计算物理

Lecture 3

傅子文

Outline of Lecture 3

- Binary representations & useful number systems
  - Character encoding
  - binary integers & hexadecimal
  - Floating point
  - IEEE 754

Digital domain

- Computers are fundamentally driven by logic and thus bits of data
  - Manipulation of bits can be done incredibly quickly

- Given **n** bits of information, there are 2^n possible combinations

- These 2^n representations can encode pretty much anything you want, letters, numbers, instructions….

Remember: ***8 bits=byte, 4 bits=nybble*** *&*
bytes *are the smallest addressible subunit of memory*

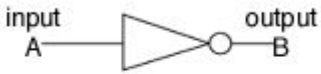
Character encoding

- Fundamentally based on bytes: so 256 possibilities are available at the simplest level
  - Mapping table then encodes each symbol to the binary value

- ASCII – "American Standard Code for Information Interchange"
  - First published 1967, it is still a universal standard
  - 7bit encoding though – early computers used the first bit to check for transmission errors
  - First 32 characters are *control characters*, were used primarily for printer and carriage control on terminals (can still use them if your UNIX keyboard mapping goes awry – try control-H)
  - Remaining characters are alpha numeric, plus a handful of symbols

Recap: Logic operations

NOT
(Inverter)



A	B
0	1
1	0

AND

A	B	C
0	0	0
1	0	0
0	1	0
1	1	1

OR

A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

Exclusive OR (EOR)

A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

The fundamental gates can be Combined to give others – e.g.

We won't use these operations very much, but every now and again the concepts may appear

NOR

A	B	C
0	0	1
1	0	0
0	1	0
1	1	0

NAND

A	B	C
0	0	1
1	0	1
0	1	1
1	1	0

Recap: Base β arithmetic

- Base 10 numbers: 0,1,2,3,4,5,6,7,8,9
  - $3107 = 3\times10^3 + 1\times10^2 + 0\times10^1 + 7\times10^0$
- Base 2 numbers: 0,1
  - 3107 = 1 2 4 8 16 32 64 128 256 512 1024 2048
  - $=1\times2^{11} + 1\times2^{10} + 0\times2^9 + 0\times2^8 + 0\times2^7 + 0\times2^6 + 1\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
  - =110000100011
- Addition, multiplication etc, all proceed same way
- Base & radix are used interchangably

We can express any number in a base system as

$$a = \sum_i a_i \beta^i \qquad (0 \le a_i \le \beta\text{-}1)$$

Example: Binary arithmetic

- Addition: 0+0=0, 0+1=1, 1+0=1, 1+1=10
  - Apply “carry” rules of decimal addition exactly the same
- Subtraction: 0-0=0,0-1=1(borrow),1-0=1,1-1=0
  - We can avoid writing negative numbers using two’s complement notation (more later)
- Multiplication: $0\times0=0, 0\times1=0, 1\times0=0, 1\times1=1$
  - Apply same rules as decimal multiplication for carries
- Division: same rules as decimal

Binary arithmetic works under the same remainder & carry rules as decimal

Base notation

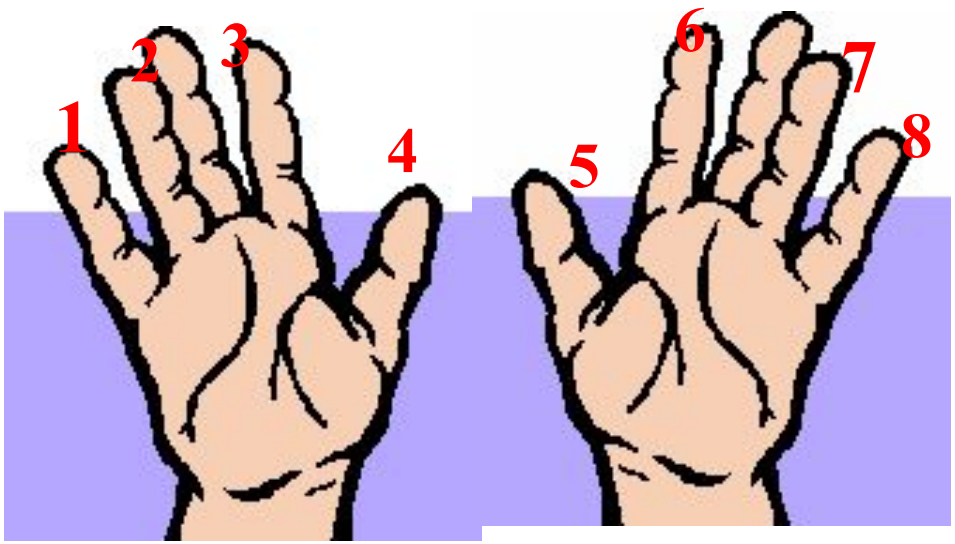
- What does 10 mean?
  - 10 in binary = 2 decimal
  - 10 in octal (base 8) = 8 decimal
  - 10 in decimal = 10 decimal
- Need some method of differentiating between these possibilities
- To avoid confusion, where necessary we write

