Unit 4: Boolean Algebra

CSE 220: System Fundamental I

Stony Brook University

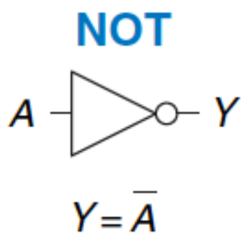
Joydeep Mitra

Digital Logic

- Digital systems perform operations on binary numbers.
- The fundamental building blocks used are *logic gates.*
- A logic gate is a digital circuit that takes *binary inputs* and produces *a binary output.*
    - Every logic gate is represented by a unique symbol
    - Inputs are drawn on the left (or top); outputs on the right (bottom)
    - Letters (e.g., A,B,C, and Y) indicate inputs and outputs
        - Letters may be subscripted with numbers (e.g., A_0, A_1)
- The relationship between inputs and output in a logic gate can also be represented in a *truth table*
    - A truth table indicates every combination of inputs and their corresponding output.
    - 1 indicates TRUE and 0 indicates FALSE
    - We will use 1, TRUE and 0, FALSE interchangeably

NOT Gate

- Has one input and one output.
- If input is TRUE, output is FALSE and vice-versa.
- The NOT gate is also called an inverter.

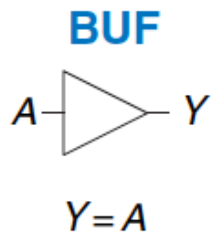


A	Y
0	1
1	0

- The line over A should be read as “NOT A”
- An alternative notation is `A’` also read as “NOT A”

Buffer

- Has one input and one output.
- If input is TRUE, output is TRUE and vice-versa.
- The buffer is like a wire; output is same as input.
- From digital perspective a buffer might seem useless.
- But, from an analog perspective, it has numerous desirable characteristics (e.g., add delay).

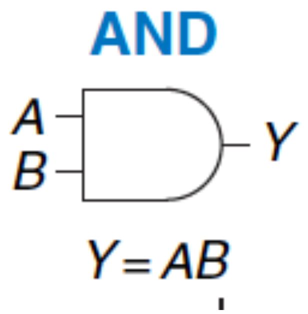


$Y = A$

A	Y
0	0
1	1

AND Gate

- Two-input gate.
- Output is TRUE *if and only if* both inputs are TRUE; FALSE otherwise.
- Typical notation is `Y = AB`, other notations are used such as `Y = A·B` or `Y = A∩B`

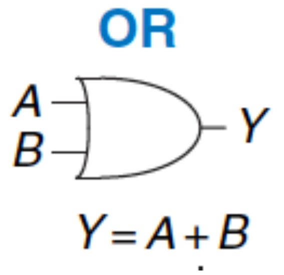


	A	B	Y
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

- By convention, input combinations in truth table are listed as 00,01,10,11

OR Gate

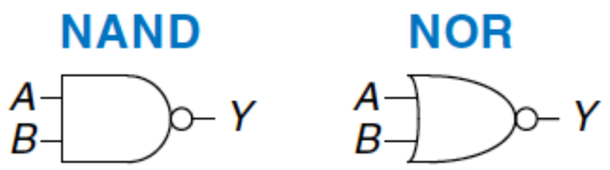
- Two-input gate
- OR Gate output is TRUE if either of the inputs (or both) are TRUE; otherwise, FALSE
- Typical notation is `Y = A+B`, read as "Y equals A or B"
- Other notations include `Y = A∪B`, read as "Y equals A union B"



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

NAND/NOR Gate

- A gate can be inverted by adding a bubble before the gate
- NAND gate is *inverted AND*
  - Output is FALSE *if and only if* both inputs are TRUE; otherwise, FALSE
- NOR gate is *inverted OR*
  - Output is TRUE *if and only if* both inputs are FALSE; otherwise, TRUE



$Y = \overline{AB}$

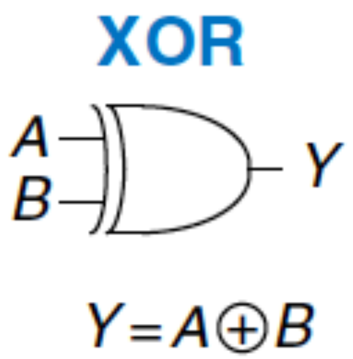
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$Y = \overline{A+B}$

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

XOR Gate

- Two-input gate
- Output is TRUE if and only if either inputs are TRUE (not both); otherwise, FALSE

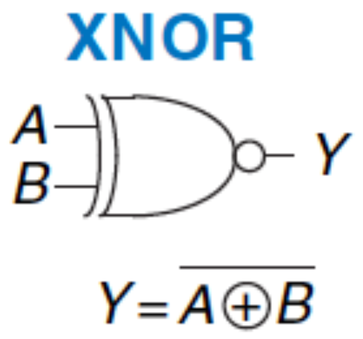


A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

- An N-input XOR gate can be used to check even/odd parity; TRUE for odd no. of inputs.

XNOR Gate

- Inverted XOR
- Output is TRUE if and only if inputs are the same; otherwise, FALSE



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

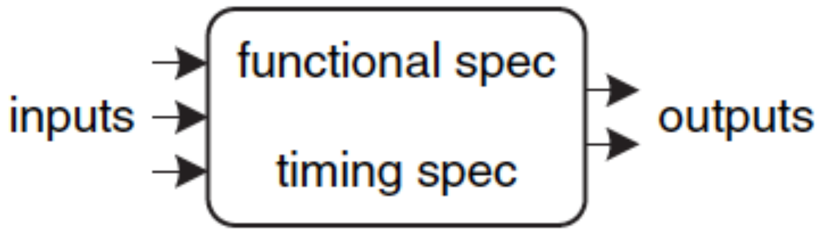
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Multiple Input Gates

- Many Boolean function (or gates) work with 3 or more inputs
- Common ones include AND, OR, XOR, NAND, NOR, and XNOR
- E.g., N-input AND produces TRUE when *all* inputs are TRUE
- E.g., N-input OR produces TRUE when *at least one* input is TRUE

Combinational Circuit Design

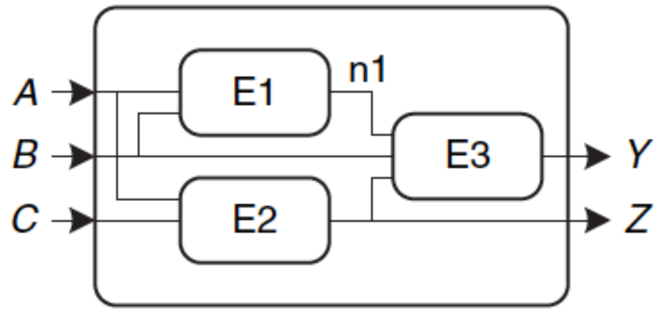
- A circuit is a network that processes *discrete-valued* variables
- You can view a circuit as a *black box* with the following elements:
    - *Input terminals*
    - *Output terminals*
    - A *functional specification* to define the relationship between inputs and outputs
    - A *timing specification* to define the delay between inputs changing and outputs responding



- A circuit can be
    - combinational – output is a function of input
    - sequential – output is a function of input and memory. More later.

Peering Into The Black Box

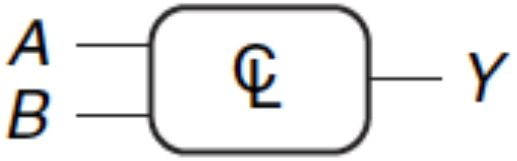
- The black box has
  - Elements – circuits with inputs, outputs, and specifications
  - Nodes – wire to convey *voltage* that indicates discrete-valued variable
  - Nodes are *input*, *output*, or internal



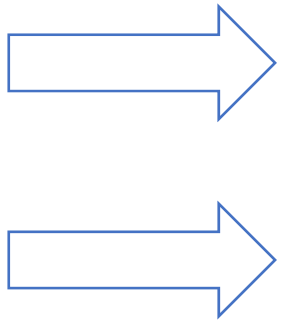
- The illustration has 3 elements – E1, E2, and E3
- Nodes A,B,C are inputs
- Nodes Y,Z are outputs
- n1 is an internal node

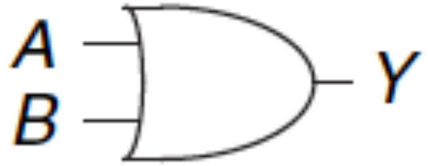
Specification vs Implementation

- Specification indicates ”what” is the circuit
- Implementation indicates “how” the circuit will be built

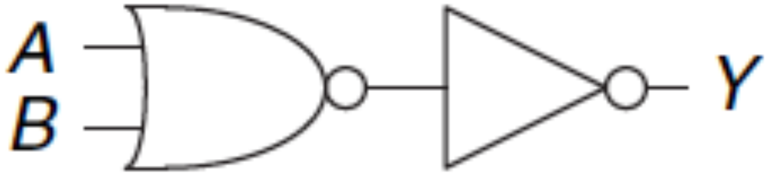


$$Y = F(A, B) = A + B$$





Implementation 1



Implementation 2

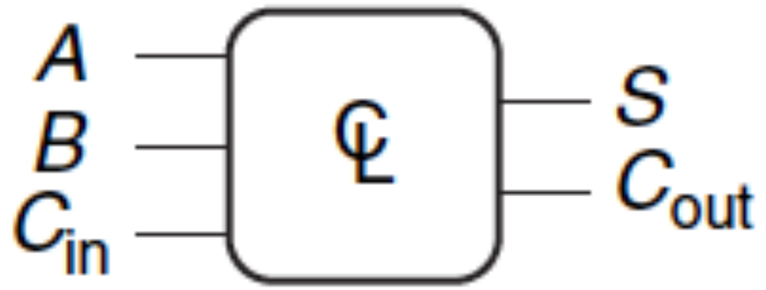
Specification
(CL indicates combinational circuit)

