Computer Architecture:
Basic Arithmetic

Hossein Asadi (asadi@sharif.edu)

Department of Computer Engineering

Sharif University of Technology

Spring 2024

Copyright Notice

- Some Parts (text & figures) of this Lecture adopted from following:
  - Computer Organization & Design, The Hardware/Software Interface, 3rd 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. Kubiatowicz, UC Berkeley, Spring 2004.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, UWisc, Spring 2021.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC, Spring 2009.
  - "Intro to Computer Organization" handouts, by Prof. Mahlke & Prof. Narayanasamy, Winter 2008.
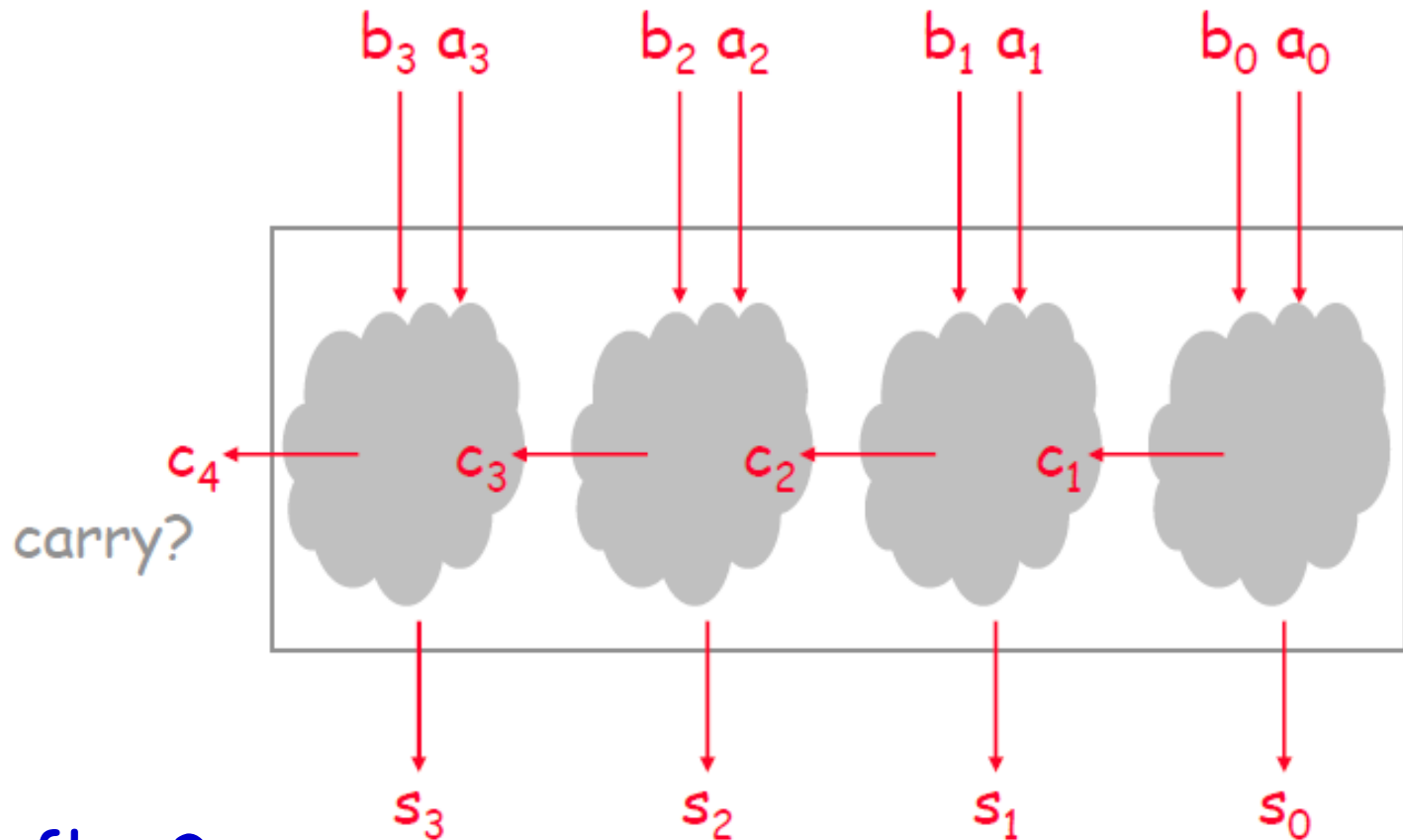
Our Lectur Today

