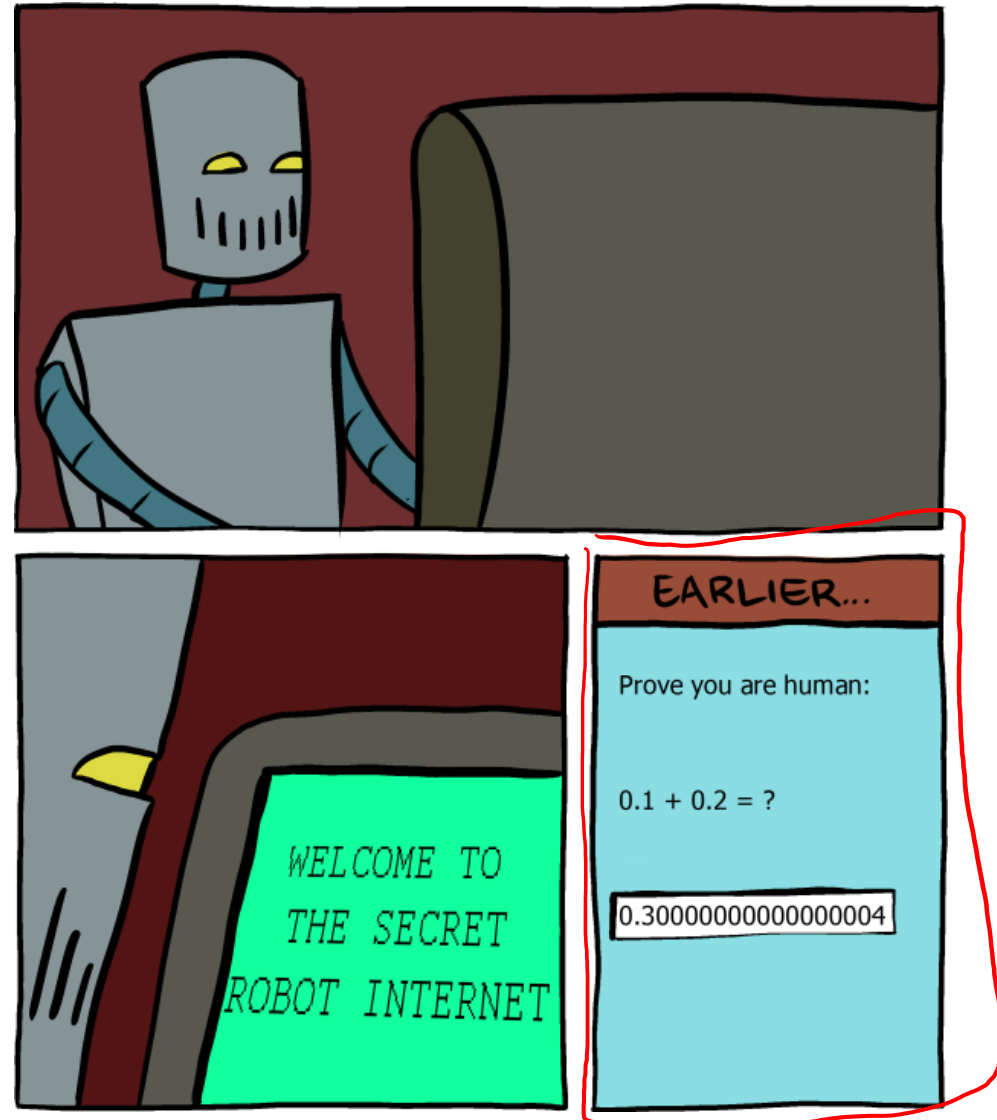


# Machine Programming

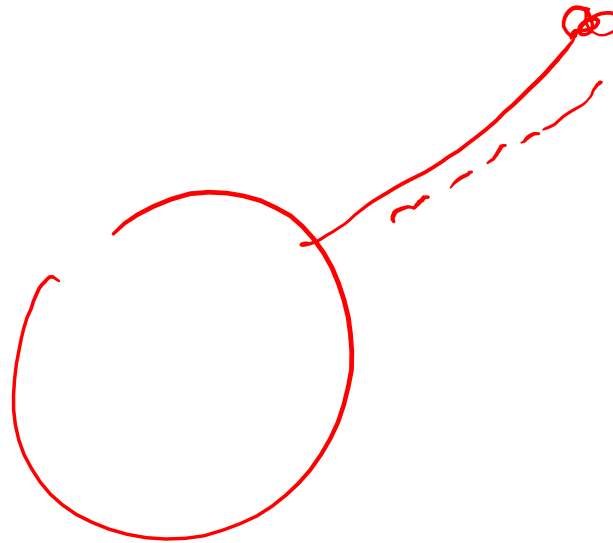
CSE 351 Winter 2017



<http://www.smbc-comics.com/?id=2999>

# Administrivia

- ❖ Lab 1 due today!
- ❖ Lab 2 out Monday 😊



# Mathematical Properties of FP Operations

$$(-1)^s \times M \times 2^E \uparrow$$

- ❖ Exponent overflow yields  $+\infty$  or  $-\infty$
- ❖ Floats with value  $+\infty$ ,  $-\infty$ , and NaN can be used in operations
  - Result usually still  $+\infty$ ,  $-\infty$ , or NaN; sometimes intuitive, sometimes not
- ❖ Floating point ops do not work like real math, due to **rounding!**
  - **Not associative:**  $(3.14 + 1e100) - 1e100 \neq 3.14 + (1e100 - 1e100)$
  - **Not distributive:**  $100 * (0.1 + 0.2) \neq 100 * 0.1 + 100 * 0.2$   
 $30.0000000000000003553 \quad 30$
  - **Not cumulative**
    - Repeatedly adding a very small number to a large one may do nothing

# Floating Point in C

→ unsigned u = 10;  
→ int i = (int) u;



- ❖ C offers two (well, 3) levels of precision

float 1.0f single precision (32-bit)

23 bits frac 8 bits exp.

double 1.0 double precision (64-bit)

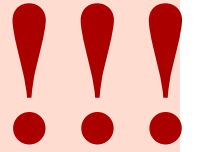
52 bits 11 bits exp

long double 1.0L (double double, quadruple, or "extended") precision (64-128 bits)

- ❖ #include <math.h> to get INFINITY and NAN constants
- ❖ Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results
  - Just avoid them!

# Floating Point in C

*int i = 1;  
float f = (float)i;*



## ❖ Conversions between data types:

- Casting between int, float, and double **changes** the bit representation.
- int → float
  - May be rounded (not enough bits in mantissa: 23)
  - Overflow impossible
- int → double or float → double
  - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
- long → double
  - Rounded or exact, depending on word size (64-bit → 52 bit mantissa ⇒ round)
- double or float → int
  - Truncates fractional part (rounded toward zero)
    - E.g. 1.999 → 1, -1.99 → -1
