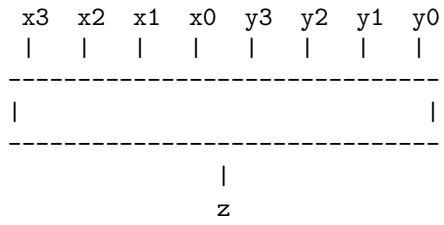
Circuits to test equality

The notes describe equality test circuits. Hardware design shares with programming the technique of abstracting common tasks into reusable modules, separating the implementation of the module from its use. In some other ways, it is quite different, since in hardware everything is "trying to happen at once" and the designer's effort is to control and orchestrate the activity, while in (sequential) software, there is a single focus of activity, which the programmer is attempting to direct in useful and economical ways. Though not exact, one possible comparison is football to golf.

Compare for equality

Suppose we have two 4-bit quantities and we want a circuit that outputs 1 if they are equal, 0 if they are different. Pictorially:



Each input wire, x_i or y_i, is either a 0 or a 1, representing one bit of the input x or y, respectively. The output wire z should have a 1 if the corresponding bits of x and y are all equal, that is, $x_0 = y_0$ and $x_1 = y_1$ and $x_2 = y_2$, and $x_3 = y_3$.

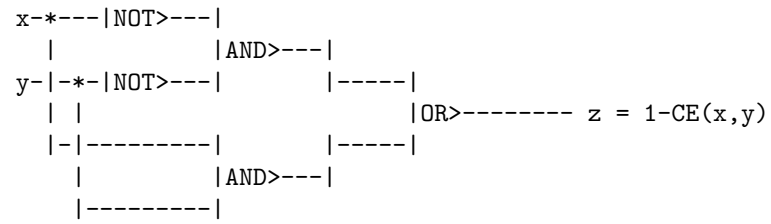
As an intermediate step, we'll design a circuit that has two inputs, x and y, and gives an output of 1 if the two input bits are equal, and 0 if they are not equal. This function we call 1-CE, for 1-bit compare-for-equality. Its truth table is:

x	y	1-CE
0	0	1
0	1	0
1	0	0
1	1	1

If we directly apply the sum-of-products algorithm to this truth table, we get the boolean expression;

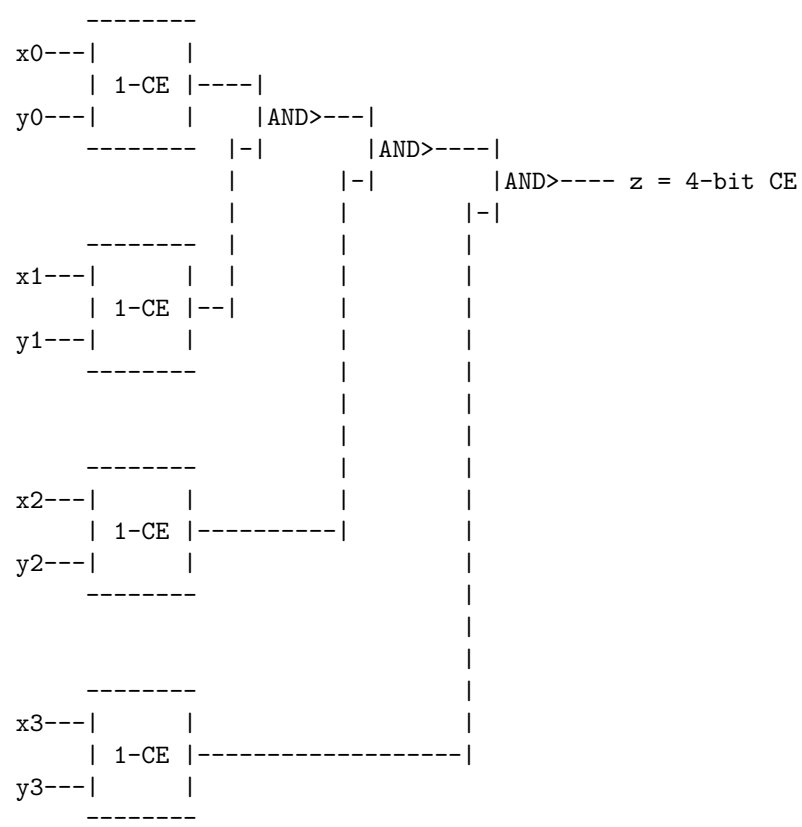
$$x'y' + xy,$$

which translates into the combinational circuit:



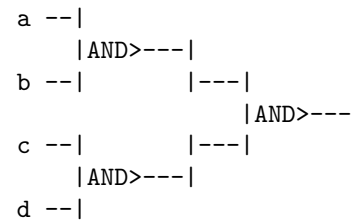
Thus, with 5 gates we can implement the 1-bit compare-for-equality function, 1-CE. If XOR gates are available, we could use the fact that the 1-CE function is the complement (NOT) of the XOR function. Now we abstract this circuit as a box that we may use in other circuits. This is analogous to defining and using a procedure in a program.

If we compare each pair of bits x_i and y_i using a 1-CE circuit, then if all of them return 1, the 4-bit compare-for-equality circuit should return 1. If any of them return 0, the 4-bit compare-for-equality circuit should return 0. Thus, if we AND together the results of the four 1-bit comparisons, we'll have the correct overall answer:



Note that for clarity, I've rearranged the inputs so that the pairs to be compared are adjacent.

The AND gates could be combined in a different order, for example:



This corresponds to the expression $((a \cdot b) \cdot (c \cdot d))$ rather than $(((a \cdot b) \cdot c) \cdot d)$. The advantage of this form is that it can compute its result faster. In a physical realization of a gate (as transistors or relays or optical elements), there is some small time delay between the time that the input signals settle down and the time the output signal settles down. An attempt to use the output signal before it has settled could lead to an error. The time is a gate delay, and depends on the implementation of the gate and its type. When the output of one gate is an input to another gate, then the output of the first gate must settle down, and only after that can the output of the second gate begin to settle down. Thus, at least two gate delays are required for a reliable output in this case.

The AND structure in the diagram for the 4-bit CE requires 3 gate delays, whereas the AND structure above only requires 2 gate delays. In general, by arranging the AND's in a binary tree, we can compute the AND of n inputs with $\lceil \log n \rceil$ gate delays, as opposed to $n - 1$, an exponential improvement. Because the output of an n-bit AND depends on all the inputs, and we assume we have only 2-input AND gates, there is also a lower bound of $\lceil \log n \rceil$ on the depth of a circuit to compute an n-bit AND. Thus, the tree-structured AND-circuit is optimal under these assumptions.