学习内容

- 通用寄存器组 ——不再细讲
- 标志寄存器
- 指令指示器
- 段寄存器

2.1 Intel微处理器的发展史

- 成立于1968年
- INTegrated ELectronics（集成电子）的缩写
- 先后推出的中央处理器：Intel4004、Intel8008、Intel8080/8085、8086/8088、80186、80286、i386、i486
- Pentium（奔腾）及其系列、Xeon（至强）及其系列、Core（酷睿）及其系列

——IA-32,X86-32架构

- 它属于复杂指令集架构

  Complex Instruction Set Computing，CISC

特点：数据线和指令线分时复用（只能通过一辆车）。存储器操作指令多、汇编程序相对简单、指令结束后响应中断、CPU电路丰富、面积大功耗大

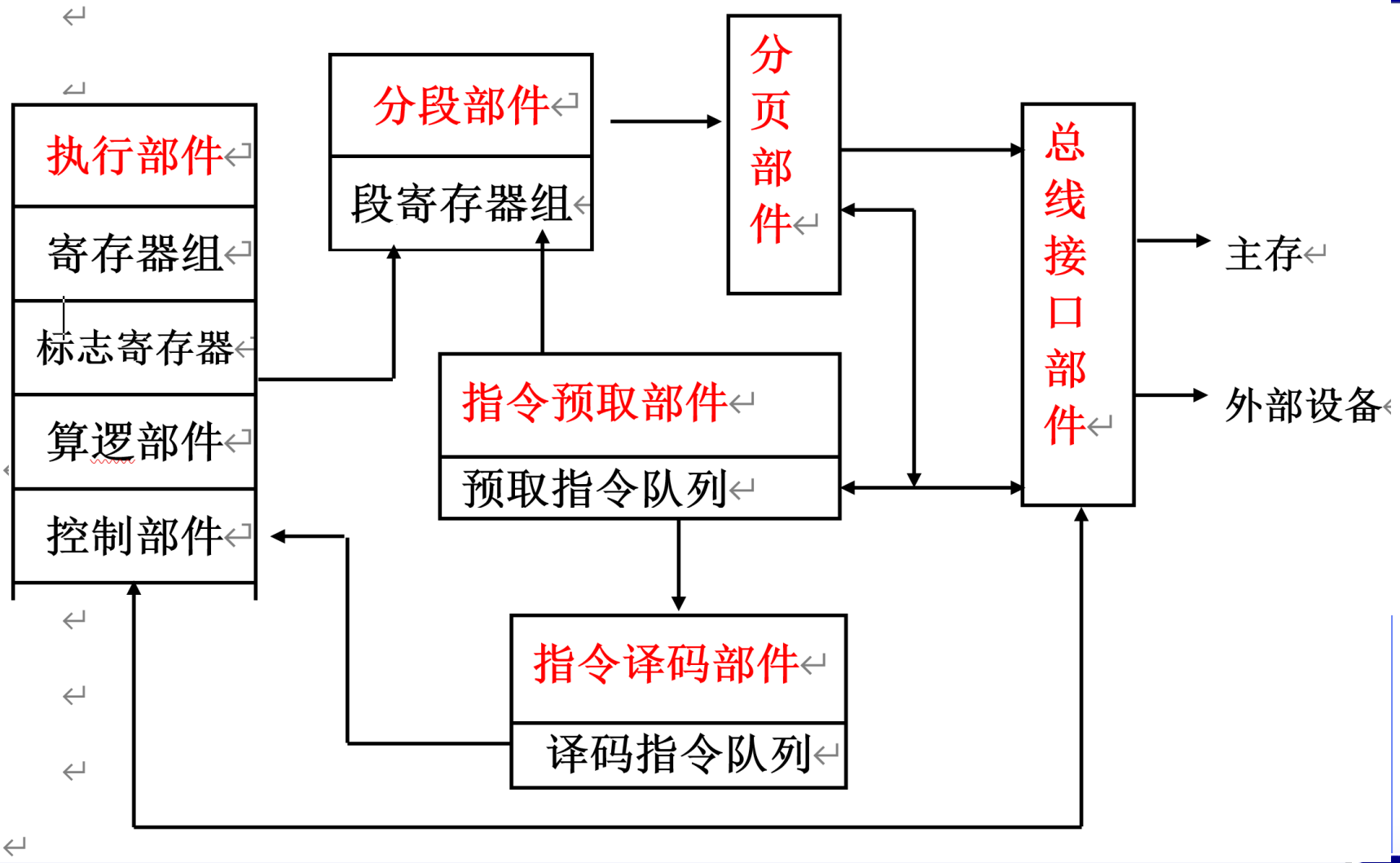
2.1 Intel微处理器的发展史

- IA-64架构是一种全新处理器架构，与x86不能兼容。
- 揣测：AMD公司抢先将32位x86指令集扩展到64位，推出了AMD64。为了竞争，Intel做出了一次不太成功的尝试。
- 为了解决与IA-32兼容的问题，Intel回到了与x86兼容的道路上，采用了x86-64结构（也称为Intel64，x64）。
- 扩充内容：物理内存、指令集、CPU寄存器结构、应用程序的虚拟内存。
- x86从32位到64位的变化，并没有像从16位到32位的变化那样，在系统软件层面带来了革命性的变化（例如页式地址管理、多任务的引入等等）。
- 操作系统仍然使用以前的各种机制来对硬件进行管理。

ARM系列处理器

- ARM 是一家英国电子公司的名字，全名是 Advanced RISC Machine，这家企业设计了大量高性能、廉 价、耗能低的 RISC（精简指令集）处理器，ARM 公司只设计芯片而不生产，它将技术授权给世界上许多公司和厂商。目前采用 ARM 技术知识产权内核的微处理器，即通常所说的 ARM 微处理器，所以 ARM 也是对一类微处理器的通称。——华为鲲鹏处理器兼容ARM指令；
- 差异：将程序指令存储和数据存储分开，中央处理器首先到程序指令存储器中读取程序指令。 解码后到数据地址，再到相应的数据存储器读取数据，然后执行指令；
- RISC 精简指令集：数据线和指令线分离。对存储器操作有限 汇编程序占空间大、在适当地方响应中断、 CPU电路较少，体积小、功耗低。

