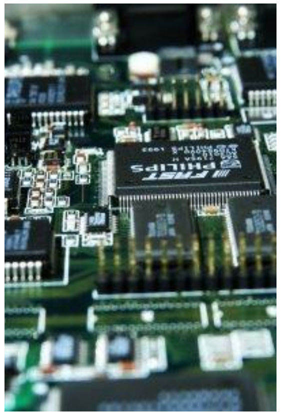
CSE 401

Computer Engineering (2)

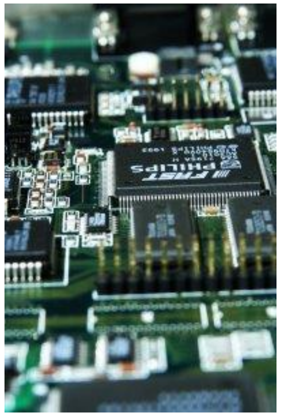
هندسة الحاسبات (2)



4th year, Comm. Engineering

Winter 2016

Lecture #8



Dr. Hazem Ibrahim Shehata

Dept. of Computer & Systems Engineering

Credits to Dr. Ahmed Abdul-Monem Ahmed for the slides

Adminstrivia

- Assignment #2:
  - Released: Yesterday. Due: Wednesday, April 13, 2016.
- Midterm:
  - Date: Thursday, April 21, 2014
  - Time: 10:30am – 12:00pm
  - Location: classroom #27321 (قاعة 4د)
  - Coverage: lectures #1 ➔ #7
- Lecture include material from another textbook:
  - "**Computer Organization and Embedded Systems**", C. Hamacher, Z. Vranesic, S. Zaky, N. Manjikian (**6th Ed.**)

Website: http://hshehata.github.io/courses/zu/cse401/
Office hours: Monday 11:30am – 12:30pm

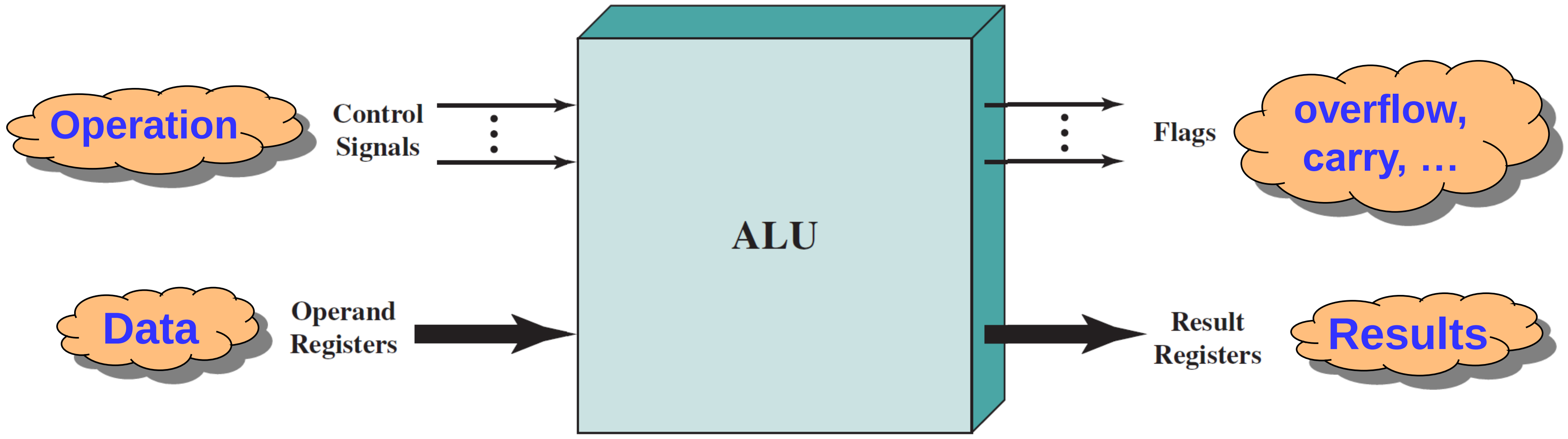
Chapter 10. Computer Arithmetic

Outline

- Integer Representation
    - —Sign-Magnitude, Two’s Complement, Biased
- Integer Arithmetic
    - —Negation, Addition, Subtraction
    - —Multiplication, Division
- Floating-Point Representation
    - —IEEE 754
- Floating-Point Arithmetic
    - —Addition, Subtraction
    - —Multiplication, Division
    - —Rounding

Arithmetic & Logic Unit (ALU)

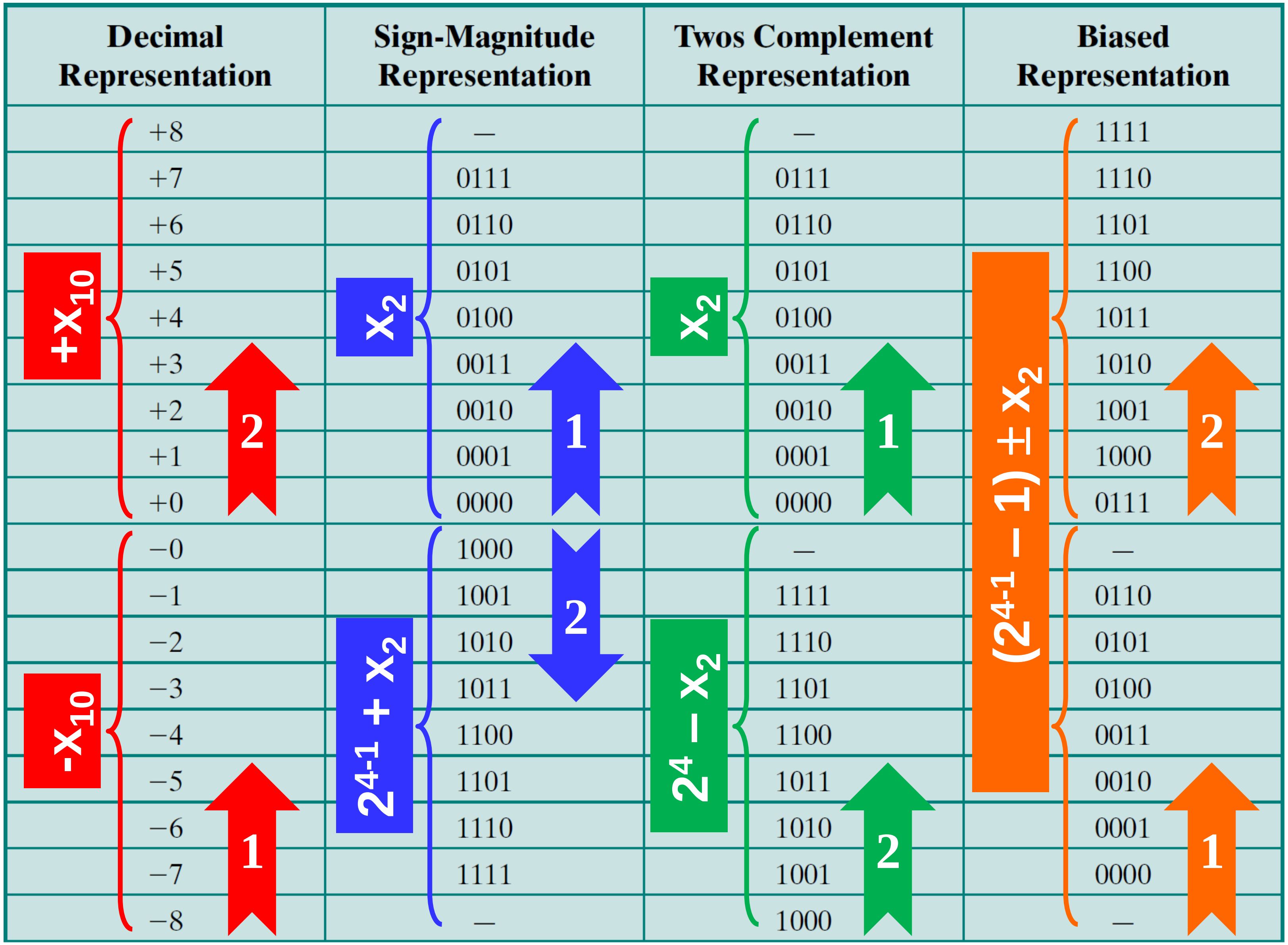
- The unit that does all the calculations!
- Everything else in computer is there to bring data to ALU and take results back out.
- It can handle both integers & real (floating point) numbers.
  - Note: In the past, Floating-Point Unit (FPU) used to be separate from ALU (off-chip) ➔ math co-processor!!



