Digital Design & Computer Arch.

Lecture 5: Combinational Logic II

Prof. Onur Mutlu

ETH Zürich
Spring 2020
5 March 2020

Assignment: Required Lecture Video

- Why study computer architecture?
- Why is it important?
- **Future Computing Architectures**

- **Required Assignment**
  - **Watch** Prof. Mutlu's inaugural lecture at ETH and understand it
  - https://www.youtube.com/watch?v=kgiZlSOcGFM

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

Assignment: Required Readings

- Last+This week
  - Combinational Logic
    - P&P Chapter 3 until 3.3 + H&H Chapter 2

- This+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

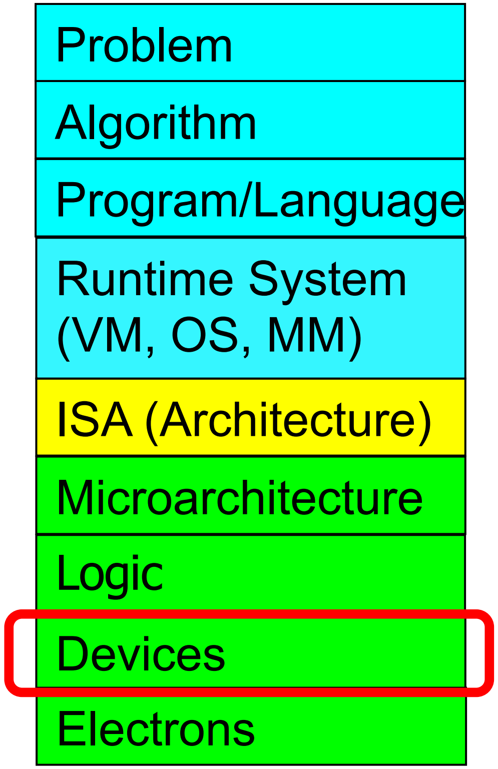
- Building blocks of modern computers
  - Transistors
  - Logic gates
- Combinational circuits
- Boolean algebra
- How to use Boolean algebra to represent combinational circuits
- Minimizing logic circuits

Recall: Transistors and Logic Gates

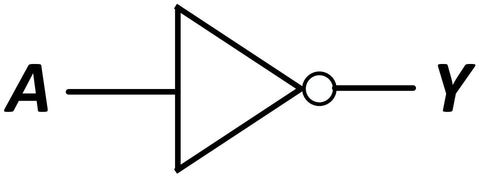
- **Now, we know how a MOS transistor works**
- How do we build logic out of MOS transistors?

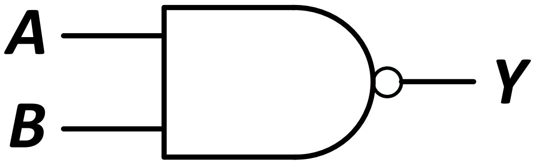
- **We construct basic logic structures out of individual MOS transistors**

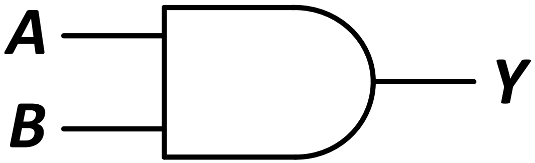
- These logical units are named logic gates
  - They implement simple Boolean functions



Recall: CMOS NOT, NAND, AND Gates



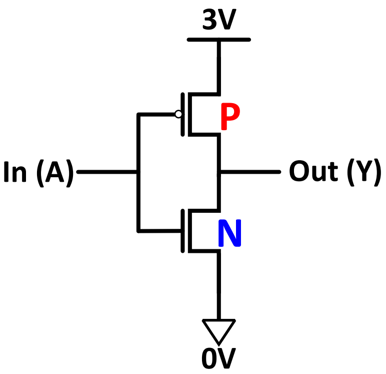


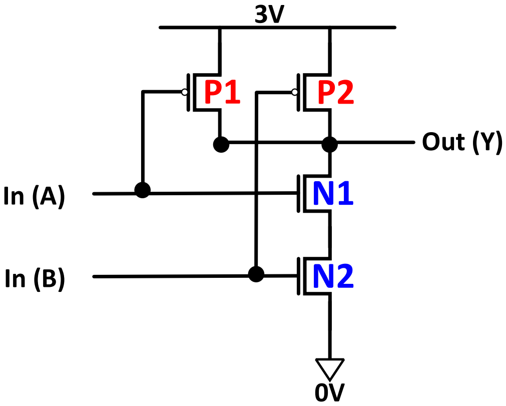


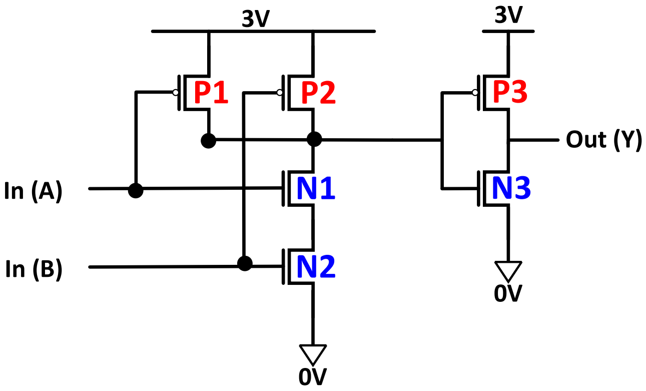
A	Y
0	1
1	0

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

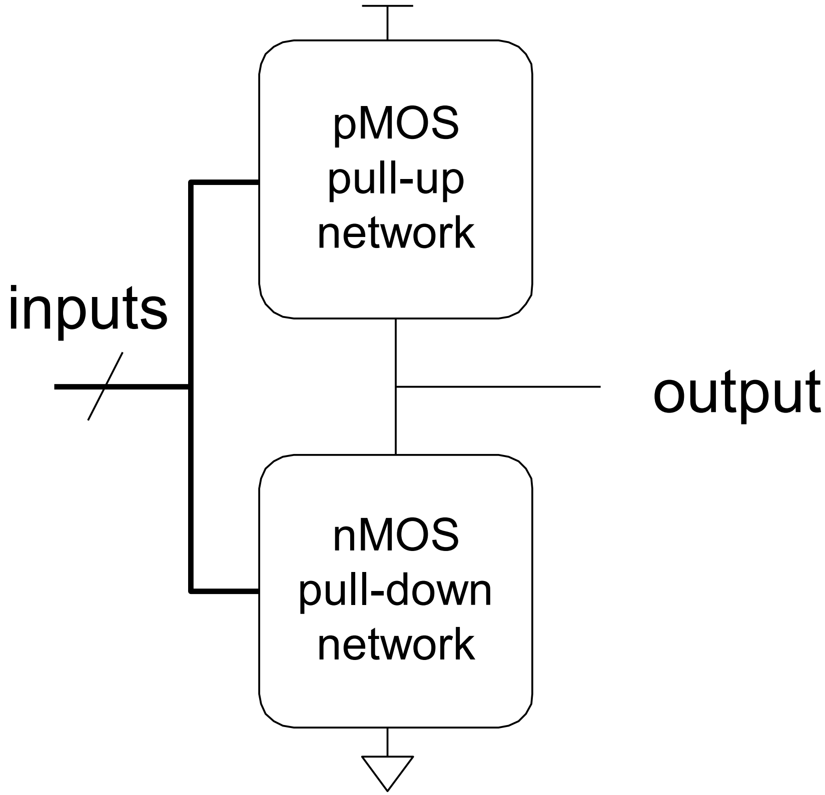






Recall: General CMOS Gate Structure

- The general form used to construct any inverting logic gate, such as: NOT, NAND, or NOR
  - The networks may consist of transistors in series or in parallel
  - When transistors are in parallel, the network is **ON** if one of the transistors is **ON**
  - When transistors are in series, the network is **ON** only if all transistors are **ON**



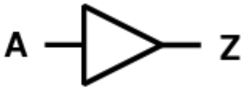
pMOS transistors are used for pull-up
nMOS transistors are used for pull-down

Recall: Digging Deeper: Power Consumption

- Dynamic Power Consumption
  - $C * V^2 * f$
    - C = capacitance of the circuit (wires and gates)
    - V = supply voltage
    - f = charging frequency of the capacitor
- Static Power consumption
  - $V * I_{leakage}$
    - supply voltage * leakage current
- Energy Consumption
  - Power * Time
- See more in H&H Chapter 1.8

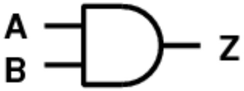
Recall: Common Logic Gates

Buffer



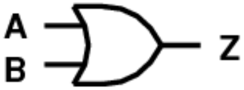
A	Z
0	0
1	1

AND



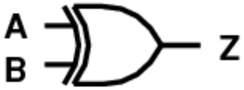
A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

OR



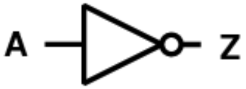
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

XOR



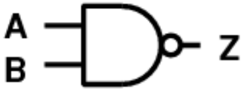
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

Inverter



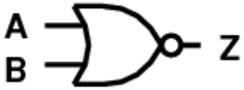
A	Z
0	1
1	0

NAND



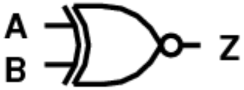
A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

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

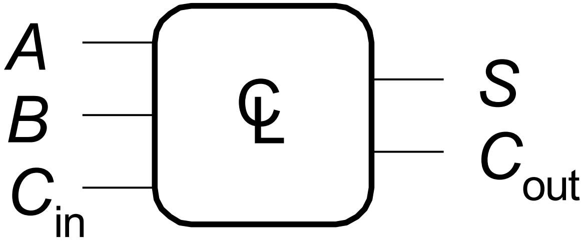
Boolean Equations

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):***

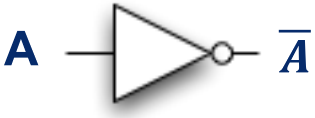
$$S = F(A, B, C_{in})$$
$$C_{out} = G(A, B, C_{in})$$



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

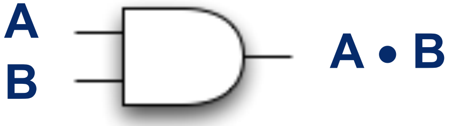
Recall: Boolean NOT / AND / OR

$\overline{A}$ *(reads "not A")* is 1 iff A is 0



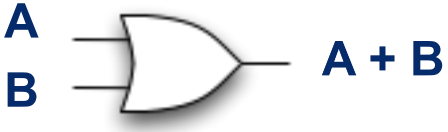
A	$\overline{A}$
0	1
1	0

A • B *(reads "A and B")* is 1 iff A and B are both 1



A	B	$A \bullet B$
0	0	0
0	1	0
1	0	0
1	1	1

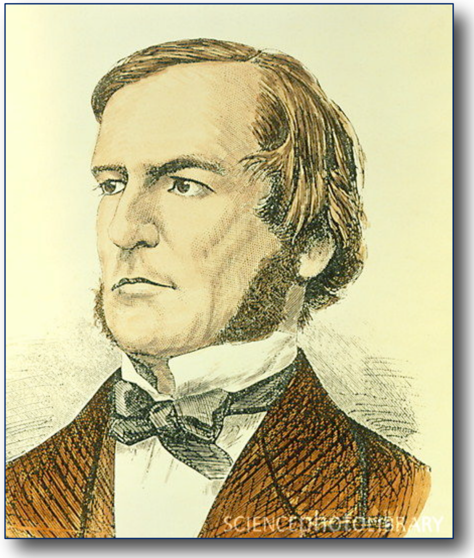
A + B *(reads "A or B")* is 1 iff either A or B is 1



A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Recall: Boolean Algebra: Big Picture

- An algebra on 1's and 0's
  - with AND, OR, NOT operations



- What you start with
  - Axioms: basic things about objects and operations you just assume to be true at the start
- What you derive first
  - Laws and theorems: allow you to manipulate Boolean expressions
  - …also allow us to do some simplification on Boolean expressions
- What you derive later
  - More "sophisticated" properties useful for manipulating digital designs represented in the form of Boolean equations

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

Recall: Boolean Algebra: Axioms

Formal version	*English version*
1. *B* contains at least two elements, *0* and *1*, *such that* $0 \neq 1$	Math formality…
2. *Closure* $a,b \in B$, (i) $a + b \in B$ (ii) $a \bullet b \in B$	Result of AND, OR stays in set you start with
3. *Commutative Laws*: $a,b \in B$, (i) (ii)	For primitive AND, OR of 2 inputs, order doesn't matter
4. *Identities*: $0, 1 \in B$ (i) (ii)	There are identity elements for AND, OR, that give you back what you started with
5. *Distributive Laws*: (i) (ii)	• distributes over +, just like algebra …but + distributes over •, also (!!)
6. *Complement*: (i) (ii)	There is a complement element; AND/ORing with it gives the identity elm.

Recall: Boolean Algebra: Duality

- Observation
  - All the axioms come in "dual" form
  - Anything true for an expression also true for its dual
  - So any derivation you could make that is true, can be flipped into dual form, and it stays true
- Duality — More formally
  - A dual of a Boolean expression is derived by replacing
    - Every AND operation with... an OR operation
    - Every OR operation with... an AND
    - Every constant 1 with... a constant 0
    - Every constant 0 with... a constant 1
    - But don't change any of the literals or play with the complements!

Example

$$a \bullet (b + c) = (a \bullet b) + (a \bullet c)$$

$$\rightarrow \; a + (b \bullet c) = (a + b) \bullet (a + c)$$

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.

9D.

10.

10D.

11.

11D.

