COMBINATIONAL LOGIC DESIGN

Digital Systems

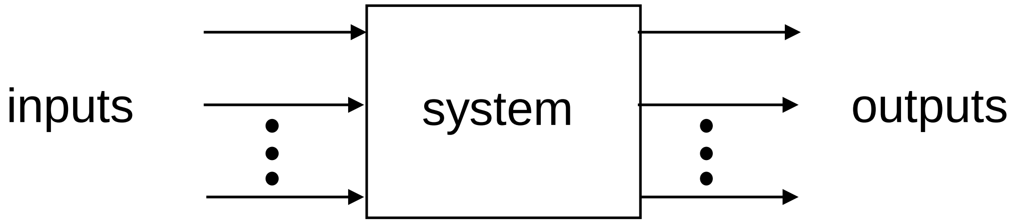
- Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage. All other voltage values are temporary and occur while transitioning between the values.

Digital Systems

- Low voltage is associated with 0s
- High voltage is associated with 1s
- The actual voltage values may differ from system to system

Logic Blocks and Block Diagrams

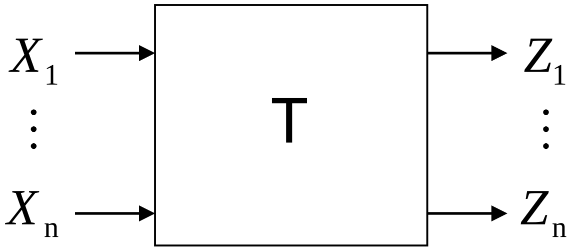
- Computational "blocks" perform a set of logical functions in either a <u>combinational</u> or <u>sequential</u> fashion.

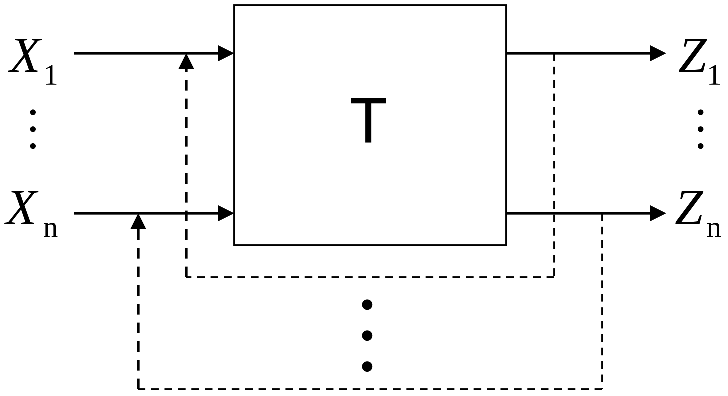


- A block diagram is a simple model of these systems that shows only the inputs and outputs.

Combinational vs. Sequential

- Combinational:
  - No Feedback
  - Output defined completely in terms of the Inputs.
- Sequential:
  - With feedback
  - System goes through different states
  - New state depends on Inputs and current state.





Combinational logic

- Truth Tables, Logic Equations, and Gates
  - NOT, AND, OR, NAND, NOR, XOR, . . .
  - Minimal set
- Axioms and theorems of Boolean algebra
  - Proofs by re-writing
  - Proofs by perfect induction
- Gate logic
  - Networks of Boolean functions
  - Time behavior
- Canonical forms
  - Two-level
  - Incompletely specified functions
- Simplification
  - Boolean cubes and Karnaugh maps
  - Two-level simplification

Truth Tables

- Combinational logic blocks can be completely specified by defining the output values for each possible set of input values.
- This is done using a truth table.
- For a logic block with *n* inputs, there are 2^n entries in the truth table. Each entry specifies the value of all the outputs for that particular input combination.

Possible Logic Functions

- If there are 2 input variables, there should be $2^2 = 4$ entries in the truth table.

- How many different functions of 2 input variables can we make?

Possible Logic Functions

X	Y	16 possible functions (F0–F15)															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		0	X and Y		X		Y	X xor Y	X or Y	X nor Y not (X or Y)	X xnor Y	not Y		not X		X nand Y not (X and Y)	1

- There are 16 possible functions of 2 input variables
- In general, there are 2^{2^n} functions of n inputs

Truth Tables

- Consider a logic function with three inputs, *A*, *B*, and *C*, and three outputs, *D*, *E*, and *F*. The function is defined as follows:
  - *D* is true if at least one input is true
  - *E* is true if exactly two inputs are true
  - *F* is true only if all three inputs are true.

Truth Tables

- Consider a logic function with three inputs, *A*, *B*, and *C*, and three outputs, *D*, *E*, and *F*. The function is defined as follows:
  - *D* is true if at least one input is true
  - *E* is true if exactly two inputs are true
  - *F* is true only if all three inputs are true.

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Truth Tables

- Consider a logic function with three inputs, *A*, *B*, and *C*, and three outputs, *D*, *E*, and *F*. The function is defined as follows:
  - *D* is true if at least one input is true
  - *E* is true if exactly two inputs are true
  - *F* is true only if all three inputs are true.

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0		
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	1		

Truth Tables

- Consider a logic function with three inputs, *A*, *B*, and *C*, and three outputs, *D*, *E*, and *F*. The function is defined as follows:
  - *D* is true if at least one input is true
  - *E* is true if exactly two inputs are true
  - *F* is true only if all three inputs are true.

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	1	1	
1	0	0	1	0	
1	0	1	1	1	
1	1	0	1	1	
1	1	1	1	0	

Truth Tables

- Consider a logic function with three inputs, *A*, *B*, and *C*, and three outputs, *D*, *E*, and *F*. The function is defined as follows:
  - *D* is true if at least one input is true
  - *E* is true if exactly two inputs are true
  - *F* is true only if all three inputs are true.

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Boolean Algebra

- Deals with a set of <u>variables</u> (<u>operands</u>) combined with a set of <u>operators</u>.
  - Variables denoted by X,Y,Z, etc.
  - Variables take binary values:

    Either “0” or “1” (“false” or “true”)

- Operators: **NOT**, **AND**, **OR**

- All logical operations can be described using these three operators.

OR

- The OR operator is written as +, as in $A + B$.

A	B	A + B (OR)
0	0	0
0	1	1
1	0	1
1	1	1

- The OR operation is also called a *logical sum.*

AND

- The AND operator is written as *, as in *A* * *B*.

A	B	A * B (AND)
0	0	0
0	1	0
1	0	0
1	1	1

- The AND operation is also called a *logical product.*

NOT

- The unary operator NOT is written as *A'*.

A	A'
0	1
1	0

Axioms and theorems of Boolean algebra

- Identity
  - X + 0 = X
  - X • 1 = X
- Null
  - X + 1 = 1
  - X • 0 = 0
- Idempotency:
  - X + X = X
  - X • X = X
- Involution:
  - (X')' = X
- Inverse:
  - X + X' = 1
  - X • X' = 0
- Commutative:
  - X + Y = Y + X
  - X • Y = Y • X
- Associativity:
  - (X + Y) + Z = X + (Y + Z)
  - (X • Y) • Z = X • (Y • Z)

Axioms and theorems of Boolean algebra

- Distributivite:
  - $X \bullet (Y + Z) = (X \bullet Y) + (X \bullet Z)$
  - $X + (Y \bullet Z) = (X + Y) \bullet (X + Z)$
- Uniting:
  - $X \bullet Y + X \bullet Y' = X$
  - $(X + Y) \bullet (X + Y') = X$
- Absorption:
  - $X + X \bullet Y = X$
  - $X \bullet (X + Y) = X$

  - $(X + Y') \bullet Y = X \bullet Y$
  - $(X \bullet Y') + Y = X + Y$

Axioms and theorems of Boolean algebra

- Factoring:
  - (X + Y) • (X' + Z) =X • Z + X' • Y
  - X • Y + X' • Z = (X + Z) • (X' + Y)

- Consensus:
- (X • Y) + (Y • Z) + (X' • Z) = X • Y + X' • Z
- (X + Y) • (Y + Z) • (X' + Z) = (X + Y) • (X' + Z)

- de Morgan's:
  - (X + Y + ...)' = X' • Y' • ...
  - (X • Y • ...)' = X' + Y' + ...

Logic functions and Boolean algebra

- Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ', +, and •

X	Y	X • Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X'	X' • Y
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

X	Y	X'	Y'	X • Y	X' • Y'	(X • Y) + (X' • Y')
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

$$(X \cdot Y) + (X' \cdot Y') = \overline{X \oplus Y}$$

Boolean expression that is true when the variables X and Y have the same value and false, otherwise

X, Y are Boolean algebra variables

Logic functions and Boolean algebra

- *D* is true if at least one input is true
- *E* is true if exactly two inputs are true
- *F* is true only if all three inputs are true.

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- D = A+B+C
- F = A*B*C
- E = ((A * B) + (A * C) + (B * C)) * (A * B * C)’
- E = (A * B * C’) + (A * C * B’) + (B * C * A’)

Proving theorems with Perfect Induction

- Using perfect induction (complete truth table):
  - e.g., de Morgan's:

$(X + Y)' = X' \cdot Y'$

NOR is equivalent to AND with inputs complemented

X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

$(X \cdot Y)' = X' + Y'$

NAND is equivalent to OR with inputs complemented

X	Y	X'	Y'	(X • Y)'	X' + Y'
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

Proving theorems with Perfect Induction

- E = ((A * B) + (A * C) + (B * C)) * (A * B * C)’
- E = (A * B * C’) + (A * C * B’) + (B * C * A’)

Inputs			E		
A	**B**	**C**	**E**	((A * B) + (A * C) + (B * C)) * (A * B * C)’	(A * B * C’) + (A * C * B’) + (B * C * A’)
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	0	0	0

Proving theorems with Rewriting

- Using the axioms of Boolean algebra:
  - e.g., prove the theorem: $X \bullet Y + X \bullet Y' = X$

Distributive law	$X \cdot Y + X \cdot Y'$	$=$	$X \cdot (Y + Y')$
Inverse law	$X \cdot (Y + Y')$	$=$	$X \cdot (1)$
Identity law	$X \cdot (1)$	$=$	X ✓

  - e.g., prove the theorem: $X + X \bullet Y = X$

Identity law	$X + X \cdot Y$	$=$	$X \cdot 1 + X \cdot Y$
Distributive law	$X \cdot 1 + X \cdot Y$	$=$	$X \cdot (1 + Y)$
Null law	$X \cdot (1 + Y)$	$=$	$X \cdot (1)$
Identity law	$X \cdot (1)$	$=$	X ✓

Proving theorems with Rewriting

- E = ((A * B) + (A * C) + (B * C)) * (A * B * C)’
- E = (A * B * C’) + (A * C * B’) + (B * C * A’)

((A * B) + (A * C) + (B * C)) * (A * B * C)’

= ((*A* * *B*) + (*A* * *C*) + (*B* * *C*)) * (A’ + B’ + C’)

DeMorgan’s law

= (A’ + B’ + C’)(A*B) + (A’ + B’ + C’)(A*C) + (A’ + B’ + C’)(B*C)

Distributive law

= (A*B*A’) + (A*B*B’) + (A*B*C’) + (A*C*A’) + (A*C*B’) + (A*C*C’) + (B*C*A’)+(B*C*B’)+(B*C*C’)

Distributive law

= (0*B)+(0*A)+(A*B*C’) + (0*C)+(A*C*B’)+(A*0) + (B*C*A’)+(C*0)+(B*0)

Inverse law

= 0 + 0 + (A*B*C’) + 0 +(A*C*B’) + 0 + (B*C*A’) + 0 +0

Null law

= $(A \cdot B \cdot C') + (A \cdot C \cdot B') + (B \cdot C \cdot A')$

Identity law

Adder, Part 1

- 1-bit binary adder
  - inputs: A, B, Carry-in
  - outputs: Sum, Carry-out

A	B	Cin	Σ	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\Sigma = A'\ B'\ Cin + A'\ B\ Cin' + A\ B'\ Cin' + A\ B\ Cin$$

$$Cout = A'\ B\ Cin + A\ B'\ Cin + A\ B\ Cin' + A\ B\ Cin$$

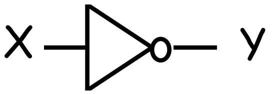
Apply the theorems to simplify expressions

- The theorems of Boolean algebra can simplify Boolean expressions
  - The Cout function is used as an example here, but the same rules apply to any function.

