CS61C

Great Ideas in Computer Architecture
(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Lisa Yan

UC Berkeley
Lecturer
Justin Yokota

Intro to Synchronous Digital Systems and Boolean Algebra

Kris Pister, Bora Nikolic, and Ali Niknejad have won the 2024 IEEE Solid-State Circuits Society Innovative Education Award in recognition of the **integrated circuit tapeout class** that they pioneered and polished over the past few years. This class has garnered worldwide attention from educators and industry practitioners alike, with other universities now mimicking its format.

UC Berkeley EECS Faculty
https://sscs.ieee.org/membership/awards/ieee-solid-state-circuits-society-innovative-education-award

Agenda

Synchronous Digital Systems

Yan, Yokota

Great Idea #1: Abstraction (Levels of Representation/Interpretation)

How do we design the hardware needed to execute machine code?

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

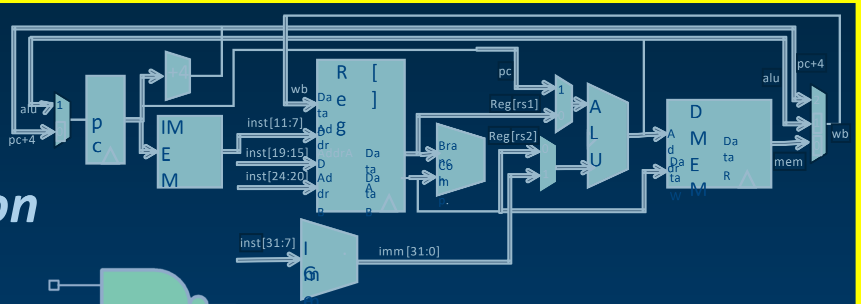
```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Assembler

Machine Language Program (RISC-V)

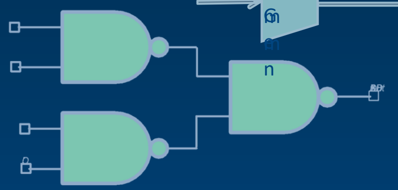
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Yan, Yokota

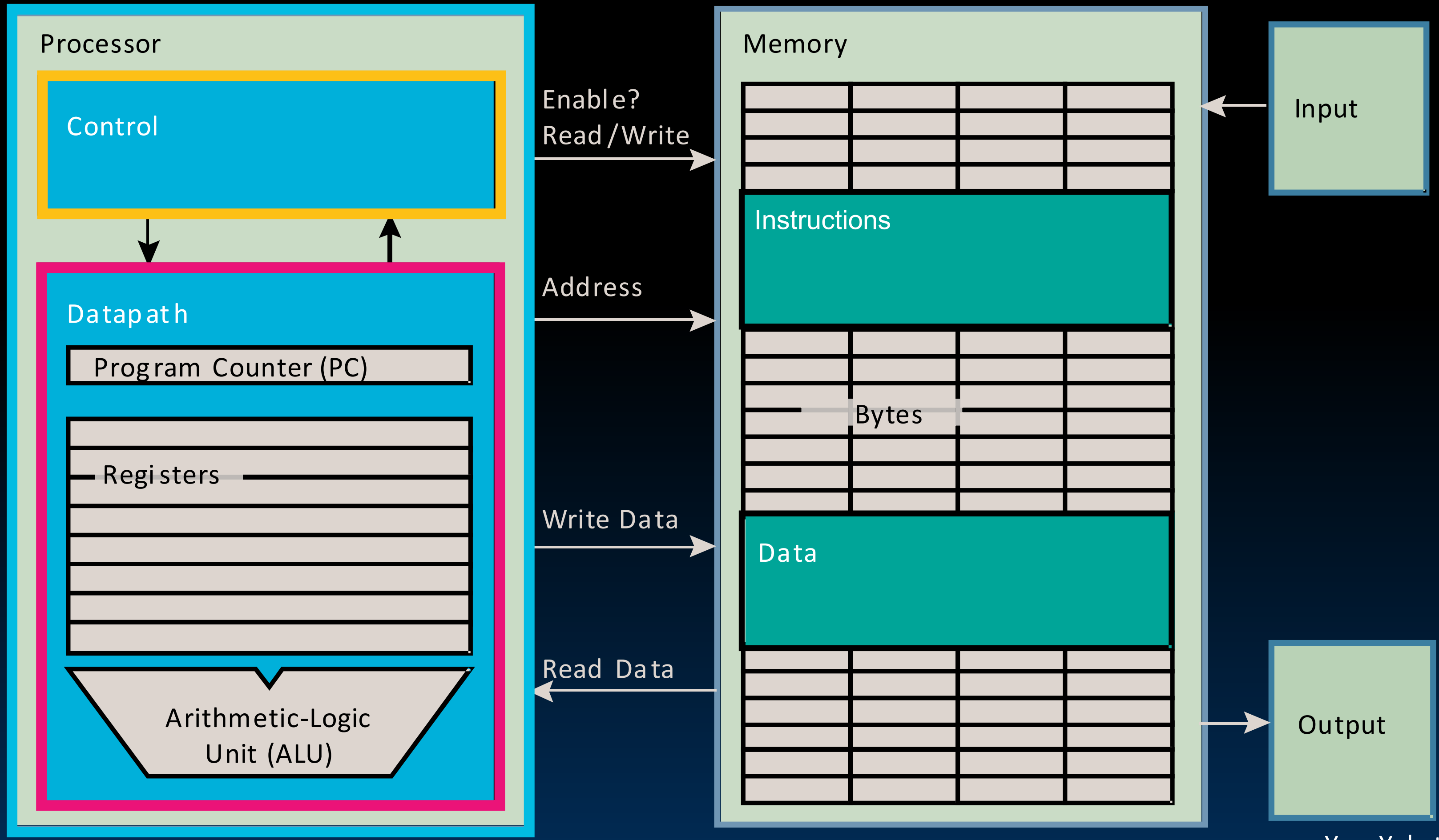
Hardware Design

- Over the next several weeks, we'll study how a modern processor is built, starting with definitions of the basic building blocks.
- Why study **hardware design**? Go beyond "just programming."
  - To really understand how computers work, we need to understand the complete stack, including the physical level.
  - Understand capabilities and limitations of HW in general, and processors in particular.
    - *Why is my computer so slow? Why does my battery run down?*
    - There is only so much you can do with standard processors!
    - For extra performance, you may need to design your own custom HW.
  - Prepare for more in depth HW courses (EECS 151, CS 152).
  - Get a Job: Apple is a traditional HW company. Even traditional SW companies (Google, Amazon, Meta) do their own hardware design!

The principles we teach now will likely still apply in 30 years, even if/when base technology changes! ☺

How do we build a Single-Core Processor?

Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making).



Yan, Yokota

Synchronous Digital System (SDS)

- The hardware underlying almost every processor is a **Synchronous Digital System**.
- **Synchronous**: All operations coordinated by a central **clock**.
  - "Heartbeat" of the system!
  - (By contrast, *asynchronous* systems must locally coordinate actions and communications b/t components; *much* harder to design/debug.)

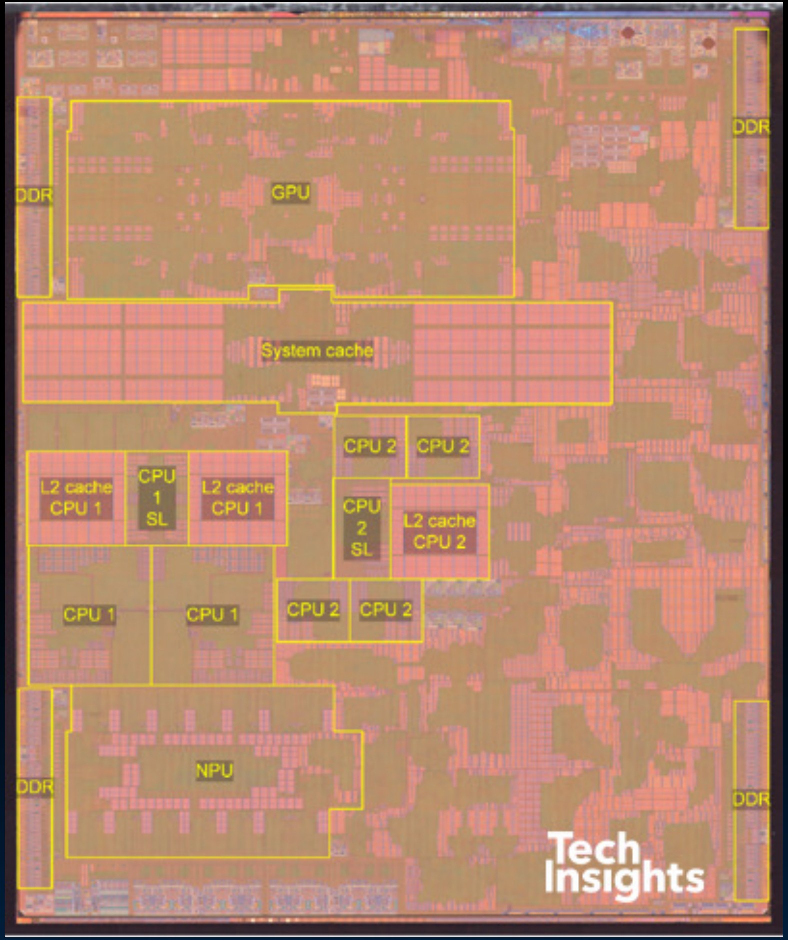
(more later)

- **Digital**: Represent all values by discrete values—specifically, as binary digits 1 and 0.