  - “Not defined” when out of range or NaN: generally sets to Tmin (even if the value is a very big positive)

# Floating Point and the Programmer

```
#include <stdio.h>
```

```
int main(int argc, char* argv[]) {
```

```
→ float f1 = 1.0;
→ float f2 = 0.0;
int i;
for (i = 0; i < 10; i++) {
    f2 += 1.0/10.0;
}
```

$$0.0 + 10 \times \left( \frac{1.0}{10.0} \right)$$

```
printf("0x%08x 0x%08x\n", *(int*)&f1, *(int*)&f2);
printf("f1 = %10.8f\n", f1);
printf("f2 = %10.8f\n\n", f2);
```

```
→ f1 = 1E30;
→ f2 = 1E-30;
float f3 = f1 + f2;
printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");

return 0;
}
```

~~\*~~ ((int\*) &f2)

\$ ./a.out

0x3f800000 0x3f800001

f1 = 1.000000000

f2 = 1.000000119

f1 == f3? yes

# Number Representation Really Matters

- ❖ 1991: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- ❖ 1996: Ariane 5 rocket exploded (\$1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- ❖ 2000: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- ❖ 2038: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- ❖ other related bugs
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug (\$475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch (\$193 million)

# Summary

- ❖ As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”
- ❖ Mathematically equivalent ways of writing an expression may compute different results
  - Violates associativity/distributivity
- ❖ **Never** test floating point values for equality!
- ❖ **Careful** when converting between ints and floats!



# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

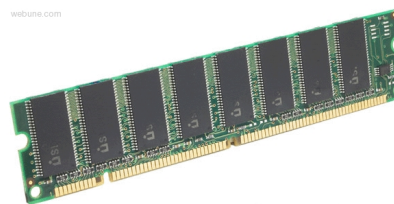
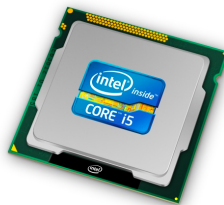
Assembly  
language:

```
get_mpg:
    pushq    %rbp
    movq     %rsp, %rbp
    ...
    popq     %rbp
    ret
```

Machine  
code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer  
system:



Memory & data ✓  
Integers & floats  
**Machine code & C**  
 x86 assembly  
 Procedures & stacks  
 Arrays & structs  
 Memory & caches  
 Processes  
 Virtual memory  
 Memory allocation  
 Java vs. C

