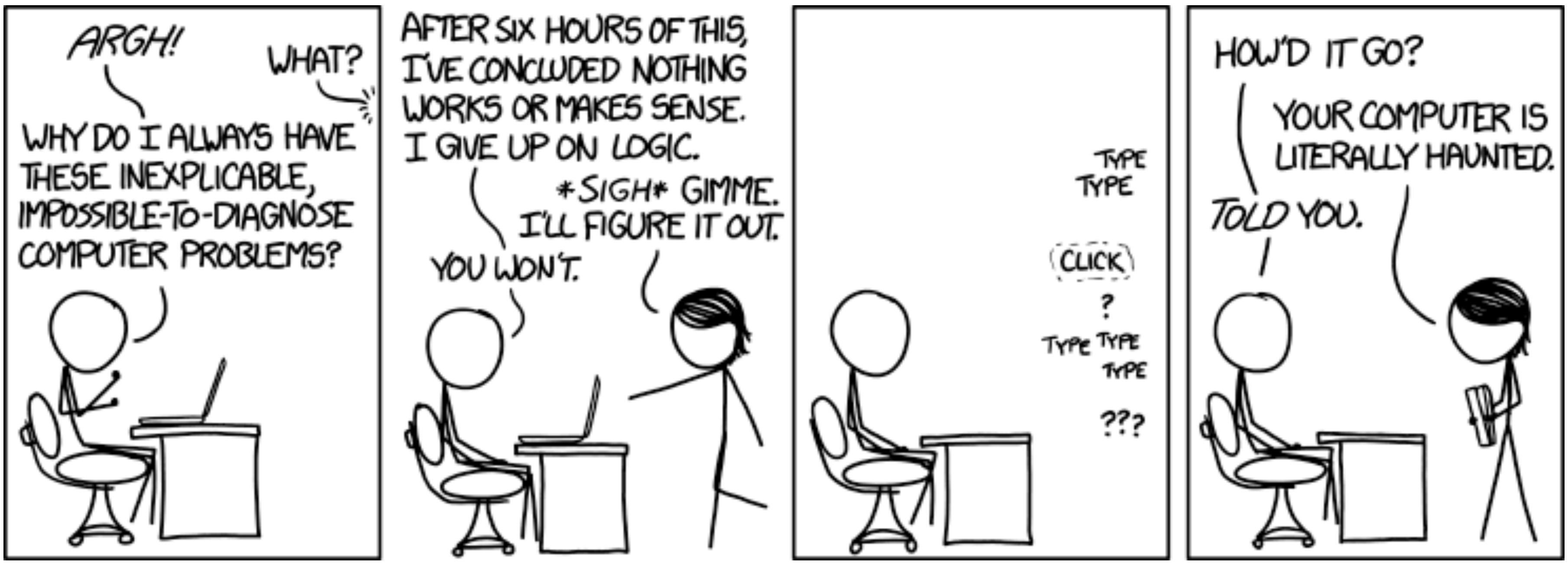
Three week "detour," featuring ...



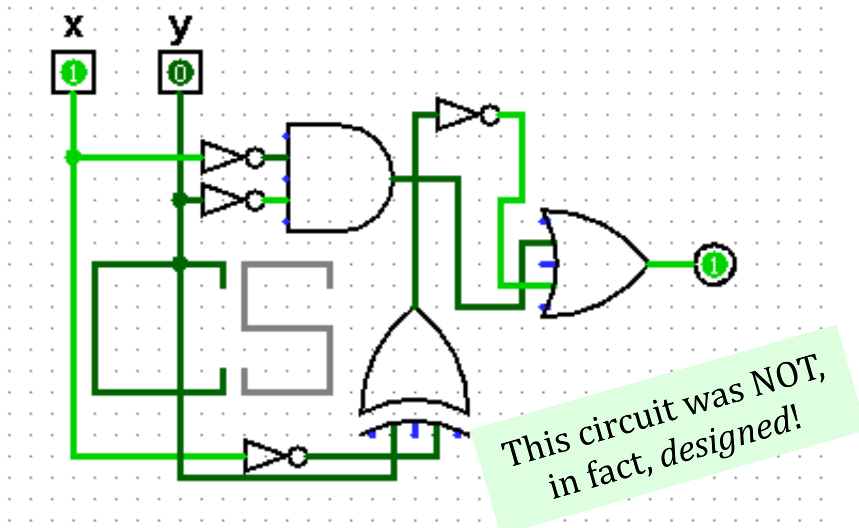
... the <u>ghost</u> in the machine

OK Recursing?
OKer than before?

Hw#4: binary + Python

CS || Today

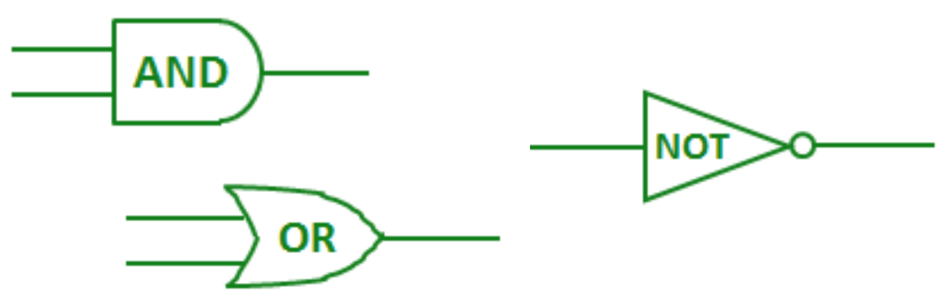
Circuit design, part 1



Minterm Expansion Principle

That's *mi<u>n</u>term*, **NOT** *mi<u>d</u>term*

A circuit for ***any function*** can be built from ...



... ***just*** these three logic gates!

Last week's solutions

too compressed, admittedly!

```
def blsort(L):
    """ returns a sorted version of L
        (L has only 1's and 0's)
    """
    return count(0,L)*[0] + count (1,L)*[1]

def decipher (S):
    """ input: string that has been shifted
        output: English rotation of S
    """
    L = [ encipher(S,n) for n in range(26) ]
    LoL = [ [wordProb(x),x] for x in L ]
    bestpr = max(LoL)
    return bestpr[1]

def gensort(L):
    """ returns a sorted version of the list L
    """
    if len(L) == 0: return L
    else:
        m = min(L)
        R=remOne(m,L)
        return [m] + gensort(R)

def jscore(S,T):
    """ returns the jotto score of S vs. T
    """
    if S == '' or T == '': return 0
    elif S[0] in T:
        return 1 + jscore(S[1:],remOne(S[0],T))
    else: return jscore(S[1:],T)
```

```
def exact_change(t,L):
    """ returns whether t can be made by summing el's in L
    """
    if t==0: return True
    elif t<0 or L==[]: return False
    else:
        useit=exact_change(t-L[0],L[1:])
        loseit=exact_change(t,L[1:])
        return useit or loseit

def LCS (S,T):
    """ returns the longest common subseq of S and T
    """
    if S == '' or T=='': return ''
    elif S[0]==T[0]: return S[0]+LCS(S[1:],T[1:])
    else:
        result1 = LCS(S[1:], T)
        result2 = LCS(S, T[1:])
        if len(result1) < len(result2): return result2
        else: return result1

def make_change(t,L):
    """ returns how t can be made by summing el's from L
        or False, if it's not possible...
    """
    if t==0: return []
    elif t<0 or L==[]: return False
    else:
        useit=make_change(t-L[0],L[1:])
        loseit=make_change(t,L[1:])
        if useit == False: return loseit
        useit = L[0:1] + useit
        return useit
```

too recursive?

Creativity with Caesar...

```
def decipher( S ):
    """ TESIJHYDW - je tusyfxuh
        jxyi tesijhydw, zkij hkd
        tusyfxuh ed yj.    """
```

... code here ...

Creativity with Caesar...

```
def decipher( S ):
    """ DOCSTRING - to decipher
        this docstring, just run
        decipher on it.    """
```

... code here ...

my ***favorite*** not-fully-working **`decipher`**...

Creativity with Caesar...

```
def decipher( S ):
    """ This works sometimes
    """
    return encipher( S, 3 )
```

and the docstring
is 100% correct!

This week

Circuits!

```
def
                    sometimes
```

```
return encipher( S, 3 )
```

Designing physical devices that work ***all the time!***

This week

Circuits!

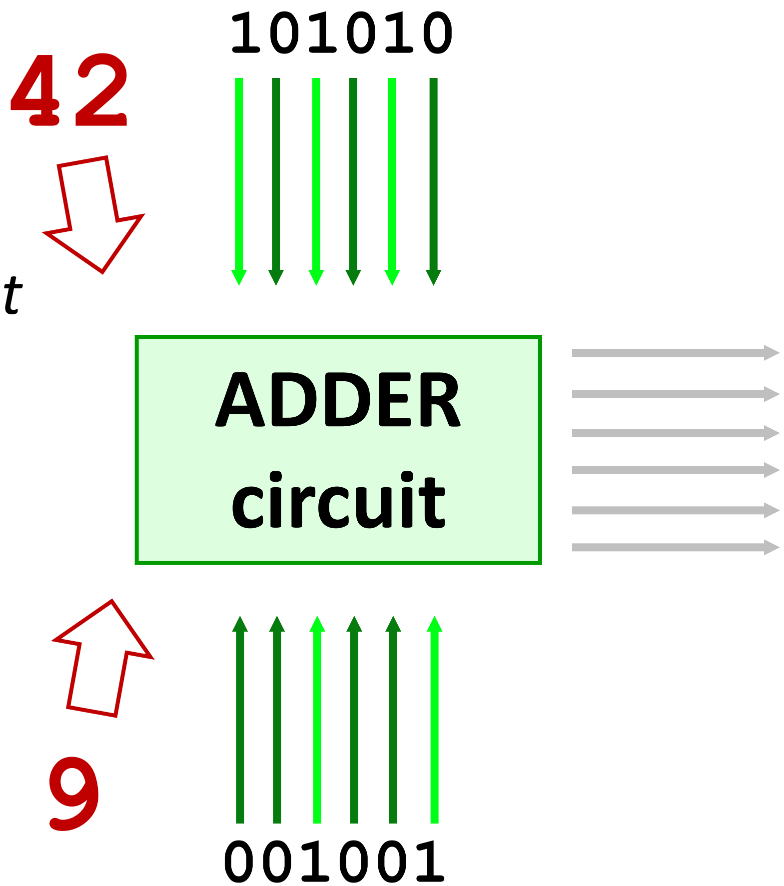
```
def
              ...ns sometimes
    return encipher( S, 3 )
```

The big picture...

In a computer, each bit is represented as a <u>voltage</u> (**1** is +5v and **0** is 0v)

Computation is simply the **deliberate combination** of those voltages!

(1) set input voltages

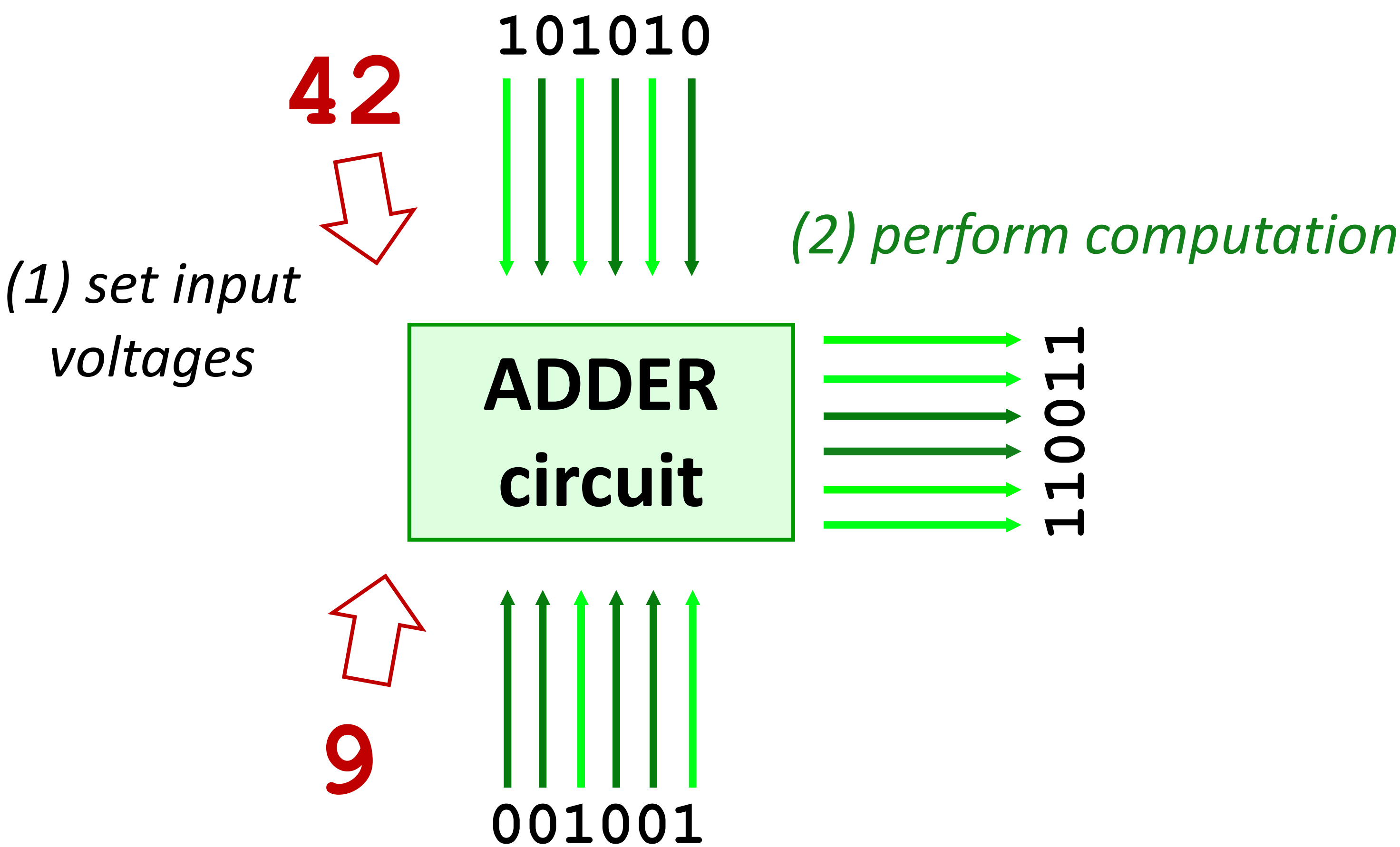


What's in that green box?

