CS 33 Week 5

Section 1G, Spring 2015
Professor Eggert (TA: Eric Kim)
v1.0

Announcements

- Lab 2 due yesterday (4/30)
- HW 4 out! (Due: 5/8)
- Lab 3 out! (Due: 5/13)
  - Stack smashing!
    - "Smashing" Lab

(couldn't help myself)

MT 1 Grades Up

- Midterm 1 grades are up
    - `Mean: 43 Median: 42 Std: 13`
        - `242 students`
- Pretty good bell curve
- Check that points add up correctly
    - Verify with my.ucla.edu

This Week

- MT 1
- Floating Point
- Program Optimizations/Performance

Floating Point

- So far, have worked with integer data types
  - signed: two's complement
  - unsigned
- Integers: 0, 1, 42, -101, 9001

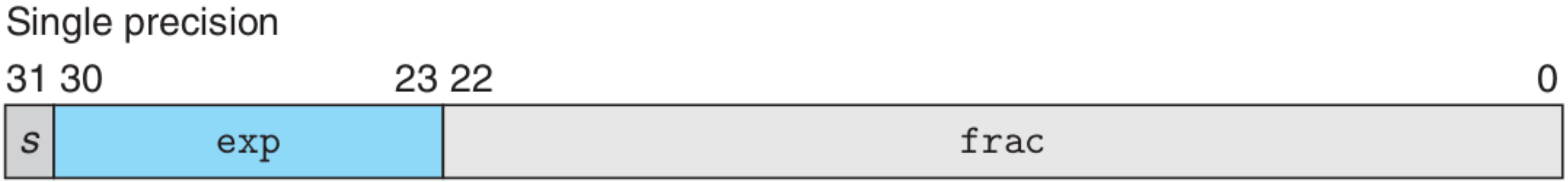
How to represent a non-integer, like 0.5?

Answer: Floating Point Representation!

Floating Point (IEEE 754)

- Goal: Represent **rational** numbers with a **fixed** number of bits
  - float: 32 bits "single" precision
  - double: 64 bits "double" precision

Floating Point: Bits of Me (Single)

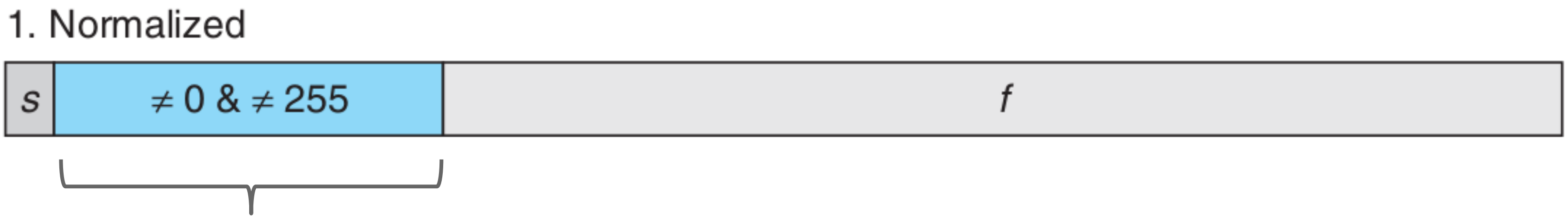


Sign bit	"Exponent" Field	"Fraction" Field
1 bit	8 bits	23 bits

Two Types of FP

- There are two different "types" of a floating point number
- Case 1: "Normalized"
  - Common case. Represent large and moderately-small values.
- Case 2: "Denormalized"
  - Represent very small values (close to 0).

