COMP1521

W7 - Signed and Float Representations

Overview

Admin

Endianness

Signed numbers (Q1, Q2, Q3)

Floating point numbers (Q4, Q5)

Bonus (Q6)

Admin

W5 and W6 weekly tests are due today.

W7 weekly test is released today, due next thursday.

W7 lab is due next monday.

Assignment 2 will be released shortly. It will mainly assess files and file OPs, with a little bit of bitwise.

Endianness

What is endianness?

Endianness describes the order that we write numbers.

As an example, we typically write decimal numbers in “big endian”, meaning that the leftmost digits are the “biggest”.

So, 123 = 100 + 20 + 3

If we were to use little endian instead, 123 would mean something different:

123 = 1 + 20 + 300.

How do computers use endianness?

Humans generally like to use big endian (it makes more sense).

However, computers are more efficient when they store numbers in little endian order (reverse order).

Note that it is not bits that are stored in the reverse order, it is BYTES. So if we take a big endian number:
00001010 00001111 11001100 11111111

Then all we need to do to convert to little endian is reverse the order of the BYTES

= 11111111 11001100 00001111 00001010

In this number, the leftmost byte is the smallest and the rightmost is the largest.

What exactly does that mean?

When we write in our code:

“int x = 23;”,

The computer stores this binary number:
00000000 00000000 00000000 00010111

And it stores it in little endian order, so in reality it would look like this in memory:

00010111 00000000 00000000 00000000

What about bitwise operations?

It seems like this would totally mess up bit shift operations and the like.

However, a C program “pretends” that the numbers are stored in big endian anyway.

Whenever we do bitwise operations, or read/write bits to numbers, C handles the conversion between big endian in our program and little endian in memory.

When you write a program, it’s appropriate to just pretend everything is in big endian.

Is there any observable difference?

There is one difference that you might have noticed when writing MIPS.

If you accidentally load an integer (word) using a lb instruction, sometimes we still get the correct result anyway.

This is the case because the numbers are stored in little endian.

Imagine we had the value 23 in little endian:

00010111 00000000 00000000 00000000

If we load only one byte, we still get the value 23:

00010111

If it was stored in big endian, we would just get 0.

Signed numbers

How can we display a negative number?

So far we can store numbers in binary notation, but there's no way to store negative numbers.

A very simple solution is to declare the leftmost bit as a "sign" bit, where 1 = negative and 0 = positive.

Then, the rest of the number is just the magnitude of the number.

3 = 0011

-3 = 1011

Why don’t we just do that?

This system is great because it’s really understandable.

However, implementing addition is difficult.

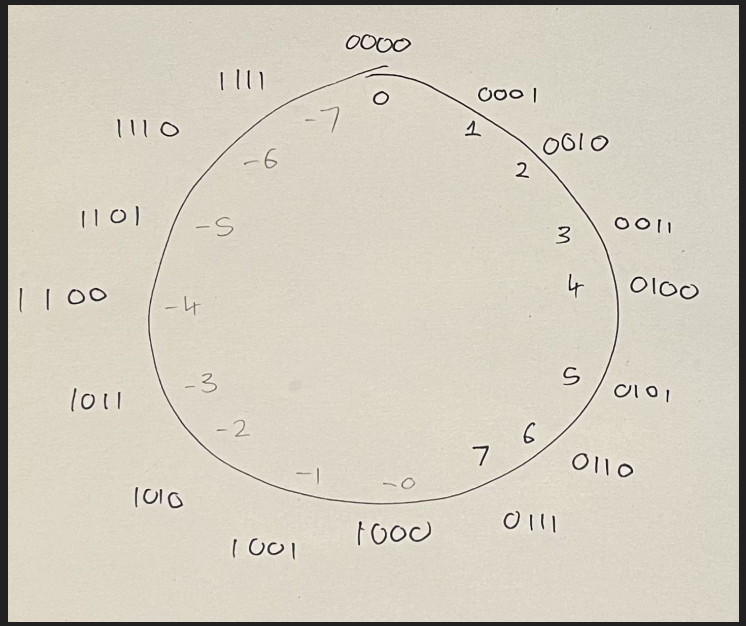
What happens if I perform -3 + 1?