The big picture...

In a computer, each bit is represented as a <u>voltage</u> (**1** is +5v and **0** is 0v)

Computation is simply the **deliberate combination** of those voltages!

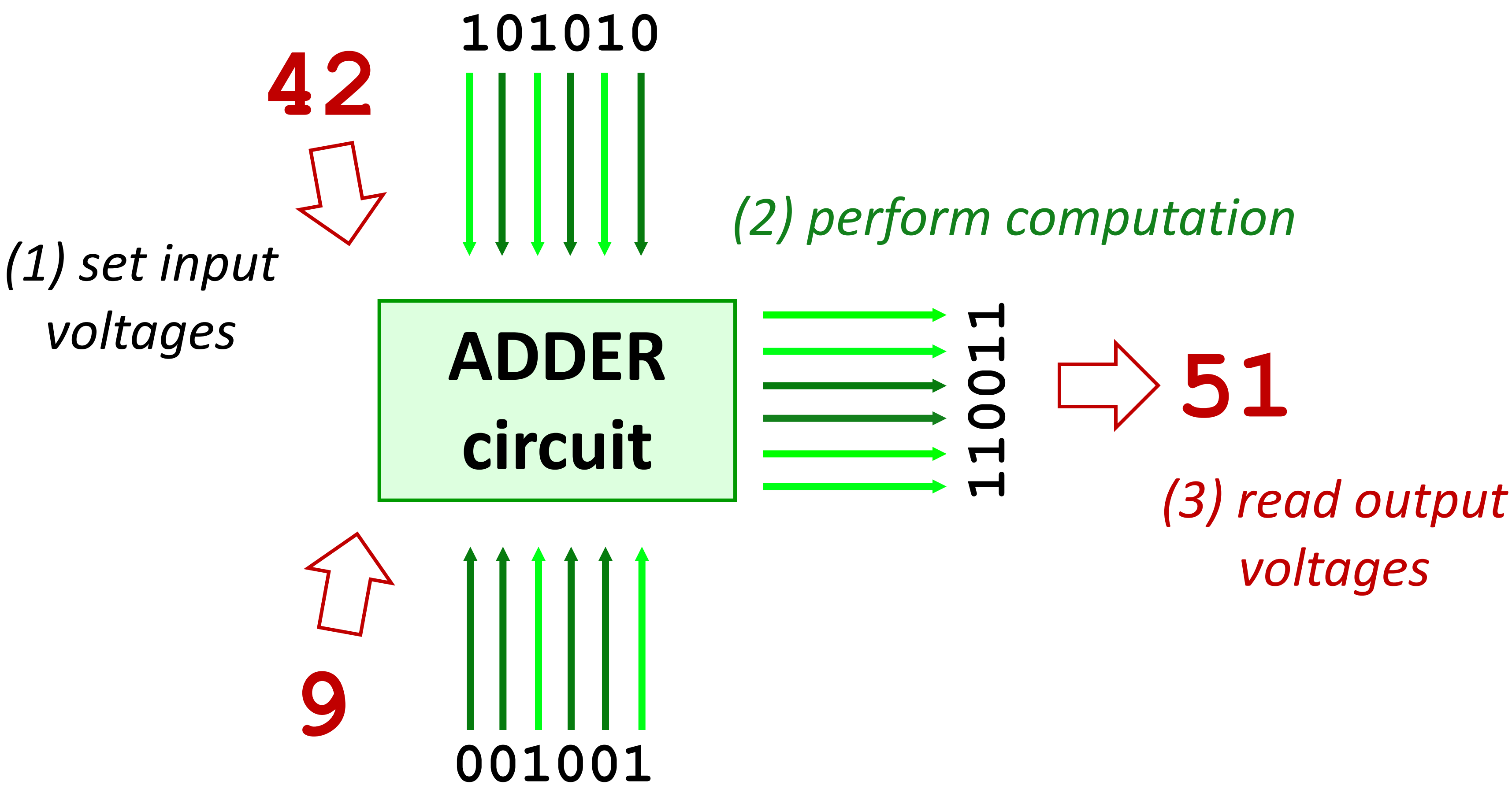


What's in that green box?

The big picture...

In a computer, each bit is represented as a <u>voltage</u> (**1** is +5v and **0** is 0v)

Computation is simply the **deliberate combination** of those voltages!



What's in that green box?

Richard Feynman: *"Computation is just a physics experiment that always works!"*

All computations… … are ***functions of bits***

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

You built this in Python last week as **addB** *purely syntactically*

A	B	add
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	[illegible]	011
[illegible]	[illegible]	[illegible]
[illegible]	10	[illegible]
11	11	110

addB

bitwise addition function

four bits in…

…three bits out

This week, you'll build the **addB** function in **Circuitverse**

Motivation: A function we want...

"three-bit" adder

3 bits of input

```
   1
   0
+  1
----
  10
```

2 bits of output

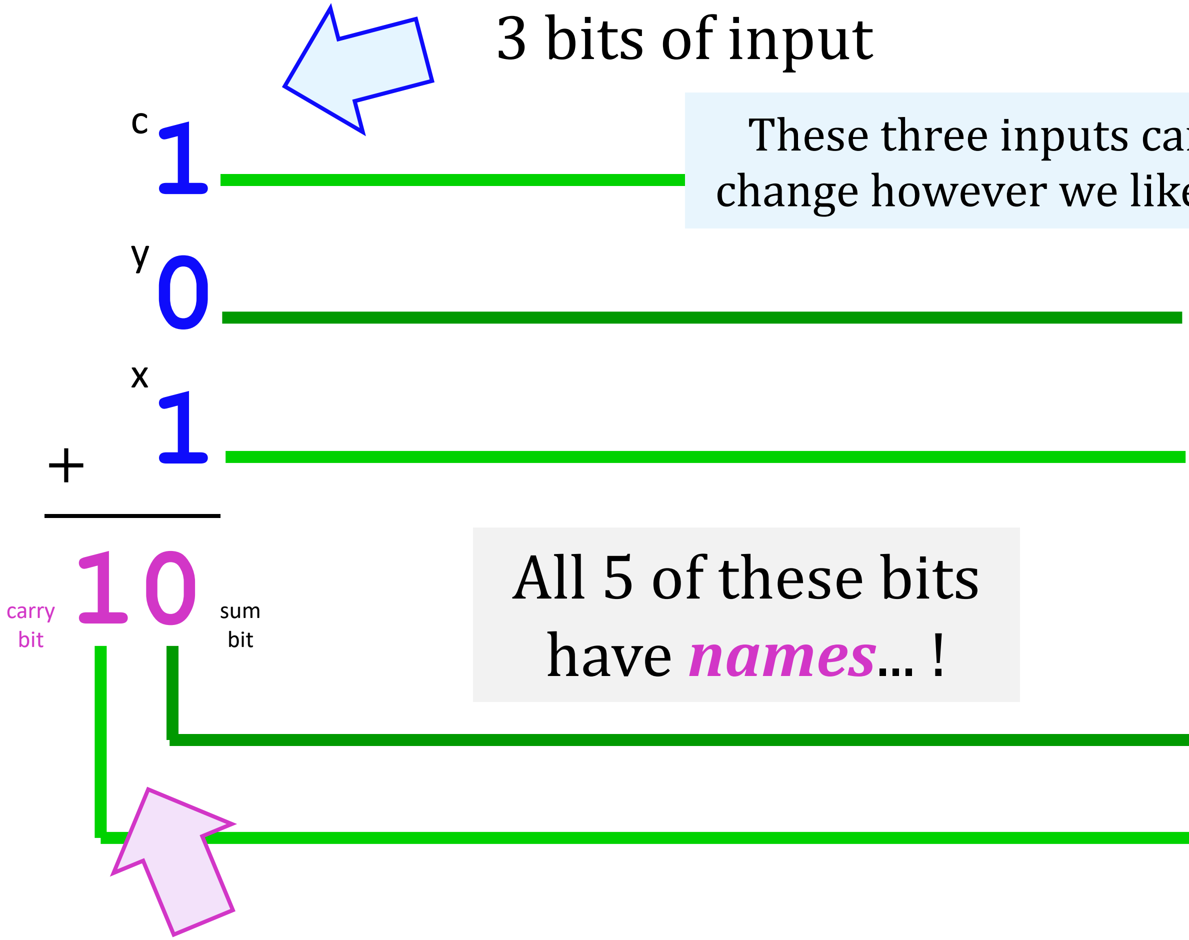
All 5 of these bits have ***names***... !

What! Why do these bits get individual names?!

Motivation: A function we <u>want</u>...

"three-bit" adder

3 bits of input



These three inputs can change however we like ...

All 5 of these bits have ***names***... !

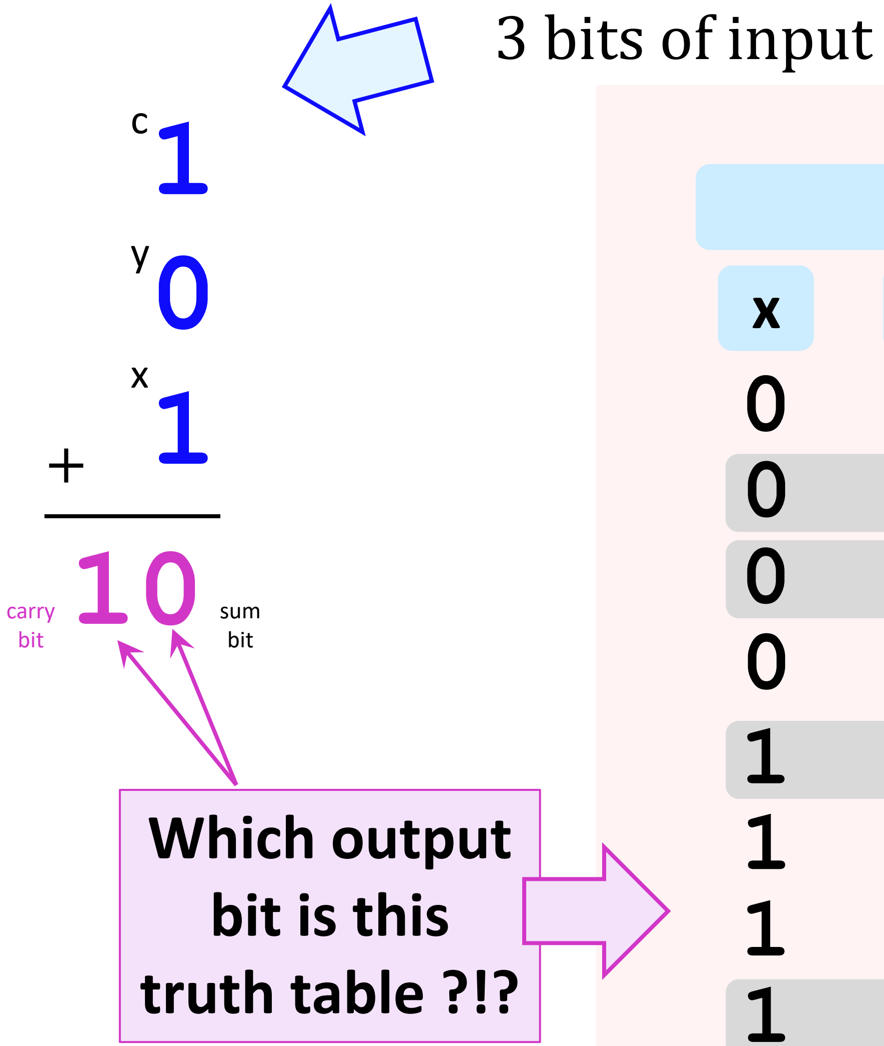
Because each is an individual *wire*!

