2023

9. Binary

R2: SCRAPY Guide

Project number: **2021-1-FR01-KA220-SCH-000031617**

Co-funded by the European Union

ECAM EPMI

30/04/2023

Table of Contents

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

1 Introduction

Number systems are the methods we use to represent numbers. Since grade school, we've all been mostly operating within the comfortable confines of a base-10 number system, but there are many others. Base-2, base-8, base-16, base-20, base...you get the point. There are an infinite variety of base-number systems out there, but only a few are especially important to electrical engineering.

The really popular number systems even have their name. Base-10, for example, is commonly referred to as the decimal number system. Base-2, which we're here to talk about today, also goes by the moniker of binary. Another popular numeral system, base-16, is called hexadecimal.

The base of a number is often represented by a subscripted integer trailing a value. So, in the introduction above, the first image would be $10010_{something}$ while the second image would be $1002_{something}$. This is a handy way to specify a number's base when there's ever any possibility of ambiguity.

2. Why Binary?

Why binary you ask? Well, why decimal? We've been using decimals forever and have mostly taken for granted the reason we settled on the base-10 number system for our everyday number needs. It's because we have 10 fingers, or it's just because the Romans forced it upon their ancient subjugates. Regardless of what led to it, tricks we've learned along the way have solidified base-10's place in our hearts; everyone can count by 10s. We are even round large numbers to the nearest multiple of 10. We're obsessed with 10!

Computers and electronics are limited in the finger-and-toe department. At the lowest level, they only have two ways to represent the state of anything: ON or OFF, high or low, 1 or 0. And so, all electronics rely on a base-2 number system to store, manipulate, and math numbers.

The heavy reliance electronics place on binary numbers means it's important to know how the base-2 number system works. You'll commonly encounter binary, or its cousins, like hexadecimal, all over computer programs. Analysis of Digital logic circuits and other very low-level electronics also requires heavy use of binary.

In this lesson, you'll find that anything you can do to a decimal number can also be done to a binary number. Some operations may be even easier to do on a binary number (though others can be more painful). We'll cover all of that and more in this lesson.

3 Counting and Converting

The base of each number system is also called the radix. The radix of a decimal number is ten, and the radix of binary is two. The radix determines how many different symbols are required to flesh out a number system. In our decimal number system, we've got 10

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

numeral representations for values between nothing and ten somethings: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Each of those symbols represents a very specific, standardized value.

In binary, we're only allowed two symbols: 0 and 1. But using those two symbols we can create any number that a decimal system can.

3.1 Counting in binary

You can count in decimals endlessly, even in your sleep, but how would you count in binary? Zero and one in base-two should look pretty familiar: 0 and 1. From there, things get decidedly binary.

Remember that we've only got those two digits, so as we do in decimal when we run out of symbols, we've got to shift one column to the left, add a 1, and turn all of the digits to the right to 0. So, after 1 we get 10, then 11, then 100. Let's start counting...

Decimal	Binary	...	Decimal	Binary
0	0		16	10000
1	1		17	10001
2	10		18	10010
3	11		19	10011
4	100		20	10100
5	101		21	10101
6	110		22	10110
7	111		23	10111
8	1000		24	11000
9	1001		25	11001
10	1010		26	11010
11	1011		27	11011
12	1100		28	11100
13	1101		29	11101
14	1110		30	11110
15	1111		31	11111

Does that start to paint the picture? Let's examine how we might convert from those binary numbers to decimals.

3.2 Converting binary to decimal

There's no one way to convert binary to decimal. We'll outline two methods below, the more "mathy" method, and another that's more visual. We'll cover both, but if the first uses too much ugly terminology skip down to the second.

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Method 1

There's a handy function we can use to convert any binary number to decimal:

$$a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_1 2^1 + a_0 2^0$$

There are four important elements to that equation:

an, an-1, a1, etc., are the digits of a number. These are the 0's and 1's you're familiar with, but in binary, they can only be 0 or 1.

The position of a digit is also important to observe. The position starts at 0, on the right-most digit; this 1 or 0 is the least significant. Every digit you move to the left increases in significance, and also increases the position by 1.

The length of a binary number is given by the value of n, actually, it's n+1. For example, a binary number like 101 has a length of 3, and something larger, like 10011110 has a length of 8.

Each digit is multiplied by a weight: the 2n, 2n-1, 21, etc. The right-most weight - 20 equates to 1, move one digit to the left and the weight becomes 2, then 4, 8, 16, 32, 64, 128, 256,... and on and on. Powers of two are of great importance to binary, they quickly become very familiar.

