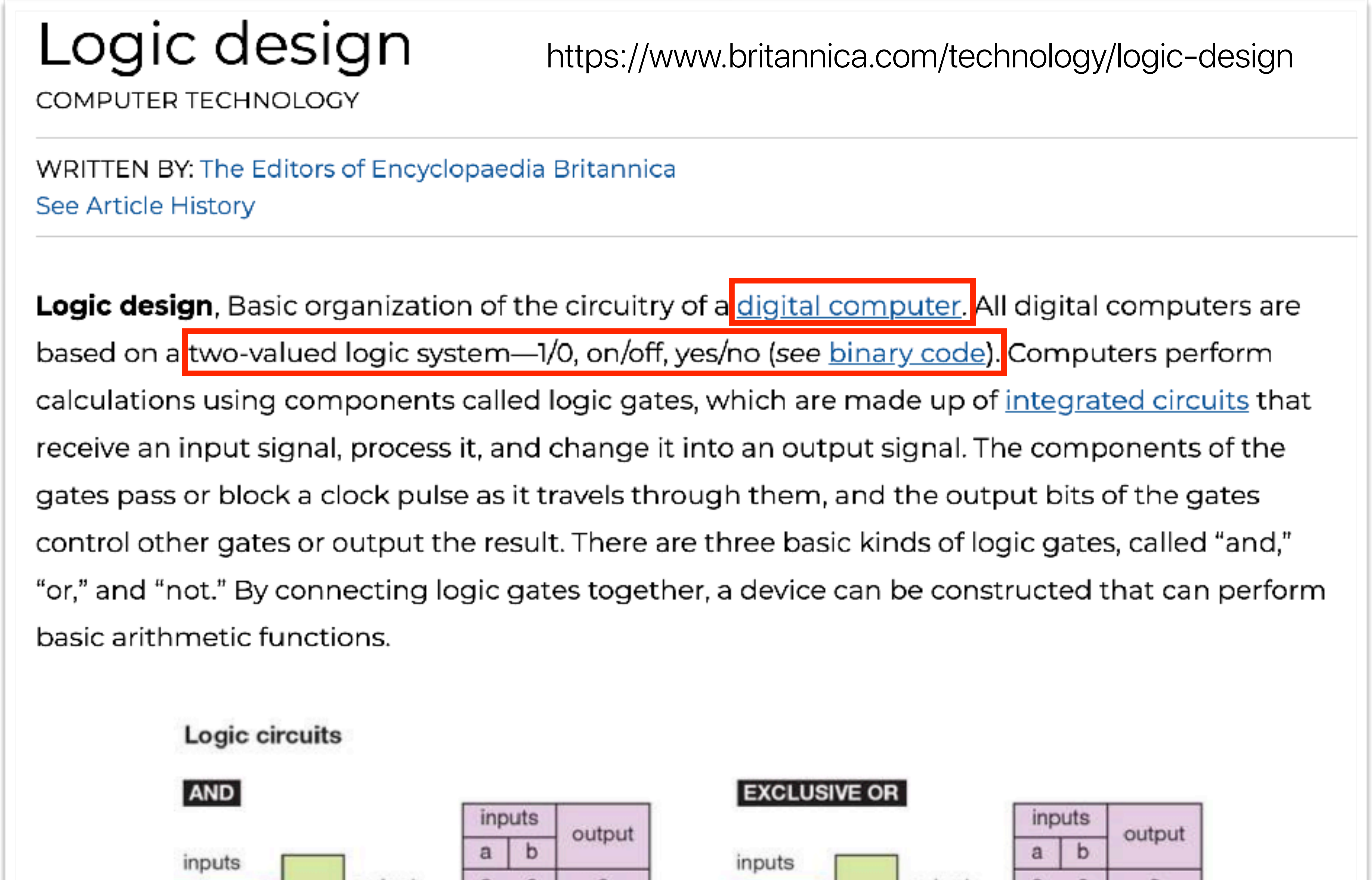
Midterm Review

Prof. Usagi

Recap: Logic Design?



Logic design

https://www.britannica.com/technology/logic-design

COMPUTER TECHNOLOGY

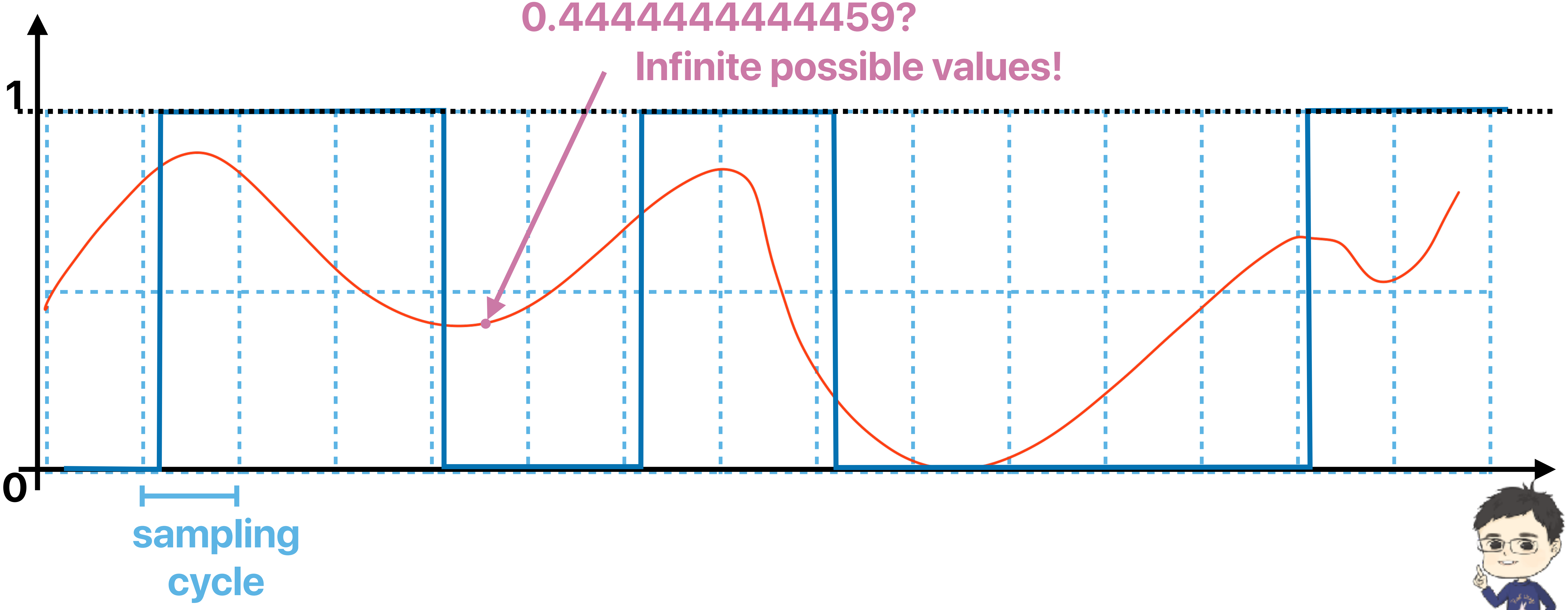
WRITTEN BY: The Editors of Encyclopaedia Britannica

See Article History

Logic design, Basic organization of the circuitry of a digital computer. All digital computers are based on a two-valued logic system—1/0, on/off, yes/no (*see* binary code). Computers perform calculations using components called logic gates, which are made up of integrated circuits that receive an input signal, process it, and change it into an output signal. The components of the gates pass or block a clock pulse as it travels through them, and the output bits of the gates control other gates or output the result. There are three basic kinds of logic gates, called "and," "or," and "not." By connecting logic gates together, a device can be constructed that can perform basic arithmetic functions.

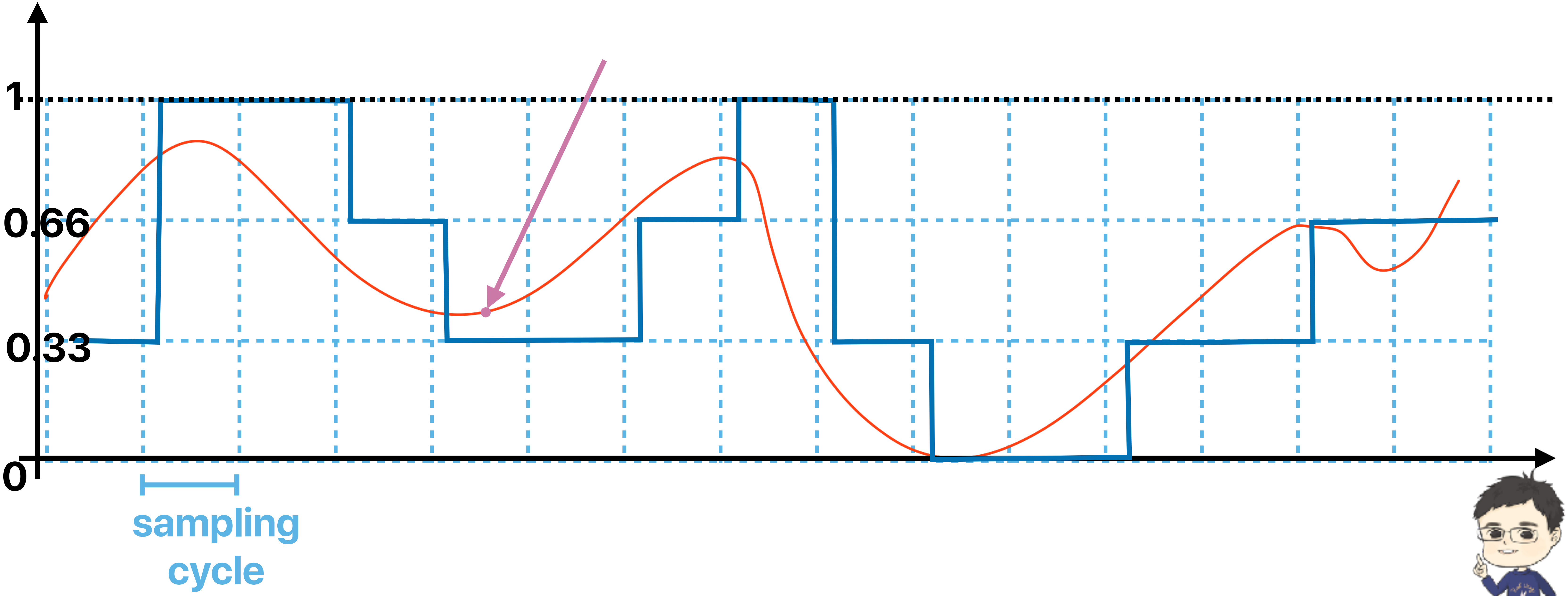
Logic circuits

Analog v.s. digital signals


0.5? 0.4? 0.45?
0.445? 0.4445? or
0.44444444444459?

Infinite possible values!

Analog v.s. digital signals



Recap: What’s 0.0004 in IEEE 754?

0	0	1	1	1	0	0	1	1	1	0	1	0	0	0	1	1	0	1	1	0	1	1	1	0	0	0	1	1	1	0	1

	after x2	> 1?
0.0004	0.0008	0
0.0008	0.0016	0
0.0016	0.0032	0
0.0032	0.0064	0
0.0064	0.0128	0
0.0128	0.0256	0
0.0256	0.0512	0
0.0512	0.1024	0
0.1024	0.2048	0
0.2048	0.4096	0
0.4096	0.8192	0
0.8192	1.6384	1
0.6384	1.2768	1
0.2768	0.5536	0
0.5536	1.1072	1
0.1072	0.2144	0
0.2144	0.4288	0
0.4288	0.8576	0
0.8576	1.7152	1
0.7152	1.4304	1

12

	after x2	> 1?
0.4304	0.8608	0
0.8608	1.7216	1
0.7216	1.4432	1
0.4432	0.8864	0
0.8864	1.7728	1
0.7728	1.5456	1
0.5456	1.0912	1
0.0912	0.1824	0
0.1824	0.3648	0
0.3648	0.7296	0
0.7296	1.4592	1
0.4592	0.9184	0
0.9184	1.8368	1
0.8368	1.6736	1
0.6736	1.3472	1
0.3472	0.6944	0
0.6944	1.3888	1
0.3888	0.7776	0
0.7776	1.5552	1
0.5552	1.1104	1

	after x2	> 1?
0.1104	0.2208	0
0.2208	0.4416	0
0.4416	0.8832	0
0.8832	1.7664	1
0.7664	1.5328	1
0.5328	1.0656	1
0.0656	0.1312	0
0.1312	0.2624	0
0.2624	0.5248	0
0.5248	1.0496	1
0.0496	0.0992	0
0.0992	0.1984	0
0.1984	0.3968	0
0.3968	0.7936	0
0.7936	1.5872	1
0.5872	1.1744	1
0.1744	0.3488	0
0.3488	0.6976	0
0.6976	1.3952	1
0.3952	0.7904	0

You can never get 0.0004 again if you convert signal/store data in IEEE 754 float

Recap: Why are digital computers more popular now?

- Please identify how many of the following statements explains why digital computers are now more popular than analog computers.
  - ✓ The cost of building systems with the same functionality is lower by using digital computers.
  - ✗ Digital computers can express more values than analog computers.
  - ✓ Digital signals are less fragile to noise and defective/low-quality components.
  - ✓ Digital data are easier to store.

A. 0

B. 1

C. 2

[✓ D. 3]

E. 4

Types of digital circuits

Combinational v.s. sequential logic

- Combinational logic
  - The output is a pure function of its current inputs
  - The output doesn’t change regardless how many times the logic is triggered — Idempotent
- Sequential logic
  - The output depends on current inputs, previous inputs, their history

Theory behind each

- A **Combinational logic** is the implementation of a **Boolean Algebra** function with only Boolean Variables as their inputs
- A **Sequential logic** is the implementation of a **Finite-State Machine**

Basic Boolean Algebra Concepts

- {0, 1}: The only two possible values in inputs/outputs
- Basic operators
  - AND (•) — a • b
    - returns 1 only if both a **and** b are 1s
    - otherwise returns 0
  - OR (+) — a + b
    - returns 1 if a **or** b is 1
    - returns 0 if none of them are 1s
  - NOT (‘) — a’
    - returns 0 if a is 1
    - returns 1 if a is 0

Truth tables

- A table sets out the functional values of logical expressions on each of their functional arguments, that is, for each combination of values taken by their logical variables

AND

Input		Output
A	B	
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

OR

Input		Output
A	B	
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

NOT

Input	Output
A	
0	1
0	1
1	0
1	0

Derived Boolean operators

- NAND — $(a \bullet b)'$
- NOR — $(a + b)'$
- XOR — $(a + b) \bullet (a' + b')$ or $ab' + a'b$
- XNOR — $(a + b') \bullet (a' + b)$ or $ab + a'b'$

NAND

Input		Output
A	B	
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

NOR

Input		Output
A	B	
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

XOR

Input		Output
A	B	
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

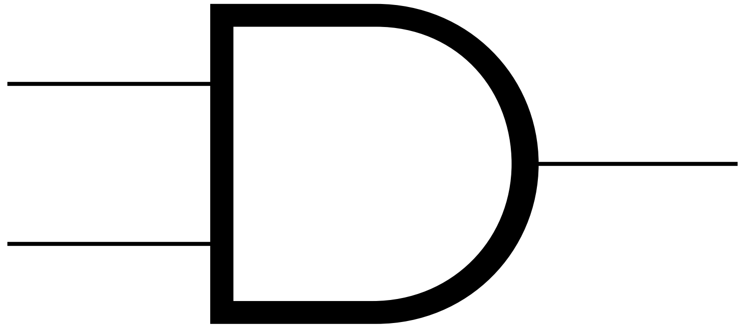
XNOR

Input		Output
A	B	
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

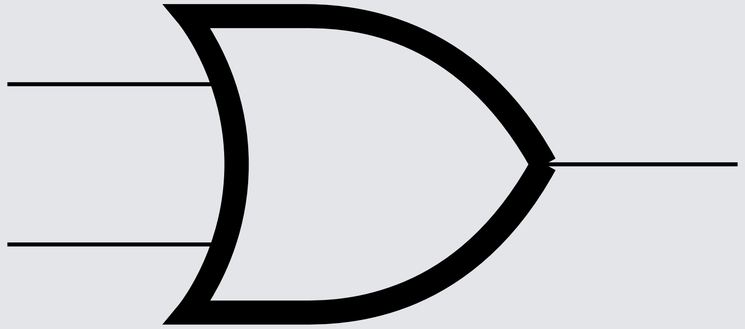
Express Boolean Operators/ Functions in Circuit “Gates”

Boolean operators their circuit “gate” symbols

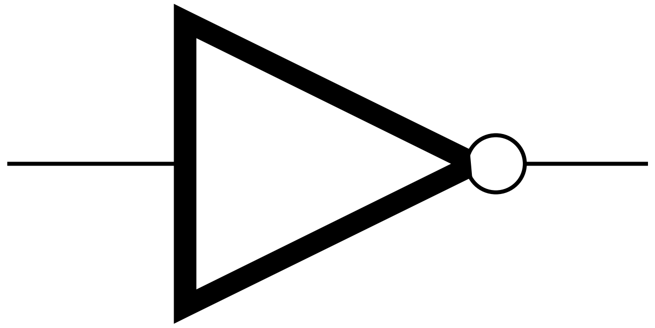
AND



OR



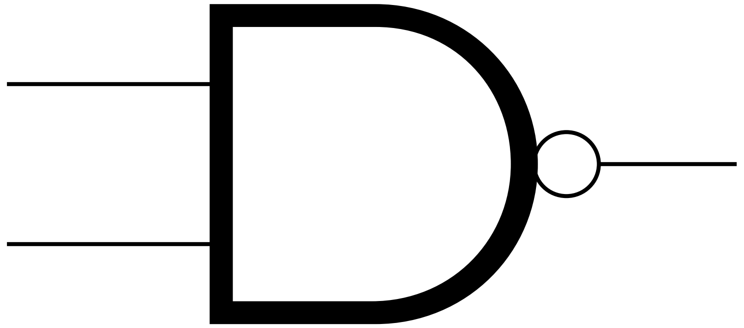
NOT



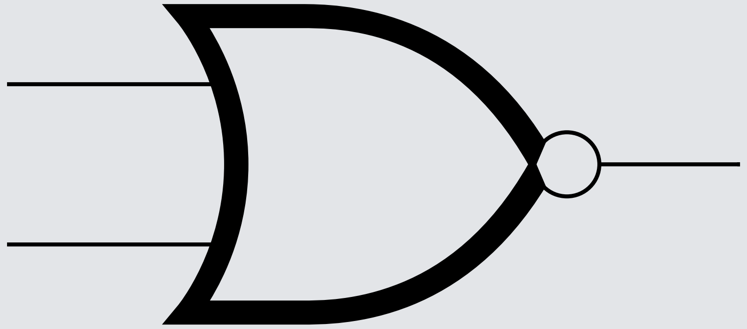
—○ represents where we take a compliment value on an input

○— represents where we take a compliment value on an output

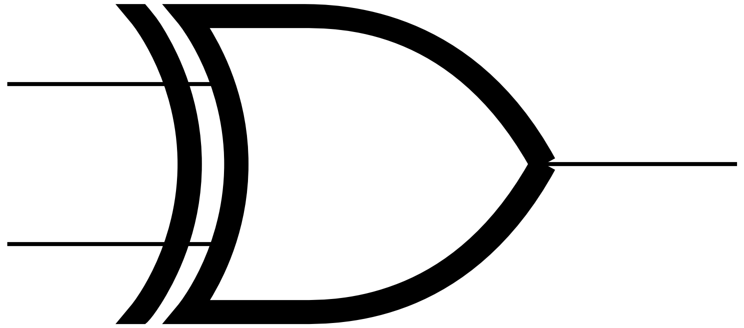
NAND



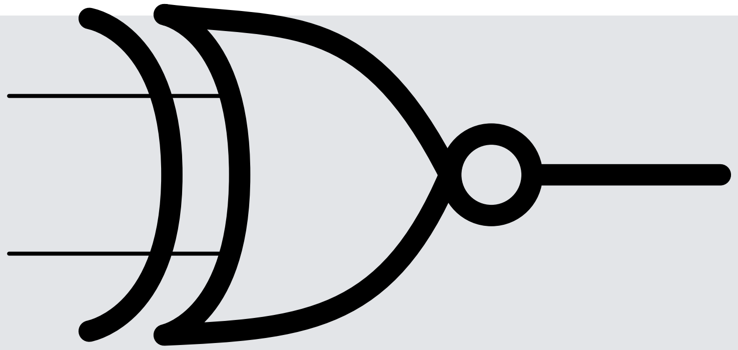
NOR



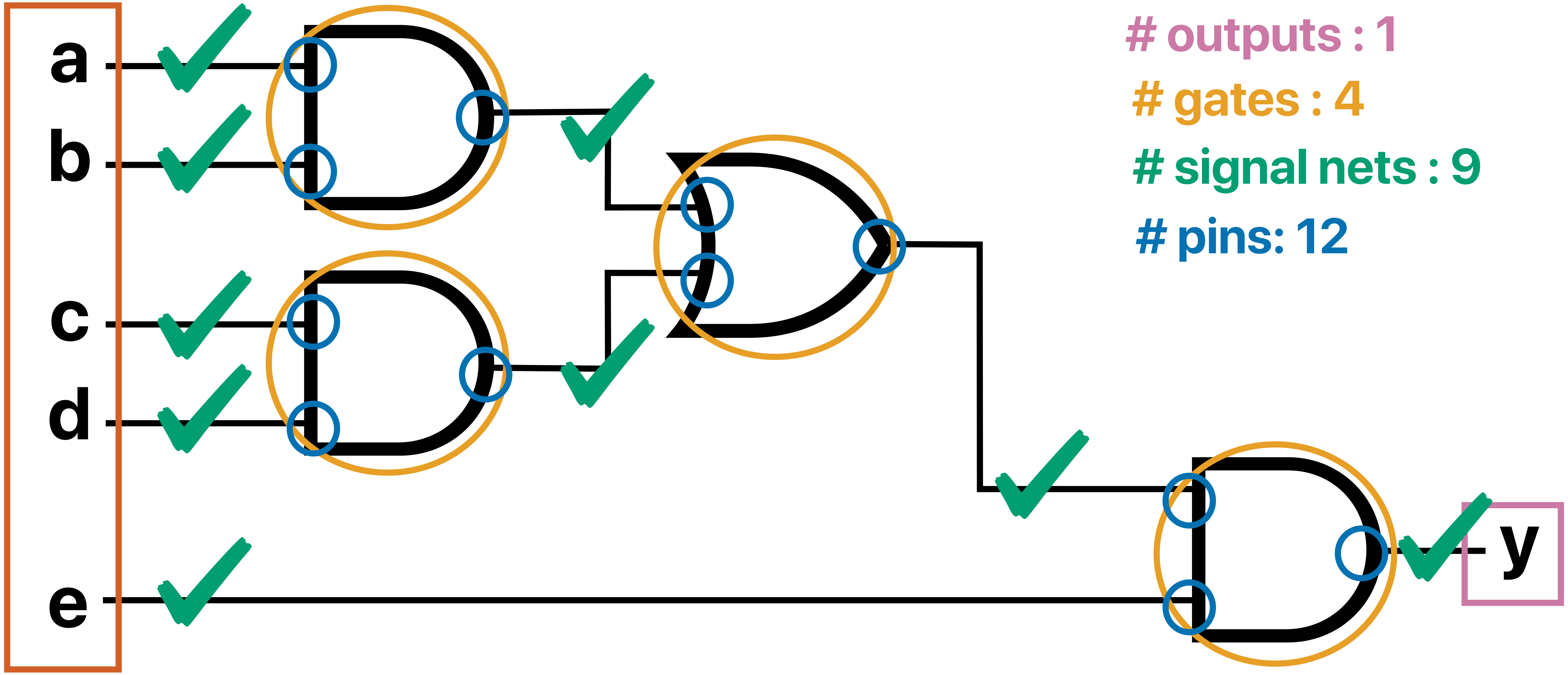
XOR



NXOR



How to express y = e(ab+cd)



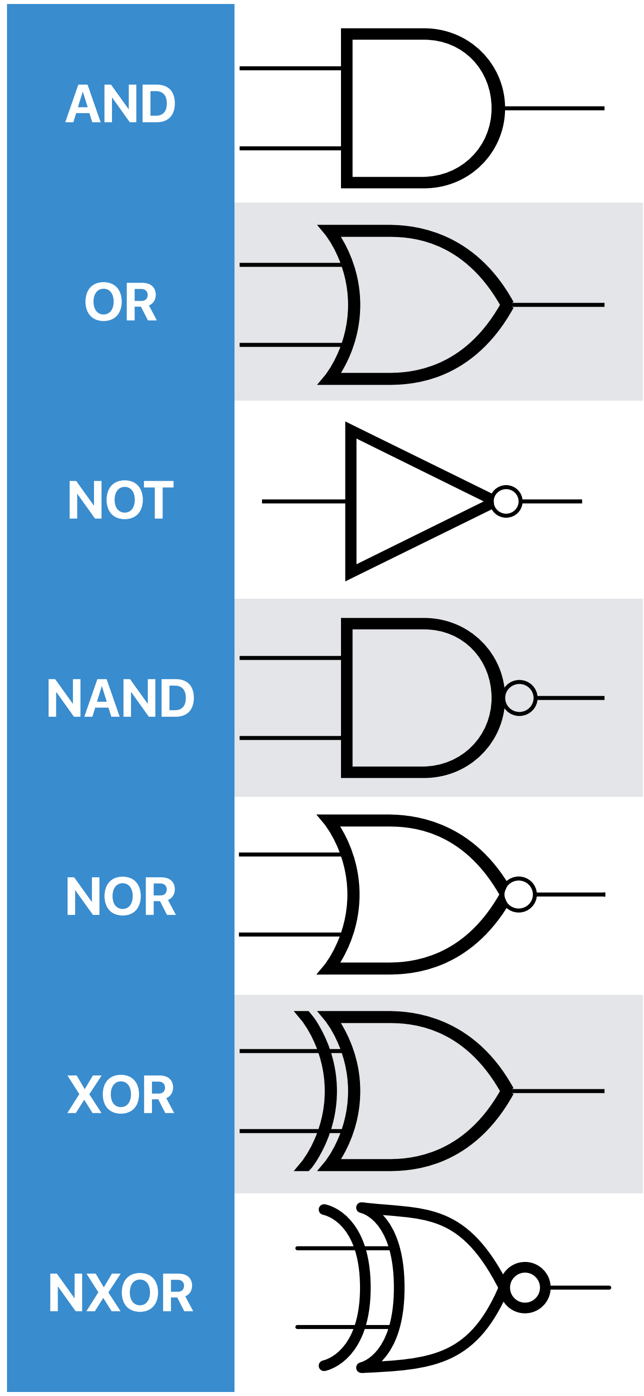
# inputs : 5

# outputs : 1

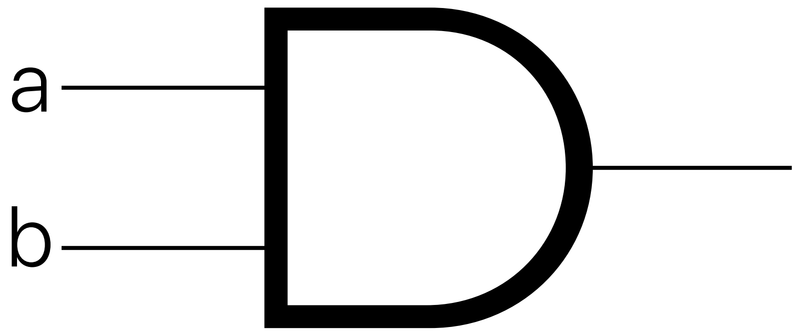
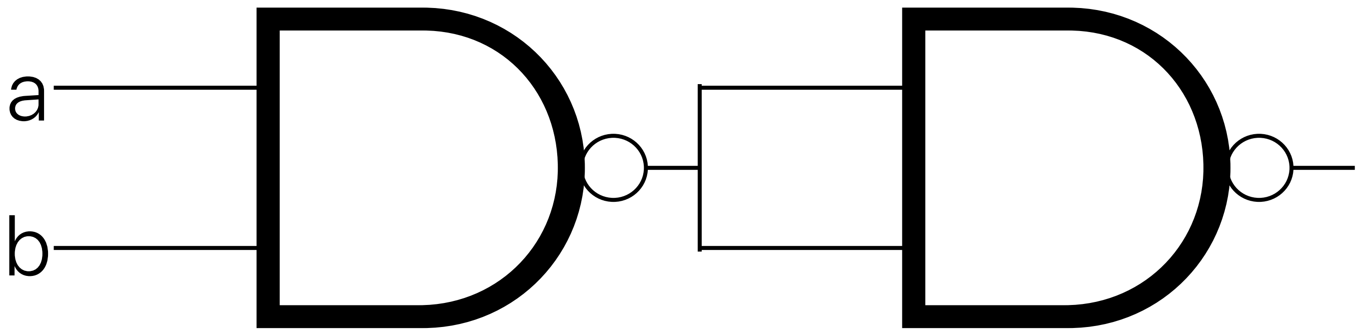
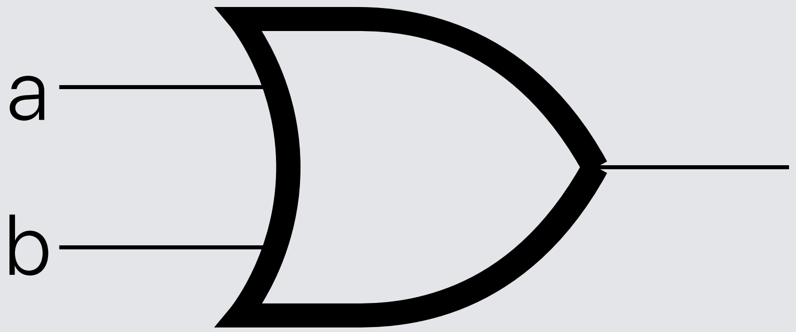
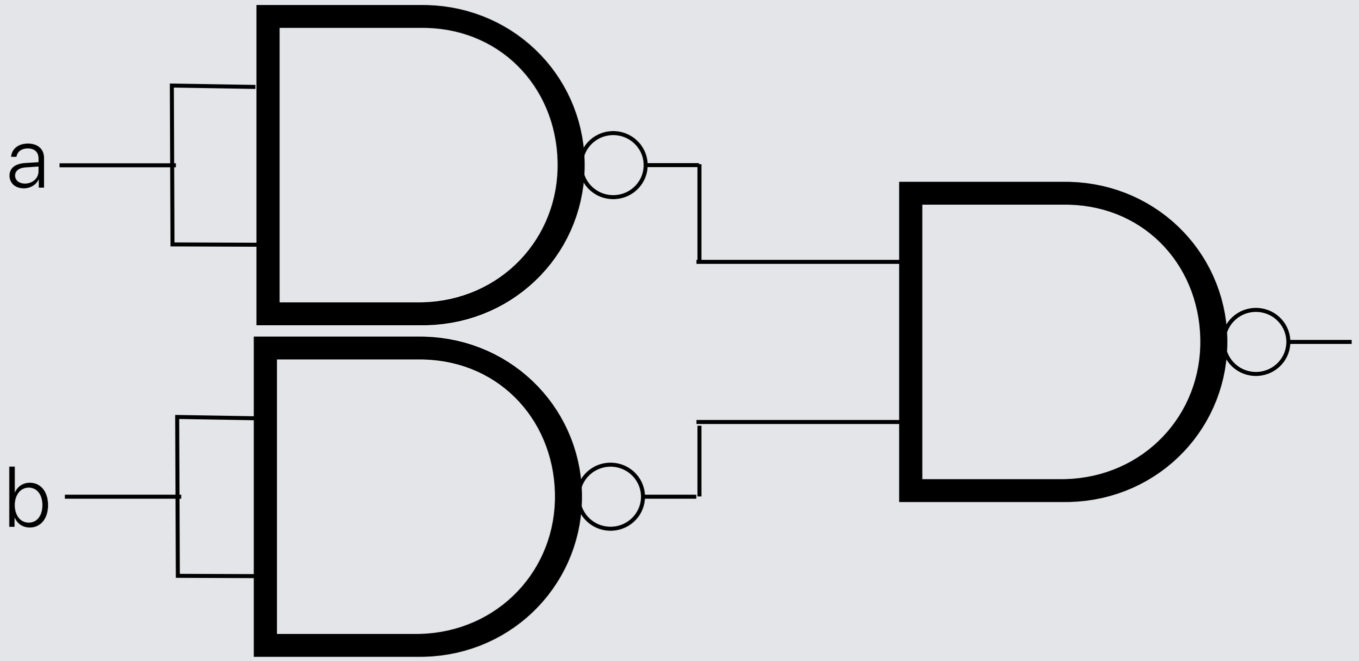
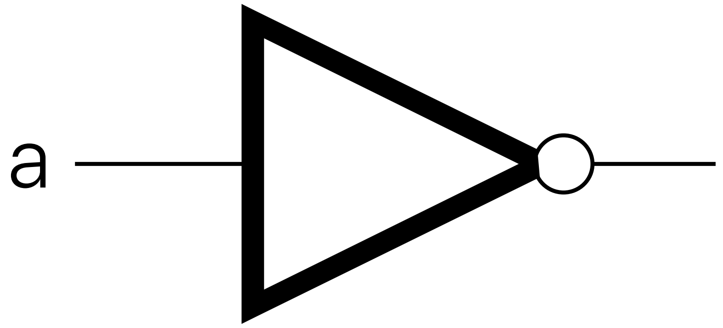
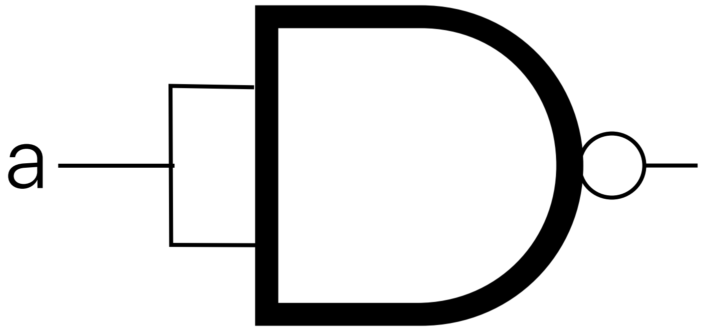
# gates : 4