2.2 Intel x86微处理器结构



x86微处理器结构

2.2 Intel x86微处理器结构

总线接口部件：接受所有的总线操作请求，并按优先权进行选择，最大限度地利用本身的资源为这些请求服务

总线：指传递信息的一组公用导线。

系统总线（System Bus）：

从微处理器引出的若干信号线。

CPU通过它们与内存和外设交换信息。

地址总线：Address Bus （单向总线）

数据总线：Data Bus

控制总线：Control Bus

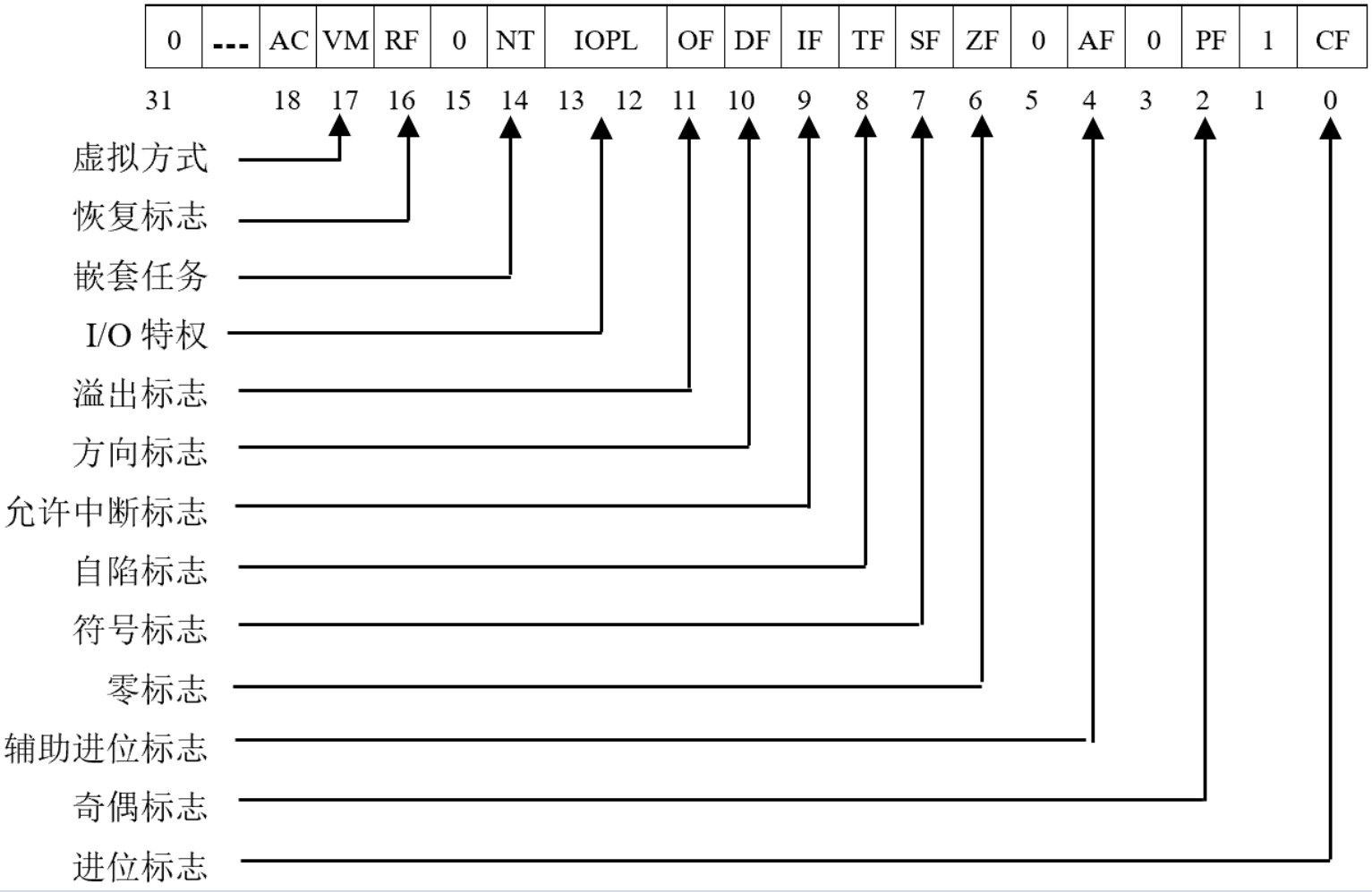
2.4 标志寄存器

- 保存一条指令执行之后， CPU所处状态的信息及运算结果的特征。
- 32位CPU中的标志寄存器是32位，称**EFLAGS**；

历史故事：

- 16位CPU中的标志寄存器是16位，称**FLAGS**；
