Computer Logic Design Fundamentals

Chapter 3 – Combinational Logic Design

Part 3– Arithmetic Functions

Prof. Yueming Wang

ymingwang@zju.edu.cn

College of Computer Science and Technology,
Zhejiang University

Overview

- **Part 3 – Arithmetic Functions**
  - **Iterative combinational circuits**
  - **Binary adders**
    - **Half and full adders**
    - **Ripple carry and carry lookahead adders**
  - **Binary subtraction**
  - **Binary adder-subtractors**
    - **Signed binary numbers**
    - **Signed binary addition and subtraction**
    - **Overflow**
  - ***Binary multiplication**
  - **Other arithmetic functions**
    - **Design by contraction**

Iterative Combinational Circuits

- **Arithmetic functions**
  - **Operate on binary vectors**
  - **Use the same subfunction in each bit position**
- **Can design functional block for subfunction and repeat to obtain functional block for overall function**
- *Cell* **- subfunction block**
- *Iterative array* **- an array of interconnected cells**
- **An iterative array can be in a <u>single</u> dimension (1D) or <u>multiple</u> dimensions**

Block Diagram of a 1D Iterative Array
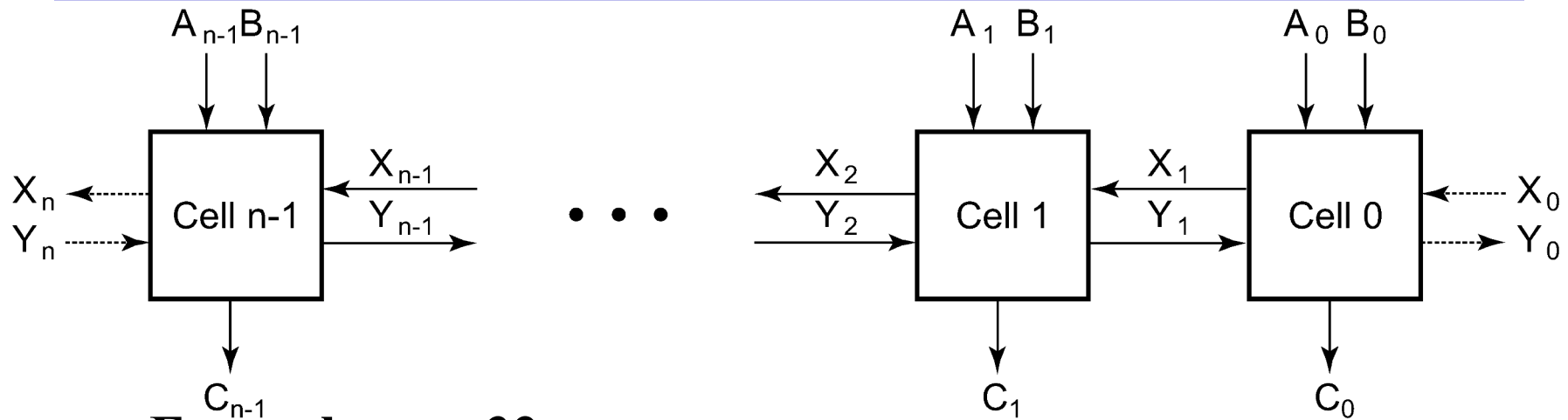
- **Example: n = 32**
  - **Number of inputs = ?**
  - **Truth table rows = ?**
  - **Equations with up to ? input variables**
  - **Equations with huge number of terms**
  - **Design impractical!**
- **Iterative array takes advantage of the regularity to make design feasible**

Functional Blocks: Addition

- **Binary addition used frequently**
- **Addition Development:**
  - *Half-Adder* (HA), a 2-input bit-wise addition functional block,
  - *Full-Adder* (FA), a 3-input bit-wise addition functional block,
  - *Ripple Carry Adder*, an iterative array to perform binary addition, and
  - *Carry-Look-Ahead Adder* (CLA), a hierarchical structure to improve performance.

Functional Block: Half-Adder

- **A 2-input, 1-bit width binary adder that performs the following computations:**

$$
\begin{array}{ccccc}
\mathbf{X} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\mathbf{+\,Y} & \mathbf{+\,0} & \mathbf{+\,1} & \mathbf{+\,0} & \mathbf{+\,1} \\
\hline
\mathbf{C\ S} & \mathbf{0\ 0} & \mathbf{0\ 1} & \mathbf{0\ 1} & \mathbf{1\ 0}
\end{array}
$$

- **A half adder adds two bits to produce a two-bit sum**

- **The sum is expressed as a <u>sum bit</u> , S and a <u>carry bit</u>, C**

- **The half adder can be specified as a truth table for S and C** $\Rightarrow$

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Logic Simplification: Half-Adder

- **The K-Map for S, C is:**
- **This is a pretty trivial map! By inspection:**

$$S = X \ \overline{Y} + \overline{X} \ Y = X \oplus Y$$

$$S = (X + Y) \ \overline{(\overline{X} + \overline{Y})}$$

- **and**

$$C = X \ Y$$

$$C = \overline{(\overline{(X \ Y)})}$$

- **These equations lead to several implementations.**

S   Y

$_0$	**1**$_1$
1$_2$	$_3$

X

C   Y

$_0$	$_1$
$_2$	**1**$_3$

X

Five Implementations: Half-Adder

- **We can derive following sets of equations for a half-adder:**

  (a) $S = X\ \overline{Y} + \overline{X}\ Y$
       $C = X\ Y$

  (b) $S = (X + Y)\ (\overline{X} + \overline{Y})$
       $C = \underline{X\ Y}$

  (c) $S = \overline{(C + \overline{X}\ \overline{Y})}$
       $C = X\ Y$

  (d) $\underline{S} = (\underline{X} + \underline{Y})\ \overline{C}$
       $C = (X + Y)$

