MSIA 413, Nikos Hardavellas
Fall 2022

Bits and Representations

Vijay Kandiah
TA section 1

1 Unsigned Integers

For a vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$

$$B2U_w(\vec{x}) := \sum_{i=0}^{w-1} x_i 2^i \tag{1}$$

where w is the number of bits (e.g $w = 32$ for a 32-bit computer), and $B2U_w$ is the function that coverts a binary to unsigned.

1.1 Examples

$$\begin{aligned}
B2U_4([0001]) &= 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\
B2U_4([0101]) &= 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\
B2U_4([1011]) &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11 \\
B2U_4([1111]) &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15
\end{aligned}$$

To convert integer to binary, start with the integer in question and divide it by 2 keeping notice of the quotient and the remainder. Continue dividing the quotient by 2 until you get a quotient of zero. Then just write out the remainders in the reverse order. [1]

Converting 263 to binary – keep dividing by 2 and keep track of remainder

Divisor	Quotient	Remainder
2	263	263%2 = 1
2	263//2 = 131	131%2 = 1
2	131//2 = 65	65%2 =1
2	32	0
2	16	0
2	8	0
2	4	0
2	2	0
2	1	1

Therefore, $263_{10} = 100000111_2$

[1] https://indepth.dev/the-simple-math-behind-decimal-binary-conversion-algorithms/

List of power of 2's

1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
32	2^5
64	2^6
128	2^7
256	2^8
512	2^9
1024	2^{10}
2048	2^{11}
...	...

Easy version:
$263_{10} = 256 + 7 = 256 + 4 + 3 = 256 + 4 + 2 + 1 = 2^8 + 2^2 + 2^1 + 2^0 = 100000111_2$

1.2 Hexadecimal

Hexadecimal representation uses base 16 rather than base 2. The logic is the same. We need 16 'digits' so we use $0, 1, 2, \ldots, 9, A, B, C, D, E, F$. The letters A, B, C, D, E, and F represent the values $10, 11, 12, 13, 14$, and 15 respectively. So $19 = 1 \cdot 16^1 + 3 \cdot 16^0 = 13_{16}$. That is, 13 in base 16 is equivalent to 19 in base 10

Question: Similar to when we converted 263 *to binary, design an algorithm that converts 263 to hexadecimal.*

2 Signed Integers

2.1 Two's Complement

For a vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$

$$B2T_w(\vec{x}) := -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2}$$

where w is the number of bits (e.g $w = 32$ for a 32-bit computer), and $B2T_w$ is the function that coverts a binary to two's complement.

The most significant bit x_{w-1} is also called the *sign bit.* Its 'weight' is -2^{w-1}, the negation of its weight is an unsigned representation. When the sign bit is set to 1, the represented value is negative, and when set to 0, the value is positive.

🐱 *Question: What is the min and the max value of a 32-bit signed int?*

2.1.1 Examples

$$\begin{aligned}
B2T_4([0001]) &= -0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+0+0+1 = 1\\
B2T_4([0101]) &= -0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+4+0+1 = 5\\
B2T_4([1011]) &= -1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+0+2+1 = -5\\
B2T_4([1111]) &= -1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+4+2+1 = -1
\end{aligned}$$

2.2 Two's Complement Inversion Trick

A fast way to find two's complement of a negative integer is to take the binary representation of the positive integer, flip all of the bits (0 to $1, 1$ to 0) and add 1. This trick is pretty useful for finding two's complement representation of negative numbers. So, first find the positive number then apply the inversion trick to get the negative.

3 Float Representation

Consider a notation of the form

$$b_m b_{m-1} \ldots b_1 b_0 \ . \ b_{-1} b_{-2} \ldots b_{-n+1} b_{-n}$$

This notation represents a number b defined as

$$b = \sum_{i=-n}^{m} 2^i \times b_i$$

The symbol '.' is the *binary point*, with bits on the left weighted by non-negative powers of 2, and those on the right weighted by negative powers of 2. So 101.11_2 represents the number $1\times 2^2 + 0\times 2^1 + 1\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2} = 4+0+1+\frac{1}{2}+\frac{1}{4} = 5\frac{3}{4}$.

3.1 IEEE Standard

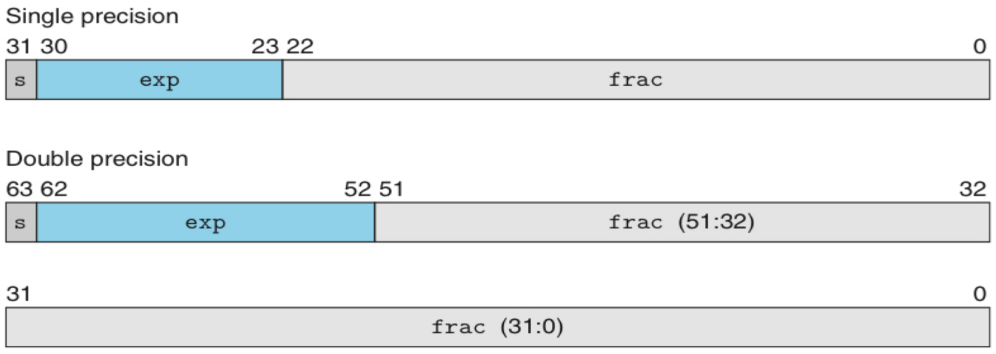

Figure 2.32 **Standard floating-point formats.** Floating-point numbers are represented by three fields. For the two most common formats, these are packed in 32-bit (single-precision) or 64-bit (double-precision) words.

