

CS 31007

Autumn 2021

# COMPUTER ORGANIZATION AND ARCHITECTURE

---

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

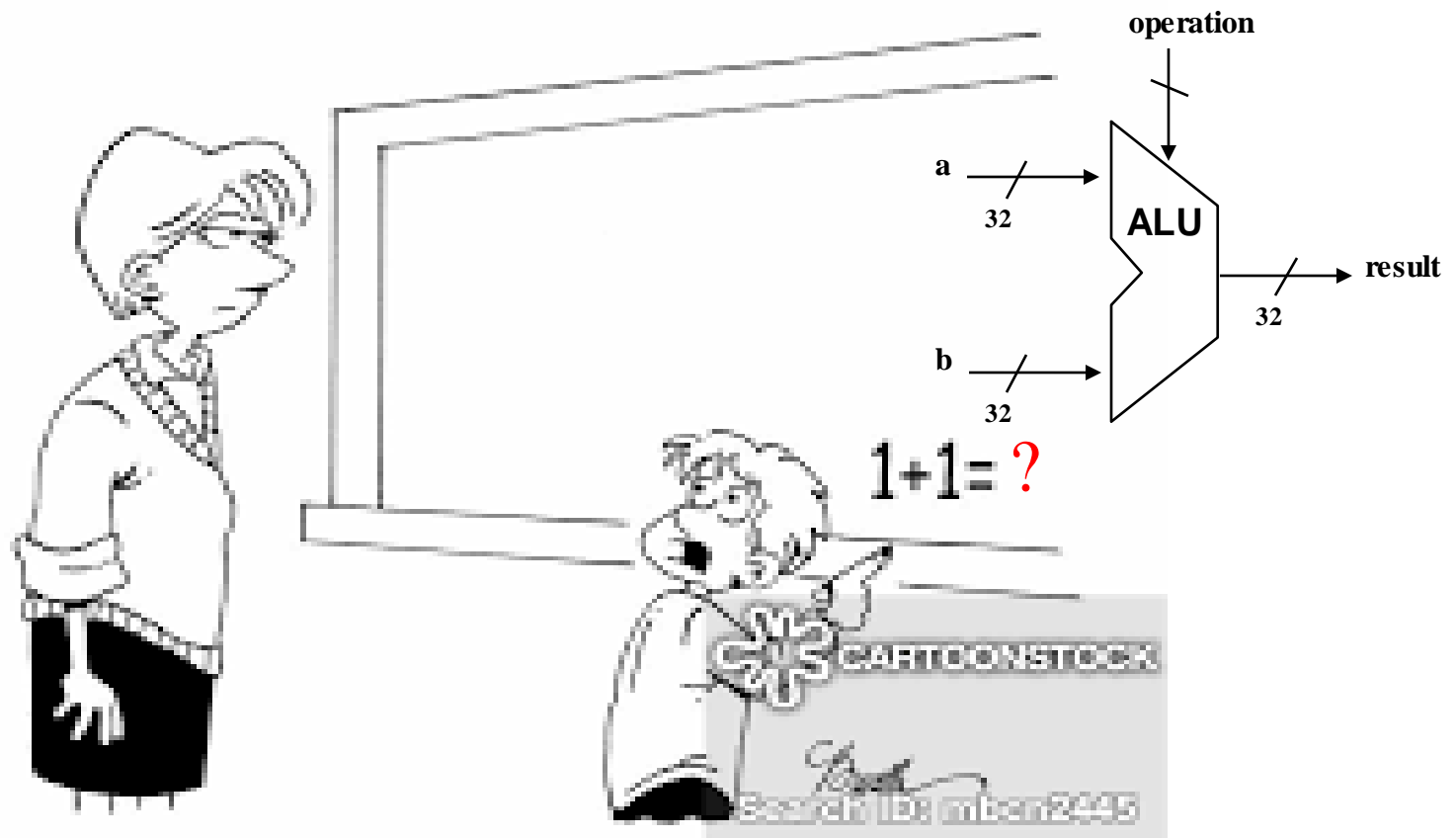
Lecture #16, #17: Computer Arithmetic

07 September 2021

---

Indian Institute of Technology Kharagpur  
*Computer Science and Engineering*

# Computer Arithmetic



"Could I have some computer time to troubleshoot this problem?"

# So far covered ...

- ❖ Evolution and history of computer design
- ❖ Basic components of a computer
- ❖ Instruction Set Architecture (ISA)
- ❖ CPU Performance
- ❖ MIPS Instruction Set, Programming

*Acknowledgement:* Patterson and Hennessy

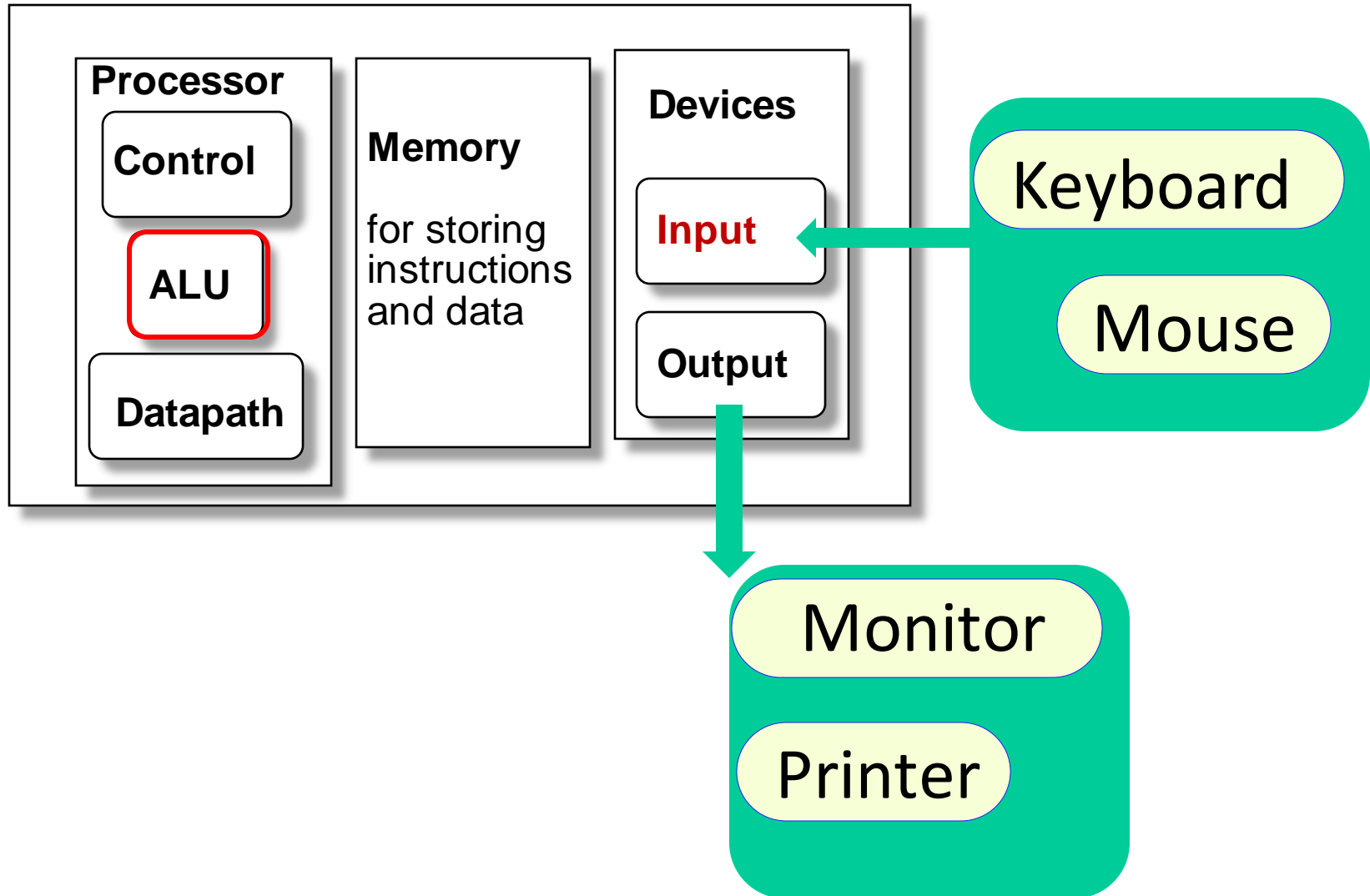
# Observed so far ...

RISC's underlying principles lead to efficient hardware design

- ❖ Simplicity favors regularity
- ❖ Make the common case fast
- ❖ Smaller is faster
- ❖ Good design demands good compromises

*Acknowledgement:* Patterson and Hennessy

# What is inside?



# Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
  - Hardware implementation of ALU
- Floating-point real numbers
  - Representation and operations
  - Overflow and underflow
  - Hardware implementation of FP-operations

# What determines the execution time of a machine/assembly-level program $P$ when it is run on a machine $M$ ?

- $P$  consists of a number of machine-level instructions (IC: *instruction count*);
- Each machine instruction requires several clock cycles to complete (CPI: average number of *clock cycles per instruction*);
- Each clock cycle has certain time period (CCT: *clock cycle time*)

**Thus, CPU-time = IC  $\times$  CPI  $\times$  CCT**

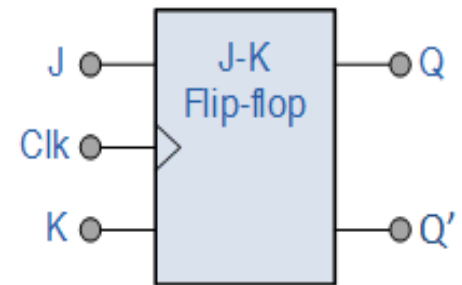
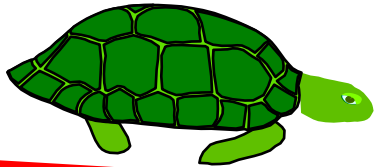
**(CPU Performance Equation)**

# Clock Timing

Does it mean that CCT is exclusively determined by the critical delay?  
Once the hardware is designed, is it fixed for a given technology?

.. Oh no, there is a catch!

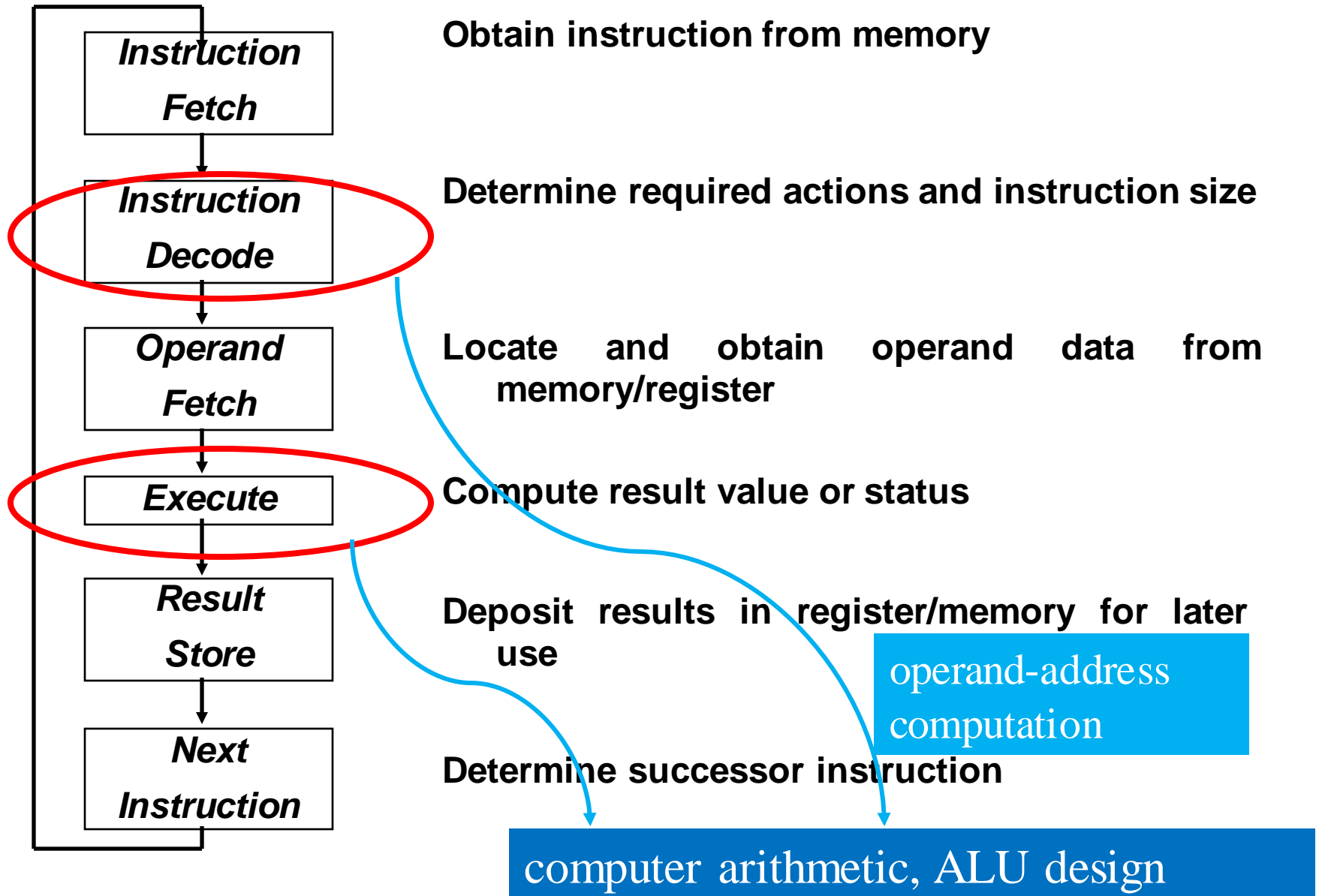
You can redesign hardware so that delay becomes smaller: (i) using low-depth logic (increases cost!), or (ii) use retiming, i.e., insert additional state elements (FFs) on long logic paths (increases #clock cycles to perform the same task, thus impacting CPI, though CCT is reduced!)



Clock period should be large enough to accommodate delays along critical paths in the circuit (longest ones); but not too large – system slows down unnecessarily



# Execution Cycle



# MIPS Example

- C code:

`f = (g + h) - (i + j);`

- `f, ..., j` in `$s0, ..., $s4`

- Compiled MIPS code:

`add $t0, $s1, $s2`

`add $t1, $s3, $s4`

`sub $s0, $t0, $t1`

Adders must be very  
efficient: cost, speed

Amdahl

Make common operations faster ...

## 2's Complement Integer Arithmetic: Dilemma of Final Carry and Overflow

| Decimal | 2's Complement |
|---------|----------------|
| 0       | 0000           |
| 1       | 0001           |
| 2       | 0010           |
| 3       | 0011           |
| 4       | 0100           |
| 5       | 0101           |
| 6       | 0110           |
| 7       | 0111           |

universe:  
4-bit 2's complement  
arithmetic

| Decimal | 2's Complement |
|---------|----------------|
| -1      | 1111           |
| -2      | 1110           |
| -3      | 1101           |
| -4      | 1100           |
| -5      | 1011           |
| -6      | 1010           |
| -7      | 1001           |
| -8      | 1000           |

$$\begin{array}{r} 0100 \quad (+4) \\ + 1011 \quad (-5) \\ \hline 1111 \quad (-1) \end{array}$$

no final carry;  
no overflow;  
result correct

$$\begin{array}{r} 1100 \quad (-4) \\ + 0110 \quad (+6) \\ \hline 10010 \quad (+2) \end{array}$$

final carry discarded;  
result still correct;  
no overflow

$$\begin{array}{r} 1000 \quad (-8) \\ + 1000 \quad (-8) \\ \hline 10000 \quad (0) \end{array}$$

final carry ✓;  
overflow ✓;  
result invalid

$$\begin{array}{r} 0100 \quad (+4) \\ + 0100 \quad (+4) \\ \hline 1000 \quad (-8) \end{array}$$

no final carry;  
overflow ✓;  
result invalid

# Overflow Detection Logic: Hardware Solution

sign-bit

$$\begin{array}{r} A: a_{n-1}, a_{n-2}, \dots, a_1, a_0 \\ B: b_{n-1}, b_{n-2}, \dots, b_1, b_0 \\ + \\ \hline S: s_{n-1}, s_{n-2}, \dots, s_1, s_0 \end{array}$$

$$\text{Overflow} = a_{n-1}b_{n-1}\overline{s_{n-1}} + \overline{a_{n-1}}\overline{b_{n-1}}s_{n-1}$$

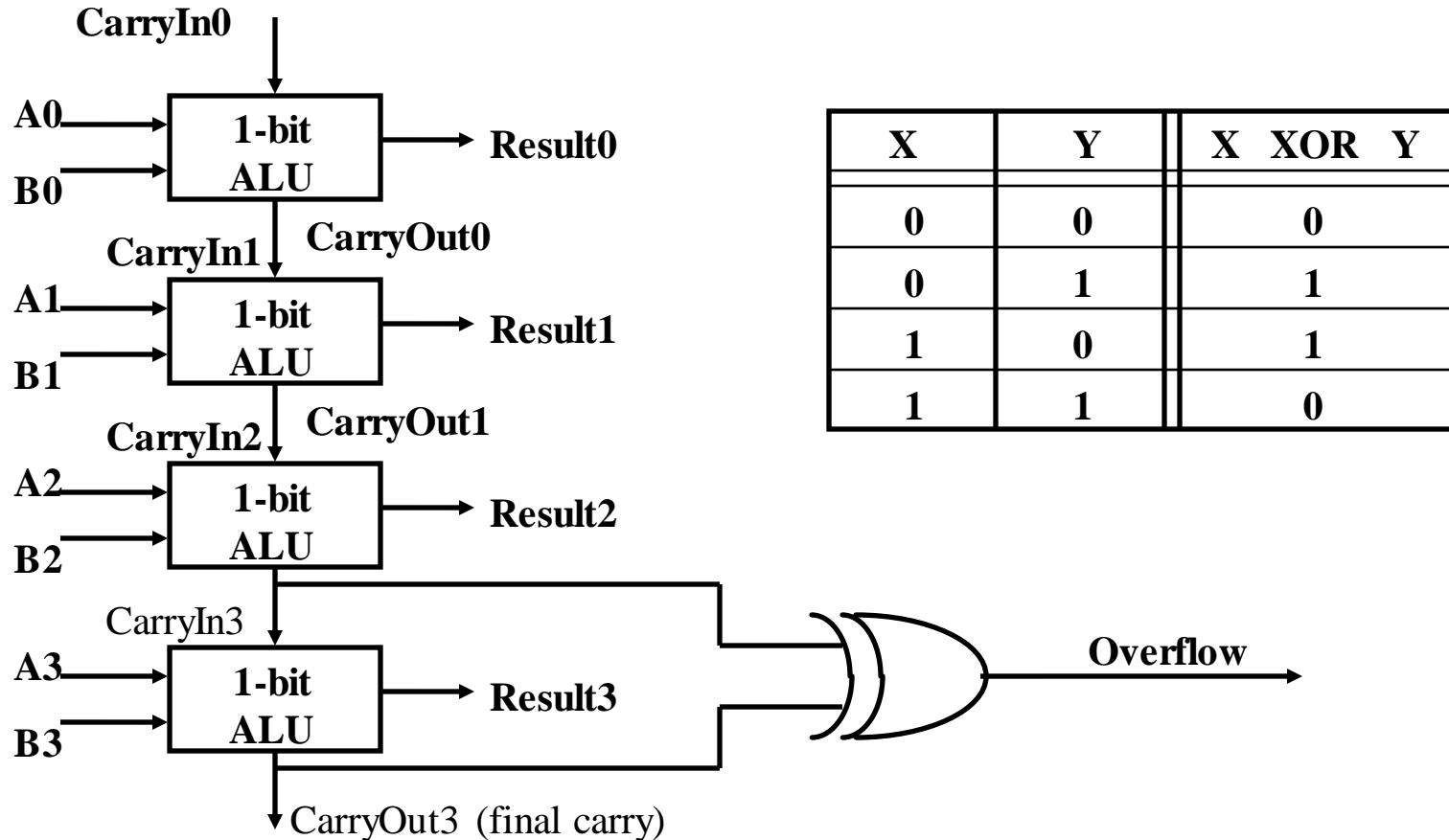
- overflow  $\rightarrow$  adding two positives yields a negative
- or, adding two negatives gives a positive

In MIPS, on detecting overflow, interrupt (exception) is invoked;  
*add, addi* (overflow considered); *addu, addiu* (overflow ignored)

# Overflow Detection Logic: Another Solution

◦ Carry into MSB ◦ Carry out of MSB

- For a N-bit ALU:  $\text{Overflow} = \text{CarryIn}[N - 1] \text{ XOR } \text{CarryOut}[N - 1]$



Show that these two solutions are logically equivalent

# Hazards of Finite-Precision Arithmetic

◦ Compute:  $X = A + B - C$ ;      $A, B, C$  are +ve integers;

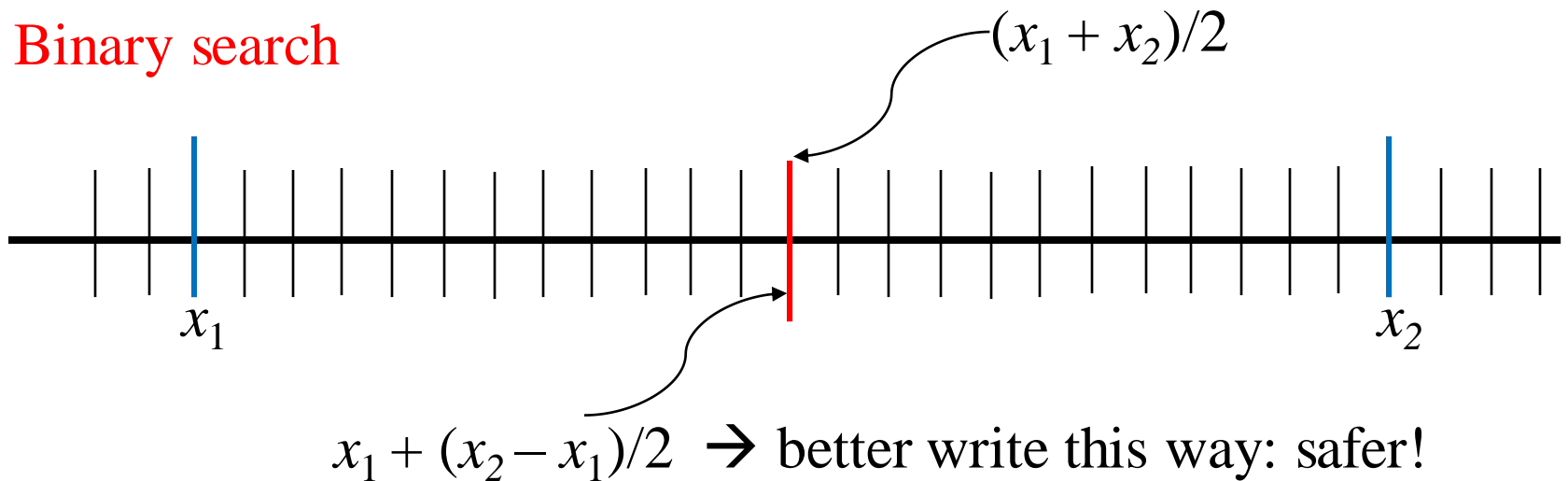
◦  $X = (A + B) - C$

Or,

◦  $X = A + (B - C) ?$

The first choice might cause *overflow* – invalid result  
The second option is *safer*

Binary search

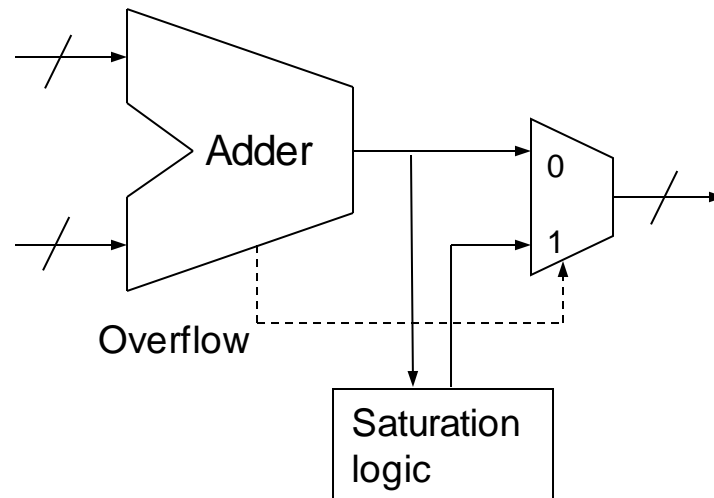


# Saturating Adders

## Saturating arithmetic:

When a result is out of range, provide the most positive or the most negative value that is representable

Required in many DSP applications



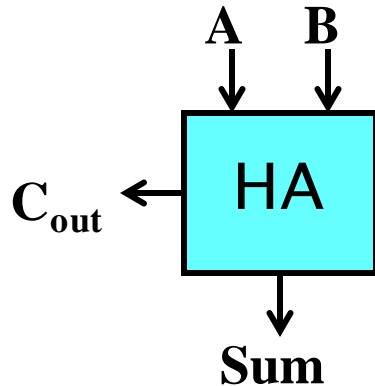
# Why Adders?

