# Physics 219 – Fall, 2005
# LabNotes 11 – Combinational Logic

## Table of Contents

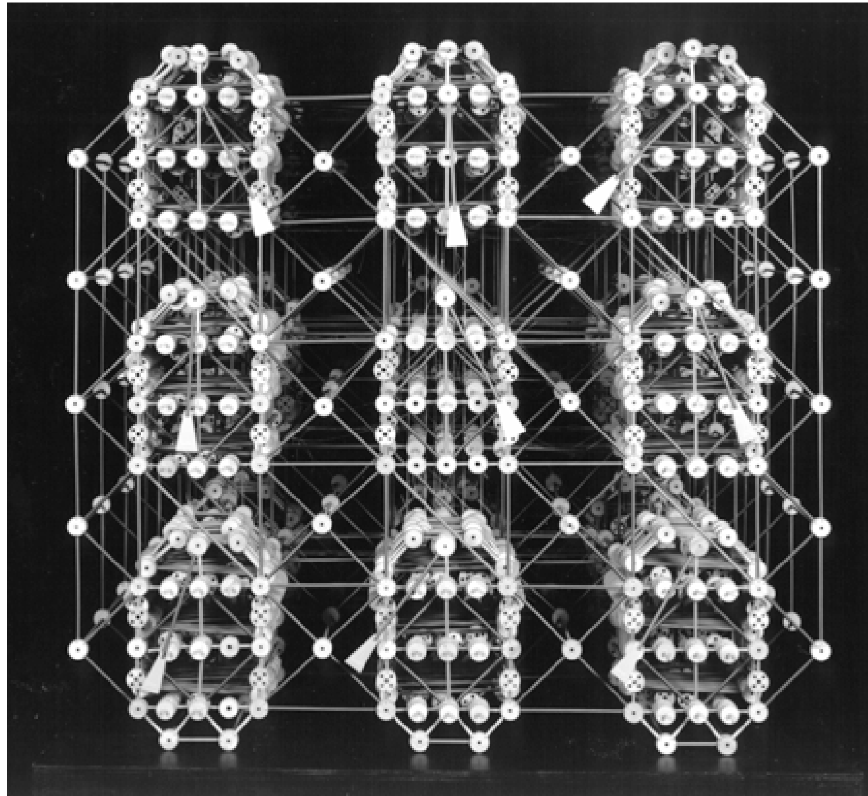

**Friday, December 2 .......... 2**
Digital Building Blocks .......... 2
Combinational Logic .......... 3
An example of combinational logic: A car's warning buzzer .......... 3
Digital Logic Gates .......... 5
AND gate .......... 5
OR gate .......... 5
INVERTER gate .......... 5
NAND gate (NOT AND) .......... 6
NOR gate (NOT OR) .......... 6
XOR gate (EXCLUSIVE OR) .......... 7
Car warning buzzer implementation using logic gates .......... 7
DeMorgan's Theorem .......... 8
*Student Manual* 13-2 .......... 8
Active High vs. Active Low Logic .......... 8
Building Logic Gates Out of Transistors .......... 9
MOSFET Inverters .......... 9
CMOS Inverter .......... 11
CMOS NAND Gate .......... 12
CMOS Three-State .......... 13
How do you get from logic gates to higher level computation? .......... 14
Adder .......... 14
Arithmetic Logic Unit (ALU) .......... 18

# Friday, December 2

## Digital Building Blocks

We started out in this course by introducing you to the LogoChip, a powerful and relatively easy to use general purpose digital device. While you quickly had a chance to build a wide range of digital circuits, we pretty much treated the LogoChip as a “black box” whose inner workings were kept hidden from view. Now it’s time to look inside the black box.

In the next few labs you will have an opportunity to see the basic building blocks of digital electronics and gain a sense how, remarkably, one can take a collection of transistor switches which, when viewed individually seem quite unremarkable, and use them to create digital circuits of awesome power and flexibility.



Our present technological state is such that we rely mainly on transistors built out of silicon (mostly MOSFETs) to create our digital circuits. But there is no fundamental reason we cannot use other materials as building blocks. For example, while they were undergraduates at MIT in the 1970s, Brian Silverman (of LogoChip fame) and his roommate Danny Hillis used Tinker Toys and fishing line to build a computer that could play tic-tac-toe. Their invention, shown in the figure

above, can be viewed in person in the lobby of the Museum of Science in Boston. Someday silicon-based transistors may seem as quaint vacuum tubes seem today, but the more basic ideas of digital systems seem certain to live on.

## Combinational Logic

The basic scheme in most digital electronics is captured in the drawing below. We can view most devices as entities that "map" a set of "inputs" onto a set of "outputs", according to a set of rules.

There are two basic ways that the inputs can be mapped onto the outputs:

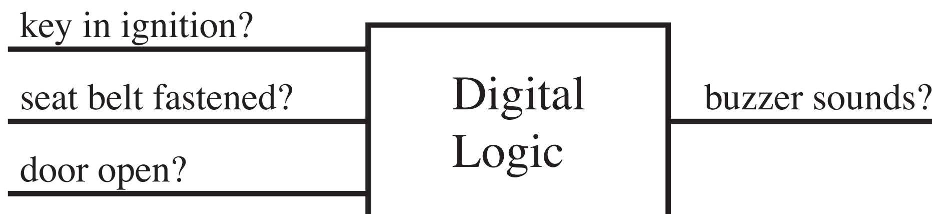
**Combinational Logic** - In this case the output values at a given moment depend only on the values of the inputs at that moment.

**Sequential Logic** - In this case the output values at a given moment depend not only on the values of the inputs at that moment, but also on the past history of the device. Thus a sort of "memory" is involved.

We will begin by studying combinational logic in this LabNotes and then move on to sequential logic in the next LabNotes.

