

9/28/26

COMP 311:

Computer Organization!

Lecture 10: CMOS Gates, Boolean Logic, Truth Tables



The University
of North Carolina
at Chapel Hill

<https://tinyurl.com/comp311-fa26>

Logistics Update



10/7

- Lab 1 is due this week!
- I'm pushing back quiz 3. The others are staying!
- WA2 released last Wednesday. Due 10/5

Plan for Class Today



- 3:35-4:00: CMOS Review
- 4:00-4:15: Logic Gates
- 4:15-4:20: Break
- 4:20-4:50: Representing Logic (Truth Tables, Equations, Gates)

Guiding Question ☺

How can we move between truth tables, Boolean equations and circuit diagrams?

UNIT 1: PROGRAMMING THE MACHINE

Transistors 😊

Transistors are the physical building blocks of modern computers!

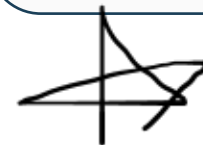


Number of Transistors in CPUs over Time



A transistor is the physical device we use to turn continuous electrical behavior into digital 0s and 1s.

You can think of it as an electrically controlled switch that turns current on or off.



Transistors are like switches



Switches are controlled by **physical contact**

Transistors are controlled by a **voltage**

Two States: Binary!



- Transistors can be in two distinct states: on or off
- This is why our machines communicate in binary (1s and 0s)!

On



Off



Transistors

n-type
↳ carrying electrons

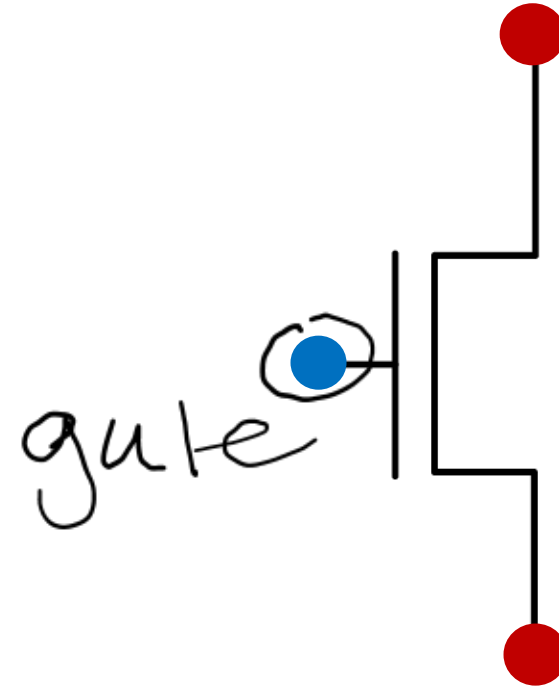


- We will be working with two types of transistors
 - nMOS transistors
 - pMOS transistors
- There are three terminals
 - • **Gate**: controls whether the transistor is on or off
 - • **Source** and **Drain**: endpoints

Transistor Symbol



Switch



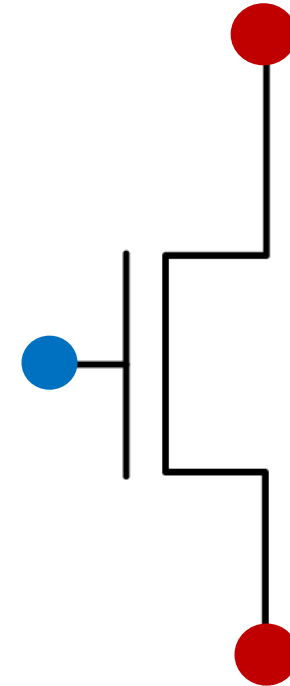
Transistor
(n-channel MOSFET)

The terminal labeled with the **blue** dot determines whether current can flow between the terminals labeled with the **red**

Transistor Symbol



Switch

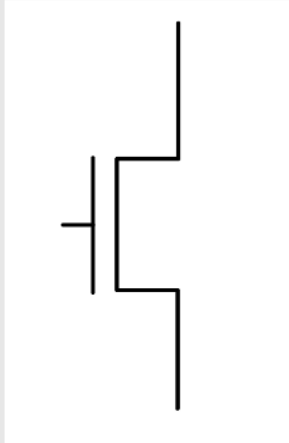
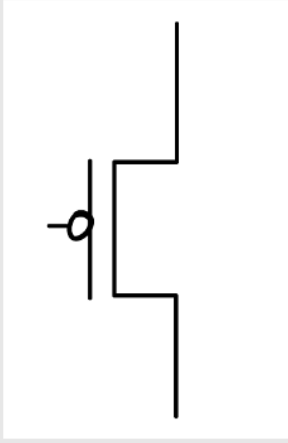


Transistor
(n-channel MOSFET)

The terminal labeled with the **blue** dot determines whether current can flow between the terminals labeled with the **red** dot.

Two Types of Transistors



	<u>nMOS</u>	<u>pMOS</u>
		
Behaves as an open switch when	V_g is low	V_g is high
Behaves as a closed switch when	V_g is high	V_g is low

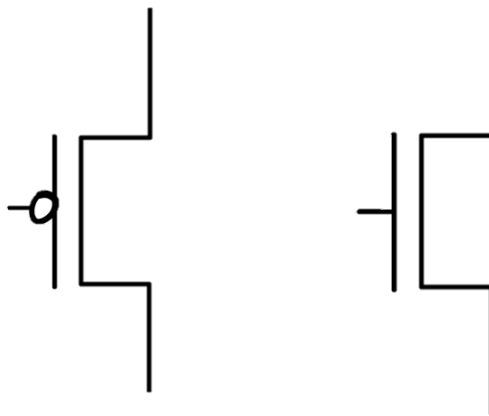
Building Logic Gates with Transistors

Abstraction 😊

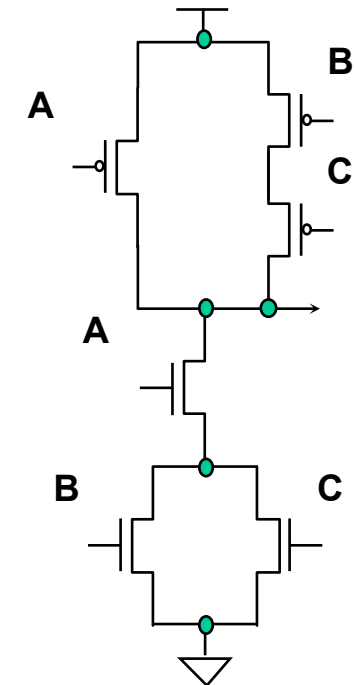
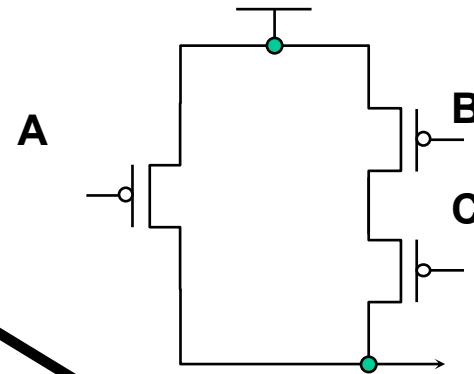


CMOS Gates

Physical Transistors



CMOS gates abstract away individual transistors, letting us design reliable logic at a higher level.



CMOS gates are the abstraction that lets many transistors behave like a single logical unit.

Building Logic Gates with Transistors



- Engineering Digital Behavior from Analog Devices
 - NMOS transistors act as controllable connections to ground
 - PMOS transistors act as controllable connections to power

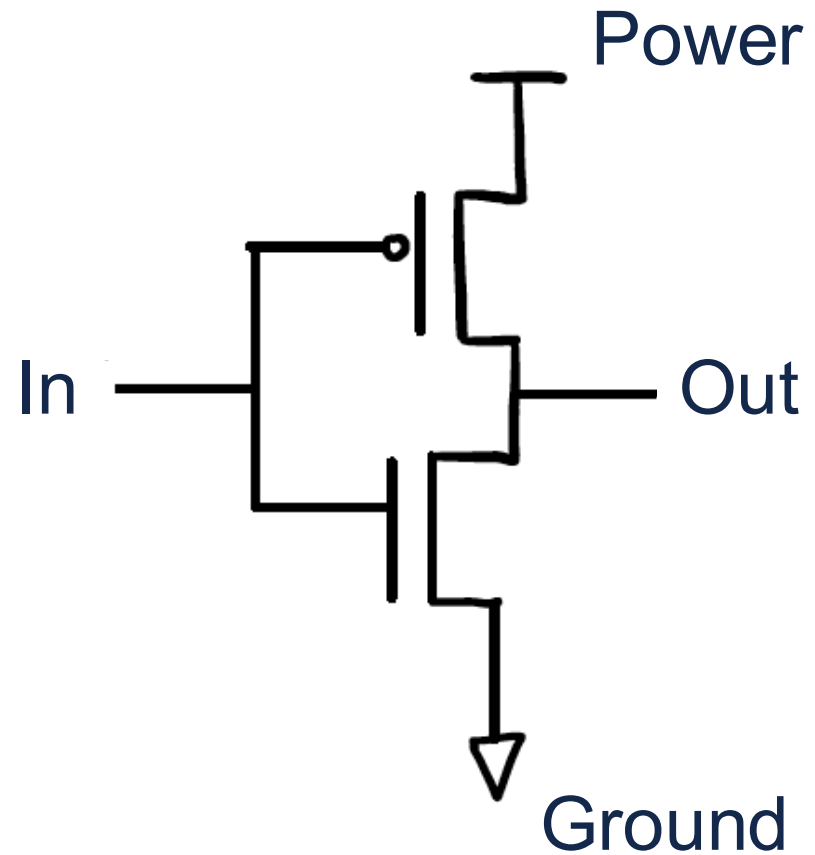
CMOS gates work by carefully arranging these devices so physics is forced to produce clean digital logic!

Another way to say this: CMOS gates work by using NMOS transistors to pull outputs down and PMOS transistors to pull outputs up.

CMOS Inverter



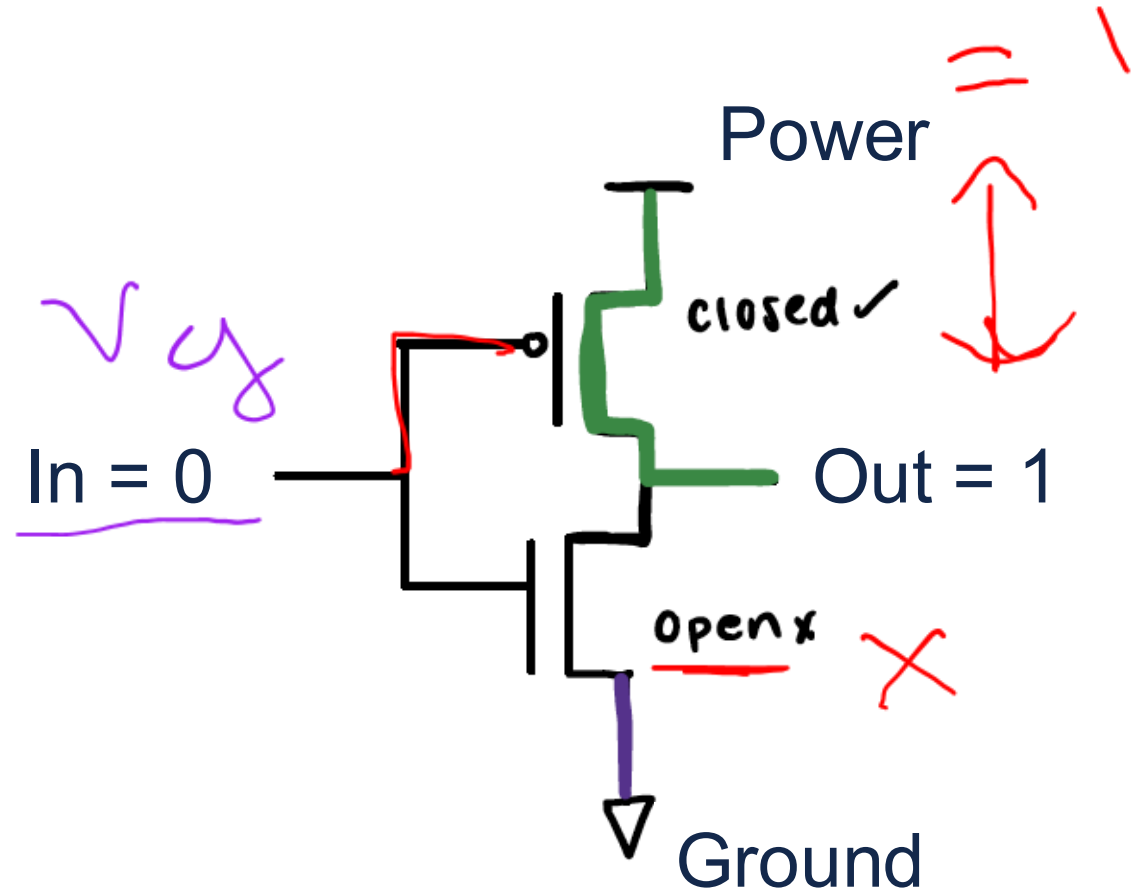
In	Out
0	1
1	0



CMOS Inverter



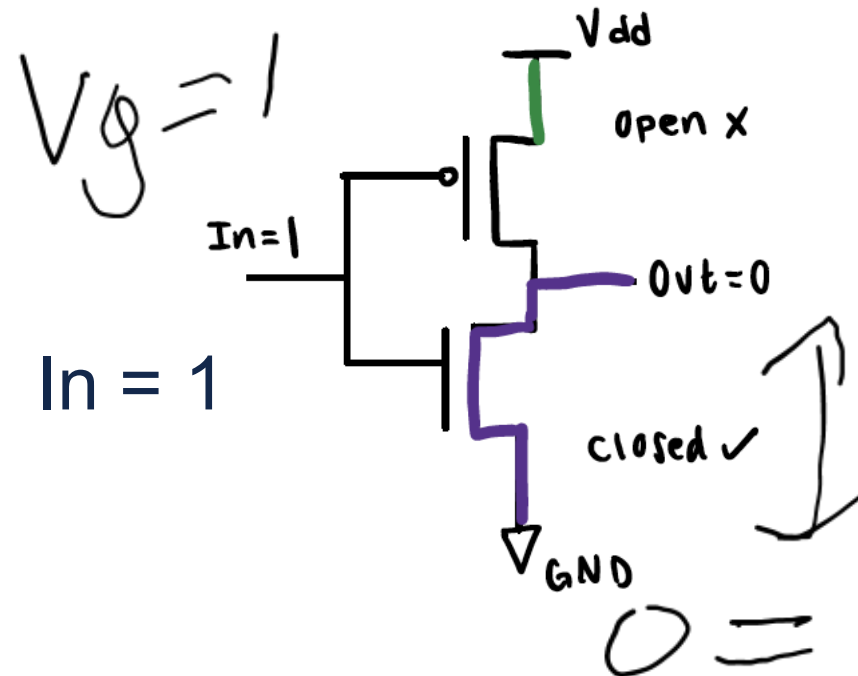
In	Out
0	1
1	0



CMOS Inverter



In	Out
0	1
1	0



Power

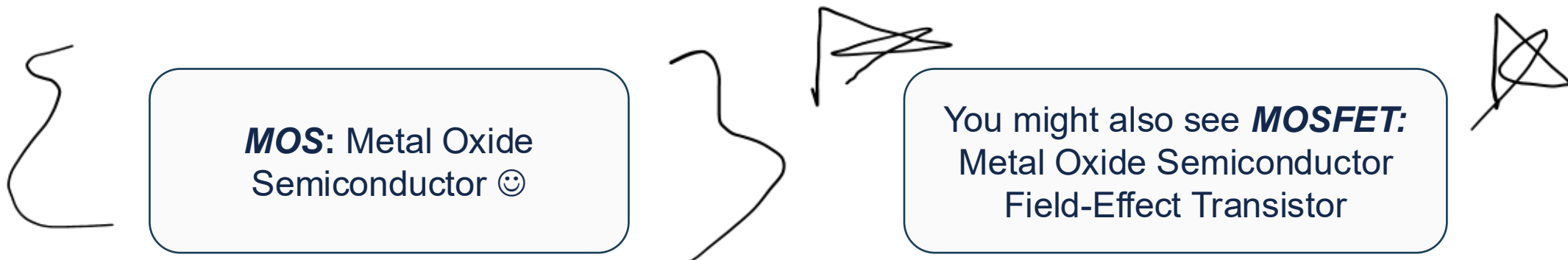
Out = 0

Ground

CMOS Gates



- CMOS = Complementary MOS
 - The pMOS and nMOS transistors complement each other to form the logic gate
- Common confusion
 - CMOS is not a type of transistor!
 - It is a technology that combines both nMOS and pMOS transistors



