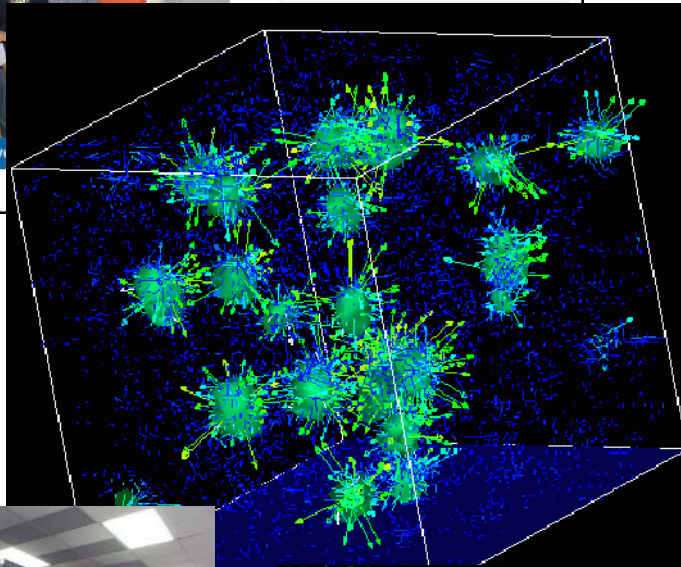




Survey of Scientific Computing (SciComp 301)

Dave Biersach
Brookhaven National
Laboratory
dbiersach@bnl.gov



Session 26
Boolean Algebra,
Logic Gates

```

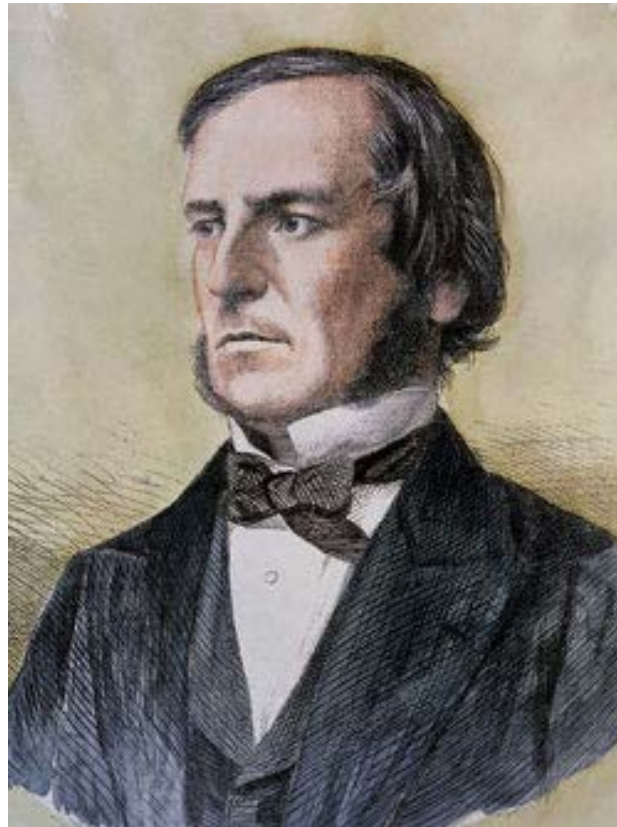
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace SimpleEvents
11 {
12     public partial class Form1 : Form
13     {
14         Person person = new Person();
15
16         public Form1()
17         {
18             InitializeComponent();
19             person.FirstName = "Christian";
20             person.LastName = "Pano";
21         }
22
23         private void button1_Click(object sender, EventArgs e)
24         {
25             person.MainColor = textBox1.Text;
26         }
27     }
28 }
  
```

Session Goals

- Understand the Boolean mathematics of the three basic logic gates: **NOT**, **AND**, and **OR**
- Learn how to draw individual logic gates and how to chain multiple logic gates *together* in a circuit
- Incorporate multiple data input and output **lines** in a circuit
- Develop **truth tables** and analyze logic circuits to calculate the output states given the input states
- Understand how **half-adders** and **full-adders** operate
- Appreciate how **memory** can be constructed from gates

George Boole (1815 – 1864)

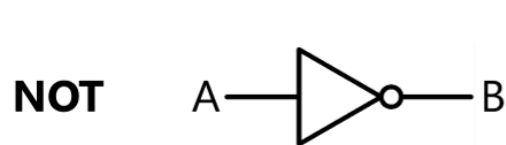
- English mathematician
- 1847 published rules of symbolic logic (“Boolean Algebra”)
- Only person with a data type named after him (“**bool**”)



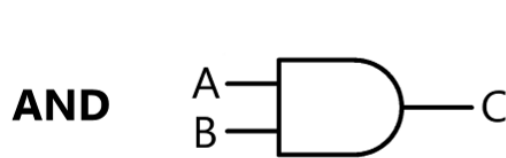
Boolean Logic Gates

- Logic Gates
 - Three types: **NOT** (inverter), **AND**, **OR**
 - Gates have **1 or 2 input** lines, and **1 output** line
- Input / Output lines are either:
 - False, F, Cold, Low, **L, 0** (Zero)
 - True, T, Hot, High, **H, 1**
- The left side of a **truth table** is counted using **Base 2** to ensure every possible *permutation* of input states is evaluated across the entire circuit

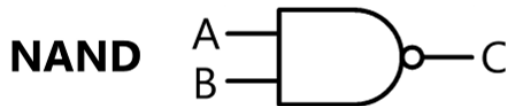
Boolean Logic Gates



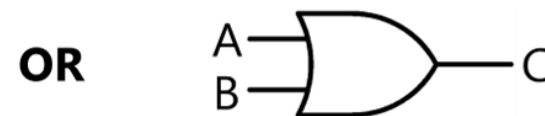
A	B
1	0
0	1



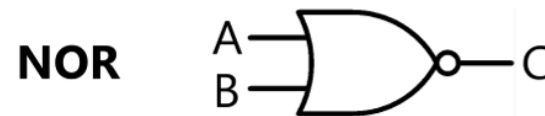
A	B	C
1	1	1
1	0	0
0	1	0
0	0	0



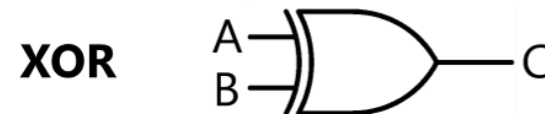
A	B	C
1	1	0
1	0	1
0	1	1
0	0	1



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



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

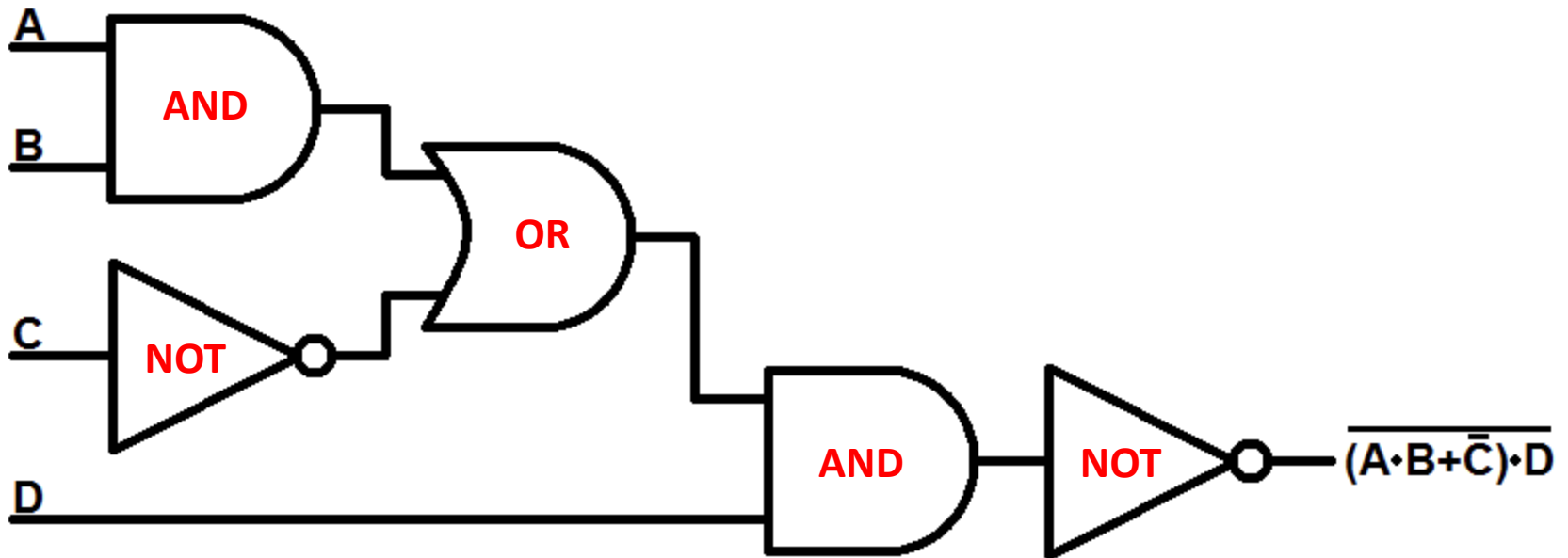


A	B	C
1	1	0
1	0	1
0	1	1
0	0	0

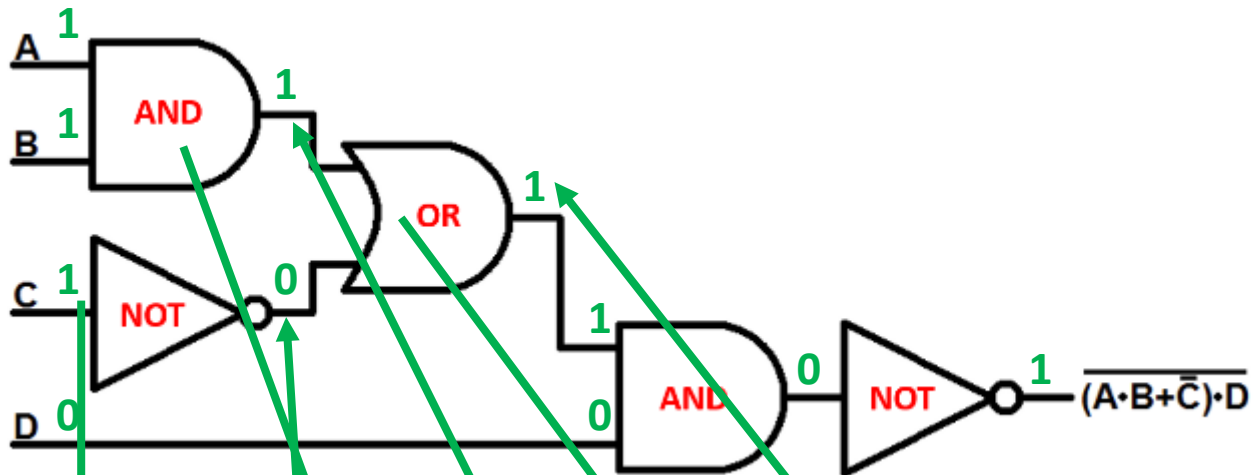
Boolean Logic Gates

- **NOT** (inversion) is sometimes shown as a “bar” on top
- **OR** is a “sum” while **AND** is a “multiply” (modulo 2)
 - OR is sometimes shown as $A + B$
 - AND is sometimes shown as $A \bullet B$
- Circuits flow (propagate state) from “**Left to Right**”
 - The output of one gate flows into the input(s) of the *next* gate(s)
 - We can evaluate a gate’s output line **only** when we know the value for every input line entering that gate

Chaining Logic Gates



Truth Tables



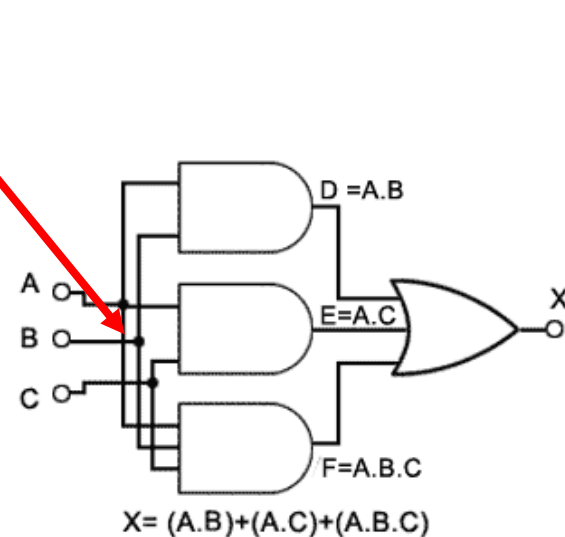
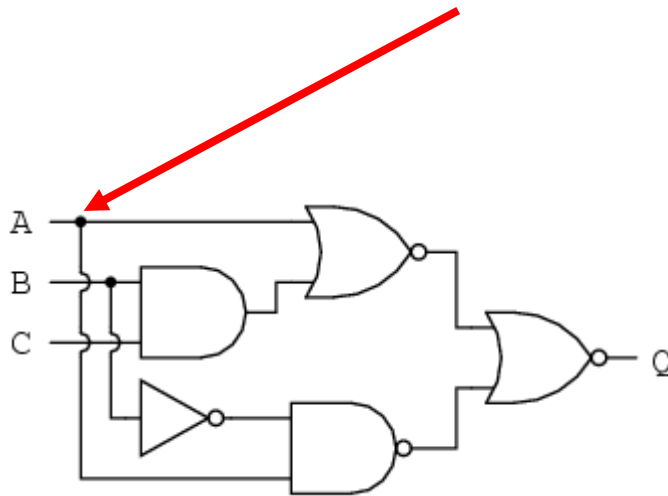
NOT Gate	
INPUT	OUTPUT
0	1
1	0

AND Gate		
INPUT		OUTPUT
0	0	0
0	1	0
1	0	0
1	1	1

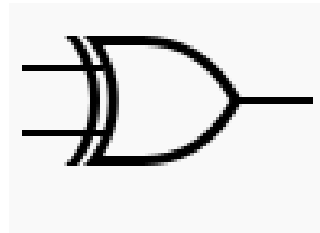
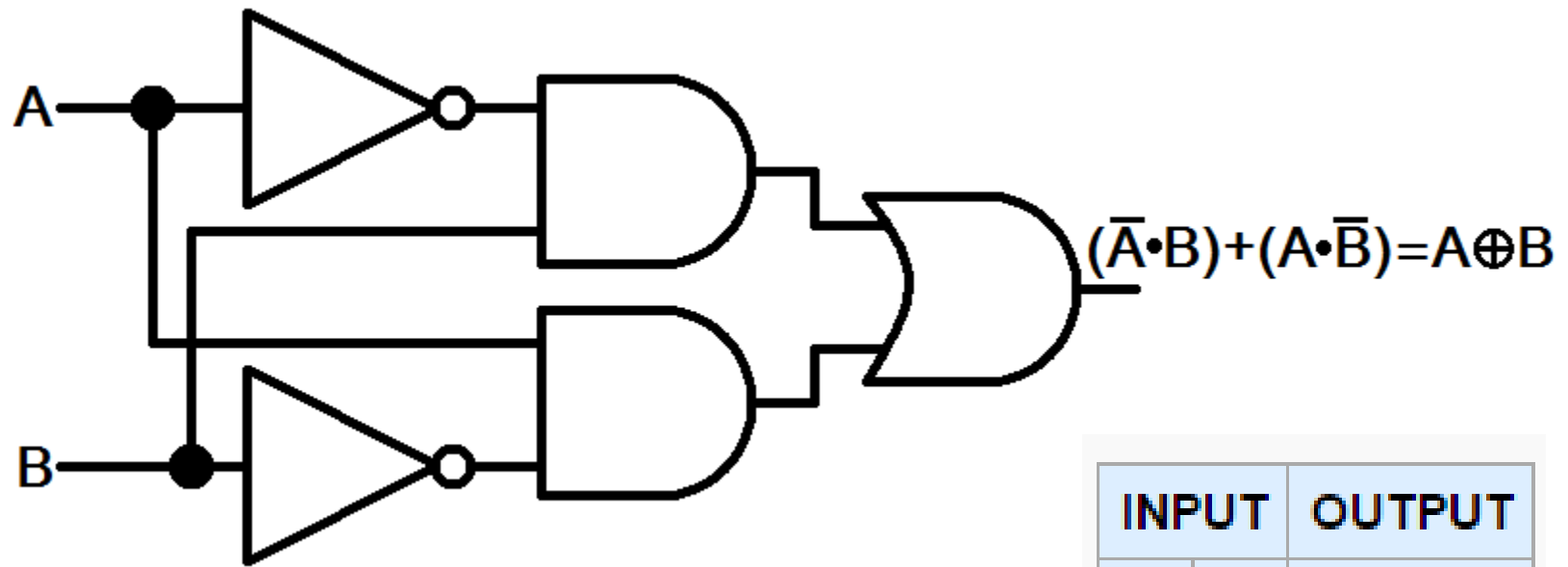
OR Gate		
INPUT		OUTPUT
0	0	0
0	1	1
1	0	1
1	1	1

Wire Overlap vs. Intersection

- Use **filled** circles *only* for electrical junction points
- Carry forward current logic state of line into each **branch**

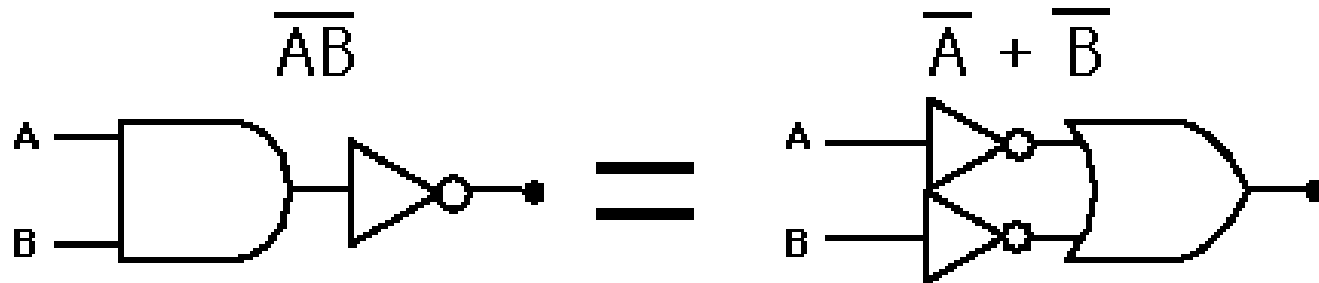


XOR Gate

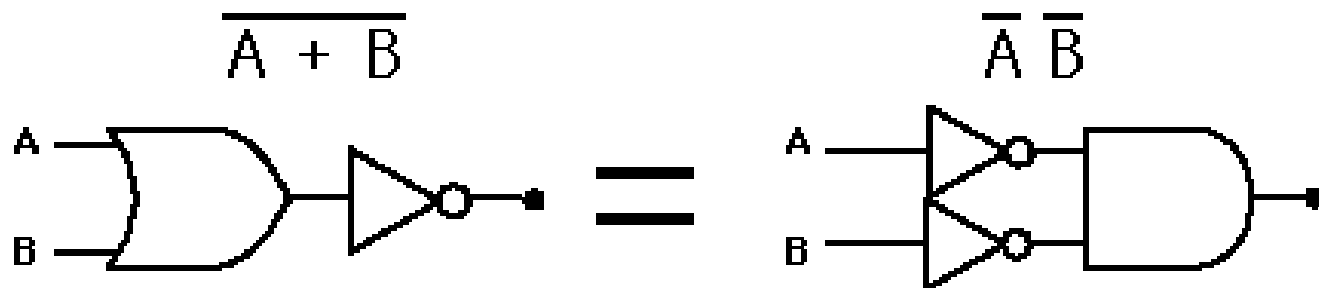


INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

NAND and NOR Gates

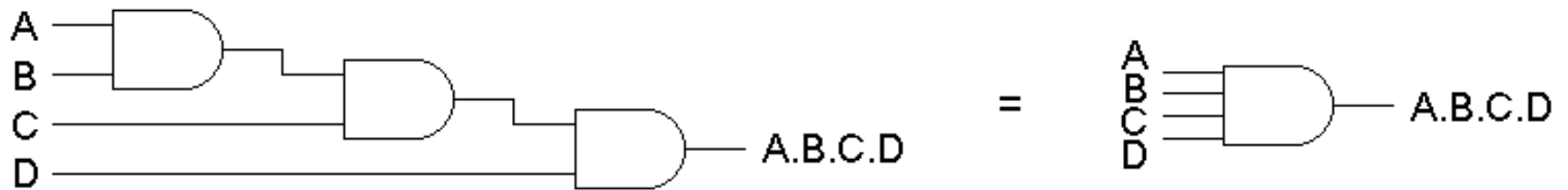


A NAND gate is equivalent to an inversion followed by an OR

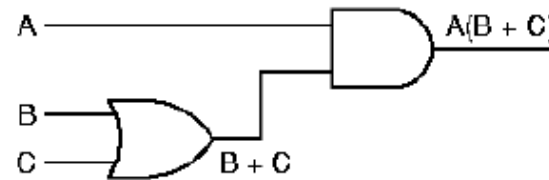
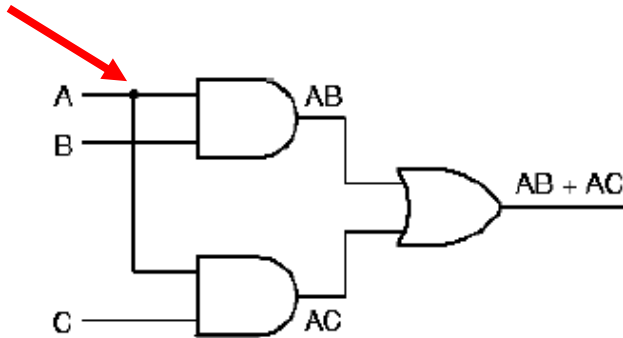


A NOR gate is equivalent to an inversion followed by an AND

Multi-Input Gates



Truth Tables



$$000_2 = 0_{10}$$

$$001_2 = 1_{10}$$

$$010_2 = 2_{10}$$

$$011_2 = 3_{10}$$

$$100_2 = 4_{10}$$

$$101_2 = 5_{10}$$

$$110_2 = 6_{10}$$

$$111_2 = 7_{10}$$

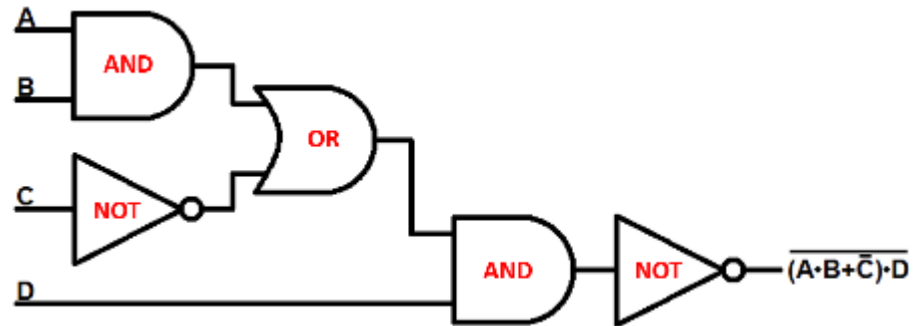
A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

