L8/9: Arithmetic Structures

Acknowledgements:

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

- **Rex Min**
- **Kevin Atkinson**
- **Prof. Randy Katz (Unified Microelectronics Corporation Distinguished Professor in Electrical Engineering and Computer Science at the University of California, Berkeley) and Prof. Gaetano Borriello (University of Washington Department of Computer Science & Engineering) from Chapter 2 of R. Katz, G. Borriello. Contemporary Logic Design. 2nd ed. Prentice-Hall/Pearson Education, 2005.**
- **J. Rabaey, A. Chandrakasan, B. Nikolic, *Digital Integrated Circuits: A Design Perspective* Prentice Hall/Pearson, 2003.**

Number Systems Basics

How to represent negative numbers?

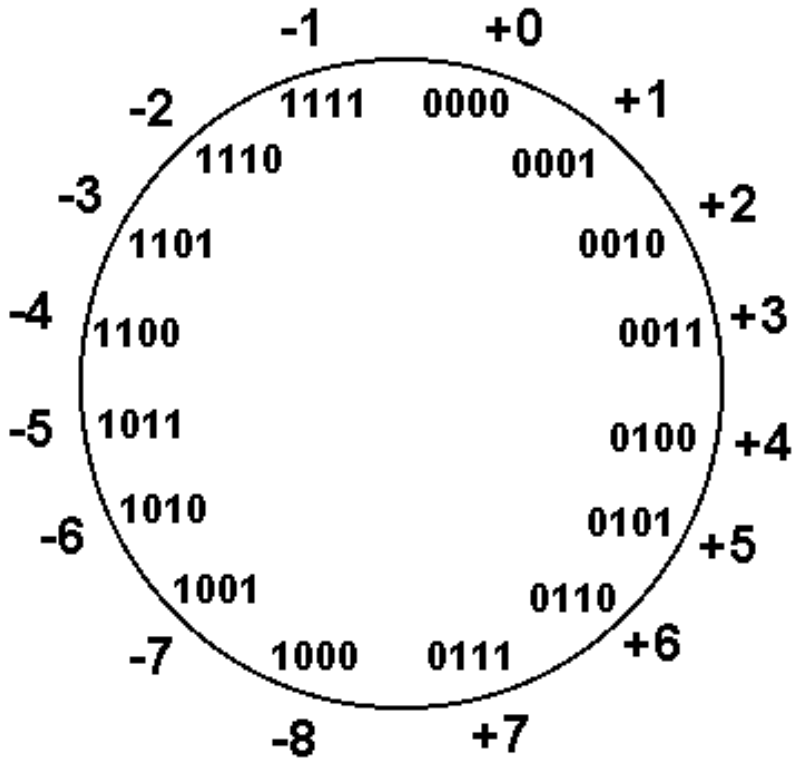
- Three common schemes: sign-magnitude, ones complement, twos complement
- Sign-magnitude: MSB = 0 for positive, 1 for negative
  - Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  - Two representations for zero: 0000… & 1000…
  - Simple multiplication but complicated addition/subtraction
- Ones complement: 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 Representation

Twos complement = bitwise complement + 1

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

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



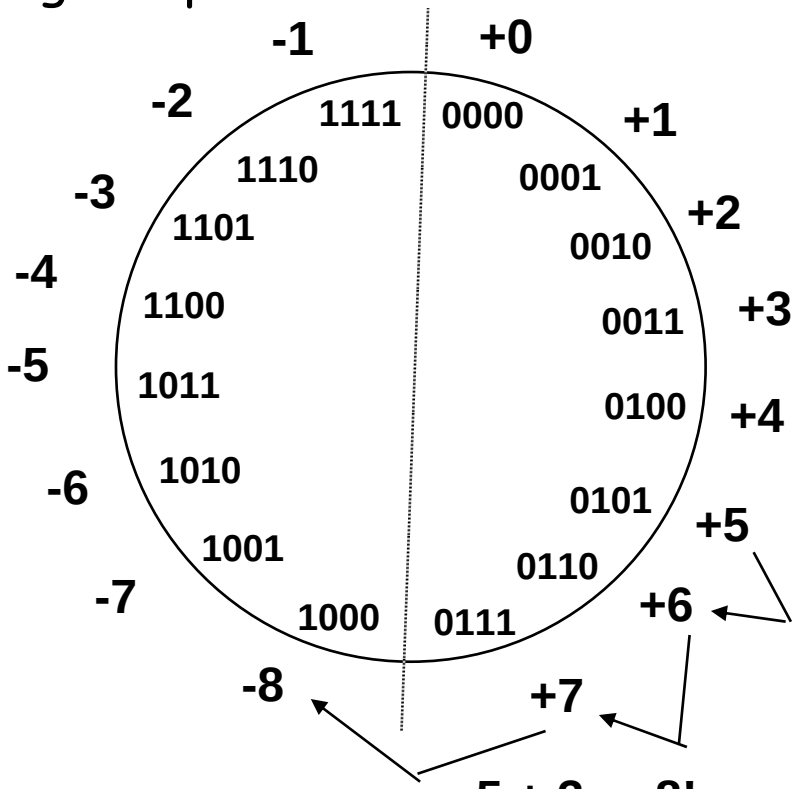
- Asymmetric range: -2^{N-1} to $+2^{N-1}-1$
- Only one representation for zero
- Simple addition and subtraction
- Most common representation

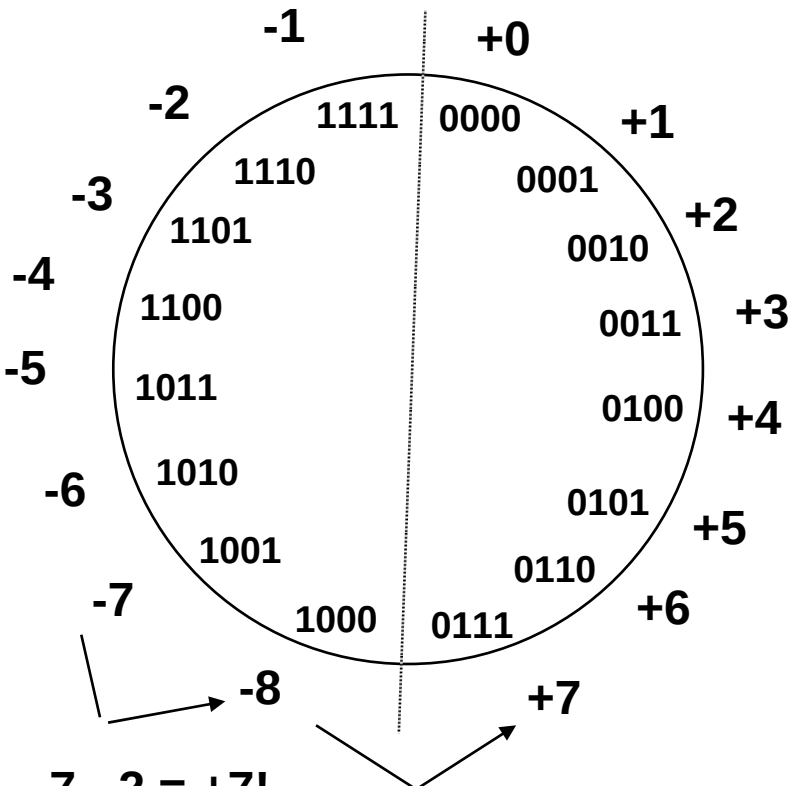
4	0100	-4	1100	4	0100	-4	1100
+ 3	0011	+ (-3)	1101	- 3	1101	+ 3	0011
7	0111	-7	11001	1	10001	-1	1111

[Katz05]

Overflow Conditions

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

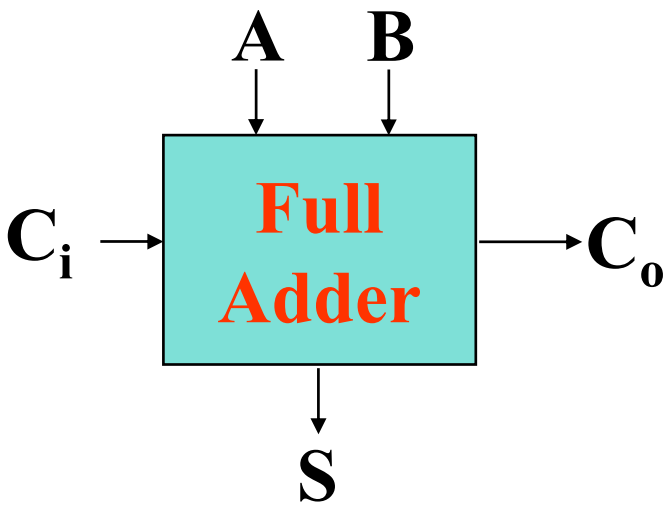




	0 1 1 1		1 0 0 0
5	0 1 0 1	-7	1 0 0 1
3	0 0 1 1	-2	1 1 0 0
-8	0 1 0 0 0	7	1 0 1 1 1

