

Digital Logic a learn.sparkfun.com tutorial

Available online at: <http://sfe.io/t97>

Contents

- [Introduction](#)
- [Combinational Logic](#)
- [Sequential Logic](#)
- [Boolean Logic in Programming](#)
- [Resources and Going Further](#)

Introduction

Digital, or boolean, logic is the fundamental concept underpinning all modern computer systems. Put simply, it's the system of rules that allow us to make extremely complicated decisions based on relatively simple "yes/no" questions.

In this tutorial you will learn about...

Digital circuitry

Digital logic circuits can be broken down into two subcategories-**combinational** and **sequential**. Combinational logic changes "instantly"- the output of the circuit responds as soon as the input changes (with some delay, of course, since the propagation of the signal through the circuit elements takes a little time). Sequential circuits have a clock signal, and changes propagate through stages of the circuit on edges of the clock.

Typically, a sequential circuit will be built up of blocks of combinational logic separated by memory elements that are activated by a clock signal.

Programming

Digital logic is important in programming, as well. Understanding digital logic makes complex decision making possible in programs.

There are also some subtleties in programming that are important to understand; we'll get into that once we've covered the basics.

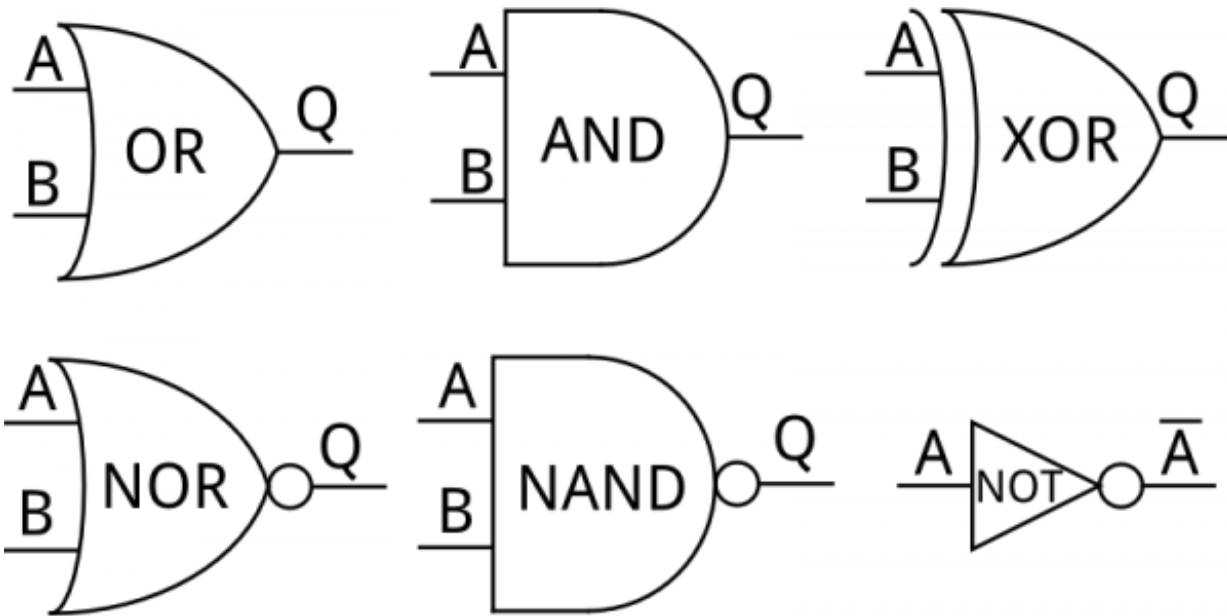
Suggested Reading

Before getting started, it might be a good idea to review our [tutorial on binary numbers](#), if you haven't already. There is a small amount of discussion on boolean logic in there, but we'll be going much further into the topic here. Here are some other topics you should be familiar with before

getting started.

- [What is Electricity?](#)
- [Binary](#)
- [Analog vs Digital](#)
- [Logic Levels](#)

Combinational Logic



Combinational circuits are built of five basic logic gates:

- AND gate - output is 1 if BOTH inputs are 1
- OR gate - output is 1 if AT LEAST one input is 1
- XOR gate - output is 1 if ONLY one input is 1
- NAND gate - output is 1 if AT LEAST one input is 0
- NOR gate - output is 1 if BOTH inputs are 0

There is a sixth element in digital logic, the inverter (sometimes called a NOT gate). Inverters aren't truly gates, as they do not make any decisions. The output of an inverter is a 1 if the input is a 0, and vice versa.

A few things of note about the above image:

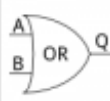
- Usually, the name of the gate is *not* printed; the symbol is assumed to be sufficient for identification.
- The A-B-Q type terminal notation is standard, although logic diagrams will usually omit them for signals which are not inputs or outputs to the system as a whole.
- Two input devices are standard, but you will occasionally see devices with more than two inputs. They will, however, only have one output.