1011 + 0001 = 1100 (-4).

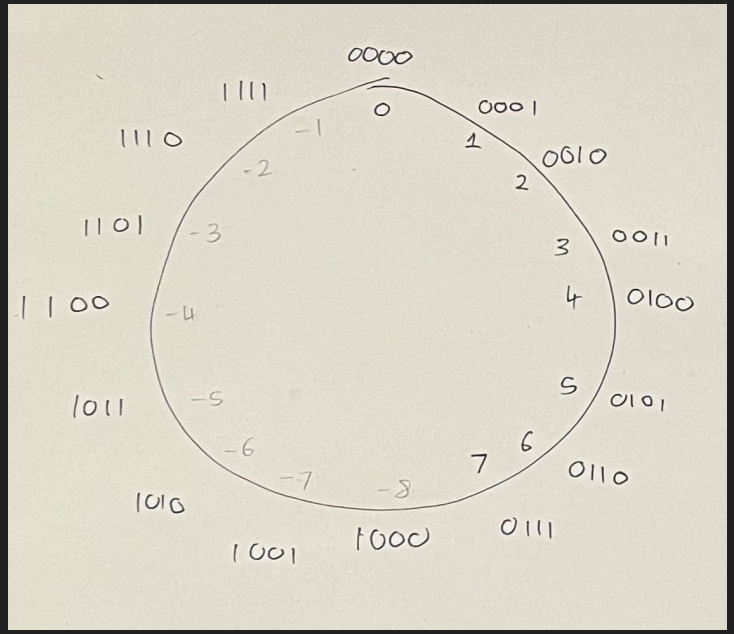
Ideally, if we do -3 + 1 we should get -2…

Negative numbers and addition

If we store the negative numbers in the opposite order, now adding 1 works properly



(normal)



(twos complement)

What about 1 + -1?

Ideally, -1 + 1 = 0.

1111 + 0001 = 10000.

But since our numbers are only 4 bits, we cut off the new 1.

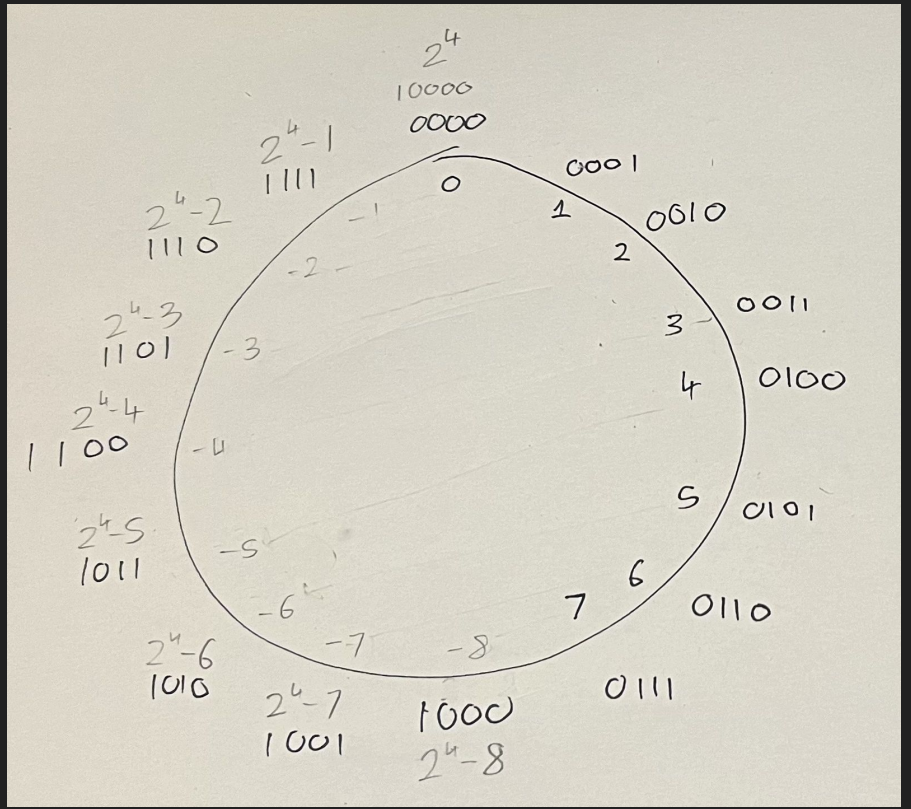
So we get 1111 + 0001 = 0000.