  (e) $S = X \oplus Y$
       $C = X\ Y$

- **(a), (b), and (e) are SOP, POS, and XOR implementations for S.**

- **In (c), the C function is used as a term in the AND-NOR implementation of S, and in (d), the $\overline{C}$ function is used in a POS term for S.**
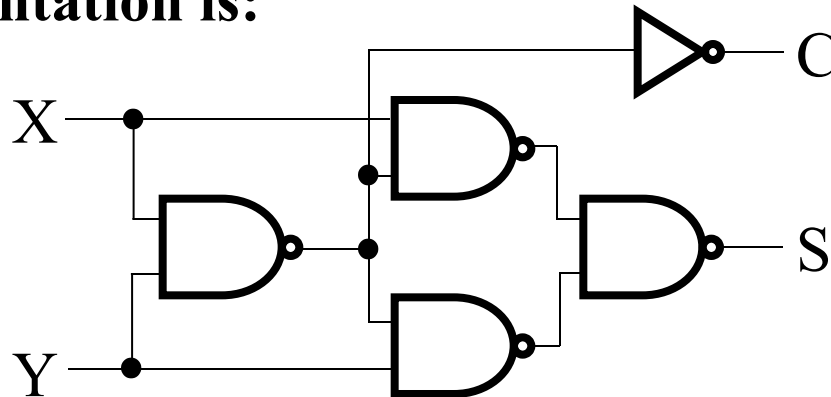
Implementations: Half-Adder

- **The most common half adder implementation is:**

$$S = X \oplus Y$$
$$C = X \cdot Y$$
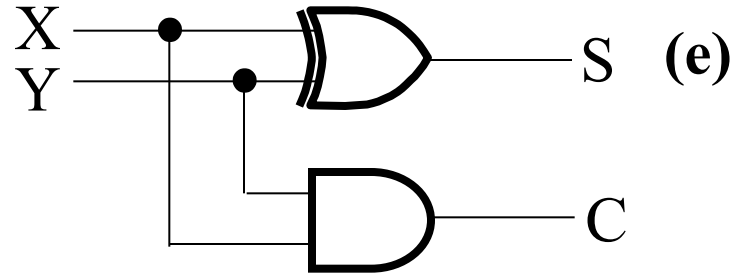
- **A NAND only implementation is:**

$$S = \overline{(X + Y) \cdot C}$$
$$C = \overline{(\overline{(X \cdot Y)})}$$

Functional Block: Full-Adder

- **A full adder is similar to a half adder, but includes a carry-in bit from lower stages.   Like the half-adder, it computes a sum bit, S and a carry bit, C.**

  - **For a carry-in (Z) of 0, it is the same as the half-adder:**

  - **For a carry- in (Z) of 1:**

Z	0	0	0	0
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 0	0 1	0 1	1 0

Z	1	1	1	1
X	0	0	1	1
+ Y	+ 0	+ 1	+ 0	+ 1
C S	0 1	1 0	1 0	1 1

Logic Optimization: Full-Adder

- **Full-Adder Truth Table:**

X	Y	Z	C	S
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

- **Full-Adder K-Map:**

Equations: Full-Adder

- **From the K-Map, we get:**

$$S = X \overline{Y} \overline{Z} + \overline{X} Y \overline{Z} + \overline{X} \overline{Y} Z + X Y Z$$
$$C = X Y + X Z + Y Z$$

- **The S function is the three-bit XOR function (Odd Function):**

$$S = X \oplus Y \oplus Z$$

- **The Carry bit C is 1 if both X and Y are 1 (the sum is 2), or if the sum is 1 and a carry-in (Z) occurs. Thus C can be re-written as:**

$$C = X Y + (X \oplus Y) Z$$

- **The term X·Y is *carry generate*.**

- **The term X⊕Y is *carry propagate*.**

Implementation: Full Adder

- **Full Adder Schematic**

- **Here X, Y, and Z, and C (from the previous pages) are A, B, C_i and C_o, respectively. Also,**

  **G = generate and
  P = propagate.**

- **Note: This is really a combination of a 3-bit odd function (for S)) and Carry logic (for C_o):**
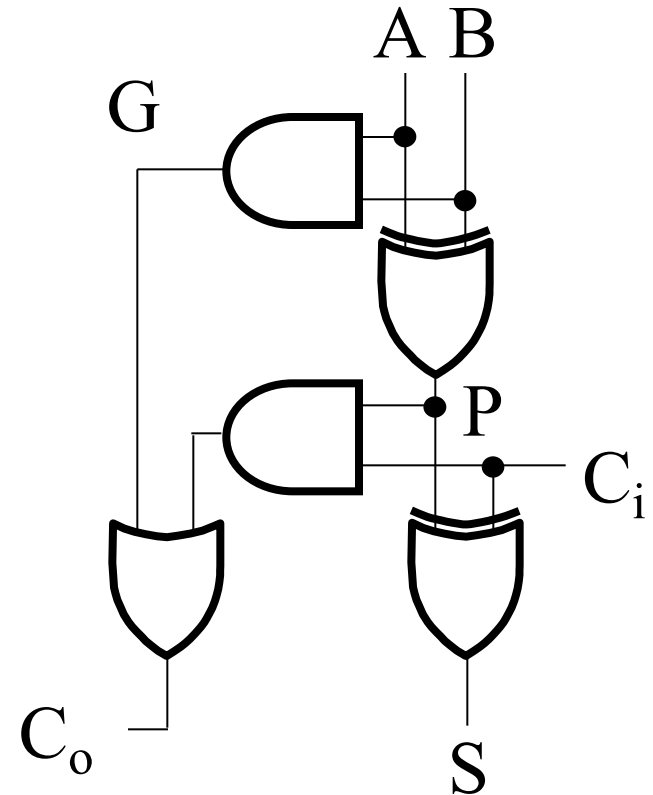
**(G = Generate) OR (P =Propagate AND C_i = Carry In)**