### An example of combinational logic: A car's warning buzzer

Consider the case of the annoying warning system in place in most cars. There are apparently a set of sensors throughout the car that provide inputs to some digital device whose job it is to decide whether or not to buzz.




At a given instant, each one of the input and output lines above can be thought of as being either TRUE or FALSE. Let's adopt (for now) a convention in which we

associate the "0" state with FALSE condition and the "1" state with the TRUE condition. (This convention is known as positive true logic. Often one ends up using opposite negative true logic convention, in which we associate the "0" state with the TRUE condition and the "1" state with the FALSE condition.)

Let's construct a **truth table** for the car warning buzzer example., a table consisting of all possible permutations of inputs along with the desired outputs:

| keys in ignition? | seat belt unfastened? | door open? | buzzer sounds? |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

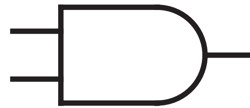
How can we implement this truth table (or any other) electronically?

## Digital Logic Gates

**Logic gates** are the building blocks used to implement truth tables. Each logic gate can perform a simple logical function. By interconnecting appropriate logic gates, it is possible to build a circuit that implements *any* desired truth table.

The truth tables and schematic symbols for the fundamental logic gates are:

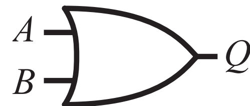
**AND gate**



**Truth Table:**

| *A* | *B* | *Q* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

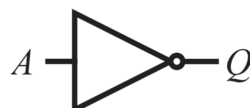
**OR gate**




Truth Table:

| *A* | *B* | *Q* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**INVERTER gate**

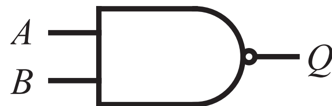


**Truth Table:**

| *A* | *Q* |
|---|---|
| 0 | 1 |
| 1 | 0 |

**NAND gate (NOT AND)**

*"Bubbles" in a the schematic symbol denote inversion:*




Truth Table:

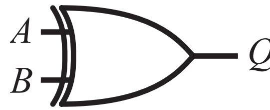
| *A* | *B* | *Q* |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**NOR gate (NOT OR)**

Truth Table:

| *A* | *B* | *Q* |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**XOR gate (EXCLUSIVE OR)**




Truth Table:

| *A* | *B* | *Q* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

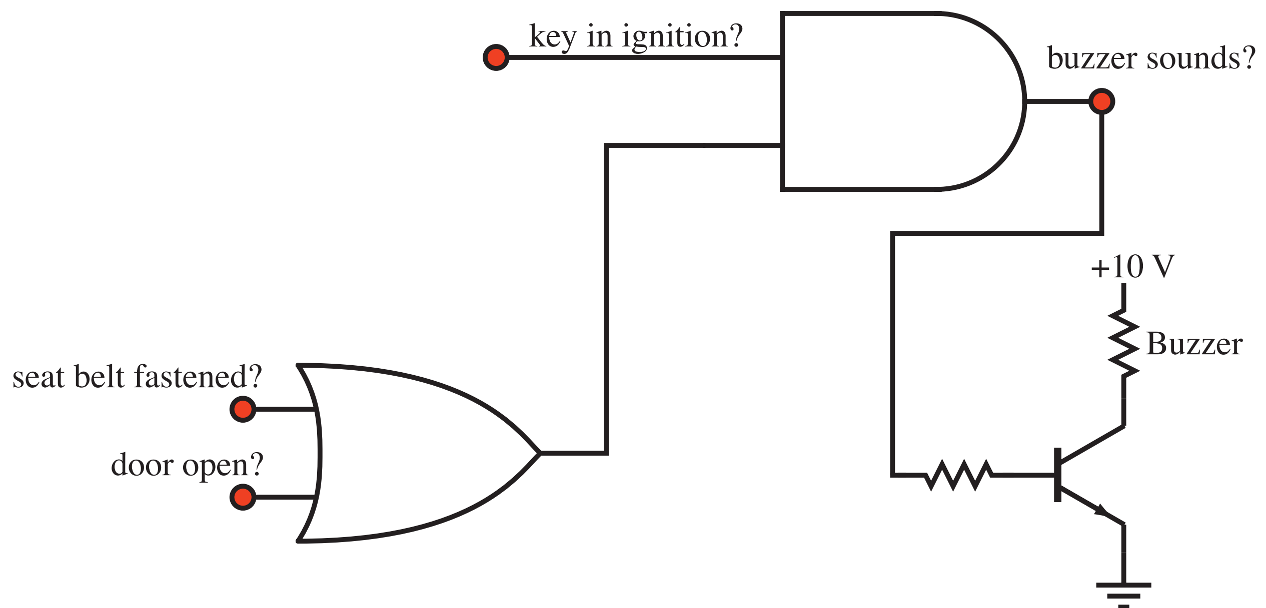
(Note that the output of an XOR gate is 0 if the input bits are the same and it is 1 if the input bits are different. Thus the XOR function provides a convenient test to see if two numbers are the same or not. This is a useful trick to remember down the road when doing assembly language programming, since the XOR function is part of the basic "instruction set" for most microprocessors.)