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

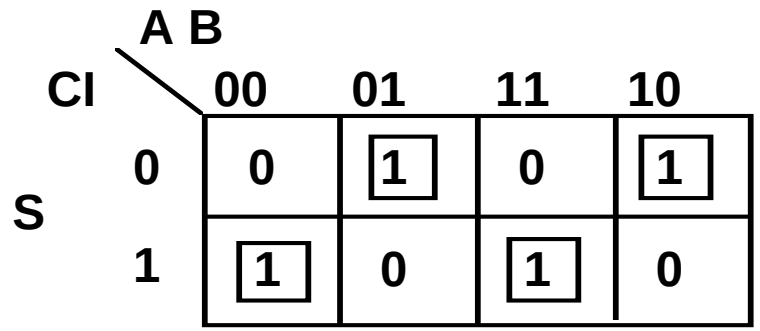
Binary Full Adder

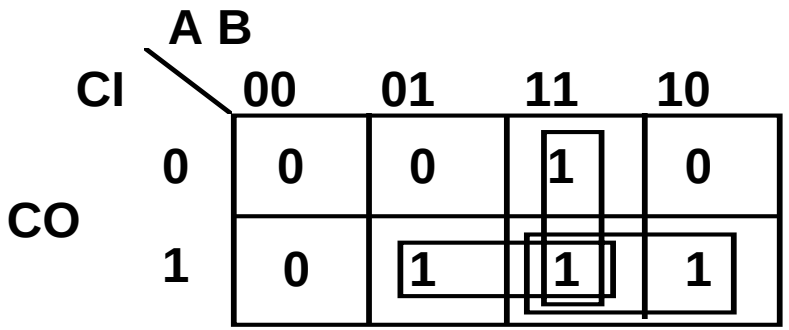


$$S = A \oplus B \oplus C_i$$
$$= A\bar{B}\bar{C}_i + \bar{A}B\bar{C}_i + \bar{A}\bar{B}C_i + ABC_i$$

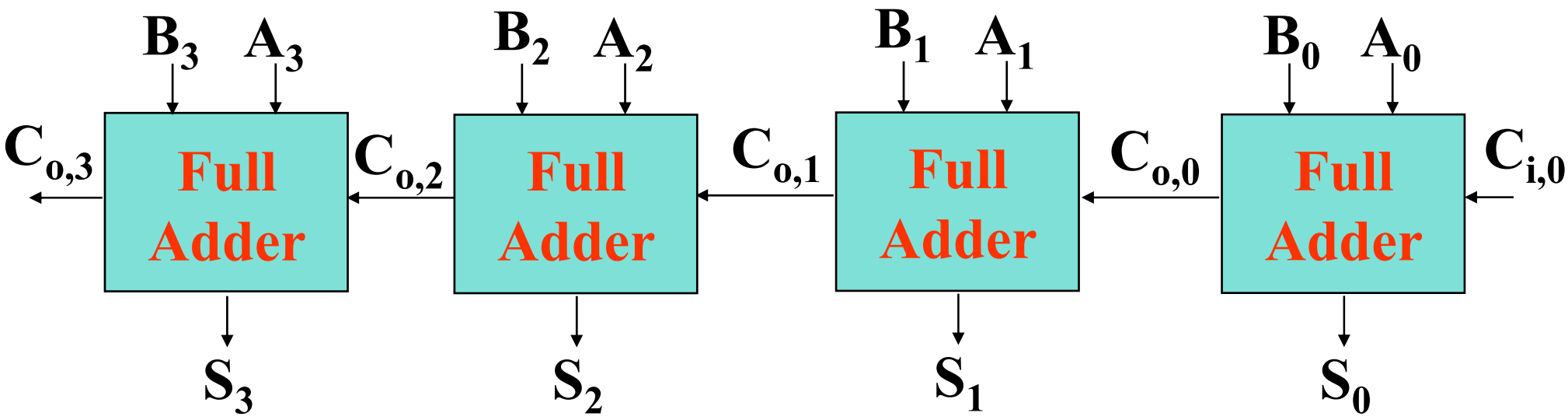
$$C_o = AB + C_i(A+B)$$

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





Ripple Carry Adder Structure



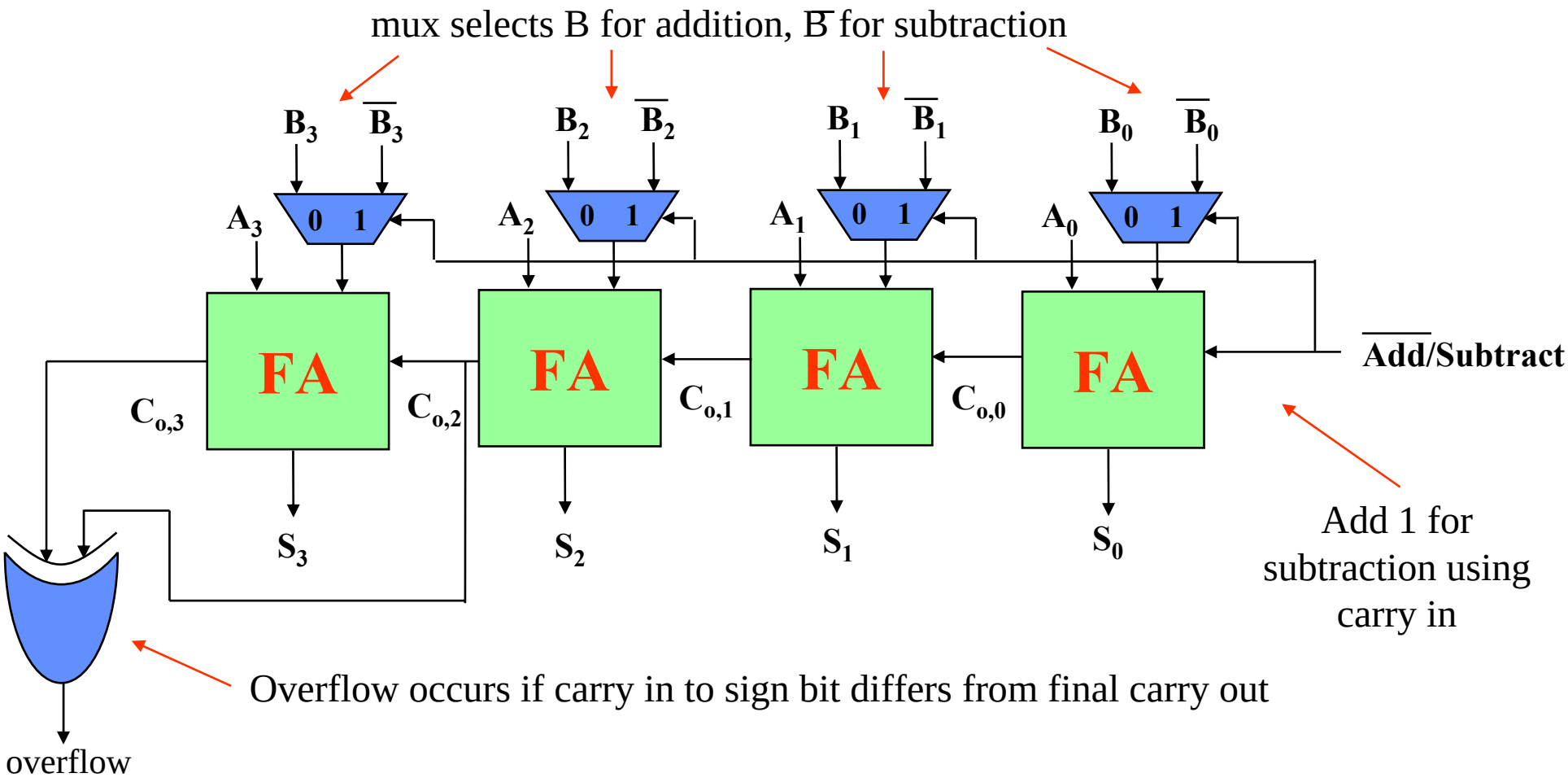
Worst case propagation delay linear with the number of bits

