Datapath Components-Part I (Adder; Subtractor)

Jia Chen
jiac@ucr.edu

Datapath

- A datapath is a collection of functional units such as arithmetic logic units (ALU) or multipliers that perform data processing operations, registers, and buses.

- Along with the control unit it composes the central processing unit (CPU).

- A larger datapath can be made by joining more than one datapaths using multiplexers.

- A datapath is the ALU, the set of registers, and the CPU's internal bus(es) that allow data to flow between them.

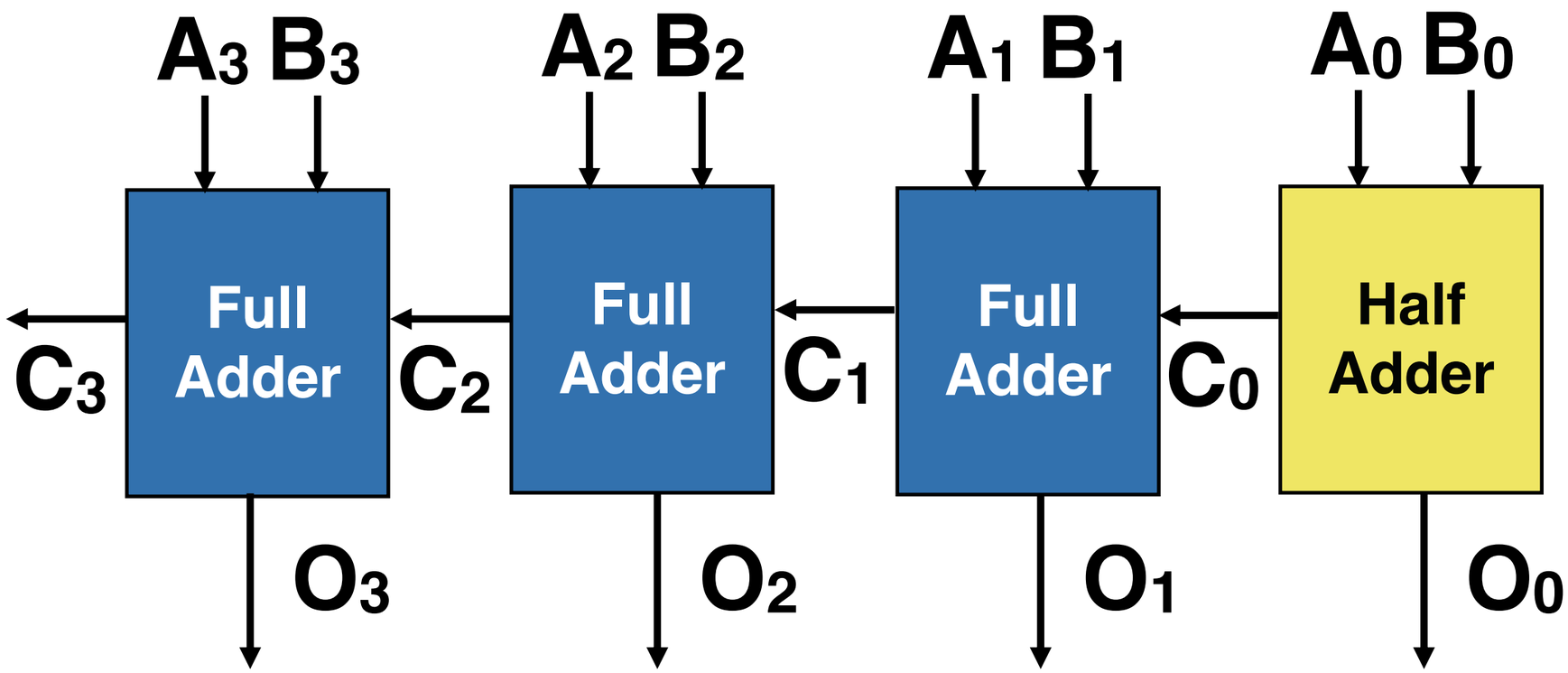
https://en.wikipedia.org/wiki/Datapath

Datapath components

Multiplexer	Decoder
Adder	Subtractor
Register	Encoder
Shifter	Counter
Multiplier	Divider
Comparator	...

Carry-Ripple Adder (CRA)

Revisit 4-bit adder



$$
\begin{array}{r}
[A_3A_2A_1A_0] \\
+\ [B_3B_2B_1B_0] \\
\hline
[C_3O_3O_2O_1O_0]
\end{array}
$$

Recap: Full Adder

Input			Output	
A	B	Cin	Out	Cout
0	0	0	**0**	**0**
0	1	0	**1**	**0**
1	0	0	**1**	**0**
1	1	0	**0**	**1**
0	0	1	**1**	**0**
0	1	1	**0**	**1**
1	0	1	**0**	**1**
1	1	1	**1**	**1**

		A'B'	A'B	AB	AB'
	Cout	0,0	0,1	1,1	1,0
Cin'	0	0	0	1	0
Cin	1	0	1	1	1

ACin

BCin AB

Cout = AB + ACin + BCin

		A'B'	A'B	AB	AB'
	Out	0,0	0,1	1,1	1,0
Cin'	0	0	1	0	1
Cin	1	1	0	1	0

Out = A'BCin' + AB'Cin' +A'B'Cin + ABCin

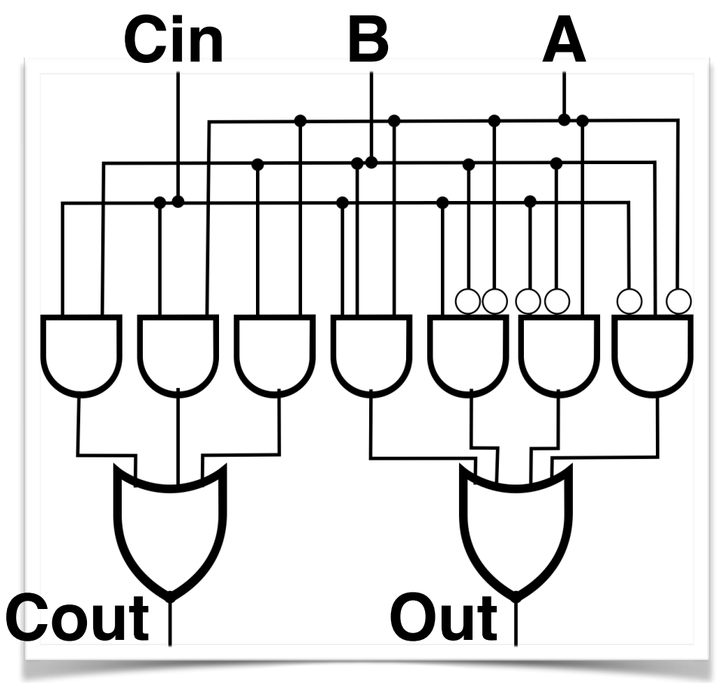
Slide adapted from Hung-Wei Tseng

Recap: Full Adder

Input			Output	
A	B	Cin	Out	Cout
0	0	0	**0**	**0**
0	1	0	**1**	**0**
1	0	0	**1**	**0**
1	1	0	**0**	**1**
0	0	1	**1**	**0**
0	1	1	**0**	**1**
1	0	1	**0**	**1**
1	1	1	**1**	**1**

Cout = AB + ACin + BCin

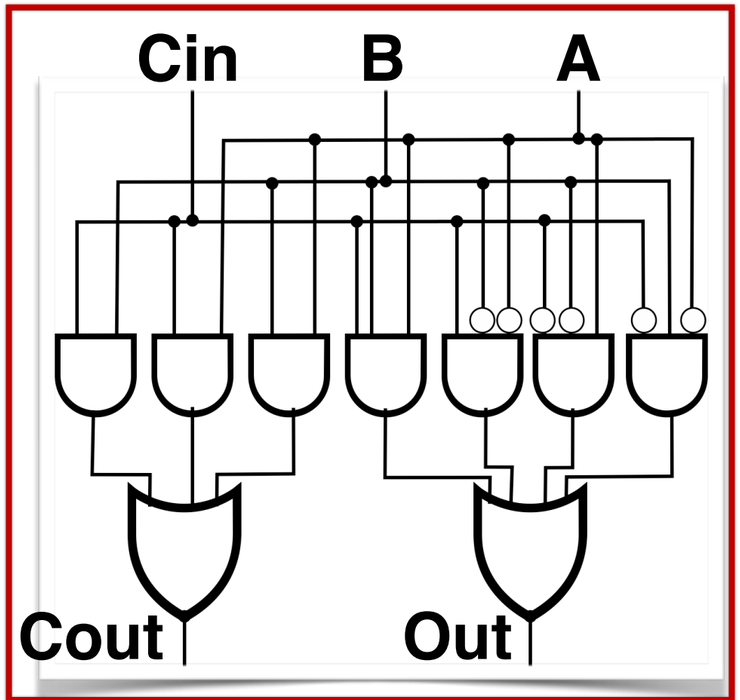
Out = A'BCin' + AB'Cin' +A'B'Cin + ABCin



How efficient is the adder?