Useful for simplifying expressions

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

Boolean Algebra: Proving Things

Proving theorems via axioms of Boolean Algebra:

EX: Prove the theorem: $X \bullet Y + X \bullet \overline{Y} = X$

Distributive (5)

Complement (6)

Identity (4)

EX2: Prove the theorem: $X + X \bullet Y = X$

Identity (4)

Distributive (5)

Identity (2)

Identity (4)

DeMorgan’s Law: Enabling Transformations

DeMorgan's Law:

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

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

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

$$F = A + B + C$$

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

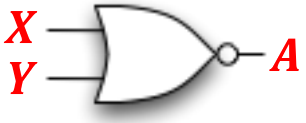
$$\boldsymbol{F} = \overline{\overline{(\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C})}} = \overline{(\overline{\boldsymbol{A}}.\overline{\boldsymbol{B}}.\overline{\boldsymbol{C}})}$$

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

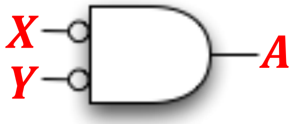
DeMorgan’s Law (Continued)

These are conversions between different types of logic functions
They can prove useful if you do not have every type of gate

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

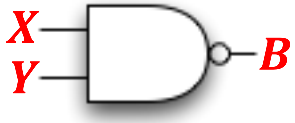


NOR is equivalent to AND with inputs complemented

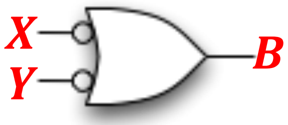


X	Y	$\overline{X+Y}$	$\bar{X}$	$\bar{Y}$	$\bar{X}\bar{Y}$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

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



NAND is equivalent to OR with inputs complemented



X	Y	$\overline{XY}$	$\bar{X}$	$\bar{Y}$	$\bar{X}+\bar{Y}$
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

Sum of Products Form: Key Idea

- Assume we have the truth table of a Boolean Function

- 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

- **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

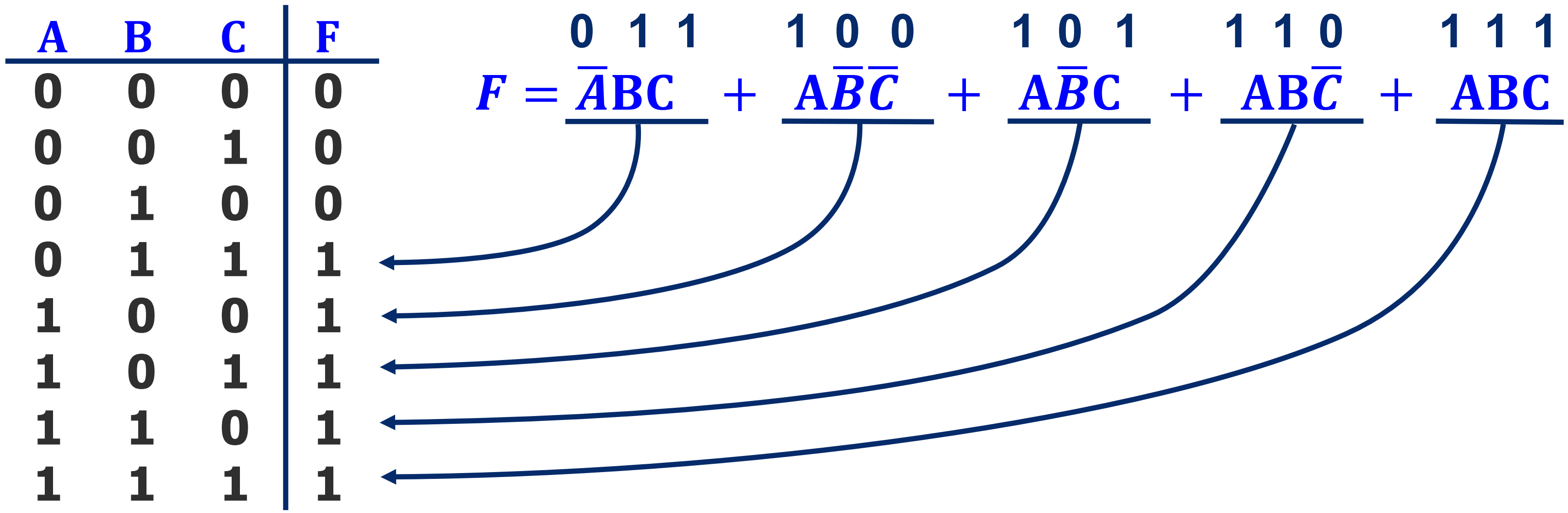
- A Boolean function can have many alternative Boolean expressions
    - i.e., many alternative Boolean expressions (and gate realizations) may have the same truth table (and function)
    - If they all say the same thing, why do we care?
        - Different Boolean expressions lead to different gate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

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

Two-Level Canonical Forms

Sum of Products Form (SOP)

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



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

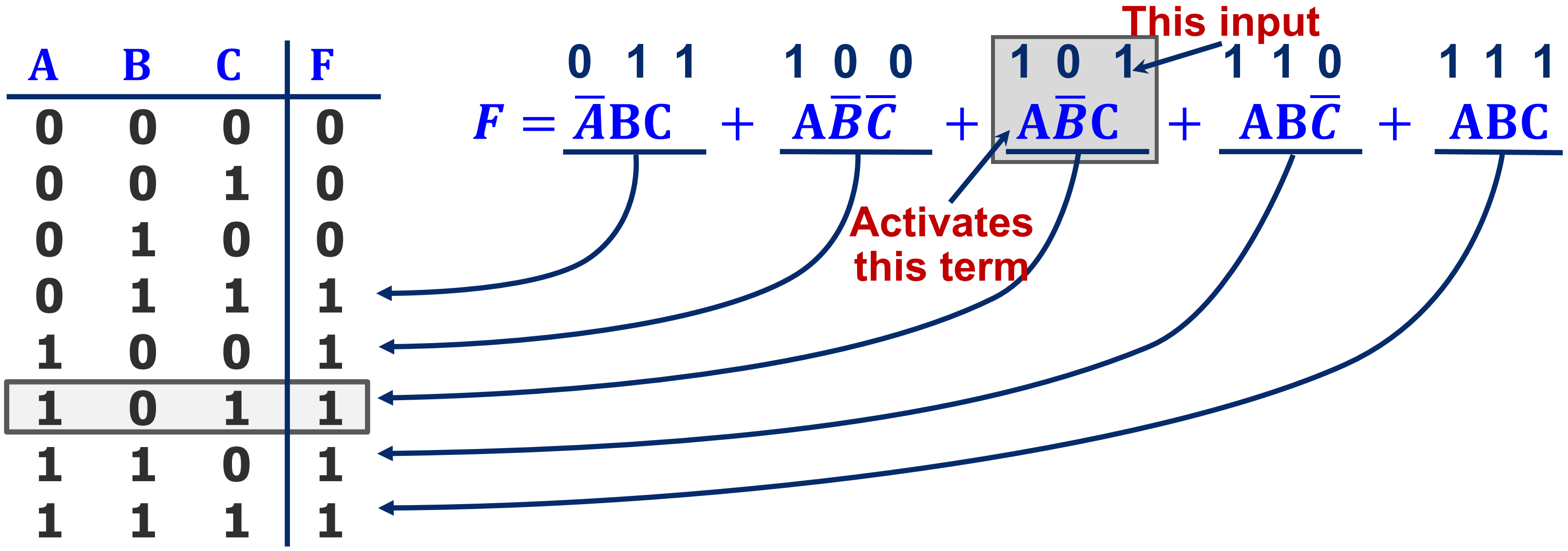
$$F = \overset{0\ 1\ 1}{\underline{\bar{A}BC}} + \overset{1\ 0\ 0}{\underline{A\bar{B}\bar{C}}} + \overset{1\ 0\ 1}{\underline{A\bar{B}C}} + \overset{1\ 1\ 0}{\underline{AB\bar{C}}} + \overset{1\ 1\ 1}{\underline{ABC}}$$

- 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?



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

$$\begin{array}{ccccccccccc} 0\,1\,1 & & 1\,0\,0 & & 1\,0\,1 & & 1\,1\,0 & & 1\,1\,1 \\ F = \bar{A}BC & + & A\bar{B}\bar{C} & + & \boxed{A\bar{B}C} & + & AB\bar{C} & + & ABC \end{array}$$

- 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

Aside: Notation for SOP

- 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 Forms

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

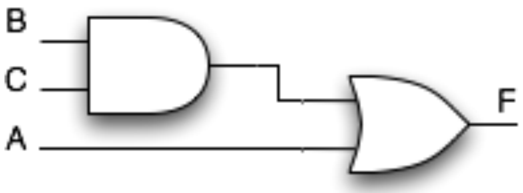
F in canonical form:

$$F(A,B,C) = \sum m(3,4,5,6,7)$$
$$= m3 + m4 + m5 + m6 + m7$$

$F =$

canonical form ≠ minimal form

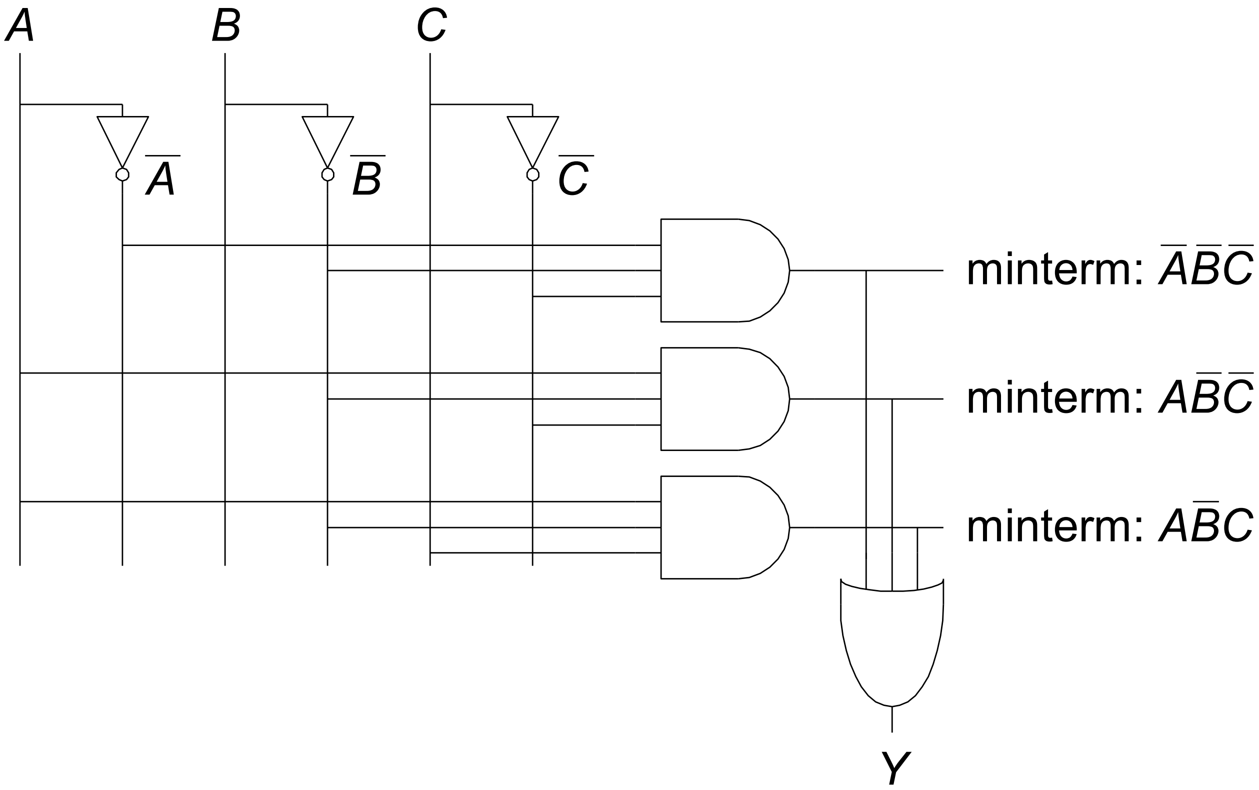
F



2-Level AND/OR Realization

From Logic to Gates

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


A
B
C
$\overline{A}$
$\overline{B}$
$\overline{C}$
minterm: $\overline{A}\overline{B}\overline{C}$
minterm: $A\overline{B}\overline{C}$
minterm: $A\overline{B}C$
Y

