Fundamentals of Digital Design

A story of how mathematics, logic and physical transistors combine to create computers.

Written and illustrated by Michelle Phung

2013

Digital Design / Binary Number System

The language of computers is written in binary code.

In the same way we use the letters ‘C’, ‘A’, and ‘R’ to represent the word ‘car’,
and the word ‘car’ to represent the concept of the automobile,
computers use ‘0’s and ‘1’s to represent words, instructions, rules, everything.
The two states describe an entire world of software.

For example,

if you can imagine that the numbers,
1000011 represent the letter ‘C’,
and 1000001 represent ‘A’,
and 1010010 represent ‘R’,
then you can start to imagine how binary code work.

Since binary is so simple (only two options ‘0’ or ‘1’),
the chance of misinterpreting signals is low,
and as a result, there are less likely to be errors.

Let’s start with how to read binary.

Digital Design / Binary Number System

The system that most of us are familiar with is the base 10 system (i.e., 1 2 3 4 5 6 7 8 9 0), also called decimal.

Base 16 system has 16 symbols (0 1 2 3 4 5 6 7 8 9 A B C D E F) and is also called hexadecimal.

There are only two numbers in the binary numbering system: '0' and '1'.

It is a base 2 system.

There are several ways to show what base a number is in:

sometimes a letter,
- 0x32 is in hexadecimal,
- 0b110010 is in binary,
- (decimal is most used, so it doesn't have a special letter notation)

or a subscript,
- 32_{16} is in hexadecimal (base 16)
- 110010_{2} is in binary (base 2)
- 50_{10} is in decimal (base 10)

is shown to clarify which base is being used.

Since there are only two numbers to work with, we have to interpret the value of the number by the position of the numbers.

$$001_2 = 1_{10}$$

$$010_2 = 2_{10}$$

$$011_2 = 3_{10}$$

Digital Design / Binary Number System

Since decimal is the easiest to understand,
lets start by figuring how to convert
binary into decimal.

First, realize that for every position left of the number,
the value of that number increases by 2^n.

The numbers in blue are in base 10.

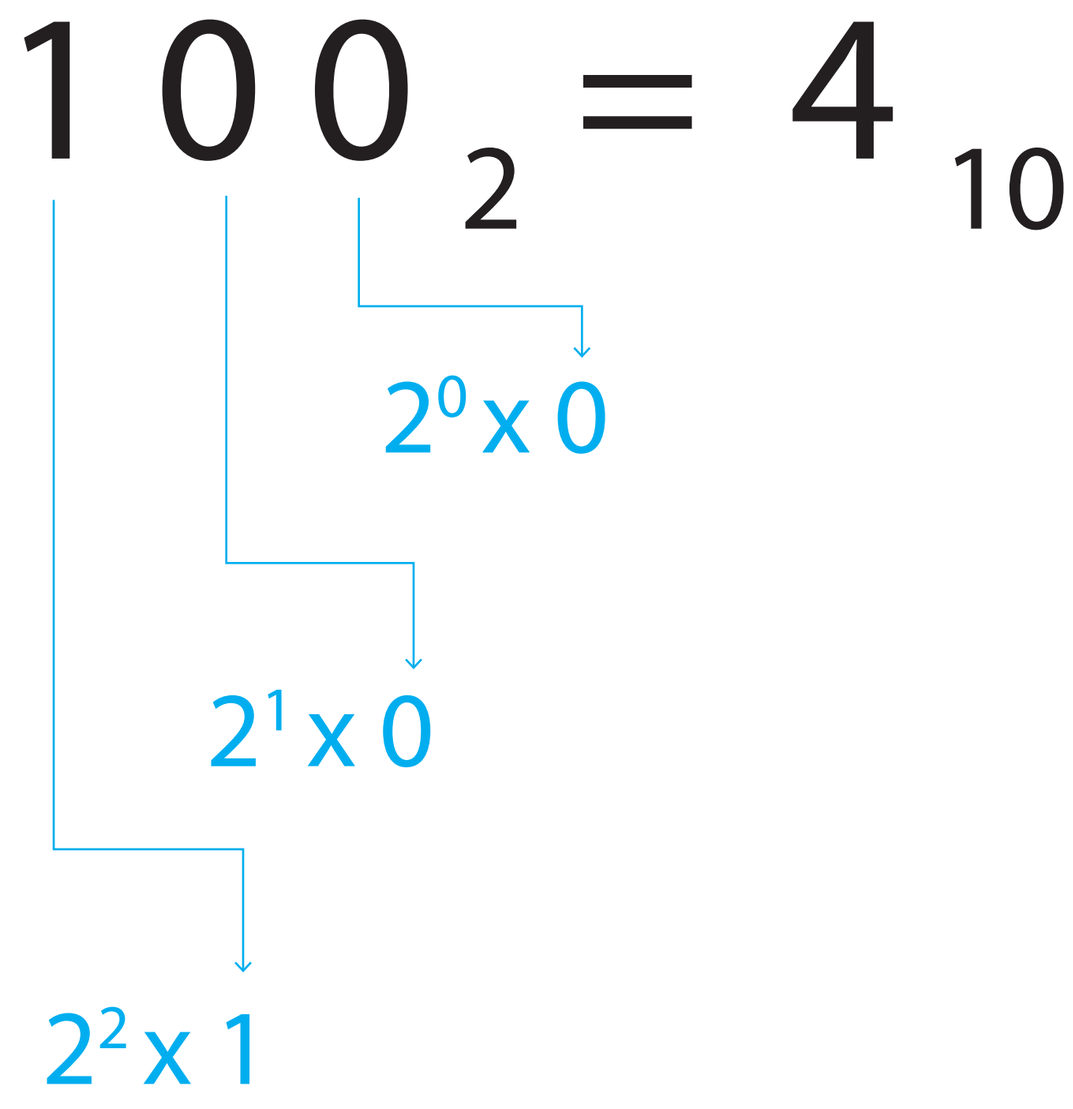
We multiply them by their position, and add the sum
to get the number in decimal.

The first position (red circle) is to be multiplied by 2^0.