Co = G + P · Ci

Binary Adders

- **To add multiple operands, we "bundle" logical signals together into vectors and use functional blocks that operate on the vectors**

- **Example: <u>4-bit ripple carry adder:</u> Adds input vectors A(3:0) and B(3:0) to get a sum vector S(3:0)**

- **Note: carry out of cell i becomes carry in of cell i + 1**

| Description	Subscript 3 2 1 0	Name
Carry In	0 1 1 0	C_i
Augend	1 0 1 1	A_i
Addend	<u>0 0 1 1</u>	B_i
Sum	1 1 1 0	S_i
Carry out	0 0 1 1	C_{i+1}

4-bit Ripple-Carry Binary Adder

- **A four-bit Ripple Carry Adder made from four 1-bit Full Adders:**
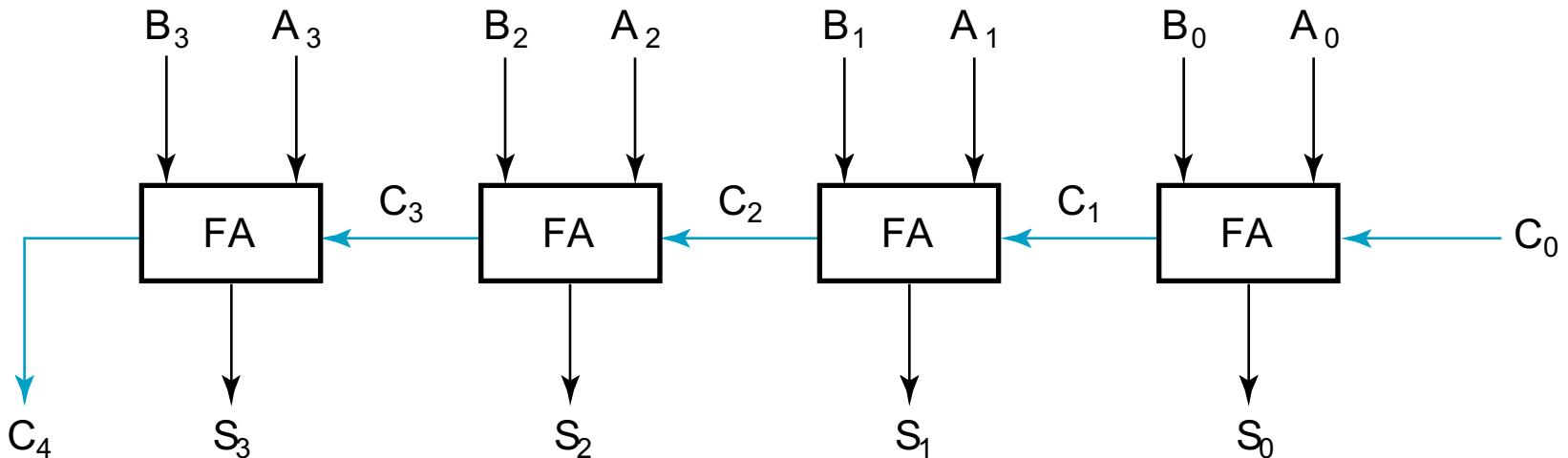
Carry Propagation & Delay

- One problem with the addition of binary numbers is the length of time to propagate the ripple carry from the least significant bit to the most significant bit.

- The gate-level propagation path for a 4-bit ripple carry adder of the last example:
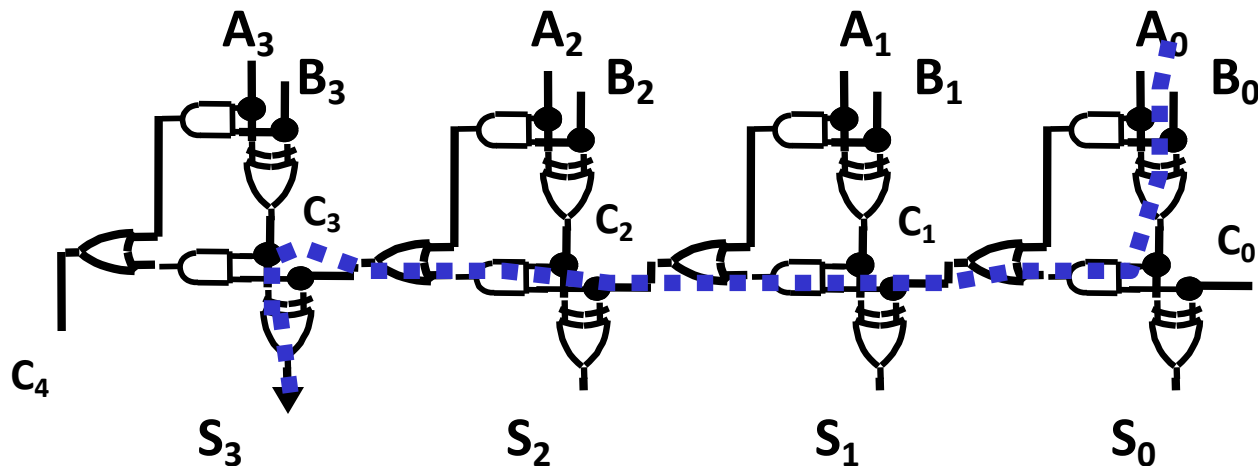
- Note: The "long path" is from A_0 or B_0 though the circuit to S_3.

