

# L8/9: Arithmetic Structures



## Acknowledgements:

Materials in this lecture are courtesy of the following people and used with permission.

- Randy H. Katz (University of California, Berkeley, Department of Electrical Engineering & Computer Science)
- Gaetano Borriello (University of Washington, Department of Computer Science & Engineering, <http://www.cs.washington.edu/370>)
- J. Rabaey, A. Chandrakasan, B. Nikolic. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2003.
- Kevin Atkinson, Rex Min

## How to represent negative numbers?

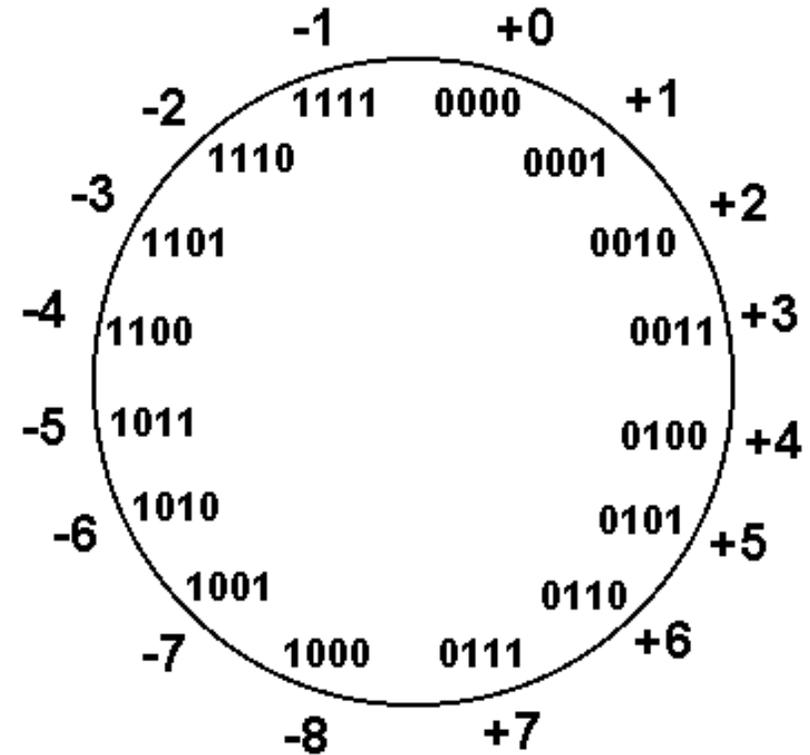
- Three common schemes: sign-magnitude, ones complement, twos complement
- Sign-magnitude: MSB = 0 for positive, 1 for negative
  - Range:  $-(2^{N-1} - 1)$  to  $+(2^{N-1} - 1)$
  - Two representations for zero: 0000... & 1000...
  - Simple multiplication but complicated addition/subtraction
- Ones complement: if N is positive then its negative is  $\bar{N}$ 
  - Example: 0111 = 7, 1000 = -7
  - Range:  $-(2^{N-1} - 1)$  to  $+(2^{N-1} - 1)$
  - Two representations for zero: 0000... & 1111...
  - Subtraction implemented as addition and negation

Twos complement = bitwise complement + 1

$$0111 \rightarrow 1000 + 1 = 1001 = -7$$

$$1001 \rightarrow 0110 + 1 = 0111 = 7$$

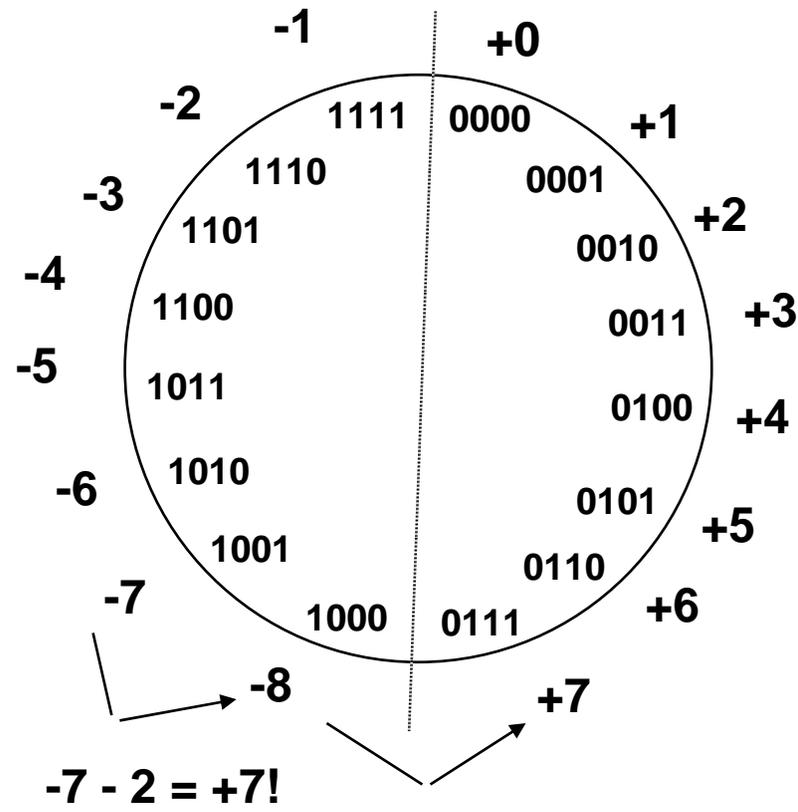
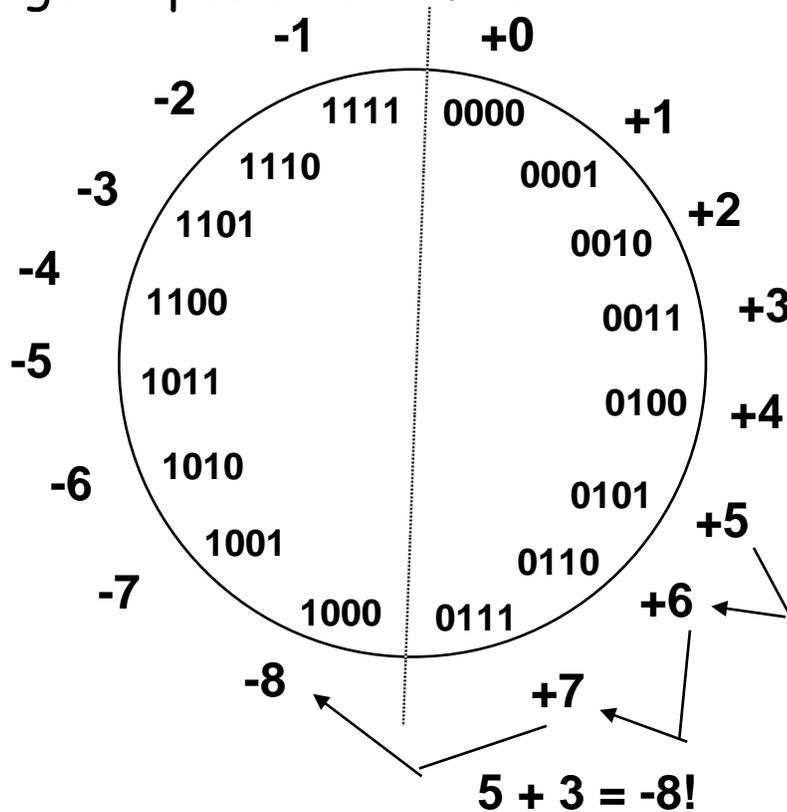
- Asymmetric range:  $-2^{N-1}$  to  $+2^{N-1}-1$
- Only one representation for zero
- Simple addition and subtraction
- Most common representation



4	0100	-4	1100	4	0100	-4	1100
<u>+ 3</u>	<u>0011</u>	<u>+ (-3)</u>	<u>1101</u>	<u>- 3</u>	<u>1101</u>	<u>+ 3</u>	<u>0011</u>
7	0111	-7	11001	1	10001	-1	1111

[Katz05]

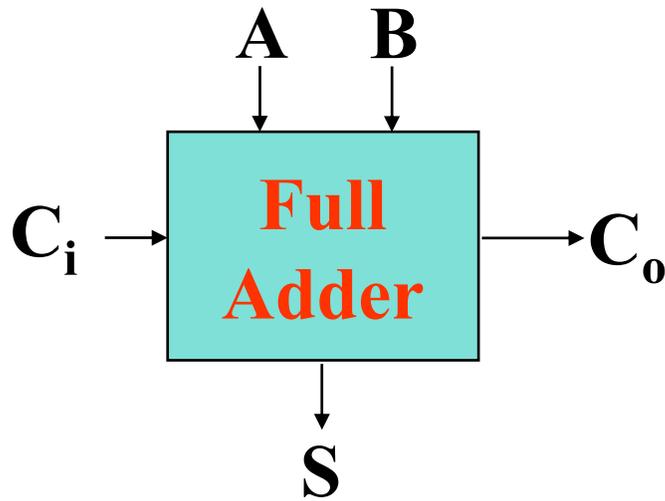
Add two positive numbers to get a negative number or two negative numbers to get a positive number