Digital logic circuits are usually represented using these six symbols; inputs are on the left and


outputs are to the right. While inputs can be connected together, outputs should never be connected to one another, only to other inputs. One output may be connected to multiple inputs, however.


Truth Tables


The descriptions above are adequate to describe the functionality of single blocks, but there is a more useful tool available: the truth table. Truth tables are simple plots which explain the output of a circuit in terms of the possible inputs to that circuit. Here are truth tables describing the six main elements:

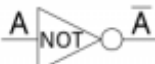
		A	
		0	1
B	0	0	1
	1	1	1

		A	
		0	1
B	0	0	0
	1	0	1

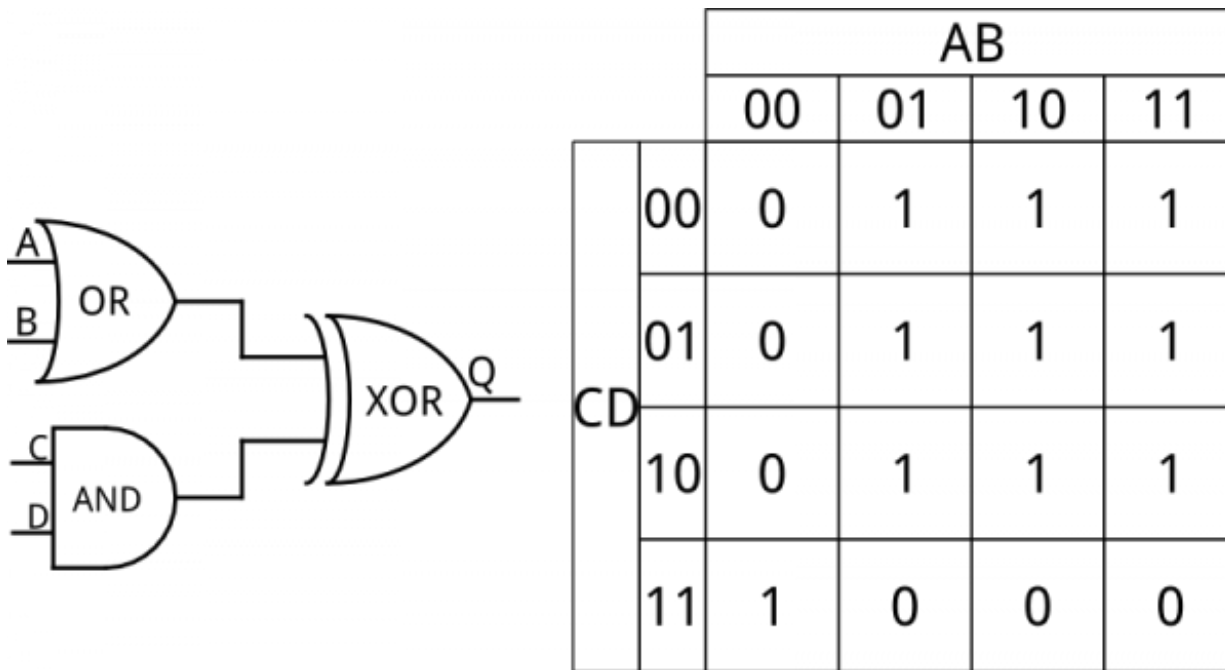
		A	
		0	1
B	0	0	1
	1	1	0

		A	
		0	1
B	0	1	0
	1	0	0

		A	
		0	1
B	0	1	1
	1	1	0

		A	
		0	1
		1	0

Truth tables can be expanded out to an arbitrary scale, with as many inputs and outputs as you can handle before your brain melts. Here's what a four-input circuit and truth table look like:



Written Boolean Logic

It is, of course, useful to be able to write in a simple mathematical format an equation representing a logical operation. To that end, there are mathematical symbols for the unique operations: AND, OR, XOR, and NOT.

- A AND B should be written as AB (or sometimes $A \cdot B$)
- A OR B should be written as $A + B$
- A XOR B should be written as $A \oplus B$
- NOT A should be written as A' or \overline{A}

You'll note that there are two missing elements on that list: NAND and NOR. Typically, those are simply represented by complementing the appropriate representation:

- A NAND B is written as $(AB)'$, $(A \cdot B)'$, or \overline{AB}
- A NOR B is written as $(A + B)'$ or $\overline{A + B}$

Sequential Logic

Combinational logic is great, but without adding sequential circuitry, modern computing would not be possible.

Sequential circuitry is what adds memory to our logical systems. As mentioned earlier, combinational logic produces results after a delay. That delay varies according to lots and lots of things: the manufacturing process of the parts involved, the temperature of the silicon, the complexity of the circuit. If the output of a circuit is dependant upon results from two other combinational circuits and the results arrive at different times (which they will, in the real world), a combinational circuit will "glitch" briefly, outputting a result which may not be consistent with the desired operation.