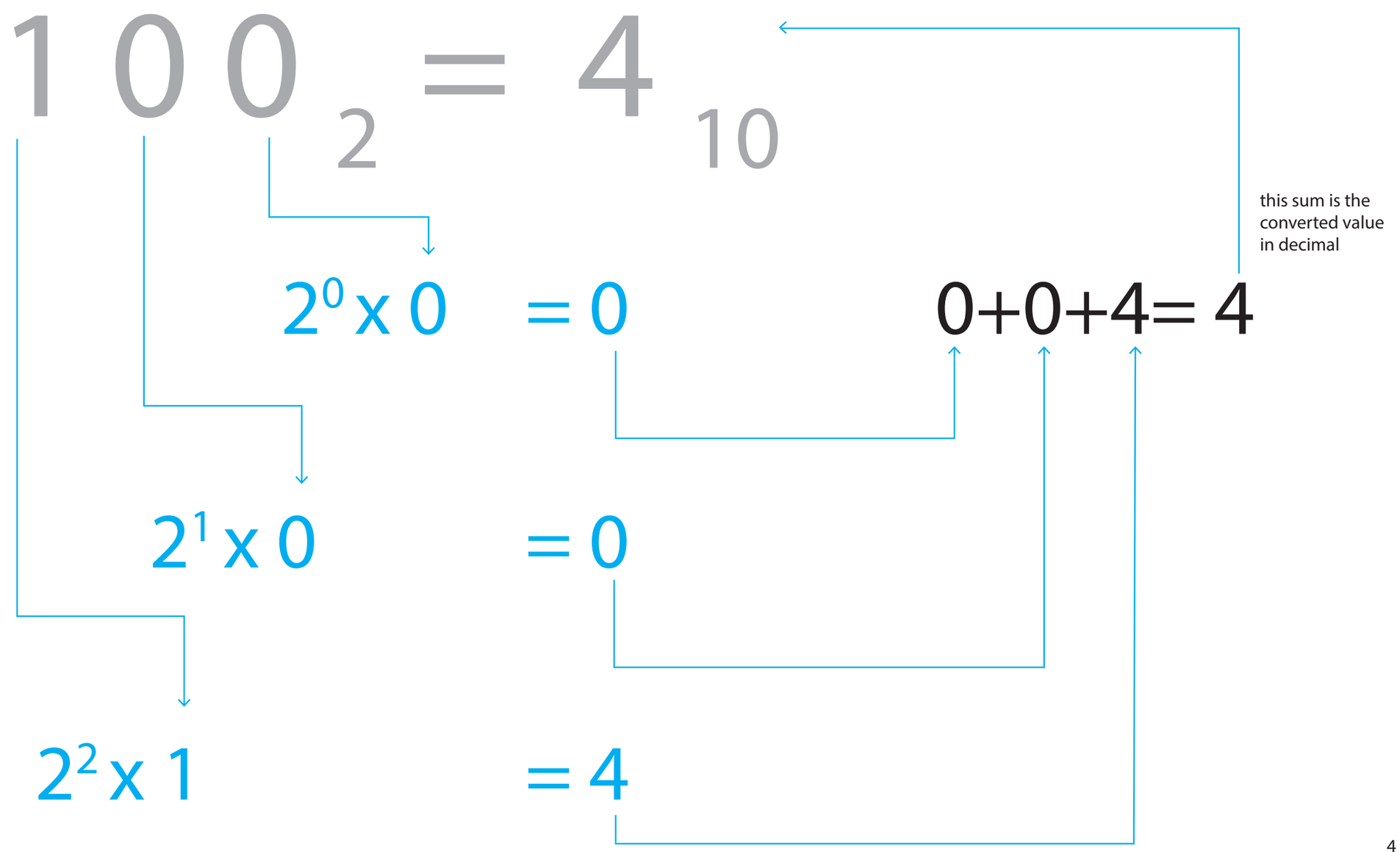
The second position (orange circle) is to be multiplied by 2^1.

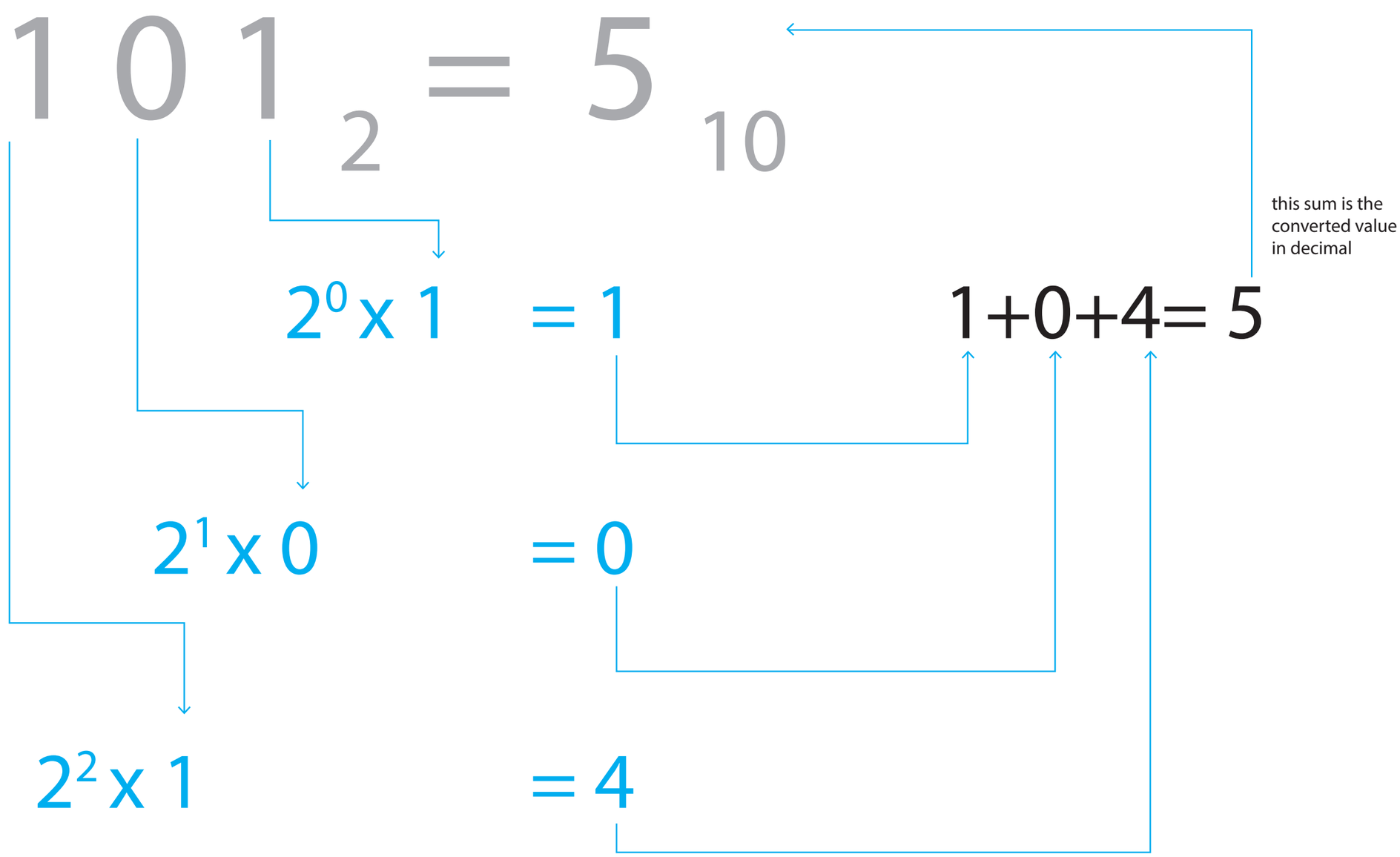
The third position (yellow circle) is to be multiplied by 2^2.

