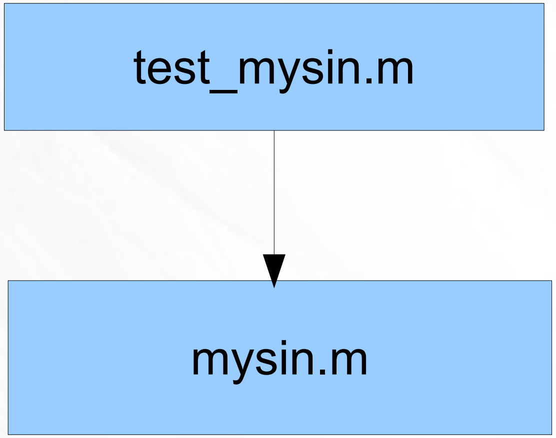
# Every program needs a test

```
function y = mysin(x, tol)
  % Computes sin by folding input into domain 0 <= x <= 2*pi
  % Then computes value using polynomial approximation

  % Initialize some constants.  Do this once instead of doing
  % it multiple times in the program in order to improve performance.
  piover2 = pi/2;
  pitimes2 = 2*pi;
  s = mod(x, pitimes2);

  % Do folding
  if (s < piover2)
    y = P(s, tol);
    return
  elseif (s < pi)
    y = P(pi-s, tol);
    return
  elseif (s < 3*piover2)
    y = -P(s-pi, tol);
    return
  elseif (s <  pitimes2)
    y = -P(pitimes2-s, tol);
    return
  else
    error('We failed!  x = %15.12e, s = %15.12e\n', x, s)
    y = nan;
    return
  end

end
```



# Example test program

```
function test_mysin()
  % This runs the function mysin for inputs over a range, and
  % checks its return against that from MATLAB.   If the difference
  % is larger than 1 ULP, then it errors out.

  for x = 0:2:100
    tol = 1*eps(x);
    y_comp = mysin(x, tol);
    y_true = sin(x);
    diff = abs(y_comp - y_true);
    fprintf('x = %20.18e, y_comp = %20.18e, y_true = %20.18e, diff = %20.18e\n', x,
y_comp, y_true, diff)
    if (diff > tol)
      error('Error is too large!!!\N')
    end
  end

  % If we get here, it's because all comparisons passed.
  fprintf('---  Test passed!  Success!  ----\n')

end
```

Get return from my program

Get “true”, analytic result

Check result here

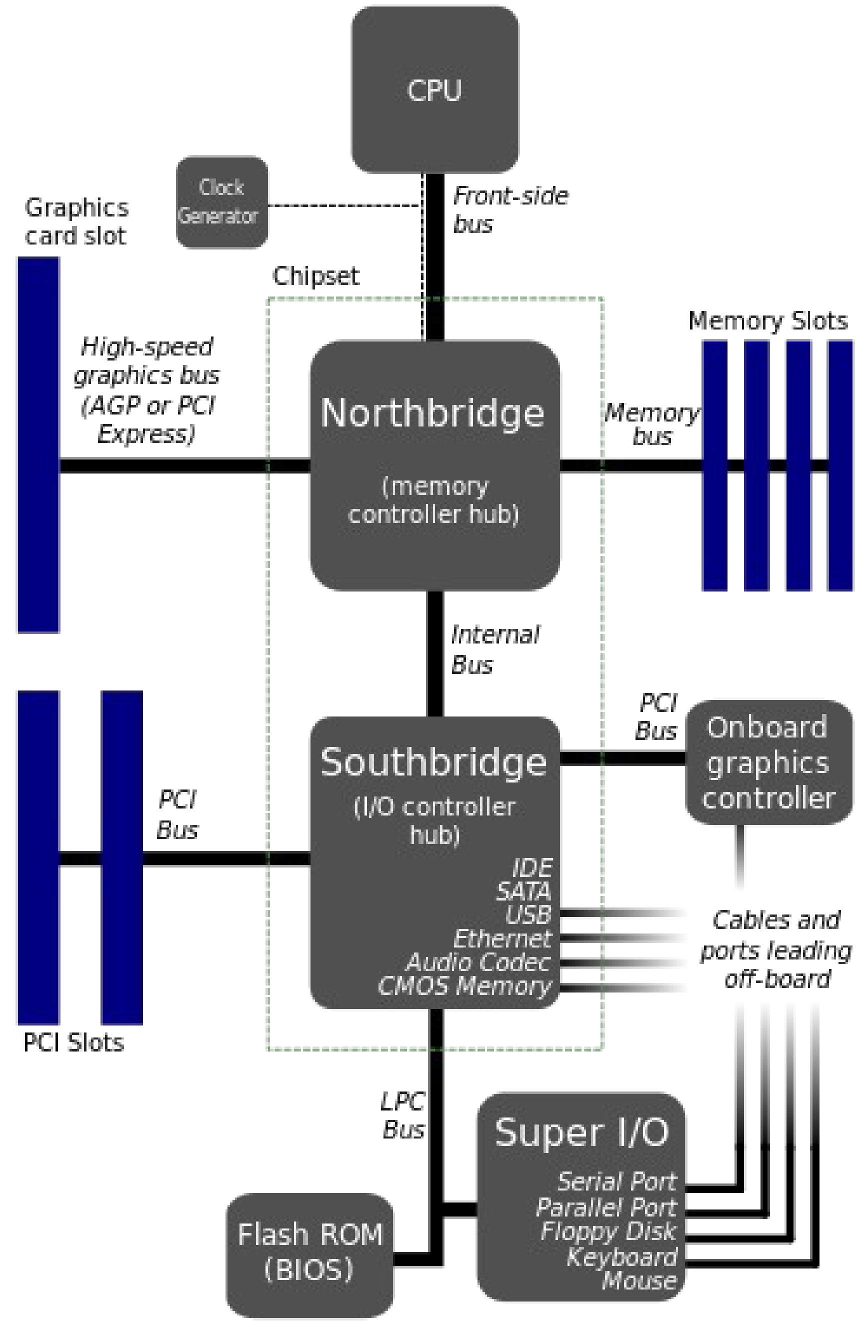
# When I run it.....

```
>> test_mysin
x = 0.00000000000000000e+00, y_comp = 0.00000000000000000e+00,
y_true = 0.00000000000000000e+00, diff = 0.00000000000000000e+00
x = 2.00000000000000000e+00, y_comp = 9.09297426825681598e-01,
y_true = 9.09297426825681709e-01, diff = 1.11022302462515654e-16
x = 4.00000000000000000e+00, y_comp = -7.56802495307928313e-01,
y_true = -7.56802495307928202e-01, diff = 1.11022302462515654e-16

Etc....

x = 1.00000000000000000e+02, y_comp = -5.06365641109755348e-01,
y_true = -5.06365641109758790e-01, diff = 3.44169137633798527e-15
---  Test passed!  Success!  ----
```

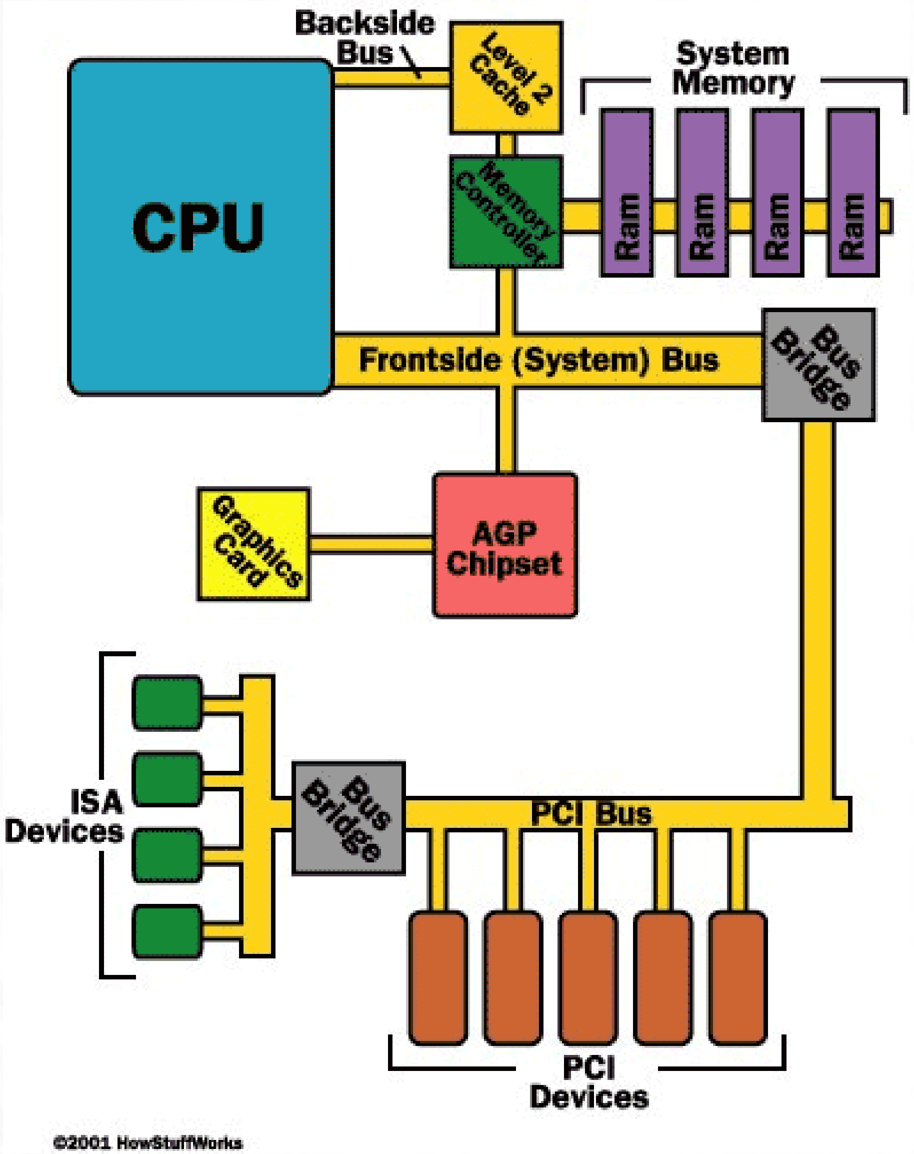
- I will run your test program when grading your HW
- Please make sure your test runs and passes.
- Please zip up fcn and test into one .zip package – one per HW problem
- Testing is an important real-world practice

