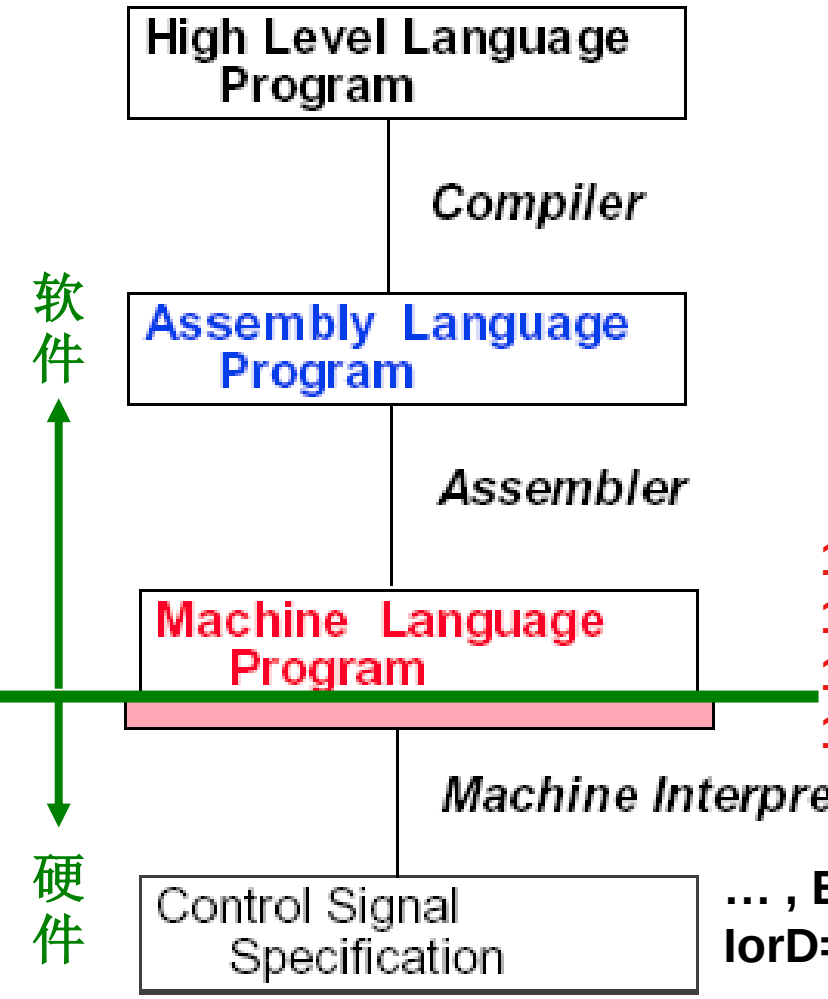
程序的机器级表示

王晶

jwang@ruc.edu.cn，信息楼124

2025年10月

Hardware/Software Interface




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

汇编指令

```
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
```

机器指令

```
1000 1100 0100 1111 0000 0000 0000 0000
1000 1100 0101 0000 0000 0000 0000 0100
1010 1100 0101 0000 0000 0000 0000 0000
1010 1100 0100 1111 0000 0000 0000 0100
```

… , EXTop=1,ALUSelA=1,ALUSelB=11,ALUop=add, IorD=1,Read,MemtoReg=1,RegWr=1,.....

微指令 → … 1 1 11 100 1 0 1 1 …

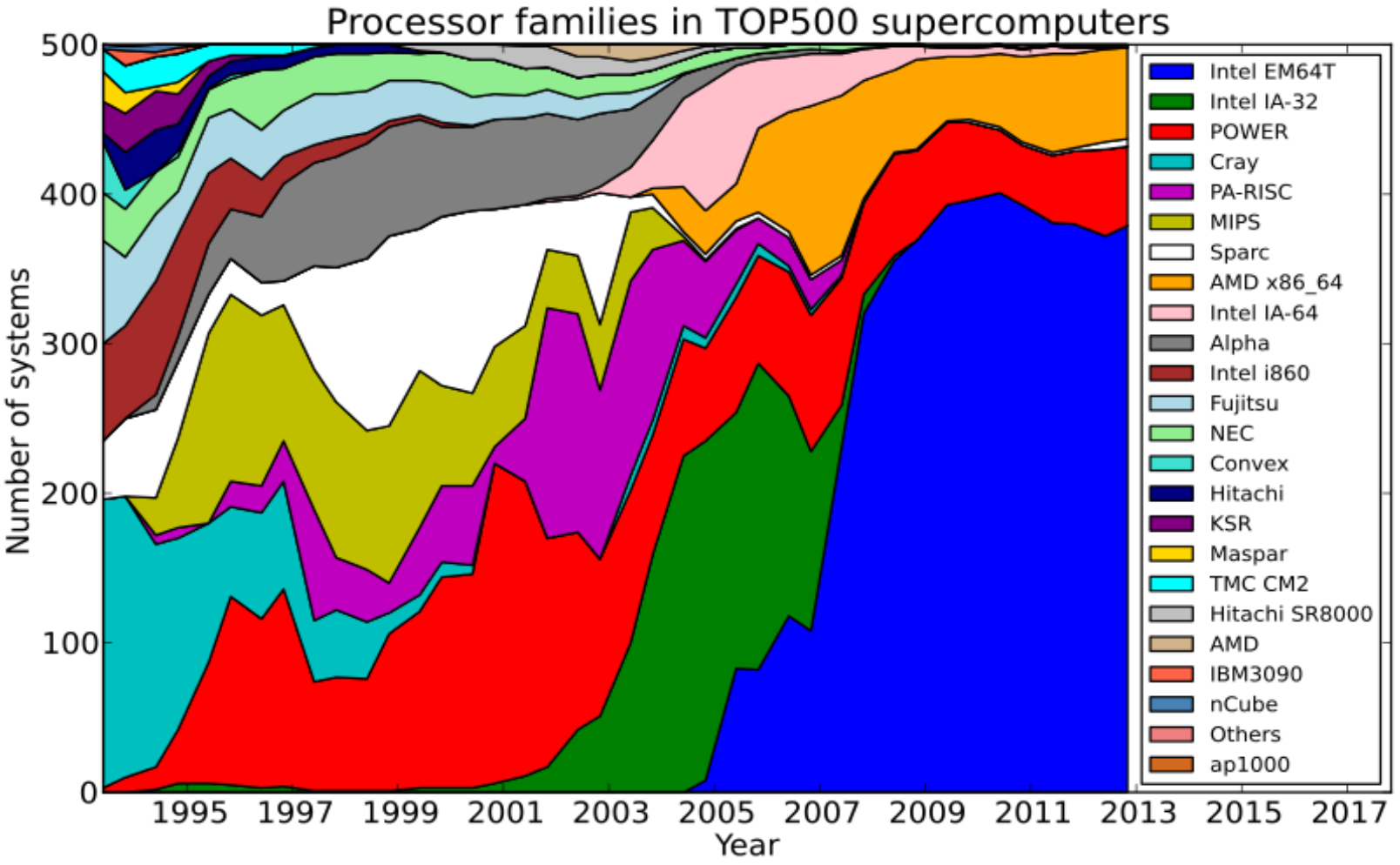
Outline

- CPU Architecture
- Memory and Registers

Intel x86 Processors

- 主导笔记本电脑 / 台式机 / 服务器市场
- 演进式设计
  - 向后兼容至 1978 年推出的 8086（架构）
  - 随着时间推移不断增加更多功能
  - 目前（相关文档）分为 3 卷，约 5000 页
- 复杂指令集计算机（CISC，Complex Instruction Set Computer ）
  - 包含多种不同格式的大量指令
    - 但在 Linux 程序中，仅会涉及其中一小部分子集
  - 难以匹敌精简指令集计算机（RISC，Reduced Instruction Set Computer ）的性能
  - 但英特尔却做到了这一点！
    - 就速度而言（确实实现了），但在低功耗方面则稍逊一筹

超级计算领域 CISC vs. RISC

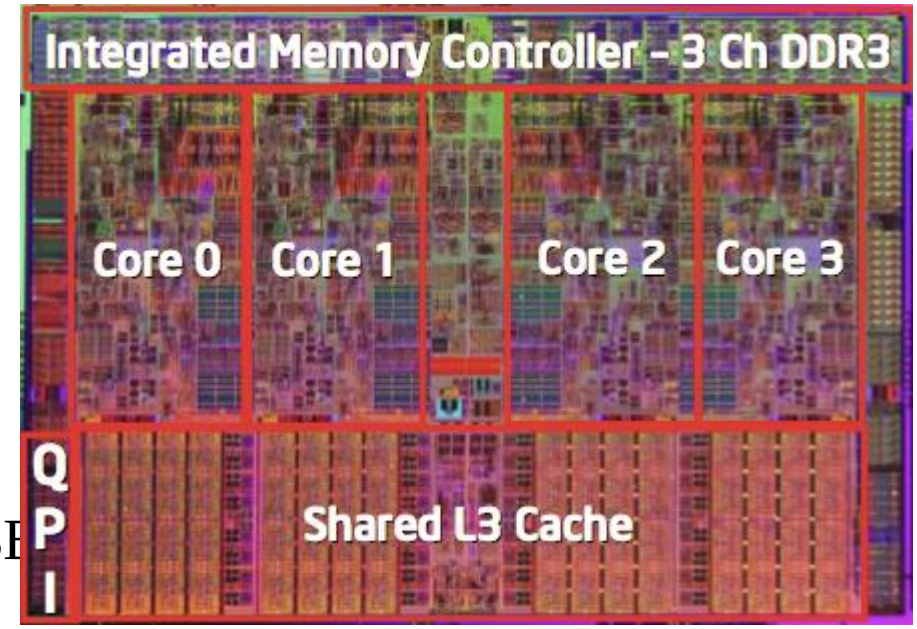


Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
• 8086	1978	29K	5-10
• First 16-bit Intel processor. Basis for IBM PC & DOS			
• 1MB address space			
• 386	1985	275K	16-33
• First 32 bit Intel processor , referred to as IA32			
• Added “flat addressing”, capable of running Unix			
• Pentium 4E	2004	125M	2800-3800
• First 64-bit Intel x86 processor, referred to as x86-64			
• Core 2	2006	291M	1060-3333
• First multi-core Intel processor			
• Core i7	2008	731M	1600-4400
• Four cores			

Intel x86 Processors, cont.