**Car warning buzzer implementation using logic gates**

It's easy if you first carefully state what you want using words:

*The buzzer should sound if (the key is in the ignition) AND ((the seat belt is unfastened) OR (the door is open)).*

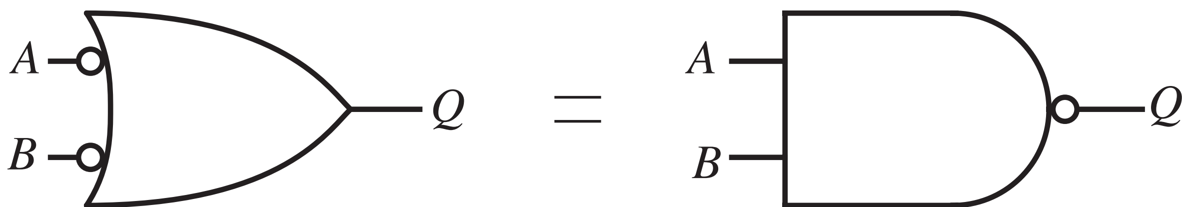
Therefore, the logic we want is:



### DeMorgan's Theorem

With the help of some inverters and an observation made a long time ago by DeMorgan it is possible to build an OR function using AND gates and vice versa. DeMorgan noticed that the following gates have the same truth tables (remember, the bubbles are really shorthand for inverters):

| A | B | Q |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




In words: "You can switch an AND gate with an OR gate if you also invert all inputs and outputs."

#### *Student Manual* 13-2

- It can be shown that the NAND gate is a **universal gate**; *any* combinational logic circuit can be built using NAND gates as building blocks. As a modest step towards convincing you this is true, and so that you can say you've done something with individual logic gates at least once in your life, try completing parts (a) and (b) of section 13-2 of the *Student Manual*, in which you are asked to use NAND gates to construct an AND gate and an OR gate.

### Active High vs. Active Low Logic

There are two possible conventions that one can adopt for identifying the conditions TRUE and FALSE with our logic states HIGH and LOW:

In **positive true logic** (also called **active high logic**) the HIGH state is identified with a condition being TRUE and the LOW state is identified with the condition being FALSE.

In **negative true logic** (also called **active low logic**) the LOW state is identified

with a condition being TRUE and the HIGH state is identified with the condition being FALSE.

Although it may seem less intuitive, active low logic is often used, for reasons that are mainly "historical". With the older TTL logic it often made sense to have "control lines" be active low. For example, recall that the TIL-311hexadecimal display chips, which use TTL logic, contain a digital input line called $\overline{\text{ENABLE}}$ that in normal use goes LOW only occasionally in order to “latch” the data.

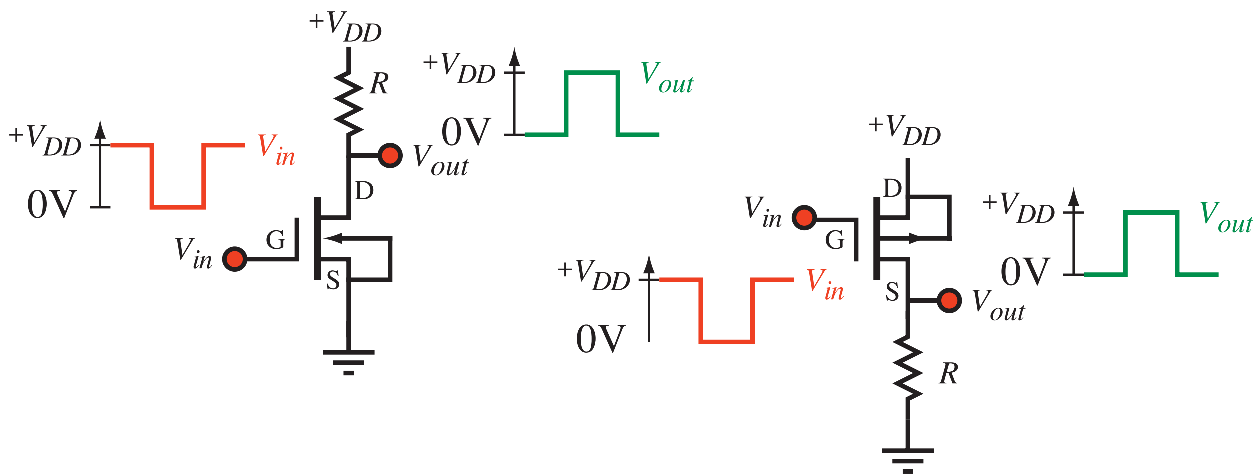
In this case it is desirable to have the most common state be the HIGH one, because the HIGH TTL state is less vulnerable to noise and because it requires essentially no power to drive a TTL input HIGH while to drive it LOW requires a fair amount of power. Thus when TTL reigned supreme, it became standard to employ active low logic, which minimized the amount of time control lines spent in the LOW state. With the now prevalent CMOS logic this asymmetry does not exist, but old habits are hard to break and active low logic remains a frequently used standard at this time.

## Building Logic Gates Out of Transistors

### MOSFET Inverters

### Passive Pull-Up

It is easy to build an inverter using a single MOSFET employing a pull-up resistor:

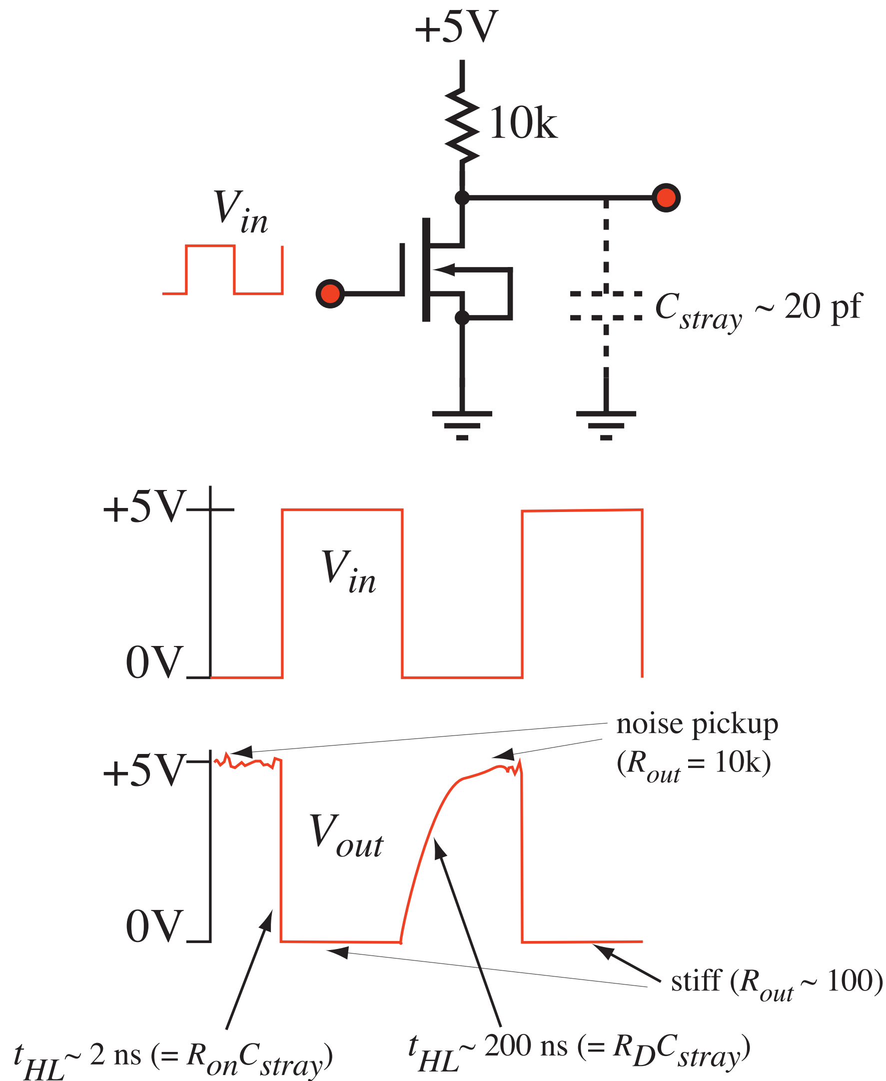



NMOS logic inverter PMOS logic inverter

This circuit is very closely analogous to the transistor switch that you built

previously. This approach is referred to as "passive" because in one of the two output states the output is not “stiffly tied down”. Although simple, this passive pull-up scheme has two major disadvantages, as indicated in the figure below:

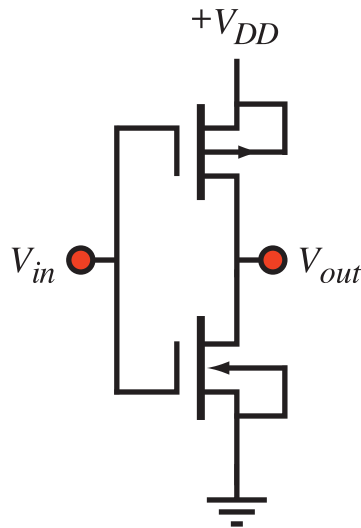
> 1) It is relatively slow due to the *RC* time constant associated with having to charge or discharge the inevitable stray capacitance through the pull-up resistor.
>
> 2) It is susceptible to noise pick-up when the output is not stiffly tied down.




Problems with Passive Pull-up

### CMOS Inverter

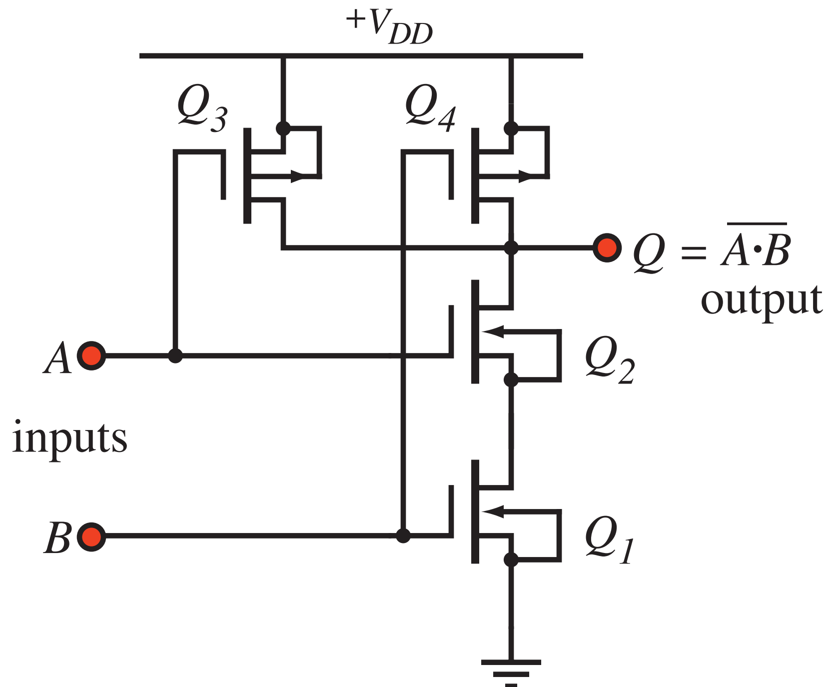
The problems with passive pull–up vanish with the CMOS logic configuration which acts as an inverter. (This is the same circuit you have previously used when constructing an H–bridge.). With CMOS the output is always connected to one side of the power supply through an "on", low resistance transistor, so the *RC* time delays associated with having to charge or discharge the inevitable stray capacitance through a pull-up resistor are not present.