Topics Covered Today

- **Adders**
  - **Ripple carry**
  - **Carry select adder**
  - **Carry-look-ahead adder**
- **Multipliers**
  - **Combinational multiplier**
  - **Sequential multiplier**
  - **Booth multiplier**
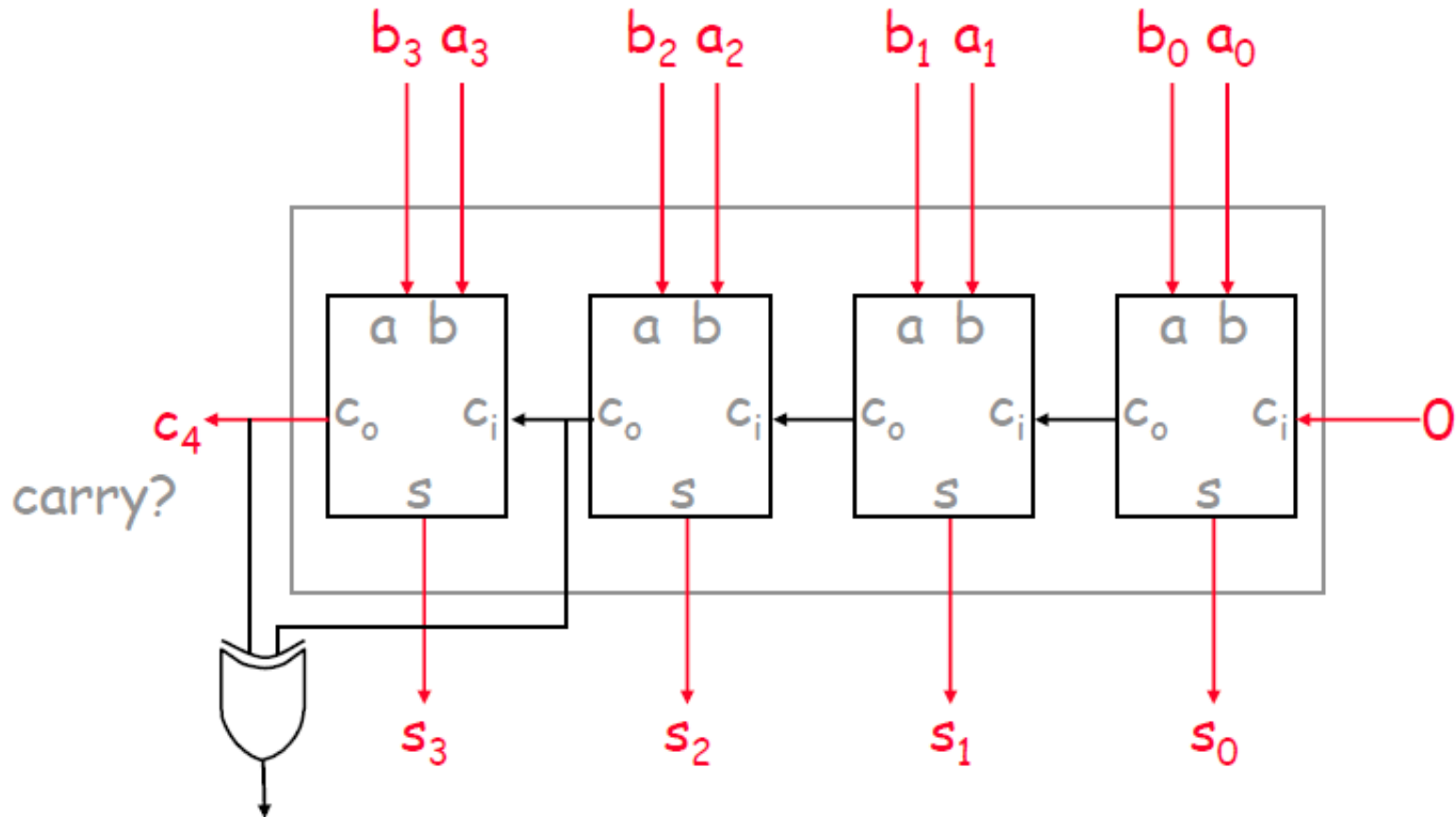
Unsigned Binary Addition

