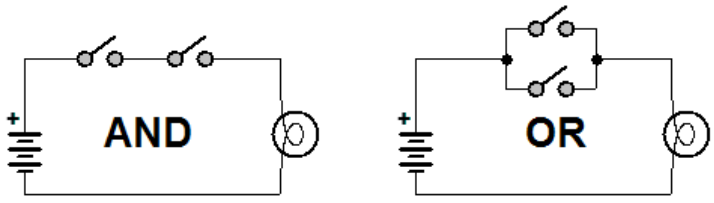
Lecture #21 – March 31, 2004

Introduction to Gates and Circuits

To this point we have looked at computers strictly from the perspective of assembly language programming. While it is possible to go a great distance with just software tools, the best understanding of the underlying models can only be achieved through study of the hardware components and associated computer architecture.

For example, we have written assembly language programs that use `AND` and `OR` instructions. `AND` instructions are used to *clear* bits in a word; for any value of `X` the bit expression `(0 AND X)` returns 0, and the bit expression `(1 AND X)` returns `X`. Similarly, inclusive-`OR` instructions are used to *set* various bits in a word; for any value of `X` the bit expression `(0 OR X)` returns `X`, and the bit expression `(1 OR X)` returns 1. For completeness, exclusive-`OR` instructions (denoted as either `XOR` or `EOR`, depending on the processor) are used to *complement* the bits in a word; the bit expression `(0 XOR X)` returns `X`, just like inclusive-`OR`, but the bit expression `(1 XOR X)` returns the 1's complement of `X`.

Rather than simply leave these functions to software, and trust that they always do what they are supposed to do, we can implement them in hardware. For example, we can easily build test circuits for `AND` and `OR` with no more than a battery, a light bulb (of the appropriate voltage), a handful of switches, and some wire. The following diagram shows the schematics for these two circuits.



Note that in the `AND` circuit *both* switches must be closed for the light bulb to light up, and if *any* switch is open the light bulb is off. In the `OR` circuit the situation is reversed: *any* switch may be closed for the light bulb to light up, and *all* must be open for the light bulb to be off. This relationship is codified in ***DeMorgan's Theorems***, which we will address at a later time. In the `AND` circuit it is pretty easy to add a third switch in series with the other two, and in the `OR` circuit it is just as easy to add a third switch in parallel. Any number of switches can be added to either circuit.

We can consider "switch closed" to mean True and "switch open" to mean False; similarly "light on" means True and "light off" means False. In binary, True maps onto the value 1, and False maps onto 0. A ***truth table*** shows all of the behaviors for a circuit in terms of either True and False or 1 and 0; the two notations are identical.

For example, truth tables for AND and OR circuits with two inputs will have four behaviors that must be cataloged. The two inputs may be both False, one False and one True (in two different ways), or both True. In general, a circuit with N inputs is described by a truth table with 2^N rows (behaviors). Each time an input is added the number of behaviors *doubles*: all of the previous behaviors with the new input False, and all of the previous behaviors with the new input True. For two inputs labeled `A` and `B` the `AND` and `OR` truth tables are as follows (using both notations):

A	B	AND	OR
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

A	B	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

For three inputs there will be eight rows in the truth table; for `AND` the only True output will be the last one, and for `OR` the only False output will be the first. The pattern is the same no matter how many inputs are present. The general rules for these two functions, regardless of the number of inputs, are:

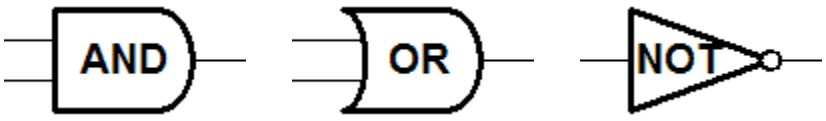
AND: The output is True if ***all*** inputs are True,
The output is False if ***any*** input is False.

OR: The output is False if ***all*** inputs are False,
The output is True if ***any*** input is True.