Case 1: Normalized



Exp is not all 0's or all 1's

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

s = sign bit

M = 1 + f

E = e - Bias

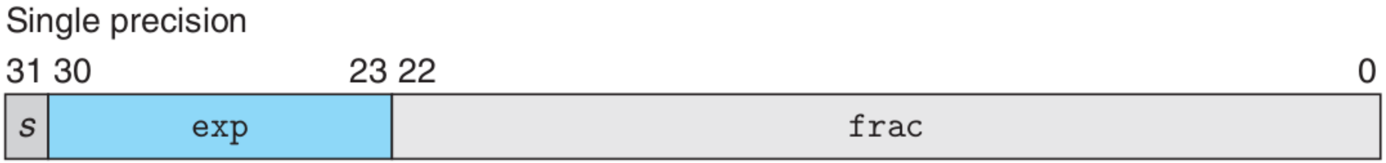
Bias = $2^{k-1} - 1$ = { 127 for single; 1023 for double }

k is # bits in exp

Case 1: Normalized

Bias = $2^{k-1} - 1$ = { 127 for single; 1023 for double }

k is # bits in exp




```
0100 0010 0010 1000 0000 0000 0000 0000
```

What is V?

exp = 100 0010 0 = 0x84 = 8*16 + 4 = 132

f = 010 1000 0000 0000 0000 0000

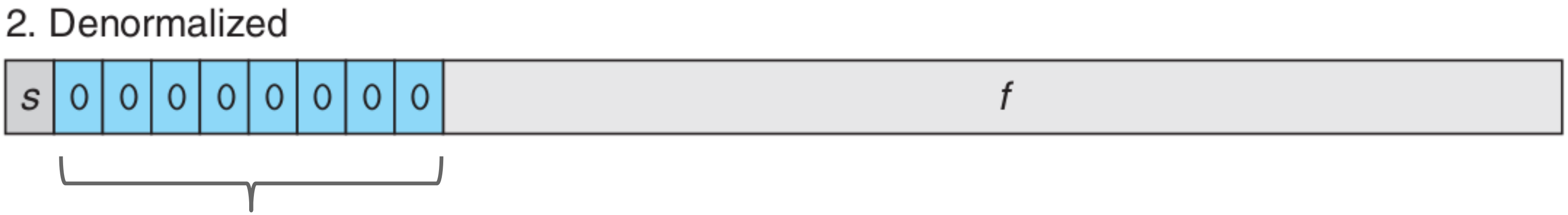
= 0*2^(-1) + 1*2^(-2) + 0*2^(-3) + 1*2^(-4) = 0.3125

V = (-1)^0 * (1 + 0.3125) * 2^(132 - 127) = **42.0**

s = sign bit
M = 1 + f
E = e - Bias

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

Case 2: Denormalized



Exp is all 0's or all 1's

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

s = sign bit
M = f
E = 1 - Bias

Bias = $2^{k-1} - 1$ = { 127 for single / 1023 for double }

k is # bits in exp

Case 2: Denormalized

Bias = $2^{k-1} - 1$ = { 127 for single; 1023 for double }

k is # bits in exp

1000 0000 0010 1100 0000 0000 0000 0000

What is V?

exp = 000 0000 0 = 0

f = 010 1100 0000 0000 0000 0000

= 1*2^(-2) + 1*2^(-4) + 1*2^(-5) = 0.34375

V = (-1)^1 * (0.34375) * 2^(1-127)

= **-4.0407618309516613e-39**

s = sign bit

M = f

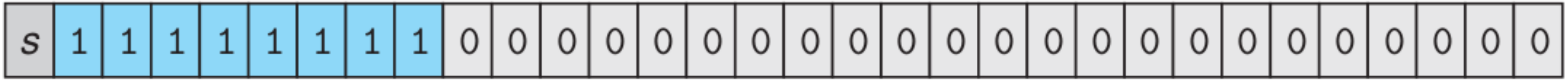
E = 1 - Bias

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

Case 3: Special Values

- Represent infinity, NaN via certain bit patterns

3a. Infinity



3b. NaN

Note: +Inf and -Inf are different (sign bit).

Case 3: Special Values

- Can represent 0.0 in two ways!
  - All bits 0, and sign bit is 1: -0.0
  - All bits 0, and sign bit is 0: +0.0
- -0.0 and +0.0 are *usually* the same

```
But: 1.f / 0.f -> +Inf
     1.f / -0.f -> -Inf
```

Example: Largest/Smallest

- Suppose we use an 8-bit floating-point format. There are 4 exponent bits, and 3 fraction bits.

What is the bias?