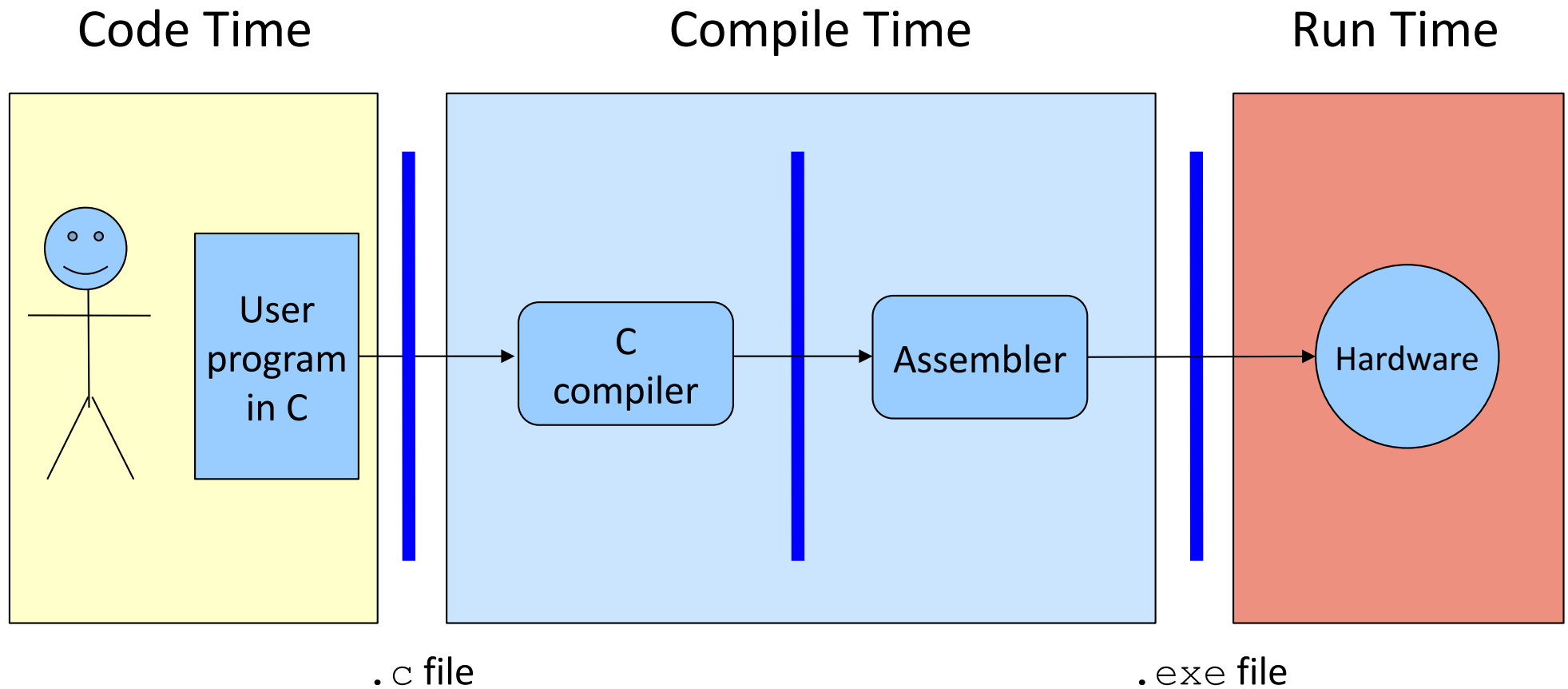
OS:



# Basics of Machine Programming & Architecture

- ❖ What is an ISA (Instruction Set Architecture)?
- ❖ A brief history of Intel processors and architectures
- ❖ C, assembly, machine code

# Translation



What makes programs run fast(er)?

