ECE 341

Lecture # 5

Instructor: Zeshan Chishti
zeshan@ece.pdx.edu

October 13, 2014

Portland State University

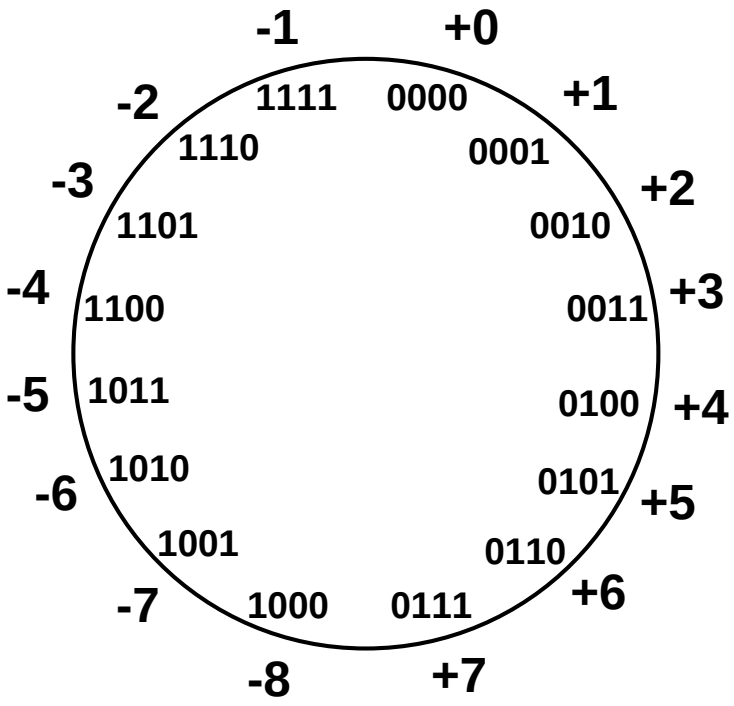
Lecture Topics

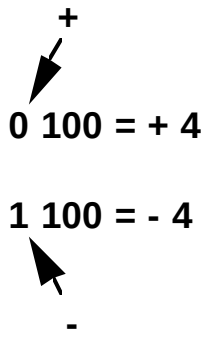
- Computer Arithmetic
  - 2's Complement Representation
  - Addition and subtraction of signed integers
    - Binary Addition
    - Overflow Conditions
    - Ripple-carry Adders
  - Design of Fast Adders
    - Carry Looakahead Adders (CLA)

- Reference:
  - Chapter 9: Sections 9.1, 9.2

Two's Complement Representation

- N-bit binary digit can represent any integer between -2^{N-1} and $+2^{N-1}-1$
- Most significant bit (MSB) indicates sign of the integer
- To obtain representation for a signed integer
  - Invert the bits (1's complement)
  - Add 1
  - E.g., +4 = 0100
    -4 = 1011 + 1 = 1100
- To add a +ve number, go clockwise,
- To add a –ve number, go anticlockwise

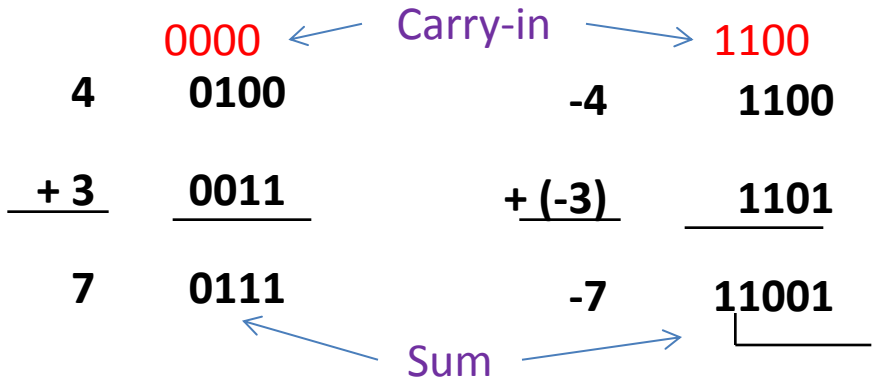




Addition and Subtraction – 2's Complement

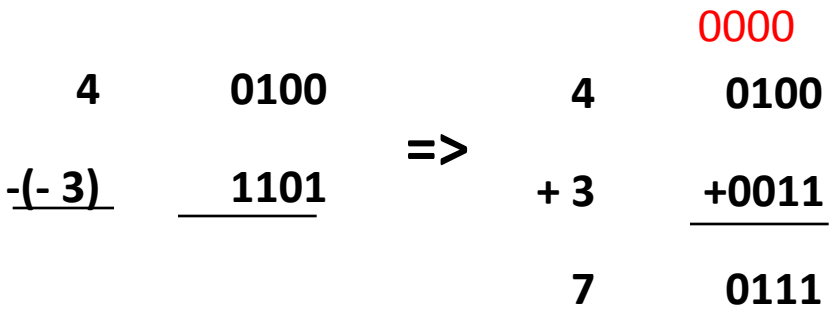
Addition

- Bit pairs added from LSB to MSB
- Each addition takes bit values and *carry-in* as input, generates *sum* and *carry-out* as outputs
- *Carry-out* at any bit position becomes *carry-in* for next bit position