CMOS logic inverter

### CMOS NAND Gate

Since the NAND gate is a universal gate it is a worthwhile exercise to see how a NAND gate can be implemented using the CMOS scheme. It's easy to understand how the CMOS NAND gate shown below works:



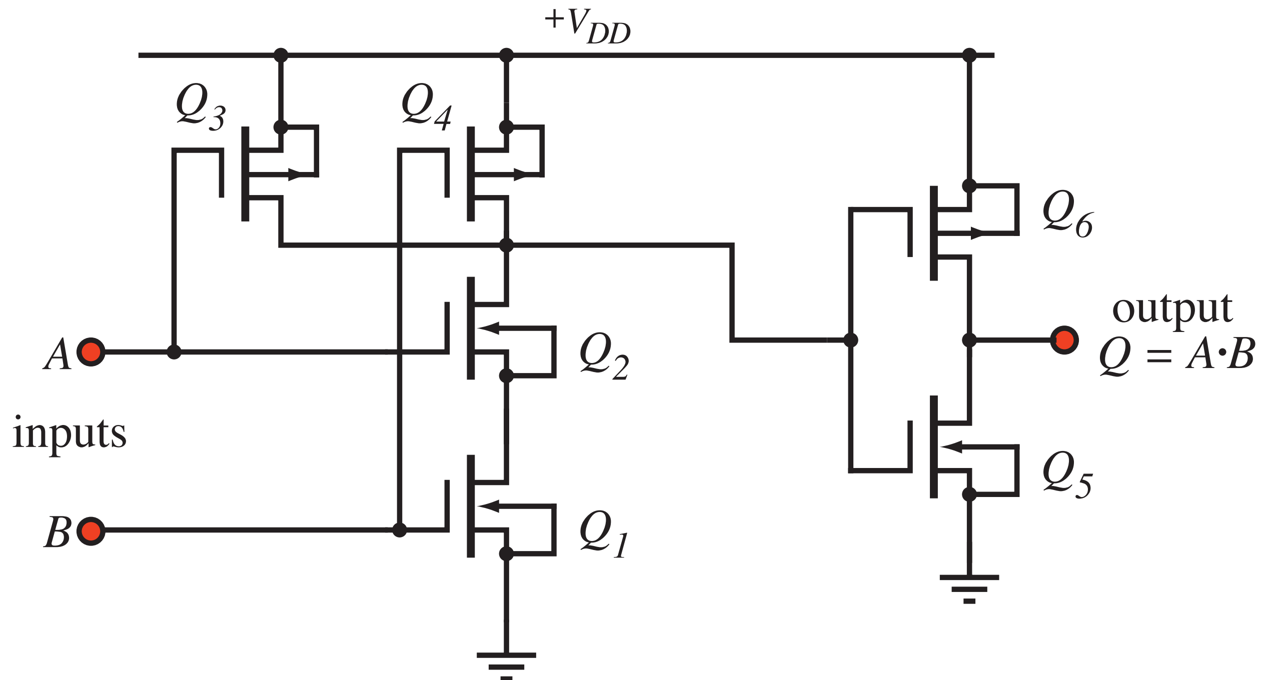

CMOS NAND Gate

If the inputs *A* **and** *B* are *both* HIGH (*i.e.* at +6 V) then the *series* NMOS switches $Q_1$ and $Q_2$ are both “on” (*i.e.* conducting) and the output *Q* is tied *stiffly* to ground. The parallel PMOS switches $Q_3$ and $Q_4$ co-operate nicely in this case by being “off” (*i.e.* non-conducting) so that *Q* is isolated from the +6 V supply. Note that there is essentially no "quiescent" current in this state.

If *either* of the inputs *A* **or** *B* are LOW (or if both *A* and *B* are LOW) then either $Q_1$ or $Q_2$ (or both $Q_1$ and $Q_2$ ) are “off” and *Q* is not pulled to ground. In this case either $Q_3$ or $Q_4$ (or both $Q_3$ and $Q_4$ ) are "on" so that Q is *actively* pulled HIGH. Again, note that there is no "quiescent" current in this state. Thus we begin to see why CMOS-based logic consumes very little power. (Note however the importance of making certain that the inputs are tied either to HIGH or LOW, even if a particular gate is not being used. If the inputs are allowed to "float" to some intermediate voltage then the both the *n*-channel and *p*-channel MOSFETs can end up in a "partially conducting" state and quiescent power consumption can soar. See

the related laboratory exercise in section 13-1 of the *Student Manual.*)

To make an AND gate we just need to add a CMOS inverter, which we already know about, to the NAND gate.

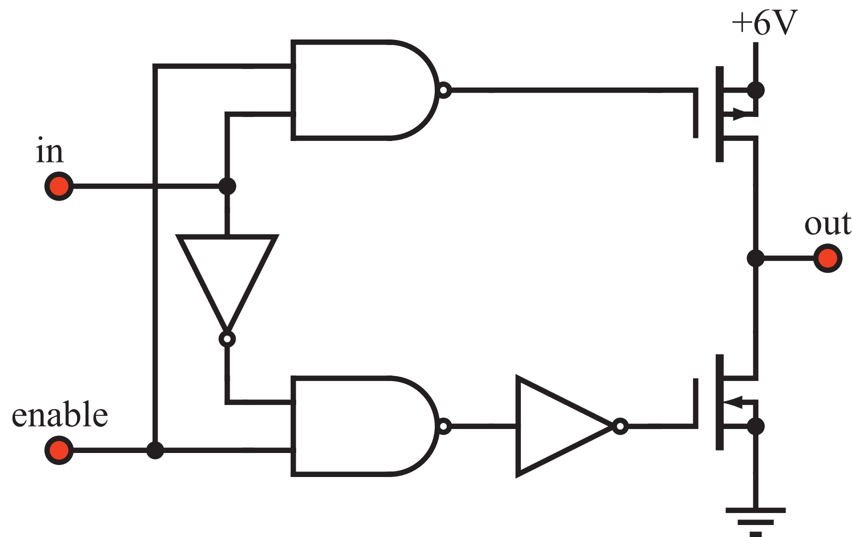



CMOS AND Gate (NAND + INVERTER)

### CMOS Three-State

As the name implies, the **three-state** output stage can go into a third state besides HIGH and LOW: OFF. This ability is of critical importance in computers: it allows multiple **drivers** to share a single driven wire or **bus** line. (If multiple outputs are connected to a common bus line it is imperative that we avoid the situation where two outputs are simultaneously trying to assert different levels onto the line. We will arrange this by always using three-state outputs and making sure that all of the three-states are in the OFF state except for one which is actively asserting a level onto the bus line.) In this section you are asked to build a three-state buffer, a gate which, when enabled, does nothing but give the signal a fresh start. When not enabled the output of the gate is OFF. Again thinking back to the LogoChip example, we see how important it is to have the ability to turn the outputs OFF. This is necessary if a single pin is to be able to serve as either an input or an output.

