COMBINATIONAL LOGIC DESIGN

Chapter 2

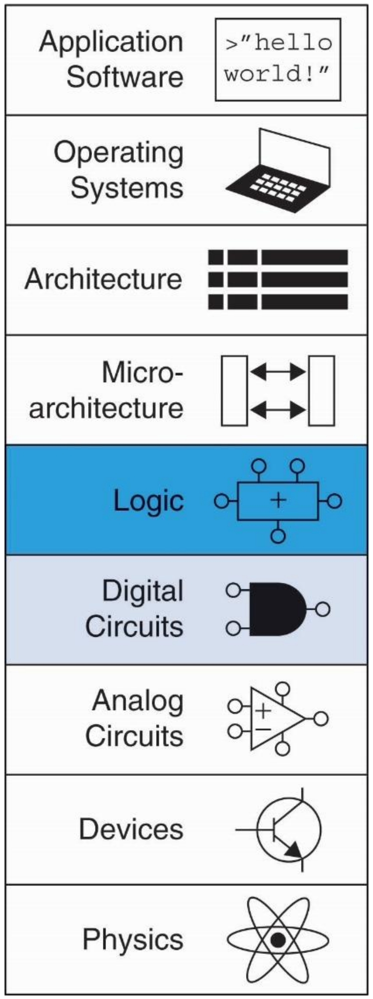
***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

ELSEVIER

Chapter 2 :: Topics

- **Introduction**
- **Boolean Equations**
- **Boolean Algebra**
- **From Logic to Gates**
- **Multilevel Combinational Logic**
- **X's and Z's, Oh My**
- **Karnaugh Maps**
- **Combinational Building Blocks**
- **Timing**

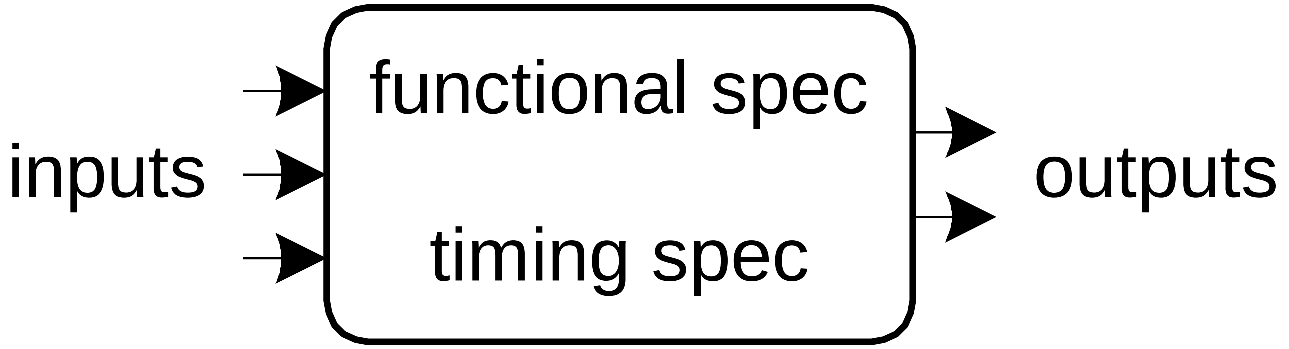


ELSEVIER

Introduction

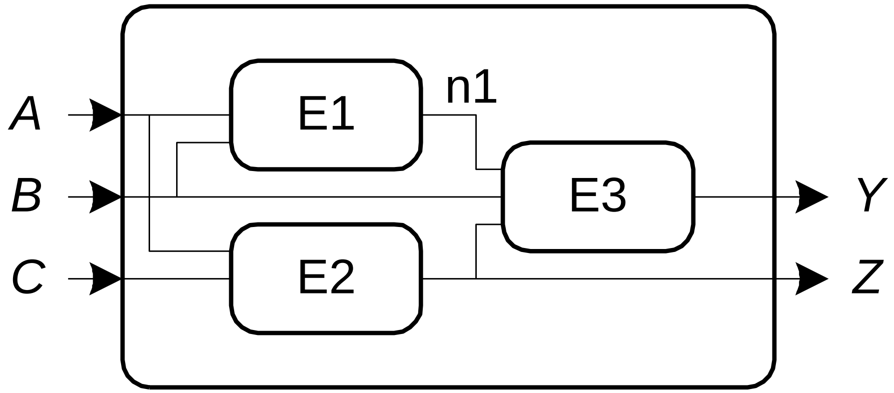
A logic circuit is composed of:

- **Input**s
- **Output**s
- **Function**al specification
- **Timing** specification



Circuits

- Nodes
  - Inputs: *A*, *B*, *C*
  - Outputs: *Y*, *Z*
  - Internal: n1
- Circuit elements
  - E1, E2, E3
  - Each a circuit



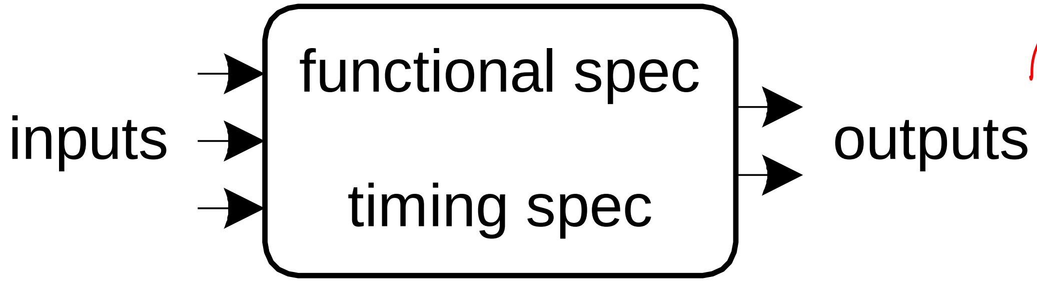
Types of Logic Circuits

- **Combinational Logic**
  - **Memoryless**
  - Outputs determined by current values of inputs

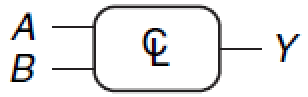
Chap 2

- **Sequential Logic**
  - Has **memory**
  - Outputs determined by previous and current values of inputs

Chap 3



Combinational Logic



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

Figure 2.3 **Combinational logic circuit**

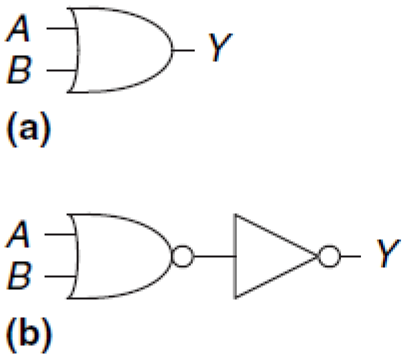
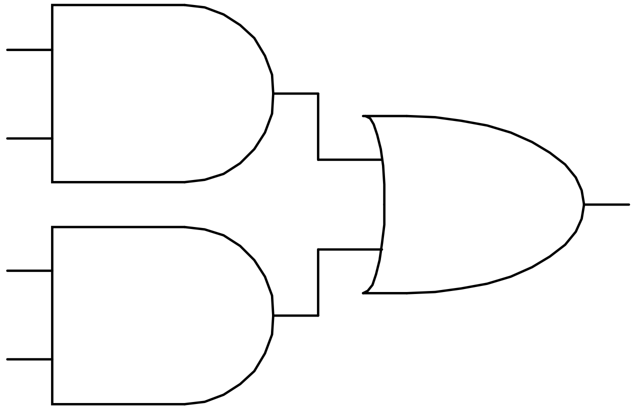


Figure 2.4 **Two OR implementations**

Rules of Combinational Composition

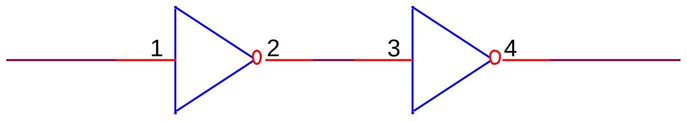
- Every element is combinational
- Every node is either an **input** or connects to ***exactly one* output**
- The circuit contains **no cyclic paths**
- **Example:**



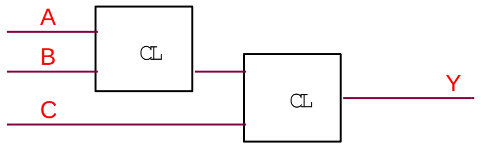
Rules of Combinational Composition

- Circuit is combinational if it consists of interconnected circuit elements such that
  - Every node in the circuit is either an input to the circuit or connected to exactly one output terminal of an element.
  - The circuit contains no cyclic paths
  - Every path in the circuit visits each node at most once
  - Every circuit element itself is combinational

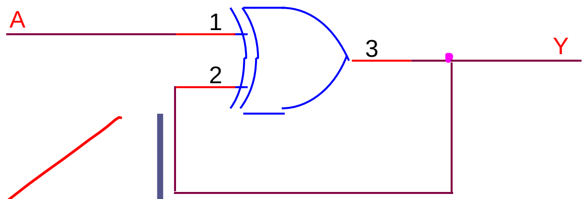
Example



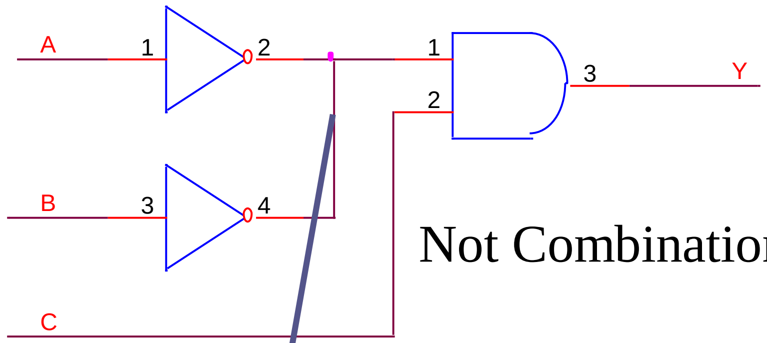
Combinational



Combinational



Not Combinational



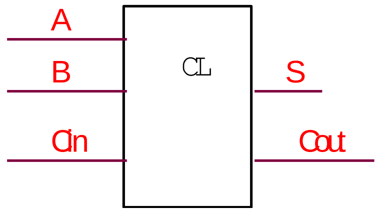
Not Combinational

The circuit contains no cyclic paths

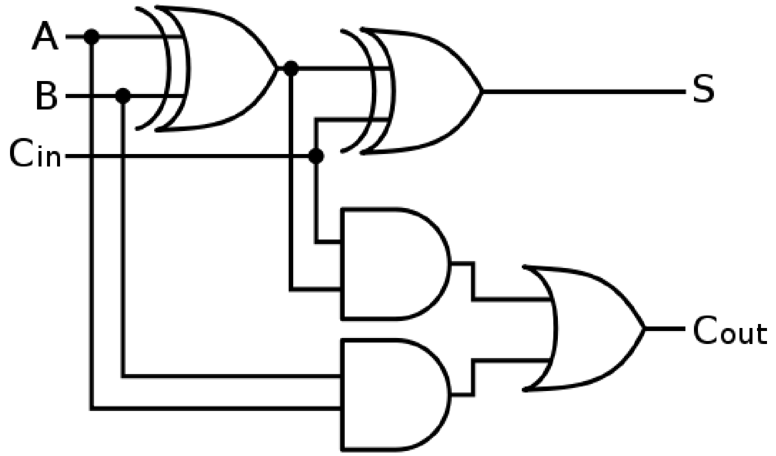
Every path in the circuit visits each node at most once

So flip flops are not combinational (sequential).

Example



Combinational with multiple outputs



Full-adder:

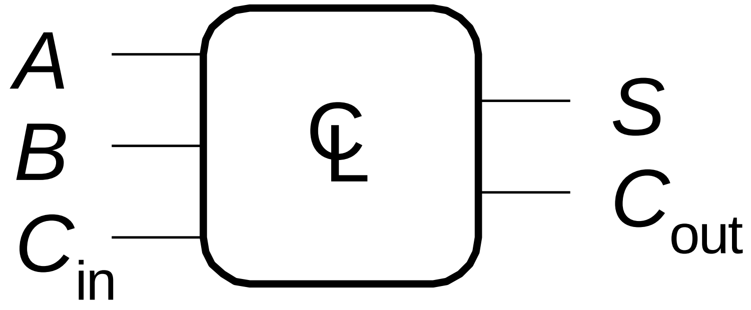
Sum = $A \oplus B \oplus C_{in}$

Cout = $AB + AC_{in} + BC_{in}$

Boolean Equations

- Functional specification of outputs in terms of inputs
- **Example:** $S = F(A, B, C_{in})$

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



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

Some Definitions

- **Complement**: variable with a bar over it

  $\bar{A}, \bar{B}, \bar{C}$
- **Literal**: variable or its complement

  $A, \bar{A}, B, \bar{B}, C, \bar{C}$
- **Implicant**: product of 1 or more literals

  $AB\bar{C}, \bar{A}C, BC, B$
- **Minterm**: product that includes all input variables

  $AB\bar{C}, A\bar{B}\bar{C}, ABC$
- **Maxterm**: sum that includes all input variables

  $(A+\bar{B}+C), (\bar{A}+B+\bar{C}), (\bar{A}+\bar{B}+C)$

Boolean Equations

Minterm: is a **Product**

Maxterm: is a **Sum**

$\Big\}$ involving **all** of the **inputs**

Minterm is needed for sum of products form

For a function of three variables (A, B, C)

- $\boldsymbol{A\overline{B}\,\overline{C}}$ is a minterm
- $\boldsymbol{A\overline{B}}$ is not, does not involve C
- $\boldsymbol{A + \overline{B} + \overline{C}}$ is a maxterm
- $\boldsymbol{A + \overline{B}}$ is not, does not involve C

Any boolean expression can be expressed as Sum Of Products or Product Of Sums

Order of logical operations: NOT, AND, OR

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
- Each row has a **minterm**
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)
- Form function by ORing minterms where the output is TRUE
- Thus, a sum (OR) of products (AND terms)

A	*B*	*Y*	**minterm**	**minterm name**
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	1	$A\ B$	m_3

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

ELSEVIER

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
- Each row has a **minterm**
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)
- Form function by ORing minterms where the output is TRUE
- Thus, a sum (OR) of products (AND terms)

A	*B*	*Y*	**minterm**	**minterm name**
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	1	$A\ B$	m_3

Y = F(A, B) =

ELSEVIER

Sum-of-Products (SOP) Form

- All equations can be written in SOP form
- Each row has a **minterm**
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)
- Form function by ORing minterms where the output is TRUE
- Thus, a sum (OR) of products (AND terms)

A	**B**	**Y**	**minterm**	**minterm name**
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	1	$A\ B$	m_3

$\Sigma(1,3)$

$$Y = F(A, B) = \overline{A}B + AB = \Sigma(1, 3)$$

Product-of-Sums (POS) Form

- All Boolean equations can be written in POS form
- Each row has a **maxterm**
- A maxterm is a sum (OR) of literals
- Each maxterm is FALSE for that row (and only that row)
- Form function by ANDing the maxterms for which the output is FALSE
- Thus, a product (AND) of sums (OR terms)

A	*B*	*Y*	**maxterm**	**maxterm name**
0	0	0	$A + B$	M_0
0	1	1	$A + \overline{B}$	M_1
1	0	0	$\overline{A} + B$	M_2
1	1	1	$\overline{A} + \overline{B}$	M_3

$$Y = F(A, B) = (A + B)(A + \overline{B}) = \Pi(0, 2)$$

Example

Example

A museum has three rooms, each with a motion sensor (m0, m1, and m2) that outputs 1 when motion is detected. At night, the only person in the museum is one security guard who walks from room to room. Create a circuit that sounds an alarm (by setting an output A to 1) if motion is ever detected in more than one room at a time (i.e., in two or three rooms), meaning there must be one or more intruders in the museum. Start with a truth table.

(a) Show the alarm expression in SOP form
(b) Show the alarm expression in POS form

A	B	C	[illegible]
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

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

$(A+B+C)(A+B+\bar{C})(A+\bar{B}+C)$
$(\bar{A}+B+C)$

ELSEVIER

Example

m1,m2,m3 are the motion detectors in the rooms. If the output A is 1 alarm will sound.

Inputs			Outputs
m2	m1	m0	A
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Minterms are :
m2'm1m0, m2m1'm0, m2m1m0',m2m1m0

A= Σ (m3, m5, m6, m7)

SOP form
A = m2'm1m0 + m2m1'm0 + m2m1m0' + m2m1m0

ELSEVIER

Example

POS form

Maxterms are
(m2+m1+m0), (m2+m1+m0’), (m2+m1’+m0), (m2’+m1+m0)

Product sums form
A= (m2+m1+m0)(m2+m1+m0’)(m2+m1’+m0)(m2’+m1+m0)

A= Π(M0, M1, M2, M4)

Review_5

- Power Consumption (Static, Dynamic Power Consumption)
- Types of Logic Circuits (Combinational, Sequential)
- Rules of Combinational Composition
- Boolean Equations-Definitions (Complement, Literal, Implicant, Minterm, Maxterm)
- Sum-of-Products (SOP) Form, Product-of-Sums (POS) Form

1st Midterm

14/11/2018

17:30

Review$_5$

- minterms-MAXters

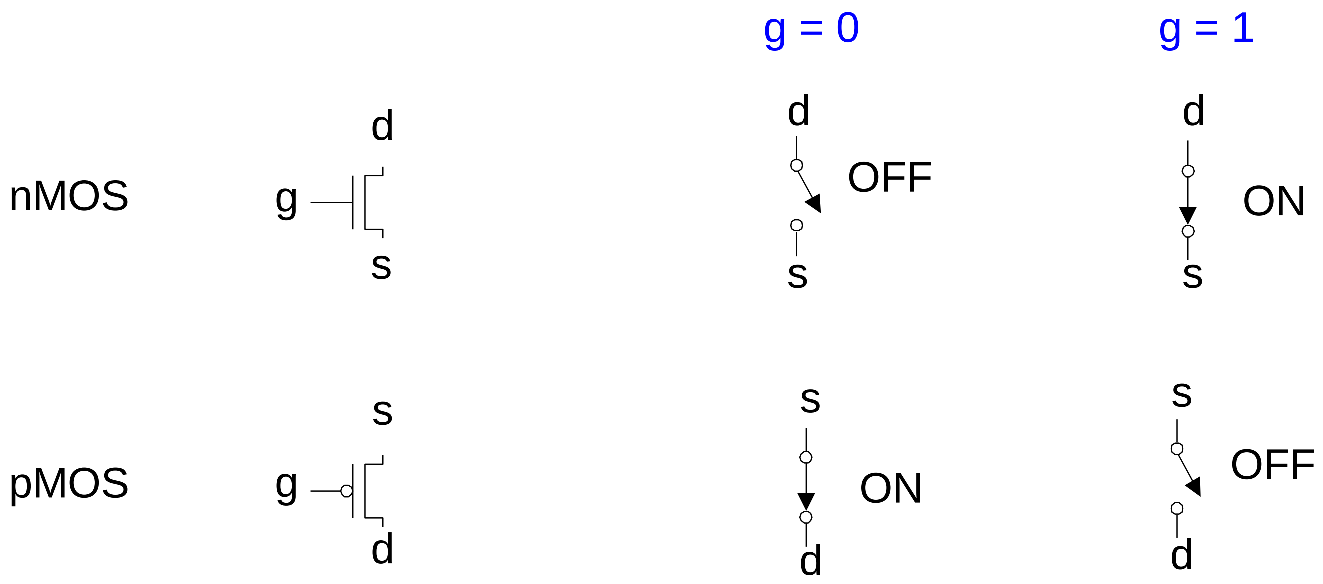
A B C	Y	Y
0 0 0	A'B'C'	m_0
0 0 1	A'B'C	m_1
0 1 0	A'B C'	m_2
0 1 1	A'B C	m_3
1 0 0	A B'C'	m_4
1 0 1	A B'C	m_5
1 1 0	A B C'	m_6
1 1 1	A B C	m_7

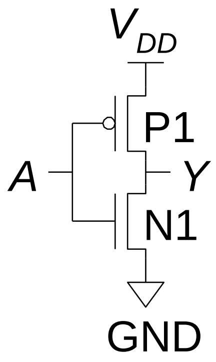
A B C	Y	Y
0 0 0	A+B+C	M_0
0 0 1	A+B+C'	M_1
0 1 0	A+B'+C	M_2
0 1 1	A+B'+C'	M_3
1 0 0	A'+B+C	M_4
1 0 1	A'+B+C'	M_5
1 1 0	A'+B'+C	M_6
1 1 1	A'+B'+C'	M_7

Review$_5$

- **Minterm to maxterm conversion** use maxterms whose indices do not appear in minterm expansion
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7) = \Pi M(0,2,4)$
- **Maxterm to minterm conversion** use minterms whose indices do not appear in maxterm expansion
  - e.g., $F(A,B,C) = \Pi M(0,2,4) = \Sigma m(1,3,5,6,7)$
- Minterm expansion of F to minterm expansion of F' use minterms whose indices do not appear
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7)$ $F'(A,B,C) = \Sigma m(0,2,4)$
- Maxterm expansion of F to maxterm expansion of F' use maxterms whose indices do not appear
  - e.g., $F(A,B,C) = \Pi M(0,2,4)$ $F'(A,B,C) = \Pi M(1,3,5,6,7)$

