



汇编语言 程序设计

第5节

80X86汇编语言与C语言-1

C程序在硬件层面的表示

• 数据

- 整数（第二讲）
- 浮点数（第三讲）
- 数组、结构（第八讲）

• 代码

- 基本概念/基本指令/寻址方式（第五讲）
- 程序控制流与相关指令（第六讲）
- 函数调用与相关指令（第七讲）

```
int array[2] = {1, 2};
int main()
{
    int val = sum(array, 2);
    return val;
}
main.c
```

编译

链接

运行

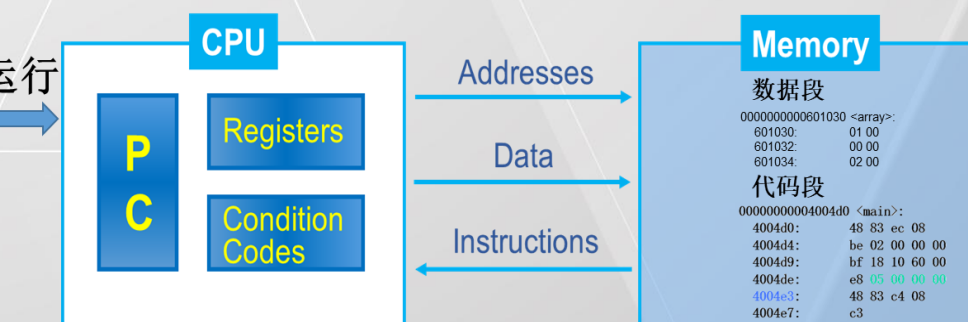
```
0000000000000000 <array>:
0: 01 00      add    %eax, (%rax)
2: 00 00      add    %al, (%rax)
4: 02 00      add    (%rax), %al
0000000000000000 <main>:
0: 48 83 ec 08  sub    $0x8, %rsp
4: be 02 00 00 00  mov    $0x2, %esi
9: bf 00 00 00 00  mov    $0x0, %edi
e: e8 00 00 00 00  callq  13 <main+0x13>
13: 48 83 c4 08  add    $0x8, %rsp
17: c3          retq    main.o
```

汇编指令

机器指令

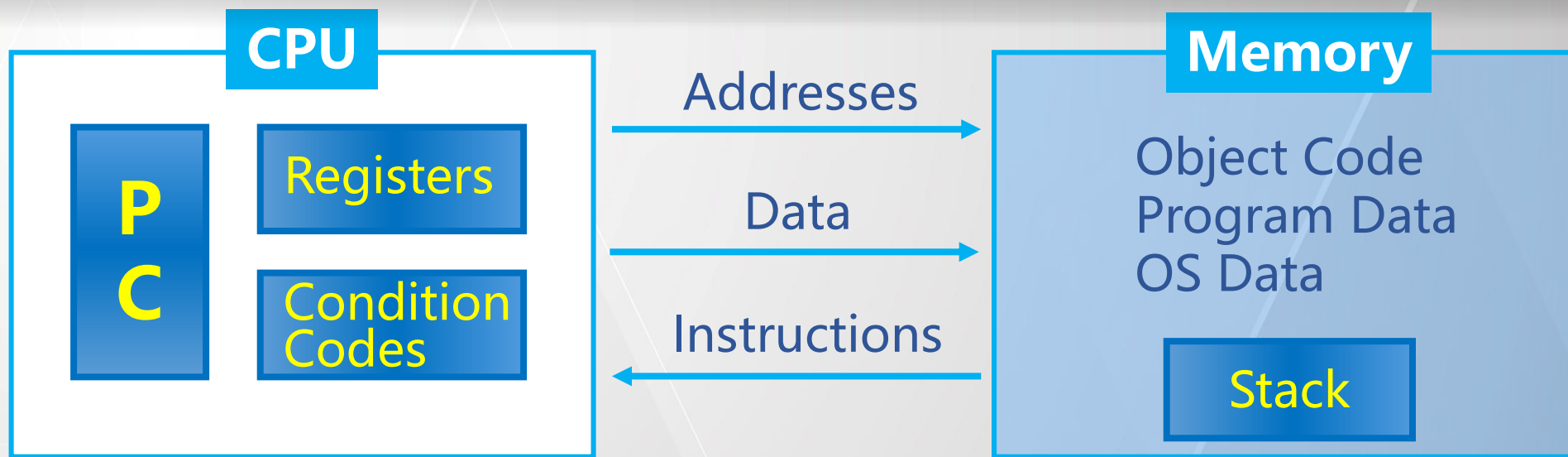
```
00000000004004d0 <main>:
4004d0: 48 83 ec 08
4004d4: be 02 00 00 00
4004d9: bf 18 10 60 00
4004de: e8 05 00 00 00
4004e3: 48 83 c4 08
4004e7: c3
00000000004004e8 <sum>:
4004e8: b8 00 00 00 00
4004ed: ba 00 00 00 00
4004f2: eb 09
4004f4: 48 63 ca
4004f7: 03 04 8f
4004fa: 83 c2 01
4004fd: 39 f2
4004ff: 7c f3
400501: f3 c3 #<array>没有给出
```

