Programming & Data Structure

CS 11002

Partha Bhowmick

http://cse.iitkgp.ac.in/~pb

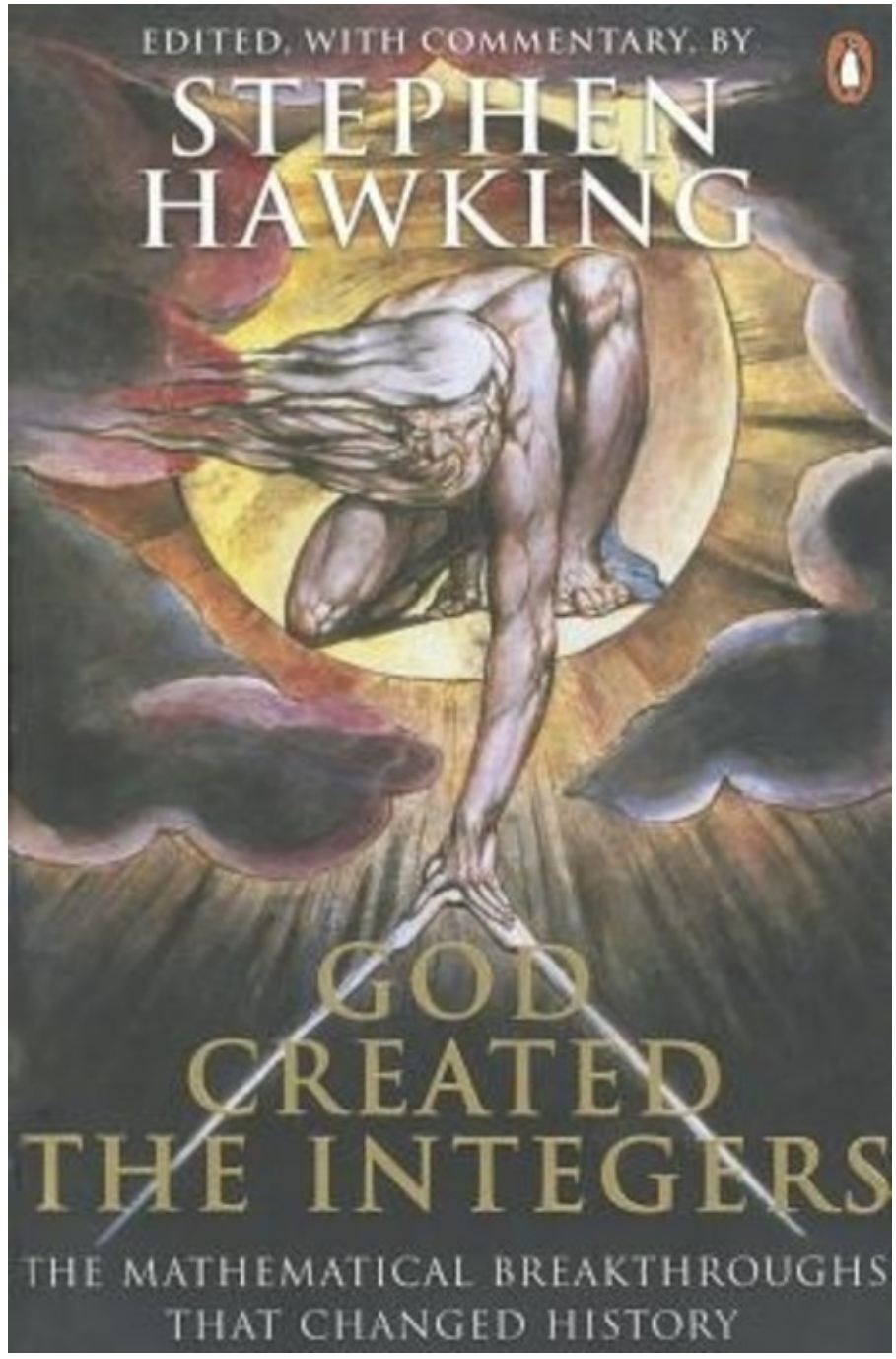
CSE Department
IIT Kharagpur

Spring 2012-2013

Integer Representation

God created the integers, all the rest is the work of man.

— L. Kronecker



Decimal Number System (1)

- Basic symbols: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
- *Radix-10* positional number system. The radix is also called the *base* of the number system.