  - We've seen how to represent many symbols via 1s and 0s. This representation extends to **electrical signals**!
    - High voltage (1), Low voltage (0)
  - (By contrast, *analog circuits* use voltage/current to represent continuous ranges of values. These days, even a lot of analog circuitry use synchronous digital design by using analog-to-digital converters and vice versa.)

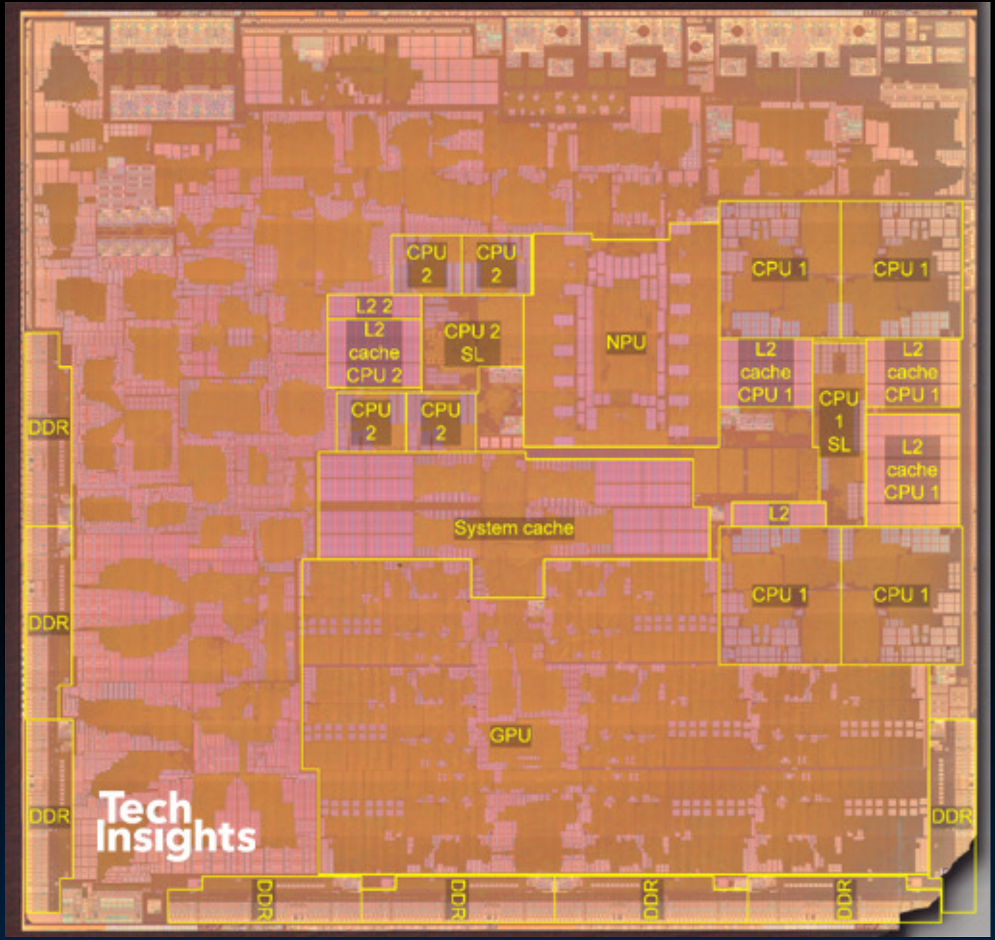
(today)

Yan, Yokota

Some IC (Integrated Chip) Photos



Apple A14 Bionic



Apple M1 Chip

On chip, all circuits are made from **transistors** and **wires**.

- (…also some "parasitic" resistors, capacitors, inductors.)

Apple Implementation of ARM v8.1-a:

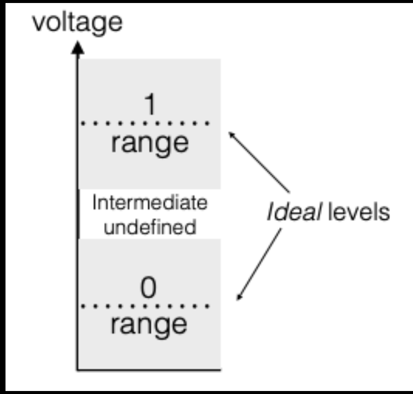
- 16M Transistors!
- 5W power consumption
- 5 nm process technology
- eight cores divided into two clusters
- four cores @ 3.2 GHz
- four cores @2.0 GHz
- CPU supports 64-bit data
- GPU for working with graphical data

Take EECS 151 and the subsequent EE 194 Tapeout Class for more! link

TechInsights [source]

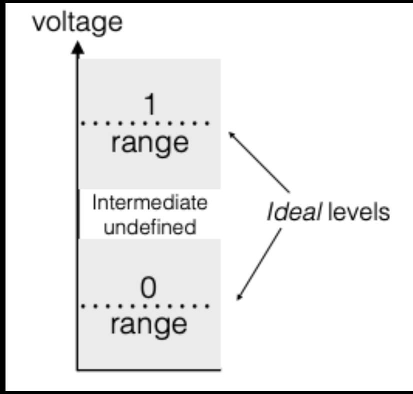
Binary Representation of Signals

- On a chip/PC board, **wires** (i.e., electrical nodes) provide electrical signals and are used to represent **variables**.
  - A wire can take on different values at different points in time.
- We choose to represent each wire as taking on two values via a **binary representation**. Use voltage levels to signal 0 or 1.

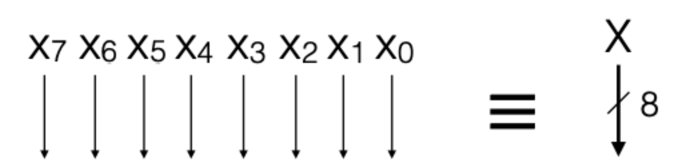


Binary Representation of Signals

- On a chip/PC board, **wires** (i.e., electrical nodes) provide electrical signals and are used to represent **variables**.
  - A wire can take on different values at different points in time.



- We choose to represent each wire as taking on two values via a **binary representation**. Use voltage levels to signal 0 or 1.
- Why not represent >2 values with the same electrical node?
  - **Reliability via good noise immunity**: All wires subject to interference /non-idealities, which grows worse at smaller sizes.
  - Circuits to discriminate between two possible inputs are simple to implement and have scaled well with **Moore's Law**. (more later)
  - Instead of making signals complex , we keep it **simple** and push complexity later into how we combine signals.
    - Notable exception: Flash (two bits per storage cell) (more much later)
    - (Note: early computers used decimal representation of signals)



SDS: Two Types of Circuits

- Synchronous Digital Systems consist of two basic types of circuits.
- (1) **Combinational Logic** circuits (today)
  - Output is a function of the inputs only.
  - Similar to a pure function in mathematics, y = f(x).
  - No way to store information from one invocation to the next, no side effects.
- (2) **State Elements** (next time)
  - Circuits that store information.
  - Example: Registers

Our Goal: Implement a **RISC-V processor** as a synchronous digital system.
This SDS should have the capabilities to execute RISC-V instructions.

Agenda

Logic Gates and Truth Tables

- Synchronous Digital Systems
- Logic Gates and Truth Tables
- Circuit Design, Part 1
- Boolean Algebra
- Circuit Design, Part 2

Yan, Yokota

Combinational Logic Circuits

- To design circuits that perform complex operations on binary signals:
  - First, define primitive operators.
