

程序的机器级表示 I: 基础 part01

Machine-Level Programming

教师：夏文

计算机科学与技术学院

哈尔滨工业大学深圳硬件与系统教研室

章节要求

- 掌握状态标志
- 尤其是溢出和进位的计算
- 了解堆栈
- 了解寻址方式
- 了解编译、链接
- IA32仅作了解
- 掌握经典例题

机器级程序设计I: 基础 (目录)

- 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 CPU及架构的发展史（目录）

- C语言的数据与操作
- 第一代8086处理器（非考研范围）
- 寄存器的分类
- 8086指令系统

审视：C语言的数据与操作

- 数据：常量、变量(全局/局部/静态)、表达式、类型、宏
- 赋值 = ,逗号操作符, 赋初值/不赋初值
- 类型转换(隐式或显式) unsigned/char/int/long/float/double
- Sizeof
- 算术操作：+ - * / % ++ -- 取正/负+- 复合"+="等
- 逻辑/位操作：逻辑&& || ! 位 & | ~ ^ 移位>> << 复合操作如 "|=" 或 "<<="等
- 关系操作：== != > < >= <=
- 数组/指针/结构操作：A[i] &v *p s.id p->id
- 控制转移：if/else switch for while do/while ?: continue break
- 函数操作：参数传递(地址/值)、函数调用()、函数返回 return

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
- 在速度方面**Intel**做到了与**RISC**相当的性能，但功耗不低

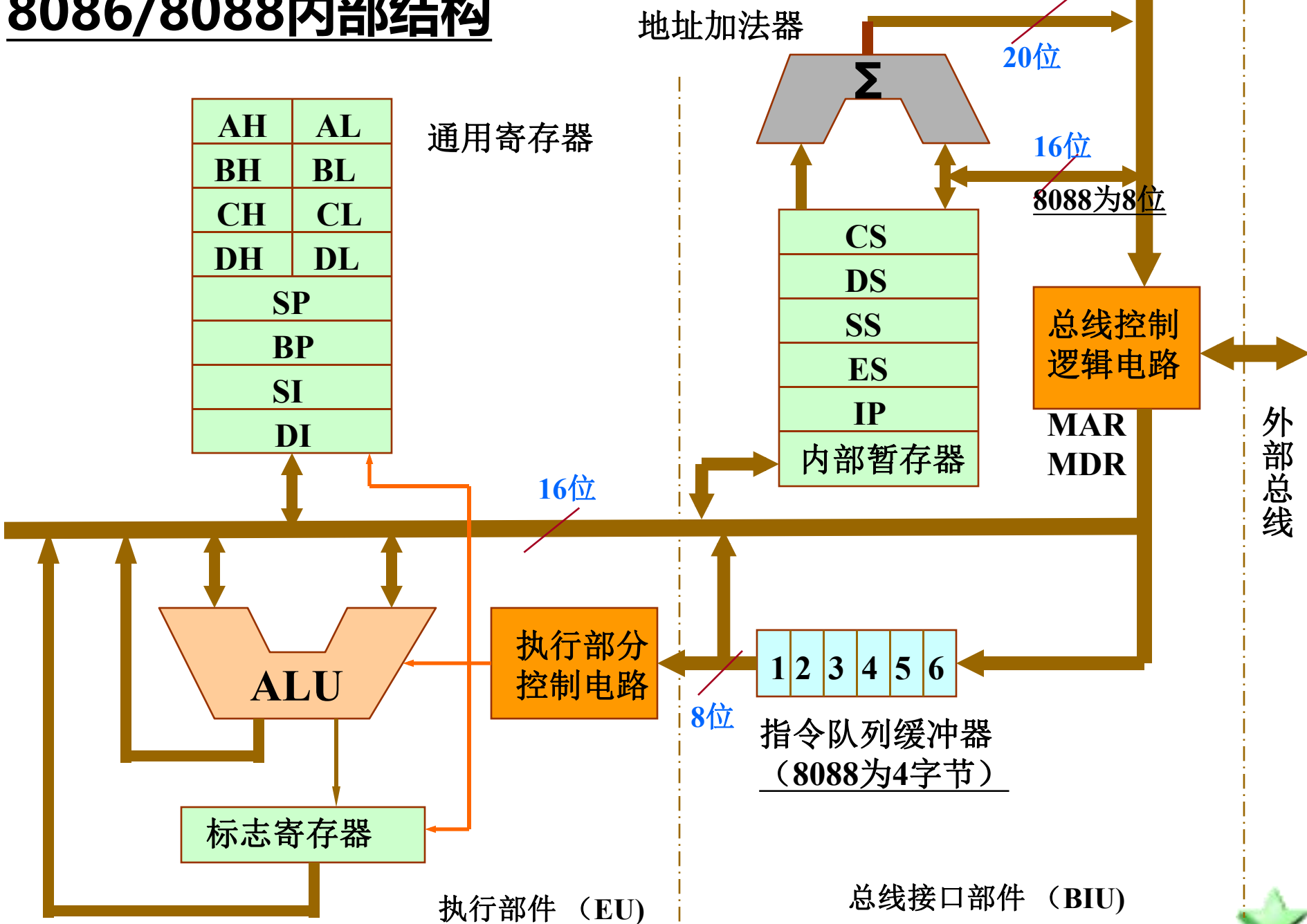
Intel x86 进化的里程碑

名字	时间	晶体管数量	主频
■ 8086	1978	29K	5-10
<ul style="list-style-type: none"> 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 			
■ 80286	1982	134K	20
<ul style="list-style-type: none"> IBM PC-AT & Windows、更多寻址模式 			
■ 386	1985	275K	16-33
<ul style="list-style-type: none"> 第一个32位intel处理器, 称为IA32 增加“平坦寻址”(flat addressing), 可运行Unix 			
■ Pentium 4E	2004	125M	2800-3800
<ul style="list-style-type: none"> 第一个64位Intel x86处理器, 称为 x86-64, 超线程(hyperthreading) 			
■ Core 2	2006	291M	1060-3500
<ul style="list-style-type: none"> 第一个多核处理器，不支持超线程（Core酷睿） 			
■ Core i7	2008	731M	1700-3900
<ul style="list-style-type: none"> 4核处理器、支持超线程 			

8086CPU的功能结构

- 8086/8内部有两个功能模块，完成一条指令的取指和执行功能
 - ❖ 模块之一：总线接口单元BIU，主要负责读取指令和操作数
 - ❖ 模块之二：执行单元EU，主要负责指令译码和执行

8086/8088内部结构



8086的寄存器组

- 对汇编语言程序员来说，8086内部结构就是可编程的寄存器组
 - ❖ 执行单元EU 8个通用寄存器
 - ❖ 1个指令指针寄存器
 - ❖ 1个标志寄存器
 - ❖ 4个段寄存器/64位增加了fs/gs也16位
- 后来的32/64位CPU(IA-32/x86-64)对通用寄存器进行了位扩展，64位增加R8-R15共8个寄存器

8086通用寄存器

- 8086的16位通用寄存器是：

AX	BX	CX	DX
SI	DI	BP	SP

- 其中前4个数据寄存器都还可以分成高8位和低8位两个独立的寄存器

- 8086的8位通用寄存器是：

AH	BH	CH	DH
AL	BL	CL	DL

- 对其中某8位的操作，并不影响另外对应8位的数据

数据寄存器

- **数据寄存器用来存放计算的结果和操作数，也可以存放地址**
- **每个寄存器又有它们各自的专用目的**
 - AX - - 累加器，使用频度最高，用于算术、逻辑运算以及与外设传送信息等；
 - BX - - 基址寄存器，常用做存放存储器地址；数组首地址。
 - CX - - 计数器，作为循环和串操作等指令中的隐含计数器；
 - DX - - 数据寄存器，常用来存放双字长数据的高16位，或存放外设端口地址。