$$12304 = 1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 4 \times 10^0$$

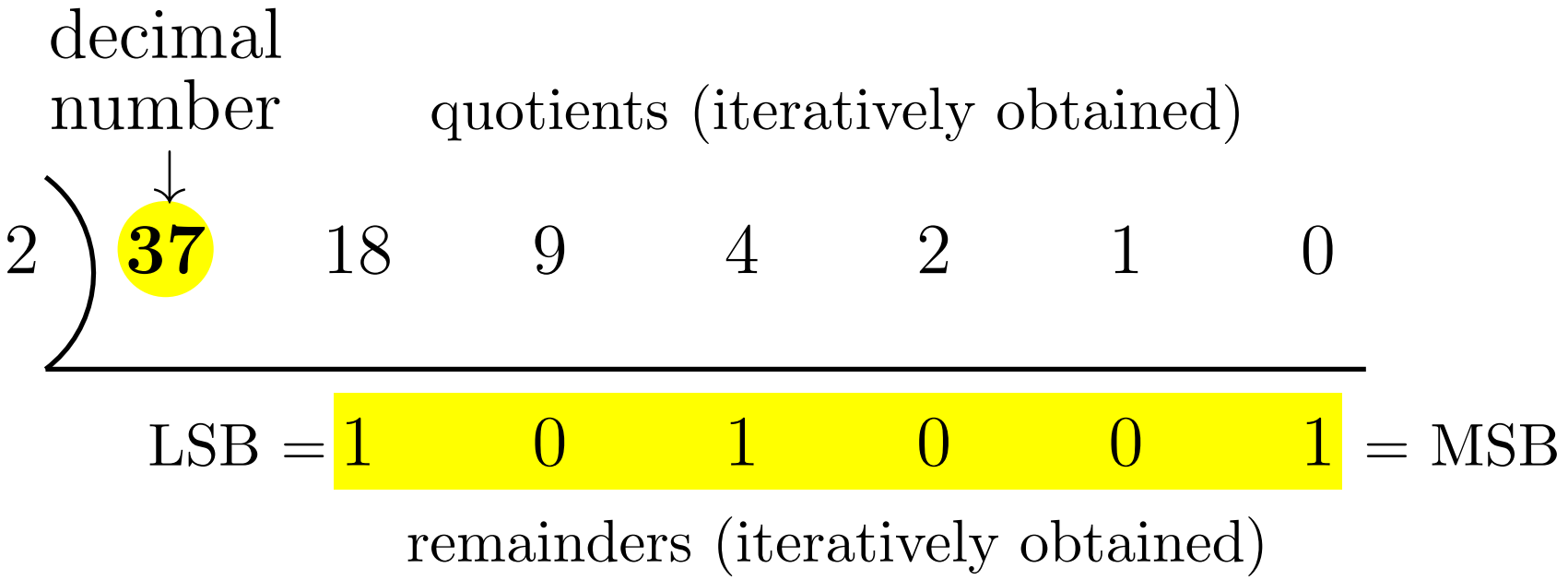
Unsigned Binary Number System (1)

- Basic symbols: `0`, `1`
- Radix-2 positional number system.

$$10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

The value is 22 in decimal.

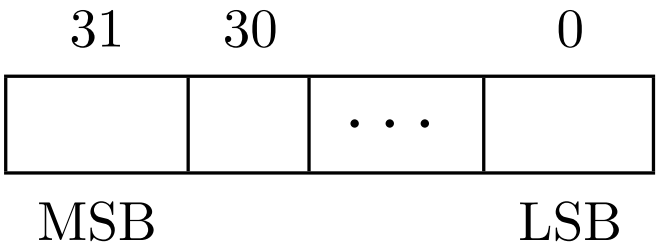
Unsigned Binary Number System (2)



$$37_{10} = 2^0 + 2^2 + 2^5 = 100101_2$$

Note: MSB = most significant bit, LSB = least significant bit.

word Size of the CPU



Largest Number: $\sum_{i=0}^{31} 2^i = 4294967295$

Smallest Number: 0

Decimal value (ex: 4-bit word) (1)

Bit String				*Decimal Value*
b_3	b_2	b_1	b_0	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

Bit String				*Decimal Value*
b_3	b_2	b_1	b_0	
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Signed Decimal Number (1)

- We use + and − symbols to indicate the sign of a decimal number.
- In a binary system, only 0 and 1are available to encode *any information*. So one extra bit is required to indicate the sign of a number. This bit is called the *sign bit*.

Three Popular Schemes

- *Signed Magnitude*
- *1's Complement*
- *2's Complement*

Signed Magnitude (1)

Consider a 4-bit word as an example.

b_3	b_2	b_1	b_0

b_3 is the *sign bit*; it is `0` for a positive number and `1` for a negative number.
Other three bits represent the magnitude of the number.

Signed Magnitude

Bit String				*Decimal Value*
b_3	b_2	b_1	b_0	
0	0	0	0	+0
0	0	0	1	+1
0	0	1	0	+2
0	0	1	1	+3
0	1	0	0	+4
0	1	0	1	+5
0	1	1	0	+6
0	1	1	1	+7

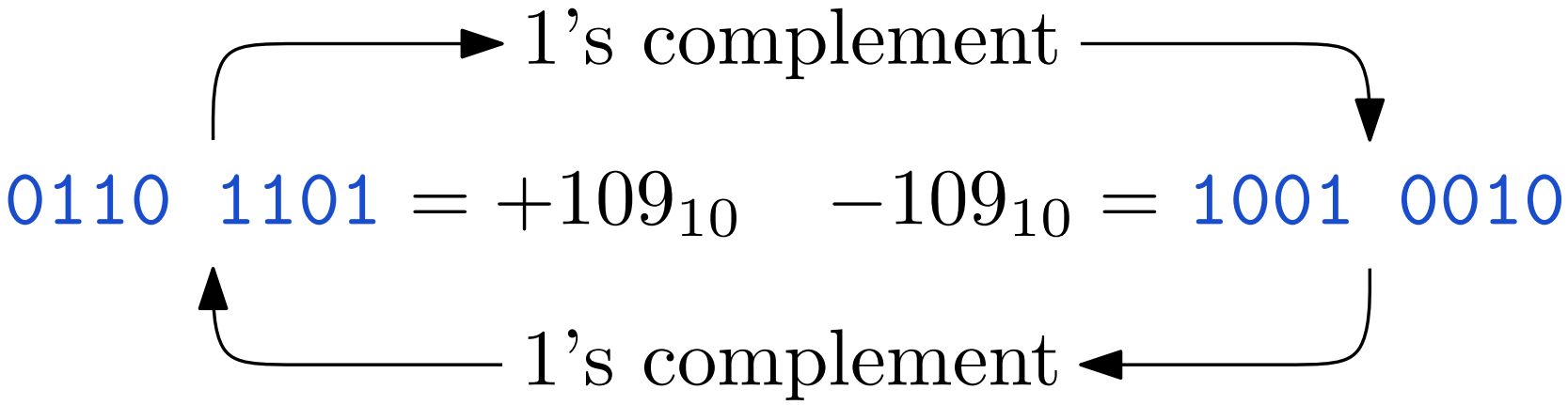
Bit String				*Decimal Value*
b_3	b_2	b_1	b_0	
1	1	1	1	−7
1	1	1	0	−6
1	1	0	1	−5
1	1	0	0	−4
1	0	1	1	−3
1	0	1	0	−2
1	0	0	1	−1
1	0	0	0	−0

