DIGITAL SYSTEM DESIGN

Basic Arithmetic and Logic

Waqar Ahmad

Decimal (base 10)	Binary (Base 2)	Octal (Base 8)	Hexadecimal (Base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Convert Decimal to Binary (Integer Part)

Example: 50 (divide by 2)

Convert Decimal to binary (Fraction Part)

Example: 0.625 (multiply by 2)

Convert Decimal to Binary (Integer and fraction)

Example:

Task:

$(41)_{10}$ to $(bbbb)_2$

$(0.6875)_{10}$ to $(bbbb)_2$

$(41.6875)_{10}$ to $(bbbb)_2$

(101001.1011) to $(ddd)_{10}$

$(101001)_2$

$(0.1011)_2$

$(101001.1011)_2$

Binary Codes for Decimal Digits

Decimal Digit	BCD (8421)	Excess-3	84-2-1	2421	(Biquinary) 5043210
0	0000	0011	0000	0000	0100001
1	0001	0100	0111	0001	0100010
2	0010	0101	0110	0010	0100100
3	0011	0110	0101	0011	0101000
4	0100	0111	0100	0100	0110000
5	0101	1000	1011	1011	1000001
6	0110	1001	1010	1100	1000010
7	0111	1010	1001	1101	1000100
8	1000	1011	1000	1110	1001000
9	1001	1100	1111	1111	1010000

Binary Logic and Binary Arithmetic

- An arithmetic variable may consist of many digits
- A logical variable is either 1 or 0

- Arithmetic operation 1 + 1 = 10
- Logical operation 1 + 1 = 1

Logic Gates Truth Table

AND			**OR**			**NOT**	
x	y	**x.y**	x	y	**x+y**	x	**x′**
0	0	**0**	0	0	**0**	0	**1**
0	1	**0**	0	1	**1**	1	**0**
1	0	**0**	1	0	**1**		
1	1	**1**	1	0	**1**		

Fixed-Point Arithmetic

Binary Numbers

- Unsigned Numbers
- Signed Numbers
  - Signed Magnitude
  - One’s Complement
  - Two’s Complement

Unsigned Magnitude

- Only positive number are presented
- No sign bit
- For N bits we can represent the signed integers between 0 and 2^N -1
- Example: 111 -> 7

Addition and Multiplication of Unsigned Numbers

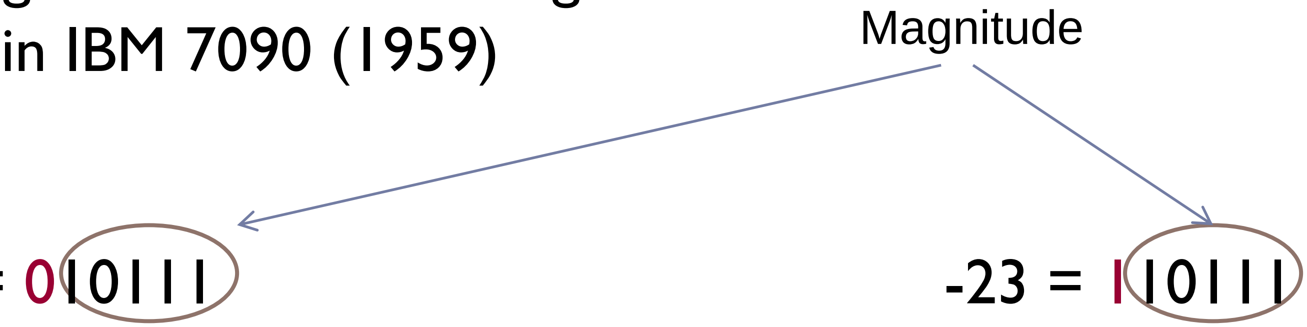
- We can add and multiply two binary words in a straightforward fashion.
  - Because all the numbers are positive, the results of addition or multiplication are also positive.

- However, the result of adding two *N*-bit words in general results in an *N*+1 bits.
  - When the result cannot be represented as an *N*-bit word, we say that an **overflow** has occurred

- In general, the result of multiplying two *N*-bit words is a 2*N* bit word.

Signed Magnitude

- The most significant bit is used to represent the sign: "1" means negative, "0" means positive
- For positive numbers the result is the same as unsigned magnitude representation
- For N bits we can represent the signed integers between $-2^{(N-1)}+1$ and $+2^{(N-1)}-1$
- Problems: +0 and -0 (e.g. 0000, 1000)
- Adding +K and –K doesn't give zero.
- Used in IBM 7090 (1959)



Magnitude

- +23 = 010111

-23 = 110111

1's Compliment

- Negative binary number is the complement (bit-wise NOT) of its positive counterpart
- +23 = 010111 -23 = 101000
- Two representations of 0 (0000, 1111)
- For N bits we can represent the signed integers between $-2^{(N-1)}+1$ and $+2^{(N-1)}-1$
- For 8 bits, 127 (01111111) to -127 (1000000)
- Used in old computers, such as PDP-1 (1960) and CDC 160A (1965)

- For binary addition, carry must be added back to the sum.

2’s Complement

- A specific case of *radix complement*
- To negate or *complement* an N–digit number, subtract it from 2^N. i.e. (-x) represented as (2^N – x)

- If you then add the number and its complement, you get 2^N.
- If you only keep N-digits (discard final carry), you have zero.

- Makes subtraction easy

2’s Complement

- 7 is 0111
- -7 is computed as 16-7=9
- 10000 - 0111 = 1001 = -7
- Add –7 and 7, discard carry, we get 0
- 0111 + 1001 = 1_0000

2’s Complement

The unified representation of both the negative and positive 2’s complement number is:

$$\boldsymbol{a} = -2^{N-1}a_{N-1} + \sum_{i=0}^{N-2} a_i 2^i$$

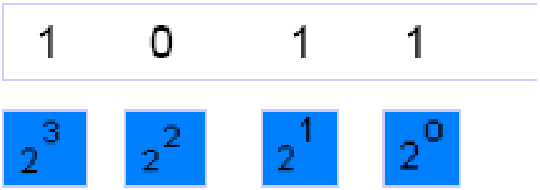
Equivalent unsigned representation of a negative No. is

$$\boldsymbol{2^N - |a|}$$

2’s Complement

Example:

Unsigned Number

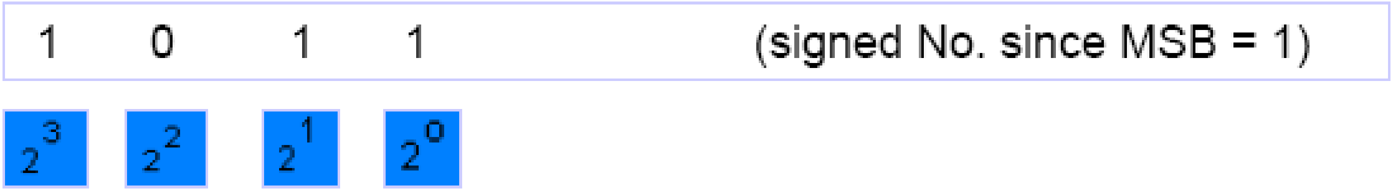

1 0 1 1

2^3 2^2 2^1 2^0

8 4 2 1 (weights)

8+2+1 = 11 (Decimal Equivalent)

2’s Compliment Number


1 0 1 1 (signed No. since MSB = 1)