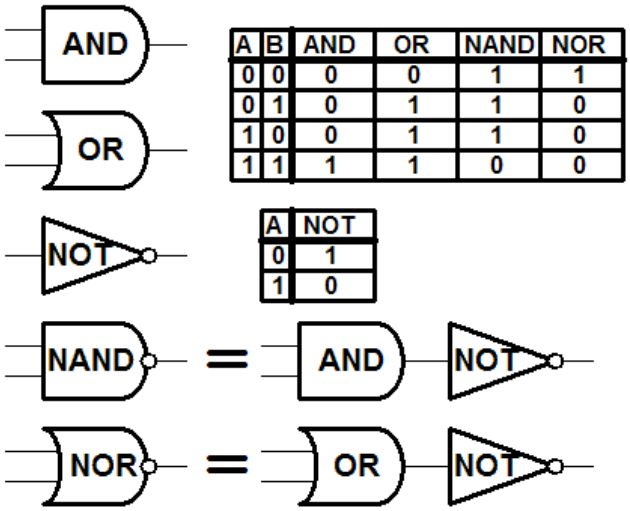
One more function is `NOT`, which always has a single input. The output is always the inverse of the input; if the input is 0 the output is not 0 (it's 1), and if the input is 1 the output is not 1 (it's 0). The general rule is:

NOT: The output is False if the input is True,
The output is True if the input is False.

As it turns out, there are any number of ways we can build hardware to perform the `AND`, `OR`, and `NOT` functions (and others). The major differences between any two hardware technologies are in size, speed, reliability, and cost, but the underlying mathematics remain invariant. We can abstract out the mathematical parts so that the technology doesn't matter; a ***gate*** therefore is a mathematical abstraction for a mechanical device that implements the function of a truth table. The standard shapes for the three most common gate types are as follows, each with inputs on the left and the output on the right:



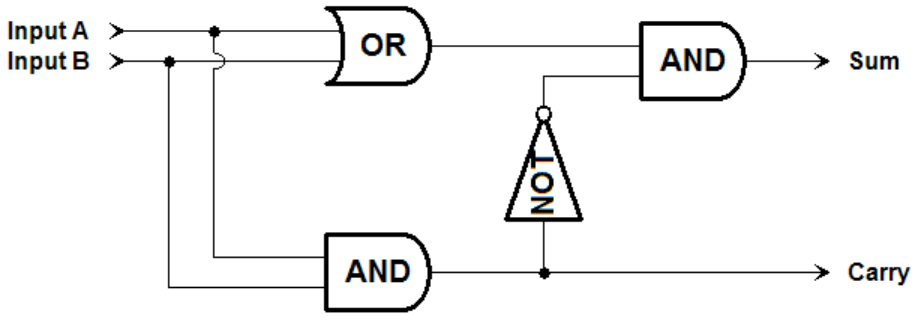
The OR and NOT gates can be combined to form a NOR gate, and the AND and NOT gates can be combined to form a NAND gate. All of these gates are shown below, along with their respective truth tables.



A	B	AND	OR	NAND	NOR
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

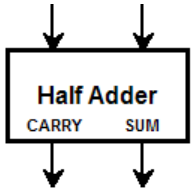
A	NOT
0	1
1	0

Gates have little use in and of themselves, but they can be combined to form powerful circuits. One of the most useful is shown below. In order to understand the operation of a device such as this you must trace it for each possible behavior. This means applying every possible set of values to the inputs, letting those values flow through the circuit (as each gate generates its appropriate outputs those values flow on to the inputs of the next gate), and observing the final outputs. Wherever two lines cross with an "overpass" or "croquet wicket" there is no signal communication from one line to the other. Where lines join with a dot a connection exists, and a value asserted at one point on the line is seen everywhere along that line, essentially simultaneously.