- Two representations of `zero`: $+0, -0$
- Range for 4 bits: $-7 \cdots +7$
- Range for n bits: $-(2^{n-1}-1) \cdots +(2^{n-1}-1)$

1's Complement Numeral (1)

- Positive numbers are same as the signed magnitude representation with MSB = 0.
- A negative number has MSB = 1.
- If n is a number in 1's complement form, then $-n$ is obtained by changing every bit to its complement. The result is called the *1's complement* of n.

1's Complement Numeral (2)



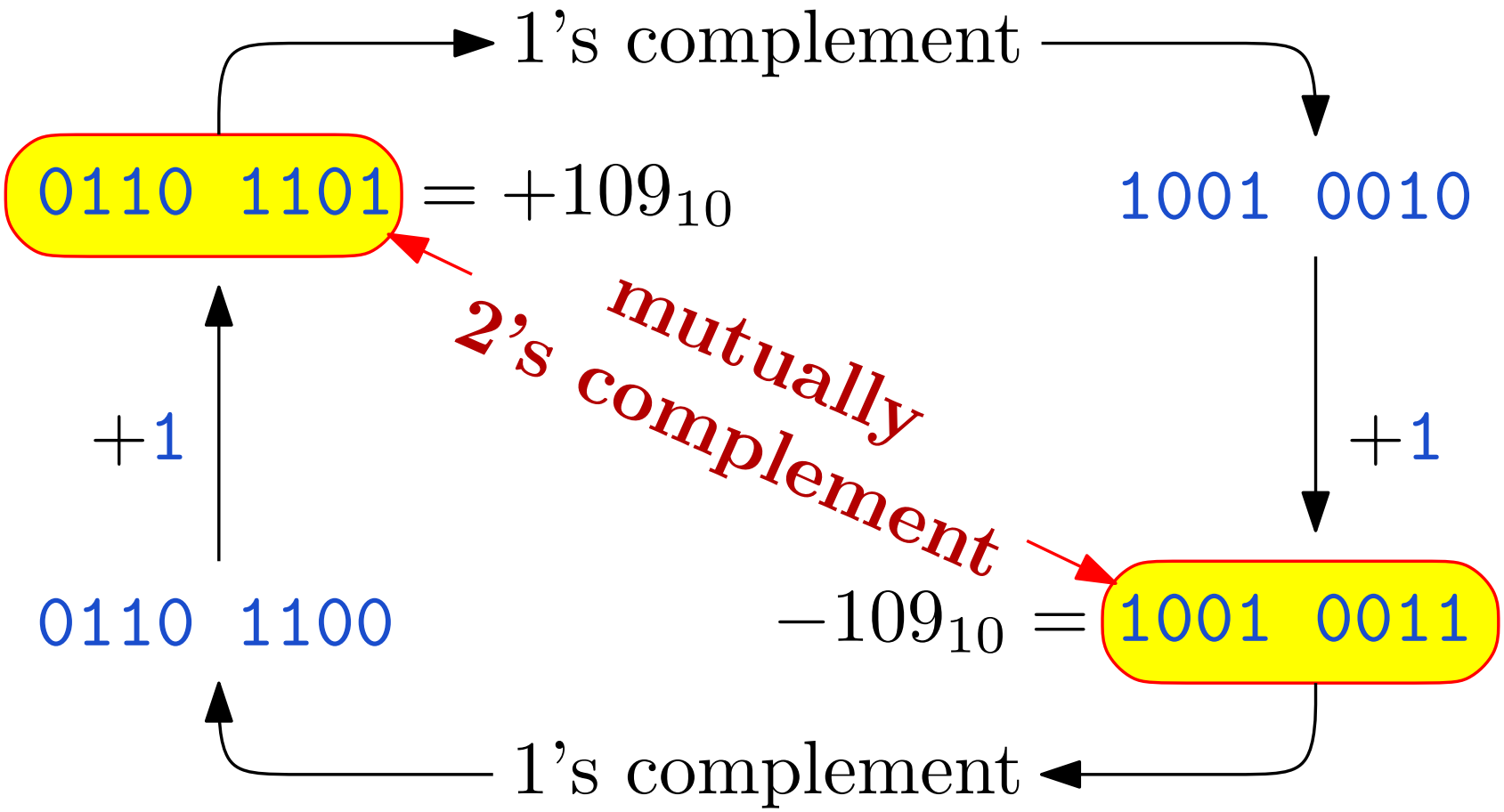
1's Complement Numeral (3)

Decimal Value	*Bit String*				*Bit String*				*Decimal Value*
	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	
$+7$	0	1	1	1	1	0	0	0	-7
$+6$	0	1	1	0	1	0	0	1	-6
$+5$	0	1	0	1	1	0	1	0	-5
$+4$	0	1	0	0	1	0	1	1	-4
$+3$	0	0	1	1	1	1	0	0	-3
$+2$	0	0	1	0	1	1	0	1	-2
$+1$	0	0	0	1	1	1	1	0	-1
$+0$	0	0	0	0	1	1	1	1	-0

2's Complement Numeral (1)

- Positive numbers are same as the signed magnitude and 1's complement representations with MSB = 0.
- If n is a number in 2's complement form, then $-n$ is obtained by changing every bit to its complement and finally adding 1 to it. The result is called the *2's complement* of n.
- A negative number has MSB = 1.

