Module 0283: Adding binary numbers

Tak Auyeung, Ph.D.

January 1, 2024

Contents

1 About this module

- Prerequisites: 0282
- Objectives: This module takes a look at how to add binary numbers.

2 Adding, in general

Let us start with something that is seemingly familiar. What is $2 + 5$ in base 10? The answer is, obviously, 7! How do we know this? When we were young, we use analog or physical ways to learn that the quantity of 2, added to the quantity of 5 has a total of the quantity of 7. We counted blocks, measured lengths and did all kind of activities to learn the concept.

How, let us make this more difficult. What is $7 + 5$ in base 10? Ah, the answer is 12. Well, twelve is the correct answer. However, note how the sum needs one digit more than the numbers being added? This is the concept of carrying, and that took us a while to learn as well.

Carrying, or having a carry of non-zero, happens when the sum is a quantity that needs one more digit to represent. With single digit base 10 addition, this happens whenever the sum is at least 10.

How did learn about the sum and the carry of all the single digit additions? Although it may not be clear to us now, we did this by memorization. We have $7 + 9 = 16$ when we were young. It took us a bit of time to master addition because the table is somewhat big.

Generally speaking, in base-10, given two single digits, `x` and `y`, we can define the following functions to compute the carry `c` and the single digit sum `r`:

```c
unsigned c(unsigned x, unsigned y)
{
  return (x+y >= 10) ? 1 : 0;
}
```

```

unsigned r(unsigned x, unsigned y)
{
  return (x+y) % 10;
}
```

Here is a quick question: what happens when we want to change from base-10 to some other base?

3 You will like binary addition

Adding numbers in base 2 is, by far, easier. It only seems difficult because we are stuck with the old and clumsy way of adding base 10 numbers!

There are $2^2 = 4$ rules to member, as opposed to $10^2 = 100$ in the case of base 10:

- $0_2 + 0_2 = 0_2$. The suffix of 2 indicates these numbers are in base 2. Well, this rule hardly needs any explanation. Zero represents nothing in any base, so the sum of itself should remain as nothing.
- $0_2 + 1_2 = 1_2$. This one is easy, too. 1 means 1 in any base. When you add nothing to 1, you get 1 back.
- $1_2 + 0_2 = 1_2$. This is just the reverse of the previous one, the same reasoning applies.
- $1_2 + 1_2 = 10_2$. Ah, finally something that is interesting. You have been expecting something that is different from base 10 addition, and here it is. Or, is it?

The sum of the quantities one and one is two, this is true regardless of the base that we choose for the representing numbering system. So the real question is why is the quantity of two represented as 10_2.

Recall that in any constant-base number, each digit represents the quantity of a power of the base. In this case, the 0 represents there is none of $2^0 = 1$. The 1 represents there is one of $2^1 = 2$.

You can also look into base 10 numbers to understand this. What is $5 + 5$ in base 10? It is 10 because the quantity of 10 requires two digits.

This is also related to carrying. You may say that $1_2 + 1_2$ is the only time when there is a carry of 1 in base 2 addition. However, it is better to say that $1_2 + 1_2$ is the only time when the carry is a 1, and for the other 3 cases, the carry is a 0.

We are separating the result of adding into two parts, x and y. The “result (single digit sum)” $r(x, y) = (x + y) \mod b$is the single digit result of addition of base-b, and the “carry” in base-b $c(x, y) = (x + y >= b)?1 : 0$ indicates whether the sum needs one more digit to represent it. We can, then, summarize binary addition as follows:

x	y	$r(x,y)$	$c(x,y)$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

As an example in base 10, $6 + 7$ has the following two values: $r(6, 7) = 3$ because that is the single digit of the result, and $c(6, 7) = 1$ because there is a carry of 1. Of course, from here on, the r and c functions are strictly for base 2 additions.

4 Multiple digit addition

Single digit addition is the basis of multi-digit addition. Let us first examine base 10 addition first, then we will look into binary addition.

The functions r and c are useful, but they need to be combined to perform multi-digit addition. This is because in multi-digit addition, regardless of the chosen base, the carry from a less significant digit needs to be added to the final sum of a particular digit.

Let us first consider a base-10 example of $467 + 78$. $7 + 8$ has a result of 5 and a carry of 1. Because in this case, this is the least significant digit, there is no carry to consider as the least significant digit of the sum. As a result, the least significant digit of the sum is just 5.