5	0	1	1	1
	0	1	0	1
<u>3</u>		<u>0</u>	<u>0</u>	<u>1</u>
-8	0	1	0	0

-7	1	0	0	0
	1	0	0	1
<u>-2</u>		<u>1</u>	<u>1</u>	<u>0</u>
7	1	0	1	1

**If carry in to sign equals carry out then can ignore carry out, otherwise have overflow**



$$S = A \oplus B \oplus C_i$$

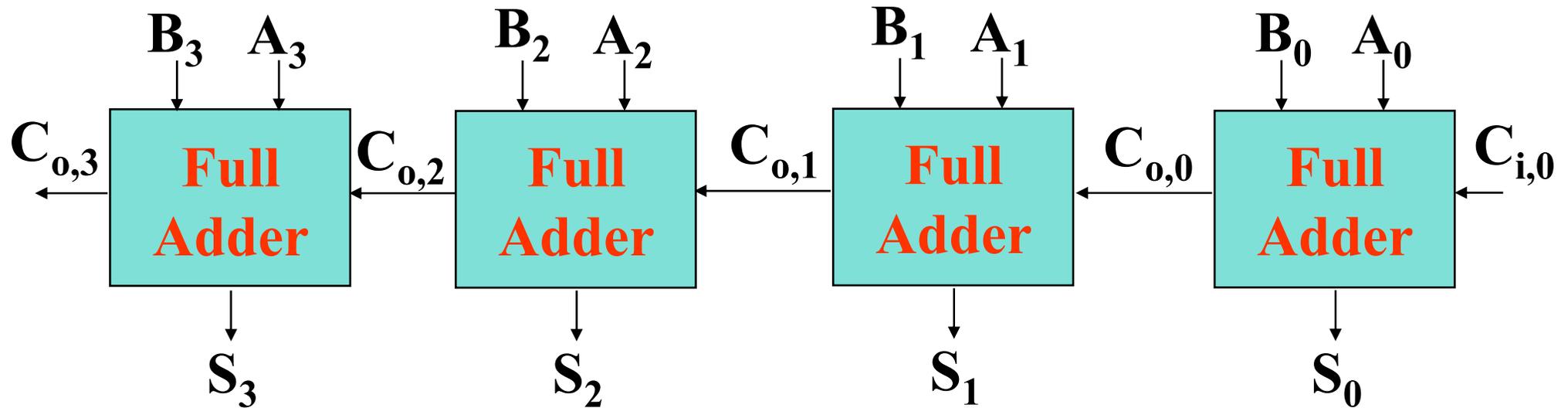
$$= \overline{A}\overline{B}C_i + \overline{A}B\overline{C}_i + A\overline{B}\overline{C}_i + ABC_i$$

$$C_o = AB + C_i(A+B)$$

A	B	CI	S	CO
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

		A B			
		00	01	11	10
S	CI 0	0	1	0	1
	CI 1	1	0	1	0

		A B			
		00	01	11	10
CO	CI 0	0	0	1	0
	CI 1	0	1	1	1

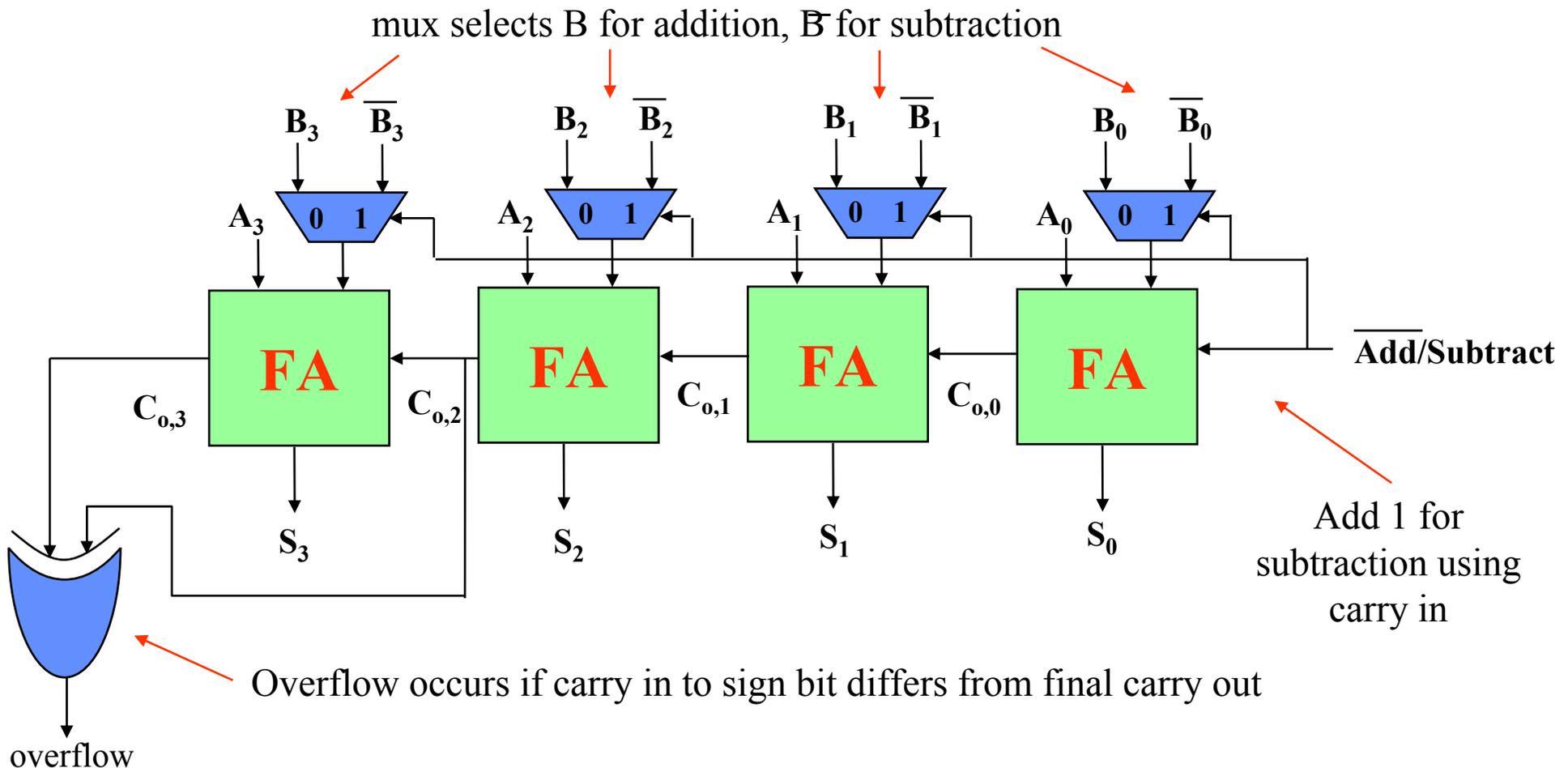


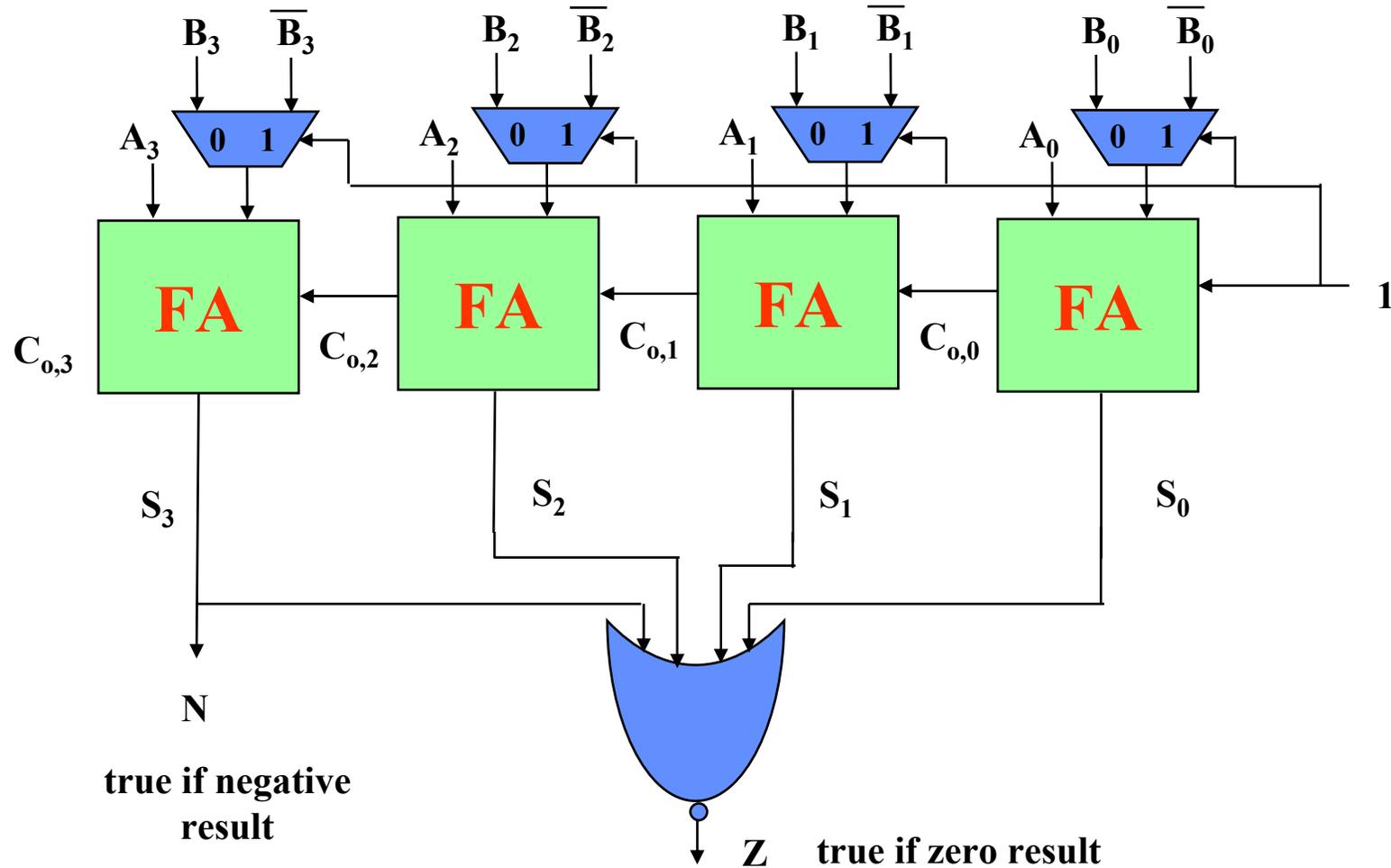
**Worst case propagation delay linear with the number of bits**

$$t_{\text{adder}} = (N-1)t_{\text{carry}} + t_{\text{sum}}$$

- Under two's complement, subtracting  $B$  is the same as adding the bitwise complement of  $B$  then adding 1

Combination addition/subtraction system:

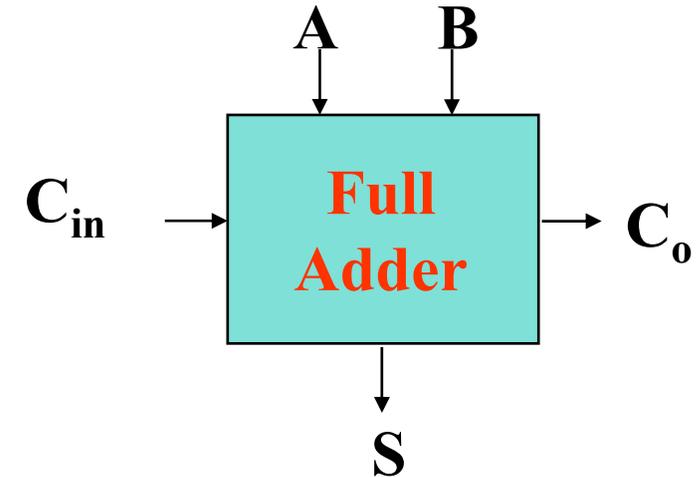




$$\begin{aligned}
 A < B &= N \\
 A = B &= Z \\
 A \leq B &= Z + N
 \end{aligned}$$

## How to Speed up the Critical (Carry) Path? (How to Build a Fast Adder?)

$A$	$B$	$C_i$	$S$	$C_o$	Carry status
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate



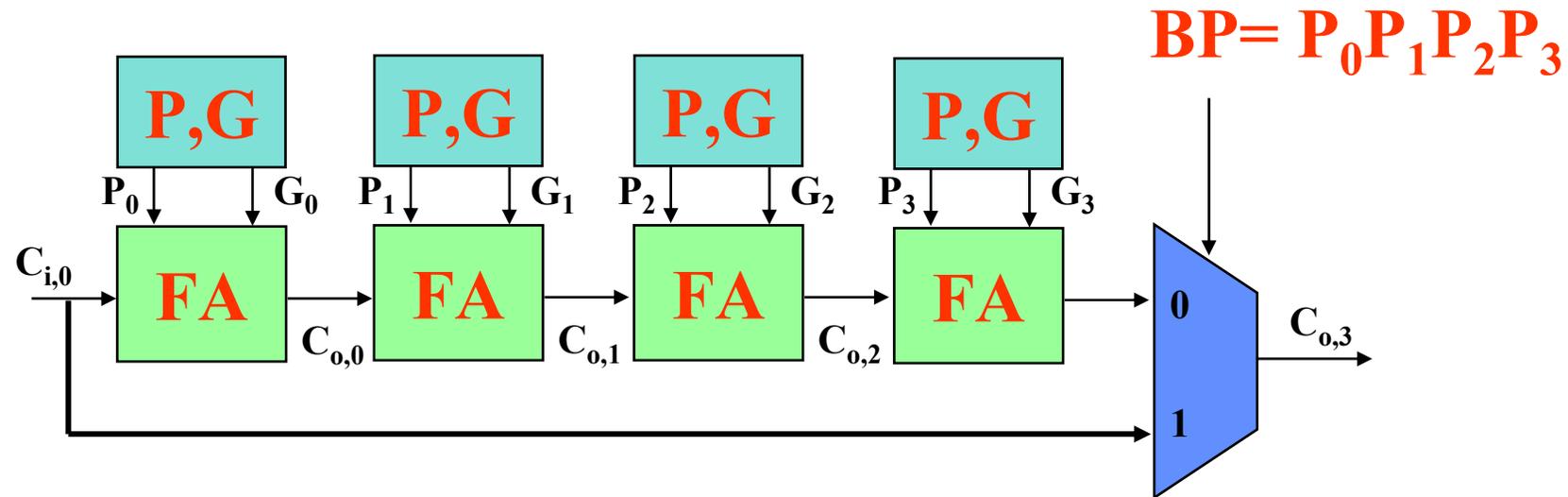
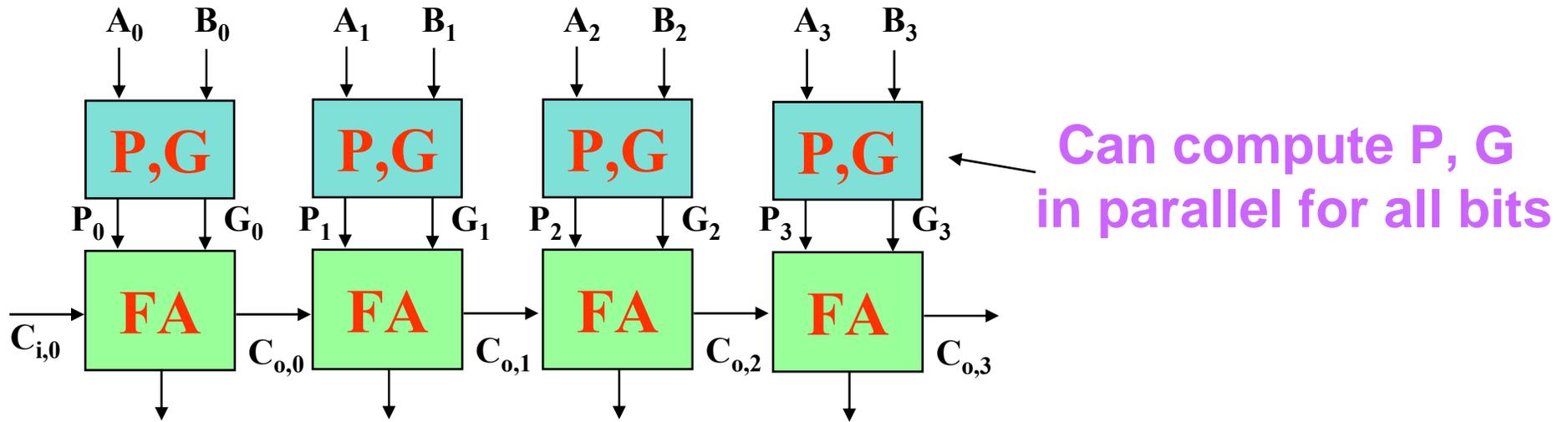
$$\text{Generate } (G) = AB$$

$$\text{Propagate } (P) = A \oplus B$$

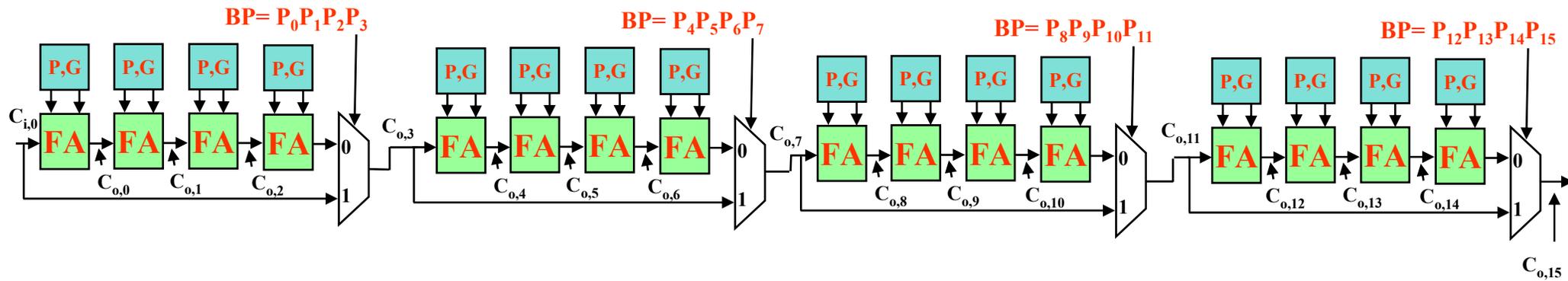
$$C_o(G, P) = G + PC_i$$

$$S(G, P) = P \oplus C_i$$

Note: can also use  $P = A + B$  for  $C_o$



**Key Idea:** if  $(P_0 P_1 P_2 P_3)$  then  $C_{o,3} = C_{i,0}$



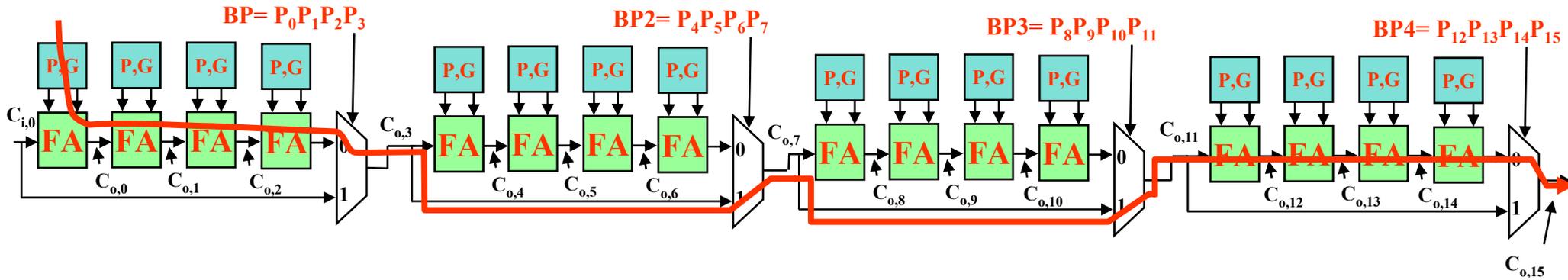
**Assume the following for delay each gate:**

**P, G from A, B: 1 delay unit**

**P, G,  $C_i$  to  $C_o$  or Sum for a FA: 1 delay unit**

**2:1 mux delay: 1 delay unit**

**What is the worst case propagation delay for the 16-bit adder?**



For the second stage, is the critical path:

**BP2 = 0 or BP2 = 1?**

**Message: Timing Analysis is Very Tricky –  
Must Carefully Consider Data Dependencies For  
False Paths**

Re-express the carry logic as follows:

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

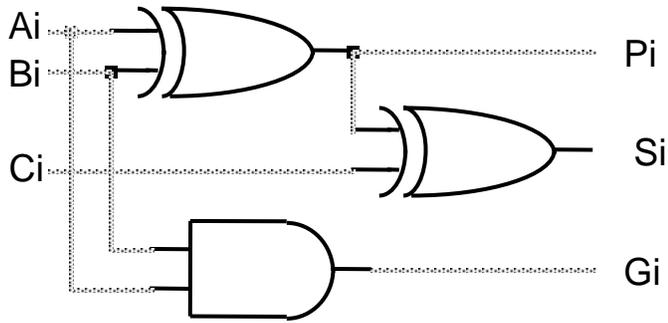
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

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

...

