Lesson 3
Machine Prog 1

ICS Seminar #9
张龄心
Sept 20, 2023

汇编与反汇编

- gcc –Og –S test.c

  编译器制备test.s汇编代码
- gcc –Og –c test.c

  编译器制备test.s汇编代码，然后汇编器制备二进制目标代码文件test.o
- objdump –d test.o

  反汇编器从.o文件得到汇编代码

数据格式

汇编的后缀

- 字节-字-双字-四字：**bwlq**

例：movb – movw – movl - movq

C声明	Intel数据类型	汇编代码后缀	大小(字节)
char	字节	b	1
short	字	w	2
int	双字	l	4
long	四字	q	8
char*	四字	q	8
float	单精度	s	4
double	双精度	l	8

图 3-1　C语言数据类型在 x86-64 中的大小。在 64 位机器中，指针长 8 字节

寄存器

- 参数(6个)：**dsdc89**

  %rdi - %rsi - %rdx - %rcx - %r8 - %r9
- 被调用者保存(6个)

  %rbp, %rsp, %r12, %r13, %r14, %r15

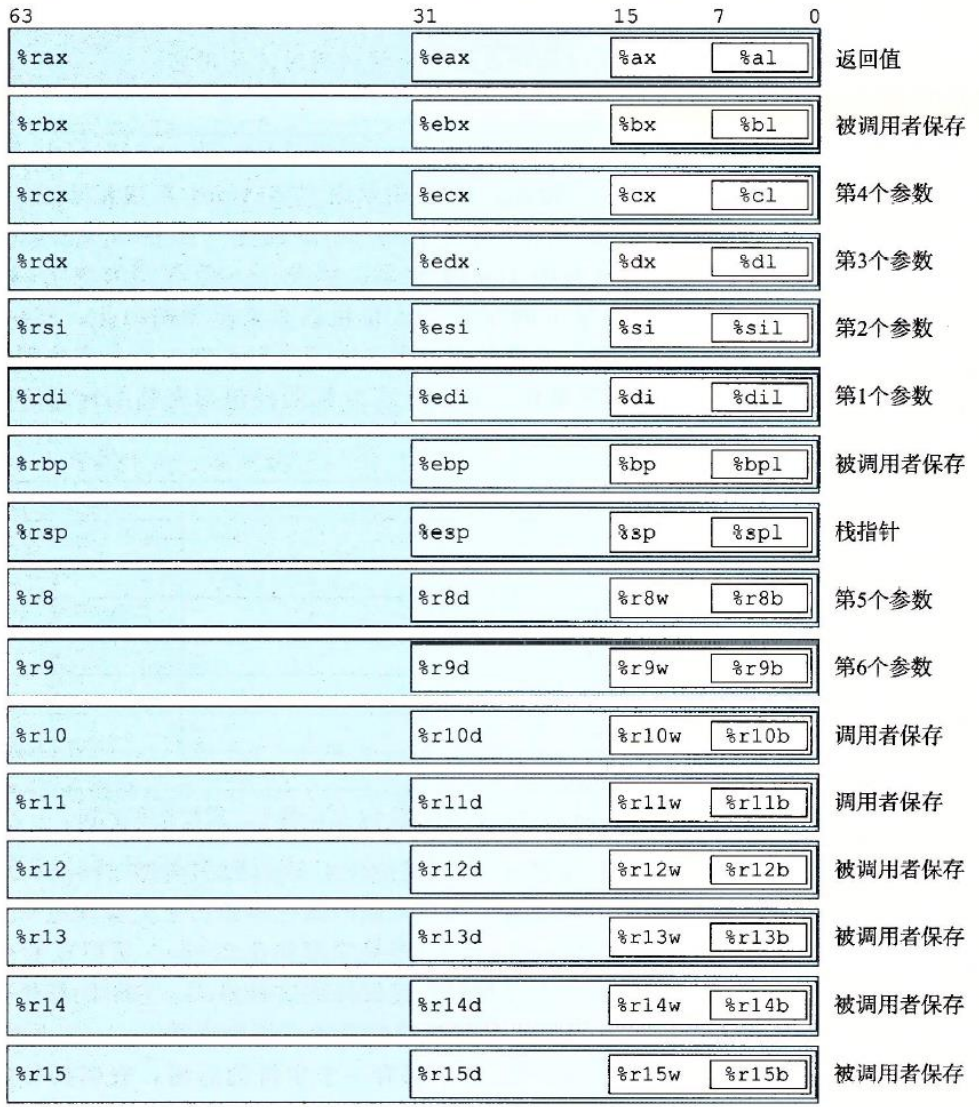
  调用者保存(2个)