# Translation

## Source code

Different applications  
or algorithms

*a = b + c*

## Compiler

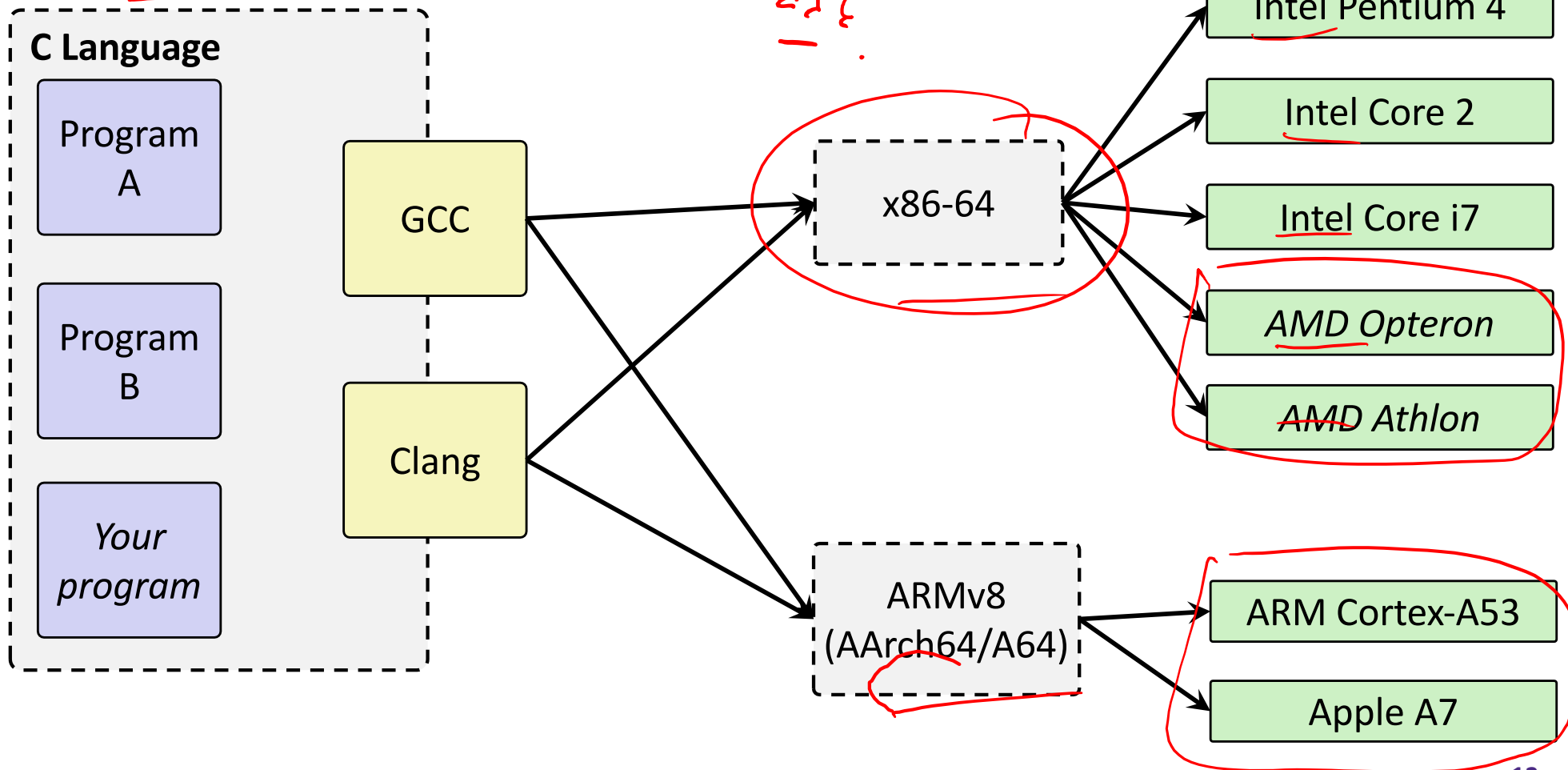
Perform optimizations,  
generate instructions

## Architecture

Instruction set

## Hardware

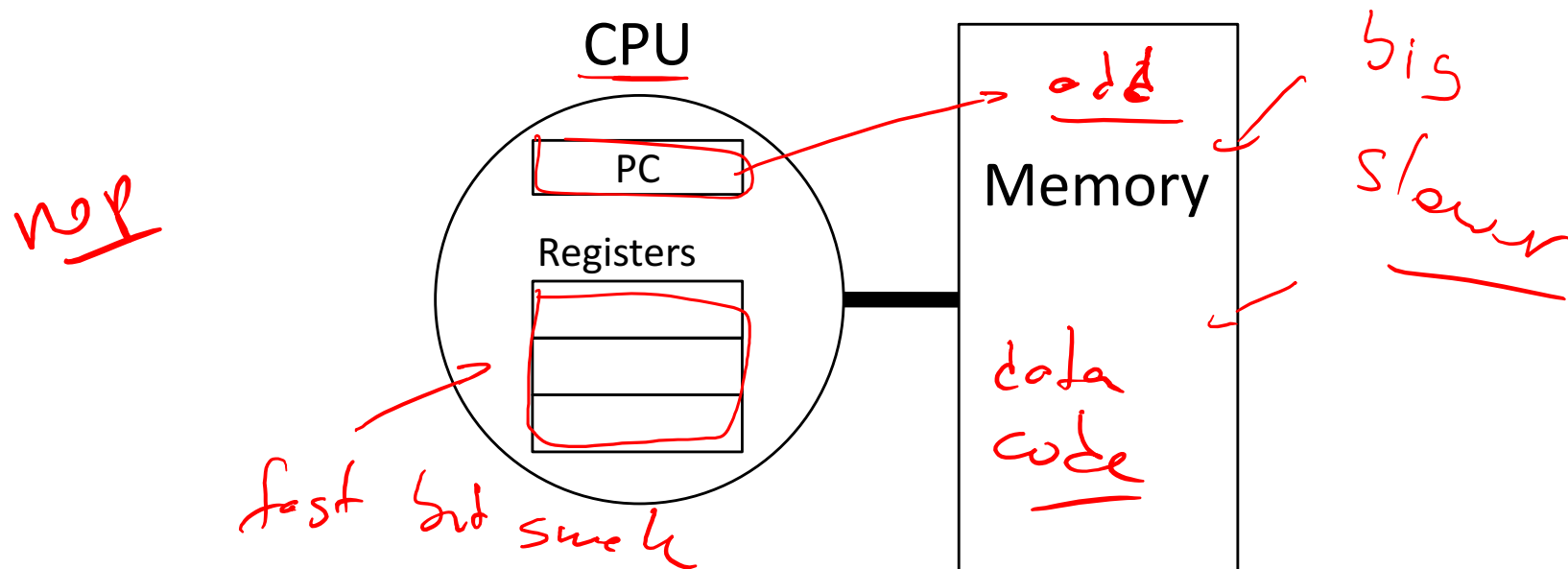
Different  
implementations



# Instruction Set Architectures

## ❖ The ISA defines:

- The system's state (e.g. registers, memory, program counter)
- The instructions the CPU can execute
- The effect that each of these instructions will have on the system state



# Instruction Set Philosophies

- ❖ *Complex Instruction Set Computing (CISC):* Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs
- ❖ *Reduced Instruction Set Computing (RISC):* Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

# General ISA Design Decisions

## ❖ Instructions

- What instructions are available? What do they do?
- How are they encoded?

