Bitwise Addition;
Gates and Circuits

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

It's All Bits!

- Another example: text

`'terriers'`

⇩

`0111010001100101011100100111001001101001011001010111001001110011`

8 ASCII characters, 8 bits each → 64 bits

- *All* types of data are represented in binary.
  - images, sounds, movies, floating-point numbers, etc...
- ***All computation*** involves manipulating bits!

It's All Bits! (cont.)

- Example: to add 42 + 9, the computer does *bitwise addition*:

```
   1
 101010
+ 001001
 ------
 110011
```

- In PS 4, you'll write a Python function for this.

```
add_bitwise('101010', '001001')
```

PS 4: add_bitwise

```
   1
 101010
+ 001001
 ------
 110011
```

- `add_bitwise(b1, b2)`
  b1 and b2 are *strings* representing binary #s

- It should look something like this:

```
def add_bitwise(b1, b2):
    if ...      # base case #1

    elif ...    # base case #2

    else:       # recursive case
        sum_rest = add_bitwise(b1[:-1],b2[:-1])
        if ...
              # rest of recursive case
```

- Let's trace through a concrete case:
  `add_bitwise('100', '010')`

How recursion works: `add_bitwise(b1, b2)`

- Recall: we get a separate stack frame for each call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
```

```
def add_bitwise(b1, b2):
    if ...       # base case #1

    elif ...     # base case #2

    else:        # recursive case
        sum_rest = add_bitwise(b1[:-1],b2[:-1])
        if ...
             # rest of recursive case
```

How recursion works: `add_bitwise(b1, b2)`

- Recall: we get a separate stack frame for each call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
def add_bitwise(b1, b2):
    if ...       # base case #1

    elif ...     # base case #2

    else:        # recursive case
        sum_rest = add_bitwise(b1[:-1],b2[:-1])
        if ...
             # rest of recursive case
```

How recursion works: `add_bitwise(b1, b2)`

- Recall: we get a separate stack frame for each call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'   b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''   b2: ''
base case: return ''
```

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'   b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''   b2: ''
base case: return ''
```

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'   b2: '0'
sum_rest = ''
```

- It replaces the recursive call.

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'   b2: '0'
sum_rest = ''
if ...
    return sum_rest + '1'
                 ''    + '1'
                     '1'
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = '1'
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'   b2: '01'
sum_rest = '1'
if ...
    return sum_rest + '1'
              '1'   + '1'
                '11'
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = '11'
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

How recursion works: `add_bitwise(b1, b2)`

- Each return value is sent back to the previous call.

```
add_bitwise('100', '010')
b1: '100'   b2: '010'
sum_rest = '11'
if ...
    return sum_rest + '0'
             '11'    + '0'
                 '110'
```

- It replaces the recursive call.
- We use it to build the next return value, and thus gradually build solutions to larger problems.

How recursion works: `add_bitwise(b1, b2)`

- Final solution!

`add_bitwise('100', '010')` → `'110'`

The Tricky Part of `add_bitwise(b1, b2)`

- What if we had this instead?

`add_bitwise('101', '011')`

The Tricky Part of `add_bitwise(b1, b2)`

- We again end up with a series of recursive calls:

```
add_bitwise('101', '011')
b1: '101'    b2: '011'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'    b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'    b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''    b2: ''
base case: return ''
```

The Tricky Part of `add_bitwise(b1, b2)`

- We again build our solution on our way back from the base case:

```
add_bitwise('101', '011')
b1: '101'    b2: '011'
sum_rest = add_bitwise('10', '01')
```

```
add_bitwise('10', '01')
b1: '10'    b2: '01'
sum_rest = add_bitwise('1', '0')
```

```
add_bitwise('1', '0')
b1: '1'    b2: '0'
sum_rest = add_bitwise('', '')
```

```
add_bitwise('', '')
b1: ''    b2: ''
base case: return ''
```

The Tricky Part of `add_bitwise(b1, b2)`

- **What do we need to do differently here?**

```
add_bitwise('101', '011')
b1: '101'   b2: '011'
sum_rest = '11'  # same as before
if ...
    ???
```

```
   101
+  011
   ---
    11
```

The Tricky Part of `add_bitwise(b1, b2)`

- **What do we need to do differently here?**

```
add_bitwise('101', '011')
b1: '101'   b2: '011'
sum_rest = '11'  # same as before
if ...
    ???
```

- We need to carry!