Review$_5$- Transistor Function





Exercise

Given the truth table for function F, find the epression;

- As a sum of minterms,
- As product of maxterms

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

$$\sum(2,4,5,6)$$

$$\prod(0,1,3,7)$$

Note: Use SOP if the output is TRUE on only a few rows of the truth table

Else

Use POS

Exercise

Write a Boolean equation for the following truth table in SOP form

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

Boolean Algebra

- Boolean algebra is used to **simplify** boolean equations
- Obtain the **simplest** set of logic gates
  - Reduce **cost** of implementation
  - Reduce **power** consumption
- Based on a set of **Axioms** (can be used to prove all the Boolean algebra theorems)
- Axioms and theorems obey the principle of **duality**.
  - If the symbols 0 and 1 and the operators AND (.) and OR (+) are **interchanged,** the boolean statement will still be correct
  - Use (‘) symbol to denote the dual of a statement

Boolean Algebra

- An equation is said to be minimized if it uses **fewest** possible **implicants** with **fewest literals**.
- **Implicant**: product of 1 or more literals
  - ABC, AC, BC, B
- An implicant is called a **prime implicant** if it can not be combined with any other implicants in the equation in order to obtain a more simple equation.
- **Simplifying** reduces the number of gates to implement the function, thus making it **smaller**, **cheaper** and **fast**

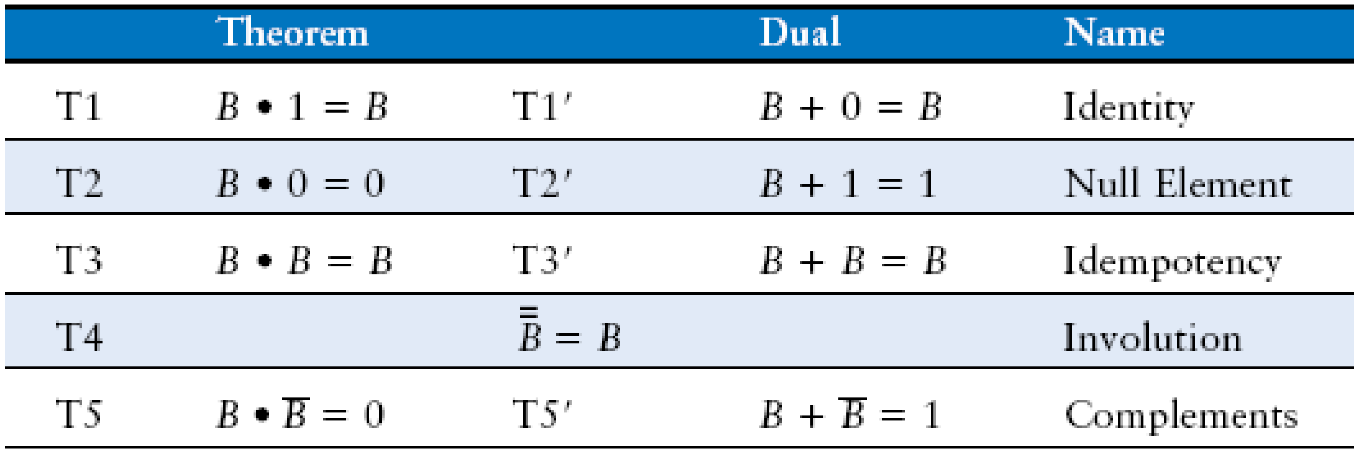
Boolean Algebra

- Axioms and theorems to **simplify** Boolean equations
- Like regular algebra, but simpler: variables have only two values (1 or 0)
- **Duality** in axioms and theorems:
  - ANDs and ORs, 0's and 1's interchanged

Boolean Axioms

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Theorems of One Variable



	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

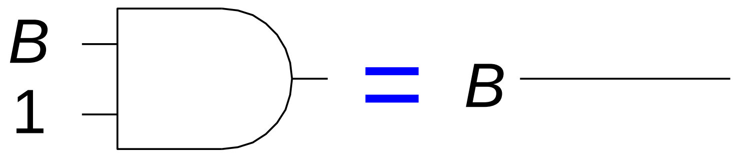
- Theorems of one variable can be used to simplify equations involving one variable

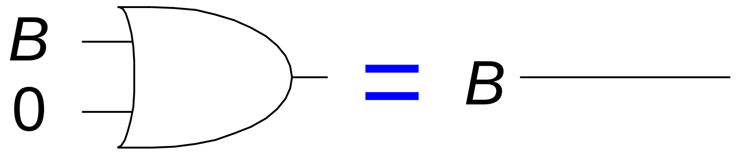
T1: Identity Theorem

- $B \cdot 1 = B$
- $B + 0 = B$

T1: Identity Theorem

- $B \bullet 1 = B$
- $B + 0 = B$



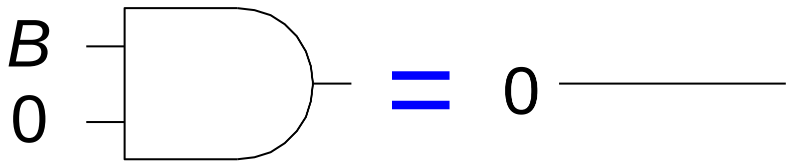


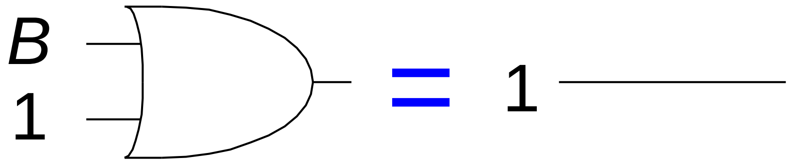
T2: Null Element Theorem

- $B \bullet 0 = 0$
- $B + 1 = 1$

T2: Null Element Theorem

- $B \cdot 0 = 0$
- $B + 1 = 1$



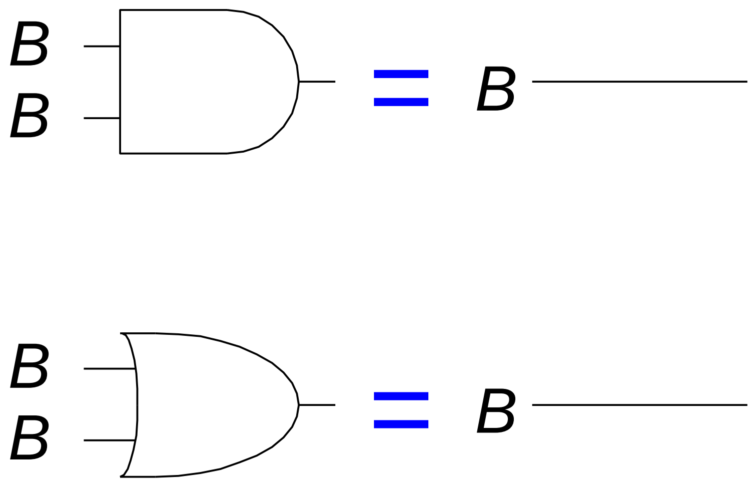


T3: Idempotency Theorem

- $B \cdot B = B$
- $B + B = B$

T3: Idempotency Theorem

- $B \bullet B = B$
- $B + B = B$



COMBINATIONAL LOGIC DESIGN

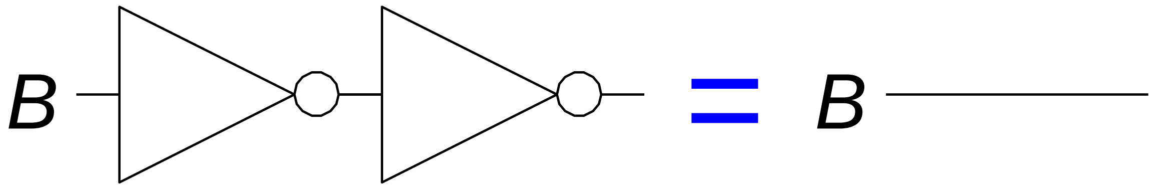
T4: Involution Theorem

- $\overline{\overline{B}} = B$

ELSEVIER

T4: Involution Theorem

- $\overline{\overline{B}} = B$

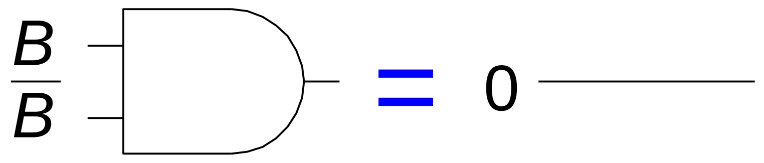


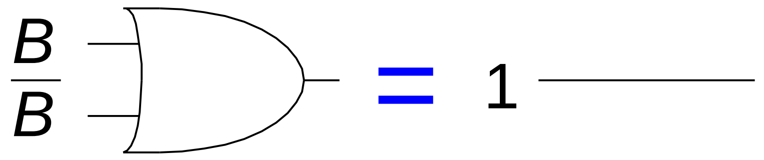
T5: Complement Theorem

- $B \bullet \overline{B} = 0$
- $B + \overline{B} = 1$

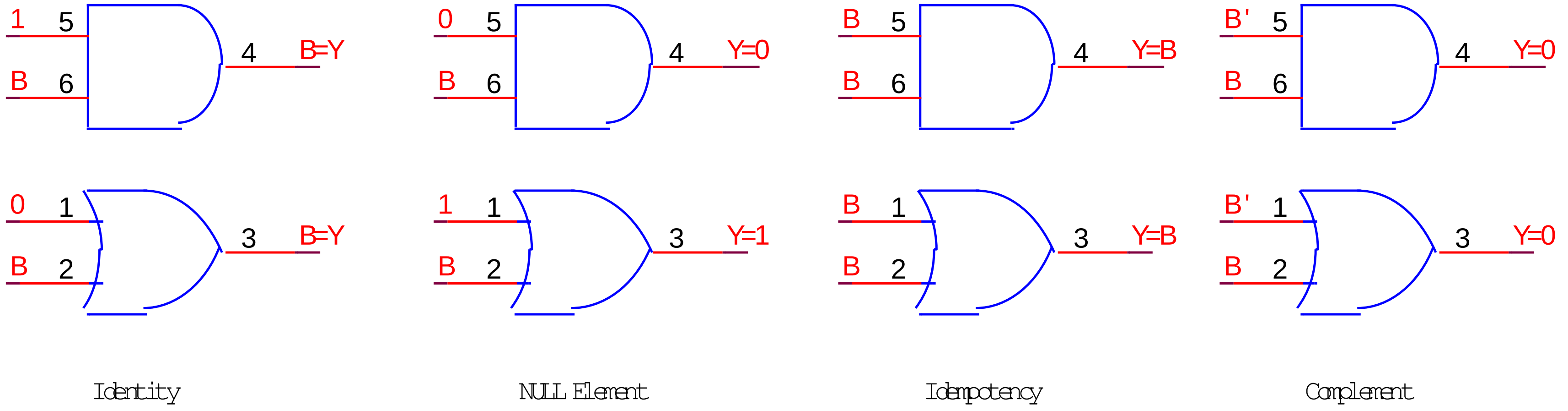
T5: Complement Theorem

- $B \bullet \overline{B} = 0$
- $B + \overline{B} = 1$





Theorems (Summary)



Boolean Theorems Summary

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\bar{\bar{B}} = B$		Involution
T5	$B \bullet \bar{B} = 0$	T5′	$B + \bar{B} = 1$	Complements

Boolean Theorems of Several Vars

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + B \bullet D = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \overline{C}) = B$	T10′	$(B + C) \bullet (B + \overline{C}) = B$	Combining
T11	$(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D)$ $= B \bullet C + \overline{B} \bullet D$	T11′	$(B + C) \bullet (\overline{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2})$	De Morgan's Theorem

- Theorems of several variables can be used to simplify equations involving more than one variable
- T9 to T11 allows us to eliminate redundant variables

Simplifying Boolean Equations

Example 1:

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

Simplifying Boolean Equations

Example 1:

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

$= B(A + \overline{A})$ T8

$= B(1)$ T5'

$= B$ T1

- 8: distributivity
- 5: complements
- 1: Identitiy

Simplifying Boolean Equations

Example 2:

- $Y = A(AB + ABC)$

$A(AB+ABC)$

$A(AB(1+C))$

$A(AB)$

AB

Simplifying Boolean Equations

Example 2:

- $Y = A(AB + ABC)$

$= A(AB(1 + C))$	T8
$= A(AB(1))$	T2’
$= A(AB)$	T1
$= (AA)B$	T7
$= AB$	T3

- 8: distributivity
- 2: null element
- 1: Identitiy
- 7: Associativity
- 3: Idempotency

Exercise

Minimize the following expression:

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

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

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

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

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

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

□ $\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}\bar{C} + A\bar{B}C$	T3: Idempotency B+B=B
□ $(A + \bar{A})\,\bar{B}\bar{C} + A\bar{B}(C+\bar{C})$	T8: Distribution
□ $(1)\,\bar{B}\bar{C} + A\bar{B}(1)$	T5: Complements $(C+\bar{C})$=1
□ $\bar{B}(\bar{C} + A)$	T1: Identity $(C.1)$=C

DeMorgan’s Theorem

- Theorem is a powerful tool in digital design
- The **complement of a product** is equal to the **sum of complement** of each term.
- Likewise, the **complement of a sum** is equal to the **product of complement** of each term.

Review_6

- Boolean Algebra (simplify boolean equations)
- Axioms
- Boolean algebra theorems
- Simplifying Boolean Equations
- DeMorgan's Theorem

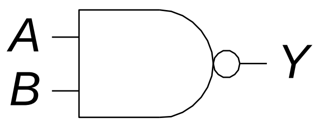
1st Midterm

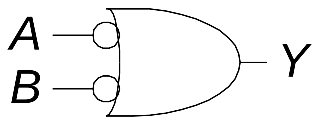
14/11/2018

17:30

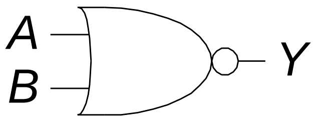
DeMorgan's Theorem

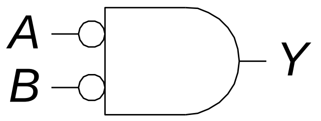
- $Y = \overline{AB} = \overline{A} + \overline{B}$





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





Bubble Pushing

- CMOS circuits prefer **NANDs** and **NORs** over ANDs and Ors. But the function of a multilevel circuit with NANDs and NORs may not be **clear by inspection**.
- Bubble pushing is a **helpful way** to **redraw** these type of circuits so that its function can be more easily determined.

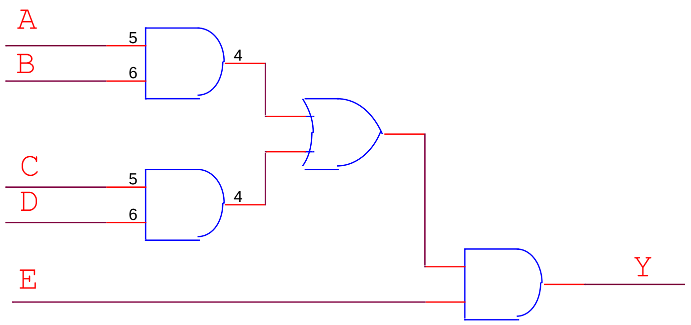
Bubble Pushing

- Inversion circle is called a bubble
- Backward bubble pushing (output->input)
- Forward bubble pushing (input->output)
- Pushing a bubble from output back to input – put bubbles on all gate inputs
  - Push any bubble at the output back
  - If a gate has an input bubble, draw the preceeding gate with an output bubble so that bubbles cancel. If the current gate doesn’t have an input bubble, draw the preceeding gate without an output bubble.
- Pushing a bubble from input to output – put bubble on the output

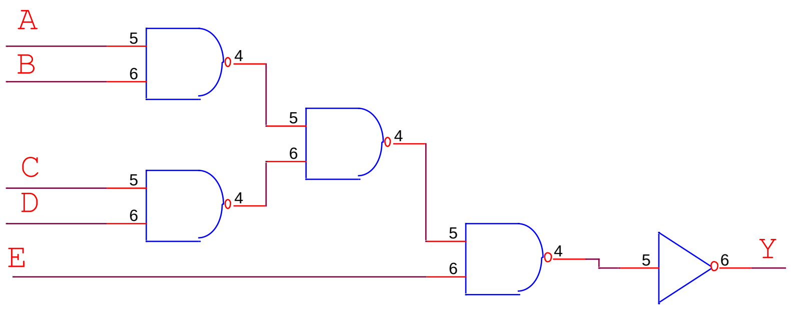
Bubble Pushing

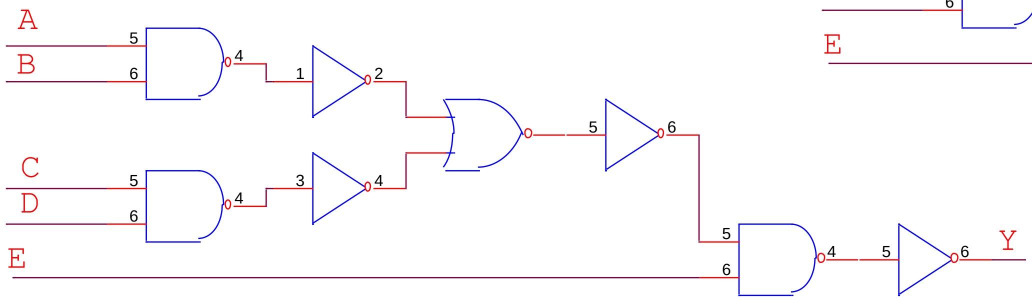
- Inversion circle is called a bubble
- Backward bubble pushing (output->input)
- Forward bubble pushing (input->output)
- Pushing a bubble from output (backward) or from the inputs (forward) changes the body of the gate from **AND to OR** or vice-versa.
- Pushing a bubble from output back to input – **put bubbles on all gate inputs**
- Pushing a bubble from input to output – **put bubble on the output**

Bubble Pushing Example



$$Y = (AB + CD).E$$

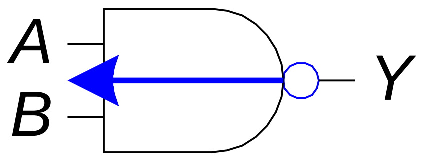


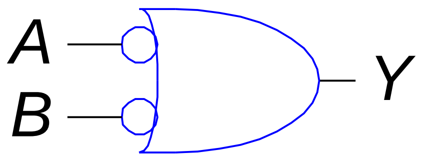


$$Y = \left(\overline{\overline{AB}.\overline{CD}}\right).E$$

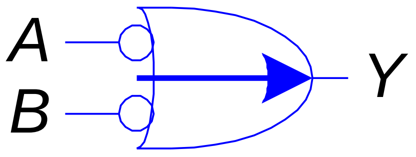
Bubble Pushing

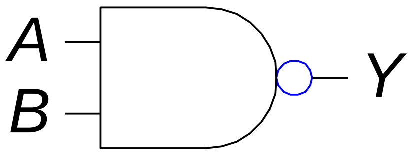
- **Backward:**
  - Body changes
  - Adds bubbles to inputs





- **Forward:**
  - Body changes
  - Adds bubble to output

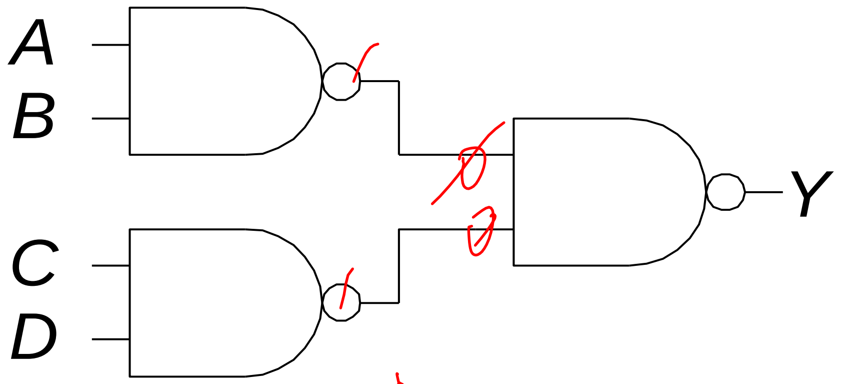


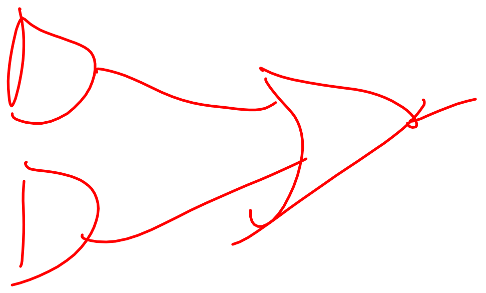


ELSEVIER

Bubble Pushing

- What is the Boolean expression for this circuit?