Carry Lookahead

- **Given Stage i from a Full Adder, we know that there will be a <u>carry generated</u> when $A_i = B_i =$ "1", whether or not there is a carry-in.**

- **Alternately, there will be a <u>carry propagated</u> if the "half-sum" is "1" and a carry-in, C_i occurs.**

- **These two signal conditions are called *generate*, denoted as G_i, and *propagate*, denoted as P_i respectively and are identified in the circuit:**

A_i B_i

G_i

P_i

C_i

C_{i+1}

S_i
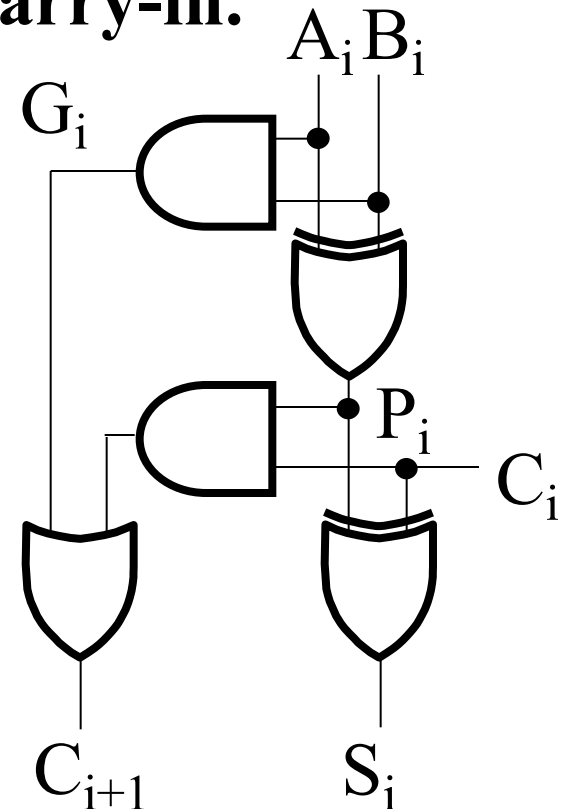
Carry Lookahead (continued)

- **In the ripple carry adder:**
  - **G_i, P_i, and S_i are <u>local</u> to each cell of the adder**
  - **C_i is also local each cell**
- **In the carry lookahead adder, in order to reduce the length of the carry chain, C_i is changed to a more global function spanning multiple cells**
- **Defining the equations for the Full Adder in term of the P_i and G_i:**

$$P_i = A_i \oplus B_i \qquad\qquad G_i = A_i B_i$$

$$S_i = P_i \oplus C_i \qquad\qquad C_{i+1} = G_i + P_i C_i$$

Carry Lookahead Development

- C_{i+1} can be removed from the cells and used to derive a set of carry equations spanning multiple cells.

- Beginning at the cell 0 with carry in C_0:

$$C_1 = G_0 + P_0\, C_0$$

$$C_2 = G_1 + P_1\, C_1 = G_1 + P_1(G_0 + P_0\, C_0)$$
$$\quad = G_1 + P_1 G_0 + P_1 P_0\, C_0$$

$$C_3 = G_2 + P_2\, C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0\, C_0)$$
$$\quad = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0\, C_0$$

$$C_4 = G_3 + P_3\, C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1$$
$$\quad + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0\, C_0$$