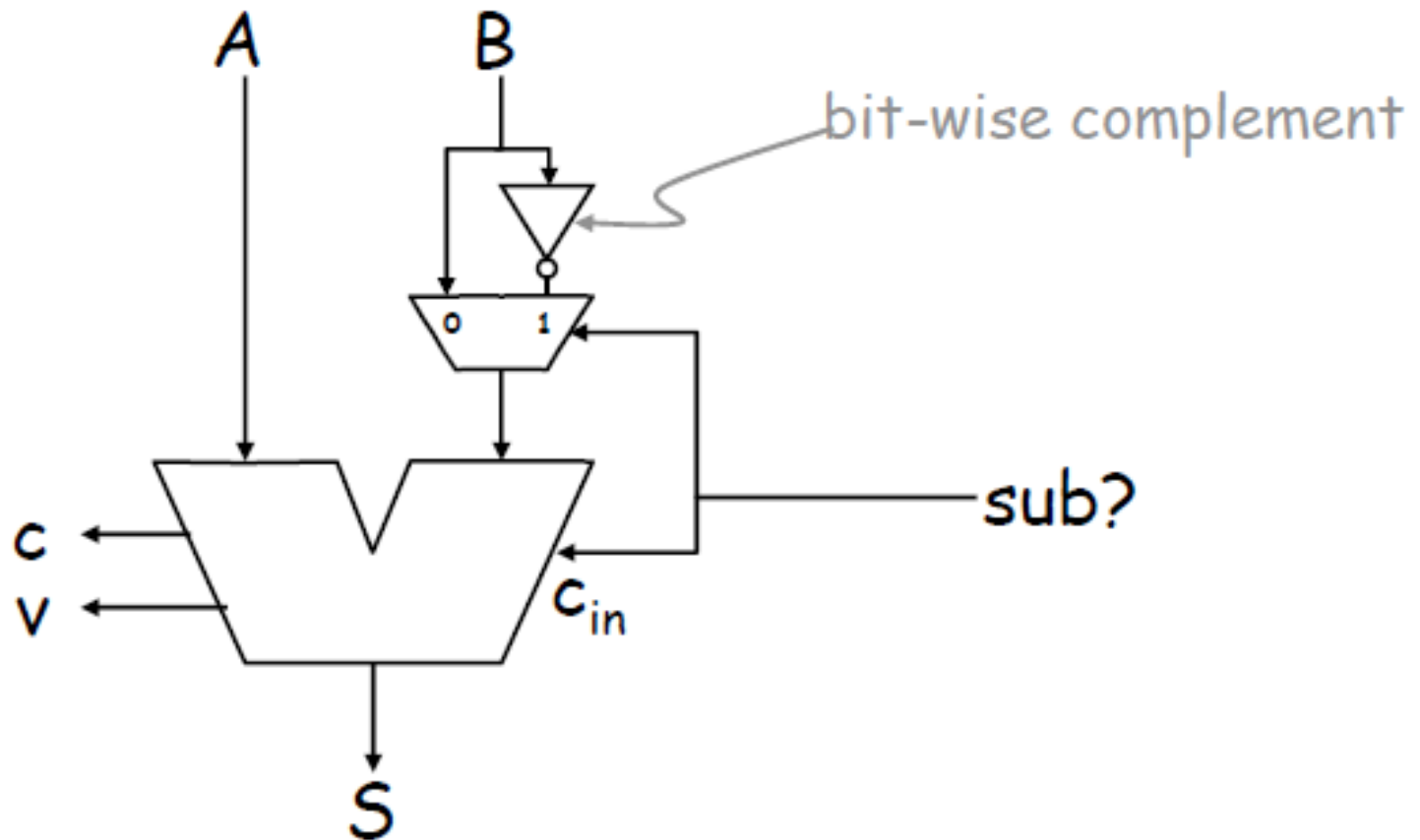
Overflow?