  %r10, %r11
- 返回值%rax, 栈指针%rsp

*caller-saved & callee-saved

生成1~2byte的指令, 不改变寄存器中高位

生成4byte的指令, 将寄存器高位4byte置0



操作数

[addr]表示寻址: 拿出地址addr处存储的值 (addr可能是表达式)

类型	格式	操作数值	名称
立即数	$\$Imm$	Imm	立即数寻址
寄存器	r_a	$R[r_a]$	寄存器寻址
存储器	Imm	$M[Imm]$	绝对寻址
存储器	(r_a)	$M[R[r_a]]$	间接寻址
存储器	$Imm(r_b)$	$M[Imm+R[r_b]]$	（基址 + 偏移量）寻址
存储器	(r_b, r_i)	$M[R[r_b]+R[r_i]]$	变址寻址
存储器	$Imm(r_b, r_i)$	$M[Imm+R[r_b]+R[r_i]]$	变址寻址
存储器	$(,r_i, s)$	$M[R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(,r_i,s)$	$M[Imm+R[r_i] \cdot s]$	比例变址寻址
存储器	(r_b, r_i, s)	$M[R[r_b]+R[r_i] \cdot s]$	比例变址寻址
存储器	$Imm(r_b, r_i, s)$	$M[Imm+R[r_b]+R[r_i] \cdot s]$	比例变址寻址

只记这个就行

图 3-3 操作数格式。操作数可以表示立即数(常数)值、寄存器值或是来自内存的值。比例因子 s 必须是 1、2、4 或者 8

一些辨析

- %dil , %dl , %di ?
- 没有movzlq, movl就可以起到这个效果 (生成四字节时, 默认将高位设为0)
- cltq = 对%eax使用movslq (%eax符号扩展到%rax)

  - 作用: 编码更短; 另有cqto (四字->八字, %rax符号扩展到%rdx+%rax)
- leaq: 在32位下, 是leal (早期往年题可能见到); 目标位置必须是寄存器
  - 结合内存寻址：lea 不会访问内存，因此可以有非法地址
  - 用 lea 计算加法和乘
    - 加载有效地址x 编码更短的简单算术指令√
    - 简洁（三元运算）

例题

4. 在下列的 x86-64 汇编代码中，错误的是：

A. `movq %rax, (%rsp)`

B. `movl $0xFF, (%ebx)`

C. `movsbl (%rdi), %eax`

D. `leaq (%rdx, 1), %rdx`

例题

4. 在下列的 x86-64 汇编代码中，错误的是：

A. `movq %rax, (%rsp)`

B. `movl $0xFF, (%ebx)`

C. `movsbl (%rdi), %eax`

D. `leaq (%rdx, 1), %rdx`

BD

B: 64位机器, 地址应为8字节

D: 错误的操作数格式

例题

3．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

例题

3．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

注意: 生成4字节的指令, 都会将高位设为0

【答】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 movzlq 等价（所以不需要 movzlq 这条指令）

例题

1. 判断下列 x86-64 ATT 操作数格式是否合法。

(1) (　　) 8(%rax, ,2)

(2) (　　) $30(%rax,%rax,2)

(3) (　　) 0x30

(4) (　　) 13(,%rdi,4)

(5) (　　) (%rsi,%rdi,6)

(6) (　　) %ecx

(7) (　　) (%ecx)

(8)★(　　) (%rbp,%rsp)

例题

1．判断下列 x86-64 ATT 操作数格式是否合法。

(1) (　　) 8(%rax, ,2)

