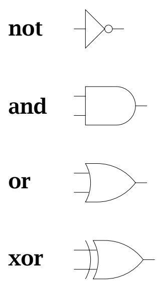
Computer Science 52

Digital Logic Circuits

Spring Semester, 2017

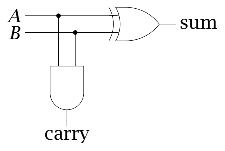
In a physical computer, bits are stored as electrical charges, and boolean operations are evaluated with components called *gates.* For each operation, there is a gate that takes in electrical signals encoding to one or two bits, and produces the corresponding result bit as the output. (Sometimes a gate will have more than two input nodes. For and- and or-gates, the behavior of such a gate is clear. Be careful with other gates, like the xor-gate.)



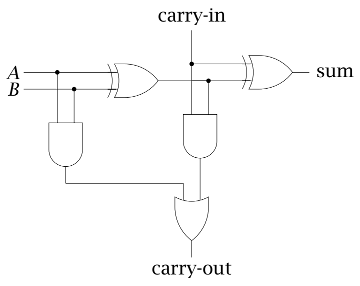
The symbols for the four most common gates appear at the right. Gates can be assembled into complicated circuits that carry out computations on many bits. A typical smartphone has a processor with nearly a billion gates and a memory with several billion bits. We shall see simple examples of such circuits here and on an Assignment.

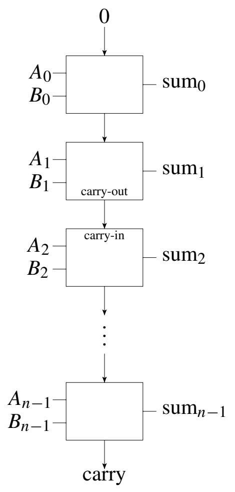
I Circuits for Addition and Subtraction

Let us begin with addition. If we are adding two bits A and B, the sum bit is simply the $A \oplus B$ and the carry bit is $A \wedge B$. The circuit below is a *half adder* that computes both the sum and the carry.



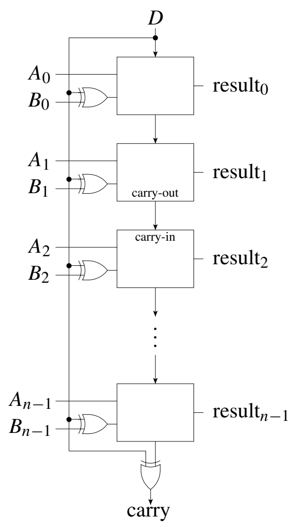
Two half adders can be combined to form a *full adder* which corresponds to one column of an addition operation. The inputs are the bits A and B and a carry-in bit from the previous column. The results is the sum bit and a a bit to carry out to the next column.





If we want to add n-bit words, we can do it with a circuit constructed from n full adders, as shown in the top diagram on the right. Each box is a full adder that corresponds to one of the n columns in the addition problem. The carry-in to the low order column is 0, and the carry-out from the high order column is discarded. The circuit is called a *ripple-carry adder.*

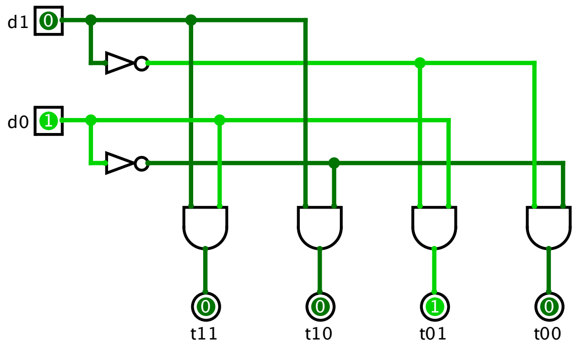
If we want to subtract n-bit words, we use a slight modification of the ripple-carry adder. Recall that subtraction is negation followed by addition and that negation is complementation followed by adding one. The bit D in the circuit on the right controls whether the operation is addition or subtraction. If D is zero, the circuit behaves just like the ripple-carry adder on the left. If D is one, the circuit computes the sum of the A bits, the complement of the B bits, and 1. That operation amounts to subtraction.



Notice the xor-gate modifying the carry when the circuit is subtracting. That allows the bit to signify a "borrow" into the high-order bit and ensures that the bit is the correct value for the C flag.

II Decoders

Another example of a basic circuit is a 2-bit decoder, pictured below. You can think of it as a binary to unary converter. There are two inputs and four outputs. Exactly one of the outputs will be 1. Which one is determined by the value, in binary, of the two inputs. As illustrated, the input value d_1d_0 is 01, and the output t_{01} is 1.