What! Why do these bits get individual names?!

2 bits of output

... but these two output bits will have to change to be correct.

Truth table



IN			OUT
x	y	c	circuit output
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

Truth table

3 bits of input

1
0
+ 1
---
1 0

carry bit: 1 — "sum" bit: 0

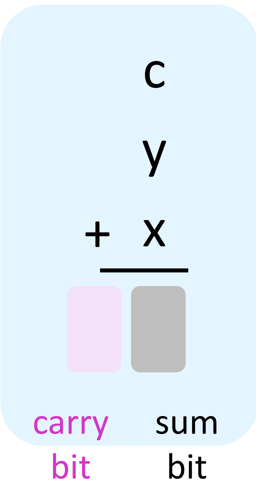
"sum" bit

Which output bit is this truth table ?!?

IN			OUT
x	y	c	circuit output ("sum" bit)
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

Part 1: Represent your function as ***bits***...

Any function can be represented using only bits...



IN			OUT
x	**y**	**c**	**circuit output**
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

This one is named the **sum** function

That's some function, all right!

three bits in...

...one bit out

Truth table

3 bits of input

c
x
\+ y

carry bit ? ? sum bit

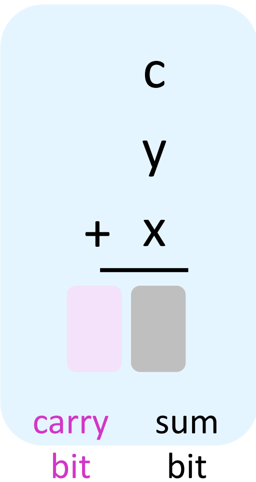
"carry" bit

IN			OUT
x	y	c	circuit output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

What truth table for the "carry" bit?

Part 1: Represent your function as ***bits***...

Any function can be represented using only bits...



IN			OUT
x	**y**	**c**	**circuit output**
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

This one is named the **carry** function

three bits in...

...one bit out

Part 1: Represent your func as ***bits***…

For any func!

Any function can be represented using only bits…

Use a truth table

IN			OUT
x	**y**	**c**	**circuit output**
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

another f'n!

fun!

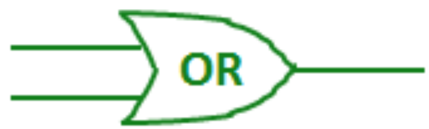
three bits in…

…one bit out

Our building blocks: ***logic gates***

AND

OR



NOT

These circuits are ***physical*** functions of bits…

not just theoretical models

… and *all* mathematical functions can be built from them!

Our building blocks: ***logic gates***

AND

NOT

These circuits are ***physical*** functions of bits…

not just theoretical models

… and *all* mathematical functions can be built from them!

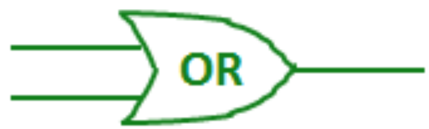
Our building blocks: ***logic gates***

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

AND

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

OR



NOT reverses its input

NOT

These circuits are ***physical*** functions of bits…

not just theoretical models

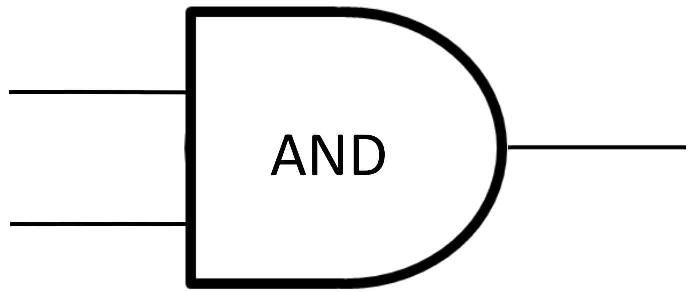
… and *all* mathematical functions can be built from them!

AND

drill sergeant metaphor?

Strict! Everything input must be True to output a True

inputs



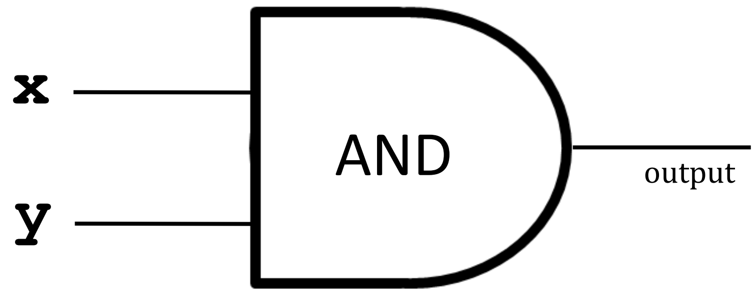
output

AND outputs 1 when **ALL** inputs are 1

otherwise it outputs 0

AND

Strict! Everything input must be True to output a True

inputs ⇨  ⇨ output

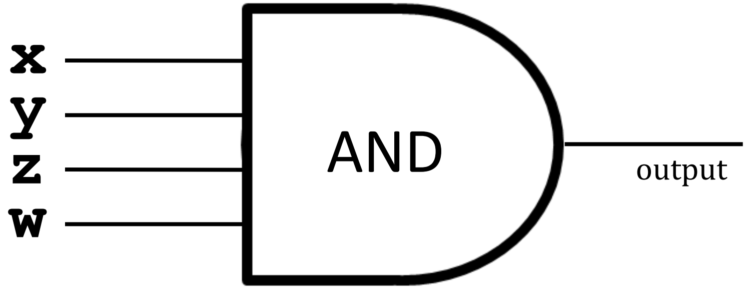
AND's function:

input		output
x	**y**	**AND(x,y)**
0	0	0
0	1	0
1	0	0
1	1	1

"Truth table"

AND

Strict! Everything input must be True to output a True

inputs → 

→ output

AND's function:

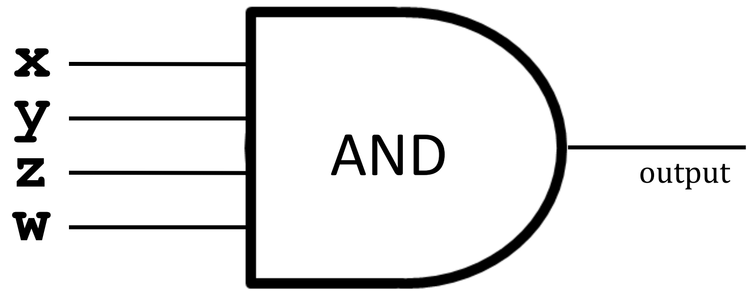
input				output
x	**y**	**z**	**w**	**AND(xyzw)**
0	0	0	0	
0	0	0	1	
...12 more rows not shown...				
1	1	1	0	
1	1	1	1	

How many of the 16 rows here will output a 1?

AND

Strict! Everything input must be True to output a True

inputs



output

AND's function:

input				output
x	**y**	**z**	**w**	**AND(xyzw)**
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

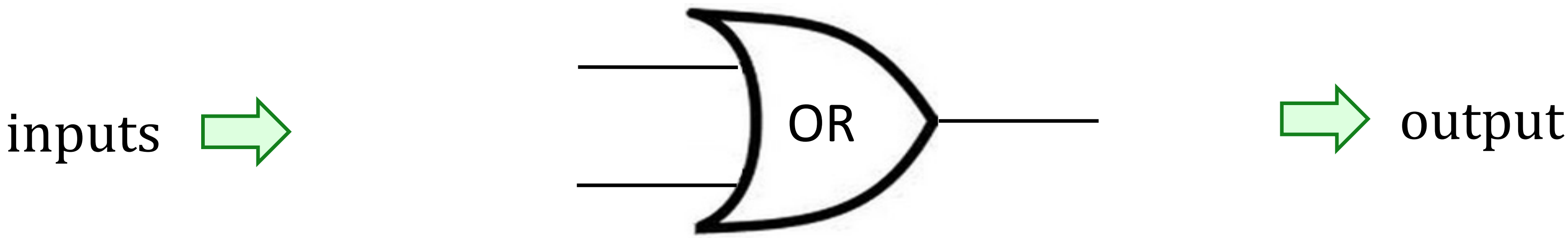
fifteen 0s

one 1

OR

camp counselor metaphor?

easy-going: if anything is OK, everything's OK



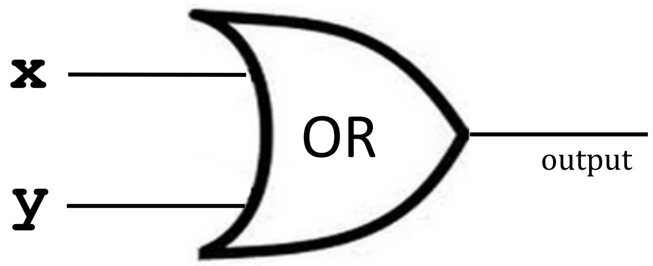
OR outputs 1 when **ANY** input is 1

It outputs 0 only if all inputs are 0.

OR

easy-going: if anything is True, the output is True

inputs ⇨



⇨ output

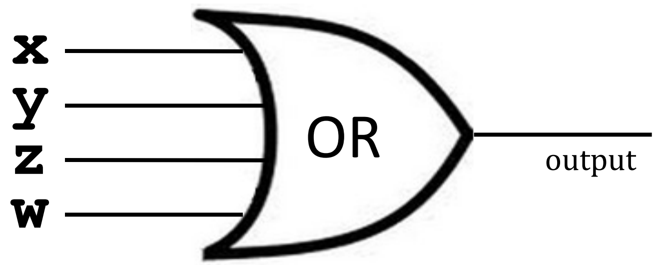
OR's function:

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

OR

easy-going: if anything is OK, everything's OK

inputs



output

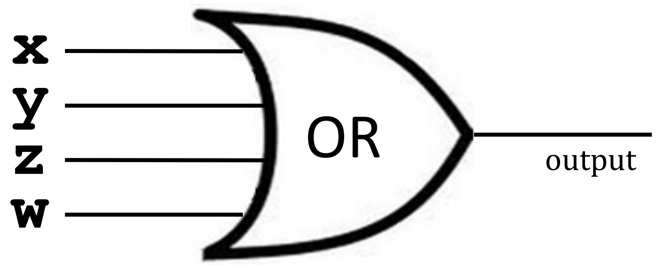
OR's function:

input				output
x	y	z	w	OR(xyzw)
0	0	0	0	
0	0	0	1	
…12 more rows not shown…				
1	1	1	0	
1	1	1	1	

How many of the 16 rows here will output a 1?

