Slide Set 4

for ENEL 353 Fall 2019

Steve Norman, PhD, PEng

Electrical & Computer Engineering
Schulich School of Engineering
University of Calgary

Fall Term, 2019

Contents

Outline of Slide Set 4

Schematics

A *schematic* is a diagram showing elements of a digital circuit along with the wires connecting the elements to each other and to overall inputs and outputs of the circuit. Usually the vital power supply and ground connections to circuit elements are omitted to reduce clutter.

The image at right shows some rules for drawing wires in schematics.

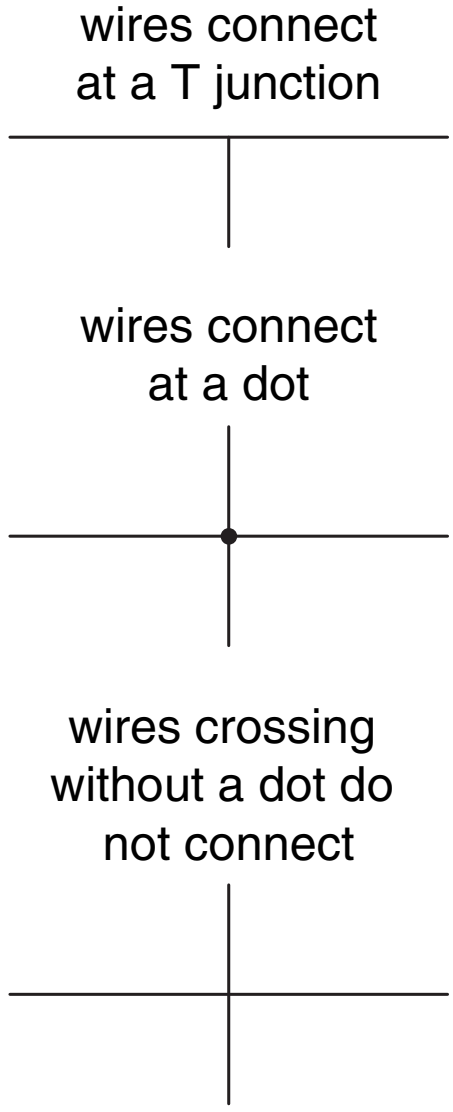


Image is Figure 2.24 from Harris D. M. and Harris S. L., *Digital Design and Computer Architecture, 2nd ed.*, © 2013, Elsevier, Inc.

What often-used graphical notation for crossing wires will NOT be used in ENEL 353?

Why not?

Schematics: Guidelines for directions of information flow through circuit elements

Please follow these guidelines:

- Left-to-right flow of information through circuit elements is best.
- Top-to-bottom flow through elements is second-best.
- Bottom-to-top flow through elements should be avoided if possible but sometimes helps to keep schematics clear and compact.
- Right-to-left flow through elements is worst, but, nevertheless, sometimes helps to keep schematics clear and compact.

What are examples of combinational circuits for which right-lo-left flow of information in schematics **helps** *with clarity?*

Example schematic, showing elements processing information left-to-right and top-to-bottom

