

# ICS 第 0x2 次小班课

2021-09-28

# Roadmap

## ■ Review

- Chapter 2
- Sect 3.1-3.6

## ■ In-class Exercise

## ■ Datalab

## ■ Next class

## Review #1/4

Be free to ask questions or challenge!

# Summarize

- 重要观点：机器级的数据表示只是对数学对象的近似
  - 表示范围的局限
    - 不同的整数类型
      - 原码/补码表示,
      - 类型转换惯例（在数值上如何变化，在位级表示上如何变化）
  - 精度
    - 运算溢出
    - 舍入（浮点运算或向浮点类型转换：向偶数舍入）
    - 浮点转整数：向零舍入
  - 不同于真实整数的运算律
    - 整数：模运算
    - 浮点：舍入或者溢出，Special value ( $\infty$ , NaN)

## Exercise 2

	Description	True?
(1)	对于任意的单精度浮点数 $a$ 和 $b$ ，如果 $a > b$ ，那么 $a + 1 > b$	Y N
(2)	对于任意的单精度浮点数 $a$ 和 $b$ ，如果 $a > b$ ，那么 $a + b > b + b$	Y N
(3)	对于任意的双精度浮点数 $d$ ，如果 $d < 0$ ，那么 $d * d > 0$	Y N
(4)	对于任意的双精度浮点数 $d$ ，如果 $d < 0$ ，那么 $d * 2 < 0$	Y N
(5)	对于任意的双精度浮点数 $d$ ， $d == d$	Y N
(6)	将 <code>float</code> 转换成 <code>int</code> 时，既有可能造成舍入，又有可能造成溢出	Y N

(1) 正确

(3)  $d$  取最大的非规格化负数

(5)  $\text{NaN} \neq \text{NaN}$

(2) 取  $a = \text{INF}$   $b = \text{FLT\_MAX}$

(4) 正确

(6) 正确

# Exercise 1





假设浮点数格式A为1符号+3阶码+4小数，浮点数格式B为1符号+4阶码+3小数。回答下列问题。

- 格式A中有多少个二进制表示对应于正无穷大？ -> 只有一个01110000
- 考虑能精确表示的实数的最大绝对值。A比B大还是比B小，还是两者一样？  
对于A格式，01101111表示了 $31/16 * 2^3 = 31/2 = 15.5$ ，  
对于B格式，01110111表示了 $15/8 * 2^7 = 240$ ，因此B大。
- 考虑能精确表示的实数的最小非零绝对值。A比B大还是比B小，还是两者一样？  
对于A格式，00000001表示了 $1/16 * 2^{(-2)} = 1/64$ ，  
对于B格式，00000001表示了 $1/8 * 2^{(-6)} = 1/512$ ，因此A大。
- 考虑能精确表示的实数的个数。A比B多还是比B少，还是两者一样？  
-> A能精确表达的非负数个数为 $7 * 16 = 112$ ，B能精确表达的非负数个数为 $15 * 8 = 120$ ，因此B能精确表达的实数更多。实际上，A格式表示NaN的数比B格式多。

# Exemplify #1/3 *Counter-intuition*

1. 在 x86-64 机上假设有以下程序片段，请分别判断表格内的表达式是否恒真。若否，请举出至少一则反例。

```
1 int x = foo();  
2 int y = bar();  
3 unsigned ux = x;  
4 unsigned uy = y;
```

Expression	Always True?	Counterexample
 $x / 8 == x \gg 3$	F	
 $ux / 8 == ux \gg 3$	T	
$x * x \geq 0$		
$ux * ux \geq 0$		
 $x \leq 0 \mid\mid -x < 0$	T	
 $x \geq 0 \mid\mid -x > 0$	F	
$!x \mid\mid (x \mid -x) \gg 31 == -1$		
$ux > -1$		

# Exemplify #2/3

## Counter-intuition

7. 以下程序的输出结果最接近选项中的哪个常数？已知该机器上 `int` 为 4 字节，程序采用 `-Og` 级别优化，且数学上成立  $1 - 1/3 + 1/5 - 1/7 + \dots = \pi/4$ .

```
1 #include <stdio.h>
2
3 int main()
4 {
5     double pi = 0.0;
6     int k;
7     for(k = 0; k >= 0; ++k)
8         pi += (k & 1 ? -1 : 1) / (double)(2 * k + 1);
9     pi *= 4;
10    printf("%f\n", pi);
11 }
```

A.0

B. $\pi/2$

C. $\pi$

D. $2\pi$

**Warning:** `-Og` 表明了这个程序多么容易引发 Undefined Behavior。因为它基于溢出来结束循环，而这本身就是 UB 的。我们极不推荐这种写法，它在较高的优化级别可能产生意想不到的行为。例如 `-O3` 优化下，汇编代码中 `for` 循环会直接被“优化”为死循环。