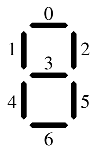


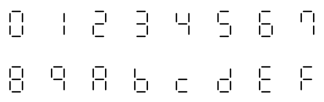
The image of a decoder is taken from Logisim, a software tool that we will be using. The dark green wires carry a value of 0, and the bright green ones carry a value of 1. The squares are input nodes, and the circles are output nodes.

It is not hard to generalize the idea to a k-bit decoder, which has k input nodes and 2^k output nodes. There is one output node for each row in a truth table with k variables.

Decoders provide a general idea for generating many kinds of circuits. As an example, consider the segment display found on many alarm clocks and digital watches. There are seven elements, as shown on the left in the illustration below. Different combinations of the elements may be illuminated to display the hexadecimal digits shown on the right.

We use lowercase `b` and `d` to distinguish them from `8` and `0`. We use lowercase `c` because it looks better along side the other letters.





Suppose we want to create a circuit that determines whether segment 1, on the upper left, is illuminated. The input to our circuit consists of four nodes, representing the value of the hexadecimal digit to be displayed. we create a 4-bit decoder with four input nodes and sixteen output nodes, numbered from `0x0` through `0xF`, one for each of the possible digits. Segment 1 is illuminated when the digit is

`0, 4, 5, 6, 7, 8, 9, A, B, E,` or `F`.

We attach the corresponding eleven output nodes of the decoder to an or-gate, and the output of the or-gate will be 1 when segment 1 is to be illuminated. We could create a separate circuit for each of the other segments of the display, but there is no need for more decoders. We can attach the output nodes of a single decoder to different or-gates to obtain the nodes controlling the seven segments of the display.

The resulting circuit is simple in the sense that it has only a few gates—four not-gates and seven (multi-input) or-gates. But it is complicated in the sense that there are many connections. One must take care in drawing such a circuit to avoid excessive entanglement.

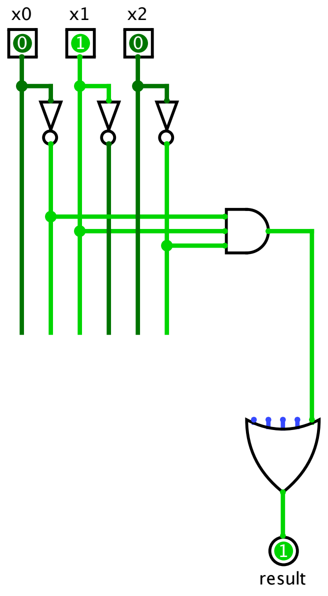
III Minterm Expansion

Any logical operation on several values can be described with a truth table. The hexadecimal display example leads us to a technique for converting a truth table into a circuit.

Given a truth table, build a decoder-like circuit, but include only those and-gates and output nodes that correspond to rows of the truth table for which the output is 1. Then connect all the outputs of the and-gates to an or-gate to produce the single output value. For example, if one row of a truth table is

x0	x1	x2	result
	⋮		
0	1	0	1
	⋮		

then a portion of the minterm circuit would look like the one on the right. Notice that we have reoriented the circuit to place the inputs at the top.



The minterm technique gives us the power to create any possible logical circuit. The circuit will always have a depth of three. That means that any signal has to pass through at most three gates to reach the output. The depth is significant because it takes a signal some period of time to propagate through a gate. The more gates there are, the longer it takes. With a maximum of three gates along any path, the circuit responds quickly.

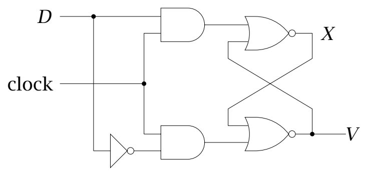
On the other hand, the minterm technique produces circuits with *lots* of gates, which affects the total power consumption and the reliability of the circuit. There are many methods, some manual and some automated, that are used by computer designers to minimize the number of gates in a circuit. We will not delve into those techniques in this course.

Another way to think of the technique is as a way to convert a truth table into a boolean formula. The formula will be a disjunction of "minterms," each of which is the conjunction of variables and negations of variables. Such a formula is said to be in *disjunctive normal form.*

There is also a *maxterm* expansion technique that produces circuits with many or-gates and a single and-gate to produce the output. The corresponding formula is a conjunction of "maxterms," each of which is a disjunction of variables and their negations. That formula is in *conjunctive normal form.* You will encounter conjunctive normal form in more advanced computer science courses.

IV A Memory Circuit

Because a computer performs a sequence of operations, there must be a way of storing the result of one step for use in a later step. The circuit below is a one-bit memory, called a *clocked D-latch*.



The circle at the output of the or-gate indicates that the result is negated. The operation is called *nor.* Although the operation is easy to state, the mechanism may be a little difficult to understand.

The line labeled clock is normally 0. While it is 0, the output line V maintains a value that does not change. It can be used as input to other circuits, possibly over a long period of time. The value of V is changed by raising the clock signal momentarily. When that happens, V takes on whatever value is on the input line D. As with an adder, it is easy to imagine several latches connected in parallel to provide memory for a word.