- Overflow = c4

2's Complement Addition

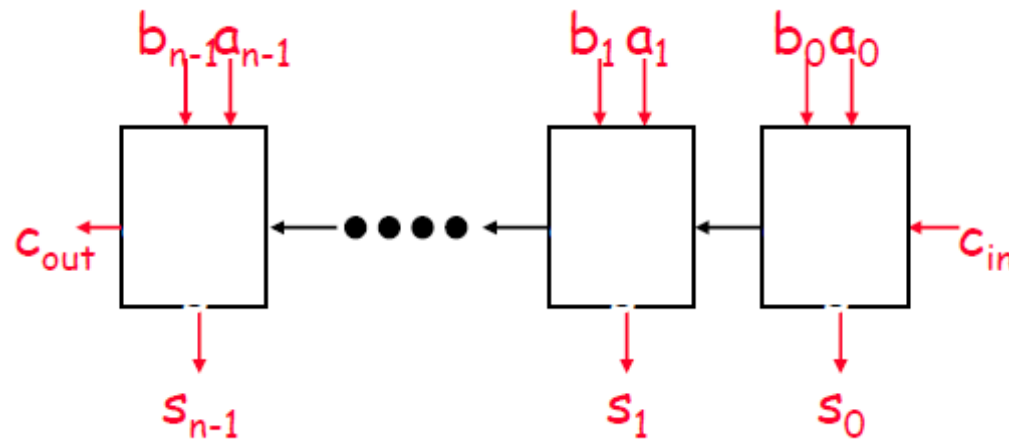- Overflow = (a3 xor b3) ? 0 : (a3 xor s3)

2's Complement Subtraction

- Subtraction
  - Similar to adding negative number

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

- ## Size/Complexity
  - n * SizeOf(Full Adder)
  - O(n)

- ## Critical Path Delay
  - n * DelayOf(Full Adder)
  - n * 2 * gate delay
  - O(n)
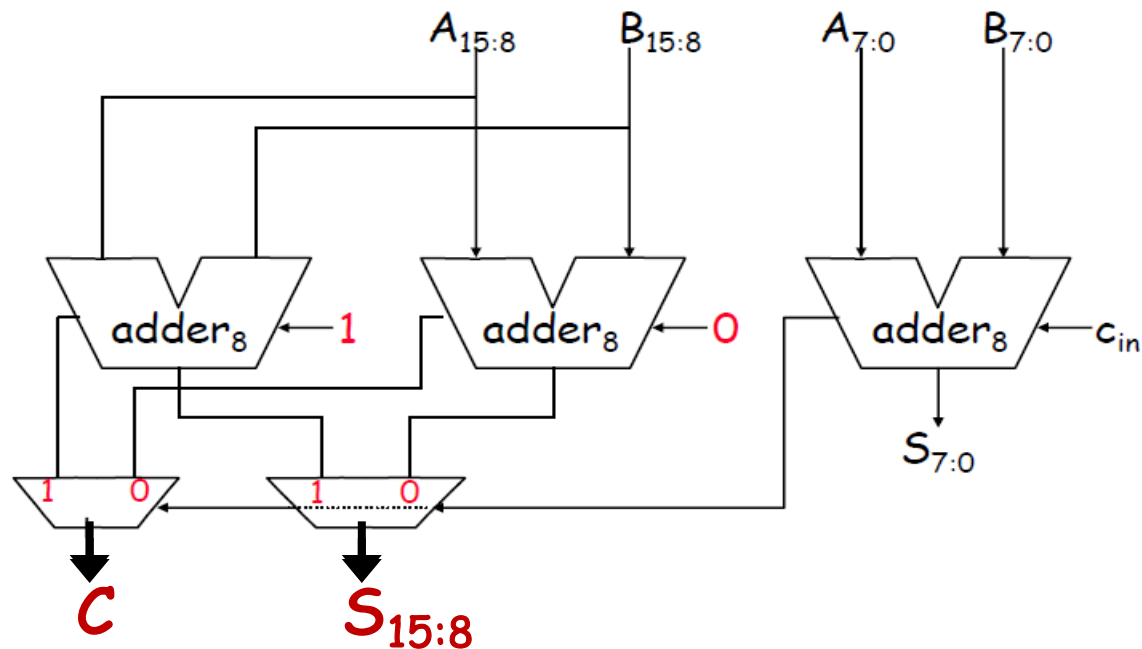
High-Performance Adders

