程序的机器级表示（2）

谢旻晖

2025.10

C and Assembly Language

Characteristics of the high level programming languages

- 抽象
  - 语句表达能力强（指令型/描述性）
  - 可靠
- 类型检查
- 像手写底层代码一样高效
- 能够编译，并在不同的机器上执行

Characteristics of the assembly programming languages

- 直接管理内存
- 低层次指令来进行计算
- 高度机器相关：
  - 特定CPU硬件->特定指令集（机器语言）->特定汇编语言
  - 例如x86汇编，MIPS汇编
  - Intel在PC和服务器领域占绝对统治地位
    - AMD也是x86架构
    - IBM等大型机已经式微（只在金融等领域应用）
    - ARM在嵌入式领域应用还是很广泛，不过汇编大同小异

Why should we understand the assembly code

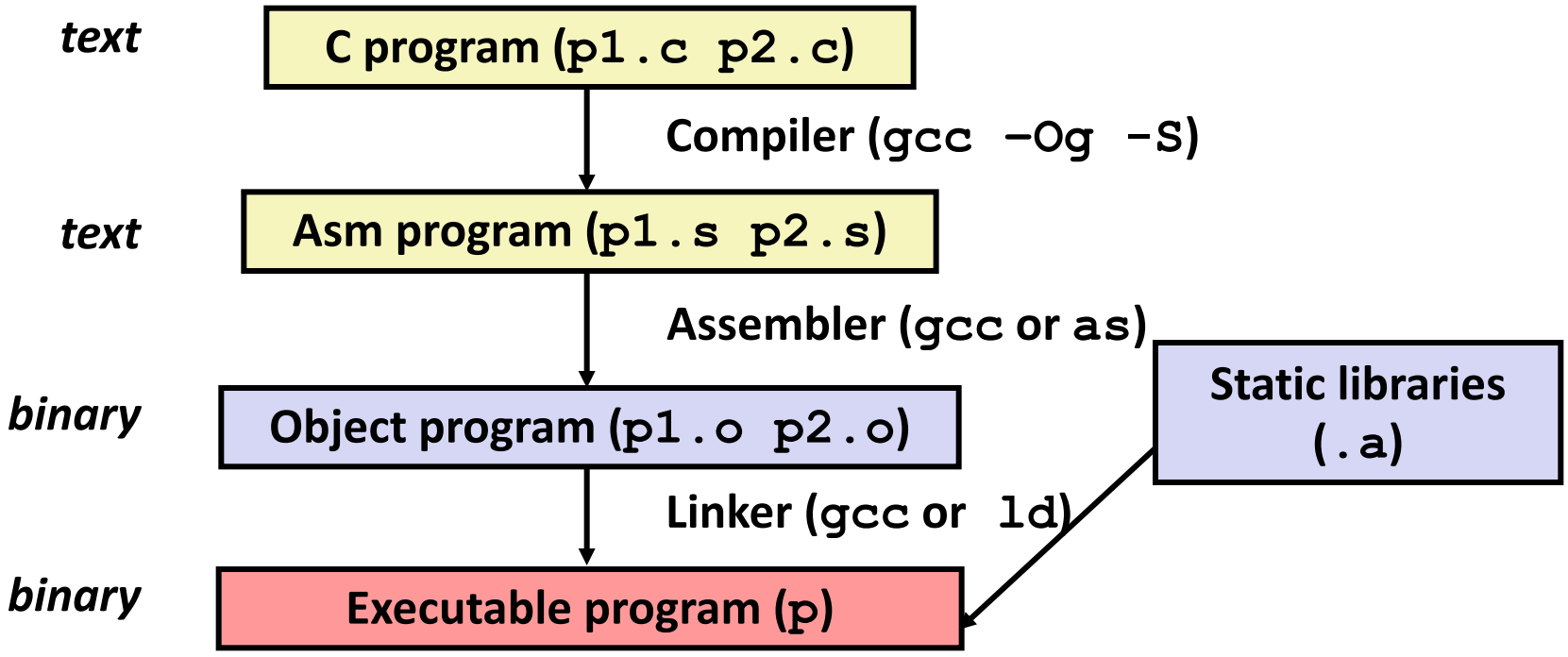
- 理解编译器的优化能力
  - 不能过于贬低：比绝大多数人工强
  - 也不能过于夸大：
    - 有些看似简单的地方，为了避免bug，不敢优化
    - 有些复杂问题还是很难优化（例如并行编译）
- 分析代码中低效的部分，以便提升程序性能
  - 汇编与性能直接对应

From writing assembly code 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
- 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

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

```
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.**

Disassembling Object Code

Disassembled