- Addition: a fundamental operation
  - Basic block of most arithmetic operations
  - Address calculation
- Faster and faster
- How?
  - Architectural-level optimization
  - Gate-level optimization
  - Speed/area trade-off



# Review: 1-bit adder

- One-bit Half Adder:

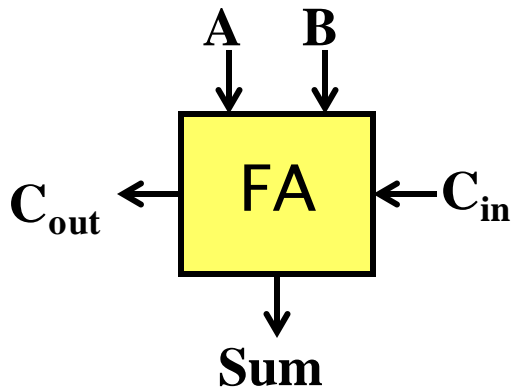


$$\text{Sum} = A \oplus B$$

$$C_{\text{out}} = A.B$$

| A | B | Sum | C <sub>out</sub> |
|---|---|-----|------------------|
| 0 | 0 | 0   | 0                |
| 0 | 1 | 1   | 0                |
| 1 | 0 | 1   | 0                |
| 1 | 1 | 0   | 1                |

- One-bit Full Adder:



$$\text{Sum} = A \oplus B \oplus C_{\text{in}}$$

$$C_{\text{out}} = A.B + B.C_{\text{in}} + A.C_{\text{in}}$$

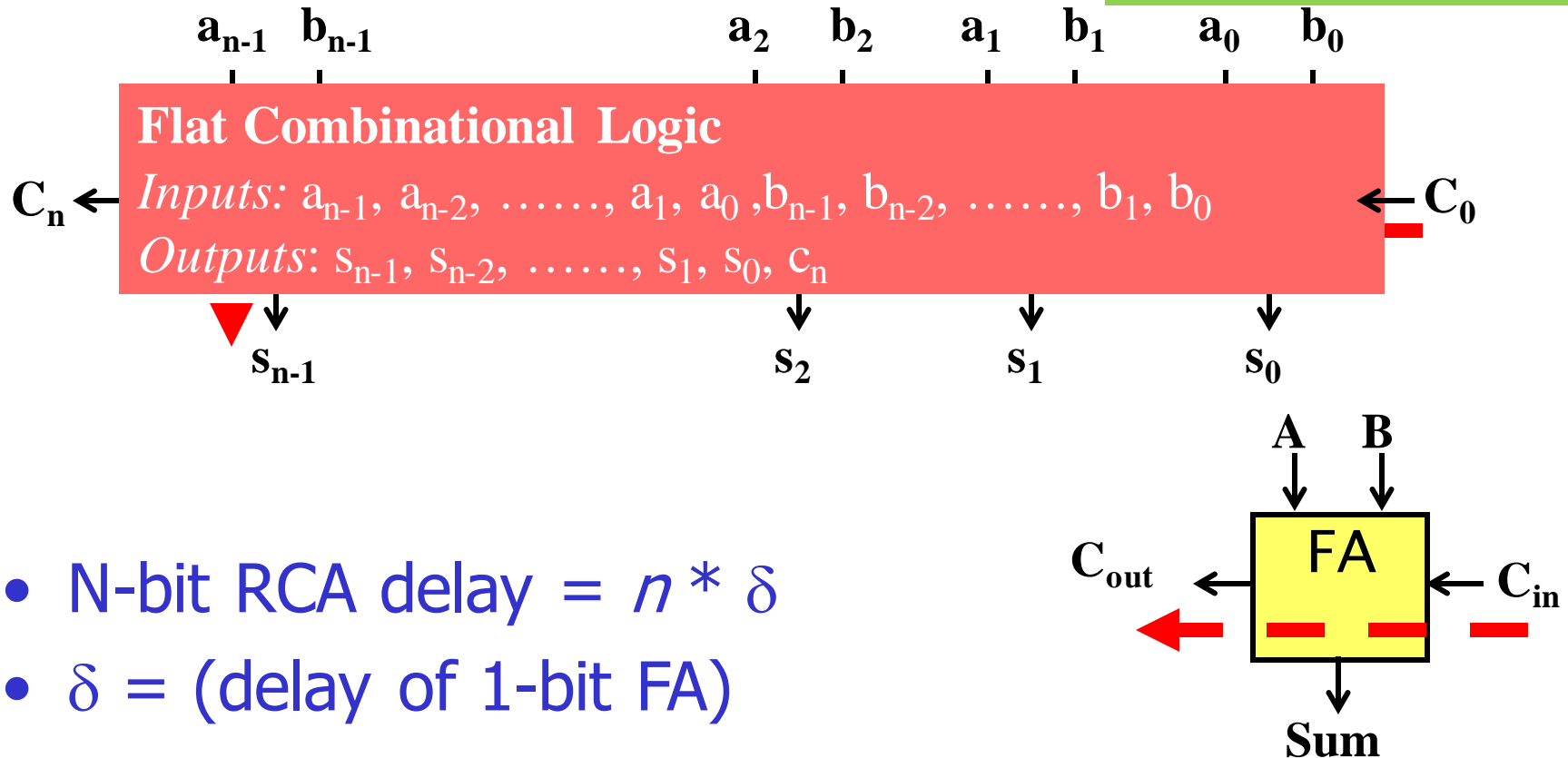
| C <sub>in</sub> | A | B | Sum | C <sub>out</sub> |
|-----------------|---|---|-----|------------------|
| 0               | 0 | 0 | 0   | 0                |
| 0               | 0 | 1 | 1   | 0                |
| 0               | 1 | 0 | 1   | 0                |
| 0               | 1 | 1 | 0   | 1                |
| 1               | 0 | 0 | 1   | 0                |
| 1               | 0 | 1 | 0   | 1                |
| 1               | 1 | 0 | 0   | 1                |
| 1               | 1 | 1 | 1   | 1                |

# $n$ -bit Ripple-Carry Adder (RCA)

- To add two  $n$ -bit numbers

A:  $a_{n-1}, a_{n-2}, \dots, a_1, a_0$

B:  $b_{n-1}, b_{n-2}, \dots, b_1, b_0$



- $N$ -bit RCA delay =  $n * \delta$
- $\delta$  = (delay of 1-bit FA)

Critical path in an  $n$ -bit ripple-carry adder

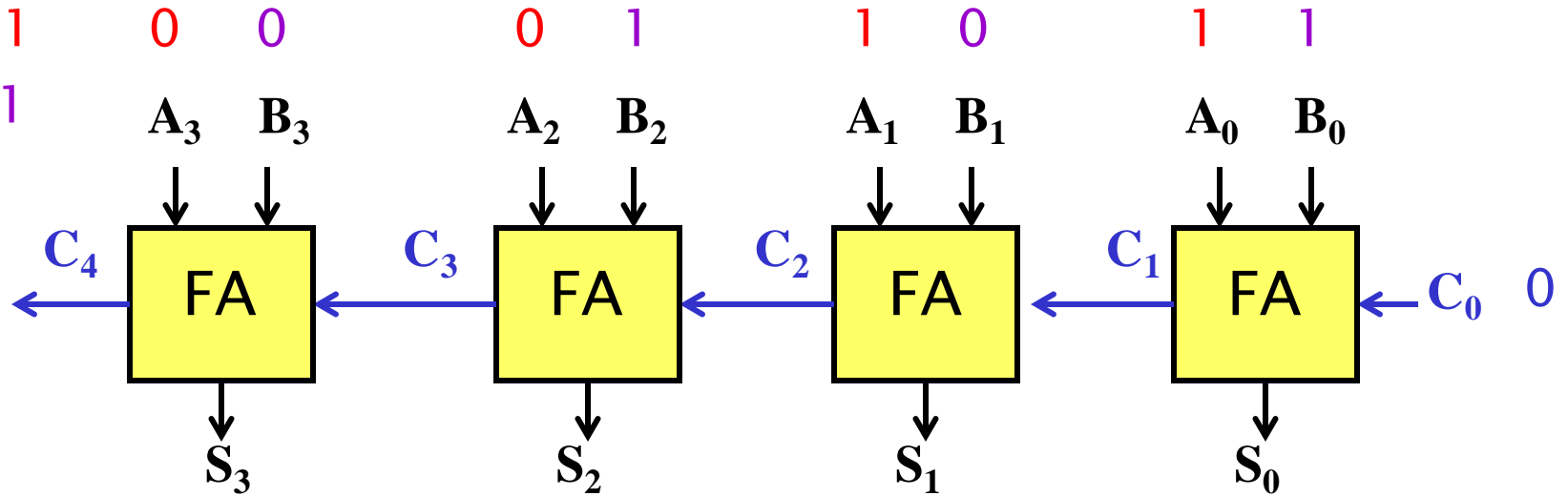
→ CCT needs to accommodate long delay

Cost =  $O(n)$ ; delay =  $O(n)$

# 4-bit Ripple-Carry Addition: Example

A=0011

B=0101



|     |   |   |   |   |   |   |   |   |        |
|-----|---|---|---|---|---|---|---|---|--------|
| T=0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | S=0000 |
| T=1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | S=0110 |
| T=2 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | S=0100 |
| T=3 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | S=0000 |
| T=4 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | S=1000 |

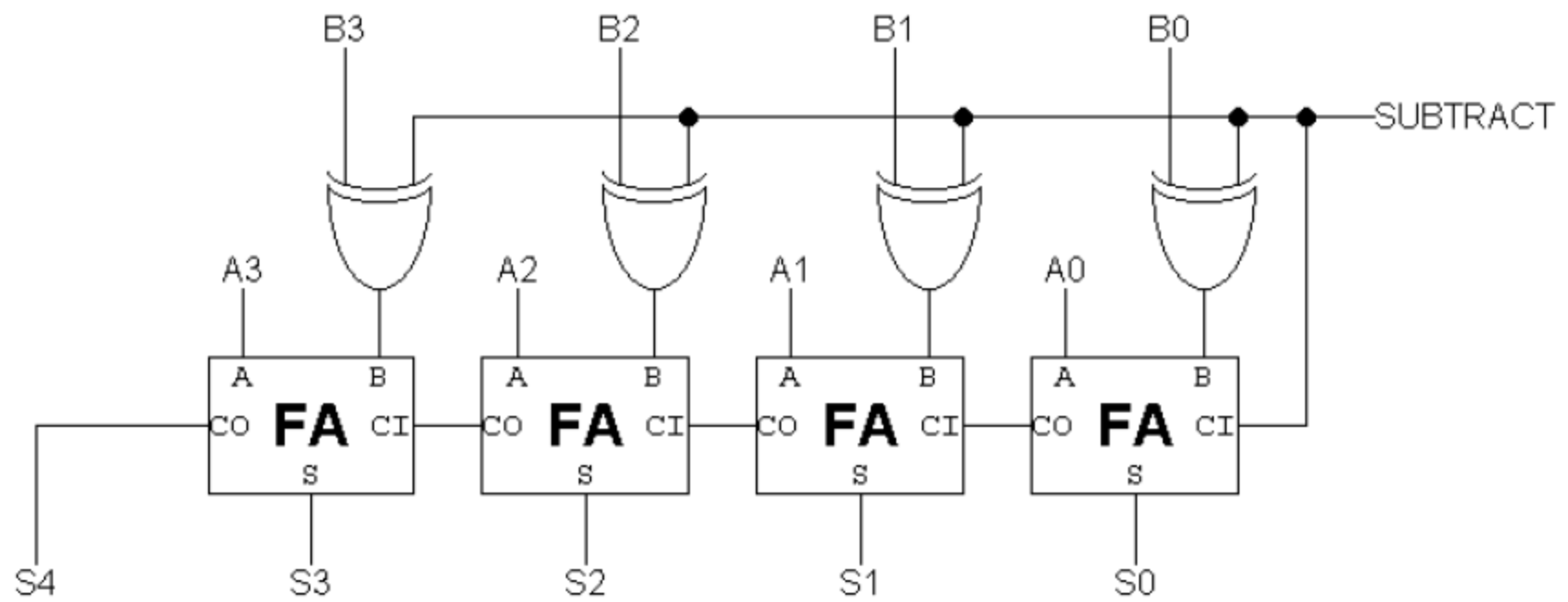
delay = 4 units; also, overflow has occurred 

Using 2's complement representation:  $-B = \sim B + 1$

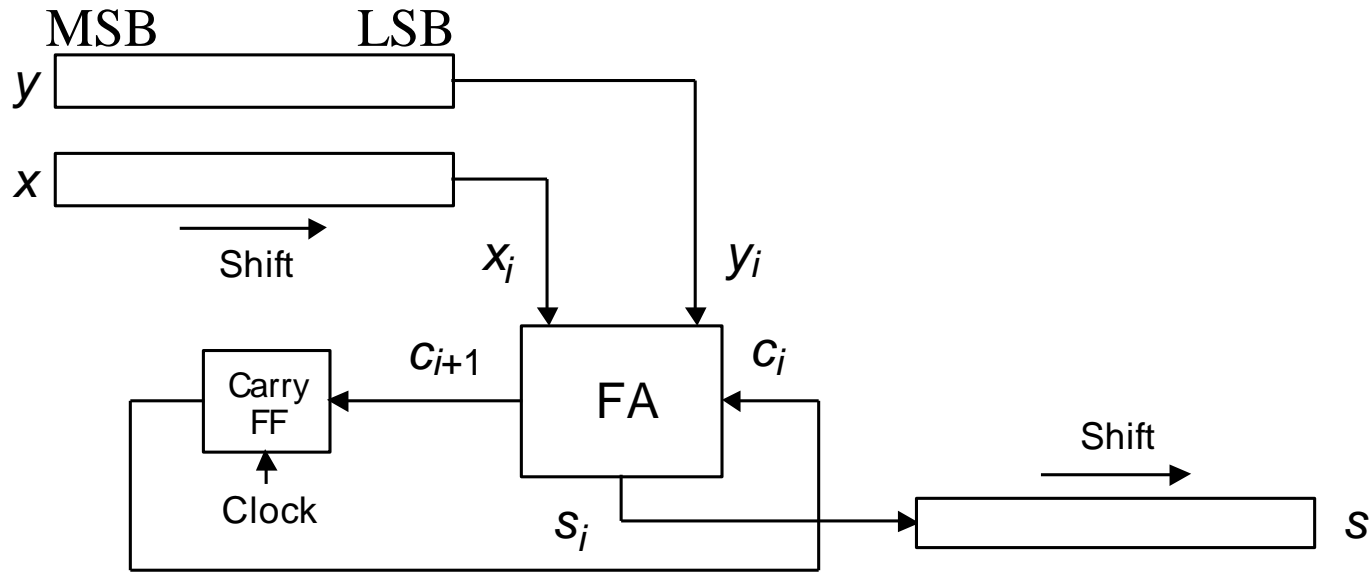
$\sim$  = bit-wise complement



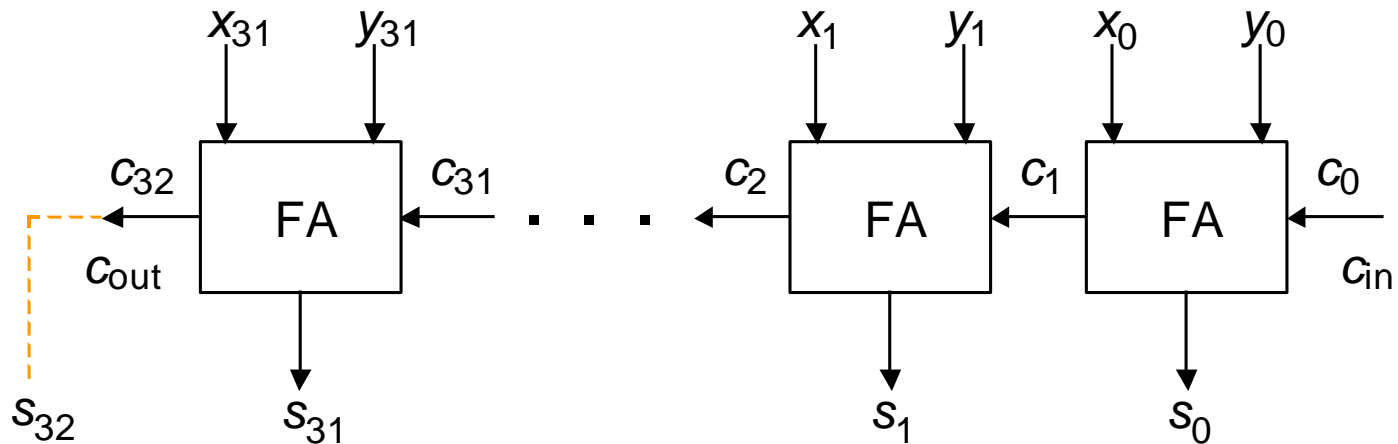
So let's build an arithmetic unit that does both addition and subtraction. Operation selected by *control input*.



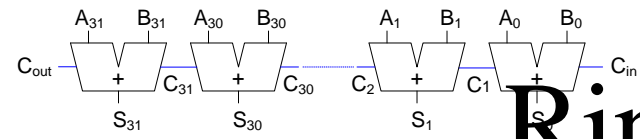
# Multi-bit Adder



(a) Bit-serial adder.



(b) Ripple-carry adder.



# Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through the entire chain
- Disadvantage: **slow**

# Possible solutions to mitigate carry-propagation delay

1. Detect the end of carry propagation rather than wait for the worst-case time
2. Speed-up propagation using techniques such as
  - lookahead
  - carry-select
  - logarithmic adder
3. Limit carry propagation to within a small number of bits
4. Trade-off: speed (or inversely, delay) *versus* logic cost

CS 31007

Autumn 2021

# COMPUTER ORGANIZATION AND ARCHITECTURE

---

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #18: Computer Arithmetic

13 September 2021

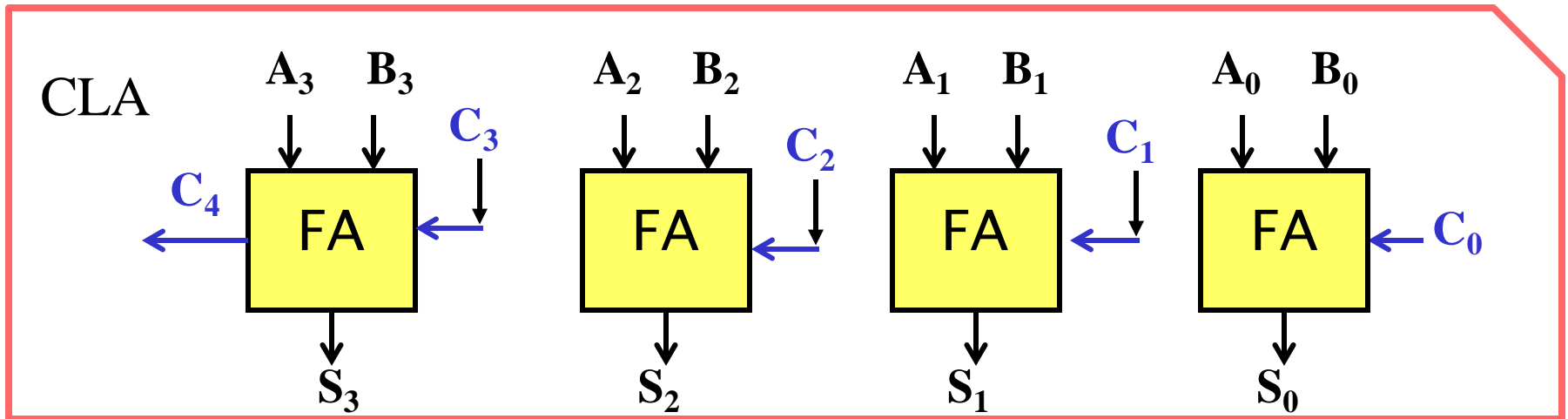
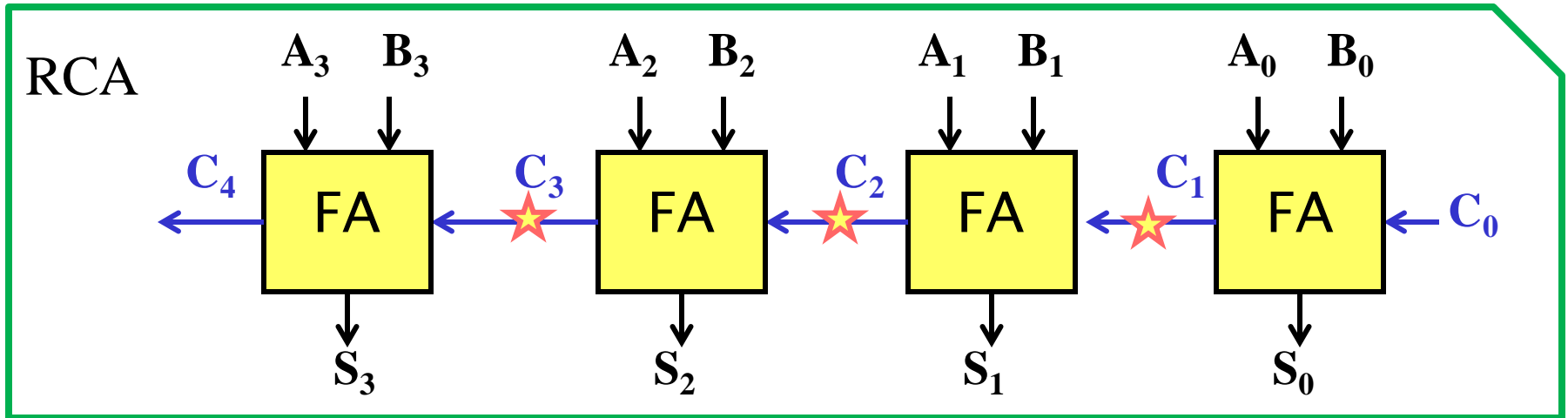
---

Indian Institute of Technology Kharagpur  
*Computer Science and Engineering*



How to speed up computer arithmetic?

# Carry-Lookahead Adder



Instead of RCA, use  $n$  independent 1-bit FA-modules working in parallel;  
no signal propagation through stages; directly generate all  
carry-bits from inputs

# Carry-Lookahead Adder

- Generate all carry bits **directly from inputs**, instead of *rippling through* FA-blocks sequentially
- Few notation:
  - Generate ( $g_i$ ) and propagate ( $p_i$ ) signals for each bit:
    - A column will generate a carry out if  $a_i$  AND  $b_i$  are both 1.

$$g_i = a_i b_i$$

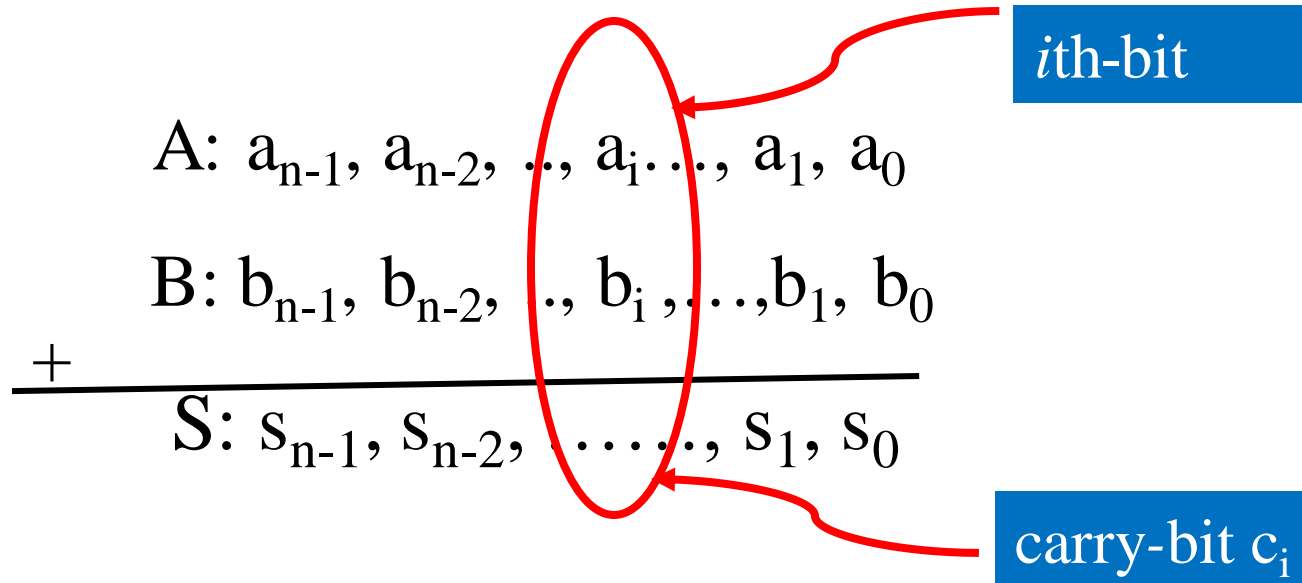
- A column will propagate a carry in to the carry out if  $A_i$  OR  $B_i$  is 1.

$$p_i = a_i + b_i$$

- The carry out of a column ( $C_i$ ) is:

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i = g_i + p_i c_i$$

## Carry Lookahead Adder (CLA): Basic Idea



At the *i*th stage, carry will be generated for propagation to the next stage if and only if

$a_i = b_i = 1; \Rightarrow g_i = a_i b_i \Rightarrow$  carry-generate function

or either  $a_i$  or  $b_i = 1$ , and previous carry  $c_{i-1}$  arrives here;  
 $\Rightarrow p_i = a_i + b_i \Rightarrow$  carry-propagate function


# Carry-Lookahead Adder

- Generate all carry bits **directly from inputs**, instead of *rippling through* FA-blocks sequentially
- Few notation:
  - Generate ( $g_i$ ) and propagate ( $p_i$ ) signals for each bit:
    - A column will generate a carry out if  $a_i$  AND  $b_i$  are both 1.
  - $g_i = a_i b_i$
  - A column will propagate a carry in to the carry out if  $A_i$  OR  $B_i$  is 1.
  - $p_i = a_i + b_i$
  - The carry out of a column ( $C_i$ ) is:

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i = g_i + p_i c_i$$

# Carry-Lookahead Adder

We can visualize carry “generate” and “propagate” functions from another perspective:

$$\begin{aligned}C_{i+1} &= a.b + b.c_{in} + a.c_{in} \\ &= a.b + c_{in} \cdot (a+b)\end{aligned}$$


- A column will generate a carry out if  $a_i$  AND  $b_i$  are both 1, i.e.,  $g_i = a_i b_i$

- A column will propagate a carry if  $A_i$  or  $B_i$  is 1, i.e.,  $p_i = a_i + b_i$

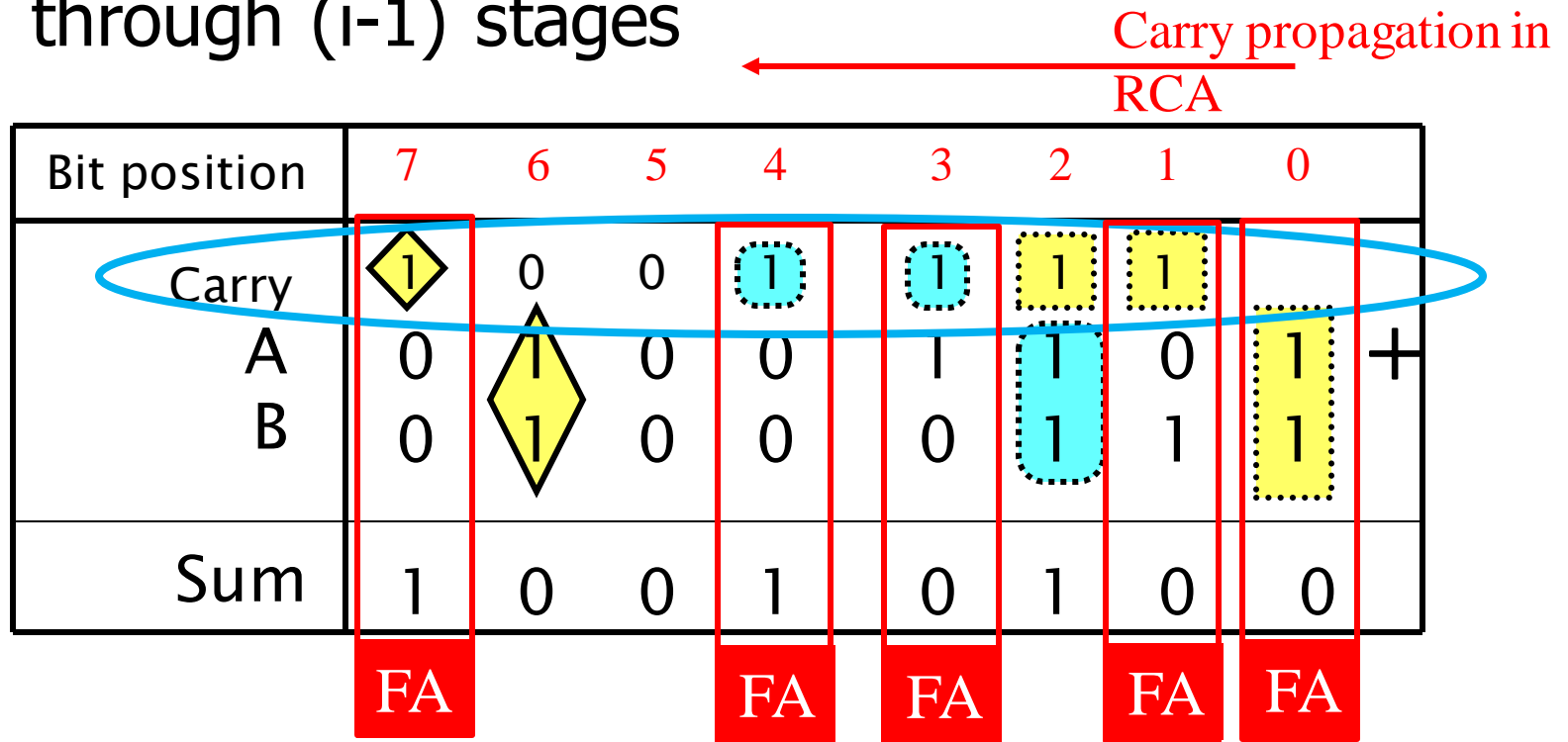
- The carry out of a column ( $C_i$ ) is:

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i = g_i + p_i c_i$$

# Carry-Lookahead Adder

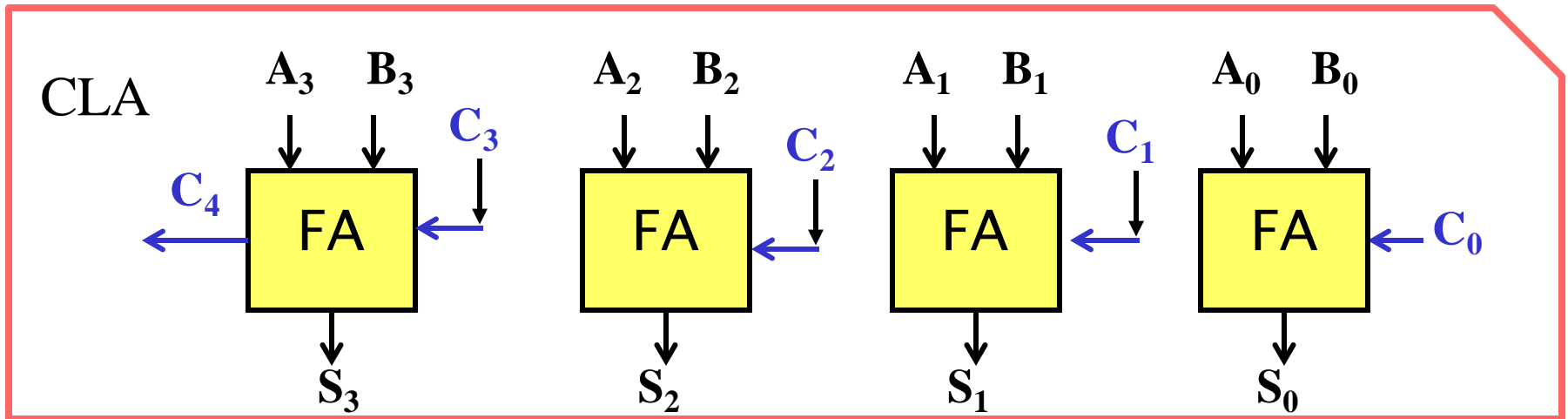
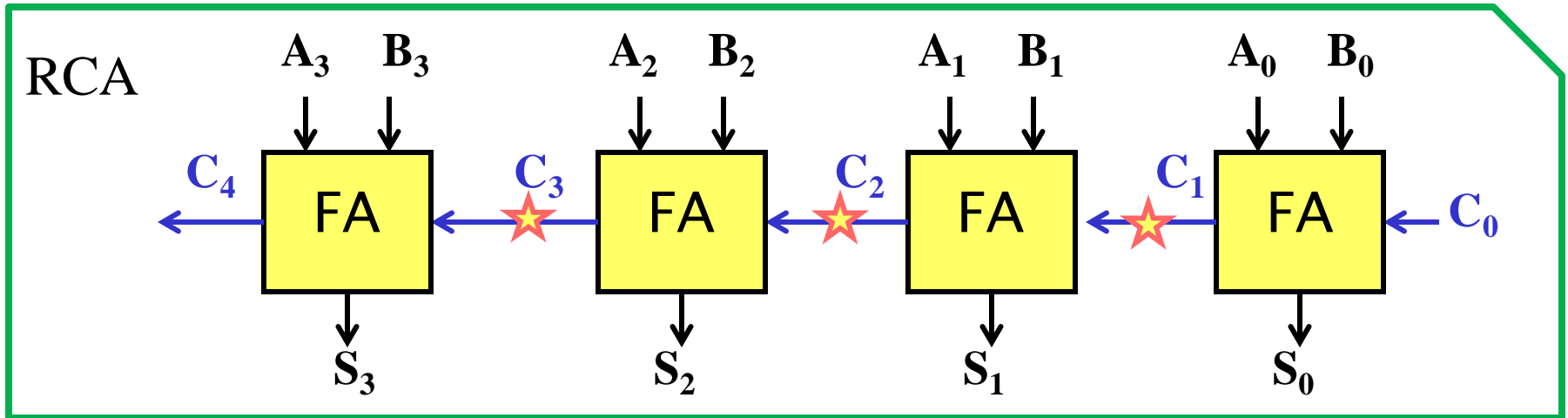
- Idea:

- Produce  $c_i$  ahead of time instead of passing through  $(i-1)$  stages



Instead of RCA, use *independent* 1-bit FA-modules  
no signal propagation through stages

# Carry-Lookahead Adder



Instead of RCA, use  $n$  independent 1-bit FA-modules working in parallel;  
no signal propagation through stages; directly generate all  
carry-bits from inputs



# Basic Signals

Generate signal:

$$g_i = a_i b_i$$

Propagate signal:

$$p_i = a_i + b_i$$

Carry recurrence

$$c_{i+1} = g_i + c_i p_i$$

# Average Carry Propagation Length

Generate signal:

$$g_i = a_i b_i$$

Propagate signal:

$$p_i = a_i + b_i$$

Given binary numbers with random bits, for each position  $i$  we have

Probability of carry generation  $= 1/4$  (both 1)

Probability of carry annihilation  $= 1/4$  (both 0)

Probability of carry propagation  $= 1/2$  (different)

Average length of the longest carry chain for  $n$ -bit addition:  $O(\log_2 n)$

# Unrolling Carry Recurrence

$$\begin{aligned}
 c_i &= g_{i-1} + c_{i-1}p_{i-1} = \\
 &= g_{i-1} + (g_{i-2} + c_{i-2}p_{i-2})p_{i-1} = g_{i-1} + g_{i-2}p_{i-1} + c_{i-2}p_{i-2}p_{i-1} = \\
 &= g_{i-1} + g_{i-2}p_{i-1} + (g_{i-3} + c_{i-3}p_{i-3})p_{i-2}p_{i-1} = \\
 &= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + c_{i-3}p_{i-3}p_{i-2}p_{i-1} = \\
 &= \dots = \\
 &= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} + \dots + \\
 &\quad + g_0p_1p_2\dots p_{i-2}p_{i-1} + c_0p_0p_1p_2\dots p_{i-2}p_{i-1} =
 \end{aligned}$$

$$= \boxed{g_{i-1} + \sum_{k=0}^{i-2} g_k \prod_{j=k+1}^{i-1} p_j + c_0 \prod_{j=0}^{i-1} p_j}$$

# 4-bit Carry-Lookahead Adder

$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$

$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$

$$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$$

$$c_1 = g_0 + c_0 p_0$$

$$g_i = a_i b_i$$
$$p_i = a_i + b_i$$

$$c_{i+1} = g_i + c_i p_i$$

---

$$s_0 = x_0 \oplus y_0 \oplus c_0 = p_0 \oplus c_0$$

$$s_1 = p_1 \oplus c_1$$

$$s_2 = p_2 \oplus c_2$$

$$s_3 = p_3 \oplus c_3$$

# Carry Lookahead Adder

one AND-gate  
and  
one OR-gate

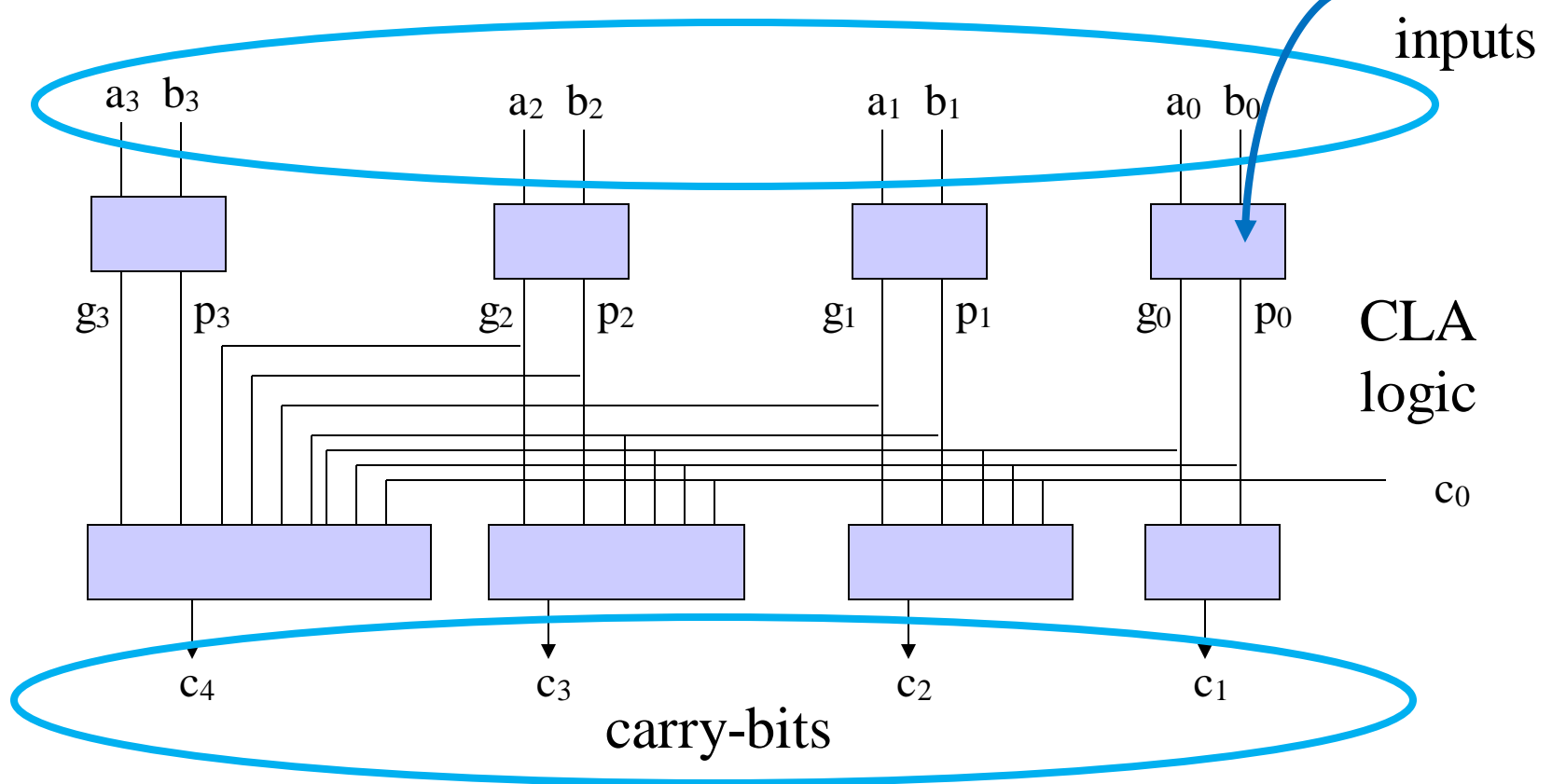
$$C_1 = a_0b_0 + (a_0+b_0)c_0 = g_0 + p_0c_0$$

$$C_2 = a_1b_1 + (a_1+b_1)c_1 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$$

$$C_3 = a_2b_2 + (a_2+b_2)c_2 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$$

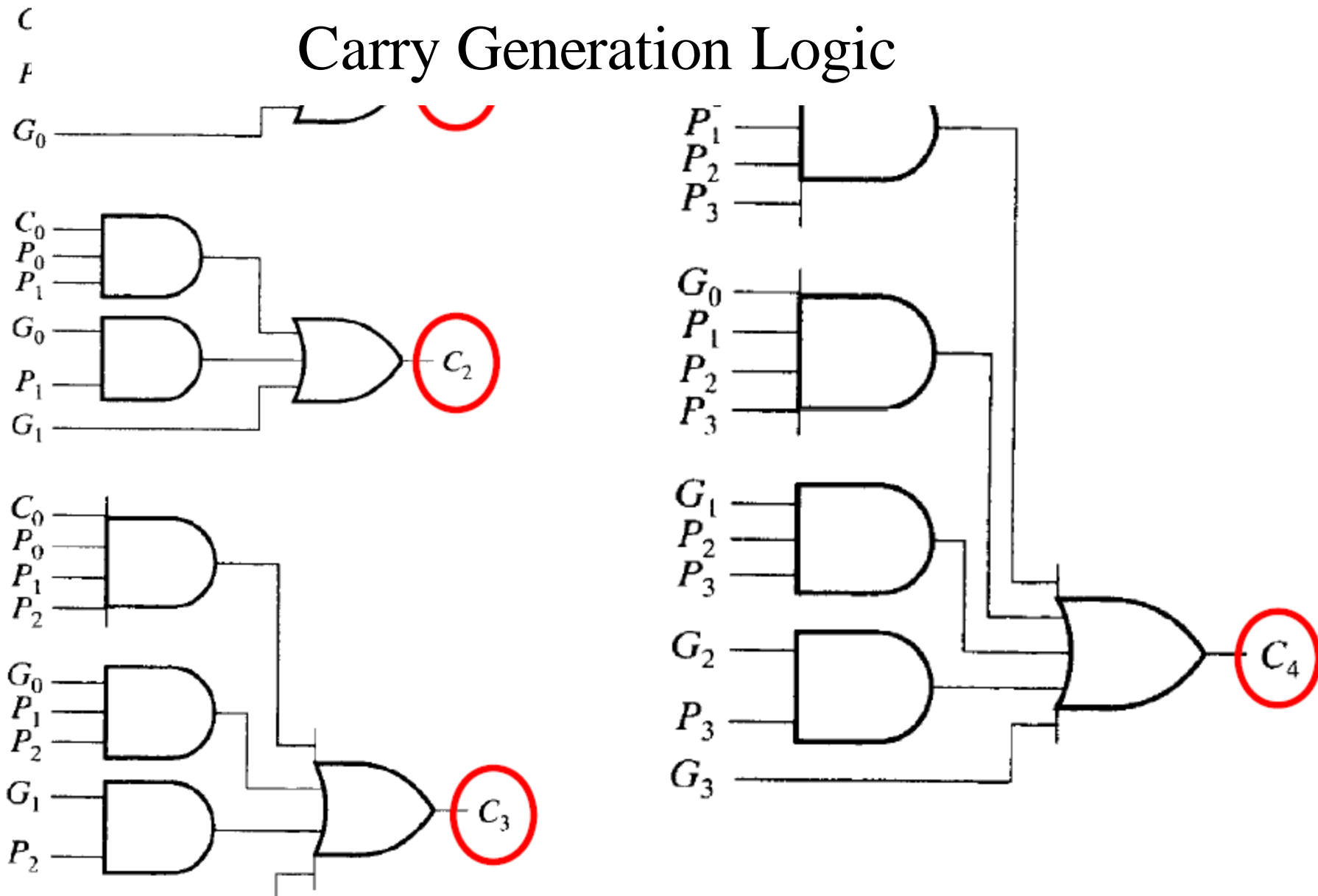
$$C_4 = a_3b_3 + (a_3+b_3)c_3 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

$$g_i = a_i b_i \quad p_i = a_i + b_i$$



# Carry Lookahead Circuits

## Carry Generation Logic



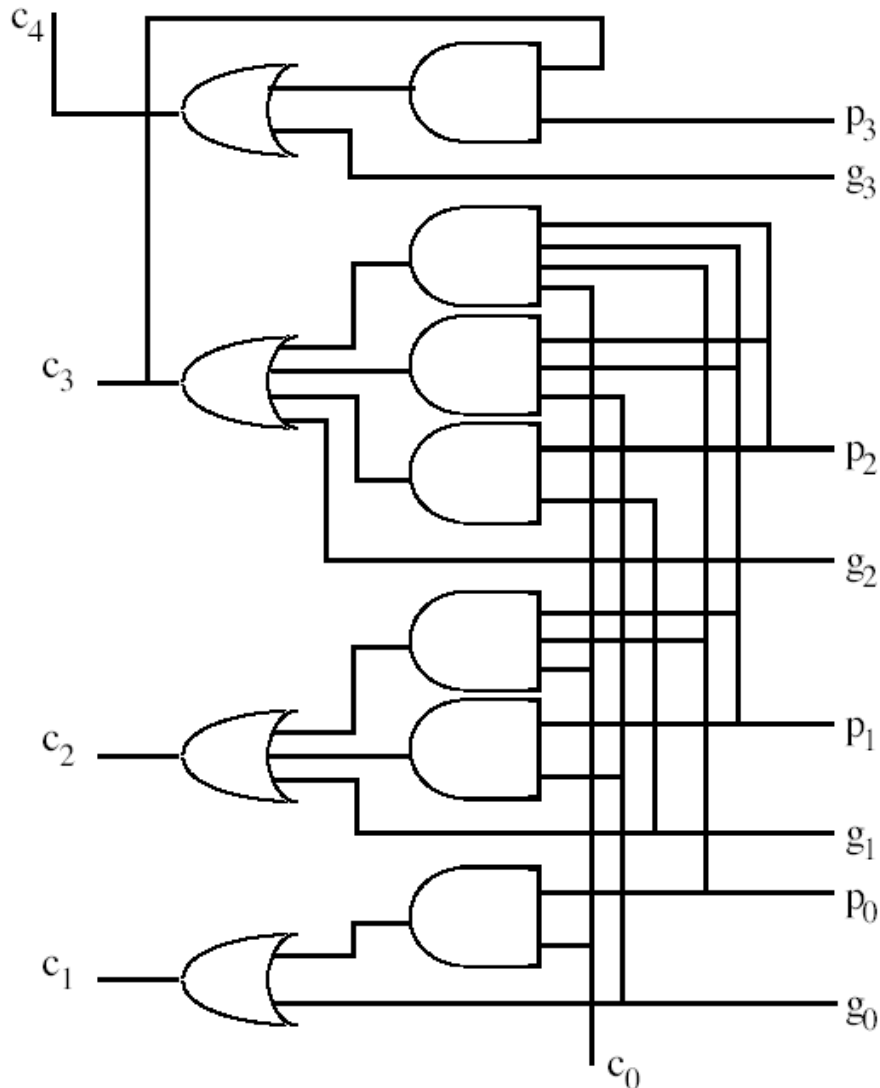
# 4-bit CLA Logic

A:  $a_3 a_2 a_1 a_0$

B:  $b_3 b_2 b_1 b_0$

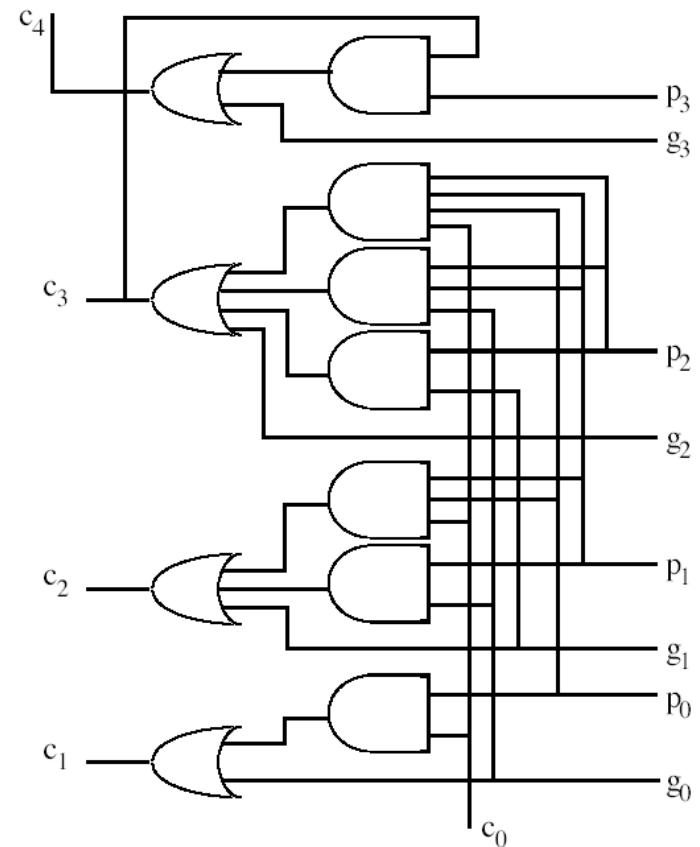
$$g_i = a_i b_i$$

$$p_i = a_i + b_i$$



all carry-bits are generated directly from inputs;  
one can thus use independent FA-modules, to produce the sum-bits in parallel!

# $n$ -bit CLA Adder



Much faster than RCA

Critical path delay in an  $n$ -bit CLA?