- One approach estimates transistors, assuming every gate input requires 2 transistors, and ignoring inverters for simplicity. A 2-input gate requires 2 inputs · 2 trans/input = 4 transistors. A 3-input gate requires 3 · 2 = 6 transistors. A 4-input gate: 8 transistors. Wires also contribute to size, but ignoring wires as above is a common approximation.

- Considering the shown 1-bit full adder and use it to build a 32-bit adder, how many transistor do we need?
  A. 1152
  B. 1600
  C. 1664
  D. 1792
  E. 1984



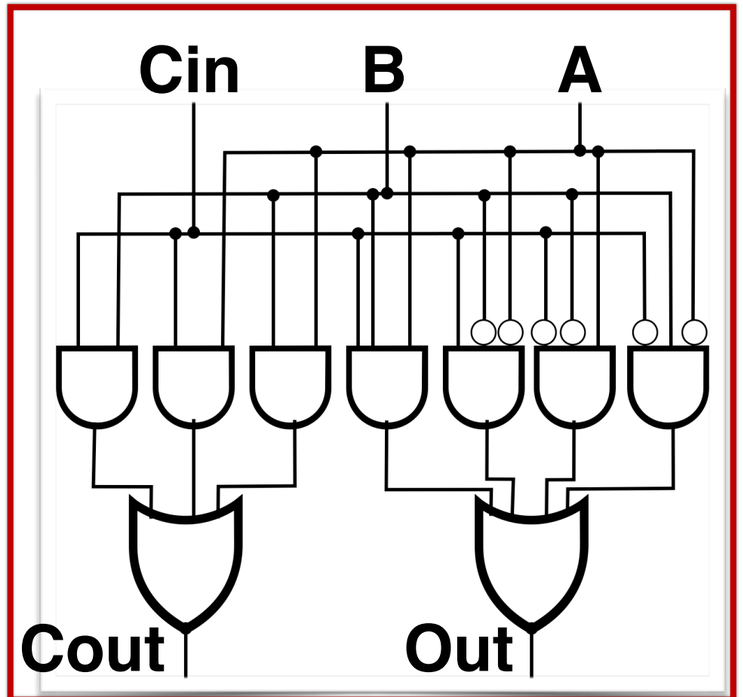
Slide adapted from Hung-Wei Tseng

How efficient is the adder?

- One approach estimates transistors, assuming every gate input requires 2 transistors, and ignoring inverters for simplicity. A 2-input gate requires 2 inputs· 2 trans/input = 4 transistors. A 3-input gate requires 3 · 2 = 6 transistors. A 4-input gate: 8 transistors. Wires also contribute to size, but ignoring wires as above is a common approximation.
- Considering the shown 1-bit full adder and use it to build a 32-bit adder, how many transistor do we need?

A. 1152
B. 1600
C. 1664
D. 1792
E. 1984

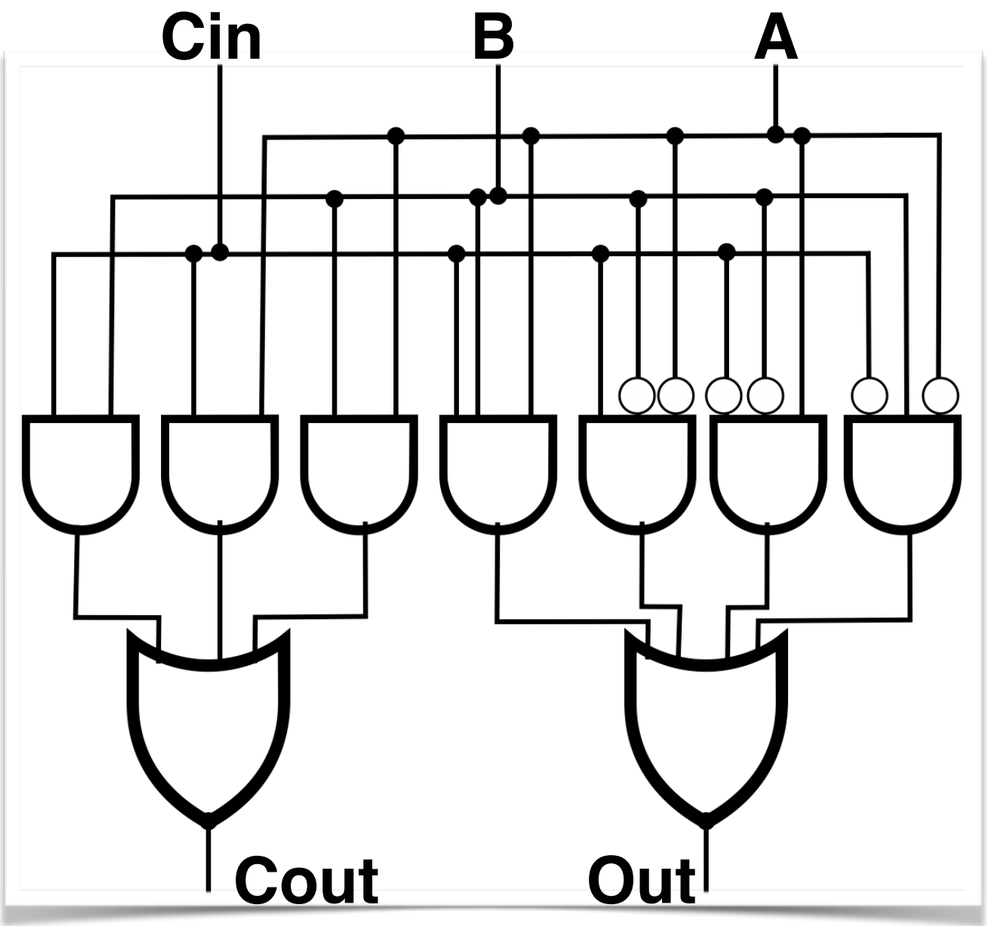
# of 2-input gates: 3
# of 3-input gates: 5
# of 4-input gates: 1
# of transistors for 1-bit adder = 3*4 + 5*6 + 1*8 = 50
# of transistors for a 32-bit adder = 50*32 = 1600