# Next: Computer internals

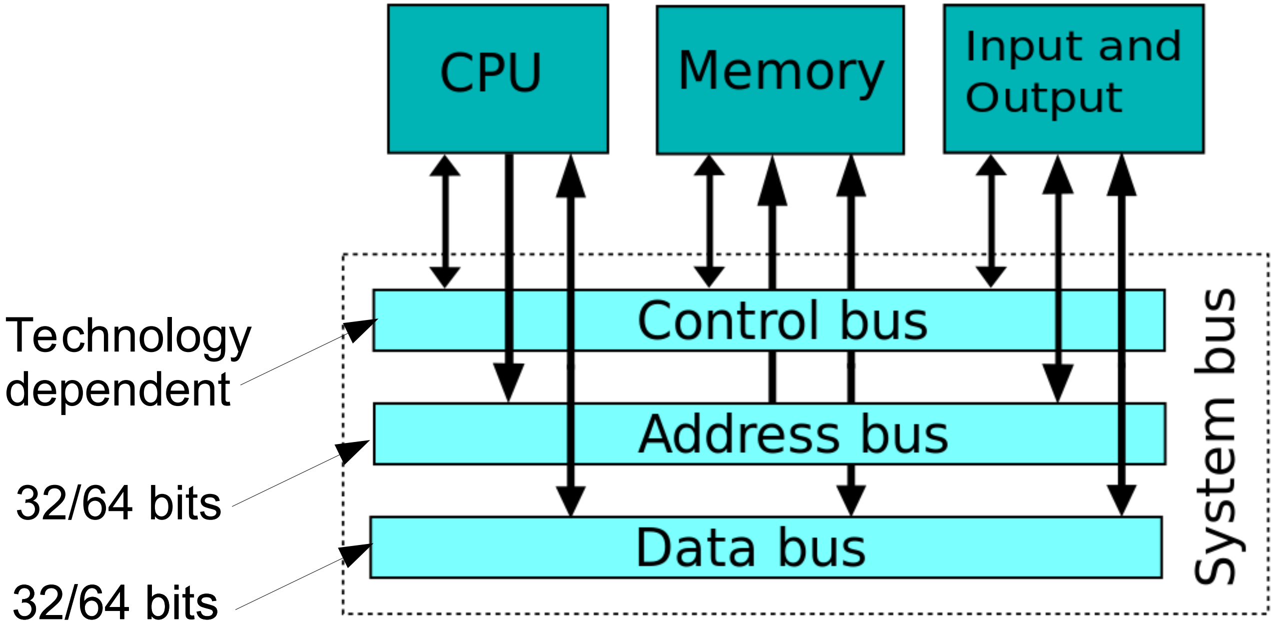
- CPU
- RAM accessed through controller
- Video memory separate from RAM.
- Note busses
- PCI slots



- CPU
- Cache next to CPU
- RAM accessed through controller
- Video memory separate from RAM.
- Note busses
- PCI slots

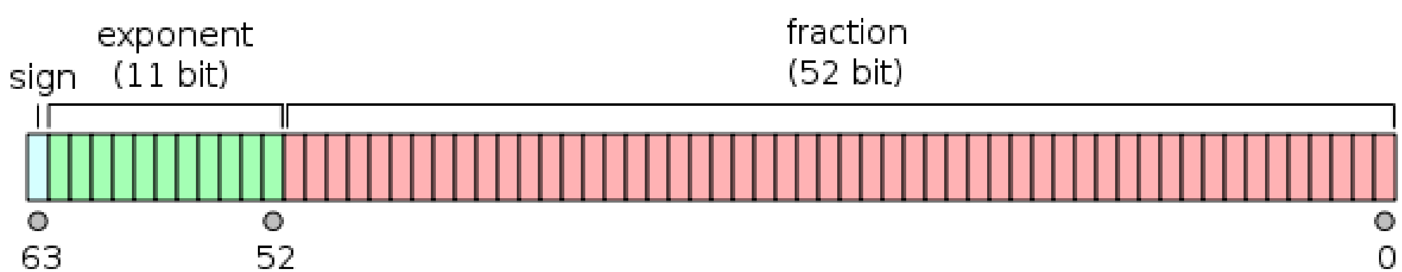


# System bus





- Bus is how data goes from memory and peripherals to CPU.
- When people talk about 32 or 64 bit computers, they are talking about the computer's word size, usually the same as the bus width.



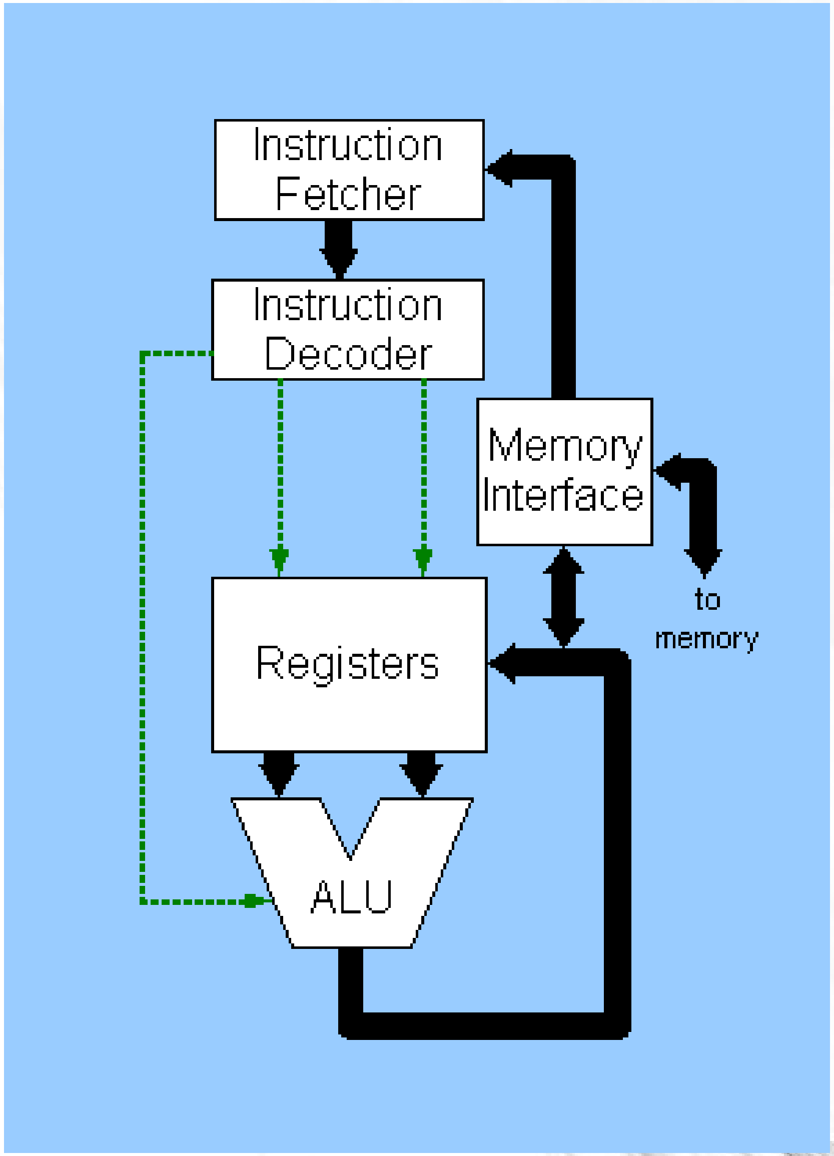
# Max memory size is dictated by bus width

- 32 bit address => 2^32 = 4GB of addressable RAM.
  - Equivalent to 23K x 23K matrix of doubles.
- 64 bit address => 2^64 = 1.8e19 bytes of addressable RAM.

# CPU

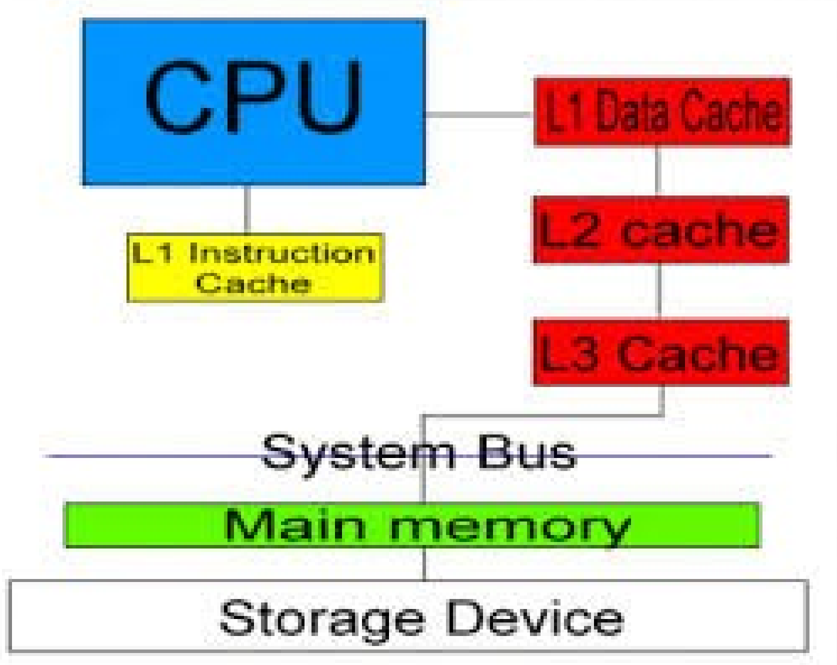
- How instructions are executed.
- Logic to fetch and execute instructions (primitive machine code).
- Registers – onboard memory locations.
- ALU – Arithmetic & logic unit.  Unary and binary operations on different types of data.
- On-chip cache (fast memory).

# Very simple view of CPU



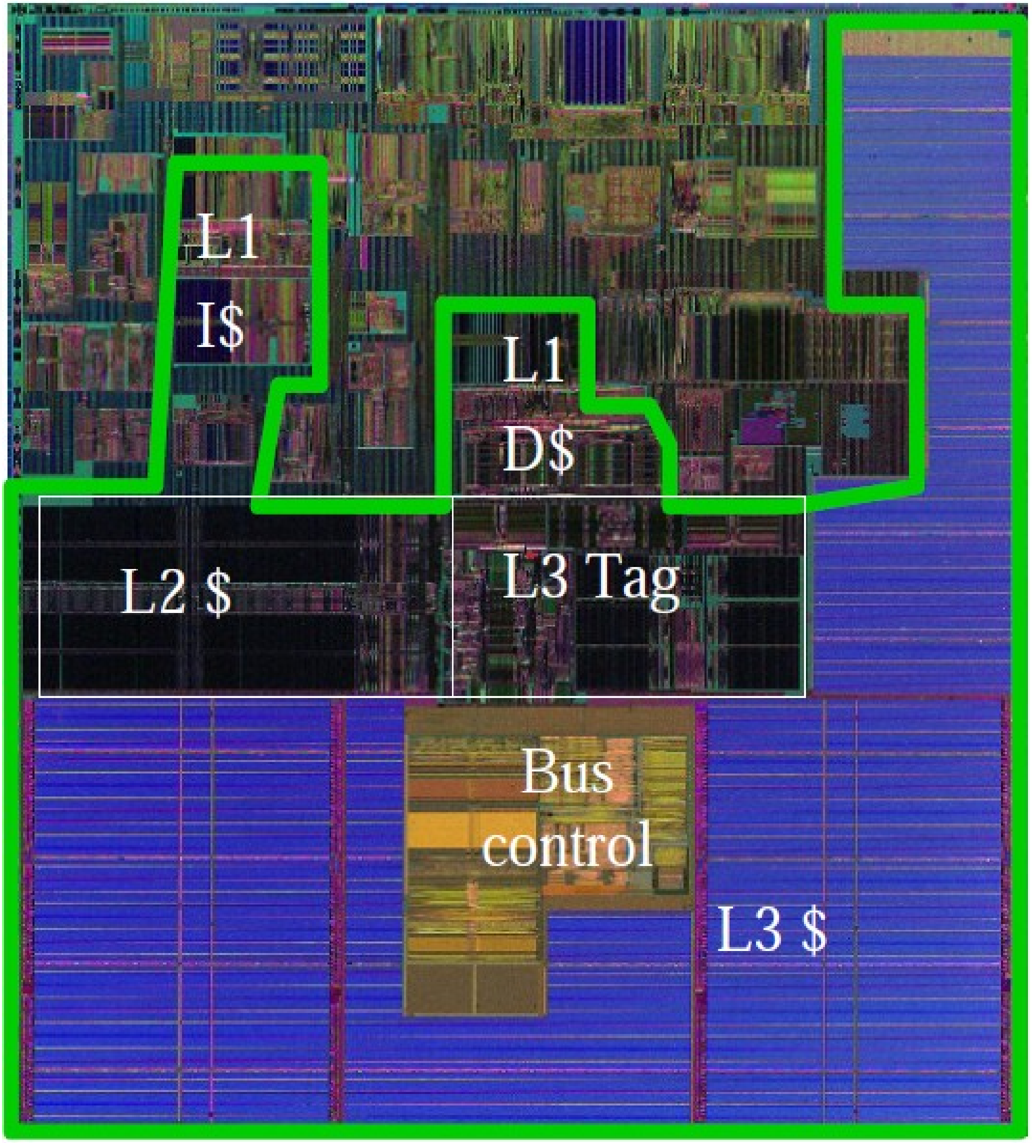

- Instruction fetch and decode
- ALU (actually, there are more than 1)
- Registers
- Memory interface