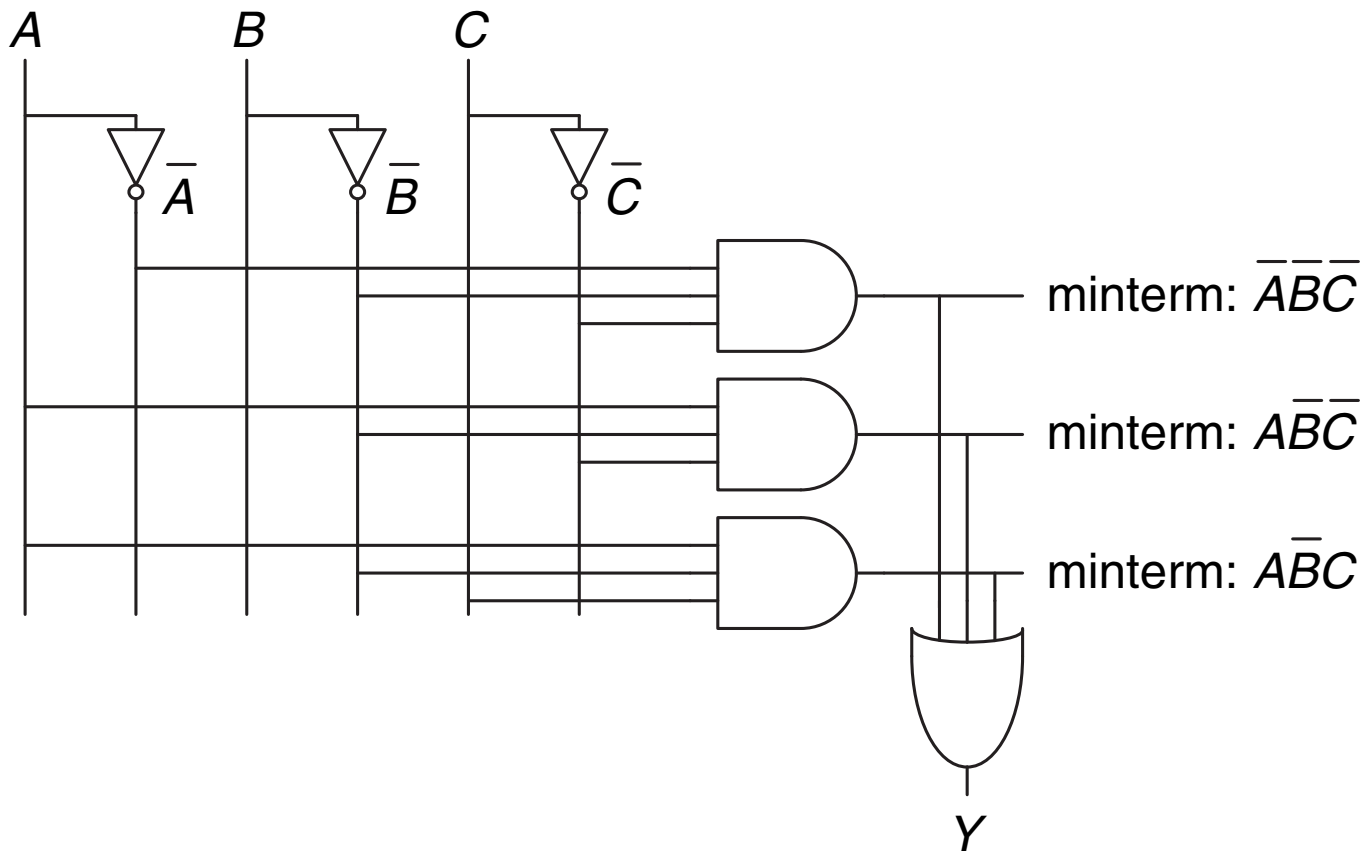


Image is Figure 2.23 from Harris D. M. and Harris S. L., *Digital Design and Computer Architecture, 2nd ed.*, © 2013, Elsevier, Inc.

Schematics: More examples

Expressions in SOP or POS form lead to easy-to-determine implementations of function with NOT, AND and OR gates.

For the C_{OUT} function of a 1-bit full adder, let's draw schematics for

- *the canonical SOP expression;*
- *the simpler SOP expression done for Set 3, Slide 48.*

Outline of Slide Set 4

Minimal SOP expressions

(This is a quick step back to textbook Section 2.3.5, to help with Problem Set 2.)

A *minimal sum-of-products* expression for a function, as you might guess, is in some sense a "simplest possible SOP expression" for that function.

Here is the precise definition:

- Among all possible SOP expressions for F, none have fewer products than a minimal SOP expression.
- Among all the possible SOP expressions for F that have the same number of products as a minimal SOP expression, none use fewer literals.

Note that a literal counts each time that it is used, so, for example, $A\bar{B} + AC$ has four literals, not three.

Minimal SOP expressions, continued

Here are three SOP expressions for a function:

$$F = A\bar{B}\bar{C} + A\bar{B}C + ABC \quad (1)$$
$$= A\bar{B} + ABC \quad (2)$$
$$= A\bar{B} + AC \quad (3)$$

Let's explain why (1) and (2) do not satisfy the definition of a minimal SOP expression for F.