2's Complement Numeral (2)



Only exception: 1000 0000 $= -2^7 = -128$ whose 2's complement is itself, not +128.

2's Complement Numeral (3)

- Only one representation of zero: `0000`.
- Positive representation is identical to signed magnitude and 1's complement, but the negative representation is different.
- Range for 4 bits: $-8 \cdots +7$.
- Range for n bits: $-(2^{n-1}) \cdots +(2^{n-1}-1)$.
- Range of `int` (32 bits): -2147483648 to 2147483647.

2's Complement Numeral

Decimal Value	*Bit String*				*Bit String*				*Decimal Value*
	b_3	b_2	b_1	b_0	b_3	b_2	b_1	b_0	
$+7$	0	1	1	1	1	0	0	0	-8
$+6$	0	1	1	0	1	0	0	1	-7
$+5$	0	1	0	1	1	0	1	0	-6
$+4$	0	1	0	0	1	0	1	1	-5
$+3$	0	0	1	1	1	1	0	0	-4
$+2$	0	0	1	0	1	1	0	1	-3
$+1$	0	0	0	1	1	1	1	0	-2
0	0	0	0	0	1	1	1	1	-1

Signed Magnitude, 1's and 2's Complements

Decimal	Sig. Mag.	1's Compl.	2's Compl.
−0	1000	1111	
−1	1001	1110	1111
−2	1010	1101	1110
−3	1011	1100	1101
−4	1100	1011	1100
−5	1101	1010	1011
−6	1110	1001	1010
−7	1111	1000	1001
−8			1000

2's Complement to Decimal (1)

If the 2's complement number is *positive*, e.g., `0110 1010`, then its decimal value is as usual

$$2^6 + 2^5 + 2^3 + 2^1 = 64 + 32 + 8 + 2 = 106.$$

If the number is *negative*, e.g., `1110 1010`, then its value is `−(1 0000 0000 − 1110 1010)`, i.e.,

$$\begin{aligned} & -[2^8 - (2^7 + 2^6 + 2^5 + 2^3 + 2^1)] \\ = \; & -2^7 + 2^6 + 2^5 + 2^3 + 2^1 \\ = \; & -22. \end{aligned}$$

Sign Bit Extension

−39	25
1011001	0011001
1 1011001	0 0011001
11 1011001	00 0011001
111 1011001	000 0011001
1111 1011001	0000 0011001
11111 1011001	00000 0011001

2's Complement Addition (1)

0011	3	1101	−3
+ 0010	+ 2	+ 1110	+ −2
0101	5	1 1011	−5
0011	3	0100	4
+ 1011	+ −5	+ 0101	+ 5
1110	−2	1001	−7
		Overflow	

2's Complement Addition (2)

	1101		−3
+	1010	+	−6
1	0111		7

Overflow

Caution!

There may be *carry-out* without *overflow*.
So, only *carry-out* is no indicator of *overflow*.

2's Complement Addition (3)

`int` in Your Machine

The `int` in your machine (gcc-Linux on Pentium) is a 32-bit 2's complement number. Its range is -2147483648 to $+2147483647$. If `1` is added to the *largest positive* number, then the result is the *smallest negative* number.

`0111 1111 1111 1111 1111 1111 1111 1111` `+1`	2147483647
`1000 0000 0000 0000 0000 0000 0000 0000`	-2147483648

10's Complement Number (1)

The 2's complement numeral is nothing special. We can use *radix-complement* numerals for any radix to represent signed numbers without using any sign symbol.
For example, for decimal system, we consider 10's complement numerals.

There are one thousand patterns (000 to 999) with three decimal digits. We interpret a 3-digit decimal number n in the following way:

10's Complement Number (2)

- If the *most significant digit* is any one of *0, 1, 2, 3, 4*, then n is a usual positive number, e.g., *341* is same as the usual decimal 341.
- If the *most significant digit* is one of *5, 6, 7, 8, 9*, then n is treated as a negative number.
- If n is a 10's complement number, then $-n$ is obtained by $1000 - n$.

10's Complement Number

Example

Consider the 3-digit 10's complement numeral *725*.

It is a negative number with actual value: $-(1000 - 725) = -275$.

The range of numbers represented in 3 digits is $-10^3/2$ to $+10^3/2 - 1$, i.e., -500 to 499.

For *n-digit* 10's complement numbers, the range is $-10^n/2$ to $+10^n/2 - 1$.