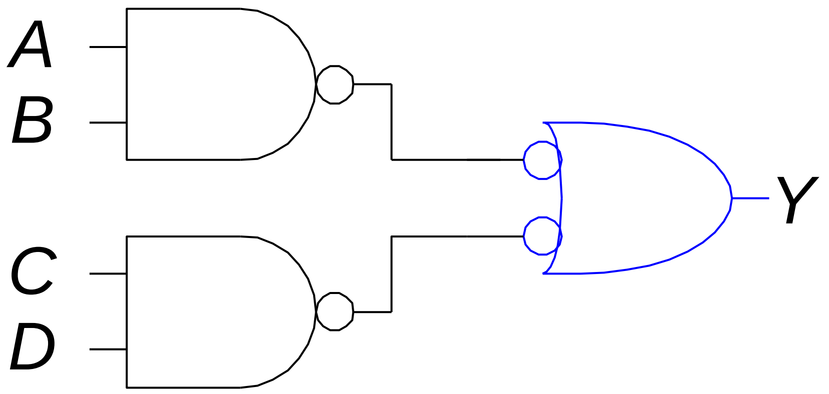




AB + CD

Bubble Pushing

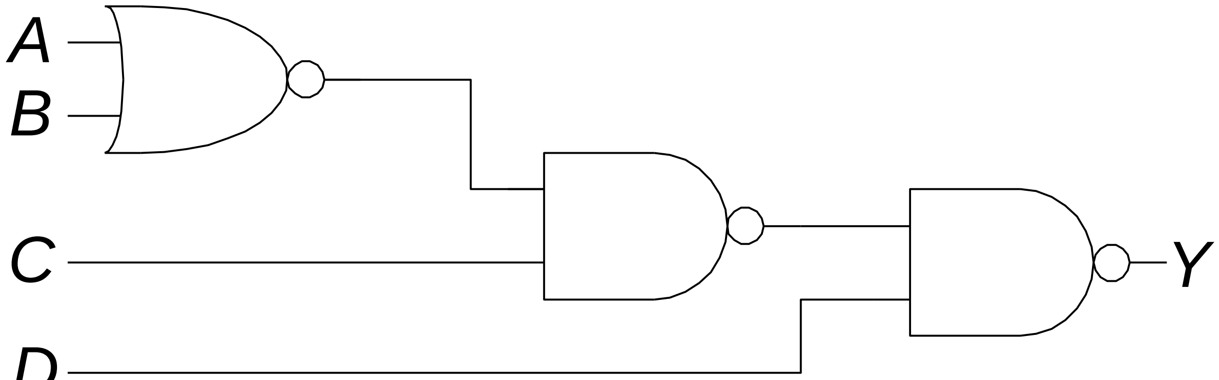
- What is the Boolean expression for this circuit?



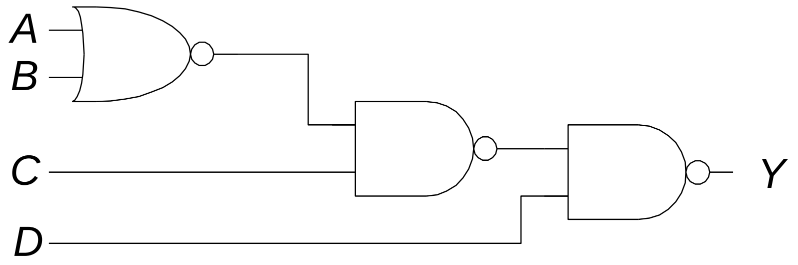
$$Y = AB + CD$$

Bubble Pushing Rules

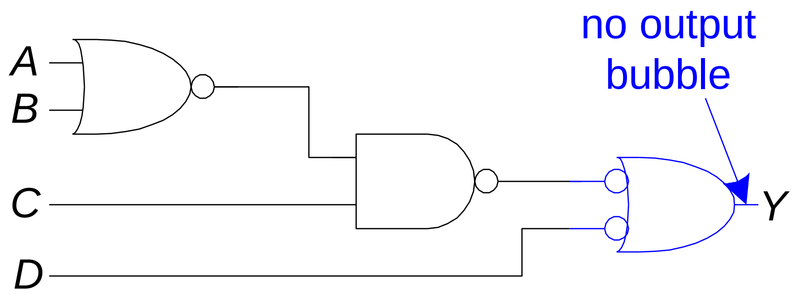
- Begin at output, then work toward inputs
- Push bubbles on final output back
- Draw gates in a form so bubbles cancel



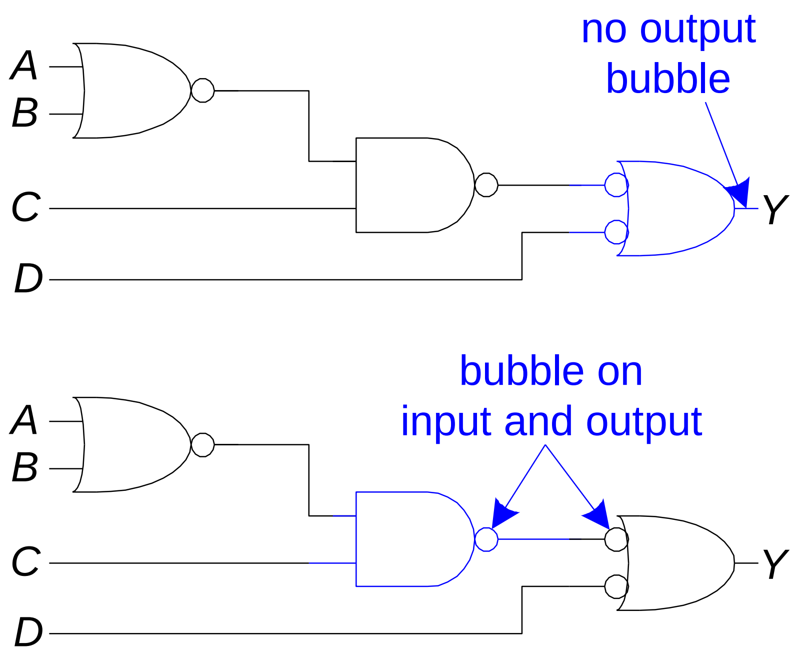
Bubble Pushing Example



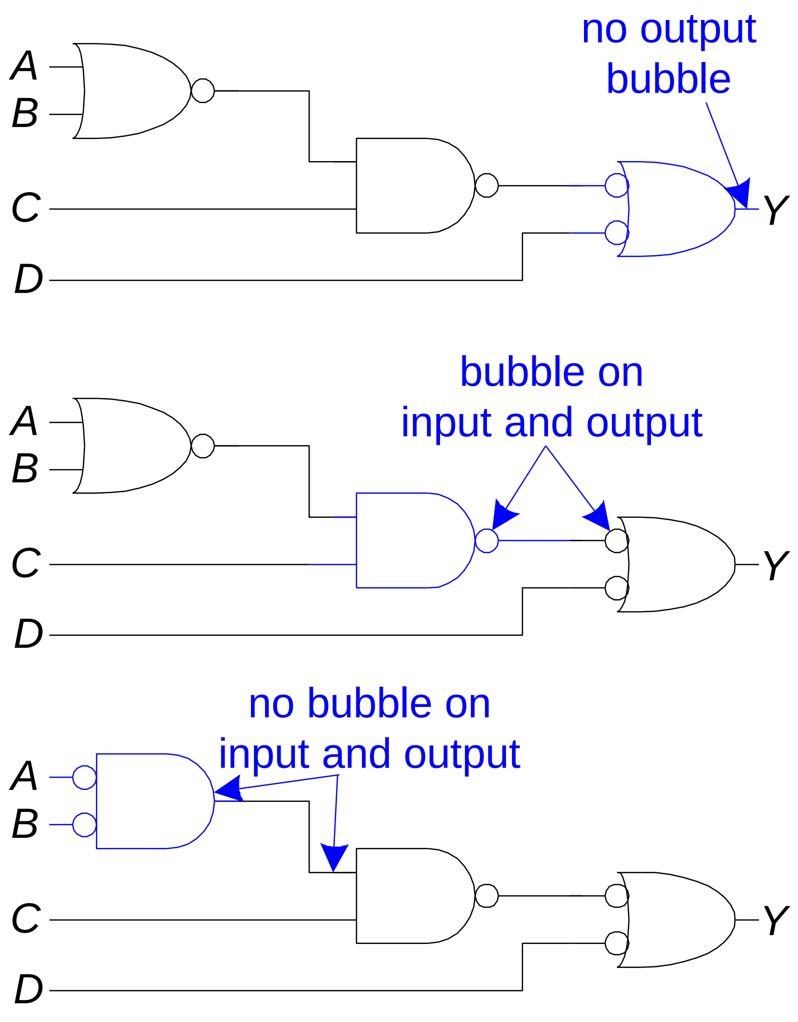
Bubble Pushing Example



Bubble Pushing Example

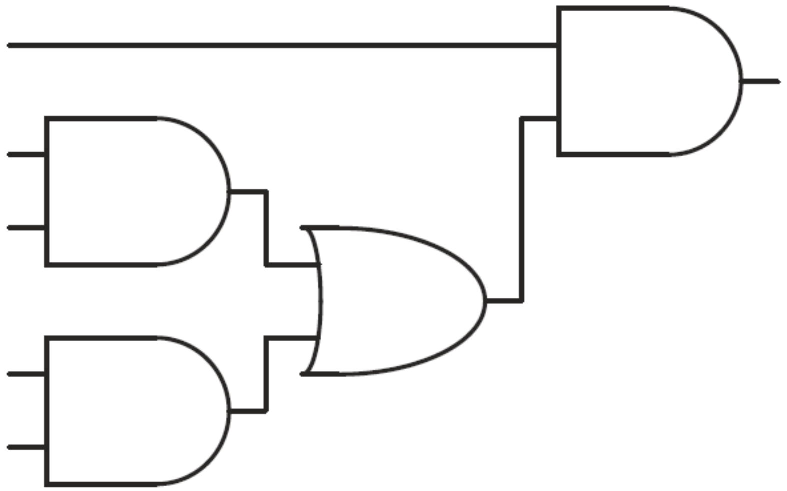


Bubble Pushing Example



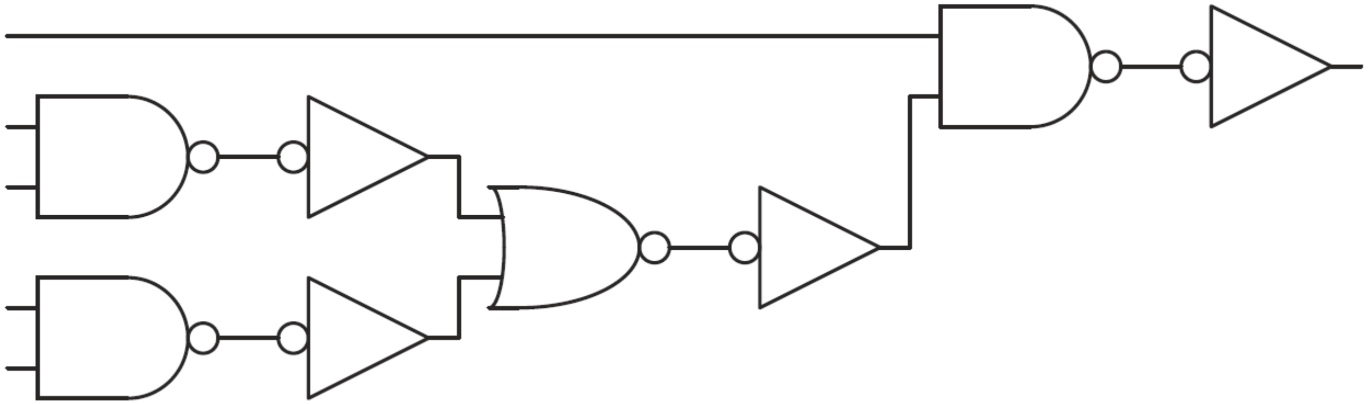
$$Y = \overline{A}\,\overline{B}C + \overline{D}$$

Bubble Pushing for CMOS



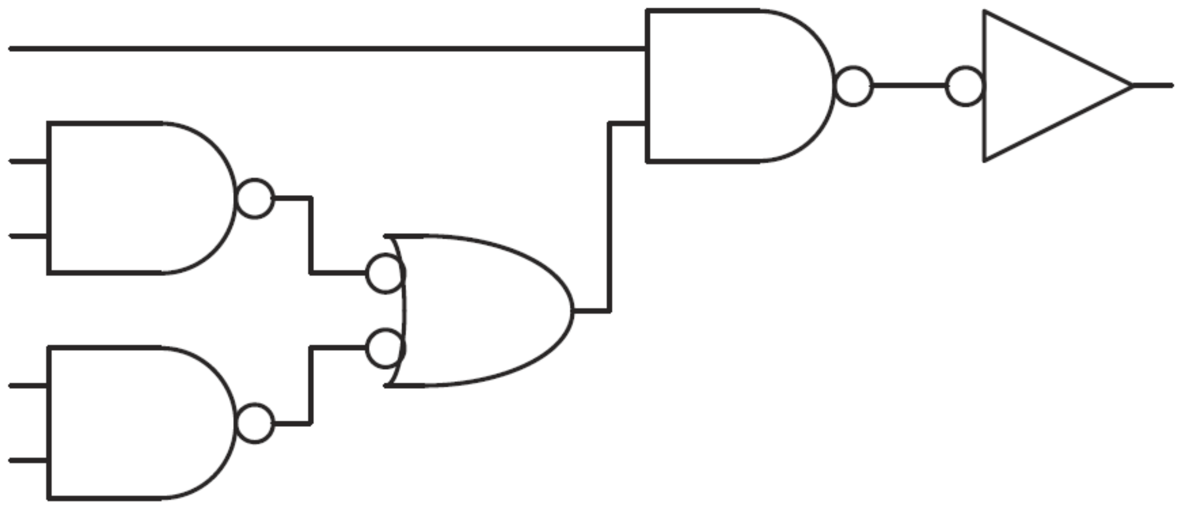
- CMOS logic favors NAND/NOR over AND/OR
- Convert the circuit above such that only NAND/NOR/INVs are used.

Bubble Pushing for CMOS

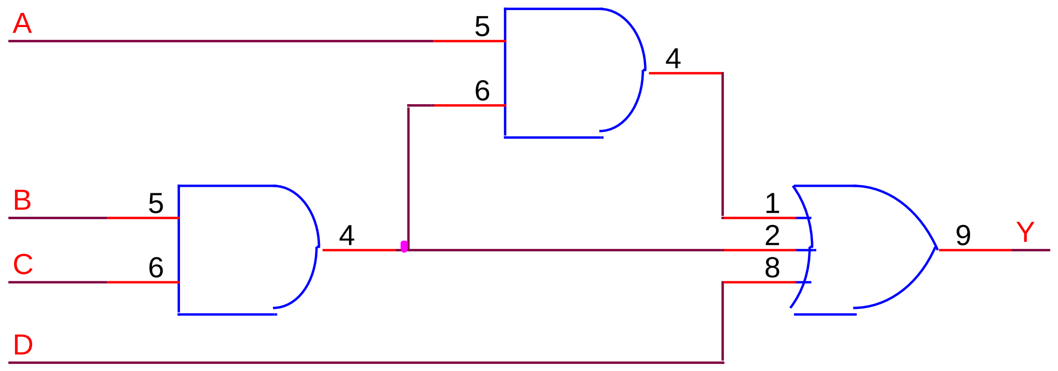


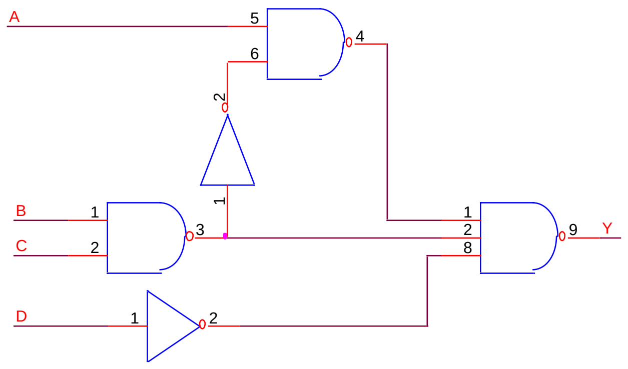
Inefficient!

Bubble Pushing for CMOS



Bubble Pushing for CMOS Example





Functional Completeness: True or False?

We can implement any logic function using:

1. Only AND, OR, INVs
2. Only AND, INVs
3. Only NANDs
4. Only NORs
5. Only ANDs

Functional Completeness: True or False?

We can implement any logic function using:

1. Only AND, OR, INVs $\boxed{\textbf{TRUE}}$
2. Only AND, INVs $\boxed{\textbf{TRUE}}$
3. Only NANDs $\boxed{\textbf{TRUE}}$
4. Only NORs $\boxed{\textbf{TRUE}}$
5. Only ANDs $\boxed{\textbf{FALSE}}$
6. Only OR, INVs $\boxed{\textbf{TRUE}}$

- NAND implementation: SUM of products form
- NOR implementation: PRODUCT of sums form

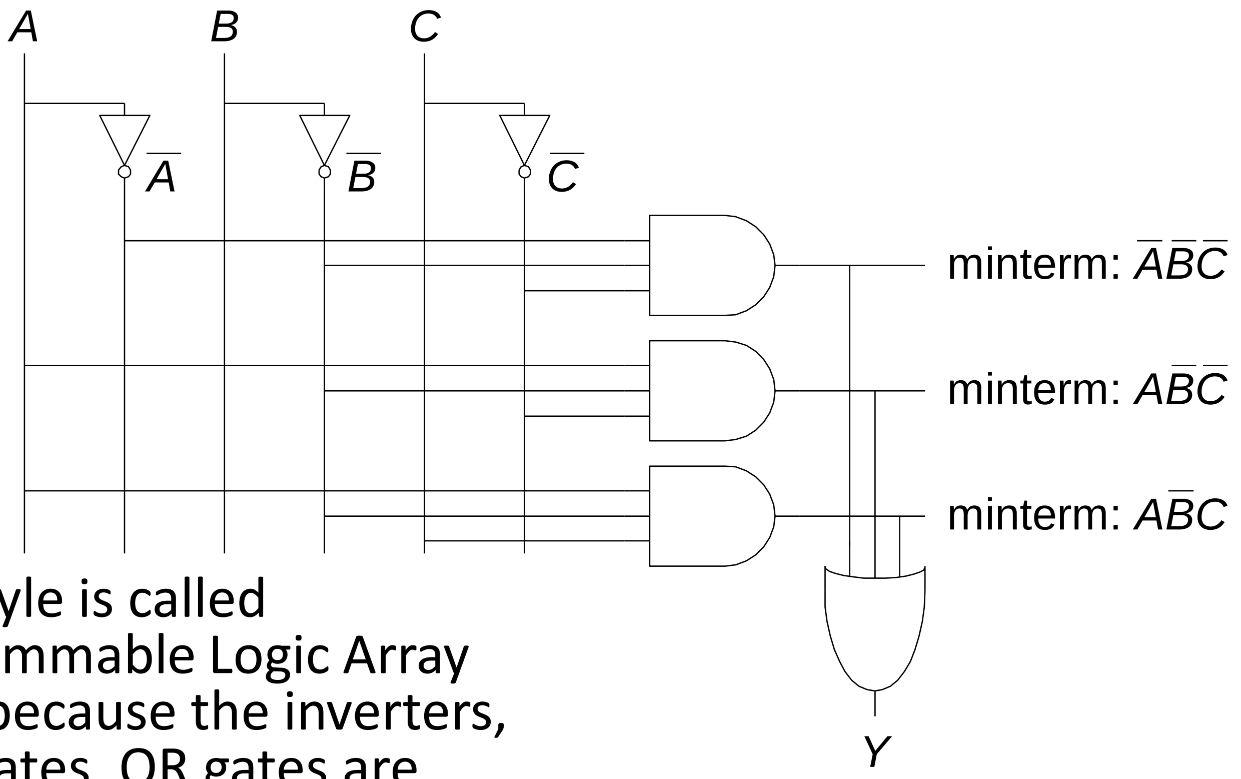
Not using Ands only?