2^(4-1)-1 = 7

What is the smallest/largest positive value?

Smallest: 0 0000 001
Largest: 0 1110 111

How to represent 1.0?

0 0111 000

Rounding

- Floating point still can't represent every rational number exactly
  - Why? Finite number of bits (32, 64)
- IEEE standard defines several **rounding modes**

Four Rounding Modes

Mode	\$1.40	\$1.60	\$1.50	\$2.50	\$–1.50
Round-to-even	\$1	\$2	\$2	\$2	\$–2
Round-toward-zero	\$1	\$1	\$1	\$2	\$–1
Round-down	\$1	\$1	\$1	\$2	\$–2
Round-up	\$2	\$2	\$2	\$3	\$–1

Figure 2.36 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

FP Operations

- After every operation on two floating point values, a round is performed.
  - Ex: (f+g) => round(f+g)

FP: Addition

- Addition commutes correctly
  - `(f+g) = (g+f)`
- Addition is generally *not* associative
  - `(3.14 + 1e10)-1e10 = 0.0`
  - `3.14 + (1e10-1e10) = 3.14`

FP: Multiplication

- Generally not associative
  - `(1e20*1e20)*1e-20 = +Inf`
  - `1e20*(1e20*1e-20) = 1e20`
- Does not distribute over addition
  - `1e20*(1e20 - 1e20) = 0`
  - `1e20*1e20 - 1e20*1e20 = NaN`

Exercise

Practice Problem 2.53

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0:

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
```

You cannot use any include files (such as `math.h`), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

Exercise

We assume that the value `1e400` overflows to infinity.

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

C FP: Casting Rules

- In C, exists float and double data types
  - Can cast to/from float types to integer types.
  - Can lose information due to rounding/truncation!

```
#include <stdio.h>
int main() {
    float v = 42.675;
    int foo = (int) v;
    printf("foo is: %d\n", foo);
    return 0;
}
```

```
$ gcc -o code code.c
$ ./code
foo is: 42
```

Casting Rules Quiz

	Exact conversion?	Can overflow/underflow?
int -> float		
int -> double		
float -> double		
double -> float		
double -> int		
float -> int		

Casting Rules Quiz

	Exact conversion?	Can overflow?
int -> float	No! Float can't repr very large ints.	No
int -> double	Yes	No
float -> double	Yes	No
double -> float	No	Yes
double -> int	No: rounded toward zero (1.99 -> 1, -1.99 -> -1)	Yes
float -> int	No: rounded toward zero	Yes

Example: int vs float

Can you find an int that can't be repr by a float?

Can you find an int that can't be repr by a double?

Midterm 1: Post mortem

9 Questions Total

In my opinion, by difficulty:

- Easy/Medium: 1, 3, 6, 8, 9
- Medium-Hard: 2, 4, 7
- Tricky: 5

MT1: Q1 (leal)

1 (12 minutes). Which integer constants can a single x86 leal instruction multiply an arbitrary integer N by? The idea is that one puts N into a register, executes the leal instruction, and the bottom 32 bits of N*K will be put into some other register, where K is a constant. For which values of K can this be done? For each such value, show an leal instruction that implement that value.

MT1: Q1 (leal)

N*1 = lea (, N, 1)
N*2 = lea (, N, 2)
N*4 = lea (, N, 4)
N*8 = lea (, N, 8)

N*2 = lea (N, N, 1)
N*3 = lea (N, N, 2)
N*5 = lea (N, N, 4)
N*9 = lea (N, N, 8)

=> {1, 2, 3, 4, 5, 8, 9}

7 possibilities

MT1: Q2 (byte reversal)

2 (12 minutes). On the x86-64, what's the fastest way to reverse each 8-bit byte in a 64-bit unsigned integer? For example, given the input integer 0x0123456789abcdef, we want to compute 0x80c4a2e691d5b3f7; this is because 0x01 is binary 00000001 and reversing it yields binary 10000000 which is 0x80, and similarly the bit-reverse of 0x23 is 0xc4, and so forth until the bit-reverse of 0xef is 0xf7. Write the code in C, and estimate how many machine instructions will be generated (justify your estimate).

MT1: Q2 (byte reversal)