```
     1
   101
+  011
   ---
   110
```

The Tricky Part of `add_bitwise(b1, b2)`

- **What do we need to do differently here?**

```
add_bitwise('101', '011')
b1: '101'    b2: '011'
sum_rest = '11'   # same as before
if ...
    ???
```

- We need to carry!

```
       1
     101
  +  011
  ------
     110
      ↓
    1000
```

- We need to add 11 + 1 to get 100.
  - how can we do this addition? call `add_bitwise` recursively a second time!

It's All Bits! (cont.)

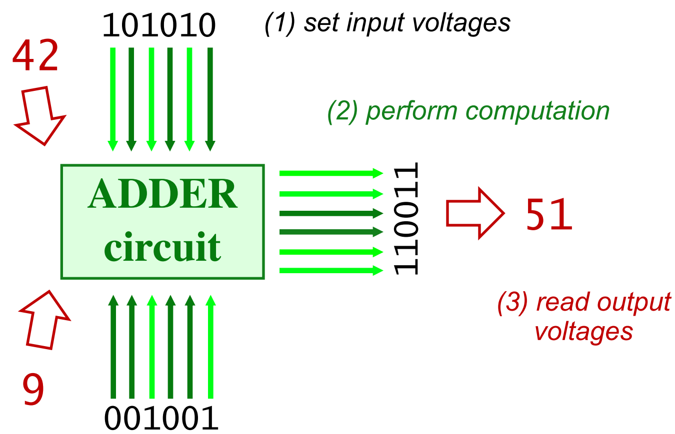
- Example: to add 42 + 9, the computer does *bitwise addition*:

```
      1
   101010
 + 001001
 --------
   110011
```

- In PS 4, you'll write a Python function for this.
  `add_bitwise('101010', '001001')`
- ***In PS 5, you'll design a circuit for it!***

How Computation Works

- In a computer, each bit is represented as a *voltage.*
  - 1 is +5 volts, 0 is 0 volts
- Computation is the deliberate combination of those voltages!



All Computation Involves *Functions* of Bits!

binary inputs **A** and **B** → output, **A+B**

bitwise addition function

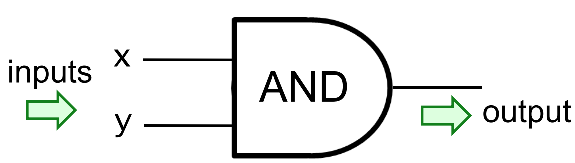
A	B	A+B
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

Bits as Boolean Values

- When designing a circuit, we think of bits as boolean values:
  - 1 = True
  - 0 = False
- In Python, we've used *logic operators* (`and`, `or`, `not`) to build up boolean expressions.
- In circuits, there are corresponding *logic gates*.

AND Gate (with two inputs)

AND outputs 1 only
if ***all*** inputs are 1



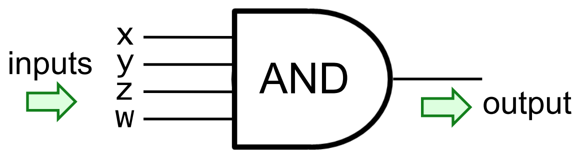
AND's function:

inputs		output
x	y	`AND(x,y)`
0	0	0
0	1	0
1	0	0
1	1	1

truth table

AND Gate (with *four* inputs)

AND outputs 1 only
if ***all*** inputs are 1



AND's function:

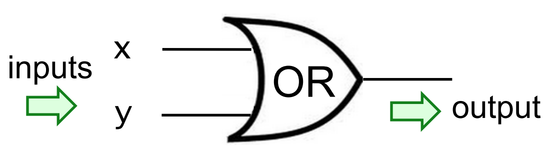
inputs				output
x	y	z	w	AND(x,y,z,w)
0	0	0	0	0
0	0	0	1	0
…12 more rows not shown…				0
1	1	1	0	0
1	1	1	1	1

fifteen 0s

one 1

OR Gate (with two inputs)

OR outputs 1 if
any input is 1

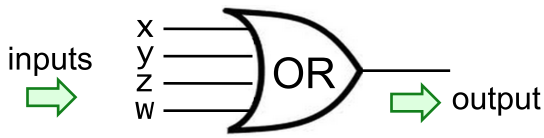


OR's function:

inputs		output
x	y	OR(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

OR Gate (with *four* inputs)

OR outputs 1 if
any input is 1

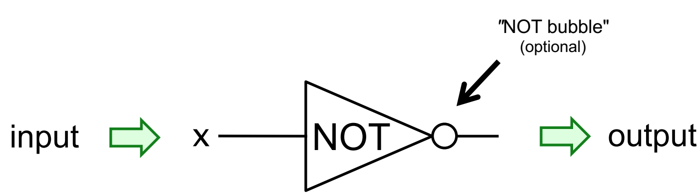


OR's function:

inputs				output	
x	y	z	w	OR(x,y,z,w)	
0	0	0	0	0	one 0
0	0	0	1	1	
…12 more rows not shown…				1	
1	1	1	0	1	
1	1	1	1	1	fifteen 1s

NOT Gate

NOT reverses
its input



NOT's function:

input	output
x	NOT(x)
0	1
1	0

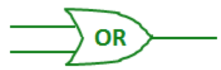
Circuit Building Blocks: Logic Gates

AND outputs 1 only if **ALL** inputs are 1

AND

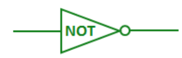
OR outputs 1 if **ANY** input is 1

OR



NOT reverses its input

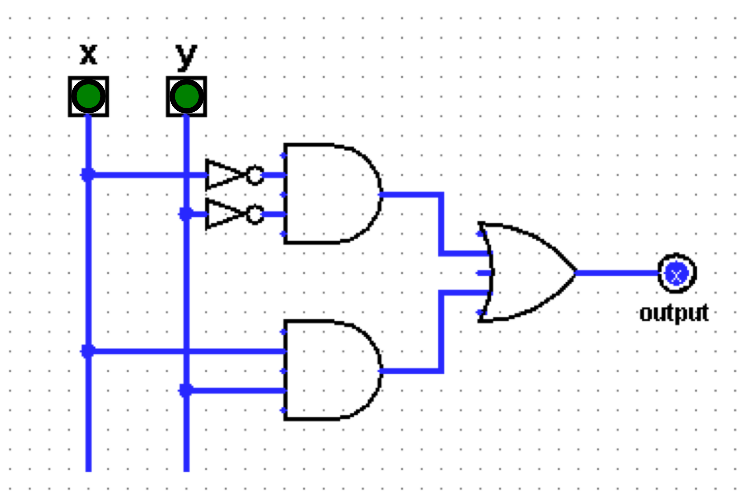
NOT



- They each define a boolean function – a function of bits!
  - take one or more bits as inputs
  - produce the appropriate bit as output
  - the function can be defined by means of a *truth table*

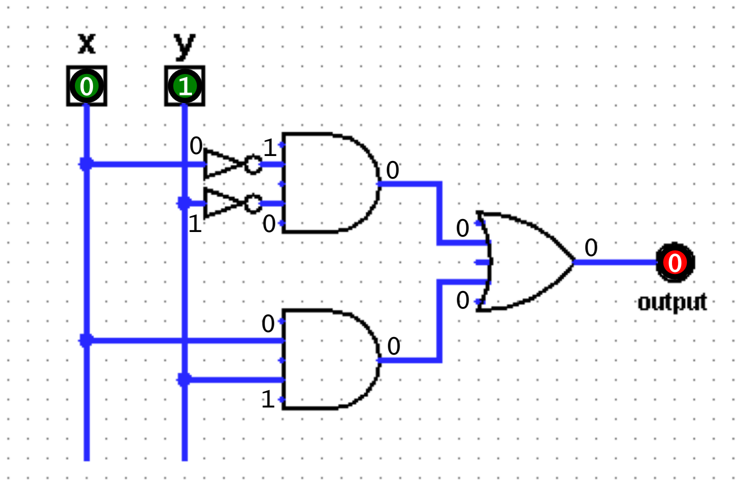
From Gates to Circuits

- We combine logic gates to form larger circuits.



From Gates to Circuits (this was in the video...)

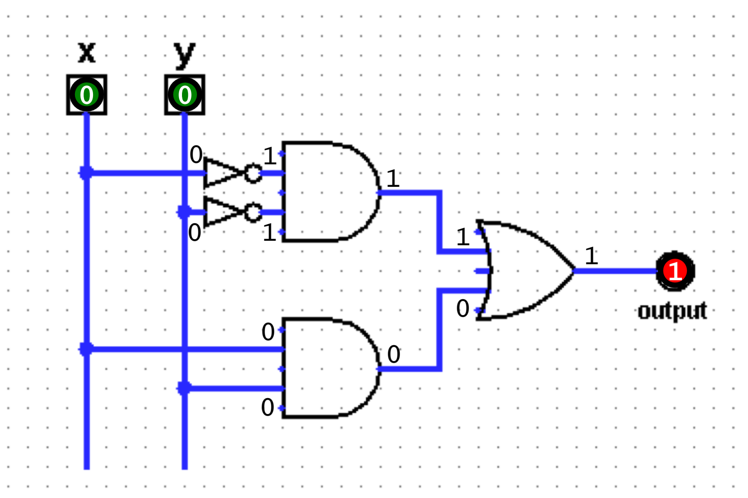
- We combine logic gates to form larger circuits.



- Example: what is the output when x = 0 and y = 1? **0**

From Gates to Circuits (Second Example)

- We combine logic gates to form larger circuits.

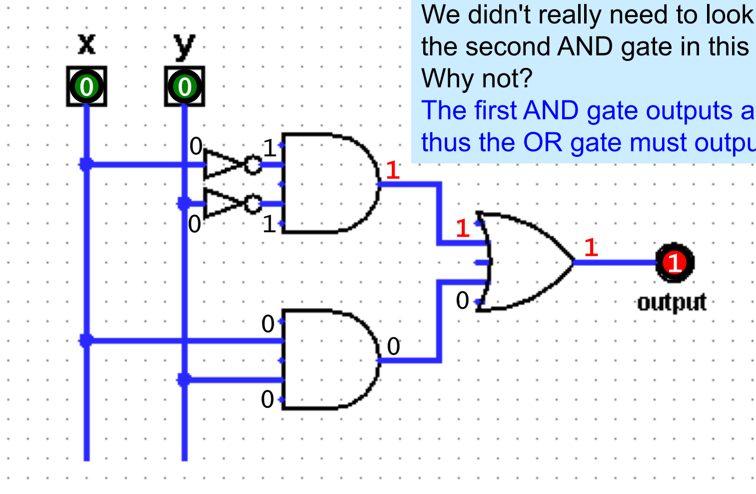


- Example: what is the output when x = 0 and y = **0**? **1**

From Gates to Circuits (Second Example)

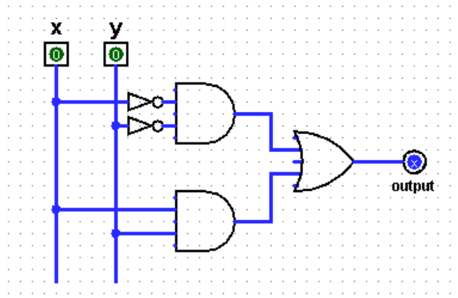
- We combine logic gates to form larger circuits.

We didn't really need to look at the second AND gate in this case. Why not?
The first AND gate outputs a 1, thus the OR gate must output 1.



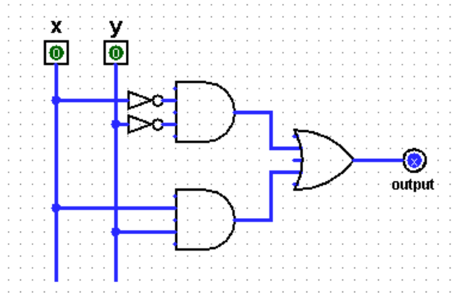
- Example: what is the output when x = 0 and y = **0**? **1**

Which column correctly completes the truth table?



inputs		A.	B.	C.	D.	E.
x	y					
0	0	1	1	0	0	0
0	1	0	0	0	0	0
1	0	0	1	1	0	1
1	1	1	1	0	1	1

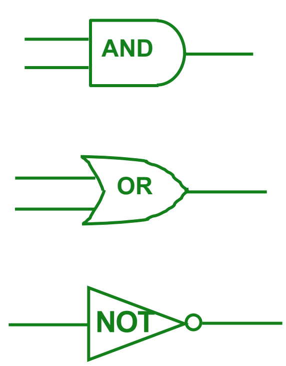
Which column correctly completes the truth table?



inputs			A.	B.	C.	D.	E.
x	y						
0	0		1	1	0	0	0
0	1		0	0	0	0	0
1	0		0	1	1	0	1
1	1		1	1	0	1	1

Claim

We need only these three building blocks to compute anything at all!