Carry Look-ahead Adder

Partial full adder
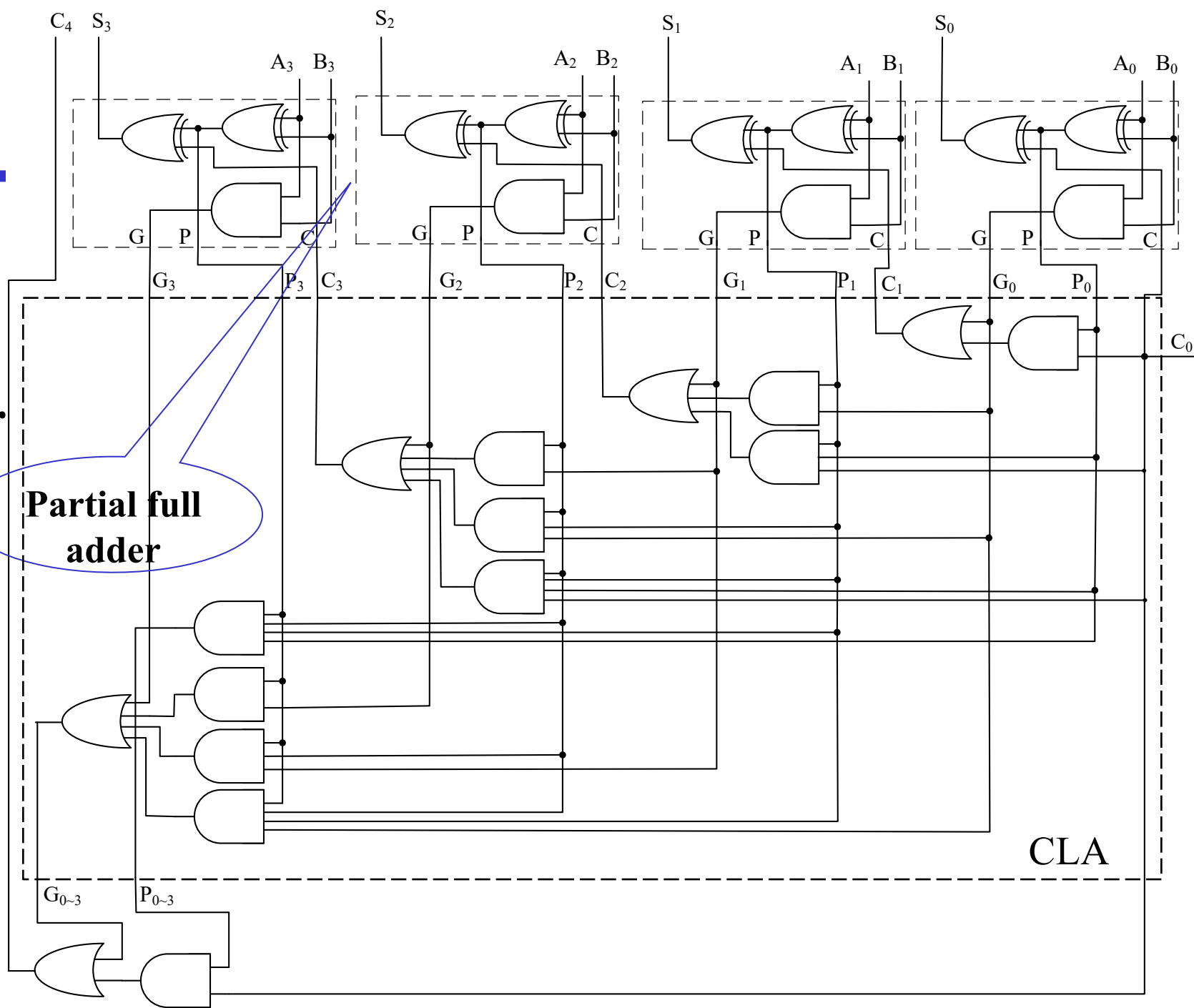
CLA

Group Carry Lookahead Logic

- **Figure 5-6 in the text shows the implementation of these equations for four bits. This could be extended to more than four bits; in practice, due to limited gate fan-in, such extension is not feasible.**

- $C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0\, C_0$

- **Instead, the concept is extended another level by considering *group generate* ($G_{0\text{-}3}$) and *group propagate* ($P_{0\text{-}3}$) functions:**

$$G_{0\sim3} = G_3 + P_3\, G_2 + P_3\, P_2\, G_1 + P_3\, P_2\, P_1\, G_0$$

$$P_{0\sim3} = P_3\, P_2\, P_1\, P_0$$

- **Using these two equations:**

$$C_4 = G_{0\sim3} + P_{0\sim3}\, C_0$$

- **Thus, it is possible to have four 4-bit adders use one of the same carry lookahead circuit to speed up 16-bit addition**

Group Carry Lookahead Logic (Cont.)

- $C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 = \textcolor{red}{G_{0 \sim 3} + P_{0 \sim 3} C_0}$

- $C_8 = G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4 + P_7 P_6 P_5 P_4 C_4 = \textcolor{red}{G_{4 \sim 7} + P_{4 \sim 7} C_4}$

- $C_{12} = G_{11} + P_{11} G_{10} + P_{11} P_{10} G_9 + P_{11} P_{10} P_9 G_8 + P_{11} P_{10} P_9 P_8 C_8 = \textcolor{red}{G_{8 \sim 11} + P_{8 \sim 11} C_8}$

- $C_{16} = G_{15} + P_{15} G_{14} + P_{15} P_{14} G_{13} + P_{15} P_{14} P_{13} G_{12} + P_{15} P_{14} P_{13} P_{12} C_{12} = \textcolor{red}{G_{12 \sim 15} + P_{12 \sim 15} C_{12}}$

Group Carry Lookahead Logic (Cont.)

Compare:

4-bit Adder

- $C_1 = G_0 + P_0 C_0$
- $C_2 = G_1 + P_1 C_1$
- $C_3 = G_2 + P_2 C_2$
- $C_4 = G_3 + P_3 C_3$

16-bit Adder

- $C_4 = G_{0\sim3} + P_{0\sim3} C_0$
- $C_8 = G_{4\sim7} + P_{4\sim7} C_4$
- $C_{12} = G_{8\sim11} + P_{8\sim11} C_8$
- $C_{16} = G_{12\sim15} + P_{12\sim15} C_{12}$

Exactly the same structure. So CLA could be used to generate Group Carry.
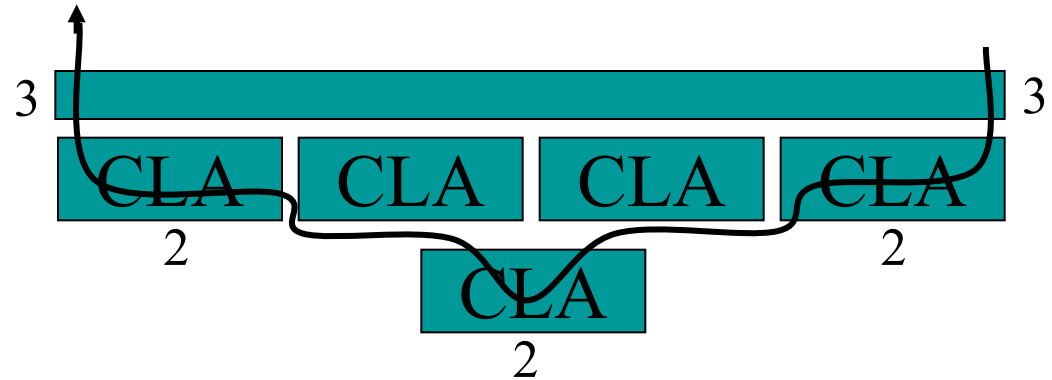
Carry Lookahead Example

- **Specifications:**
  - **16-bit CLA**
  - **Delays:**
    - **NOT = 1**
    - **XOR = Isolated AND = 3**
    - **AND-OR = 2**

- **Longest Delays:**
  - **Ripple carry adder* = $3 + 15 \times 2 + 3 = 36$**
  - **CLA = $3 + 3 \times 2 + 3 = 12$**