- 32位的EFLAGS包含了16位FLAGS的全部标志位且向下兼容。

2.4 标志寄存器


0 --- AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
31 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
虚拟方式
恢复标志
嵌套任务
I/O 特权
溢出标志
方向标志
允许中断标志
自陷标志
符号标志
零标志
辅助进位标志
奇偶标志
进位标志

2.4 标志寄存器

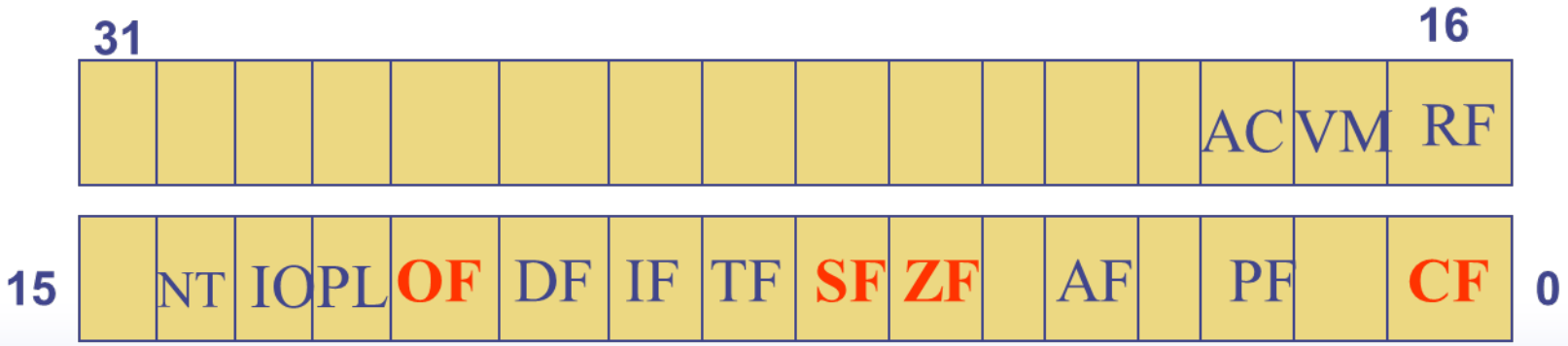
2.4.1 条件标志位

Sign Flag 符号标志

Zero Flag 零标志

Overflow Flag 溢出标志

Carry Flag 进位标志



2.4 标志寄存器

1、符号标志 SF (Sign Flag)