The second least significant digit of the sum, however, has to additions. The first one is $6 + 7$ (second least significant digit from the two input numbers), yielding a carry of 1 and a result of 3. However, this digit is not done yet because we need to add the carry from the addition of the least significant digit. This brings the final result of the second least significant digit of the addition from 3 to 4.

The most significant digit has two additions. The first one is $4 + 0$, the 0 being the implied number of hundreds of the second number. However, remember that we also have a carry from the second least significant digit. This brings the final result of the most significant digit from 4 to 5.

As a result, the sum of $467 + 78$ is 545.

We learned how to do this a long time ago. For our current discussion, we need to abstract the process of multi-digit addition using some symbols.

Let us name the first addend (number to be added, input to addition) x, and the second addend y. The final sum is s. We will use the notion $x[i]$ to identify the i-th digit of x in this case.

Two new constructs are introduced to express multi-digit summation. Let $k[i]$ be the carry ("karry") contributed $\underline{\text{to}}$ digit i. Let $q[i]$ be the result of just $r(x[i], y[i])$.

As an exercise, work out the digits for the following base-10 additions:

- $456 + 78$
- $999 + 204$
- $928 + 108$

Now that we are more familiarized with the notation, let us figure out what is going on, especially with the carry digits.

First of all, $q[i] = r(x[i], y[i])$ by definition. This means that the q digits have nothing to do with carrying. Given that we are only working with integers, we also know that $k[0] = 0$ because nothing contributes to the carry of digit 0 (the least significant digit).

$k[1]$ is rather easy because only the addition of $x[0]$ and $y[0]$ can carry to digit 1. But this means $s[1] = r(q[1], k[1])$ because the carry from digit 0 needs to be considered. The trickiest part is $k[2]$. There $\underline{\text{two}}$ potential sources to contribute to $k[2]$. The first is $c([x[1], y[1])$. The second one is $c(q[1], k[1])$.

When we are only adding two numbers, it is guaranteed that only at the most one of the sources can be a 1. It is, therefore, OK to define $k[2] = c(x[1], y[1]) + c(q[1], k[1])$.

In fact, we can "cascade" this and generalize the following:

- $k[0] = 0$. We will challenge this assumption later, but for now it is okay.
- $q[i] = r(x[i], y[i])$, this is due to the definition of the p digits.
- $k[i + 1] = c(x[i], y[i]) + c(q[i], k[i])$ in general.
- $s[i] = r(q[i], k[i])$ in general.

Does this match the result of the exercise that you did earlier?

Remember that up to this point, we are using base-10 numbers. Base-10 is nasty because it has pretty big r and c tables to memorize. Although we don't think of it this way, we do memorize that $r(6, 7) = 3$, $c(6, 7) = 1$ and so on for all pairs of single digit.

The r and c look up tables for base-2 is already discussed earlier. Each one only has 4 entries! The best part is what we generalize for multi-digit addition works for binary numbers, too. All you need to do is to switch the r and c functions to the base-2 versions, and make sure x and y are represented in binary (each digit is either a 0 or a 1).

Now, using the binary number r and c functions, figure out the following:

- $1101_2 + 0110_2$
- $1111_2 + 0001_2$
- $1010_2 + 0101_2$

5 But how does the computer do this using transistors?

Recall from 0281 that transistors can make logic gates that implement all the logical/boolean operators that we use. The big question now is how do transistors perform addition.

When you refer back to the table that describes r and c for binary numbers, let us example each column individually.

The easier one is $c(x, y)$, let's deal with this one first. See how the only time $c(x, y) = 1$ is when $x = 1, y = 1$. Doesn't this remind you of a certain boolean operator? That's right, conjunction, or logical-and, does exactly this! This means we can define $c(x, y) = xy$ (the notation of multiplication means conjunction in boolean algebra).

The other one is a little harder. $r(x, y) = 1$ in two cases, when x and y are different. Technically, there is a boolean operator called exclusive-or that does exactly this. However, exclusive-or is not a very common operator. We can define $r(x, y) = x!y+!xy$. In this notation, I am borrowing $!x$ to mean the logical negation (not) of x from C/C++ syntax. The plus symbol means disjunction (logical-or) in boolean algebra. The priority of operation is that negation has the highest priority, conjunction has the second highest priority, and disjunction has the lowest priority.

The one problem needs to be solved. $k[i + 1] = c(x[i], y[i]) + c(q[i], k[i])$ has an arithmetic addition in it, and that is not a logical operation. This can be fixed by switching the semantics of the additional symbol from arithmetic addition to disjunction. This can be done because we know that only up to one of the components, $c(x[i], y[i])$ or $c(q[i], k[i])$ can be a 1.