Alternative Canonical Form: POS

We can have another from of representation

DeMorgan of SOP of $\overline{F}$

A product of sums (POS)

products

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

sums

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

This input

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

Activates this term

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

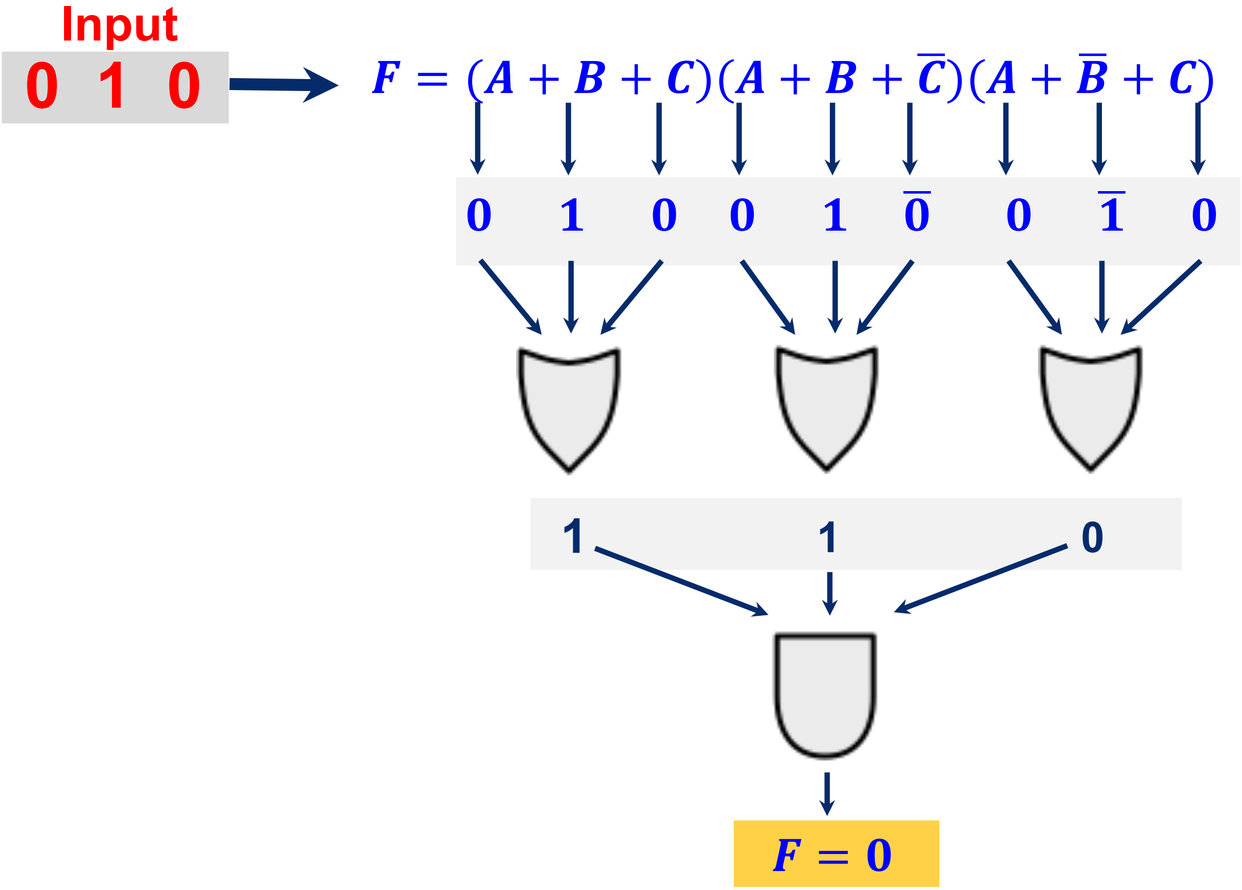
For the given input, only the shaded sum term will equal 0

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

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

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

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

$0 \quad 1 \quad 0 \quad 0 \quad 1 \quad \overline{0} \quad 0 \quad \overline{1} \quad 0$

1 1 0

$F = 0$

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

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

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

$$A \; + \; \overline{B} \; + \; C$$

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 $\overline{F}$!!

Canonical POS Forms

Product of Sums / Conjunctive Normal Form / Maxterm Expansion

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

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

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

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

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

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

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

$$= \prod M(0,1,2) \longrightarrow = \prod M(3,4,5,6,7)$$

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

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

$$= \prod M(0,1,2) \longrightarrow = \sum m(0,1,2)$$

Combinational Building Blocks used in Modern Computers

Combinational Building Blocks

- Combinational logic is often grouped into larger building blocks to build more complex systems

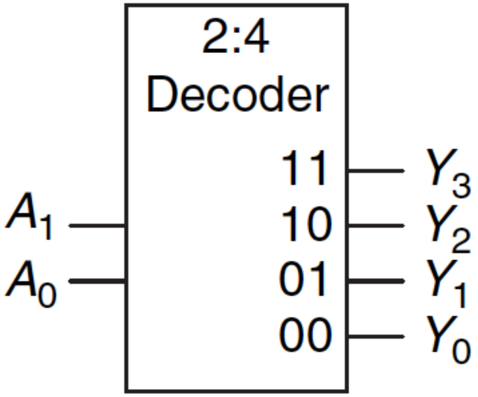
- Hides the unnecessary gate-level details to emphasize the function of the building block

- We now look at:
    - Decoder
    - Multiplexer
    - Full adder
    - PLA (Programmable Logic Array)

Decoder

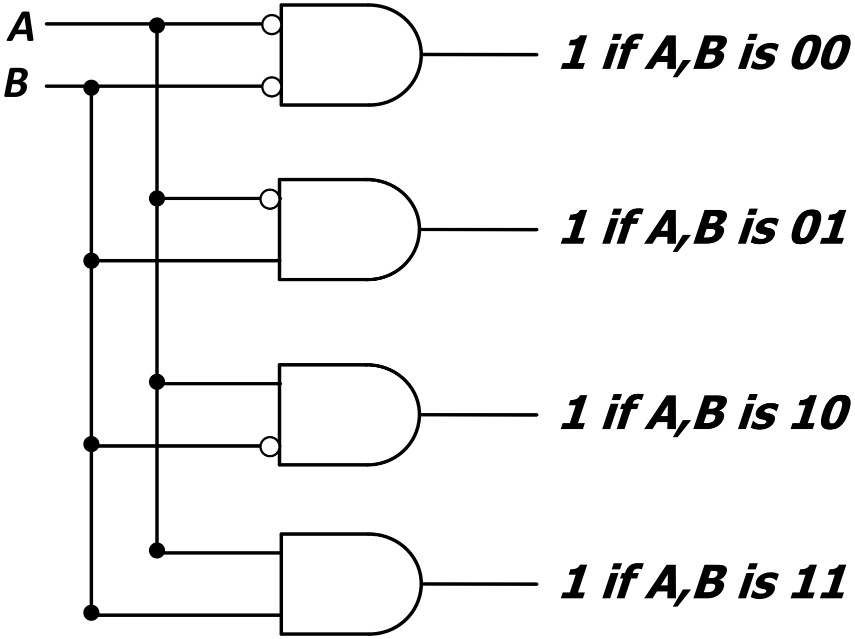
- "Input pattern detector"
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The one output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

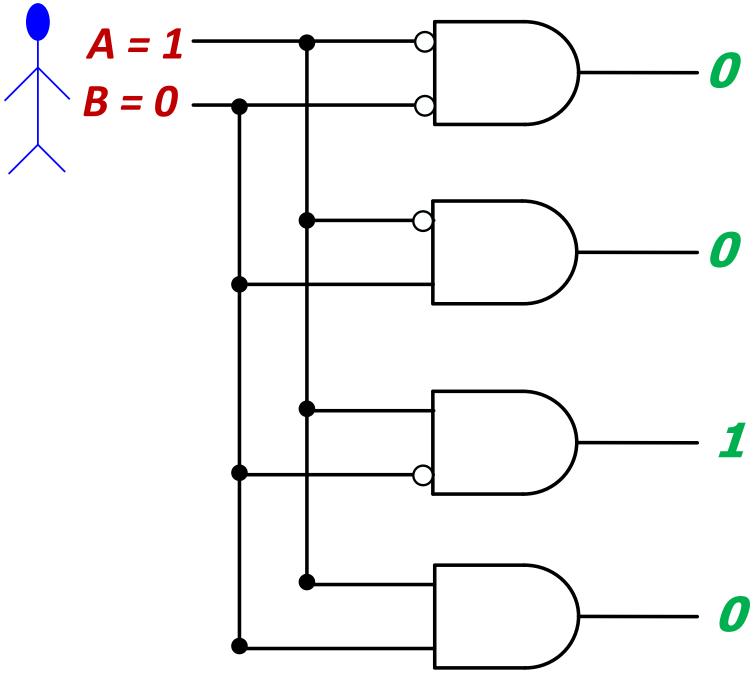
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



Decoder (I)

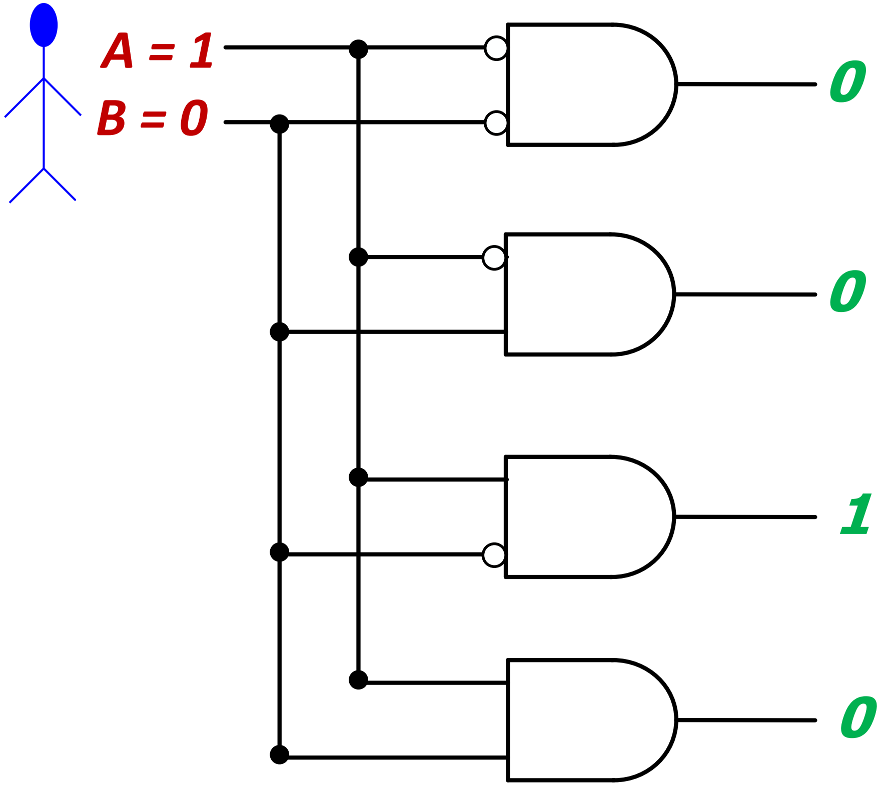
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The one output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect





Decoder (II)

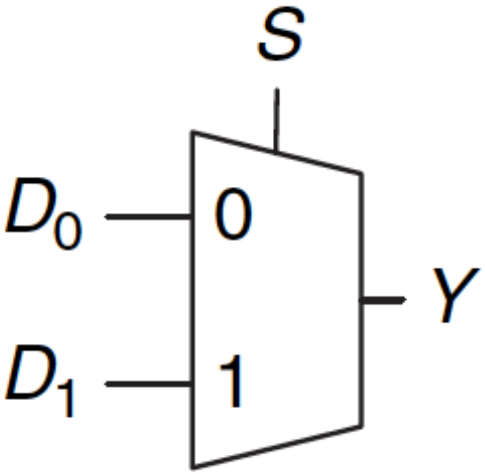
- The decoder is useful in determining how to interpret a bit pattern
  - **It could be the address of a row in DRAM, that the processor intends to read from**
  - **It could be an instruction in the program and the processor has to decide what action to do! (based on *instruction opcode*)**



Multiplexer (MUX), or Selector

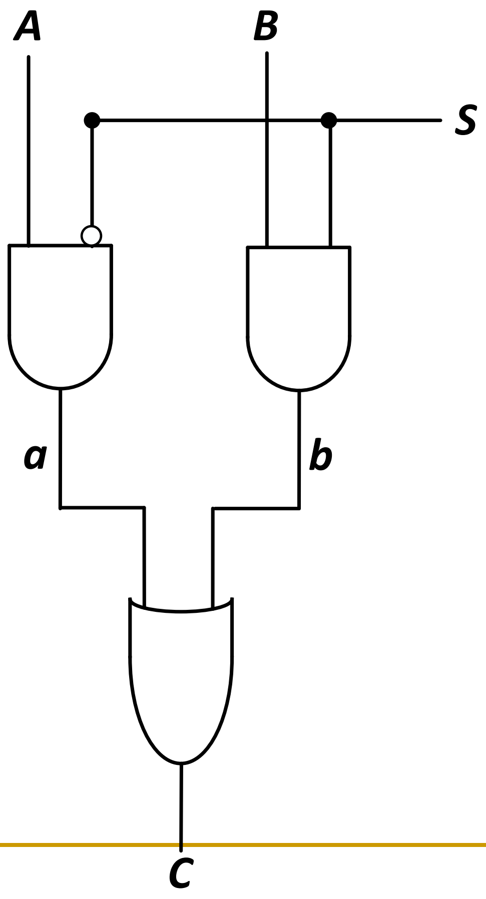
- Selects one of the *N* inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called select
- Example: 2-to-1 MUX

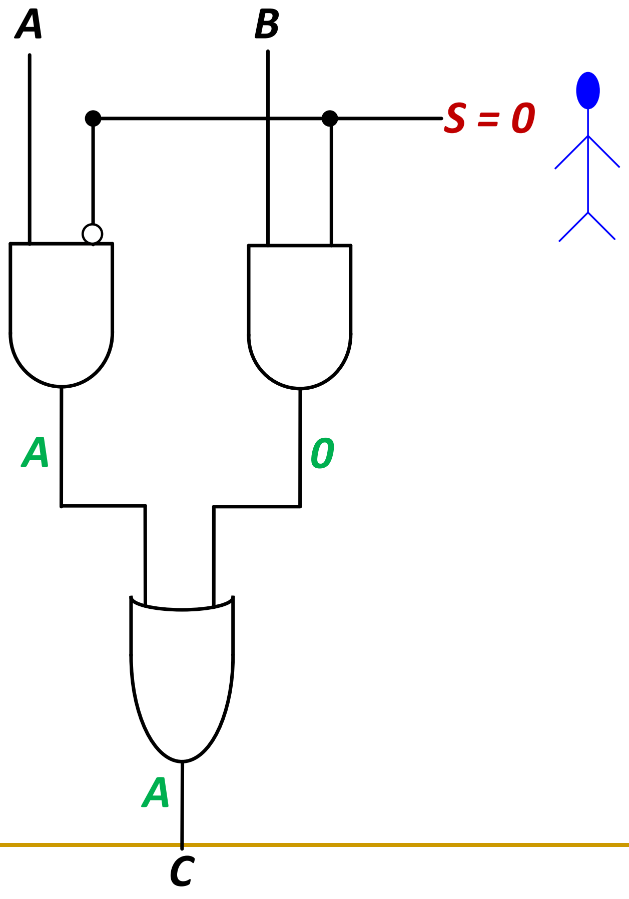
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



