IA-32中的传送指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

IA-32常用指令类型

（1）传送指令

- 通用数据传送指令

  MOV：一般传送，包括movb、movw和movl等

  MOVS：符号扩展传送，如movsbw、movswl等

  MOVZ：零扩展传送，如movzwl、movzbl等

  XCHG：数据交换

  PUSH/POP：入栈/出栈，如pushl,pushw,popl,popw等

- 地址传送指令

  LEA：加载有效地址，如leal (%edx,%eax), %eax”的功能为
  R[eax]←R[edx]+R[eax]，执行前，若R[edx]=*i*,
  R[eax]=*j*，则指令执行后，R[eax]=*i*+*j*

- 输入输出指令

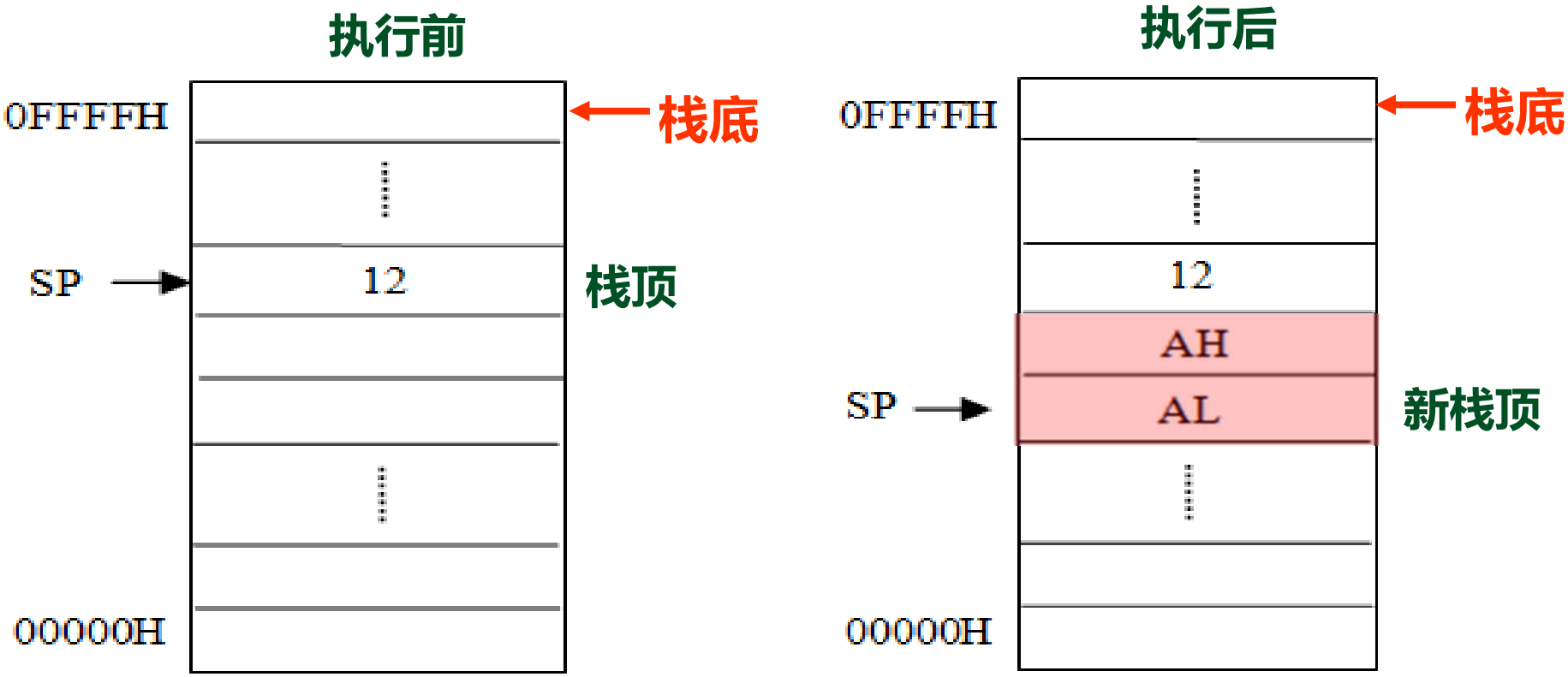
  IN和OUT：I/O端口与寄存器之间的交换

- 标志传送指令

  PUSHF、POPF：将EFLAG压栈，或将栈顶内容送EFLAG