# Cache



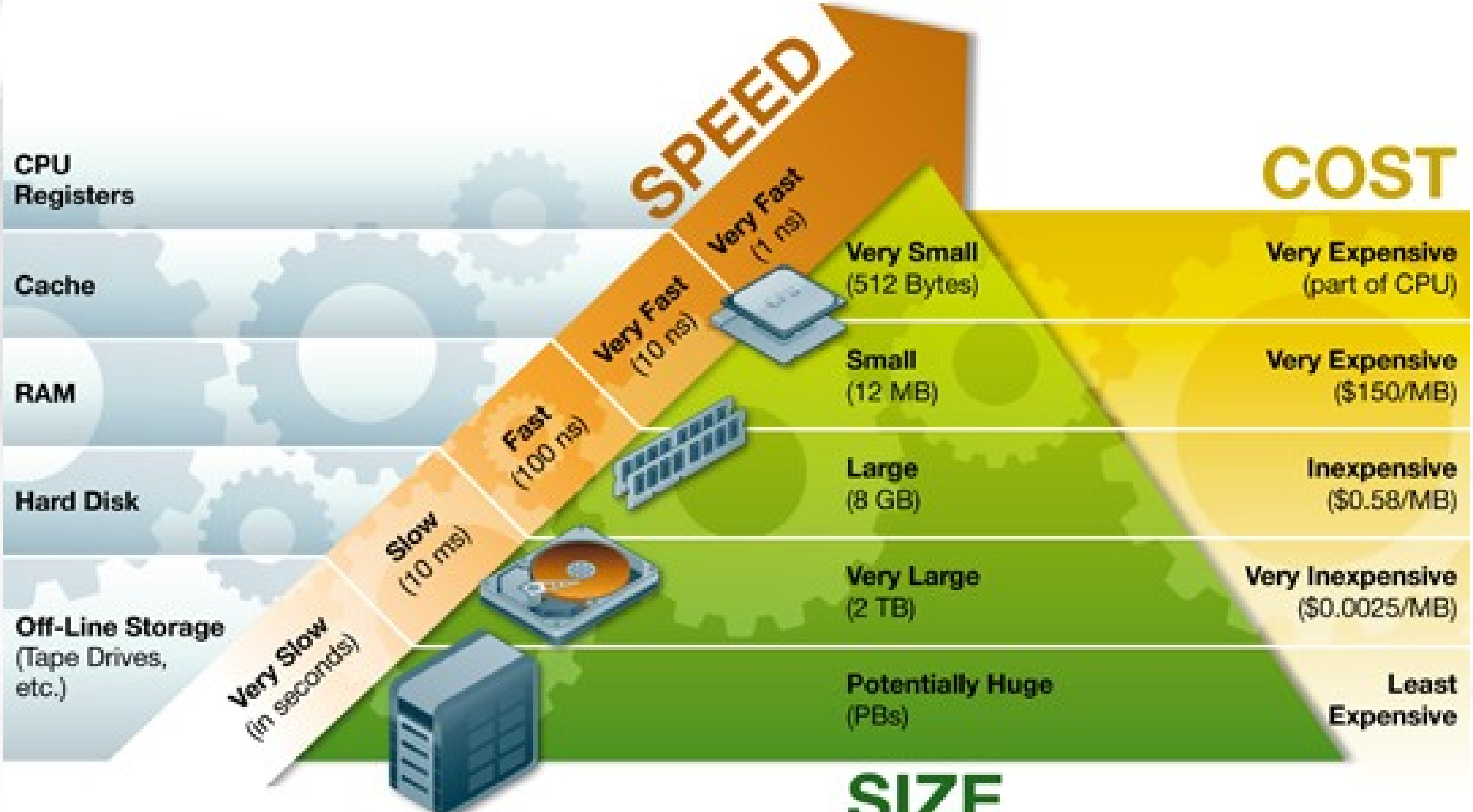

- High-speed memory on-board CPU
- Up to 3 “levels” in modern computers.
- Closer to CPU -> faster & smaller, away from CPU -> slower & larger.

# Itanium Caches




- Intel Itanium II
  - 2 L1: (16kB + 16 kB)
  - L2: 256kB
  - L3: 3072kB

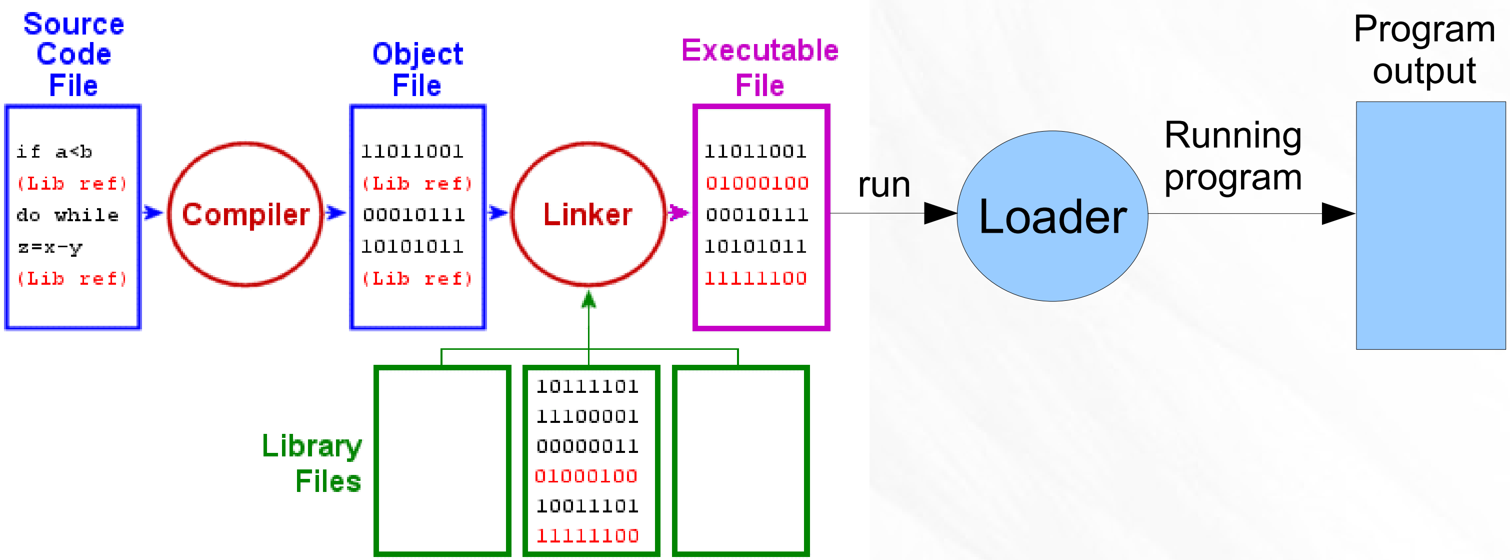
# Hierarchy of Storage



# Important pieces to remember

- CPU
- Cache
- Bus
- Memory (RAM)
- Peripherals – disk drives, mouse, keyboard, sound card, display screen, USB stuff, etc.

# New topic: Software Compilation of code – the old way



# Example C code

```
case(GET_BYTE_START):
  // Now wait here until we get a zero
  rcv_bit_ctr = 0x00;
  byte_ctr = 0x00;
  if (rcv_bit == 0x00) {
    rcv_state = GET_BYTE;
  } else {
    rcv_state = GET_BYTE_START;
  }
  break;

case(GET_BYTE):
  rcv_byte = (rcv_byte << 1) | rcv_bit;  // shift bit into rcv_byte
  if (rcv_bit_ctr == 8) {
    bytes[byte_ctr] = rcv_byte;
    byte_ctr++;
    if (byte_ctr > 6) {
      rcv_state = REFRAME;      // Error -- we need to reframe
    }
    rcv_byte = 0x00;            // We're done.  Clear out rcv_byte
    rcv_state = GET_PACKET_END;
  } else {
    rcv_state = GET_BYTE;     // Go back and get next bit.
  }
  break;
```

C program written for AVR microcontroller

# Assembly code

```
     // Sampled first part of 0 bit.  Set bit ready, then set timer to skip next
     // comparator interrupt.
     bit_ready = 0x01;
104:   81 e0          ldi     r24, 0x01       ; 1
106:   80 93 86 00    sts     0x0086, r24
     OCR0A = 60;
10a:   8c e3          ldi     r24, 0x3C       ; 60
10c:   86 bf          out     0x36, r24       ; 54
     TCNT0 = 0x00;                  // Set timer 0 count to 0
10e:   12 be          out     0x32, r1        ; 50
     TIMSK |= (1 << OCIE0A);        // Re-enable timer 0 A interrupts
110:   89 b7          in      r24, 0x39       ; 57
112:   81 60          ori     r24, 0x01       ; 1
114:   89 bf          out     0x39, r24       ; 57
     TCCR0B |= (1 << CS01);         // Turn on timer 0, use /8 prescalar
116:   83 b7          in      r24, 0x33       ; 51
118:   82 60          ori     r24, 0x02       ; 2
11a:   83 bf          out     0x33, r24       ; 51
 // turn the comparator interrupt back on again.
 ACSR  |=  (1<<ACI);         // clear Analog Comparator interrupt
 ACSR  |=  (1<<ACIE);        // Re-enable Analog Comparator interrupt
```

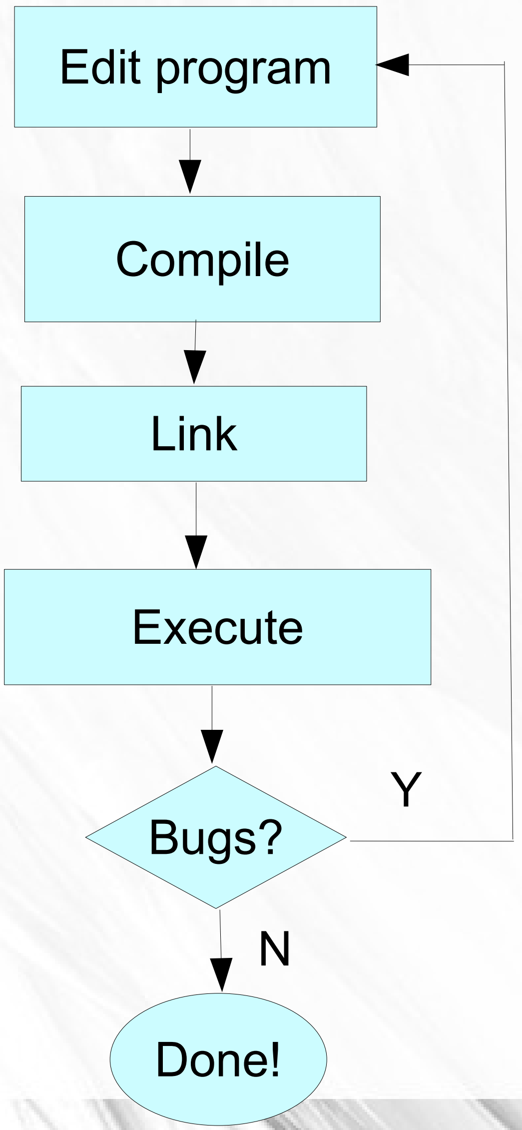
C program written for AVR microcontroller – assembler statements in .lst file.

