Digital Circuit Design

Eric Hehner
University of Toronto

Introduction

The word “circuit” is related to the word “circle”. It was used to describe a closed electrical path that goes from one end of an electrical battery, through wires and machines, doing work on the way, and returning to the other end of the battery. That's why there are two wires in the cord of an electrical device (such as a lamp or computer): electricity goes from its source to the device in one wire, and returns from the device back to the source in the other wire, completing the circle or circuit. These days, we use the word “circuit” for any sort of electrical or electronic device. The word “digital” is related to the word “digit”, which means finger. Since fingers are used for counting, digit also means a symbol used in the representation of a number. A digital circuit is an electronic device that has discrete states, like the display of a digital clock. You can count the number of different possible states of the device. This is in contrast to the old, round clocks that had hands that move; they have continuous states. This short course is about the design of digital circuits, including but not limited to computers.

In the 1940s when computers were new, they were programmed in machine language, which is a sequence of 0s and 1s, using physical switches that were off or on. In the 1950s, programmers realized that there is a better way. They invented what they called “high-level” programming. They used three-letter mnemonics for the machine instructions, and identifiers (names) for the addresses. Today, we call that “assembly language”. At the end of the 1950s, programmers invented what we today call “high-level languages”. Programs now are not just mnemonic versions of machine instructions. They describe the data structures and algorithms of the computation, rather than the machine instructions that perform the computation.

Digital circuit design is going through the same evolution, but much more slowly. For many years, circuits have been designed by choosing the circuit elements (gates, switches, flip-flops), connecting them with wires to achieve the right functionality, and arranging the circuit elements and wires in the best spatial layout. In the 1980s and 1990s, circuit designers invented what they called a “high-level” approach to circuit design. They used mnemonics for the circuit elements, and identifiers (names) for the wires, in a circuit-design language. The two most popular circuit-design languages are VHDL and Verilog. These languages are like assembly languages in the sense that a program directly describes a circuit, just like an assembly-language program directly describes a machine-language program.

In this short course, we take the next step. We use a modern high-level programming language (C++ or Java or Python or ...) to write programs for the algorithms we want to compute, not to describe the circuits to compute the algorithms. Then we compile the programs to produce circuits to compute the algorithms. For example, you write a program to compute the number of times a given word occurs in a given text, and compile the program to produce a circuit that computes the number of times a given word occurs in a given text. You can write a browser program, and compile it to produce a browser circuit. You can write a program to interpret a machine language, and compile it to produce a computer with that machine language.

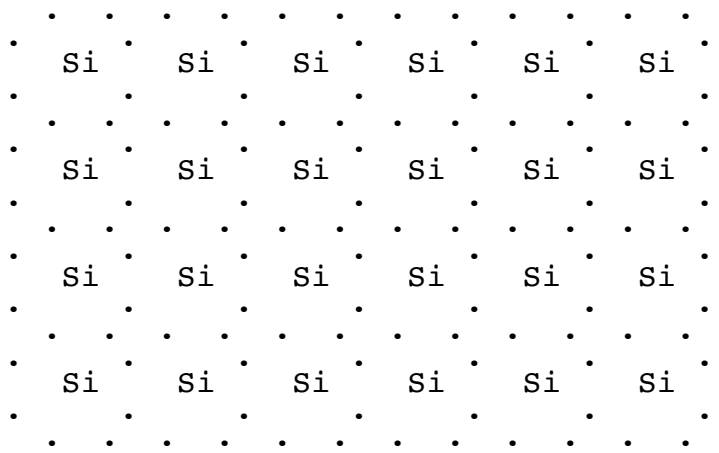
But before we get to the programs, we start at the very beginning of circuit design: we start with how circuit elements are built from elementary particles. When we are done, you will know how circuits are designed and how they work, from the elementary particles right up to the most complex circuits in existence.

Physical Base

Matter is made of molecules; molecules are made of atoms; atoms are made of protons, neutrons, and electrons. Protons are positively charged, and electrons are equally negatively charged. A neutron, which consists of a proton and an electron, is neutral, or uncharged. The protons and neutrons are the nucleus, at the center, of an atom. Electrons surround the nucleus in shells, or layers. The outermost shell of electrons determines the electrical characteristics of the material. If the electrons in the outer shell are strongly bound to the atom, the material is called a "resistor". Silicon, which is what glass and ceramics are made of, is a resistor. If the electrons in the outer shell are weakly bound to the atom, the material is called a "conductor". Copper is a conductor. Different materials occupy different places along the spectrum between very conducting and very resisting. Toward the conducting end of the spectrum, electrons can move easily from atom to atom. Toward the resisting end, electrons can still move from atom to atom, but it takes more energy. Here are the electrical symbols for a conductor (the line on the left) and a resistor (the zig-zag on the right; it's just a symbol, not the shape of the resistor).

A flow of electrons through the material in one direction is called "electricity", or "electric current". In a solid, the nucleus and electrons in the inner shells stay still, and do not flow. In a liquid or gas, atoms can flow, so electricity is a combination of free electrons flowing one way and atoms with more protons than electrons flowing the other way. But our circuits are made of solid material, so electricity is just electron flow.

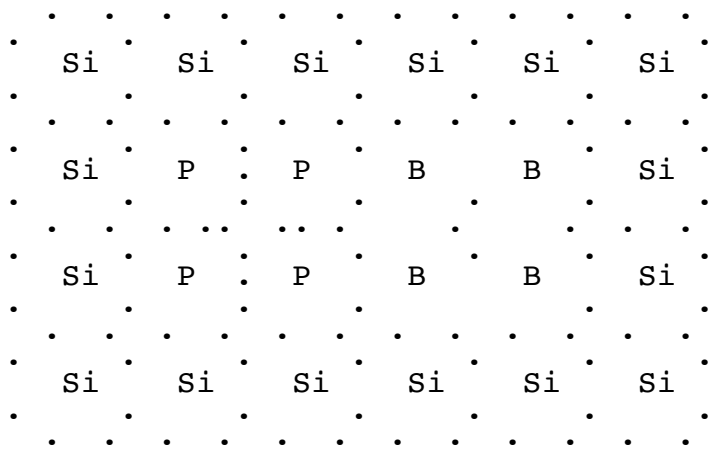
VLSI (Very Large Scale Integration) is the way digital circuits are currently built. First, silicon is heated so hot that it melts and becomes liquid. Then it is poured onto a clean flat surface in thin pancake-size circles, called "wafers". The silicon wafers cool and solidify into a crystal structure. The room where this takes place must be dust-free (it is called a "cleanroom"), and insulated from all noise and vibrations, in order to get a good crystal structure. Here is a picture of a silicon crystal structure.



The letters Si stand for the nucleus of the silicon atoms. The dots are the electrons in the outer shell of these atoms. There are 4 electrons in the outer shell of each silicon atom, but as you can see, these electrons are shared between pairs of atoms. You can't say which atom an electron belongs to; you can say only that it belongs to a pair of neighboring atoms. The picture is inadequate in two ways. First, a crystal wafer is 3-dimensional, not 2-dimensional as shown; the atoms are arranged in a regular tetrahedron pattern. Second, an electron is not a dot sitting in a specific place. What the picture is intended to convey is that the silicon crystal is a regular

structure of atoms, each atom has 4 nearest neighbors, and each electron in an outer shell is strongly bound to a pair of atoms. A silicon crystal structure is a resistor.

The next phase in VLSI fabrication is called “doping”. Some specially selected silicon atoms are chemically replaced by phosphorus atoms, which have 5 electrons in the outer shell. Other specially selected silicon atoms are replaced by boron atoms, which have 3 electrons in the outer shell. Here is a picture (with the same inadequacies as before):



The letter P stands for a phosphorus nucleus, and B stands for a boron nucleus. The extra electron of a phosphorus atom is attracted to the nucleus of the atom, but it has no place in the crystal structure; so it is weakly bound to its atom, and easily dislodged. A region that is doped with phosphorus is said to be negatively doped. Each boron atom makes a place in the crystal structure for an electron that isn't there; that place is called a “hole”. A region doped with boron is said to be positively doped.

Along the boundary between the negative and positive doping, some of the extra, weakly bound electrons on the negative side drift away from their atoms, and some of them drift over to the positive side, where there are places for them in the crystal structure. But they are still weakly bound there because there is no corresponding proton in the boron nucleus. Electrons continue to drift around on both sides of the boundary, but they soon reach an equilibrium of density on the two sides. Silicon with positively and negatively doped regions is called a “semiconductor” (it could just as well have been called a “semiresistor”).

The next phase in VLSI fabrication coats the semiconductor surface with an oxide resistor, and then metal wire conductors are painted on the oxide surface, poking through the oxide into the semiconductor at just the right places. A “diode” is a pair of negatively and positively doped regions beside each other, with a wire poking into each region. If the wire in the negatively doped region is a source of electrons, and the wire in the positively doped region is a destination for electrons (or we could say a source of holes, or places for electrons to go), here's what happens. Electrons that drift across from the negatively doped region to the positively doped region continue into the wire that is an electron destination, and they get replaced from the wire that is an electron source into the negative region. The replacement electrons cross the boundary, and then go to the destination, and so on. A current flows through the diode until either the source of electrons is depleted, or the source of holes is depleted (the holes are filled up).

Now, suppose the wire in the negatively doped region is a destination for electrons. As before, some electrons from the negatively doped region cross the boundary into the positively doped region, but they do not get replaced in the negatively doped region, so nothing more happens; there is no current. Similarly if the wire in the positively doped region is a source for

electrons, the electrons that cross the boundary from the negatively doped region into the positively doped region have no further place to go, so there is no current. A diode is a one-way gate for current.

A destination for electrons is called a “positive voltage” or “high voltage” or “power”, and denoted ⊤ . A source of electrons is called a “negative voltage” or “low voltage” or “ground”, and denoted ⊥ . In a typical digital circuit today, the difference is about 5 volts. Voltage is a relative term; we could call the ⊤ and ⊥ voltages 5 volts and 0 volts, or we could call them 3 volts and –2 volts, or any other two voltages that differ by 5.

Electrons flow from ⊥ to ⊤ . Here is a series of pictures of electrons flowing from left to right. First, suppose we have

⊥ ● ● ● ○ ● ⊤

where each ● is an electron, and ○ is a hole. Then suppose the electron to the left of the hole moves into the hole.

⊥ ● ● ○ ● ● ⊤

Then again the electron to the left of the hole moves into the hole.

⊥ ● ○ ● ● ● ⊤

As electrons flow from left to right, the hole flows from right to left. Now suppose there are lots of electrons and holes, randomly placed.

⊥ ● ○ ● ● ○ ● ○ ○ ● ○ ⊤

When there's an electron to the left of a hole, the electron moves into the hole.

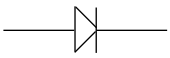
⊥ ○ ● ● ○ ● ○ ● ○ ○ ● ⊤

Also, the ⊥ at the left end fills the hole at the left end, and the electron at the right end moves into the ⊤ at the right end.

⊥ ● ● ○ ● ○ ● ○ ● ○ ○ ⊤

Electrons flow from ⊥ to ⊤ , and holes flow from ⊤ to ⊥ . Some people say that electricity flows from ⊤ to ⊥ ; they are talking about the flow of holes.

A diode allows electricity to flow in one direction only. Another way to say the same thing is that a diode supports a voltage difference in one direction only. It supports the voltage difference when it does not allow the current to flow. When it does allow the current to flow, either the electron source gets depleted or the holes fill up, and the voltages become the same. The electrical symbol for a diode is



The horizontal lines at the left and right ends are the wires into the positively (on the left) and negatively (on the right) doped regions. The triangle and vertical line are not shaped like the doped regions; they are just a symbol. Electrons can flow from right to left, or to say the same thing differently, holes can flow from left to right. The left end can be ⊥ and the right end can be ⊤ (no current), or both ends can be the same (no current). If the left end is ⊤ and the right end is ⊥ , current flows until both ends are the same; that direction of voltage difference doesn't last long.

A transistor is three doped regions in a row, either positive-negative-positive, or negative-positive-negative, with a wire in each region. The middle wire, called the “base”, can be ⊤ or ⊥ , and that determines whether the path between the end wires, called the “collector” and the “emitter”, is conducting or nonconducting. Its electrical symbol is

There are many variations in the way VLSI is produced and in the way it works, and many details that are not presented here. This short course is not intended to be about VLSI fabrication; we have presented just enough so that you can see how circuits, including computers, are built and how they work.

