程序的机器级表示 I：基础
Machine-Level Programming

教师：郑贵滨
计算机科学与技术学院
哈尔滨工业大学

程序的机器级表示 I：基础

- Intel CPU及架构的发展史
- IA32处理器体系结构
- 汇编语言
- Linux汇编程序

Intel x86 处理器

- 笔记本、台机、服务器市场的统治者
- 进化设计
  - 向后兼容，直至1978年推出的8086CPU
  - 与时俱进：不断引入新特征
- 复杂指令集计算机(Complex instruction set computer,CISC)
  - 指令多、指令格式多
    - Linux程序设计只用到其中较小的子集
  - 性能难与精简指令计算机(Reduced Instruction Set Computers,RISC)相比
  - 但，Intel做到了：主要在速度方面、功耗不低

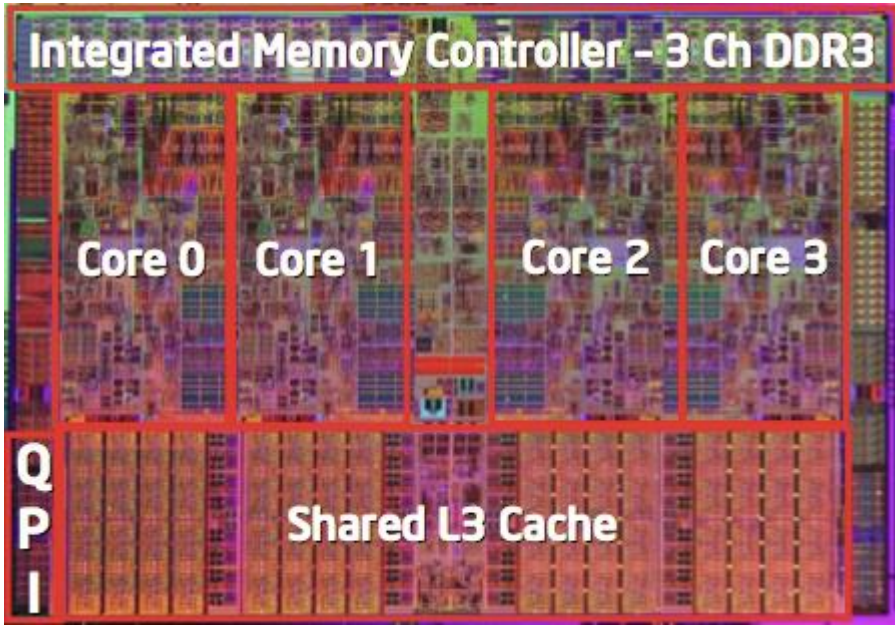
Intel x86 进化的里程碑

名字	*时间*	*晶体管数量*	*主频*
8086	1978	29K	5-10
80286	1982	134K	20
386	1985	275K	16-33
Pentium 4E	2004	125M	2800-3800
Core 2	2006	291M	1060-3500
Core i7	2008	731M	1700-3900

- 第一个16位intel处理器，主要用于IBM PC & DOS
- 1MB 地址空间，程序可用640KB，8087浮点运算协处理器
- IBM PC-AT & Windows、更多寻址模式
- 第一个32位intel处理器, 称为IA32
- 增加 “平坦寻址”(flat addressing),可运行Unix
- 第一个64位Intel x86处理器,称为 x86-64，超线程(*hyperthreading*)
- 第一个多核处理器，不支持超线程 （Core酷睿）
- 4核处理器、支持超线程

Intel x86 处理器(续...)

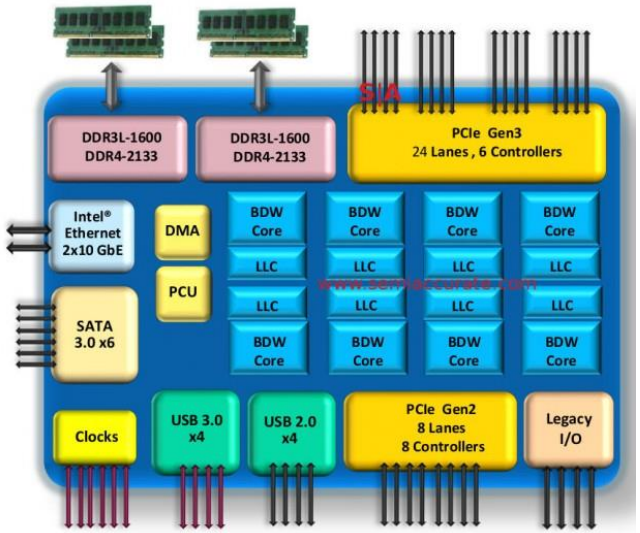
- 机器的演变
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2001 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M



- 增加的特征
  - 支持多媒体计算的指令
  - 支持更高效的条件运算指令
  - 从32位进化到64位
  - 多核

Intel最新状态

- **2015 Core i7 Broadwell架构: 14nm工艺、低功耗**
  - 台机：Intel Corei7 6950X
    - 10核/20线程25MB
    - 3.0-3.5GHz
    - 140W
    - 集成显卡
  - 服务器：Xeon(至强) E5-2699 v4
    - 22核/44线程/55MB/2.2-3.6GHz/145W



- **2016 Xeon E7-8890 v4**
  - 24核/48线程/60MB/2.2-3.4GHz/ 165w
- **2017 Core i9-7980XE：18核/36线程**

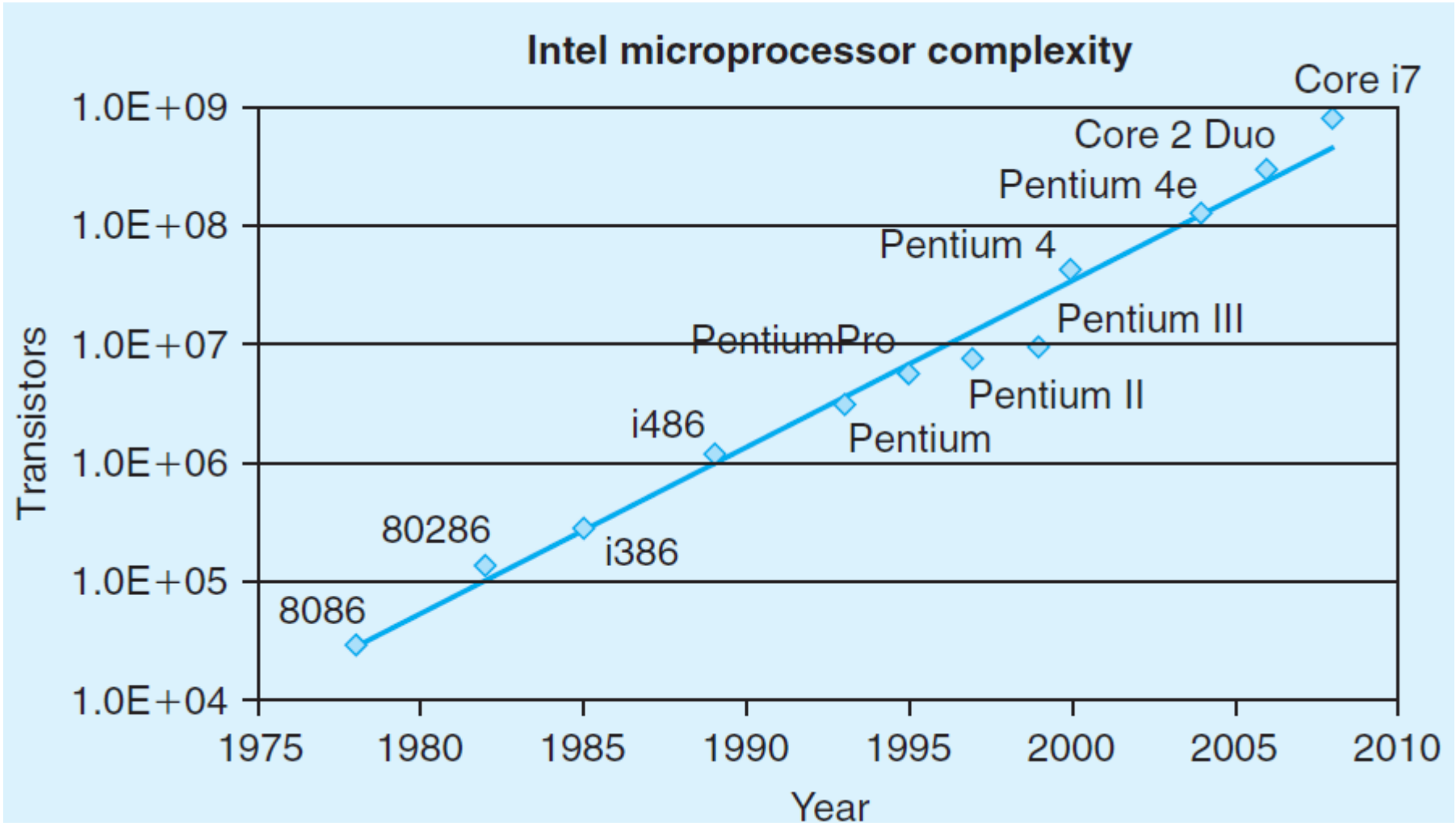
x86 的克隆: Advanced Micro Devices (AMD)