(2) (　　) $30(%rax,%rax,2)

(3) (　　) 0x30

(4) (　　) 13(,%rdi,4)

(5) (　　) (%rsi,%rdi,6)

(6) (　　) %ecx

(7) (　　) (%ecx)

(8)★(　　) (%rbp,%rsp)

1-4 F F T T

5-8 F T F F

【答】1 不正确；2 不正确，$；3 正确，是内存地址 0x30；4 正确；5 不正确，缩放比例只能是 1，2，4，8；6 正确；7 不正确，x86-64 不允许将除了 64 位寄存器以外的寄存器作为寻址模式基地址；8 不正确，%rsp 不能作为操作数（参考 Intel 手册 Vol.1　3-23 与 Vol.2A　2-7）。

条件码

CF = carrying flag, 最近操作使最高位产生进位

ZF = zero flag, 最近操作结果为0

SF = sign flag, 最近操作结果为负

OF = overflow flag, 最近操作产生溢出

- 算术运算: 直接看结果即可
- 逻辑运算(如XOR): 设置CF=0, OF=0, ZF/SF看结果
- 位移运算(如SAR): CF为最后一个移出的位, OF=0, ZF/SF看结果
- 自增自减 (INC和DEC): 不改变CF, ZF/SF/OF看结果
    - 不改变CF的原因: 硬件问题, 目的主要是让指针(地址)进行自增/自减

条件码

- 有符号数: setg, setl -> 用SF和OF
- 无符号数: seta, setb -> 用CF

- 只需要记两个
  - setl = SF^OF (结果小于0且不溢出 / 大于0但溢出了)

    -> setle = (SF^OF) | ZF
  - setb = CF (无符号运算溢出)

    -> setbe = CF | ZF

条件码

- set 指令

  对象: 某寄存器最低字节

  或 内存中的一个字节

- jmp 指令
  - 条件跳转只能是直接的
  - 间接跳转: %rsp(下一指令地址)+偏移量 = 目标地址
- cmov: 可以不带后缀; 仅支持2/4/8 bytes, 不支持1字节传输
- cmp = sub ; test = and

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

例题

3. 在如下代码段的跳转指令中，目的地址是：

```
400020: 74 F0     je _________
400022: 5d     pop %rbp
```

A. 400010　　B. 400012　　C. 400110　　D. 400112

例题

3. 在如下代码段的跳转指令中，目的地址是：

```
400020: 74 F0     je _________
400022: 5d     pop %rbp
```

A. 400010　　B. 400012　　C. 400110　　D. 400112

B

本来%rsp指向0x400022

现在需要加上偏移量0xF0=-16 = -0x10

例题

(　　)2.下列关于数据传送指令的说法中,正确的是____.

A.常规的movq指令只能以表示为32位补码数字的立即数作为源操作数(注：这里的"只能"强调的是不能以64位数作为源操作数),然后把这个值零扩展到64位的值,放到目的位置.movabsq指令能够以任意64位立即数值作为源操作数,并且只能以寄存器为目的.

B.在Imm(rb,ri,s)这一寻址模式中,s必须是1,2,4或者8,基址和变址寄存器必须是64位寄存器.

C.cqto指令不需要额外操作数,它的作用是把%eax符号扩展到%rax.

D.CPU执行指令"movl -1 %eax"后,%rax的值为0x00000000ffffffff.

例题

(　　)2. 下列关于数据传送指令的说法中,正确的是____.

A. 常规的movq指令只能以表示为32位补码数字的立即数作为源操作数(注：这里的"只能"强调的是不能以64位数作为源操作数),然后把这个值零扩展到64位的值,放到目的位置. movabsq指令能够以任意64位立即数值作为源操作数,并且只能以寄存器为目的.

B. 在Imm(rb,ri,s)这一寻址模式中,s必须是1,2,4或者8,基址和变址寄存器必须是64位寄存器.

C. cqto指令不需要额外操作数,它的作用是把%eax符号扩展到%rax.

D. CPU执行指令"movl -1 %eax"后,%rax的值为0x00000000ffffffff.

B(A符号扩展,C是cltq,D缺少$符)

例题

(　　)3.下列关于通用目的寄存器和条件码的说法中,正确的是____.