Water Circuits

There is a close analogy between water and electricity. The flow of electricity is like the flow of water. A power source or battery is like a pump that pumps water uphill. Voltage is like a height difference of water flowing downhill. A conductor is like a river bed or a wide pipe. A resistor is like a narrow pipe or a pipe with a kink in it. A diode is like a one-way valve. And a transistor is like a siphon. Actually, it's more than an analogy: falling water can be used to generate electricity, and electricity can be used to pump water uphill. Electricity and water flow are interconvertible.

Binary Abstraction

Digital circuit designers use only two voltage levels, which we call $\top$ and $\bot$, typically 5 volts apart. They pretend that the voltage level at any point in a digital circuit is one of those two levels. Over time, the voltage level at that point may change, but, according to digital circuit designers, it is always one of those two levels. That is the binary abstraction. Here is a picture of the voltage level at a point in a digital circuit, changing over time, according to digital circuit designers.

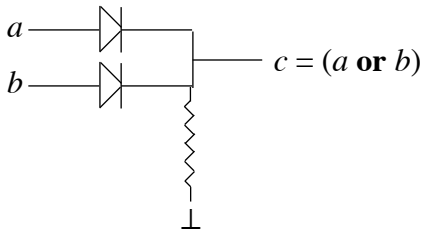
Time increases from left to right in the picture, and the voltage goes up and down. A sequence of changes from $\bot$ to $\top$ to $\bot$ is called a "pulse". This picture shows two pulses. What really happens is more like this.

Electrical engineers do their best, and they do amazingly well, to make the voltage levels be the way digital circuit designers want them to be: nice and square. That's so circuit designers can use simple binary algebra in their designs, rather than having to deal with complicated continuous voltage changes.

Gates

A simple circuit using the binary abstraction is called a "gate".

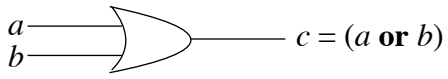
The first gate we present is the **or** gate. It has two inputs a and b, and one output c. If either input a or input b or both are $\top$, output c is $\top$. If both inputs are $\bot$, the output is $\bot$. We can build an **or** gate from two diodes and a resistor.



A voltage is the same all along a wire, so the voltage at input a is the voltage at the left side of the top diode. Similarly the voltage at input b is the voltage at the left side of the bottom diode. The wire to the right of the top diode is connected to the wire to the right of the bottom diode, and to the output c; that's all one wire, so the voltage to the right of the two diodes and at the output must be the same. Suppose the inputs a and b are both $\top$. Then the output c has to be $\top$ because the diodes cannot maintain the voltage difference with $\top$ on the left and $\bot$ on the right. If a is $\top$ and b is $\bot$, c has to be $\top$ because we can't have $\top\,\bot$ across the a diode, but we can have $\bot\,\top$ across the b diode. Similarly if a is $\bot$ and b is $\top$, c has to be $\top$. Now suppose both a and b are $\bot$. The diodes allow the output c to be either $\top$ or $\bot$. If c is $\top$, there is a voltage difference across the resistor, so there will be an electron flow from the electron source marked $\bot$ through the resistor to c until c is $\bot$. (That takes about 10^{-10} seconds, which is slower than through a conductor, but still superfast for humans.) If c is $\bot$, there is no voltage difference across the resistor, and so no electron flow through the resistor.

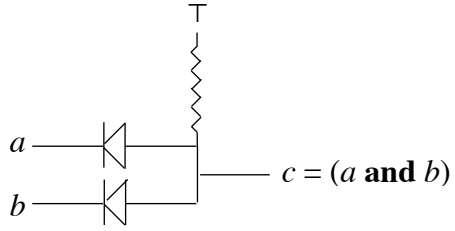
The water analogy may be helpful for understanding the operation of an **or** gate. Since electrical engineers like to put $\top$ at the top and $\bot$ at the bottom of a picture, as in the **or** gate above, and since water flows downhill, we'll make the flow of water analogous to the flow of holes. Imagine that water is pumped into the pipes at the a and b inputs. The diodes are one-way gates that allow the water to pass through to pipe c. The resistor is a pipe with a kink in it, so water goes through the resistor pipe slowly, dripping out the bottom. Water that drips out the resistor is replenished from the sources at a and b, so pipe c stays full of water. If we remove the source of water at b, and let pipe b drain empty, pipe c will remain full; any water dripping out the resistor pipe is replenished from a. It is possible for c to be full and b to be empty because the diode does not allow water to flow from c to b. Now suppose we also remove the source of water at a, and let pipe a drain empty too. Pipe c loses water though the resistor until it is empty; it is not replenished from a or b.

The previous picture is an electrical picture, showing what really happens to the voltage levels. After we make the binary abstraction, the circuit design symbol of an **or** gate is:



An **or** gate can have more than two inputs. The output is $\top$ if one or more inputs are $\top$, and $\bot$ if all inputs are $\bot$. The implementation uses one diode per input.

The next gate we present is the **and** gate. It has two inputs a and b, and one output c. If both input a and input b are $\top$, output c is $\top$. If either a or b or both are $\bot$, the output is $\bot$. We can build an **and** gate from two diodes and a resistor.



Suppose the inputs a and b are both $\top$. The diodes allow the output c to be either $\top$ or $\bot$. If c is $\bot$, there is a voltage difference across the resistor, so there will be an electron flow from c through the resistor to the electron sink marked $\top$ until c is $\top$. If c is $\top$, there is no voltage difference across the resistor, and so no electron flow through the resistor. If either or both of the inputs becomes $\bot$, then electrons flow from those input(s) to c until c is $\bot$.

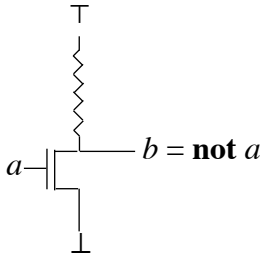
Electrons will then flow from c (slowly) through the resistor to $\top$, but they are immediately replenished from the input(s) that are $\bot$.

In the water analogy, if water is pumped into the a and b pipes, it fills those pipes, but has nowhere to go since it is stopped by the diodes, which are one-way gates the wrong way. So water drips through the resistor until c is full. If either or both of a and b is emptied, the water in c drains out through a and/or b (whichever is empty). Water continues to drip into c from the resistor, but it is immediately drained out through a and/or b, and so c remains empty.

The binary abstraction picture of an **and** gate is:

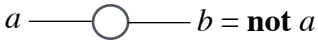
An **and** gate can have more than two inputs. The output is $\top$ if all inputs are $\top$, and $\bot$ if one or more inputs are $\bot$. The implementation uses one diode per input.

The simplest gate is the **not** gate, with one input a and one output b. If the input is $\top$, the output is $\bot$; if the input is $\bot$, the output is $\top$. The output is <u>not</u> what the input is. It can be built from a resistor and a transistor:

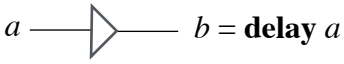


When input a is $\top$, the path between $\bot$ and b is conducting. Since the voltage is the same all along a conducting path, the output b is $\bot$. Some of the electrons at b will move through the resistor to $\top$, but they are immediately replaced from $\bot$, and the voltage at b stays $\bot$. When input a is $\bot$, the path between $\bot$ and b is not conducting. If b is $\bot$, there is a voltage difference across the resistor, so there will be a (slow 10^{-10} seconds) electron flow from b through the resistor to the electron destination $\top$ until b is $\top$.

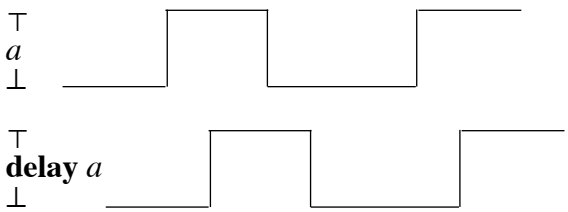
After we make the binary abstraction, the circuit design symbol of a **not** gate is a circle:



The last gate we present is the **delay**. Its circuit design symbol looks like this:

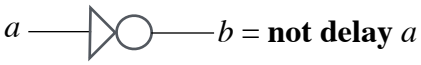


If the input a to a delay is represented by a voltage versus time diagram, then the output **delay** a is the same diagram but shifted to the right by the amount of the delay.



A delay can be built by putting an even number of **not** gates in sequence. For more delay use more **not** gates; for less delay use fewer **not** gates.

Gates can be combined. For example, a **delay** followed by a **not** looks like this:



Even if the input to a digital circuit starts off right on the $\top$ and $\bot$ voltage levels, changing levels with beautifully square changes, after going through a few **and** and **or** gates, the signal becomes degraded; the changes are less square and the levels may not be exactly $\top$ and $\bot$. Fortunately, a **not** gate improves the signal in addition to negating it. The output of a **not** gate is right on $\top$ or $\bot$ even if the input was a little off. A **delay** gate, which is made from **not** gates, also improves the signal.

There are 4 different gates with one binary input and one binary output. Two of them do not pay attention to their input, giving a constant output. They are represented by $\top$ and $\bot$. One of them is the identity function, whose output is always the same as the input. If the output is essentially instant, the identity function is represented by a wire. If the output is delayed from the input, the identity function is represented by the **delay** gate. And finally there is the **not** gate, whose output is always the opposite of the input.

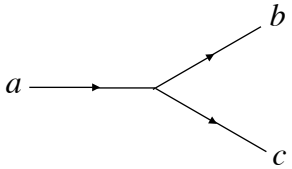
There are 16 different gates with 2 binary inputs and 1 binary output. Six of them do not pay attention to both inputs, so they are of no interest. Two of them are the **and** and **or** gates, which we have presented. The other 8 are useful, and a thorough presentation of binary algebra should present them. But we stop here. Binary algebra is very useful for circuit design, and if you want to read about it, you might try from Boolean Algebra to Unified Algebra and also Unified Algebra.

Information Flow

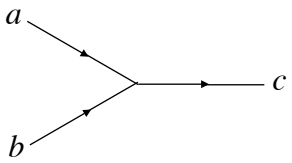
We have talked about electron flow, and we have talked about hole flow, which is the opposite direction to electron flow. We have talked about water flow, which we used as an analogy to hole flow. We could just as well use water flow as an analogy to electron flow if we draw the pictures the other way up, with $\bot$ at the top and $\top$ at the bottom.

Information flow is the direction from inputs to outputs. It is not related in any way to the direction of electrons, holes, or water; it may happen to be the same direction, or it may happen to be the opposite direction. A wire is an input if we choose its voltage, choosing either $\top$ or $\bot$ as we wish. A wire is an output if we measure its voltage. We tend to draw inputs on the left side and outputs on the right side, but that is not what makes them inputs and outputs, and sometimes that's not the convenient way to draw them.

At the electrical level, wires can be joined together, making a single wire coming and going in various directions, all with the same voltage. We have seen this in the electrical pictures of the **or** gate and the **and** gate. It is possible for an information path to split into two or more paths, as in this picture, showing the direction of information flow by arrows.

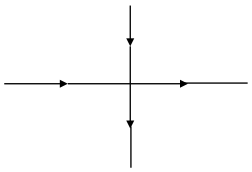


We can choose $\top$ or $\bot$ for input a, and then outputs b and c will both have the same value as a. But it is not possible for two information paths to join into one. We cannot have



If we choose different values for inputs a and b, then output c is in the impossible position of having to be both $\top$ and $\bot$. The only way to join two information paths is to use a gate. When we use an **and** gate or an **or** gate, the gate says how the output is determined by the inputs.

We sometimes need to show wires crossing each other, as in this picture:



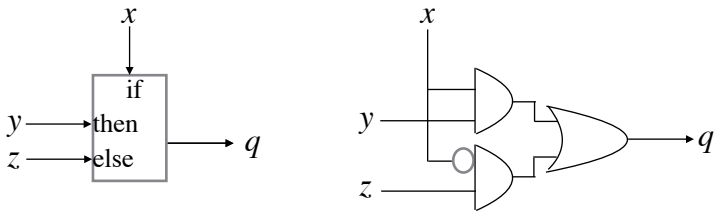
Wherever wires cross, it means that they are not connected to each other.

We now leave the electrical level behind; we make the binary abstraction. We stop talking about voltages, and we talk only about binary values $\top$ and $\bot$. We stop talking about electrons and holes, and talk only about information flow. We stop talking about wires, and talk instead about information paths, or just paths. The basic unit of information is a binary value, which is also called a "bit".