# signal nets : 9

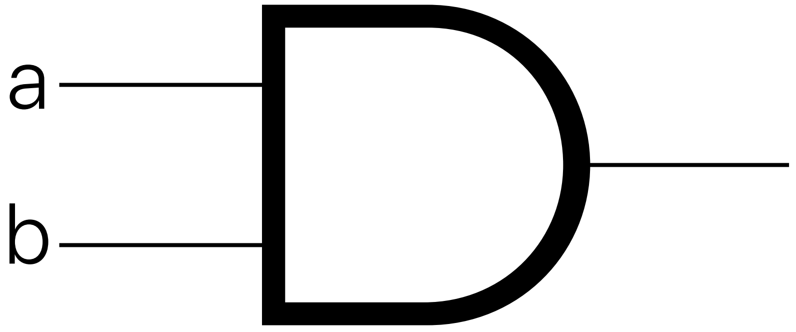
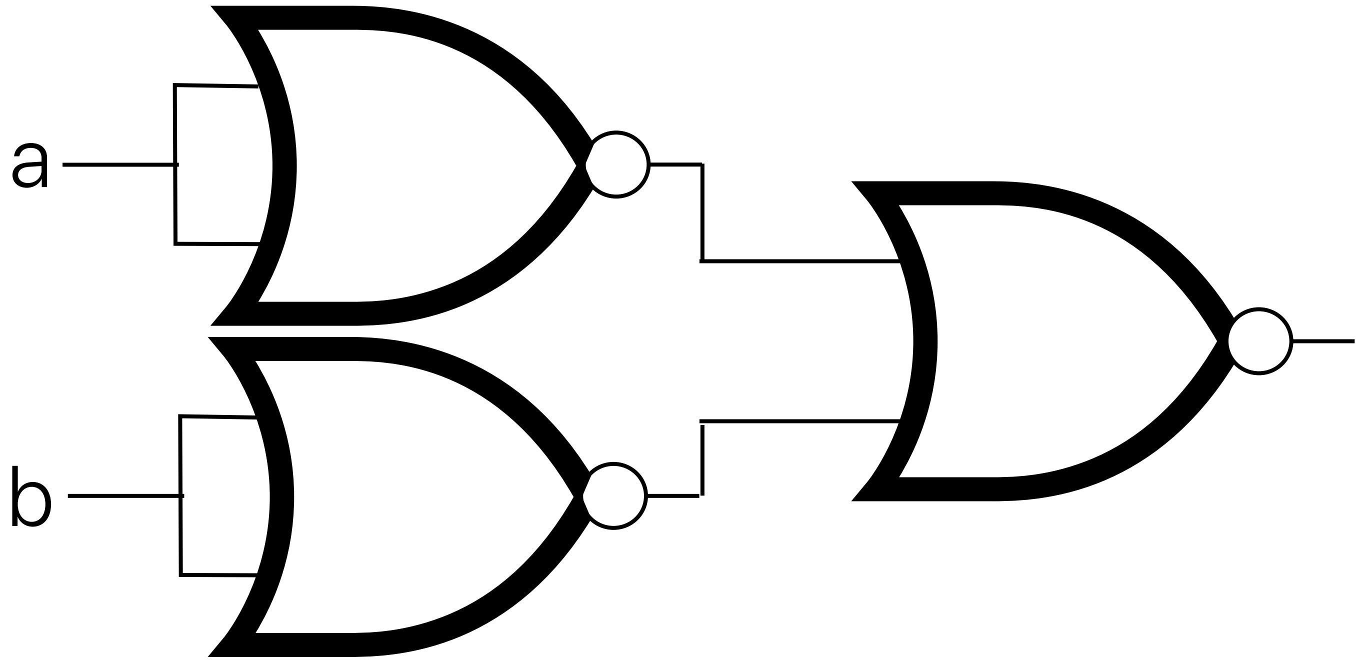
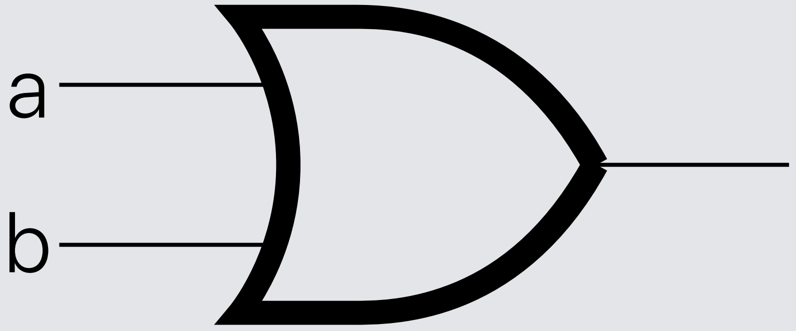
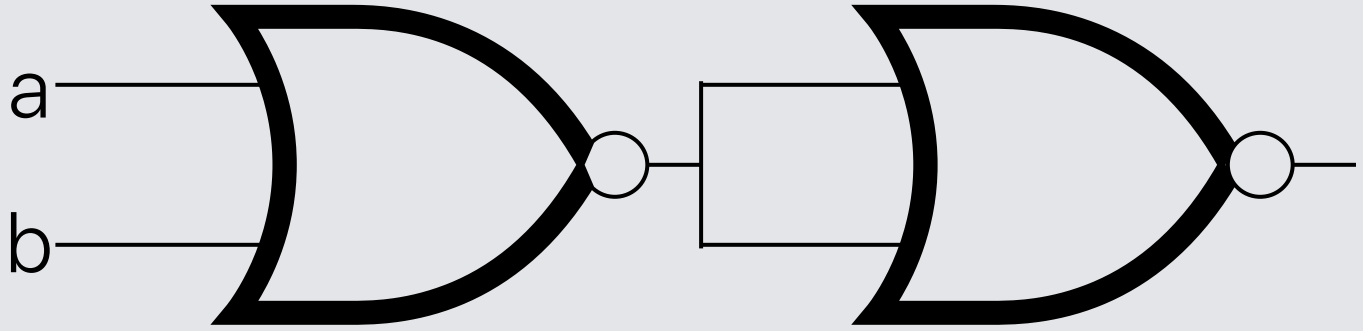
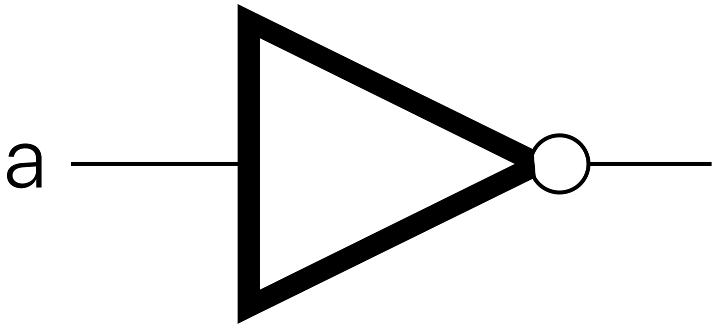
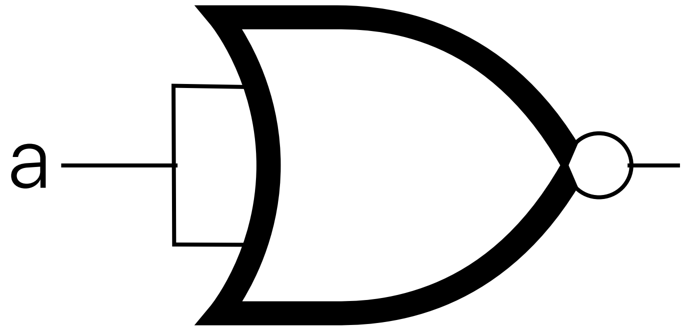
# pins: 12



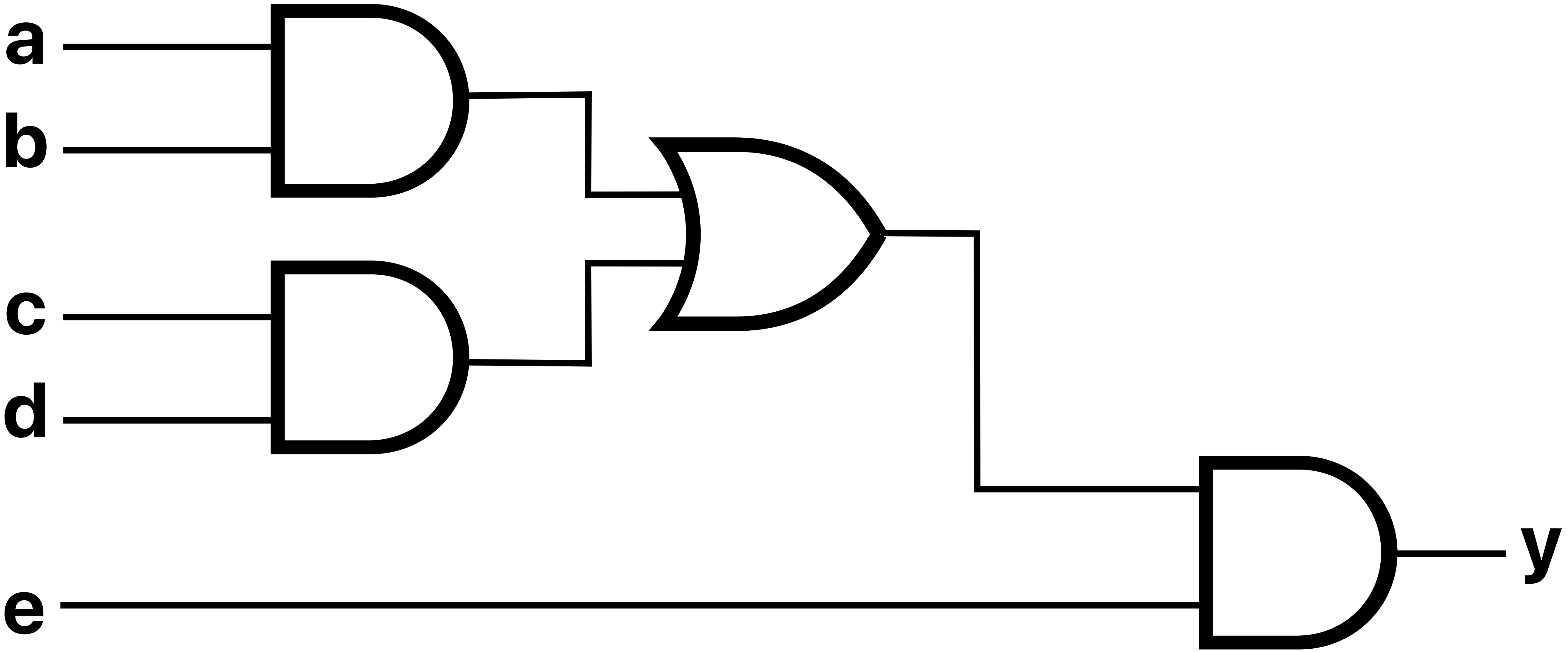
We can make everything NAND!

	Original	NAND
AND		
OR		
NOT		

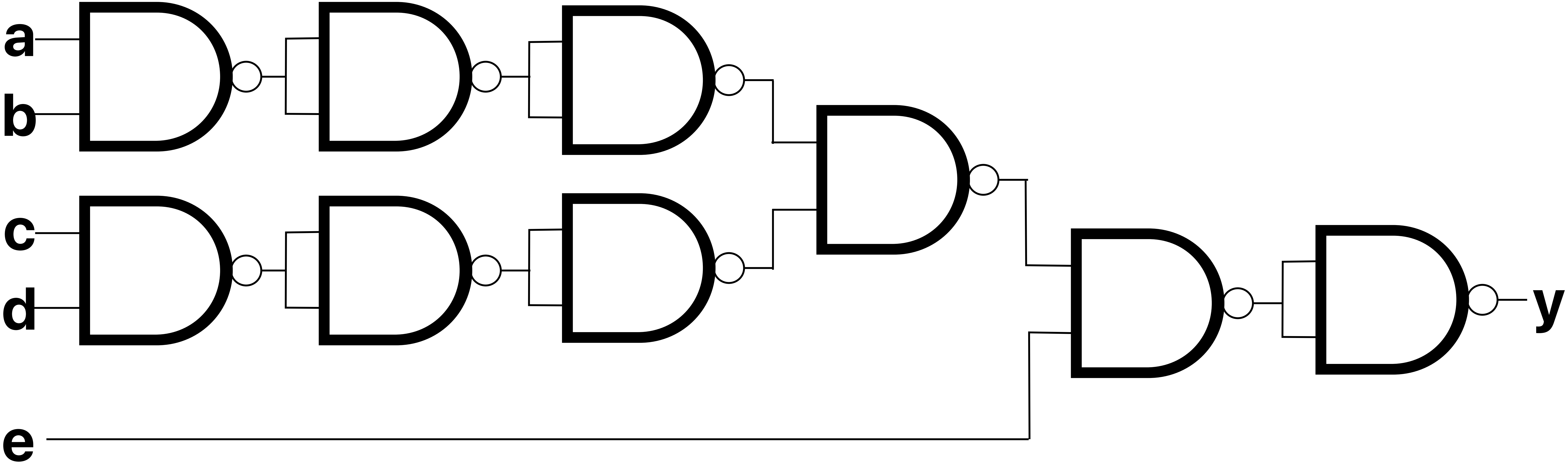
We can also make everything NOR!

	Original	NAND
AND		
OR		
NOT		

How to express y = e(ab+cd)



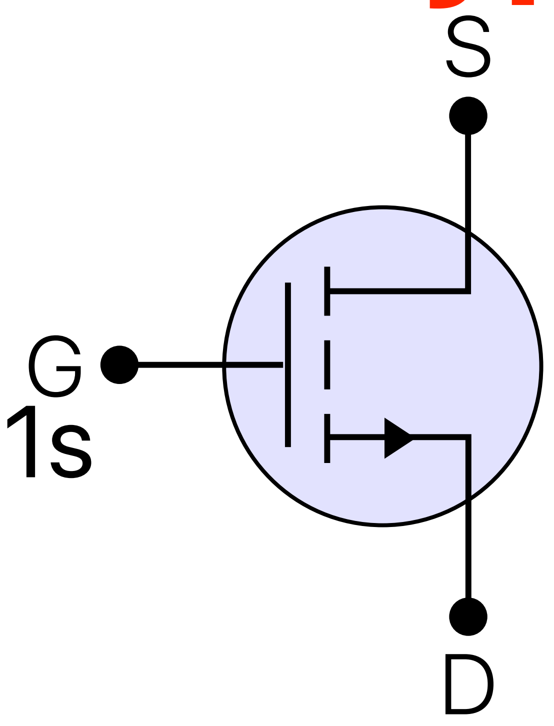
How to express y = e(ab+cd)



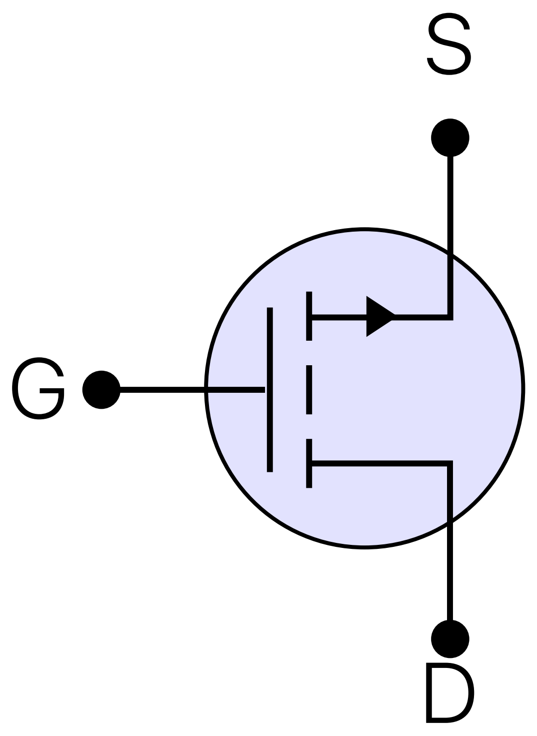
How gates are implemented?

Gates are made by — two type of CMOSs

- nMOS
  - Turns on when G = 1
  - When it's on, passes 0s, but not 1s
  - Connect S to ground (0)
  - Pulldown network

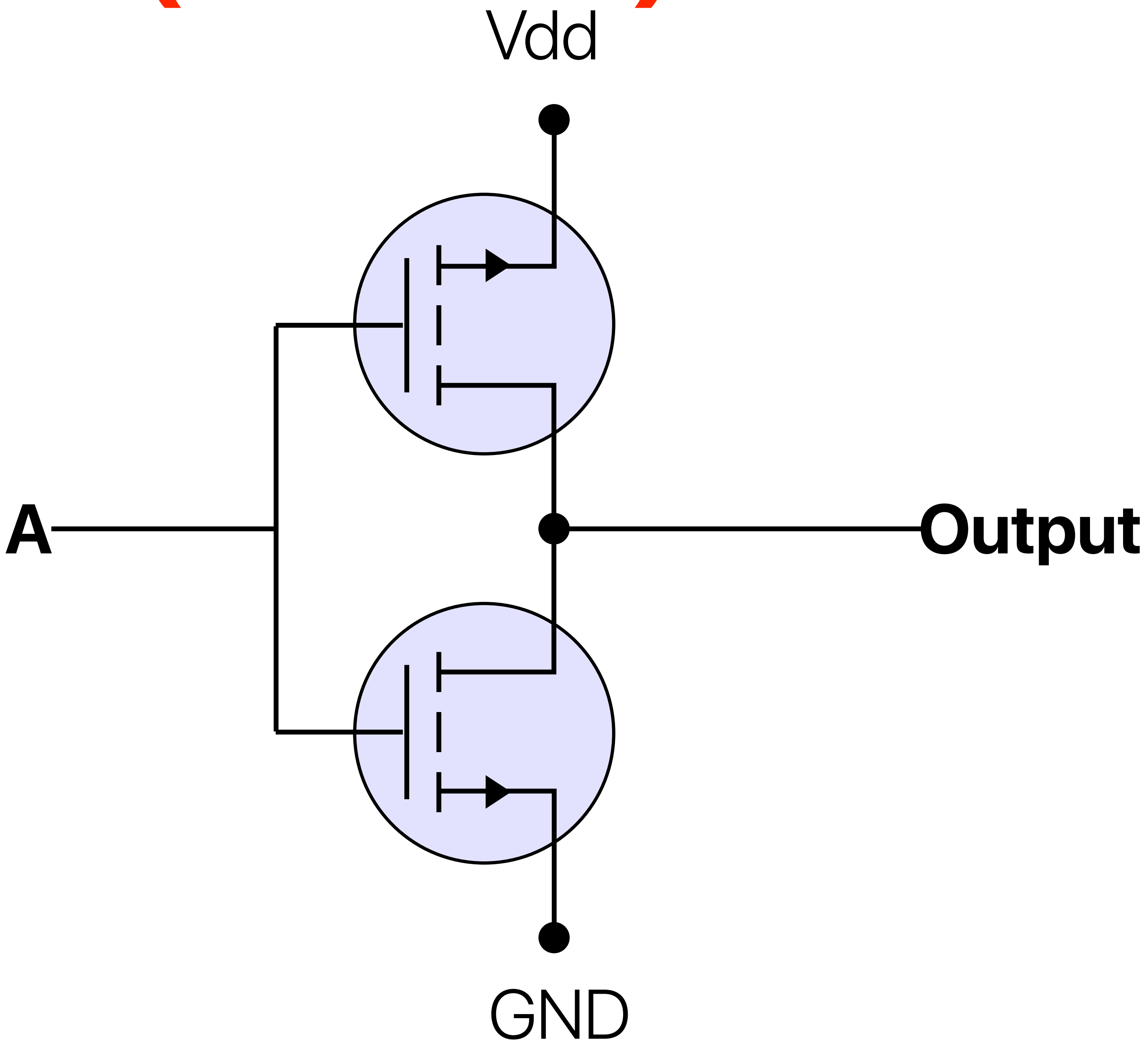


- pMOS
  - Turns on when G = 0
  - When it's on, passes 1s, but not 0s
  - Connect S to Vdd (1)
  - Pullup network

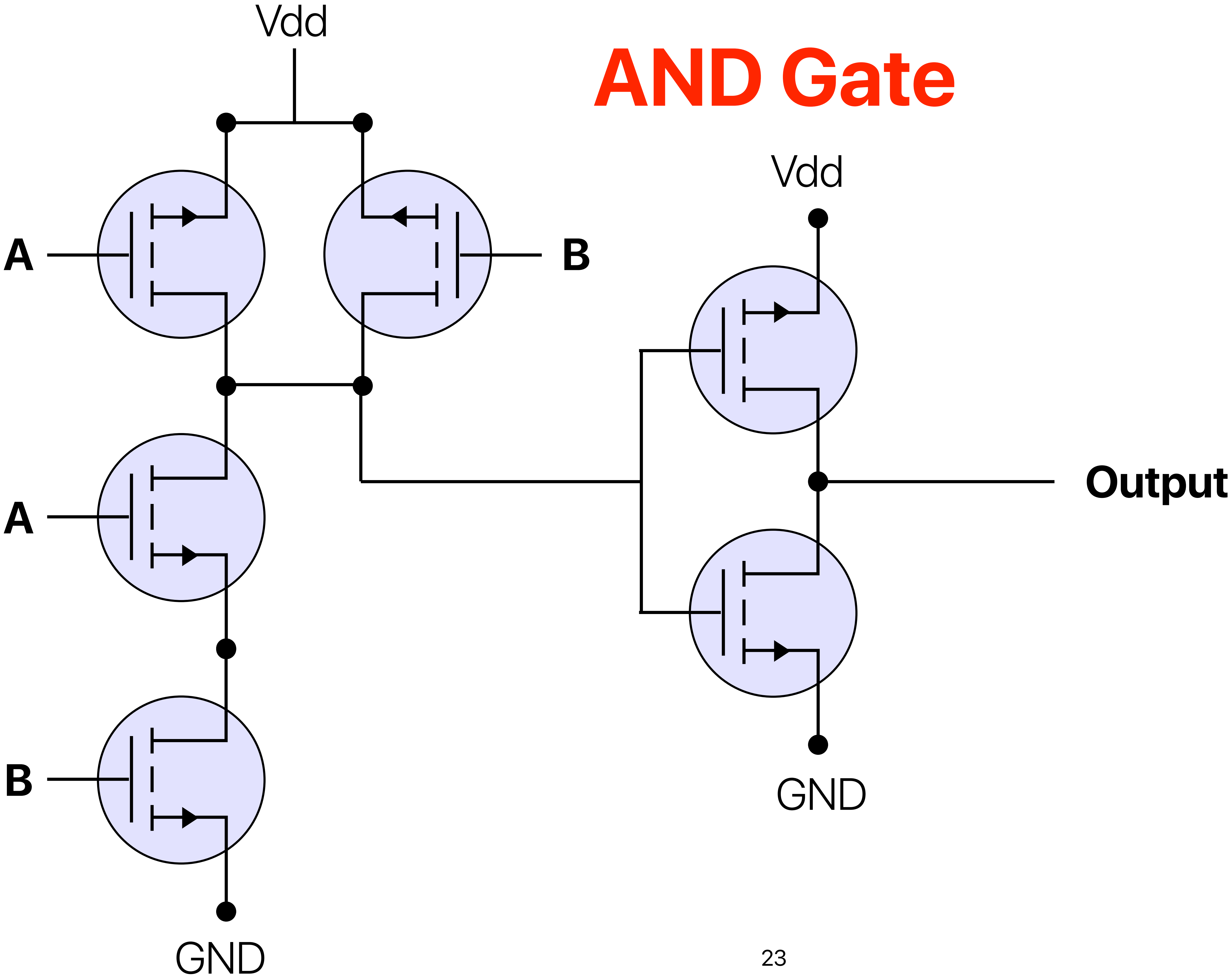


NOT Gate (Inverter)

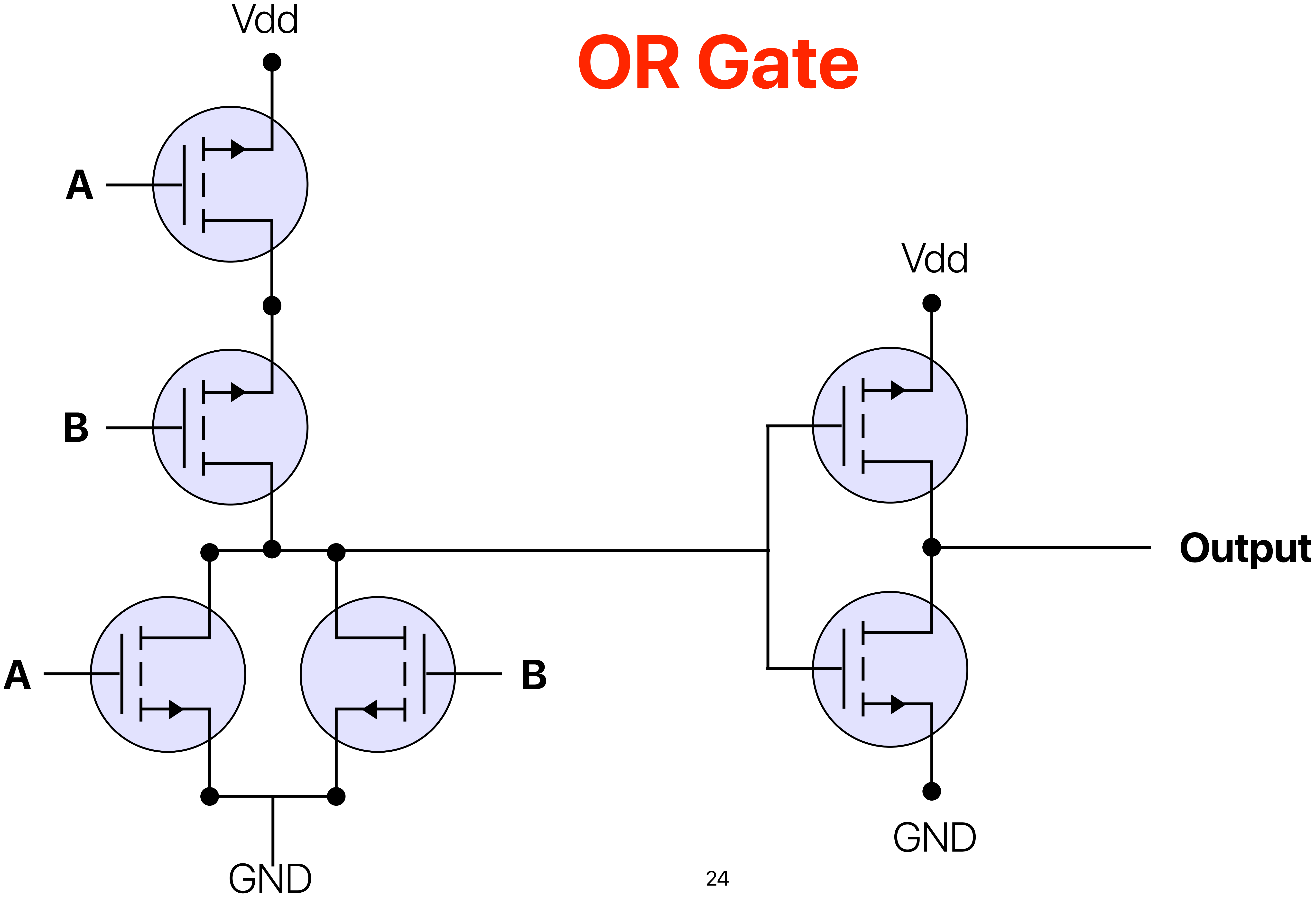
Input	NMOS (passes 0 when on G=1)	PMOS (passes 1 when on G=0)	Output
A			
0	OFF	ON	1
1	ON	OFF	0



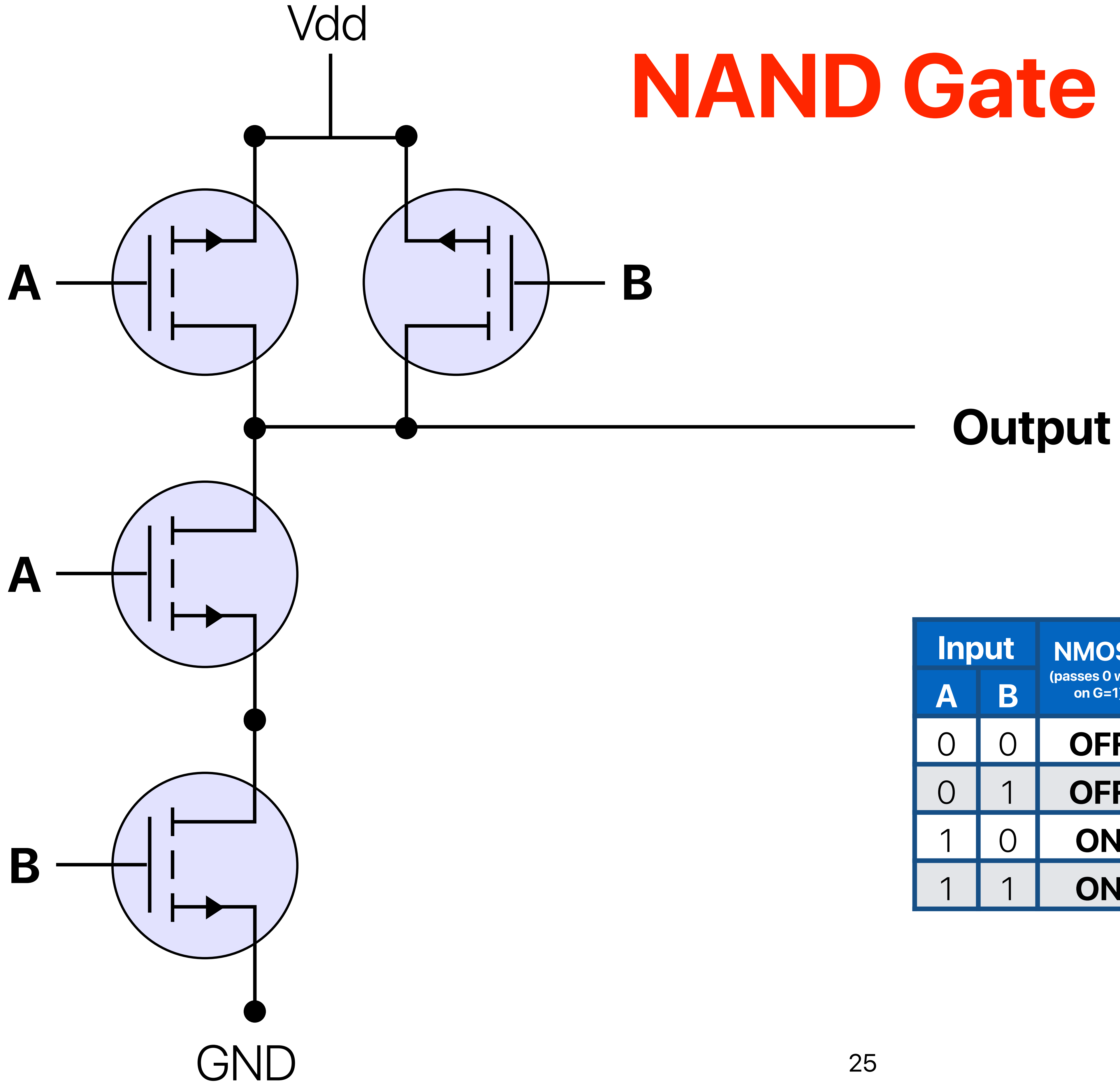
AND Gate



OR Gate



NAND Gate

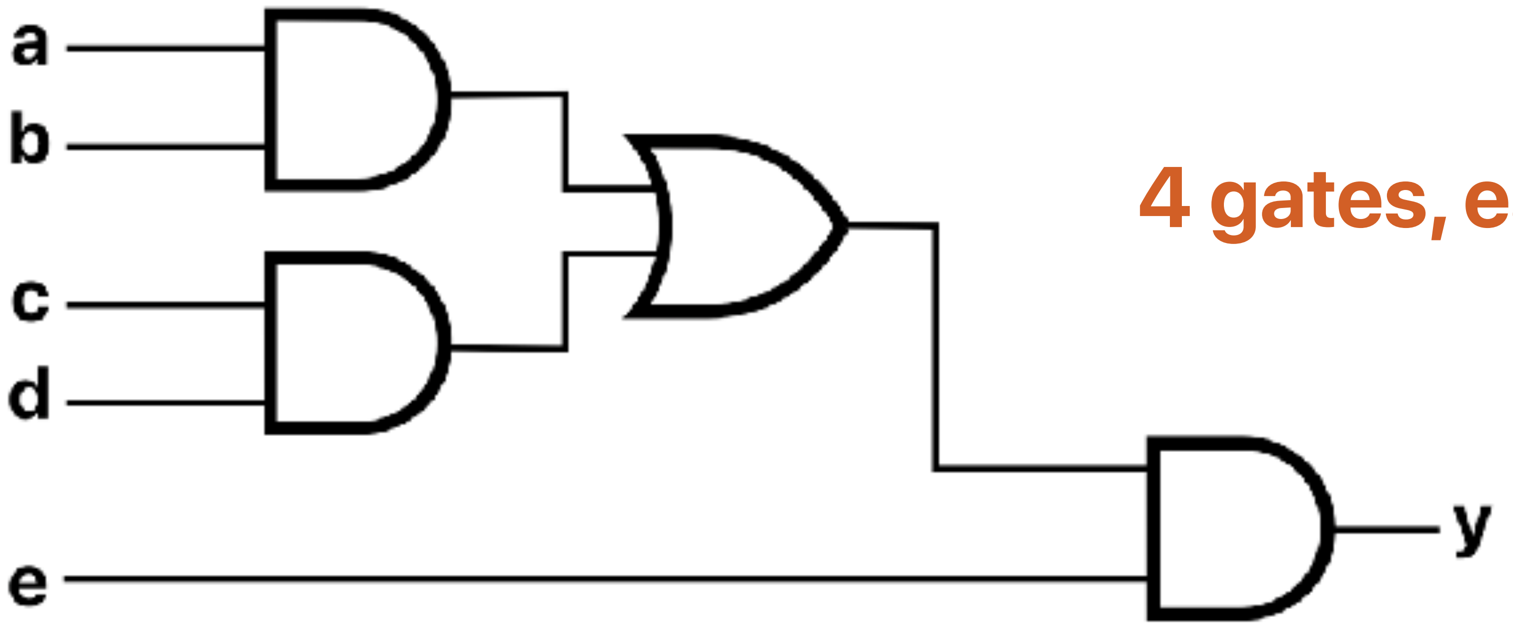


Input		NMOS1 (passes 0 when on G=1)	PMOS1 (passes 1 when on G=0)	NMOS2 (passes 0 when on G=1)	PMOS2 (passes 1 when on G=0)	Output
A	B					
0	0	OFF	ON	OFF	ON	1
0	1	OFF	ON	ON	OFF	1
1	0	ON	OFF	OFF	ON	1
1	1	ON	OFF	ON	OFF	0

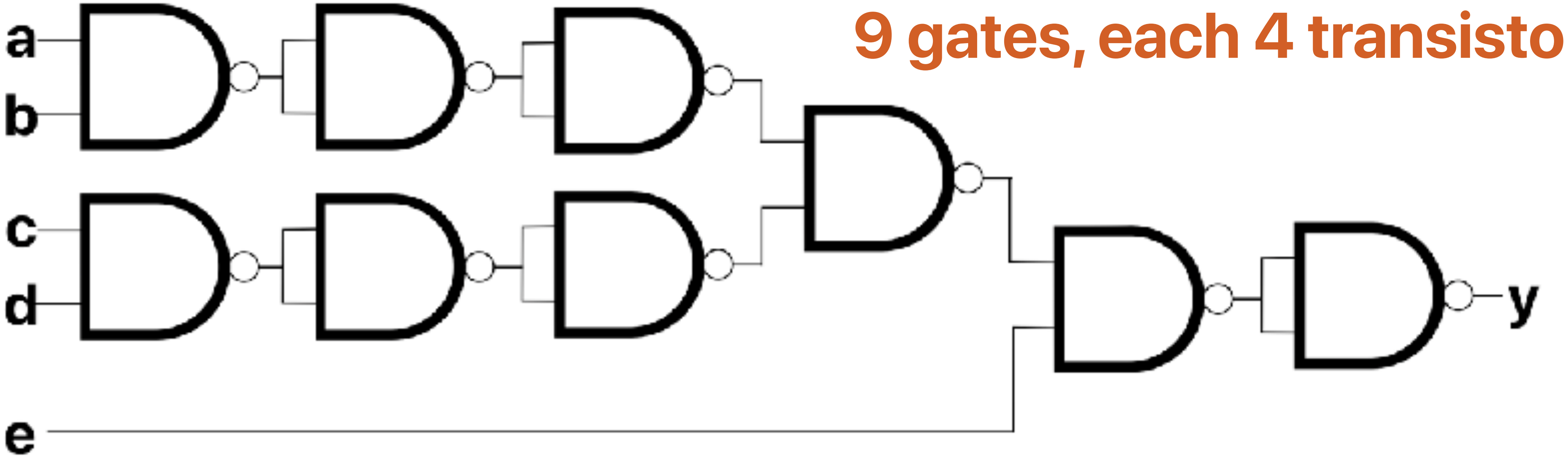
Universal Gates

- NAND and NOR are “universal gates” — you can build any circuit with everything NAND or NOR
- Simplifies the design as you only need one type of gate
- NAND only needs 4 transistors — gate delay is smaller than OR/AND that needs 6 transistors
- NAND is slightly faster than NOR due to the physics nature

How about total number of transistors?



