Digital Design & Computer Arch.

Lecture 5: Combinational Logic II

Prof. Onur Mutlu

ETH Zürich

Spring 2022

10 March 2022

Assignment: Lecture Video (Due April 1)

- Why study computer architecture? Why is it important?
- Future Computing Platforms: Challenges & Opportunities

- **Required Assignment**
  - **Watch one of** Prof. Mutlu's lectures and analyze either (or both)
  - https://www.youtube.com/watch?v=kgiZlSOcGFM (May 2017)
  - https://www.youtube.com/watch?v=mskTeNnf-i0 (Feb 2021)

- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page summary** of one of the lectures and email us
    - What are your key takeaways?
    - What did you learn?
    - What did you like or dislike?
    - Submit your summary to Moodle by April 1

Extra Credit Assignment: Due March 15

- **Attend and watch Sean Lie's talk on Feb 28**
  - Either on Zoom or Youtube
  - https://safari.ethz.ch/safari-live-seminar-sean-lie-28-feb-2022/
  - https://www.youtube.com/watch?v=x2-qB0J7KHw

- **Optional Assignment – for 1% extra credit**
  - **Write and submit a 1-page summary** of the talk
    - What are the key ideas used in the Cerebras system?
    - What are your key takeaways from the talk?
    - What did you learn?
    - What did you like or dislike?
    - Submit your summary to Moodle: https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=722952

Assignment: Required Readings

- This week
  - Combinational Logic
    - P&P Chapter 3 until 3.3 + H&H Chapter 2
- Next week
  - Hardware Description Languages and Verilog
    - H&H Chapter 4 until 4.3 and 4.5
  - Sequential Logic
    - P&P Chapter 3.4 until end + H&H Chapter 3 in full
- By the end of next week, make sure you are done with
  - **P&P Chapters 1-3 + H&H Chapters 1-4**

Combinational Logic Circuits and Design

What We Will Learn in This Lecture

- Building blocks of modern computers
  - Transistors
  - Logic gates
- Combinational circuits
- Boolean algebra
- Using Boolean algebra to represent combinational circuits
- Basic combinational logic blocks
- Simplifying combinational logic circuits

All Computers are Built Upon the Same Building Blocks

Modern General-Purpose Microproc

5-nanometer process

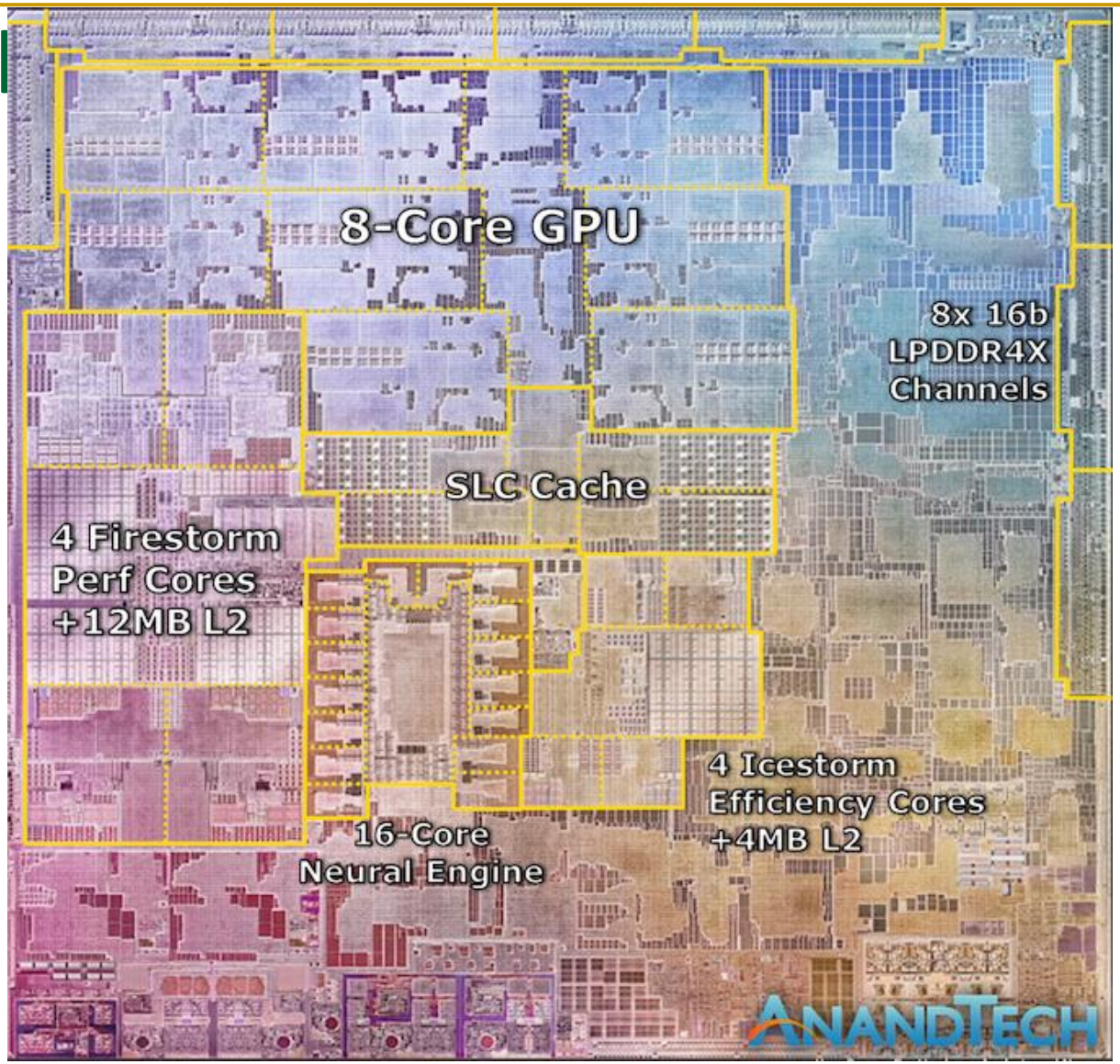
The first personal computer chip built with this cutting-edge technology.

16 billion transistors

The most we've ever put into a single chip.

Source: https://www.apple.com/mac/m1/

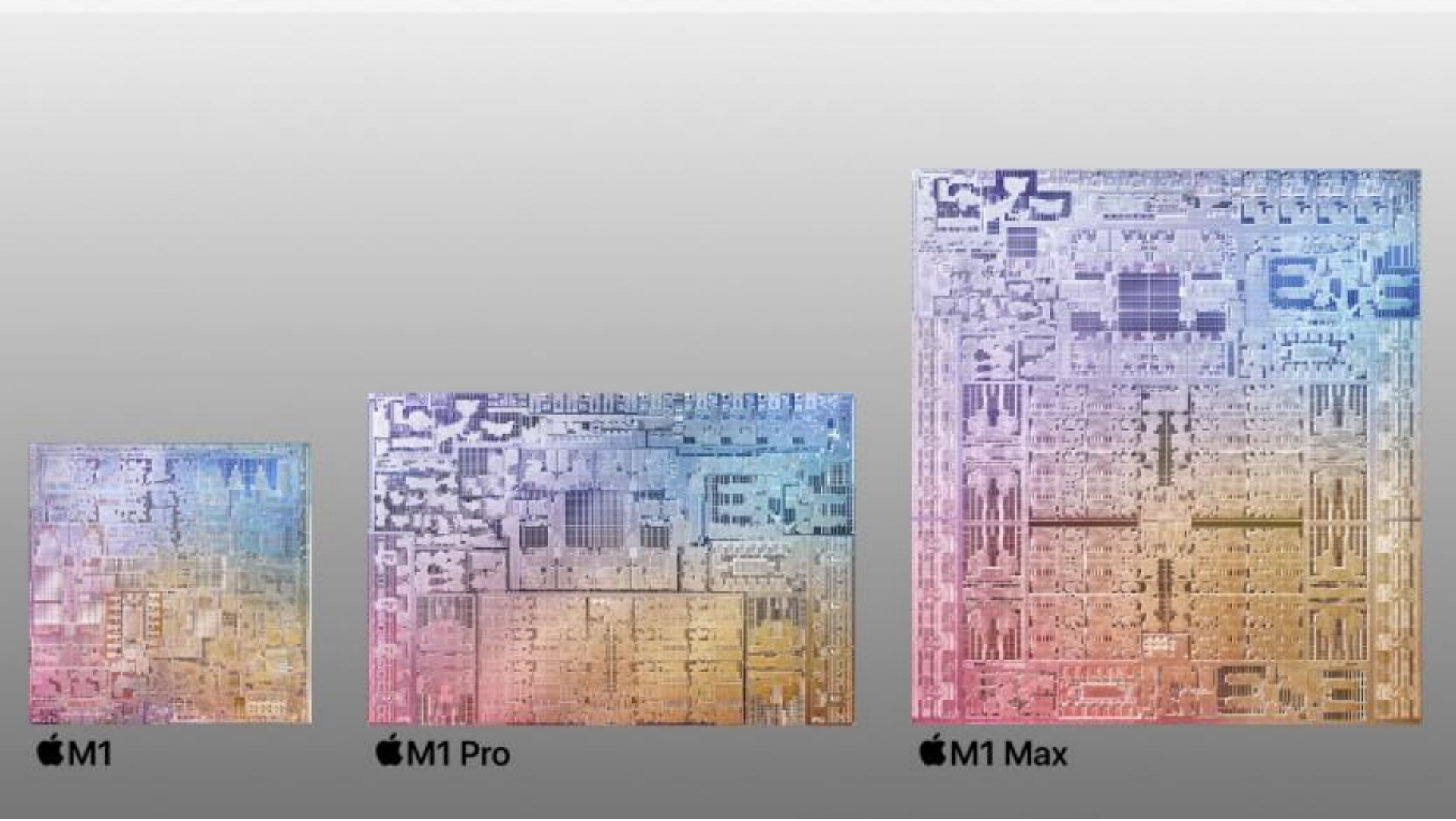
Modern General-Purpose Microp



Apple M1, 2021

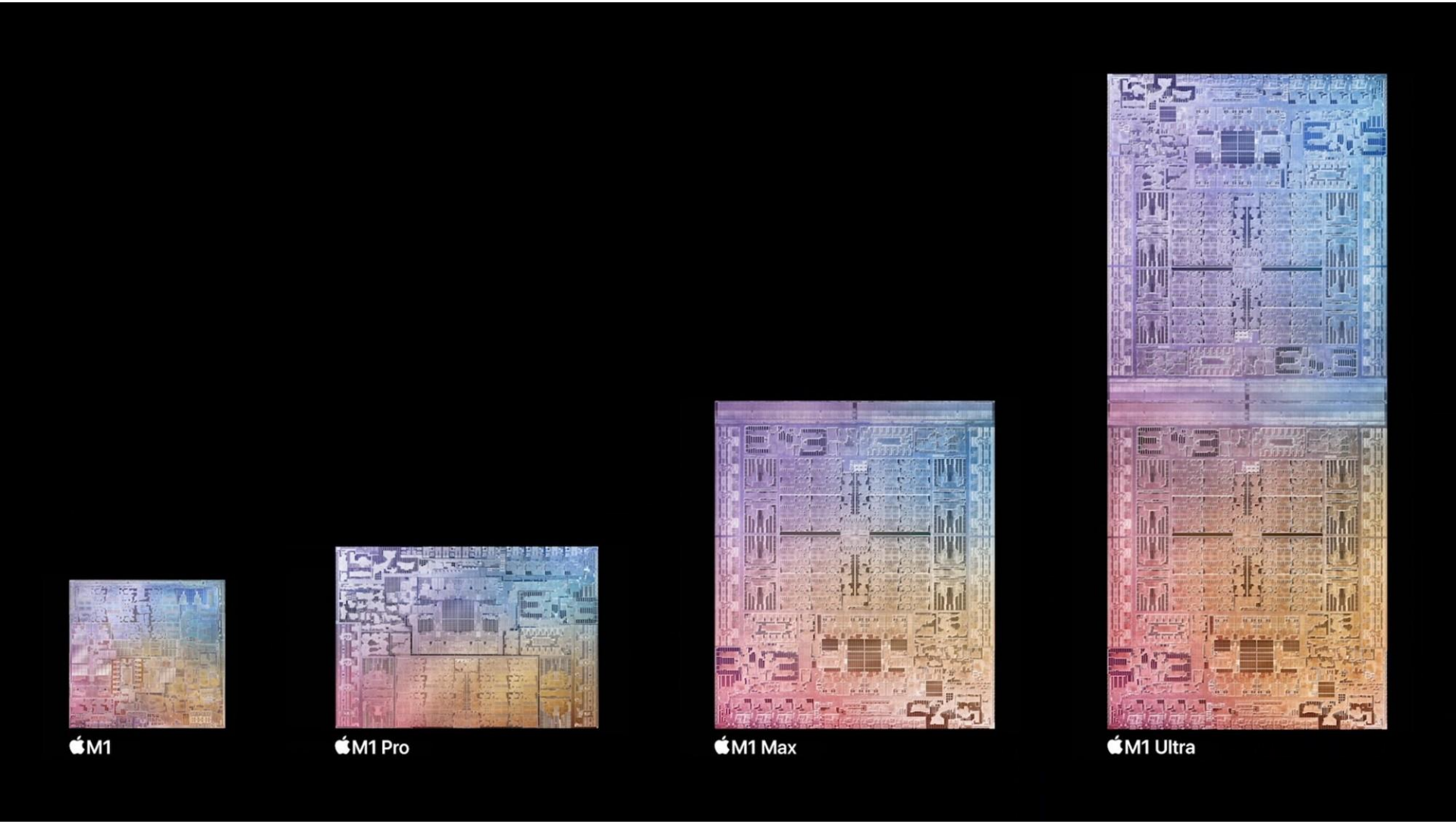
Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

Modern General-Purpose



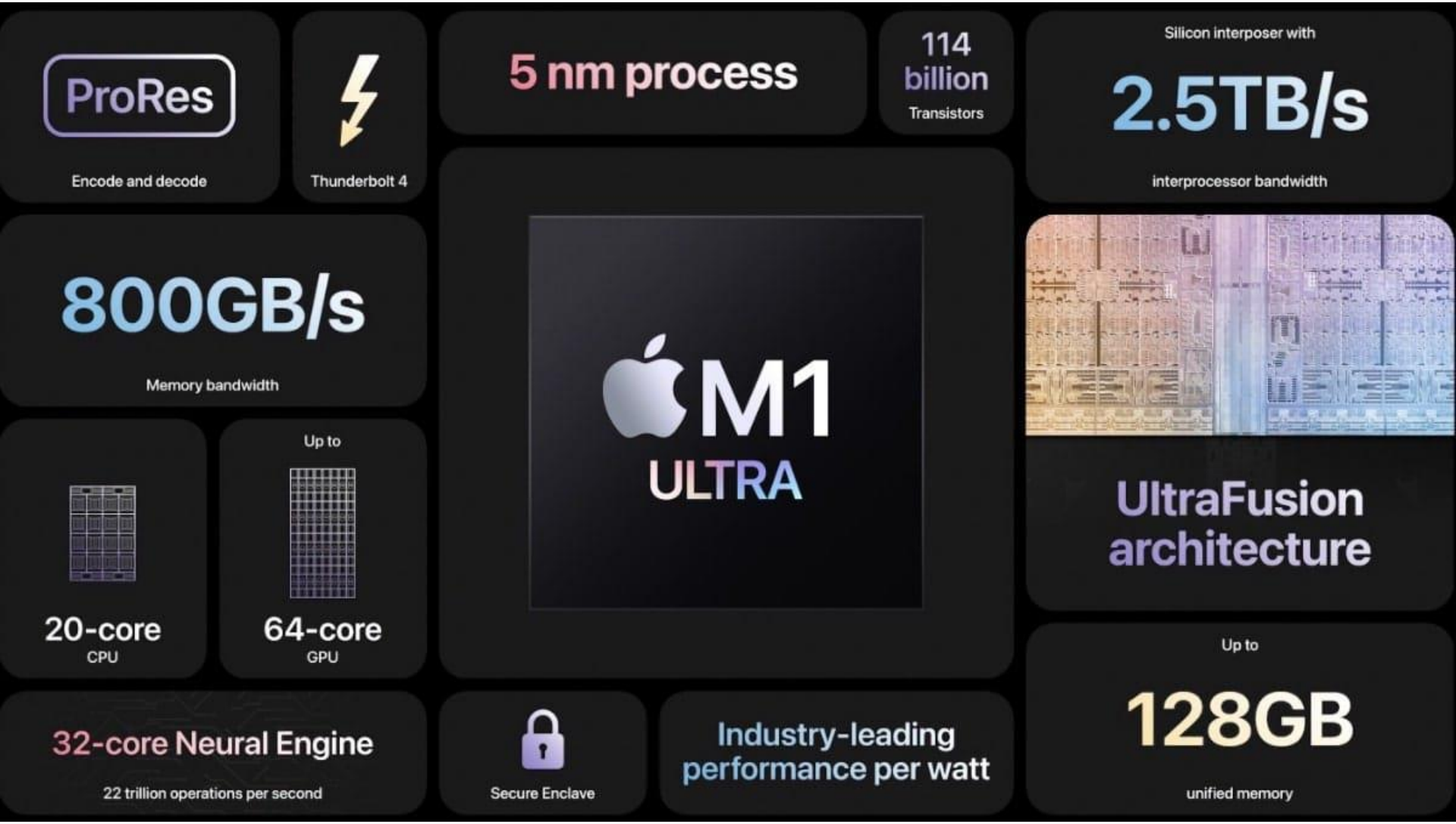
Source: https://www.golem.de/news/m1-pro-max-dieses-apple-silicon-ist-gigantisch-2110-160415.html

Apple M1 Ultra (2022)



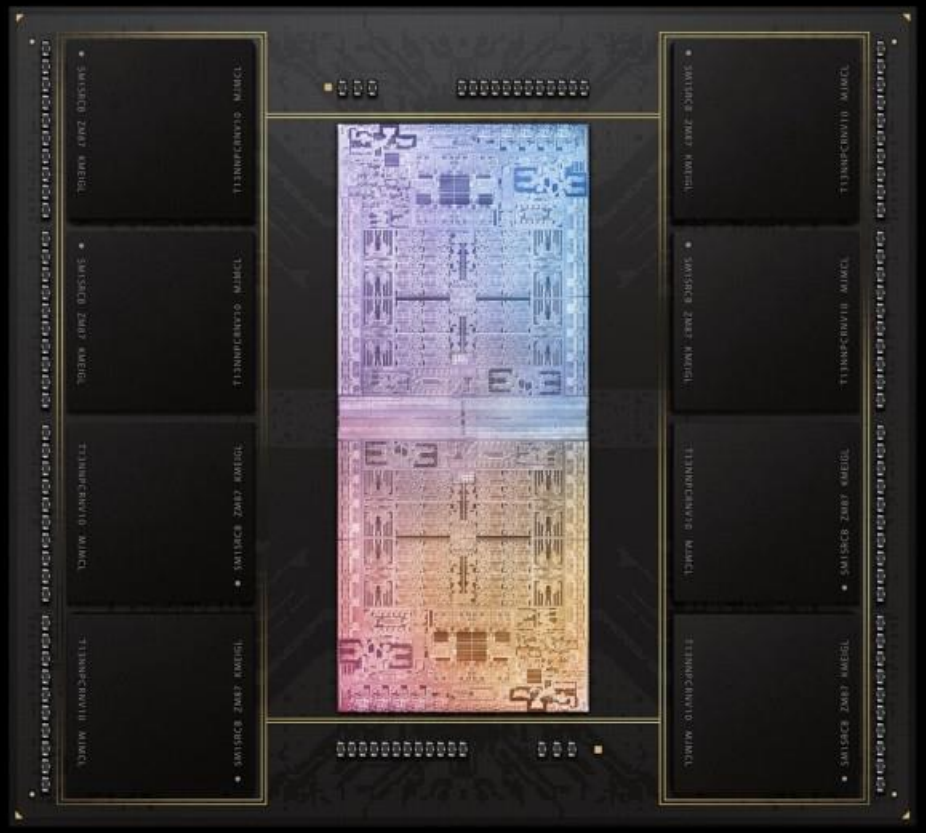
https://www.theverge.com/2022/3/9/22968611/apple-m1-ultra-gpu-nvidia-rtx-3090-comparison

Apple M1 Ultra (2022)



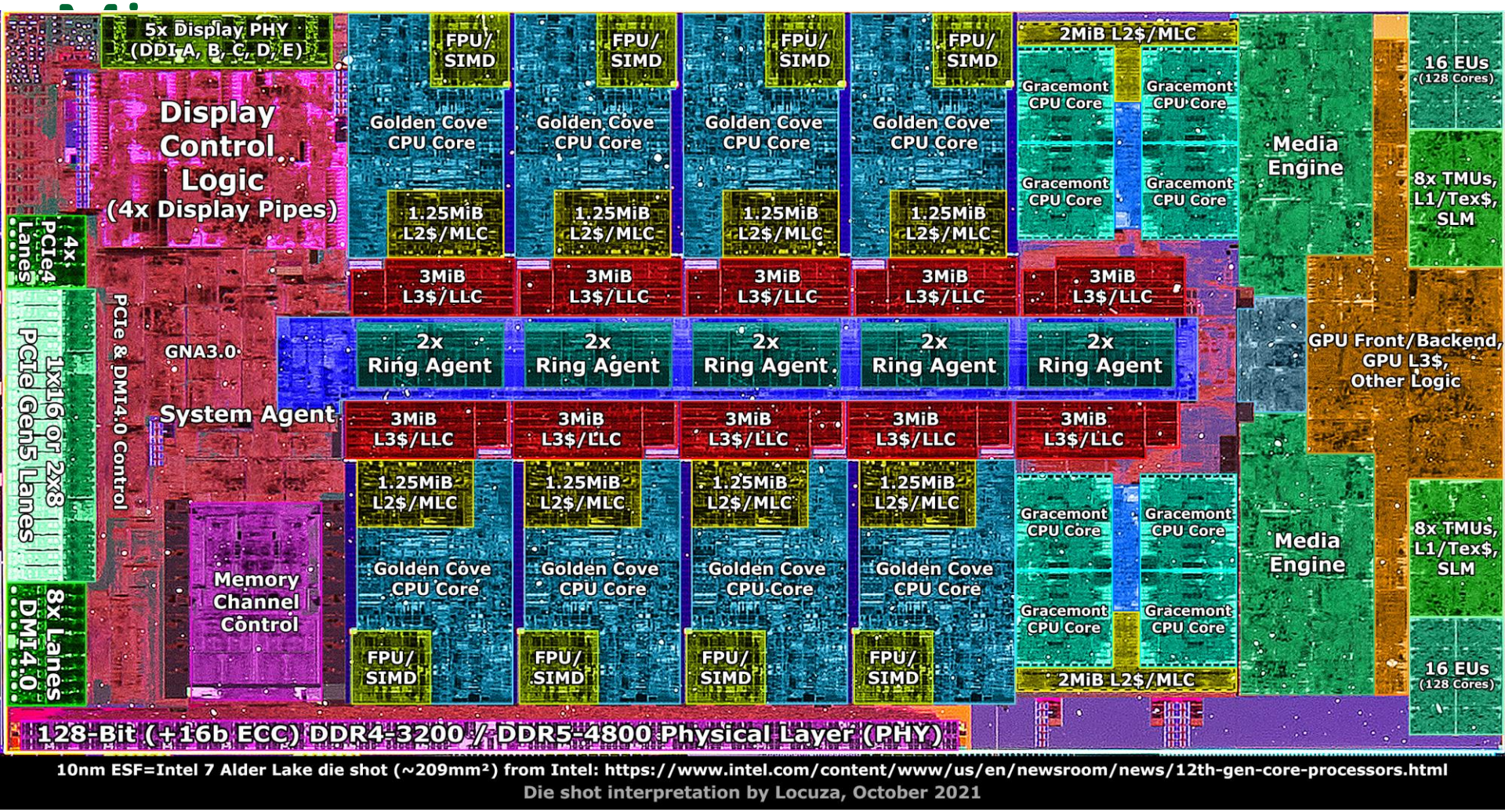
https://stadt-bremerhaven.de/apple-neuer-m1-ultra-chip-ist-offiziell/

Apple M1 Ultra with DRAM (2022)



https://www.gsmarena.com/apple_announces_m1_ultra_with_20core_cpu_and_64core_gpu-news-53481.php

Modern General-Purpose



10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html
Die shot interpretation by Locuza, October 2021

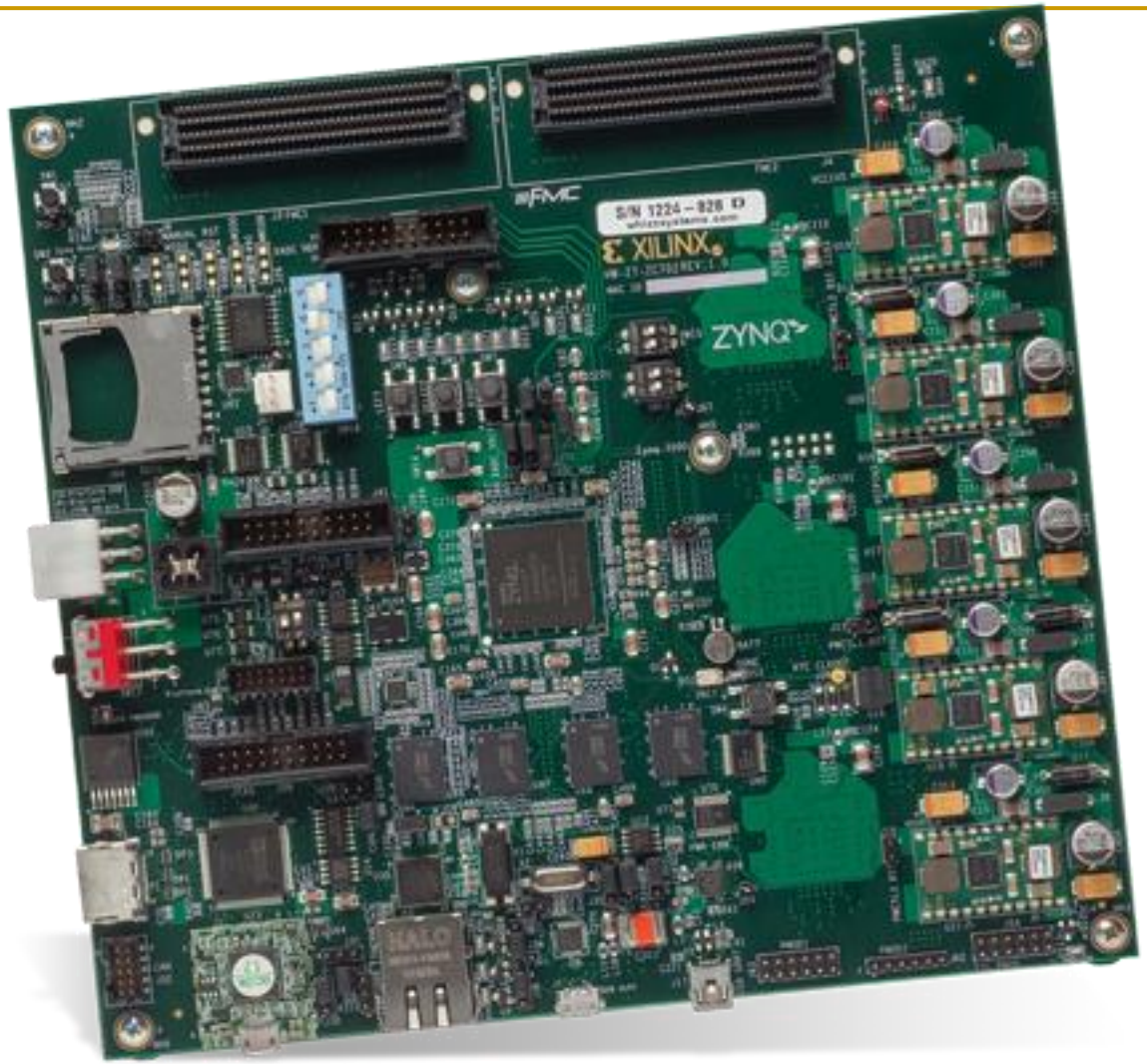
Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