*See slide 16

Unsigned Subtraction

- **Algorithm:**
  - **Subtract the subtrahend N from the minuend M**
  - **If no end borrow occurs, then M $\geq$ N, and the result is a non-negative number and correct.**
  - **If an end borrow occurs, the N > M and the difference M $-$ N $+$ 2^n is subtracted from 2^n, and a minus sign is appended to the result.**

- **Examples:**

$$
\begin{array}{cc}
0 & 1 \\
1001 & 0100 \\
-\ 0111 & -\ 0111 \\
\hline
0010 & 1101 \\
 & 10000 \\
 & -\ 1101 \\
\hline
 & (-)\ 0011
\end{array}
$$

Unsigned Subtraction (continued)

- **The subtraction, $2^n - N$, is taking the 2's complement of N**
- **To do both unsigned addition and unsigned subtraction requires:**
- **Quite complex!**
- **Goal: Shared simpler logic for both addition and subtraction**
- **Introduce complements as an approach**

A

B

Binary adder

Borrow

Binary subtractor

Selective
2's complementer

Complement

Subtract/$\overline{\text{Add}}$

S

0   1
Quadruple 2-to-1
multiplexer

Result
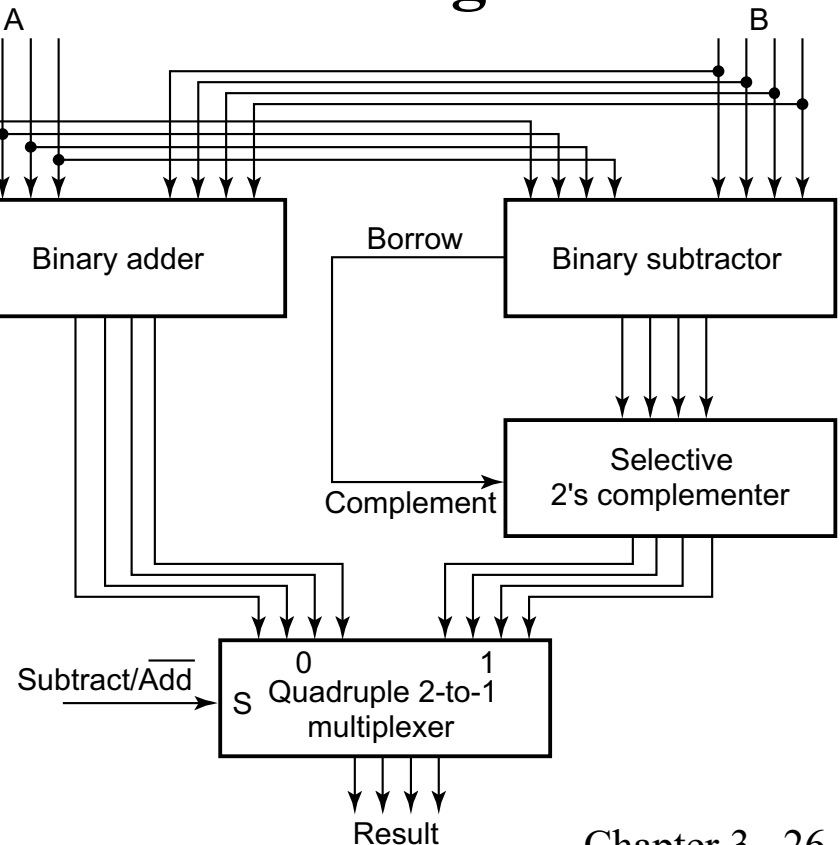
Complements

- **Two complements:**
  - **Diminished Radix Complement of N**
    - $(r - 1)$'s complement for radix r
    - 1's complement for radix 2
    - Defined as $(r^n - 1) - N$
  - **Radix Complement**
    - r's complement for radix r
    - 2's complement in binary
    - Defined as $r^n - N$
- **Subtraction is done by adding the complement of the subtrahend**
- **If the result is negative, takes its 2's complement**

Binary 1's Complement(反码)

- **For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits):**

    $(r^n - 1) = 256 - 1 = 255_{10}$ **or** 11111111_2

- **The 1's complement of 01110011_2 is then:**

    $$\begin{array}{r} 11111111 \\ - \ \underline{01110011} \\ 10001100 \end{array}$$

- **Since the $2^n - 1$ factor consists of all 1's and since $1 - 0 = 1$ and $1 - 1 = 0$, the one's complement is obtained by <u>complementing each individual bit</u> (bitwise NOT).**

Binary 2's Complement(补码)

- **For $r = 2$, $N = 01110011_2$, $n = 8$ (8 digits), we have:**

  $(r^n) = 256_{10}$ or 100000000_2

- **The 2's complement of 01110011 is then:**

$$\begin{array}{r} 100000000 \\ -\ \underline{01110011} \\ 10001101 \end{array}$$

- **Note the result is the 1's complement plus 1, a fact that can be used in designing hardware**

Alternate 2's Complement Method

- **Given: an *n*-bit binary number, beginning at the least significant bit and proceeding upward:**
  - Copy all least significant 0's
  - Copy the first 1
  - Complement all bits thereafter.
- **2's Complement Example:**

  10010<u>100</u>
  - Copy underlined bits:

    <u>100</u>
  - and complement bits to the left:

    <u>01101</u>100

Subtraction with 2's Complement

- **For n-digit, <u>unsigned</u> numbers M and N, find M − N in base 2:**
  - **Add the 2's complement of the subtrahend N to the minuend M:**
    $$M + (2^n − N) = M − N + 2^n$$
  - **If M ≥ N, the sum produces end carry r^n which is discarded; from above, M − N remains.**
  - **If M < N, the sum does not produce an end carry and, from above, is equal to $2^n − (N − M)$, the 2's complement of (N − M).**
  - **To obtain the result − (N − M) , take the 2's complement of the sum and place a − to its left.**

