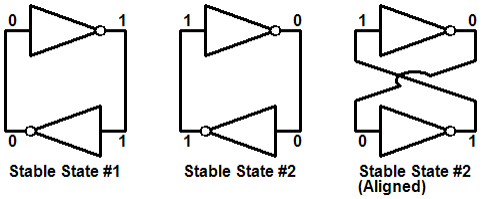
Lecture #23 – April 5, 2004

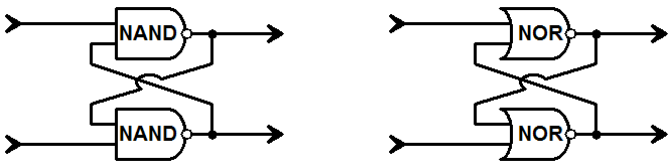
Flip-Flops

Up to this point we have looked at circuits without feedback, where the outputs on the right are strictly a function of the inputs on the left. In this lecture we will look at devices which *do* have feedback. These devices provide memory functions for computers, particularly to the central processing unit in the form of registers. We will also begin our look at mechanisms for creating main memory (RAM) devices.

For the first example, take two NOT-gates and cross-couple the output of each one to the input of the other. The resulting circuit has no external inputs, so we have to imagine that values "somehow" get established somewhere. If we assume that the input to the top NOT-gate is 0, then its output is 1, which goes to the input of the bottom NOT-gate. The output of the bottom NOT-gate is 0, feeding back to the input of the top, which agrees with our initial assumption. This configuration is stable, as long as the power holds. If we make the opposite assumption, and set the top input to 1, we find that this is also a stable state. Such a device is called ***bistable*** because it has two stable states. The figure below shows the circuit in each of its two stable states. By twisting the bottom NOT-gate around so that it faces the "normal" left-to-right direction of signal flow we see that the feedback loop now crosses over itself.

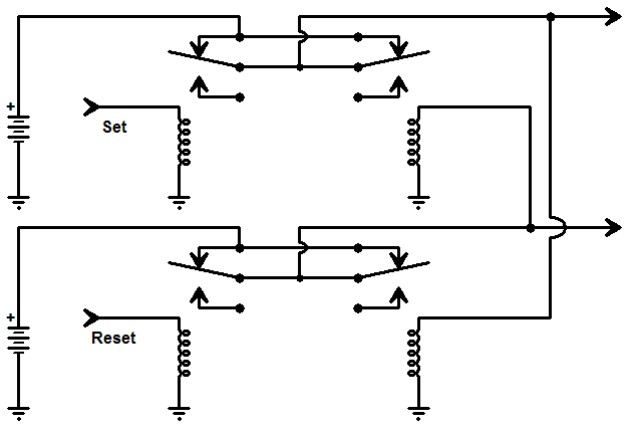


As stated earlier, the device as it stands has no inputs, so it cannot be used in any practical circumstance. The NOT-gates can acquire more inputs by turning them into either NAND-gates or NOR-gates. For the NAND-gate version, the "resting state" is when the inputs are equal to 1, in which case the NAND-gates act just like the earlier NOT-gates. Setting an input line to 0 forces the corresponding output to 1; this output then feeds around to the other NAND-gate, forcing its output to 0. As soon as this 0 feeds back to the first NAND-gate the circuit is stable and the input line can return to its resting state. Similarly, the resting state for the NOR-gate version is when its inputs are equal to 0. Setting an input to 1 forces the corresponding output to 0 and the other output to 1. The feedback loop works in the same manner as the NAND-gate version. Both versions are shown below:



These circuits are called ***flip-flops*** because they "flip" to one state and "flop" to the other. Flip-flops are also called ***bistable multivibrators*** (there are also ***astable multivibrators*** and ***monostable multivibrators***, which we will discuss at a later time). Flip-flops form the basic framework of memory and register bits.

In modern electronic circuitry we don't use relays, although it is fairly easy to do so. The circuit shown below is a relay version of the NAND-gate flip-flop.



Modern circuits use ***MOS transistors (Metal Oxide Semiconductor)***, which are microscopic three-terminal semiconductor switching elements. Two basic types exist; one type "closes its switch" and allows current to flow between two of the terminals when the third terminal sees a positive voltage, and the other type closes its switch when the third terminal sees a negative voltage. When one of each type is used together, the transistors form a complementary pair, or ***CMOS (Complementary Metal Oxide Semiconductors)***. A characteristic of gates built from CMOS devices is that they use nearly no power except when they are changing state from 0 to 1 or from 1 to 0. The faster they switch the more power they use.

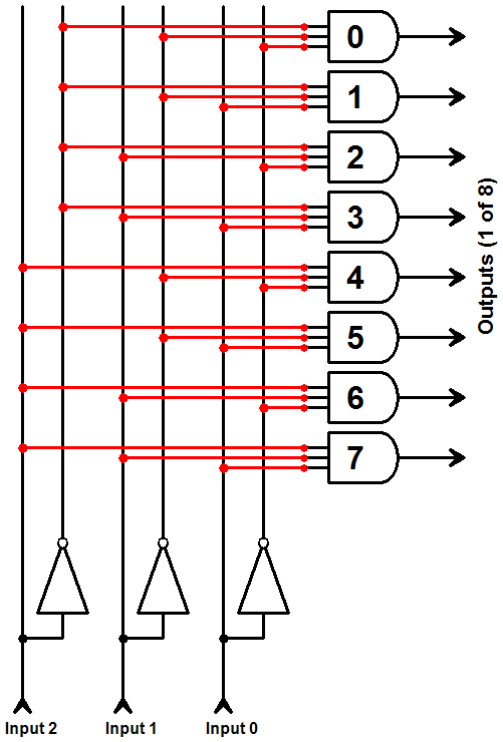
A memory cell with cross-coupled gates and appropriate input-output lines requires six such transistors. Memory cells built this way are called ***static*** because they retain their contents as long as power is applied, but are also called ***volatile*** because they "lose their mind" when power is removed. Static memory is very fast to read from and write to, but it is expensive in large quantities because of the number of transistors per cell. Typically, static memory is used in registers and caches where a small amount of very fast memory is necessary, but it is not used in primary memory.