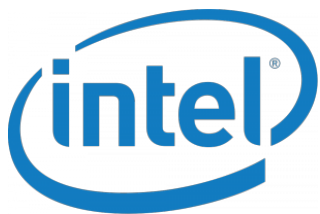
## ❖ Registers

- How many registers are there?
- How wide are they?

## ❖ Memory

- How do you specify a memory location?

# Mainstream ISAs



## x86

<b>Designer</b>	Intel, AMD
<b>Bits</b>	16-bit, 32-bit and 64-bit
<b>Introduced</b>	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
<b>Design</b>	CISC
<b>Type</b>	Register-memory
<b>Encoding</b>	Variable (1 to 15 bytes)
<b>Endianness</b>	Little

Macbooks & PCs  
(Core i3, i5, i7, M)  
[x86-64 Instruction Set](#)



## ARM architectures

<b>Designer</b>	ARM Holdings
<b>Bits</b>	32-bit, 64-bit
<b>Introduced</b>	1985; 31 years ago
<b>Design</b>	RISC
<b>Type</b>	Register-Register
<b>Encoding</b>	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 <a href="#">user-space compatibility</a> <sup>[1]</sup>
<b>Endianness</b>	Bi (little as default)

Smartphone-like devices  
(iPhone, iPad, Raspberry Pi)  
[ARM Instruction Set](#)



## MIPS

<b>Designer</b>	MIPS Technologies, Inc.
<b>Bits</b>	64-bit (32→64)
<b>Introduced</b>	1981; 35 years ago
<b>Design</b>	RISC
<b>Type</b>	Register-Register
<b>Encoding</b>	Fixed
<b>Endianness</b>	Bi

Digital home & networking  
equipment  
(Blu-ray, PlayStation 2)  
[MIPS Instruction Set](#)



# Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
❖ 8086	1978	29K	5-10
<ul style="list-style-type: none"><li>■ First 16-bit Intel processor. Basis for IBM PC &amp; DOS</li><li>■ 1MB address space</li></ul>			
❖ 386	1985	275K	16-33
<ul style="list-style-type: none"><li>■ First 32 bit Intel processor , referred to as IA32</li><li>■ Added “flat addressing,” capable of running Unix</li></ul>			
❖ Pentium 4E	2004	125M	2800-3800
<ul style="list-style-type: none"><li>■ First 64-bit Intel x86 processor, referred to as x86-64</li></ul>			
❖ Core 2	2006	291M	1060-3500
<ul style="list-style-type: none"><li>■ First multi-core Intel processor</li></ul>			
❖ Core i7	2008	731M	1700-3900
<ul style="list-style-type: none"><li>■ Four cores</li></ul>			

