CPG 5 Coding and Programming

Introduction to Game Programming

Paul Sinnett, London South Bank University <paul.sinnett@gmail.com>

Table of Contents

Week 4

Arithmetic with gates

This week's subject areas are:

- representing negative numbers
- adding numbers with logic
- comparing numbers

Representing negative numbers

Last week we looked at representing numbers using the binary system. However, I forgot to describe how negative binary numbers are represented. Go back to the notes for last week and read again how we represent numbers in binary, then continue below to see how negative numbers can be represented.

We could have chosen to have 1 bit reserved as a sign bit. If the bit is set, the rest of the bits represent a negative number, otherwise they represent a positive number.

This is a simple system but it means we'd have to treat the addition of positive and negative numbers differently. Instead, the method used by modern computers is known as 2's complement. Here is a table of 4 bits in 2's complement representation. This same pattern continues for 8, 16, 32, and 64 bit representation:

Table 1. The table two's complement representation of signed integers

Binary	**Decimal**
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

The table two's complement representation of signed integers

What do you notice about this representation of negative numbers?

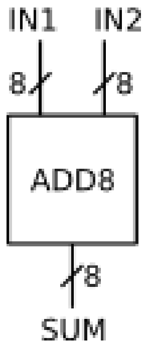
Can you see how to negate a number (make a positive number into a negative number or vice versa) with this system?

Adding numbers with logic

We saw last week that we could use the logical operations on multiple bits in parallel. But if we want to add two binary numbers together, we have to deal with carrying bits over. We can't do that in parallel because the carry of the addition of the first pair of bits is needed in calculating the second pair of bits and so on. However, at the top level, what we want is a chip that can take a pair of numbers encoded as binary and output the result:

Figure 1. An 8 bit adder

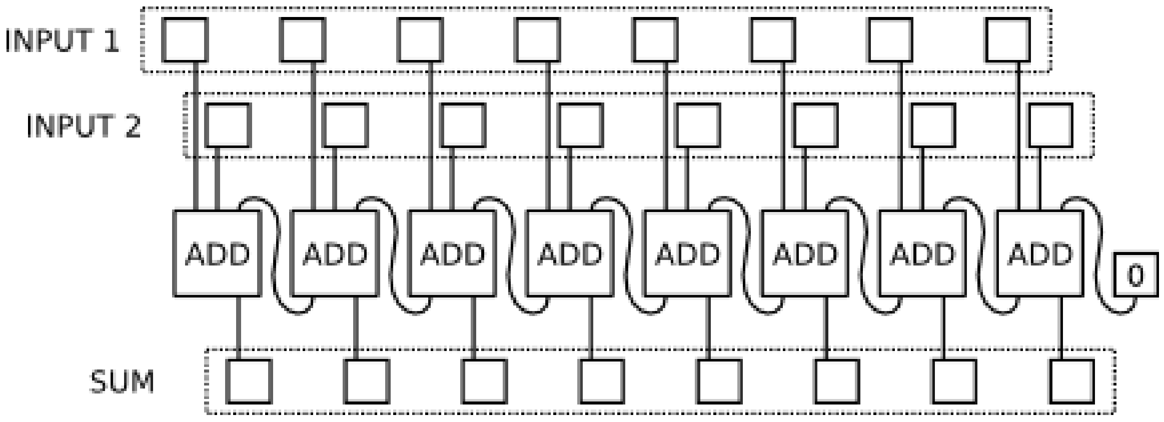
The black box representation of an 8 bit adder circuit. It takes two 8 bit encoded numbers as input, and produces an 8 bit output:



If we examine this problem we can break it up into smaller pieces. Imagine we had a simpler circuit that can take in 2 bits and a carry, and output a sum and a carry. We could join the carry output from each addition into the carry input of the circuit to the left. With this method we can construct an adder for any number of bits. We don't yet need to know the details of how this simpler circuit works. So long as what we want here is possible, the rest of the circuit will work as we expect:

Figure 2. An 8 bit ripple carry adder

This circuit is often called the ripple carry adder as you can imagine the carry bits rippling along to complete the calculation:



Note that I've interleaved the bits of the input numbers and drawn the diagram from right to left so it more closely mirrors our number system. In this diagram the logic flows from top to bottom and from right to left. The carry outputs flow from the bottom of each adder into the top of the adder to its left.

