

第2章 Intel 中央处理器



华中科技大学

学习内容

- 通用寄存器组
- 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



华中科技大学

- 成立于1968年
- INTEgrated ELectionics（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）、Pentium II、Pentium III、Pentium IV
- Xeon（至强）、Xeon3、Xeon5、Xeon7
- Itanium（安腾）、Itanium 2
- Core（酷睿）、i3、i5、i7、i9 系列





2.1 Intel微处理器的发展史

- 微处理器的结构分成两种：IA-32和IA-64。
- 在推出80486之后，英特尔以IA (Intel Architecture) 指称该架构，也称为 x86-32架构。
- 由于从8086开始其后产品以80186、80188、80286、i386、i486等为代号命名，因而被外界称为x86架构。
- 奔腾系列、Xeon系列、酷睿系列全部是基于x86架构的产品。
- 它属于复杂指令集架构

Complex Instruction Set Computer, CISC



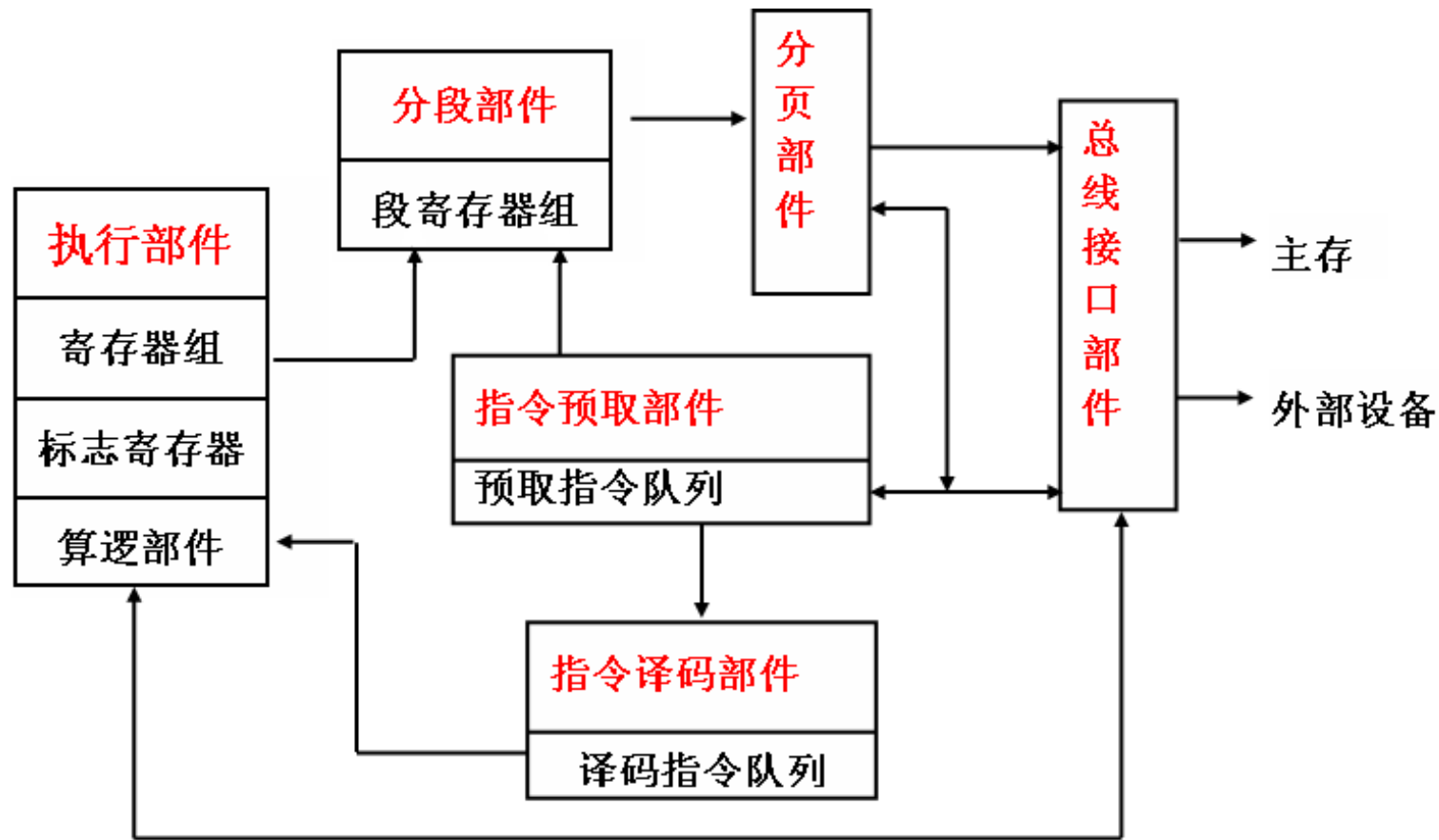


2.1 Intel微处理器的发展史

- **IA-64**架构是一种全新处理器架构，与x86不能兼容。
- 猜测：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。



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

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





2.3 执行部件

寄存器 Register

- 寄存器是什么？
- 为什么要有寄存器？
- 寄存器中可以存储什么呢？
- 有哪些寄存器？
- 如何使用寄存器？





2.3 执行部件

寄存器 Register

存储0、1串；可以是操作数、操作数地址等等

寄存器组中有8个32位的寄存器（通用寄存器）





2.3.1 32位CPU中的通用寄存器

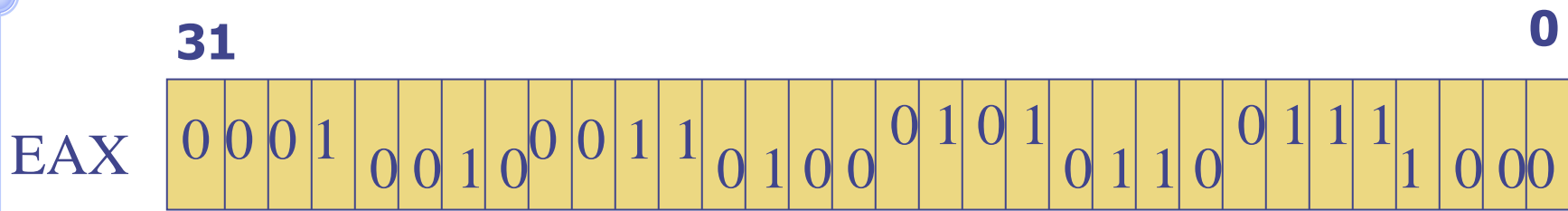
EAX、EBX、ECX、EDX、ESI、EDI、ESP、EBP

EAX: 累加器	Accumulator
EBX: 基址寄存器	Base
ECX: 计数器	Count
EDX: 数据寄存器	Data
ESI: 源变址寄存器	Source Index
EDI: 目的变址寄存器	Destination Index
ESP: 堆栈指示器	Stack Pointer
EBP: 堆栈基址寄存器	Base Pointer





2.3.1 32位CPU中的通用寄存器



(EAX)=12345678H

EAX 可以看成是某个存储单元的地址，即一存储单元地址的符号表示。

(EAX编号为 000，参见第4章 P72 表 4.4)。

(XX): 表示XX单元中的内容。

用符号代替数字编码的好处：便于记忆。

思考题：计算机是**0**、**1**的世界，**EAX**中的**0**，**1**串可代表什么含义？



2.3.1 32位CPU中的通用寄存器

EAX		AH	AL	累加器 (AX)		
EBX		BH	BL	基址寄存器 (BX)		
ECX		CH	CL	计数寄存器 (CX)		
EDX		DH	DL	数据寄存器 (DX)		
ESI		SI		源变址寄存器		
EDI		DI		目的变址寄存器		
EBP		BP		堆栈基址寄存器		
ESP		SP		堆栈指示器		
	31	16	15	8	7	0

16位寄存器: AX、BX、CX、DX、SI、DI、BP、SP

8位寄存器: AH、AL、BH、BL、CH、CL、DH、DL



2.3.1 32位CPU中的通用寄存器

- 为什么要设置寄存器?
- 什么叫通用寄存器?
- 通用寄存器又为何给予特定含义的名称?



