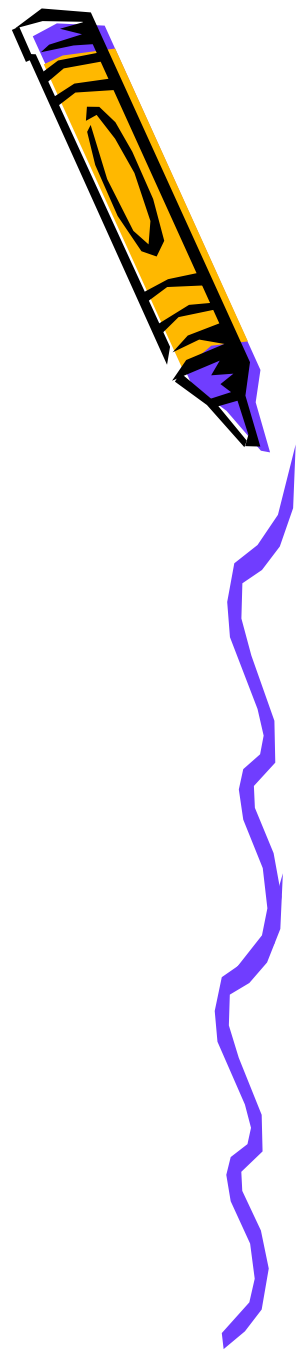




# Computer Architecture

Hossein Asadi  
Department of Computer Engineering  
Sharif University of Technology  
[asadi@sharif.edu](mailto:asadi@sharif.edu)

# Today's Topics



- Adders
  - Ripple carry
  - Carry select adder
  - Carry-look-ahead adder
- Multipliers
  - Combinational multiplier
  - Sequential multiplier
  - Booth multiplier



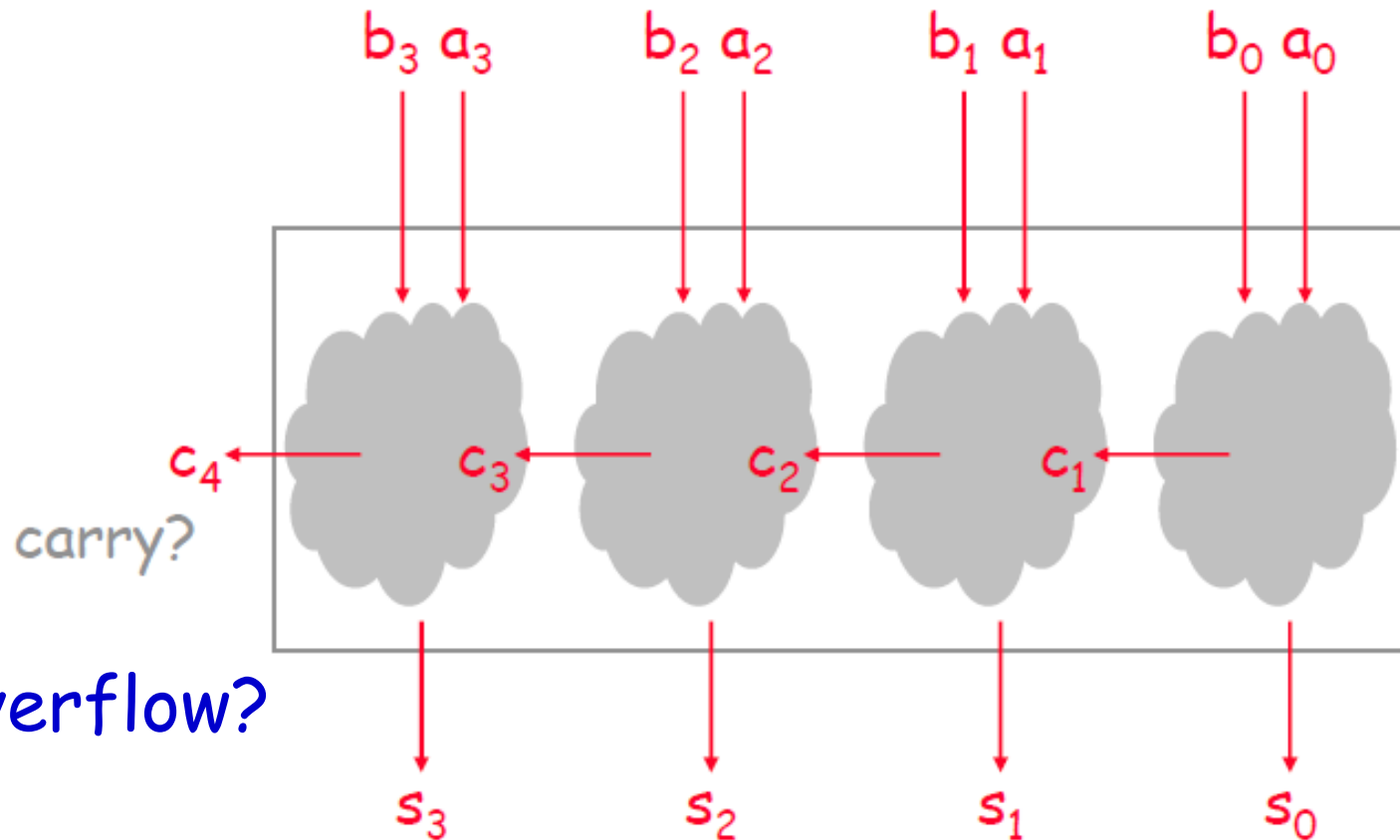
# Copyright Notice



- Parts (text & figures) of this lecture adopted from:
  - Computer Organization & Design, The Hardware/Software Interface, 3<sup>rd</sup> Edition, by D. Patterson and J. Hennessey, MK publishing, 2005.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, CMU, Spring 2009.
  - "Computer Architecture & Engineering" handouts, by Prof. Kubiawicz, UC Berkeley, Spring 2004.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, UWisc, Spring 2019.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC, Spring 2009.
  - "Intro to Computer Organization" handouts, by Prof. Mahlke & Prof. Narayanasamy, Winter 2008.