FPGAs

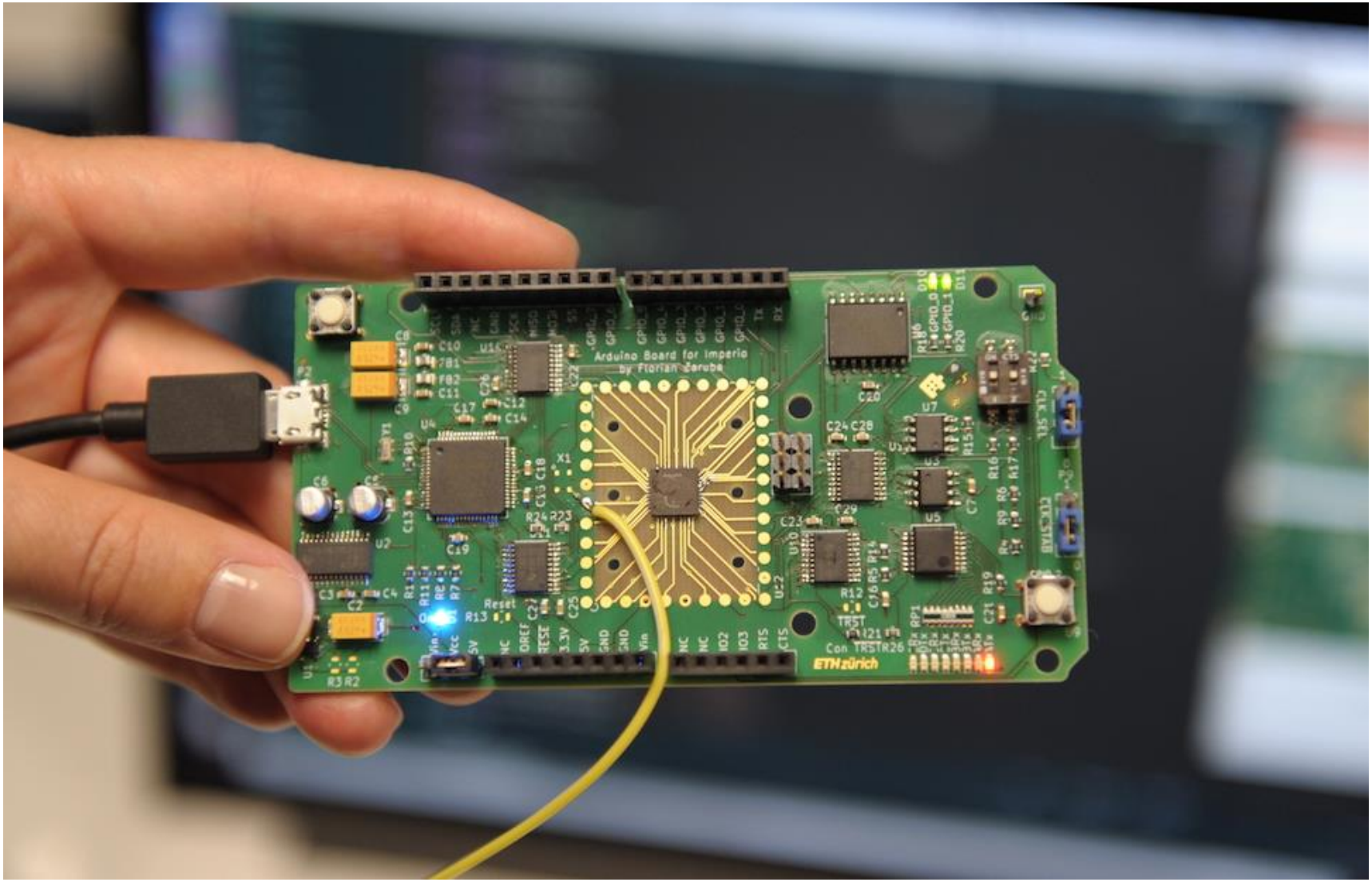

DIGILENT
www.digilentinc.com
XILINX
UNIVERSITY PROGRAM
LINEAR TECHNOLOGY
BASYS 3

Modern FPGAs



Source: https://www.mouser.ch/new/xilinx/xilinx-zynq-7000-zc702-eval-kit/

Special-Purpose ASICs (App-Specific Integrated Circuits)



Modern Special-Purpose ASICs

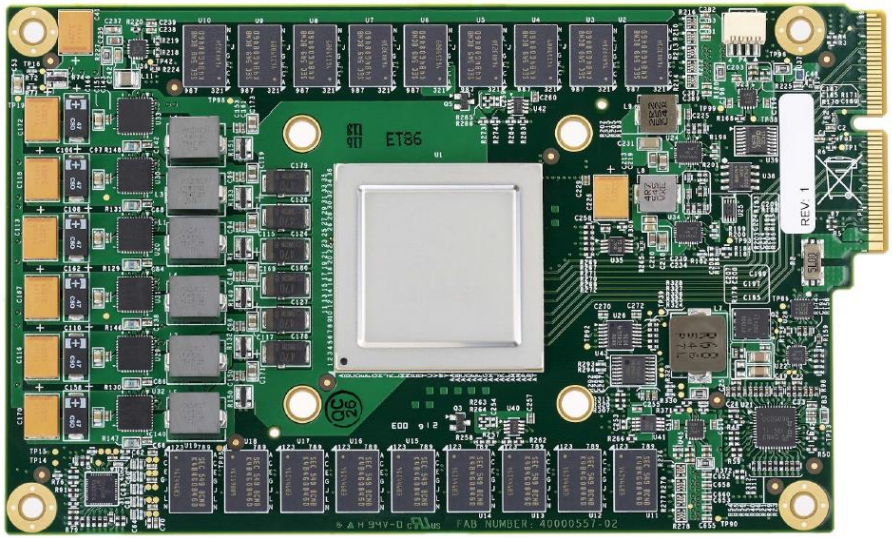


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

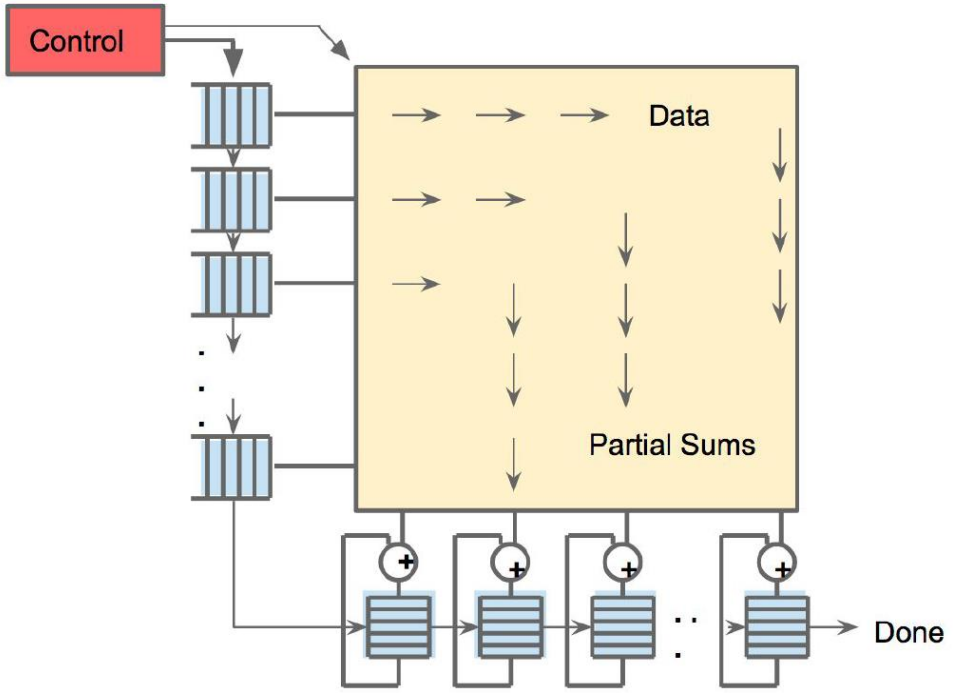
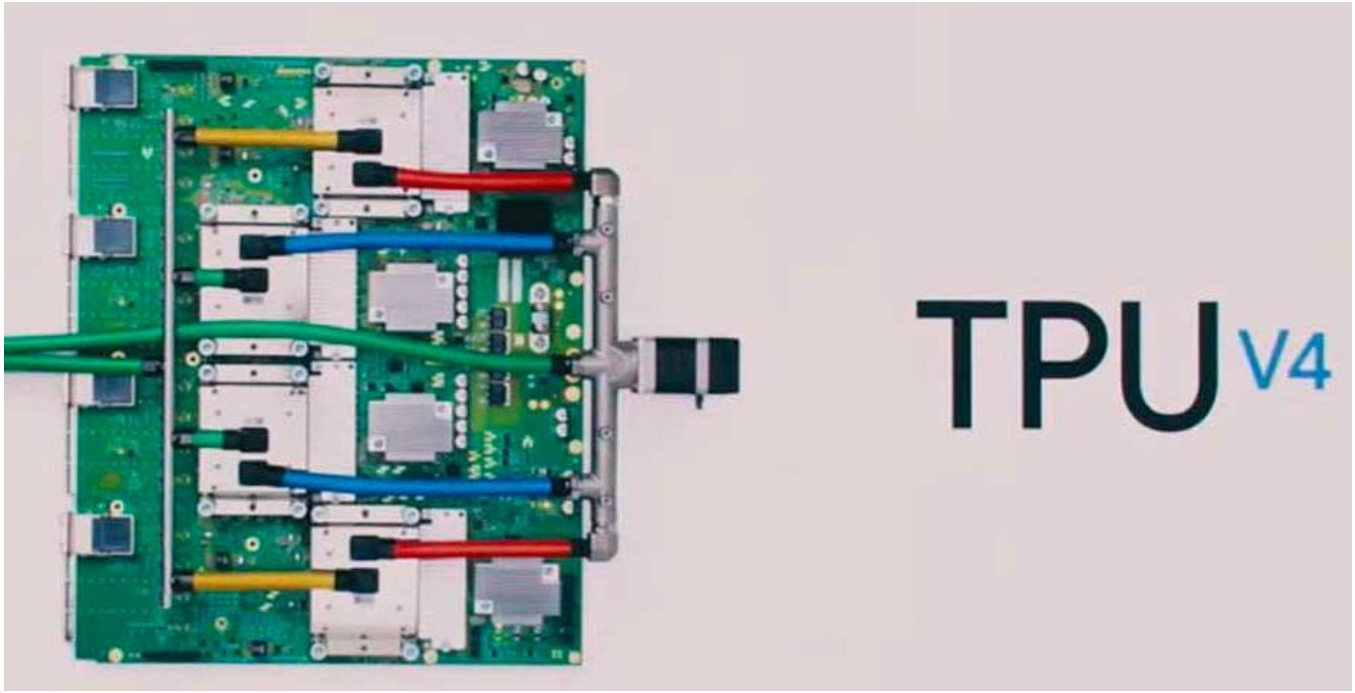


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit", ISCA 2017.

Modern Special-Purpose ASICs



New ML applications (vs. TPU3):

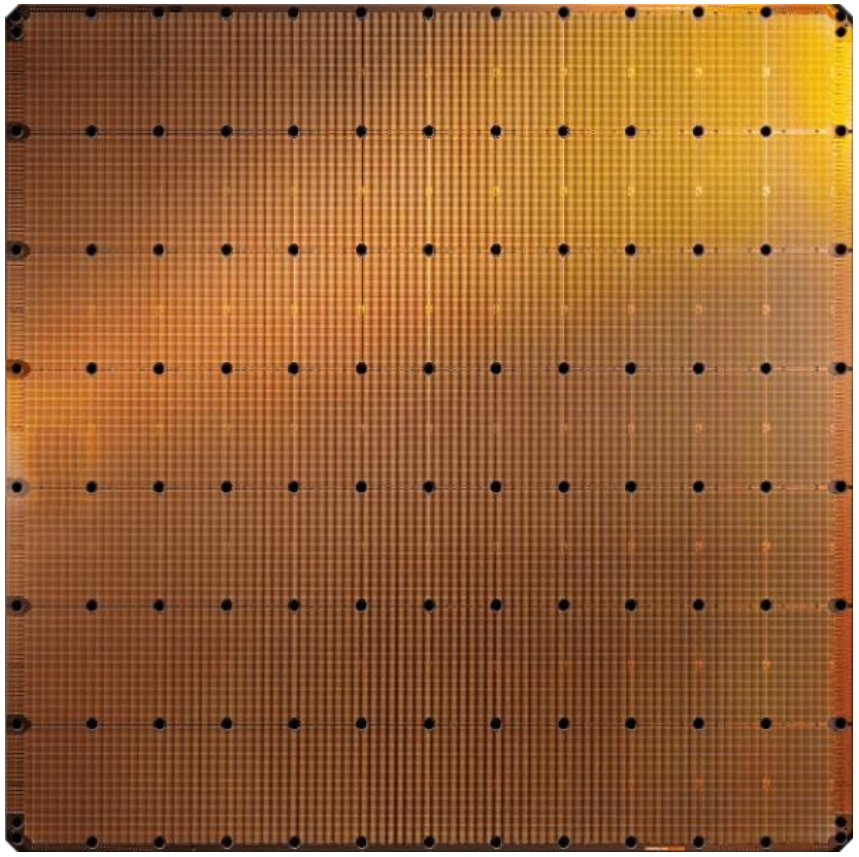
- Computer vision
- Natural Language Processing (NLP)
- Recommender system
- Reinforcement learning that plays Go

250 TFLOPS per chip in 2021
vs 90 TFLOPS in TPU3

1 ExaFLOPS per board

https://spectrum.ieee.org/tech-talk/computing/hardware/heres-how-googles-tpu-v4-ai-chip-stacked-up-in-training-tests

Modern Special-Purpose ASICs



- The largest ML accelerator chip (2021)
- 850,000 cores

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm^2

Largest GPU
54.2 Billion transistors
826 mm^2

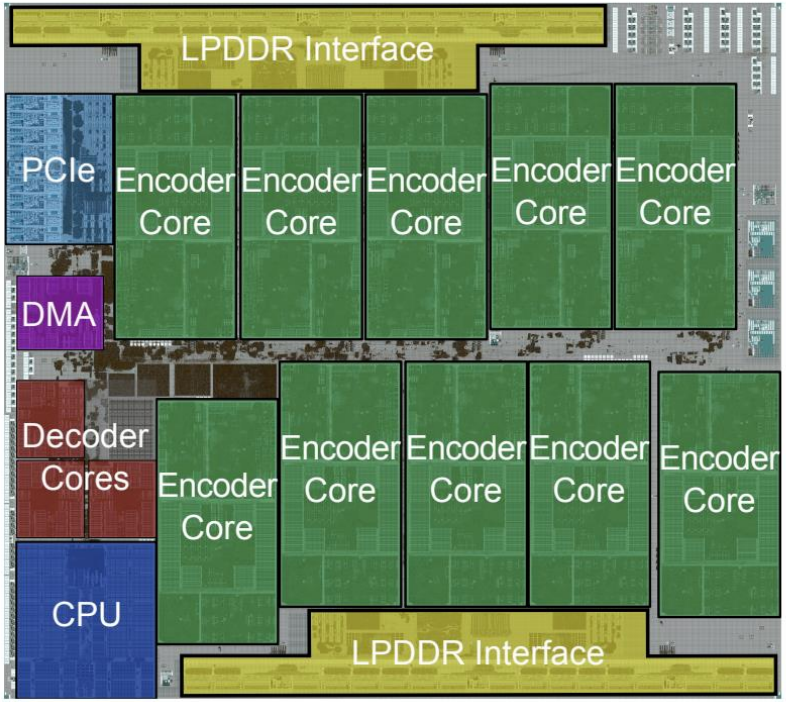
NVIDIA Ampere GA100

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

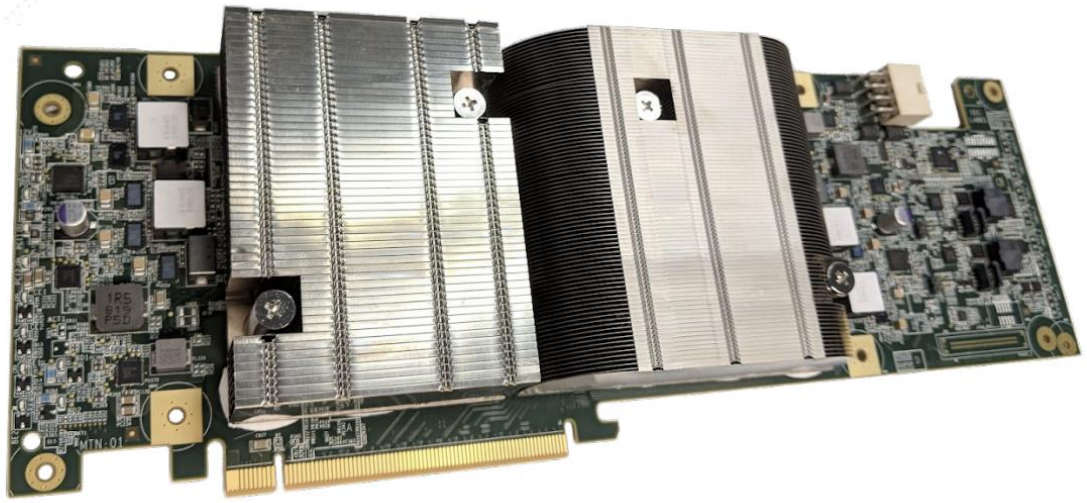
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/