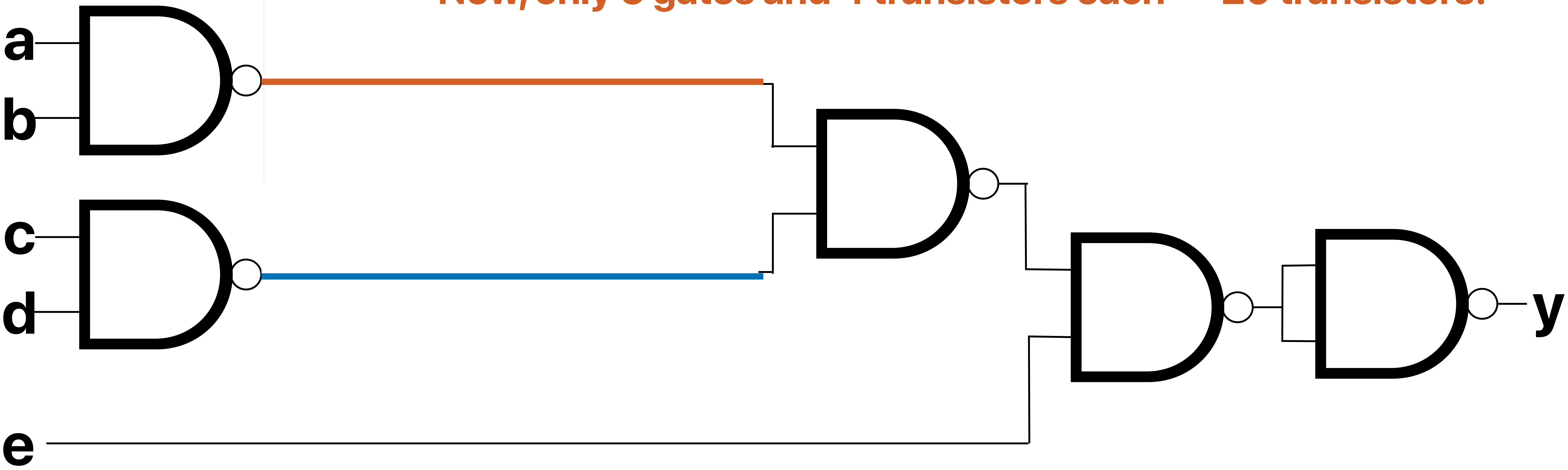
4 gates, each 6 transistors : total 24 transistors



9 gates, each 4 transistors : total 36 transistors

However ...

Now, only 5 gates and 4 transistors each — 20 transistors!

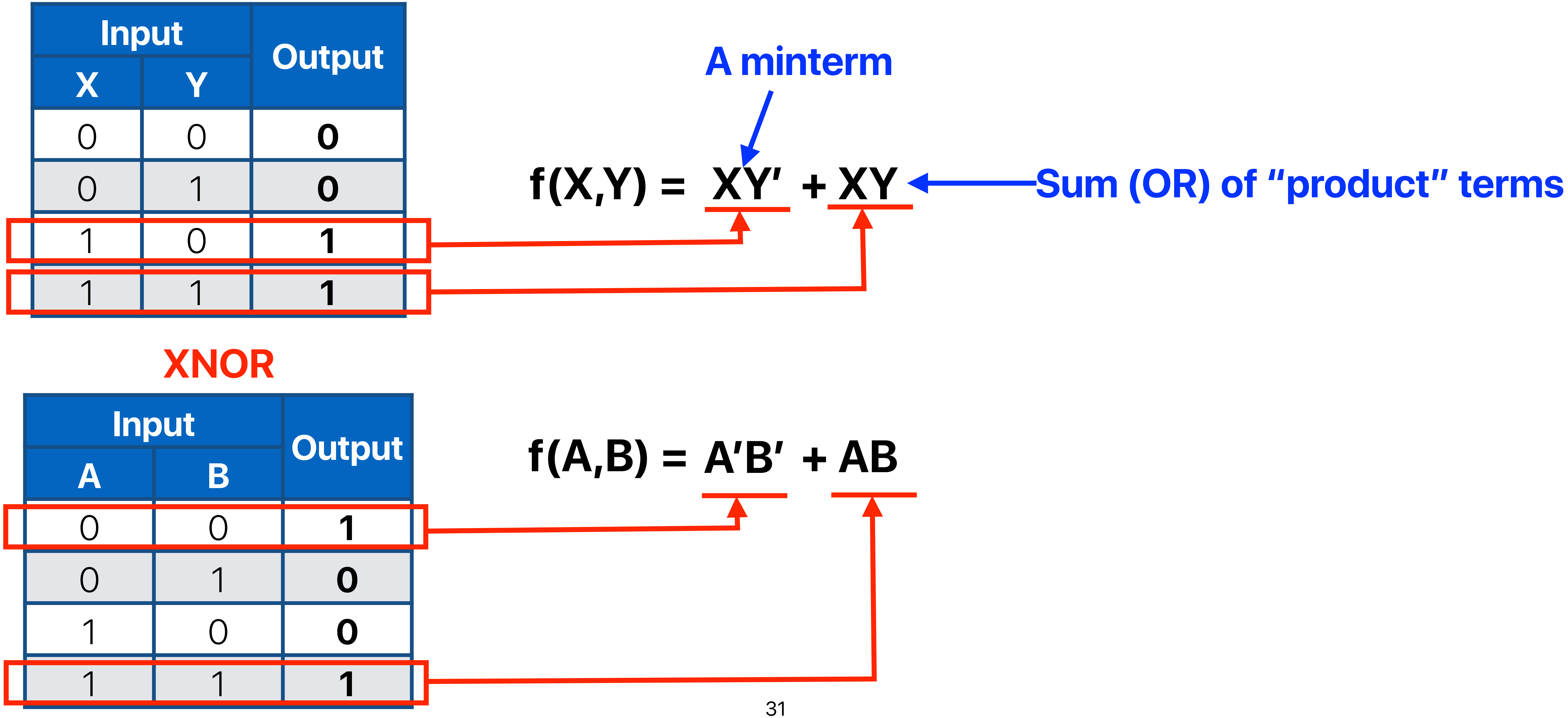


Estimating the size of a design

- 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.

Truth tables —> Boolean functions

Canonical form — Sum of “Minterms”



Input		Output
X	Y	
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**1**

A minterm

$$f(X,Y) = \underline{XY'} + \underline{XY}$$

Sum (OR) of “product” terms

XNOR

Input		Output
A	B	
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

$$f(A,B) = \underline{A'B'} + \underline{AB}$$

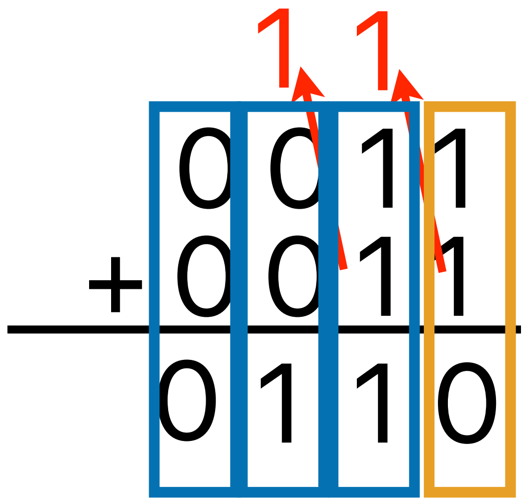
Binary addition

3 + 2 = 5

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

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**

3 + 3 = 6



full adder — adder with a carry as an input

half adder — adder without a carry as an input

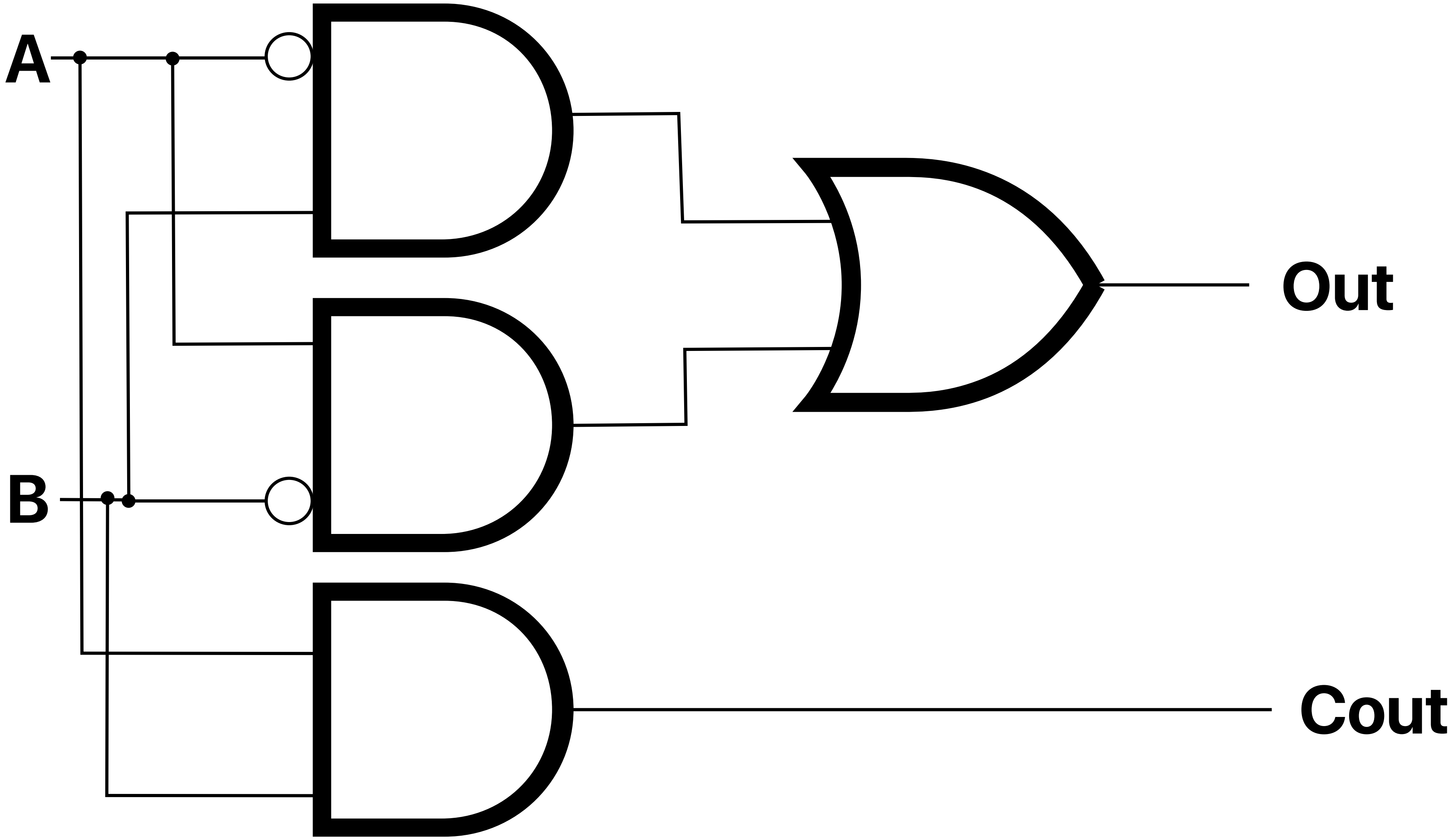
Input		Output	
A	B	Out	Cout
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

Half adder

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

Out = A′B + AB′

Cout = AB



The sum-of-product form of the full adder

- How many of the following minterms are part of the sum-of-product form of the full adder in generating the output bit?

① A′B′Cin′

[✓ ② A′BCin′]

[✓ ③ AB′Cin′]

④ ABCin′

[✓ ⑤ A′B′Cin]

⑥ A′BCin

⑦ AB′Cin

[✓ ⑧ ABCin]

A. 0

B. 1

C. 2

D. 3

[✓ E. 4]

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**

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

Cout = ABCin′ + A′BCin + AB′Cin + ABCin

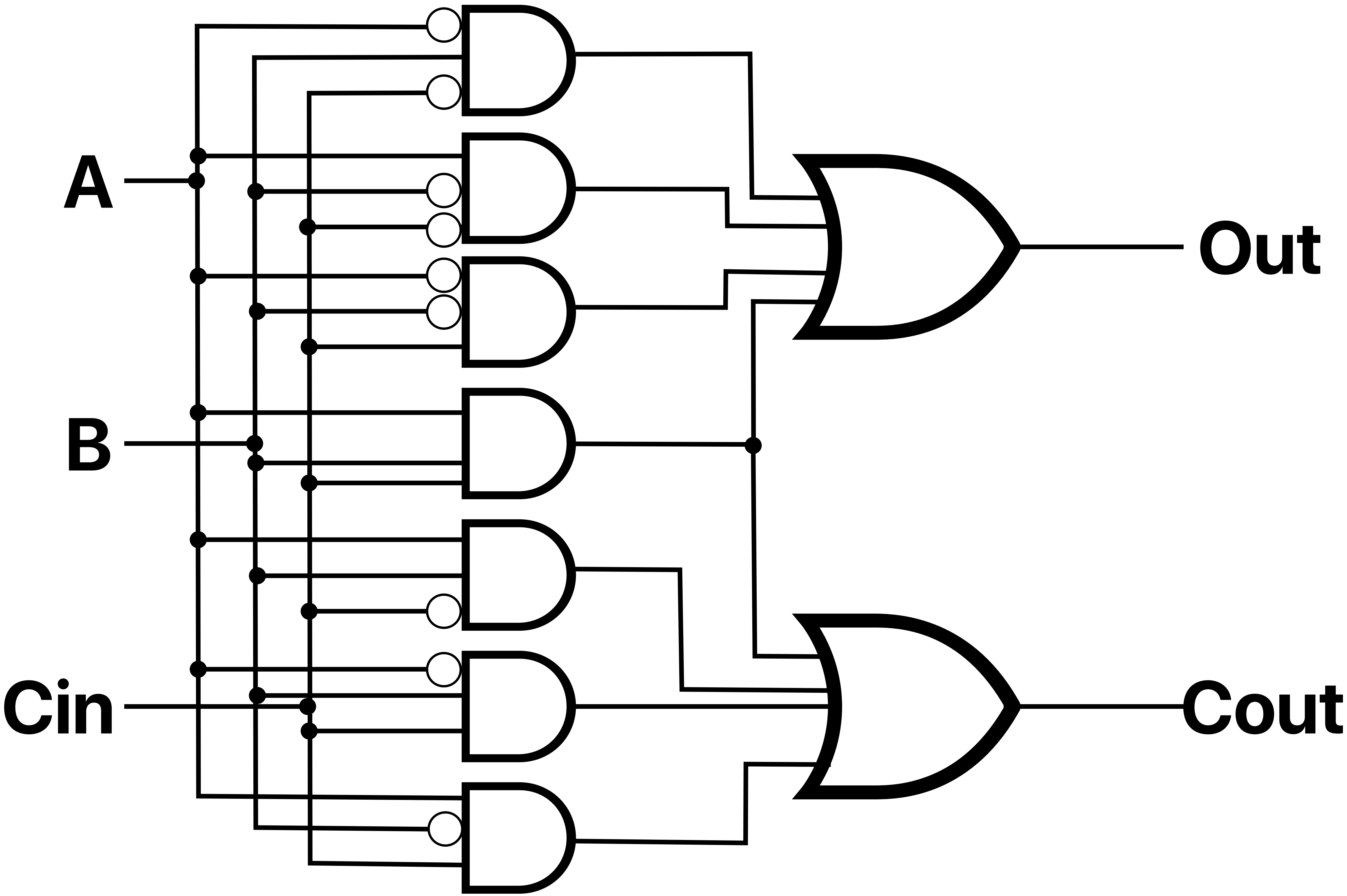
The full adder

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

Cout = ABCin′ + A′BCin + AB′Cin + ABCin

The same

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**



Laws in Boolean Algebra

	OR	AND
Associative laws	$(a+b)+c=a+(b+c)$	$(a\cdot b)\cdot c=a\cdot(b\cdot c)$
Commutative laws	$a+b=b+a$	$a\cdot b=b\cdot a$
Distributive laws	$a+(b\cdot c)=(a+b)\cdot(a+c)$	$a\cdot(b+c)=a\cdot b+a\cdot c$
Identity laws	$a+0=a$	$a\cdot 1=a$
Complement laws	$a+a'=1$	$a\cdot a'=0$
DeMorgan's Theorem	$(a+b)'=a'b'$	$a'b'=(a+b)'$
Covering Theorem	$a(a+b)=a+ab=a$	$ab+ab'=(a+b)(a+b')=a$
Consensus Theorem	$ab+ac+b'c=ab+b'c$	$(a+b)(a+c)(b'+c)=(a+b)(b'+c)$
Uniting Theorem	$a(b+b')=a$	$(a+b)\cdot(a+b')=a$
Shannon's Expansion	$f(a,b,c)=a'b'+bc+ab'c$ $f(a,b,c)=a\,f(1,b,c)+a'\,f(0,b,c)$	

How many "OR"s?

- For the truth table shown on the right, what's the minimum number of "OR" gates we need?

[✓ A. 1]
B. 2
C. 3
D. 4
E. 5

F(A, B, C) =

Uniting Theorem

$$\boxed{A'B'C' + A'B'C} + \boxed{A'BC' + A'BC} + \boxed{AB'C' + ABC'}$$

$$= A'B'(C'+C) + A'B(C'+C) + AC'(B'+B)$$

$$= A'B' + A'B + AC'$$

$$= A' + AC' = A'(1+C') + AC'$$

Distributive Laws

$$= A' + A'C' + AC'$$

$$= A' + (A'+A)C'$$

$$= A' + C'$$

Input			Output
A	B	C	
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

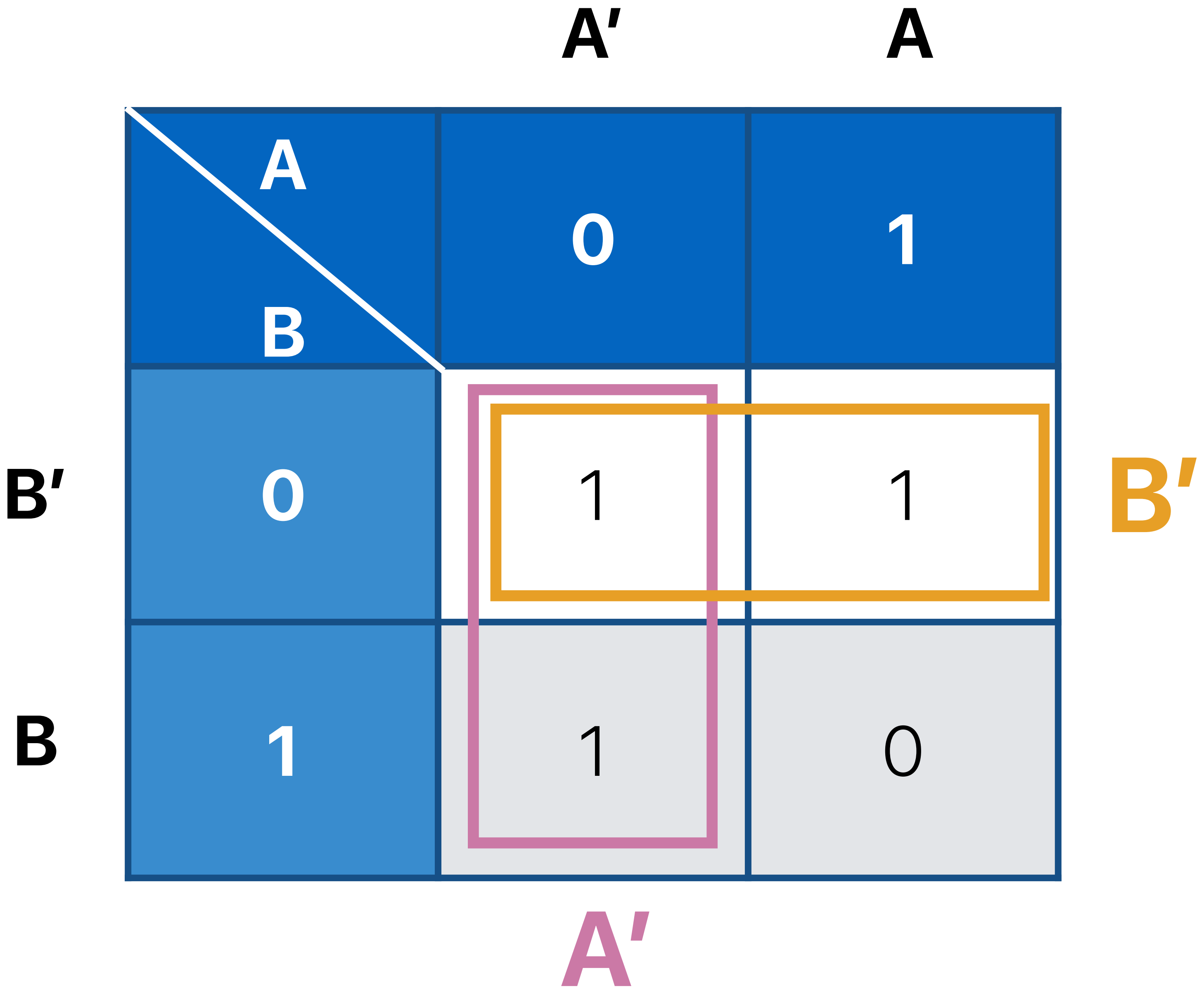
Simplifying circuits using Karnaugh maps

Karnaugh maps

- Alternative to truth-tables to help visualize adjacencies
- Guide to applying the uniting theorem
- Steps
  - Create a 2-D truth table with input variables on each dimension, and adjacent column(j)/row(i) only change one bit in the variable.
  - Fill each (i,j) with the corresponding result in the truth table
  - Identify ON-set (all 1s) with size of power of 2 (i.e., 1, 2, 4, 8, …) and "unite" them terms together (i.e. finding the "common literals" in their minterms)
  - Find the "minimum cover" that covers all 1s in the graph
  - Sum with the united product terms of all minimum cover ON-sets

2-variable K-map example

Input		Output
A	B	
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**



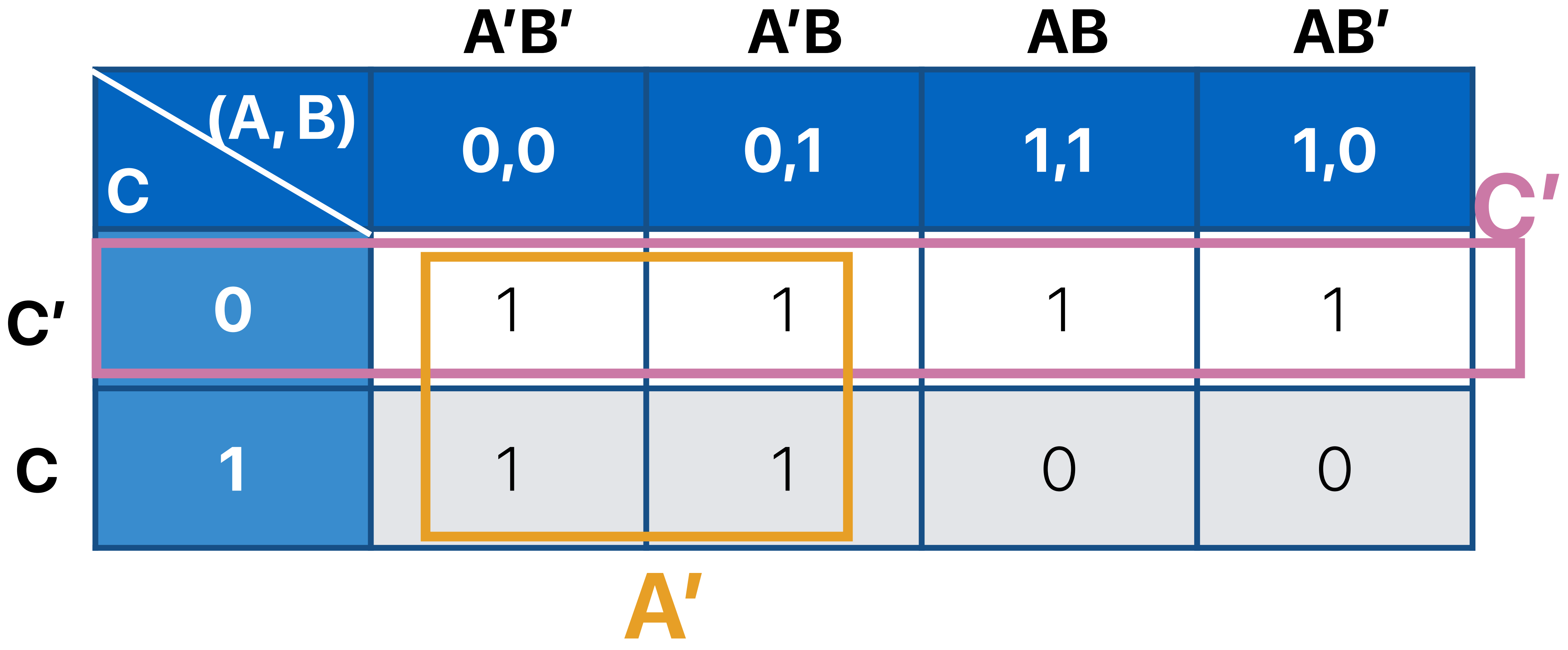
		A′	A
	A B	0	1
B′	0	1	1
B	1	1	0

$$F(A, B) = A' + B'$$

3-variable K-map?

- Reduce to 2-variable K-map — 1 dimension will represent two variables
- Adjacent points should differ by only 1 bit
  - So we only change one variable in the neighboring column
  - 00, 01, 11, 10 — such numbering scheme is so-called **Gray–code**

Input			Output
A	B	C	
0	0	0	**1**
0	0	1	**1**
0	1	0	**1**
0	1	1	**1**
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	**0**



		A′B′	A′B	AB	AB′
	C \ (A, B)	0,0	0,1	1,1	1,0
C′	0	1	1	1	1
C	1	1	1	0	0

F(A, B, C) = A′ + C′

Valid K-Maps

- How many of the followings are "valid" K-Maps?

(1) ✓

	0,0	0,1	1,1	1,0
0	0	1	0	1
1	1	0	1	0

(2) ✓

	0,1	1,1	1,0	0,0
0	1	0	1	0
1	0	1	0	1

(3)

	1,1	1,0	0,1	0,0
0	0	1	1	0
1	1	0	0	1

(4)

	0,0	0,1	1,0	1,1
0	0	1	1	0
1	1	0	0	1

A. 0

B. 1

[✓ C. 2]

D. 3

E. 4

Minimum SOP for a full adder

- Minimum number of SOP terms to cover the “Cout” function for a one-bit full adder?

  A. 1

  B. 2

  [✓ C. 3]

  D. 4

  E. 5

		A′B′	A′B	AB	AB′
	Out(A, B)	**0,0**	**0,1**	**1,1**	**1,0**
Cin′	**0**	0	0	1	0
Cin	**1**	0	1	1	1

BCin **AB** **ACin**

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**

Minimum number of SOP terms

- Minimum number of SOP terms to cover the following function?