- 历史
  - AMD 紧随Intel
  - 慢一点点、便宜很多!
- 随后
  - 从Digital Equipment Corp. 和其他发展趋势下降的公司招募顶级电路设计师
  - Opteron（皓龙处理器）: Pentium 4的强劲对手
  - 研发了x86-64, 向64扩展的自主技术
- 近期
  - 与Intel合作,引领世界半导体技术的发展
  - AMD 已经落后

Intel的64位CPU发展史

- 2001: Intel激进地尝试从IA32跨到IA64
  - 采用完全不同的架构(Itanium)
  - 仅将运行IA32程序作为一种遗产
  - 性能令人失望
- 2003: AMD采用了进化的解决方案
  - x86-64 (现在称为 “AMD64”)
- Intel 被动聚焦于IA64
  - 难以承认错误或承认AMD更好
- 2004: Intel 宣布了IA32的扩充EM64T
  - 64位技术的扩展内存
  - 几乎和x86-64一样
- 除了低端x86处理器外，都支持x86-64
  - 很多程序依旧运行在32位模式

摩尔定律(Moor's Law)



机器级程序设计I: 基础

- Intel CPU及架构的发展史
- **IA32处理器体系结构**
- 汇编语言
- Linux汇编程序

IA32处理器体系结构

- 0、概念
- 1、微机的基本结构
- 2、IA32的寄存器
- 3、IA32的内存管理
- 4、指令的执行过程——指令执行周期
- 5、程序是如何运行的
- 6、系统是如何启动的

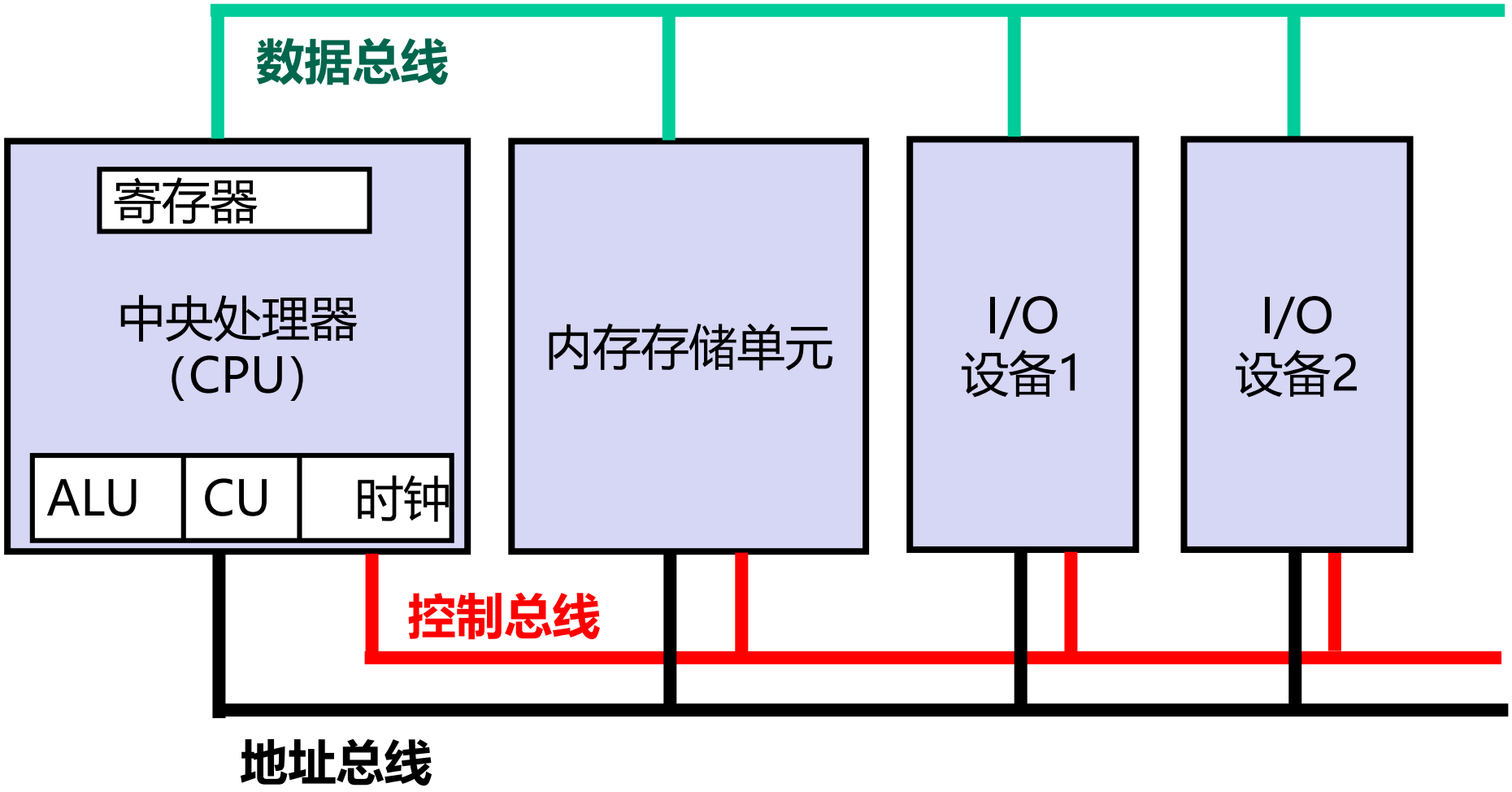
0、概念

- 架构(Architecture)

  即，指令集体系结构，处理器设计的一部分，理解或编写汇编/机器代码时需要知道。
  - 例如：指令集规范,寄存器
- 微架构(Microarchitecture)

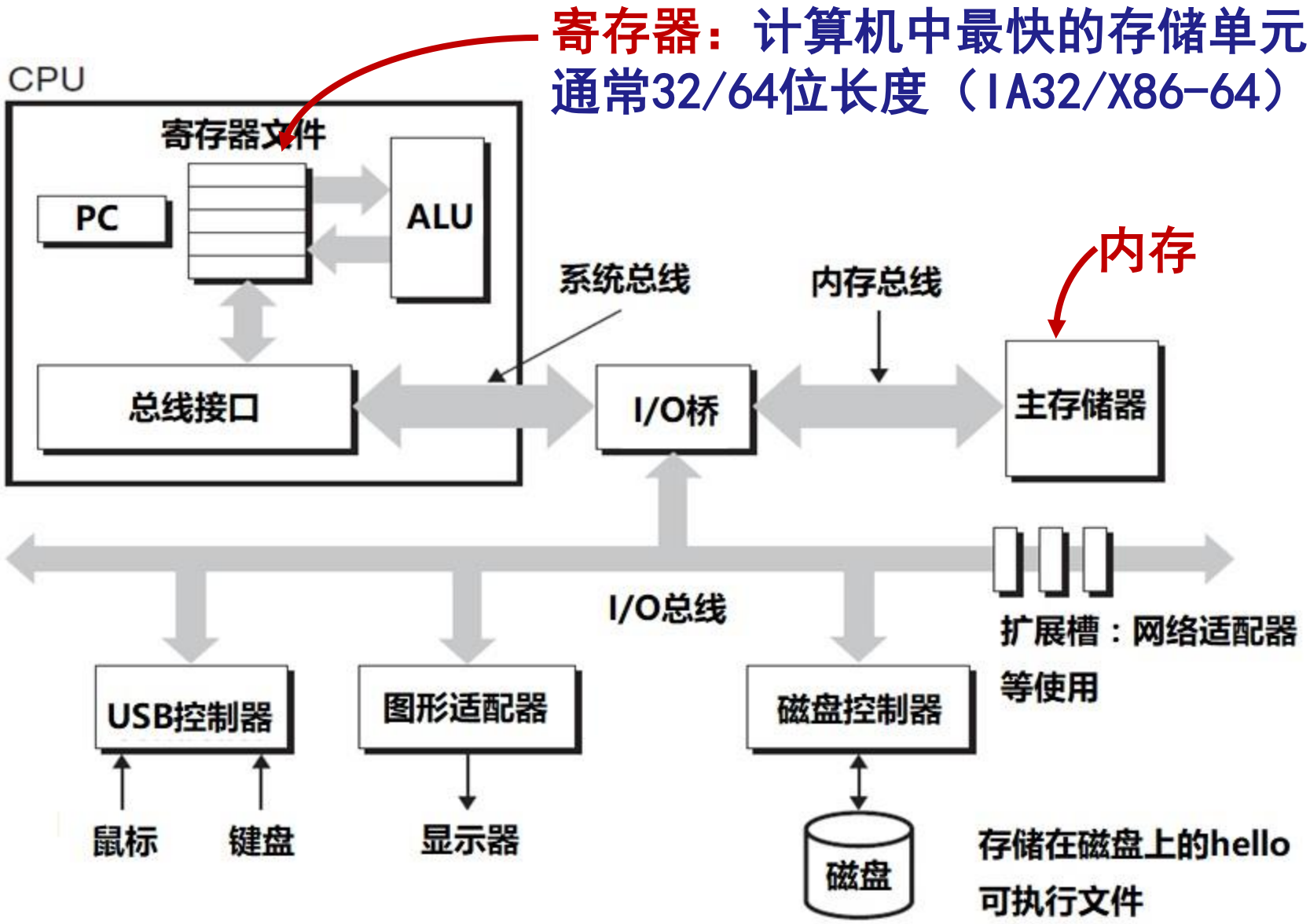
  架构的具体实现
  - 例如：缓存大小、核心的频率
- 代码格式(Code Forms)
  - 机器码(Machine Code):处理器接执行的字节级程序
  - 汇编码(Assembly Code): 机器码的文本表示
- 指令体系结构(ISA)例子:
  - Intel: **x86, IA32**, Itanium, **x86-64**
  - ARM: 广泛用于移动电话