Slide adapted from Hung-Wei Tseng

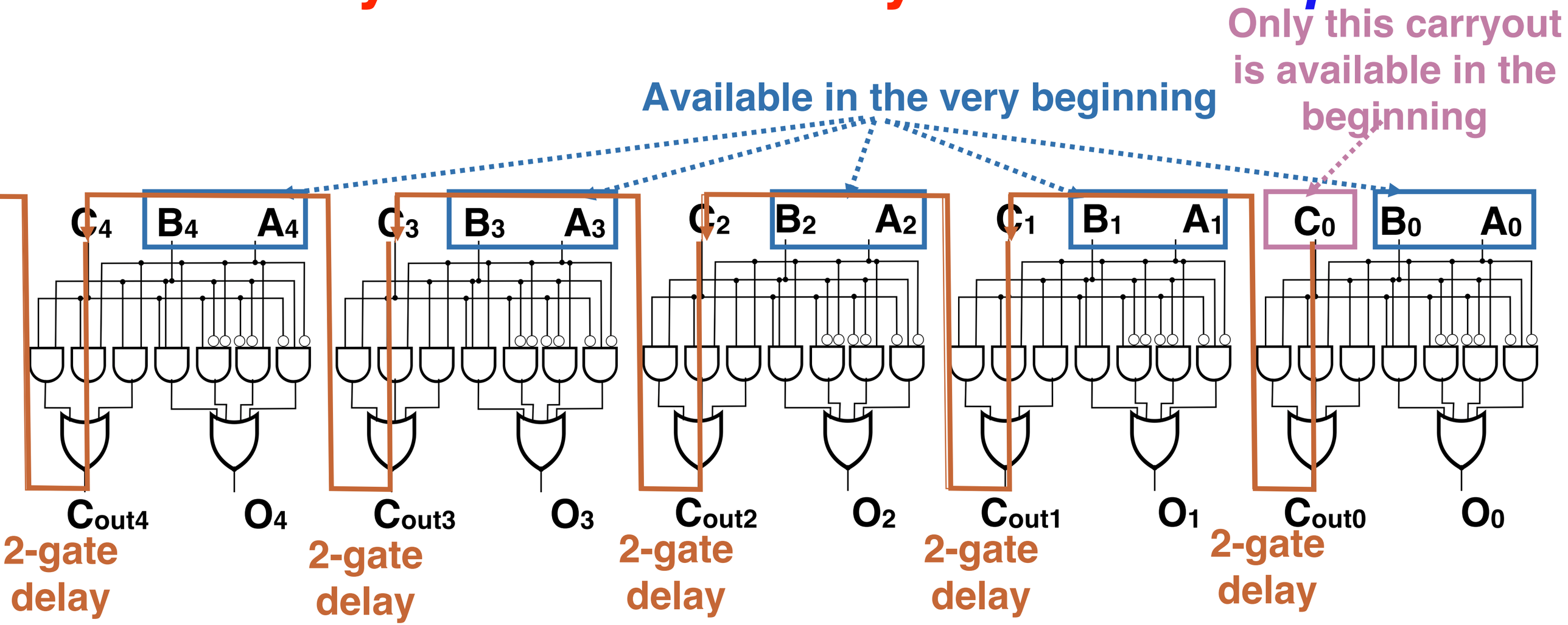
How efficient is the adder?

- Considering the shown 1-bit full adder and use it to build a 32-bit adder, how many ***gate-delays*** are we suffering to getting the final output?
  A. 2
  B. 32
  C. 64
  D. 128
  E. 288



Slide adapted from Hung-Wei Tseng

The delay is determined by the "*critical path*"

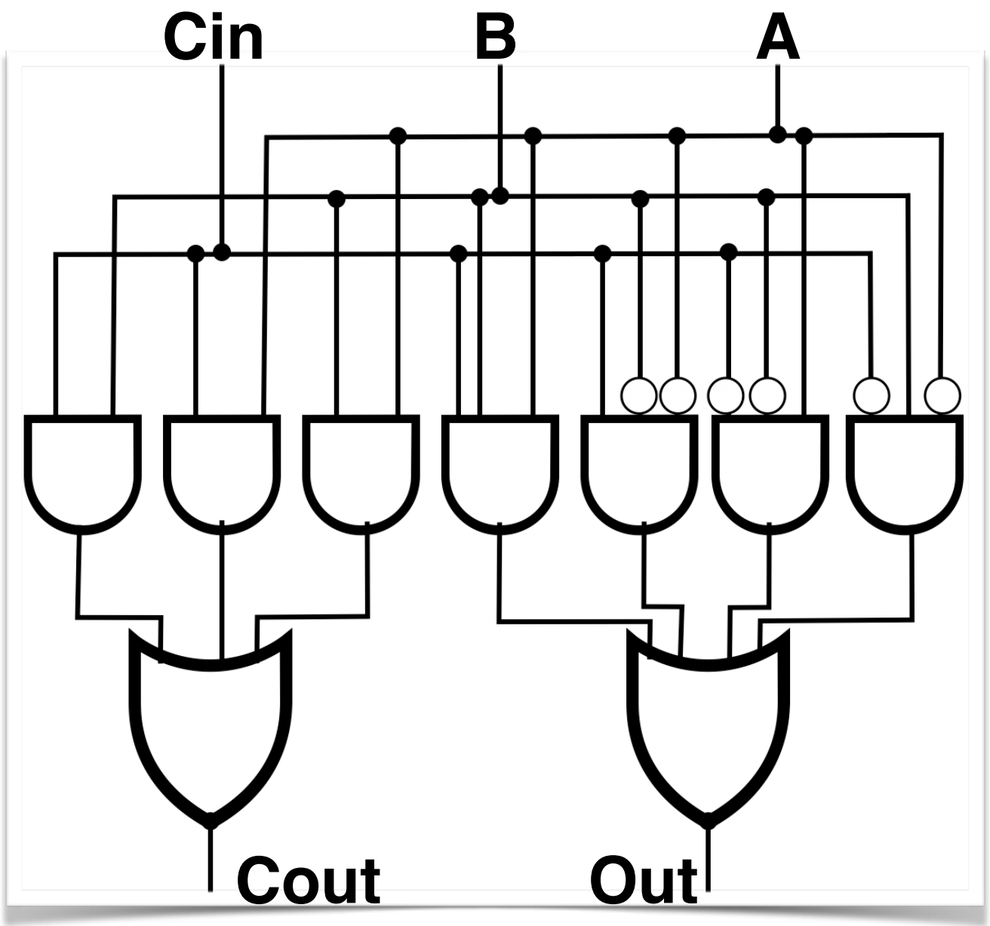


Carry-Ripple Adder (CRA)

Slide adapted from Hung-Wei Tseng

How efficient is the adder?

- Considering the shown 1-bit full adder and use it to build a 32-bit adder, how many gate-delays are we suffering to getting the final output?
  A. 2
  B. 32
  C. 64
  D. 128
  E. 288



2-gate delay per bit adder -> 2x32=64

Slide adapted from Hung-Wei Tseng

Carry-Lookahead Adder (CLA)

Carry-Lookahead Adder (CLA)

- Uses logic to quickly pre-compute the carry for each digit

Input			Output	
A	B	Cin	Out	Cout
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 = 0 --> Cout = 0 (no carry) (Delete)

Needs to wait for Cin to decide Cout (Propagate)

A = B = 1 --> Cout = 1 (must carry) (Generate)

Generate for CLA

- Uses logic to quickly pre-compute the carry for each digit

Input			Output	
A	B	Cin	Out	Cout
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**

G = AB

This means when A=B=1, we "Generate" the carryout bit

A = B = 1 --> Cout = 1 (must carry) (Generate)

Propagate for CLA

- Uses logic to quickly pre-compute the carry for each digit

Input			Output	
A	B	Cin	Out	Cout
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**

Needs to wait for Cin to decide Cout (Propagate)