Multiplexer and Demultiplexer

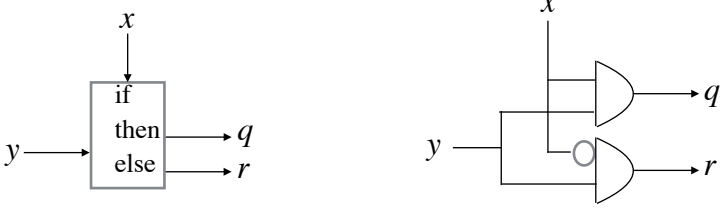
Each circuit has two pictures. One picture is a box, with inputs pointing to it, and outputs pointing away from it, and in the box there are words or symbols to identify the function of the circuit. The other picture shows what's in the box, so we can see how the circuit is built.

The first circuit we present is the multiplexer.



It has inputs x, y, and z, and output q. If $x{=}\top$ then $q{=}y$. If $x{=}\bot$ then $q{=}z$. To see that the **and**, **or**, and **not** gates on the right are a correct implementation, it is necessary to look at 8 cases. The inputs $x\ y\ z$ can be $\bot\bot\bot$, $\bot\bot\top$, $\bot\top\bot$, $\bot\top\top$, $\top\bot\bot$, $\top\bot\top$, $\top\top\bot$, and $\top\top\top$. You should check each of these cases to see that the output is correct.

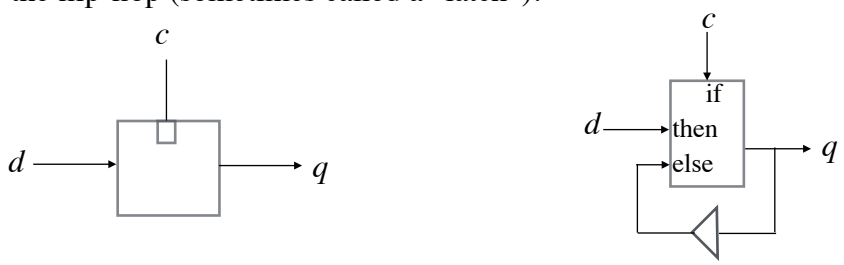
The next circuit is the demultiplexer.



It has inputs x and y, and outputs q and r. If $x{=}\top$ then $q{=}y$ and $r{=}\bot$. If $x{=}\bot$ then $r{=}y$ and $q{=}\bot$. To see that the **and** and **not** gates on the right are a correct implementation, it is necessary to look at 4 cases. You should check each of them.

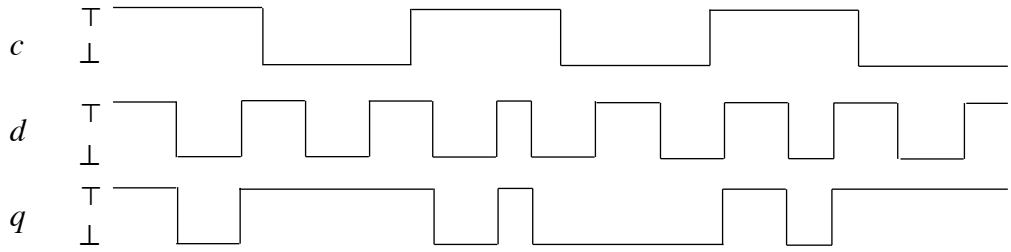
Flip-flop

The next circuit is the flip-flop (sometimes called a “latch”).



The inputs are d (for data) and c (for control. Textbooks often say that c is for clock because sometimes this input comes from the output of a clock circuit. But we should not name an input by where it may or may not come from). The output is q (for output, but o looks too much like zero, so q is used). A flip-flop behaves as follows. If $c=\top$, then $q=d$. If $c=\bot$, q remains as it was. The circuit on the right has an interesting feature called “feedback”. The output q goes through a delay and then is fed back into an input of the multiplexer. When $c=\bot$, the multiplexer's “else” input is what q was a moment earlier; that's how q remains what it was. Without the delay, there would be no constraint on q when $c=\bot$; the multiplexer would say $q=q$, which is always true no matter what q is. So a delay is necessary: when $c=\bot$, q = **delay** q. However, a multiplexer is not instant, and its delay is sufficient. A delay is implemented as an even number of **not** gates, and 0 is an even number. The delay in the flip-flop circuit represents the delay already present in the multiplexer, without adding more.

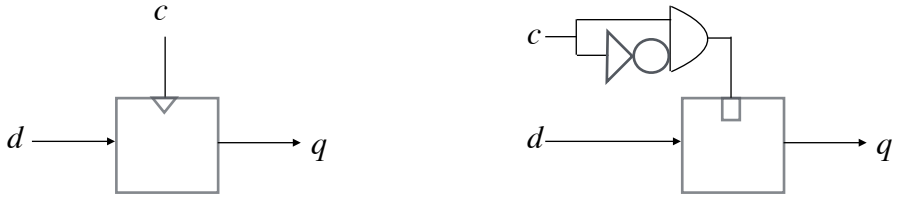
Here is a binary versus time diagram of a flip-flop.



Both c and d are inputs, so we can choose to make them whatever we want. They are chosen here to illustrate that while c is $\top$, q is identical to d, and while c is $\bot$, q remains as it was when c changed from $\top$ to $\bot$. Feedback is characteristic of memory. A flip-flop is one bit of memory; it remembers what d was the last time c was $\top$.

Edge-trigger

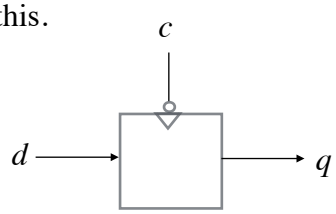
The flip-flop we have just seen is called “sensitive” because the output equals the d input all the time that c is $\top$. Next we look at a mechanism called “edge-triggering” that keeps the output constant except when c changes. A rising edge-trigger keeps the output constant except when c changes from $\bot$ to $\top$. It looks like this.

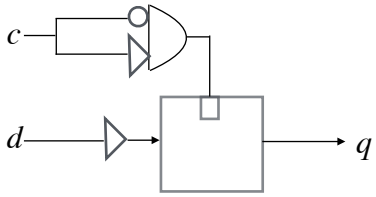


The box picture on the left has a little triangle at the control input to say it is rising edge-

triggered. We build it from a sensitive flip-flop (with a little square at the control input) and some edge-triggering circuitry. The only time the control of the sensitive flip-flop is $\top$ is when c is $\top$ and it was just previously $\bot$ (c **and not delay** c). We need the delay to be long enough to detect a rising edge, but no longer.

A falling edge-trigger keeps the output constant except when c changes from $\top$ to $\bot$. It looks like this.

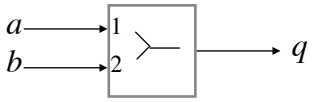


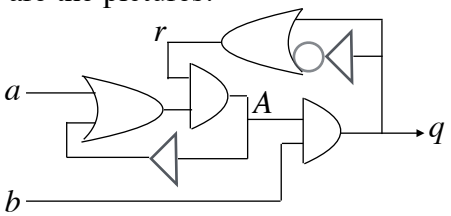


The box picture has a little circle and triangle at the control input to say it is falling edge-triggered. In the right diagram we see that the only time the control of the flip-flop is $\top$ is when c is $\bot$ and it was just previously $\top$ (**not** c **and delay** c). The delay from d into the flip-flop's data input is so that the output will be the value of d just before c changed from $\top$ to $\bot$.

Merge

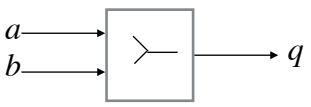
The 1-2-merge has inputs a and b and output q. When pulses arrive on a and b in that order (a first, b second), or simultaneously, it outputs a pulse, and then resets itself ready to go again. The output is $\bot$ at all other times. Here are the pictures.

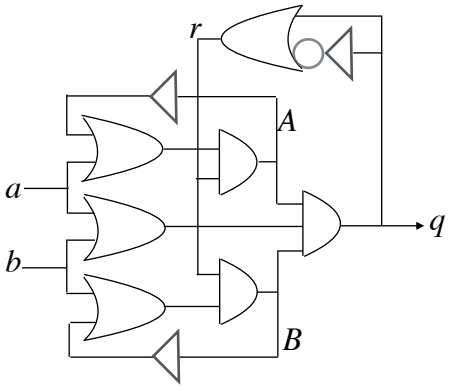




For the purpose of explanation, two internal paths have been labeled. Path r is $\top$ when q is $\top$ or was just previously $\bot$, and r is $\bot$ when q is $\bot$ and was just previously $\top$. In other words, r is $\bot$ at the falling edge of q, when an output pulse has just occurred (the r circuitry is an edge-trigger). The delay between q and r must be just enough to detect the edge. Suppose an output pulse has not just occurred, and q has not just changed from $\top$ to $\bot$. Then r is $\top$. So A is $\top$ if input a is $\top$ or was previously $\top$. As long as b is $\bot$, q will be $\bot$; when b becomes $\top$, q will become $\top$; when b becomes $\bot$, q will become $\bot$. If there has already been a pulse on a, or at least a pulse on a has started, then a pulse on b becomes a pulse on q. At the end of the pulse on q, r momentarily becomes $\bot$, so A becomes $\bot$ until the next pulse on a. The delay after A represents the delay already present in the **and** and **or** gates, without adding more.

Next we look at a symmetric merge. It also has inputs a and b and output q. When pulses arrive on a and b in either order or simultaneously, it outputs a pulse, and then resets itself ready to go again.





As before, r is $\perp$ at the falling edge of q, when an output pulse has just occurred. As before, A is $\top$ if input a is $\top$ or was $\top$ since the end of the last pulse on q. Symmetrically, B is $\top$ if input b is $\top$ or was $\top$ since the end of the last pulse on q. Output q coincides with whichever pulse on a or b occurs second. At the end of the pulse on q, r momentarily becomes $\perp$, so A becomes $\perp$ until the next pulse on a, and B becomes $\perp$ until the next pulse on b.

The way the merges function has been explained by naming internal paths r, A, and B. But the merges are designed the other way around. First, we decide that it would be helpful to know whether there has been a pulse on input a since the last pulse on output q, and similarly whether there has been a pulse on input b since the last pulse on output q. That is, we decide that there will be internal paths A and B before we know what gates there will be. To implement A and B, we decide that it will be helpful to know when is the falling edge of q; that is, we decide to have internal path r. And finally, we decide what gates are needed.

Binary Number Representation

We are familiar with the natural numbers expressed in decimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on. In binary, it's much the same, except that there are only two digits instead of ten. The natural numbers expressed in binary are: 0, 1, 10, 11, 100, 101, 110, 111, 1000, and so on. The sequence of binary digits 1101 represents the number expressed in decimal as

$$1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 1\times8 + 1\times4 + 0\times2 + 1\times1 = 13$$

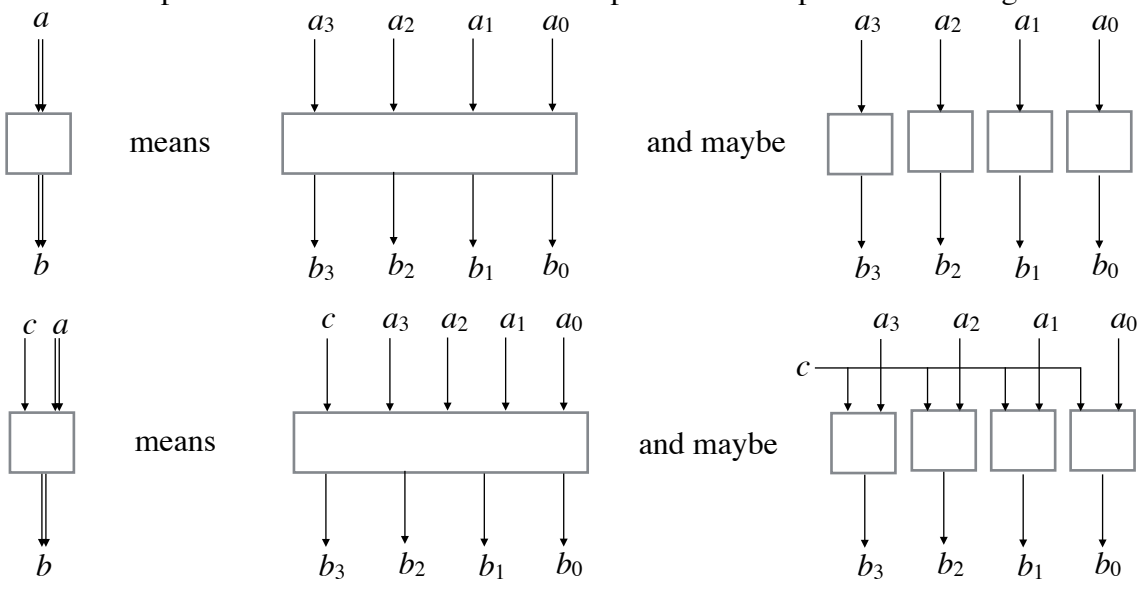
The exponents (powers of 2) are 0, 1, 2, ... for the digits going from right to left. We therefore number the digits that same way: in 1101, digit number 0 is the rightmost 1, digit number 1 is 0, digit number 2 is 1, and digit number 3 is the leftmost 1. If $a = 1101$, then