  - Then, compose these primitive operators to perform more complex operations.
- Primitive operators for combinational logic are called **logic gates**.
  - The simplest logic gates are unary/binary operators that take as input **one/two binary variables** and output **one binary value**.

Yan, Yokota

Combinational Logic Circuits

- To design circuits that perform complex operations on binary signals:
  - First, define primitive operators.
  - Then, compose these primitive operators to perform more complex operations.
- Primitive operators for combinational logic are called **logic gates**.
  - The simplest logic gates are unary/binary operators that take as input **one/two binary variables** and output **one binary value**.
- Example: 2-input **AND** logic gate. Two binary inputs **a**, **b**; Binary output **y**

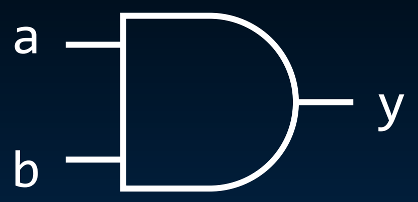
AND

`y = AND(a,b) = 1`
iff both a, b are 1
(= 0 otherwise)

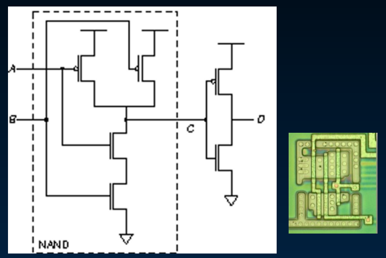
Function Definition

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table: Enumerate each input combination and the corresponding output value.



The symbol for a 2-input **AND gate**.

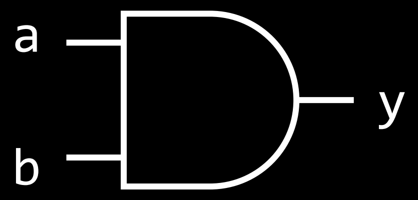


CMOS transistor circuit for AND logic gate
(out of scope, source)

Yan, Yokota

2-Input Logic Gates: AND, OR, NOT

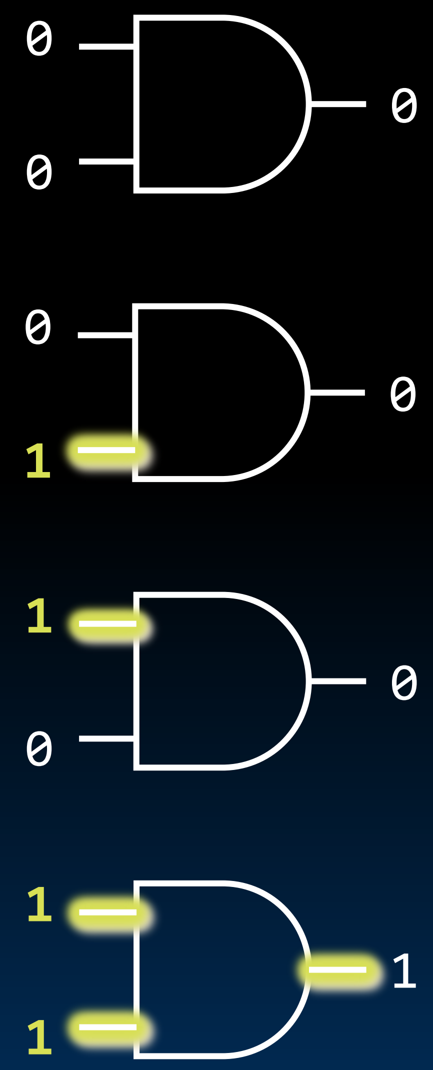
AND



`y = AND(a,b) = 1`
iff both a, b are 1
(= 0 otherwise)

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

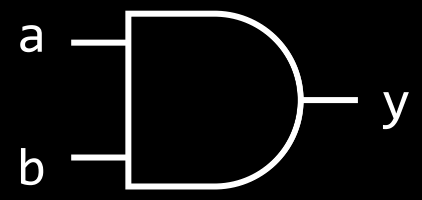
Truth Table: Enumerate each input combination and the corresponding output value.



Yan, Yokota

2-Input Logic Gates: AND, OR, NOT

AND

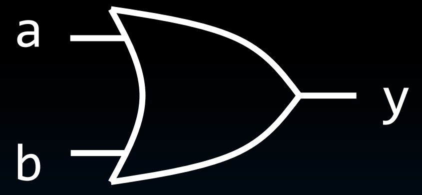


a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

`y = AND(a,b) = 1`
iff both a, b are 1

(mnemonic for remembering AND gate symbol)

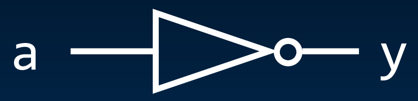
OR



a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

`y = OR(a,b) = 0`
iff both a, b are 0

NOT



a	y
0	1
1	0

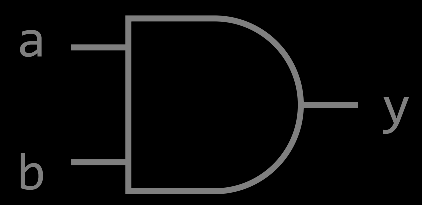
`y = NOT(a) = 1`
iff a is 0

[typo, fixed during lecture]

Yan, Yokota

2-Input Logic Gates: AND, OR, NOT

AND



y = AND(a,b) = 1
iff both a, b are 1

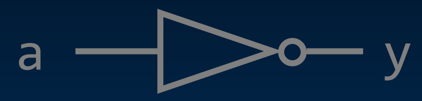
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

OR

y = OR(a,b) = 0
iff both a, b are 0

a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

NOT

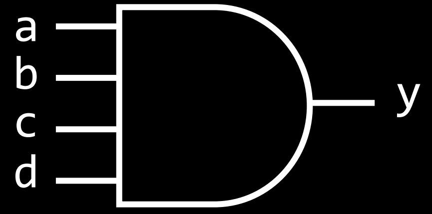


y = NOT(a) = 1
iff a is 0

[typo, fixed during lecture]

a	y
0	1
1	0

- AND, OR 2-input gate definitions extend to **n-input definitions**.
  - y = AND(a, b, c, d) = 1 iff all a, b, c, d are 1



- This small subset of logic gates is sufficient for implementing any (stateless) discrete function!
  - There are 3 more core logic gates in this course: XOR, NAND, NOR (more later)

For now, let's use the AND, OR, and NOT gates as our new basic building blocks to design circuits that perform meaningful functions.

Yan, Yokota

Agenda

Circuit Design, Part 1

- Synchronous Digital Systems
- Logic Gates and Truth Tables
- Circuit Design, Part 1
- Boolean Algebra
- Circuit Design, Part 2

Yan, Yokota

Designing Combinational Logic Circuits

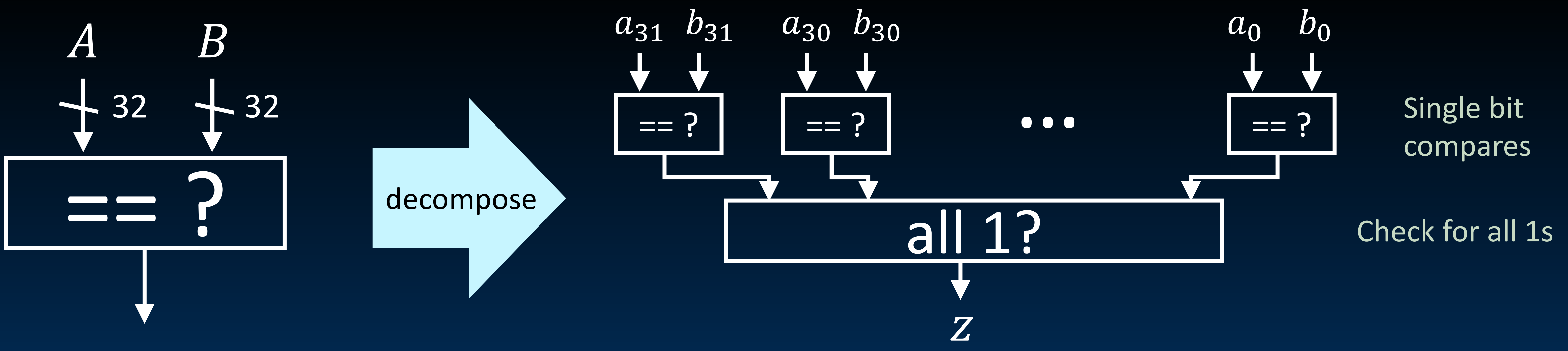
- Logic gates are basic building blocks to design circuits (gate diagrams) that perform meaningful functions.
