Machine Programming
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

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

- Lab 1 due on Friday @ 5pm
- Homework 1 released
  - On number representation (signed, unsigned, floating point) and x86 (starting today)
- Section room change: AD/AH now in EEB 045
Floating point topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
Rounding modes

- Possible rounding modes (money example):
  - Round-toward-zero: $1, $1, $1, $2, –$1
  - Round-down (−∞): $1, $1, $1, $2, –$2
  - Round-up (+∞): $2, $2, $2, $3, –$1
  - Round-to-nearest: $1, $2, ??, ??, ??
  - Round-to-even: $1, $2, $2, $2, –$2

- **Round-to-even** avoids statistical bias in repeated rounding
  - Rounds up about half the time, down about half the time
  - This is the default rounding mode for IEEE floating-point
Mathematical Properties of FP Operations

- 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; but not always intuitive
- Floating point operations do not work like real math, due to rounding
  - Not associative: $(3.14+1e100)-1e100 \neq 3.14+(1e100-1e100)$
  - Not distributive: $100*(0.1+0.2) \neq 100*0.1+100*0.2$
    - $30.000000000000003553 \neq 30$
- Not cumulative
  - Repeatedly adding a very small number to a large one may do nothing
    - $3.14 + 1e100 \rightarrow 1e100$
Floating Point in C

- C offers two (well, 3) levels of precision
  - `float 1.0f` single precision (32-bit)
  - `double 1.0` double precision (64-bit)
  - `long double 1.0L` ("double double" or quadruple) 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, so just avoid them!
  
  Instead use $\text{abs}(f1 - f2) < 2^{-20}$

  \[ \text{some arbitrary threshold} \]
Floating Point Conversions in C

- Casting between `int`, `float`, and `double` changes the bit representation
  - `int → float`
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - `int or float → double`
    - Exact conversion (all 32-bit ints representable)
  - `long → double`
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - `double or float → int`
    - Truncates fractional part (rounded toward zero)
    - “Not defined” when out of range or NaN: generally sets to $T_{min}$
      (even if the value is a very big positive)
# Floating Point and the Programmer

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;  // specify float constant
    float f2 = 0.0;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;
    f2 should == 1.0 * 1/10 = 1
    printf("%08x %08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.8f\n", f1);
    printf("f2 = %10.8f\n\n", f2);

    f1 = 1E30;  \#10^{30}
    f2 = 1E-30;  \#10^{-30}
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
    \#10^{30} == \#10^{-30} + \#10^{-30}
    return 0;
}
```

$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes
Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - “Gaps” produced in representable numbers means we can lose precision, unlike ints
    - Some “simple fractions” have no exact representation (e.g., 0.2)
    - “Every operation gets a slightly wrong result”

- Floating point arithmetic not associative or distributive
  - Mathematically equivalent ways of writing an expression may compute different results

- **Never** test floating point values for equality!
- **Careful** when converting between ints and floats!
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)
**Roadmap**

**C:**

```c
int main(void) {
    car *c = malloc(sizeof(car));
    c->miles = 100;
    c->gals = 17;
    float mpg = get_mpg(c);
    free(c);
    return 0;
}
```

**Java:**

```java
public class Car {
    private int miles;
    private int gals;

    public Car() {
        this(100, 17);
    }

    public Car(int miles, int gals) {
        this.miles = miles;
        this.gals = gals;
    }

    public int getMiles() {
        return miles;
    }

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public int getGals() {
        return gals;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        return (float) (miles / gals);
    }
}
```

**Assembly language:**

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

**Machine code:**

