14/10/2019 Daniel Hannon (19484286)

Assignment 2 – Numbering systems (Part 2)

Once you are able to perform the basics of adding and subtracting positive intergers in various numbering systems, the logical progression is that of the more complex number, such as: Negative Numbers and fractions within the next few pages I intend to outline various methods of representing negative numbers and fractions.

Negative Numbers in Binary

In order to calculate negative numbers in binary there is two methods in which I will discuss and mention the pros and cons of either system.

Sign Magnitude

The first one being Sign Magnitude, the way this represents negative numbers is by using a signed bit, alone, this is where the rightmost number represents the sign on the number, converting this number to binary is relatively easy

lets use the example of the number -124
Using the method previously explained, we can deduce that 124 in binary is 01111100
Now add a signed bit 11111100 and voilá! It's a negative number.

There are a few downfalls of this method of negative number representation, the first one is that there are two representations of zero, those being 10000000 and 00000000, where the second one is that arithmetic is computationally hard let's use the example of (-3) + 4

```
  00000100
- 10000011
  --------
  00000001
```

This calculation is rather difficult for a computer so it would require more complex hardware in order to perform this at speed.

Complimentary Representation

The next system I will introduce is complimentary representation, within binary, there are two ways in which this can be implimented, 1's compliment or the more common (and superior) 2's compliment. These have base 10 equivalents 9's compliment and 10's compliment respectively.

1's Compliment
Lets start by explaining 1's compliment

Imagine a negative number within the range of -127 and 0

let's say -45

first lets convert 45 to binary

32 + 8 + 4 + 1

00101101

Okay now to represent this we subtract this from the absolute value (in this case 11111111)

	1	1	1	1	1	1	1	1
-	0	0	1	0	1	1	0	1
	1	1	0	1	0	0	1	0

so Using 1's compliment -45 is 11010010!
in order to return it to 45 all you have to do is subtract the compliment value from the absolute value.

There are some pitfalls with 1's compliment. The first being that there is two representations of zero. And the second being that arithmetic is still somewhat awkward, albeit easier to compute, and the last one being the limited range of 255 different numbers being representable. Rather than the standard 256 for 8 bits in binary. To tackle this issue I am going to discuss 2's compliment.

2's Compliment

The main and Inital Difference between 1's compliment and 2's compliment is that 2's compliment involves subtraction from a value worth one more than the maximum possible value rather than the maximum possible value itself, this allows the numbers to range from -128 to 127, which accounts for the maximum of 256 possible numbers being representable by an eight bit binary value.
This also simplifies addition as when it reaches the maximum possible value the number will overflow and rollover to 00000000 rather than having to convert at 11111111 to 00000000, thus being faster and easier to calculate. The way it achives this is that the value -0 is worth one more than the maximum representable number (in the case below it's 100000000_2 in an 8 bit binary number)

Let us consider the value of -89 and get it's binary equivalent in 2's compliment

1	0	0	0	0	0	0	0	0
-	0	1	0	1	1	0	0	1
	1	0	1	0	0	1	1	0

Decimal Equivalents

9’s Compliment

9’s Compliment works in a fashion like that of 1’s compliment, that being that -0 is represented by the maximum value representable within the allowed range of digits (in the case below 99999) which implies a range of -49999 to 49998. let us demonstrate using a random value of -32425

	9	9	9	9	9
-	3	2	4	2	5
	6	7	5	7	4

As we have established that -32425’s 9’s compliment equivalent is 67574, we can get the original manitude of the number by subtracting it from 99999 which would return 32425.

10’s Compliment

10’s equivalent functions exactly like that of the binary equivalent. That being that -0 is represented by the maximum value +1 (in the case below 100000) this allows for easier calculation as the overflow accomodates an immediate rollover from -1 to +0, this also allows us to represent the range -49999 to 49999, thus being more efficient numerically, let us consider -27265 in 10’s compliment

1	0	0	0	0	0
-	2	7	2	6	5
	7	2	7	3	5

Just like before, in order to get the negative value of the number just subtract the 10’s compliment equivalent from (maximum value + 1) again.

Overflow Versus Carry

Overflow and Carry work very similarly but they are quite different, Carry is when there is a place to put the bit so the value is kept where Overflow is when the new value exceeds the highest possible number and it rolls over as there is nowhere to keep the bit and it is ultimately discarded.

Worked Examples

Overflow
(Data Lost)

	1111	1111
+	0000	0001
	0000	0000

Carry
(Data Carried)

	0111	1111
+	0000	0001
	1000	0000

Exponential Notation

For extremely large or small numbers it does not make sense to write out every single digit as it would occupy arbitrarily large amounts of space and if this consists of large amounts of trailing or preceeding zeroes this does not make sense to write out in its full decimal format. In order to tackle this issue we introduce the concept of scientific notation.

Six key items are required to define and implement such notation
1. The sign of the number
2. The Magnitude of the number
3. The Sign of the Exponent
4. The base of the exponent
5. The magnitude of the exponent
6. the location of the floating point.