Rules of Combinational Composition

- In general, a circuit is combinational if
  - It is made of interconnected circuit elements
  - Every circuit is combinational
  - Every node is an input terminal, output terminal, or connected to the output terminal of an element
  - circuit has no cycles

Specification vs Implementation

- Consider another example – *full adder*
- The equations specify the function of outputs in terms of inputs



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

- More about implementation later

Boolean Equation

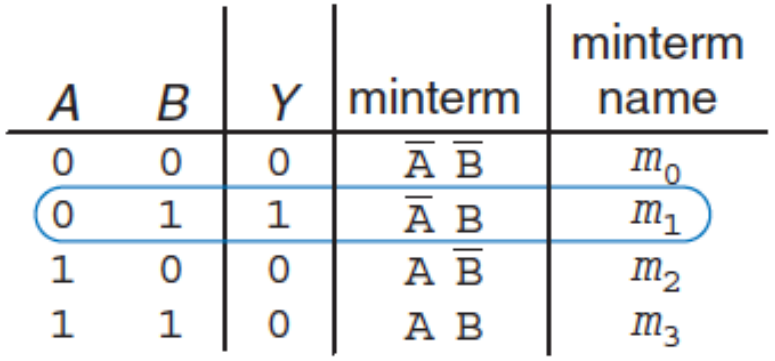
- Boolean equation involve variables that are either TRUE or FALSE
- They can be used to describe the relationship between inputs and outputs
- But, first let’s define a few terms

Terminology

- **Complement**: variable with a bar over it or a tick mark:
  - A', B', C'
- **Literal**: an *instance* of a variable or its complement present in an expression
  - A, A', B, B', C, C'
- **Product** of literals: $AB'C$
- **Sum** of literals: $A + B' + C$
- **Minterm**: a product that includes *all* input variables
  - For a 3-input circuit (inputs: *A*, *B*, *C*), $AB'C$, $A'BC'$ and ABC would be minterms, but AB' and BC would not be minterms
- **Maxterm**: a sum that includes *all* input variables
  - For a 3-input circuit, $A + B' + C$ and $A' + B + C'$ would be maxterms, but $A + B'$ and $B + C$ would not be maxterms

Sum-of-Products Form

- A truth table has 2^N rows for N inputs, one for each combination of inputs
- Each row is associated with a **minterm that is TRUE** for the row
- Minterms are numbered starting with 0
  - Minterm 0 is m_0, minterm 1 is m_1, and so on …

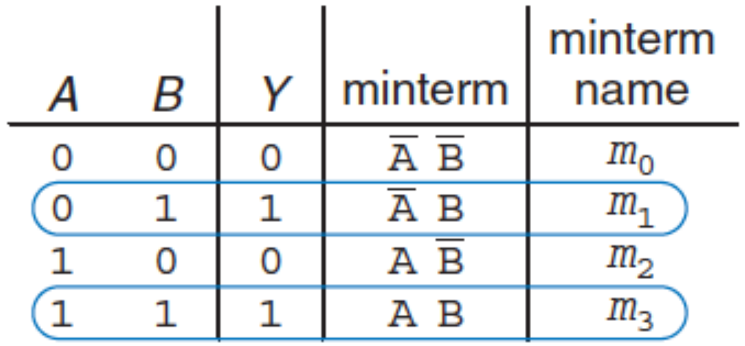


A	B	Y	minterm	minterm name
0	0	0	$\overline{A}\ \overline{B}$	m_0
0	1	1	$\overline{A}\ B$	m_1
1	0	0	$A\ \overline{B}$	m_2
1	1	0	$A\ B$	m_3

- A Boolean equation can be represented as the **sum of minterms that have output 1 or TRUE**
  - `Y = A’B`

Sum-of-Products Form

- This *sum of minterms* equation is called the **sum-of-products** form or **SOP** form.
- It is the sum (OR) of products (ANDs forming minterms).
- Generally, for the sake of consistency, if there is more than one minterm with output 1, we write them in order that they appear in the truth table.



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

$$Y = A'B + AB$$

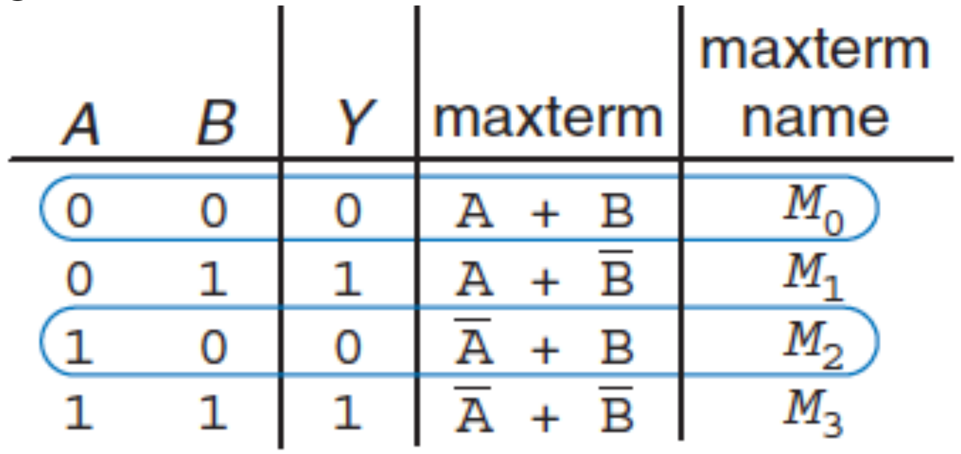
Notice the equation has **4 literals**

Sum-of-Products Form

- SOP form can also be written in **sigma notation**
- For example, `Y = A'B + AB` could also be written $F(A,B) = \Sigma(m_1, m_3)$ or $F(A,B) = \Sigma(1,3)$ or $F(A,B) = \Sigma m(1,3)$
- Note that you MUST list the input variables in the same order as the truth table: start with 0 and increment the count from there

Product-of-Sums Form

- Product-of-sum or **POS** is an alternative way to express Boolean equations
- Each row in the truth table corresponds to a **maxterm that is FALSE** for that row
- Boolean equation is expressed as the **product of maxterms with output FALSE or 0**



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

$$Y = (A + B)(A' + B)$$

- SOP and POS are equivalent. Why?
  - **De Morgan's Law**
  - More on that shortly!

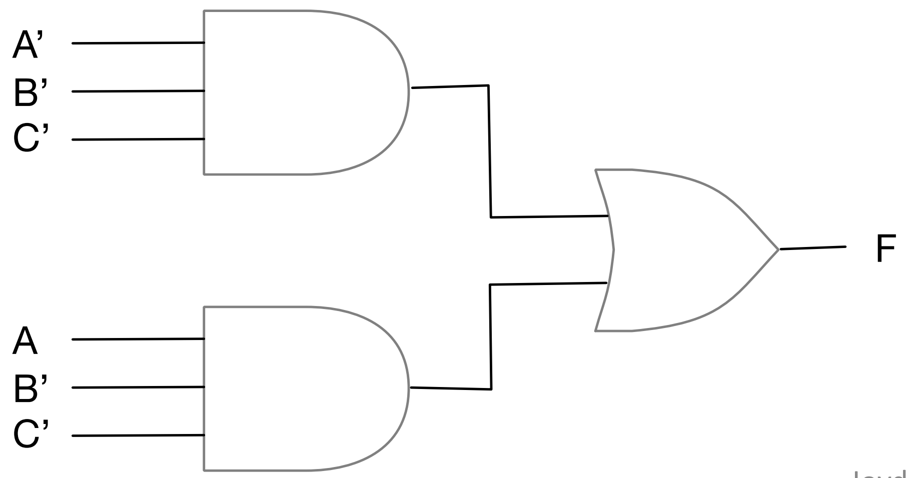
Product-of-Sums Form

- POS form can also be written in **pi notation**
- For example, `Y = (A + B)(A′ + B)` could also be written as $F(A,B) = \Pi(M_0, M_2)$ or $F(A,B) = \Pi(0,2)$ or $F(A,B) = \Pi M(0,2)$

Example: DivBy4

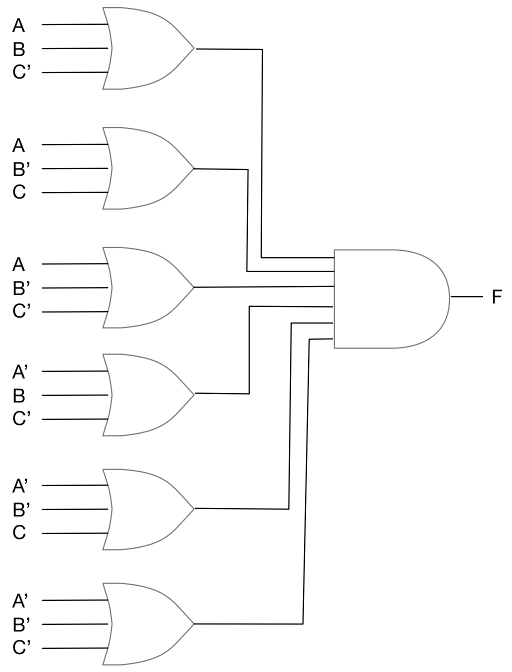
- Consider designing a circuit that returns TRUE if a 3-bit number is divisible by 4 and FALSE otherwise
- Write a Boolean equation in **SOP** form and draw its circuit diagram
- $\Sigma(m_0, m_4) = \mathrm{A'B'C'} + \mathrm{AB'C'}$
  - The formula has 6 literals