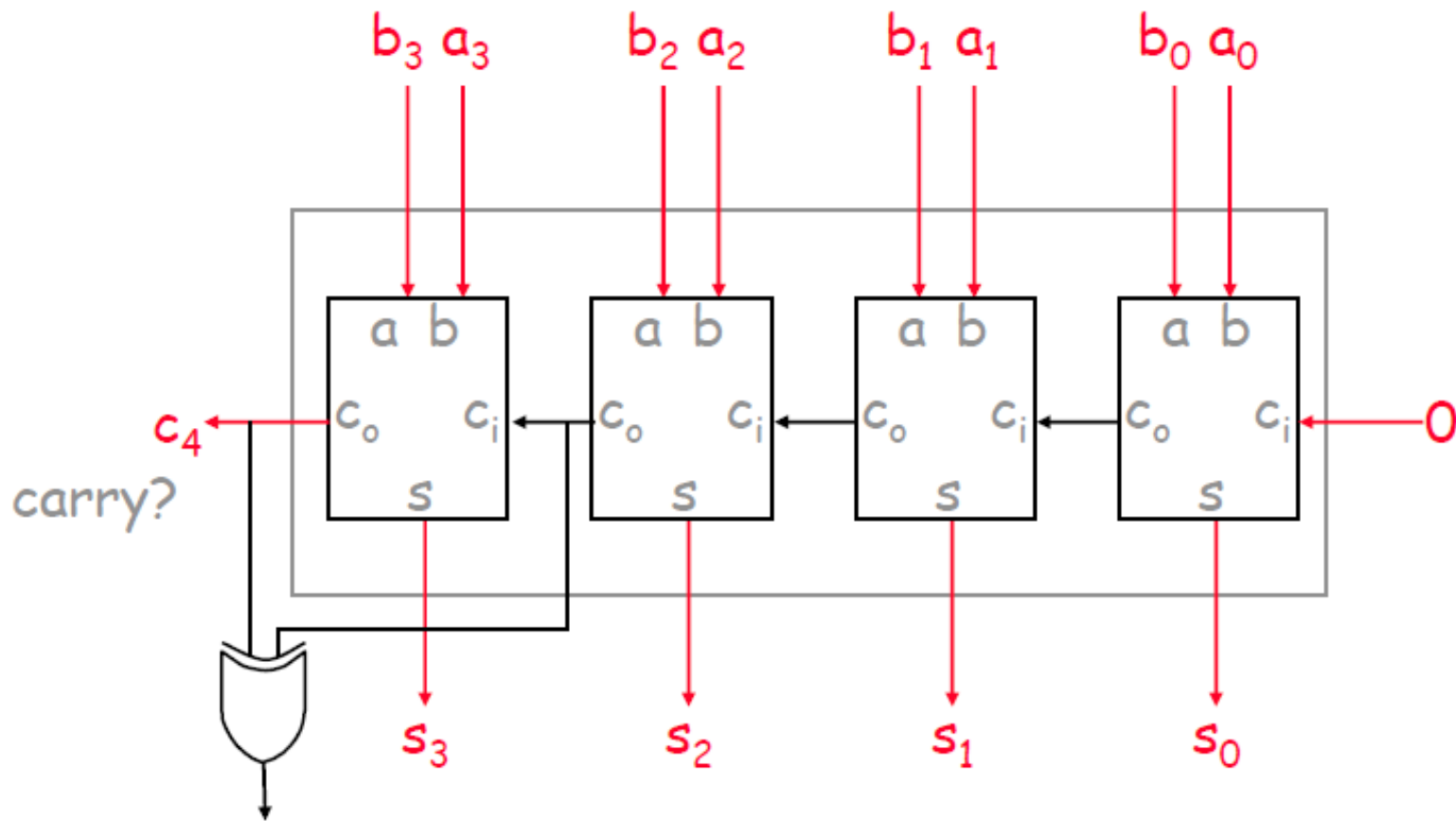
# Unsigned Binary Addition



Overflow?

- Overflow =  $c_4$

# 2's Complement Addition

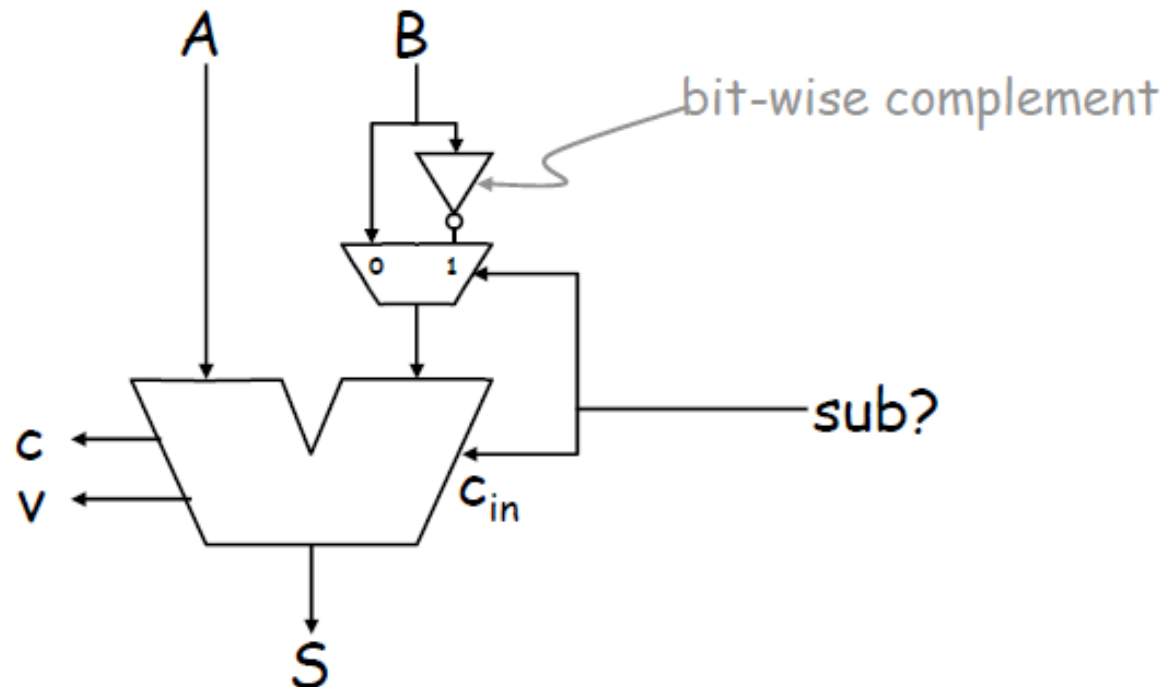


- $\text{Overflow} = (a_3 \text{ xor } b_3) ? 0 : (a_3 \text{ xor } s_3)$

# 2's Complement Subtraction



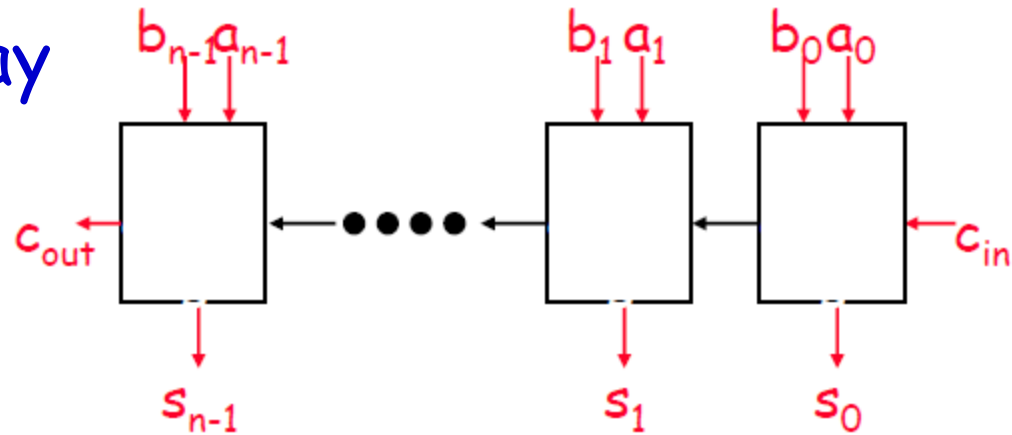
- Subtraction
  - Similar to adding negative number



# Analysis of an "n-bit" Ripple-Carry (RC) Adder



- Size/Complexity
  - $n * \text{SizeOf(Full Adder)}$
  - $O(n)$
- Critical Path Delay
  - $n * \text{DelayOf(Full Adder)}$
  - $n * 2 * \text{gate delay}$
  - $O(n)$



# High-Performance Adders



- Question:
  - Any adder running faster than RC adder?
    - How to reduce carry propagation delay?
- Answer:
  - Compute intermediate carry signal
    - E.g., compute  $C1$ ,  $C2$ , and  $C3$  in parallel