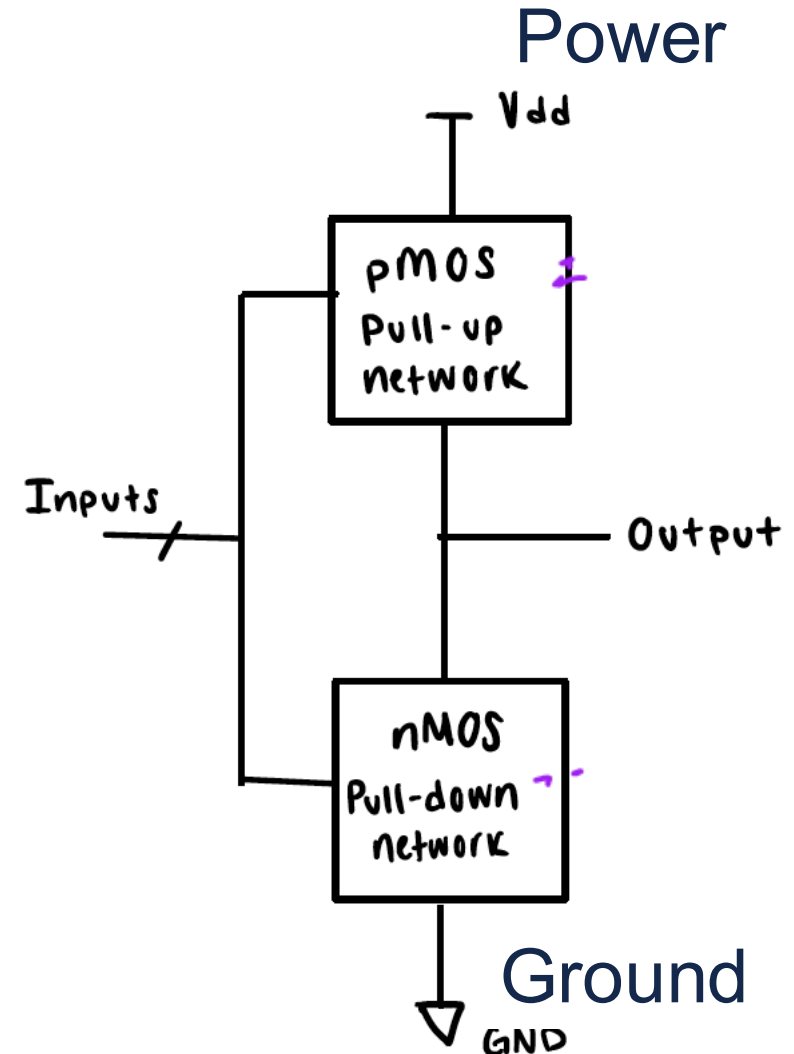
Pull-up and Pull-down Networks



- Only one network will be on at a time
- The pull-up network drives the output when the output is 1
- The pull-down network drives the output when the output is 0

PMOS

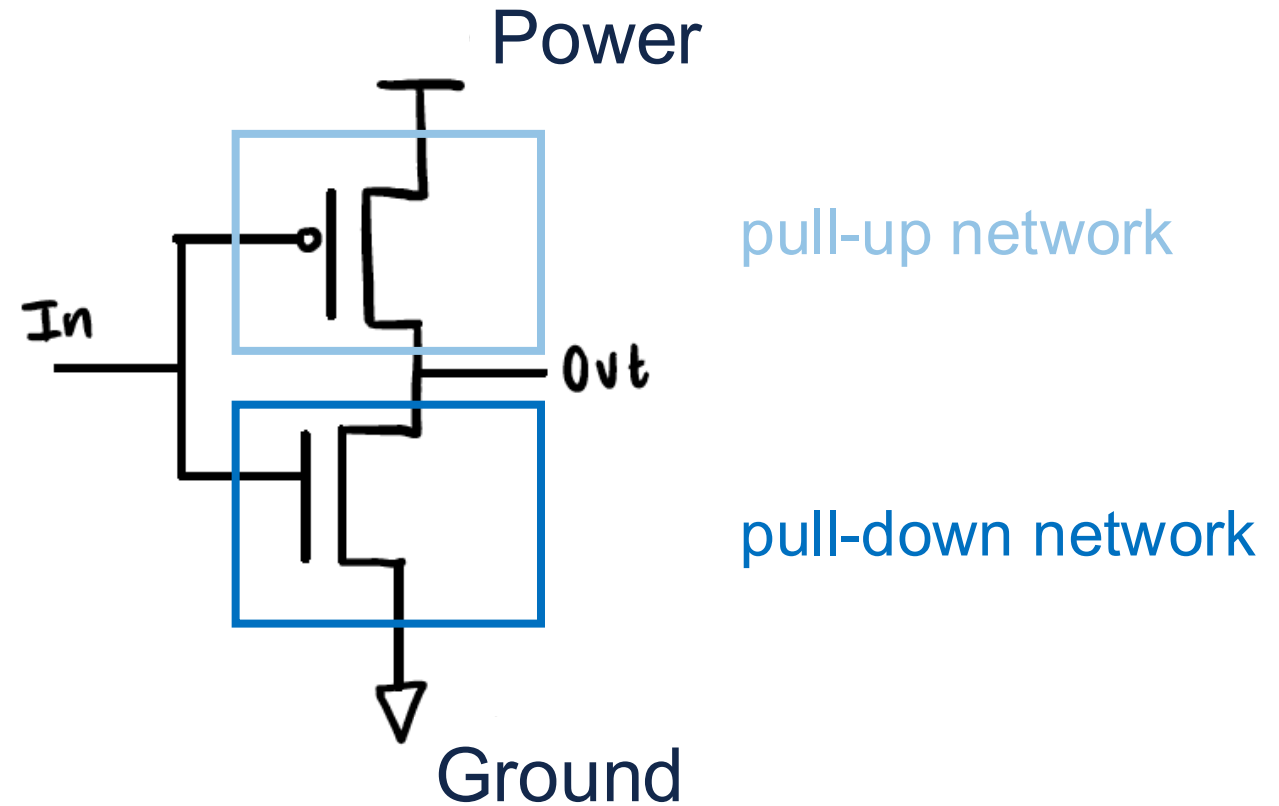
nmos



CMOS Inverter



In	Out
0	1
1	0



General CMOS Gate Recipe

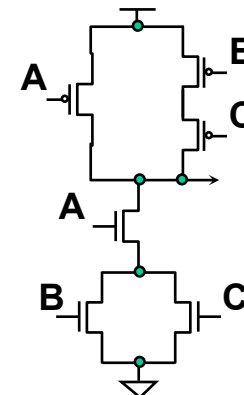
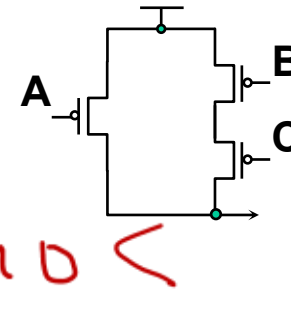
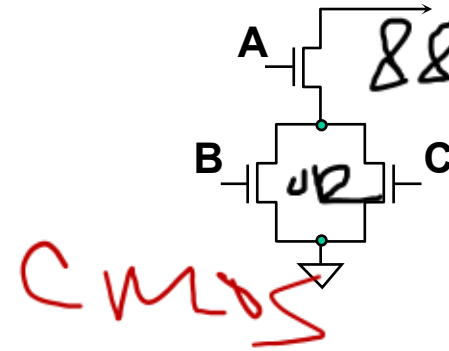


Step 1. Figure out pulldown network that does what you want (i.e the set of conditions where the output is '0')

e.g., $F = A*(B+C)$

Step 2. Build the "opposite" version of that network to figure out when the output should be 1.

Step 3. Combine the two so you have a circuit that always pulls the output either up to 1 or down to 0.



Recall:
Series = *AND*
Parallel = *OR*

This comes from DeMorgan's Laws! In practice, we swap series + parallel connections, use pMOS instead of nMOS

But isn't it hard to wire it all up?



Yes it is!
This does not scale...

Designing hardware at the transistor level...



- Does not scale!
- What should we do?
 - Abstract again!
 - Logic gates are next!
- But before we jump to the next level of abstraction, let's cover some ***standard, easy-to-wire*** primitives
 - NAND and NOR
 - These functions are universal, meaning any Boolean function or digital logic can be built from them



CMOS NOR

CMOS NOR

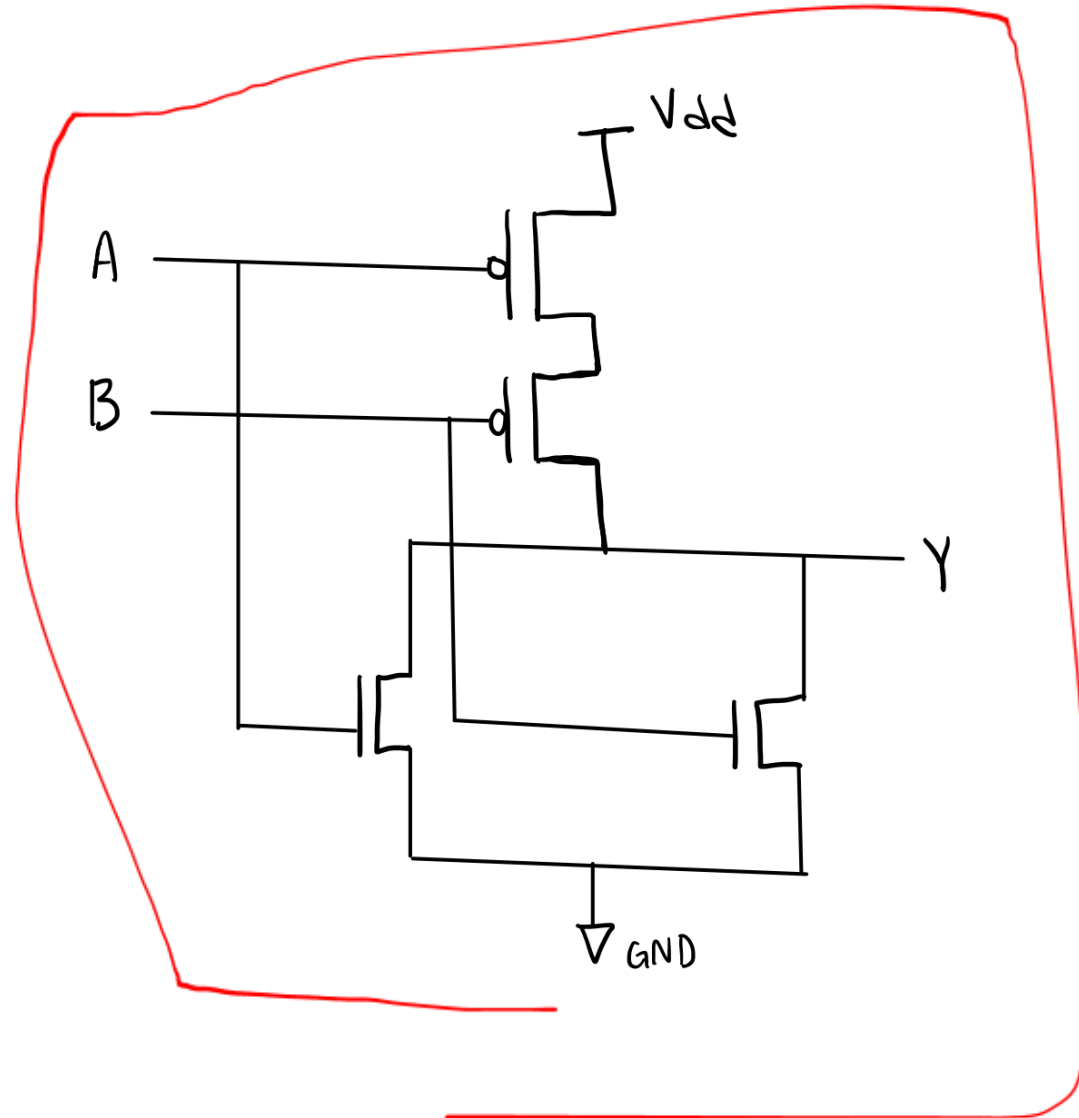


A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

CMOS NOR



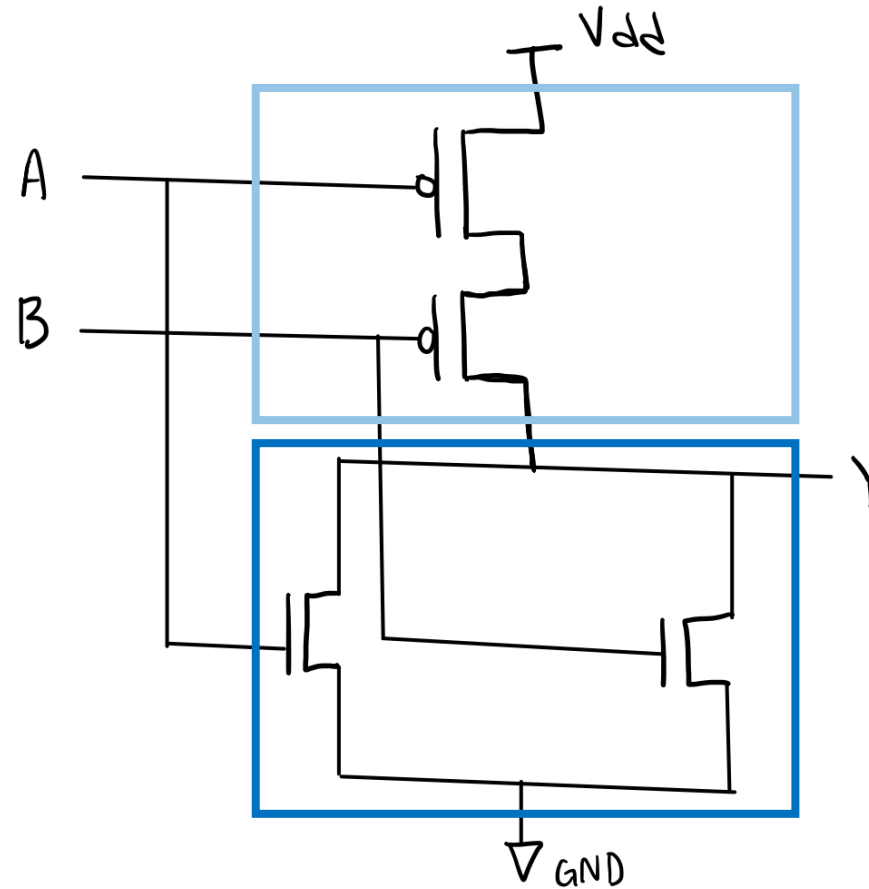
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