Cost =  $O(?)$ ; delay  $O(?)$



# Carry Lookahead Adder: Analysis

$$C_1 = a_0b_0 + (a_0+b_0)c_0 = g_0 + p_0c_0$$

$$C_2 = a_1b_1 + (a_1+b_1)c_1 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$$

$$C_3 = a_2b_2 + (a_2+b_2)c_2 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$$

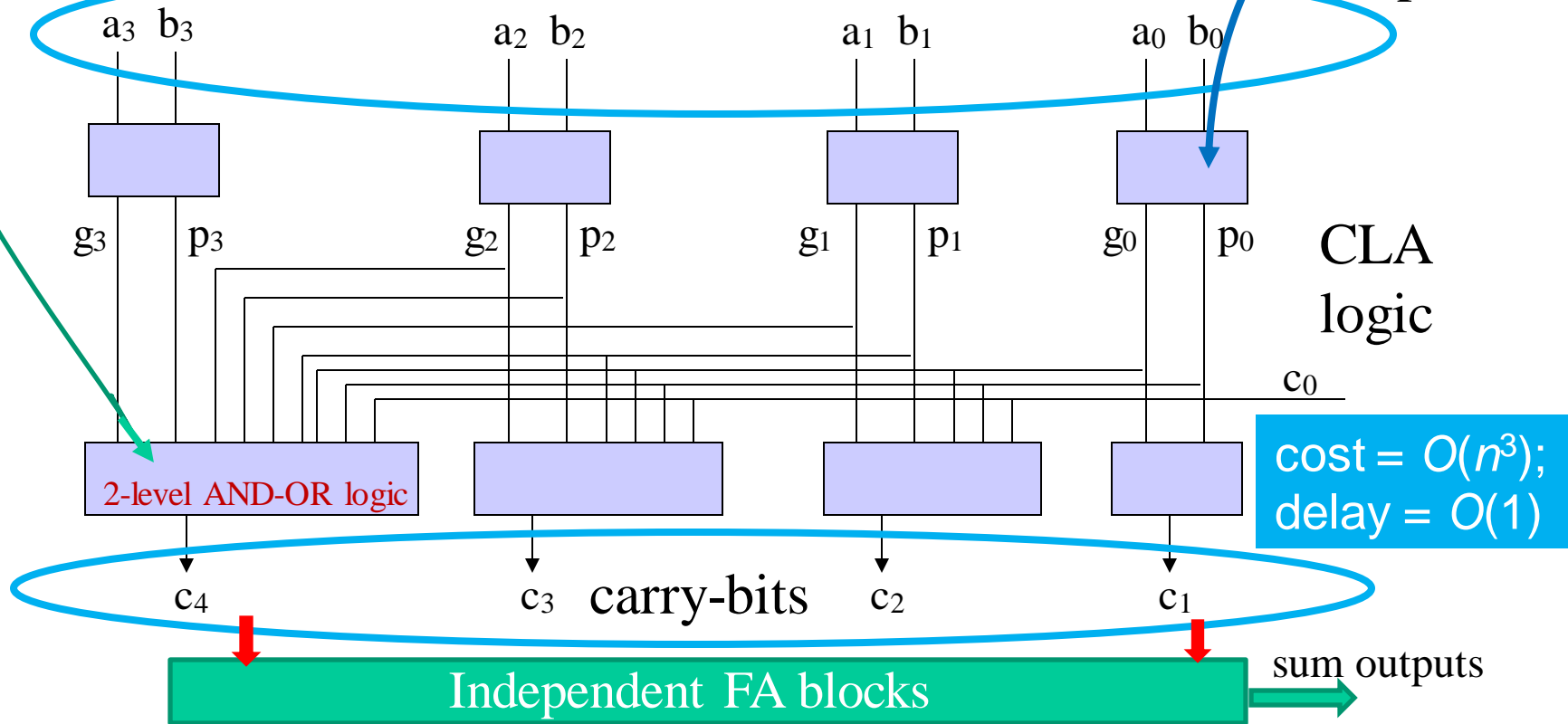
$$C_4 = a_3b_3 + (a_3+b_3)c_3 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

$$g_i = a_i b_i \quad p_i = a_i + b_i$$

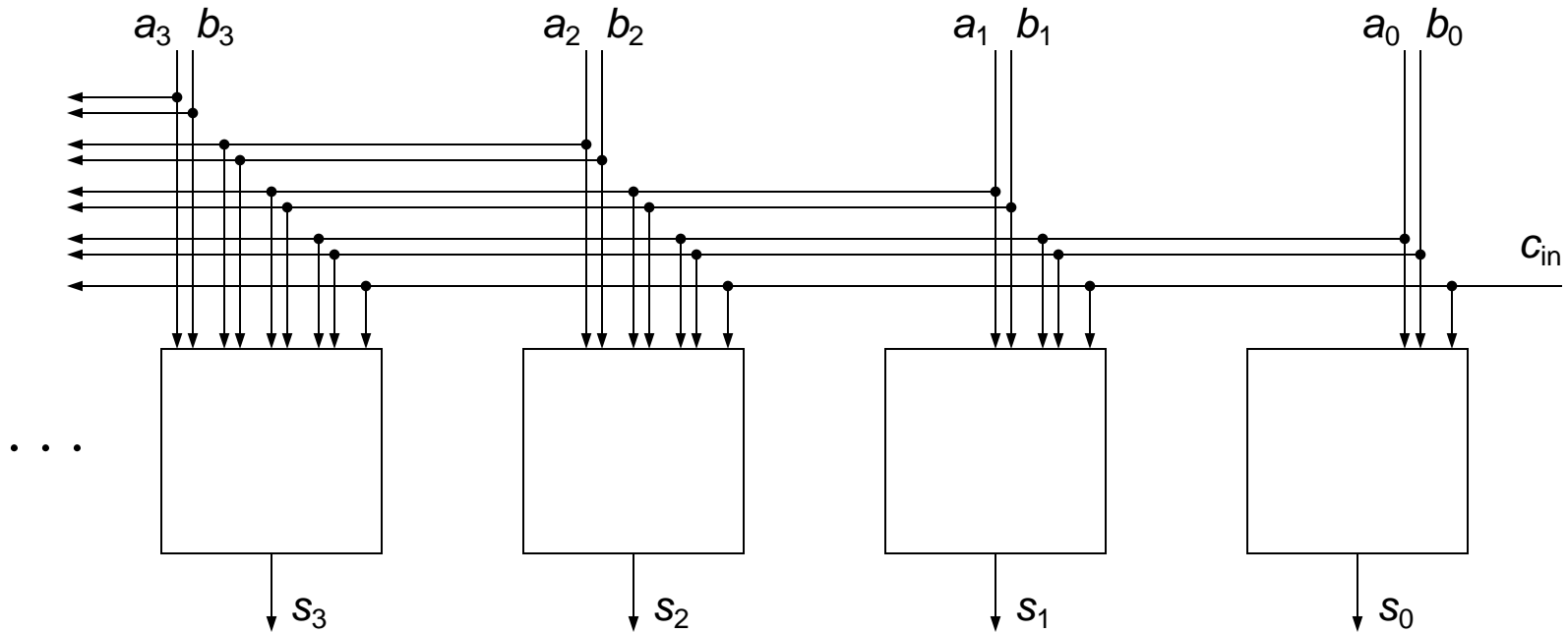
Cost =  $O(?)$ ; delay =  $O(?)$

one AND-gate  
and  
one OR-gate

inputs



# Full Carry Lookahead



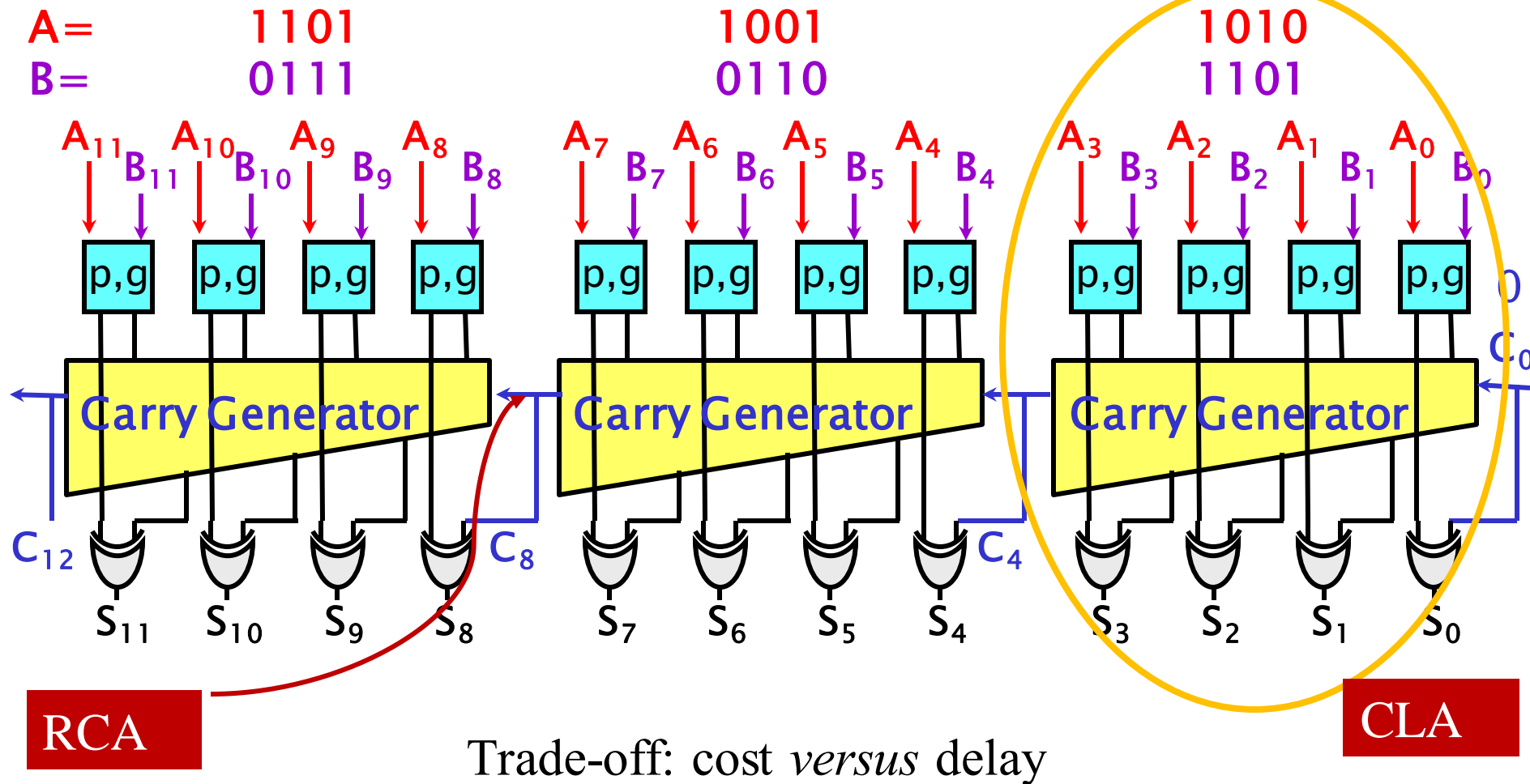
Theoretically, it is possible to derive each sum digit directly from the inputs that affect it

Full carry-lookahead adder is impractical for large  $n$

# Solution 1: $n$ -bit Hybrid CLA Adder

- Implementation of lookahead for the complete adder is impractical because of cost
  - Divide  $n$  stages into smaller groups
  - Full carry lookahead within each group
  - Ripple carry among groups
  - Compromise between cost and delay

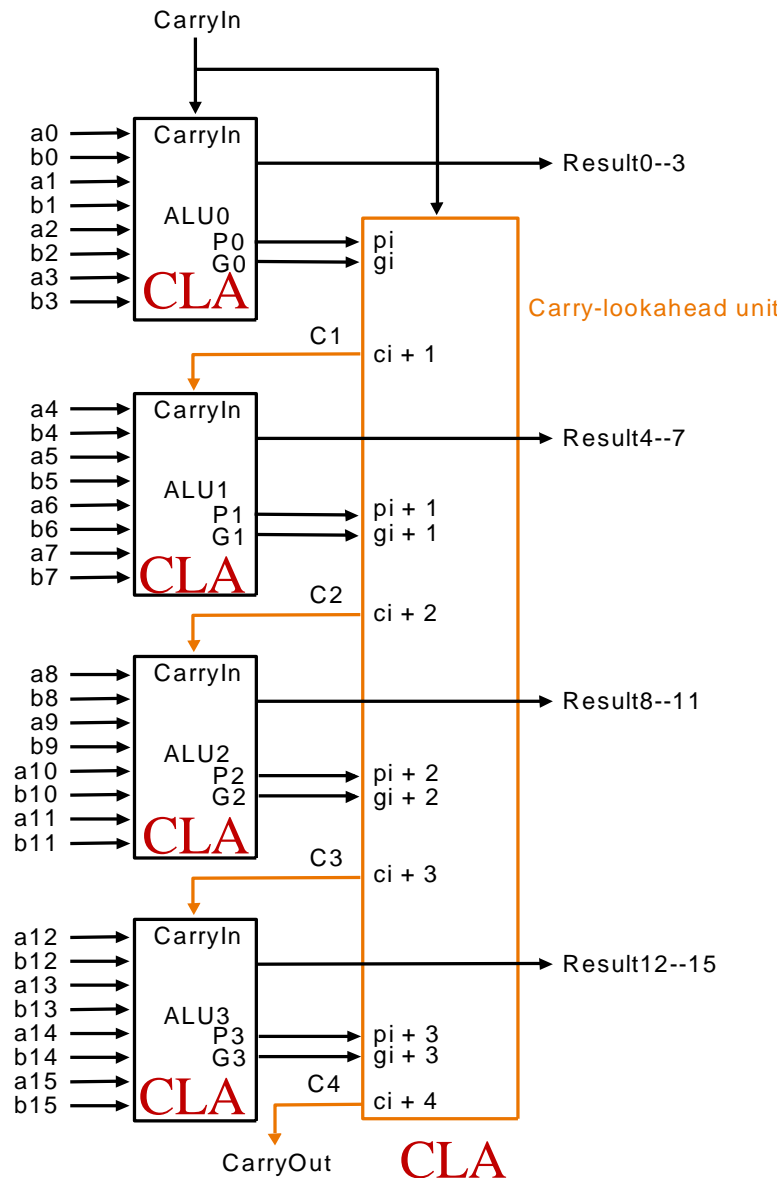
# 12-Bit Hybrid Adder (CLA + RCA)



**Another solution:** 12-Bit Hybrid (RCA + CLA)??

Cost =  $O(?)$ ; delay =  $O(?)$

CLA principle can be used recursively to build bigger adders  
 => Carry-Lookahead Tree (CLT)

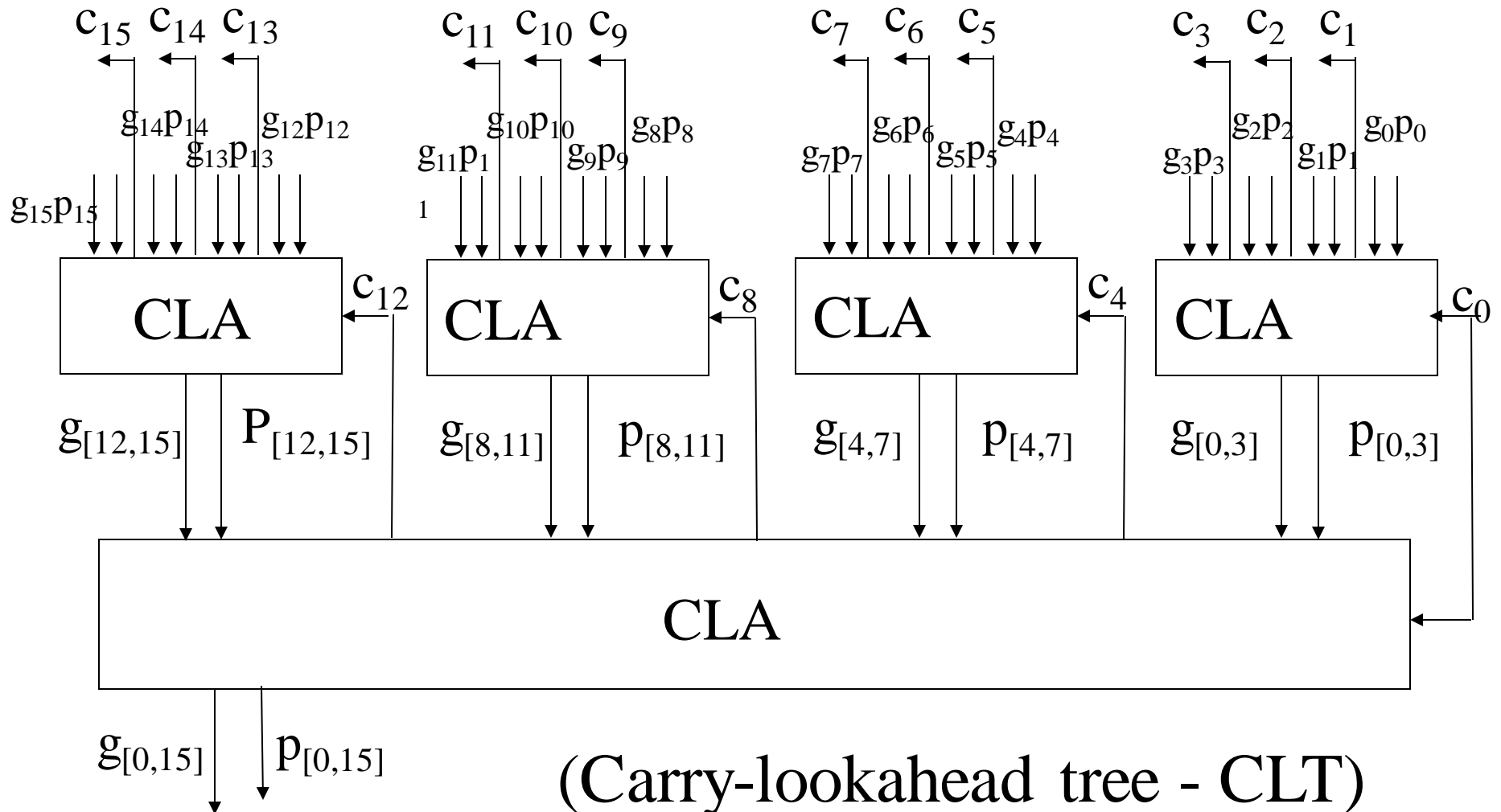


- Example: A **16-bit adder** - Use four 4-bit adders
- It takes block  $g$  and  $p$  terms and  $c_{in}$  to generate block-carry-bits
- Use the CLA principle again to design lower level adders

We have seen (i) CLA + RCA;  
 (ii) RCA + CLA

**Solution – 2**  
 (CLA + CLA + ...)

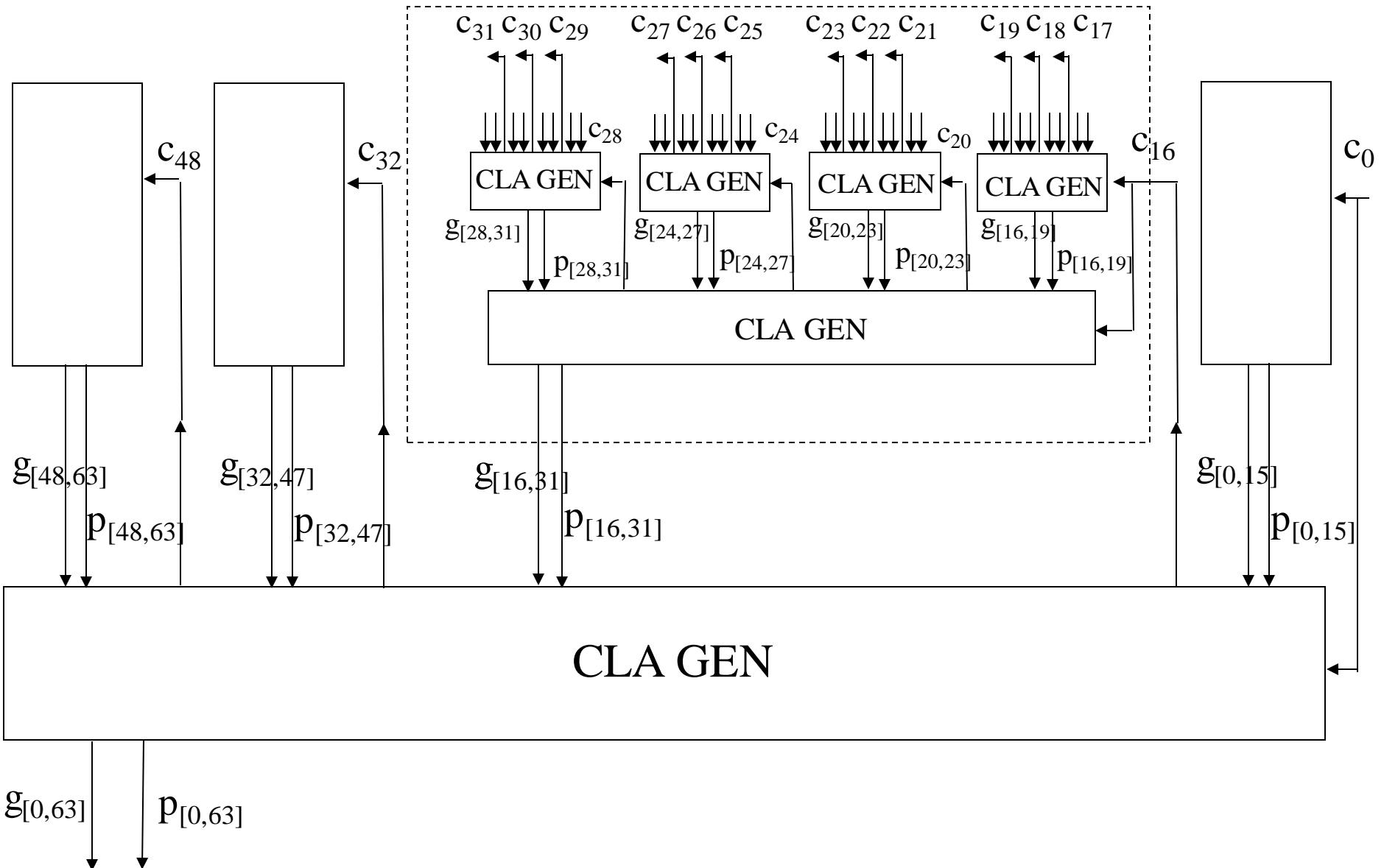
# 16-bit 2-level Carry-Lookahead Tree (CLT)



(Carry-lookahead tree - CLT)

Cost =  $O(?)$ ; delay  $O(?)$

# 64-bit 3-level Carry-Lookahead Tree (CLT)



## **Solution – 3: Carry-Select Adders (CSA)**

*Idea:*

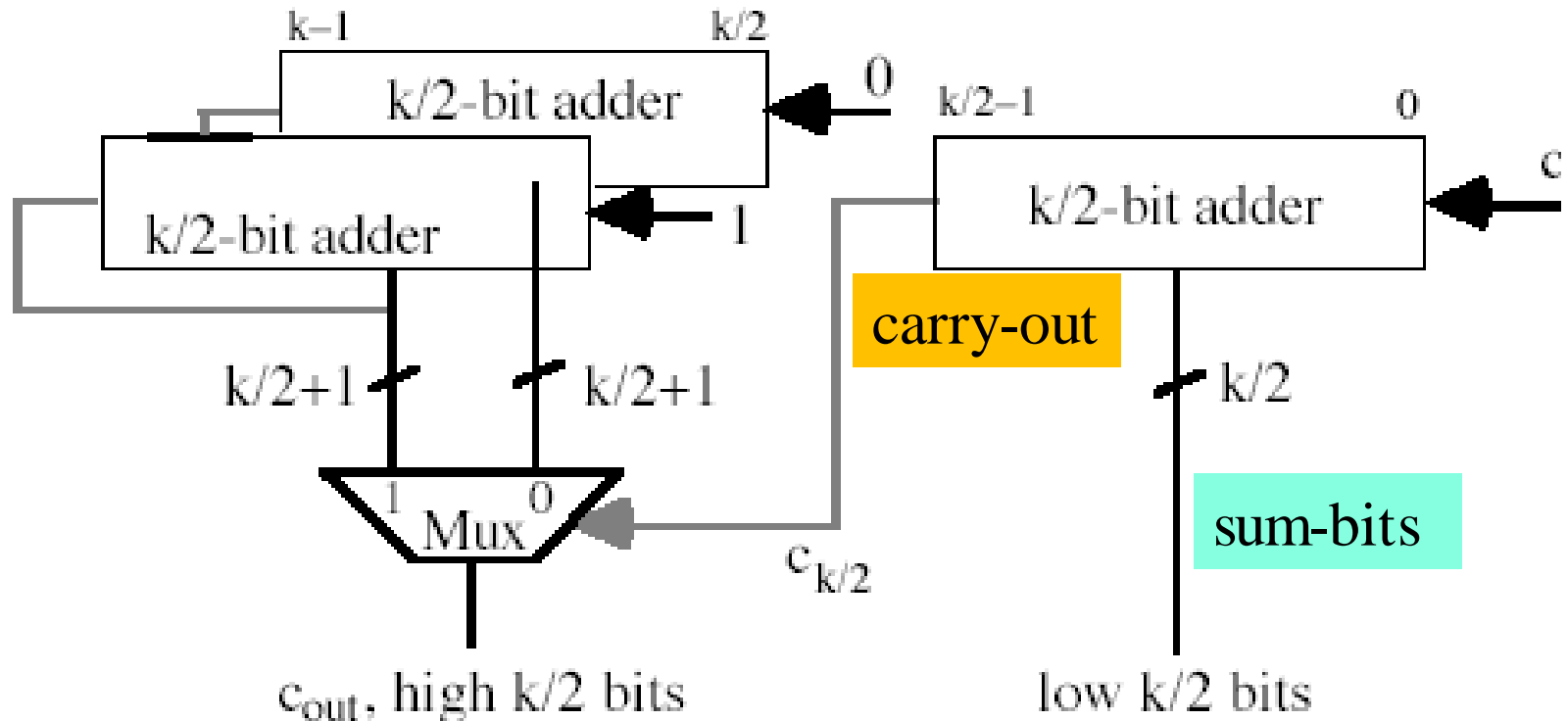
Compute sum-bits in parallel for two possible carry-bits;  
When the previous carry actually arrives, just select the pre-computed “sum”

CLA: Looking ahead in time (pre-compute carry-bits)

CSA: Looking ahead in space (pre-compute sum-bits)



