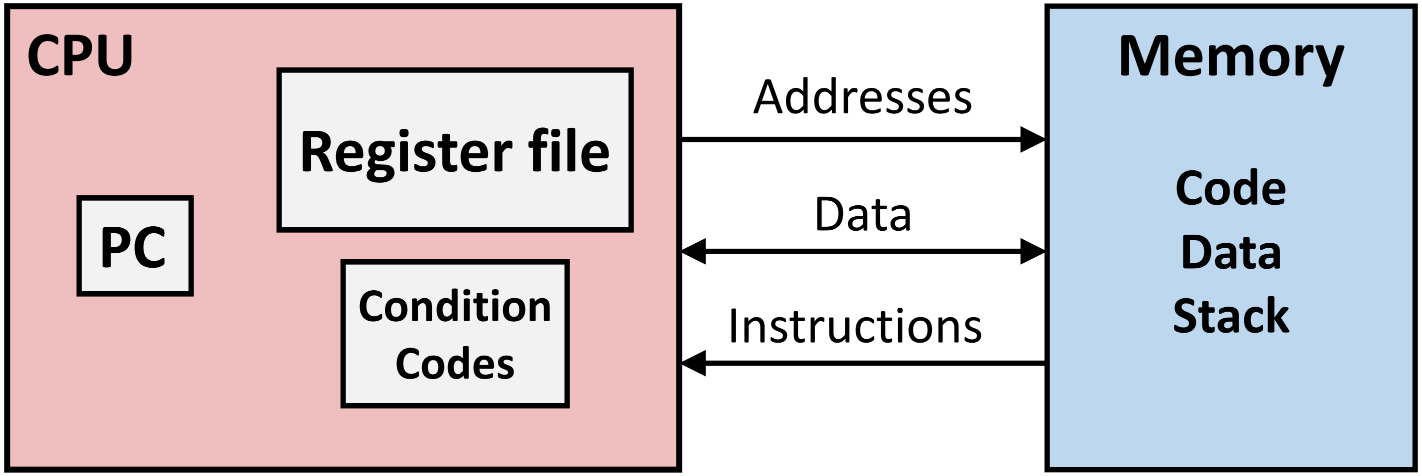
程序的机器级表示(2)

王晶

jwang@ruc.edu.cn，信息楼124

2023年10月

ISA = Assembly/Machine Code View



Programmer-Visible State

✓ **PC: Program counter**
- Address of next instruction

✓ **Register file**
- Heavily used program data

✓ **Condition codes**
- Store status information about most recent arithmetic or logical operation
- Used for conditional branching

– **Memory**
- Byte addressable array
- Code and user data
- Stack to support procedures

汇编语言

- 一种较低级的程序设计语言
- 主要是机器语言的符号化描述
- 通常为特定计算机或计算机系列专门设计

x86汇编语言示例

```
MOV   AX,  0H
MOV   BX,  [20H]
ADD   AX,  BX
ADD   BX,  1H
JMP   label_next
```

ARM汇编语言示例

```
MOV   R0,  #0
LDR   R2,  #0x10020
ADD   R0,  R0,  R2
ADD   R2,  R2,  #1
B     label_next
```

学习汇编语言的目的

借助汇编语言学习计算机的工作原理

使用汇编语言编写底层驱动程序、实时控制程序等

通过了解高级语言可以如何转换成高效的可执行代码，提高编程质量

借助反汇编工具分析、调试机器语言代码

学习汇编语言的目的

- 编译器也能生成汇编，为什么还要学习汇编
- 理解编译器的优化能力
  - 不能过于贬低：比绝大多数人工强
  - 也不能过于夸大：
    - 有些看似简单的地方，为了避免bug，不敢优化
    - 有些复杂问题还是很难优化（例如并行编译）
- 分析代码中低效的部分，以便提升程序性能
  - 汇编与性能直接对应

From writing to understand assembly code

- 直接写汇编程序
  - 熟悉汇编语言
  - 习惯汇编的思维方式（可以直接操控硬件）
  - 建立高级语言（C）和汇编之间的联系

实变函数学十遍
汇编语言不会编
随机过程随机过
量子力学量力学

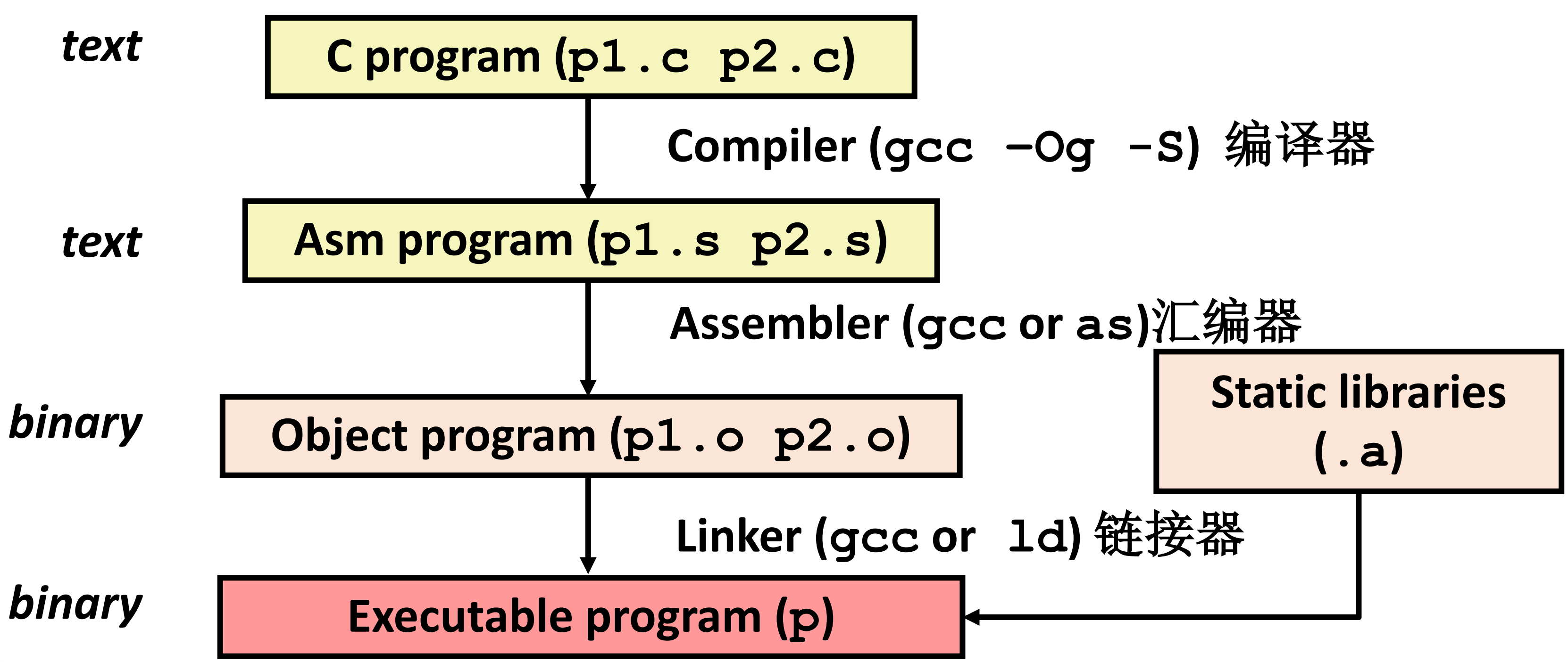
- 逆向工程
  - 从C翻译到汇编
  - 理解在系统中程序到底做了什么
  - 帮助找出bug
  - 帮助找出程序的安全隐患
  - 帮助找出性能差的环节

程序的机器级表示

- 如何生成汇编语言程序
- 汇编语言程序的结构

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 binary in file `p`



Compiling Into Assembly

C Code (sum.c)

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

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

Generated x86-64 Assembly

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

Obtain (on shark machine) with command

```
gcc –Og –S sum.c
```

Produces file `sum.s`

***Warning*: Will get very different results on other machines (Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.**

汇编和反汇编

```c
#include  <stdio.h>

int  main ( )
{
      printf ( "hello, world\n" ) ;
      return 0;
}
```

gcc -E test.c -o test.i
gcc -S test.i -o test.s
或
test.s gcc –S test.c –o test.s

```
add:
pushl %ebp
movl  %esp, %ebp
subl  $16, %esp
movl  12(%ebp), %eax
movl  8(%ebp), %edx
leal  (%edx, %eax), %eax
movl  %eax, -4(%ebp)
movl  -4(%ebp), %eax
leave
ret
```

"gcc –c test.s –o test.o"将test.s汇编为test.o

"objdump -d test.o"将test.o 反汇编为.s文件

00000000 <add>:

位移量	机器指令	汇编指令
0:	55	push %ebp
1:	89 e5	mov %esp, %ebp
3:	83 ec 10	sub $0x10, %esp
6:	8b 45 0c	mov 0xc(%ebp), %eax
9:	8b 55 08	mov 0x8(%ebp), %edx
c:	8d 04 02	lea (%edx,%eax,1), %eax
f:	89 45 fc	mov %eax, -0x4(%ebp)
12:	8b 45 fc	mov -0x4(%ebp), %eax
15:	c9	leave
16:	c3	ret

编译得到的与反汇编得到的汇编指令形式稍有差异

汇编和反汇编工具

- 汇编器
  - ✓as, gcc依赖的汇编器
- 链接器
  - ✓ld
- 调试器
  - ✓Gdb
- 反汇编器
  - ✓`objdump -d sum`
  - ✓ Useful tool for examining object code
  - ✓ Analyzes bit pattern of series of instructions
  - ✓ Produces approximate rendition of assembly code
  - ✓ Can be run on either `a.out` (complete executable) or `.o` file

```
$ as hello.s -o hello.o
$ ld hello.o -o hello
$ ./hello
$ echo $?   #打印返回值
```

```
[root@master test]# vim hello.s
[root@master test]# as -o hello.o hello.s
[root@master test]# ld -o hello hello.o
[root@master test]# ./hello
hello world!
```

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- Within gdb Debugger

  `gdb sum`

  `disassemble sumstore`

  - Disassemble procedure

  `x/14xb sumstore`

  - Examine the 14 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:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by
Microsoft End User License Agreement

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

程序的机器级表示

- 如何生成汇编语言程序
- 汇编语言程序的结构

真实的汇编程序

```
        .globl sumstore
        .type  sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq  %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq   %rdx, %rbx
        call   plus
        movq   %rax, (%rbx)
        popq   %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size  sumstore, .-sumstore
```

引导语句

```
        .globl sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

Things that look weird and are preceded by a '.' are generally directives.

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

“Hello World”

```
.data
    msg : .string "hello world!\n"
    len = . – msg
.text
    .global
    _start:
            movq $len, %rdx     ;第4个参数，字符串长度
            movq $msg, %rcx     ;第3个参数，字符串地址
            movq $1, %rbx       ;第2个参数，1指定标准输出（0-标准输入）
            movq $4, %rax       ;第1个参数，4为系统调用号：sys_write
            int $0x80           ;80中断，系统调用
            movl $0, %ebx       ;参数1，退出代码，指定的返回值
            movl $1, %eax       ;1号系统调用：sys_exit
            int $0x80
```

程序的机器级表示

- 如何生成汇编语言程序
- 汇编语言程序的结构
    - **.text 程序段**
    - .data 数据段
    - .stack 堆栈段

Code Examples

```
//C code
int accum = 0;
int sum(int x, int y)
{
  int t = x+y;
  accum += t;
  return t;
}
```

Obtain with command

```
gcc -O2 -S code.c
```

Assembly file `code.s`

instruction

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    addl %eax, accum
    movl %ebp,%esp
    popl %ebp
    ret
```

From C Codes to Assembly codes

- Instruction
  - Performs a very elementary operation only
- Add two signed integers
  - C code:
    - int t = x+y;
  - Assembly code:
    - addl   8(%ebp),%eax
    - Add 2 4-byte integers
    - Similar to expression x +=y

Machine Instruction

```
*dest = t;
```

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

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

- C 代码
  - Store value `t` where designated by `dest`
- 汇编代码Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - 操作数Operands:

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

- 目标代码Object Code
  - 3-byte 指令
  - 存储在地址 `0x40059e`

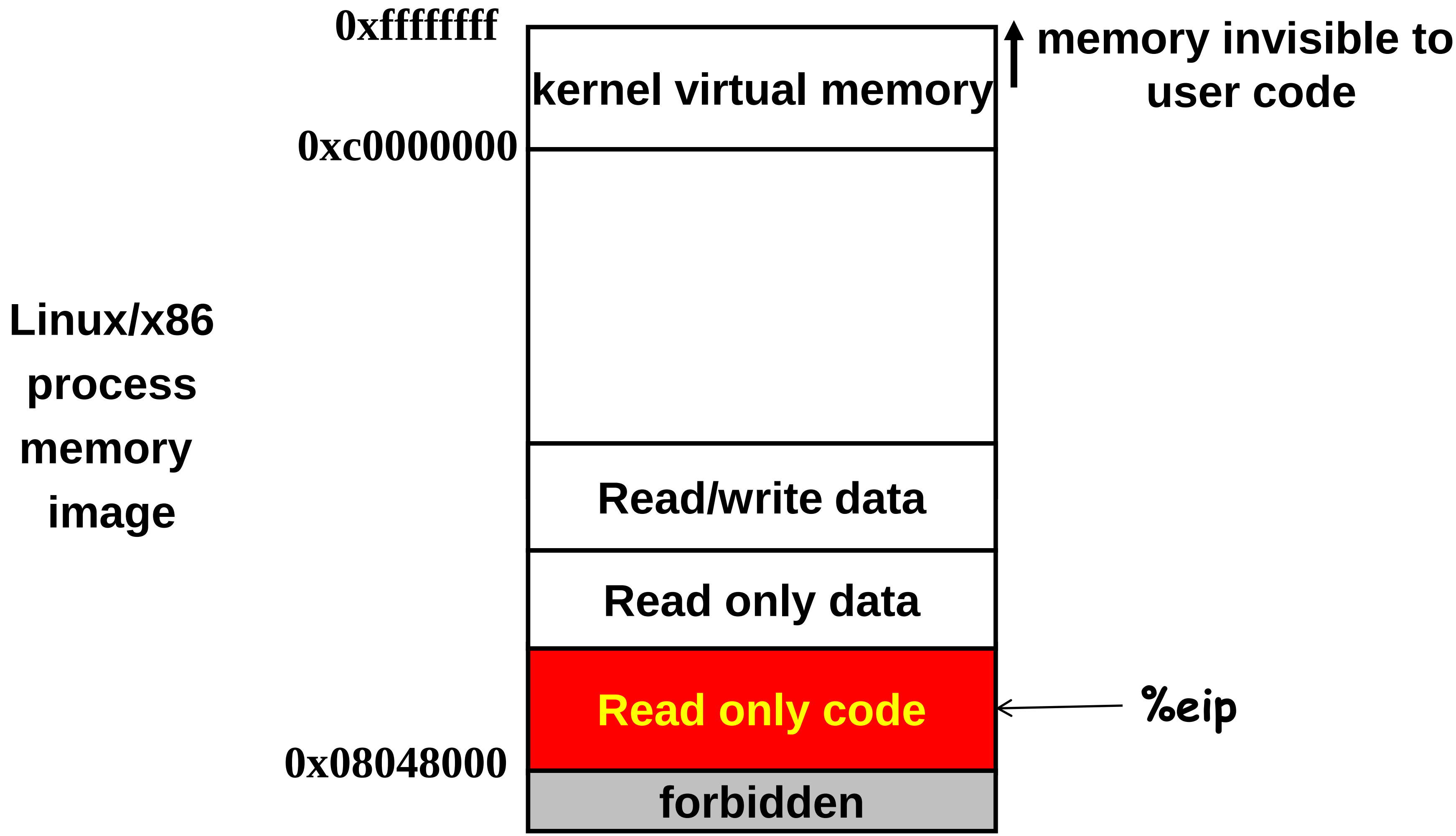
汇编指令的执行

- 程序
  - 一个指令的序列
- 指令
  - 机器执行动作的最小单位
- 执行模式
  - 顺序执行下一条指令
  - 条件转移指令跳到新的不连续地址
- 操作数可以存储在寄存器和存储器中，大多数在寄存器中
- 并可以在寄存器和存储器之间传输数据

Understanding Machine Execution

- 指令存在哪里?
  - 主存中
  - 代码段
- 指令如何执行?
  - %eip 保存了一个存储器地址,
  - 从这个存储器地址读取一条指令
  - 增加 %eip
    - %eip 是program counter (PC)
  - 执行这条指令

Code Layout

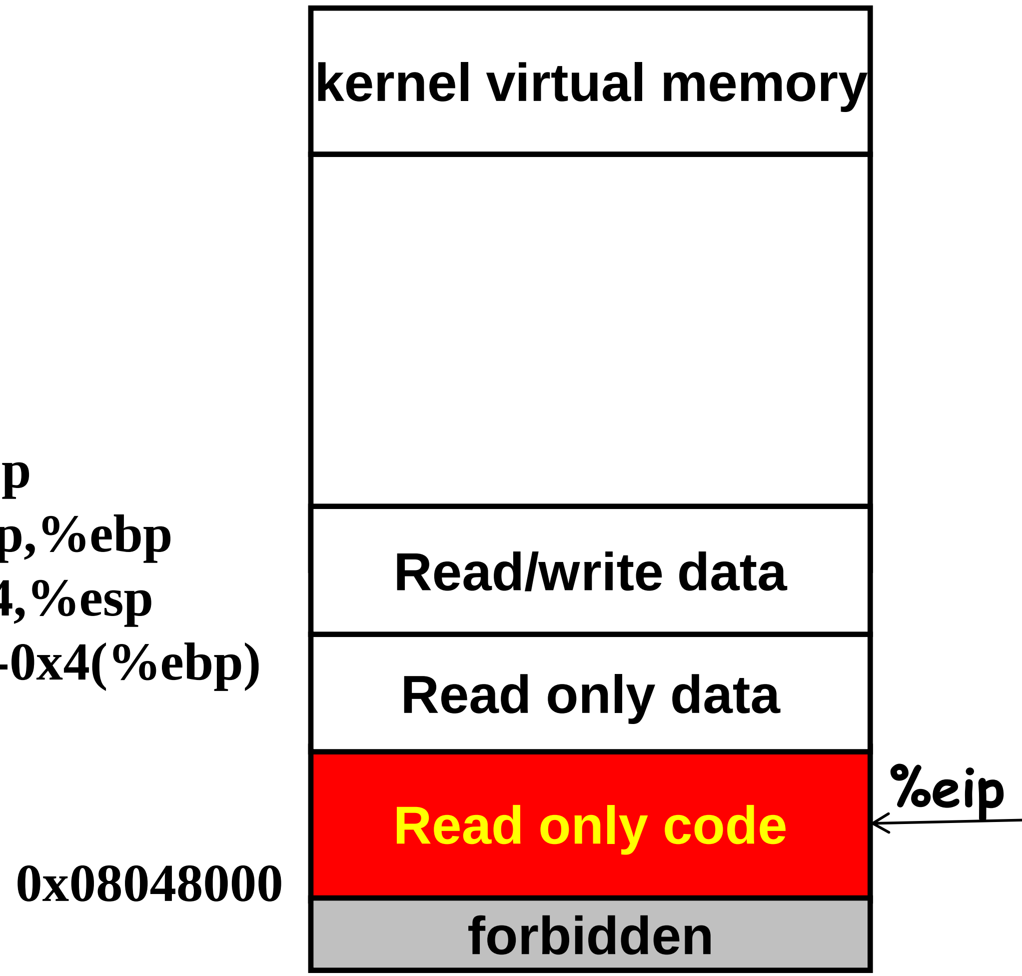


0xffffffff
kernel virtual memory
memory invisible to user code
0xc0000000
Linux/x86 process memory image
Read/write data
Read only data
Read only code
%eip
0x08048000
forbidden

Sequential execution

```
f()
{
   int i = 3 ;
}
```

```
08048390 <_f>:
   90:  55                  push   %ebp
   91:  89 e5               mov    %esp,%ebp
   93:  83 ec 14            sub    $0x14,%esp
   96:  c7 45 fc            movl   $03, -0x4(%ebp)
   9d:  c9  03 00 00 00     leave
   9e:  c3                  ret
```



Sequential execution

000000 f>
0: 55
1: 89 e5 mov %esp,%ebp
3: 83 ec 14 sub $0x14,%esp
6: c7 45 fc 03 00 00 00
movl $0x3,-0x4(%ebp)
d: c9 leave
e: c3 ret

PC 08 04 83 9e

PC 08 04 83 9d

PC 08 04 83 96

PC 08 04 83 93

PC 08 04 83 91

PC 08 04 83 90

	c3	ret
	c9	leave
9c	00	
	00	
	00	
	03	
98	fc	
	45	
	c7	movl $0x3,-0x4(%ebp)
	14	
94	ec	
	83	sub $0x14,%esp
	e5	
	89	mov %esp,%ebp
90	55	push %ebp

程序的机器级表示

- 如何生成汇编语言程序
- **汇编语言程序的结构**
    - .text 程序段
    - **.data 数据段**
        - 数据类型：是什么
        - 数据寻址：怎么找
        - 数据移动：怎么用
    - .stack 堆栈段

Operands（操作数）

- 高级语言
  - 常量
  - 变量

- 汇编语言
  - Immediate（立即数）:
  - Register（寄存器）
  - Memory（存储器）

A = A + 4

variable constant

memory

movl 8(%ebp), %eax

register

addl $4, %eax

immediate

汇编语言的数据类型

- 高级语言
  - Integer data of 1, 2, 4, or 8 bytes
  - Floating point data of 4, 8, or 10 bytes
  - SIMD vector data types of 8, 16, 32 or 64 bytes
- 汇编级别
  - 数据的值
  - 地址 (没有类型信息的指针)
  - 没有数组和结构体，不区分 signed 和 unsigned integers
  - 仅以字节为单位，分配内存空间
  - 代码：字节序列，解码为指令

常量vs变量

```
.data
        msg : .string "hello world!\n"
        len = . – msg
```

- **符号常量**
  - Len = . – msg
  - 符号常量使用标识符表达一个数值
  - 在变为机器代码时会直接替换为对应的值
- **变量**
  - 本质是(临时)拥有一段内存，变量即内存的地址
  - 在变为机器代码时会直接替换为对应内存段的首地址
  - .ascii 字符串
  - .short 短整型 16位2字节
  - .int .long 长整型 32位4字节 (同一平台C语言中long 为8字节）
  - .byte 1个字节
  - .float 浮点单精度
  - .double 浮点双精度

例题

- 假设是8086，DS=2000H，要求画出内存状态图
  - Hint: intel系列CPU采用小端方式
- .data
  - byte1 .byte 10110111b
  - byte2 = . – byte1
  - word1 .short -9
  - dword1 .int 2274H
  - word2 .short 2274Q

0xB7
0xF7
0xFF
0x74
0x22
0x00
0x00
0xBC
0x04

010 010 111 100

0100 1011 1100

4 B C

程序的机器级表示

- 如何生成汇编语言程序
- **汇编语言程序的结构**
  - .text 程序段
  - **.data 数据段**
    - 数据类型：是什么
    - 数据寻址：怎么找
    - 数据移动：怎么用
  - .stack 堆栈段