As the *Student Manual* says: "The trick is to add some logic that can turn off both the pull-up and pull-down transistors. When that happens the output is disconnected from both + 6 V and ground; the output is then OFF or 'floating'". When the gate is enabled the output level simply follows the input level. An implementation of a three-state CMOS buffer is shown in the *Student Manual* in section 13-6. Can you explain how the circuit below works?




**How do you get from logic gates to higher level computation?**

Above we saw how you could use transistors to construct simple logic gates. The next step is to see how these logic gates can be combined to perform more sophisticated computation.

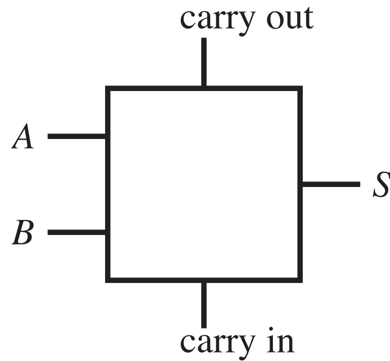
**Adder**

Let's consider one relatively simple, yet instructive, example: an "adder" circuit that can add together two binary numbers. We can get started by thinking about how you do addition in base two. For example:

$$\begin{array}{r} 1\,0\,1\,0 \\ +\,0\,1\,1\,1 \\ \hline \end{array} \quad \longrightarrow \quad \begin{array}{r} 1\,1\phantom{\,0\,0} \\ 1\,0\,1\,0 \\ +\,0\,1\,1\,1 \\ \hline 1\,0\,0\,0\,1 \end{array}$$

Our goal is to be able to implement this adder with simple logic gates. Towards

this end, the first challenge is to build a “one-bit adder” of the form



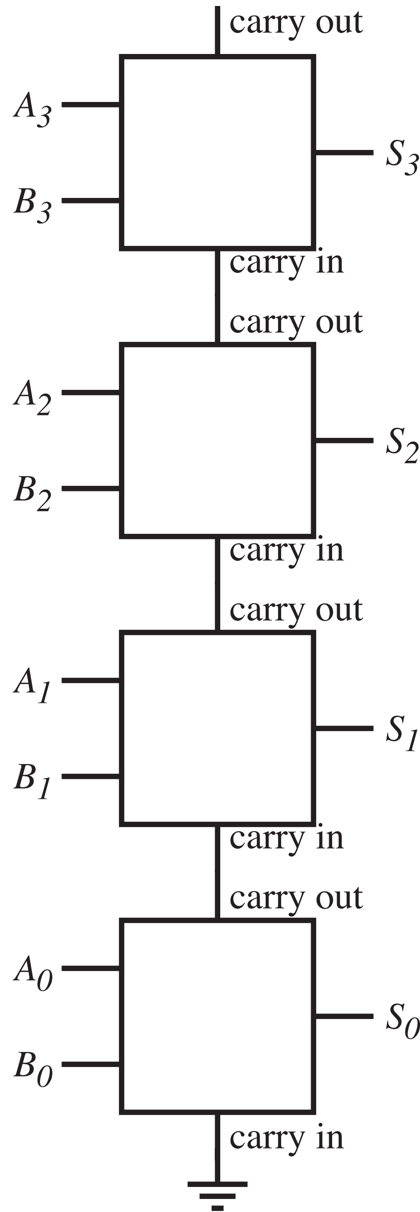

This adder takes as inputs three one bit numbers, *A* and *B* , along with a “carry in” bit and adds them together to produce a sum between 0 and 3. The sum is expressed as a two bit binary number. The least significant bit of the result is produced on the line labeled *S*, while the most significant bit of the result is produced on the line labeled “carry out”.

The desired truth table of the one bit adder is thus:

| **INPUTS:** | **Carry In** | **A** | **B** | **OUTPUTS :** | **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 |

The one bit adders can be readily “cascaded” together by tying the carry out of one adder to the carry in of another adder to produce a multi–bit adder. For example

the scheme shown in the figure below will add two arbitrary 4 bit numbers together to produce a 5 bit result (the most significant bit of the sum being contained on the carry out line of the last stage.)

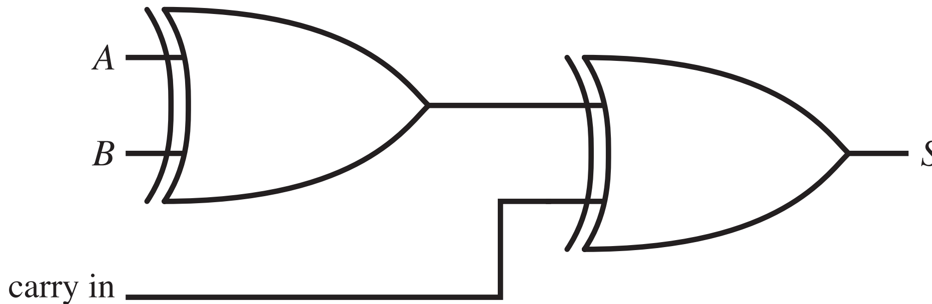



**4-bit adder**

The main challenge then in implementing the above plan is to figure out how to implement the one bit adder using simple logic gates.