- Machine Evolution
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - Pentium Pro 1995 6.5M
  - Pentium 4 2000 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M
  - Core i7 Skylake 2015 1.9B
  - Xeon Skylake-SP 2017 8B



- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores

Intel x86 Processors, cont.

- Past Generations

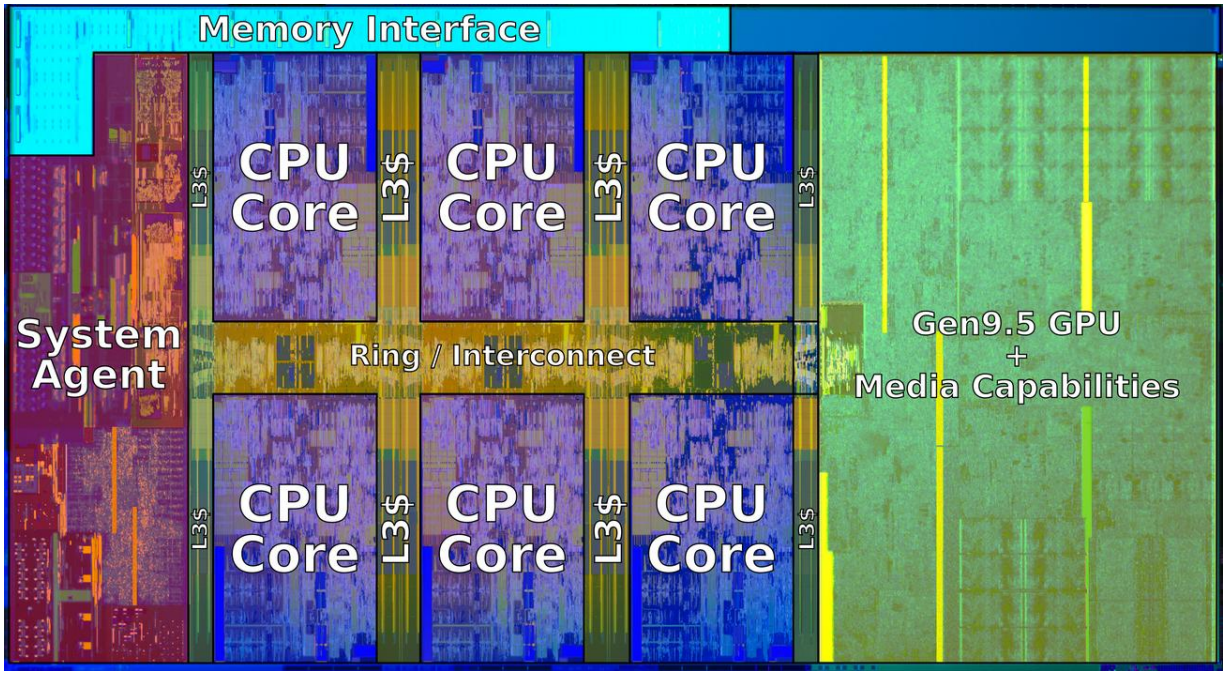
		Process technology
• 1st Pentium Pro	1995	600 nm
• 1st Pentium III	1999	250 nm
• 1st Pentium 4	2000	180 nm
• 1st Core 2 Duo	2006	65 nm

- Recent & Upcoming Generations

1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
8.	Coffee Lake	2017	14 nm
9.	Cannon Lake	2018	10 nm
10.	Ice Lake	2019	10 nm
11.	Tiger Lake	2020	10 nm

Process technology dimension = width of narrowest wires (10 nm ≈ 100 atoms wide)

2018 : Coffee Lake



- Mobile Model: Core i7
  - 2.2-3.2 GHz
  - 45 W

- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W

- **Server Model: Xeon E**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W

x86 Clones: Advanced Micro Devices (AMD)

- 历史上
  - 超微半导体公司（AMD）一直紧随英特尔之后
  - （产品）速度略慢，但价格便宜得多
- 随后
  - 从Digital Equipment Corp. 及其他发展趋势下滑的企业中招募了顶尖电路设计师
  - 推出皓龙（Opteron）处理器：成为奔腾 4（Pentium 4）的强劲竞争对手
  - 研发出 x86-64 架构，这是其自主开发的 64 位扩展架构
- 近年来
  - 英特尔重整旗鼓
    - 1995-2011 年：位居全球领先半导体 “晶圆代工厂” 之列
    - 2018 年：按营收计算位列全球第二（第一为三星）
    - 2019 年：重新夺回全球第一的位置
  - 超微半导体公司（AMD）曾一度落后
    - 依赖外部半导体制造商Global Foundaries生产芯片
    - 约 2019 年推出的处理器（如锐龙 Ryzen 系列）再次具备竞争力

Intel's 64-Bit History

- 2001 年：英特尔试图从 IA32 架构向 IA64 架构实现彻底转型
  - 采用完全不同的架构（即安腾 Itanium 架构）
  - 仅将 IA32 代码作为遗留代码（兼容模式）执行
  - 性能表现未达预期（令人失望）
- 2003 年：超微半导体公司（AMD）推出演进式解决方案
  - 研发出 x86-64 架构（现称 “AMD64 架构”）
- 英特尔当时被迫专注于 IA64 架构
  - 难以承认（自身战略）失误，也不愿承认 AMD 的方案更优
- 2004 年：英特尔宣布为 IA32 架构推出 EM64T 扩展技术
  - 即 “扩展内存 64 位技术”（Extended Memory 64-bit Technology，简称 EM64T）
  - 该技术与 x86-64 架构几乎完全一致！
- 如今，除低端 x86 处理器外，几乎所有 x86 处理器均支持 x86-64 架构
  - 但仍有大量代码以 32 位模式运行

x86体系结构

体系结构		厂商	微处理器型号	字长	年代
x86	“x86-16”	Intel	**8086**, 8088, 80186, 80188 80286	16位	1978年起
	IA-32	Intel	**80386**, 80486, Pentium, Pentium Pro/II/III/4, Core, Atom	32位	1985年起
		AMD	Am386, Am486, AM5x86, K5, K6, Athlon		
		Others	**Cyrix** 5x86; **VIA** C3/C7 **Transmeta** Crusoe, Efficeon		
	x86-64	AMD	**Athlon 64**, Opteron Phenom, Phenom II	64位	2003年起
		Intel	Pentium 4 Prescott, Core 2 Core i3/i5/i7		
		Others	**VIA** Nano		

注：Intel提出的IA-64是独立于x86的一种新的体系结构，不兼容IA-32

Our Coverage

- IA32
  - The traditional x86
  - `shark> gcc –m32 hello.c`
- x86-64
  - The emerging standard
  - `shark> gcc hello.c`
  - `shark> gcc –m64 hello.c`