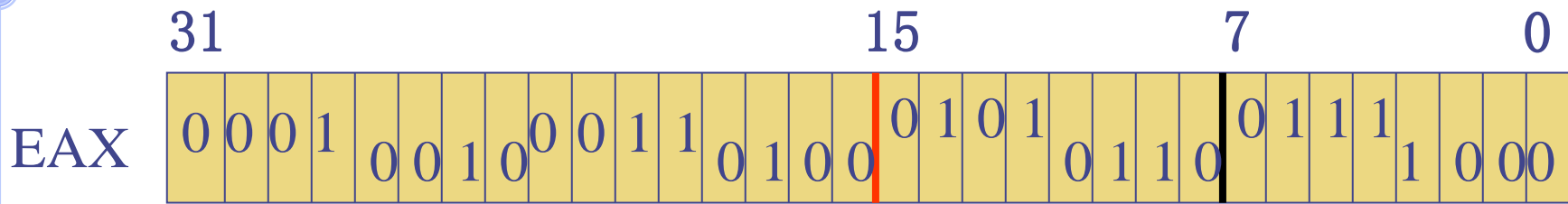
2.3.1 32位CPU中的通用寄存器

32 位寄存器	16 位寄存器	8 位寄存器	二进制编码
EAX	AX	AL	000
ECX	CX	CL	001
EDX	DX	DL	010
EBX	BX	BL	011
ESP	SP	AH	100
EBP	BP	CH	101
ESI	SI	DH	110
EDI	DI	BH	111

- 在机器指令码中，若知道使用了一个编号为000的寄存器，还需要什么信息，才能判断出使用的是EAX、AX、还是AL？
- 为什么不给所有的寄存器（24个寄存器）用5位二进制来编码呢？
- 如何理解寄存器的名字也是一个单元的符号地址？



2.3.2 通用寄存器应用示例



MOV EAX, 12345678H

Q: 将EAX的低16位全部置成0, 指令如何写?

MOV AX, 0

Q: 将EAX的低8~15位全部置成1, 指令如何写?

MOV AH, 0FFH



2.3.2 通用寄存器应用示例

Q: 将EAX的低16位全部置成0, 指令如何写?

MOV	AX, 0	
SUB	AX, AX	subtract
XOR	AX, AX	exclusive or
AND	AX, 0	and
SHL	AX, 16	SHift Left
IMUL	AX, 0	multiply

? 若想将EAX的高16位全部置成0, 怎么办?

AND EAX, 0000FFFFH \Leftrightarrow AND EAX, 0FFFFFFFH

SHL EAX, 16 SHR EAX, 16





x86-64位 CPU 中的通用寄存器

➤ 16个64位的通用寄存器

rax、rbx、rcx、rdx、rbp、rsi、rdi、rsp、
r8、r9、r10、r11、r12、r13、r14、r15

➤ 这些寄存器的低双字、最低字、最低字节都可以被独立的访问，各自都有自己的名字

➤ 16个 低双字寄存器（32位）

eax、ebx、ecx、edx、ebp、esi、edi、esp、
r8d -- r15d



x86-64位 CPU 中的通用寄存器

➤ 16个低字寄存器（16位）

ax、 bx、 cx、 dx、 bp、 si、 di、 sp、

r8w -- r15w

➤ 16个 最低字节寄存器（16个）

al、 bl、 cl、 dl、 bpl、 sil、 dil、 spl、

r8b -- r15b



x86-64位 CPU 中的通用寄存器

- AH、BH、CH、DH 依然可以使用
- 但是，一条指令不能同时引用旧的高字节 (AH、BH、CH、DH) 和一个新的字节寄存器最低字节寄存器 (bp1、sil、dil、spl、r8b – r15b)

```
mov ah, al
```

```
mov ah, 88h
```

```
mov ah, r8b
```



x86-64位 CPU 中的通用寄存器

Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

- REX前缀字节的组成部分为：0 1 0 0 W R X B
- REX.W : 为 1 时，操作数是 64 位
- REX.R : 用来扩展 ModR/M 中Reg字段
- REX.X : 用来扩展变址寄存器的编号
- REX.B : 用来扩展基址寄存器的编号
- 即由 R、X、B 与寻址方式中的三位寄存器编号一起，组成寄存器的 4位编码。

REX 的取值： 40H ——4FH。





华中科技大学

VS2019 中观察寄存器

The screenshot shows the Visual Studio 2019 interface during a debug session. The 'Debug' menu is open, and the 'Registers' option is highlighted under the 'View' submenu. The main editor displays the source code of a C program. The 'Registers' window on the right shows the current state of the CPU registers, including the instruction pointer (RIP) and various general-purpose registers (RAX, RBX, etc.). The status bar at the bottom indicates the current line of code and the character position.

调试(D) 测试(S) 分析(N) 工具(T) 扩展(X) 窗口(W) 帮助(H) 搜索 (Ctrl+Q)

窗口(W)

- 图形(C)
- 继续(C) F5
- 全部中断(K) Ctrl+Alt+Break
- 停止调试(E) Shift+F5
- 全部折离(L)
- 全部终止(M)
- 重新启动(R) Ctrl+Shift+F5
- 应用代码更改(A) Alt+F10
- 性能探查器(F)... Alt+F2
- 重启性能探查器(L) Shift+Alt+F2
- 附加到进程(P)... Ctrl+Alt+P
- 其他调试目标(H)

逐语句(S) F11

逐过程(O) F10

跳出(T) Shift+F11

快速监视(Q)... Shift+F9

切换断点(G) F9

新建断点(B)

删除所有断点(D) Ctrl+Shift+F9

启用所有断点(N)

禁用所有断点(N)

清除所有数据提示(A)

导出数据提示(X) ...

导入数据提示(I) ...

将转储另存为(V) ...

选项(O)...

c_example 调试属性

断点(B) Ctrl+Alt+B

异常设置(X) Ctrl+Alt+E

输出(O)

XAML 绑定失败(F)

显示诊断工具(T) Ctrl+Alt+F2

GPU 线程(U)

任务(S) Ctrl+Shift+D, K

并行堆栈(K) Ctrl+Shift+D, S

并行监视(R)

监视(W)

自动窗口(A) Ctrl+Alt+V, A

局部变量(L) Ctrl+Alt+V, L

即时(I) Ctrl+Alt+I

实时可视化树(V)

实时属性资源管理器(P)

调用堆栈(C) Ctrl+Alt+C

线程(H) Ctrl+Alt+H

模块(O) Ctrl+Alt+U

进程(P) Ctrl+Alt+Z

诊断分析(D) Ctrl+Shift+Alt+D

内存(M)

反汇编(D) Ctrl+Alt+D

寄存器(G) Ctrl+Alt+G

类型

名称	类型
argv	int
argv	char **
x	int
y	int
z	int

诊断工具

诊断会话: 0 秒

事件

进程内存

CPU (所有处理器的百分比)

摘要

事件

内存 73%

CPU 使用率

截取快照

启用堆分析(会影响性能)