Let us first figure out the logic required to produce each of the two output bits, $S$ and $C_{out}$.

**1) Least Significant bit $S$.** Note that the sum $S = A \text{ XOR } B$ when $C_{in}$ is LOW. When $C_{in}$ is HIGH $S = \overline{A \text{ XOR } B}$. Thus we can implement this part of the truth table using two XOR gates:



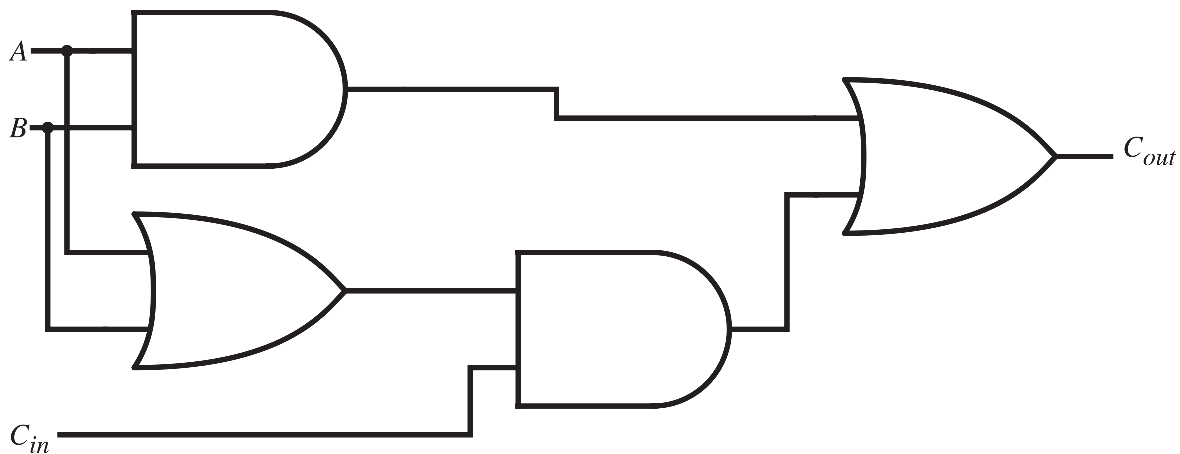

**2) Most Significant bit $C_{out}$.** Note from the above truth table that $C_{out}$ is HIGH when either

(*A* and *B* are both HIGH) OR (($C_{in}$ is HIGH) AND (*A* OR *B* is HIGH))

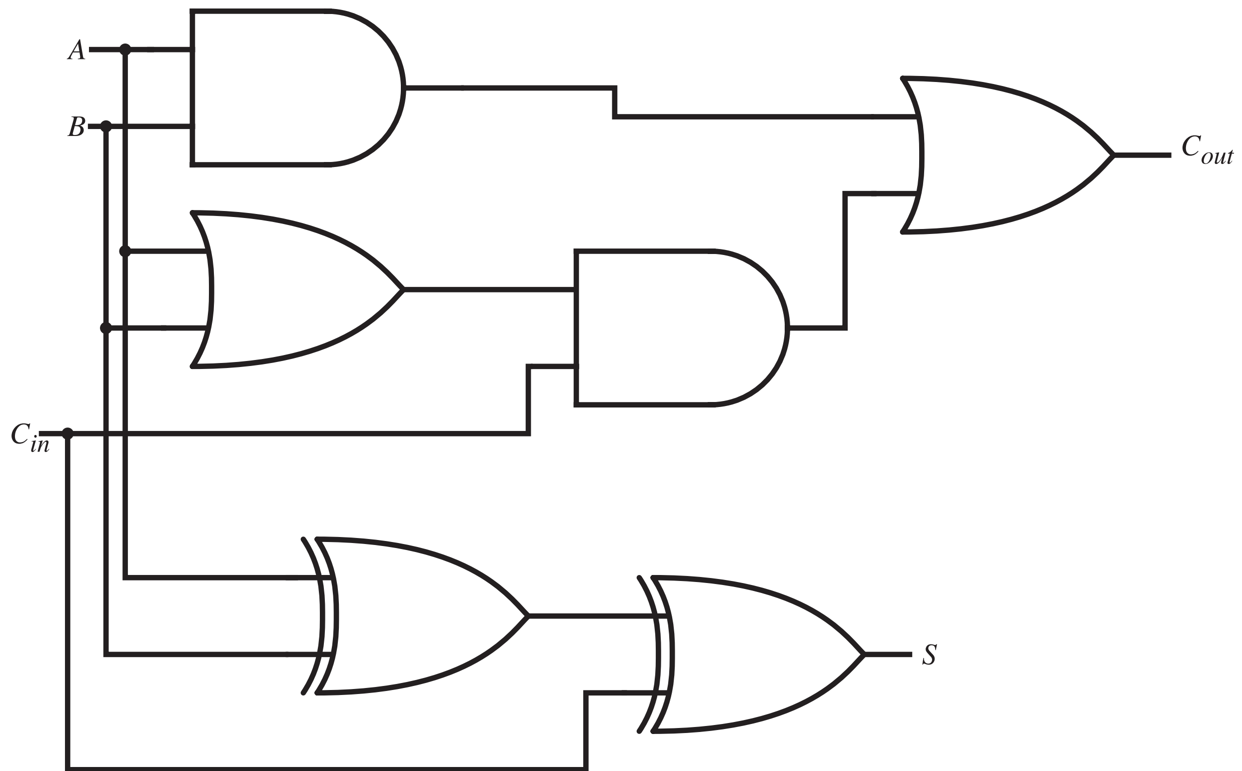
which we can represent symbolically by

$$C_{out} = AB + C_{in}(A + B)$$

We can implement this with the following logic gates:




Putting the two parts together, the logic gate implementation of a **one bit full adder** is:




Being able to add two numbers together might not seem like that big a deal, but you will see that the "instruction set" of operations that our microcontrollers is capable of performing consists of only a handful of very basic operations, most of which are not much more complicated than simple addition.