  - For now: AND, OR, NOT

Example:

- Recall the RISC-V instruction: `beq rs1, rs2, label`
  - If value in **rs1 == value** in rs2, then go to instruction at label, else go to next instruction.
- Somewhere in the processor must be a circuit that **compares** 32-bit values.
- Let's implement this!

(1/3) Implement an Equality Compare Circuit

- Recall the RISC-V instruction: `beq rs1, rs2, label`
  - If value in **rs1 == value** in rs2, then go to instruction at label, else go to next instruction
- Somewhere in the processor must be a circuit that **compares** 32-bit values.
- For now, assume a dedicated "**equal compare**" circuit of two 32-bit inputs:
  - We could also subtract the two and check for result == 0… (more on Project 3)



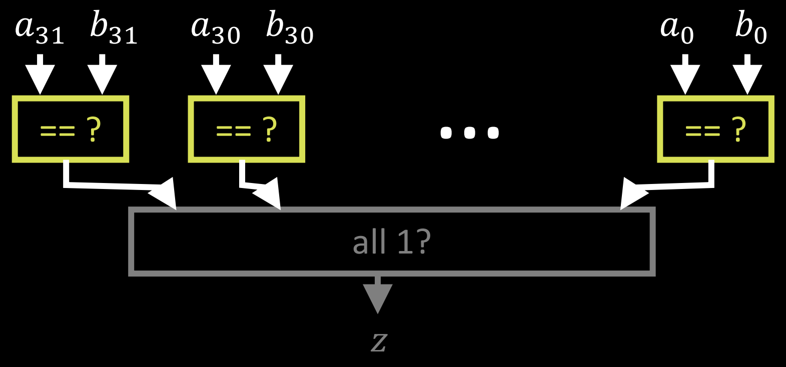
z = 1 iff A == B

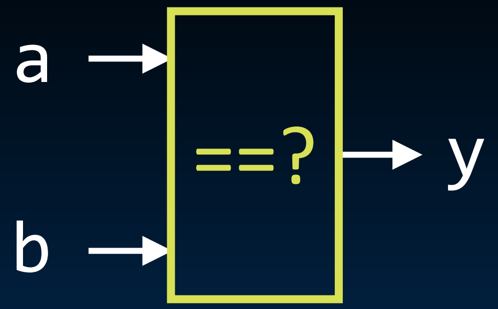
= 1 iff $a_{31} == b_{31}$ and $a_{30} == b_{30}$ and … and $a_0 == b_0$

Yan, Yokota

(2/3) Single-bit Compare Circuit

- If we don't already have a single-bit compare circuit* in our technology library, we can implement it from scratch with logic gates.
- Truth Table → Gate Diagram:
  1. Construct the truth table for the function definition by enumerating all input/output pairs.
  2. Then, use the truth table to construct a gate diagram.





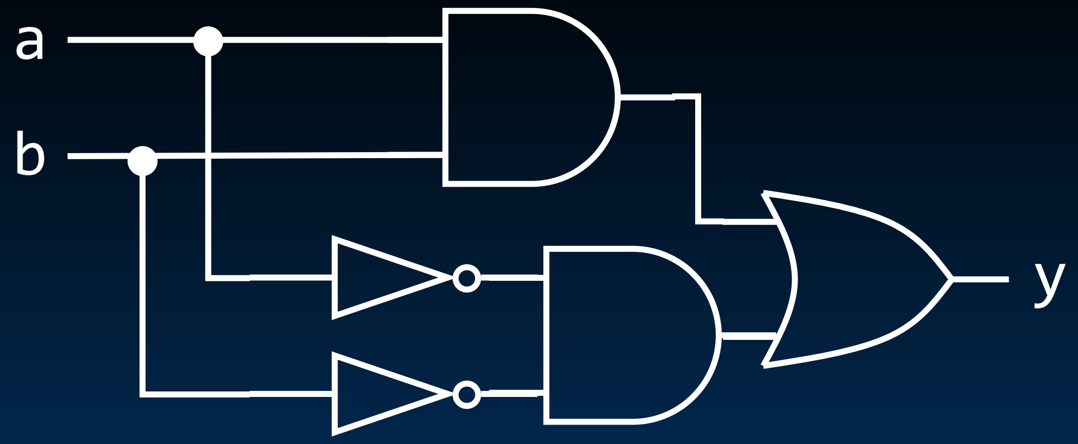
y = 1 iff a, b equal

1.

a	b	y
0	0	1
0	1	0
1	0	0
1	1	1

y = 1 iff a=b=0
or a=b=1

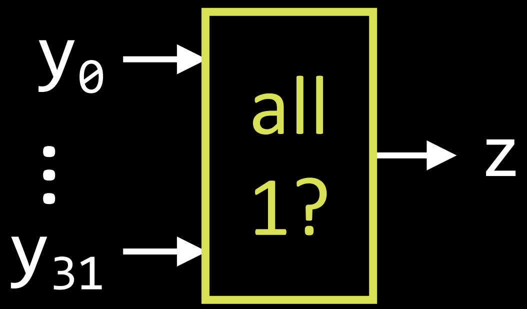
2.



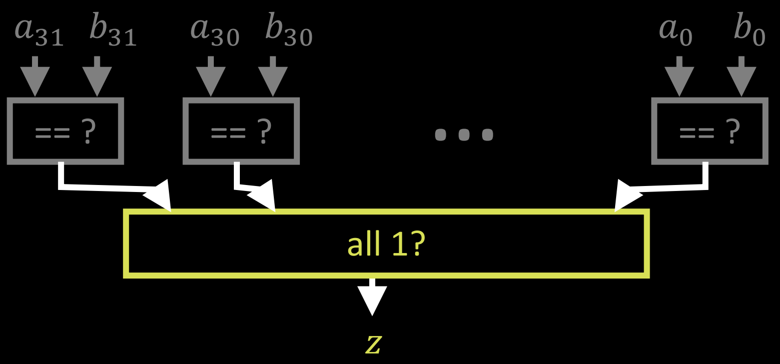
*single-bit compare is XNOR

(3/3) Check-for-all 1 Circuit

- Functionally, this is a 32-input AND gate!