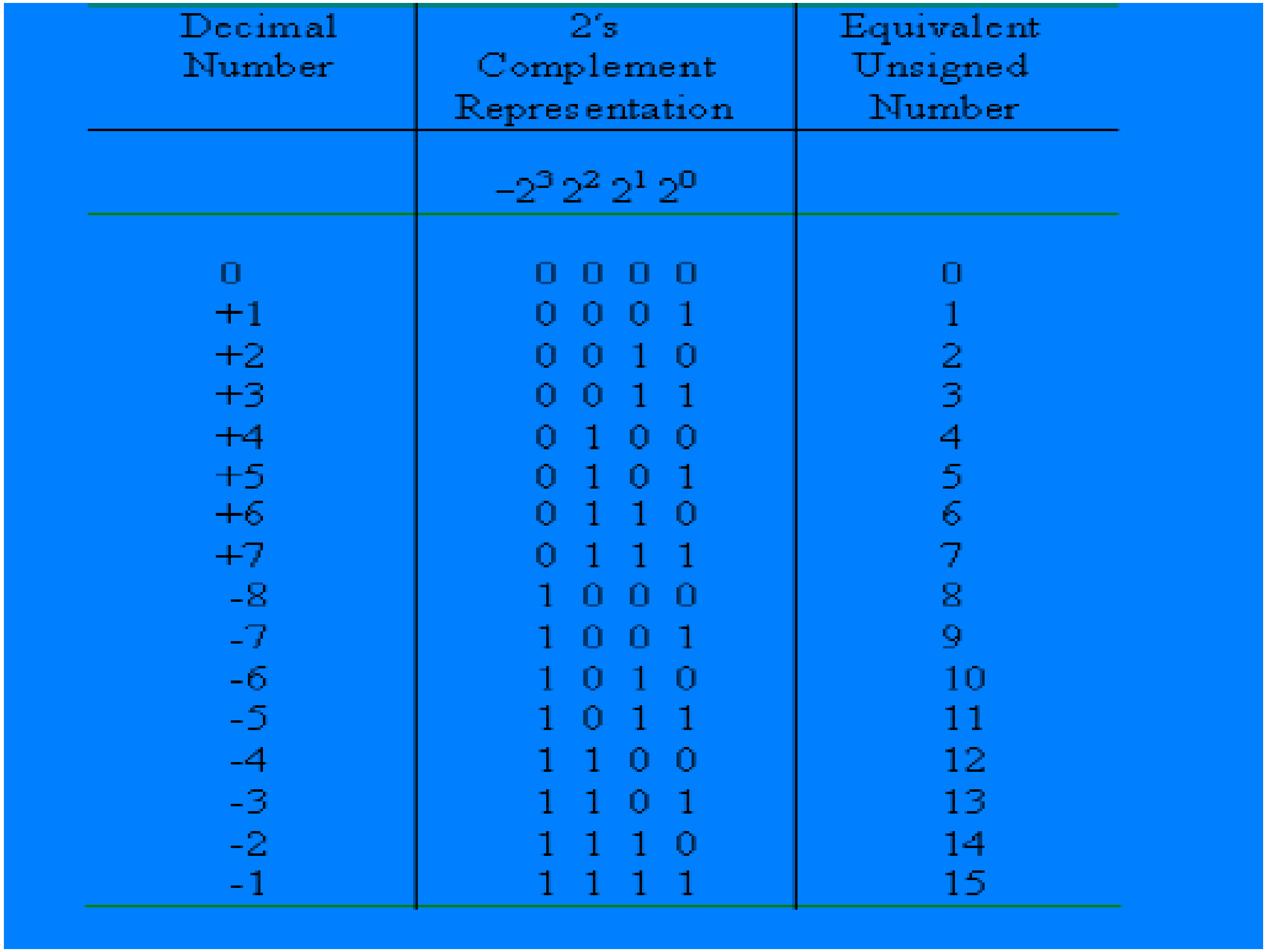
2^3 2^2 2^1 2^0

-8 4 2 1 (weights)

-8+2+1 = -5 (Decimal Equivalent)

2’s Complement

2’s Representation of 4-bit Numbers & Their Unsigned Equivalent Numbers



Decimal Number	2’s Complement Representation	Equivalent Unsigned Number
	$-2^3\ 2^2\ 2^1\ 2^0$	
0	0 0 0 0	0
+1	0 0 0 1	1
+2	0 0 1 0	2
+3	0 0 1 1	3
+4	0 1 0 0	4
+5	0 1 0 1	5
+6	0 1 1 0	6
+7	0 1 1 1	7
-8	1 0 0 0	8
-7	1 0 0 1	9
-6	1 0 1 0	10
-5	1 0 1 1	11
-4	1 1 0 0	12
-3	1 1 0 1	13
-2	1 1 1 0	14
-1	1 1 1 1	15

2's Complements

Representation Characteristics

- The representation of –ve No. facilitates the H/W implementation of many basic arithmetic operations
- This representation is widely used for executing arithmetic operation in specified algorithms and general purpose architecture

- In the example, the maximum +ve No. is 7
- The min –ve No. is -8

Several examples

Binary	Decimal
00000000	0
00000001	1
01000000	64
01111111	127
10000000	-128
10000001	-127
11000000	-64
11111111	-1

2’s Complement Computation

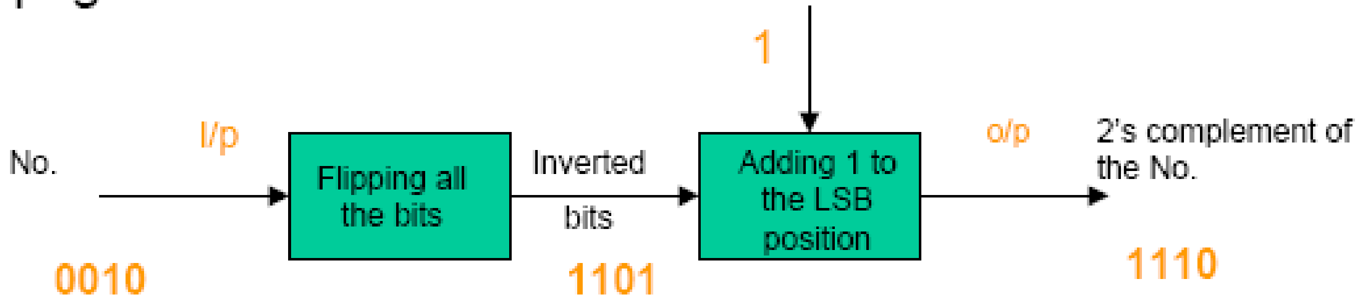
Method1

- we move from LSB to MSB leaving the first Non-Zero bit as it is and flipping all the rest of the bits.
- This is certainly not the best way of taking 2’c complement in H/W
- 2’s complement of 0010 is 1110

2's Complement Computation

Method 2

- The best way of taking 2's complement in hardware is to first flip all the bits and then add 1.
- But in this case there is another limitation, we need a carry propagate adder.



Sign Extension

- Sometimes, you need to convert an 8-bit 2's complement number to a 16-bit number.
  - What is the 16-bit 2's complement number representing the same value as the 8-bit numbers 01001011_2 and 10010111_2?
    - The answer is to sign extend the 8-bit numbers:
      - □ 0000000001001000_2 and
      - □ 1111111110010111_2.

Overflow

overflow occurs when:

- POS+POS=NEG or
- NEG+NEG=POS

- XOR (C_{out} and C_{out-1})

Addition/subtraction and Overflow

- +3 0011 +5 0101 +5 0101
- +4 0100 +6 0110 -6 1010
- --------- --------- ---
- +7 0111 -5 **1011** -1 1111

- -5 1011 -3 1101 -5 1011
- +6 0110 -4 1100 -6 1010

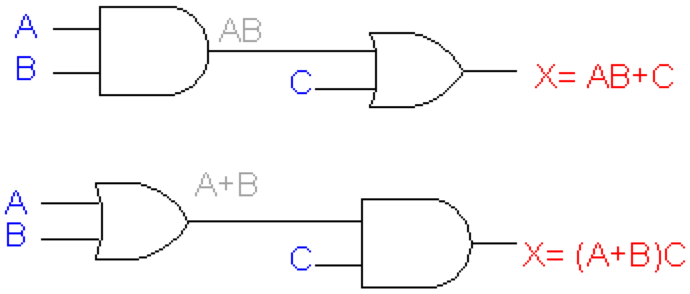
--------- --------- --------

- +1 0001 -7 1001 +5 **0101**

Boolean Algebra and Logic Gates

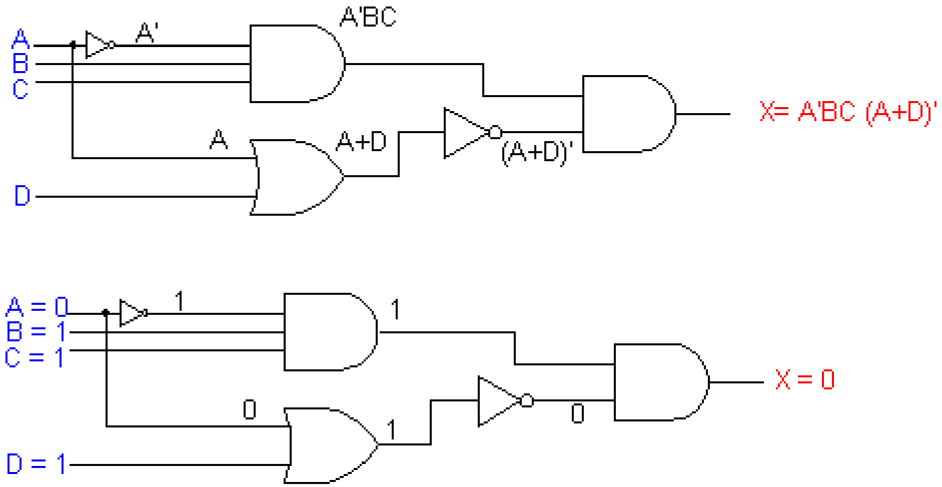
Describing Logic Circuits Algebraically

- Any logic circuit, no matter how complex, may be completely described using the Boolean operations, because the OR gate, AND gate, and NOT circuit are the basic building blocks of digital systems.