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

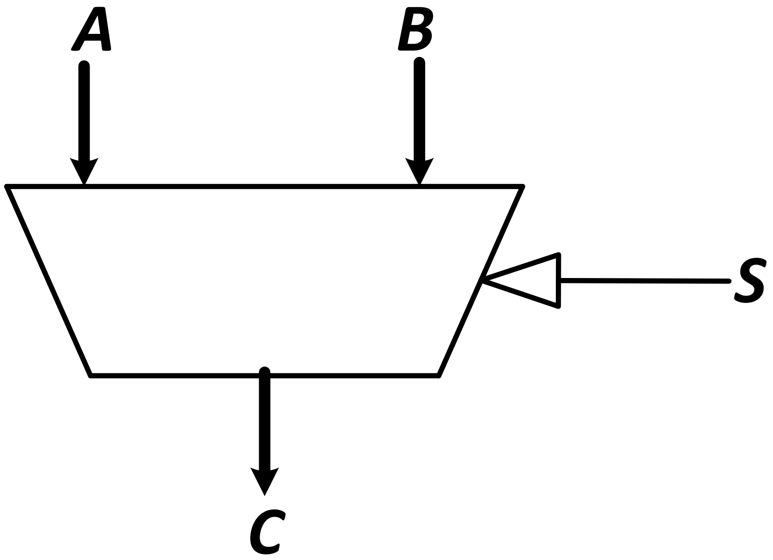




Multiplexer (MUX), or Selector (III)

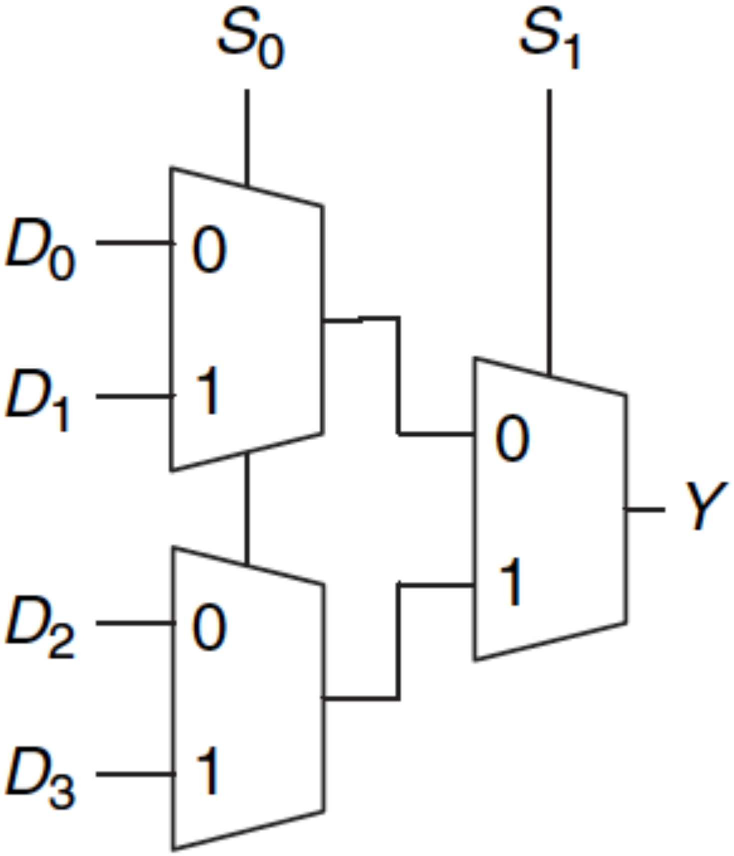
- The output C is always connected to either the input A or the input B
  - Output value depends on the value of the select line S

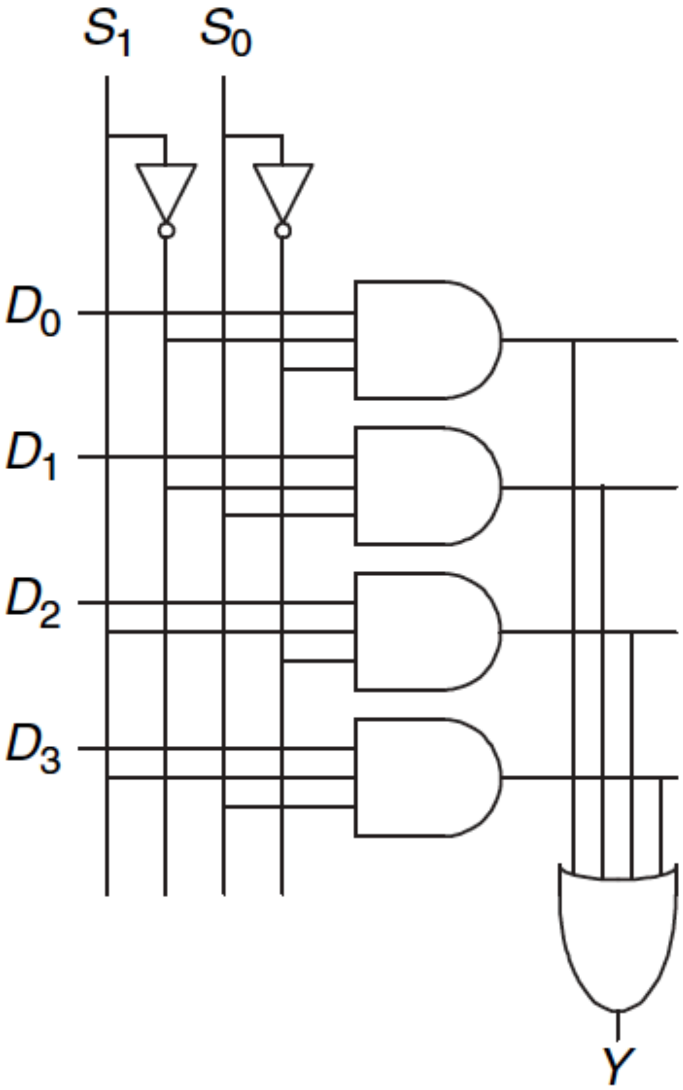
S	C
0	A
1	B



- 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





Full Adder (I)

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

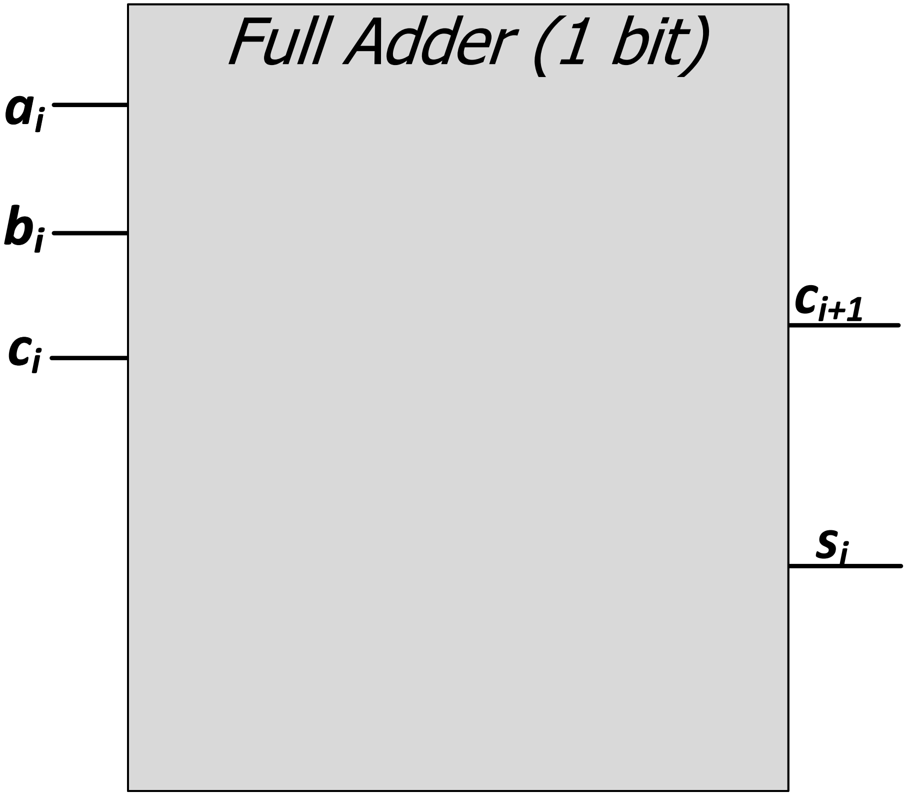
$$\begin{array}{r}
a_{n-1}a_{n-2} \dots a_1a_0 \\
b_{n-1}b_{n-1}b_{n-2} \dots b_1b_0 \\
C_n\ C_{n-1} \quad \dots \quad C_1 \\
\hline
S_{n-1} \quad \dots \quad S_1S_0
\end{array}$$

- 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**

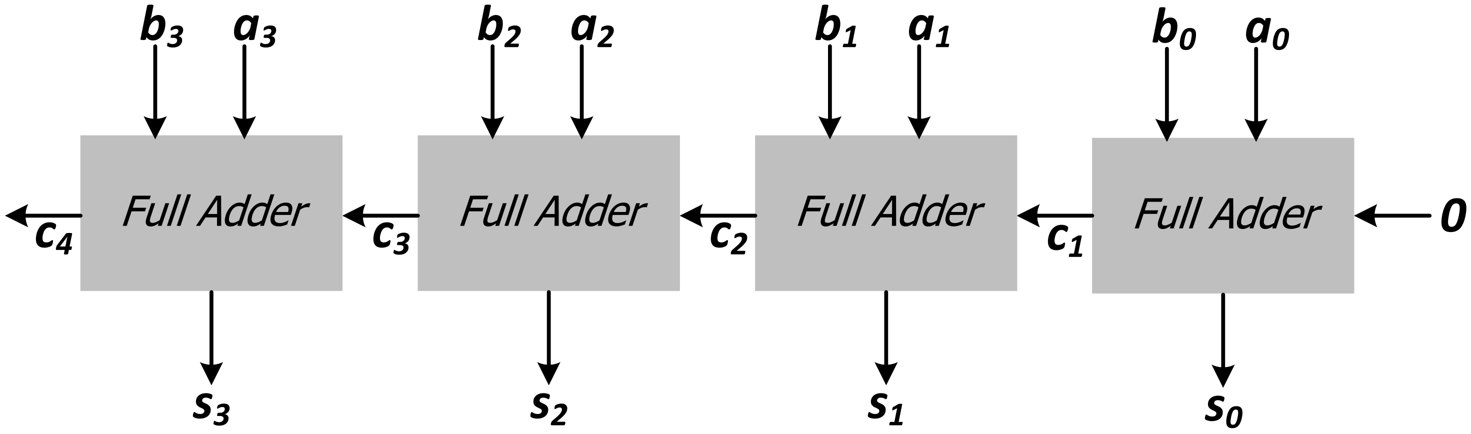


$$\begin{array}{r} a_{n-1}a_{n-2}\ldots a_1a_0 \\ b_{n-1}b_{n-2}\ldots b_1b_0 \\ C_n\,C_{n-1}\quad\ldots\quad C_1 \\ \hline S_{n-1}\quad\ldots\quad S_1S_0 \end{array}$$