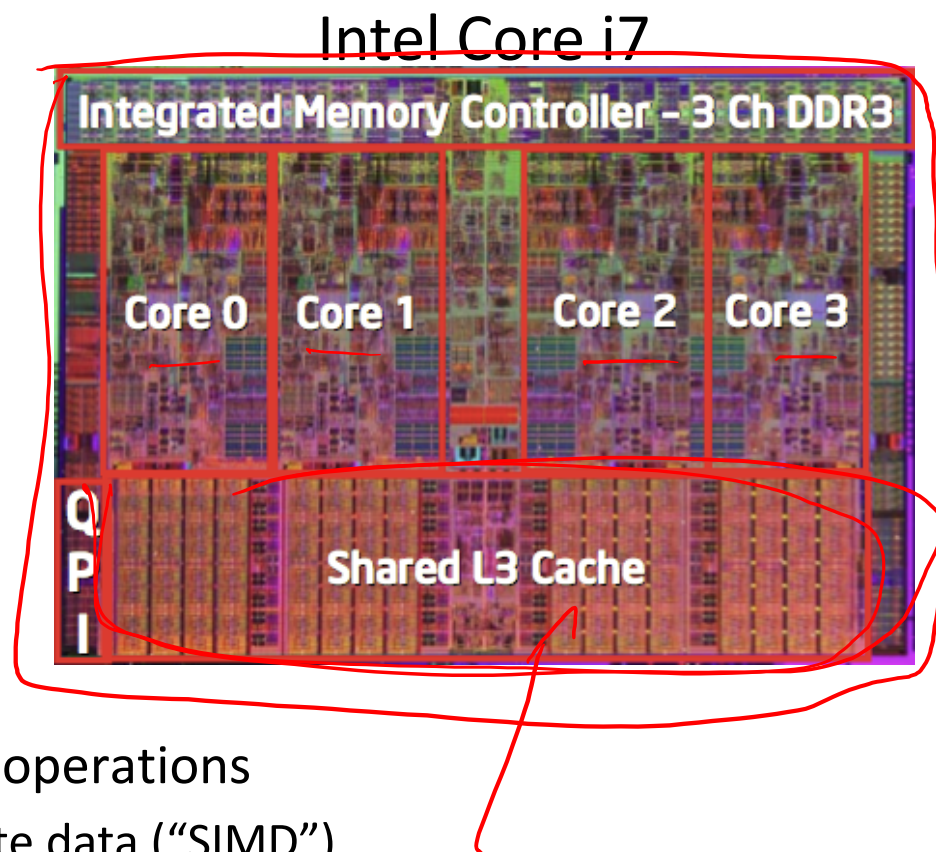
# Intel x86 Processors

## ❖ Machine Evolution

■ 486	<u>1989</u>	<u>1.9M</u>
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ Pentium Pro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	<u>2008</u>	<u>731M</u>

## ❖ Added Features

- Instructions to support multimedia operations
  - Parallel operations on 1, 2, and 4-byte data (“SIMD”)
- Instructions to enable more efficient conditional operations
- Hardware support for virtualization (virtual machines)
- More cores!



# More information

- ❖ References for Intel processor specifications:
  - Intel's "automated relational knowledgebase":
    - <http://ark.intel.com/>
  - Wikipedia:
    - [http://en.wikipedia.org/wiki/List\\_of\\_Intel\\_microprocessors](http://en.wikipedia.org/wiki/List_of_Intel_microprocessors)

# x86 Clones: Advanced Micro Devices (AMD)

- ❖ Same ISA, different implementation
- ❖ Historically AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- ❖ Then recruited top circuit designers from Digital Equipment Corporation (DEC) and other downward-trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension of x86 to 64 bits

# Intel's Transition to 64-Bit

- ❖ Intel attempted radical shift from IA32 to IA64 (2001)
  - Totally different architecture (Itanium) and ISA than x86
  - Executes IA32 code only as legacy
  - Performance disappointing
- ❖ AMD stepped in with *evolutionary* solution (2003)
  - x86-64 (also called “AMD64”)
- ❖ Intel felt obligated to focus on IA64
  - Hard to admit mistake or that AMD is better
- ❖ Intel announces “EM64T” extension to IA32 (2004)
  - Extended Memory 64-bit Technology
  - Almost identical to AMD64!
- ❖ Today: all but low-end x86 processors support x86-64
  - But, lots of code out there is still just IA32