  A. 1

  [✓ B. 2]

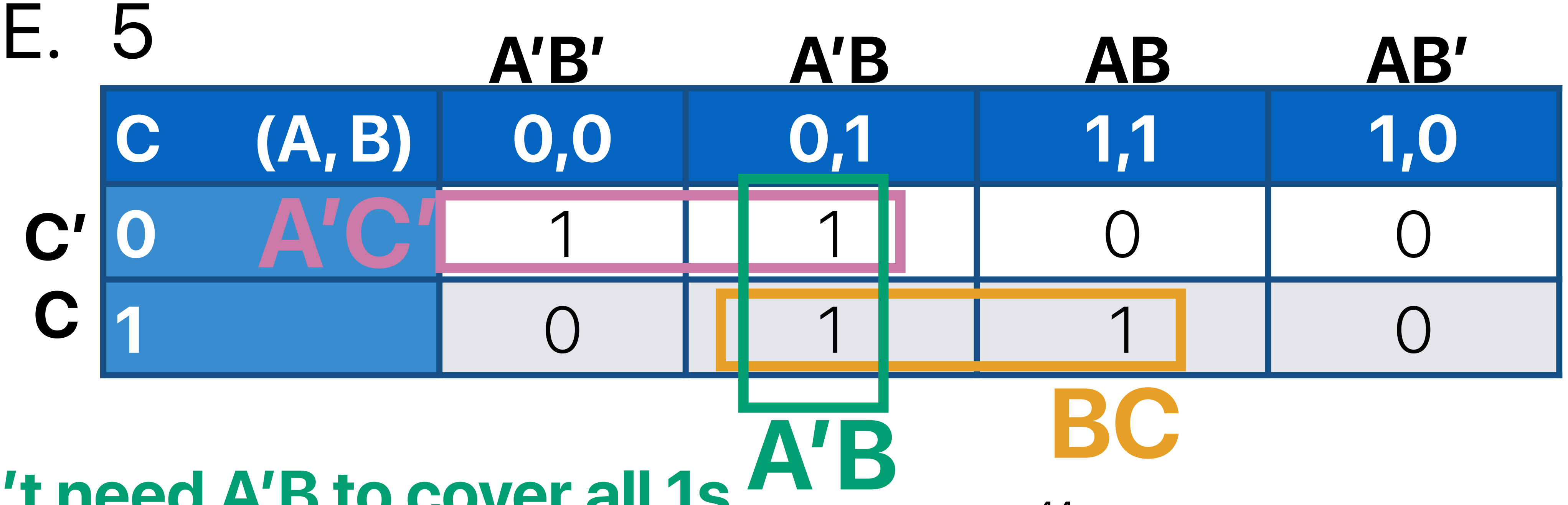
  C. 3

  D. 4

  E. 5

Input			Output
A	B	C	
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**0**
1	1	0	**0**
1	1	1	**1**

F(A, B, C) = A′C′ + BC′

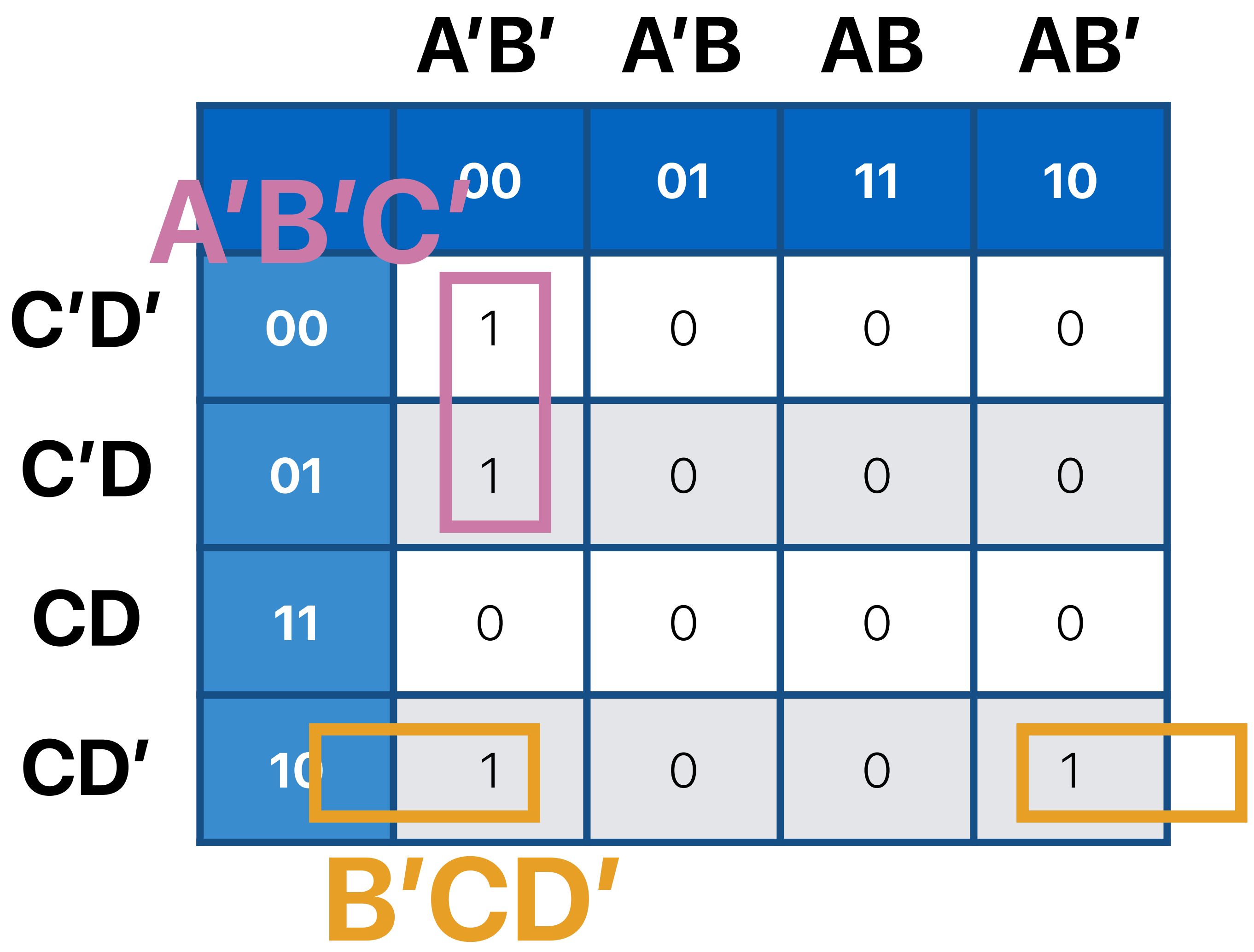


		A′B′	**A′B**	**AB**	**AB′**
	C (A, B)	**0,0**	**0,1**	**1,1**	**1,0**
C′	0	1	1	0	0
C	1	0	1	1	0

We don’t need A′B to cover all 1s

4-variable K-map

- Reduce to 2-variable K-map — both dimensions will represent two variables
- Adjacent points should differ by only 1 bit
  - So we only change one variable in the neighboring column
  - Use Gray-coding — 00, 01, 11, 10



		A′B′	A′B	AB	AB′
		00	01	11	10
C′D′	00	1	0	0	0
C′D	01	1	0	0	0
CD	11	0	0	0	0
CD′	10	1	0	0	1

F(A, B, C) = A′B′C′+B′CD′

4-variable K-map

- What′s the minimum sum-of-products expression of the given K-map?

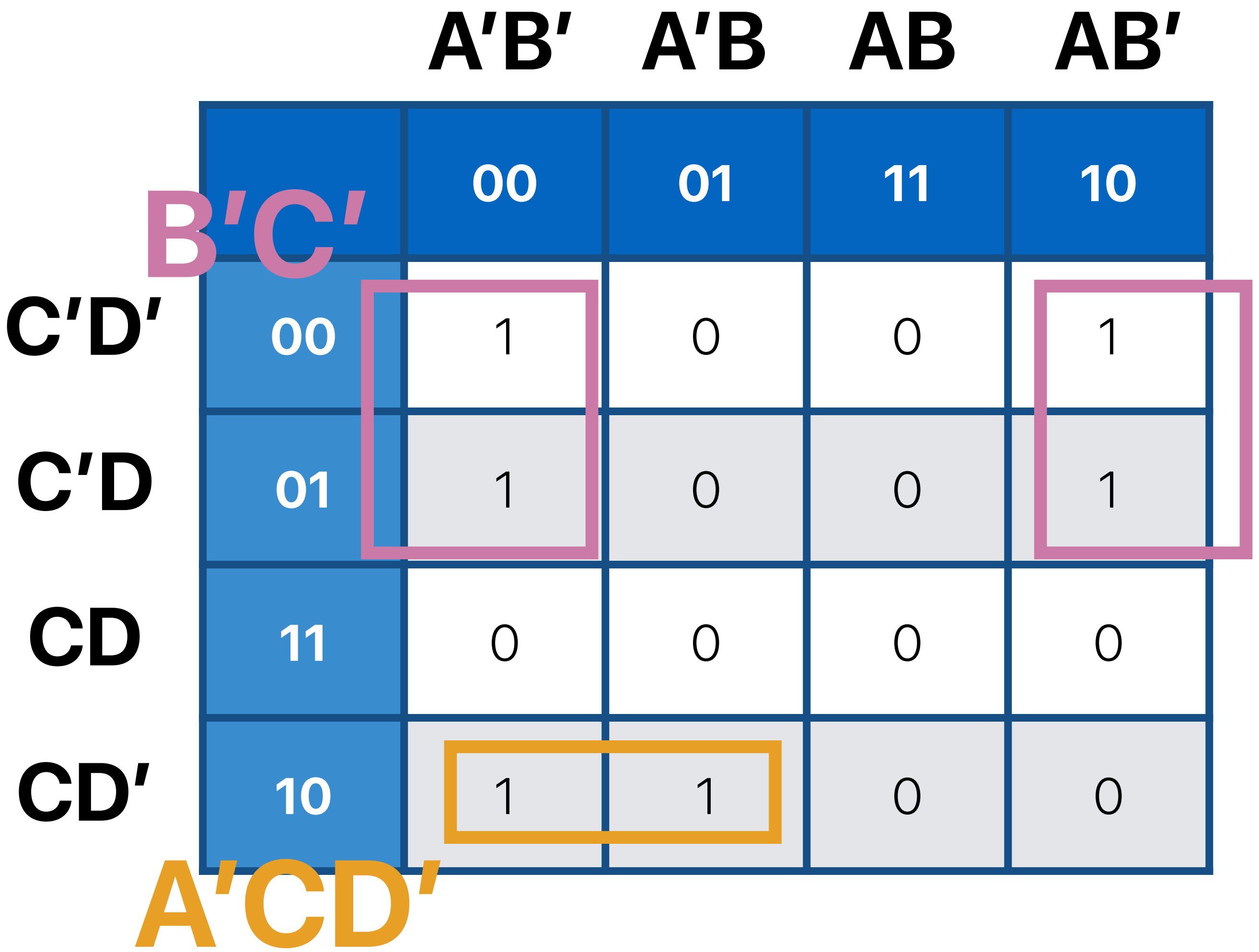
  A. B′C′ + A′B′

  B. B′C′D′ + A′B′ + B′C′D′

  C. A′B′CD′ + B′C′

  D. AB′ + A′B′ + A′B′D′

  [✓ E. B′C′ + A′C′D′]



		A′B′	A′B	AB	AB′
		00	01	11	10
C′D′	00	1	0	0	1
C′D	01	1	0	0	1
CD	11	0	0	0	0
CD′	10	1	1	0	0

Incompletely Specified Functions

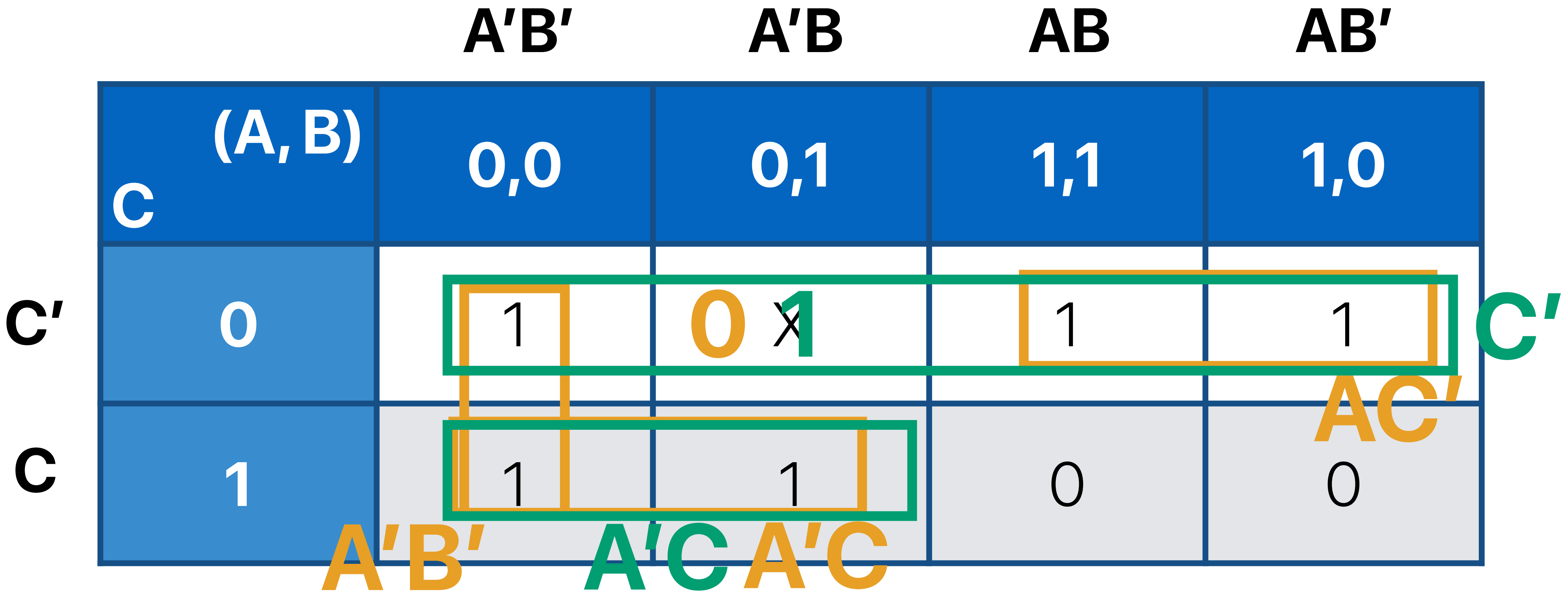
- Situations where the output of a function can be either 0 or 1 for a particular combination of inputs
- This is specified by a don’t care in the truth table
- This happens when
    - The input does not occur. e.g. Decimal numbers 0… 9 use 4 bits, so (1,1,1,1) does not occur.
    - The input may happen but we don’t care about the output. E.g. The output driving a seven segment display – we don’t care about illegal inputs (greater than 9)

Don’t care

A B	0	1
0	0	0
1	1	X

K-Map with “Don′t Care”s

You can treat “X” as either 0 or 1
— depending on which is more advantageous



		A′B′	A′B	AB	AB′
	C (A, B)	0,0	0,1	1,1	1,0
C′	0	1	X	1	1
C	1	1	1	0	0

If we treat the “X” as 0?

F(A,B,C)=A′B′+A′C+AC′

If we treat the “X” as 1?

F(A,B,C) = C′ + A′C

Digital Arithmetics

What do we want from a number system?

- Obvious representation of 0, 1, 2, ...
- Represent positive/negative/integer/floating points
- Efficient usage of number space
- Equal coverage of positive and negative numbers
- Easy hardware design
  - Minimize the hardware cost/reuse the same hardware as much as possible
  - Easy to distinguish positive numbers
  - Easy to negation

The third proposal — 2's complement

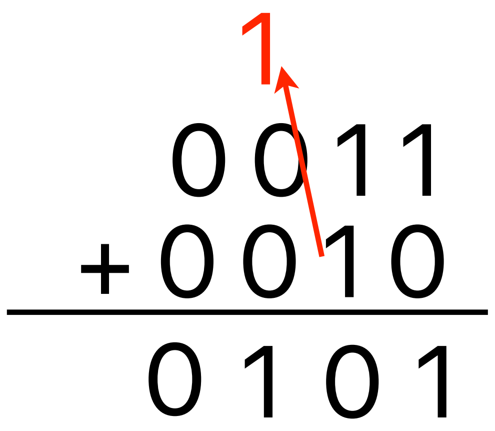
- How many of the following goals can "2's complement — **take the 1's complement of corresponding positive number and then +1**" to represent a negative number fulfill in the number system?
  - ✓ ① Obvious representation of 0, 1, 2, ...
  - ✓ ② Efficient usage of number space
  - ✓ ③ Equal coverage of positive and negative numbers
  - ✓ ④ Easy hardware design
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - [✓ E. 4]

Decimal	Binary	Decimal	Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

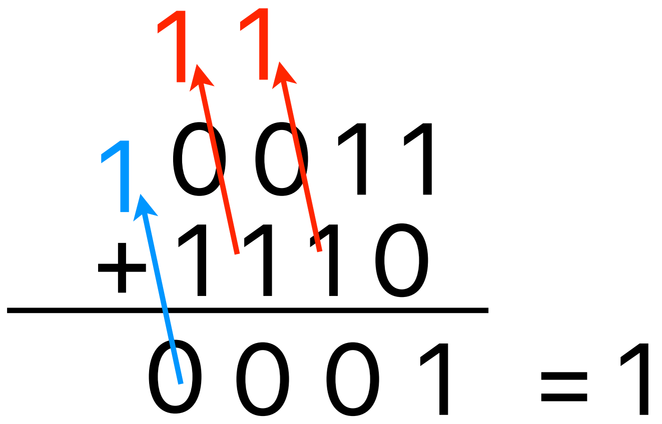
Evaluating 2's complement

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

- 3 + 2 = 5



- 3 + (-2) = 1



A. No. The same procedure applies

[✓ B. No. The same "procedure" applies but it changes overflow detection]

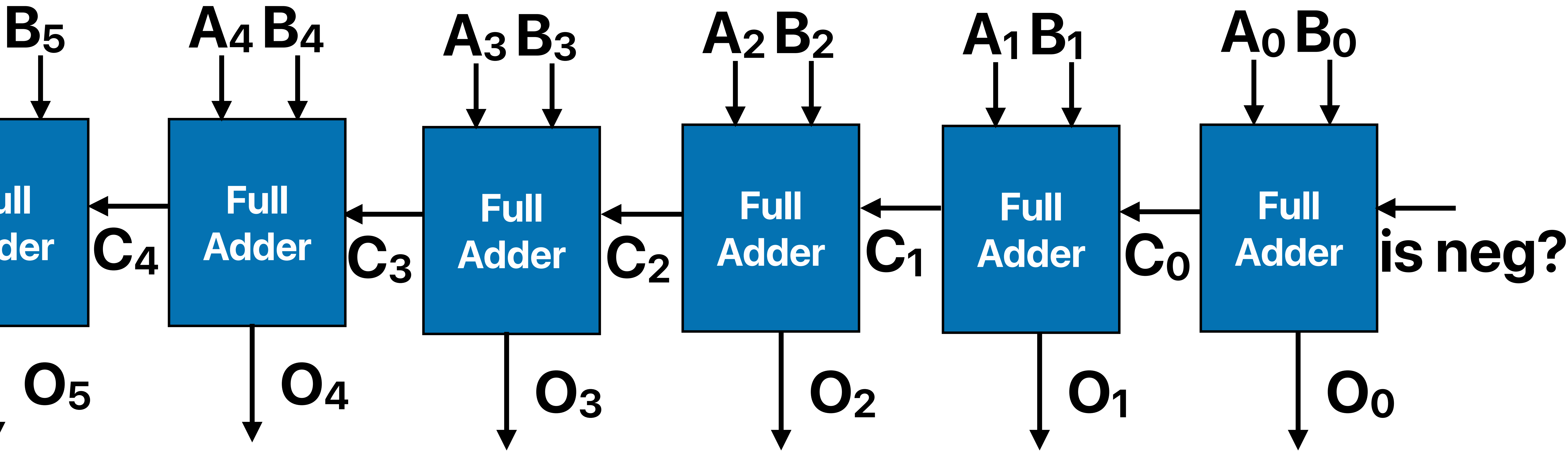
C. Yes, and we need a new procedure

D. Yes, and we need a new procedure and a new hardware

E. None of the above

Adder

We can support more bits!



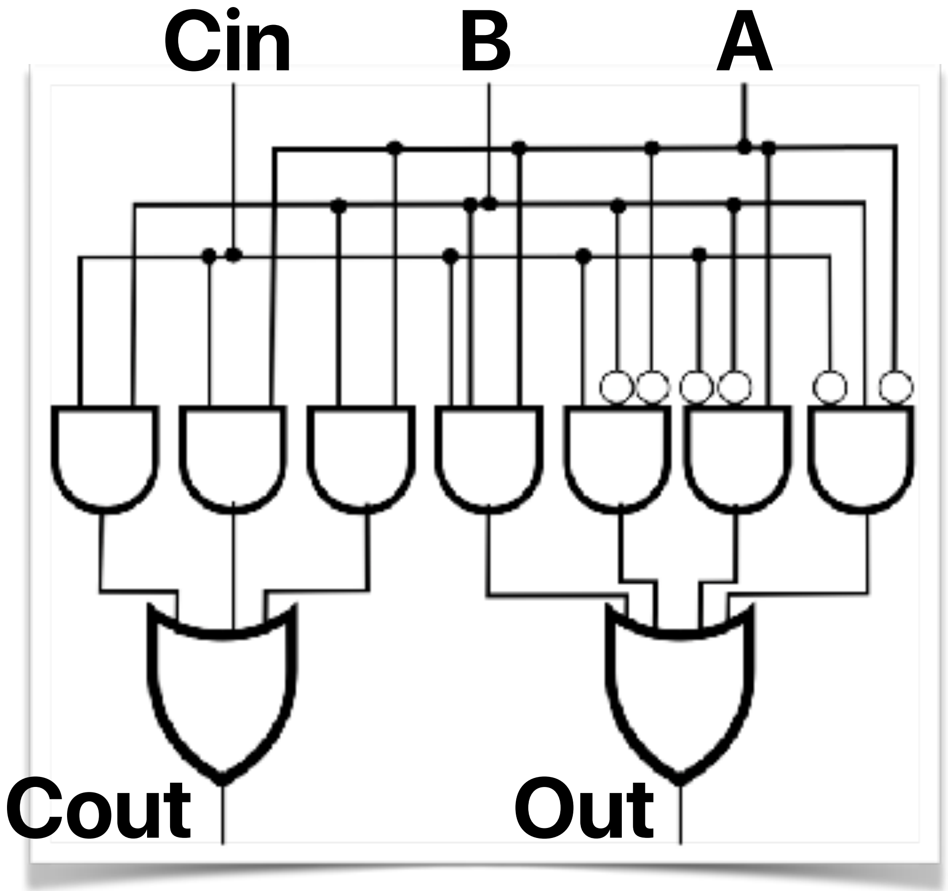
Recap: Full Adder

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**

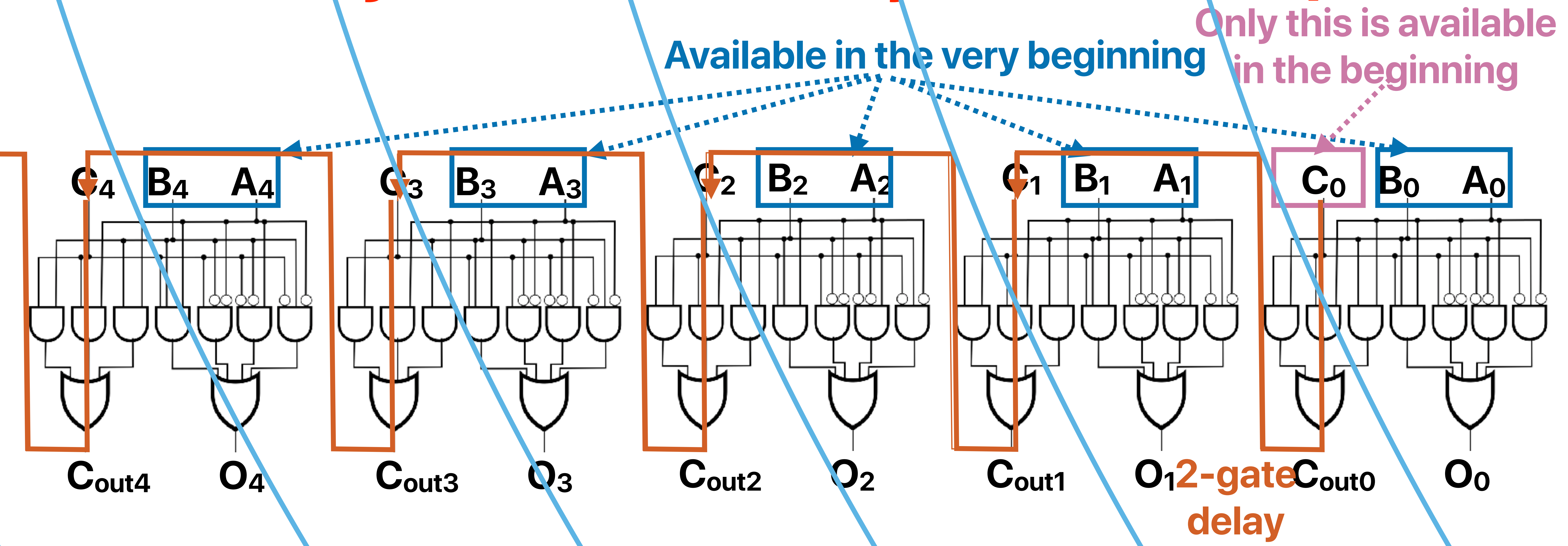
		A′B′	A′B	AB	AB′
	Cout(A, B)	0,0	0,1	1,1	1,0
Cin′	0	0	0	1	0
Cin	1	0	1	1	1

BCin AB ACin

		A′B′	A′B	AB	AB′
	Out(A, B)	0,0	0,1	1,1	1,0
Cin′	0	0	1	0	1
Cin	1	1	0	1	0



The delay is determined by the “critical path”



Carry-Ripple Adder

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?

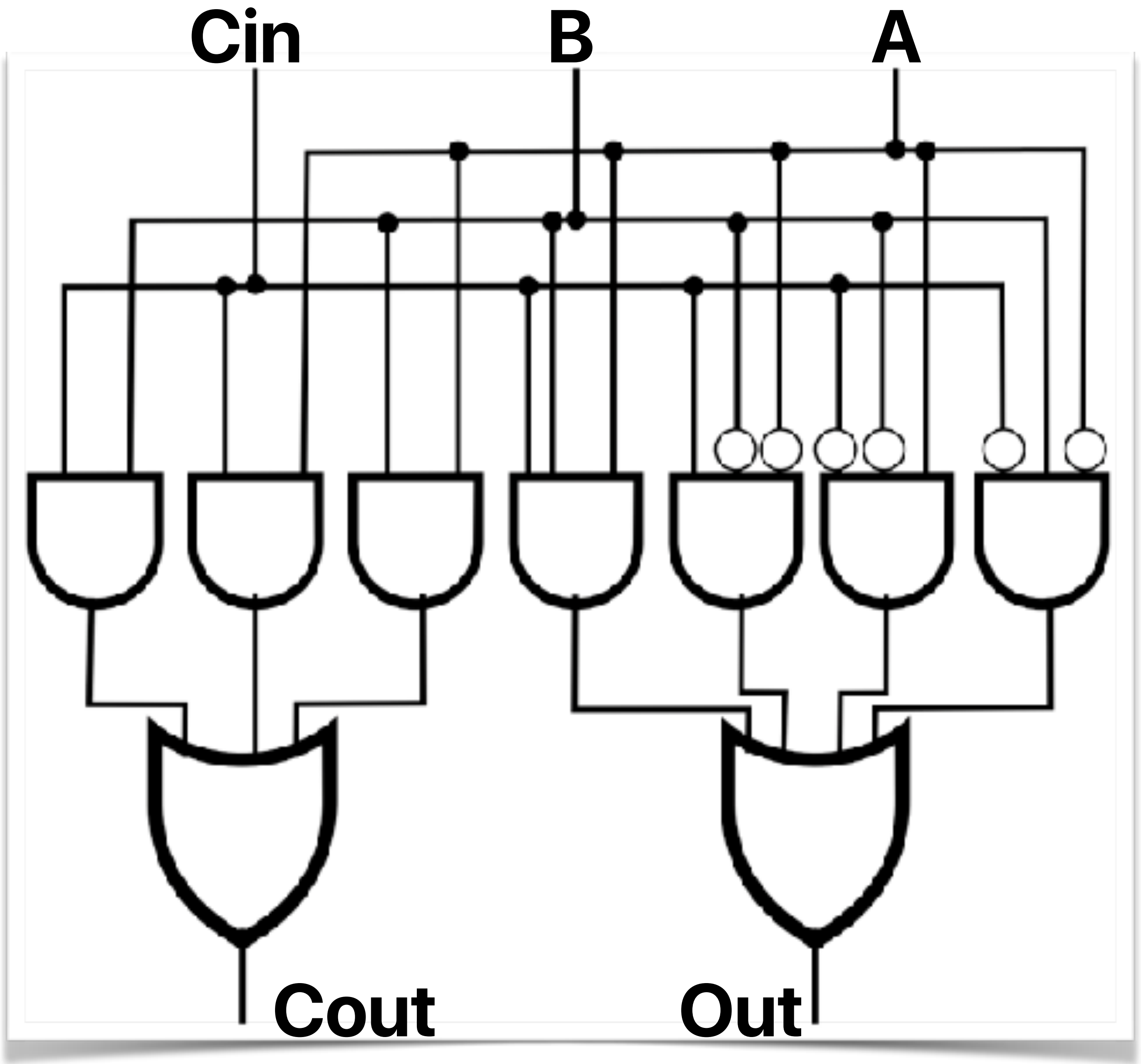
A. 2

B. 32

[✓ C. 64]

D. 128

E. 288



Carry-lookahead adder

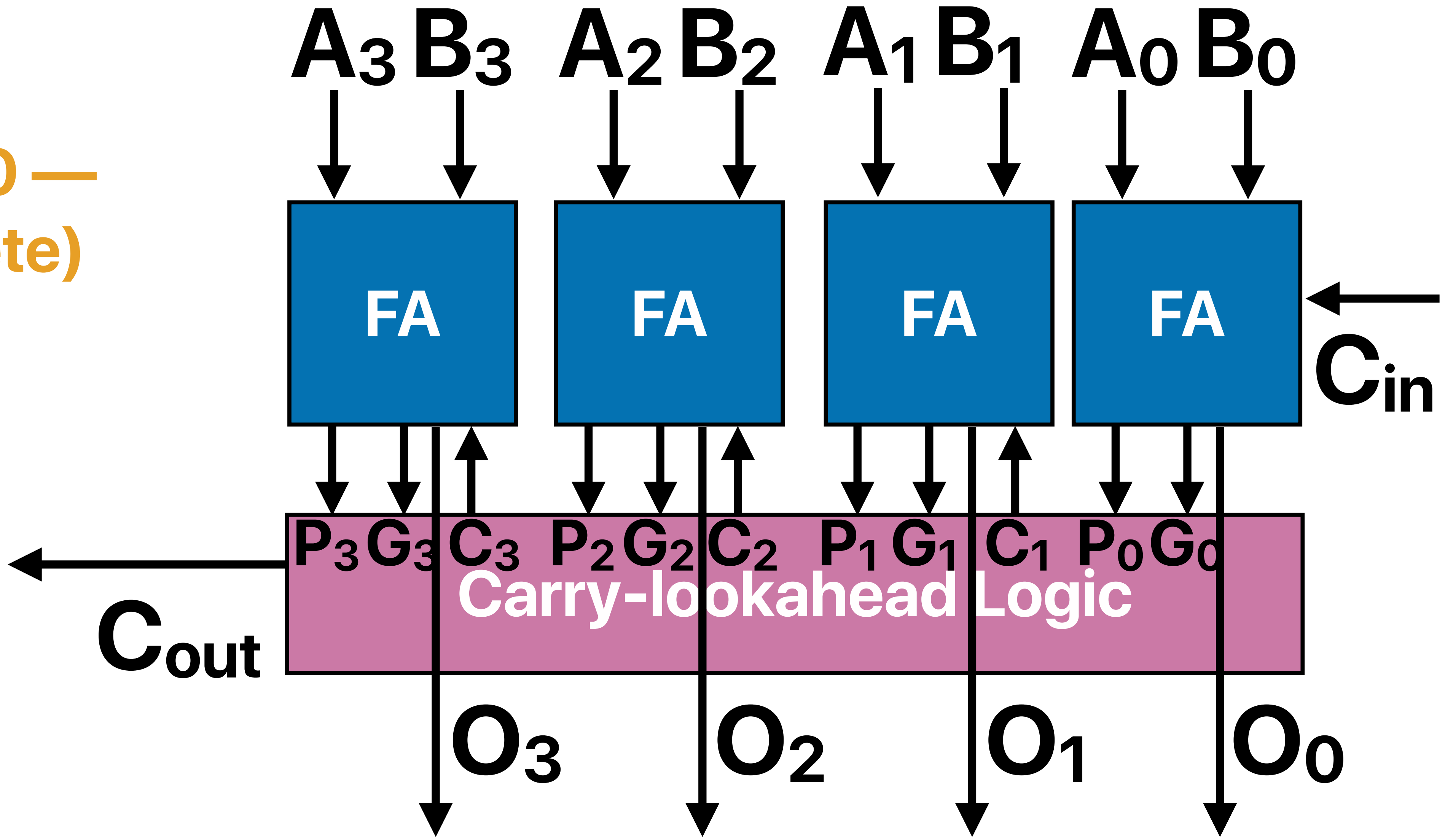
- Uses logic to quickly pre-compute the carry for each digit

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**

Both A, B are 0 — no carry (Delete)

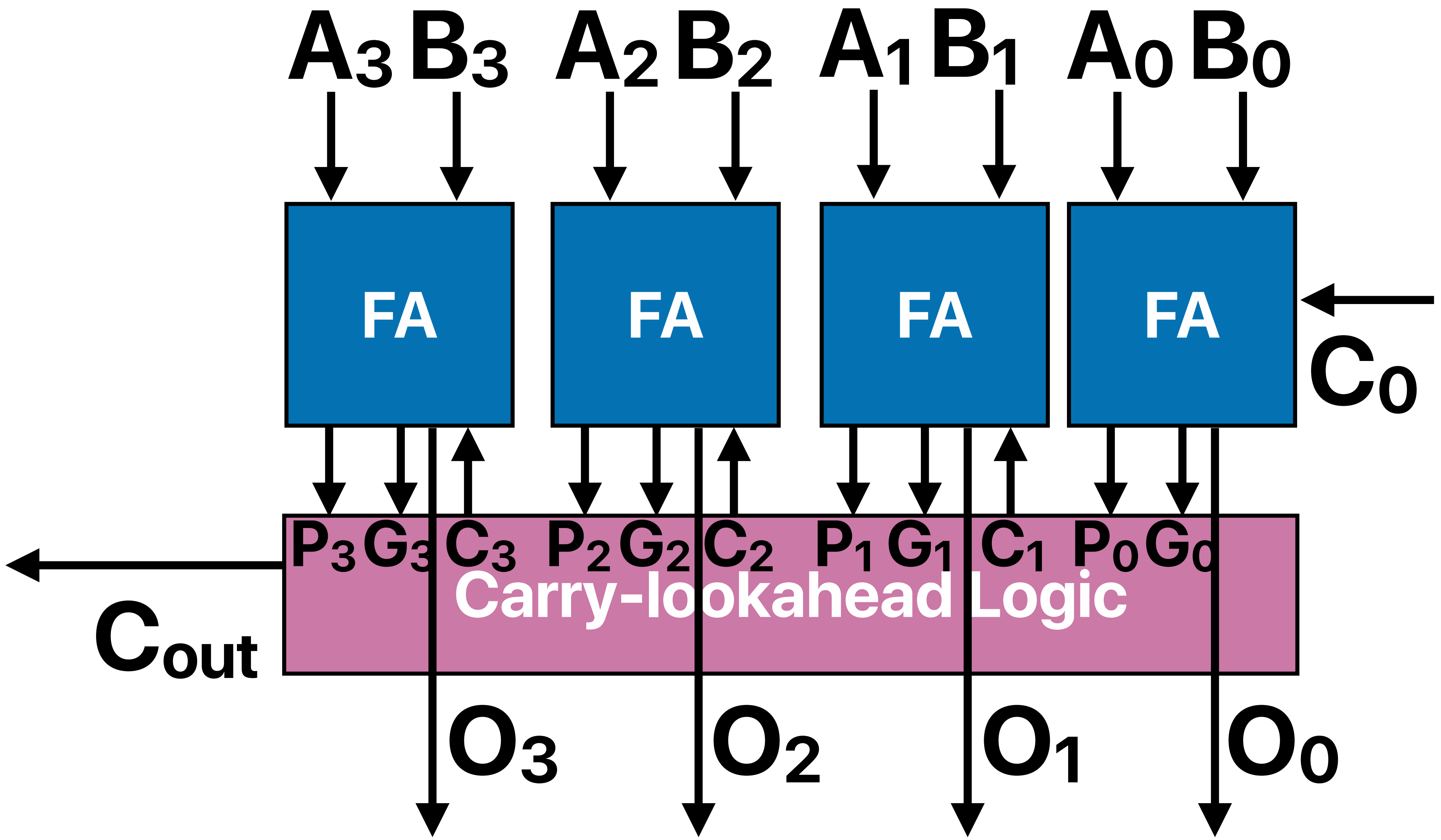
Needs to wait Cin (Propagate)

Both A, B are 1 — must carry (Generate)


A_3 B_3 A_2 B_2 A_1 B_1 A_0 B_0
FA
FA
FA
FA
C_{in}
P_3 G_3 C_3 P_2 G_2 C_2 P_1 G_1 C_1 P_0 G_0
Carry-lookahead Logic
C_{out}
O_3
O_2
O_1
O_0

CLA (cont.)