$$Number_{base}$$

- 10_{10}=10 in decimal, 10_2=10 in binary

A note on octal

- Decimal seems natural because we count on the basis of fingers
- What if you counted on the basis of the spaces between fingers?
- That would give an octal system
- Yuki language in California & Pamean languages in Mexico both use **octal**



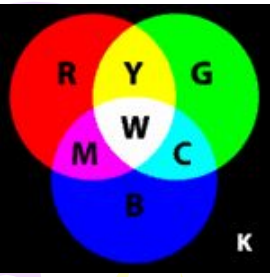
You can also convert binary to octal easily: group into three binary digits from right: e.g. 1001010 = 1 001 010 = 112 in octal

Hexadecimal representation ("hex")

- Remembering long binary number is exceedingly hard
  - Sometimes necessary when debugging memory addresses
  - Decimal is easier but still difficult
  - Brain has problems with more than 6-9 digits
  - Reduce number of digits – easier to remember
- Hexadecimal:
  - 0-9, plus A, B, C, D, E, F: base 16
  - $111_{16}=1_{10}+16_{10}+256_{10}=273_{10}$
  - $FFF_{16}=15_{10}+240_{10}+3840_{10}=4095_{10}$
- Alternative notation include placing an h behind the hex number, e.g. A03h or beginning with 0x, e.g. 0xA4B

More on hex

- To relate to binary:
  - Recall binary → octal group into threes
  - Binary to hex → group into fours
- Hex particular useful for colour space
  - Three numbers on 0-255=six hex digits
  - RGB colours often represented this way
    - e.g. #FFFFFF is red 255, green 255, blue 255 = white



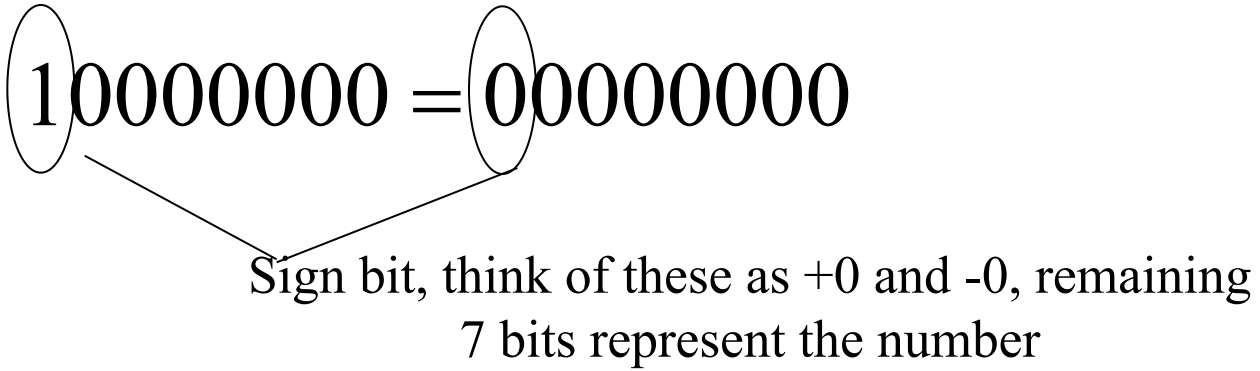
Decimal	Hex	Binary
1	0	0000
2	1	0001
3	2	0010
4	3	0011
5	4	0100
6	5	0101
7	6	0110
8	7	0111
9	8	1000
10	9	1001
11	A	1010
12	B	1011
13	C	1100
14	D	1101
15	E	1110
16	F	1111