# Machine code

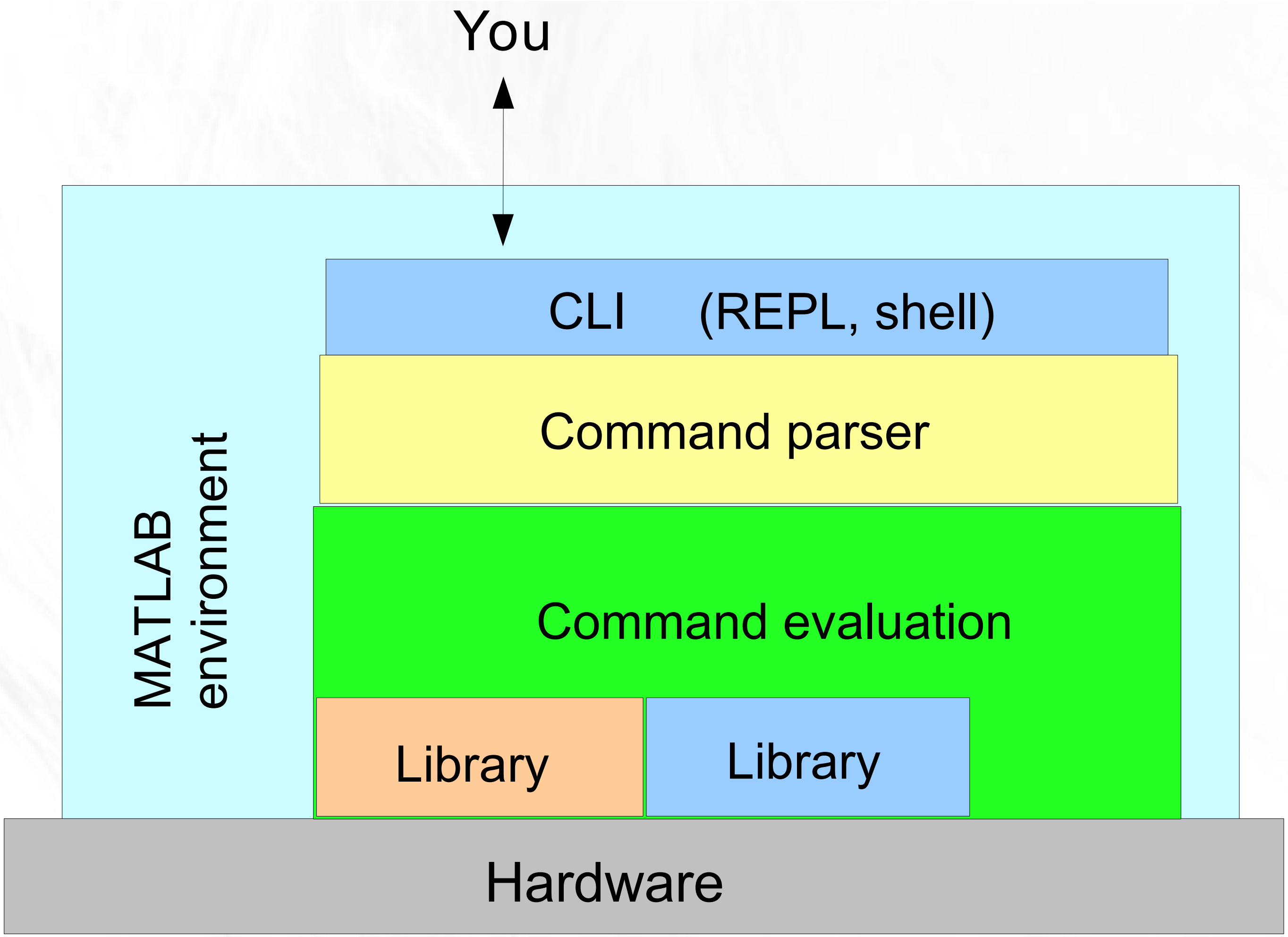
```
:1000000023C032C031C030C02FC02EC02DC02CC084
:100010002BC02AC02AC028C027C041C025C024C088
:1000200023C022C021C020C01FC07BC268C2BEC284
:10003000AFC2A1C27DC215C3D0C274C133C2EBC16D
:10004000E1C1D2C1BFC1F1C211241FBECFE5D1E0D1
:10005000DEBFCDBF10E0A0E6B0E001C01D92A73822
:10006000B107E1F74BD1E6C4CBCF1F920F920FB689
:100070000F9211248F9343981092840010928300062
:1000800012BE89E186BF89B7816089BF83B782606C
:1000900083BF8F910F900FBE0F901F9018951F92E6
:1000A0000F920FB60F9211248F939F9389B78E7F73
:1000B00089BF83B78D7F83BF98B18091830095FB03
:1000C000992790F9880F892B8093830080918400071
:1000D0008F5F80938400809184008130 69F18091EA
:1000E0008400823049F0449A439A9F918F910F90F7
:1000F0000FBE0F901F901895809183008370 8150E0
:10010000823098F081E0809386008CE386BF12BE37
:1001100089B7816089BF83B7826083BF9F918F91C8
:100120000F900FBE0F901F90189581E0809386006E
:1001300012BE449A439AD9CF87E3E8CF82E090E099
```