汇编的知识将在第三章介绍，而优化方面的内容在第五章介绍。



# Exemplify #3/3 Counter-intuition

1. 在遵守 IEEE 754 标准的一台机器上声明如下三个变量 `double f, g, h;` 以及一个函数原型 `int foo();` 判断以下表达式在给定条件下是否恒真。

Condition	Expression	Always True?
<code>f &gt; g</code>	<code>f + 1 &gt; g + 1</code> <small>g 最大的负数 f 0</small>	<b>F</b>
<code>f &gt; g &amp;&amp; g &gt; 1</code>	<code>f - 1 &gt; g - 1</code> <small>f 2^53+4 g 2^53+6</small>	<b>F</b>
<code>f = foo(); g = foo(); h = foo();</code>	<code>(f + g) + h == f + (g + h)</code> <small>如果不保证 f, g, h 是从 int 转的, false</small>	<b>T</b>
<code>f != 0.0</code>	<code>f / f * f == f</code> <small>f inf</small>	<b>F</b>
<code>f != 0.0</code>	<code>f * f / f == f</code> <small>f inf</small>	<b>F</b>

- 注意单调性的表述。遇到断言声称 `>` 而非 `>=` 请谨慎对待！
  - Aside: 浮点的电路级大小比较 & 为什么日常比较浮点数大小都用 `eps` (即  $\epsilon$ )
- Special value ( $\infty$ , NaN) 是否需要考虑

2. 在遵守 IEEE 754 标准的一台 64 位机器上有如下源程序，它的作用是对任意浮点数  $x$ ，返回  $\lfloor \log_2 x \rfloor$ ，其中  $x$  表示  $x$  的数值。程序中使用了一个函数原型，它的实现能够正确地对每个非零的无符号长整型返回位表示中最高的 1 所在的位数。位数从低到高编号为  $0, 1, \dots, 63$ 。现在，你的任务是在空格处填入合适的常数，使得程序正常工作。

```
1  #include <stdio.h>
2
3  #define UNDEF      0x7FFFFFFFFFFFFFFFL
4  #define MASK      0x7FF0000000000000L
5
6  /* A function returns the highest digit of 1
7   in the bit-representation of x,
8   the lowest bit is 0 and the highest bit is 63 */
9  long Highest1(unsigned long x);
10
11 /* The function you need to fulfill */
12 long Log2Floor(double x){
13     long y = *(long *)&x;
14     long exp = y & MASK;
15
16     /* negative or special value */
17     if(x <= 0.0 || exp == MASK) return UNDEF;
18     else if(exp) /* normalized value */
19         return (exp >> 52) - 1023;
20     else /*denormalized value */
21         return -1074 + Highest1(y);
22 }
23
24 // other codes
```

3. Alice 刷完树洞后开始漫不经心地写起第 2 题。Alice 突然很有兴趣想测试一下这个程序是否至少对于整数都是正确的，于是就写了以下程序。当顺利编译并和第 2 题中的源程序链接后，Alice 发现程序始终没有结果输出，也没有结束。请帮助 Alice 检查这是为什么。该机器上 `float` 是 1 + 8 + 23 位的。

```
1  /* Alice's version */
2
3  #include <stdio.h>
4  #include <math.h>
5  #include <limits.h>
6
7  // some declarations
8
9  /* test Log2Floor */
10 int main(){
11     int float i = 1;
12     for(; i < INT_MAX; i += 1){
13         /* correct answer, this line contains no bug! */
14         long std_ans = floor(log(i) / log(2));
15
16         if(Log2Floor(i) != std_ans)
17             printf("error\n");
18     }
19     return 0;
20 }
```

Alice 很感谢你帮助 debug，但是 Alice 还希望你帮助他/她修改这个程序使之正常工作。请做尽可能少的改动完成这一点。

# 常见术语 Terms of Today

## ■ 通用寄存器 General-Purpose Register

- What does it mean by *general-purpose*?

<https://stackoverflow.com/questions/36529449/why-are-rbp-and-rsp-called-general-purpose-registers>

## ■ 64位 / 32位 64-bit / 32-bit

- On which level are they different?
- Why many 32-bit softwares can not run on a 64-bit machine?
  - Datum Format : especially size of int and the pointer
  - Linkage problem: Not bother with it now. Till tomorrow.

## ■ 惯例 Conventions

- Routines which were long entrenched and were normal to accept.
- Some conventions have to be obeyed (e.g. %rsp & pushq / popq), while others do not (e.g. %rbp as the base pointer).
- Every general-purpose register except %r8-%r15 got its name from some old-time conventions somehow.