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

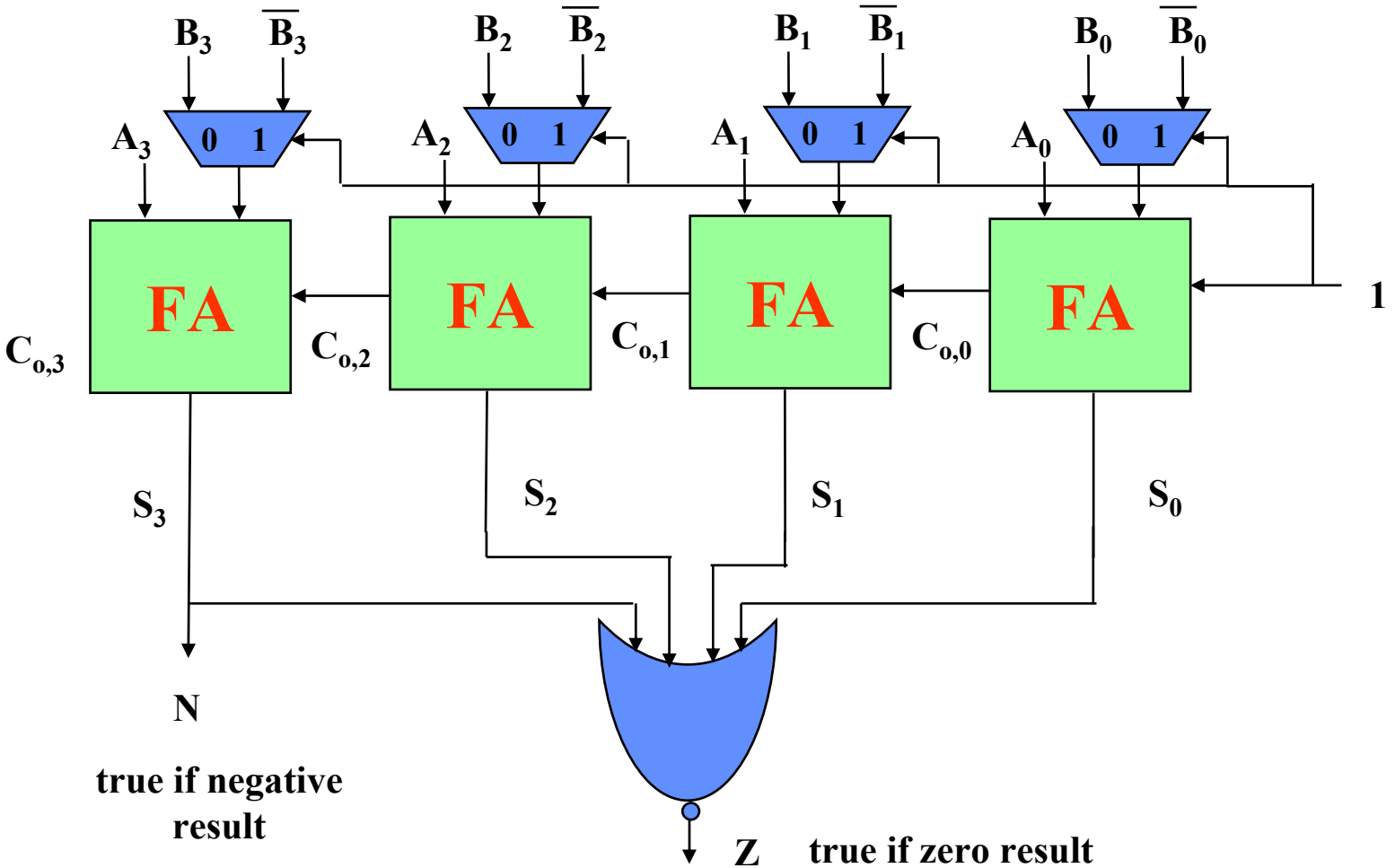
Extension to Subtraction

- **Under twos complement, subtracting B is the same as adding the bitwise complement of B then adding 1**

Combination addition/subtraction system:



Comparator (one approach)



$A < B = N$

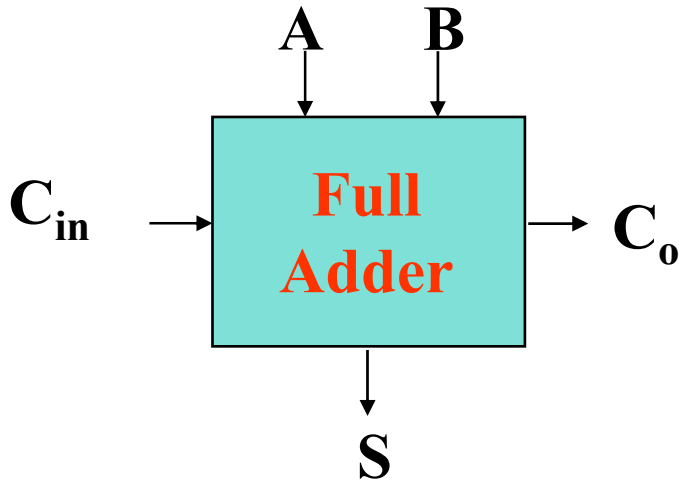
$A = B = Z$

$A \leq B = Z + N$

Alternate Adder Logic Formulation

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



Generate (G) = AB

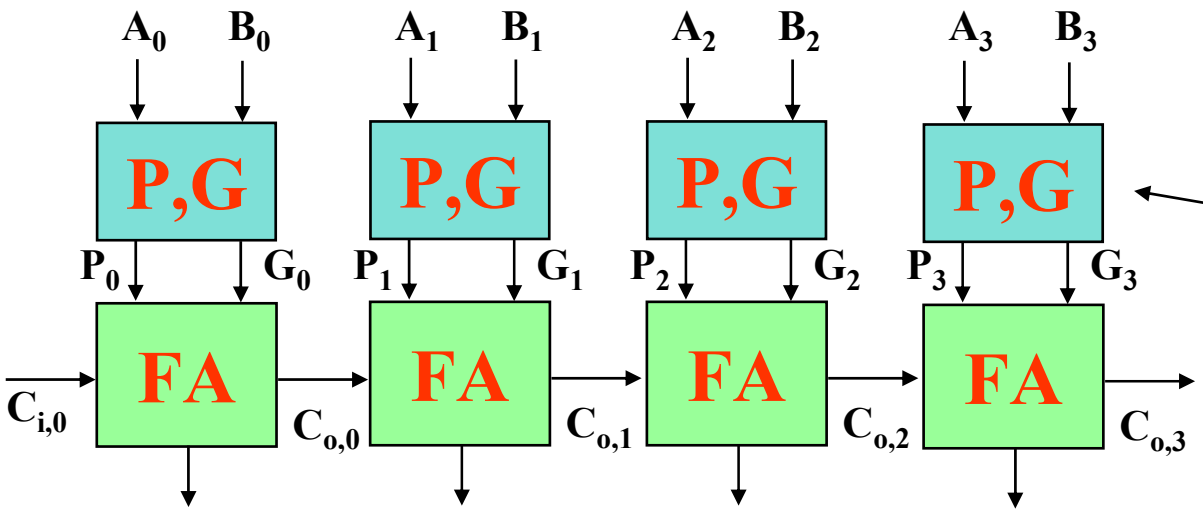
Propagate (P) = A ⊕ B

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

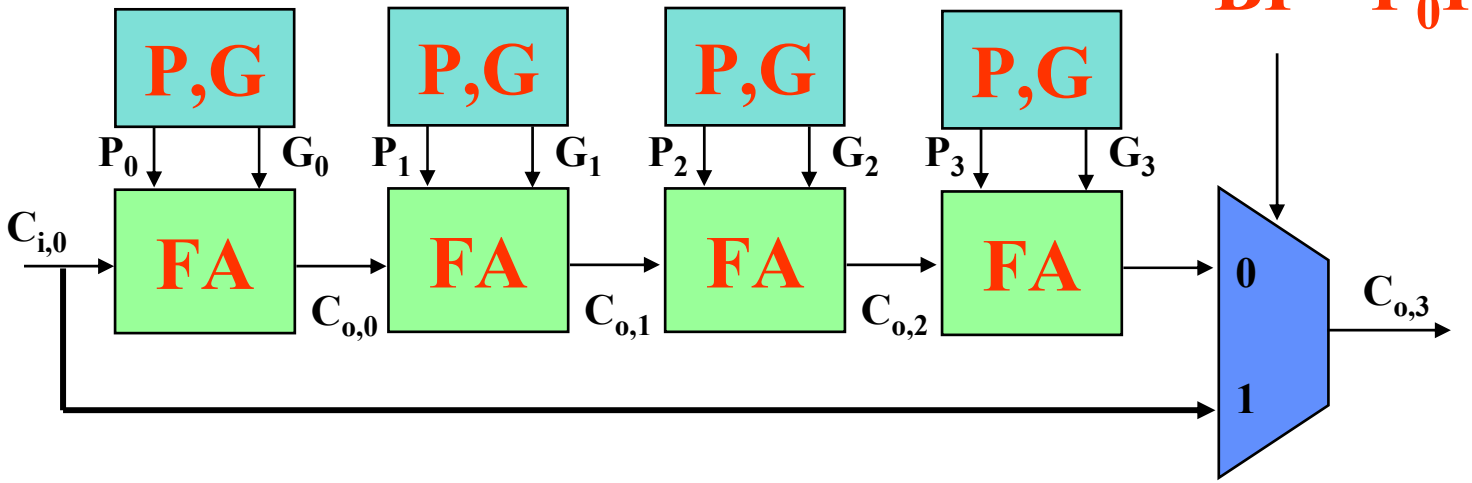
$$S(G,P) = P \oplus C_i$$

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

Carry Bypass Adder

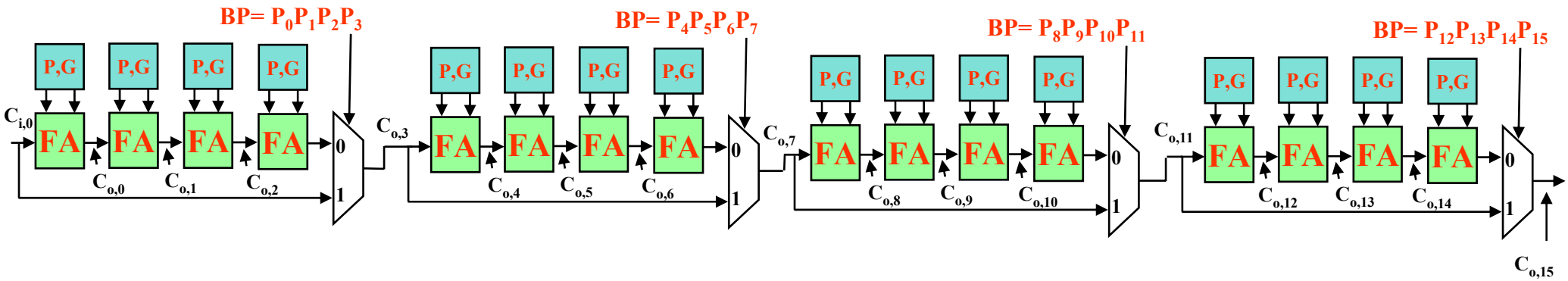


Can compute P, G in parallel for all bits



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

16-bit Carry Bypass Adder



Assume the following for delay each gate:

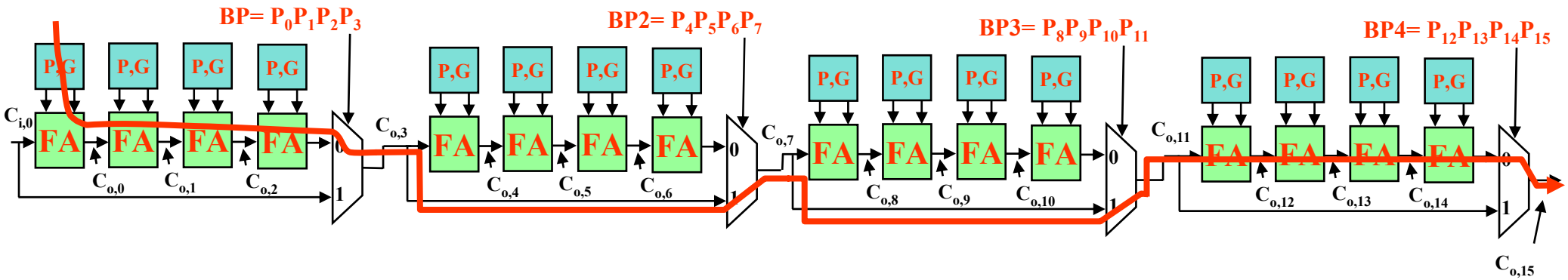
P, G from A, B: 1 delay unit

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?