1 微机的基本结构



微机的结构示意图

1 微机的基本结构


寄存器：计算机中最快的存储单元、
通常32/64位长度（IA32/X86-64）
CPU
寄存器文件
PC
ALU
总线接口
系统总线
内存总线
内存
I/O桥
主存储器
I/O总线
扩展槽：网络适配器
等使用
USB控制器
图形适配器
磁盘控制器
鼠标
键盘
显示器
磁盘
存储在磁盘上的hello
可执行文件

2、IA32的寄存器

- 2.1 基本寄存器

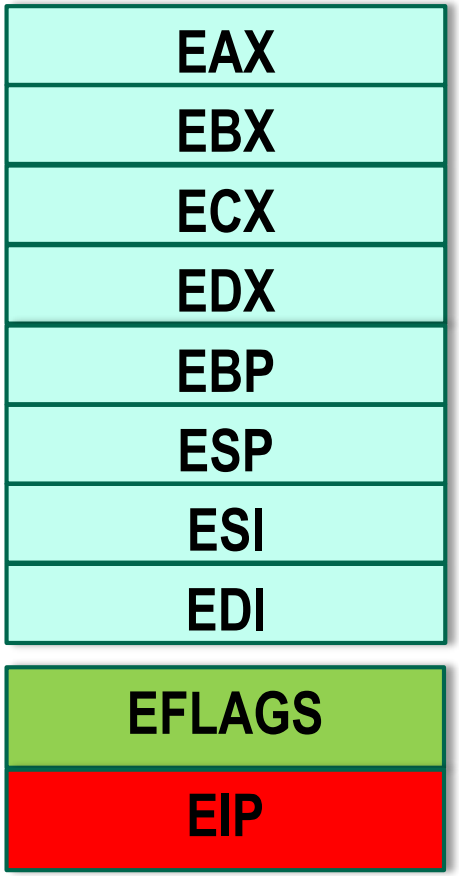
  寄存器是CPU内部的高速储存单元，访问速度比常规内存快得多。包括：

  - ✓ **8个32位通用寄存器**
  - ✓ **6个16位段寄存器**
  - ✓ **一个存放处理器标志的寄存器(EFLAGS)**
  - ✓ **一个指令指针寄存器(EIP)**

- 2.2 系统寄存器
- *2.3 浮点单元*

2、 IA32的寄存器

- 2.1 基本寄存器



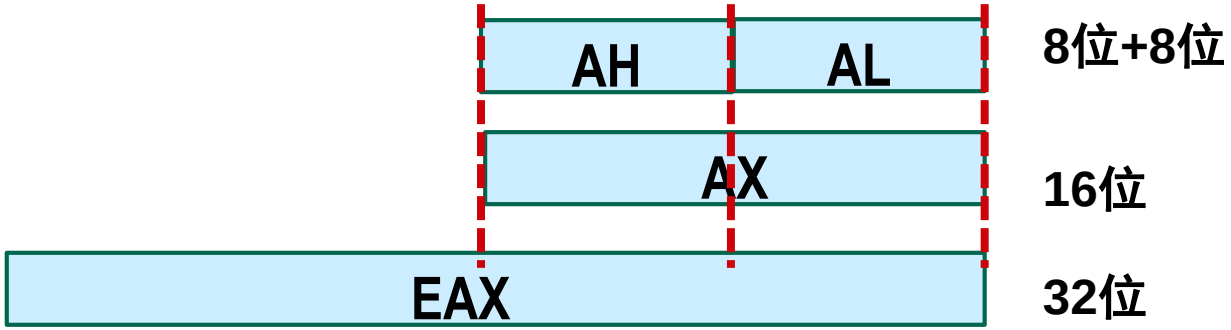
16位段寄存器

IA-32处理器的基本寄存器

2、 IA32的寄存器

2.1.1 通用寄存器

- 32位通用寄存器：主要用于算术运算和数据传送



32位	16位	高8位	低8位
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

2、 IA32的寄存器

- 2.1.1 通用寄存器

EBP ESP ESI EDI只有低16位有特别名字，通常在编写实地址模式程序时使用：

32位	16位
EBP	BP
ESP	SP
ESI	SI
EDI	DI

2、 IA32的寄存器

- 2.1.1 通用寄存器

  通用寄存器的特殊用法

  - **EAX**：扩展累加寄存器。在乘法和除法指令中被自动使用；
  - **ECX**：循环计数器。
  - **ESI**和**EDI**：扩展源指针寄存器和扩展目的指针寄存器。用于内存数据的存取；
  - **ESP**：扩展堆栈指针寄存器。一般不用于算术运算和数据传送，而用于寻址堆栈上的数据。
  - **EBP**：扩展帧指针寄存器。用于引用堆栈上的函数参数和局部变量；

2、 IA32的寄存器

- 2.1.2 段寄存器
  - 在实地址模式下，段寄存器用于存放段的基址；段寄存器包括：CS、SS、DS、ES、FS、GS。

✓CS往往用于存放代码段(程序的指令)地址；

✓DS存放数据段(程序的变量)地址；

✓SS存放堆栈段(函数的局部变量和参数)地址；

✓ES、FS和GS则可指向其他数据段。

  - 保护模式下，段寄存器存放段描述符表的指针(索引)。

2、 IA32的寄存器

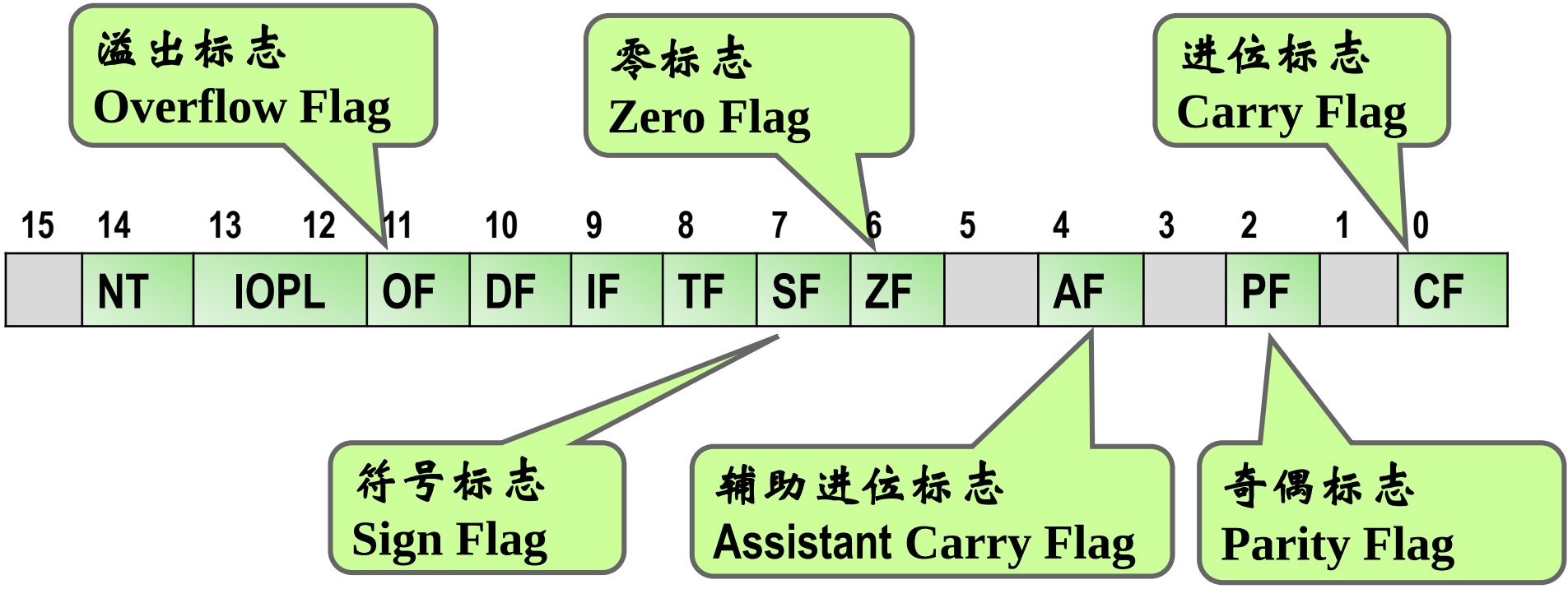
- 2.1.3 指令指针寄存器EIP
  - 也称为：程序计数器(**Program counter,PC)**
  - EIP始终存放下一条要被CPU执行的指令的地址。
  - 有些机器指令可以修改EIP，使程序分支转移到新的地址执行。例如：JMP, RET

2、 IA32的寄存器

- 2.1.4 EFLAGS寄存器(标志寄存器、条件码寄存器)
  - EFLAGS由控制CPU的操作或反映CPU某些运算结果的二进制位构成。
  - 处理器标志包括两种类型：**状态标志**和**控制标志**。
    - 说某标志被设置意味着使其等于1；被清除意味着使其等于0
    - 程序员可以通过设置EFLAGS中的**控制标志控制CPU的操作，如方向和中断标志**。
    - 一些机器指令可以测试和控制这些标志，例如：JC 或 STC