```
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
```

- Disassembler

  `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

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    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

Code Examples

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

上次讲到这里

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

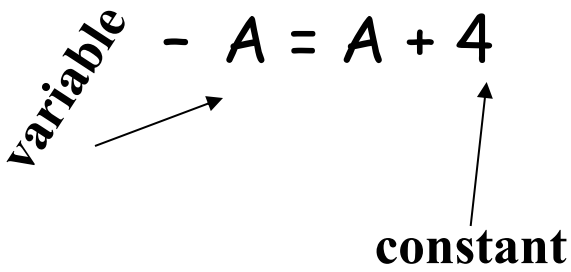
```
_sum:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%eax
        addl 8(%ebp),%eax
        addl %eax, accum
        movl %ebp,%esp
        popl %ebp
        ret
```

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

Operands

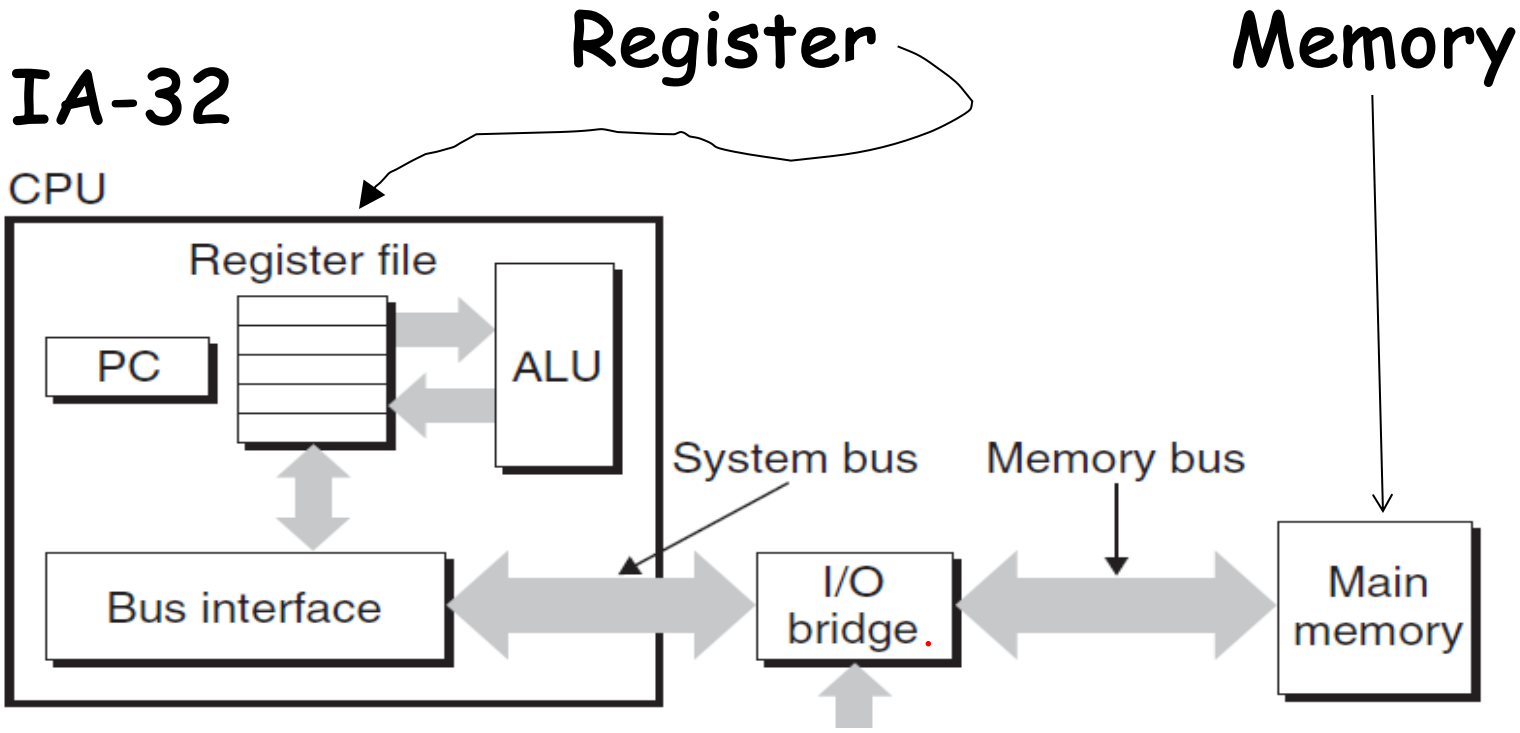
- In high level languages
  - Either constants
  - Or variable
- Example
  - A = A + 4



Assembly Code

- Operands:
  - x: Register %eax
  - y: Memory M[%ebp+8]
  - 4: Immediate $4

IA-32



The fastest storage units in computer systems, 32-bit long

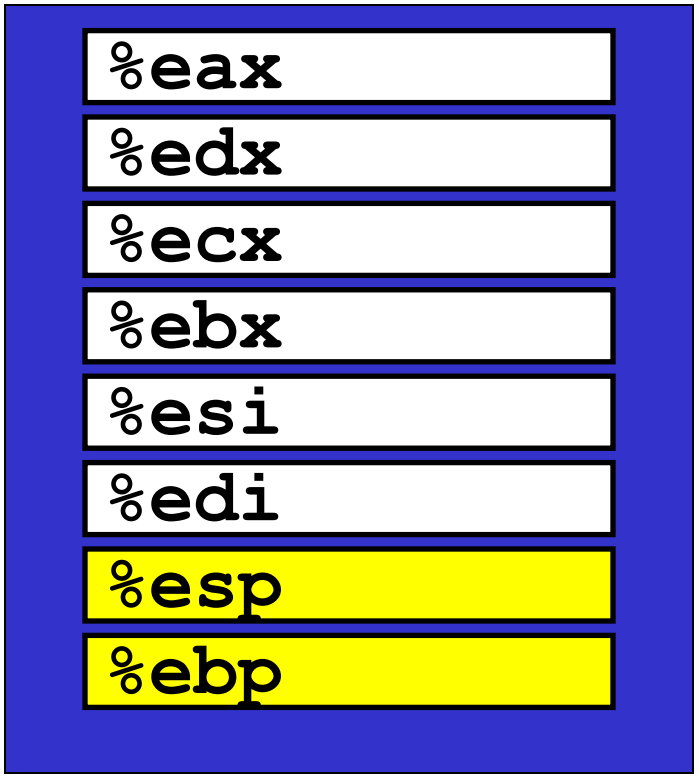
Register& Memory: Name & Value

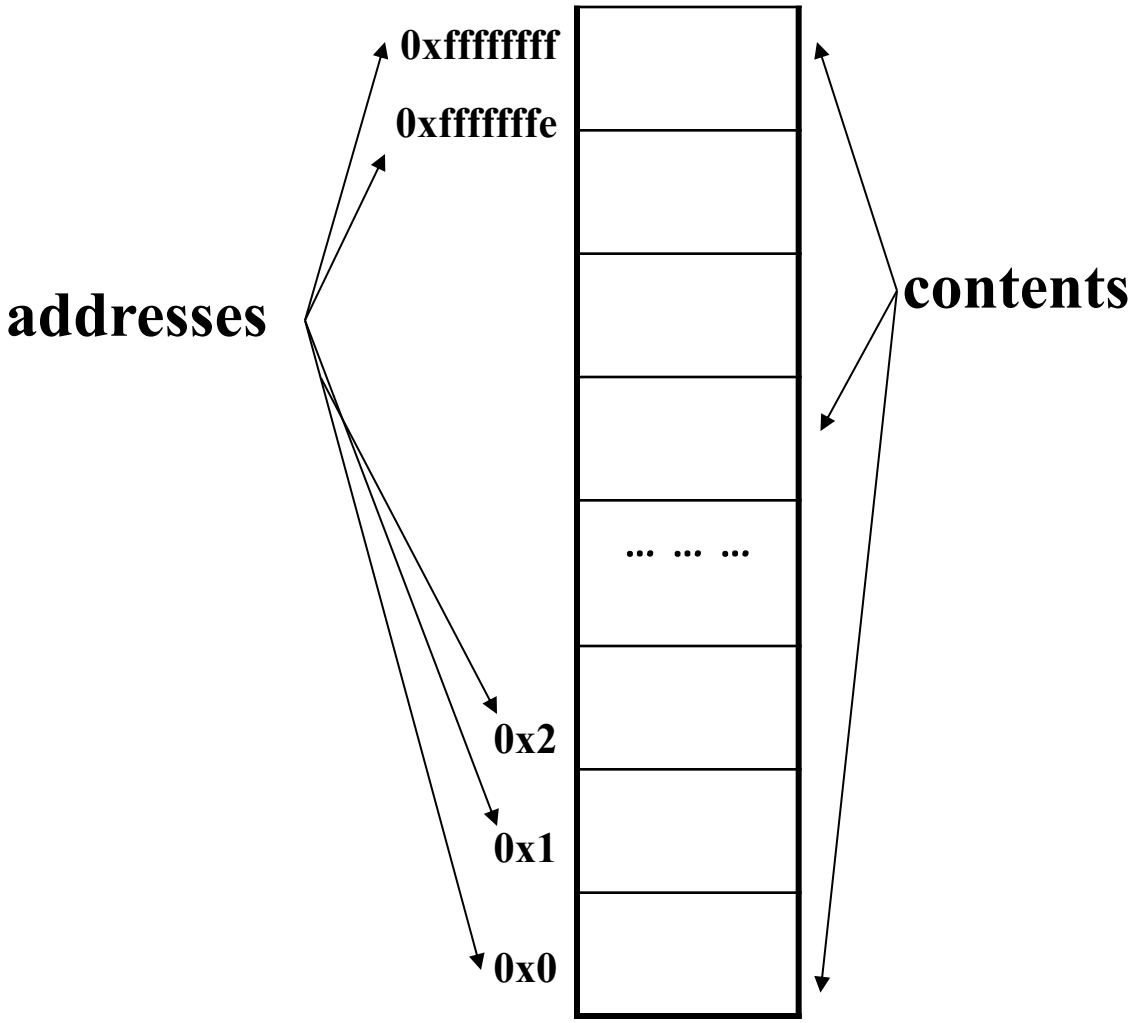
- Two important concepts
  - Name and value
- WRITE(*name, value*) value ← READ(*name*)
- WRITE operation specifies
  - a value to be remembered
  - a name by which one can recall that value in the future
- READ operation specifies
  - the name of some previous remembered value