Cout	=	A' B Cin + A B' Cin + A B Cin' + A B Cin
(Idempotency)	=	A' B Cin + A B' Cin + A B Cin' + A B Cin + A B Cin
(Commutative)	=	A' B Cin + A B Cin + A B' Cin + A B Cin' + A B Cin
(Distributive)	=	(A' + A) B Cin + A B' Cin + A B Cin' + A B Cin
(Inverse)	=	(1) B Cin + A B' Cin + A B Cin' + A B Cin
(Idempotency)	=	B Cin + A B' Cin + A B Cin' + A B Cin + A B Cin
(Commutative)	=	B Cin + A B' Cin + A B Cin + A B Cin' + A B Cin
(Distributive)	=	B Cin + A (B' + B) Cin + A B Cin' + A B Cin
(Inverse)	=	B Cin + A (1) Cin + A B Cin' + A B Cin
(Distrivutive)	=	B Cin + A Cin + A B (Cin' + Cin)
(Inverse)	=	B Cin + A Cin + A B (1)
	=	B Cin + A Cin + A B

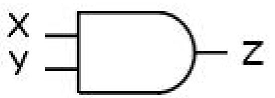
Logic Gates

- NOT X' $\overline{X}$ ~X



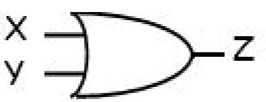
x	y
0	1
1	0

- AND $X \bullet Y$ $X \wedge Y$



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

- OR $X + Y$ $X \vee Y$



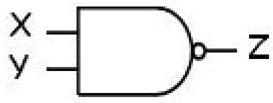
x	y	z
0	0	0
0	1	1
1	0	1
1	1	1

Logic Gates and Inverters

- Rather than draw inverters explicitly, a common practice is to add “bubbles” to the inputs or outputs of a gate to cause the logic value on that input line or output line to be inverted.

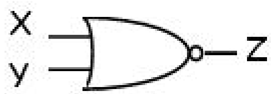
Logic Gates

- NAND



X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- NOR



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	0

- XOR

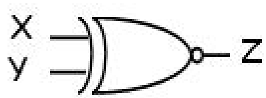
  $X \oplus Y$

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

X xor Y = X Y' + X' Y
X or Y but not both
("inequality", "difference")

- XNOR

  $\overline{X \oplus Y}$



X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

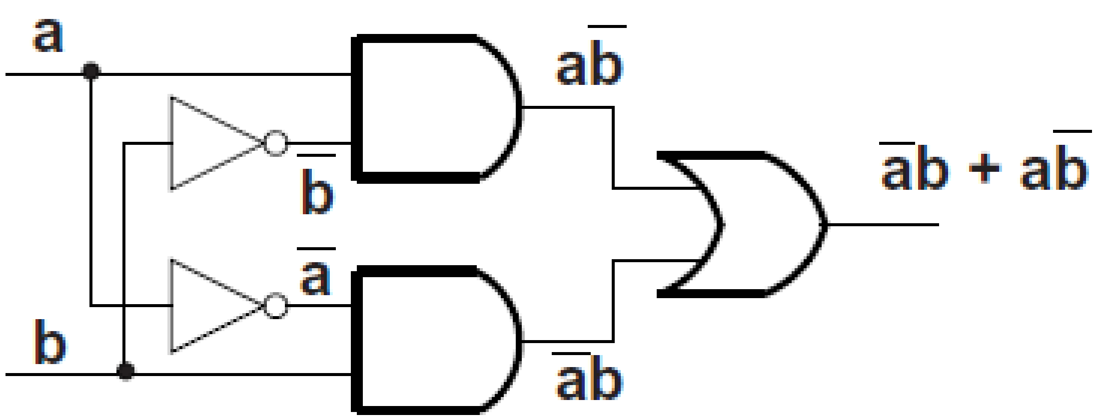
X xnor Y = X Y + X' Y'
X and Y are the same
("equality", "coincidence")

Adder, Part 2

- Logic gates for Sum (shown without Cin)

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	0

$$Sum = A'\,B + A\,B'$$

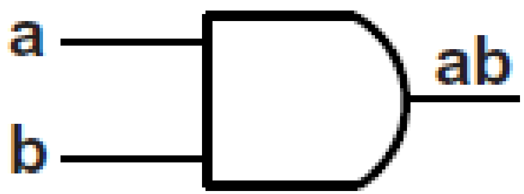


Adder, Part 2

- Logic gates for Cout (shown without Cin)

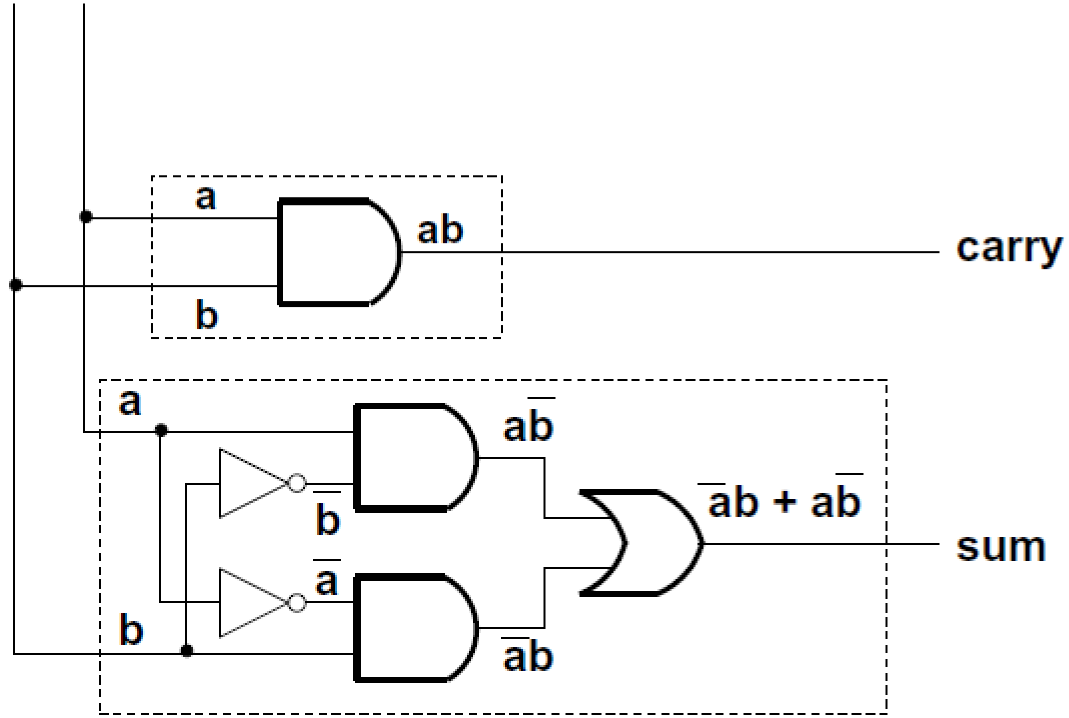
A	B	Cout
0	0	0
1	0	0
0	1	0
1	1	1

Cout = A B



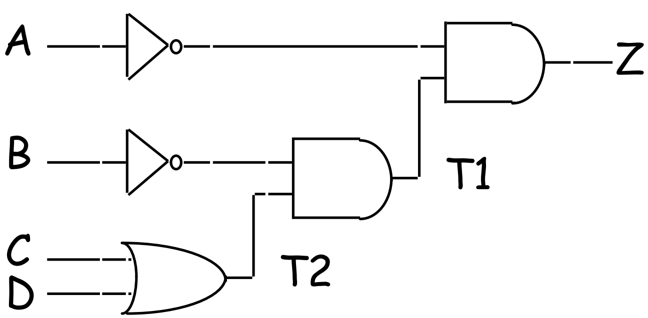
Adder, Part 2

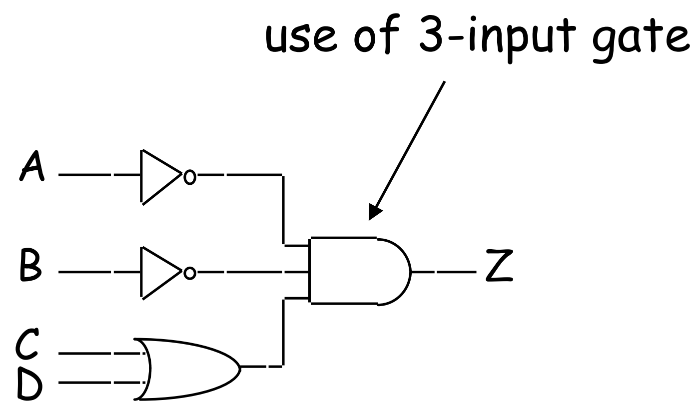
- Logic gates for Sum and Cout (shown without Cin)



Logic Gates

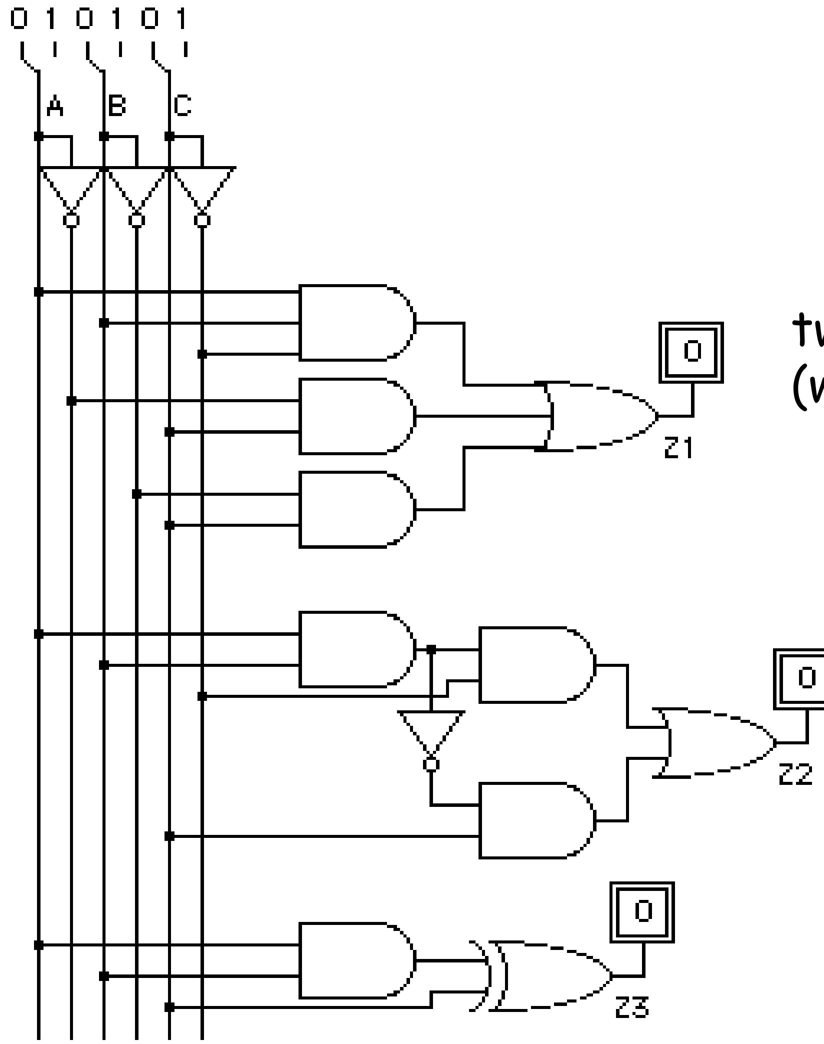
- More than one way to map expressions to gates
  - e.g., $Z = A' \bullet B' \bullet (C + D) = (A' \bullet \underbrace{(B' \bullet \overbrace{(C + D)}^{T2})}_{T1})$





Different realizations of a function

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



two-level realization
(we don't count NOT gates)

multi-level realization
(gates with fewer inputs)

XOR gate (easier to draw
but costlier to build)

Which realization is best?

- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transistors per literal
    - why not count inverters?
  - Fewer literals means less transistors
    - smaller circuits and reduced electric connections
  - Fewer inputs implies faster gates
    - gates are smaller and thus also faster
  - Fan-ins (# of gate inputs) are limited in some technologies
- Reduce number of gates
  - Fewer gates (and the packages they come in) means smaller circuits

Which realization is best?

- Reduce number of levels of gates
  - Fewer level of gates implies reduced signal propagation delays
  - Minimum delay configuration typically requires more gates
    - wider, less deep circuits
- How do we explore tradeoffs between increased circuit delay and size?
  - Automated tools to generate different solutions
  - Logic minimization: reduce number of gates and complexity
  - Logic optimization: reduction while trading off against delay

Canonical forms

- Any logic function can be implemented with only AND, OR, and NOT functions.

- Any logic function can be written in canonical form, where every input is either a true or complemented variable and there are only two levels of gates
  - AND and OR

Canonical forms

- Truth table is the unique signature of a Boolean function
- Many alternative gate realizations may have the same truth table
- Canonical forms
  - Standard forms for a Boolean expression
  - Provides a unique algebraic signature

Canonical forms

- These are called two-level representations
  - sum of products
    - A logical sum (OR) of products (terms using the AND operator)
  - product of sums
    - A logical product (AND) of sums (terms using the OR operator)

Logic functions in Canonical Form

- *D* is true if at least one input is true
- *E* is true if exactly two inputs are true
- *F* is true only if all three inputs are true.

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

- D = A+B+C — Product of Sums
- F = A*B*C — Sum of Products
- E = ((A * B) + (A * C) + (B * C)) * (A * B * C)' — Non-canonical
- E = (A * B * C') + (A * C * B') + (B * C * A') — Sum of Products

Sum-of-products canonical form

- Also known as disjunctive normal form
- Also known as minterm expansion

F = 001 011 101 110 111

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

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

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

Sum-of-products canonical form

- Product term (or minterm)
  - ANDed product of literals – input combination for which output is true
  - Each variable appears exactly once, in true or inverted form (but not both)

A	B	C	minterms	
0	0	0	A'B'C'	m0
0	0	1	A'B'C	m1
0	1	0	A'BC'	m2
0	1	1	A'BC	m3
1	0	0	AB'C'	m4
1	0	1	AB'C	m5
1	1	0	ABC'	m6
1	1	1	ABC	m7

short-hand notation for minterms of 3 variables

F in canonical form:

$$
\begin{aligned}
F(A, B, C) &= \Sigma m(1,3,5,6,7) \\
&= m1 + m3 + m5 + m6 + m7 \\
&= A'B'C + A'BC + AB'C + ABC' + ABC
\end{aligned}
$$

canonical form $\neq$ minimal form

$$
\begin{aligned}
F(A, B, C) &= A'B'C + A'BC + AB'C + ABC + ABC' \\
&= (A'B' + A'B + AB' + AB)C + ABC' \\
&= ((A' + A)(B' + B))C + ABC' \\
&= C + ABC' \\
&= ABC' + C \\
&= AB + C
\end{aligned}
$$

Sum-of-products canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

Sum-of-products canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

- A’ * B’ * C

Sum-of-products canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

- A’ * B’ * C
- A’ * B * C’

Sum-of-products canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

- A’ * B’ * C
- A’ * B * C’
- A * B’ * C’

Sum-of-products canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

- A’ * B’ * C
- A’ * B * C’
- A * B’ * C’
- A * B * C

Sum-of-products canonical form

	Inputs		Output
A	**B**	**C**	**D**
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**

- D = (A’ * B’ * C) + (A’ * B * C’) + (A * B’ * C’) + (A * B * C)

Product-of-sums canonical form

- Also known as conjunctive normal form
- Also known as maxterm expansion

$$F = \quad 000 \qquad\qquad 010 \qquad\qquad 100$$

$$F = (A + B + C)\ (A + B' + C)\ (A' + B + C)$$

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

$$F' = (A + B + C')\ (A + B' + C')\ (A' + B + C')\ (A' + B' + C)\ (A' + B' + C')$$

Product-of-sums canonical form

- Sum term (or maxterm)
  - ORed sum of literals – input combination for which output is false
  - each variable appears exactly once, in true or inverted form (but not both)

A	B	C	maxterms	
0	0	0	A+B+C	M0
0	0	1	A+B+C'	M1
0	1	0	A+B'+C	M2
0	1	1	A+B'+C'	M3
1	0	0	A'+B+C	M4
1	0	1	A'+B+C'	M5
1	1	0	A'+B'+C	M6
1	1	1	A'+B'+C'	M7

short-hand notation for
maxterms of 3 variables

F in canonical form:

$$
\begin{aligned}
F(A, B, C) &= \Pi M(0,2,4) \\
&= M0 \cdot M2 \cdot M4 \\
&= (A + B + C)\,(A + B' + C)\,(A' + B + C)
\end{aligned}
$$

canonical form $\neq$ minimal form

$$
\begin{aligned}
F(A, B, C) &= (A + B + C)\,(A + B' + C)\,(A' + B + C) \\
&= (A + B + C)\,(A + B' + C) \\
&\quad\ (A + B + C)\,(A' + B + C) \\
&= (A + C)\,(B + C)
\end{aligned}
$$

Product-of-sums canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

Product-of-sums canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

- A + B + C

Product-of-sums canonical form

	Inputs		Output
A	**B**	**C**	**D**
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**

- A + B + C
- A + B’ + C’

Product-of-sums canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

- A + B + C
- A + B’ + C’
- A’ + B + C’

Product-of-sums canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

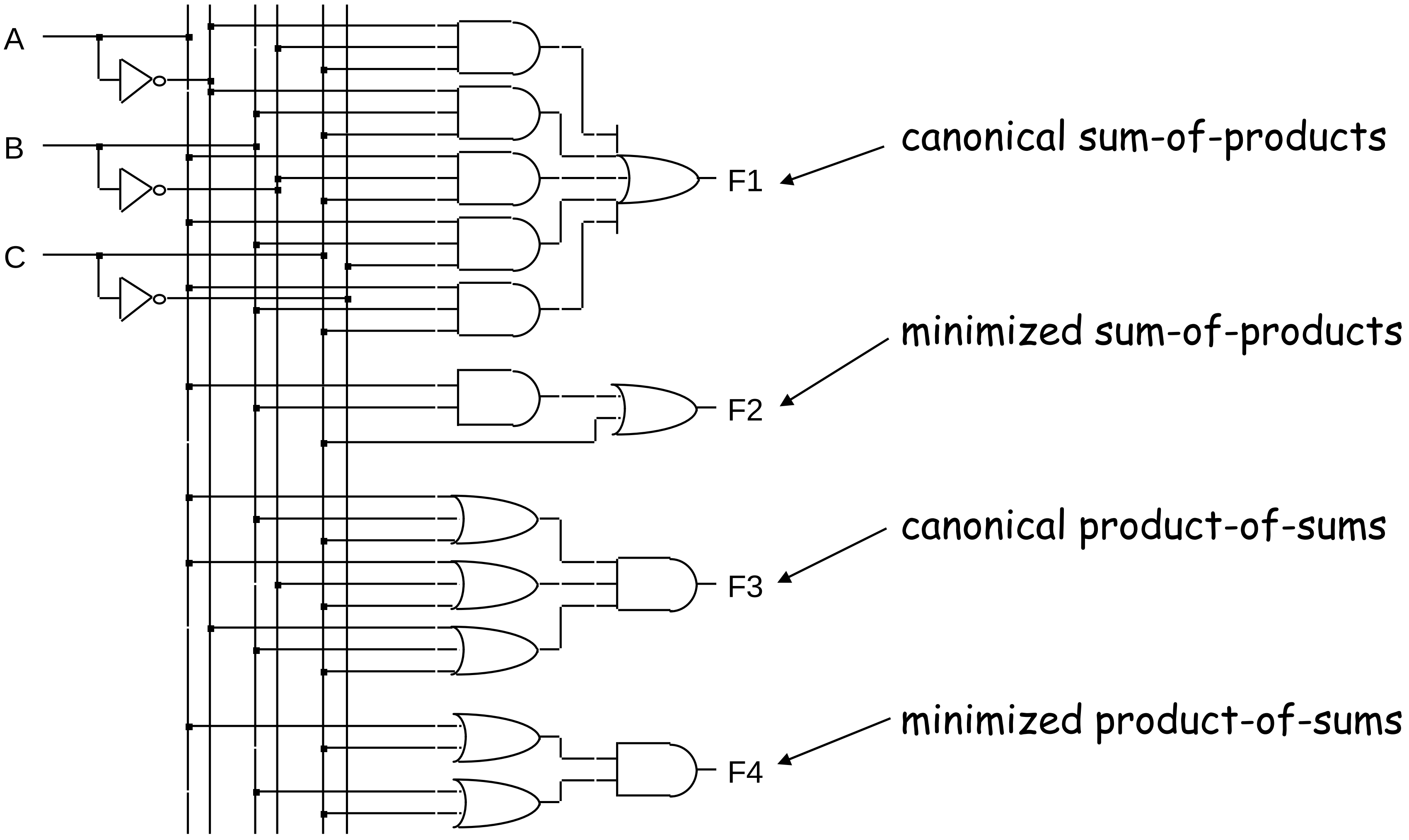
- A + B + C
- A + B’ + C’
- A’ + B + C’
- A’ + B’ + C

Product-of-sums canonical form

Inputs			Output
A	**B**	**C**	**D**
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**

- D= (A + B + C)(A + B’ + C’)(A’ + B + C’)(A’ + B’ + C)

Four alternative two-level implementations of F = AB + C



Mapping between canonical forms

- 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., $F(A,B,C) = \Pi M(0,2,4)$ $F'(A,B,C) = \Pi M(1,3,5,6,7)$