P = A XOR B

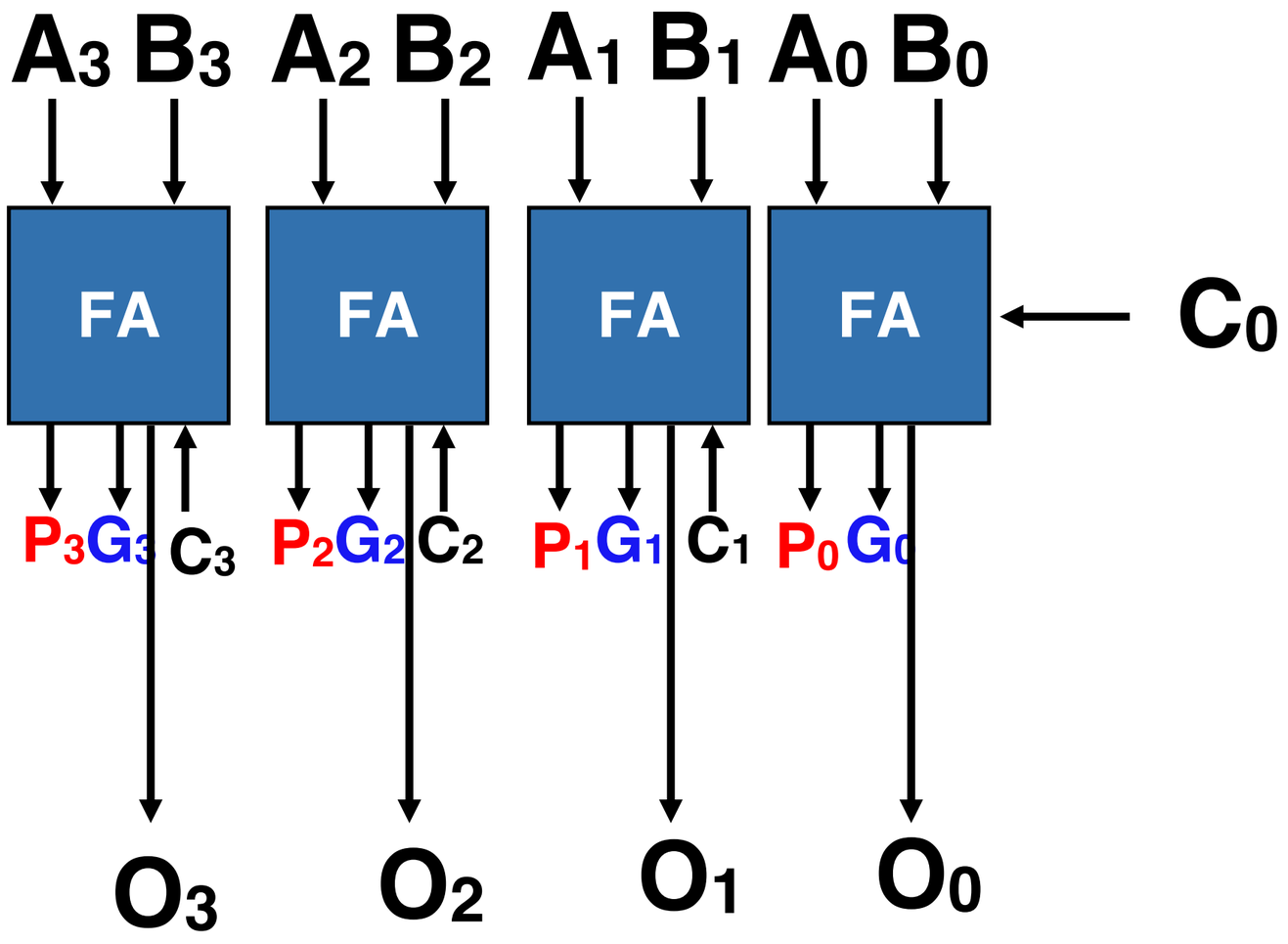
This means when A and B are different, we "Propagate"

Generate and Propagate for CLA

- All "**G**" and "**P**" are immediately available (only need to look over A_i and B_i)

$G_i = A_i B_i$
(Generate)

$P_i = A_i \text{ XOR } B_i$
(Propagate)

