Chapter 1

Binary Arithmetic

---

1. Binary Addition

We are familiar with adding decimal (base-10) numbers. But how do we add binary numbers? Consider the equation you encountered in first grade:

$$1 + 1 = 2$$

If you converted each decimal number in this equation to binary, the result would be:

$$1 + 1 = 10$$

We see that when we add two binary ones together, we have a carry into the next column (from the ones column to the twos column). This idea of "carrying over to the next column" when adding binary numbers together is analogous to carrying over when adding decimal numbers.

Think of performing the addition of the decimal numbers 8 and 4:

$$\begin{array}{r} 8 \\ \underline{+\,4} \\ ? \end{array}$$

The result (which we know is the decimal number twelve) cannot be represented by a single decimal symbol (of which the choices are 0, 1, 2, 3, 4, 5, 6, 8 and 9) contained just within the ones column. We have to carry over a quantity of ten to the tens column (where it becomes 1 since it is now in the tens column), and subtract ten from the two numbers we are adding: 8 + 4 – 10 = 2.

$$\begin{array}{r} 8 \\ \underline{+\,4} \\ 12 \end{array}$$

The same principle carries over to binary addition (or addition in any number system): If the result of the addition is a number too big to be represented as a single digit in the number system, we carry over to the next column, and subtract the quantity from the numbers being added together.

Think of performing the addition of the binary numbers 1 and 1:

$$\begin{array}{r} 1 \\ \underline{+\,1} \\ ? \end{array}$$

The result (which we know to be equivalent to the decimal number two) cannot be represented by a single binary symbol (0 or 1) contained just within the ones column. We have to carry over a quantity of two to the twos column (where it becomes 1 since it is in the twos column), and subtract two from the numbers we are adding: 1 + 1 – 2 = 0.

$$\begin{array}{r} 1 \\ \underline{+\ 1} \\ 10 \end{array}$$

Think of performing the addition of the three binary numbers 1 + 1 + 1:

$$\begin{array}{r} 1 \\ 1 \\ \underline{+\ 1} \\ ? \end{array}$$

The result (which we know to be equivalent to the decimal number three) cannot be represented by a single binary symbol (0 or 1) contained just within the ones column. We have to carry over a quantity of two to the twos column (where it becomes 1 since it is now in the twos column), and subtract two from the three numbers we are adding: 1 + 1 + 1 – 2 = 1.

$$\begin{array}{r} 1 \\ 1 \\ \underline{+\ 1} \\ 11 \end{array}$$

Addition of multi-bit binary numbers is accomplished using the results above, on a column-by-column basis. That is, for each column, use the results:

0 + 0 = 0
1 + 0 = 1
1 + 1 = 0 with a carry of 1 to the left
1 + 1 + 1 = 1 with a carry of 1 to the left

For instance, consider the addition of the two binary numbers 11 and 01:

$$\begin{array}{r} 11 \\ \underline{+\ 01} \\ ?? \end{array}$$

Starting with the right column, 1 + 1 results in 0, with a 1 carried to the left:

$$\begin{array}{r} \textit{1} \\ 11 \\ \underline{+\ 01} \\ ?0 \end{array}$$

Now, in the left column, we are adding 1 + 1 + 0, which results in 0, with a 1 carried to the left:

$$\begin{array}{r} \textit{11} \\ 11 \\ \underline{+\ 01} \\ 00 \end{array}$$

Finally, we consider the leftmost column (which consists of just the carryover from the column to its right), resulting in:

```
  11
+ 01
 100
```

Example

Perform the addition of the binary numbers:

```
     1 0 1 1 0 1              1 1 1 1
+        1 0 1 1         +      1 0 1
   -------------              --------
```

Solution:

```
     1 0 1 1 0 1              1 1 1 1
+        1 0 1 1         +      1 0 1
   -------------              --------
```