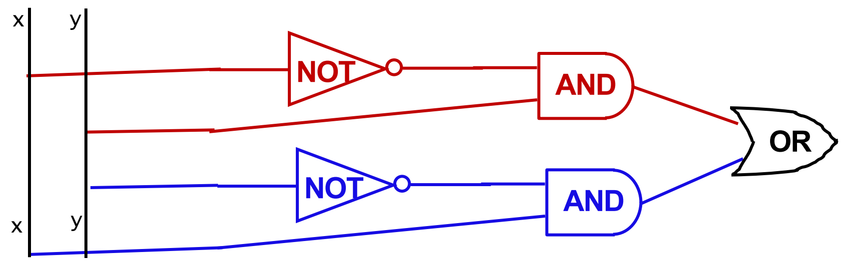
Constructive Proof!

① Specify a **truth table** defining ***any*** function you want.

input		output
x	y	fn(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

② For each input row whose output needs to be 1, build an **AND** circuit that outputs 1 *only for that specific input!*

③ **OR** them all together .



Constructive Proof!

① Specify a **truth table** defining ***any*** function you want.

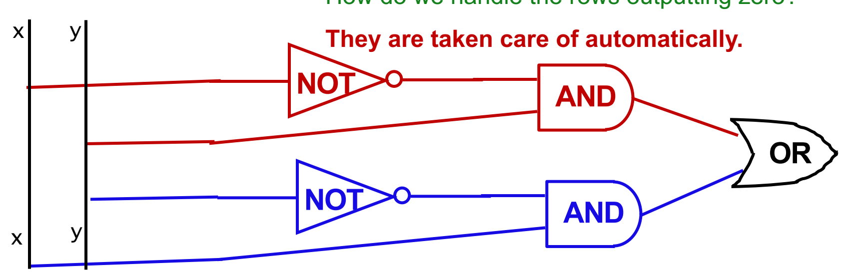
input		output
x	y	fn(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

② For each input row whose output needs to be 1, build an **AND** circuit that outputs 1 *only for that specific input!*

③ **OR** them all together .

How do we handle the rows outputting zero?

They are taken care of automatically.



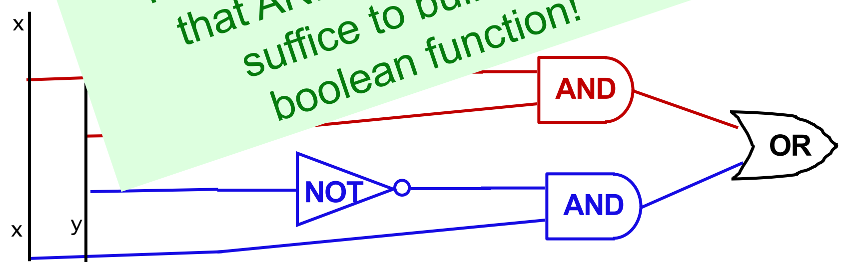
Constructive Proof!

① Specify a **truth table** defining ***any*** function you want.

input		output
x	y	fn(x,y)
0	0	0
0	1	1
1	0	1
1	1	

② For each input row whose output needs to be 1, build an **AND** circuit th t outputs 1 *only for t c input!*

This is a constructive proof that AND, OR, and NOT suffice to build any boolean function!



Constructive Proof!

① Specify a **truth table** defining ***any*** function yo t

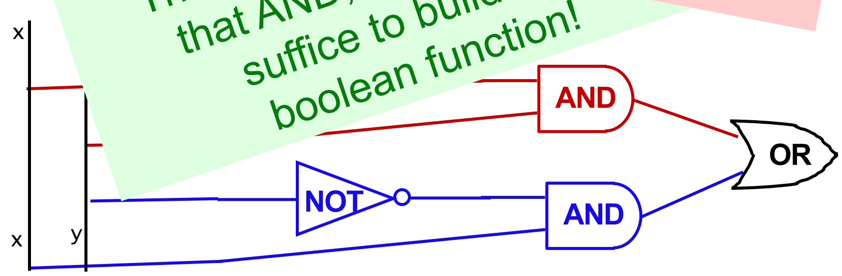
input		
x	y	fn
0	0	
0	1	1
1	0	1
1	1	

② For each input row whose output needs to be 1, build an **AND** circuit th t outputs 1 *c input!*

This is a co that AND, OR, suffice to build a boolean function!

But... **ALL** functions are just boolean functions: because everything is in binary!

?



Boolean Notation

- Recall:

inputs x	inputs y		output x AND y
0	* 0	=	0
0	* 1	=	0
1	* 0	=	0
1	* 1	=	1

inputs x	inputs y		output x OR y
0	+ 0	=	0
0	+ 1	=	1
1	+ 0	=	1
1	+ 1	=	1

1 + 1 = 2, but anything non-0 is considered True, and thus 1 + 1 is equivalent to 1