S-o-P, P-o-S, and de Morgan’s theorem

- Sum-of-products
  - F' = A'B'C' + A'BC' + AB'C‘

- Apply de Morgan's
  - (F')' = (A'B'C' + A'BC' + AB'C')'
  - F = (A + B + C) (A + B' + C) (A' + B + C)

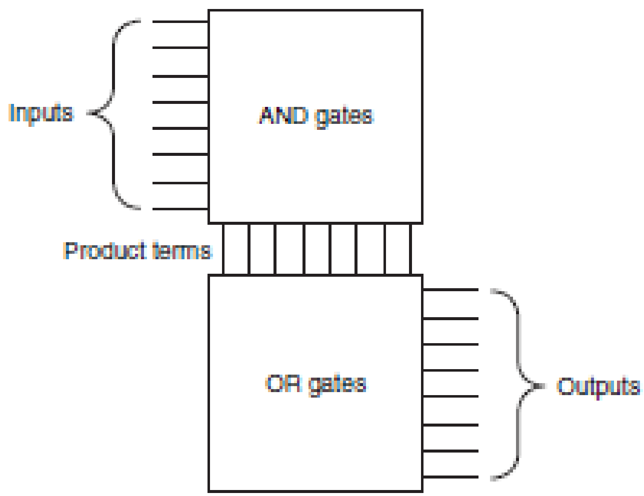
- Product-of-sums
  - F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')

- Apply de Morgan's
  - (F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'
  - F = A'B'C + A'BC + AB'C + ABC' + ABC

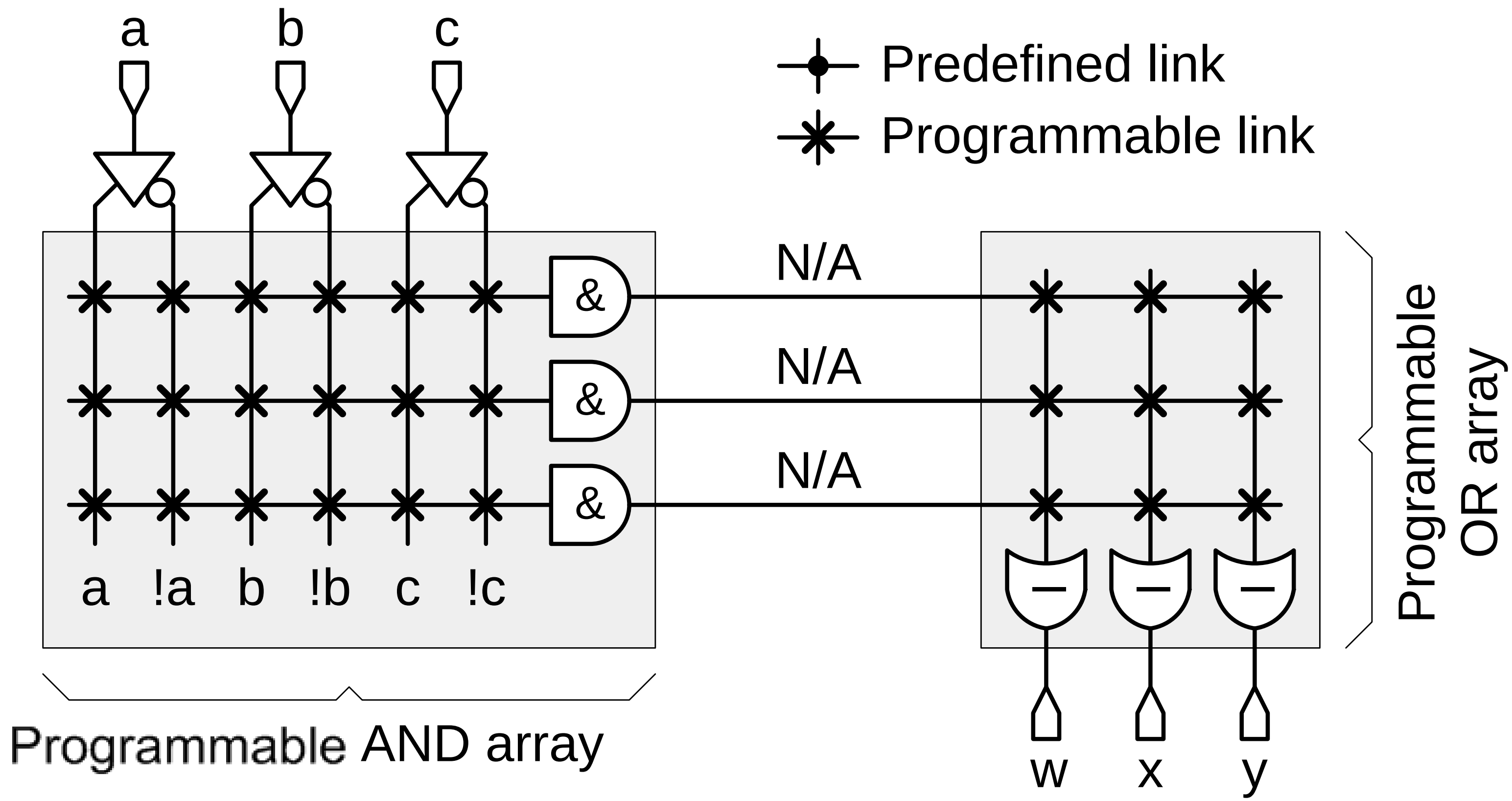
Programmable Logic Array

- The relationship between a truth table and a two-level representation allows us to generate a gate-level implementation of any set of logic functions.

- The sum-of-products corresponds to a programmable logic array.



Programmable Logic Array



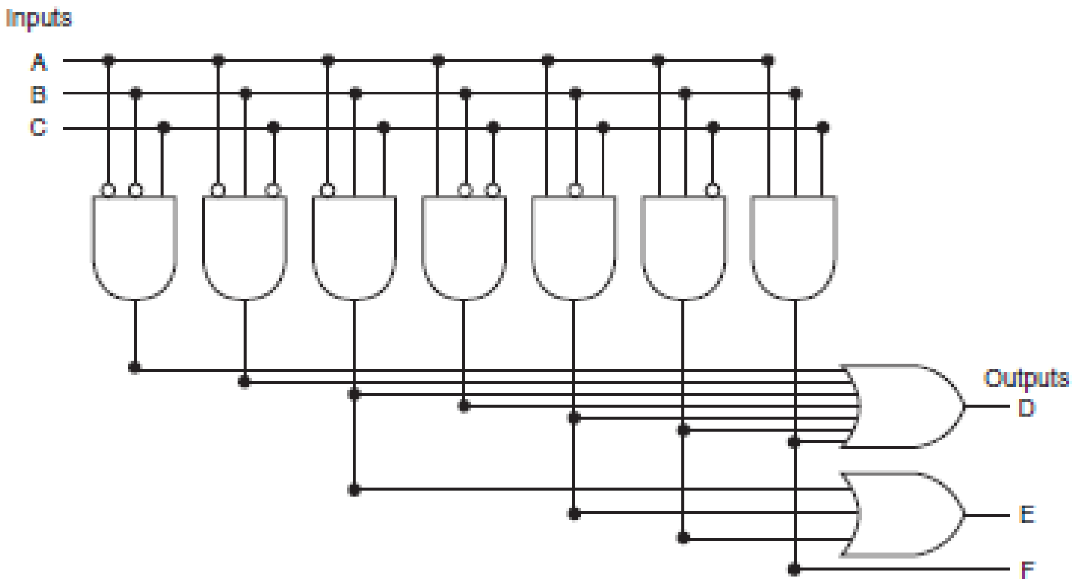
The Design Warrior's Guide to FPGAs
Devices, Tools, and Flows. ISBN 0750676043

Programmable Logic Array

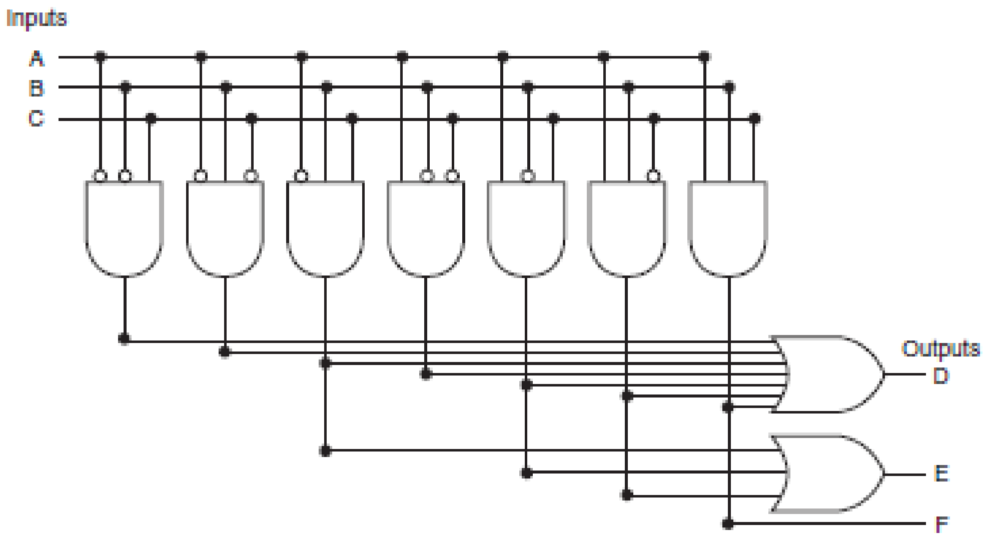
- Efficient Characteristics
  - only the truth table entries that produce a true value for at least one output have any logic gates associated with them.
  - each different product term will have only one entry in the PLA, even if the product term is used in multiple outputs.

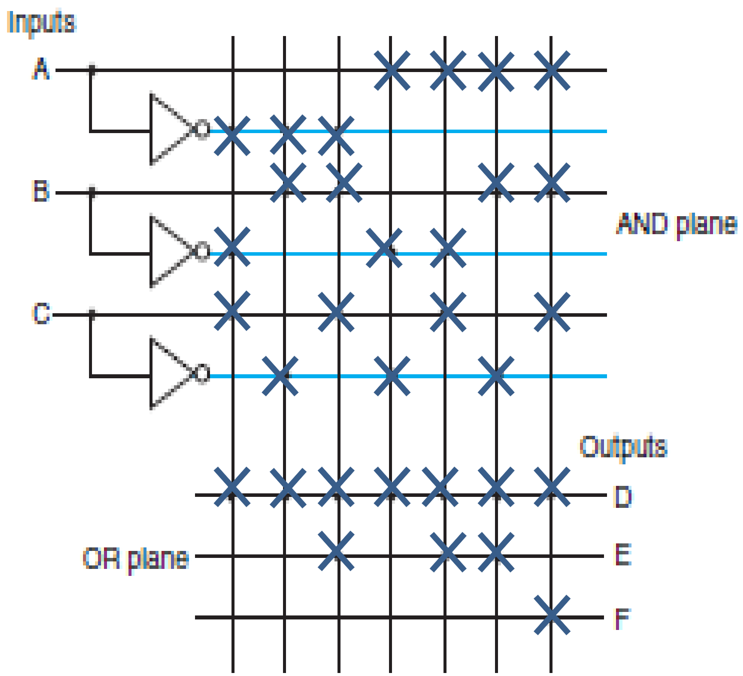
Programmable Logic Array

Inputs			Outputs		
A	**B**	**C**	**D**	**E**	**F**
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1



Programmable Logic Array





Incompleteley specified functions

- Example: binary coded decimal increment by 1
  - BCD digits encode decimal digits 0 – 9 in bit patterns 0000 – 1001

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

off-set of W

on-set of W

don't care (DC) set of W

these input patterns should never be encountered in practice – "don't care" about associated output values, can be exploited in minimization

Notation for incompletely specified functions

- Don't cares and canonical forms
  - So far, only represented on-set
  - Also represent don't-care-set
  - Need two of the three sets (on-set, off-set, dc-set)

- Canonical representations of the BCD increment by 1 function:

  - Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15
  - Z = Σ [m(0,2,4,6,8) + d(10,11,12,13,14,15)]

  - Z = M1 • M3 • M5 • M7 • M9 • D10 • D11 • D12 • D13 • D14 • D15
  - Z = Π [M(1,3,5,7,9) • D(10,11,12,13,14,15)]

Simplification of two-level combinational logic

- Recall that canonical forms guarantee us 2 levels of logic. However, canonical forms do not guarantee us the most minimal version of the function.