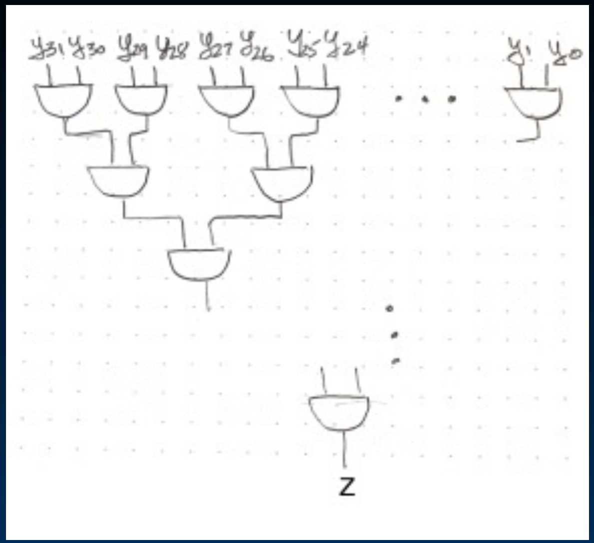


$z = 1$ iff $y_{31} = y_{30} = \ldots = y_0 = 1$



- If available in our technology library:

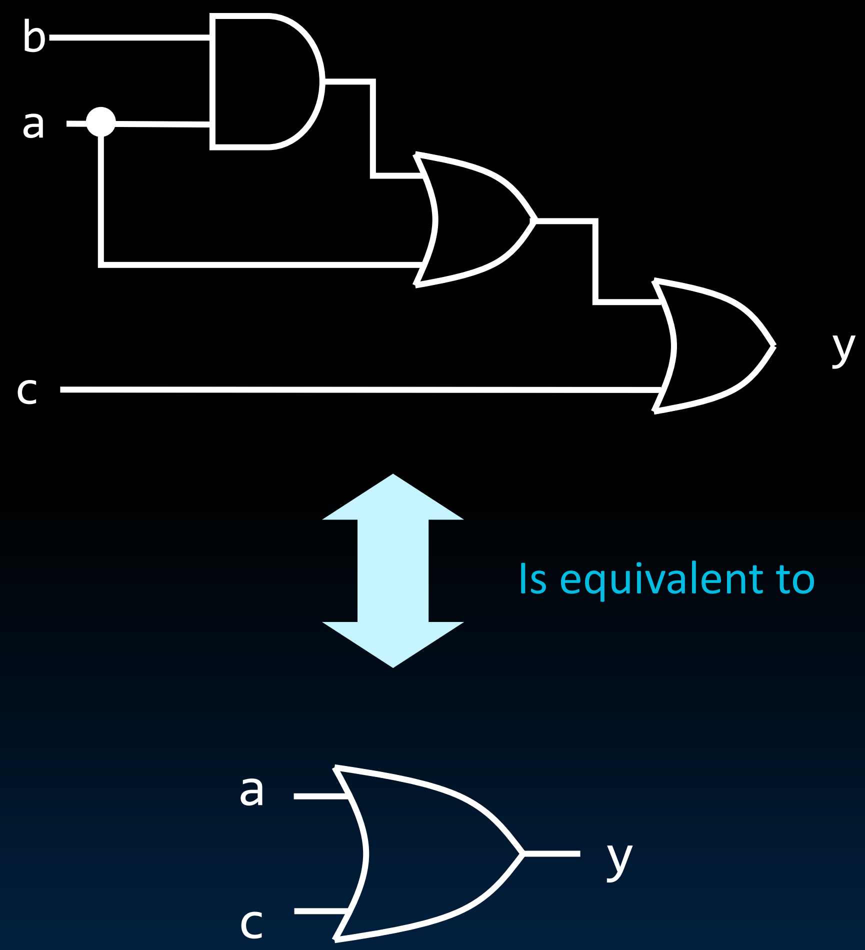
- Otherwise, build recursively:



Yan, Yokota

So far: Combinational Logic Block Design

- Truth Table → Gate Diagram:
  1. Construct the truth table for the function definition by enumerating all input/output pairs.
  2. Then, use the truth table to construct a gate diagram.
- **Modular design**: If truth tables are too big to construct, define smaller blocks first.
- Drawbacks:
  - Going from the truth table to a working gate diagram could be complex!
  - Multiple gate diagrams can represent the same truth table.
    - How do we prove that two gate diagrams are **equivalent**?
    - How do we choose the **simplest** gate diagram?



Yan, Yokota

Agenda

Boolean Algebra

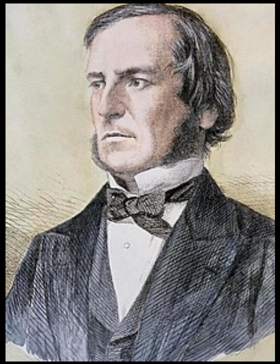
- Synchronous Digital Systems
- Logic Gates and Truth Tables
- Circuit Design, Part 1
- Boolean Algebra
- Circuit Design, Part 2

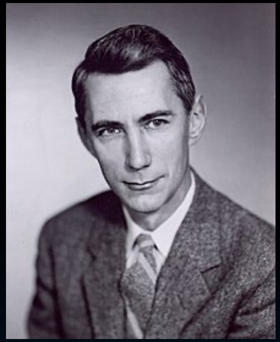
Yan, Yokota

Historical Note: Boolean Algebra

- Boole, a 19th century mathematician, developed a mathematical system (algebra) for logic.
  - Primitive functions: And, OR, NOT
  - Later known as **Boolean Algebra**
- Early 20th century computer designers noticed common patterns in their work: ANDs, ORs, …
- Shannon, 1940, wrote his M.S. thesis that linked Boolean Algebra to **logic gates**.
  - There is a one-to-one correspondence between circuits composed of AND, OR, NOT gates and Boolean Algebra equations!
  - Can now apply math to give theory to hardware design, minimization, …

George Boole (1815–1864) *was too cool, literally*





Claude Shannon (1916–2001) *was too cool, figuratively*

Logic Gate Function	Algebraic Expression
OR(a,b)	$a + b$
AND(a,b)	$a \cdot b$ aka ab
NOT(a)	$\bar{a}$ (or a')

Yan, Yokota

Laws of Boolean Algebra

- The Laws of Boolean Algebra allow us to simplify expressions.

AND form	OR form	
$x \cdot y = y \cdot x$	$x + y = y + x$	Commutativity
$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$	Associativity
$x \cdot 1 = x$	$x + 0 = x$	Identity
$x \cdot 0 = 0$	$x + 1 = 1$	Laws of 0's and 1's
$xy + x = x$	$(x + y)x = x$	Uniting Theorem
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$	Distributivity
$x \cdot x = x$	$x + x = x$	Idempotence
$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$	Inverse (Complement)
$\overline{(xy)} = \bar{x} + \bar{y}$	$\overline{(x + y)} = \bar{x} \cdot \bar{y}$	DeMorgan's Laws

- Some similar to ordinary algebra…
  - match AND to multiplication
  - match OR to addition
- …but many are BA-specific, i.e., variables take on two truth values:
  - 1 (true)
  - 0 (false)

See Reference Card!

Place a pin on the Boolean Algebra Law you'd like to go over. Vote once.

0

Proving Laws of Boolean Algebra

[reference]

- Most Laws can be proved via an exhaustive proof (i.e., truth tables).

$$\boxed{x + 1 = 1}$$

Law of 1's

x	x+1
0	1
1	1

$$\boxed{x \cdot x = x}$$

Idempotence (AND)

x	xx
0	0
1	1

$$\boxed{xy + x = x}$$

Uniting Theorem (AND)

x	y	xy	xy+x
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

- Others can be proven using other Laws.

$$\boxed{x + yz = (x + y)(x + z)}$$

Distributivity (Property 2)

$$(x + y)(x + z)$$

$$= x \cdot x + xz + xy + yz \qquad \text{Distributivity (Prop. 1)}$$

$$= x + xz + xy + yz \qquad \text{Idempotence (AND)}$$

$$= x + x(y + z) + yz \qquad \text{Distributivity (Prop. 1)}$$

$$= x + yz \qquad \text{Uniting Theorem (AND)}$$

Yan, Yokota

DeMorgan's Laws

[reference]

AND form	OR form	
$\overline{(xy)} = \bar{x} + \bar{y}$	$\overline{(x + y)} = \bar{x} \cdot \bar{y}$	DeMorgan's Laws

x	y	$\bar{x}$	$\bar{y}$	$\bar{x}+\bar{y}$	xy	$\overline{(xy)}$
0	0	1	1	1	0	1
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	0	1	0

x	y	$\bar{x}$	$\bar{y}$	$\bar{x} \cdot \bar{y}$	x+y	$\overline{(x+y)}$
0	0	1	1	1	0	1
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	1	0	0	0	1	0

Yan, Yokota

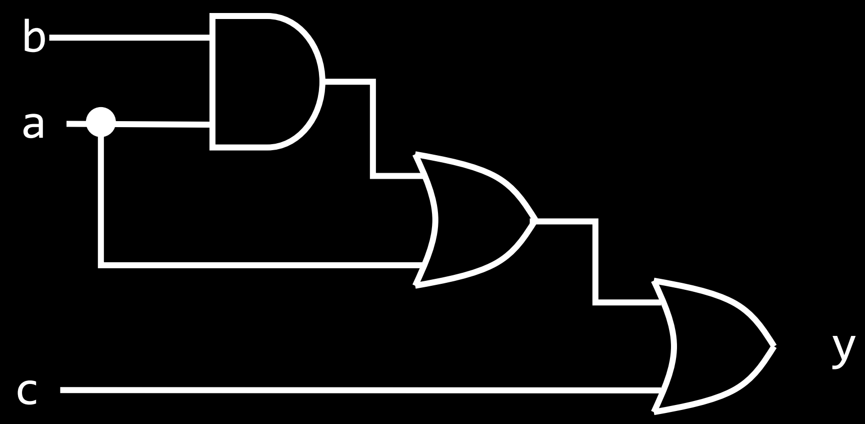
Agenda

Circuit Design, Part 2

- Synchronous Digital Systems
- Logic Gates and Truth Tables
- Circuit Design, Part 1
- Boolean Algebra
- Circuit Design, Part 2

From Before: Combinational Logic Block Design

- Truth Table → Gate Diagram:
  1. Construct the truth table for the function definition by enumerating all input/output pairs.
  2. Then, use the truth table to construct a gate diagram.
- **Modular design**: If truth tables are too big to construct, define smaller blocks first.