A	B	C	$F(A,B,C)$
0	**0**	**0**	**1**
0	0	1	0
0	1	0	0
0	1	1	0
1	**0**	**0**	**1**
1	0	1	0
1	1	0	0
1	1	1	0



Example: DivBy4

- Write a Boolean equation in **POS** form and draw its circuit diagram
- $\Pi(M_1, M_2, M_3, M_5, M_6, M_7)$
  $= (A + B + C')(A + B' + C)(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C')$



A	B	C	$F(A,B,C)$
0	0	0	1'
0	**0**	**1**	**0**
0	**1**	**0**	**0**
0	**1**	**1**	**0**
1	0	0	1
1	**0**	**1**	**0**
1	**1**	**0**	**0**
1	**1**	**1**	**0**

SOP vs POS

- They are **duals** of each other; equivalent ways of expressing the same function
- Generally, we use SOP if the truth table contains fewer 1s in output than 0s
- Similarly, we use POS if the truth table contains fewer 0s in output than 1s
- SOP and POS are called *two-level logic* because they use two gate levels : AND-OR or OR-AND
- A term is the minimal unit of a Boolean expression
- A term denotes the first-level gate from the perspective of a circuit
  - ABC has one term, but AB + BC has two terms
  - Likewise, (A+B+C) has one term but (A+C)(A+B'+C) has two terms

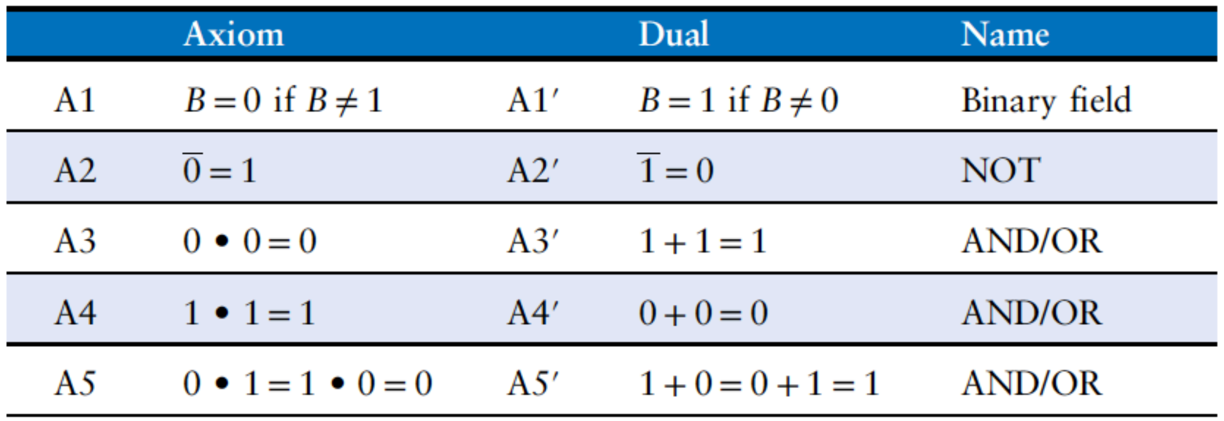
Minterm vs Maxterm

- A Boolean equation is *satisfiable* if at least one combination of inputs makes the output TRUE.
- SOP needs at least one term to make the equation *satisfiable*. Hence, its terms are called *minterms*
- POS needs all terms to make the equation *satisfiable*. Hence, its terms are *maxterms*.

Boolean Algebra

- So far, we have used truth tables to create Boolean expressions (SOP and POS)
- But such expressions may not lead to the simplest set of logic gates
- Boolean Algebra provides axioms and laws/theorems to simplify Boolean equations
    - Axioms are like definitions assumed to be true.
    - Theorems are derived and proved using axioms.

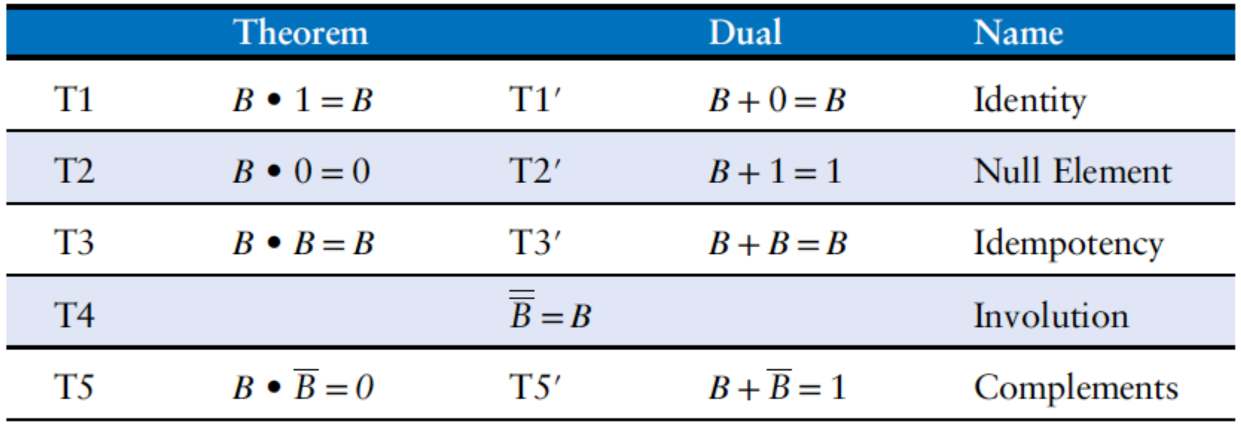
Axioms of Boolean Algebra



	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

- There are 5 axioms and their duals
- Axioms `A1` and `A1′` indicate that we are in a *binary field,* i.e., Boolean variable B is 0 when its not 1 and vice-versa
- Axioms `A2` and `A2′` indicate the NOT gate
- Axioms `A3`-`A5` denote the AND gate
- Axioms `A3′`-`A5′` denote the OR gate

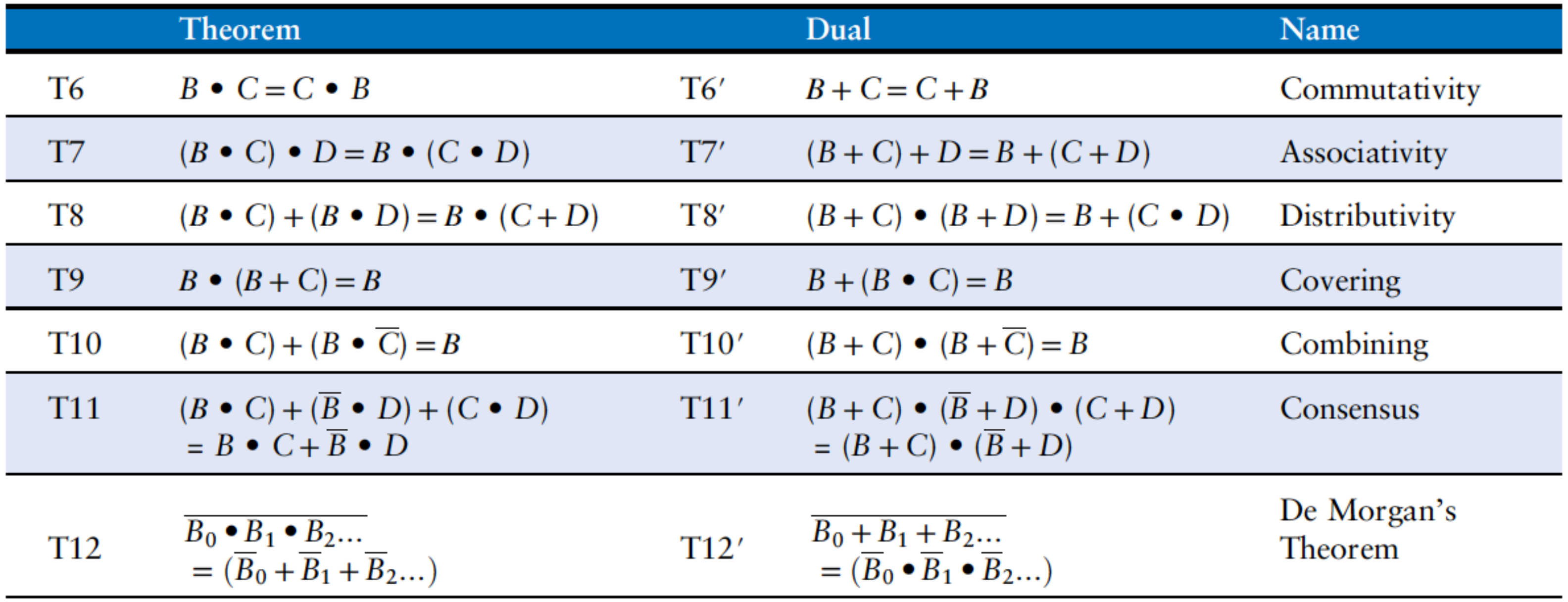
One Variable Theorems



	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

- **Identity**. The result of (any variable) `B AND 1` is always B; its dual states that `B OR 0` is always B
- **Null Element**. The result of ANDing any variable with 0 is always 0; its dual, ORing anything with 1 is always 1
- **Idempotency**. AND/ORing a variable B with itself gives back variable B
- **Involution**. Complementing a variable twice gives back the same variable
- **Complement**. A variable AND its complement results in 0; its dual OR results in 1

Several Variable Theorems



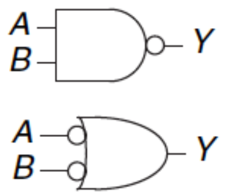
	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity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

- **Commutativity**. Like in algebra the order of the variables does not matter.
- **Associativity**. The variable groupings do not matter. Same as algebra.
- **Distributivity**. T8 shows that AND distributes over OR and its dual shows that OR distributes over AND. Notice T8' is not allowed in algebra.
- **Covering, Combining,** and **Consensus** are used to eliminate redundant variables.