Let's get rid of those n's and exponents, and carry out our binary positional notation equation out eight positions:

$$a_7 \cdot 128 + a_6 \cdot 64 + a_5 \cdot 32 + a_4 \cdot 16 + a_3 \cdot 8 + a_2 \cdot 4 + a_1 \cdot 2 + a_0 \cdot 1$$

Taking that further, let's plug in some values for the digits. What if you had a binary number like 10011011? That would mean (an) values of:

1	0	0	1	1	0	1	1
a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

Method 2

Another, more visual way to convert binary numbers to decimals is to begin by sorting each 1 and 0 into a bin. Each bin has a successive power of two weights to it, the 1, 2, 4, 8, 16,... we're used to. Carrying it out to eight places would look something like this:

128	64	32	16	8	4	2	1

So, if we sorted our 10011011 binary number into those bins, it'd look like this:

128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	1

For every bin that has a binary 0 value in it, just cross out and remove it.

128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	1

And then add up any remaining weights to get your number!

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

3.3 Converting from decimal to binary

Just like going from binary to decimal, there's more than one way to convert decimal to binary. The first uses division and remainders, and the second uses subtraction. Try both and stick to one you're comfortable with!

Method 1

It isn't quite as simple to convert a decimal number to binary. This conversion requires repeatedly dividing the decimal number by 2, until you've reduced it to zero. Every time you divide the remainder of the division becomes a digit in the binary number you're creating.

Don't remember how to do remainders? If it's been a while, remember that, since we're dividing by two, if the dividend is even, the remainder will be 0; an odd dividend means a remainder of 1.

For example, to convert 155 to binary you'd go through this process:

155 ÷ 2 = 77 R 1 (That's the right-most digit, 1st position)
77 ÷ 2 = 38 R 1 (2nd position)
38 ÷ 2 = 19 R 0 (3rd position)
19 ÷ 2 = 9 R 1
9 ÷ 2 = 4 R 1
4 ÷ 2 = 2 R 0
2 ÷ 2 = 1 R 0
1 ÷ 2 = 0 R 1 (8th position)

The first remainder is the least-significant (right-most) digit, so read from top-to-bottom to flesh out our binary number right to left: 10011011. Match it up with the example above...that's a bingo!

Method 2

If dividing and finding remainders isn't your thing, there may be an easier method for converting decimal to binary. Start by finding the largest power of two that's still smaller than your decimal number and subtract it from the decimal. Then, continue to subtract by the largest possible power of two until you get to zero. Every weight position that was subtracted, gets a binary 1 digit; digits that weren't subtracted get a 0.

Continuing with our example, 155 can be subtracted by 128, producing 27:

155 - 128 = 27

128	64	32	16	8	4	2	1
1							

Our new number, 27, can't be subtracted by either 64 or 32. Both of those positions get a 0. We can subtract by 16, producing 11.

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

27 - 16 = 11

128	64	32	16	8	4	2	1
1	0	0	1				

And 8 subtracts from 11, producing 3. After that, no such luck with 4.

11 - 8 = 3

128	64	32	16	8	4	2	1
1	0	0	1	1	0		

Our 3 can be subtracted by 2, producing 1. And finally, the 1 subtracts by 1 to make 0.

3 - 2 = 1
1 - 1 = 0

128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	1

We've got a binary number!

Bits, Nibbles, and Bytes

In discussing the make of a binary number, we briefly covered the length of the number. The length of a binary number is the amount of 1's and 0's it has.

4 Common binary number lengths

Binary values are often grouped into a common length of 1's and 0's, this number of digits is called the l**ength** of a number. Common bit-lengths of binary numbers include bits, nibbles, and bytes (hungry yet?). Each 1 or 0 in a binary number is called a **bit**. From there, a group of 4 bits is called a **nibble**, and 8-bits make a **byte**.

Bytes are a pretty common buzzword when working in binary. Processors are all built to work with a set length of bits, which is usually this length is a multiple of a byte: 8, 16, 32, 64, etc.

To sum it up:

Length	Name	Example
1	Bit	0
4	Nibble	1011
8	Byte	10110101

Word is another length buzzword that gets thrown out from time to time. Word is much less delicious sounding and much more ambiguous. The length of a word is usually dependent on the architecture of a processor. It could be 16 bits, 32, 64, or even more.

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

5. Padding with leading zeros

You might see binary values represented in bytes (or more), even if making a number 8-bits-long requires adding leading zeros. Leading zeros are one or more 0's added to the left of the most-significant 1-bit in a number. You usually don't see leading zeros in a decimal number: 007 doesn't tell you any more about the value of the number 7 (it might say something else).

Leading zeros aren't required on binary values, but they do help present information about the bit-length of a number. For example, you may see the number 1 printed as 00000001, just to tell you we're working within the realm of a byte. Both numbers represent the same value, however, the number with seven 0's in front adds information about the bit-length of a value.