C program written for AVR microcontroller – Hex values in .hex programming file

# Edit-Compile-Link-Test Loop

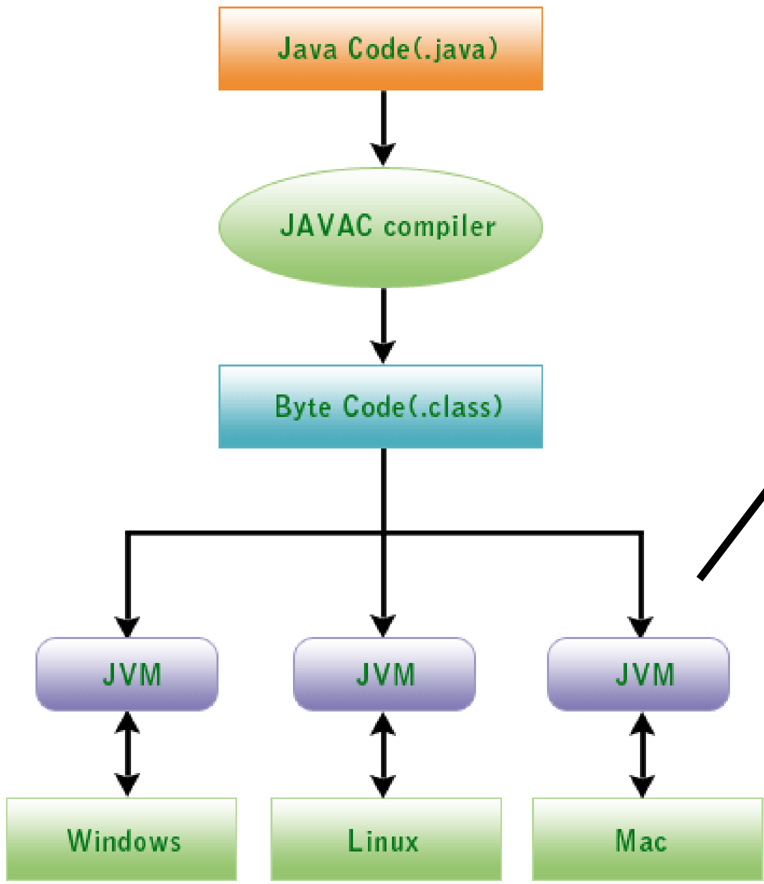


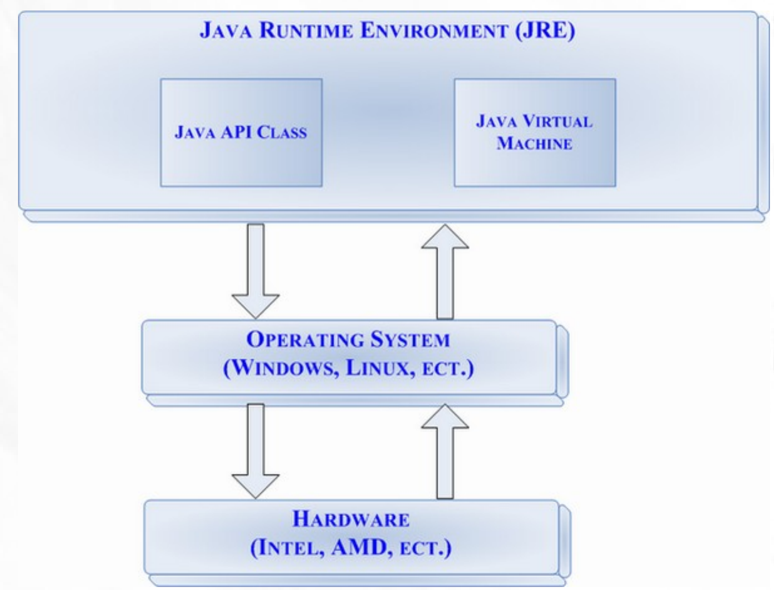
# Matlab:  Interpreted Language




- Advantage:  Interactive
- Disadvantage: Slow

# A modern approach: runtime engines




Your program runs on JVM

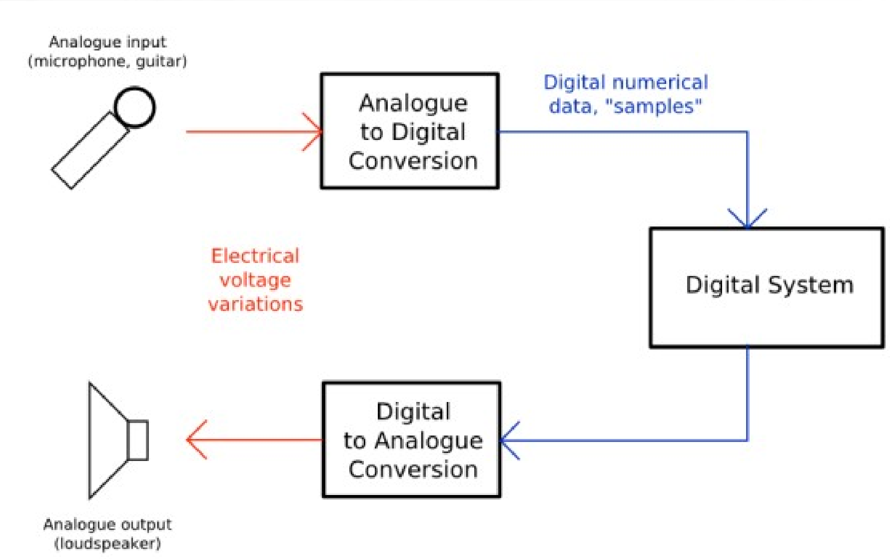
JVM runs on any OS/Hardware

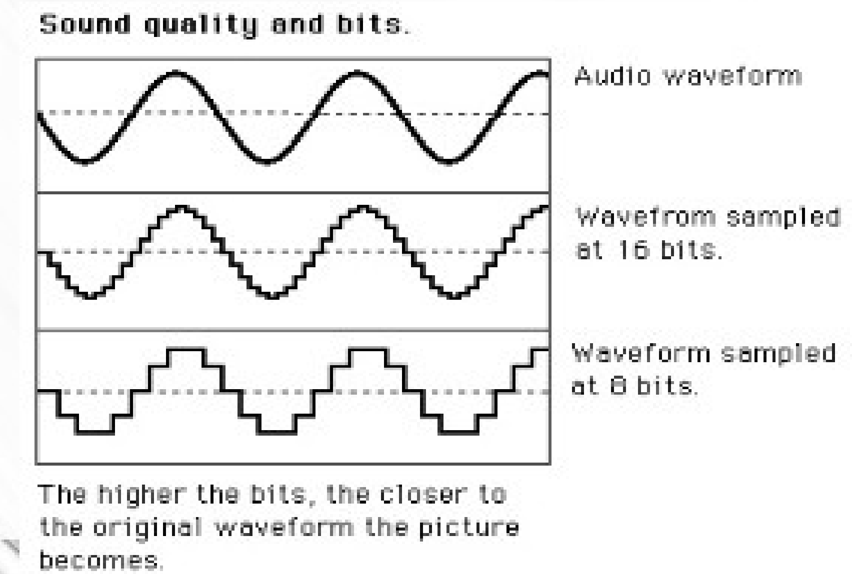
# What you need to remember

- Basic properties of numerical computing are determined by hardware considerations. (Example: Size of ints: 8, 16, 32, and 64 bit.)
- Knowing where your data is held is often important for performance (cache).
- Integer and floating point arithmetic occur in hardware – in the CPU's ALU.

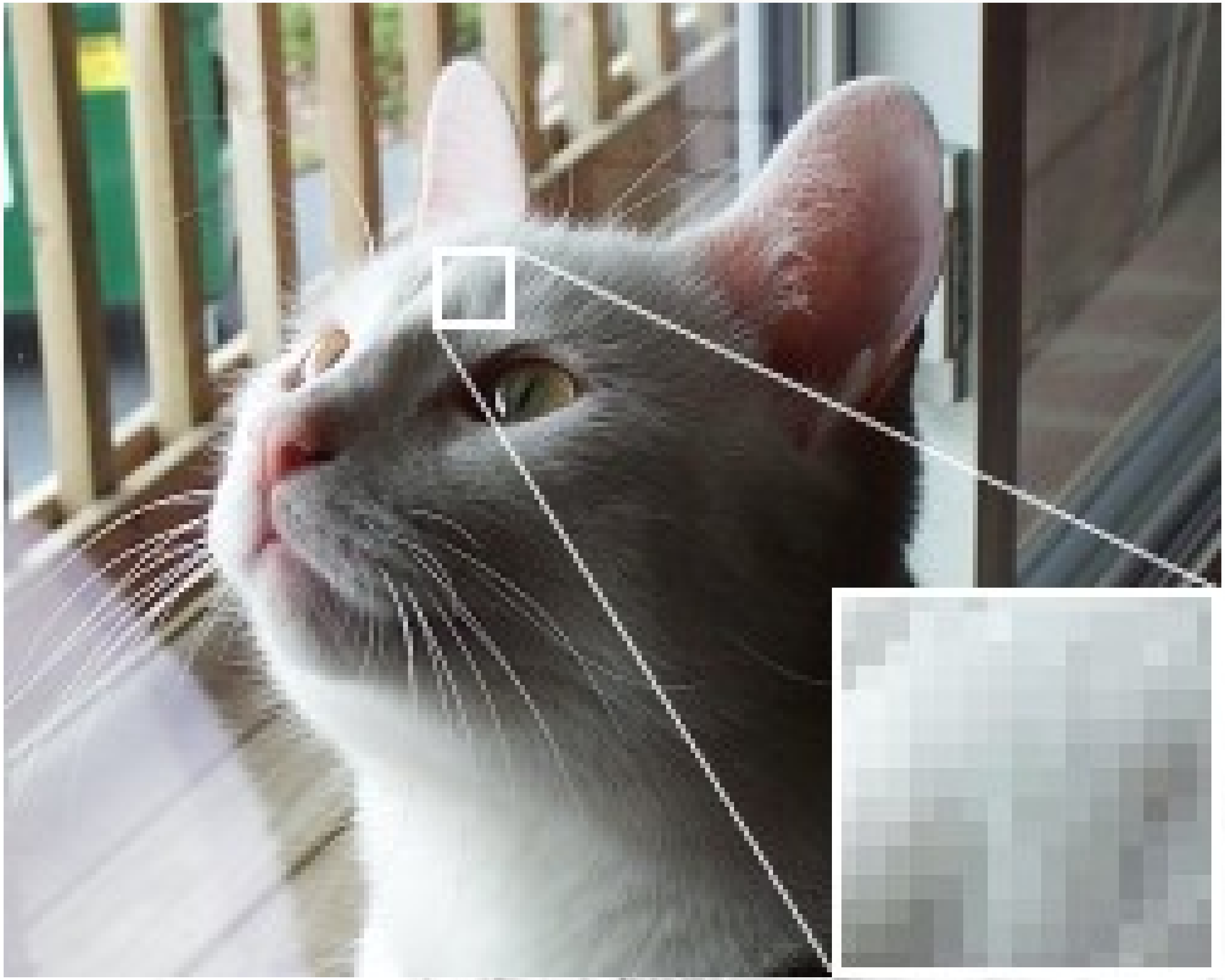
# New topic: Integers

- Numeric type supported in hardware
- Audio
- Images
- Sampled data (real time data acquisition)
  - Often handled as "fixed point" data.



# Every pixel is a triples of 24 bit integers corresponding to RGB value

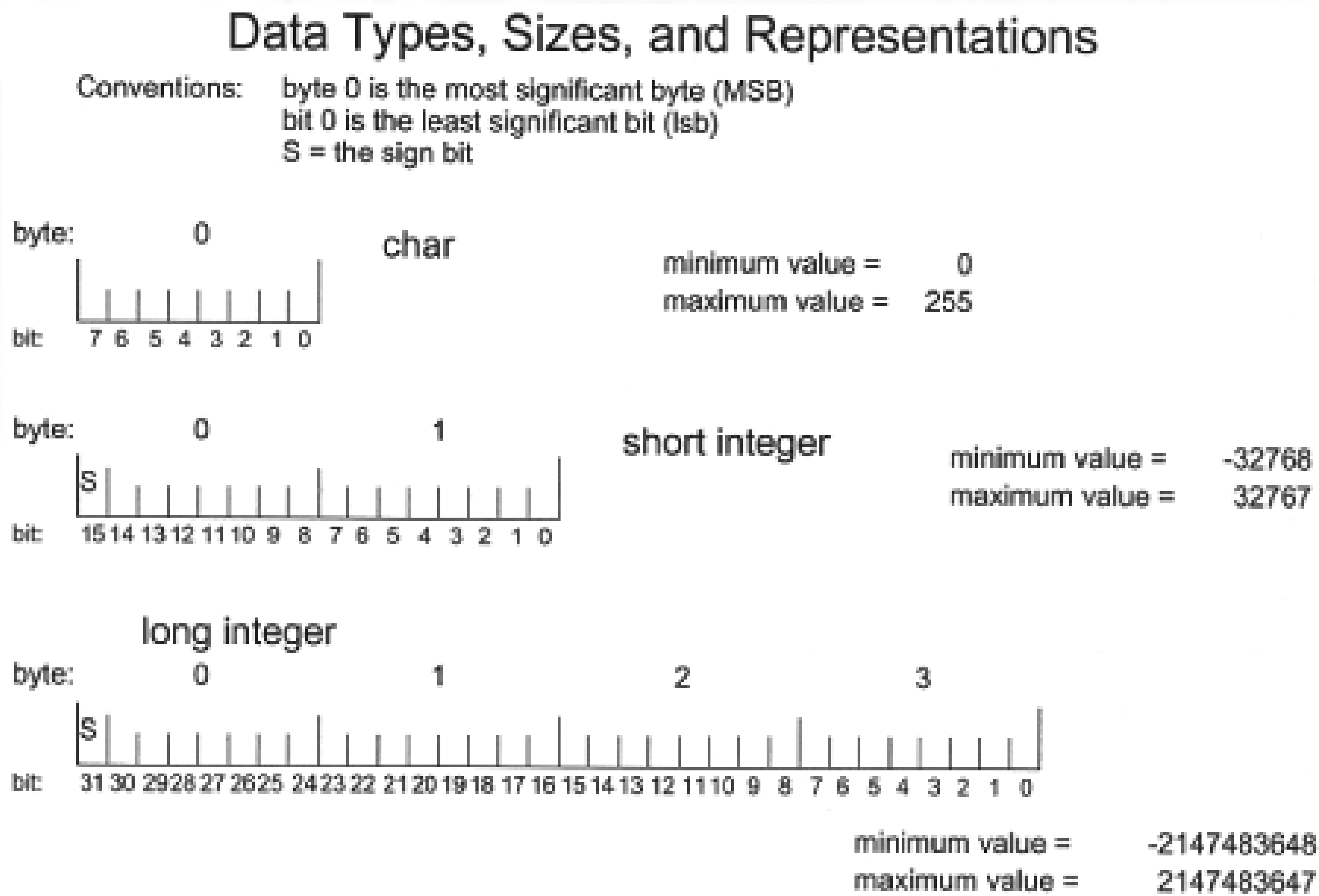


# Unsigned integers

- Uint8: 0 -> 255 = $2^8 - 1$ = 11111111 binary.
- Uint16:  0 -> 65535
- Uint32: 0 -> 4294967295
- Uint64: 0 -> big

- Ints have max and min values
- Intmax('type')