10's Complement Number

Addition of 10's Complement Numeral

$$\begin{array}{rr} & 127 \\ + & 205 \\ \hline & 332 \\ & 427 \\ + & 305 \\ \hline & \mathbf{7}32 \end{array}$$

overflow

$$\begin{array}{rr} & 127 \\ + & 821\ (-179) \\ \hline & 948\ (-52) \end{array}$$

$$\begin{array}{rr} & 821\ (-179) \\ + & 753\ (-247) \\ \hline \not{1} \leftarrow & 574\ (-426) \\ & 521\ (-179) \\ + & 753\ (-247) \\ \hline \not{1} \leftarrow & 274 \end{array}$$

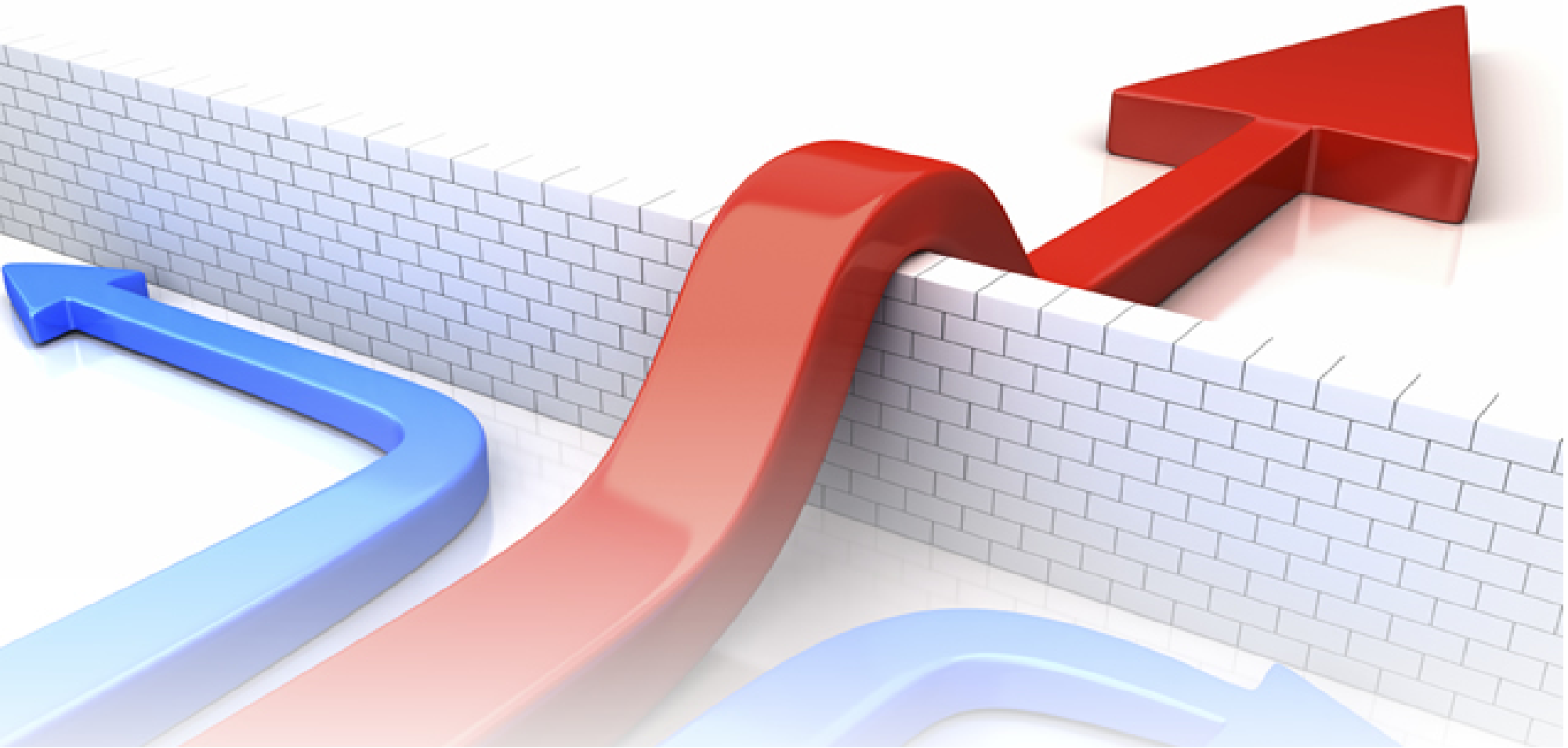
Caution! *Overflow* & *carry-out* as in 2's complement addition.

10's Complement Number

Digit Extension

3-digit	4-digit	5-digit	Decimal value
234	0234	00234	234
721	9721	99721	−279

Real Number Representation



IEEE 754 Floating-Point Format

Floating-Point Decimal Number (1)

$$
\begin{aligned}
-123456 \times 10^{-1} &= 12345.6 \times 10^{0} \\
&= 1234.56 \times 10^{1} \\
&= 123.456 \times 10^{2} \\
&= 12.3456 \times 10^{3} \\
&= 1.23456 \times 10^{4} \quad \textit{(normalized)} \\
&\approx 0.12346 \times 10^{5} \\
&\approx 0.01235 \times 10^{6}
\end{aligned}
$$

Floating-Point Decimal Number (2)

- There are different representations for the same number and there is *no fixed position* for the decimal point.
- Given a fixed number of digits, there may be a loss of precision.
- Three pieces of information represent a number: *sign*, *significand/mantissa*, *signed exponent of 10* (s, m, e).

  Ex: $-2.71 \times 10^5 \Rightarrow s = -1, m = 2.71, e = +5.$

Floating-Point Decimal Number (3)

Advantage

Given a fixed number of digits, the floating-point representation covers a *wider range* compared to fixed-point representation.

Example: Consider numbers with 6 digits, of which two are after the decimal point. Then the ranges are:

Fixed-point: 0.0 to 9999.99.
Floating-point[1]: $0.0, 0.001 \times 10^0$ to 9.999×10^{99}.

[1] *radix 10* in floating-point representation is implicit.