From Logic to Gates

- A schematic is a diagram of a digital circuit showing the elements and the wires that connect them together.
- By drawing schematics in a consistent fashion, we make them easier to read and debug
- Inverters, AND gates, OR gates are arrayed in a systematic way->**P**rogrammable **L**ogic **A**rray
- Prefer NANDs and NORs over ANDs and Ors (in CMOS implementations)
- NAND implementation: SUM of products form
- NOR implementation: PRODUCT of sums form

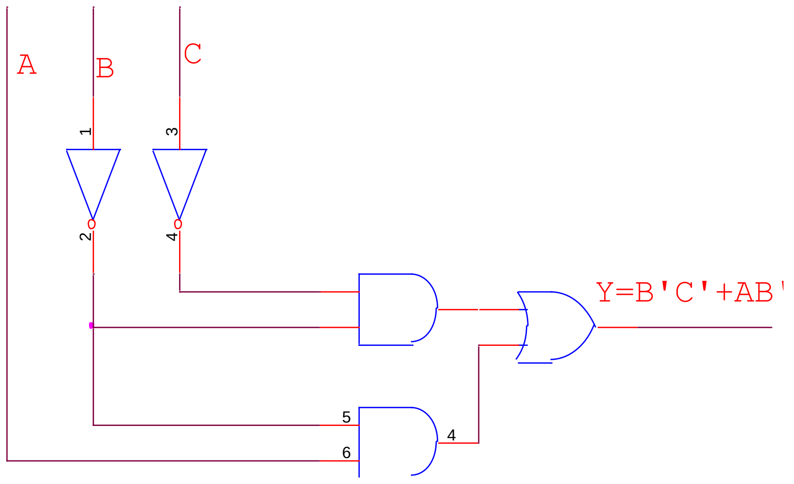
From Logic to Gates

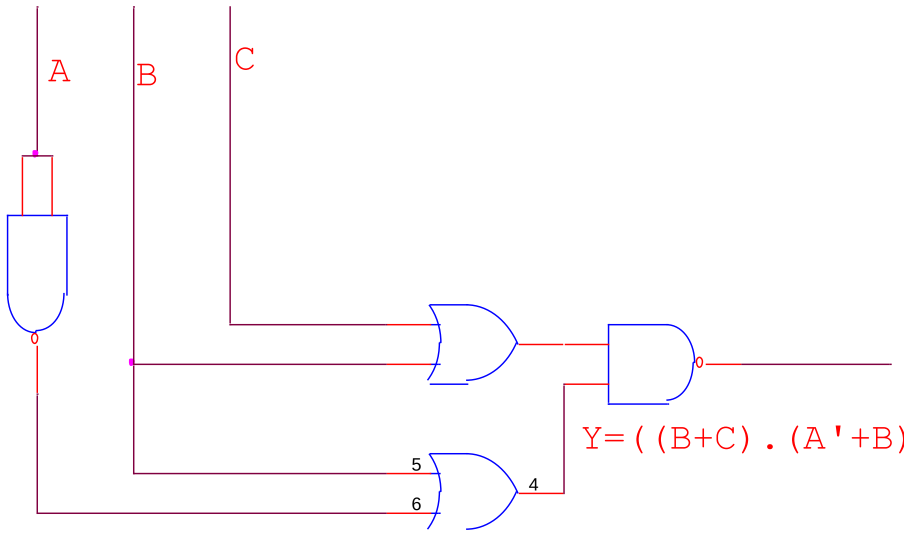
- Two-level logic: ANDs followed by ORs
- Example: $Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$


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

- This style is called Programmable Logic Array (PLA) because the inverters, AND gates, OR gates are arrayed in a systematic way.

From Logic to Gates





- NANDs and NORs are **preferred** over ANDs and Ors in CMOS implementations.

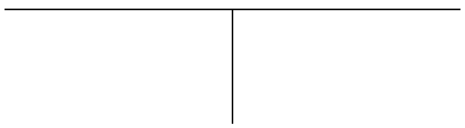
Circuit Schematics Rules

- Inputs on the left (or top)
- Outputs on right (or bottom)
- Gates flow from left to right
- Straight wires are best

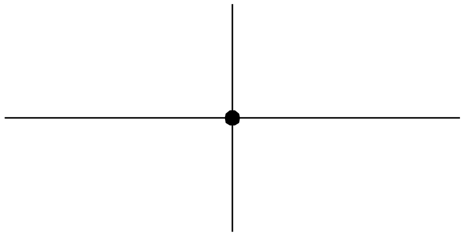
Circuit Schematic Rules (cont.)

- Wires always connect at a T junction
- A dot where wires cross indicates a connection between the wires
- Wires crossing *without* a dot make no connection

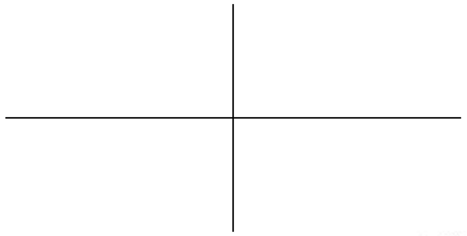
wires connect
at a T junction



wires connect
at a dot



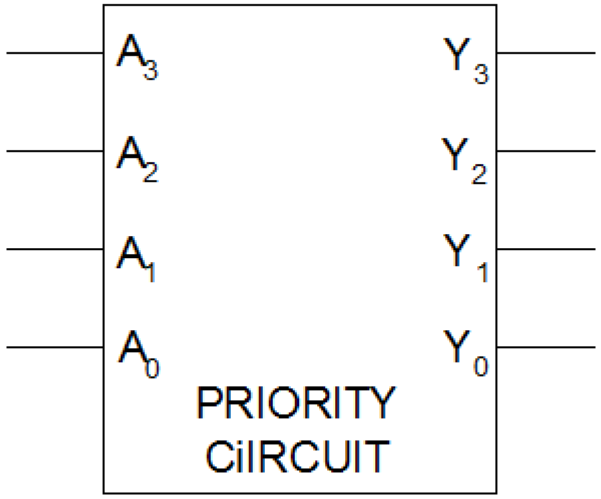
wires crossing
without a dot do
not connect



Multiple-Output Circuits

- **Example: Priority Circuit**

  Output asserted corresponding to most significant TRUE input



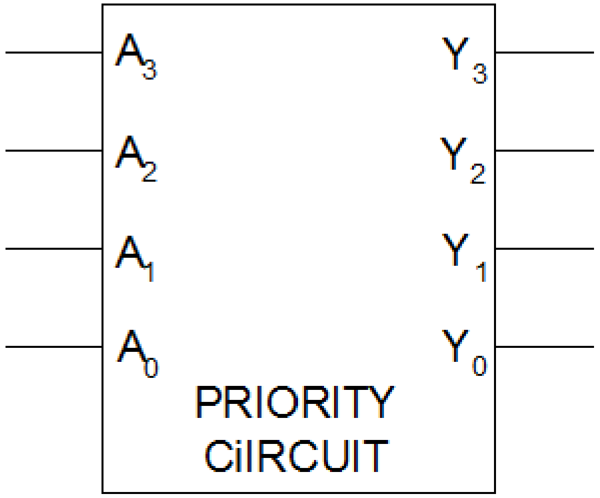
A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0				
0	0	0	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				
0	1	1	0				
0	1	1	1				
1	0	0	0				
1	0	0	1				
1	0	1	0				
1	0	1	1				
1	1	0	0				
1	1	0	1				
1	1	1	0				
1	1	1	1				

Multiple-Output Circuits

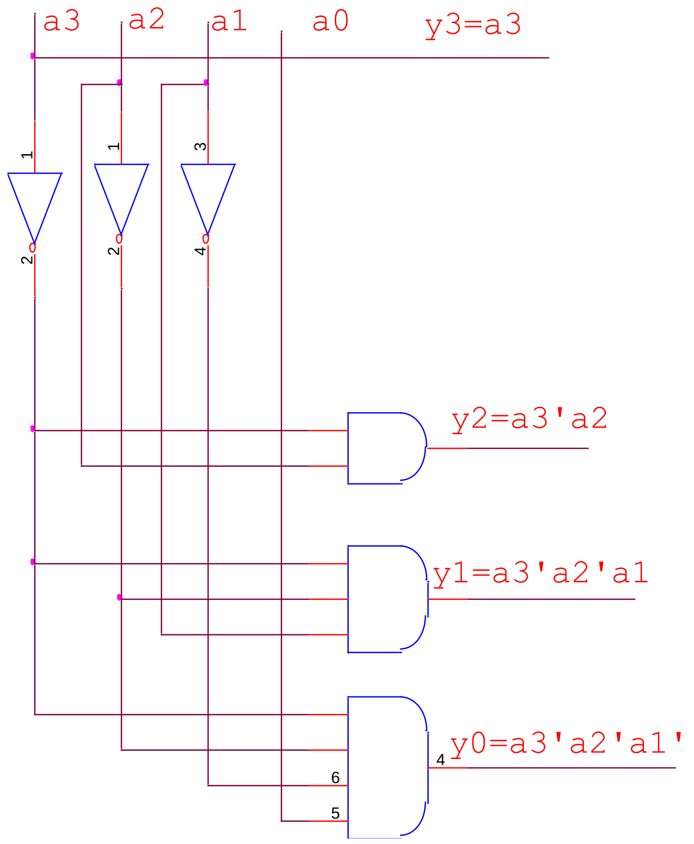
- **Example: Priority Circuit**

  Output asserted corresponding to most significant TRUE input

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



Multiple-Output Circuits

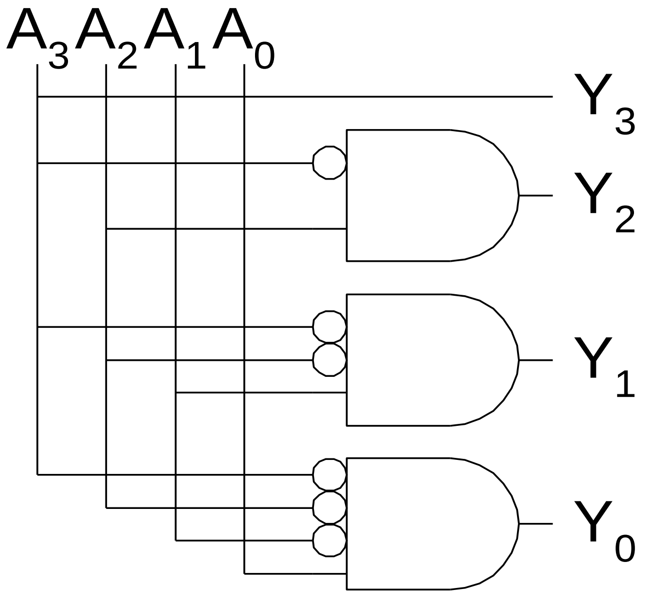


- Priority circuit schematic
- Priority circuit trur-th-table with dont cares

a3	a2	a1	a0	y3	y2	y1	y0
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

Priority Circuit Hardware

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



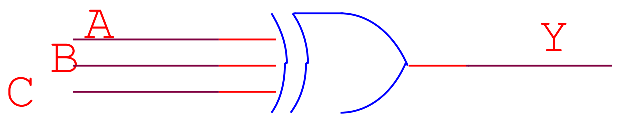
Don’t Cares

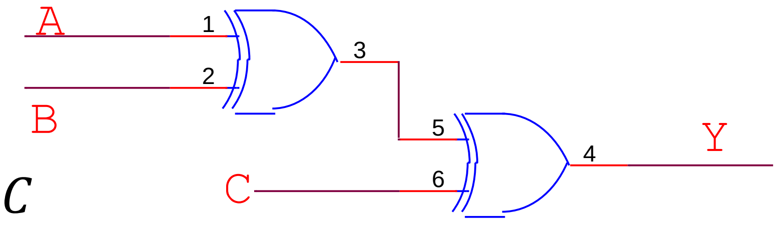
A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

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

Multilevel Combinational Logic

- Sum of Products (SOP) form is called two-level logic since it consists of **a level of AND gates** connected to **a level of OR gates**.
- In some cases, multilevel combinational circuits may use less hardware than their two-level counterparts.
- For 3 input xor gate we need to use 3 inverters, 4 AND Gates (3 input), and 1 OR gate (4-input) (Can also be implemented by 2 XOR gates cascaded)





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

$$Y = A \oplus B \oplus C$$

$$Y = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$

$$Y = \sum(1,2,4,7)$$

Review$_7$

- DeMorgan’s Theorem
- Bubble Pushing
- Functional Completeness
- From Logic to Gates
- Circuit Schematic Rules
- Don’t Cares
- SystemVerilog Modules (Behavioral, Structural)
- HDL (Simulation, Synthesis)
- SystemVerilog Syntax

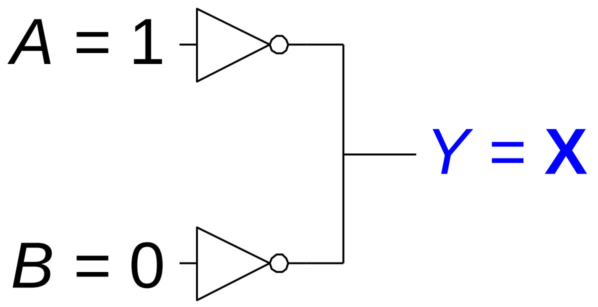
1^{st} Midterm

14/11/2018

17:30

Contention: X

- Contention: circuit tries to drive output to 1 **and** 0
  - Actual value somewhere in between
  - Could be 0, 1, or in forbidden zone
  - Might change with voltage, temperature, time, noise
  - Often causes excessive power dissipation

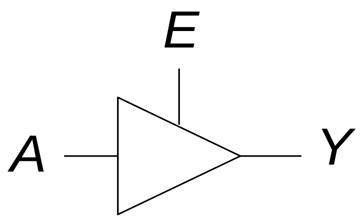


- **Warnings:**
  - Contention usually indicates a **bug**.
  - **X is used for "don't care" and contention** - look at the context to tell them apart

Floating: Z

- Floating, high impedance, open, high Z
- Floating output might be 0, 1, or somewhere in between
  - A voltmeter won't indicate whether a node is floating

Tristate Buffer

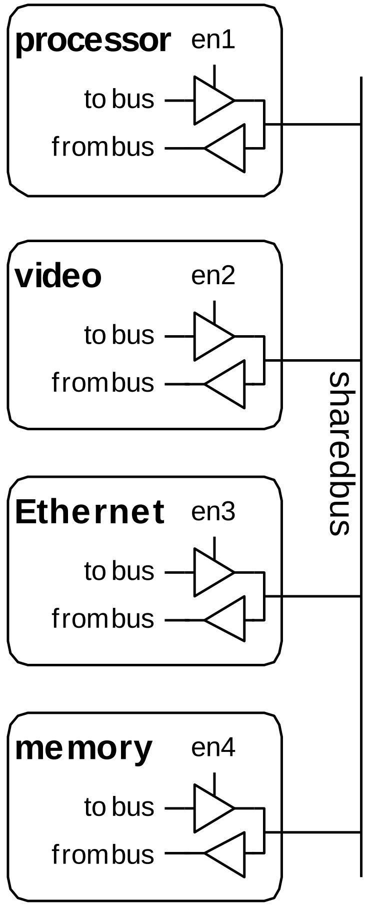


E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

the enable switch

Tristate Busses

- Floating nodes are used in tristate busses
  - Many different drivers
  - Exactly one is active at once



Karnaugh Maps (K-Maps)

- Karnaugh Maps are used to simplify Boolean equations. Recall that **simplification** involves **combining terms**.
- Each square represents a minterm
- Each square differs from an adjacent square by a change in only one bit.
- Note that the order on top row is 00, 01, 11, 10 and this order is called **Gray Code**
- For example, 3-bit Gray Code sequence is
- 000, 001, 011, 010, 110, 111, 101, 100

m_0	m_1
m_2	m_3

Karnaugh Maps (K-Maps)

- Boolean expressions can be minimized by combining terms
- K-maps minimize equations graphically
- $PA + P\overline{A} = P$

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

Y

C \ AB	00	01	11	10
0	**1**	0	0	0
1	**1**	0	0	0

Y

C \ AB	00	01	11	10
0	$\boldsymbol{\overline{A}\,\overline{B}\,\overline{C}}$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\,\overline{C}$
1	$\boldsymbol{\overline{A}\,\overline{B}C}$	$\overline{A}BC$	ABC	$A\overline{B}C$

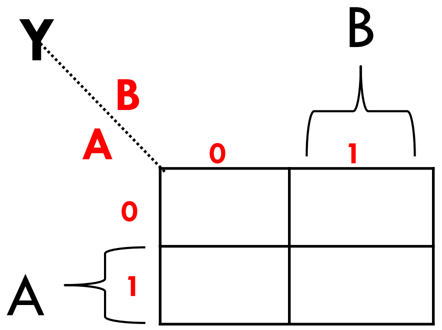
Karnaugh Maps (K-Maps)

- You can illustrate a Boolean function by Venn diagram
  - F=x'y+xz'
- If number of variables are large we need more effective ways. (**karnaugh map**)
- There are other methods: (Veitch map – quine McCuskey tabulation)
- For n variables $\mathbf{2^n}$ squares
- In a karnaugh map each square represents a **minterm**
- Between two adjacent squares only **one term** changes

$A'B'$	$A'B$
AB'	AB

2-Input K-Map

m_0	m_1
m_2	m_3



$A'B'$	$A'B$
AB'	AB

$$F(A,B) = \sum(0,1)$$

1	1
0	0

$$F(A,B) = \sum(1,3)$$

0	1
0	1

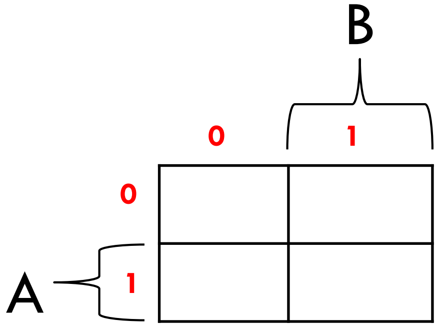
Karnaugh Maps (K-Maps)

- Two squares in the map are called **adjacent** if they differ by only **one** variable.
- For n variables there are **n adjacent** squares
- ***Sum of two adjacent n-variable** minterms can be simplified to an **AND** term consisting of **n-1** variables.

$F(A,B) = \sum(0,1)$

1	1
0	0

$\Big\}$ F=A'



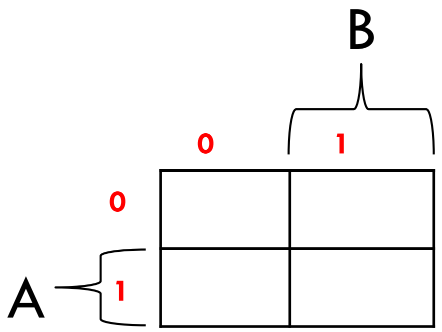
Karnaugh Maps (K-Maps)

$$F(A,B) = \sum(1,2,3)$$

$$Y = A + A'B \quad \Big\} \text{ Not minimum}$$

$$Y = A + B \quad \Big\} \text{ minimum}$$

0	1
1	1

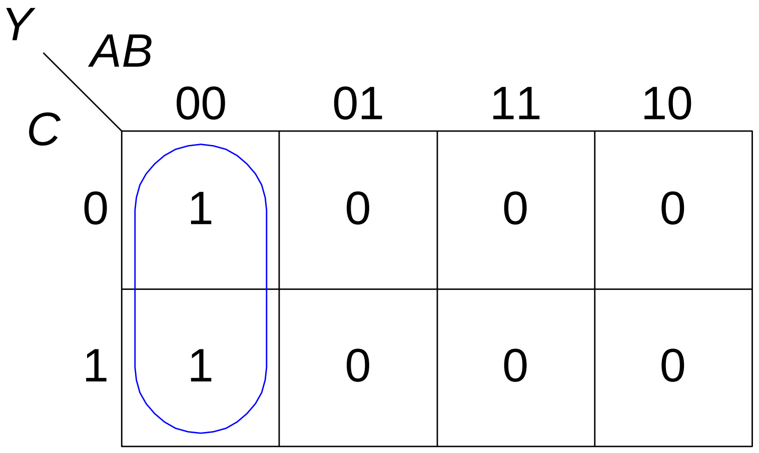


- *Can use one square **more than once**

K-Map

- Circle 1's in adjacent squares
- In Boolean expression, include only literals whose true and complement form are ***not*** in the circle

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



$$Y = \bar{A}\bar{B}$$

3-Input K-Map

Y

C \ AB	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

Truth Table

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

K-Map

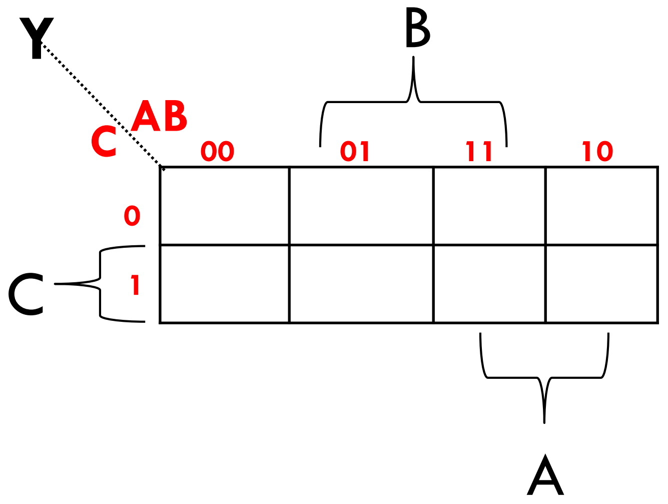
Y

C \ AB	00	01	11	10
0				
1				

3-Input K-Map

m_0	m_2	m_6	m_4
m_1	m_3	m_7	m_5

$A'B'C'$	$A'BC'$	ABC'	$AB'C'$
$A'B'C$	$A'BC$	ABC	$AB'C$



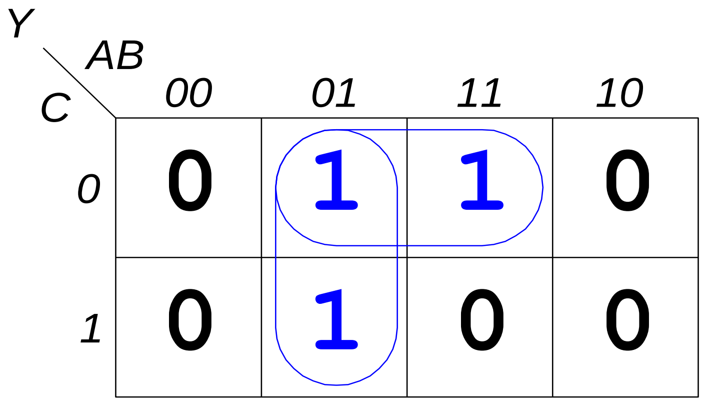
3-Input K-Map

Y: C \ AB	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

Truth Table

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

K-Map



Y: C \ AB	00	01	11	10
0	0	1	1	0
1	0	1	0	0

$$Y = \bar{A}B + B\bar{C}$$

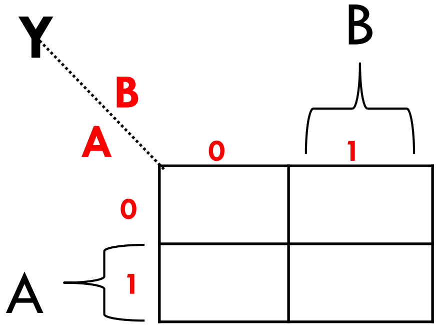
Review$_8$

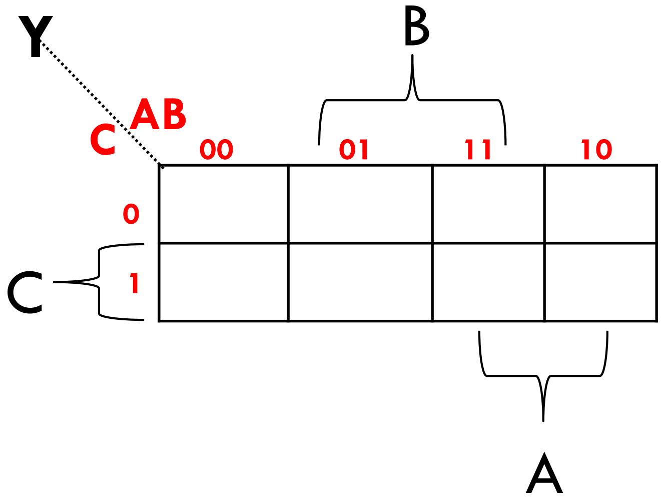
- Don’t Cares
- Floating: Z
- Karnaugh Maps (K-Maps)
- SystemVerilog Modules (Behavioral, Structural)
- HDL (Simulation, Synthesis)
- SystemVerilog Syntax