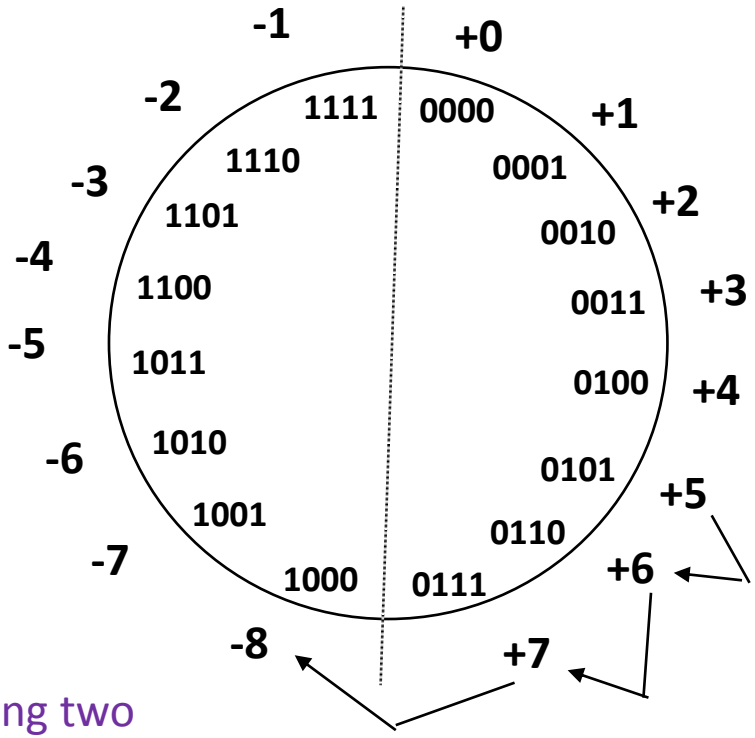
Subtraction

- Take the 2's complement of the second operand
- Add to the first operand



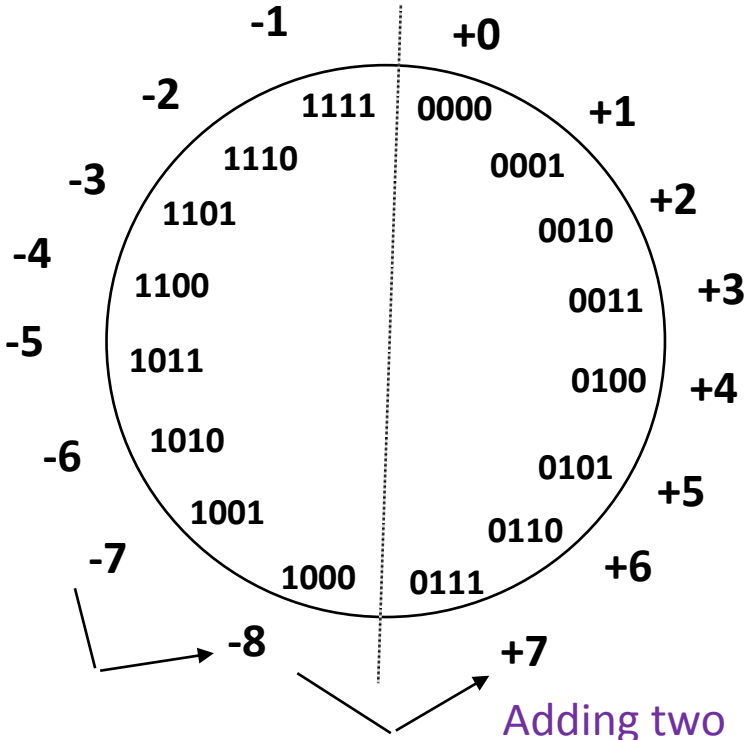
Simpler addition/subtraction scheme makes two's complement the most common choice for integer number systems

Overflow in Integer Arithmetic



Adding two positive integers results in a negative integer

5 + 3 = -8
Should be +8



-7 - 2 = +7
Should be -9

Adding two negative integers results in a positive integer

When the result of an add/subtract cannot be represented by the same number of bits as the operands, overflow occurs

Overflow Conditions

	0 1 1 1	
5	0 1 0 1	
3	0 0 1 1	
-8	1 0 0 0	
Overflow		

	1 0 0 0	
-7	1 0 0 1	
-2	1 1 0 0	
7	1 0 1 1 1	
Overflow		

For addition:

- Overflow cannot occur if the sign of two operands is different
- Overflow occurs when the sign of two operands is the same AND the sign of result is different from the sign of operands

For both addition and subtraction:

- Overflow occurs when the carry-in to MSB is different from the carry-out of MSB

Logic Specification for Binary Addition

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

At the i^{th} stage:

Input:

c_i is the carry-in

Output:

s_i is the sum

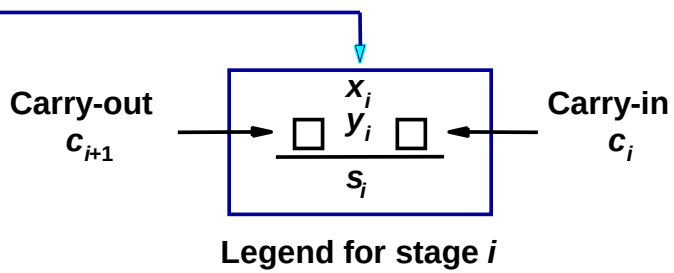
c_{i+1} carry-out to $(i+1)^{st}$ stage

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

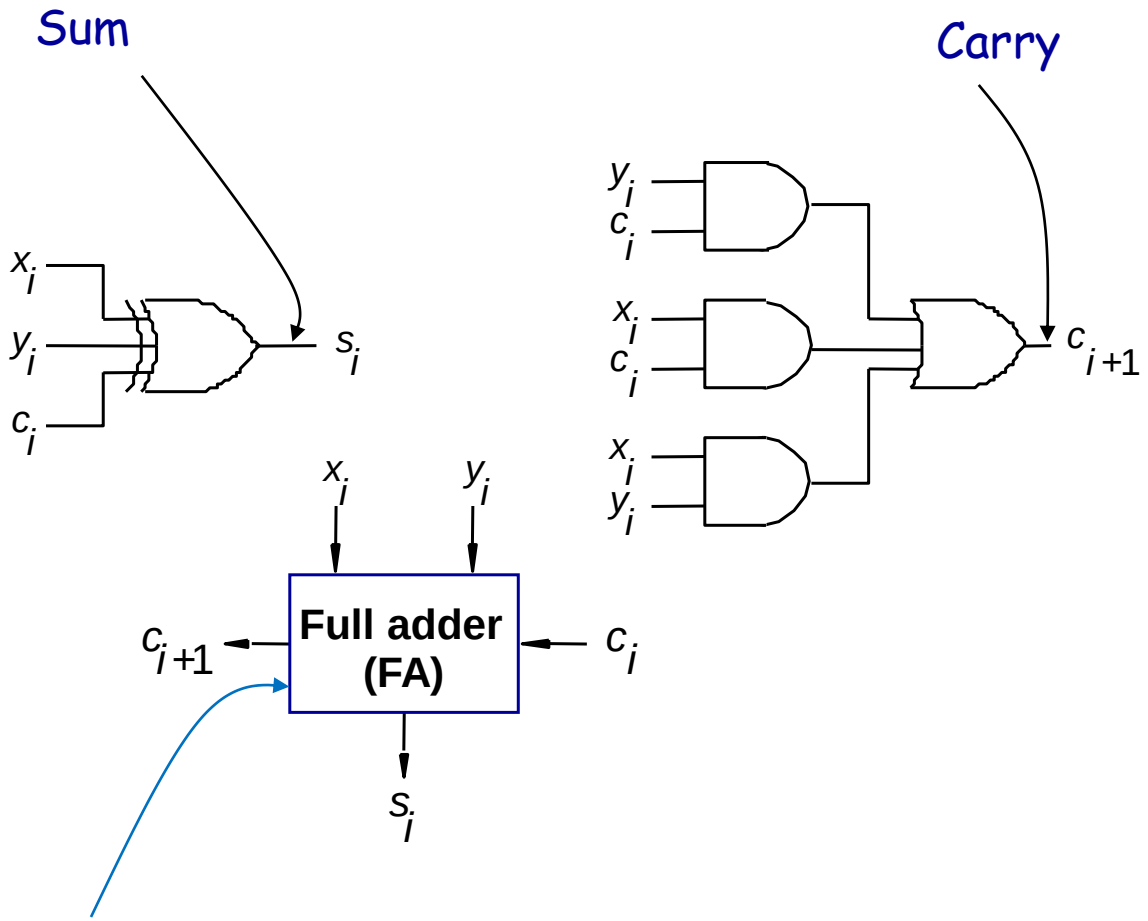
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

$$\begin{array}{r} X \\ +\ Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\ 6 \\ \hline 13 \end{array} = \begin{array}{r} 0\ \ 1 \\ +\ 0\ 0\ 1\ 1 \\ \hline 1\ \ 1\ \ 0\ \ 1 \end{array} \quad \begin{array}{r} 1\ \ 1 \\ 1\ 1\ 0\ 0 \end{array}$$