2、 IA32的寄存器

- 2.1.4 EFLAGS寄存器…
  - 其中反映CPU执行的算术和逻辑操作结果的状态标志，包括溢出、符号、零、辅助进位、奇偶和进位标志。



2、 IA32的寄存器

- 2.1.4 EFLAGS寄存器的状态标志（条件码）
  - **进位标志CF**：在无符号算术运算的结果，无法容纳于目的操作数中时被设置。
  - **溢出标志OF**：在有符号算术运算的结果位数太多，而无法容纳于目的操作数中时被设置。
  - **符号标志SF**：在算术或逻辑运算产生的结果为负时被设置。
  - **零标志ZF**：在算术或逻辑运算产生的结果为零时被设置。
  - **辅助进位标志AC**：8位操作数的位3到位4产生进位时被设置,BCD码运算时使用。
  - **奇偶标志PF**：结果的最低8位中，为1的总位数为偶数，则设置该标志；否则清除该标志。

2、 IA32的寄存器

2.2 系统寄存器

仅允许运行在最高特权级的程序（例如：操作系统内核）访问的寄存器，任何应用程序禁止访问。

- 中断描述符表寄存器IDTR：保存中断描述符表的地址。
- 全局描述符表寄存器GDTR：保存全局描述符表的地址，全局段描述符表包含了任务状态段和局部描述符表的指针。
- 局部描述符表寄存器LDTR：保存当前正在运行的程序的代码段、数据段和堆栈段的指针。
- 任务寄存器：保存当前执行任务的任务状态段的地址。
- 调试寄存器：用于调试程序时设置端点。

2、 IA32的寄存器

- 2.3 浮点单元FPU

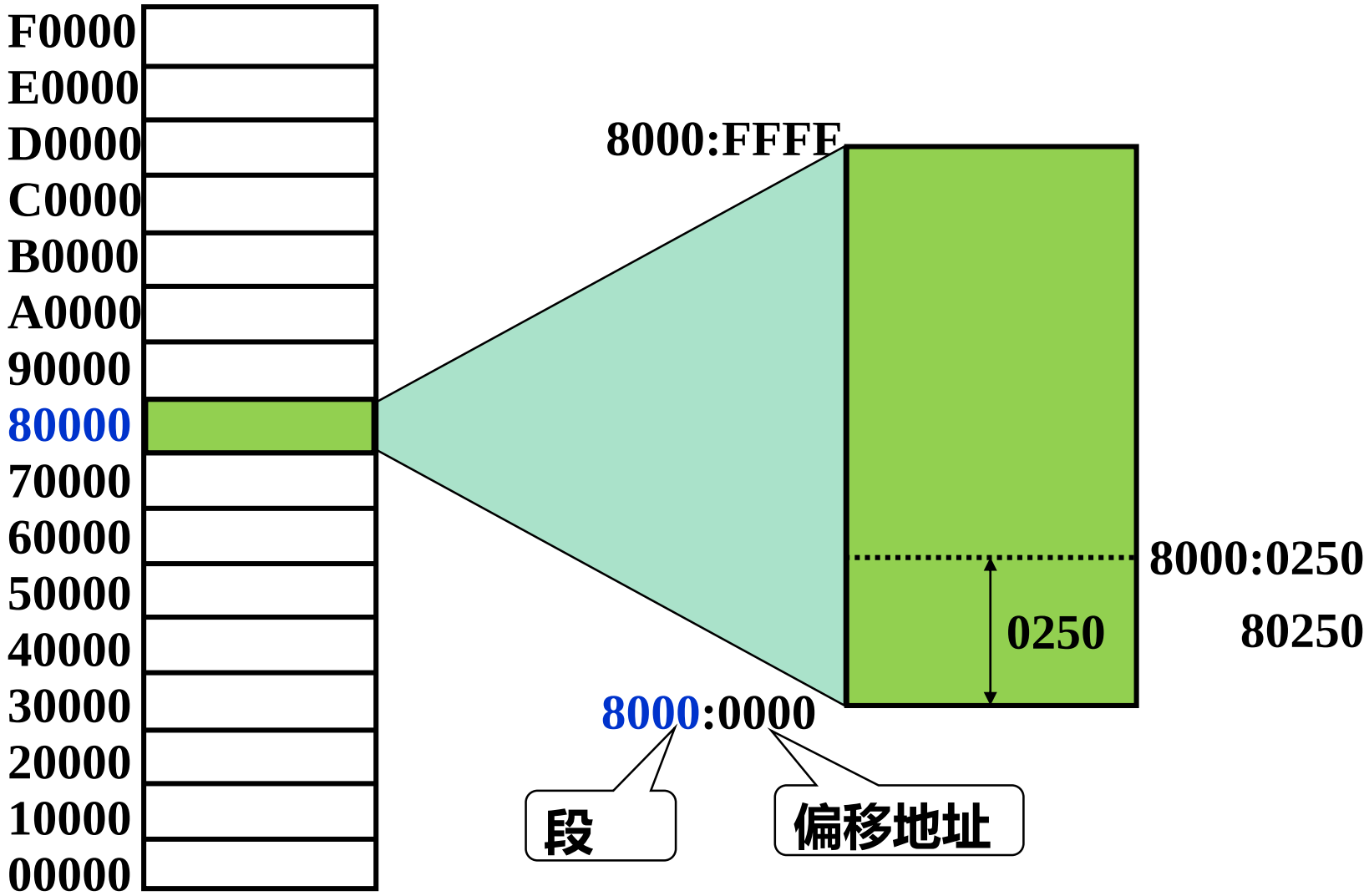
  适合于高速浮点运算，从Intel 486开始集成到主处理器芯片中。

  - 8个80位的浮点数据寄存器： st(0) — st(7)
  - 2个48位的指针寄存器
  - 3个16位的控制寄存器

3、IA32的内存管理

- 3.1 实地址模式
  - 在实地址模式下，处理器使用20位的地址总线，可以访问1MB(0~FFFFF)内存。
  - 8086的模式，只有16位的地址线，不能直接表示20位的地址，采用内存分段的解决方法。
  - 段：将内存空间划分为64KB的段Segment；
  - 段地址存放于16位的段寄存器中（CS、DS、ES或SS）：
    - CS用于存放16位的代码段的段地址/基地址
    - DS用于存放16位的数据段的段地址/基地址
    - SS用于存放16位的堆栈段的段地址/基地址
    - ES用于存放16位的附加段的段地址/基地址

段-偏移地址


F0000
E0000
D0000
C0000
B0000
A0000
90000
80000
70000
60000
50000
40000
30000
20000
10000
00000
8000:FFFF
8000:0250
0250
80250
8000:0000
段
偏移地址

- **20位线性地址的计算**

例：

08F1: 0100

➔ 08F1H*10H + 0100H = 09010H

8000:0250

➔ 8000H*10H + 0250H = 80250H

3、 IA32的内存管理

- 3.2 保护模式
  - 寄存器、地址总线都是32位
  - 32位地址总线寻址，每个程序可寻址4GB内存：0~FFFFFFFF
- 段寄存器无用？更有用：“保护”的实现！
  - 保护——访问权限：可写、可写程序的优先级、可执行
  - 段描述符(Segment Descriptor): 8字节,包括：段的参数、（保护）属性等
  - 段描述符集中存放➔段描述符表(Segment Descriptor Table)
  - 段寄存器：保存段描述符在段描述符表中的索引值，称为段选择器(Segment Selector)，而非段地址

3、 IA32的内存管理

- 3.2 保护模式
  - **段寄存器(CS、DS、SS、ES、FS和GS)指向段描述符:**
    - ✓CS**代码段描述符的索引值**
    - ✓DS**数据段描述符的索引值**
    - ✓SS**堆栈段描述符的索引值**

3、 IA32的内存管理

- 全局描述符表GDT(Global Descriptor Table)