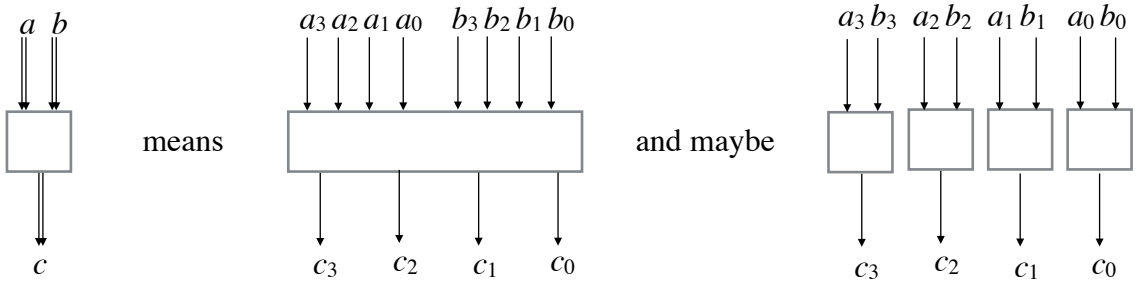
$$a = a_3\, a_2\, a_1\, a_0 = 1101$$

We use binary value $\perp$ to represent binary digit 0, and binary value $\top$ to represent binary digit 1. The word “bit” has already been introduced as the basic unit of information, the basic unit of memory, and as a synonym for “binary value”. It is also a contraction of “binary digit”. Fortunately, these four meanings are compatible, not conflicting.

Multipath Notation

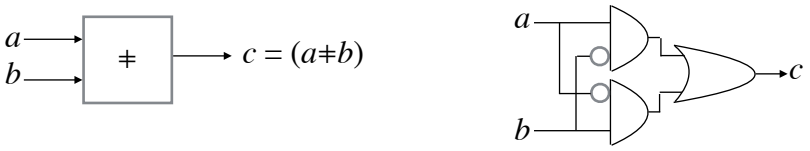
We use a double arrow to mean several paths, without being specific about how many. In each row below, the picture on the left means the picture in the middle (as a 4-bit example). Sometimes the picture in the middle can be decomposed into the picture on the right.





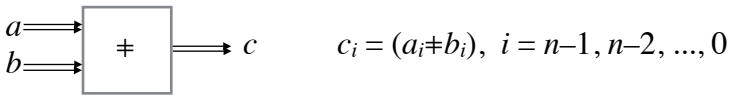
Bit Comparison

We now compare two binary inputs to see if they are unequal.



If a and b are unequal, then $c=\top$; if they are equal, then $c=\bot$. (This circuit is sometimes called "exclusive or", sometimes called "parity", and sometimes called "addition modulo 2".)

We might want to compare many pairs of bits to see which pairs are unequal. The box picture is:



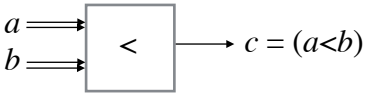
The a input is really n inputs a_{n-1} , a_{n-2} , ..., a_2 , a_1 , a_0 . Similarly for the b input and the c output. This comparison is called "bitwise"; it means $c_0 = (a_0 \neq b_0)$, $c_1 = (a_1 \neq b_1)$, and so on. It is implemented by putting n single-bit $\neq$ circuits beside each other.

Number Comparison

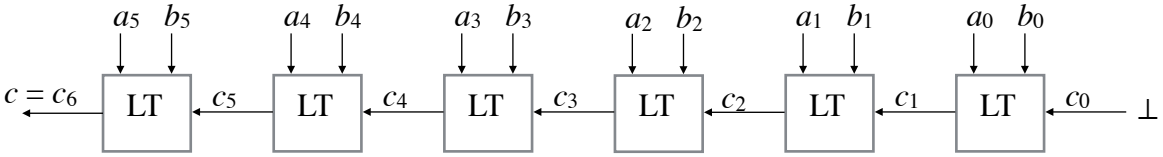
More often, we want to compare two n-bit numbers to see if they are unequal; we want a single bit of result, not n bits. Here is the box picture and its implementation.

If a and b differ in one or more bit positions, then a and b are unequal.

Numbers can also be compared to see which is smaller and which larger. Here is the box picture.

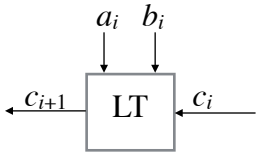


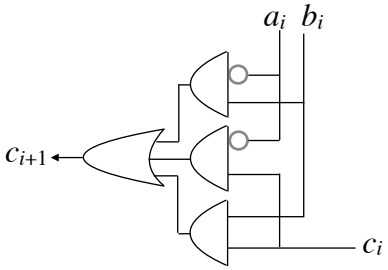
It can be implemented by a sequence of boxes, called LT boxes (for Less Than). Each box gets one bit of a input and one bit of b input, and they are connected as in the following 6-bit example.



Bit c_i has the meaning $a_{i-1..0} < b_{i-1..0}$. In other words, $c_i = \top$ if and only if the part of a to the right of c_i is less than the part of b to the right of c_i . And therefore the output c is c_6 . Each

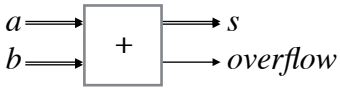
LT box gives its output c_{i+1} the value $\top$ in the following three cases: a_i=0 and b_i=1, because then the part of a to the right of c_{i+1} is less than the part of b to the right of c_{i+1}, regardless of the inputs further to the right; a_i=0 and c_i=$\top$, because then, either b_i=1 as before, or b_i=0 and the relation is determined by c_i=$\top$; b_i=1 and c_i=$\top$, because then, either a_i=0 as before, or a_i=1 and the relation is determined by c_i=$\top$. Here is an LT box and its implementation.





Arithmetic

We now present a circuit that adds two binary numbers. Its box picture is



Inputs a and b and output s could be 32 paths each, for an adder that adds two 32-bit numbers producing a 32-bit sum. When you add two 32-bit numbers, the sum may not be representable in 32 bits; the *overflow* result is $\bot$ if the sum is representable, and $\top$ if it is not.

To implement an adder, we need to know how to add. Here is a 6-bit example, in which we add 011011 and 011010.

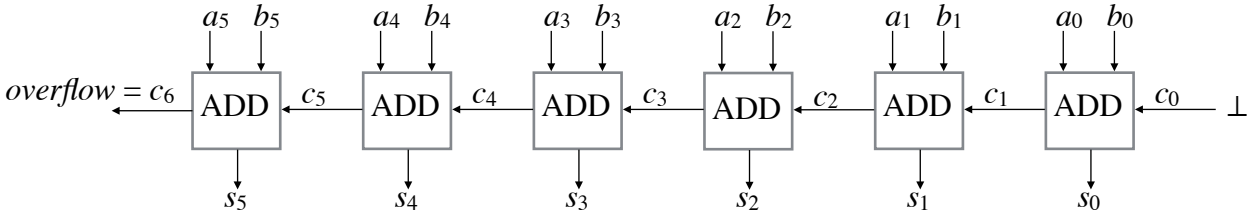
bit position	5	4	3	2	1	0
a =	0	1	1	0	1	1
b =	0	1	1	0	1	0
	1	1		1		
s =	1	1	0	1	0	1

Starting from the right end, we add a_0+b_0 = 1+0 = 1 = s_0. Next we add a_1+b_1 = 1+1 = 10 so s_1=0 and 1 is carried to bit position 2. Next a_2+b_2+(the carry) = 0+0+1 = 1 = s_2. Next a_3+b_3 = 1+1 = 10 so s_3=0 and 1 is carried to bit position 4. Next a_4+b_4+(the carry) = 1+1+1 = 11 so s_2=1 and 1 is carried to bit position 5. Finally a_5+b_5+(the carry) = 0+0+1 = 1 = s_5. The sum is representable in 6 bits, so *overflow*= $\bot$.

Here is the same example again, but this time when there is no carry, we say the carry is 0. To make all bit positions the same, the carry in position 0 is 0. Calling the carry c, we have

bit position	5	4	3	2	1	0
a =	0	1	1	0	1	1
b =	0	1	1	0	1	0
c =	0	1	1	0	1	0	0
s =	1	1	0	1	0	1

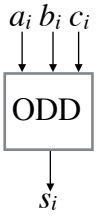
The carry into bit position 6 is the overflow bit (no overflow in this example). In each bit position i, there is a sum of 3 bits: a_i+b_i+c_i, producing 2 bits of result: c_{i+1} s_i. So a 6-bit adder circuit looks like this:

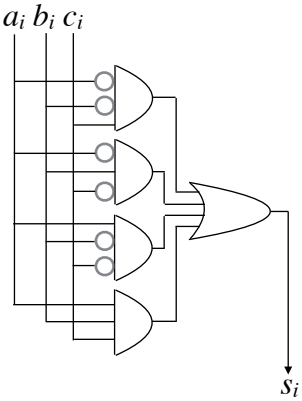


Each ADD box represents one column of the addition. The pattern is easily generalized to any

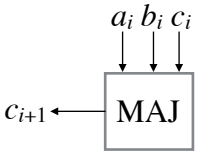
number of bit positions.

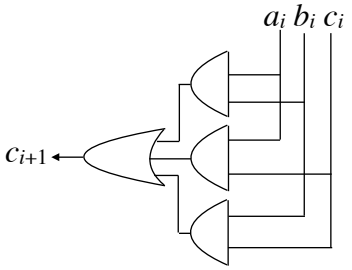
Now we have to design an ADD box; it has to add 3 bits, with a 2-bit result. The right bit of the result is 1 if there are an odd number of 1's among the inputs. In other words, s_i is 1 if $a_i\ b_i\ c_i$ is 001, 010, 100, or 111. Let's make a circuit for the right bit of the result.



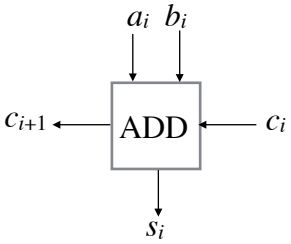


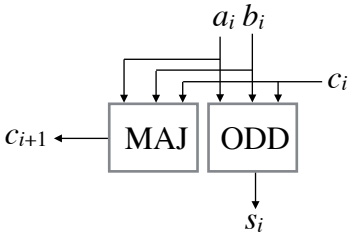
The left bit of the result is 1 if there are two or more 1's among the inputs. In other words, c_{i+1} is 1 if $a_i\ b_i\ c_i$ is 011, 101, 110, or 111. Let's make a circuit for the left bit of the result, and we'll call it MAJ for "majority".





We put the left and right bits together to make the ADD box.





And that completes the adder.