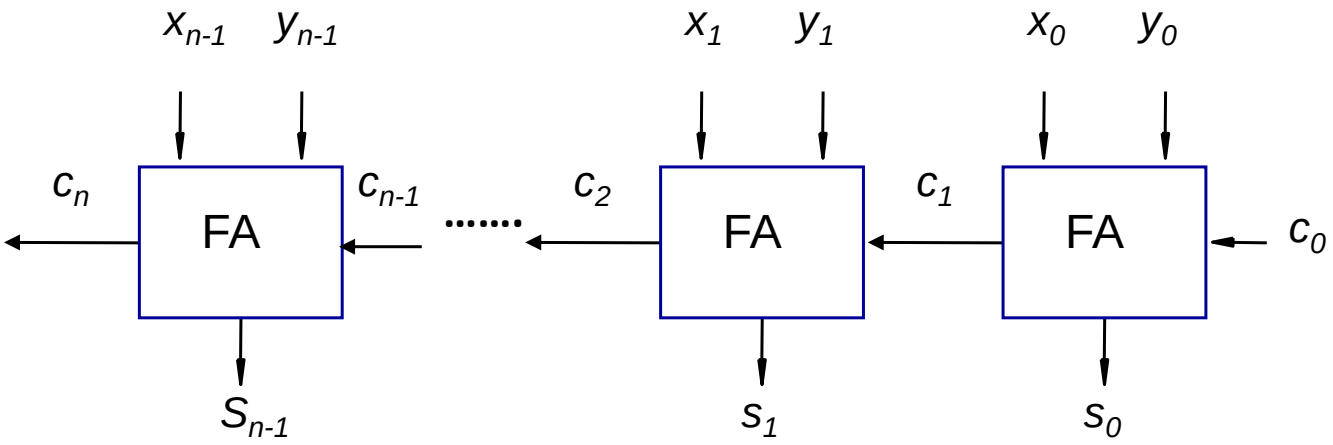
Addition Logic for a Single Stage



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

n-bit 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**

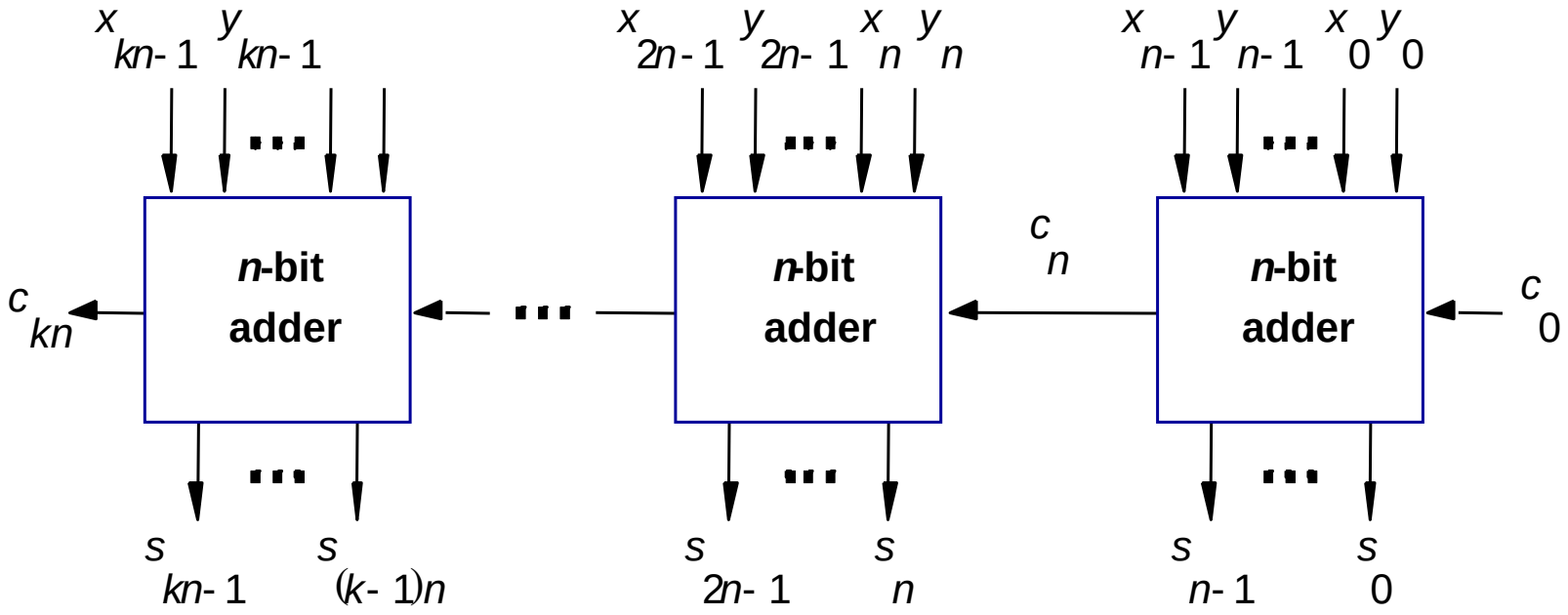


Most significant bit (MSB) position

Least significant bit (LSB) position