  - the memory device returns that value

Registers vs. Virtual Memory

- How to name registers
  - Using specific names,
  - For example, in IA-32
    - %eax, %ecx, %edx, %ebx, %esi, %edi, %esp, %ebp
- How to name virtual memory
  - Using address as we have studied
- What's the difference
  - Accessing values in Registers is fast
  - Number of the registers is small
  - Most modern instructions can access registers only（RISC）

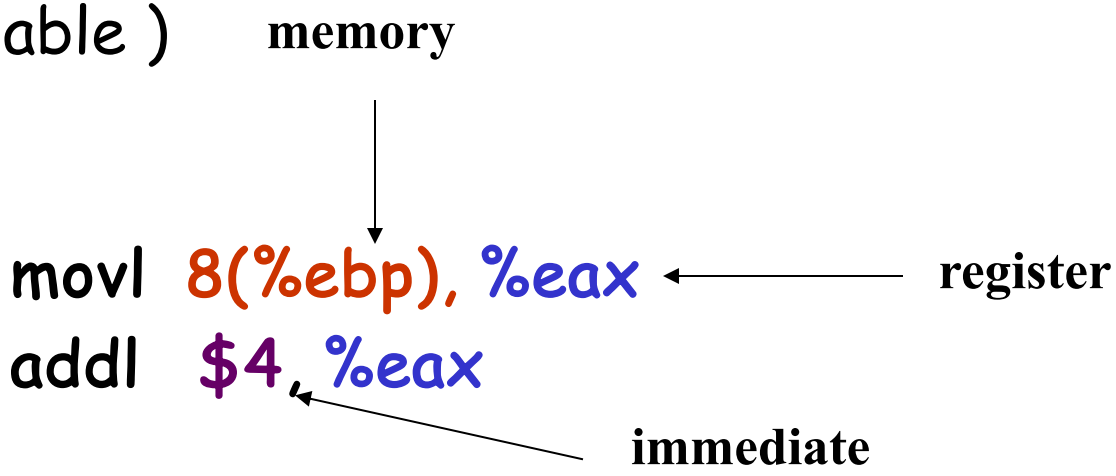
Where are the variables? — registers & Memory





Operands

- Counterparts in assembly languages
  - Immediate (constant)
  - Register (variable)
  - Memory (variable)
- Example

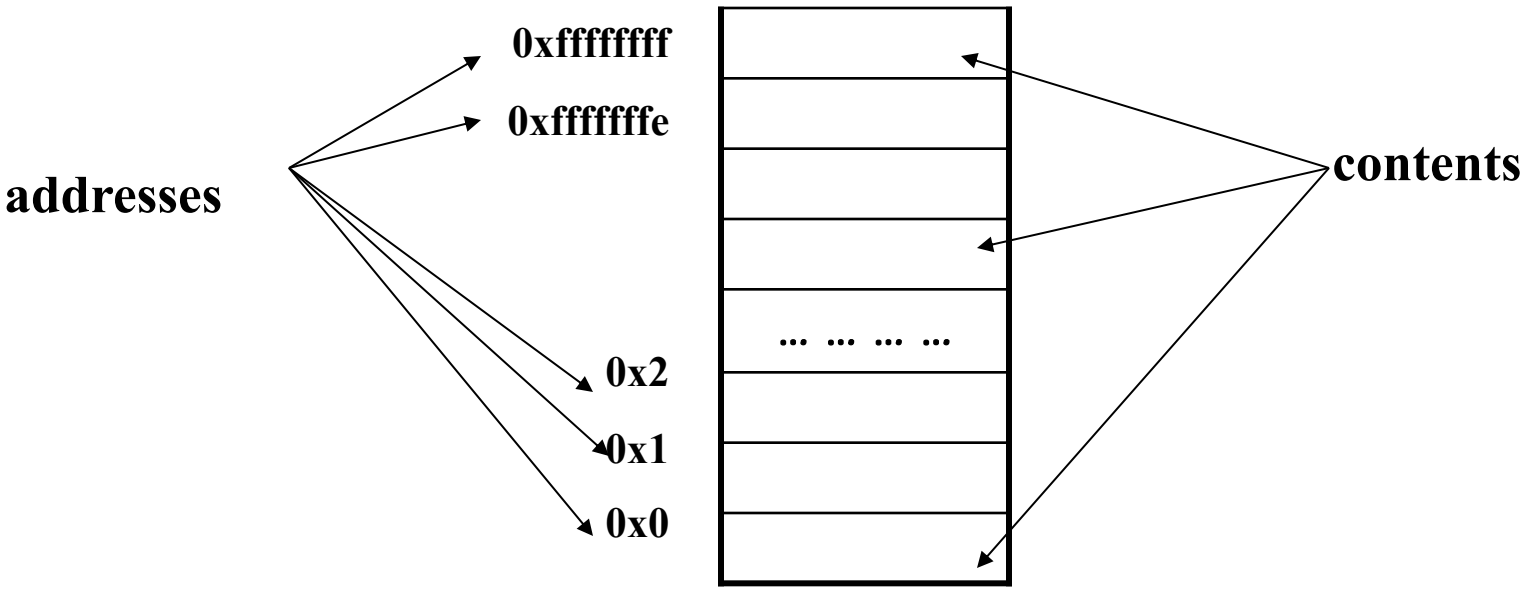


Express Operands in Assembly (Addressing Mode)

- Immediate
  - represents a constant
  - The format is $imm ($4, $0xffffffff)
- Registers
  - Register mode E_a
    - %eax
    - The value stored in the register %eax
      - Noted as R[E_a] (R[%eax])

Virtual spaces

- A linear array of bytes
  - each with its own unique address (array index) starting at zero



Memory References

- The name of the array is annotated as *M*
- If *addr* is a memory address
- *M*[*addr*] is the content of the memory starting at *addr*
- *addr* is used as an array index
- How many bytes are there in *M*[*addr*]?
  - It depends on the context

Indexed Addressing Mode

- An expression for