CMOS NOR



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



pull-up network

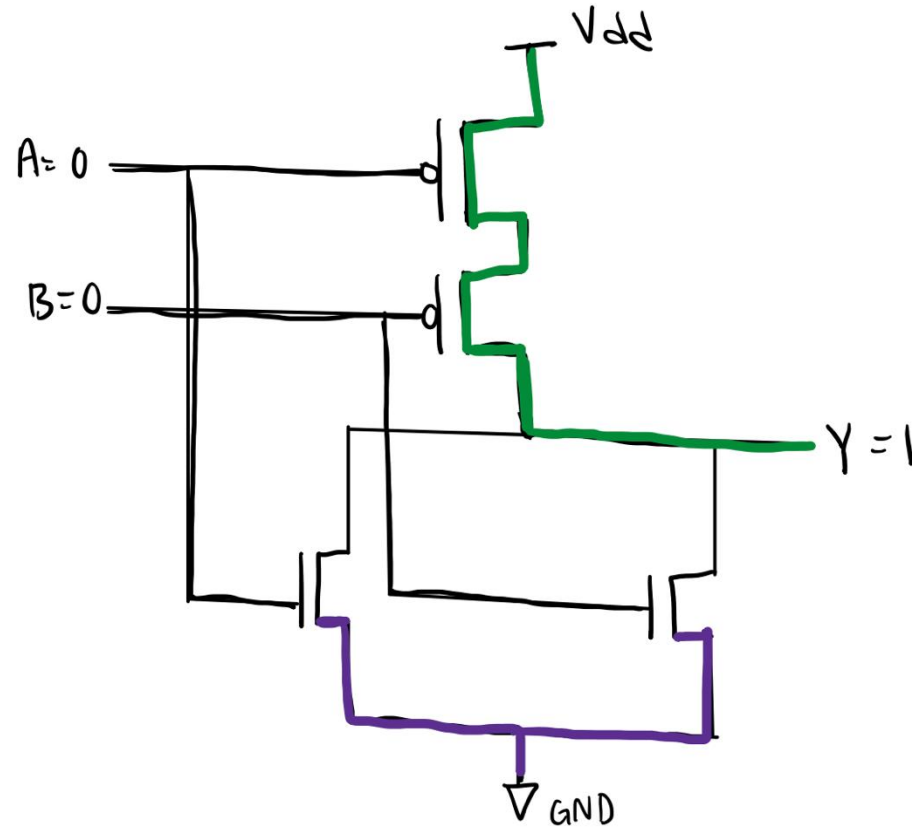
pull-down network

CMOS NOR



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

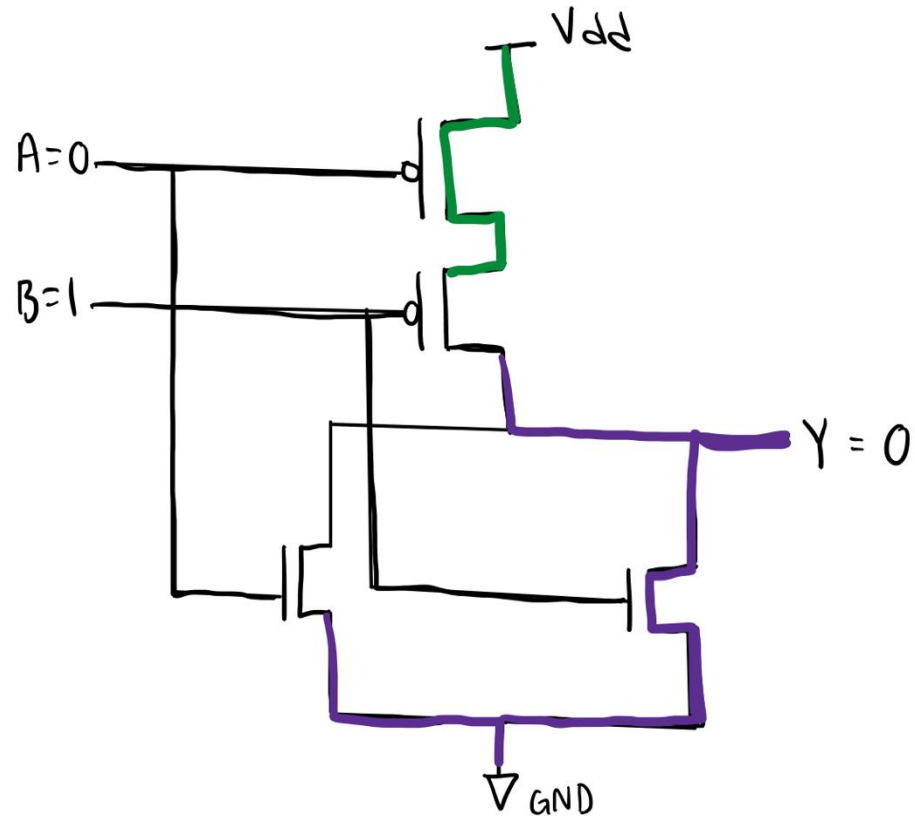
both pMOS on,
output pulled up



CMOS NOR



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0



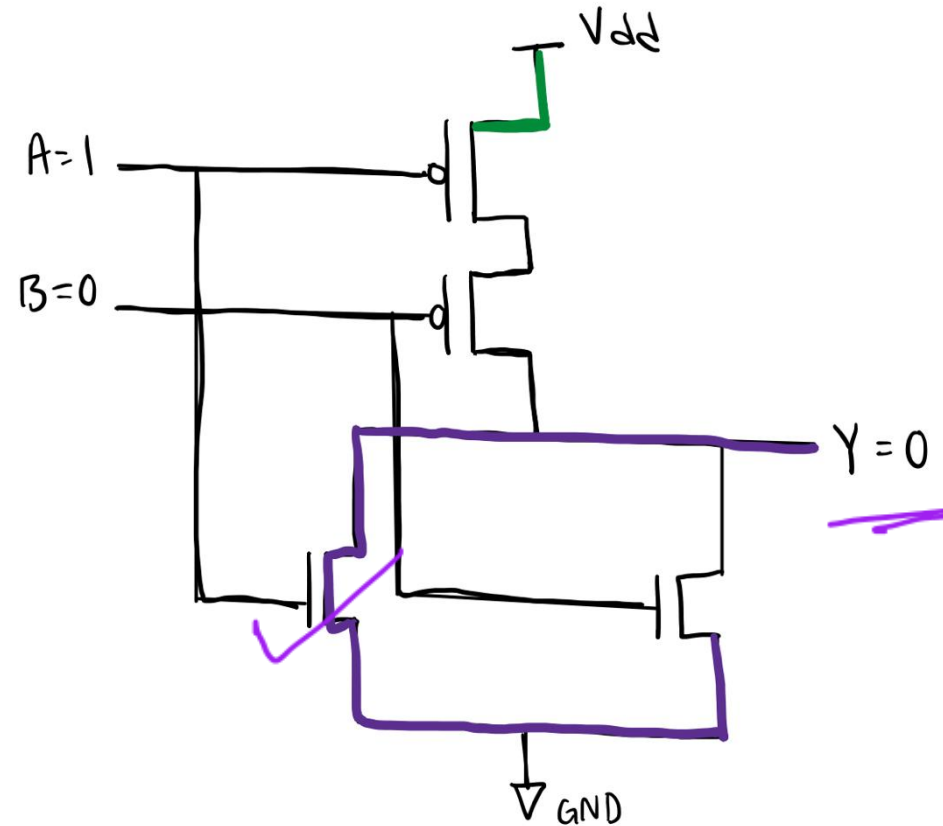
B's nMOS on,
pulls down

CMOS NOR



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

A's nMOS on,
pulls down

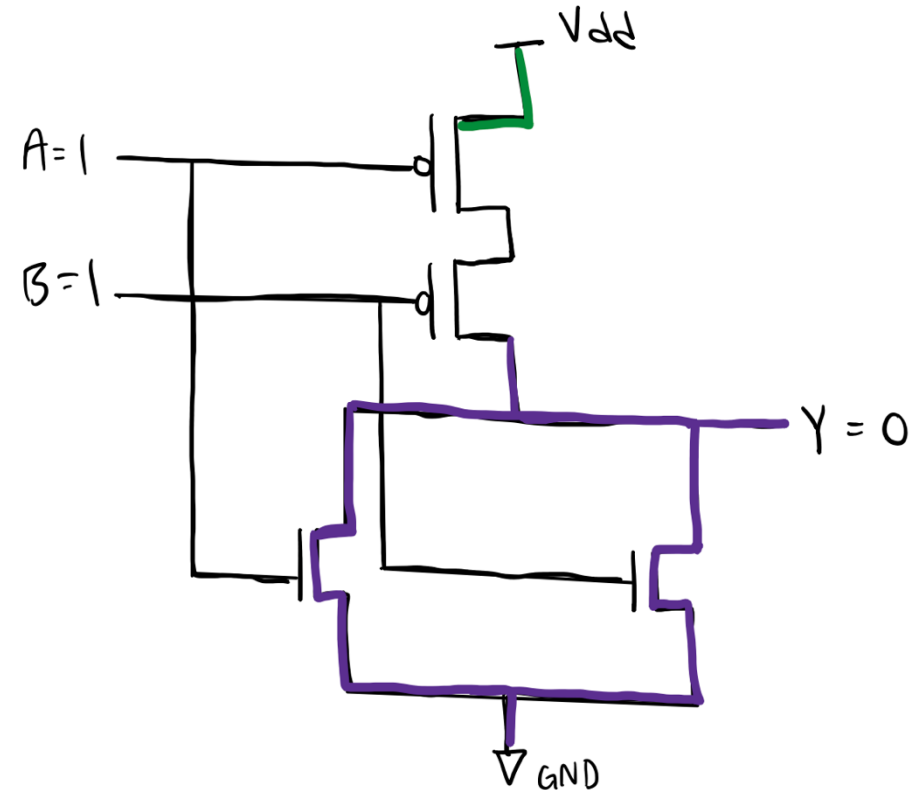


CMOS NOR



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

both nMOS on,
strong pull down





CMOS NAND

CMOS NAND

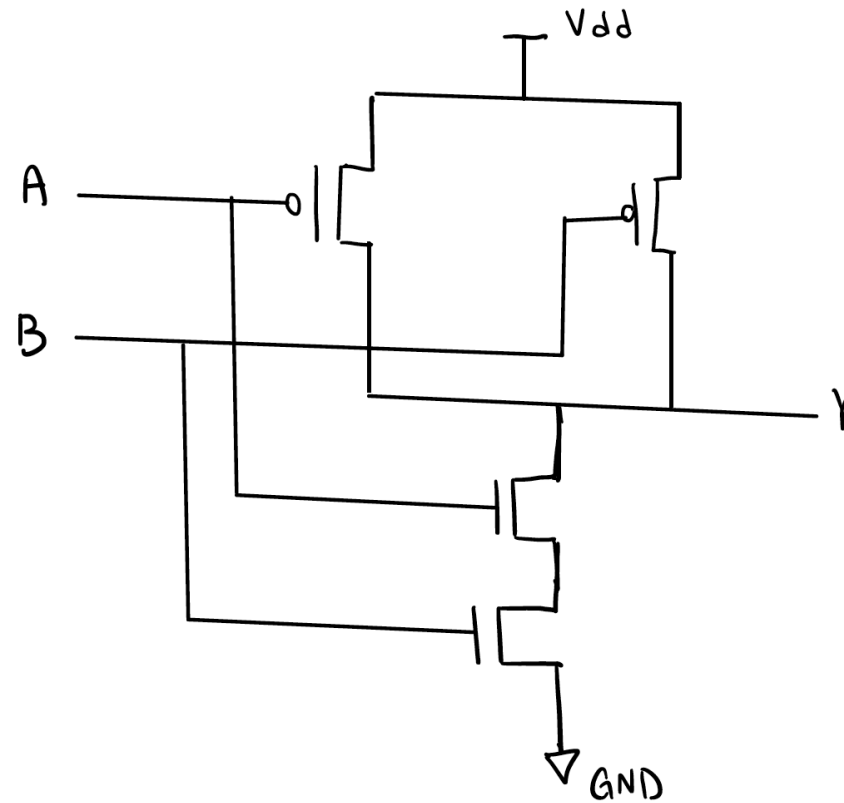


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

CMOS NAND



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

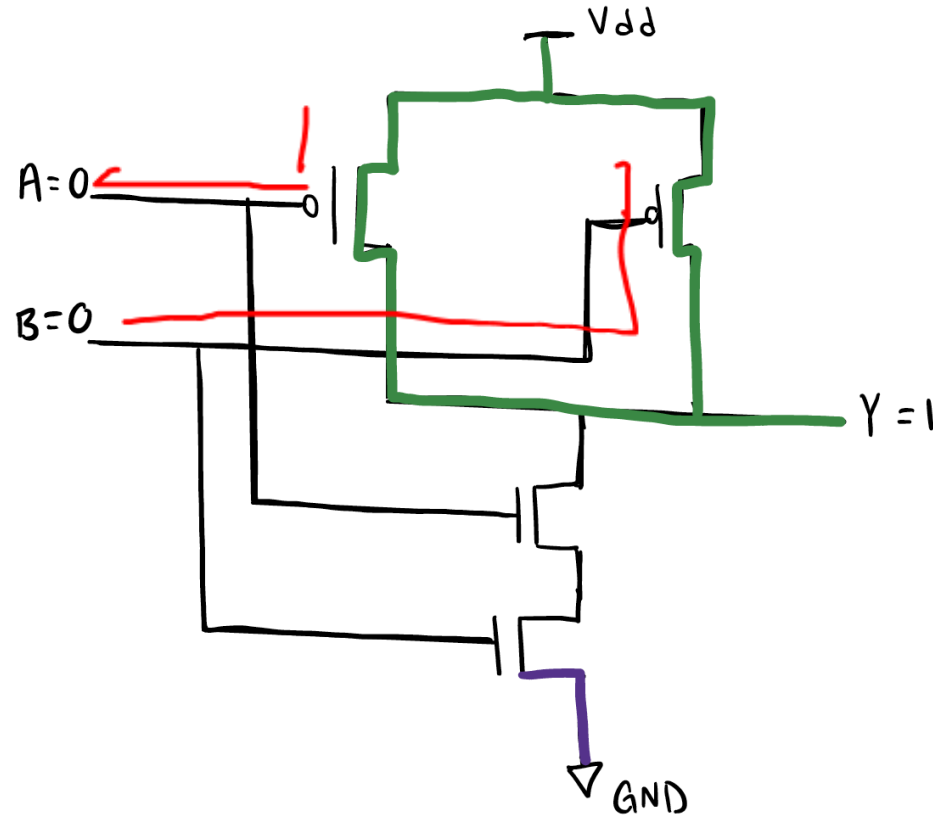


CMOS NAND



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

both pMOS on,