Critical Path Analysis



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*

Carry Lookahead Adder

Re-express the carry logic as follows:

$$C1 = G0 + P0\ C0$$

$$C2 = G1 + P1\ C1 = G1 + P1\ G0 + P1\ P0\ C0$$

$$C3 = G2 + P2\ C2 = G2 + P2\ G1 + P2\ P1\ G0 + P2\ P1\ P0\ C0$$

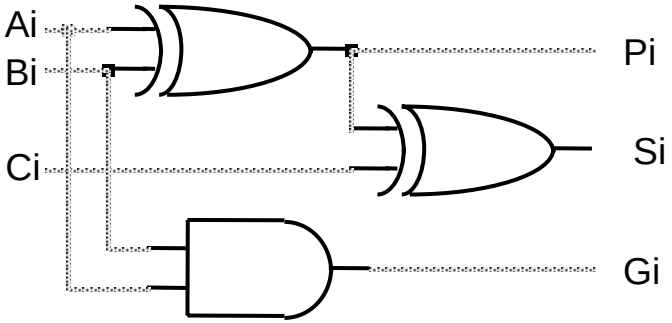
$$C4 = G3 + P3\ C3 = G3 + P3\ G2 + P3\ P2\ G1 + P3\ P2\ P1\ G0 + P3\ P2\ P1\ P0\ C0$$

…

- 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

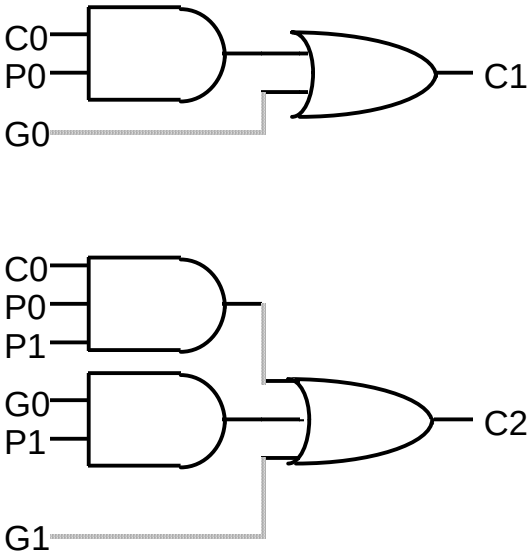
Ripple effect has been eliminated!

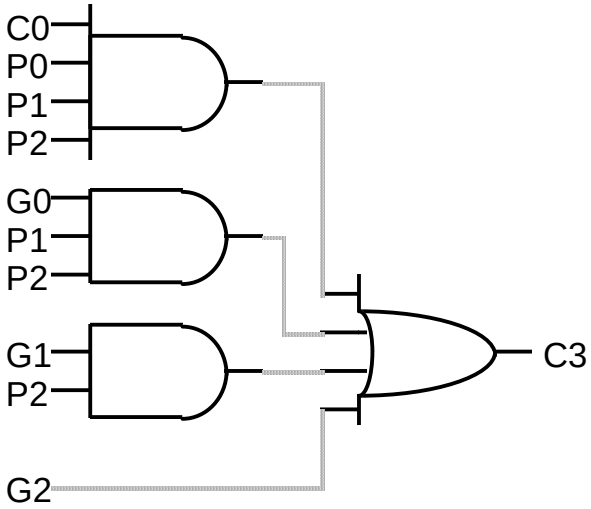
Carry Lookahead Logic

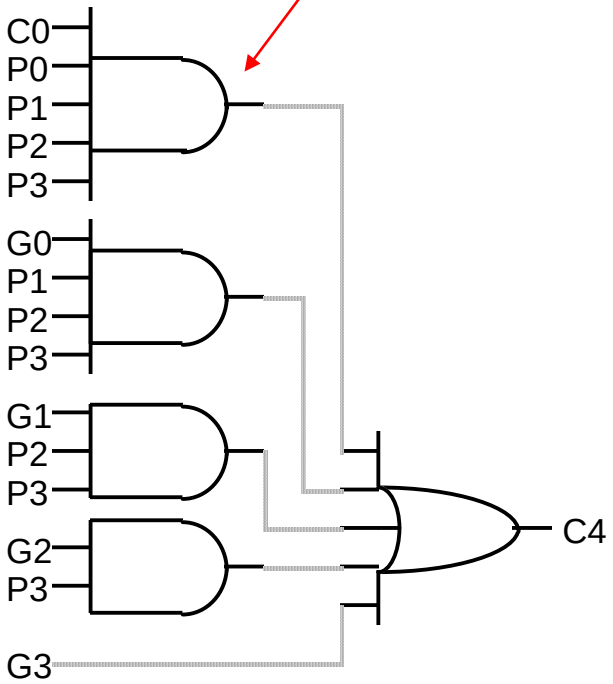


Adder with propagate and generate outputs

Later stages have increasingly complex logic







Block Generate and Propagate

$G_{j:i}$ 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. $G_{3:2} = G_3 + P_3G_2$. $P_{3:2}$ is true if an incoming carry propagates through both bit positions 2 and 3. $P_{3:2} = P_3P_2$

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

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

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

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

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.

More Definitions…

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

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

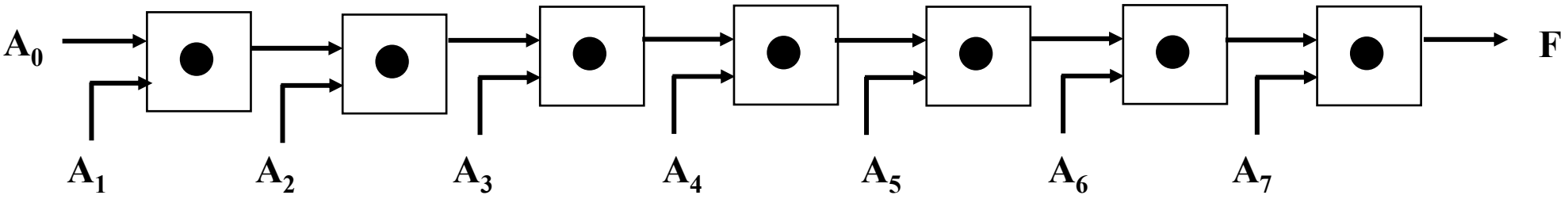
$$(G_{3:2}, P_{3:2}) = (G_3, P_3) \bullet (G_2, P_2)$$

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

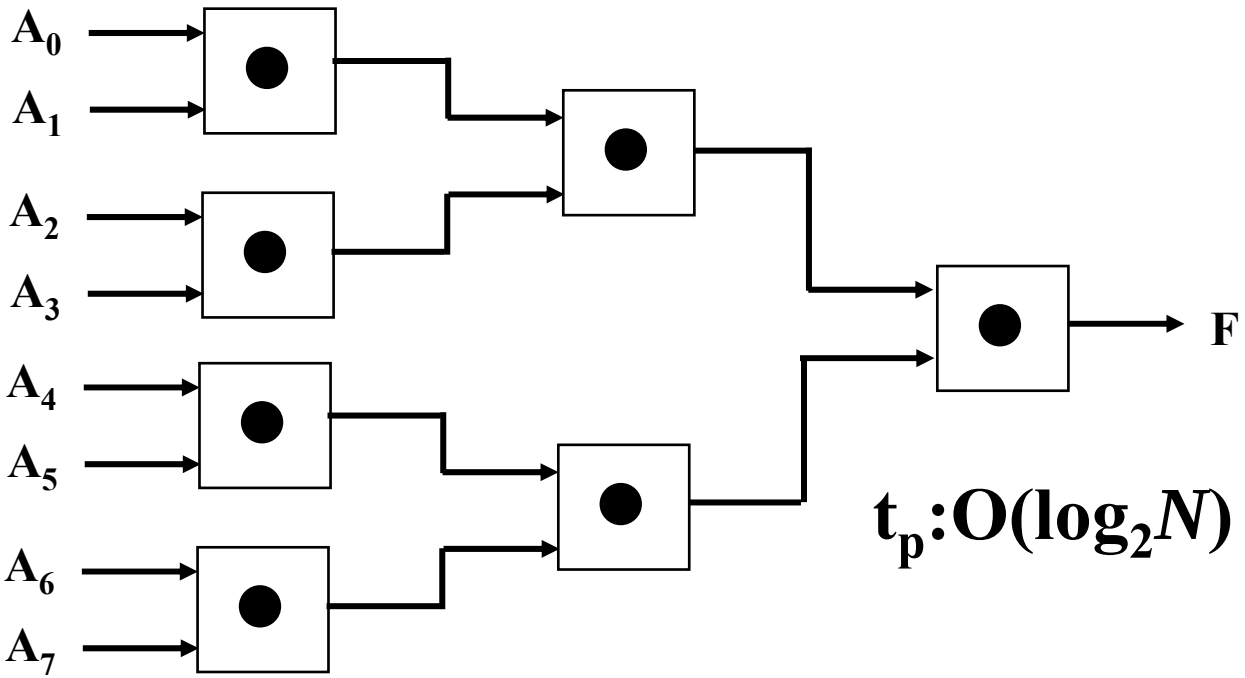
$$\begin{aligned}(G_{3:0}, P_{3:0}) &= [(G_3, P_3) \bullet (G_2, P_2)] \bullet [(G_1, P_1) \bullet (G_0, P_0)] \\ &= (G_{3:2}, P_{3:2}) \bullet (G_{1:0}, P_{1:0})\end{aligned}$$