变址寄存器

- **变址寄存器常用于存储器寻址时提供地址**
 - SI是源变址寄存器
 - DI是目的变址寄存器
- **串操作类指令中，SI和DI具有特别的功能(SI用作隐含的源串地址；DI用做隐含的目的串地址)**

指针寄存器

- 指针寄存器用于寻址内存堆栈内的数据
- SP为堆栈指针寄存器，指示栈顶的偏移地址
- SP不能再用于其他目的，具有专用目的
- BP为基址指针寄存器，表示数据在堆栈段中的基地址
- SP和BP寄存器与SS段寄存器联合使用以确定堆栈段中的存储单元地址

堆栈 (Stack)

- 堆栈是主存中一个特殊的区域，用于保存程序调用时的返回地址，用于依序保存程序调用时的值参，还用于程序中动态的申请或释放内存。
- 它采用**先进后出FILO** (First In Last Out) 或**后进先出LIFO** (Last In First Out) 的原则进行存取操作，而不是随机存取操作方式。
- 堆栈通常由处理器自动维持。在8086中，由堆栈段寄存器SS和堆栈指针寄存器SP共同指示。SP指向栈顶，BP指向值参的首地址（值得注意的是：栈是由栈底向下扩展）



指令指针IP

- **指令指针寄存器IP，指示代码段中指令的偏移地址**
- **它与代码段寄存器CS联用，确定下一条指令的物理地址**
- **计算机通过CS：IP寄存器来控制指令序列的执行流程**
- **IP寄存器是一个专用寄存器**

标志寄存器

- 标志（Flag）用于反映指令执行结果或控制指令执行形式
- 8086处理器的各种标志形成了一个16位的标志寄存器FLAGS（程序状态字PSW寄存器）

 程序设计需要利用标志的状态

15 12 11 10 9 8 7 6 5 4 3 2 1 0

OF

DF

IF

TF

SF

ZF

AF

PF

CF

标志的分类

- **状态标志** - - 用来记录程序运行结果的状态信息，许多指令的执行都将相应地设置它

CF ZF SF PF OF AF

- **控制标志** - - 可由程序根据需要用指令设置，用于控制处理器执行指令的方式

DF IF TF

进位标志CF (Carry Flag)

- 当运算结果的**最高有效位**有进位（加法）或借位（减法）时，进位标志置1，即 $CF = 1$ ；否则 $CF = 0$ 。
-

$3AH + 7CH = B6H$ ，没有进位： $CF = 0$

$AAH + 7CH = (1) 26H$ ，有进位： $CF = 1$

零标志ZF (Zero Flag)

- 若运算结果为0，则ZF = 1；
否则ZF = 0


● 注意：ZF为1表示的结果是0

3AH + 7CH = B6H，结果不是零：ZF = 0

84H + 7CH = (1) 00H，结果是零：ZF = 1

符号标志SF (Sign Flag)

- 运算结果最高位为1, 则SF = 1; 否则SF = 0


 有符号数据用最高有效位表示数据的符号
所以, 最高有效位就是符号标志的状态

3AH + 7CH = B6H, 最高位D₇ = 1: SF = 1

84H + 7CH = (1) 00H, 最高位D₇ = 0: SF = 0

奇偶标志PF (Parity Flag)

- 当运算结果最低字节中“1”的个数为零或偶数时, $PF = 1$; 否则 $PF = 0$

 PF标志仅反映最低8位中“1”的个数是偶或奇, 即使是进行16位字操作

$3AH + 7CH = B6H = 10110110B$

结果中有5个1, 是奇数: **$PF = 0$**

溢出标志OF (Overflow Flag)

- 若算术运算的结果有溢出，
则OF = 1； 否则 OF = 0

3AH + 7CH = B6H，产生溢出：OF = 1

AAH + 7CH = (1) 26H，没有溢出：OF = 0

溢出标志OF (Overflow Flag)

问题

什么是溢出？

溢出和进位有什么区别？

处理器怎么处理，程序员如何运用？

如何判断是否溢出？



什么是溢出

- 处理器内部以补码表示有符号数
- 8位表达的整数范围是： $+127 \sim -128$
- 16位表达的范围是： $+32767 \sim -32768$
- 如果运算结果超出这个范围，就产生了溢出
- 有溢出，说明有符号数的运算结果不正确

**$3AH + 7CH = B6H$ ，就是 $58 + 124 = 182$ ，
已经超出一 $128 \sim +127$ 范围，产生溢出，故 $OF = 1$ ；
另一方面，补码 $B6H$ 表达真值是 -74 ，
显然运算结果也不正确**



溢出和进位

- 溢出标志OF和进位标志CF是两个意义不同的标志
 - 进位标志表示**无符号数**运算结果是否超出范围，运算结果仍然正确；
 - 溢出标志表示**有符号数**运算结果是否超出范围，运算结果已经不正确。
-

溢出和进位的对比

例1：3AH + 7CH = B6H

无符号数运算： $58 + 124 = 182$
范围内，无进位

有符号数运算： $58 + 124 = 182$
范围外，有溢出

例2：AAH + 7CH = (1) 26H

无符号数运算： $170 + 124 = 294$
范围外，有进位

有符号数运算： $-86 + 124 = 28$
范围内，无溢出



如何运用溢出和进位

- 处理器对两个操作数进行运算时，按照无符号数求得结果，并相应设置进位标志CF；同时，根据是否超出有符号数的范围设置溢出标志OF。
- 应该利用哪个标志，则由程序员来决定。也就是说，如果将参加运算的操作数认为是无符号数，就应该关心进位；认为是有符号数，则要注意是否溢出。



溢出的判断


- 判断运算结果是否溢出有一个简单的规则：
- 只有当两个相同符号数相加（包括不同符号数相减），而运算结果的符号与原数据符号相反时，产生溢出；因为，此时的运算结果显然不正确
- 其他情况下，则不会产生溢出



- 有符号无符号都不溢出： $4+11=?$ **15**
- 无符号溢出： $7+251=?$ **2**
- 有进位 $7+(-5)=?$ **2** $-1+(-2)=?$ **-3**
- 有符号溢出： $9+124=?$ **-123**
- 都溢出： $135+245=?$ **124**
 $-121+(-11)=?$ **124**
- 溢出吗？： $-121-7=?$ **不溢出**

辅助进位标志AF (Auxiliary Carry Flag)

- 运算时 D_3 位（低半字节最高位）有进位或借位时， $AF = 1$ ；否则 $AF = 0$ 。

 这个标志主要由处理器内部使用，用于十进制算术运算调整指令中，用户一般不必关心

$3AH + 7CH = B6H$, D_3 有进位: $AF = 1$

段寄存器

- **8086有4个16位段寄存器**
 - CS（代码段）指明代码段的起始地址
 - SS（堆栈段）指明堆栈段的起始地址
 - DS（数据段）指明数据段的起始地址
 - ES（附加段）指明附加段的起始地址
- **每个段寄存器用来确定一个逻辑段的起始地址，每种逻辑段均有各自的用途**
- **32/64位CPU采用段选择子—数组下标/段描述符表—数组（含段地址、范围、权限等）的形式来确定段地址。**

8086的指令系统

8086 / 8088的指令系统包含了六种类型，其中数据传送指令14条，算术运算指令20条，逻辑运算指令13条，串操作指令10条，控制转移指令28条，处理器控制指令12条。

1)数据传送指令(14条)

MOV: 传送;

PUSH, POP: 堆栈操作;

XCHG: 交换;

IN、OUT: 输入、输出;

XLAT: 转换;

LEA、LDS、LES: 地址传送;

PUSHF、POPF、LAHF、SAHF: 标志传送。

2) 算术运算指令(20条)