pull-up works

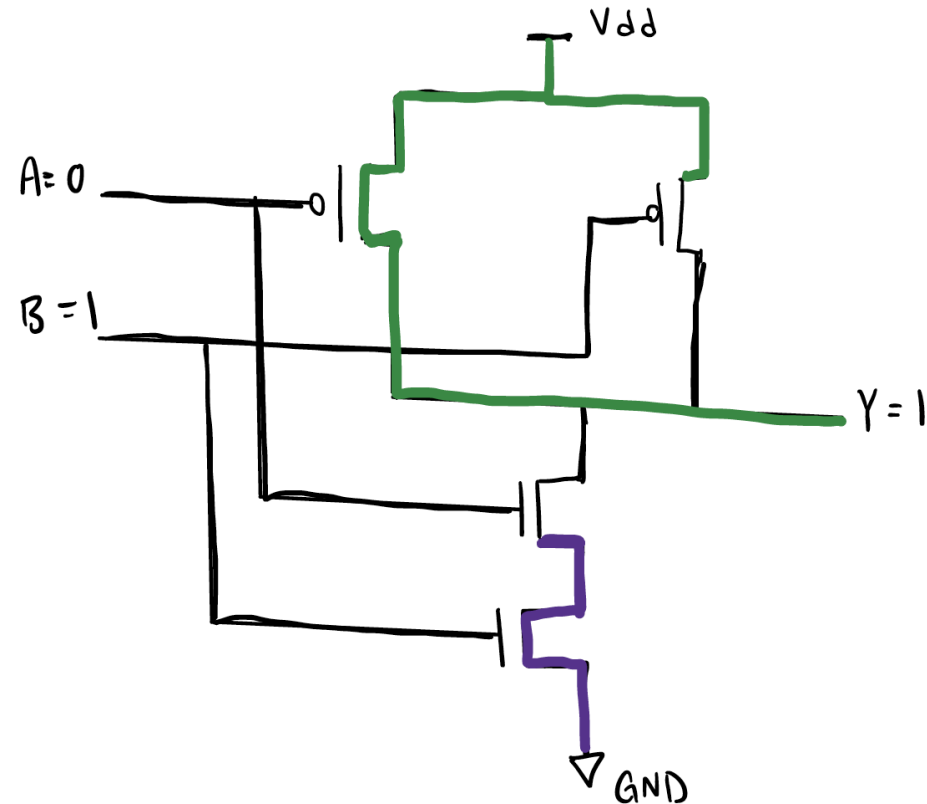


CMOS NAND



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

A's pMOS still
on, output
pulled high

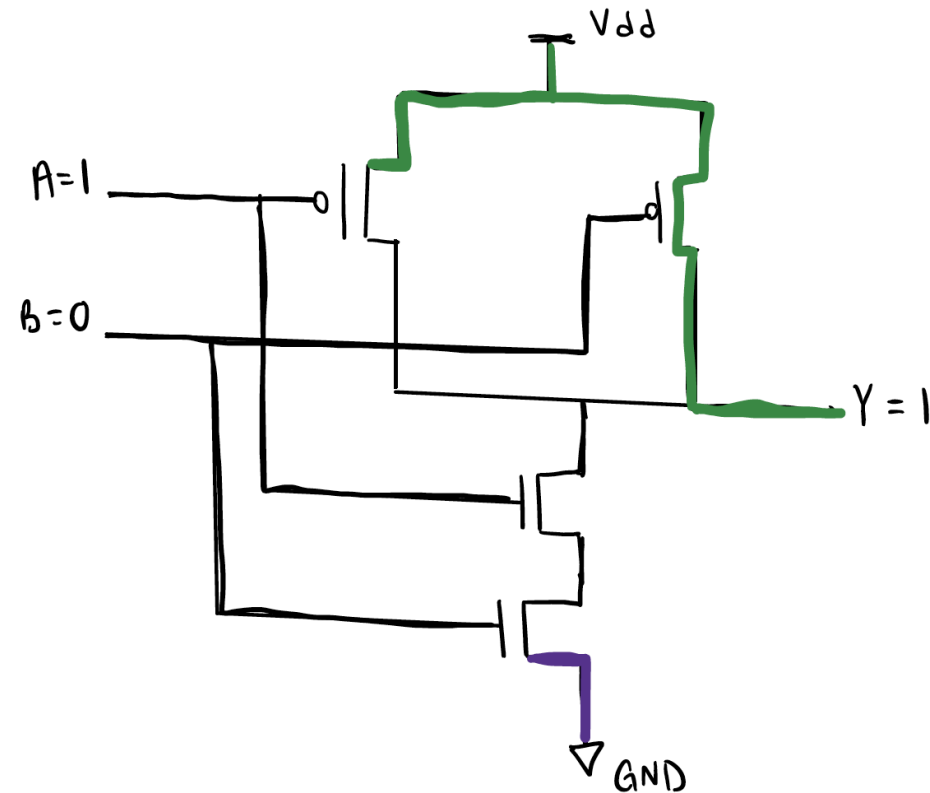


CMOS NAND



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

B's pMOS still
on, output
pulled high

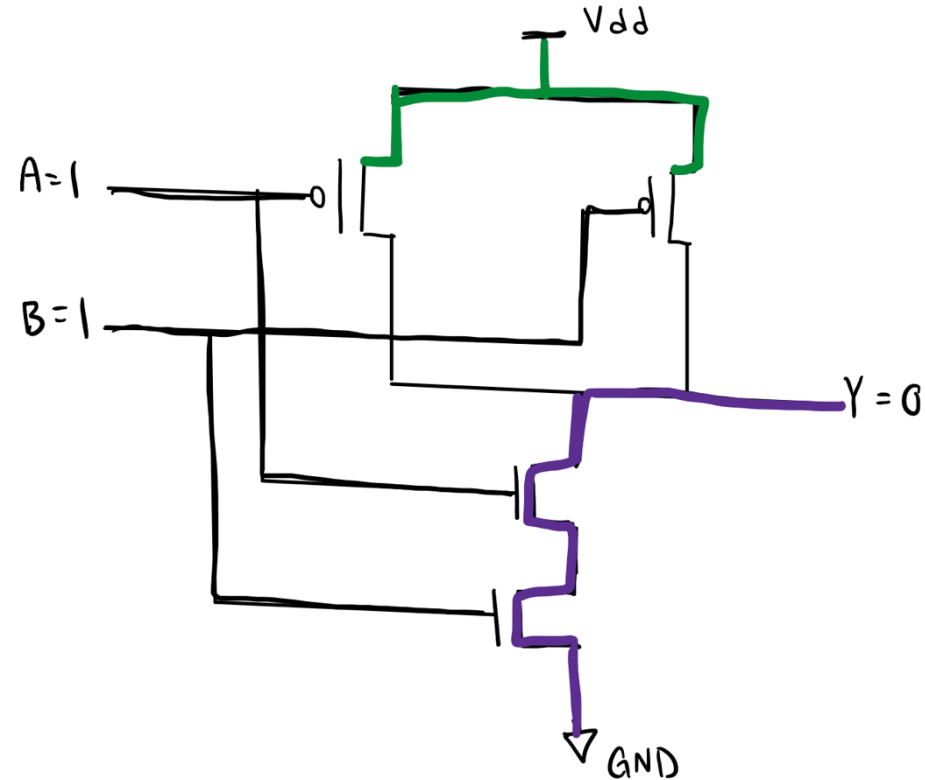


CMOS NAND



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

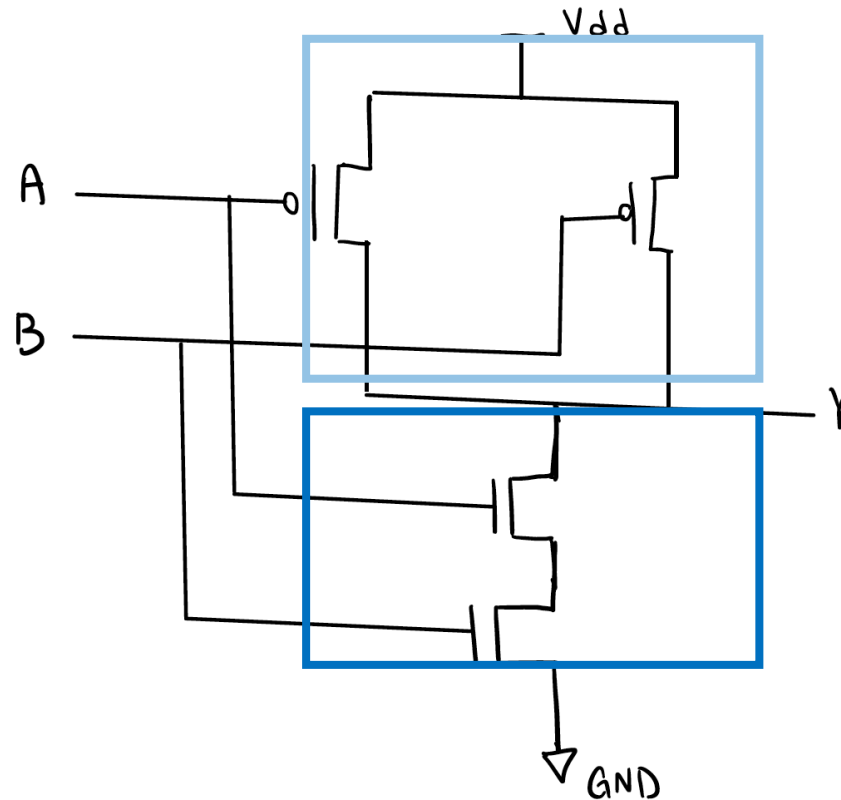
both pMOS off,
both nMOS on
→ path to
ground



CMOS NAND



A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



pull-up network

pull-down network



CMOS AND

CMOS AND

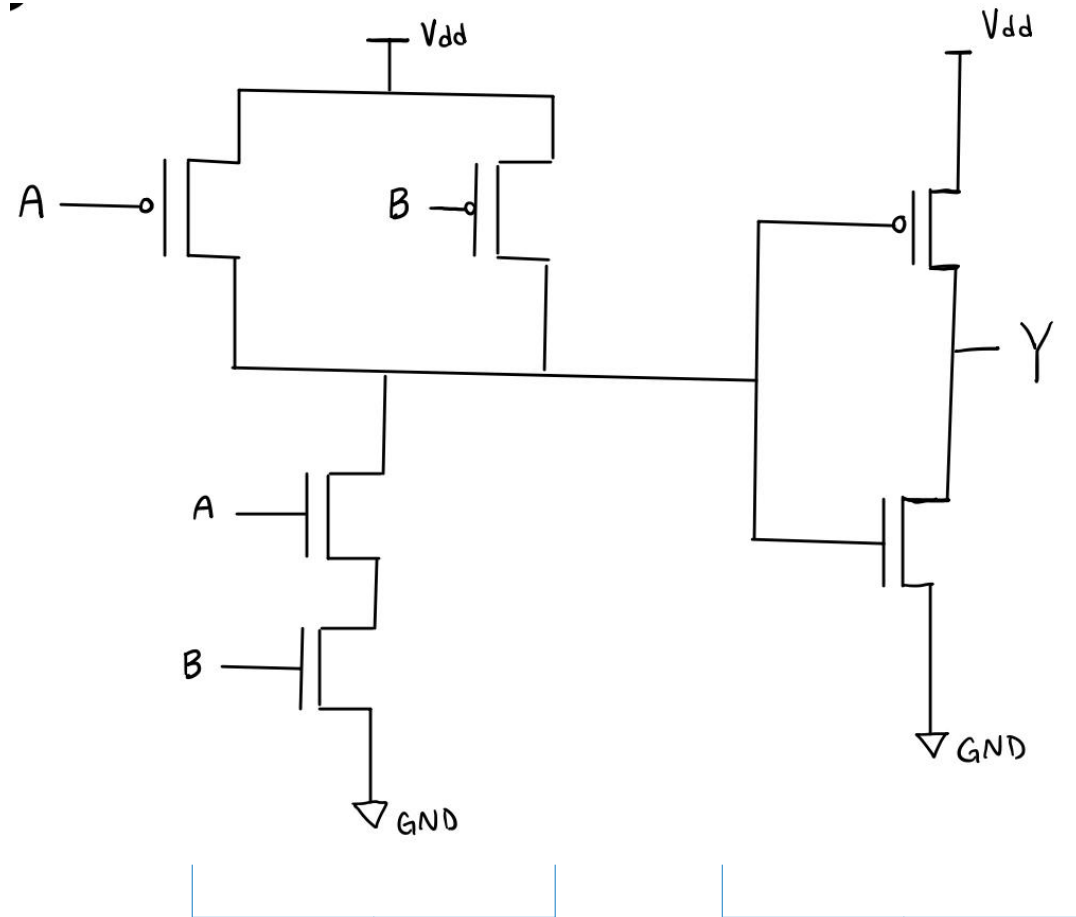


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

CMOS AND



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



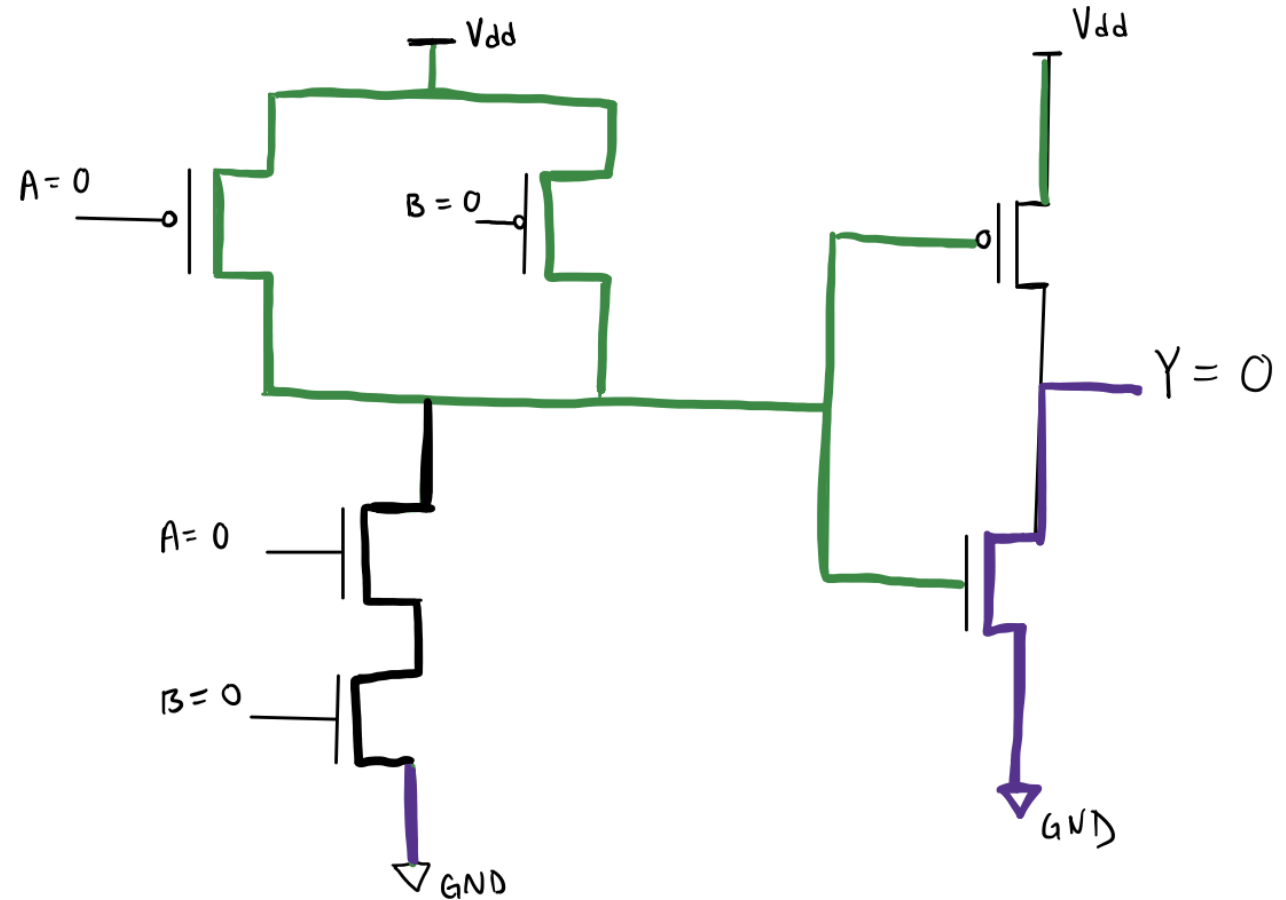
NAND

INVERTER

CMOS AND



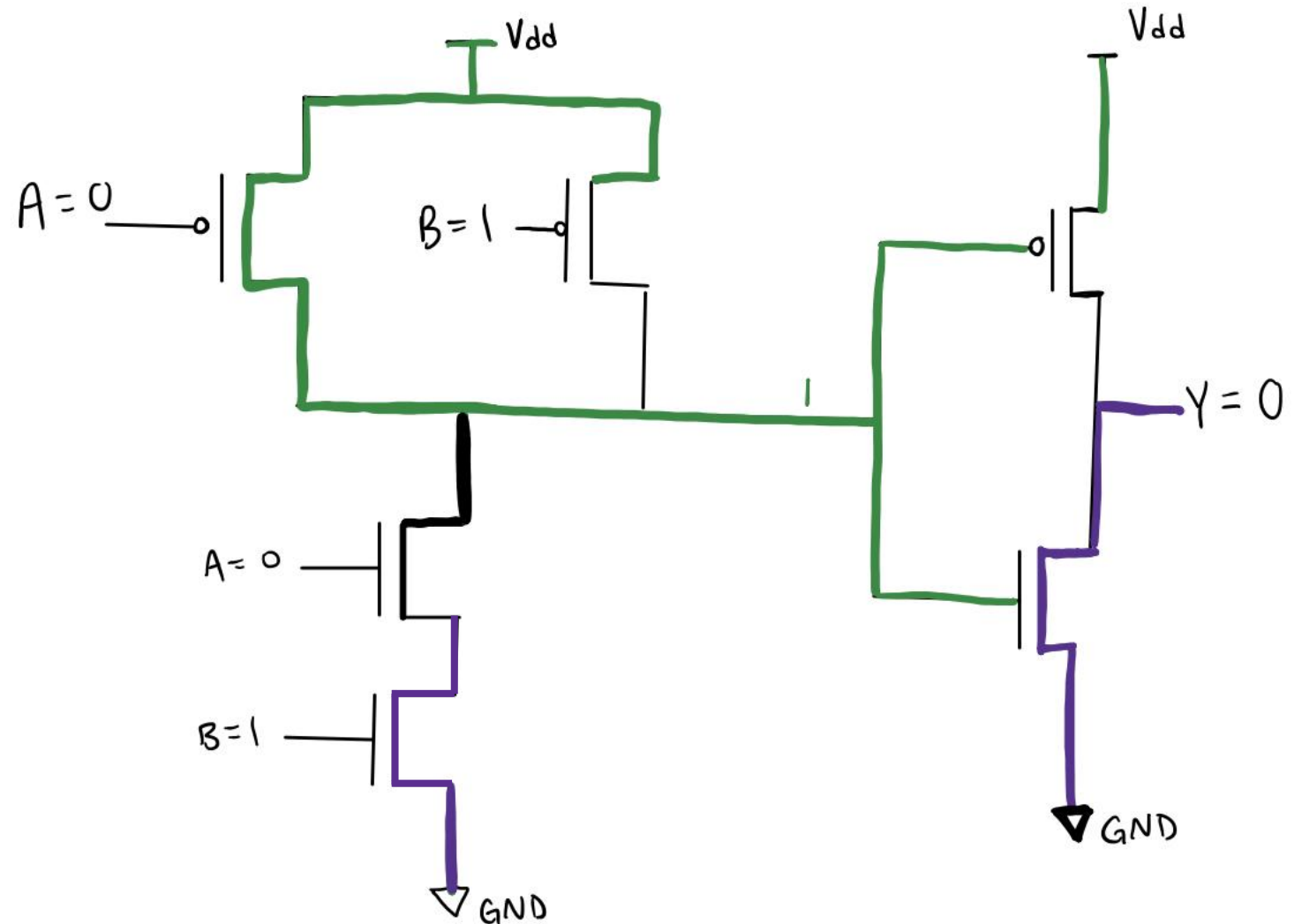
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