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

Logarithmic Look-Ahead Adder

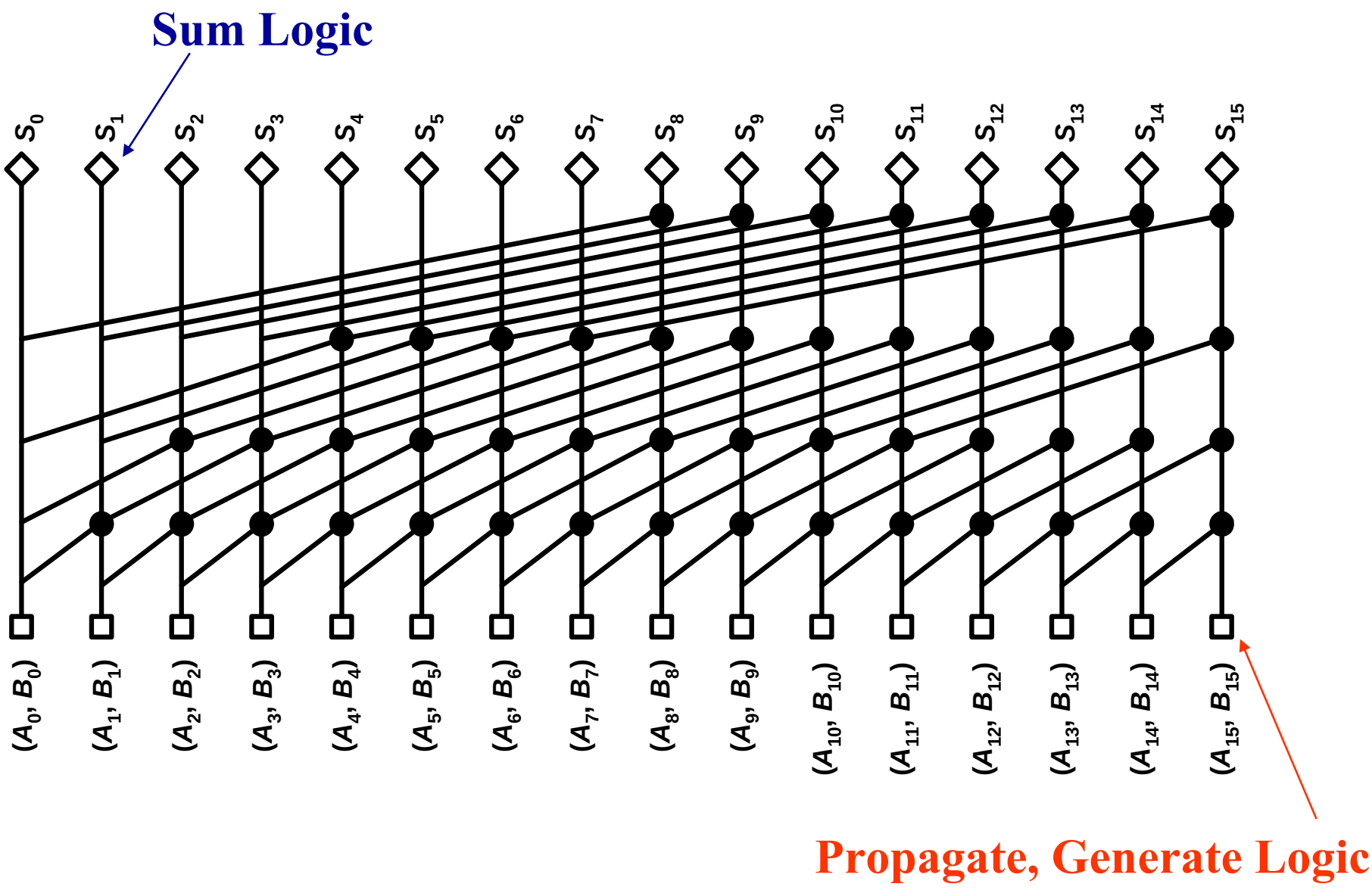


t_p: O(N)

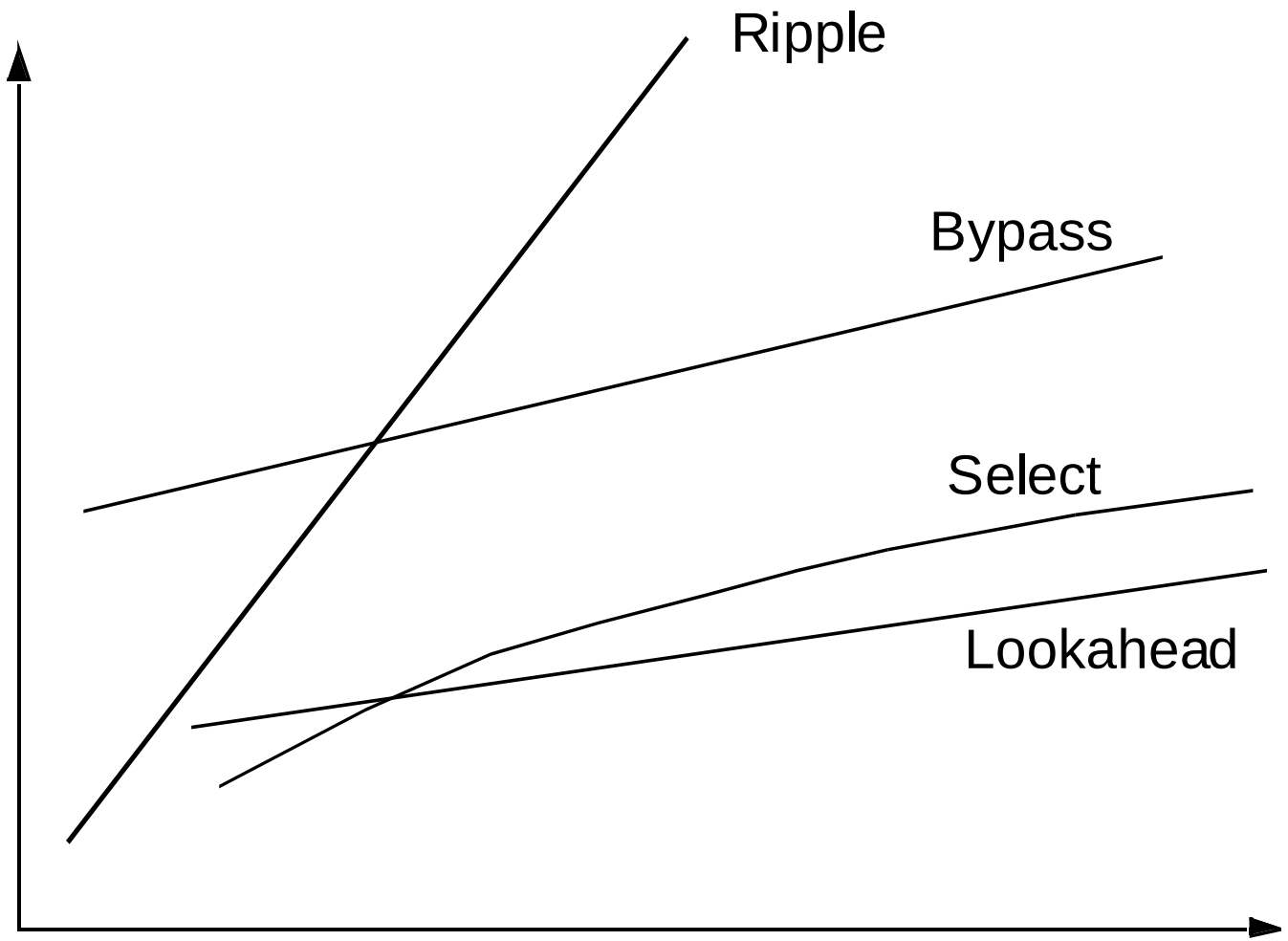


t_p:O($\log_2 N$)