运算结果的最高二进制位为1，则SF=1，否则SF=0

2、零标志 ZF (Zero Flag)

运算结果为0，则 ZF＝1，否则 ZF＝0

3、进位标志 CF (Carry Flag)

运算时从最高位向前产生了进位（或借位），则CF=1；否则 CF=0。

	1	1	1	1	1	1	1	1
+	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	1

符号位 SF =0

进位 CF=1

```
MOV   AH,  0FFH
ADD   AH,   2
```

(AH) = 1

2.4 标志寄存器

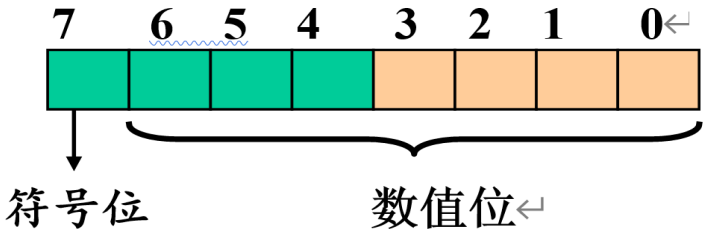
数值数据在计算机内的表示形式

1. 数值数据的表示

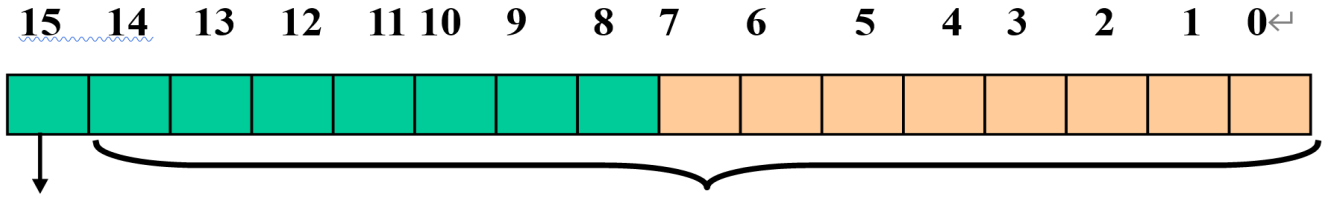
(1) 有符号数

有符号数：

假设机器字长为8位：



假设机器字长为16位：



符号位 ＝0　　正数　　数值位

　　＝1　　负数

2.4 标志寄存器

(2)有符号数的补码表示

——注意汇编往往是针对指令相关的数据类型长度的

- 补码表示的数

补码的计算：所有正数的补码为其本身；所有负数的补码由其相反数连符号位一起按位取反加 1。

例：设 n=8，[0110010]补=00110010

[-0110010]补=11001110

讨论：设 n=8，一个数的补码是 10000000B，这个数是正数还是负数？表示哪个数的原码？

有符号数表示范围？
-128—127

一个二进制数的补码表示的最高位，向左扩展若干位，得到的仍然是该数的补码

(3) 无符号数

有符号数和无符号数的表数范围：

讨论：当 n=16 时，计算机内表示的数值数据 9A37H 到底是有符号数还是无符号数？ 计算机怎样看？

问题：为什么要引入无符号数？

2.4 标志寄存器

有符号和无符号整数的表示形式

```
#include <stdio.h>

void main()

{
      short   x;
      unsigned short  y;
      x= -1;
      y= -1;
      printf("%d   %d\n", x,  y);

}
```

