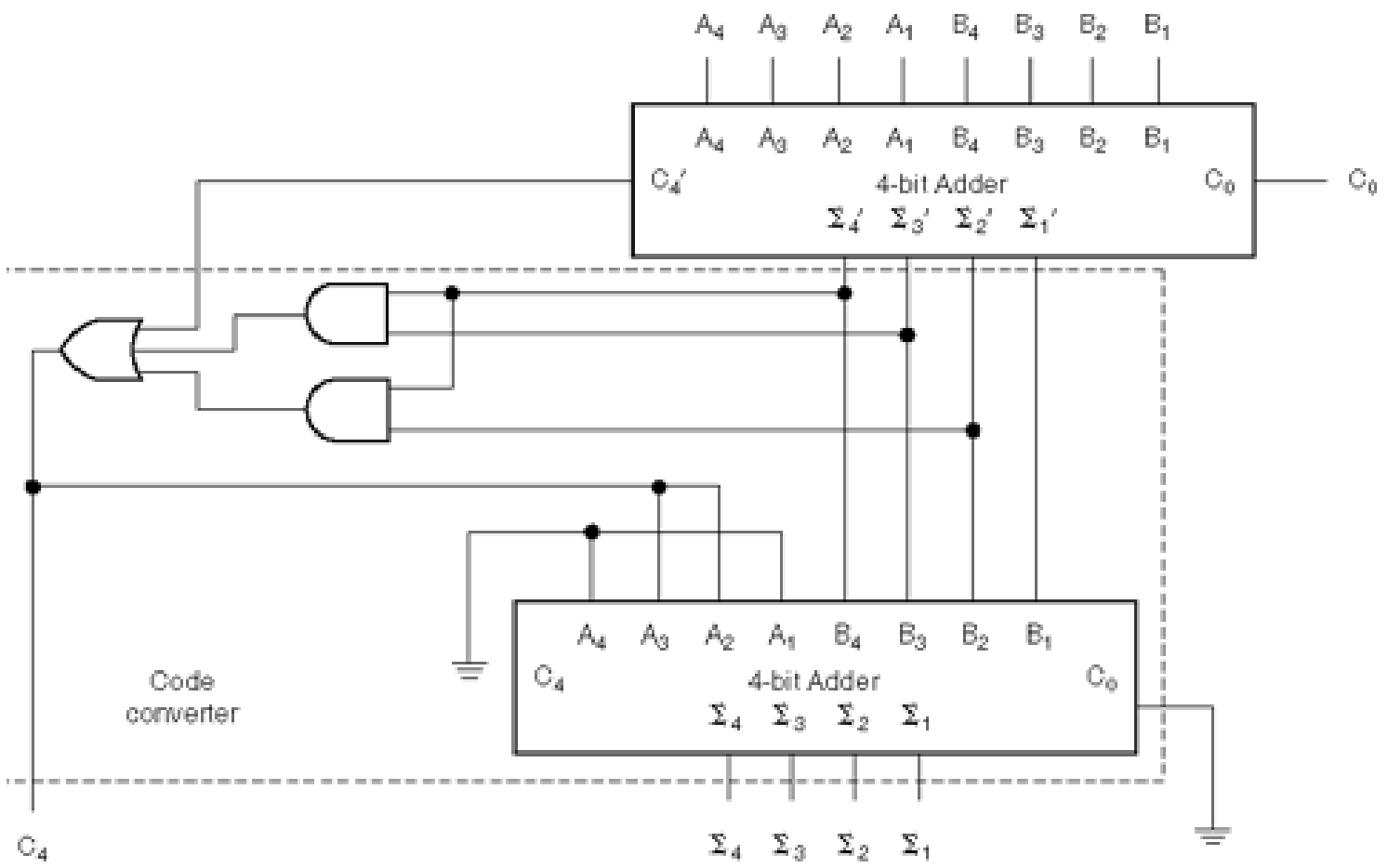
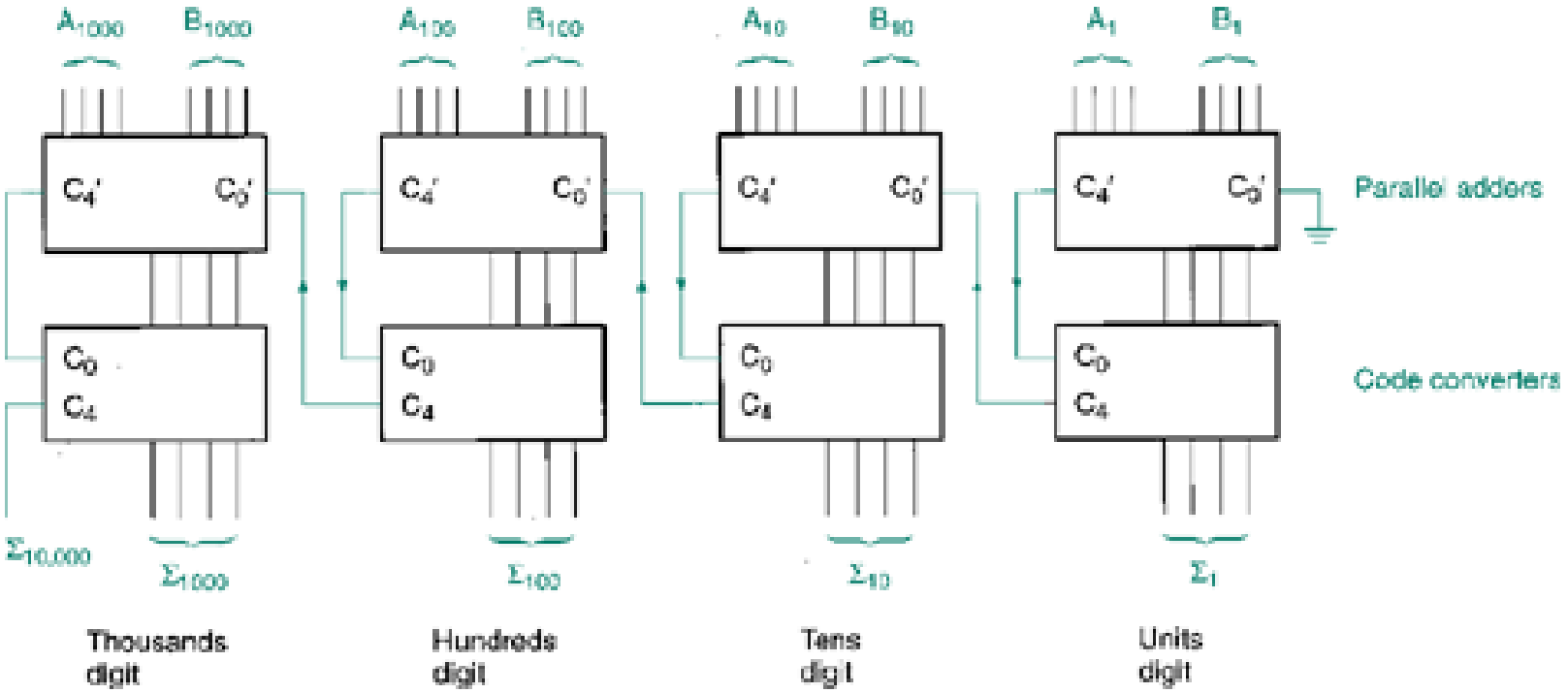