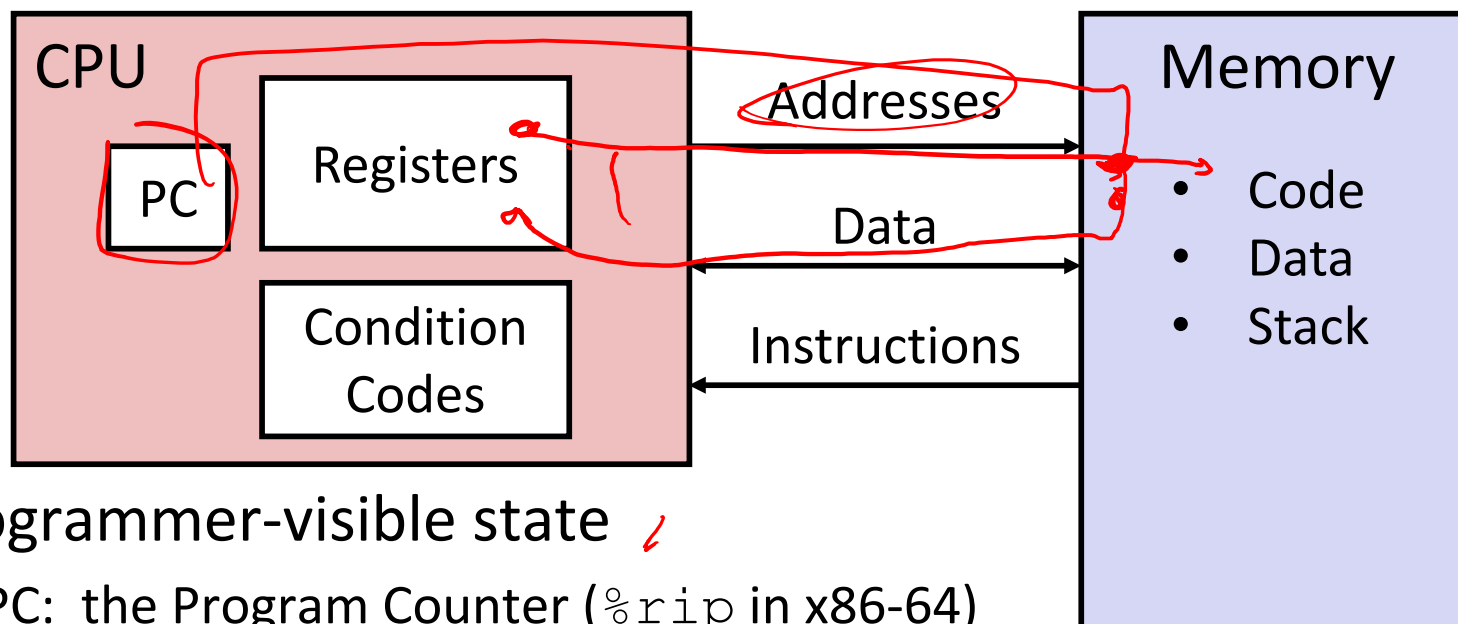
# Our Coverage in 351

- ❖ x86-64
  - The new 64-bit x86 ISA – all lab assignments use x86-64!
  - Book covers x86-64
- ❖ Previous versions of CSE 351 and 2<sup>nd</sup> edition of textbook covered IA32 (traditional 32-bit x86 ISA) and x86-64
  - We will only cover x86-64 this quarter

# Definitions

- ❖ **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”
- ❖ **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469, 470
- ❖ Are the following part of the architecture?
  - Number of registers? Yes
  - How about CPU frequency? No
  - Cache size? Memory size? No

# Assembly Programmer's View



## ❖ Programmer-visible state ↴

- PC: the Program Counter (%rip in x86-64)
  - Address of next instruction ///
- Named registers
  - Together in “register file”
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

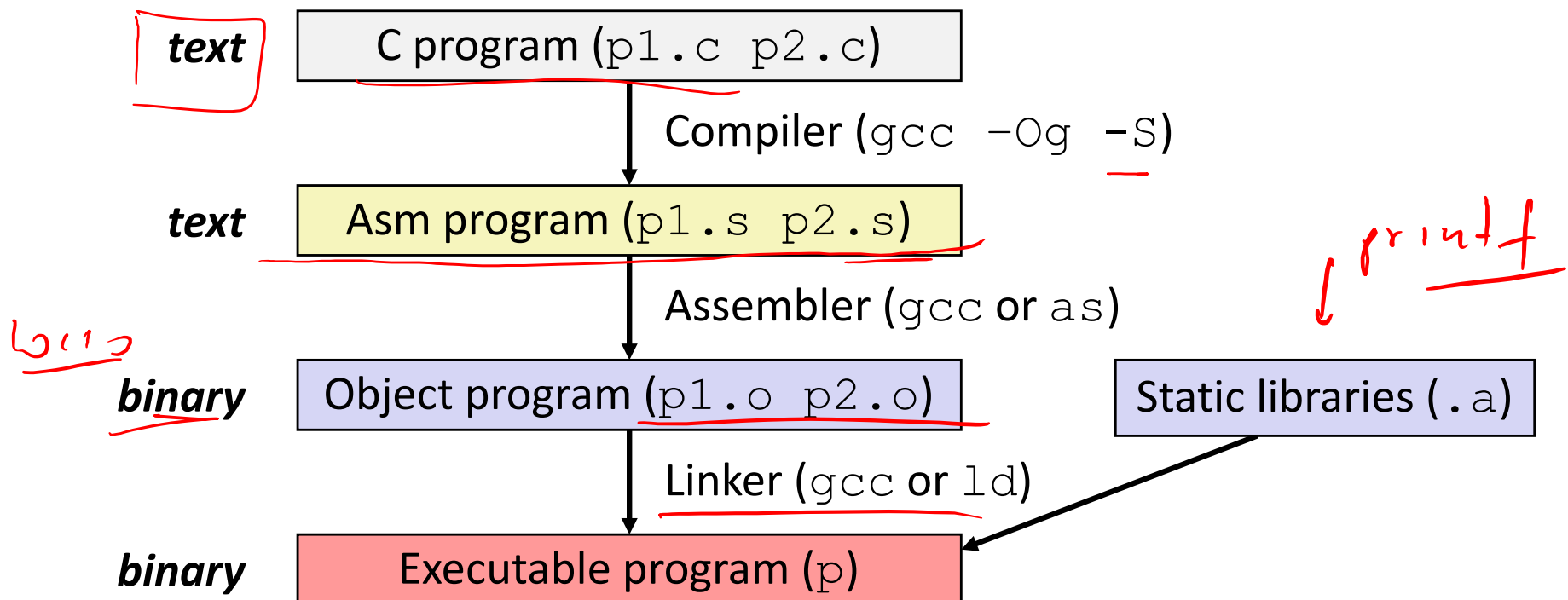
## ❖ Memory

- Byte-addressable array
- Code and user data
- Includes *the Stack* (for supporting procedures)



# Turning C into <sup>Binary</sup> Object Code

- ❖ Code in files p1.c p2.c
- ❖ Compile with command: gcc -Og p1.c p2.c -o p
  - Use basic optimizations (-Og) [New to recent versions of GCC]
  - Put resulting machine code in file p