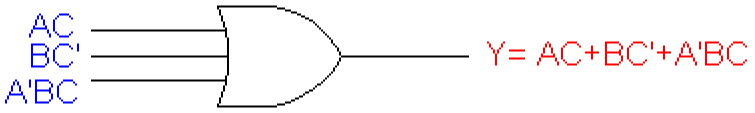
Determining Output Level from a Diagram

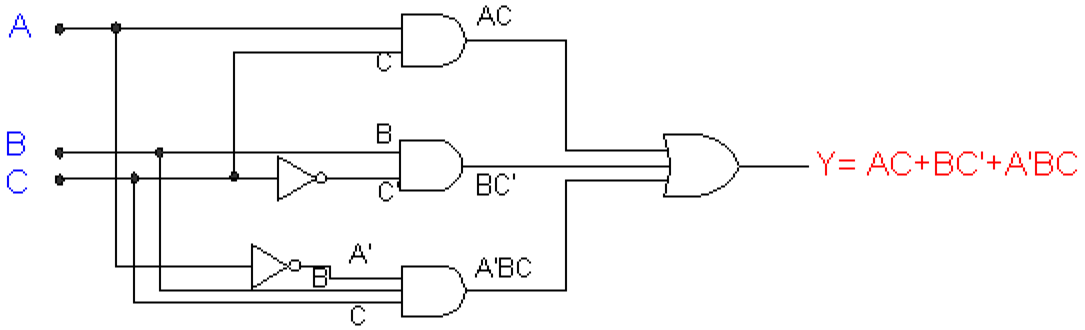
- The output logic level for given input levels can also be determined directly from the circuit diagram without using the Boolean expression.



Implementing Circuits From Boolean Expression

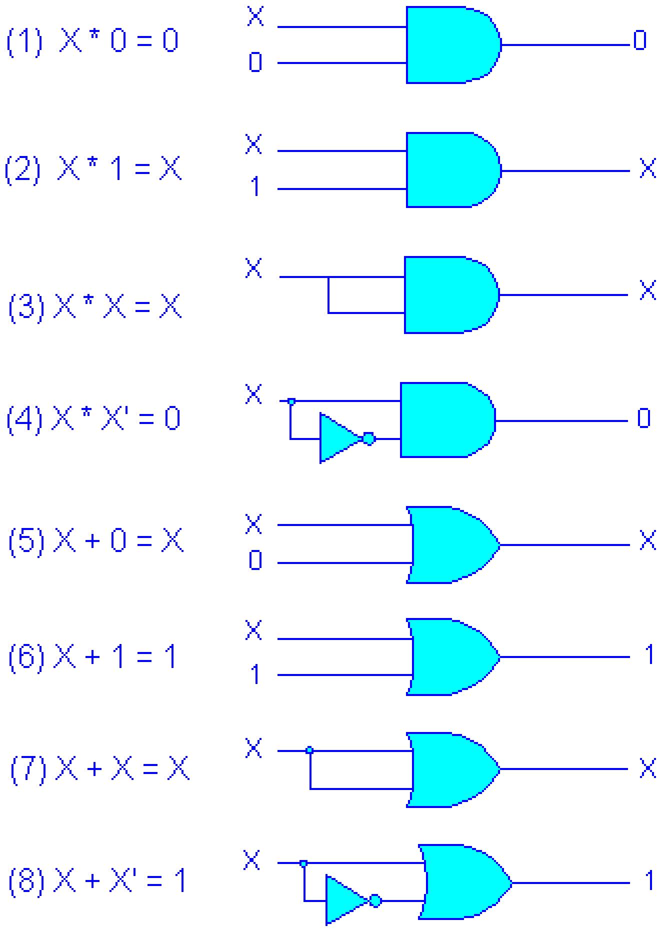
- If the operation of a circuit is defined by a Boolean expression, a logic-circuit diagram can be implemented directly from that expression.
- Suppose that we wanted to construct a circuit whose output is y = AC+BC' + A'BC. This Boolean expression contains three terms (AC, BC', A'BC), which are ORed together. This tells us that a three-input OR gate is required with inputs that are equal to AC, BC', and A'BC, respectively.
- Each OR-gate input is an AND product term, which means that an AND gate with appropriate inputs can be used to generate each of these terms. Note the use of INVERTERs to produce the A' and C' terms required in the expression.





Boolean Theorems

- various Boolean theorems (rules) can help us to simplify logic expressions and logic circuits.



(1) $X * 0 = 0$

(2) $X * 1 = X$

(3) $X * X = X$

(4) $X * X' = 0$

(5) $X + 0 = X$

(6) $X + 1 = 1$

(7) $X + X = X$

(8) $X + X' = 1$

Multivariable Theorems

(9) $x + y = y + x$ *(commutative law)*

(10) $x * y = y * x$ *(commutative law)*

(11) $x+ (y+z) = (x+y) +z = x+y+z$ *(associative law)*

(12) $x (yz) = (xy) z = xyz$ *(associative law)*

(13a) $x (y+z) = xy + xz$ (distributive law)

(13b) $x+yz = (x+y) (x+z)$ (distributive law-- DUAL)

(13c) $(w+x)(y+z) = wy + xy + wz + xz$

(14) $x + xy = x$ [see proof below]

(15) $x + x'y = x + y$

Proof of (14)

$x + xy = x (1+y)$

$= x * 1$ *[using theorem (6)]*

$= x$ *[using theorem (2)]*

DeMorgan's Theorem

- DeMorgan's theorems are **extremely** useful in simplifying expressions in which a product or sum of variables is inverted. The two theorems are:

(16) (x+y)' = x' * y'
(17) (x*y)' = x' + y`

- Theorem (16) says that when the OR sum of two variables is inverted, this is the same as inverting each variable individually and then ANDing these inverted variables.
- Theorem (17) says that when the AND product of two variables is inverted, this is the same as inverting each variable individually and then ORing them.

DeMorgan's Theorem

- **Example:**

X = [(A'+C) * (B+D')]’

= (A'+C)' + (B+D')' *[by theorem (17)]*

= (A''*C') + (B’*D'') *[by theorem (16)]*

= AC' + B'D

Complement of a Function

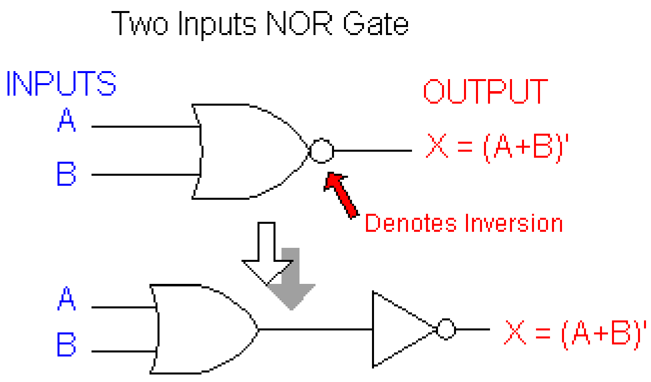
- Find Complement using DeMorgan’s Theorem:

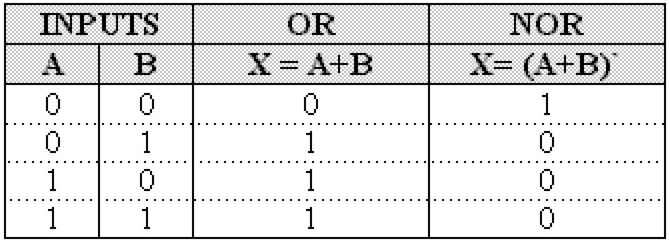
- $F1 = x'yz' + x'y'z$
- $F1' = (x+y'+z)(x+y+z')$

- $F2 = x(y'z' + yz)$
- $F2' = x'+ (y+z)(y'+z')$

NOR Operation

- NOR and NAND gates are used extensively in digital circuitry. These gates combine the basic operations AND, OR and NOT, which make it relatively easy to describe then using Boolean Algebra.
- NOR is the same as the OR gate symbol *except* that it has a small circle on the output. This small circle represents the inversion operation. Therefore the output expression of the two input NOR gate is: X = (A + B)'





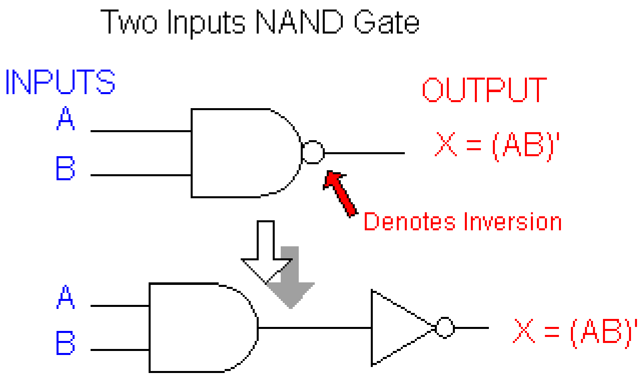
INPUTS		OR	NOR
A	B	X = A+B	X= (A+B)'
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

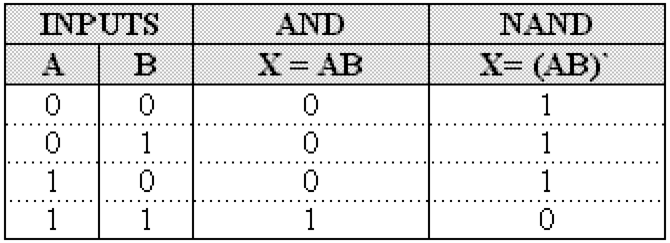
NAND Operation

- NAND is the same as the AND gate symbol *except* that it has a small circle on the output. This small circle represents the inversion operation.