kn-bit Adder

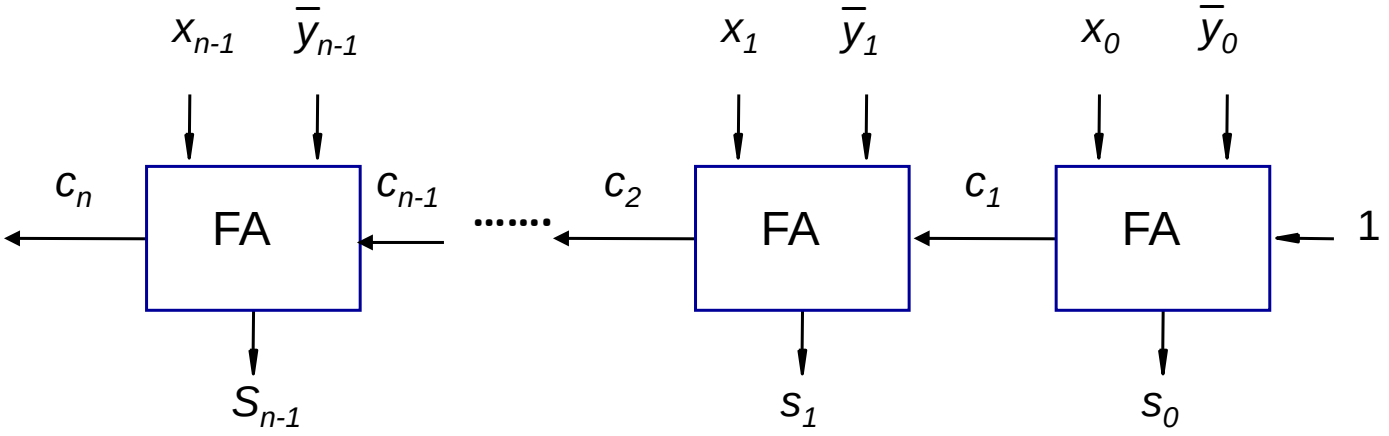
Two numbers each having *kn*-bits can be added by cascading *k* *n*-bit adders
For example, 8 4-bit adders can be cascaded to add two 32-bit numbers



Each *n*-bit adder forms a block
Carries ripple or propagate through blocks, **Blocked Ripple Carry Adder**