OR

easy-going: if anything is OK, everything's OK

inputs →  → output

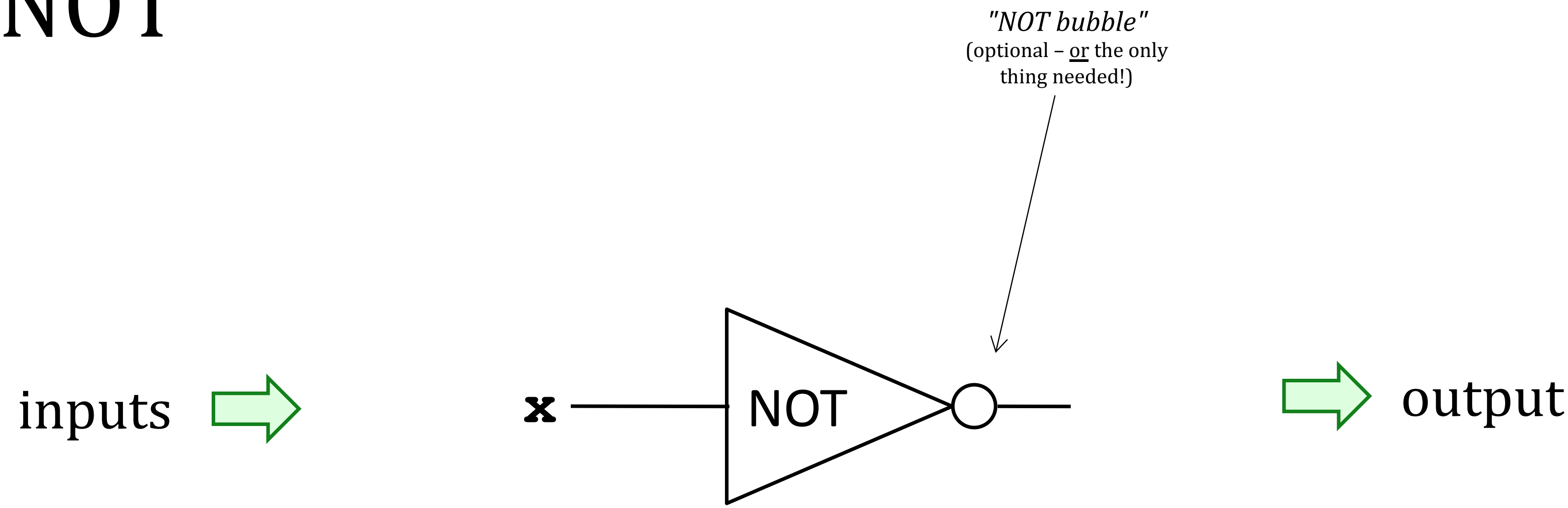
OR's function:

input				output
x	y	z	w	OR(xyzw)
0	0	0	0	0
0	0	0	1	1
…12 more rows not shown…				1
1	1	1	0	1
1	1	1	1	1

one 0

fifteen 1s

NOT



NOT's function:

input x	output NOT(x)
0	1
1	0

one 1

one 0

Our building blocks: ***logic gates***

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

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

input	output
x	NOT(x)
0	1
1	0

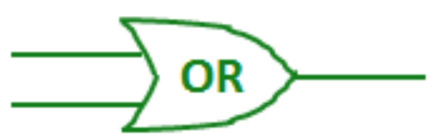
AND outputs 1 only if **ALL** inputs are 1

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

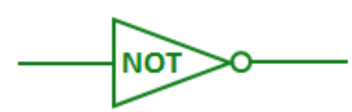
NOT reverses its input

AND

OR



NOT



Our building blocks: ***logic gates***

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

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

AND

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

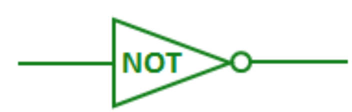
OR outputs 1 if **ANY** input is 1

OR

input	output
x	NOT(x)
0	1
1	0

NOT reverses its input

NOT



Our building blocks: *logic gates*

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

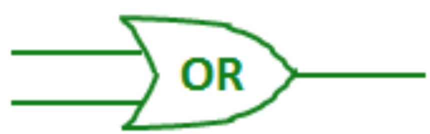
AND outputs 1 only if **ALL** inputs are 1

AND **ALL**

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

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

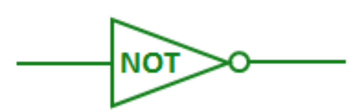
OR **ANY**



input	output
x	NOT(x)
0	1
1	0

NOT reverses its input

NOT

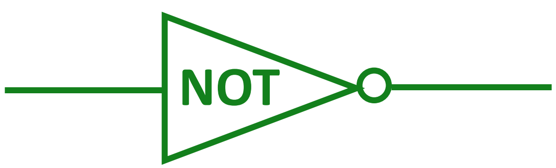


Claim !?

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

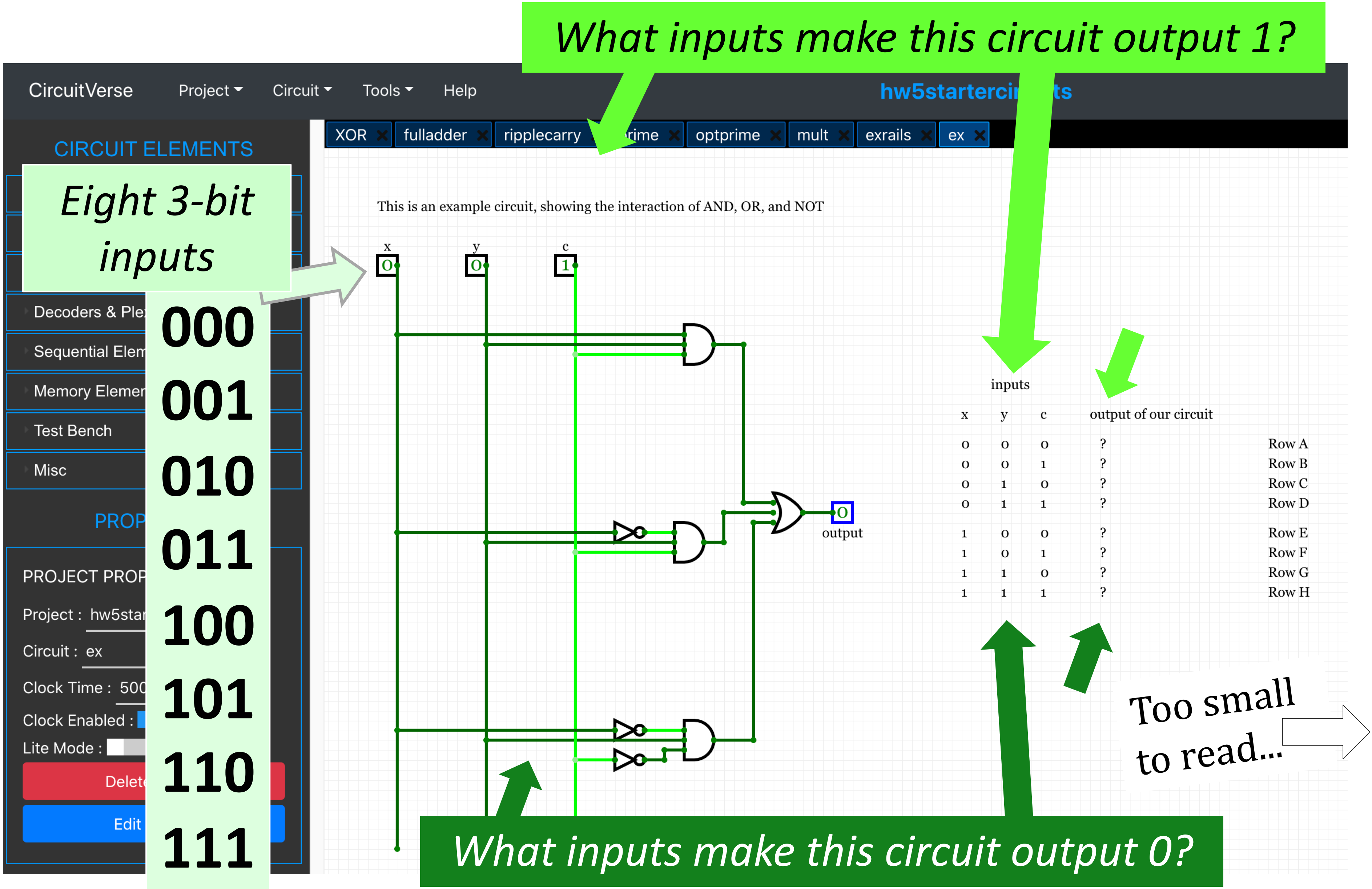
AND outputs 1 iff ***ALL*** its inputs are 1

OR outputs 1 iff ***ANY*** input is 1

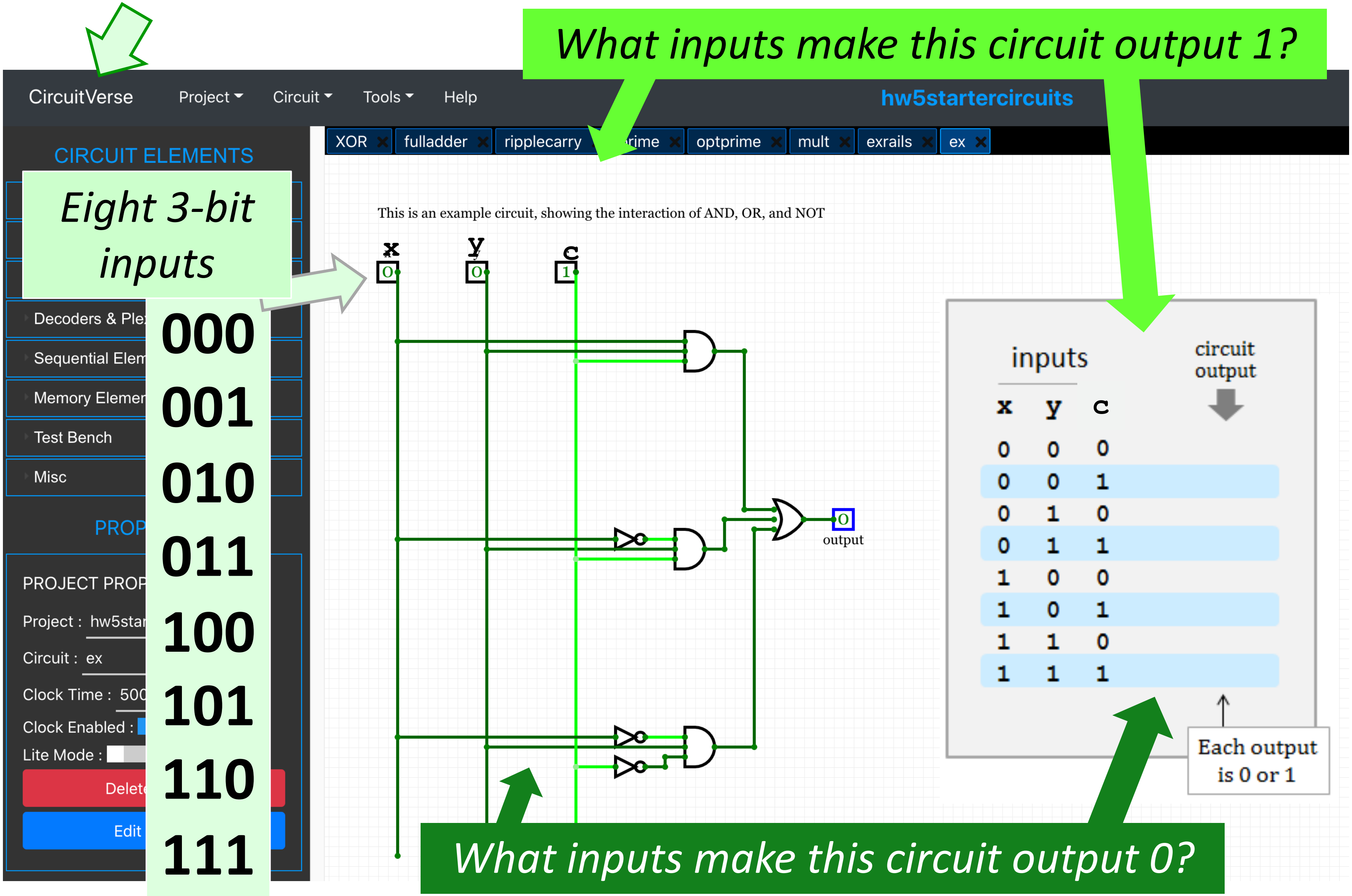


NOT reverses its input

From gates to ***circuits***...



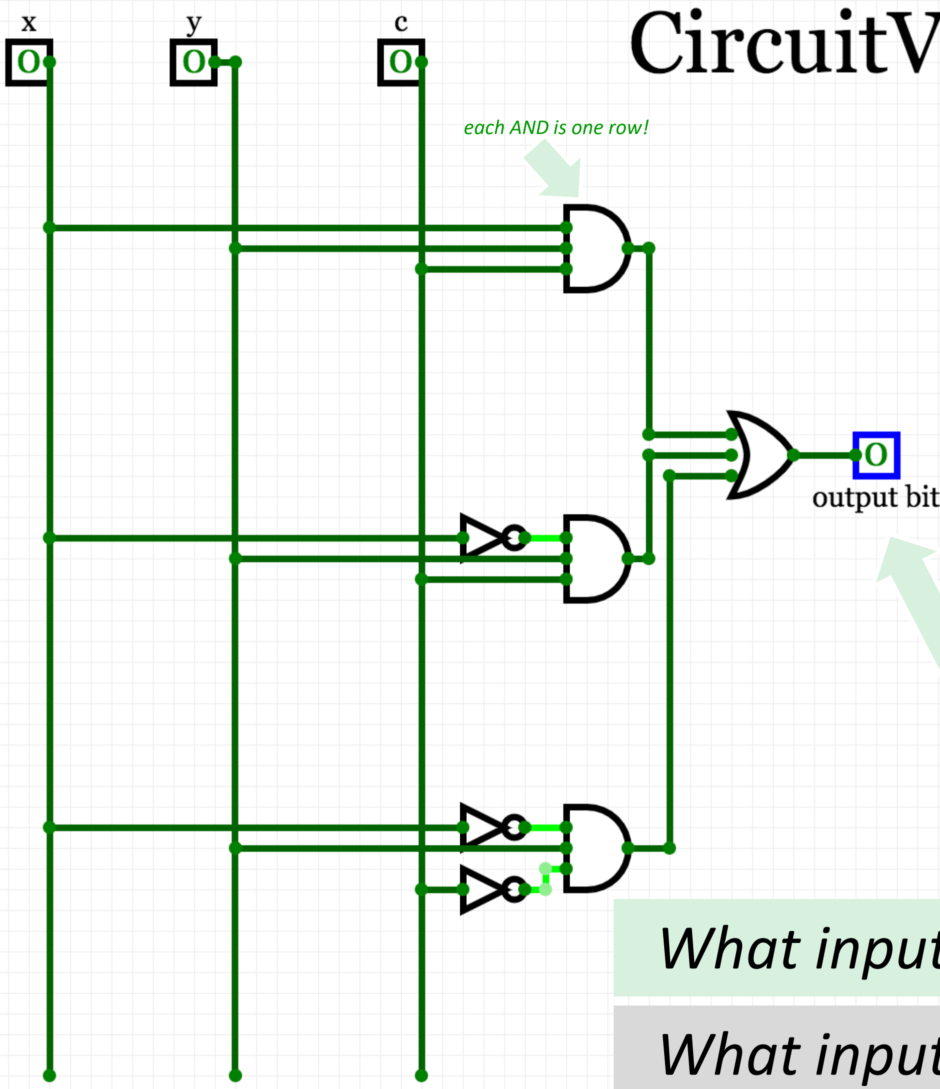
From gates to *circuits*...