Many people were taught to subtract in a poor way. For example, in decimal, to subtract 3456 from 13002, you may have been taught

position	4	3	2	1	0
	0	12			
		~~2~~	9	9	12
a =	1	~~3~~	~~0~~	~~0~~	~~2~~
b =	0	3	4	5	6
d =		9	5	4	6

In position 0, you can't take 6 from 2, so you borrow from position 1, but you can't borrow from 0, so you borrow from position 2, but again you can't borrow from 0, so you borrow from position 3. That reduces the 3 in position 3 to 2, changes the 0 in position 2 to 9, changes the 0 in position 1 to 9, and finally changes the 2 in position 0 to 12. All of that just to get the rightmost digit of the answer. We want to make a box for each position, and connect each position to its immediate neighbors, just as we did for addition. We don't want to have to connect each position to all other positions. So here is a better way to subtract.

```
position     4 3 2 1 0
a =          1 3 0 0 2
c =        0 1 1 1 1 0
b =          0 3 4 5 6
d =          0 9 5 4 6
```

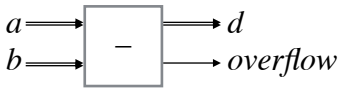
We are subtracting a–b and getting the difference d. We don't borrow, we carry, and the carries are written in between the two operands. To make all positions the same, we carry 0 into position 0. We add the b digit and the carry digit, and subtract that from the a digit. In position 0, we can't subtract 6+0 from 2, so we carry 1 to position 1. Now we subtract 6+0 from 12 (you can see the 12 inside the oval) and get 6 as the rightmost answer digit. We didn't have to look further left than the next position. In position 1, we can't subtract 5+1 from 0, so we carry 1 to position 2. Now we subtract 5+1 from 10 (to see the 10, slide the oval left one position) and get 4 as the answer digit in position 1. In position 2, we can't subtract 4+1 from 0, so we carry 1 to position 3. Now we subtract 4+1 from 10 and get 5 as the answer digit in position 2. In position 3, we can't subtract 3+1 from 3, so we carry 1 to position 4. Now we subtract 3+1 from 13 and get 9 as the answer digit in position 3. In position 4, we can subtract 0+1 from 1 and get 0 as the answer digit in position 4. The carry to position 6, which is 0, says that b is less than or equal to a, and the answer is correct.

Now in 6-bit binary, we subtract 011010 from 100101.

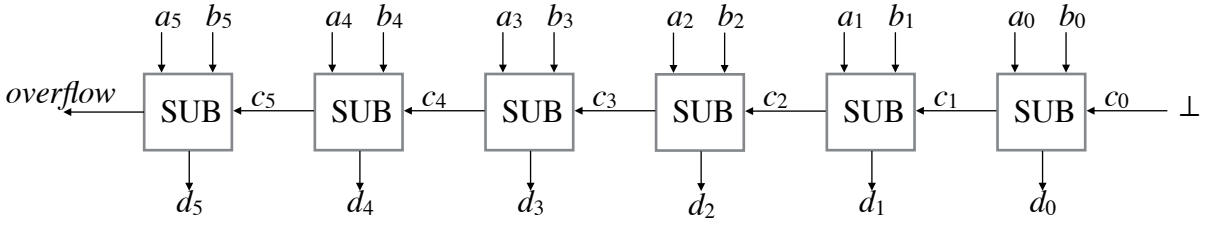
```
position     5 4 3 2 1 0
a =          1 0 0 1 0 1
c =        0 1 1 0 1 0 0
b =          0 1 1 0 1 0
d =          0 0 1 0 1 1
```

The carry into position 0 is 0. In position 0, we subtract 0+0 from 1 and get 1 as the answer bit. In position 1, we can't subtract 1+0 from 0, so we carry 1 to position 2. Now we subtract 1+0 from 10 (you can see the 10 inside the oval) and get 1. In position 2, we subtract 0+1 from 1 and get 0. In position 3, we can't subtract 1+0 from 0, so we carry 1 to position 4. Now we subtract 1+0 from 10 (to see the 10, slide the oval left one position) and get 1. In position 4, we can't subtract 1+1 from 0, so we carry 1 to position 5. Now we subtract 1+1 from 10 and get 0. In position 5, we subtract 0+1 from 1 and get 0. The carry to position 6, which is 0, says that b is less than or equal to a, and the answer is correct.

Here is the box picture of a subtractor.



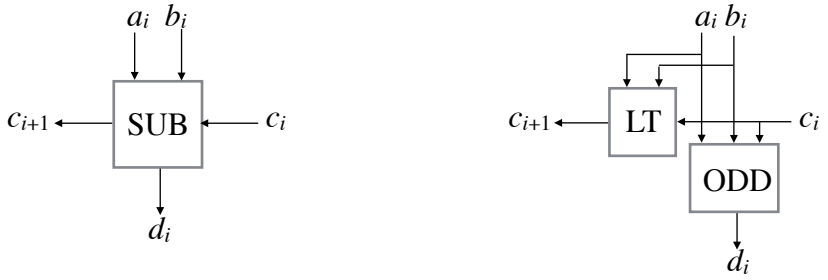
The word “overflow” is not the right name for this output, but it is the name that is always used. This output is 0 when $a \geq b$ and d is the desired difference. This output is 1 when $a<b$ and d is not the desired difference. Here is the implementation of a 6-bit subtractor.



Each SUB box represents one column of the subtraction. The pattern is easily generalized to any number of bit positions.

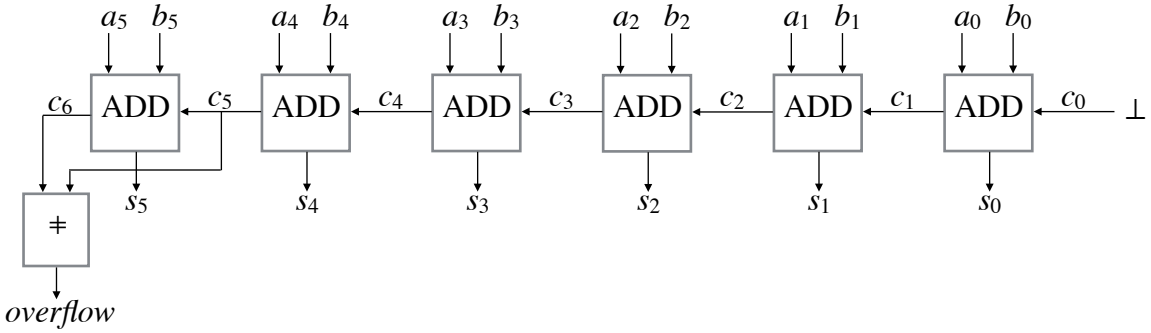
Now we have to design a SUB box. The right bit of the result is 1 if there are an odd

number of 1's among the inputs, exactly the same as for addition. The left bit of the result is given by the LT box that we designed earlier.



You should check all 8 combinations of inputs for the SUB box to make sure they have the right outputs. And that completes the subtractor.

We have looked at addition and subtraction of natural numbers. We can easily adapt these circuits to add and subtract integers: positive, zero, and negative. The standard representation of integers is called "two's complement". The easiest definition of two's complement is the representation that uses the same addition and subtraction algorithms as natural numbers. The only alteration is overflow. Here is 6-bit addition.

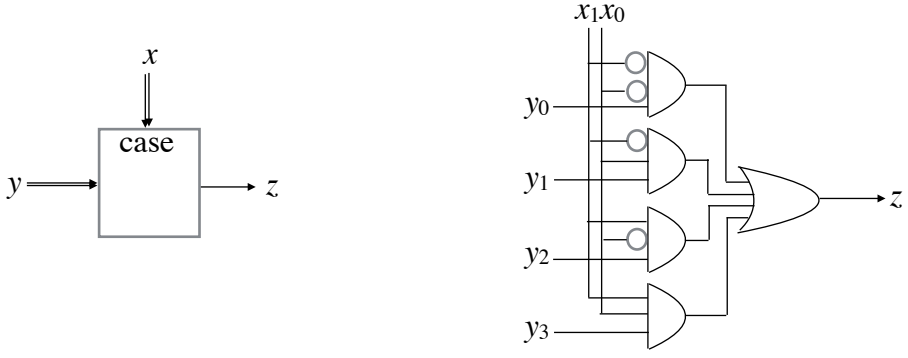


For two's complement subtraction, make exactly the same change to the overflow output.

We have looked at addition and subtraction of integers. We will look at multiplication of integers later. We will not look at division, which is harder. And we will not look at floating-point arithmetic. In this short course, we show how basic arithmetic can be done in order to remove the magic. Fast arithmetic is so important that an immense amount of work has gone into finding better, faster ways of doing arithmetic than are presented here. If you are interested, consult a textbook devoted to arithmetic. And you might like to look at a New Representation of the Rational Numbers for Fast Easy Arithmetic.

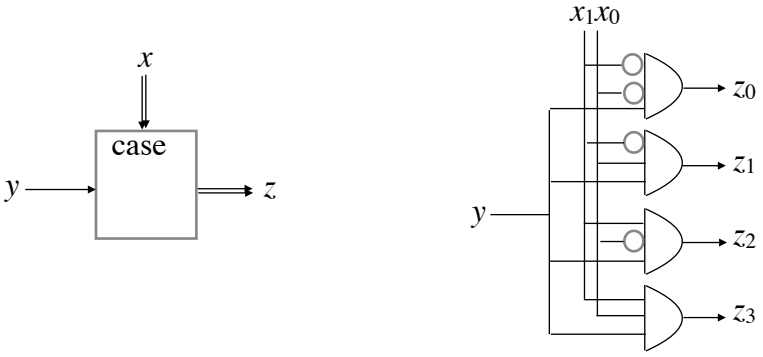
General Multiplexer and Demultiplexer

We have already seen the circuit for a multiplexer. We now generalize it: instead of a 1-bit input x that chooses between 2 other inputs y and z, this generalization has n bits of input x that choose among 2^n other inputs y, where n is any natural number. Here is the n=2 version.



In the implementation on the right, x is 2 paths, and the x inputs are a binary representation of one of the numbers 0, 1, 2, and 3. If x represents 0, then $z=y_0$. If x represents 1, then $z=y_1$. And so on. In general, $z=y_x$.

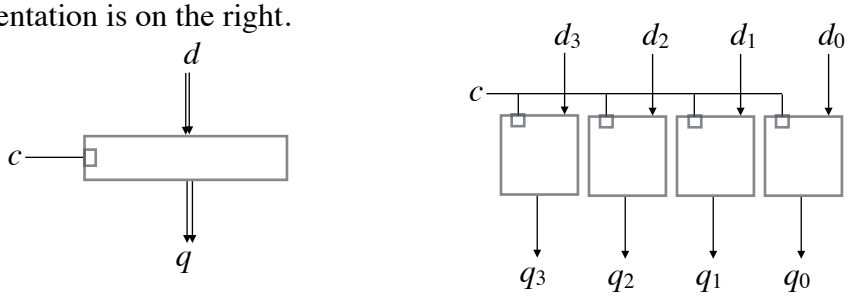
We make the same generalization of a demultiplexer.



If x represents 0, then $z_0=y$; if x doesn't represent 0, then $z_0=\perp$. If x represents 1, then $z_1=y$; if x doesn't represent 1, then $z_1=\perp$. And so on. In general, for any number of inputs, $z_x=y$, and $z_i=\perp$ for $i\neq x$.

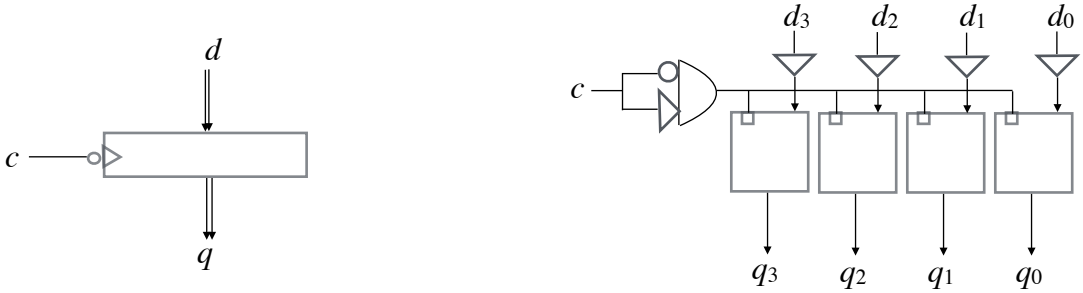
Register

A register is a group of flip-flops connected by a common control. Its box picture is on the left, and its implementation is on the right.



This implementation uses 4 flip-flops to make a 4-bit register, but it should be clear how to make a register with any other number of bits. An n-bit register holds n bits of information. That just means that there are n bits of output q from the n flip-flops. This information could be an n-bit number, or the codes for some characters, or a computer instruction, or just n bits. To change the information in a register, put the new information on the data input paths d, and then send a pulse on the control path c.