A.%rbx,%rbp,%r12-r15是被调用者保存寄存器,其他寄存器是调用者保存寄存器.

B.xorq %rax %rax可以用于清空%rax;cmpq %rax,%rax可以用于判断%rax的值是否为零.

C.调用一个过程时,最多可以传递6个整型参数,依次存储在(假设参数都是64位的)%rdi,%rsi,%rdx,%rcx,%r8,%r9中.当需要更多的参数时,调用者过程可以在自己的栈帧里按照地址由低到高的顺序依次存储这些参数.

D.leaq指令不改变条件码;逻辑操作(如XOR)会将进位标志和溢出标志都设置为零;移位操作会把溢出标志设置为最后一个被移出(shift out)的位,而把进位标志设置为零.

例题

(　　)3.下列关于通用目的寄存器和条件码的说法中,正确的是____.

A.%rbx,%rbp,%r12-r15是被调用者保存寄存器,其他寄存器是调用者保存寄存器.

B.xorq %rax %rax可以用于清空%rax;cmpq %rax,%rax可以用于判断%rax的值是否为零.

C.调用一个过程时,最多可以传递6个整型参数,依次存储在(假设参数都是64位的)%rdi,%rsi,%rdx,%rcx,%r8,%r9中.当需要更多的参数时,调用者过程可以在自己的栈帧里按照地址由低到高的顺序依次存储这些参数.

D.leaq指令不改变条件码;逻辑操作(如XOR)会将进位标志和溢出标志都设置为零;移位操作会把溢出标志设置为最后一个被移出(shift out)的位,而把进位标志设置为零.

3.C(A%rsp是例外,B可用于判断是否为零的是test,D移位操作设置进位条件码,但把溢出条件码设置为零)

例题

8. 假设某条 C 语言 switch 语句编译后产生了如下的汇编代码及跳转表：

```
    movl 8(%ebp), %eax                  .L7:
    subl $48, %eax                      .long .L3
    cmpl $8, %eax                       .long .L2
    ja .L2                              .long .L2
    jmp *.L7(, %eax, 4)                 .long .L5
                                        .long .L4
                                        .long .L5
                                        .long .L6
                                        .long .L2
                                        .long .L3
```

在源程序中，下面的哪些（个）标号出现过：

A. '2', '7'

B. 1

C. '3'

D. 5

例题

8. 假设某条 C 语言 switch 语句编译后产生了如下的汇编代码及跳转表：

```
    movl 8(%ebp), %eax                  .L7:
    subl $48, %eax                      .long .L3
    cmpl $8, %eax                       .long .L2
    ja .L2                              .long .L2
    jmp *.L7(, %eax, 4)                 .long .L5
                                        .long .L4
                                        .long .L5
                                        .long .L6
                                        .long .L2
                                        .long .L3
```

在源程序中，下面的哪些（个）标号出现过：

A. '2', '7'

B. 1

C. '3'

D. 5

选择 C. 标号的取值范围为'0' - , '1'、'2'、'7'、'9'、…等没有出现。

例题

9．对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

long f1(long a, long b) {
  return (++a > --b) ? a : b;
}
	Y N

long f2(long *a, long *b) {
  return (*a > *b) ? --(*a) : (*b)--;
}
	Y N

long f3(long *a, long *b) {
  return a ? *a : (b ? *b : 0);
}
	Y N

long f4(long a, long b) {
  return (a > b) ? a++ : ++b;
}
	Y N

例题

9．对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

long f1(long a, long b) {
  return (++a > --b) ? a : b;
}
	(Y) N

long f2(long *a, long *b) {
  return (*a > *b) ? --(*a) : (*b)--;
}
	Y (N)

long f3(long *a, long *b) {
  return a ? *a : (b ? *b : 0);
}
	Y (N)

long f4(long a, long b) {
  return (a > b) ? a++ : ++b;
}
	(Y) N

【答】f1 由于比较前计算出的 a 与 b 就是条件传送的目标，因此会被编译成条件传送；f2 由于比较结果会导致 a 与 b 指向的元素发生不同的改变，因此会被编译成条件跳转；f3 由于指针 a 可能无效，因此会被编译为条件跳转；f4 会被编译成条件传送，注意到 a 和 b 都是局部变量，return 的时候对 a 和 b 的操作都是没有用的。使用-O1 可以验证 gcc 的行为。