Hex riddle

From www.thinkgeek.com

Signed versus unsigned numbers

- What about representing negative numbers digitally?
  - We could use a sign bit – but this causes a problem…



- We could also apply logic operations to the number
  - One's complement – take a number and apply the NOT logic e.g. for 42 00101010 gives 11010101 for -42
  - Another problem though 00000000=11111111

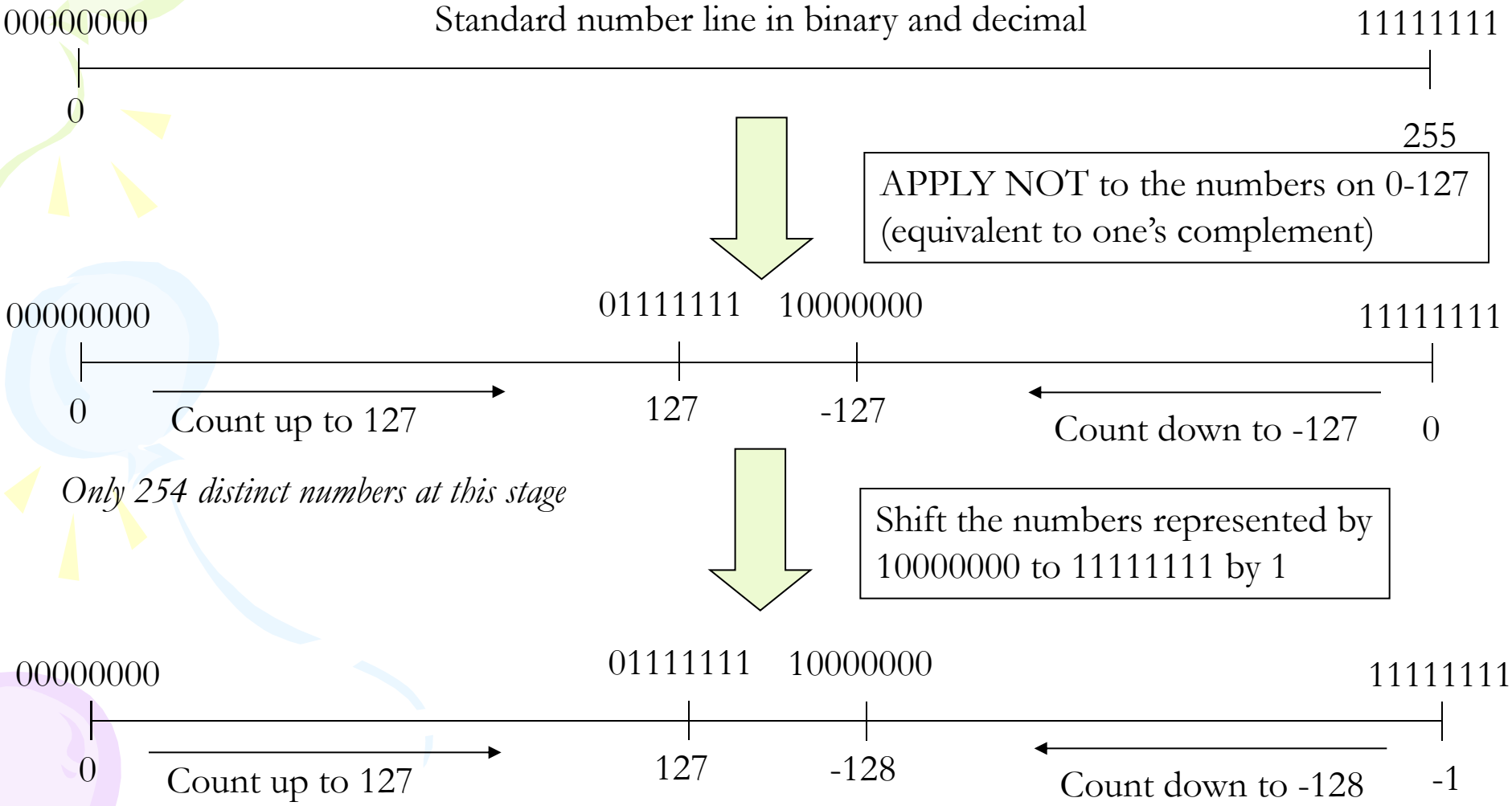
Two's complement

- Need a system such that 0 is not doubly defined
- Suppose we set the negative value by
  - NOT all digits and then add 1
  - Faster way: search from right to find first 1, invert all bits to the left

01011100
↓↓↓↓↓
10100100

Binary	Unsigned value	Two's complement value
00000000	0	0
00000001	1	1
11111111	255	-1
00000010	2	2
11111110	254	-2

Why does it work?



Have one more number on the negative axis than on the positive

Addition of two's complement integers

- Almost works! Look -

01111111		127
+10000010	⟺	+-126
----------		------
100000001		1
----------		------

Extra digit is called an ***overflow***
We have exceed the space of numbers described by 8bits

If we discard the extra digit then addition is just fine

Beauty of two's complement

- **Subtraction** is now easy – convert the number to be subtracted into it's two's complement and add
  - No need for separate circuitry to do the subtraction operation
- **Multiplication** follows the same rules as normal binary multiplication