- Drawbacks:
  - Going from the truth table to a working gate diagram could be complex!
  - Multiple gate diagrams can represent the same truth table.
    - How do we prove that two gate diagrams are **equivalent**?
    - How do we choose the **simplest** gate diagram?



Is equivalent to

Let's prove that these diagrams are equivalent.

Yan, Yokota

[Example 1] Circuit → Boolean Expression

Which expression represents this gate diagram?

A. $y = (a + b)ac$

B. $y = (a + b)\bar{a}c$

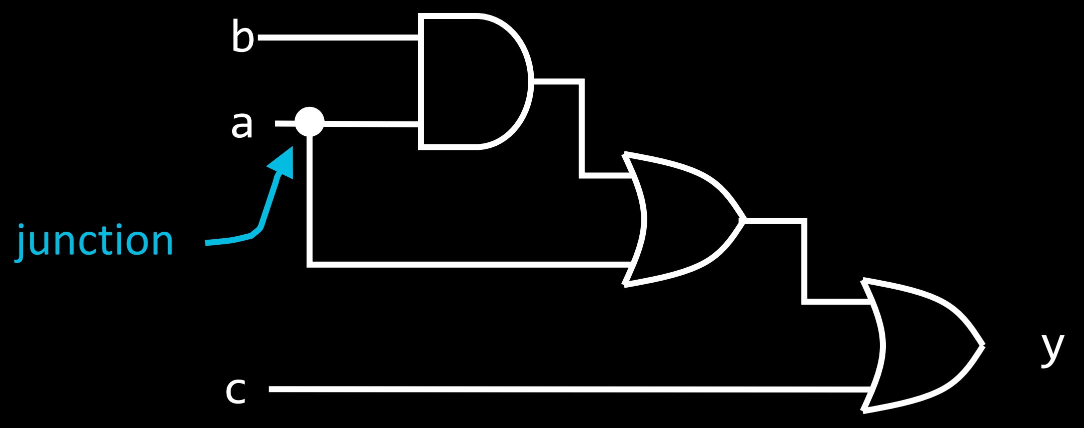
C. $y = \overline{(a + b)}ac$

D. $y = ab + a + c$

E. $y = ab + \bar{a} + c$

F. $y = \overline{ba} + a + c$

G. Something else



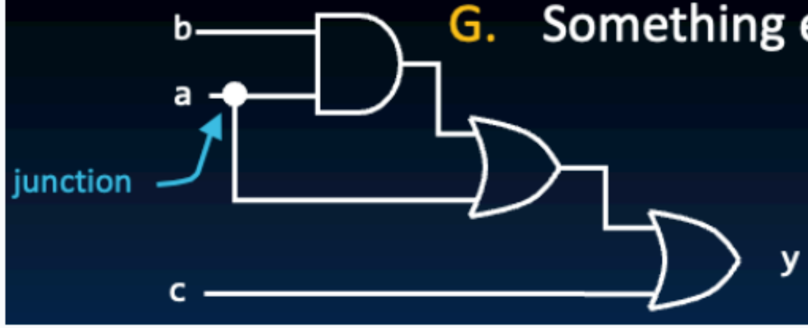
original circuit

Yan, Yokota

Which expression represents this gate diagram?

0

A. $y = (a+b)ac$
B. $y = (a+b)\bar{a}c$
C. $y = \overline{(a+b)}ac$
D. $y = ab + a + c$
E. $y = ab + \bar{a} + c$
F. $y = \overline{ba} + a + c$
G. Something else



A	0%
B	0%
C	0%
D	0%
E	0%
F	0%
Something else	0%

[Example 1] Circuit → Boolean Expression

[for next time]

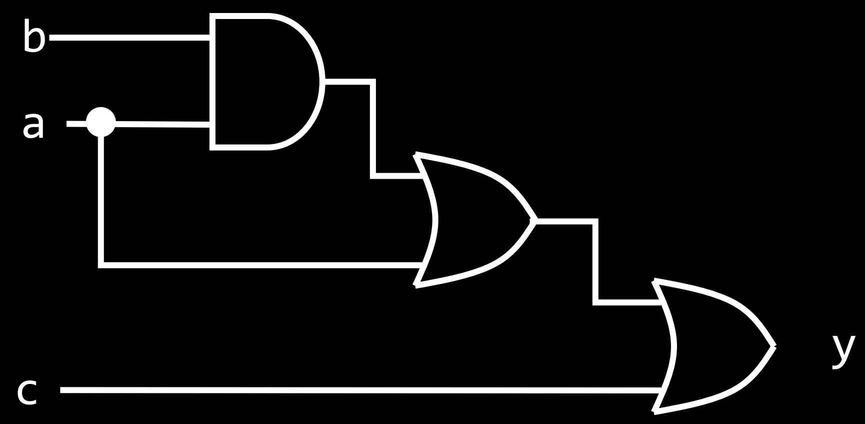
D. $y = ab + a + c$ — Equation derived from original circuit

$= a(b+1) + c$ — Distributivity

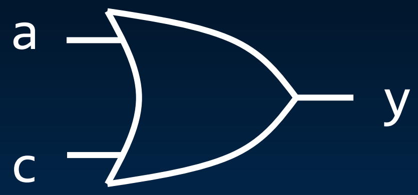
$= a(1) + c$ — Law of 1's

$= a + c$ — Identity (AND)

G. Something else



original circuit



new circuit

To represent a function: There is one unique truth table, but there are multiple Boolean expressions and multiple circuit diagrams.

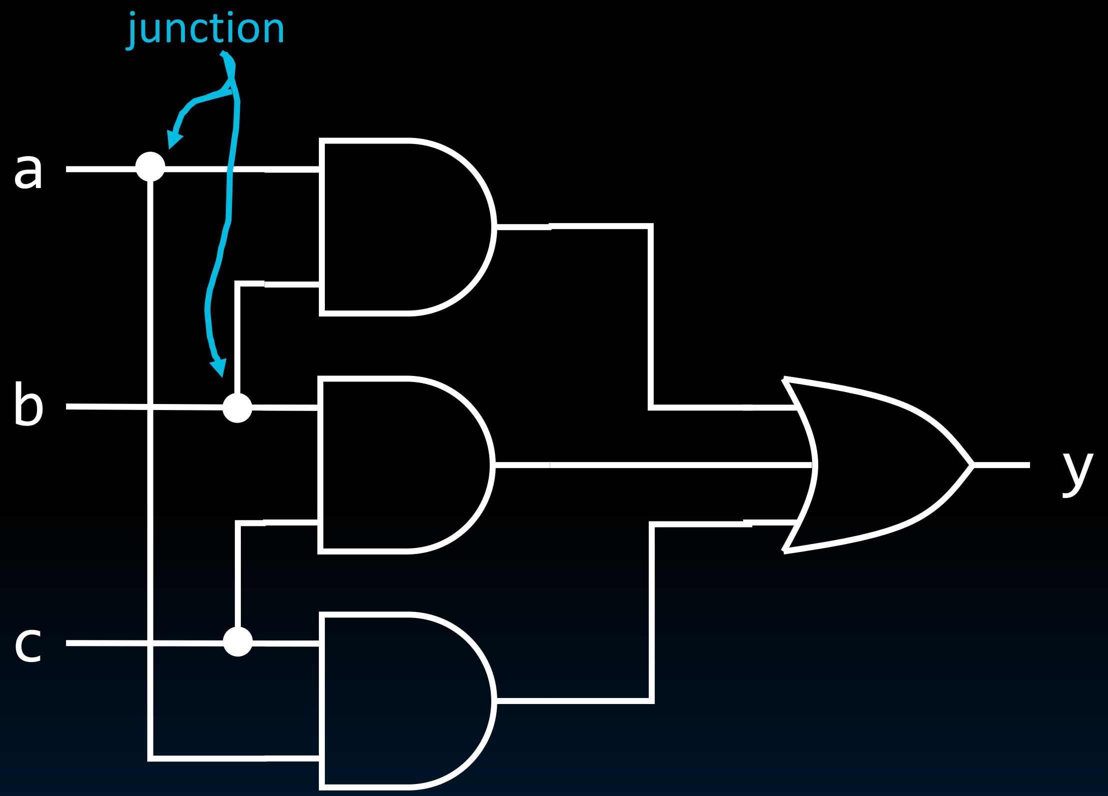
a	b	c	orig	new
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

(verification via truth table)

Yan, Yokota

[Example 2] Circuit → Boolean Expression

[for next time]



What does this circuit do?