With this, we can technically implement multi-digit addition in transistors because both r and c are now expressed in common boolean operators, and from module 0281 we know transistors can make nand gates, and all the common boolean operators can be implemented by nand gates.

5.1 Exercise: implement a 3-bit by 3-bit binary adder using and/or/not gates in LogiSim

To get this done, it is convenient to first create a half-adder component. Logisim allows the definition of components so that these components can be used as "classes" to instantiate actual components. Kind of like subroutines or classes in programming.

A half adder has the following inputs: one x bit and one y bit as input, one r bit as the result (corresponding to $r(x, y)$, and one c bit as the carry (corresponding to $c(x, y)$).

Each column of adding manually needs two half adders (hence the name half adder). Two half adders combine to one full adder. As a single unit, a full adder has 3 inputs:

- a bit x from one number being added
- a bit y from the other number being added
- a bit c_i as a carry-in from the less significant digit

There are two outputs from a full adder:

- a bit c_o as a carry-out to the next digit
- a bit s as the actual sum bit for this column/bit

How do we wire up the two half adders in a full adder? Refer back to the equations in the previous section to get an idea. Furthermore, as a hint, a single or-gate is needed in addition to two half adders in a full adder.

Once a full adder is implemented (as a template or component in Logisim), it is time to "string" full adders together to make a multi-bit adder. How do we wire up the full adders in a multi-bit adder? Again, the equations from the previous section will help you find the answer.

Two big questions remain. What is the c_i of bit 0, and c_o of bit 2? Bit 0 has no less significant digits, there are usually two approaches. You can wire up a constant of 0 to c_i of the bit 0 full adder, or leave it as an input pin. Leaving it as a input pin gives you the flexibility of using it later on, and also to specify a value of 0 when that is appropriate.

c_o of bit 2 has no where to go because the width of a 3-bit adder only has 3 bits. As a result, c_o of bit 2 (or whatever the most significant bit is) connects to an output pin as a part of the output of the 3-bit adder. In other words, a 3-bit adder has a sum of 3 bits, but it also has an extra carry bit output to specify whether the sum is the actual value or whether there is a final carry that is not a part of the 3-bit sum.

6 But wait, there is more!

Because the $k[i]$ digits are defined explicitly, the overall design of an adder is quite structured because the same $k[i]$ digit can be used to compute $k[i+1]$ as well as $s[i+1]$. However, because $k[i+1]$ depends on $k[i]$, an adder cannot be very fast. The results of the carries must be propagated from the least significant digits to the most significant digits sequentially.

We want to perform addition operations as quickly as possible. But how do we do this?

The main problem has to do with the carry bits. The first step of the solution is to expand the term $k[i]$ in the definition of $k[i+1]$ so that the computation of $k[i+1]$ can utilize the other bits ($x[i]$ and $y[i]$) directly to avoid the cascading.

The use of brackets [] is going to make the equations very messy. We are now switching to use of subscript instead. This means that $x[i]$ is now written as x_i.

The following shows the expansion of the c function to compute k_{i+1} (note that these are all boolean expressions!):

$$\begin{aligned}
k_{i+1} &= c(x_i, y_i) + c(q_i, k_i) && \text{definition} \\
&= x_i y_i + q_i k_i && \text{binary c function} \\
&= x_i y_i + r(x_i, y_i) k_i && \text{definition of q} \\
&= x_i y_i + (x_i!y_i + !x_i y_i) k_i && \text{definition of r} \\
&= x_i y_i + x_i!y_i k_i + !x_i y_i k_i && \text{distribution} \\
&= x_i y_i (1) + x_i!y_i k_i + !x_i y_i k_i && \text{identity} \\
&= x_i y_i (1 + k_i) + x_i!y_i k_i + !x_i y_i k_i && \text{1+x=1} \\
&= x_i y_i + x_i y_i k_i + x_i!y_i k_i + !x_i y_i k_i && \text{identity and distribution} \\
&= x_i y_i + x_i y_i k_i + x_i!y_i k_i + x_i y_i k_i + !x_i y_i k_i && \text{x+x=x} \\
&= x_i y_i + (x_i y_i k_i + x_i!y_i k_i) + (x_i y_i k_i + !x_i y_i k_i) && \text{commutative} \\
&= x_i y_i + (x_i k_i)(y_i + !y_i) + (y_i k_i)(x_i + !x_i) && \text{factoring} \\
&= x_i y_i + (x_i k_i)(1) + (y_i k_i)(1) && \text{x+!x=1} \\
&= x_i y_i + x_i k_i + y_i k_i && \text{identity} \\
&= x_i y_i + (x_i + y_i) k_i && \text{factoring}
\end{aligned}$$

After all these steps, we still end up with k_i! However, the equation is cleaned up a bit. At this point, we define $g_i = x_i y_i$ and $p_i = x_i + y_i$. These two definitions are not necessary, but they will make the rest of the steps easier.

