Datatypes – Chapter 1, For CS 2110

Shreyas Casturi

Contents

1 Chapter 1: Datatypes

If we aim to have any sort of discussion about the architectures that underpin computers, we must first understand *binary* numbers and their importance. Indeed, a base 10 numerical system does not lend itself well to computing.

When I say, "what is five?", you may draw the number five, the word "five", five lines, a mathematical expression that evaluates to five, so on and so forth. What we notice is that these are simply *ways or schemes* of denoting something.

So, our base-10 numeric system is another *scheme* of denoting quantities that have no real name. After all, numbers exist independently of us.

Might there be an easier way of expressing what we already know?

1.1 Binary Numbers

In the early days of computing, much detail was devoted to the electronics and hardware. Scientists realized that circuits obeyed a simple rule – either they were on, or they were off. There was no inbetween.

This made it easier to accept the *base 2 numeric system*. Numbers expressed in this system are known as *binary* numbers. A single element of a binary number is known as a *binary digit*, or a *bit*. A binary digit has one of two values: 0, or 1. A binary number is composed of *only* binary digits.

So, a random binary number might look like this

$$number = 01000101010100010011100001010 1$$

Given n bits, how many possible binary numbers can I generate?

Remark. *Given n binary digits, there are 2^n possible binary numbers.*

Proof. There are 2 numbers possible for each of the n digits. Multiply this out to obtain the result. Start with 1 or 2 digits, and work up from there. □

Of course, these numbers are worthless on their own. If someone told you that there were "0101010101" cats stuck in that burning house on the left, you'd probably look at them funny. You might even inhale some smoke and leave. Those poor cats are dead.

1.2 Assignations in Binary

For four binary digits, 16 combinations are generated. We may naturally try to assign our old numbers (which we'll call integers) to these 16 combinations. But how many orderings are possible?

There are $n!$ orderings possible, and we'd like to create an ordering that is easy enough for all of us to follow. Indeed, we could just do something like this:

Integers	Binary numbers
0	0001
1	1010
2	1101
⋮	⋮

This really isn't helpful, is it.

Well, what happens if we assign 0000 to 0, naturally, and 1111 to be the largest possible integer that could be generated? For 16 possible combinations, obviously the range is from $[0 \to 15]$, or to put it a little more mathematically, $[0 \to 2^n - 1]$.

The most interesting part is this: Let us have only two binary digits. There are then four possible binary numbers, and four integers we could assign. If 00 is 0, then 11 is 3. Let 01 be 1, and let 10 be 2.

Then we get this chart:

Integers	Binary numbers
0	00
1	01
2	10
3	11

Observe the progression of the binary numbers. Each time we add 1 to our integers, digits are being flipped, but exactly what digits?

Let us do this one last time for 3 binary digits. How many binary numbers can we generate? And what is the range of the integers for these binary numbers?

The chart follows.

Real numbers	Binary numbers
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

A relationship is obvious from the numbers generated, but may not be intuitive. Here is what we know, so far, about binary numbers:

1. A binary number is formed from binary digits.
2. Each binary digit can be 0 or 1.
3. There are 2^n binary numbers generated from n digits.
4. We know that we could assign 2^n of our integers to the 2^n binary numbers we generate from n binary digits.
5. We know there are certain intuitive ways to begin this ordering.
6. Let the n binary digits be all 0 to denote 0, and let the n digits be all 1 to denote the largest possible integer.
7. But... what happens after that?

The more general question we might ask ourselves is *how do we actually determine what numbers to assign to what combinations*?

One way to do this is to view *each* binary digit as a power of 2. Let us do this like so, for four binary digits:

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

Then, we can intuitively break down any binary number given to us by writing the powers of 2 above these numbers, starting with 2^0 for the right-most digit, and ending with 2^{n-1} for the n digits given to us.

Notice then that all of these numbers can be written as **sums of various powers of 2**.

Let us ask ourselves some questions:

1. Given n binary digits, how many possible binary numbers can be made?

2. Given n binary digits, what is the largest power of 2 possible, and what digit is it found at?

3. Given n binary digits, how many possible integers can be assigned, and what is the range of these integers, assuming only 0 and the positive integers are used?

Remark. *Some Facts On Binary Digits:*

1. *For n binary digits, 2^n binary numbers can be generated.*

2. *Following from the chart above, the largest power of 2 possible is 2^{n-1}. It is found at the left-most digit.*

3. *Given that we map integers to binary numbers in a one-to-one correspondence, then 2^n integers can be assigned. Assuming only 0 and the positive integers are used, then the range of integers is $[0 \to (2^n - 1)]$.*

Before we do anything else with binary numbers, we should first make sure we are comfortable with basic operations involving those binary numbers.

1.3 Addition and Subtraction In Binary

We should get comfortable thinking in binary terms, so this means we should have some experience operating with binary numbers – namely in addition and subtraction. We will use the binary-integer relationships we have described above to help us figure out what our binary result should be.