- 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
  - $24*FA + Mux$

Multi-Stage CSA

- Delay
  - $4*D_{FA} + 3*D_{mux}$
- Cost
  - $28*FA + 3*Mux$
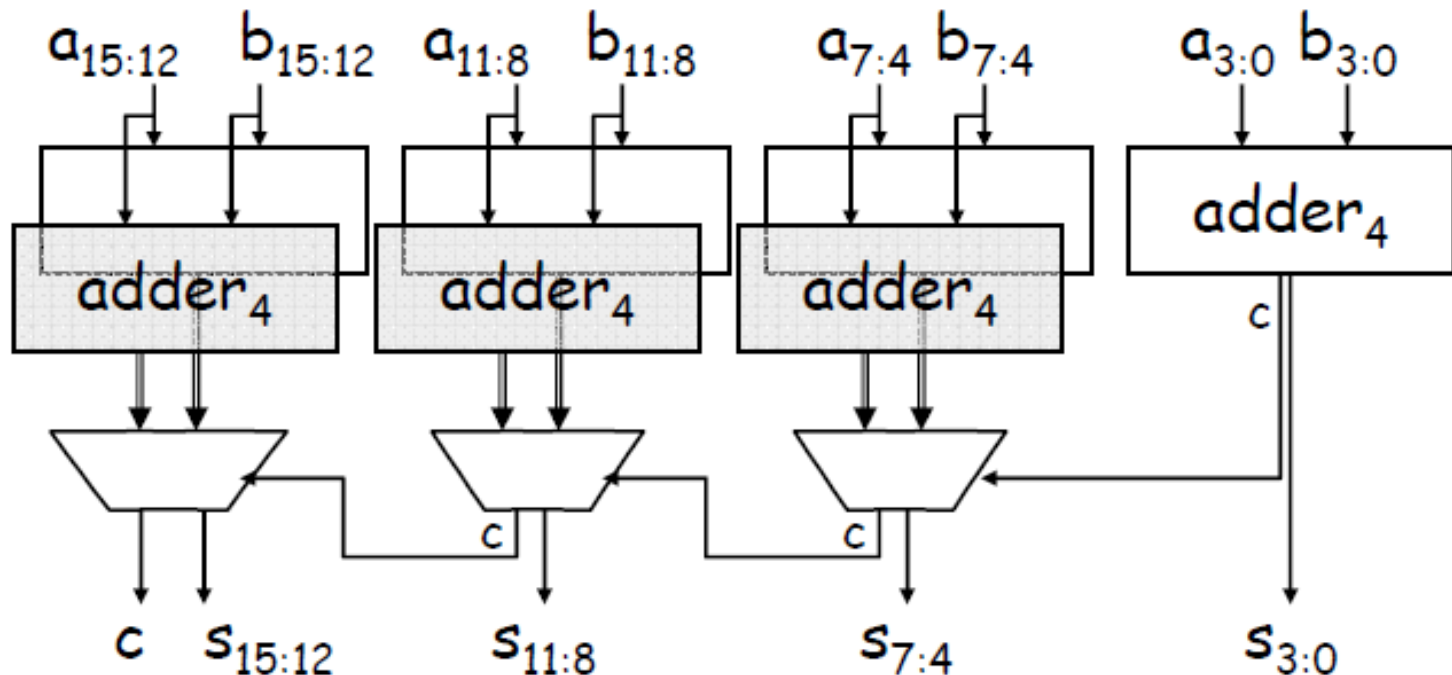
Multi-Stage CSA (cont.)