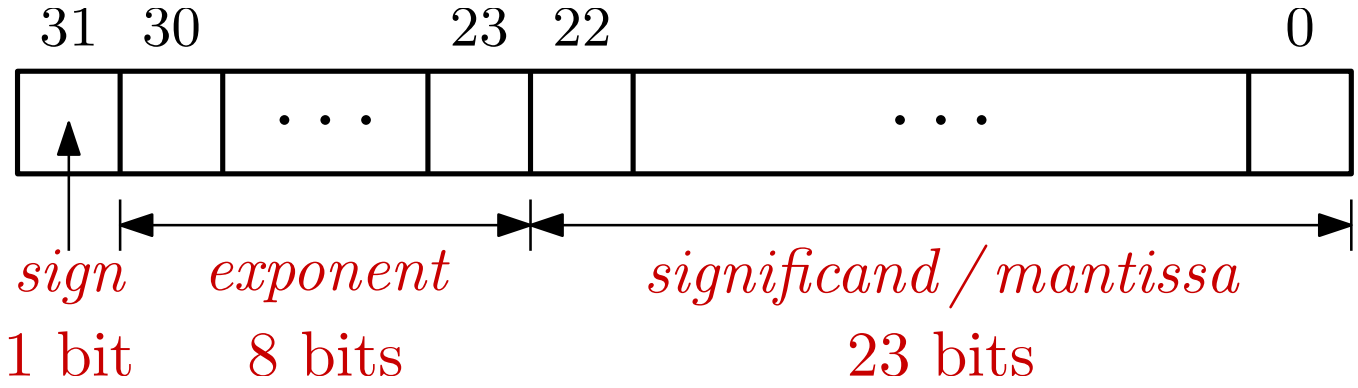
IEEE 754 Standard (1)

- Most of the binary floating-point representations follow the IEEE-754 standard.[2]
- The data type `float` uses *IEEE 32-bit single precision* format, and the data type `double` uses *IEEE 64-bit double precision* format.
- A floating-point constant is treated as a double precision number by GCC.

IEEE 754 Standard

IEEE 754 *Single Precision*

$(|s| + |e| + |m| = 1 + 8 + 23 = 32$ **bits**$)$

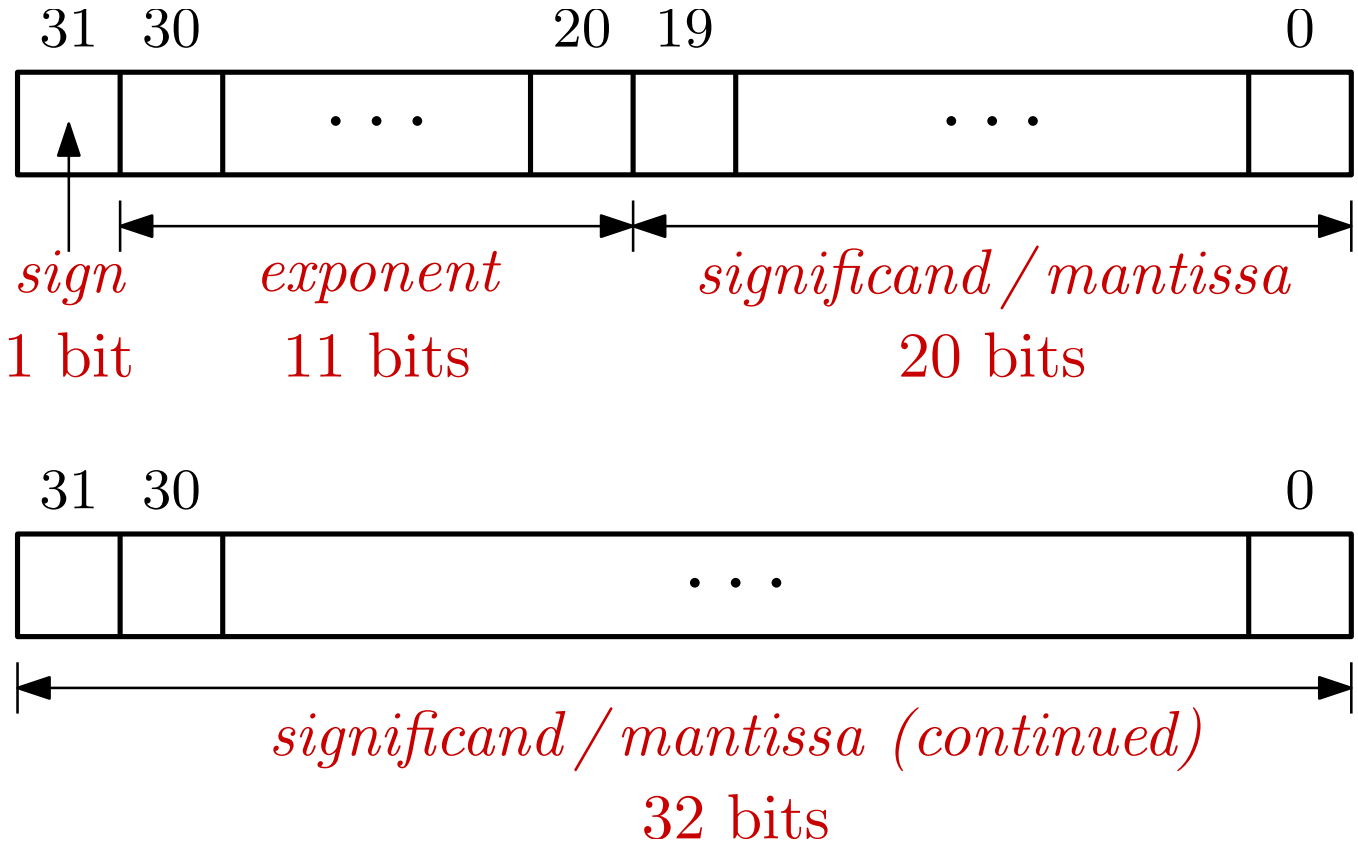


Bit 31	*sign bit*, s $(0 \Rightarrow +ve, 1 \Rightarrow -ve)$
Bits 30–23	***biased*** *exponent*, e
Bits 22–0	*mantissa* (significand or fraction), m