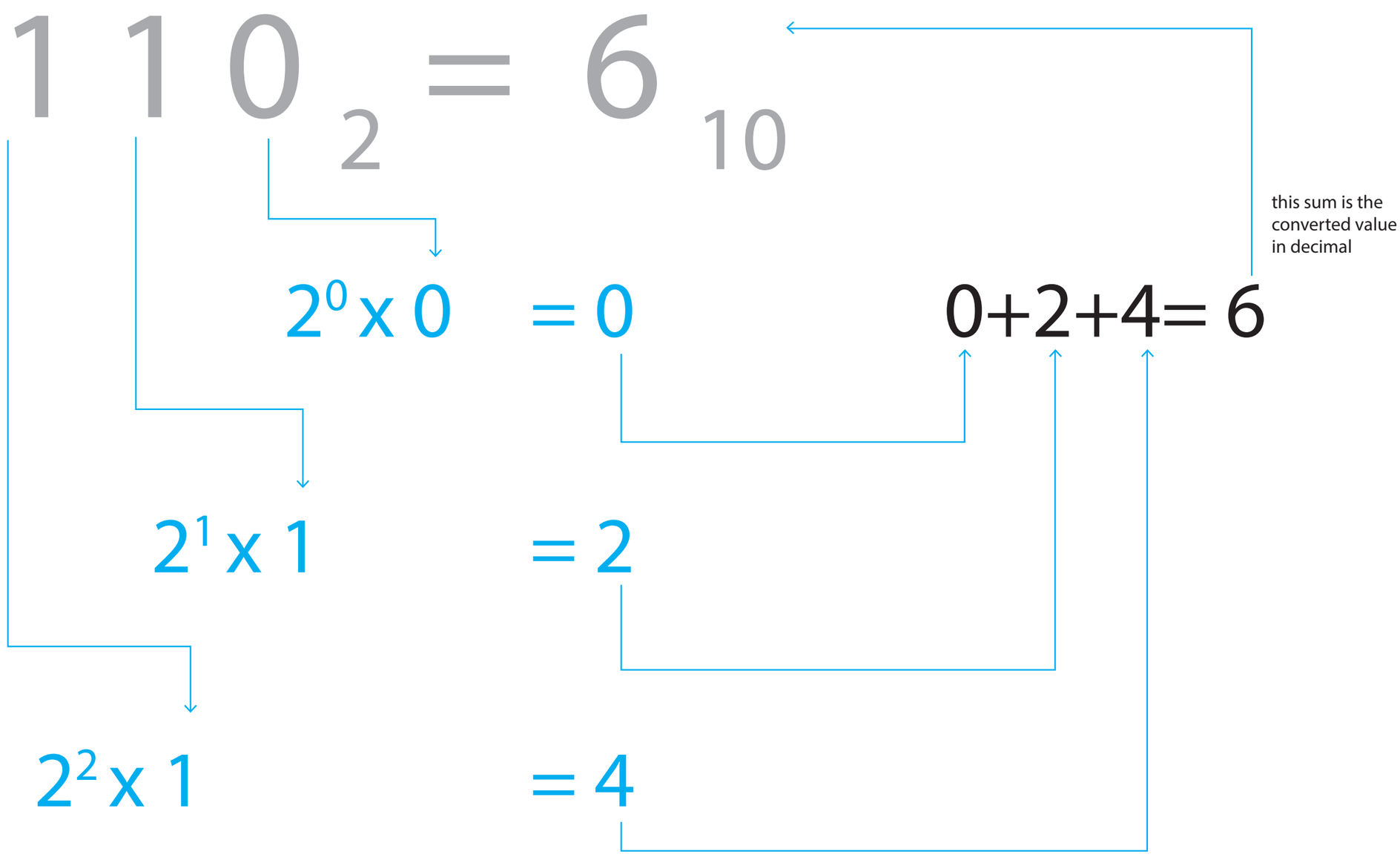
If there was a fourth position, it would be multiplied by 2^3,
and if there was a fifth position, it would be multiplied by 2^4.

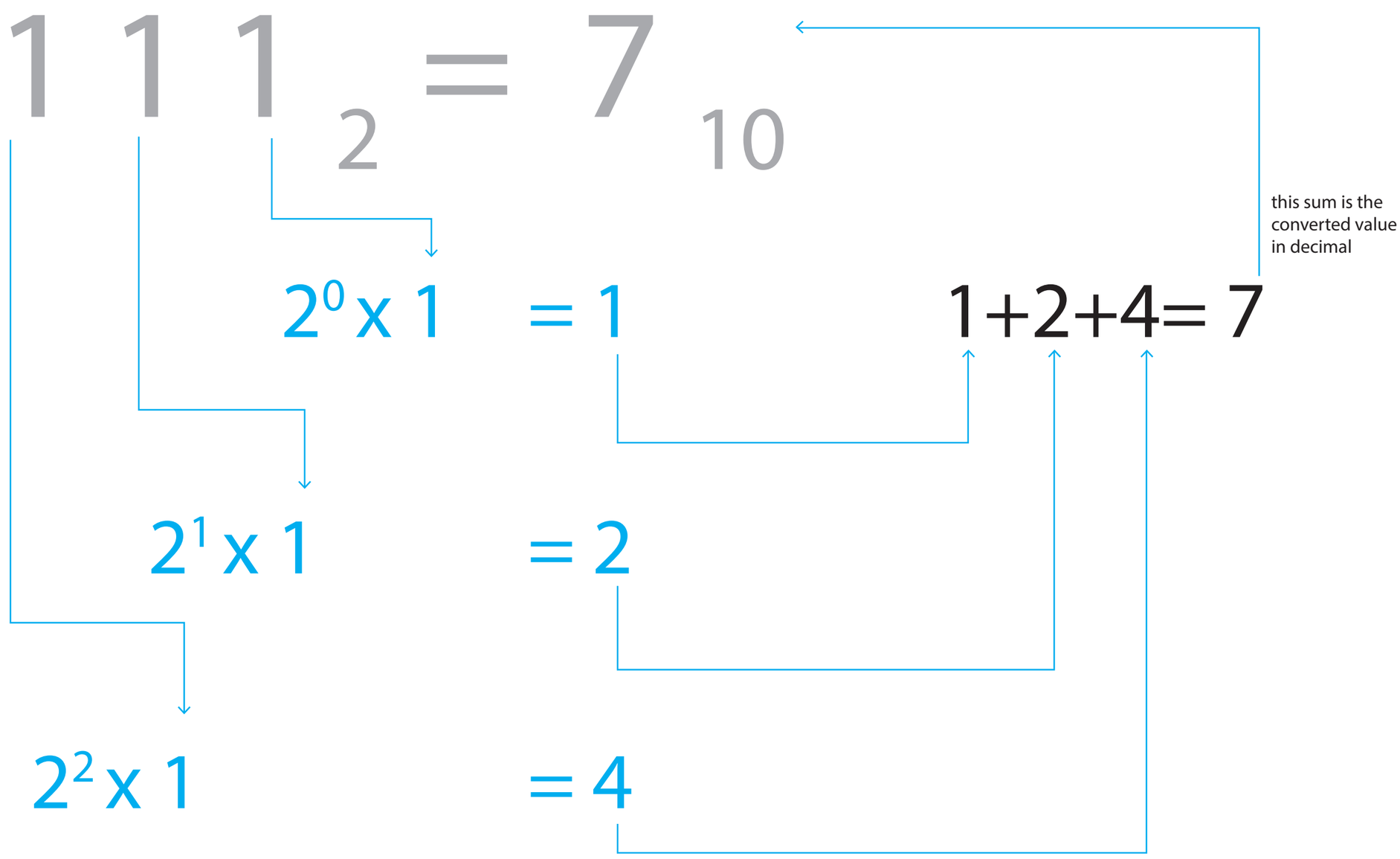


Once we get the products (0, 0, and 4), from the positions, we add them all together, and the resultant sum is the converted value in decimal.




1 0 1 2 = 5 10
this sum is the converted value in decimal
2⁰ x 1 = 1
2¹ x 0 = 0
2² x 1 = 4
1+0+4= 5


1 1 0 2 = 6 10
this sum is the converted value in decimal
2⁰ x 0 = 0
2¹ x 1 = 2
2² x 1 = 4
0+2+4= 6


1 1 1 2 = 7 10
2⁰ x 1 = 1
2¹ x 1 = 2
2² x 1 = 4
1+2+4= 7
this sum is the converted value in decimal

$1000_2 = 8_{10}$

$2^0 \times 0 = 0$

$2^1 \times 0 = 0$

$2^2 \times 0 = 0$

$2^3 \times 1 = 8$

this sum is the converted value in decimal

$0+0+0+8 = 8$

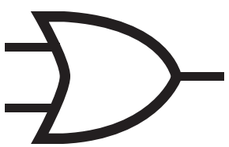
Digital Design / Logic Gates

The special nature of the binary numbering system gives way to a subdivision of algebra, called Boolean algebra (also known as Boolean Logic).

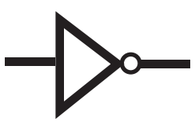
Since there are only two numbers in the binary numbering system, it lends itself to other interpretations. The two numbers, '0' and '1' can represents states, like on and off, true and false, or HIGH and LOW.

Boolean algebra has its own rules, notations, and operations.
There are three basic operations, AND, OR, and NOT.

This is the AND gate symbol.



This is the OR gate symbol.



This is the NOT gate symbol.

The gates pictured to the right are symbols that represent Boolean algebra operations. Similar to a plus or minus symbol, they represent an operation taking place that produces a single result.