# $k$ -bit Carry-Select Adder

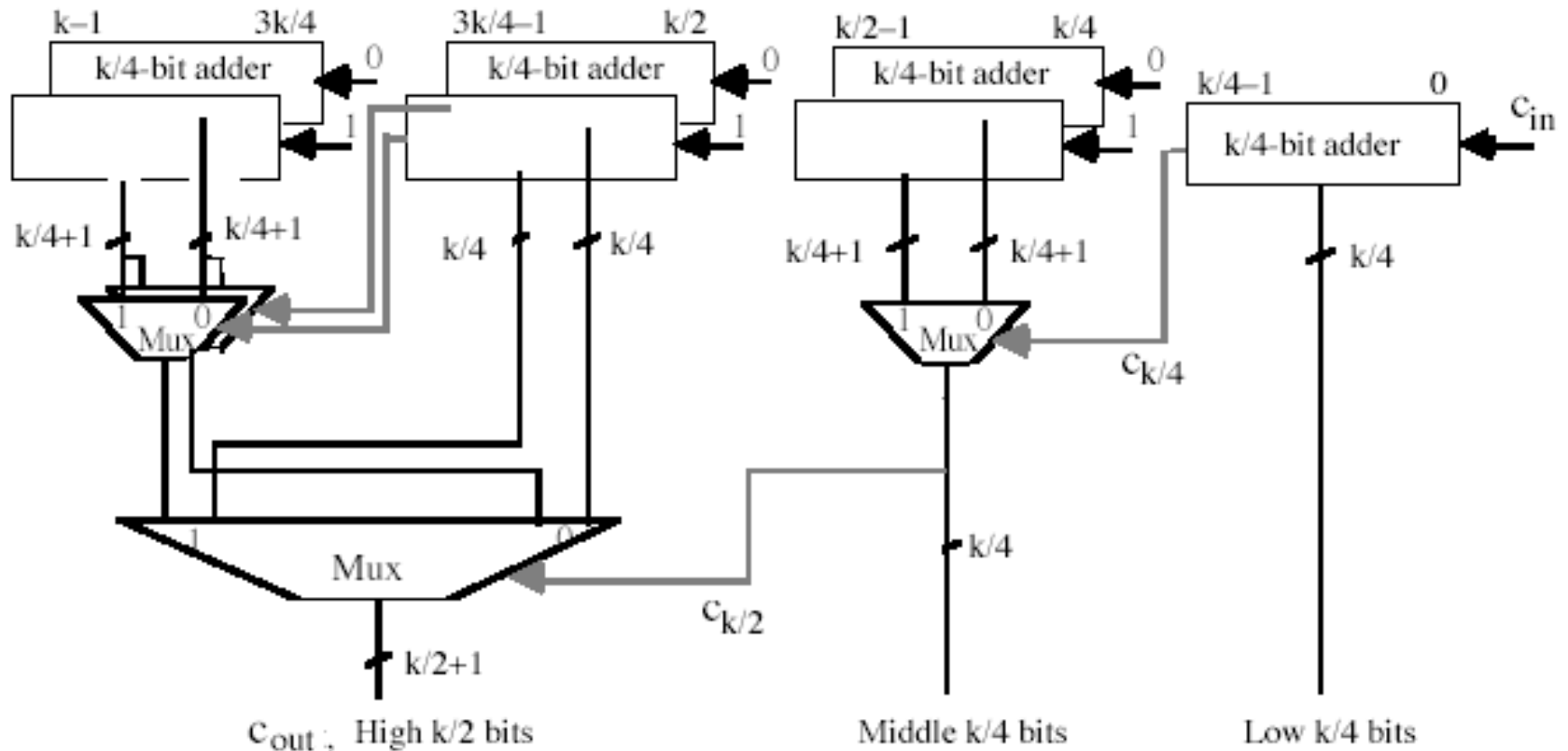


$$C_{\text{select-add}}(k) = 3C_{\text{add}}(k/2) + k/2 + 1$$

$$T_{\text{select-add}}(k) = T_{\text{add}}(k/2) + 1$$

Units: cost and delay of a single 2-to-1 multiplexer

# Multi-level $k$ -bit Carry-Select Adder



cost = ?  
delay = ?

Other adder designs:  
Carry-skip addition

# Pitfalls of CLA Adder

- Implementation of lookahead for the complete adder is impractical because of cost
- Analyze the implementation complexity of CLA

CS 31007

Autumn 2021

# COMPUTER ORGANIZATION AND ARCHITECTURE

---

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #19, #20: Computer Arithmetic

14 September 2021

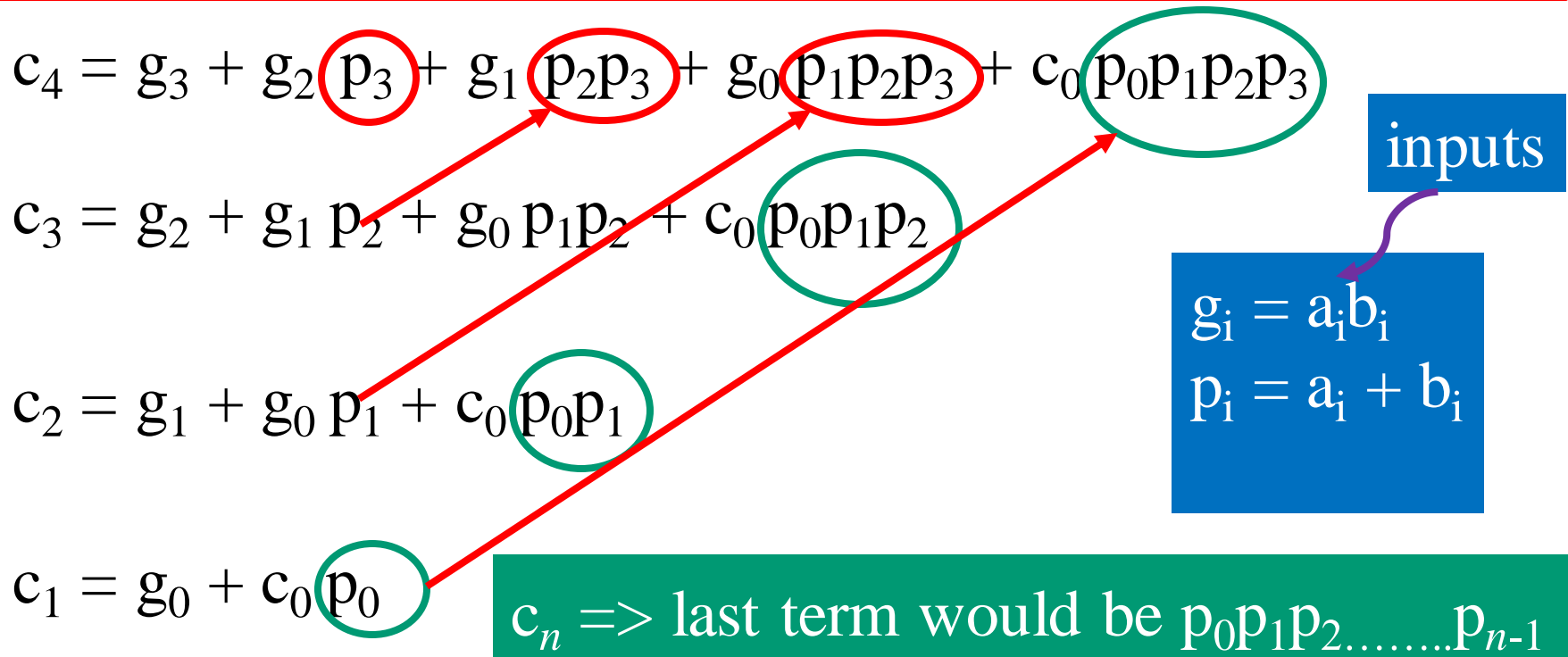
---

Indian Institute of Technology Kharagpur  
*Computer Science and Engineering*

# Solution – 4: Tree-Based Adder

Recall: 4-bit CLA

In CLA, carry-logic cost grows @  $O(n^3)$  for  $n$ -input adders



In general, the problem boils down to  $\Rightarrow$

$$c_{i+1} = g_i + c_i p_i$$

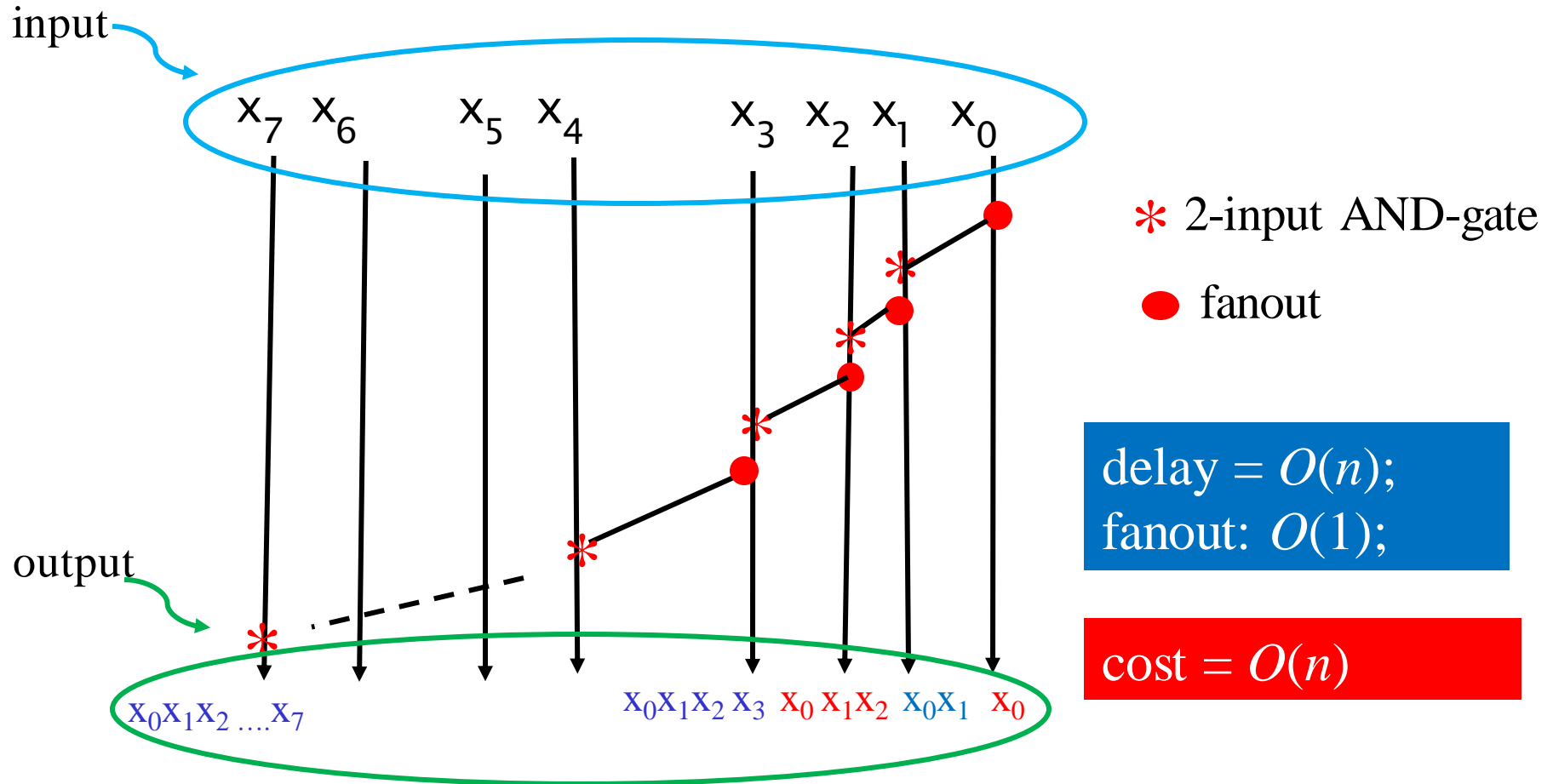
How to compute efficiently in reasonable cost and time:

$$x_0, x_0 x_1, x_0 x_1 x_2, x_0 x_1 x_2 x_3, \dots, x_0 x_1 x_2 x_3 \dots x_{n-1}$$

(Prefix-sum/product problem)

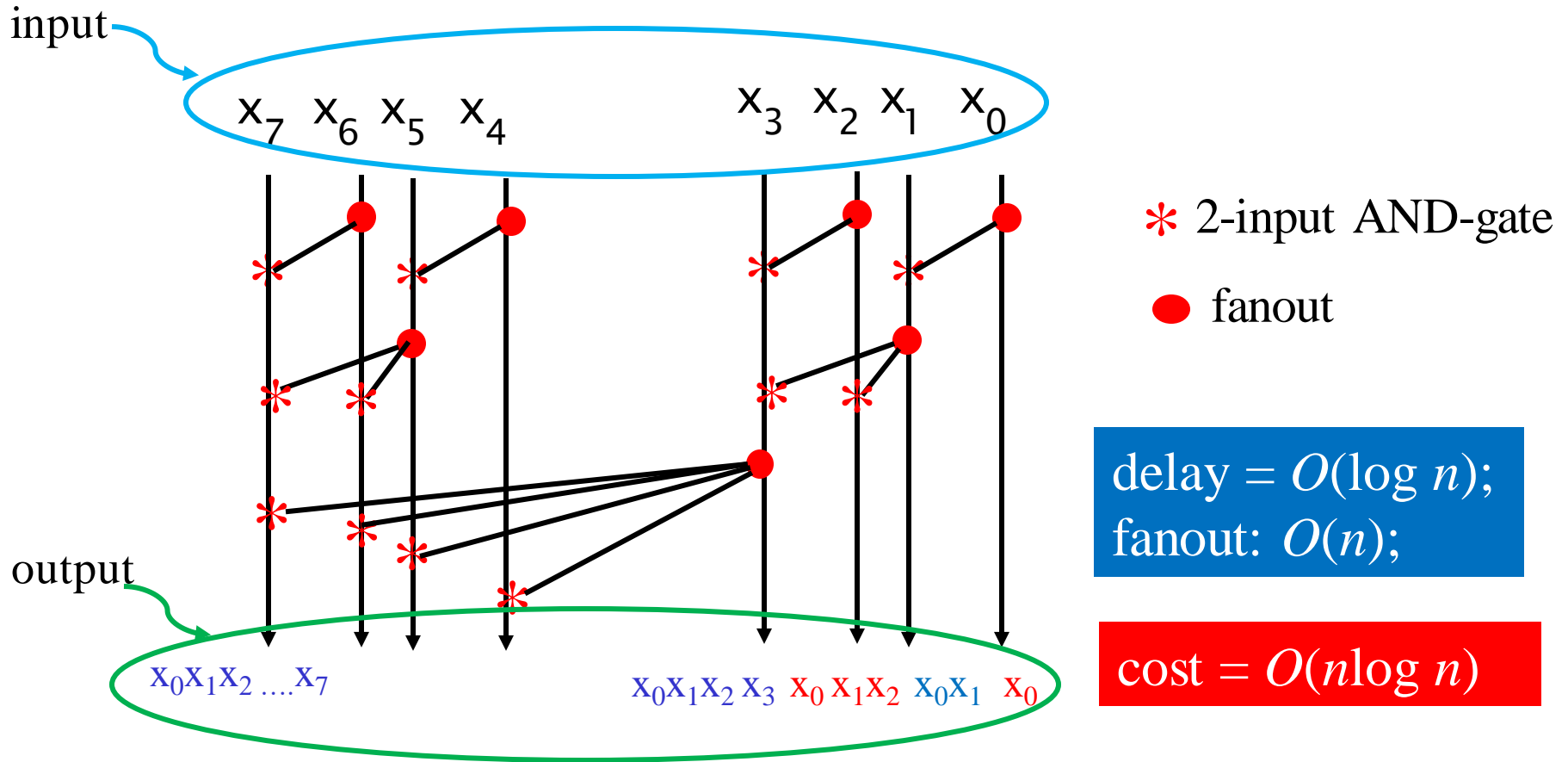
# Why so fuss about prefix computation?

Use a serial chain .... .. done!



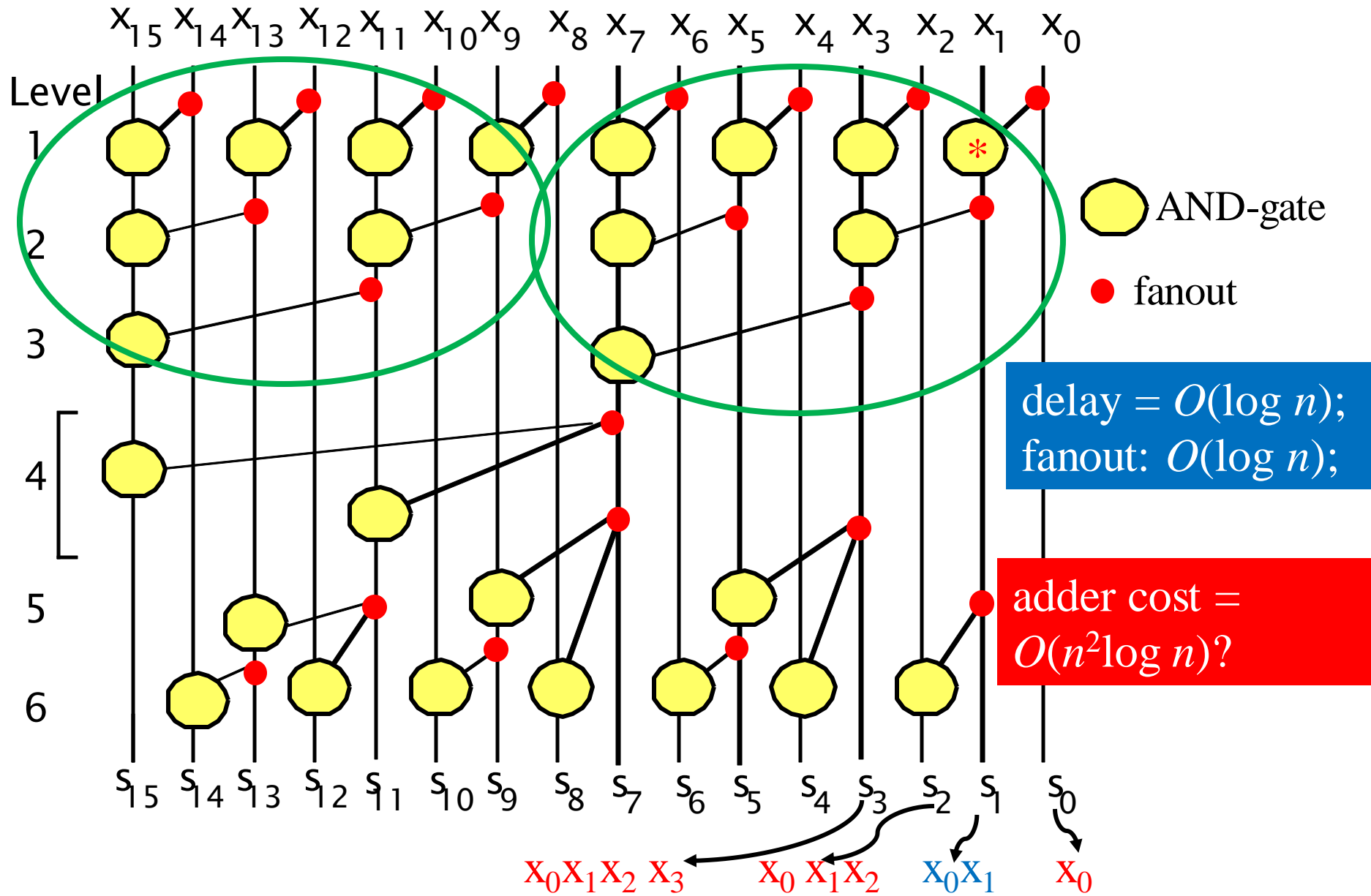
However, its delay is again  $O(n)$ , so the whole purpose of CLA design is defeated. Can we reduce delay, compromising cost and fanout?

# Prefix tree: Basic idea



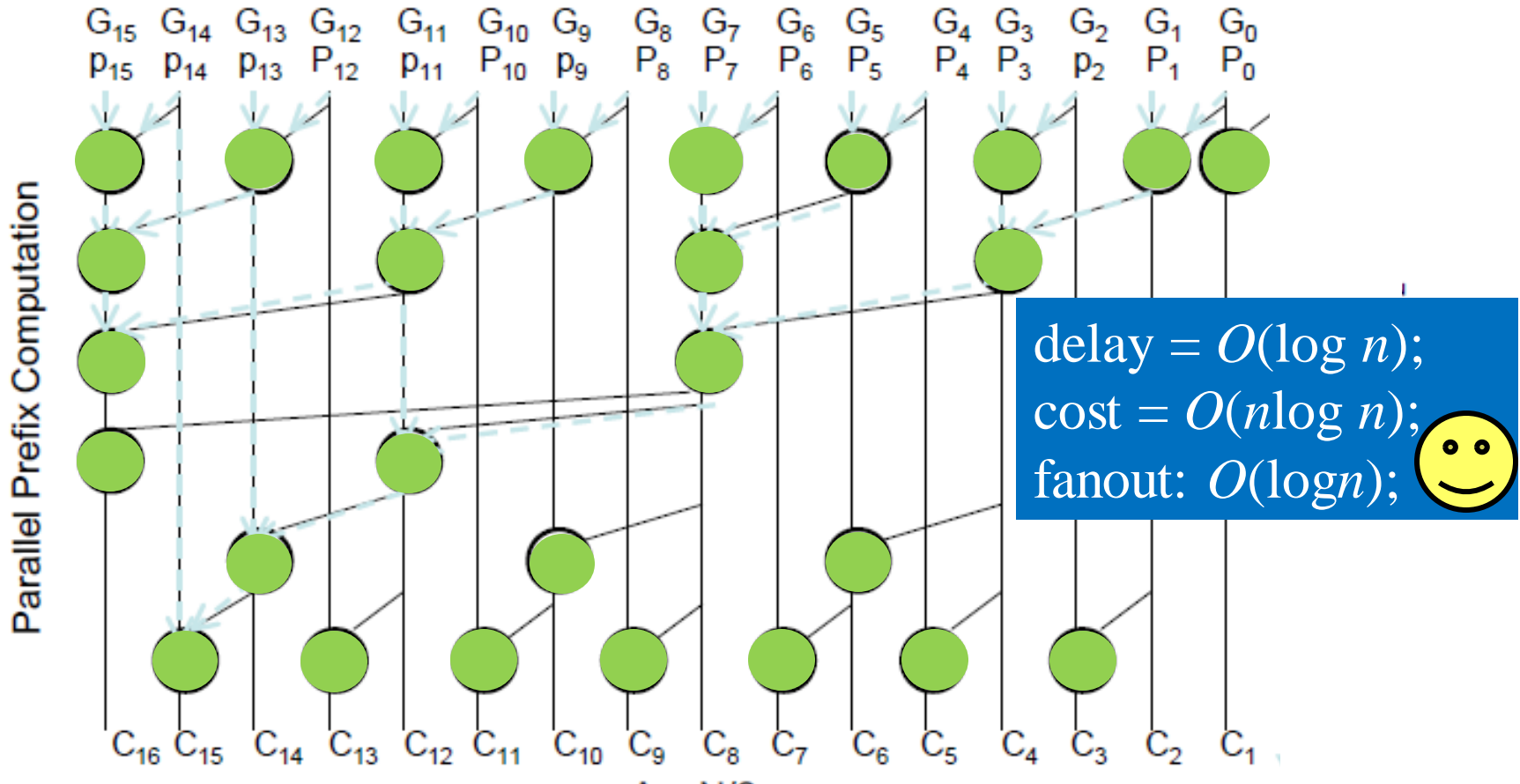
Large fanout slows down  $0 \rightarrow 1$  or  $1 \rightarrow 0$  transitions at logic nodes and thus may increase consumed power and signal delay indirectly