CMOS AND



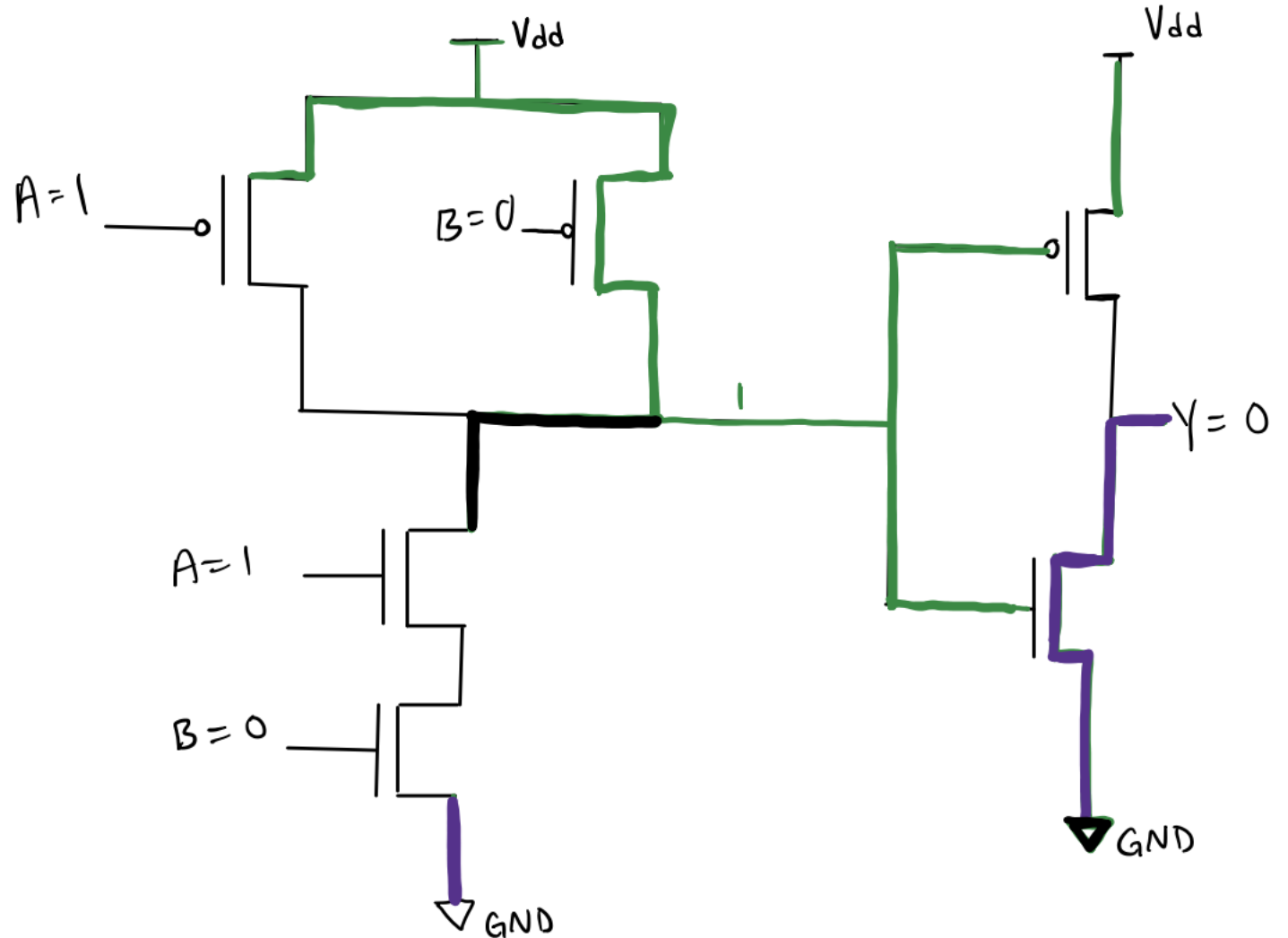
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



CMOS AND



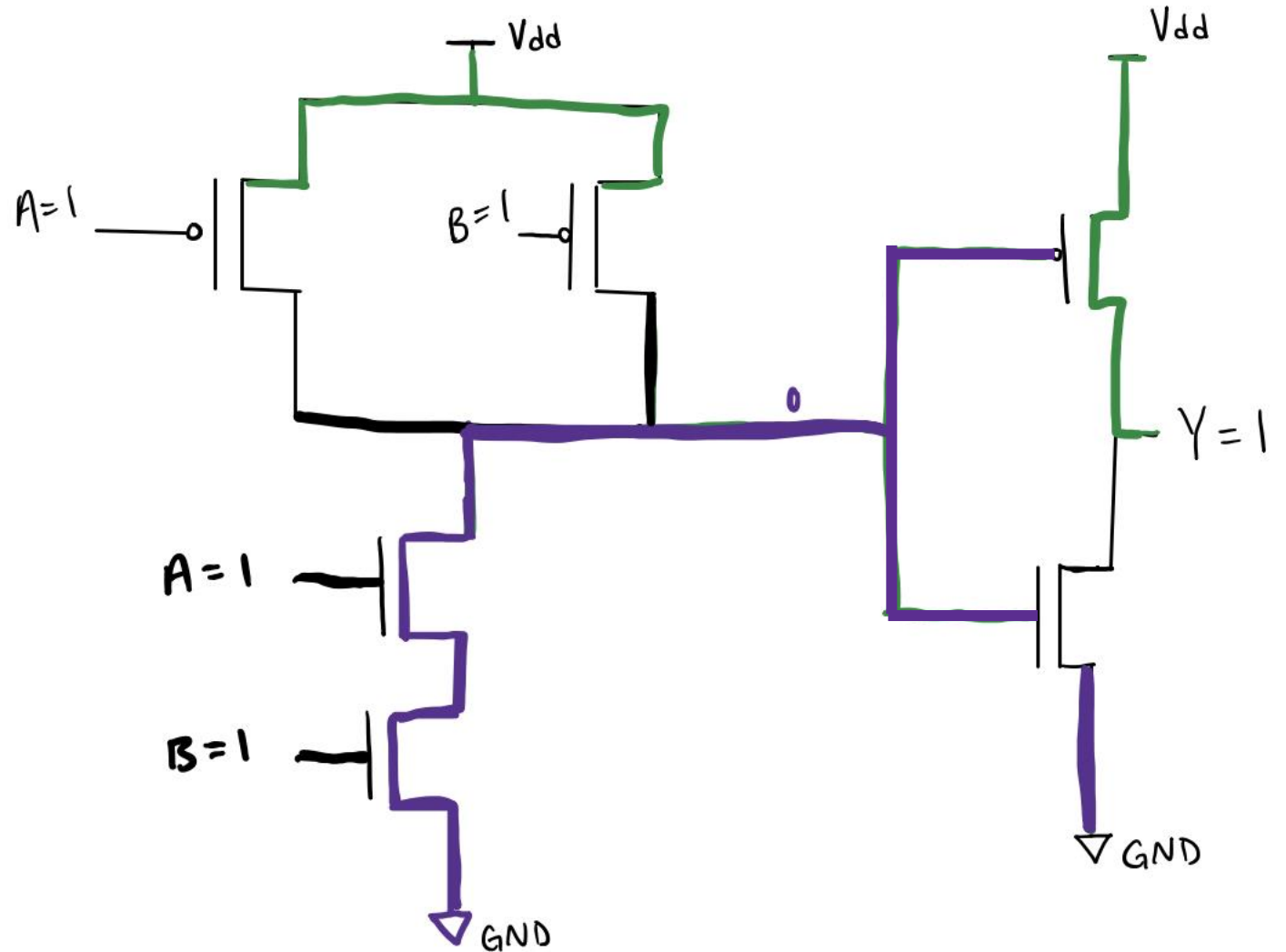
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



CMOS AND



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1





CMOS OR

CMOS OR

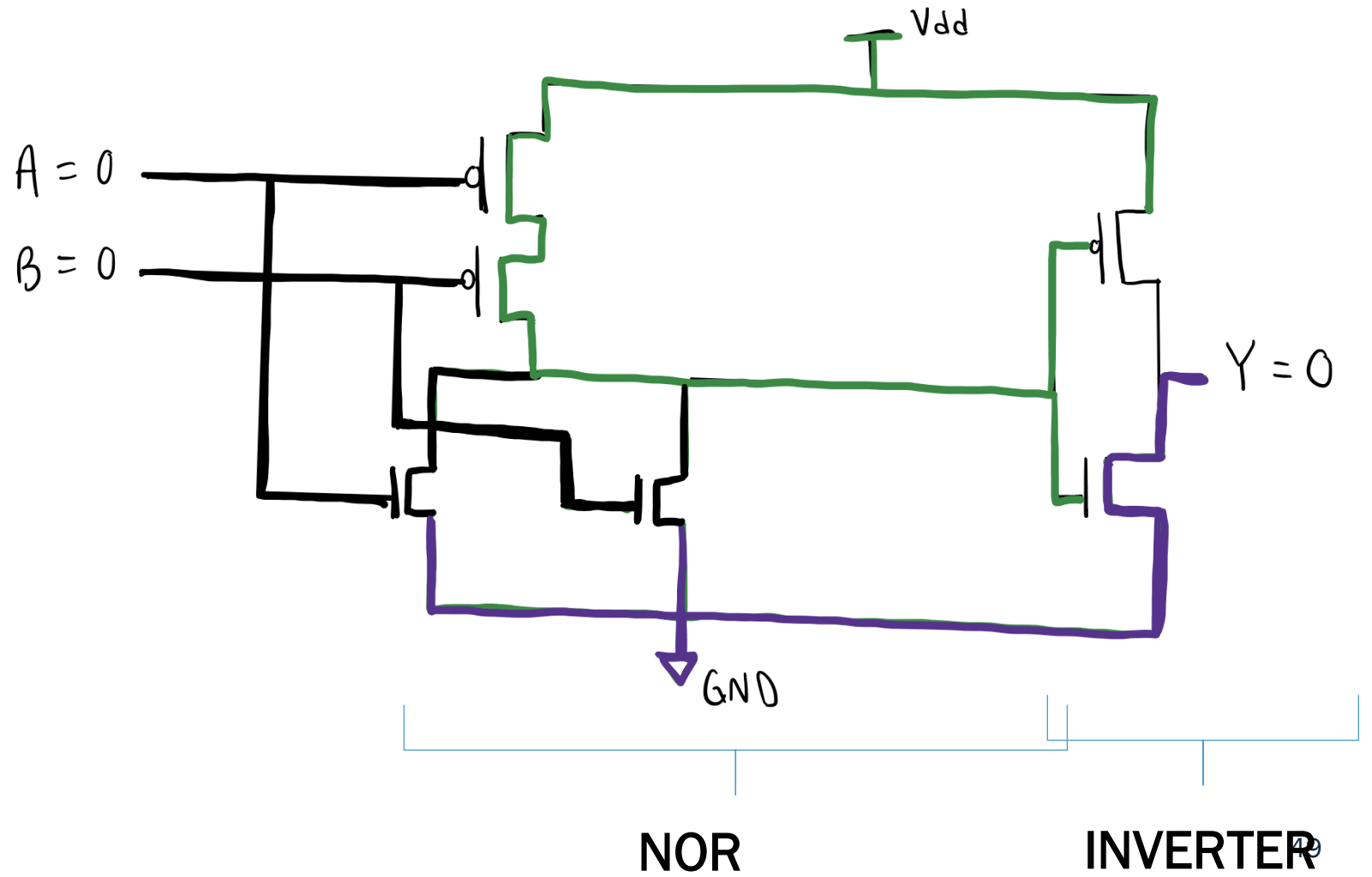


A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

CMOS OR



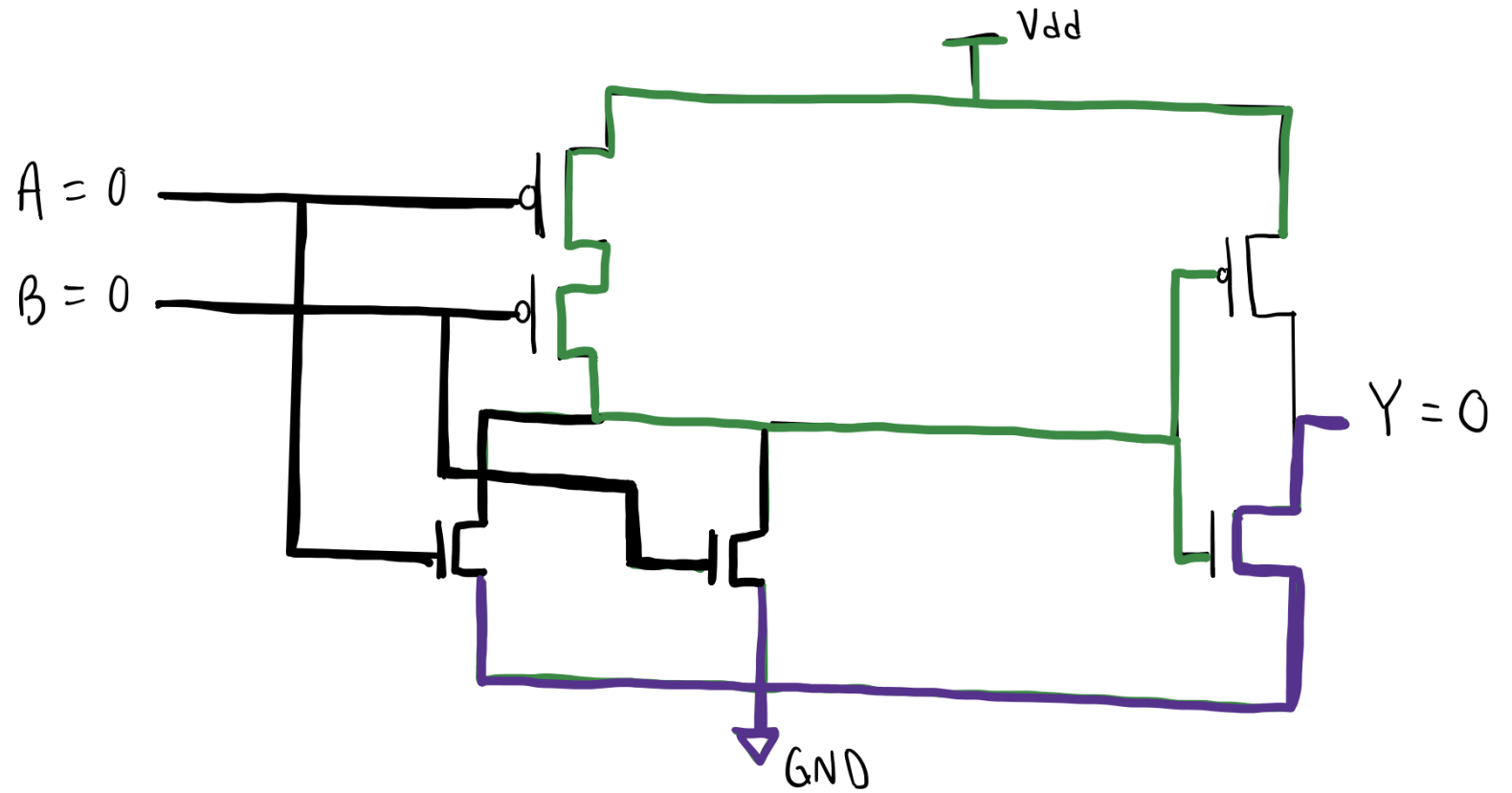
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



CMOS OR



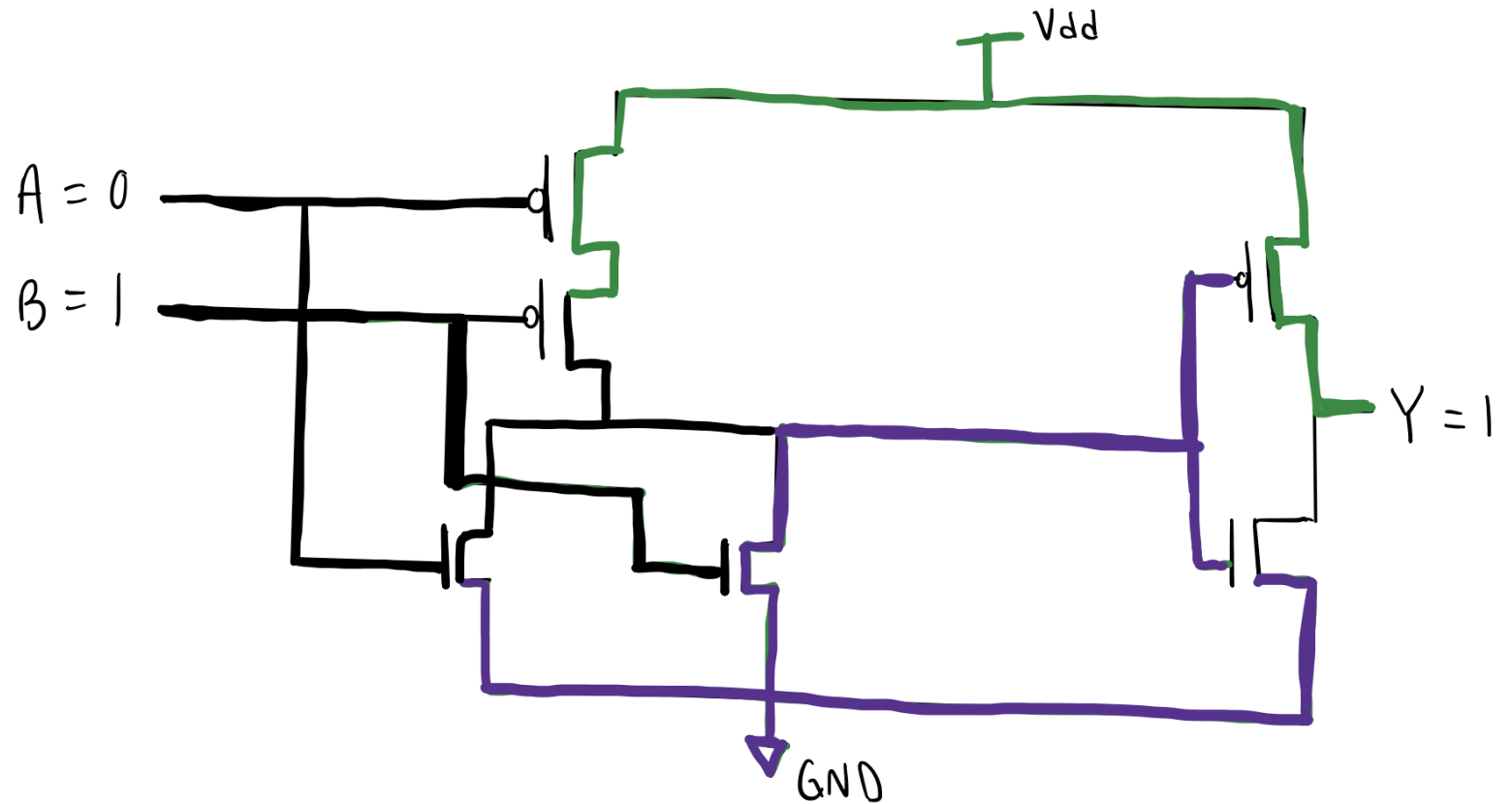
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