# Signed integers



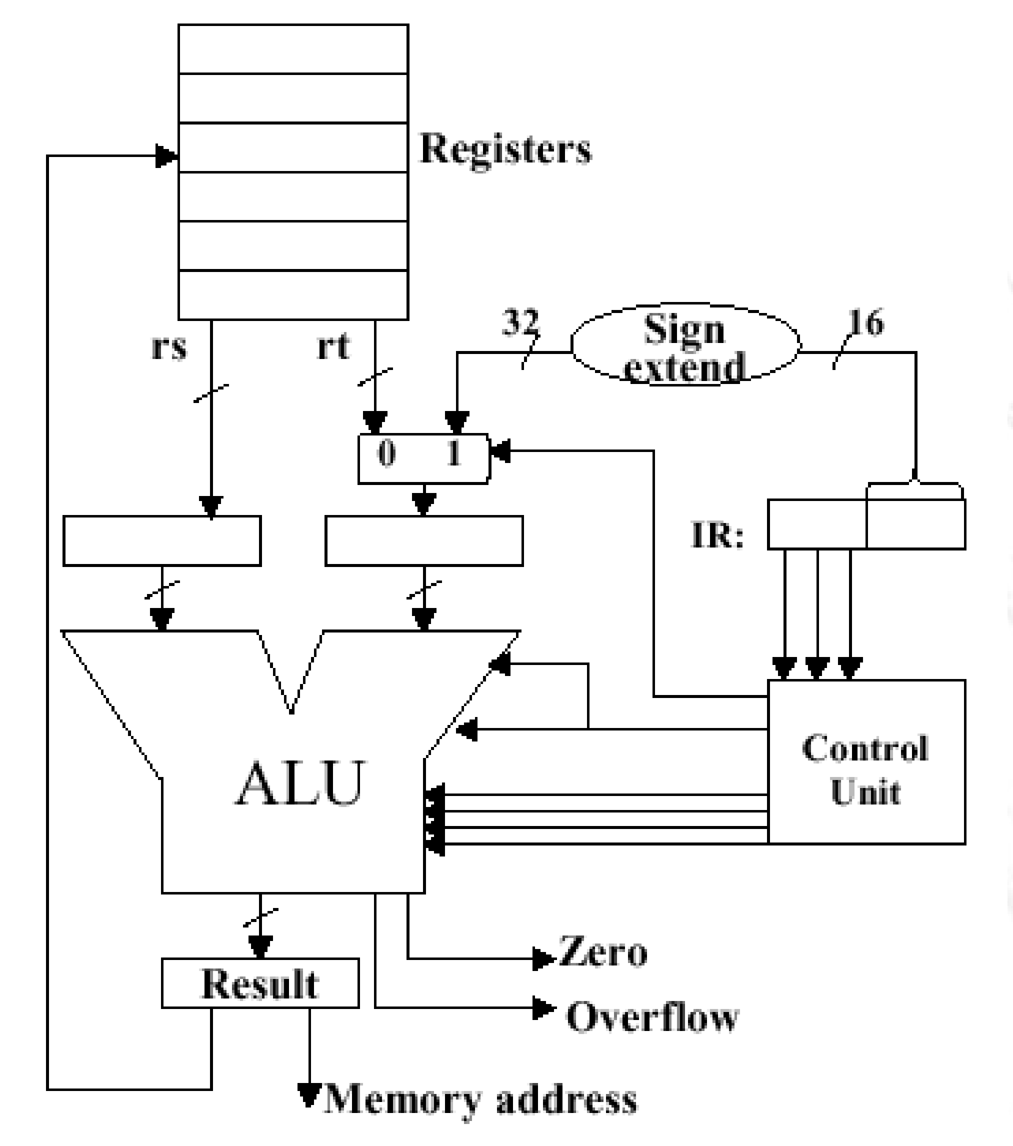
# Two's complement representation of signed ints

- MSB is sign bit
  - 0 = pos
  - 1 = neg
- Take positive value
- Invert all bits
- Add 1
- Example on blackboard

| Decimal value | Binary |
|---|---|
| 127 | 0111 1111 |
| 4 | 0000 0100 |
| 3 | 0000 0011 |
| 2 | 0000 0010 |
| 1 | 0000 0001 |
| 0 | 0000 0000 |
| -1 | 1111 1111 |
| -2 | 1111 1110 |
| -3 | 1111 1101 |
| -4 | 1111 1100 |
| -127 | 1000 0001 |

# Integer ALU

- Integer arithmetic is fast.
- Always supported in hardware.
- Very common in applications:
  - Sound.
  - Images.
  - Real-world signals from an A/D.



# Integer roll-over vs. saturation

- Matlab: ints saturate

```
>> int8(23)
ans =
   23

>> int8(233)
ans =
  127
```

- Python/NumPy: ints roll over.

```
In [11]: import numpy

In [12]: numpy.int8(23)
Out[12]: 23

In [13]: numpy.int8(233)
Out[13]: -23
```

# Why do we care about computer architecture?

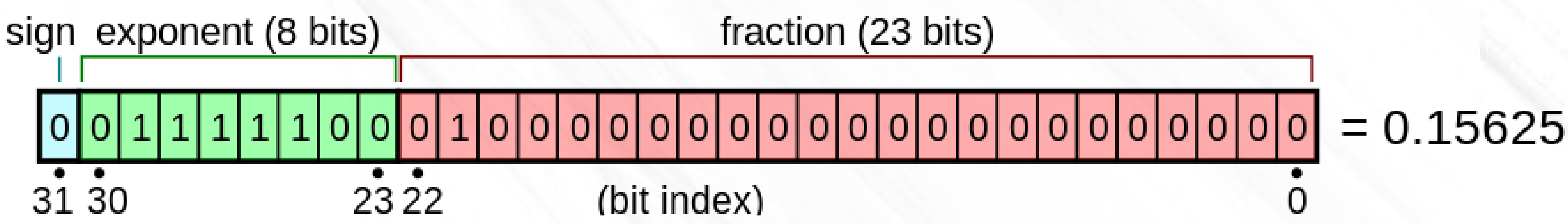
- Operations performed in hardware are fast.
  - Integer computation.
  - Floating point computation (32 and 64 bit).
- Memory accesses are slow.  Cache accesses are fast.
- Performance is one of the themes of this class.

# Recall structure of 32 bit float

Floats are of form s*2$^{e}$*mantissa

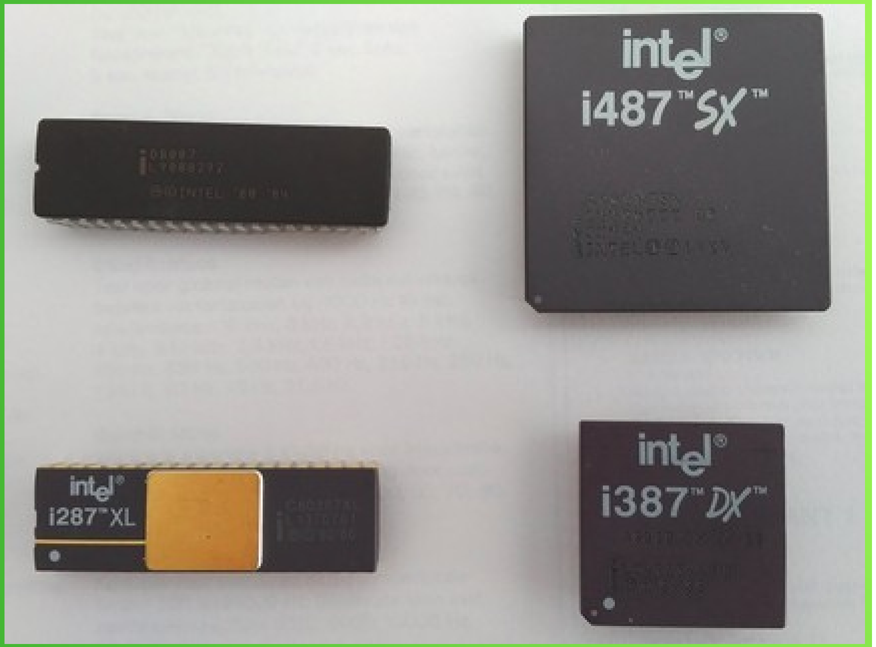
s = sign bit
e = exponent
Mantissa (significand) = 1.xxx

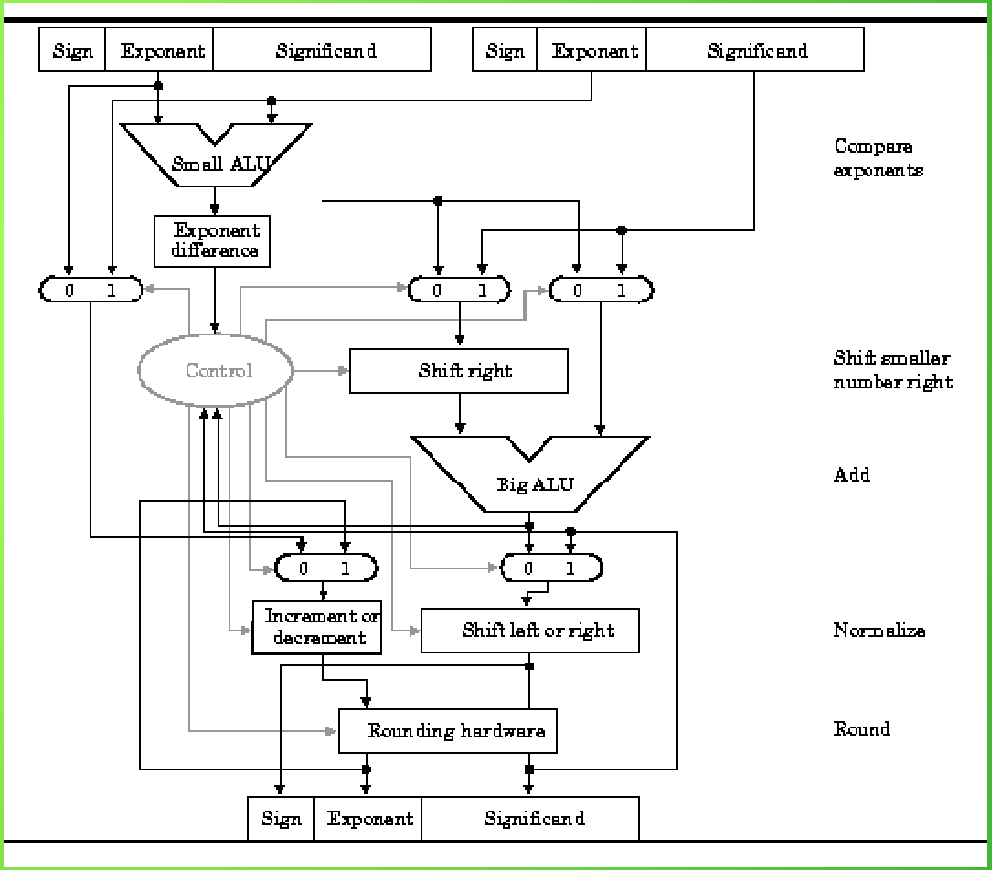
Each group is encoded into some field in the 32 bit word as binary.



# Advantage: Fast floating point

- IEEE 754 floating point computations occur using specialized hardware.
  - Fast!