Unsigned 2's Complement Subtraction Example 1

- **Find $01010100_2 - 01000011_2$**

$$\begin{array}{r} 01010100 \\ - \quad \underline{01000011} \end{array} \xrightarrow{\text{2's comp}} \begin{array}{r} {}^{1}01010100 \\ + \quad \underline{10111101} \\ 00010001 \end{array}$$

- **The carry of 1 indicates that result is positive and no correction of the result is required.**

Unsigned 2's Complement Subtraction Example 2

- **Find $01000011_2 - 01010100_2$**

$$\begin{array}{r} 01000011 \\ - \ \underline{01010100} \end{array} \xrightarrow{\text{2's comp}} \begin{array}{r} ^0 01000011 \\ + \ \underline{10101100} \\ 11101111 \end{array} \xrightarrow{\text{2's comp}} 00010001$$

- **The carry of 0 indicates that result is negative and a correction of the result is required.**

- **Result = – (00010001)**

Signed Integers

- **Positive numbers and zero can be represented by unsigned n-digit, radix *r* numbers.   We need a representation for negative numbers.**

- **To represent a sign (+ or –) we need exactly one more bit of information (1 binary digit gives $2^1 = 2$ elements which is exactly what is needed).**

- **Since computers use binary numbers, by convention, the most significant bit is interpreted as a sign bit:**

$$s\ a_{n-2}\ \dots\ a_2 a_1 a_0$$

where:

   $s = 0$ for Positive numbers

   $s = 1$ for Negative numbers
and $a_i = 0$ or 1 represent the magnitude in some form.

Signed Integer Representations

- *Signed-Magnitude* – here the n – 1 digits are interpreted as a positive magnitude.

- *Signed-Complement* – here the digits are interpreted as the rest of the complement of the number.   There are two possibilities here:
  - *Signed 1's Complement*
    - Uses 1's Complement Arithmetic
  - *Signed 2's Complement*
    - Uses 2's Complement Arithmetic

Signed Integer Representation Example

- **r =2, n=3**

Number	Sign -Mag.	1's Comp.	2's Comp.
+3	011	011	011
+2	010	010	010
+1	001	001	001
+0	000	000	000
– 0	100	111	—
– 1	101	110	111
– 2	110	101	110
– 3	111	100	101
– 4	—	—	100

Signed-Magnitude Arithmetic

- **If the parity of the three signs is 0:**

  1. Add the magnitudes.
  2. Check for overflow (a carry out of the MSB)
  3. The sign of the result is the same as the sign of the first operand.

- **If the parity of the three signs is 1:**

  1. Subtract the second magnitude from the first.
  2. If a borrow occurs:
     - take the two's complement of result
     - and make the result sign the complement of the sign of the first operand.
  3. Overflow will never occur.

Sign-Magnitude Arithmetic Examples

- **Example 1:**   **0010**
               +0101

- **Example 2:**   **0010**
               +1101

- **Example 3:**   **1010**
              − 0101

Signed-Complement Arithmetic

■ **Addition:**

1. Add the numbers including the sign bits, discarding a carry out of the sign bits (2's Complement), or using an end-around carry (1's Complement).

2. If the sign bits were the same for both numbers and the sign of the result is different, an overflow has occurred.

3. The sign of the result is computed in step 1.

■ **Subtraction:**

Form the complement of the number you are subtracting and follow the rules for addition.

Signed 2's Complement Examples

- **Example 1: 1101**
                +0011

- **Example 2: 1101**
                −0011

2's Complement Adder/Subtractor

- **Subtraction can be done by addition of the 2's Complement.**
  - **1. Complement each bit (1's Complement.)**
  - **2. Add 1 to the result.**
- **The circuit shown computes A + B and A − B:**
- **For S = 1, subtract, the 2's complement of B is formed by using XORs to form the 1's comp and adding the 1 applied to C_0.**
- **For S = 0, add, B is passed through unchanged**
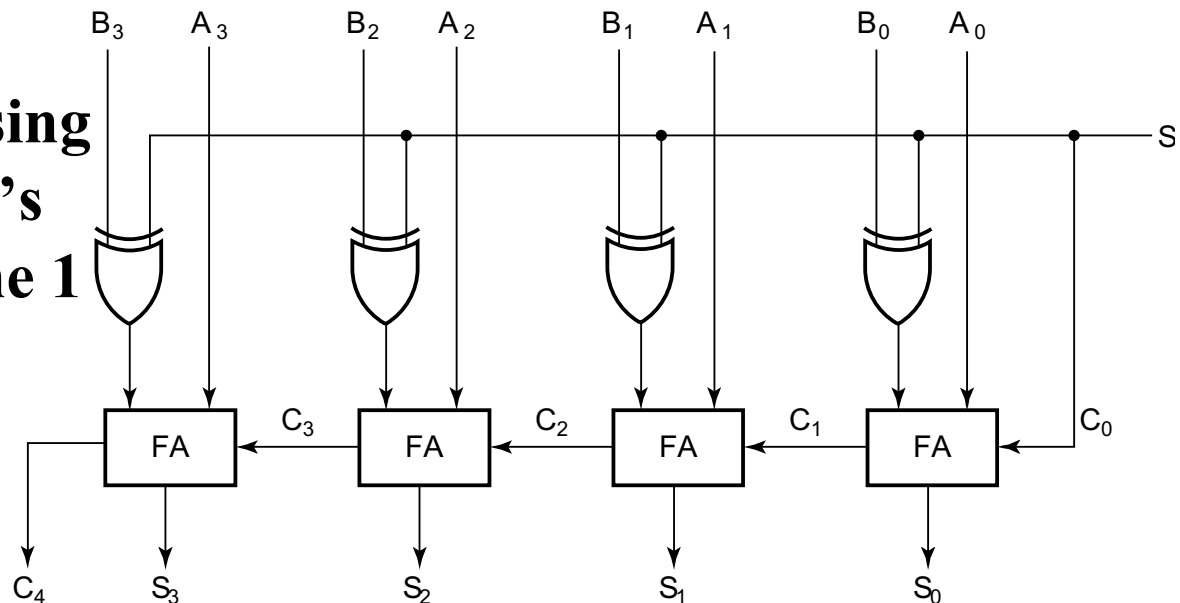
Overflow Detection