  - a memory address (or an array index)
- Most general form
  - $Imm(E_b, E_i, s)$
  - Constant "displacement" Imm: 1, 2 or 4 bytes
  - Base register E_b: Any of 8 integer registers
  - Index register E_i : Any, except for `%esp`
  - `S`: Scale: 1, 2, 4, or 8

Memory Addressing Mode（寻址方式）

- $Imm(E_b, E_i, s)$

- The address represented by the above form
  - $imm + R[E_b] + R[E_i] * s$
- It gives the value
  - $M[imm + R[E_b] + R[E_i] * s]$
- E.g., `260(%ecx,%edx,2)`
  - `M[260+%ecx+2*%edx]`

Addressing Mode（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
Indexed	(E_a)	$M[R[E_a]]$	Indirect
Indexed	$Imm(E_b)$	$M[Imm+ R[E_b]]$	Base+displacement
Indexed	(E_b, E_i)	$M[R[E_b]+ R[E_i]]$	Indexed
Indexed	$Imm(E_b, E_i)$	$M[Imm+ R[E_b]+ R[E_i]]$	Scaled indexed
Indexed	$(, E_i, s)$	$M[R[E_i]*s]$	Scaled indexed
Indexed	(E_b, E_i, s)	$M[R[E_b]+ R[E_i]*s]$	Scaled indexed
Indexed	$Imm(E_b, E_i, s)$	$M[Imm+ R[E_b]+ R[E_i]*s]$	Scaled indexed

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

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

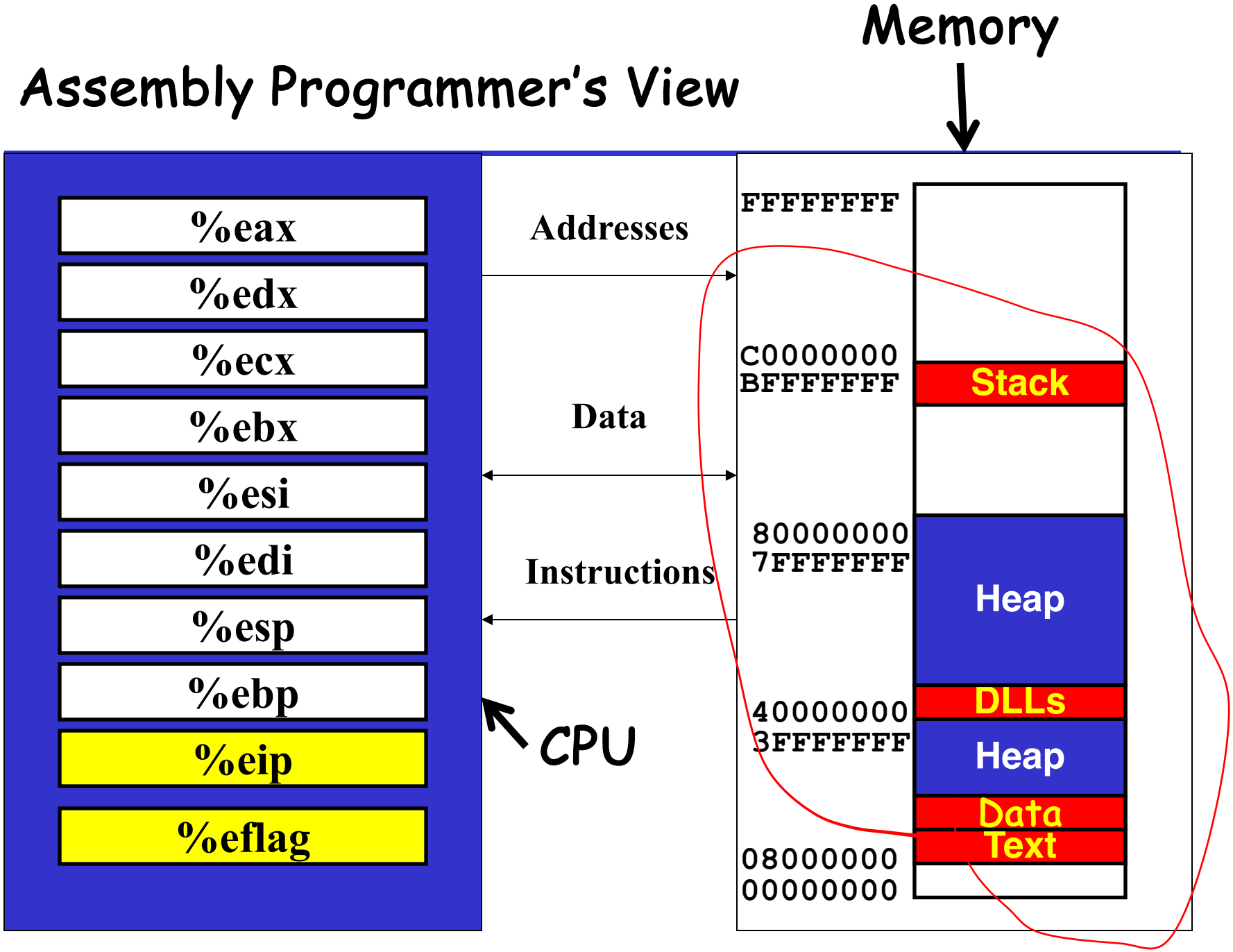
Operand	Value
%eax	0x100
(%eax)	0xFF
$0x108	0x108
0x108	0x13
260(%ecx,%edx)	(0x108) 0x13
(%eax,%edx,4)	(0x10C) 0x11

Understanding Machine Execution

Outline

- Machine states
- Machine execution
- Virtual Memory Layout

Assembly Programmer's View