  - Try it and see: what is 00000100×11111100?
- **Division** requires repeated versions of the subtract operation but follows the same rules otherwise (but clearly takes longer the multiplication)
  - Division operations always take longer than multiplications whether it is a integer or floating point operation

Rules all apply in the same way for longer strings of bits, ie 32bits

Excess-N representations

- A value is represented by an unsigned number N larger
  - Frequently $N=2^m-1$
- These are called *biased representations* where N is the *bias*
- Avoids double counting problems
- This representation is used in floating point numbers (see later)
  - Easier to compare values in this format

decimal	binary	Excess-5
0	0000	-5
1	0001	-4
2	0010	-3
3	0011	-2
15	1111	10

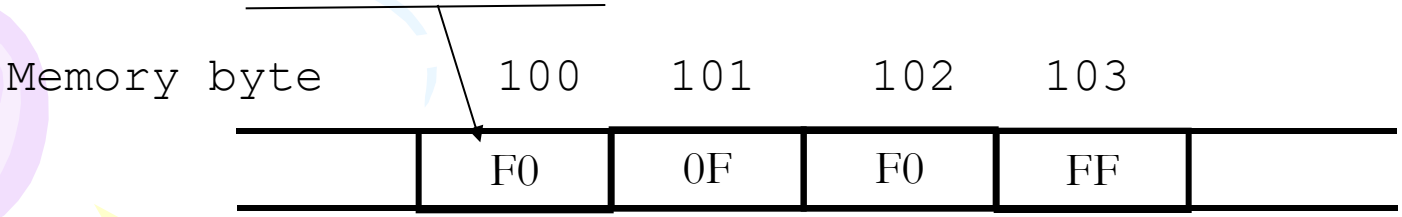
Fixed versus arbitrary precision

- Thus far we have assumed a fixed number of bits to represent a number
  - We call this fixed precision computation
    - It does not mean that the numbers are regularly spaced, just that a *fixed amount of information* is used
      - It doesn't necessarily mean integers either
    - Computers support fixed precision in hardware
- Arbitrary precision allows you to define numbers of any size
  - Very useful for cryptography, number theory
  - Still comparatively uncommon, and only supported in software – so still comparatively slow