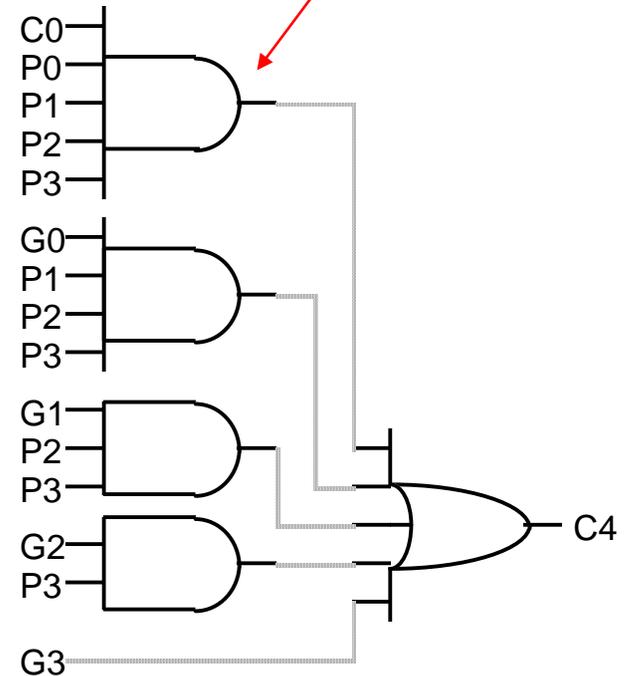
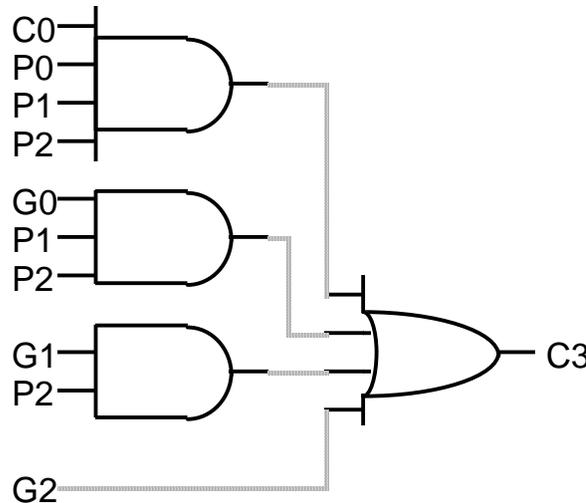
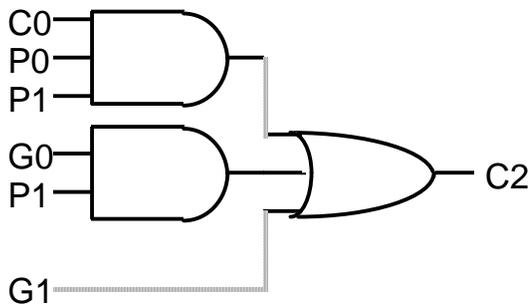
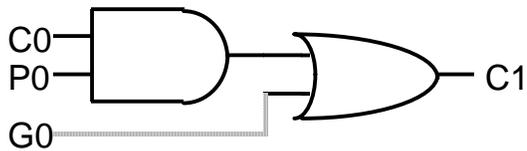
- Each of the carry equations can be implemented in a two-level logic network
- Variables are the adder inputs and carry in to stage 0

**Ripple effect has been eliminated!**



**Adder with propagate and generate outputs**

Later stages have **increasingly complex** logic



$G_{j:i}$  and  $P_{j:i}$  denote the **Generate** and **Propagate** functions, respectively, for a group of bits from positions  $i$  to  $j$ . We call them **Block Generate** and **Block Propagate**.  $G_{j:i}$  equals 1 if the group generates a carry **independent** of the incoming carry.  $P_{j:i}$  equals 1 if an incoming carry propagates **through the entire group**. For example,  $G_{3:2}$  is equal to 1 if a carry is generated at bit position 3, or if a carry out is generated at bit position 2 and propagates through position 3.  $G_{3:2} = G_3 + P_3 G_2$ .  $P_{3:2}$  is true if an incoming carry propagates through both bit positions 2 and 3.  $P_{3:2} = P_3 P_2$

$$C_2 = (G_1 + P_1 G_0) + (P_1 P_0)C_0 = G_{1:0} + P_{1:0} C_0$$

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

$$= (G_3 + P_3 G_2) + (P_3 P_2)C_{0,1} = G_{3:2} + P_{3:2} C_2$$

$$= G_{3:2} + P_{3:2}(G_{1:0} + P_{1:0} C_0) = G_{3:0} + P_{3:0} C_0$$

The carry out of a 4-bit block can thus be computed using only the block generate and propagate signals **for each 2-bit section**, plus the carry in to bit 0. The same formulation will be used to generate the carry out signals for a 16-bit adder using the block generate and propagate from 4-bit sections.

$$(g, p) \bullet (g', p') = (g + pg', pp')$$

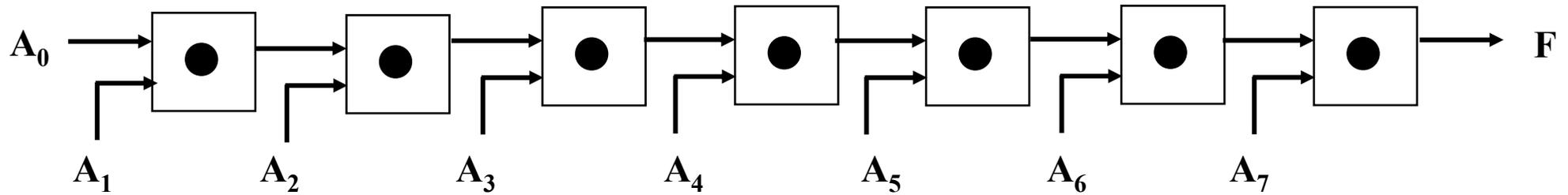
The above dot operator obeys the associative property, but it is not commutative

$$(G_{3:2}, P_{3:2}) = (G_3, P_3) \bullet (G_2, P_2)$$

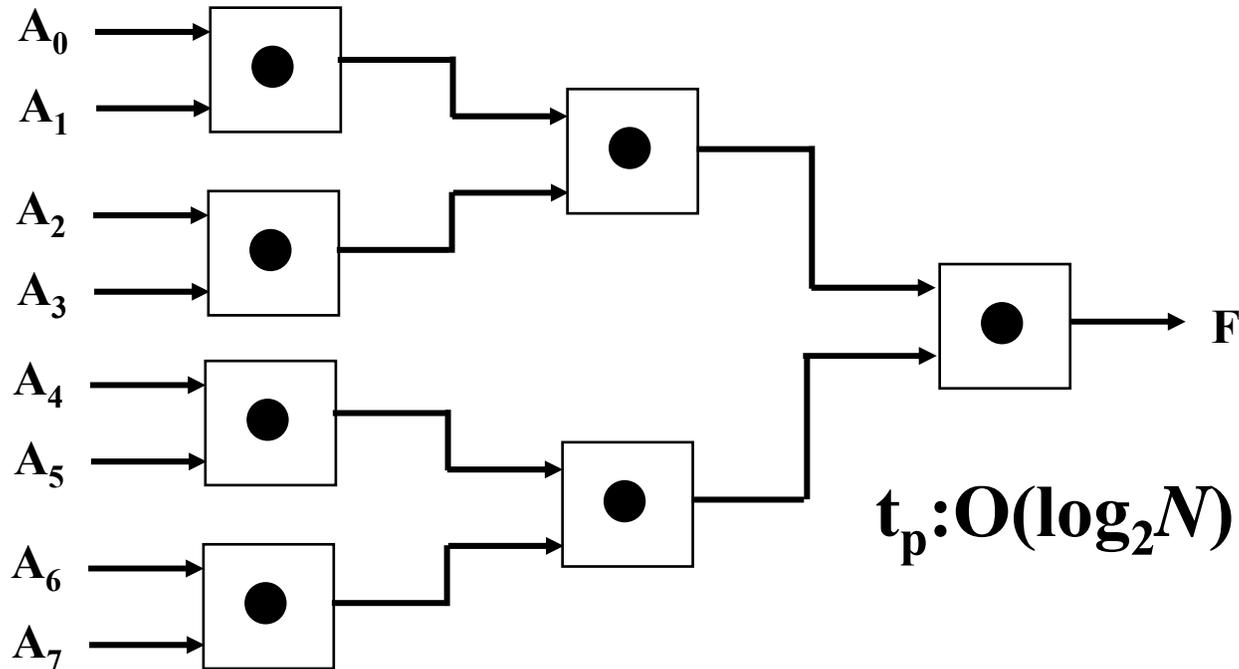
$$(C_{0,3}, 0) = ((G_3, P_3) \bullet (G_2, P_2) \bullet (G_1, P_1) \bullet (G_0, P_0)) \bullet (C_{i,0}, 0)$$

$$\begin{aligned} (G_{3:0}, P_{3:0}) &= [(G_3, P_3) \bullet (G_2, P_2)] \bullet [(G_1, P_1) \bullet (G_0, P_0)] \\ &= (G_{3:2}, P_{3:2}) \bullet (G_{1:0}, P_{1:0}) \end{aligned}$$