  整个系统只有一个，包含：

  - 操作系统使用的代码段、数据段、堆栈段的描述符
  - 各任务/程序的LDT（局部描述符表）段

- 局部描述符表LDT（Local Descriptor Table）

  每个任务/程序有一个独立的LDT(intel 80386)，包含：

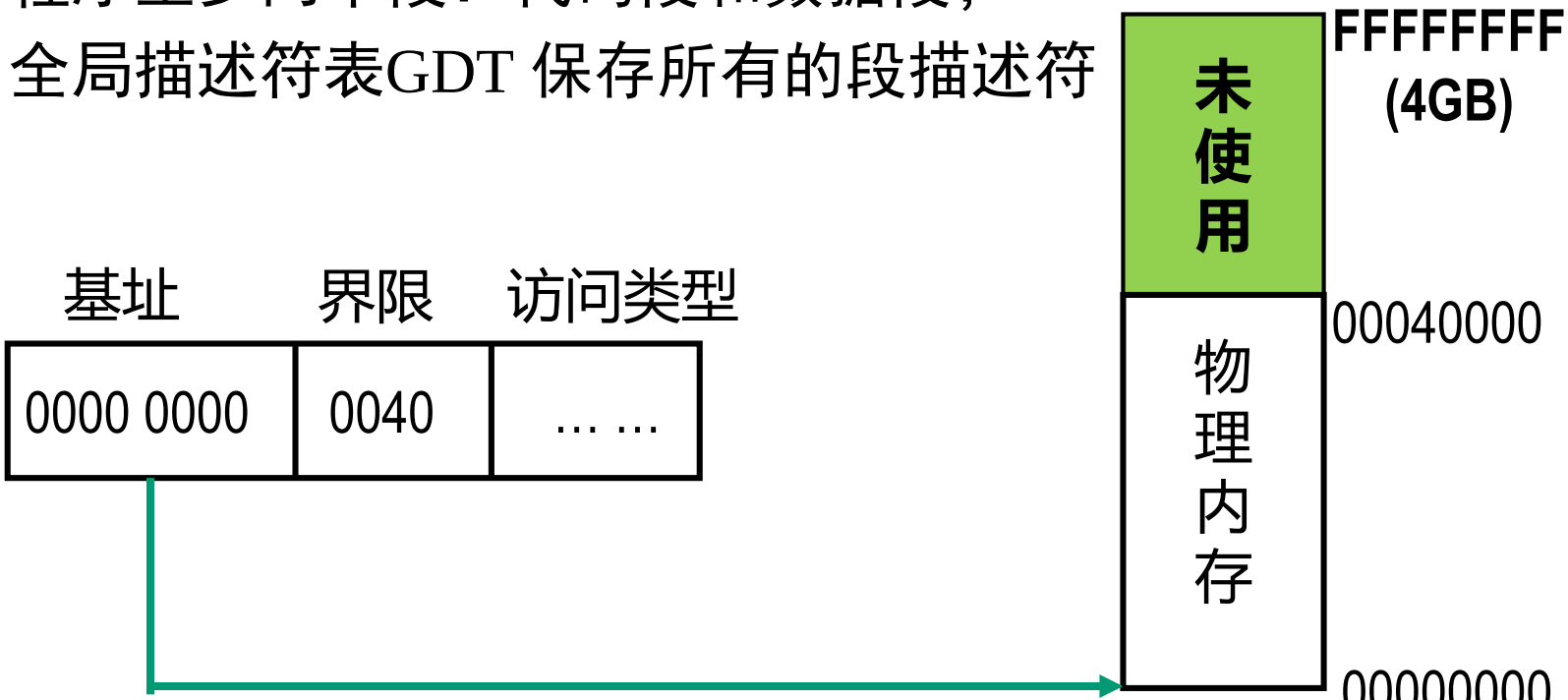
  - 对应任务/程序私有的代码段、数据段、堆栈段的描述符
  - 对应任务/程序使用的门描述符：任务门、调用门等。

3、 IA32的内存管理

- 48位的全局描述符表寄存器GDTR
  - 指向GDT(Global Descriptor Table)，即GDT在内存中的具体位置
- 16位局部描述符表寄存器LDTR
  - 指向LDT段在GDT中的位置（索引）
- 16位段选择器的编码
  - 高13位表示索引值
  - 0、1位表示程序的当前优先级RPL；
  - 第2位是TI位，表示段描述符的位置：
    - TI=0，段描述符在GDT中
    - TI=1，段描述符在LDT中

3、 IA32的内存管理

- 3.2.1 平坦分段模式
  - 所有段被映射到32位物理地址空间；
  - 程序至少两个段：代码段和数据段；
  - 全局描述符表GDT 保存所有的段描述符



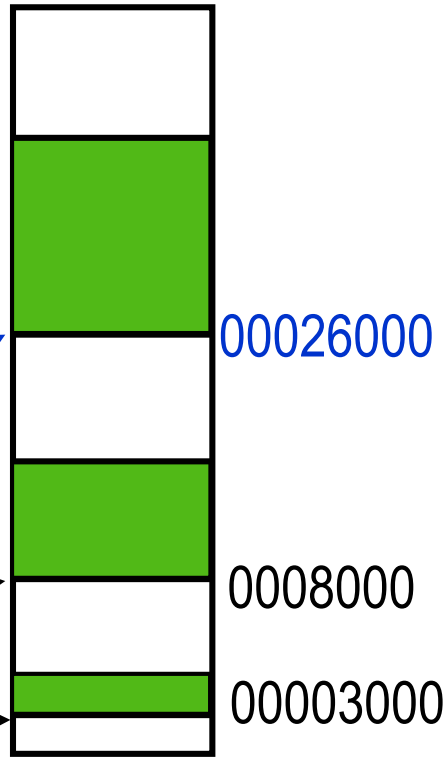
全局段描述符表(GDT)中的段描述符（示意图）

3、 IA32的内存管理

- 3.2.2 多段模式（Multi-Segment）
  - 每个任务/程序有自己的局部段描述符表(LDT)
  - 每个任务/程序有代码段、数据段、堆栈段
  - 每个段有独立的地址空间

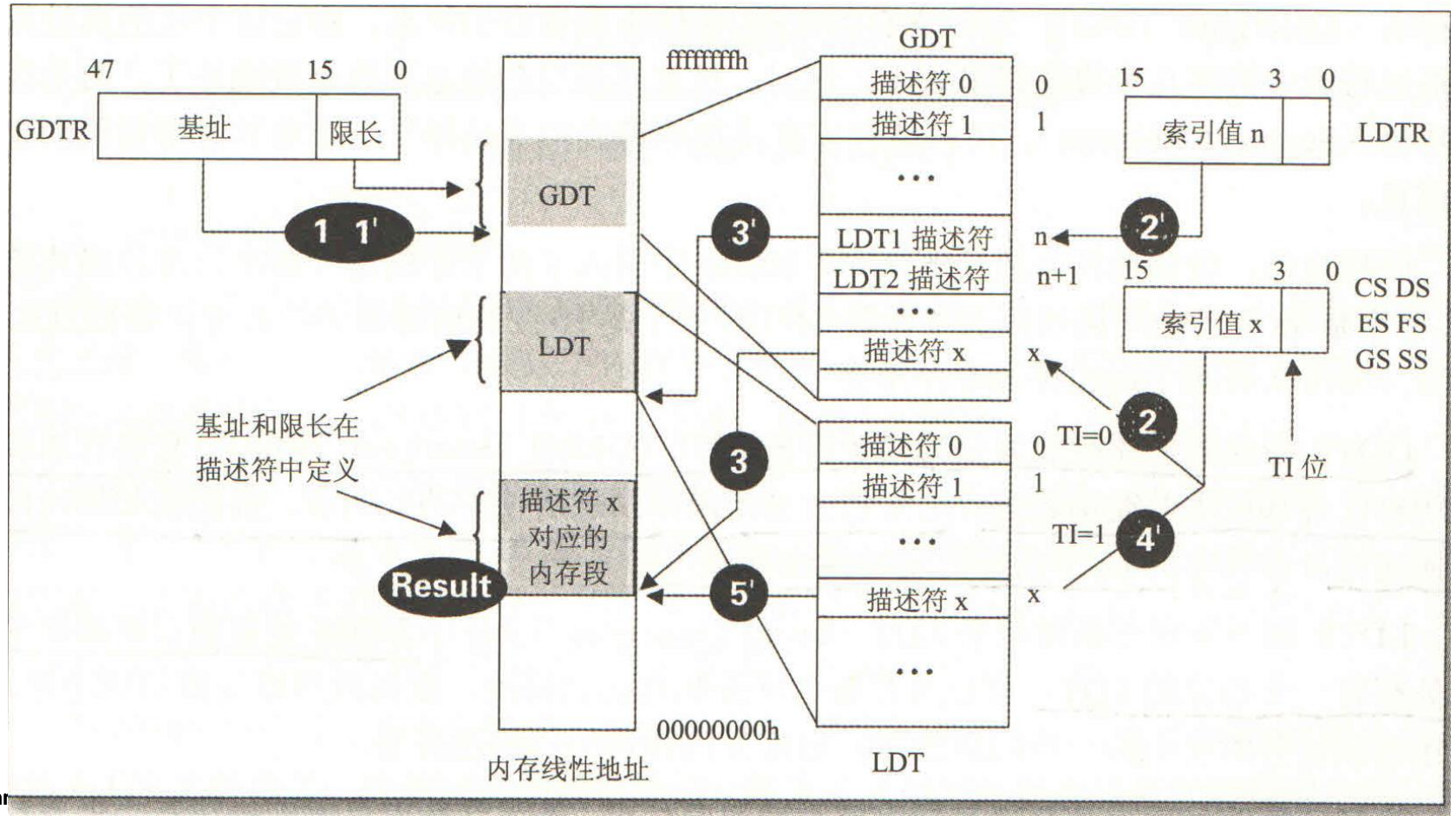
局部描述符表（LDT）

基址	界限	访问类型
0002 6000	0010	……
0000 8000	000A	……
0000 3000	0002	……



保护模式下的段寻址...