We have already discussed that there are only ever 2 values in each bit of a binary number – 0 or 1, and that the place values are simply powers of 2, starting from 2^0, all the way to 2^{n-1}.

With this in mind, we shall pose some problems.

1.3.1 Basic Addition

Problem 1: Let $A = 0010\ 1110$, and let $B = 0110\ 1101$. What is $A + B$?

Answer:
We can write the problem as

$$\begin{array}{r} 0010\ 1110 \\ +\ 0110\ 1101 \\ \hline xxxx\ xxxx \end{array}$$

1. (Both Zero Bits): $0 + 0 = 0$

2. (Bits Not Equal): $0 + 1 = 1 + 0 = 1$.

3. (Both Bits == 1): $1 + 1 = 0$(carry over 1)

The rule for addition is simple – if both bits are 0, the answer bit is 0. If both of the bits are not the same, then it's a 1, with no carry. If both of the bits are 1, then the result is a 0, and you carry a 1 over. Basically, 1 + 1 = 0 (with a carry of 1 to the next digit).

So, our addition problem then becomes

$$\begin{array}{r} {\scriptstyle 1\,1\ \ 1\,1}\\ 0010\,1110\\ +\,0110\,1101\\ \hline 1001\,1011 \end{array}$$

What is this answer in base 10 format?

I won't write out the whole tables, but you get

$$1001\ 1011_2 = 2^7 + 2^4 + 2^3 + 2^1 + 2^0 = 128 + 16 + 8 + 2 + 1 = 155.$$

If you turn A and B into their base-10 expressions and add those two integers together, you should get the same result. If you get stuck, try turning your adding terms into their base-10 forms and adding from there. It is a handy reference.

1.3.2 Basic Subtraction

This will be filled in later.

Now that we know how to work with binary numbers and perform operations on them, we may ask ourselves a simple question – we've only covered the positive integers. *What about negative numbers?*

1.4 Beyond The Positive

We usually deal with both positive and negative integers, but so far we've only dealt with positive integers and zero, strictly. We must find a way to incorporate negative integers.

Before we do that, we should define some terms.

Assuming there is a number (binary or otherwise) labeled x, the **additive inverse** of x, labeled as x^{-1} is another number such that $x + x^{-1} = 0$. This means that the additive inverse of 4 is -4, and in binary, this means $0100 + 1100 = 0000$; $4 + -4 = 0$.

Now, we can move onto actual methods of representation.

1.4.1 Signed Magnitude

One logical idea is that given a binary number of n bits, reserve one bit – the leftmost bit, or the most significant bit – to represent the sign of the number. This is called the *signed magnitude*.

So 0000 is 0, 1000 is -0, 0001 is 1, and 1001 is -1, so on and so forth.

We will make use of a chart from the previous sections and continue to update as such, with each scheme we get/invent.

We shall use this chart to show four-bit numbers and our various numbering schemes.

The first two are the unsigned integers and signed magnitude.

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

1.4.2 Problems with Signed Magnitude

While the signed magnitude is intuitive, there are some problems – namely, why are we going to have two zeroes (0000 and 1000)?

Is there a way to improve upon this signed magnitude scheme?

There in fact is, but it isn't immediately intuitive. If we start with a number line as so

INSERT THE FUCKING NUMBER LINE

We'll put 0 and 0000 at the exact middle, and go up by 1 **until** our left-most significant bit becomes 1. At this first instance (1000, to be exact), we're going to assign this number $-(2^3)$, or -8. Visually, we'll **wrap around** the number line and start from the back, going back towards 0000. Our final number, 1111, will be -1.

This number scheme is known as **two's complement**, and it is the major scheme that we'll be using throughout this course and, for that matter, all computers.

Why is this scheme so widely used? Notice how the difference between positive and negative numbers is nothing more than a **bitwise inverse** (flip the bits and add 1). Also note that because we can find the additive inverse of a number easily, there's no reason to actually perform (or remember) *how* to perform subtraction, when all you're doing is still doing the act of addition with an additive inverse.

This allows us to gain another column in our chart:

Binary Numbers	Unsigned Integers	Signed Ints	2's Complement
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-0	-8
1001	9	-1	-7
1010	10	-2	-6
1011	11	-3	-5
1100	12	-4	-4
1101	13	-5	-3
1110	14	-6	-2
1111	15	-7	-1

We immediately gain some questions:

1. Given an n-bit binary number, what is the largest positive value we can have, and what is the largest negative value we can have, assuming we're working in a 2's complement system?

2. What is always the smallest negative number (-1) represented as in 2's complement, for any given n-bit number?

The above is good food for thought. But now that we know about 2's complement, we must ask ourselves how to find the *additive inverses* of positive numbers, as we'll need to be able to do subtraction (addition with additive inverses) easily.

UNLESS OTHERWISE STATED, WE WILL BE USING 2'S COMPLEMENT FROM NOW ON. BE CAREFUL.