These gates also represent physical electrical components, which can be used to build and control circuits, and which, in turn can be used to build complex computer systems.

The gates have inputs on the left hand side. The AND and OR gates pictured here take 2 inputs each, and result in one output.

The NOT gate takes one input (on the left hand side), and results in one output (on the right hand side).

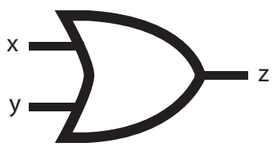
Digital Design / Logic Gates - AND, OR, NOT

There are more logic gate symbols and operations, but these are the basics.
To gain an idea of how the AND, OR, and NOT operations work, let us consider that '0' is 'false' and '1' is 'true'.

The AND gate outputs a true result only if both of the inputs are true. Otherwise, the result is false.

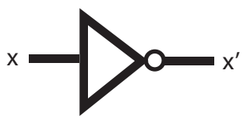
Truth Tables can quickly show the result of an operation on the inputs.
Each operation has it's own truth table. Logic gates can have more than two inputs but the most shown here are two. For larger gates that have more than two inputs, a different truth table has to be made, but the rules stay the same in each case.

input x	input y	output z	
0	0	0	if the x input is 0 and the y input is 0, then the result is 0
0	1	0	if the x input is 0 and the y input is 1, then the result is 0
1	0	0	if the x input is 1 and the y input is 0, then the result is 0
1	1	1	if the x input is 1 and the y input is 1, then the result is 1



The OR gate outputs a true signal in cases where either one of the inputs are true. It's output is only false when both inputs are false.

input x	input y	output z
0	0	0
0	1	1
1	0	1
1	1	1



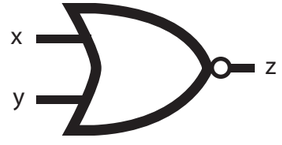
The NOT gate outputs the opposite of the input.

input	output
0	1
1	0

Digital Design / Logic Gates - NAND, NOR, XOR, XNOR

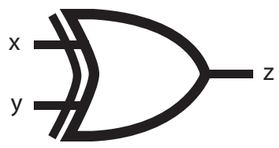
The NAND gate outputs a false result only if both of the inputs are true. Otherwise, the result is true.

input x	input y	output z
0	0	1
0	1	1
1	0	1
1	1	0



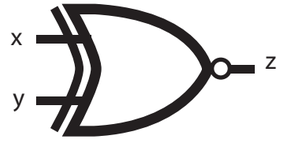
The NOR gate outputs a true signal in cases when both inputs are false. Otherwise, the result is false.

input x	input y	output z
0	0	1
0	1	0
1	0	0
1	1	0



The XOR gate outputs a true signal if the inputs are not the same.

input x	input y	output z
0	0	0
0	1	1
1	0	1
1	1	0

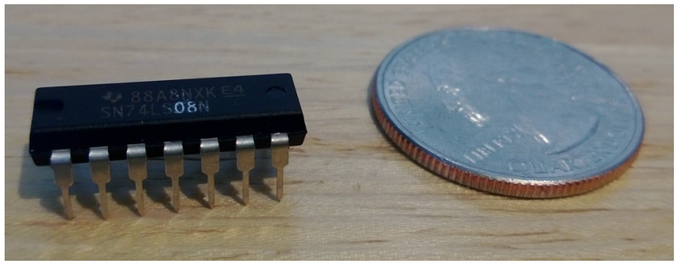


The XNOR gate outputs a true signal if both inputs are the same.

input x	input y	output z
0	0	1
0	1	0
1	0	0
1	1	1

Digital Design / Integrated Circuits

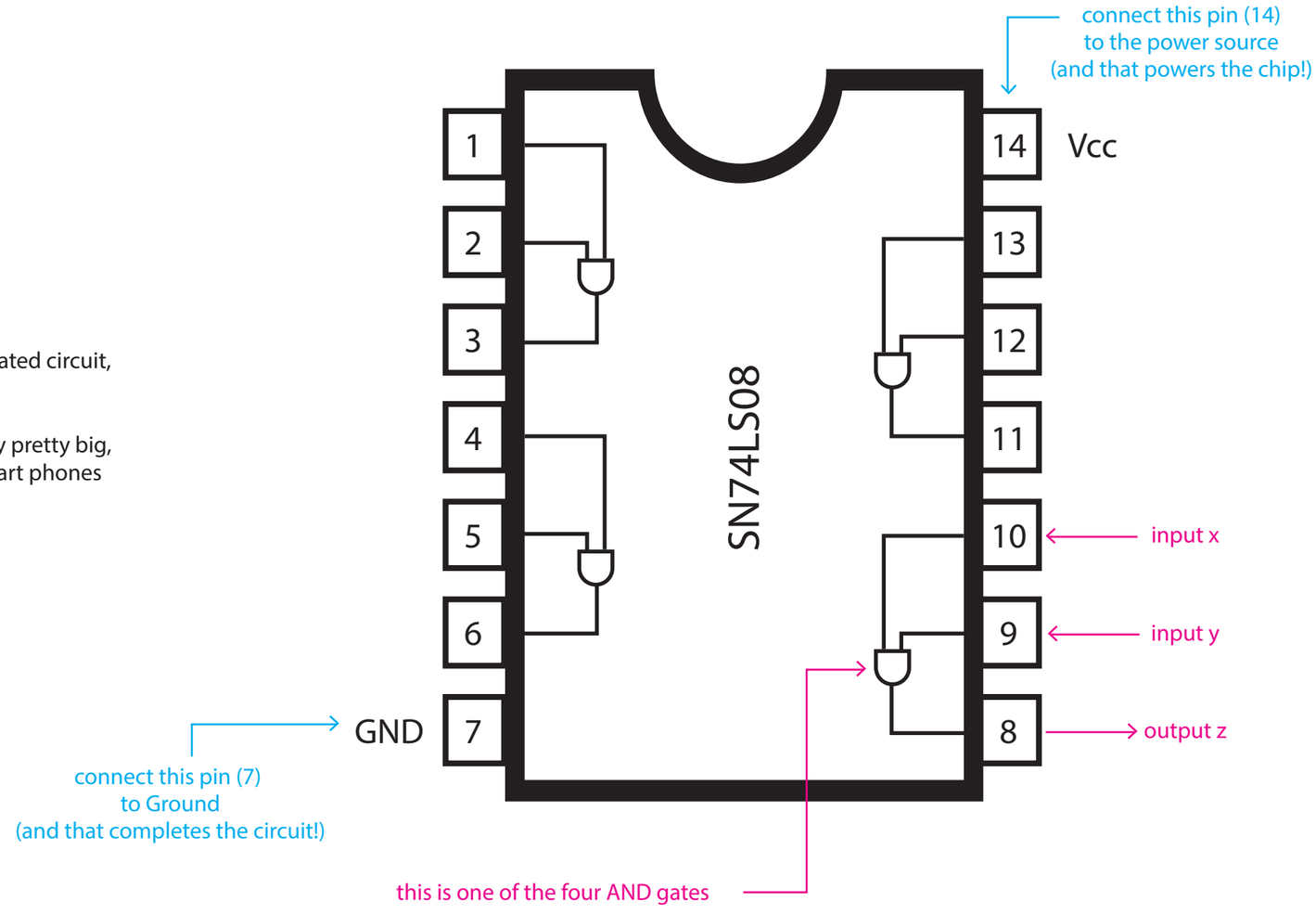
Chips in computers are simply these logic gates, arranged in a specific way, in order produce a desired output.



Pictured above (on the left) is a Texas Instrument SN74LS08 integrated circuit, which houses 4 AND gates within it.

This integrated circuit (aka IC chip or a microchip) shown is actually pretty big, relatively speaking. The ICs used in computers, calculators, and smart phones are much, much smaller.

The inside connections are pictured on the right.



Digital Design / Half-Adder

A basic circuit is a half-adder. It adds two (one digit) numbers.