“入栈”（pushw %ax）

- 栈（Stack）是一种采用“先进后出”方式进行访问的一块存储区，用于嵌套过程调用。从高地址向低地址增长
- “栈”不等于“堆栈”（由“堆”和“栈”组成）

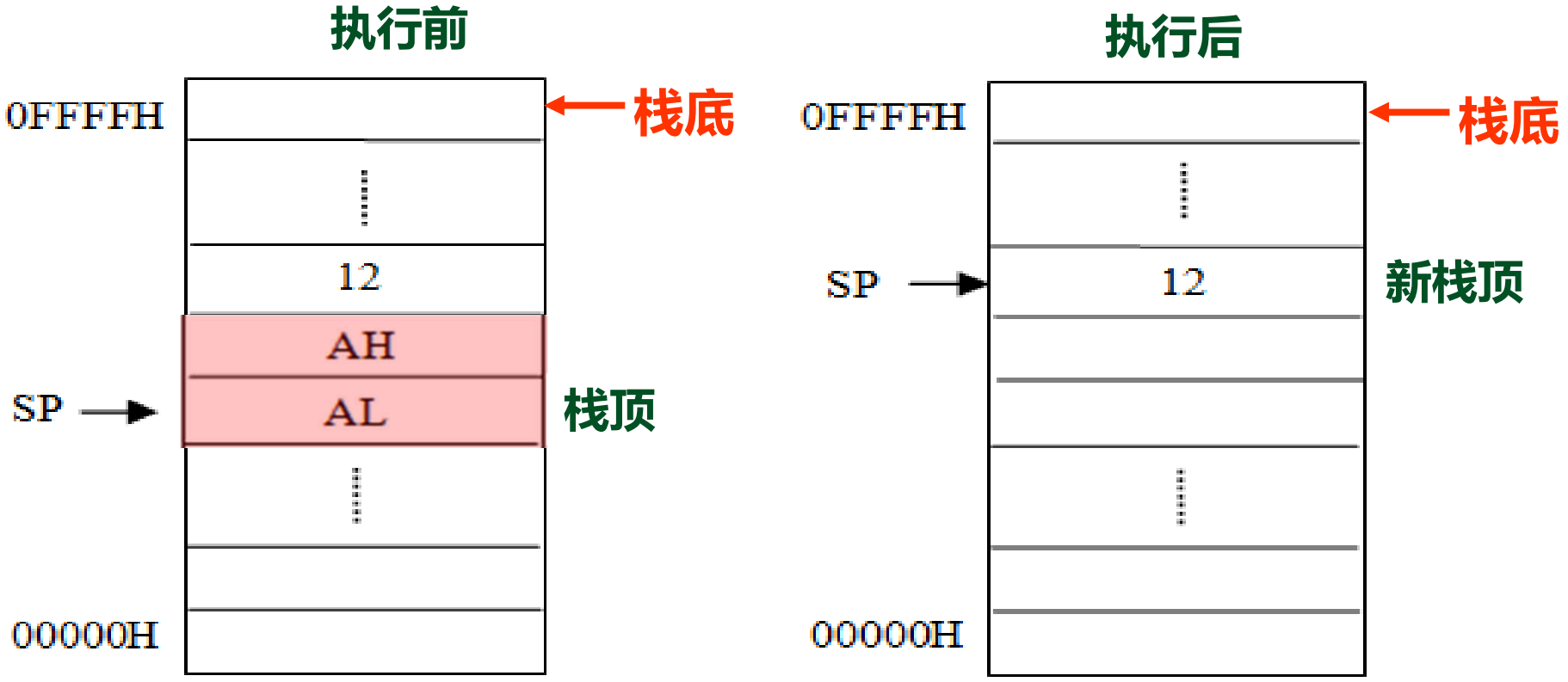


R[sp]←R[sp]-2、
M[R[sp]]←R[ax]

为什么AL在栈顶?　小端方式!

“出栈” （popw %ax）

- 栈（Stack）是一种采用“先进后出”方式进行访问的一块存储区，用于嵌套过程调用。从高地址向低地址增长



R[ax]←M[R[sp]]、
R[sp]←R[sp]+2

原栈顶处的数据送AX

程序由指令序列组成

```
1   // test.c
2   #include <stdio.h>
3   int add(int i, int j )
4   {
5       int x = i +j;
6       return x:
7   }
```

add函数中有哪些传送指令?

每一条传送指令的功能是什么?