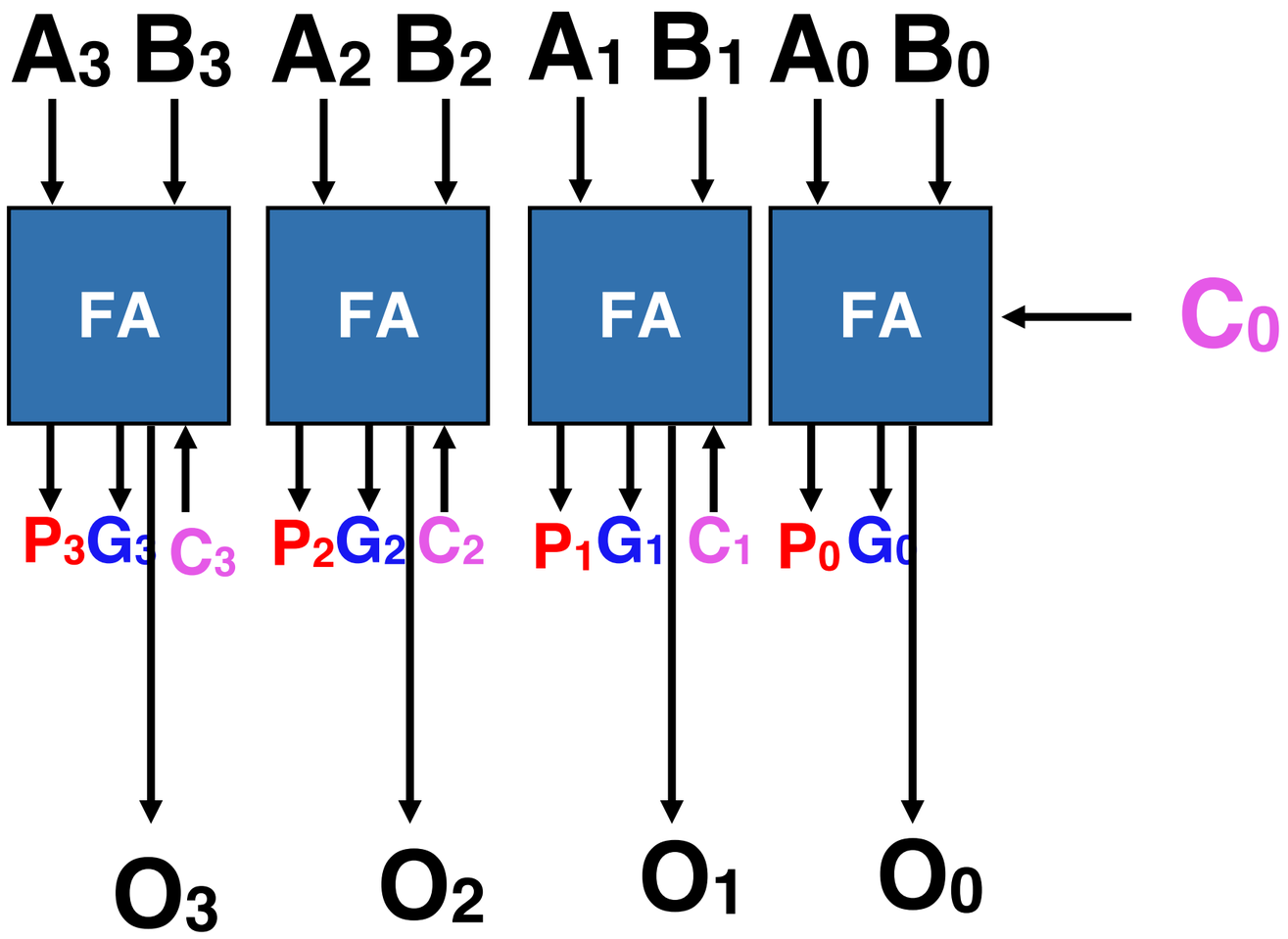


Carry for CLA

- C_0 is immediately available but the other carries "C_i"s are not

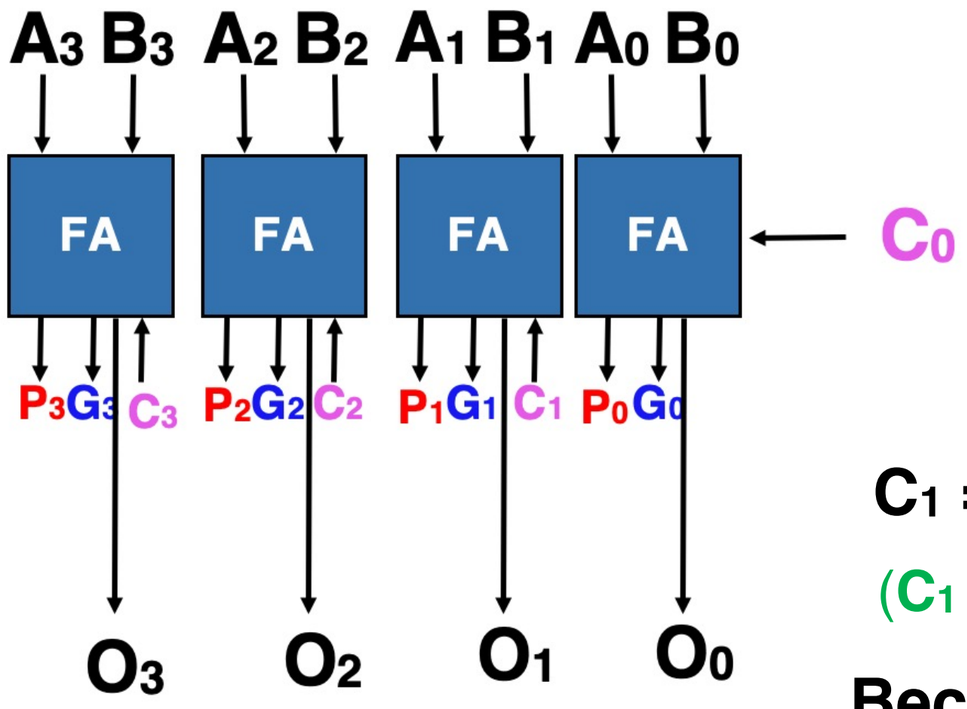
$G_i = A_i B_i$
(Generate)

$P_i = A_i \text{ XOR } B_i$
(Propagate)



Carry for CLA

- C_0 is immediately available but the other carries "C_i"s are not



C_1 is Carry Out of 0th bit and Carry In of 1st bit

$$\begin{array}{r} C_1 \quad C_0 \\ A_0 \\ +\ B_0 \\ \hline O_0 \end{array}$$

$$C_1 = A_0 B_0 + A_0 C_0 + B_0 C_0 = A_0 B_0 + C_0 (A_0' B_0 + A_0 B_0')$$

($C_1 = 1$ as long as there are at least two "1"s among A_0, B_0, C_0)

Because $G_0 = A_0 B_0$ and $P_0 = A_0$ XOR $B_0 = A_0' B_0 + A_0 B_0'$

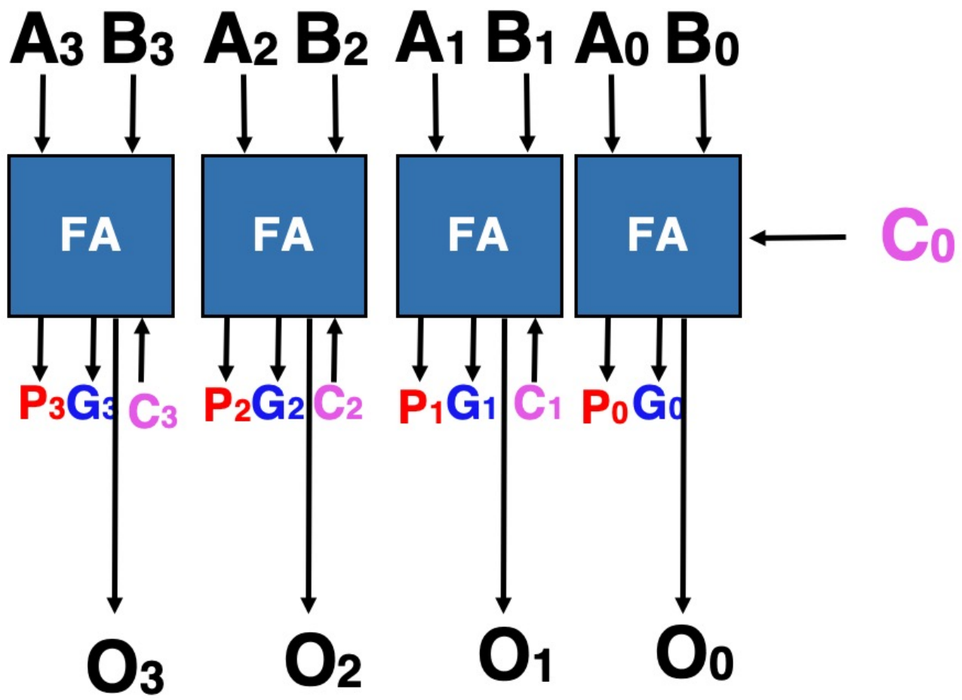
$G_i = A_i B_i$
(Generate)

$P_i = A_i$ XOR B_i
(Propagate)

$$C_1 = G_0 + P_0 C_0$$

Carry for CLA

- C_0 is immediately available but the other carries "C_i"s are not



$$C_1 = G_0 + P_0\, C_0$$

In general,

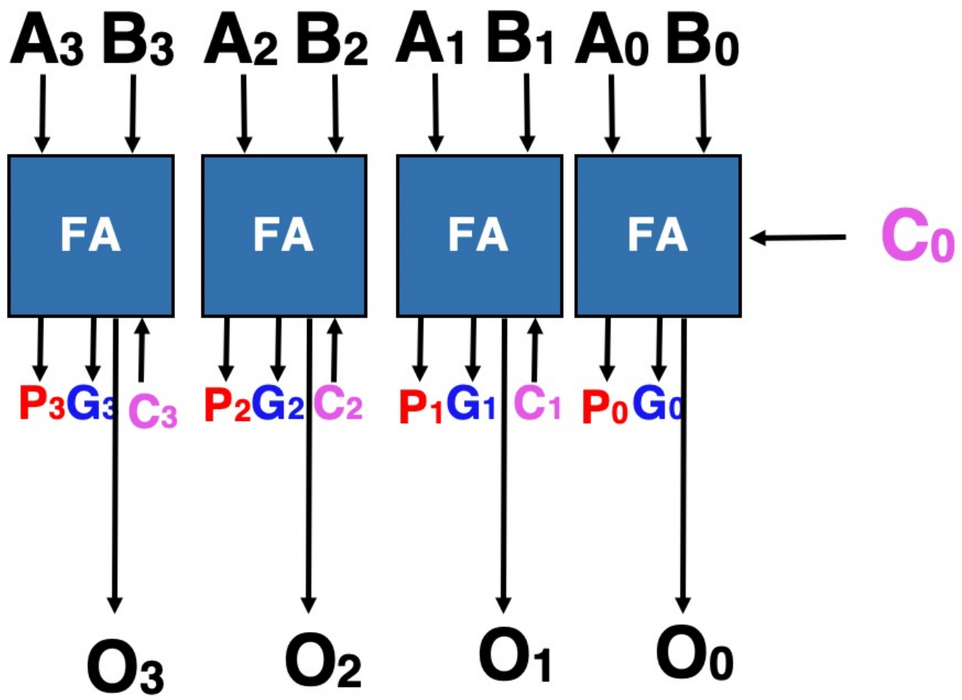
$$C_i = G_{i-1} + P_{i-1}\, C_{i-1}$$

$G_i = A_i B_i$
(Generate)

$P_i = A_i \text{ XOR } B_i$
(Propagate)

Carry for CLA

- C_0 is immediately available but the other carries "C_i"s are not



$G_i = A_i B_i$ (Generate)