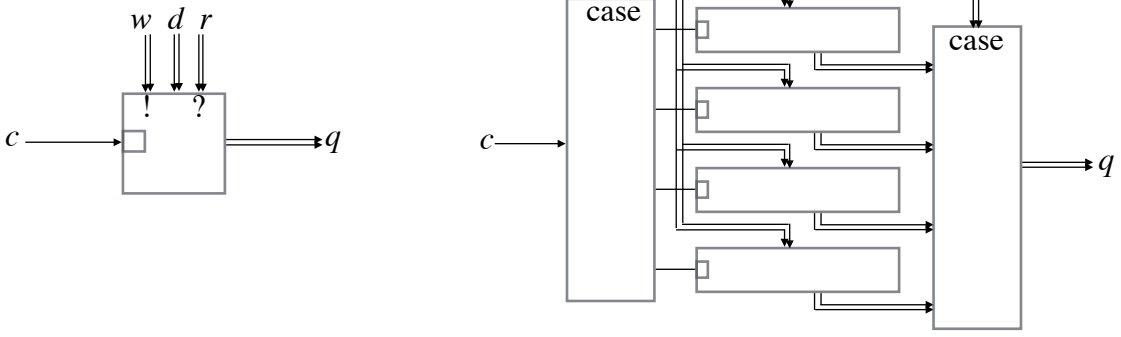
In the sensitive register pictured above (square at the control input), the information in the register (the outputs from the register) may change at the rising edge of the control pulse, and continue changing according to any changes in the data inputs for as long as the control pulse is up, and then become unchanging at the falling edge of the control pulse. To create an edge-triggered register, we do not need to use edge-triggered flip-flops. The edge-triggering can be done once for all flip-flops. Here is a falling edge-triggered register.



In the falling edge-triggered register (circle and triangle at the control input c), the information in the register doesn't change until the falling edge of the control pulse. At the falling edge, the bits of data input become the bits in the register. After that, the content of the register is again unchanging. In a rising edge-triggered register (just a triangle at the control input c), the bits of data input become the bits in the register at the rising edge of the control pulse. After that, the content of the register is unchanging.

Memory

A memory (or random access memory, or RAM) is a collection of registers connected by common data paths. Its box picture is on the left, and its implementation is on the right.



For the memory of a computer, each register is typically 8 bits, which is called a “byte”. A 4 gigabyte memory has 2^{32} registers, rather than the 4 shown here. The registers are numbered 0, 1, 2, 3, ... , and a register's number is called its “address”. A 4 gigabyte memory has 32 writing address inputs (w), 32 reading address inputs (r), 8 data inputs (d), 8 outputs (q), and 1 control input (c).

To write something into a register, put the address of the register you want to write in input w, and at the same time, put the data you want to write in input d; then put a pulse in input c. The data goes to all registers, but the demultiplexer on the left side of the implementation picture routes the pulse to the correct register, so only that register changes its contents. To read the contents of a register from the memory, put the address of the register you want to read in input r; the contents of that register is output q. On the right side of the implementation picture, the multiplexer is really several multiplexers: one for each bit in a register. The address r goes to each multiplexer. Multiplexer 0 takes all the bit 0s from all the registers, and produces bit 0 of the result. And so on for the other bits.

The memory pictured is sensitive (square at the control input). To make an edge-triggered memory, we do not need to make each register edge-triggered; we just need to add the edge-triggering once at the c input.

(Aside: The previous diagram shows how a memory works. But for a computer, 8 bits wide and 2^{32} bits long is not a practical layout for a VLSI circuit. To be practical, the 32-bit address is divided into halves. The first 16 bits are used to address an x-coordinate, and the last 16 bits are used to address a y-coordinate, in a square array. Each byte of memory sits at the intersection of an x-coordinate and a y-coordinate. This gives a memory a square shape. End of Aside)

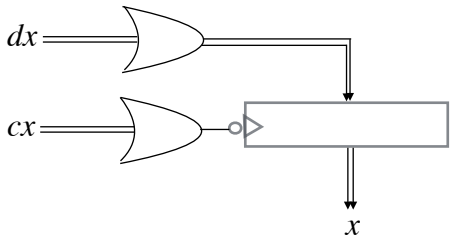
Memory that you buy differs from the memory presented here in one respect: it has only one address input, not two as shown here. Its one address goes to both the demultiplexer for writing and to the multiplexer(s) for reading. The way computers have been built up to now, an

instruction being executed might read from memory, or it might write to memory, but not both at the same time. The memory shown here, with two addresses, allows writing at one address to happen at the same time as reading at another address. In fact, it allows writing and reading to happen at the same address at the same time, but the result is unpredictable, so that's not a good idea.

Programs

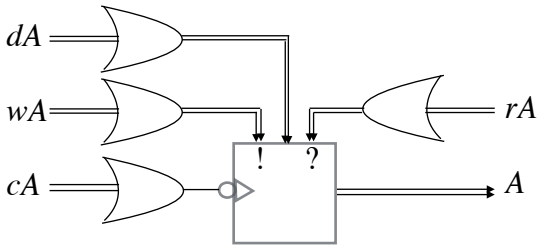
We now have all we need to show how programs become circuits. The material presented here is one of the circuit design methods in the paper High-Level Circuit Design.

Write a program in any imperative programming language, such as C, Java, Python, or any other language of your choice. Each variable in the program becomes a falling edge-triggered register with enough bits to store the values that can be assigned to the variable. For example, if we have variable x in the program that can be assigned 32-bit integers, then in the circuit we have



Just before the control input, there is an **or** gate; its inputs cx come from all the places where x is being assigned a value. The **or** gate going into the register's data input is really several **or** gates: one for each bit in the register. So in this example, it is 32 **or** gates. The inputs dx to these **or** gates come from all the places where x is being assigned a value, just like the control inputs cx. If there are 4 such places, then each of these 32 **or** gates has 4 inputs. The outputs x go to all places that require the value of variable x. If there are 5 such places, then each of the 32 outputs splits into 5 directions.

For each array in the program, there is a falling edge-triggered memory. If there are 3 arrays in the program, there will be three memories. A memory has just enough registers for the number of elements in its array; if the array size is dynamic, the memory must have enough registers for the maximum size of its array. Each register in the memory has the number of bits necessary to store the value of an array element. For example, if the program has an array A, then the circuit has:

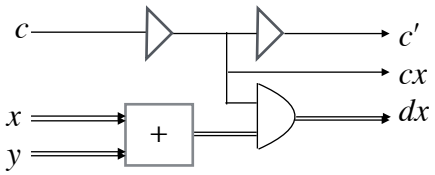


The inputs labeled dA (data for A), wA (writing address for A), and cA (control for A) come from all places where an element of array A is being assigned a value. The inputs labeled rA (reading address for A) come from all places that use the value of an element of array A. The outputs labeled A go to all places that use the value of an element of array A. The cA inputs go into a single **or** gate. The wA and rA inputs go into as many **or** gates as there are address bits. The dA inputs go into as many **or** gates as there are data bits.

For each statement in the program we have a circuit with control input c and output c'.

A pulse on c starts the circuit working, and when it is finished, it sends out a pulse on c'.

In a program, a variable or array element gets a value by means of an assignment. To assign to variable x the sum of the values of x and y, the syntax may be $x{:=}x{+}y$ or $x{=}x{+}y$; or something else, depending on the programming language. This assignment is compiled to the following circuit.

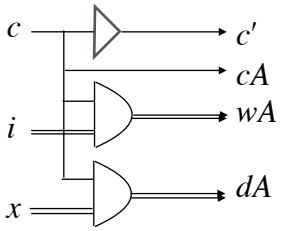


The inputs marked x and y come from the registers for variables x and y. If x and y are 32 bit variables, there are 32 paths for each of them. They go through an adder that has 32 bits of output (we are ignoring the *overflow* output). A pulse arriving on c goes through a delay that is just long enough to allow the adder to add the current values of variables x and y. The **and** gate in the picture is really 32 **and** gates, each of which has one input from the sum and the **delay** c input. The 32 outputs from the **and** gates become the data input dx for variable x. Meanwhile, the pulse from **delay** c goes to the control input for variable x, to give x its new value. Also, **delay** c goes through another delay to provide a small separation between this assignment and the next operation, becoming a pulse on c' to say "all done this assignment". When this assignment is not being executed, **delay** $c = \perp$, so the outputs dx of the 32 **and** gates are all $\perp$, so they do not interfere with any other assignments to x that may be taking place.

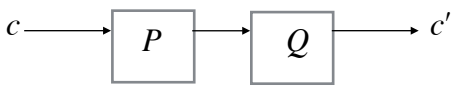
Our example assignment statement has an addition in it, and the circuit has an adder in it. So the circuit for the whole program will have as many adders as there are additions appearing in the program. Similarly for subtractors, multipliers, and so on. Alternatively, an adder or other circuit can be shared among several expressions (by means of the function call circuitry which we present later), at the programmer's discretion.

An expression may depend on an array element; if so, the reading address for that array element must be output from the expression circuit, conjoined with c, and routed to the memory for that array. There may be references to elements of several arrays, but there can be at most one array element reference per array in the expression. If you want more than one element of the same array, you must break up the expression. For example, $x{:=}\ A[i]{+}A[j]$ becomes $x{:=}\ A[i];\ x{:=}\ x{+}A[j]$ (or whatever syntax your language uses).

An array element assignment is a little more complicated. For example, $A[i]{:=}\ x$ produces the circuit



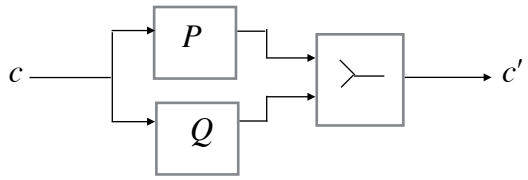
When a statement P is followed sequentially by a statement Q in a program, the circuit for P is connected to the circuit for Q by connecting the c' output of P to the c input of Q.



The pulse that P sends out to say it is done is the pulse that starts Q.

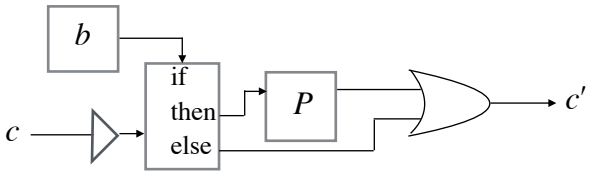
Most commonly used programming languages either do not have parallel composition, or have a parallel construct that is restricted in how it can be used and is not easy to use. If a statement P is to be executed in parallel with a statement Q, the circuits for P and Q are

connected like this:



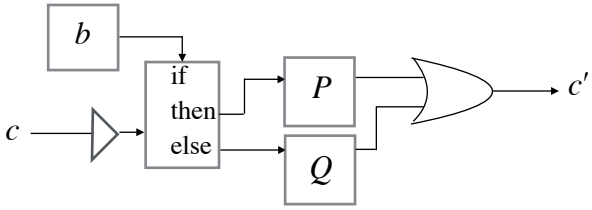
The control signal c starts both P and Q at the same time. They may finish at different times. Each sends a pulse on its c' output when it finishes. These pulses go to a merge box, which emits a pulse only when it has received input pulses on both its inputs. Simultaneous access to different variables or arrays poses no problem. Even for the same variable, simultaneous reads are no problem. But simultaneously reading and writing the same variable, or two simultaneous writes to the same variable, have unpredictable results, and is not recommended.

Every programming language has a conditional statement. It may look like **if** b **then** P or it may look like **if** (b) P or it may look some other way. The binary expression b is evaluated, and if b is true, statement P is executed, and if b is false, nothing more happens. It produces the circuit:

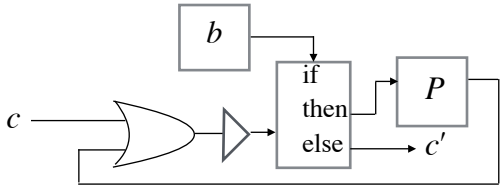


The box labeled b evaluates expression b, and has inputs that come from any variables and array elements used in expression b. The delay is just long enough to evaluate expression b. The control pulse c goes either to P and then out c', or directly out c'.

The conditional statement may have a two-tailed version that looks like **if** (b) P **else** Q, or like **if** b **then** P **else** Q, or perhaps some other syntax. The binary expression b is evaluated, and if b is true, statement P is executed, and if b is false, statement Q is executed. It produces the circuit:

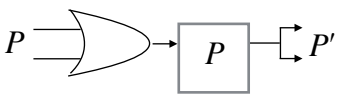


A loop may have a syntax like **while** b **do** P or like **while** (b) P or something else. Here is its circuit.

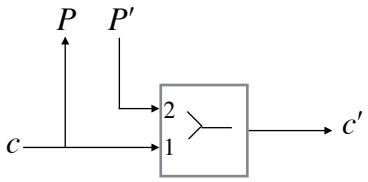


We can similarly construct a circuit for a loop whose exit condition comes at the end, and even for a loop that has several exits in the middle.