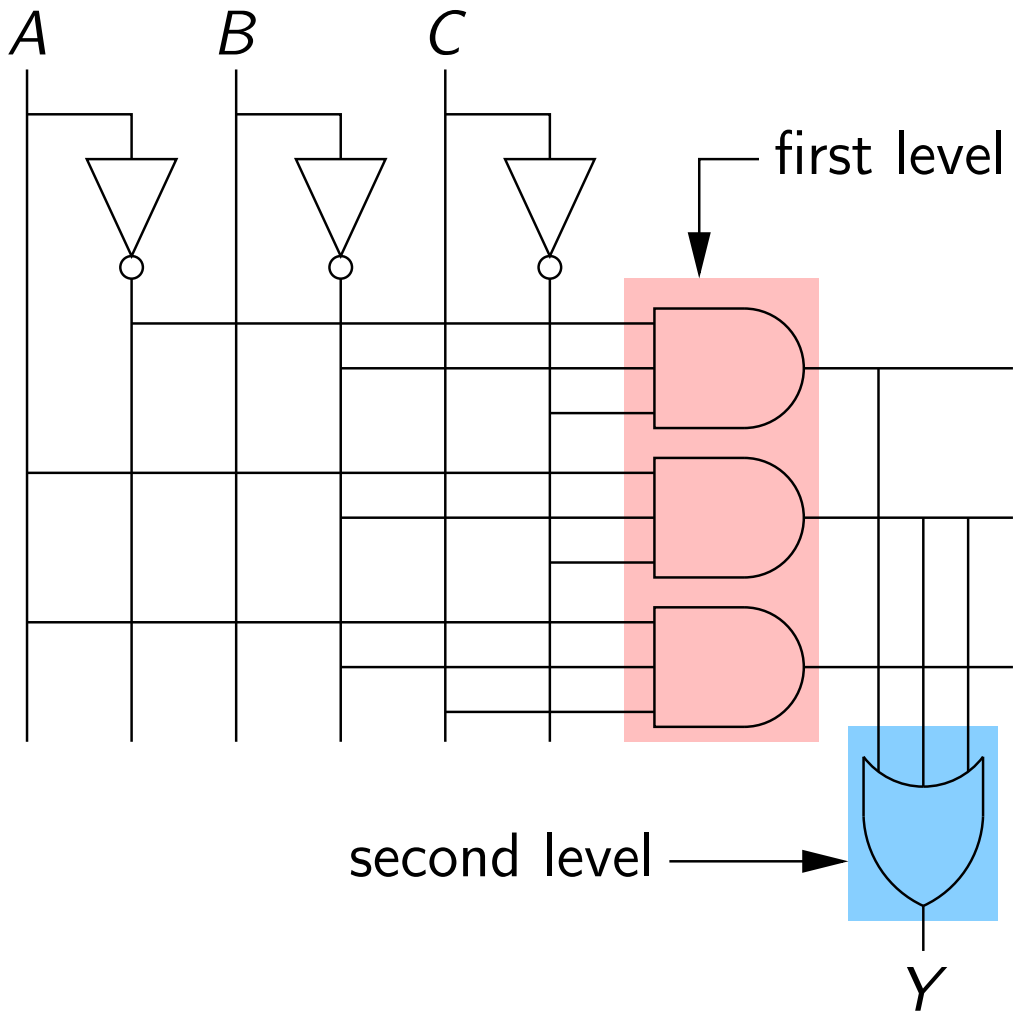
It turns out to be true that (3) **is** minimal, but we don't have a way to **prove** that yet. We'll return to that issue when we get to **Karnaugh maps**.

Outline of Slide Set 4

Two-level combinational logic

Circuits designed to implement SOP expressions using AND gates and OR gates are called *two-level logic*.

(NOT gates needed to provide complements of input variables don't count as levels in this definition.)



Multilevel combinational logic

Two-level SOP-based designs often work well, but if they do not, designs with three or more levels of gates may be better choices.

Textbook Section 2.5.1 give examples of cases where non-SOP designs work much better than two-level SOP-based designs.

Textbook Section 2.5.2 describes a technique called "bubble pushing" that can be very helpful in understanding multilevel designs that use NAND and NOR gates.

Hardware reduction via multilevel design

Textbook reference: Section 2.5.1.

Definition of N-input XOR:

$$\text{XOR}(A_1, A_2, \ldots, A_N) = \begin{cases} 1 & \text{if the number of 1 inputs is odd} \\ 0 & \text{if the number of 1 inputs is even} \end{cases}$$

Page 70 in the textbook shows that the 3-input XOR (which happens to be the sum function of a 1-bit full adder) requires four 3-input AND gates and a 4-input OR gate to implement the minimal SOP expression.

The textbook goes on to show that 3-input XOR can be implemented using only two 2-input XOR gates—a significant improvement over the SOP-based circuit.

Hardware reduction via multilevel design, continued

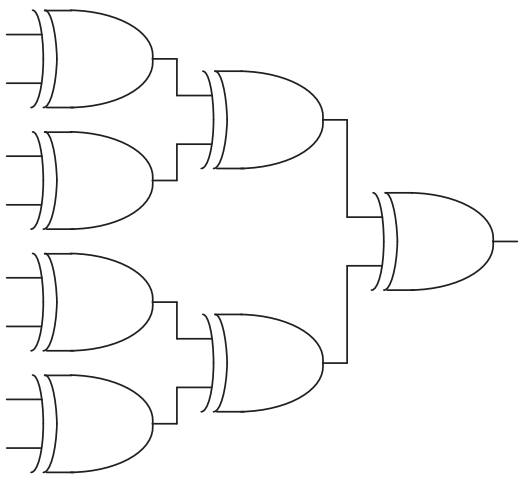
Textbook reference: Section 2.5.1.

Continuing to follow the textbook presentation, consider the problem of implementing an 8-input XOR function. Below are two choices. (Of course there are other choices as well.)

Two-level, SOP-based:

- 128 8-input AND gates!
- One 128-input OR gate!