```
long long reverseBits(long long x) {
  unsigned long long m1 = 0xF0F0F0F0F0F0F0F0;
  unsigned long long m1_2 = 0x0F0F0F0F0F0F0F0F;
  unsigned long long m2 = 0xCCCCCCCCCCCCCCCC;
  unsigned long long m2_2 = 0x3333333333333333;
  unsigned long long m3 = 0xAAAAAAAAAAAAAAAA;
  unsigned long long m3_2 = 0x5555555555555555;
  unsigned long long l = x;
  l = ((l&m1) >> 4) | ((l&m1_2) << 4);
  l = ((l&m2) >> 2) | ((l&m2_2) << 2);
  l = ((l&m3) >> 1) | ((l&m3_2) << 1);
}
```

This requires 6 AND's, 6 shifts, and 3 OR's for a total of 15 instructions.

MT1: Q2 (byte reversal)

```
long long reverseBits(long long x) {
  unsigned long long m1 = 0xF0F0F0F0F0F0F0F0;
  unsigned long long m2 = 0xCCCCCCCCCCCCCCCC;
  unsigned long long m3 = 0xAAAAAAAAAAAAAAAA;
  unsigned long long l = x;

  l = (l&m1)>>4 | (l<<4)&m1;
  l = (l&m2)>>2 | (l<<2)&m2;
  l = (l&m3)>>1 | (l<<1)&m3;
  return l;
}
```

Alternate solution

MT1: Q3 (get eip)

3 (12 minutes). On the x86, there is no 'pushl %eip' instruction. Suppose you want to push the instruction pointer onto the stack anyway. What's the best way to do it? If your method takes three instructions A, B, C, the value pushed onto the stack should be the address of D, the next instruction after C.

MT1: Q3 (get eip)

```
    call foo
foo:
    … // here, top of stack is eip
```

MT1: Q4 (gdb + fin)

4 (12 minutes). Explain two different methods that GDB can implement its 'fin' command (which finishes execution of the current function), one method with hardware breakpoints and one without. For each method, say what happens if the current function calls another function via tail recursion.

MT1: Q4 (gdb + fin)

Idea: Set a breakpoint at the saved return address `addr`

Hardware (x86): Move `addr` into one of the debug registers DR0 - DR3. When the processor does the return, and is about to execute addr, hardware will throw a debug exception.

MT1: Q4 (gdb + fin)

Software: Modify gdb to replace the instruction at the saved return address with an `INT3` (debug interrupt instruction).

If the user wishes to continue executing after the `INT3` breakpoint is triggered, gdb can "restore" the overwritten instruction.

MT1: Q4 (gdb + fin)

For both methods: tail-recursion will still work, since we are putting a breakpoint at the caller-saved return address.

MT1: Q5 (compiler bug)

Consider the following C functions and their translations to x86 code.

```
int f (int *p, long *q) {
  ++*p;
  ++*q;
  return *p;
}
int g (int *p, char *q) {
  ++*p;
  ++*q;
  return *p;
}
```

```
f:  movl  4(%esp), %ecx
    movl  8(%esp), %edx
    movl  (%ecx), %eax
    addl  $1, %eax
    movl  %eax, (%ecx)
    addl  $1, (%edx)
    ret

g:  movl  4(%esp), %ecx
    movl  8(%esp), %edx
    movl  (%ecx), %eax
    addb  $1, (%edx)
    addl  $1, %eax
    movl  %eax, (%ecx)
    ret
```

There's a compiler bug: one of these functions is translated incorrectly, and the other one is OK. Identify the bug, and explain why the other function is translated correctly even though one might naively think that its translation has a similar bug.

MT1: Q5 (compiler bug)

Key: Pointer aliasing bug.

Aliasing in g is the bug. In g, q can point to a byte within the int of p. Thus, ++*p can affect the value of *q. However, g's assembly code adds 1 to q before adding 1 to p. To correct this, the instructions should add 1 to %eax first, then movl %eax, (%ecx), then addb $1, (%edx), then movl (%ecx), % eax. f is correct because integer and long are stored in different memory address so that order doesn't matter.

MT1: Q5 (compiler bug)

Consider the following C functions and their translations to x86 code.