- canonical form ≠ minimal form

$$
\begin{aligned}
F(A, B, C) &= (A + B + C)\,(A + B' + C)\,(A' + B + C) \\
&= (A + B + C)\,(A + B' + C)\,(A + B + C)\,(A' + B + C) \\
&= (A + C)\,(B + C)
\end{aligned}
$$

Simplification of two-level combinational logic

- The goal is to find a minimal sum of products or product of sums realization
  - Exploit don't care information in the process
- Algebraic simplification
  - Not an algorithmic/systematic procedure
  - How do you know when the minimum realization has been found?
- Computer-aided design tools
  - Precise solutions require very long computation times, especially for functions with many inputs (> 10)
  - Heuristic methods employ "educated guesses" to reduce amount of computation and yield good if not best solutions
- Hand methods still relevant
  - To understand automatic tools and their strengths and weaknesses
  - Ability to check results (on small examples)

The uniting theorem

- Key tool to simplification: A (B' + B) = A
- Essence of simplification of two-level logic
  - Find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

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

A	B	F
$\boxed{0}$	$\boxed{0}$	1
0	1	0
$\boxed{1}$	$\boxed{0}$	1
1	1	0

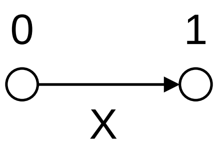
B has the same value in both on-set rows
– B remains

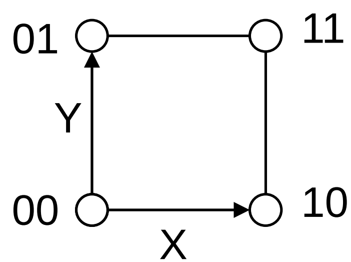
A has a different value in the two rows
– A is eliminated

Boolean cubes

- Visual technique for identifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"

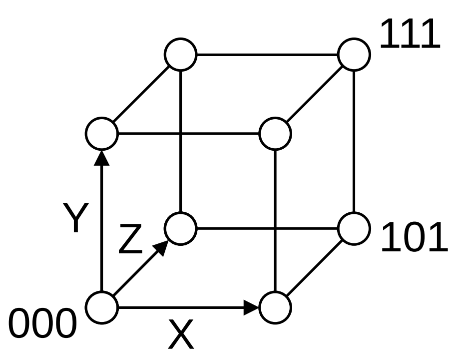
1-cube

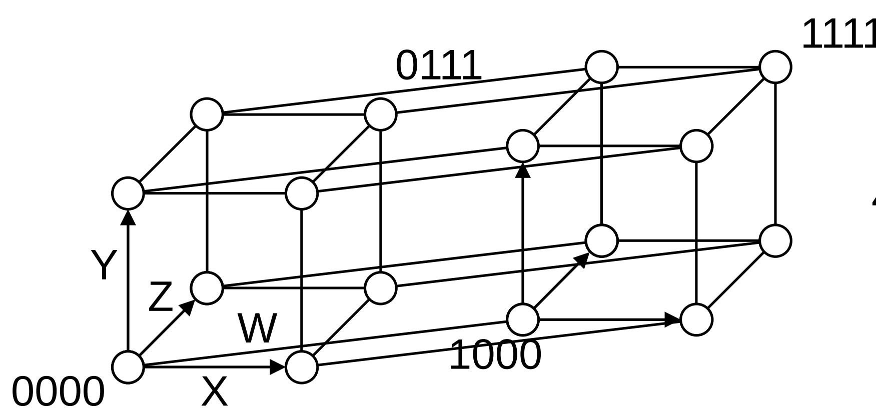




2-cube

3-cube



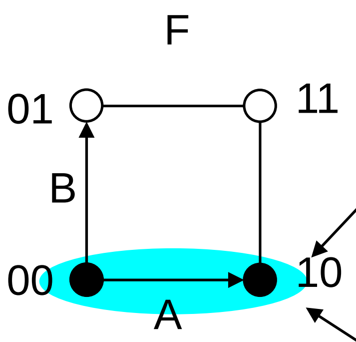


4-cube

Mapping truth tables onto Boolean cubes

- Uniting theorem combines two "faces" of a cube into a larger "face"
- Example:

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



two faces of size 0 (nodes) combine into a face of size 1(line)

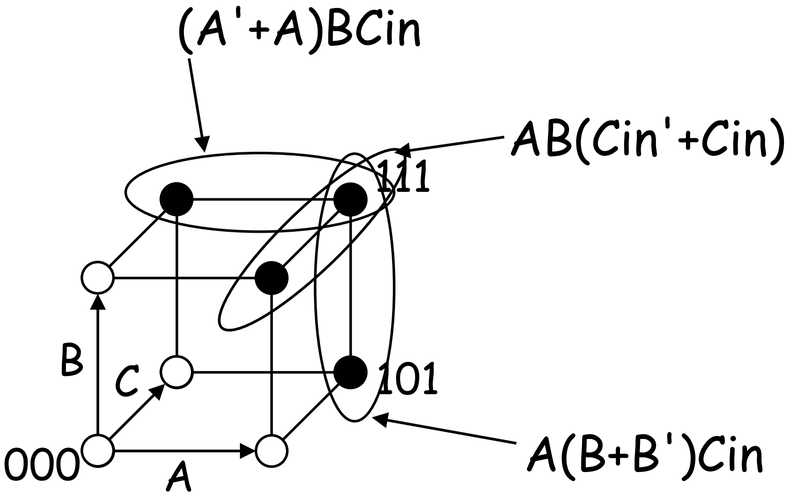
A varies within face, B does not this face represents the literal B'

ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×'d nodes

Three variable example

- Binary full-adder carry-out logic

A	B	Cin	Cout
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

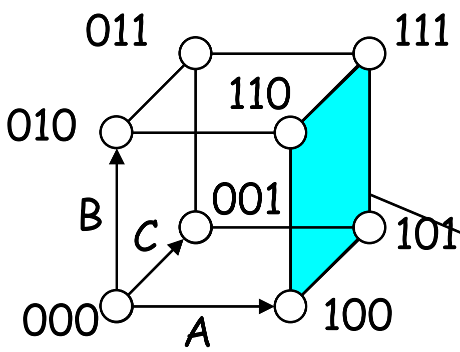


the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

$$Cout = BCin+AB+ACin$$

Higher dimensional cubes

- Sub-cubes of higher dimension than 2



$F(A,B,C) = \Sigma m(4,5,6,7)$

on-set forms a square
i.e., a cube of dimension 2

represents an expression in one variable
i.e., 3 dimensions – 2 dimensions

A is asserted (true) and unchanged
B and C vary

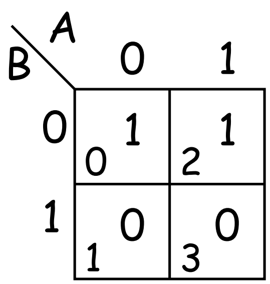
This subcube represents the literal A

m-dimensional cubes in a n-dimensional Boolean space

- In a 3-cube (three variables):
  - 0-cube, i.e., a single node, yields a term in 3 literals
  - 1-cube, i.e., a line of two nodes, yields a term in 2 literals
  - 2-cube, i.e., a plane of four nodes, yields a term in 1 literal
  - 3-cube, i.e., a cube of eight nodes, yields a constant term "1"
- In general,
  - m-subcube within an n-cube ($m < n$) yields a term with $n – m$ literals

Karnaugh maps

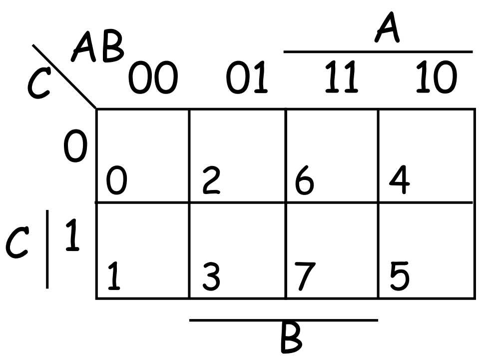
- Flat map of Boolean cube
  - Wrap–around at edges
  - Hard to draw and visualize for more than 4 dimensions
  - Virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - Guide to applying the uniting theorem
  - On-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table

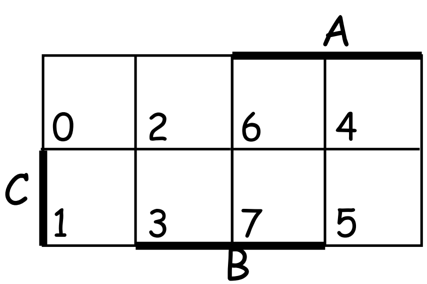


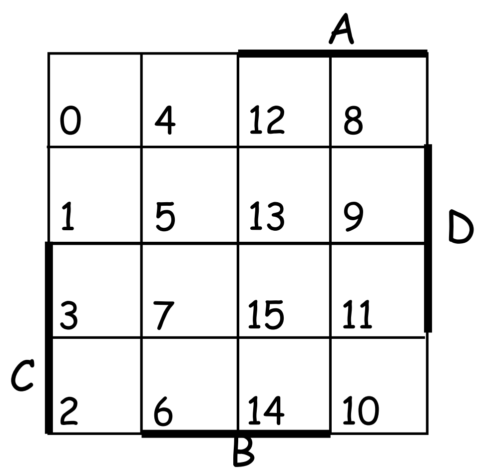
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

Karnaugh maps (cont'd)

- Numbering scheme based on Gray–code
  - e.g., 00, 01, 11, 10
  - Only a single bit changes in code for adjacent map cells