CPU 使用率

- 记录 CPU 配置文件

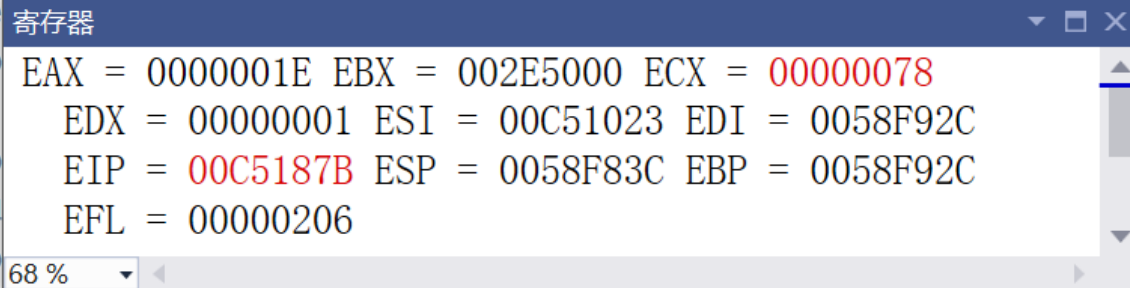
调试 -> 窗口 -> 寄存器





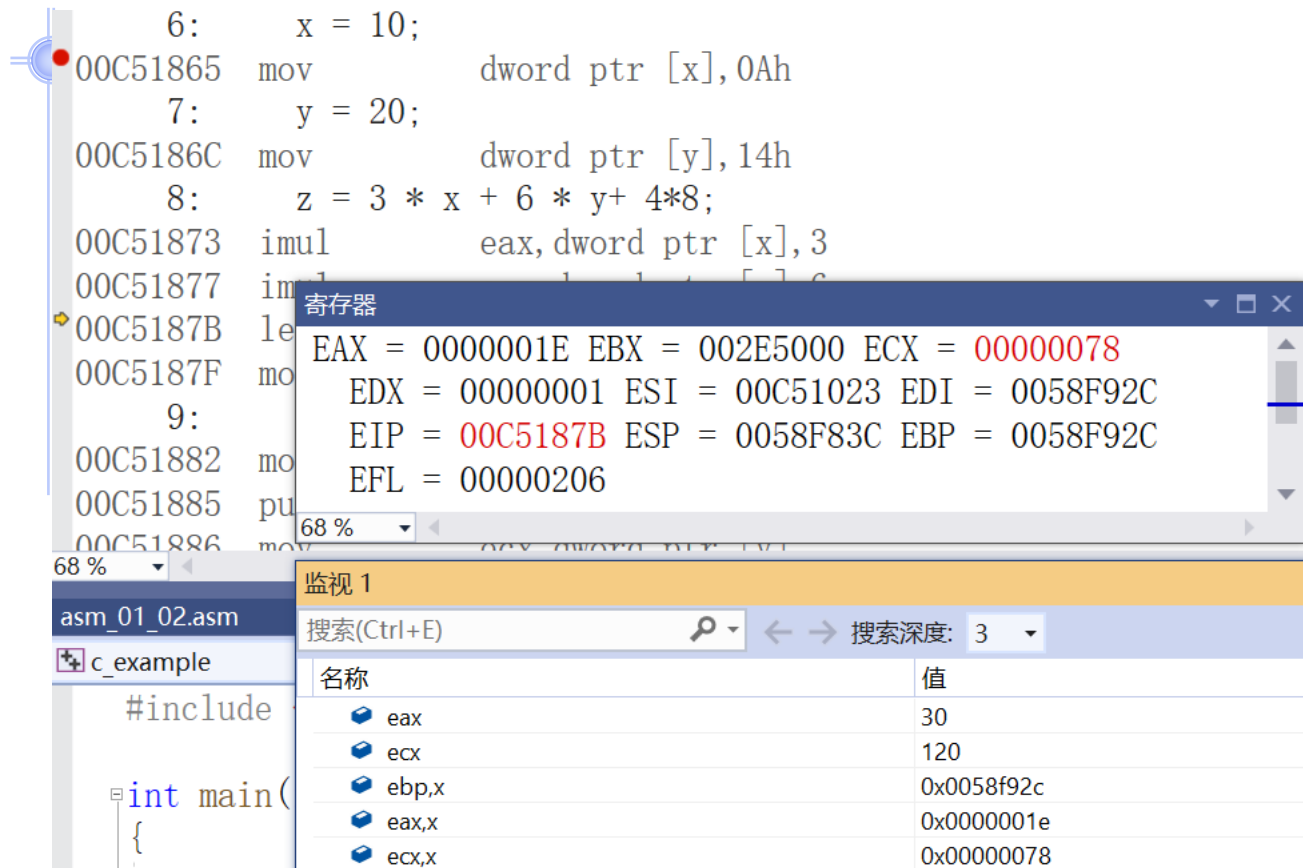
VS2019 中观察寄存器

```
5:      int  x, y, z;
6:      x = 10;
00C51865 mov     dword ptr [x], 0Ah
7:      y = 20;
00C5186C mov     dword ptr [y], 14h
8:      z = 3 * x + 6 * y + 4*8;
00C51873 imul    eax, dword ptr [x], 3
00C51877 imul    ecx, dword ptr [y], 6
00C5187B lea     edi, dword ptr [y], 6
00C5187F mo     edi, dword ptr [y], 6
9:
00C51882 mo     edi, dword ptr [y], 6
00C51885 pu     edi, dword ptr [y], 6
00C51886 mo     edi, dword ptr [y], 6
```



执行 `imul eax, dword ptr [x], 3` 后 (eax) = 0000001E 即 30
执行 `imul ecx, dword ptr [y], 6` 后 (ecx) = 00000078 即 120
寄存器窗口 中 以 16进制显示寄存器中的值

VS2019 中观察寄存器



The screenshot displays the Visual Studio 2019 interface during a debugging session. The assembly window shows the following code:

```
6:      x = 10;
00C51865 mov     dword ptr [x], 0Ah
7:      y = 20;
00C5186C mov     dword ptr [y], 14h
8:      z = 3 * x + 6 * y + 4 * 8;
00C51873 imul    eax, dword ptr [x], 3
00C51877 imul    eax, eax, 6
00C5187B lea     eax, [eax + 32]
00C5187F mov     ecx, 4
00C51882 mov     ecx, ecx * 8
00C51885 push    ecx
00C51886 mov     ecx, 4
```

The **寄存器** (Registers) window shows the current state of the CPU registers:

- EAX = 0000001E
- EBX = 002E5000
- ECX = 00000078
- EDX = 00000001
- ESI = 00C51023
- EDI = 0058F92C
- EIP = 00C5187B
- ESP = 0058F83C
- EBP = 0058F92C
- EFL = 00000206

The **监视 1** (Watch 1) window shows the values of the selected registers:

名称	值
eax	30
ecx	120
ebp,x	0x0058f92c
eax,x	0x0000001e
ecx,x	0x00000078

观察寄存器：寄存器窗口
监视窗口，输入要观察的寄存器的值



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**；
- 64位CPU中的标志寄存器是64位，称**RFLAGS**；

历史故事：

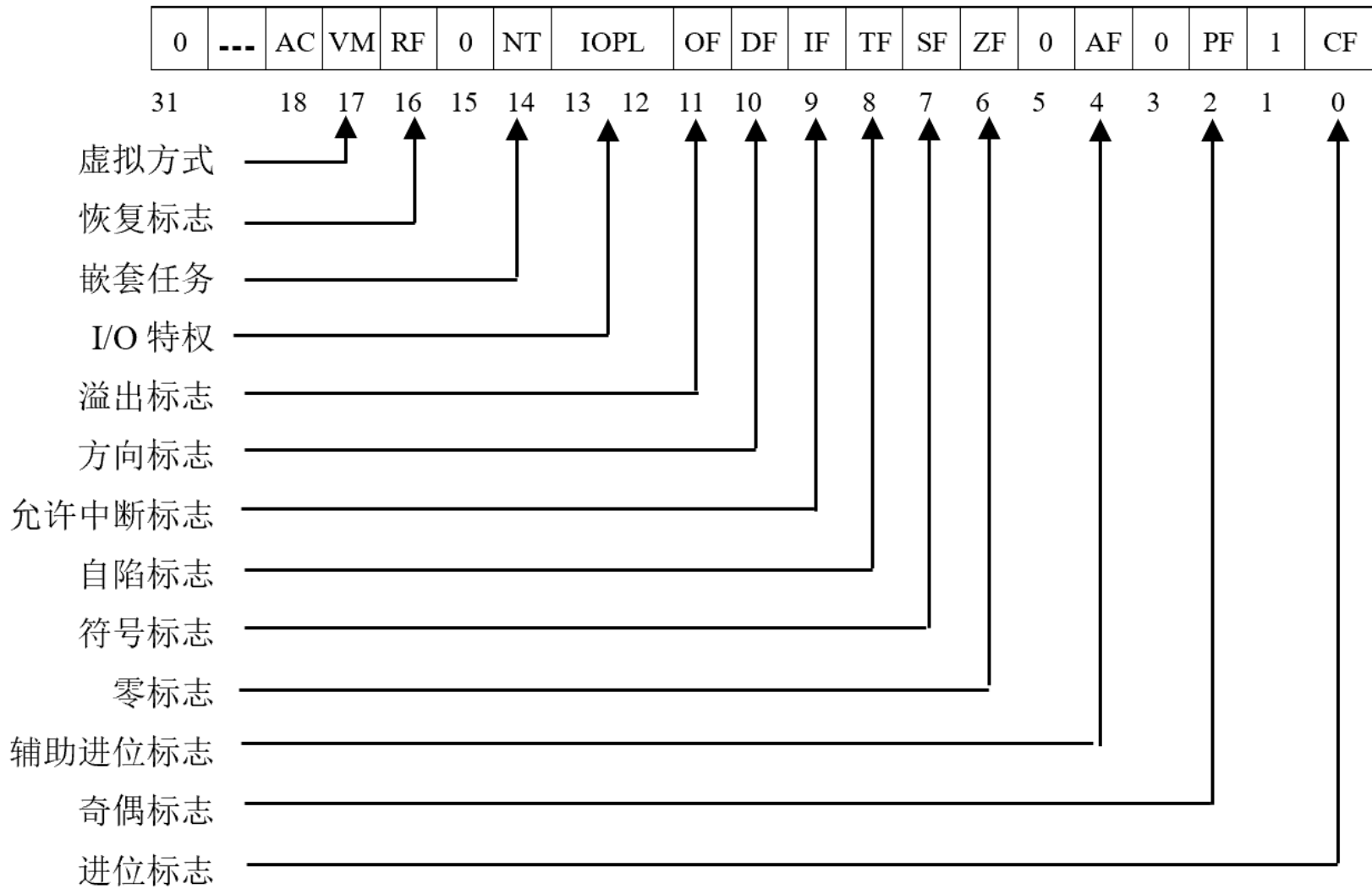
- 16位CPU中的标志寄存器是16位，称FLAGS；
- 32位的EFLAGS包含了16位FLAGS的全部标志位且向下兼容。





