2.1 Two-level combinational logic simplification

- Logic simplification (minimization): simplify a Boolean expression before converting to a circuit to yield a smaller circuit

- Number of variables vs number of literals
    - r = abc’ + ab’c’
    - Number of variables is 3, a, b, c
    - Number of literals is 6, each appearance is a literal

- Number of transistors
    - If each AND or OR gate **input** requires 2 transistors, how many transistors does the original expression circuit require?
    - 16:
        - two AND needs 3 * 2 = 6 inputs, one OR needs 2 inputs
        - 8 inputs in total, 2 transistors for each input

- Seeking i(j + j’) opportunities
    - Eliminate redundancy

2.2 K-maps: introduction

- K-map: a graphical function representation that eases the simplification process for expressions involving a few variables, by adjacently placing miniterms that differ in exactly one variable
- Example: y = a’b’ + ab’

a \ b	0	1
0		a'b
1	ab'	ab

a	b	y
0	0	1
0	1	0
1	0	1
1	1	0

a \ b	0	1
0	1	0
1	1	0

- Simplification rules
    - 1. Cover every 1 at least once using circles. Add circle’s term to expression
    - 2. Use fewest and largest circles possible, to achieve the simplest expression

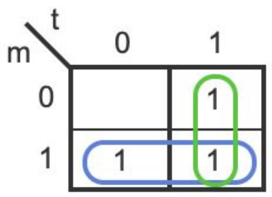
- Powerfulness of K-map

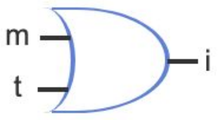
Algebraic simplification

i = mt' + tm' + tm
i = mt' + m't + mt
i = mt' + m't + mt + mt
i = mt' + mt + m't + mt
i = m(t' + t) + (m' +m) t
i = m(1) + (1)t
i = m(1) + t(1)
i = m + t

K-map simplification

i = mt' + m't + mt
i = m + t





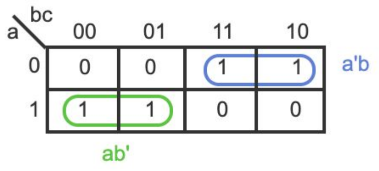
-

- K-maps are a **convenient** way of applying algebraic properties
- Each circle corresponds to a specific sequence of algebraic properties

2.3 3- and 4-variable K-maps

- 3- : two variables across the top
    - Simplification

y = ab'c' + ab'c + a'bc + a'bc'
y = ab' + a'b



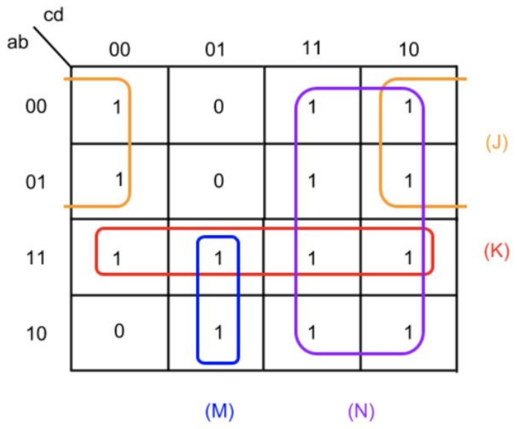
ab'c' + ab'c
ab'(c' + c)
ab'(1)
ab'

a'bc + a'bc'
a'b(c + c')
a'b(1)
a'b

    -

- Larger circles
    - All combinations have to appear

- 4-
    - Valid circle size: 1, 2, 4, 8, and 16
    - Example:
        - J: a'd'
        - K: ab
        - M: ac'd
        - N: c



2.4 K-map examples

- Arbiter
    - Decides which of several competing items wins
    - Example:
        - If two devices simultaneously try to access a resource like a printer, an arbiter can decide which device will get access

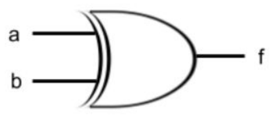
2.5 DeMorgan’s Law

- (a + b)’ = a’b’
- (ab)’ = a’ + b’, similar for 3 variables, (abc)’ = a’ + b’ + c’