# Compiling Into Assembly

## ❖ C Code (sum.c)

```
void sumstore(long x, long y, long *dest) {  
    long t = x + y;  
    *dest = t;  
}
```

## ❖ x86-64 assembly (gcc -Og -S sum.c)

- Generates file `sum.s` (see <https://godbolt.org/g/pQUhIZ>)

```
sumstore(long, long, long*):  
  addq    %rdi, %rsi       $x + y$   
  movq    %rsi, (%rdx)  
  ret
```

Warning: You may get different results with other versions of gcc and different compiler settings

# Machine Instruction Example

```
*dest = t;
```

## ❖ C Code

- Store value `t` where designated by `dest`

```
movq %rsi, (%rdx)
```

## ❖ Assembly

- Move 8-byte value to memory
  - Quad word (`q`) in x86-64 parlance
- Operands:

`t`                      Register `%rsi`

`dest`                  Register `%rdx`

`*dest`                Memory `M[%rdx]`

```
0x400539:  48  89  32
```

## ❖ Object Code

- 3-byte instruction (in hex)
- Stored at address `0x40059e`

# Object Code

Function starts at  
this address

0x00400536 <sumstore>:  
0x48  
0x01  
0xfe  
0x48  
0x89  
0x32  
0xc3

Total of 7 bytes  
• Each instruction  
here is 1-3 bytes  
long

❖ **Assembler** translates .s into .o

- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

❖ **Linker** resolves references between files

- Combines with static run-time libraries
  - e.g., code for `malloc`, `printf`
- Some libraries are *dynamically linked*
  - Linking occurs when program begins execution

# Disassembling Object Code

## ❖ Disassembled:

```
0000000000400536 <sumstore>:  
  400536:  48 01 fe      add    %rdi,%rsi  
  400539:  48 89 32      mov    %rsi, (%rdx)  
  40053c:  c3           retq
```

## ❖ Disassembler (objdump -d sum)

- Useful tool for examining object code (man 1 objdump)
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can run on either a .out (complete executable) or .o file

# Alternate Disassembly in GDB

```
$ gdb sum
(gdb) disassemble sumstore
Dump of assembler code for function sumstore:
    0x0000000000400536 <+0>:      add     %rdi,%rsi
    0x0000000000400539 <+3>:      mov     %rsi, (%rdx)
    0x000000000040053c <+6>:      retq
End of assembler dump.

(gdb) x/7bx sumstore
0x400536 <sumstore>: 0x48  0x01  0xfe  0x48  0x89  0x32  0xc3
```

- ❖ Within gdb debugger (gdb sum):
  - disassemble sumstore: disassemble procedure
  - x/7bx sumstore: show 7 bytes starting at sumstore

# What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:      file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

**Reverse engineering forbidden by  
Microsoft End User License Agreement**

- ❖ Anything that can be interpreted as executable code
- ❖ Disassembler examines bytes and attempts to reconstruct assembly source

# Summary

- ❖ Converting between integral and floating point data types *does* change the bits
- ❖ Floating point rounding is a HUGE issue!
  - Limited mantissa bits cause inaccurate representations
  - In general, floating point arithmetic is NOT associative or distributive
- ❖ x86-64 is a complex instruction set computing (CISC) architecture
- ❖ An executable binary file is produced by running code through a **compiler**, **assembler**, and **linker**



# BONUS SLIDES

More details for the curious.

- ❖ Rounding strategies
- ❖ Floating Point Puzzles

# Closer Look at Round-To-Even

## ❖ Default Rounding Mode

- Hard to get any other kind without dropping into assembly
- All others are statistically biased
  - Sum of set of positive numbers will consistently be over- or under- estimated

## ❖ Applying to Other Decimal Places / Bit Positions

- When exactly halfway between two possible values
  - Round so that least significant digit is even
- E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

# Rounding Binary Numbers

## ❖ Binary Fractional Numbers

- “Half way” when bits to right of rounding position =  $100..._2$

## ❖ Examples

- Round to nearest  $1/4$  (2 bits right of binary point)

Value	Binary	Rounded	Action	Round Val
$2 + \frac{3}{32}$	$10.00011_2$	$10.00_2$	(<1/2—down)	2
$2 + \frac{3}{16}$	$10.00110_2$	$10.01_2$	(>1/2—up)	$2 + \frac{1}{4}$
$2 + \frac{7}{8}$	$10.11100_2$	$11.00_2$	( 1/2—up)	3
$2 + \frac{5}{8}$	$10.10100_2$	$10.10_2$	( 1/2—down)	$2 + \frac{1}{2}$

# Floating Point Puzzles



❖ For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```
int x = ...;
float f = ...;
double d = ...;
double d2 = ...;
```

Assume neither `d` nor `f` is NaN

- 1) `x == (int)(float) x`
- 2) `x == (int)(double) x`
- 3) `f == (float)(double) f`
- 4) `d == (double)(float) d`
- 5) `f == -(-f);`
- 6) `2/3 == 2/3.0`
- 7) `(d+d2)-d == d2`