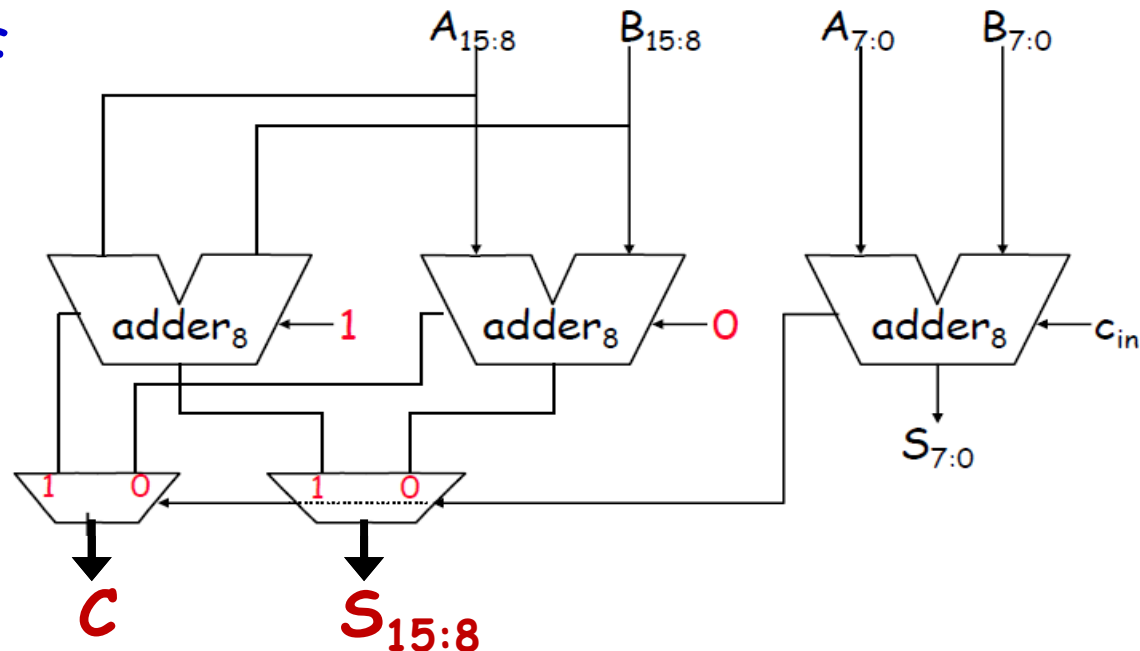




# Carry-Select Adder (CSA)



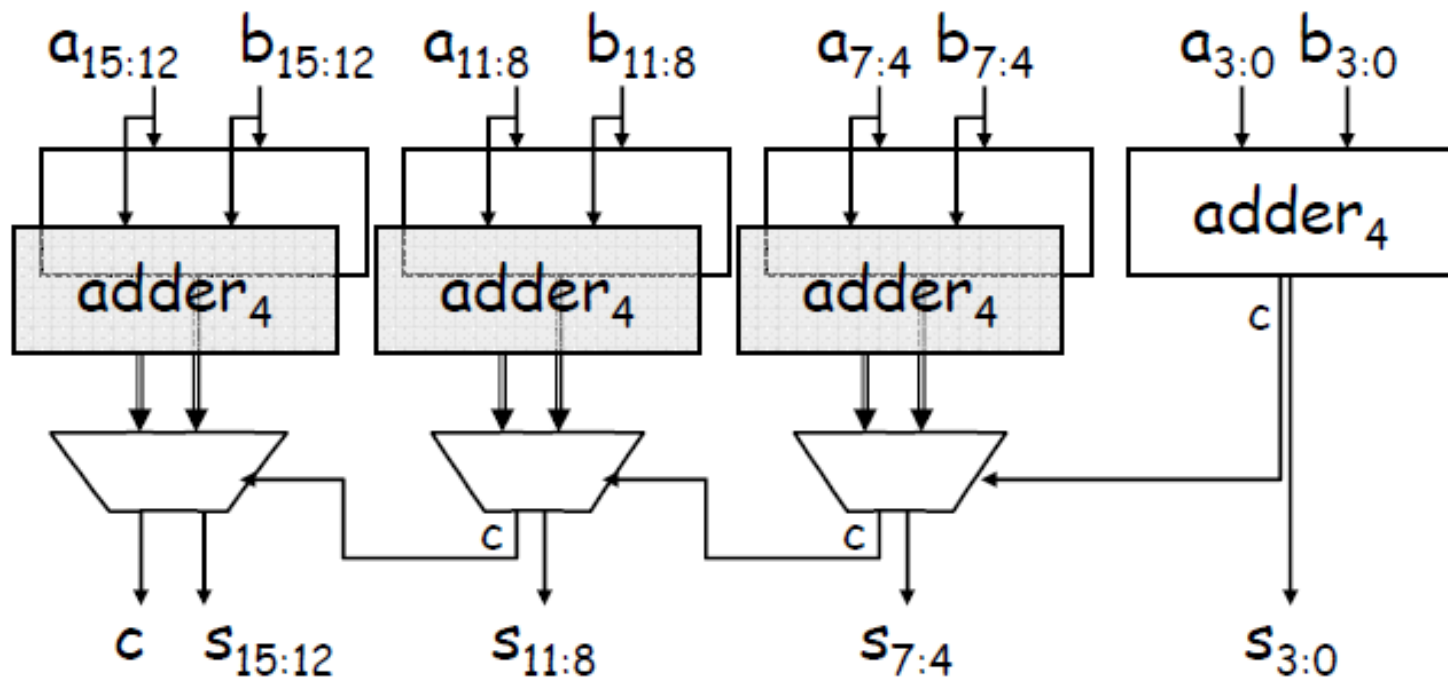
- Delay
  - $8 * D_{FA} + D_{mux}$
- Cost
  - $24 * F$



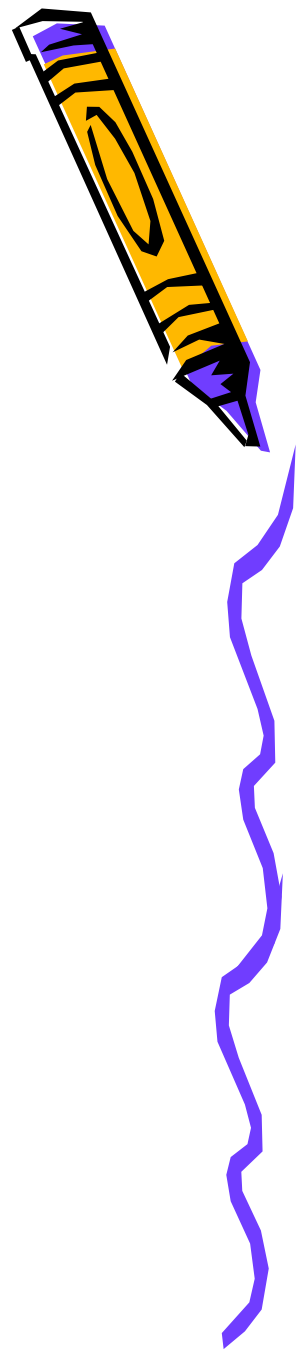
# Multi-Stage CSA



- Delay
  - $4 * D_{FA} + 3 * D_{mux}$
- Cost
  - $28 * FA + 3 * Mux$



# Multi-Stage CSA (cont.)



- K-Stage n-bit CSA
- Delay
  - $(n/k) * D_{FA} + (k-1) * D_{mux}$
- Cost
  - $N * \{(2k-1)/k\} * FA + (k-1) * Mux$



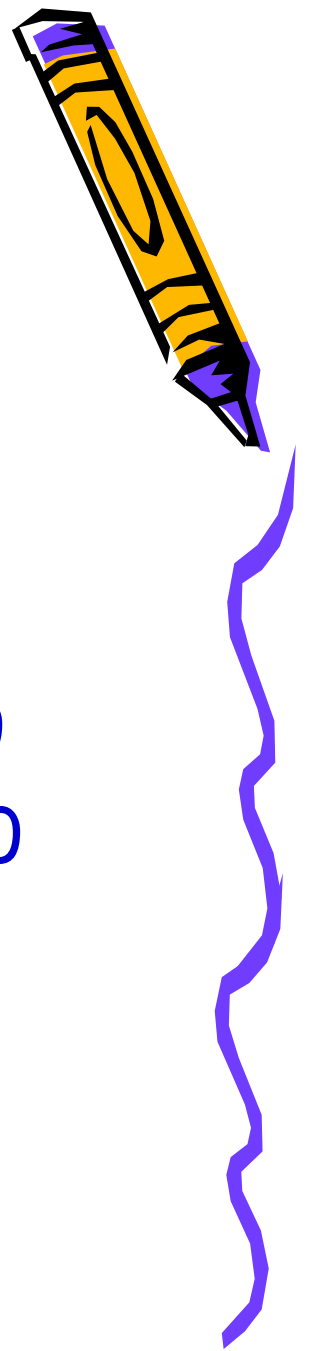
# Carry Generate & Propagate



- If  $a.b = 1 \Rightarrow C_{out} = 1$  regardless of  $C_{in}$ 
  - Carry generate
- If  $a \text{ xor } b = 1 \Rightarrow C_{out} = C_{in}$ 
  - Carry propagate
- We define
  - $G_i = a_i.b_i$  (Generate)
  - $P_i = a_i \text{ xor } b_i$  (Propagate)
  - $\Rightarrow C_{i+1} = G_i + P_i.C_i$