Three-level "tree" of seven 2-input XOR gates:

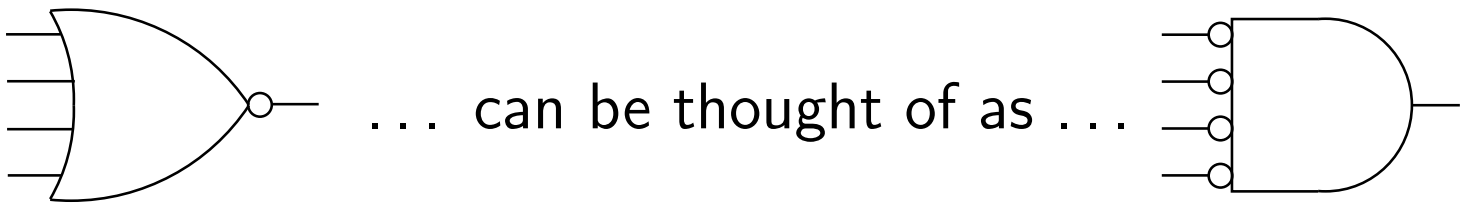


Outline of Slide Set 4

"Bubble-pushing" with NAND and NOR gates

Remember that De Morgan's Theorem says things such as $\overline{ABC} = \bar{A} + \bar{B} + \bar{C}$ and $\overline{A + B + C + D} = \bar{A}\bar{B}\bar{C}\bar{D}$.

That results in alternate symbols for NAND and NOR gates, for example:



You can think of moving bubbles through gates and interchanging AND with OR as *bubble pushing*. Bubble pushing does **not** change the **behaviour** of a gate.

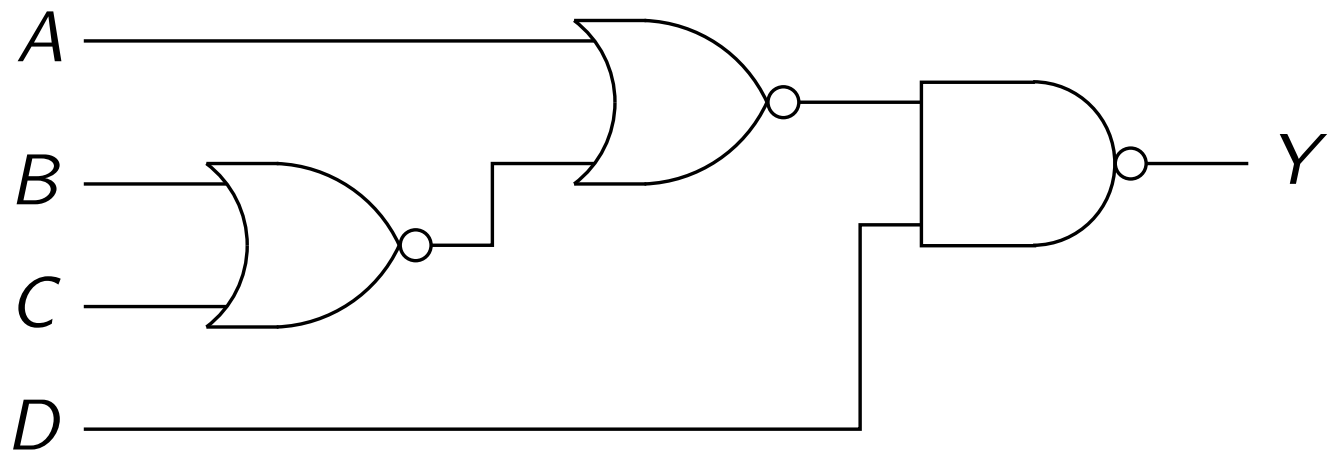
Bubble pushing for analysis of circuits with NAND and NOR gates

Textbook reference: Section 2.5.2.

This is a procedure to allow interpretation of logic in terms of ORs and ANDs rather than NANDs and NORs ...

- Start at the output and work towards the inputs.
- If the gate that drives the overall output is NAND or NOR, push its bubble to its inputs.
- For all the other gates, push bubbles as necessary so that each internal wire has either **no bubbles** or **cancelling bubbles at both ends**.

Bubble pushing for analysis of circuits with NAND and NOR gates—example



Let's use bubble pushing to express Y in terms of AND and OR operations, perhaps with NOT applied to some of the inputs.

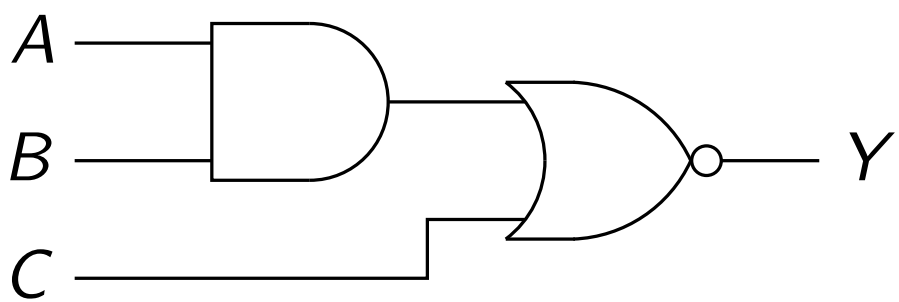
There is a similar example starting near the bottom of page 71 in the textbook.

Bubble pushing with AND and OR gates

If a schematic shows a mix of NAND, NOR, AND and/or OR gates, it may be useful to do bubble-pushing on an AND gate or an OR gate.