With 3 *input* lines, there should be $2^3 = 8$ rows in the circuit's truth table (0-7)

Lab 1 – Complete a Truth Table

Base ₁₀	INPUT				OUTPUT
	A	B	C	D	
0	0	0	0	0	
1	0	0	0	1	
2	0	0	1	0	
3	0	0	1	1	
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	
8	1	0	0	0	
9	1	0	0	1	
10	1	0	1	0	
11	1	0	1	1	
12	1	1	0	0	
13	1	1	0	1	
14	1	1	1	0	
15	1	1	1	1	

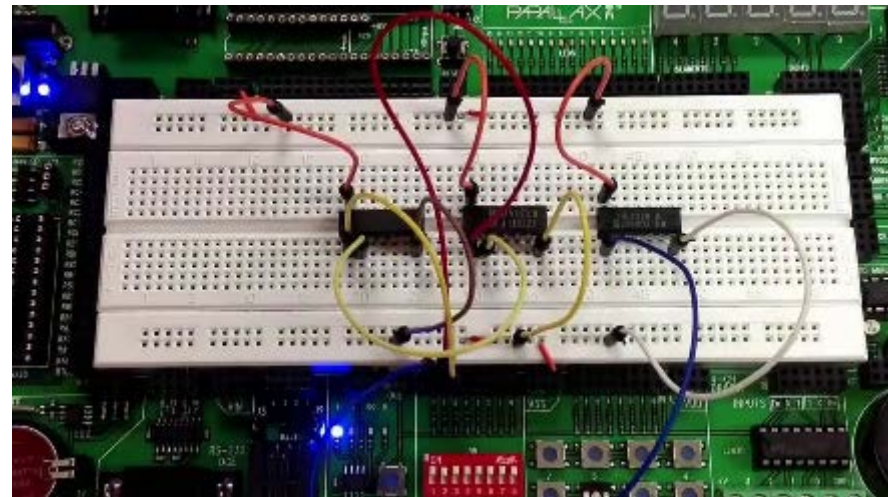
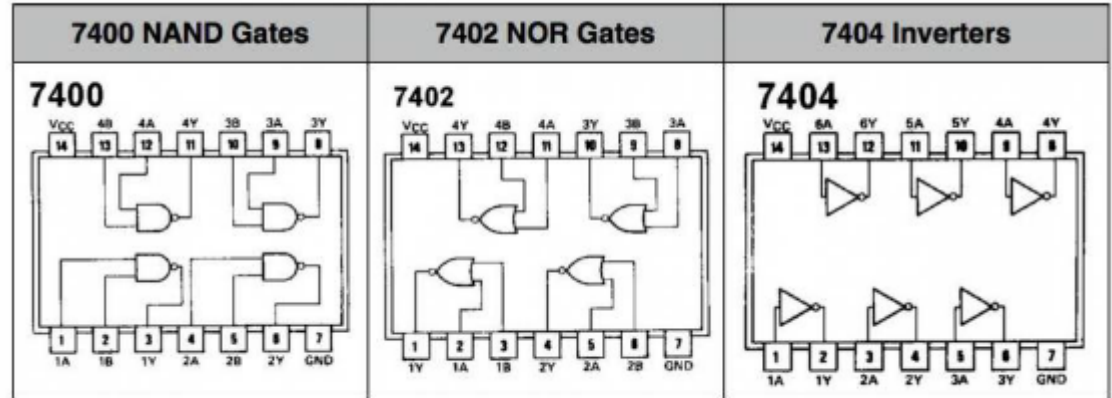
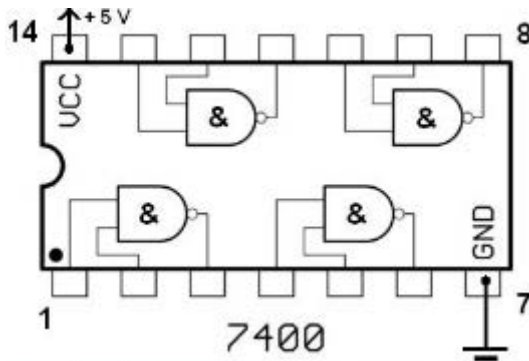


NOT Gate	
INPUT	OUTPUT
0	1
1	0

AND Gate		
INPUT		OUTPUT
0	0	0
0	1	0
1	0	0
1	1	1

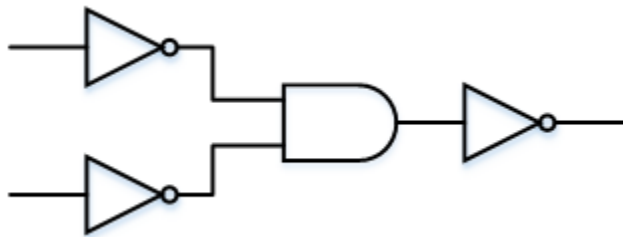
OR Gate		
INPUT		OUTPUT
0	0	0
0	1	1
1	0	1
1	1	1

Logic Gates in the Real World

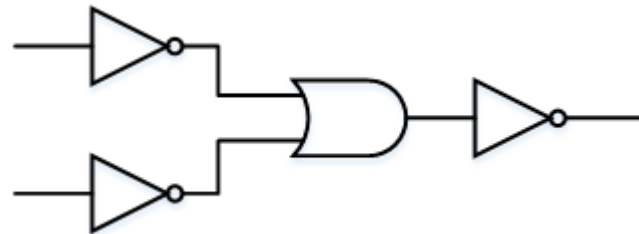


Gate Equivalence

- **Augustus De Morgan's laws:**
 - Can make an AND gate from 3 NOTs and 1 OR
 - Can make an OR gate from 3 NOTs and 1 AND
 - Simply invert both inputs and the output!

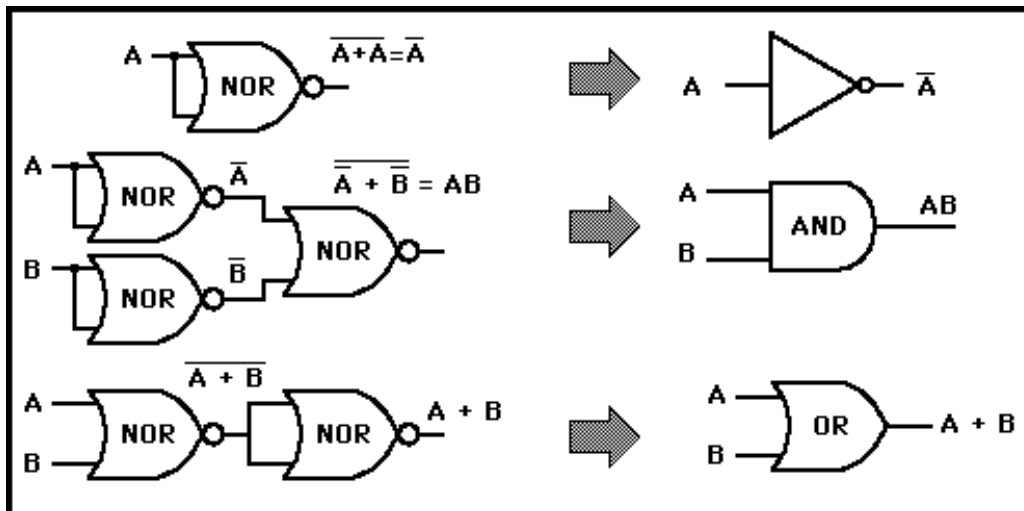
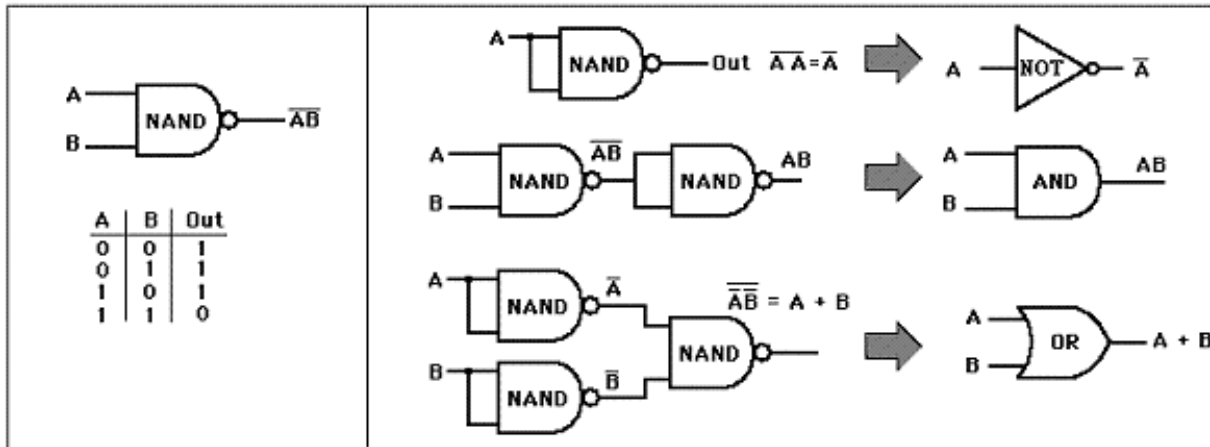


OR from AND



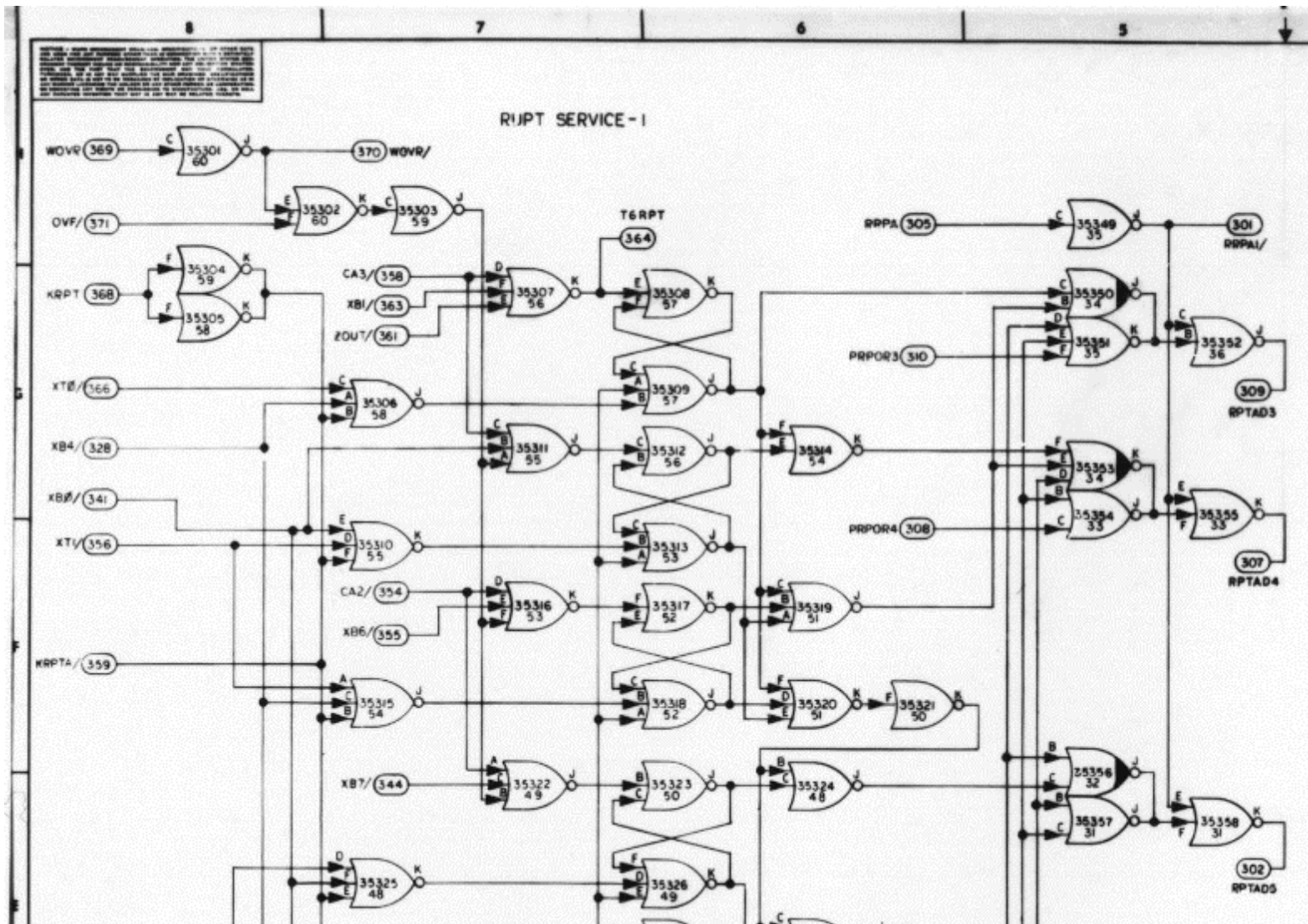
AND from OR

De Morgan's Laws

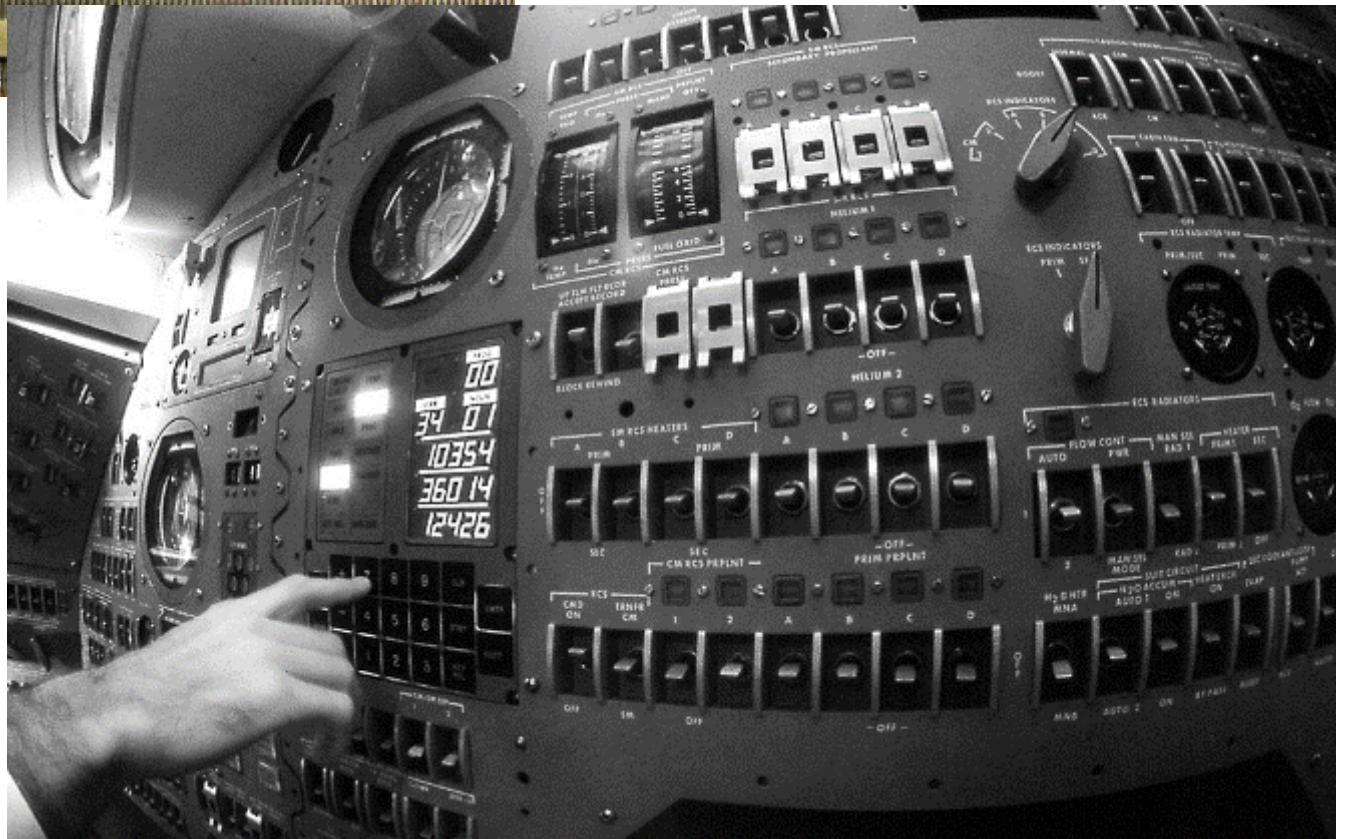


We can build any circuit using just NAND or NOR gates!

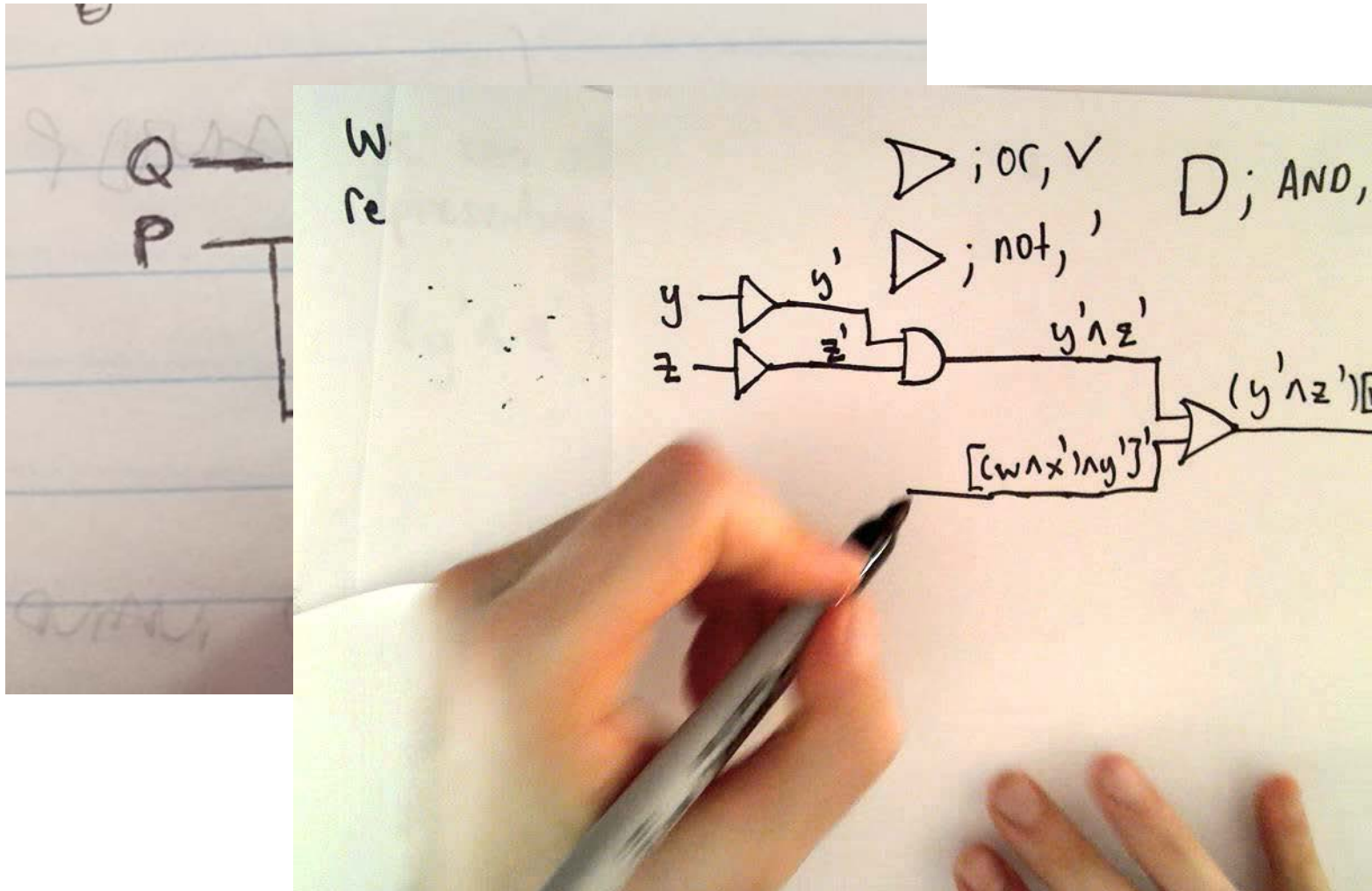
Apollo Guidance Computer



Apollo Guidance Computer

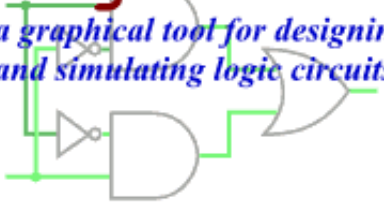


Drawing A Digital Circuit



Logisim

*a graphical tool for designing
and simulating logic circuits*



<http://www.cburch.com/logisim>

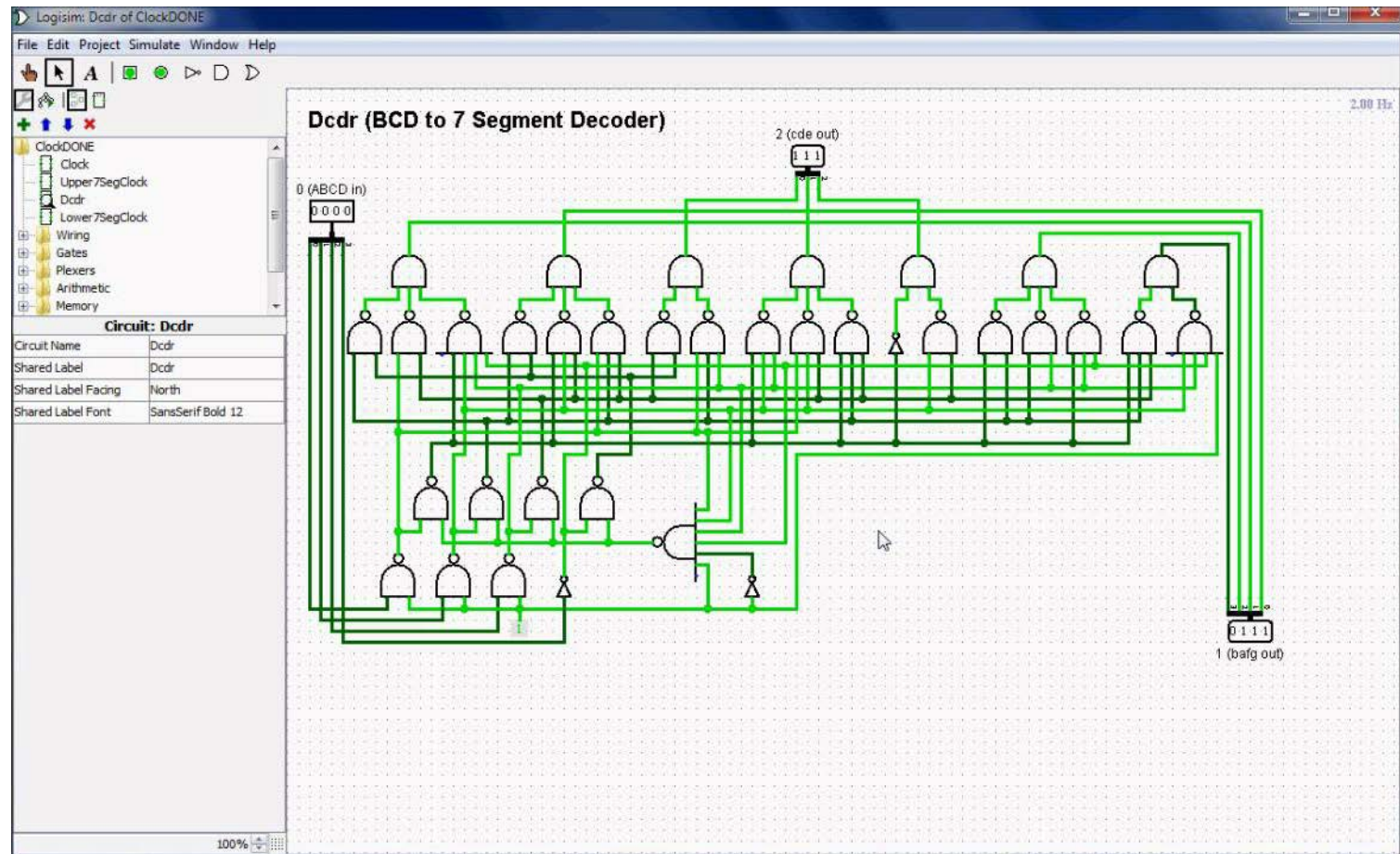
Logisim is an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as you build them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller subcircuits, and to draw bundles of wires with a single mouse drag, Logisim can be used (and is used) to design and simulate entire CPUs for educational purposes.