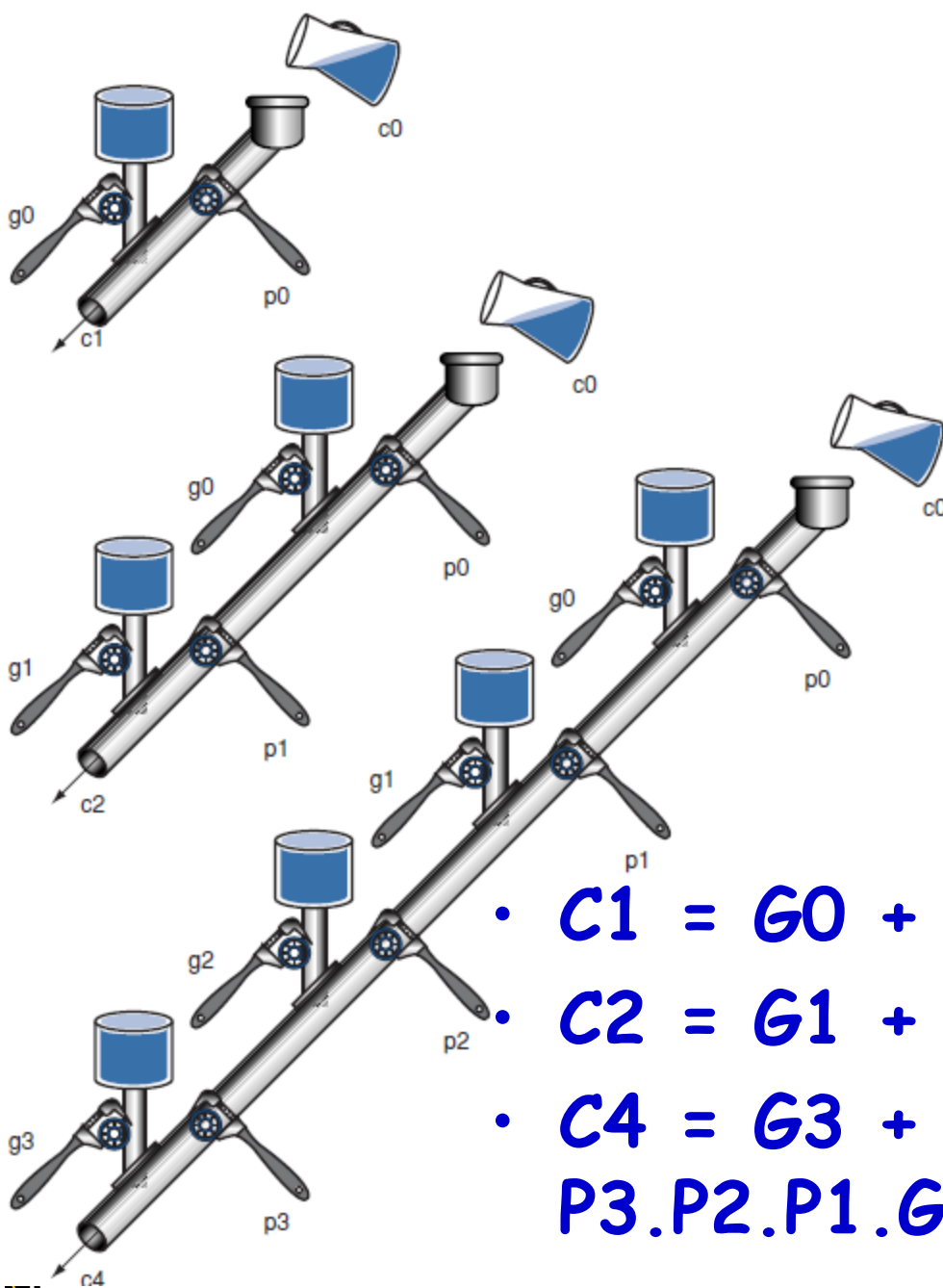


# Carry Look-Ahead Adder

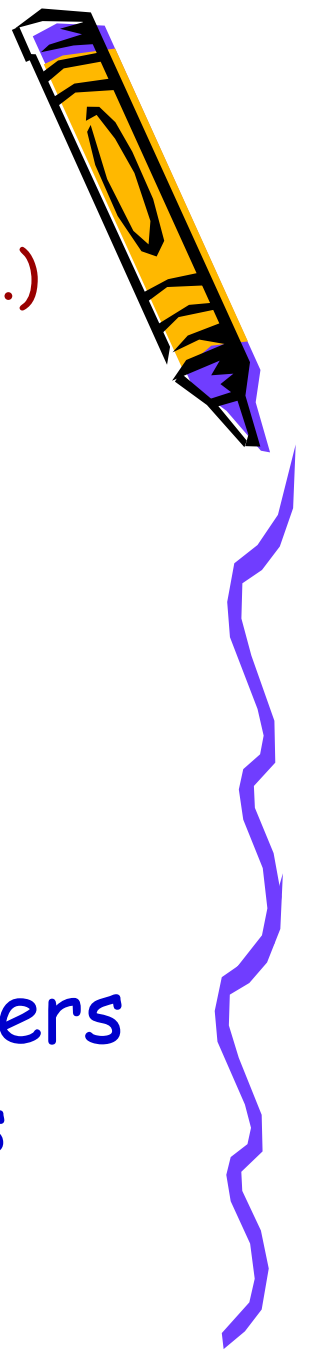


- $C1 = G0 + P0.C0$
- $C2 = G1 + P1.C1 = G1 + P1.(G0 + P0.C0)$   
 $= G1 + P1.G0 + P1.P0.C0$
- $C3 = G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0$
- $C4 = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0$   
 $+ P3.P2.P1.P0.C0$





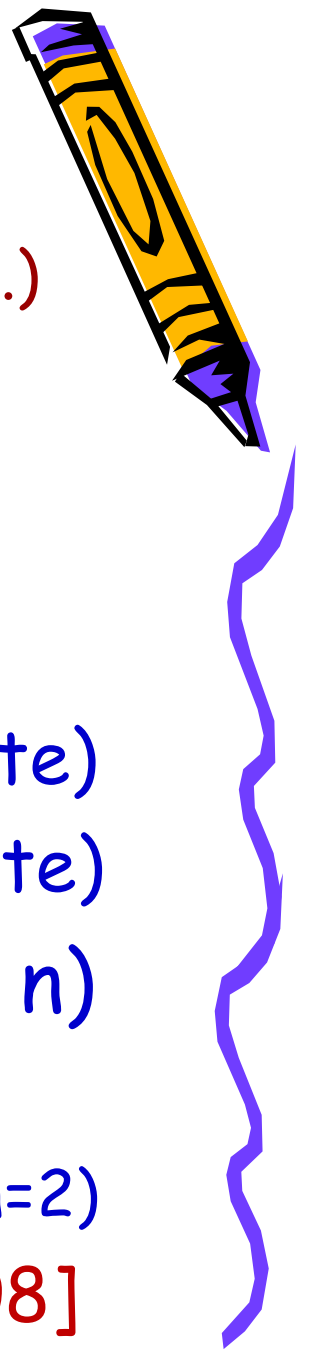
- $C_1 = G_0 + P_0.C_0$
- $C_2 = G_1 + P_1.G_0 + P_1.P_0.C_0$
- $C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.C_0$



# Carry Look-Ahead Adder (cont.)

- Delay Complexity
  - $O(\log n)$
- Size Complexity
  - $O(n^2)$
- Manageable for Small  $n$ 's
- Can be used in **Two-Level** CLA Adders
  - 16-bit adder using 4-bit CLA modules