内存地址



程序在机器层面的表示与运行

■ 汇编程序员眼中的系统结构 (冯诺依曼架构 / Control Flow)



指令寄存器 (PC Program Counter)

下一条指令的地址

RIP (x86-64)

寄存器与寄存器堆

在处理器内部的以名字来访问的快速存储单元

条件码

用于存储最近执行指令的结果状态信息

用于条件指令的判断执行

存储器 (Memory)

以字节编码的连续存储空间

存放程序代码、数据、运行栈以及操作系统数据

寄存器与存储器的比较

项目	寄存器	存储器
位置	在CPU内部	在CPU外部
访问速度	快	慢
容量	小	大
成本	高	低
表示方式	用名字表示	用地址表示
地址	没有	可用多种方式形成

如何从C代码生成汇编代码

C 代码

```
long plus(long x, long y);

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

对应的X86-64汇编 (AT&T汇编格式)

```
sumstore:
    pushq    %rbx
    movq     %rdx, %rbx
    call     plus
    movq     %rax, (%rbx)
    popq     %rbx
    ret
```

生成汇编代码的命令: `gcc -Og -S sum.c`

生成文件sum.s

`#-fno-stack-protector`

■ 汇编语言的基本特点——数据类型

- **整型数据，数据宽度为 1, 2, 4, or 8 bytes**
 - 表示数值
 - 或者内存地址 (untyped pointers)
- **浮点数据，数据宽度为4, 8, or 10 bytes**
- **无复杂数据类型（结构或者数组之类的）**
 - Just contiguously allocated bytes in memory

汇编语言的基本特点——操作类型

- **对寄存器数据或者内存数据进行算术/逻辑操作**
- **内存与寄存器之间、或者寄存器/寄存器之间传递数据**
 - Load data from memory into register
 - Store register data into memory
- **程序执行顺序的转移 (transfer control)**
 - Unconditional jumps to / from procedures
 - Conditional branches

汇编语言数据格式

Suffix	Name	Size
B	BYTE	1 byte (8 bits)
W	WORD	2 bytes (16 bits)
L	LONG	4 bytes (32 bits)
Q	QUADWORD	8 bytes (64 bits)

在X86中，使用“字 (word)”来表示16位整数类型，“双字”表示32位，“四字”表示64位
汇编语言指令所处理的数据类型一般是采用汇编指令的后缀来进行区分的。

第一条汇编指令实例

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

- C语言

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

- 汇编

- Move 8-byte value to memory

- 操作数:

t:	Register	%rax
dest:	Register	%rbx
*dest:	Memory	M[%rbx]

- 目标代码

- 三字节指令
 - 指令自身的地址为0x40059e

反汇编命令

```
0000000000400595 <sumstore>:
400595: 53                push    %rbx
400596: 48 89 d3          mov     %rdx,%rbx
400599: e8 f2 ff ff ff    callq   400590 <plus>
40059e: 48 89 03          mov     %rax, (%rbx)
4005a1: 5b                pop     %rbx
4005a2: c3                retq
```

- 反汇编程序
objdump -d sum



另一种反汇编方式

Object

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

Dump of assembler code for function sumstore:

0x00000000000400595 <+0>: push %rbx

0x00000000000400596 <+1>: mov %rdx,%rbx

0x00000000000400599 <+4>: callq 0x400590 <plus>

0x0000000000040059e <+9>: mov %rax, (%rbx)

0x000000000004005a1 <+12>: pop %rbx

0x000000000004005a2 <+13>: retq

- gdb命令反汇编

`gdb sum`

`disassemble sumstore`

- 反汇编某个过程/函数

`x/14xb sumstore`

- 显示从“sumstore”开始的14个字节

`x /nfu <addr>`



- **gdb：命令行调试工具，建议在gcc 编译时加 -g 参数**
 - 安排有专门的课时来讲解gdb等的使用

gdb 介绍

gdb 是一个调试器，可以在运行程序的时候，设置断点，中途查看程序运行的状态，并且逐步观察程序运行行为。

我们引入下面的代码作为例子，来观察 gdb 的使用方式：