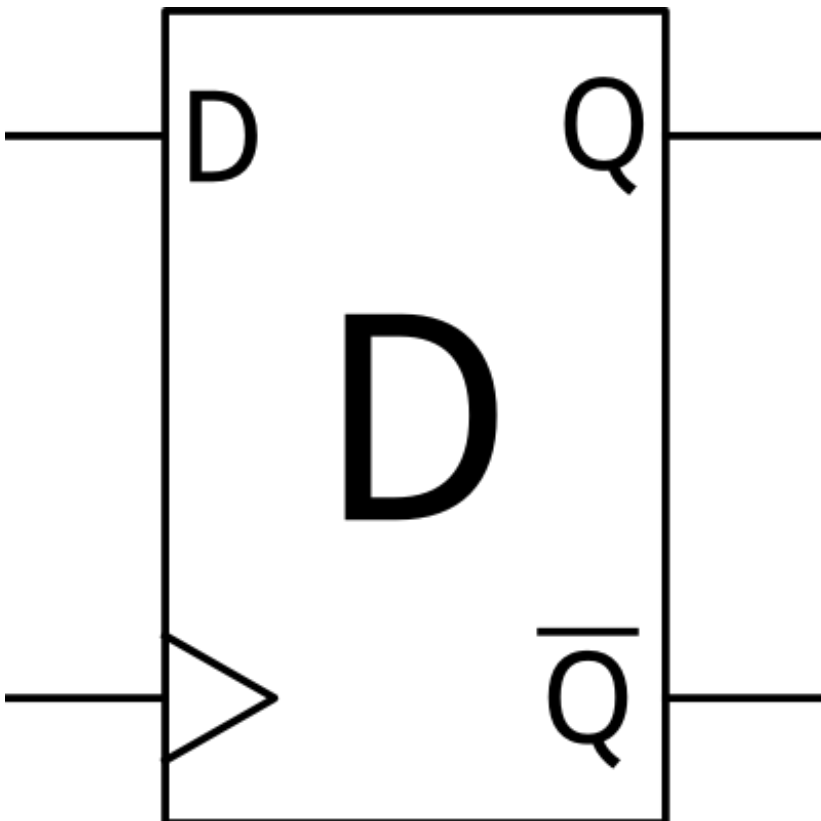
A sequential circuit, however, only samples and propagates the output at specific times. If the input changes between those times, it is ignored. The sampling time is usually synchronized across the entire circuit and is referred to as the "clock". When a computer's "speed" is cited, this is the value in question. It is possible to design "asynchronous" sequential circuits, which do not rely on a synchronized global clock. However, those systems pose great difficulties, and we won't be discussing them here.

As a side note, any section of digital logic will have two characteristic delay values: the minimum delay time and the maximum delay time. If the circuit fails the minimum delay time (i.e., is faster than it should be), the circuit will fail, irreparably so. If that circuit is part of a larger device, like a computer CPU, the entire device is garbage and cannot be used. If the maximum delay time fails (i.e., the circuit is slower than it should be), the clock speed can be reduced to accommodate the slowest circuit in the system. Maximum delay times tend to go up as the silicon forming a circuit warms up, which is why computers become unstable when they overheat or as the clock speed is increased (as is the case with overclocking).

Sequential Circuit Elements

As is the case with combinational logic, there are several basic circuit elements which form the building blocks of sequential circuits. These blocks are built up from the basic combinational elements, using feedback from the output to stabilize the input. They come in two "flavors": latches and flip-flops. While the terms are frequently used interchangeably, latches are generally less useful, as they are not clocked; we'll focus on flip-flops.