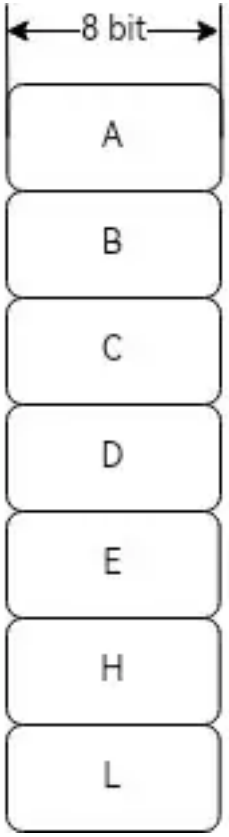


Programmer-Visible States

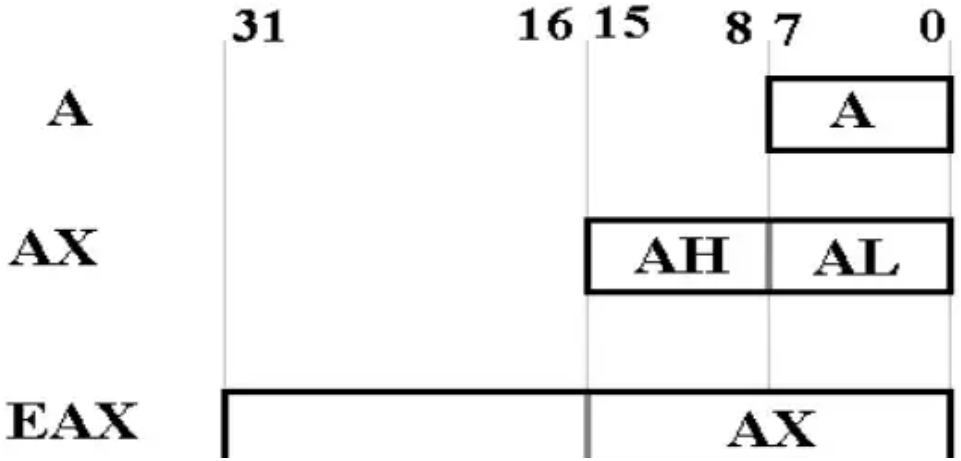
- Program Counter 缩写为PC (%eip)
  - Address of the next instruction
- Register File
  - Heavily used program data
  - Integer and floating-point

补充背景：

- 为什么寄存器叫AX、EAX、RAX



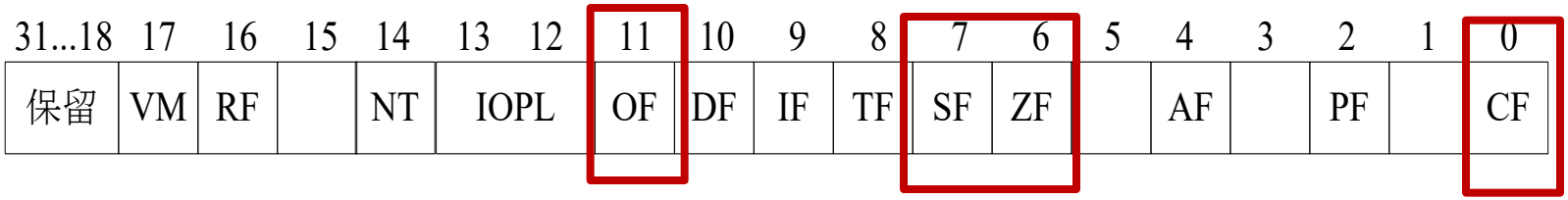
1972年 Intel 8008



1978 Intel 8086: AX (eXtended)
1985 Intel 80836: Extended AX
2003 AMD: Rextended AX

Programmer-Visible States

- Conditional code register (%eflags)
  - Hold status information about the most recently executed instruction
  - Implement conditional changes in the control flow



- Virtual Memory

Operations in Assembly Instructions

- **Instruction**
  - Performs only
- Program
  - is a sequence o
- Sequential exec
  - Normally one b
  - Conditionally b
- Typically opera
- Transfers data

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

Understanding Machine Execution

- 55 89 e5 8b 45 0c
  03 45 08 01 05 00
  00 00 00 89 ec 5d
- c3

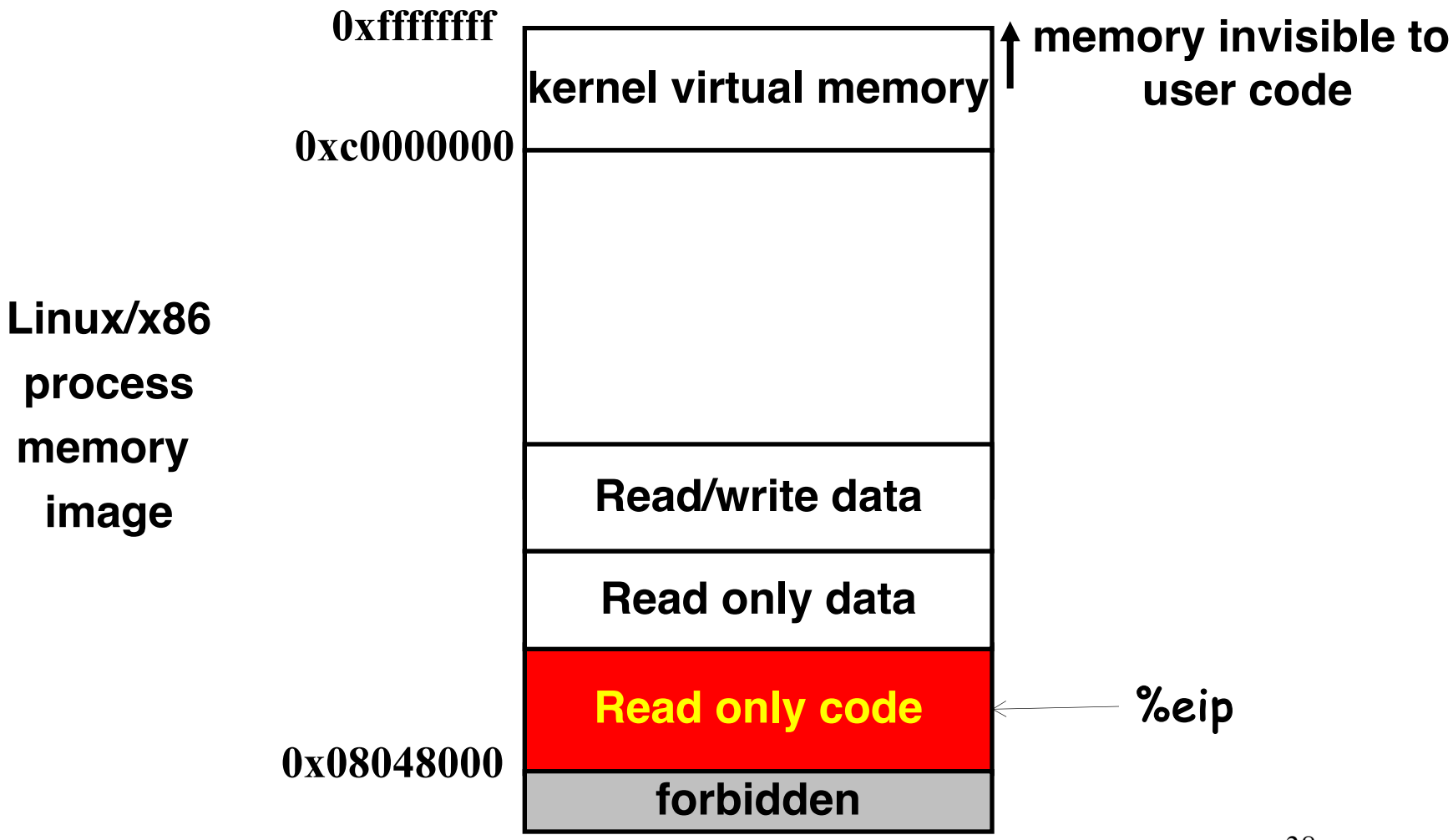
Obtain with command

`gcc -O2 -c code.c`

Relocatable object file `code.o`

  - increase %eip
    - %eip is also called program counter (PC)

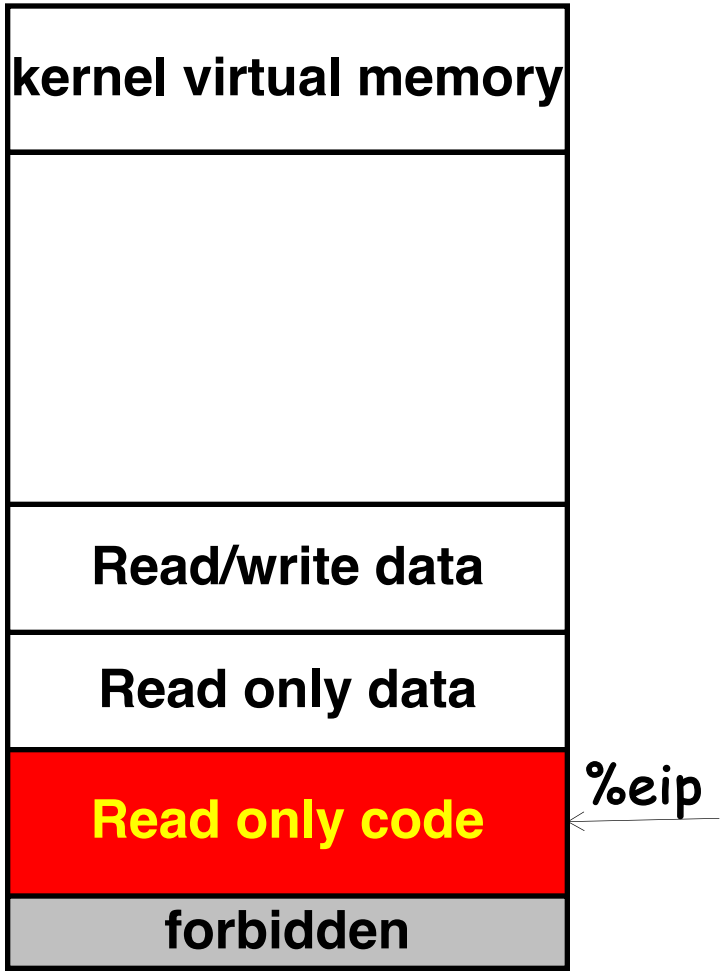
Code Layout


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

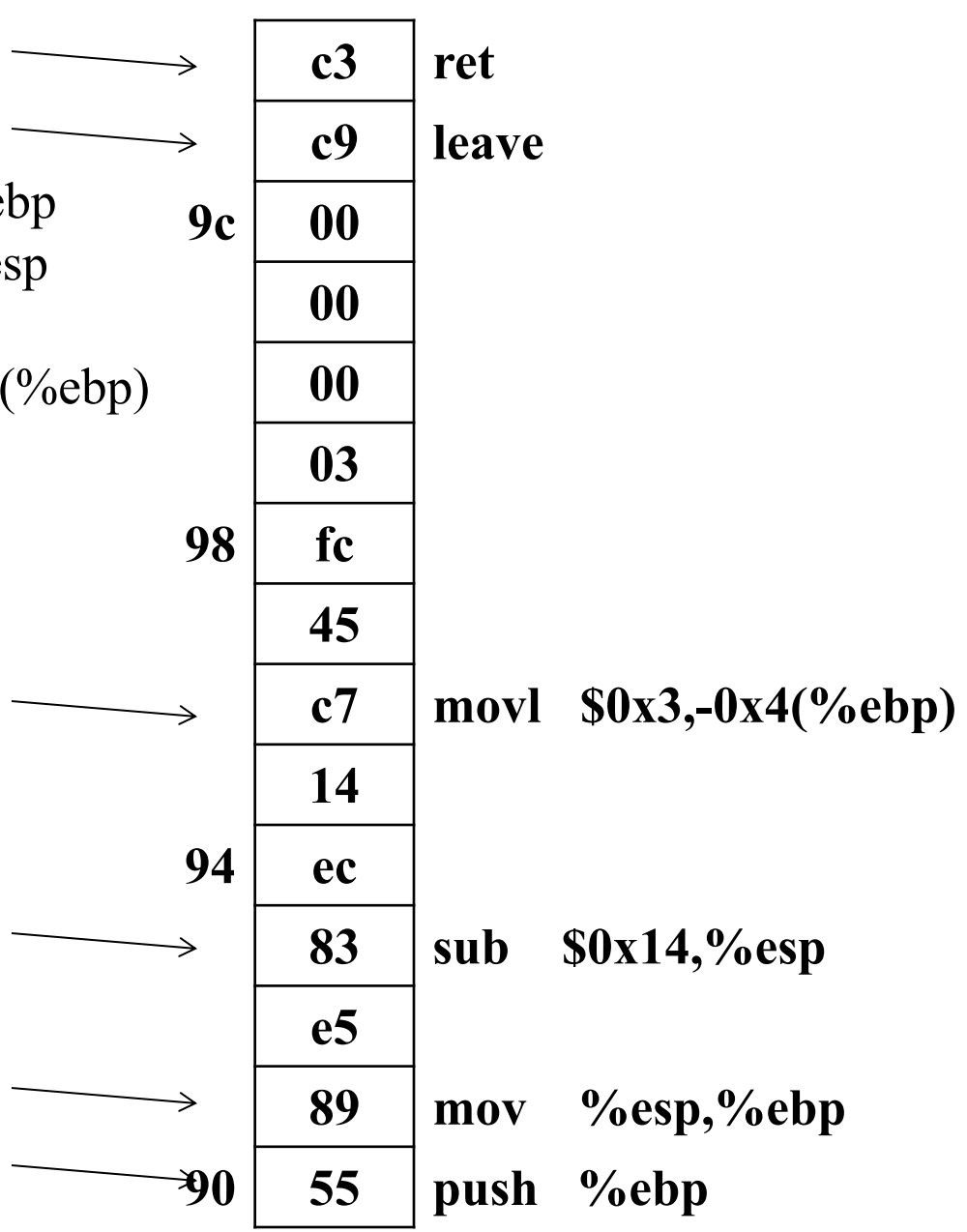
```
f()
{
   int i = 3 ;
}
```