With these terms defined, $k_{i+1} = g_i + p_i k_i$. Our objective is to get rid of k_i on the right hand side by expanding it.

At this point, we will try some values out and see if there is a pattern.

The recursive definition ends with k_0 because it depends on no other carries. In fact, it is usually assumed zero, but not always.

The next one up is $k_1 = g_0 + p_0 k_0$.

Then we have

$$\begin{aligned}
k_2 &= g_1 + p_1 k_1 \\
&= g_1 + p_1 (g_0 + p_0 k_0) \\
&= g_1 + p_1 g_0 + p_1 p_0 k_0
\end{aligned}$$

It is important to see how the "propagation"effect seems to be gone. We will work out one more term:

$$\begin{aligned} k_3 &= g_2 + p_2 k_2 \\ &= g_2 + p_2(g_1 + p_1 g_0 + p_1 p_0 k_0) \\ &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 k_0 \end{aligned}$$

There are several important observations. First, it seems that the carry terms can always be evaluated using a big disjunction of conjunctions of simple terms. Each of the g_i and p_i terms can be evaluated quickly based on the original input bits x_i and y_i. This means the evaluation of k_i is now almost constant time instead of relying on a chain of propagation of values.

Second, there seems to be a general pattern to k_i. The p_i terms are "propagation" terms, while the g_i terms are generation terms.

In general, we can have the following equation:

$$k_{n+1} = \bigvee_{i=0}^{n} \left(g_i \left(\bigwedge_{j=i+1}^{n} p_j \right) \right) + k_0 \bigwedge_{i=0}^{n} p_i$$

In this notation, $\bigvee_{i=0}^{n}$ is a notation to represent a big disjunction, where each term is partially determined by the index variable i. Likewise, $\bigwedge_{i=0}^{n}$ is a big conjunction.

The following C code illustrates how $\bigwedge_{i=a}^{b} P(i)$ and $\bigvee_{i=a}^{b} P(i)$ are evaluated, where $P(i)$ is a predicate (a function returning a boolean value) that takes an index as its argument.

```c
int bigAnd(int (*P)(unsigned), unsigned a, unsigned b)
{
  int result = 1;
  int i;
  for (i = a;  i <= b; ++i)
  {
    result = result && (*P)(i);
  }
  return result;
}

int bigOr(int (*P)(unsigned), unsigned a, unsigned b)
{
  int result = 0;
  int i;
  for (i = a;  i <= b; ++i)
  {
    result = result || (*P)(i);
  }
  return result;
}
```

The key of this general form is the confirmation that theoretically 3 levels of gates will computer any carry term. The lowest level to compute g_i and p_i. The middle level to compute conjunctions, and the top level to compute disjunction. This ideal 3-level structure is, in reality, impossible. This is because gates have a maximum limit of inputs. Even given this limitation, the number of levels of gates needed to compute k_n is $1 + 2log_f(n)$ where f is the fan-in ratio (number of input pins) of the gates. This is far better than anything that scales linearly with n.

This mechanism of computing k_i without requiring the carry bit being propagated sequentially is called "carry look-ahead".

6.1 Exercise

As an exercise, implement a 3-bit adder again, but this time make use of the carry look-ahead method discussed in the previous section. You may find it handy to use multi-bit gates to compute the p_i and g_i terms. To make the overall design clean, you can also consider making sub-components even though these may be instantiated once in the final design.

Last, but not least, experiment with the use of a splitter/merger in Logisim. A merger/splitter allows multiple individual wires be combined to a single logical "wire". This allows the use of multi-bit gates that simplifies the overall design. A multi-bit gate is actually the same as bitwise operators in C/C++ where the same logical operation is performed for each bit position in an integer.

7 Significance of this module

This module is significant because it illustrates how to implement an arithmetic adder using logic gates. The first implementation with the carry bit propagating sequentially is slow, whereas the second implementation using the "carry lookahead" method is more efficient.