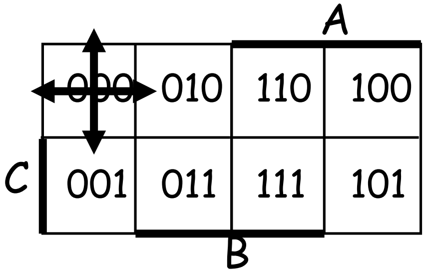


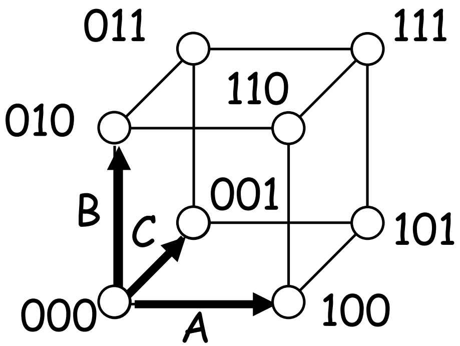


13 = 1101= ABC'D

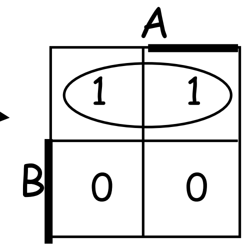
Adjacencies in Karnaugh maps

- Wrap from first to last column
- Wrap top row to bottom row

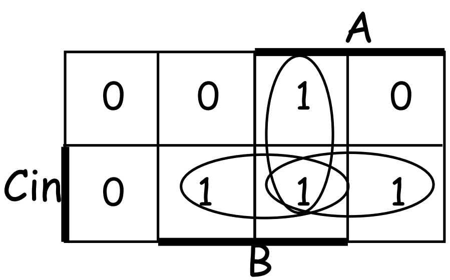




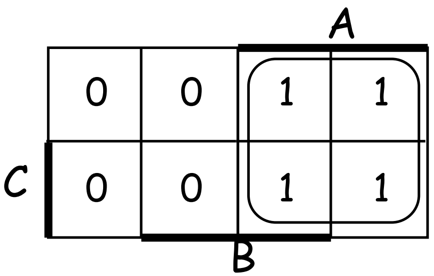
Karnaugh map examples

- F = 

B'

- Cout = 

AB + ACin + BCin

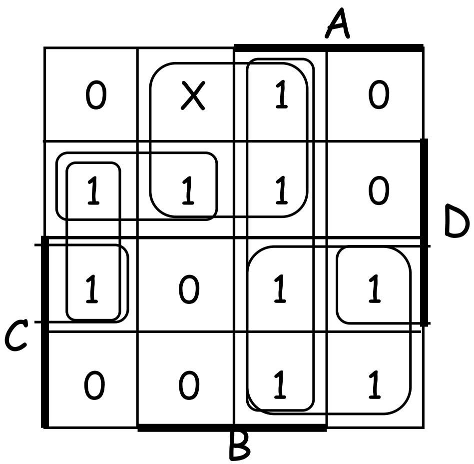
- G(A,B,C) = 

A

Definition of terms for two-level simplification

- Implicant
  - Single element of ON-set or DC-set or any group of these elements that can be combined to form a subcube
- Prime implicant
  - Implicant that can't be combined with another to form a larger subcube
- Essential prime implicant
  - Prime implicant is essential if it alone covers an element of ON-set
  - Will participate in ALL possible covers of the ON-set
  - DC-set used to form prime implicants but not to make implicant essential
- Objective:
  - Grow implicant into prime implicants (minimize literals per term)
  - Cover the ON-set with as few prime implicants as possible (minimize number of product terms)

Examples to illustrate terms



6 prime implicants:

A'B'D, BC', AC, A'C'D, AB, B'CD

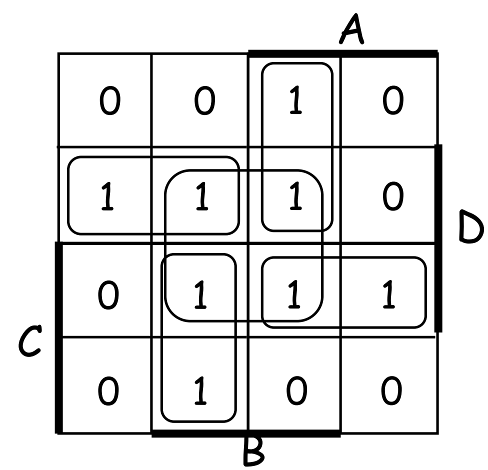
essential

minimum cover: AC + BC' + A'B'D

5 prime implicants:

BD, ABC', ACD, A'BC, A'C'D

essential



minimum cover: 4 essential implicants

Karnaugh map examples

- $f(A,B,C) = \Sigma m(0,4,5,7)$

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

$AC + B'C'$

~~$+ AB'$~~

- Can we also determine f'?
  - Option 1:

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

$BC' + A'C$

We can obtain the complement of the function by covering the 0s instead of 1s

  - Option 2:

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

$BC' + A'C$

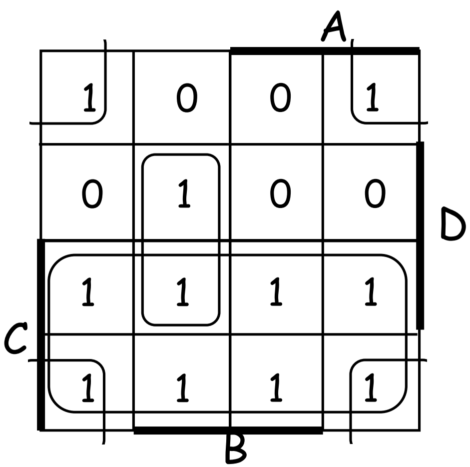
We can obtain the complement by replacing 1's with 0's and vice versa

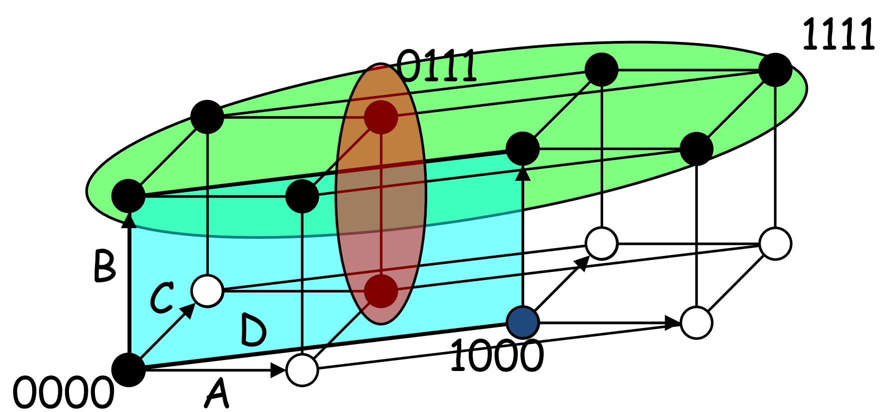
Karnaugh map: 4-variable example

- F(A,B,C,D) = Σm(0,2,3,5,6,7,8,10,11,14,15)

F =

C + A'BD + B'D'

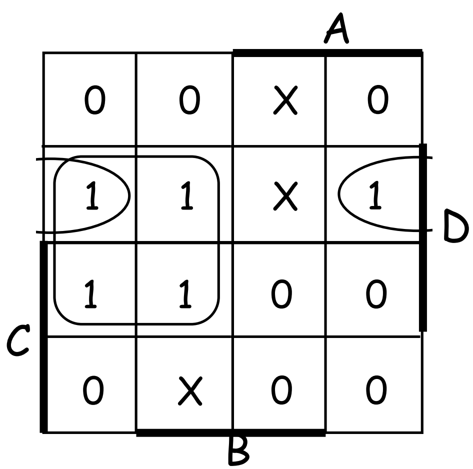




find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)

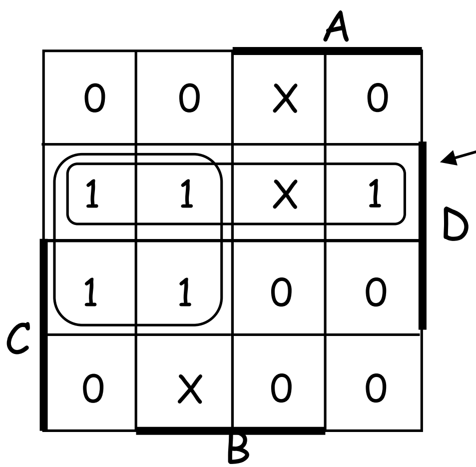
Karnaugh maps: don’t cares

- $f(A,B,C,D) = \Sigma\, m(1,3,5,7,9) + d(6,12,13)$
  - without don't cares
    - $f = A'D + B'C'D$



Karnaugh maps: don’t cares

- $f(A,B,C,D) = \Sigma\, m(1,3,5,7,9) + d(6,12,13)$
  - $f = A'D + B'C'D$ without don't cares
  - $f = A'D + C'D$ with don't cares



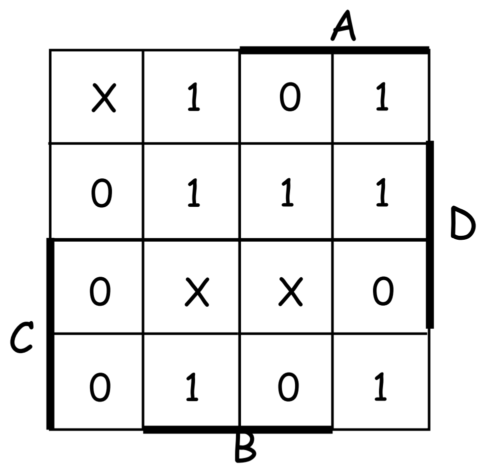
by using don't care as a "1"
a 2-cube can be formed
rather than a 1-cube to cover
this node

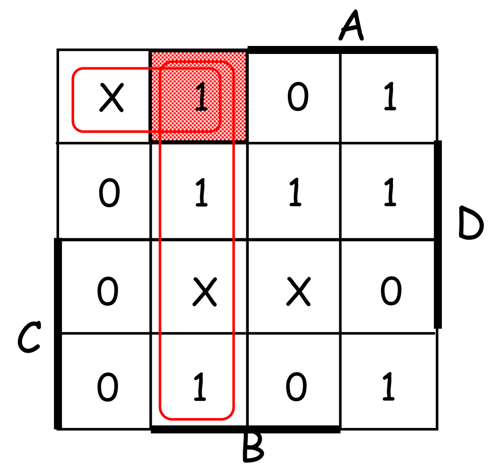
<u>don't cares</u> can be treated as
1s or 0s
depending on which is more
advantageous

Algorithm for two-level simplification

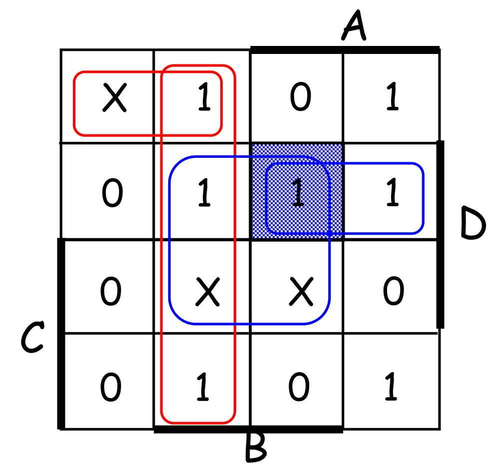
- To get the minimum sum-of-products expression from a Karnaugh map:
  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number of elements always a power of 2)
  - Repeat Steps 1 and 2 to find all prime implicants
  - Step 3: revisit the 1s in the K-map
    - if covered by single prime implicant, it is essential, and participates in final cover
    - 1s covered by essential prime implicant do not need to be revisited
  - Step 4: if there remain 1s not covered by essential prime implicants
    - select the smallest number of prime implicants that cover the remaining 1s

Algorithm for two-level simplification (example)