  Therefore the output expression of the two input NAND gate is: $X = (AB)'$

Two Inputs NAND Gate

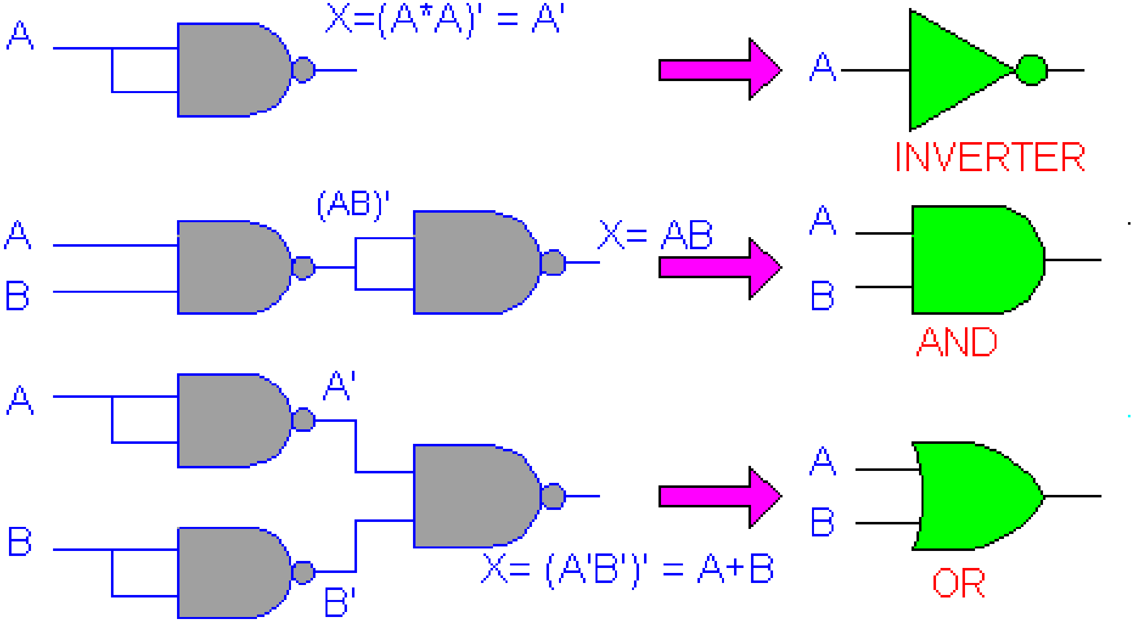




INPUTS		AND	NAND
A	B	X = AB	X= (AB)'
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

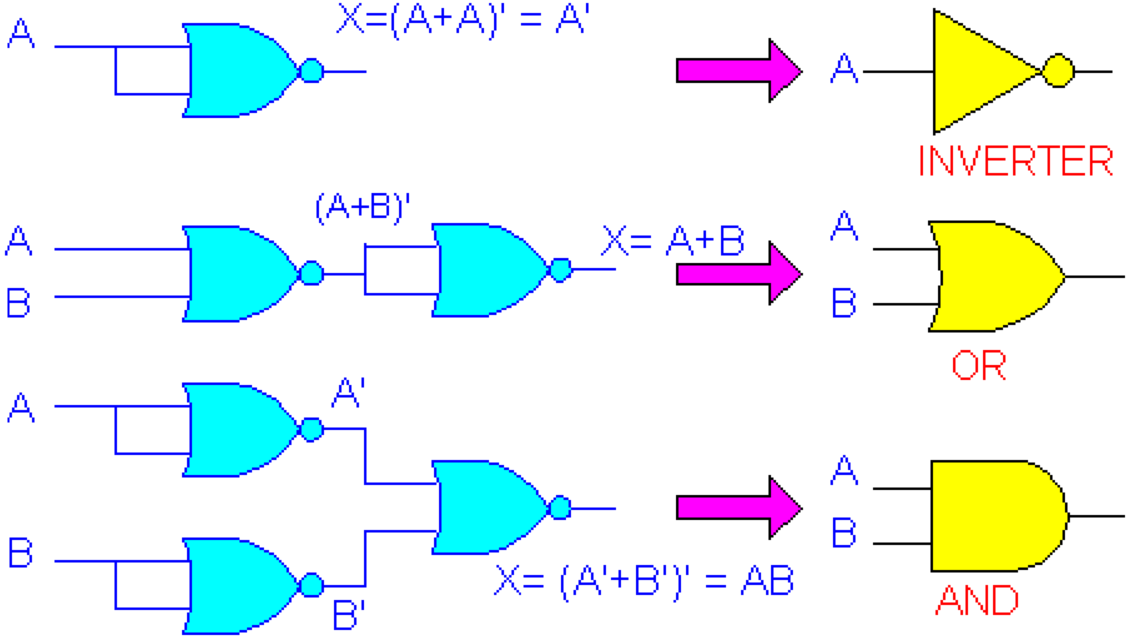
Universality of NAND

- It is possible to implement *any* logic expression using only NAND gates and no other type of gate. This is because NAND gates, in the proper combination, can be used to perform each of the Boolean operations OR, AND, and INVERT.



Universality of NOR

- In a similar manner, it can be shown that NOR gates can be arranged to implement any of the Boolean operations



Minterms

- The Order of the Variables in Each Term Is As Specified in the Function, e.g.,

$$f(\underset{\text{MSB}}{A},\underset{\text{LSB}}{B}) = \underset{\text{minterm }(\ 01\ )_2\ =\ (\ 1\ )_{10}}{\bar{A}B} + \underset{\text{minterm }(\ 11\ )_2\ =\ (\ 3\ )_{10}}{AB}$$

Minterm Example

Minterm	A	B	f(A,B)
0	0	0	0
1	0	1	1
2	1	0	0
3	1	1	1

$$f(A,B)=\sum m(1,3)$$

TT from SOP Form

$$g(x, y, z) = x + yz$$

Minterms	x	y	z	g(x,y,z)
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

yz true

x true

Adders

How to add binary numbers

- Consider adding two 1-bit binary numbers *x* and *y*
  - 0+0 = 0
  - 0+1 = 1
  - 1+0 = 1
  - 1+1 = 10

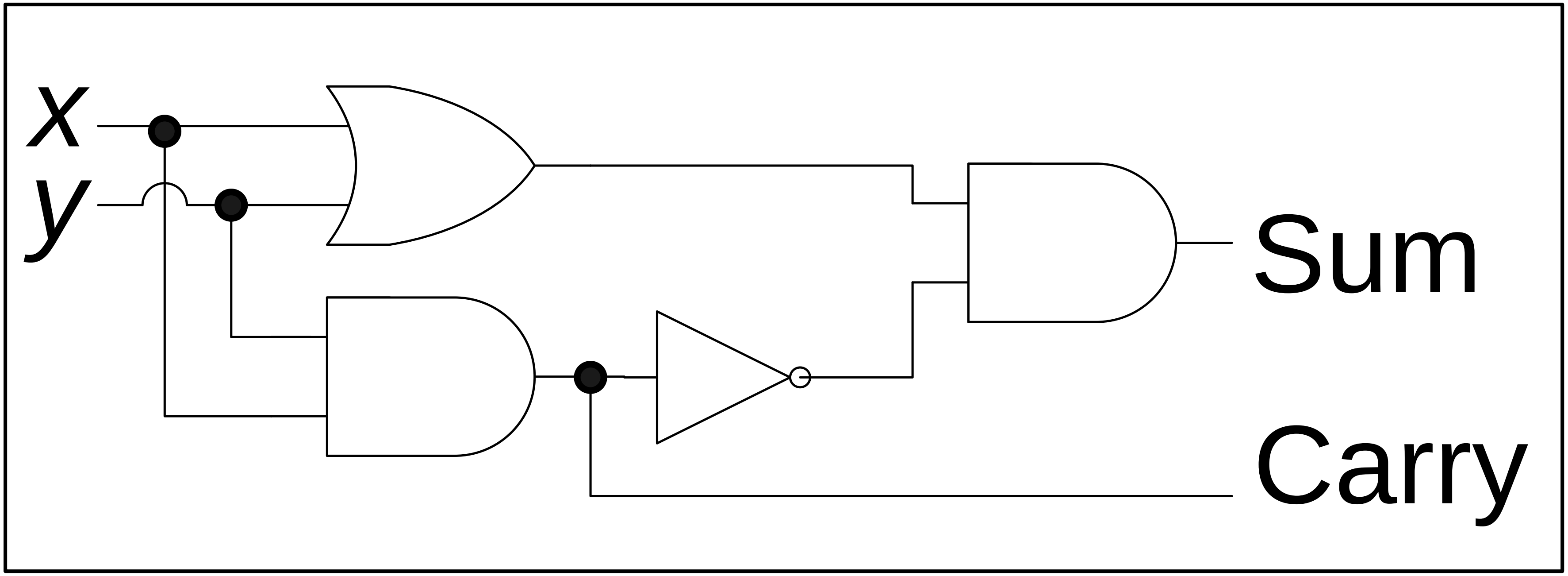
x	*y*	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Carry is *x* AND *y*
- Sum is *x* XOR *y*
- The circuit to compute this is called a half-adder