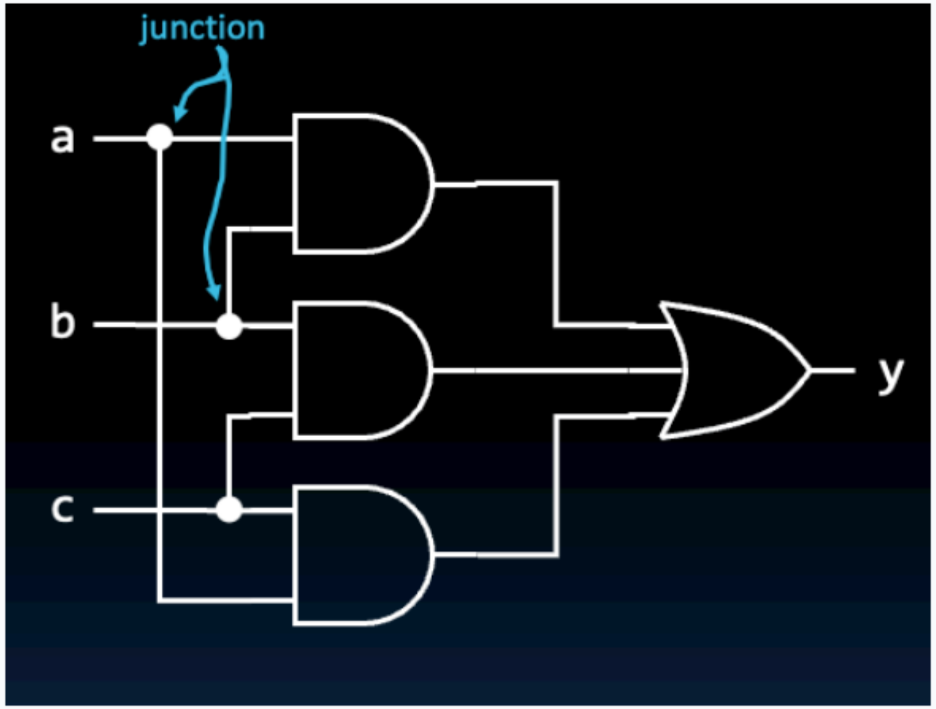
(select all that apply)

A. 1 if a = c = 1
B. 1 if b = c = 1
C. 1 if a = c = 1
D. 1 if a = b = c = 1
E. 1 if at least two of a, b, c are 1
F. The "majority" bit among a, b, and c
G. None of the above

Yan, Yokota

What does this circuit do? (select all that apply)

0



1 if a = c = 1

0

1 if b = c = 1

0

1 if a = c = 1

0

1 if a = b = c = 1

0

1 if at least two of , b, c are 1

0

The “majority” bit among a, b, and c

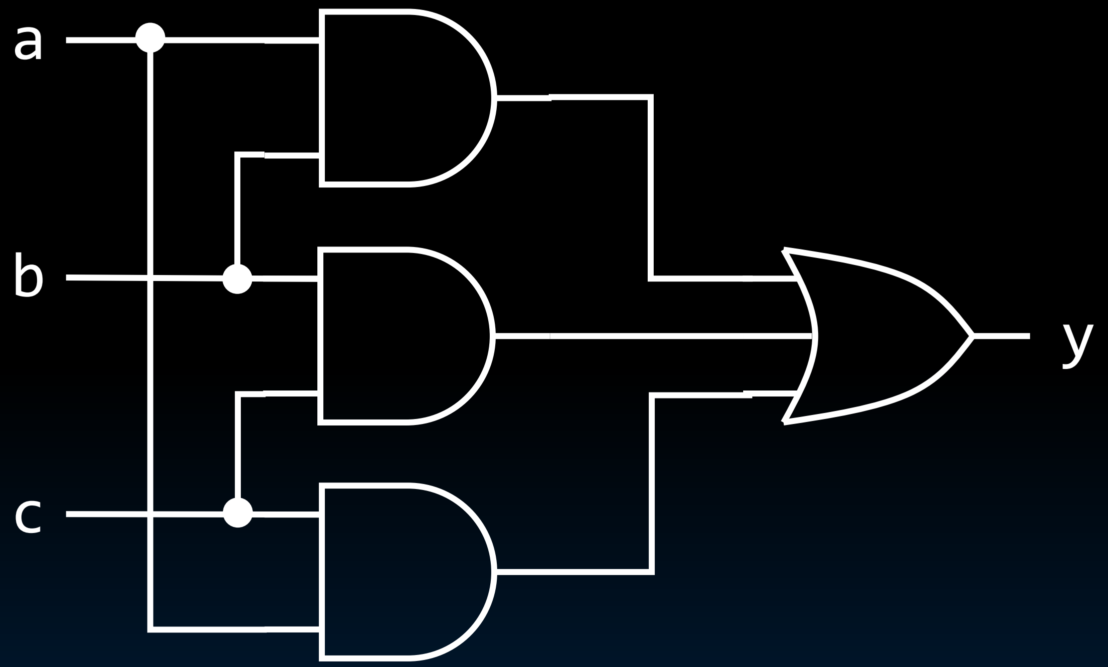
0

None of the above

0

[Example 2] Majority Circuit → Boolean Expression

(defined directly from gate diagram)



$$y = a \cdot b + a \cdot c + b \cdot c$$
$$= ab + ac + bc$$

Yan, Yokota

[Example 3] Truth Table → Bool Exp. → Gates

1. Use the truth table to write the canonical form (i.e., **Sum of Products**).
2. Simplify using laws of Boolean Algebra.
3. Then construct a gate diagram.

a	b	c	y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Given this truth table, how do we construct a gate diagram?

Yan, Yokota

[Example 3] Truth Table → Bool Exp. → Gates

1. Use the truth table to write the canonical form (i.e., **Sum of Products**).
2. Simplify using laws of Boolean Algebra.
3. Then construct a gate diagram.

a	b	c	y	
0	0	0	1	$\bar{a} \cdot \bar{b} \cdot \bar{c}$
0	0	1	1	$\bar{a} \cdot \bar{b} \cdot c$
0	1	0	0	
0	1	1	0	
1	0	0	1	$a \cdot \bar{b} \cdot \bar{c}$
1	0	1	0	
1	1	0	1	$a \cdot b \cdot \bar{c}$
1	1	1	0	

$$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}$$

Why does **Sum of Products** work?

- y = 1 if at least one of these expressions is 1.
- y = 0 otherwise (i.e., none of these expressions are 1).

Yan, Yokota

[Example 3] Truth Table → Bool Exp. → Gates

1. Use the truth table to write the canonical form (i.e., **Sum of Products**).
2. Simplify using laws of Boolean Algebra.
3. Then construct a gate diagram.

a	b	c	y	
0	0	0	1	$\bar{a} \cdot \bar{b} \cdot \bar{c}$
0	0	1	1	$\bar{a} \cdot \bar{b} \cdot c$
0	1	0	0	
0	1	1	0	
1	0	0	1	$a \cdot \bar{b} \cdot \bar{c}$
1	0	1	0	
1	1	0	1	$a \cdot b \cdot \bar{c}$
1	1	1	0	

$$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}$$

$$= \bar{a}\bar{b}(\bar{c} + c) + a\bar{c}\left(\bar{b} + b\right)$$ Distributivity

$$= \bar{a}\bar{b}(1) + a\bar{c}(1)$$ Inverse (OR) $x + \bar{x} = 1$

$$= \bar{a}\bar{b} + a\bar{c}$$ Identity (AND) $x \cdot 1 = x$

Yan, Yokota

[Example 3] Truth Table → Bool Exp. → Gates