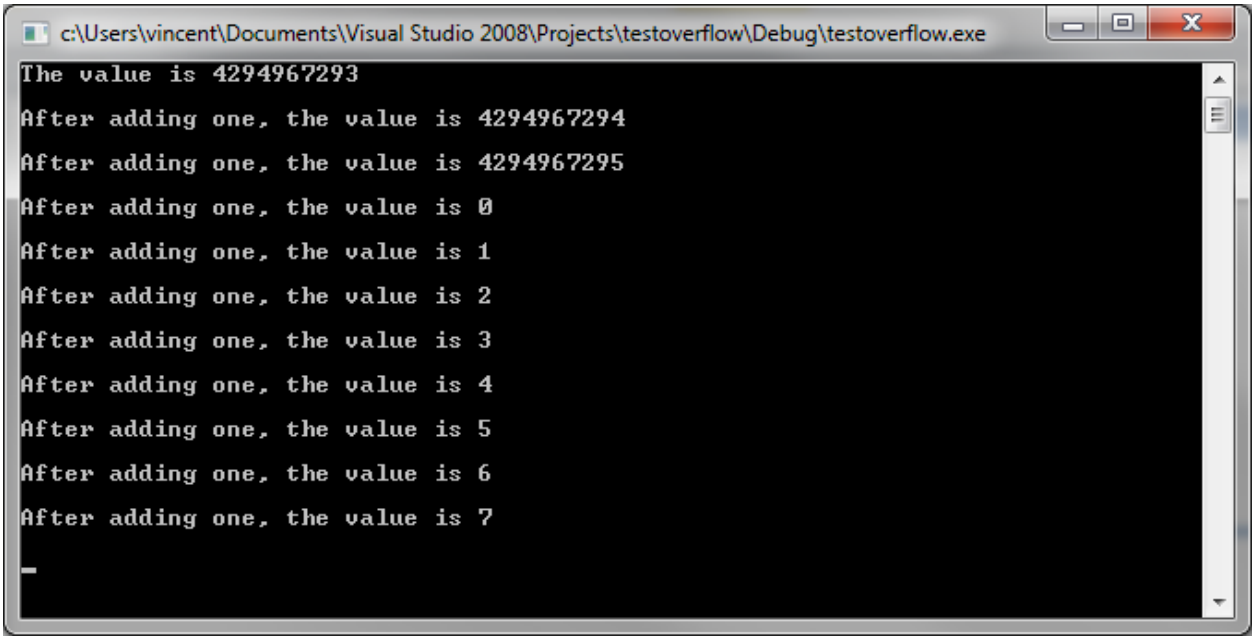
Example

Suppose we use 4 bits to store integers. Show the addition of the decimal numbers 13 + 2 in binary.

Solution:

2. Overflow

Your friend has written a computer program that initializes a natural number (i.e., a nonnegative integer) to the number 4,294,967,293 and then successively adds 1 to the number, printing the result to the screen. He believes that his program must have an error since it produces the results shown below. He asks for your help.



Putting aside the obvious question *"Why would your friend have written this program?"*, you decide to help. You notice that the program seems to start out well, adding one and printing out the correct result for the first two iterations, but then, upon reaching the value 4,294,967,295, the addition of one seems to "reset" the value back to zero. The program then seems to resume normal operation, adding one and printing out the correct result.

What's going on here?

To explore this problem, first consider a computer that uses 4 bits to store integers. The number 15 would be stored as `1111`. What would be the result of the calculation 15 + 1 in binary? Performing binary addition:

```
        1 1 1 1
+       0 0 0 1
       --------
      1 0 0 0 0
```

The result appears to be `10000` which would seem to be correct, since this is, after all, equivalent to 16 in decimal, and that is the right result for 15 + 1.

Except that this would not be the answer the computer would supply! The correct answer to the binary addition, `10000`, requires five bits. But the computer (in our example) stores all integers in four bits. All bits beyond the rightmost four bits are discarded. So, the answer to 15 + 1 in binary on our 4-bit computer would be zero (i.e., `0000`).

This scenario, where an answer requires more bits than the computer has available, is called ***overflow***.

Example

Consider a computer that uses 4 bits to store integers. Show the binary calculation of 13 + 5.

Solution:

Since only the rightmost four bits are retained, the answer would be `0010`. This is, of course, incorrect, since overflow has occurred.

Overflow in the storage of binary numbers is a real practical problem, and has been the bane of many a programmer. The most visually spectacular overflow error occurred in 1996 when the European Space Agency's *Ariane 5* rocket exploded 37 seconds into its maiden launch, sending $500 million dollars into oblivion. The programmers asked an on-board computer to store a 64-bit velocity in a 16-bit space. Only the rightmost 16-bits were retained. The rocket attempted to respond to the erroneous velocity, swerving itself into disintegration.

So, if a computer stores positive integers in a certain number of bits, what is the largest number that can be stored before overflow occurs?