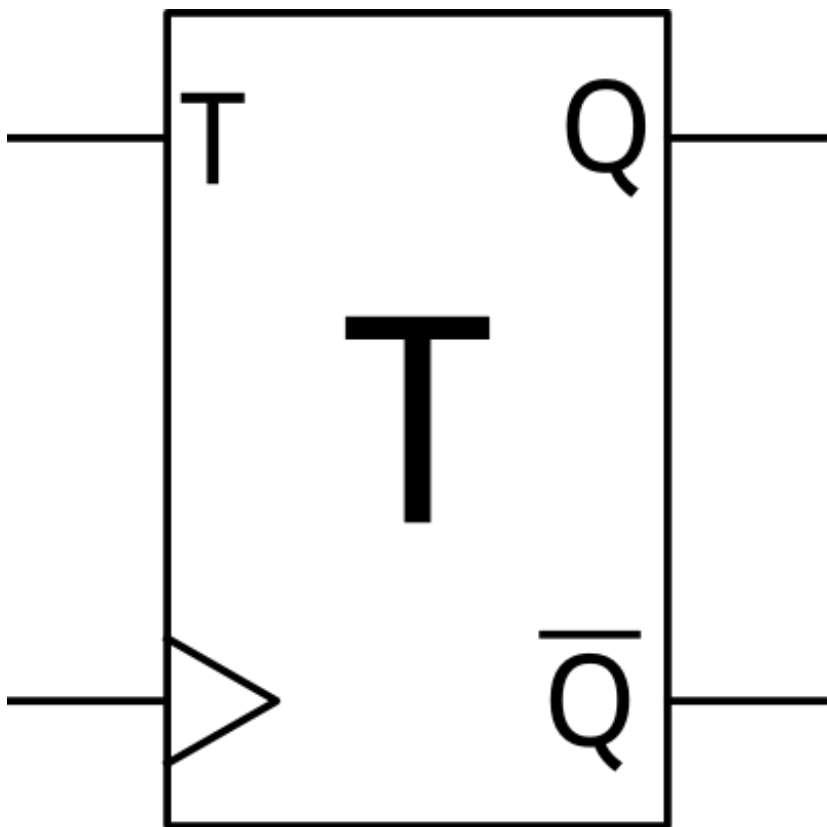
D-type Flip-Flop



The simplest type of flip-flop is the D-type. D flip-flops are simple -- upon a clock edge (normally rising, although they can be found with a built-in inverter to clock in on the falling edge instead), the input is latched to the output.

Usually, the clock input is denoted by the small triangle impinging on the symbol. Most flip-flops provide two outputs: the "normal" output, and the complemented output.

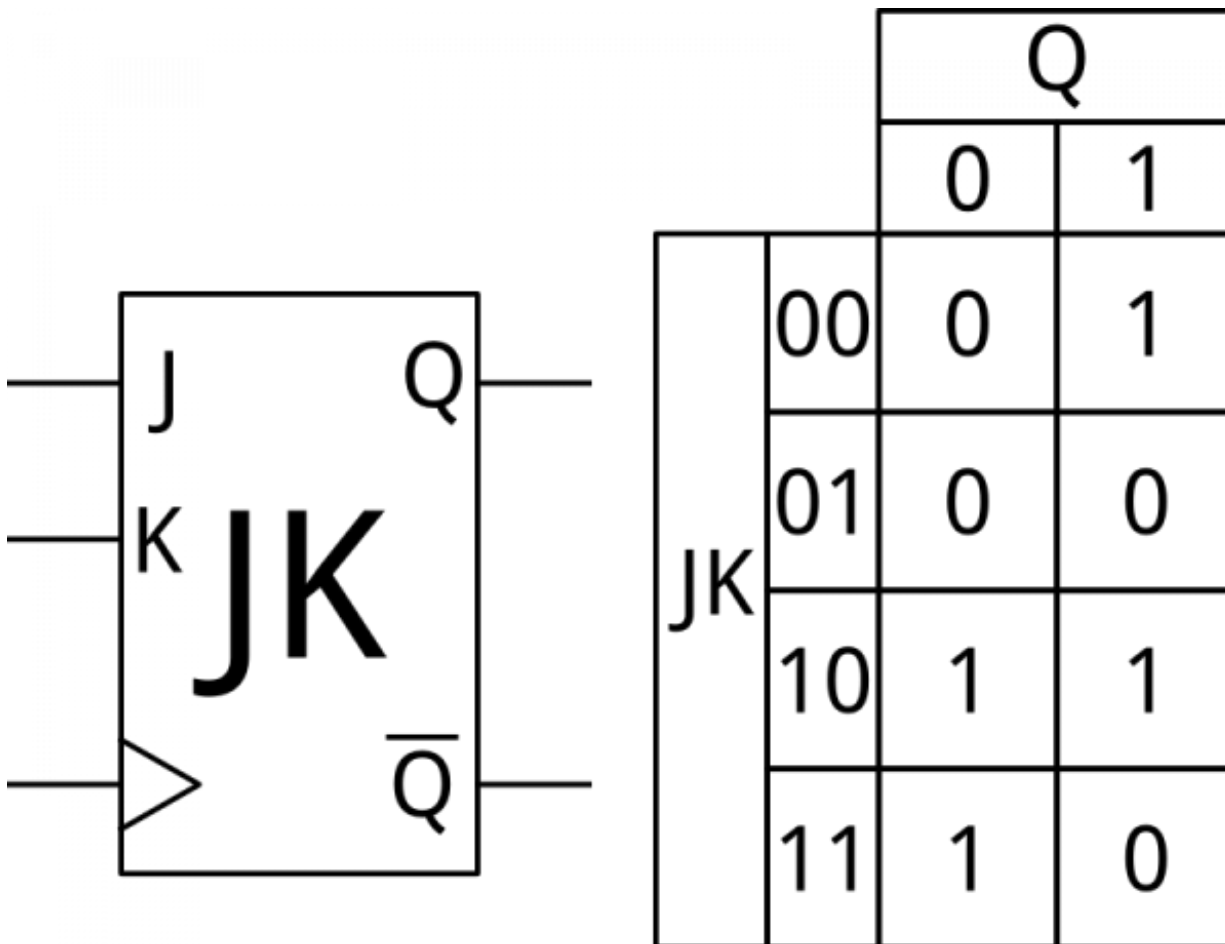
T-type Flip-Flop



Only slightly more complex is the T-type. The 'T' stands for "toggle." When a clock edge occurs, if the input T is a 1, the output changes state. If the input is a 0, the output remains the same. As with the D-type, the complement of the output is usually provided.