input x	output NOT x
0	1
1	0

- In boolean notation:
  - x AND y is written as multiplication: xy
  - x OR y is written as addition: $x + y$
  - NOT x is written using a bar: $\bar{x}$

- Example:

(x AND y) OR (x AND (NOT y)) ↔ ??

Boolean Notation

- Recall:

inputs x	inputs y		output x AND y
0	* 0	=	0
0	* 1	=	0
1	* 0	=	0
1	* 1	=	1

inputs x	inputs y		output x OR y
0	+ 0	=	0
0	+ 1	=	1
1	+ 0	=	1
1	+ 1	=	1

1 + 1 = 2, but anything non-0 is considered True, and thus 1 + 1 is equivalent to 1

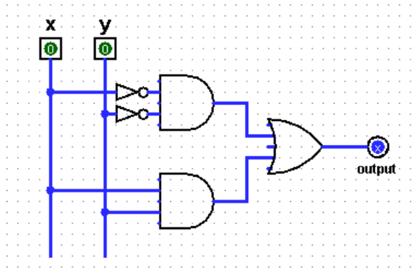
input x	output NOT x
0	1
1	0

- In boolean notation:
  - x AND y is written as multiplication: xy
  - x OR y is written as addition: $x + y$
  - NOT x is written using a bar: $\bar{x}$

- Example:

(x AND y) OR (x AND (NOT y)) ↔ $xy + x\bar{y}$

Boolean Expressions for Truth Tables



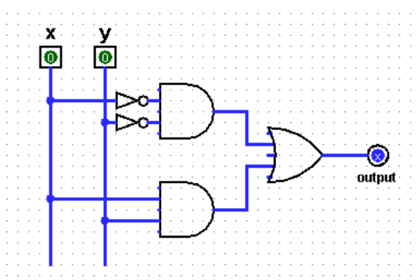
inputs		output
x	y	
0	0	1
0	1	0
1	0	0
1	1	1

- This truth table/circuit can be summarized by the expression:

$$\bar{x}\bar{y} + xy$$

inputs		output	$\bar{x}\bar{y} + xy$
x	y		
0	0	**1**	1*1 + 0*0 = **1**
0	1	**0**	1*0 + 0*1 = **0**
1	0	**0**	0*1 + 1*0 = **0**
1	1	**1**	0*0 + 1*1 = **1**

Boolean Expressions for Truth Tables



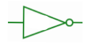
inputs		output
x	y	
0	0	1
0	1	0
1	0	0
1	1	1

- This truth table/circuit can be summarized by the expression:

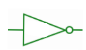
$$\bar{x}\bar{y} + xy$$

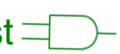
- This expression is the *minterm expansion* of this truth table.
  - one *minterm* for each row that has an output of 1
  - combined using OR

Two Helpful Suggestions

Do  f  get
to get started on PS 4!

Two Helpful Suggestions

Do  f  get
to get started on PS 4!

Make sure you underst 
the material covered through last Friday,
and begin preparing for next week's midterm!