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



Full Adder, Full Adder, Full Adder, Full Adder

b_3 a_3 b_2 a_2 b_1 a_1 b_0 a_0

c_4 c_3 c_2 c_1 0

s_3 s_2 s_1 s_0

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

Adder Design: Ripple Carry Adder

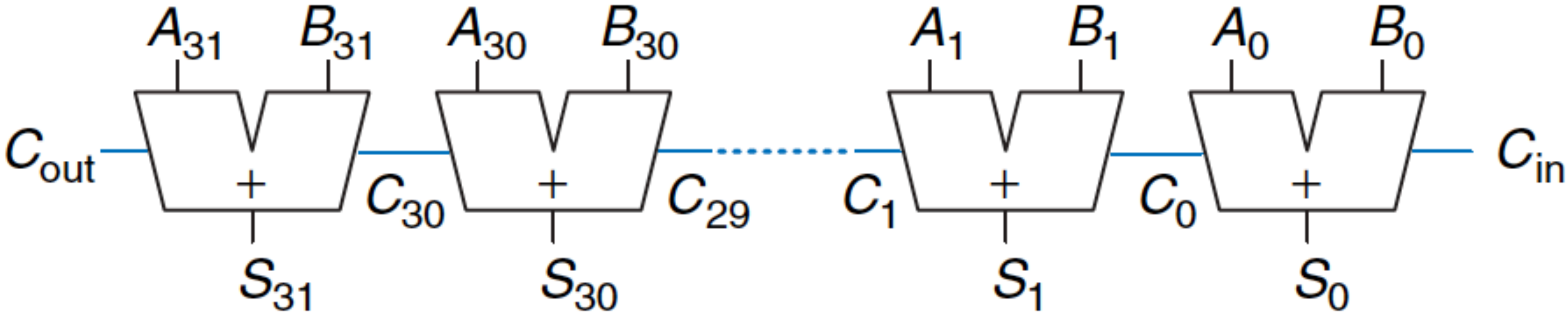
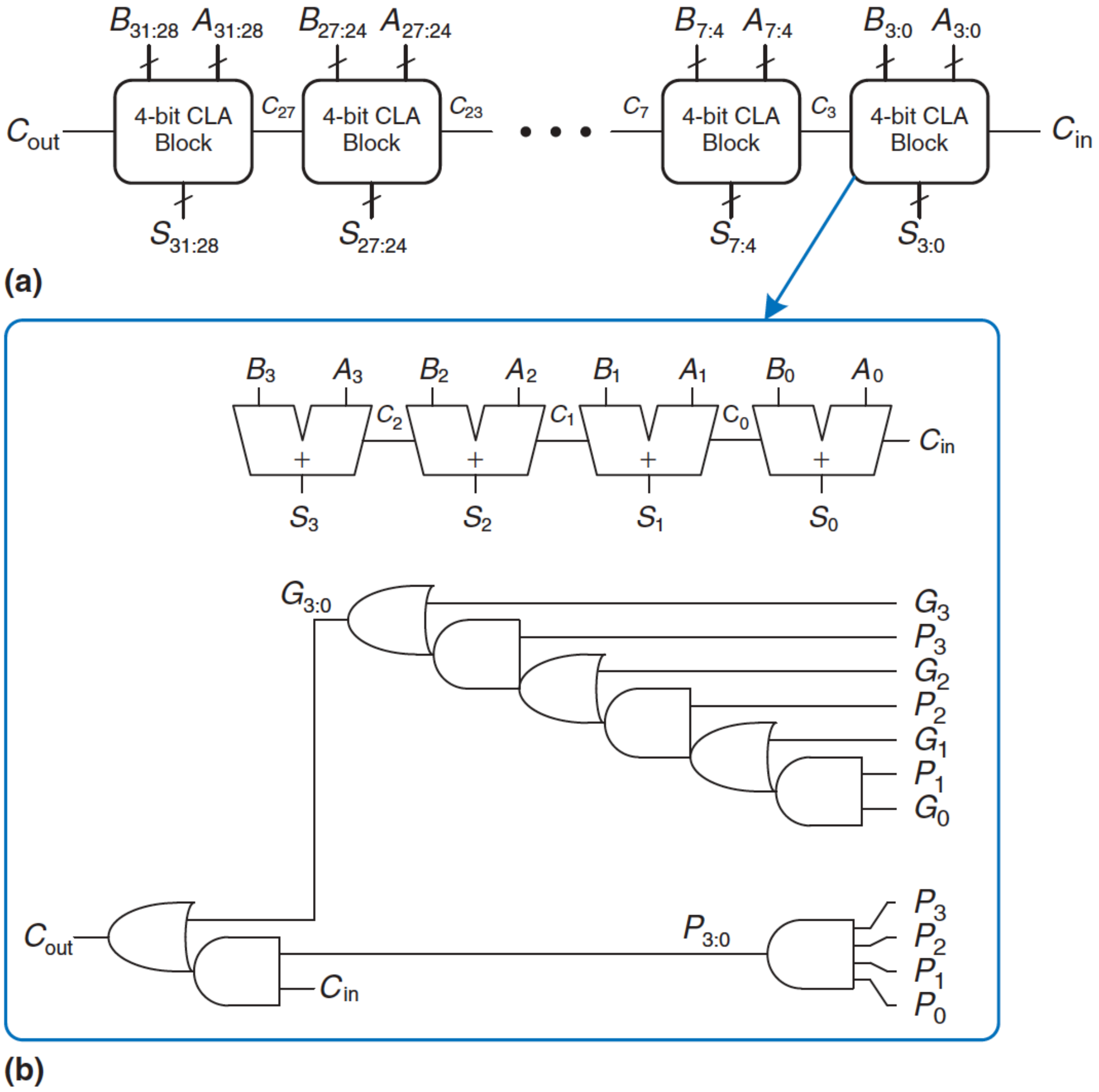


Figure 5.5 **32-bit ripple-carry adder**

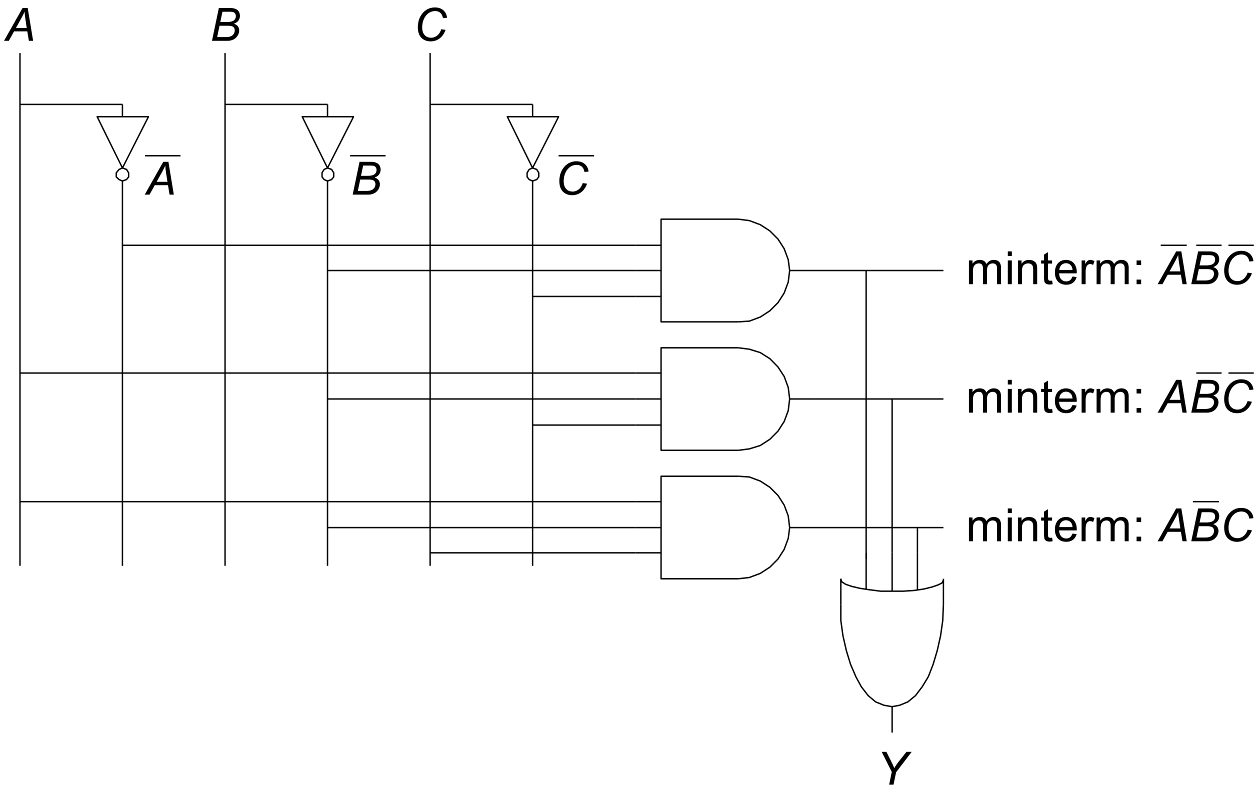
Adder Design: Carry Lookahead Adder



PLA: Recall: From Logic to Gates

- **SOP (sum-of-products) leads to two-level logic**

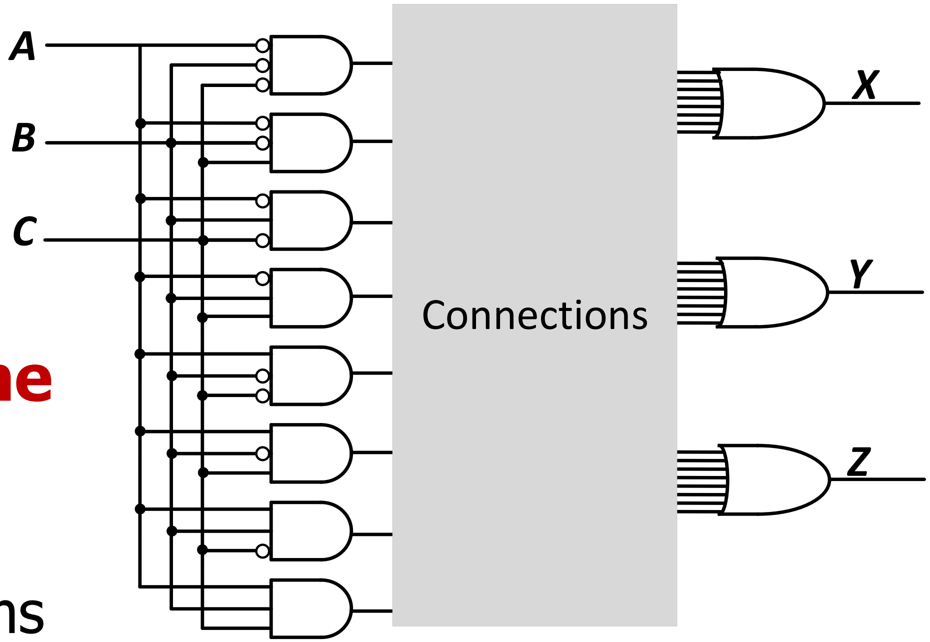
- Example: $\boldsymbol{Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C)}$



A
B
C
$\overline{A}$
$\overline{B}$
$\overline{C}$
minterm: $\overline{A}\,\overline{B}\,\overline{C}$
minterm: $A\overline{B}\,\overline{C}$
minterm: $A\overline{B}C$
Y

The Programmable Logic Array (PLA)

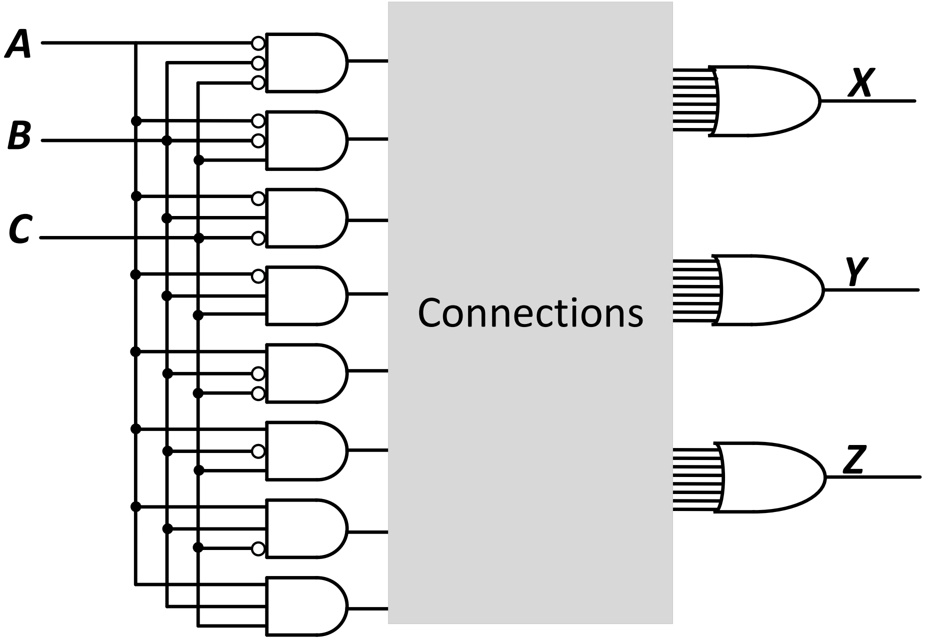
- The below logic structure is a very common building block for implementing any collection of logic functions one wishes to



- 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

The Programmable Logic Array (PLA)

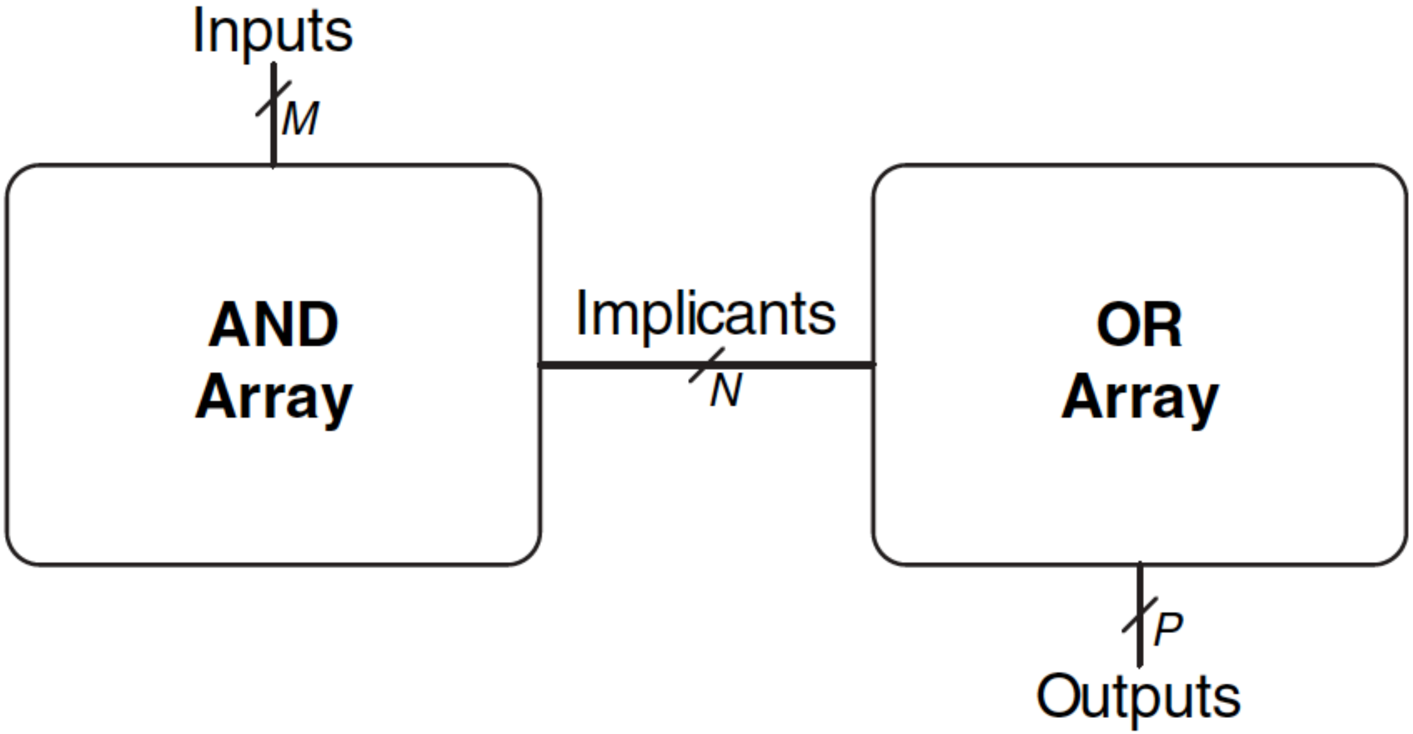
- How do we implement a logic function?
  - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP
  - This is a simple programmable logic