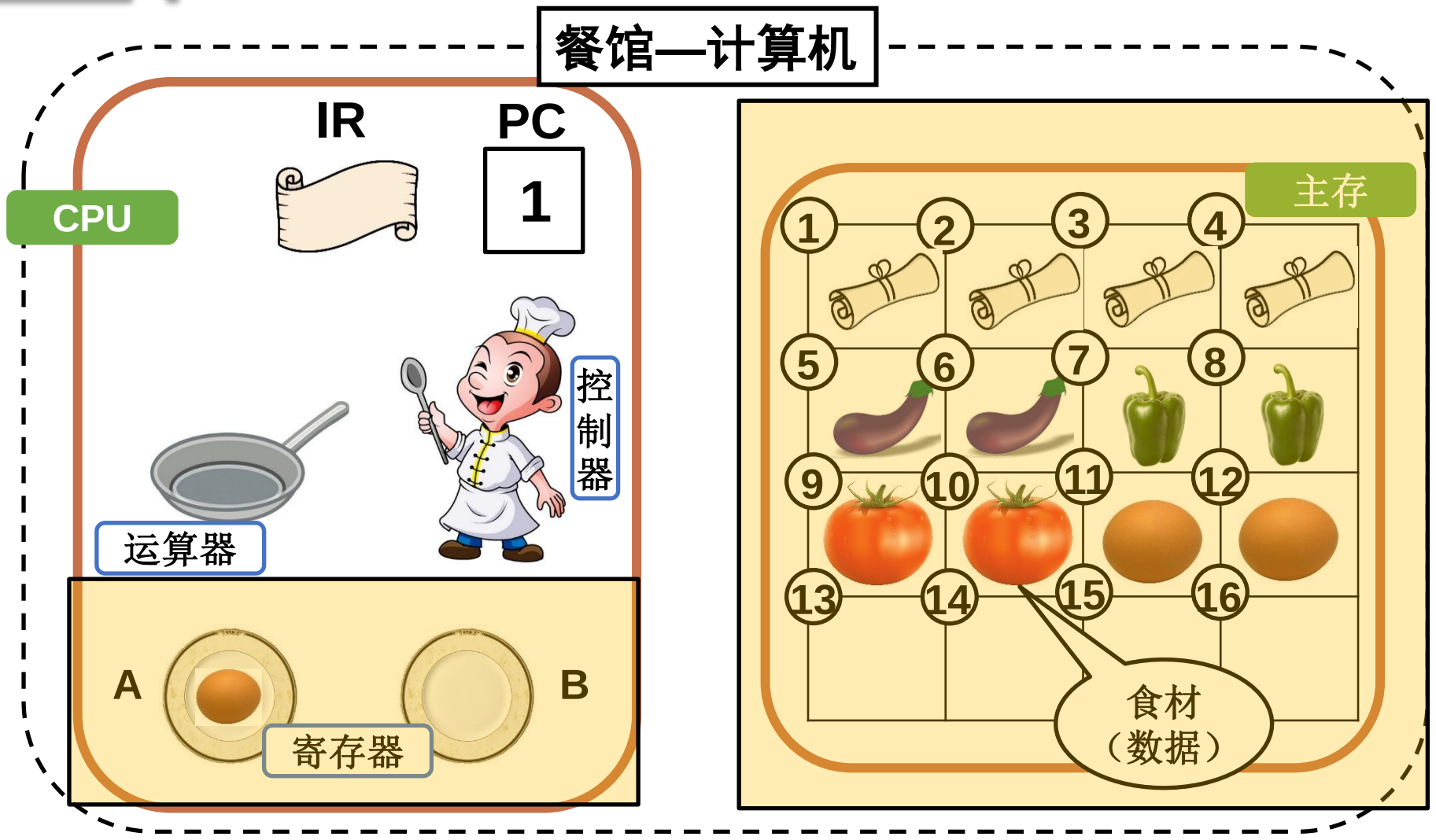
处理器的编程结构

- 编程人员看到的处理器的软件结构模型

处理器内部：
① 寄存器的功能、操作和限制
② 程序设计中如何使用寄存器

处理器外部：
① 存储器中数据如何组织
② 如何从存储器中取指令和数据

回顾：计算机中数据的存储



Outline

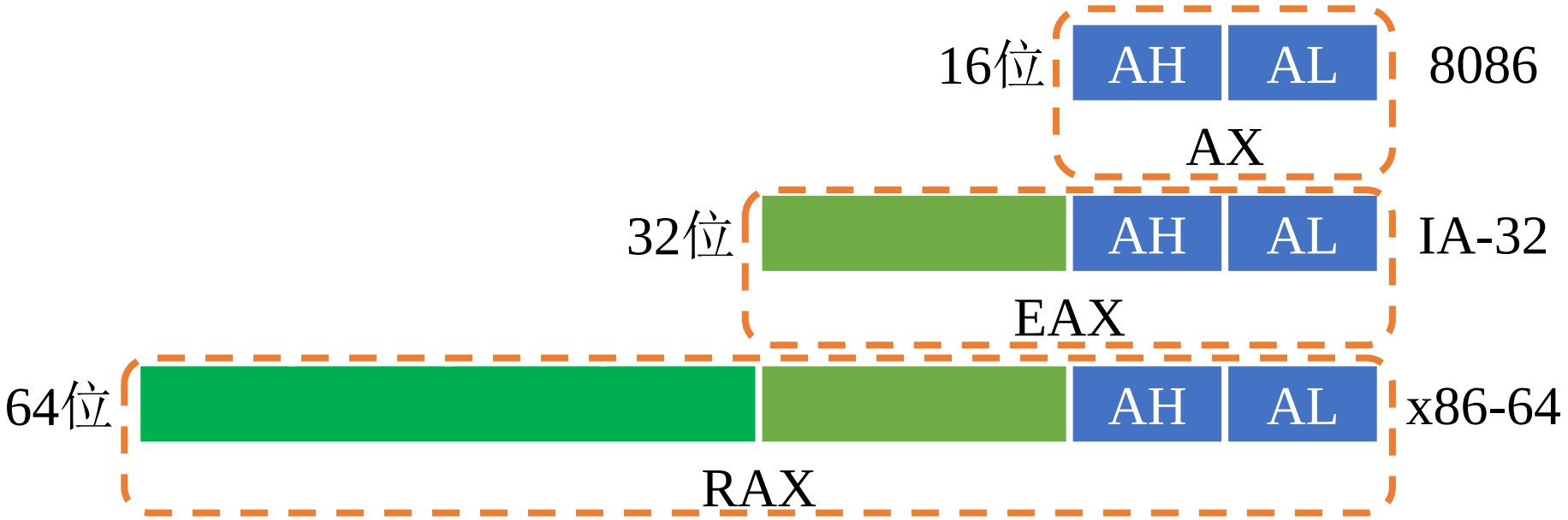
- CPU Architecture
- Memory and Registers
  - Registers
  - Memory