- (ab)’ is **not** in sum-of-product form
- But after applying DeMorgan’s Law, a’ + b’ is in sum-of-product form

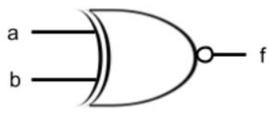
2.6 XOR / XNOR gates

a	b	f
0	0	0
0	1	1
1	0	1
1	1	0



- XOR: outputs 1 if the input values differ
- y = a XOR b is equivalent to y = ab’ + a’b

a	b	f
0	0	1
0	1	0
1	0	0
1	1	1



- XNOR: outputs 1 if the inputs values are the same
- Opposite (NOT) of XOR gate, y = ab + a’b’

- Multi-input XOR and XNOR
    - Output is 1 if the number of input 1’s is odd
    - Otherwise output 0
    - Vice versa for XNOR

- Even parity bit
    - To maintain the total number of 1’s even
    - Output 1 when need to create an even number of 1s

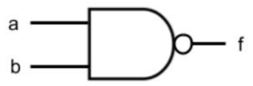
- Deriving XNOR’s expression using DeMorgan’s Law

(a XOR b)'
(a'b + ab')'
(a'b)' · (ab')' DeMorgan's Law
(a'' + b')(a' + b'') DeMorgan's Law (again)
(a + b')(a' + b)
aa' + ab + b'a' + b'b
0 + ab + a'b' + 0
ab + a'b'
a XNOR b

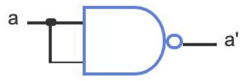
-

2.7 NAND / NOR (universal gates)

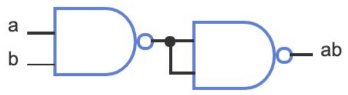
a	b	f
0	0	1
0	1	1
1	0	1
1	1	0



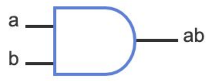
- NAND: not AND
    - Is a universal gate
    - Can implement any combinational circuit

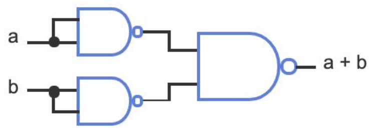


(aa)' = a' + a' = a' (NOT)

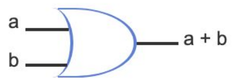


((ab)')' = (ab)'' = ab (AND)





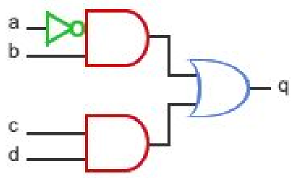
(a'b')' = a'' + b'' = a + b (OR)

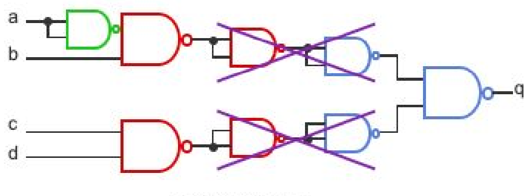


NAND is thus a universal gate

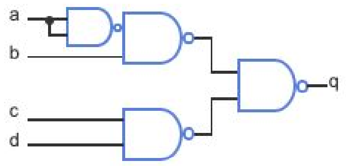
    -
    - Inverting the inputs of a NAND gate produces an OR gate
    - Converting to a NAND-only implementation