Let's show example transformations for 2-input AND and 3-input OR gates.

Let's use one of those transformations to get an expression for Y using only literals, AND, and OR . . .



Outline of Slide Set 4

Illegal (X) and Floating (Z) Values in Logic Circuits

Textbook reference: Section 2.6.

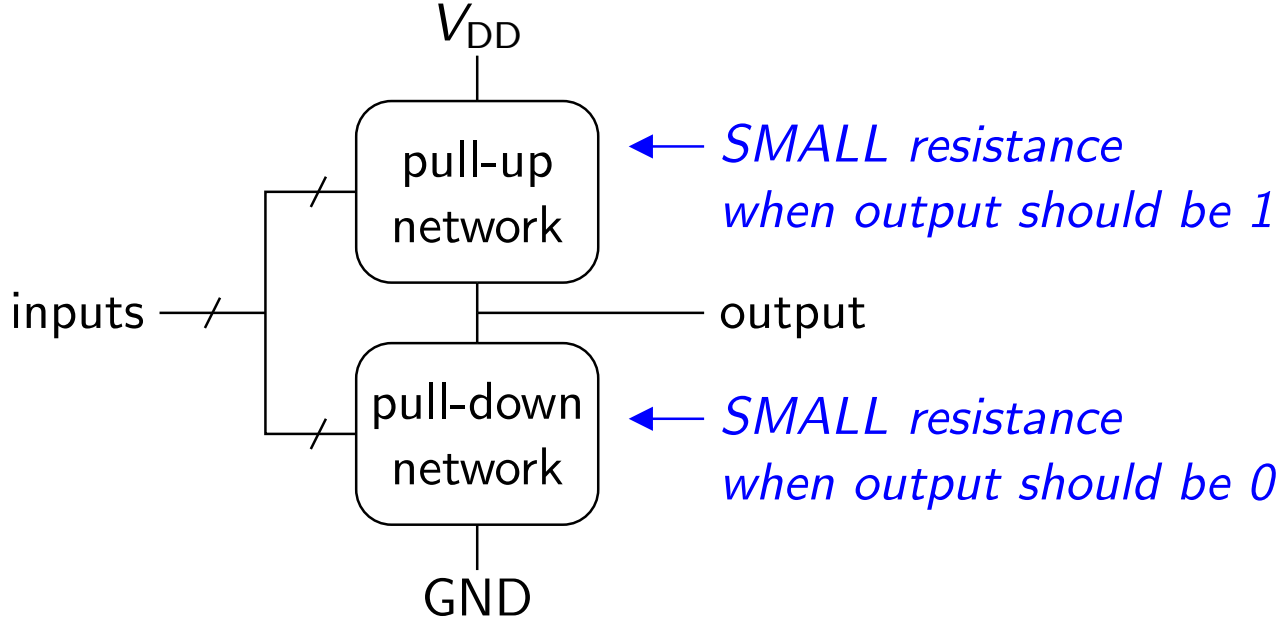
In pure Boolean algebra, a variable can have only one of two values: 0 or 1.

However, in modeling logic circuits, it's sometimes useful to have a more complex model. This model says that a node in a circuit can have one of four values: 0, 1, X or Z.

Before getting into the details of what X and Z values mean, it's helpful to understand a very approximate model of how logic gates work . . .

Basic structure of a CMOS logic gate

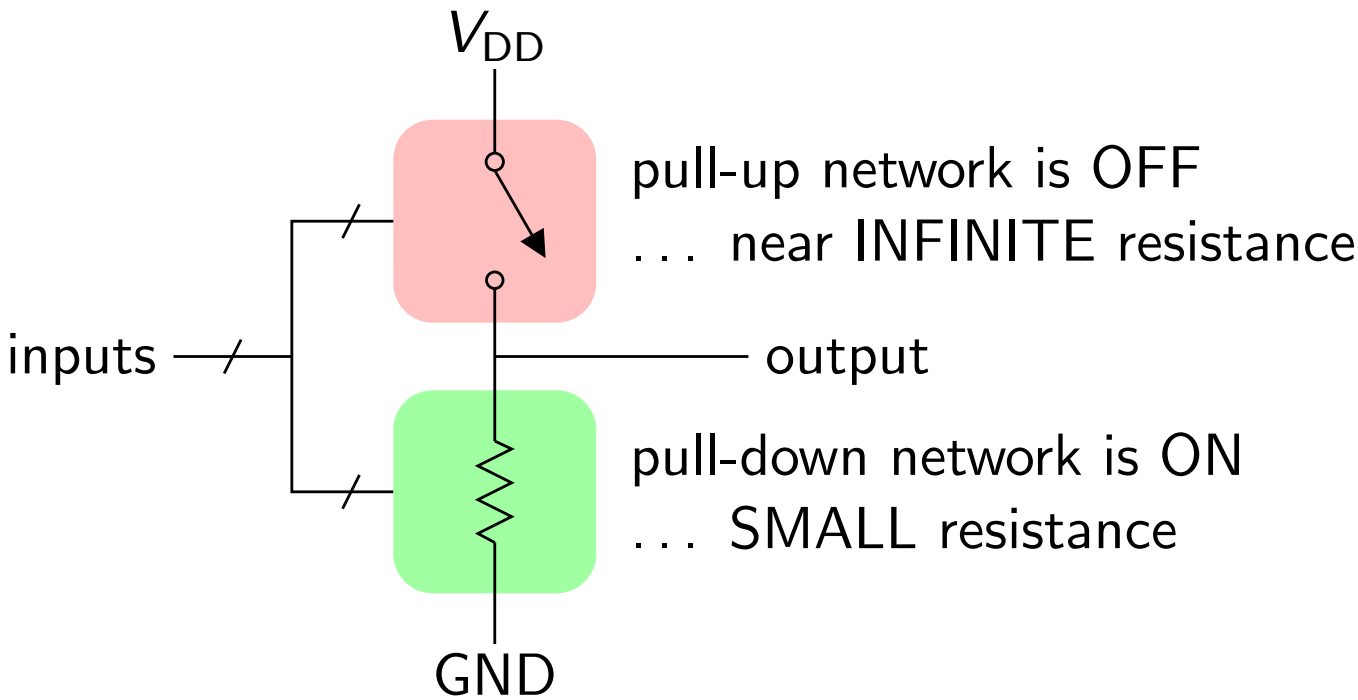
The pull-up and pull-down networks are collections of **MOS transistors**. (Details of MOS transistors are a major topic in courses later in the ENEL degree program.)