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

Carry Look-Ahead Adder

- $C1 = G0 + P0.C0$

- $C2 = G1 + P1.C1 = G1 + P1.(G0+P0.C0)$

  $= G1 + P1.G0 + P1.P0.C0$

- $C3 = G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0$

- $C4 = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0$
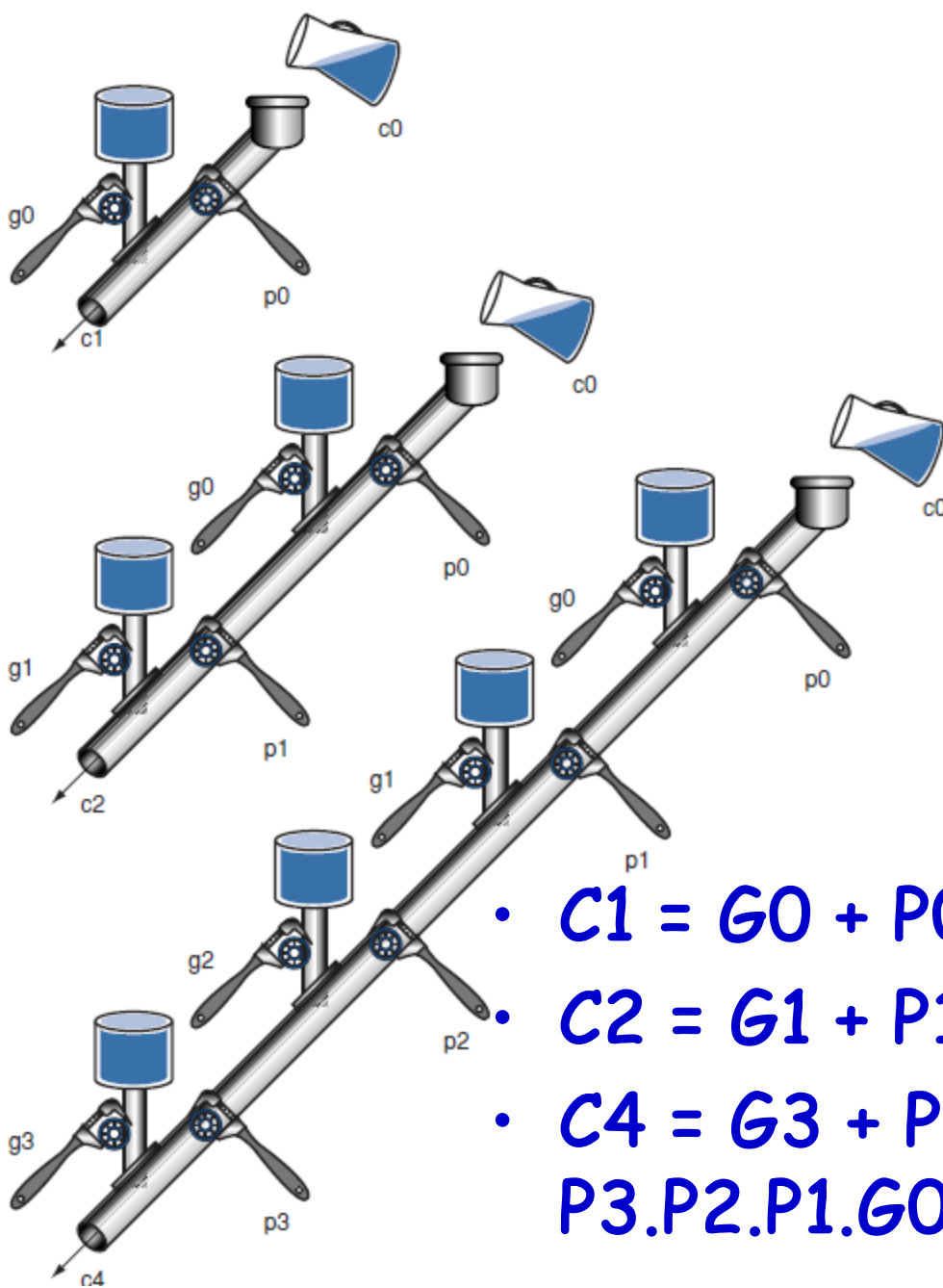
  $+ P3.P2.P1.P0.C0$

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

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.)

- Dg: Delay of a Single Gate

- At each Stage
  - Dg for generating all Pi and Gi
  - 2*Dg for generating all Ci (2-level gate)
  - 2*Dg for generating all Si (2-level gate)