例题

(　　)4. 在下面的代码中,A和B是用#define定义的常数:

```
typedef struct {int x[A][B]; long y;} str1;
typedef struct {char array[B]; int t; short s[A]; long u;} str2;
void setVal(str1 *p, str2 *q) {
        long v1 = q->t;long v2 = q->u;
        p->y = v1+v2;
}
```

GCC为setVal产生下面的代码:

```
setVal:
movslq 8(%rsi), %rax
addq 32(%rsi), %rax
movq %rax, 184(%rdi)
ret
```

则A=____,B=____.

A. 8,6　　B.10,8　　C.10,5　　D. 9.5

例题

(　　)4. 在下面的代码中,A和B是用#define定义的常数:

```
typedef struct {int x[A][B]; long y;} str1;
typedef struct {char array[B]; int t; short s[A]; long u;} str2;
void setVal(str1 *p, str2 *q) {
        long v1 = q->t;long v2 = q->u;
        p->y = v1+v2;
}
```

GCC为setVal产生下面的代码:

```
setVal:
movslq 8(%rsi), %rax
addq 32(%rsi), %rax
movq %rax, 184(%rdi)
ret
```

则A=____,B=____.

A. 8,6　　B.10,8　　C.10,5　　D. 9.5

4.D(4 < B <= 8,5 < A <= 10,44 < A*B <= 46,解得 A=9 B=5)

例题

9. 已知 float 的格式为 1 符号+8 阶码+23 小数，则下列程序的输出结果是：

```
for (int x = 0; ; x++) {
    float f = x;
    if (x != (int)f) {
        printf("%d", x);
        break;
    }
}
```

A. 死循环

B. 4194305 $(2^{22}+1)$

C. 8388609 $(2^{23}+1)$

D. 16777217 $(2^{24}+1)$

例题

9. 已知 float 的格式为 1 符号+8 阶码+23 小数，则下列程序的输出结果是：

```
for (int x = 0; ; x++) {
    float f = x;
    if (x != (int)f) {
        printf("%d", x);
        break;
    }
}
```

A. 死循环

B. 4194305 $(2^{22}+1)$

C. 8388609 $(2^{23}+1)$

D. 16777217 $(2^{24}+1)$

【答】D. 表示 16777217 需要 25 位，除了前导 1 以外还要 24 位，float 无法表示。

bomblab

- 无需提交, 完成一个phase之后, autolab上会自动更新评分
- 一篇老文: 如何优雅地拆弹

  https://mp.weixin.qq.com/s/glcorCvgv48w-2em3KbCyA
- 防止炸弹爆炸: 在爆炸函数入口打断点
- 读汇编技巧
  - 将反汇编代码变成asm文件, 然后用vscode阅读
  - 汇编语言vscode插件: **MASM/TASM**
    - 这样就有语法高亮和折叠了
  - vscode折叠全文快捷键: ctrl+K, 然后ctrl+0

    展开全文: ctrl+K, 然后ctrl+J

bomblab

- 请在class machine上完成: 将bombXXX.tar文件scp传送到class machine, 然后解压
- 反汇编: objdump –d ./bomb
  - 重定向到文件中以方便查看: objdump –d bomb > bomb.asm
- Gdb启动: gdb bomb
  - 打断点: b [函数名] (其他用法: b)
  - 单步执行

    n: 源码下一行, 不进入调用的函数

    s: 源码下一行, 会进入调用的函数 (如果没有源码, 这俩会运行到下一个断点)

    ni: 汇编代码下一行,不进入调用的函数

    si: 汇编代码下一行, 会进入调用的函数
  - 非常好用的layout系列指令
    - layout regs: 显示实时寄存器状态; layout asm: 显示反汇编代码

Thank you!