结果是：
-1   65535

在汇编语言中也有类似于unsigned的约定。

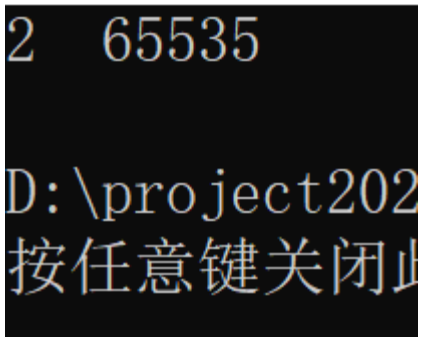
工程： c_有符号和符号的比较

2.4 标志寄存器

有符号和无符号整数的表示形式

```
#include <stdio.h>
void main()
{
        short   x;
          unsigned short  y;
        x=-1;    y=-1;
          if (x>3)
              printf("1 %d\n",x);
        if (y>3)
              printf("2 %d\n", y);
}
```

结果？



工程： c_有符号和符号的比较

(4) 数据在程序中的书写与转换

例： 把下面几种形式书写的数据转换成字节型的机内部存储形式。

0E3H, －29, －11101B

书写要注意的问题：16 进制的 H，二进制的 B 不要漏掉。

汇编程序转换，内部形式完全相同，都是 11100011B——全部转换为二进制

计算机的转换原则：所有的数先全部作为无符号数处理。若为负数，则先将数值作为无符号数处理后，再连同符号位求补。

. 在计算机内的存储形式相同的数据可以有多种书写形式。

(5) 数值数据的运算掌握要点：

. 计算机在进行算术逻辑运算时，总是把参与运算的、用补码表示的操作数作为无符号数处理；

溢出标志设置的直观理解

```
#include <stdio.h>
int main()
{
    short   x, y, z;
    x = 32766;      // 7FFEH
    y = 3;
    z = x + y;
    printf("%d %d %d \n", x, y, z);
    return 0;
}
```

```
0111 1111 1111 1110
                  11
1000 0000 0000 0001    OF=1
```

溢出标志设置的直观理解

溢出：两个正数相加，结果为负。
　　两个负数相加，结果为正。

Question ： 对于减法运算呢？

A：若被减数与减数的最高位同时为1，或为0，则OF一定为0，即决不会溢出。

若被减数与减数的最高位不同，差的最高位与被减数的最高位不同，则OF=1;否则OF=0.

已知 8 位二进制数X1,X2的值，求[X1]补＋[X2]补，并指出执行该运算后,SF,ZF,OF,CF各是多少？

X1 = + 110 0101B　　X2= - 101 1101B

[X1]补 = 0110 0101 B

[X2]补 = 1010 0011 B

\+ 1 0000 1000 B

-x2 = 0101 1101 B

求反 1010 0010 B

加1 1010 0011 B

SF = 0

ZF = 0

OF = 0

CF = 1

101 + (- 93) = 8

实验验证

2.4 标志寄存器


```
.686P
.model flat,stdcall
ExitProcess proto  :dword
.code
start:
    mov al,1100101B
    mov ah, -1011101B
    add ah, al  已用时间 <= 1ms
    invoke ExitProcess, 0
end start
```

寄存器

```
EAX = 012FA365 EBX = 0101E000
  ECX = 003D1000 EDX = 003D1000
  ESI = 003D1000 EDI = 003D1000
  EIP = 003D1004 ESP = 012FFEA4
  EBP = 012FFEB0 EFL = 00000246

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0
  PE = 1 CY = 0
```

OV：溢出位　　UP:方向标志位　　EI：中断标志位

PL：符号标志位　　ZR：零标志位　　AC：辅助标志位

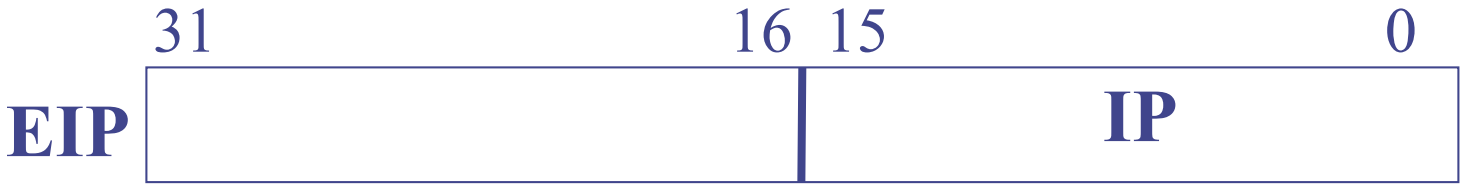
PE：奇偶标志位　　CY：进位标志位

2.5 指令预取部件和指令译码部件

- 指令预取部件：将要执行的指令从主存中取出，送入指令排队机构中排队。
- 指令译码部件：从指令队列中读出指令并译码，再送入译码指令队列排队供执行部件使用。
- 指令的提取、译码、执行重叠进行，形成了指令流水线。