```
0111010000011000
100011010000010000000010
100100111000010
1100000111111010000011111
```

**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:**

- Windows 8
- Mac
- Linux
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)?
HW Interface Affects Performance

**Source code**
Different applications or algorithms

**Compiler**
Perform optimizations, generate instructions

**Architecture**
Instruction set

**Hardware**
Different implementations

- Intel Pentium 4
- Intel Core 2
- Intel Core i7
- AMD Opteron
- AMD Athlon
- ARM Cortex-A53
- Apple A7

---

C Language

- Program A
- Program B
- Your program

- GCC
- Clang

x86-64

ARMv8 (AArch64/A64)
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? *instructions are data!*

- Registers
  - How many registers are there?
  - How wide are they? *size of a word*

- Memory
  - How do you specify a memory location? *different ways to build up an address*
# Mainstream ISAs

<table>
<thead>
<tr>
<th>Designer</th>
<th>Intel, AMD</th>
<th>ARM Holdings</th>
<th>MIPS Technologies, Inc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>16-bit, 32-bit and 64-bit</td>
<td>32-bit, 64-bit</td>
<td>64-bit (32→64)</td>
</tr>
<tr>
<td>Introduced</td>
<td>1978 (16-bit), 1985 (32-bit), 2003 (64-bit)</td>
<td>1985; 31 years ago</td>
<td>1981; 35 years ago</td>
</tr>
<tr>
<td>Design</td>
<td>CISC</td>
<td>RISC</td>
<td>RISC</td>
</tr>
<tr>
<td>Type</td>
<td>Register-memory</td>
<td>Register-Register</td>
<td>Register-Register</td>
</tr>
<tr>
<td>Encoding</td>
<td>Variable (1 to 15 bytes)</td>
<td>AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[^1]</td>
<td>Fixed</td>
</tr>
<tr>
<td>Endianness</td>
<td>Little</td>
<td>Bi</td>
<td>Bi</td>
</tr>
</tbody>
</table>

[^1]: little as default

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

Smartphone-like devices (iPhone, iPad, Raspberry Pi) **ARM Instruction Set**

Digital home & networking equipment (Blu-ray, PlayStation 2) **MIPS Instruction Set**
# Intel x86 Evolution: Milestones

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 16-bit Intel processor. Basis for IBM PC &amp; DOS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1MB address space</td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 32 bit Intel processor, referred to as IA32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added “flat addressing,” capable of running Unix</td>
<td></td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First 64-bit Intel x86 processor, referred to as x86-64</td>
<td></td>
</tr>
<tr>
<td>Core 2</td>
<td>2006</td>
<td>291M</td>
<td>1060-3500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>First multi-core Intel processor</td>
<td></td>
</tr>
<tr>
<td>Core i7</td>
<td>2008</td>
<td>731M</td>
<td>1700-3900</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Four cores</td>
<td></td>
</tr>
</tbody>
</table>

"Heat death" (Moore's Law)
Intel x86 Processors

- **Machine Evolution**
  - 486  1989  1.9M
  - 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  2008  731M

- **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:
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 2nd 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 - modular**
Assembly Programmer’s View

- **Programmer-visible state**
  - **PC**: the Program Counter ($\%\text{rip}$ 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 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`

```
<table>
<thead>
<tr>
<th>text</th>
<th>C program (p1.c p2.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler (gcc -Og -S)</td>
</tr>
<tr>
<td>text</td>
<td>Asm program (p1.s p2.s)</td>
</tr>
<tr>
<td></td>
<td>Assembler (gcc or as)</td>
</tr>
<tr>
<td>binary</td>
<td>Object program (p1.o p2.o)</td>
</tr>
<tr>
<td></td>
<td>Linker (gcc or ld)</td>
</tr>
<tr>
<td>binary</td>
<td>Executable program (p)</td>
</tr>
<tr>
<td></td>
<td>Loader</td>
</tr>
<tr>
<td></td>
<td>Static libraries (.a)</td>
</tr>
</tbody>
</table>
```
Compiling Into Assembly

- **C Code (sum.c)**

```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](https://godbolt.org/g/pQUhIZ))

```assembly
sumstore(long, long, long*):
    addq %rdi, %rsi
    movq %rsi, (%rdx)
    ret
```

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

- **C Code**
  - Store value \( t \) where designated by \( dest \)

- **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]

- **Object Code**
  - 3-byte instruction (in hex)
  - Stored at address \( 0x40059e \)
Object Code

- **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

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
Disassembling Object Code

Disassembled:

```
00000000000400536 <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 0xc3 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:      55             push   %ebp
30001001:      8b ec mov %esp,%ebp
30001003:      6a ff         push   $0xffffffff
30001005:      68 90 10 00 30 push   $0x30001090
3000100a:      68 91 dc 4c 30 push   $0x304cdc91

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
More details for the curious. We won’t be using or testing you on any of these extras in 351.

- 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)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Round Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 + \frac{3}{32}$</td>
<td>$10.00011_2$</td>
<td>$10.00_2$</td>
<td>($&lt;1/2$—down)</td>
<td>2</td>
</tr>
<tr>
<td>$2 + \frac{3}{16}$</td>
<td>$10.00110_2$</td>
<td>$10.01_2$</td>
<td>($&gt;1/2$—up)</td>
<td>$2 \frac{1}{4}$</td>
</tr>
<tr>
<td>$2 + \frac{7}{8}$</td>
<td>$10.11100_2$</td>
<td>$11.00_2$</td>
<td>( 1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>$2 + \frac{5}{8}$</td>
<td>$10.10100_2$</td>
<td>$10.10_2$</td>
<td>( 1/2—down)</td>
<td>$2 \frac{1}{2}$</td>
</tr>
</tbody>
</table>
Floating Point Puzzles

For each of the following C expressions, either:

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

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

Assume neither \( d \) nor \( f \) is NaN

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