To Demonstrate let's consider a number ($244,236,200,000,000,000,000,000_{10}$) which we can represent as 2.4424×10^{23} to conserve space and have little loss in precision of the number and it can easily be converted to a larger number by multiplying it out.

Excess n notation

Just like we saw with 10's, 9's, 1's, and 2's compliments. Within some numbering conventions it is fine to use positive numbers to represent negative numbers and in order to properly introduce floating point numbers, I must introduce the system that is Excess-127 notation.

Within this system $\mathbf{0111\ 1111_2}$ is the same as $\mathbf{0_{10}}$ and this system covers the number range from $\mathbf{-127_{10}}$ to $\mathbf{128_{10}}$ which are represented as $\mathbf{0000\ 0000_2}$ and $\mathbf{1111\ 1111_2}$ respectively. The following table may be of some use.

Base-10	Excess-127	Binary
128	255	1111 1111
64	191	1011 1111
32	159	1001 1111
0	127	0111 1111
-32	95	0101 1111
-64	63	0011 1111
-127	0	0000 0000

Floats

Hopefully you have noticed that you cannot represent decimal numbers very accurately with this system, this is where floats come in.

Floats are typically 32 bit numbers with a very large range of possible values

Within those 32 Bits they follow the following format

S EEEE EEEE MMMMMMMMMMMMMMMMMMMMMMM

S: Signed Bit (0 is positive, 1 is negative)
E: Exponent (Typically uses Excess-127 Notation)
M: Mantissa

Which is then converted into the form

$\mathbf{(-1^S) * 1.M_2 * 2^E}$ To calculate it in it's Base-10 Format.

Converting to a Float

Consider an arbitary number of value 24.8432646

first we split it in to two, the interger value, and the decimal value

24 0.8432646

Using previous knowledge we know that 24 is 11000 in binary

now to get 0.8432646 to about 10 bits of accuracy

0.8432646	0.
1.6865292	1
1.3730584	1
0.7461168	0
1.4922336	1
0.9844672	0
1.9689344	1
1.9378688	1
1.8757376	1
1.7514752	1
1.5029504	1

we have established that $.8432646_{10}$ is 0.1101011111_2 accurate to ten places

Now we combine them to get 11000.1101011111_2
now we divide by 2 until there is only one 2 left on the left hand side

$1.10001101011111_2 * 2^4 = 11000.1101011111_2$

Now, recording the value of **n** within 2^n we get 4, as floats use 127 to equate to 2^0 we know that 131 is 2^4 and that 131_{10} can be converted to binary quickly 10000011_2 as it is a positive number we know that the signed bit is 0, therefore 24.8432646 as a float is

0 10000011 10001101011111000000000

Float Addition

Adding two floats is simple but it requires several steps. Let us represent the addition of

2.5 + 5.125 = 7.625 in it's floating point equivalent

Using previous knowledge we can establish the float equivalents of both numbers

0 10000000 01000000000000000000000
0 10000001 01001000000000000000000

Step 1: Get the Mantissas of both

$2.5 = 1.01_2 \text{ x} 2^1$
$5.125 = 1.01001_2 \text{ x} 2^2$

Step 2: Adjust the mantissas to be of the same magnitude

$2.5 = 0.101_2 \text{ x} 2^2$
$5.125 = 1.01001_2 \text{ x} 2^2$

Step 3: Perform Addition Operation

	0	.	1	0	1	0	0
+	1	.	0	1	0	0	1
	1	.	1	1	1	0	1

New Mantissa value is $1.11101 \text{ x} 2^2$

Step 4: Adjust the mantissa to fit the format 1.x, if necessecary

Step 5: reassemble the mantissa

0 10000001 11101000000000000000000
Now that was easy, wasn't it?

Multiplication of Floating Point numbers

Let us implement the following multiplication as floats

$6.125_{10} * 3.625_{10} = 22.203125_{10}$

First we need both numbers as floats

0 10000001 10001000000000000000000
0 10000000 11010000000000000000000

Step 1: Multiply the Mantissae

1.10001
1.1101

	1	.	1	0	0	0	1				
X	1	.	1	1	0	1	0				
	1	.	1	0	0	0	1				
	0	.	1	1	0	0	0	1			
	0	.	0	1	1	0	0	0	1		
	0	.	0	0	0	0	0	0	0		
+	0	.	0	0	0	1	1	0	0	0	1
1	0	.	1	1	0	0	0	1	1	0	1

adjust this result to fit the standard mantissa format

1.0110001101

Step 2: Adjust the exponent

Add the Exponents

	1	0	0	0	0	0	0	1
+	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1

Then subtract this value from 127_{10} ($0111\ 1111_2$)

1	0	0	0	0	0	0	0	1
-	0	1	1	1	1	1	1	1
	1	0	0	0	0	0	1	1

The new exponent is $1000\ 0100_2$

Step 3: Assemble the new float
0 10000100 01100011010000000000 And That's it!

References

https://www.h-schmidt.net/FloatConverter/IEEE754.html (For verification of floating point values and such)