In normal operation one of the pull-up/pull-down networks is ON and the other one is OFF. **Which network is ON and which is OFF depends on the bit pattern on the input wires.**

Simple model for a CMOS gate with LOW output

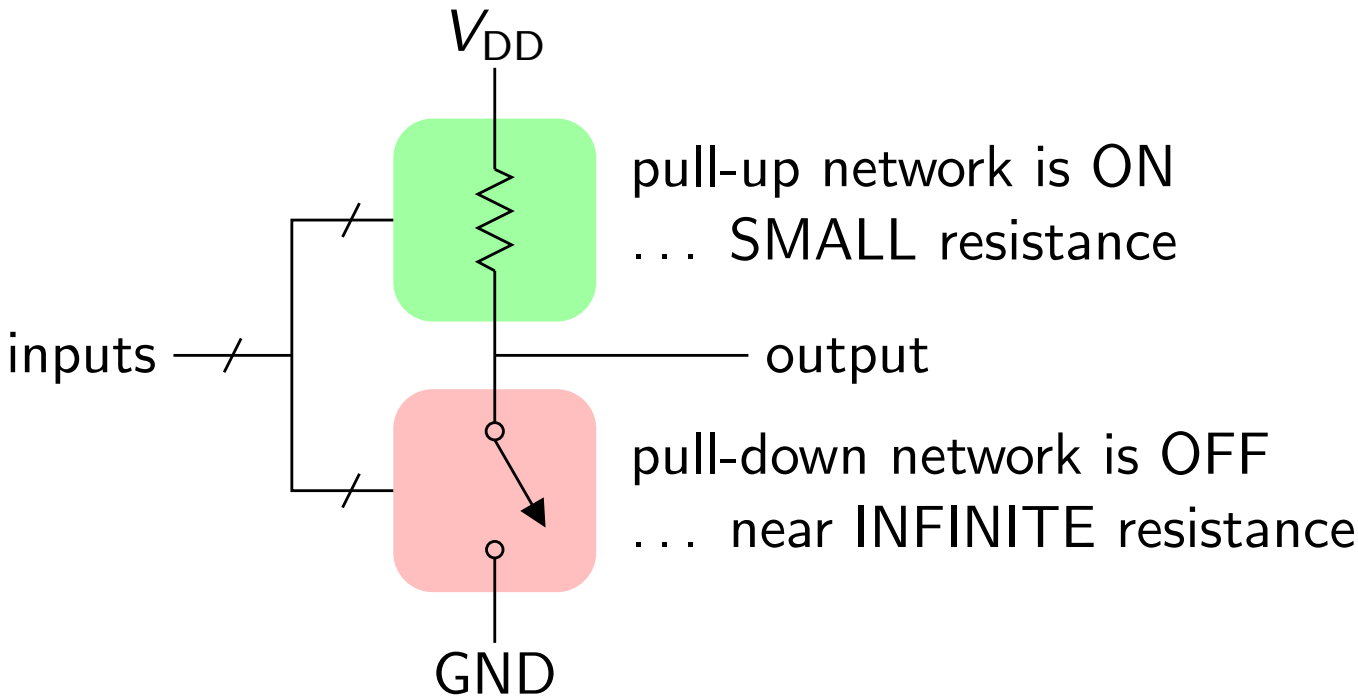
This should give you an idea why the voltage on the output is near 0 V ...