```
1  #include <stdio.h>
2  int number;
3  int main(int argc, char *argv[]) {
4      printf("%d %s\n", argc, argv[0]);
5      scanf("%d", &number);
6      printf("%d\n", number);
7      return 0;
8  }
```

按照前面讲述的方法，编译成二进制：g++ test.cpp -o test

gdb 介绍

用 gdb 调试 test 程序，首先运行 gdb test，然后输入 run 命令开始运行：

```
1  $ gdb test
2  GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
3  ...
4  Reading symbols from test...
5  (No debugging symbols found in test)
6  (gdb) run
7  Starting program: /home/jiegec/test
8  1 /home/jiegec/test
9  1234
10 1234
11 [Inferior 1 (process 1773151) exited normally]
12 (gdb)
```

程序运行以后，输出了 argc argv，这时候可以正常向程序输入数据。程序退出以后，回到 gdb 的命令。接下来，我们要设置断点，来观察程序的行为。

x86-64的通用寄存器

%rax	%eax
%rdx	%edx
%rcx	%ecx
%rbx	%ebx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d



x86-32的通用寄存器

general purpose	(mostly obsolete)				
	%eax	%ax	%ah	%al	accumulate
	%ecx	%cx	%ch	%cl	counter
	%edx	%dx	%dh	%dl	data
	%ebx	%bx	%bh	%bl	base
	%esi	%si			source index
	%edi	%di			destination index
	%esp	%sp			stack pointer
	%ebp	%bp			base pointer
16-bit virtual registers (backwards compatibility)					

X86-64指令集体系结构下指令寄存器（program counter）的名字是 [填空1]；相比于存储器，寄存器位于处理器 [填空2]（内部/外部？），访问速度 [填空3]（快/慢？）。

x86-64的通用寄存器个数为 [填空4]，x86-16的通用寄存器个数为 [填空5]。



数据传送指令 (mov)

■ 数据传输

`movq Source, Dest`

■ 操作数类型

- 立即数: 整型常数
 - Example: `$0x400`, `$-533`
 - Like C constant, but prefixed with ``$'`
 - Encoded with 1, 2, or 4 bytes
- 寄存器: 16个通用处理器之一
 - Example: `%rax`, `%r13`
 - But `%rsp` reserved for special use
 - Others have special uses for particular instructions
- 内存: 8个连续字节, 起始地址由相应寄存器内容指定
 - Simplest example: `(%rax)`
 - Various other “address modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

■ 数据传送指令支持的不同操作数类型组合

	Source	Dest	Src, Dest	类似的C语言表示
movl	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

但是不能两个操作数都为内存地址~

简单的寻址模式

间接寻址 **(R)** **Mem[Reg[R]]**

寄存器R指定内存地址

`movq (%rcx),%rax`

基址+偏移量 寻址 **D(R)** **Mem[Reg[R]+D]**

寄存器R指定内存起始地址

常数D给出偏移量

`movq 8(%rbp),%rdx`



X86-64下的简单寻址示例

```
void swap
(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

参数通过寄存器来传递

First (xp) in %rdi, second (yp) in %rsi
64-bit pointers

当参数少于7个时，参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。当参数为7个及以上时，前6个传送方式不变，但后面的依次从“右向左”放入栈中。

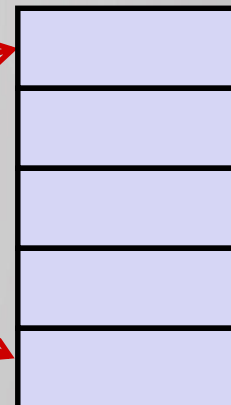
Swap函数执行过程分解-1

```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Registers

%rdi	
%rsi	
%rax	
%rdx	

Memory



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Swap函数执行过程分解-2

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

Address
0x120
123
0x118
0x110
0x108
0x100
456

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Swap函数执行过程分解-3

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

Memory

Address
0x120
123
0x118
0x110
0x108
0x100
456

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Swap函数执行过程分解-4

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

Address
0x120
123
0x118
0x110
0x108
0x100
456

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```


Swap函数执行过程分解-5

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

Address
0x120
0x118
0x110
0x108
0x100



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```


Swap函数执行过程分解-6

Registers

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

Address
0x120
0x118
0x110
0x108
0x100

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```



X86-64下另一个swap

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    retq
```