ADD、ADC、AAA、DAA: 加法;

INC: 加“1”;

SUB、SBB、AAS、DAS: 减法;

DEC: 减“1”;

CMP: 比较;

NEG: 求补;

MUL、IMUL、AAM: 乘法;

DIV、IDIV、AAD: 除法;

CBW, CWD: 符号扩展。

MOVSX, MOVZX符号扩展

3) 逻辑运算指令(13条)——一位操作指令

NOT: 求反;

AND: 逻辑乘;

OR: 逻辑加;

XOR: 异或;

TEST: 测试位;

SHL、SHR、SAL、SAR: 左/右移位;

ROL、ROR、RCL、RCR: 左/右循环移位。

4)字符串操作指令(10条)

MOVS、MOVSB / MOVSW: 传送串;

CMPS、CMPSB / CMPSW: 串比较;

SCAS、SCASB / SCASW: 串扫描;

LODS、LODSB / LODSW: 取字符串;

STOS、STOSB / STOSW: 存字符串。

(REP、REPZ, REPNZ: 重复前缀)。

5)控制转移指令(28条)

CALL: 子程序调用;

RET: 子程序返回;

JMP: 无条件转移;

JZ、JNZ、JC、JNC、JO、J

NO、JS、JNS、JP、JNP;

JA、JAE、JB、JBE、JG、

JGE、JL、JLE: 条件转移;

LOOP: 循环;

LOOPNE、LOOPE: 条件循环;

JCXZ: 寄存器CX=0转移;

INT、INTO: 中断;

IRET: 中断返回。

6) 处理机控制指令(12条)

CLC: 清除CF标志;

CMC: 进位位CF求反;

STC: 置CF标志;

STD: 置DF标志;

CLD: 清除DF标志;

STI: 置IF标志;

CLI: 清除IF标志;

HLT: 处理机暂停;

WAIT: 等待状态;

ESC: 将数据传送给协处理器(提供到地址 / 数据线上);

LOCK: 保证总线的控制;

NOP: 无操作。

操作数寻址方式

- 指令由**操作码**和**操作数**组成
- 操作数是一个常数值—立即数，也可以在CPU、内存、IO端口（IN/OUT指令）中
- 操作数寻址方式有：
 - 立即数寻址：MOV EAX,12345678H
 - 寄存器寻址：MOV EAX,EBX
 - 存储器寻址：MOV EAX,DS:[20000H]
- 存储器寻址 **D(Rb,Ri,S)** 段寄存器可用默认，省略
 - 段址：[基址Rb+变址Ri*比例因子S+偏移D]
 - 这三部分可以任意组合。32位以上CPU才有比例因子(1/2/4/8)。
 - 16位CPU：基址BX/BP,变址SI/DI。其他CPU变址不用ESP等

存储器寻址

■ 其组合与我们高级语言编程时的各种类型与结构的全局变量、局部变量、参数的访问有关系。

- 直接寻址 `MOV AX,[1000H]`
- 寄存器间接寻址 `MOV AX,[BX]`
- 寄存器相对寻址 `MOV AX,[BX+1]`
- 基址变址寻址 `MOV AX,[BX+SI]`
- 相对基址变址寻址 `MOV AX,[BX+SI+100]`
- 比例寻址：32/64位CPU，变址不用ESP/RSP
 - [变址*比例因子] `MOV EAX,[EDI*4]`
 - [变址*比例因子+偏移] `MOV EAX,[EDI*4+100]`
 - [基址+变址*比例因子] `MOV EAX,[EBX+EDI*4]`
 - [基址+变址*比例因子+偏移] `MOV EAX,[EBX+EDI*4+8]`