# Prefix Tree: Logarithmic-Delay





# Brent-Kung Parallel-Prefix Adder (log-delay)

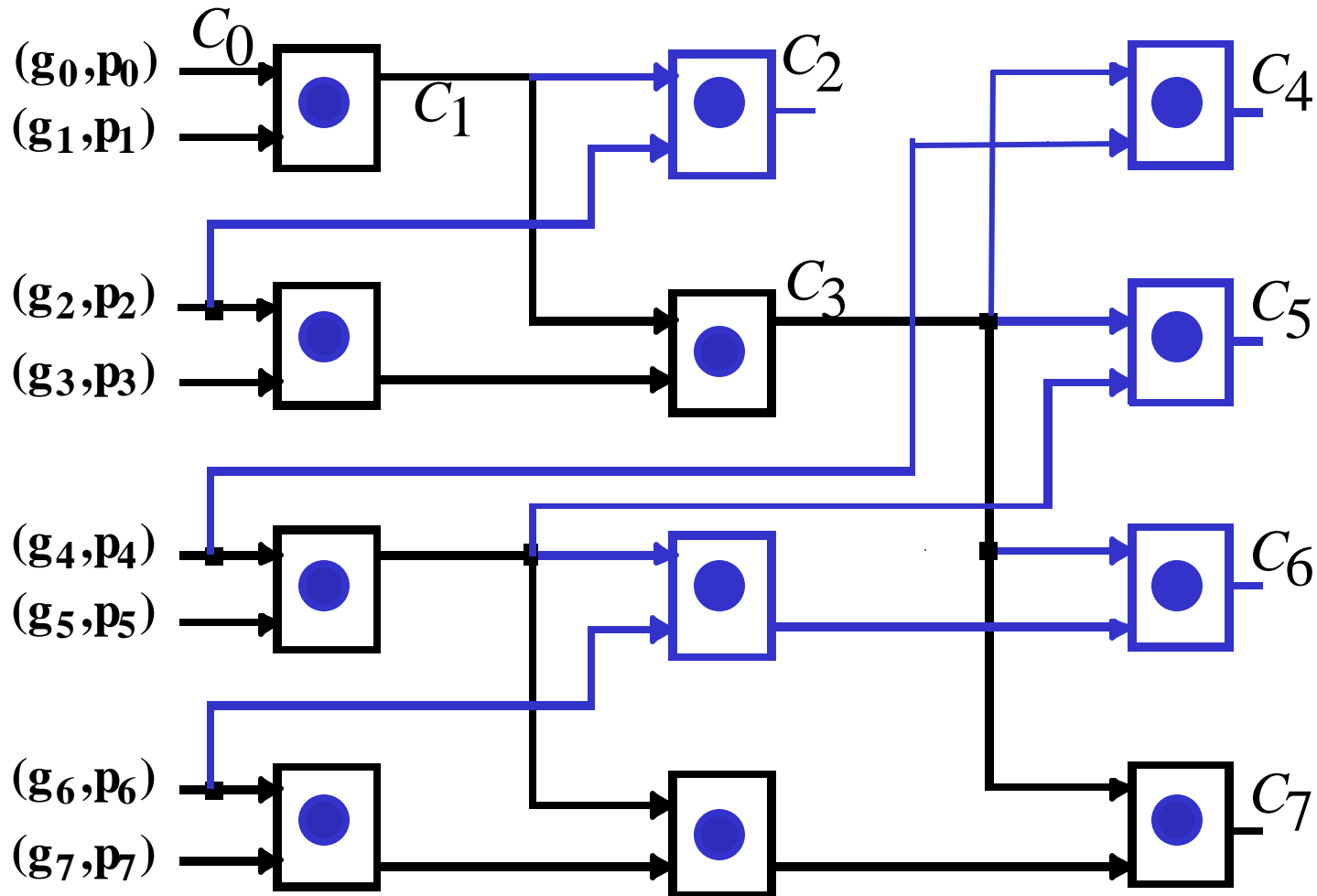


cost can further  
be improved

●  $c_{i+1} = g_i + c_i p_i$

~~cost =  $O(n^2 \log n)$ ?~~

# Brent-Kung: Carry-Generation Tree for 8-Bit Adder

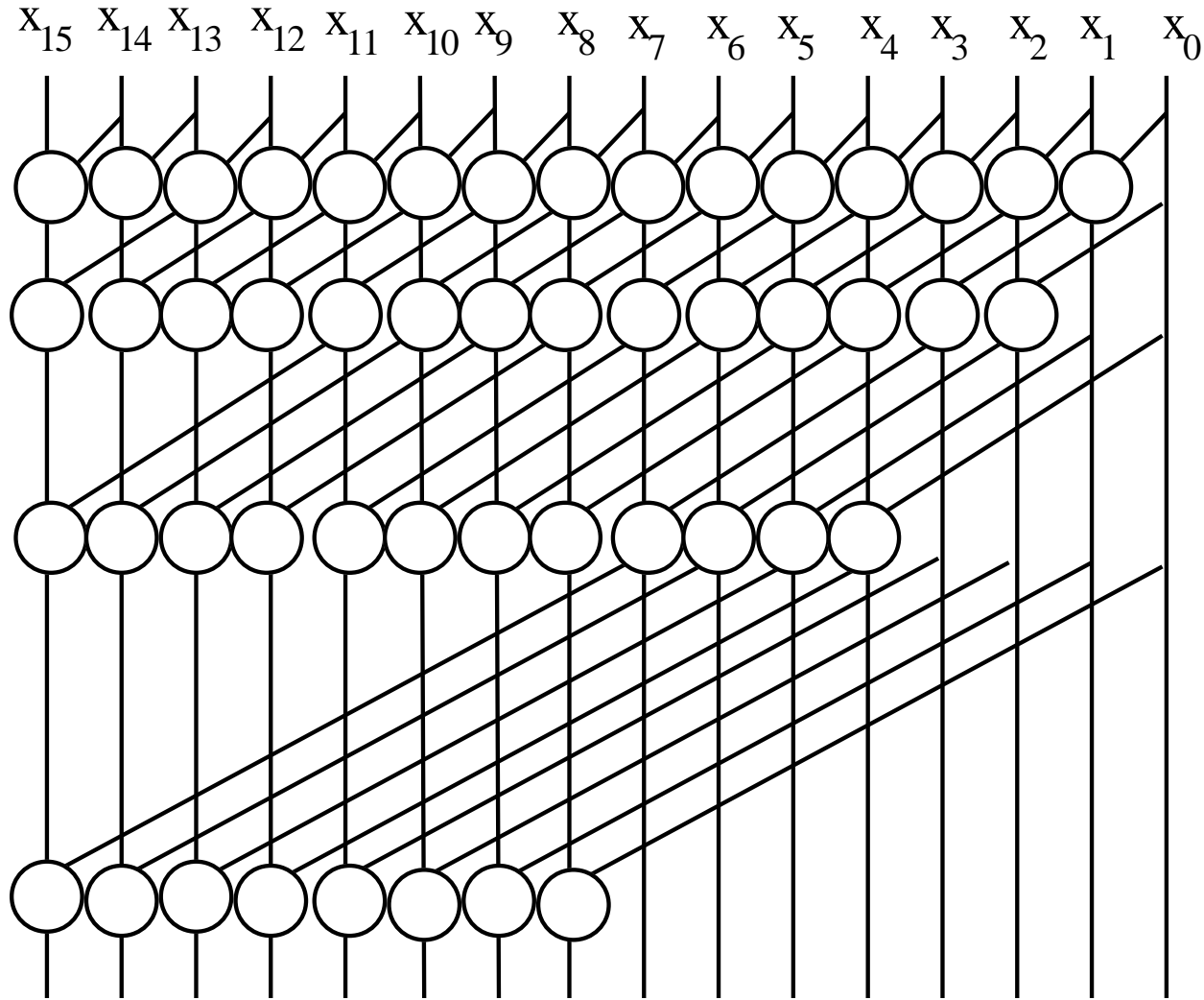


cost  $\sim O(n \log n)$

delay  $\sim O(\log n)$

# Other Tree-Based Prefix Adders

## Kogge-Stone Adder (16-Bit)

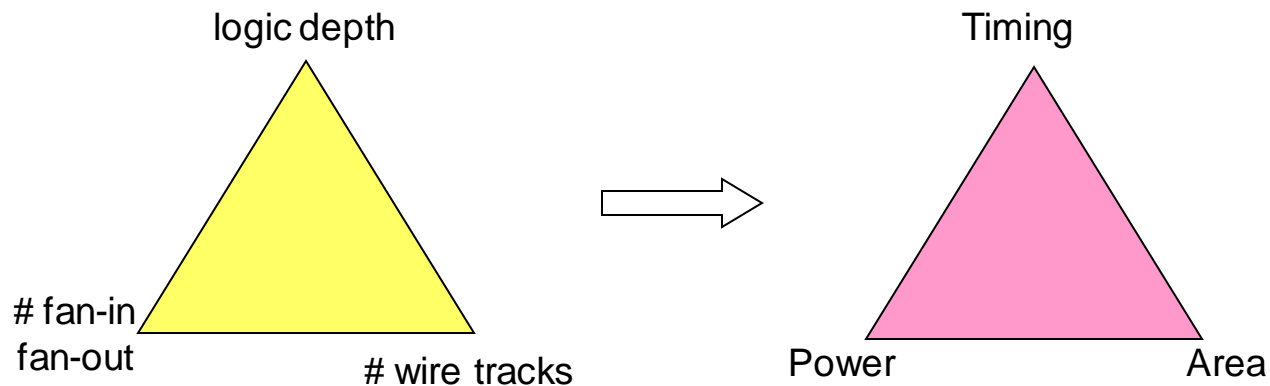


- fanout: constant
- delay =  $O(\log n)$ ;
- cost =  $O(n \log n)$ ;
- high wiring complexity

# Adder Design

## Design considerations

- Timing (delay)
- Power/energy, heat dissipation
  - fan-in, fan-out, logic switching
- Area (cost)
  - logic, wiring



# Adding multiple numbers

**ANANDHA BHAVAN A/C**  
13 A ECR MAIN ROAD PILLAIYARKUPPAM  
GSTIN 34AAYFA4506B1Z0  
WISH YOU ALL A VERY HAPPY NEW YEAR

**BILL NO : 78060**  
**DATE : 26/01/2018** **TIME : 09:53 AM**  
**WAITER : SIVA.A** **TABLE : 8**

| Name               | Rate  | Qty | Amount |
|--------------------|-------|-----|--------|
| GHEE PONGAL        | 50.00 | 1   | 50.00  |
| VADAI              | 20.00 | 3   | 60.00  |
| ROAST              | 50.00 | 3   | 150.00 |
| POORI MASAL        | 50.00 | 2   | 100.00 |
| TEA                | 25.00 | 1   | 25.00  |
| Qty : 10           |       |     | 385.00 |
| 2.5% CGST Amount : |       |     | 9.63   |
| 2.5% SGST Amount : |       |     | 9.63   |
| Round Amount :     |       |     | -0.26  |
|                    |       |     | 404.00 |

**CASH RECEIVED**

**NON A/C**

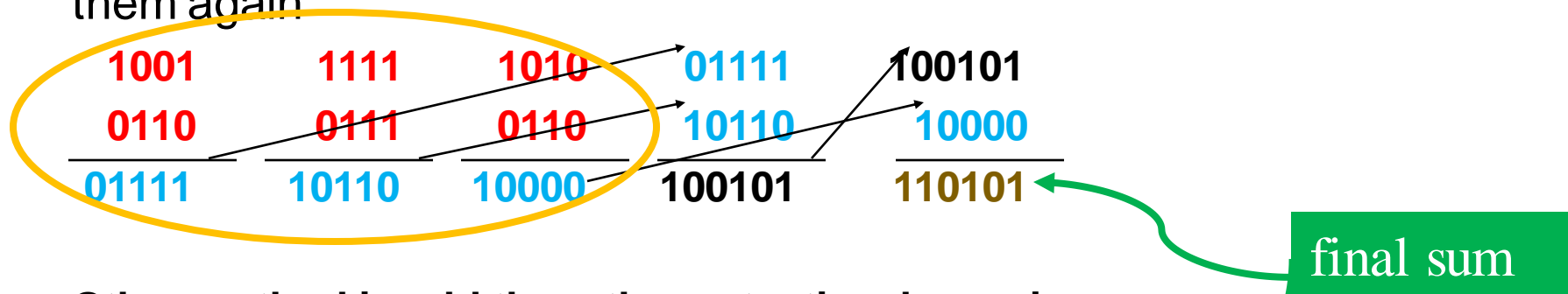
**Rs. 404.00**  
THANK YOU...VISIT AGAIN



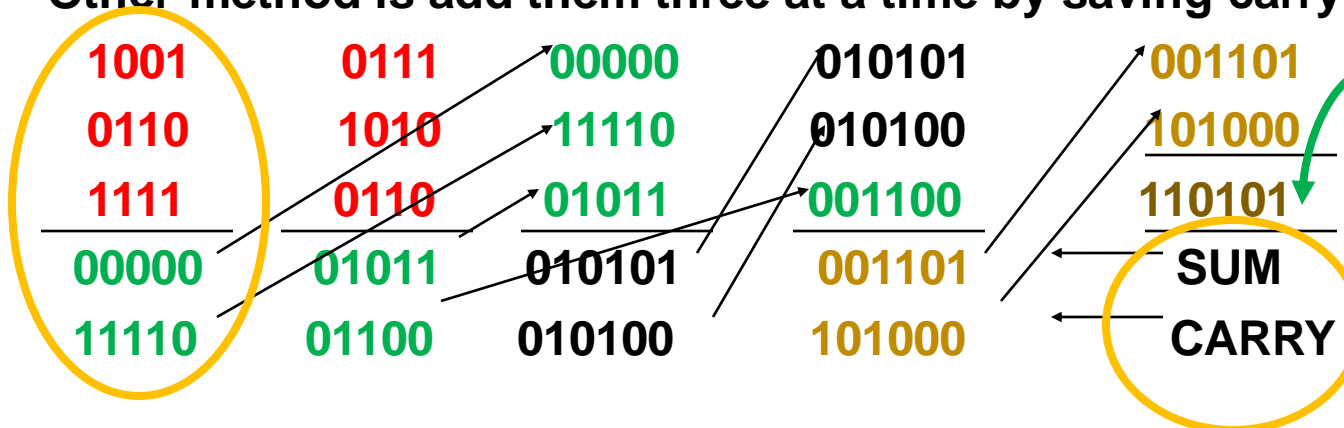
can we do better?

# Carry-Save Addition: Adding multiple operands

- Consider adding six numbers (4 bits each)
- 1001, 0110, 1111, 0111, 1010, 0110 (all unsigned +ve)
- One way is to add them pair wise, getting three results, and then adding them again

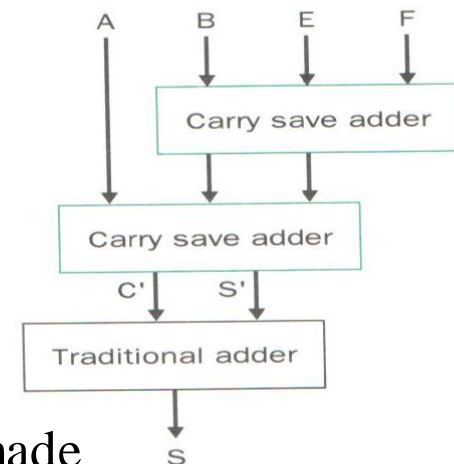
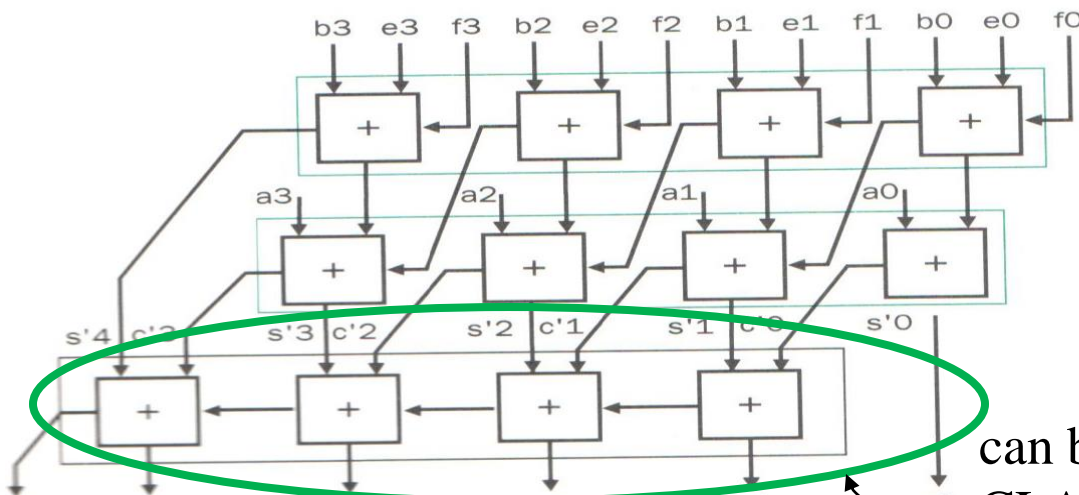
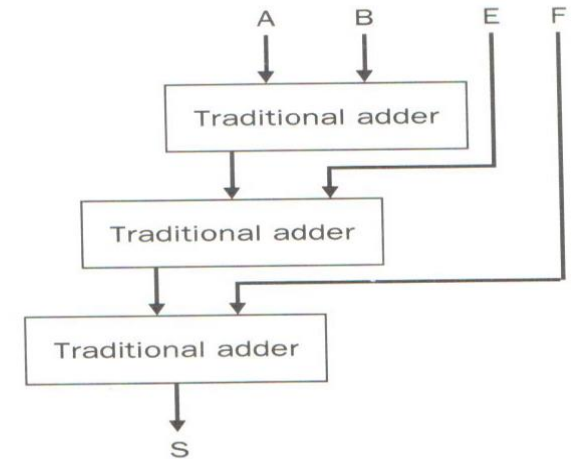
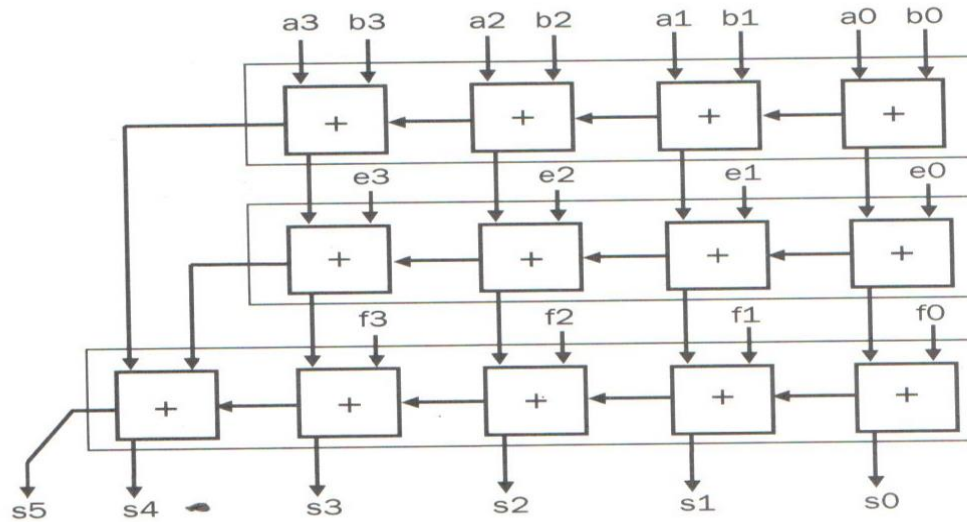


- Other method is add them three at a time by saving carry



# CARRY-SAVE ADDER (addition of multiple operands)

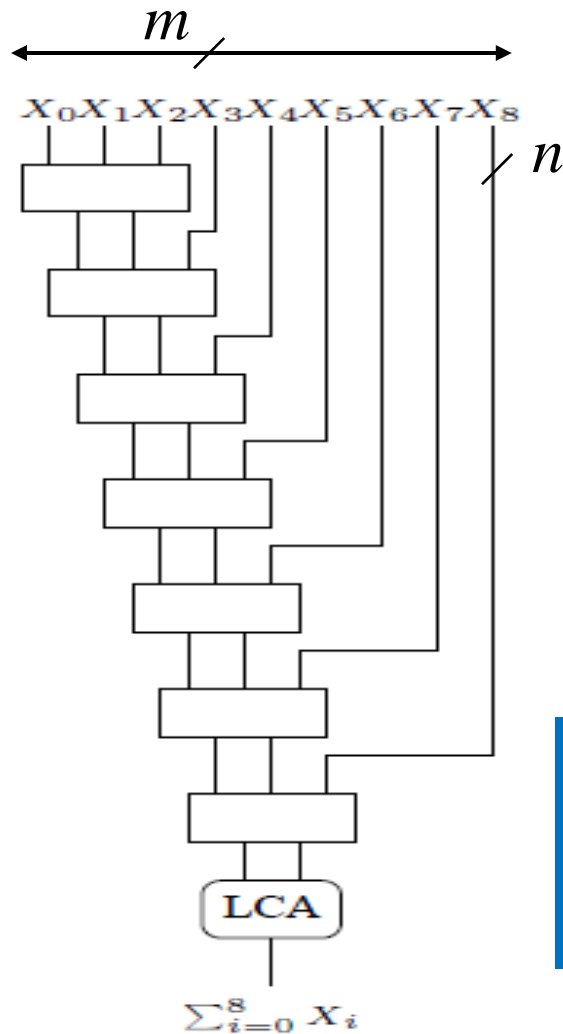
Add four 4-bit integers, A, B, E, F



Cost =  $O(?)$ ; delay  $O(?)$

can be made  
CLA

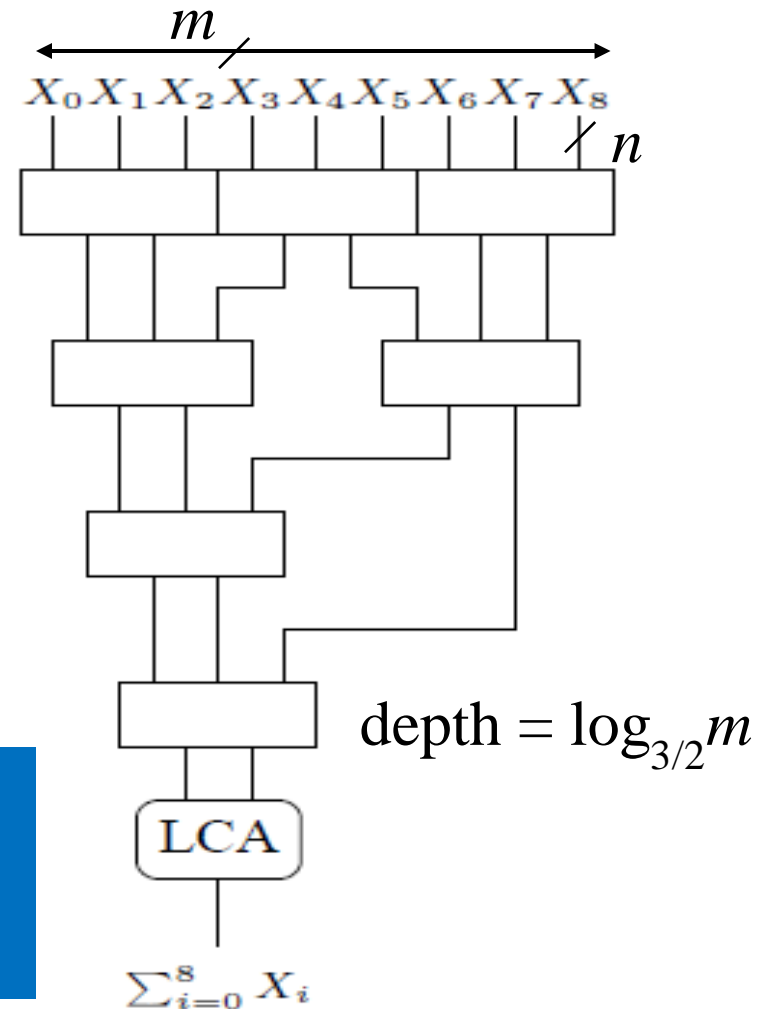
# Adding $m$ , $n$ -bit numbers with CSA and log-adder



LCA: Lookahead  
carry adder with  
log-delay ( $\sim$ CLT)

Linear chain;

$$\text{Delay} = O(m + \lg(n + m))$$



Wallace tree;

$$\text{Delay} = O(\log m + \lg(n + \log m))$$



# Integer Multiplication



# Multiplication

- Multiplicand
- Multiplier
- Partial products
- Final sum

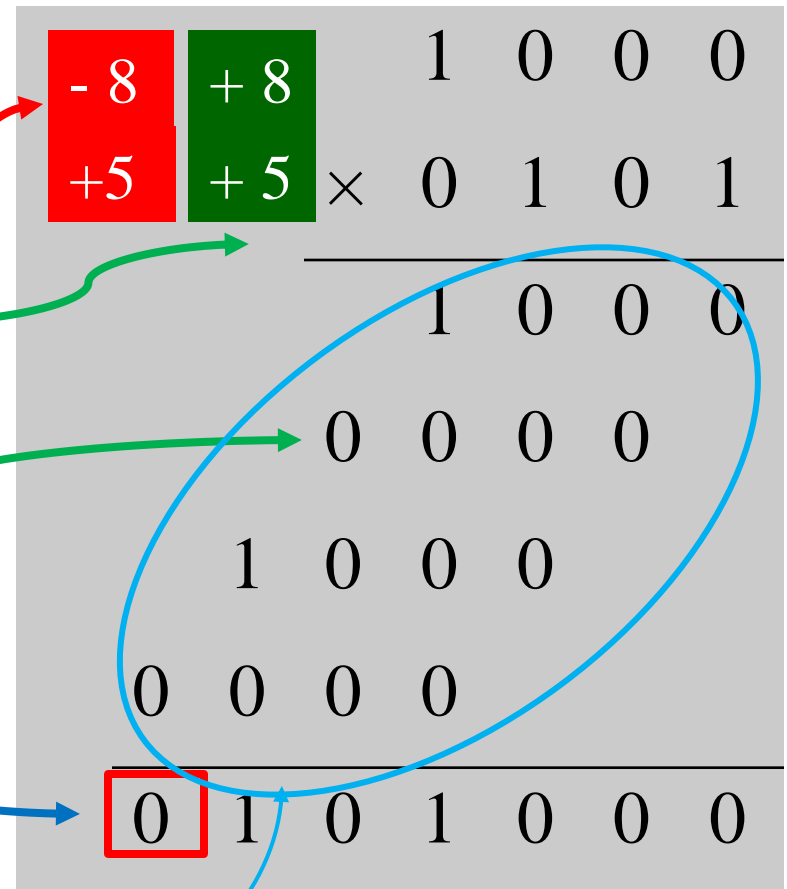
- Base 10:  $8 \times 5 = 40$

$$\Rightarrow 32 + 0 + 8 = 40$$

- How wide is the result?