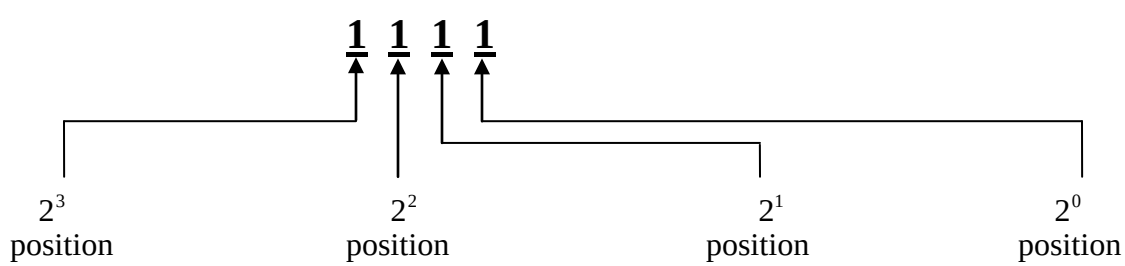
Using 3 bits the largest binary integer that can be stored is:

Using 4 bits the largest binary integer that can be stored is:

Using 5 bits the largest binary integer that can be stored is:

We would be tempted to conclude that using *n* bits the largest binary integer that can be stored is:
This is in fact correct, but the above argument is not a proof you should be comfortable with.

To determine the largest number that can be stored in *n* bits, first consider the largest number that can be stored in 4 bits:



Note that the most significant bit position is the 2^3 position. If we add 1 to this number (and ignore the fact that overflow will occur), the result will be `10000`. Converting this to decimal is easy: There is only a single 1 in the 2^4 position. So, one number larger than `1111` is equal to 2^4. So `1111` must be equal to $2^4 - 1$.

More generally, the most significant bit position in an *n*-bit number will be the 2^{n-1} position. The largest number that can be stored in *n* bits will be a string of *n* ones (where the leftmost bit is in the 2^{n-1} position). Now, if we add 1 to this string of *n* ones (and ignore the fact that overflow will occur), the result will be a

single 1 in the 2^n position, followed by zeros. So, one number larger than the largest number that can be stored in *n* bits is equal to 2^n. So, the largest number that can be stored in *n* bits is $2^n - 1$.

Example

What is the largest positive integer that can be stored on a computer if the computer stores positive integers in four bytes?

Solution:

Can you now inform your friend about the problem in his program?

3. Negative Binary Numbers

We need to represent negative integers also. How do we handle that?

You might be tempted to reply: "Easy, just use a negative sign!" But this won't work. Remember: *everything* must be represented as bits—that is, ones and zeros! A negative sign is neither a one nor a zero.

The most intuitive solution might be the *sign and magnitude* representation. With this notation:

- Let the leftmost bit represent the sign, say 0 for positive and 1 for negative.
- The remaining bits represent the magnitude.

For example, suppose we store integers in four bits with the sign and magnitude scheme. Then 3 would be stored as

```
0011
```

and -2 would be stored as

```
1010
```

As appealing as this scheme might seem at first, it is not used. To see why, consider the addition of 3 and – 2. Using addition in the typical way, we would see

```
  3        0011
 -2        1010
 ---       ----
```

But

So, addition is not straightforward if the signs of the two arguments differ. To make the sign-magnitude scheme work, think about what you would need to do:

- You would have to first determine which of the two arguments has the larger magnitude (ignoring the sign bit)
- You would then subtract the small magnitude from the larger magnitude
- You would then attach to this result the sign of the quantity that had the higher magnitude

While this could be made to work, the resulting hardware would be unnecessarily complex.

As a further annoyance with this scheme, it has two representations for zero: . (It took mathematicians over a thousand years before they could handle the concept of zero…Surely we should pause before casually introducing a system with two zeros!)

The Two's Complement Notation

All computers represent integers in a manner that allows both positive and negative numbers. To represent positive numbers, simply use its binary representation, as we have done before. Negative binary numbers are represented in what is called "*two's complement notation*". To find the two's complement of a binary number, change each 1 to a zero and each zero to a one (i.e., invert all the bits), then add one to this quantity.

Example

What is the two's complement of the binary number `10010010` ?

Solution:

So…how do we find the representation of a negative decimal number? Do the following:

- First find the binary representation of the number without the negative sign
- Then take the two's complement. The result is the representation of the negative number.

Example

Express the decimal integer –13 as an eight bit binary number.

Solution:

Example

Express the decimal integer -53 as an eight bit binary number.

Solution:

Example

Suppose we store integers in 4 bits. Represent -1 in two's complement notation.