CMOS OR



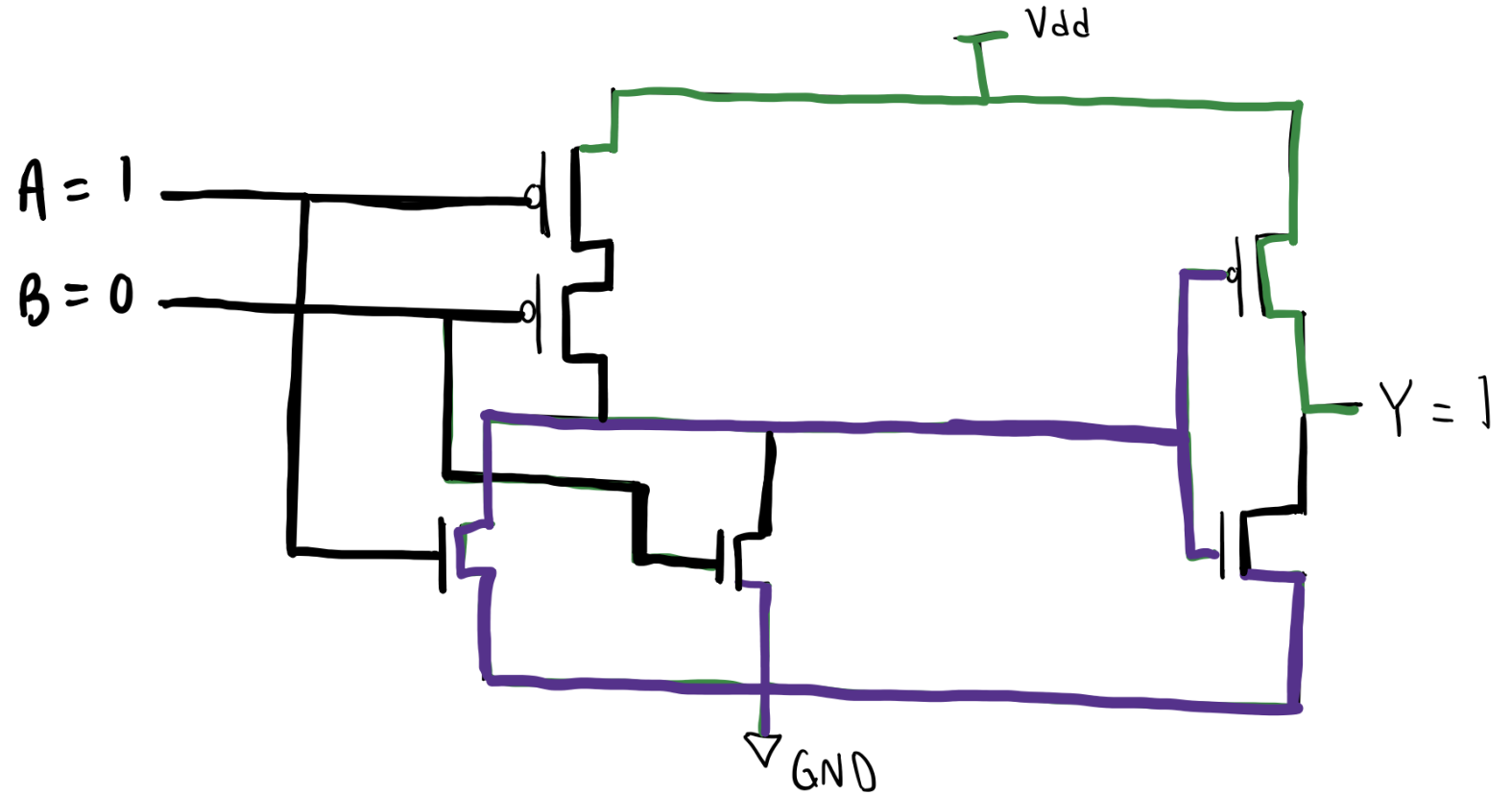
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



CMOS OR



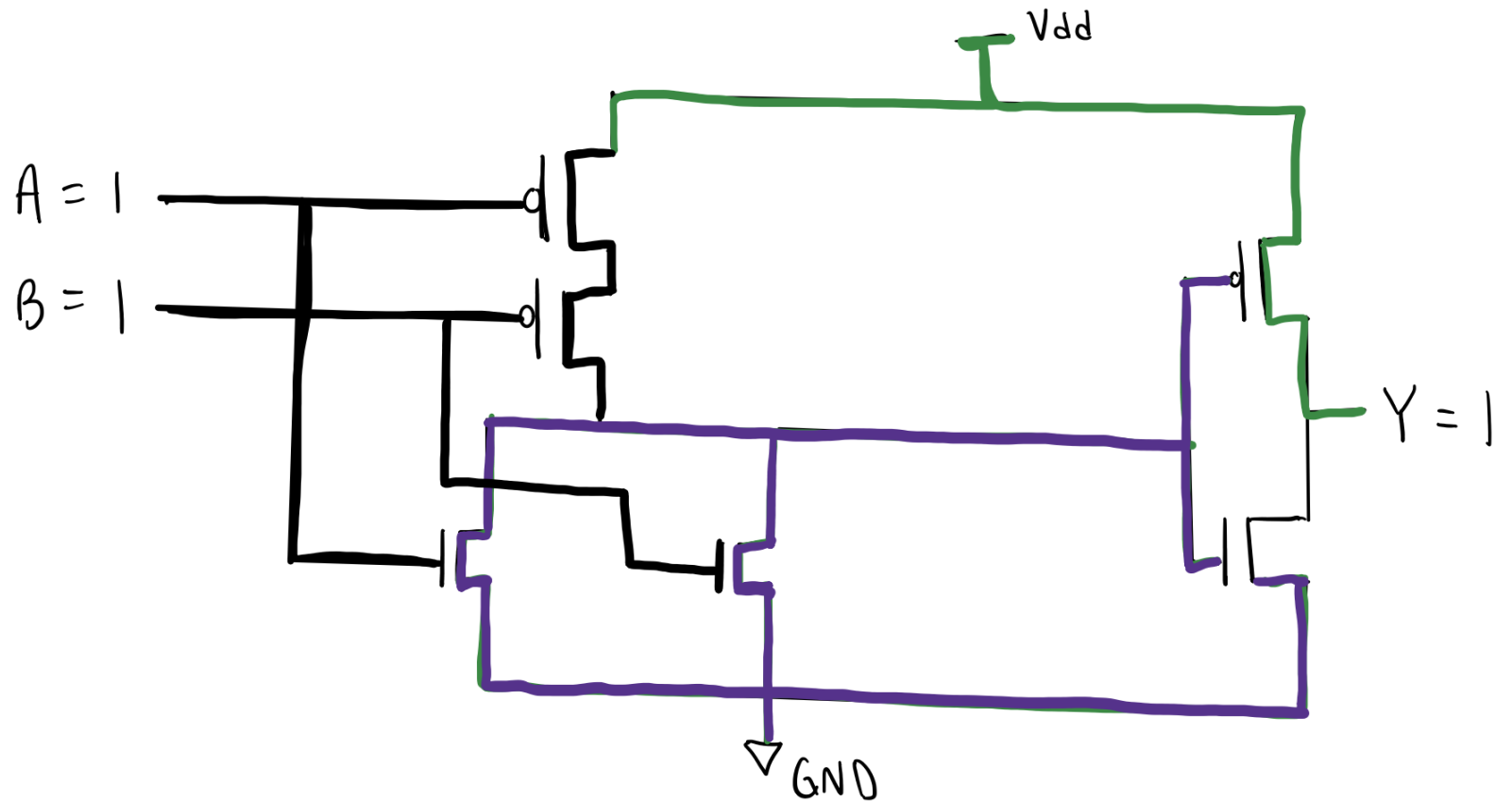
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



CMOS OR



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



Transistor Count



Gate		Number of Transistors
	NOT	2
	AND	6
	OR	6
↻	NAND	4
↻	NOR	4
	XOR	12
	XNOR	12



**Popping up a level of
abstraction! 😊**

Logic Gates!

Inverter/Not Gate



Symbol



Truth Table

A	Y
0	1
1	0

Equation

$$Y = \bar{A}$$

What logic gates can we build?



Recall, we need to design our gates using a pull-up network of pMOS transistors and a pull-down network of nMOS transistors

What gates can we
- build?
- define?

Let's start by
considering only
2-input gates.

AND		OR		NAND		NOR	
AB	Y	AB	Y	AB	Y	AB	Y
00	0	00	0	00	1	00	1
01	0	01	1	01	1	01	0
10	0	10	1	10	1	10	0
11	1	11	1	11	0	11	0

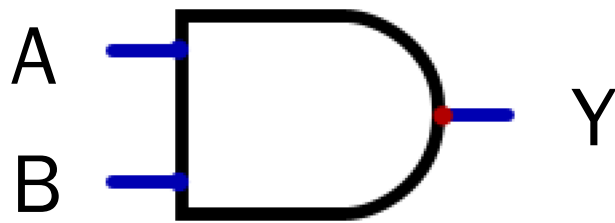
How many possible 2-input gates are there?

KEY IDEA: As many as there are 2-input truth tables.

AND Gate



Symbol



Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Equation

$$Y = A \times B$$

or
$$Y = AB$$

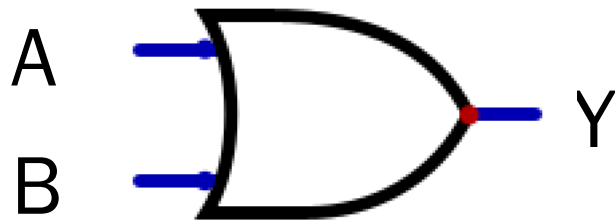
$$Y = A \cdot B$$

$$A * B$$

OR Gate



Symbol



Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

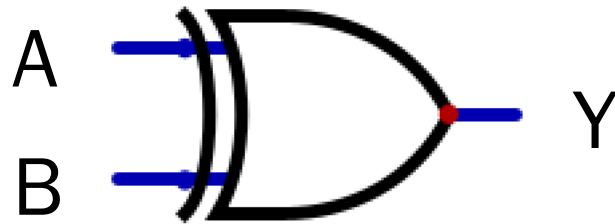
Equation

$$Y = A + B$$

XOR Gate



Symbol



Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

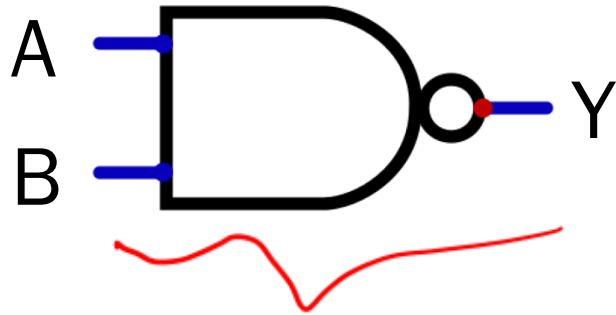
Equation

$$Y = A \oplus B$$

NAND



Symbol



Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Equation

$$Y = \overline{A \times B}$$

or

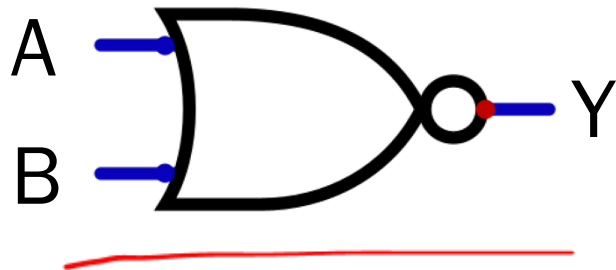
$$Y = \overline{AB}$$

$$Y = \overline{A \times B}$$

NOR



Symbol



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

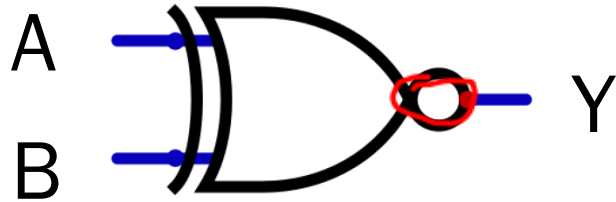
Equation

$$Y = \overline{A + B}$$

XNOR Gate



Symbol



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Equation

$$Y = \overline{A \oplus B}$$

All the gates!



There are only 16 possible 2-input gates... Let's examine all of them. Some we already know, others are just silly.

I N P U T AB	Z	E	A	A		B		X								
	R	N	>		>			O	O							
	O	D	B	A	A	B	R	R								
	O	D	B	A	A	B	R	R								
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	0	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

How many of these gates can be implemented using a single CMOS gate?



Do we really need all of these gates?

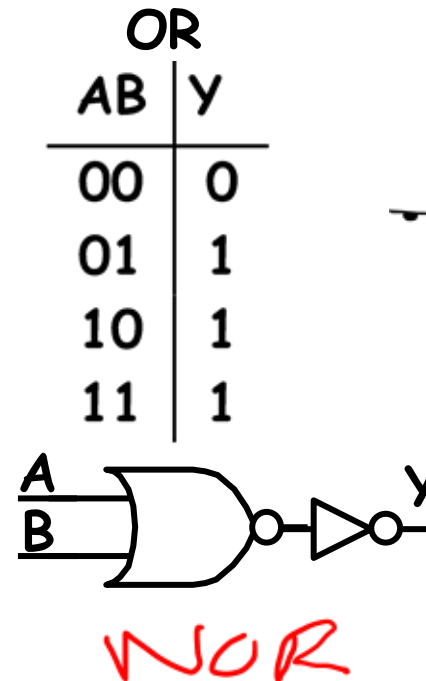
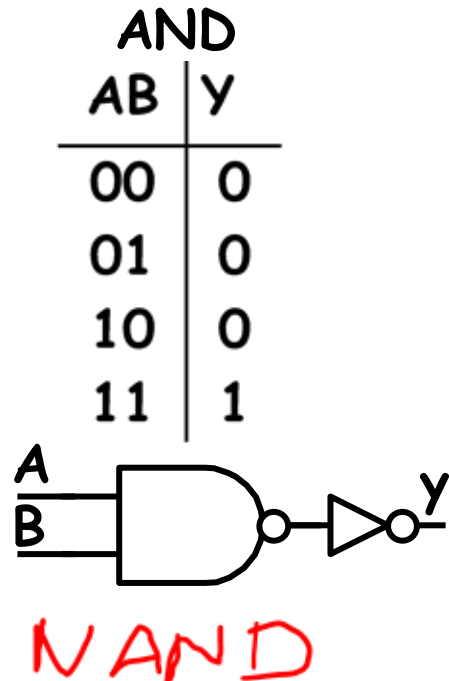
Nope! Once we realize that we can describe all of them using just AND, OR, and NOT

Composing gates: AND and OR



Each can be constructed using a pair of CMOS gates

AND is just NAND with an inverter, and OR is just NOR with an inverted output.



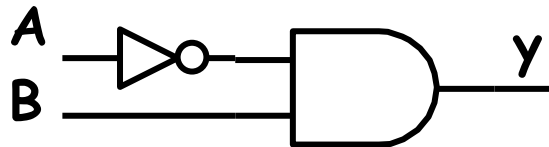
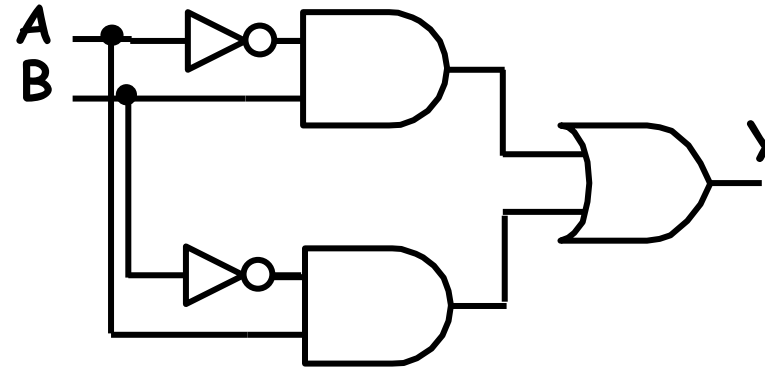
These two gates are particularly important. Using them will allow us to develop a systematic approach for constructing any combinational function.