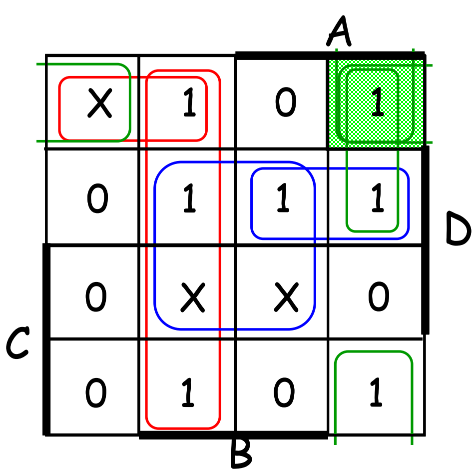




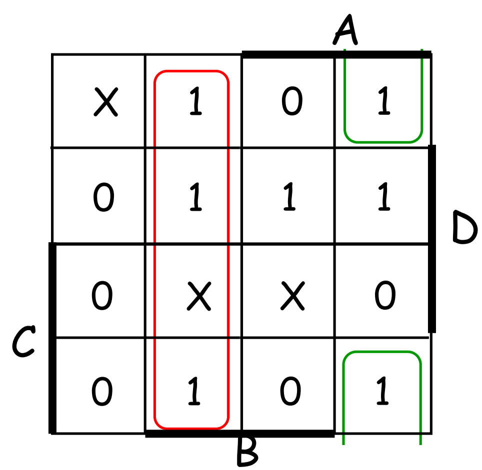
2 primes around A'BC'D'



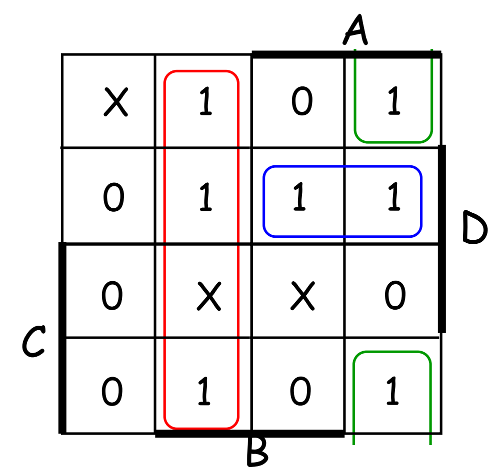
2 primes around ABC'D



3 primes around AB'C'D'

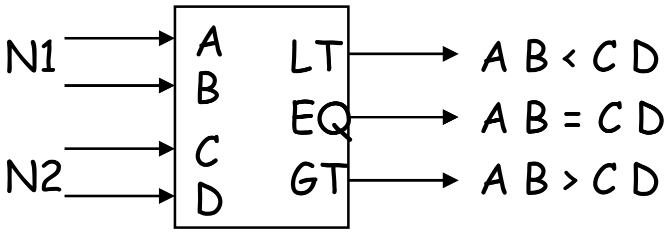


2 essential primes



minimum cover (3 primes)

Design example: two-bit comparator



block diagram
and
truth table

A	B	C	D	LT	EQ	GT
0	0	0	0	0	1	0
		0	1	1	0	0
		1	0	1	0	0
		1	1	1	0	0
0	1	0	0	0	0	1
		0	1	0	1	0
		1	0	1	0	0
		1	1	1	0	0
1	0	0	0	0	0	1
		0	1	0	0	1
		1	0	0	1	0
		1	1	1	0	0
1	1	0	0	0	0	1
		0	1	0	0	1
		1	0	0	0	1
		1	1	0	1	0

we'll need a 4-variable Karnaugh map
for each of the 3 output functions

Design example: two-bit comparator

0	0	0	0
1	0	0	0
1	1	0	1
1	1	0	0

K-map for LT

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

K-map for EQ

0	1	1	1
0	0	1	1
0	0	0	0
0	0	1	0

K-map for GT

$LT = A'B'D + A'C + B'CD$

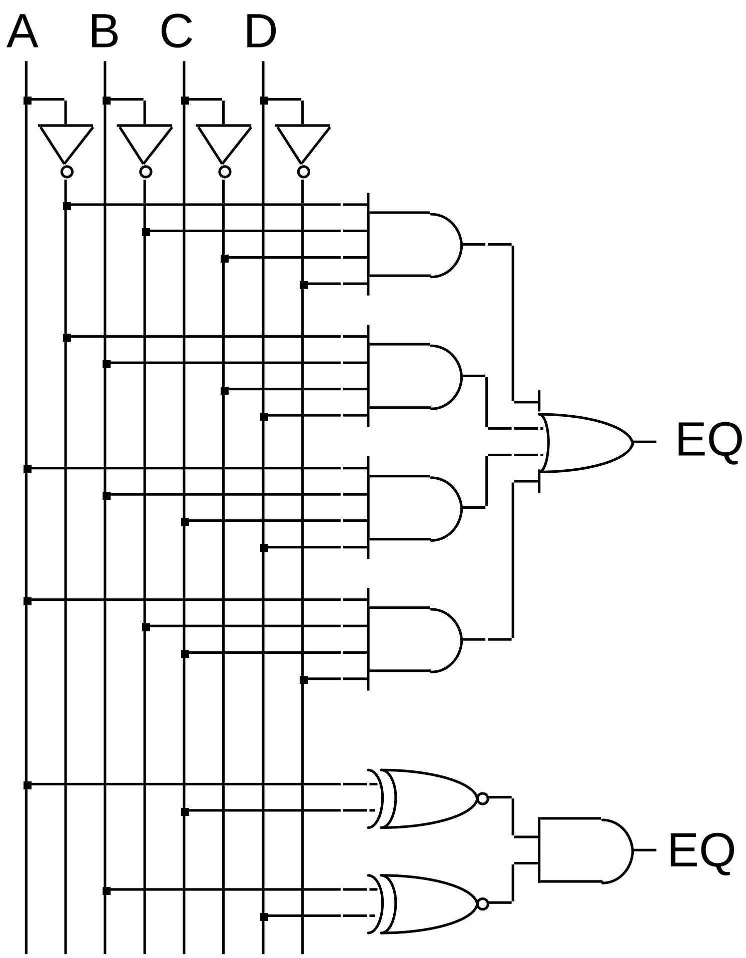
$EQ = A'B'C'D' + A'BC'D + ABCD + AB'CD'$

$= (A \text{ xnor } C) \cdot (B \text{ xnor } D)$

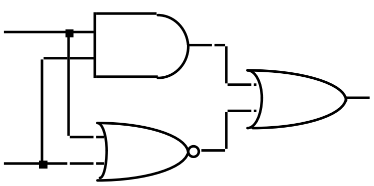
$GT = BC'D' + AC' + ABD'$

LT and GT are similar (flip A/C and B/D)

Design example: two-bit comparator

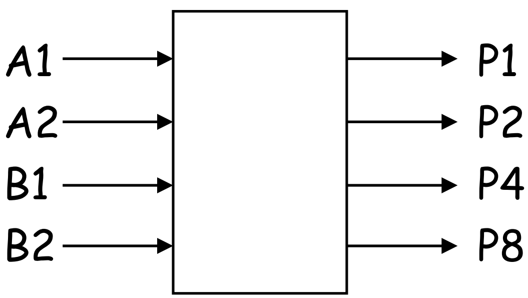


two alternative implementations of EQ with and without XNOR



XNOR is implemented with at least 3 simple gates

Design example: 2x2-bit multiplier



block diagram
and
truth table

A2	A1	B2	B1	P8	P4	P2	P1
0	0	0	0	0	0	0	0
		0	1	0	0	0	0
		1	0	0	0	0	0
		1	1	0	0	0	0
0	1	0	0	0	0	0	0
		0	1	0	0	0	1
		1	0	0	0	1	0
		1	1	0	0	1	1
1	0	0	0	0	0	0	0
		0	1	0	0	1	0
		1	0	0	1	0	0
		1	1	0	1	1	0
1	1	0	0	0	0	0	0
		0	1	0	0	1	1
		1	0	0	1	1	0
		1	1	1	0	0	1

4-variable K-map
for each of the 4
output functions

Design example: 2x2-bit multiplier (cont'd)

K-map for P8

A2

0	0	0	0
0	0	0	0
0	0	1	0
0	0	0	0

B1 B2 A1

P8 = A2A1B2B1

K-map for P4

A2

0	0	0	0
0	0	0	0
0	0	0	1
0	0	1	1

B1 B2 A1

P4 = A2B2B1'
+ A2A1'B2

K-map for P2

A2

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0

B1 B2 A1

P2 = A2'A1B2
+ A1B2B1'
+ A2B2'B1
+ A2A1'B1

K-map for P1

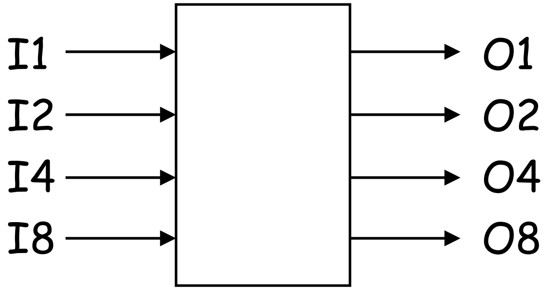
A2

0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0

B1 B2 A1

P1 = A1B1

Design example: BCD increment by 1

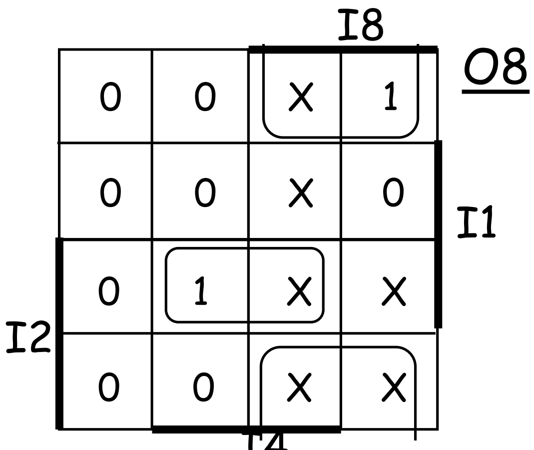


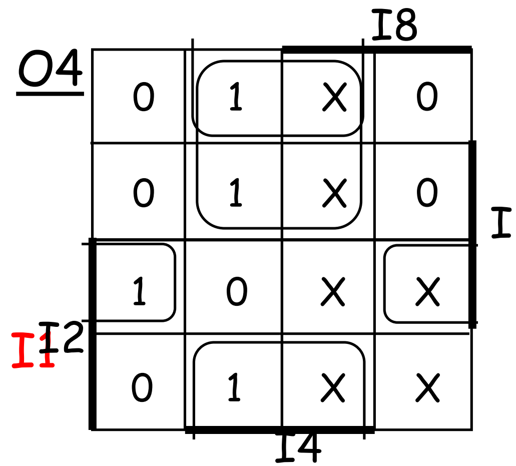
block diagram
and
truth table

I8	I4	I2	I1	O8	O4	O2	O1
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

4-variable K-map for each of
the 4 output functions

Design example: BCD increment by 1 (cont'd)