Solution:

There is a neat shortcut to find the two's complement of a binary number. Starting at the right, and moving to the left, leave all bits unchanged until the first 1 is encountered. Leaving this very first 1 unchanged, continue moving to the left inverting all bits from that point onward. For instance, we noted above that the decimal number +53 in binary is

```
0 0 1 1 0 1 0 1
```

To find the two's complement, we start at the right and move to the left, stopping at the first 1 that we encounter, which in this case happens to be the rightmost bit. We leave this bit unchanged, but invert all other bits to the left, resulting in

```
1 1 0 0 1 0 1 1
```

Example

Use the shortcut technique to determine the two's complement representation of -92 using 8 bits

Solution:

You are likely wondering: Why is the two's complement notation a good representation for negative numbers? Why do we flip all the bits and add one instead of, say, flipping half the bits and adding 5?

Here's why: A binary number, added to its two's complement, results in zero. This is "how its '*sposed* to be: a number added to its negative should give zero.

For example, let's presume integers are represented using 8 bits. Back in Section 1.3 you determined that the binary representation of 53 was `00110101`. Two examples back you determined that –53 was `11001011`. Let's add 53 to –53:

```
 53:   0 0 1 1 0 1 0 1
-53:   1 1 0 0 1 0 1 1
       ---------------
     1 0 0 0 0 0 0 0 0
```

If we are only using 8 bits, then only the eight least significant bits are retained, and the result is zero (as it should be).

If we represent negative numbers using two's complement notation, then binary addition involving negative numbers will give the correct result. Every computer today uses two's complement notation for the storage of integers.

Example

Suppose a computer stores integers in 4 bits using two's complement notation. Show the calculation – 5 + 7.

Solution:

4. Two's Complement: A Closer Look

Suppose we have a computer that stores integers in four bits. If we only have non-negative integers, then we can represent the integers 0 – 15 using four bits:

Decimal Number	Binary Number	Decimal Number	Binary Number
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Note that we are limited to 16 possible four-bit strings. These 16 unique strings are shown in the right column of the table above. Since we have 16 unique 4-bit strings possible, we can represent 16 unique decimal numbers. These 16 decimal numbers (0 – 15) are shown in the left column of the table above. The assignment of decimal number to bit string is performed by the usual decimal to binary conversion.

Now, suppose we still want to restrict ourselves to storing integers in four bits, but we want to include both positive and negative integers (using two's complement notation, since we know that works!). What integers can we now store, if only four bits are available?

It should be clear that we cannot store the integers –15 to +15 within our 4-bit space. The range –15 to +15 comprises 31 unique integers, but since we have only 16 possible bit strings available, we can only store 16 unique integers within our four bit allotment.

You should agree that we should store at least the integers –7 to +7. But this range will comprise 15 integers, and we have the ability to store sixteen—so there is room for one more! Should it be +8? Or should it be –8? It can't be both. Let's begin by filling in the integers 0-7:

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

Let's fill in the range of integers from –1 to –7 one integer at a time. Starting with –1, we determine that this integer is represented by `1111` in 2's complement notation, and we have:

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

Moving on to –2, we determine that this integer is represented by `1110` in 2's complement notation:

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

Adding the two's complement representations for –3, –4, –5, –6 and –7, we have:

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

We have one bit string left: `1000`. Should this be +8 or –8?

It would be easy to have this string, `1000`, represent +8. After all, in binary, +8 is equal to `1000`.

But let's consider –8. In two's complement notation this integer is represented by… `1000` ! So, we really do have a choice: we can let our last bit string `1000` be either +8 or –8. Both would work.

But one choice is much better than the other! Look carefully at the table above: For all the positive numbers, the most significant bit in the four-bit string is a zero. For all the negative numbers, the most significant bit in the four-bit string is a one. If we assign the remaining bit string (`1000`) to the integer –8, it will continue to be the case that the most significant bit in the four-bit string is a one for all negative numbers and a zero for all positive numbers. This is very handy—we can immediately tell if a number represents a positive or negative integer just by examining the most significant bit! This is the way to go!

The final assignment is:

Decimal Number	Binary Number
0	0000
1	0001

2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

To summarize: If we use four bits to store integers using two's complement notation, the range of integers that can be stored is – 8 through +7, and, given the four-bit binary number, we can immediately tell if it is positive or negative by looking at the leftmost bit. If the leftmost bit is a 0, the number is positive. If the leftmost bit is a 1, the number is negative.

How do we "reverse" the two's complement process; i.e., given that a 4-bit two's complement number is stored as, say, `1011`, how can I easily determine that the bit string is storing the decimal value `-5`?

To determine the decimal equivalent of a number stored in two's complement notation, perform the following:

- If the leftmost bit is a 0, the number is positive. Simply convert the bit string to a decimal number.
- If the leftmost bit is a 1, the number is negative. To determine the decimal number:
  - Take the two's complement of the bit string (i.e., flip all the bits and add one)
  - Convert the bit string to a decimal number
  - Append a negative sign to the result.