- **Programming a PLA**: we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function

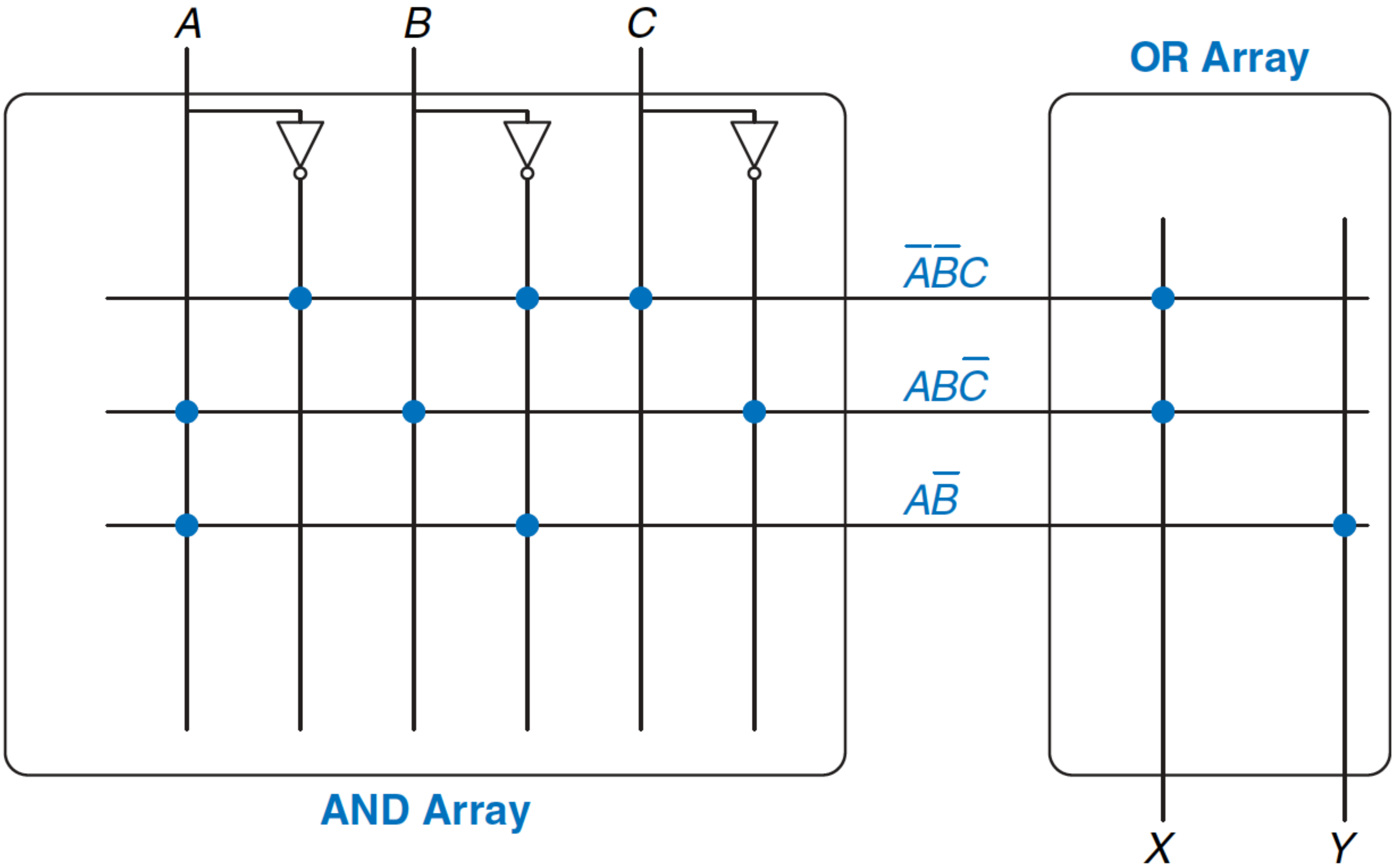
- Have you seen any other type of programmable logic?
  - Yes! An FPGA…
  - An FPGA uses more advanced structures, as we saw in Lecture 3

PLA Example (I)



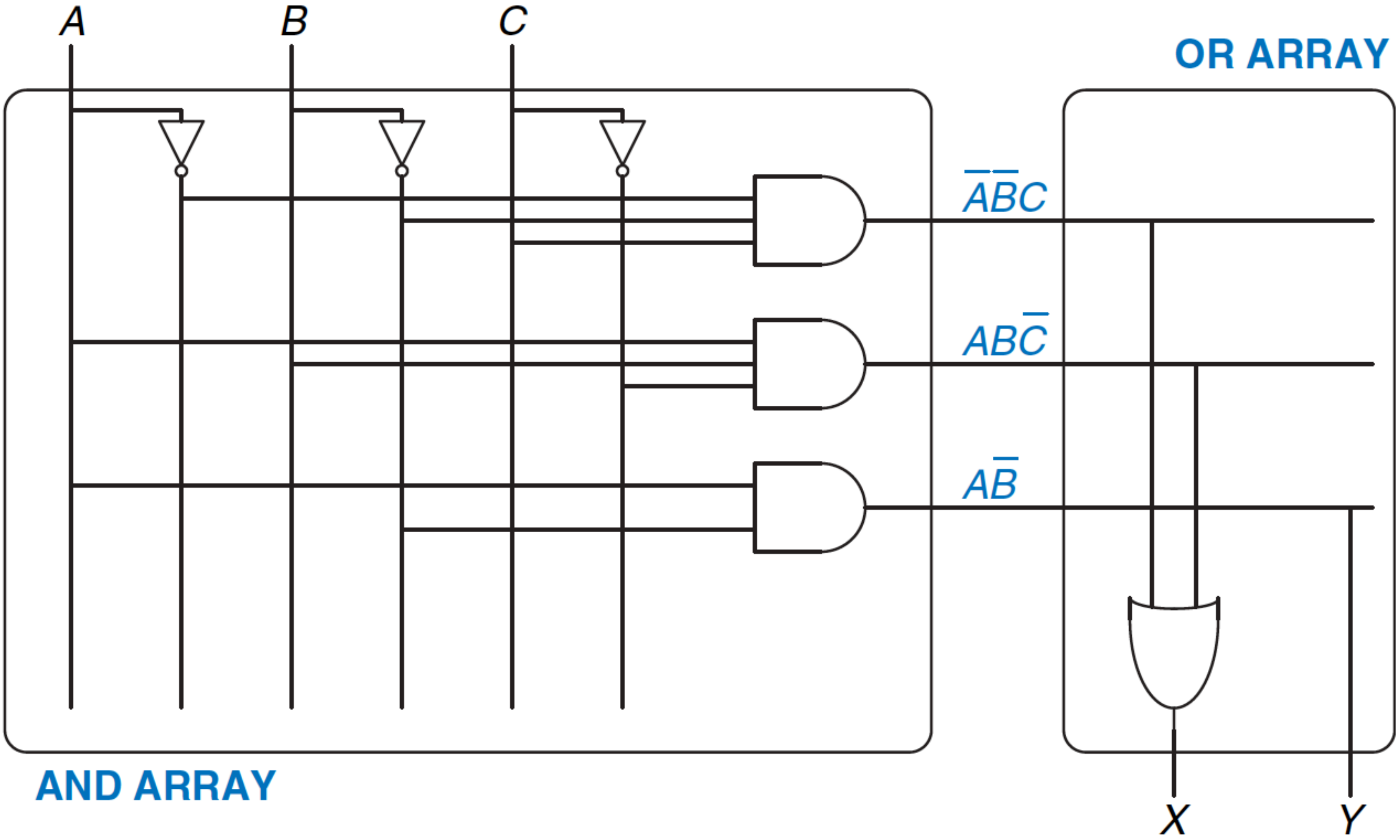
Read H&H Chapter 5.6.1

PLA Example Function (II)



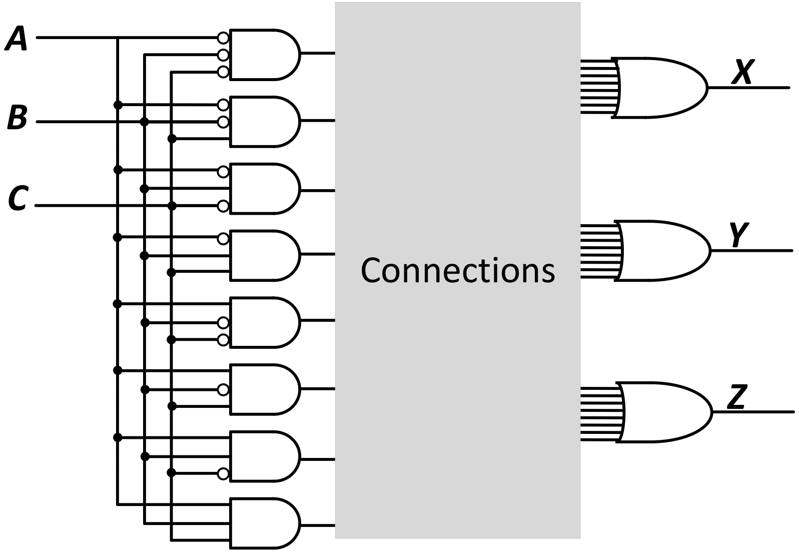
Read H&H Chapter 5.6.1

PLA Example Function (III)



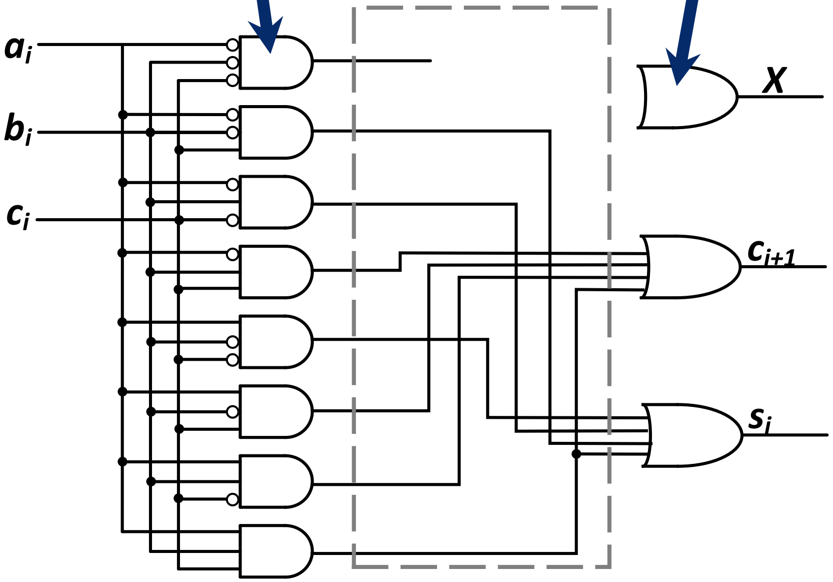
Read H&H Chapter 5.6.1

Implementing a Full Adder Using a PLA



This input should not be connected to any outputs

We do not need this output



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

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 Building Blocks

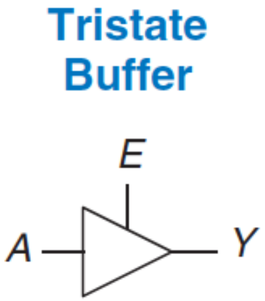
- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6