Modern Special-Purpose ASICs

Warehouse-Scale Video Acceleration: Co-design and Deployment in the Wild



(a) Chip floorplan

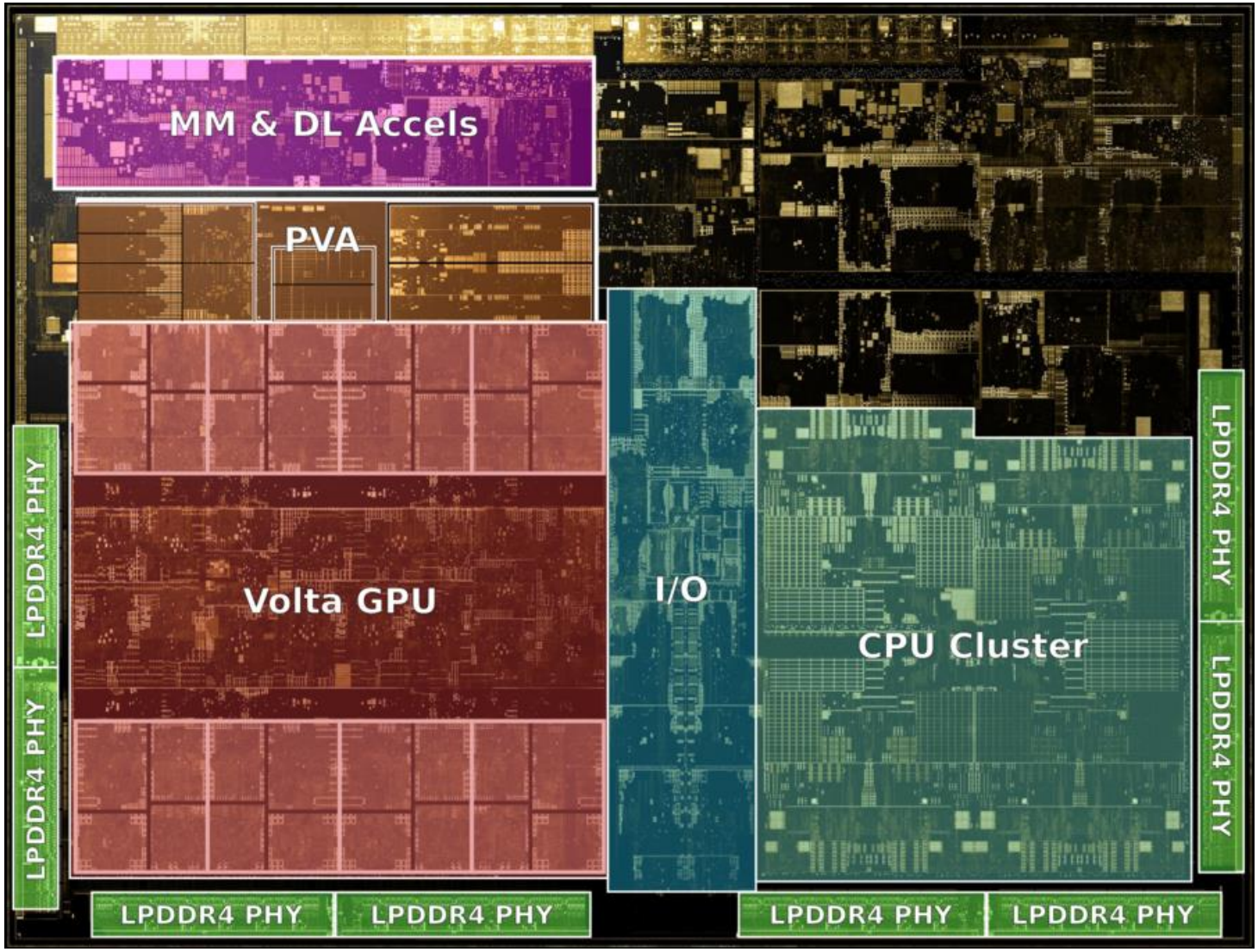


(b) Two chips on a PCBA

Figure 5: Pictures of the VCU

Source: https://dl.acm.org/doi/pdf/10.1145/3445814.3446723

Modern GPUs

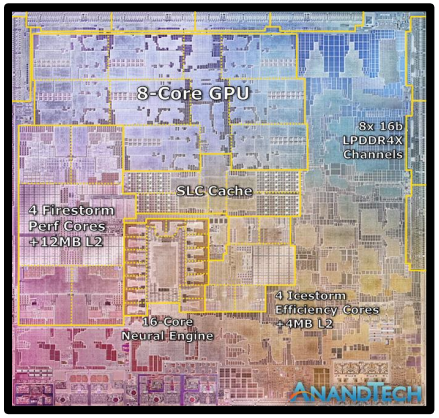


Source: https://en.wikichip.org/wiki/nvidia/tegra/xavier

General Purpose vs. Special Purpose Systems

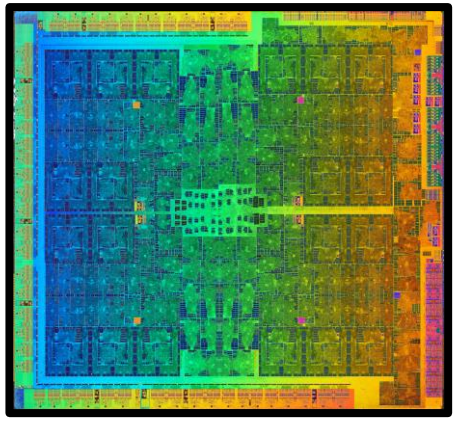
General Purpose | **Special Purpose**

CPUs



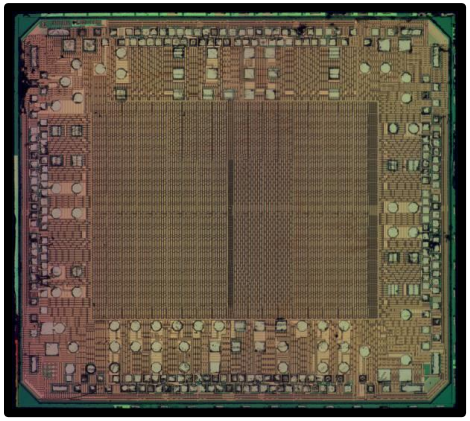
Apple M1

GPUs



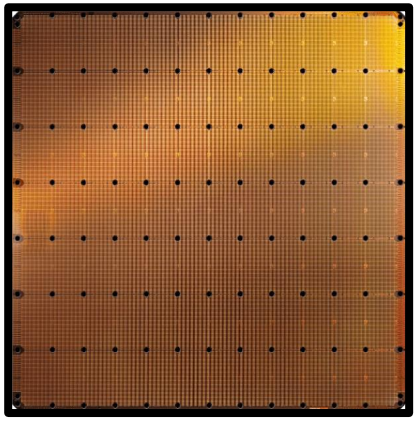
Nvidia GTX 1070

FPGAs



Xilinx Spartan

ASICs



Cerebras WSE-2

Flexible: Can execute any program

Easy to program & use

Not the best performance & efficiency

Efficient & High performance

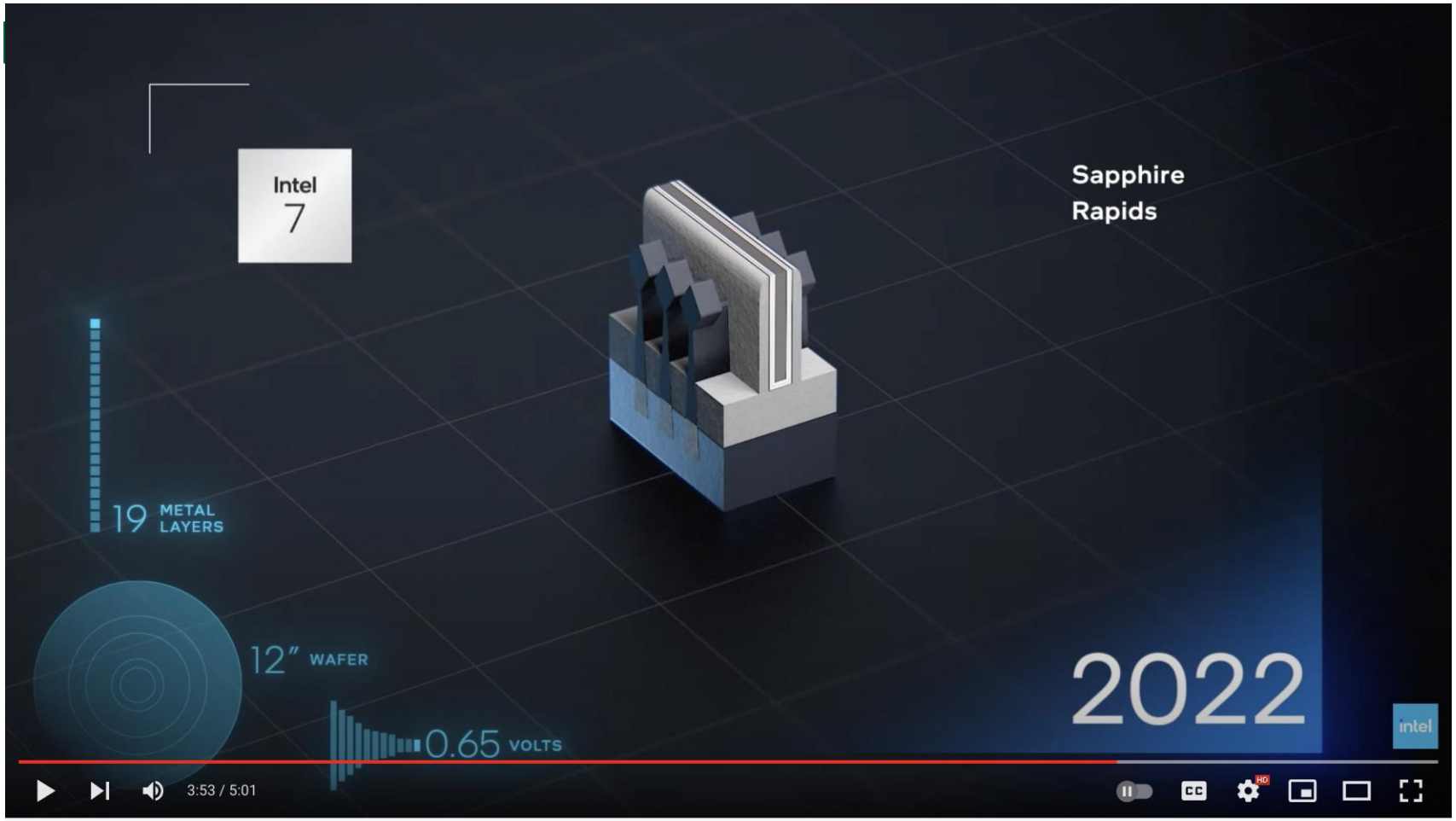
(Usually) Difficult to program & use

Inflexible: Limited set of programs

All Computers are Built Upon the Same Building Blocks

Transistors

A 5-Minute Video on Transistor I



Evolution of Transistor Innovation

12,441 views • Feb 22, 2022

628 DISLIKE SHARE CLIP SAVE ...

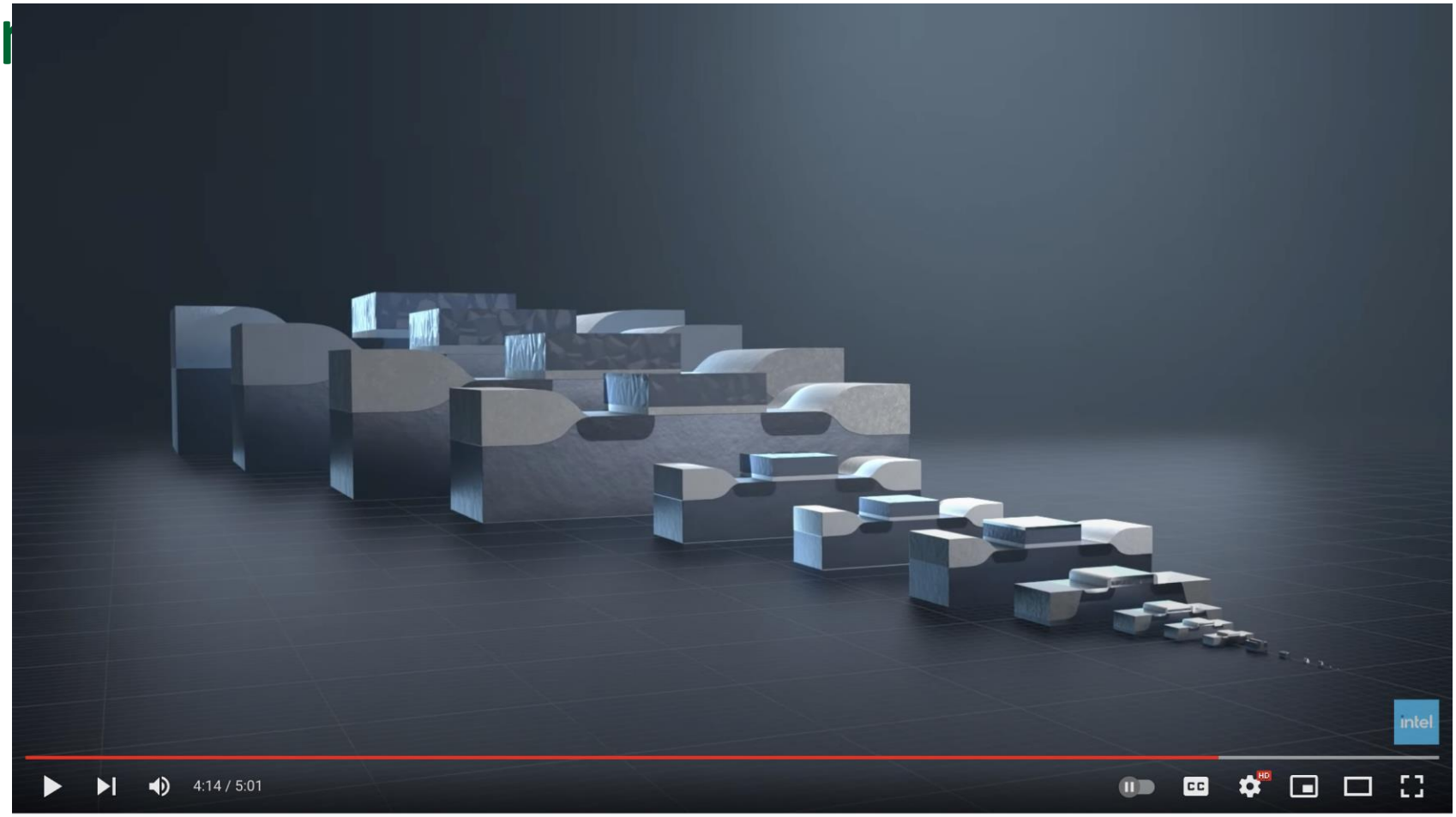
Intel Technology
15.5K subscribers

SUBSCRIBE

https://www.youtube.com/watch?v=Z7M8etXUEUU

A 5-Minute Video on Transistor

I



Evolution of Transistor Innovation

12,460 views • Feb 22, 2022

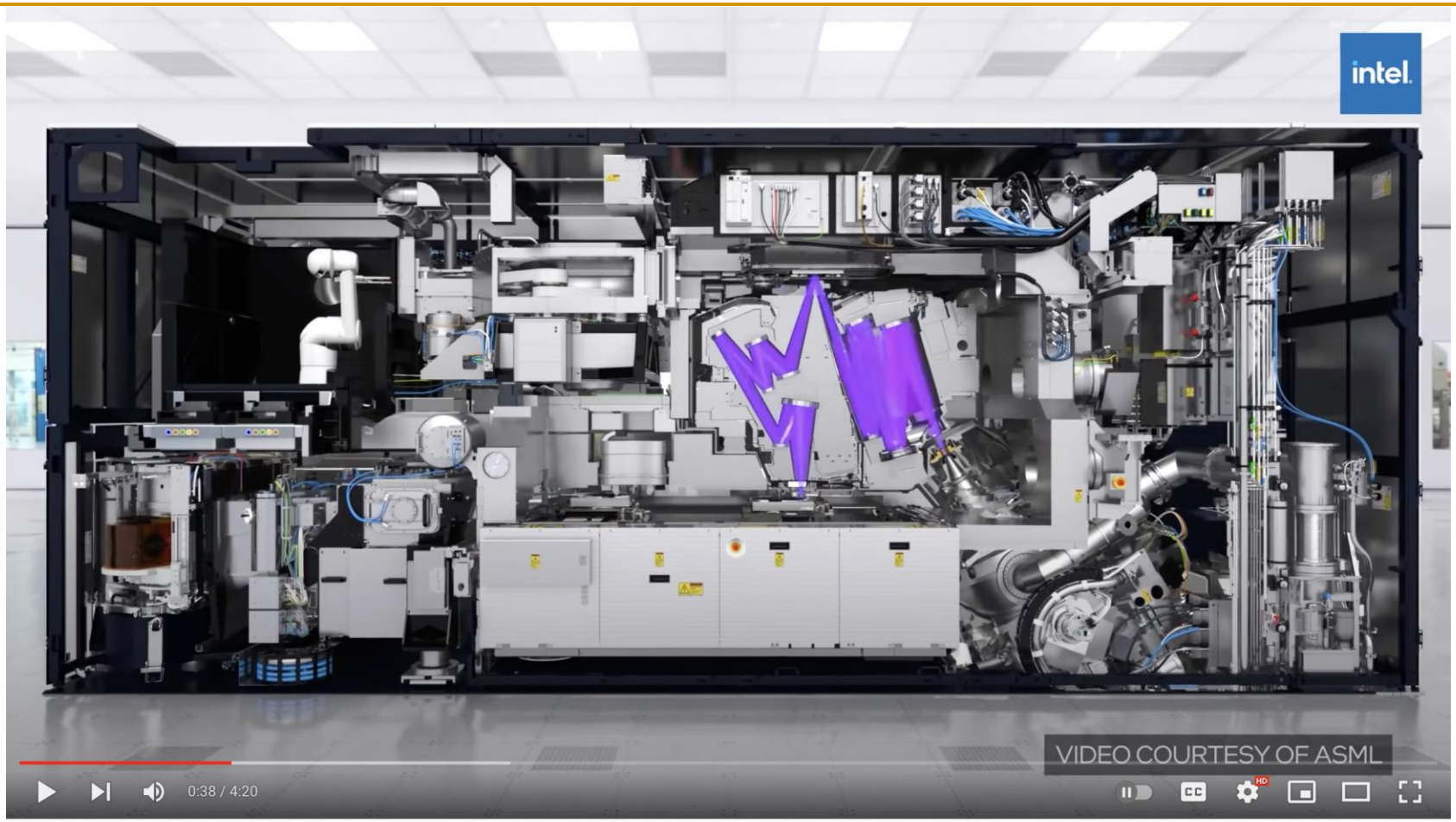
628 DISLIKE SHARE CLIP SAVE ...

intel Intel Technology
15.5K subscribers

SUBSCRIBE

https://www.youtube.com/watch?v=Z7M8etXUEUU

Enabling Manufacturing Tech: EUV



#EUV #chip #Intel

Behind this Door: Learn about EUV, Intel's Most Precise, Complex Machine

78,354 views • Dec 21, 2021

LIKE DISLIKE SHARE CLIP SAVE ...

Intel Newsroom
25.9K subscribers

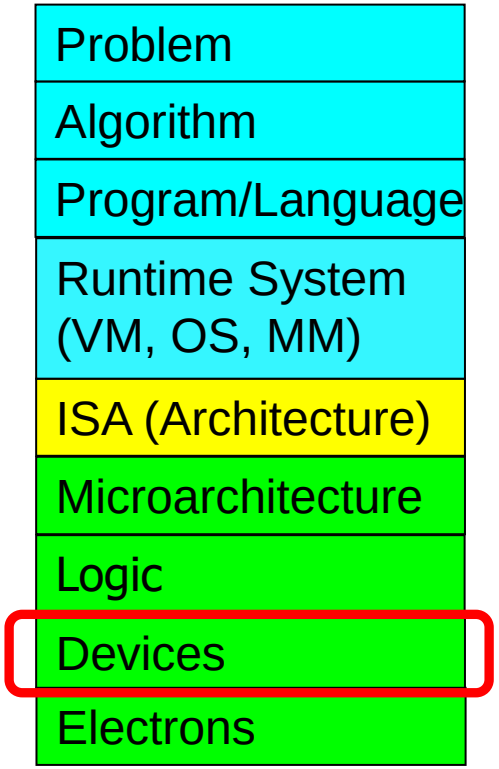
SUBSCRIBE

https://www.youtube.com/watch?v=Jv40Viz-KTc

Logic Gates

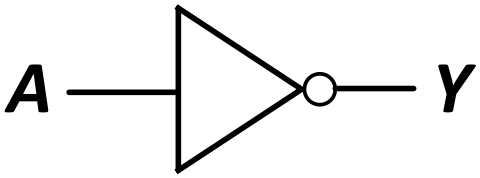
Recall: Transistors to Logic Gates

- **Now, we know how a MOS transistor works**
- How do we build logic structures out of MOS transistors?
- We construct basic logical units out of individual MOS transistors
- These logical units are called logic gates
  - They implement simple Boolean functions

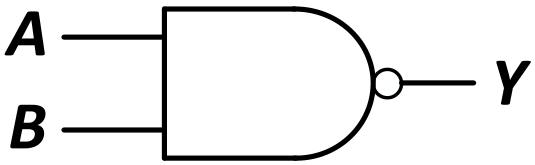


George Boole, "The Mathematical Analysis of Logic," 1847.

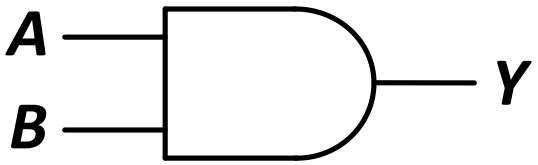
Recall: CMOS NOT, NAND, AND Gates



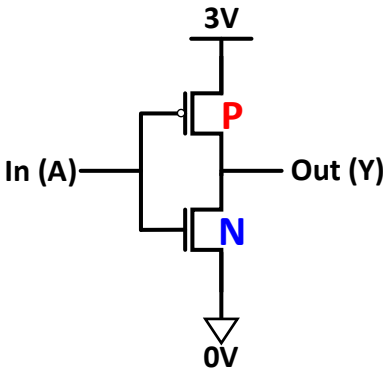
A	Y
0	1
1	0

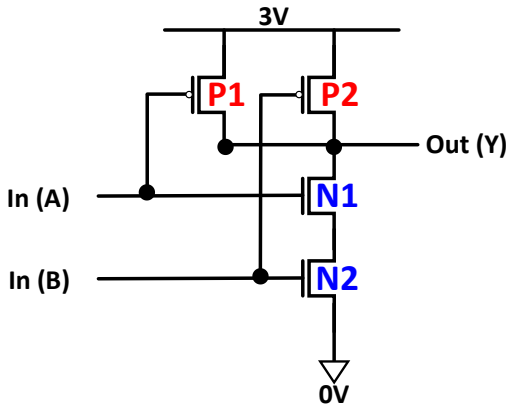


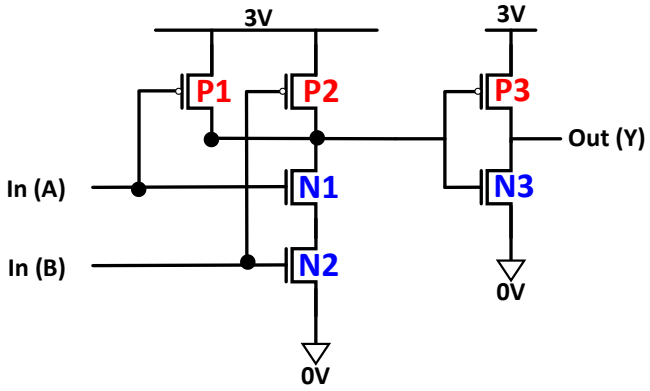
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



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







Recall: Common Logic Gates

Buffer

A	Z
0	0
1	1

AND

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

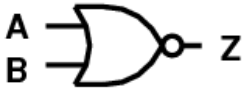
Inverter

A	Z
0	1
1	0

NAND

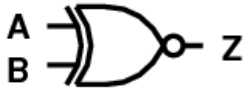
A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

NOR



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

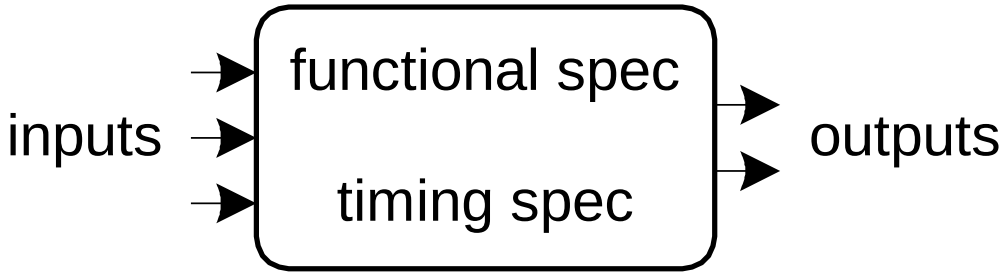
XNOR



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

Combinational Logic Circuits

Recall: Types of Logic Circuits



- **Combinational Logic**
  - Memoryless
  - Outputs are strictly dependent on the combination of input values that are being applied to circuit *right now*
  - In some books called Combinatorial Logic
- **Later we will learn: Sequential Logic**
  - Has memory
    - Structure stores history → Can ”store” data values
  - Outputs are determined by previous (historical) and current values of inputs

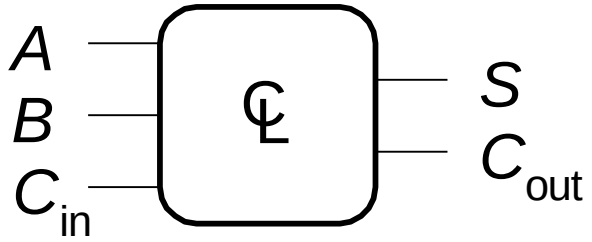
Boolean Logic Equations

Recall: Functional Specification

- Functional specification of outputs in terms of inputs
- What do we mean by "function"?
  - Unique mapping from input values to output values
  - The same input values produce the same output value every time
  - No memory (does not depend on the history of input values)

- ***Example (full 1-bit adder – more later):***

$$S = F(A, B, C_{in})$$
$$C_{out} = G(A, B, C_{in})$$



$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Boolean Equations Enable Us To…

- Represent the function of a combinational logic block
  - Functional Specification
- Methodically transform the function into simpler functions
  - which lead to different hardware realizations
  - Logic Minimization or Logic Simplification
  - We can automate this process → Computer-Aided Design or Electronic Design Automation
- Different Boolean expressions lead to different logic gate implementations

  → Different hardware area, cost, latency, energy properties

Recall: Boolean Algebra: Useful Laws

Dual

Operations with 0 and 1:

1. $X + 0 = X$ — 1D. $X \bullet 1 = X$
2. $X + 1 = 1$ — 2D. $X \bullet 0 = 0$

AND, OR with identities gives you back the original variable or the identity

Idempotent Law:

3. $X + X = X$ — 3D. $X \bullet X = X$

AND, OR with self = self

Involution Law:

4. $\overline{(\overline{X})} = X$

double complement = no complement

Laws of Complementarity:

5. $X + \overline{X} = 1$ — 5D. $X \bullet \overline{X} = 0$

AND, OR with complement gives you an identity

Commutative Law:

6. $X + Y = Y + X$ — 6D. $X \bullet Y = Y \bullet X$

Just an axiom…

Recall: Useful Laws (continued)

Associative Laws:

7. $(X + Y) + Z = X + (Y + Z)$
   $= X + Y + Z$

7D. $(X \bullet Y) \bullet Z = X \bullet (Y \bullet Z)$
$= X \bullet Y \bullet Z$

Parenthesis order does not matter

Distributive Laws:

8. $X \bullet (Y + Z) = (X \bullet Y) + (X \bullet Z)$

8D. $X + (Y \bullet Z) = (X + Y) \bullet (X + Z)$

Axiom

Simplification Theorems:

9. $X \bullet Y + X \bullet \overline{Y} = X$

9D. $(X + Y) \bullet (X + \overline{Y}) = X$

10. $X + X \bullet Y = X$

10D. $X \bullet (X + Y) = X$

11. $(X + \overline{Y}) \bullet Y = X \bullet Y$

11D. $(X \bullet \overline{Y}) + Y = X + Y$

Useful for simplifying expressions

Actually worth remembering — they show up a lot in real designs…

DeMorgan's Law: Enabling Transformations

DeMorgan's Law:

12. $\overline{(X + Y + Z + \cdots)} = \overline{X}.\overline{Y}.\overline{Z}. \ldots$

12D. $\overline{(X . Y . Z . \ldots)} = \overline{X} + \overline{Y} + \overline{Z} + \ldots$

- **Think of this as a transformation**
  - Let's say we have:

$$F = A + B + C$$

  - Applying DeMorgan's Law (12), gives us

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

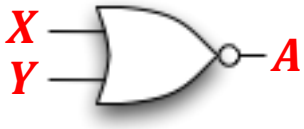
At least one of A, B, C is TRUE --> It is **not** the case that A, B, C are **all** false

DeMorgan's Law (Continued)

These are conversions between different types of logic functions
They can prove useful if you do not have every type of gate…
Or, if some types of gates are more desirable to use than others…

$$A = \overline{(X + Y)} = \overline{X}\,\overline{Y}$$

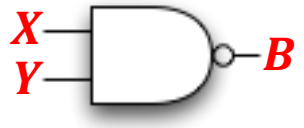
NOR is equivalent to AND with inputs complemented



X	Y	$\overline{X + Y}$	$\overline{X}$	$\overline{Y}$	$\overline{X}\,\overline{Y}$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

$$B = \overline{(XY)} = \overline{X} + \overline{Y}$$

NAND is equivalent to OR with inputs complemented



X	Y	$\overline{XY}$	$\overline{X}$	$\overline{Y}$	$\overline{X} + \overline{Y}$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Using Boolean Equations to Represent a Logic Circuit

Standardized Function Representations

- Enable a single, universally-agreed-on way of representing a Boolean function starting from its truth table
  - Also called "canonical representations"

- Sum of Products (SOP) form

- Product of Sums (POS) form

Sum of Products Form: Key Idea

- Assume **we have the truth table of Boolean Function F**
- How do we express the function in terms of the inputs in a **standard** manner?
- Idea: **Sum of Products** form
- Express the truth table as a two-level Boolean expression
  - that contains **all** input variable combinations that result in a 1 output
  - If ANY of the combinations of input variables that results in a 1 is TRUE, then the output is 1
  - F = OR of all input variable combinations that result in a 1

Some Definitions (for a 3-Input Function)

- **Complement:** variable with a bar over it
  $\overline{A}, \overline{B}, \overline{C}$
- **Literal:** variable or its complement
  $A, \overline{A}, B, \overline{B}, C, \overline{C}$
- **Implicant:** product (AND) of literals
  $(A \cdot B \cdot \overline{C})$, $(\overline{A} \cdot C)$, $(B \cdot \overline{C})$
- **Minterm:** product (AND) that includes **all** input variables
  $(A \cdot B \cdot \overline{C})$, $(\overline{A} \cdot \overline{B} \cdot C)$, $(\overline{A} \cdot B \cdot \overline{C})$
- **Maxterm:** sum (OR) that includes **all** input variables
  $(A + \overline{B} + \overline{C})$, $(\overline{A} + B + \overline{C})$, $(A + B + \overline{C})$

Two-Level Canonical (Standard) Forms

- Truth table is the unique signature of a Boolean *function* …
  - But, it is an expensive representation

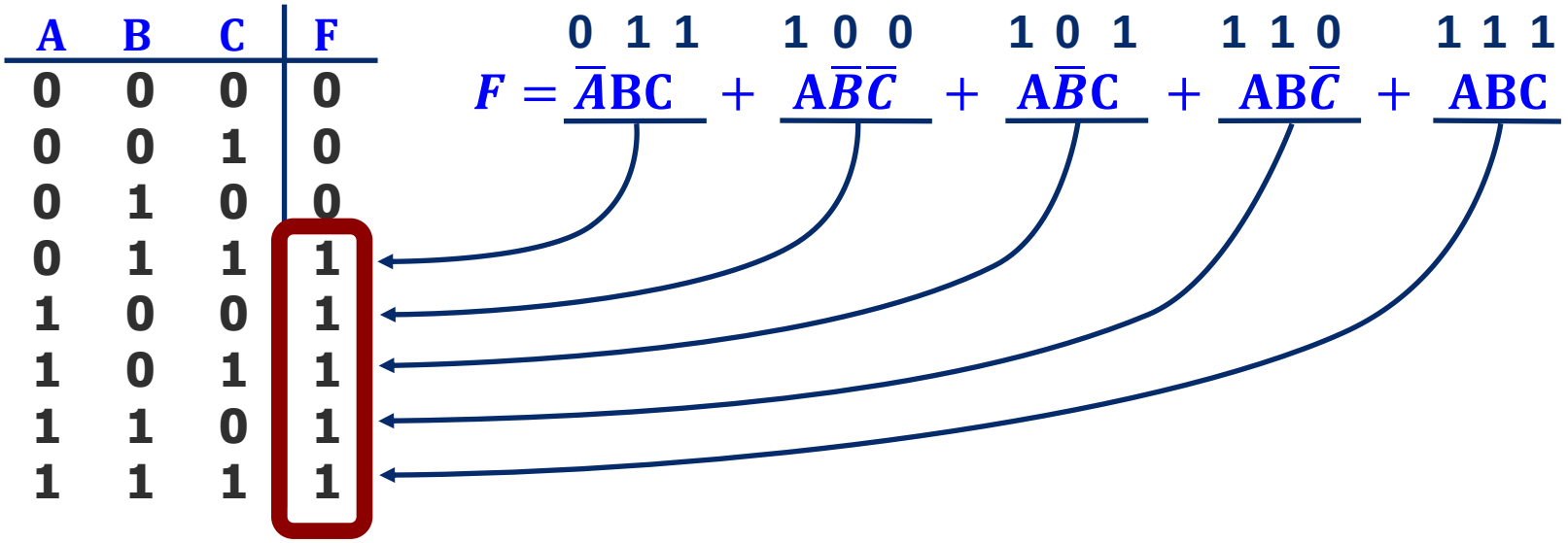
- A Boolean function can have many alternative Boolean expressions
  - i.e., many alternative Boolean expressions (and gate realizations) may have the same truth table (and function)
  - If they all specify the same thing, why do we care?
    - Different Boolean expressions lead to different logic gate implementations → Different cost, latency, energy properties