- 段寄存器为全局描述符表项的寻址：1-2-3
- 段寄存器为局部描述符表项的寻址：1’- 2’- 3’-4’-5’



3、 IA32的内存管理

- **3.2.3 分页(Paging)**
  - 将内存分割成4KB大小的页（Pages），同时将程序段的地址空间按内存页的大小进行划分。
  - 分页模式的基本思想：当任务运行时，当前活跃的执行代码保留在内存中，而程序中当前未使用的部分，将继续保存在磁盘上。当CPU需要执行的当前代码存储在磁盘上时，产生一个缺页错误，引起所需页面的换进(从磁盘载入内存)。
  - 通过分页以及页面的换进、换出，一台内存有限的计算机上可以同时运行多个大程序，让人感觉这台机器的内存无限大，因此称为虚拟内存。

4、指令周期

- 指令周期：单条机器指令的执行可以分解成一系列的独立操作，这些操作被称为指令执行周期。
- 单条指令的执行有三种基本操作：取指令、解码和执行。
- 程序在开始执行之前必须首先被装入内存。执行过程中，指令指针(IP)包含着要执行的下一条指令的地址，指令队列中包含了一条或多条将要执行的指令。
- 当CPU执行使用内存操作数的指令时，必须计算操作数的地址，将地址放在地址总线上并等待存储器取出操作数。

4、指令周期

- 如指令使用内存操作数，需要5种基本操作：
  - **取指令：** 控制单元从指令队列取得指令并增加指令指针RIP/EIP的值。
  - **解码：** 控制单元确定指令要执行的操作，把输入操作数传递给算术逻辑单元ALU，并向ALU发送信号指明要执行的操作。
  - **取操作数：** 如果使用了内存操作数，控制单元通过读操作，获取操作数，复制到**内部寄存器（用户程序不可见）**；
  - **执行：** 算术逻辑单元执行指令，以有名寄存器、内部寄存器为操作数，进行计算，将结果数据送出，并更新反映处理器状态的状态标志。
  - **写内存（存储输出/目的操作数）：** 如输出/目的操作数为内存操作数，控制单元就执行一个写操作，将结果数据写入内存。
- 机器指令的执行至少需要一个时钟周期。

5、程序是如何运行的

- 前提：

  计算机(CPU)的工作过程

  （1）从CS:IP/EIP/RIP指向内存单元读取指令，读取的指令进入指令缓冲器；

  （2）令IP/EIP/RIP指向下一条指令：

  IP/EIP/RIP = IP/EIP/RIP + 所读取指令的长度

  （3）执行指令。 转到步骤 （1），重复这个过程。

5、程序是如何运行的

- (1) 装入和执行进程

计算机操作系统(OS)加载和运行程序的步骤：

- 用户发出特定程序的命令。
- OS在当前磁盘目录中查找程序文件名，如果未找到就在预先定义的目录列表中查找，如果还是找不到，就发出一条错误信息；
- 如找到程序文件，OS获取磁盘上程序文件的基本信息，如文件大小、在磁盘驱动器上的物理位置等；
- OS确定下一个可用的内存块的地址，并将程序文件载入内存，然后将程序的大小和位置等信息登记在描述符表中；

5、程序是如何运行的

(1) 装入和执行进程(续...)

- 操作系统执行一条分支转移指令，使CPU从程序的第一条机器指令开始执行。一旦程序运行就被称为一个进程，操作系统为进程分配一个唯一的标识号称为进程ID。
- 进程自身开始运行，操作系统的任务就是跟踪进程的执行并响应进程对系统资源的请求。
- 进程终止时，其句柄被删除，使用的内存也被释放以便能够由其他程序使用。

5、程序是如何运行的

- **(2)多任务**
  - 操作系统运行的可以是一个进程或一个执行线程。当操作系统能够**同时**运行多个任务时，就被认为是多任务的。

    注意：多任务中进程的“**同时**”运行包含的是**并发**运行的含义。
  - 并发可以看成是在系统中同时有几个进程在活动着，也就是同时存在几个程序的执行过程。如果进程数与处理机数相同，则每个进程占用一个处理机，但更一般的情况是处理机数少于进程数，于是处理机就应被共享，在进程间切换使用。如果相邻两次切换的时间间隔非常短，而观察时间又相当长，那么各个进程都在前进，造成一种宏观上并发运行的效果。

5、程序是如何运行的

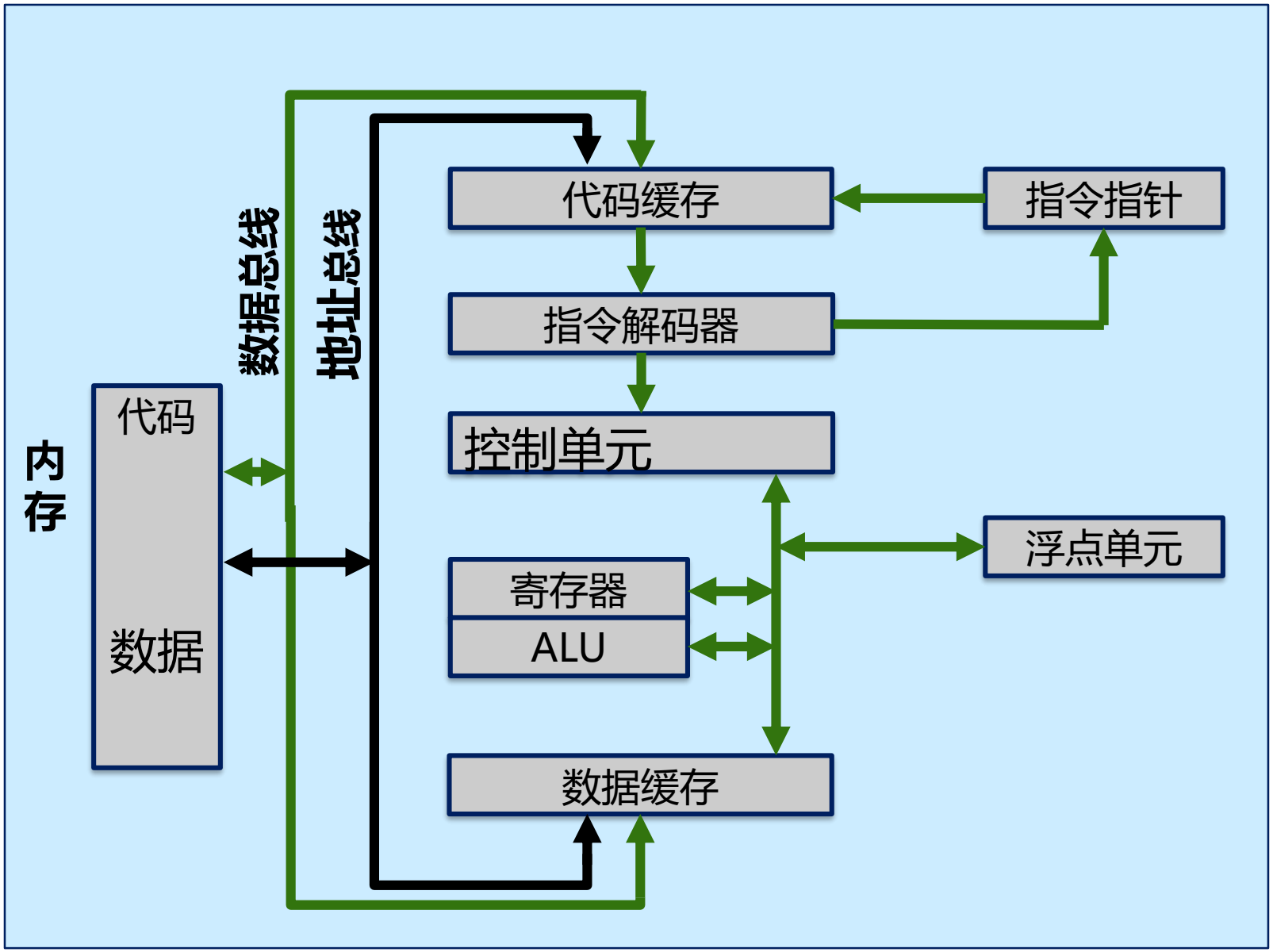
■ 多任务的实现

如何实现处理器在各个进程之间共享？

操作系统的调度程序(scheduler)为每个任务分配一小部分CPU时间(称为时间片)，在时间片内，CPU将执行一部分该任务的指令，并在时间片结束的时候停止执行，并迅速切换到下一个任务的指令执行。通过在多个任务之间的快速切换，给人以同时运行多个任务的假象。

6、 计算机是如何启动的

- 8086 PC的启动方式
  - 在 8086CPU 加电启动或复位后（ 即 CPU刚开始工作时）CS和IP被设置为CS=FFFFH，IP=0000H，即在8086PC机刚启动时，CPU从内存FFFF0H单元中读取指令执行，FFFF0H单元中的指令是8086PC机开机后执行的第一条指令。
  - F0000~FFFFFH:系统ROM，BIOS中断服务例程。