Another technology which is used in primary memory is called ***dynamic memory***. Dynamic memory requires only one transistor and a small capacitor per memory cell, so it is significantly smaller in size and cheaper than static memory. Bit values are stored as charges in the capacitor; a large charge for a 1 and little or no charge for a 0. The transistor allows current to pass only when reading from or writing to the capacitor; when the transistor is turned off the capacitor is isolated from the rest of the circuitry and (theoretically) retains its charge. Unfortunately, the capacitor is so small that its charge leaks away in a few milliseconds. The value of every capacitor in the memory system must be read out and the appropriate charge replaced before the charges leak away to the

point where the data values cannot be recovered. This is why this type of memory is called *dynamic*; it must be monitored and maintained continuously or it won't work. In the original IBM PC dynamic memory was refreshed under software control; every few milliseconds the PC would have to stop what it was doing to go refresh the memory. The interval between refreshes could be lengthened to give a larger proportion of time to application programs, and thus improve performance by a small amount, but if the interval was lengthened too much then the charges would leak away and the system crashed. Today, dynamic memory chips contain built-in refresh circuitry, so the issue is mostly moot.

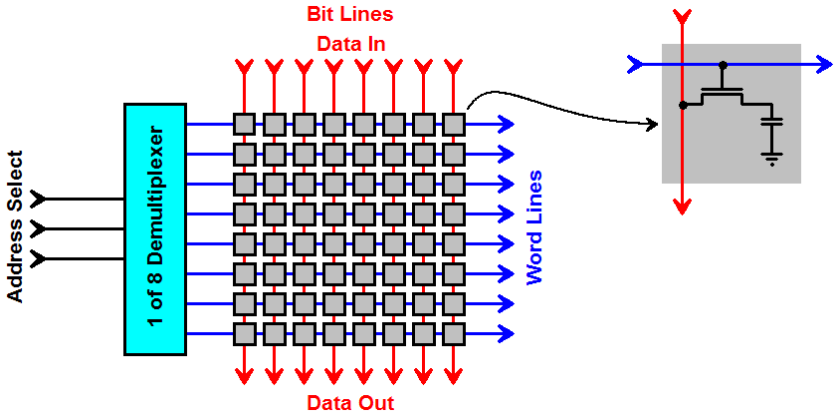
Regardless of the type of memory used, static or dynamic, each memory cell has a ***word line*** which selects the cell, and a ***bit line*** for writing a new value into the cell or reading out its value. (Cells of static memory require both a bit line and its complement, but dynamic memory cells require only the bit line.) All memory cells that are part of the same location in memory (byte, word, double word, whatever) are connected to the same word line so that they are all activated at the same time. All corresponding bits from different words are connected to the same bit lines; that is, all bit 0s are connected together, all bit 1s are connected together, etc. This forms a two-dimensional grid of memory cells, requiring that at most one word line be activated at any one time.

Insuring that only one word is activated at any time requires a circuit called a ***demultiplexer***, which has N input lines and 2^N output lines. The basic demultiplexer requires 2^N AND-gates, one for every output, and each with N inputs. All of the inputs are available to the AND-gates, as are the complements of all inputs. Each AND-gate taps into a unique combination of either an input line or its complement, as shown below:



The example shown has three inputs, so it must have 2^3=8 outputs, numbered from 0 to 7. Every binary pattern activates exactly one of the AND-gates. Every AND-gate in the example is inscribed with the number that corresponds to its binary trigger value. Place any binary value on the input lines and trace the signal flow through the circuit to prove to yourself that only a single AND-gate is ever activated.

The following diagram shows an 8×8 grid of memory cells, corresponding to a ***random access memory*** system containing eight bytes of RAM. The address of the desired byte is placed onto the input lines of the demultiplexer, which activates the word line to the correct byte in the memory grid. Placing any new value on the bit lines writes those values into the selected byte. In this case the memory bits are dynamic, so activating a word line causes its transistors to connect its corresponding capacitor to the bit lines; placing values onto the bit lines set the charges in only those selected capacitors, and reading out the byte means measuring the charges placed onto the bit lines by those same capacitors.



If you try to generalize this mechanism, you'll soon come to the realization that a demultiplexer would require over four billion 32-input AND-gates to fully populate a 32-bit address space with memory. This is impractical for a number of reasons, not the least of which is that no single device can drive four billion inputs. Instead, we can get by with a much smaller demultiplexer if each word line activates many more bits than are required by any single storage location, and then a special selection circuit (called a ***multiplexer***) picks out the desired portion of what was actually fetched. In our example we have eight words of eight bits per word; we could instead have had four words of 16 bits per word, and then picked the desired 8-bit byte out of the 16-bits that were fetched.

For any specified memory size with an N-bit address, we can have a complete 2^N demultiplexer and no multiplexer, no demultiplexer and a 2^N multiplexer, or something in between. For any such system there is a "sweet spot" which minimizes the overall amount of required hardware. We will look at such systems in a future lecture.