A useful function of the T flip-flop is as a clock division circuit. If T is held high, the output will be the clock frequency divided by two. A chain of T flip-flops can thus be used to produce slower clocks from a device's master clock.

JK-type Flip-Flop



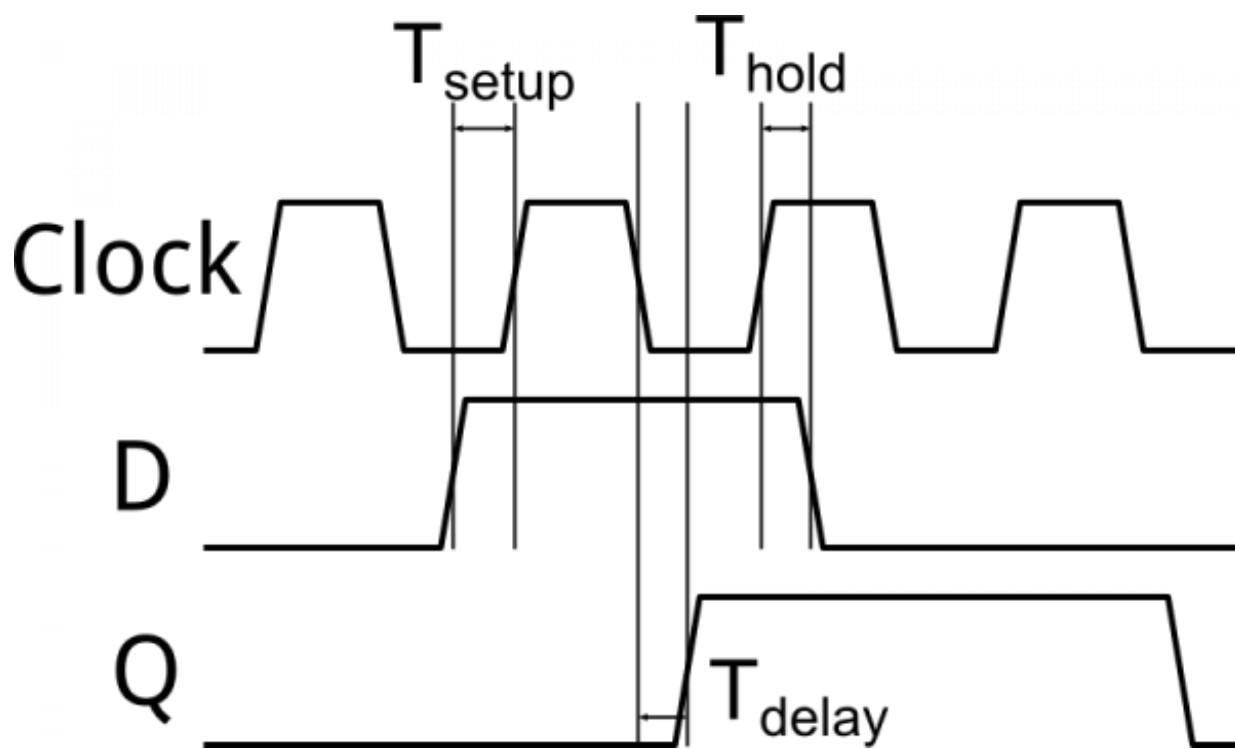
Finally, we have the JK-type. The JK-type is the only one of the three which truly requires a truth table to explain; it has two inputs (J and K), and the output can be left the same, set, cleared, or toggled, depending on the combination of input signals present. Of course, as with all flip-flops, the input at the moment of the clock is the only thing that matters.

Setup, Hold, and Propagation Times

All sequential circuits have what are called "**setup**" and "**hold**" times, as well as a **propagation delay**. Understanding these three things is critical to designing sequential circuits that work as expected.

The setup time is the minimum amount of time before a rising clock edge occurs that a signal must arrive at the input of a flip-flop in order for the flip-flop to latch the data correctly. Likewise, the hold time is the minimum time a signal must remain stable after the rising clock edge occurs before it can be allowed to change.