```
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
```

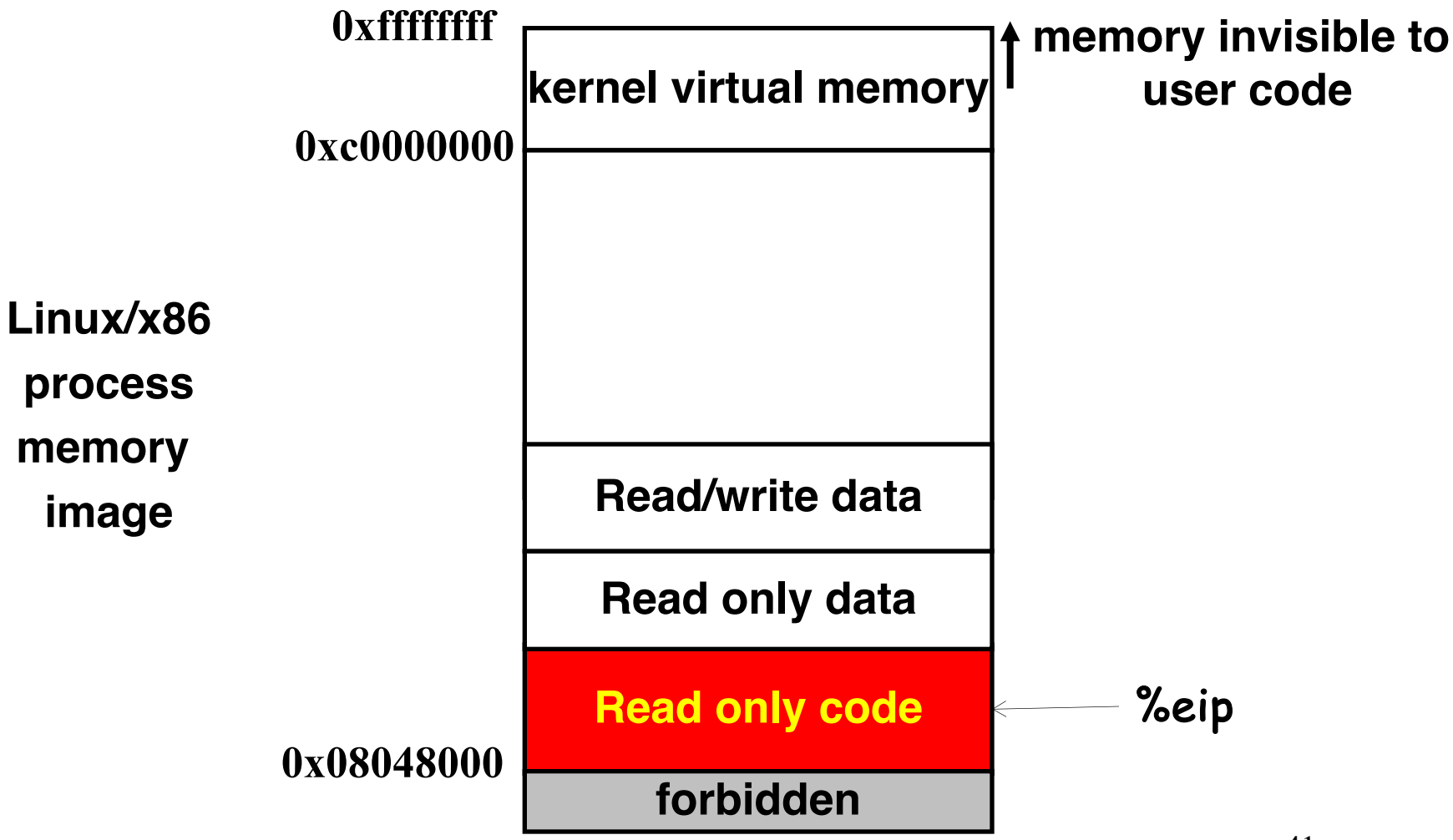


Sequential execution