The actual behaviour of the transistors inside the pull-up and pull-down networks is more complicated than this model suggests!

Simple model for a CMOS gate with HIGH output

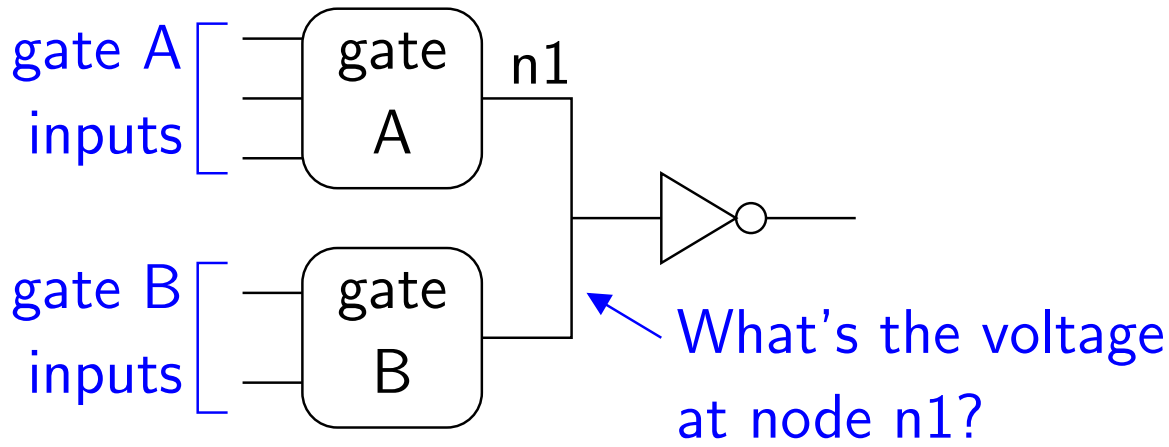
This should give you an idea why the voltage on the output is near V_{DD} ...



Again, real circuit behaviour is not quite this simple.

What happens if two or more gate outputs are wired together?

Usually, this is a **bad thing** to do. *Let's use our simple model for CMOS gates to understand what happens when Gate A is trying to output a 1 and Gate B is trying to output a 0 in this circuit . . .*



Note: For some **special** kinds of gates, it is actually useful to wire gate outputs to each other.

Illegal value: X at a node rather than 0 or 1

The symbol X is used to indicate that a node has an **illegal** or **unknown** value.

Complex digital designs written in hardware description languages like VHDL or SystemVerilog can be tested for correctness using **logic simulator software**.

A logic simulator would use X to represent the gate A/gate B output for the situation in the previous slide.

An X value for a node in a simulation result could also indicate one of **many other kinds of defect** in the circuit being simulated.

Warning:
There's another meaning for X in digital logic

We'll soon see that X is also used to indicate a **don't care** value in some truth tables and Karnaugh maps. X for don't care **does not mean the same thing** as X for illegal/unknown value.

It's unfortunate that X has two different meanings.

X for illegal/unknown value is used as a possible state of a node in a specific **circuit design**.

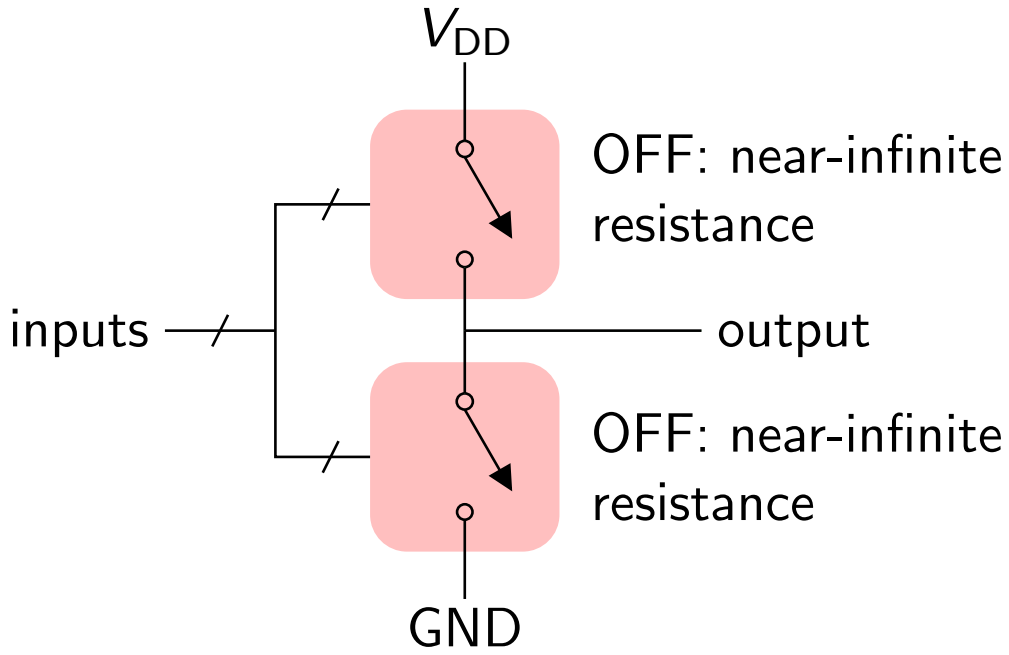
We'll soon discover that X for don't-care shows up in some **truth tables** and **Karnaugh maps** corresponding to those truth tables.