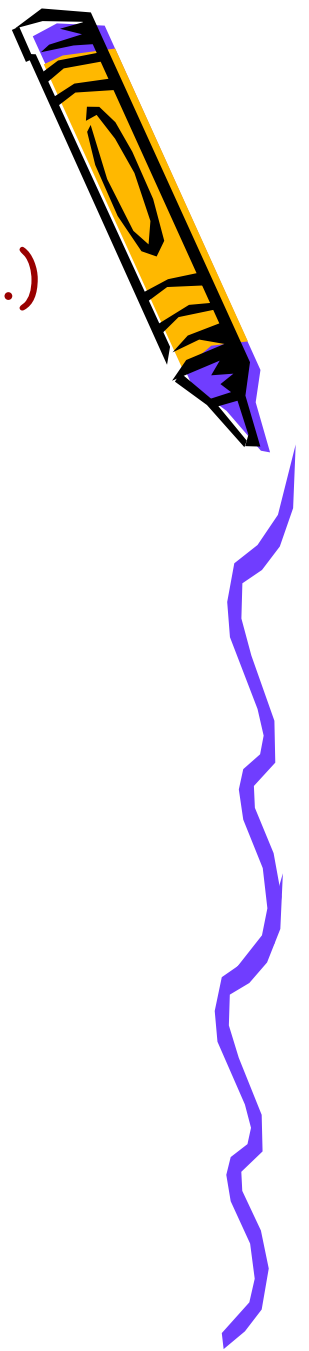


# Carry Look-Ahead Adder (cont.)

- $D_g$ : Delay of a Single Gate
- At each Stage
  - $D_g$  for generating all  $P_i$  and  $G_i$
  - $2 \cdot D_g$  for generating all  $C_i$  (2-level gate)
  - $2 \cdot D_g$  for generating all  $S_i$  (2-level gate)
- Total Delay:  $5 \cdot D_g$  (independent of  $n$ )
  - Issue?
    - $D(\text{gate with } f_{\text{ainin}}=32) \gg D(\text{gate w } f_{\text{ainin}}=2)$
  - [Copyright I. Kormen, umass, spring'08]







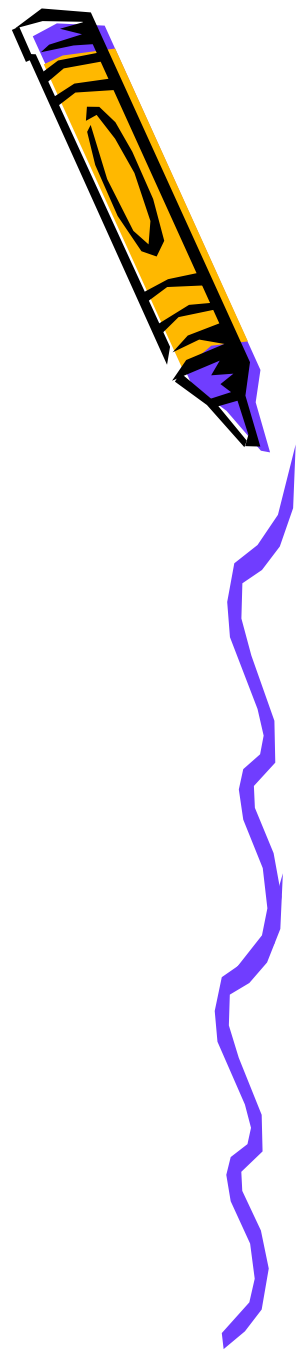
# Carry Look-Ahead Adder (cont.)

- n Stages Divided into Groups
  - Separate CLA in each group
  - Group interconnected by RC
- Example: Group Size = 4
  - $D_g$  for generating all  $P_i$  and  $G_i$
  - $2 \cdot D_g$  to propagate carry through a group
  - $(n/4) \cdot 2 \cdot D_g$  to propagate carry using RC
  - $2 \cdot D_g$  to generate  $S_i$
  - Total:  $[2(n/4) + 3]D_g = [n/2 + 3] \cdot D_g \rightarrow$
  - 75% reduction compared to full RC

