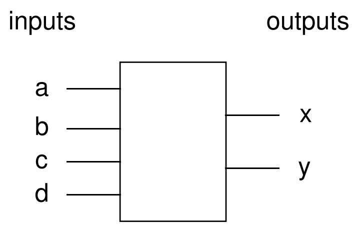
Basic Combinational Circuit

- A *combinational circuit*
  - Maps a set of inputs to a set of outputs
    - These inputs / outputs are called *signals*
    - Each signal has the value 1 or 0



  - Whenever the input values change, the output values change after a short delay (called the *propagation delay*)
  - Is purely functional
    - Output values depend only on inputs
    - There is no concept of *state* in a combinational circuit

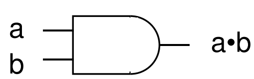
Boolean Operations

- In Boolean algebra, variables assume one of two logical values:
  - True = 1
  - False = 0
- Boolean algebra has three basic operators:
  - *and*, denoted as • (sometimes omitted)
  - *or*, denoted as +
  - *not* (or *complement*, or *inversion*), denoted as ' or $\overline{}$
- A *gate* is an electronic device that implements a simple Boolean operation
  - Gates are the basic building blocks of a digital circuit

4 Basic Boolean Operations & Gates

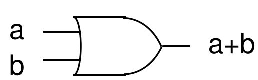
and

a	b	a•b (or ab)
0	0	0
0	1	0
1	0	0
1	1	1



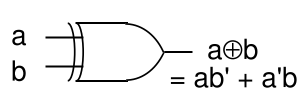
or

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1



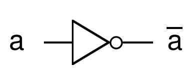
xor

a	b	a⊕b
0	0	0
0	1	1
1	0	1
1	1	0



not

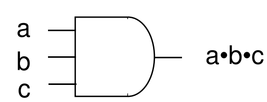
a	$\overline{a}$ (or a')
0	1
1	0



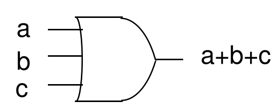
Generalized Gates

- These gates can also be generalized to an arbitrary number of inputs

a	b	c	a•b•c
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



a	b	c	a+b+c
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

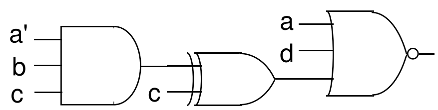


Building Combinational Circuits

- Consider the 4-input Boolean expression:

  $z = ((a'bc \oplus c) + a + d)'$

- The equivalent circuit (logic) diagram would be:



Conventions in drawing logic diagrams:

- Inputs go on the left, outputs on the right
- Simplify inputs: don’t show inverters
- Signal lines are drawn in rectangular fashion (i.e., no diagonals, no curves)
- It’s OK for signal lines to cross; but if signal lines merge / split, indicate the junction with a black dot

Boolean Algebra

- The 10 fundamental properties are:
  - *commutative*

    $x + y = y + x$ $\qquad$ $x \bullet y = y \bullet x$
  - *associative*

    $x + (y + z) = (x + y) + z$

    $x \bullet (y \bullet z) = (x \bullet y) \bullet z$
  - *distributive*

    $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$

    $x + (y \bullet z) = (x + y) \bullet (x + z)$
  - *identity*

    $x + 0 = x$ $\qquad$ $x \bullet 1 = x$
  - *complement*

    $x + x' = 1$ $\qquad$ $x \bullet x' = 0$

- Each property has a *dual* property, created by:
  - Exchanging + and •
  - Exchanging 1 and 0

Boolean Algebra (cont.)

- These postulates can be used to prove a number of useful theorems:
  - *idempotency*

    $x + x = x$

    $x \bullet x = x$
  - *null*

    $x + 1 = 1$

    $x \bullet 0 = 0$
  - *absorption*

    $x + (x \bullet y) = x$

    $x \bullet (x + y) = x$
  - *De Morgan’s law*

    $(x + y)' = x' \bullet y'$

    $(x \bullet y)' = x' + y'$
  - *other complement theorems*

    $(x')' = x$ $\qquad$ *involution*

    $1' = 0$

    $0' = 1$

2-Level Circuits

- ***Every*** Boolean expression can be written as ***a sum of products***, and as a ***product of sums***, and implemented as a 2-level circuit

$z = ((a'bc \oplus c) + a + d)'$	given
$= (a'bcc' + (a'bc)'c + a + d)'$	defn of $\oplus$
$= ((a'bc)'c + a + d)'$	complem.
$= ((a+b'+c')c + a + d)'$	De Morg.
$= (ac + b'c + a + d)'$	distributive
$= (ac)'(b'c)'a'd'$	De Morg.
$= (a'+c')(b+c')a'd'$	De Morg.
$= (a'+a'c')(b+c')d'$	distributive
$= (a'b + a'c' + a'bc' + a'c')d'$	distributive
$= (a'b + a'c' + a'bc')d'$	x+x=x
$= a'bd' + a'c'd' + a'bc'd'$	distributive