Endianness

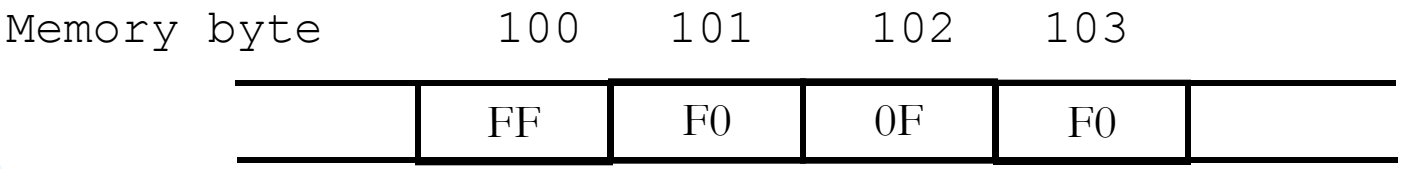
- How do you store binary representations of memory words?
  - Remember bytes are the fundamental subunit of memory
  - 32bits = 4 bytes e.g. 11110000000011111111100001111111=F00FF0FF
  - Bytes are F0 0F F0 FF

- One choice for byte ordering:
  - Traditional left-to-right writing sequence seems most natural
    - F00FF0FF would be stored exactly in this order
    - **"Big Endian"** ("big end in")
    - *Most significant byte* is stored at the lowest memory value



Little Endian

- Computer designs do not have a natural left or right so we could equally well store as:



- Ordering is reversed – *least significant byte* is first
- Bit endianness is less of an issue since we can't address them (although we can manipulate bits via either math or logic instructions)
- There appear to be no advantages to one endian style over another
  - Different machines use different conventions solely on the basis of design choices

Even more strange ordering standards have been applied but we'll ignore them

Portability issues

- You can write files to disk that store data precisely as in the computer memory (binary files)
- Such files are ***not portable***
  - Reading on one computer will produce a different behaviour on one machine compared to another
  - Why? Because both computers will read the file in the same way – the first pieces of data will go in the lowest memory address and so on
    - When operating on these words the computer will interpret the byte ordering differently!
- Not a problem for files that are stored using a standard encoding format such as ASCII (more later on ASCII)
  - Very easily read, but there can be a lot of wasted space in these files

Endianness of popular CPUs

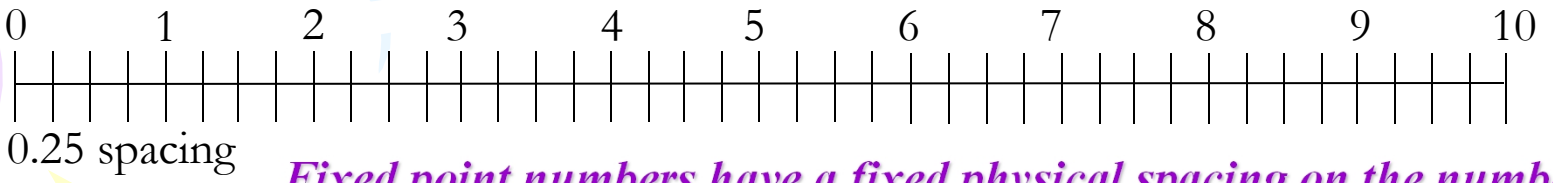
CPU	Endianness
All Intel & Intel compatible (including AMD)	Little (no surprise this is sometimes called Intel ordering)
Sun Microsystems (SPARC)	Big
IBM (Power series)	Big
SGI (MIPS)	Big

So in all likelyhood you will be using a little endian machine – but be careful when transferring data from your Intel machine to a Sun Sparc workstation (for example)

Representing fractions

- In decimal we use the decimal point: 123.456_{10}
  - $123.456_{10} = 1\times10^2 + 2\times10^1 + 3\times10^0 + 4\times10^{-1} + 5\times10^{-2} + 6\times10^{-3}$
- In other base notations we call the separator between integer and fraction the "radix point"
  - Consider $10.01_2 = 1\times2^1 + 0\times2^0 + 0\times2^{-1} + 1\times2^{-2}$
    $= 1\times2^1 + 0\times2^0 + 0\times(1/2) + 1\times(1/4)$
    $= 2.25_{10}$
- Same approach applies to any other base
- Quick Q: if I use 8 bits to encode a fraction what is the smallest fraction I can represent given the first digit represents 2^{-1}?
- By choosing a fixed number of digits before and after the radix point you set a precise and even distribution of numbers
  - *Fixed point arithmetic:* $d_1d_2\ldots d_n.f_1f_2\ldots f_m$
  - Can also apply two's complement to the integer part to represent negative numbers