- Canonical form: standard form for a Boolean expression
  - Provides a unique algebraic signature

Two-Level Canonical Forms: SOP

Sum of Products Form (SOP)

Also known as **disjunctive normal form** or **minterm expansion**

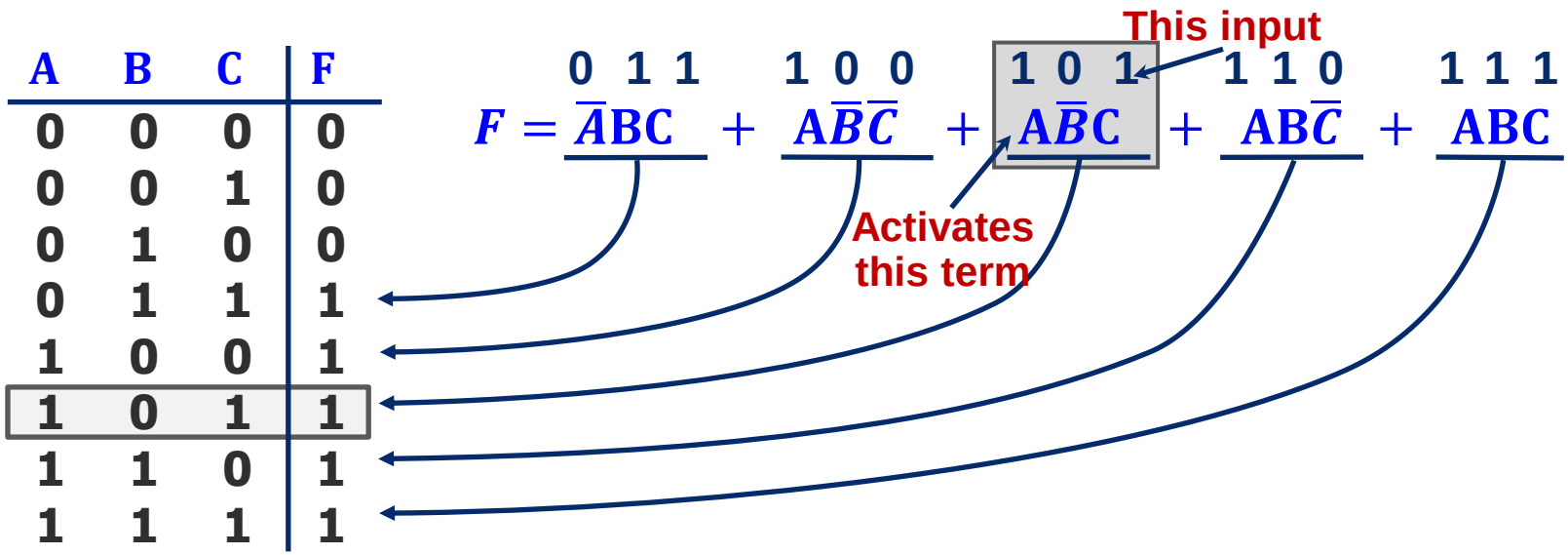


- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

All Boolean equations can be written in SOP form

Find all the input combinations (minterms) for which the output of the function is TRUE.

SOP Form — Why Does It Work?



- Only the shaded product term — $A\bar{B}C = 1 \cdot \bar{0} \cdot 1$ — will be 1
- No other product terms will "turn on" — they will all be 0
- So if inputs A B C correspond to a product term in expression,
  - We get 0 + 0 + … + 1 + … + 0 + 0 = 1 for output
- If inputs A B C do not correspond to any product term in expression
  - We get 0 + 0 + … + 0 = 0 for output

The function evaluates to TRUE (i.e., output is 1)
if **any** of the **Products** (minterms) causes the output to be 1

Standard Notation for SOP Form

- Standard "shorthand" notation
  - If we agree on the order of the variables in the rows of truth table…
    - then we can enumerate each row with the decimal number that corresponds to the binary number created by the input pattern

A	B	C	F	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	
1	0	0	1	**100 = decimal 4 so this is minterm #4, or m4**
1	0	1	1	
1	1	0	1	
1	1	1	1	**111 = decimal 7 so this is minterm #7, or m7**

f =

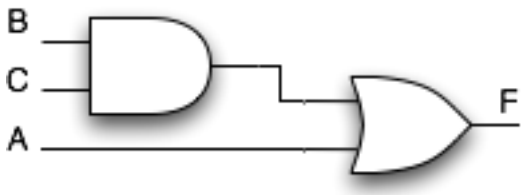
We can write this as a sum of products

Or, we can use a summation notation

Canonical SOP Form

A	B	C	minterms
0	0	0	$\bar{A}\bar{B}\bar{C}$ = m0
0	0	1	$\bar{A}\bar{B}C$ = m1
0	1	0	$\bar{A}B\bar{C}$ = m2
0	1	1	$\bar{A}BC$ = m3
1	0	0	$A\bar{B}\bar{C}$ = m4
1	0	1	$A\bar{B}C$ = m5
1	1	0	$AB\bar{C}$ = m6
1	1	1	ABC = m7

Shorthand Notation for Minterms of 3 Variables



2-Level AND/OR Realization

F in canonical form:

F(A,B,C) = ∑m(3,4,5,6,7)

= m3 + m4 + m5 + m6 + m7

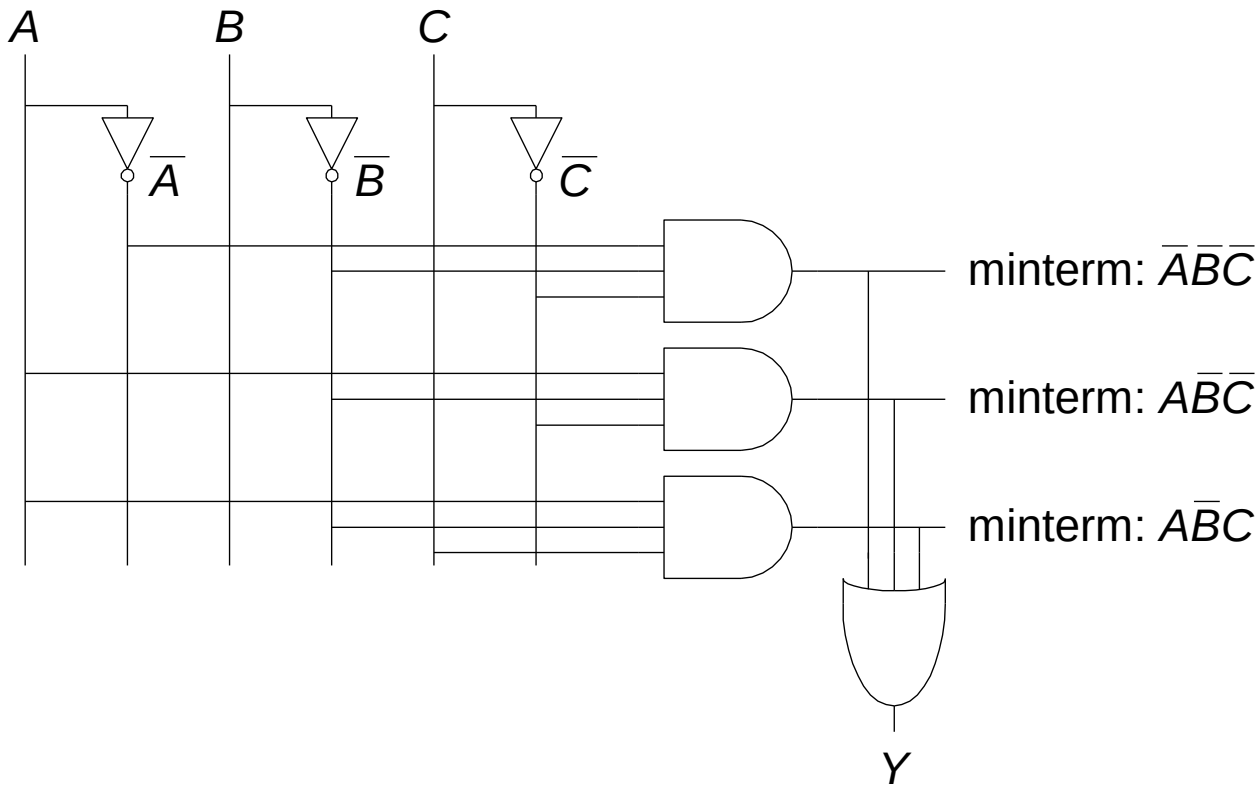
$F =$

canonical form ≠ minimal form

F

From SOP to Gates

- **SOP (sum-of-products) leads to two-level logic**
- Example: $Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C)$



SOP form does NOT directly lead to minimal logic

Canonical Sum of Products Form: Key Idea

- Any 1-bit function can be represented as a Sum of Products
- A "Product" is the Boolean AND that includes ALL input variables of the function → minterm
- The 1-bit Output of the Function can be represented as
  - Sum (OR) of all minterms that lead to a 1 in the Output
- Logically
  - The function evaluates to TRUE (i.e., output is 1) if ANY of the Products (minterms) causes the Output to be 1
  - SOP form represents the function as the SUM (OR) of all Products (minterms) that cause the Output to be 1

Alternative Canonical Form: POS

DeMorgan of SOP of $\overline{F}$

Find all the input combinations (maxterms) for which the output of the function is FALSE.

Product of Sums (POS)

product

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

sums

This input

Each sum term represents one of the "**zeros**" of the function

$$F = \underset{0\ 0\ 0}{(A + B + C)}\ \underset{0\ 0\ 1}{(A + B + \overline{C})}\ \underset{0\ 1\ 0}{(A + \overline{B} + C)}$$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Activates this term

For the given input, only the shaded sum term will equal 0

$$A + \overline{B} + C = 0 + \overline{1} + 0$$

Anything ANDed with 0 is 0; Output F will be 0

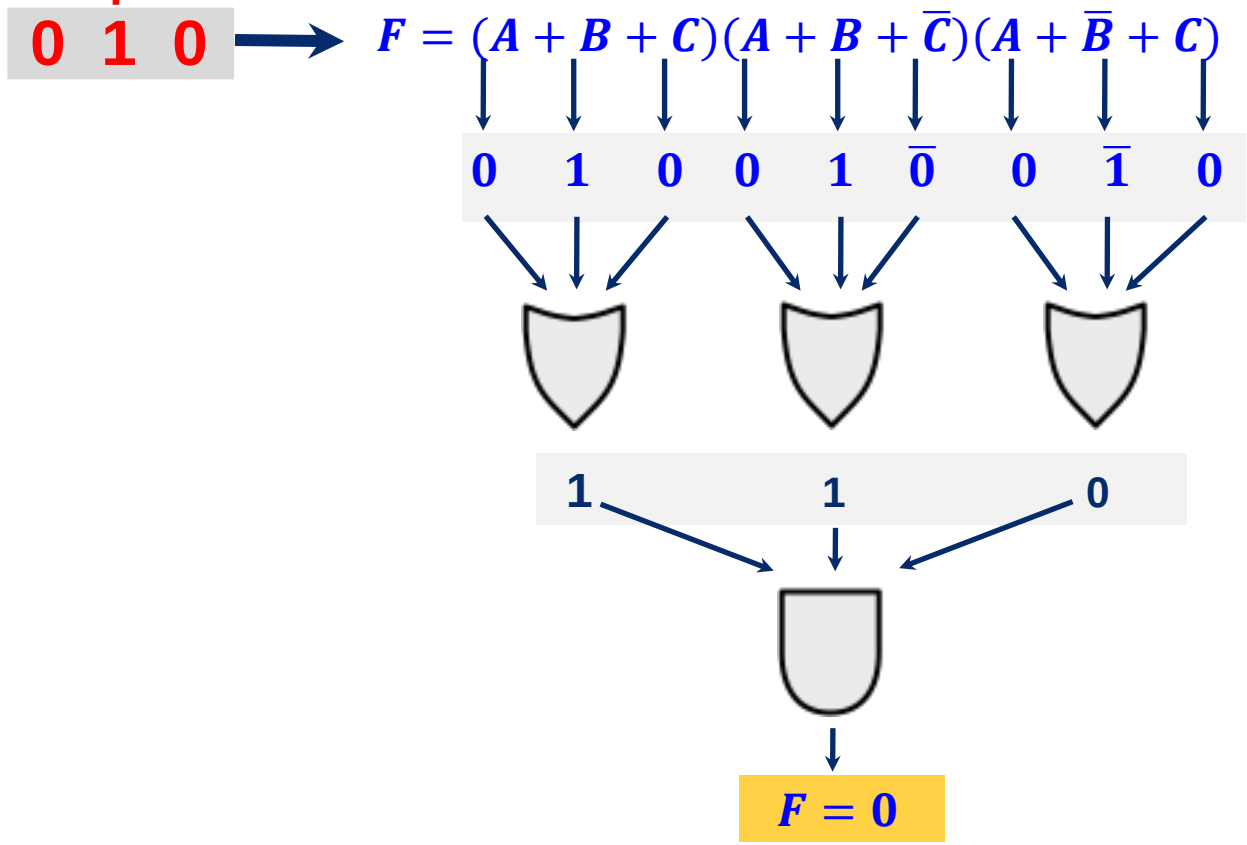
The function evaluates to FALSE (i.e., output is 0)
if **any** of the Sums (maxterms) causes the output to be 0

Consider A=0, B=1, C=0

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Input

0 1 0



Only one of the products will be 0, anything ANDed with 0 is 0

Therefore, the output is F = 0

POS: How to Write It

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

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

$A \quad \overline{B} \quad C$

$A + \overline{B} + C$

Maxterm form:

1. Find truth table rows where F is 0

2. 0 in input col → true literal
3. 1 in input col → complemented literal

4. OR the literals to get a Maxterm
5. AND together all the Maxterms

Or just remember" POS of F is the same as the DeMorgan of SOP of $\overline{F}$

Notation for the Canonical POS Form

Product of Sums / Conjunctive Normal Form / Maxterm Expansion

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

$$\prod M(0, 1, 2)$$

A	B	C	Maxterms	
0	0	0	$A + B + C$	= M0
0	0	1	$A + B + \overline{C}$	= M1
0	1	0	$A + \overline{B} + C$	= M2
0	1	1	$A + \overline{B} + \overline{C}$	= M3
1	0	0	$\overline{A} + B + C$	= M4
1	0	1	$\overline{A} + B + \overline{C}$	= M5
1	1	0	$\overline{A} + \overline{B} + C$	= M6
1	1	1	$\overline{A} + \overline{B} + \overline{C}$	= M7

Maxterm shorthand notation for a function of three variables

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Note that you form the maxterms around the "zeros" of the function

This is not the complement of the function!

Useful Conversions

1. **Minterm to Maxterm conversion:**
   rewrite minterm shorthand using maxterm shorthand
   replace minterm indices with the indices not already used
   E.g., $\mathbf{F}(A,B,C) = \sum m(3,4,5,6,7) = \prod M(0,1,2)$

2. **Maxterm to Minterm conversion:**
   rewrite maxterm shorthand using minterm shorthand
   replace maxterm indices with the indices not already used
   E.g., $\mathbf{F}(A,B,C) = \prod M(0,1,2) = \sum m(3,4,5,6,7)$

3. **Expansion of** $\mathbf{F}$ **to expansion of** $\overline{F}$ **:**

$$\mathbf{E.g.,}\ \mathbf{F}(A,B,C) = \sum m(3,4,5,6,7) \longrightarrow \overline{F}(A,B,C) = \sum m(0,1,2)$$
$$= \prod M(0,1,2) \longrightarrow = \prod M(3,4,5,6,7)$$

4. **Minterm expansion of** $\mathbf{F}$ **to Maxterm expansion of** $\overline{F}$ **:**
   rewrite in Maxterm form, using the same indices as $\mathbf{F}$

$$\mathbf{E.g.,}\ \mathbf{F}(A,B,C) = \sum m(3,4,5,6,7) \longrightarrow \overline{F}(A,B,C) = \prod M(3,4,5,6,7)$$
$$= \prod M(0,1,2) \longrightarrow = \sum m(0,1,2)$$

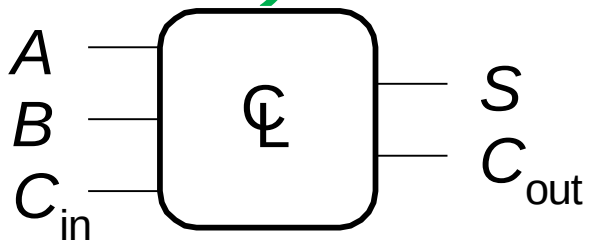
Logic Simplification (or Minimization)

- Using Boolean Algebra, we can simplify the SOP or POS form of any function in a methodical way

- Starting with the canonical SOP or POS form enables convenience and automation
  - Truth table → SOP/POS form → Boolean Simplification Rules

- ***Example (full 1-bit adder – more later):***

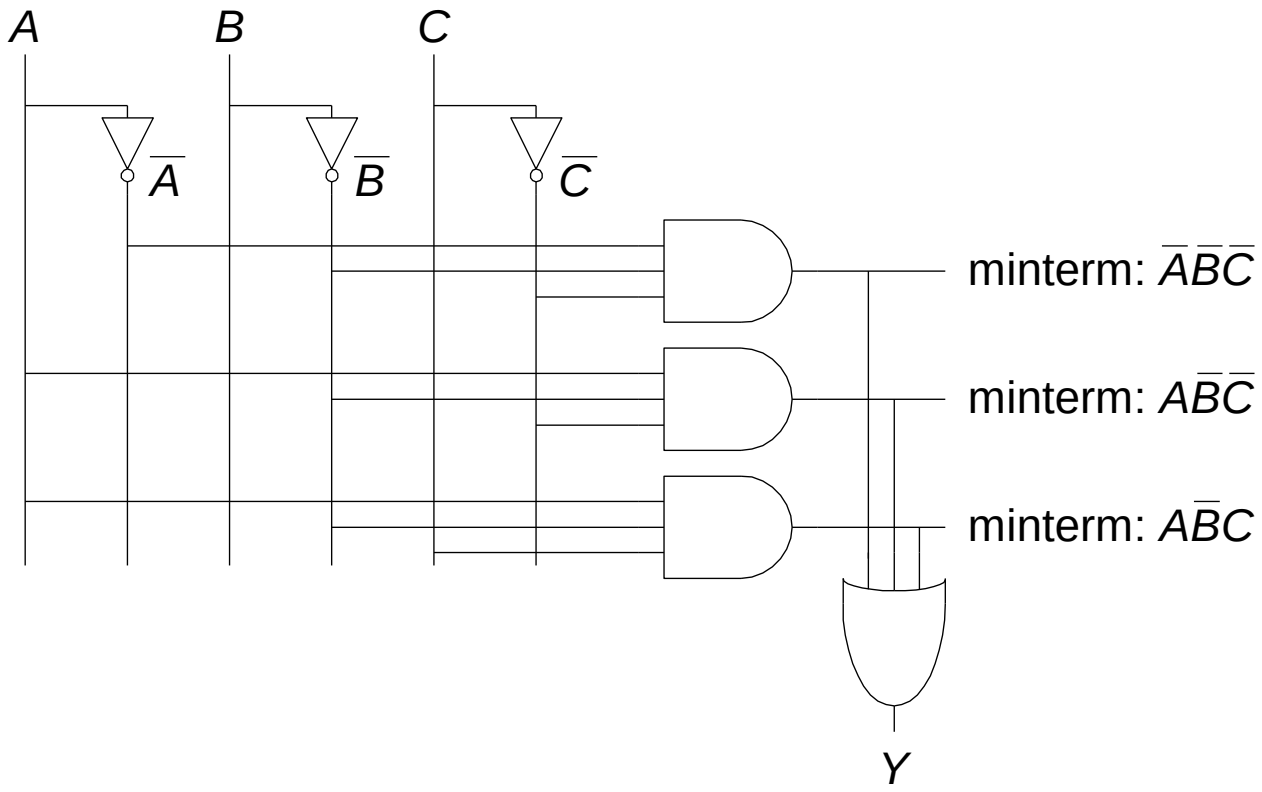
$$S = F(A, B, C_{in})$$
$$C_{out} = G(A, B, C_{in})$$



$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Logic Simplification Example: SOP Form

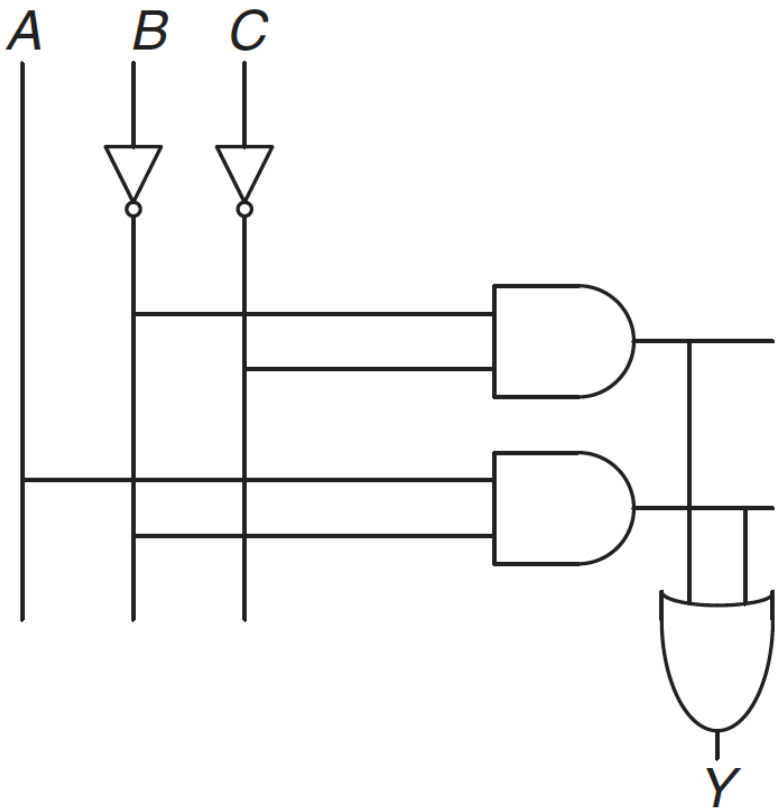
- **SOP (sum-of-products) form of function Y**
- Example: $Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C)$



SOP form does NOT directly lead to minimal logic

Logic Simplification Example: Simplified

- **SOP (sum-of-products) form of function Y**
- Example: $Y = (\overline{B} \cdot \overline{C}) + (A \cdot \overline{B})$



Let's Cover Some Basic Combinational Blocks

Combinational Building Blocks used in Modern Computers

Recall: Common Logic Gates

Buffer

A	Z
0	0
1	1

AND

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

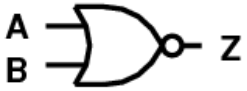
Inverter

A	Z
0	1
1	0

NAND

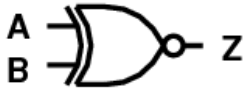
A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

NOR



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

XNOR



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

Combinational Building Blocks

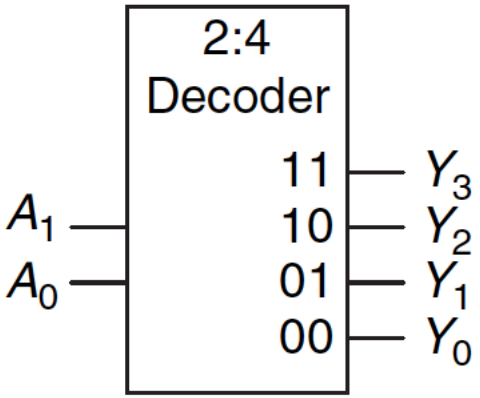
- Combinational logic is often grouped into larger building blocks to build more complex systems
- Hides the unnecessary gate-level details to emphasize the function of the building block
- We now examine:
  - Decoder
  - Multiplexer
  - Full adder
  - PLA (Programmable Logic Array)