x86体系结构寄存器模型

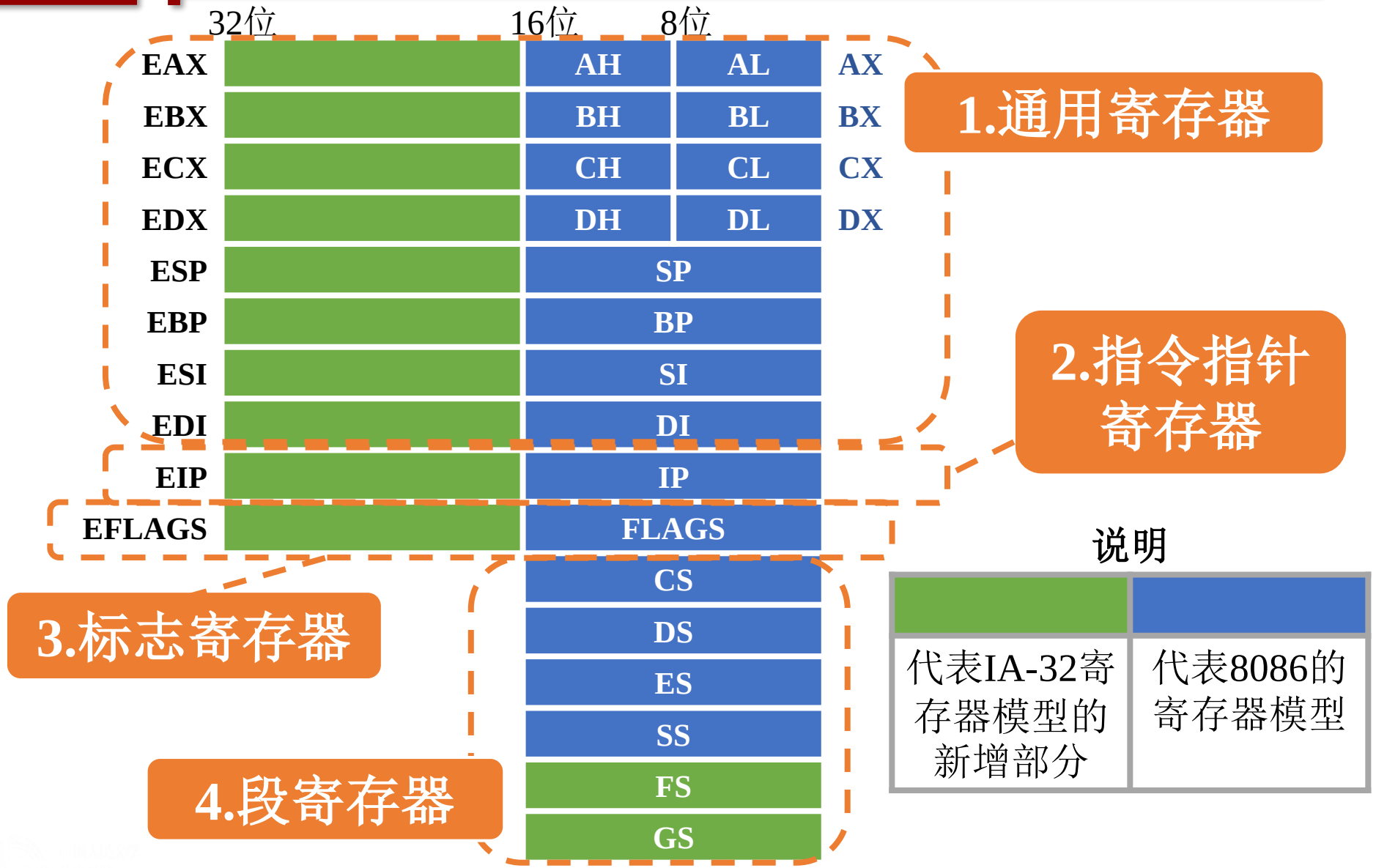
① 8086（包括80286等）：16位寄存器组

② IA-32：基于8086扩展到32位寄存器组

③ x86-64：基于IA-32扩展到64位寄存器组

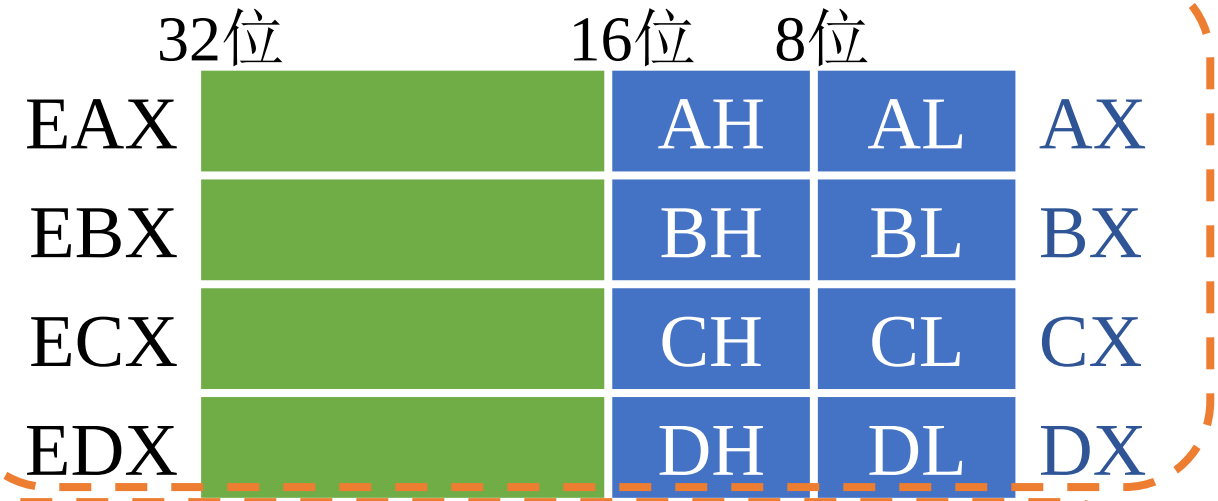


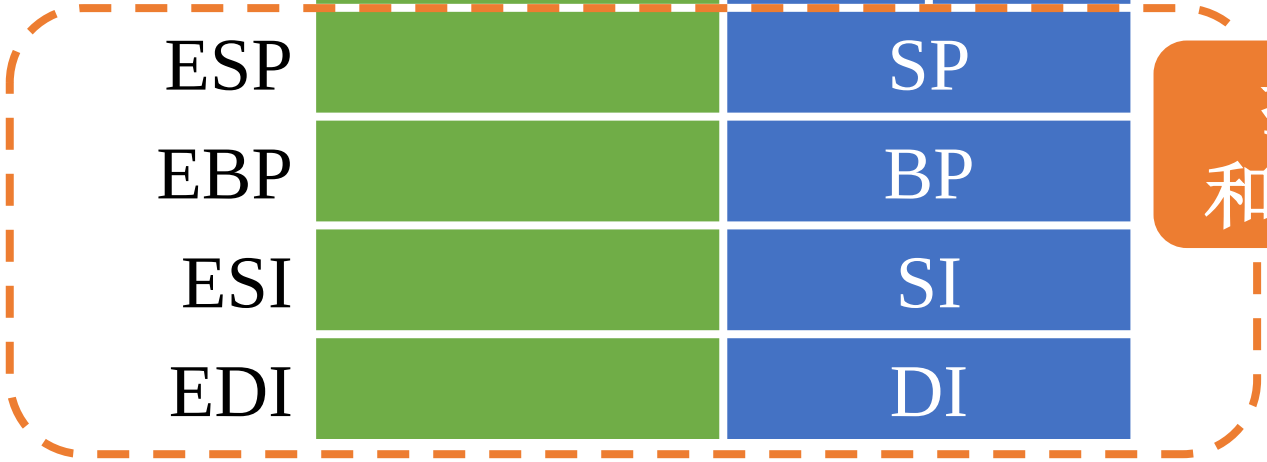
IA-32和8086的寄存器模型



1. 通用寄存器

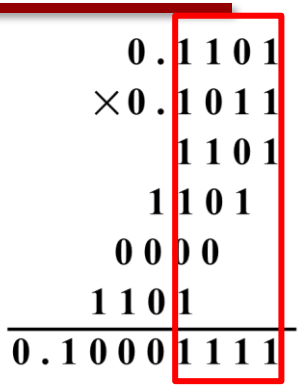
通用数据寄存器





指针寄存器
和 变址寄存器

通用数据寄存器

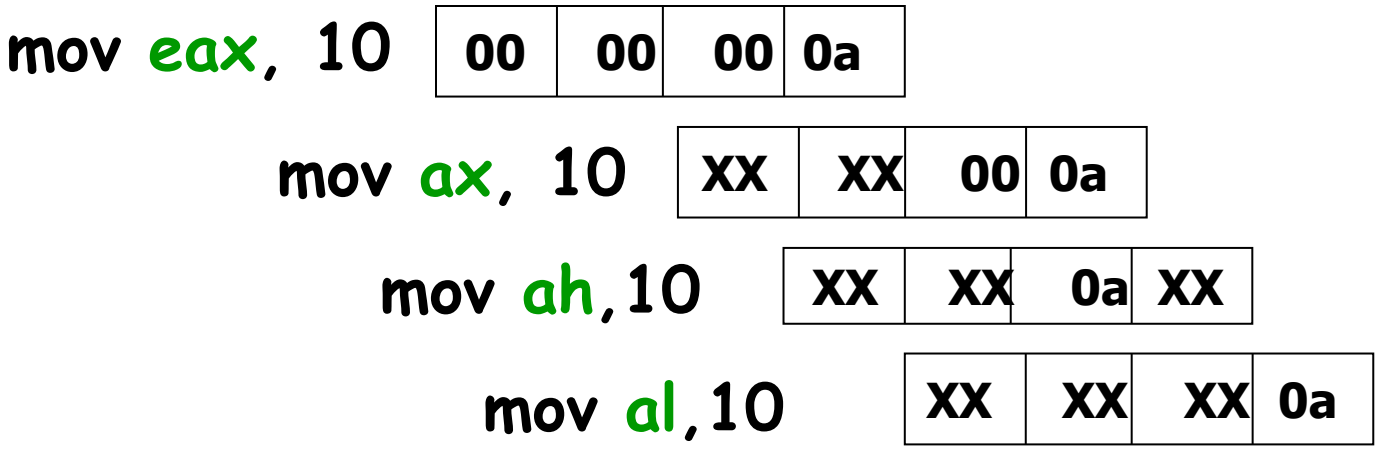


- 此类寄存器共有4个
  - 可用来存放8位、16位或32位的操作数
  - 适用大多数算术运算和逻辑运算指令
  - 除存放通用数据外，各有一些专门的用途：

EAX/AX	Accumulator	存放乘除等指令的操作数
EBX/BX	Base	存放存储单元的偏移地址
ECX/CX	Count	存放计数值
EDX/DX	Data	乘法运算产生的部分积 除法运算的部分被除数

寄存器模型

- EAX、EBX、ECX、EDX



指针寄存器 和 变址寄存器

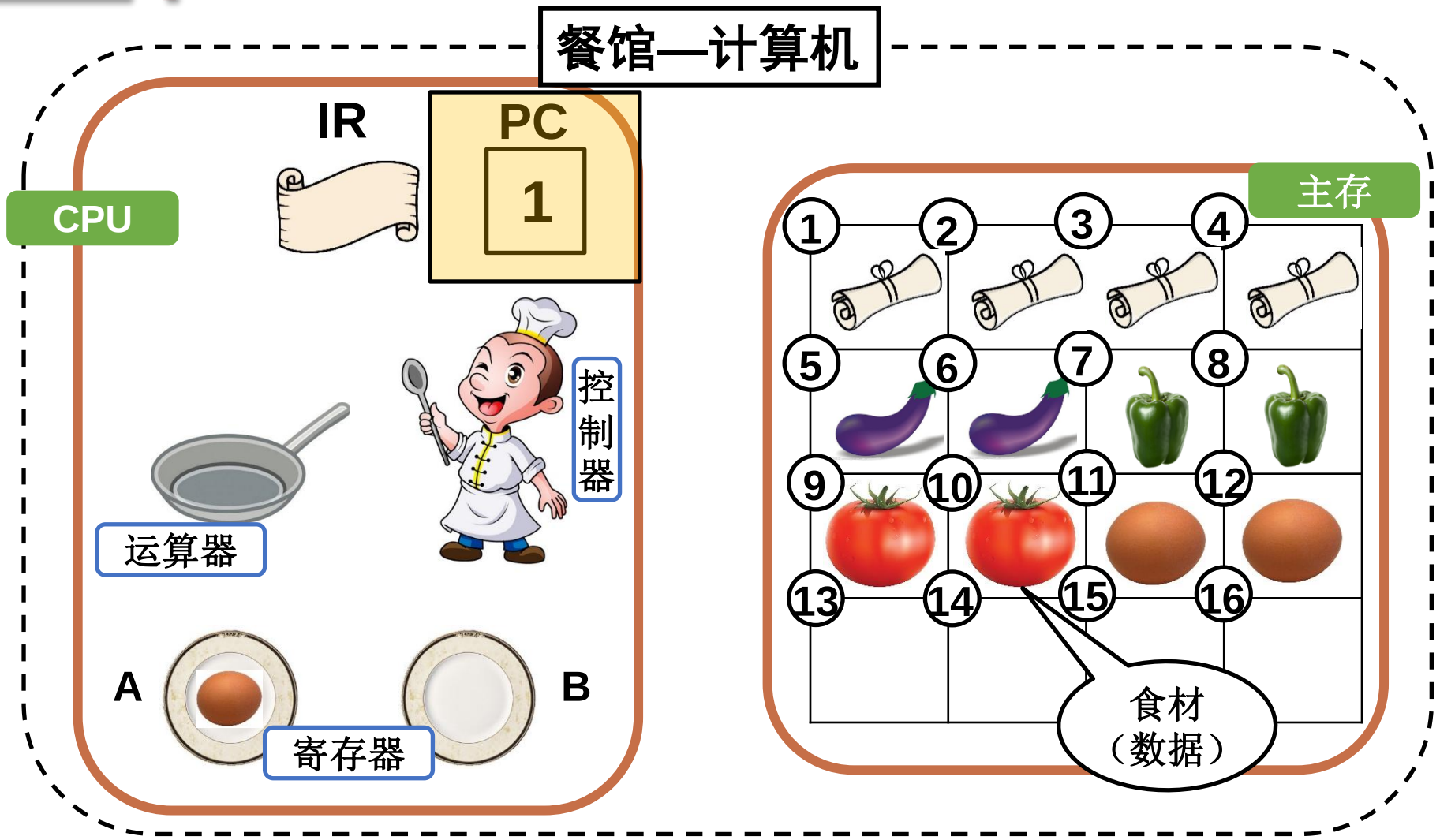
- 此类寄存器共有4个
  - 既可作为32位寄存器，又可作为16位寄存器
  - ESP和EBP（或 SP和BP），用于堆栈操作
  - ESI和EDI（或 SI和DI），用于串操作
  - 都可以作为数据寄存器使用