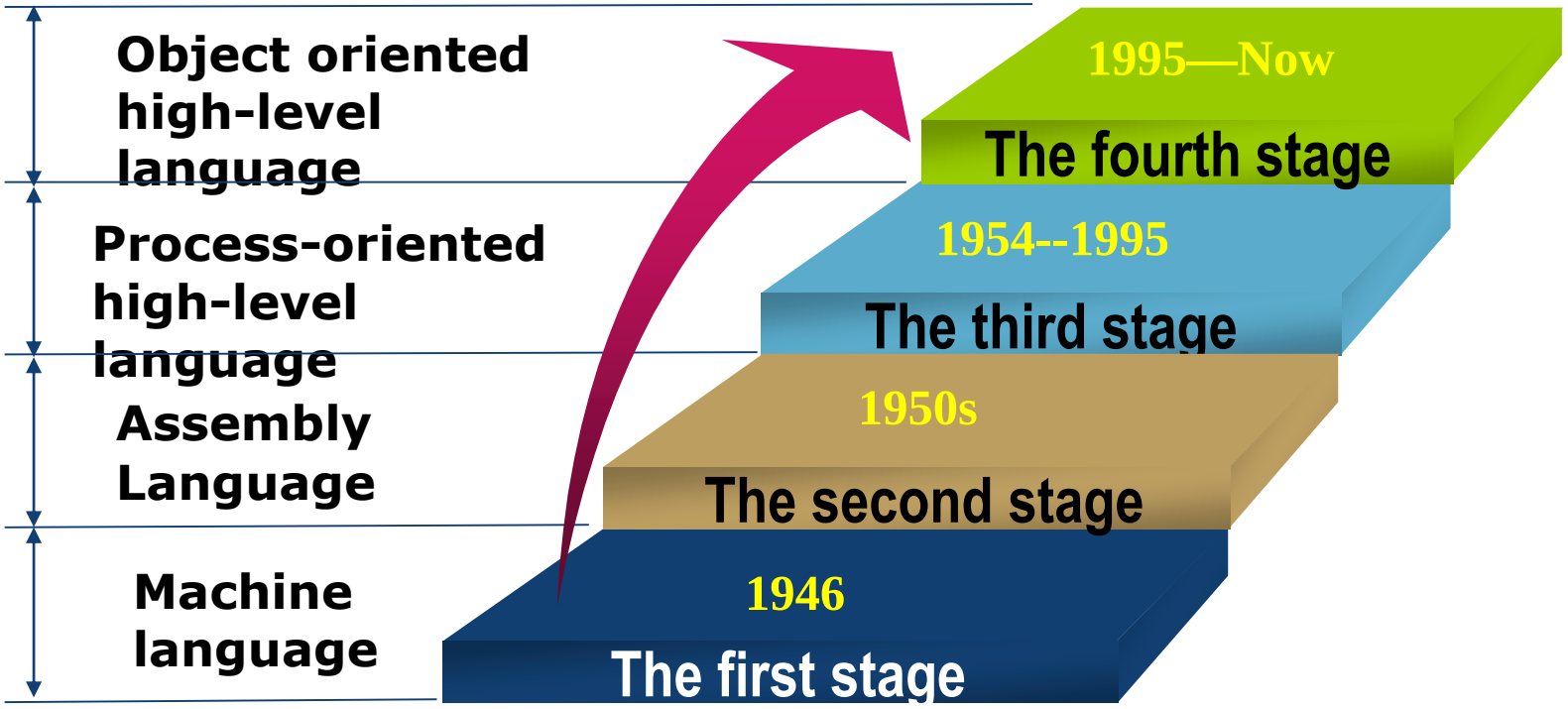
简化的奔腾CPU结构图

机器级程序设计I: 基础

- Intel CPU及架构的发展史
- IA32处理器体系结构
- 汇编语言
- Linux汇编程序

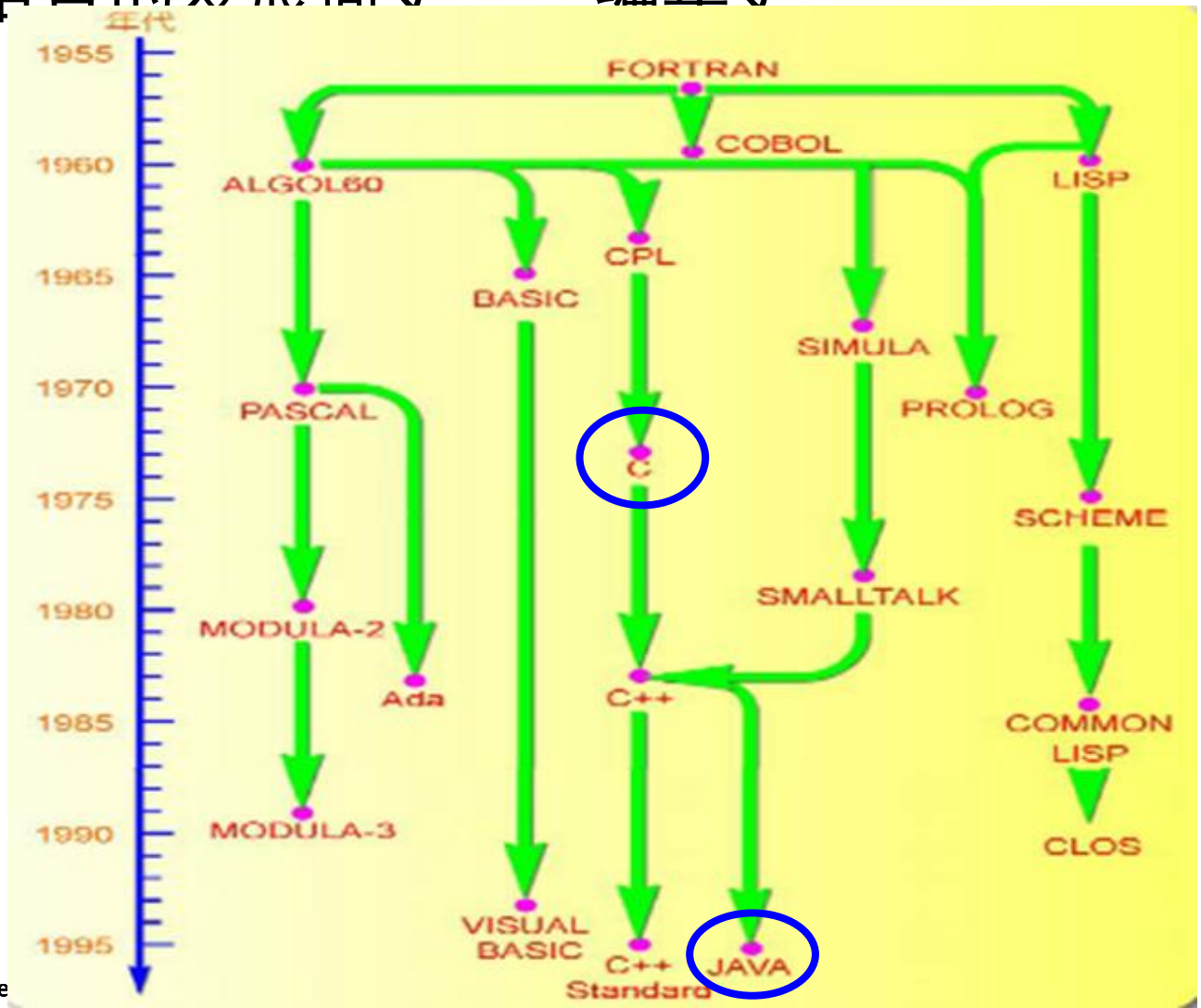
汇编语言简介

- 从计算机诞生至今，编程语言总数超过2500种
- 编程语言的发展简史——四个阶段



汇编语言简介

- 编程语言的发展简史——编年史



（1） 机器语言

- 是一种二进制语言，由二进制数0、1组成的指令代码的集合，机器能直接识别和执行。
- 每一条语句都是二进制形式的代码。

  例如：1000 0000（加法）
- 每条指令都简单到能够用相对较少的电子电路单元即可执行。
- 各种机器的指令系统互不相同。

（1） 机器语言

- 采用穿孔纸带保存程序(1打孔，0不打孔)

优点：

1.速度快

2.占存储空间小

3.翻译质量高

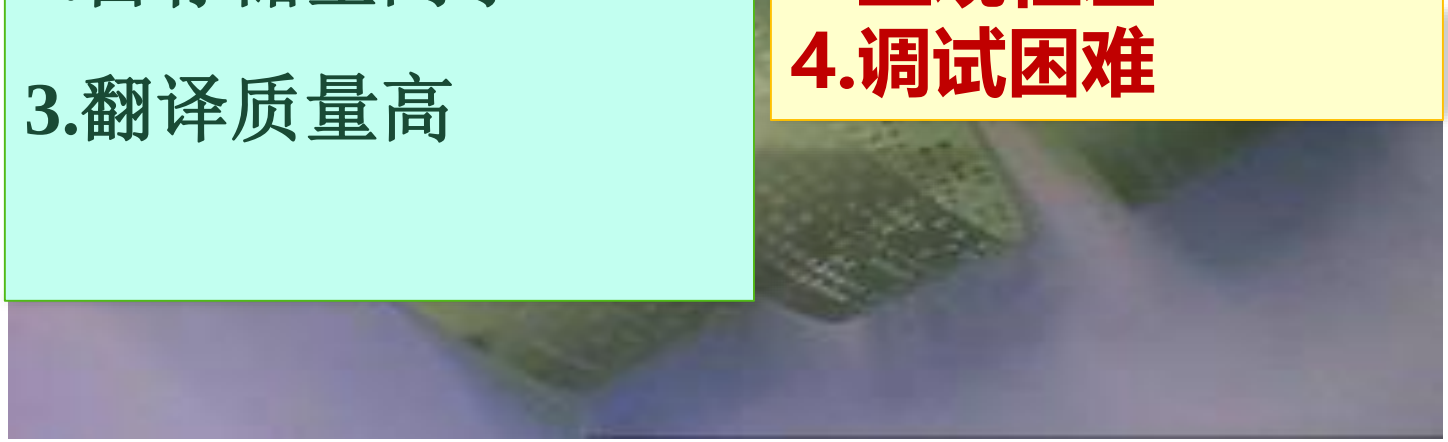
缺点：

1.可移植性差

2.编译难度大

3.直观性差

4.调试困难



（1） 机器语言

- 示例

应用8086CPU完成运算：

S = 768 + 12288 - 1280

机器指令码：

10110000000000000000011

00000101000000000110000

00101101000000000000101

假如将程序错写成以下这样，请找处错误：

10110000000000000000011

00000101000000000110000

00010110100000000000101

（2）汇编语言

- 汇编语言的产生
  - 汇编语言指令——汇编语言的主体
    - 汇编指令是机器指令便于记忆和阅读的书写格式——**助记符，**与人类语言接近，add、mov、sub和call等。
    - 用助记符代替机器指令的操作码，用地址符号或标号代替指令或操数的地址。