The significance of a half adder being able to sum two numbers is that addition is an important instruction at a very low level of computing, and thus that importance propagates to higher levels of computing.

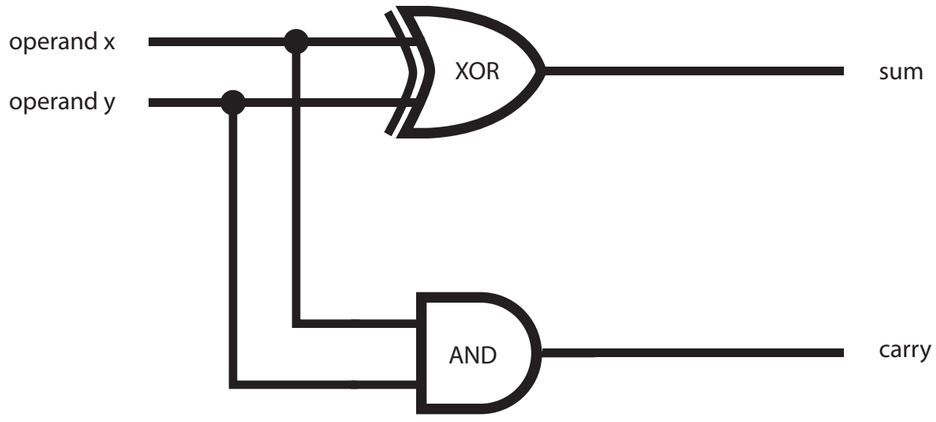
Since there are only 2 numbers in the binary system, there are only so many combinations of addition that can occur.

If operand x =0 and operand y =0 then the sum = 0.
If x=0 and y=1 then 0+1=1 (sum is 1).
If x=1 and y=1 then 1+0=1 (sum is 1).
If x=1 and y=1 then 1+1=10 (sum is 10...remember we're in binary)*.

*Please note the special case,
where x=1 and y=1

The trouble with this case is that the output of each logic gate can only have one place, meaning, it is only one digit. The output of the entire half-adder circuit is 2 outputs, with the output of the AND gate acting as the carry, or over flow, when the sum is bigger than one digit.

When this happens then carry=1 and sum=0, which, in this interpretation can be read as 10, or the correct sum of 1+1=10.



Digital Design / Multiple Digit Addition

For handling the addition of two 1-bit numbers, the half adder works great. However, the limitation of the half-adder is that it only sums 1-bit numbers.

For addition to be useful, the number of bits must be magnitudes larger. If we take a look at a common method to work out addition, we can start to recognize the issues that a circuit may have.

The order of the steps we take to add two numbers are in blue circles.

1

```
  165
+ 237
-----
```

Conventionally the sum is placed here.

2

```
  165
+ 237
-----
```

We start with the right most column, where 5+7=12.

3

```
    1
  165
+ 237
-----
    2
```

Since 12 has more than one digit, the 1 in the tens place is "carried over" to the next column, to be added into the sum of the next two numbers.

4

```
  1 1
  165
+ 237
-----
   02
```

The addition continues to move left, carrying over as necessary.

5

```
  1 1
  165
+ 237
-----
  402
```

The final sum is 402.

Digital Design / Multiple Bit Addition

Addition in binary works in the same fashion.

The order of the steps we take to add two numbers are in blue circles.

1

```
  10100101
+ 11101101
----------
```

We start with the right most column, adding the "least significant bit" (LSB) from each number together.

2

```
         1
  10100101
+ 11101101
----------
         0
```

In binary, 1+1=10.
10 has two places, so the 1 is carried over to next column.

3

```
        1
  10100101
+ 11101101
----------
        10
```

The 1 that was "carried over" to the next column, is added into the sum of the next column, 0+0+1=1.

4

```
      1 1
  10100101
+ 11101101
----------
       010
```

The addition continues to move left, carrying over as necessary.

5

```
1 1 1  1 1  1
  10100101
+ 11101101
----------
110010010
```

The final sum is 110010010.

Notice that while the two addends have 8 bits each (8 bits = 1 byte), the sum has 9 bits in it.

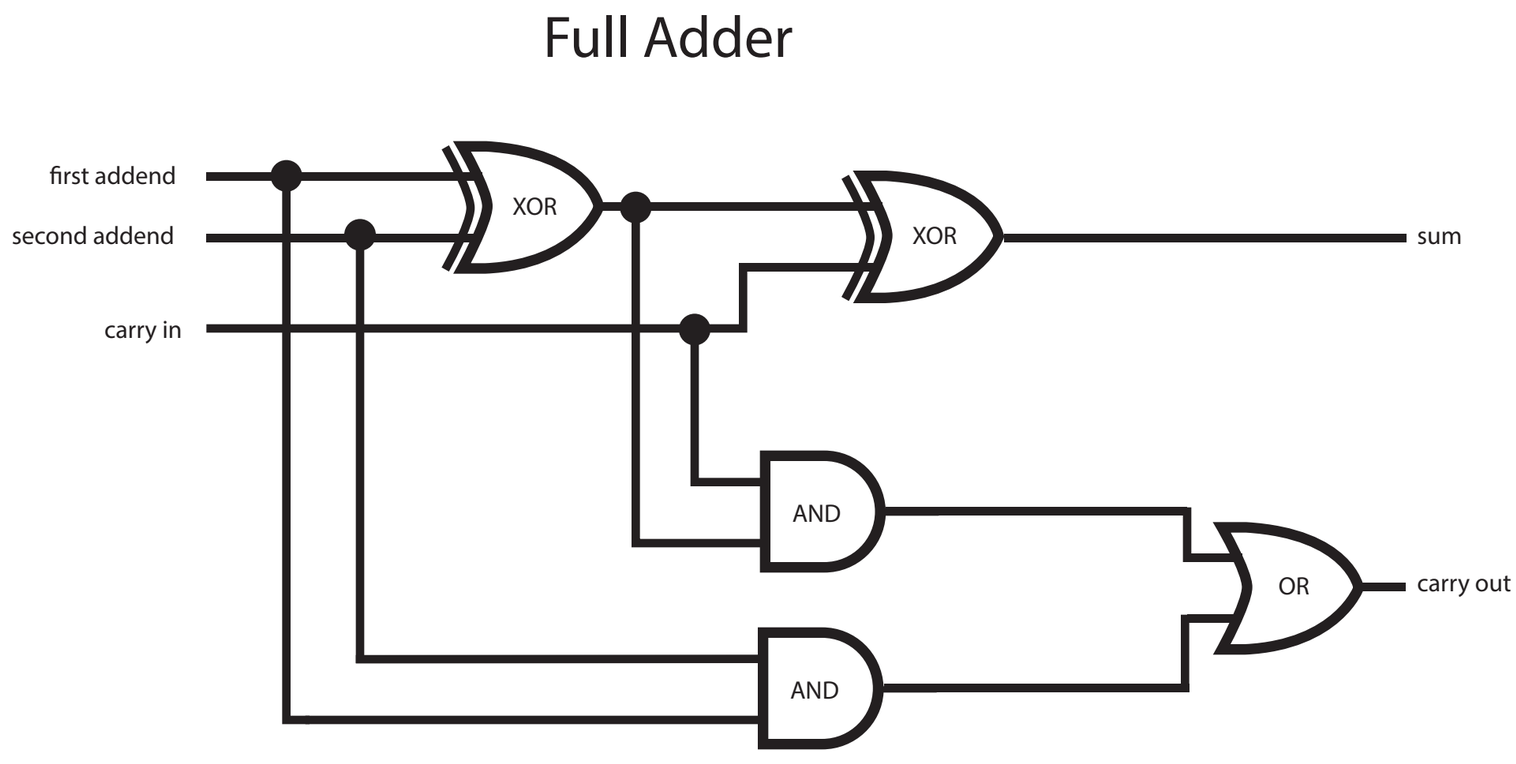
This extra 9th bit is the leftmost bit, and the left most bit is called the most significant bit (MSB)*.