IEEE 754 Standard

IEEE 754 *Double Precision*

$(|s| + |e| + |m| = 1 + 11 + 52 = 64$ **bits**$)$



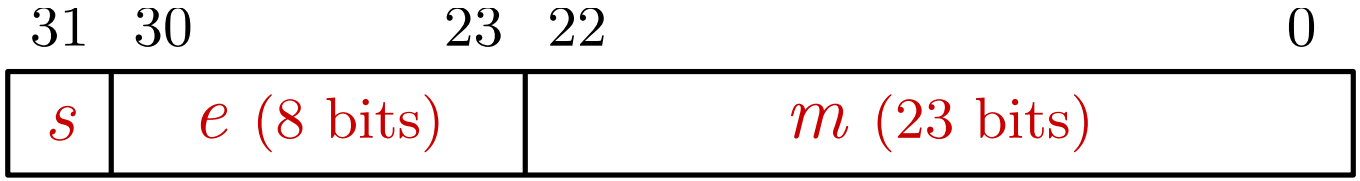
[2]IEEE stands for *Institute of Electrical and Electronics Engineers*.

IEEE 754 *Single Precision* (1)

Five Classes of `float`

- Normalized numbers
- Zeros
- Subnormal (de-normal) numbers
- Infinity
- not-a-number (`nan`)

Single-Precision Normalized Number (1)



Normalized Exponent

- $e = 0^8$ and 1^8 reserved for ± 0, `nan`, `inf`.
- For normalized numbers, $e = 1, 2, \ldots, 254$.
- *Normalized (biased) exponent* $= e - 127$.
  So, minimum exponent $= 1 - 127 = -126$,
  maximum exponent $= 254 - 127 = 127$.
- Normalized range: 2^{-126} *to* 2^{127}.

Single-Precision Normalized Number (2)

Normalized Significand: $1.m$ with the *implicit* 1 before 23-bit m and a *binary dot*.
Range: 1.0 *to* $(2.0 - 2^{-23})$.

Normalized Number: $\boxed{(-1)^s \times 1.m \times 2^{e-127}}$.

Example

1	1011 0110	011 0000 0000 0000 0000 0000

$$\begin{aligned}
\text{Nor. value} &= (-1)^{1} \times 1.011 \times 2^{10110110-01111111} \\
&= -1.375 \times 2^{55} \\
&= -49539595901075456.0 \\
&= -4.9539595901075456 \times 10^{16}
\end{aligned}$$

Single-Precision Normalized Number (3)

Reverse Example

Consider +105.625. Its binary representation is

$$
\begin{aligned}
& +1101001.101 \\
= \; & +1.101001101 \times 2^{6} \\
= \; & +1.101001101 \times 2^{133-127} \\
= \; & +1.101001101 \times 2^{10000101-01111111}
\end{aligned}
$$

IEEE 754 format:

0	1000 0101	101 0011 0100 0000 0000 0000

Single-Precision Normalized Number (4)

Reverse Example 2

Consider +2.7. Its binary representation is

$$
\begin{aligned}
& +10.10\ 1100\ 1100\ 1100\cdots \\
= \ & +1.010\ 1100\ 1100\cdots \times 2^{1} \\
= \ & +1.010\ 1100\ 1100\cdots \times 2^{128-127} \\
= \ & +1.010\ 1100\cdots \times 2^{10000000-01111111}
\end{aligned}
$$

IEEE 754 format (*LSB approximated*):

0	1000 0000	010 1100 1100 1100 1100 1101

Count of Single-Precision Normalized Numbers

Count of 23-bit significand $= 2^{23}$.

So, count of normalized numbers within the range $[2^{-i}, 2^{-i+1})$ is 2^{23} $\forall$ $i \in [-126, 126]$.

So, total count of +ve numbers is $\boxed{253 \cdot 2^{23}}$.

On including ± 0 and $-$ve numbers, we get $\boxed{253 \cdot 2^{24} + 2}$.

Note: These numbers are non-uniformly spaced over the real-number line.

Count of Single-Precision Normalized Numbers

Count of 23-bit significand $= 2^{23}$.

So, count of normalized numbers within the range $[2^{-i}, 2^{-i+1})$ is 2^{23} $\forall$ $i \in [-126, 126]$.

So, total count of +ve numbers is $\boxed{253 \cdot 2^{23}}$.

On including ± 0 and $-$ve numbers, we get $\boxed{253 \cdot 2^{24} + 2}$.

Note: These numbers are non-uniformly spaced over the real-number line.

Count of Single-Precision Normalized Numbers

Count of 23-bit significand $= 2^{23}$.

So, count of normalized numbers within the range $[2^{-i}, 2^{-i+1})$ is 2^{23} $\forall$ $i \in [-126, 126]$.

So, total count of +ve numbers is $\boxed{253 \cdot 2^{23}}$.

On including ± 0 and $-$ve numbers, we get $\boxed{253 \cdot 2^{24} + 2}$.

Note: These numbers are non-uniformly spaced over the real-number line.

Count of Single-Precision Normalized Numbers

Count of 23-bit significand $= 2^{23}$.
So, count of normalized numbers within the range $[2^{-i}, 2^{-i+1})$ is 2^{23} $\forall$ $i \in [-126, 126]$.
So, total count of +ve numbers is $\boxed{253 \cdot 2^{23}}$.
On including ± 0 and $-$ve numbers, we get $\boxed{253 \cdot 2^{24} + 2}$.