Example

Suppose we store integers in 4 bits using two's complement notation. What is the decimal representation of the number stored as `1111` ?

Solution:

Example

Suppose we store integers in 4 bits using two's complement notation. What is the decimal representation of the number stored as `0101` ?

Solution:

The foregoing results presumed that we were storing integers in two's complement notation using 4 bits, but the results can be easily generalized to the case where we store integers in two's complement notation using *n* bits.

Example

What is the largest positive integer that can be stored in *n* bits using two's complement notation?

Solution:

Example

What is the smallest negative integer that can be stored in *n* bits using two's complement notation?

Solution:

We summarize important results. If we store integers in two's complement notation using *n* bits:

- The left most bit functions as a sign bit. If the leftmost bit is a 0, the number is positive; if the leftmost bit is a 1, the number is negative.
- The range of values that can be stored is
- Regardless of the number of bits used, the number zero is always represented as
- Regardless of the number of bits used, the number -1 is always represented as
- Regardless of the number of bits used, the most positive number is:
- Regardless of the number of bits used, the most negative number is:

Example

What is the decimal equivalent of this 32-bit two's complement number?

1111 1111 1111 1111 1111 1111 1111 1100

Solution:

Example

Show the calculation – 5 + 3 in 4-bit two's complement notation. Convert the answer back to decimal to confirm your answer is correct.

Solution: _____

Example

Show the calculation 6 + 3 in 4-bit two's complement notation. Convert the answer back to decimal to confirm your answer is correct.

Solution: _____

Example

Show the calculation – 6 + (– 3) in 4-bit two's complement notation. Convert the answer back to decimal to confirm your answer is correct.

Solution:

Overflow is recognized when we add 2 numbers of the same sign and get a result of the opposite sign.

5. Binary Subtraction

Subtraction in binary entails nothing new! To compute $a - b$, we compute $a + (-b)$ where $-b$ is represented using two's complement notation.

Example

Show the subtraction $3-2$ in a two's complement computer that stores integers in 4 bits.

Solution:

Example

Suppose we use six bits to represent a two's complement binary number. Do the following subtraction.

```
        011101
  –     100101
```

Show the subtraction above with all numbers converted to decimal, and explain your results.

Solution:

We have overflow if the addition of two positive numbers yields a negative number, or if the addition of two negative numbers yields a positive number. It turns out that there is a quick check that can be done to detect if overflow has occurred: If the carry into the most significant bit (which serves as a sign bit) differs from the carry out from the most significant bit, overflow has occurred.

Problems

1. Perform the addition of the following unsigned (i.e., not two's complemented) binary numbers:

```
(a)   1111        (b)   1011        (c)   1110011
    + 1010            + 0011            + 0011110
```

2. Suppose that unsigned numbers are stored in five bits. Perform the following additions, indicating which additions produce overflow.

```
(a)   111100        (b)   101100        (c)   111001
    + 001011            + 001100            + 011001
```

3. Suppose we use 6 bits to store integers. Show the following additions in binary:

(a) 10 + 5
(b) 32 + 31
(c) 1 + 31
(d) 21 + 21

4. Express the following numbers as 8-bit binary numbers in two's complement notation:

(a) 60
(b) -60
(c) 1
(d) -1
(e) 0
(f) 12
(g) -15
(h) -128

5. Convert the following six-bit two's complement numbers to decimal:

(a) 010111
(b) 110111
(c) 011111
(d) 100000

6. Show each of the following calculations in 4-bit two's complement notation. In each case, convert the result (in binary) back to a decimal number.

(a) – 5 + 7
(b) – 3 + 3
(c) 6 – 4
(d) –2 – 3

7. Suppose we use six bits to represent a two's complement binary number.

(a) What is the largest number that can be represented?
(b) What is the smallest number that can be represented?
(c) How many total base-10 numbers can be represented?

8. Suppose we use six bits to represent a two's complement binary number. Perform the following additions, indicating when overflow occurs:

(a)
```
    010101
+   001101
```

(b)
```
    100000
+   110011
```

(c)
```
    010101
+   101111
```

(d)
```
    100110
+   111111
```

9. Suppose we use six bits to represent a two's complement binary number. Perform the following subtractions, indicating when overflow occurs:

(a)
```
    010101
-   001101
```

(b)
```
    100000
-   110011
```

(c)
```
    010101
-   101111
```

(d)
```
    100110
-   111111
```