- All "G" and "P" are immediately available (only need to look over Ai and Bi), but "c" are not (except the c0).



A_3 B_3 A_2 B_2 A_1 B_1 A_0 B_0
FA
FA
FA
FA
C_0
P_3 G_3 C_3 P_2 G_2 C_2 P_1 G_1 C_1 P_0 G_0
Carry-lookahead Logic
C_{out}
O_3 O_2 O_1 O_0

$G_i = A_iB_i$

$P_i = A_i \text{ XOR } B_i$

$C_1 = G_0 + P_0 C_0$

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

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2 C_2$

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

$C_4 = G_3 + P_3 C_3$

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

CLA's gate delay

- What's the gate-delay of a 4-bit CLA?
  - [✓ A. 2]
  - B. 4
  - C. 6
  - D. 8
  - E. 10

$G_i = A_iB_i$

$P_i = A_i \text{ XOR } B_i$

$C_1 = G_0 + P_0 C_0$

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

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2 C_2$

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

$C_4 = G_3 + P_3 C_3$

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

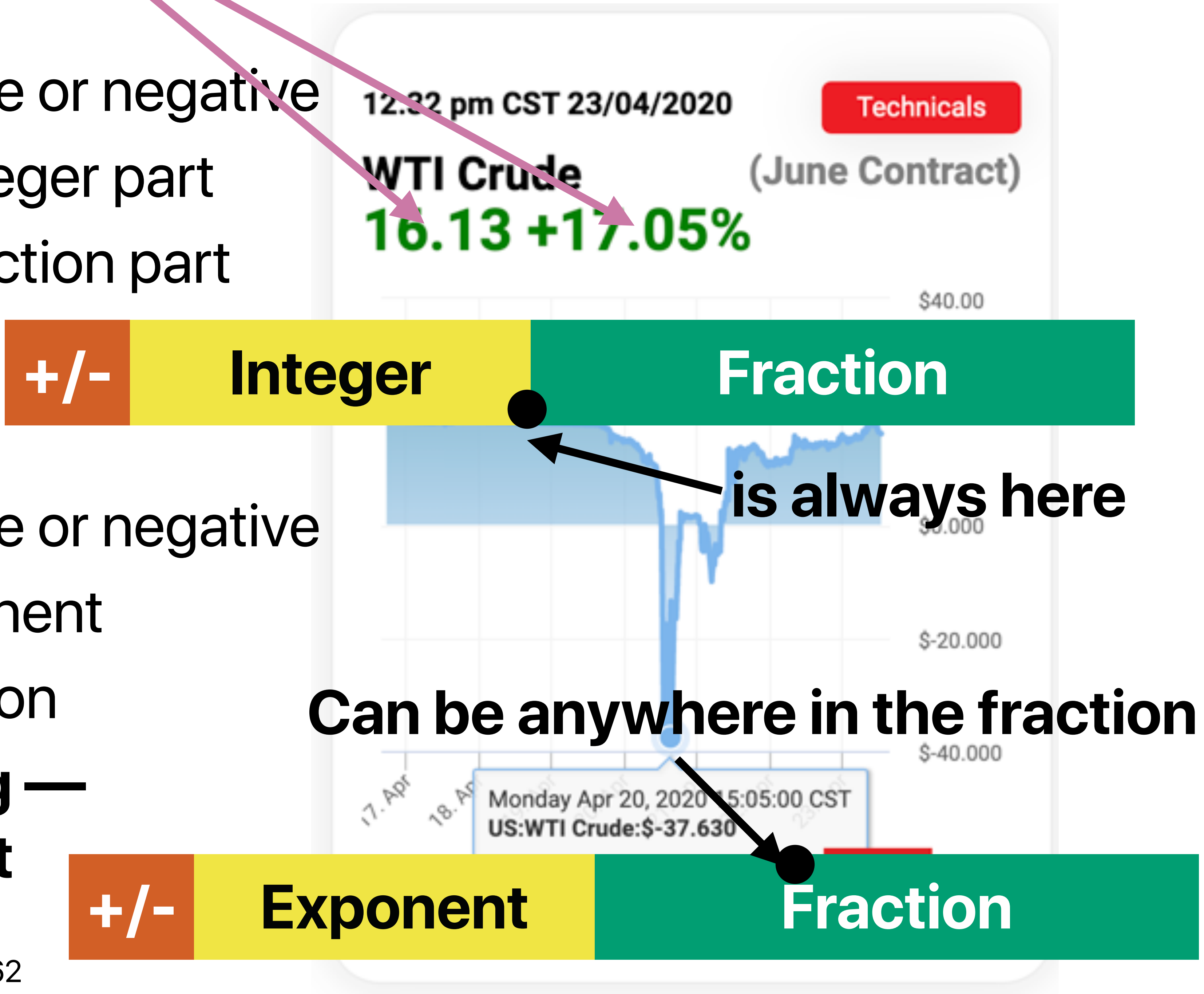
CLA v.s. Carry-ripple

- Size:
  - 32-bit CLA with 4-bit CLAs — requires 8 of 4-bit CLA
    - Each requires 116 for the CLA 4*(4*6+8) for the A+B — 244 gates
    - 1952 transistors
  - 32-bit CRA
    - 1600 transistors **Win!**
- Delay
  - 32-bit CLA with 8 4-bit CLAs
    - 2 gates * 8 = 16 **Win!**
  - 32-bit CRA
    - 64 gates

Area-Delay Trade-off!

Integer is not the only type of number we need to deal with!

- There is number with a **decimal point**
- Fixed point
  - One bit is used for representing positive or negative
  - Fixed number of bits is used for the integer part
  - Fixed number of bits is used for the fraction part
  - Therefore, the decimal point is **fixed**
- Floating point
  - One bit is used for representing positive or negative
  - A fixed number of bits is used for exponent
  - A fixed number of bits is used for fraction
  - Therefore, the decimal point is **floating — depending on the value of exponent**



is always here

Can be anywhere in the fraction

IEEE 754 format

32-bit float	+/-	Exponent (8-bit)	Fraction (23-bit)

- Realign the number into 1.**F** * 2$^{\mathbf{e}}$
- Exponent stores **e** + 127
- Fraction only stores **F**
- Convert the following number
  1 1000 0010 0100 0000 0000 0000 0000 000

A. - 1.010 * 2^130

[✓ B. -10]

C. 10

D. 1.010 * 2^130

E. None of the above

1	1000 0010	0100 0000 0000 0000 0000 000
-	**e = 130** **-127 = 3**	**1.f = 1.01 = 1 + 0*2^{-1} + 1* 2^{-2} = 1.25**

1.25 * 2^3 = 10

The advantage of floating/fixed point

- Regarding the pros of floating point and fixed point expressions, please identify the correct statement

  A. Fixed point can be express wider range of numbers than floating point numbers, but the hardware design is more complex

  [✓ B.] Floating point can be express wider range of numbers than floating point numbers, but the hardware design is more complex

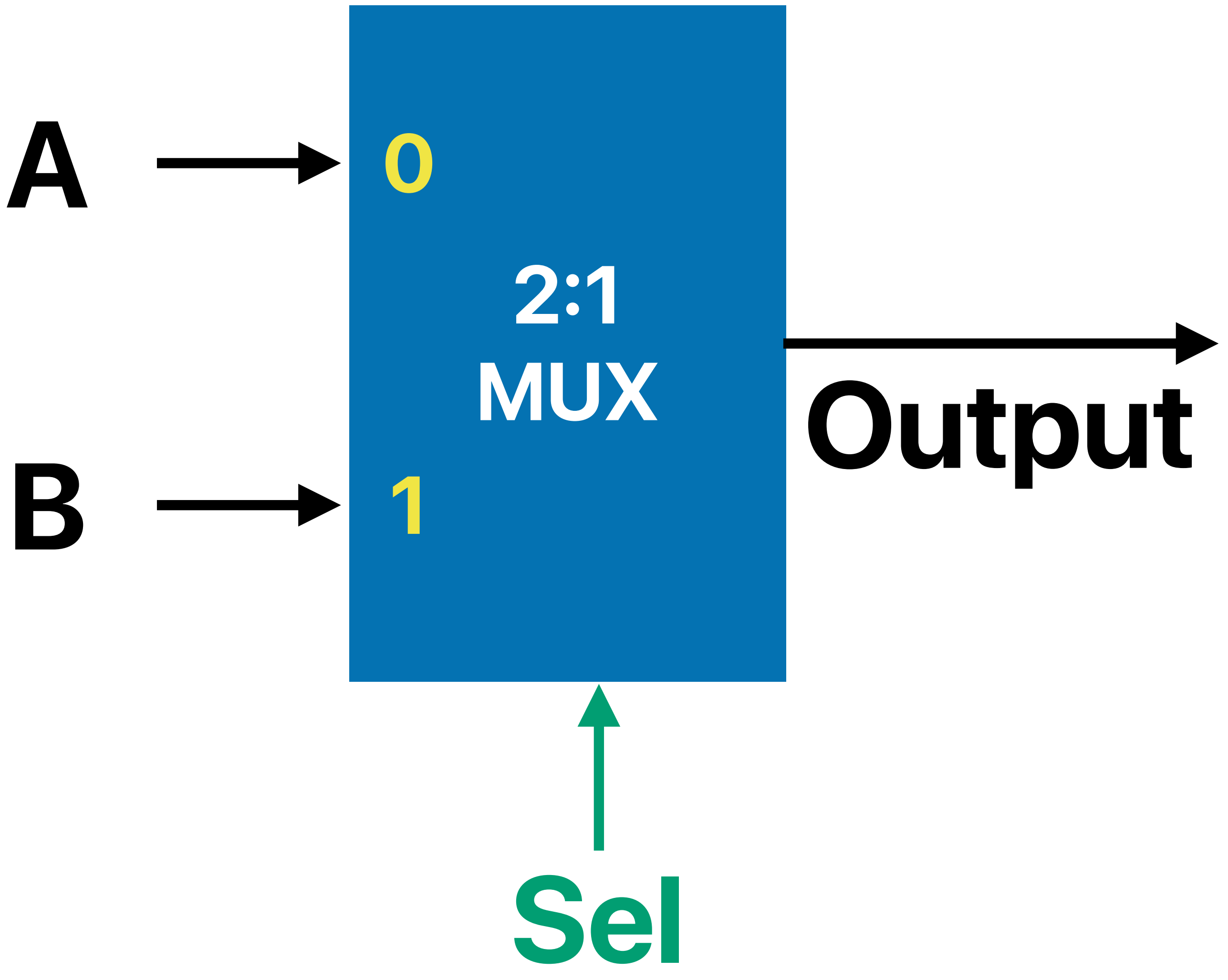
  C. Fixed point can be express wider range of numbers than floating point numbers, and the hardware design is simpler

  D. Floating point can be express wider range of numbers than floating point numbers, and the hardware design is simpler

Multiplexer

Let's start with a 2-to-1 MUX

- The MUX has two input ports — numbered as 0 and 1
- To select from two inputs, you need a 1-bit control/select signal to indicate the desired input port



Input			Output
A	B	Sel	
0	0	0	**0**
0	1	0	**0**
1	0	0	**1**
1	1	0	**1**
0	0	1	**0**
0	1	1	**1**
1	0	1	**0**
1	1	1	**1**

Use K-Map

Input			Output
A	B	Sel	
0	0	0	**0**
0	1	0	**0**
1	0	0	**1**
1	1	0	**1**
0	0	1	**0**
0	1	1	**1**
1	0	1	**0**
1	1	1	**1**

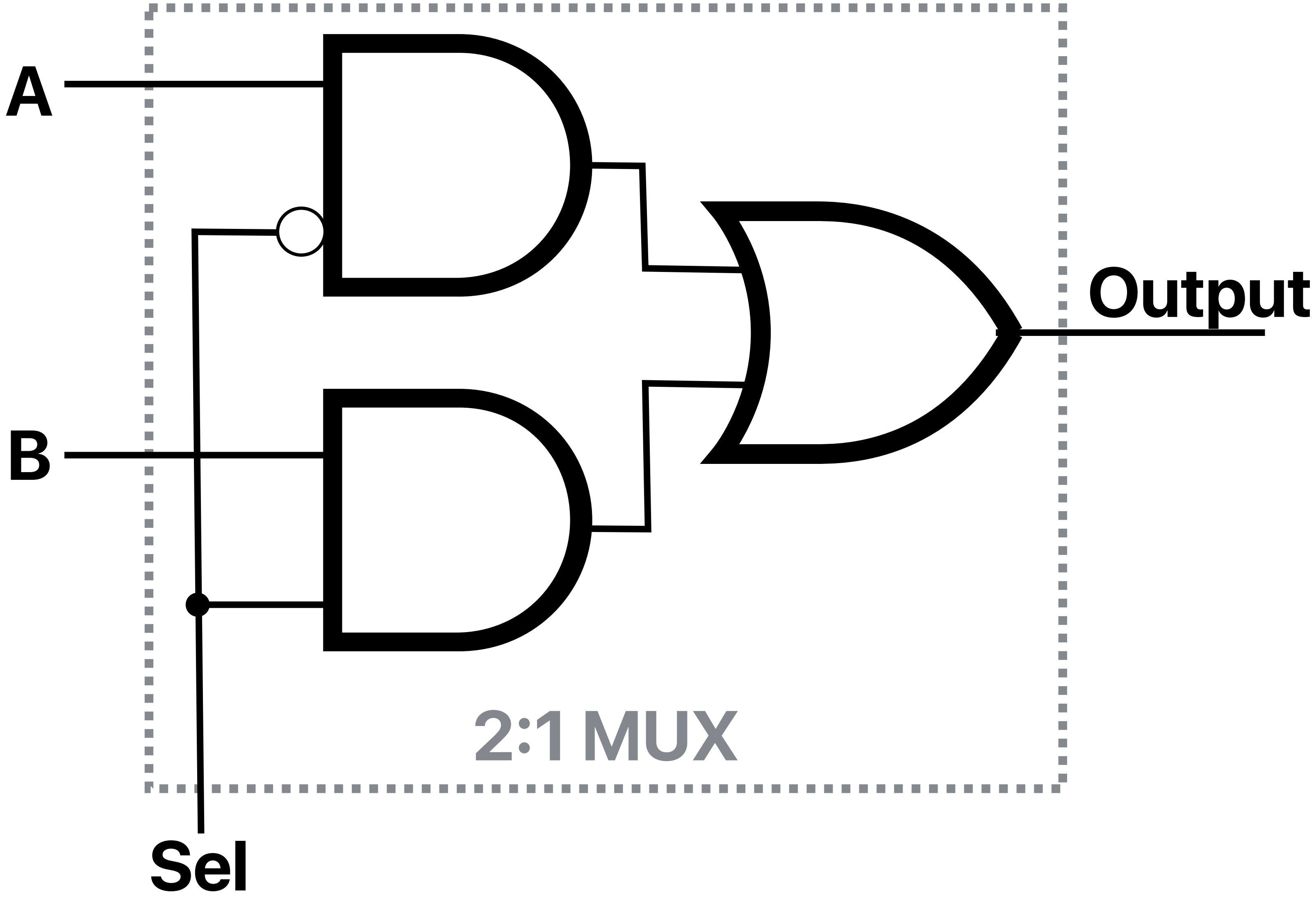
Sel′ means output A
Sel means output B

Output = ASel′ + BSel

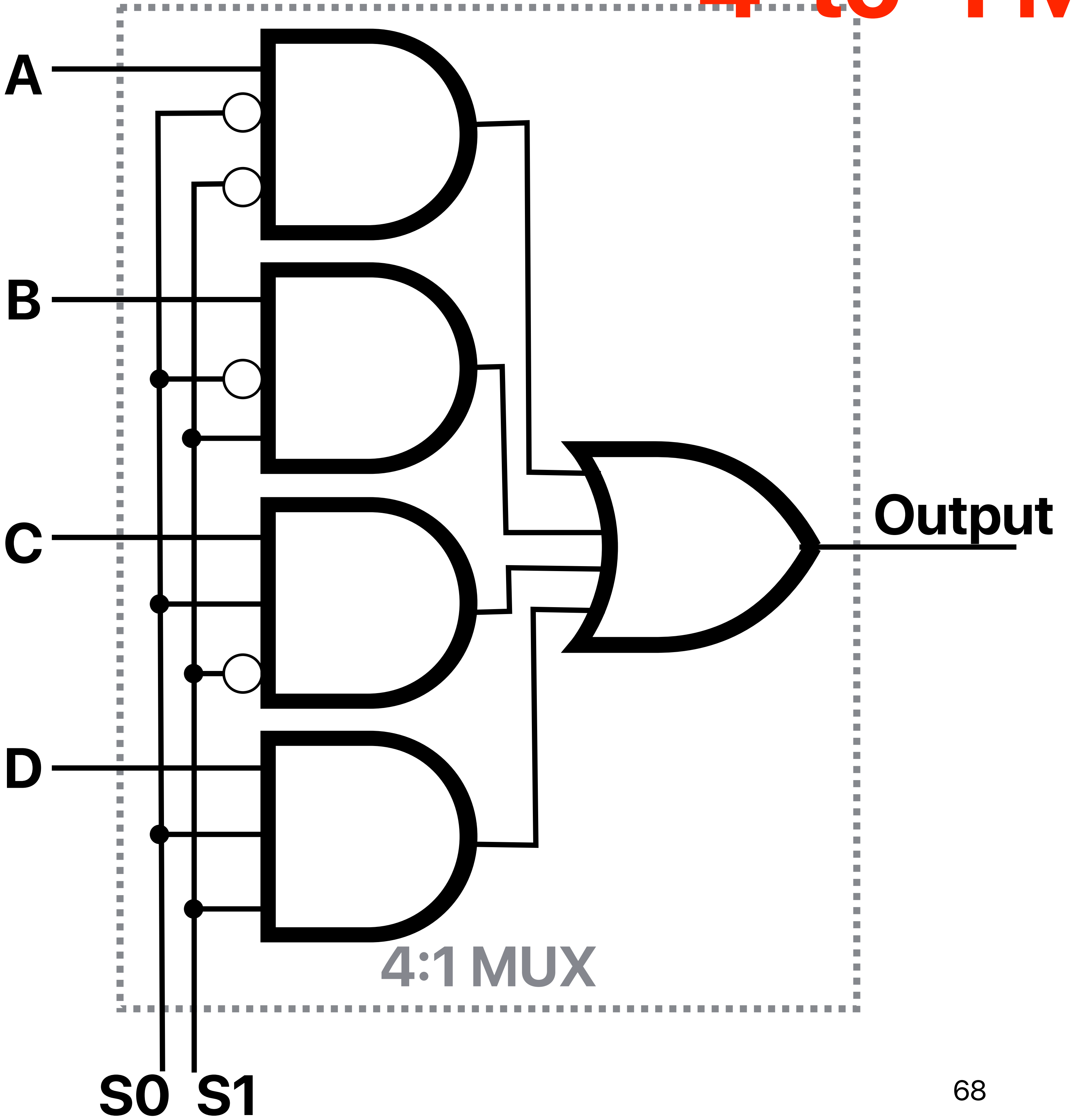
(A, B) Sel	A′B′ 0,0	A′B 0,1	AB 1,1	AB′ 1,0
Sel′ 0	0	0	1	1
Sel 1	0	1	1	0

ASel′

BSel

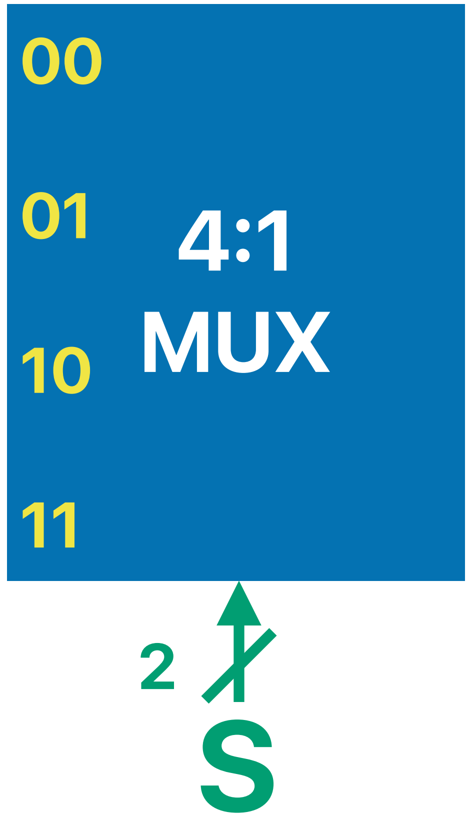


4-to-1 MUX



S0==0 && S1==0 output A
S0==0 && S1==1 output B
S0==1 && S1==0 output C
S0==1 && S1==1 output D

Output = AS0′S1′ + BS0′S1 + CS0S1′ + DS0S1



Gate delay of 8:1 MUX

- What’s the estimated gate delay of an 8:1 MUX?

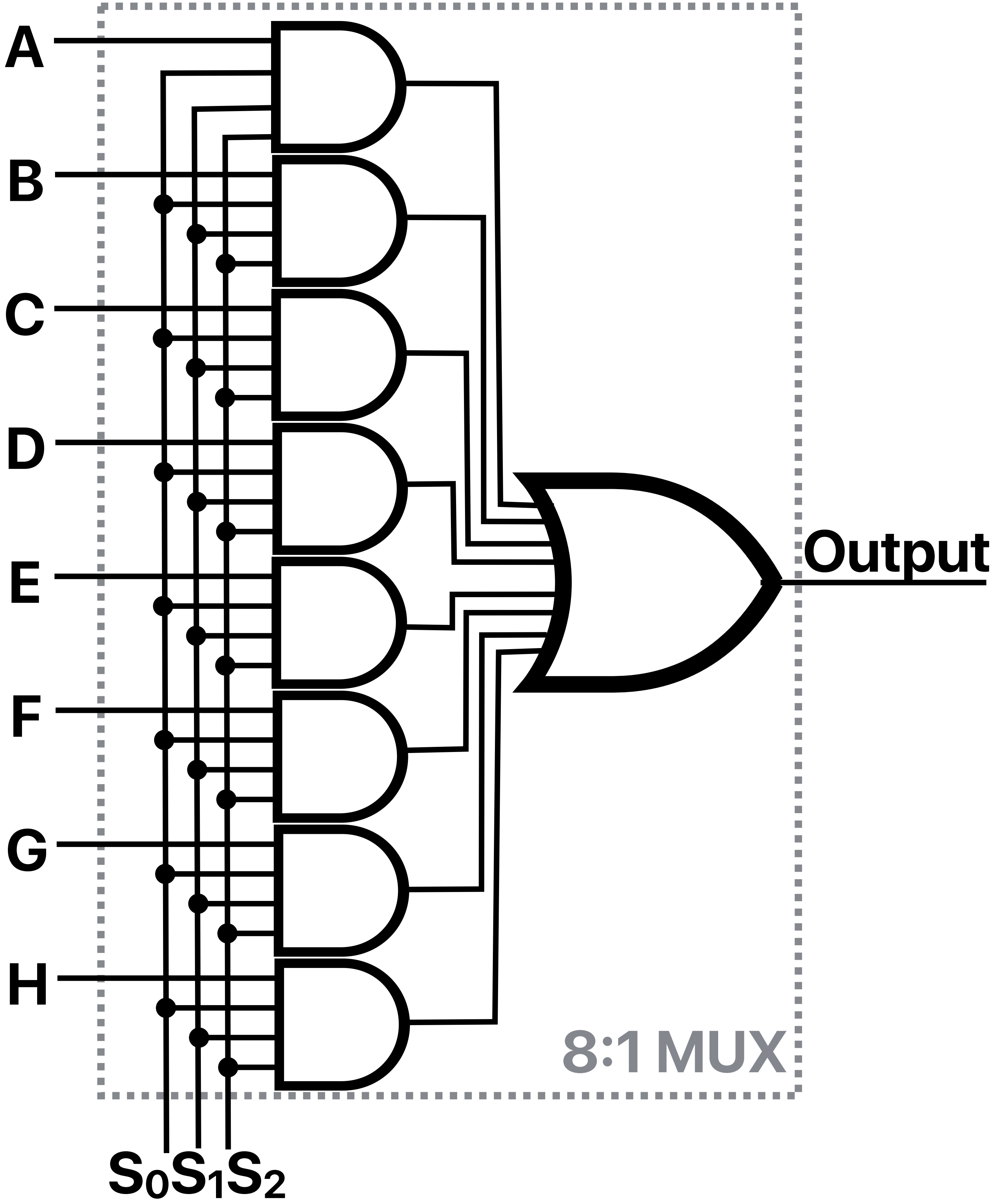
  A. 1

  [✓ B. 2]

  C. 4

  D. 8

  E. 16



Shifters

What's after shift?

- Assume we have a data type that stores 8-bit unsigned integer (e.g., unsigned char in C). How many of the following C statements and their execution results are correct?

Statement		C = ?
I	`c = 3; c = c >> 2;`	**0**
II	`c = 255; c = c << 2;`	252
III	`c = 256; c = c >> 2;`	**0**
IV	`c = 128; c = c << 1;`	**0**

A. 0

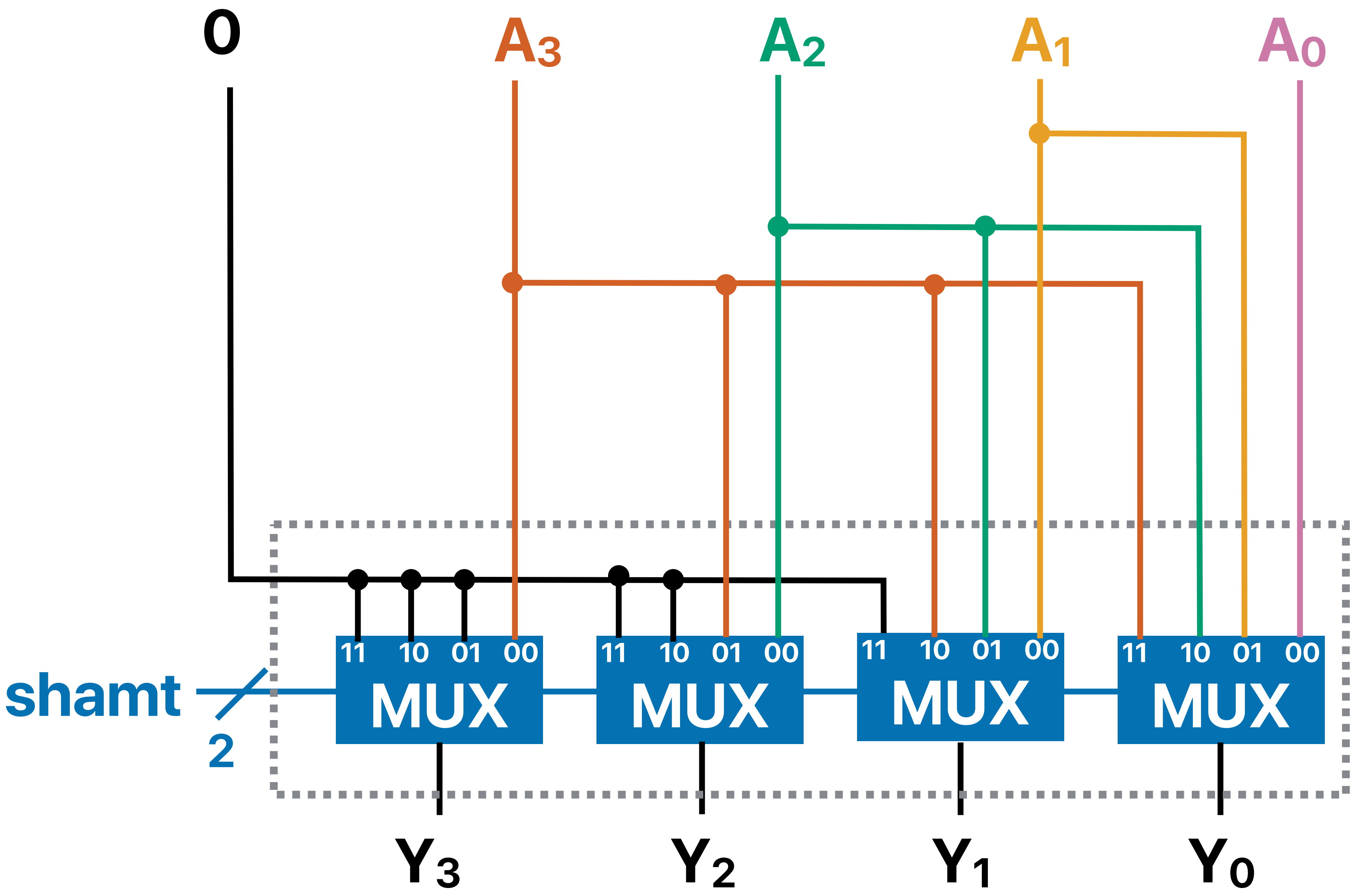
[✓ B. 1]

C. 2

D. 3

E. 4

Shift "Right"



Based on the value of the selection input (shamt = shift amount)

Example:
if S = 11
then
Y3 = 0
Y2 = 0
Y1 = 0
Y0 = A3

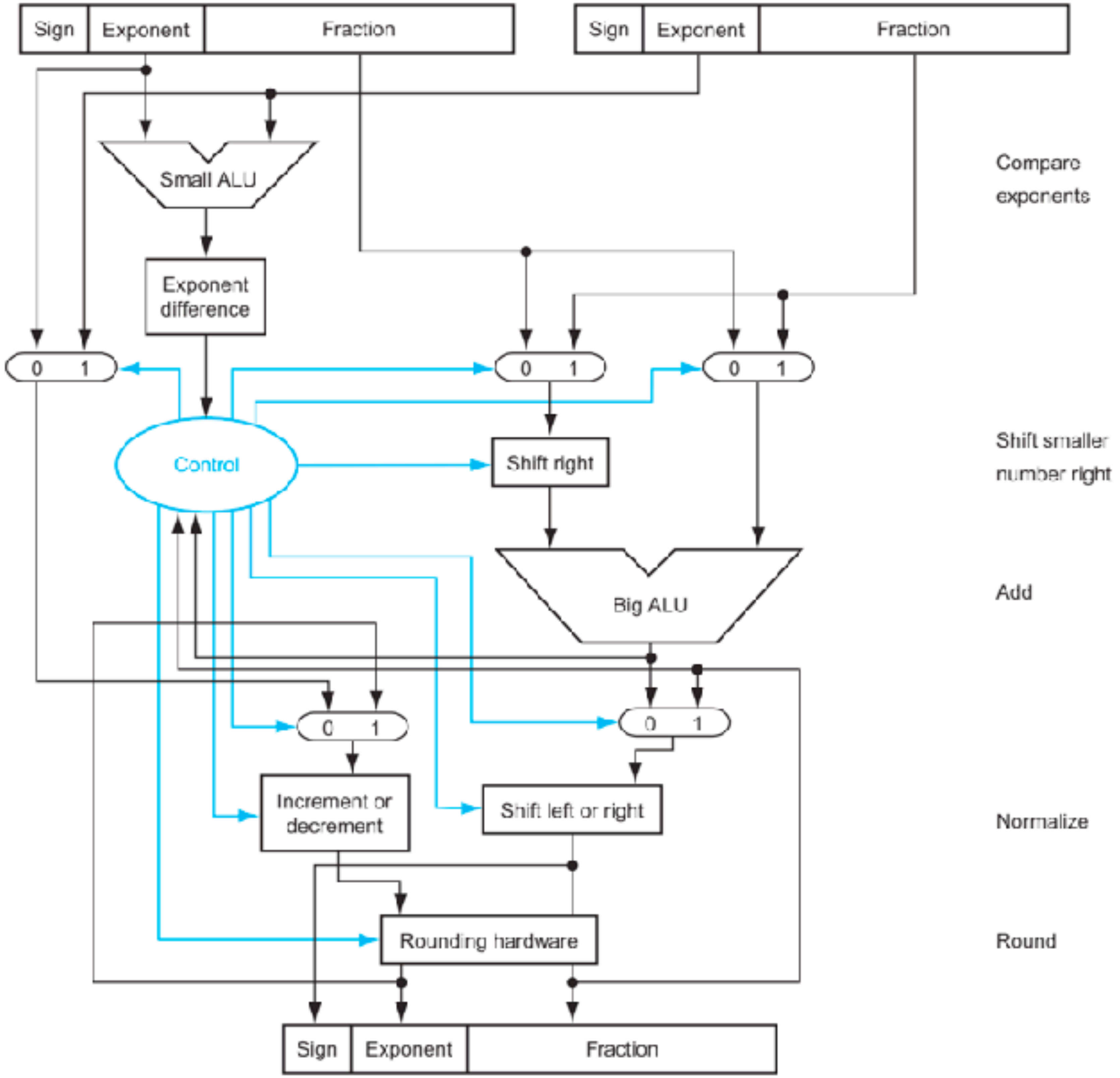
Example:
if S = 10
then
Y3 = 0
Y2 = 0
Y1 = A3
Y0 = A2

Example:
if S = 01
then
Y3 = 0
Y2 = A3
Y1 = A2
Y0 = A1

The "chain" of multiplexers determines how many bits to shift

Floating point hardware

Floating point adder



Why — Will the loop end?

- Consider the following two C programs.

X

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int i=0;
    while(i >= 0) i++;
    printf("We're done! %d\n", i);
    return 0;
}
```

Y

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float i=0.0;
    while(i >= 0) i++;
    printf("We're done! %f\n",i);
    return 0;
}
```

Because Floating Point Hardware Handles “sign”, “exponent”, “mantissa” separately

Please identify the correct statement.