16-bit Kogge-Stone Tree Adder

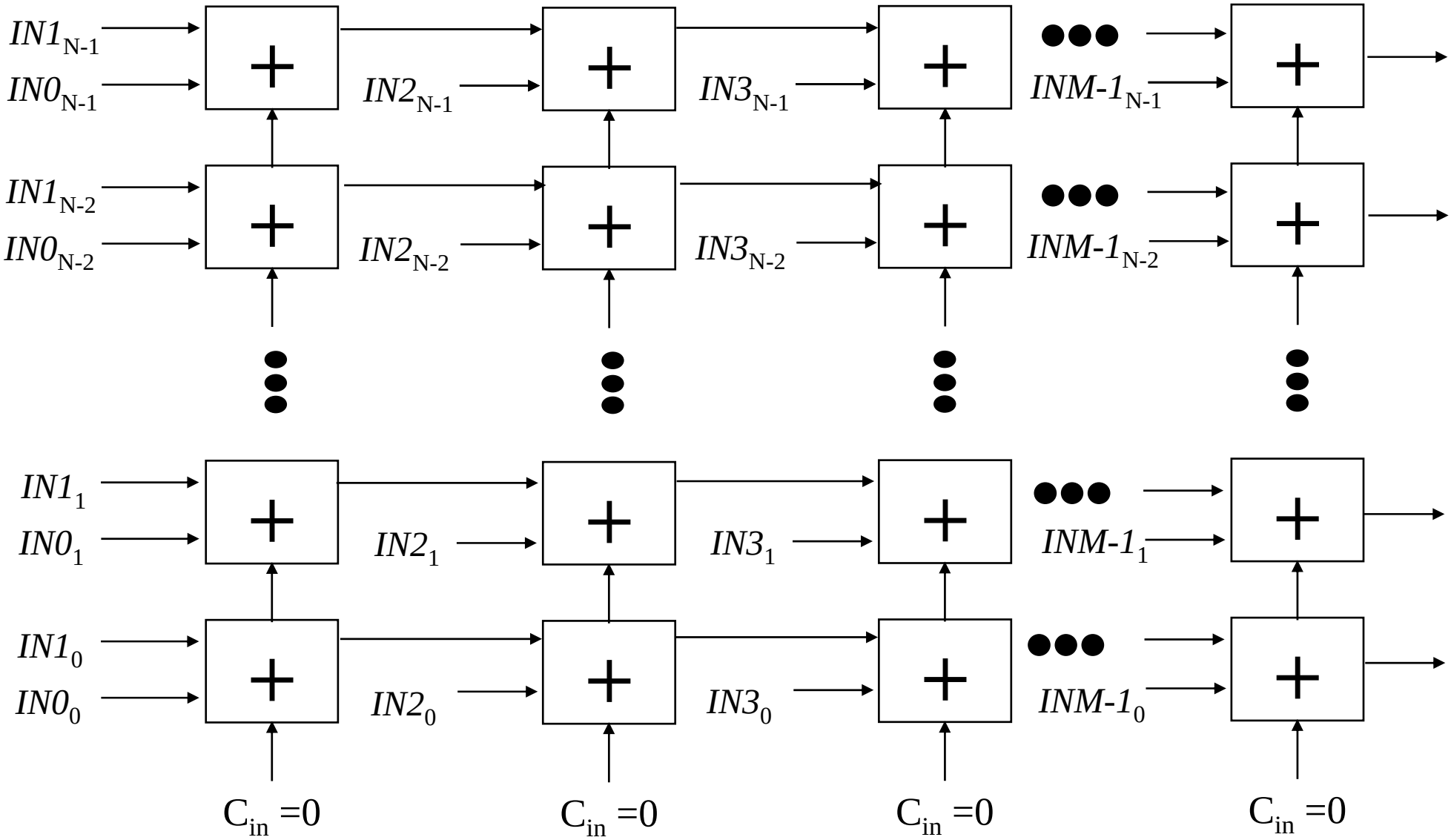


Adder Performance

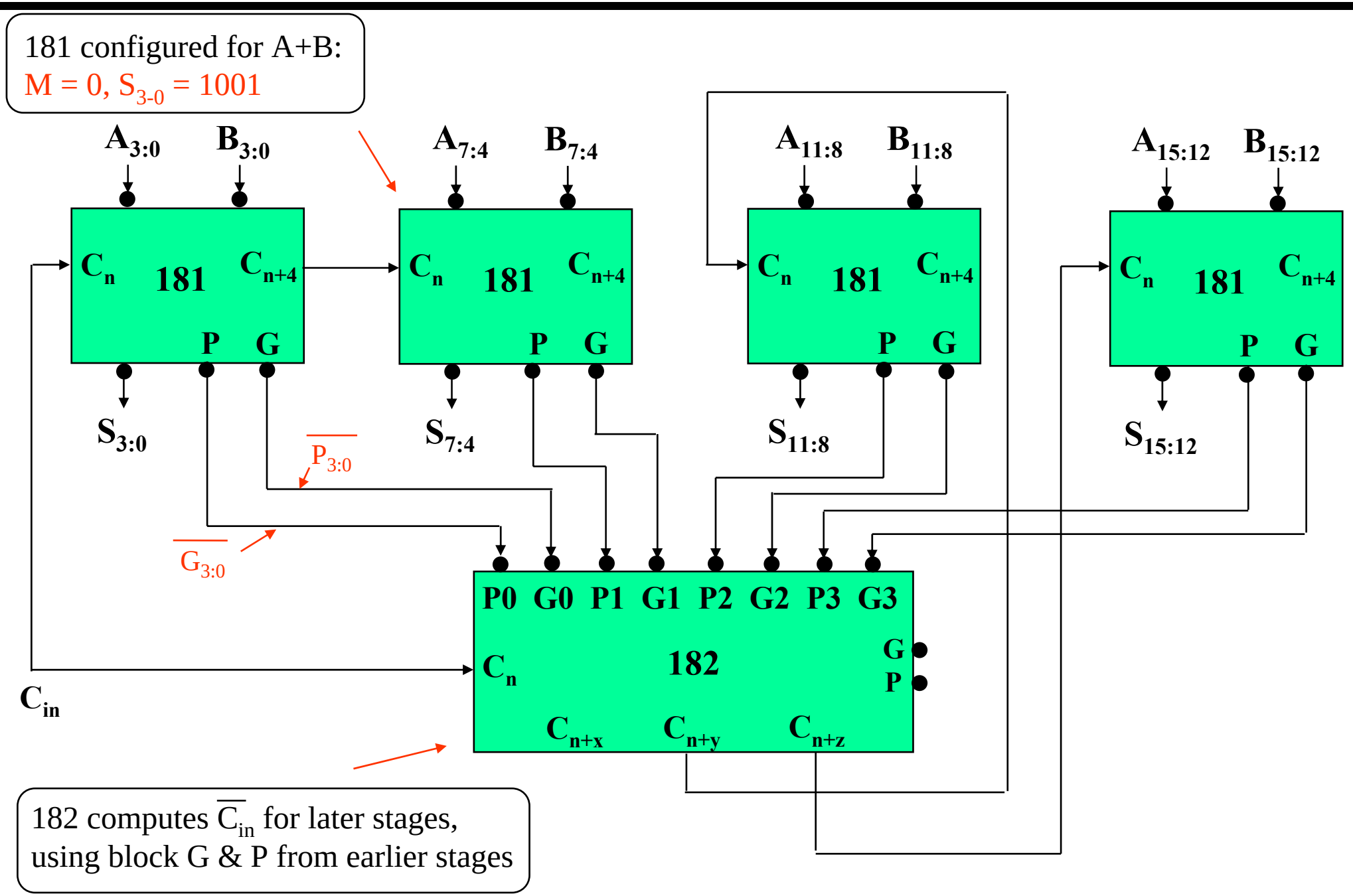


Delay vs. number of bits

Addition of M, N-bit Numbers



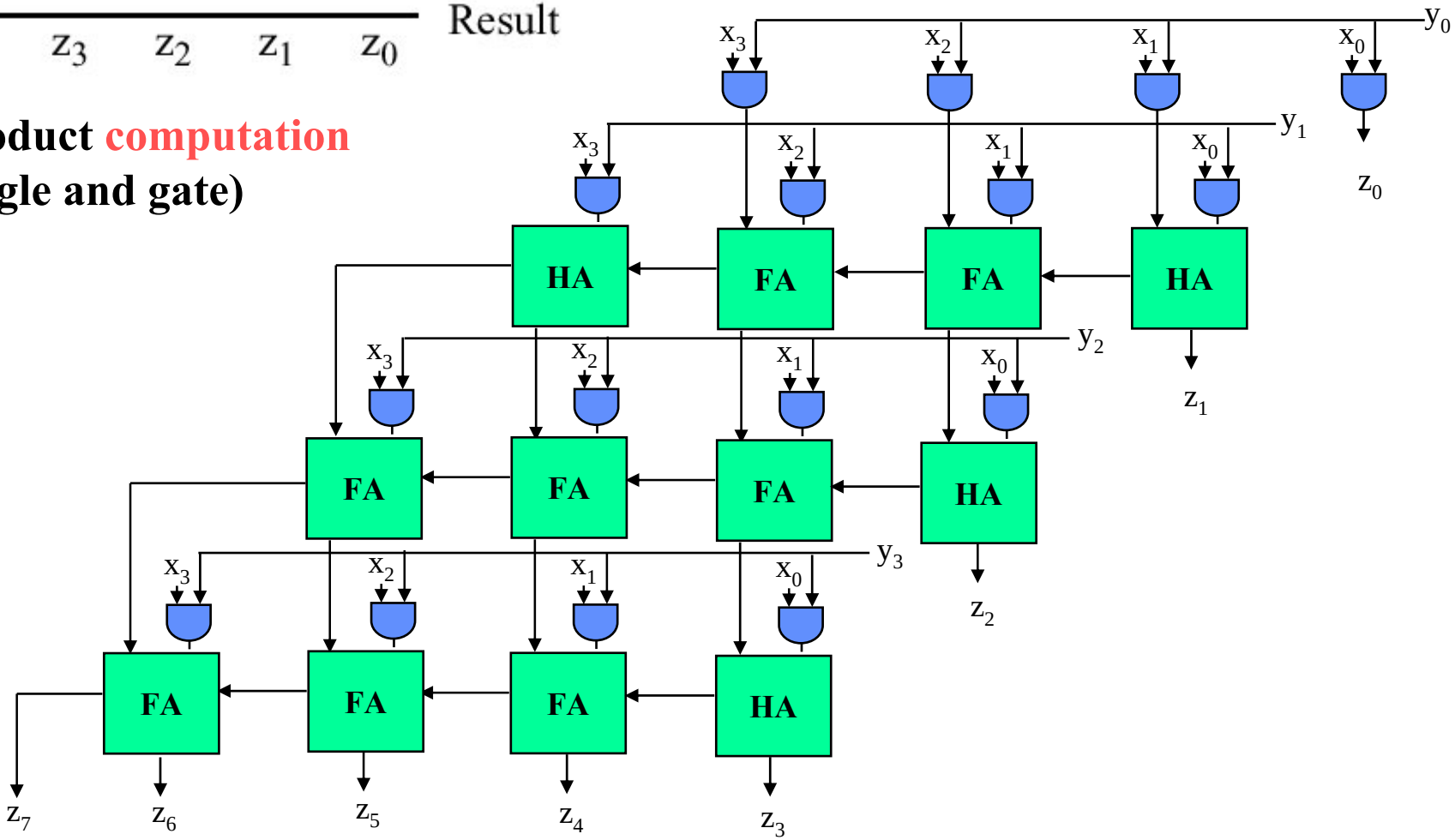
16-bit Carry Lookahead Schematic



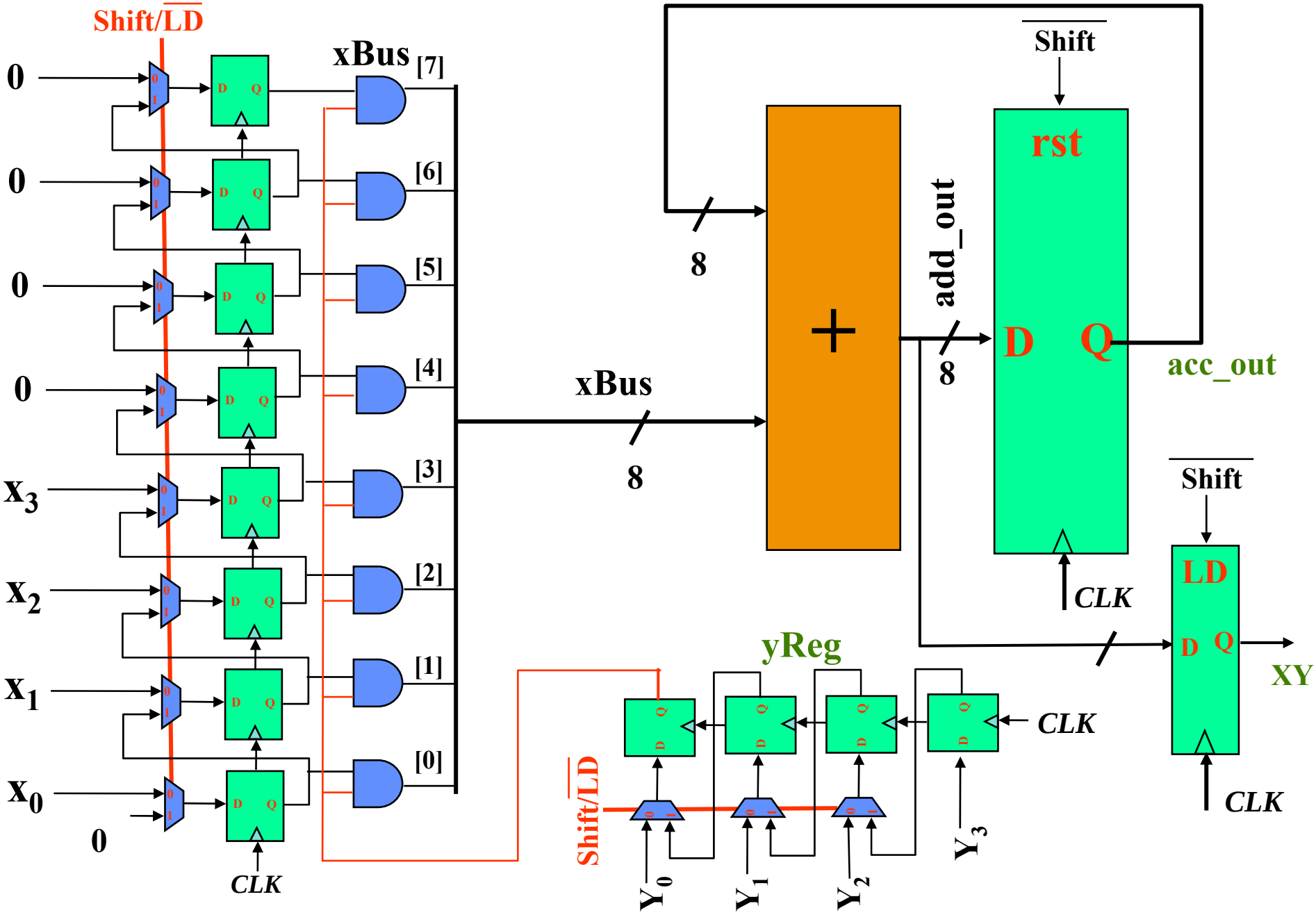
Binary Multiplication

				x_3	x_2	x_1	x_0		Multiplicand