Why does the clocked D-latch work? One could trace through the circuit with all possible values of D, clock, and V, but that is confusing and unenlightening. An alternative is to translate to logical formulas and use some identities. For notational convenience, let us use C for the clock signal and V for the output. Let X be the output of the top nor-gate as indicated above. The outputs of the two and-gates are $D \wedge C$ and $\neg D \wedge C$, so we have

$$\begin{aligned} X &= (D \wedge C) \mathbin{\textbf{nor}} V \\ V &= (\neg D \wedge C) \mathbin{\textbf{nor}} X \end{aligned} \tag{1}$$

When C is 0, the outputs of both and-gates are also 0. Using the identity $0 \mathbin{\textbf{nor}} A = \neg A$, which is easy to check, the equations become

$$\begin{aligned} X &= \neg V \\ V &= \neg X \end{aligned}$$

This is the stable situation mentioned above.

When C is true, the equations (1) become

$$\begin{aligned} X &= D \mathbin{\textbf{nor}} V \\ V &= \neg D \mathbin{\textbf{nor}} X \end{aligned}$$

This is an application of the identity $A \wedge 1 = A$ from Table 1 in *Bits, Words, and Integers.* Because working with **nor** is unintuitive, we use another easily-checked identity, $A \mathbin{\textbf{nor}} B = \neg A \wedge \neg B$, to rewrite the equations.

$$\begin{aligned} X &= \neg D \wedge \neg V \\ V &= D \wedge \neg X \end{aligned}$$

Eliminate X by substituting the right-hand side of the first equation for X in the second, and use a DeMorgan law to obtain

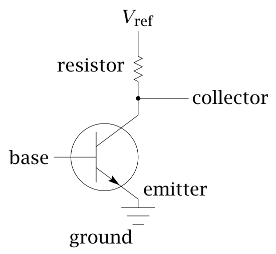
$$V = D \wedge \neg(\neg D \wedge \neg V) = D \wedge (D \vee V).$$

Truth tables now tell us that the only way that this can hold is if $V = D$, which is what we claimed happens when the clock signal is high. When the clock goes low again, V retains its value.

This kind of memory is usually used on the processor unit itself. It is too complicated and expensive to be used for the computer's main memory. The random access memory on a computer is usually implemented with simple capacitors that store an electrical charge.

Appendix: Transistors

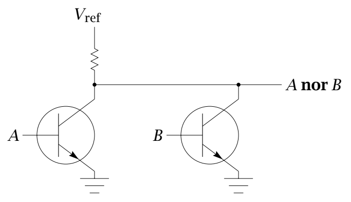
Although most computer scientists can function quite well without knowing exactly how gates work, it is interesting to go one level deeper into the physical computer. Most gates are constructed from semi-conductor devices called *transistors.* A transistor is a kind of switch, shown as the circular object in the diagram on the right. The signal on one wire, called the *base*, controls the current flowing between two other wires, the *collector* and the *emitter*.

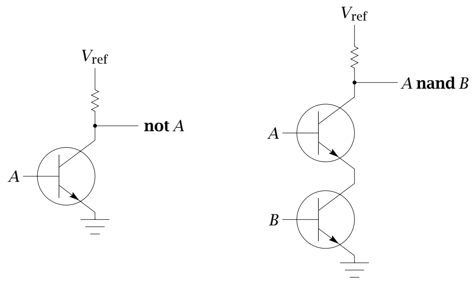


In digital circuits, there are only two relevant voltages—low and high. They are the ones at the points marked ground and V_{ref}, respectively, in the diagram above. In other applications, like stereo sound equipment, transistors are used as amplifiers, and the variations in the signal at the base are reproduced with greater magnitude at the collector.

We take the voltage at the point marked "ground" to be zero, and supply a higher voltage at V_{ref}. When the voltage at the base is near zero, there is no connection between the collector and the emitter, and the voltage at the collector is close to the reference voltage. Conversely, when the voltage at the base is high, current can flow between the collector and the emitter. The collector is effectively attached to the ground, and the voltage level is zero there.

The two voltage levels can be interpreted as bits, with the ground state being 0 and a reference voltage being 1. When the base of a transistor is 0, the collector is 1, and *vice versa.* Under this interpretation, the transistor illustrated above is a not-gate whose input is the base, and output is the collector. With two transistors, we can construct nor- and nand-gates, as shown in below.





Using nor, not, and nand and the logical identities like those in Table 1 of *Bits, Words, and Integers,* one can construct all the gates that we have discussed. With a large enough supply of transistors, one can construct a computer. In the year 2000, a processor chip had about twenty million transistors—give or take a factor of four. Today, a typical number is a few billion.