q = a'b + cd

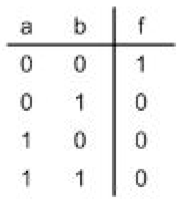




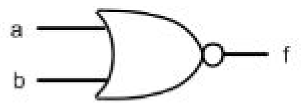
Direct replacement



    -

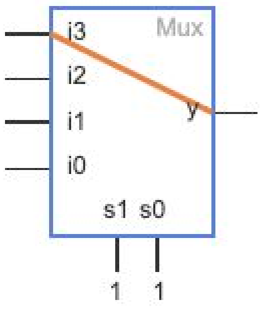
- NOR: not OR
    - 

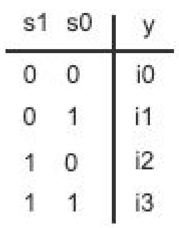
a	b	f
0	0	1
0	1	0
1	0	0
1	1	0



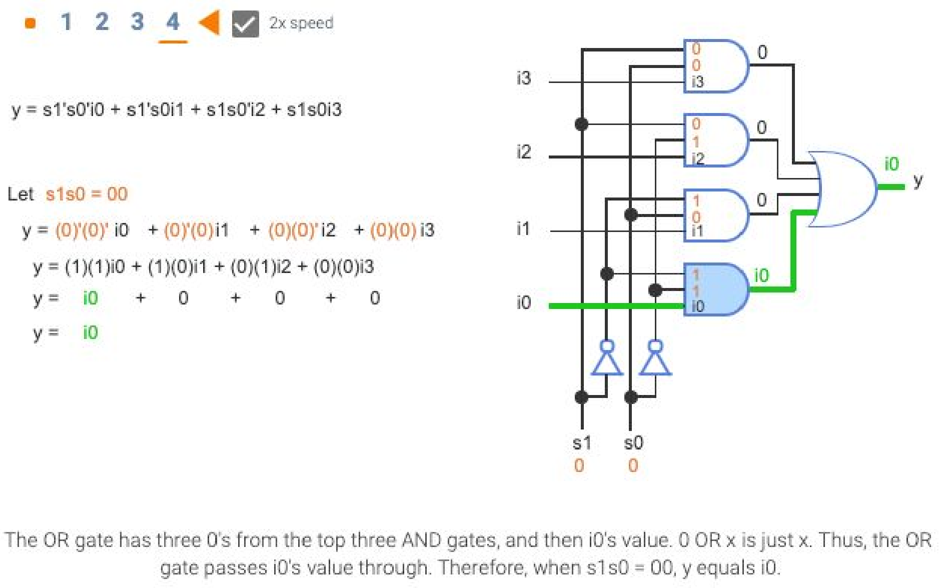
    - Is also a **universal gate**

<u>2.8 Muxes</u>

- Multiplexor
    - A combinational circuit that passes one of multiple data inputs through to a single output, selecting which one based on additional control inputs.
    - Mux is short for multiplexor
    - Control inputs ⇒ called **select lines**
    - 



s1	s0	y
0	0	i0
0	1	i1
1	0	i2
1	1	i3

- Mux equation and circuit
    - 

y = s1's0'i0 + s1's0i1 + s1s0'i2 + s1s0i3

Let s1s0 = 00

y = (0)'(0)' i0 + (0)'(0)i1 + (0)(0)' i2 + (0)(0) i3

y = (1)(1)i0 + (1)(0)i1 + (0)(1)i2 + (0)(0)i3

y = i0 + 0 + 0 + 0

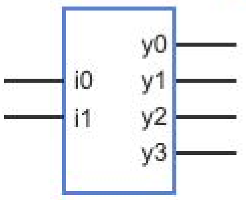
y = i0

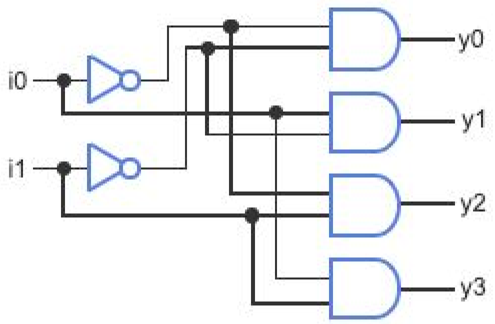
The OR gate has three 0's from the top three AND gates, and then i0's value. 0 OR x is just x. Thus, the OR gate passes i0's value through. Therefore, when s1s0 = 00, y equals i0.

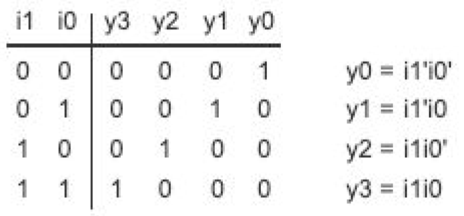
- Mux size
    - Requires $\log_2 N$ select inputs
    - So the sizes should be 2x1, 4x1, 8x1, 16x1, etc.