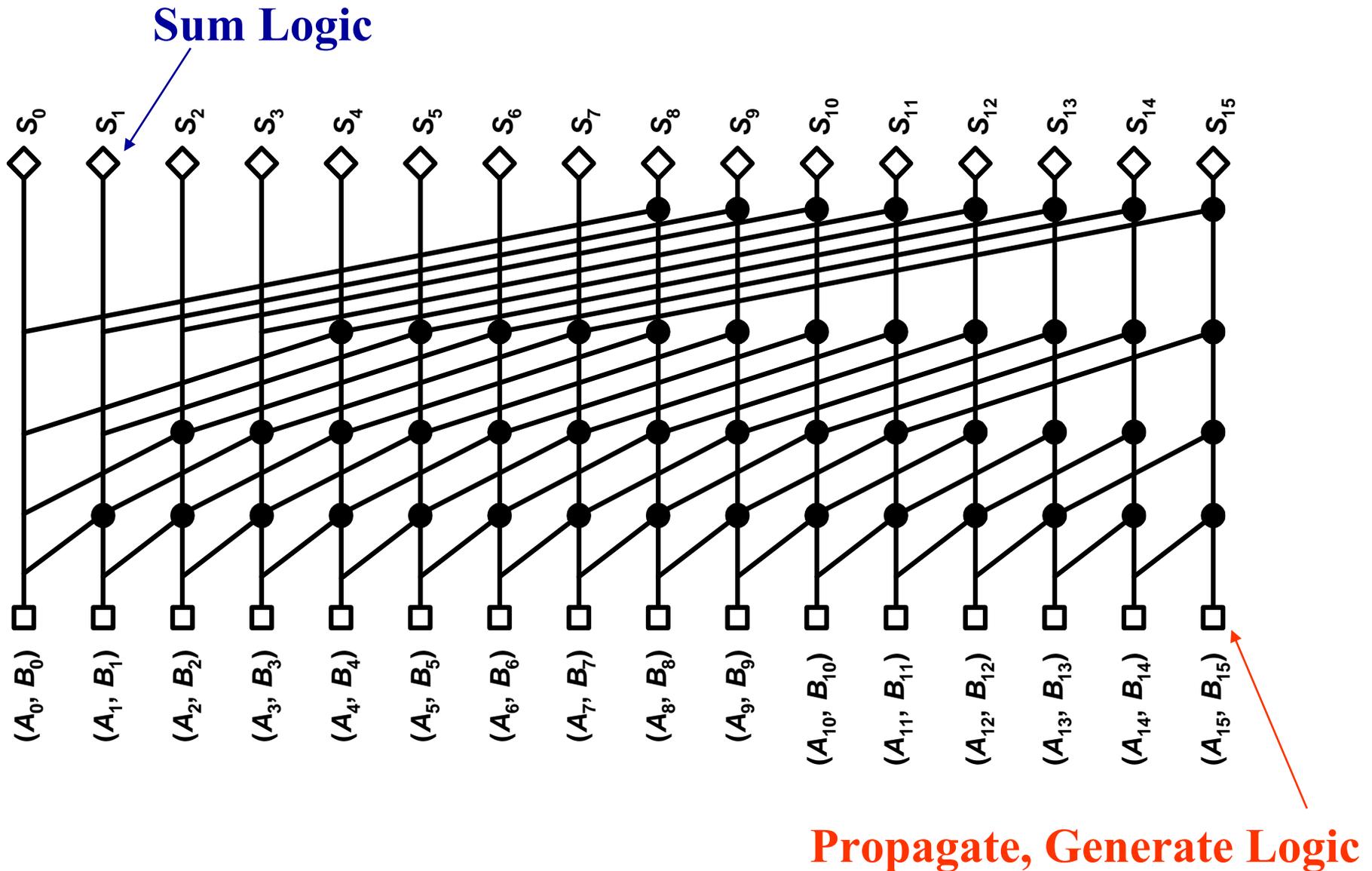
$$(C_{0,k}, 0) = ((G_k, P_k) \bullet (G_{k-1}, P_{k-1}) \bullet \dots \bullet (G_0, P_0)) \bullet (C_{i,0}, 0)$$