Integer Representation

- General-case number: –548.923
- Only have 0 & 1 to represent everything!
  - No minus sign!!
  - No radix point (period)!!!
- Unsigned (i.e., always positive) integers:
  - Straightforward ➔ represent integer value in binary!
  - An n-bit word can represent the numbers: $0 \rightarrow 2^n-1$
  - Ex.: $(41)_{10}$ represented using 8-bits as “00101001”.
- Signed integers:
  - Not straightforward!
    - Sign-magnitude representation
    - Biased representation
    - Two’s complement representation

Representations of 4-Bit Signed Integers



Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
−0	1000	—	—
−1	1001	1111	0110
−2	1010	1110	0101
−3	1011	1101	0100
−4	1100	1100	0011
−5	1101	1011	0010
−6	1110	1010	0001
−7	1111	1001	0000
−8	—	1000	—

Sign-Magnitude Representation

- Left most bit is sign bit.
  - "0" means positive. "1" means negative.
- Rest of the bits represent the magnitude.
- Example:
  - +18 = 00010010
  - –18 = 10010010
- Range of n-bit Numbers: $-(2^{n-1} - 1) \rightarrow 2^{n-1} - 1$.
- Problems:
  - Need to consider both sign & magnitude in arithmetic.
  - Two representations of zero (+0 and –0)
    - More difficult to test for 0!
    - One wasted bit combination!!

Biased Representation

- A bias is added to the binary value of the number.
  - Bias = $2^{n-1} - 1$ (if numbers are represented by n bits).
- Example:
  - +18 = 00010010 + 01111111 = 10010001
  - –18 = -00010010 + 01111111 = 01101101
- Range of n-bit Numbers: $-(2^{n-1} - 1) \rightarrow 2^{n-1}$.
- Problems:
  - Need to compensate for the bias in arithmetic (by adding/subtracting a value to/from result)!!
    - Example: Suppose numbers are represented using 4 bits.

      $2_{10} + 1_{10}$ = 1001 + 1000 = 10001 ➔ Wrong result!!

      Result is biased twice ➔ subtract one bias from the result

      10001 – 0111 = 1010 = 3_{10}

Two’s Complement Representation

- Like sign-magnitude representation, leftmost bit is used as a sign bit.
- Differs from sign-magnitude representation in how the remaining bits are interpreted.
- Positive number: convert to binary
- Negative number: 2’s complement
- Example: **8-bit** 2’s complement representation

$+3 = \underline{0}0000011$

$+2 = \underline{0}0000010$

$+1 = \underline{0}0000001$

$+0 = \underline{0}0000000$

$-1 = \underline{1}1111111$

$-2 = \underline{1}1111110$

$-3 = \underline{1}1111101$

n-bit Two's Complement Representation

- Suppose we want to represent a set of signed integer numbers using n bits.
- Then, we have 2^n different combinations ➔ we can represent 2^n different numbers.
  1. Represent the number: 0 by the combination: "00…0".
     - We now have 2^n-1 different combinations left.
  2. Represent each positive number: +A by a combination (whose value is): A ➔ positive integers: $1, 2, \ldots, 2^{n-1}-1$ are represented by combinations: $1, 2, .., 2^{n-1}-1$.
  3. Represent each negative number: –A by a combination (whose value is): $2^n - A$ ➔ negative integers: $-2^{n-1}, -2^{n-1}+1, \ldots, -1$ are represented by combinations: $2^{n-1}, 2^{n-1}+1, \ldots, 2^n-1$.
- Range of representable numbers is: -2^{n-1} ➔ $2^{n-1}-1$.

Characteristics of 2's Comp. Rep. & Arithmetic

Consider n-bit 2's complement representation

Range	-2^{n-1} to $2^{n-1} - 1$
Number of Representations of Zero	One
Equivalent to: $2^n - x_2$ **Negation**	**1's complement** Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract *B* from *A*, take the twos complement of *B* and add it to *A*.

Benefits

- One representation of zero.
- Arithmetic works easily (see later).
- Negating is fairly easy
  - 3_{10} = 00000011
  - Boolean (one's) complement gives 11111100
  - Add 1 to LSB 11111101
  - This is equivalent to $\underbrace{2^8 - 3}_{\text{2's complement of 3}} = 253 = 11111101$

Conversion between 2's Comp. & Decimal

–128	64	32	16	8	4	2	1

Value Box

–128	64	32	16	8	4	2	1	
1	0	0	0	0	0	1	1	
–128						+2	+1	= –125

- Result obtained using value box is correct because:
  - ➔ Sign bit is 1
  - ➔ Number = -(2's comp. of 10000011)

    = -(2^8 - 10000011)

    = -125

Conversion Between Lengths

- Positive numbers ➔ pack with leading zeros
  - +18 =   00010010
  - +18 = 00000000 00010010
- Negative numbers ➔ pack with leading ones
  - -18 =   11101110
  - -18 = 11111111 11101110
- i.e. pack with MSB (sign bit) ➔ Sign extension

Addition and Subtraction

- Addition → Normal binary addition.
  - Monitor sign bit for overflow.
- Subtraction → Take two’s complement of subtrahend and add to minuend
  - A – B = A + (–B)
- So we only need addition and complement circuits.

Why Addition of Numbers in 2's Comp. Works?