The half-adder

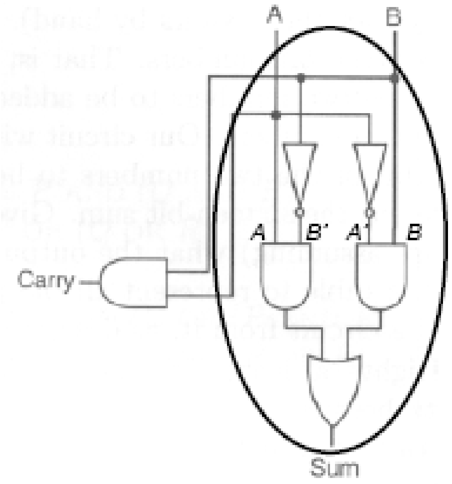
- Sum = *x* XOR *y*
- Carry = *x* AND *y*

Alternate Circuit

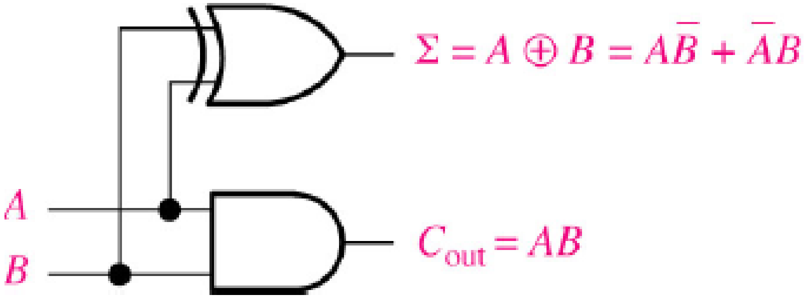


Half Adder Implementations

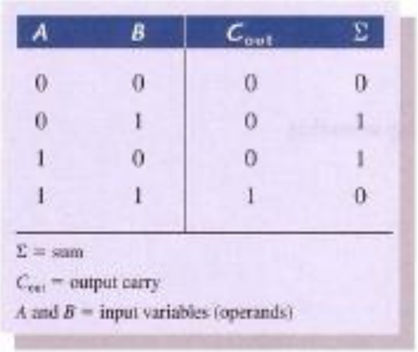
- The half adder is most simply represented with an exclusive OR, but we rarely use such devices and so it is re-written in terms of other gates. Half Adders have various implementations



- HW: Look at the various implementations of Half Adder in your book and confirm (by simplifying) that they all can be used to represent the XOR function `F = A'B + AB'`



Logic Symbol for A Half Adder



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

Σ = sum

C_{out} = output carry

A and B = input variables (operands)

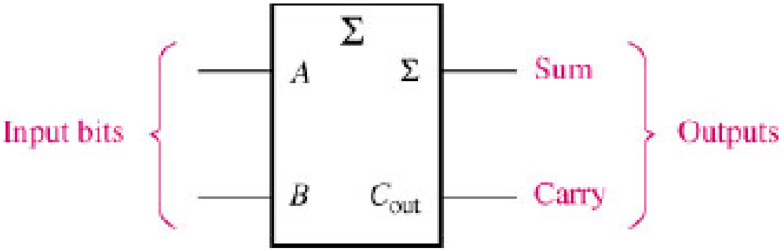
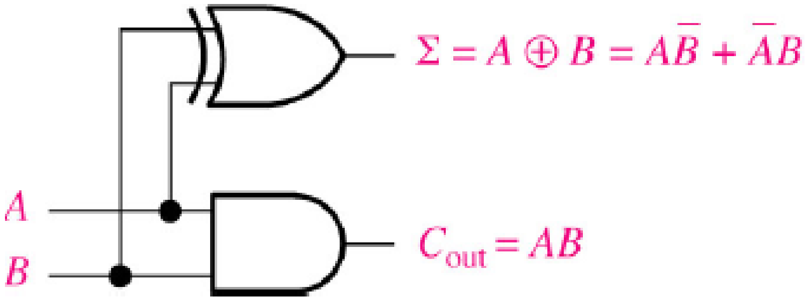


Figure 6–1 Logic symbol for a half-adder

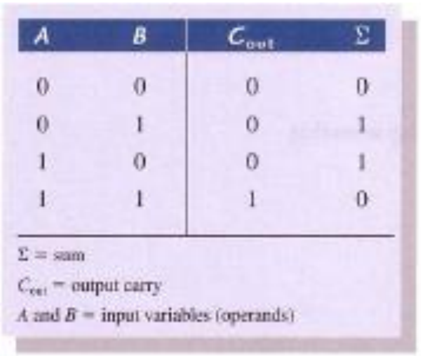


$\Sigma = A \oplus B = A\overline{B} + \overline{A}B$

$C_{out} = AB$

Using half adders

- We can then use a half-adder to compute the sum of two Boolean numbers

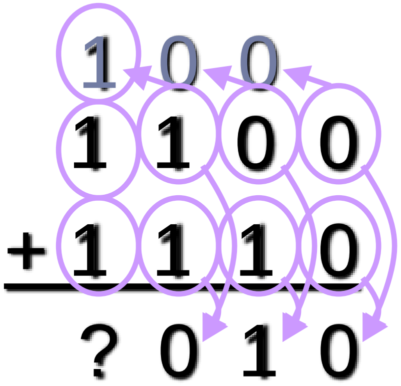


A	B	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Σ = sum

C_{out} = output carry

A and B = input variables (operands)



```
   1 0 0
   1 1 0 0
 + 1 1 1 0
 ---------
   ? 0 1 0
```

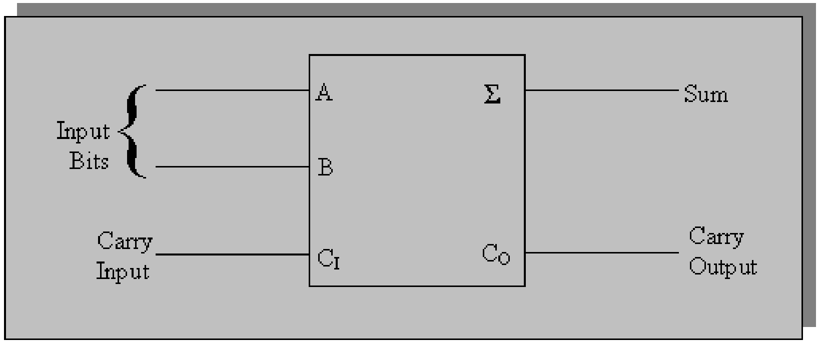
How to fix this

- We need to create an adder that can take a carry bit as an additional input
  - Inputs: *x*, *y*, carry in
  - Outputs: sum, carry out
- This is called a full adder

x	*y*	*c*	carry	sum
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

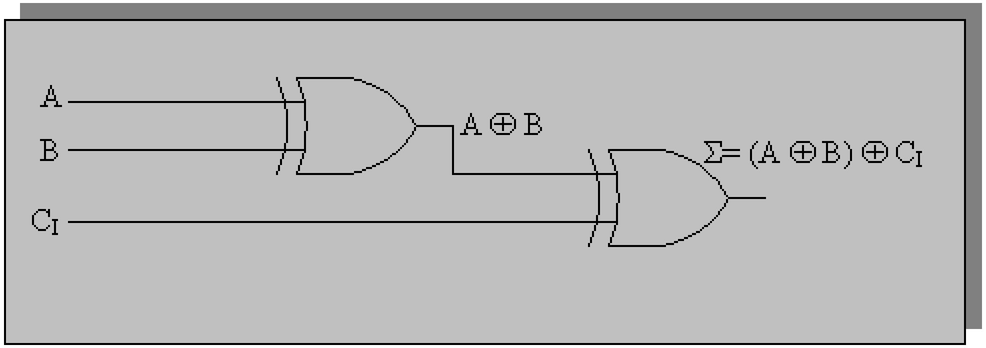
Full Adder

- The second basic category of adder is the full-adder. This combinational circuit performs the arithmetic addition of three input bits. The noticeable difference between the full- and the half-adder is the ability of the former to handle input carries (Cin). The logical symbol for the full-adder is shown in Figure



Full Adder 'Sum' Equation

- By the very nature of the full adder we know that the two input bits must be added to the carry input bit. Recall that for the half-adder the sum of A and B is the XOR of those two variables



$$\Sigma = A \oplus B$$

- Similarly, for the three variables A, B and Cin the sum becomes

$$\Sigma = (A \oplus B) \oplus C_I$$

We can also use the Full Adder Truth Table to come up with the same equation (Recall the XOR Truth Table)