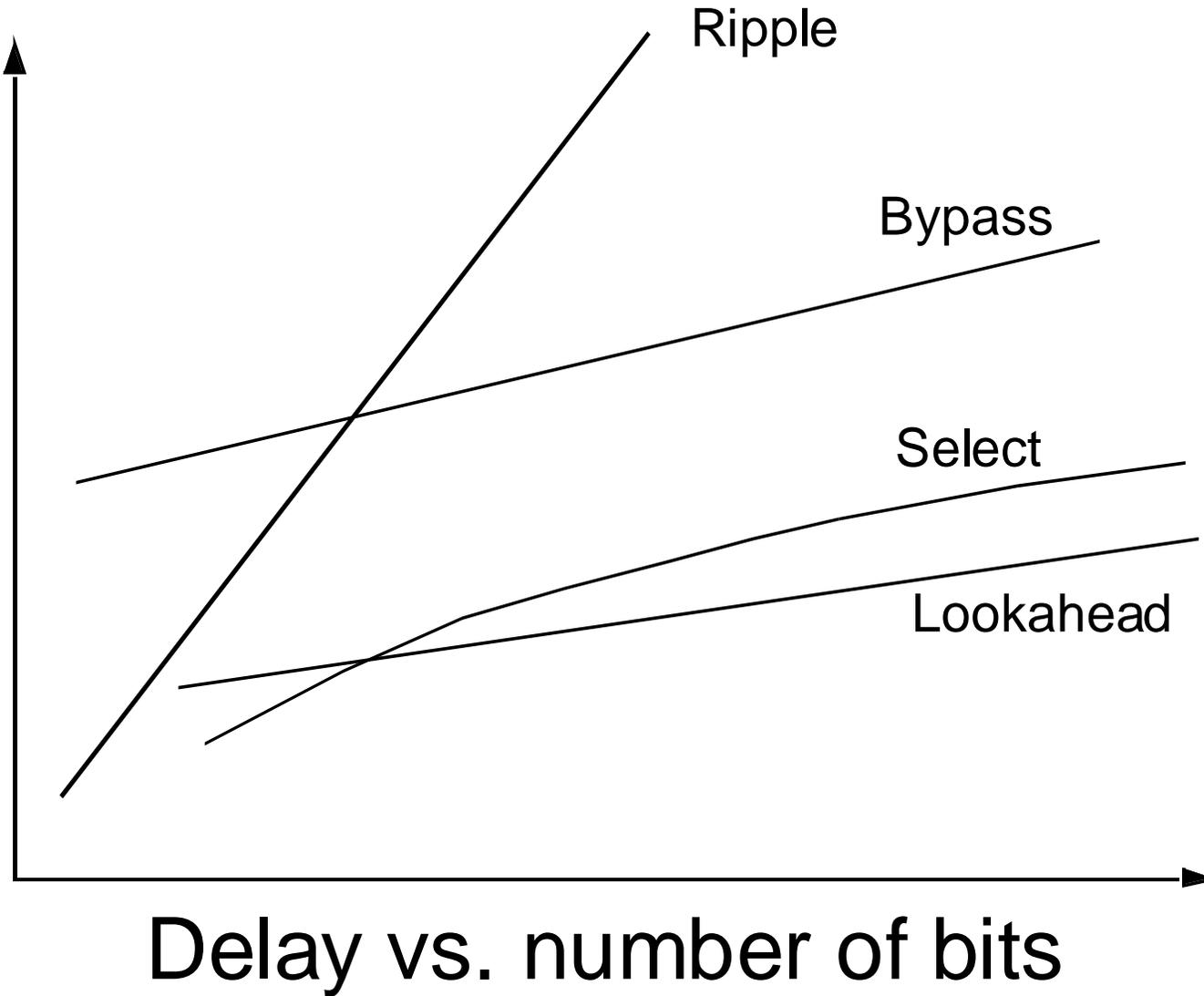


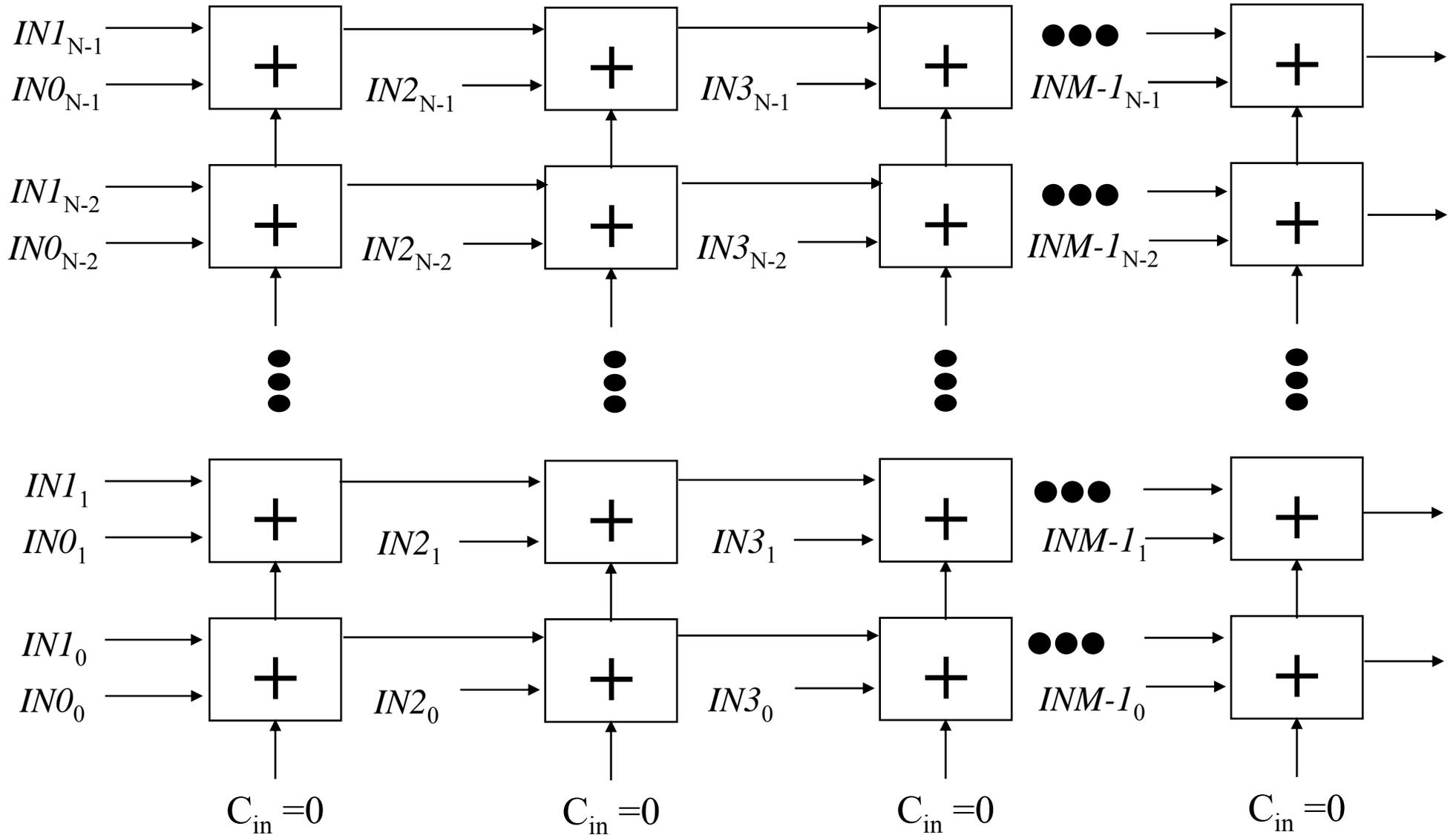
$$t_p: O(N)$$



$$t_p: O(\log_2 N)$$



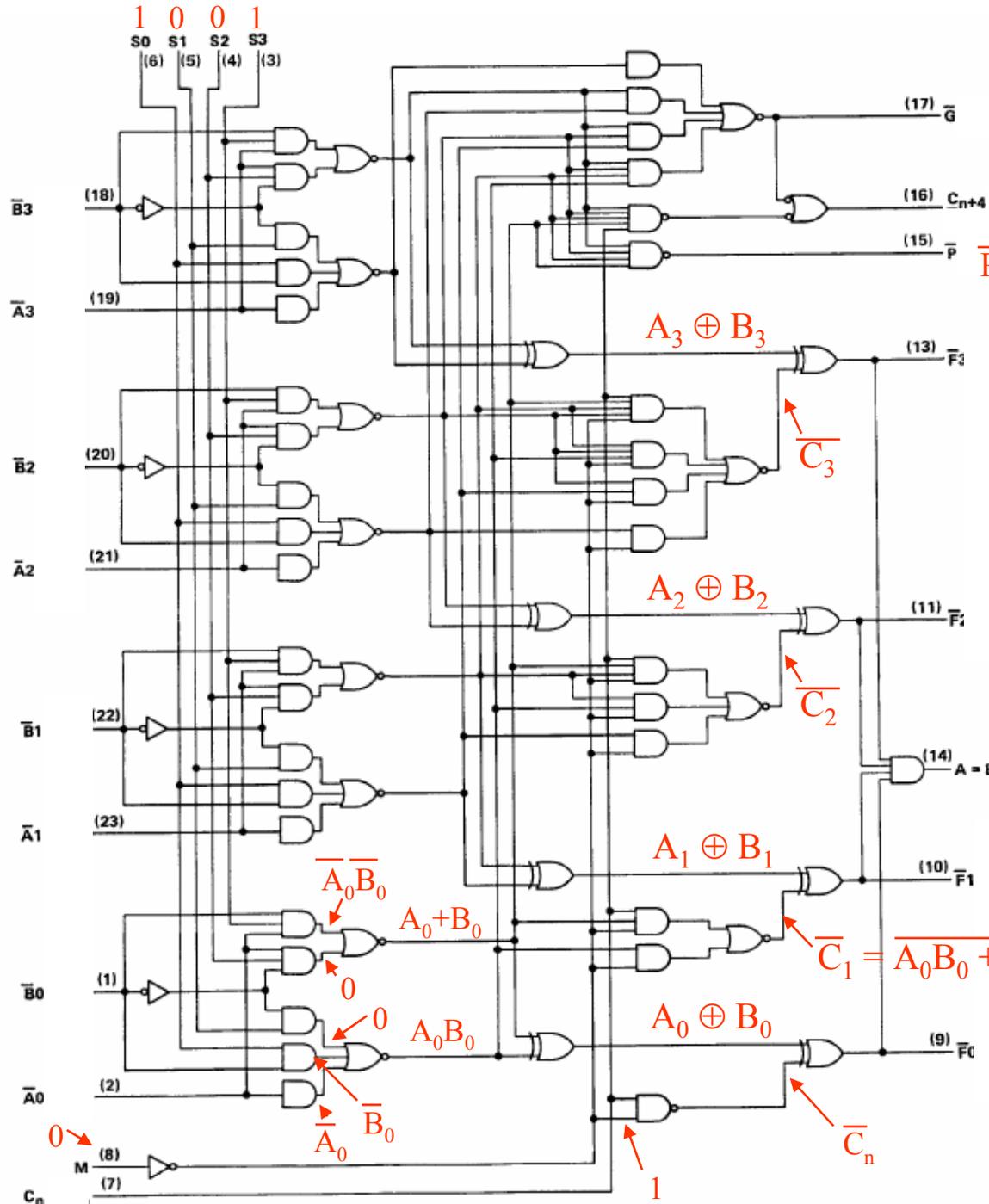




Images courtesy Texas Instruments. Images removed due to copyright restrictions.

- **16 logic functions and 16 arithmetic operations**
- **Internal 4-bit carry lookahead adder**
- **Inputs can be active high or active low (active low is shown here)**
- **Carry in and out are **opposite** polarity from other inputs/outputs**

# 74181 Addition (Active Low)



$$\bar{G} = \overline{A_3B_3 + (A_3+B_3)A_2B_2 + (A_3+B_3)(A_2+B_2)A_1B_1 + (A_3+B_3)(A_2+B_2)(A_1+B_1)A_0B_0}$$

$$\bar{P} = \overline{(A_3+B_3)(A_2+B_2)(A_1+B_1)(A_0+B_0)}$$

A	B	AB	A+B	(AB)⊕(A+B)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

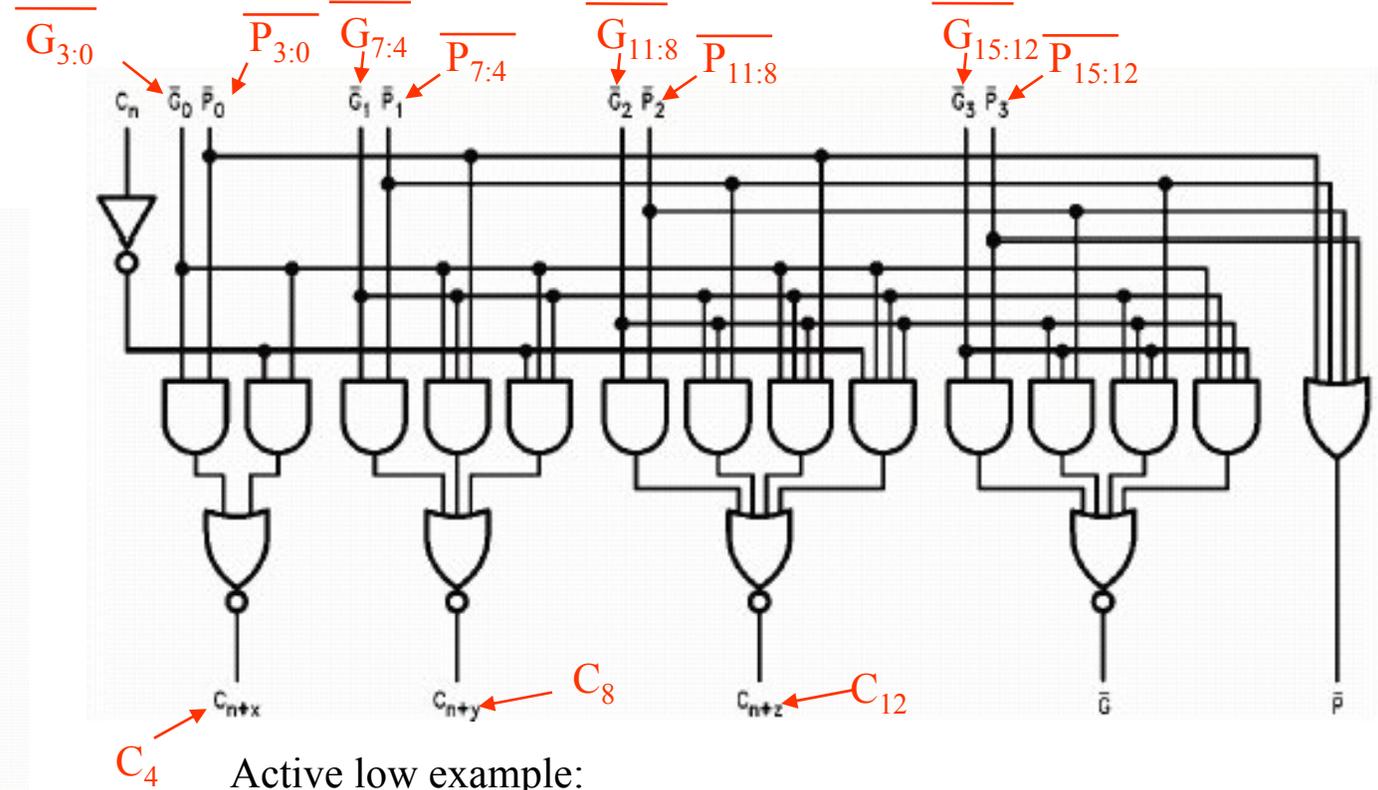
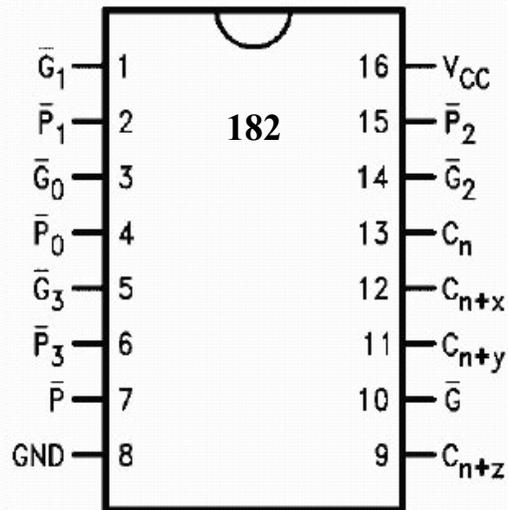
↑  
A⊕B

$$\bar{F}_1 = \overline{A_1 \oplus B_1 \oplus C_1}$$

$$\bar{C}_1 = \overline{A_0B_0 + A_0C_n + B_0C_n}$$

$$\begin{aligned} \bar{F}_0 &= \overline{A_0 \oplus B_0 \oplus \bar{C}_n} \\ &= \overline{A_0 \oplus B_0 \oplus C_n} \\ &= \overline{A_0 \oplus B_0 \oplus C_n} \end{aligned}$$