1.4.3 Additive Inverses in 2's Complement

The easiest, most surefire way to do find 2's complement additive inverses (negative numbers) – that is, a way that is foolproof and can never fail, is as follows:

Remark. *Naive Inversion: Given a number x in 2's complement, in order to find the additive inverse of this number, given as x^{-1}, simply flip the bits of x and add 1 to the least significant bit.*

Of course, for long numbers, this may be a bit of a pain to do. There is an easier way, though it is sometimes harder to remember – I myself have to always do a bit of work to remember/rederive the rule.

Remark. *Bossut's Rule: Given a number x in 2's complement, in order to find the additive inverse of this number, start from the least-significant bit and go left. Find the* ***first*** *bit that is 1. Then,* ***flip*** *all the bits* ***to the left*** *of this first 1 bit. The result will be the additive inverse of x, given as x^{-1}.*

Either way works.

1.5 Overflow

In computing, we have often seen errors involving "not-a-number", or integer overflow, or some other weird problem – for example, if adding `ints` in Java, you may find that very very large ints being added together will cause... unexpected results. This is due to a concept known as *overflow*, where a computer (or particular datatype) simply doesn't have enough bits to represent all pertinent information.

When dealing with binary numbers, we must obviously consider this problem of overflow.

What generates overflow?

In order to handle overflow, we must first check the carry-in and carry-out of the leading bit, or rather, the most-significant bit.

When we add or subtract (add with additive inverse) two binary numbers, we have overflow if either of these two conditions are satisified:

1. We get a carry into the signed bit/most-significant bit but no carry out.
2. We get a carry out of the signed bit/most-significant bit but no carry in.

Numbers that either have no carry-in and carry-out *or* carry-in and carry-out with regards to their most-significant bits are not considered *overflowing*.

Let us look at some examples of overflow and no overflow.

1.5.1 Overflow in Addition And Subtraction

Problem 1: Let us take $A = -5, B = -3$. Does the result of $A + B$ overflow?

We may write our addition out as such

$$\begin{array}{r} 1110 \\ +\,1101 \\ \hline xxxx \end{array}$$

We can use our rules of binary addition to observe that

$$\begin{array}{r} {\scriptstyle 1} \\ 1110 \\ +\,1101 \\ \hline 11011 \end{array}$$

There is a carry-into the most significant bit, and the extra 1 at the bottom is the carry-out. Hence, this sum **doesn't** overflow.

Problem 2: Let $A = 4, B = 5$. Does $A + B$ overflow?
We write our addition as such:

$$\begin{array}{r} 0100 \\ +\,0101 \\ \hline xxxx \end{array}$$

Then, our answer is

$$\begin{array}{r} {\scriptstyle 1} \\ 0100 \\ +\,0101 \\ \hline 1001 \end{array}$$

Note that the most significant bit has a 1 carried-into it, but nothing carried out of it, because $1 + 0 + 0 = 1$. Hence, this sum will overflow.

1.6 Sign Extension

Let us assume we have a four-bit binary number and eight-bit binary number that we have to add together, but we have an adder (a special device that operates on bits) that can only add two eight-bit binary numbers.

We need a way to turn the four-bit binary number into an eight-bit binary number *without* changing the number itself.

One way we can do this is to *extend* the amount of bits that a number has. The way to do this is to add some amount of bits to the left of the binary number, based on the *signed* bit of the number. So, if a number is negative, the signed bit is 1, and we'll add some amount of 1's to the left of the number.

To answer our original question, we'd take the sign of the four-bit binary number and add another four bits based on the sign, so we'd then have two eight-bit binary numbers that we could add with our adder.

1.6.1 An Example of Addition with Sign Extension

Problem: Let $A = 15, B = -63$. Find $C = A + B$.
A can be written as

$$A = 01111$$

and B can of course be written as

$$B = 1000001$$

So the addition will be

$$\begin{array}{r} 01111 \\ +\,1000001 \\ \hline \end{array}$$

We can sign-extend A to have the same amount of bits as B by adding zeroes to the left of the most-significant bit. If A were negative, we would add ones to the left of the most significant bit.

$$\begin{array}{r} 0001111 \\ +\,1000001 \\ \hline \end{array}$$

The actual addition is

$$\begin{array}{r} 0001111 \\ +\,1000001 \\ \hline 1010000 \end{array}$$

This is of course -48 – I didn't show the carries because that's kind of a pain to do.

1.7 Fractional Binary Numbers

We may sometimes want to get a decimal number – an integer with some decimal component added to it, like 10.25.

What we can do is have a regular binary number with a decimal point affixed to the right of the *least significant bit*. Here is the layout of such a number, with positive powers of 2 to the left of the decimal point, and powers of 2 less than 1 to the right of the decimal point, like so

$$[2^3 \mid 2^2 \mid 2^1 \mid 2^0 \mid . \mid 2^{-1} \mid 2^{-2} \mid 2^{-3} \mid \dots]$$