```
000000 <f>:
  0:  55                    push   %ebp
  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
```



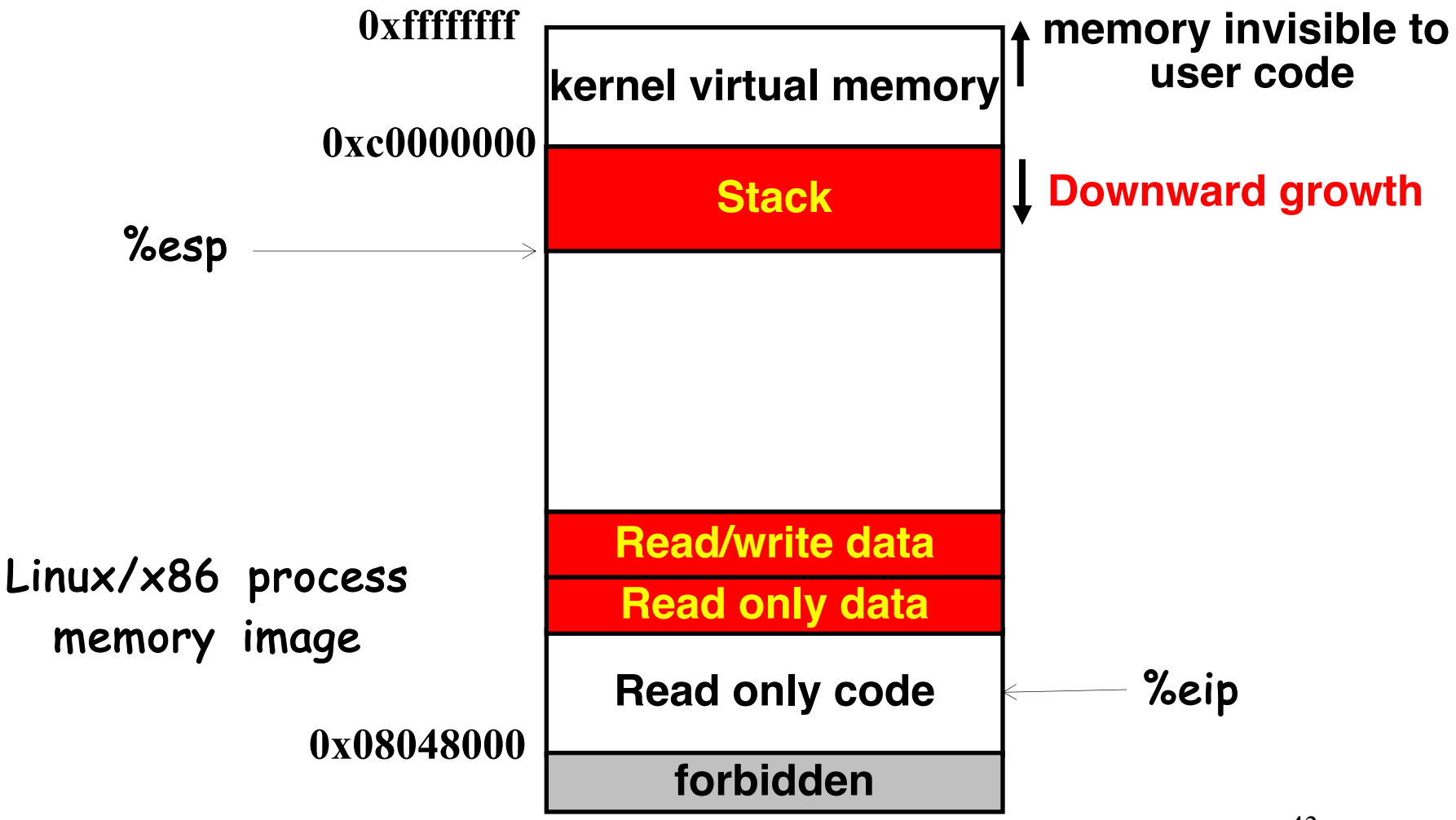
Code Layout


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

Data layout

- Object model in assembly
  - A large, byte-addressable array
  - No distinctions even between signed or unsigned integers
  - Code, user data, OS data
  - Memory Management
    - Run-time stack for managing procedure call and return
    - Blocks of memory allocated by user

Data Layout


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

Example (C Code)