1. Use the truth table to write the canonical form (i.e., **Sum of Products**).
2. Simplify using laws of Boolean Algebra.
3. Then construct a gate diagram.

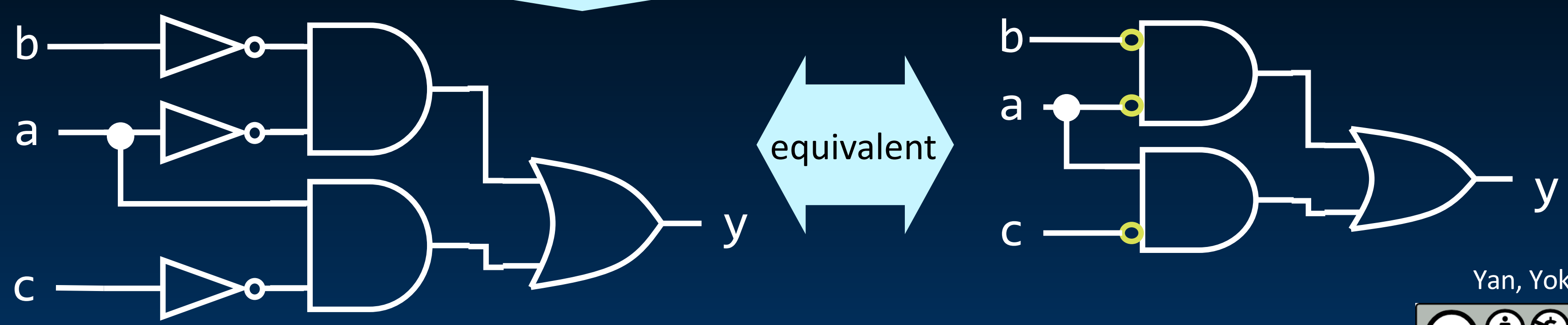
a	b	c	y	
0	0	0	1	$\bar{a}\cdot\bar{b}\cdot\bar{c}$
0	0	1	1	$\bar{a}\cdot\bar{b}\cdot c$
0	1	0	0	
0	1	1	0	
1	0	0	1	$a\cdot\bar{b}\cdot\bar{c}$
1	0	1	0	
1	1	0	1	$a\cdot b\cdot\bar{c}$
1	1	1	0	

$$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c + a\bar{b}\bar{c} + ab\bar{c}$$

$$= \bar{a}\bar{b}(\bar{c} + c) + a\bar{c}\left(\bar{b} + b\right)$$ Distributivity

$$= \bar{a}\bar{b}(1) + a\bar{c}(1)$$ Inverse (OR) $x + \bar{x} = 1$

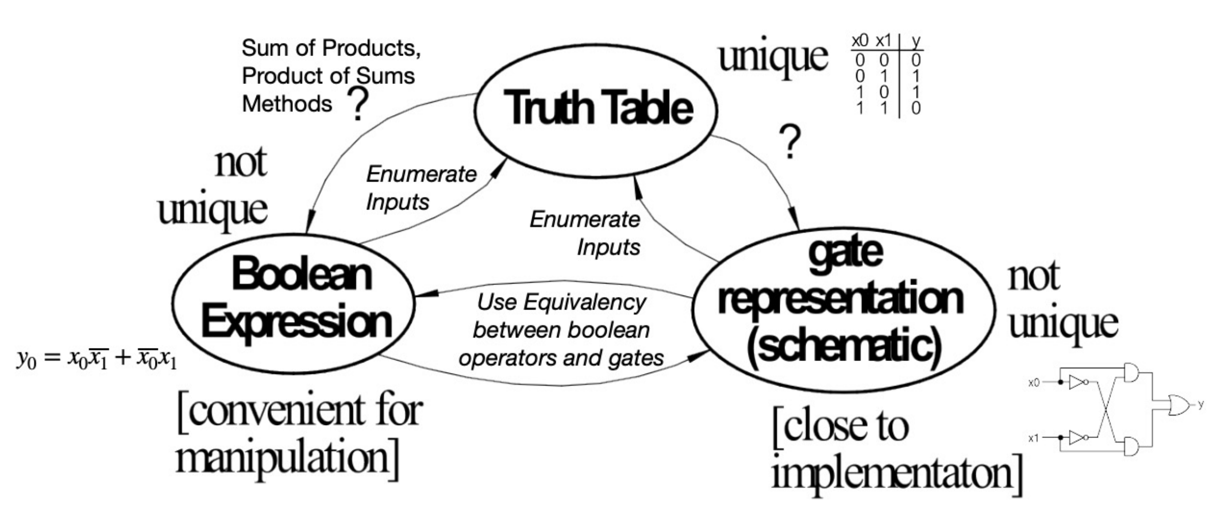
$$= \bar{a}\bar{b} + a\bar{c}$$ Identity (AND) $x \cdot 1 = x$



Yan, Yokota

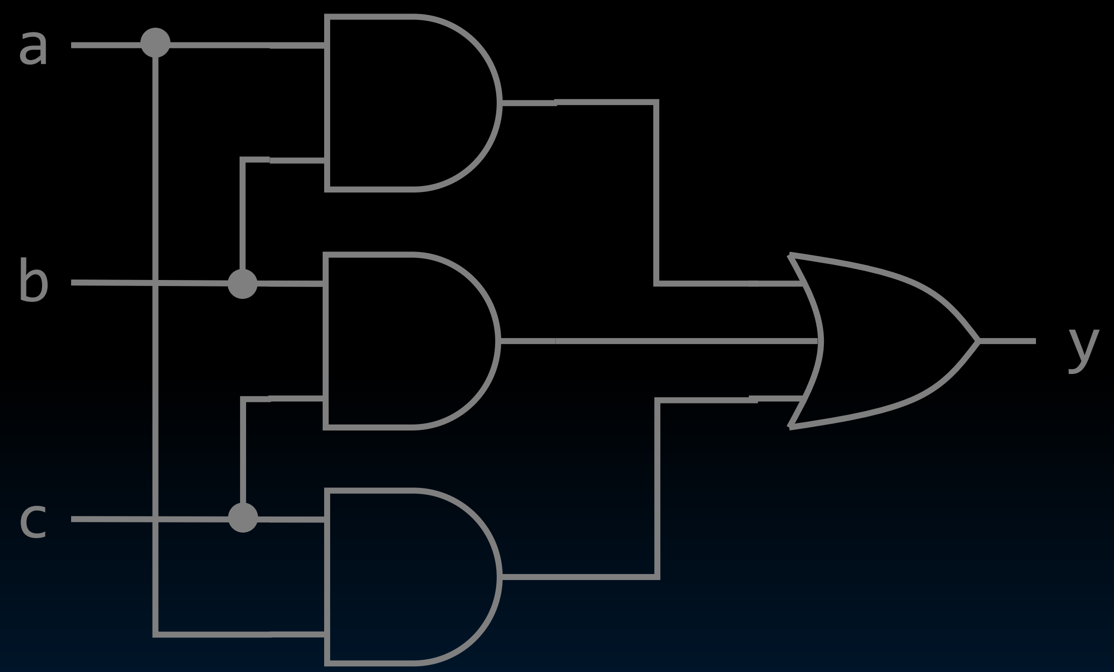
Summary: Combinational Logic Block Design

- Truth Table → Gate Diagram:
  1. Construct the truth table for the function definition by enumerating all input/output pairs.
  2. Then, use the truth table to construct a gate diagram.
- **Modular design**: If truth tables are infeasible, define smaller blocks first.
- **Simple design** with **Boolean Algebra**:
  1. (if possible) Write a Boolean expression based on the existing gate diagram. (otherwise) Use the truth table to write the canonical form (i.e., **Sum of Products**).
  2. Simplify using laws of Boolean Algebra.
  3. Then construct a gate diagram.



[Example 4] Majority Circuit → Boolean Expression

(defined directly from gate diagram)



$$y = a \cdot b + a \cdot c + b \cdot c$$
$$= ab + ac + bc$$

(or, derive simple expression from canonical form)

$$
\begin{aligned}
y &\\
&= \bar{a}bc + a\bar{b}c + ab\bar{c} + abc \\
&= \bar{a}bc + abc + a\bar{b}c + ab\bar{c} + abc \\
&= (\bar{a} + a)bc + a\bar{b}c + ab\bar{c} + abc \\
&= (1)bc + a\bar{b}c + ab\bar{c} + abc \\
&= bc + a\bar{b}c + abc + ab\bar{c} + abc \\
&= bc + a(\bar{b} + b)c + ab\bar{c} + abc \\
&= bc + a(1)c + ab\bar{c} + abc \\
&= bc + ac + ab(\bar{c} + c) \\
&= bc + ac + ab(1) \\
&= bc + ac + ab
\end{aligned}
$$

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

[for reference]