“objdump -d test” 结果

指令的功能用RTL描述

```
080483d4 <add>:
 80483d4:    55          push   %ebp     R[esp]←R[esp]-4; M[R[esp]]←R[ebp]
 80483d5:    89 e5       mov    %esp, %ebp     R[ebp]←R[esp]
 80483d7:    83 ec 10    sub    $0x10, %esp
 80483da:    8b 45 0c    mov    0xc(%ebp), %eax     R[eax]←M[R[ebp]+12]
 80483dd:    8b 55 08    mov    0x8(%ebp), %edx      R[edx]←M[R[ebp]+8]
 80483e0:    8d 04 02    lea    (%edx,%eax,1), %eax     R[eax]←R[edx]+R[eax]
 80483e3:    89 45 fc    mov    %eax, -0x4(%ebp)     M[R[ebp]-4]]←R[eax]
 80483e6:    8b 45 fc    mov    -0x4(%ebp), %eax     R[eax]←M[R[ebp]-4]]
 80483e9:    c9          leave
 80483ea:    c3          ret
```

程序由指令序列组成

```
1   // test.c
2   #include <stdio.h>
3   int add(int i, int j )
4   {
5       int x = i +j;
6       return x;
7   }
```

程序的执行过程如何?

周而复始执行指令!

指令如何执行?

取并执行指令:
- 根据EIP取指令
- 指令译码
- 取操作数
- 指令执行
- 回写结果
- 修改EIP的值

"objdump -d test" 结果

EIP←0x80483d4

```
080483d4 <add>:
 80483d4:   55              push   %ebp
 80483d5:   89 e5           mov    %esp, %ebp
 80483d7:   83 ec 10        sub    $0x10, %esp
 80483da:   8b 45 0c        mov    0xc(%ebp), %eax
 80483dd:   8b 55 08        mov    0x8(%ebp), %edx
 80483e0:   8d 04 02        lea    (%edx,%eax,1), %eax
 80483e3:   89 45 fc        mov    %eax, -0x4(%ebp)
 80483e6:   8b 45 fc        mov    -0x4(%ebp), %eax
 80483e9:   c9              leave
 80483ea:   c3              ret
```