- 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

- 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

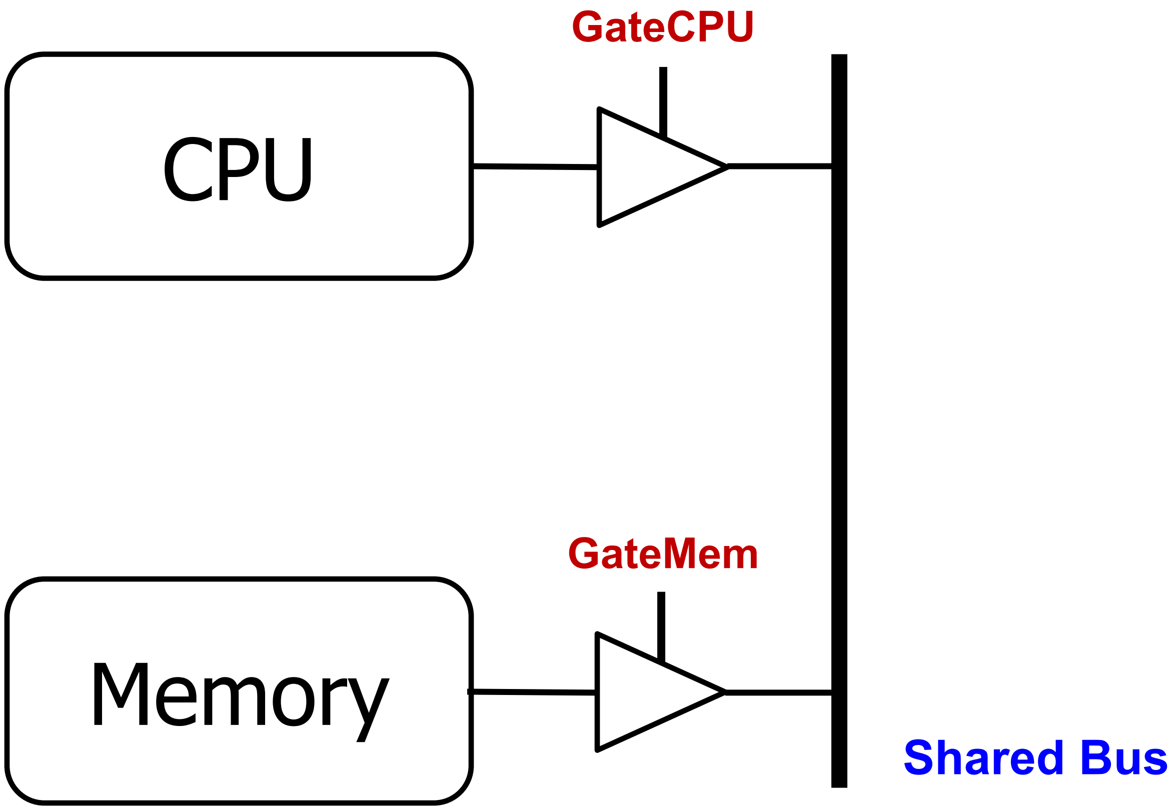
Figure 2.40 Tristate buffer

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

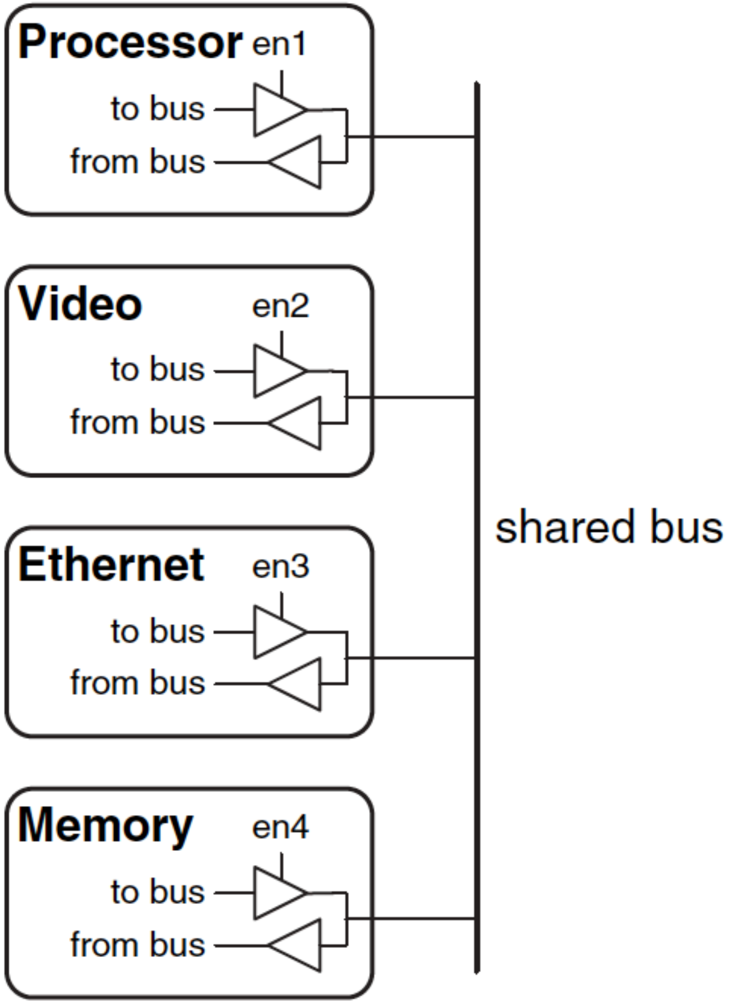
Example: Use of Tri-State Buffers

- 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

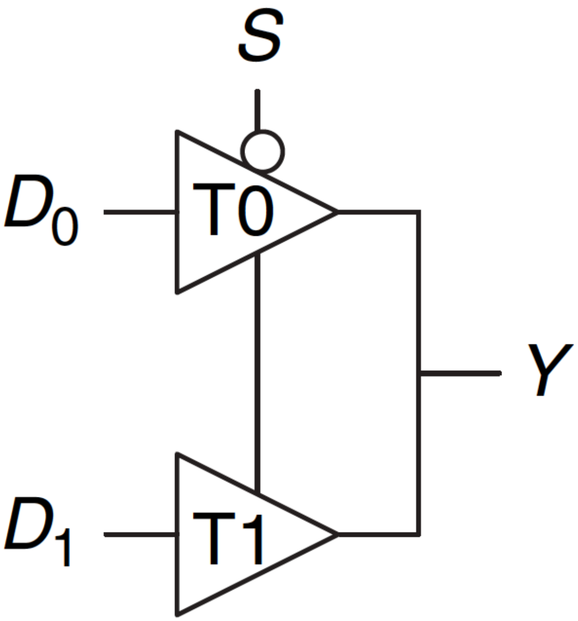
Example Design with Tri-State Buffers



Another Example

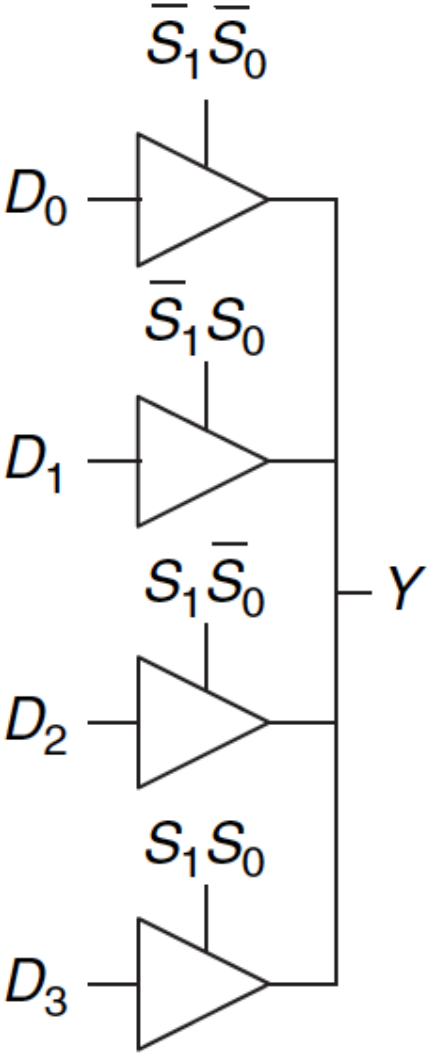


Multiplexer Using Tri-State Buffers



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

Figure 2.56 **Multiplexer using tristate buffers**



Digital Design & Computer Arch.

Lecture 5: Combinational Logic II

Prof. Onur Mutlu

ETH Zürich
Spring 2020
5 March 2020

We did not cover the remaining slides. They are for your preparation for the next lecture.

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

$Y = AB$

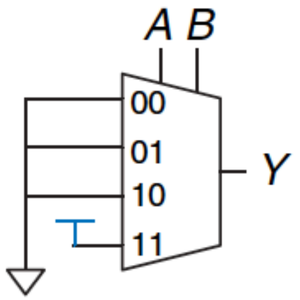
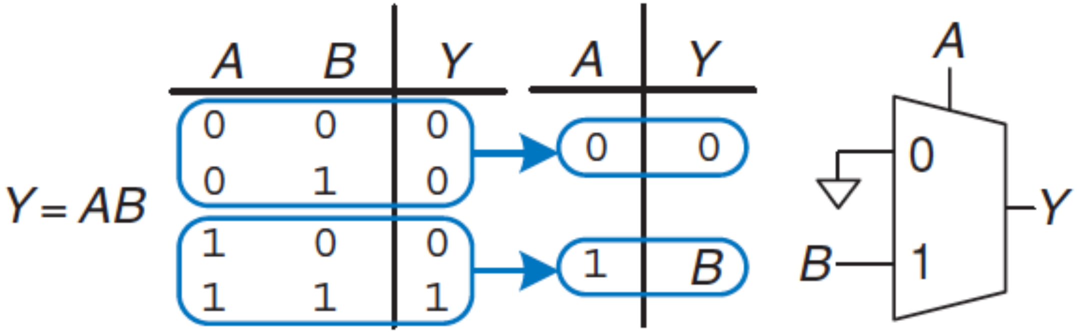
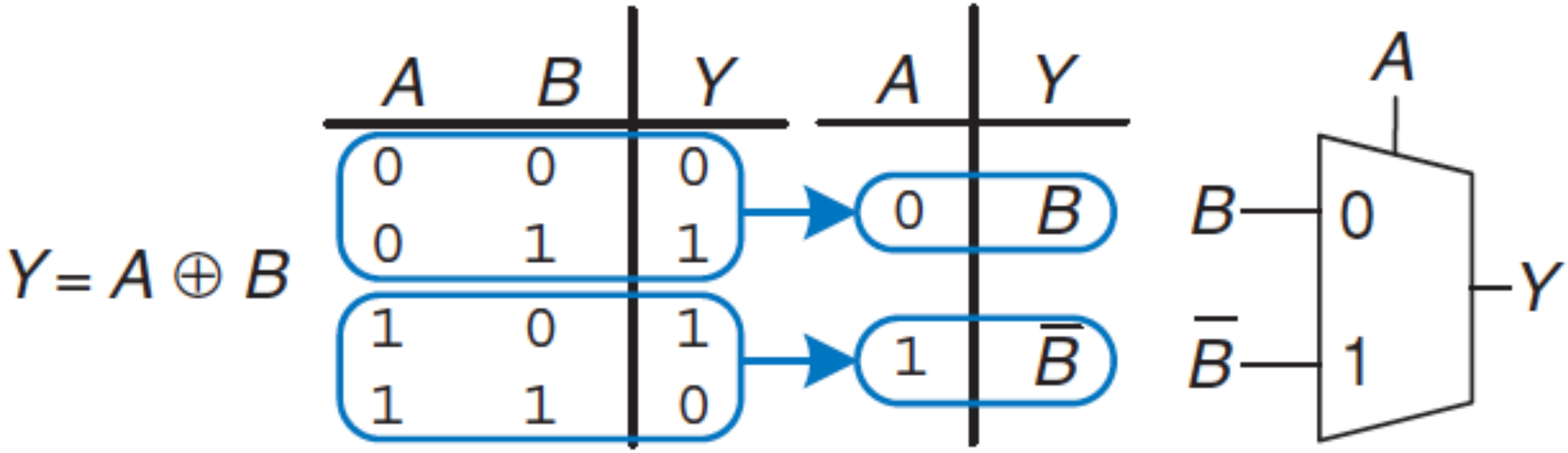


Figure 2.59 **4:1 multiplexer implementation of two-input AND function**



Aside: Logic Using Multiplexers (II)

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



$Y = A \oplus B$

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

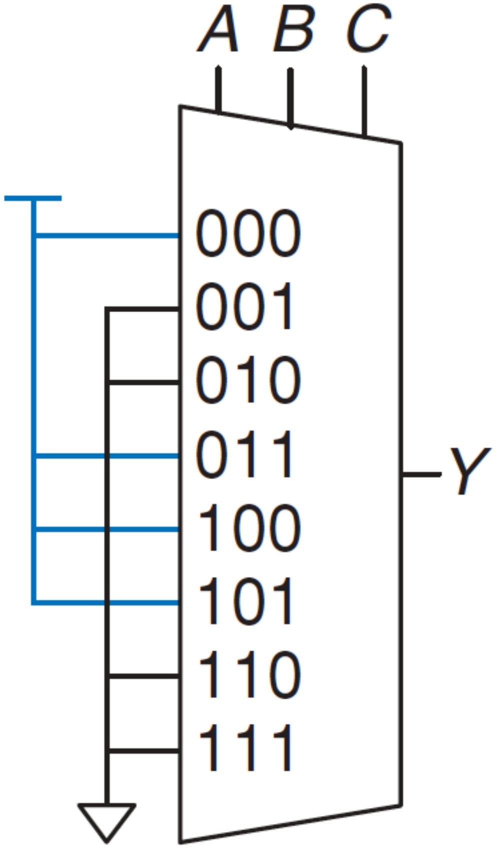
A	Y
0	B
1	$\overline{B}$

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

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



Aside: Logic Using Decoders (I)

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

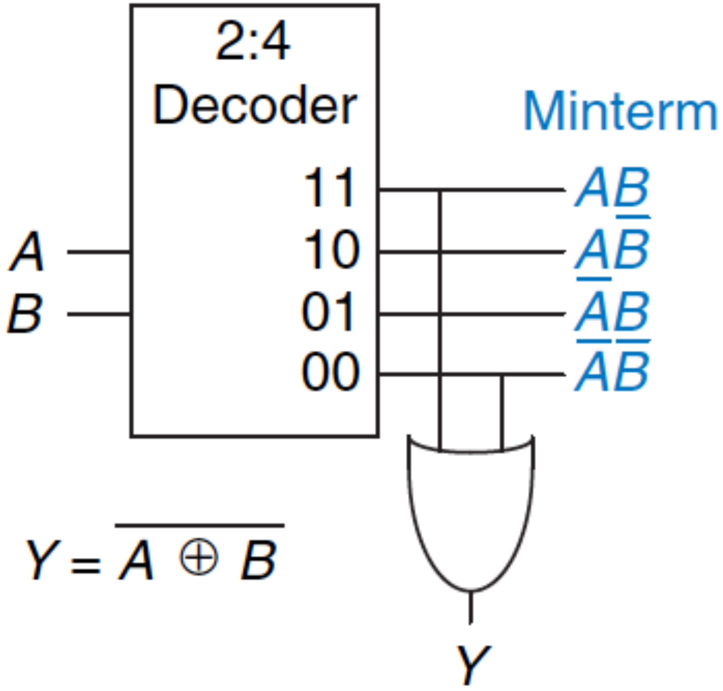
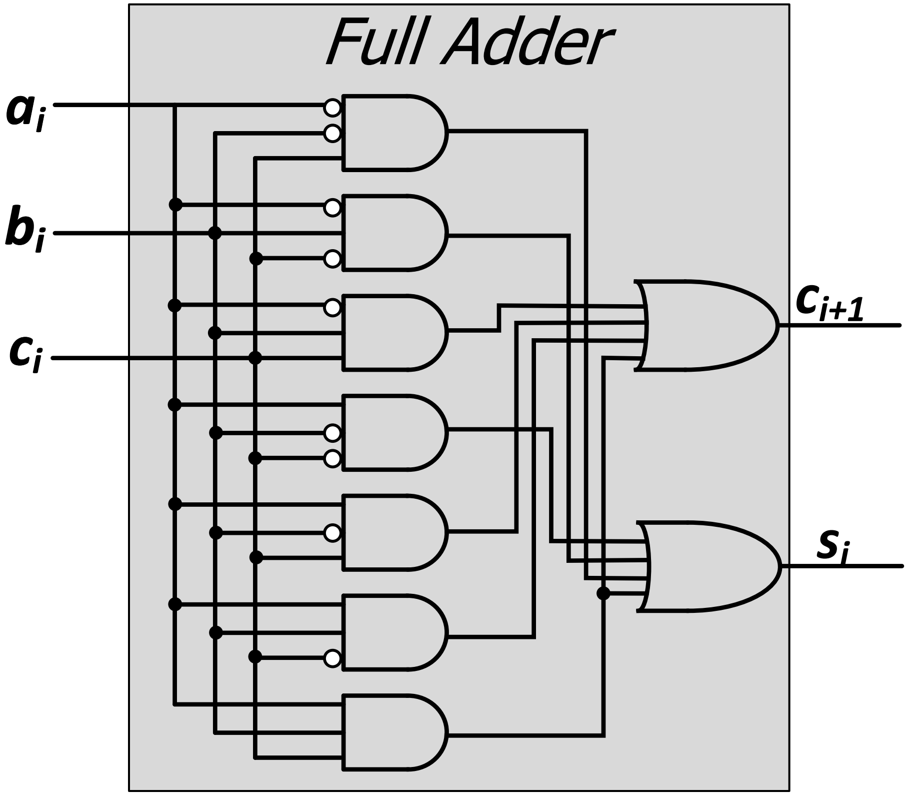