HW: Drive the same equation using Algebraic manipulation of the Minterms

Full Adder Carry Equation

Following are the minterms of Cout:

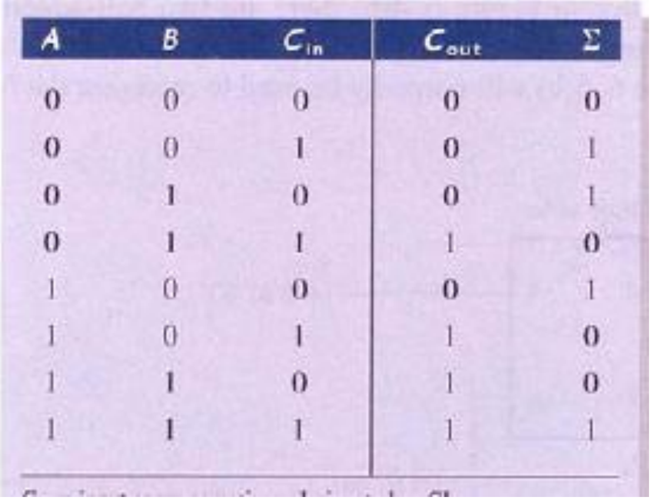
$\text{Cout} = A'BC + AB'C + ABC' + ABC$

$= C(A'B + AB') + AB\,(C + C')$

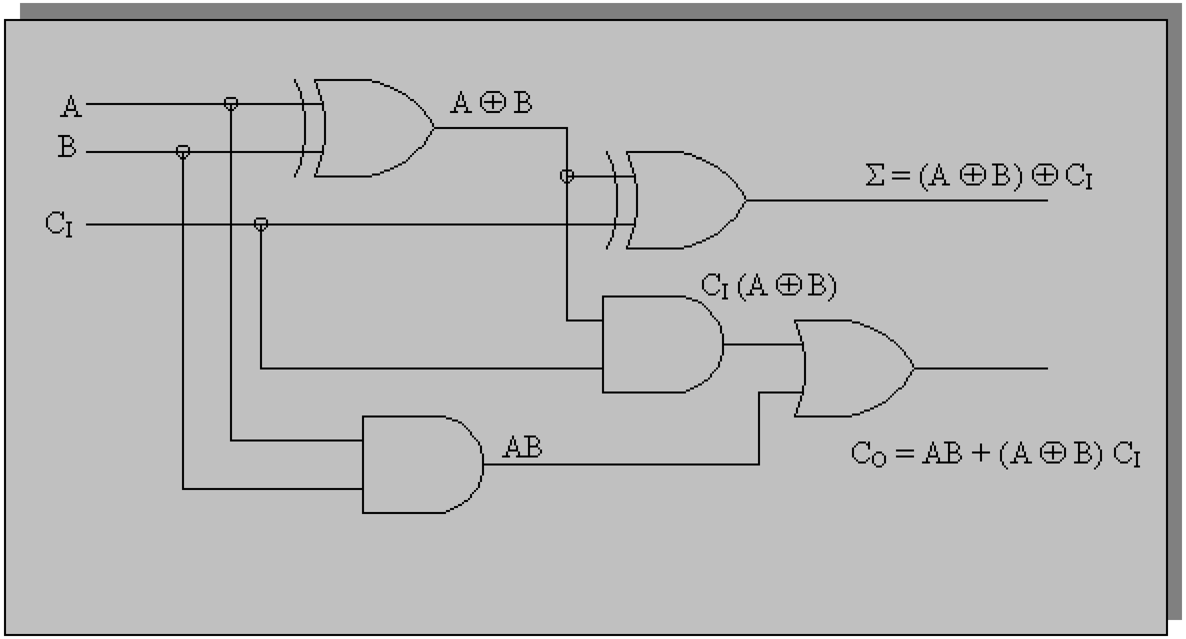
$= C(A'B + AB') + AB$

$= C(A \text{ XOR } B) + AB$

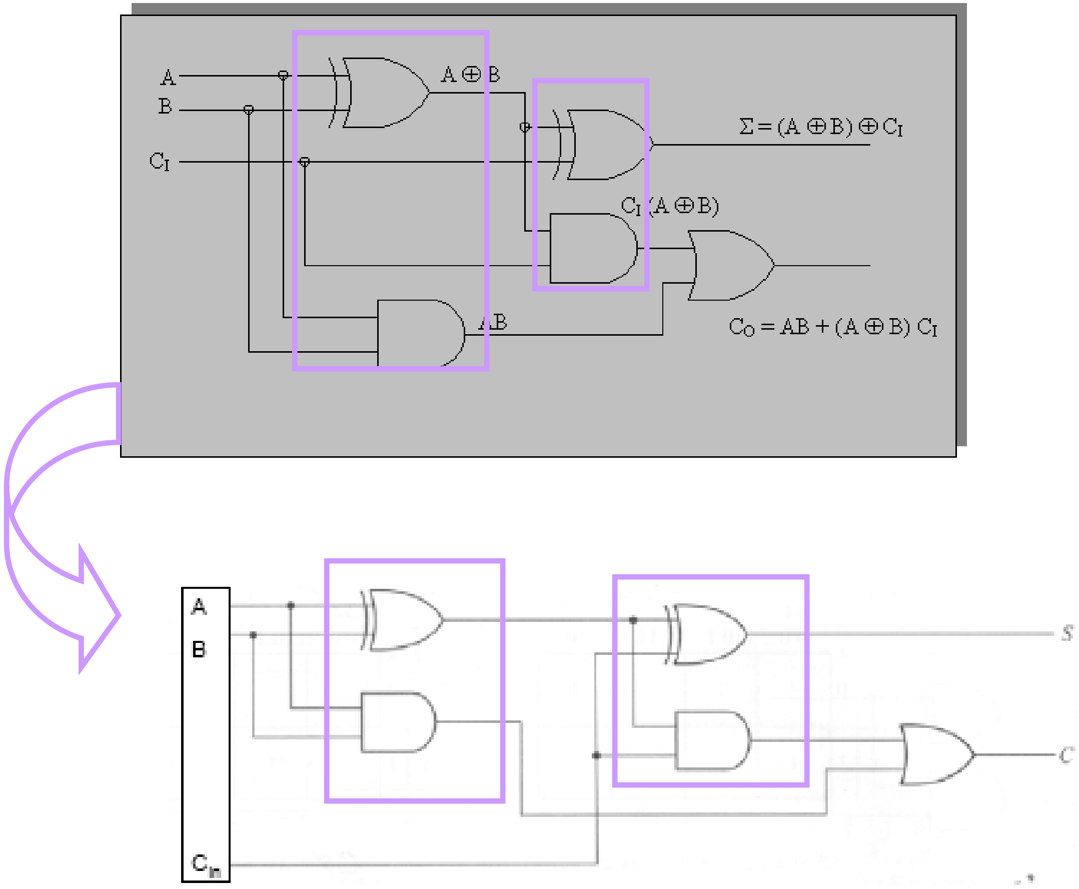
NOTE: C is used to represent Cin



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


A
B
C_I
$A \oplus B$
$\Sigma = (A \oplus B) \oplus C_I$
$C_I\,(A \oplus B)$
AB
$C_O = AB + (A \oplus B)\,C_I$

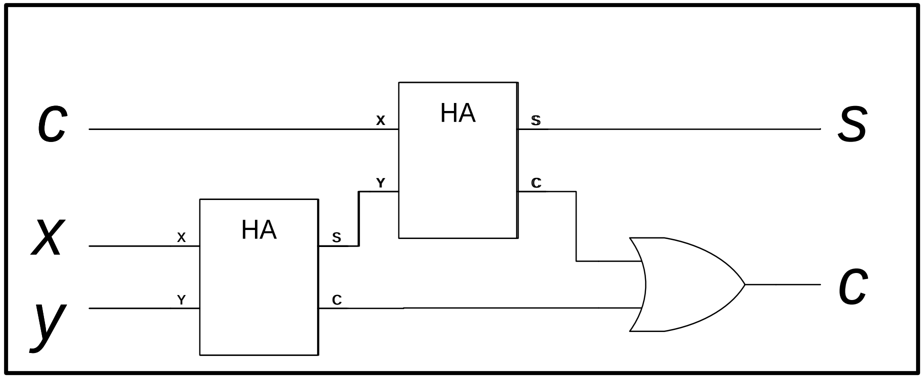
The Full Adder Circuit is made up of Halves

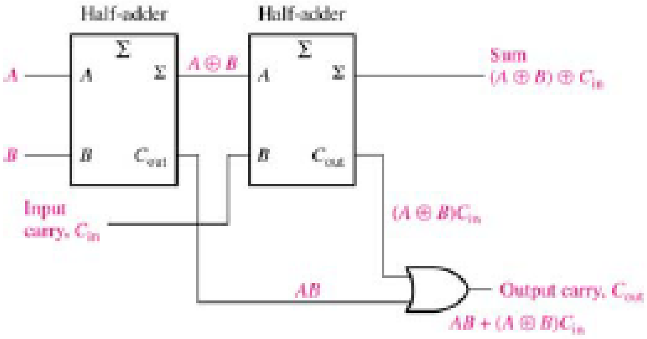


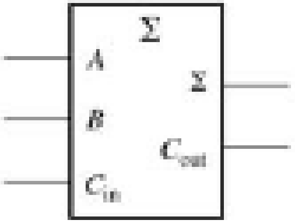
- Can you recognize the 2 Half Adders in this circuit?