- Total Delay: 5*Dg (independent of n)
  - Issue?
    - D(gate with fainin=32) >> D(gate w fainin=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
  - $2*D_g$ to propagate carry through a group
  - $(n/4)*2*D_g$ to propagate carry using RC
  - $2*D_g$ to generate S_i
  - Total: $[2(n/4)+3]D_g = [n/2+3]*D_g$ ➔
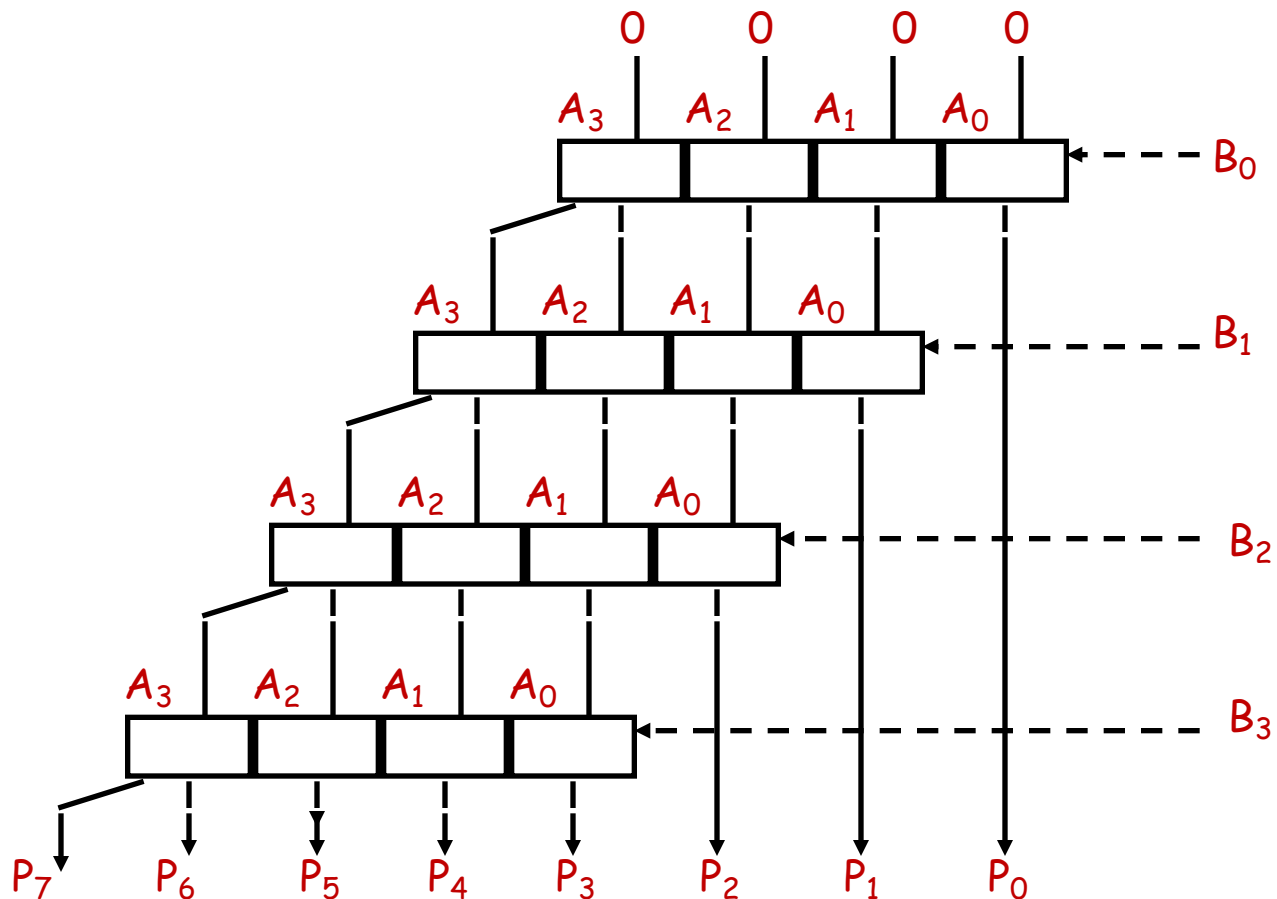 - 75% reduction compared to full RC