2.9 Decoders

- A ***decoder*** is a combinational circuit converts N inputs to a **single** 1 on one of 2^N outputs
- Helps to reduce the output pins needed
- The number of output set to one at **any given time** is 1



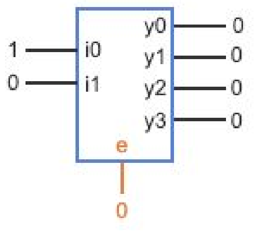


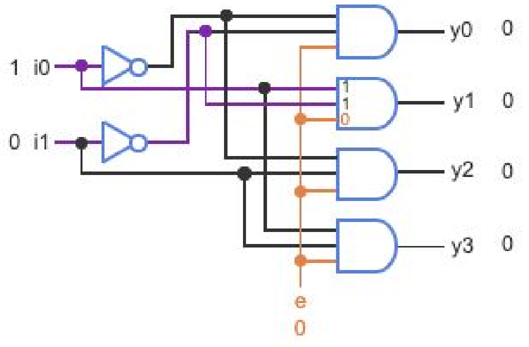


i1	i0	y3	y2	y1	y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$y0 = i1'i0'$
$y1 = i1'i0$
$y2 = i1i0'$
$y3 = i1i0$

-
- Size
    - Input: N, output: 2^N
    - 2^N AND gates, 0 OR gate
- Decoder with enable
    - Some decoders have an additional input called an ***enable input*** that when 0 sets all outputs to 0s, and when 1 enables the decoder for normal behavior.

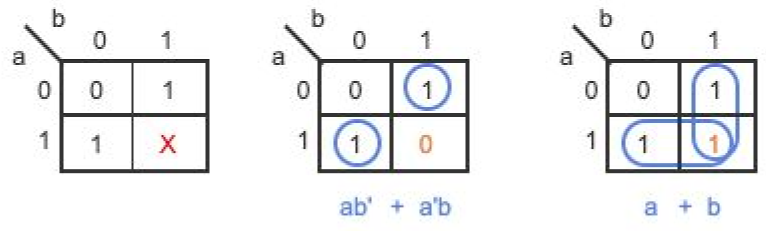




However, when e = 0, all four AND gates output 0, regardless of the values on i1i0, because 0 AND x is 0. The decoder is "disabled".

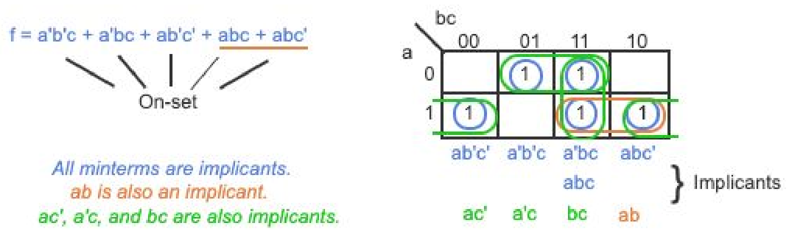
    -

2.10 Don't Cares

- Incompletely specified functions
    - An ***incompletely specified function*** does not define an output value for every input combination
    - Ex: A 3-position knob may set 2 inputs to 00, 01, or 10. Combination 11 is not possible and thus f is not specified for that combination.
    - Note
        - By convention, if only the input value combinations when the output should be 1 are specified, the other combinations are assumed to output 0.
        - "Does not matter" is incomplete because completely specified functions require the output to be either 0 or 1
- MIN with don't care miniterms
    - 

2.11 Prime implicants and minimal covers

- On-sets and implicants
    - On-set:
        - A function's ***on-set*** is the set of minterms that define when the function is 1. Ex: f = a'b'c + a'bc + ab'c' + abc + abc' has 5 minterms in the function's on-set, as listed
    - Cover:
        - A term ***covers*** a minterm if the term evaluates to 1 whenever the minterm does. Ex: Term ab covers minterm abc, as well as minterm abc'
    - Implicants
        - An ***implicant*** of a function is a term that covers only minterms in the function's on-set.
    - Example

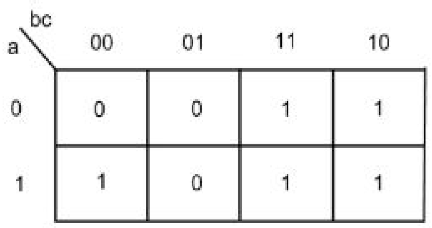


f = a'b'c + a'bc + ab'c' + abc + abc'
On-set
All minterms are implicants.
ab is also an implicant.
ac', a'c, and bc are also implicants.
bc
a
00
01
11
10
ab'c'
a'b'c
a'bc
abc'
abc
Implicants
ac'
a'c
bc
ab

Each valid K-map circle represents an implicant.