- Two positive number
  - Normal binary addition if no overflow.
- Two negative numbers: –A and –B
  - Represent –A as $2^n - A$
  - Represent –B as $2^n - B$
  - Do the addition: Result = $(2^n - A) + (2^n - B)$

$$= \underbrace{2^n}_{\text{Extra bit} \rightarrow \text{ignored}} + \underbrace{[2^n - (A+B)]}_{\text{2's comp. of }(A+B) \rightarrow -(A+B)}$$

- One positive and one negative: A and –B
  - Represent A as A
  - Represent –B as $2^n - B$
  - Result = $A + 2^n - B = \underbrace{2^n - (-A + B)}_{\text{2's comp. of }(-A+B) \rightarrow (A-B)}$

Addition of Numbers in 2’s Comp. Rep.

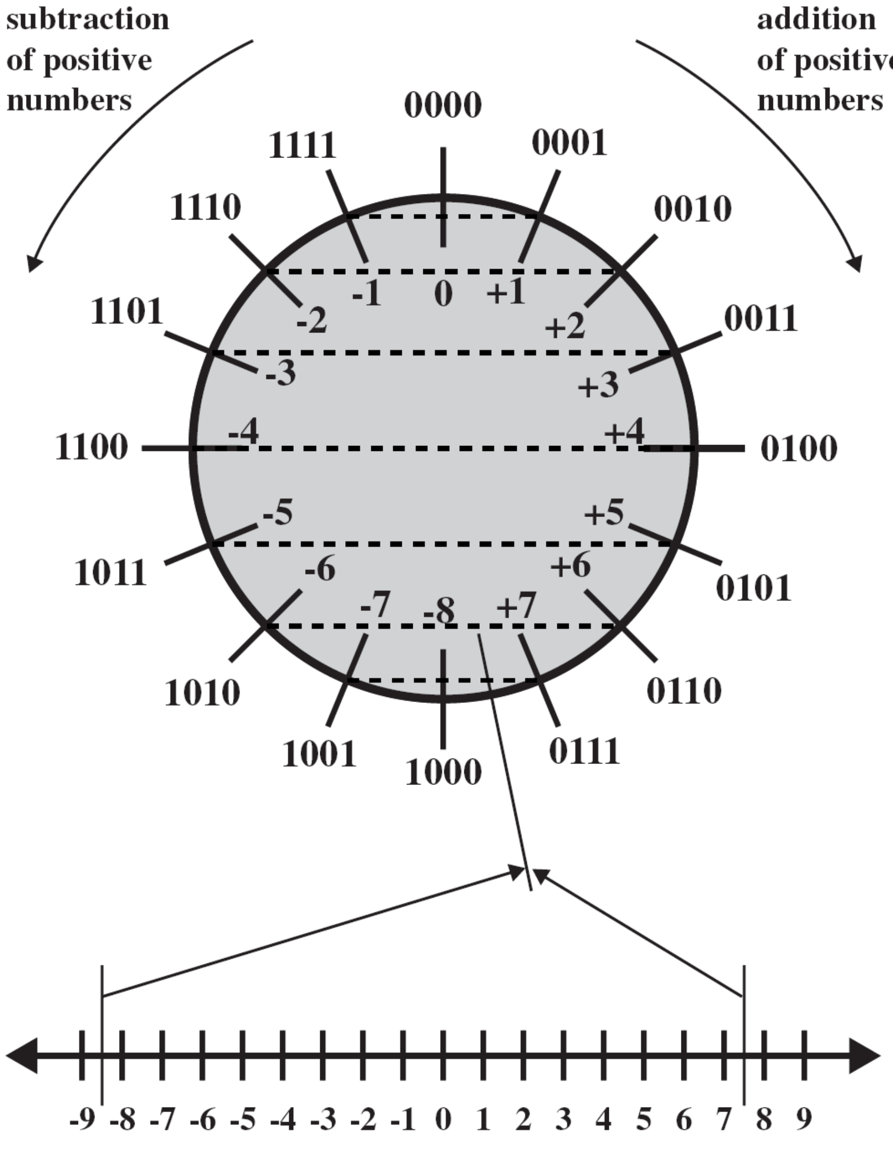
4-bit 2’s comp. representation

$$\begin{array}{r l}
1001 & = -7\\
+\underline{0101} & = 5\\
1110 & = -2
\end{array}
\qquad
\begin{array}{r l}
1100 & = -4\\
+\underline{0100} & = 4\\
10000 & = 0
\end{array}$$

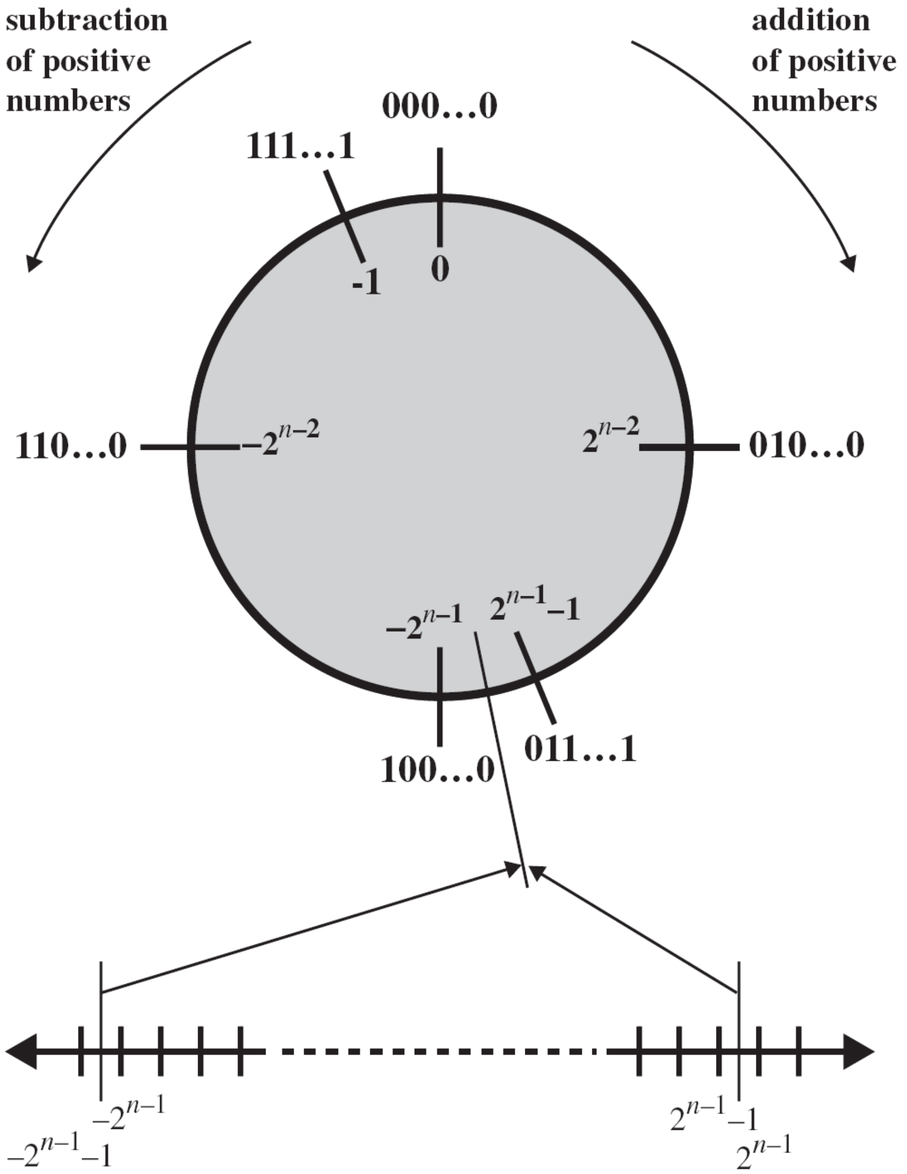
$$\begin{array}{r l}
0011 & = 3\\
+\underline{0100} & = 4\\
0111 & = 7
\end{array}
\qquad
\begin{array}{r l}
1100 & = -4\\
+\underline{1111} & = -1\\
11011 & = -5
\end{array}$$

$$\begin{array}{r l}
0101 & = 5\\
+\underline{0100} & = 4\\
1001 & = \text{Overflow}
\end{array}
\qquad
\begin{array}{r l}
1001 & = -7\\
+\underline{1010} & = -6\\
10011 & = \text{Overflow}
\end{array}$$