It is the ability of the digital circuits to do *lots* of fairly simple operations very quickly and accurately that leads to the incredible power of modern computers. This leads to the very powerful idea of **abstraction**: Very simple building blocks can be "glued together" to creative more powerful (and specialized) building blocks, which can in turn be glued together to create even more powerful building blocks...

### Arithmetic Logic Unit (ALU)

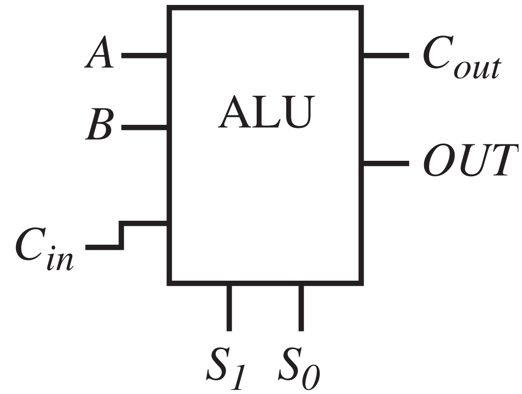
(See page 307 of *Student Manual*.)

An **arithmetic logic unit (ALU)**, is a device which can perform any one of a selected instruction from the instruction set. It is a central feature of just about any computer.

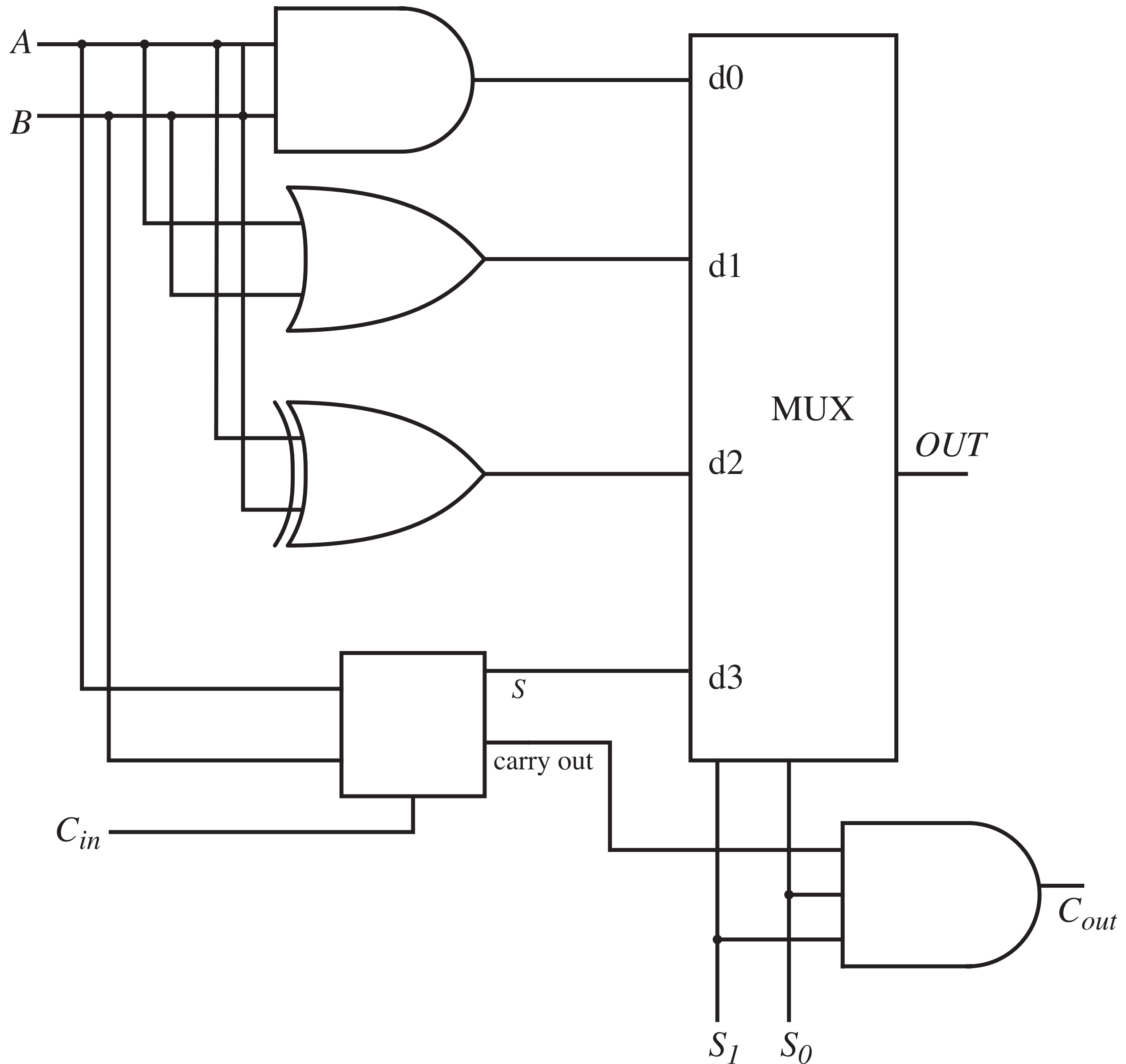
**Design a "1-bit ALU".** Two select lines should determine which of the following set of instructions the ALU performs on two input bits *A* and *B*: AND, OR, XOR,

and ADD. Include "carry in" and "carry out" lines to support the ADD function.

Schematically the ALU will look like:



**Implementation:**




The MUX is a “multiplexer” unit. A two bit “select” number chooses one of four input lines to be output. It can readily be built out of simple logic gates. See pages 296 – 299 of the *Student Manual* for more on multiplexers.