n-bit Subtractor

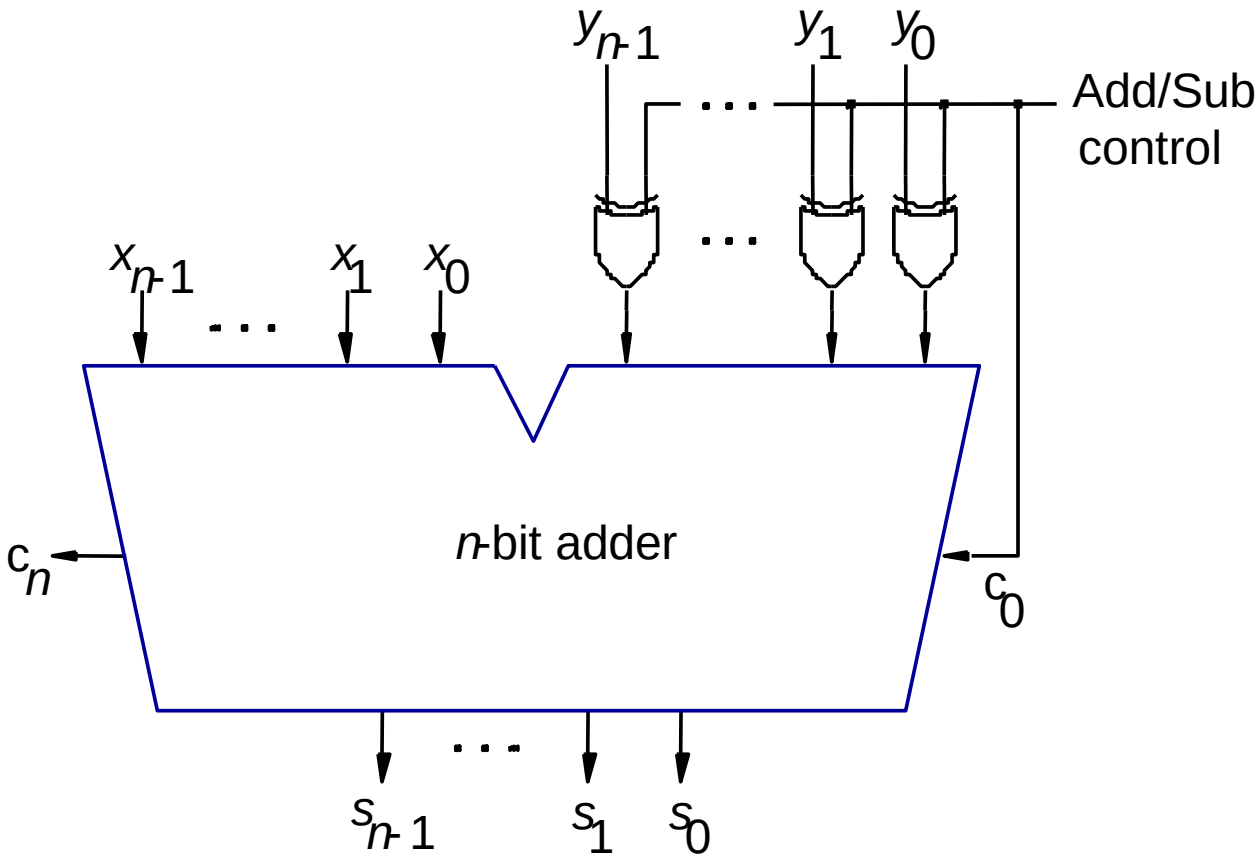
- Recall $X - Y$ is equivalent to adding 2's complement of Y to X.
- 2's complement is equivalent to 1's complement + 1.
- $X - Y = X + \overline{Y} + 1$
- Carry-in c_0 into the LSB position is equal to "1" to perform subtraction



Most significant bit (MSB) position

Least significant bit (LSB) position

n-bit Adder/Subtractor



Add/sub control = 0 implies addition.

Add/sub control = 1 implies subtraction.

Detecting Overflows

Recall that for **addition**

- Overflow can occur only when the sign of two operands is the same
- Overflow occurs if the sign of result is different from the sign of operands
- Recall that the MSB represents the sign.
  - x_{n-1}, y_{n-1}, s_{n-1} represent the sign of operand *x,* operand *y* and result *s* respectively
- Circuit to detect overflow in an **n-bit adder** can be implemented as:

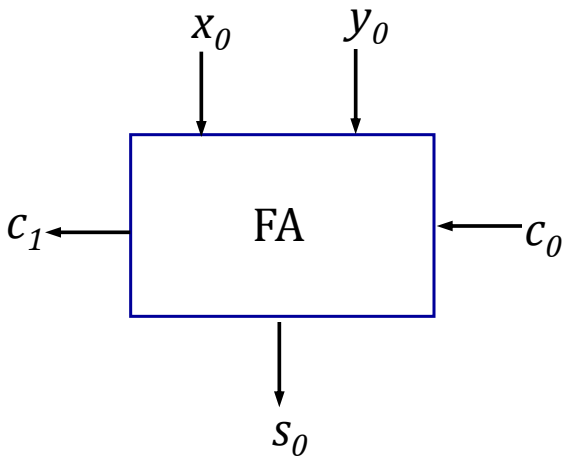
$$Overflow = x_{n-1}y_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1}$$

Recall that for both **addition/subtraction**,

- Overflow occurs when carry-in to MSB is different from carry-out of MSB
- Simpler circuit to detect overflow in an **n-bit adder/subtractor** is:

$$Overflow = c_n \oplus c_{n-1}$$

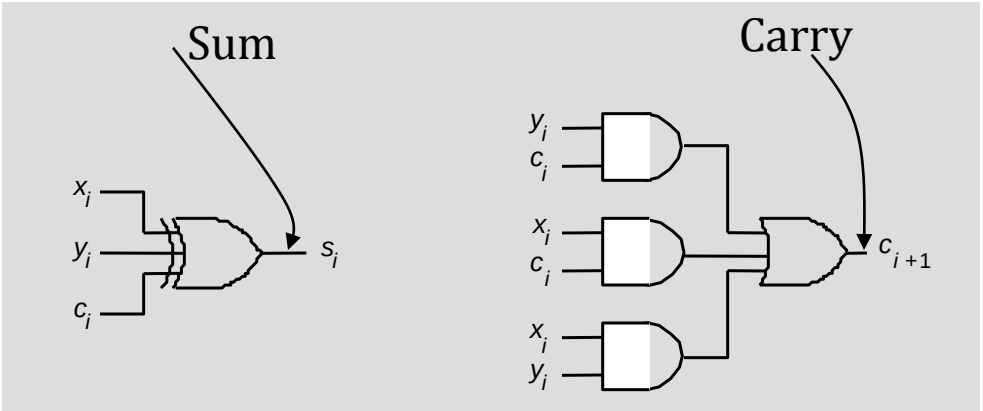
Computing the Add Time



Consider 0^{th} stage:

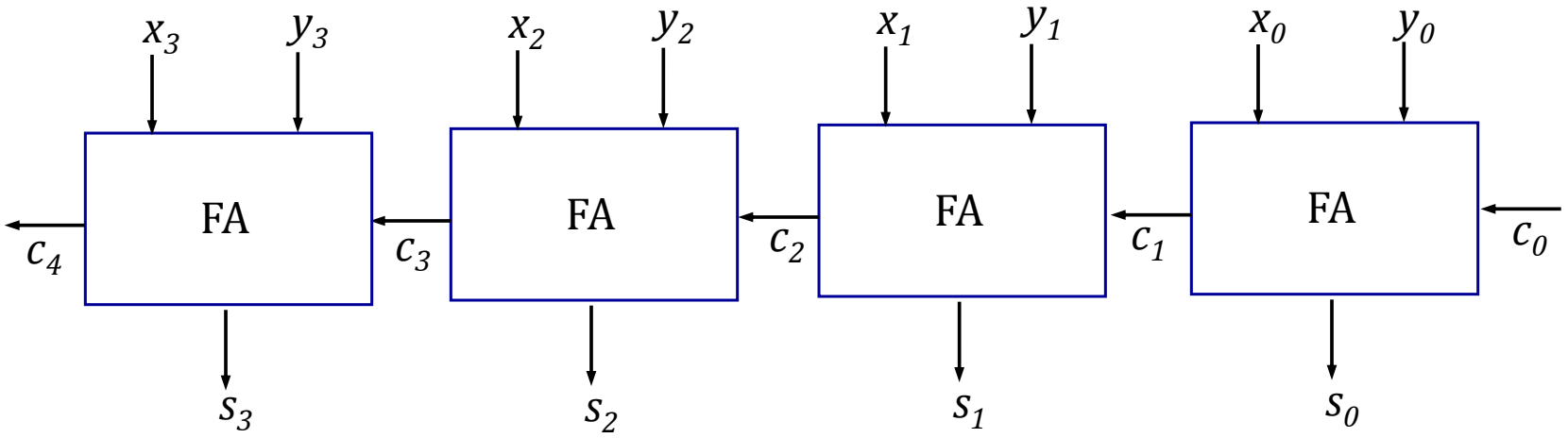
- c_1 is available after 2 gate delays.
- s_0 is available after 1 gate delay.

- c_i needed to compute s_i
- Carry propagation is the critical path in a ripple-carry adder



Computing the Add Time (contd..)

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



- s_0 available after 1 gate delays, 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 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:

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

We can write:

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

$$where\ G_i = x_i y_i\ and\ P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i computed *only* from x_i and y_i and *not* c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder

Carry Lookahead Adder (CLA)

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

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

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

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

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

- c_{i+1} in the above equation depends only on propagate functions, generate functions and c_0
- Calculation of c_{i+1} no longer requires calculation of c_i, c_{i-1}, c_{i-2}……. etc. => all carries can be computed in parallel
- This is called **carry lookahead addition (CLA)**

Understanding Carry-Lookahead Addition

- CLA relies on **generate (G)** and **propagate (P)** functions
- An *n*-bit CLA computes G_i and P_i at each of its *n* stages

$$G_i = x_i y_i \text{ and } P_i = x_i + y_i$$

- $\mathbf{G_i = 1}$ implies that stage *i* **generates** a carry-out of 1, independent of its carry-in
- $\mathbf{P_i = 1}$ implies that stage *i* **propagates** a carry-in of 1 from its input to its output