Which is what we want.

How does this work (maths)?

Mathematically, we represent a negative binary number as **2^4 - n**

We essentially record a negative number by a negative distance from 10000 (10000 = 2^4).

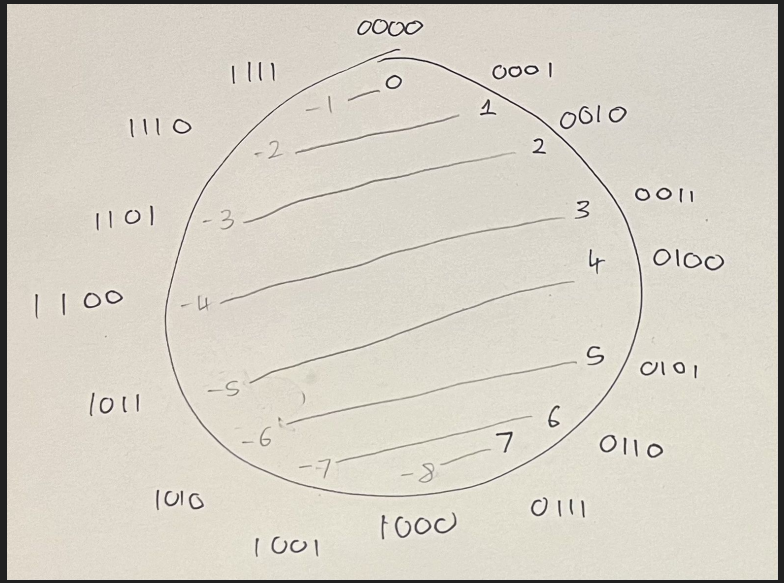


How to convert between positive and negative

You might notice that each positive number is horizontally opposite to its negative counterpart.

So, we can get pretty close by just flipping the bits.

To get the exact result, we find that we need to add one after we flip the bits.



Why does conversion work (maths)?

Take the number 2

-2 is just 2^4 - 2 (negative distance from 2^4)

-0010 = 10000 - 0010 (we can’t actually write 10000, so need to write 1 + 1111)

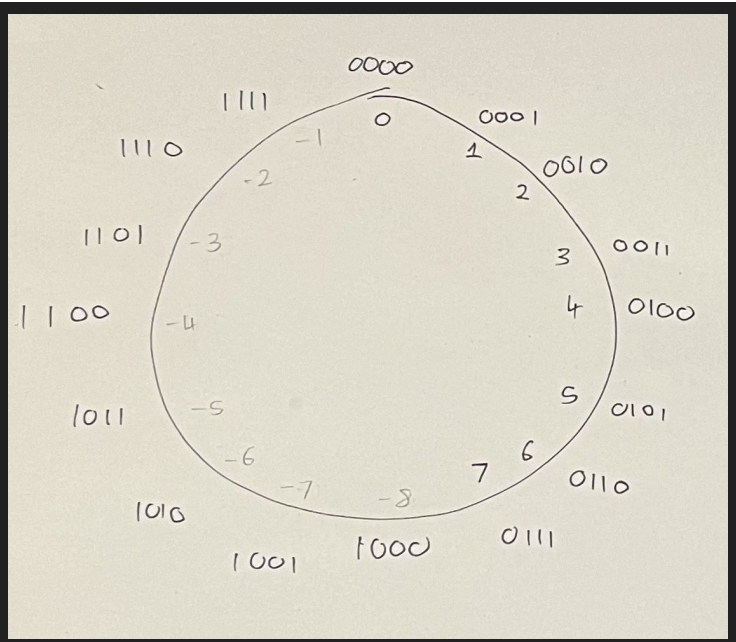
-0010 = (1 + 1111) - 0010 (1111 - 0010 is what flips the bits to 1101)