(this result is sum of products form)

Truth Tables

- A truth table specifies *what* a logic circuit does, but not *how* it does it
  - It lists, for every possible combination of input values, the output value(s)

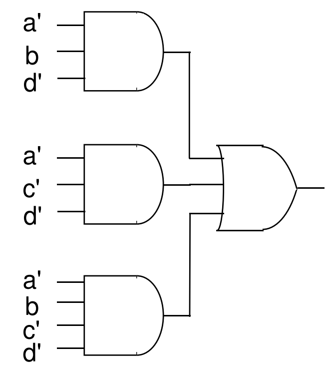
a	b	c	d	x
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

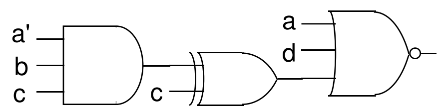
- Our goal is to build the digital circuit that implements this truth table

Implementing a Truth Table

- For a given truth table, there may be more than one valid implementation

a	b	c	d	x
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0





Implementing a Truth Table (cont.)

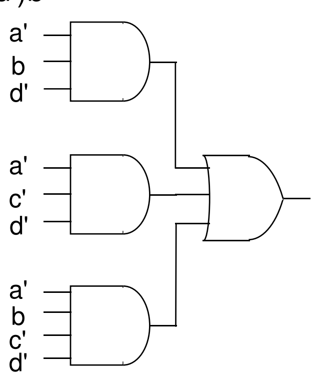
- Unfortunately, there also may be more than one 2-level implementation:

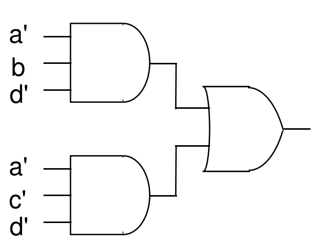
  z = a'bd' + a'c'd' + a'bc'd'

  = a'bd' + (a'c'd') + (a'c'd')b

  = a'bd' + a'c'd'

a	b	c	d	x
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0





3-Variable Karnaugh Maps

- Given a Boolean expression, we can construct a *Karnaugh map* by drawing a box as shown below and writing 1's in all the appropriate boxes whenever that term is true

  z = a'bc + abc + ab

c

	0	0	1	0
a	0	0	1	1

b

- Then we find the *smallest* set of ovals of size 1, 2, 4, or 8, that cover all the 1s (but none of the 0s), and write out the *minimized expression* from those ovals

  z = bc + ab

c

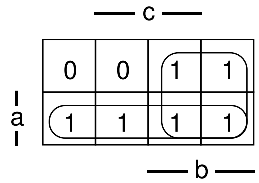
	0	0	1	0
a	0	0	1	1

b

3- Variable Karnaugh Maps (cont.)

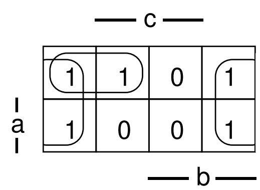
- The ovals should be as big as possible, and two or more ovals can overlap

z = a + b



- Ovals can even wrap around the ends (or top and bottom) of the table

z = c' + a'b'



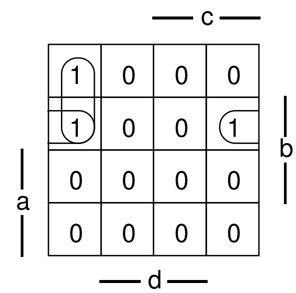
- But ovals can't go diagonally

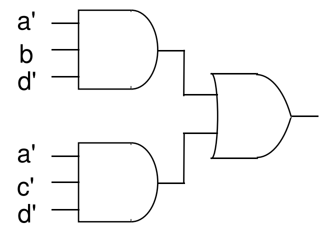
4-Variable Karnaugh Maps

- 4-variable Karnaugh maps are similar, but now ovals can be of size 16 as well

- Now we can finally make sure that we have the minimum 2-level *and-or* (SOP) implementation of our example:

z = ((a'bc ⊕ c) + a + d)'





= a'bd' + a'c'd'

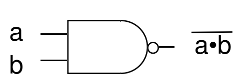
  - Could also build Karnaugh map directly from truth table, entering a “1” in each box that corresponds to a combination of inputs that produce an output of 1