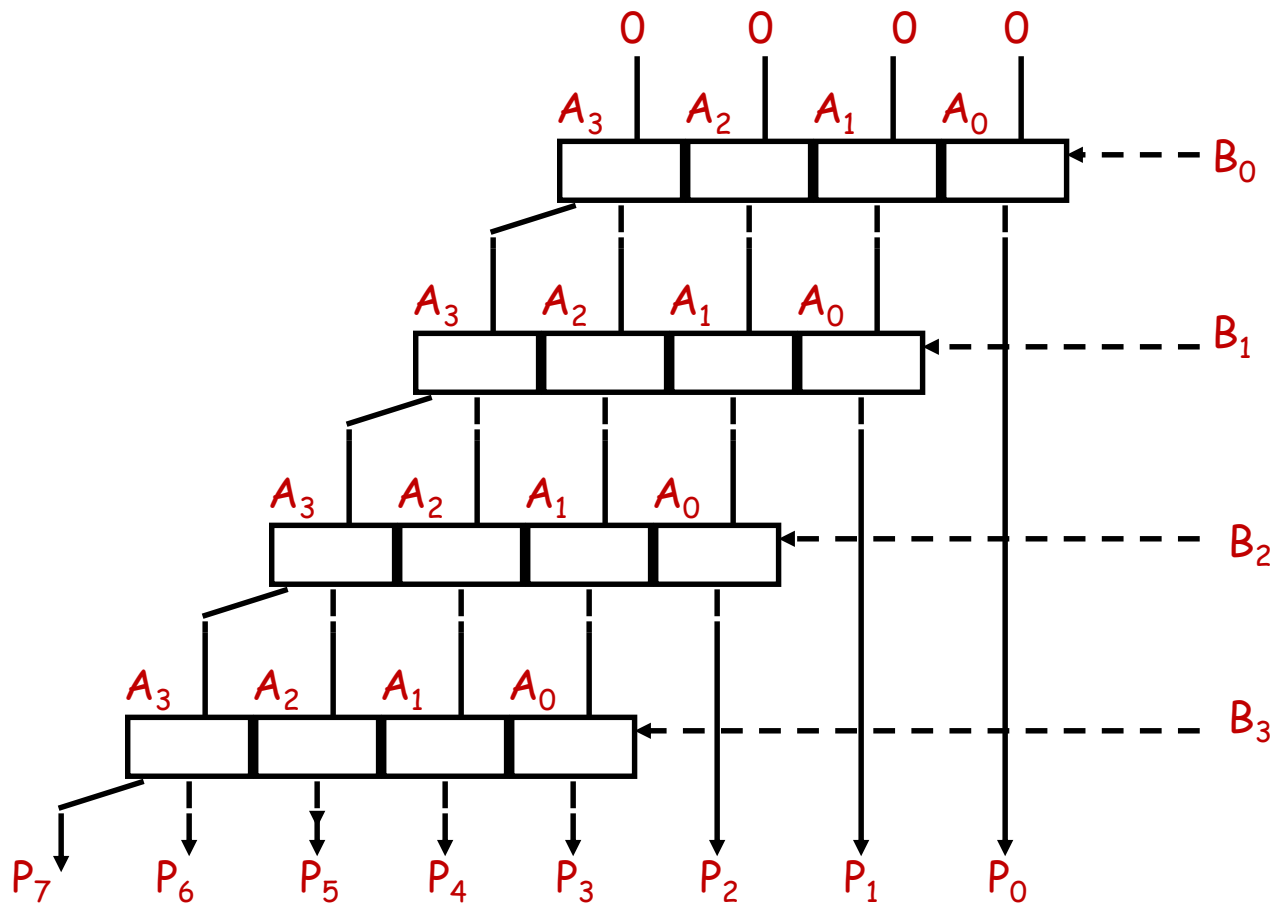


# Multiplier

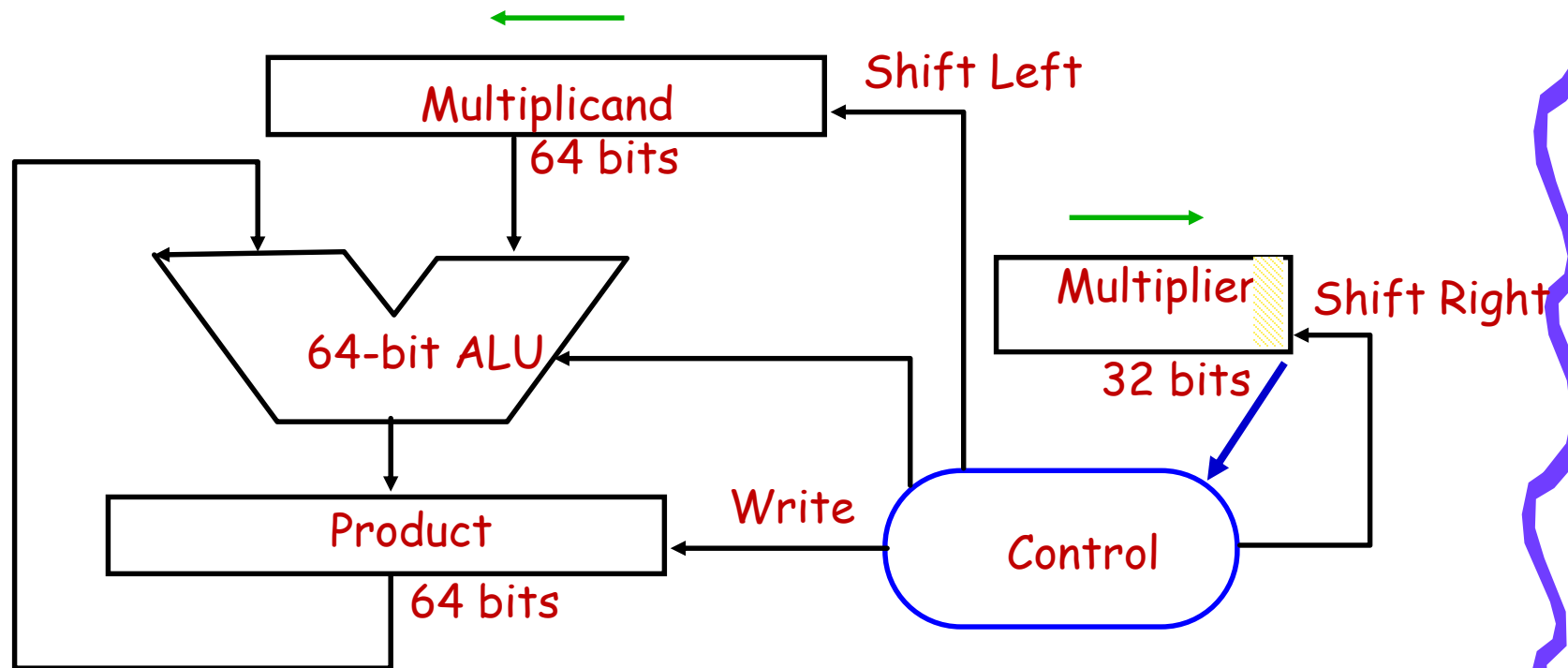
- Combinational Multiplier
  - Also called array multiplier
- Shift-Add Multiplier
- Booth Multiplier



# Combinational Multiplier



# Shift-Add Multiplier



# Booth's Algorithm



- Example  $2 \times 6 = 0010 \times 0110$ :

	0010	
x	0110	
<hr/>		
+	0000	shift (0 in multiplier)
+	0010	add (1 in multiplier)
+	0010	add (1 in multiplier)
+	0000	shift (0 in multiplier)
<hr/>		
	00001100	

- ALU with add or subtract gets same result in more than one way:

$$6 = -2 + 8$$

$$0110 = -00010 + 01000 = 11110 + 01000$$

- For example

	0010	
x	0110	
<hr/>		
	0000	shift (0 in multiplier)
-	0010	sub (first 1 in multiply)
	0000	shift (mid string of 1s)
+	0010	add (prior step had last 1)
<hr/>		
	00001100	



# Booth's Algorithm (cont.)

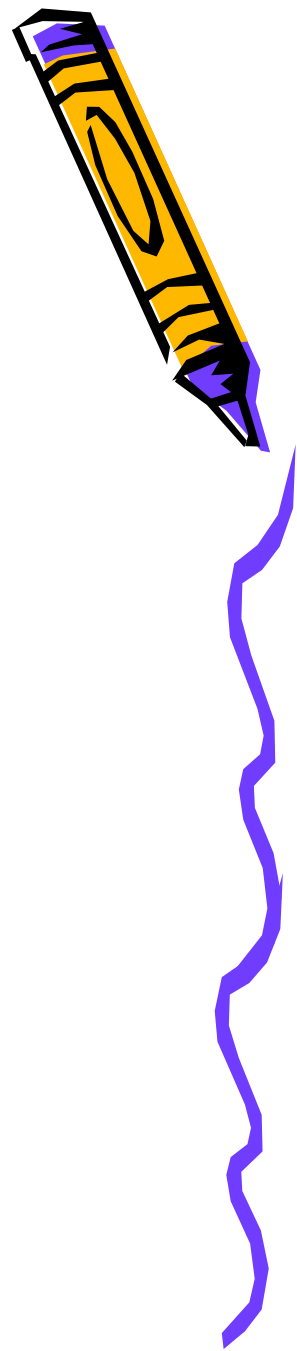


Current Bit	Bit to Right	Explanation	Example	Op
1	0	Begins run of 1s	000111 <u>1</u> 000	sub
1	1	Middle of run of 1s	00011 <u>11</u> 000	none
0	1	End of run of 1s	00 <u>01</u> 111000	add
0	0	Middle of run of 0s	0 <u>00</u> 1111000	none

- Originally developed for Speed
  - When shift was faster than add



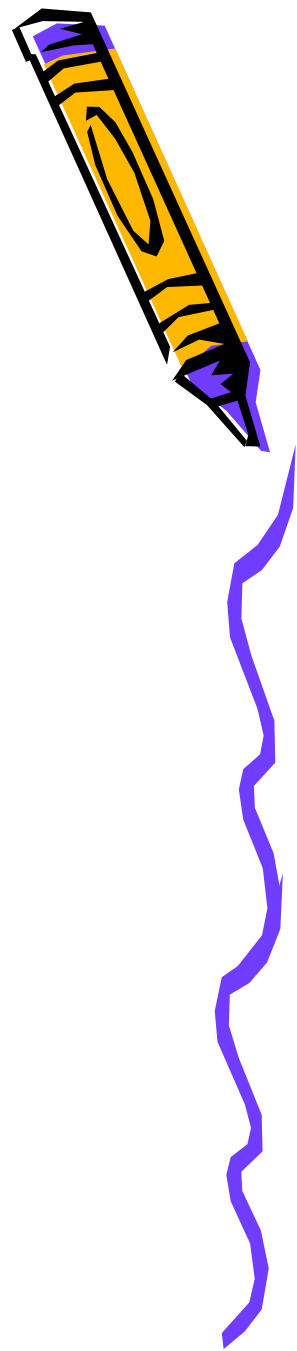
# Booth Encoding: Example



1	0	0	1	1	1	1	0	1	0
-1	0	1	0	0	0	-1	1	-1	0



# Booth's Algorithm: Example



- Example
  - multiply **(-5) x 2**
- Let's use 5-bit 2's complement:
  - A: -5 is 11011 (multiplier)
  - B: 2 is 00010 (multiplicand)





# Beginning Product



- Multiplier is:

11011

- Add 5 leading zeros to **multiplier** to get **beginning product**:

00000 11011



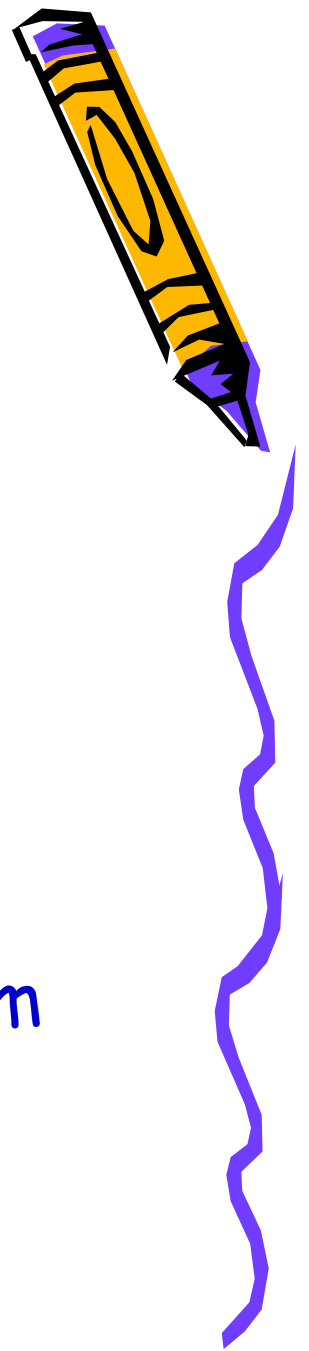
# Step 1 for each pass



- Use **LSB** (least significant bit) and **previous LSB** to determine arithmetic action
  - If it is **FIRST** pass, use **0** as previous LSB
- Possible arithmetic actions:
  - **00** → no arithmetic operation
  - **01** → add multiplicand to left half of product
  - **10** → subtract multiplicand from left half of product
  - **11** → no arithmetic operation



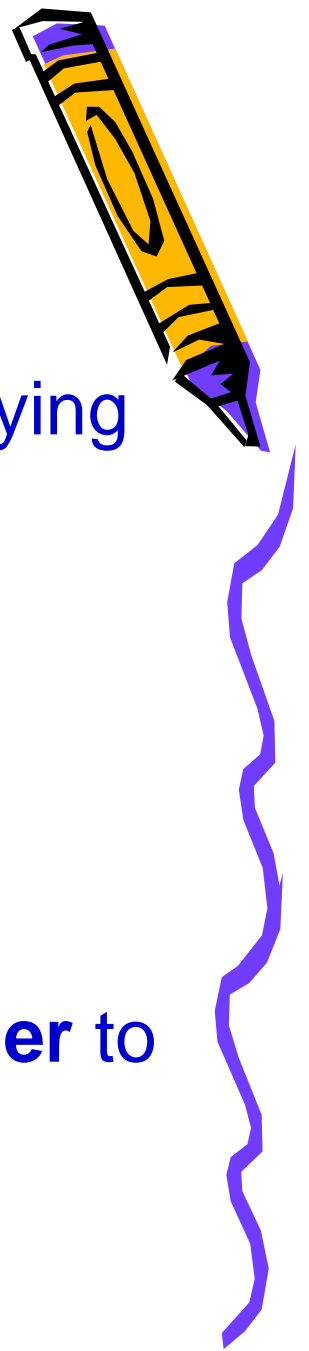
# Step 2 for Each Pass



- Perform an **arithmetic right shift** (ASR) on entire product
- Note:
  - For X-bit operands, Booth's algorithm requires X passes



# Example



- Let's continue with our example of multiplying  **$(-5) \times 2$**
- Remember:
  - $-5$  is 11011 (multiplier)
  - $2$  is 00010 (multiplicand)
- And we added 5 leading zeros to **multiplier** to get **beginning product**:

00000 11011



# Example



- Initial Product and previous LSB

00000 11011 0

(Note: Since this is first pass, we use 0 for previous LSB)

- Pass 1, Step 1: Examine last 2 bits

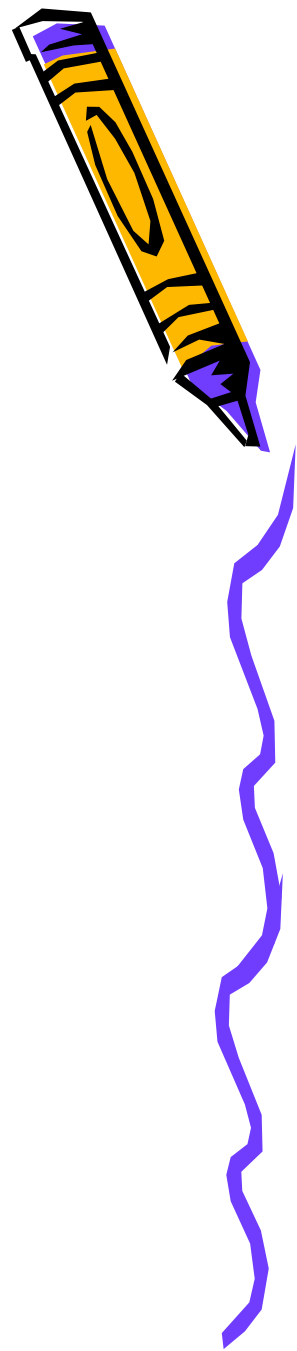
00000 1101**1** 0

Last two bits are **10**, so we need to:

subtract **multiplicand** from left half of product



# Example: Pass 1 (cont.)



- Pass 1, Step 1: Arithmetic action

$$\begin{array}{rcl} (1) & 00000 & \text{(left half of product)} \\ & -00010 & \text{(multiplicand)} \\ \hline & 11110 & \text{(uses a phantom borrow)} \end{array}$$

- Place result into **left half** of product

11110 11011 0



# Example: Pass 1 (cont.)



- Pass 1, Step 2: ASR (arithmetic shift right)

– Before ASR

11110 11011 0

– After ASR

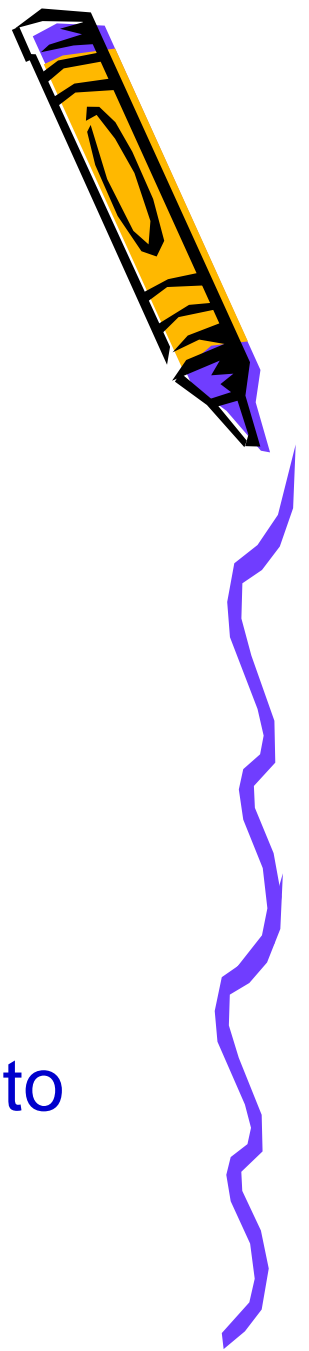
11111 01101 1

(left-most bit was 1, so a 1 was shifted in on the left)

- Pass 1 is complete



# Example: Pass 2



- Current Product and previous LSB

11111 01101 1

- Pass 2, Step 1: Examine last 2 bits

11111 01101 1

Last two bits are 11, so we do NOT need to perform an arithmetic action --

just proceed to step 2.