Geometric Depiction of 2's Comp. Integers

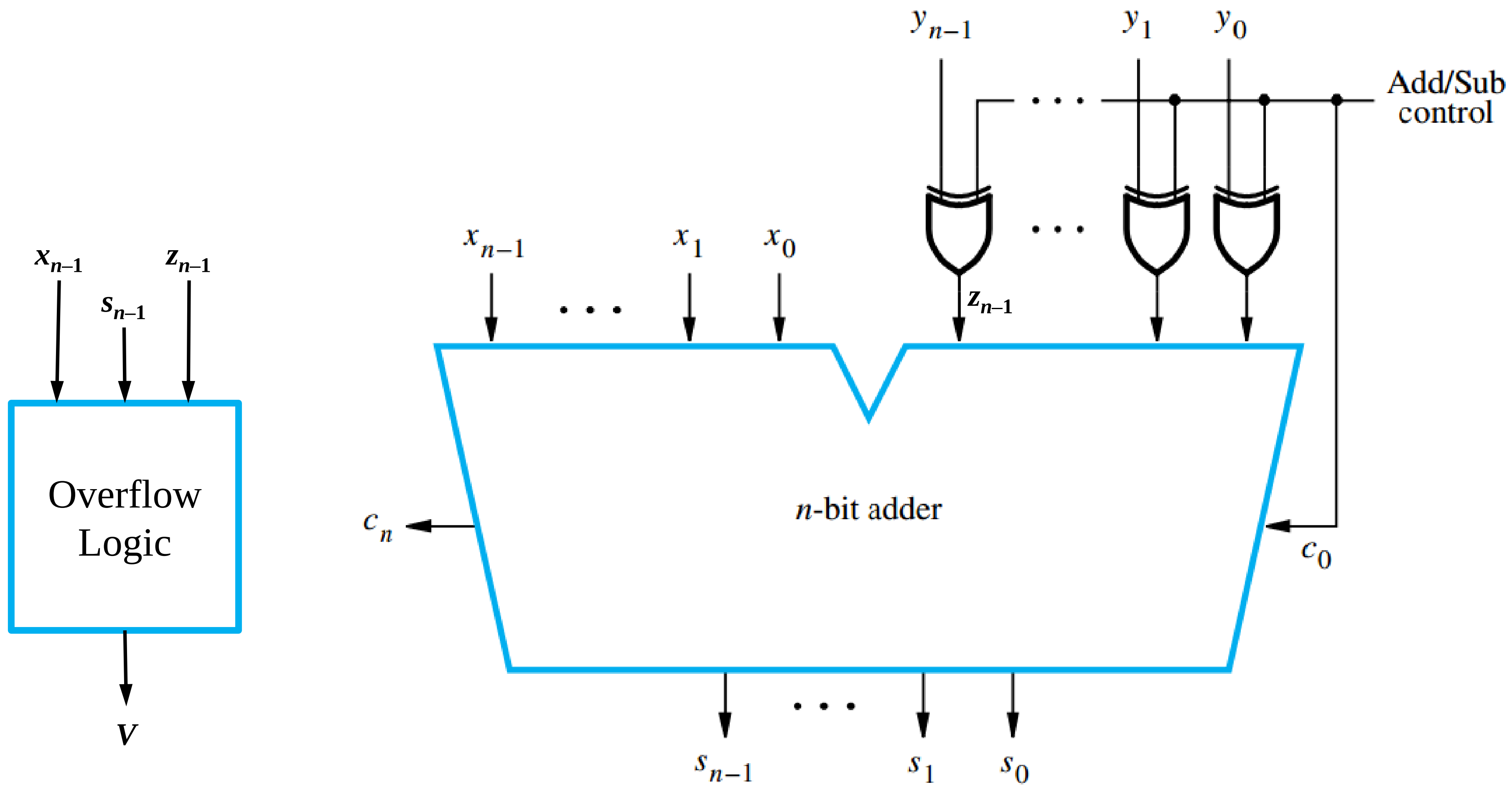


(a) 4-bit numbers



(b) *n*-bit numbers

Binary Addition/Subtraction Logic Circuit.



- Addition ➔ Add/sub control = 0.
- Subtraction ➔ Add/sub control = 1

1-Bit Addition (Full Adder)

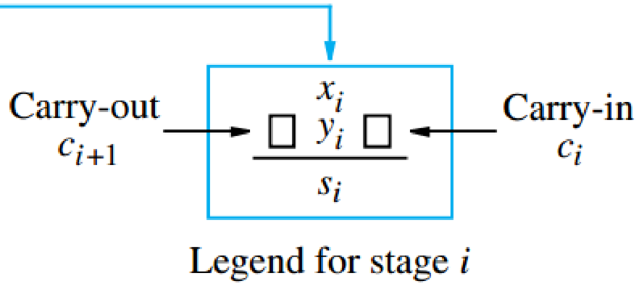
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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

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

Example:

$$\begin{array}{r} X \\ +\,Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\,6 \\ \hline 13 \end{array} = \begin{array}{rccccccccc} & & 0 & & 1 & & \boxed{1} & & 1 & \\ + & {\scriptstyle 0} & 0 & {\scriptstyle 1} & 1 & {\scriptstyle 1} & 1 & {\scriptstyle 0} & 0 & {\scriptstyle 0} \\ \hline & & 1 & & 1 & & 0 & & 1 & \end{array}$$



Legend for stage i

At the stage *i*:

<u>Input:</u>

x_i is ith bit of x

y_i is ith bit of y

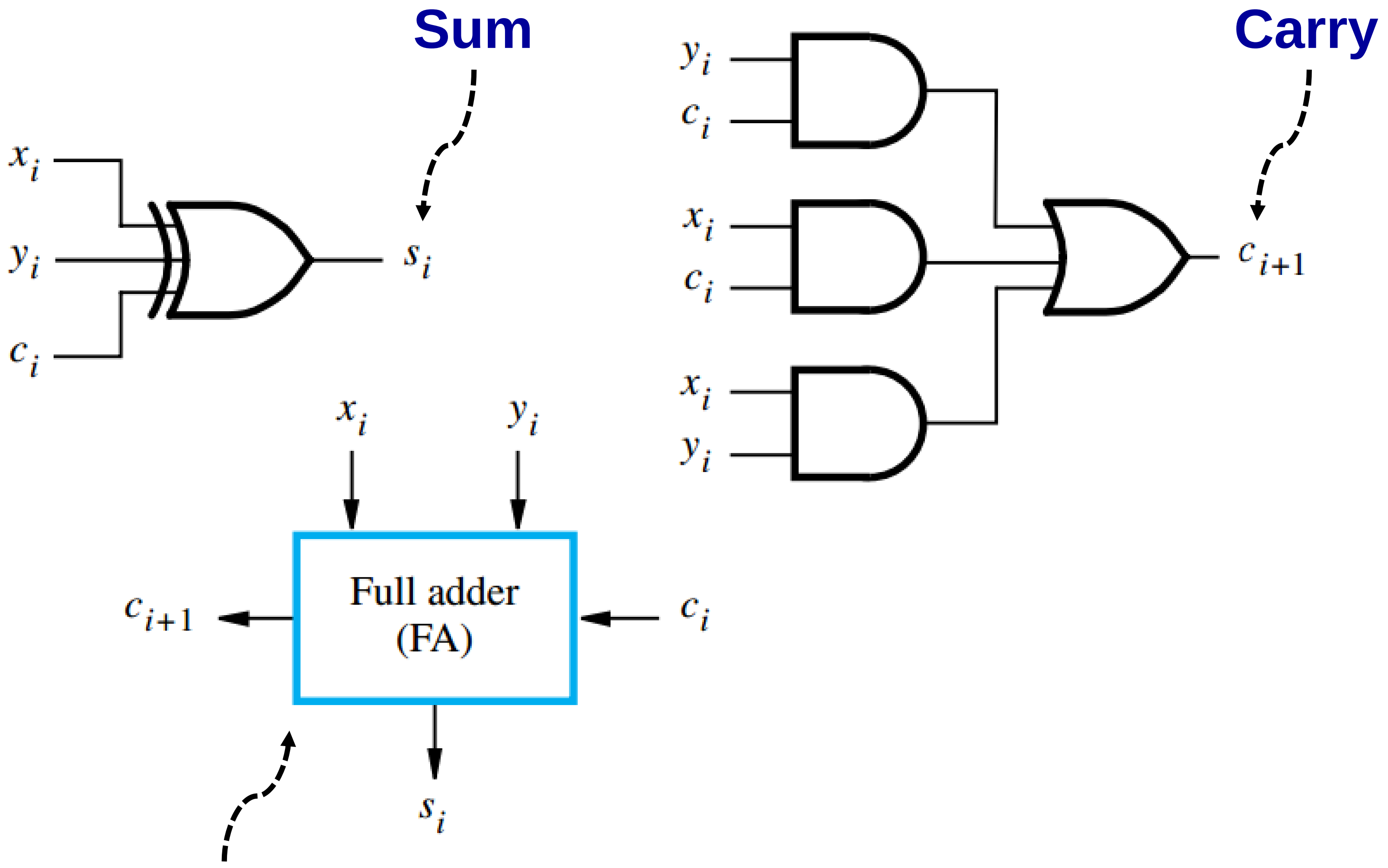
c_i is carry-in from stage i-1

<u>Output:</u>

s_i is the sum

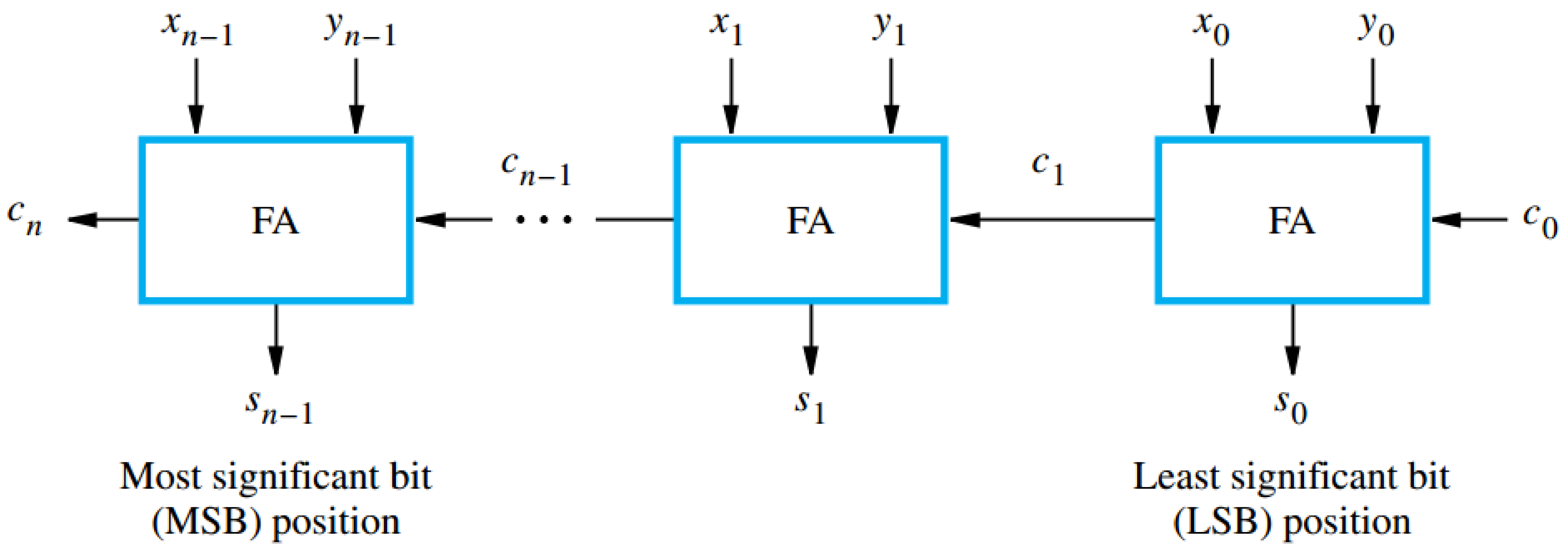
c_{i+1} carry-out to stage i+1

Addition Logic for a Single Stage



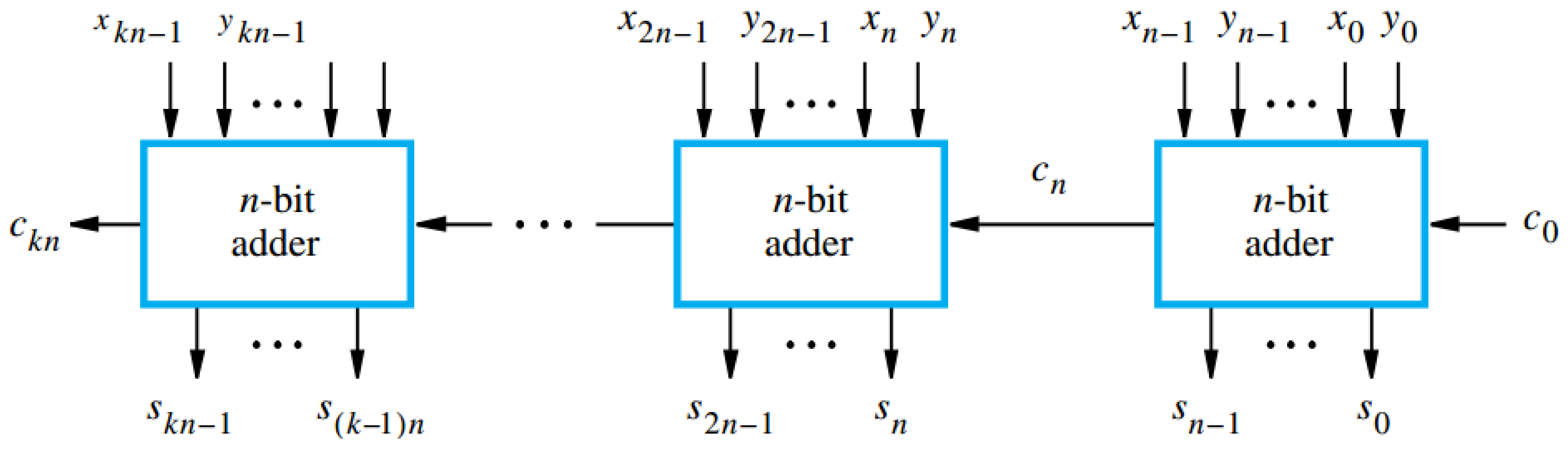
Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

An *n*-bit Ripple-Carry Adder



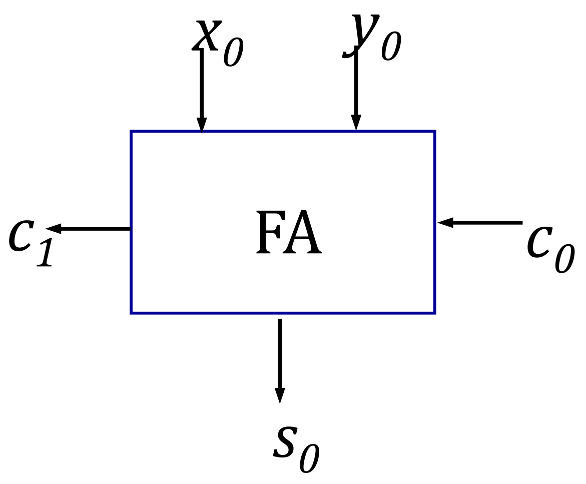
- Cascade *n* full adder (FA) blocks to form a *n*-bit adder.
- Carries propagate or ripple through this cascade ➔ *n*-bit ripple carry adder.
- Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

Cascade of *k* *n*-bit Adders



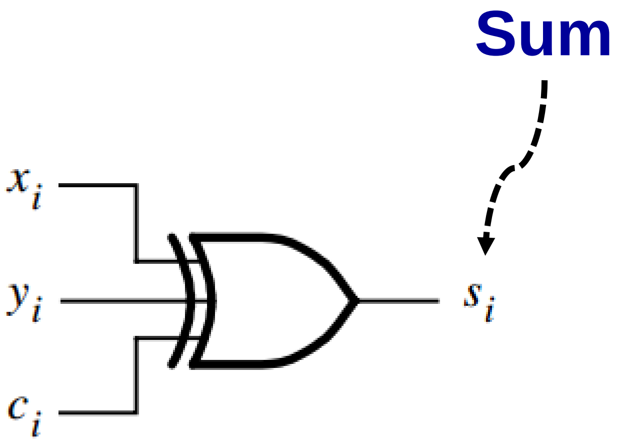
- *k* *n*-bit numbers can be added by cascading *k* *n*-bit adders.
- Each *n*-bit adder forms a block, so this is cascading of blocks.
- Carries ripple or propagate through blocks ➔ Blocked Ripple Carry Adder.

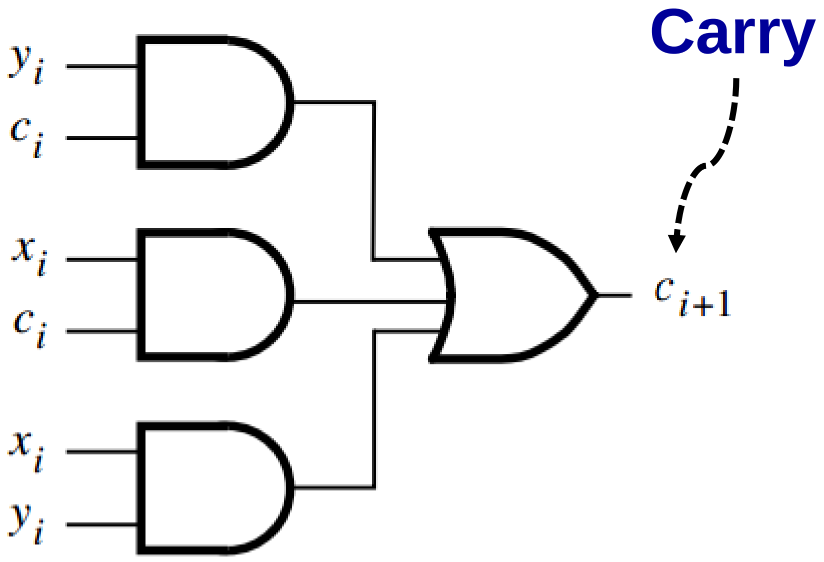
Computing the Add Time



Consider *0th* stage:

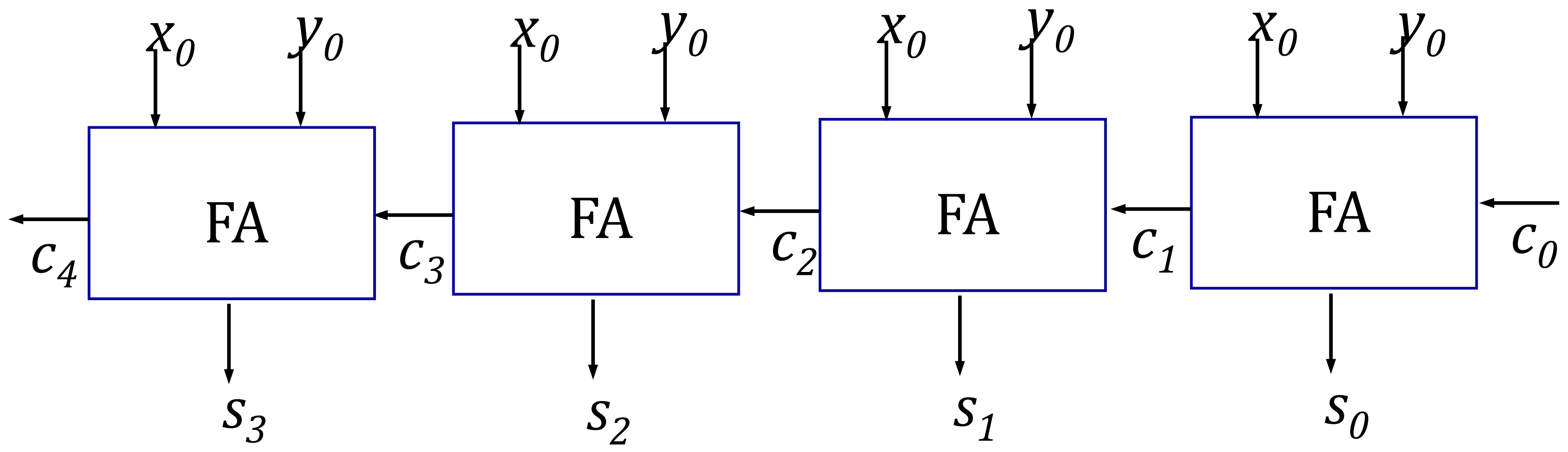
- c_1 is available after 2 gate delays.
- s_1 is available after 1 gate delay.