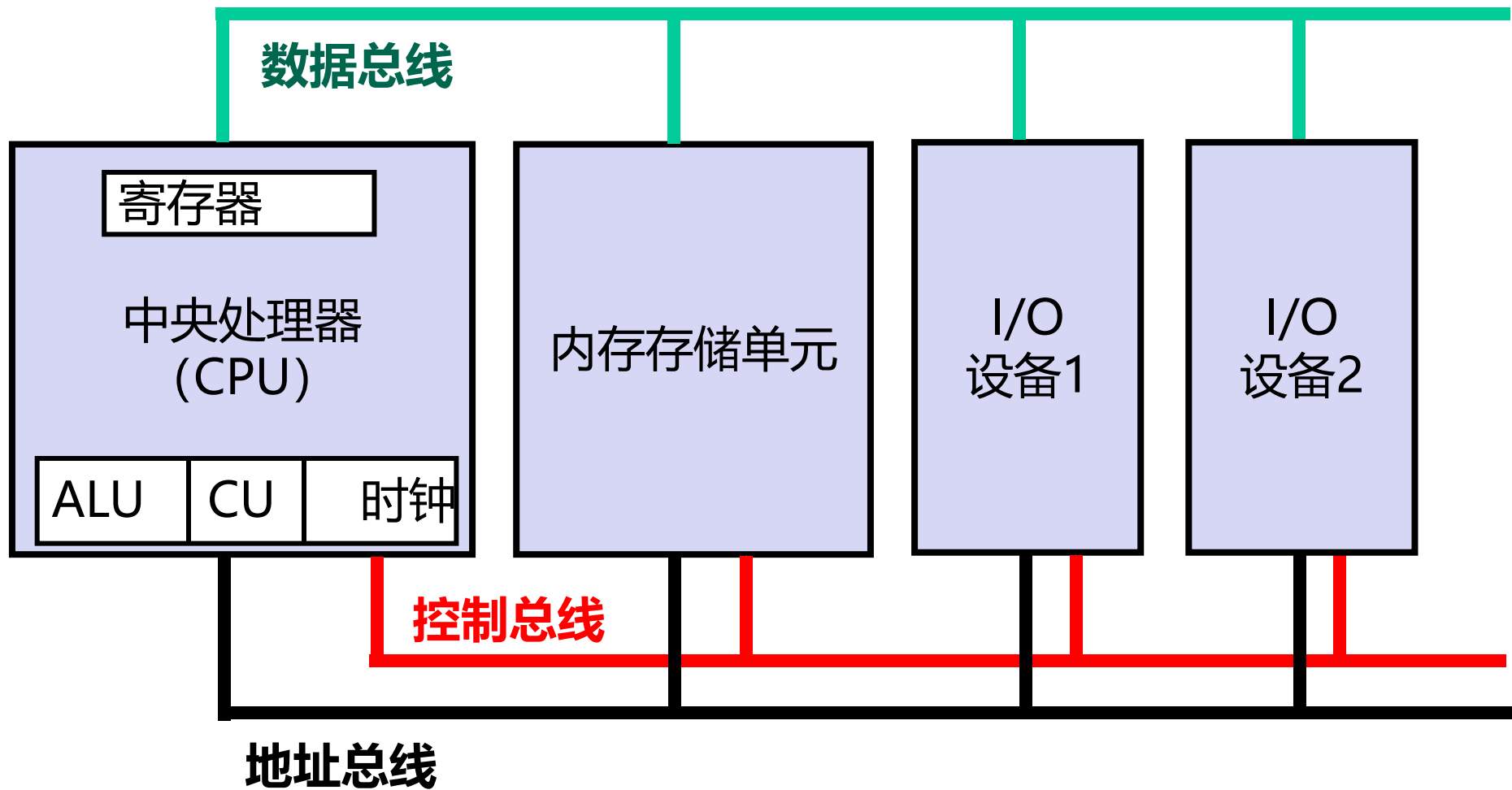
机器级程序设计I: 基础

- 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

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

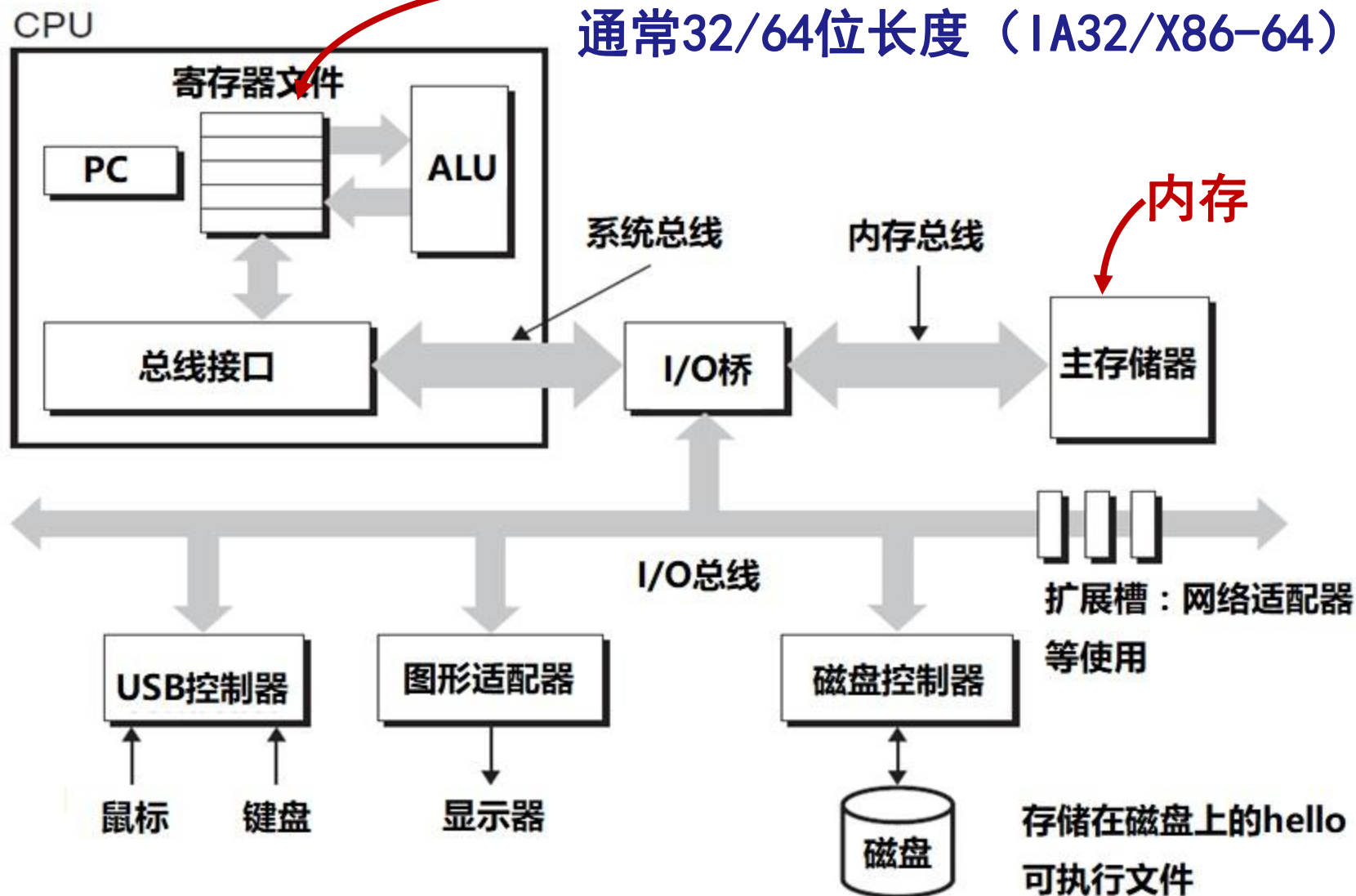
1 微机的基本结构



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）



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：多了FS、GS
- ✓ 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：支持OS与调试等的寄存器

■ 2.3 浮点单元

2、 IA32的寄存器

■ 2.1 基本寄存器

EAX
EBX
ECX
EDX
EBP
ESP
ESI
EDI
EFLAGS
EIP

16位段寄存器

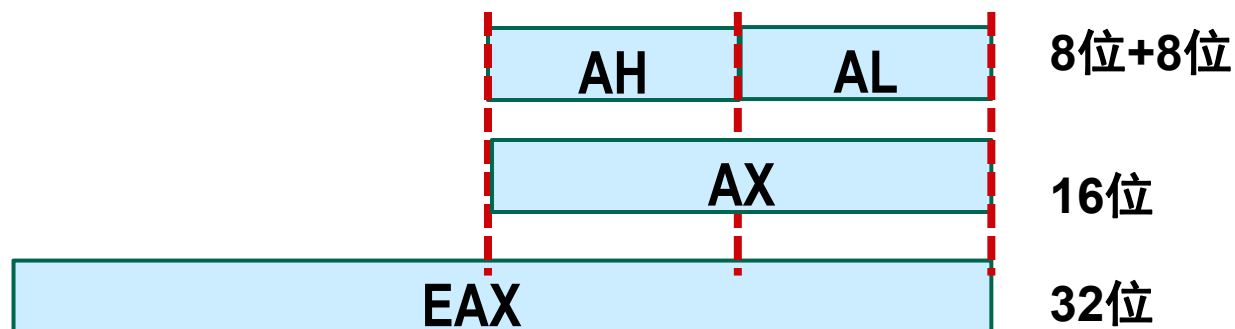
CS	ES
SS	FS
DS	GS

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

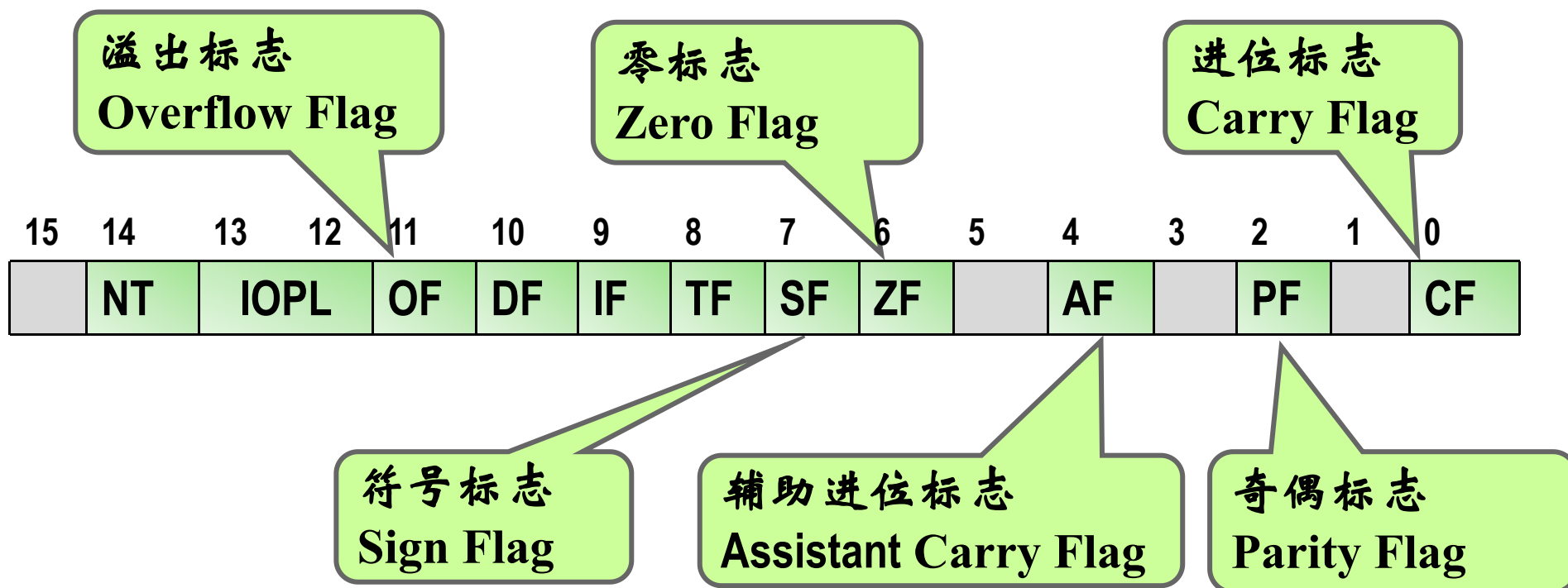
■ 2.1.4 EFLAGS寄存器(标志寄存器、 条件码寄存器)