## 74182 carry lookahead unit



Active low example:

$$C_{n+x} = \overline{\overline{G_0 \cdot P_0} + \overline{G_0 \cdot C_n}}$$

$$= \overline{\overline{G_0 \cdot P_0} \cdot \overline{G_0 \cdot C_n}}$$

$$= (G_0 + P_0) \cdot (G_0 + C_n) = G_0 + P_0 C_n$$

$$\rightarrow C_4 = G_{3:0} + P_{3:0} C_n$$

$$C_{n+y} = C_8 = G_{7:4} + P_{7:4} G_{3:0} + P_{7:4} P_{3:0} C_{i,0} = G_{7:0} + P_{7:0} C_n$$

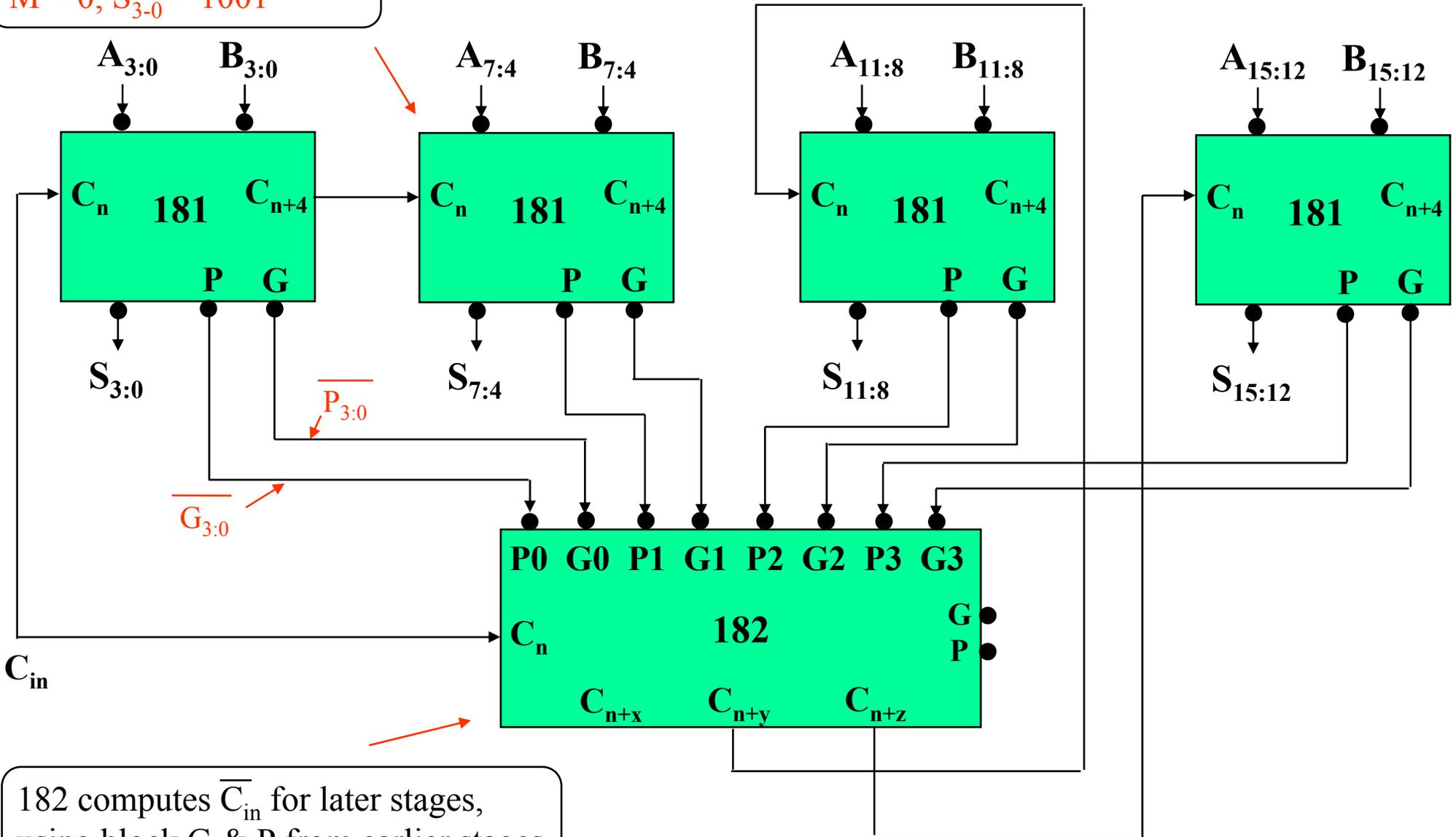
$$C_{n+z} = C_{12} = G_{11:8} + P_{11:8} G_{7:4} + P_{11:8} P_{7:4} G_{3:0} + P_{11:8} P_{7:4} P_{3:0} C_n$$

$$= G_{11:0} + P_{11:0} C_n$$

- high speed carry lookahead generator
- used with 74181 to extend carry lookahead beyond 4 bits
- correctly handles the carry polarity of the 181

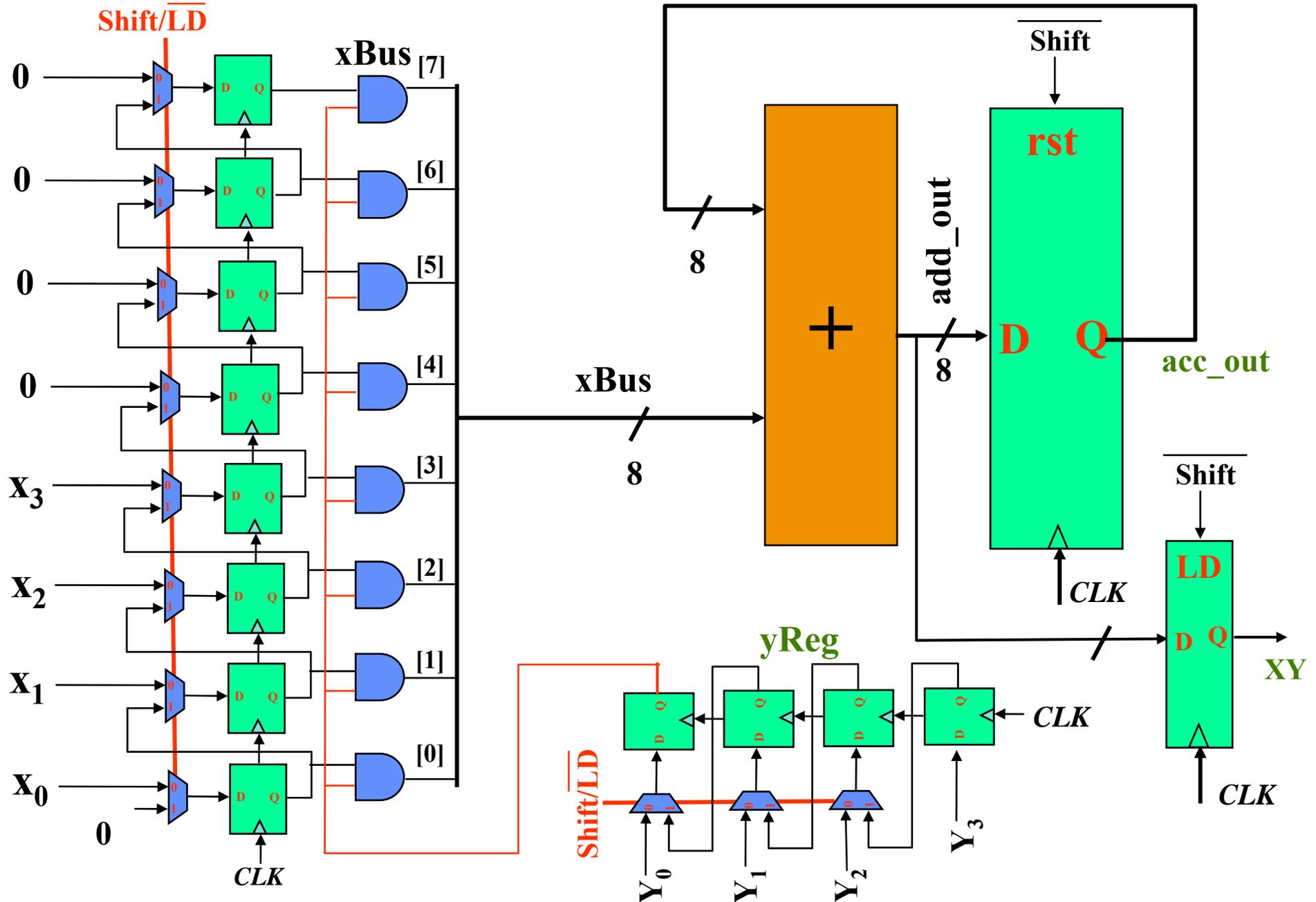
181 configured for A+B:  
 $M = 0, S_{3:0} = 1001$