2.4 标志寄存器

】



2.4 标志寄存器

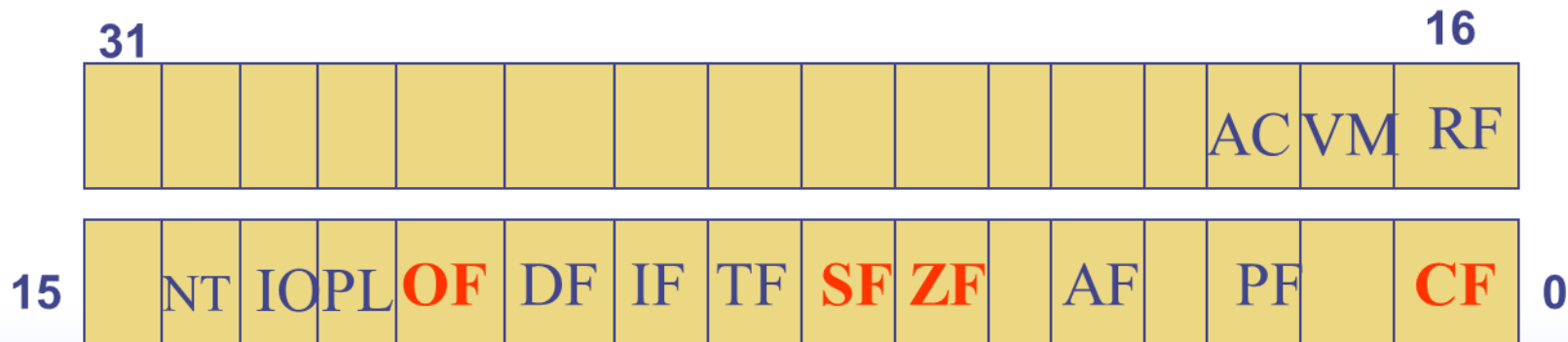
2.4.1 条件标志位

Sign Flag 符号标志

Zero Flag 零标志

Overflow Flag 溢出标志

Carry Flag 进位标志





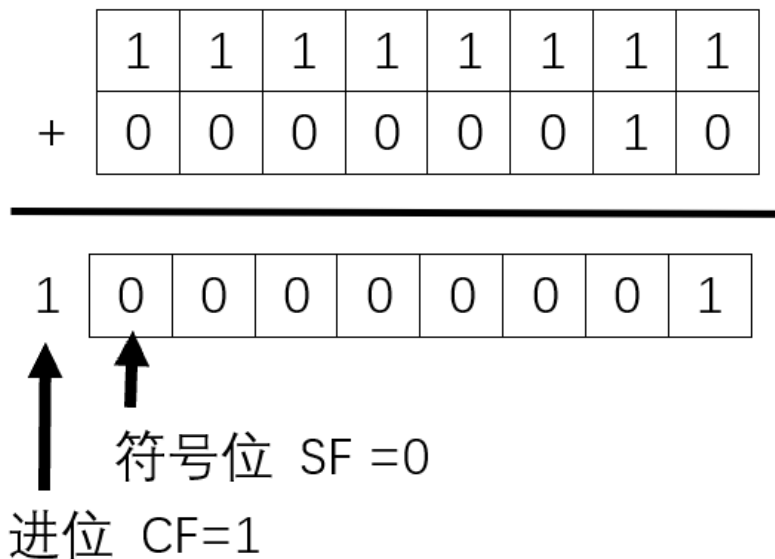
2.4 标志寄存器

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、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。



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

(AH) = 1



2.4 标志寄存器

3、零标志 ZF (Zero Flag)

运算结果为0，则 $ZF=1$ ，否则 $ZF=0$

4、溢出标志 OF (Overflow Flag)

加法：执行前两个加数的最高二进制位相同，
且加的结果的最高位与加数最高位相反，
则 $OF=1$ 。



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

$(AH) = 1$
 $CF=1, SF=0$
 $ZF=0, OF=0$

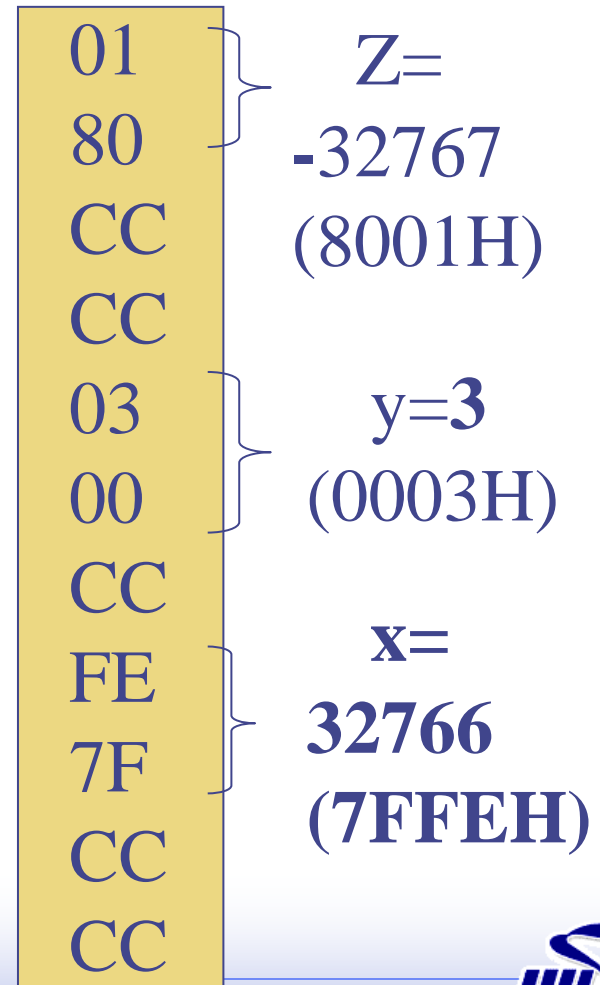


2.4 标志寄存器

溢出标志设置的直观理解

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

32766 3 -32767





2.4 标志寄存器

溢出标志设置的直观理解

```
x = 32766;    // 7FFEh
y = 3;        // 0003h
z = x + y;    // 8001h    -32767
```

监视 1

搜索(Ctrl+E) 🔍 < > 搜索深度: 3 ▼ A

名称	值
x	32766
y	3
z	-32767
&x	0x0136f7f4 {32766}
&y	0x0136f7f0 {3}
&z	0x0136f7ec {-32767}

C:\新书示例\C02_Intel ...