Decoder

Decoder

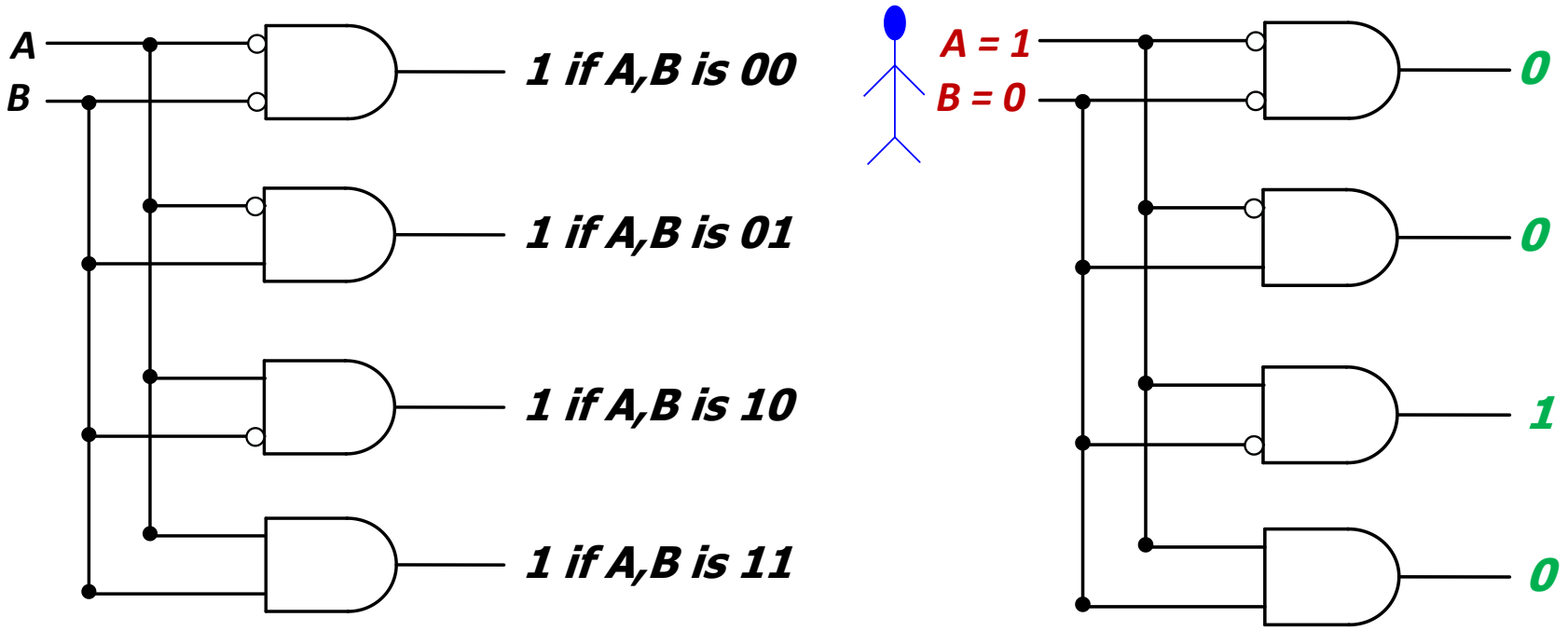
- "Input pattern detector"
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



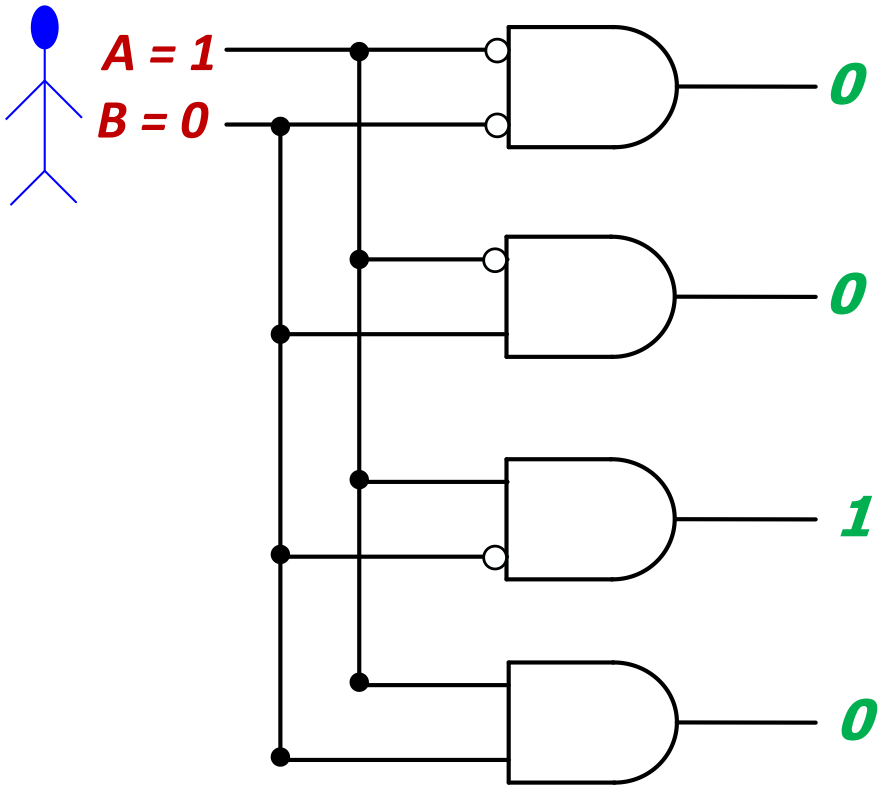
Decoder (I)

- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect



Decoder (II)

- The decoder is useful in determining how to interpret a bit pattern
  - **It could be the address of a location in memory, that the processor intends to read from**
  - **It could be an instruction in the program and the processor needs to decide what action to take (based on *instruction opcode*)**

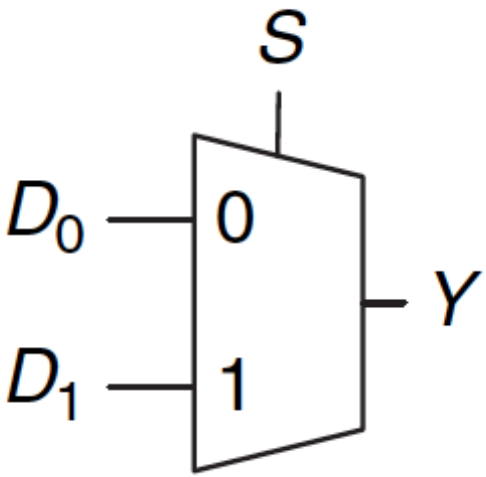


Multiplexer (MUX)

Multiplexer (MUX), or Selector

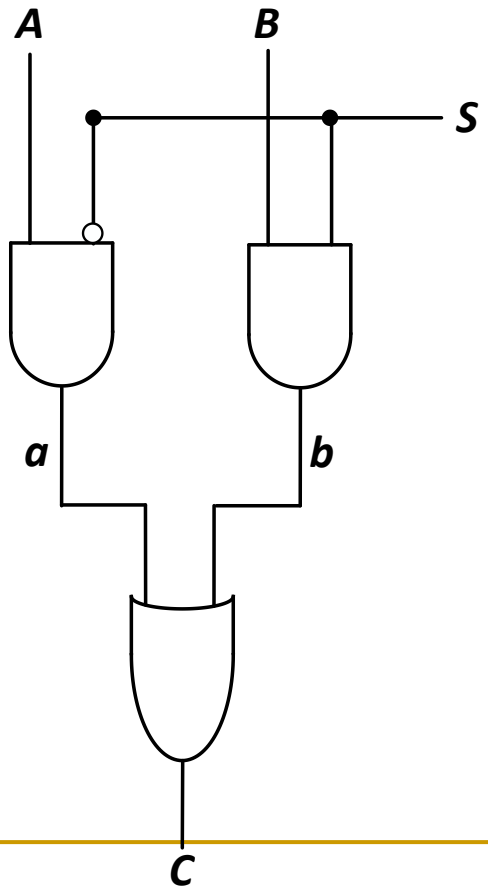
- Selects one of the N inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called select
- Example: 2-to-1 MUX

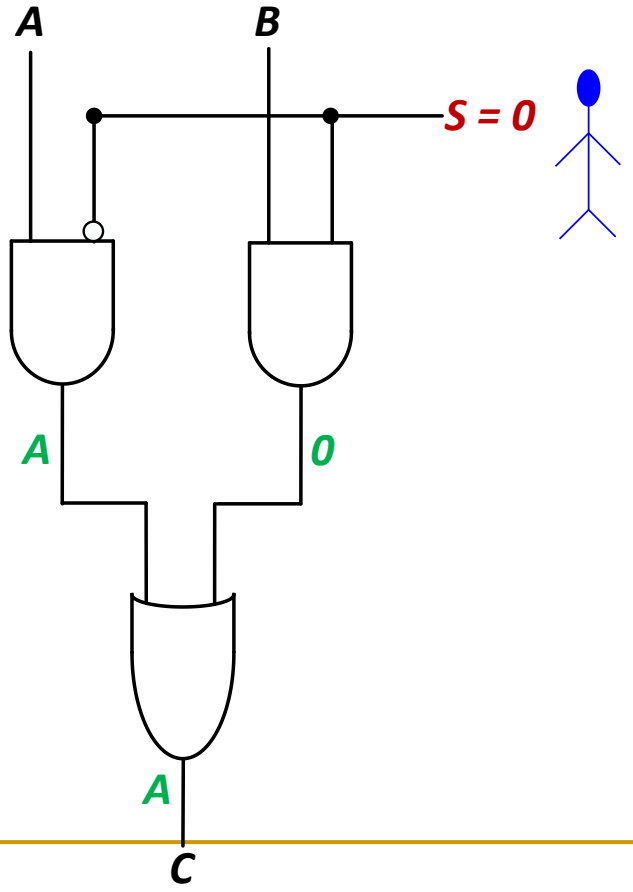
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Multiplexer (MUX), or Selector (II)

- Selects one of the N inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called select
- Example: 2-to-1 MUX

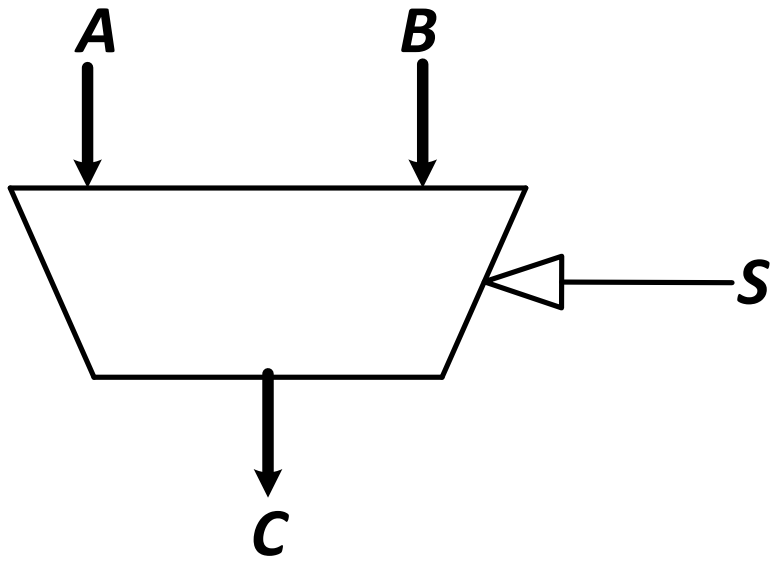




Multiplexer (MUX), or Selector (III)

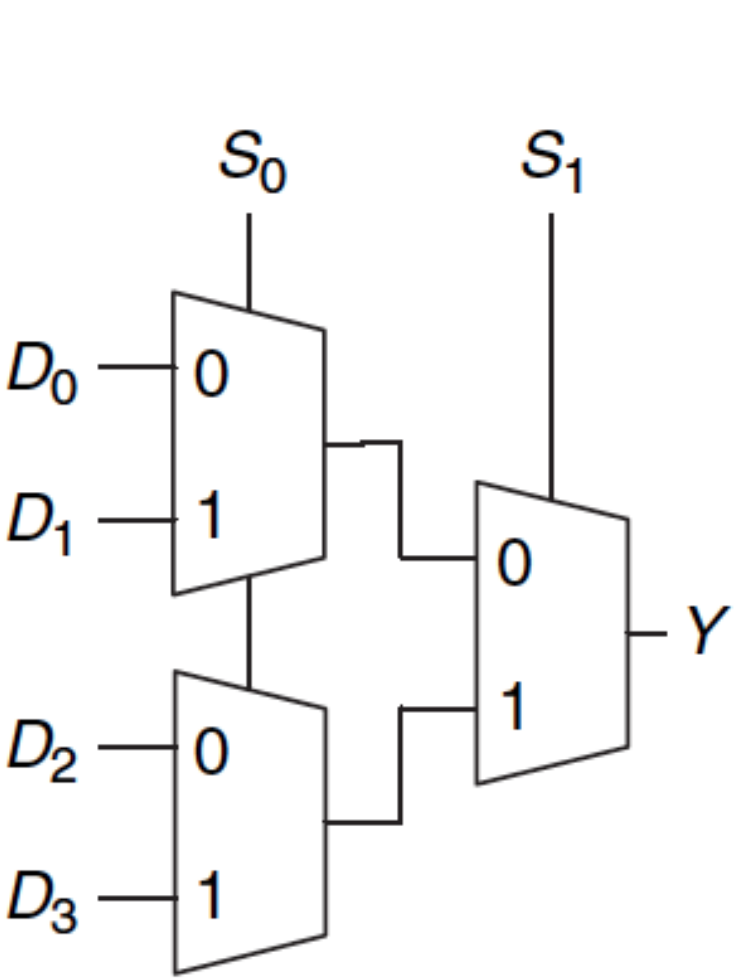
- The output C is always connected to either the input A or the input B
  - Output value depends on the value of the select line S

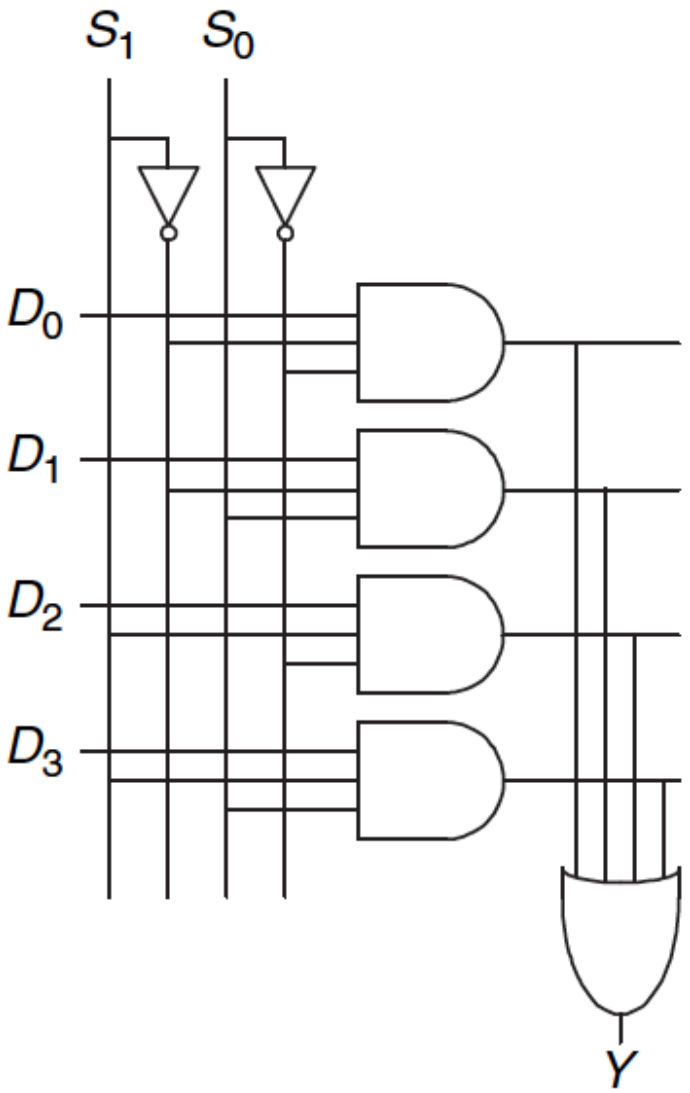
S	C
0	A
1	B



- Your task: Draw the schematic for an 4-input (4:1) MUX
  - Gate level: as a combination of basic AND, OR, NOT gates
  - Module level: As a combination of 2-input (2:1) MUXes

A 4-to-1 Multiplexer



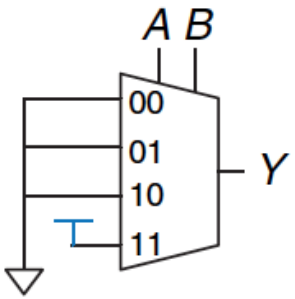


Aside: Logic Using Multiplexers

- Multiplexers can be used as lookup tables to perform logic functions

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

$Y = AB$



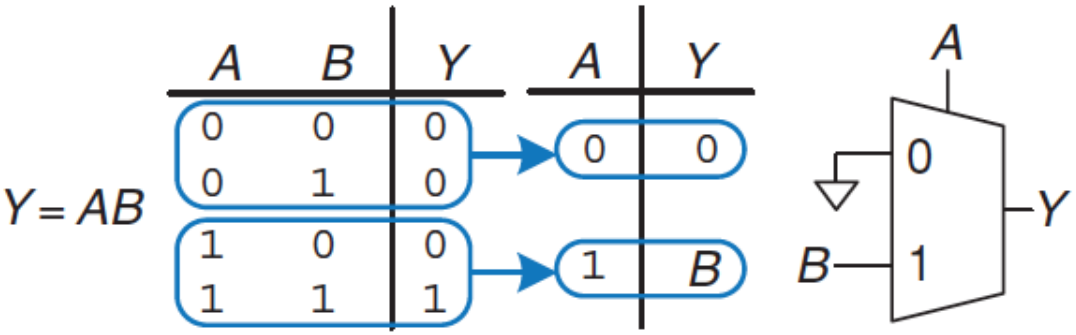


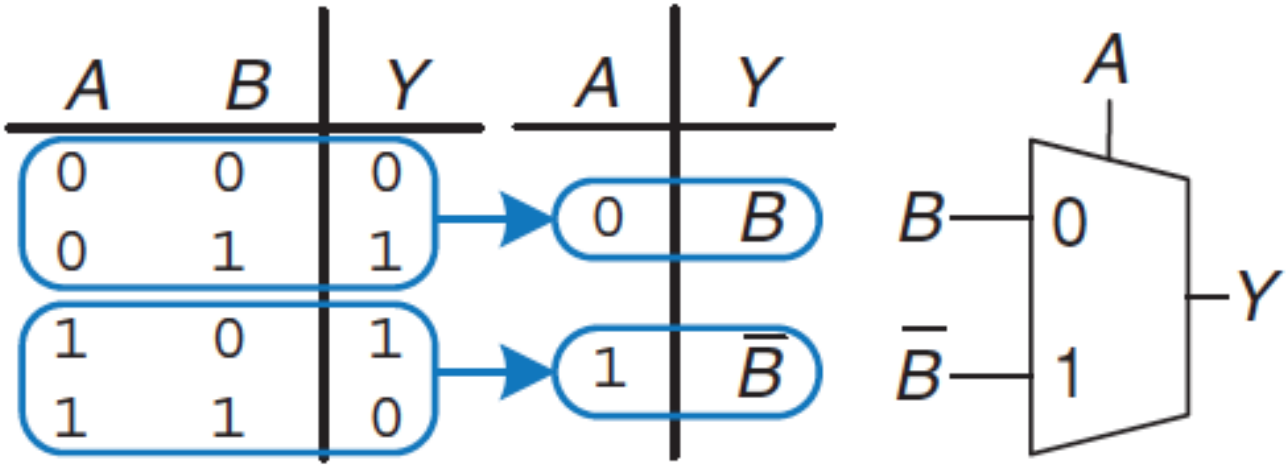
Figure 2.59 **4:1 multiplexer implementation of two-input AND function**

Idea: Formulate the truth table as a multiplexer

Aside: Logic Using Multiplexers (II)

- Multiplexers can be used as lookup tables to perform logic functions

$Y = A \oplus B$

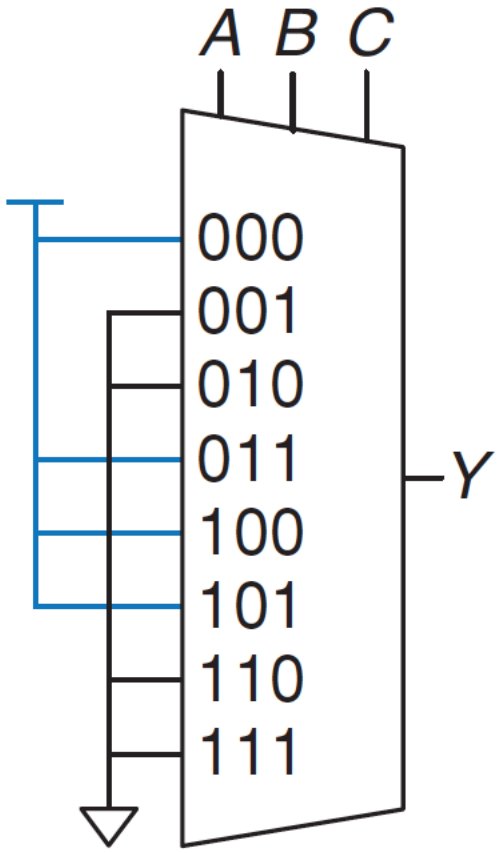


Aside: Logic Using Multiplexers (III)

- Multiplexers can be used as lookup tables to perform logic functions

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

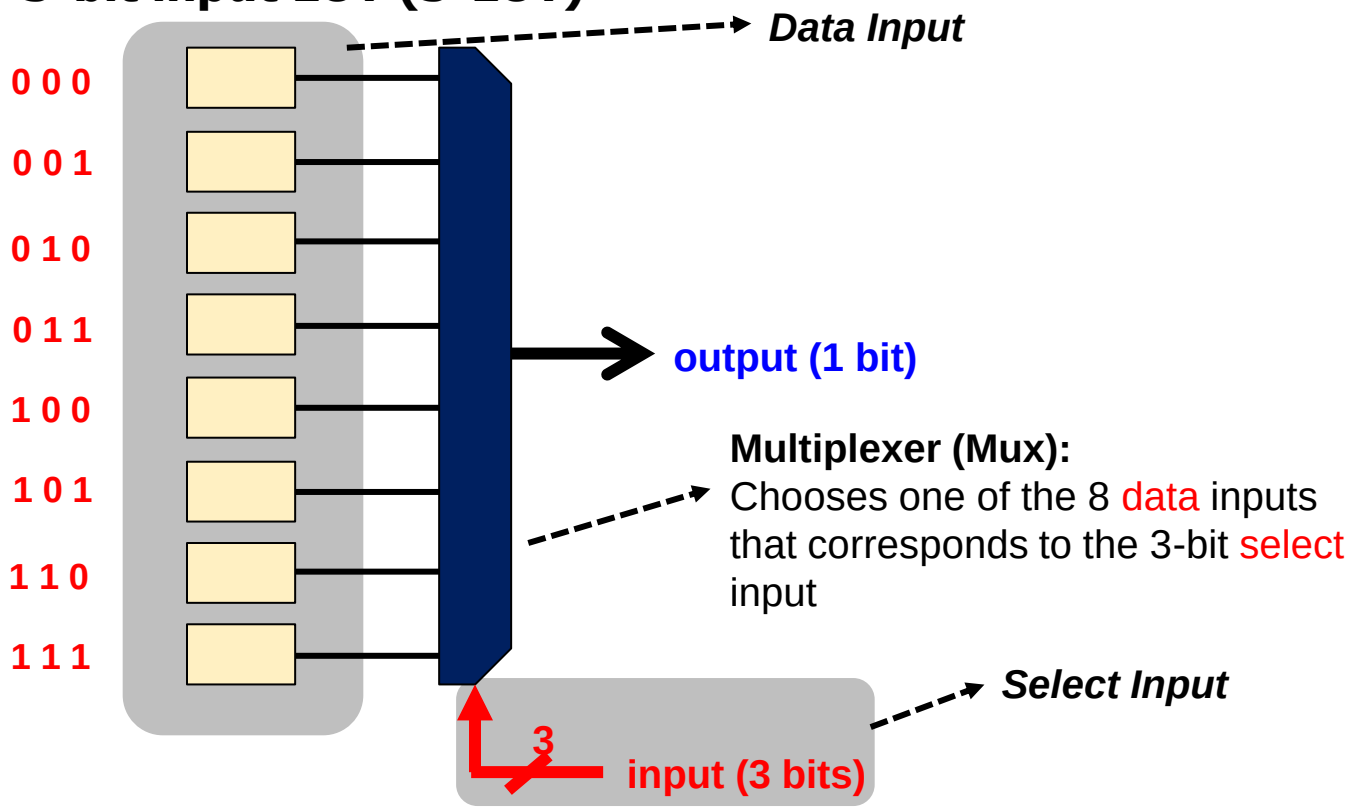
$$Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$$



Read H&H Chapter 2.8

Recall: 8-Input Lookup Table (LUT)

- **3-bit input LUT (3-LUT)**



3-LUT can implement
any 3-bit input function

Recall: An Example of Programming a LUT

- Let's implement a function that outputs '1' when there are at least two '1's in a 3-bit input

In C:

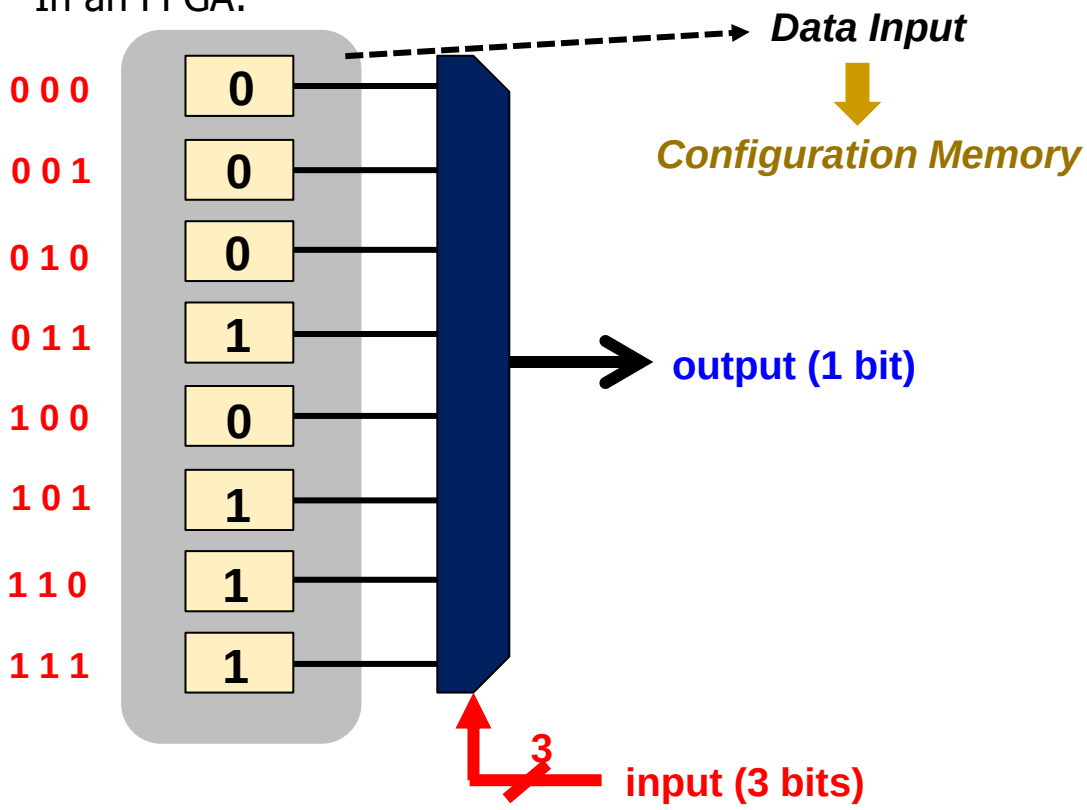
```
int count = 0;
for(int i = 0; i < 3; i++) {
    count += input & 1;
    input = input >> 1;
}

if(count > 1) return 1;

return 0;
```

```
switch(input){
   case 0:
   case 1:
   case 2:
   case 4:
      return 0;
   default:
      return 1;}
```

In an FPGA:



Aside: Logic Using Decoders (I)

- Decoders can be combined with OR gates to build logic functions.