De Morgan’s Theorem

- De Morgan’s Theorem is a significant result (named after Augustus De Morgan).
- From De Morgan, we can infer that:
  - A NAND gate is equivalent to an OR gate with inverted inputs!
  - A NOR gate is equivalent to an AND gate with inverted inputs!

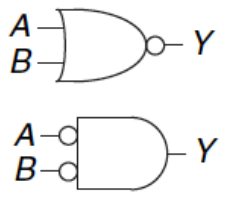
NAND



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

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

NOR

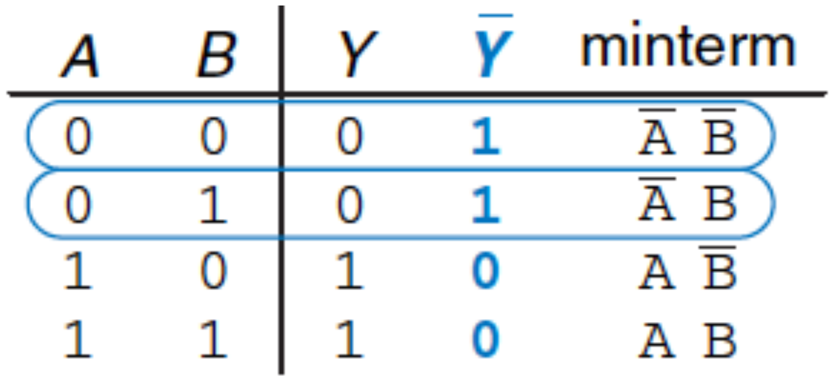


$Y = \overline{A+B} = \overline{A}\,\overline{B}$

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

De Morgan’s Theorem

- De Morgan’s shows that SOP and POS are equivalent
- Consider the truth table:



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

$Y' = A'B' + A'B$

From involution, $Y'' = Y$

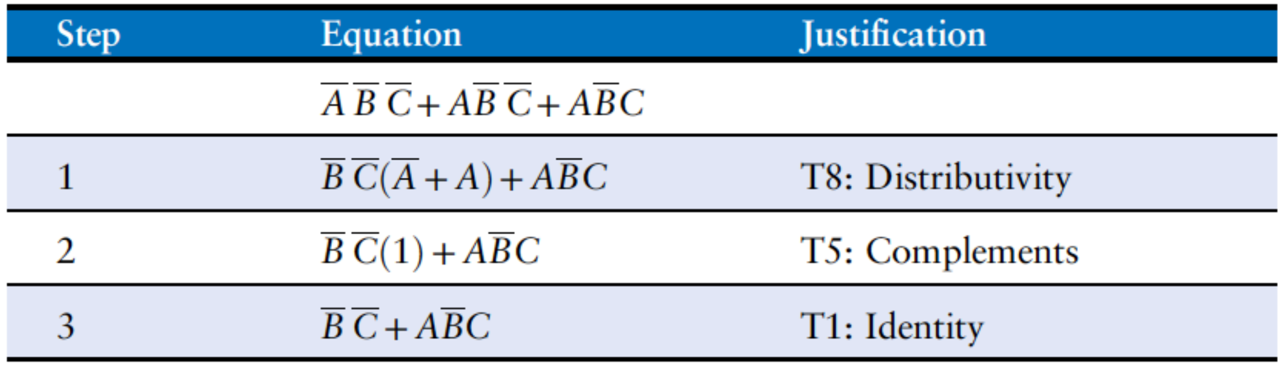
Hence, $Y'' = (A'B'+A'B)'$

$$\left.\begin{array}{l} => (A'B')' \ . \ (A'B)' \\ => (A+B).(A+B') \end{array}\right\} \textbf{De Morgan's}$$

Simplifying Equations – Prime Implicant Rule

- Given a Boolean equation in SOP, look for the form `PA + PA′` and then combine them; `P` is any **implicant**
- The idea is to obtain a **minimized equation** with *fewest possible implicants*
- If several *minimized* equations exists, pick the one with the fewest no. of literals
- A **prime implicant** is one which cannot be combined with other implicants to create a new implicant with fewer literals
- The implicants in a *minimal* equation are *prime implicants.*

Simplifying Equations – Prime Implicant Rule

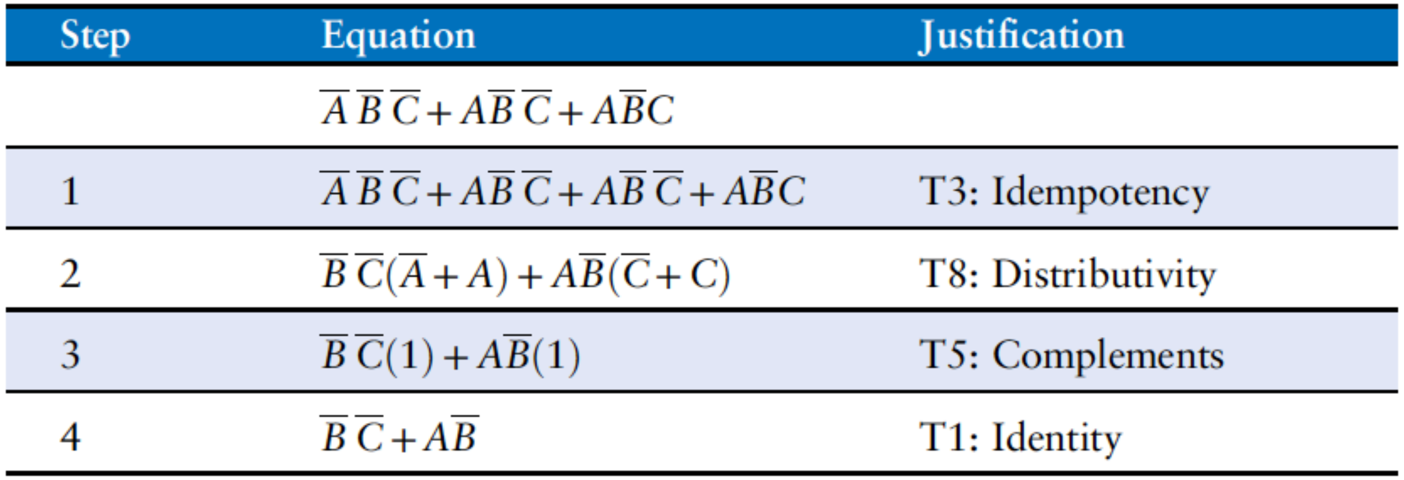


Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}C$	
1	$\overline{B}\,\overline{C}(\overline{A}+A)+A\overline{B}C$	T8: Distributivity
2	$\overline{B}\,\overline{C}(1)+A\overline{B}C$	T5: Complements
3	$\overline{B}\,\overline{C}+A\overline{B}C$	T1: Identity

- Notice that the pair of terms A′B′C′ + AB′C′ has the form PA + PA′
- We combine them to get B′C′
- We now have B′C′ + AB′C; both terms are prime implicants so further simplification by combining implicants is not possible!
- But we can further simplify using Boolean Algebra theorems

```
B′C′ + AB′C
= B′(C′ + AC)          (Distributivity)
= B′(C′ + A)(C′ + C)   (Distributivity)
= B′(C′ + A)(1)        (Identity)
= B′(C′ + A)
```

Simplifying Equations – Duplicating Term Rule



Step	Equation	Justification
	$\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}C$	
1	$\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}\,\overline{C}+A\overline{B}C$	T3: Idempotency
2	$\overline{B}\,\overline{C}(\overline{A}+A)+A\overline{B}(\overline{C}+C)$	T8: Distributivity
3	$\overline{B}\,\overline{C}(1)+A\overline{B}(1)$	T5: Complements
4	$\overline{B}\,\overline{C}+A\overline{B}$	T1: Identity

- But we can also pair **AB'**C' + **AB'**C ?
- We can combine them to get AB'
- Further, implicants B'C' and AB' share the minterm AB'C'
- When implicants share a minterm, the minterm can be **duplicated** for more simplification
- Thus, A'B'C' + **AB'C'** + AB'C = A'B'C' + **AB'C' + AB'C'** + AB'C
- Suddenly we have two instances of PA + PA'!
- For each, we use the previous method of combination to get B'C' + AB'
- Simplify further by Distributivity to gets fewer literals B'(A + C')

Simplifying Equations – Expand Term Rule

- Sometimes it helps to expand an equation to enable the implicant approach (e.g., turn `AB` into `ABC + ABC’`)
  - This works! But why? recall *identity* and *complement.*
- Example:

```
AB + A’C + BC
      => AB + A’C + ABC + A’BC
      => AB(1+C) + A’C(1+B)
      => AB + A’C
```