$P_i = A_i \text{ XOR } B_i$ (Propagate)

$$C_i = G_{i-1} + P_{i-1}\, C_{i-1}$$

Specifically,

$$C_1 = \boxed{G_0 + P_0\, C_0}$$

$$C_2 = G_1 + P_1\, C_1 = G_1 + P_1\,(G_0 + P_0\, C_0)$$

$$= \boxed{G_1 + P_1 G_0 + P_1 P_0 C_0}$$

$$C_3 = G_2 + P_2\, \boxed{C_2}$$

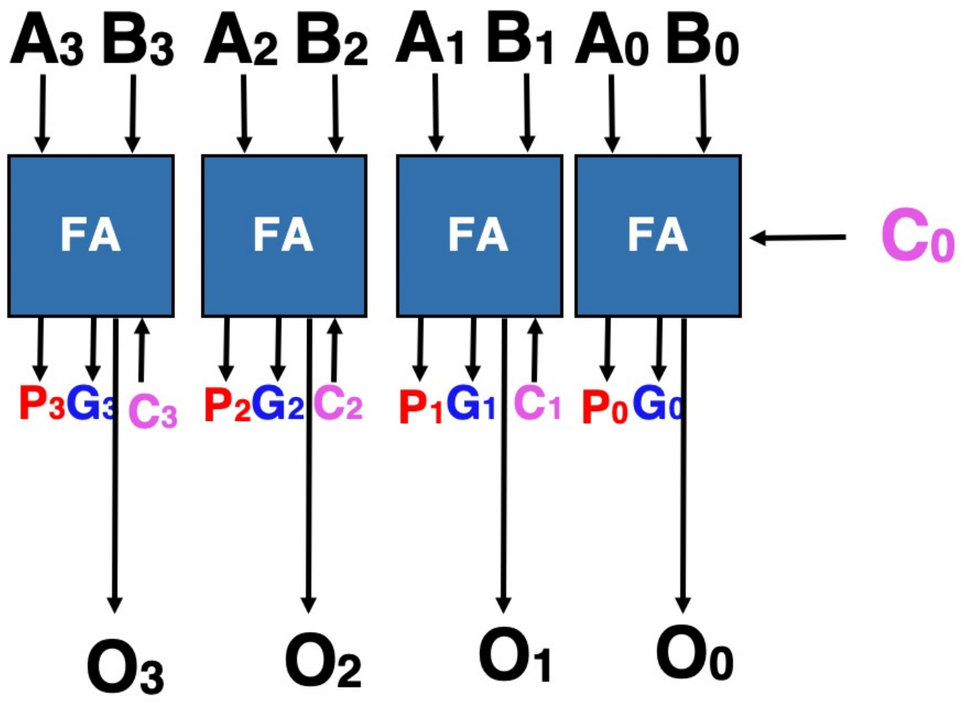
$$= \boxed{G_2 + P_2\, G_1 + P_2\, P_1 G_0 + P_2\, P_1 P_0 C_0}$$

$$C_4 = G_3 + P_3\, \boxed{C_3}$$

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

Carry for CLA

- All "**G**" and "**P**" are immediately available (only need to look over A_i and B_i)
- C_0 is immediately available but the other carries "C_i"s are not



In summary,

$$C_1 = G_0 + P_0 C_0$$

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

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

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

Only have "G_i", "P_i", and C_0

$G_i = A_i B_i$ (Generate)

$P_i = A_i \text{ XOR } B_i$ (Propagate)

Carry for CLA

- All "**G**" and "**P**" are immediately available (only need to look over A_i and B_i)
- C_0 is immediately available but the other carries "C_i"s are not

In summary,

$$G_i = A_i B_i$$

$$P_i = A_i \text{ XOR } B_i$$

$$C_1 = G_0 + P_0 C_0$$

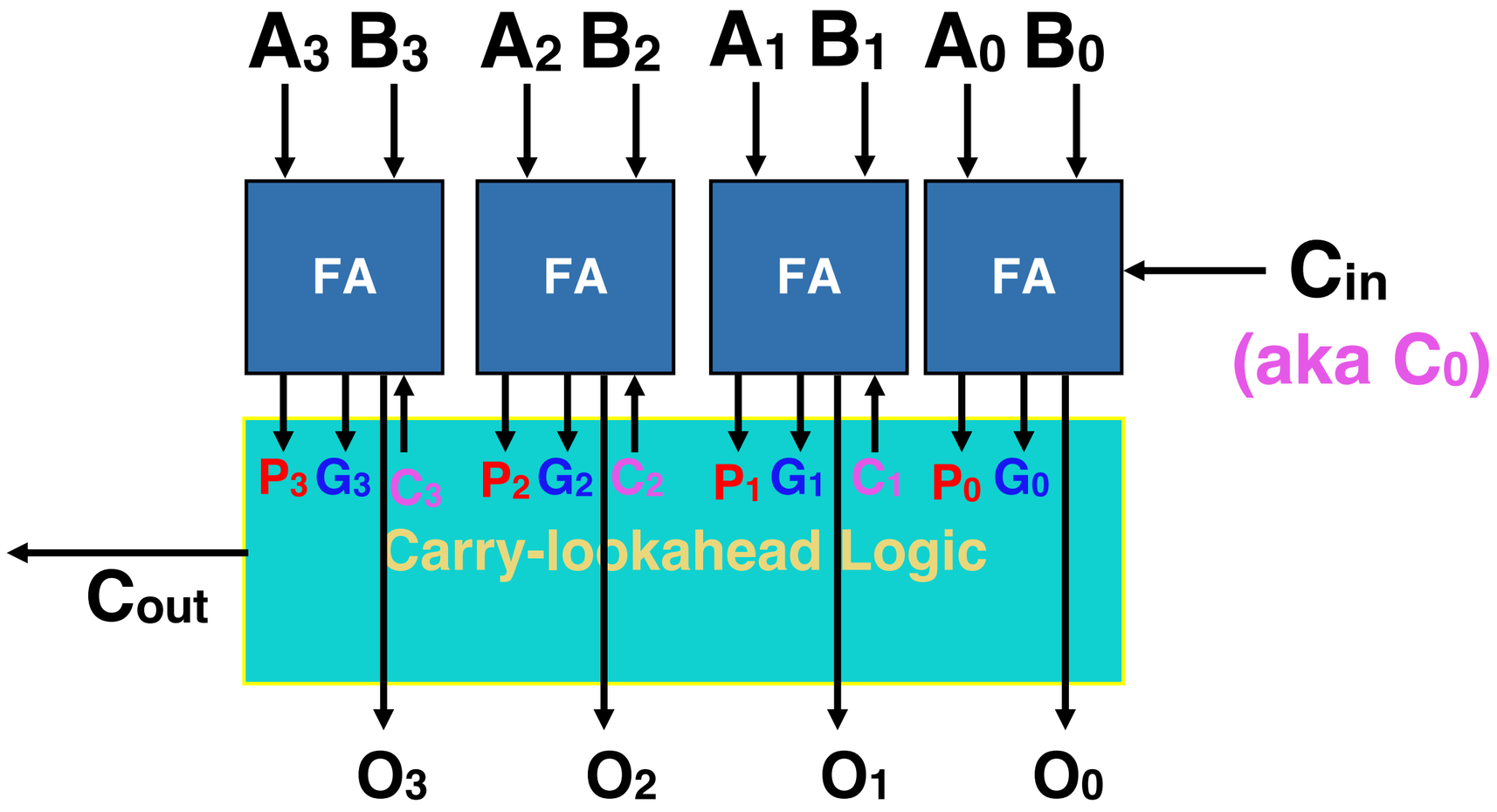
$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