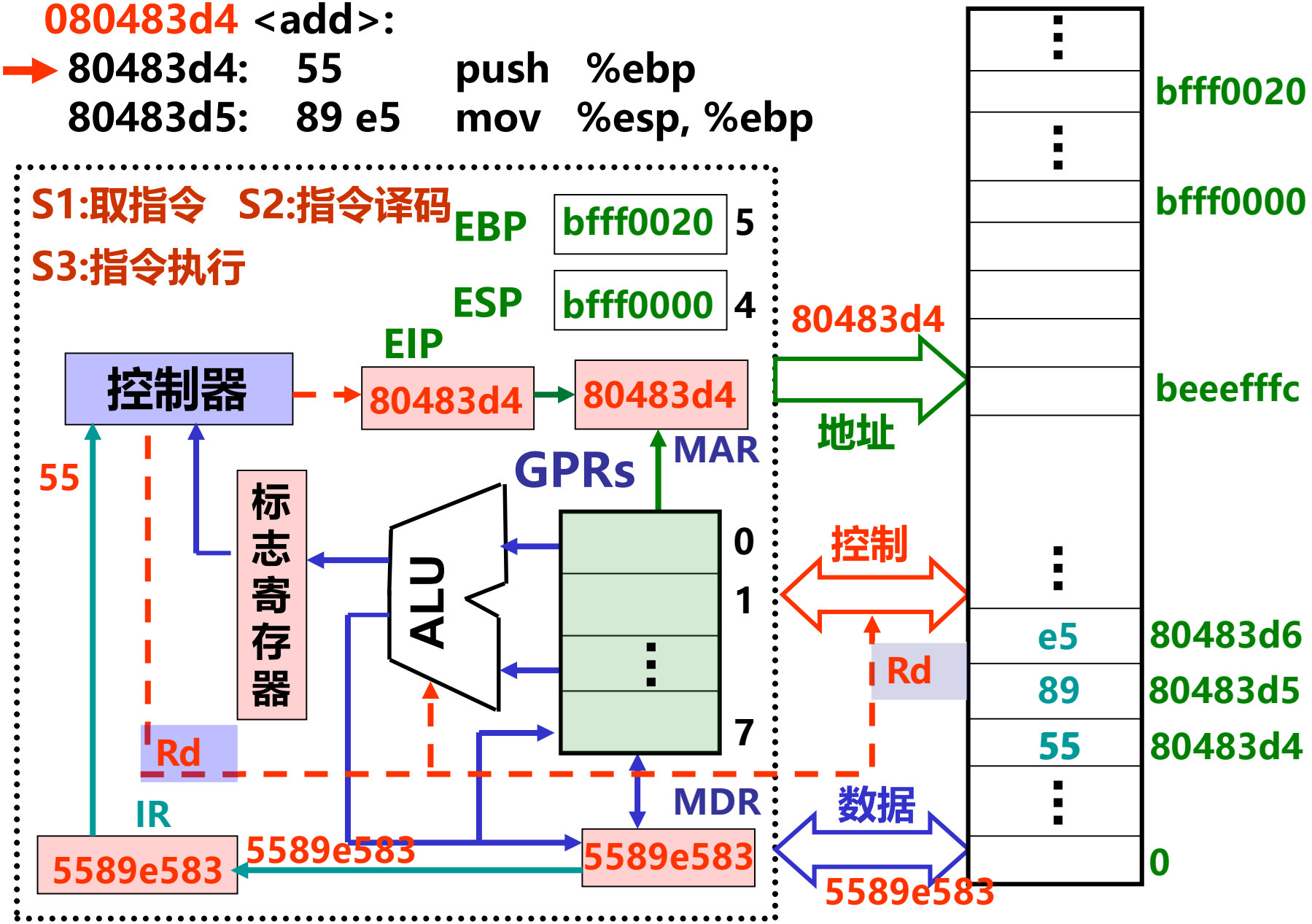
举例

OP

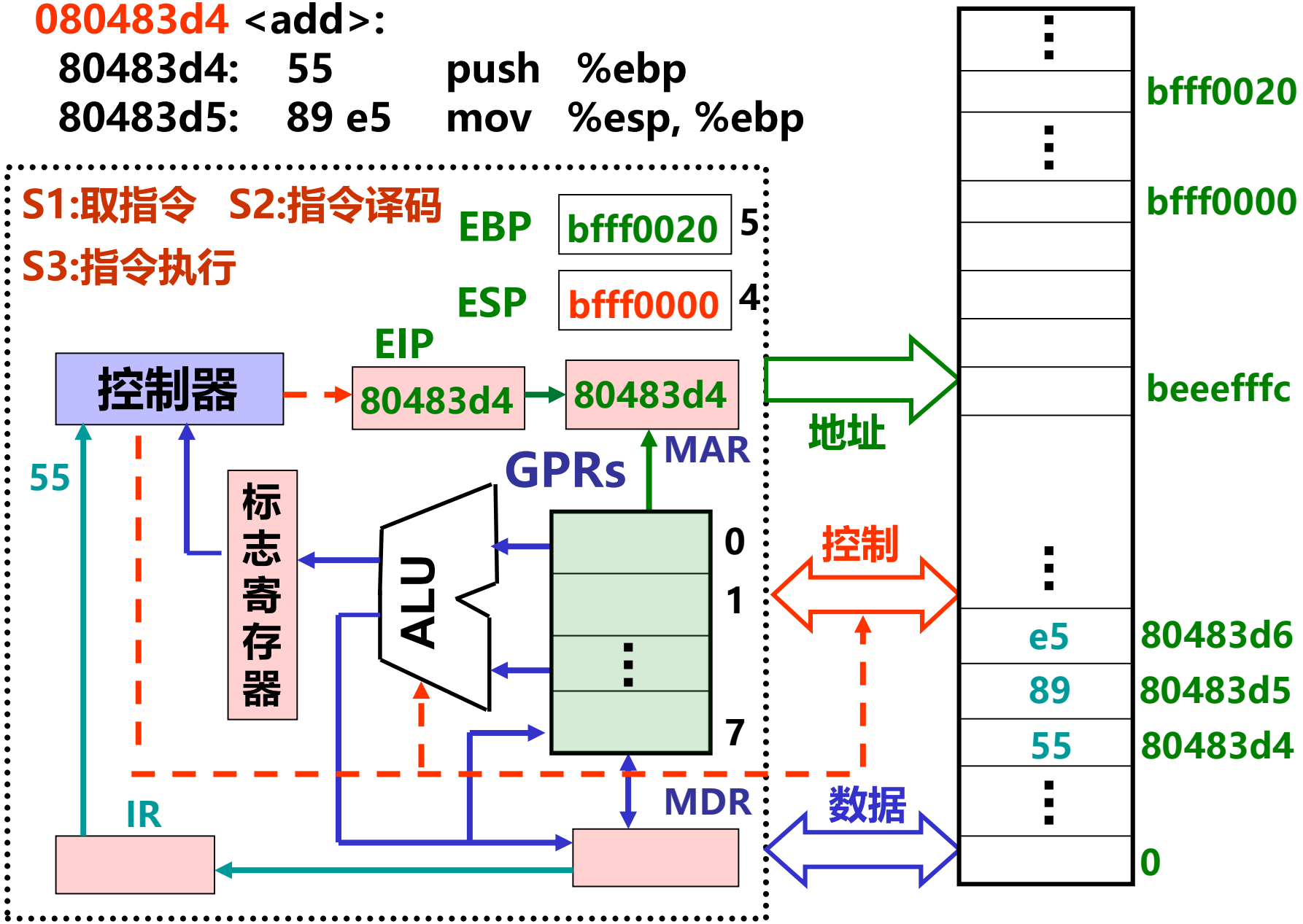
add函数从80483d4开始!