- 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（仅了解）

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（仅了解）

适合于高速浮点运算，从Intel 486开始集成到主处理器芯片中。

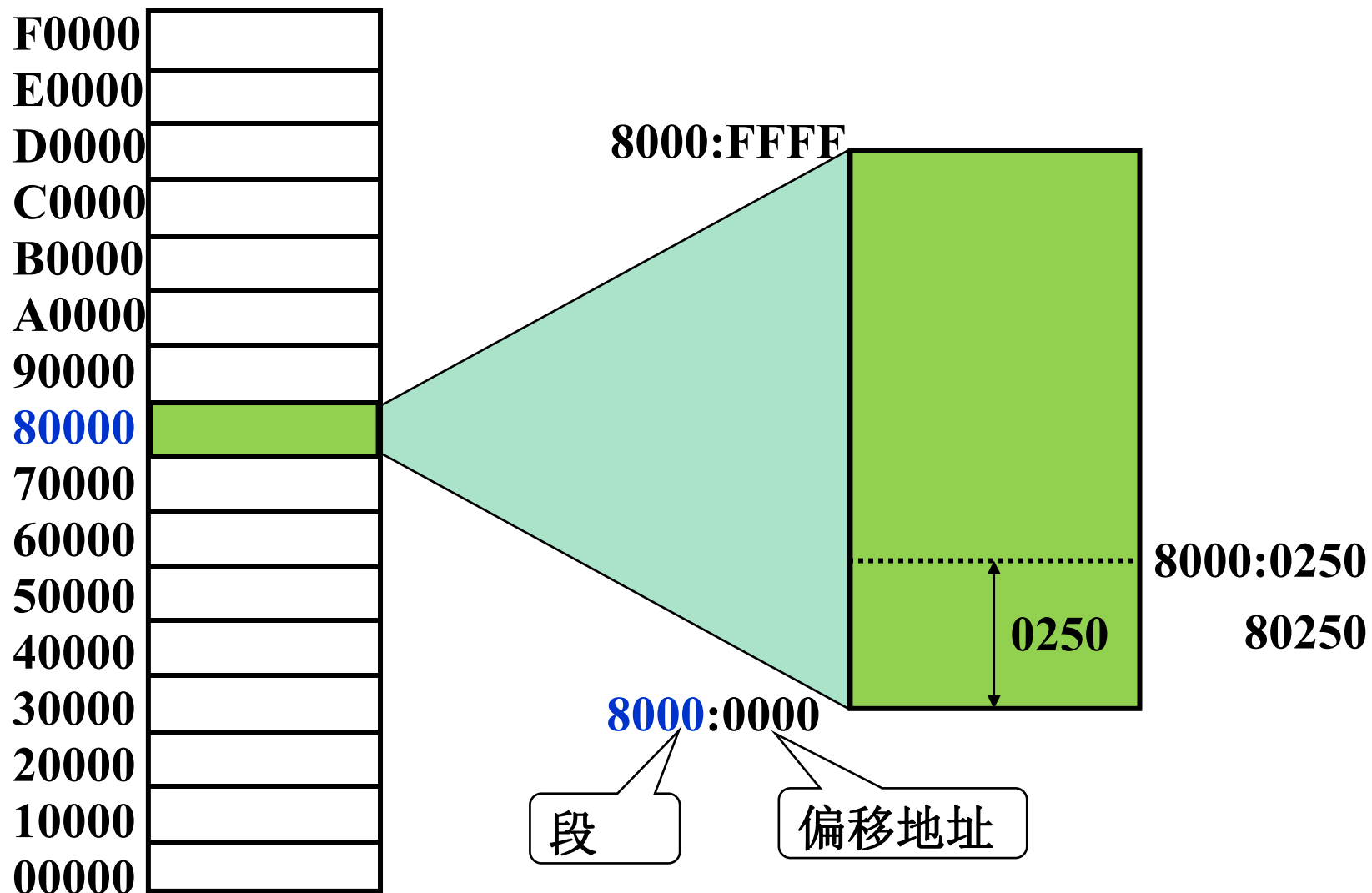
- 8个80位的浮点数据寄存器： st(0) — st(7)
- 2个48位的指针寄存器
- 3个16位的控制寄存器

3、IA32的内存管理（仅了解）

■ 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~FFFFFF)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位的代码段基地址
 - DS用于存放16位的数据段基地址
 - SS用于存放16位的堆栈段基地址

段-偏移地址



■ 20位线性地址的计算

例：

08F1: 0100

$$\rightarrow 08F1H * 10H + 0100H = 09010H$$

8000:0250

$$\rightarrow 8000H * 10H + 0250H = 80250H$$

■ 3.2 保护模式

- 32位地址总线寻址，每个程序可寻址4GB内存：
0~FFFFFFFF
- **段寄存器(CS、DS、SS、ES、FS和GS)指向段描述符表**，操作系统使用段描述符表定位程序使用的段的位置。
 - ✓ CS存放**代码段描述符表**的地址
 - ✓ DS存放**数据段描述符表**的地址
 - ✓ SS存放**堆栈段描述符表**的地址

■ 3.2 保护模式...

■ 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。

■ 多段模式 (Multi-Segment)

■ 分页模式 (Paging)