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

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

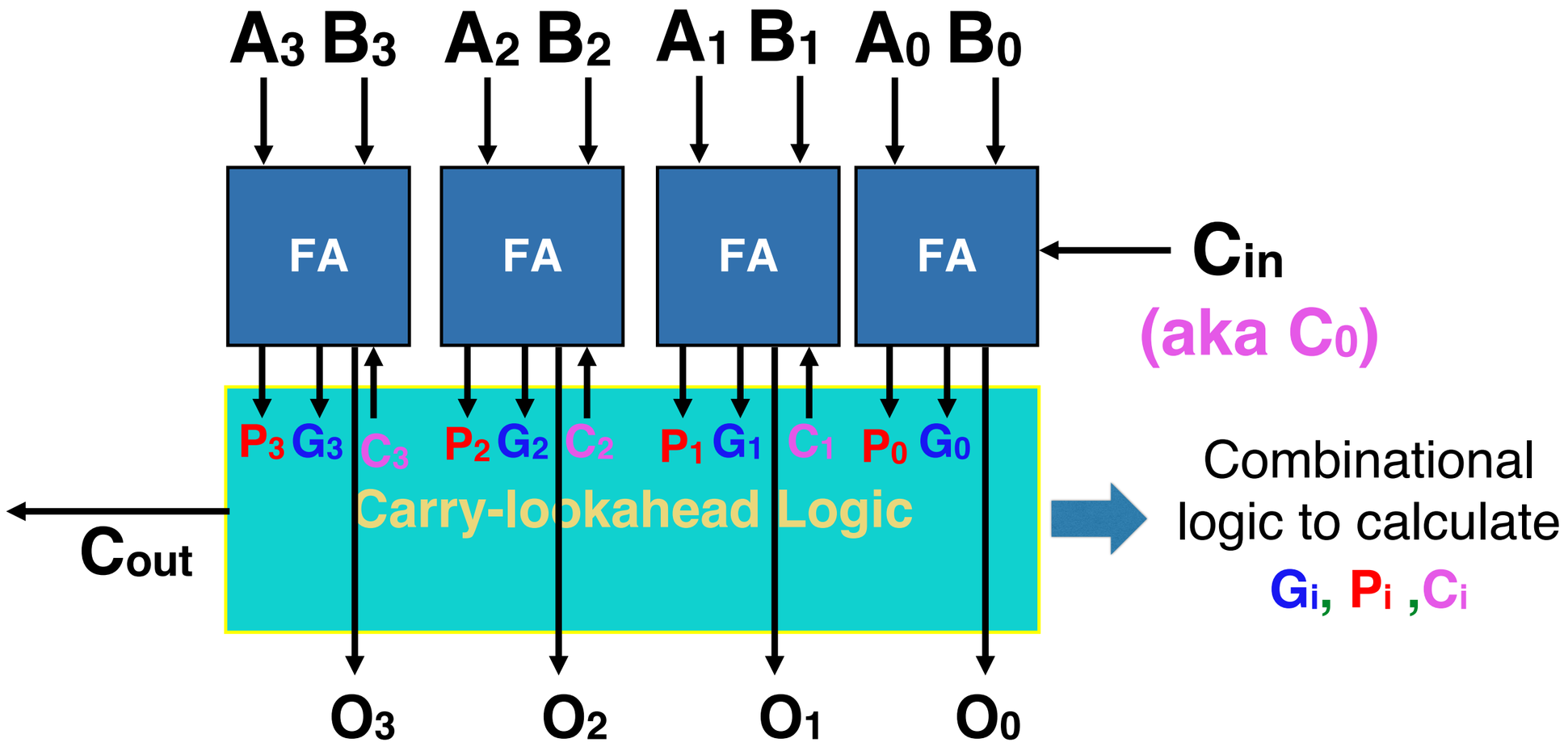
Carry-Lookahead Adder (CLA)

- Uses logic to quickly pre-compute the carry for each digit



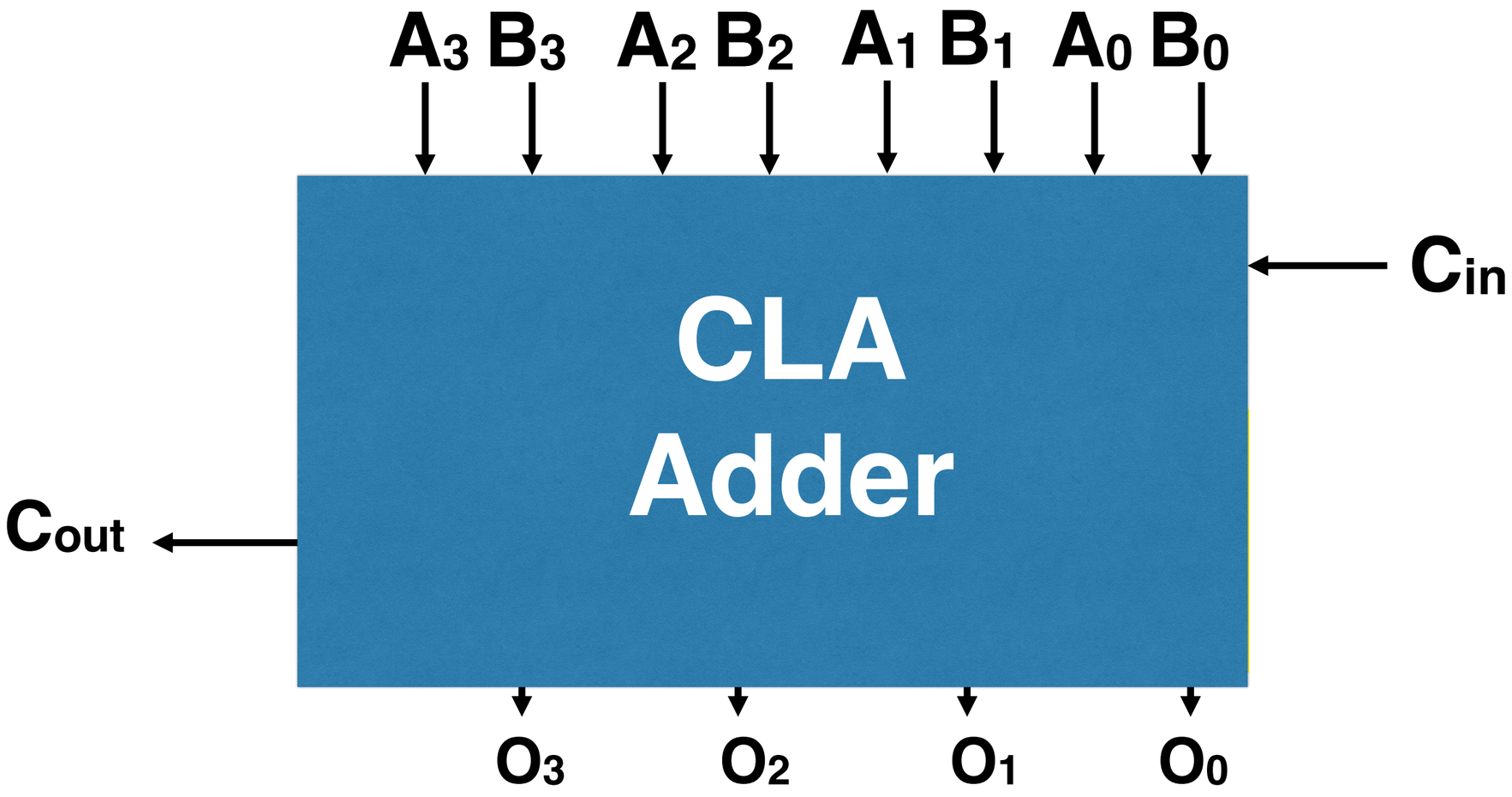
Carry-Lookahead Adder (CLA)

- Uses logic to quickly pre-compute the carry for each digit



Carry-Lookahead Adder (CLA)

- Uses logic to quickly pre-compute the carry for each digit



How efficient is the adder?

- What's the ***gate-delay*** of a 4-bit CLA?

$G_i = A_iB_i$ $\quad$ $P_i = A_i \text{ XOR } B_i$

1 gate-delay (all the G_i and P_i are calculated in parallel)

$C_1 = G_0 + P_0 C_0$ $\quad$ $C_2 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2 G_1 + P_2 P_1G_0 + P_2 P_1P_0C_0$

$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1G_0 + P_3 P_2 P_1P_0C_0$

2 gate-delay: calculate all C_i given P_i, G_i, C_0 (SoP: AND gates + OR gate)

$O_i = A_i'B_iC_i' + A_iB_i'C_i' + A_i'B_i'C_i + A_iB_iC_i$

2 gate-delay: calculate O_i given A_i, B_i, C_i

1+2+2= 5 gate-delays

How efficient is the adder?

- However,

$G_i = A_iB_i$ $\quad$ $P_i = A_i \text{ XOR } B_i$

$C_1 = G_0 + P_0 C_0$ $\quad$ $C_2 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2 G_1 + P_2 P_1G_0 + P_2 P_1P_0C_0$

$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1G_0 + P_3 P_2 P_1P_0C_0$

$O_i = A_i'B_iC_i' + A_iB_i'C_i' + A_i'B_i'C_i + A_iB_iC_i$

$O_i = f_i (A_i, B_i, C_0)$

O_i can be written as a SoP or PoS form

2 gate-delays!!!!