Logisim is used by students at colleges and universities around the world in many types of classes, ranging from a brief unit on logic in general-education computer science surveys, to computer organization courses, to full-semester courses on computer architecture.

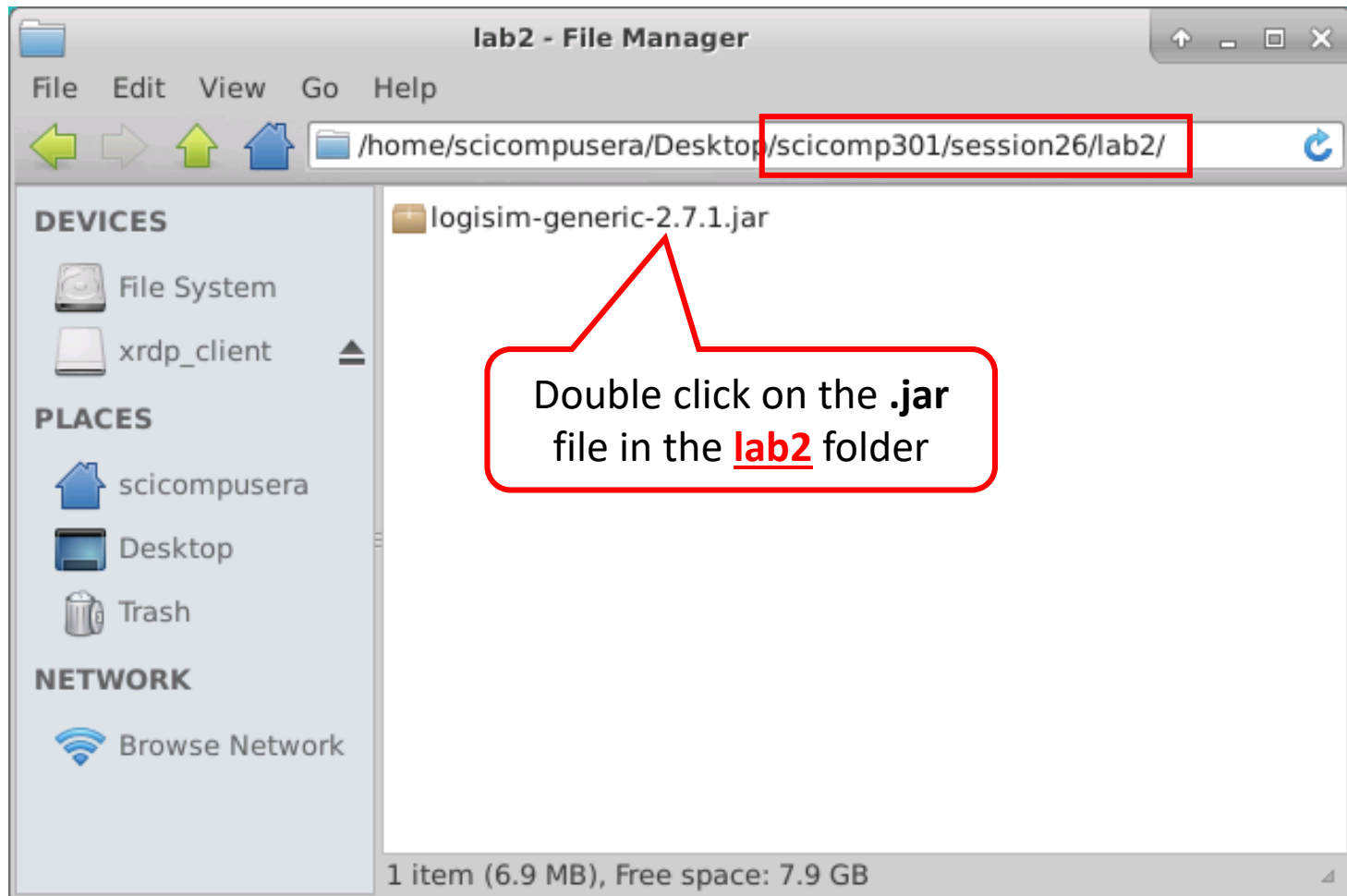
Logisim

a graphical tool for designing
and simulating logic circuits

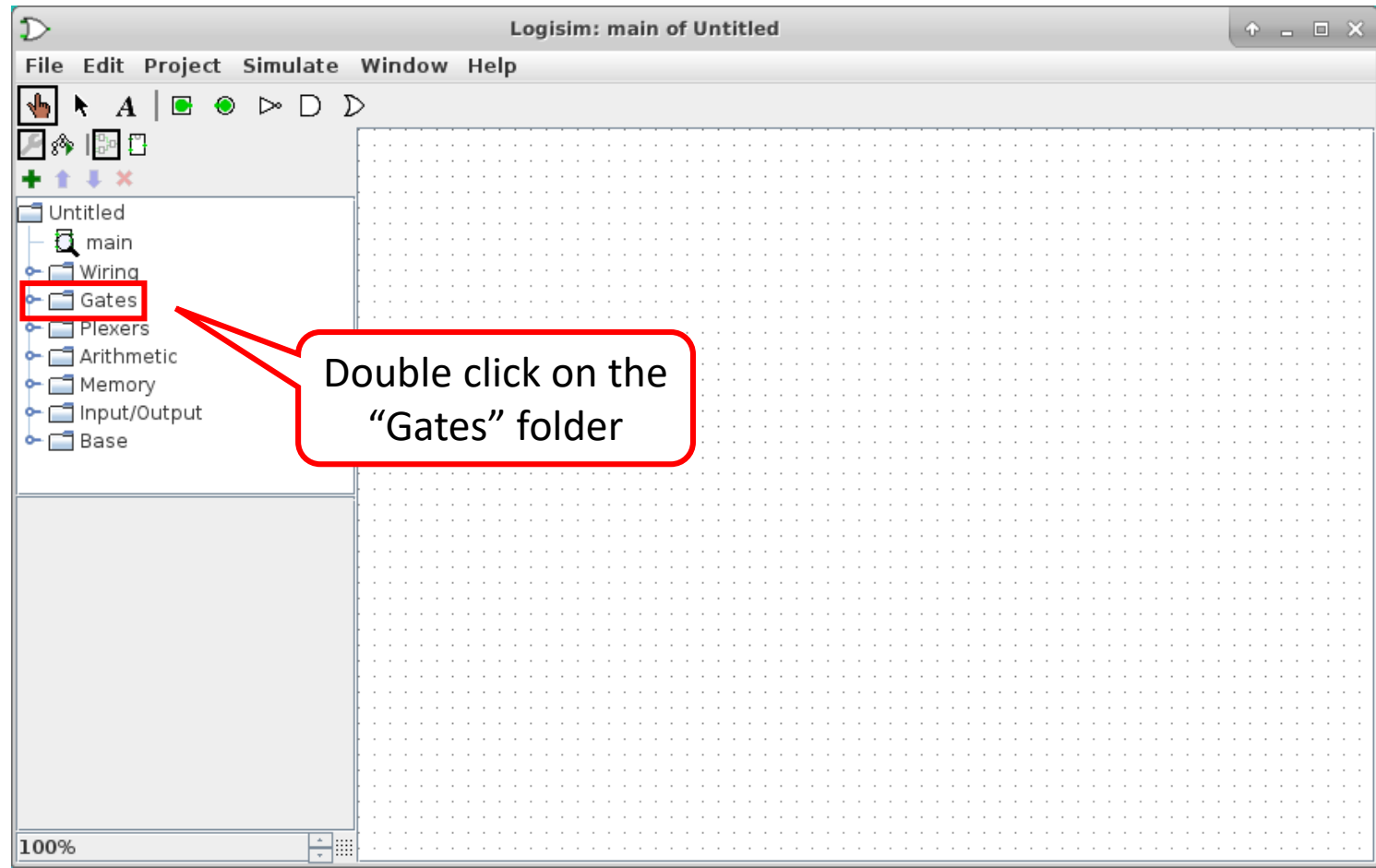
<http://www.cburch.com/logisim>



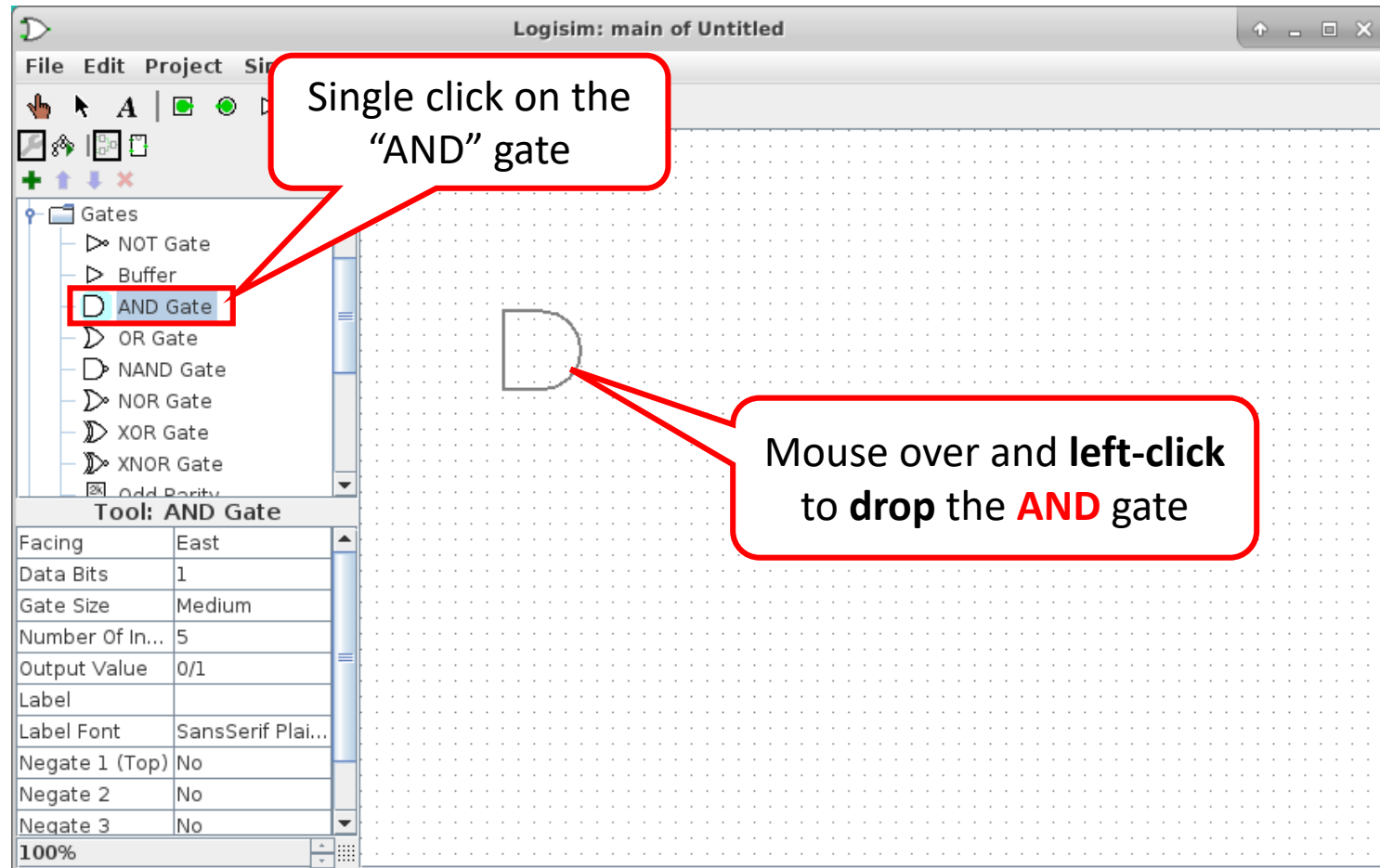
Lab 2 - Using Logisim



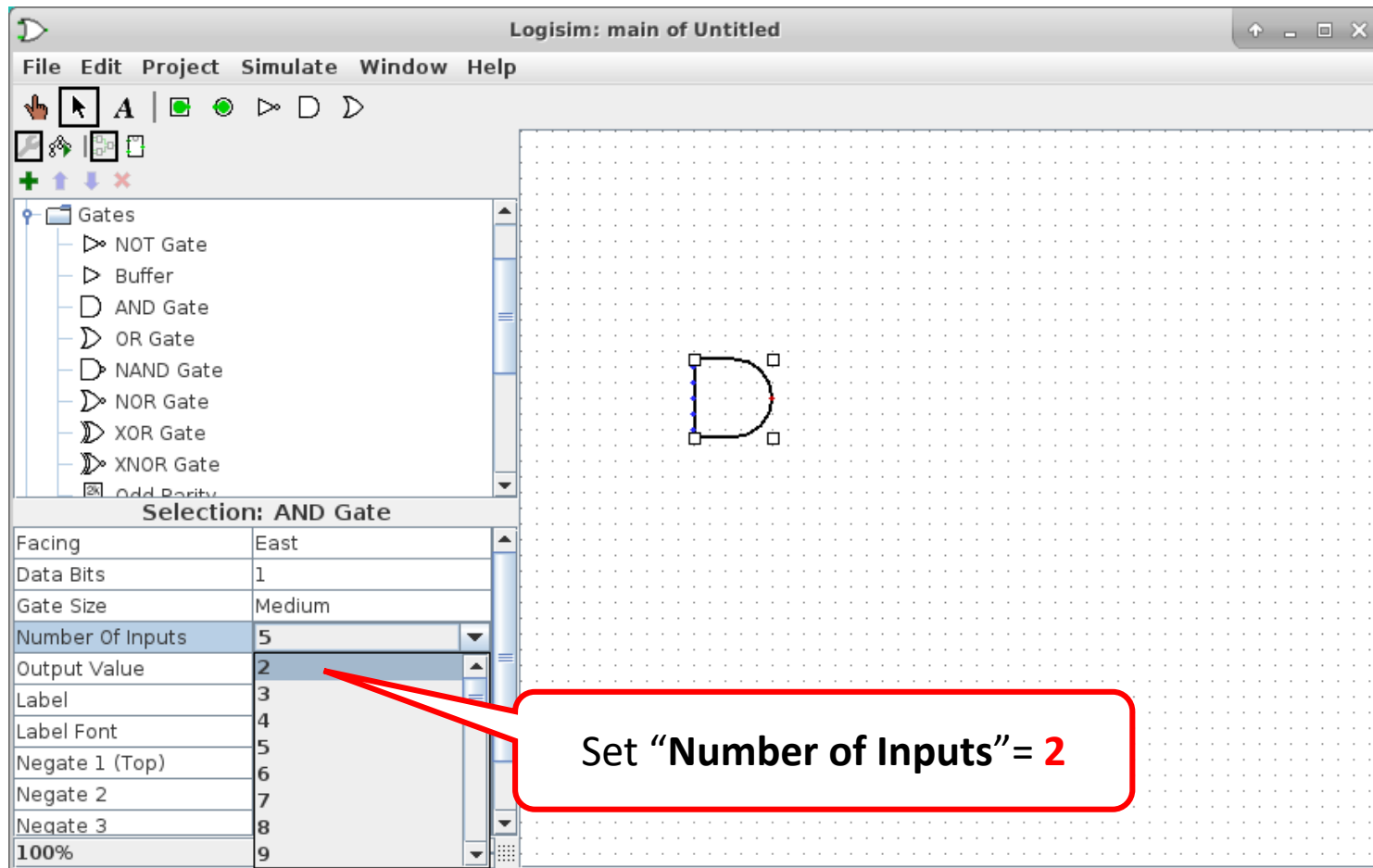