# **x86-64 Assembly**

## ■ x86-64惯例

- 仅生成4字节指令的数字会把高位4个字节置为0

## ■ 操作数格式

- 立即数与存储器的绝对寻址的区别 (\$)
- 内存访问时基址和变址寄存器都是64位寄存器，比例因子必须是1,2,4,8

## ■ 指令

- `movq` vs. `movabsq`
- `cqto` vs. `cltq`
- 分支指令仅 `jmp` 允许间接跳转
  - `jmp .Label`
  - `jmp *Operand`      e.g. `jmp *%rax`      `jmp *(%rax)`
- `cmov` vs. `mov`      传送地址，传送长度

## ■ 条件码

- `leaq` 不设置CC，不真正访问内存
- `setl` 不是设置一个 long !
- `jae` 与 `jns` 虽然有/无符号数运算电路一样，对条件码的解释却不一样

## Exercise 3

已知float的格式为1符号+8阶码+23小数，有下列代码，其运行结果为？

```
int x = 33554466; // 2^25 + 34
int y = x + 8;
for ( ; x < y; x++) {
    float f = x;
    printf("%d ", x - (int)f);
}
```

2 -1 0 1 -2 -1 0 1

注意 Round to Even 规则！

# Exercise 4

假设%rax、%rbx的初始值都是0。根据下列一段汇编代码，写出每执行一步后两个寄存器的值。

	%rax	%rbx
movabsq \$0x0123456789ABCDEF, %rax		
---->	0x0123456789ABCDEF	0x0000000000000000
movw %ax, %bx		
---->	0x0123456789ABCDEF	(1) ????????????????
movswq %bx, %rbx		
---->	0x0123456789ABCDEF	(2) ????????????????
movl %ebx, %eax		
---->	(3) ????????????????	(2) ????????????????
movabsq \$0x123456789ABCDEF, %rax		
---->	0x0123456789ABCDEF	(2) ????????????????
cltq		
---->	(4) ????????????????	(2) ????????????????

(1) 0x0000000000000CDEF

(2) 0xFFFFFFFFFFFFCDEF

(3) 0x00000000FFFCDEF

(4) 0xFFFFFFFF89ABCDEF



# Exercise 5

下列操作不等价的是?

- A. `movzbq` 和 `movzbl`
- B. `movzwq` 和 `movzwl`
- C. `movl` 和 `movslq`
- D. `movslq %eax, %rax` 和 `cltq`

C

- A. B. `movzbl/movzwl` 生成了四字节, 把高位设为0;
- D. `cltq` 是对 `%eax` 的符号拓展;
- C. `movl` 和 `movzblq` 等价 (所以不需要 `movzblq` 这条指令)

# Exercise 6

func:

movl \$1, %eax

jmp .L2

.L4:

testb \$1, %sil

je .L3

imulq %rdi, %rax

.L3:

sarq %rsi

imulq %rdi, %rdi

.L2:

testq %rsi, %rsi

jg .L4

rep ret

*// a in %rdi, b in %rsi*

long func(long a, long b)

{

long ans = 1;

while ( b > 0 ) {

if ( b & 1 )

ans = ans \* a;

b = b >> 1;

a = a \* a;

}

return ans;

}

# Exercise 7

func:

```
    movq    %rsi, %rax
    testq   %rdi, %rdi
    jne     .L7
    rep ret
```

.L7:

```
    subq    $8, %rsp
    imulq   %rdi, %rax
    movq    %rax, %rsi
    subq    $1, %rdi
    call    func
    addq    $8, %rsp
    ret
```

```
long func(long n, long m)
```

```
{
```

```
    if ( !n )
```

```
        return m;
```

```
    return func ( n-1, n*m );
```

```
}
```

# Exercise 8

2. 以下对一个 C 语言片段给出了若干可能的 x86-64 汇编代码，请指出哪些是正确的汇编。

```
long rand();  
long tmp = rand();  
long a = tmp / 8, b = tmp % 8;
```

假设 tmp in %rax, a in %rdx, b in %rcx;且后续不再使用tmp。

leaq 7(%rax), %rdx	movq %rax, %rdx	cqto
testq %rax, %rax	sarq \$3, %rdx	movl \$8, %ecx
cmovns %rax, %rdx	leaq (,%rdx,8), %rcx	idivq %rcx
sarq \$3, %rdx	subq %rcx, %rax	movq %rdx, %rcx
leaq (,%rdx,8), %rcx	movq %rax, %rcx	movq %rax, %rdx
subq %rcx, %rax		
movq %rax, %rcx		
A	B	C

# Next Class

## ■ x86-64汇编如何表示复合数据结构和浮点操作

### ■ 基本要求

- 理解struct union语义的不同，数据对齐，理解结构体和联合体的结构信息是怎样在汇编中表现的。
- 数组的汇编级表示与访问，理解指针的汇编翻译，指针类型对汇编的影响。
- 浮点数指令不需记忆，简单识记过程管例，简单了解SSE, AVX 的历史，了解 SIMD 的概念。

### ■ 拓展(Not in Exams)：复合数据结构在过程中如何传递并返回。

## ■ 缓冲区溢出攻击

### ■ 基本：

- 理解各种攻击形式和他们的逻辑关系，例如ASLR vs. nop-sled  
Permission bits vs. ROP Overwrite vs. Canary etc.

### ■ 拓展(Absolutely Not in Exams)：SROP Attack

# Any questions?

**Labs:** 私は必ず帰ってくる!