A circuit...

CircuitVerse.org

What are all the outputs?



CircuitVerse!

inputs

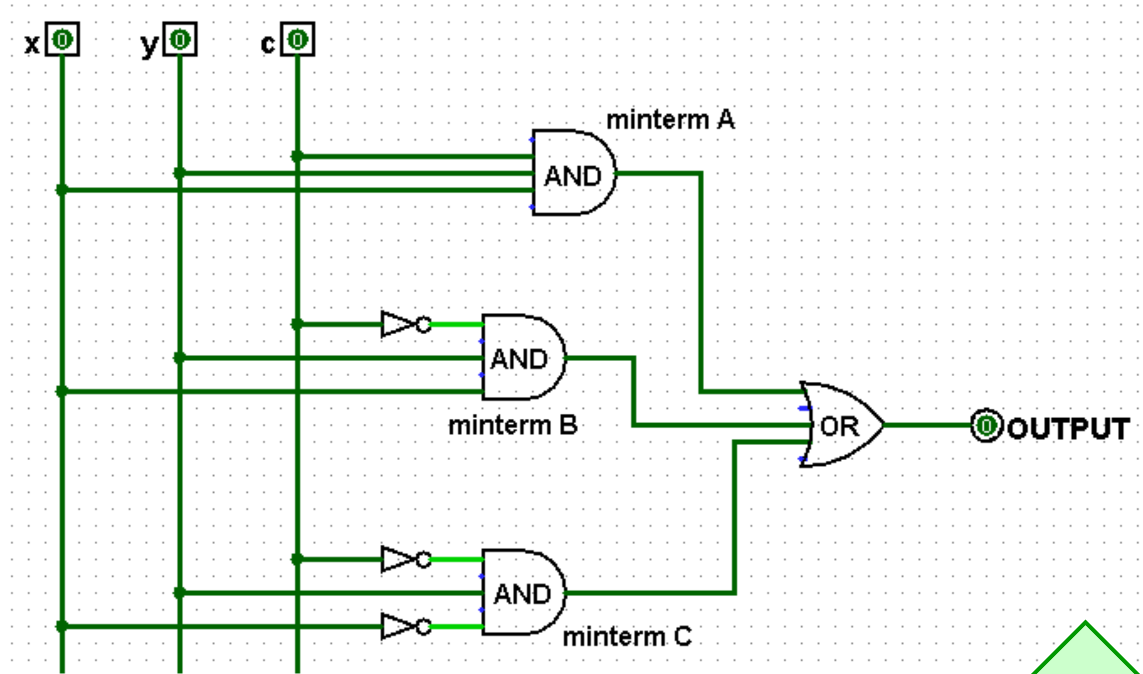
x	y	c	output(x,y,c)	
0	0	0	?	Row A
0	0	1	?	Row B
0	1	0	?	Row C
0	1	1	?	Row D
1	0	0	?	Row E
1	0	1	?	Row F
1	1	0	?	Row G
1	1	1	?	Row H

What inputs make this circuit output 1?

What inputs make this circuit output 0?

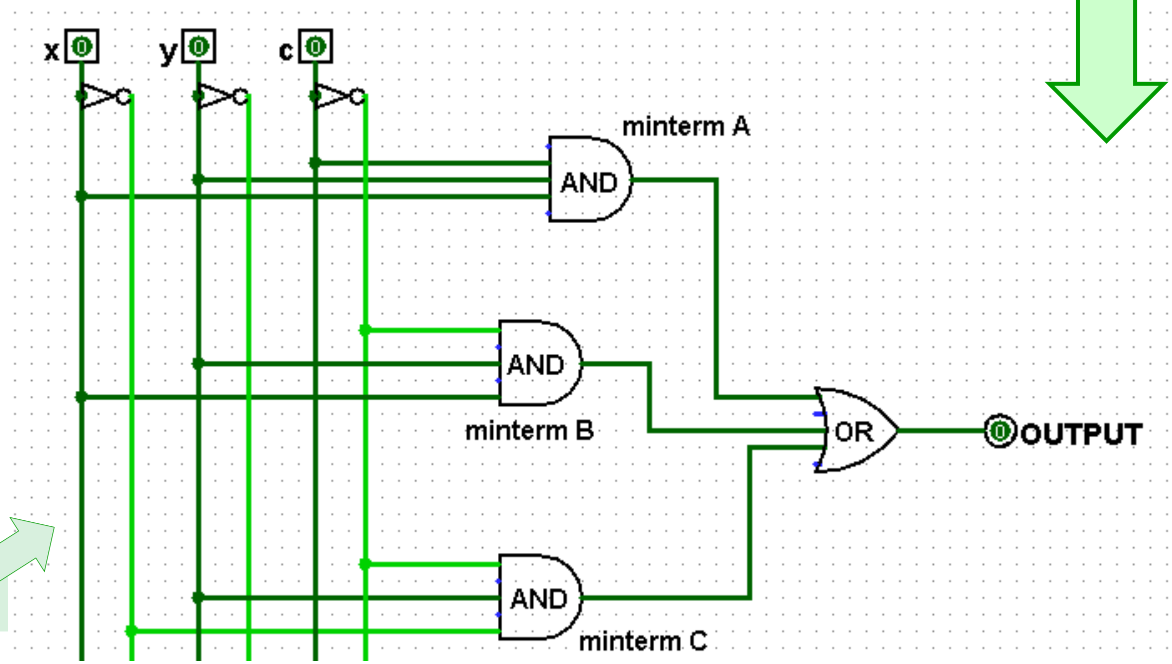
Rails

There is ***NO*** difference between these two circuits!



How?

Any *disadvantages* of this ***"rails"*** approach?



Any *advantages* ?

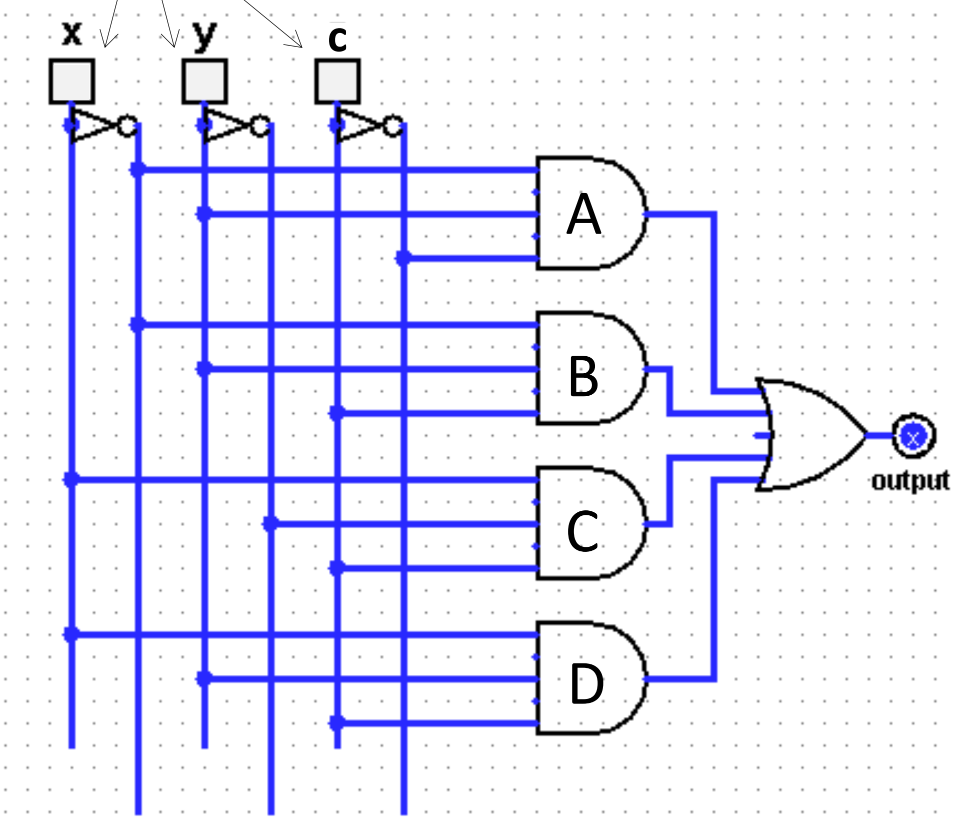
using *rails* for **not x**, **not y**, **not c**

Try it!

Names: ______________________________

Fill in the function values for this circuit (the truth table)

Each input x, y, and z can ***independently*** be 0 or 1, for *eight* total possible inputs:



inputs			circuit output	Gate?
x	y	c		
0	0	0		
0	0	1		
0	1	0	1	A
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Each output is 0 or 1

together

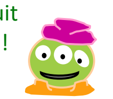
(1) This circuit uses 8 logic gates – *how many of each?* **AND** ___ **OR** ___ **NOT** ___

(2) Follow <u>upstream</u> from A. What x,y,c bits make A output 1 ? (and why is that all we need to know for A?)

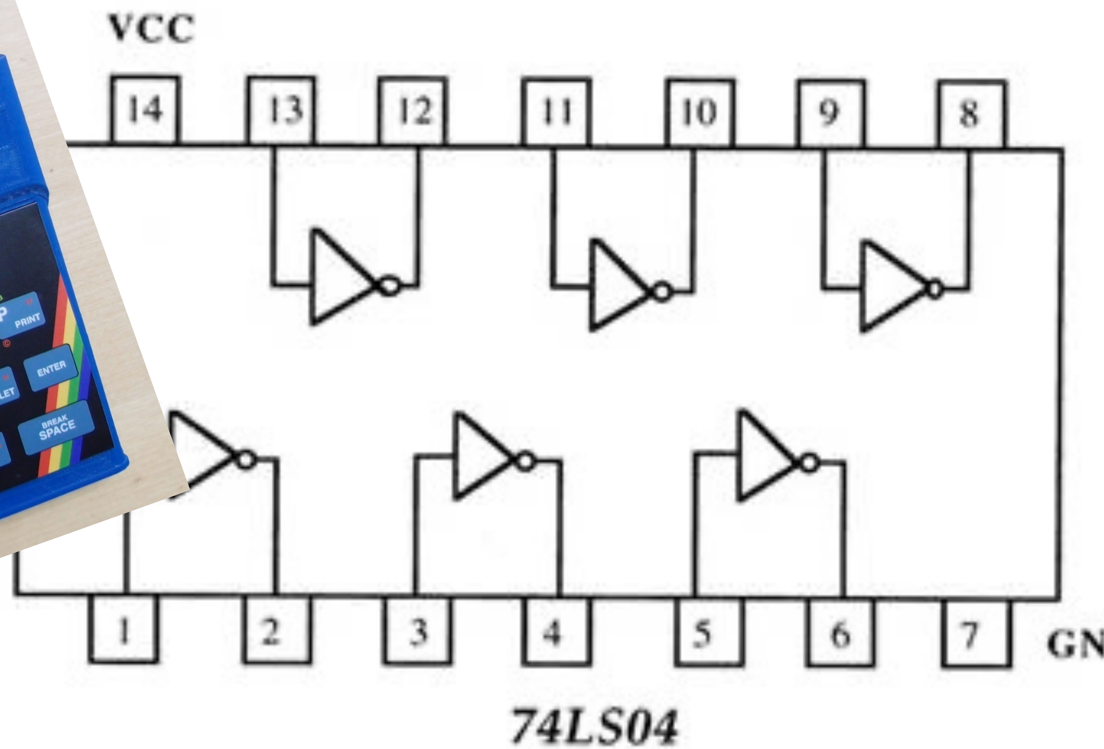
(3) For ***each*** possible input, write the circuit output in the truth table above.