-0010 = 1 + 1101

Tutorial Q1

1. On a machine with 16-bit `int` s, the C expression `(30000 + 30000)` yields a negative result.

   Why the negative result? How can you make it produce the correct result?



Tutorial Q2

2. Assume that the following hexadecimal values are 16-bit twos-complement. Convert each to the corresponding decimal value.

   i. `0x0013`
   ii. `0x0444`
   iii. `0x1234`
   iv. `0xffff`
   v. `0x8000`

Tutorial Q3

Don’t spend too much time on this

3. Give a representation for each of the following decimal values in 16-bit twos-complement bit-strings. Show the value in binary, octal and hexadecimal.

   i. 1
   ii. 100
   iii. 1000
   iv. 10000
   v. 100000
   vi. -5
   vii. -100

Floating point numbers

How can we store fractions?

How do we write decimal fractions?
12.12

= 1 * 10^1 + 2 * 10^0 + 1 * 10^-1 + 2 * 10^-2

= 12.12 (10 + 2 + 0.1 + 0.002)

So, what if we do that for binary?

101.101

= 1 * 2^2 + 0 * 2^1 + 1*2^0 + 1*2^-1 + 0*2^-2 + 1*2^-3

= 5.625 (4 + 1 + 0.5 + 0.125)

Where should the decimal point go?

How about in the middle?
00000000 00000000 . 00000000 00000000

So we could have a number like

00001010 11001100 . 11001010 00100100

= 2764.7901611328125

(2048 + 512 + 128 + 64 + 8 + 4 + 1/2 + 1/4 + 1/32 + 1/128 + 1/1024 + 1/8192)

This is called “fixed point” notation.

Okay, but the range isn’t impressive

The largest number we can store is

11111111 11111111 11111111 11111111

= 65535.999969482421875

Another interesting idea, is to take advantage of something more like “scientific notation”.

If you are not familiar, we can write decimal numbers like

4.234 * 10^10 = 42340000000

This can be used to store fractions as well:

4.234 * 10^-4 = 0.0004234

Let’s do scientific notation in binary then

It should like like x * 2^n

Where x is a number between 1 and 2.

So instead of writing 3 * 2^3, we would write 1.5 * 2^4.

Instead of writing 0.7 * 2^3, we would write 1.4 * 2^2.

So we can write the number 2.5 as

1.25 * 2^1

= 1.01 * 2 ^ 1

IEEE 754 standard

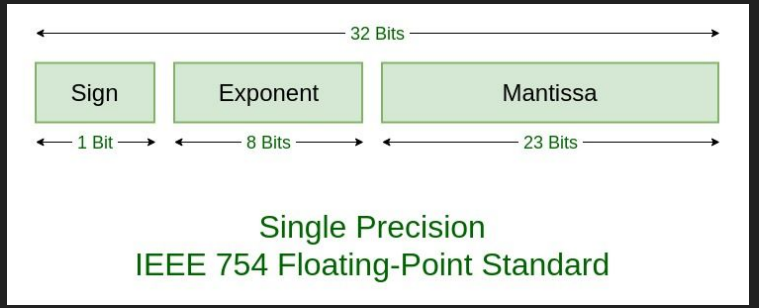
The IEEE 754 standard describes how we can store a floating point number (ie, using scientific notation) inside 32 bits.

We use the leftmost bit to represent sign (0=positive/1=negative).

The next 8 bits represent a signed exponent from -127 to 128.