*When the left most bit = MSB, we are using in "Big Endian" notation.

In "Little Endian" notation the left most bit would be the LSB.

One could imagine that chaining multiple half-adders together could produce our desired output for a multi-bit sum. However, this concept of "carrying over" is not considered when using just the half-adder.

This is where the full adder comes into play. The full-adder has an extra input called the 'carry in' or commonly denoted by 'C_{in}'.



Digital Design / Carry In and Carry Out

Look at the earlier addition diagram:

```
  10100101
+ 11101101
----------
```

notice this phantom row that gets dynamically filled with $_1$s as the resultant sum from the previous columns addition takes place

This 'phantom row' is the row in which all the carry in's get placed.
When a column has a carry over $_{(1)}$ on top of it,
the result of that row is the sum of $_3$ numbers:
the two in that column from the original two numbers, plus the additional carry over.

Let's make some distinctions between carry in and carry out:

The carry in is the $_1$ that is added to the next column of numbers,

While the carry out is the $_1$ that is the result of the sum of a column.

```
   1
  10100101
+ 11101101
----------
  10
```

this '$_1$' is the carry out of the right most column

it is also the carry in of the next column

The addition of a third addend means that there are more combinations of addition.

Note that because of the nature of the binary system, for n inputs, we know there are $_{2^n}$ unique combinations for inputs.

A truth table for a full adder reveals how the carry in can influence the result:

input A	input B	carry in	carry out	sum bit
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

*notice the sum of these bits, are represented here in binary

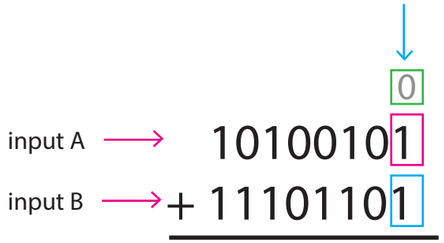
input A		input B		carry in		carry out	sum bit
0	+	0	+	0	=	0	0
0	+	0	+	1	=	0	1
0	+	1	+	0	=	0	1
0	+	1	+	1	=	1	0
1	+	0	+	0	=	0	1
1	+	0	+	1	=	1	0
1	+	1	+	0	=	1	0
1	+	1	+	1	=	1	1

It may help to imagine 'plus' and 'equal' signs in this truth table.

Digital Design / Multiple Bit Addition Redux - Part 1

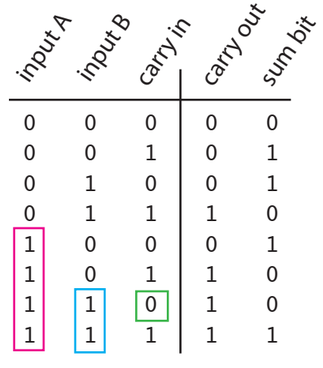
Let's take a look at binary addition again, this time keeping notice of the carry ins and carry outs.

1



We begin with the right most column, and this time, we can look at how reading the truth table could help us.

This column has :
1 for input A and
1 for input B, and
0 carry in.



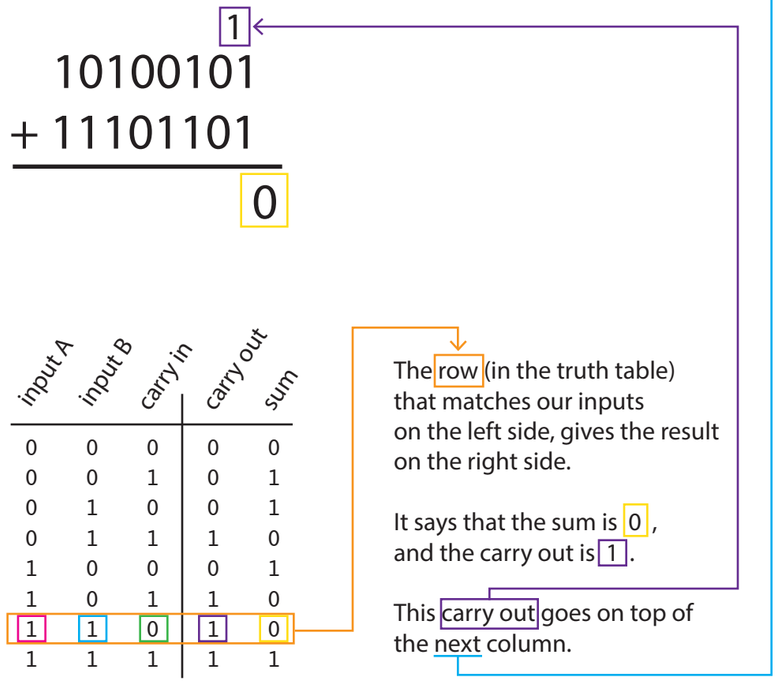
input A	input B	carry in	carry out	sum bit
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This 1 narrows down the table to the last 4 rows

This 1 narrows the table down further, we know that the row we want is in th last 2 rows

This 0 is implied (since it's the first row), so the carry in = 0. This narrows down the table to the desired row.

2



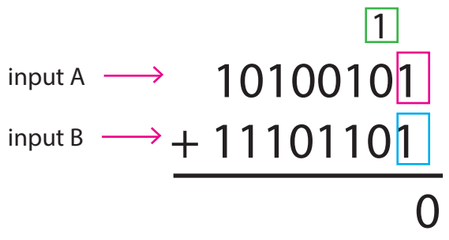
input A	input B	carry in	carry out	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The row (in the truth table) that matches our inputs on the left side, gives the result on the right side.

It says that the sum is 0, and the carry out is 1.

This carry out goes on top of the next column.

3



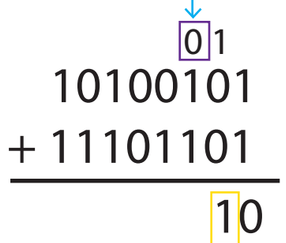
To reiterate:
the 1 that was "carried over" to the next column, is added to the sum of the next column.

Now we do addition to the next column (indicated above with the blue arrow).

Again, we can use the truth table, but realize that the carry out from the last computation, is now the carry in this addition.

input A	input B	carry in	carry out	sum bit
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

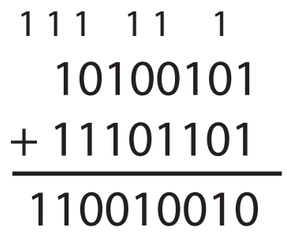
4



From the truth table, the sum bit = 1 , (0+0+1=1), and the carry out bit = 0.

This continues left, carrying over as necessary. When the carry out bit = 0, it is usually omitted, and understood that 'nothing' is added.

5



The final sum is 110010010.

Digital Design / 4-bit Adder Introduction

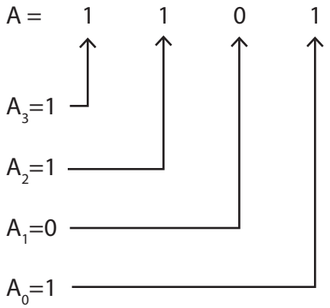
Building 4 bit adder gives context to how adders work in a system.

A 4-bit adder adds two 4-bit numbers. It has four, and only four, places in each number. This means that the minimum number that one of the addends can be is 0000. The maximum an addend can be is 1111_2. (recall: $1111_2 = 15_{10}$)

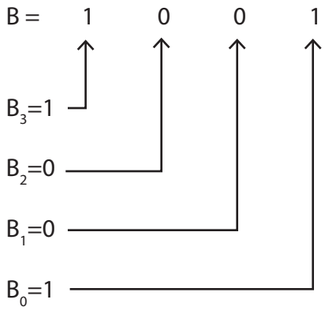
An example:

A=1101
B=1001

Since the adder is going to sum the LSB of A with the LSB of B, and then every corresponding bits after that, it is common to name each bit.