$$\Rightarrow \log(n \times m) = \log n + \log m$$

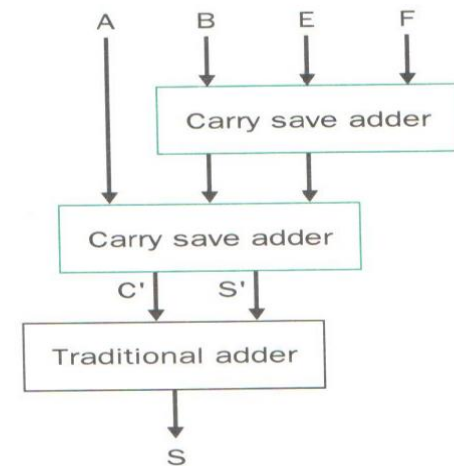
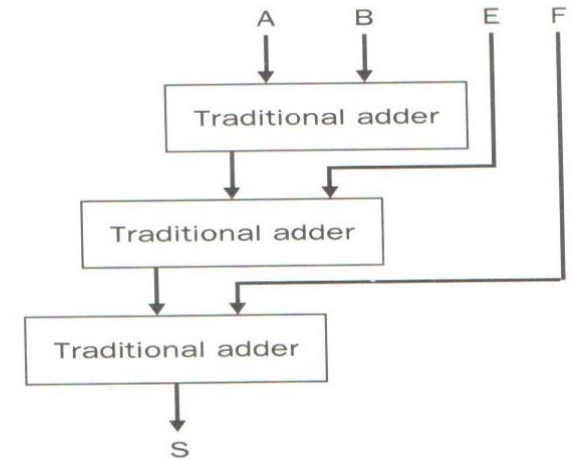
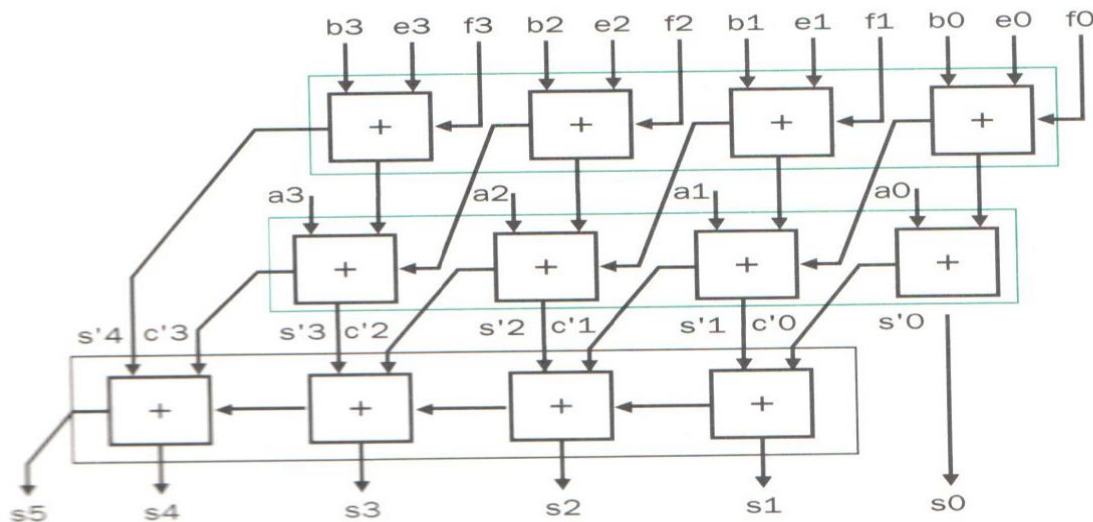
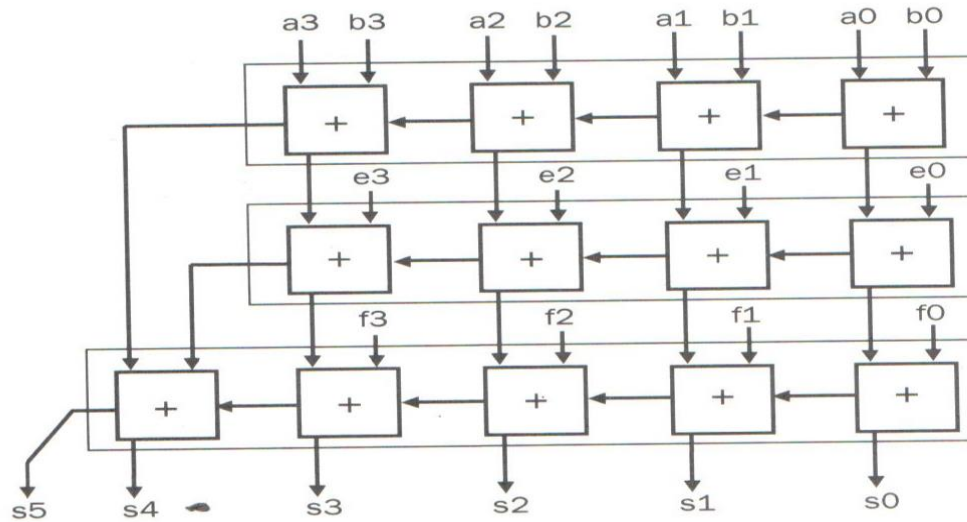
$$\Rightarrow 32\text{-bit} \times 32\text{-bit} = 64\text{-bit result}$$



Can you notice the need for Carry-Save Addition? We need to add multiple numbers!

# CARRY-SAVE ADDER (addition of multiple operands)

Add four 4-bit integers, A, B, E, F



CS 31007

Autumn 2021

# COMPUTER ORGANIZATION AND ARCHITECTURE

---

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #21: Computer Arithmetic

16 September 2021

---

Indian Institute of Technology Kharagpur  
*Computer Science and Engineering*

$$\begin{array}{r}
 \begin{array}{cccc}
 & A_3 & A_2 & A_1 & A_0 \\
 \times & B_3 & B_2 & B_1 & B_0
 \end{array} \\
 \hline
 \begin{array}{cccc}
 AB_i \text{ called a "partial product"} \longrightarrow & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
 + & & & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3
 \end{array} \\
 \hline
 \underbrace{\hspace{15em}}
 \end{array}$$

Multiplying N-bit number by M-bit number gives (N+M)-bit result

$A_i$  and  $B_i$  are all 0 or 1

# Combinational Multiplier (unsigned)

|   |    |    |    |    |                |
|---|----|----|----|----|----------------|
|   | X3 | X2 | X1 | X0 | ← multiplicand |
| * | Y3 | Y2 | Y1 | Y0 | ← multiplier   |

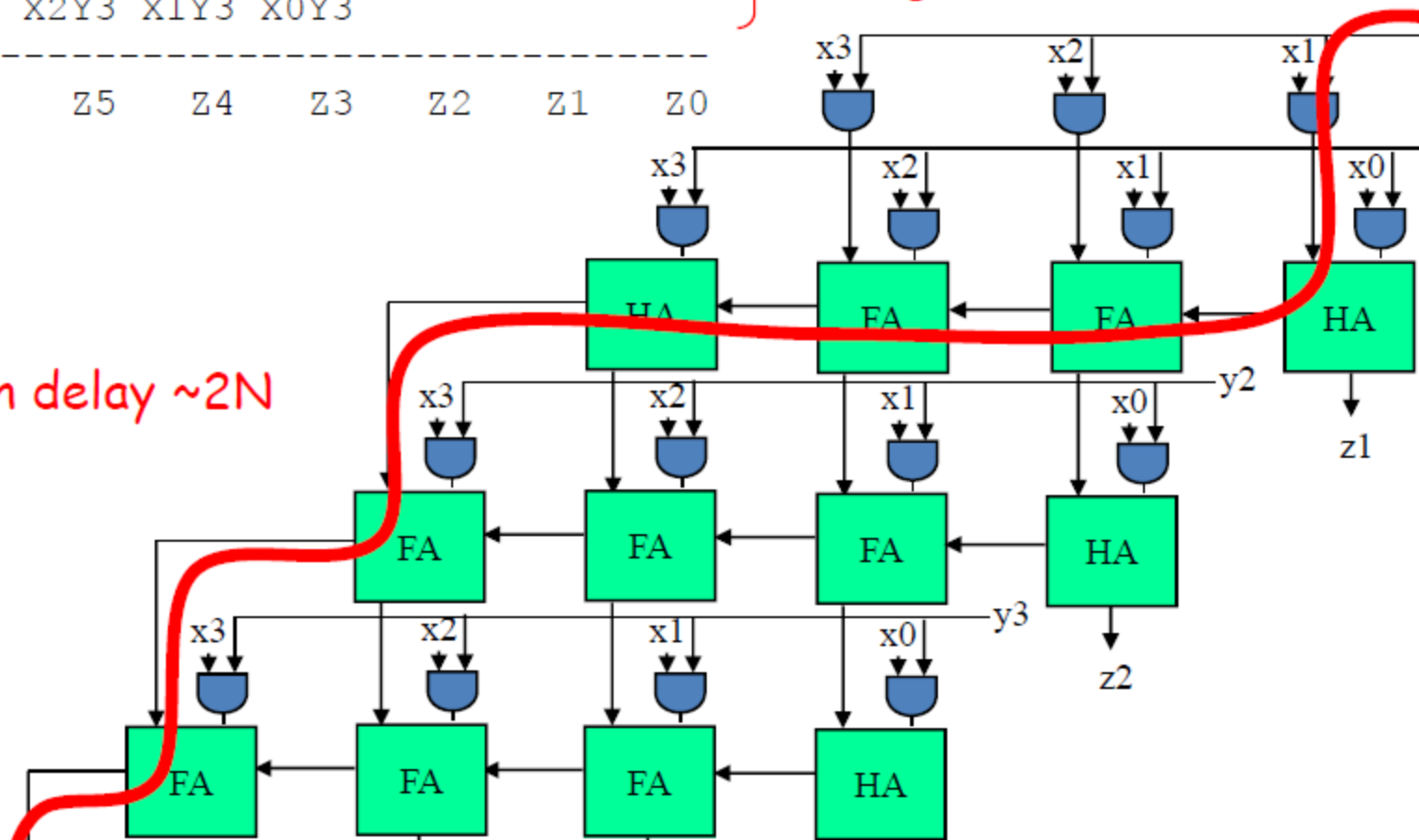
|   |      |      |      |      |      |      |  |
|---|------|------|------|------|------|------|--|
|   |      |      | X3Y0 | X2Y0 | X1Y0 | X0Y0 |  |
| + |      |      | X3Y1 | X2Y1 | X1Y1 | X0Y1 |  |
| + |      | X3Y2 | X2Y2 | X1Y2 | X0Y2 |      |  |
| + | X3Y3 | X2Y3 | X1Y3 | X0Y3 |      |      |  |

-----

z7    z6    z5    z4    z3    z2    z1    z0

Partial products, one for each multiplier (each bit needs just AND gate)

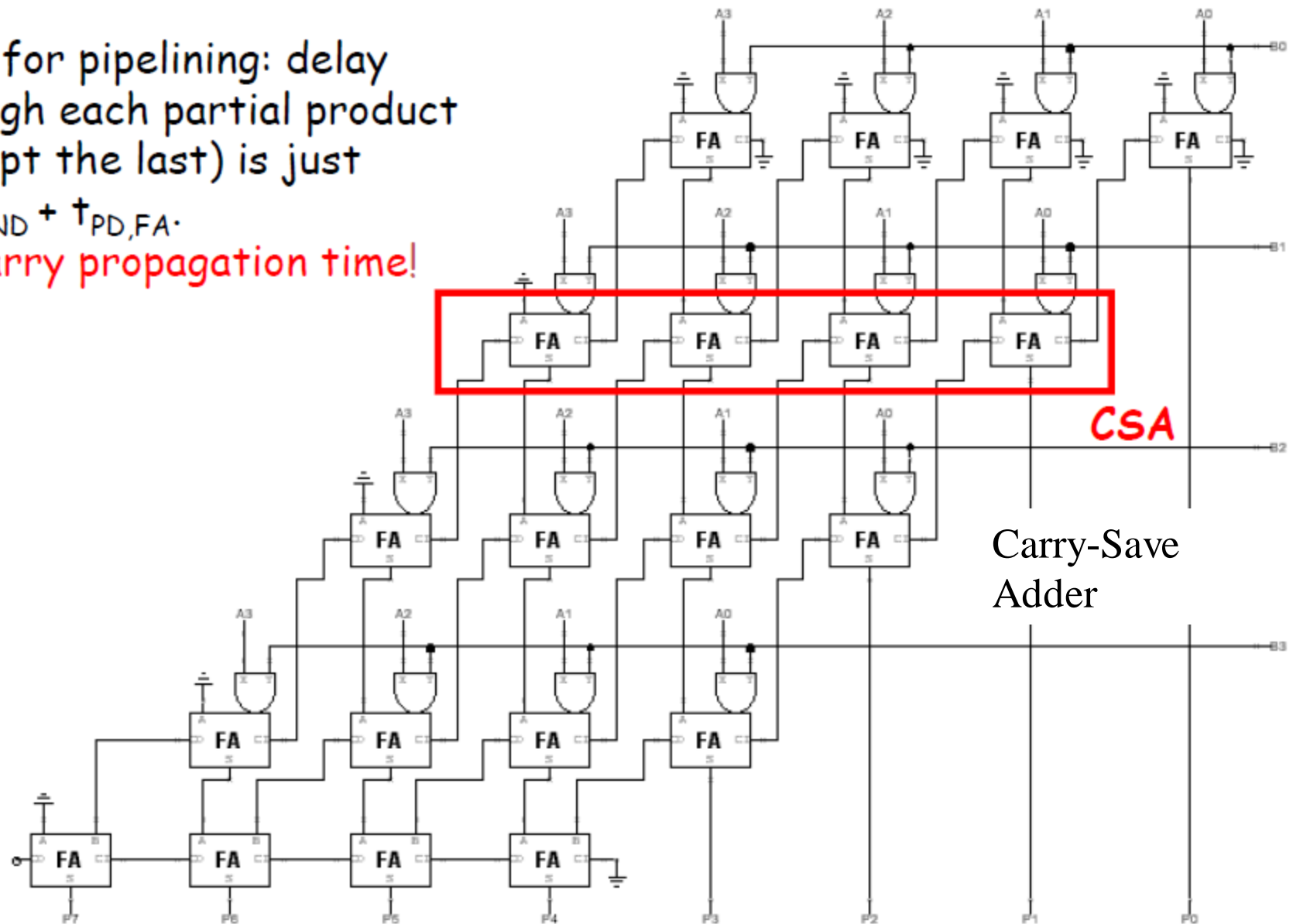
➤ Propagation delay  $\sim 2N$

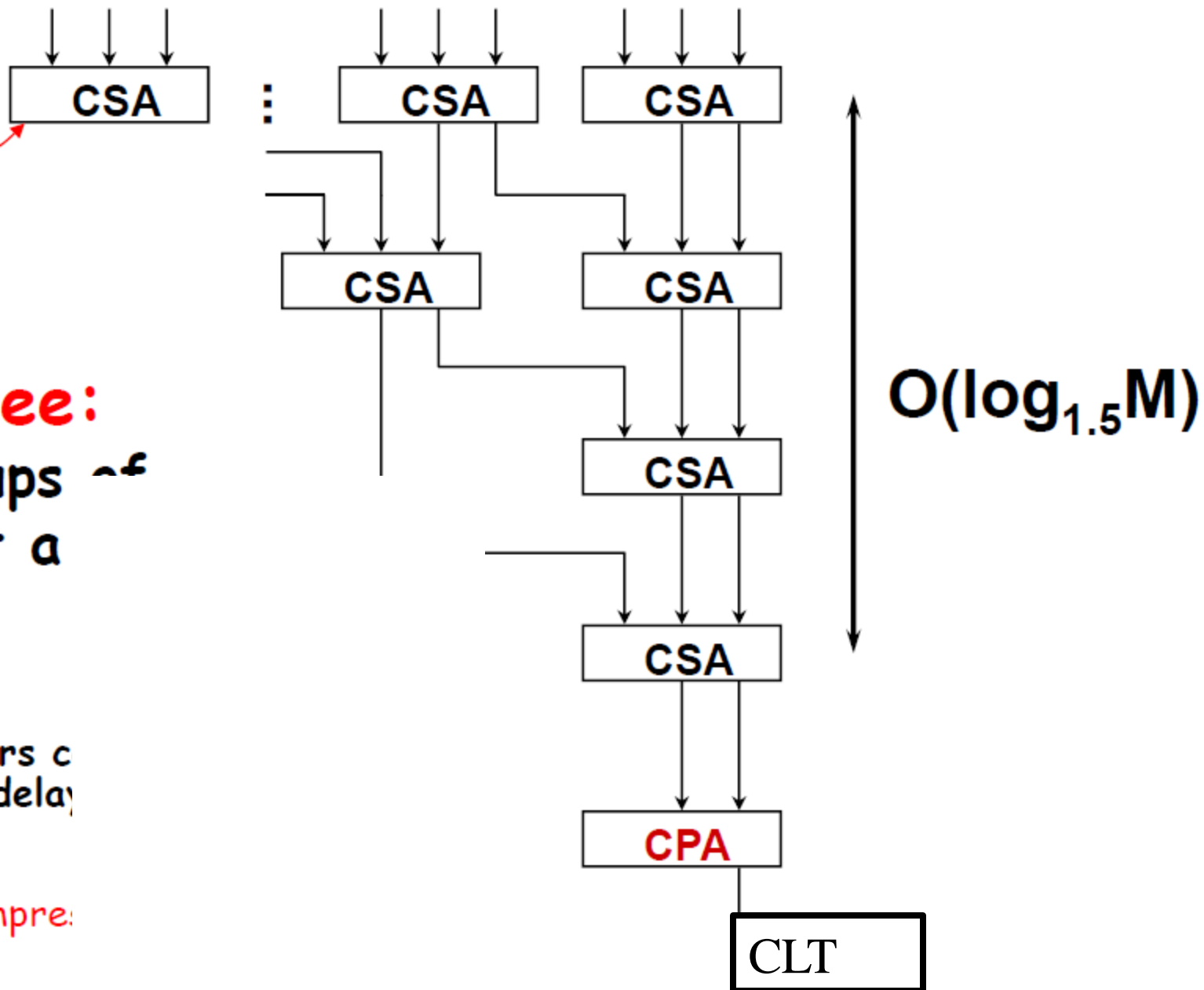


Good for pipelining: delay through each partial product (except the last) is just

$$t_{PD,AND} + t_{PD,FA}$$

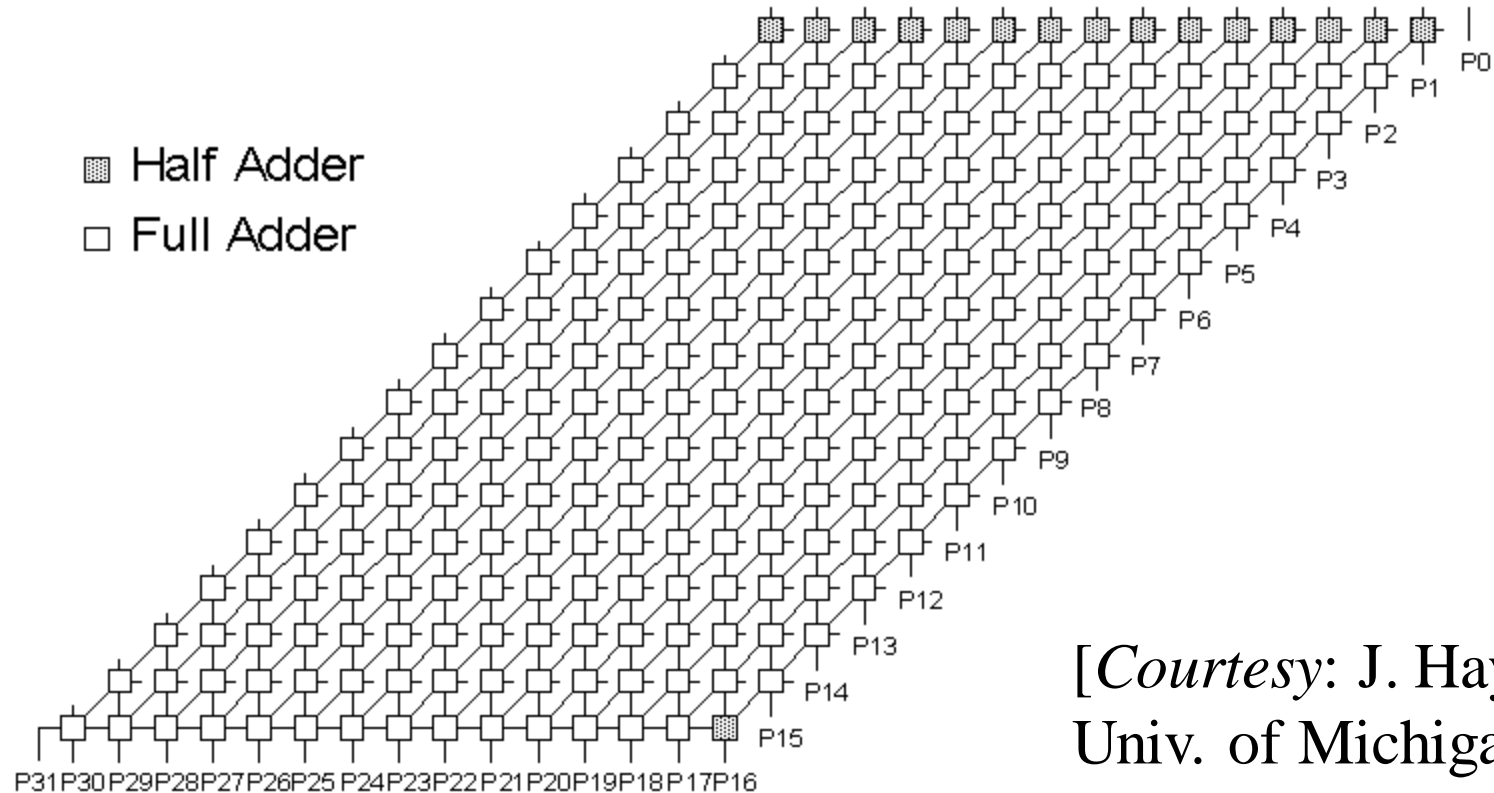
No carry propagation time!







# 16-bit Array Multiplier using Carry Sav-Adder

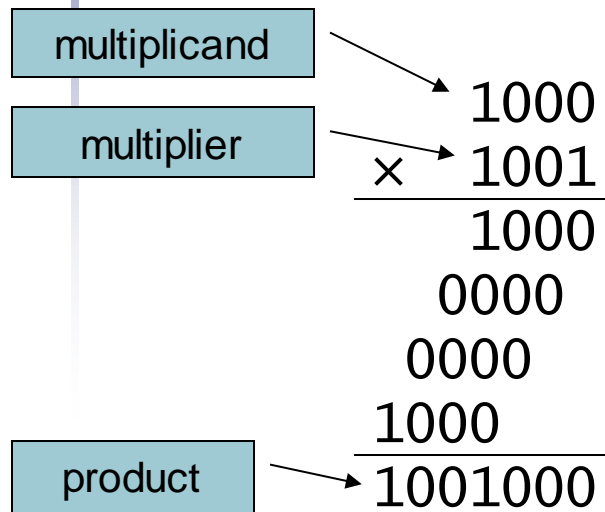


[*Courtesy: J. Hayes,  
Univ. of Michigan*]

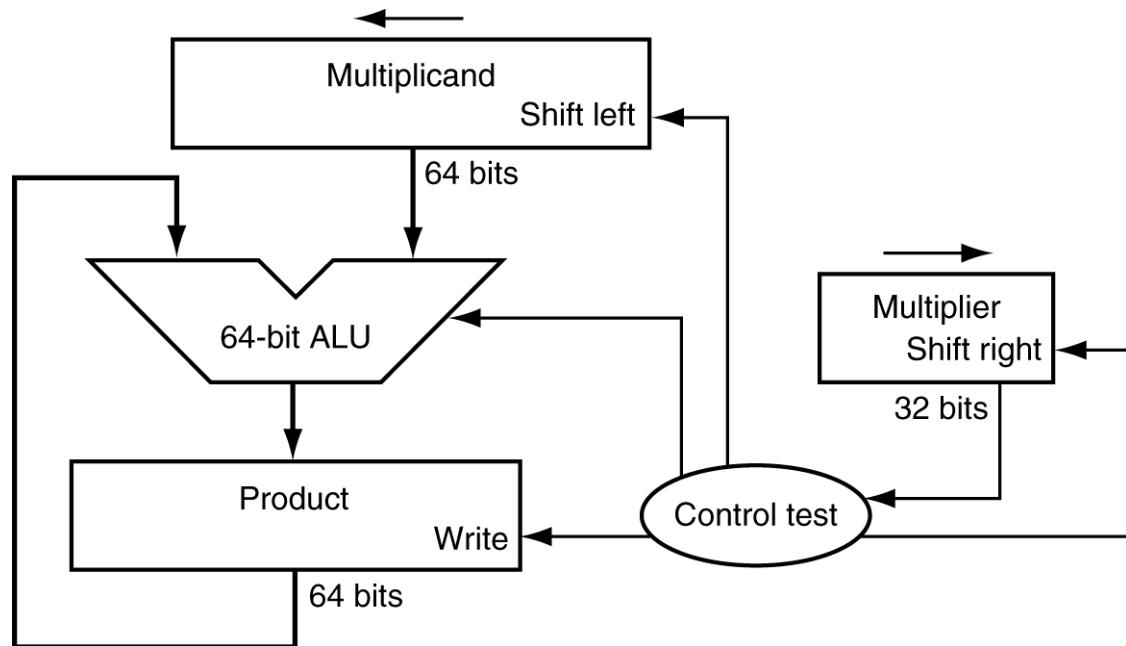
- Conceptually straightforward
- Fairly expensive hardware