Example: A piece of the number line from a represenation $d_1d_2d_3d_4.f_1f_2$ in base 2 is given below



Fixed point numbers have a fixed physical spacing on the number line

Floating point representations

A system of floating point numbers can be defined with parameters $\beta, t \in \mathrm{N}$ and $e_{\min}, e_{\max} \in \mathrm{Z}$

$$a = m \times \frac{\beta^e}{\beta^t} = m \times \beta^{e-t} = \pm \beta^e \times 0.d_1 d_2 \ldots d_t$$

$(0 \le d_i \le \beta\text{-}1$ & $\boldsymbol{d_1 > 0})$

- β defines the ***base***
  - 2 is natural for a computer system
- *m*, the *integer* ***mantissa****, must obey $|m| < \beta^t$, so that the numbers are in a given range (they will have *t* digits)
  - *t* is called the *significance*, and β^{-t} is essentially a normalization
- *e*, the ***exponent***, must be in the range $e_{\min} \le e \le e_{\max}$
- The number *a* is thus represented by the mantissa-exponent pair (*m*,*e*) (for a specified base)

***Since 2005 there has been a move to use "significand" in place of mantissa.**

Aside: FLOPS

- The importance of floating point operations (ops) leads to the rate of floating point ops being a useful metric
- FLOP=floating point operation
- FLOPS=floating point operations per second
- A PC can typically achieve several giga-FLOPS (GFLOPS)
  - The most powerful parallel computers currently achieve hundreds of tera-FLOPS
  - Peta-FLOPS computers was built in 2008

世界最强超级计算机

Remember though, FLOPS are dependent upon the calculation being done

"Humour": Intel's Pentium bug

- In 1994 Thomas Nicely discovered a bug in the Pentium floating point divide unit
- Results were off by 61 parts per million
- Intel initially refused to provide replacement parts saying the results "were good enough" for most "non-technical" people – but later gave in under pressure

Q: How many Pentium designers does it take to screw in a light bulb?

A: 1.99904274017, but that's close enough for non-technical people.

Properties of floating point systems

- ***Machine epsilon***: the smallest number that when added to 1 gives the next representable number

$$\varepsilon_{mach} = \beta \times 0.1...1 - \beta \times 0.1...0 = \beta \times 0.0..1 = \beta^{1-t}$$

- ***Unit roundoff*** is often referred to and is defined by

$$u = \frac{\varepsilon_{mach}}{2} = \frac{1}{2}\beta^{1-t}$$

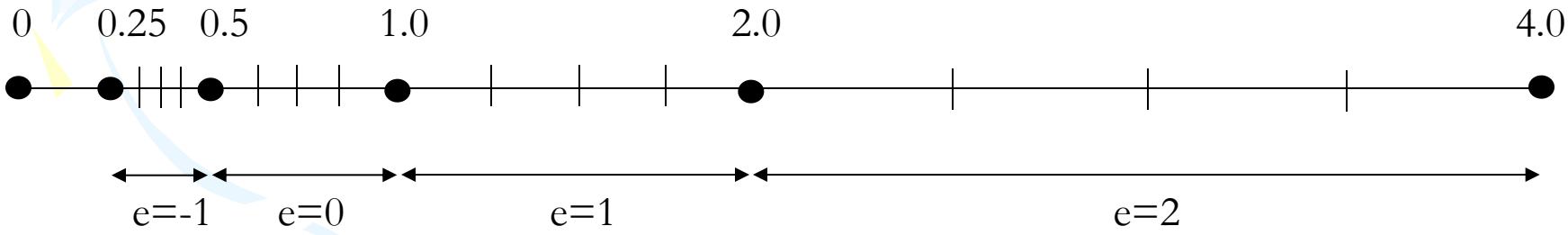
- The ***range*** of numbers that are approximated by the floating point system is given by

$$\beta^{e_{\min}-1} \le |a| \le \beta^{e_{\max}}(1-\beta^{-t})$$

Non-uniform distribution

- Floating point numbers are *not* distributed evenly on the number line

- Consider a system with β =2, t=3 and e_{min},e_{max} limits of -1,2 (non negative in this case)
  - Numbers are given by $2^e \times 0.d_1d_2d_3$ and smallest is $2^{-1} \times 0.1_2$