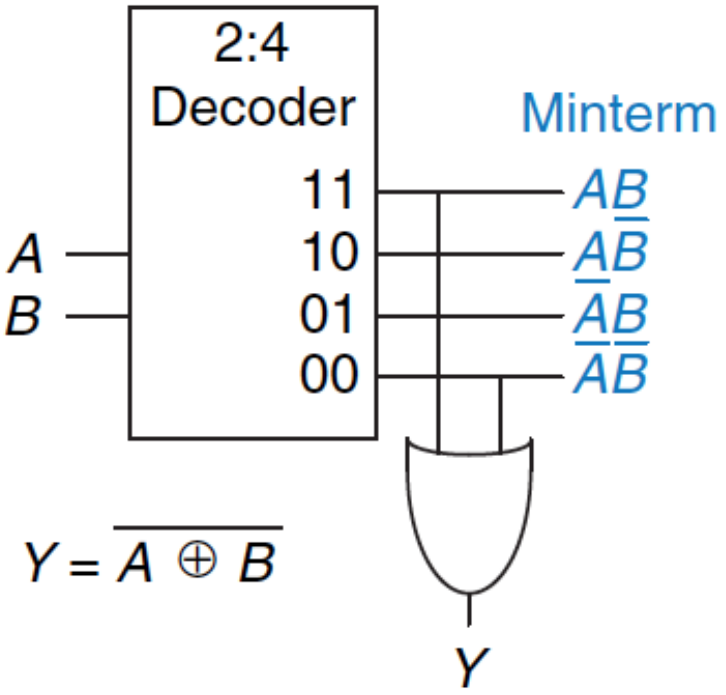


Figure 2.65 Logic function using decoder

Read H&H Chapter 2.8

Full Adder

Full Adder (I)

- **Binary addition**
  - Similar to decimal addition
  - From right to left
  - One column at a time
  - One sum and one carry bit

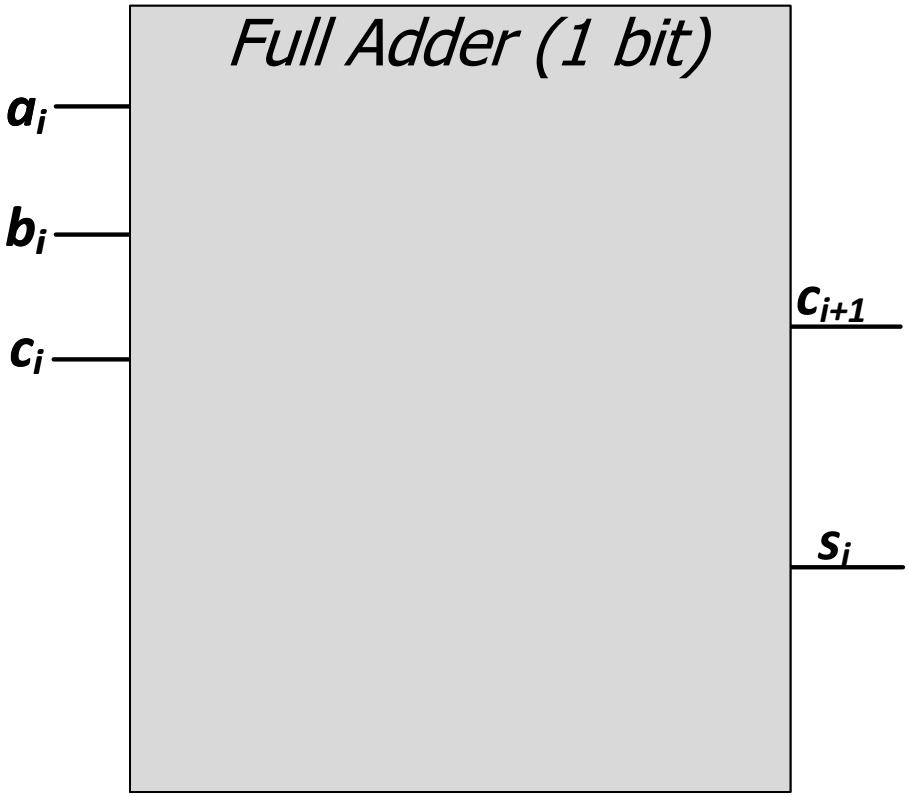
$$\begin{array}{c} a_{n-1}a_{n-2}\ldots a_1a_0 \\ b_{n-1}b_{n-2}\ldots b_1b_0 \\ C_n\ C_{n-1}\quad\ldots\quad C_1 \\ \hline S_{n-1}\quad\ldots\quad S_1S_0 \end{array} \downarrow$$

- Truth table of binary addition on one column of bits within two n-bit operands

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder (II)

- **Binary addition**
  - N 1-bit additions
  - **SOP of 1-bit addition**

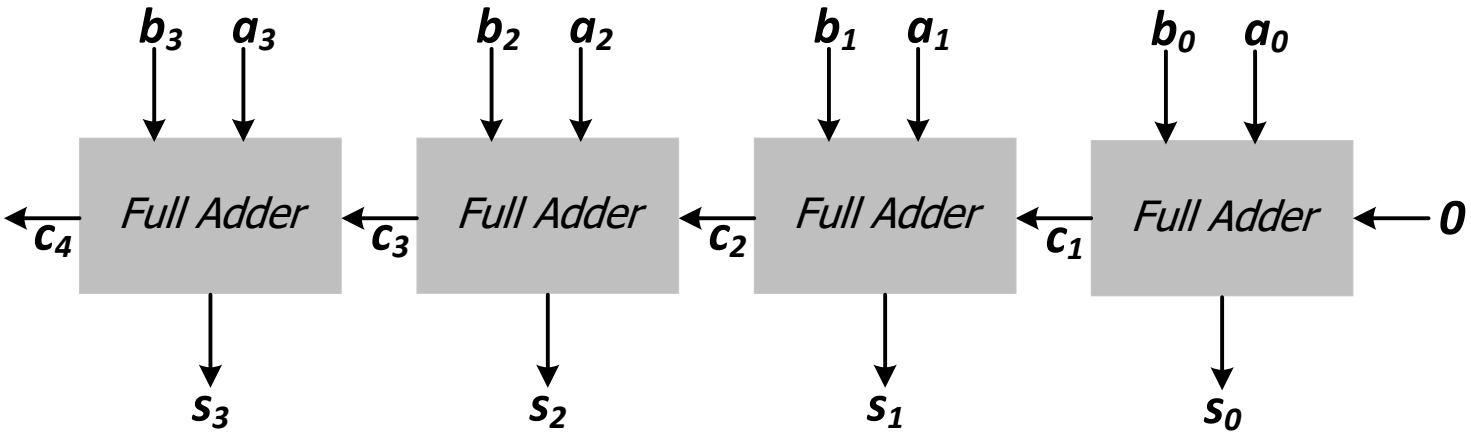


$$\begin{array}{r} a_{n-1}a_{n-2}\ldots a_1a_0 \\ b_{n-1}b_{n-2}\ldots b_1b_0 \\ \hline C_n\ C_{n-1}\quad\ldots\quad C_1 \\ \hline S_{n-1}\quad\ldots\quad S_1S_0 \end{array}$$

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

4-Bit Adder from Full Adders

- Creating a **4-bit adder** out of 1-bit full adders
  - To add two 4-bit binary numbers A and B



$$\begin{array}{cccccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline c_4 & c_3 & c_2 & c_1 & \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array} \qquad \begin{array}{cccccc} & 1 & 0 & 1 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & \\ \hline & 0 & 1 & 0 & 0 \end{array}$$

Adder Design: Ripple Carry Adder

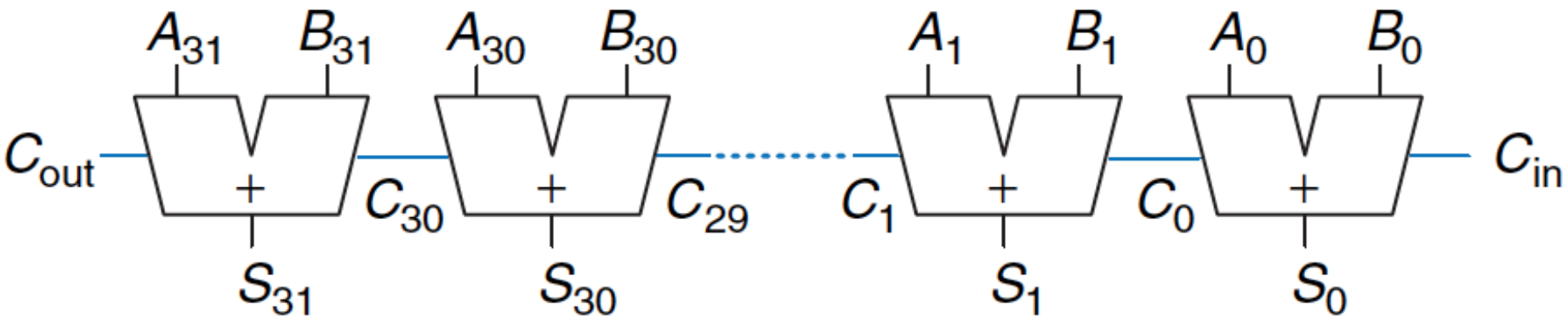
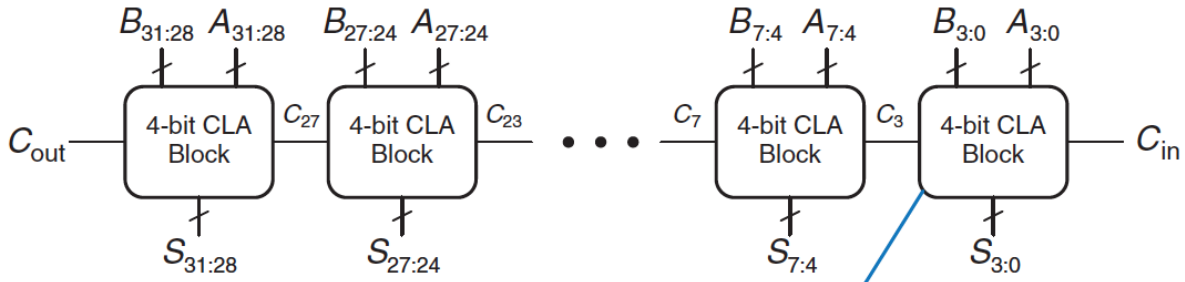
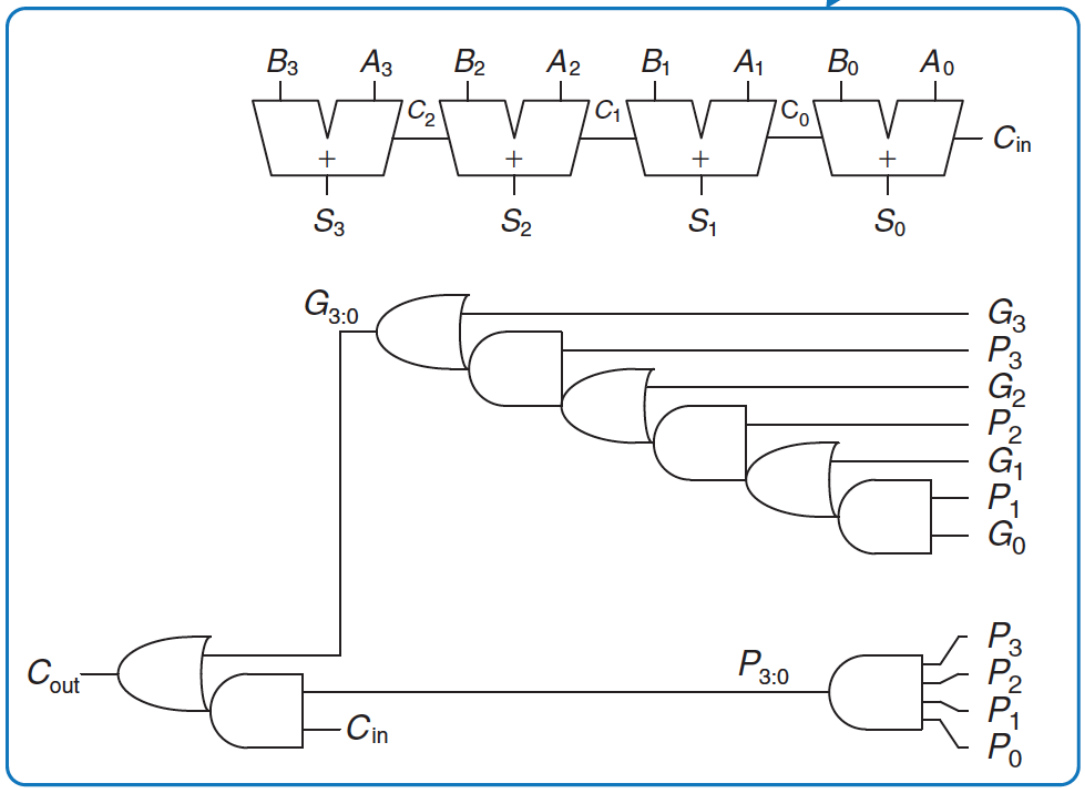


Figure 5.5 32-bit ripple-carry adder

Adder Design: Carry Lookahead Adder



(a)



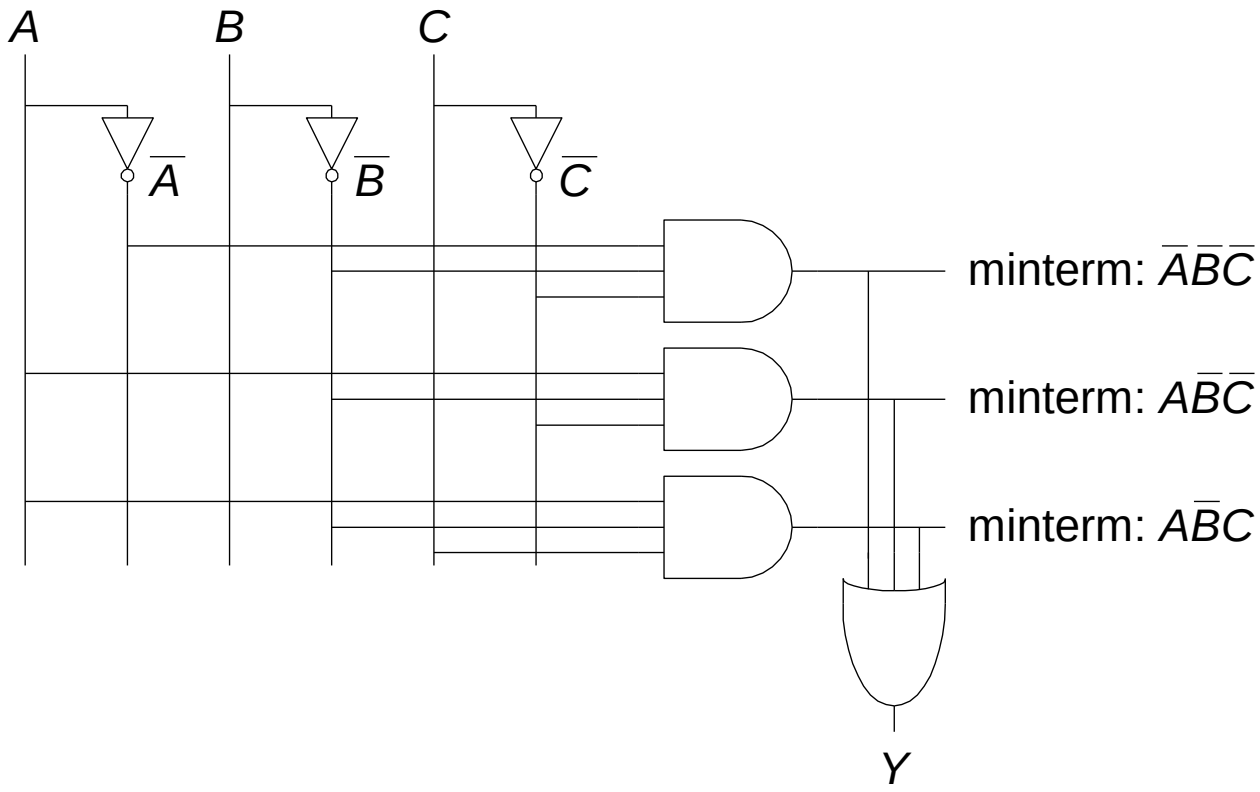
(b)

Example of logic specialization: Specialized logic for fast carry generation

Programmable Logic Array (PLA)

PLA: Recall: SOP Form

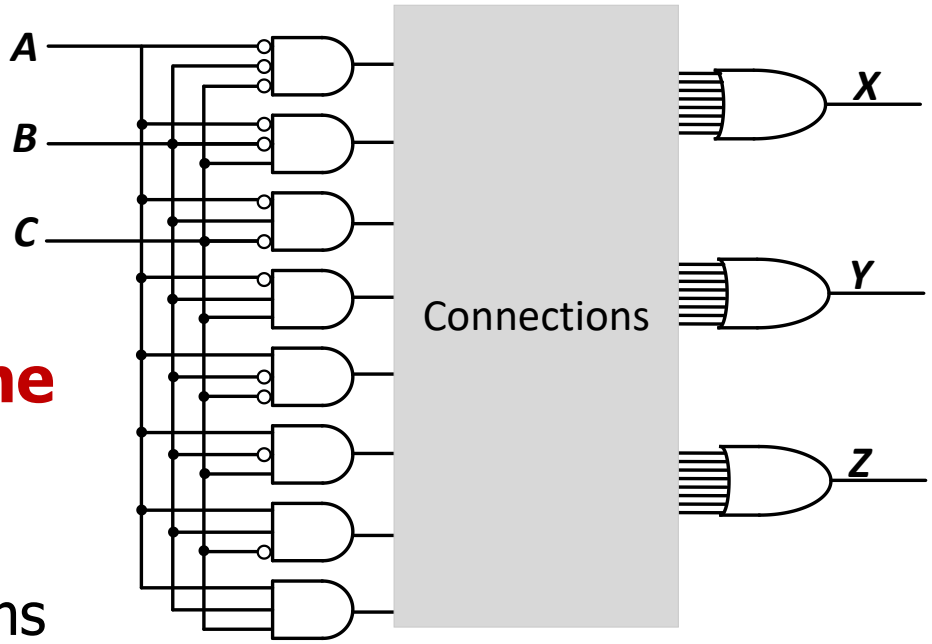
- **SOP (sum-of-products) leads to two-level logic**
- Example: $Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C)$



A PLA enables the two-level SOP implementation of **any** N-input M-output function

The Programmable Logic Array (PLA)

- The below logic structure is a very common building block for implementing any collection of logic functions one wishes to

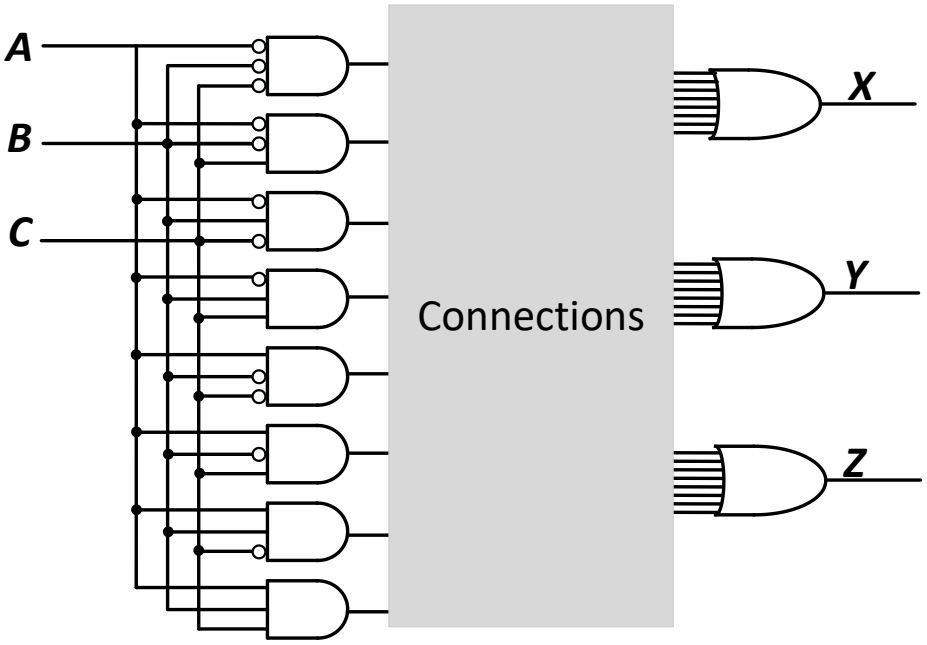


  - An array of AND gates followed by an array of OR gates
  - **How do we determine the number of AND gates?**
    - **Remember SOP:** the number of possible minterms
    - For an n-input logic function, we need a PLA with 2^n n-input AND gates
  - **How do we determine the number of OR gates?** The number of output columns in the truth table

A PLA enables the two-level SOP implementation of **any** N-input M-output function

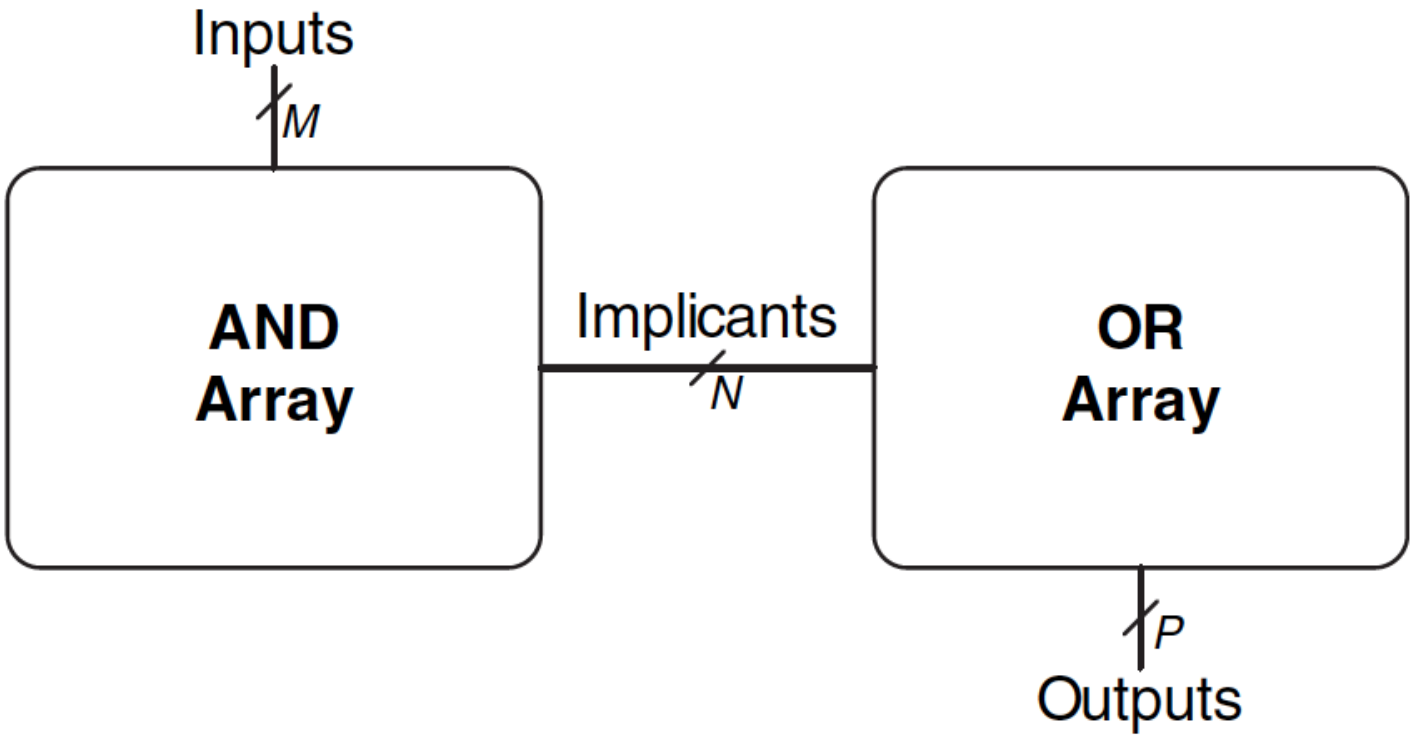
The Programmable Logic Array (PLA)

- How do we implement a logic function?
  - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP
  - This is a simple programmable logic construct