# Multiplication (Shift and Repeated Additions)

- Mimic the multiplication process in hardware



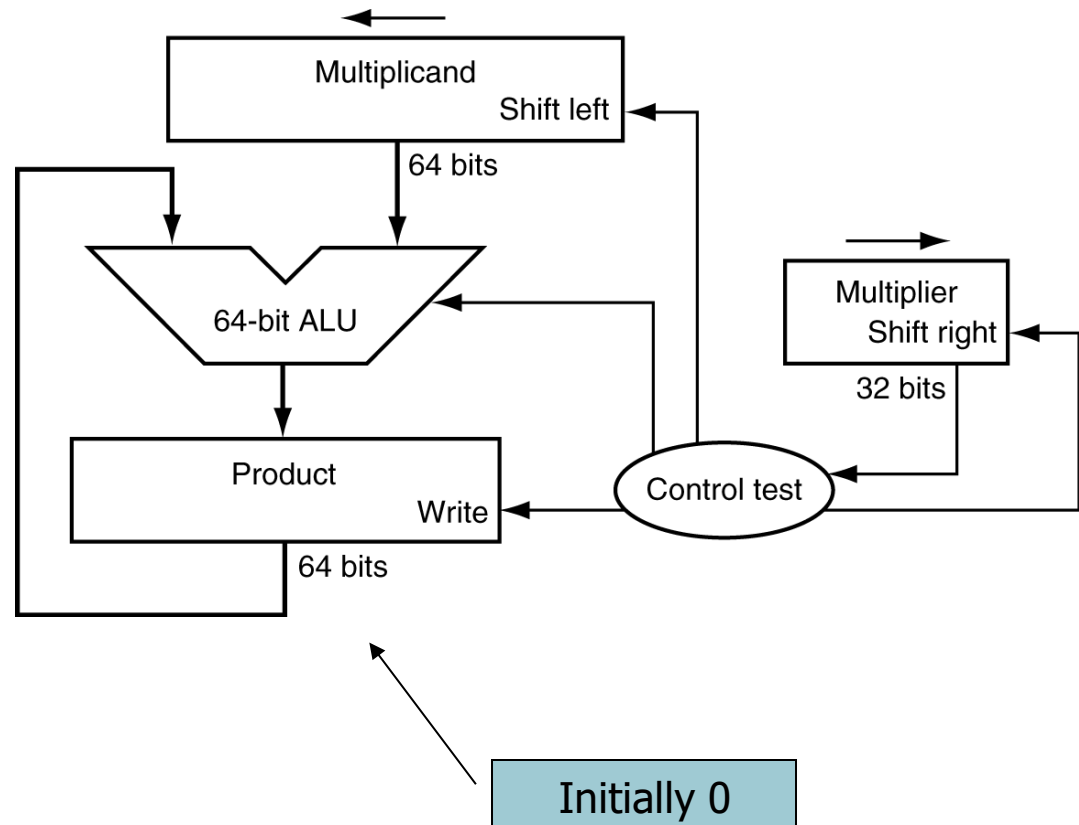
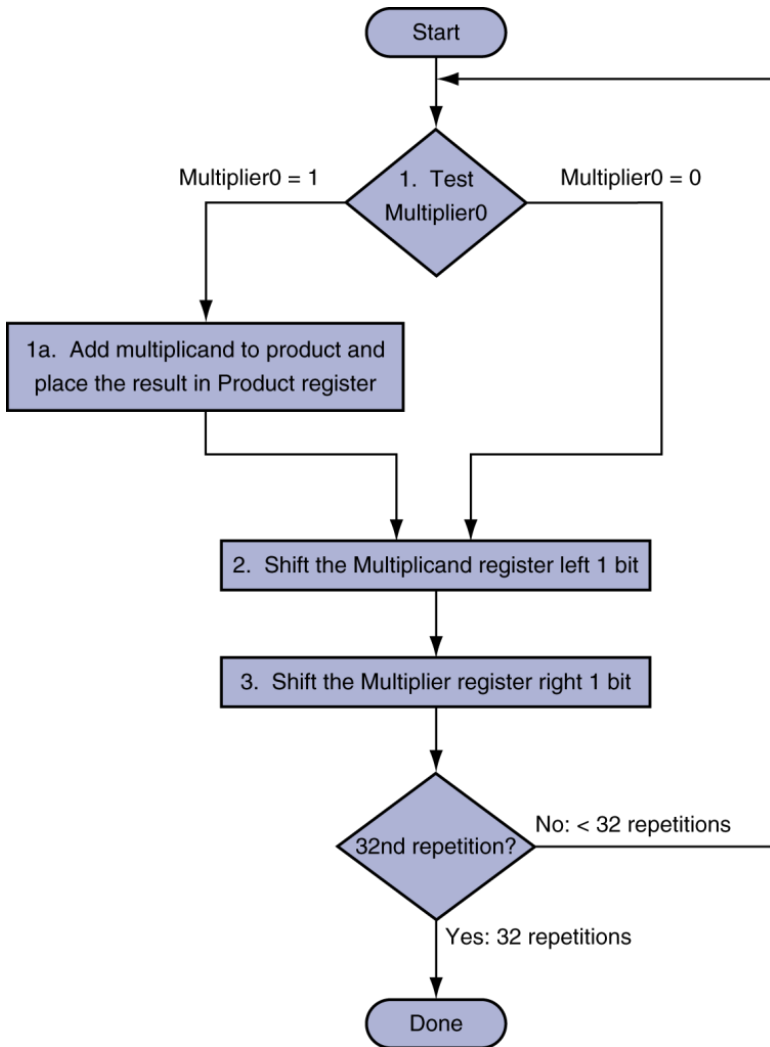
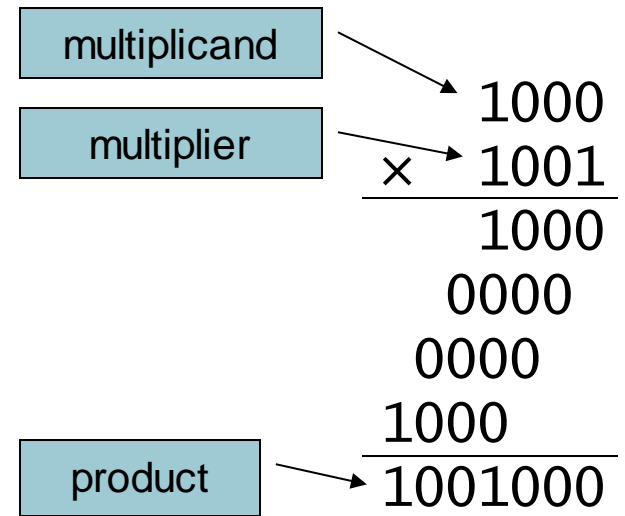
Length of product is the sum of operand lengths



# Explanations

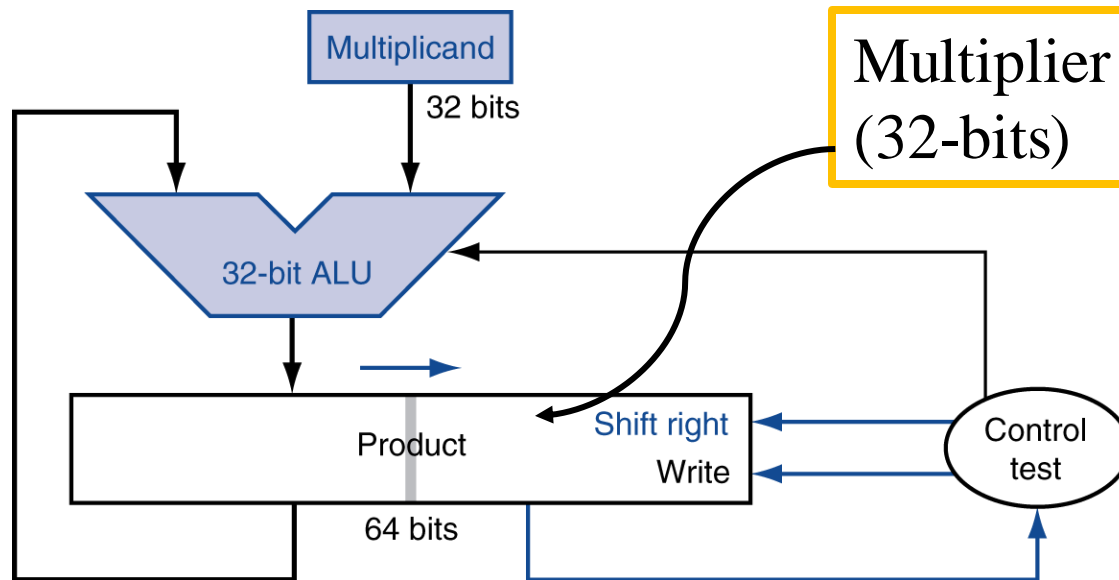
- Multiplicand register is kept 64-bit wide because 32-bit multiplicand will be shifted 32 times to the left
  - Requires a 64-bit ALU
- Product register must be 64-bit wide to accommodate the result
- Contents of multiplier register is shifted 32 times to the right so that each bit successively appears as the least significant bit (LSB) to be checked by the controller

# Multiplication Hardware



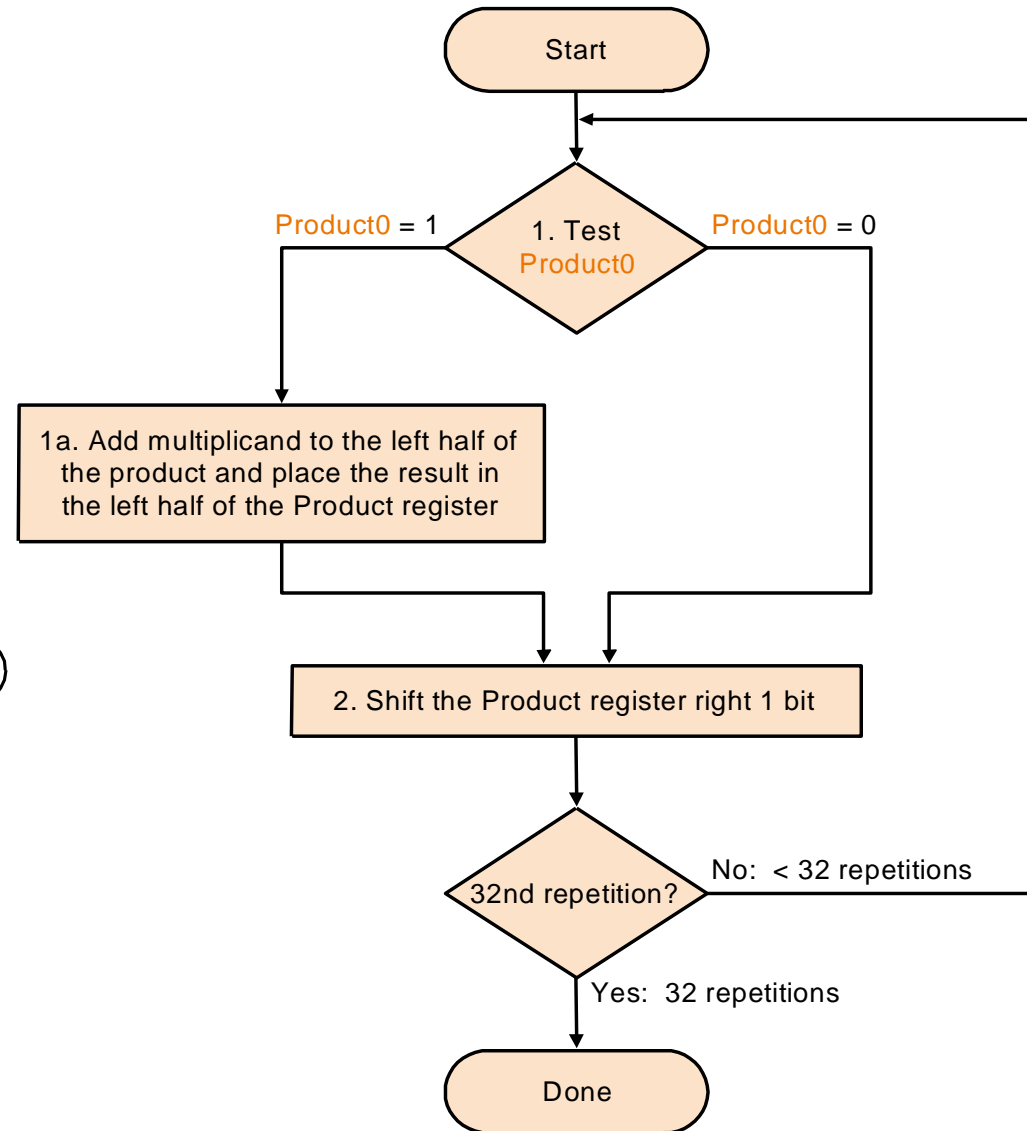
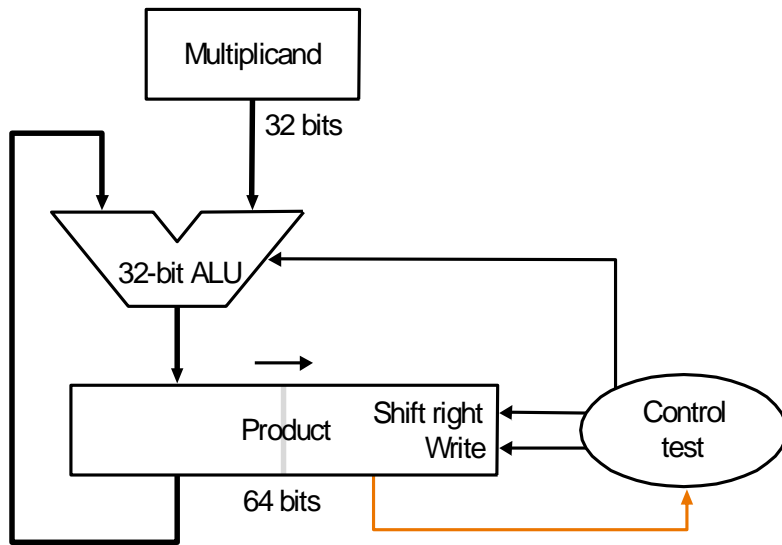
# An Optimized Version of Multiplier

- Perform steps in parallel: add/shift



- One cycle per partial-product addition

# Optimized Multiplication: Implementation



# Multiplication Example

$0010 \times 0110 = ?$  0010 (+ 2, multiplicand); 0110 (+ 6, multiplier)

| Iteration | multiplier | Original algorithm                          |           |
|-----------|------------|---|-----------|
|           |            | Step  | Product   |
| 0         | 0010       | Initial values                              | 0000 0110 |
| 1         | 0010       | 1:0 $\Rightarrow$ no operation              | 0000 0110 |
|           | 0010       | 2: Shift right Product <b>logical shift</b> | 0000 0011 |
| 2         | 0010       | 1a:1 $\Rightarrow$ prod = Prod + Mcand      | 0010 0011 |
|           | 0010       | 2: Shift right Product                      | 0001 0001 |
| 3         | 0010       | 1a:1 $\Rightarrow$ prod = Prod + Mcand      | 0011 0001 |
|           | 0010       | 2: Shift right Product                      | 0001 1000 |
| 4         | 0010       | 1:0 $\Rightarrow$ no operation              | 0001 1000 |
|           | 0010       | 2: Shift right Product <b>+12</b>           | 0000 1100 |

# Signed Multiplication

- Recall
  - For  $p = a \times b$ , if either  $a < 0$  or  $b < 0$ , then  $p < 0$
  - If  $(a < 0 \text{ and } b < 0)$  or  $(a > 0 \text{ and } b > 0)$  then  $p > 0$
  - Hence  **$\text{sign}(p) = \text{sign}(a) \oplus \text{sign}(b)$**
- Hence
  - Convert multiplier and multiplicand to positive number each
  - Multiply two positive numbers
  - Compute sign, convert product accordingly



# Multiplication: A Fundamental Question

What is the complexity of multiplying two  $n$ -bit integers?

$$\begin{array}{r}
 \begin{array}{cccc}
 A_3 & A_2 & A_1 & A_0 \\
 \times B_3 & B_2 & B_1 & B_0
 \end{array} \\
 \hline
 \begin{array}{cccc}
 A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
 A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
 + A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3
 \end{array}
 \end{array}$$

$AB_i$  called a "partial product"

→

$A_i$  and  $B_i$  are all 0 or 1

Multiplying  $N$ -bit number by  $M$ -bit number gives  $(N+M)$ -bit result

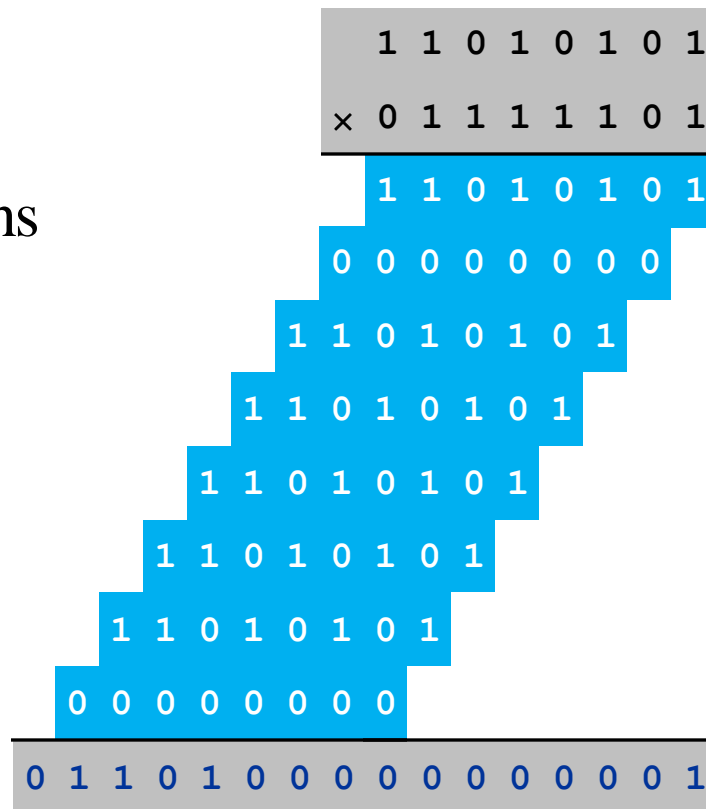
# Integer Multiplication

## Multiplication

Given two  $n$ -bit integers  $a$  and  $b$ , compute  $a \times b$

School method

$\Theta(n^2)$  bit operations



Ref: Jon Kleinberg and Éva Tardos,  
Algorithm Design  
Slides by Kevin Wayne  
Copyright © 2005 Pearson-Addison  
Wesley

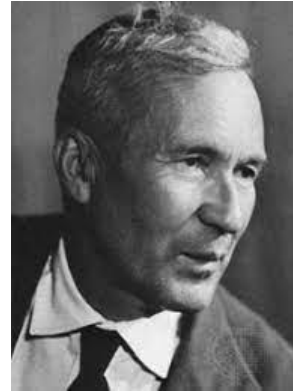
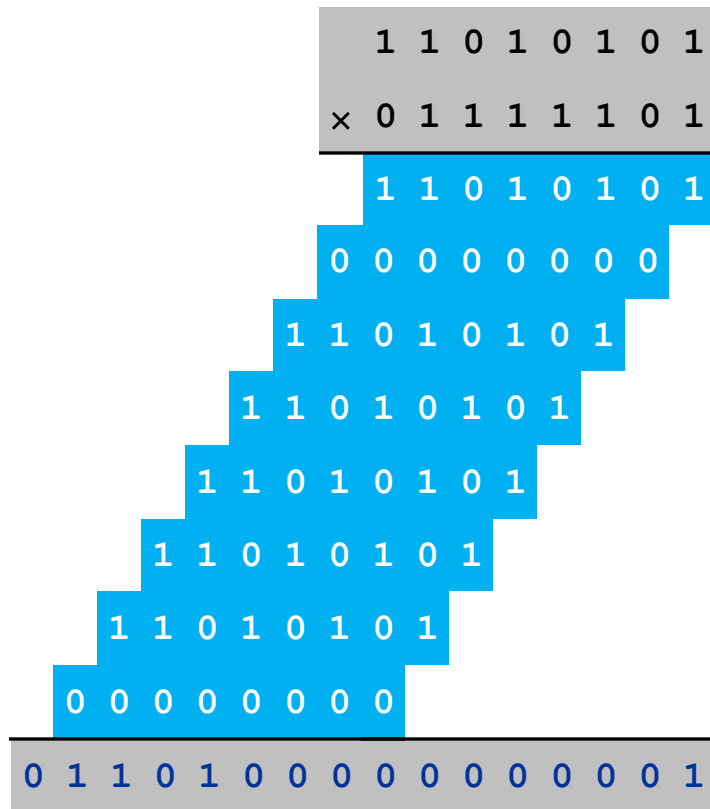
Q. Is school multiplication algorithm optimal?

# Integer Multiplication

School method  
 $\Theta(n^2)$  bit operations

## Multiplication

Given two  $n$ -bit integers  $a$  and  $b$ , compute  $a \times b$



Andrey Nikolaevich Kolmogorov (1903 – 1987)

1952: Kolmogorov conjectured that any multiplication algorithm will take  $\Omega(n^2)$  bit operations, i.e., it is asymptotically optimal

1960: Kolmogorov announced in a seminar at Moscow State University that it is indeed  $\Omega(n^2)$

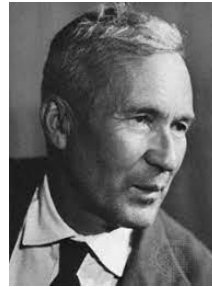
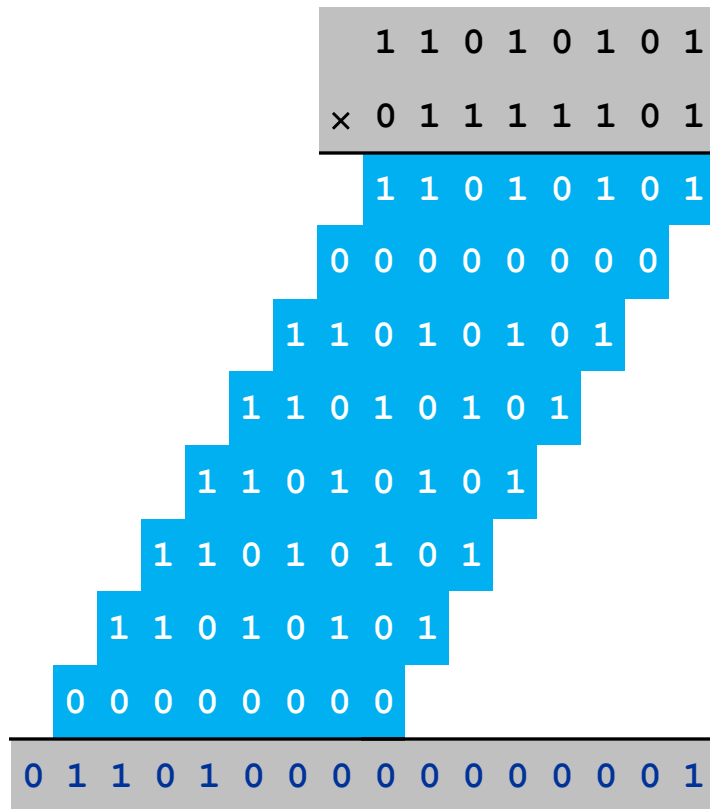
Q. Is grade-school multiplication algorithm optimal?

# Integer Multiplication

School method  
 $\Theta(n^2)$  bit operations

## Multiplication

Given two  $n$ -bit integers  $a$  and  $b$ , compute  $a \times b$



Kolmogorov

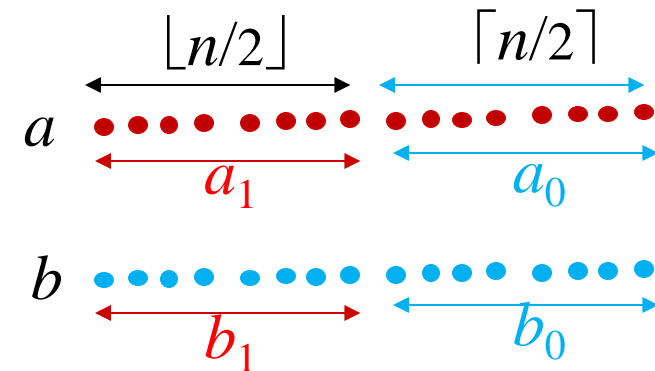
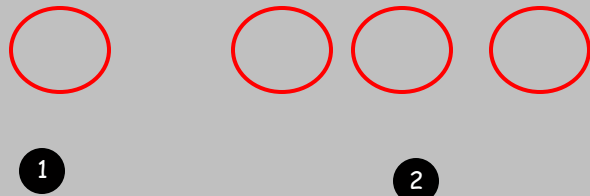
1960: Kolmogorov announced in a seminar at Moscow State University that multiplication is indeed  $\Omega(n^2)$

Anatoly Alexeyevich  
Karatsuba (1937 – 2008)

1960: Within one week,  
Karatsuba disproved the claim,  
showing that multiplication can  
be done in  $O(n^{1.585})$  time!



# Karatsuba multiplication algorithm for two $n$ -bit integers $a, b$



To multiply two  $n$ -bit integers  $a$  and  $b$ :

- Add two  $\frac{1}{2}n$  bit integers.
- Multiply **three**  $\frac{1}{2}n$ -bit integers, recursively
- Add, subtract, and shift to obtain result

$$\begin{aligned} & a_1 b_0 + a_0 b_1 \\ = & (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 \end{aligned}$$

**Theorem:** [Karatsuba-Ofman 1962] One can multiply two  $n$ -bit integers in  $O(n^{1.585})$  bit operations  $\square$

# Multiplication of two $n$ -bit integers $a, b$

**Theorem:** [Schönhage and Strassen] It is possible to multiply two  $n$ -bit integers in  $O(n \log n \log \log n)$  bit operations

Ref: D. E. Knuth, The Art of Computer Programming,  
Vol. 2: Seminumerical Algorithms, Addison Wesley