Computing the Add Time (*cont.*)

Cascade of 4 Full Adders, or a 4-bit adder



- s_0 available after 1 gate delay, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

For an *n*-bit ripple-carry adder:
s_{n-1} is available after *2n-1* gate delays
c_n is available after *2n* gate delays.

Fast Addition

Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as:

$$c_{i+1} = x_i y_i + (x_i \oplus y_i) c_i$$

We can write:

$$c_{i+1} = G_i + P_i c_i$$
$$where\; G_i = x_i y_i \;\; and \;\; P_i = x_i \oplus y_i$$

- G_i is called **generate function.**
- P_i is called **propagate function.**
- G_i and P_i are computed only from x_i and y_i and not c_i

➔ they can be computed in **one gate delay** from X and Y.

Carry-Lookahead Adder – Main Idea

$$c_{i+1} = G_i + P_i c_i$$

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow c_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} c_{i-1})$$

$$continuing$$

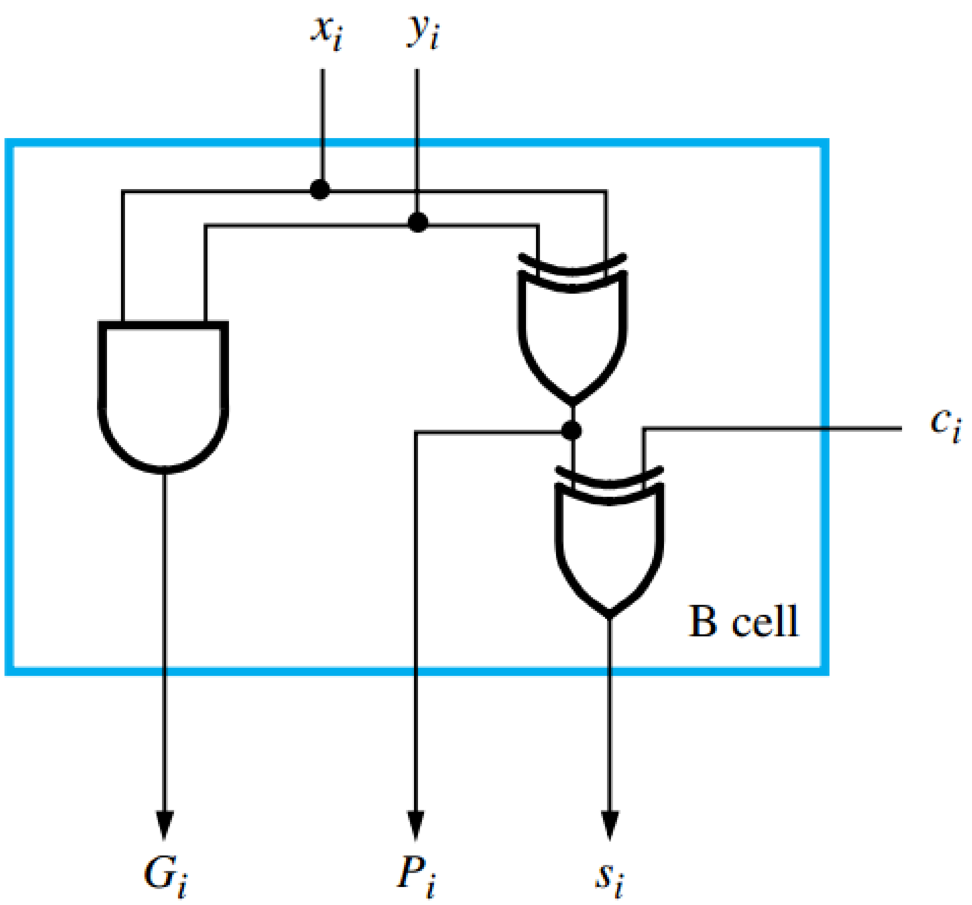
$$\Rightarrow c_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} c_{i-2}))$$

$$until$$

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

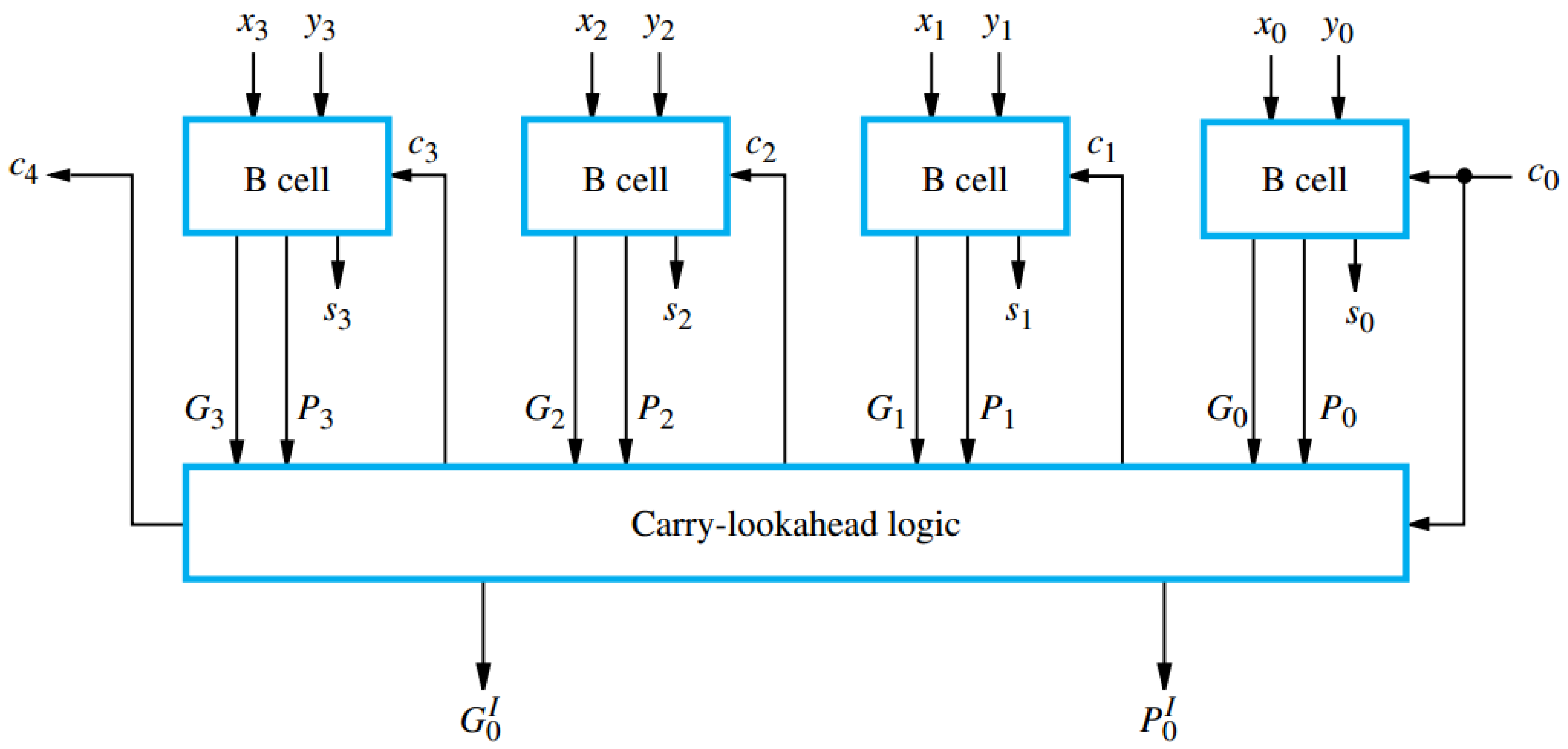
- All carries can be obtained **3 gate** delays from *x*, *y* and c_0.
  - One gate delay for P_i and G_i
  - Two gate delays in the AND-OR circuit for c_{i+1}
- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of *n*, *n*-bit addition requires only **4 gate delays**.
- This is called Carry Lookahead adder.