(ec) Could this circuit use ***fewer*** logic gates? ***If so, how?!*** ***If not, how do you know?!***

This circuit is prime!



Real! logic gates...



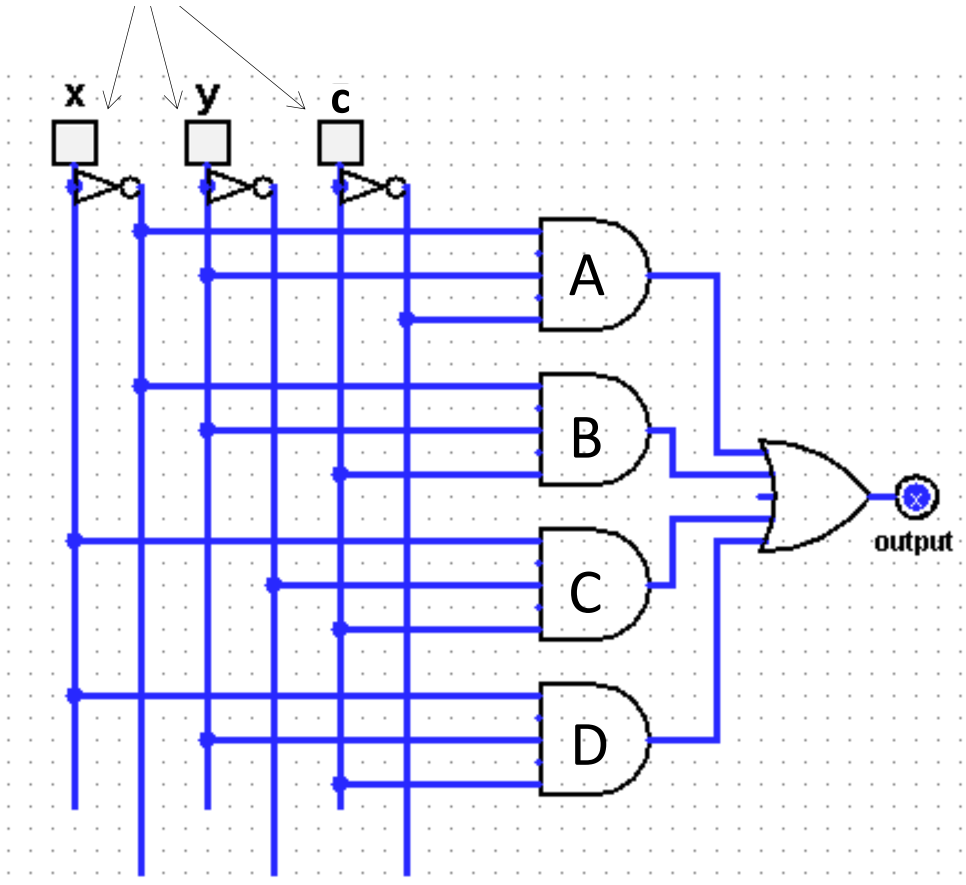
74LS04 NOT gate

Try it!

Try this on the other page first…

Fill in the function values for this circuit (the truth table)

Each input x, y, and z can ***independently*** be 0 or 1, for *eight* total possible inputs:



inputs			circuit output	Gate?
x	y	c		
0	0	0	0	
0	0	1	0	
0	1	0	1	A
0	1	1	1	B
1	0	0	0	
1	0	1	1	C
1	1	0	0	
1	1	1	1	D

Each output is 0 or 1

together

(1) This circuit uses 8 logic gates – *how many of each?* **AND** 4 **OR** 1 **NOT** 3

it's zero otherwise

(2) Follow <u>upstream</u> from A. What x,y,c bits make A output 1 ? 010 hy is that all we need to know for A?)

(3) For ***each*** possible input, write the circuit output in the truth table above. see above!

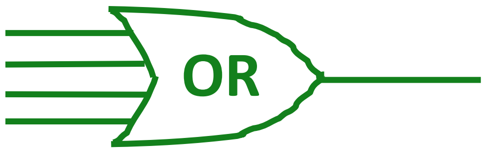
two can be combined!

Could this circuit use ***fewer*** logic gates? ***If so, how?!*** ***If not, how do you know?!***

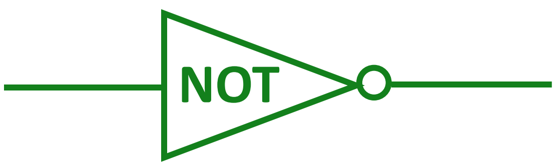
This circuit is prime!

The claim…

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



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



NOT reverses its input

We need <u>only</u> these three building blocks to compute ***anything at all***

The proof... !

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

We prove this ***constructively*** using the ***minterm expansion principle***.

NOT reverses its input

We need <u>only</u> these three building blocks to compute ***anything at all***

A constructive proof...

i Specify a **truth table** defining ***any*** function you want

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

ii For each input row whose output needs to be 1, build an **AND** circuit that outputs 1 <u>*only for that specific input*</u>*!*

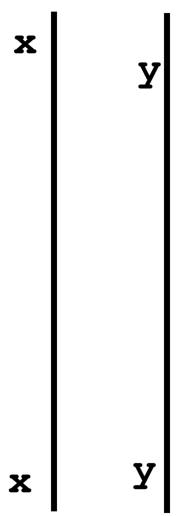
iii **OR** them all together

Hey! This is a 3-i'ed proof!

A constructive proof...

i Specify a **truth table** defining ***any*** function you want

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

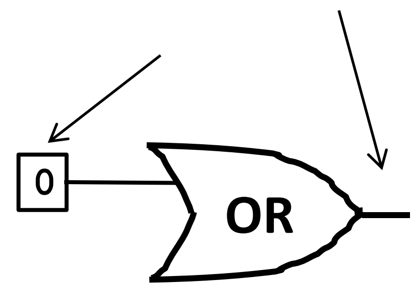


ii For each input row whose output needs to be 1, build an **AND** circuit that outputs 1 *only for that specific input!*

iii **OR** them all together

The ZERO rows ALREADY work – with no connections at all !

We ensure this **OR** outputs zero by default.



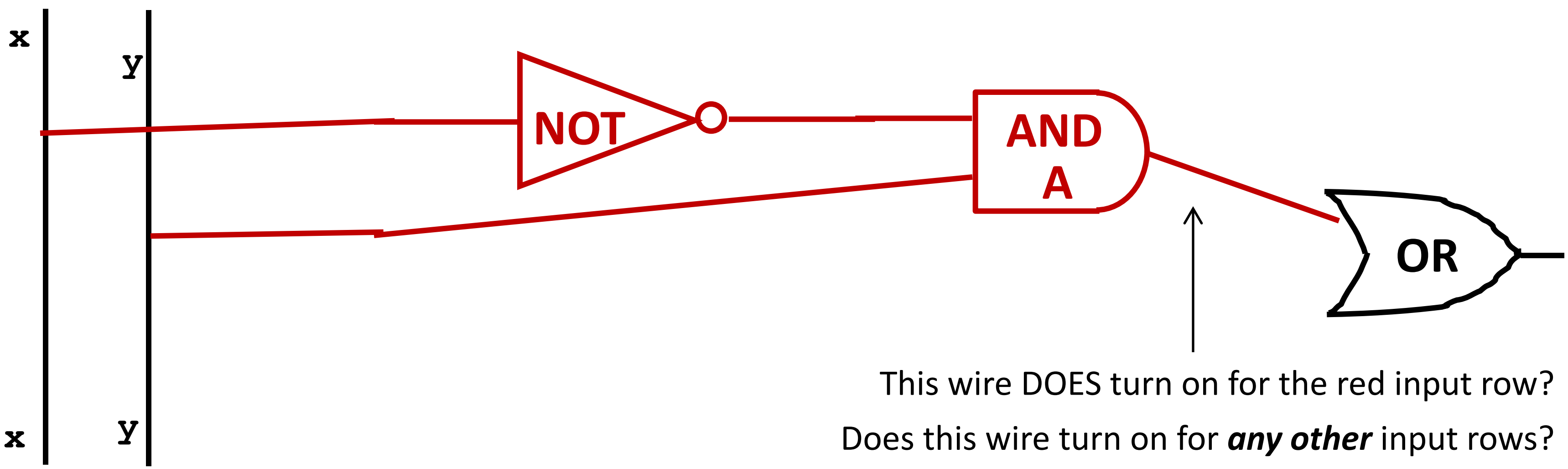
A constructive proof...

i Specify a **truth table** defining ***any*** function you want

	input		output
	x	y	f(x,y)
	0	0	0
A	0	1	1
	1	0	1
	1	1	0

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

iii **OR** them all together



This wire DOES turn on for the red input row?

Does this wire turn on for ***any other*** input rows?

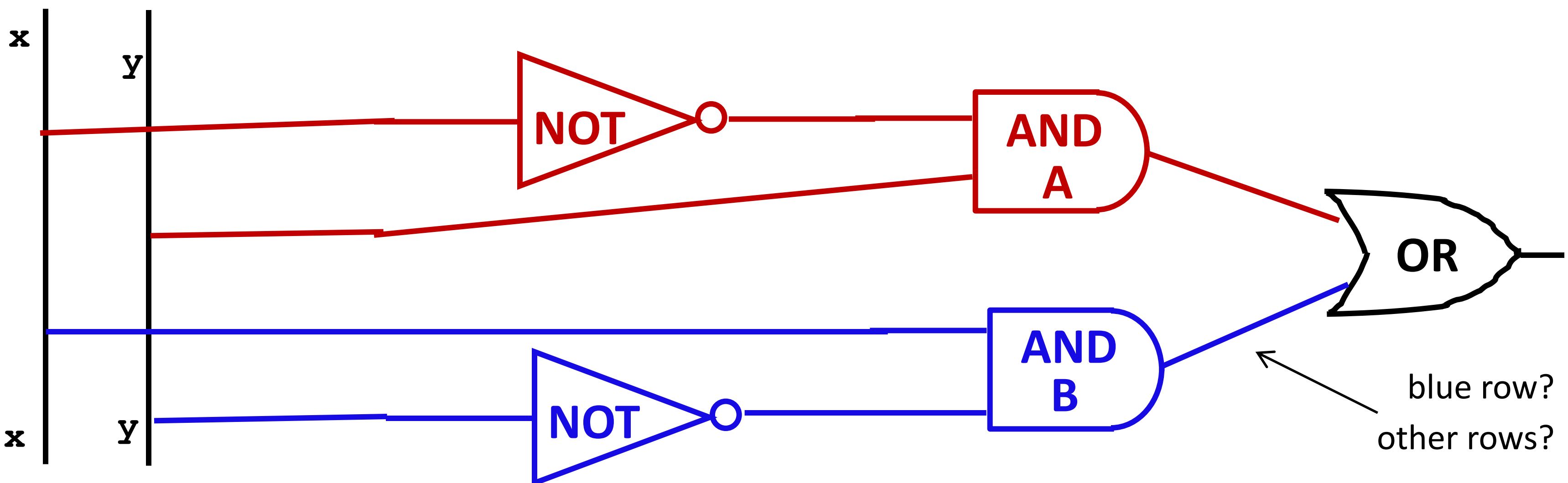
A constructive proof...

(i) Specify a **truth table** defining ***any*** function you want

	input x	input y	output f(x,y)
	0	0	0
A	0	1	1
B	1	0	1
	1	1	0

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

(iii) **OR** them all together



A constructive proof...