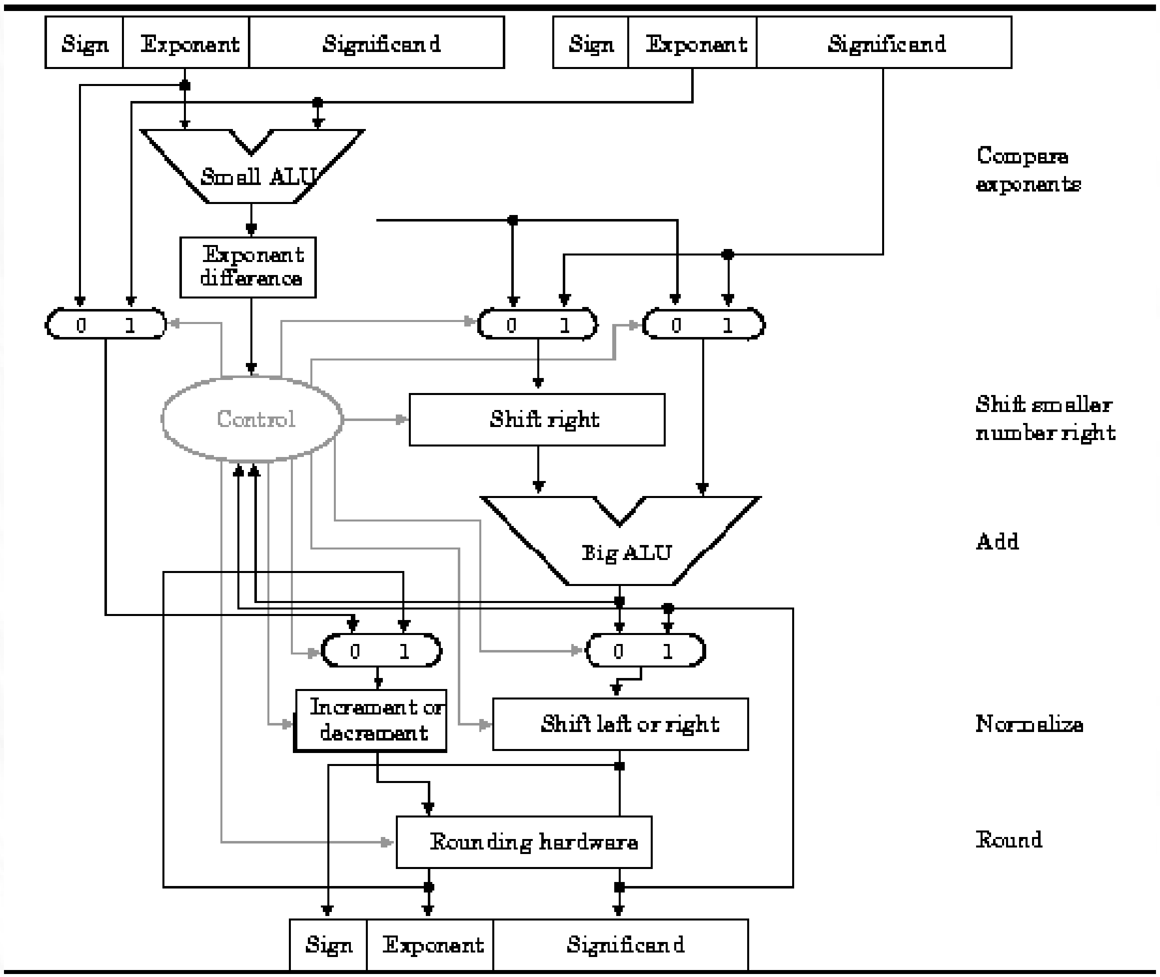



  - But precision is fixed.

# A floating point ALU

32 and 64 bit floating point operations are implemented in hardware (***fast***).



# Next: Accuracy of computations

- Accuracy requirements for basic operations called out in IEEE spec.
- Spec defines “exactly rounded” operations.
  - Exactly rounded: Computed result must lie within ½ ULP of “true” result.
- Addition/Subtraction: Exactly rounded.
- Multiplication: Exactly rounded.
- Sqrt(): Exactly rounded.
- Other functions.

# Accuracy depends upon the magnitude of your numbers

- Decimal numbers may not have exact representation in binary floating point.
- Error depends upon the ULP of your number.

```
octave:3> x = single(1000000.1)
x =  1.0000e+06
octave:4> y = single(1000000.2)
y =  1.0000e+06
octave:5> x - y
ans = -0.062500

octave:6> x = single(1.1)
x =  1.1000
octave:7> y = single(1.2)
y =  1.2000
octave:8> x - y
ans = -0.10000
```

# Floating point operations are ***not*** associative!

```
octave:85> x = single(1.000001)
x =  1.00000095367432
octave:86> y = single(1.000002)
y =  1.00000202655792
octave:87> z = single(1e-6)
z =  9.99999997475243e-07
octave:88> z+(x-y)
ans = -7.28836084817885e-08
octave:89> (z+x)-y
ans = -1.19209289550781e-07
```

- In general, this happens when your variables have different scale.
- Therefore, be careful about scaling your variables.

# New Subject: Dealing with computational errors

- Sources of error:
  - Your mistakes (Not the subject of this section!)
  - Round-off error (Finite word size of computer.)
  - Truncation error (i.e. You stop a series summation before it has fully converged.)
  - Inherent conditioning of the function (e.g. Rapidly varying functions.)
  - Stability of your algorithm (Build-up of errors due to bad algorithm.)
- The following are a bunch of techniques to deal with the imprecision of computer numerics

# Round-off error

- Finite word length acts as source of error by forcing values to lie on the floating point grid.

```
octave:28> format long
octave:29> single(1.1)
ans =  1.10000002384186
```

- Can be interpreted as an error or noise source in your computation.
- The non-uniform floating point grid makes things more complicated....

# Example round-off error effect

```
octave:85> x = single(1.000001)
x =  1.00000095367432
octave:86> y = single(1.000002)
y =  1.00000202655792
octave:87> z = single(1e-6)
z =  9.99999997475243e-07
octave:88> z+(x-y)
ans = -7.28836084817885e-08
octave:89> (z+x)-y
ans = -1.19209289550781e-07
```

**Floating point arithmetic is not associative!**

# Catastrophic cancellation

- x – y can give wrong answer if x and y are close

```
octave:24> x = single(1.000001)
x =  1.0000
octave:25> y = single(1.000002)
y =  1.0000
octave:26> x-y
ans = -1.0729e-06
```

- This phenomenon is also called “loss of significance”.

# Catastrophic cancellation is why we have the Matlab functions expm1 and logp1

- expm1:

$$\exp(x) - 1 = \left(1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \ldots\right) - 1$$

$$expm1(x) = x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \ldots$$

**Catastrophic cancellation for small x**

```
octave:34> format long
octave:35> expm1(1e-10)
ans =  1.00000000005000e-10
octave:36> exp(1e-10) - 1
ans =  1.00000008274037e-10
```

- log1p = log(1+x)

# More Effects of Round-off

```
octave:94> sin(0)
ans = 0
octave:95> sin(pi)
ans =  1.22464679914735e-16
```

```
octave:14> x = 5.123; n = 1; sin(x) - sin(x + 2*n*pi)
ans = -2.2204e-16
octave:15> x = 5.123; n = 10; sin(x) - sin(x + 2*n*pi)
ans = -7.7716e-16
octave:16> x = 5.123; n = 100; sin(x) - sin(x + 2*n*pi)
ans = -2.0317e-14
octave:17> x = 5.123; n = 1000; sin(x) - sin(x + 2*n*pi)
ans =  4.1933e-13
```

Plot by: ill_conditioned_sin_identity

# Comparisons when doing floating point math

- Bad:

```
x = sin(pi);
if (x == 0)
  fprintf('sin(pi) == 0\n');
else
  fprintf('sin(pi) != 0\n');
end
```

**Bad!**

- Good:

```
tol = eps(1);
if (abs(x) < tol)
  fprintf('sin(pi) == 0\n');
else
  fprintf('sin(pi) != 0\n');
end
```

**Good**

# Catastrophic cancellation -- Solving quadratics

$$a x^2 + b x + c = 0 \longrightarrow x = \frac{-b \pm \sqrt{b^2 - 4\mathrm{ac}}}{2\mathrm{a}}$$

- Don't! Can be inaccurate if the two subtracted terms in b are close (if 4ac << $b^2$). This happens if two roots are far apart.
- Example: quadratic_test.m $(x-2)(x+1.23\mathrm{e}17)$

# The right way to compute roots of quadratics

Exploit the fact that $r_1 r_2 = \frac{c}{a}$

Add positive to positive

If b < 0:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad r_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}}$$

Subtract positive from negative

Else:

$$r_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad r_2 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}$$

No catastrophic cancellation

# The right way to compute roots of quadratics

```
function [x1, x2] = quadratic_solve_good(a, b, c)
  if b < 0
    % Compute temp using stable computation which avoids
    % large - large cancellation
    temp = -b + sqrt(b*b - 4*a*c);
    x1 = temp/(2*a);
    x2 = (2*c)/temp;
  else
    % Compute temp using stable computation which avoids
    % large - large cancellation
    temp = -b - sqrt(b*b - 4*a*c);
    x1 = temp/(2*a);
    x2 = (2*c)/temp;
  end
end
```

# Caution when doing sums

- Common suggestion for summing a vector: sort vector first from lowest to highest.  Then sum.
- Kahan summation.

# Final topic: Error and numerical stability

- Concept of stability: Your algorithm might return a result with error, but only unavoidable error.
- We just looked at a bunch of techniques to deal with avoidable errors.
- Now some theoretical concepts dealing with how much error to expect, if your algorithm is stable.