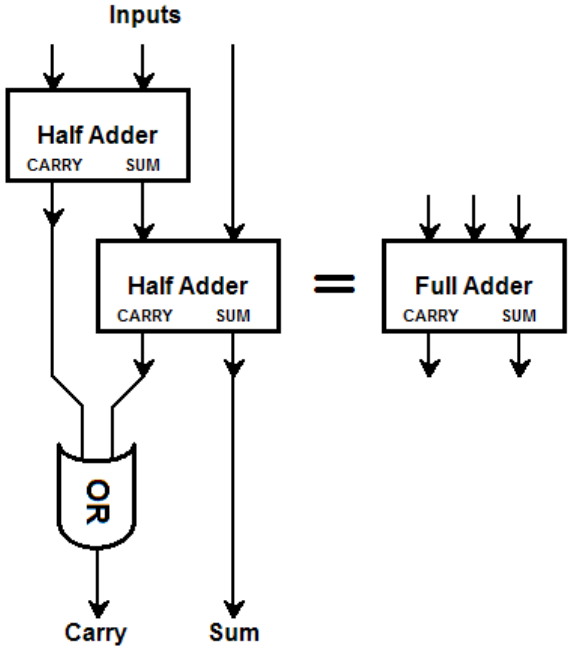


This circuit adds two bits together in binary, generating a sum and a carry. Treating the carry-sum combination as a 2-bit binary number, tracing the circuit for all possible behaviors reveals that 0+0=00, 0+1=01, 1+0=01, and 1+1=10. This device is called a ***half-adder*** because it cannot also add in a carry from a lower-order bit, and so does only about half of what is needed in a general purpose binary adding machine.

This circuit occurs so frequently that it has its own diagram. Indeed, this is extremely common in circuit design; as circuit modules are designed they are abstracted into "black boxes" that can be used in subsequent designs without worrying about the implementation details. The block for a half-adder is shown below:

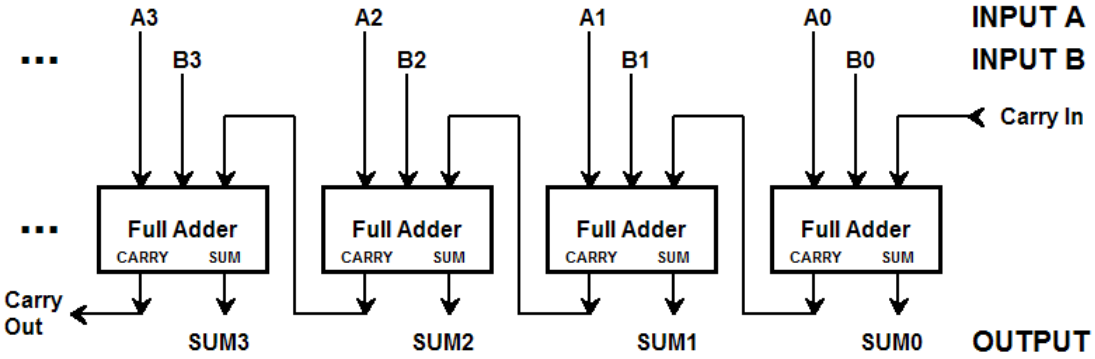


To create a ***full-adder***, or a device that can add together three bits (two operand bits and a carry from a lower stage), you need two half-adders plus an extra `OR` gate to "glue" them together. As before, that combination occurs so frequently that it has its own abstract symbol, as shown below:



With enough full-adders you can build a general purpose ***ripple-carry binary adder***. Adding two N-bit operands requires N full-adders. Each full-adder in the chain adds together a bit from each operand and the carry-out from the full-adder on the right, producing a sum bit and a carry to the next full adder on the left. While this is a general and extensible design it suffers from speed problems, as it is possible for a carry to ripple all the way from the right-most bit to the left-most bit. Modern adder designs use techniques to reduce or eliminate carry-ripple.

The following circuit shows a ripple-carry adder. Each full-adder contains two half-adders and an `OR` gate, and each half-adder contains two `AND` gates, an `OR` gate, and a `NOT` gate. Multiply those counts by the number of bits in the adder and you can get a feeling for the underlying complexity of the circuit (and this isn't a particularly complicated circuit).



Questions that we will address in the next lecture include:

1. What do we do with the carry in and carry out lines?

2. How can we use the adder to also perform two's-complement subtraction?