`=> AB + A’C + ABC + A’BC`	*Distributivity*
`=> AB(1+C) + A’C(1+B)`	*Distributivity*
`=> AB + A’C`	*Null Element*

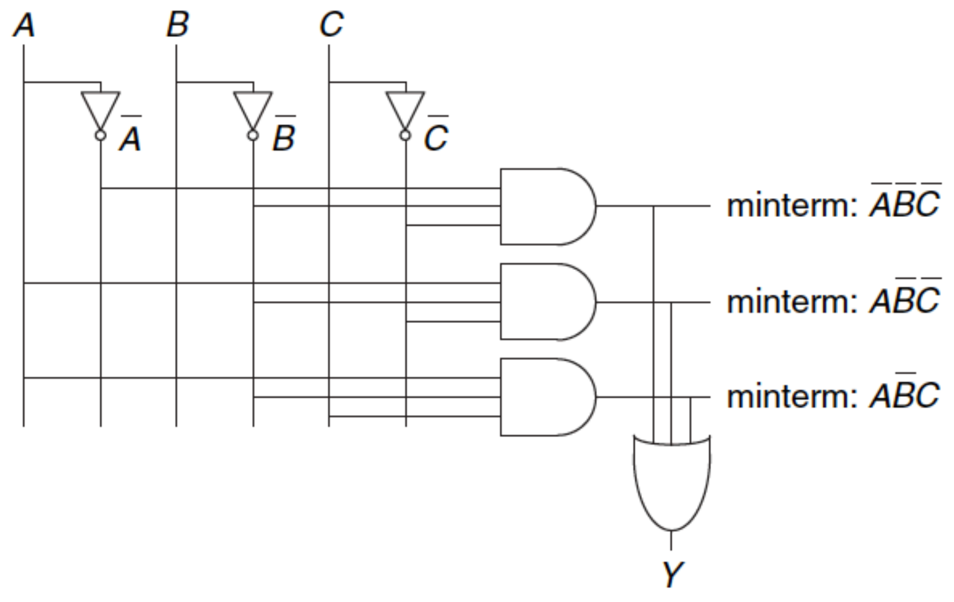
Simplifying Equations

- Let’s try another example
- Show that Y = A’BC + (BC’)’ + BC can be simplified to Y = B’ + C

A’BC + (BC’)’ + BC	
=> BC(A’+1) + (BC’)’	*Distributivity*
=> BC + (BC’)’	*Null Element*
=> BC + B’ + C	*De Morgan’s*
=> B’ + C(B+1)	*Distributivity*
=> B’ + C	*Null Element*

From Logic to Gates

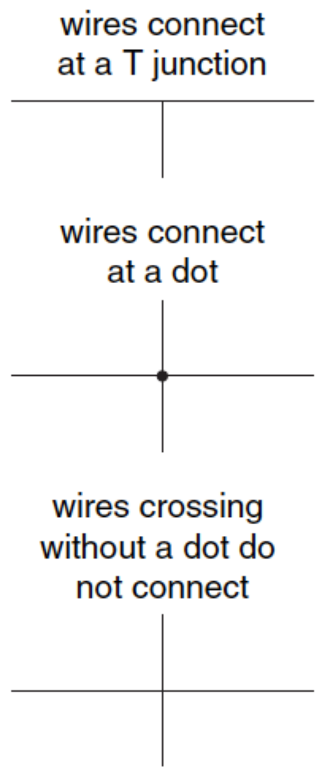
- Any Boolean equation can be represented as a *schematic*
  - A circuit diagram showing the elements and the wires that connect them
- Any SOP equation can be drawn systematically
  - Draw columns for inputs
  - Inverters in next column (if complement is necessary)
  - Draw AND gates for each minterm
  - Draw an OR gate that takes each output from the AND gates as inputs and has one output


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

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

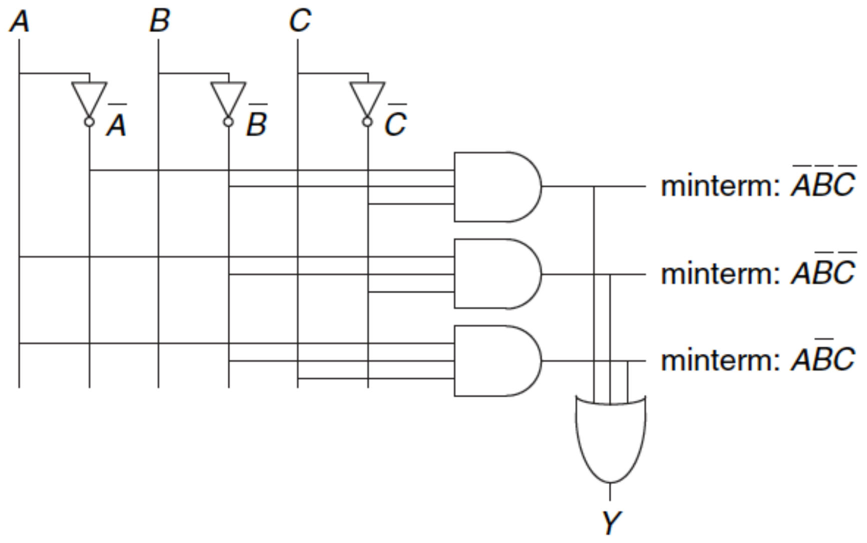
From Logic to Gates

- When drawing circuits, we follow certain conventions for consistency
  - Inputs on the left (top) side of the schematic
  - Outputs on the right (bottom) side of the schematic
  - Generally, gates flow from left to right
  - Use straight wires
  - Wires always connect at a T junction
  - A dot where wires connect indicates a connection
  - Wires crossing without a dot indicates no connection

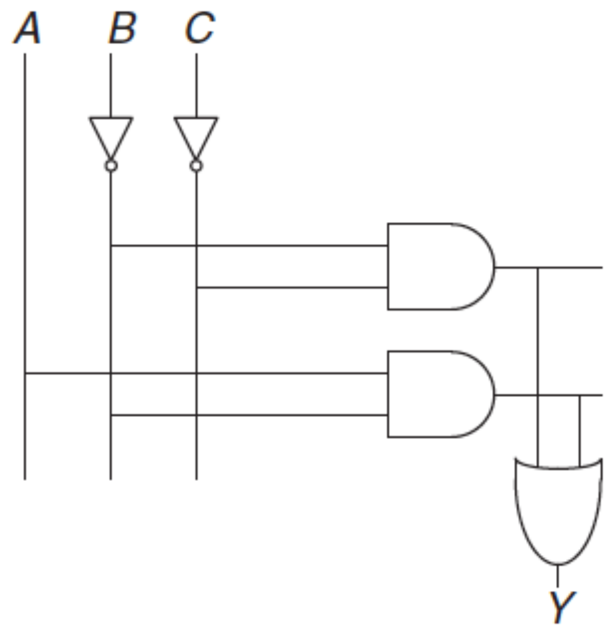

wires connect at a T junction
wires connect at a dot
wires crossing without a dot do not connect

From Logic to Gates

- Recall that `A′B′C′ + AB′C′+ AB′C = B′C′ + AB′`
- The simplified circuit will have less gates
  - Cheaper and faster!

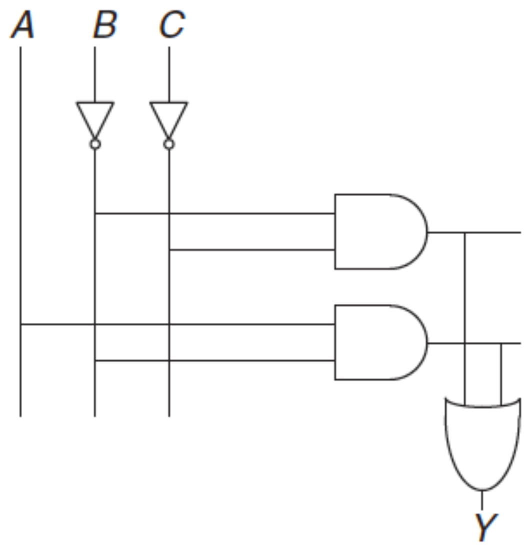

A
B
C
$\overline{A}$
$\overline{B}$
$\overline{C}$
minterm: $\overline{A}\overline{B}\overline{C}$
minterm: $A\overline{B}\overline{C}$
minterm: $A\overline{B}C$
Y

vs.

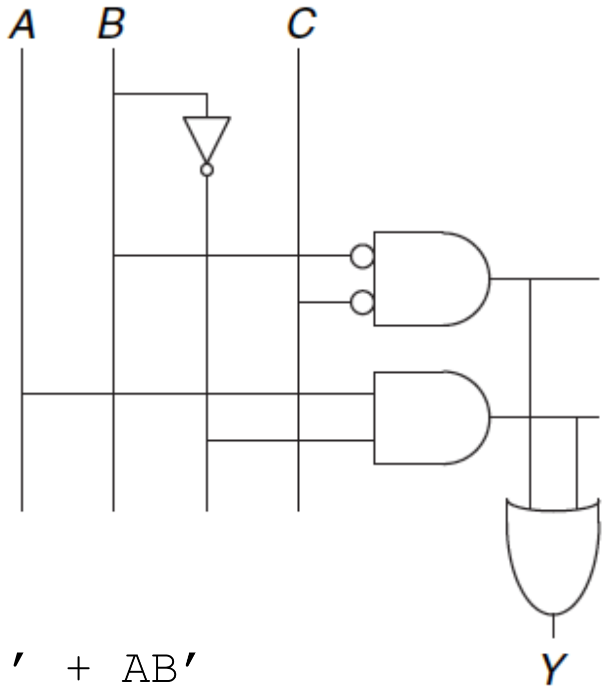


From Logic to Gates

- Can we do better?
  - Notice B’C’ is an AND gate with inverted inputs (same as NOR)
  - We can remove 1 inverter!