Then, the last 23 bits represent a fraction to multiply the exponent by.

value = fraction * (2 ^ exponent)



IEEE 754 standard

To expand on the fraction part, it looks like this:

0000000 00000000 00000000

It actually stores a value between 0 and 1, and to find our actual fraction to multiply into the exponent, we should add 1.

So, if we wanted the number 1.625 * 2^7

The fraction would be

101000 00000000 00000000

(1 * 2^-1 + 0 * 2^-2 + 1 * 2^-3 … = 0.625, add 1 = 1.625)

IEEE 754 standard

To expand on the exponent part, it stores a signed value from -127 to 128.

However, for some reason (??) we don’t use twos complement to store that number.

We just store an unsigned value between 0-255, and then we take that value and subtract 127.

This means if we store the value 127, it really means an exponent of 127-127 = 0.

If we store the value 129, it really means an exponent of 129-127 = 2.

IEEE 754 standard

So, the full number 1.625 * 2^7 (208) is represented:

0 10000110 10100000000000000000

0 = positive

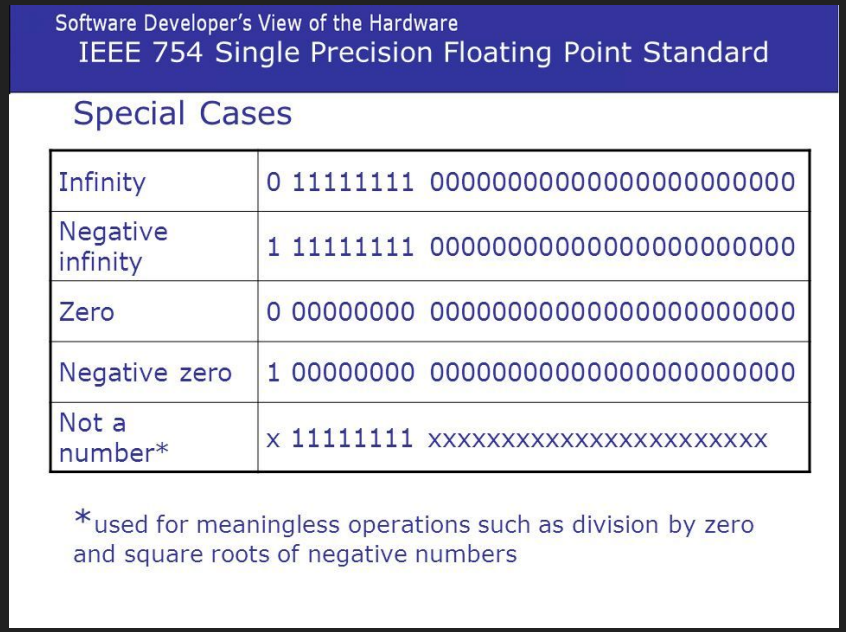
10000110 = 134. 134-127 = 7, so exponent = 7.

10100000000000000000 = 0.625, so fraction = 1.625.

So we get positive 1.625 * 2^7.

Special values

There are a few “special” cases which allow us to store nonstandard numbers



When programmers encounter these values, they typically handle them as edge cases. For example, division by zero might yield a “nan” value.

Tutorial Q4

4. What decimal numbers do the following single-precision IEEE 754-encoded bit-strings represent?

a. `0 00000000 00000000000000000000000`
b. `1 00000000 00000000000000000000000`
c. `0 01111111 10000000000000000000000`
d. `0 01111110 00000000000000000000000`
e. `0 01111110 11111111111111111111111`
f. `0 10000000 01100000000000000000000`
g. `0 10010100 10000000000000000000000`
h. `0 01101110 10100000101000001010000`

Each of the above is a single 32-bit bit-string, but partitioned to show the sign, exponent and fraction parts.

Tutorial Q5

5. Convert the following decimal numbers into IEEE 754-encoded bit-strings:

a. 2.5
b. 0.375
c. 27.0
d. 100.0

Bonus: Tutorial Q6

6. Write a C function, `six_middle_bits`, which, given a `uint32_t`, extracts and returns the middle six bits.