数据寻址方式的分类

① 立即寻址

数据在指令中

② 寄存器寻址

数据在寄存器中

③ 直接寻址

④ 寄存器间接寻址

⑤ 寄存器相对寻址

⑥ 基值变址寻址

⑦ 相对基址变址寻址

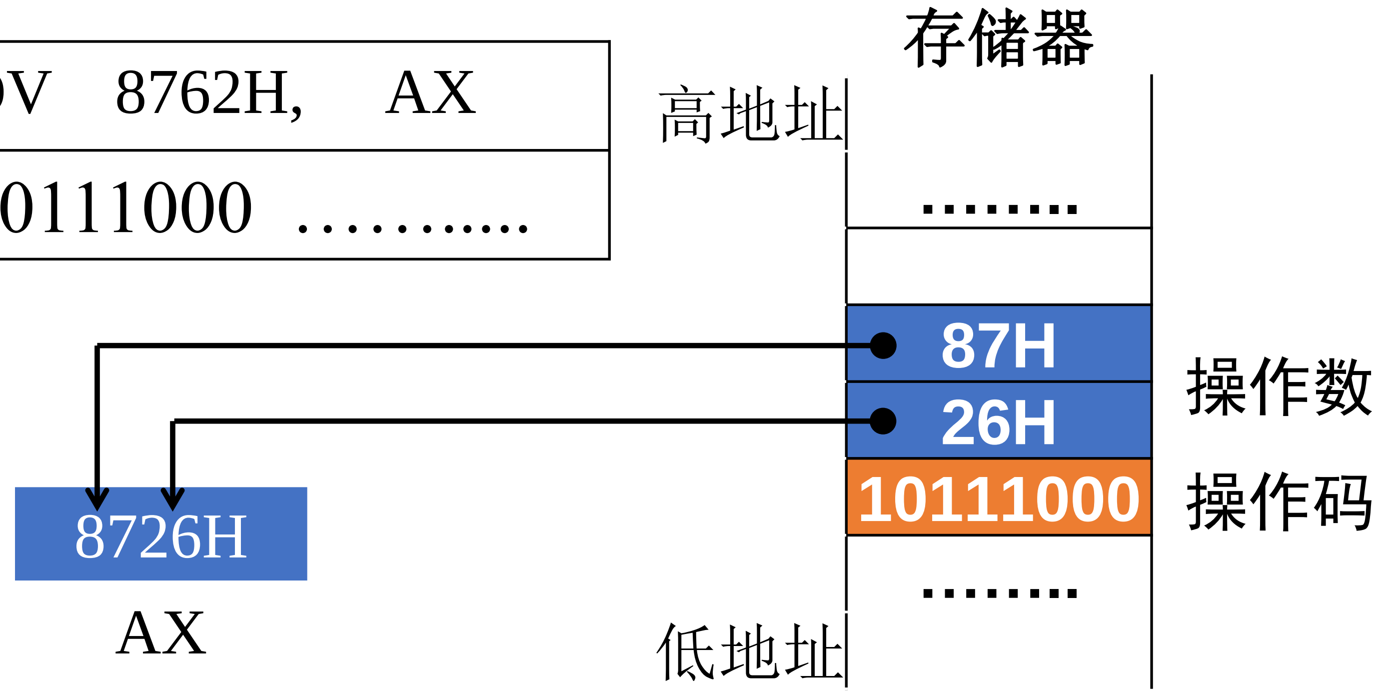
⑧ 比例变址寻址

数据在存储器中

1.立即寻址（Immediate Addressing）

- 操作数作为指令机器码的一部分，在取出指令的同时就取出了操作数

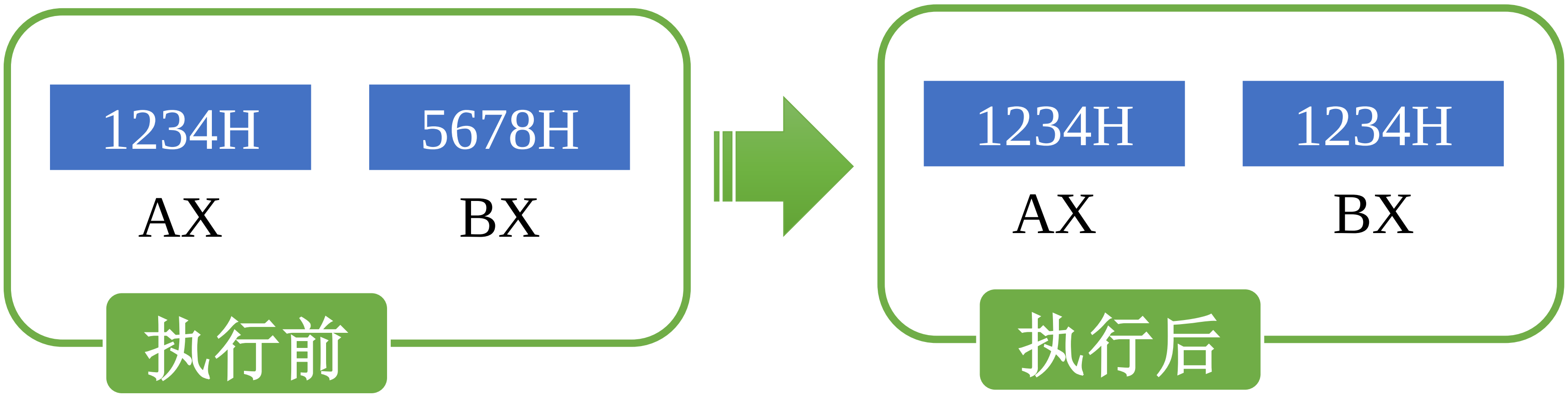
汇编指令	MOV 8762H, AX
机器码	10111000 …………



2.寄存器寻址（Register Addressing）

- 操作数在寄存器中，指令中指明寄存器号

汇编指令	MOV AX, BX
机器码	1000101111000011



寄存器寻址的优点

执行速度快

- 操作数在CPU内部, 不需要访问存储器来取得操作数

指令编码较短

- 寄存器号比立即数、内存地址短

*在编程中，应尽量使用这种寻址方式的指令
*常用数据大多存放在寄存器中，int/float

数据寻址方式的分类

① 立即寻址
② 寄存器寻址
③ 直接寻址
④ 寄存器间接寻址
⑤ 寄存器相对寻址
⑥ 基值变址寻址
⑦ 相对基址变址寻址
⑧ 比例变址寻址

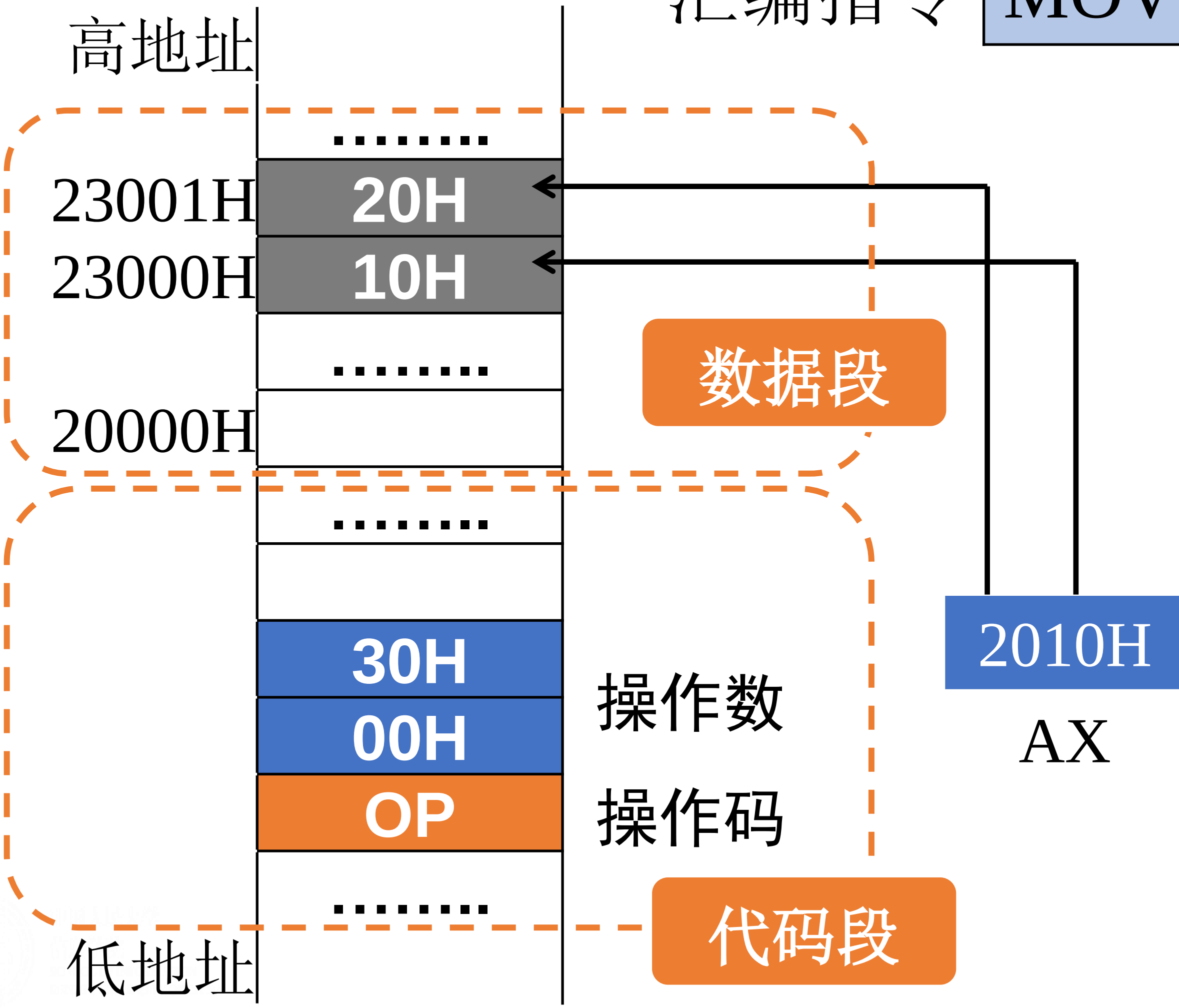
数据在存储器中

3.直接寻址（Direct Addressing）

- 指令中直接给出操作数的有效地址，有效地址指向存放在存储器中的操作数

存储器

汇编指令 MOV %AX, （3000H）



操作数默认存放在DS指向的数据段中，即(3000H)=DS:(3000H)

设：DS=2000H，

则：物理地址

=2000H×10H+3000H

=23000H

要点说明

- 如果数据存放在数据段以外的其它段（例如附加段），则应在指令中给出“段跨越前缀”
  - 示例1：MOV AX , ES:[3000H]
  - 示例2：ES: MOV AX , [3000H]

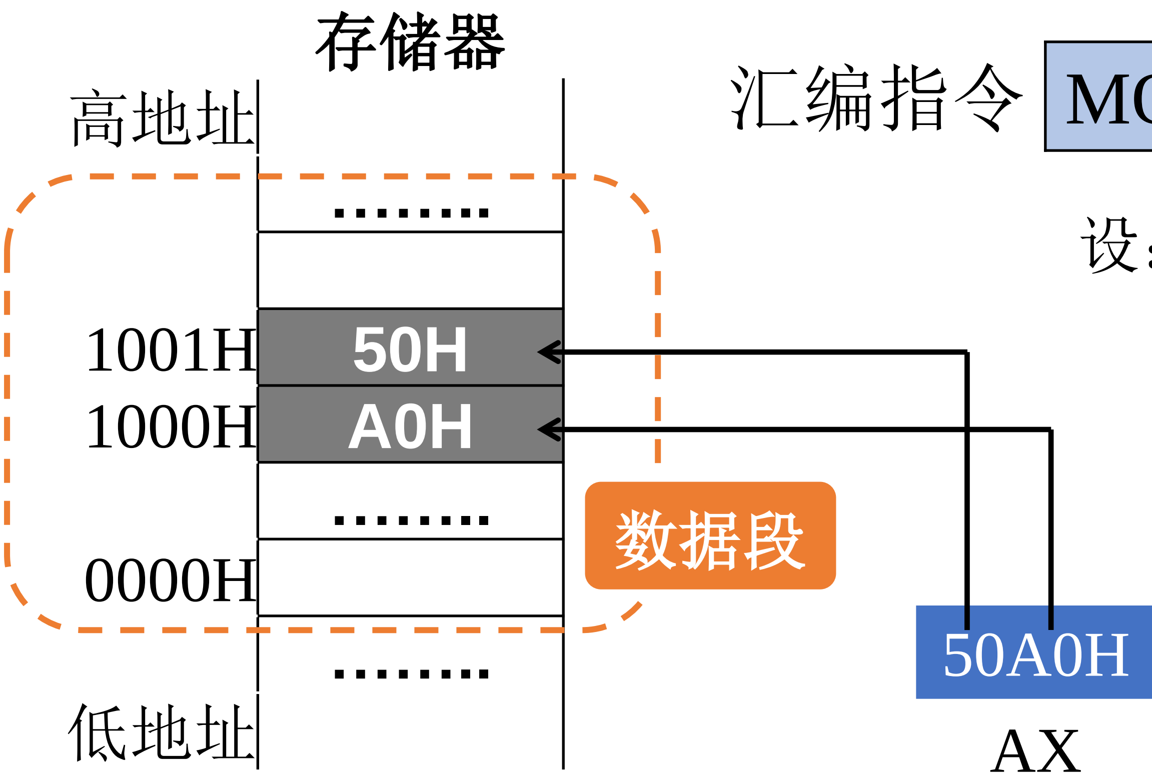
- 为避免指令字过长，规定双操作数指令不能两个操作数都用直接寻址方式
  - 错误示例：MOV DS:[2000H] , DS:[3000H]

4.寄存器间接寻址（Register Indirect Addressing）

- 指令中给出寄存器号，指定的寄存器中存放着操作数的有效地址

汇编指令 MOV %AX, （%BX）

设：BX=1000H



5.寄存器相对寻址（Register Relative Addressing）

- 操作数的有效地址是一个基址或变址寄存器的内容与一个位移量之和

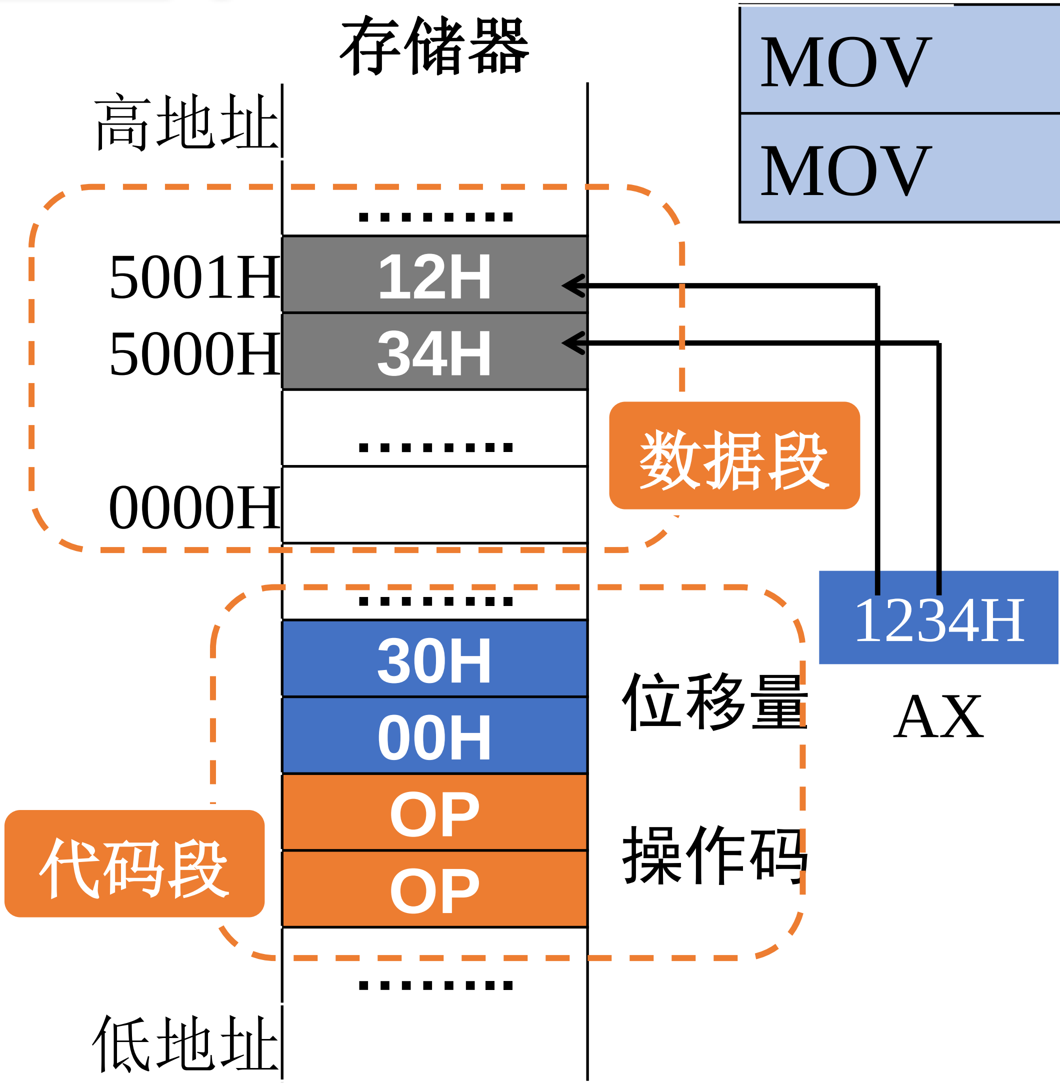
有效地址 =	BX BP SI DI	+	8位 16位 32位 （带符号数）
	基址或变址		位移量