While setup and hold times are given as minimum values, the propagation delay is given as a maximum. Simply put, the propagation delay is the greatest amount of time after a falling edge at the clock before you can expect to see the signal on the outputs. Here's a graphic explaining them:

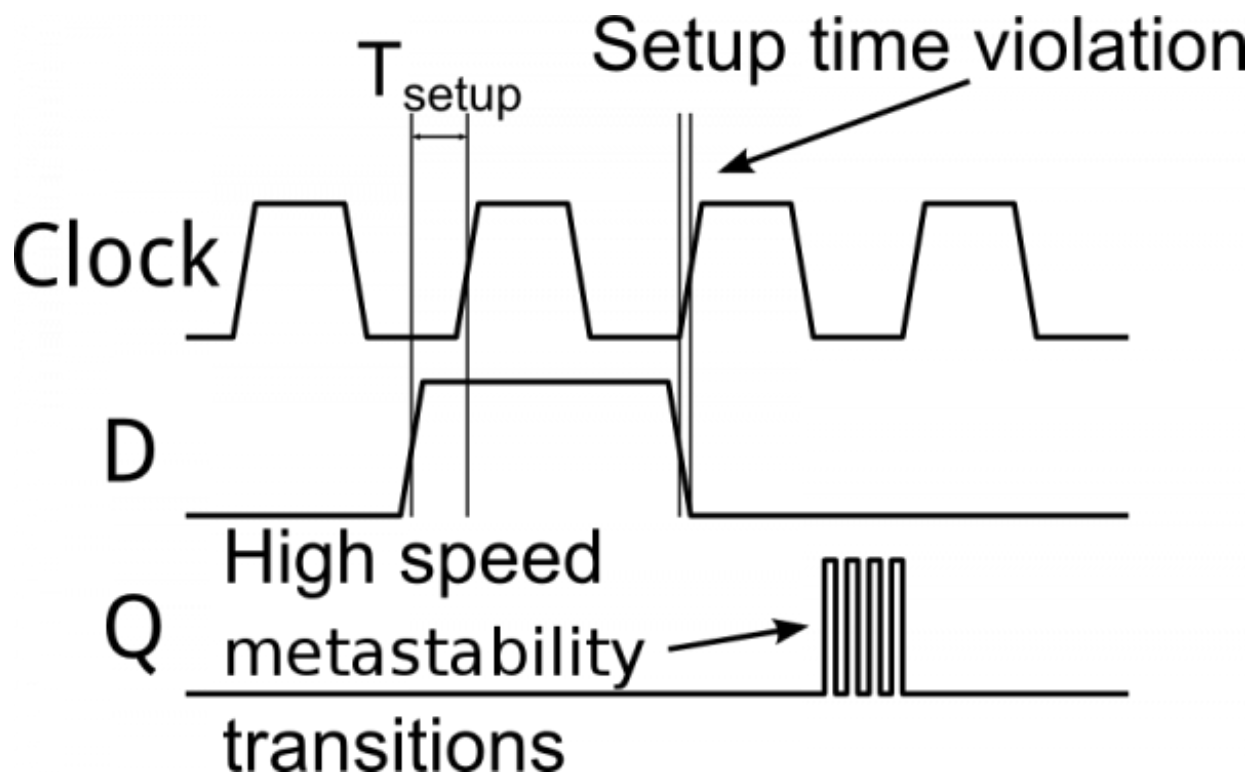


Note that in the above image, transitions are drawn as being slightly angled. This serves two purposes: it reminds us that clock and data edges are *never* truly right angles and will always have some non-zero rise or fall time, and it makes it easier to see where the vertical lines marking the various times intersect with the signals.

The combination of these three values determines the highest clock speed a device may use. If the propagation delay of one part plus the setup time of the next part in the circuit exceeds the time between the falling edge of one clock pulse and the rising edge of the next, the data will not be stable on the input of the second component, causing it to behave in an unexpected manner.

Metastability

Failing to adhere to setup and hold times can lead to a problem called "metastability". When a circuit is in a metastable state, the output of a flip-flop can oscillate rapidly between the two normal states -- often at a rate far above the clock rate of the circuit.



Metastability problems can range from spurious operation up to damage of the chip, since they can increase current consumption. While metastability usually resolves on its own, by the time it does so, the system may be in a totally unknown state and need to be completely reset to restore proper operation.

A common way in which metastability issues arise is when a signal crosses clock domains -- in other words, when a signal passes between devices which are being clocked by different sources. Since the clocks are not synchronized (and even if the clocks are at the same nominal frequency, reality dictates that they will be *slightly* different), eventually a clock edge and a data edge are bound to be too close for comfort, resulting in a setup time violation. A simple fix for this issue is to run all inputs into a system through a pair of cascaded D flip-flops. Even if the first flip-flop goes into metastability, it will (hopefully) have settled down to a steady state before the next clock pulse, allowing the second flip-flop to read the correct data. This results in a one-cycle delay in incoming data edges, which is almost always insignificant compared to the risk of metastability.