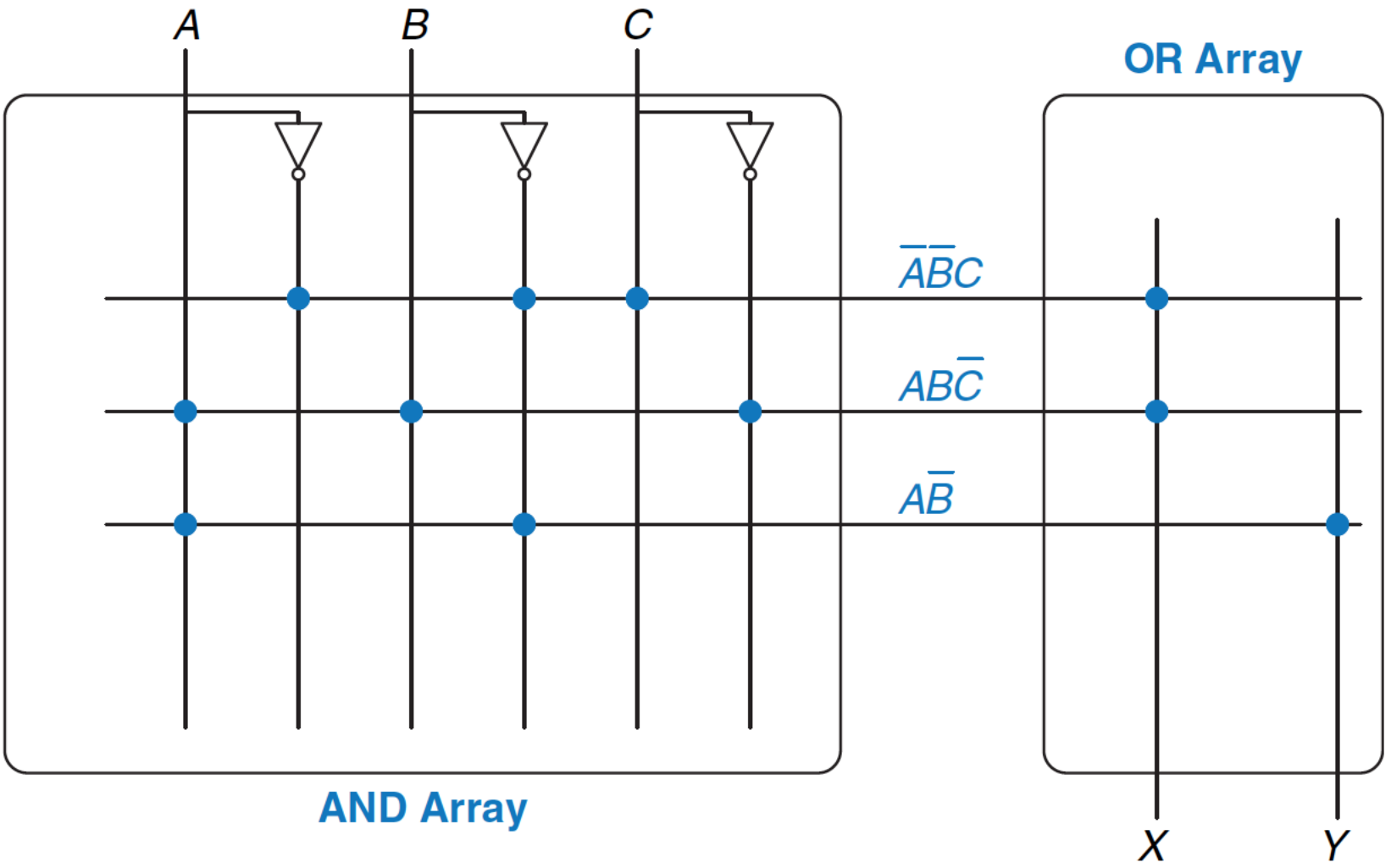
- **Programming a PLA**: we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function
- Have you seen any other type of programmable logic?
  - Yes! An FPGA…
  - An FPGA uses more advanced structures, as we saw in Lecture 3

PLA Example (I)



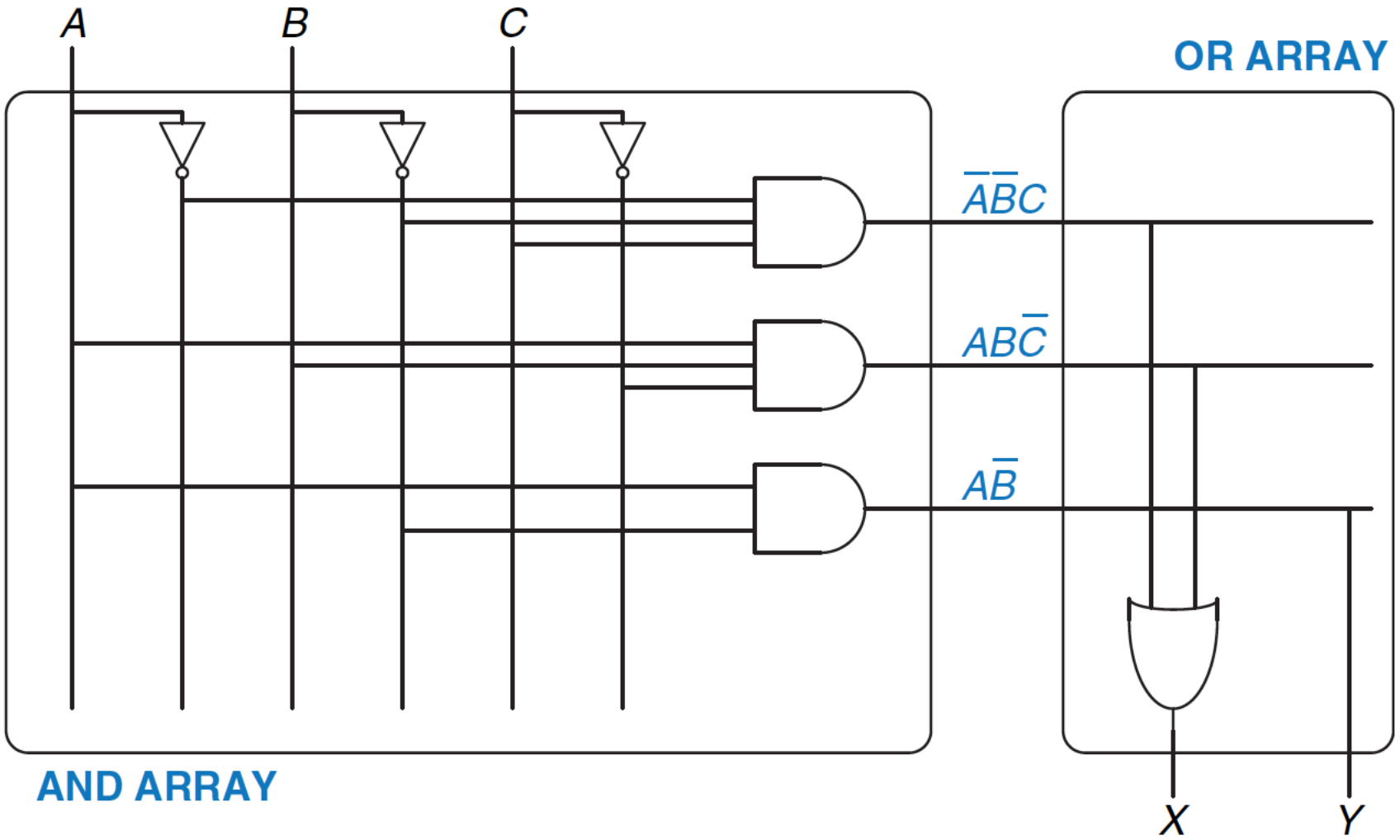
Read H&H Chapter 5.6.1

PLA Example Function (II)



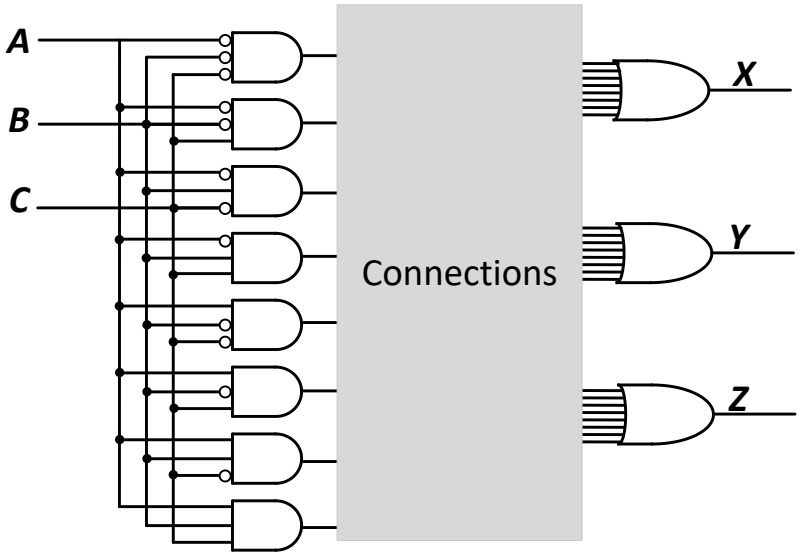
Read H&H Chapter 5.6.1

PLA Example Function (III)



Read H&H Chapter 5.6.1

Implementing a Full Adder Using a PLA

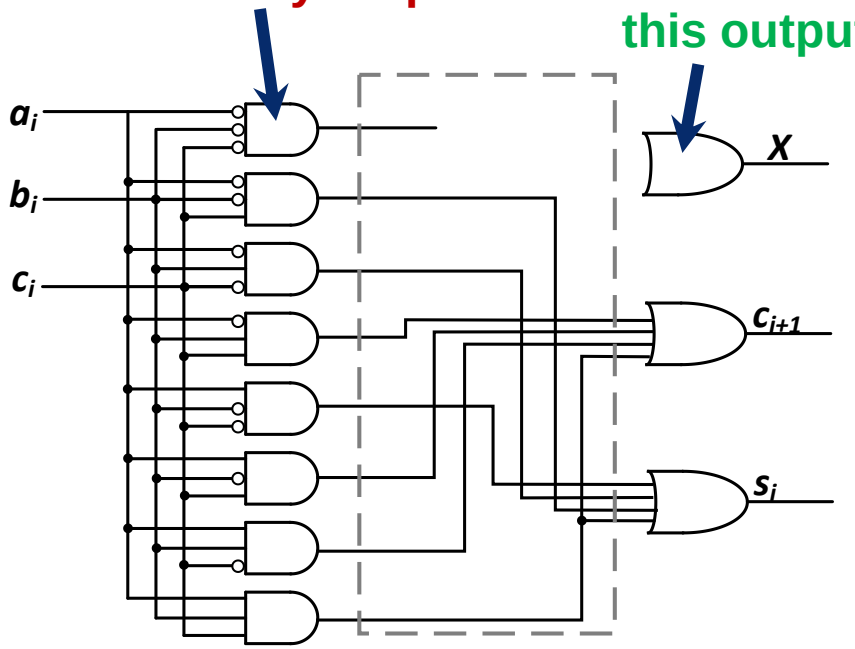


Truth table of a full adder

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This input should not be connected to any outputs

We do not need this output



Logical Completeness

Logical (Functional) Completeness

- Any logic function we wish to implement could be accomplished with a PLA
  - PLA consists of only AND gates, OR gates, and inverters
  - We just have to program connections based on SOP of the intended logic function

- The set of gates {AND, OR, NOT} is logically complete because we can build a circuit to carry out the specification of any truth table we wish, without using any other kind of gate

- NAND is also logically complete. So is NOR.
  - Your task: Prove this.

More Combinational Blocks

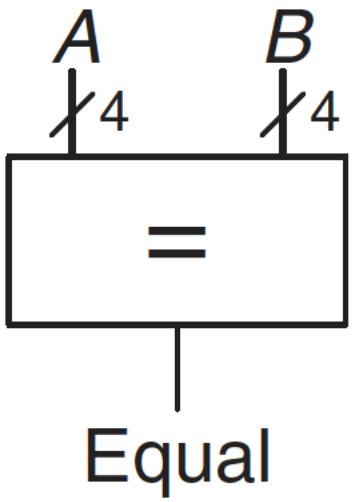
More Combinational Building Blocks

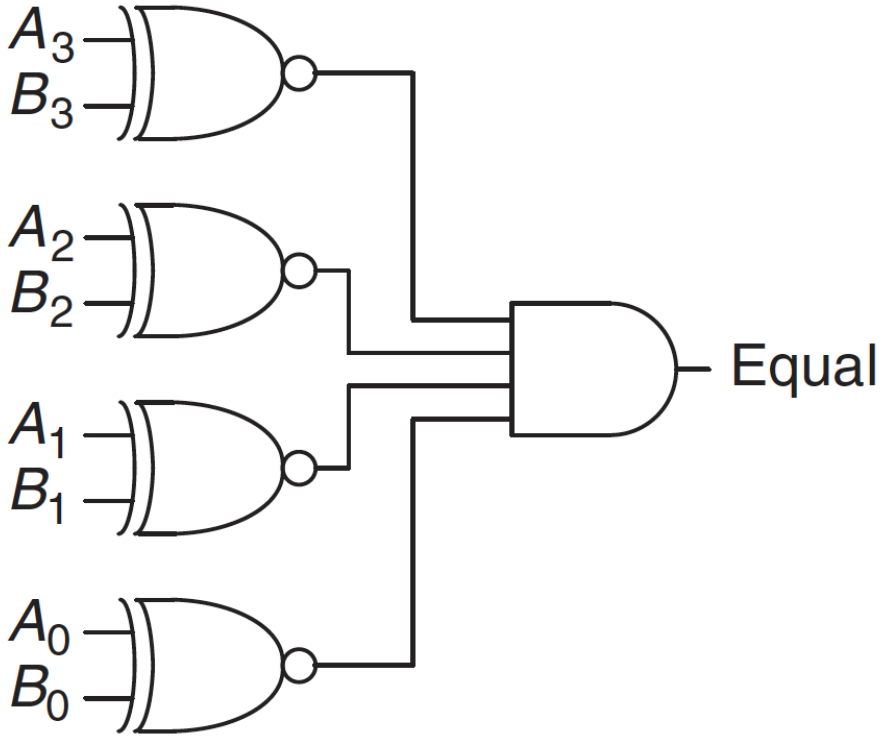
- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6
- H&H Chapter 5
  - Will be required reading soon.
- You will benefit greatly by reading the "combinational" parts of Chapter 5 soon.
  - Sections 5.1 and 5.2

Comparator

Equality Checker (Compare if Equal)

- Checks if two N-input values are exactly the same
- Example: 4-bit Comparator





ALU (Arithmetic Logic Unit)

ALU (Arithmetic Logic Unit)

- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)
- Usually denoted with this symbol:

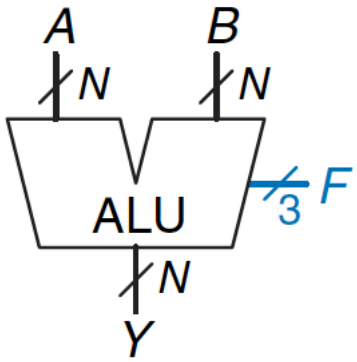


Figure 5.14 ALU symbol

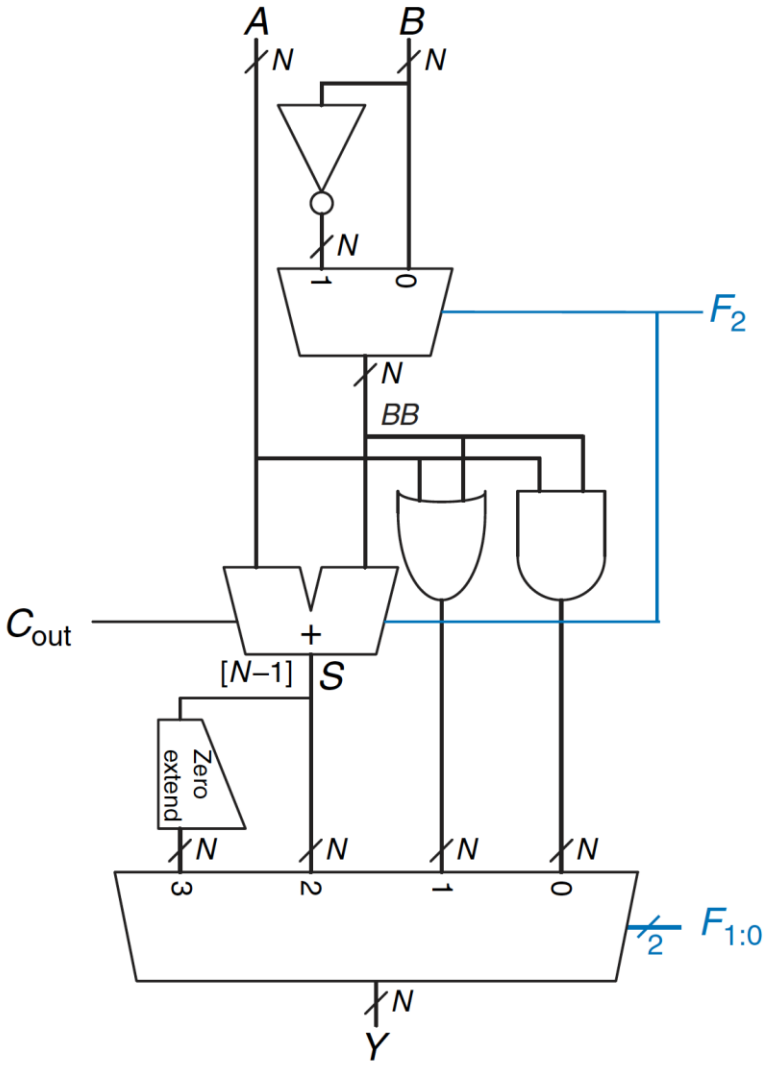
Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

Example ALU (Arithmetic Logic Unit)

Table 5.1 ALU operations

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



More Combinational Building Blocks

- See H&H Chapter 5.2 for
  - Subtractor (using 2's Complement Representation)
  - Shifter and Rotator
  - Multiplier
  - Divider
  - …

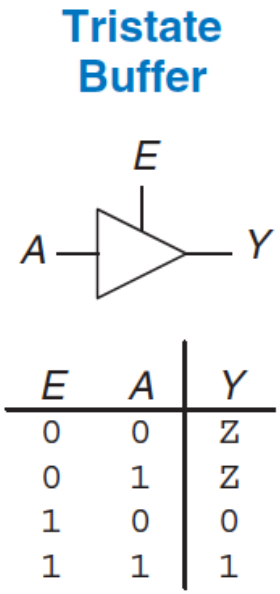
More Combinational Building Blocks

- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6
- H&H Chapter 5
  - Will be required reading soon.
- You will benefit greatly by reading the "combinational" parts of Chapter 5 soon.
  - Sections 5.1 and 5.2

Tri-State Buffer

Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer

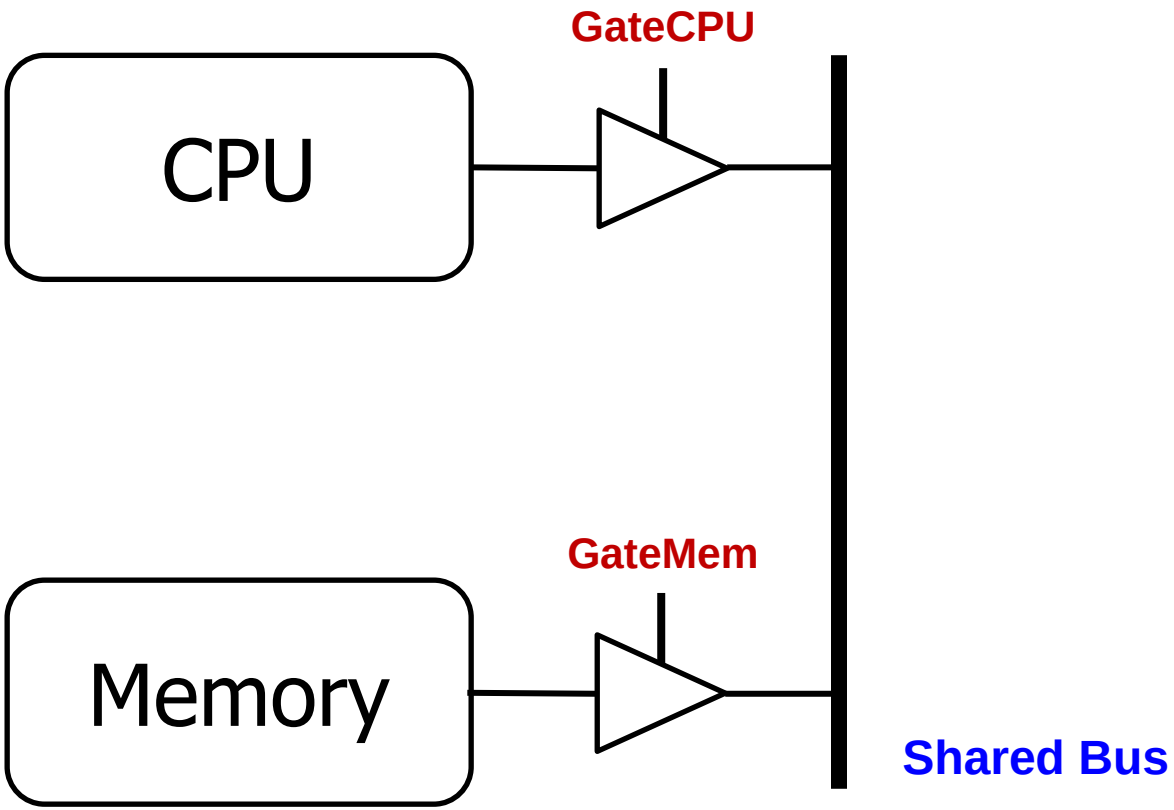
A tri-state buffer acts like a switch

- Floating signal (Z): Signal that is not driven by any circuit
  - Open circuit, floating wire

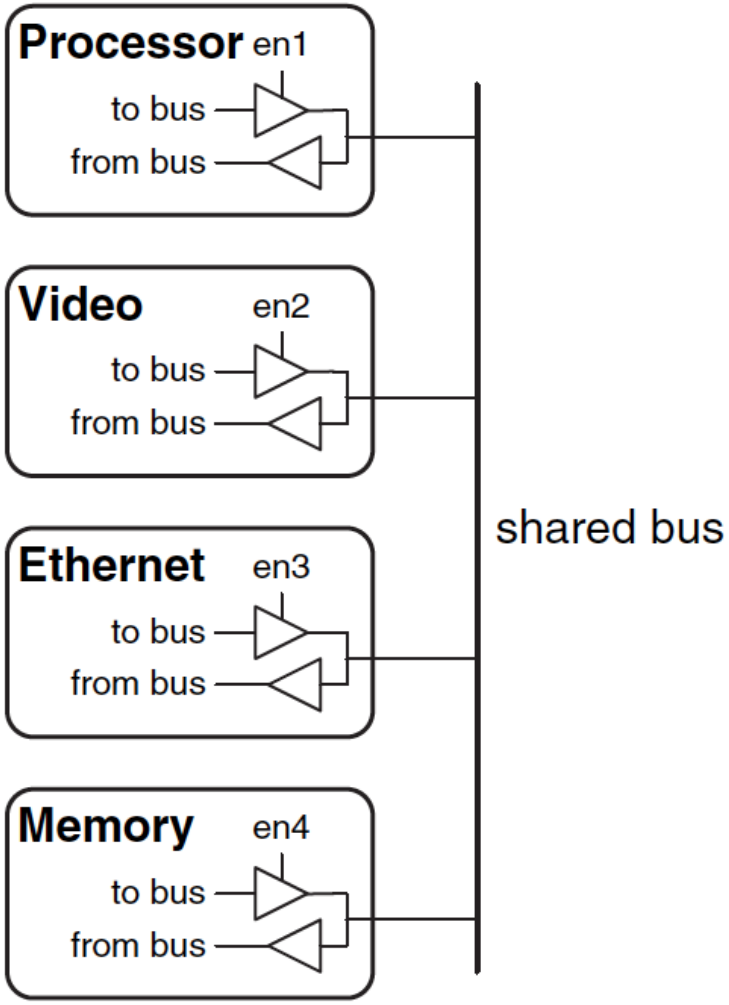
Example: Use of Tri-State Buffers

- Imagine a wire connecting the CPU and memory
  - At any time only the CPU or the memory can place a value on the wire, both not both
  - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