主存以字节为可寻址单位

地址的加减是以字节为单位

寻址方式示例

MOV	%AX,	(%SI+3000H)
MOV	%AX,	3000H(%SI)



设：SI=2000H

则：地址
=2000H+3000H
=5000H

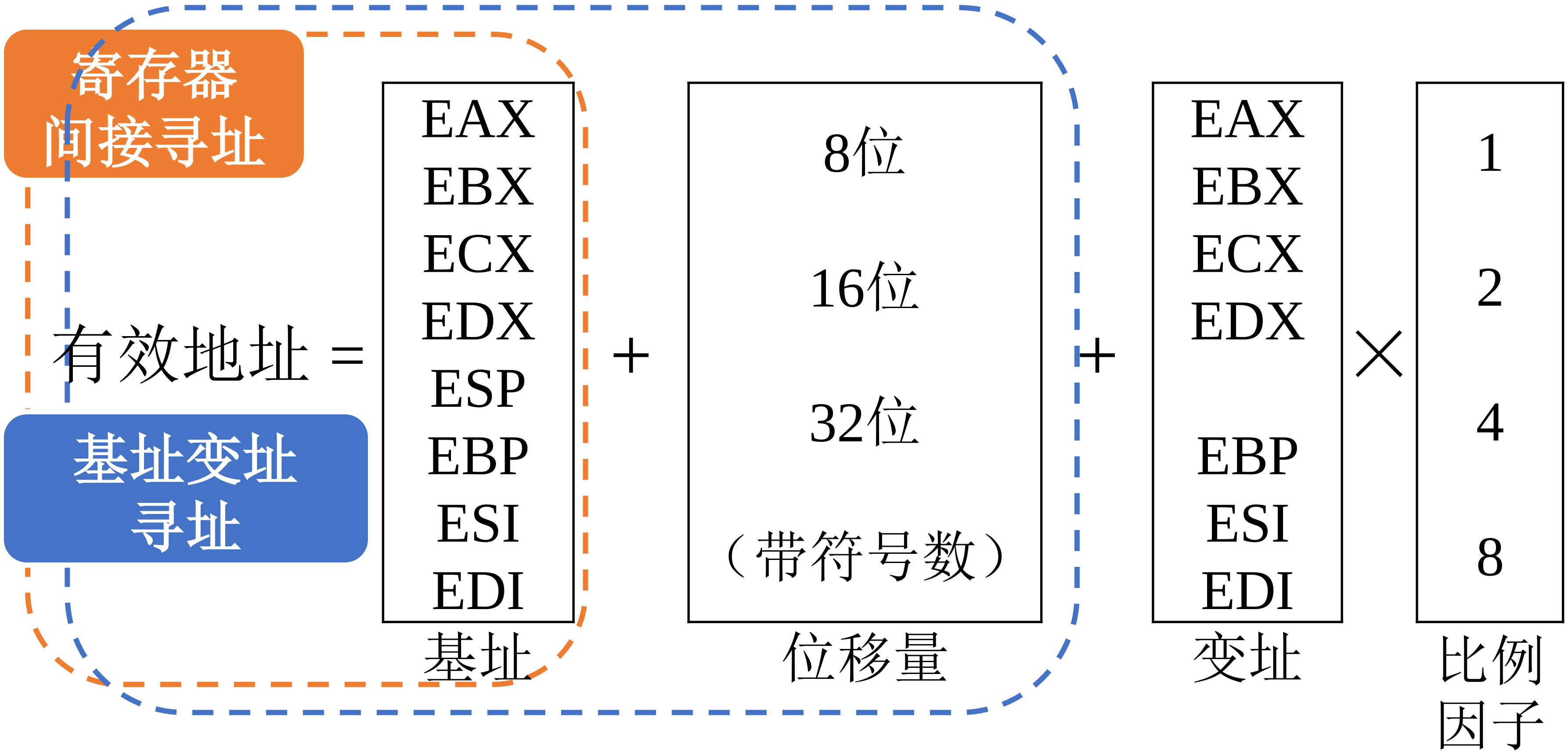
6.比例变址寻址（Scaled Indexed Addressing）

- 操作数的有效地址由基址寄存器、变址寄存器、比例因子和位移量按如下公式生成

有效地址 =	EAX EBX ECX EDX ESP EBP ESI EDI	+	EAX EBX ECX EDX EBP ESI EDI	×	1 2 4 8	+	8位 16位 32位 （带符号数）
	基址		变址		比例因子		位移量

通用公式

- 比例变址寻址是IA-32新增的寻址方式
- 提供了存储器操作数寻址方式的通用公式



地址的索引

- 内存地址的表达式
- Imm(E_b, E_i, s)
  - 立即数: 1, 2 or 4 bytes
  - 基址寄存器Base register E_b: Any of 8 integer registers
  - 变址寄存器Index register E_i : Any, except for `%esp`
  - `s`： Scale: 1, 2, 4, or 8
- 地址 Imm(E_b, E_i, s)
  - imm + R[E_b] + R[E_i] * s
- 表达的值
  - M[imm + R[E_b] + R[E_i] * s]

寻址方式

Type	Form	Operand value	Name
Immediate	$Imm	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Indexed	Imm	M[Imm]	Absolute直接
Indexed	(E_a)	$M[R[E_a]]$	Indirect间接
Indexed	$Imm(E_b)$	$M[Imm+ R[E_b]]$	Base+displacement基址
Indexed	(E_b, E_i)	$M[R[E_b]+ R[E_i]]$	Indexed变址
Indexed	$Imm(E_b, E_i)$	$M[Imm+ R[E_b]+ R[E_i]]$	Scaled indexed比例
Indexed	$(, E_i, s)$	$M[R[E_i]*s]$	Scaled indexed比例
Indexed	(E_b, E_i, s)	$M[R[E_b]+ R[E_i]*s]$	Scaled indexed比例
Indexed	$Imm(E_b, E_i, s)$	$M[Imm+ R[E_b]+ R[E_i]*s]$	Scaled indexed比例

举例

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Value
%eax	0x100
(%eax)	0xFF
$0x108	0x108
0x108	0x13
260(%ecx,%edx)	(0x108)0x13
(%eax,%edx,4)	(0x10C)0x11

有$表示值，没有表示地址

程序的机器级表示

- 如何生成汇编语言程序
- 汇编语言程序的结构
    - .text 程序段
    - **.data 数据段**
        - 数据类型：是什么
        - 数据寻址：怎么找
        - 数据移动：怎么用
    - .stack 堆栈段

数据移动指令

- 数据移动

  `movq` ***Source, Dest*:**

- 操作数类型
  - ***Immediate立即数*:** 整数常量
    - **`$0x400`, `$-533`**
    - 类似C语言的常量，用**`'$'`**做前缀
    - 1, 2, or 4 bytes
  - ***Register寄存器*:**
    - Example: **`%rax, %r13`**
    - **`%rsp`** 保留有特殊用途
    - 有些指令中部分寄存器有特殊用途
  - ***Memory存储器*:** 8 consecutive bytes of memory at address given by register
    - **`(%rax)`**
    - 多种寻址模式

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`

注意: Intel 使用格式
mov *Dest, Source*

MOV 指令的操作数

	Source	Dest	Src,Dest	C Analog	
MOV	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`	
		Mem	`movq $-147,(%rax)`	`*p = -147;`	store
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`	
		Mem	`movq %rax,(%rdx)`	`*p = temp;`	store
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`	load

Cannot do memory-memory transfer with a single instruction

Data Formats

- Move data instruction
  - mov (general)
  - movb (move byte)
  - movw (move word)
  - movl (move double word)
  - movq (move quadruple word)

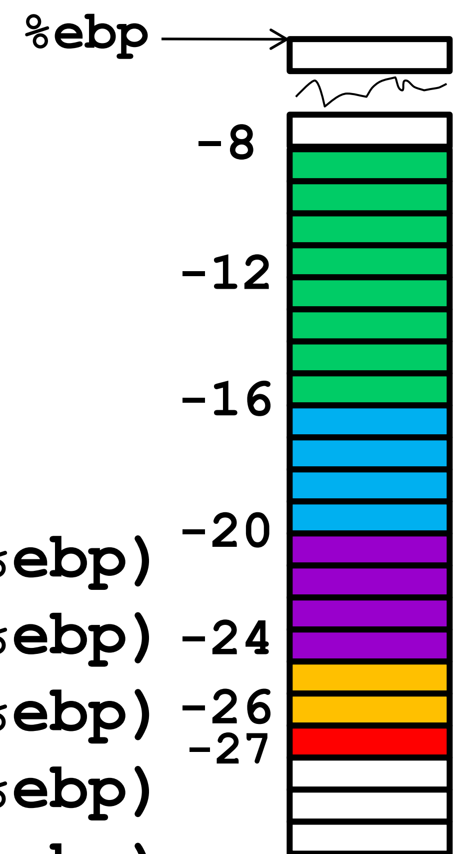
movl	$0x4050, %eax	immediate	register
movl	%ebp, %esp	register	register
movl	(%edx, %ecx), %eax	memory	register
movl	$-17, (%esp)	immediate	memory
movl	%eax, -12(%ebp)	register	memory

Data Movement

Instruction	Effect	Description
movl S, D	D ← S	Move double word
movw S, D	D ← S	Move word
movb S, D	D ← S	Move byte
movsbl S, D	D ← SignedExtend(S)	Move sign-extended byte
movzbl S, D	D ← ZeroExtend(S)	Move zero-extended byte
pushl S	R[%esp] ← R[%esp]-4 M[R[%esp]] ← S	Push
popl D	D ← M[R[%esp]] R[%esp] ← R[%esp]+4	Pop

Access Objects with Different Sizes

```
int main(void){
   char c = 1;
   short s = 2;
   int i = 4;
   long l = 4L;
   long long ll = 8LL;
   return;
}
8048335:c6 movb   $0x1,0xffffffe5(%ebp)
8048339:66 movw   $0x2,0xffffffe6(%ebp)
804833f:c7 movl   $0x4,0xffffffe8(%ebp)
8048346:c7 movl   $0x4,0xffffffec(%ebp)
804834d:c7 movl   $0x8,0xfffffff0(%ebp)
8048354:c7 movl   $0x0,0xfffffff4(%ebp)
```



汇编中的数组

存储数组首地址

```
void f(void){
    int i, a[16];
    for(i=0; i<16; i++)
        a[i]=i;
}

movl %edx,-0x44(%ebp,%edx,4)
a:   -0x44(%ebp)
i:   %edx
```

Data Movement Example

Initial value %dh=8d %eax =98765432

1	movb	%dh, %al	%eax=9876548d
2	movsbl	%dh, %eax	%eax=FFFFFF8d
3	movzbl	%dh, %eax	%eax=0000008d

- 1-byte registers-8个
  - %al, %ah, %cl, %ch, %dl, %dh, %bl, %bh
- 2-byte registers-8个
  - %ax, %cx, %dx, %bx, %si, %di, %sp, %bp

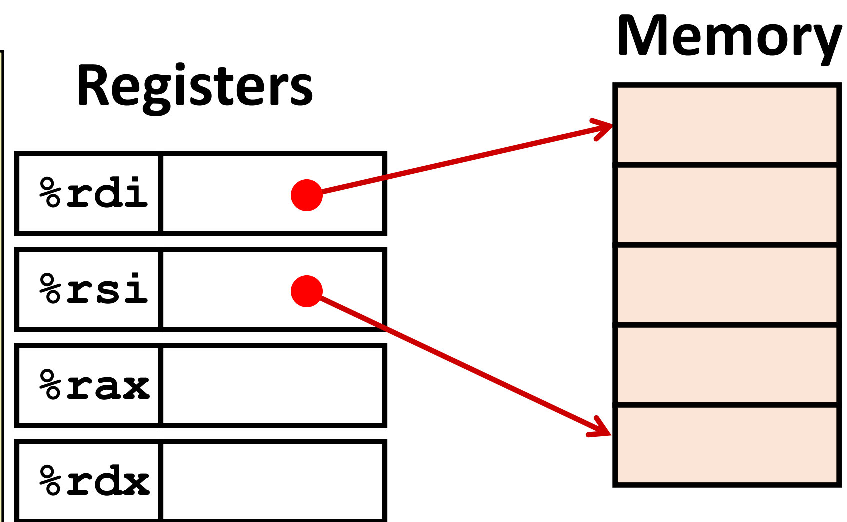
Example of Simple Addressing Modes

```
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
```

Understanding Swap()