1st Midterm
14/11/2018
17:30

Review$_8$

- 2 input/3 input Karnaugh Maps (K-Maps)





K-Map Definitions

- **Complement:** variable with a bar over it

  $\bar{A}, \bar{B}, \bar{C}$
- **Literal:** variable or its complement

  $\bar{A}, A, \bar{B}, B, C, \bar{C}$
- **Implicant:** product of literals

  $A\bar{B}C, \bar{A}C, BC, B$
- **Prime implicant:** implicant corresponding to the largest circle in a K-map

K-Map Rules

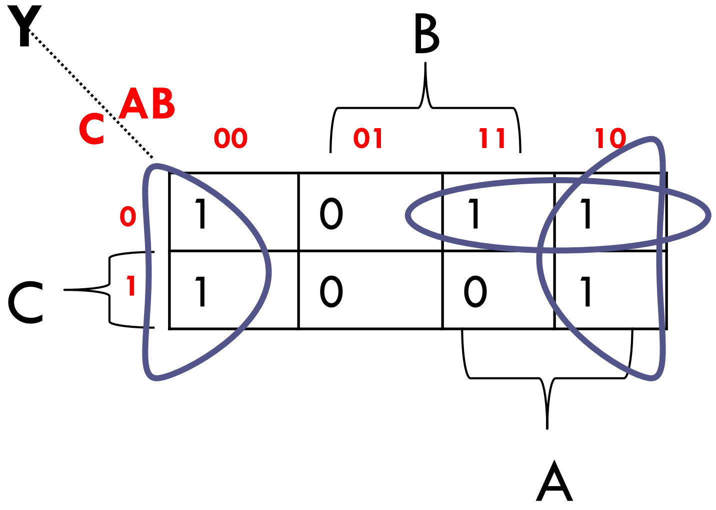
- Every 1 must be circled at least once
- Each circle must span a power of 2 (i.e. 1, 2, 4) squares in each direction
- Each circle must be as large as possible
- A circle may wrap around the edges
- A “don't care” (X) is circled only if it helps minimize the equation
- Goal: find the **smallest number of largest-possible circles**

Example

Ex. 1:

$$F(A,B,C) = \sum(0,1,4,5,6)$$

$$F(A,B,C) = A'B'C' + A'B'C + AB'C' + AB'C + ABC'$$

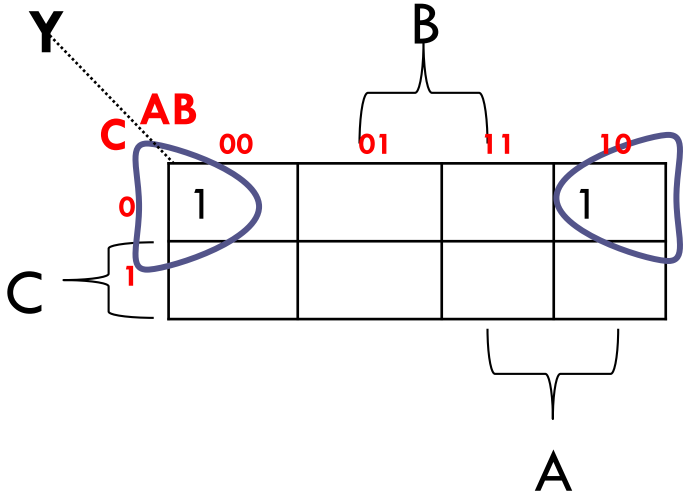


C \ AB	00	01	11	10
0	1	0	1	1
1	1	0	0	1

Example

Y = A′B′C′+AB′C′

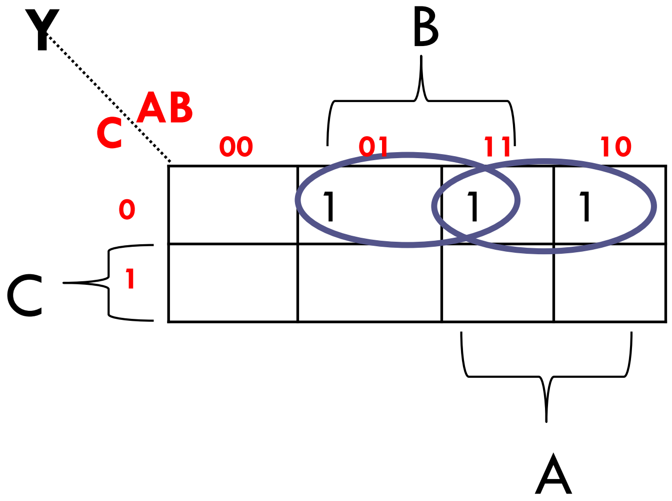
Y = B′C′



- *A circle may wrap around the **edges** of the map.

Example

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

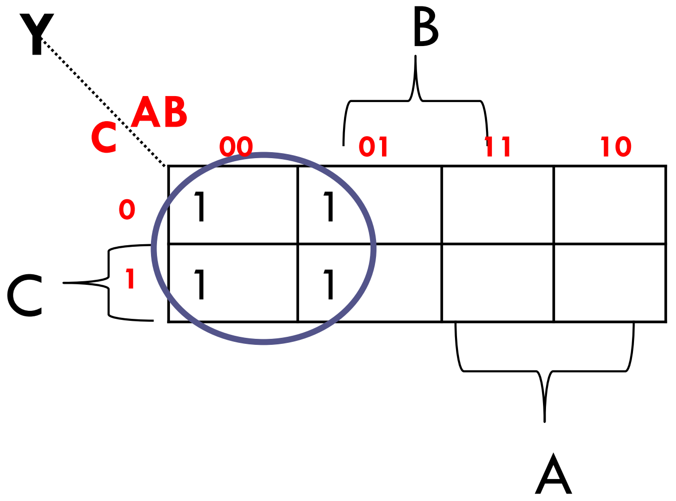


F = BC′+AC′

- *A 1 in a map may be used by **multiple** circles.

Example

$F = A'B'C' + A'B'C + A'BC' + A'BC$



$F(A,B) = \sum(0,1)$

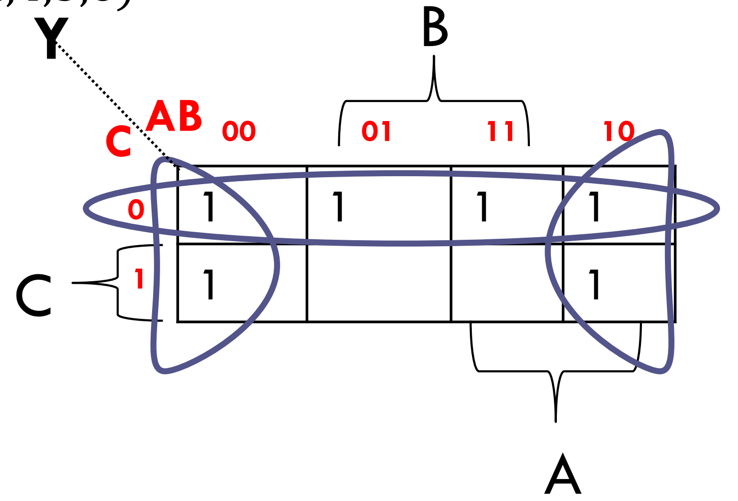
$F = A'$

- The **number** of **adjacent squares** must be $\mathbf{2^k}$ (1,2,4,8,16..)
- K-map simplification results in expressions with **fewer literals**
- Combine **the largest number of adjacent squares** as you can

Example

Ex. 1:

$F(A,B,C) = \sum(0,1,2,4,5,6)$



$F = B' + C'$

Ex. 2:

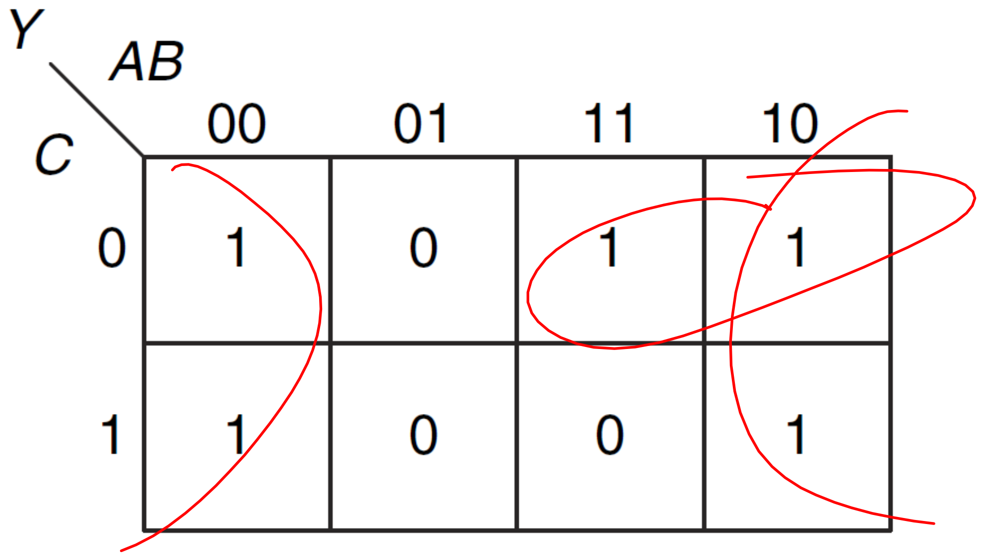
$F(A,B,C) = \sum(1,3,2,4,5,6)$

Ex. 3:

		1	
	1	1	1

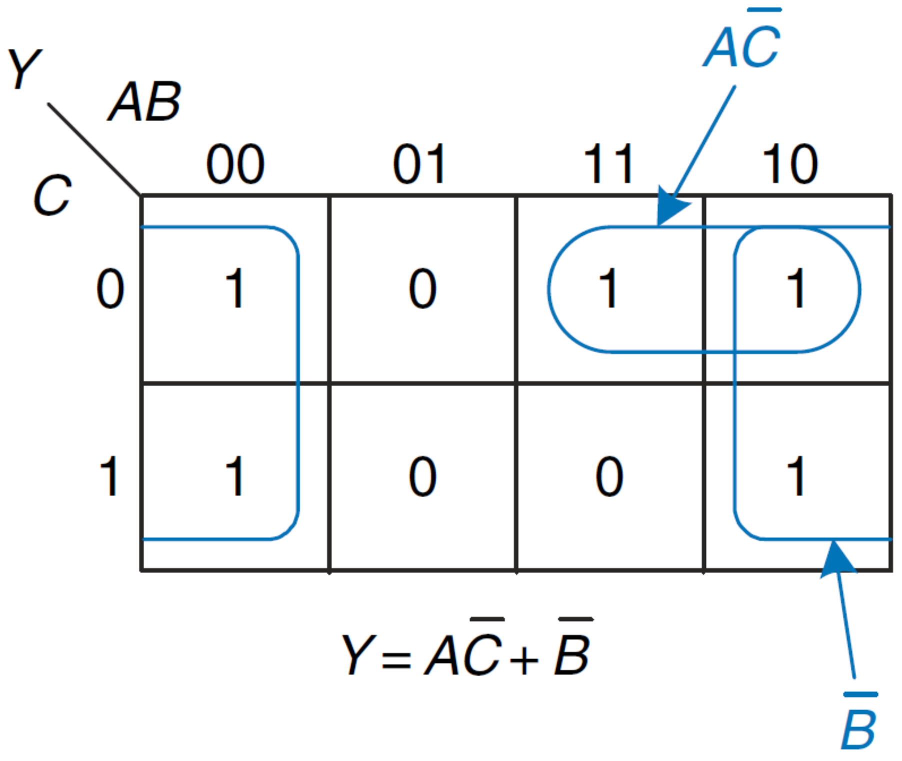
$F(A,B,C) = ?$

Exercise



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

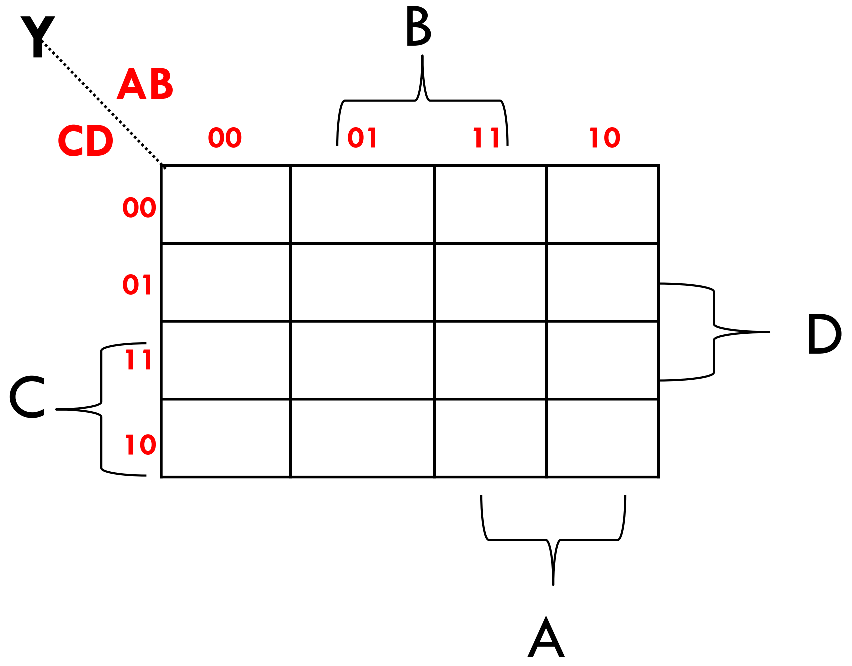
Solution



$$\begin{array}{c|cccc} C \backslash AB & 00 & 01 & 11 & 10 \\ \hline 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{array}$$

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

4-Input K-Map



- A and B combinations in the top row are in a peculiar order: 00, 01, 11, 10 (**Gray code)**
- Adjacent entries differ only in a single variable.

4-Input K-Map

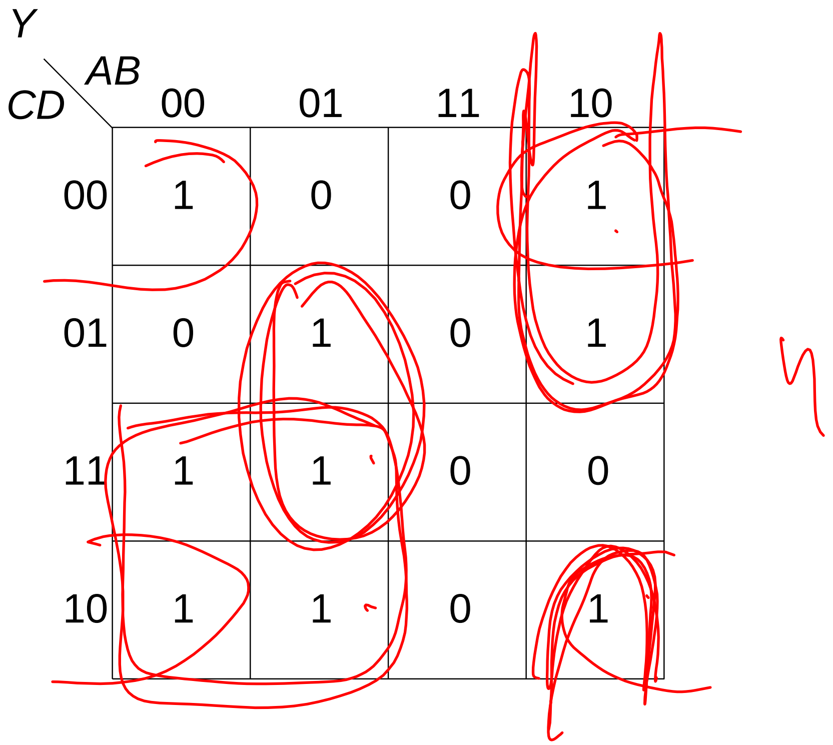
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Y

CD \ AB	00	01	11	10
00				
01				
11				
10				

4-Input K-Map

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

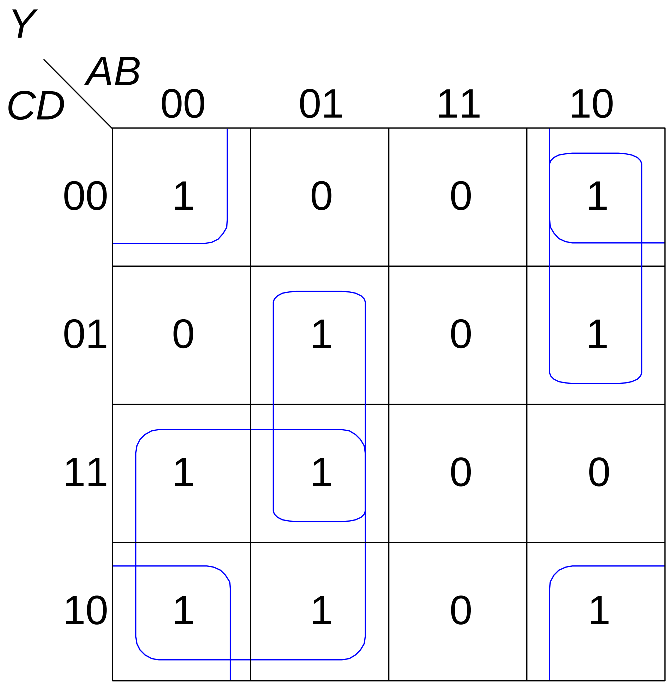


Y

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

4-Input K-Map

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



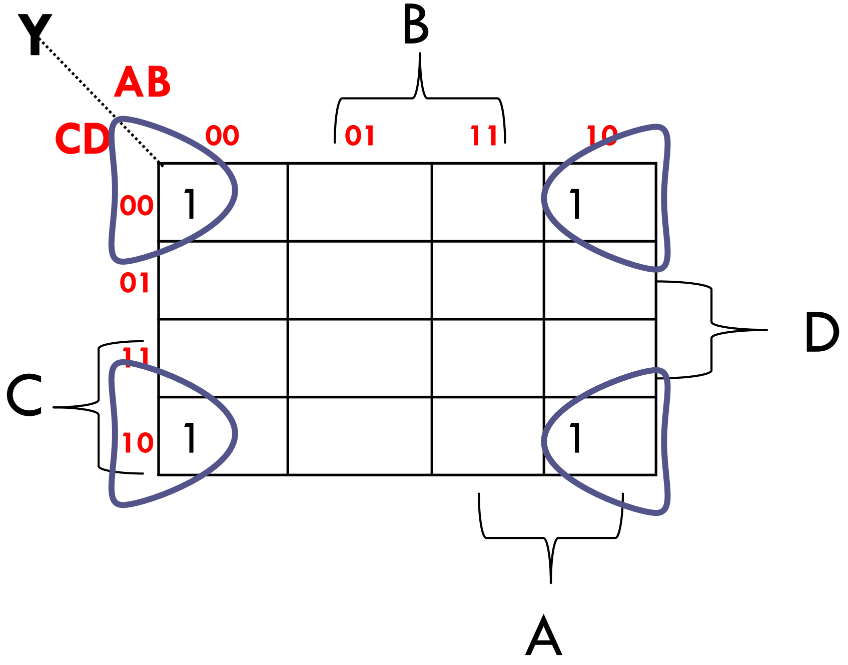
Y — *CD* \ *AB*	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	1

$$Y = \bar{A}C + \bar{A}BD + A\bar{B}\bar{C} + \bar{B}\bar{D}$$

Example

$$F(A,B,C,D) = A'B'C'D' + A'B'CD' + AB'C'D' + AB'CD'$$

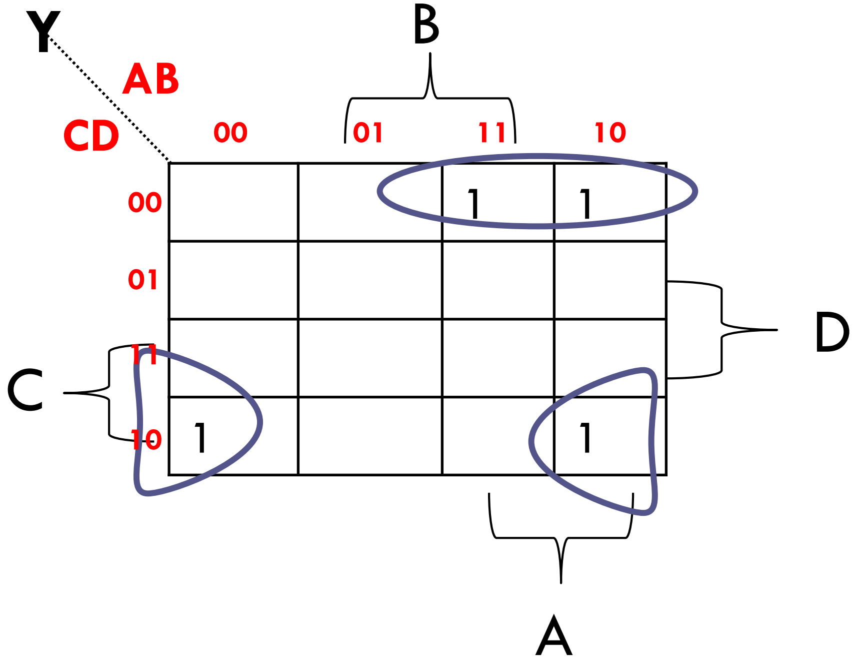
$$F(A,B,C,D) = A'B'D' + AB'D' = B'D'$$



Example

$F(A,B,C,D) = AB'C'D' + ABC'D' + AB'CD + A'B'CD$

$F(A,B,C,D) = AC'D' + CB'D'$



Review$_9$

- Karnaugh Maps (K-Maps)