Note: These numbers are non-uniformly spaced over the real-number line.

Count of Single-Precision Normalized Numbers

Count of 23-bit significand $= 2^{23}$.
So, count of normalized numbers within the range $[2^{-i}, 2^{-i+1})$ is 2^{23} $\forall$ $i \in [-126, 126]$.
So, total count of +ve numbers is $\boxed{253 \cdot 2^{23}}$.
On including ± 0 and $-$ve numbers, we get $\boxed{253 \cdot 2^{24} + 2}$.

Note: These numbers are non-uniformly spaced over the real-number line.

Bit Patterns

Bit Patterns

Precision			
Single	**Double**		
Exponent		*Significand*	*Data Class*
0	0	0	$\pm$ 0
0	0	$\neq$ 0	$\pm$ subnormal
1 - 254	1 - 2046	anything	$\pm$ *normalized*
255	2047	0	$\pm\infty$
255	2047	$\neq$ 0	`nan`

nan

There are two types of `nan`s: *quiet* `nan` and *signaling* `nan`.

A few cases where we get `nan`:

- $0.0/0.0$
- $\pm\infty/\pm\infty$, $0\times\pm\infty$, $-\infty+\infty$
- $\sqrt{-1.0}$
- $\log(-1.0)$

To check them, please write C programs.

nan (2)

```
#include <stdio.h>
#include <math.h>
int main(){ // nan.c
  printf("0.0/0.0: %f\n", 0.0/0.0);
  printf("inf/inf: %f\n", (1.0/0.0)/(1.0/0.0));
  printf("0.0*inf: %f\n", 0.0*(1.0/0.0));
  printf("-inf + inf: %f\n", (-1.0/0.0) + (1.0/0.0));
  printf("sqrt(-1.0): %f\n", sqrt(-1.0));
  printf("log(-1.0): %f\n", log(-1.0));
  return 0;}
```

nan


```
$ cc -Wall nan.c -lm
$ a.out
0.0/0.0: -nan
inf/inf: -nan
0.0*inf: -nan
-inf + inf: -nan
sqrt(-1.0): -nan
log(-1.0): nan
$
```

nan

Not a number: `signaling nan` `0 11111111 00000000000000000000001`
Not a number: `quiet nan` `0 11111111 10000000000000000000001`
Infinity: `inf` `0 11111111 00000000000000000000000`
Largest Normal: $(2 - 2^{-23}) \times 2^{127} = 3.403e + 38$ `0 11111110 11111111111111111111111`
Smallest Normal: $1.0 \times 2^{-126} = 1.175e - 38$ `0 00000001 00000000000000000000000`

Subnormals or De-normals (1)

Largest De-normal:

$(2 - 2^{-23}) \times 2^{-126} = 1.175e - 38$

`0 00000000 11111111111111111111111`

Smallest De-normal:

$2^{-126-23} = 2^{-149} = 1.401e - 45$

`0 00000000 00000000000000000000001`

Zero: `0.000000e+00`

`0 00000000 00000000000000000000000`

Subnormals or De-normals (2)

- For a subnormal number, $e = 0$ and $m \neq 0$. There is no implicit 1 in the significand; so the value is $(-1)^s \times 0.m \times 2^{-126}$.
- The smallest subnormal: 2^{-149} is closer to 0.
- The largest subnormal: $0.99999988 \times 2^{-126}$ is closer to the smallest normalized number 1.0×2^{-126}.
- Due to the presence of the subnormal numbers, there are 2^{23} numbers within the range $[0.0, 1.0 \times 2^{-126})$.

Subnormals or De-normals (3)

- The smallest difference between two *normalized* numbers is 2^{-149}. This is same as the difference between any two consecutive *subnormal* numbers.
- The largest difference between two consecutive *normalized* numbers is 2^{104}.

inf

Infinity

`inf` = `1111 1111` `000 0000 0000 0000 0000 0000`
(without sign bit) is greater than (as an unsigned integer) the largest normal number:

`1111 1110` `111 1111 1111 1111 1111 1111`

Caution: Largest `int` +1 = ∞!

If we treat the largest normalized number as an `int` data and add `1` to it, then we get ∞.

[See code `inf.c` in next slide]

inf


```
#include <stdio.h>
int main(){ // inf.c
    float f = 1.0/0.0;
    int *iP;
    printf("f: %f\n", f);
    iP = (int *)&f;  --(*iP);
    printf("f: %f\n", f);
    return 0;}
```

```
$ cc -Wall inf.c
$ ./a.out
f:  inf
f:  340282346638528859811704183484516925440.000000
```

inf

Using inf

inf can be used in a computation, e.g., to compute $\tan^{-1}\infty$.

```
#include <stdio.h>
#include <math.h>
int main(){ \\atanInf.c
  float f = 1.0/0.0;
  printf("atan(%f) = %f\n",f,atan(f));
  printf("1.0/%f = %f\n", f, 1.0/f); return 0;}
```

```
$ cc -Wall atanInf.c
$ ./a.out
atan(inf) = 1.570796
1.0/inf = 0.000000
```

±0

(No) Caution! ±0

There are two zeros (±0) in IEEE representation, but testing their equality gives TRUE.

```
#include <stdio.h>
int main(){ //twoZeros.c
  double a = 0.0, b = -0.0;
  if(a == b) printf("Equal\n");
  else printf("Unequal\n");
  return 0;}
```

```
$ cc -Wall twoZeros.c
$ a.out
Equal
```