```
#include <stdio.h>

int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}

int main()
{
    int s;
    s = sum(4,3);
    printf(" %d %d \n", s, accum);
    return 0;
}
```

Example (object Code)

```
08048360 <sum>:
 8048360:   55                      push   %ebp
 8048361:   89 e5                   mov    %esp,%ebp
 8048363:   8b 45 0c                mov    0xc(%ebp),%eax
 8048366:   8b 55 08                mov    0x8(%ebp),%edx
 8048369:   5d                      pop    %ebp
 804836a:   01 d0                   add    %edx,%eax
 804836c:   01 05 f0 95 04 08       add    %eax, 0x80495f0
 8048372:   c3                      ret
```

Example (object Code)

```
08048360 <sum>:
 8048360:   55                    push   %ebp
 8048361:   89 e5                 mov    %esp,%ebp
 8048363:   8b 45 0c              mov    0xc(%ebp),%eax
 8048366:   8b 55 08              mov    0x8(%ebp),%edx
 8048369:   5d                    pop    %ebp
 804836a:   01 d0                 add    %edx,%eax
 804836c:   01 05 f0 95 04 08     add    %eax, 0x80495f0
 8048372:   c3                    ret
```

Example (object Code)

```
08048360 <sum>:
 8048360:   55                    push   %ebp
 8048361:   89 e5                 mov    %esp,%ebp
 8048363:   8b 45 0c              mov    0xc(%ebp),%eax
 8048366:   8b 55 08              mov    0x8(%ebp),%edx
 8048369:   5d                    pop    %ebp
 804836a:   01 d0                 add    %edx,%eax
 804836c:   01 05 f0 95 04 08     add    %eax, 0x80495f0
 8048372:   c3                    ret
```

Organization of an Assembly Program

Example 1

```
$ 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!
[root@master test]#
[root@master test]#
[root@master test]# echo $?
0
```

```
.data
    msg : .string "hello world!\n"
    len = . – msg
.text
    .global _start
    _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
```

语法

- X86汇编
  - AT&T格式
    - 寄存器%
    - 常量、变量$
    - 寄存器参数：src, dst (→)
    - Len = . - msg
  - Intel格式
    - 寄存器参数： dst, src (←)
    - Len equ $ - msg
    - Len = $ - msg
  - gcc -O1 -S -masm=intel code.c，可以生成intel风格的汇编代码

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

不加长度后缀

语法

- .data 数据段
  - 全局变量
  - 常量
- .text 程序段
- .stack 堆栈段
  - 现代操作系统中已经很少需要手动指定栈段
- .bss段
  - 用于存储未初始化的全局变量和静态变量
  - 这些变量在程序加载到内存时会被初始化为 0

演示: Example 1

```
$ 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!
[root@master test]#
[root@master test]#
[root@master test]# echo $?
0
```

```
.data
    msg : .string "hello world!\n"
    len = . – msg
.text
    .global _start
    _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
```

常量vs变量

- 符号常量
  - 符号常量使用标识符表达一个数值
  - Len = . - msg
  - 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　　一个字节
  - .float　　浮点单精度
  - .double　　浮点双精度

工具

- 汇编器
  - as, gcc依赖的汇编器
- 链接器
  - ld
- 调试器
  - Gdb
  - 将在bomblab时详细讲解

Example 2

- 查询CPU的厂商ID

- .data
  output: .ascii "The processor Vendor ID is 'xxxxxxxxxxxx'\n"
- .text
- .global _start
- _start:
- movl $0, %eax
- cpuid

- movl $output, %edi
  movl %ebx, 28(%edi)
  movl %edx, 32(%edi)
  movl %ecx, 36(%edi)
- movl $4, %eax
  movl $1, %ebx
  movl $output, %ecx
  movl $42, %edx
  int $0x80
- movl $1, %eax
  movl $0, %ebx
  int $0x80

课堂练习

- .data
  - byte1 .byte 10110111b
  - byte2 = . - byte1
  - byte3 BYTE 61+1
  - word1 .short -9
  - dword1 .int 2274h
  - word2 .short 2274q
- 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

主观题 10分

设置

课堂练习

- .data
  - byte1 .byte 10110111b
  - byte2 = . - byte1
  - byte3 BYTE 61+1
  - word1 .short -9
  - dword1 .int 2274h
  - word2 .short 2274q
- 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

作答

2000:0000 B7 byte1
2000:0001 01 byte2
2000:0002 3E byte3
2000:0003 FF word1低
2000:0004 F7 word1高
2000:000574d word1低
2000:0006 22 dword1次低
2000:0007 00 dword1次高
2000:0008 00 dword1高
2000:0009 F0 word2低
2000:000A 24 word2高