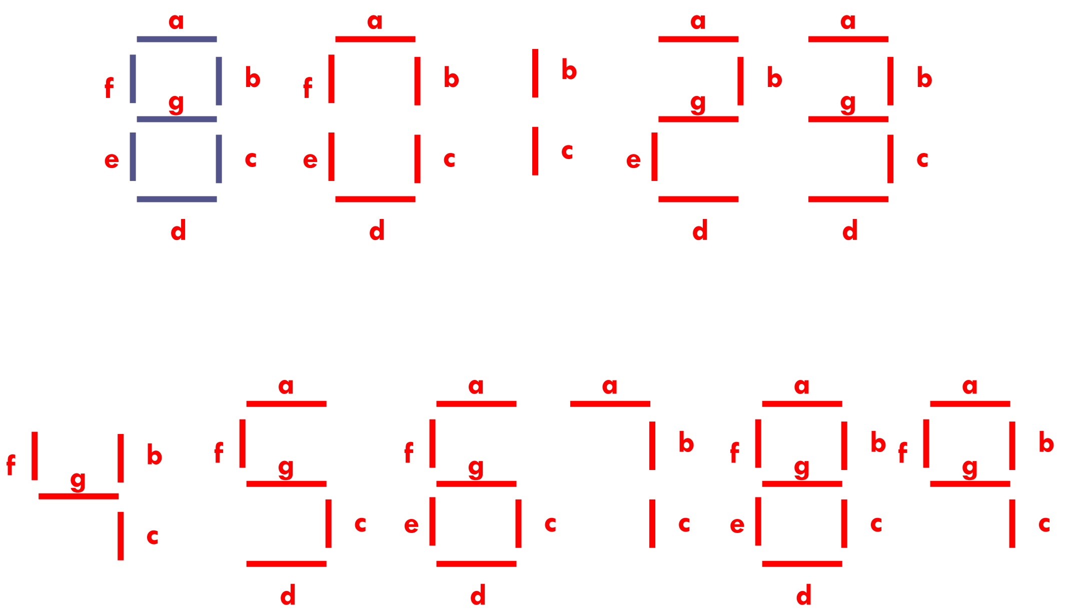
1^{st} Midterm

14/11/2018

17:30

Example-7 segment

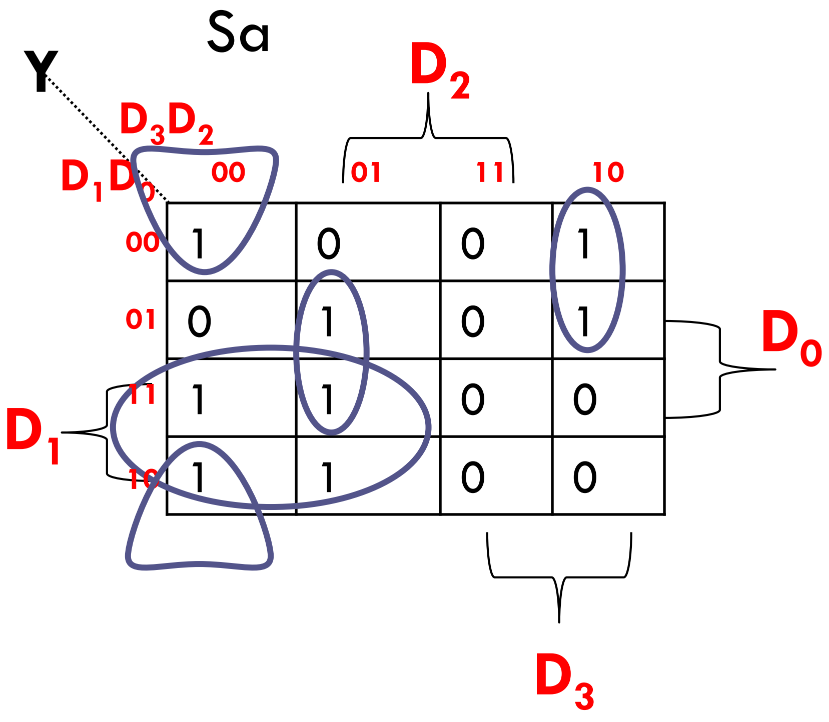
- Takes 4 bit data and produces 7 outputs to display 0..9



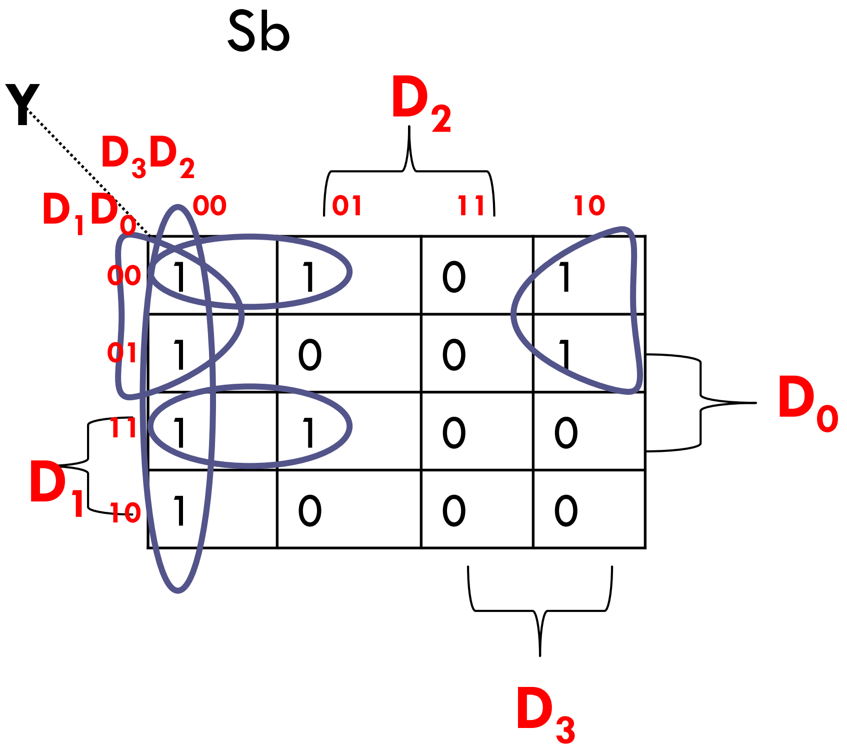
D3	D2	D1	D0	Sa	Sb	Sc	Sd	Se	Sf	Sg
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	1	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Example-7 segment

- Each seven segment outputs is an independent function of 4 input variables
- Find boolean equations for the outputs Sa, Sb



D_1D_0 \ D_3D_2	00	01	11	10
00	1	0	0	1
01	0	1	0	1
11	1	1	0	0
10	1	1	0	0



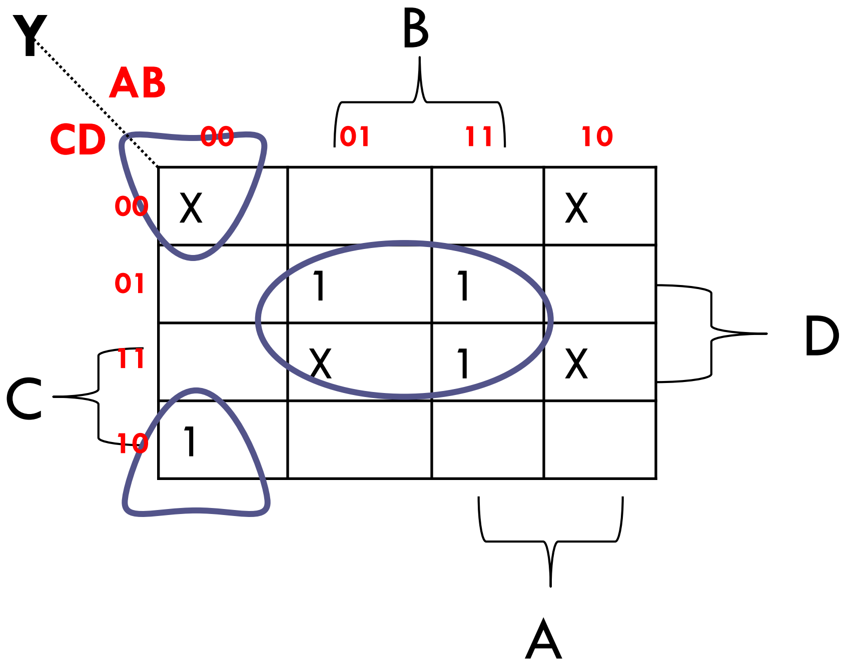
D_1D_0 \ D_3D_2	00	01	11	10
00	1	1	0	1
01	1	0	0	1
11	1	1	0	0
10	1	0	0	0

K-Maps with Don’t Cares

- Sometimes we don’t care abbout the output value for some input combinations (**problem specification**)
- These input combinations **never appear** at the **input**
- The **output is not important** for these input combinations.
- For such input combinations output may take a value of **0 or 1** (we don’t care)
- We put a **X** in the k-map to represent **don’t care**
- X can be taken as **0 or 1** to get the simplest expression
- Complement of **X** is **X**

Example

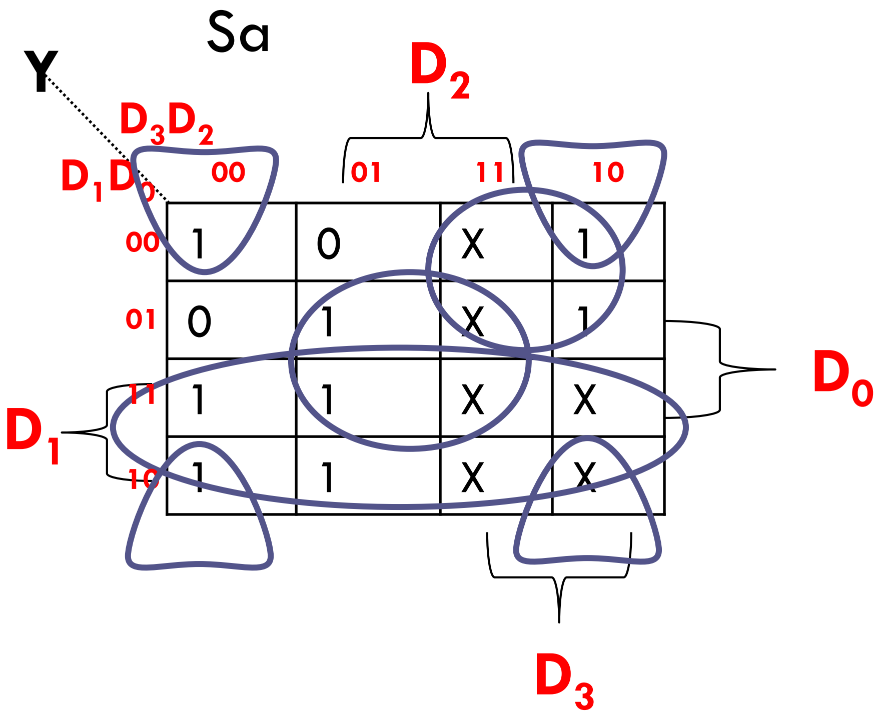
$$F(A,B,C,D) = \sum(2,5,13,15) + \sum_d(0,8,11,7)$$



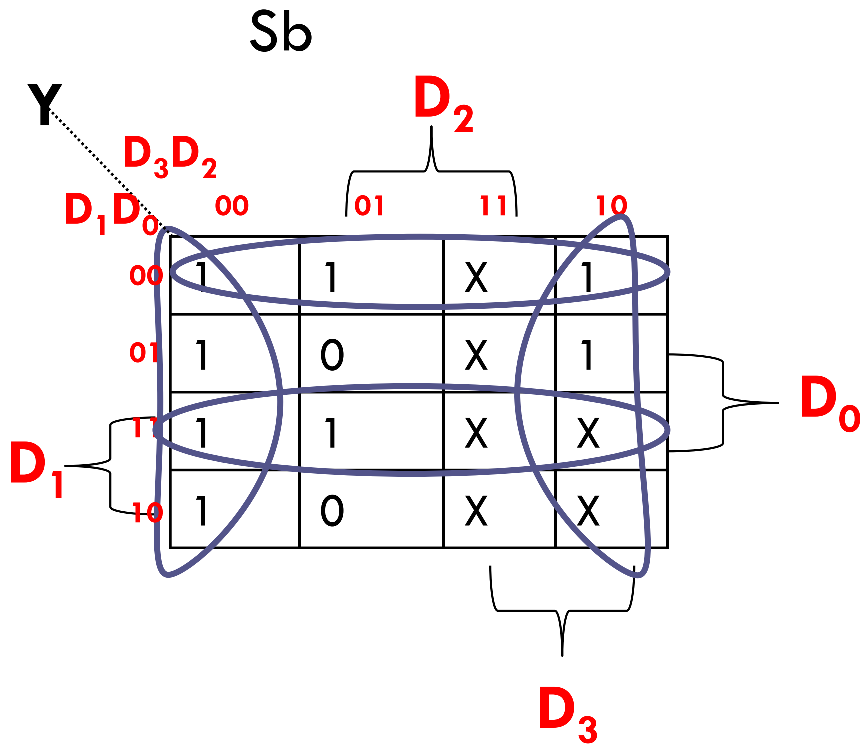
$$F = BD + A'B'D'$$

Example-7 segment

- Consider the outputs of 7-segment display as X for illegal input values of 10-15



D1D0 \ D3D2	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	1	X	X
10	1	1	X	X



D1D0 \ D3D2	00	01	11	10
00	1	1	X	1
01	1	0	X	1
11	1	1	X	X
10	1	0	X	X

K-Maps with Don’t Cares

A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

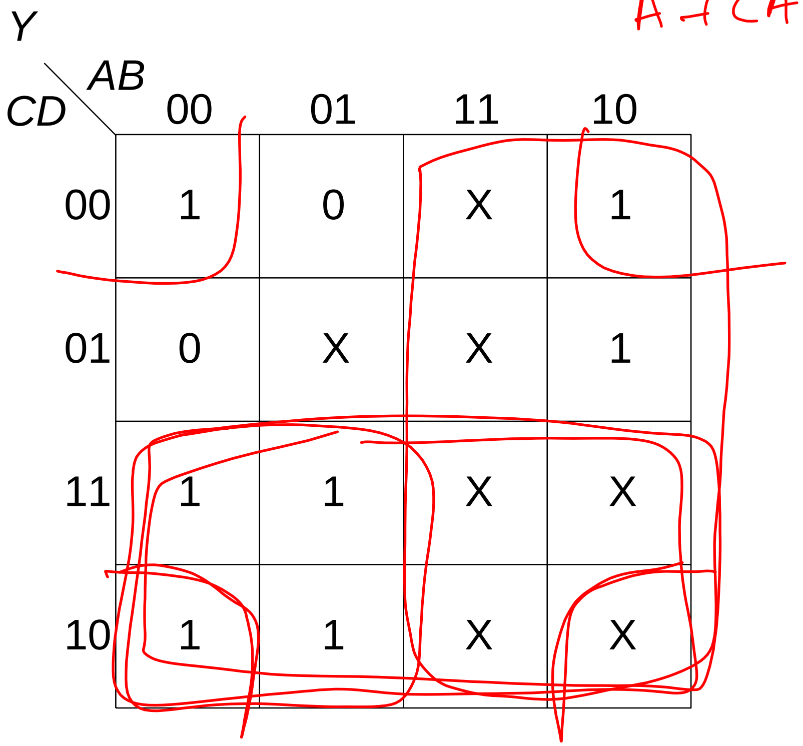
Y

CD \ *AB*	00	01	11	10
00				
01				
11				
10				

K-Maps with Don't Cares

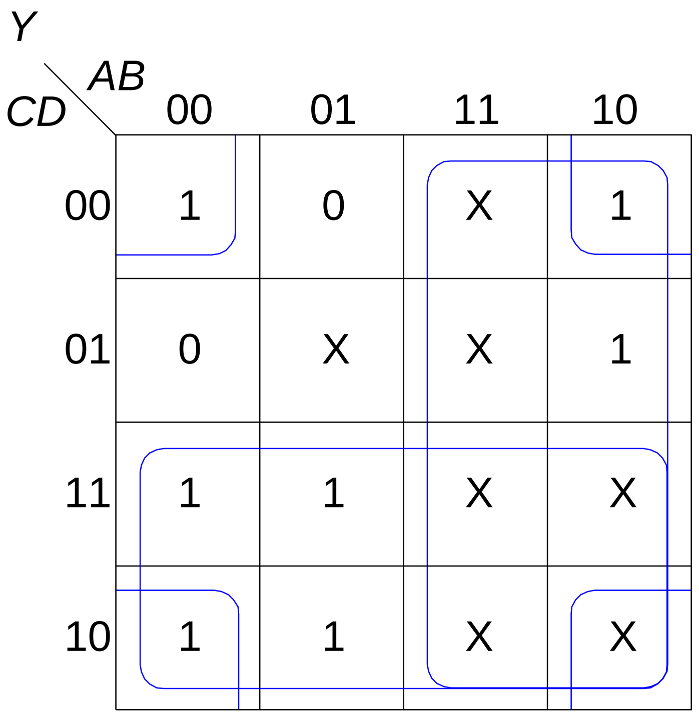
A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

$A + C\bar{A}$



K-Maps with Don’t Cares

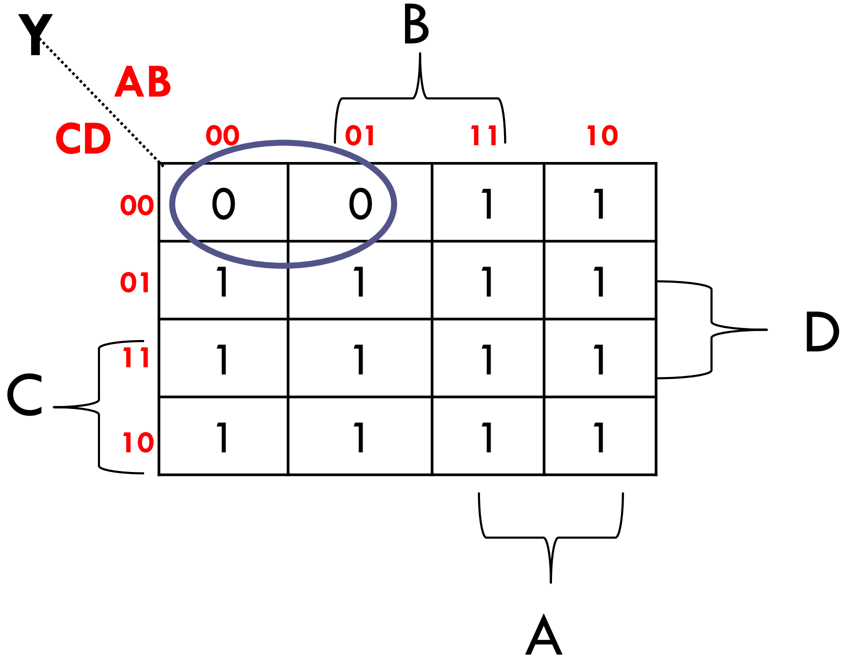
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X



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

Example

$F(A, B, C, D) = ?$



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

- POS can also be used, be careful with the output function, boolean expression will be the **complement** of the output function.