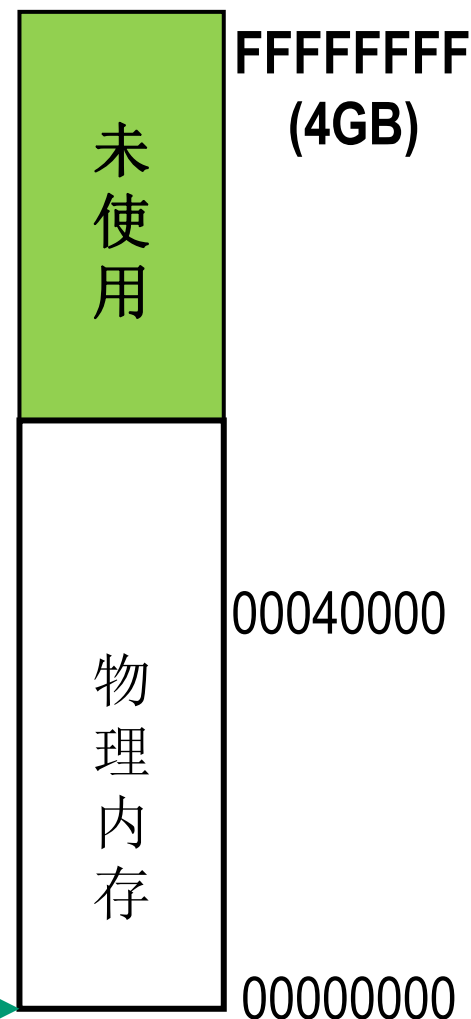
- 将一个段分割成称为页 (Pages) 的4KB的内存块

■ 3.2.1 平坦分段模式

全局段描述符表(GDT)中的段描述符

基址 界限 访问类型

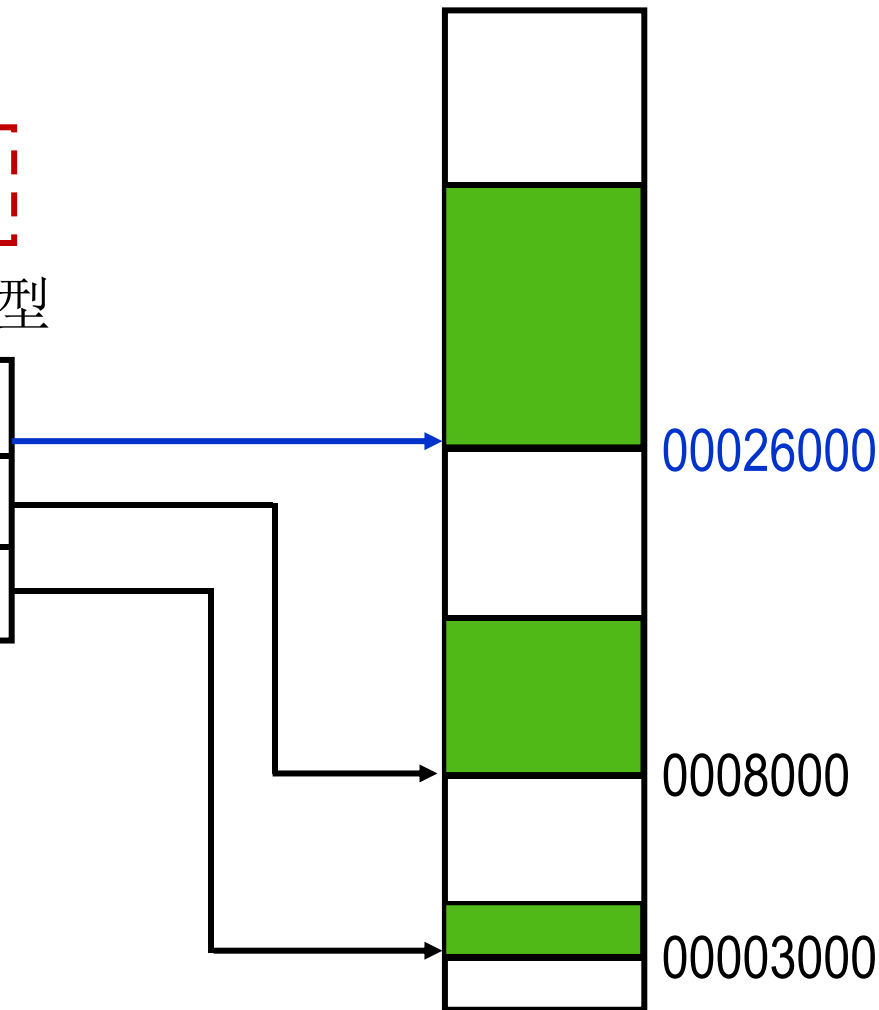
00000000	0040
----------	------	-------



■ 3.2.2 多段模式

局部描述符表 (LDT)

基址	界限	访问类型
00026000	0010
00008000	000A
00003000	0002



■ 3.2.3 分页模式

- 将内存分割成4KB大小的页面，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、指令执行周期

- 指令执行周期：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、指令执行周期

■ 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**；
 - **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，将运算结果送到输出操作数中(有名寄存器/内存)，并更新反映处理器状态的状态标志。
 - **存储输出操作数：** 如果输出操作数在存储器中，控制单元就执行一个写操作将数据存储在内存。
- ### ■ 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 CS:FFFF IP:0000 其他全0

(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：

$$IP/EIP/RIP = IP/EIP/RIP + \text{所读取指令的长度}$$

(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（这种情况称为**并行**），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，造成一种宏观上同时运行的效果（称为**并发**）。

5、程序是如何运行的

■ 多任务的实现

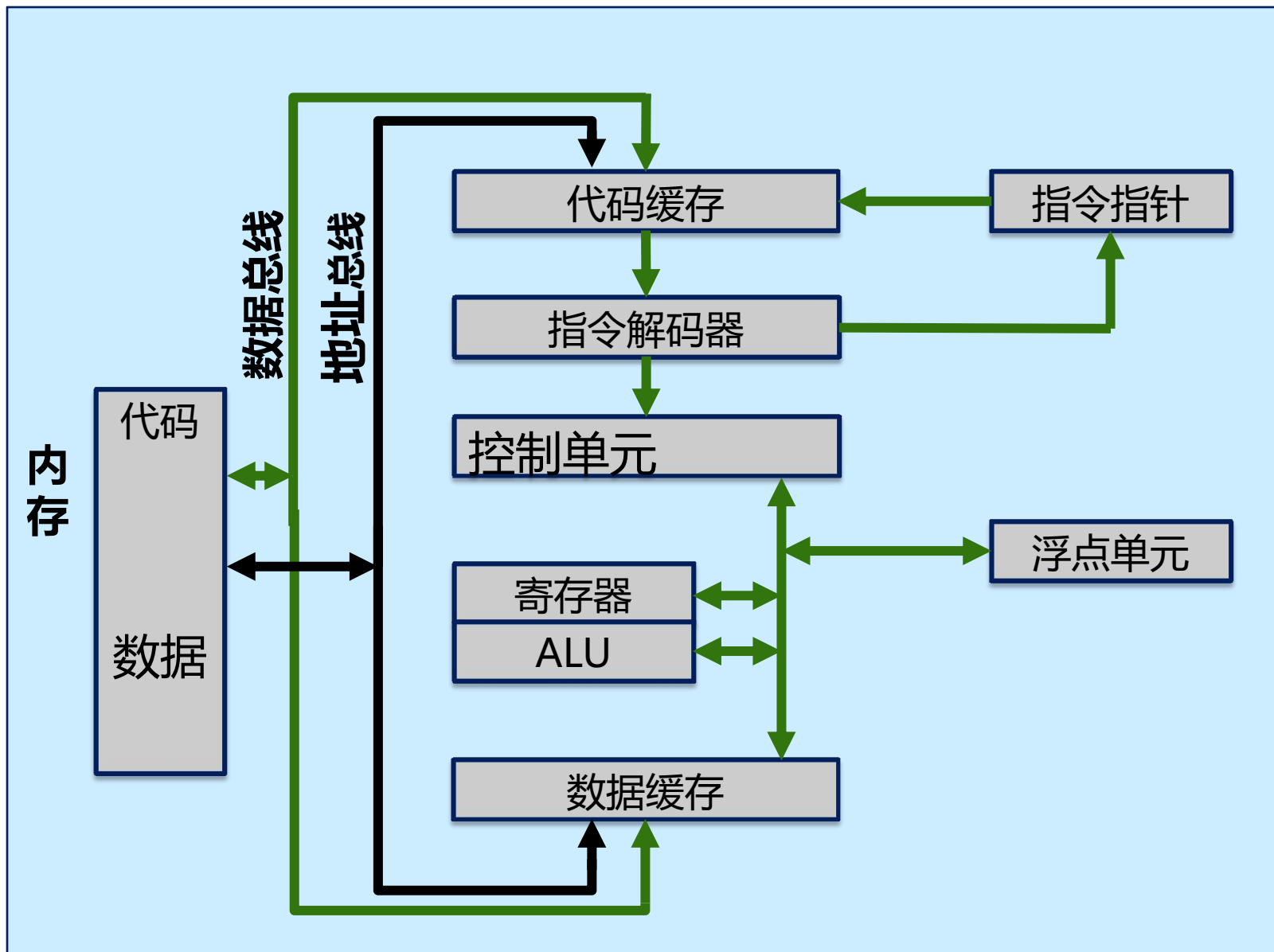
如何实现处理器在各个进程之间共享？

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（仅作了解）

■ 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~FFFFFFH:系统ROM，BIOS中断服务例程。