执行add时，起始EIP=?

功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

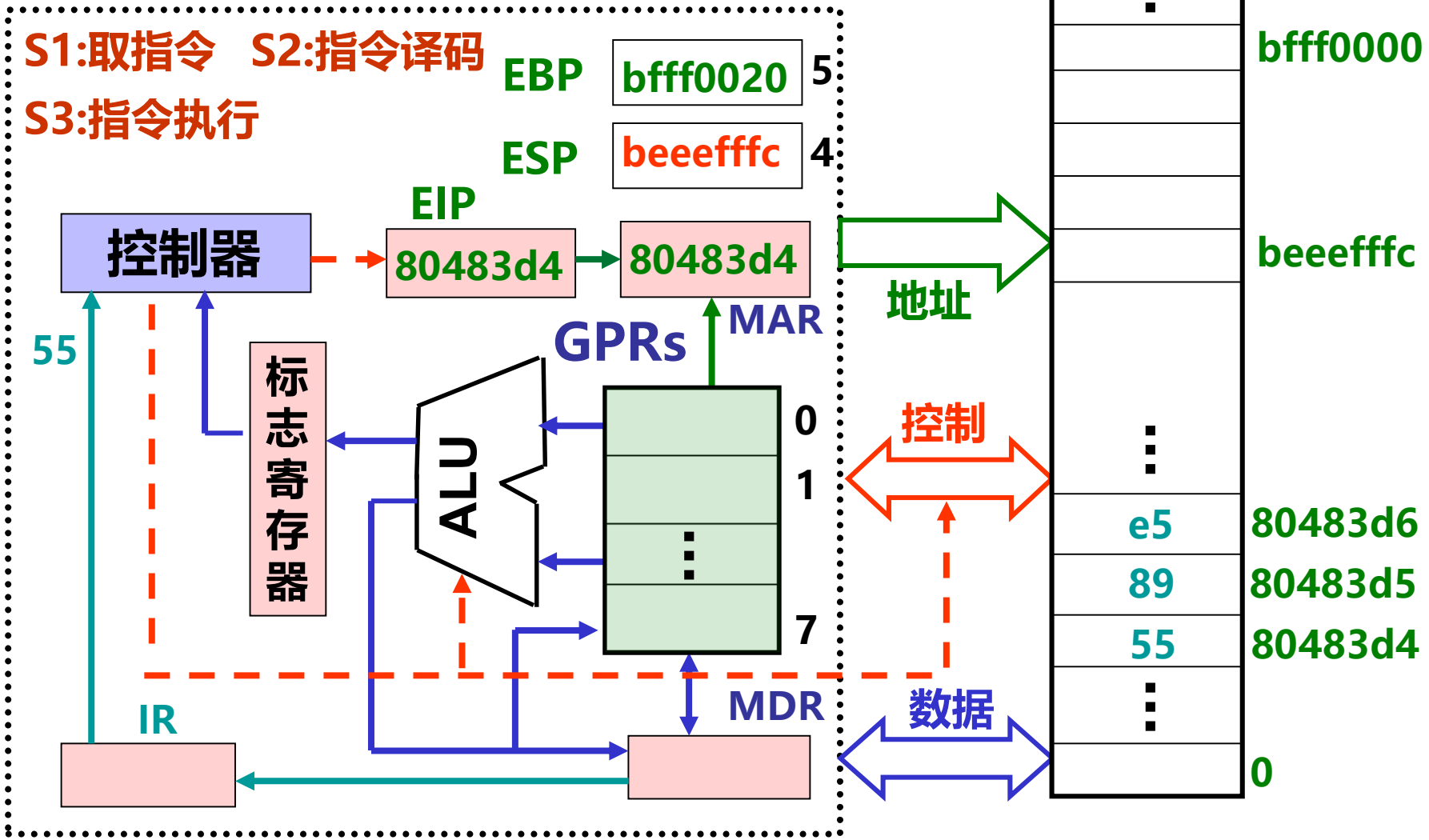