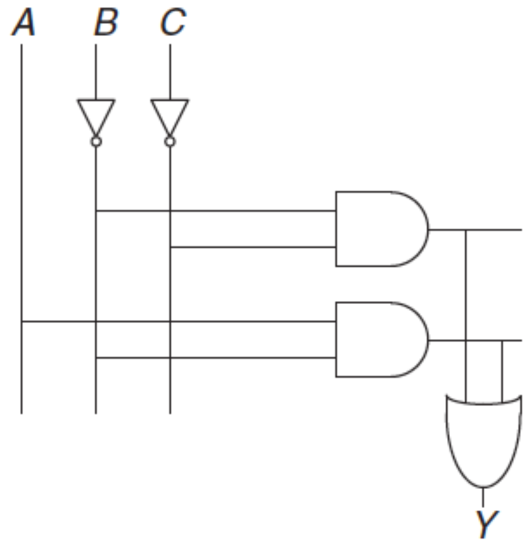
vs.



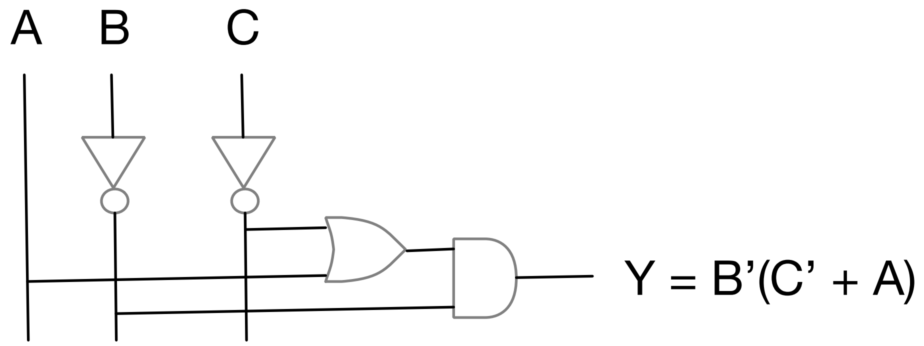
B′C′ + AB′ = (B+C)′ + AB′

From Logic to Gates

- An alternative simplified circuit!



vs.



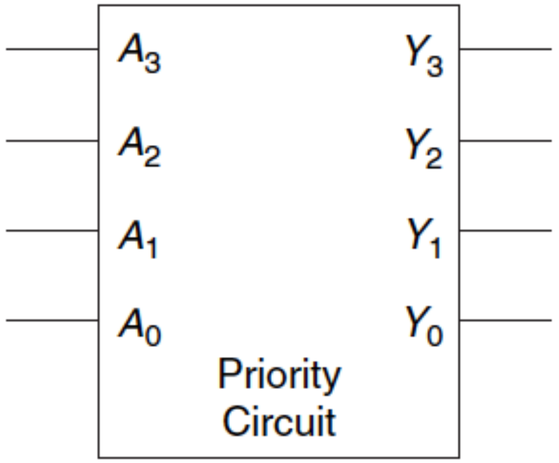
$B'C' + AB' = B'(C' + A)$

Multiple Output Circuits

- Circuits may have more than one output (e.g., *priority circuit*)
- Assume a circuit with 4 inputs (A_3, A_2, A_1, A_0) and 4 outputs (Y_3, Y_2, Y_1, Y_0)
  - A_3 has highest priority and A_0 lowest
  - Y_3 is enabled when A_3 is enabled
  - Y_2 is enabled when A_2 is enabled but A_3 is not
  - Y_1 is enabled when A_1 is enabled but A_3 and A_2 are not
  - Y_0 is enabled when ONLY A_0 is enabled
- We will write truth tables, Boolean equation, and sketch a circuit for this system

4-bit Priority Circuit

- Start with the truth table and the black box representation



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

4-bit Priority Circuit

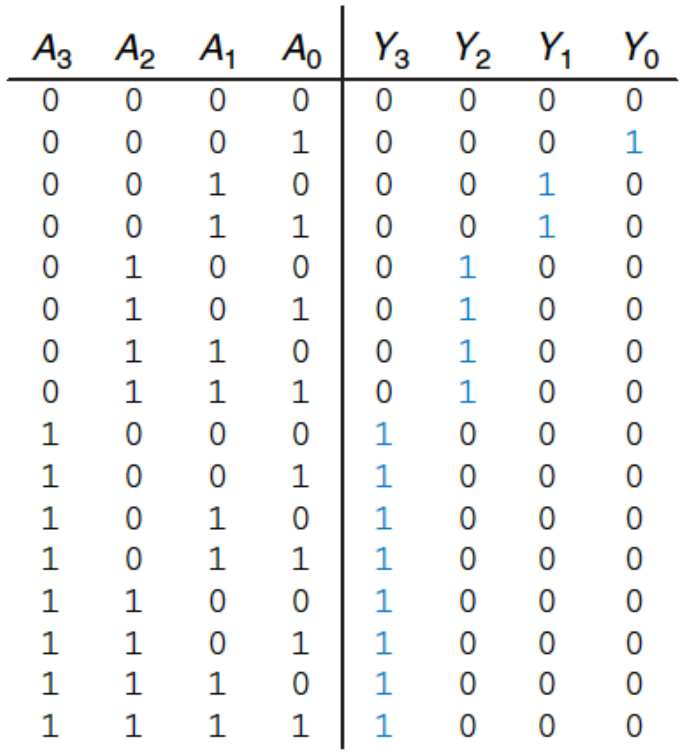
- Come up with Boolean equations.

$Y_3 = A_3$

$Y_2 = A_3'A_2$

$Y_1 = A_3'A_2'A_1$

$Y_0 = A_3'A_2'A_1'A_0$



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	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

4-bit Priority Circuit

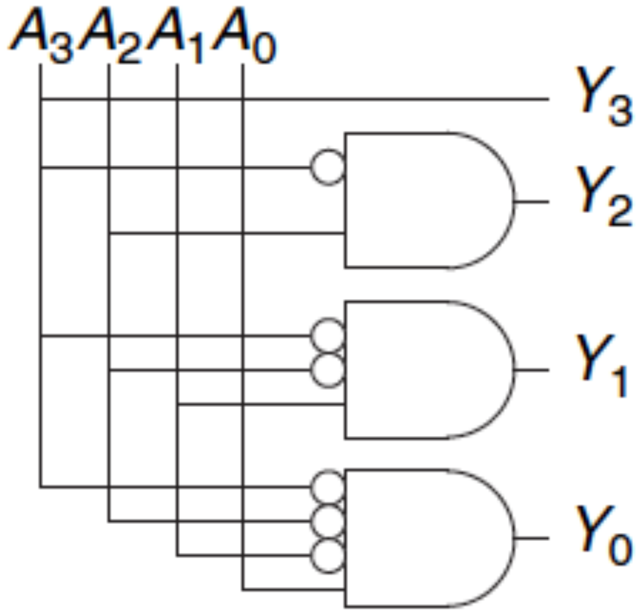
- Design the circuit.

$Y_3 = A_3$

$Y_2 = A_3'A_2$

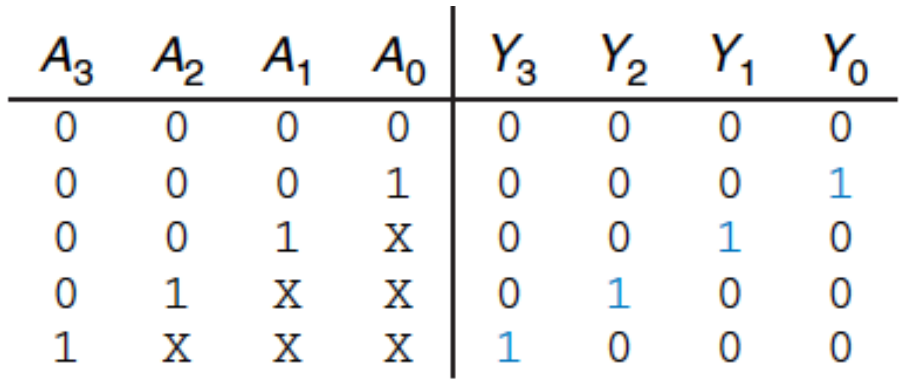
$Y_1 = A_3'A_2'A_1$

$Y_0 = A_3'A_2'A_1'A_0$



4-bit Priority Circuit

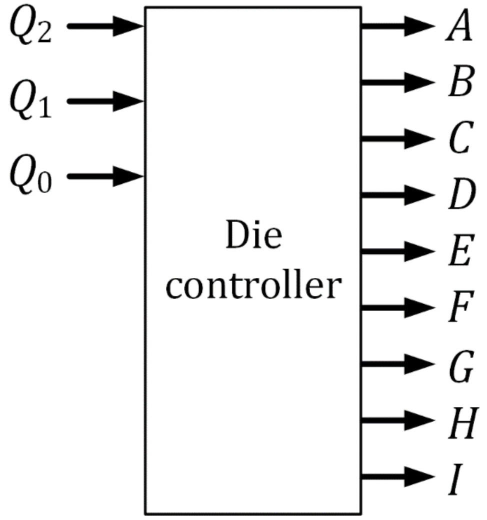
- Notice that if A_3 is enabled then we don't care about other inputs.
- This notion can be used to shorten the truth table.
- Don't cares are represented as X in a truth table.
- X indicates the value can be 0 or 1.
- Later we will see how this can be used to simplify Boolean equations.