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



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
```

Understanding Swap()

Registers

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

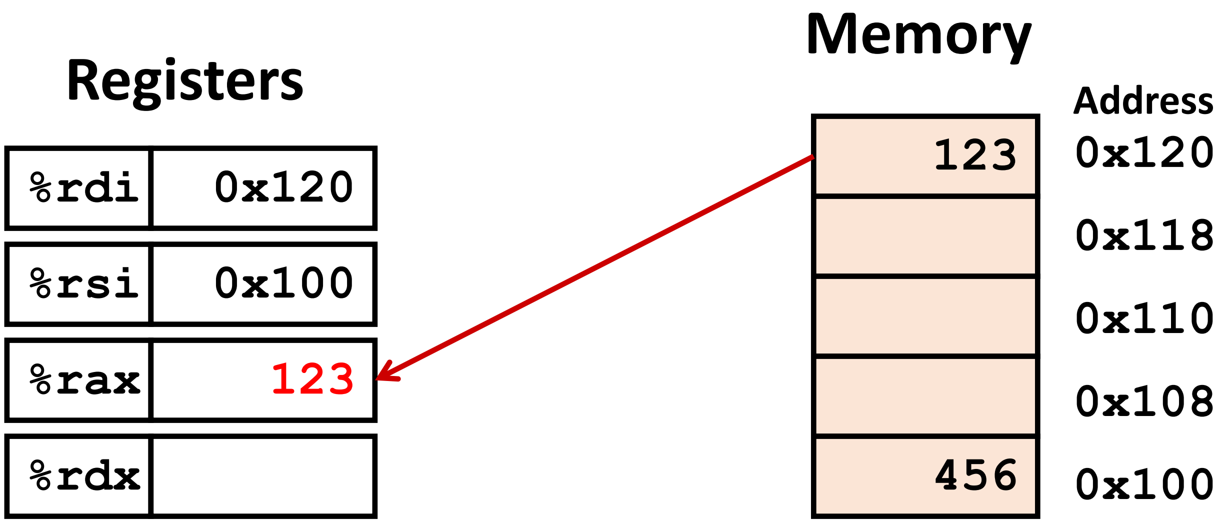
Memory

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

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

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

Understanding Swap()




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

Understanding Swap()

Registers

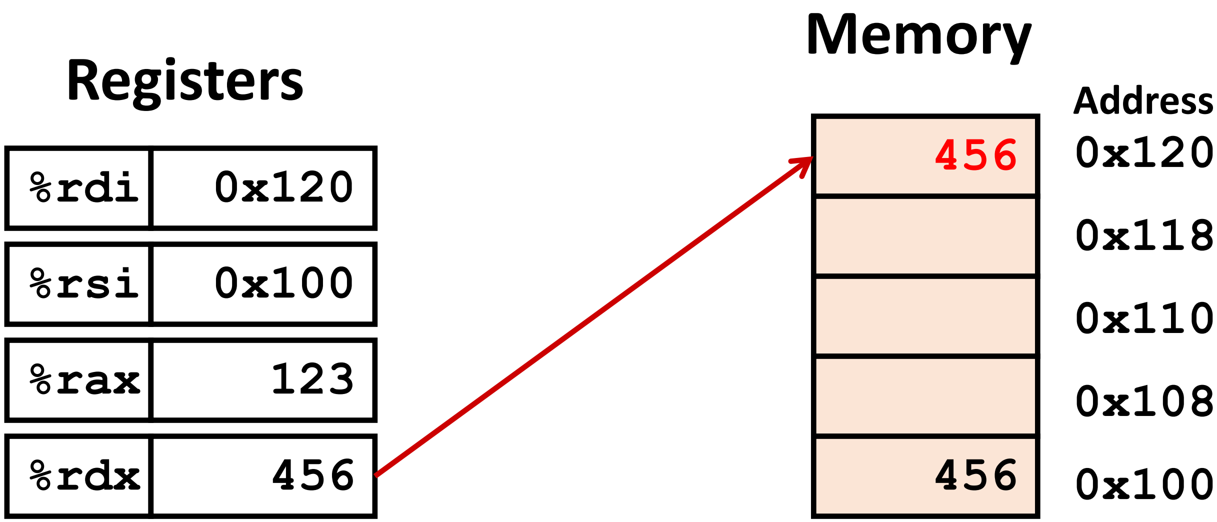
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory

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

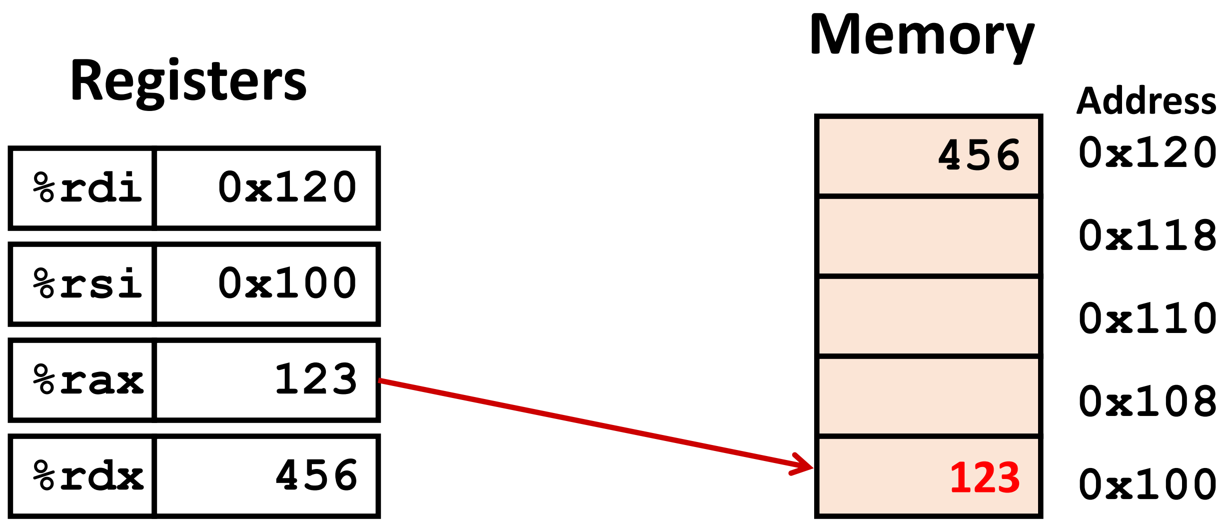
```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()




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

Understanding Swap()




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

MOV 指令遵循的规则

- 两个操作数的尺寸必须一致
- 两个操作数不能同时为内存操作数
- 目的操作数不能是CS，EIP和IP
- 立即数不能直接送至段寄存器
- 段寄存器之间不能直接传送数据
- 不能一次传送多个对象
- 标识寄存器（EFlags）和指令指针寄存器（IP）不能用MOV指令设置

课堂练习

- 下列汇编语句错在哪里？
  - mov $0xF, (%bl)
  - movl %ax, (%esp)
  - movw (%eax), 4(%esp)
  - movl %eax, %dx
  - movb %si, 8(%esp)
  - movb %ah, %sh
  - movl %eax, 0x123

主观题 10分

设置

已知原型为

Void decode1 (long *xp, long *yp, long *zp);

的函数编译成汇编，得到如下代码

```
# xp in %rdi, yp in %rsi, zp in %rdx
Decode1:
  Movq    (%rdi), %r8
  Movq    (%rsi), %rcx
  Movq    (%rdx), %rax
  Movq    %r8, (%rsi)
  Movq    %rcx, (%rdx)
  Movq    %rax, (%rdi)
```

请写出等效的C语言代码

作答

课堂练习

- 已知原型为
- Void decode1 (long *xp, long *yp, long *zp);
- 的函数编译成汇编，得到如下代码
- # xp in %rdi, yp in %rsi, zp in %rdx
- Decode1:
  - Movq (%rdi), %r8
  - Movq (%rsi), %rcx
  - Movq (%rdx), %rax
  - Movq %r8, (%rsi)
  - Movq %rcx, (%rdx)
  - Movq %rax, (%rdi)

```
long x = *xp;
long y = *yp;
long z = *zp;
*yp = x;
*zp = y
*xp = z;
```

- 请写出等效的C语言代码

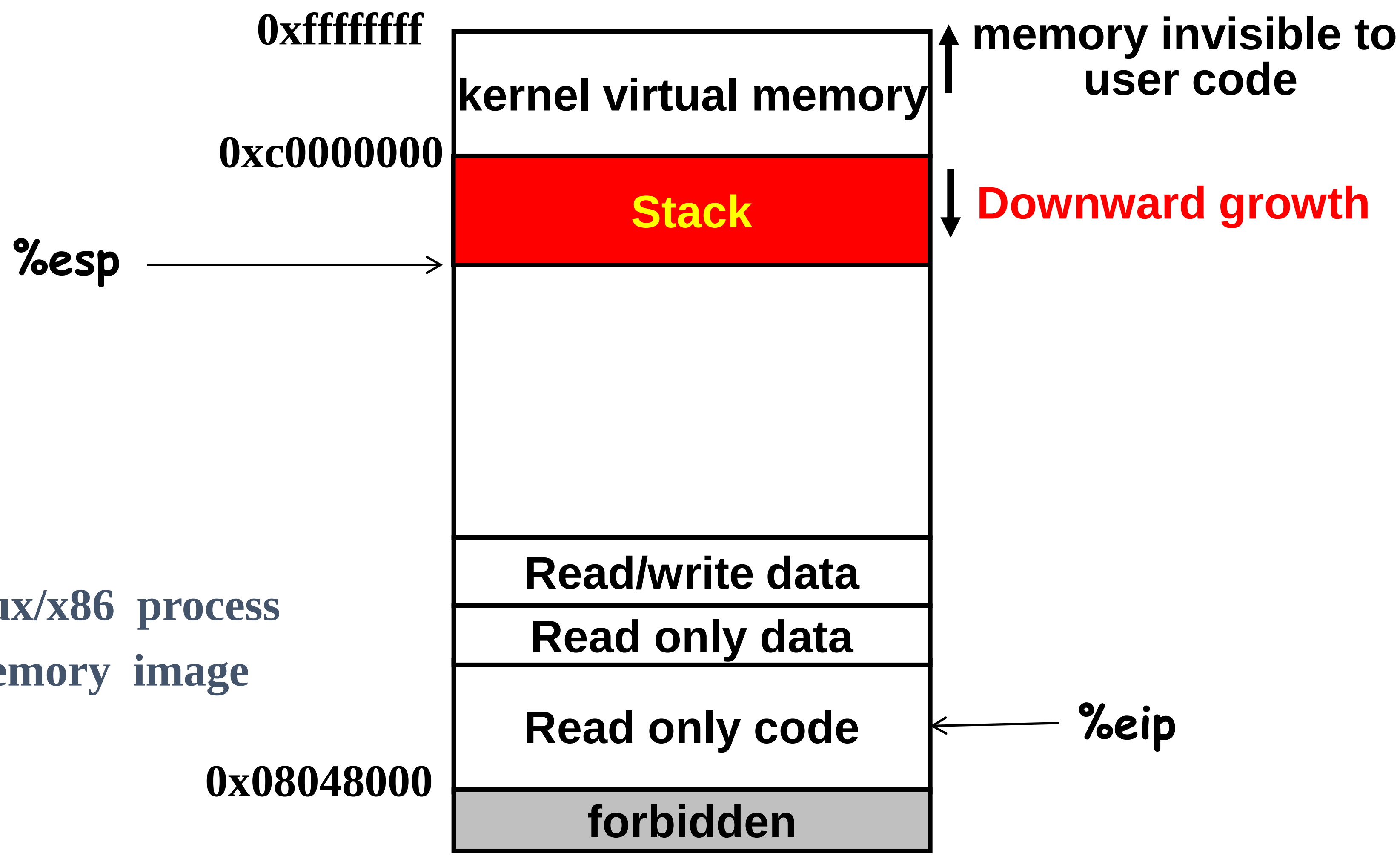
程序的机器级表示

- 如何生成汇编语言程序
- 汇编语言程序的结构
    - **.text 程序段**
    - .data 数据段
    - .stack 堆栈段

堆栈操作

- 栈是一种特殊的数据结构
  - 只能储存相同的数据类型
- 栈顶必须被明确指定
  - 表示为 top
- 栈中包含有两种操作
  - push 和 pop
- X86中存在硬件栈
  - 速度快，通过push, pop等指令操作
  - 栈底的地址更高
  - 栈顶由%esp表示

Stack Layout



Linux/x86 process memory image

Stack Operation

Instruction	Effect
pushl S	decreases the %esp (enlarge the stack) stores the value in a register into the stack: R[%esp] ← R[%esp]-4 M[R[%esp]] ← S
popl D	stores the value in the top of the stack into a register increases the %esp (shrink the stack): D ← M[R[%esp]] R[%esp] ← R[%esp]+4