Figure 2.65 **Logic function using decoder**

Read H&H Chapter 2.8

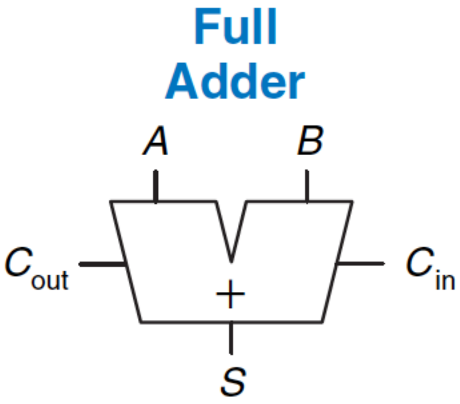
Logic Simplification: Karnaugh Maps (K-Maps)

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?

Quick Recap on Logic Simplification

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

$$\boxed{F = \sim A(A + B) + (B + AA)(A + \sim B)}$$

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

$$\boxed{F = A + B}$$

- 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		
1	1	1

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

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

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

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

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

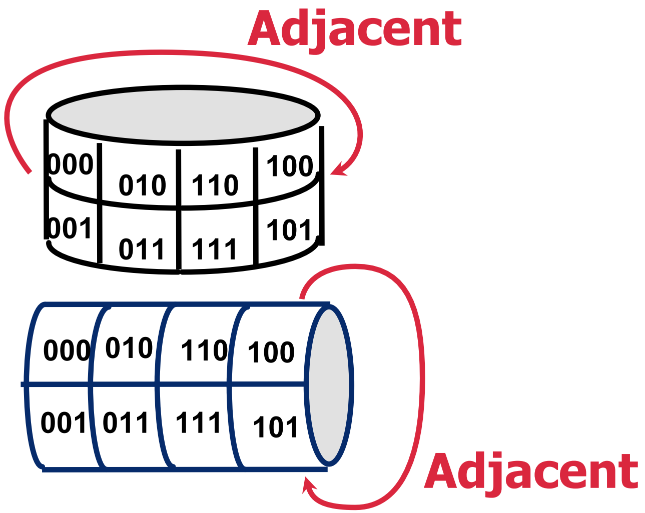
4-variable K-map

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

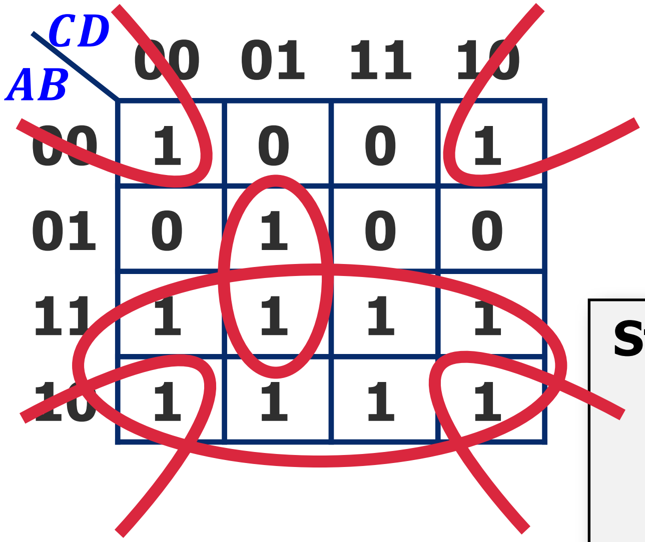


K-map adjacencies go "around the edges"

Wrap around from first to last column

Wrap around from top row to bottom row

K-map Cover - 4 Input Variables



AB \ CD	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

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

$$\mathrm{F} = \mathrm{A} + \overline{B}\,\overline{D} + \mathrm{B}\overline{C}D$$

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

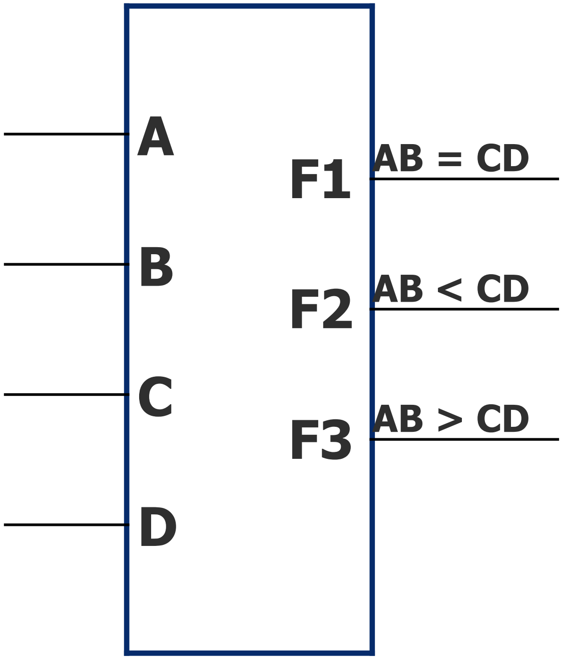
K-map Rules

- **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



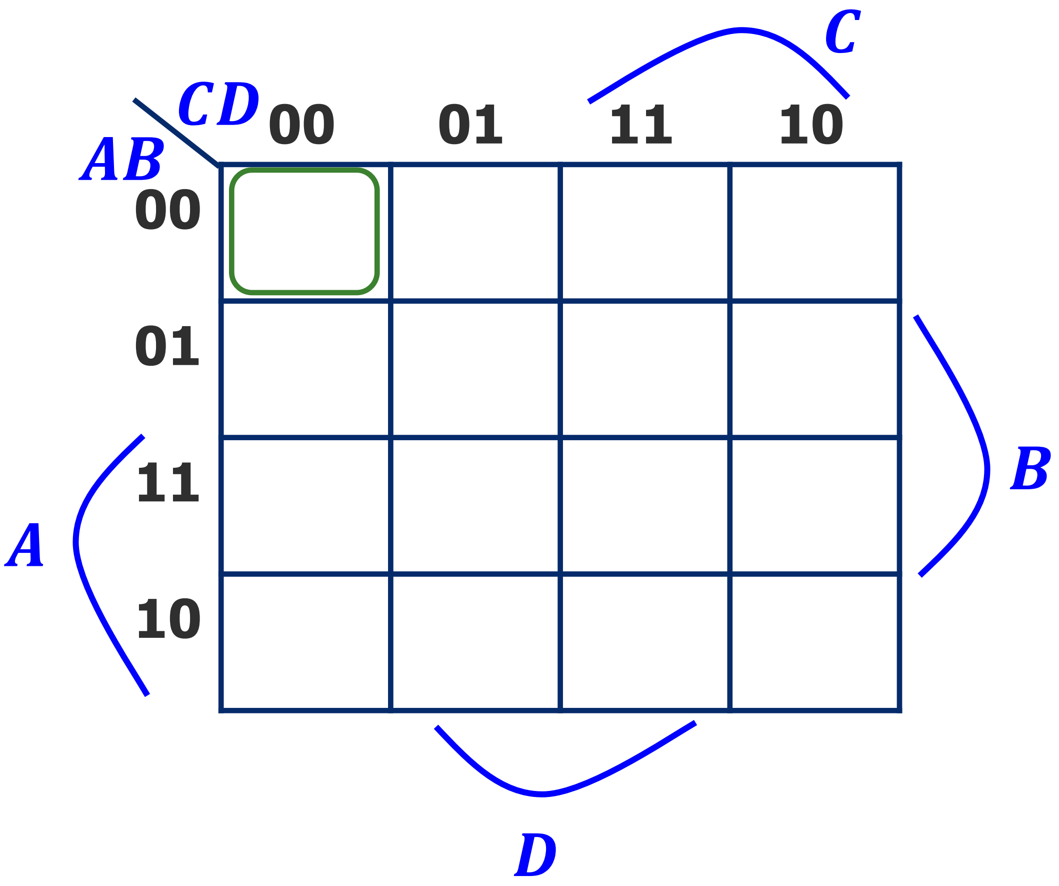
A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

Design Approach:

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

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

K-map for F1

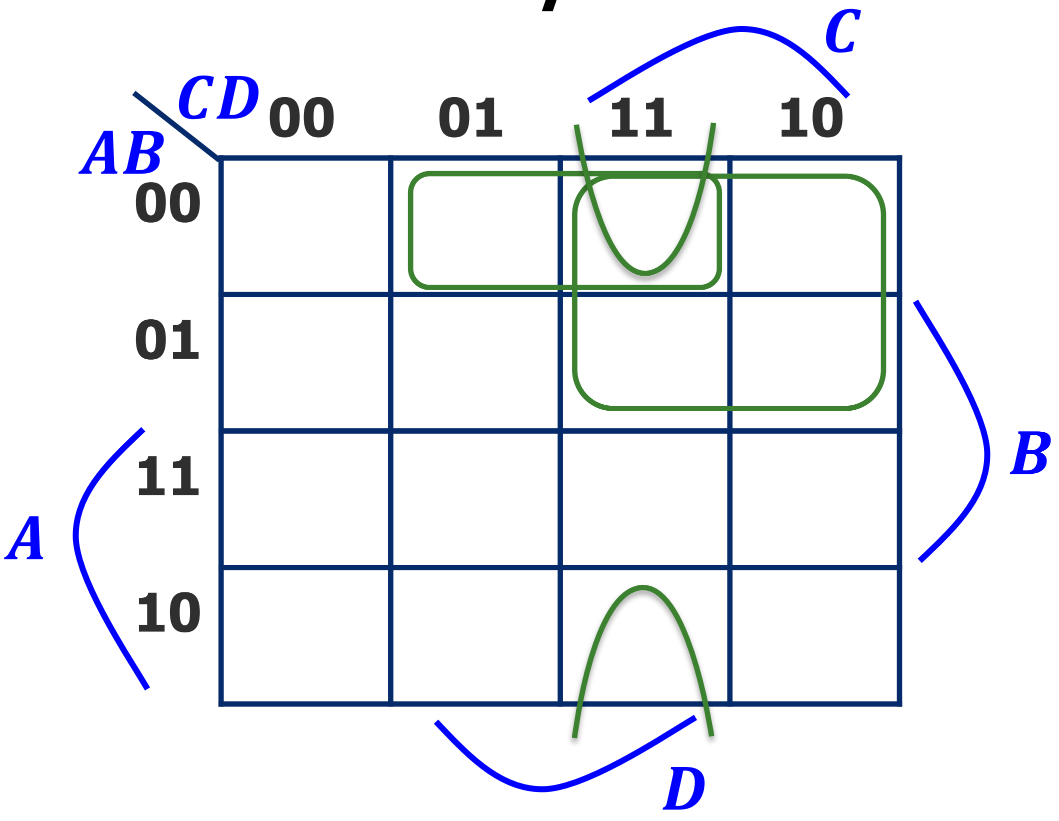


F1 =

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

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

K-map for F2



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		
• • •					

I can pick 00, 01, 10, 11 independently of below

I can pick 00, 01, 10, 11 independently of above

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

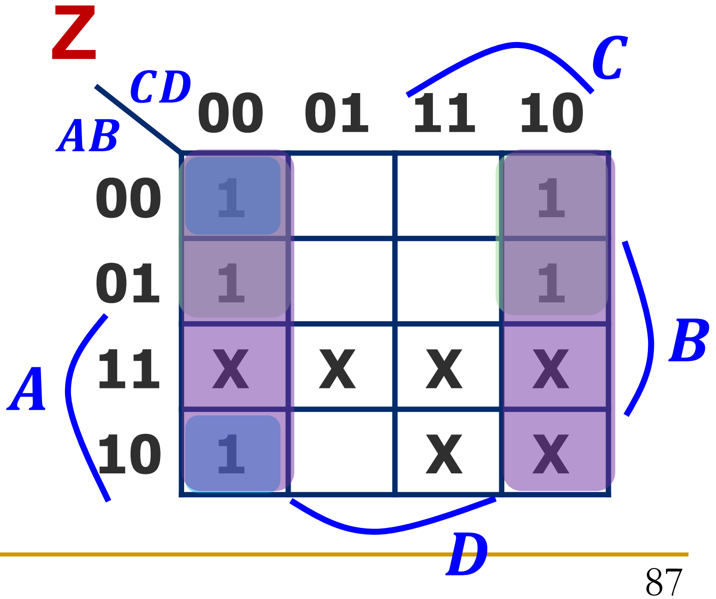
0

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

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