```
int f (int *p, long *q) {
  ++*p;
  ++*q;
  return *p;
}
```

```
f:  movl  4(%esp), %ecx
    movl  8(%esp), %edx
    movl  (%ecx), %eax
    addl  $1, %eax
    movl  %eax, (%ecx)
    addl  $1, (%edx)
    ret
```

```
int g (int *p, char *q) {
  ++*p;
  ++*q;
  return *p;
}
```

```
g:  movl  4(%esp), %ecx
    movl  8(%esp), %edx
    movl  (%ecx), %eax
    addb  $1, (%edx)
    addl  $1, %eax
    movl  %eax, (%ecx)
    ret
```

There's a compiler bug: one of these functions is translated incorrectly, and the other one is OK. Identify the bug, and explain why the other function is translated correctly even though one might naively think that its translation has a similar bug.

```
g:  movl  4(%esp), %ecx
    movl  8(%esp), %edx
    movl  (%ecx), %eax
    addl  $1, %eax
    movl  %eax, (%ecx)
    addb  $1, (%edx)
    ret
```

MT1: Q6 (stack limits)

6 (12 minutes). Suppose we have allocated memory locations 0xffff0000 through 0xffffffff for the stack, and we are worried that our x86 program might overflow the stack. We decide to institute the ironclad rule that if a function ever attempts to grow the stack past the allocated bounds, the function immediately stops what it's doing and returns 0, thus shrinking the stack. Explain the problems you see in implementing this rule. Don't worry about the effects of this rule on the user program; worry only about implementing the rule correctly.

MT1: Q6 (stack limits)

Key: Need assistance from compiler to implement this. Before the compiler makes a decrement to the stack pointer, the compiler should issue checks to see if the decrement would go past 0xffff0000. If it does, then the function should return 0. Else, esp should be decremented as normal, and execution should resume as normal.

Some possible problems:

- Since the stack pointer has to be checked prior to every modification to esp, this will dramatically slow down programs.
- Wrap around: suppose we decrement esp by a value so large, that the esp wraps around. For instance, suppose esp=0xFFFF0004:

```
subl $0x7fffffff, %esp     // esp=0x7FFF0005
```

The compiler needs to carefully handle this case.

MT1: Q7 (x86 -> C)

7 (12 minutes). Give C source code that corresponds to the following x86-64 assembly language code. Explain briefly and at a high level what useful thing the function does.

```
sub:    movq    %rdi, %rdx
        subq    %rsi, %rdx
        xorq    %rdi, %rsi
        xorq    %rdi, %rdx
        movq    %rdx, %rax
        andq    %rsi, %rax
        shrq    $63, %rax
        ret
```

MT1: Q7 (x86 -> C)

The function returns 1 if x-y overflows, x being %rdi and y being %rsi.

```
((x ^ y) & (x ^ (x - y))) >> 63
```

MT1: Q8 (C, x86 matchmaker)

a=D

b=A

c=E

d=L

e=J

f=C

g=B

h=H

i=K

j=I

k=F

l=G

MT1: Q9 (ack!)

Consider the following program:

```
1 unsigned ack (unsigned m, unsigned n) {
2   if (m == 0)
3     return n + 1;
4   if (n == 0)
5     return ack (m - 1, 1);
6   return ack (m - 1,
7               ack (m,
8                    n - 1));
9 }
```

For each instruction in the implementation, identify the corresponding source-code line number. If an instruction corresponds to two or more source-code line numbers, write them all down and explain.

MT1: Q9 (ack!)

Implementation Line	Source Line
1	N/A
2	1
3	1
4	1
5	1
6	2
7	2
8	5&6
9	4
10	N/A
11	2
12	3
13	6
14	2
15	N/A
16	6
17	N/A
18	4
19	6
20	5
21	7
22	8
23	8
24	7
25	7
26	7
27	2
28	6
29	6
30	N/A
31	3/9
32	3
33	3/9
34	3/9