			×	y_3	y_2	y_1	y_0		Multiplier
			x_3y_0	x_2y_0	x_1y_0	x_0y_0			
		x_3y_1	x_2y_1	x_1y_1	x_0y_1				Partial Product
	x_3y_2	x_2y_2	x_1y_2	x_0y_2					
+	x_3y_3	x_2y_3	x_1y_3	x_0y_3					
z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0		Result

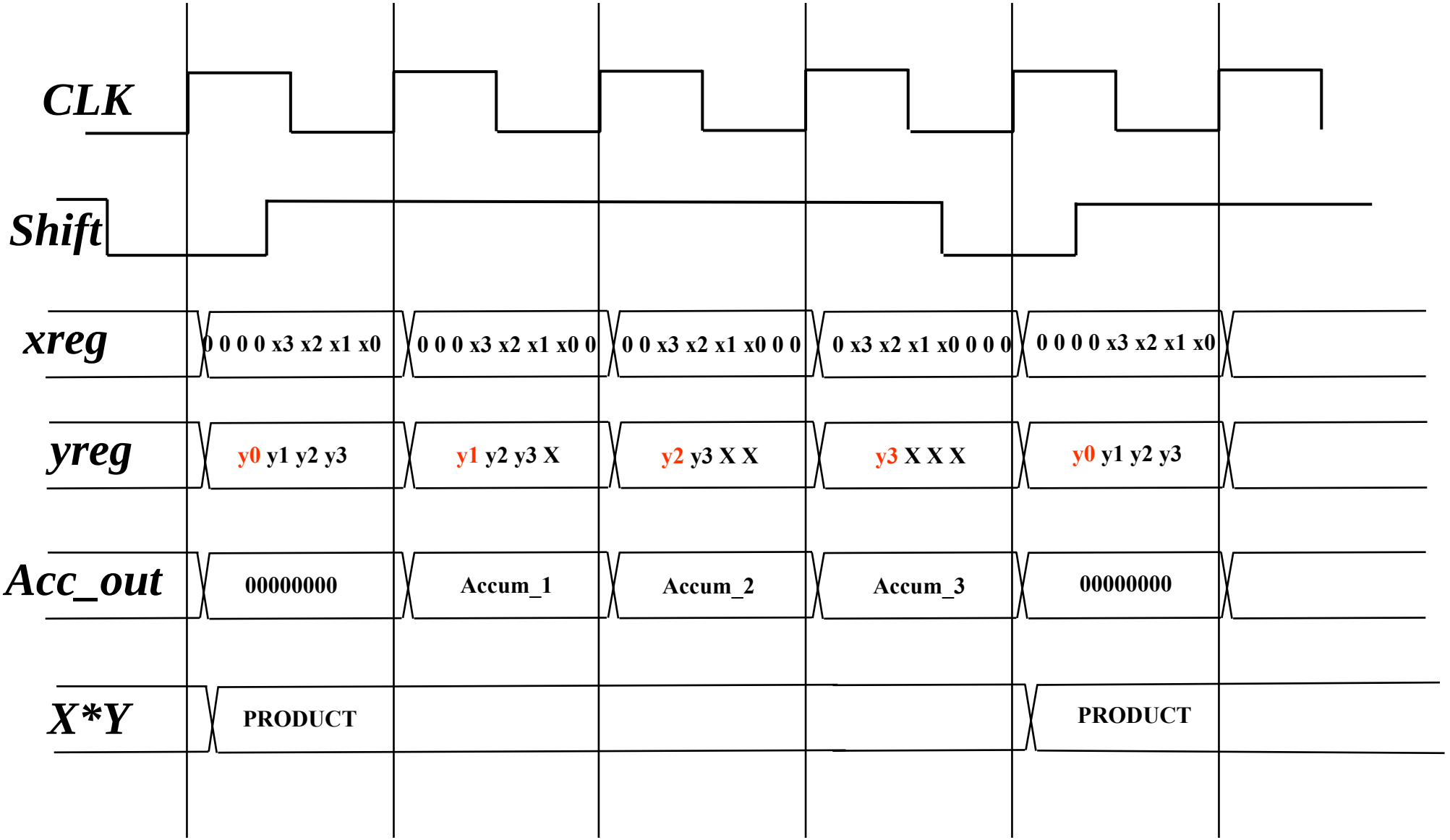
➢ **Partial product computation is simple (single and gate)**