ESP/SP	stack pointer	堆栈指针寄存器
EBP/BP	(stack)base pointer	(堆栈)基址指针寄存器
ESI/SI	source index	源变址寄存器
EDI/DI	destination index	目的变址寄存器

2. 指令指针寄存器

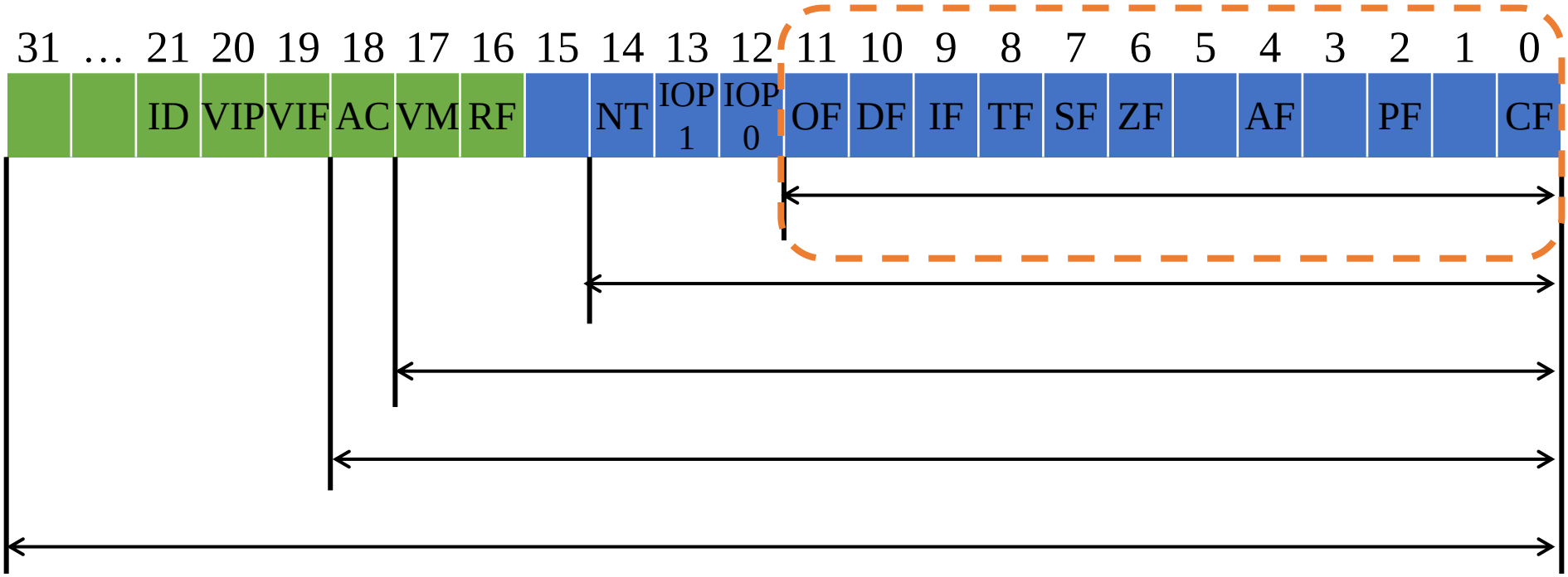
- 指令指针寄存器EIP/IP（Instruction Pointer）
  - 保存一个内存地址，指向当前需要取出的指令
  - 当CPU从内存中取出一个指令后，EIP/IP就自动增加，指向下一指令的地址
  - 程序员不能对EIP/IP进行存取操作
  - 程序中的转移指令、返回指令以及中断处理会改变EIP/IP的内容

回顾：计算机中数据的存储



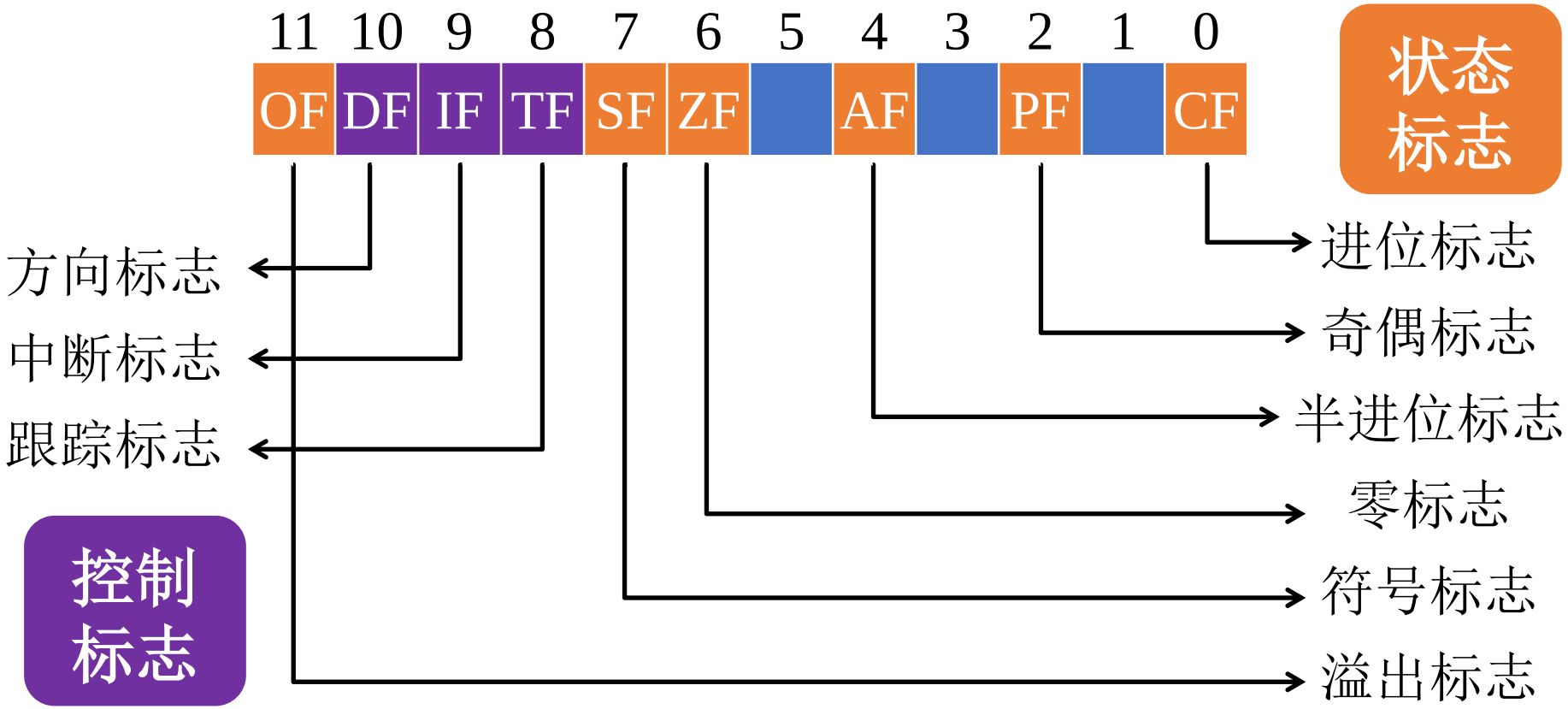
3. 标志寄存器

- 标志寄存器EFLAGS/FLAGS
  - 用于指示微处理器的状态并控制它的操作
  - 从8086到80286，从80386到Pentium，标志寄存器的内容在不断扩充



实模式标志位

- 8086系统中所定义的9个标志位也是80x86微处理器实模式下所使用的标志位



4. 段寄存器

- 段寄存器
  - 与微处理器中其他寄存器联合生成存储器地址
  - 对于同一个微处理器，段寄存器的功能在实模式下和保护模式下是不相同的

CS	代码段寄存器（Code Segment）
DS	数据段寄存器（Data Segment）
ES	附加段寄存器（Extra Segment）
SS	堆栈段寄存器（Stack Segment）
FS	80386新增的附加段寄存器
GS	80386新增的附加段寄存器

- 80x86不允许直接将立即数送入段寄存器
  - Mov ds, 1000H　　非法
  - Mov ax, 1000H　　mov ds, ax合法

寄存器模型

- 64位计算机
  - X86-64指令集扩展了IA32的32位寄存器
    - 用’r’代替’e’，%eax->%rax, %esp->%rsp
  - 增加了8个寄存器
    - %r8~%r15
  - 浮点数XMM寄存器
    - %xmm0~%xmm15，128位，能放下4个float，或2个double

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Outline

- CPU Architecture
- Memory and Registers
  - Registers
  - Memory

主存储器的组织形式

- 主存储器通常按字节为单元进行划分

地址：每个存储单元对应的序号

内容：存储单元中存放的二进制信息

地址	内容
…	……
0011	00001100
0010	00100010
0001	00000000
0000	01101101

存储器组织

- 主存储器容量很大，被划分成许多存储单元
- 每个存储单元被编排一个号码
  - 即存储单元地址
  - 称为存储器地址（Memory Address）
- 每个存储单元以字节为基本存储单位
  - 即字节编址（Byte Addressable）
  - 一个字节（Byte）等于8个二进制位（Bit）
  - 二进制位是计算机存储信息的最小单位
  - 16位（2个字节）构成一个字（Word）
  - 32位（4个字节）构成一个双字（Double Word）