Negation / Universal Gate

- *Nand* and *nor* gates are very easy to construct

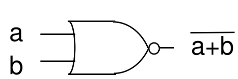
nand

a	b	$\overline{a \bullet b}$
0	0	1
0	1	1
1	0	1
1	1	0

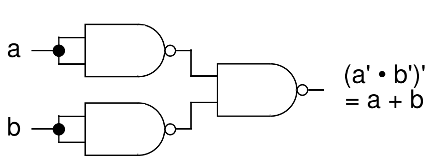


nor

a	b	$\overline{a+b}$
0	0	1
0	1	0
1	0	0
1	1	0

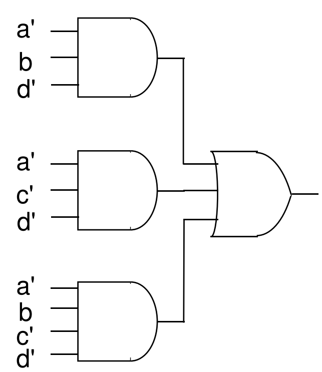


- Any digital circuit can be implemented using only *nand* or *nor* gates

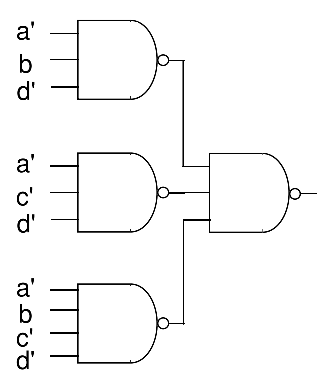


Example from Earlier in the Lecture

- In 2-level *and-or* form:



- In 2-level *nand* form:



Half Adder

- Consider what happens when you add two binary digits:

```
   0       0       1       1
  +0      +1      +0      +1
 ----    ----    ----    ----
 0) 0    0) 1    0) 1    1) 0
                 carry   sum
                 out
```

- We can construct a truth table for a *half adder* (HA) — a device that adds two binary digits *a* and *b*, producing a *sum* and a *carry out*

a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Implementing a Half Adder

- A half adder can be implemented directly in 2-level SOP form:

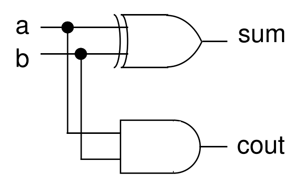
		– b –
	0	1
a	1	0

sum = ab' + a'b

		– b –
	0	0
a	0	1

cout = ab

- A half adder can also be implemented using an **xor** gate:



- Which implementation is better? Why?

Full Adder

- A half adder is fine for the *least significant bit* (LSB) of a multi-bit number, but not the other bits

```
                         carry
                         in
           1       1       1
   00     01      01      11
  +01    +01     +11     +11
  ----   ----    ----    ----
 0) 01  0) 10   1) 00   1) 10
                carry   sum
                out
```

- A *full adder* (FA) adds two binary digits *a* and *b*, plus a *carry in* (from the previous digit), producing a *sum* and a *carry out*

a	b	cin	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder (cont.)

- A full adder can be implemented directly in 2-level SOP form:

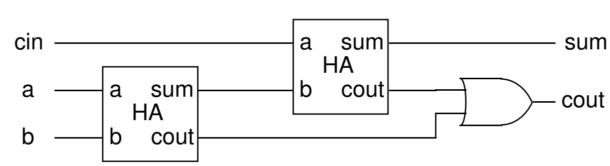
		— cin —		
0	0	1	0	
0	1	1	1	a
		— b —		

cout = a•cin + b•cin + ab

		— cin —		
0	1	0	1	
1	0	1	0	a
		— b —		

sum = ab'cin' + a'b'cin + abcin + a'bcin'

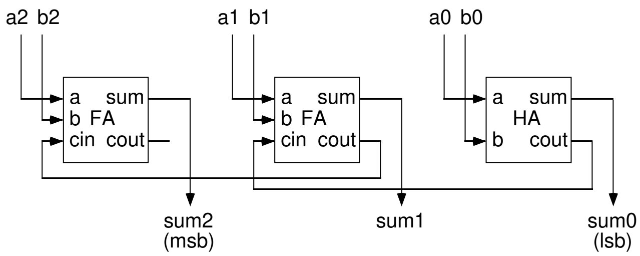
- A full adder can also be implemented by two half adders:



- Again, which implementation is better, and why? Various alternatives left as an exercise for anyone interested…

n-Bit Adder

- An *n*-bit adder can now be constructed out of *n*-1 full adders, and 1 half adder



- This kind of *n*-bit adder is called a *ripple adder*
  - Why?
  - What are its limitations?

Which Implementation of the Half Adder is Better?

- Given this number of transistors and amount of propagation delay (in ns)

	2-input		3-input		4-input	
and	6	2.4	8	2.8	10	3.2
or	6	2.4	8	2.8	10	3.2
xor	14	4.2				
nand	4	1.4	8	1.8	10	2.2
nor	4	1.4	8	1.8	10.	2.2

inverter (1-input) 2 1.0

- In SOP form, using and and or gates:
  - sum = ___ transistors,
    cout = ___ transistors, total ___
  - Max delay until outputs are ready = ___ns
- In SOP form, using nand gates:
- In xor form (using a single complex gate instead of three simple gates):