A Serial (Magnitude) Multiplier



Timing Diagram

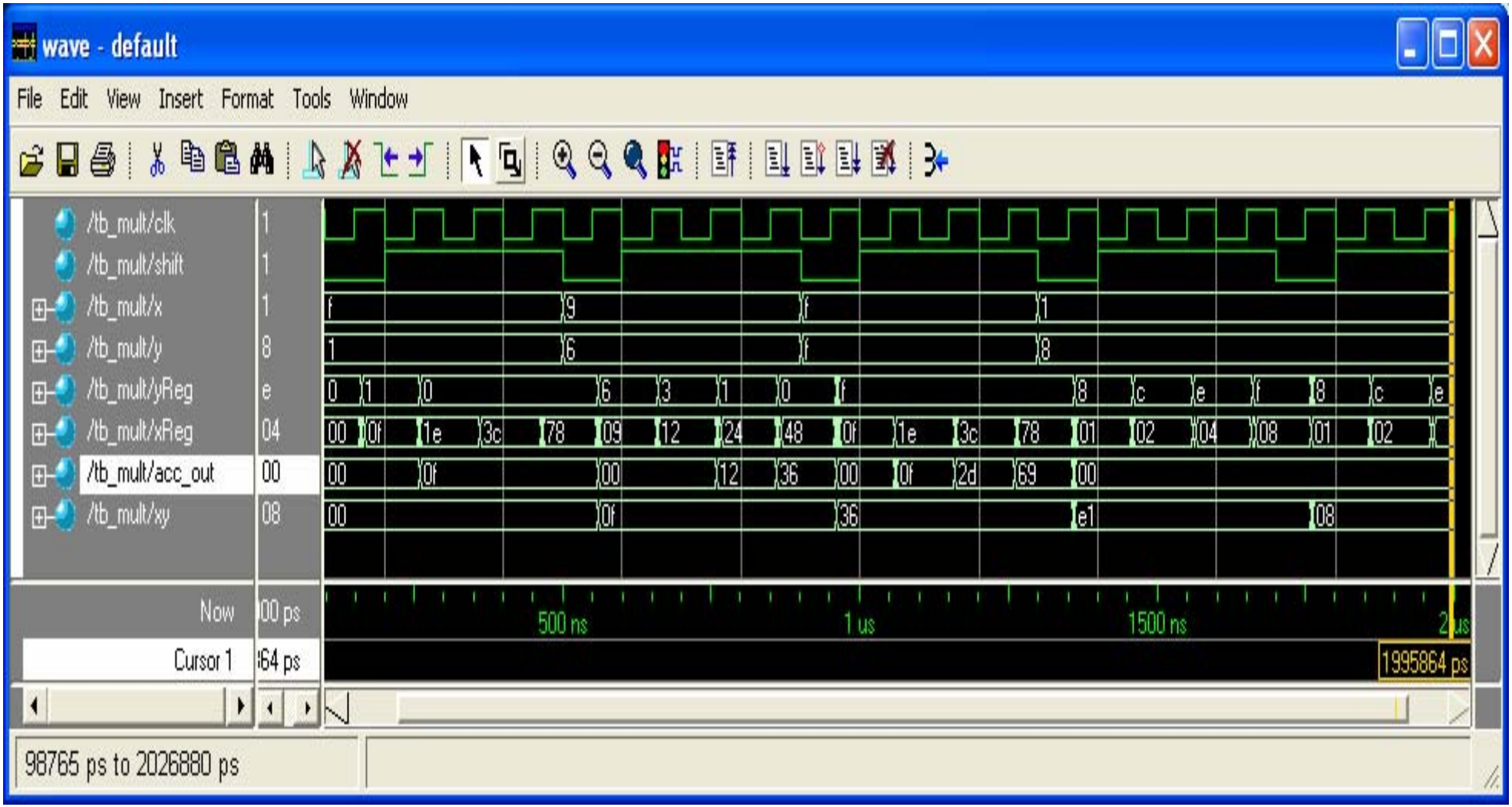


Verilog of Serial Multiplier

```
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
```

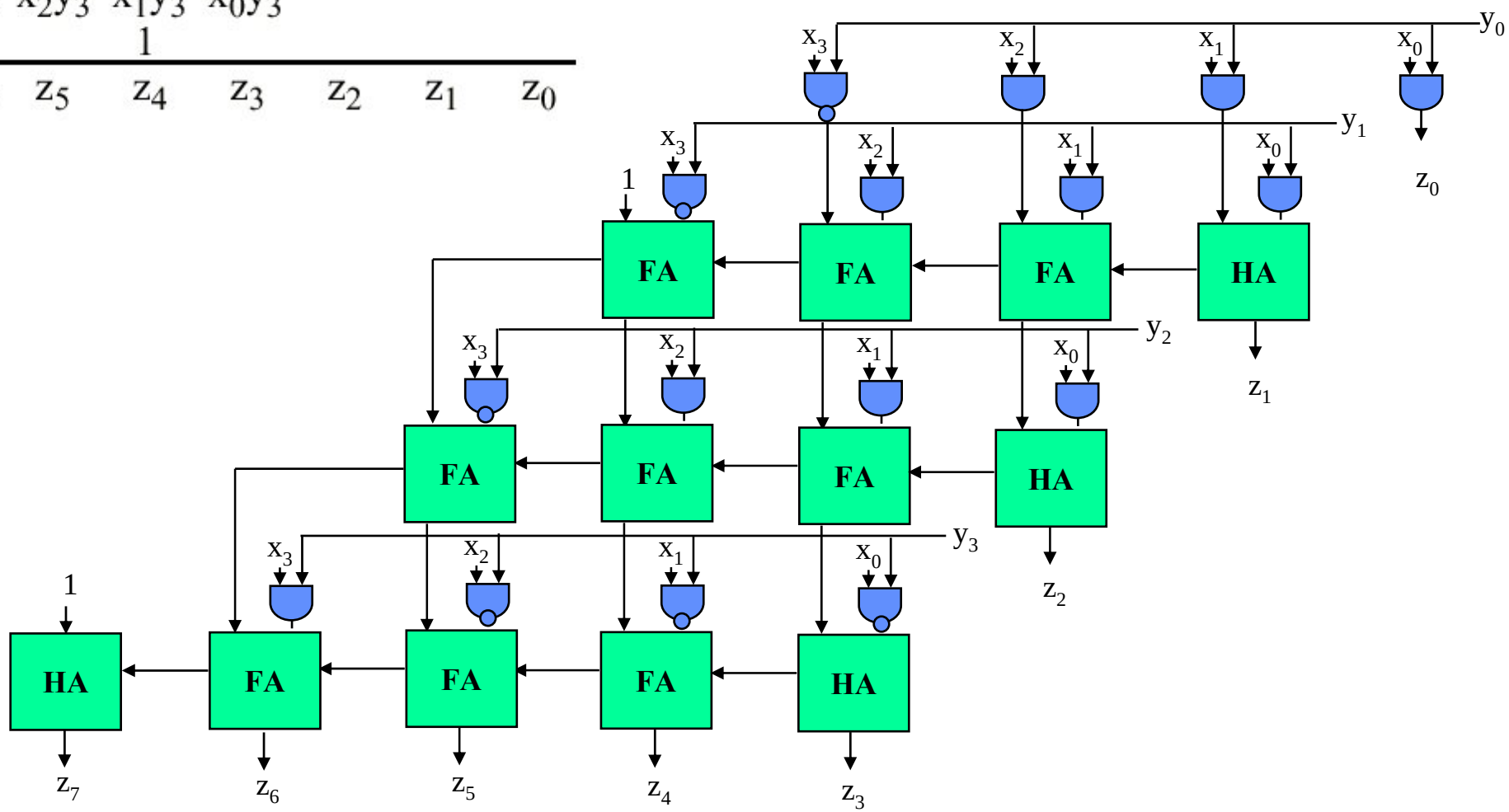
```
always @ (posedge clk)

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
```

Simulation


wave - default
File Edit View Insert Format Tools Window
/tb_mult/clk
/tb_mult/shift
/tb_mult/x
/tb_mult/y
/tb_mult/yReg
/tb_mult/xReg
/tb_mult/acc_out
/tb_mult/xy
Now
Cursor 1
500 ns
1 us
1500 ns
1995864 ps
98765 ps to 2026880 ps

Twos Complement Multiplication

$$
\begin{array}{rcccccccc}
 & & & & x_3 & x_2 & x_1 & x_0 & \text{Multiplicand} \\
 & & & \times & y_3 & y_2 & y_1 & y_0 & \text{Multiplier} \\
\hline
 & & & \overline{x_3y_0} & x_2y_0 & x_1y_0 & x_0y_0 & & \\
 & & \overline{x_3y_1} & x_2y_1 & x_1y_1 & x_0y_1 & & & \\
 & \overline{x_3y_2} & x_2y_2 & x_1y_2 & x_0y_2 & & & & \\
x_3y_3 & \overline{x_2y_3} & \overline{x_1y_3} & \overline{x_0y_3} & & & & & \\
+1 & & & 1 & & & & & \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0 &
\end{array}
$$



Summary

- 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)