Fig. 6.26 BCD Adder



Multiple-Digit BCD Adders

Fig. 6.27 4 ½ Digit BCD Adder



BCD Code Converter

- Code Converter designed is based on the 4 Bit Adder used with Table 6.10 in the text.
- The complete design is shown in Figure 6.26.
- The Ai Inputs of the Code Converter Adder are fixed to be either a 0000 (C4 =0) or 0110 (C4=1). The 0110 corrects Binary Overflow to BCD.

Carry Generation in MAX+PLUS II

- Global Project Logic Synthesis

1. FAST : A fast synthesis but a large gate size
2. NORMAL : A default synthesis
3. WYSIWYG : As synthesis a design without altering design format as possible

Refer to Fig. 6.30 & Fig. 6.31

=> We do not need to design an adder circuit to have a fast carry function

- EXP(expander buffer) : shared logic expander in the same LAB

SUMMARY I

• ***A Hierarchical VHDL Design***

To use a component in a VHDL design hierarchy, we require a design entity that defines the component declaration and component instantiation statements. For example:

```
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;
```

• ***A Simple Port Map***

If all ports of a component are to be used in the same order as in the component definition in the original component design entity, the port map can simply contain the user names in the same order. For example:

```
Adder1 : full_add
  PORT MAP ( a(1), b(1), c0, c(1), sum(1));
```

SUMMARY II

• ***A Explicit Port Map***

If only a portion of the component ports are to be used or they are not used in the same sequence as they are declared, the port map must be more explicit. For example:

```
adder1: full_add
   PORT MAP (a => A(1), b=> B(1), c_in => C0,
                c_out => C(1), sum=> SUM(1));
```

• ***A Generate statement***

can be used to instantiate multiple instances of a component. For example:

```
label:
FOR index_variable IN range GENERATE
   statement;
END GENERATE;
```

• ***A Synthesis Strategy***

If you use ‘normal’ option, MAX+PLUS II will synthesize an adder to minimize carry delays.