Lab 2 - Using Logisim



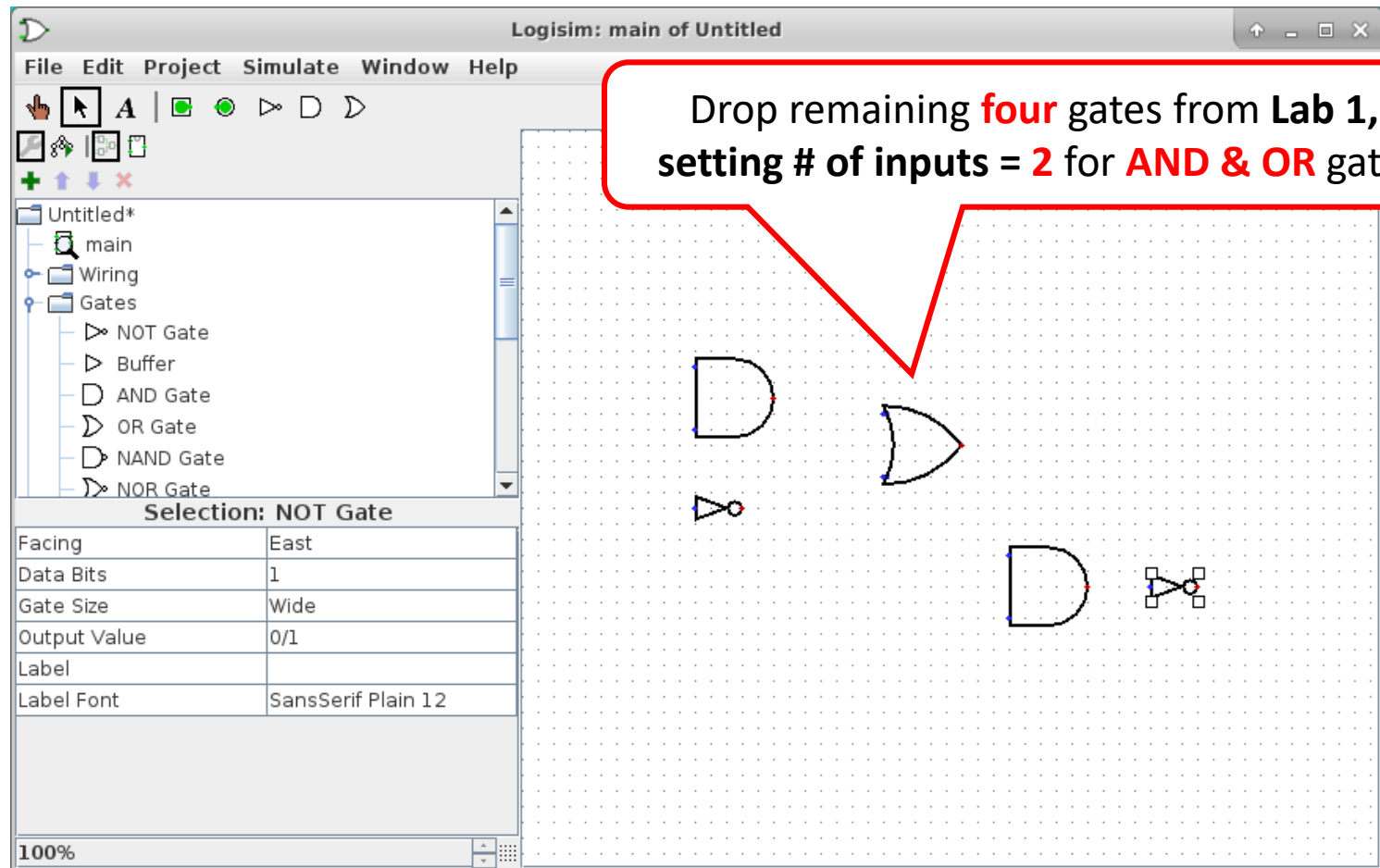
Lab 2 - Using Logisim



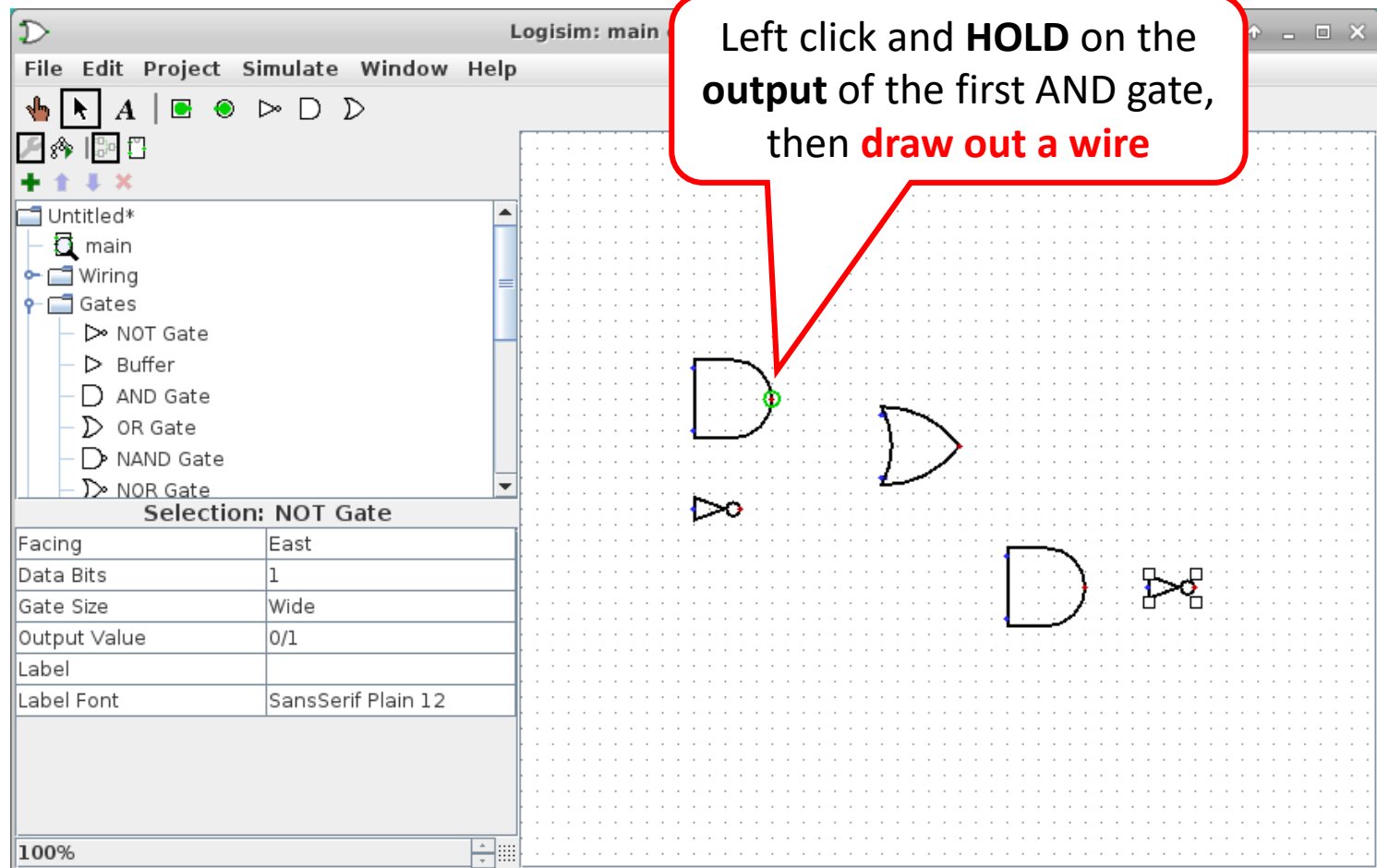
Lab 2 - Using Logisim



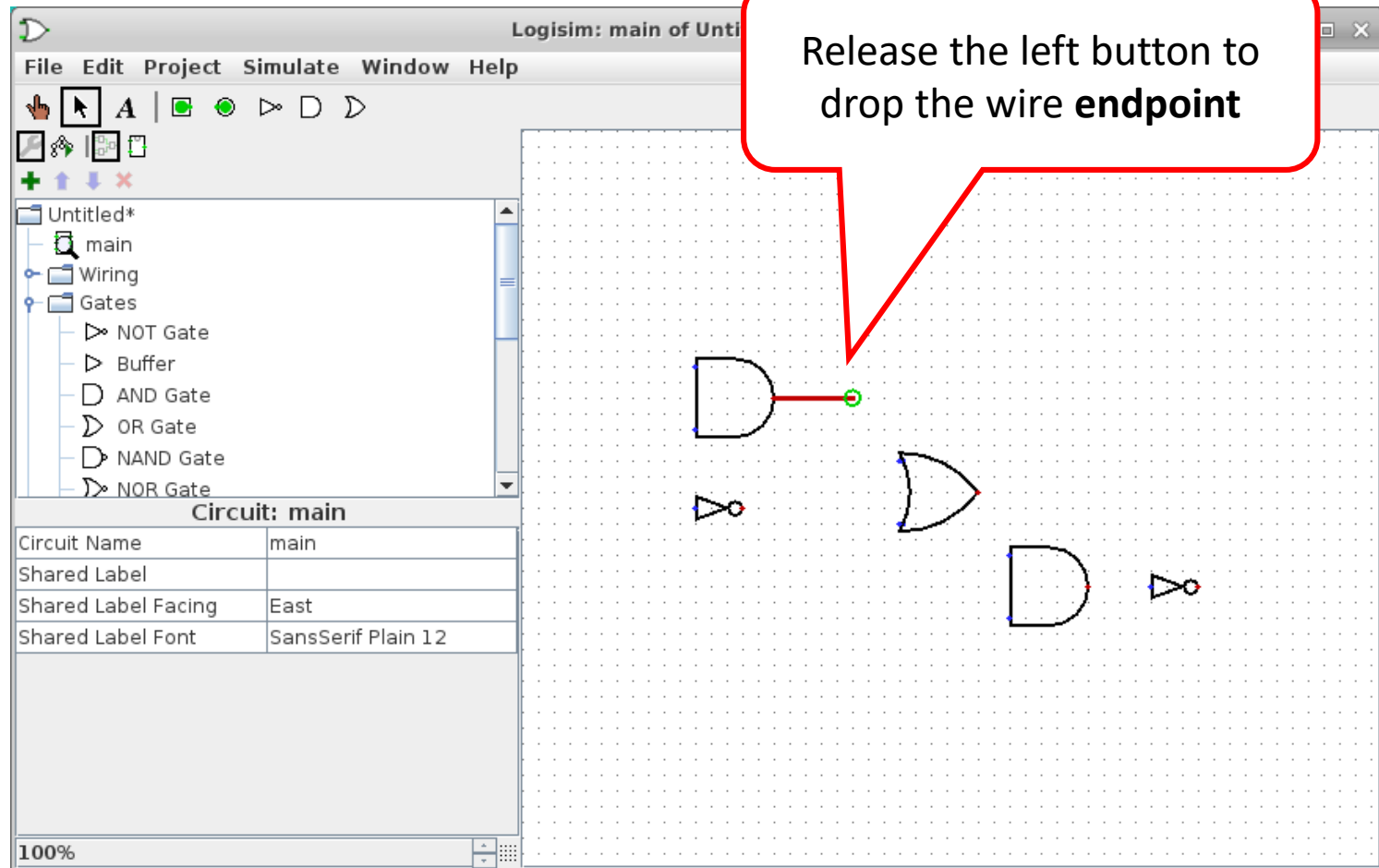
Lab 2 - Using Logisim



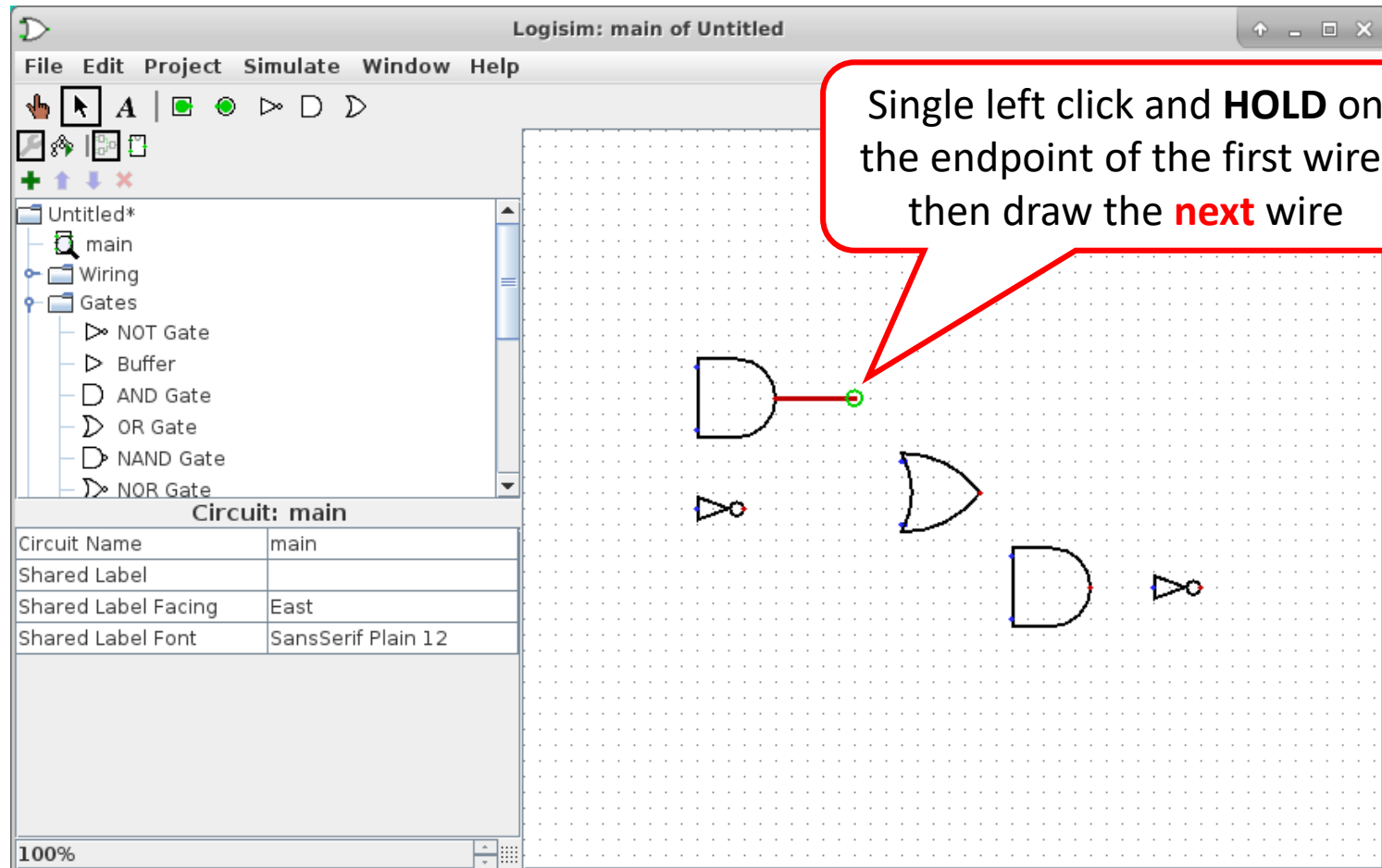
Lab 2 - Using Logisim



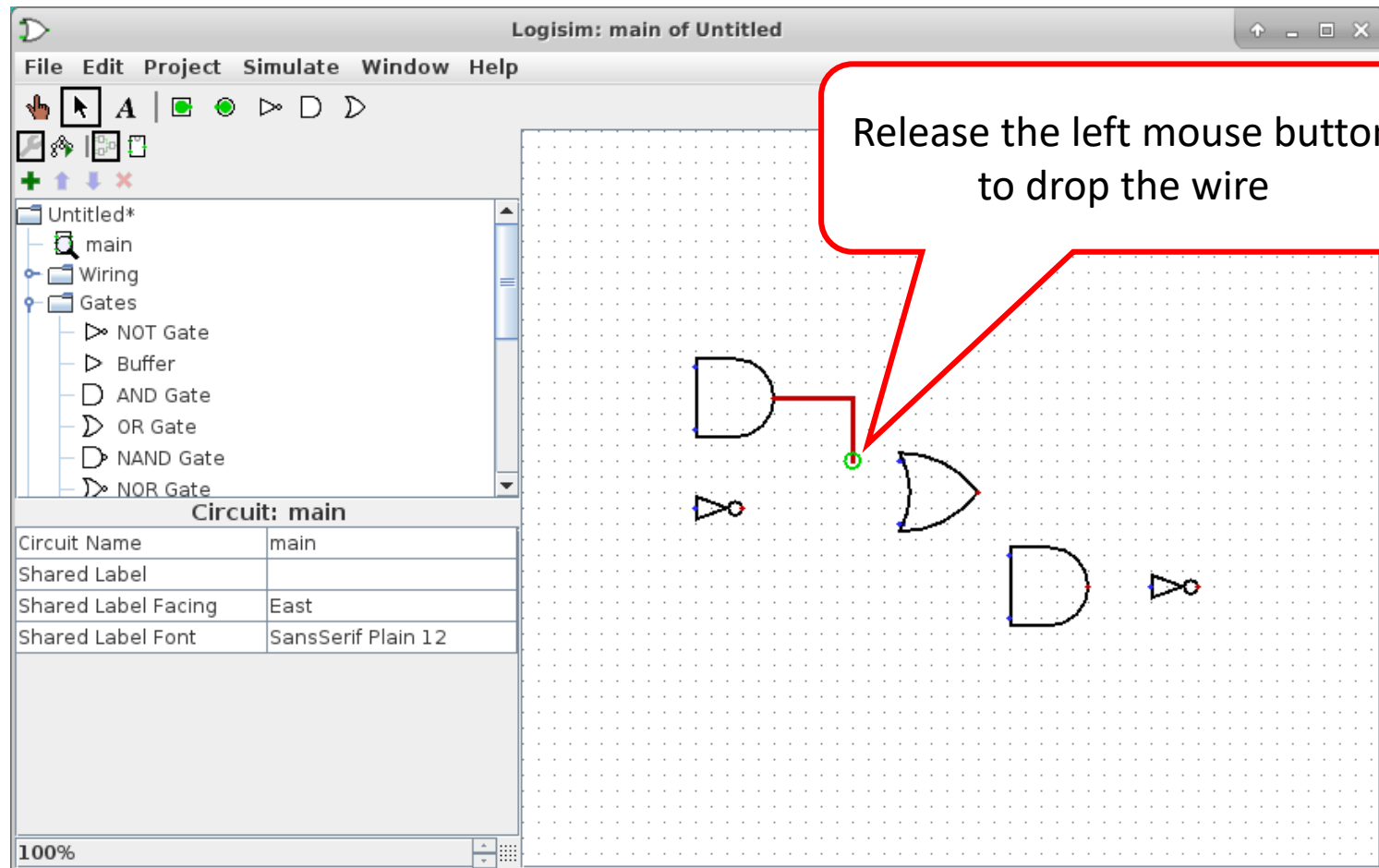
Lab 2 - Using Logisim



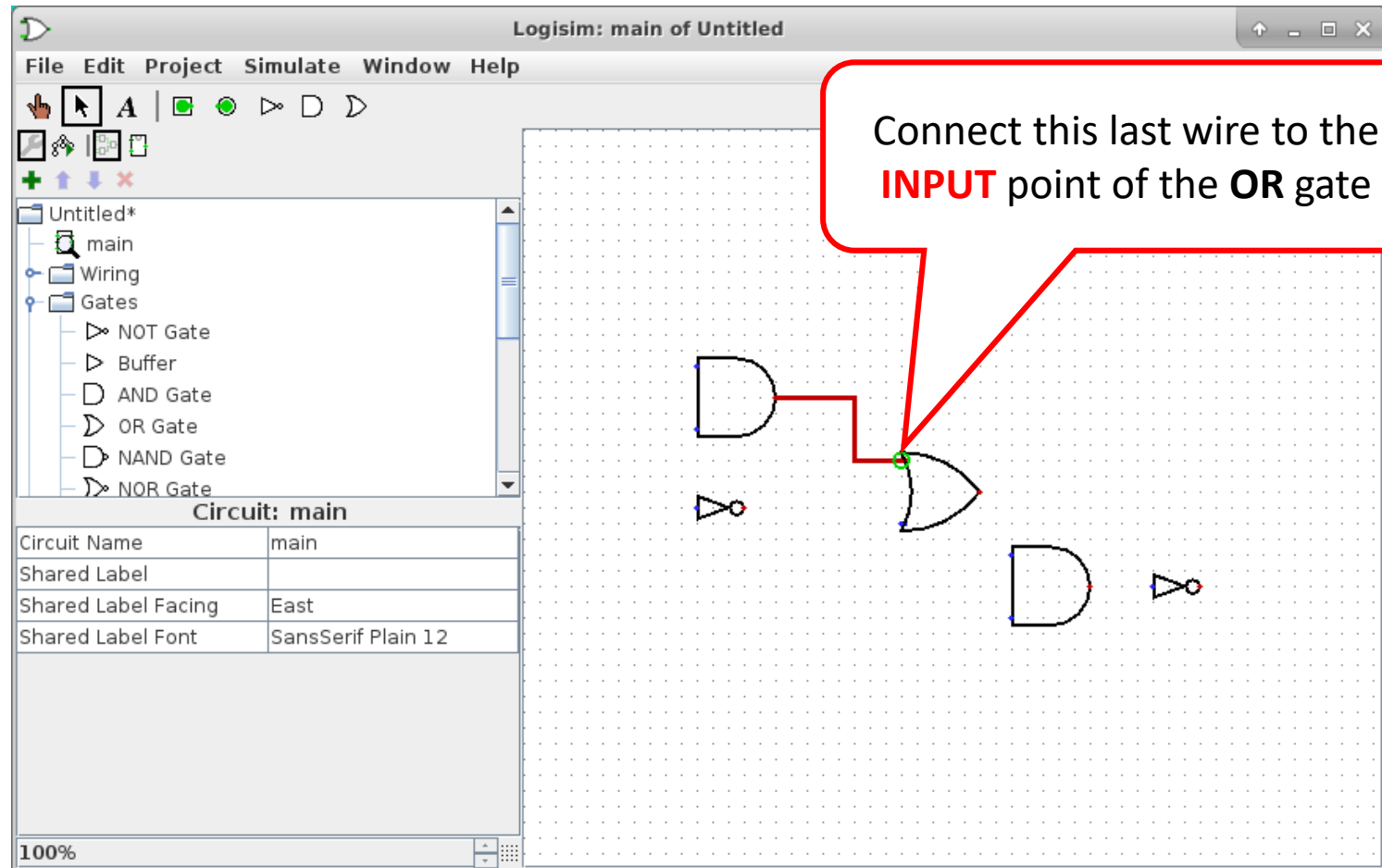
Lab 2 - Using Logisim



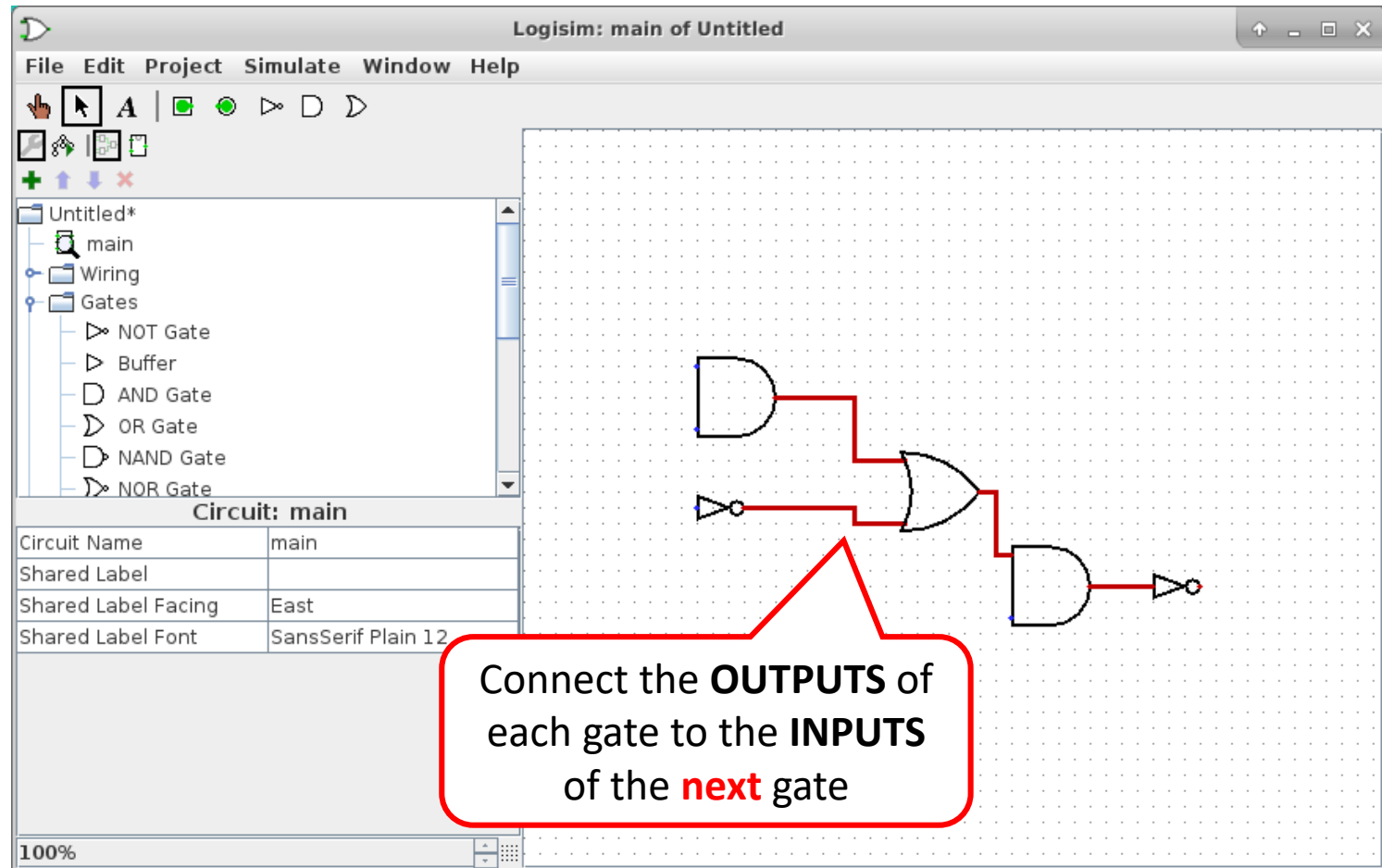
Lab 2 - Using Logisim



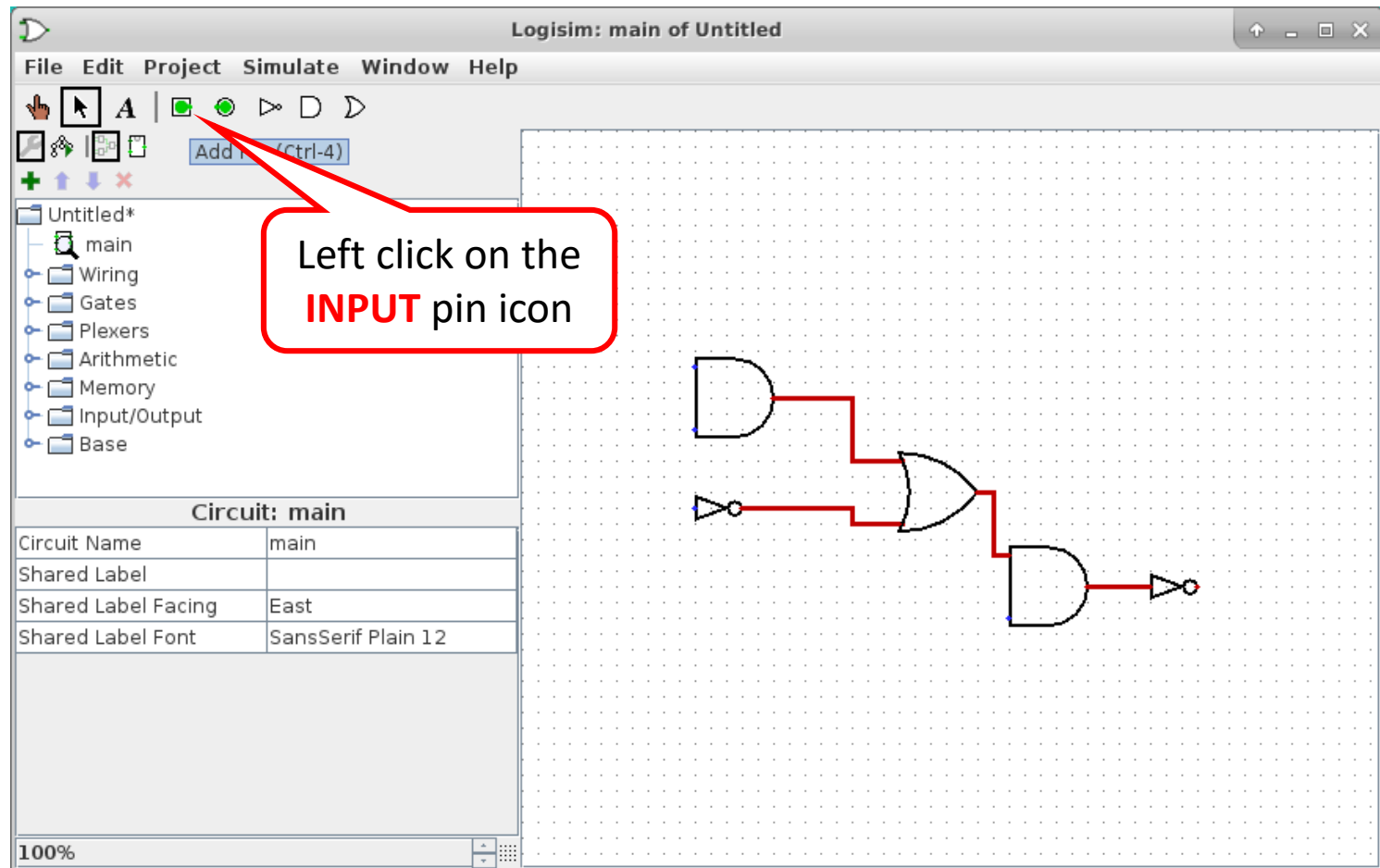
Lab 2 - Using Logisim



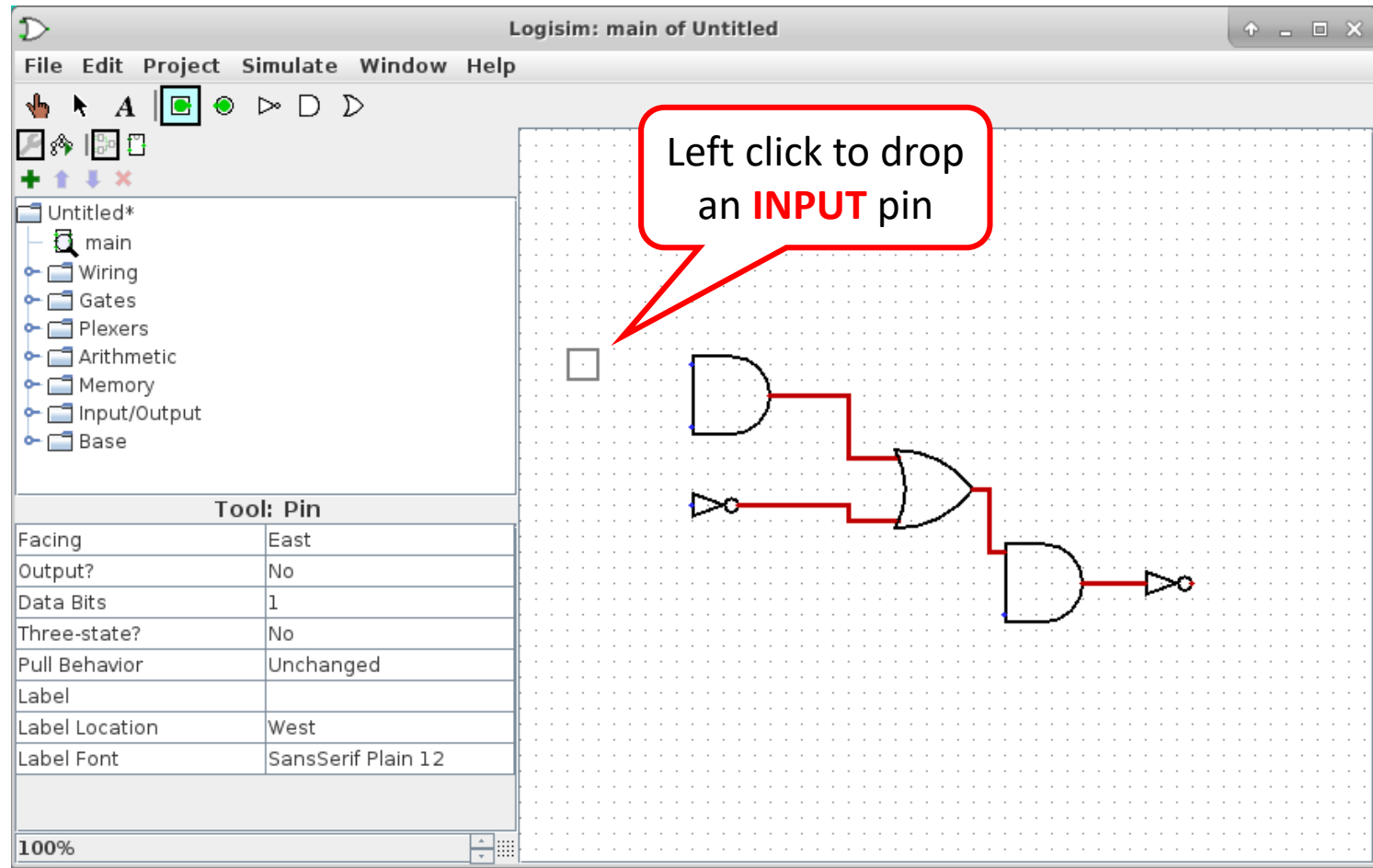