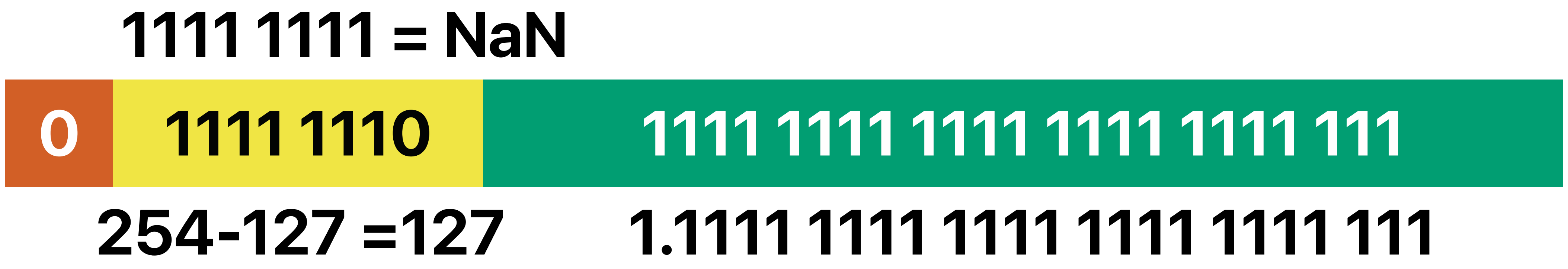
A. X will print “We’re done” and finish, but Y will not.

B. X won’t print “We’re done” and won’t finish, but Y will.

C. Both X and Y will print “We’re done” and finish

D. Neither X nor Y will finish

Maximum and minimum in float



1111 1111 = NaN

0

1111 1110

1111 1111 1111 1111 1111 111

254-127 =127

1.1111 1111 1111 1111 1111 111

= 340282346638528859811704183484516925440
= 3.40282346639e+38

max in int32 is 2^31-1 = 2147483647

But, this also means that float cannot express all possible numbers between its max/min — lose of precisions

Special numbers in IEEE 754 float

+0	0	0000 0000	0000 0000 0000 0000 0000 000
-0	1	0000 0000	0000 0000 0000 0000 0000 000
+Inf	0	1111 1111	0000 0000 0000 0000 0000 000
-Inf	1	1111 1111	0000 0000 0000 0000 0000 000
+NaN	0	1111 1111	xxxx xxxx xxxx xxxx xxxx xxx
-Nan	1	1111 1111	xxxx xxxx xxxx xxxx xxxx xxx

What’s 0.0004 in IEEE 754?

0	0	1	1	1	0	0	1	1	1	0	1	0	0	0	1	1	0	1	1	0	1	1	1	0	0	0	1	1	1	0	1

	after x2	> 1?
0.0004	0.0008	0
0.0008	0.0016	0
0.0016	0.0032	0
0.0032	0.0064	0
0.0064	0.0128	0
0.0128	0.0256	0
0.0256	0.0512	0
0.0512	0.1024	0
0.1024	0.2048	0
0.2048	0.4096	0
0.4096	0.8192	0
0.8192	1.6384	1
0.6384	1.2768	1
0.2768	0.5536	0
0.5536	1.1072	1
0.1072	0.2144	0
0.2144	0.4288	0
0.4288	0.8576	0
0.8576	1.7152	1
0.7152	1.4304	1

12

	after x2	> 1?
0.4304	0.8608	0
0.8608	1.7216	1
0.7216	1.4432	1
0.4432	0.8864	0
0.8864	1.7728	1
0.7728	1.5456	1
0.5456	1.0912	1
0.0912	0.1824	0
0.1824	0.3648	0
0.3648	0.7296	0
0.7296	1.4592	1
0.4592	0.9184	0
0.9184	1.8368	1
0.8368	1.6736	1
0.6736	1.3472	1
0.3472	0.6944	0
0.6944	1.3888	1
0.3888	0.7776	0
0.7776	1.5552	1
0.5552	1.1104	1

-12 + 127 = 115 = 0b01110011

Demo — Are we getting the same numbers?

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float a, b, c;
    a = 1280.245;
    b = 0.0004;
    c = (a + b)*10.0;
    printf("(1280.245 + 0.0004)*10 = %f\n",c);
    c = a*10.0 + b*10.0;
    printf("1280.245*10 + 0.0004*10 = %f\n",c);
    return 0;
}
```

Commutative law is broken!!!

Are we getting the same numbers?

- For the following code, please identify how many statements are correct

  ① We will see the same output at X and Y

  ② X will print — 12802.454

  ③ Y will print — 12802.454

  ④ Neither X nor Y will print the right result, but X is closer to the right answer

  ⑤ Neither X nor Y will print the right result, but Y is closer to the right answer

  A. 0

  [✓ B. 1]

  C. 2

  D. 3

  E. 4

```
#include <stdio.h>

int main(int argc, char **argv) {
    float a, b, c;
    a = 1280.245;
    b = 0.0004;
    c = (a + b)*10.0;
    printf("%f\n”,c); // X
    c = a*10.0 + b*10.0;
    printf("%f\n”,c); // Y
    return 0;
}
```

Recap: Will the loop end? (one more run)

- Consider the following C program.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float i=1.0;
    while(i > 0) i++;
    printf("We're done! %f\n",i);
    return 0;
}
```

Please identify the correct statement.

A. The program will finish since i will end up to be +0

B. The program will finish since i will end up to be -0

C. The program will finish since i will end up to be something < 0

[✓ D. The program will not finish since i will always be a positive non-zero number.]

E. The program will not finish but raise an exception since we will go to NaN first.

Why stuck at 16777216?