Outline of Slide Set 4

What if pull-up and pull-down networks are BOTH OFF?

Most logic gates are designed to switch between one of two states:

- pull-up OFF, pull-down ON
- pull-up ON, pull-down OFF

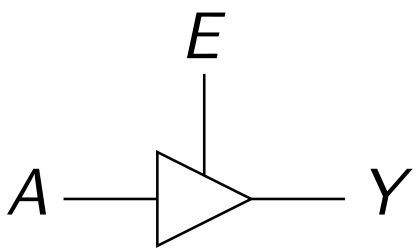


However, some **special** kinds of gates can go into the state shown in the diagram, in response to certain input combinations.

What can we say about the output node in this situation?

Tristate buffers

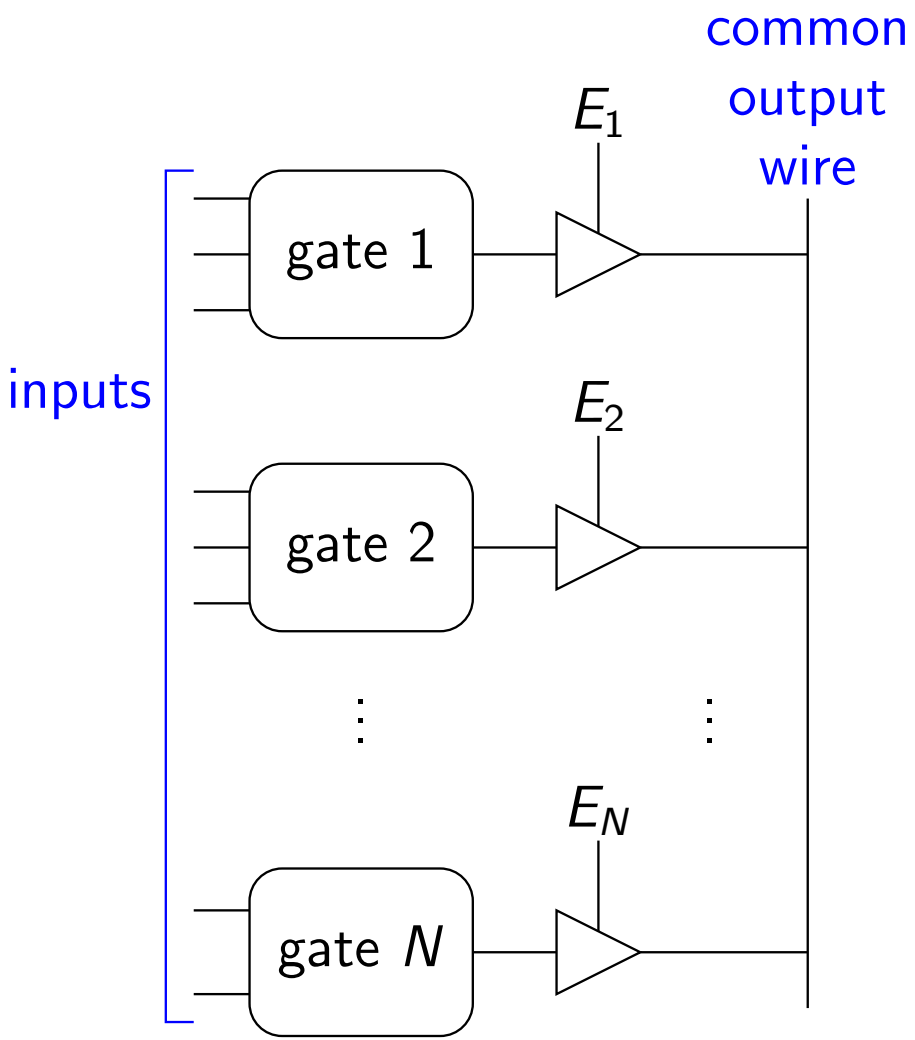
A *tristate buffer* has two inputs, often called A and E — E is for **enable**. (Sometimes the **enable** signal gets the two-letter symbol EN.) Here are the logic symbol and a table showing what the pull-up and pull-down networks do for all possible input combinations:



E	A	pull-up	pull-down
0	0	OFF	OFF
0	1	OFF	OFF
1	0	OFF	ON
1	1	ON	OFF

Let's use the table to describe in words how a tristate buffer behaves.

An application of tristate buffers



The goal of this design is that sometimes gate 1 will control the common output wire, sometimes gate 2 will be in control, and so on.

To meet this goal, what has to be true about the logic driving the signals E_1, E_2, . . . , E_N?