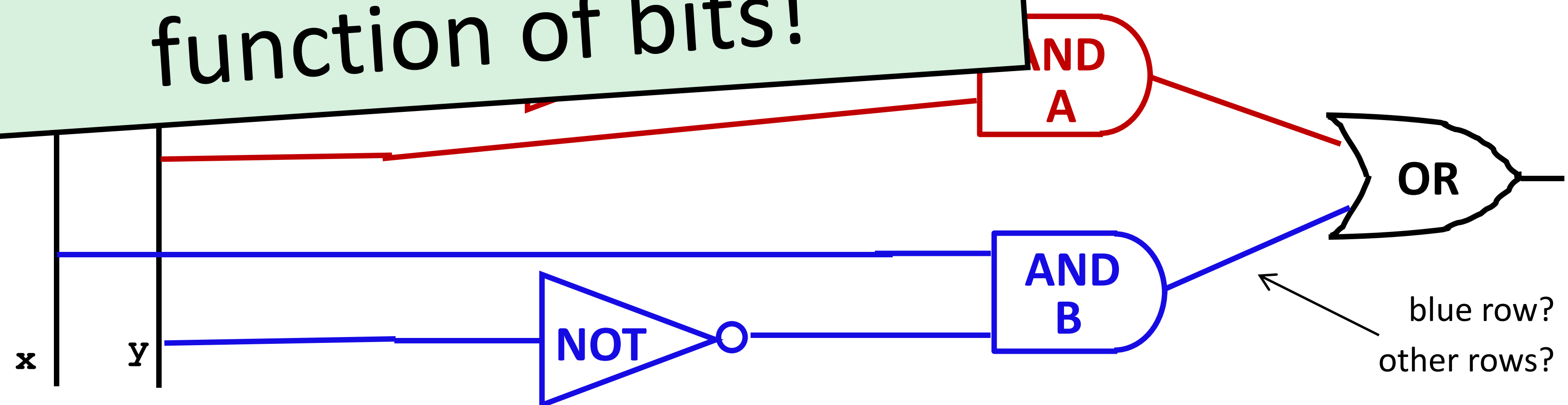
i

MINTERM expansion princple

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

This is a *constructive* proof that AND, OR, NOT suffice to build ***any*** function of bits!

all together



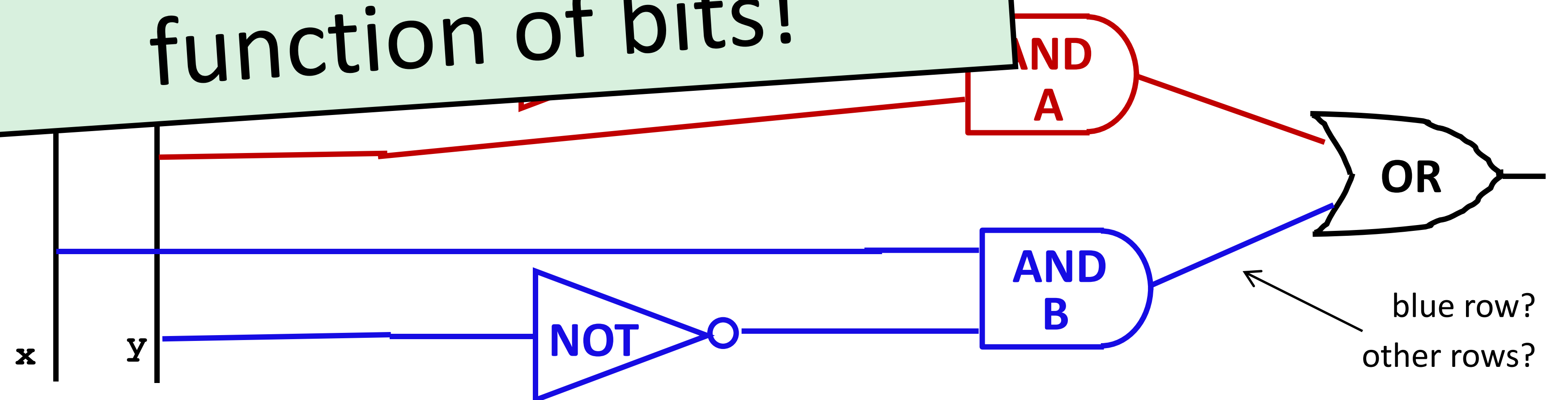
A ...structive proof...

and **ALL** functions are just functions of bits !

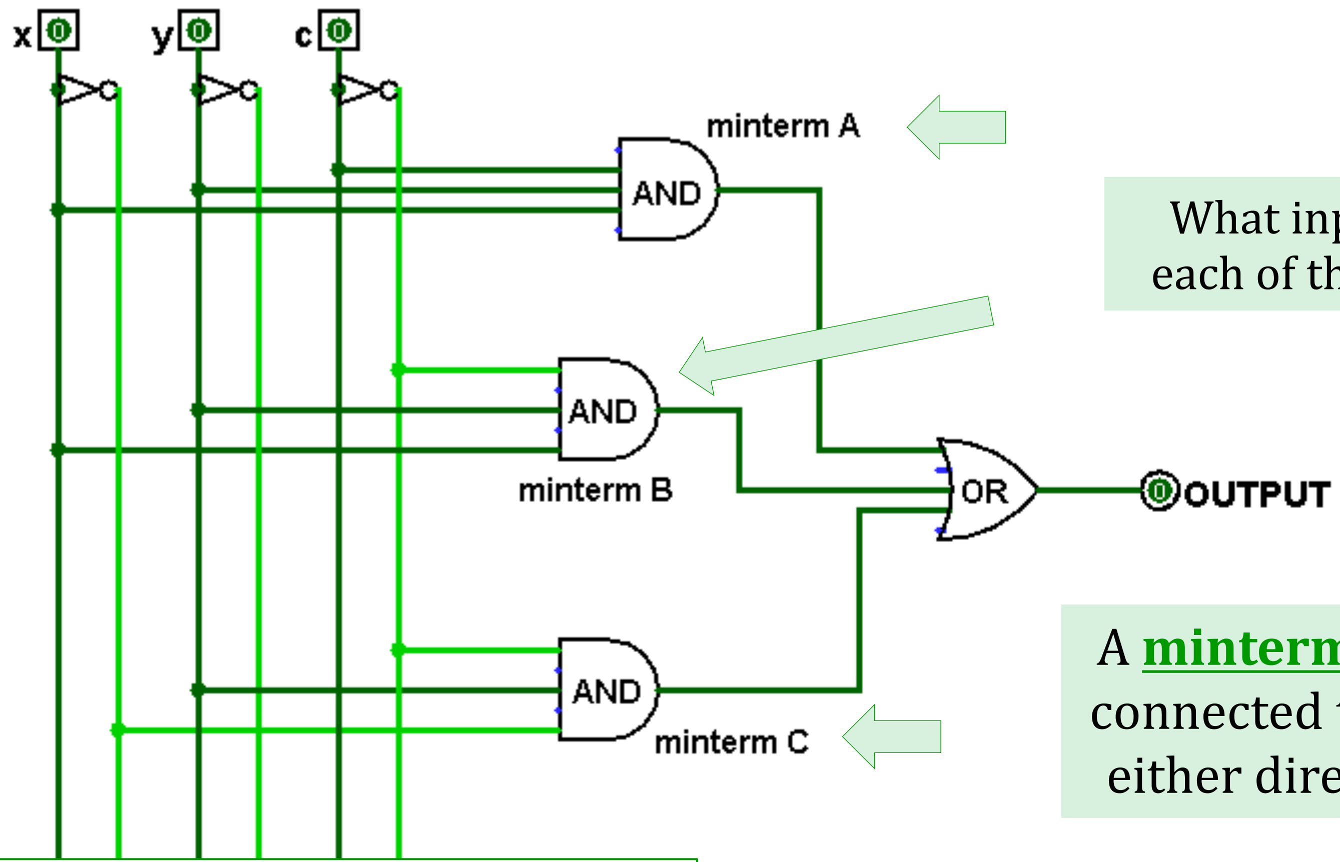
Minterm expansion princple

This is a *constructive* proof that AND, OR, NOT suffice to build ***any*** function of bits!

...all together



Minterm Expansion Principle



What input "activates" each of these minterms?

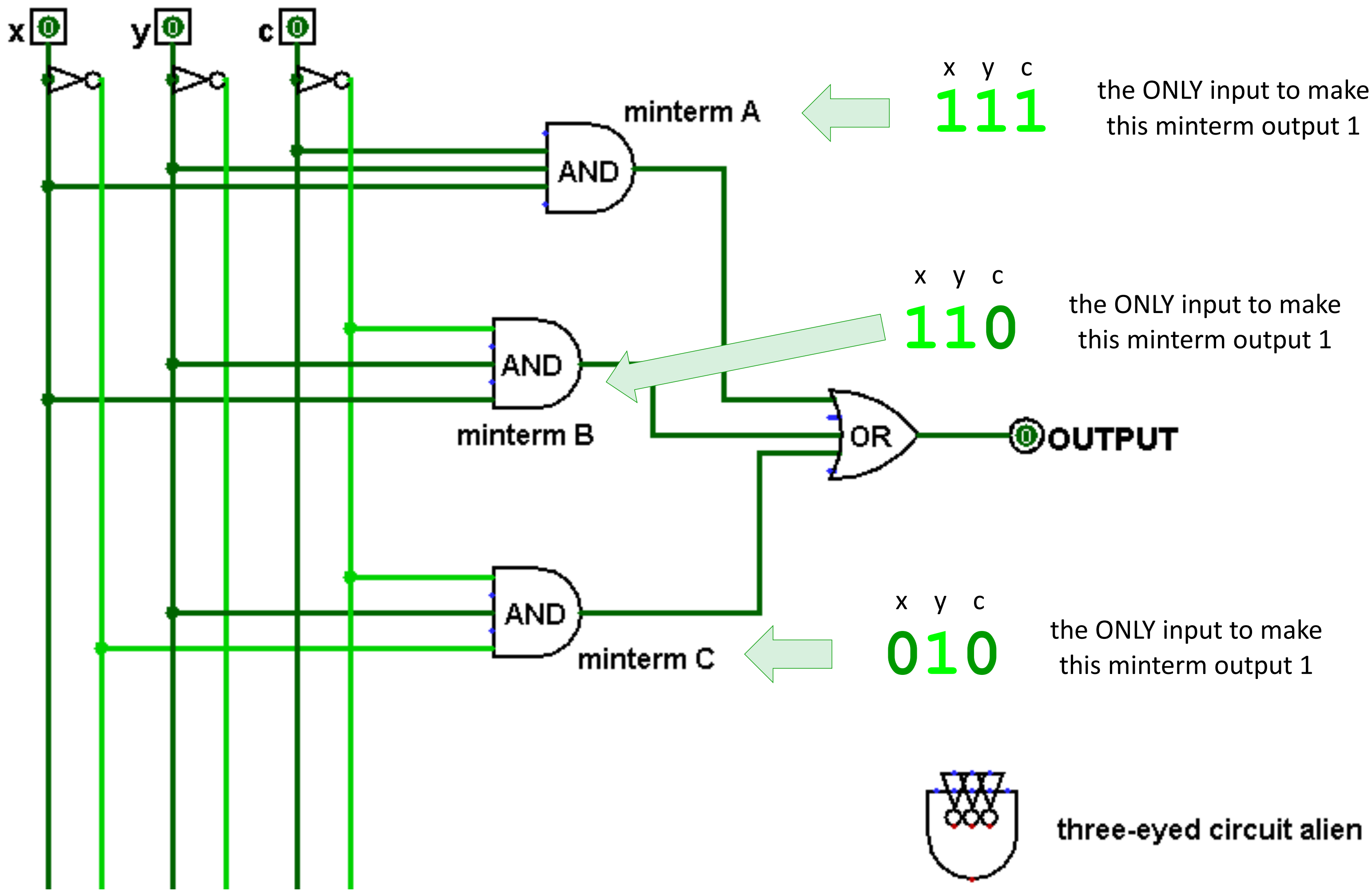
we did this before!

A **minterm** is an **AND** gate connected to *all* input bits - either directly or inverted

For each **1** in the truth table, use one AND gate, called a **minterm**.