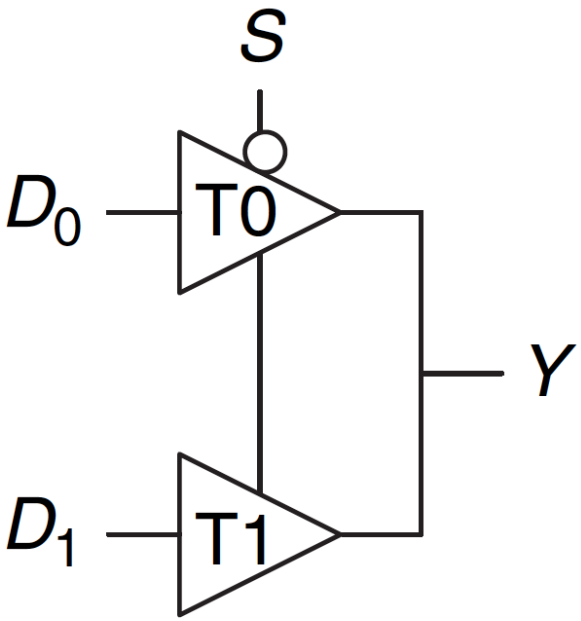
Example Design with Tri-State Buffers



Another Example



Multiplexer Using Tri-State Buffers



$$Y = D_0\bar{S} + D_1 S$$

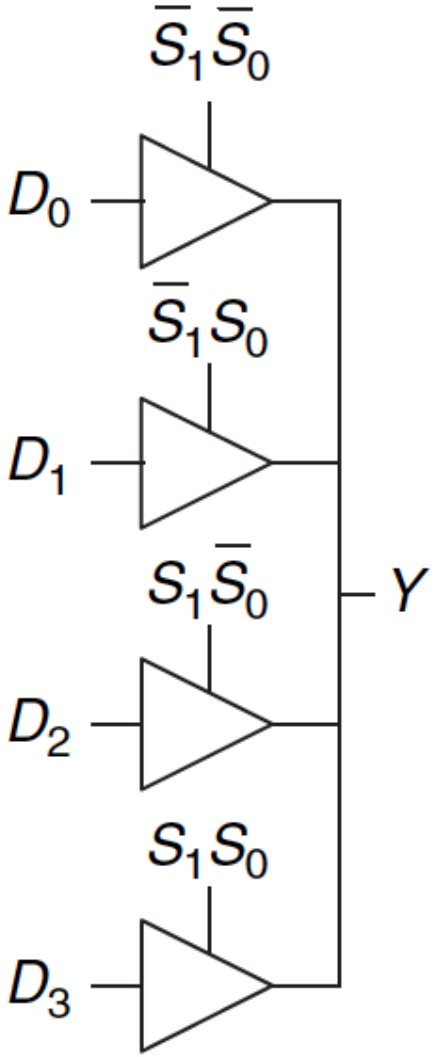
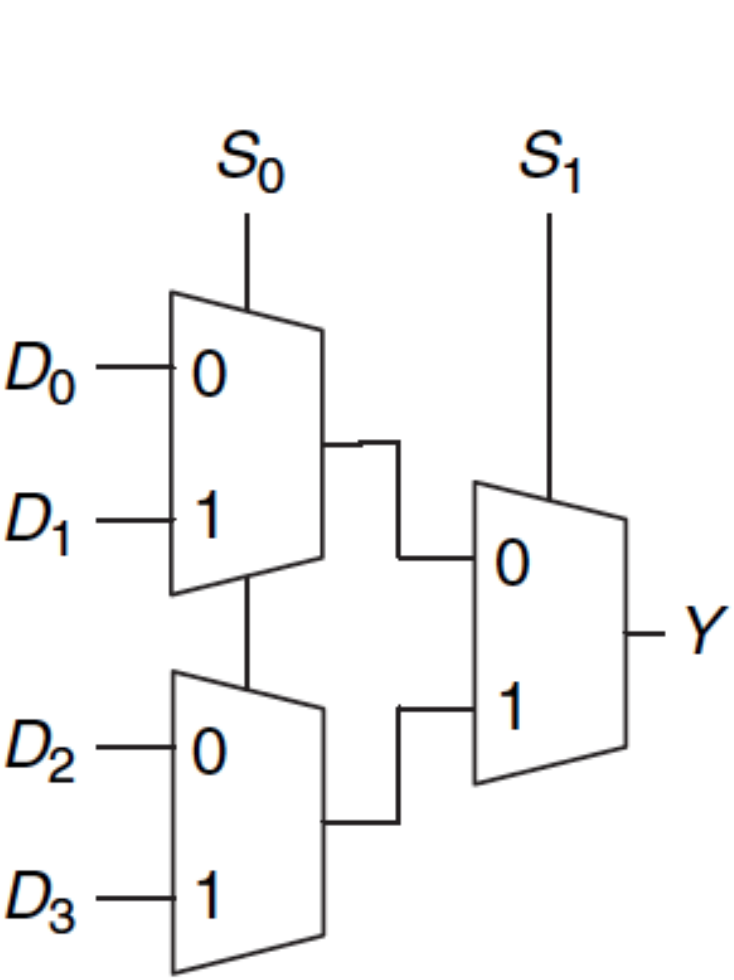
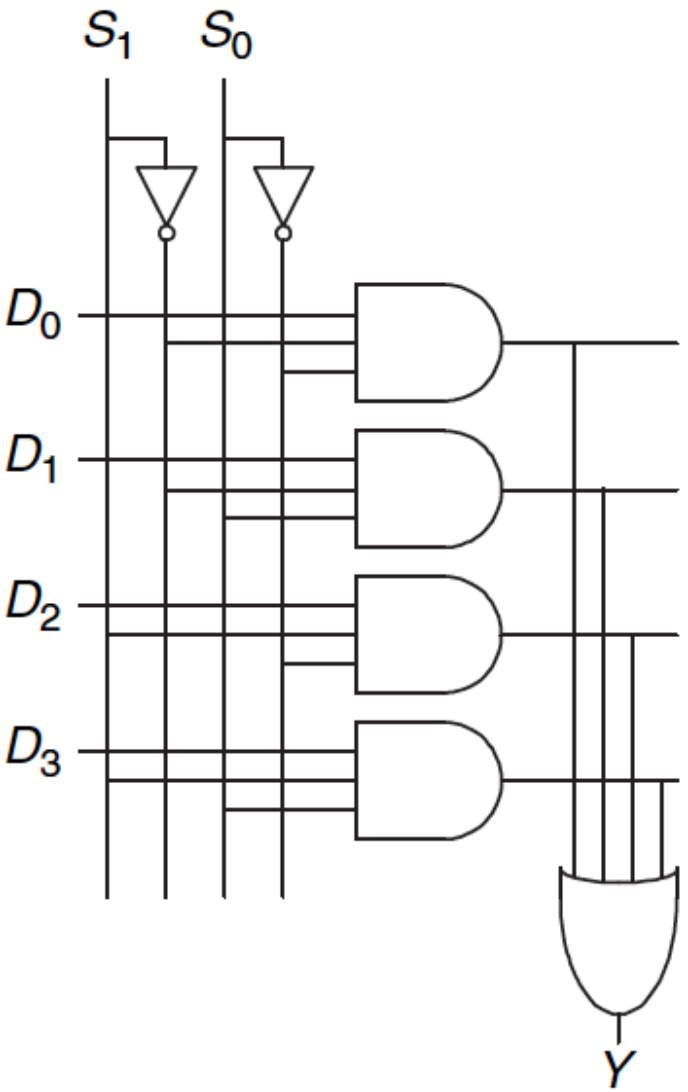


Figure 2.56 Multiplexer using tristate buffers

Recall: A 4-to-1 Multiplexer

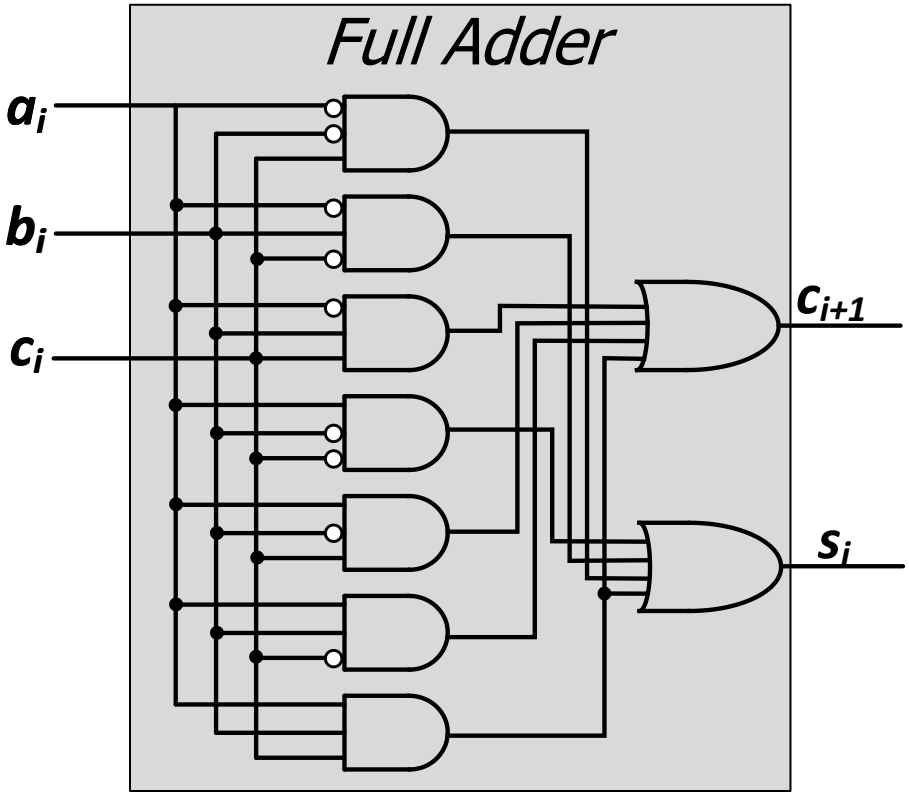




We Covered Until This Point in the Lecture

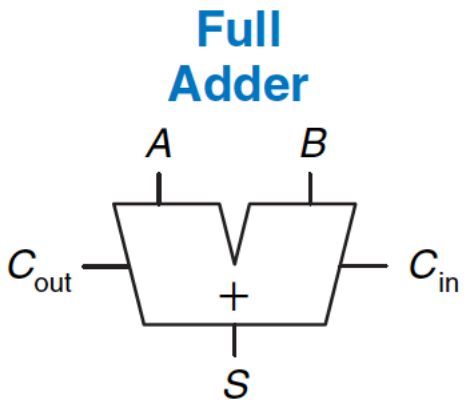
Logic Simplification using Boolean Algebra Rules

Recall: Full Adder in SOP Form Logic



a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Goal: Simplified Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

How do we simplify Boolean logic?

How do we automate simplification?

Quick Recap on Logic Simplification

- The original Boolean expression (i.e., logic circuit) may not be optimal

$$F = \sim A(A + B) + (B + AA)(A + \sim B)$$

- Can we reduce a given Boolean expression to an equivalent expression with fewer terms?

$$F = A + B$$

- The goal of logic simplification:
  - Reduce the number of gates/inputs
  - Reduce implementation cost

A basis for what the automated design tools are doing today

Logic Simplification

- Systematic techniques for simplifications
  - amenable to automation

Key Tool: The Uniting Theorem — $F = A\overline{B} + AB$

A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

$$F = A\overline{B} + AB = A(\overline{B} + B) = A(1) = A$$

Essence of Simplification:

Find two element subsets of the ON-set where only one variable changes its value. This single varying variable *can be eliminated!*

value is not needed

→ *B is eliminated, A remains*

A	B	G
0	0	1
0	1	0
1	0	1
1	1	0

$$G = \overline{A}\overline{B} + A\overline{B} = (\overline{A} + A)\overline{B} = \overline{B}$$

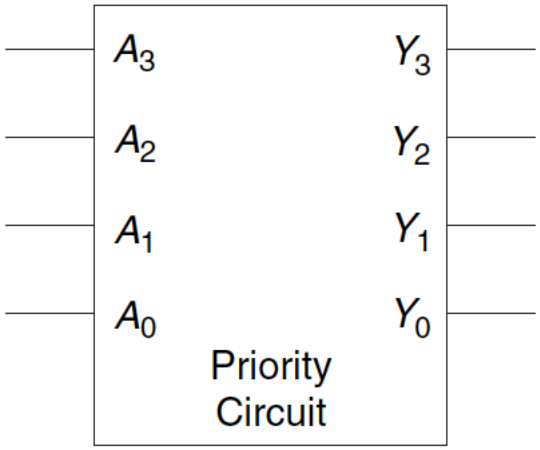
B's value stays the same within the ON-set rows

A's value changes within the ON-set rows

→ *A is eliminated, B remains*

Logic Simplification Example: Priority Circuit

- Priority Circuit
  - Inputs: "Requestors" with priority levels
  - Outputs: "Grant" signal for each requestor
  - Example 4-bit priority circuit



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

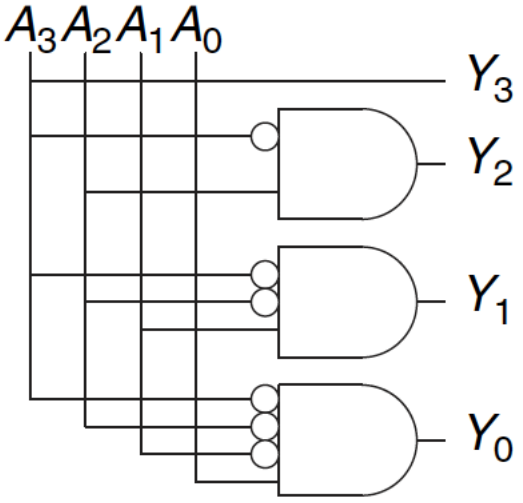
Simplified Priority Circuit

- Priority Circuit
  - Inputs: "Requestors" with priority levels
  - Outputs: "Grant" signal for each requestor
  - Example 4-bit priority circuit

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 Priority circuit truth table with don't cares (X's)



X (Don't Care) means *I don't care what the value of this input is*

Logic Simplification: Karnaugh Maps (K-Maps)

Karnaugh Maps are Fun…

- A pictorial way of minimizing circuits by visualizing opportunities for simplification
- They are for you to **study on your own**…

- See remaining slides
- Read H&H Section 2.7
- Watch videos of Lectures 5 and 6 from 2019 DDCA course:
  - https://youtu.be/0ks0PeaOUjE?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=4570
  - https://youtu.be/ozs18ARNG6s?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=220

Backup Slides on Karnaugh Maps (K-Maps)

Complex Cases

- One example

$$Cout = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

- Problem
  - Easy to see how to apply Uniting Theorem…
  - Hard to know if you applied it in all the right places…
  - …especially in a function of many more variables

- Question
  - Is there an easier way to find potential simplifications?
  - i.e., potential applications of Uniting Theorem…?

- Answer
  - Need an intrinsically *geometric* representation for Boolean f( )
  - Something we can draw, see…

Karnaugh Map

- Karnaugh Map (K-map) method
  - K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions
  - Physical adjacency ↔ Logical adjacency

2-variable K-map

A \ B	0	1
0	00	01
1	10	11

3-variable K-map

A \ BC	00	01	11	10
0	000	001	011	010
1	100	101	111	110

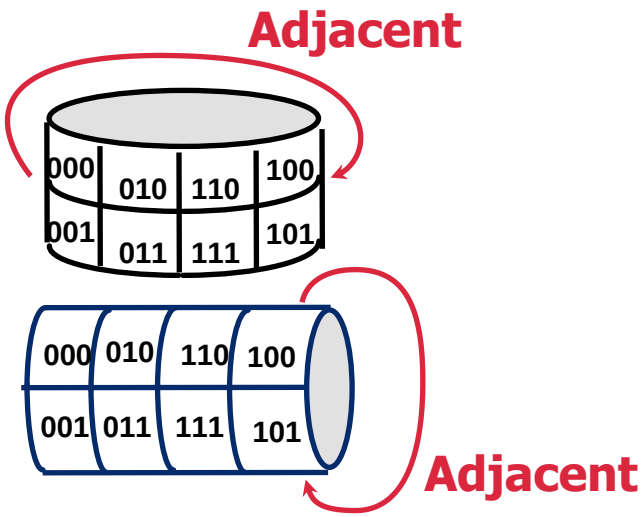
4-variable K-map

AB \ CD	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

Numbering Scheme: **00, 01, 11, 10 is called a "Gray Code" — only a *single bit (variable) changes* from one code word and the next code word**

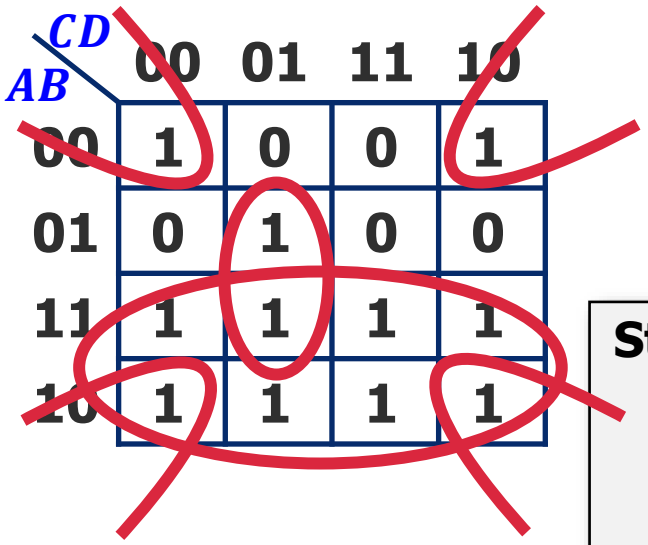
Karnaugh Map Methods

A \ BC	00	01	11	10
0	000	001	011	010
1	100	101	111	110



K-map adjacencies go "around the edges"
Wrap around from first to last column
Wrap around from top row to bottom row

K-map Cover - 4 Input Variables



$$F(A, B, C, D) = \sum m(0, 2, 5, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$F = A + \bar{B}\bar{D} + B\bar{C}D$$

Strategy for "circling" rectangles on Kmap:

Biggest "oops!" that people forget:

Logic Minimization Using K-Maps

- Very simple guideline:
  - Circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles
    - Each circle should be as large as possible
  - Read off the implicants that were circled

- More formally:
  - A Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants
  - Each circle on the K-map represents an implicant
  - The largest possible circles are prime implicants

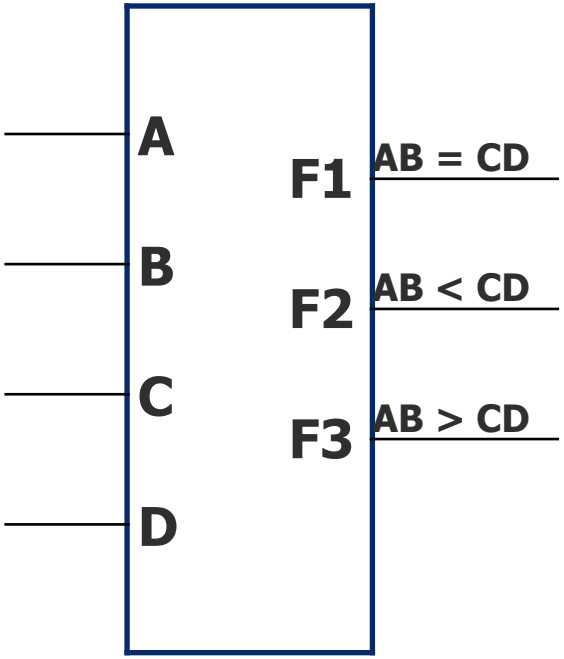
K-map Rules

- **What can be legally combined (circled) in the K-map?**
  - Rectangular groups of size 2^k for any integer k
  - Each cell has the same value (1, for now)
  - All values must be adjacent
    - Wrap-around edge is okay

- **How does a group become a term in an expression?**
  - Determine which literals are constant, and which vary across group
  - Eliminate varying literals, then AND the constant literals
    - constant 1 → use X, constant 0 → use $\overline{X}$

- **What is a good solution?**
  - Biggest groupings → eliminate more variables (literals) in each term
  - Fewest groupings → fewer terms (gates) all together
  - OR together all AND terms you create from individual groups

K-map Example: Two-bit Comparator

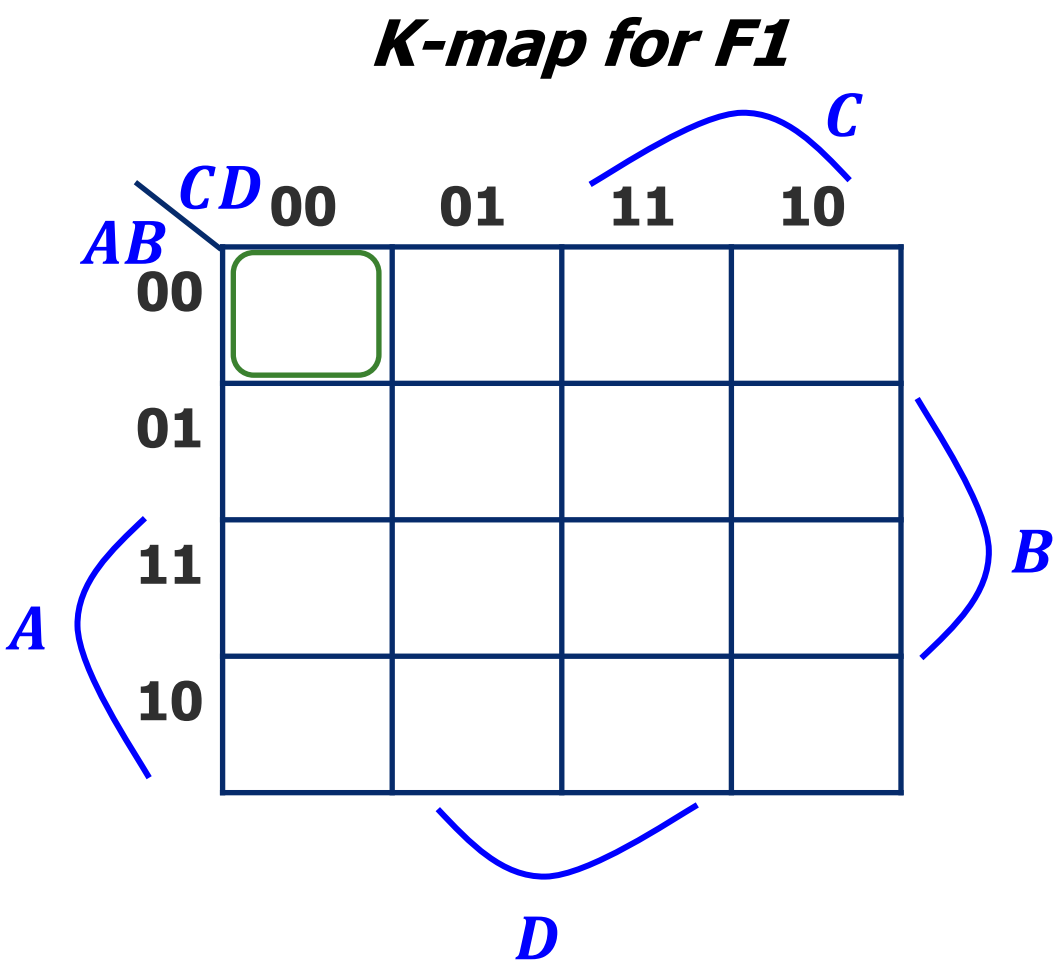


A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

Design Approach:

Write a 4-Variable K-map for each of the 3 output functions

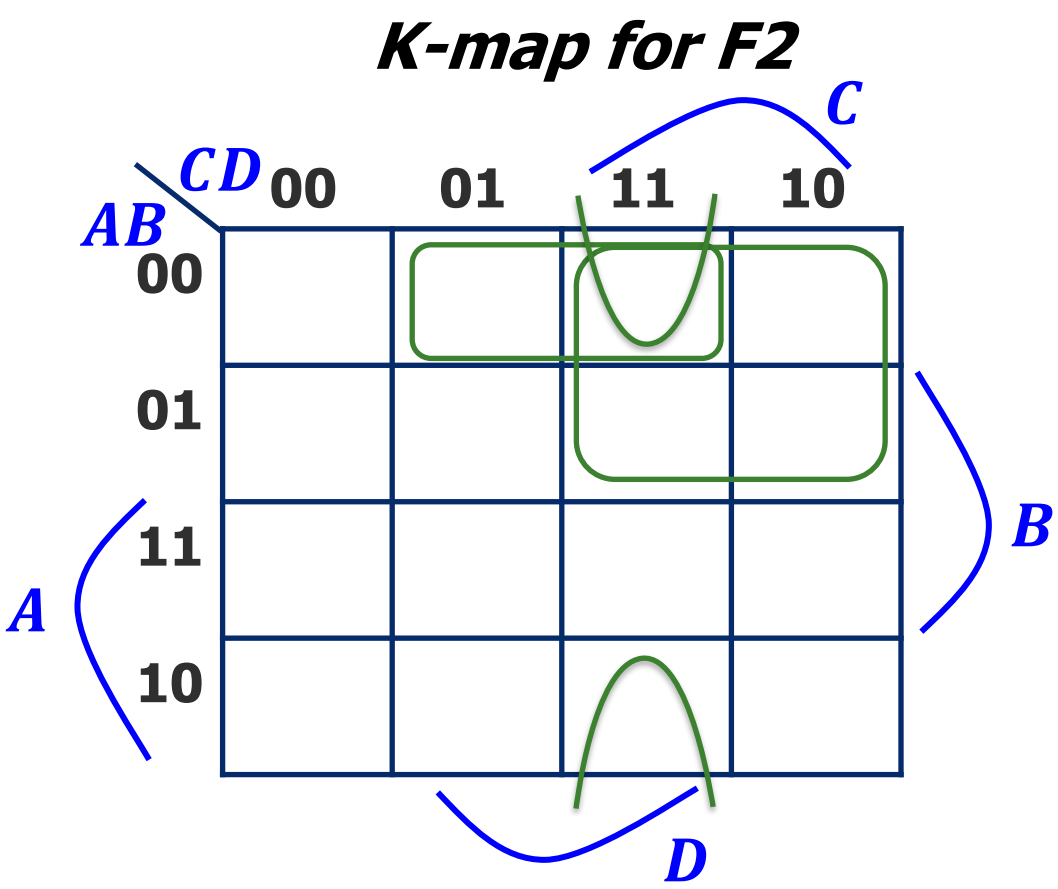
K-map Example: Two-bit Comparator (2)



A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	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	1	0	0

F1 =

K-map Example: Two-bit Comparator (3)



A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	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	1	0	0

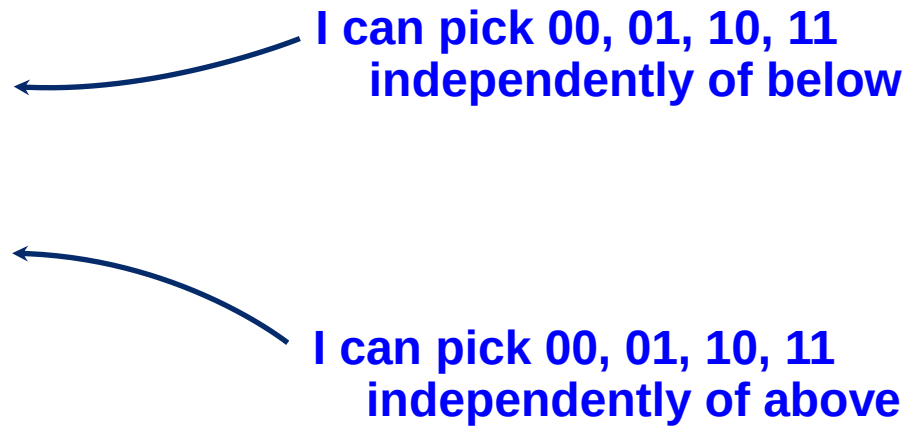
F2 =

F3 = ? (Exercise for you)

K-maps with “Don’t Care”

- Don’t Care really means *I don’t care what my circuit outputs if this appears as input*
  - You have an engineering choice to use DON’T CARE patterns intelligently as 1 or 0 to better simplify the circuit

A	B	C	D	F	G
…					
0	1	1	0	X	X
0	1	1	1		
1	0	0	0	X	X
1	0	0	1		
…					



Example: BCD Increment Function

- BCD (Binary Coded Decimal) digits
  - Encode decimal digits 0 - 9 with bit patterns 0000_2 — 1001_2
  - When incremented, the decimal sequence is 0, 1, …, 8, 9, 0, 1

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

These input patterns should never be encountered in practice (hey -- it's a BCD number!) So, associated output values are "Don't Cares"

K-map for BCD Increment Function

A B

\+

W X

Z (without don't cares) =

Z (with don't cares) =

10	1		X	X	10			X	X

Y

AB \ CD	00	01	11	10
00		1		1
01		1		1
11	X	X	X	X
10			X	X

Z

AB \ CD	00	01	11	10
00	1			1
01	1			1
11	X	X	X	X
10	1		X	X

C B A D

K-map Summary

- Karnaugh maps as a formal systematic approach for logic simplification
- 2-, 3-, 4-variable K-maps
- K-maps with "Don't Care" outputs
- H&H Section 2.7

Digital Design & Computer Arch.

Lecture 5: Combinational Logic II

Prof. Onur Mutlu

ETH Zürich

Spring 2022

10 March 2022