32766 3 -32767

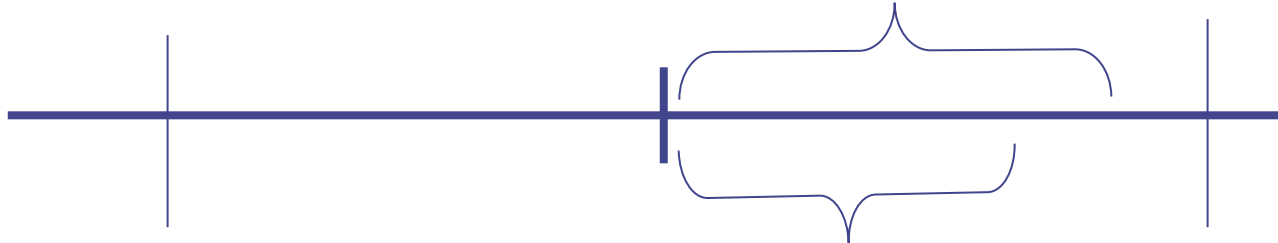
搜狗拼音输入法 半 :



2.4 标志寄存器

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。

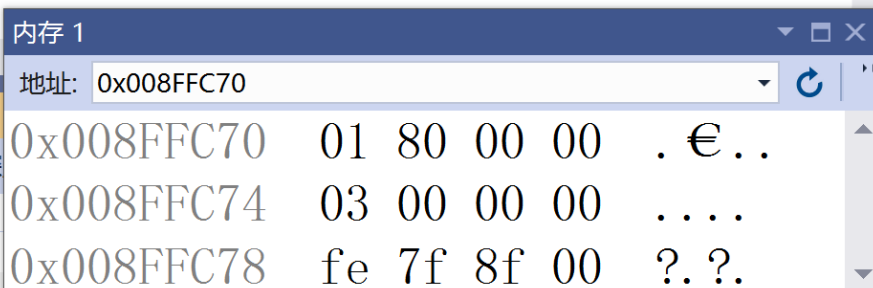
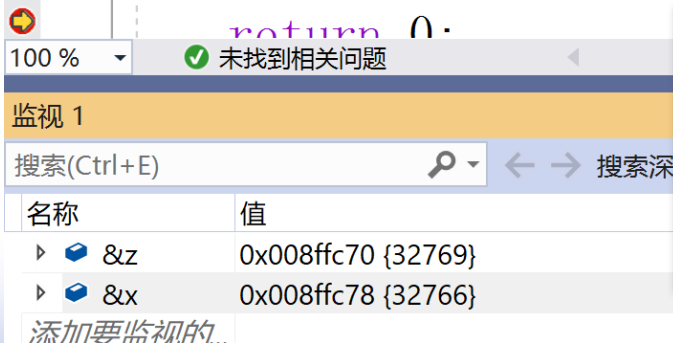
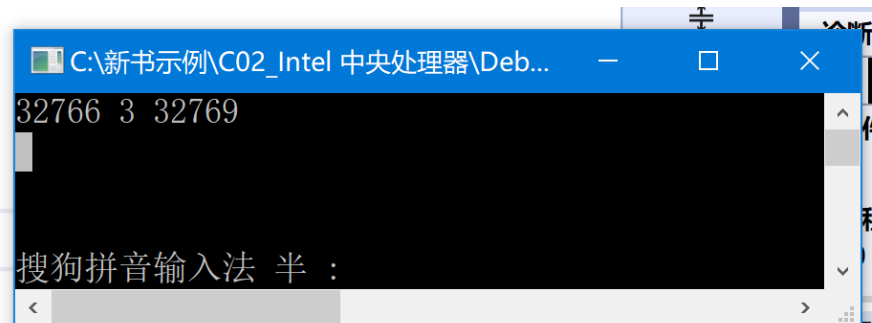
若被减数与减数的最高位不同，差的最高位与
被减数的最高位不同，则OF=1；否则OF=0.



2.4 标志寄存器

Q: 将 x, y, z 的定义 改为 unsigned short,
x=32766; y=3; z=x+y;
printf(“%d”,z); 结果如何?

```
int main()  
{  
    unsigned short  x, y, z;  
    x = 32766;      // 7FFEh  
    y = 3;  
    z = x + y;  
    printf(“%d %d %d \n”, x, y, z);  
    return 0;  
}
```





2.4 标志寄存器

结论：

不论将 x, y, z 定义为 unsigned short, 还是 short,
执行：`x=32766; y=3; z=x+y;`
 z 都是 8001H; 并且标志位设置是相同的。

add 指令不区分有符号数加法和无符号数加法。

- 两次显示结果不同。即将 8001H, 分别当成有符号数、无符号数解释, 得到的结果不同。
- 对于**有**符号数运算, 是否溢出可判断 **OF** 是否为 1;
对于**无**符号数运算, 是否溢出可判断 **CF** 是否为 1;



讨论



华中科技大学

8001H 当无符号short数，是 32769

当有符号short数，是 -32767

为什么一个数，能当有符号看，也可以当成无符号数看？

这两个数之间有什么关系？



8 点，也可视为 到 0 点 差 4 个小时

12 点 即 0 点

从 0 出发，顺时针走 8 格

从 0 出发，逆时针走 4 格

对时钟：[-4]补 = 12 - 4 = 8



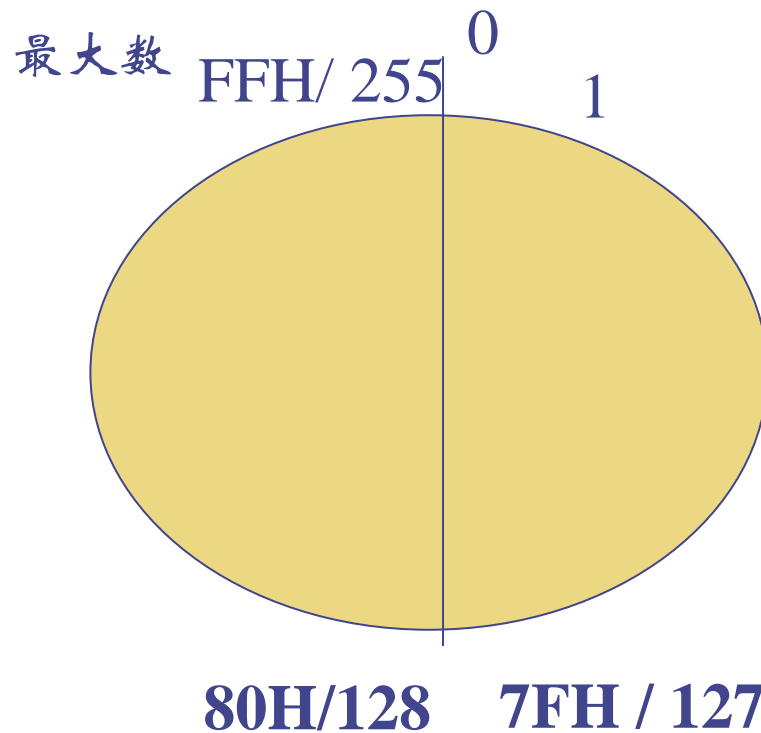
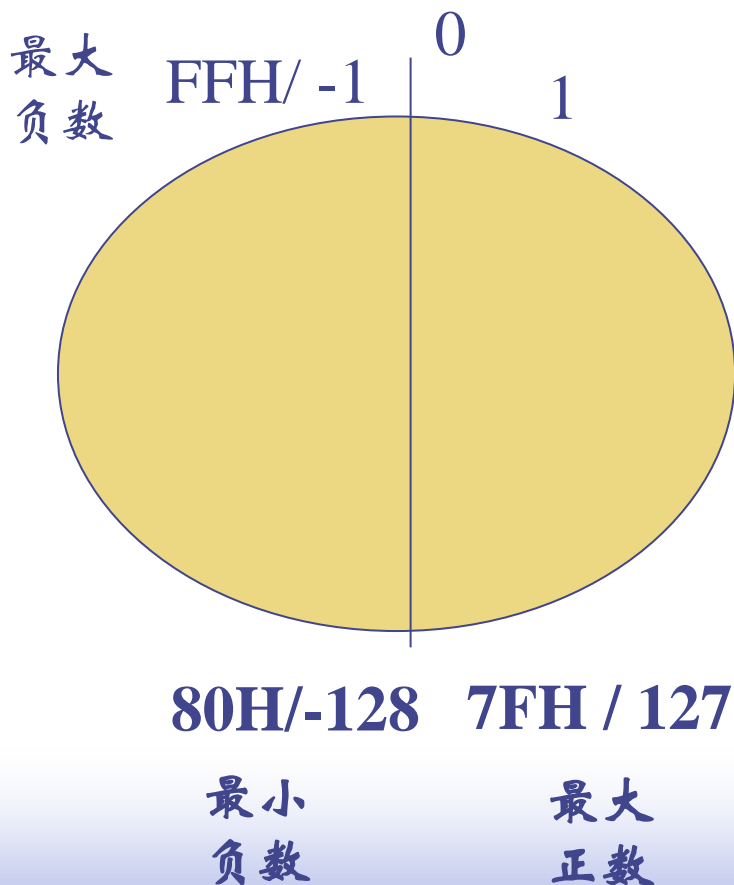
讨论