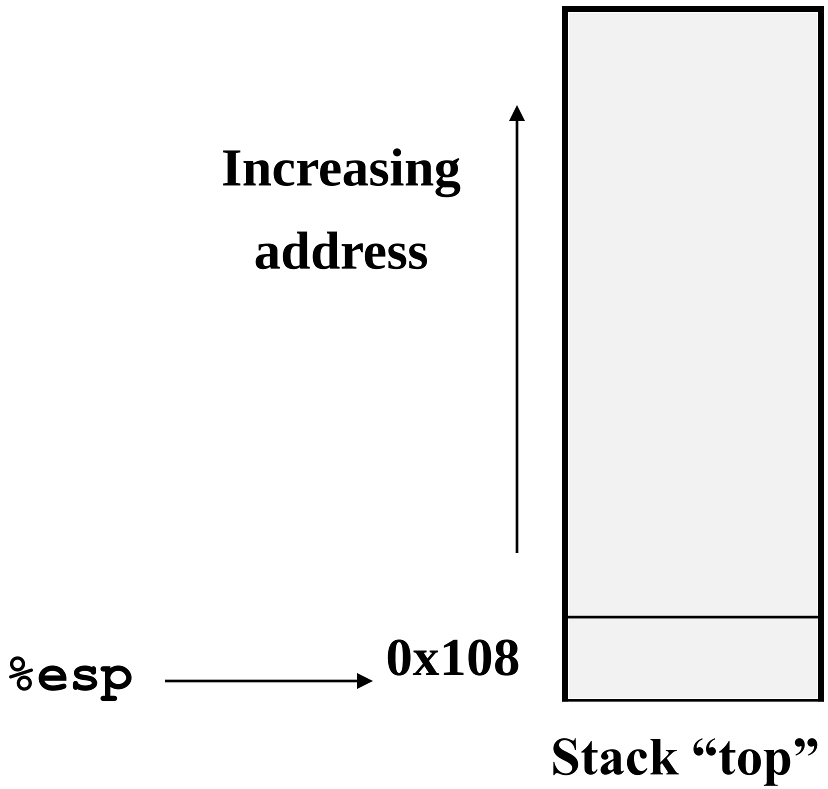
压栈时，
先修改栈指针，
后压入数据

弹栈时，
先弹出数据，
后修改栈指针

Stack operations

%eax	0x123
%edx	0
%esp	0x108

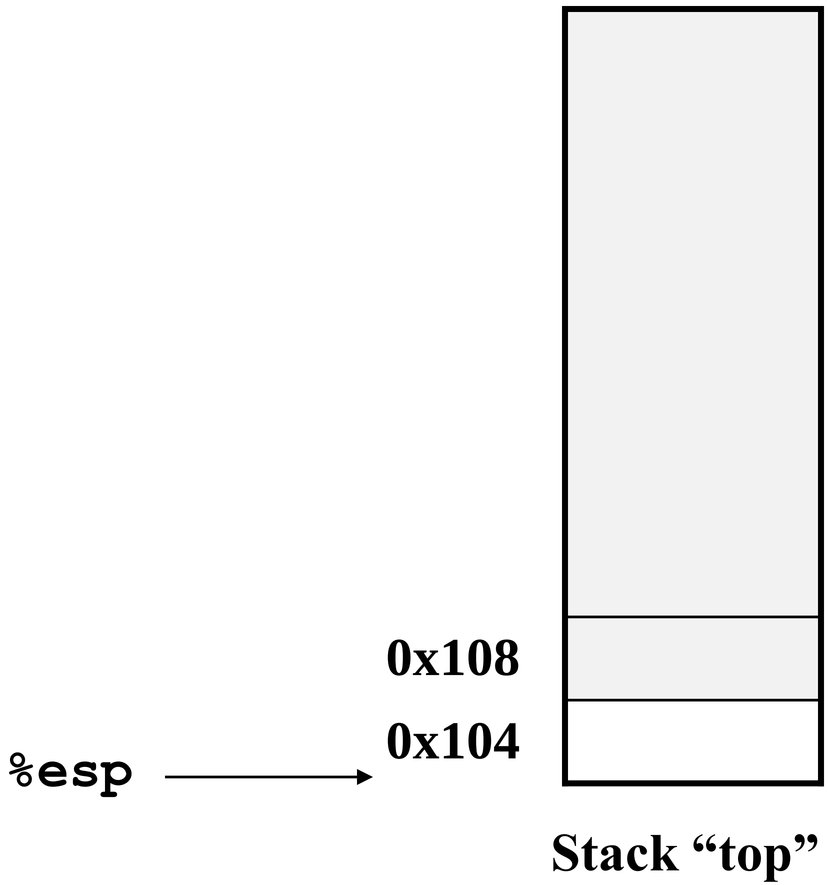
pushl %eax ?



Stack operations

%eax	0x123
%edx	0
%esp	0x104

pushl %eax

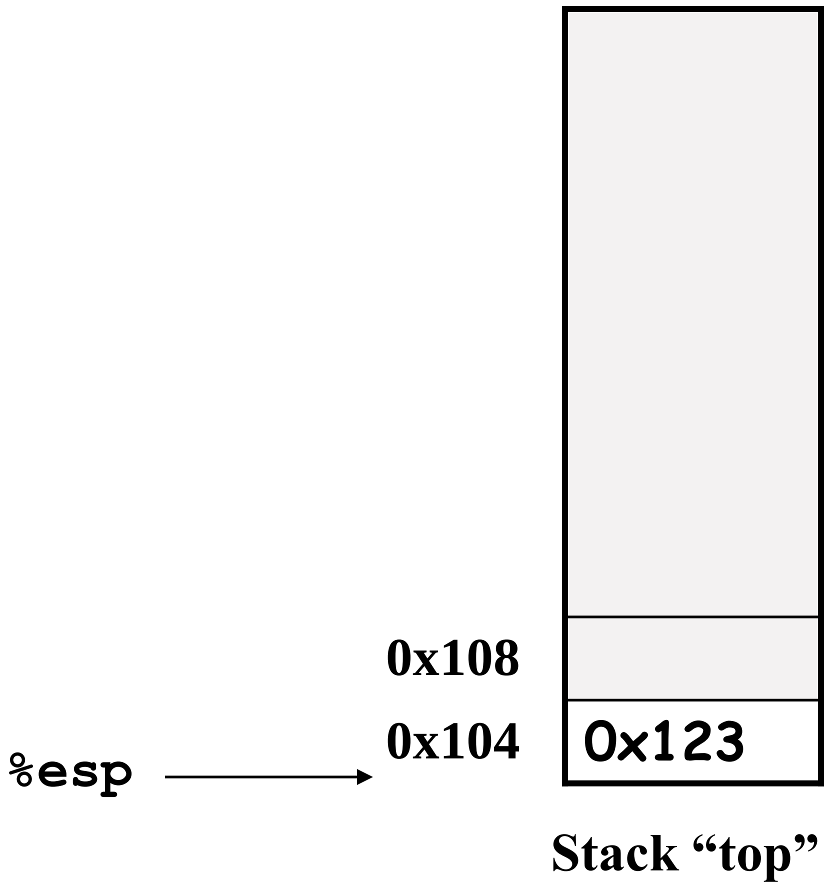


Stack operations

%eax	0x123
%edx	0
%esp	0x104

pushl %eax

popl %edx ?

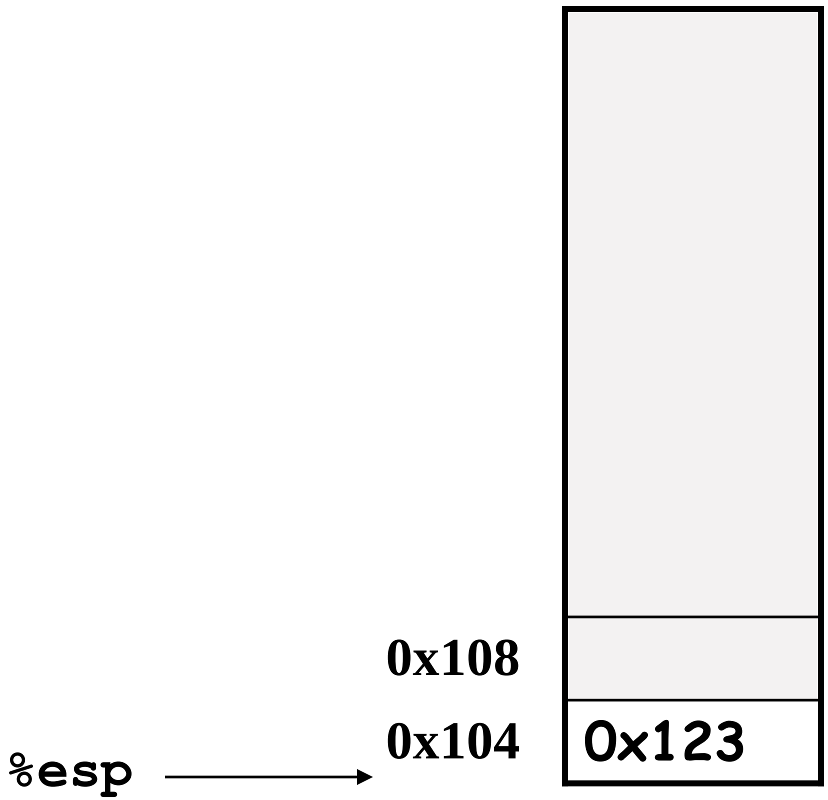


Stack operations

%eax	0x123
%edx	0x123
%esp	0x104

pushl %eax

popl %edx ?

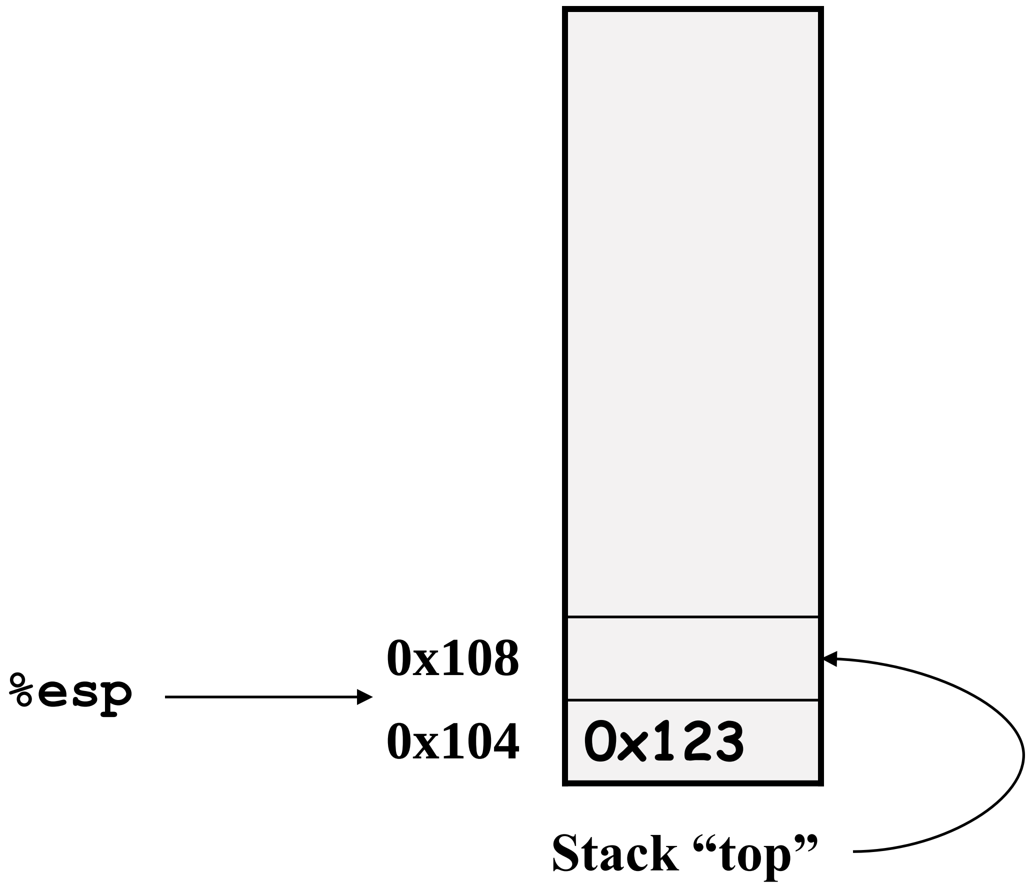


Stack “top”

Stack operations

%eax	0x123
%edx	0x123
%esp	0x108

popl %edx



Pointers in C

- 在C中，一个对象可以是如下形式：
  - 整数对象
  - 结构对象
  - 其它程序单元(如浮点数等)
- 声明指针

```
T *p;
```

C中的指针

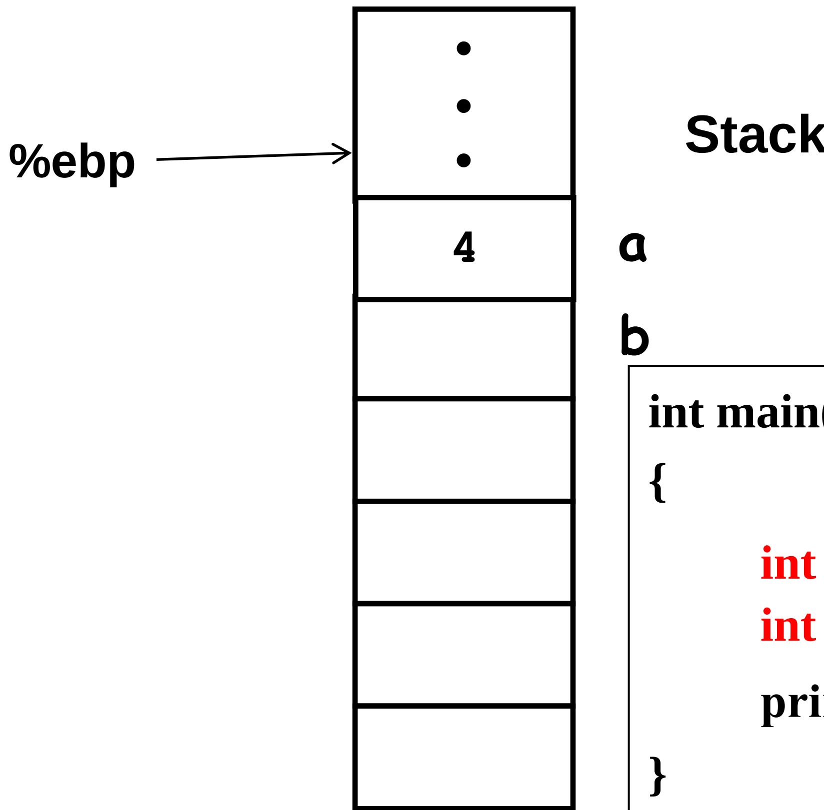
- C中指针的值满足以下条件
  - 是部分储存块虚拟地址的首字节
  - 类别信息用于以下操作：
    - 确定对象的长度
    - 解引用 *p
- 生成指针

  p = &obj

Example

```
int main()
{
    int a = 4 ;
    /* "address of" operator creates a pointer */
    int b = exchange(&a, 3);
    printf("a = %d, b = %d\n", a, b);
}
```

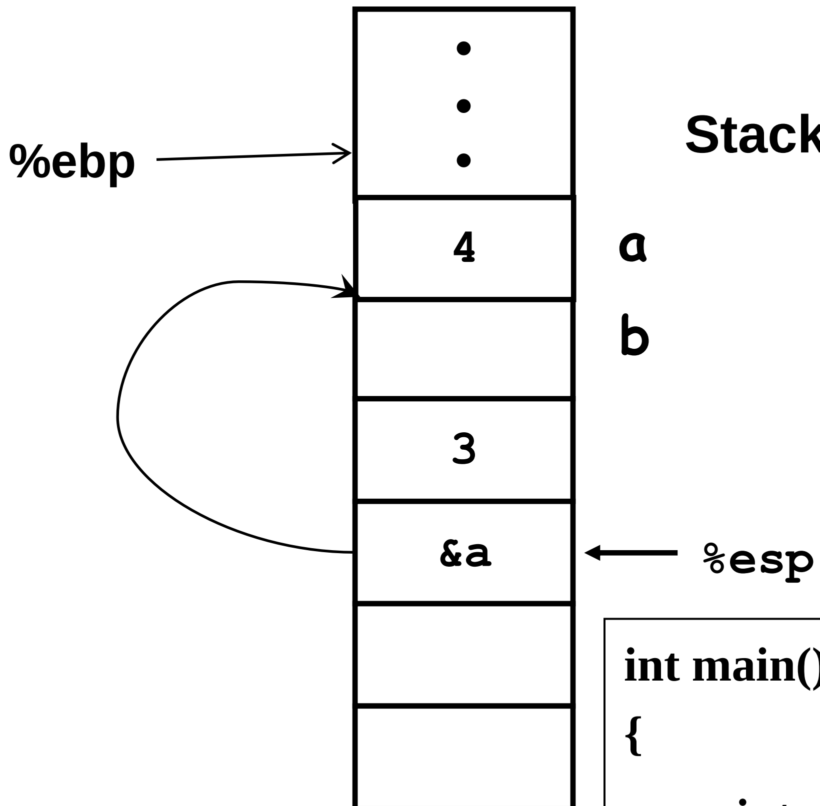
Example




```
int main()
{
        int a = 4 ;
        int b = exchange(&a, 3);
        printf("a = %d, b = %d\n", a, b);
}
```

Can the variable a be in a register?