Composing gates



B > A		
AB	Y	
00	0	
01	1	
10	0	
11	0	

XOR		
AB	Y	
00	0	
01	1	
10	1	
11	0	



How many different gates do we really need?

We can always do it with 3 different types of gates (AND, OR, INVERT), and sometimes with 2, but, can we use fewer?

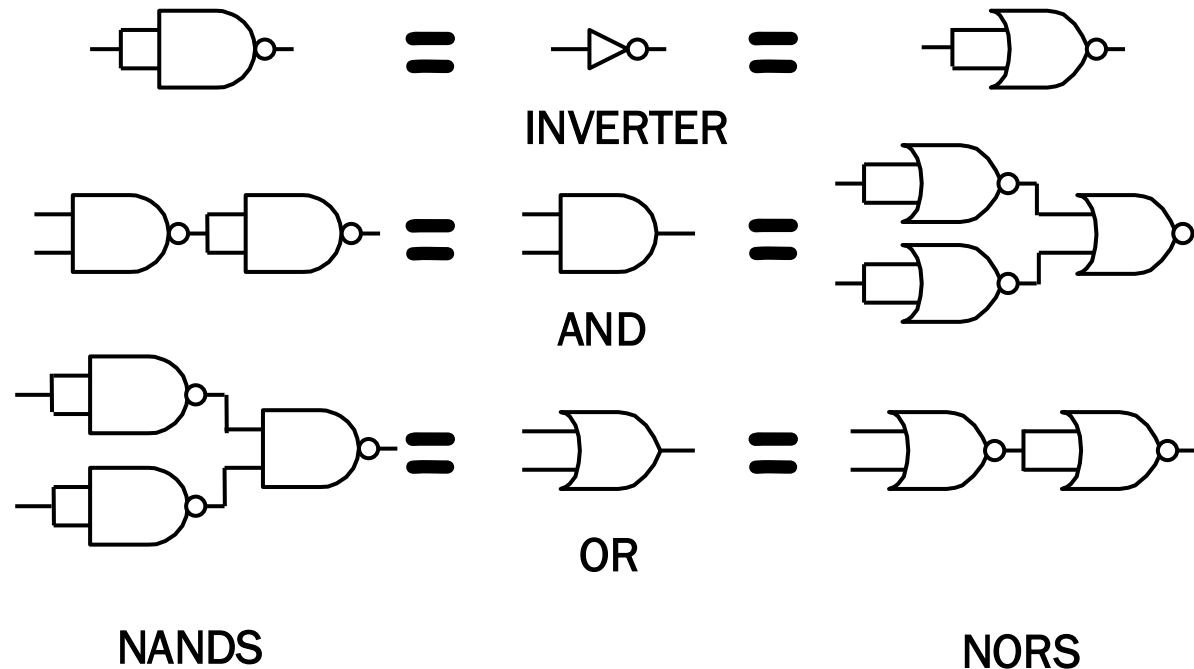
One gate will do!



A UNIVERSAL gate is one that can be used to implement *ANY* COMBINATIONAL FUNCTION. There are many UNIVERSAL gates, but not all gates are UNIVERSAL.



NANDs and NORs are UNIVERSAL!



Implementing XOR with AND, OR, NOT!



Truth Table

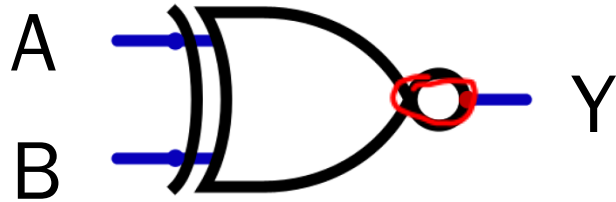
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Challenge question: convert this use ONLY NAND gates. How many NAND gates did you use?

Goal: Going between these 3 formats!