华中科技大学

一个字节数据：有符号

VS 无符号



讨论



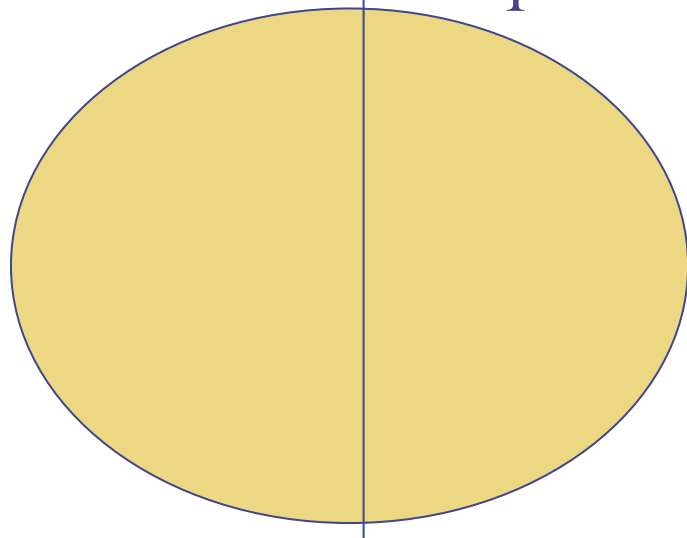
华中科技大学

一个字数据：有符号

VS 无符号

最大
负数

FFFFH/ -1 0 1



8000H/-32768

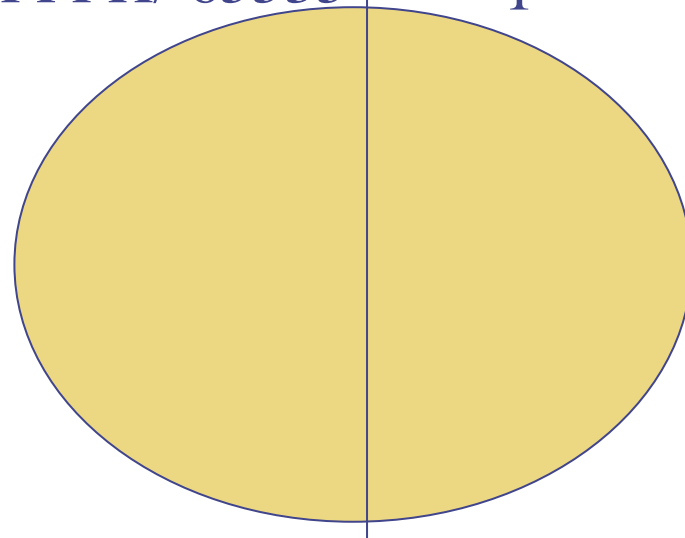
最小
负数

7FFFH / 32767

最大
正数

最大数

FFFFH/ 65535 0 1



8000H/
32768

7FFFH /
32767



测验



华中科技大学

```
void f()
{
    char x = -255;
    char y = 1;
    if (x == y)
        printf("%d %d is equal \n", x, y);
    else printf("%d %d not equal\n",x,y);
}
```

1 1 is equal

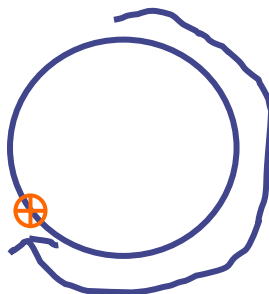
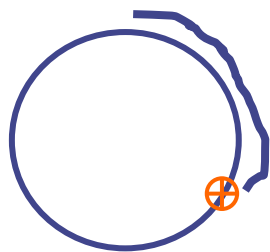
若 char x = 255, y= -1;
结果又如何?



理解正、负

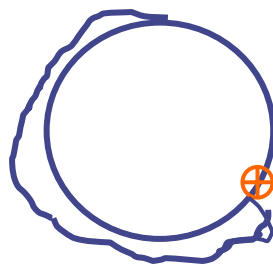
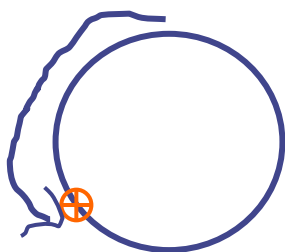


华中科技大学



正：顺时针转

负：逆时针转



N=8时 (char) ,

+255 《=》 -1

-255 《=》 1

+257 《=》 1

-257 《=》 -1

补码的计算：有正号的数，视为顺时针转的结果
有符号的数，视为逆时针转的结果

注：C编译器对超范围的数，采用截断的处理策略



补码的计算方法

设 $n=16$, $-69DAH$ 的补码表示是多少?

$$2^{16} - 69DAH = 9626H$$

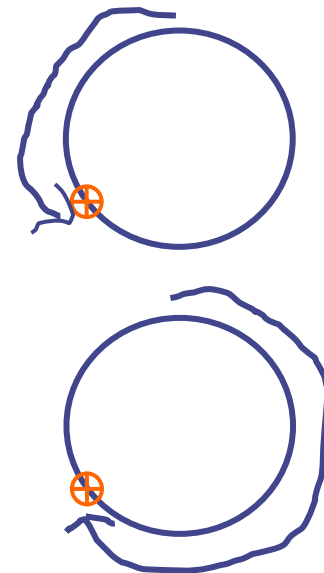
$$[-69DAH]_{补} = 9626H$$

$$69DAH + [-69DAH]_{补} = 2^n \quad (n=16)$$

$$2^{16} = FFFFH + 1$$

$$\begin{aligned} 2^{16} - 69DAH &= (FFFFH - 69DAH) + 1 \\ &= FFFFH - (69DAH - 1) \end{aligned}$$

$$\begin{array}{r} 69DAH \\ + \quad ? ? ? ? \quad (9625H) \\ \hline FFFFH \end{array}$$



方法1: $2^n - \text{数}$

方法2: 求反后加1

方法3: 减1后求反

验证结果的正确性:

补码与负数的和为 2^n

补码运算

设 $n=8$

$$\begin{aligned} 5 - 1 &= 5 + (-1) \\ &= 5 + [-1]_{\text{补}} \\ &= 05 + \text{FFH} \\ &= 4 \end{aligned}$$



8点钟，逆时针转1个小时，为7点；
等同于从8点钟，顺时针转11个小时

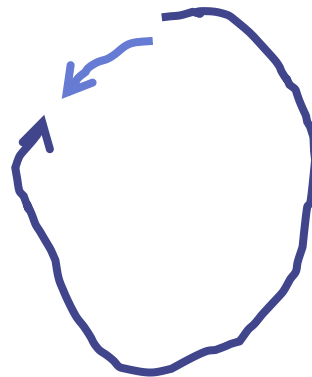
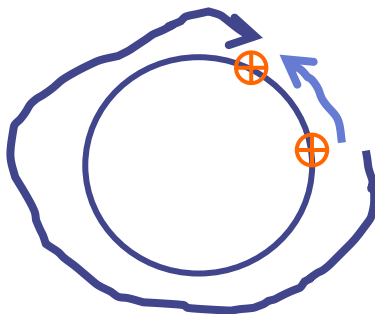
加：顺时针转

减：逆时针转

补码运算

设 $n = 8$

$[-1]_{\text{补}} = \text{FFH}$



$m - n$ 为什么等于 $m + [-n]_{\text{补}}$?

$$5 - 1 = 0x05 + 0xFF = 4$$



2.4 标志寄存器

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?

$$X1 = + 110\ 0101B$$

$$X2 = - 101\ 1101B$$

$$[X1]_{\text{补}} = 0110\ 0101\ B$$

$$\begin{array}{r} [X2]_{\text{补}} = 1010\ 0011\ B \\ + \quad \boxed{1}0000\ 1000\ B \end{array}$$