Each minterm selects ***one*** input:

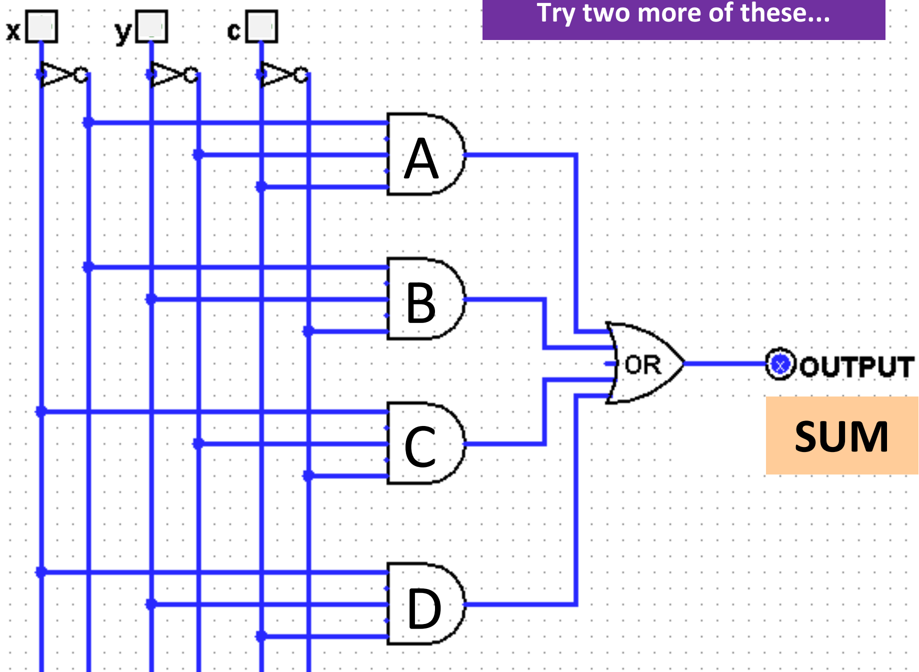
a ***minterm*** is an AND gate that "***selects***" a single input row



Minterm Expansion Principle

Try two more of these…

Take 2…

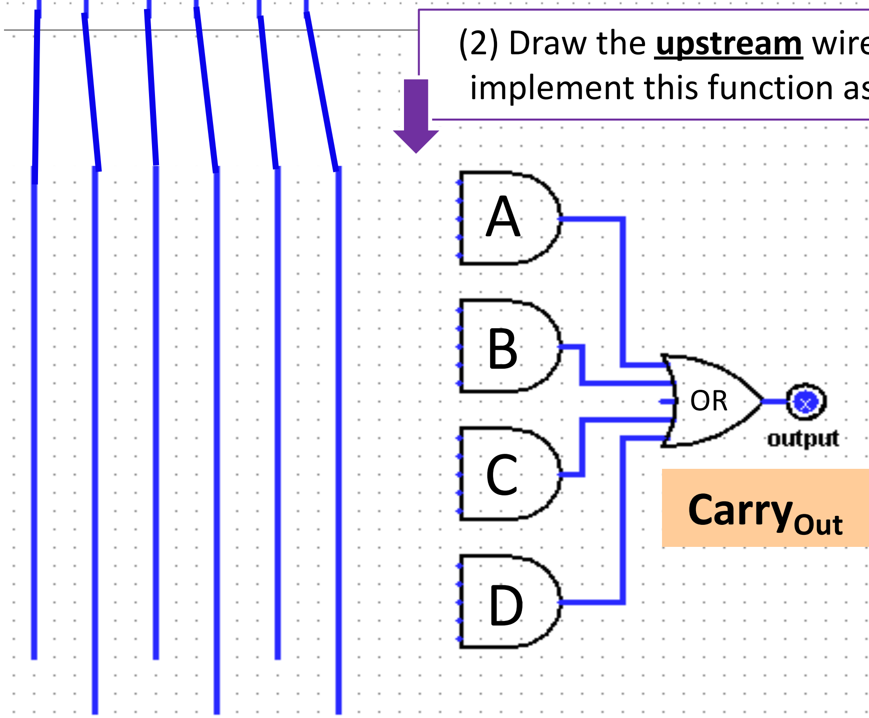


input			output
x	y	c	
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(1) Fill in all 8 rows of the function values (truth table) for this circuit…

Hint: Determine the input that turns each AND gate – each *minterm* -- to `True`

(2) Draw the **upstream** wires that will implement this function as a circuit.



input				output
x	y	c		
0	0	0		0
0	0	1		0
0	1	0		0
0	1	1	A	1
1	0	0		0
1	0	1	B	1
1	1	0	C	1
1	1	1	D	1

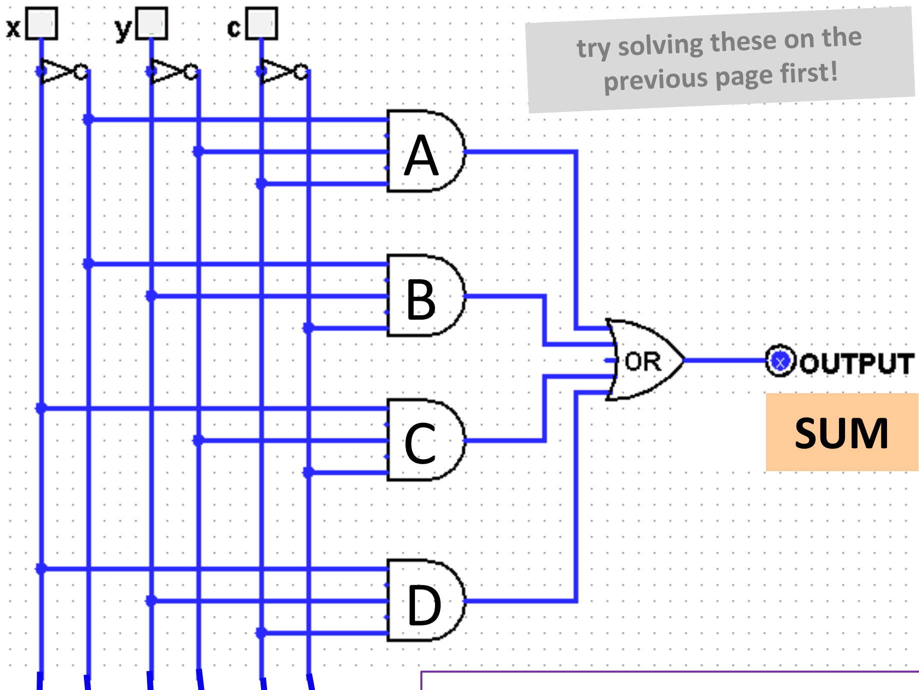
(Extra #1) Any gates you can optimize away here?

(Extra #2) How could you replace the OR with only ANDs and NOTs? ***ORs aren't needed!***

(Extra #3) How do the two circuits on this page implement ***addition of any two binary #s!?***

```
   1 1 1
   1011   x
 + 1111   y
  11010
```

Take 2…

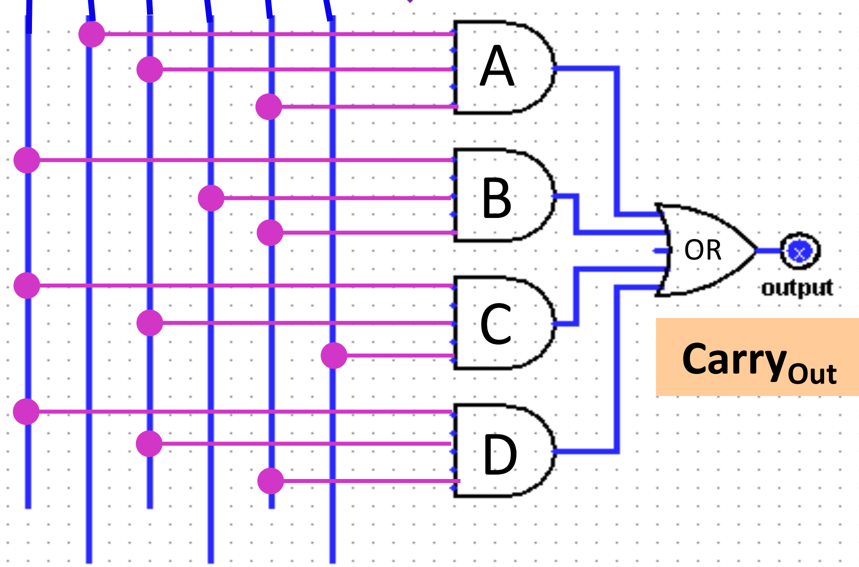


input				output
x	y	c		
0	0	0		0
0	0	1	A	1
0	1	0	B	1
0	1	1		0
1	0	0	C	1
1	0	1		0
1	1	0		0
1	1	1	D	1

(1) Fill in all 8 rows of the function values (truth table) for this circuit…

Hint: Determine the input that turns each AND gate – each *minterm* -- to `True`

(2) Draw the **upstream** wires that will implement this function as a circuit.



input				output
x	y	c		
0	0	0		0
0	0	1		0
0	1	0		0
0	1	1	A	1
1	0	0		0
1	0	1	B	1
1	1	0	C	1
1	1	1	D	1

(Extra #1) Any gates you can optimize away here?

(Extra #2) How could you replace the OR with only ANDs and NOTs? ***ORs aren't needed!***

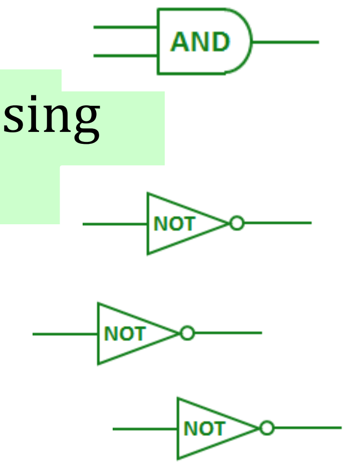
(Extra #3) How do the two circuits on this page implement ***addition of any two binary #s!?***

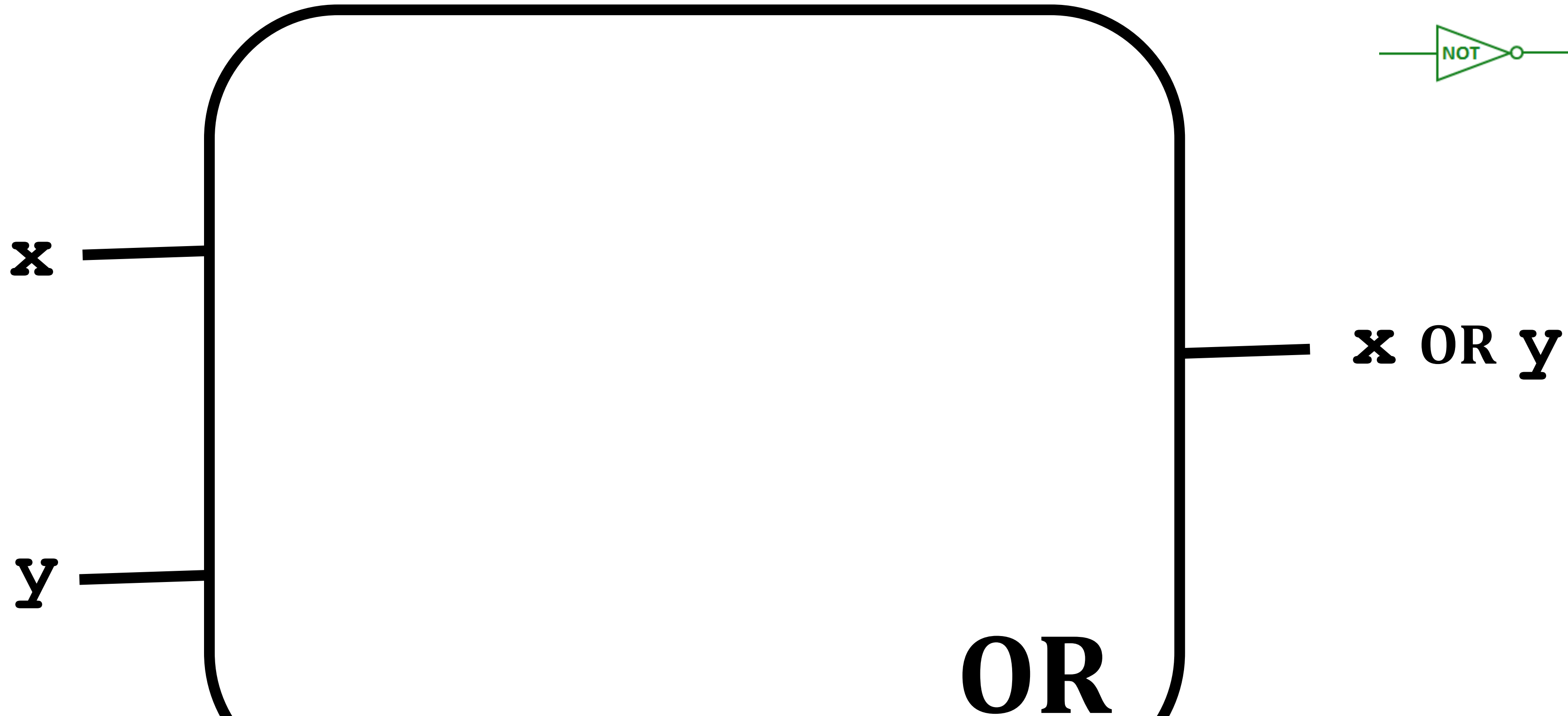
```
  1 1 1
   1011  x
 + 1111  y
  11010
```

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

OR else ?!

Can you get rid of ORs by using only NOTs and ANDs?





Lab5: *adders!*

A ***full adder*** sums three input bits to create *a 2-bit* ***binary*** *output*

x	y	c_{in}	$carry_{out}$	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

3 bits of input

(considered individually)

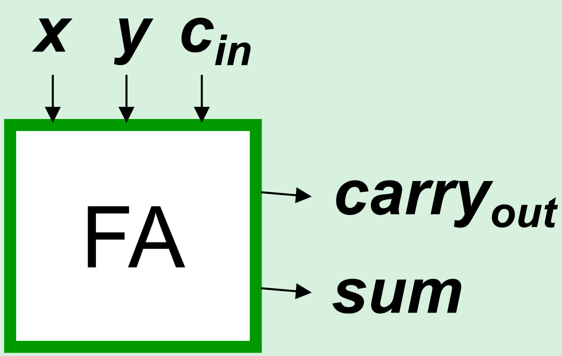
2 bits of output

(considered a binary #)

Full Adder (FA)

these columns look familiar!

the full adder



Full Adder (FA)