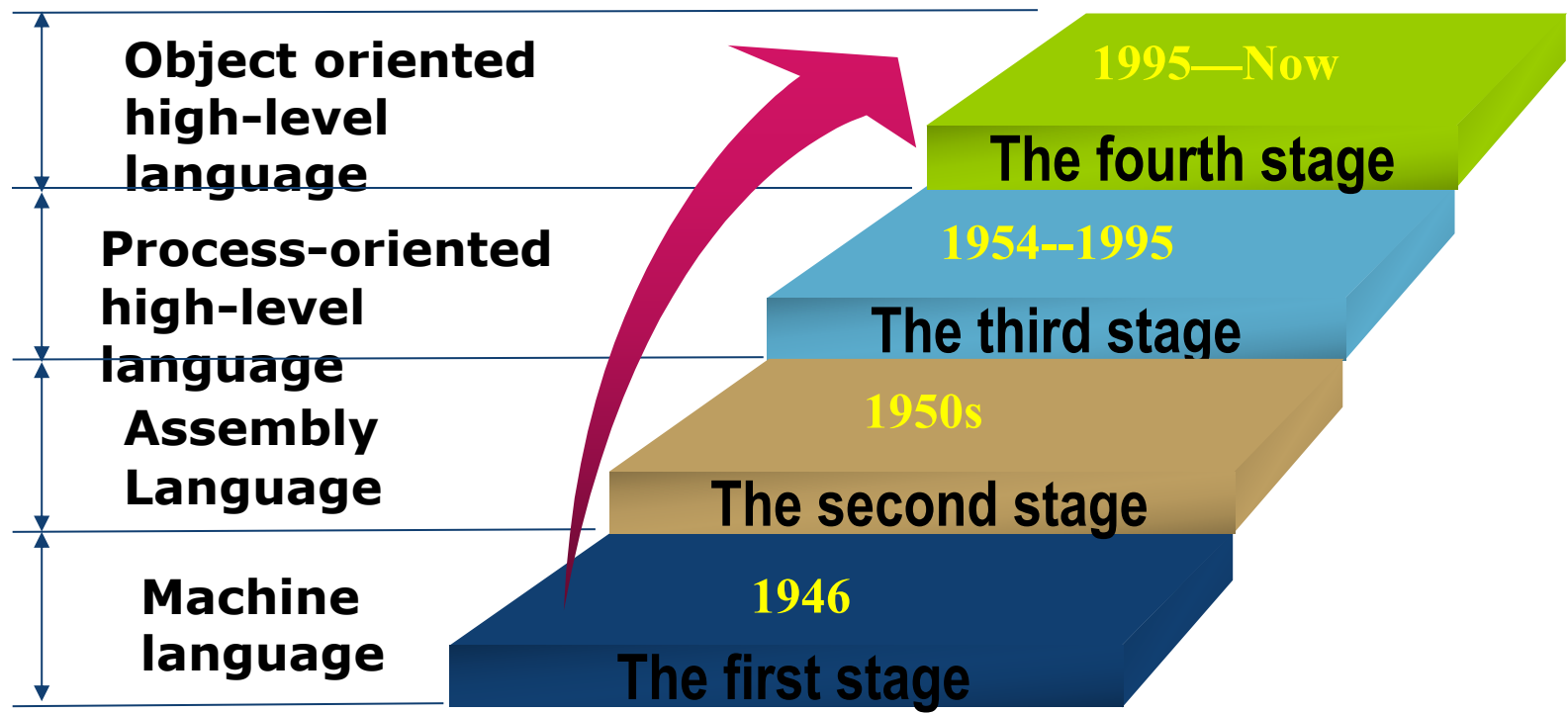
简化的奔腾CPU结构图

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（仅作了解）
- Linux汇编程序

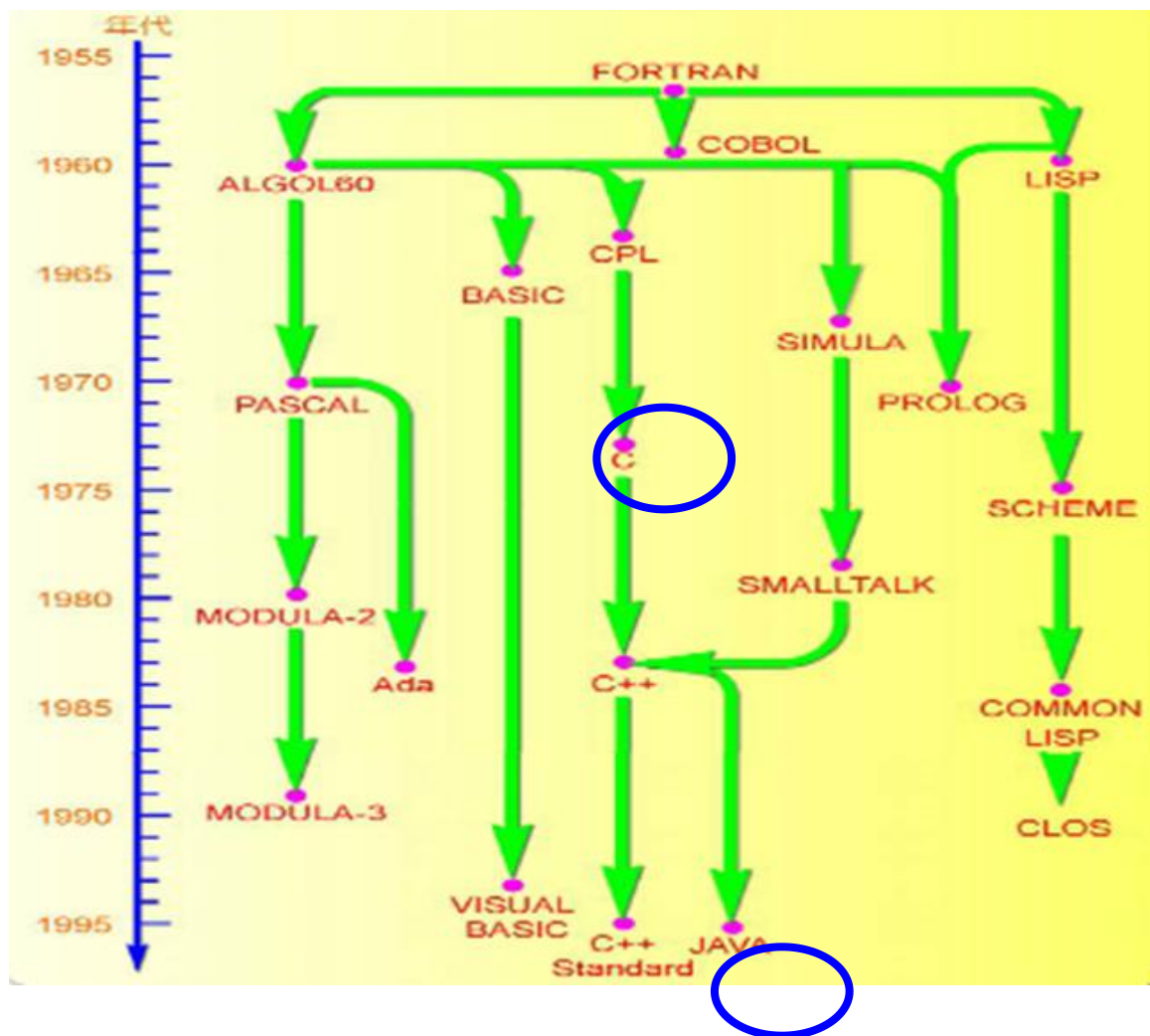
汇编语言简介

- 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：

```
10110000000000000000000011
0000010100000000000110000
00101101000000000000000101
```