Full adder using 2 half adders

- The “HA” boxes are half-adders





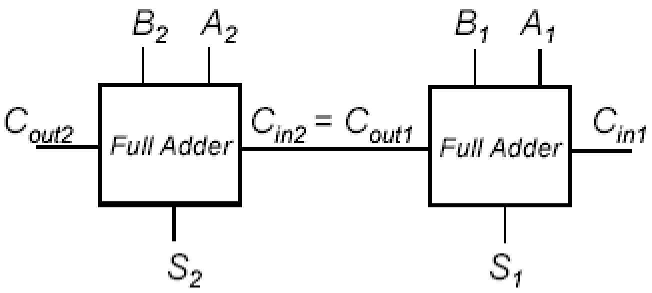


(b) Full-adder logic symbol

Full Adder Logic
Symbol

2 bit Parallel Adder

$$\begin{array}{rcr} A_2A_1 & \Leftrightarrow & 1\ 1 \\ +\ \underline{B_2B_1} & \Leftrightarrow & +\ \underline{1\ 0} \end{array}$$



OR

Note that Cin of the first Full Adder will always be ZERO

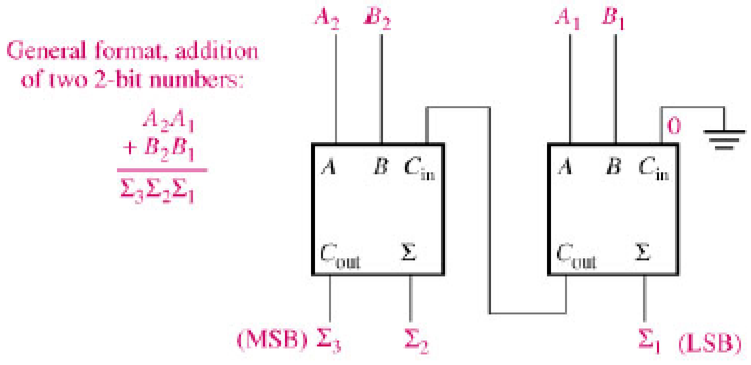
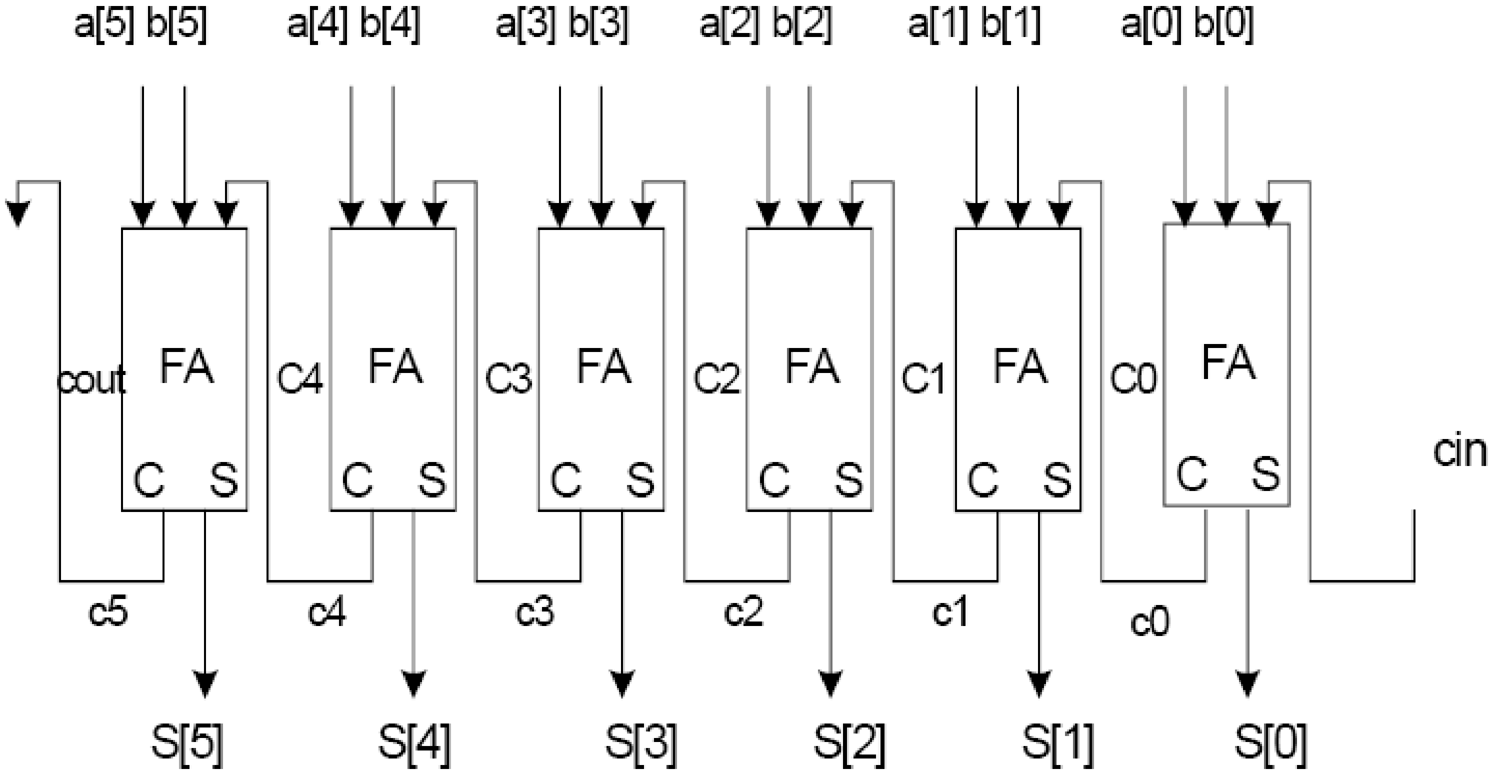


Figure 6–7 Block diagram of a basic 2-bit parallel adder using two full-adders.

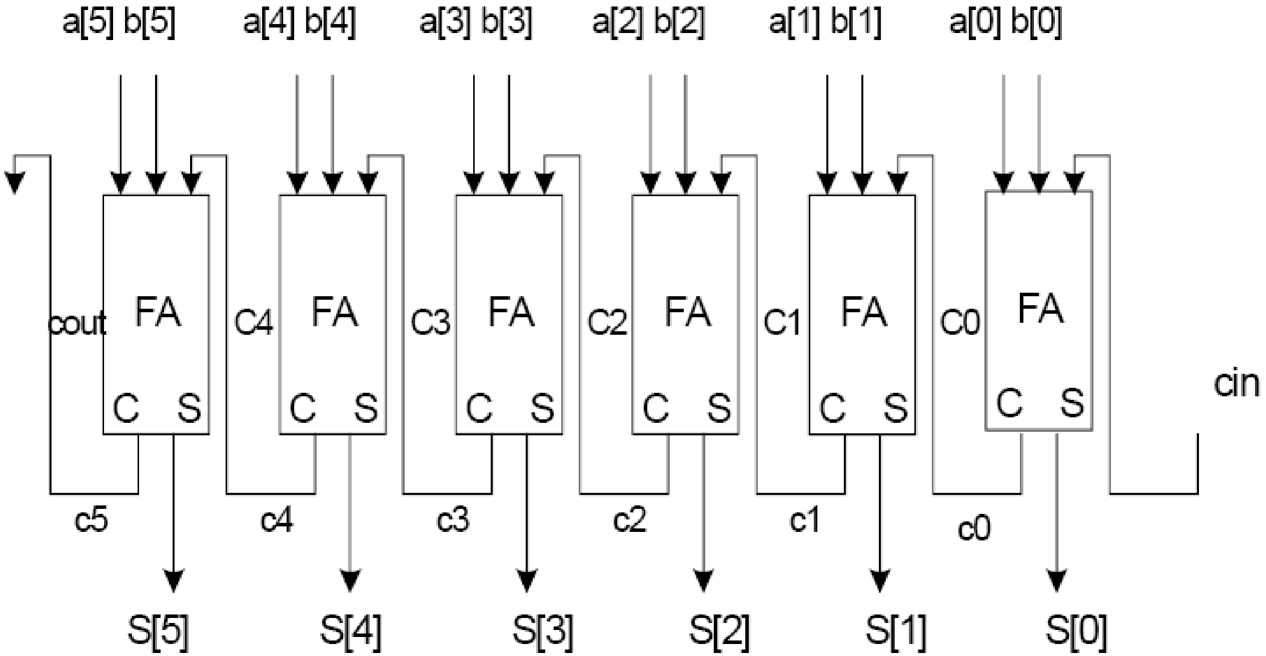
6 bit Parallel; Ripple-Carry Adder

Four full adders connected in a ripple-carry chain form a four-bit ripple-carry adder.