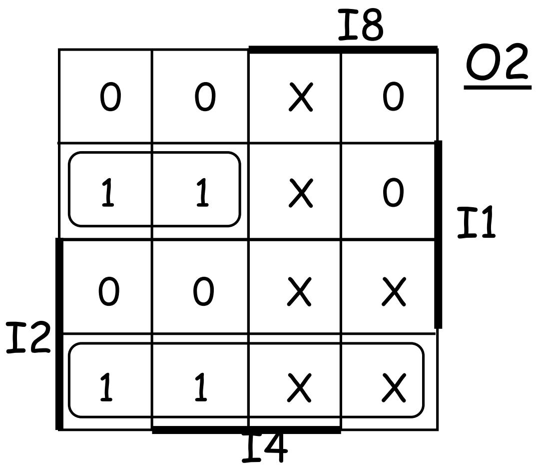


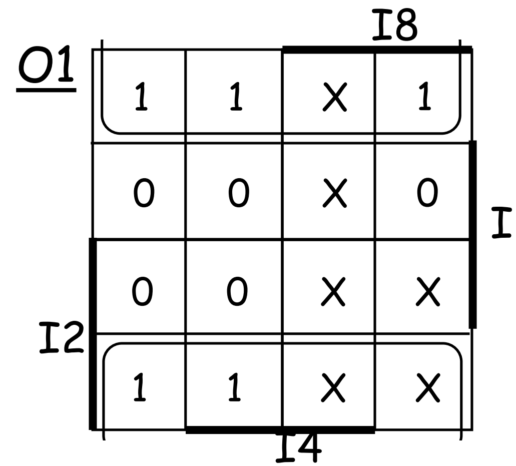
$O8 = I4\ I2\ I1 + I8\ I1'$

$O4 = I4\ I2' + I4\ I1' + I4'\ I2\ I1$

$O2 = I8'\ I2'\ I1 + I2\ I1'$

$O1 = I1'$

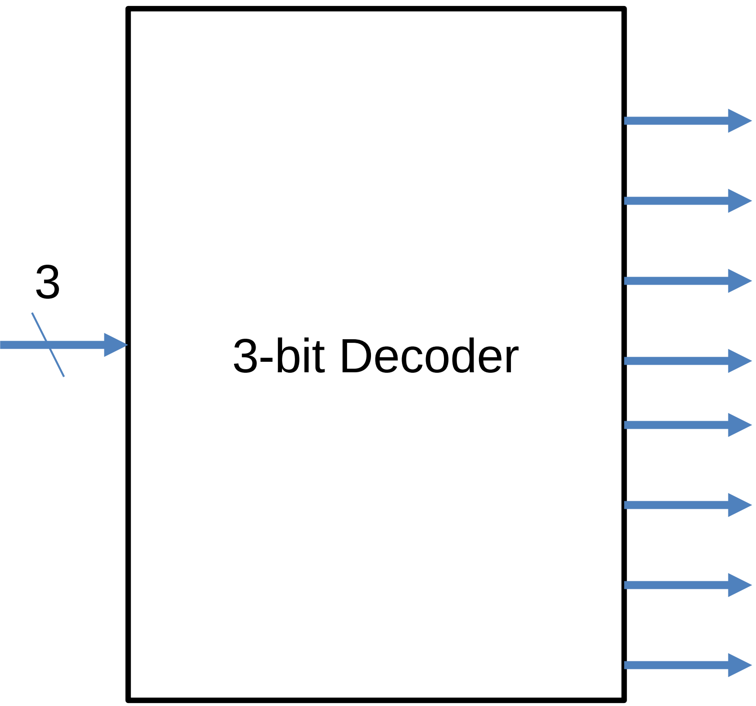




Combinational Hardware: Decoders

- A decoder is a logic block that takes in an n-bit input and selects from 2^n outputs.
  - One output is asserted for each possible input combination.
  - Outputs are labeled Out0, Out1, …, $\text{Out}2^n - 1$
  - If the input is k, then Outk will be true

Combinational Hardware: Decoders



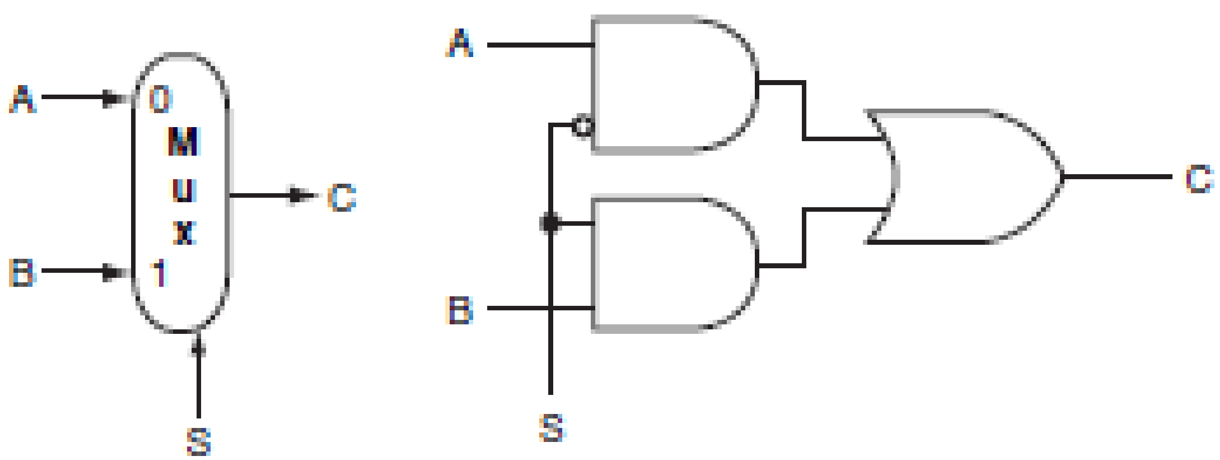
i2	i1	i0	o7	o6	o5	o4	o3	o2	o1	o0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Combinational Hardware: Multiplexors

- A multiplexor is a logic block that takes in n inputs and selects one to be the output.
  - Could also be called a selector
  - The output is one of the inputs, selected by a control value.

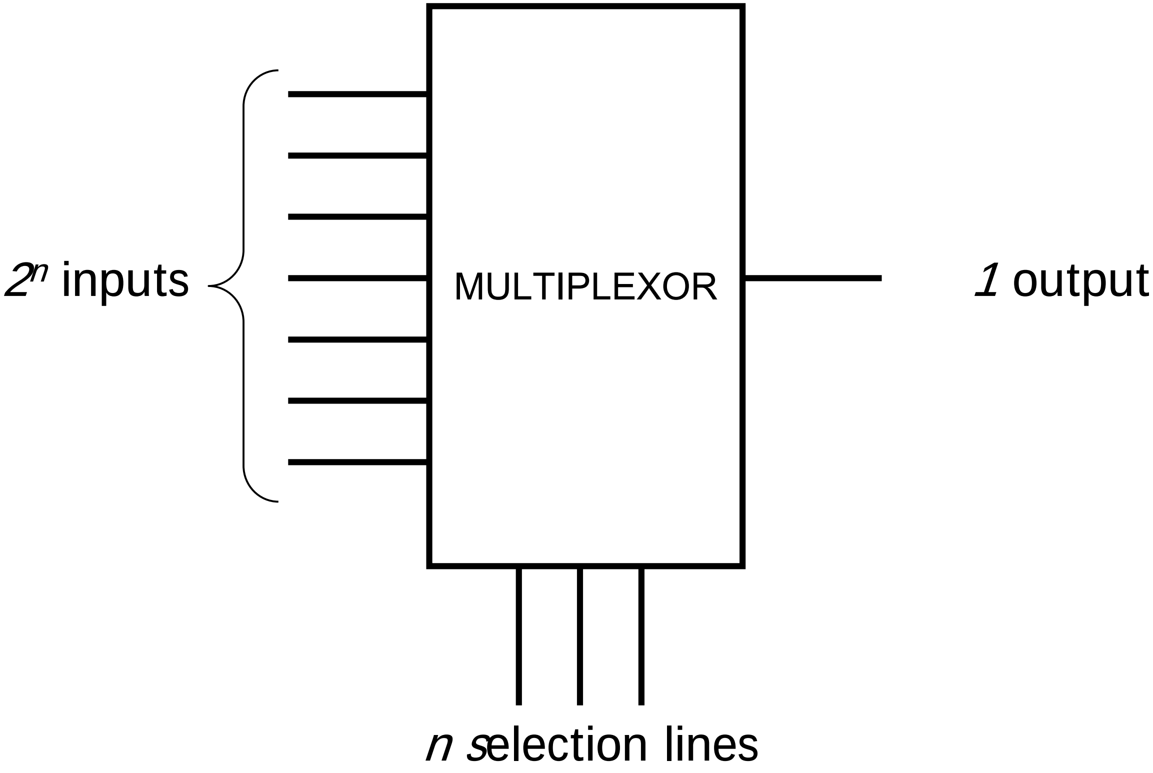
Combinational Hardware: Multiplexors

- A multiplexor is a logic block that takes in n inputs and selects one to be the output.
  - Could also be called a selector
  - The output is one of the inputs, selected by a control value.

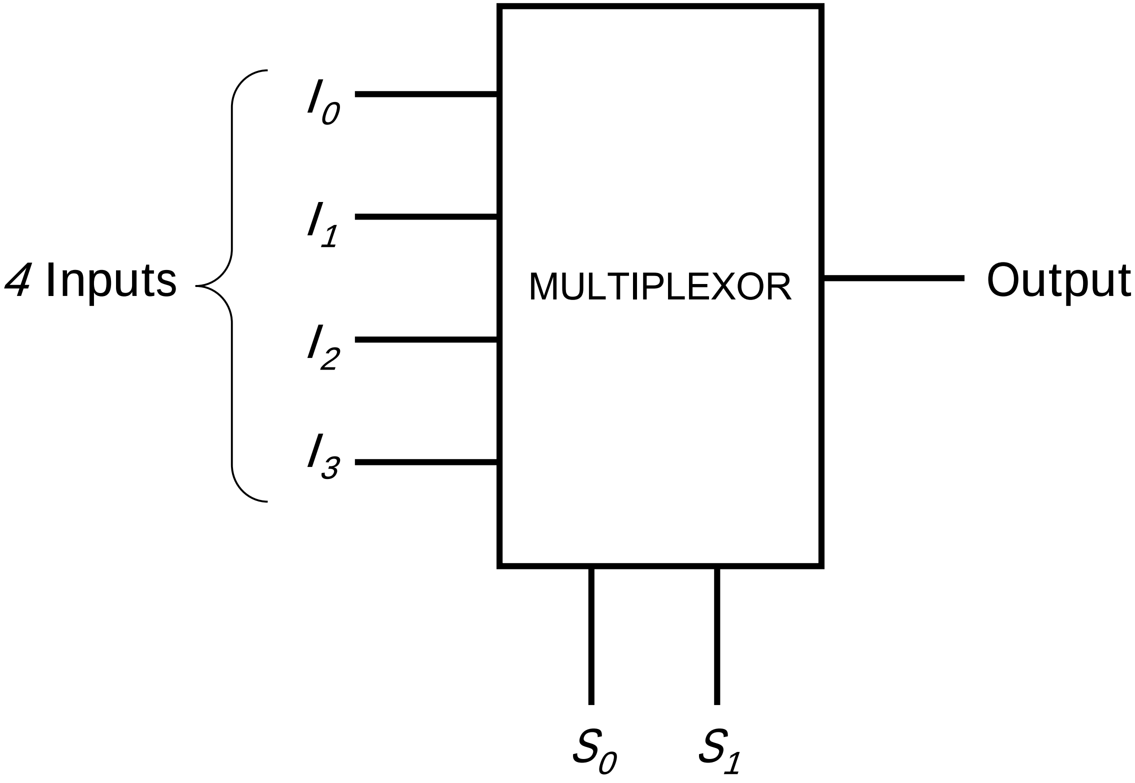


Combinational Hardware: Multiplexors

- Multiplexors can be created with an arbitrary number of data inputs.



Combinational Hardware: Multiplexors



S_1	S_0	Output
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Combinational Hardware: Multiplexors

- Logic Gates