$$SF = 0$$

$$ZF = 0$$

$$OF = 0$$

$$CF = 1$$

$$-x2 = 0101\ 1101\ B$$

$$\text{求反 } 1010\ 0010\ B$$

$$\text{加1 } 1010\ 0011\ B$$

$$101 + (-93) = 8$$

$$[X1]_{\text{补}} + [X2]_{\text{补}} = 8$$





2.4 标志寄存器

实验验证

```
.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

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 EE
 ECX = 003D1000
 ESI = 003D1000
 EIP = 003D1004
 EBP = 012FFEB0

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



在寄存器窗口按鼠标右键，在弹出的窗口中单击“标志”（勾选标志），则会显示标志寄存器的几个位

OV : 1 为 OverFlow; PL : 1 为 Negative
 ZR : 1 为 Zero CY : 1 为 Carry



2.4 标志寄存器

2.4.2 控制标志位

1. 方向标志DF (Direction Flag)

串操作指令中使用，DF=0，正向(即从低地址向高地址)处理数据串。DF =1 (Down)

2. 中断允许标志IF (Interrupt Flag)

IF=1，CPU开中断，即CPU响应外设的中断请求。

3. 跟踪标志 TF (Trace Flag)

TF=1，CPU处于单步工作方式，即每执行完一条指令后，CPU自动产生一个类型为1的中断，使程序单步执行。





2.4 标志寄存器

2.4.3 系统标志位

1. IO特权标志 IOPL

IOPL共占2位，它指定了要求执行I/O指令的特权级。

如果CPU当前特权级等于或高于IOPL时，

则I/O指令可以执行，否则会产生一个保护异常。

2. 嵌套任务标志 NT

控制中断返回指令的执行。NT=1，CPU当前执行的任务

嵌套在另一个任务之中，待执行完该任务时，应返回

原来的任务中；否则，按堆栈中保存的断点返回。





2.4 标志寄存器

2.4.3 系统标志位

3. 重启动标志 RF

该标志位与系统调试寄存器一起使用，以确定是否接受调试故障。当一条指令成功执行时，CPU将RF清0。若RF置1时，下一条指令的所有调试故障被忽略，否则接受调试故障。

4. 虚拟8086方式标志 VM

当VM置1时，说明CPU在虚拟8086方式下工作，否则在保护方式下工作。





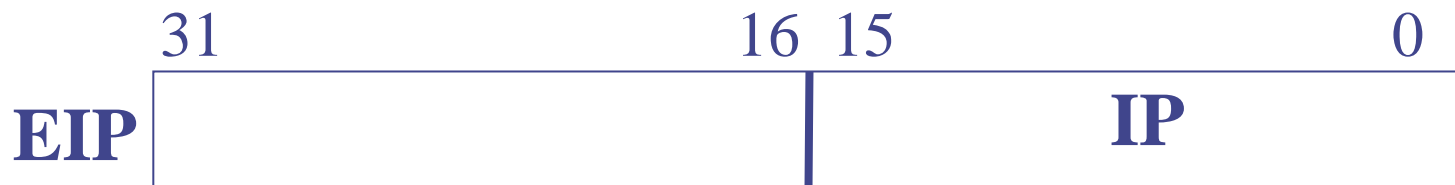
2.5 指令预取部件和指令译码部件

- **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



指令流水线的基本概念

◆ 五段流水线

取指令(IF): 根据PC的值从存储器取出指令。

指令译码(ID): 产生指令执行所需的控制信号。