Verify for yourself that this will work, by putting some binary digits in the input boxes and work it through to make sure the right answer comes out.

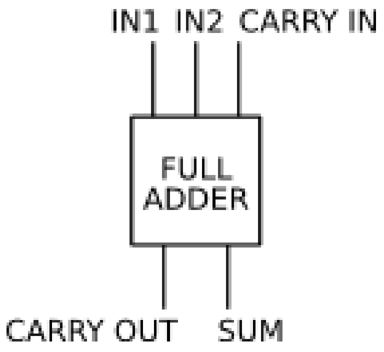
Divide and conquer

Although we haven't solved the problem yet, we have divided the problem into 8 smaller ones. And, as each part of that solution is a duplicate, we've significantly reduced the size of the problem we need to solve. This is an important skill to master when working with computers in general. You'll use this method a lot when designing your game code.

All we need to do is invent a 1 bit adder that can also accept a carry input and we can follow the pattern above to add any number of bits together. This circuit is known as a full adder:

Figure 3. The full adder

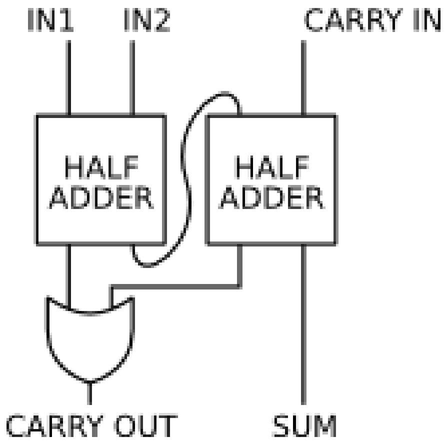
The *full adder* circuit takes 3 inputs and produces 2 outputs. The inputs are 2 bits to add and a carry from a previous adder (or a 0 if this is the first adder in the sequence.) The outputs are the sum of the inputs and a carry out if the sum overflows. Here it is as a black box diagram:



If we break down this problem again, we can see that it should be possible to solve it with 2 smaller circuits. These are called "half adder" circuits because they each do about half of the job of the full adder. Each half adder takes 2 inputs, and each outputs their sum and a carry.

Figure 4. Full adder from two half adders

With 3 inputs and 2 outputs we know from our earlier look at Boolean logic that we'd have to pick the correct 2 solutions from the 256 possible logical outputs of this circuit. But using the divide an conquer approach we can break this problem down again into something easier to solve:



Again, we haven't yet solved the problem, but we have divided it again into an easier one. Now all we need is a design for a half adder and we can build an adder circuit to add any number of binary digits.

Figure 5. A half adder

The *half adder* takes two inputs, and outputs a sum, and a carry should the sum overflow.

With just 2 inputs and 2 outputs we should be able to arrange a combination of AND, OR and NOT gates to cover all 4 possible input combinations:

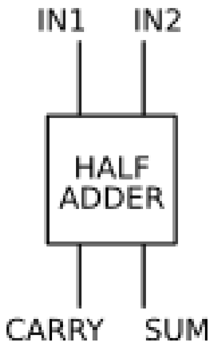


Table 2. The table of inputs and outputs for adding two bits

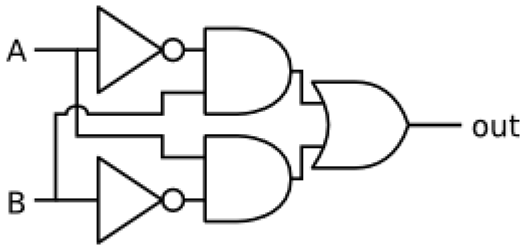
IN1	IN2	SUM	CARRY
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

The table of inputs and outputs for adding two bits

Take a moment to ignore what the columns represent and just look at the pattern of bits. We have seen both of these output bit patterns before. The CARRY column is the logical AND of inputs IN1 and IN2. And the SUM column we can create through a combination of AND, OR, and NOT commonly known as the exclusive or and sometimes written as XOR:

Figure 6. XOR

The *XOR* results in *false* when the inputs are the same and *true* when the inputs are different. As a Boolean logic diagram it can be created as follows, where *A* and *B* are the inputs and *out* is the output:



In the C family of programming languages, this expression can be written as:

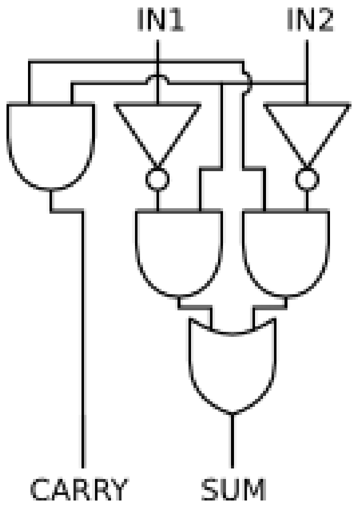
```
((A && !B) || (!A && B))
```

You can read this rule as "if (A is true and B is false) or (A is false and B is true)."

By combining the XOR and AND gates we can create the complete circuit to implement the half adder design above:

Figure 7. The complete half adder

As a Boolean logic diagram this is an AND and XOR gate in parallel:



So we have a half adder. An therefore we can make a full adder. And therefore we can make an adder for any number of bits by rippling the carry outputs into the carry inputs as described above. Q. E. D.

Video summary including a 4 bit adder built from individual transistors [https://www.youtube.com/watch?v=xISG4nGTQYE]

Video of an 8 bit adder built in Little Big Planet [https://www.youtube.com/watch?v=I-2-vdNItoI]

Comparing numbers

In many of the game rules you create for your games you will need to reference numbers. Special conditions will occur when, for example, the player runs out of lives, or bullets, or hit points.

Given that we have addition figured out, we can extend it to support subtraction simply by negating the second input.

For example, `A - B` could be written as `A + (-B)`.

Can you think of a circuit to test if a number is 0? This circuit must take an integer as input and output 1 bit either *true* or *false*.

How about a test for a negative number? Look again at the table representing negative numbers above. What do you notice about all the negative numbers that is different to all of the positive ones?

Example 1. Test if a value is equal to 0

In the C family of languages, we test if a number is equal to zero using double equals symbols. Note, don't use single equals for testing as this doesn't do what you want and has a destructive side effect.

```
if (lives == 0) { gameOver; }
```

The result of this expression is a Boolean value. That is, it is either *true* or *false*. Therefore it can be combined with other Boolean operations. For example, to test if a number is not equal to zero:

```
if (!(bullets == 0) && fireButtonDown) { fireGun; }
```

Example 2. Test if a value is negative

To test if a number is negative, use the < symbol:

```
if (shields < 0) { takeDamage; }
```

Again, the result of this is a Boolean value and can be combined with other Boolean operations:

```
if (!(damage == 0) && !(damage < 0)) { flashRed; }
```

Example 3. Comparing two values

Although we don't have any single circuit that can perform a test to see if two values are equal, or if one is greater than the other, we can emulate this ability by subtracting the numbers and testing against zero. For example, to test if we have 10 points, we could subtract 10 points from our score and test to see if the result is 0:

```
if (points + (-10) == 0) { playerWins; }
```

Equally, to test if *damage* is greater than your *shields* you could use:

```
if (shields + (-damage) < 0) { loseLife; }
```

So with just a few circuits we can add and subtract and compare number values. And all of this using no more that AND, OR, and NOT in various combinations.

Exercise

For this week's exercise I'd like you construct some game rules again. This time I'd like you to consider the rules containing numbers. You can reformulate the rules you generated in week 2 or create a new set of rules that you might use in your game program for assessment.

As before you can only include rules that resolve to either true or false, but you can now also use plus and negate in your expressions. You can also compare the result of an expression as equal to 0 or see if it is negative.

You can draw out your rules as circuit diagrams or write them out in as a C language family expression, whichever you find easiest. If you decide to draw out your rules, remember to convert them into a file format I can read before submitting. Save them as either JPEG or PDF.

You must submit your files containing your list of mathematical game rules to source control. Don't forget to keep track of your time and also submit your log files. The deadline for submission is Sunday.