- *Overflow* occurs if $n + 1$ bits are required to contain the result from an n-bit addition or subtraction

- Overflow can occur for:
  - Addition of two operands with the same sign
  - Subtraction of operands with different signs

- Signed number overflow cases with correct result sign

$$
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\
0 & 0 & 1 & 1
\end{array}
$$

- Detection can be performed by examining the result signs which should match the signs of the top operand

Overflow Detection

- **An example:**
- **Add two 8-bit registers with +70 and +80, or -70 and -80**

Carries: 01   **Carries:10**

+70 01000110	-70 10111010
+80 01010000	-80 10110000
+150 10010110	-150 01101010

- **C_{n-1} and C_n are different**

Overflow Detection

- **Signed number cases with carries C_n and C_{n-1} shown for correct result signs:**

$$\begin{array}{cccc} 00 & 00 & 11 & 11 \\ 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 0 & 0 & 1 & 1 \end{array}$$

- **Signed number cases with carries shown for erroneous result signs (indicating overflow):**

$$\begin{array}{cccc} 01 & 01 & 10 & 10 \\ 0 & 0 & 1 & 1 \\ +\underline{0} & -\underline{1} & -\underline{0} & +\underline{1} \\ 1 & 1 & 0 & 0 \end{array}$$

- **Simplest way to implement overflow $V = C_n \oplus C_{n-1}$**

- **This works correctly only if 1's complement and the addition of the carry in of 1 is used to implement the complementation! Otherwise fails for $-10\ldots0$**

Arithmetic Logic Unit (ALU)

- **Arithmetic circuit design**
  - **Decompose the arithmetic circuit into:**
    - **An n-bit parallel adder**
    - **A block of logic that selects four choices for the B input to the adder**
    - **See next slide for diagram**

Arithmetic Circuit Design (continued)

- **There are only four functions of B to select as Y in G = A + Y:**

$$C_{in} = 0 \qquad C_{in} = 1 \qquad Y_i = B_i S_0 + \overline{B}_i S_1$$

	$C_{in} = 0$	$C_{in} = 1$
• 0	G = A	G = A + 1
• B	G = A + B	G = A + B + 1
• $\overline{B}$	G = A + $\overline{B}$	G = A + $\overline{B}$ + 1
• 1	G = A – 1	G = A

- **What functions are implemented with carry-in to the adder = 0? =1?**
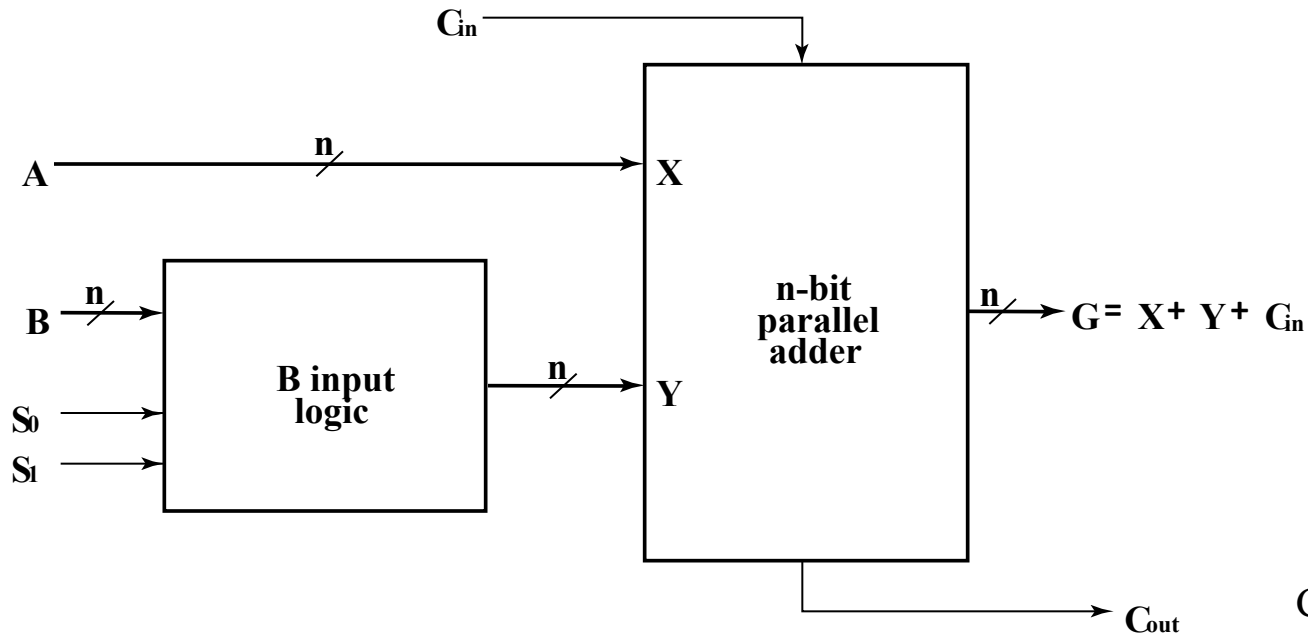
Assignment

- **3-50；3-51；3-52；3-59**

- **Supplement：Assume the binary numbers in Problem 3-51 are all signed binary numbers in Signed-Magnitude form, repeat Problem 3-51.**