      机器指令： 1000100111011000

      操　　作：寄存器bx的内容送到ax中

      汇编指令： mov %bx, %ax,
    - **汇编指令同机器指令是一一对应的关系。**

（2）汇编语言

- 示例

应用8086CPU完成运算：

S = 768 + 12288 - 1280

机器指令：

1011000000000000000011

00000101000000000110000

00101101000000000000101

汇编指令：

```
movw  $768,  S      # S是长度16位的字变量
addw  $12288,  S
subw  $1280,  S
```

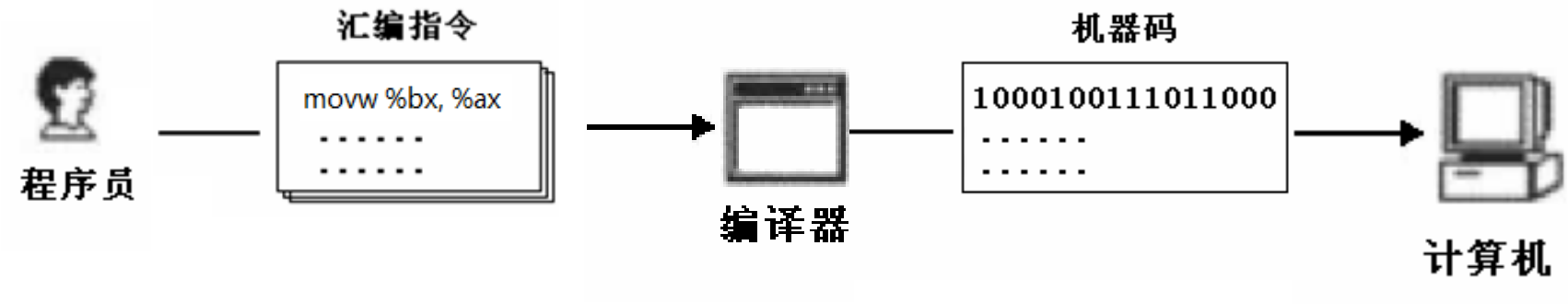
（2）汇编语言

- 除汇编指令，汇编语言还包括：
  - 伪指令　（由编译器执行）
  - 其它符号（由编译器识别）

汇编指令是汇编语言的核心，决定汇编语言的特性。

- **汇编语言的程序如何运行？**

  计算机能读懂的只有机器指令



（2）汇编语言

优点：
1.执行速度快；
2.占存储空间小；
3.可读性有所提高。

缺点：
1.类似机器语言；
2.可移植性差；
3.与人类语言还相差很悬殊。

（3）高级语言

- C++和Java等高级语言与汇编语言的关系

  C++和Java等**高级语言与汇编语言及机器语言之间是一对多的关系**。一条简单的C++语句会被扩展成多条汇编语言或者机器语言指令。

```
X = （Y ＋ 4 ）* 3;
```

```
movl Y,%eax
addl $4, %eax
movl $3, %ebx
imull  %ebx
movl  %eax, X
```

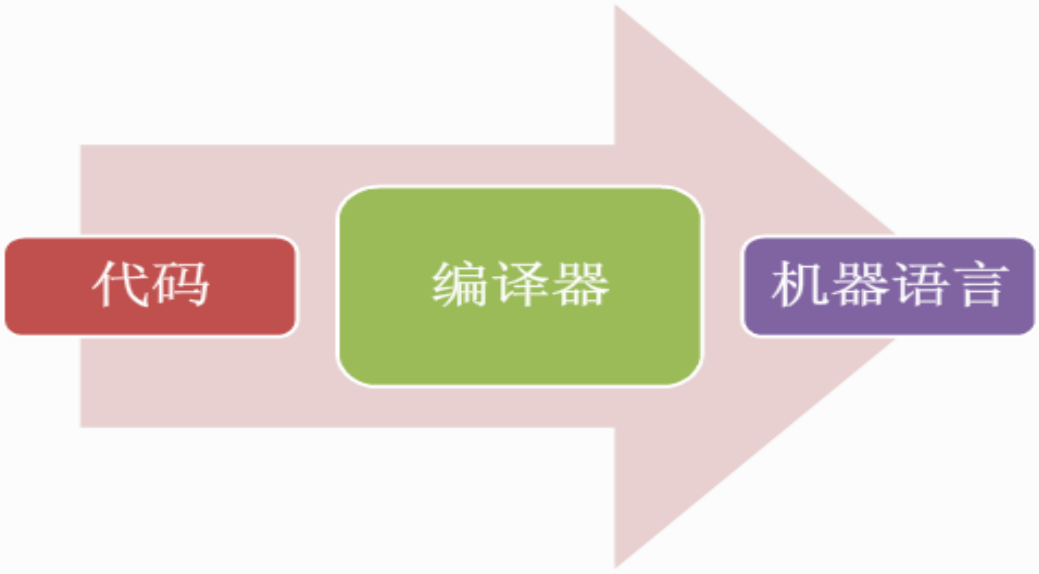
（4）高级语言到机器语言的转换方法

- 解释方式

通过解释程序，逐行转换成机器语言，转换一行运行一行。

- 编译方式（翻译方式）

通过编译程序（编译、链接）将整个程序转换成机器语言。



（5）汇编语言和高级语言的比较

- **可移植性：** 如果一种语言的源程序代码可以在多种计算机系统上编译运行，那么这种语言就是可移植的。
  - 汇编语言总是和特定系列的处理器捆绑在一起。
  - 当今有多种不同的汇编语言，每种都是基于特定系列的处理器或特定计算机的。
  - 汇编语言没有可移植性。
  - 高级语言的可移植性好。

（5）汇编语言和高级语言的比较

应用程序类型	高级语言	汇编语言
用于单一平台的中到大型商业应用软件	正式的结构化支持使组织和维护大量代码很方便	最小的结构支持使程序员需要人工组织大量代码，使各种不同水平的程序员维护现存代码的难度极高
硬件驱动程序	语言本身未必提供直接访问硬件的能力，即使提供了也因为要经常使用大量的技巧而导致维护困难	硬件访问简单直接。当程序很短并且文档齐全时很容易维护
多种平台下的商业应用软件	可移植性好，在不同平台可以重新编译，需要改动的源代码很少	必须为每种平台重新编写程序，通常要使用不同的汇编语言，难于维护
需要直接访问硬件的嵌入式系统和计算机游戏	由于生成的执行代码过大，执行效率低	很理想，执行代码很小并且运行很快

（6）为什么学汇编？

- 深入了解计算机体系结构和操作系统
- 在机器层次思考并处理程序设计中遇到的问题
- 在许多专业领域，汇编语言起主导作用：
  - 嵌入式系统
  - 游戏程序
  - 设备驱动程序
- 软件优化，通过汇编语言使用最新最快的CPU指令，获得最高的处理速度。***…速度比较示例***

❖**后继课程的学习**

Linux汇编程序：AT&T 格式程序

```
 #hello.s
.data # 数据段声明
    msg : .string "Hello, world! --------- AT&T ASM\r\n " # 要输出的字符串
    len = . - msg # 字串长度
.text # 代码段声明
.global _start # 指定入口函数
_start: # 在屏幕上显示一个字符串
    movl $len, %edx # 参数三：字符串长度
    movl $msg, %ecx # 参数二：要显示的字符串
    movl $1, %ebx # 参数一：文件描述符(stdout)
    movl $4, %eax # 系统调用号(sys_write)
    int $0x80 # 调用内核功能
    # 退出程序
    movl $0,%ebx # 参数一：退出代码
    movl $1,%eax # 系统调用号(sys_exit)
    int $0x80 # 调用内核功能
```

Linux汇编程序： Intel格式程序

```
; hello.asm
.data ; 数据段声明
    msg db "Hello, world! --------- Intel ASM .", 0xA ; 要输出的字符串
    len equ $ - msg ; 字串长度
.text ; 代码段声明
global _start ; 指定入口函数
_start: ; 在屏幕上显示一个字符串
    mov edx, len ; 参数三：字符串长度
    mov ecx, msg ; 参数二：要显示的字符串
    mov ebx, 1 ; 参数一：文件描述符(stdout)
    mov eax, 4 ; 系统调用号(sys_write)
    int 0x80 ; 调用内核功能
    ; 退出程序
    mov ebx, 0 ; 参数一：退出代码
    mov eax, 1 ; 系统调用号(sys_exit)
    int 0x80 ; 调用内核功能
```

Linux汇编程序——编译、链接

- **两种汇编格式：AT&T 汇编、Intel汇编**
- **汇编器**
  - GAS汇编器——**AT&T汇编格式** Linux 的标准汇编器，GCC 的后台汇编工具

    as -gstabs -o hello.o hello.s

    -gstabs：生成的目标代码中包含符号表，便于调试。
  - NASM——**intel汇编格式**
    - 提供很好的宏指令功能，支持的目标代码格式多，包括 bin、a.out、coff、elf、rdf 等。
    - 采用人工编写的语法分析器，执行速度要比 GAS 快

    nasm -f elf hello.asm
- **连接器**

ld 将目标文件链接成可执行程序：

ld -o hello hello.o