Lab 2 - Using Logisim



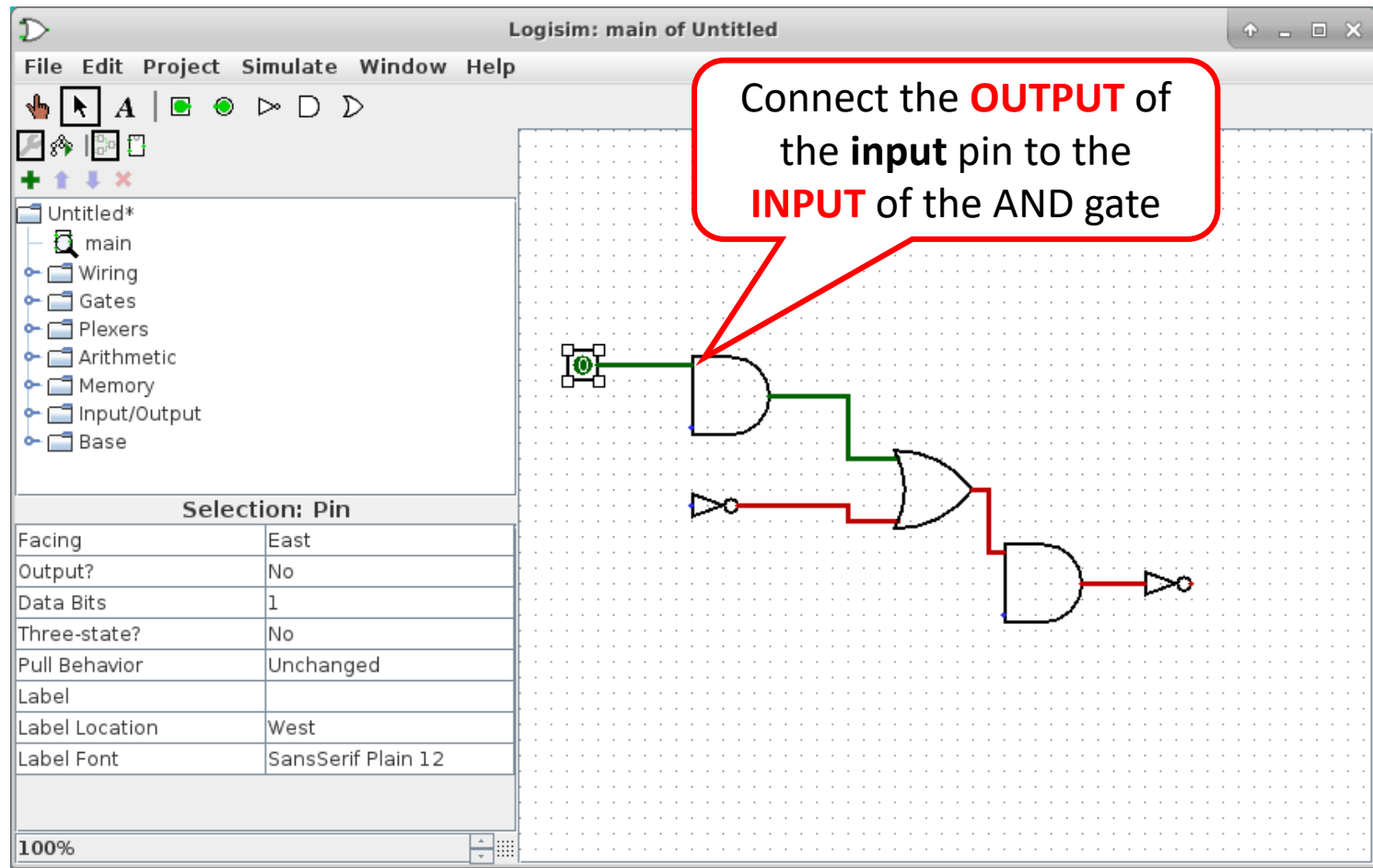
Lab 2 - Using Logisim



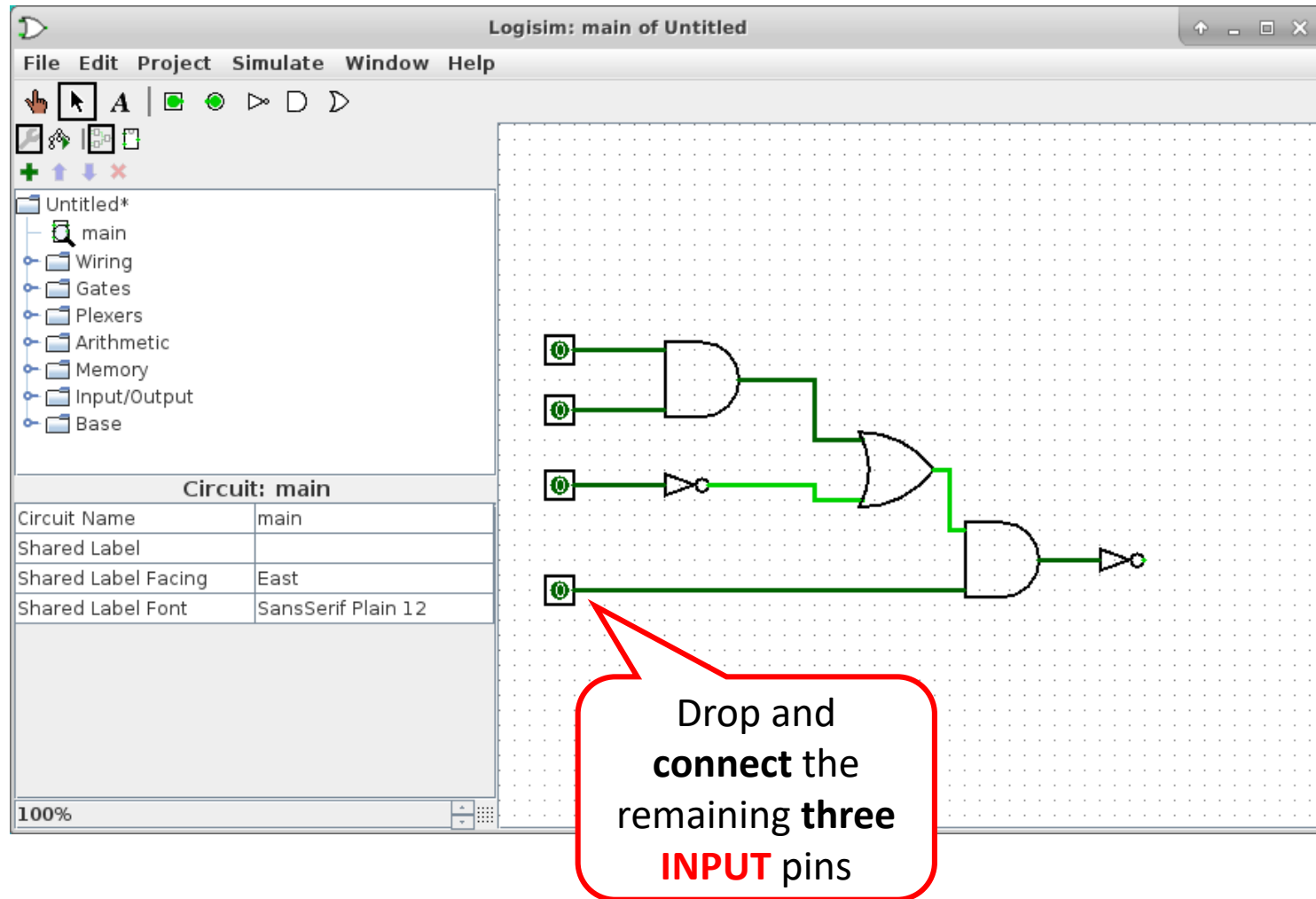
Lab 2 - Using Logisim



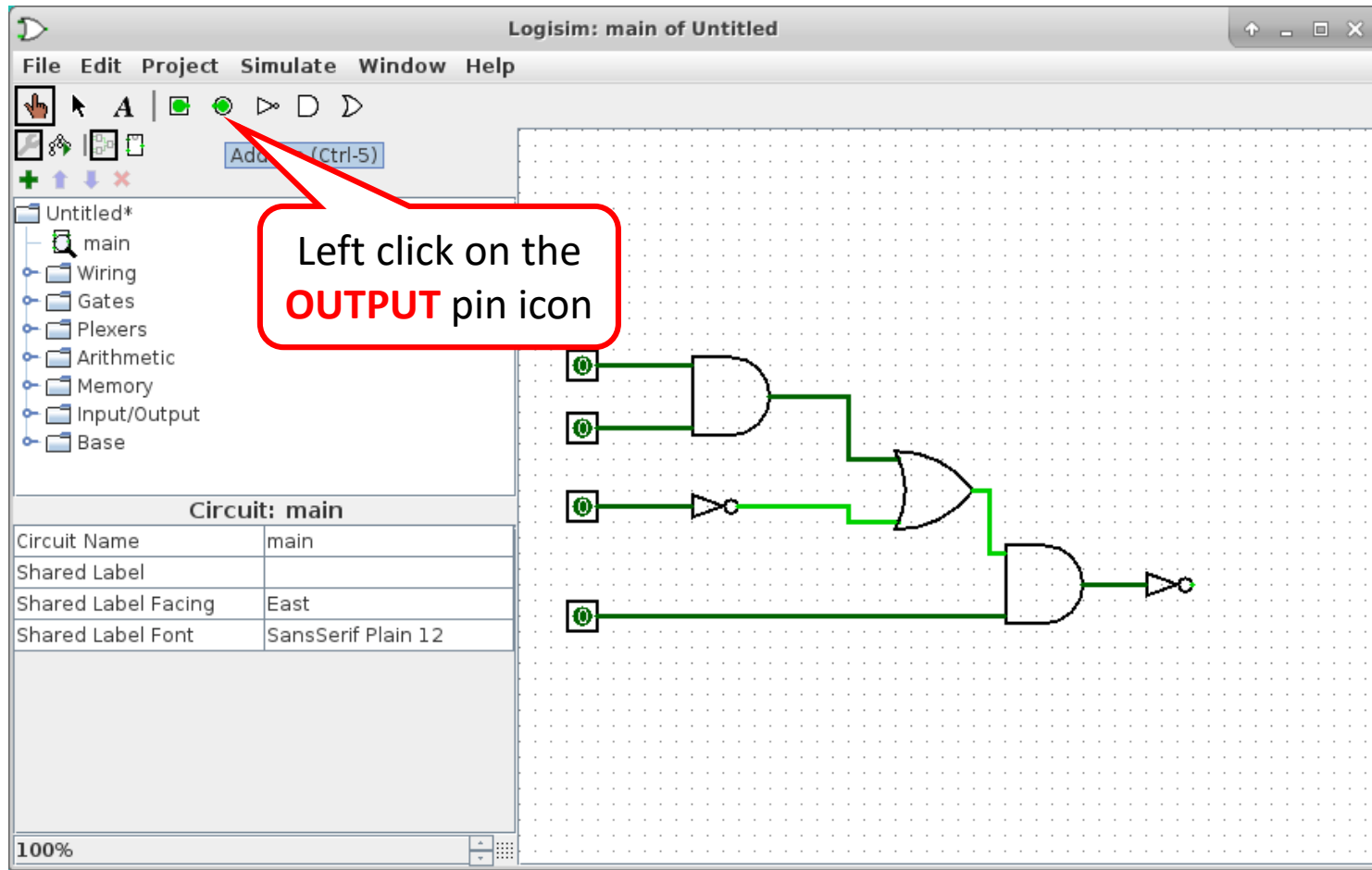
Lab 2 - Using Logisim



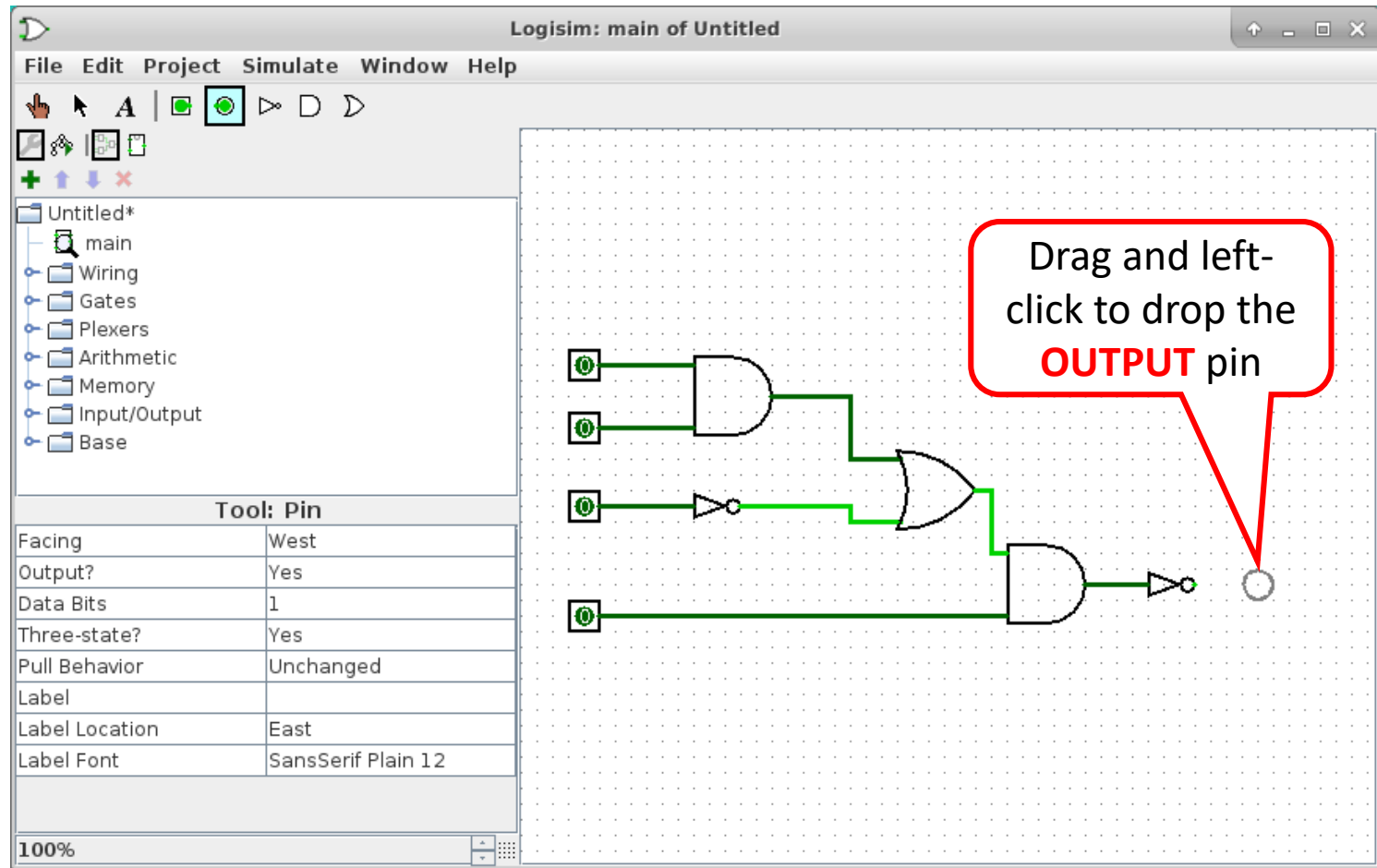
Lab 2 - Using Logisim



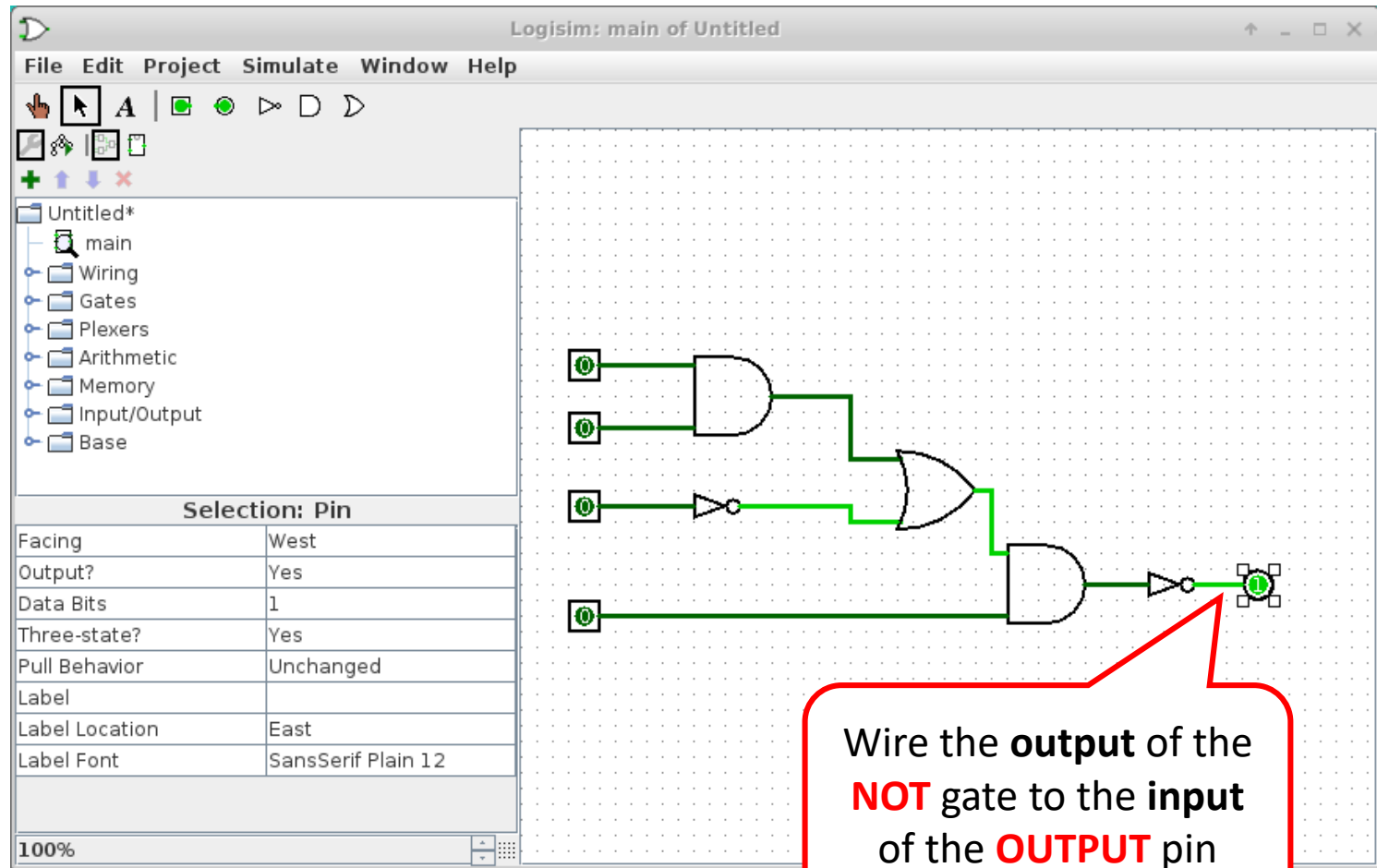
Lab 2 - Using Logisim



Lab 2 - Using Logisim

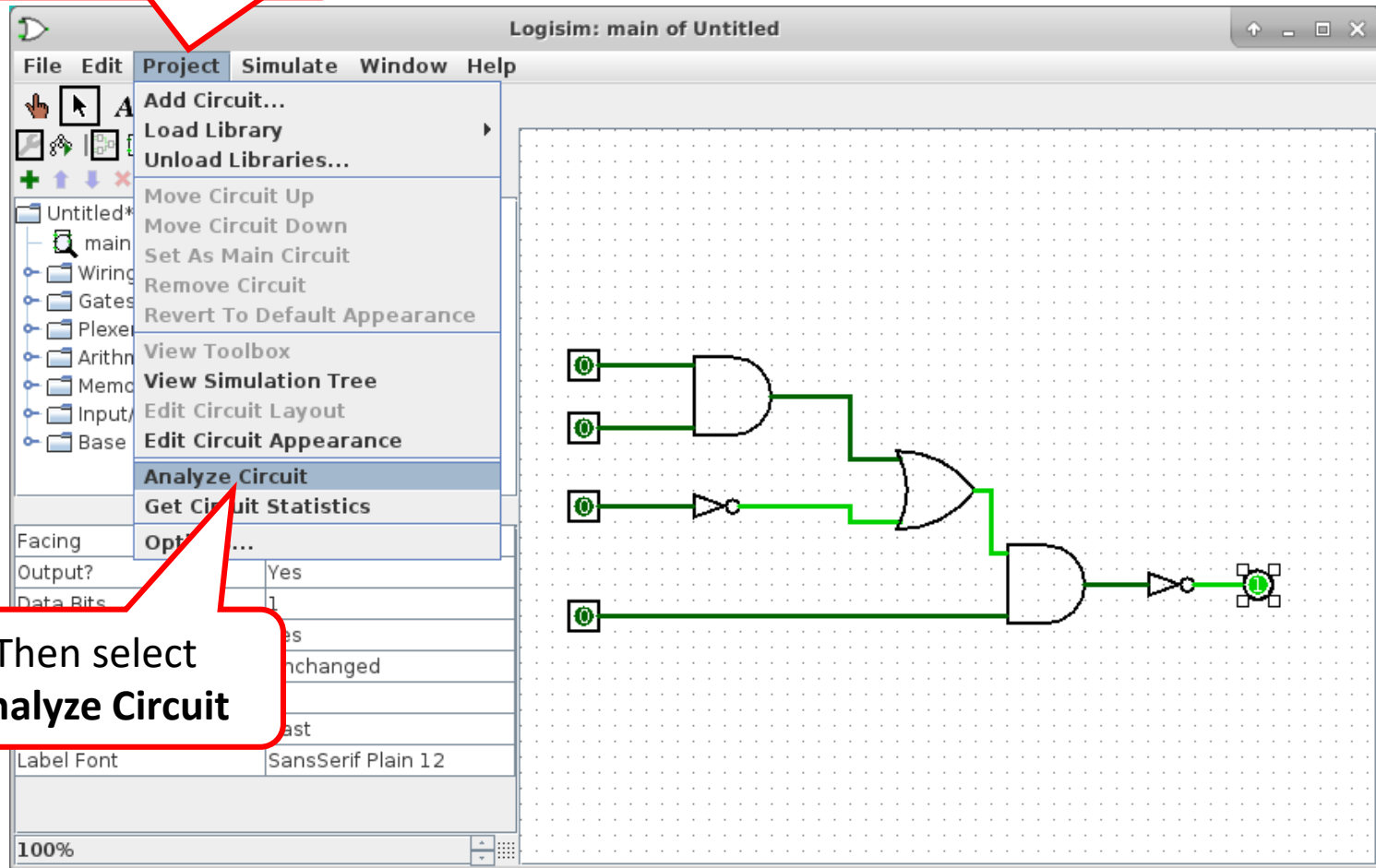


Lab 2 - Using Logisim

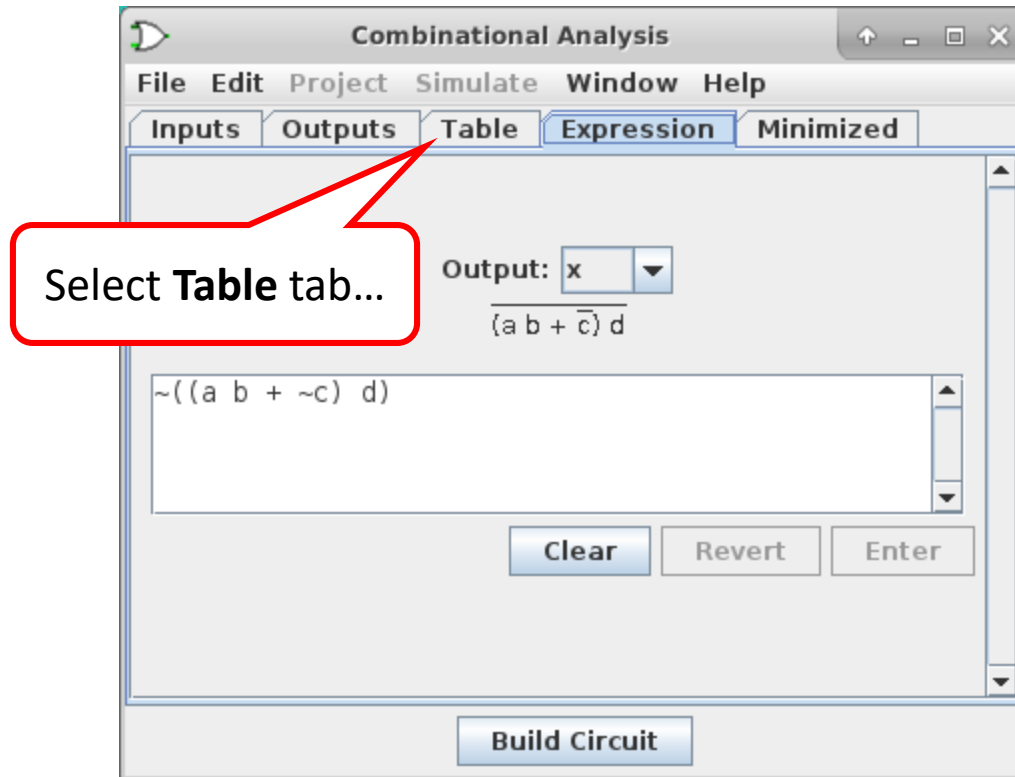


Lab 2 - Using Logisim

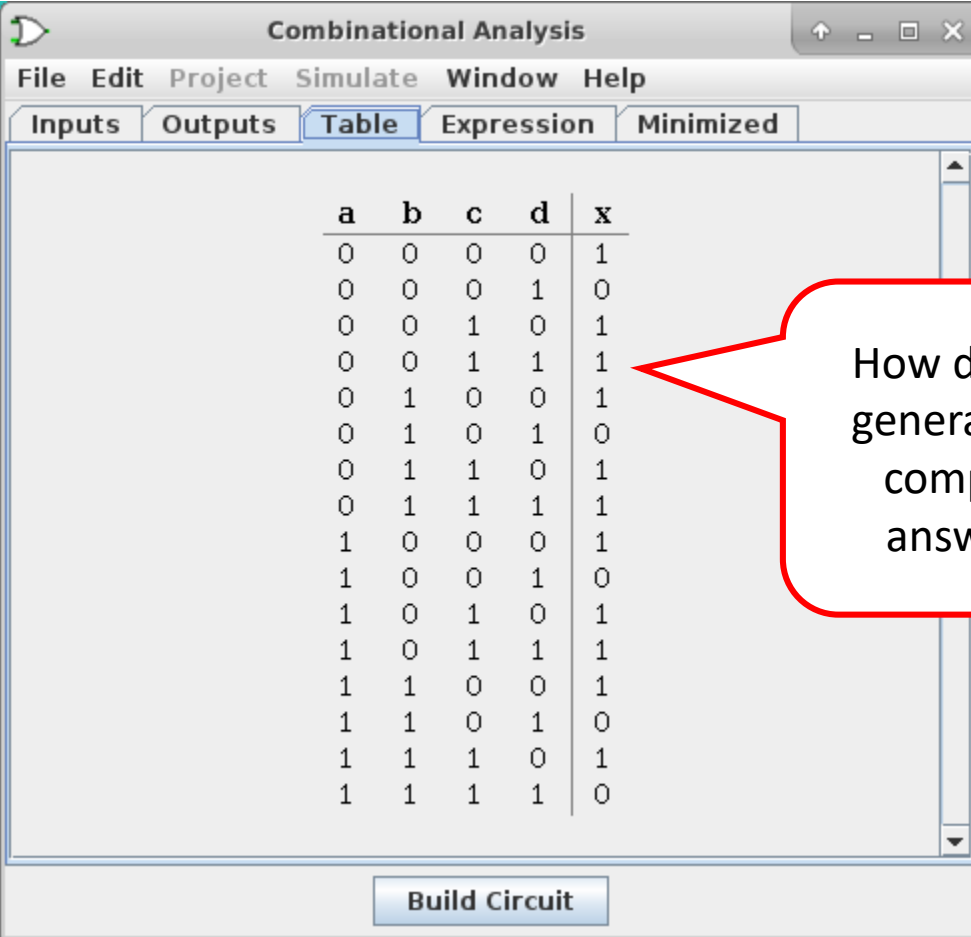