Symbol



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Equation

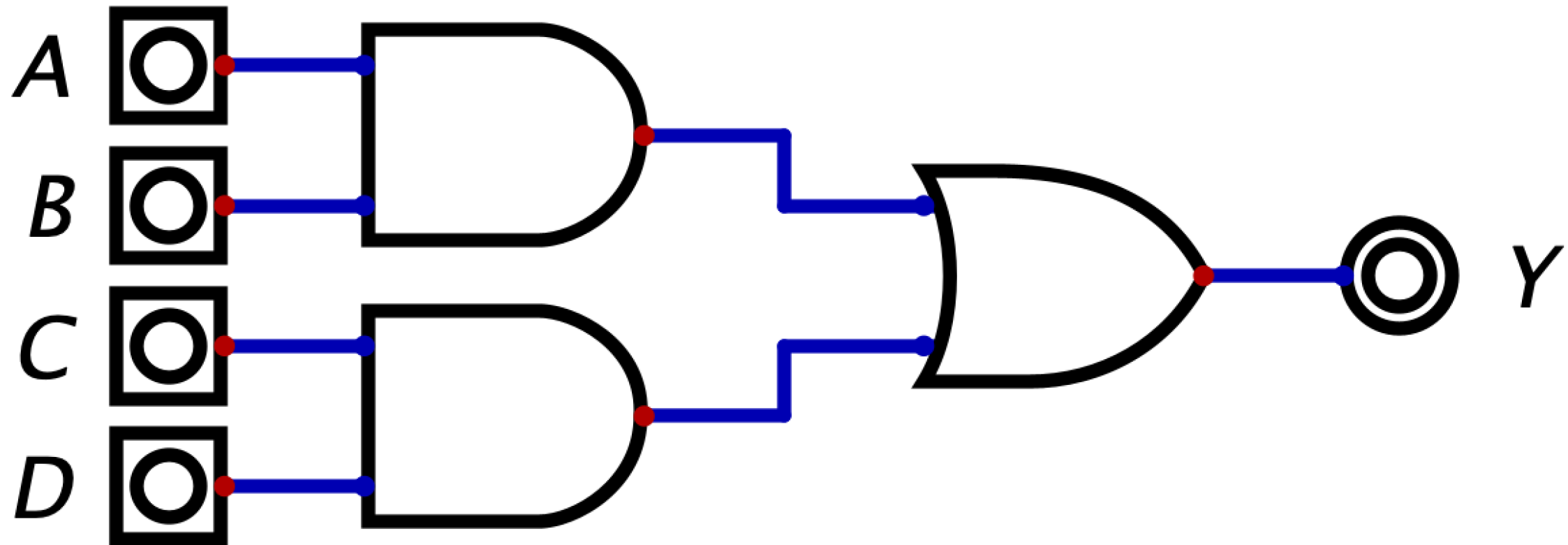
$$Y = \overline{A \oplus B}$$



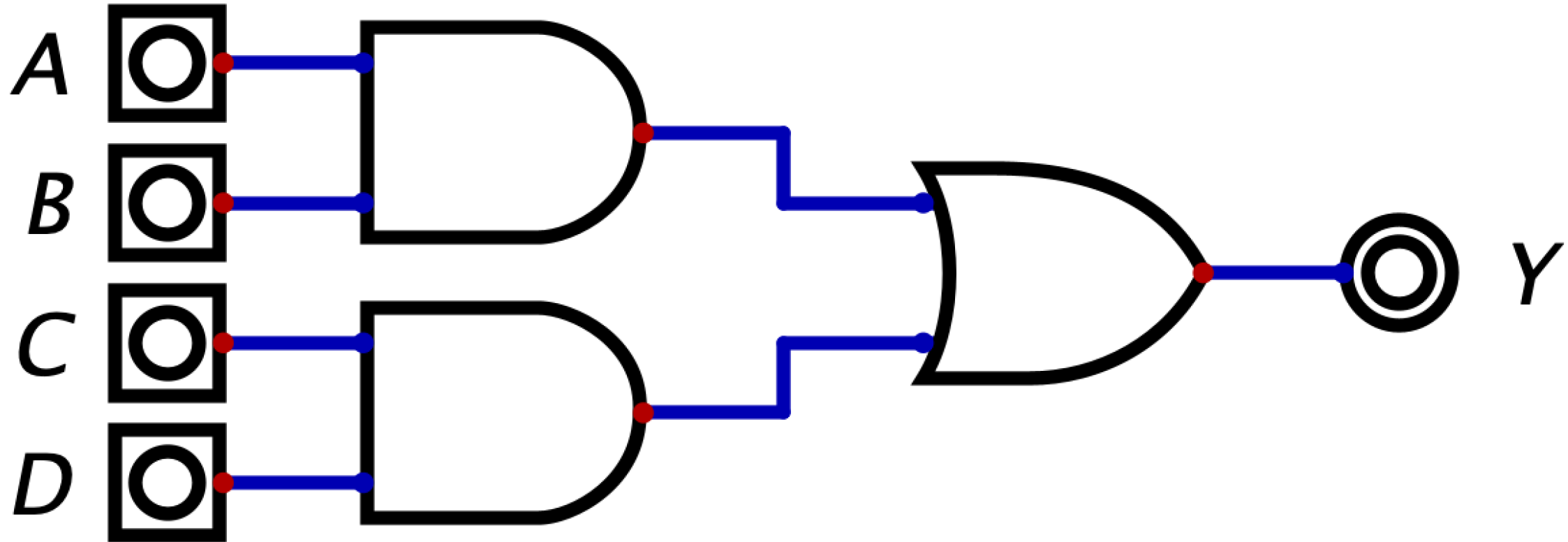


Forming *Equations* from *Logic Diagrams*

Ex1: Forming Equations from Logic Diagrams

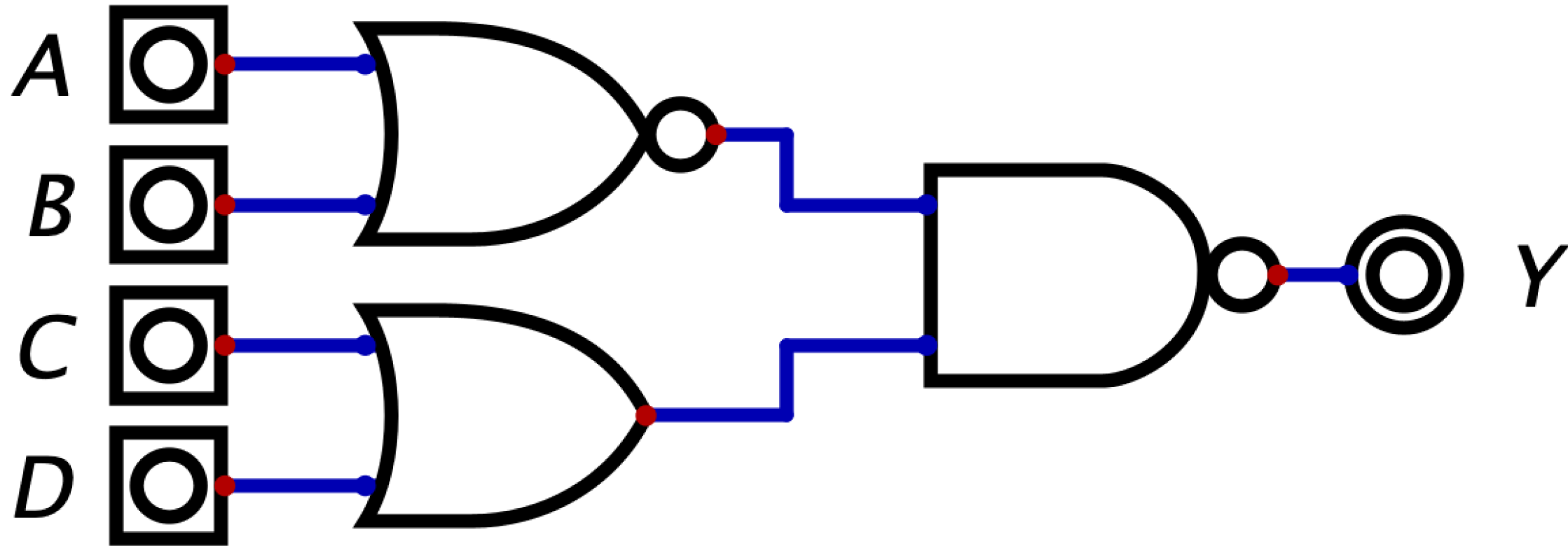


Ex1: Forming Equations from Logic Diagrams

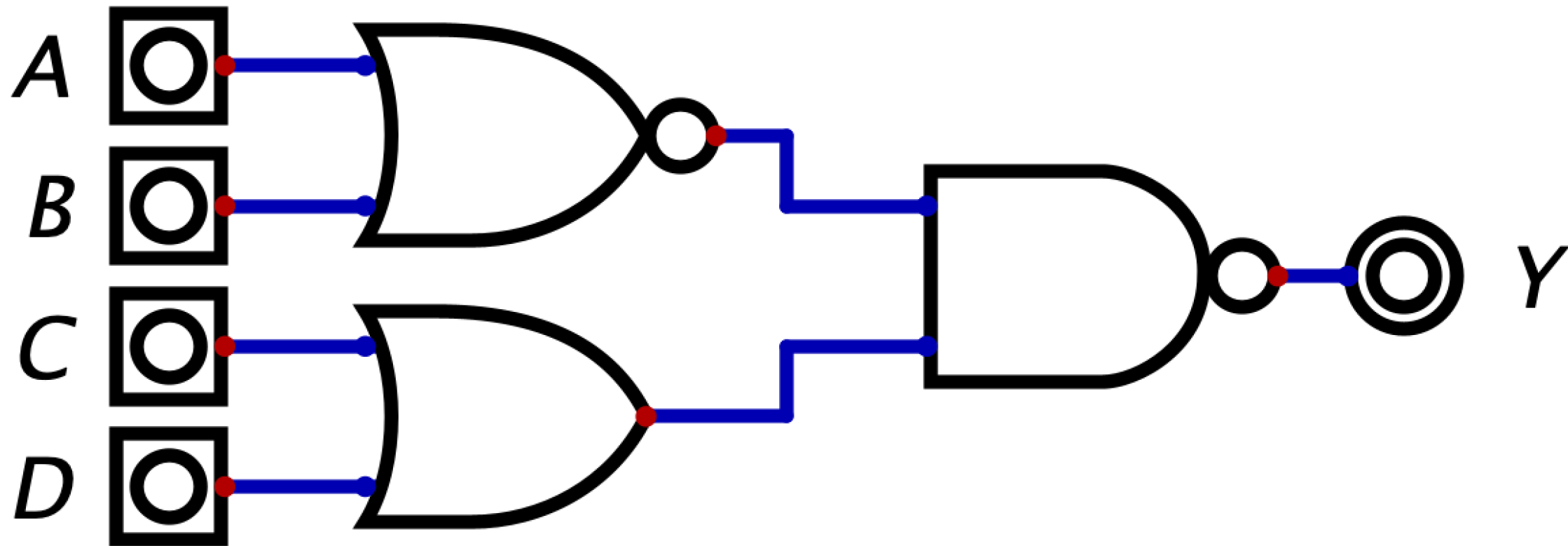


$$Y = AB + CD$$

Ex2: Forming Equations from Logic Diagrams

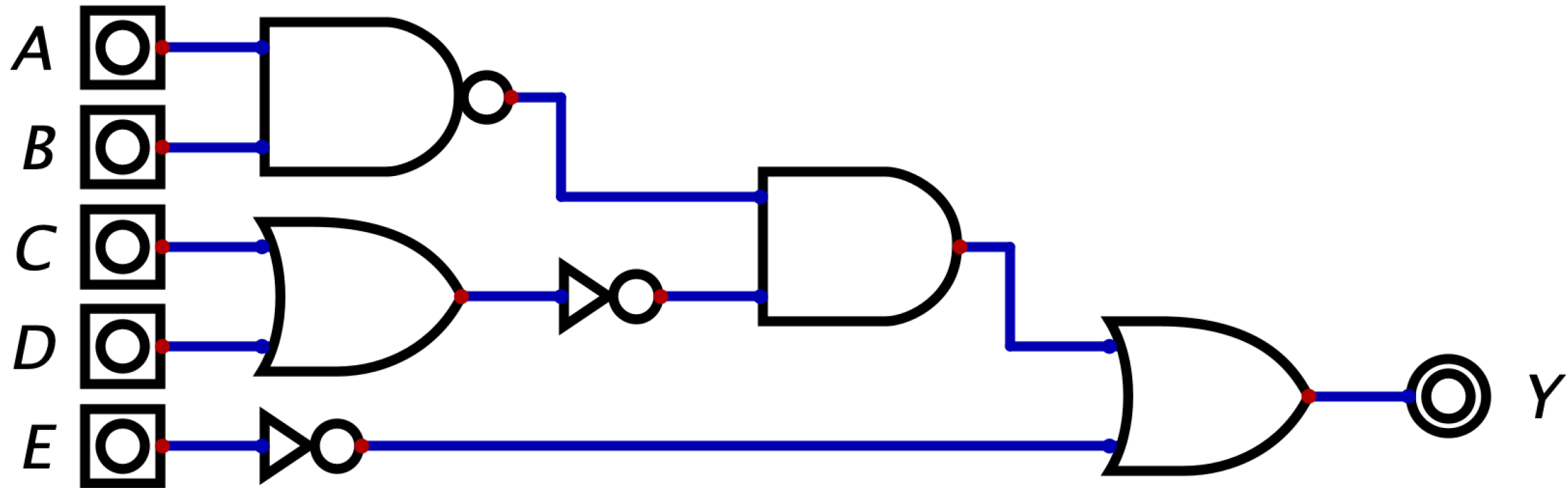


Ex2: Forming Equations from Logic Diagrams

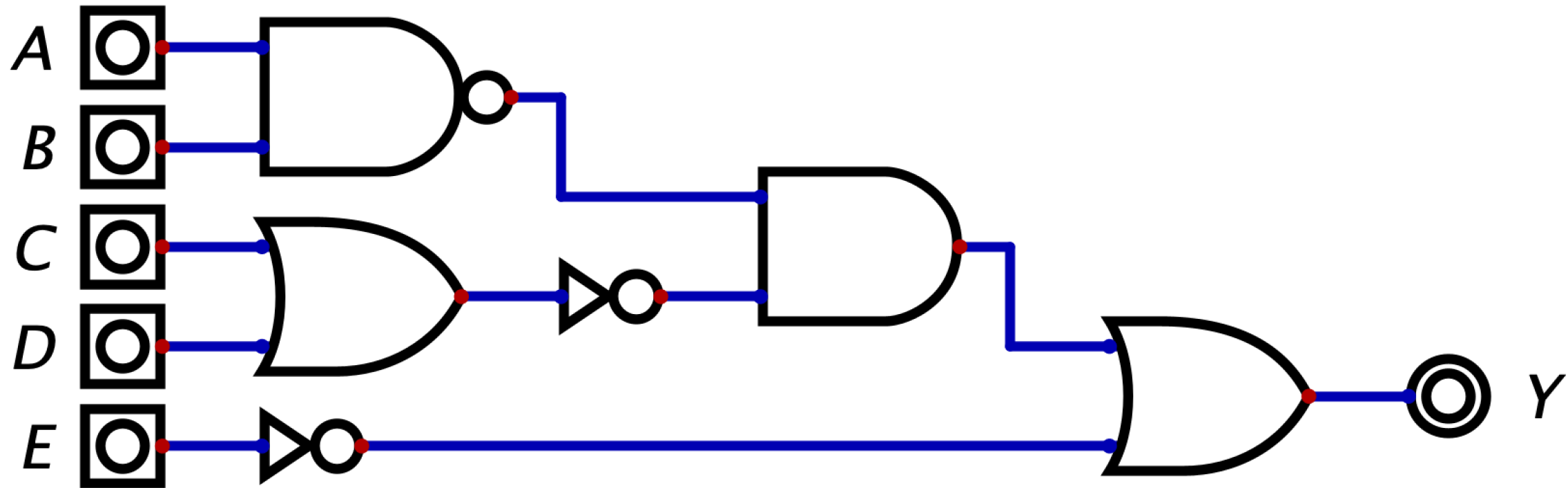


$$Y = \overline{(A + B)(C + D)}$$

Ex3: Forming Equations from Logic Diagrams

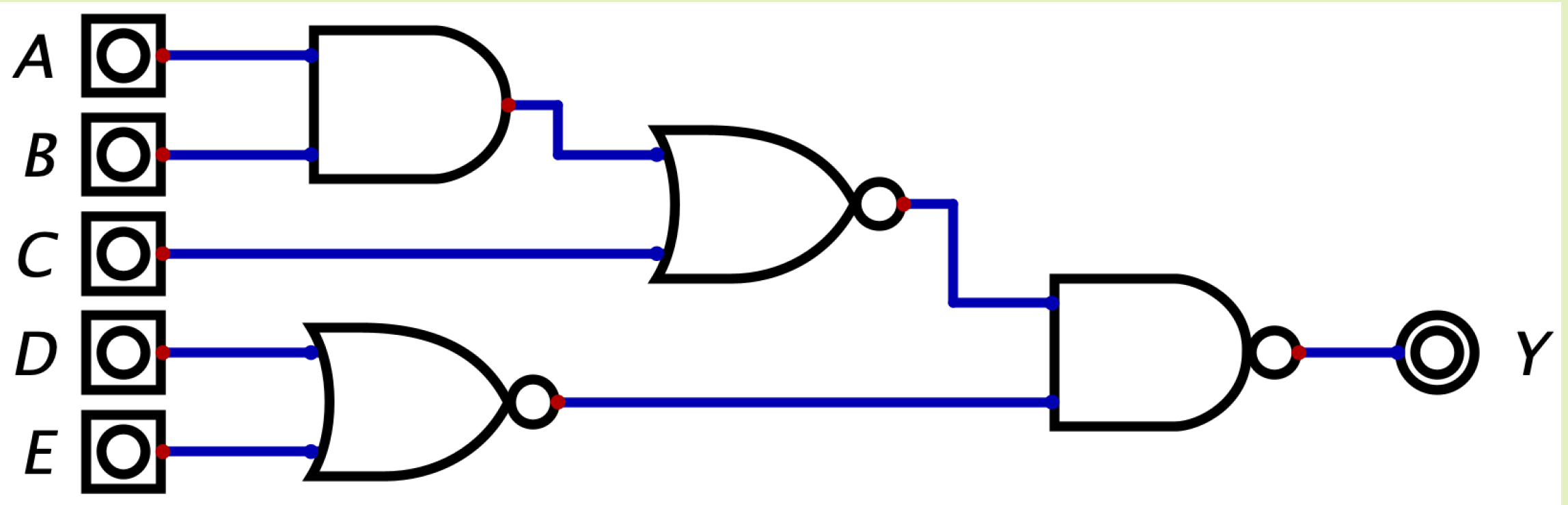


Ex3: Forming Equations from Logic Diagrams

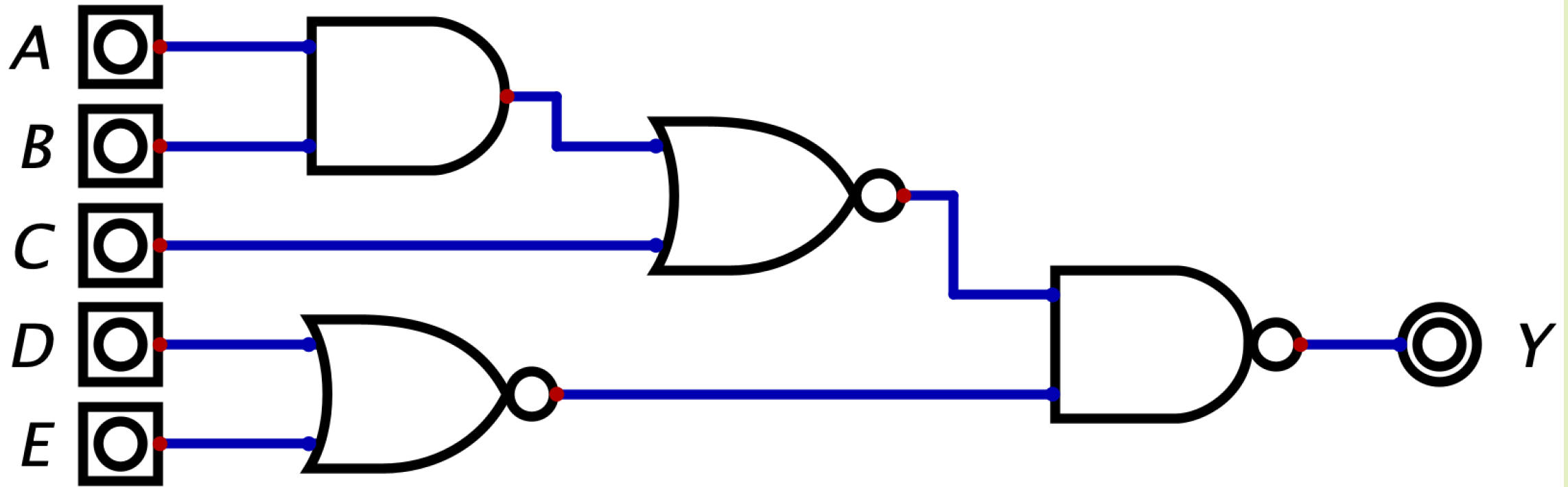


$$Y = (\overline{AB})(\overline{C + D}) + \overline{E}$$

EX4: Forming Equations from Logic Diagrams



Forming Equations from Logic Diagrams



$$Y = \overline{(\overline{AB + C})(\overline{D + E})}$$



Forming *Logic Diagrams* from *Equations*

Ex1: Forming Logic Diagrams from Equations

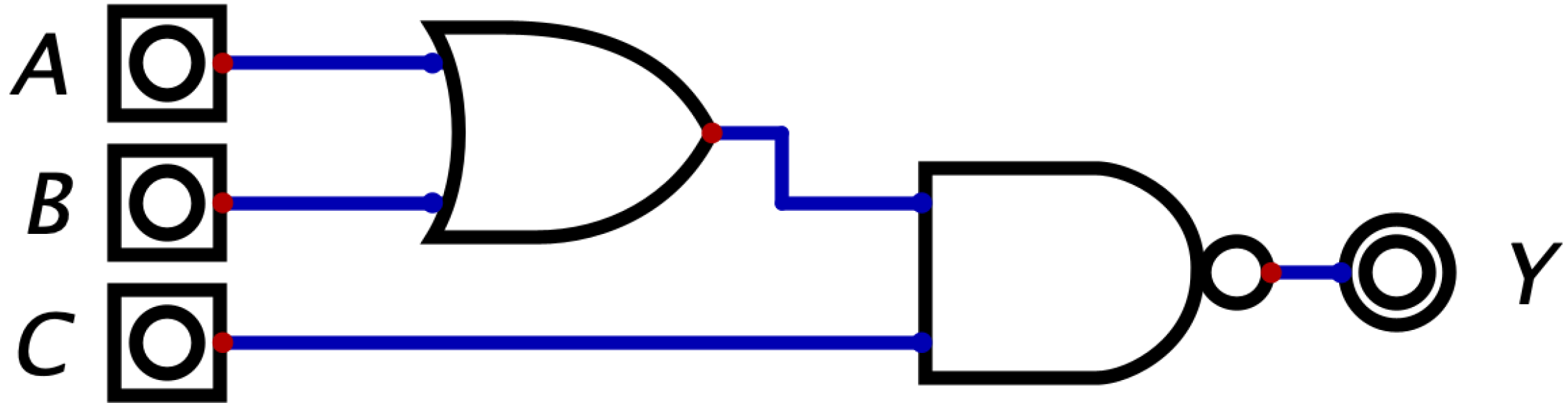


$$Y = \overline{(A + B)C}$$

Ex1: Forming Logic Diagrams from Equations



$$Y = \overline{(A + B)C}$$



Ex2: Forming Logic Diagrams from Equations

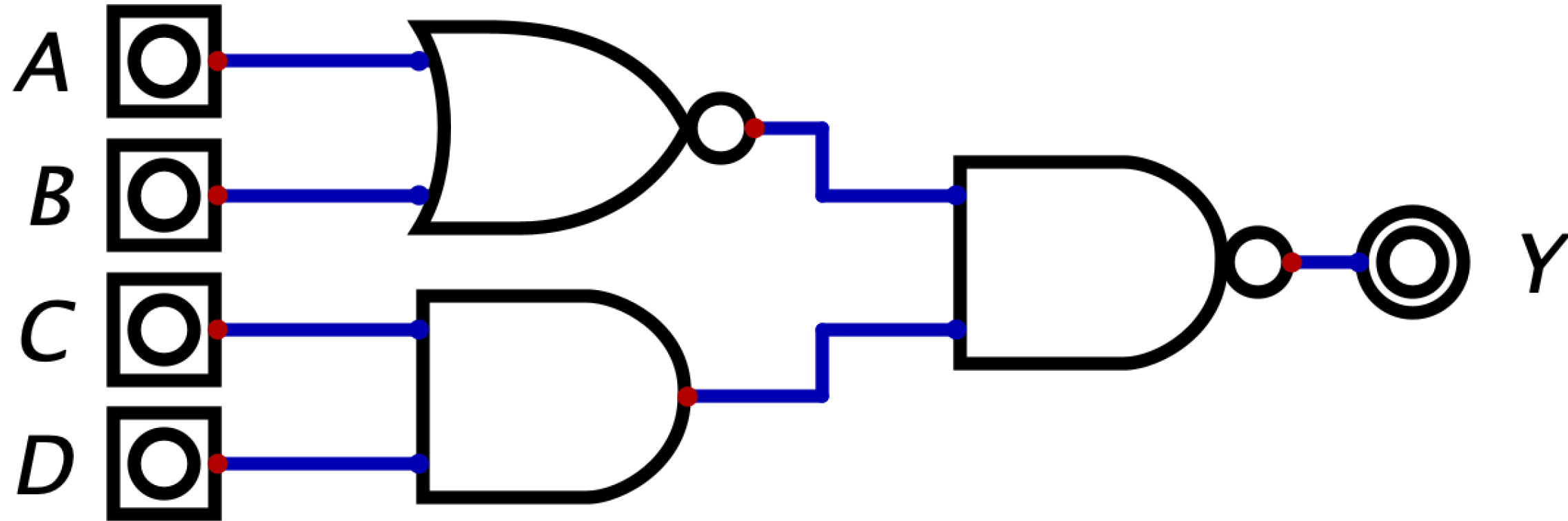


$$Y = \overline{\overline{(A + B)}CD}$$

Ex2: Forming Logic Diagrams from Equations



$$Y = \overline{\overline{(A + B)}CD}$$



Ex3: Forming Logic Diagrams from Equations

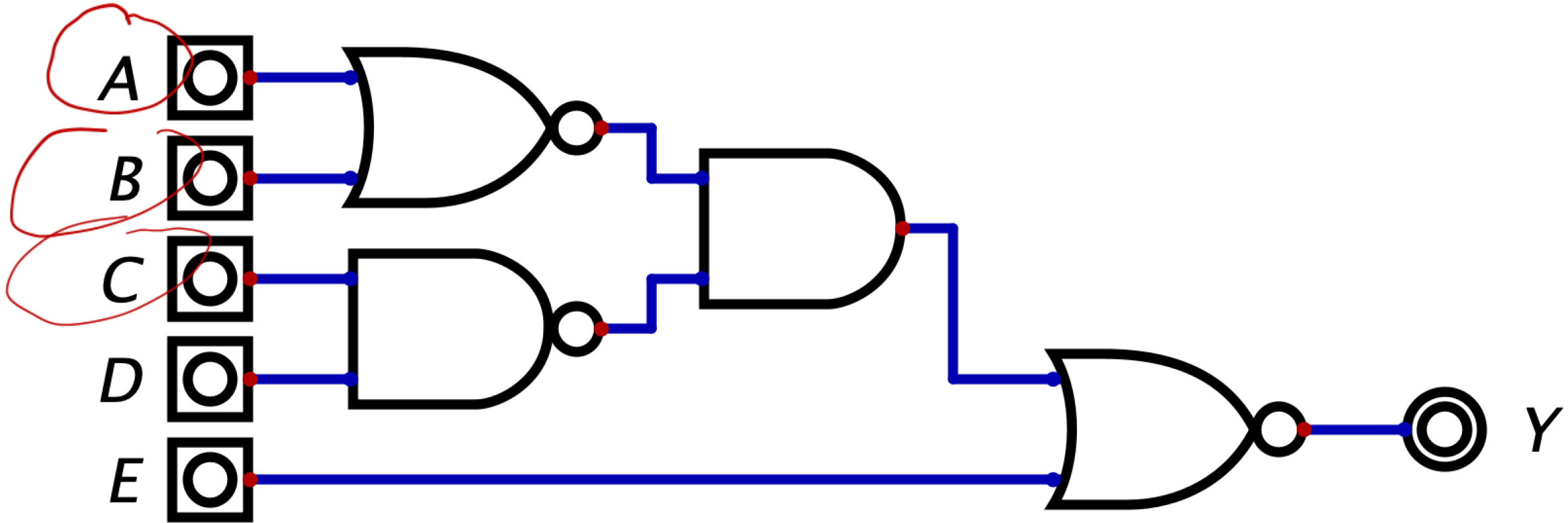


$$Y = \overline{(\overline{A + B})(\overline{CD}) + E}$$

Ex3: Forming Logic Diagrams from Equations



$$Y = \overline{(\overline{A + B})(\overline{CD})} + E$$



Forming Logic Diagrams from Equations



$$Y = \overline{(\bar{A})(\bar{B} + \bar{C}) + \bar{D}\bar{E}}$$

Forming Logic Diagrams from Equations



$$Y = \overline{(\bar{A})(\overline{B + C}) + \overline{DE}}$$