# Concept: Relative vs. Absolute error

You want to compute: $y = f(x)$

The mathematically "true" answer is: $y_{true} = f_{true}(x)$

The computer returns: $y_{computed} = f_{computed}(x)$

In general, $y_{true} \neq y_{computed}$ **Error!**

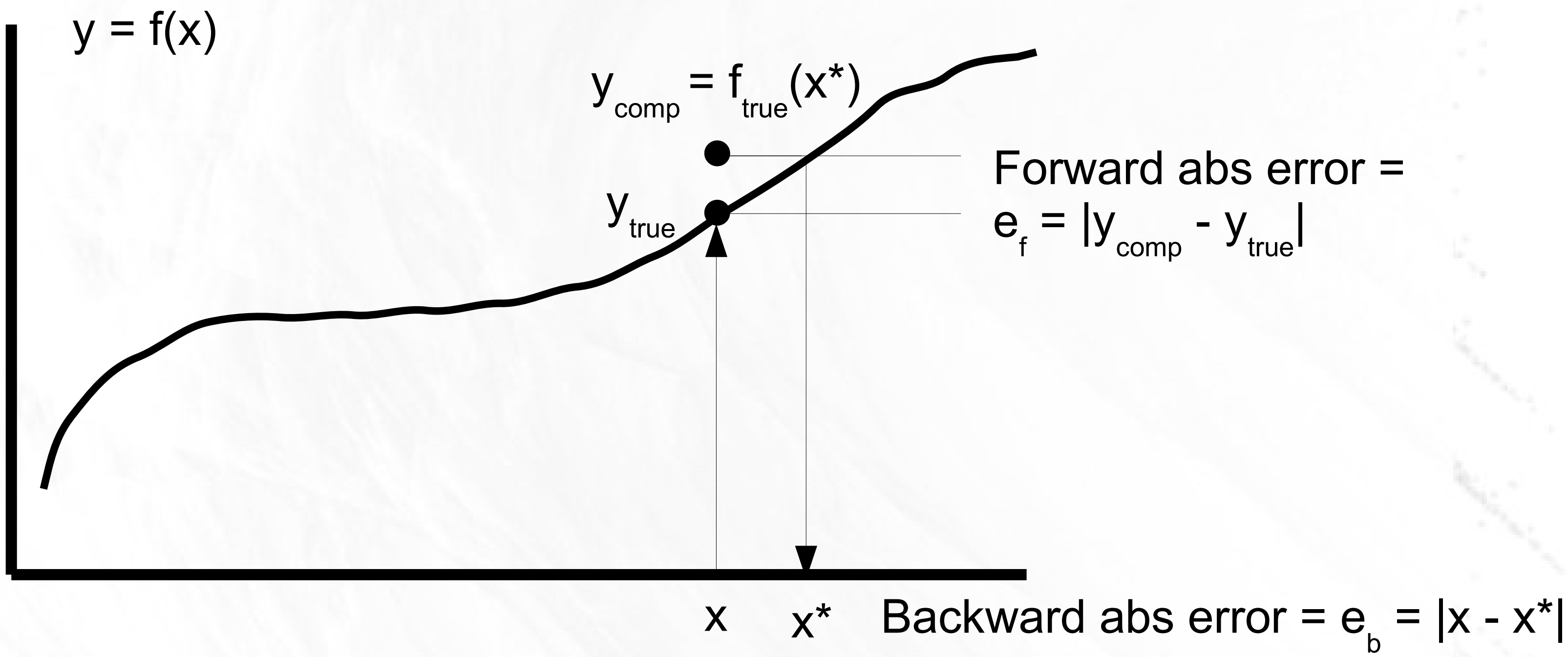
- Absolute error: $e_f = |y_{true} - y_{computed}|$

- Relative error: $\epsilon_f = \dfrac{|y_{true} - y_{computed}|}{|y_{true}|}$

Note one is e and one is epsilon

**You want relative error to be on the order of "machine eps" = eps(1) in Matlab.**

# Forward and backward error




- Forward error: Difference between true and computed values: $|y_{comp} - y_{true}|$
- Backward error: $|x – x^*|$ where $x^*$ yields $y_{comp}$
- Forward error = output error, backward error = equivalent input error.
- You generally have access to forward error, but not backward error.

# Condition number for scalar functions

- Condition number is defined to relate forward (relative) error to backward (relative) error.

True function

Recall definition of backward error

$$y_{comp} = f_t(x + e_b)$$

Compute relative forward error

$$\frac{|y_{true} - y_{comp}|}{|y_{true}|} = \frac{|f_t(x) - f_t(x + e_b)|}{|f_t(x)|}$$

Mean value theorem

$$= \frac{|e_b f_t'(\xi)|}{|f_t(x)|}$$

Relative backward error

Relative forward error

$$\frac{|y_{true} - y_{comp}|}{|y_{true}|} \approx \left|\frac{x f'(x)}{f(x)}\right| \left|\frac{e_b}{x}\right|$$

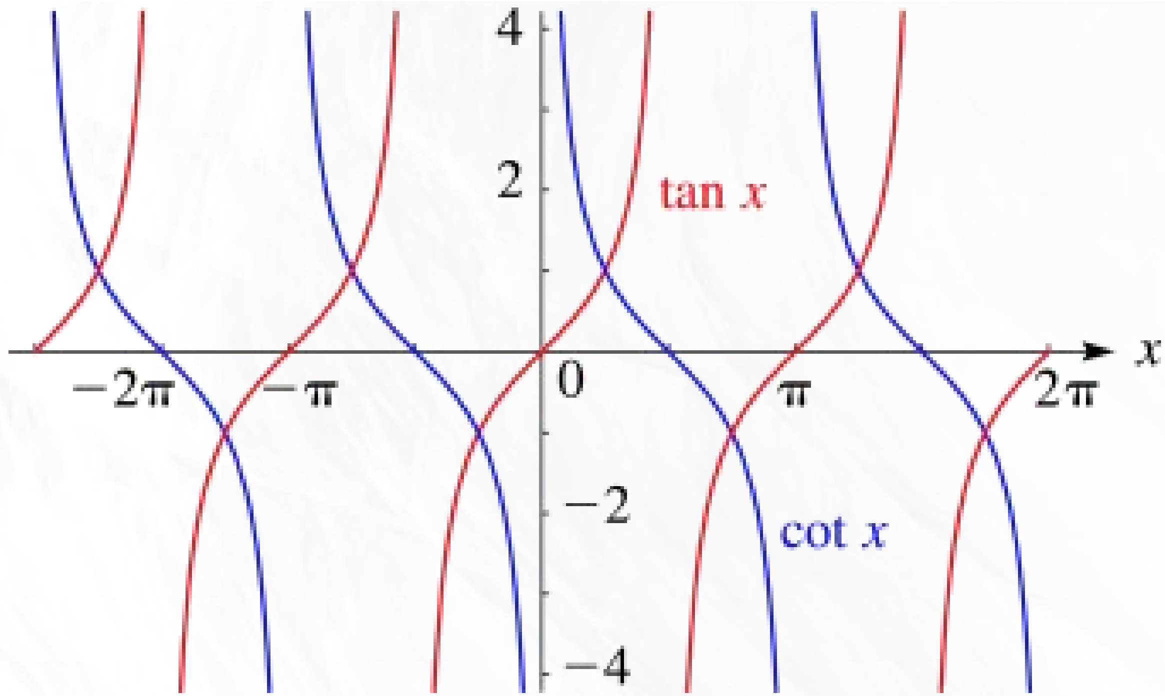
Condition number

# Condition number

$$\kappa(x) = \frac{\epsilon_f}{\epsilon_b} = \left| \frac{x f'(x)}{f(x)} \right|$$

- Relates relative backward error to relative forward error. Characterizes input error to output error!
- Characterizes how error will grow if you compute f(x) -- in best case.
- Ideally, you want k small for good results.
- High k means “watch out!”
- One says a problem is “well conditioned” vs. “badly conditioned”.

# Consider evaluating tan(x)




- Derivative approaches inf at pi/2, 3pi/2, etc.
- Badly conditioned at these points -> small error in x creates large error in y!
- Derivative in definition of condition number.

# Another view: Expected error when evaluating a scalar function

$$e_f = |f_{computed}(x) - f_{true}(x)|$$

$$= |f_{true}(x + e_b) - f_{true}(x)|$$

$$= \left| \frac{\Delta f_{true}(x)}{e_b} \right| |e_b|$$

$$\approx C |f'(x)| eps(x)$$

$e_b$ = C eps(x)

- eps(x) is Matlab function which returns ULP.
- Identify C as a multiplier (hopefully close to 1) which measures the error of the algorithm.

# Relationship to condition number

$$absolute\ error = C\,|f'(x)|\,eps(x) \qquad \kappa = \left|\frac{x f'(x)}{f(x)}\right|$$

$$absolute\ error = C\,\kappa\,|f(x)|\,eps(1)$$

$$relative\ error = \frac{absolute\ error}{|f(x)|} = C\,\kappa\,eps(1)$$

- Stability: Algorithm's relative output error is close to theoretical value.
- Example: A good algorithm should have C near 1, so the relative error of a stable algorithm should be close to k*1e-16 (for doubles).

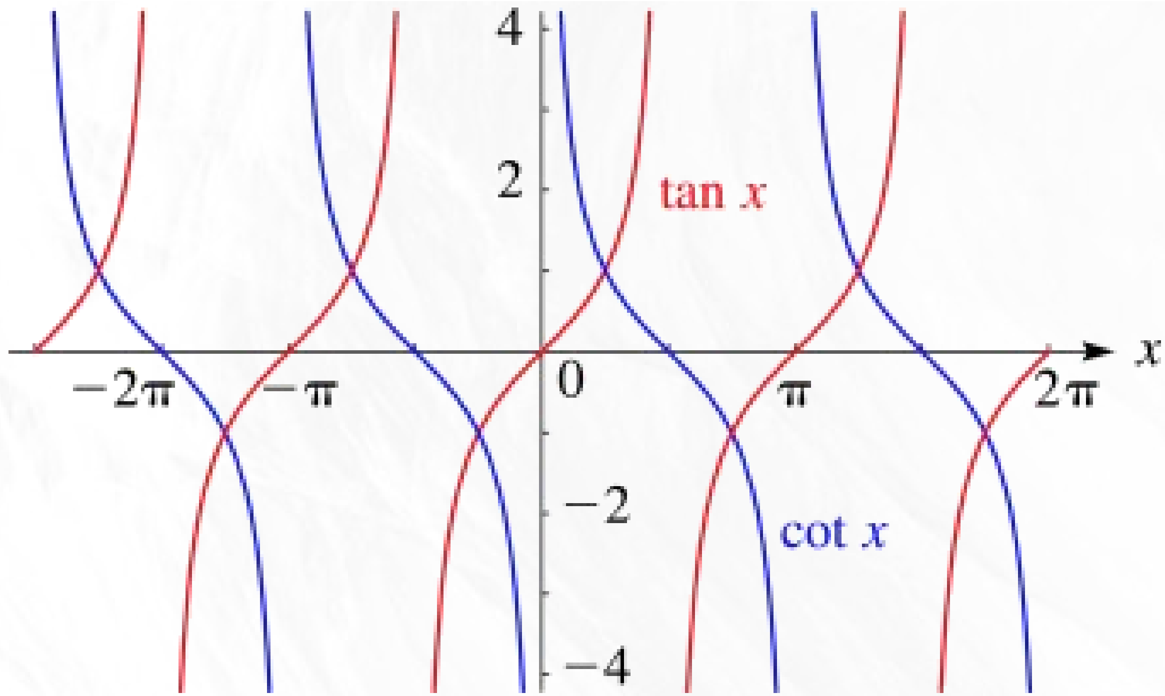
# Conditioning and condition number for scalar functions

$$\kappa(x) = \frac{\epsilon_f}{\epsilon_b} = \left| \frac{x f'(x)}{f(x)} \right|$$

$$relative\ error = C \kappa\ eps(1)$$

- One speaks of a "well conditioned" or "ill conditioned" problem. It is intrinsic property of the function f(x).
- Condition number k characterizes the expected growth of the computational error when the function f(x) is evaluated.
- However, k says nothing about the stability of your algorithm.
- If k is close to 1, and your algorithm is stable, relative error can be around eps(1) (1e-16 for doubles).

# Consider evaluating tan(x)




- Derivative approaches inf at pi/2, 3pi/2, etc.
- Badly conditioned at these points -> small error in x creates large error in y!
- Libm implementation has C = 1, so relative error depends only upon condition number.

# A word about the homework.....

- Write the assigned function using your favorite language (default Matlab).
- Important:  Write a test which calls your function!
    - Loop over some input values and find way to verify the returns are correct.
    - Test a few “corner cases”.
- Hand in both your function implementation and your test harness.

# Major points to remember

- Computer architecture impacts numerics.
  - Performance
- Integer and floating point numbers
- Numerical stability and error
  - Beware of round-off error and “catastrophic cancellation”.
  - The stability of the function you are computing is characterized by its “condition number”.
- Always write a test for your function!