Select **Project** menu...



Lab 2 - Using Logisim



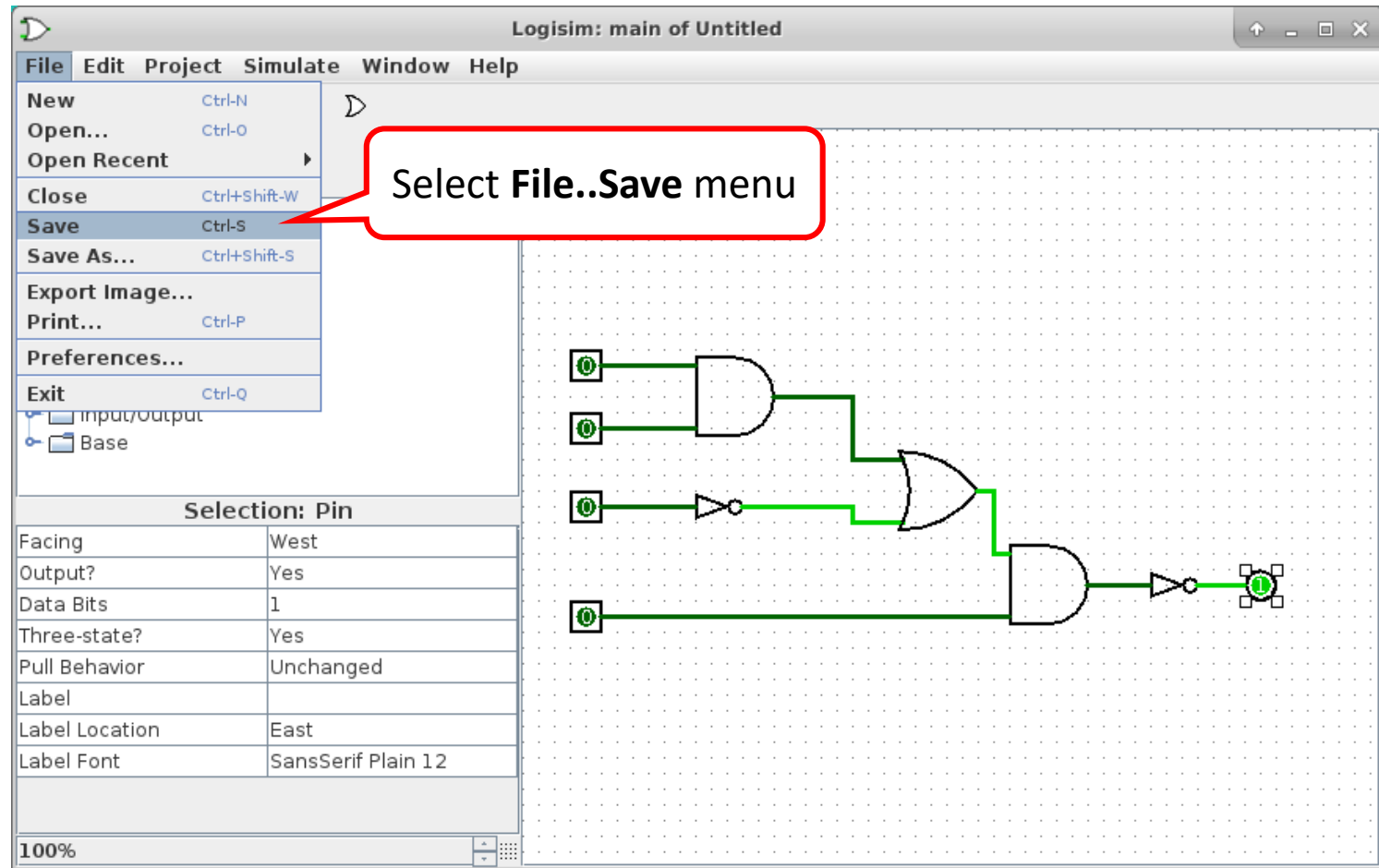
Lab 2 - Using Logisim



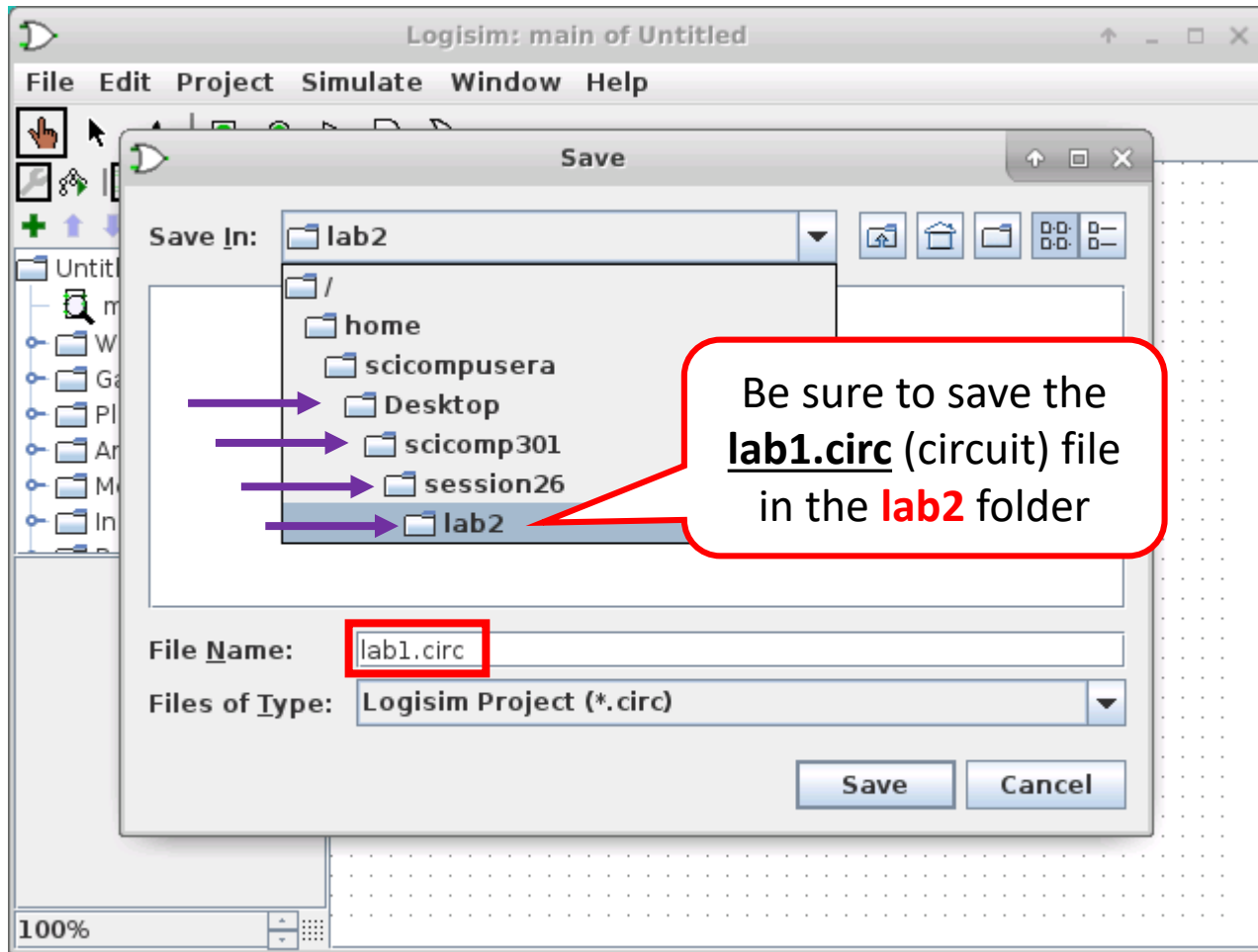
The screenshot shows the 'Combinational Analysis' window in Logisim. The 'Table' tab is selected, displaying a truth table with 5 columns: 'a', 'b', 'c', 'd', and 'x'. The table contains 16 rows of binary data. A red speech bubble points to the table with the text: 'How does the **Logisim** generated **Truth Table** compare with your answers for **Lab 1**?'. At the bottom of the window is a 'Build Circuit' button.