Example




```
int main()
{
    int a = 4 ;
    int b = exchange(&a, 3);
    printf("a = %d, b = %d\n", a, b);
}
```

Example

```
int exchange(int *xp, int y)
{
     /* operator * performs deferencing */
    int x = *xp ;
    *xp = y ;
    return x ;
}
```

Example

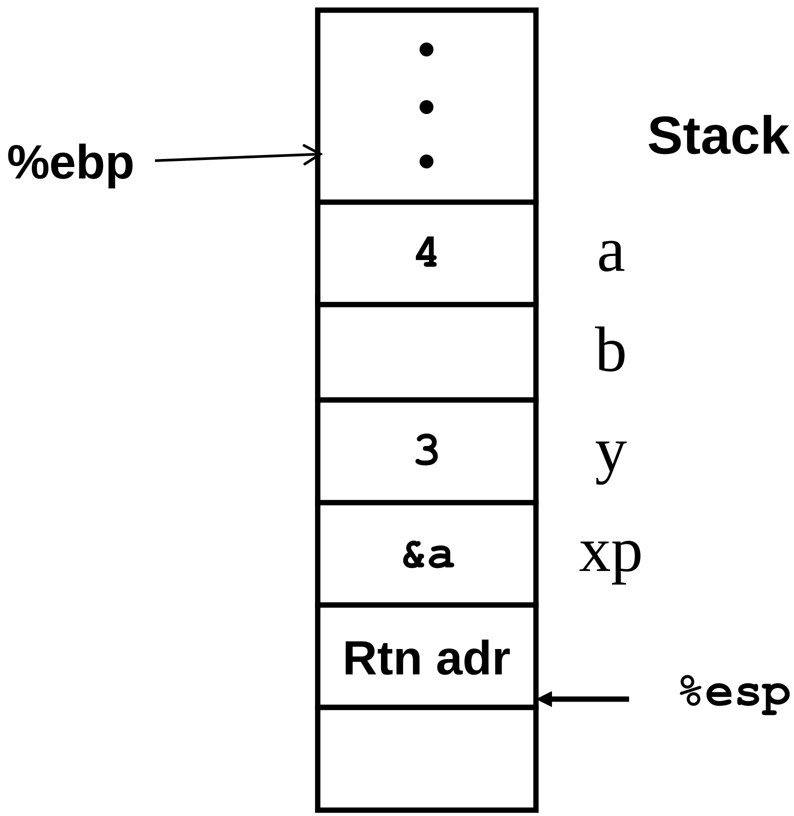
```
int exchange(int *xp, int y)
{
    int x = *xp ;

    *xp = y ;
    return x ;
}
```

```
1 pushl  %ebp
2 movl   %esp, %ebp
3 movl   8(%ebp),  %eax #xp
4 movl   12(%ebp), %edx #y
5 movl   (%eax), %ecx # x
6 movl   %edx,   (%eax)
7 movl   %ecx,   %eax
8 movl   %ebp, %esp
9 popl   %ebp
```

Example

在pushl之前call调用就会存入Rtn adr（主函数返回处）



1 pushl %ebp

2 movl %esp, %ebp

3 movl 8(%ebp), %eax #xp

4 movl 12(%ebp), %edx #y

5 movl (%eax), %ecx # x

6 movl %edx, (%eax)

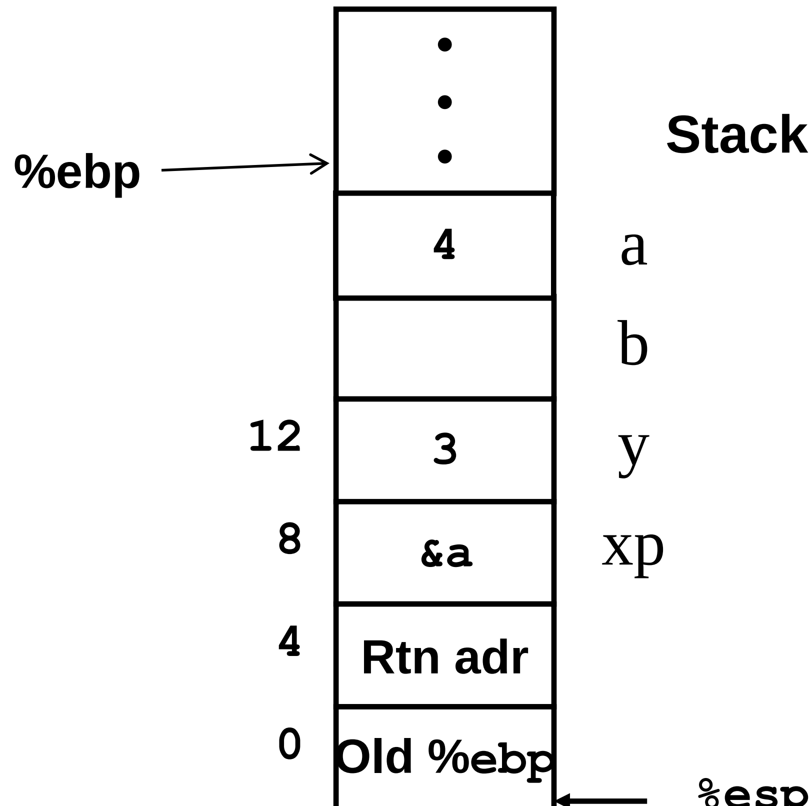
7 movl %ecx, %eax

8 movl %ebp, %esp

9 popl %ebp

Example

1 pushl %ebp



1 pushl %ebp

2 movl %esp, %ebp

3 movl 8(%ebp), %eax #xp

4 movl 12(%ebp), %edx #y

5 movl (%eax), %ecx # x

6 movl %edx, (%eax)

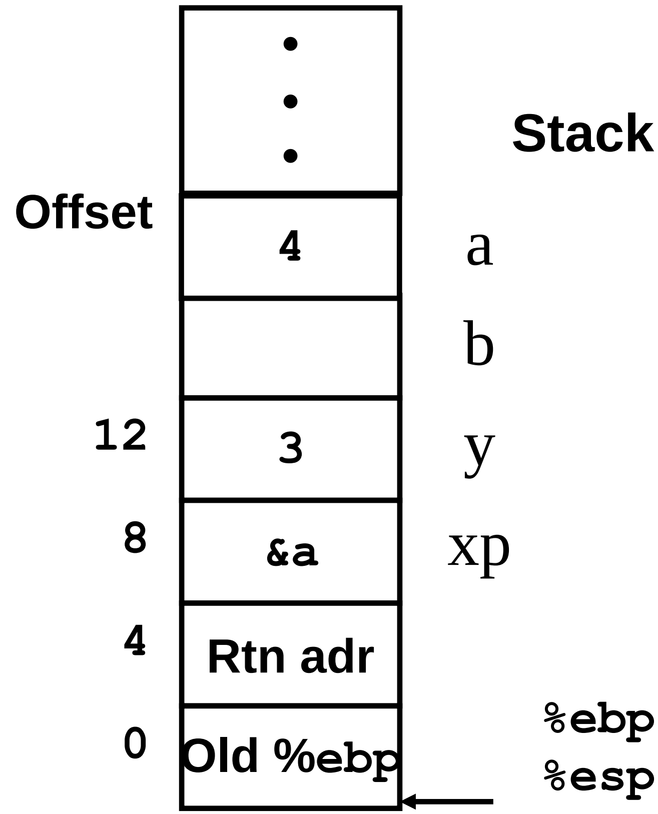
7 movl %ecx, %eax

8 movl %ebp, %esp

9 popl %ebp

Example

2 movl %esp, %ebp




```
1 pushl  %ebp
2 movl   %esp, %ebp
3 movl   8(%ebp),  %eax #xp
4 movl   12(%ebp), %edx #y
5 movl   (%eax), %ecx # x
6 movl   %edx,   (%eax)
7 movl   %ecx,   %eax
8 movl   %ebp, %esp
9 popl   %ebp
```

Example

3 movl 8(%ebp), %eax

%eax: xp

1 pushl %ebp

2 movl %esp, %ebp

3 movl 8(%ebp), %eax #xp

4 movl 12(%ebp), %edx #y

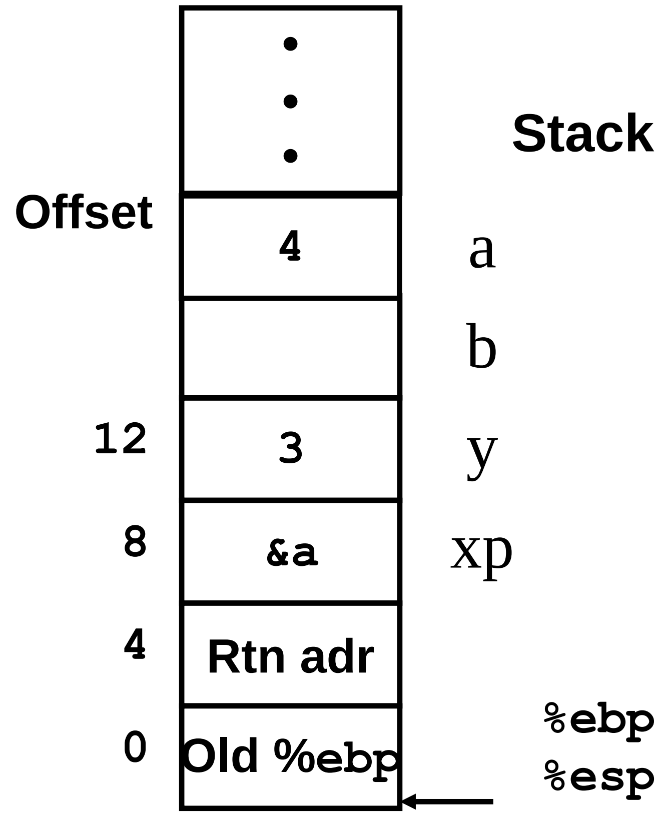
5 movl (%eax), %ecx # x

6 movl %edx, (%eax)

7 movl %ecx, %eax

8 movl %ebp, %esp

9 popl %ebp



Example

3 movl 8(%ebp), %eax

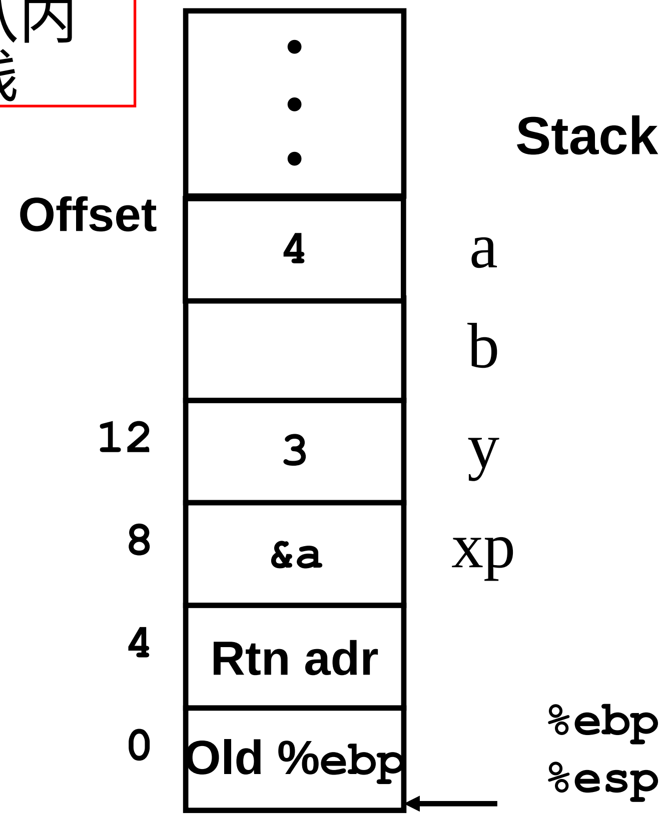
4 movl 12(%ebp), %edx

可以理解为主函数本身需要一个栈维护，a的值已经从内存push进栈

%eax: xp

%edx: 3

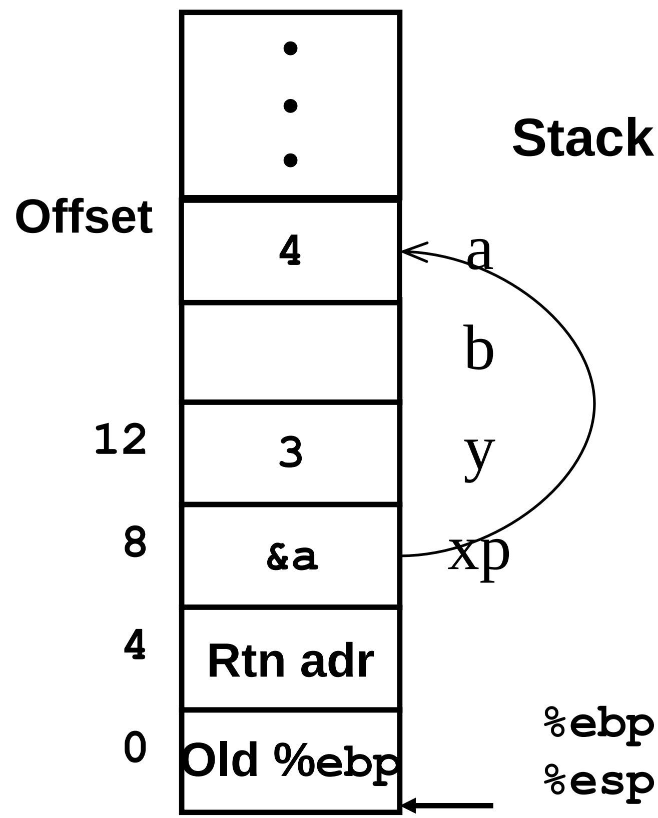
```
1 pushl  %ebp
2 movl   %esp, %ebp
3 movl   8(%ebp),  %eax #xp
4 movl   12(%ebp), %edx #y
5 movl   (%eax), %ecx # x
6 movl   %edx,   (%eax)
7 movl   %ecx,   %eax
8 movl   %ebp, %esp
9 popl   %ebp
```