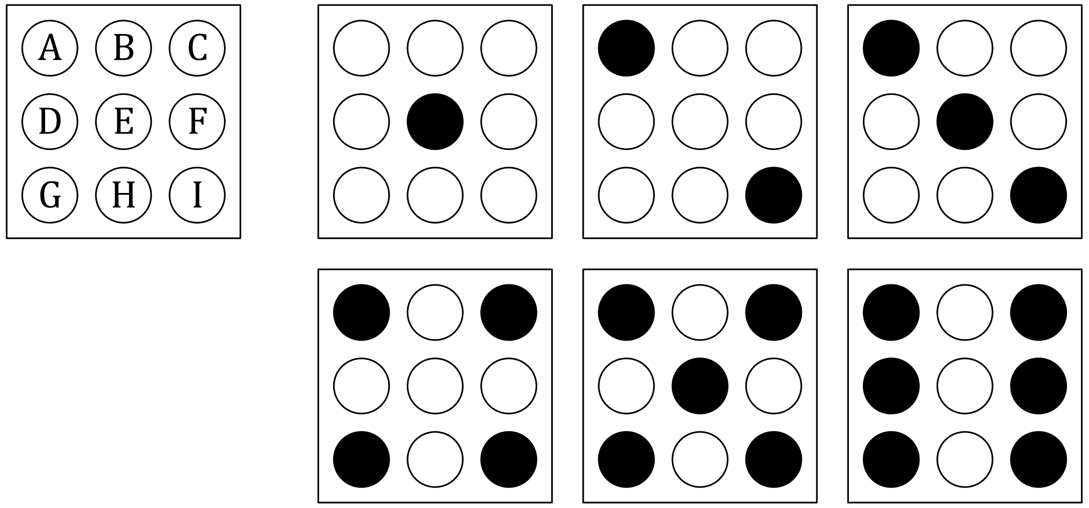


A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Electronic Die Example

- Let's design a circuit for an electronic die with the following specifications:
  - Inputs to the controller represent a 3-bit number (Q_2, Q_1, Q_0), which indicates the number to be displayed
  - There are nine outputs, *A* – *I*, one for each of the dots
  - Based on the input no., the output dots will light up.
  - If the value 0 or 7 is given, no dots should light

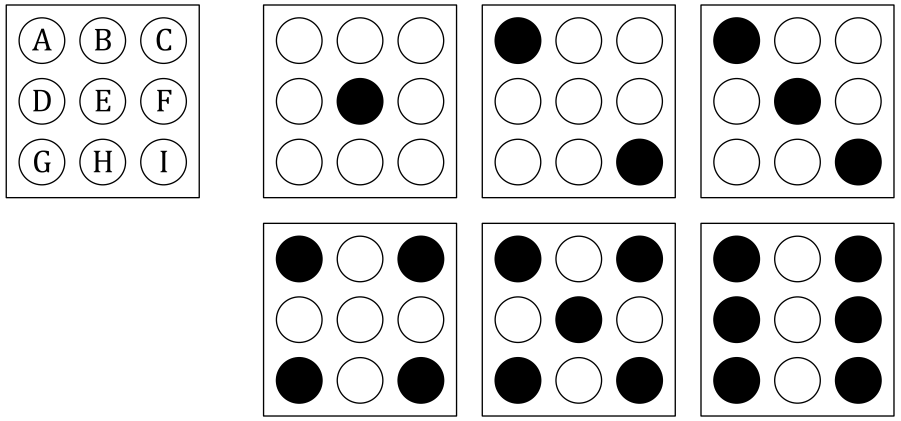




Electronic Die Example

- First step is to draw the truth table.

Q_2	Q_1	Q_0	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	0	0	0



Electronic Die Example

- Write the Boolean expression for each of the 9 outputs, using a minimal number of terms (SOP or POS)
- $A = I$
  $= (Q_2 + Q_1 + Q_0)(Q_2 + Q_1 + Q_0')$
  $(Q_2' + Q_1' + Q_0')$

$= ((Q_2 + Q_1) + Q_0Q_0')(Q_2' + Q_1' + Q_0')$

$= (Q_2 + Q_1)(Q_2' + Q_1' + Q_0')$

Q_2	Q_1	Q_0	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	0	0	0

Electronic Die Example

- Write the Boolean expression for each of the 9 outputs, using a minimal number of terms (SOP or POS)
- $B = H = 0$

Q_2	Q_1	Q_0	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	0	0	0

Electronic Die Example

- Write the Boolean expression for each of the 9 outputs, using a minimal number of terms (SOP or POS)
- $C = G = Q_2Q_1'Q_0' + Q_2Q_1'Q_0 + Q_2Q_1Q_0'$
  $= Q_2Q_1' + Q_2Q_1Q_0'$

Q_2	Q_1	Q_0	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	0	0	0

Electronic Die Example

- Write the Boolean expression for each of the 9 outputs, using a minimal number of terms (SOP or POS)
- $D = F = Q_2 Q_1 Q_0'$

Q_2	Q_1	Q_0	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	0	0	0

Electronic Die Example

- Write the Boolean expression for each of the 9 outputs, using a minimal number of terms (SOP or POS)
- $E = Q_2'Q_1'Q_0 + Q_2'Q_1Q_0 + Q_2Q_1'Q_0$

$$= Q_2'\,Q_1'Q_0 + Q_2'Q_1Q_0 + Q_2Q_1'Q_0 + Q_2'\,Q_1'Q_0$$

$$= Q_2'Q_0 + Q_2Q_1'Q_0 + Q_2'Q_1'Q_0$$

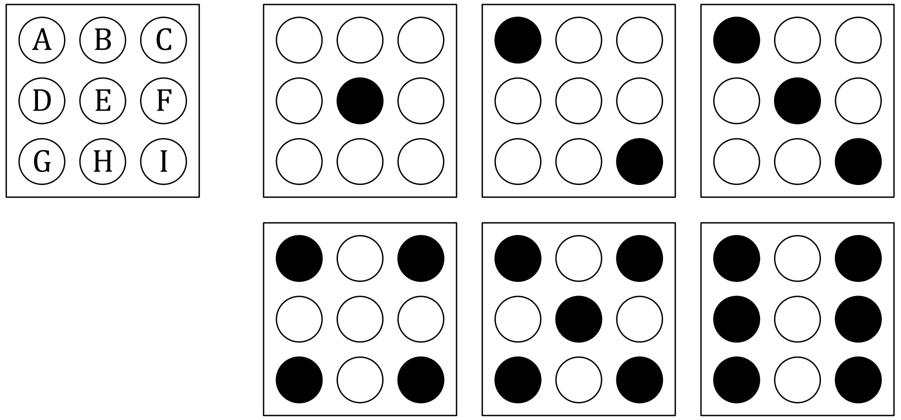
$$= Q_2'Q_0 + Q_1'Q_0$$

Q_2	Q_1	Q_0	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	0	0	0

Electronic Die Example

- Recall the electronic die problem.

Q_2	Q_1	Q_0	A	B	C	D	E	F	G	H	I
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	1
0	1	1	1	0	0	0	1	0	0	0	1
1	0	0	1	0	1	0	0	0	1	0	1
1	0	1	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	1	0	1	1	0	1
1	1	1	0	0	0	0	0	0	0	0	0