参数通过寄存器来传递

First (xp) in %rdi, second (yp) in %rsi

64-bit pointers

被操作的数据仍是32位

所以使用寄存器 %eax、%edx
以及movl 指令

简单的寻址模式

间接寻址 **(R)** **Mem[Reg[R]]**

寄存器R指定内存地址

`movq (%rcx),%rax`

基址+偏移量 寻址 **D(R)** **Mem[Reg[R]+D]**

寄存器R指定内存起始地址

常数D给出偏移量

`movq 8(%rbp),%rdx`

变址寻址

常见形式

$D(Rb, Ri, S) \quad Mem[Reg[Rb] + S * Reg[Ri] + D]$

D: 常量 (地址偏移量)

Rb: 基址寄存器: 16个通用寄存器之一

Ri: 索引寄存器: `%rsp` 不作为索引寄存器

S: 比例因子 1, 2, 4, or 8

其他变形

$(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri]]$

$D(Rb, Ri) \quad Mem[Reg[Rb] + Reg[Ri] + D]$

$(Rb, Ri, S) \quad Mem[Reg[Rb] + S * Reg[Ri]]$

寻址模式实例

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8(%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx,%rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx,%rcx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%rdx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>



地址计算指令

`leaq Src, Dest`

Src 是地址计算表达式

计算出来的地址赋给 *Dest*

使用实例

地址计算 (无需访存)

E.g., translation of `p = &x[i];`

进行 $x + k*y$ 这一类型的整数计算

$k = 1, 2, 4, \text{ or } 8.$

示例:

```
long m12(long x)
{
    return x*12;
}
```

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax             # return t<<2
```

整数计算指令

Format

双操作数指令

addq Src, Dest

subq Src, Dest

imulq Src, Dest

salq Src, Dest

sarq Src, Dest

shrq Src, Dest

xorq Src, Dest

andq Src, Dest

orq Src, Dest

Computation

$Dest = Dest + Src$

$Dest = Dest - Src$

$Dest = Dest * Src$

$Dest = Dest \ll Src$

$Dest = Dest \gg Src$

$Dest = Dest \gg Src$

$Dest = Dest \wedge Src$

$Dest = Dest \& Src$

$Dest = Dest | Src$

Multiply（取低64位）

与shll等价

算术右移

逻辑右移

整数计算指令

Format

单操作数指令

`incq Dest`

`decq Dest`

`negq Dest`

`notq Dest`

Computation

$Dest = Dest + 1$

$Dest = Dest - 1$

$Dest = - Dest$

$Dest = \sim Dest$

将leal指令用于计算

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx
    leaq    4(%rdi,%rdx), %rcx
    imulq    %rcx, %rax
    ret
```

关键指令:

- *leaq*: 地址计算
- *salq*: 移位操作
- *imulq*: 乘法
 - 但只用了一次

将leal指令用于计算

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```
arith:
    leaq    (%rdi,%rsi), %rax    # t1
    addq    %rdx, %rax          # t2
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx            # t4
    leaq    4(%rdi,%rdx), %rcx   # t4+x+4
    imulq    %rcx, %rax          # rval
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

将leal指令用于计算

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

$$2^{13} = 8192, 2^{13} - 7 = 8185$$

logical:

```
movl    %edi, %eax
xorl    %esi, %eax
sarl    $17, %eax
andl    $8185, %eax
ret
```

Register	Use(s)
%edi	Argument x
%esi	Argument y
%eax	t1, t2, rval

x86-32 与 x86-64的数据类型宽度

Sizes of C Objects (in Bytes)

C Data Type	Typical 32-bit	Intel IA32	x86-64
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char *	4	4	8

Or any other pointer

小结

X86-64指令的特点

支持多种类型的指令操作数

立即数, 寄存器, 内存数据

算逻指令可以以内存数据为操作数

支持多种内存地址计算模式

$$Rb + S * Ri + D$$

也可用于整数计算(如leal / leaq指令)

变长指令

from 1 to 15 bytes

X86汇编的格式

Intel/Microsoft Format

(-masm=intel, Intel语法)

```
lea    eax, [ecx+ecx*2]
sub     esp, 8
cmp     dword ptr [ebp-8], 0
mov     eax, dword ptr [eax*4+100h]
```

AT&T Format

```
leal    (%ecx,%ecx,2),%eax
subl    $8,%esp
cmpl    $0,-8(%ebp)
movl    0x100(,%eax,4),%eax
```

Intel/Microsoft Differs from GAS

- Operands listed in opposite order

mov Dest, Src

movl Src, Dest

- Constants not preceded by '\$', Denote hex with 'h' at end

100h

\$0x100

- Operand size indicated by operands rather than operator suffix

sub

subl

- Addressing format shows effective address computation

[eax*4+100h]

0x100(,%eax,4)