Summary

- Boolean algebra and Karnaugh maps are methods of logic **simplification**.
- Goal is to find a **low-cost** method of implementing a logic function
- Method is useful for small problems but for large scaled problems logic synthesizers (computer programs) are used

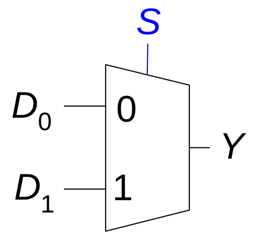
Combinational Building Blocks

- Multiplexers
- Decoders

Multiplexer (MUX)

- Selects between one of N inputs to connect to output
- $\log_2 N$-bit select input – control input
- **Example:**

2:1 Mux

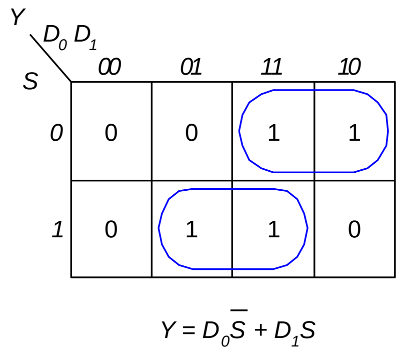


S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

S	Y
0	D_0
1	D_1

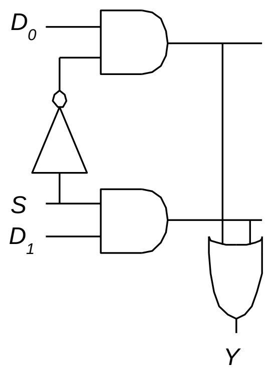
Multiplexer Implementations

- **Logic gates**
  - Sum-of-products form

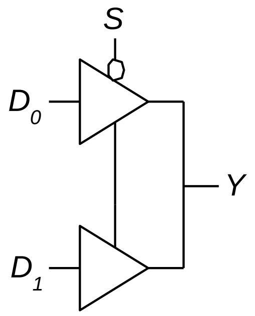


Y ($S \backslash D_0D_1$)	00	01	11	10
0	0	0	1	1
1	0	1	1	0

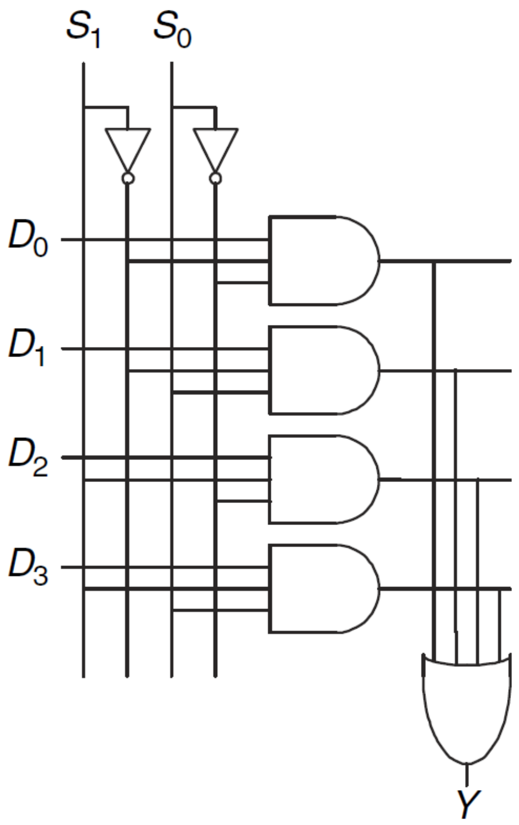
$Y = D_0\overline{S} + D_1S$

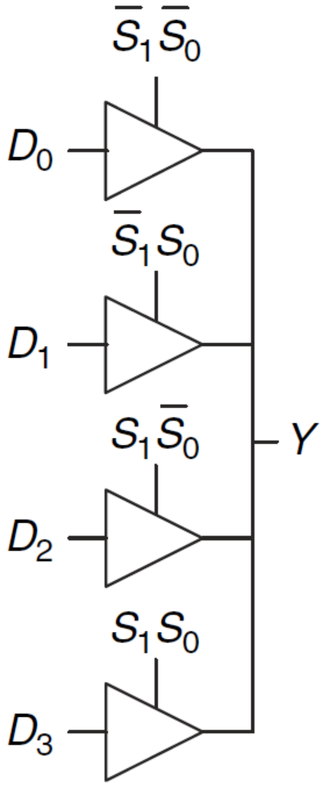


- **Tristates**
  - For an N-input MUX, use N tristates
  - Turn on exactly one to select the appropriate input



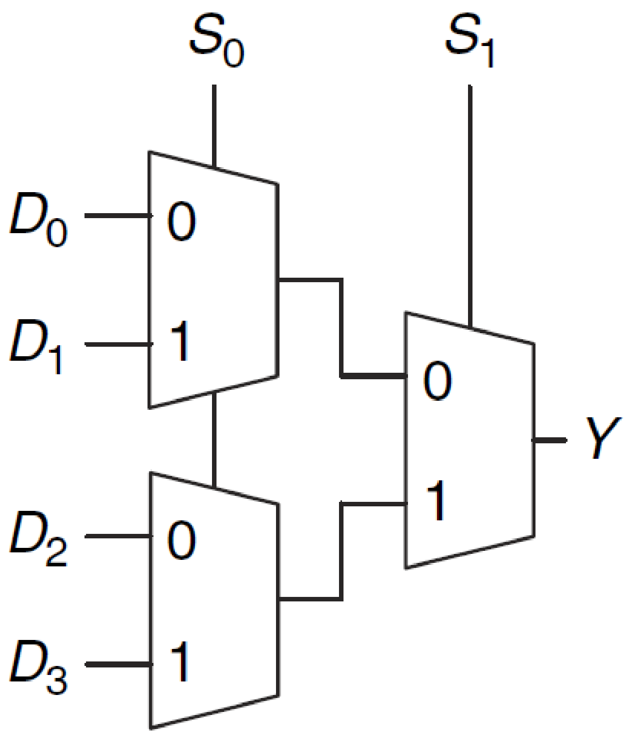
4:1 Mux Implementations





4:1 Mux Implementations

Implement a 4:1 Mux using 2:1 Muxes

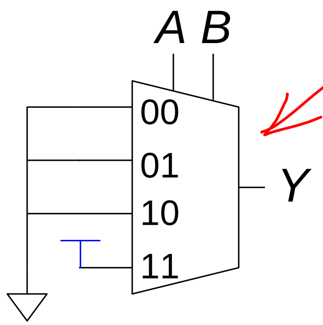


Logic using Multiplexers

- Using the MUX as a lookup table

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

$Y = AB$



to implement the output from a truth table.

Can we implement the same logic with a 2-to-1 MUX?

- 2^N-to-1 MUX needed for n-inputs

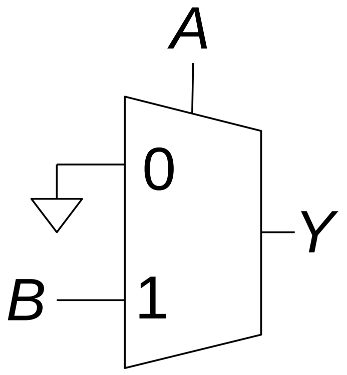
Logic using Multiplexers

- Reducing the size of the MUX

$Y = AB$

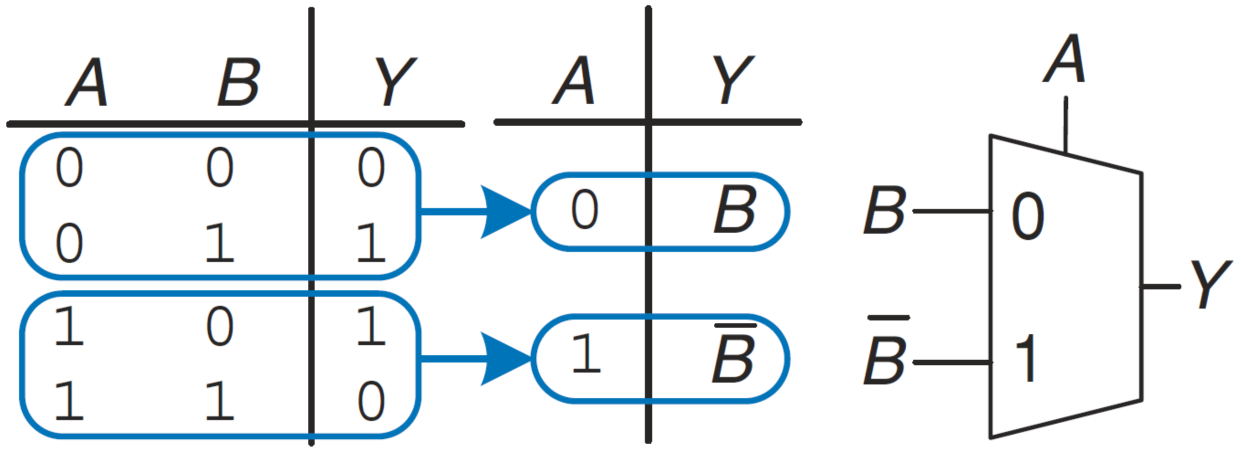
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

A	Y
0	0
1	B



- 2^{N-1}-to-1 MUX needed for n-inputs

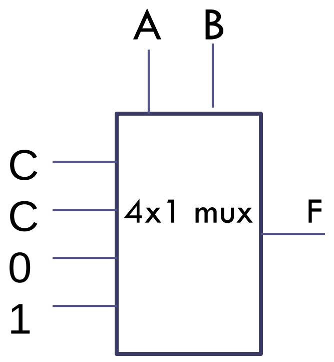
Exercise: Implement XOR using a 2:1 MUX



Exercise

Realize $F(A,B,C) = \Sigma\,(1,3,6,7)$ using 4x1 Mux

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



Or, you may connect A to inputs and B and C to select inputs. Then

A'	0	1	2	3
A	4	5	6	7
	0	A'	A	1

Exercise

Realize F(A, B, C, D)= S (1,2,3,4,8,11,12,15) using 8x1 mux

Use A,B,C as select inputs.

D’	0	2	4	6	8	10	12	14
D	1	3	5	7	9	11	13	15
	D	1	D’	0	D’	D	D’	D

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

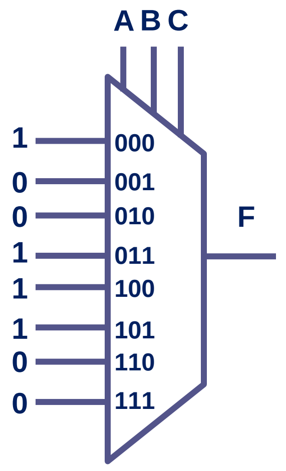
D	A	B	C	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

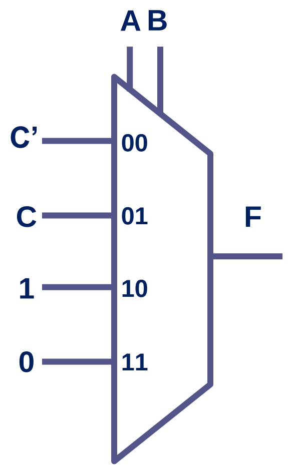
Exercise

Ex. 1: Implement the function F using an 8:1 MUX.

$$F(A,B,C) = AB' + B'C' + A'BC$$

A	B	C	F
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



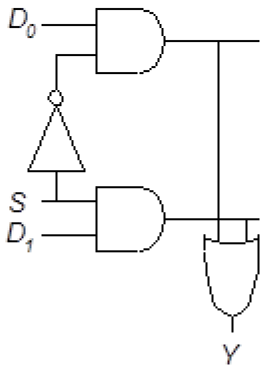


*The multiplexer can act as a lookup table where each row in the truth table corresponds to a mux input.

System Verilog

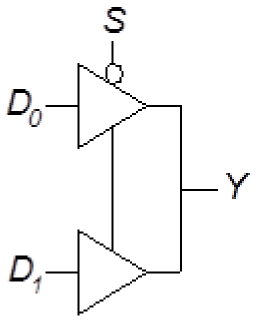
Multiplexer Description

```
// 2x1 multiplexer

module mux2to1( input logic in1, in0, s,
                output logic y);

        assign y = s&in1 | ~s&in0;
endmodule
```



Using conditional assignment, we have a simpler statement:

```
        assign  y = s ? in1 :  in0
```



Conditional Assignment

Four 2x1 muxes

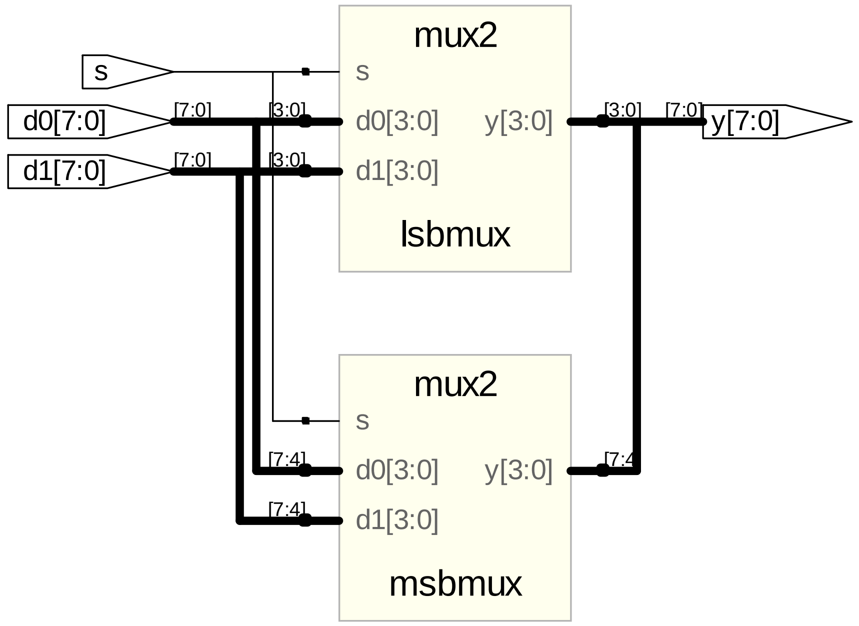
```
module mux2(input  logic [3:0] d0, d1, 
            input  logic       s, 
            output logic [3:0] y);
   assign y = s ? d1 : d0; 
endmodule
```

`? :`   is also called a *ternary operator* because it operates on 3 inputs: `s`, `d1`, and `d0`.

Implement an 8-bit mux2 using two 4-bit mux2s

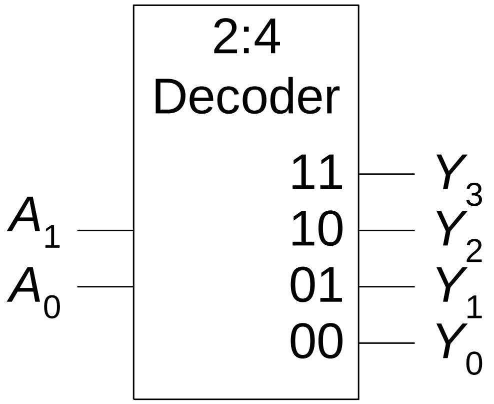
SystemVerilog:

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```



Decoders

- N inputs, 2^N outputs → Decoders have more outputs than inputs
- "One-hot" outputs: only one output HIGH at once → one hot



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

Decoders

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

Each output represents a single minterm

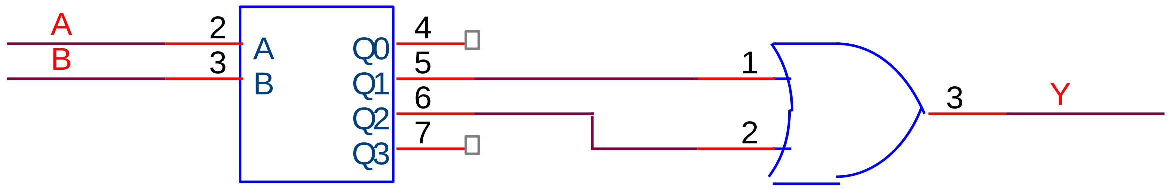
$\mathbf{Y_3} = \mathbf{m_3} = AB$

$\mathbf{Y_2} = \mathbf{m_2} = AB'$

$\mathbf{Y_1} = \mathbf{m_1} = A'B$

$\mathbf{Y_0} = \mathbf{m_0} = A'B'$

- 2:4 decoder can be implemented using 2 inverters and 4 two-input AND gates.
- An N:2^N decoder can be constructed by using N inverters and 2^N N-input AND gates
- Decoders can be combined with OR gates to build logic functions.
- Ex:



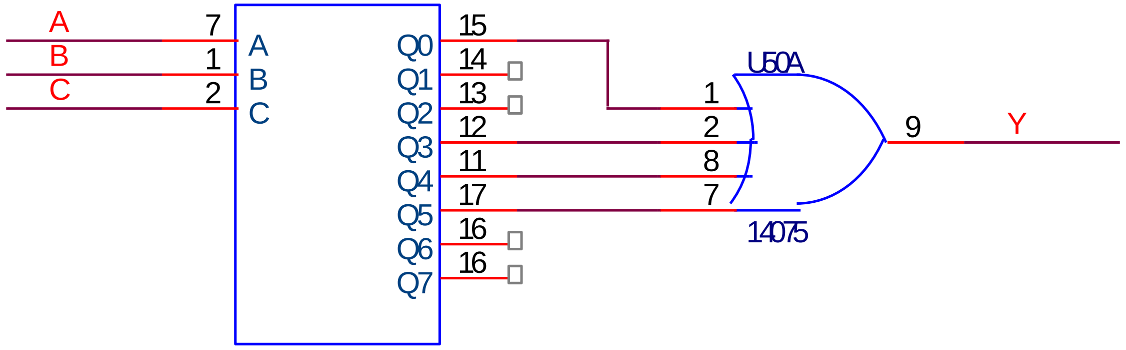
$Y = AB' + A'B$ $\quad$ $Y = A \oplus B$

Decoders

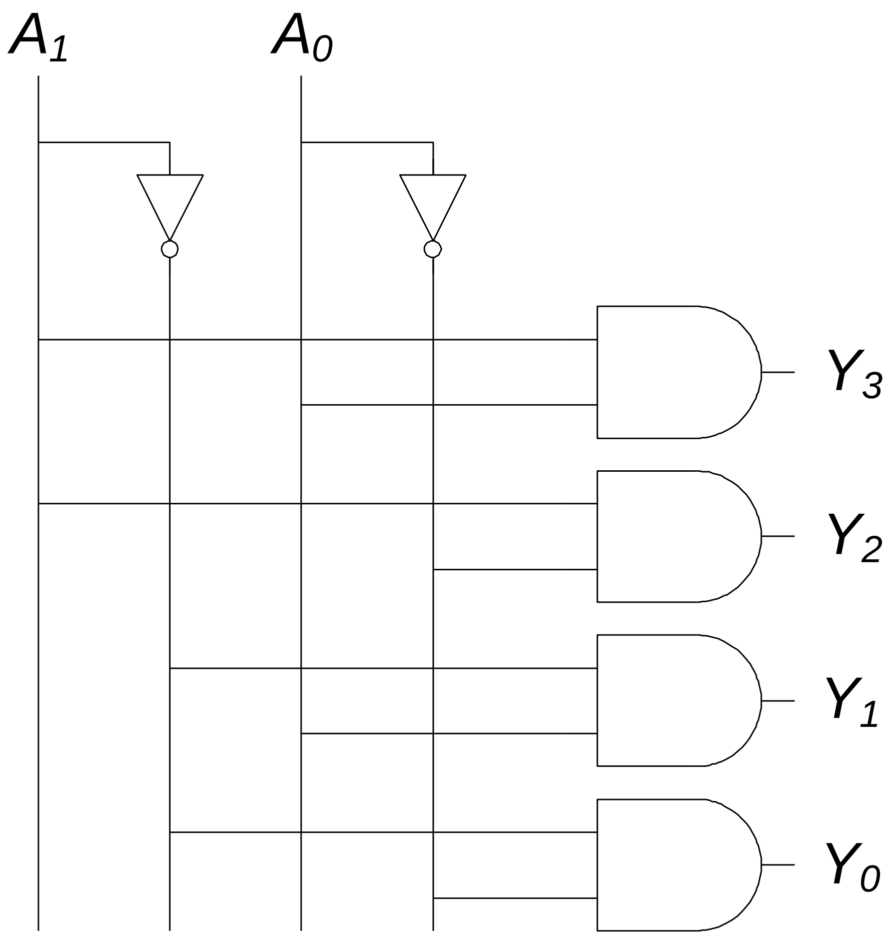
- Ex: Realize Y by using 3:8 decoder and OR gate