Similarly for B:



So when we add them together, roughly, it would look something like this:

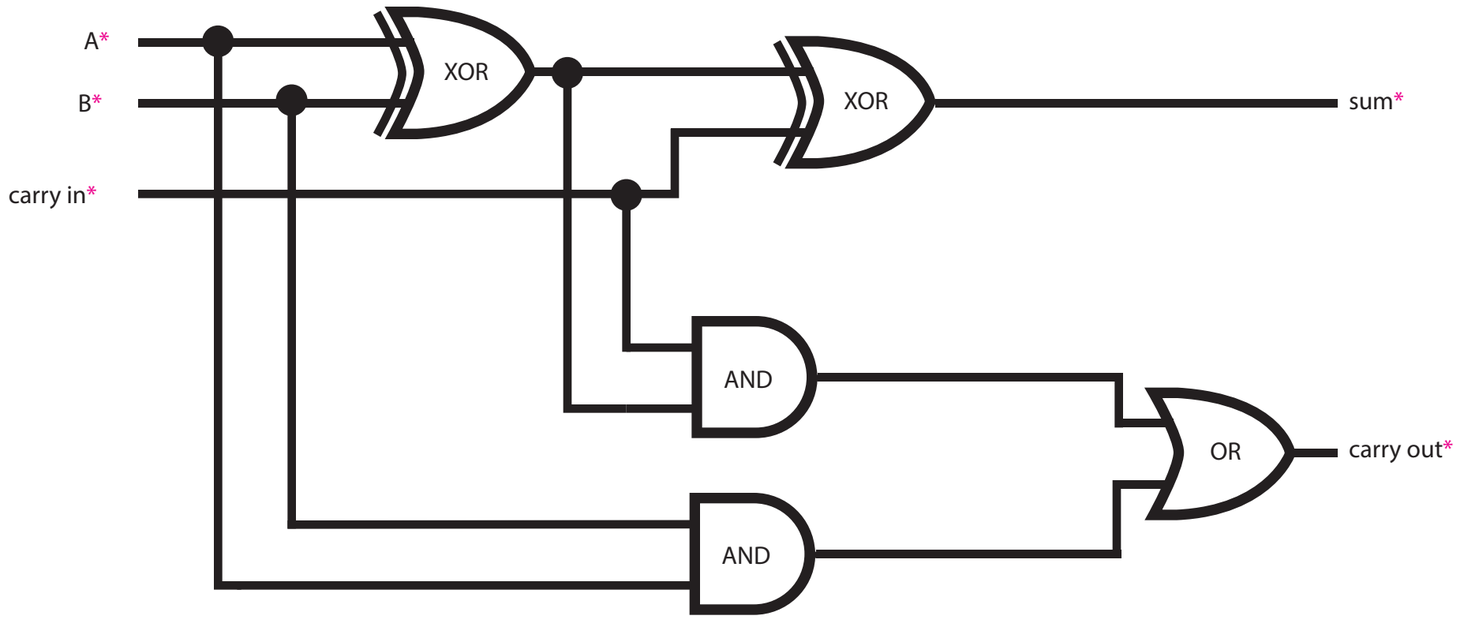
Carry out_3	Carry out_2	Carry out_1	Carry out_0	0
0	A_3	A_2	A_1	A_0
+ 0	+ B_3	+ B_2	+ B_1	+ B_0
Carry out_3	Sum_3	Sum_2	Sum_1	Sum_0

Notice that the "carry-outs" are the "carry-ins" for the next column.

It may help to realize:
$\text{Carry in}_1 = \text{Carry out}_0$
$\text{Carry in}_2 = \text{Carry out}_1$
$\text{Carry in}_3 = \text{Carry out}_2$
$\text{Carry in}_4 = \text{Carry out}_3$ and this is the final carry out, it represents the MSB of the sum, it is exceptional because for adding two numbers with 4 places (4-bits), the sum can be a 5-bit number (it has 5 places).

Digital Design / 4-bit Adder - Each Bit Counts

Revisiting the full adder again:



And the previous equations:

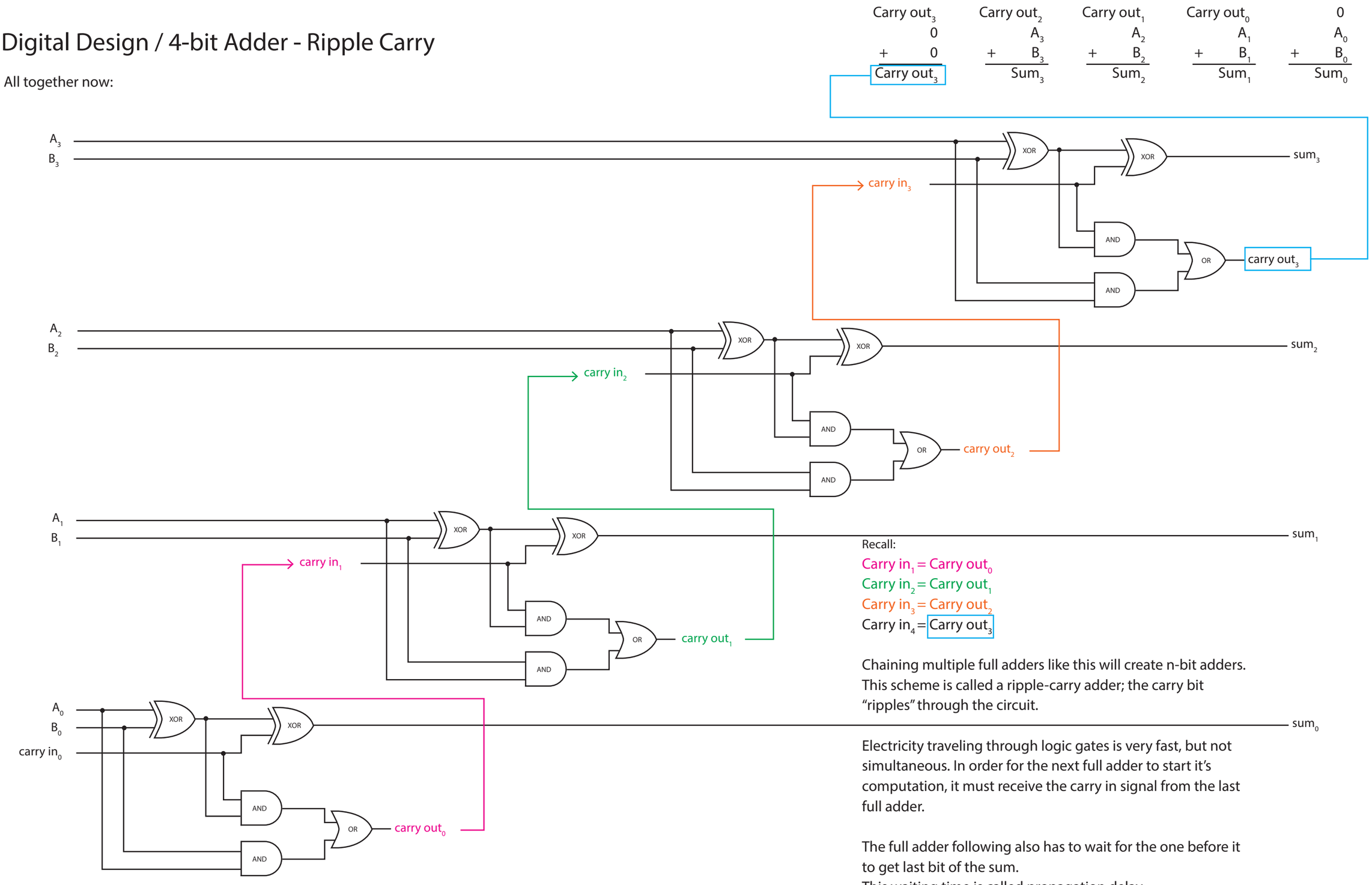
$$\begin{array}{r} \text{Carry out}_3 \\ 0 \\ +\quad 0 \\ \hline \text{Carry out}_3 \end{array} \qquad \begin{array}{r} \text{Carry out}_2 \\ A_3 \\ +\quad B_3 \\ \hline \text{Sum}_3 \end{array} \qquad \begin{array}{r} \text{Carry out}_1 \\ A_2 \\ +\quad B_2 \\ \hline \text{Sum}_2 \end{array} \qquad \begin{array}{r} \text{Carry out}_0 \\ A_1 \\ +\quad B_1 \\ \hline \text{Sum}_1 \end{array} \qquad \begin{array}{r} 0 \\ A_0 \\ +\quad B_0 \\ \hline \text{Sum}_0 \end{array}$$

*Note that each of the inputs and outputs are 1 bit signals.