A procedure or void function or method is a unit of program that can be named, so that it can be called from several places, and return back to the place where it was called. Ignoring parameters for a moment, procedure P has this circuit:



The inputs labeled P come from all the places where P is called (the picture shows 2 inputs, but it could be any number). The outputs labeled P' go back to all the places where P is called (the picture shows 2 outputs, but it could be any number). The calling points each become



When a control pulse arrives at a call, it is sent to the called procedure to start the procedure. It also goes into the 1-input of a 1-2-merge box. A 1-2-merge box emits a pulse only when it receives input pulses in a specific order: first into the 1-input, then into the 2-input. When the procedure finishes, it sends a pulse back to all calling points. When the pulse arrives at this calling point, it goes into the 2-input, causing the 1-2-merge box to emit a pulse. All those pulses that arrive at calling points that did not call the procedure go into the 2-input of the 1-2-merge box there, but there was no previous pulse in the 1-input there, so no pulse is emitted by those 1-2-merge boxes.

This implementation does not work for recursive calls in general, but it does work for tail-recursive calls. An internal (non-tail) recursive call requires an up-down-counter, but we don't pursue that here. The present implementation also doesn't work if parallel calling points call the procedure at the same time or at overlapping times, so that should be avoided.

A parameter declaration can be treated exactly as though it were introducing a local variable instead of a parameter, and parameter passing can be treated the same as assignment, at least for some kinds of parameters. For a function call that returns a result, we can again use the assignment circuit.

There are other ways to produce circuits from programs, and there are many other language features we could cover, but perhaps that's enough to give you the main idea. Now let's put it all together with an example.

Greatest Common Divisor

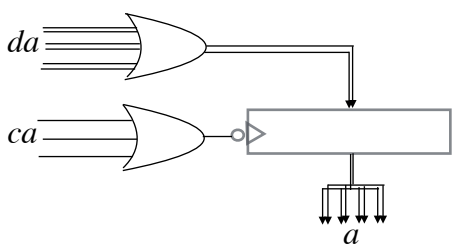
Here is a program in C, except that || is being used for parallel composition; it should also be understandable to Java programmers.

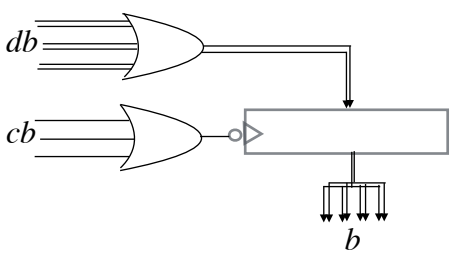
int *a*, *b*;

void *gcd* (**void**) { **while** (*a*!=*b*) **if** (*a*<*b*) *b* = *b*–*a*; **else** *a* = *a*–*b*; }

main ( ) { {*a* = 3; || *b* = 27;} *gcd* ( ); {*a* = 12; || *b* = 30;} *gcd* ( ); }

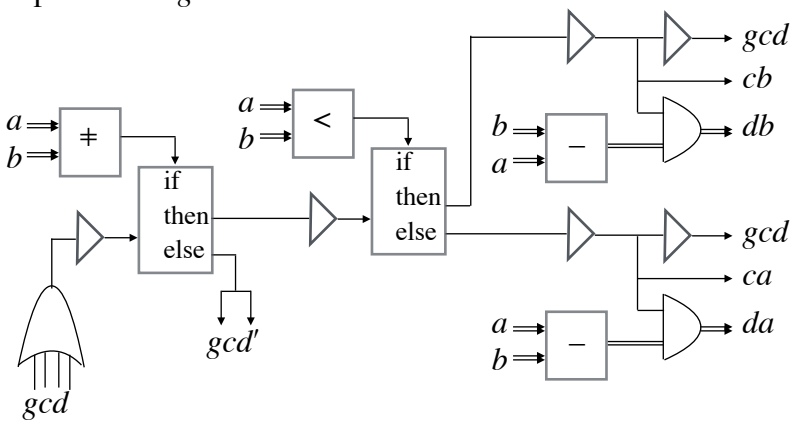
If a and b have positive integer values, then procedure gcd computes their greatest common divisor, and presents the answer as the value of both a and b. Here are the circuits of variables a and b.





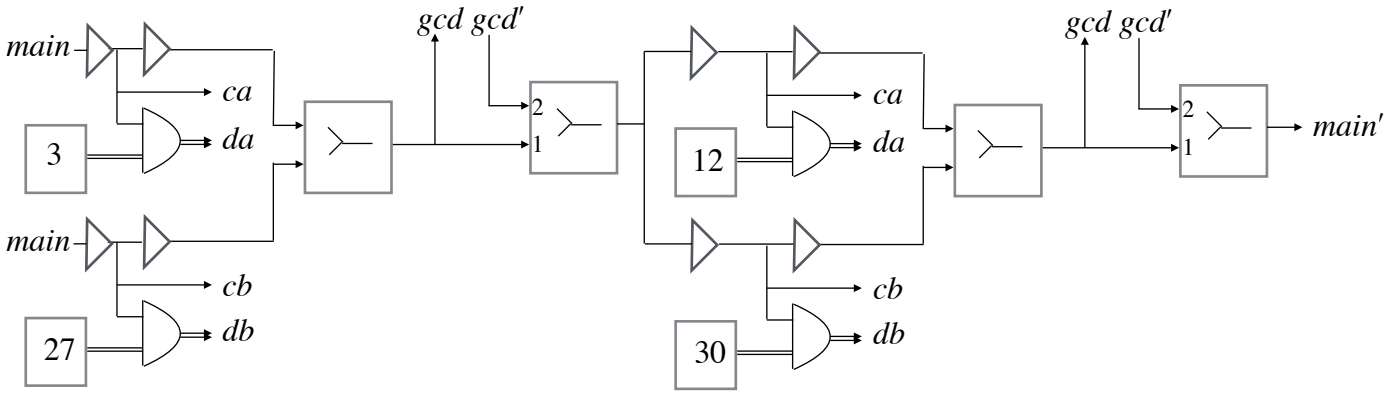
Variable a is assigned a value in 3 different places (one in *gcd* and two in *main*), so ca is 3 paths going into one **or** gate. Variable a is 32 bits, so the upper **or** gate is really 32 **or** gates, one for each data bit, and da is 3×32 bits. Output a is 32 bits, and they go to the 4 places (all four in *gcd*) where the value of a is needed. Variable b is just like variable a.

The circuit for procedure *gcd* is as follows:



The compiler must calculate the amount of delay necessary for each delay gate. For example, the delays for the subtractions can each be 0 because the delay for < was already sufficient, and variables a and b have not changed value since then. The delay for < has to be the additional delay over that needed for ∓.

Here is the circuit for *main*.



The box with 3 in it has no input, and it outputs the binary representation of 3, which is 00000000000000000000000000000011. Each 0 is produced by using ⊥ as source, and each 1 is produced by using ⊤ as source. Similarly for the other boxes with numbers in them.

This example shows how easily parallelism is introduced. It is just as easy at any scale, small or large. The example also shows that the circuits produced are neither synchronous (there is no clock) nor asynchronous (there is no handshaking).

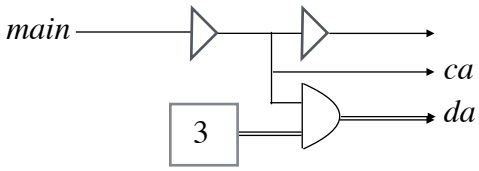
These drawings show what the parts of the circuit are, and what the connections are, but they are not intended to show the spatial layout. The best layout makes the paths as short as possible, and has the fewest path crossings (those two criteria may be conflicting). These circuit designs allow variables and arrays to be stored near the gates that use them, unlike the standard computer design. We do not consider layout further.

Optimization

After a program has been compiled to a circuit, much can be done to improve the circuit. We can reduce the number of gates, and so reduce the space required. We can reduce the time it requires to execute. When we are lucky, we can reduce both time and space, but sometimes there's a trade-off, and we need to decide which is more important.

An example of the time-space trade-off is the use of arithmetic operations. In the *gcd* circuit, there are two subtractors. We could decide to have one subtractor, and to call it from two places. The **if - else** logic ensures that these two calling sites will not call the subtraction procedure at the same time. We reduce the space at the cost of the time to call and return.

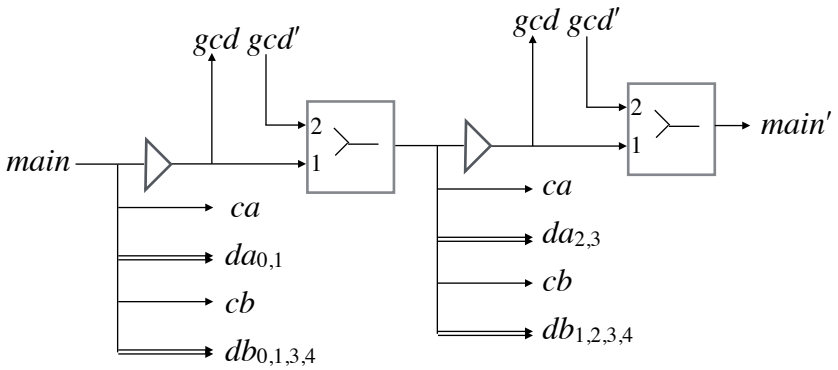
Whenever an input to an **or** gate or **and** gate or **not** gate is a constant, the gate can be eliminated, saving both time and space. This occurs, for example, in an assignment, when the expression being assigned is a constant. In our example program, the assignment *a*=3; gives us the circuit



The box with 3 in it outputs the binary representation of 3, which has 1s at bit positions 0 and 1. This takes no time, so the first delay is 0 and the delay gate is unnecessary. From the 3 box, all the 0s go to **and** gates, causing their outputs to be $\perp$, and these $\perp$s go to the **or** gates at the *da* input of variable *a*, so these inputs have no effect, and they can be eliminated. The two 1s that come from the 3 box make the outputs of those two **and** gates equal to *main*, so all the **and** gates can be eliminated. We just need to route *main* to *ca* and to bit positions 0 and 1 of the *da* input for variable *a*. All that remains is one delay gate to separate this assignment from subsequent operations.

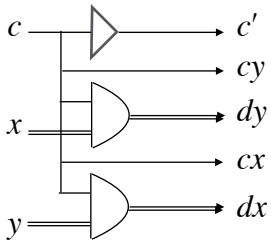
The same optimization applies to the assignment *b*=27;. So *main* is routed to *ca*, to bits 0 and 1 of *da*, to *cb*, and to bits 0, 1, 3, and 4 of *db*. All that remains of the two assignments *a*=3; and *b*=27; is two delay gates. These two delay gates have the same input (*main*), and are the same length (latch time). They go into a merge. So we can eliminate the merge and one of the delay gates.

Exactly the same optimizations occur with the two assignments *a*=12; and *b*=30;. There's very little left of the circuit for *main*.



In general, the most common constant to assign is 0, which routes the control path to the variable, and through a delay for latching, and no other circuitry. If an assignment assigns a variable to a variable, evaluation takes no time, and the corresponding delay is unnecessary. But we cannot eliminate the **and** gates as we could for a constant.

Let x and y be two variables. After optimization, the parallel composition of the two assignments $x{:=}y \parallel y{:=}x$ creates the following circuit.



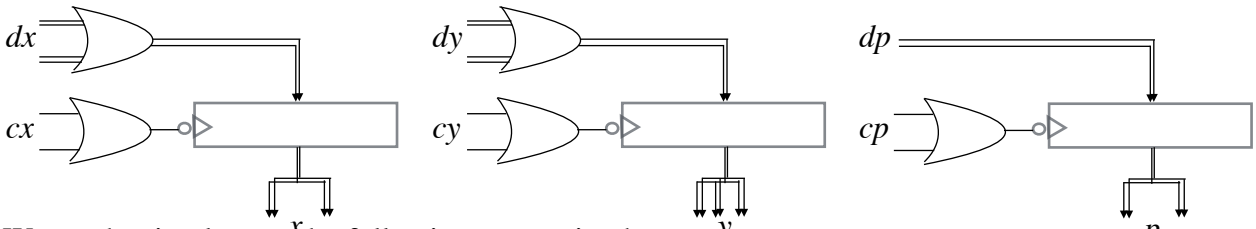
A variable is a falling-edge-triggered register, and that has delays on its data inputs so that, just after the control pulse has fallen, the register receives the data as it was just before the control pulse fell. Therefore each variable receives the value of the other variable before the other variable changes value. In other words, the two variables swap values.