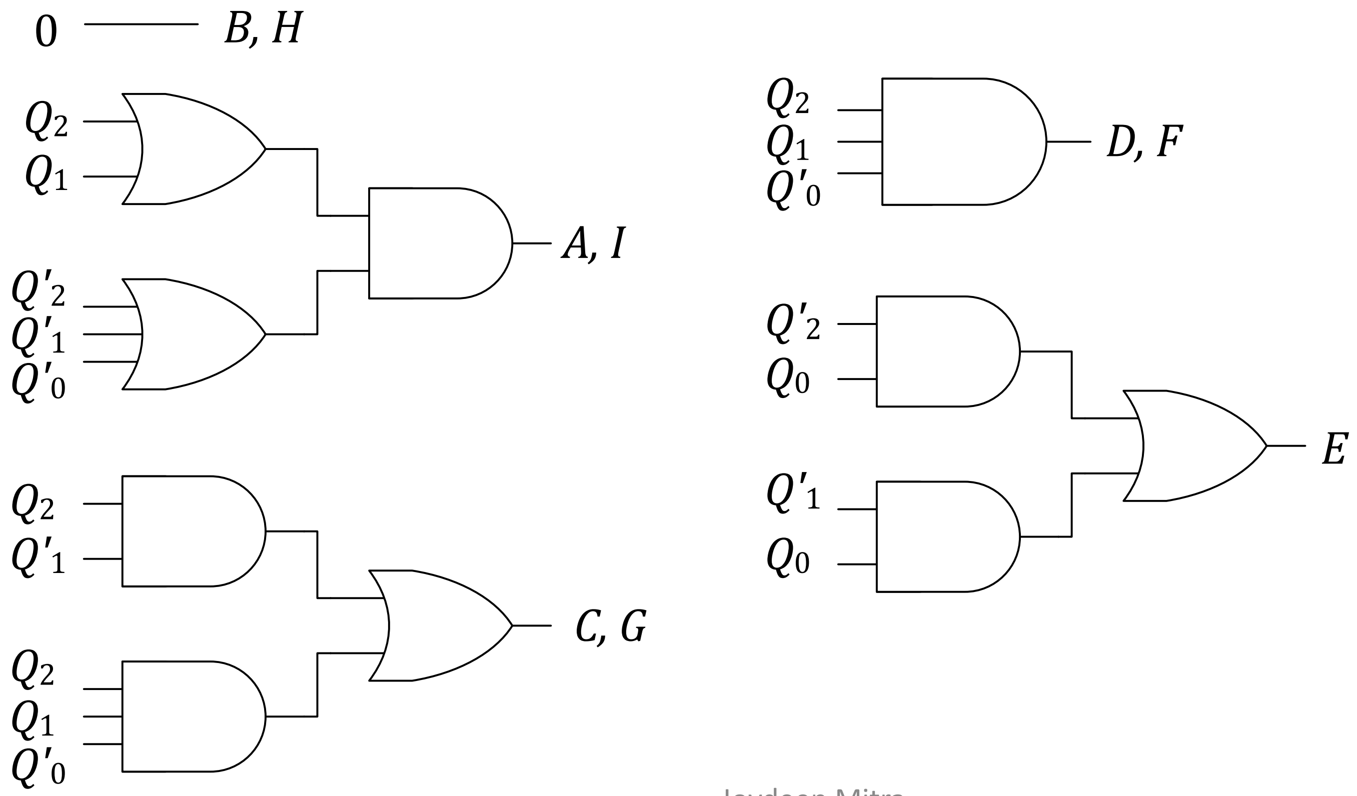


Electronic Die Example

- $A = I = (Q_2 + Q_1 + Q_0)(Q_2 + Q_1 + Q_0')(Q_2' + Q_1' + Q_0')$
  $= ((Q_2 + Q_1) + Q_0Q_0')(Q_2' + Q_1' + Q_0')$
  $= (Q_2 + Q_1)(Q_2' + Q_1' + Q_0')$
- $B = H = 0$
- $C = G = Q_2Q_1'Q_0' + Q_2Q_1'Q_0 + Q_2Q_1Q_0'$
  $= Q_2Q_1' + Q_2Q_1Q_0'$
- $D = F = Q_2Q_1Q_0'$
- $E = Q_2'Q_1'Q_0 + Q_2'Q_1Q_0 + Q_2Q_1'Q_0$
  $= Q_2'Q_0 + Q_2Q_1'Q_0$
  $= Q_2'Q_0 + Q_2Q_1'Q_0 + Q_2'Q_1'Q_0 = Q_2'Q_0 + Q_1'Q_0$

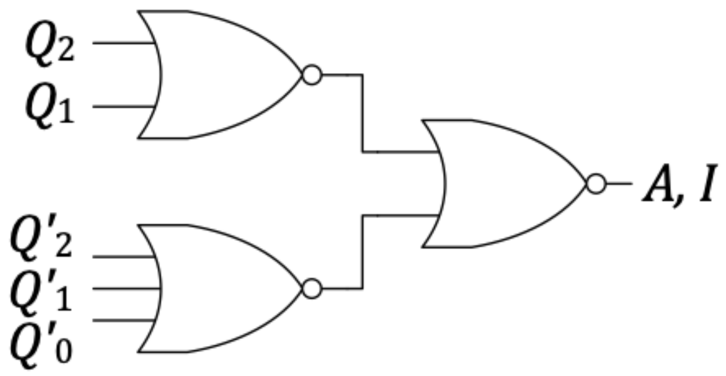
Electronic Die Example

- Circuits in AND-OR or OR-AND Networks



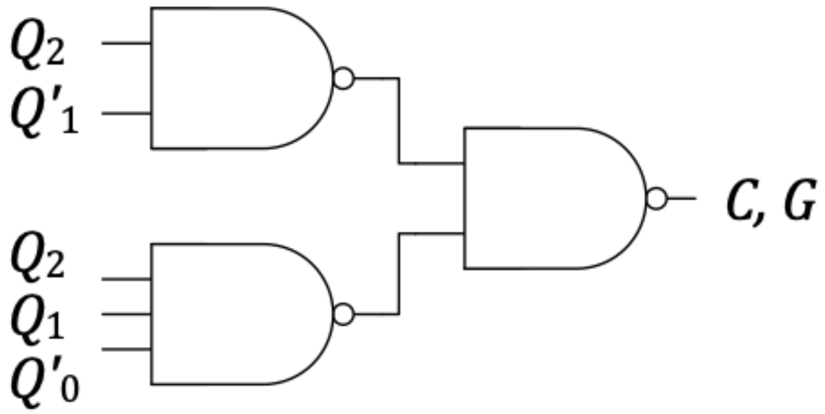
Electronic Die Example

- Can you implement SOP in NAND-NAND and POS in NOR-NOR?
- Recall, Y’’ = Y (involution) and DeMorgan’s.
- Let’s apply them to each output.
- $A = I = A'' = I'' = \big((Q_2 + Q_1)(Q_2' + Q_1' + Q_0')\big)''$
  $= ((Q_2 + Q1)' + (Q_2{}' + Q_1{}' + Q_0{}'))'$



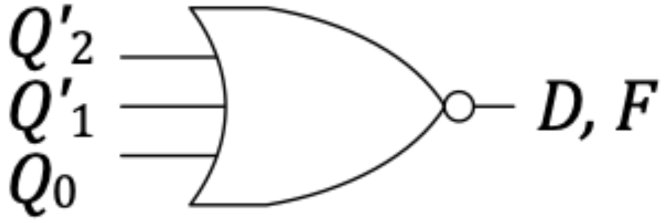
Electronic Die Example

- $C = G = C'' = G'' = (Q_2Q_1' + Q_2Q_1Q_0')''$
  $= ((Q_2Q_1')'.(Q_2Q_1Q_0')')'$

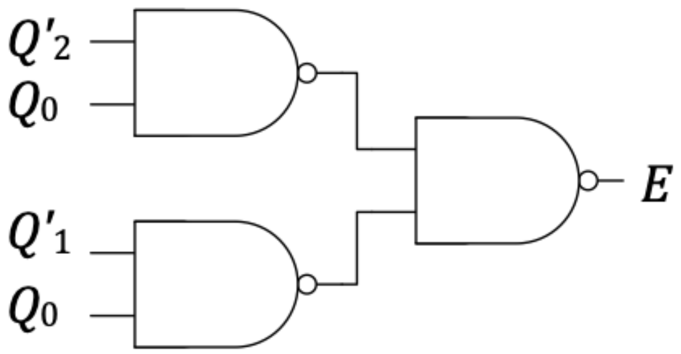


Electronic Die Example

- $D = F = D'' = F'' = (Q_2 Q_1 Q_0')'' = (Q_2' + Q1' + Q0)'$



- $E = E'' = (Q_2' Q_0 + Q_1' Q_0)'' = ((Q_2' Q_0)' . (Q_1' Q_0)')'$



Multiplier Example

- Let's design a black box with a decimal digit (0-9) encoded. in 4-bit binary as input (A_3,A_2,A_1,A_0).
- The output is the decimal digit multiplied by 3.
- The maximum possible output of the circuit is 27, so we need at least 5 bits for the output.
- We will draw a truth table first with 16 rows (4 inputs).
- But we don't care about value greater than 9!

A	$3A$
0	0
1	3
2	6
3	9
4	12
5	15
6	18
7	21
8	24
9	27

A3	A2	A1	A0	X3	X2	X1	X0	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	1	1	0
0	0	1	1	0	0	0	0	1	0	0	1
0	1	0	0	0	0	0	0	1	1	0	0
0	1	0	1	0	0	0	0	1	1	1	1
0	1	1	0	0	0	0	1	0	0	1	0
0	1	1	1	0	0	0	1	0	1	0	1
1	0	0	0	0	0	0	1	1	0	0	0
1	0	0	1	0	0	0	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X

Multiplier Example

- Let's derive the equation for X_0.
- Recall X's are don't cares (considered 0 or 1 as per convenience.)

Let's consider all X's as 0.

$$X_0 = A_3'A_2A_1{A_0}' + A_3'A_2A_1A_0 + A_3A_2'A_1'A_0' + A_3A_2'A_1'A_0$$

$$= A_3'A_2A_1(A_0+{A_0}') + A_3A_2'A_1'(A_0' + A_0)$$

$$= A_3'A_2A_1A_0 + A_3A_2'A_1'$$

A3	A2	A1	A0	X0
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
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

Multiplier Example

- Let’s consider all X’s as 1 instead.
    - Create SOP with minterms.
    - Duplicate the minterm $A_3A_2A_1A_0$
    - Apply the prime implicant rule to obtain $X_0 = A_3 + A_2A_1$
- Considering X’s as 1’s helped us simplify the equation further!

A3	A2	A1	A0	X0
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
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	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Multiplier Example

- Let's try another example.
- Derive the equation for X_1.
- Clearly X_1 is all 0s.
- We must set all X's to 0.
- We get $X_1 = 0$.

Takeaway:

- *It may be useful to have X's in a truth table.*
- *X's can be 0 or 1.*
- *They are 0 or 1 if it helps us simplify further.*

A3	A2	A1	A0	X1
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
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