## Example: Pass 2 (cont.)



- Pass 2, Step 2: ASR (arithmetic shift right)
  - Before ASR

11111 01101 1

- After ASR

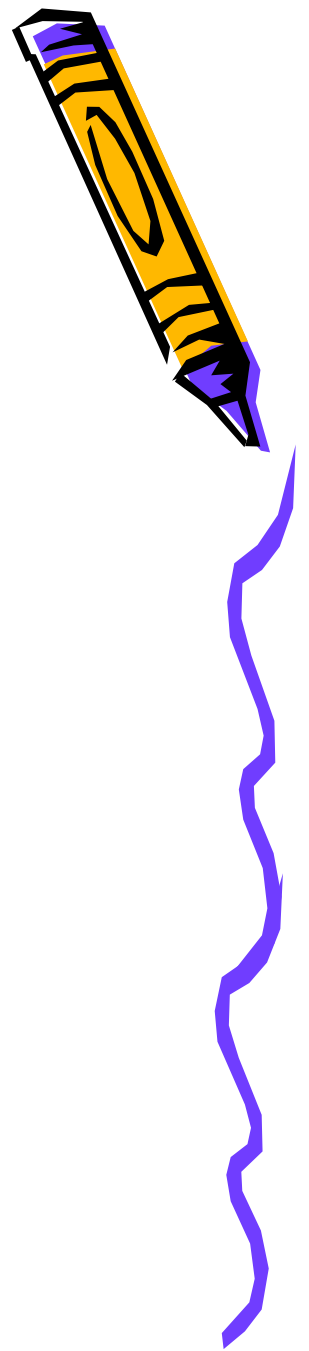
11111 10110 1

(left-most bit was 1, so a 1 was shifted in on left)

- Pass 2 is complete



# Example: Pass 3



- Current Product and previous LSB

11111 10110 1

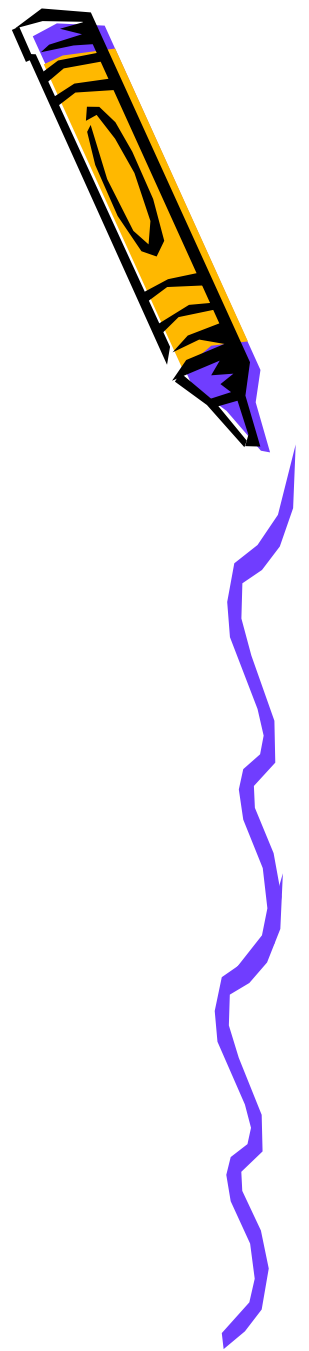
- Pass 3, Step 1: Examine last 2 bits

11111 10110 1

Last two bits are 01, so we need to:  
add **multiplicand** to left half of product



## Example: Pass 3 (cont.)



- Pass 3, Step 1: Arithmetic action

$$\begin{array}{rcl} \textcolor{red}{(1)} & 11111 & \text{(left half of product)} \\ + & 00010 & \text{(multiplicand)} \\ \hline & 00001 & \text{(drop the leftmost carry)} \end{array}$$

- Place result into **left half** of product

00001 10110 1



## Example: Pass 3 (cont.)



- Pass 3, Step 2: ASR (arithmetic shift right)

– Before ASR

00001 10110 1

– After ASR

00000 11011 0

(left-most bit was 0, so a 0 was shifted in on left)

- Pass 3 is complete



# Example: Pass 4



- Current Product and previous LSB

00000 11011 0

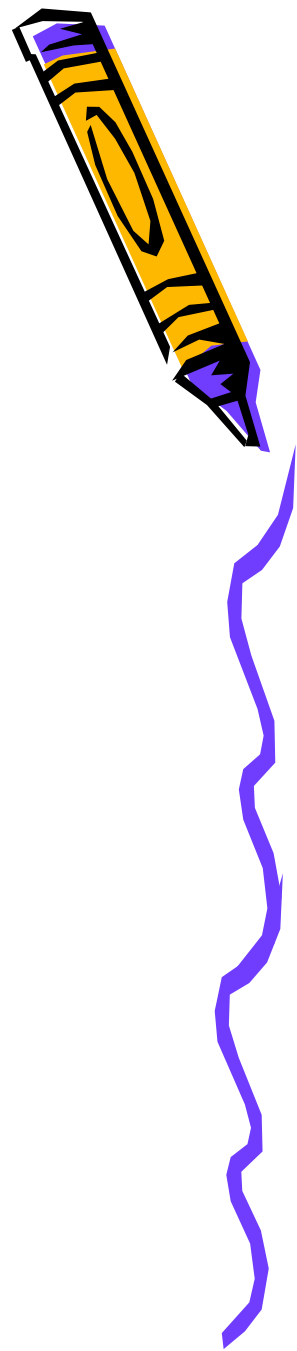
- Pass 4, Step 1: Examine last 2 bits

00000 1101**1** 0

Last two bits are **10**, so we need to:  
subtract **multiplicand** from left half of product



# Example: Pass 4 (cont.)



- Pass 4, Step 1: Arithmetic action

$$\begin{array}{rcl} (1) & 00000 & \text{(left half of product)} \\ & -00010 & \text{(multiplicand)} \\ \hline & 11110 & \text{(uses a phantom borrow)} \end{array}$$

- Place result into **left half** of product

11110 11011 0



## Example: Pass 4 (cont.)



- Pass 4, Step 2: ASR (arithmetic shift right)

– Before ASR

11110 11011 0

– After ASR

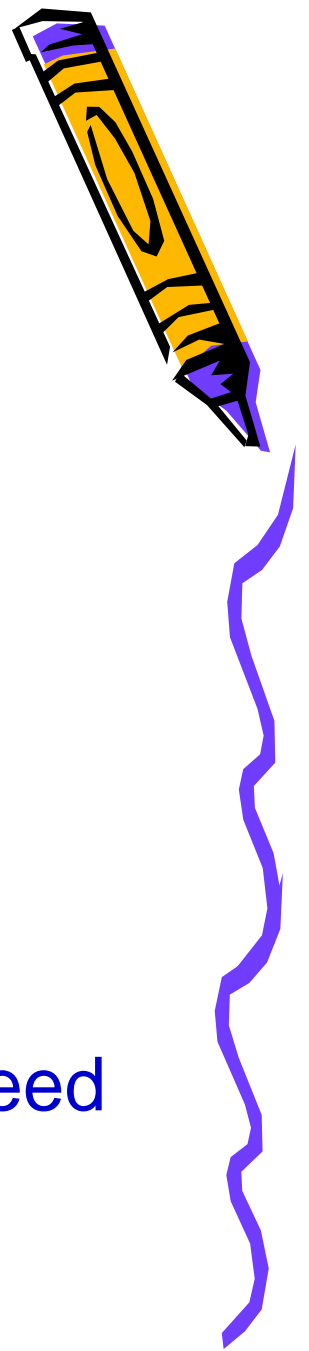
11111 01101 1

(left-most bit was 1, so a 1 was shifted in on left)

- Pass 4 is complete



# Example: Pass 5



- Current Product and previous LSB

11111 01101 1

- Pass 5, Step 1: Examine last 2 bits

11111 0110**1 1**

The last two bits are **11**, so we do NOT need to perform an arithmetic action --

just proceed to step 2.





## Example: Pass 5 (cont.)



- Pass 5, Step 2: ASR (arithmetic shift right)

– Before ASR

11111 01101 1

– After ASR

11111 10110 1

(left-most bit was 1, so a 1 was shifted in on left)

- Pass 5 is complete



# Final Product



- We have completed 5 passes on 5-bit operands, so we are done.
- Dropping the previous LSB, resulting final product is:

11111 10110



# Verification



- To confirm we have correct answer, convert the 2's complement final product back to decimal
- Final product: 11111 10110
- Decimal value: -10

which is CORRECT product of:

$$(-5) \times 2$$



# Backup