Multiplier

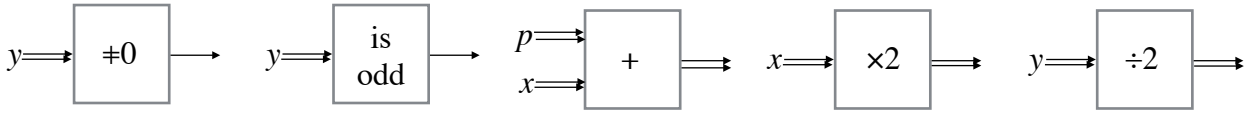
As promised earlier, we now present a circuit for an integer multiplier. But we don't design it from scratch as we did for the adder. We write a multiplier program that can be compiled to a circuit. Its inputs will be 32-bit integers a and b. Its output will be 32-bit integer p. The circuit ought to have an overflow output also, but to keep the example simple, we'll neglect that. We have nonlocal declarations for variables x, y, and p, and procedure *mult*. Before calling *mult*, assign the operand values to x and y; after calling *mult*, the result is the value of p.

int x, y, p;
void *mult* (**void**) {p=0; **while** (y != 0) {**if** (y%2 == 1) $p = p+x$; || $x = x$*2; || $y = y$/2;} }

The variables give us the usual circuits. Note that x and y each have one external assignment, and p has one external use. Also, the assignment p=0; requires no data input.



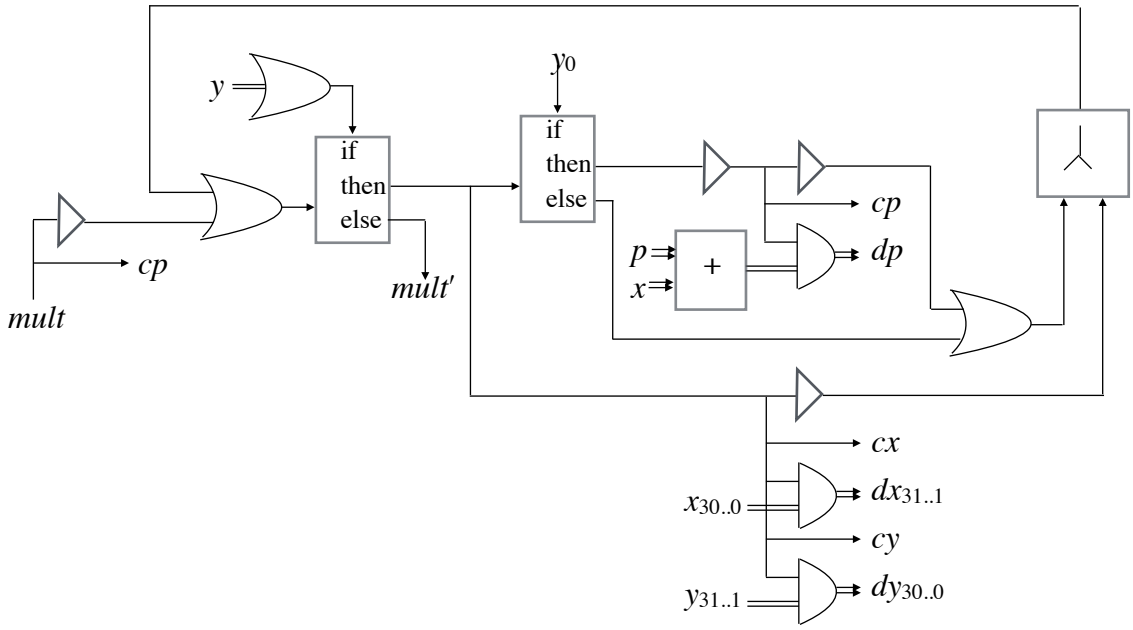
We need to implement the following expression boxes.



- The binary expression (y≠0) can be implemented as an **or** gate.
- The binary expression (y%2 == 1) is the way C tests to see if y is an odd number. It can be implemented just by using bit 0 of variable y. It requires no gates.
- Expression $p+x$ requires an adder.
- Expression x*2 is a shift left. Bit 0 of the output is 0 or $\perp$. Bit 1 of the output comes from bit 0 of the input x. Bit 2 of the output comes from bit 1 of the input x. And so on. Bit 31 of input x is unused. So x*2 requires no gates.

- Expression $y/2$ is a shift right. Bit 0 of input y is unused. Bit 0 of the output comes from bit 1 of the input y. Bit 1 of the output comes from bit 2 of the input y. And so on. Bit 31 of the output is 0 or $\perp$. So $y/2$ requires no gates.

In the body of the loop, the three assignments (one of them conditional) are to different variables, so they can all be in parallel. Like all our circuits, it also has a control input, which we'll call *mult*, and a control output, which we'll call *mult'*. After the optimizations, here is the resulting circuit.



That completes the implementation of the multiplier. Similarly, by writing programs, we can build an integer divider circuit, and circuits for floating-point arithmetic.

Computer

A computer's cpu (central processing unit) is a digital circuit. So it can be designed by the same method we have just seen for a greatest common divisor circuit and a multiplier circuit. First we write a program. To illustrate, we will write a program in C for a cpu that is simpler than commercially available cpu's, but it can compute anything that any computer can compute. The program is on the next page. You don't need to read it in detail. Just notice that it begins by declaring an array named RAM; it is the only array, and it becomes the computer's memory. Then there are some variable declarations that become registers. There's an accumulator register named AC, an instruction register named IR, a program counter named PC, and a condition code named E. The procedure is an infinite loop, which is known to cpu designers as the fetch-execute cycle. The body of the loop begins with fetch, which gets from RAM the instruction whose address is in PC, and puts this instruction in IR, and decodes it. The execute part of the loop is the switch statement. As a circuit, a switch statement becomes a general demultiplexer. The "case" input is the instruction code (op-code) from IRop, which is part of IR. The demultiplexer sends the control pulse to the circuit for one case, which executes one instruction. (If C had parallelism, we could use it effectively to improve the speed.)

If this program is executed on a computer, we would say that it simulates the cpu. If it is compiled to a circuit, then that circuit is the cpu. (For a full description of the cpu, see this webpage, and for the circuits, see this webpage.)

```
#define RamSize 16777216  /* 2^24 = 16 Megawords = 64 Megabytes */

union {unsigned int u; signed int i; float f;} RAM[RamSize], AC;
unsigned char IRop;  unsigned int IRadr;  /* IR in two parts */
unsigned int PC;   /* program counter */
unsigned char E;   /* condition code: overflow, error */

void execute (void) /*executes instructions in RAM starting at the
                      address in PC and ending normally
                      with a branch to RamSize (that means stop)
                      or abnormally with an illegal op-code
                      or an address past the end of memory*/
{ while (1)
  { if(PC==RamSize) return;
    if(PC>RamSize)
      {printf("\ninstruction address past end of memory\n"); return;}

    /* fetch starts here */
    IRop = RAM[PC].u / 0x01000000;  IRadr = RAM[PC].u & 0x00FFFFFF;
    if(IRadr>=RamSize && (IRop<=15 || IRop==18  || IRop==19))
      {printf("\ndata address past end of memory\n"); return;}

    /* execute starts here */
    switch(IRop)
    { case 0:  /*LDA*/ AC = RAM[IRadr]; PC++;  break;
      case 1:  /*STA*/ RAM[IRadr] = AC;  PC++;  break;
      case 2:  /*ADD*/ E = 0;  AC.i += RAM[IRadr].i;  PC++;  break;
      case 3:  /*SUB*/ E = 0;  AC.i -= RAM[IRadr].i;  PC++;  break;
      case 4:  /*MUL*/ E = 0;  AC.i *= RAM[IRadr].i;  PC++;  break;
      case 5:  /*DIV*/ if (RAM[IRadr].i==0) E = 1;
                       else {E=0; AC.i /= RAM[IRadr].i;}
                       PC++;  break;
      case 6:  /*MOD*/ if (RAM[IRadr].i==0) E = 1;
                       else{E=0; AC.i %= RAM[IRadr].i;}
                       PC++;  break;
      case 7:  /*FLA*/ E = 0;  AC.f += RAM[IRadr].f;  PC++;  break;
      case 8:  /*FLS*/ E = 0;  AC.f -= RAM[IRadr].f;  PC++;  break;
      case 9:  /*FLM*/ E = 0;  AC.f *= RAM[IRadr].f;  PC++;  break;
      case 10: /*FLD*/ if (RAM[IRadr].f==0.0) E = 1;
                       else {E=0; AC.f /= RAM[IRadr].f;}
                       PC++;  break;
      case 11: /*CIF*/ AC.f = (float) RAM[IRadr].i;  PC++;  break;
      case 12: /*CFI*/ if (RAM[IRadr].i>=0.0) AC.i = (int)(RAM[IRadr].f+0.5);
                       else AC.i = (int) (RAM[IRadr].f - 0.5);
                       PC++;  break;
      case 13: /*AND*/ AC.u &= RAM[IRadr].u;  E = (AC.u!=0);  PC++;  break;
      case 14: /*IOR*/ AC.u |= RAM[IRadr].u;  E = (AC.u!=0);  PC++;  break;
      case 15: /*XOR*/ AC.u ^= RAM[IRadr].u;  E = (AC.u!=0);  PC++;  break;
      case 16: /*BUN*/ PC = IRadr;  break;
      case 17: /*BZE*/ if (E) PC++; else PC = IRadr;  break;
      case 18: /*BSA*/ RAM[IRadr].u = PC+1;  PC = IRadr+1;  break;
      case 19: /*BIN*/ PC = RAM[IRadr].u & 0x00FFFFFF;  break;
      case 20: /*INP*/ AC.u = getchar ( );  PC++;  break;/*waits; no branch*/
      case 21: /*OUT*/ putchar (AC.u % 256);  PC++;  break; /* no branch */
      default: {printf("\nillegal op-code\n"); return;}
    } /*end switch*/
  } /*end while*/
} /*end execute*/
```

The entire program is just one page long. For a commercially available cpu, the program might be ten pages long, and it has the same structure. For comparison, a browser program, or a text editor program, might be a thousand pages long.

Cpu circuits are among the most complex circuits ever built; it's a long, difficult, and error-prone task to choose the gates and their connections to each other. The use of a gate-level language does not reduce the complexity. But in a general-purpose high-level programming language, a cpu program is short, and is much easier to debug and get right. From it, a circuit can be compiled automatically, and optimized automatically.

Conclusion

After you have written a program, there are two ways to get it executed.
(a) Compile it to the machine language of a computer, and run it on the computer.
(b) Compile it to a circuit design, fabricate the circuit, and power up the circuit.
Way (a) is easier and less expensive. While you are writing and testing the program, way (a) is the only reasonable option. Once the program has been written and tested, the best way depends on the characteristics of its use. If the program will be changed frequently, way (a) is better; software updates can be sent and installed more easily than sending a new circuit or getting a new circuit fabricated.

Cars contain circuits to control propulsion, steering, braking, and many other functions. Updates are not frequent, and a new circuit can be plugged in by the dealer during regular maintenance. In appliances, software is almost never updated. In general, programs embedded in things can reasonably be implemented as circuits rather than as a combination of software and computer.

When a program is compiled to a circuit, execution is many times faster than when it is compiled to the machine language of a computer and executed on the computer. As a circuit, there is no fetch-execute cycle, no memory bottleneck, and much more parallelism. If speed is important, way (b) is better.

If security is important, way (b) is better. Software running on a computer is subject to viruses, trojan horses, and other malware; circuits are not.

Even when way (a) is the better option, the computer that the program runs on is best designed and implemented using way (b).

Finally, when circuits are too complex to create directly, writing a program and compiling it to a circuit is the only option.

Acknowledgement This method of circuit design is joint work with Theo Norvell.

References

E.C.R.Hehner: from Boolean Agebra to Unified Algebra, *the Mathematical Intelligencer* v.26 n.2 p.3-19, 2004

E.C.R.Hehner: Unified Algebra, *International Journal of Mathematical Sciences* v.1 n.1 p.20-37, 2007

E.C.R.Hehner, R.N.S.Horspool: a New Representation of the Rational Numbers for Fast Easy Arithmetic, *SIAM Journal on Computation*, v.8 n.2 p.124-134, 1979 May

E.C.R.Hehner, T.S.Norvell, R.F.Paige: High-Level Circuit Design, *Programming Methodology*, Morgan, McIver (eds.), Chapter 18 p.381-412, Springer, 2003