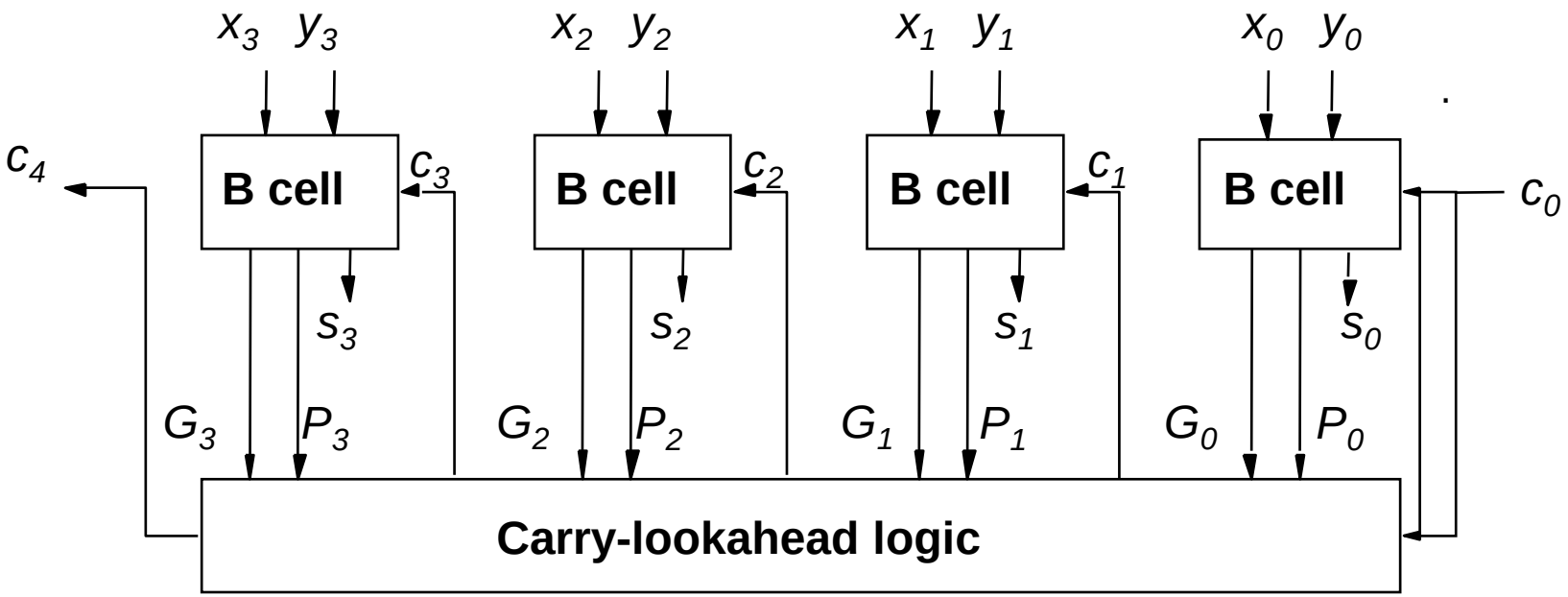
- The carry-out c_{i+1} of stage *i* is equal to 1, if:
  - Stage *i* generates a carry-out (i.e., $G_i = 1$), **or**
  - Any stage *j*, such that $j < i$, generates a carry-out (i.e., $G_j = 1$) and all the stages between *j* and *i*, including *i*, propagate the carry (i.e., $P_{j+1} = P_{j+2} = \ldots = P_i = 1$), **or**
  - $c_0 = 1$ and all the stages up to and including stage *i* propagate the carry (i.e., $c_0 = P_0 = P_1 = \ldots\ldots = P_i = 1$)
- Therefore, the general expression for c_{i+1} in a CLA is:

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

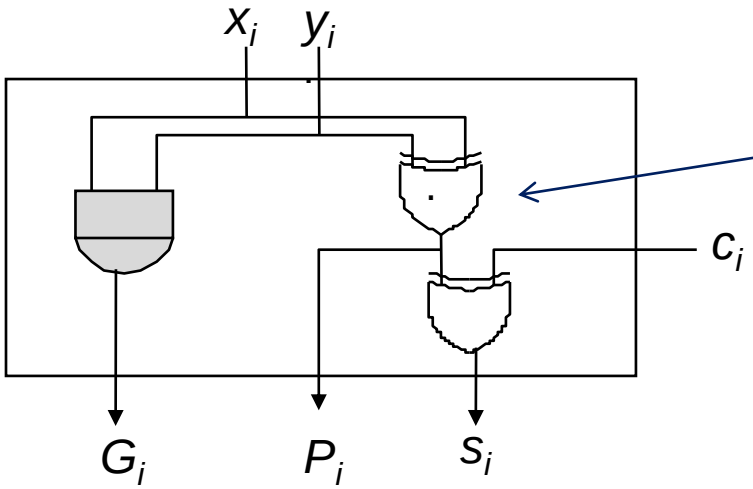
CLA Example

- The general expression for c_{i+1} in CLA is:
  - $c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots\ldots\ldots + P_iP_{i-1}\ldots P_1G_0 + P_iP_{i-1}\ldots P_0c_0$
- Consider 4-bit addition. The carries can be implemented as:
  - $c_1 = G_0 + P_0c_0$
  - $c_2 = G_1 + P_1G_0 + P_1P_0c_0$
  - $c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$
  - $c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$
- Above equations can be used to implement a 4-bit carry-lookahead adder

4-bit Carry Lookahead adder



B-cell for a single stage



P_i implemented as XOR of x_i and y_i instead of $x_i + y_i$. This is ok because when $x_i = 1$, $y_i = 1$, G_i is equal to 1.