Number line divides up into regions of size $[\beta^{e-1}, \beta^e)$

Floating Point Arithmetic

- How do we add floating point numbers?
  - For example what about 0.110×2^2 and 0.100×2^1
  - Exponents need to be made the same first, so multiply 0.100×2^1 by 2^1 and shift radix point to give 0.010×2^2
  - Now we can add the mantissa's to give 1.000×2^2
  - Problem! We've got more digits than allowed now, so we must truncate to *t* digits
  - 1.000×2^2 -> 0.100×2^3

Scaling step is the key cause of errors in floating point

Truncation is unavoidable in floating point arithmetic

IEEE 754 floating point standard

- This is a representation that is available on all modern fp-capable computers (defined in 1982)
  - The standard ensures consistent behaviour across platforms
    - This is *not trivial*, prior to the standard different computers had different behaviours
- The standard defines
  - Number formats, conversion between formats
  - Special values
  - Rounding modes
  - Exceptions and how they should be handled

IEEE Floating Point Precisions

Single	Double	Quadruple
32 bits	64 bits	128 bits
1 sign bit 23 bit mantissa 8 exponent bits	1 sign bit 52 bit mantissa 11 exponent bits	1 sign bit 112 bit mantissa 15 exponent bits
Unit roundoff $u \approx 5.96 \times 10^{-8}$	Unit roundoff $u \approx 1.11 \times 10^{-16}$	Unit roundoff $u \approx 10^{-36}$
range is $\approx 10^{\pm 38}$	range is $\approx 10^{\pm 308}$	range is $\approx 10^{\pm 4932}$
$\beta=2$, t=24, $e_{min}=-126$, $e_{max}=127$	$\beta=2$, t=53, $e_{min}=-1022$, $e_{max}=1023$	$\beta=2$, t=113, $e_{min}=-16382$, $e_{max}=16383$

- $x = (-1)^s \times \text{Mantissa} \times 2^{exp}$

$\text{mantissa} = m_1 2^{-1} + m_2 2^{-2} + \ldots$

$\text{bias} = 0111\ 1111_2 = 127_{10}$

	s	exp+bias	Mantissa	
float_{64}	0	00001111111	10000000000000000000000000000000000000000000000000000	$= \frac{1}{2}$

Range

$\pm 2.48 \times 10^{-324} \leq \text{float}_{64} < \pm 1.8 \times 10^{308}$

Precision

float_{64}: 16 decimals

1 sign bit
52 bit mantissa
11 exponent bits

$$2^{-52} \approx 2.22 \times 10^{-16}$$

Why wouldn't you use the highest precision all the time?

Two primary reasons:

- Storage
  - If you have a problem that requires a lot of memory you may be forced to use a lower precision to fit it in to the available memory
- Speed
  - Double precision calculations tend to be a bit slower than single (although the speed difference is getting smaller). Quad precision is usually very slow

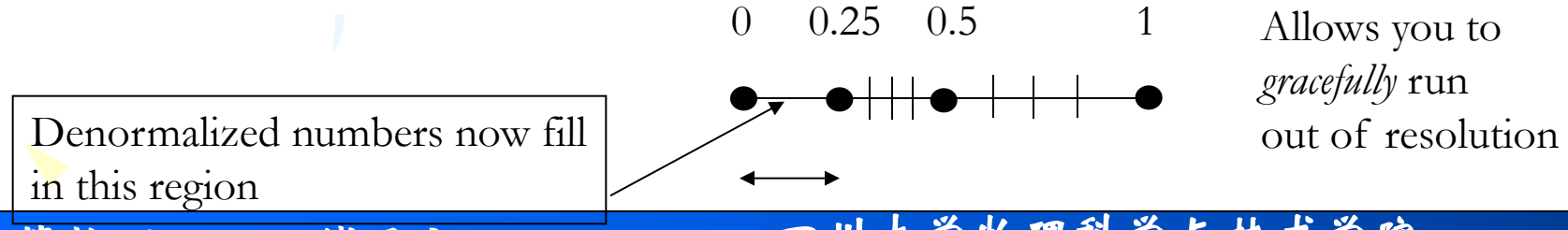
Variable names for IEEE formats in various programming languages