Boolean Logic in Programming

All of this can be applied in the programming world, as well. Most programs are simply decision trees: "if this is true, then do this". To explain this, we'll use C-code in an Arduino context.

Bitwise Logic

When we talk about "bitwise" logic, what we really mean is logical operations which return a value. Take, for example, this piece of code:

```
language:cpp
byte a = b01010101;
```

```
byte b = b10101010;
byte c;
```

We can do a bitwise operation using 'a' and 'b' and putting the result into 'c'. Here's what that looks like:

```
c = a & b; // bitwise AND-ing of a and b; the result is b00000000
c = a | b; // bitwise OR-ing of a and b; the result is b11111111
c = a ^ b; // bitwise XOR-ing of a and b; the result is b11111111
c = ~a; // bitwise complement of a; the result is b10101010
```

In other words, the each bit in the result is equal to the operation applied to the two corresponding bits in the operands:

Okay, that's great, but what of it? It turns out we can do some pretty useful things by using bitwise operators to manipulate registers: we can selectively clear, set, or toggle single bits, check to see if a bit is set or clear, or if several bits are set or clear. Here are some examples using these operations:

```
c = b00001111 & a; // clear the high nibble of a, but leave the low nibble alone.
                      // the result is b00000101.
c = b11110000 | a; // set the high nibble of a, but leave the low nibble alone.
                      // the result is b11110101.
c = b11110000 ^ a; // toggle all the bits in the high nibble of a.
                      // the result is b10100101.
```

Any bitwise operation can be self-applied by combining it with the equal sign:

```
a ^= b11110000; // XOR a with b11110000 and store the result back in a
b |= b00111100; // OR b with b00111100 and store the result back in b
```

Bit Shifting

Another useful bitwise operation that can be performed on a piece of data is a bit shift. This is simply a slide of the data left or right by a certain number of places; data which is shifted out disappears and is replaced by a 0 being shifted in from the other end.

```
byte d = b11010110;
byte e = d>>2; // right-shift d by two positions; e = b00110101
e = e<<3; // left-shift e by three positions; e = b10101000
```

We'll demonstrate some uses of bit shifting later. One *very* useful application for bit shifts is multiplication and division: each right shift is the same as a division by two (although remainder information is lost) and each left shift is the same as a multiplication by two. This is useful because multiply and divide are often very time expensive operations on small processors, like the Arduino's, but bit shifts are usually very efficient.

Comparison and Relational Operators

We'll want some way to compare two values: there is a family of operators which do just that and return "TRUE" or "FALSE" depending on the result of the comparison.

- `==` "is equal to" (true if values are equal, false otherwise)
- `!=` "is not equal to" (true if values are different)
- `>` "is greater than" (true if left operand is greater than right operand)
- `<` "is less than" (true if left operand is less than right operand)
- `>=` "is greater than, or equal to" (true if left operand is greater than, or exactly equal to, right operand)
- `<=` "is less than, or equal to" (true if left operand is less than, or exactly equal to, right operand)

It is generally quite important that the values compared be of the same data type; unexpected things can happen if you compare a "byte" and an "int", for example.

Logical Operators

Logical operators are operators which produce a "TRUE" or "FALSE", rather than a new value of the same type. They're more like what we tend to think of as conjunctions: "If it's not raining, and it's windy, go fly a kite". Translated into C, that sentence might look like this:

```
if ( (raining != true) && (windy == true) ) flyKite();
```

Note the parentheses around the two subclauses. Though not strictly necessary, it's good practice to keep your code as readable as possible by grouping subclauses together.

Also note that the logical AND operator (`&&`) produces a true/false answer based on whether or not the subclauses produced true/false answers. We could just as easily have had a numeric value in one of the subclauses:

```
if ( (raining != true) && ( (windSpeed >= 5) || (reallyBusy != true) ) ) flyKite();
```

This clause will send me out to fly a kite, so long as it's not raining, but only if there's some wind or I'm not busy (I *will* try and fly with no wind).

Again, note the parentheses. If we remove the parentheses around "`(windSpeed >= 5) || (reallyBusy != true)`" -- with `||` representing the OR operator -- we create an ambiguous statement which may or may not do what we want it to do.

Flow Control

Now that we can create complex logical statements, let's look at the things we can do with the answers to those questions.

if/else if/else Statements

The simplest decision is "if/else". If/else if/else allow you to set up a series of tests, of which only one can ever be executed at any time:

```
if ( reallyBusy == true ) workHarder();
else if ( (raining != true) && (windy == true) ) flyKite();
else work();
```

With those three statements, I'll *never* fly a kite if I'm really busy, and if I'm not really busy, and it's not a good day for it, I'll just keep working. Let's change the else if() to if(), like so:

```
if ( reallyBusy == true ) workHarder();  
if ( (raining != true) && (windy == true) ) flyKite();  
else work();
```

Now, if we've got a nice kite flying day on our hands, even if I'm really busy, I'll only work harder for a very, very short period--basically, right up until I notice that it's nice out. Furthermore, if it's *not* a nice day, my work harder status will be downgraded to just plain old work immediately after I start working harder!