假如将程序错写成以下这样，请找出错误：

```
10110000000000000000000011
0000010100000000000110000
00010110100000000000000101
```

(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——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：

```
101100000000000000000000011
0000010100000000000110000
00101101000000000000000101
```

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

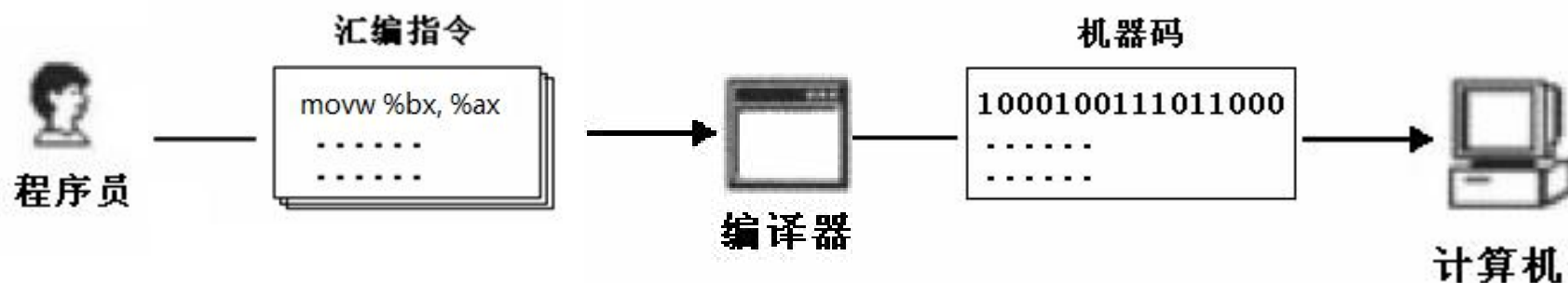
■ 除汇编指令，汇编语言还包括：

- 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
```

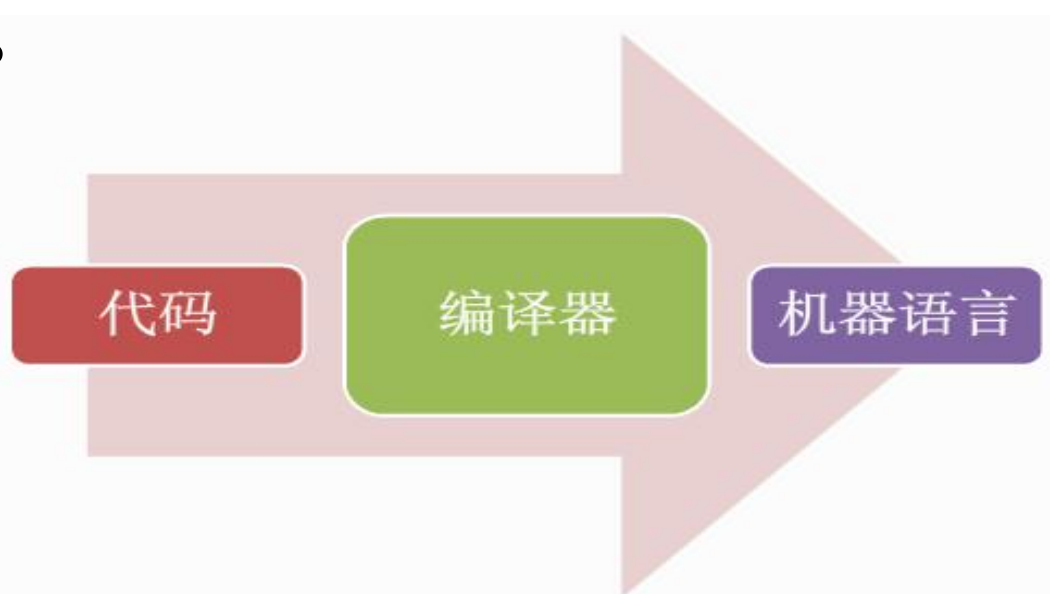
(4) 高级语言到机器语言的转换方法

■ 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中编译运行，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。
- 后继课程的学习

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**

Linux汇编程序——两种格式的语法对比

■ 两种汇编格式: **AT&T 汇编**、Intel汇编

■ 1、寄存器前缀%

AT&T: %eax

Intel: eax

■ 2、源/目的操作数顺序

AT&T: movl %eax,%ebx

Intel: mov ebx,eax

■ 3、常数/立即数的格式 \$

AT&T: movl \$ _value, %ebx **#把变量_value的地址放入ebx**
 movl \$0xd00d, %ebx

Intel: mov eax, offset _value
 mov ebx,0d00dh

■ 4、操作数长度标识:b-1字节, w-2, L-4 ,q-8

AT&T: mov**w** var_x, %bx Intel: mov bx, **word ptr** var_x

Linux汇编程序——两种格式的语法对比

■5、寻址方式

AT&T: $D(Rb, Ri, S)$

Intel: $[Rb + Ri * S + D]$ 或 $D[Rb][Ri * 4]$

Linux工作于保护模式下, 使用32位线性地址, 计算地址时不用考虑 segment:offset的问题, 上式地址为: $D + Rb + Ri * S$

(1) 直接寻址

AT&T: `movl $0xd00d, var`

var是一个全局变量

注意: $\$var$ 表示变量地址引用, var 表示变量值引用

Intel: `mov var, 0d00dh`

;等价于 `mov [var], 0xd00d`

注意: offset var表示变量地址, var表示值。

(2) 寄存器间接寻址/寄存器相对寻址

AT&T :

`movl (%ebx), %eax`

`movl 3(%ebx), %eax`

Intel :

`mov eax, [ebx]`

`mov eax, [ebx+3]`

`mov eax, 3[ebx]`

Linux汇编程序——两种格式的语法对比

(3) 变址寻址/基址变址寻址/相对基址变址寻址/带比例因子的**

AT&T: `movl %ecx, var(,%eax)`
`movl %ecx, array(,%eax,4)`
`movl %ecx, array(%ebx,%eax,8)`

Intel: `mov [eax + var], ecx`
`mov [eax*4 + array], ecx`
`mov [ebx + eax*8 + array], ecx`

■ C中嵌入式汇编

```
asm( "pushl %eax\n\t"
      "movl $0,%eax\n\t"
      "popl %eax");
```

```
asm("movl %eax,%ebx");
asm("xorl %ebx,%edx");
asm("movl $0,_booga);
```

```
asm{
    pushl eax;
    mov eax,0;
}
asm mov ebx,eax;
asm xor  edx,ebx;
asm mov _booga,0;
```

Linux汇编程序：AT&T 格式程序

```

#hello.s
.data
    msg : .string "Hello, world! ----- AT&T ASM\r\n "
    len = . - msg
.text
.global _start
_start:
    movl $len, %edx
    movl $msg, %ecx
    movl $1, %ebx
    movl $4, %eax
    int $0x80
    # =====退出程序
    movl $0,%ebx
    movl $1,%eax
    int $0x80

```

数据段声明
要输出的字符串
字串长度
代码段声明
指定入口函数
在屏幕上显示一个字符串
参数三：字符串长度
参数二：要显示的字符串
参数一：文件描述符(stdout)
系统调用号(sys_write)
调用内核功能

参数一：退出代码
系统调用号(sys_exit)
调用内核功能

Linux汇编程序： Intel格式程序（非微软）

```
; 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 ; 字符串长度
```

```
.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 80h ; 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 80h ; 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

经典例题

1. C语言程序中的整数常量、整数常量表达式是在（ ）阶段变成2进制补码的。

(A) 预处理 (B) 编译 (C) 连接 (D) 执行

2. C语言程序中的常量表达式的计算是由 编译器 完成的。

本章主要以选择、填空或分析题考查

Enjoy!