$Y = AB' + B'C' + A'BC$

$$Y = \sum(0,3,4,5)$$



Decoder Implementation



Decoders

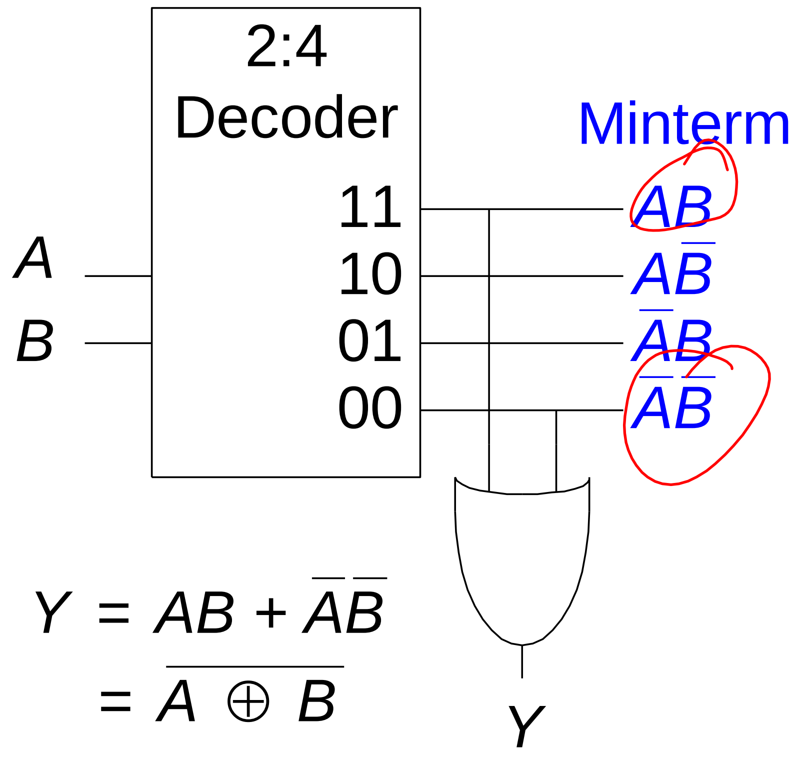
- n to m line decoders ($m \leq 2^n$)
- 2^n **minterms** generated
- Each **output** represents a **minterm**
- Any combinational circuit with n inputs and m outputs can be implemented by a "**n to 2**" line decoder and **m** OR gates

System Verilog Implementation of a 2:4 Decoder

```
module  decoder2to4 ( input logic in1,in0,
                      output logic y3,y2,y1,y0) ;

        assign y3 = in1 & in0 ;
        assign y2 = in1 & ~in0 ;
        assign y1 = ~in1 & in0 ;
        assign y0 = ~in1 & ~in0 ;
endmodule
```

Logic Using Decoders

- OR minterms

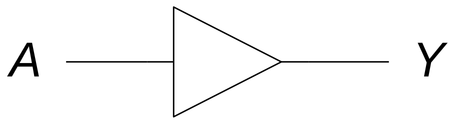


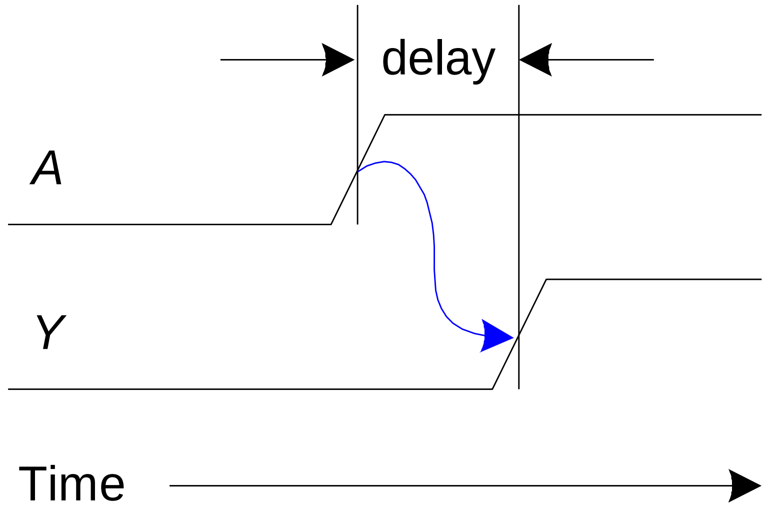
$$Y = AB + \overline{A}\,\overline{B}$$
$$= \overline{A \oplus B}$$

Logic using decoders

Timing

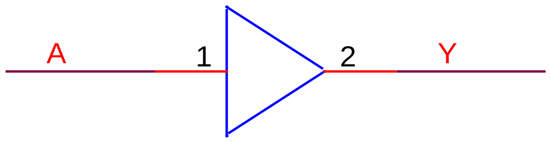
- Delay between input change and output changing
- How to build fast circuits?

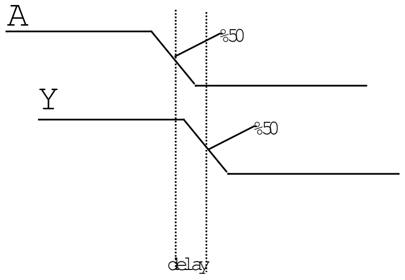




Timing

- Delay is measured from the %50 point of input signal to %50 point of output signal.
- **Contamination delay: minimum** time from when an input changes until any output starts to change its value. (t_{cd})
- **Propogation delay: maximum** time from when an input changes until the outputs reach their final value. (t_{pd})
- Circuit delays are on the order pico-nano seconds.
- Primary cause of delay in circuits is charging/discharging **capacitances** in the circuits.
- Delays are specified within datasheets for each gate.
- Delays are also determined by the **path** a signal **travels**.

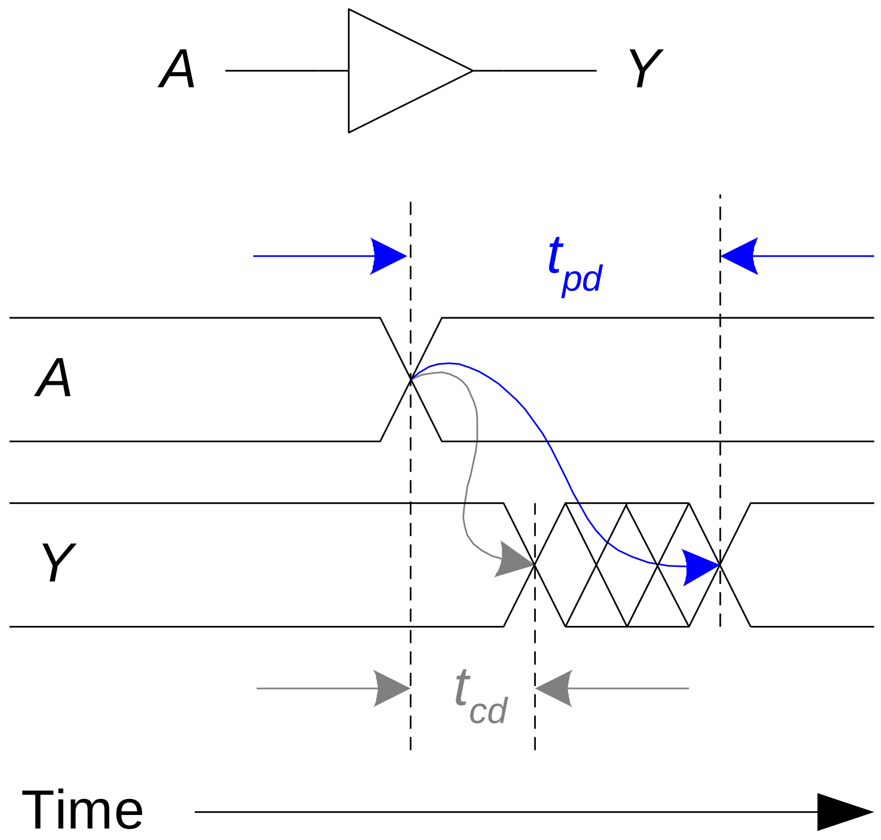




Propagation & Contamination Delay

- **Propagation delay:** t_{pd} = max delay from input to output
- **Contamination delay:** t_{cd} = min delay from input to output

[illegible]



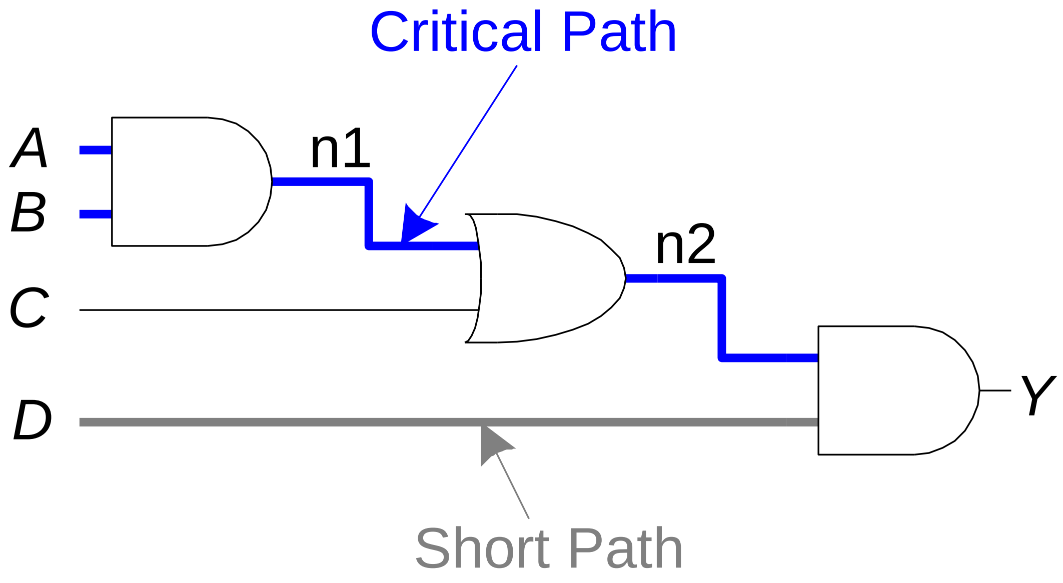
[illegible]

ELSEVIER

Propagation & Contamination Delay

- Delay is caused by
  - Capacitance and resistance in a circuit
  - Speed of light limitation
- Reasons why t_{pd} and t_{cd} may be different:
  - Different rising and falling delays
  - Multiple inputs and outputs, some of which are faster than others
  - Circuits slow down when hot and speed up when cold

Critical (Long) & Short Paths

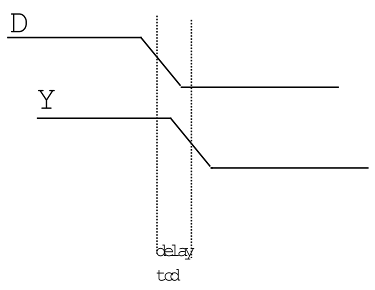


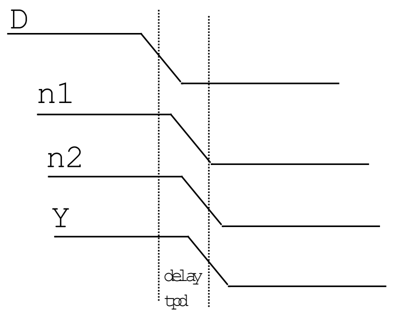
Critical (Long) Path: $t_{pd} = 2t_{pd\_AND} + t_{pd\_OR}$

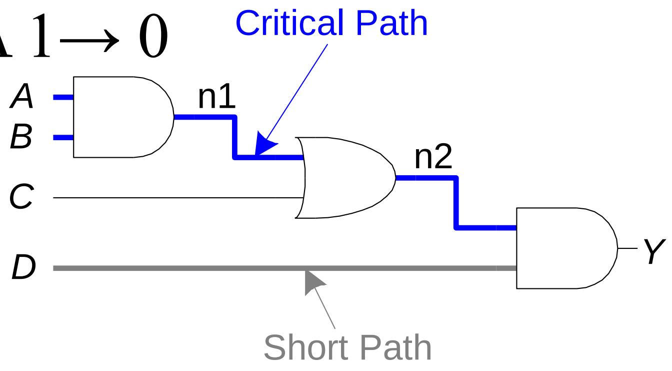
Short Path: $t_{cd} = t_{cd\_AND}$

Critical (Long) & Short Paths

- Ex:
  - a. Assume A=1, B=1, C=0 and D $1 \rightarrow 0$
  - b. Assume B=1, C=0, D=1 and A $1 \rightarrow 0$





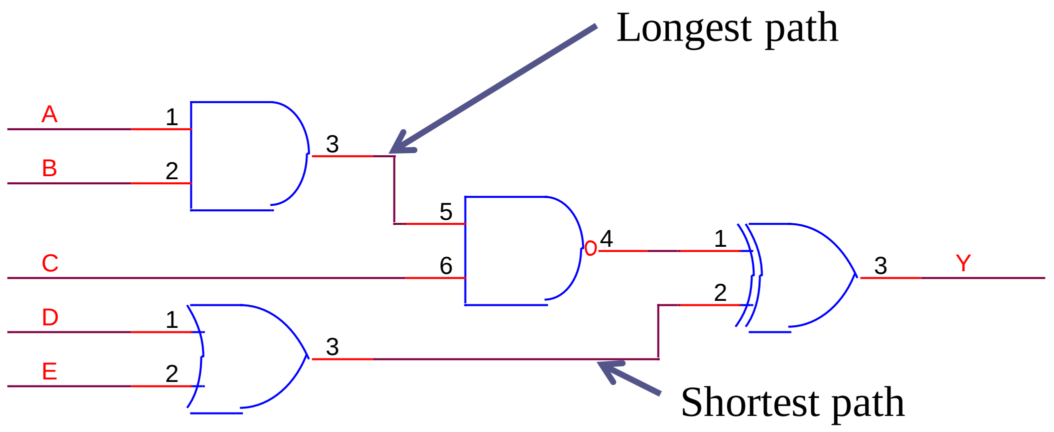


- Delays are also determined by the **path a signal travels**.
- **Propogation delay** is the sum of propogation delays of each element on the **critical path**.
- **Contamination delay** is the sum of contamination delays through each element on the **shortest path**.

Exercise

- Find the propogation delay and contamination delay of the circuit. Each gate has delays

$$t_{pd} = 100ps \text{ and } t_{cd} = 60ps$$



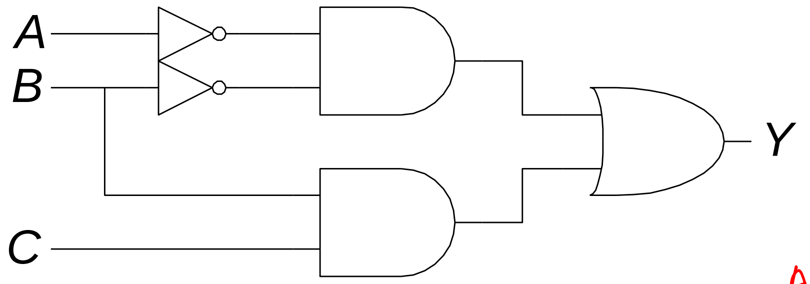
Glitches

- When a single input change causes multiple output changes

$\uparrow$ one change everything changes

Glitch Example

- What happens when A = 0, C = 1, B falls?



$A_{(7:0)}$: 0 1 2 ... 4 2 1

000 001 010 111 = 7

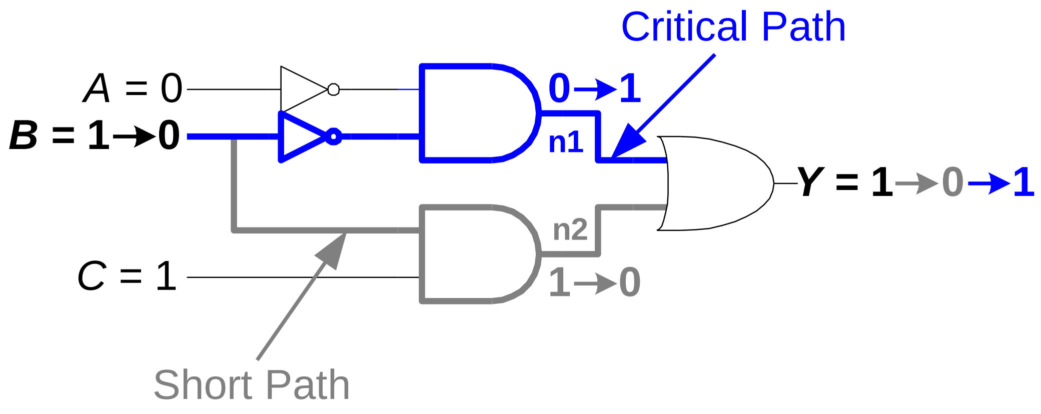
000 = 0

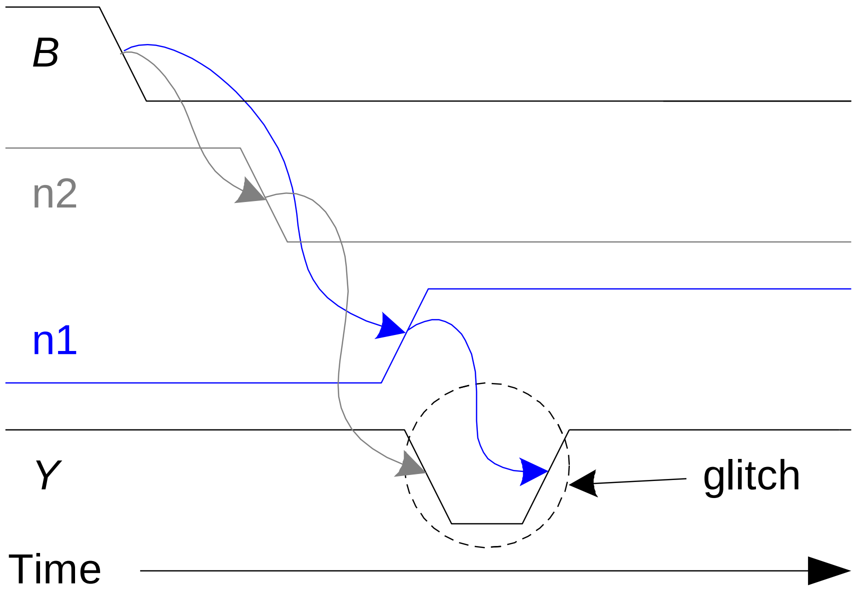
Y

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

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

Glitch Example (cont.)

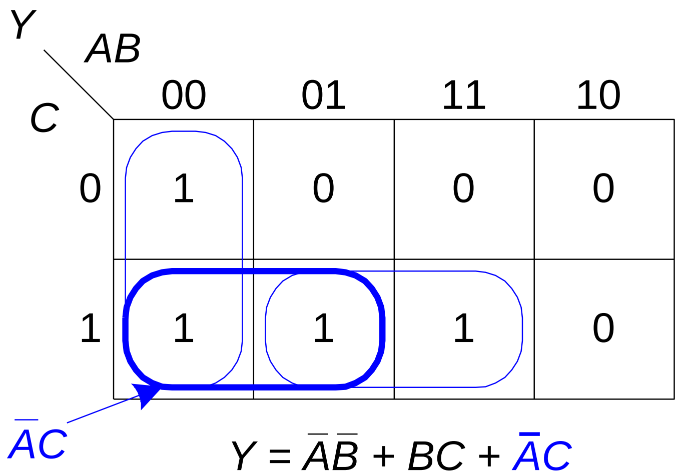




Fixing the Glitch

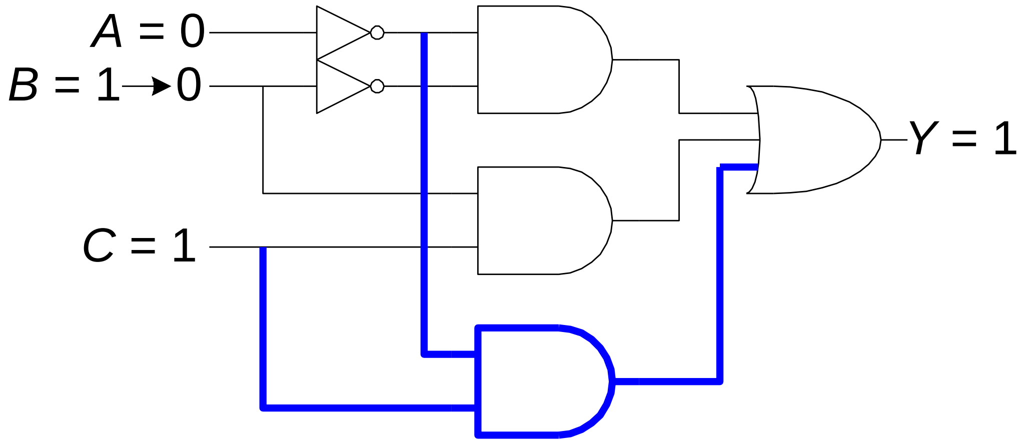
- Glitch can occur since one of the prime implicants turns on/off before the other prime implicant turn on/off.
- To prevent this type of glitch another circle can be added which will cover the boundaries of neighbour prime implicants on k-map
- This redundant implicant increases the cost of hardware

Fixing the Glitch



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

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



ELSEVIER

Why Understand Glitches?

- Glitches don't cause problems when **synchronous design** conventions are used (see Chapter 3)
- It's important to **recognize** a glitch: in simulations or on oscilloscope
- Can't get rid of all glitches – simultaneous transitions on multiple inputs can also cause glitches