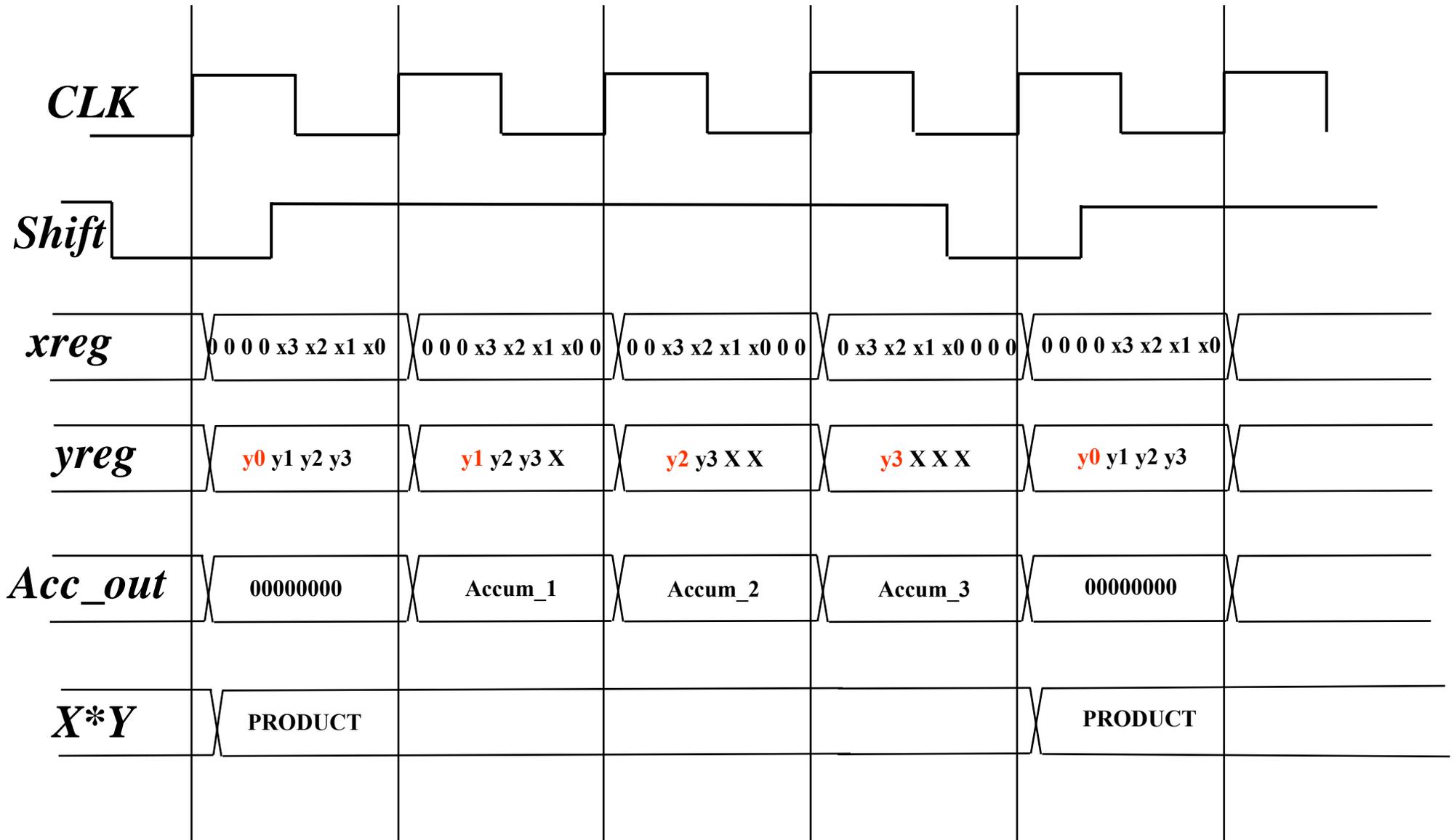
$M = 0, S_{3:0} = 1001$



182 computes  $\overline{C}_{in}$  for later stages,  
using block G & P from earlier stages







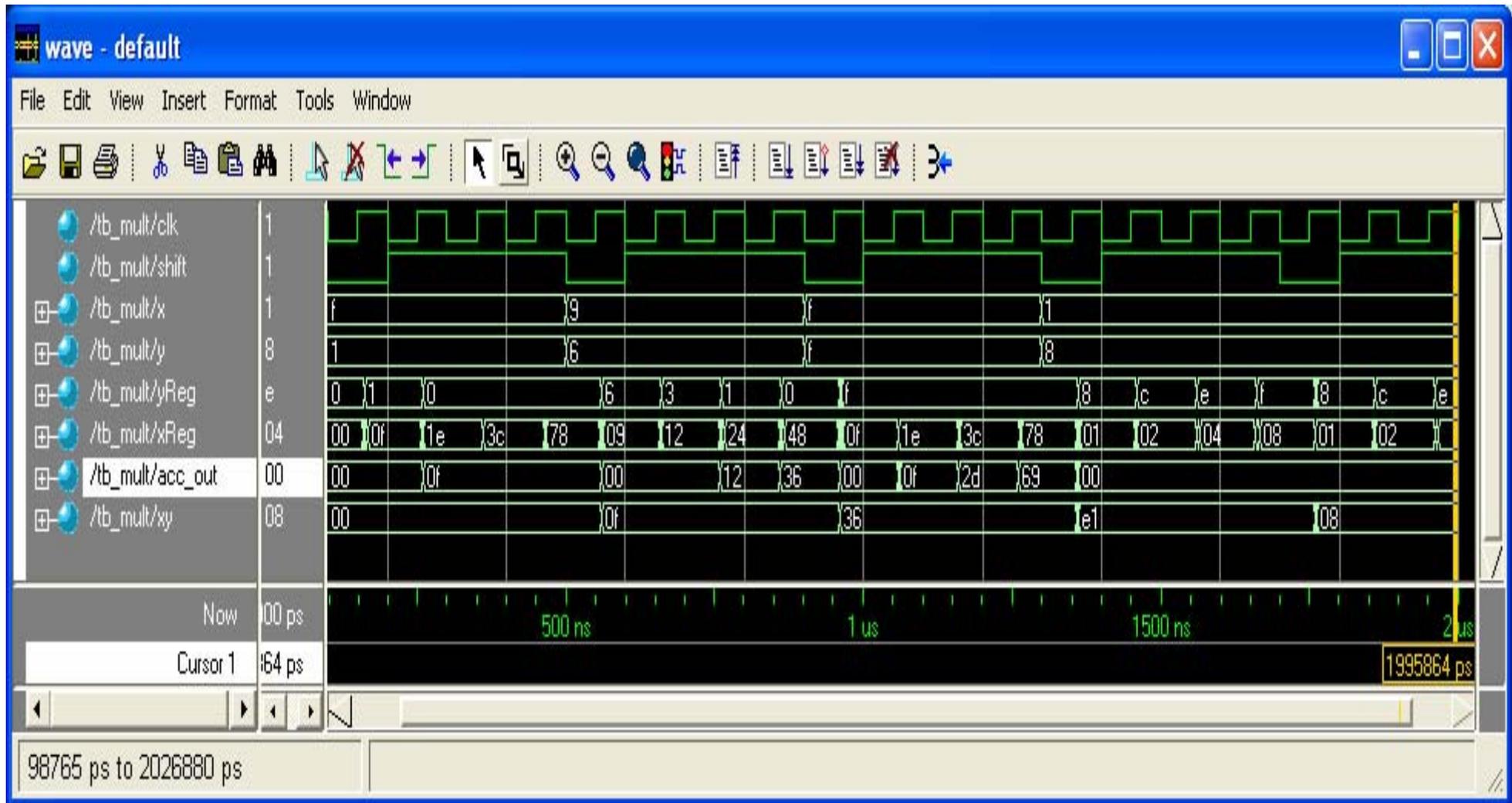
```

module serialmult(shift, clk,
x, y, xy);
input shift, clk;
input [3:0] x, y;
output [7:0] xy;
reg [7:0] xReg;
reg [3:0] yReg;
reg [7:0] xBus, acc_out,
xy_int;
wire[7:0] add_out;
assign add_out = xBus +
acc_out;
assign xy = xy_int;

always @ (yReg[0] or xReg)
begin
if (yReg[0] == 1'b0) xBus =
8'b0;
else xBus = xReg;
end

always @ (posedge clk)
begin
if (shift == 1'b0)
begin
xReg <= {4'b0, x};
yReg <= y;
acc_out <= 8'b0;
xy_int <= add_out;
end
else
begin
xReg <= {xReg[6:0], 1'b0};
yReg <= {y[3], yReg[3:1]};
acc_out <= add_out;
xy_int <= xy;
end // if shift
end // always
endmodule

```



Assuming X and Y are 4-bit twos complement numbers:

$$X = -2^3x_3 + \sum_{i=0}^2 x_i2^i \quad Y = -2^3y_3 + \sum_{i=0}^2 y_i2^i$$

The product of X and Y is:

$$XY = x_3y_32^6 - \sum_{i=0}^2 x_iy_32^{i+3} - \sum_{j=0}^2 x_3y_j2^{j+3} + \sum_{i=0}^2 \sum_{j=0}^2 x_iy_j2^{i+j}$$

For twos complement, the following is true:

$$-\sum_{i=0}^3 x_i2^i = -2^4 + \sum_{i=0}^3 \bar{x}_i2^i + 1$$

The product then becomes:

$$\begin{aligned} XY &= x_3y_32^6 + \sum_{i=0}^2 \bar{x}_iy_32^{i+3} + 2^3 - 2^6 + \sum_{j=0}^2 \bar{x}_3y_j2^{j+3} + 2^3 - 2^6 + \sum_{i=0}^2 \sum_{j=0}^2 x_iy_j2^{i+j} \\ &= x_3y_32^6 + \sum_{i=0}^2 \bar{x}_iy_32^{i+3} + \sum_{j=0}^2 \bar{x}_3y_j2^{j+3} + \sum_{i=0}^2 \sum_{j=0}^2 x_iy_j2^{i+j} + 2^4 - 2^7 \\ &= -2^7 + x_3y_32^6 + (\bar{x}_2y_3 + \bar{x}_3y_2)2^5 + (\bar{x}_1y_3 + \bar{x}_3y_1 + x_2y_2 + 1)2^4 \\ &\quad + (\bar{x}_0y_3 + \bar{x}_3y_0 + x_1y_2 + x_2y_1)2^3 + (x_0y_2 + x_1y_1 + x_2y_0)2^2 + (x_0y_1 + x_1y_0)2^1 \\ &\quad + (x_0y_0)2^0 \end{aligned}$$



- Performance of arithmetic blocks dictate the performance of a digital system
- Architectural and logic transformations can enable significant speed up (e.g., adder delay from  $O(N)$  to  $O(\log_2(N))$ )
- Similar concepts and formulation can be applied at the system level
- **Timing analysis is tricky**: watch out for false paths!
- Area-Delay trade-offs (serial vs. parallel implementations)