Ripple Carry Adder Example

```
0 0 1 1 0 0                    a[5]  a[4] a[3] a[2] a[1] a[0]
0 0 1 1 1 0                    b[5]  b[4] b[3] b[2] b[1] b[0]
----------- +                  ------------------------------  +
0 1 1 0 1 0        Cout  <---  s[5]  s[4] s[3] s[2] s[1] s[0]
-----------
```



Drawback !!!

- **There is a delay of full adder at each stage Carry ripples through the entire adder**

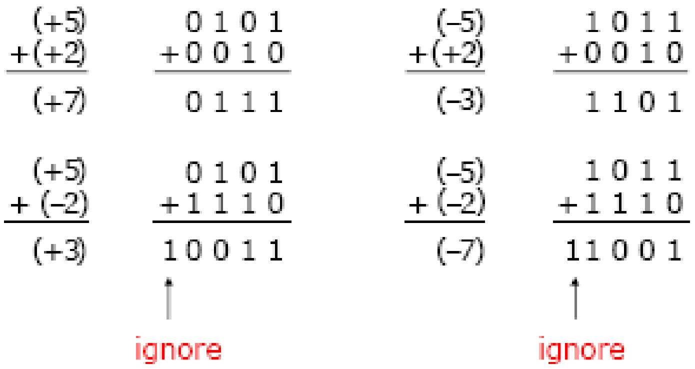
Subtractors

2s Complement Subtractors

- **If we design an adder, then by using 2's complement arithmetic we can use the same adder for addition and subtraction.**
- **So this results in reduced and simplified H/W. For example, we can achieve both addition and subtraction by an adder**
- **a+b**
- **a-b =a+(-b)**

Subtraction

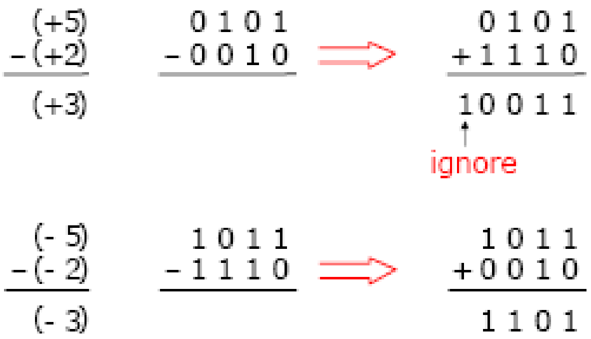
- For addition, the result is always correct
- Any carry-out from the sign-bit position is simply ignored



```
  (+5)      0 1 0 1       (–5)      1 0 1 1
+(+2)     +0 0 1 0      +(+2)     +0 0 1 0
  (+7)      0 1 1 1       (–3)      1 1 0 1

  (+5)      0 1 0 1       (–5)      1 0 1 1
+ (–2)    +1 1 1 0      + (–2)    +1 1 1 0
  (+3)    1 0 0 1 1       (–7)    1 1 0 0 1
          ↑                       ↑
        ignore                  ignore
```

- The easiest way of performing subtraction is to negate the subtrahend and add it to the minuend
  - Find the 2's complement of the subtrahend and then perform addition



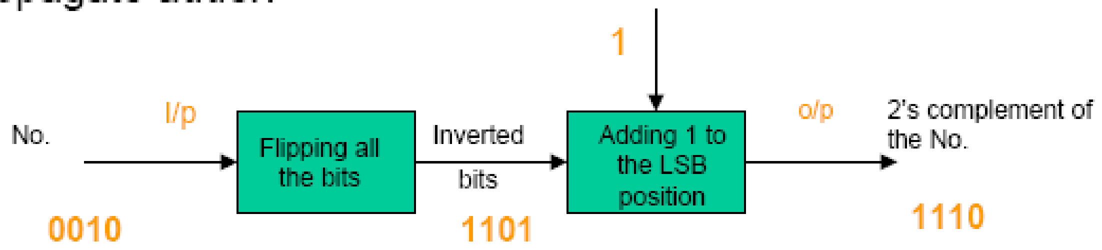
```
  (+5)      0 1 0 1            0 1 0 1
– (+2)    – 0 0 1 0    ⟹     + 1 1 1 0
  (+3)                       1 0 0 1 1
                             ↑
                           ignore

 (- 5)      1 0 1 1            1 0 1 1
–(- 2)    – 1 1 1 0    ⟹     + 0 0 1 0
 (- 3)                         1 1 0 1
```

Review

Method 2

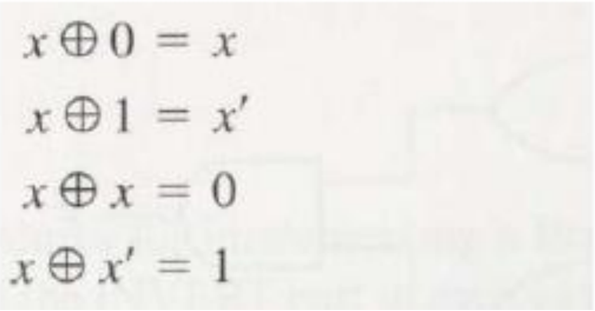
- The best way of taking 2's complement in hardware is to first flip all the bits and then add 1.
- But in this case there is another limitation, we need a carry propagate adder.



Remember 2s Complement!!

XOR: x ⊕ y = xy' + x'y

Remember XOR!!



$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

$$x \oplus x = 0$$

$$x \oplus x' = 1$$

Design of Adder-Subtractor

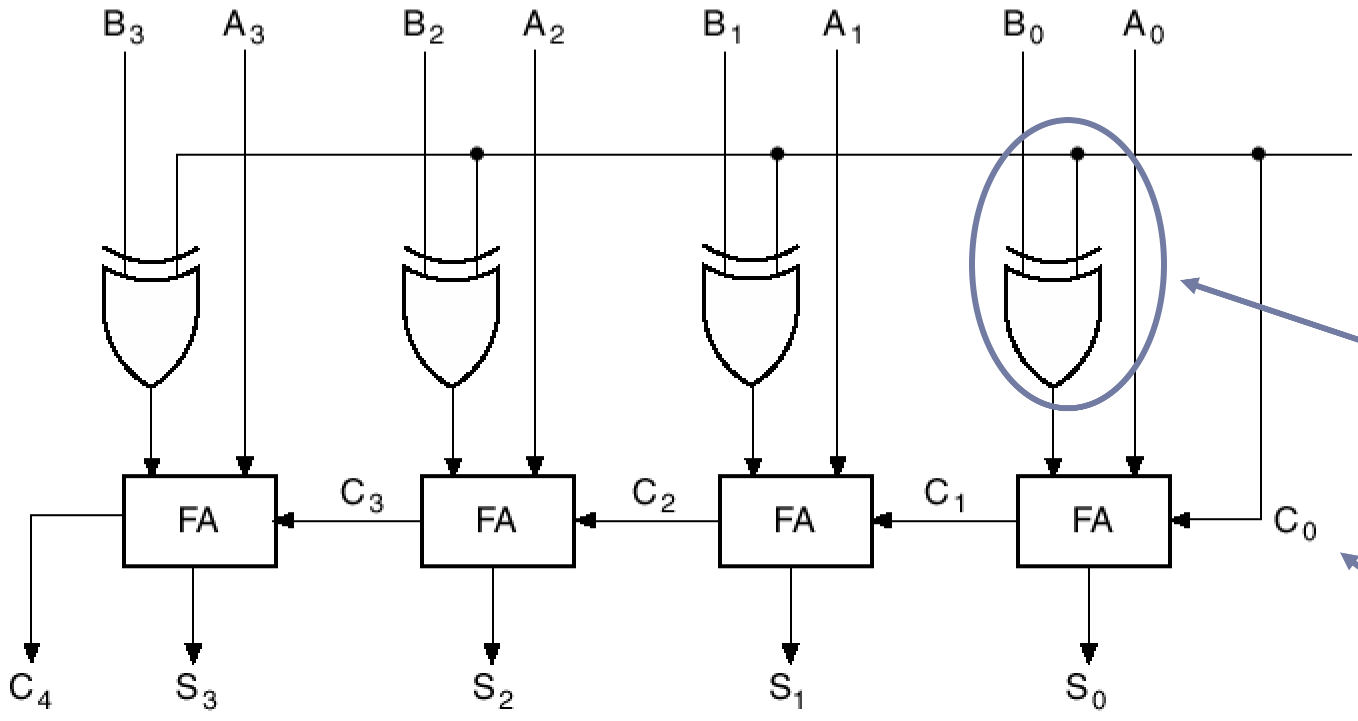


Fig. 3-31 Adder-Subtractor Circuit

S low for add, high for subtract

Inverts each bit of B if S is 1

Adds 1 to make 2's complement