6 Bitwise Operators

There are several ways to manipulate binary values. Just as you can with decimal numbers, you can perform standard mathematical operations - addition, subtraction, multiplication, and division - on binary values (which we'll cover on the next page). You can also manipulate individual bits of a binary value using bitwise operators.

Bitwise operators perform functions bit-by-bit on either one or two full binary numbers. They make use of boolean logic operating on a group of binary symbols. These bitwise operators are widely used throughout both electronics and programming.

7 Complement (NOT)

The complement of a binary value is like finding the exact opposite of everything about it. The complement function looks at a number and turns every 1 into a 0 and every 0 becomes a 1. The complement operator is also called NOT.

For example, to find the complement of 10110101:

NOT 10110101 (decimal 181)
-------- =
01001010 (decimal 74)

NOT is the only bitwise operator which only operates on a single binary value.

8 OR

OR takes two numbers and produces the union of them. Here's the process to OR two binary numbers together: line up each number so the bits match up, then compare each of their bits that share a position. For each bit comparison, if either or both bits are 1, the value of the result at that bit position is 1. If both values have a 0 at that position, the result also gets a 0 at that position.

The four possible OR combinations and their outcome are:

- 0 OR 0 = 0
- 0 OR 1 = 1

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

- 1 OR 0 = 1
- 1 OR 1 = 1

For example, to find the 10011010 OR 01000110, line up each of the numbers bit-by-bit. If either or both numbers have a 1 in a column, the result value has a 1 there too:

10011010
OR 01000110
-------- =
11011110

Think of the OR operation as binary addition, without a carry-over. 0 plus 0 is 0, but 1 plus anything will be 1.

9 AND

AND takes two numbers and produces the conjunction of them. AND will only produce a 1 if both of the values it's operating on are also 1.

The process of AND'ing two binary values together is similar to that of OR. Line up each number so the bits match up, then compare each of their bits that share a position. For each bit comparison, if either or both bits are 0, the value of the result at that bit position is 0. If both values have a 1 at that position, the result also gets a 1 at that position.

The four possible AND combinations and their outcome are:

- 0 AND 0 = 0
- 0 AND 1 = 0
- 1 AND 0 = 0
- 1 AND 1 = 1

For example, to find the value of 10011010 AND 01000110, start by lining up each value. The result of each bit-position will only be 1 if both bits in that column are also 1.

10011010
AND 01000110
-------- =
00000010

Think of AND as multiplication. Whenever you multiply by 0 the result will also be 0.

10 XOR

XOR is the exclusive OR. XOR behaves like regular OR, except it'll only produce a 1 if either one or the other numbers has a 1 in that bit-position.

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

The four possible XOR combinations and their outcome are:

- 0 XOR 0 = 0
- 0 XOR 1 = 1
- 1 XOR 0 = 1
- 1 XOR 1 = 0

For example, to find the result of 10011010 XOR 01000110:

10011010
XOR 01000110
-------- =
11011100

Notice the 2nd bit, a 0 resulting from two 1's XOR'ed together.

11 Bit shifts

Bit shifts aren't necessarily a bitwise operator like those listed above, but they are a handy tool for manipulating a single binary value.

There are two components to a bit shift - the direction and the amount of bits to shift. You can shift a number either to the left or right, and you can shift by one bit or many bits.

When shifting to the right, one or more of the least-significant bits (on the right side of the number) just get cut off and shifted into the infinite nothing. Leading zeros can be added to keep the bit length the same.

For example, shifting 10011010 to the right two bits:

RIGHT-SHIFT-2 10011010 (decimal 154)
-------- =
00100110 (decimal 38)

Shifting to the left adds pushes all of the bits toward the most-significant side (the left side) of the number. For each shift, a zero is added in the least-significant-bit position.

For example, shifting 10011010 to the left one bit:

LEFT-SHIFT-1 10011010 (decimal 154)
-------- =
100110100 (decimal 308)

That simple bit shift performs a complicated mathematical function. Shifts to the left n bits multiply a number by 2n (see how the last example multiplied the input by two?), while a shift in bits to the right will do an integer divide by 2n. Shifting to the right to divide can get weird - any fractions produced by the shift division will be chopped off, which is why 154

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

shifted right twice equals 38 instead of 154/4=38.5. Bit shifts can be a powerfully fast way to divide or multiply by 2, 4, 8, etc.

12 Conclusion

The binary is the building block of all computations, calculations, and operations in electronics. So, there are many places to go from here.

Now that you can convert between decimal and binary, you can apply that knowledge to understanding how characters are encoded universally: ASCII

Or you can apply your shiny new knowledge to low-level circuits and IC's:

- Digital Logic
- Shift registers

You can also have a look at how binary plays an important role in this communication protocols:

- Serial Communication
- Serial Peripheral Interface
- I2C

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.