How efficient is the adder?

- How many transistors do we need to implement a 4-bit CLA logic?

Assumption: A 2-input gate requires 2 inputs · 2 trans/input = 4 transistors. A 3-input gate requires 3 · 2 = 6 transistors. A 4-input gate: 8 transistors.

How efficient is the adder?

- How many transistors do we need to implement a 4-bit CLA logic?

$G_i = A_iB_i$ 2-inputs for each G_i (4 transistors); 4*4=16

$P_i = A_i \text{ XOR } B_i$ 2-inputs for each P_i (4 transistors); 4*4=16

$C_1 = G_0 + P_0 C_0$ One 2-input AND and one 2-input OR; 4 + 4 = 8

$C_2 = G_1 + P_1G_0 + P_1P_0C_0$

One 2-input AND, one 3-input AND, and one 3-input OR; 4+6+6=16

$C_3 = G_2 + P_2 G_1 + P_2 P_1G_0 + P_2 P_1P_0C_0$

One 2-input AND, one 3-input AND, one 4-input AND, and one 4-input OR; 4+6+8+8=26

$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1G_0 + P_3 P_2 P_1P_0C_0$

One 2-input AND, one 3-input AND, one 4-input AND , one 5-input AND, and one 5-input OR; 4+6+8+10+8=38

$O_i = A_i'B_iC_i' + A_iB_i'C_i' + A_i'B_i'C_i + A_iB_iC_i$

For each O_i , four 3-input AND and one 4-input OR (4*6+8=32 transistors); 32*4=128

Advantages and Disadvantages of CLA

(+)

- The propagation delay is reduced.
- It provides the fastest addition logic.
- It's gate delay is lower than RCA.

(-)

- The CLA circuit gets complicated as the number of variables increase.
- It need more transistors than RCA.

Area-Delay Trade-off!

Subtractor

2's Complement Addition

(+5)	0 1 0 1	(–5)	1 0 1 1
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+7)	0 1 1 1	(–3)	1 1 0 1

(+5)	0 1 0 1	(–5)	1 0 1 1
+ (–2)	+ 1 1 1 0	+ (–2)	+ 1 1 1 0
(+3)	1 0 0 1 1	(–7)	1 1 0 0 1
	ignore		ignore

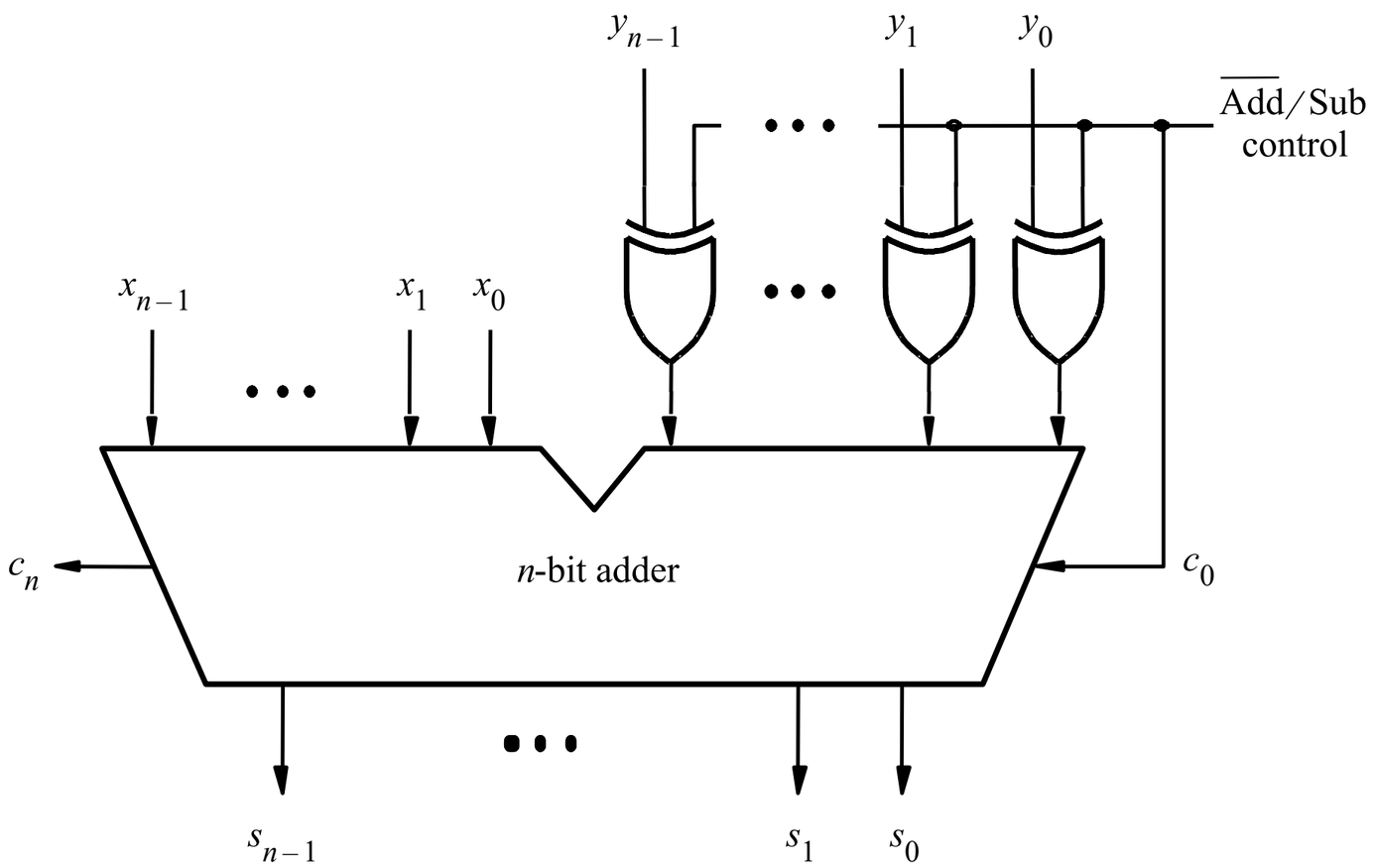
- If there is a carry-out from the sign-bit position, it is simply ignored.
- The 2's complement notation is highly suitable for addition.

2's Complement Subtraction

- Easy way: find the 2's complement of the subtrahend, and add

(+5)	0 1 0 1		0 1 0 1
– (+2)	– 0 0 1 0	⟹	+ 1 1 1 0
(+3)			1 0 0 1 1
			↑ ignore
(–5)	1 0 1 1		1 0 1 1
– (+2)	– 0 0 1 0	⟹	+ 1 1 1 0
(–7)			1 1 0 0 1
			↑ ignore
(+5)	0 1 0 1		0 1 0 1
– (–2)	– 1 1 1 0	⟹	+ 0 0 1 0
(+7)			0 1 1 1
(–5)	1 0 1 1		1 0 1 1
– (–2)	– 1 1 1 0	⟹	+ 0 0 1 0
(–3)			1 1 0 1

Adder and Subtractor Unit



- Control signal = 0: adder
- Control signal = 1: subtractor (carry-in c0=1 due to 2's complement of subtrahend; -y obtained by flipping each bit of of y and adding 1)

Evaluating 2's complement

- Do we need a separate procedure/hardware for adding positive and negative numbers?

A. No. The same procedure applies
B. No. The same "procedure" applies but it changes overflow detection
C. Yes, and we need a new procedure
D. Yes, and we need a new procedure and a new hardware
E. None of the above

Evaluating 2's complement

- Do we need a separate procedure/hardware for adding positive and negative numbers?
  - 3 + 2 = 5

```
   1
  0 0 1 1
+ 0 0 1 0
---------
  0 1 0 1
```

  - 3 + (-2) = 1

```
  1 1
1 0 0 1 1
+ 1 1 1 0
---------
  0 0 0 1  = 1
```

A. No. The same procedure applies
B. No. The same "procedure" applies but it changes overflow detection
C. Yes, and we need a new procedure
D. Yes, and we need a new procedure and a new hardware
E. None of the above