存储模型

- 物理存储器以字节为基本存储单位
- 每个存储单元被分配一个唯一的地址
- 这个地址就是物理地址
- 物理地址空间从0开始顺序编排，直到处理器支持的最大存储单元
  - 8086处理器支持1MB存储器：00000H～FFFFFH
  - IA-32处理器支持4GB存储器：00000000H～FFFFFFFFH
- IA-32处理器提供3种存储模型，用于程序访问存储器

物理地址与逻辑地址

- 物理地址
  - 信息在存储器中实际存放的地址
  - CPU访问存储器时实际输出的地址
- 逻辑地址
  - 编程时所使用的地址
  - 实模式下由“段基值”和“偏移量”构成
- 使用逻辑地址的优势
  - 编程时不需要知道代码或数据在存储器中的具体物理位置，从而简化存储资源的动态管理

逻辑地址与物理地址

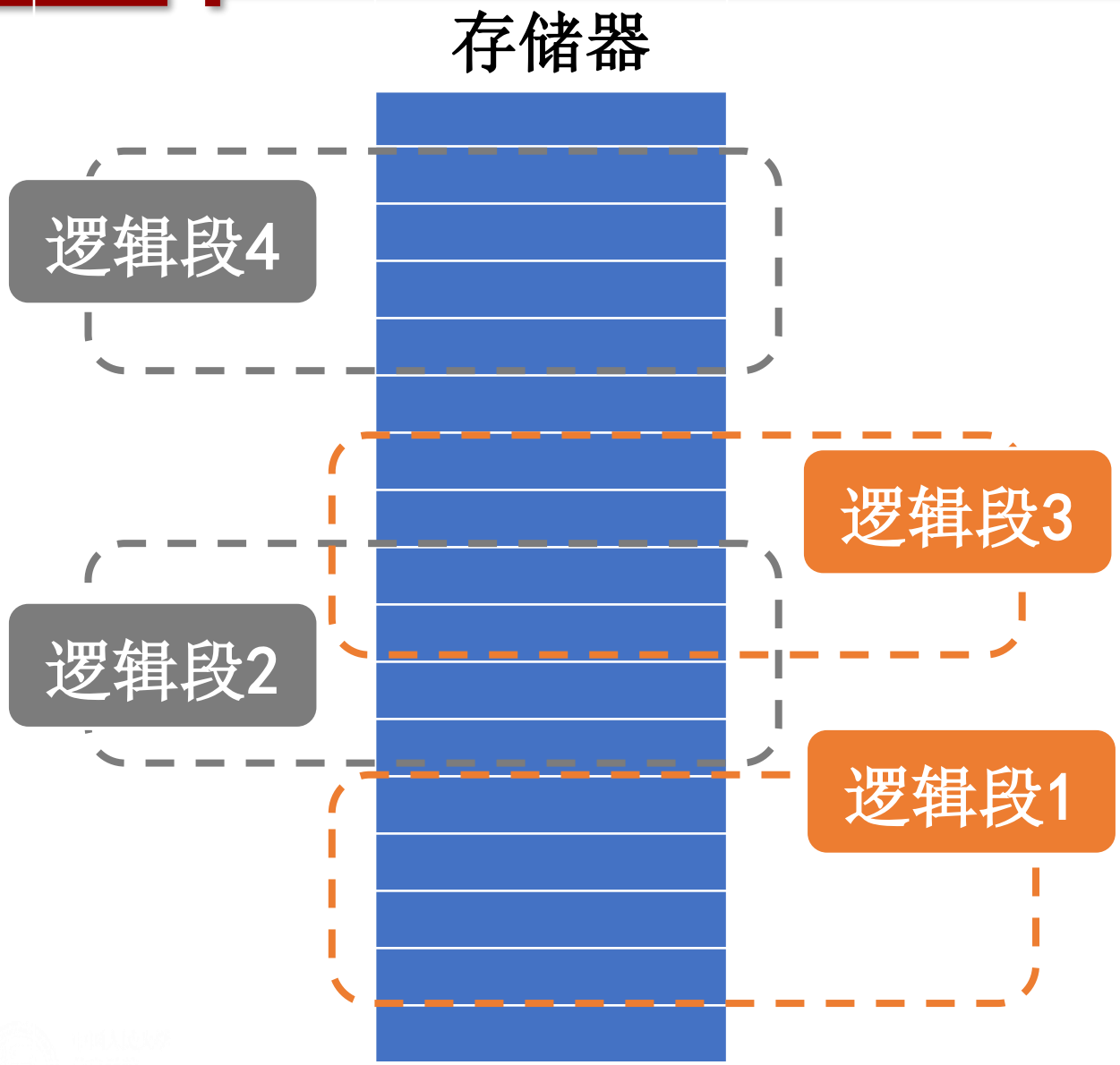
物理地址＝绝对地址：15（第15号房间）

21	22	23	24	25	26	27	28	29	30
11	12	13	14	15	16	17	18	19	20
01	02	03	04	05	06	07	08	09	10

逻辑地址＝相对地址：205（2层05号房间）

301	302	303	304	305	306	307	308	309	310
201	202	203	204	205	206	207	208	209	210
101	102	103	104	105	106	107	108	109	110

逻辑段在物理存储器中的位置



各个逻辑段在实际存储空间中可以完全分开，也可以部分重叠，甚至完全重叠

段起始地址的计算和分配通常由操作系统完成，不需要普通用户参与

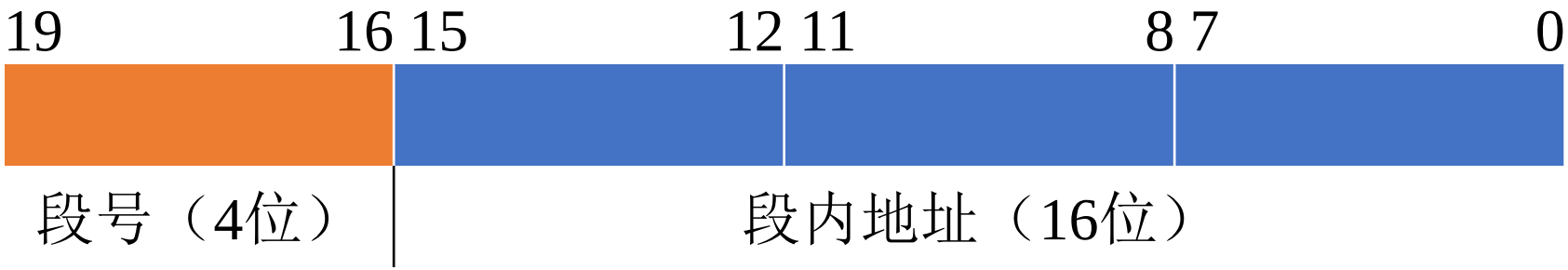
存储器的寻址需求和寻址能力

- 8086的外部寻址需求
  - 20位宽的地址线
  - 寻址范围：2^{20}=1M字节单元
- 8086的内部寻址能力
  - 16位宽的寄存器，ALU也只能进行16位运算
  - 寻址范围：2^{16}=65536(64K)字节单元

- 解决方案
  - 存储器分段技术

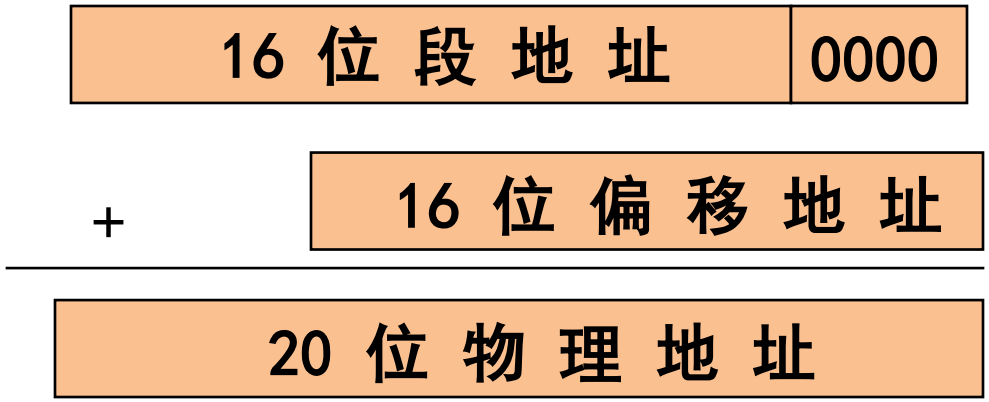
直观的存储器分段方法

- 实现方法：将20位物理地址分成两部分
  - 高4位为段号，用“段号寄存器”来保存
  - 低16位为段内地址，也称“偏移地址”
- 不足之处
  - 段号寄存器与其他寄存器不兼容，操作麻烦
  - 每个逻辑段固定占用64K字节，会浪费存储空间
- 相比之下，8086的分段技术更为灵活