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

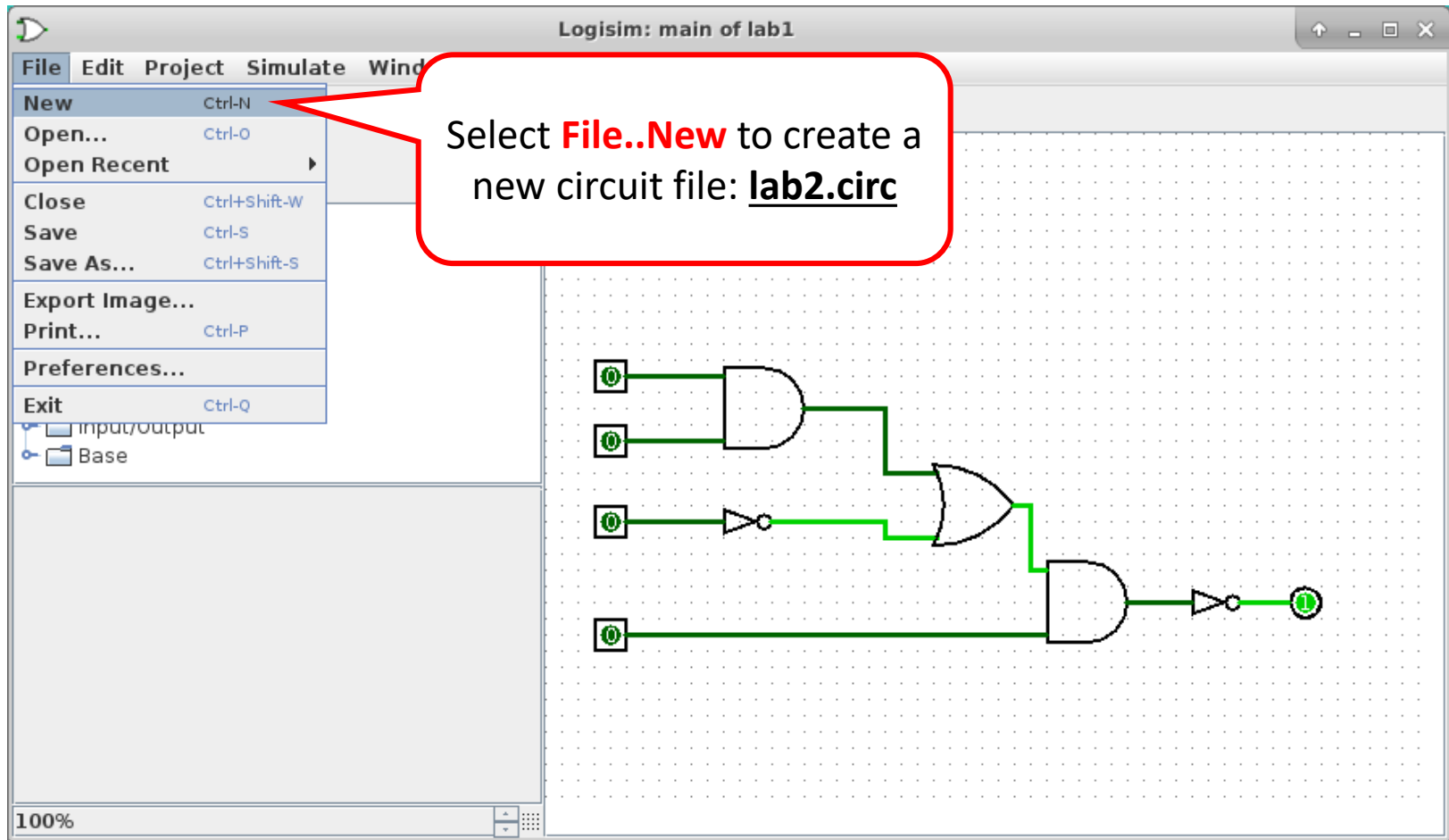
Lab 2 - Using Logisim



Lab 2 - Using Logisim



Lab 2 - Using Logisim



Lab 2 – Majority Voting (2 of 3)

- Create a truth table with input variables (A, B, C) that represent the vote (yeah or nay) of **three** people
- Design a circuit that emits **1** as output *if and only if* at least 2 out of the 3 input lines are also **1**
 - Start out by making a truth table for all 8 possible input permutations
 - Then use **AND** gates to select only those input permutations that that represent valid (1/high/true) “majority” output
 - Then use **OR** gates to gather the output of those **AND** gates into a single output line

Lab 2 – Majority Voting (2 of 3)

2 of 3 Majority Voting Truth Table

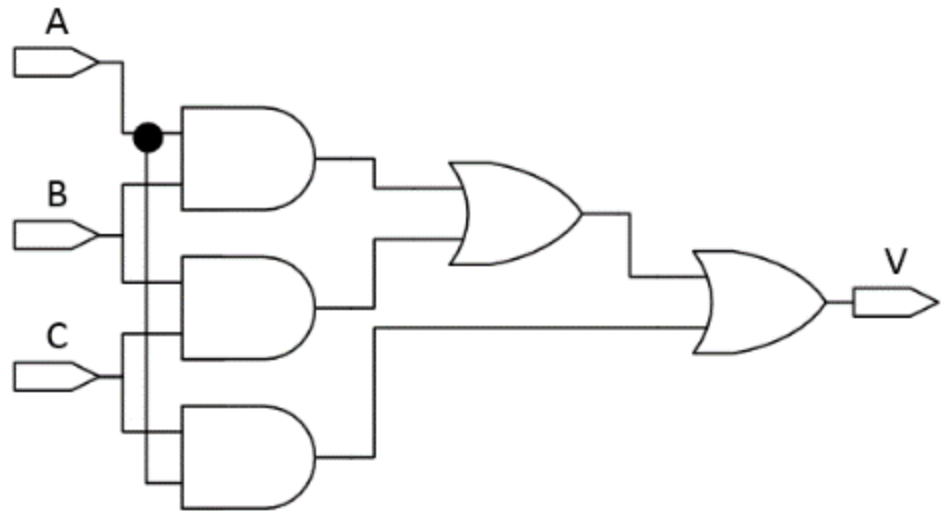
INPUT			OUTPUT
A	B	C	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Lab 2 – Majority Voting (2 of 3)

2 of 3 Majority Voting Truth Table

INPUT			OUTPUT
A	B	C	V
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



Lab 2 – Majority Voting (2 of 3)

Be sure to save your **lab2.circ** in the **lab2** folder

The screenshot displays the Logisim software interface. The main window shows a circuit with three input switches on the left, each connected to a 0V ground symbol. These inputs are connected to three 3-input AND gates. The outputs of these AND gates are connected to three 3-input OR gates. The outputs of these OR gates are connected to a single 3-input OR gate, which is connected to a 0V ground symbol. The bottom panel shows the 'Tool: main' settings, including 'Facing', 'Label', 'Label Location', 'Label Font', 'Circuit Name', 'Shared Label', 'Shared Label Facing', and 'Shared Label Font'. The 'Combinational Analysis' window is open, showing a truth table with columns 'a', 'b', 'c', and 'x'. The table lists all possible combinations of inputs and the resulting output 'x'.

a	b	c	x
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

Build Circuit

Binary Addition

• $5_{\text{ten}} + 6_{\text{ten}}$

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ (5_{\text{ten}}) \\
 +\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110\ (6_{\text{ten}}) \\
 \hline
 =\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011\ (11_{\text{ten}})
 \end{array}$$

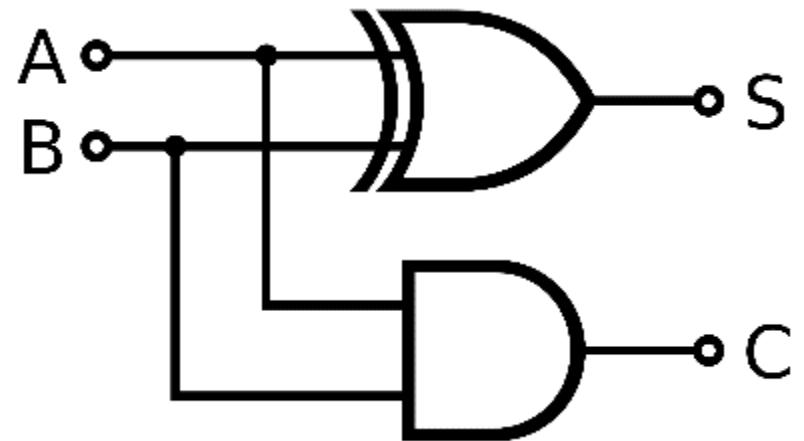
Carries

$$\begin{array}{r}
 \dots (0) \quad (1) \quad (0) \quad (0) \quad (0) \\
 \dots 0 \quad 0 \quad 1 \quad 0 \quad 1 \\
 + \dots 0 \quad 0 \quad 1 \quad 1 \quad 0 \\
 \hline
 \dots 0 \quad (0)1 \quad (1)0 \quad (0)1 \quad (0)1
 \end{array}$$

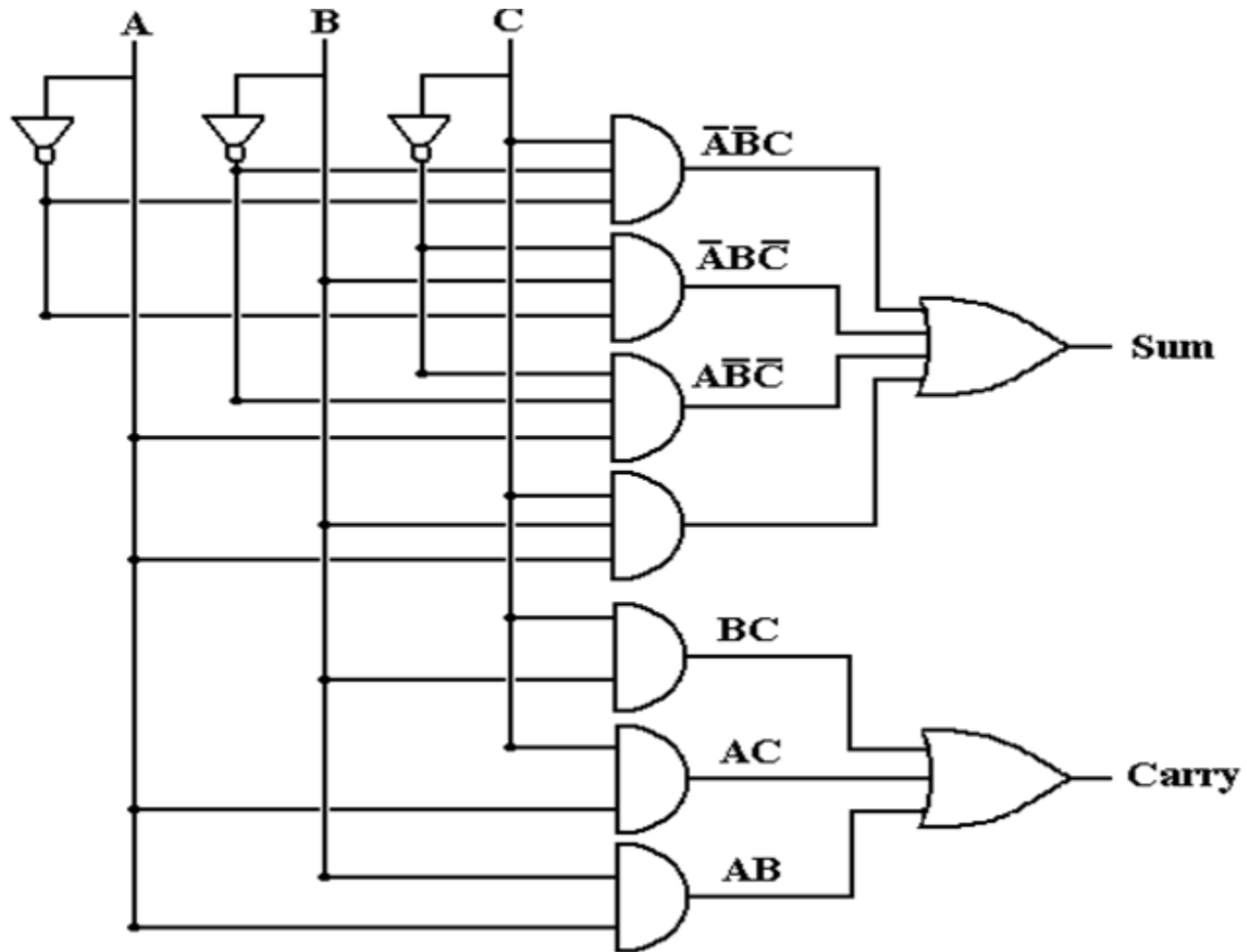
$$\begin{array}{r}
 0\ 1\ 1\ 1 \\
 00111 \quad 7 \\
 10101 \quad 21 \\
 \hline
 11100 = 28
 \end{array}$$

Half-Adder Circuit

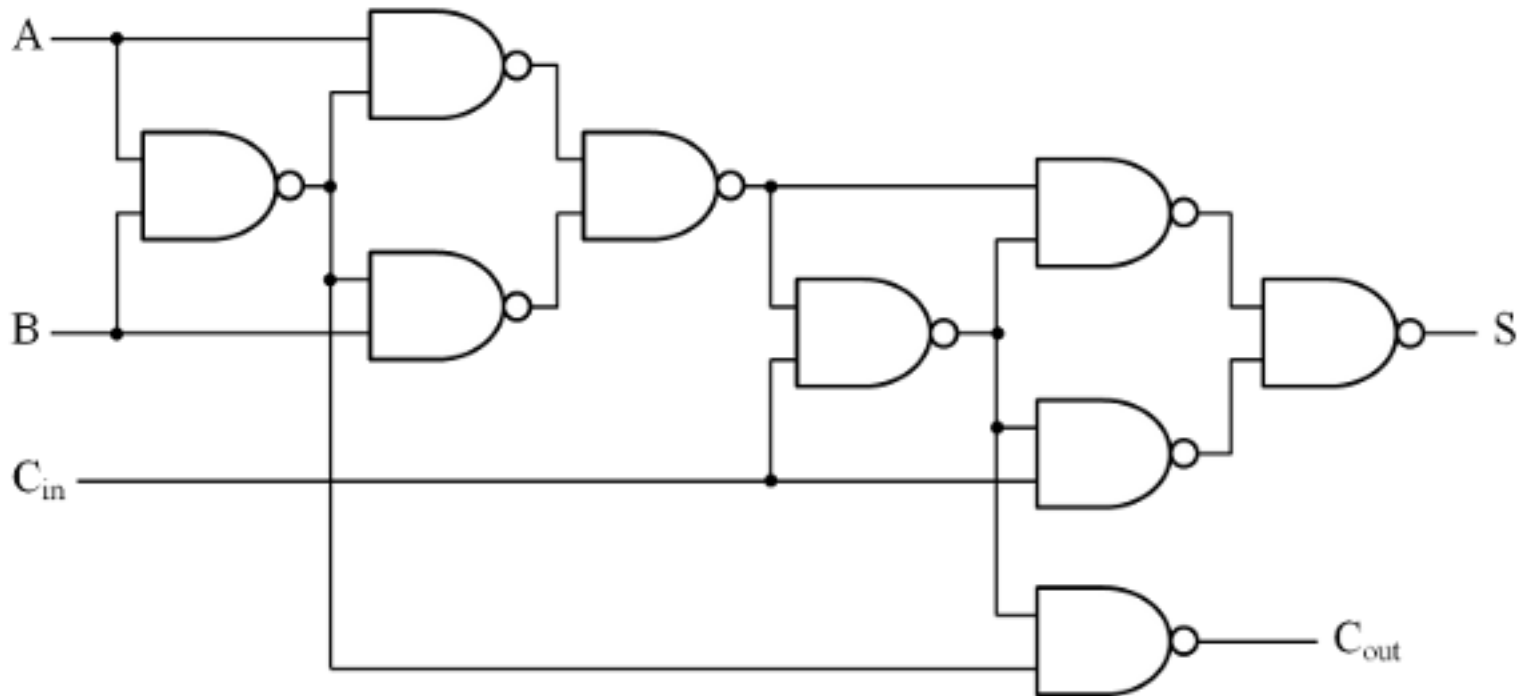
Half ADDER Truth Table				
INPUT			OUTPUT	
A	B		S	C _{out}
0	0		0	0
0	1		1	0
1	0		1	0
1	1		0	1



Full Adder Circuit



Full Adder Circuit using **NAND** gates



Lab 3

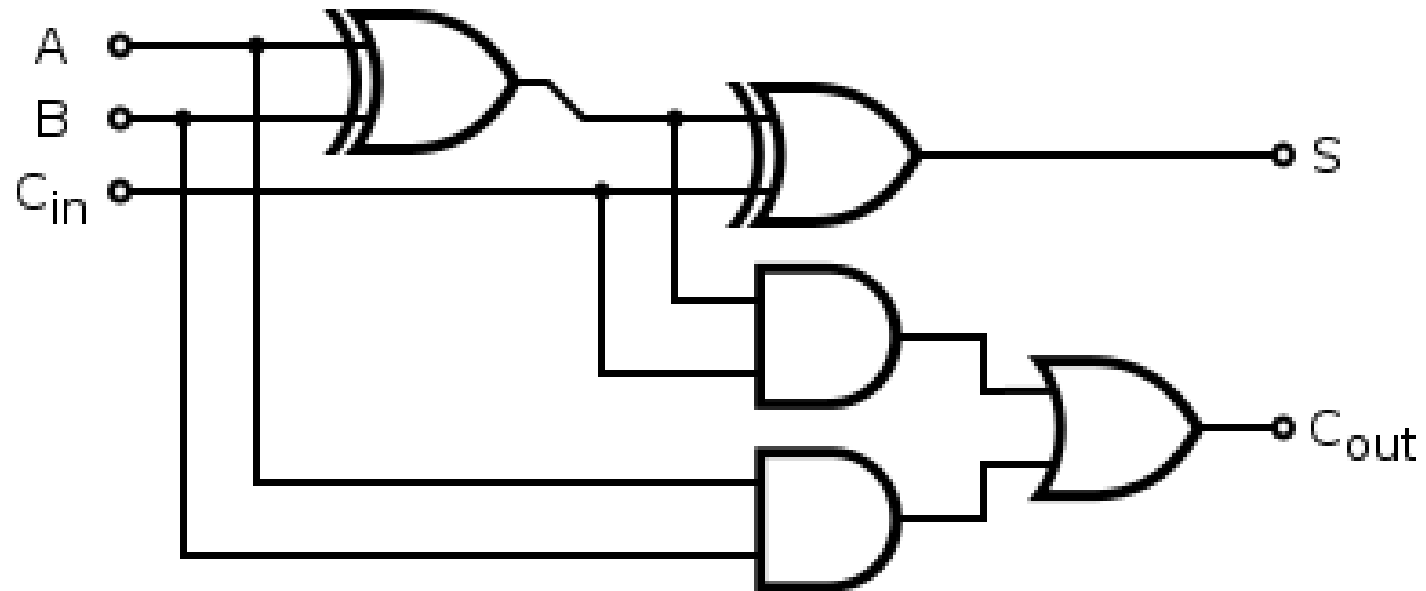
- Using only 1 **OR**, 2 **XOR**, and 2 **AND** gates, design a **FULL ADDER** circuit
- The circuit has **3 input lines**, and **2 output lines** to indicate the sum and if there needs to be a carry to the next column
- Consider how FULL ADDERs can be chained to **sum two 3-bit numbers**

FULL ADDER Truth Table					
INPUT				OUTPUT	
A	B	C _{in}		S	C _{out}
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