2.5 指令预取部件和指令译码部件

指令指示器



保存着下一条将要被CPU执行的指令的偏移地址(简称EA)。

- EIP/IP的值由微处理器硬件自动设置
- 不能由指令直接访问
- 执行转移指令、子程序调用指令等可使其改变

2.6 分段部件和分页部件

分段部件中的段寄存器

6个16位的段寄存器：

CS：代码段寄存器 Code Segment

---不能由程序直接修改

DS：数据段寄存器 Data Segment

SS：堆栈段寄存器 Stack Segment

ES：附加数据段寄存器

FS:

GS:

2.6 分段部件和分页部件

- “.CODE”表示代码段，该段的选择子是段寄存器CS；
- “.DATA”表示数据段，该段的选择子是段寄存器DS；

  C 程序中定义的全局变量就存储在数据段中；
- “.STACK”表示堆栈段，该段的选择子是段寄存器SS；

  C程序中定义的局部变量、函数参数存储在堆栈段中。

好像感觉不到这些段寄存器的作用？
写程序时根本就没用？？？

简化段定义的Win32程序框架—回顾

華中科技大學

```
.686P
.model flat, c
  ExitProcess proto stdcall :dword
  includelib  kernel32.lib
  printf      proto c :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
 lpFmt    db "%d",0ah, 0dh, 0
.stack    200
.code
main proc
invoke ExitProcess, 0
main  endp
end
```

完整段定义的Win32程序


```
.686P
ExitProcess proto stdcall:dword
includelib kernel32.lib
printf      proto c :ptr sbyte, :vararg
includelib  libcmt.lib
includelib  legacy_stdio_definitions.lib
data segment
lpFmt db "%d", 0ah, 0dh
data ends
stack segment stack
db 200 dup(0)
stack ends
```

完整段定义的Win32程序


```
code segment EXECUTE
assume cs:code, ss:stack, ds:data
main proc c
    mov eax, 0
    mov ebx, 1
lp:cmp ebx,100
    jg  exit
    add eax, ebx
    inc ebx
    jmp lp
exit:
    invoke printf, offset lpFmt, eax
    invoke ExitProcess, 0
main endp
code ends
end
```

完整段定义的16位段程序


```
.386
data segment use16
  x  dd
data en
stack s
  db 20
stack e
code se
assume
start:
    mov ax, data
    mov ds, ax
```

```
    mov eax, 0
    mov ebx, 1
```

```
    mov ah, 4ch
    int 21h
code ends
end start
```