Carry-Lookahead adder – Basic Cell



Bit-stage cell

Carry-Lookahead Adder – Structure



4-bit carry-lookahead adder

Carry-Lookahead adder – Limitation

- Performing *n*-bit addition in 4 gate delays independent of *n* is good only theoretically because of fan-in constraints!

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

- Last AND gate and OR gate require a fan-in of (n+1) for an n-bit adder.
  - For a 4-bit adder (n=4) fan-in of 5 is required.
  - Practical limit for most gates!
- In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders.

➔ Blocked Carry-Lookahead adder.

Blocked Carry-Lookahead adder – Main Idea

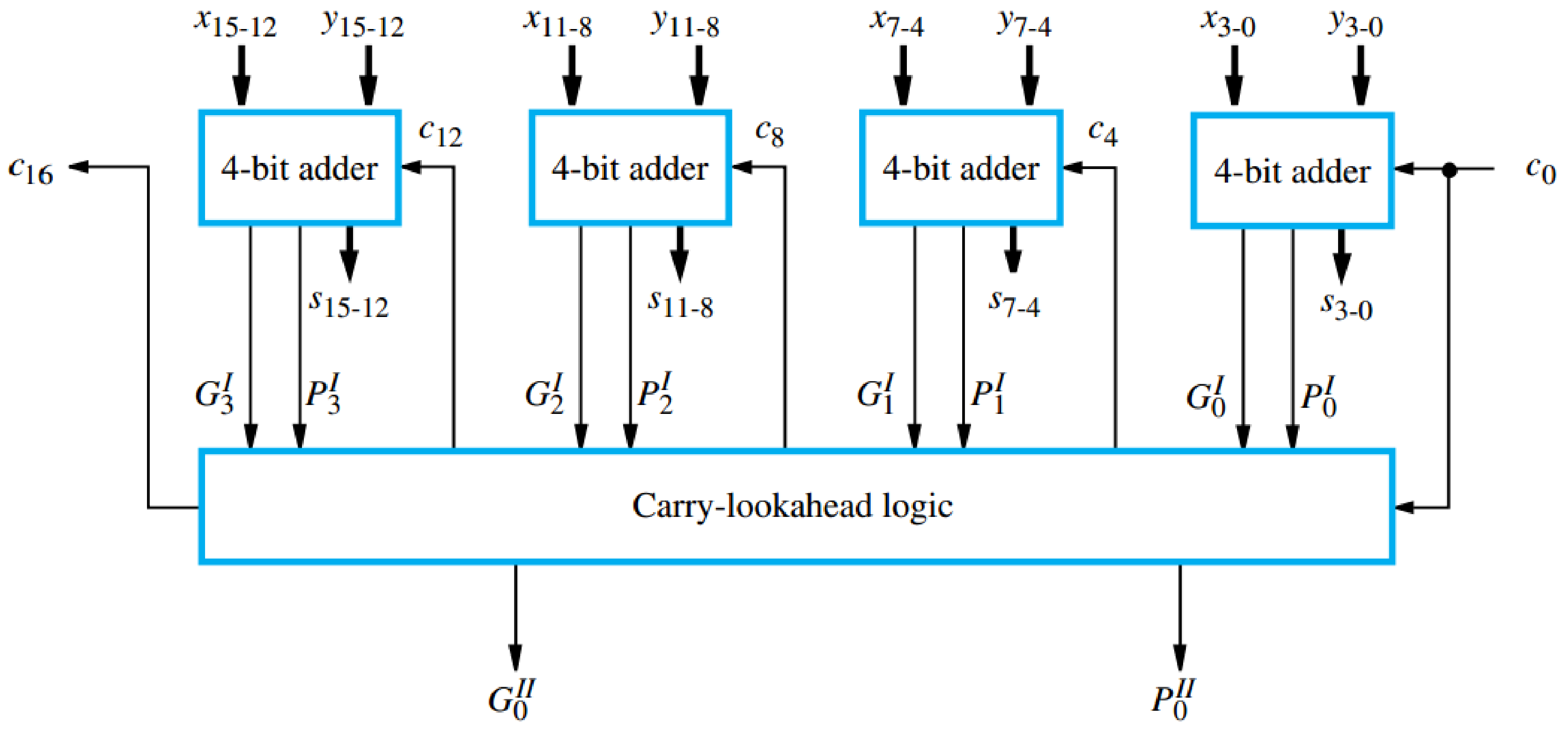
- Carry-out from a 4-bit block can be given as:

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

- Rewrite this as: $c_4 = G_0^I + P_0^I c_0$
  - Where: $G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$
  - And: $P_0^I = P_3P_2P_1P_0$
  - Known as: **high-order** generate/propagate functions.
- To build a 16-bit blocked carry-lookahead adder:
  - Use a carry-lookahead logic block to connect the high-order generate/propagate functions from **4 4-bit carry-lookahead adders** such that:

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^I G_0^I + P_3^I P_2^I P_1^I P_0^I c_0$$

Blocked Carry-Lookahead adder – Structure



- Time taken to produce s_{15}

  $= 1\ (X,Y \rightarrow P,G) + 2\ (P,G \rightarrow P^I,G^I)$

  $+ 2\ (P^I,G^I \rightarrow c_{12}) + 2\ (c_{12} \rightarrow c_{15})$

  $+ 1\ (c_{15} \rightarrow s_{15}) =$ **8 gate delays**

Reading Material

- Stallings, Chapter 10:
  - Pages 320-331
- Hamacher, Chapter 9:
  - Pages 336-344