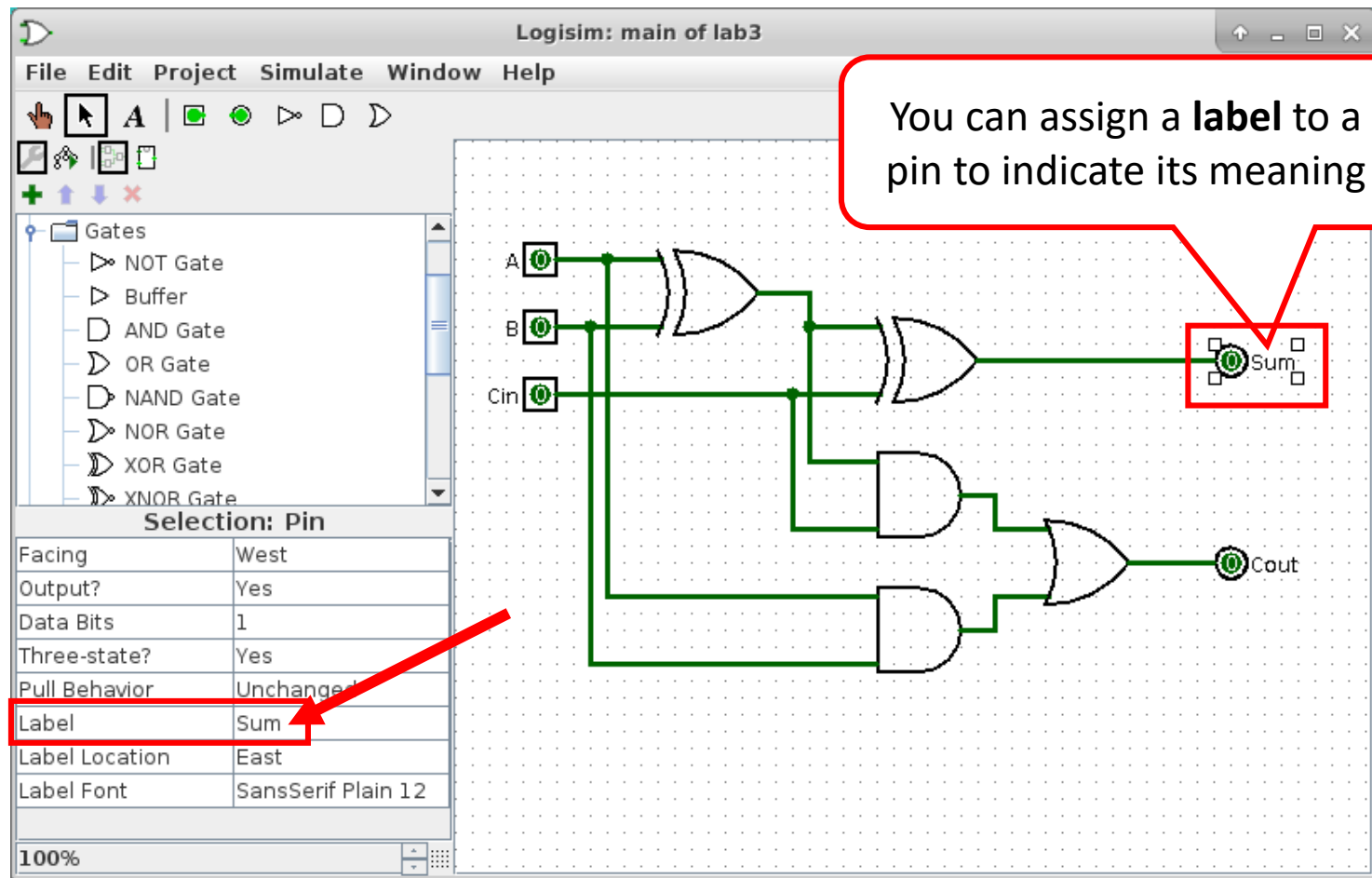
Lab 3 - Full Adder using **XOR** gates



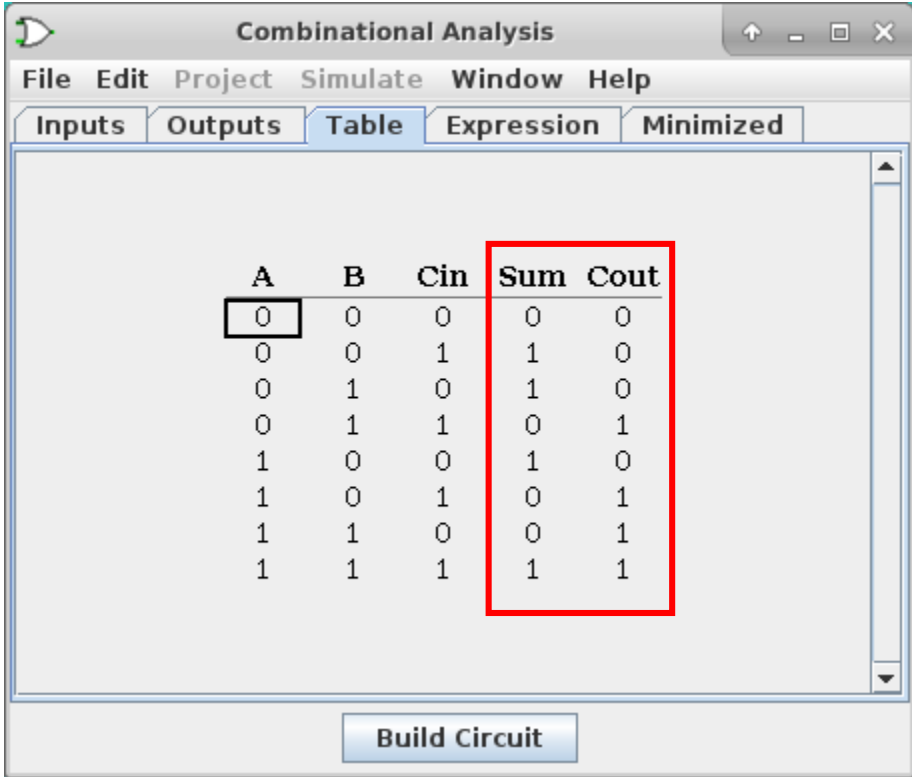
Lab 3 - Full Adder using **XOR** gates



Lab 3 - Full Adder using **XOR** gates



Lab 3 - Full Adder using **XOR** gates



Combinational Analysis

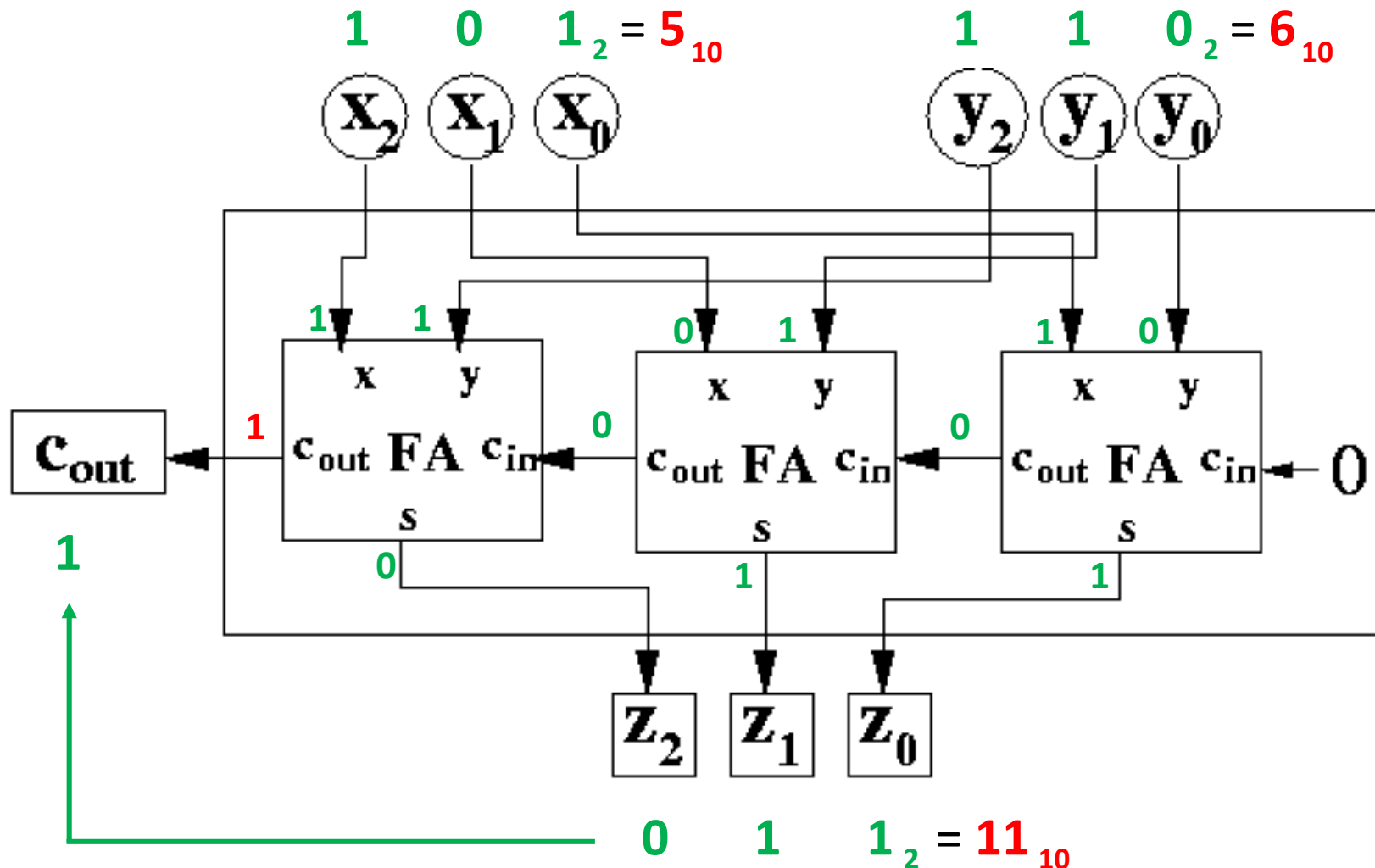
File Edit Project Simulate Window Help

Inputs Outputs Table Expression Minimized

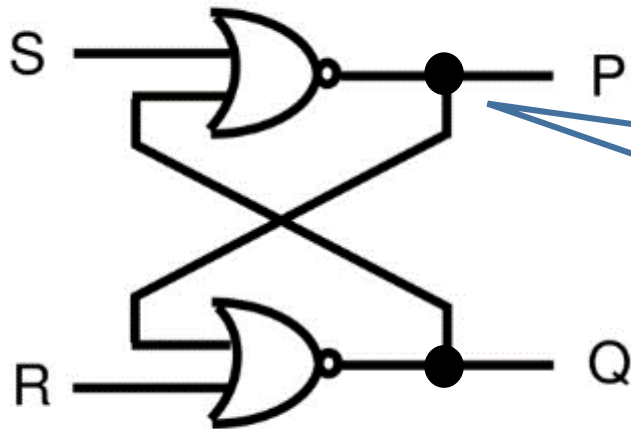
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Build Circuit

Chaining Full Adders with **Ripple** Carry



1 Bit Memory : **Set-Reset** Latch



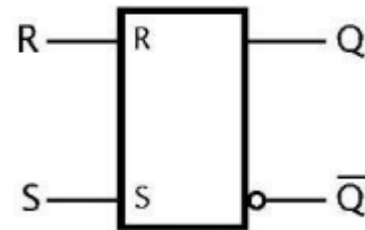
The key was to send the output back into the input!

Input		Output	
S	R	P	Q
0	0	Hold Output	
0	1	1	0
1	0	0	1
1	1	Invalid Input	

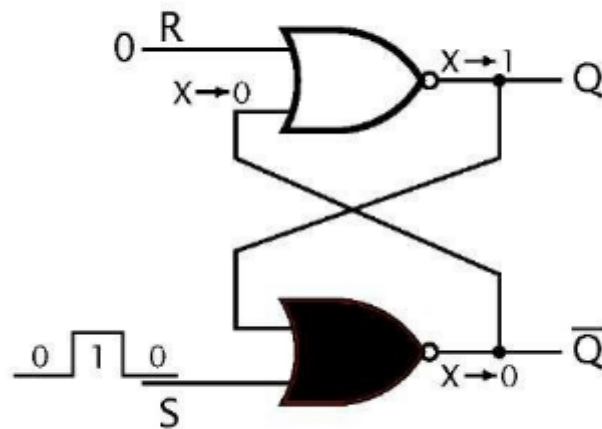
1 Bit Memory : **Set-Reset** Latch

R	S	Q
0	0	Q (no change)
0	1	1 (set)
1	0	0 (reset)

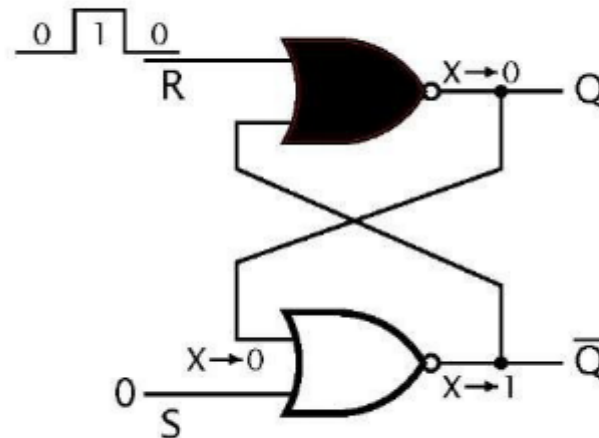
(a) Defining RS latch truth table



(b) Logic symbol with true/complement outputs

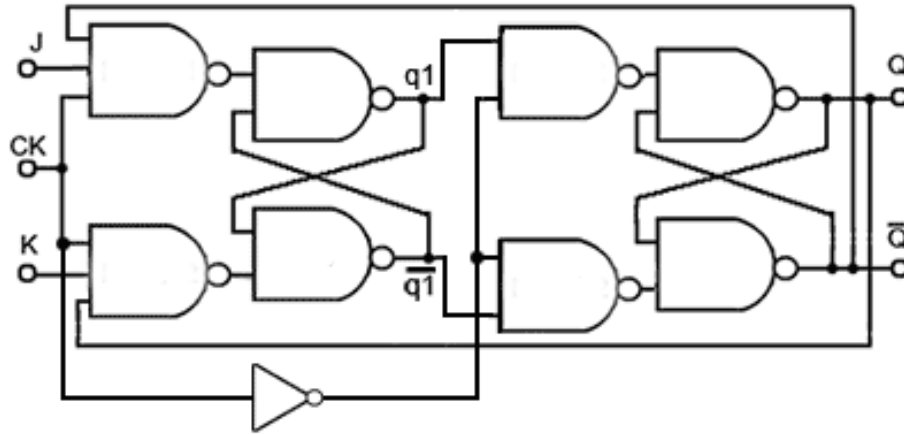


(c) Setting the latch

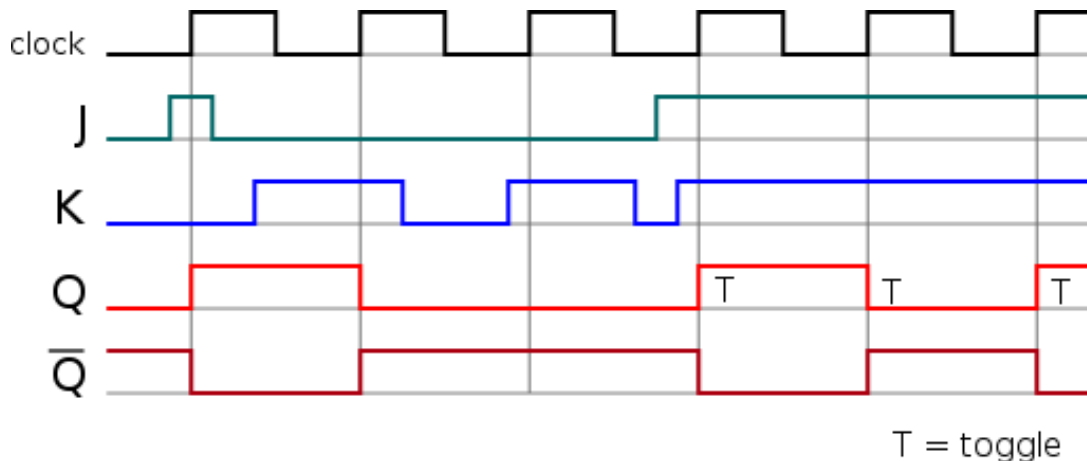


(d) Resetting the latch

1 Bit Memory: A clocked J-K Flip-flop



C	J	K	Q	\bar{Q}
0	0	0	latch	latch
0	0	1	0	1
0	1	0	1	0
0	1	1	toggle	toggle
x	0	0	latch	latch
x	0	1	latch	latch
x	1	0	latch	latch
x	1	1	latch	latch



A J-K flip-flop can be set/reset/toggled only during the *rising edge* of the clock signal

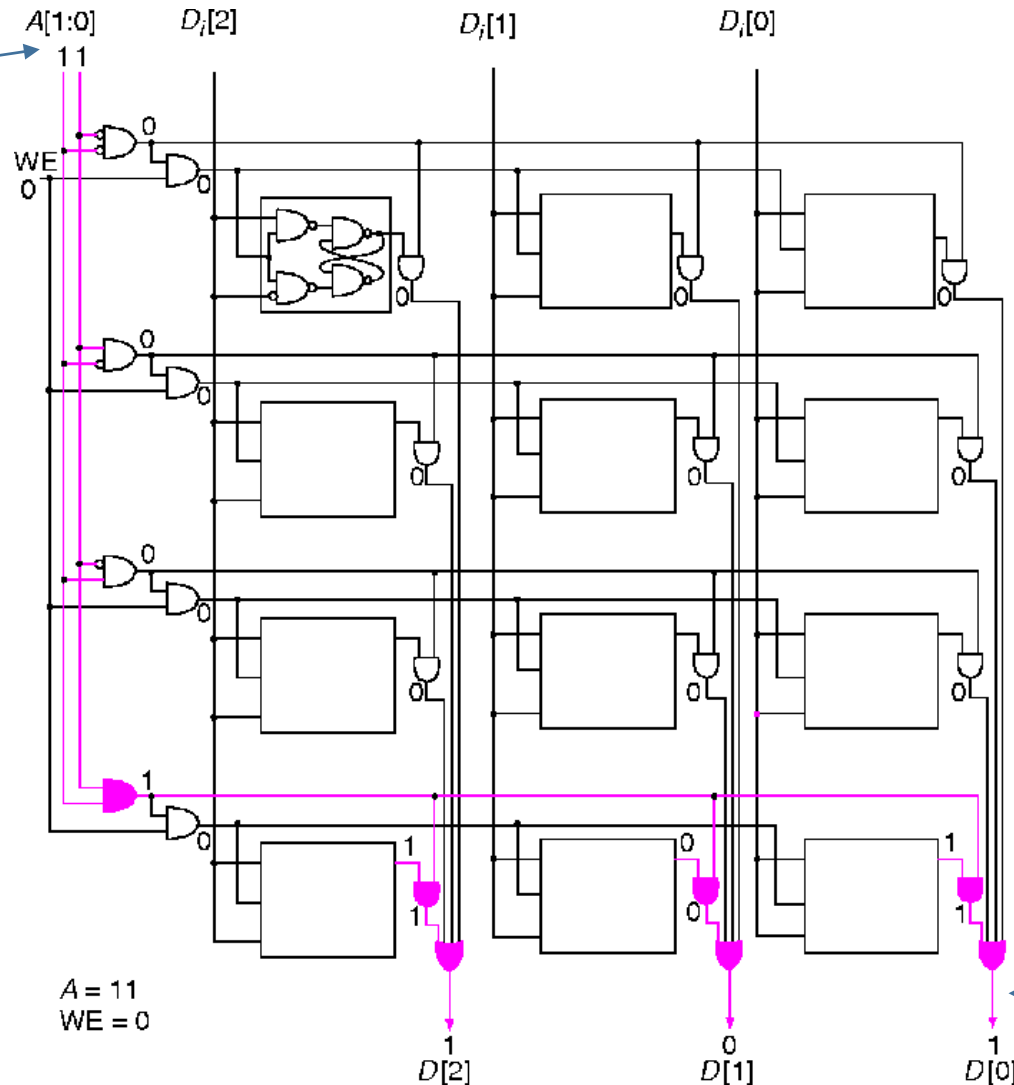
When the clock is low a **latch** maintains its prior output value

Reading **3** bits from a **4** address memory

Address
Lines

This **12-bit**
memory
is made from
a 4 x 3 matrix
of flip-flops

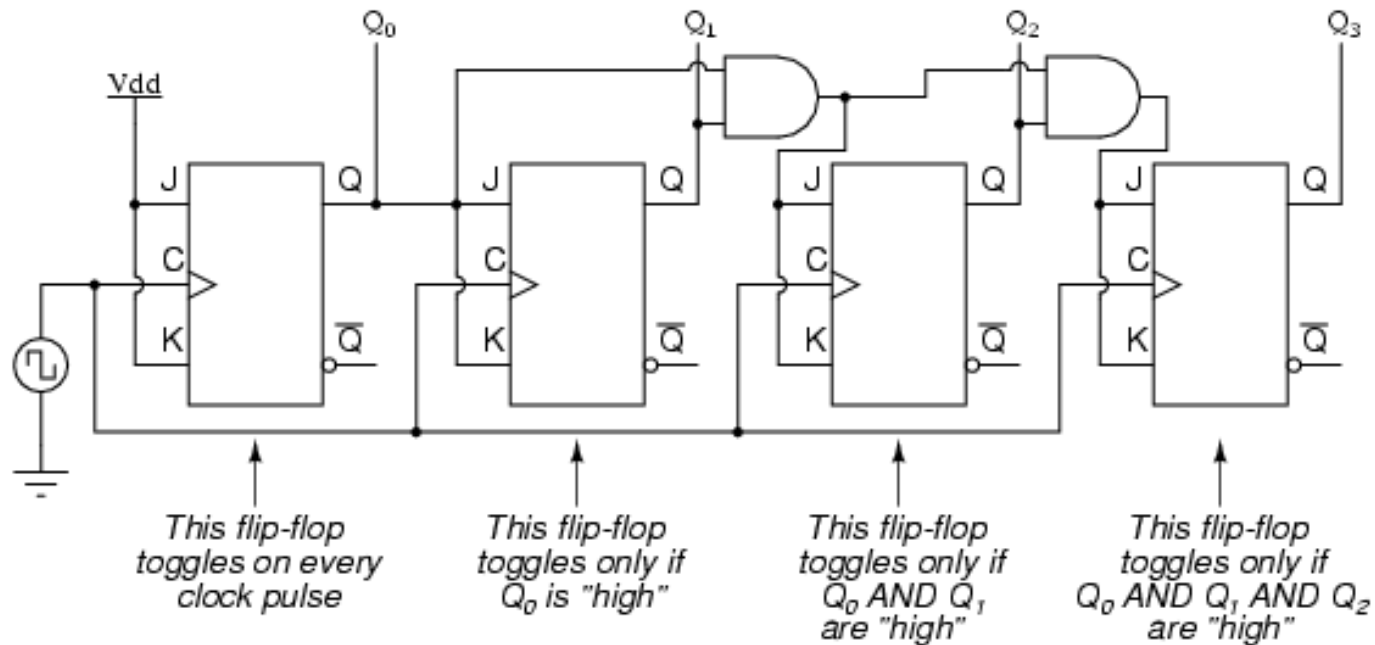
Imagine: a
typical 256 GB
smart phone has
 2×10^{12}
bits of memory!



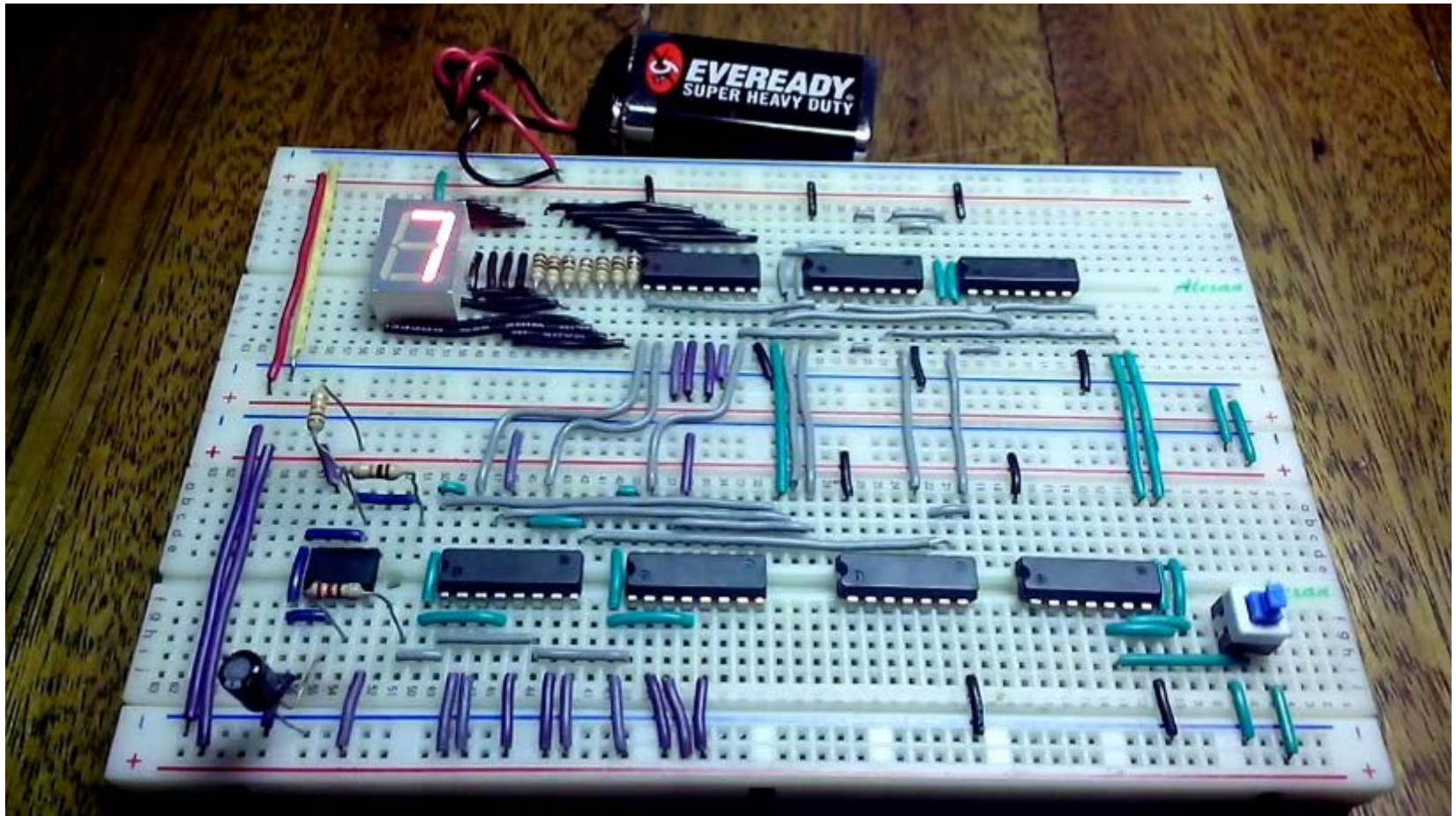
Data Lines

Counter = An **Adder** with **Memory**

A four-bit synchronous "up" counter



Counter = An Adder *with* **Memory**



Now You Know...

- Digital Logic Circuits
 - All computers are made from **chains** of simple logic gates
 - **NAND** and **NOR** gates are *universal* → they can make all other gates!
- How a computer performs arithmetic = **full adders**
 - Subtraction is just addition with **inverted** logic
 - Multiplication is just repeated addition
 - Division is just repeated subtraction
- How a computer stores numbers in memory = **flip-flops**
 - Four gates make a bit, eight bits make up a byte
 - Imagine how many gates are in your 32GB smartphone!