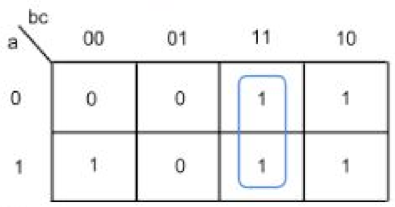
- Prime Implicants
    - A ***prime implicant*** of a function is an implicant that cannot have a literal removed without becoming a non-implicant
    - On a K-map representation of a function, each *largest* possible valid circle represents a *prime* implicant
    - Example

Consider the following K-map for function F = ab'c' + a'bc + abc + a'bc' + abc'.



a \ bc	00	01	11	10
0	0	0	1	1
1	1	0	1	1

1) Does this circle represent a prime implicant?

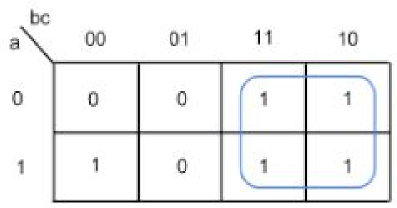


a \ bc	00	01	11	10
0	0	0	1	1
1	1	0	1	1

Yes [✓ No]

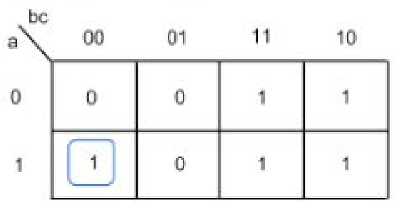
Correct

The circle represents implicant bc, but literal c can be removed. In other words, the circle can be enlarged to cover the four rightmost cells, yielding b. No additional literal can be removed, so b is a prime implicant.



a \ bc	00	01	11	10
0	0	0	1	1
1	1	0	1	1

2) Does this circle represent a prime implicant?

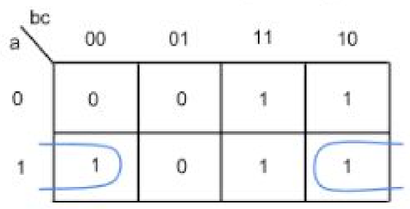


a \ bc	00	01	11	10
0	0	0	1	1
1	1	0	1	1

Yes [✓ No]

Correct

The circle represents implicant ab'c', but literal b' can be removed. In other words, the circle can be enlarged to cover the bottom left and bottom right cells (which are adjacent in a K-map), yielding implicant ac'. No additional literal can be removed, so ac' is a prime implicant.



a \ bc	00	01	11	10
0	0	0	1	1
1	1	0	1	1

- **Essential prime implicants and minimal covers**
    - An ***essential prime implicant*** is the only prime implicant to cover a particular minterm in a function's on-set.
    - On a K-map, an essential PI is a largest circle that is the only circle to cover a particular 1.

    - Essential PI's must be in the function's cover.

<u>**2.12 Quine-McCluskey**</u>

- an algorithm for two-level logic optimization, suitable for computer automation due to using a tabular method (rather than graphical method like K-maps). Given a function's minterms, the algorithm's steps are:
    1. *Generate PI's:* Create a table of minterms, then pairwise check minterms for i(j + j') opportunities, combining into new terms in a new column, repeating with new terms until no more combinations can be made. Each term that wasn't combined with another (minterms or new terms) is a prime implicant (PI).
    2. *Find essentials:* Draw a table with PI's as rows and minterms as columns, putting a mark to indicate a PI covers a minterm. For any column with only one mark, the PI for that row is essential so is added to the cover. All minterms covered by that PI are also checked off as covered.
    3. *Cover remaining:* Select minimal unadded prime implicants to cover remaining minterms
-