IEEE Precision	C,C++	FORTRAN
single	float	real, real*4
double	double	double precision or real*8
Quad/extended double precision*	long double	real*16

*whether you get full resolution depends upon the platform

Denormalized numbers

- Recall the standard, *normalized* representation of a floating point number $a = \pm\beta^e \times 0.d_1 d_2 ... d_t$ where d_1 is 1
- When the exponent reaches its minimum value (-126) we reach a limit on the size of representable numbers
- *If we allow d_1 to be zero we can allow smaller numbers!*
  - But these numbers are no longer correctly normalized
- Allows system to fill in numbers down to 0, although mantissa loses resolution as you do so
- Recall earlier β=2, *t*=3, example



Summary

- Hex is a simple way of representing 32 bit words
- Signed integers can be represented using two's complement
- Byte ordering (endianness) is not consistent across all computers
- Floating point representations use an exponent and mantissa
  - Follow the IEEE 754 standard

Exercise 1

Symbol	Action
?	match any single character
*	match any size string
[abc]	match any enclosed character
[a-f]	match any character in range
[!abc]	match all but enclosed characters
~	current user home directory
~user	home directory of a user

Symbol	Redirection
>	redirect to standard output
>&	redirect to standard error
>>	append to standard output
|	pipe standard output to another command
|&	pipe standard error to another command
<	input redirection
<<String	read from standard input until "String" is encountered as the only thing on the line.

Symbol	Action
;	command separator
&	run command in background
&&	run next command upon success
||	run next command if unsuccessful
`command`	execute command first & substitute result
\	escape the following character

Shell Scripts

- A shell scripts is a special text file containing a list of shell commands. By executing the script, the shell executes all shell commands line by line.

  - Use an editor like *emacs* or *nedit* to write a shell script
    - The first line of the shell script should be "*#!/bin/sh*"
  - After writing shell script set the execute permission for your script
    - syntax: *chmod +x your-script-name.sh*
  - Execute your script
    - syntax: *./your-script-name.sh* or *sh your-script-name.sh*

File Edit Search Preferences Shell Macro Windows Help

```
#!/bin/sh

whoami
echo "Hello `whoami`"
echo "The date and time is "
date
echo "You are currently in the `pwd` directory"
echo "Good Bye `whoami`. It was nice chatting with you!"
```

```
$ sh hello_me.sh
Kevin
Hello Kevin
The date and time is
Thu Mar 10 10:00:03     2016
You are currently in the /home/Kevin directory
Good Bye Kevin. It was nice chatting with you!
```

Homwork 2： 03/13/2019

Problem 1: Use Gregory - leibniz infinite series

$$\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots+(-1)^{n-1}\frac{1}{2n-1}$$

to calculate π accurately 10 decimals, (n≈500,000).

In practice, this is a Simple Sum,
we can also calculate is in down order

$$\frac{\pi}{4}=(-1)^{n-1}\frac{1}{2n-1}+(-1)^{n-2}\frac{1}{2n-3}\ldots-\frac{1}{7}+\frac{1}{5}-\frac{1}{3}+1$$

In this exercise, please calculate π in botth method, and discuss which one will give more precise result, and explain why?

(1636–1716)

(1638–1675)

It was discovered by the Indian mathematician Madhava of Sangamagrama (1350- 1410)

Problem 2: Use Machin's formula

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

to compute πto 100 decimal place.

[Hints:

1: $\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots + (-1)^{n-1}\frac{x^{2n-1}}{2n-1}$

2: Quadmath.h]

Machin enjoyed a high mathematical reputation. His ingenious quadrature of the circle was investigated by Hutton, and in 1706 he computed the value of π by Halley's method to one hundred decimals places.