- Consider the following C program.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    float i=1.0;
    while(i > 0) i++;
    printf("We're done! %f\n",i);
    return 0;
}
```

Why i stuck at 16777216.000?

A. It’s a special number in IEEE 754 standard that an adder will treat it differently

B. It’s a special number like +Inf/-Inf or +NaN/-NaN with special meaning in the IEEE 754 standard

C. It’s just the maximum integer that IEEE 754 standard can represent

[✓ D. It’s nothing special, but just happened to be the case that 16777216.0+1.0 will produce 16777216.0]

E. It’s nothing special, but just happened to be the case that 16777216.0 add anything will become 16777216.0

What's 16777216 special about?

0 | 10010111 | 0000 0000 0000 0000 0000 000

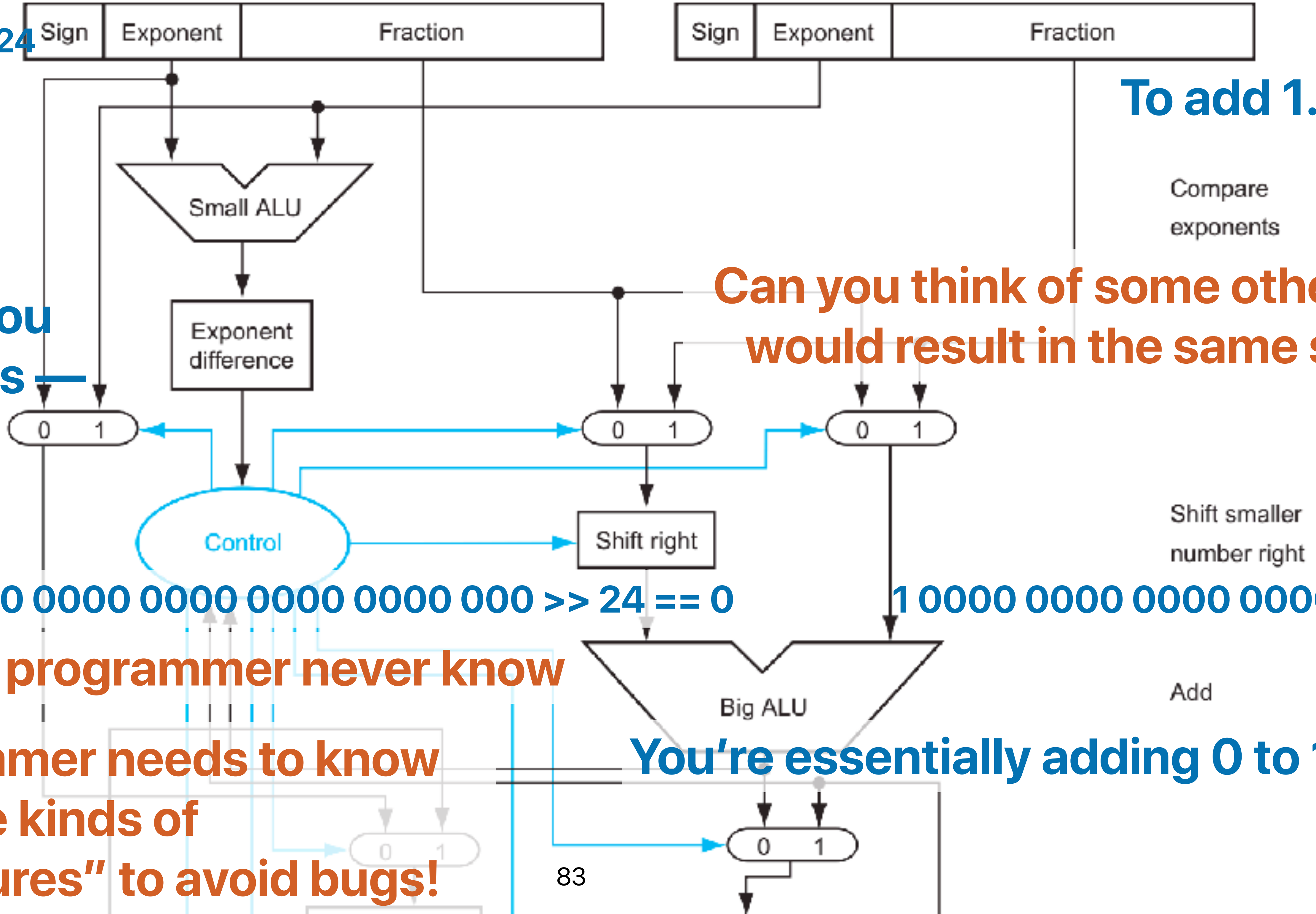
0 | 0111 1111 | 0000 0000 0000 0000 0000 000

$16777216 = 1.0 * 2^{24}$

To add $1.0 = 1.0 * 2^0$

to this number, you have to shift 24 bits —

Can you think of some other numbers would result in the same situation?



1 0000 0000 0000 0000 0000 000 >> 24 == 0

1 0000 0000 0000 0000 0000 000

— even worse — programmer never know

You're essentially adding 0 to 16777216

A good programmer needs to know these kinds of "hardware features" to avoid bugs!

Sequential Circuits

Recap: Combinational v.s. sequential logic

- Combinational logic
  - The output is a pure function of its current inputs
  - The output doesn't change regardless how many times the logic is triggered — Idempotent
- Sequential logic
  - The output depends on current inputs, previous inputs, their history

Sequential circuit has memory!

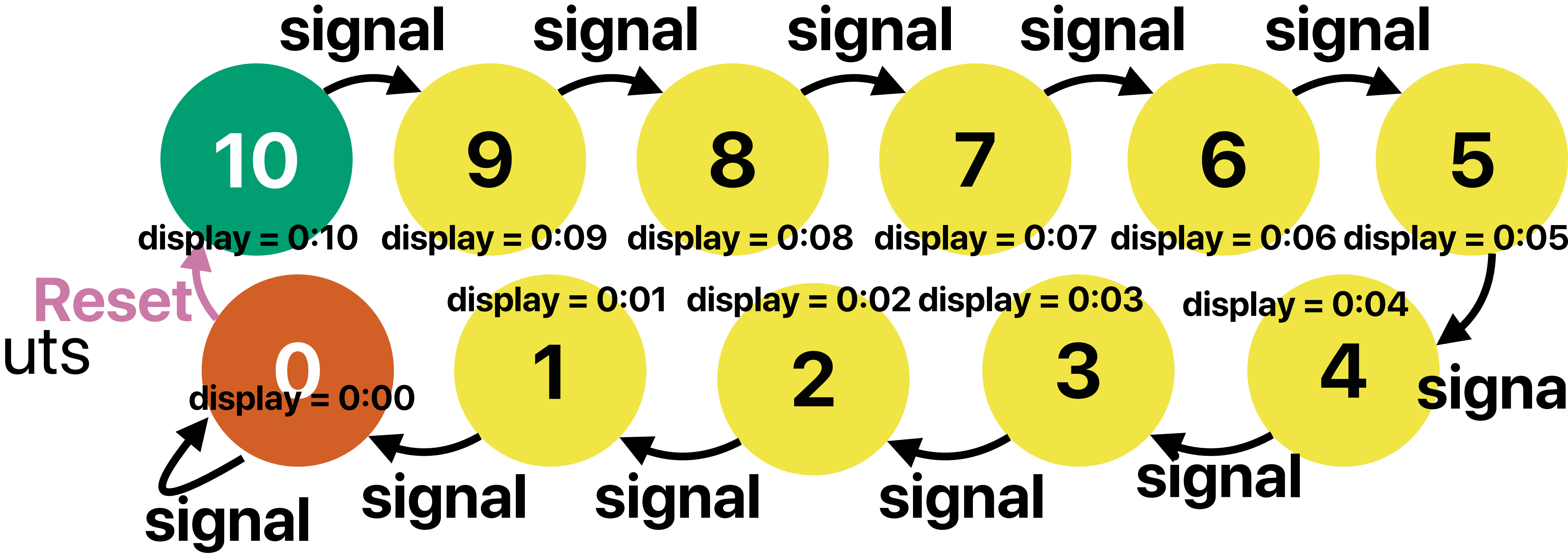
Recap: Theory behind each

- A **Combinational logic** is the implementation of a **Boolean Algebra** function with only Boolean Variables as their inputs
- A **Sequential logic** is the implementation of a **Finite-State Machine**

Finite-State Machines

Finite State Machines

- FSM consists of
  - Set of states
  - Set of inputs, set of outputs
  - Initial state
  - Set of transitions
    - Only one can be true at a time
- FSM representations:
  - State diagram
  - State table



Current State	Next State Signal	
	0	1
10	10	9
9	9	8
8	8	7
7	7	6
6	6	5
5	5	4
4	4	3
3	3	2
2	2	1
1	1	0
0	0	0

Life on Mars

- Mars rover has a binary input x. When it receives the input sequence x(t-2, t) = 001 from its life detection sensors, it means that the it has detected life on Mars and the output y(t) = 1, otherwise y(t) = 0 (no life on Mars).
- This pattern recognizer should have

A. One state because it has one output

B. One state because it has one input

C. Two states because the input can be 0 or 1

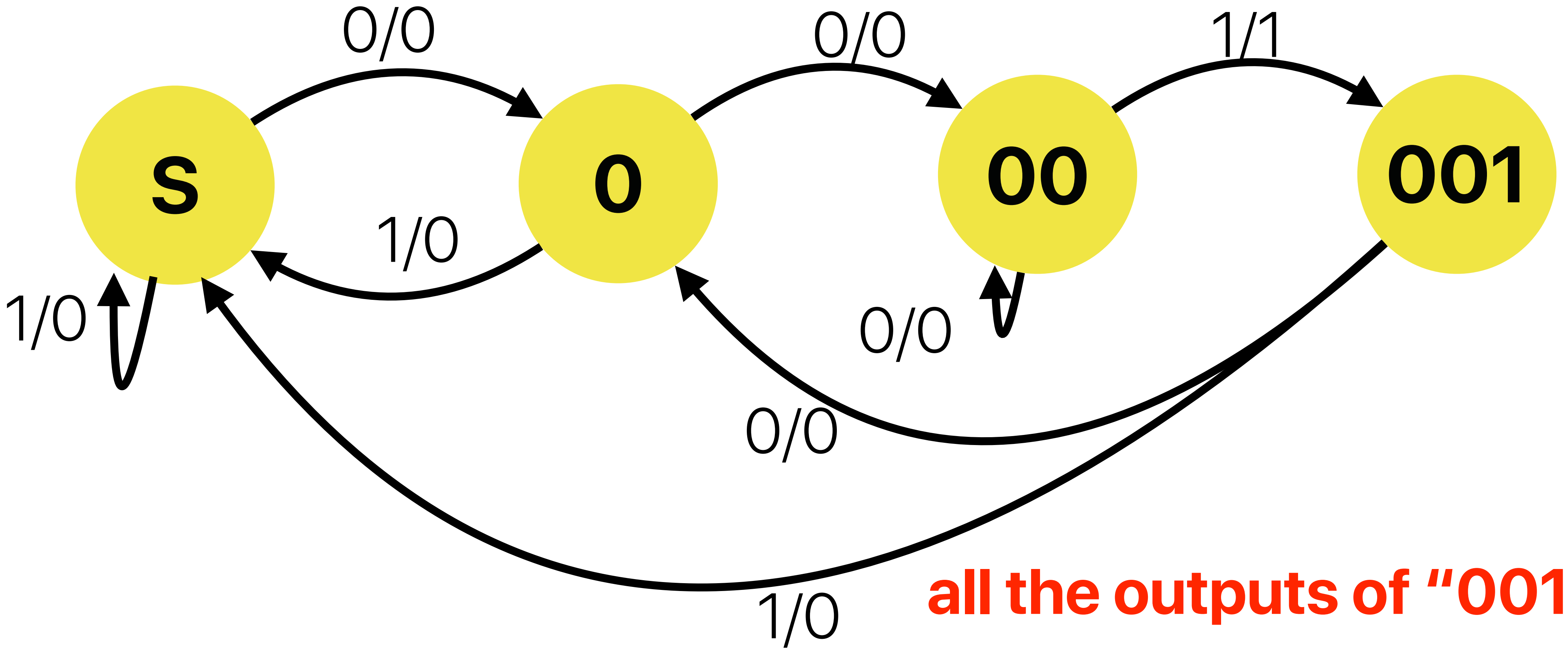
[✓ D. More than two states because ....]

E. None of the above

“Procedure” for creating an FSM

- Reduce the problem to a “sequence recognizer”
- To recognize a sequence with length N, you need N+1 states by default
- Layout the states and connect states with arrows (or create a state transition table)
- Merge states with exactly the same transitions (same input lead to exactly the same output) together

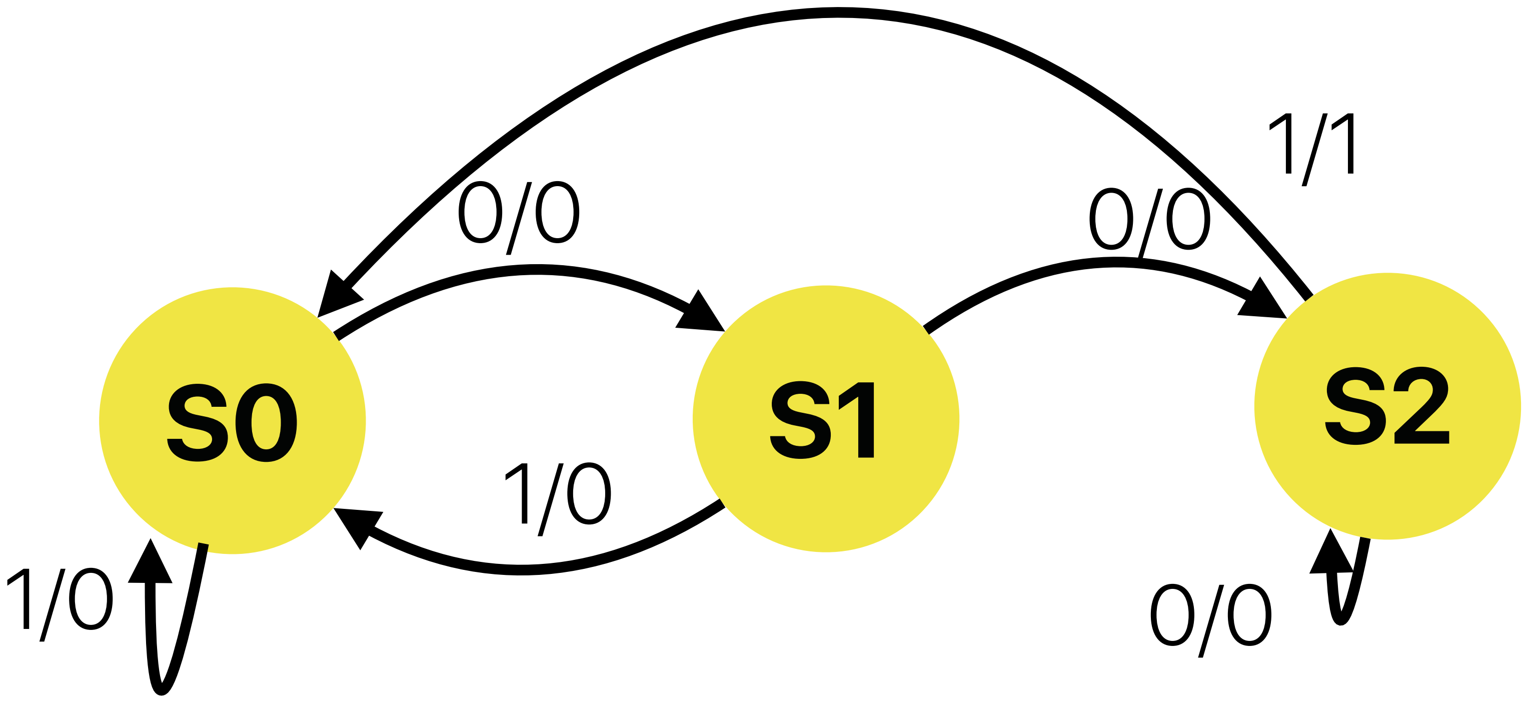
FSM for Life on Mars



all the outputs of “001" are equal to S!

Merge “001” into S

FSM for Life on Mars



Merge S3 into S0

State Transition Table of Life on Mars

Current State	Next State	
	Input	
	0	1
S0 — something else	S1, 0	S0, 0
S1 — 0	S2, 0	S0, 0
S2 — 00	S2, 0	S3, 1
S3 — 001	S1, 0	S0, 0

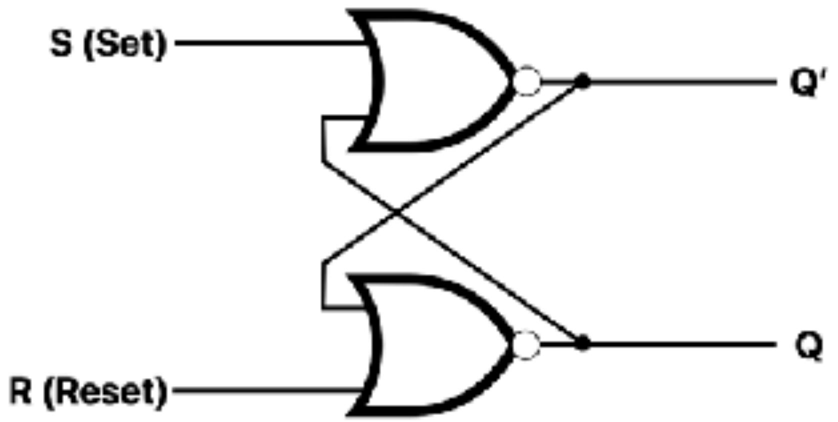
How make FSM true?

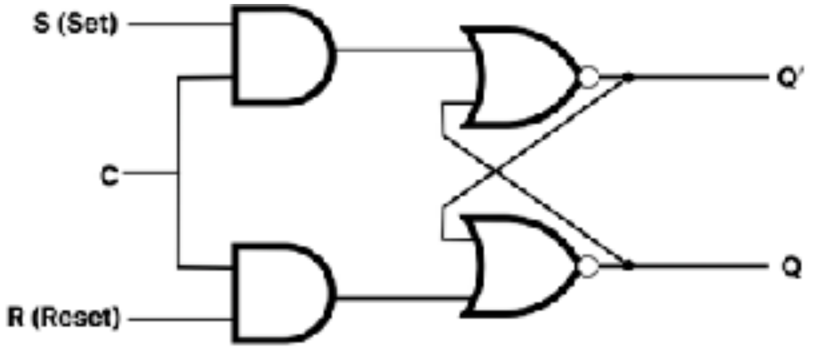
What do we need to physically implement the timer?

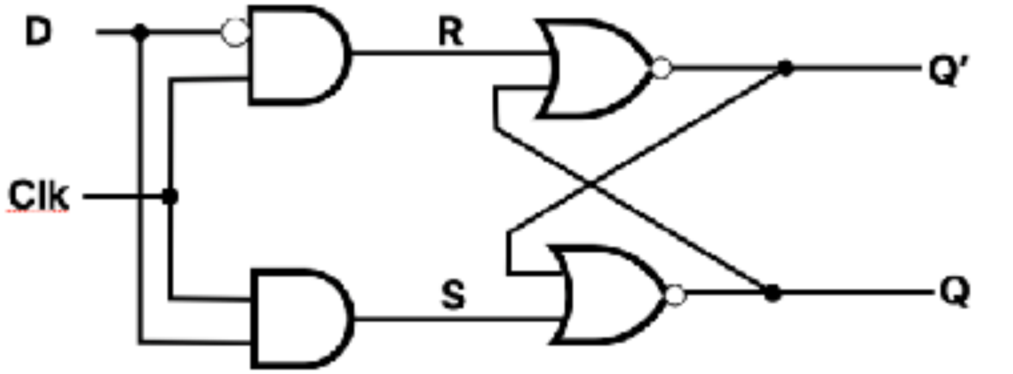
- A set of logic to display the remaining time — **we know how to do this already**
- A logic to keep track of the "current state" **— memory**
- A set of logic that uses the "current state" and "a new input" to transit to a new state and generate the output — **we also know how to build this**
- A control signal that helps us to transit to the right state at the right time **— clock**

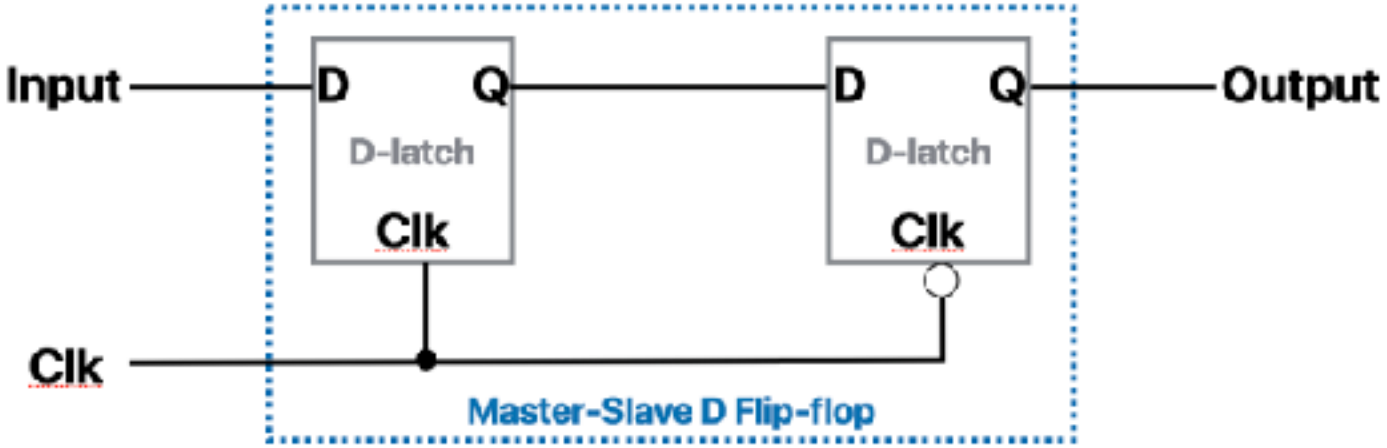
4-different types of bit storage

- SR-latch
  - S = 1 sets Q = 1
  - R = 1 sets Q = 0
  - Problem: S = 1, R = 1, Q = undefined
- Level-sensitive SR-latch
  - S, R only become effective when C = 1
  - Problem: avoid the case of signal oscillation, but cannot avoid the "intensional" 1,1 inputs
- D-latch
  - SR can never be 11 if the Clk is set appropriately
  - Problem: D single needs to be stably long enough to set the memory
- D-flip-flop
  - Only loads the value into memory in the beginning of the rising edge. Values can hold for a complete clock cycle
  - Problem: more gates

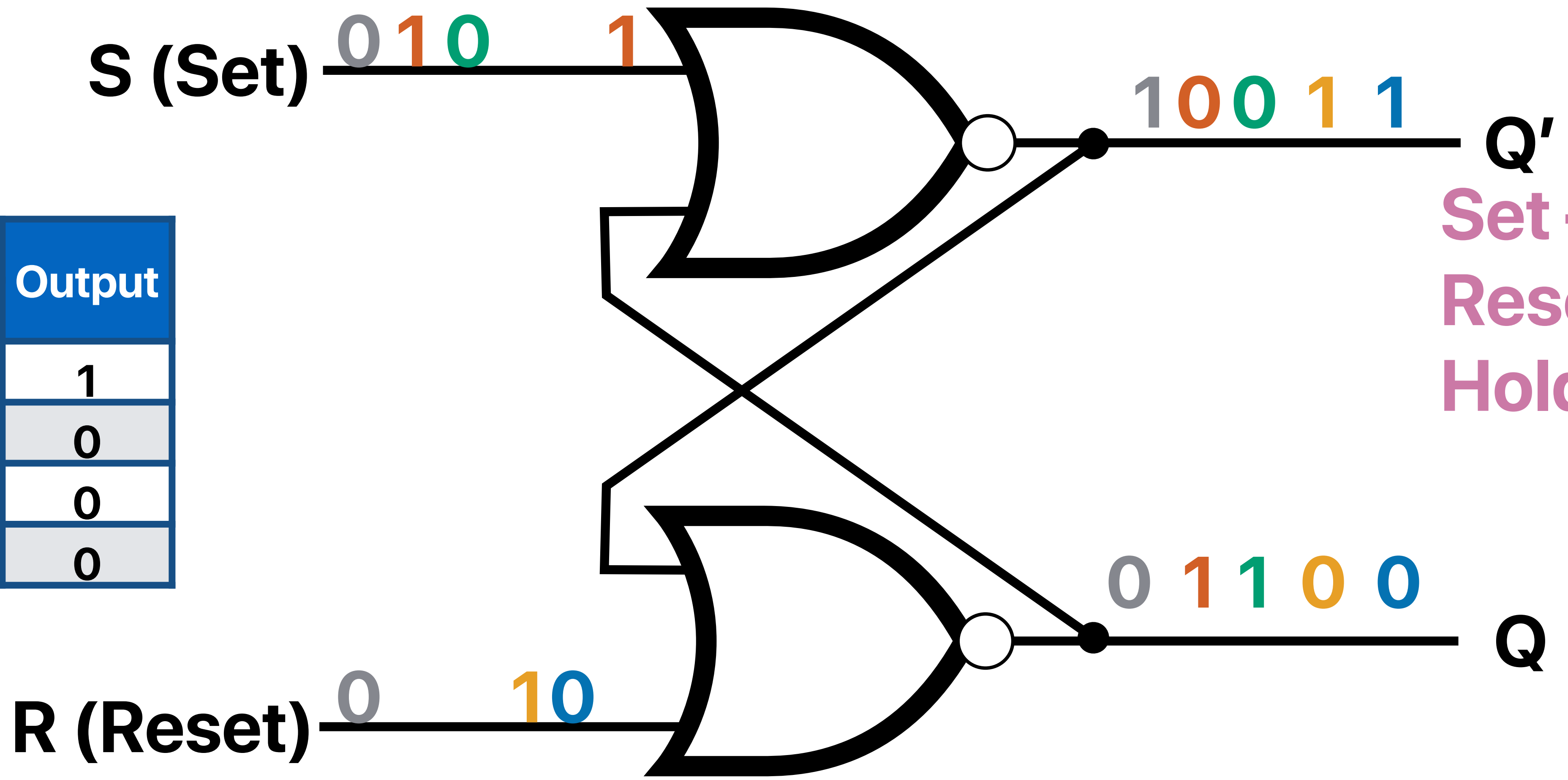








SR-Latch: the very basic “memory”



Input		Output
A	B	
0	0	1
0	1	0
1	0	0
1	1	0

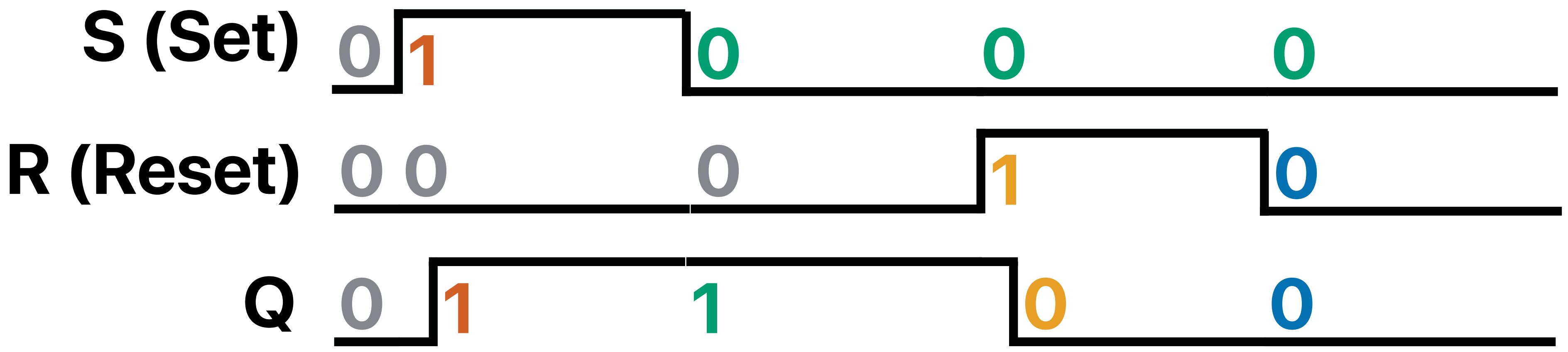
Set — Make the “stored bit 1”

Reset — Make the “stored bit 0”

Hold — both set/reset are 0

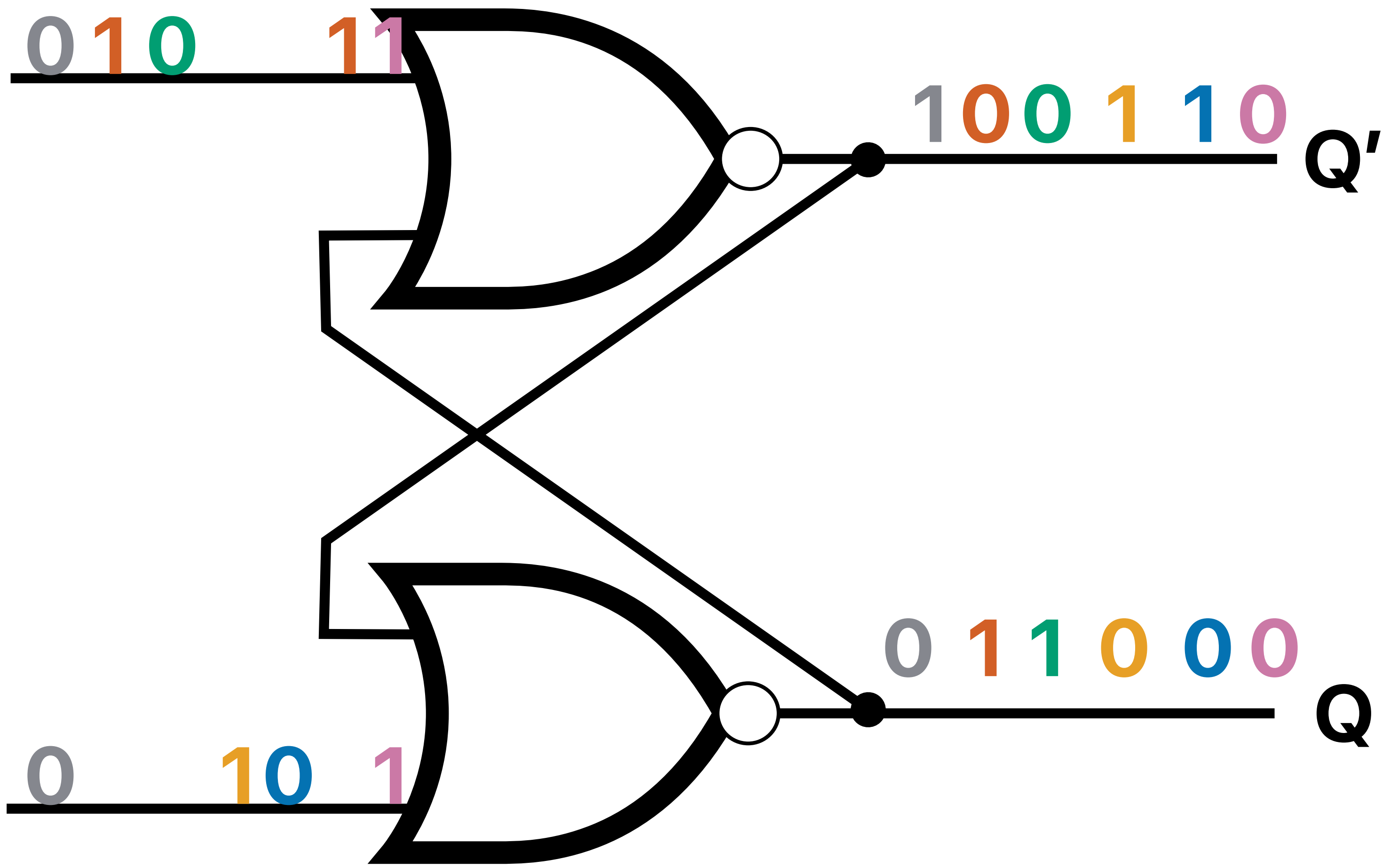
The circuit has memory!

S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	
1	1	1	



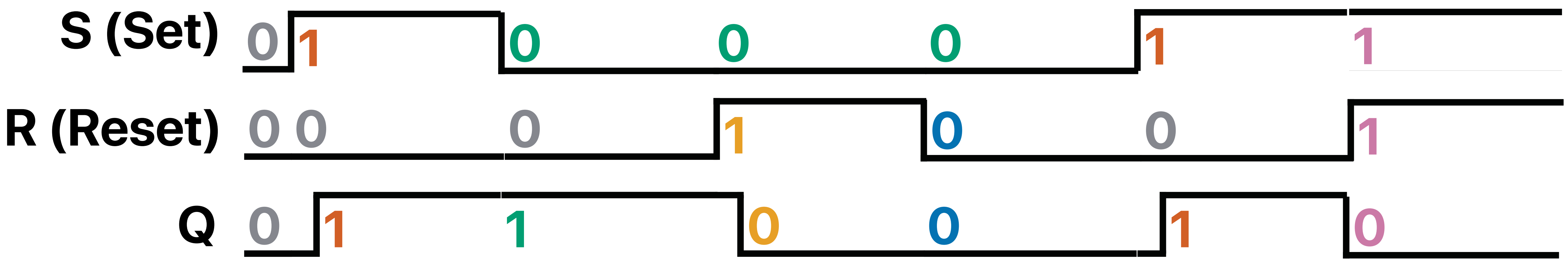
What if S/R are both 1s?

Input		Output
A	B	
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**



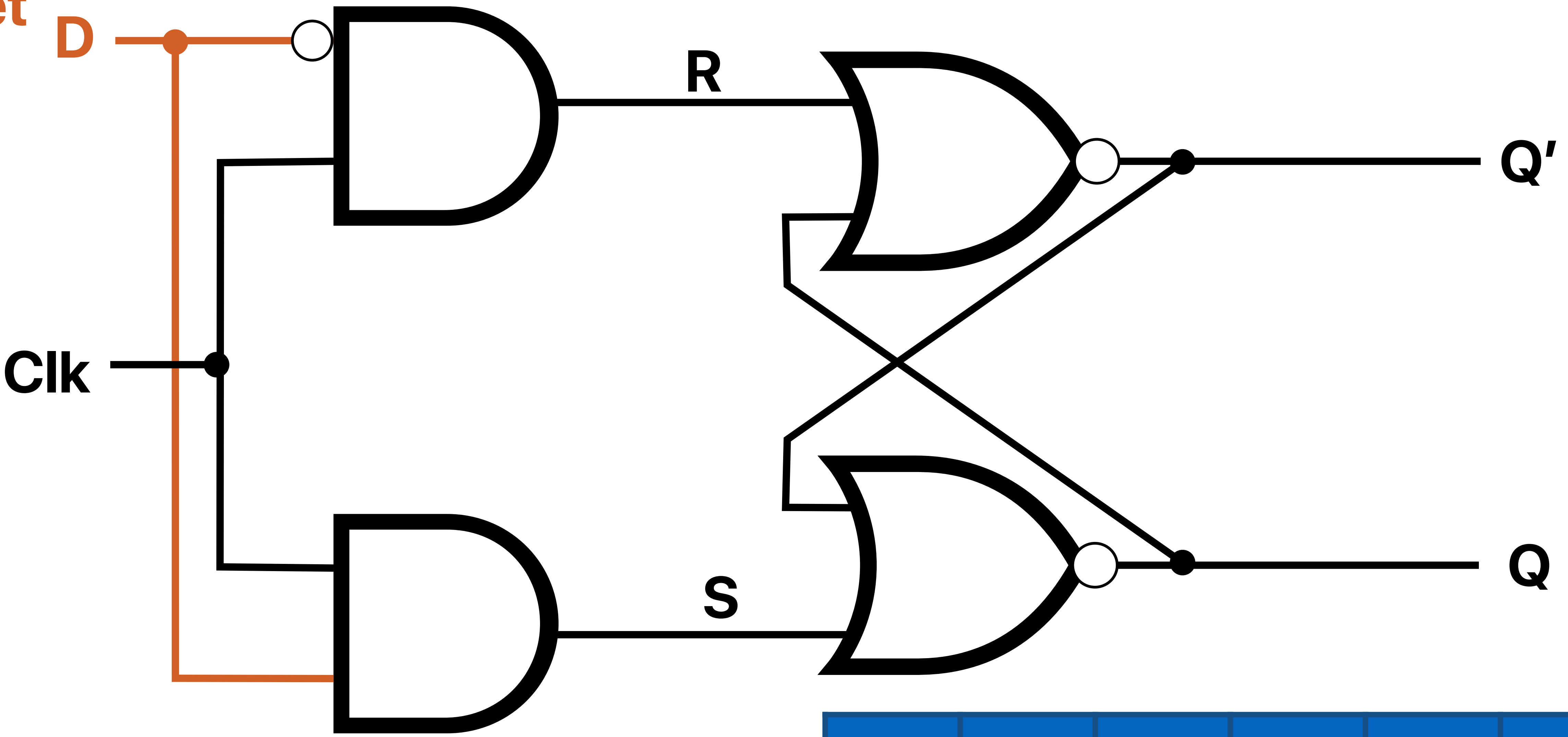
S	R	Q(t)	Q(t+1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Doesn’t function if both are 1s!



D-Latch

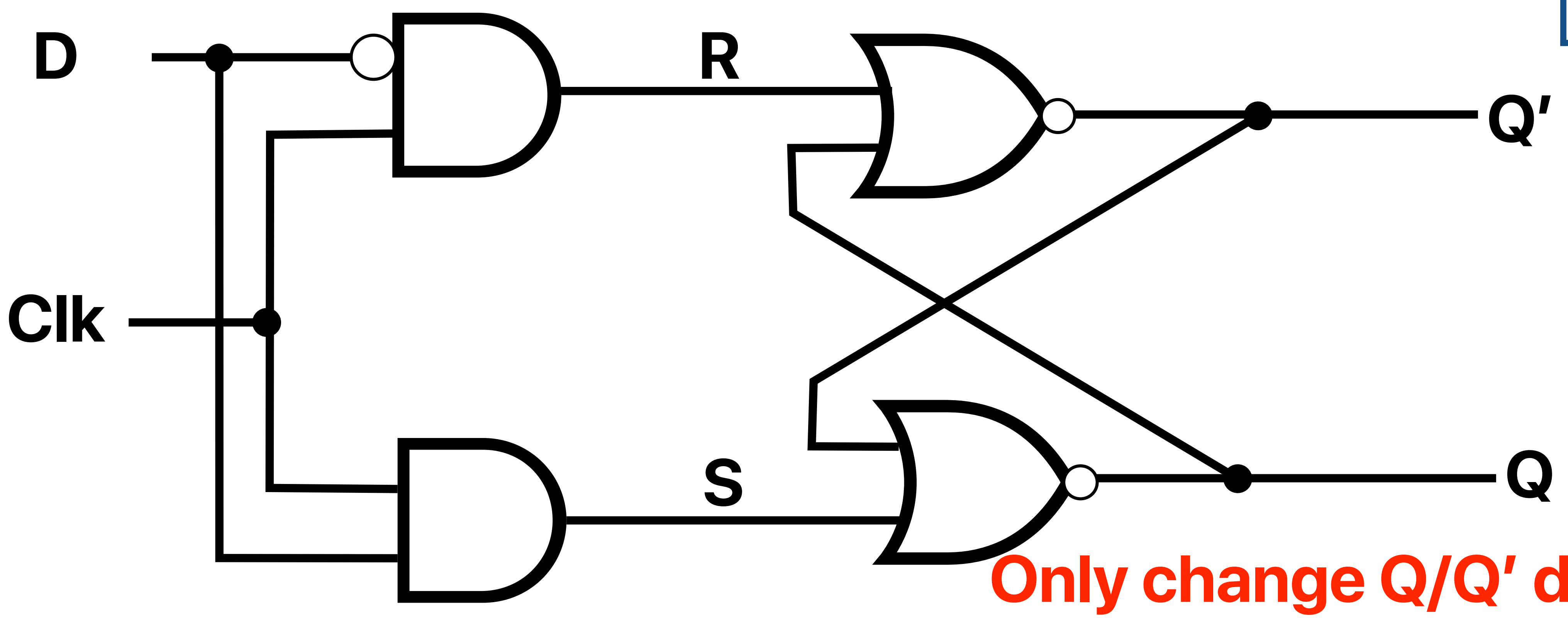
We will never get 1, 1 in this way



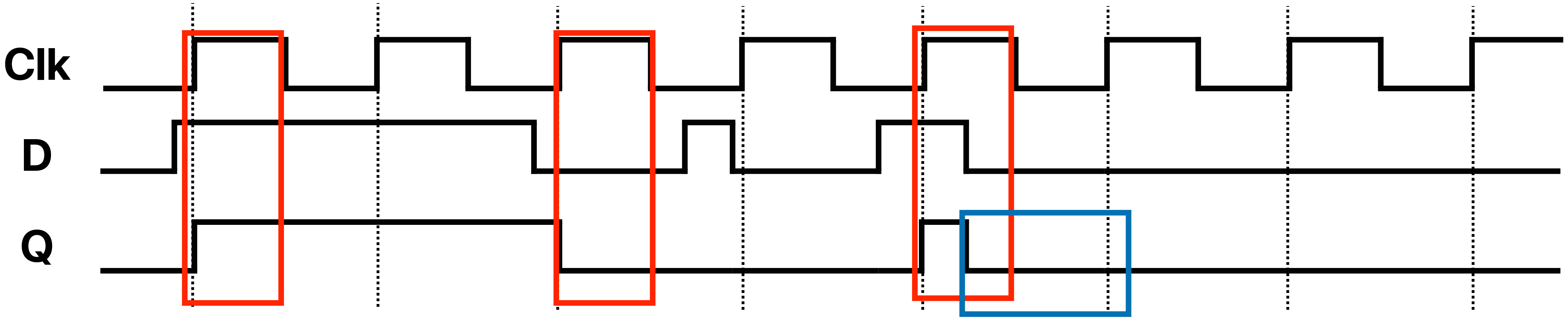
CLK	D	D′	S	R	Q	Q′
0	X	X′	0	0	Qprev	Qprev′
0	0	1	0	1	0	1
1	1	0	1	0	1	0

D-Latch

CLK	D	D′	S	R	Q	Q′
0	X	X′	0	0	Qprev	Qprev′
0	0	1	0	1	0	1
1	1	0	1	0	1	0

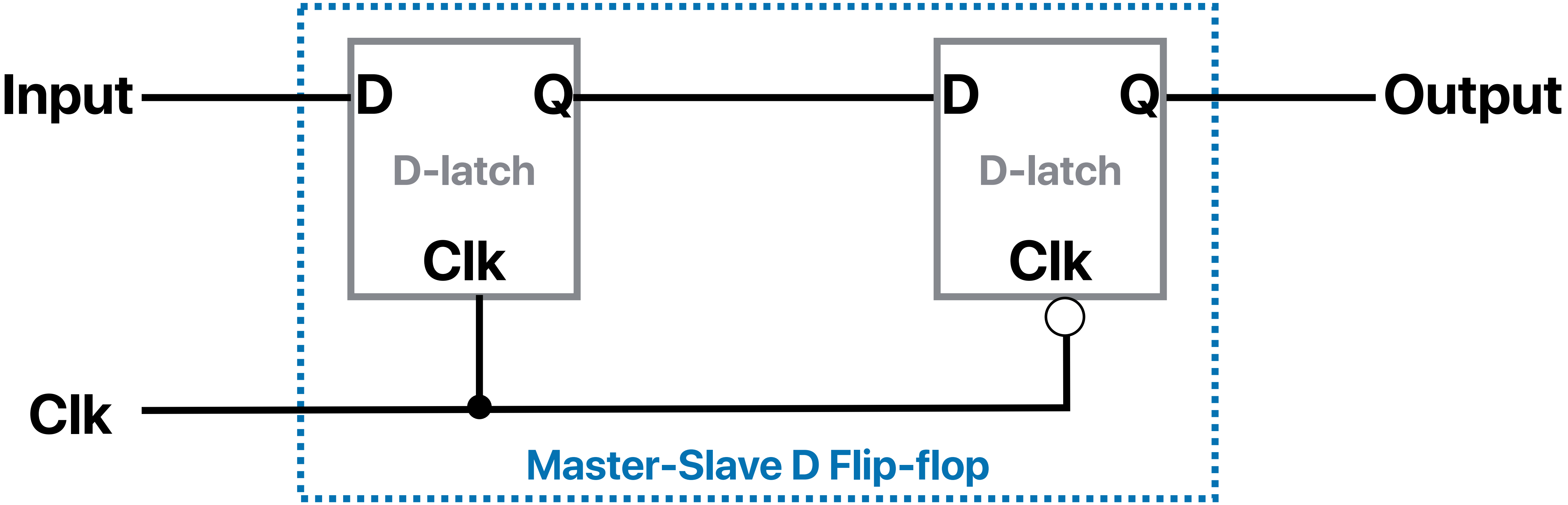


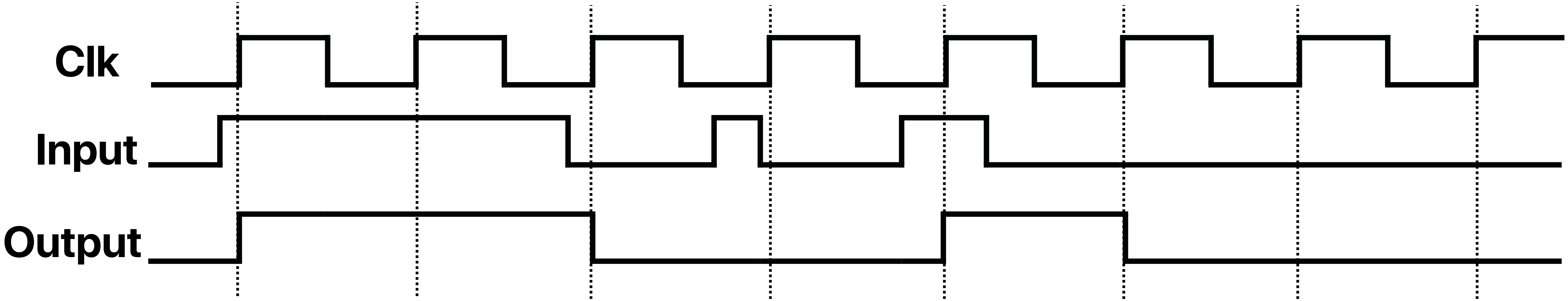
Only change Q/Q′ during positive clock edges



Output doesn’t hold for the whole cycle

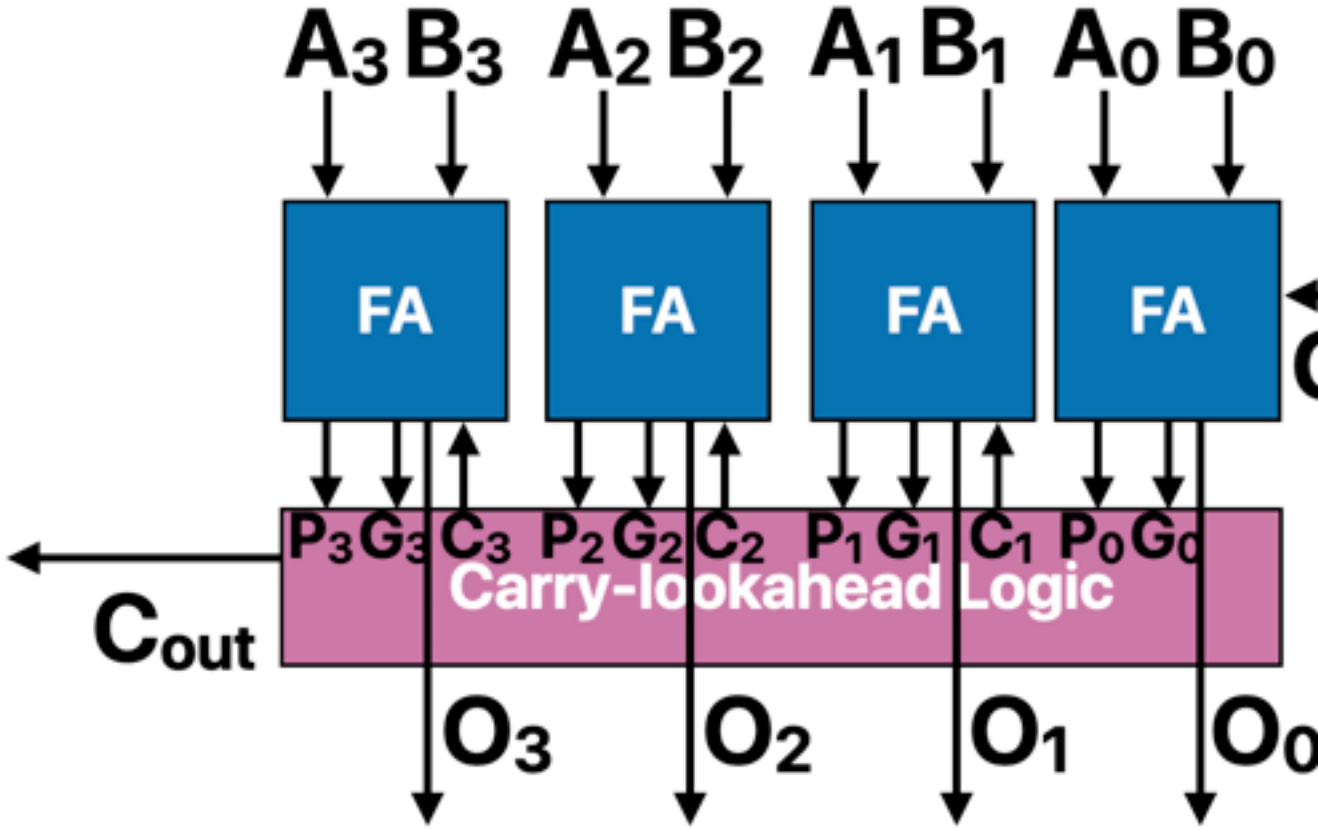
D flip-flop


Input
D
Q
D-latch
Clk
D
Q
D-latch
Clk
Output
Clk
Master-Slave D Flip-flop


Clk
Input
Output

What if ?

- Consider a 32-bit carry-lookahead adder built with 8 4-bit carry-lookahead adders. If we take the output after 4 gate delays and feed another input at that time, which of the following would be true?

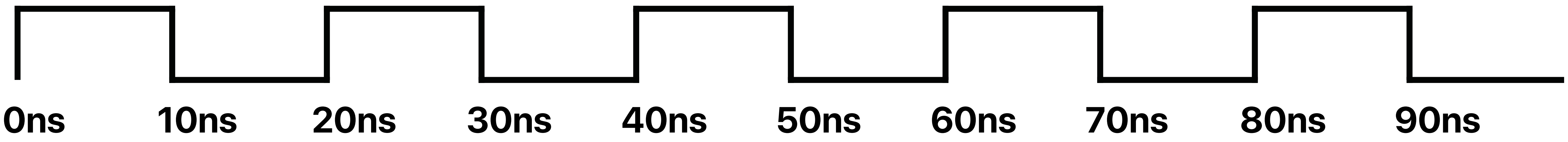

A_3 B_3 A_2 B_2 A_1 B_1 A_0 B_0
FA
FA
FA
FA
P_3 G_3 C_3 P_2 G_2 C_2 P_1 G_1 C_1 P_0 G_0
Carry-lookahead Logic
C_{out}
O_3
O_2
O_1
O_0

A. At the time we take the output, we can get the correct result

[✓ B. At the time we take the output, we cannot get the correct result]

C. At the time we take the output, we cannot get the correct result, but we can get the correct result after another 8 gate delays

Clock signal



- Clock -- Pulsing signal for enabling latches; ticks like a clock
- Synchronous circuit: sequential circuit with a clock
- Clock period: time between pulse starts
  - Above signal: period = 20 ns
- Clock cycle: one such time interval
  - Above signal shows 3.5 clock cycles
- Clock duty cycle: time clock is high
  - 50% in this case
- Clock frequency: 1/period
  - Above : freq = 1 / 20ns = 50MHz;

Sample Midterm

Midterm Format

- Format
  - Multiple choices * 30
  - Free answer questions (filling the blanks) *3
- Make sure your answer follow EXACTLY the same format that the question requires, otherwise, the auto-grader won't grade it correctly
- You may open book, create cheatsheets, just don't cheat
- Once opened, you only have one chance to finish — if your browser crashes because you opened too many windows/programs, I won't help you.
- If your submission is late by x sec, your grade is max(raw_score * ((100-x)/100),0)

Recap: Why are digital computers more popular now?

- Please identify how many of the following statements explains why digital computers are now more popular than analog computers.
  - ① The cost of building systems with the same functionality is lower by using digital computers.
  - ② Digital computers can express more values than analog computers.
  - ③ Digital signals are less fragile to noise and defective/low-quality components.
  - ④ Digital data are easier to store.
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Let’s practice!

- X, Y are two Boolean variables. Consider the following function:
  $X \bullet Y + X$
  How many of the following the input values of X and Y can lead to an output of 1
  ① X = 0, Y = 0
  ② X = 0, Y = 1
  ③ X = 1, Y = 0
  ④ X = 1, Y = 1
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

A Boolean equation is converted to a circuit in what order?

- A Boolean equation is converted to a circuit in what order
  A. Items within parentheses, then NOT, then AND, then OR.
  B. OR, then NOT, then AND, then items within parentheses.
  C. Items within parentheses, then AND, then OR, then NOT.
  D. NOT, then items within parentheses, then AND, then OR.

Boolean Equation from Truth Table

- Which equation best captures the following logic: Bob will pass the class only if doing all of the following: Bob attends all lectures, completes all assignments, passes all exams. Inputs: A = 1 indicates attends all lectures, Z = 1 indicates completes all assignments, E = 1 indicates passes all exams Outputs: P = 1 indicates passes the class

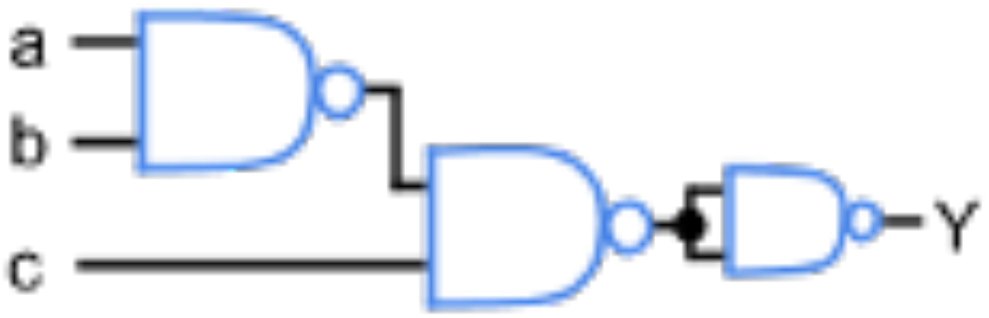
  A. P = A AND Z OR NOT(E)

  B. P = A OR Z OR E

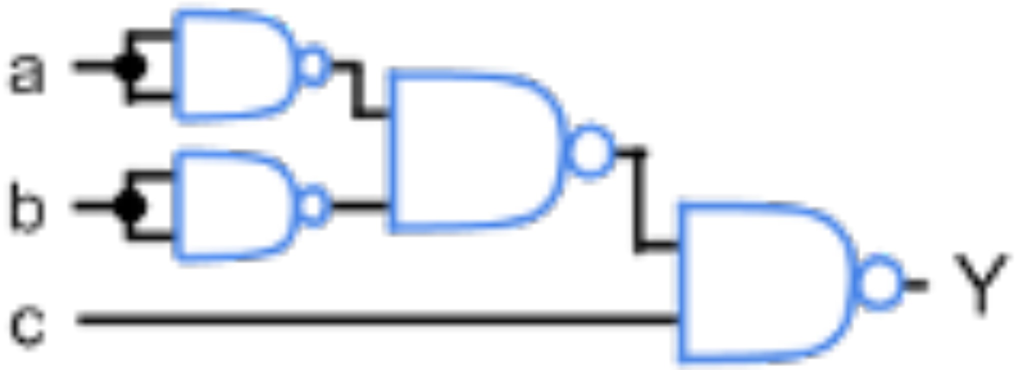
  C. P = A AND Z OR E

  D. P = A AND Z AND E

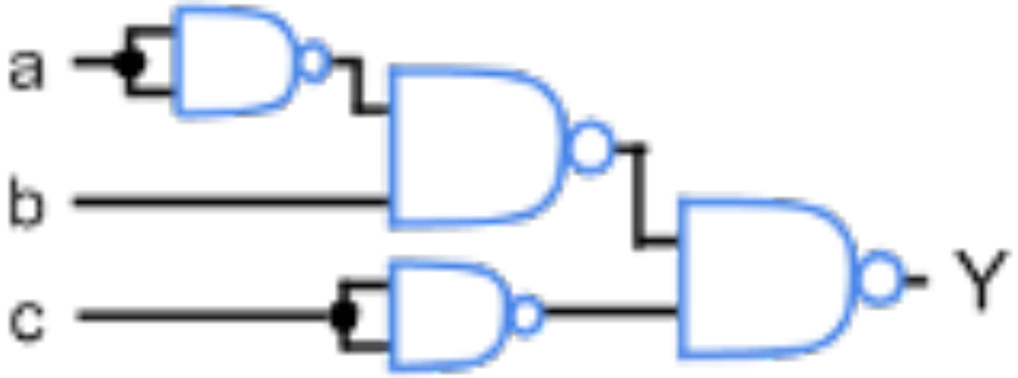
This equation Y = (a' + b)c is implemented by which circuit?



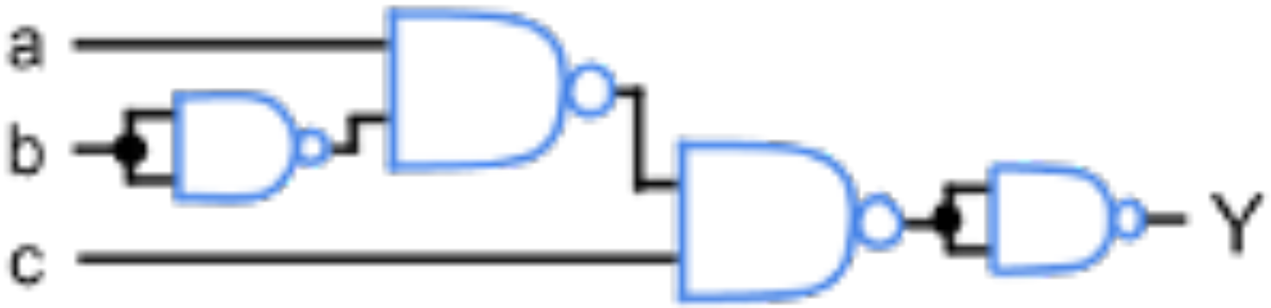
(1)



(2)



(3)



(4)

The sum-of-product form of the full adder

- How many of the following minterms are part of the sum-of-product form of the full adder in generating the output bit?

  ① A′B′Cin′

  ② A′BCin′

  ③ AB′Cin′

  ④ ABCin′

  ⑤ A′B′Cin

  ⑥ A′BCin

  ⑦ AB′Cin

  ⑧ ABCin

  A. 0

  B. 1

  C. 2

  D. 3

  E. 4

Practicing 2-variable K-map

- What’s the simplified function of the following K-map?
  A. A’
  B. A’B
  C. AB’
  D. B
  E. A

A B	0	1
0	0	0
1	1	1

Valid K-Maps

- How many of the followings are “valid” K-Maps?

(1)

	0,0	0,1	1,1	1,0
0	0	1	0	1
1	1	0	1	0

(2)

	0,1	1,1	1,0	0,0
0	1	0	1	0
1	0	1	0	1

(3)

	1,1	1,0	0,1	0,0
0	0	1	1	0
1	1	0	0	1

(4)

	0,0	0,1	1,0	1,1
0	0	1	1	0
1	1	0	0	1

A. 0

B. 1

C. 2

D. 3

E. 4

Minimum number of SOP terms

- Minimum number of SOP terms to cover the following function?
  A. 1
  B. 2
  C. 3
  D. 4
  E. 5

Input			Output
A	B	C	
0	0	0	**1**
0	0	1	**1**
0	1	0	**0**
0	1	1	**0**
1	0	0	**1**
1	0	1	**1**
1	1	0	**0**
1	1	1	**0**

Minimum number of SOP terms

- Minimum number of SOP terms to cover the following function?
  A. 1
  B. 2
  C. 3
  D. 4
  E. 5

Input			Output
A	B	C	
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**0**
1	1	0	**0**
1	1	1	**1**

Minimum SOP terms

- What′s the minimum sum-of-products expression of the given truth table?
  A. A′B′C′ + A′BC′+ A′BC + AB′C′
  B. A′B′C + AB + AC
  C. AB′C′ + B′C′
  D. A′B + B′C′
  E. A′C′ + A′B + AB′C′

Input			Output
A	B	C	
0	0	0	**1**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**1**
1	0	1	**0**
1	1	0	**0**
1	1	1	**0**

4-variable K-map

- What’s the minimum sum-of-products expression of the given K-map?
  1. B′C′ + A′B′
  2. B′C′D′ + A′B′ + B′C′D′
  3. A′B′CD′ + B′C′
  4. AB′ + A′B′ + A′B′D′
  5. B′C′ + A′C′D′

		A′B′	A′B	AB	AB′
		00	01	11	10
C′D′	00	1	0	0	1
C′D	01	1	0	0	1
CD	11	0	0	0	0
CD′	10	1	1	0	0

LT?

- What’s the minimum SOP presentation of LT?
  A. A′B′D′ + AC′ + BCD
  B. A'B'D + A'C + B′CD
  C. A'B'C'D' + A'BC'D + ABCD + AB′CD′
  D. ABCD + AB′CD′ + A′B′C′D′ + A′BC′D
  E. BC'D' + AC' + ABD'

Input				Output		
A	B	C	D	LT	EQ	GT
0	0	0	0	**0**	**1**	**0**
0	0	0	1	**1**	**0**	**0**
0	0	1	0	**1**	**0**	**0**
0	0	1	1	**1**	**0**	**0**
0	1	0	0	**0**	**0**	**1**
0	1	0	1	**0**	**1**	**0**
0	1	1	0	**1**	**0**	**0**
0	1	1	1	**1**	**0**	**0**
1	0	0	0	**0**	**0**	**1**
1	0	0	1	**0**	**0**	**1**
1	0	1	0	**0**	**1**	**0**
1	0	1	1	**1**	**0**	**0**
1	1	0	0	**0**	**0**	**1**
1	1	0	1	**0**	**0**	**1**
1	1	1	0	**0**	**0**	**1**
1	1	1	1	**0**	**1**	**0**

Input/output of a design

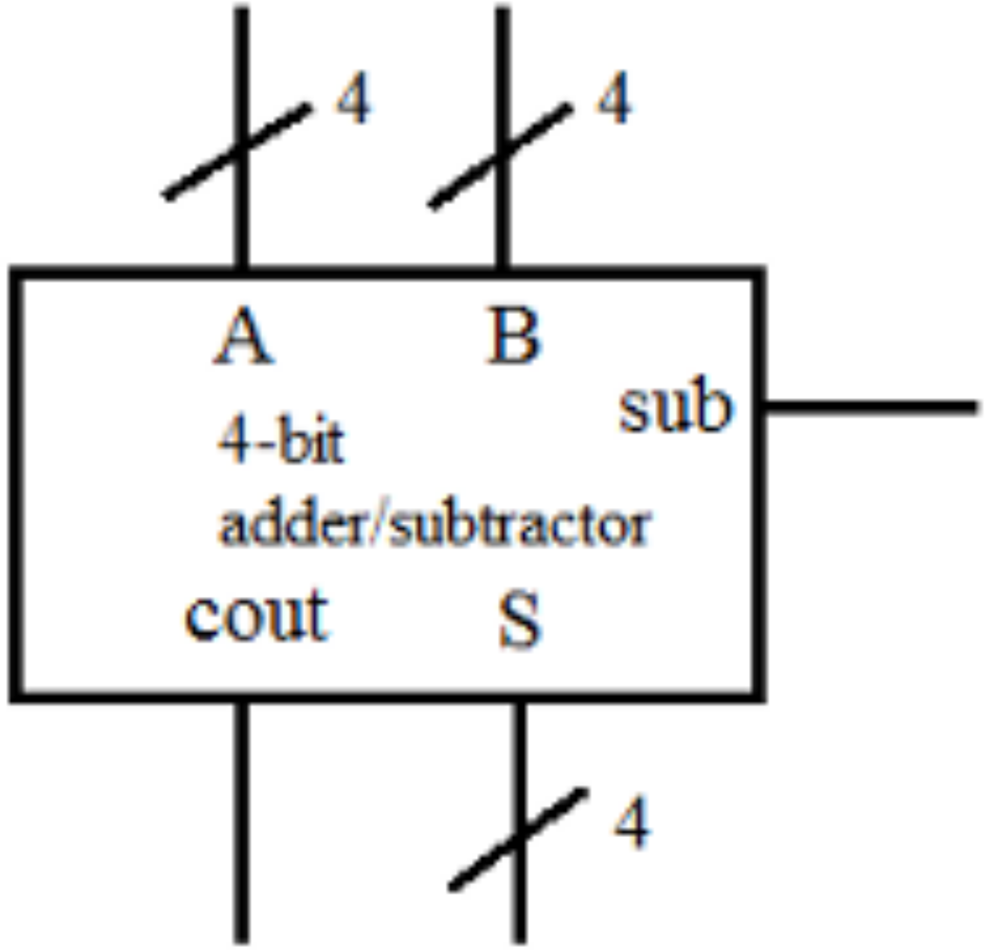
- A 4-bit adder/subtractor has inputs A = 0100, and B = 0010. What value of sub outputs sum S = 0110 and cout = 0000?

  A. 0

  B. 1

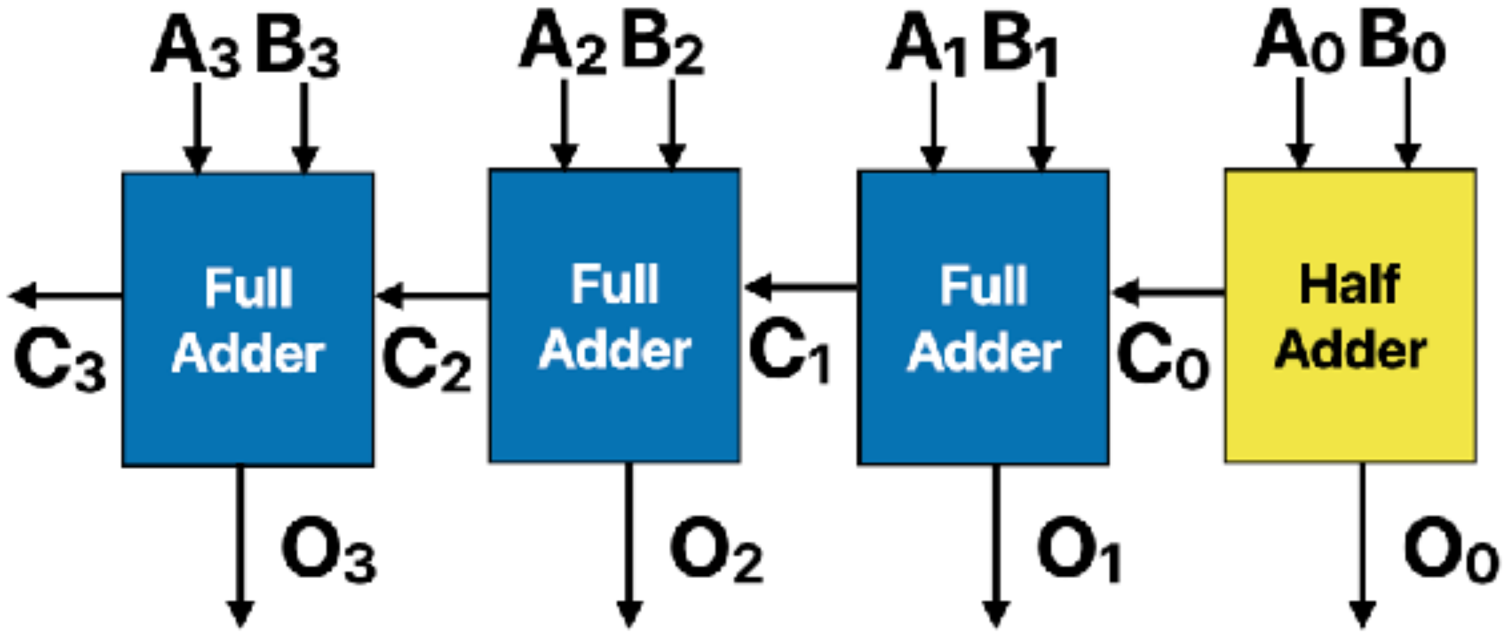
  C. 0000

  D. 1111



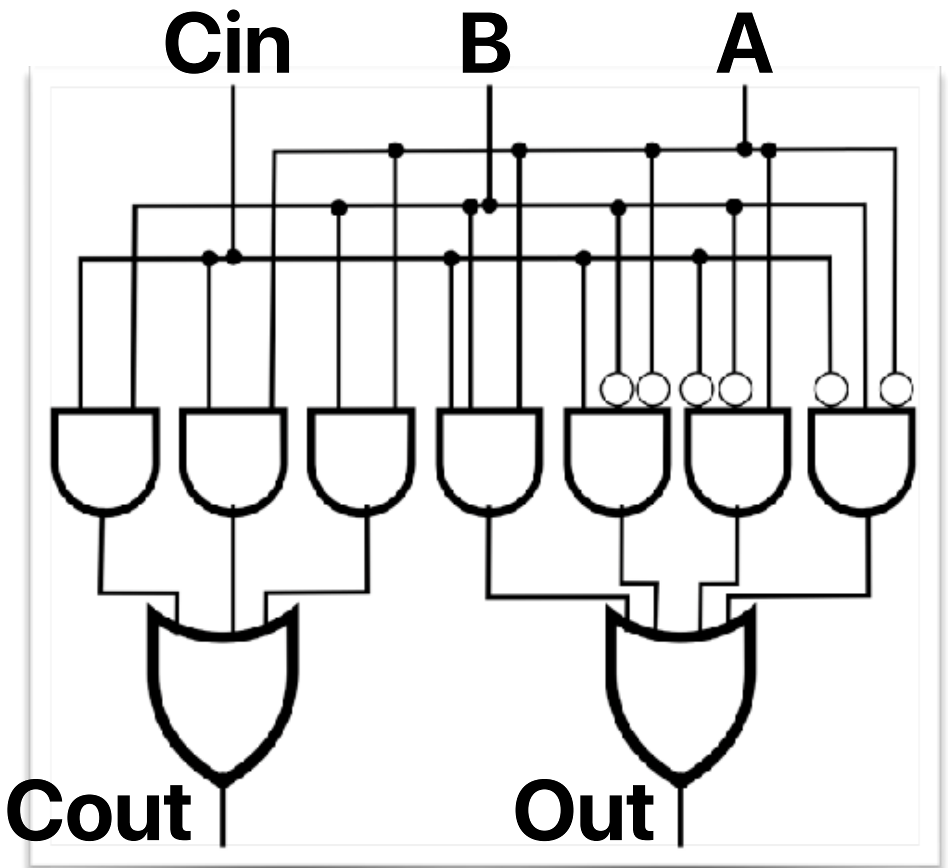
If we want to support subtraction?

- If we would like to extend the 4-bit adder that we've built before to support "A-B" with 2's complement, how many of the followings should we add at least?
  ① Provide an option to use bitwise NOT A
  ② Provide an option to use bitwise NOT B
  ③ Provide an option to use bitwise A XOR B
  ④ Provide an option to add 0 to the input of the half adder
  ⑤ Provide an option to add 1 to the input of the half adder
  A. 1
  B. 2
  C. 3
  D. 4
  E. 5



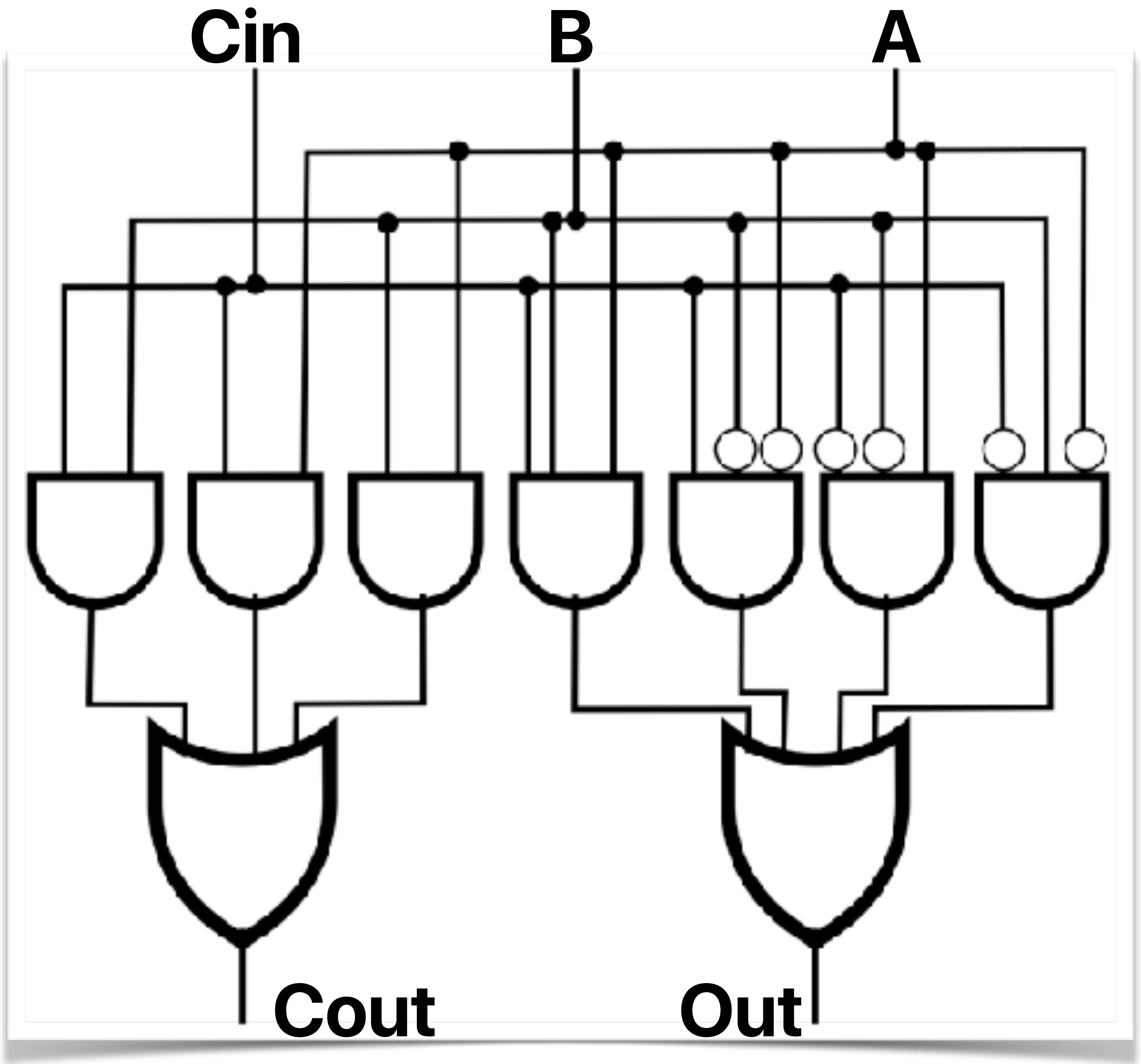
How efficient is the adder?

- 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



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



CLA's gate delay

- What's the gate-delay of a 4-bit CLA?
  A. 2
  B. 4
  C. 6
  D. 8
  E. 10

$G_i = A_iB_i$

$P_i = A_i \text{ XOR } B_i$

$C_1 = G_0 + P_0 C_0$

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

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2 C_2$

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

$C_4 = G_3 + P_3 C_3$

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

CLA's size

- How many transistors do we need to implement a 4-bit CLA logic?
  A. 38
  B. 64
  C. 88
  D. 116
  E. 128

$G_i = A_iB_i$

$P_i = A_i \text{ XOR } B_i$

$C_1 = G_0 + P_0 C_0$

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

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2 C_2$

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

$C_4 = G_3 + P_3 C_3$

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

How big is the 4-bit 4:1 MUX?

- How many estimated transistors are there in the 4-bit 4:1 MUX?
  A. 48
  B. 64
  C. 80
  D. 128
  E. 192

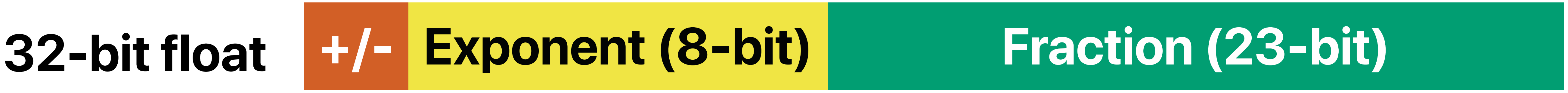
Gate delay of 8:1 MUX

- What’s the estimated gate delay of an 8:1 MUX?
  A. 1
  B. 2
  C. 4
  D. 8
  E. 16

16-1 MUX

- How many AND gates does a 16x1 mux require?
  A. 2
  B. 4
  C. 8
  D. 16

IEEE 754 format



32-bit float +/- Exponent (8-bit) Fraction (23-bit)

- Realign the number into 1.**F** * $2^{\mathbf{e}}$
- Exponent stores **e** + 127
- Fraction only stores **F**
- Convert the following number
  1 1000 0010 0100 0000 0000 0000 0000 000
  A. - 1.010 * 2^130
  B. -10
  C. 10
  D. 1.010 * 2^130
  E. None of the above

Why stuck at 16777216?

- Consider the following C program.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float i=1.0;
    while(i > 0) i++;
    printf("We're done! %f\n",i);
    return 0;
}
```