Multiplier

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

Combinational Multiplier

Shift-Add Multiplier
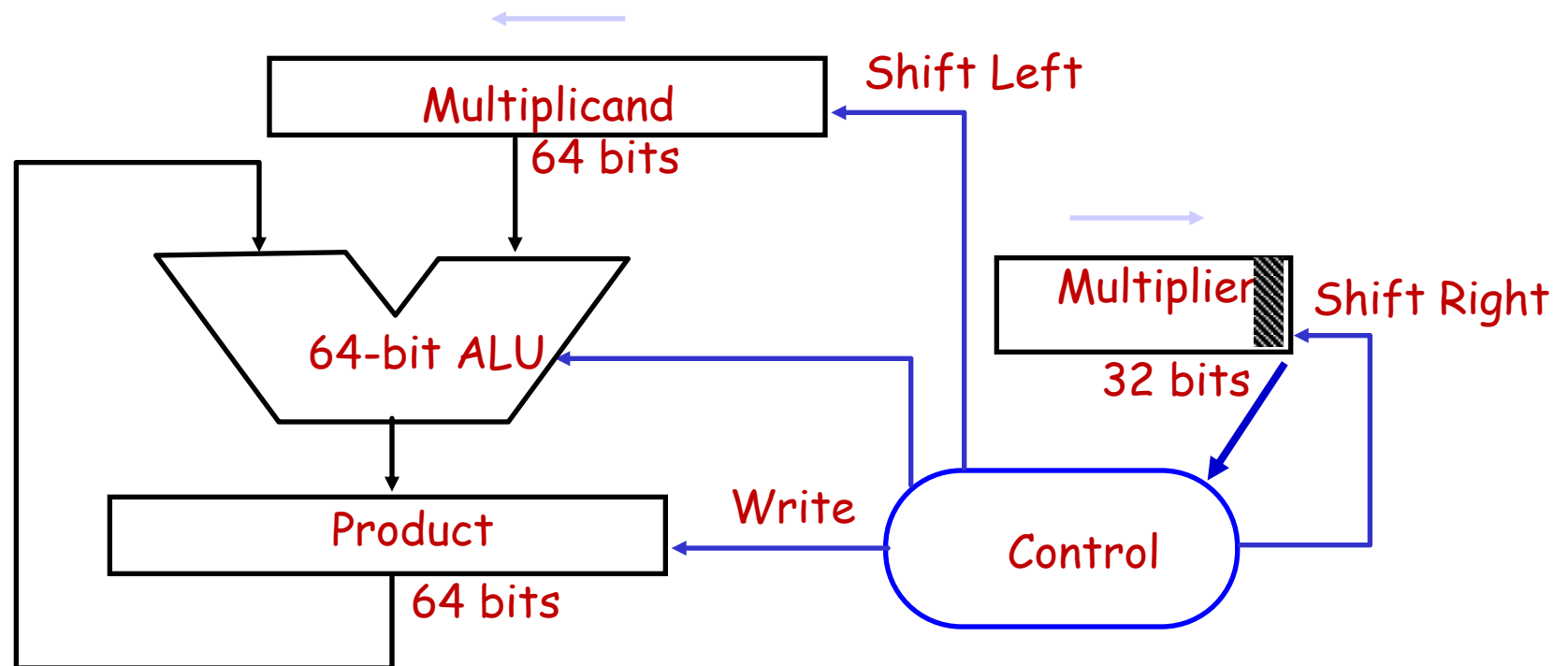
Booth's Algorithm

- Example 2 x 6 = 0010 x 0110:

```
                    0010
    x               0110
    +               0000     shift (0 in multiplier)
    +              0010      add (1 in multiplier)
    +             0010       add (1 in multiplier)
    +            0000        shift (0 in multiplier)
                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
                    0000 shift (0 in multiplier)
    -              0010   sub (first 1 in multply)
                   0000   shift (mid string of 1s)
    +             0010    add (prior step had last 1)
                00001100
```

Booth's Algorithm (cont.)

Current Bit	Bit to Right	Explanation	Example	Op
1	0	Begins run of 1s	0001111000	sub
1	1	Middle of run of 1s	0001111000	none
0	1	End of run of 1s	0001111000	add
0	0	Middle of run of 0s	0001111000	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) x 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 11011 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

(1)   `00000`   (left half of product)

    `–00010`   (mulitplicand)

     `11110`   (uses a phantom borrow)

- 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

  (1) `11111`   (left half of product)

   `+00010`   (mulitplicand)
   _____
    `00001`   (drop the leftmost carry)

- 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

$$00000 \; 11011 \; 0$$

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

  (1) `00000`   (left half of product)

      `–00010`   (mulitplicand)

      `11110`   (uses a phantom borrow)

- 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 01101 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:

$$\texttt{11111 10110}$$

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) x 2

Thanks for Your Attention!