Example

3 movl 8(%ebp), %eax
4 movl 12(%ebp), %edx
5 movl (%eax), %ecx

%eax: xp
%edx: 3
%ecx: 4

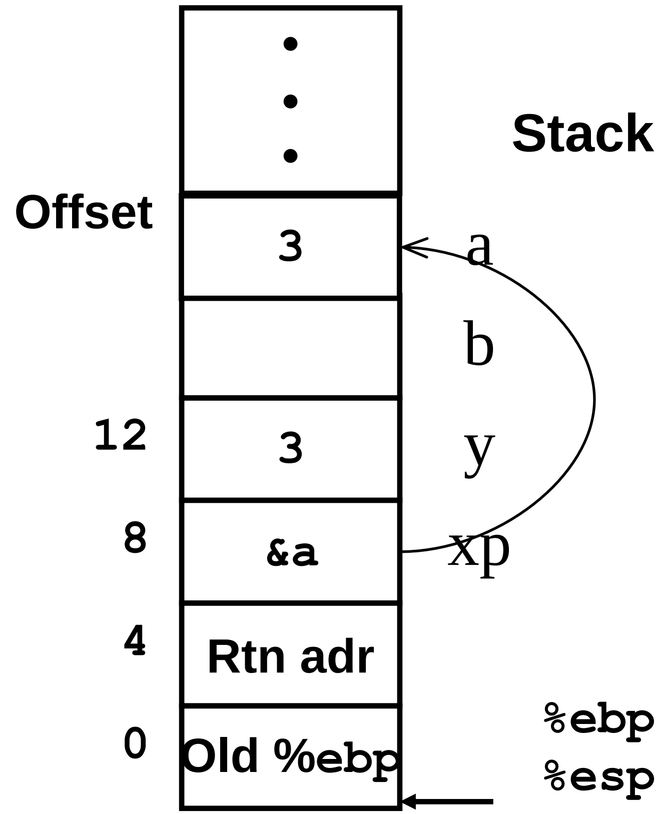



```
1 pushl  %ebp
2 movl   %esp, %ebp
3 movl   8(%ebp),  %eax #xp
4 movl   12(%ebp), %edx #y
5 movl   (%eax), %ecx # x
6 movl   %edx,   (%eax)
7 movl   %ecx,   %eax
8 movl   %ebp, %esp
9 popl   %ebp
```

Example

```
3 movl    8(%ebp),  %eax
4 movl    12(%ebp), %edx
5 movl    (%eax),   %ecx
6 movl    %edx,     (%eax)
```

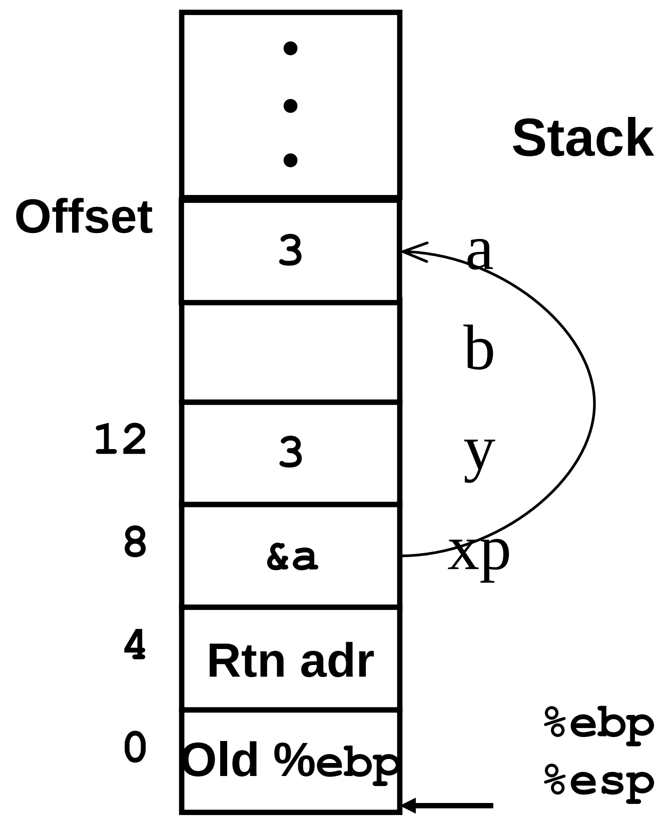
%eax: xp
%edx: 3
%ecx: 4
*xp(a): 3



Data Movement Example

3 movl 8(%ebp), %eax
4 movl 12(%ebp), %edx
5 movl (%eax), %ecx
6 movl %edx, (%eax)
7 movl %ecx, %eax

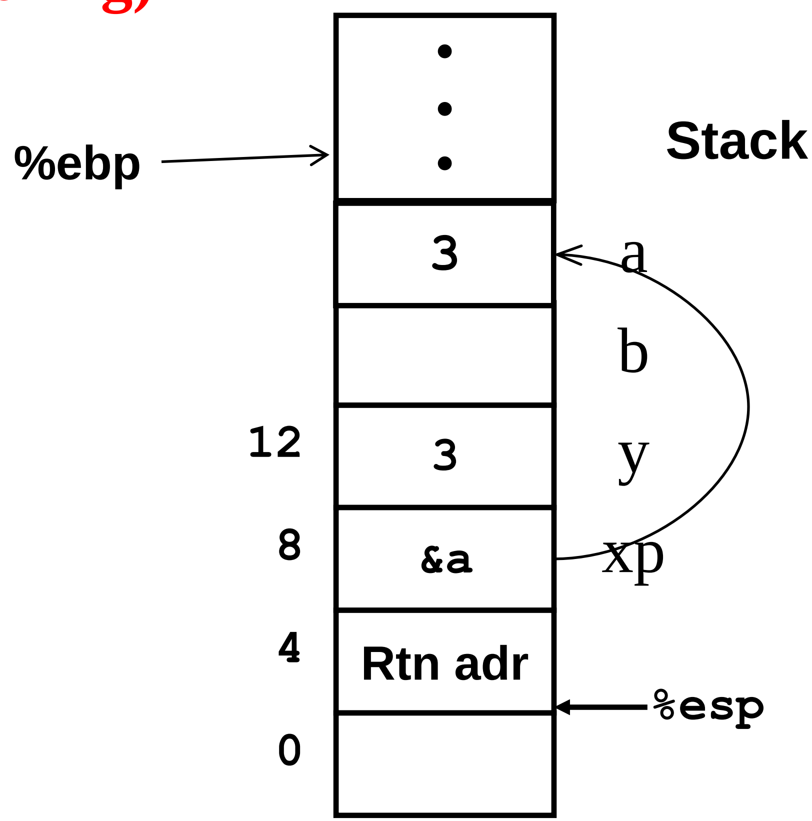
%eax: xp
%edx: y
%ecx: 4
*xp(a): 3
%eax: 4 (old *xp) return value



Data Movement Example

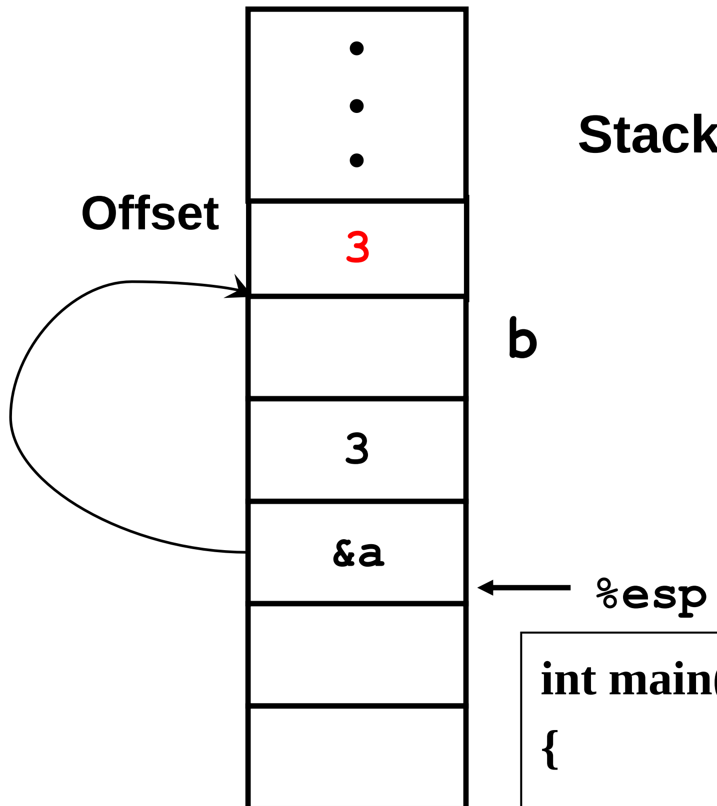
8 movl %ebp, %esp **(do nothing)**

9 popl %ebp




```
1 pushl  %ebp
2 movl   %esp, %ebp
3 movl   8(%ebp),  %eax #xp
4 movl   12(%ebp), %edx #y
5 movl   (%eax), %ecx # x
6 movl   %edx,   (%eax)
7 movl   %ecx,   %eax
8 movl   %ebp, %esp
9 popl   %ebp
```

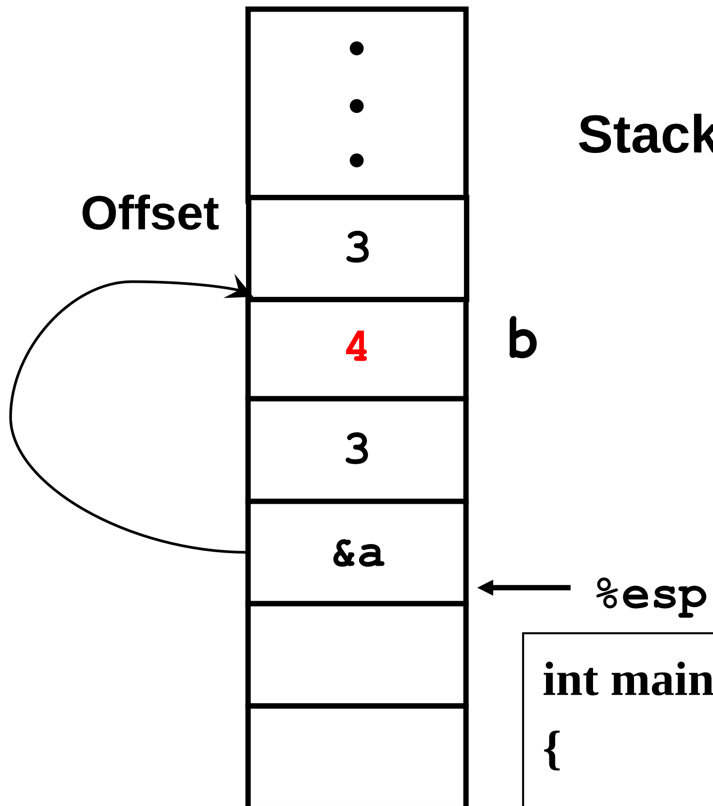
Example



%eax 4
Return value

```
int main()
{
        int a = 4 ;
        int b = exchange(&a, 3);
        printf("a = %d, b = %d\n", a, b);
}
```

Example




```
int main()
{
        int a = 4 ;
        int b = exchange(&a, 3);
        printf("a = %d, b = %d\n", a, b);
}
```

语法

- Intel格式
  - 寄存器参数：

    dst, src (←)
  - Len equ $ - msg
  - Len = $ - msg

Intel 语法	AT&T 语法
Mov eax,8	movl $8,%eax
Mov ebx,0ffffh	movl $0xffff,%ebx
int 80h	int $0x80

gcc –O1 –S –masm=intel code.c可以生成intel风格的汇编代码

- AT&T格式
  - 寄存器参数：

    src, dst (→)
  - 寄存器%
  - 常量、变量$
  - Len = . – msg
  - 加长度后缀
    - movb传送字节
    - movw传送字
    - movl传送双字

      (long word)
    - movq传送四字