Paired with each equation, we would need 4 full adders to create the 5-bit wide sum (C_{out}, Sum_3, Sum_2, Sum_1, Sum_0).

Digital Design / 4-bit Adder - Ripple Carry

All together now:



Recall:

Carry in_1 = Carry out_0

Carry in_2 = Carry out_1

Carry in_3 = Carry out_2

Carry in_4 = Carry out_3

Chaining multiple full adders like this will create n-bit adders. This scheme is called a ripple-carry adder; the carry bit "ripples" through the circuit.

Electricity traveling through logic gates is very fast, but not simultaneous. In order for the next full adder to start it's computation, it must receive the carry in signal from the last full adder.

The full adder following also has to wait for the one before it to get last bit of the sum.
This waiting time is called propagation delay.

Digital Design / 4-bit Adder - Carry Look Ahead

The propagation delay from the time the adder starts obtaining the first carry to the time it gets the last carry out in a ripple carry adder means that for an n-bit adder, the n-th adder must wait n times the propagation delay of one full adder. For example, if there are 64-bit numbers being added together, the last bit and its carry-out would have had to wait 64 times the propagation delay of each of the full adders.

This lag is exacerbated when larger number of bits are being added together.

There are other types of adders. The most efficient currently is called the Carry-Look Ahead Adder. The trade off with the Carry-Look Ahead adder is that it uses many more logic gates than the Ripple-Carry Adder, but is much faster.

The reason that the Ripple-Carry adder is slow is because the carrys must ripple through the circuit, one at a time, as this is how the circuit is set up. However, if the circuit could recieve all of carrys at the same time, then all the bits of the sum would occur at once, instead of obtaining them one bit at a time.

To get all the carry-outs at the same time, we can rely on the following equations:

$c_1 = g_0+p_0c_0$
$c_2 = g_1+p_1c_1$
$c_3 = g_2+p_2c_2$
$c_4 = g_3+p_3c_3$

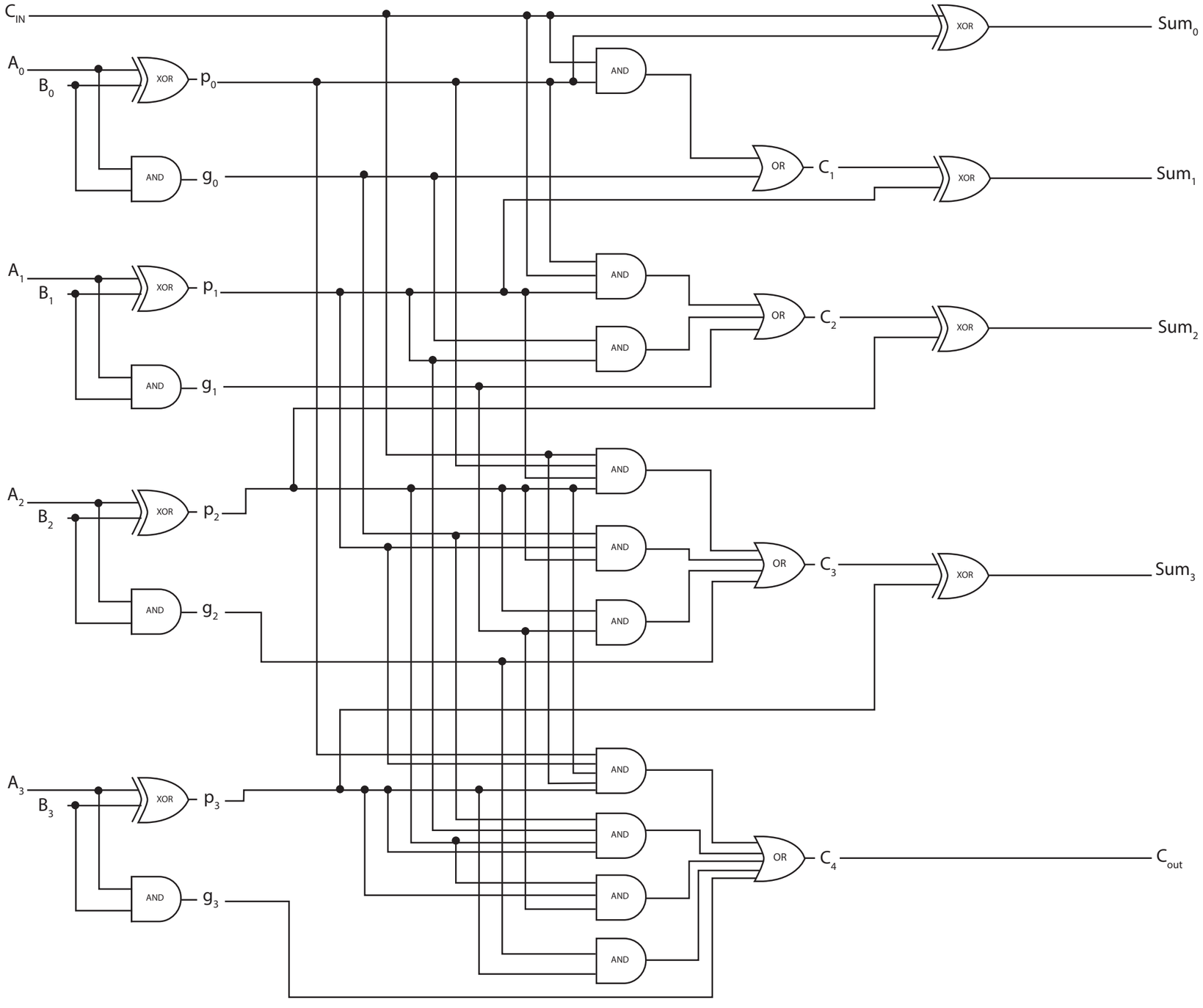
Note that all of the equations follow the form: $c_{i+1}=g_i+p_ic_i$
where :

i goes from 1 to 4,
c_0 is the intial carry in,
g_i is called the 'generate' term, obtained from A_i AND B_i
p_i is called the 'propagate' term, obtained from A_i OR B_i

It looks as though each c_{i+1} would depend on c_i, indicating a delay, but since all the p's and g's are received simultaneously, the equations can be expanded:

$c_1 = g_0+p_0c_0$
$c_2 = g_1+p_1c_1 = g_1+p_1(g_0+p_0c_0)$
$c_3 = g_2+p_2c_2 = g_2+p_2(g_1+p_1(g_0+p_0c_0))$
$c_4 = g_3+p_3c_3 = g_3+p_3(g_2+p_2(g_1+p_1(g_0+p_0c_0)))$

Digital Design / 4-bit Adder - Carry Look Ahead Adder



This is a diagram of a Carry-Look Ahead 4-bit Adder.

When there are more bits being added, say in a 16-bit or 32-bit adder, the number of logic gates increases, but it takes the same amount of time to arrive at the sum.