The IEEE floating-point standard represents a number in a form

$$V := (-1)^s \times M \times 2^E$$

The bit representation of a floating point number is divided into three fields to encode these values:

- The single sign bit s directly encodes the sign s. The sign s determines whether the number is negative ($s = 1$) or positive ($s = 0$)
- The k-bit exponent field $exp = e_{k-1} \dots e_1 e_0$ encodes the exponent E. The *exponent* E weights the value by a power of 2.
- The n-bit fraction field $frac = f_{n-1} \dots f_1 f_0$ encodes the significand (or mantissa) M. The *significand* M represents the fractional binary number.

Single Precision (`float` in C) : the fields $s, exp, frac$ are $1, k = 8, n = 23$ each. A 32-bit representation
Double Precision (`double` in C) : the fields $s, exp, frac$ are $1, k = 11, n = 52$ each. A 64-bit representation

Normalized Values:

This is what we commonly see. It occurs when the bit pattern of *exp* field is neither all zeros (numeric value 0) or all ones (numeric value 255 for single precision). In this case, the exponent field is in the biased form. That is, exponent value is $E = e - Bias$, where e is the unsigned number having bit representation $e_{k-1} \dots e_1 e_0$ and $Bias$ is a bias equal to $2^{k-1} - 1$ (127 for single prec. and 1023 for double prec.). Yielding exponent ranges from -126 to $+127$ for single and -1022 $+1023$ for double. [2]

[2] In single precision floating point, you get 8 bits in which to store the exponent. Instead of storing it as a signed two's complement, it was decided it'd be easier to just add 127 to the exponent (since the lowest it could be in 8 bit signed is -127) and just store it as an unsigned number. If the stored value is greater than the bias, that means the value of the exponent is positive, if it's lower than the bias, it's negative, if it's equal, it's zero

The fraction field $frac$ represents the fractional value f, having binary representation $0.f_{n-1}\dots f_1 f_0$.

The significand/mantissa is $M = 1 + f$, sometimes called *implied leading 1* representation, because we can view M to be the number with binary representation $1.f_{n-1}f_{n-2}\dots f_0$.

Thus, a 32-bit floating point number is $(-1)^s \times (1.frac)_2 \times 2^{exp-127}$.

Special Values

s	exp	frac	Meaning
0	All 0's	All 0's	+0.0
1	All 0's	All 0's	−0.0
0	All 0's	Not 0	+Denormalized
1	All 0's	Not 0	−Denormalized
0	All 1's	All 0's	$+\infty$
1	All 1's	All 0's	$-\infty$
0	All 1's	Not 0	*NaN*
1	All 1's	Not 0	*NaN*

4 Why do we care about all of this ¯_(ツ)_/¯

4.1 Underflow Overflow and Spooky behavior

4.1.1 Unsigned Int's

Unsigned integers are defined by modular arithmetic, values that are too large wrap around to the smallest values in the range, and values that are too small wrap around to the largest values in the range. So, simple arithmetic on the value like addition, multiplication, and subtraction can result in a value that can't be represented with 32-bits .

4.1.2 Signed Int's (Two's complement)

Signed integers are slightly different. When an overflow or underflow condition occurs on signed integers the result will wrap around the sign and causes a change in sign. For example: $0\underbrace{11\dots1}_{31\text{ times}}{}_2$ or $2^{31}-1$ is the largest 32-bit number that two's complement can represent. If we add 1 to this number the results will be $1\underbrace{00\dots0}_{31\text{ times}}{}_2$ equivalent to -2^{31} in decimal.

4.1.3 Floats

Consider this very simple program.

1: Simple Python subtraction

```
>>> 1.2 - 1.0
0.19999999999999996
```

This is not a bug in Python. Some decimal numbers can't be represented exactly in binary, resulting in small roundoff errors. This is similar to decimal math, e.g $1/3 = 0.333333$ can't be represented by a fixed number of digits.

Floating-point numbers suffer from *loss of precision* when represented with a fixed number of bits (e.g., 32-bit or 64-bit). One of the consequences of this is that it is dangerous to compare the result of some computation to a float with == ! Tiny inaccuracies may mean that == fails. Instead, calculate the absolute difference and compare with the machine epsilon.

Because floating-point numbers have a limited number of digits, they cannot represent all real numbers accurately: when there are more digits than the format allows, the leftover ones are omitted - the number is rounded. For example, with a floating point format that has 2 digits in the significand (e.g., $1.xx_2$), 1000_2 does not require rounding, and neither does 10000_2 or 1110_2 - but 1001_2 will have to be rounded. Why? Because leading and trailing zero's (within the range provided by the exponent) don't need to be stored. This is a problem of *too many significant digits*.[3]

[3] For more see https://floating-point-gui.de/ 'What every Programmer Should Know About Floating-Point Arithmetic!?!'

5 Practice Questions

Question 1

Assuming $w = 4$ we can assign a numeric value to each possible hexadecimal digit, assuming either and unsigned or a two's complement interpretation. Fill in the following table according to these interpretations by writing out the nonzero powers of 2 in the summations.

Hexadecimal	Binary	Unsigned	2's Complement
0xA			
	[1011]		
		7	
			-4

Question 2

What is the largest positive finite number that can be represented by IEEE single precision float?