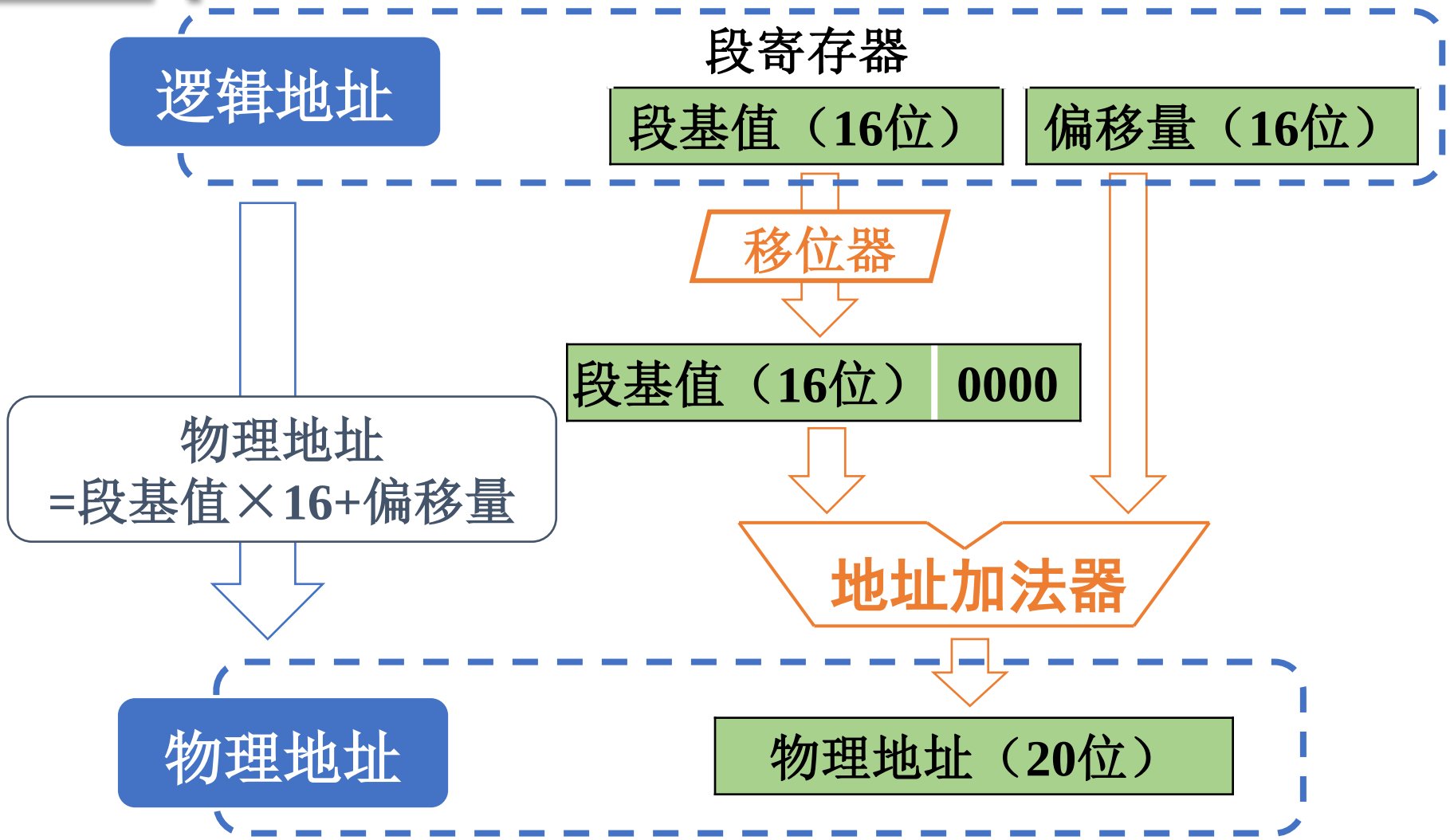


X86的存储器分段方法

逻辑地址 vs. 物理地址

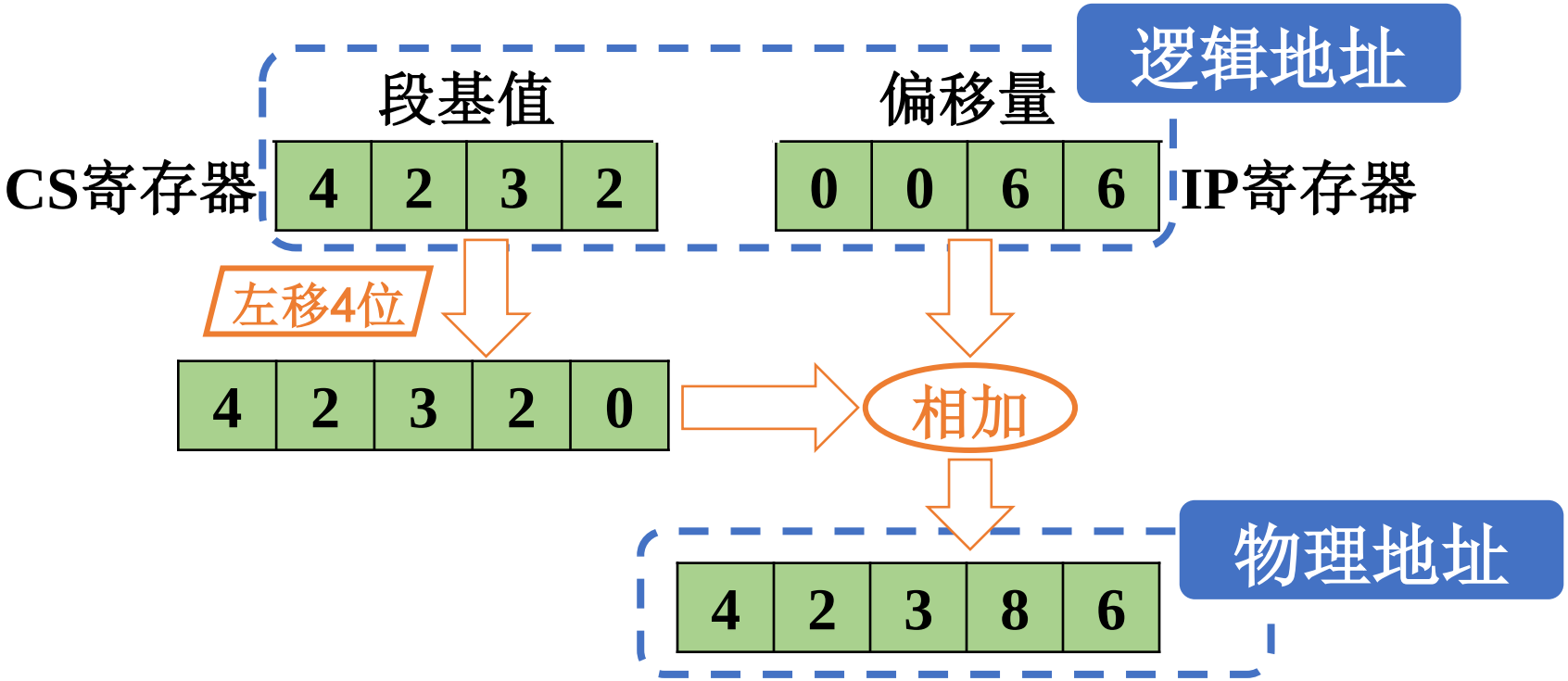


实模式下物理地址的产生



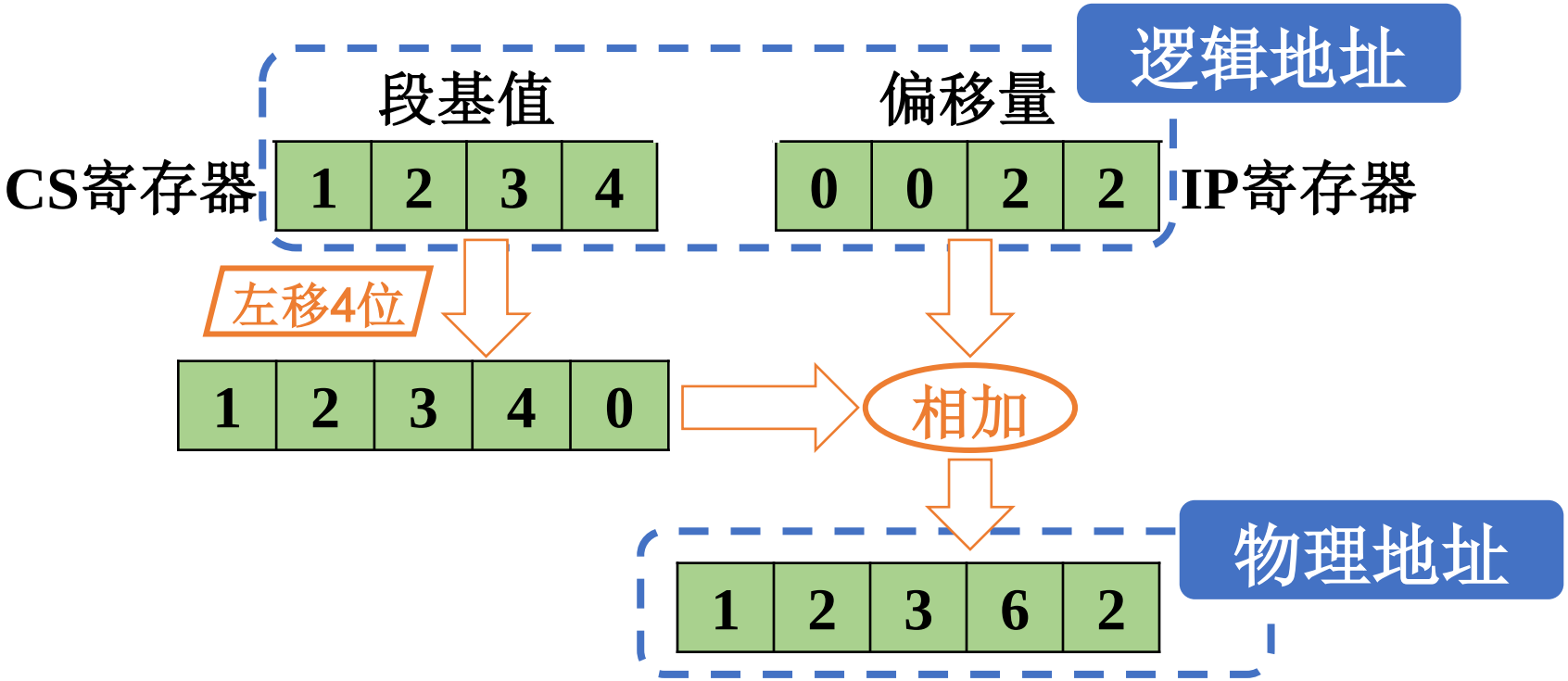
实模式下物理地址的产生-示例1

- 设：代码段寄存器CS的内容为4232H，指令指针寄存器IP的内容为0066H
- 求：访问代码段存储单元的物理地址

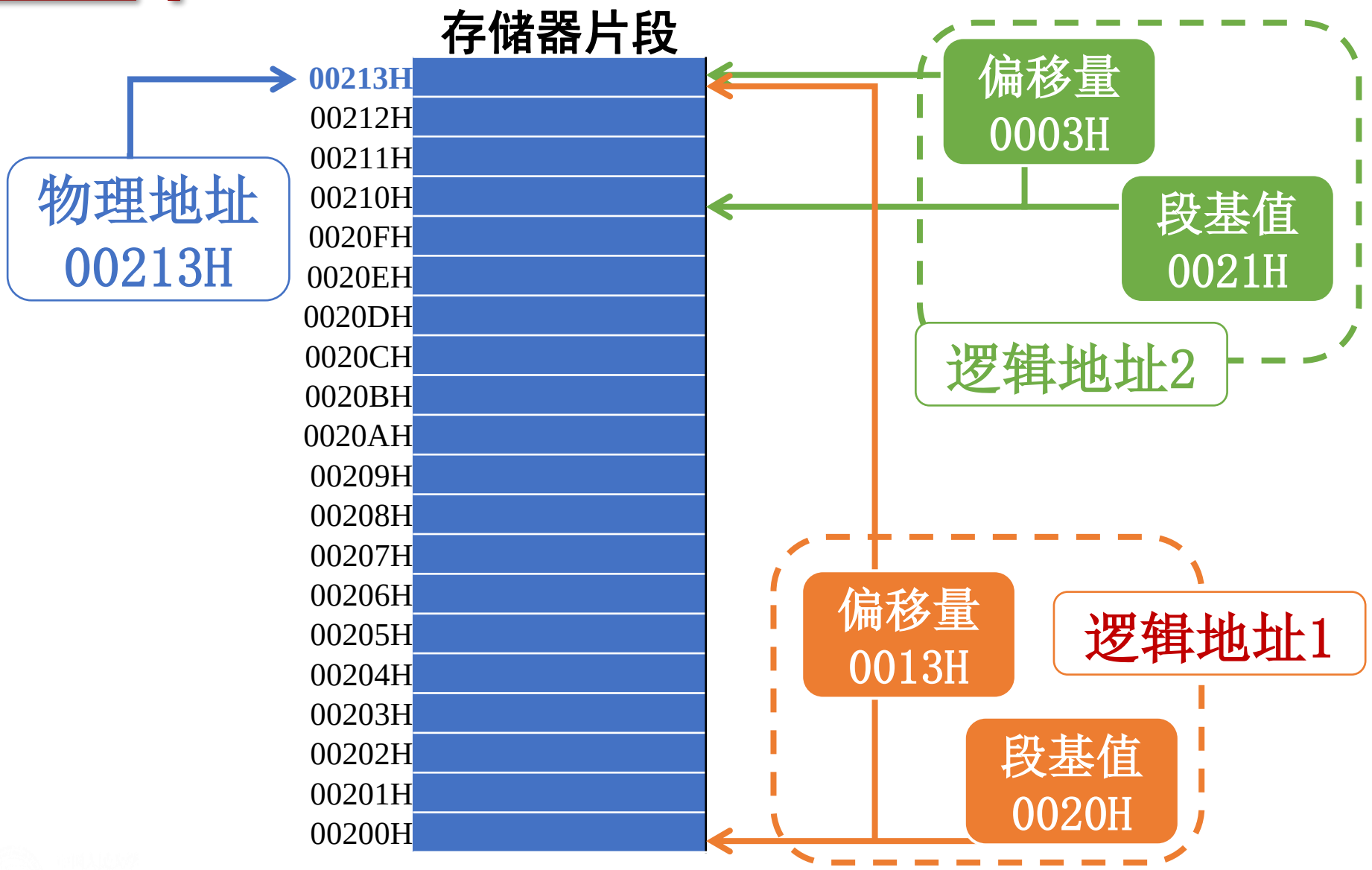


实模式下物理地址的产生-示例2

- 设：数据段寄存器DS的内容为1234H，基址寄存器BX的内容为0022H
- 求：访问数据段存储单元的物理地址

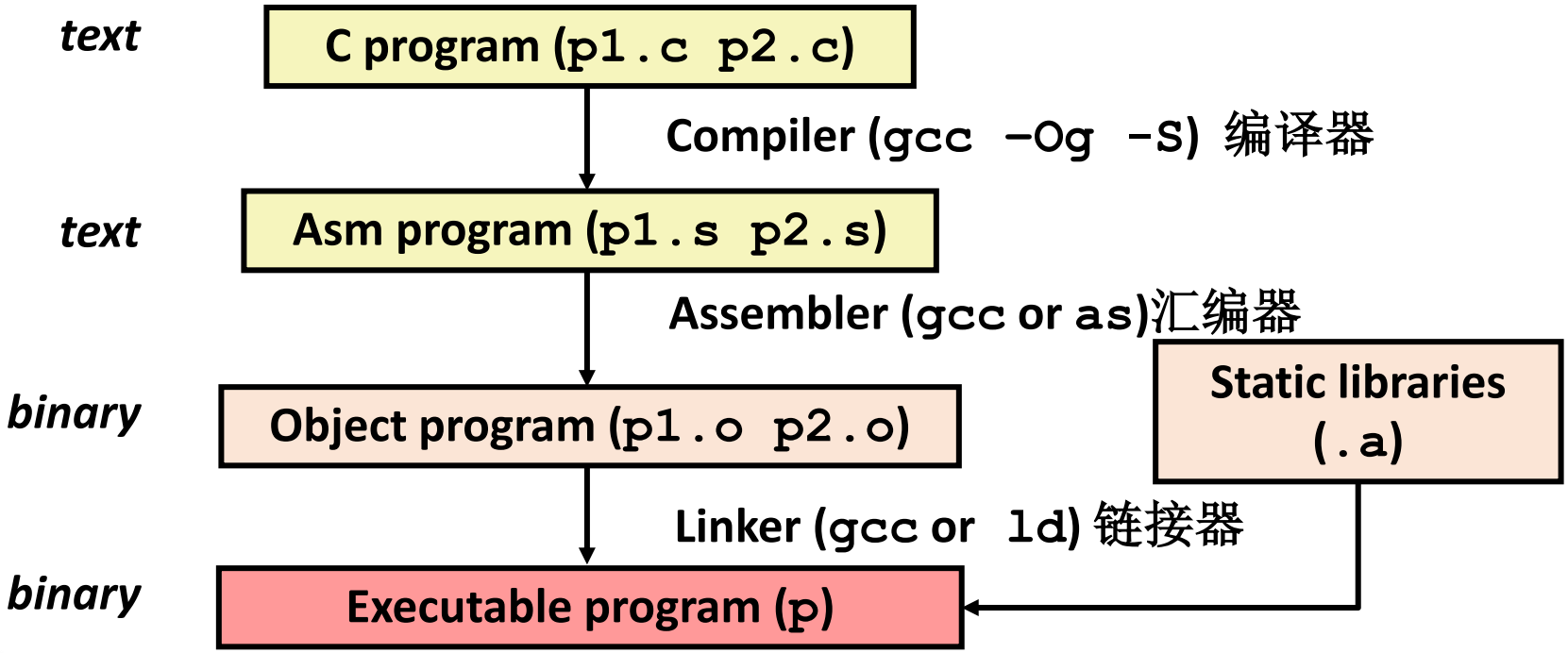


同一物理地址与多个逻辑地址对应的示例



从高级语言程序获得汇编程序

- 2个C语言文件：`p1.c p2.c`
- 编译命令：`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

```
#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
gcc –S test.c –o test.s

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>:

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

位移量　机器指令　汇编指令

编译得到的与反汇编得到的汇编指令形式稍有差异

工具

- 汇编器
  - as, gcc依赖的汇编器
- 链接器
  - ld
- 调试器
  - Gdb

```
$ 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!
```

➢ 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

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

主观题 10分

设置

- 内存中存放的机器码和对应的汇编指令如下图所示，设CPU初始状态：CS=2000H，IP=0000H，请写出指令执行序列。

地址	内存	对应汇编指令	地址	内存	对应汇编指令
10000H	B8	mov ax, 0123H	20000H	B8	mov ax, 6622H
	23			22	
	01			66	
10003H	B8	mov ax, 0000	20003H	EA	jmp 1000:3
	00			03	
	00			00	
10006H	8B	mov bx, ax		00	
	D8			10	
10008H	FF	jmp bx	20008H	89	mov cx, ax
10009H	E3			C1	

作答

练习答案

- Mov ax, 6622H
- Jmp 1000:3
- Mov ax, 0000
- Mov bx, ax
- Jmp bx
- Mov ax, 0123H
- →第3步

主观题 10分

设置

8086计算机上，如果希望能够寻址到物理地址0x54000，请问段地址的最大、最小分别是多少？

作答