总结：

不管是简化的段定义，还是完整的段定义，段寄存器的作用都是存在的！

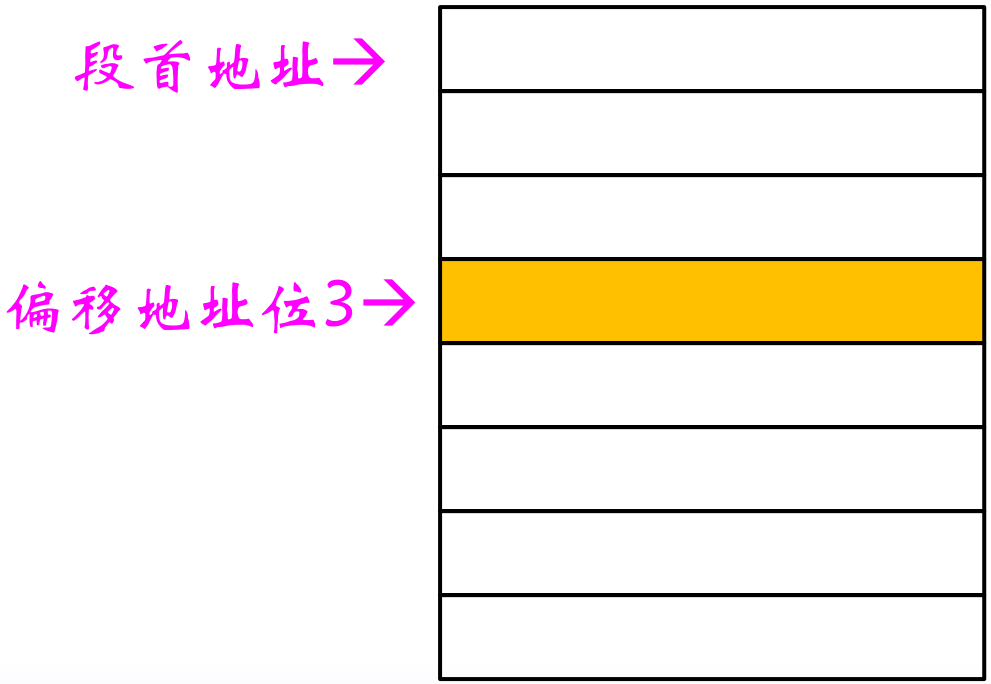
二维逻辑地址概念：

段寄存器：偏移地址

2.6 分段部件和分页部件

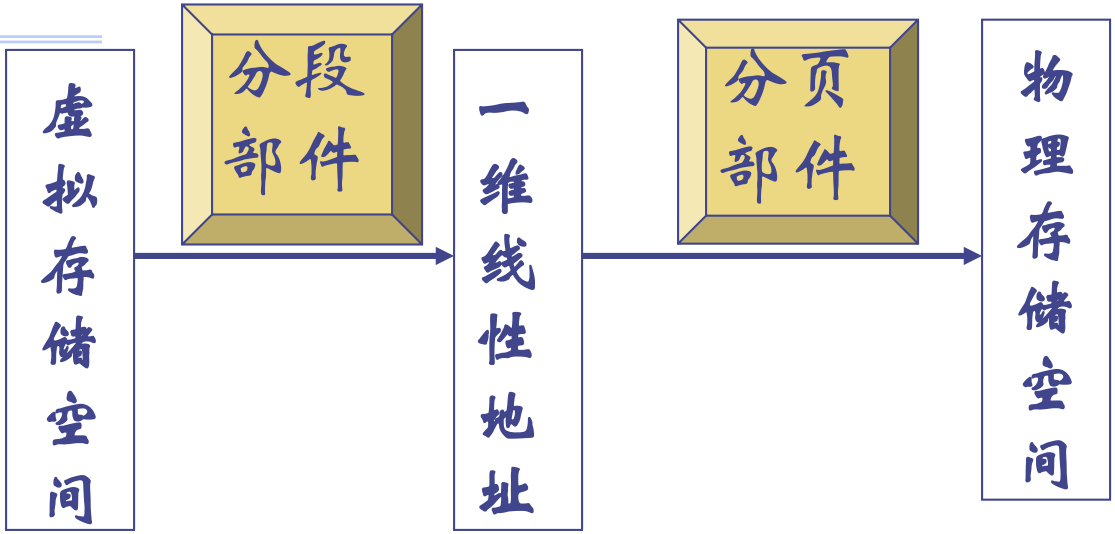
偏移地址：内存中的某个存储单元到段首址的字节距离

下图每个方框都代表一个字节



二维逻辑地址的概念：段寄存器：偏移地址

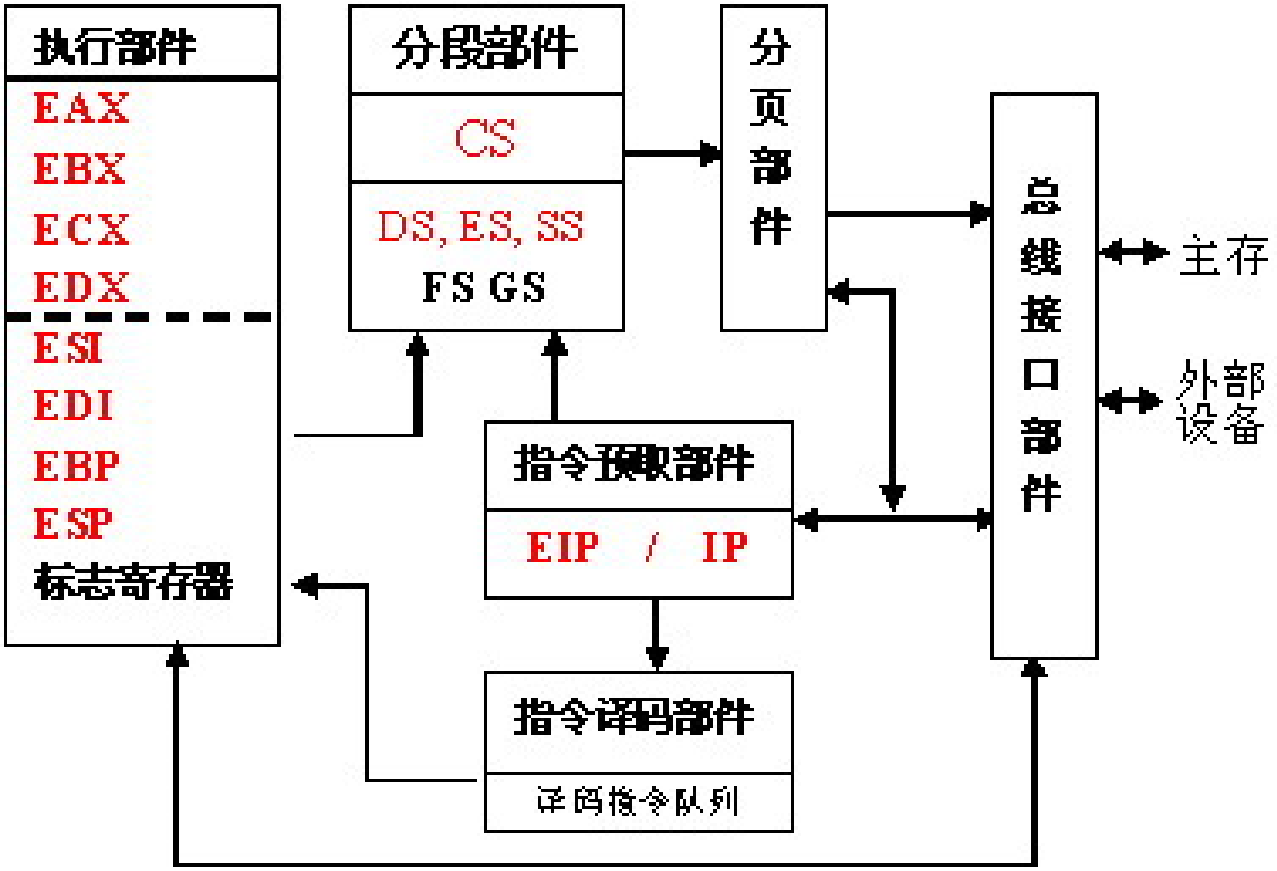
2.6 分段部件和分页部件



- 程序投入运行时，系统会为每个程序分配一片独立的虚拟存储空间。虚拟存储地址是概念性的逻辑地址，并非实际空间地址；
- 程序员编写程序时不用考虑物理存储器的大小。
- 存储管理单元MMU进行虚地址到实地址的自动变换。
- 地址变换对应用程序是透明的。

Intel x86微处理器结构

指令的执行过程



CS:EIP

2.7 x86的3种工作方式

1. 实地址方式

简称实方式

- 可以使用32位寄存器和32位操作数
- 可以采用32位的寻址方式。
- 此时的32位CPU与16位CPU一样，只能寻址1MB物理存储空间，程序段的大小不超过64KB，段基址和偏移地址都是16位的，这样的段也称为“16位段”。

2.7 x86的3种工作方式

2. 保护方式

- 使用32位地址线，寻址4GB的物理存储空间，段基址和段内偏移量都是32位的。
- 提供了支持多任务的硬件机构，能为每个任务提供一台虚拟处理器来仿真多台处理器；
- 实施执行环境的隔离和保护，对不同的段设立特权级并进行访问权限检查，以防不同的程序之间的非法访问和干扰破坏，使操作系统和各应用程序都受到保护。

2.7 x86的3种工作方式

3. 虚拟8086方式

- 在保护方式下运行的类似实方式的工作环境；
- 能充分利用保护方式提供的多任务硬件机构、强大的存储管理和保护能力；
- 多个8086程序可以通过分页存储管理机制，将各自的1MB地址空间映射到4GB物理地址的不同位置，从而共存于主存且并行运行。

第2章 Intel 中央处理器

作业： P33

2.1 2.5 2.7 2.8 2.9

上机实践：

将 2.5 题中完成的计算写成程序，观察运行结果及标志位的设置。将观察结果作为书面作业