Why i stuck at 16777216.000?

A. It's a special number in IEEE 754 standard that an adder will treat it differently

B. It's a special number like +Inf/-Inf or +NaN/-NaN with special meaning in the IEEE 754 standard

C. It's just the maximum integer that IEEE 754 standard can represent

D. It's nothing special, but just happened to be the case that 16777216.0+1.0 will produce 16777216.0

E. It's nothing special, but just happened to be the case that 16777216.0 add anything will become 16777216.0

Will the loop end? (one more run)

- Consider the following C program.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    float i=1.0;
    while(i > 0) i++;
    printf("We're done! %f\n",i);
    return 0;
}
```

Please identify the correct statement.

A. The program will finish since i will end up to be +0
B. The program will finish since i will end up to be -0
C. The program will finish since i will end up to be something < 0
D. The program will not finish since i will always be a positive non-zero number.
E. The program will not finish but raise an exception since we will go to NaN first.

Will the loop end? (last run)

- Consider the following C program.

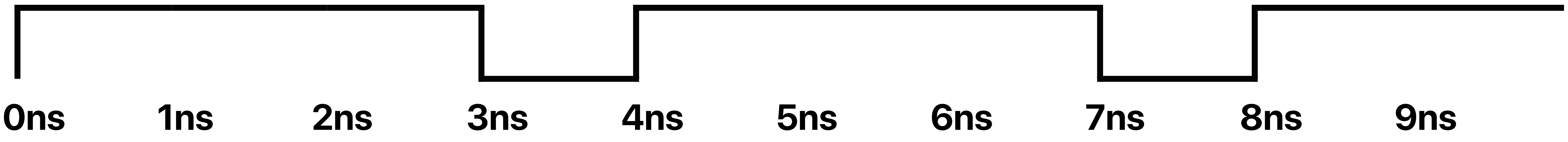
```
#include <stdio.h>

int main(int argc, char **argv)
{
    float i=1.0;
    while(i > 0) i+=i;
    printf("We're done! %f\n",i);
    return 0;
}
```

Please identify the correct statement.

A. The program will finish since i will end up to be +0
B. The program will finish since i will end up to be something < 0
C. The program will not finish since i will always be a positive non-zero number.
D. The program will not finish since i will end up staying at some special FP32 presentation
E. The program will not finish but raise an exception since we will go to NaN first.

Clock signal



- Regarding the above clock signal, please identify how many of the following statements are correct?

① Clock period of 4ns with 250MHz frequency

② Clock duty cycle 75%

③ Clock period of 1ns with 1GHz frequency

④ The above contains two complete clock cycles.

A. 0

B. 1

C. 2

D. 3

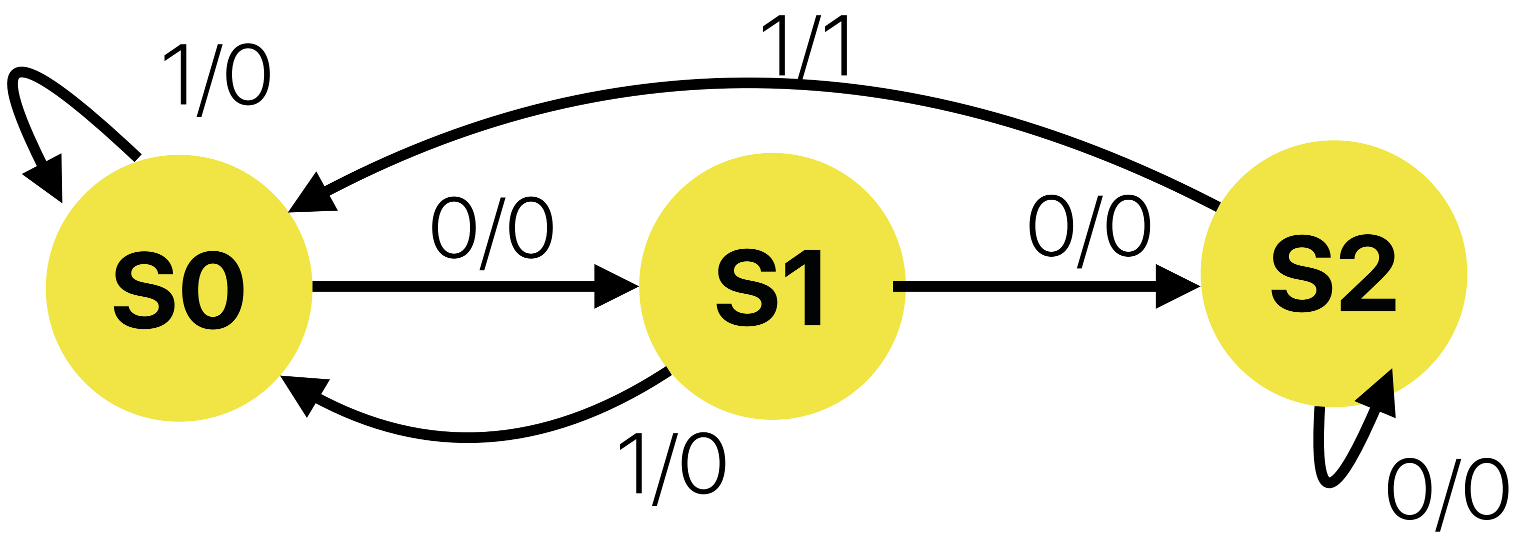
E. 4

FSM for Life on Mars

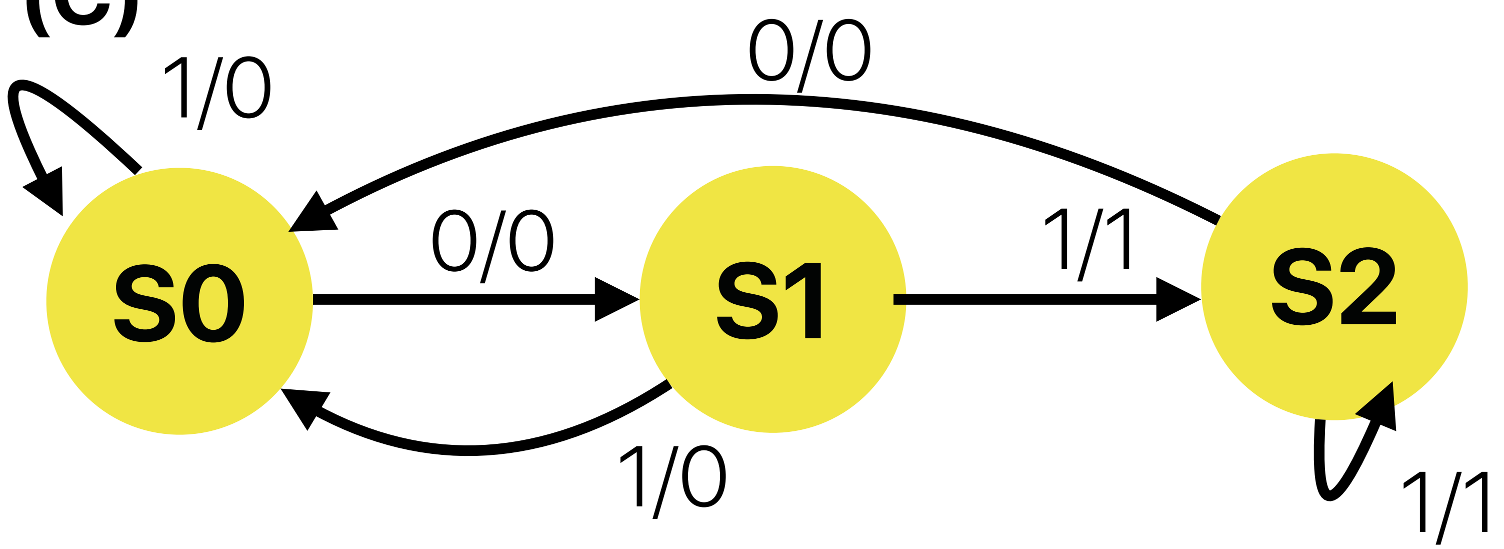
1/0 == Input 1/Output 0

- Which of the following diagrams is a correct FSM for the 001 pattern recognizer on the Mars rover? (If sees “001”, output “1”)

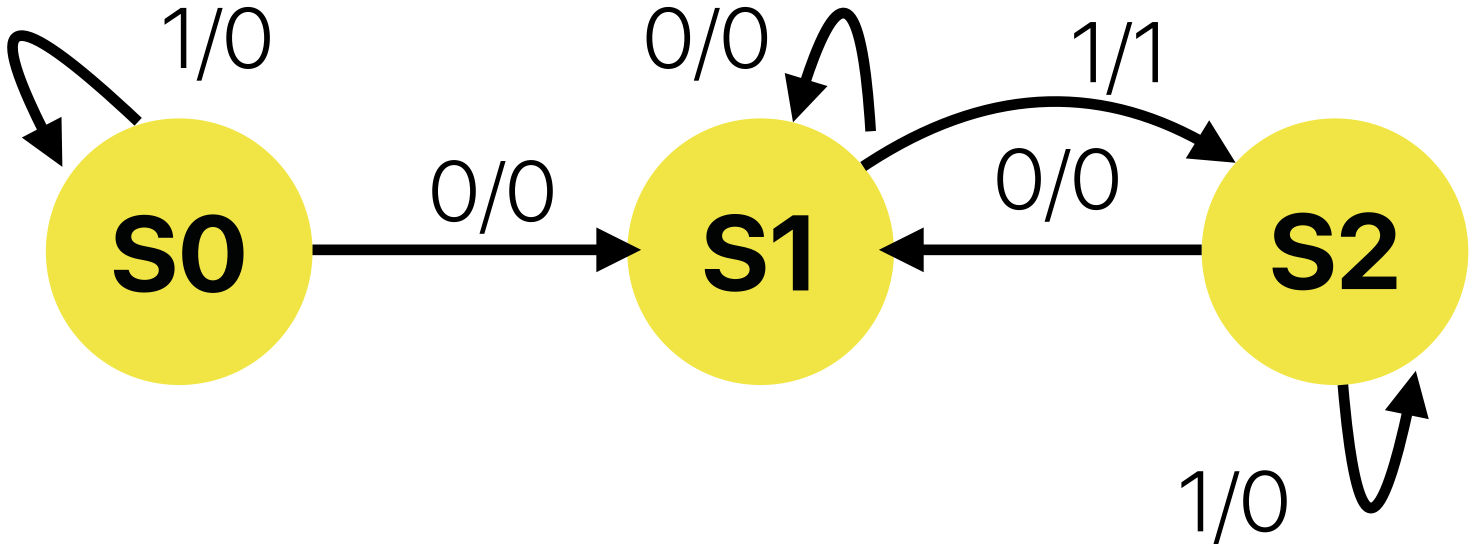
(A)



(C)



(B)

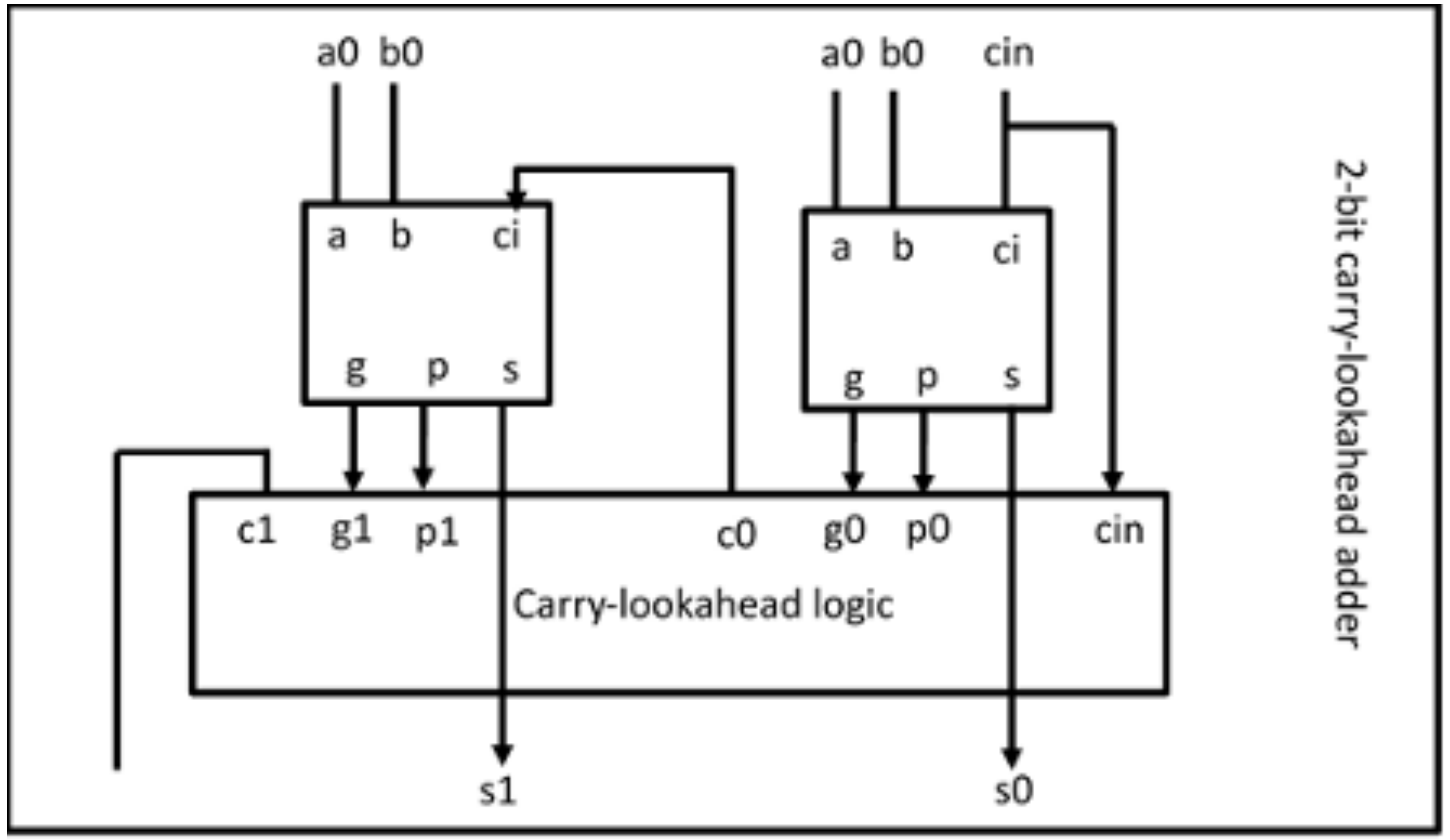


(D) All are correct

(E) None is correct

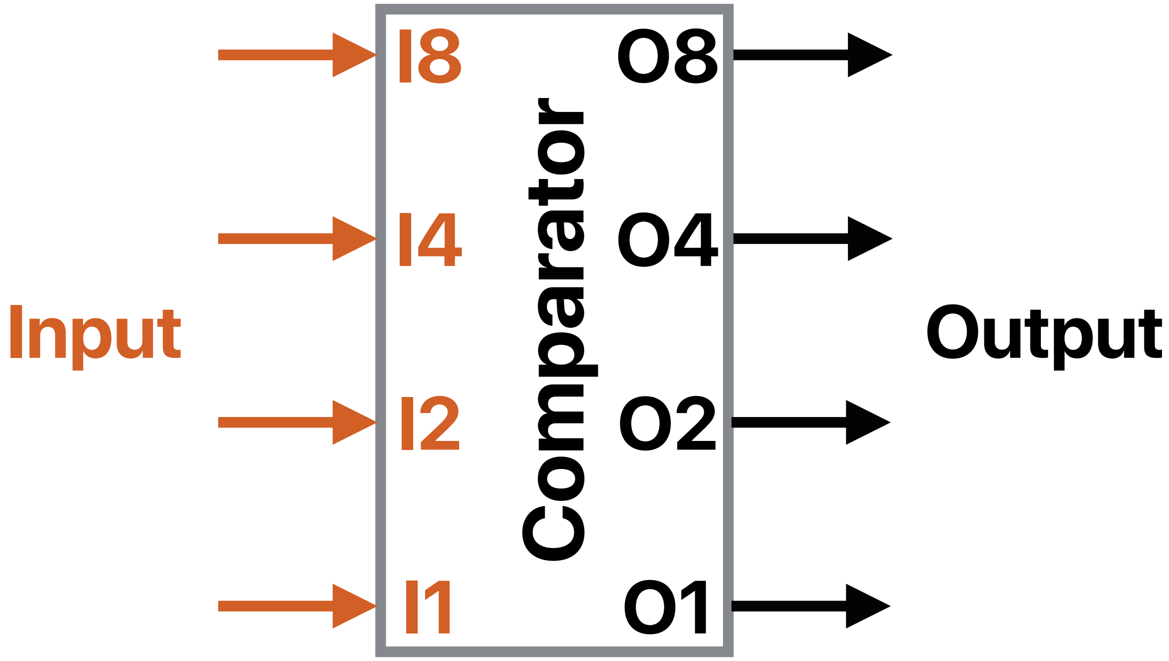
2-bit CLA

- Which is true about the given 2-bit carry-lookahead adder? Hint: g = ab, p = a + b, and the expression for each digit's carry-out is co = ab + (a + b)ci = g + p·ci.
  A. c0 = 0, when a0 = 1, b0 = 1, and cin = 0
  B. c0 = 1, when a0 = 1, b0 = 1, and cin = 1
  C. c1 = 1, when cin = 1, g0 = 1, p0 = 1, g1 = 0, and p1 = 0
  D. c1 = 0, when cin = 0, g0 = 1, p0 = 1, g1 = 1, and p1 = 1



BCD+1 — Binary coded decimal + 1

- 0x0 — 1
- 0x1 — 2
- 0x2 — 3
- 0x3 — 4
- 0x4 — 5
- 0x5 — 6
- 0x6 — 7
- 0x7 — 8
- 0x8 — 9
- 0x9 — 0
- 0xA — 0xF — Don’t care



Can you write the truth table?
Can you create a K-map?
Can simplify the boolean equation?

What’s the output of this? and Why?

```
#include <stdio.h>

int main(int argc, char **argv)
{
    float a, b, c, d;
    int i = 0;
    a = 1.2;
    b = 1.0;
    c = a + b;
    printf("A: %d\n", c==2.2);

    a = 33554432.0;
    b = 2.0;
    c = a+b;
    printf("B: %d\n", c, d, c==33554434.0);

    a = 1.0;
    for(i=0;i<200;i++)
        a += a;
    printf("C: %f\n", a);

    a = a/0.0;
    printf("D: %f\n", a);
    return 0;
}
```

Other questions to think about

- What are the differences among SR-latch, D-latch, D-flip flop?
- What's pMOS? What's nMOS?
- What's the difference between sequential logic and combinational logic?

Announcement

- Assignment #3 due tonight— **Chapter 3.6-3.16 & 4.1-4.4 & 4.8-4.9**
- Midterm on 5/7 during the lecture time, access through iLearn
  - No late submission is allowed — make sure you will be able to take that at the time
  - Covers: Chapter 1, Chapter 2, Chapter 3.1 — 3.12, Chapter 3.15 & 3.16, Chapter 4.1—4.9
- Lab 4 is up — due after final (5/12).
- Check your grades in iLearn

Electrical
Computer Science
Engineering

120A

つづく