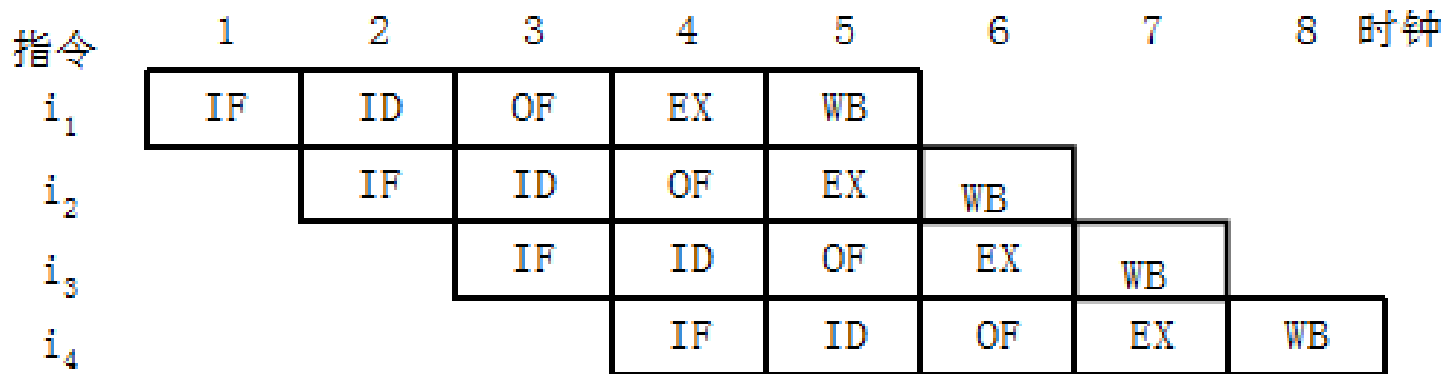
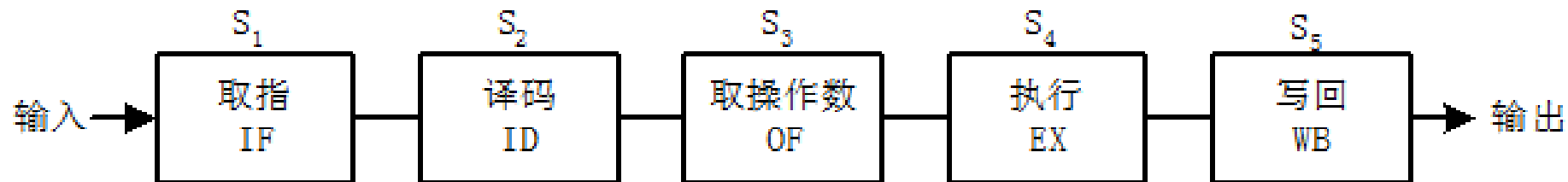
取操作数(OF): 读取存储器操作数或寄存器操作数。

执行(EX): 对操作数完成指定操作。

写回(WB): 将操作结果写入存储器或寄存器。



指令流水线的基本概念

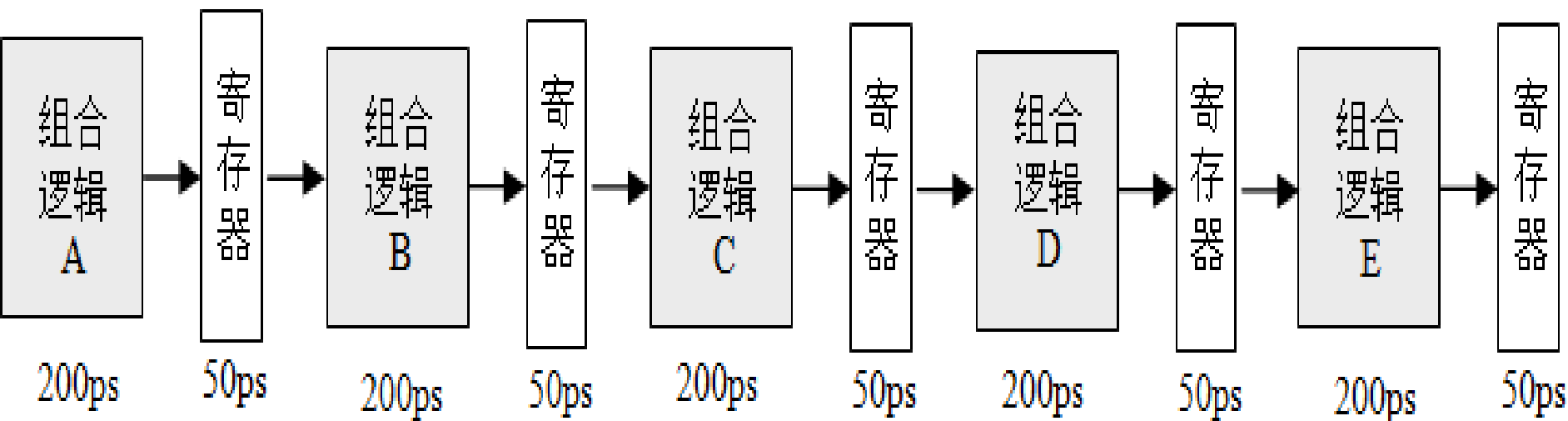


Q: 指令流水线的优点是什么?

编写什么样的程序, 能够更好地发挥流水线的作用?



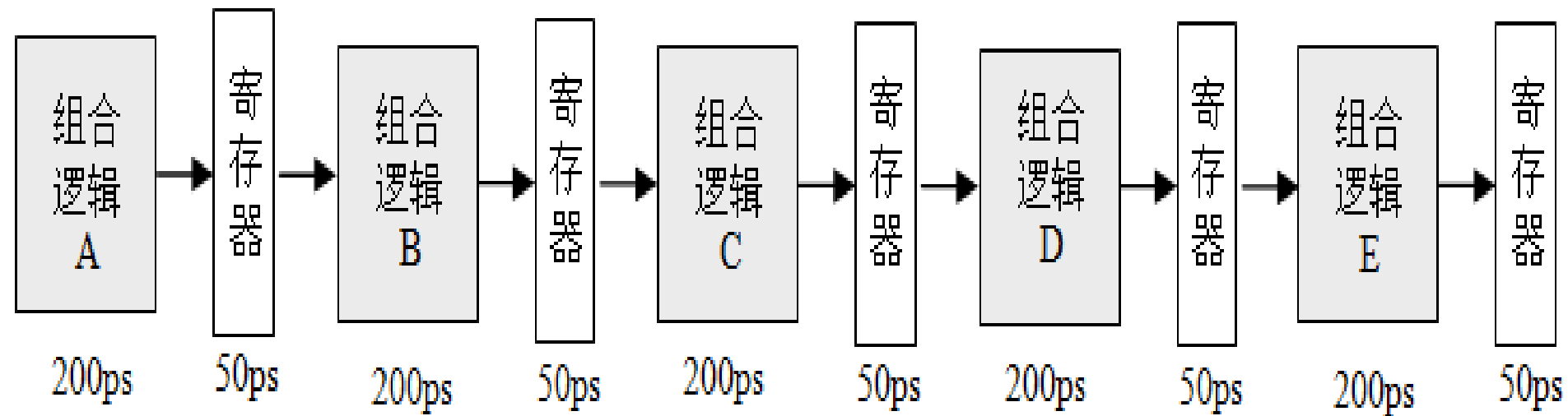
指令流水线的基本概念



取指令	译码/读数	ALU运算	读/写存储器	写结果
1A				
2A	1B			
3A	2B	1C		
4A	3B	2C	1D	
	4B	3C	2D	1E
		4C	3D	2E
			4D	3E
				4E

时间 ↓

指令流水线的基本概念



Q: 指令流水线有无效率不高的场景?

举例说明

Q: 简单地使用指令流水线, 有无可能产生错误?

或者说, 在生成执行程序时, 要避免的问题?



流水线的冲突/冒险(hazard)情况

Hazards: 指流水线遇到无法正确执行后续指令或执行了不该执行的指令

结构冒险 (hardware resource conflicts, 硬件资源冲突):

现象: 同一个部件同时被不同指令所使用

一个部件每条指令只能使用1次, 且只能在特定周期使用

设置多个部件, 以避免冲突。如指令存储器IM 和数据存储器DM分开

数据冒险 (data dependencies, 数据相关):

现象: 后面指令用到前面指令结果数据时, 前面指令的结果还没产生

转发(Forwarding/Bypassing旁路) 或 前半周期读后半周期写

Load-use冒险不能通过转发解决, 需阻塞(stall)一个时钟周期

编译程序优化指令顺序

控制 (分支) 冒险 (changes in program flow, 改变控制流):

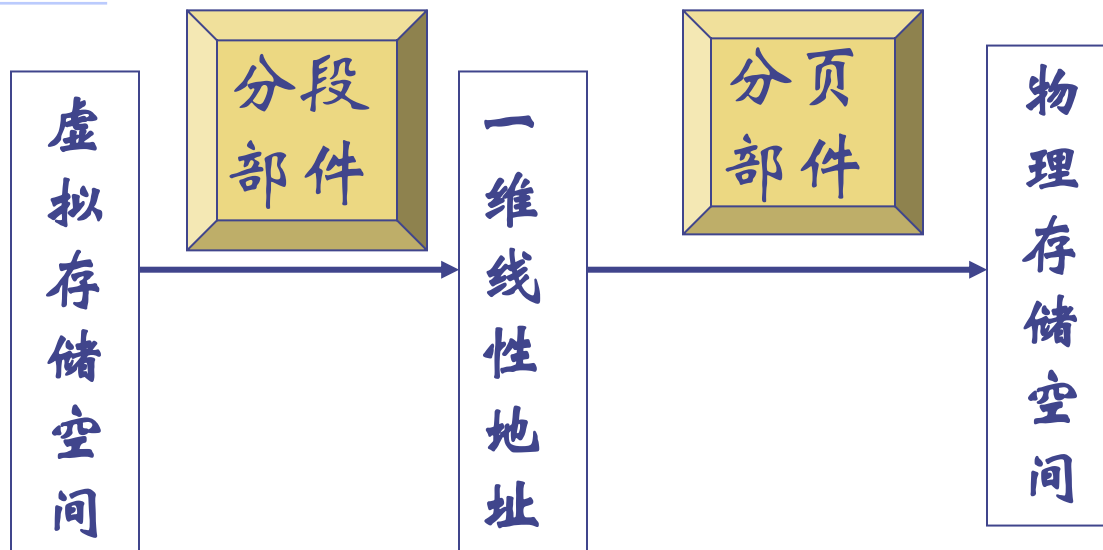
现象: 转移或异常改变执行流程, 后继指令在目标地址产生前已被取出

采用静态或动态分支预测

编译程序优化指令顺序(分支延迟)



2.6 分段部件和分页部件



- 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。



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

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。





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

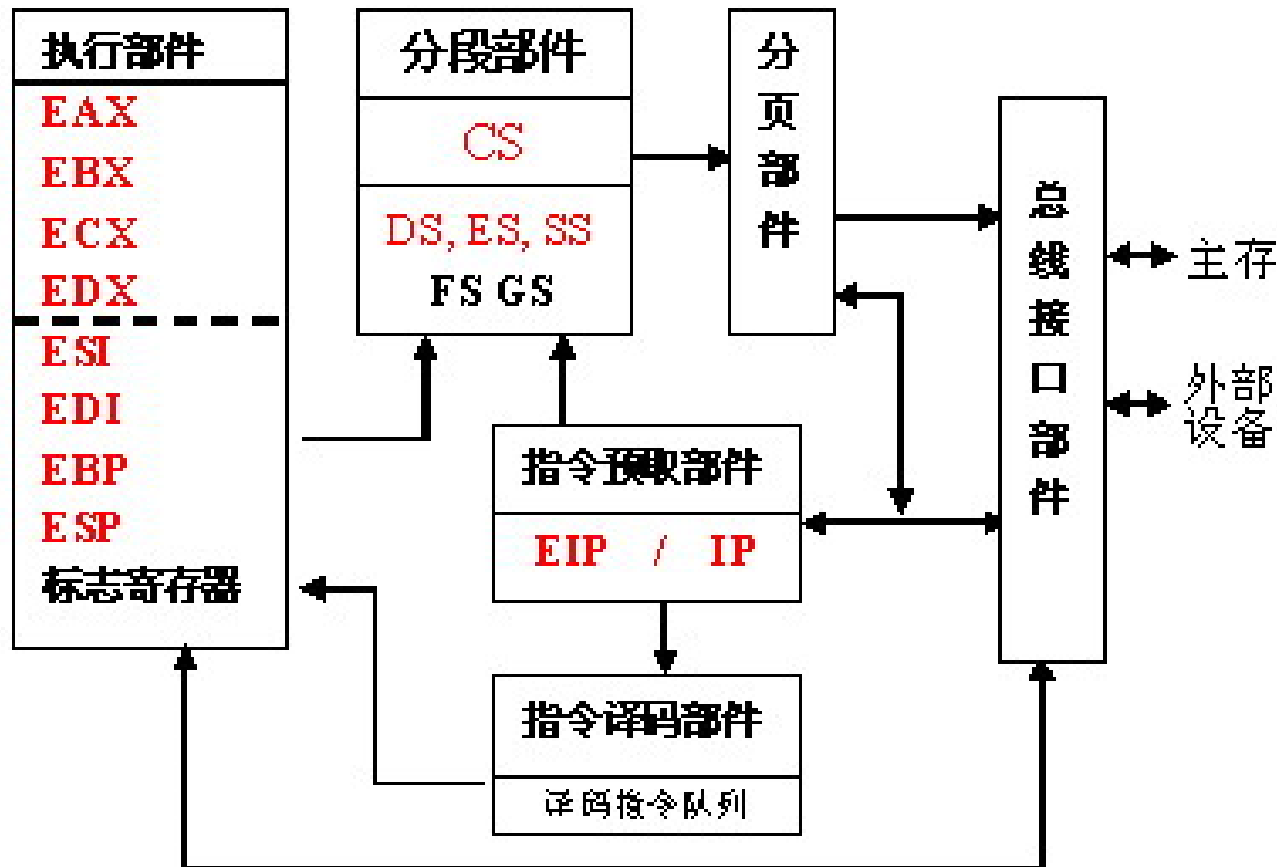
3. 虚拟8086方式

- 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

指令的执行过程



x86微处理器结构



華中科技大學