You can imagine what would happen if we replaced "workHarder()" with "turnLEDOn()" and "work()" with "turnLEDOff()". In the first case, the LED may be on for some time, or off for some time. In the second case, however, regardless of the state of the "reallyBusy" flag, the LED will turn off almost instantly after the first if() statement turned it on, and you'd find yourself sitting around wondering why the "reallyBusy" light never turns on!

switch/case/default Statements

Less powerful but more readable than a long chain of if/else statements, switch/case/default allows you to make a decision based on the value of a variable:

```
switch(menuSelection) {  
  case '1':  
    doMenuOne();  
    break;  
  case '2':  
    doMenuTwo();  
    break;  
  case '3':  
    doMenuThree();  
    break;  
  default:  
    flyKite();  
    break;  
}
```

The switch() statement *only* allows us to check equivalence, but since that's a fairly common thing to want to do, it comes in pretty handy. There are two really important things to notice about this: the "break;" statements and the "default:" case.

"default:" is what will be executed if none of the others match. It's not strictly necessary; if there's no default case, then nothing happens if all the matches fail. Of course, you usually *want* something to happen, and it's best not to assume that it's impossible for all matches to fail.

"break;" jumps out of the current conditional. It can be used inside of any type of conditional (more on that later), and in this case, a failure to include a break at the end of each case will result in code after the case being executed, *even if subsequent case matches fail*

while/do...while Loops

So far, we've looked at code for making a decision once. What if you want to repeat an action, over and over, as long as a condition holds? That's where `while()` and `do...while()` come into play.

```
while (windy == true) flyKite();
```

When your code reaches a `while()` statement, the program evaluates the conditional ("Is it windy?") and, if it evaluates to "TRUE", executes the code. Once code execution is complete, the conditional will be evaluated once more. If the conditional is still "TRUE", the code will execute again. This repeats over and over, until the conditional evaluates to "FALSE" or a `break` statement is encountered.

You can nest an `if()` statement (or a `switch()`, or another `while()`, or in fact anything you want) inside your `while()` loop:

```
while (windy == true) {  
    flyKite();  
    if (bossIsMad == true) break;  
}
```

So, with that loop, I'll fly my kite until the wind gives out *or* my boss gets mad at me.

A variation on `while()` loops is the `do...while()` loop.

```
do {  
    flyKite();  
} while (windy == true);
```

In this case, the code inside the brackets runs once, *even if the conditional is false*. In other words, regardless of the state of the wind, I'll go out and drag a kite around, but if the wind isn't there, I'll give up.

Finally, by sticking "TRUE" into the conditional, it's possible to create code that will execute forever:

```
while(true) {  
    flyKite();  
}
```

With that chunk of code, I'll just keep dragging my kite around the field forever, regardless of wind, my boss's satisfaction with it, hunger, cougars, etc. It's still possible to break out of that code using the `break` statement, of course; it will just never cease execution on its own.

for() Loops

The last type of conditional execution we need to consider is the `for()` loop. `for()` loops allow us to execute a chunk of code a specific number of times. The syntax of a `for` loop looks like this:

```
for (byte i = 0; i < 10; i++) {  
    Serial.print("Hello, world!");  
}
```

Within the `for()` loop parentheses are three semicolon separated statements. The first one is the iterator: the variable which we are changing with each pass. It's also where the iterator's initial value is set. The center one is the comparison we'll do after each pass. Once that comparison fails, we break out of the loop. The last statement is what we want to do after each pass through the loop. In this case, we want to increment the iterator by one.

The most common error in a `for()` loop is an off-by-one error: you *mean* for the code to execute 10 times, but it ends up executing 9 times, or 11 times. This is usually a result of using a "`<=`" instead of "`<`" or vice versa.

Resources and Going Further

Understanding digital logic is a crucial skill in electronics. For more information on the topic, see the below sources:

- [Digital Logic](#) - A good site covering most of the material presented here
- [Boolean algebra](#) - Wikipedia's page on Boolean algebra, which is the discipline underpinning this topic.
- [Quine-McCluskey method](#) - Q-M is a method of simplifying digital circuits to a minimum set of necessary gates, given a number of inputs and a desired output map.
- [LogicBlocks and an Introduction to Digital Logic](#)

Or check out this related blog post.

[According to Pete: Digital Circuits](#)

[October 1, 2018](#)

[Read Post](#)

learn.sparkfun.com | [CC BY-SA 3.0](#) | SparkFun Electronics | Niwot, Colorado