功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

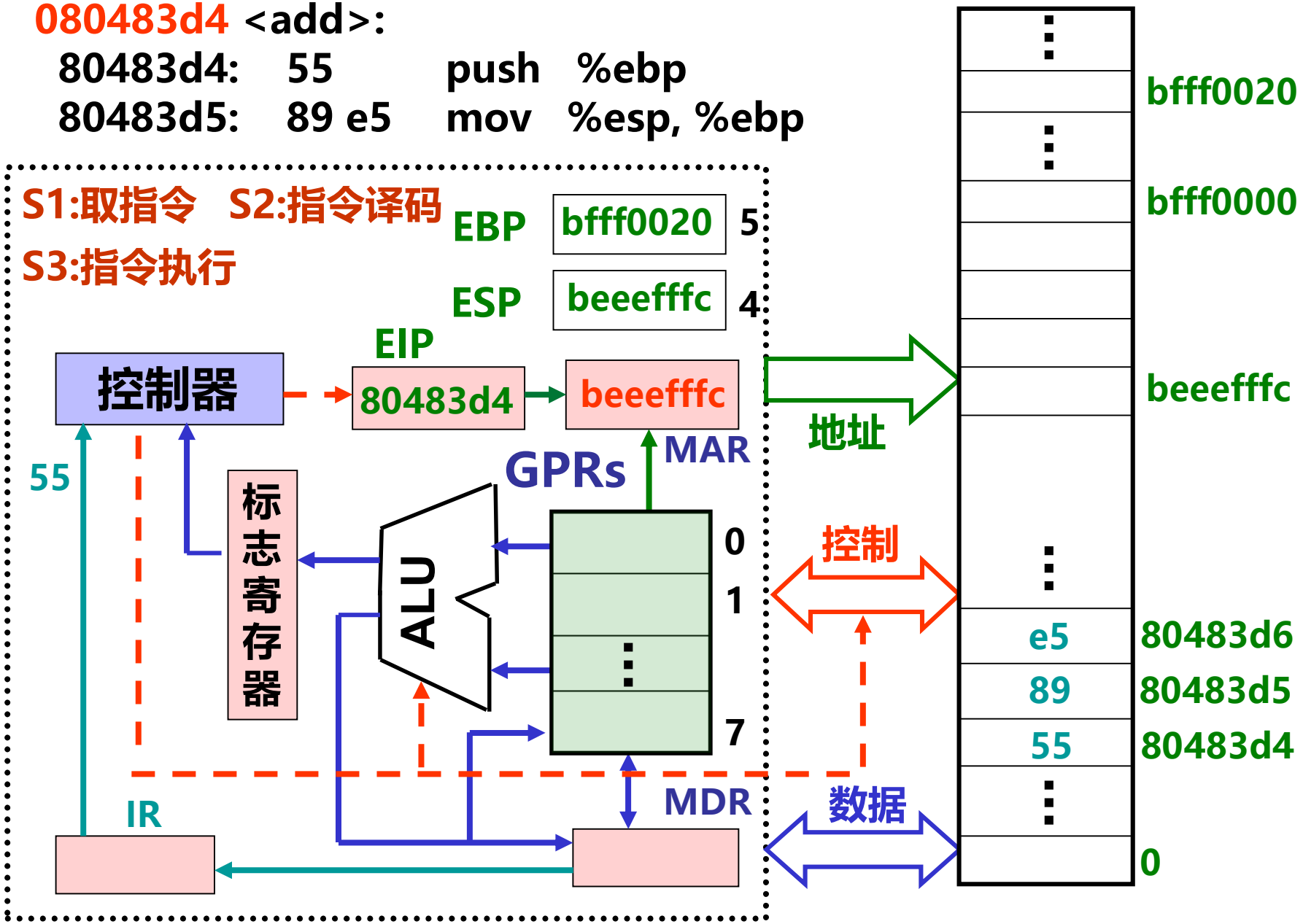


功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

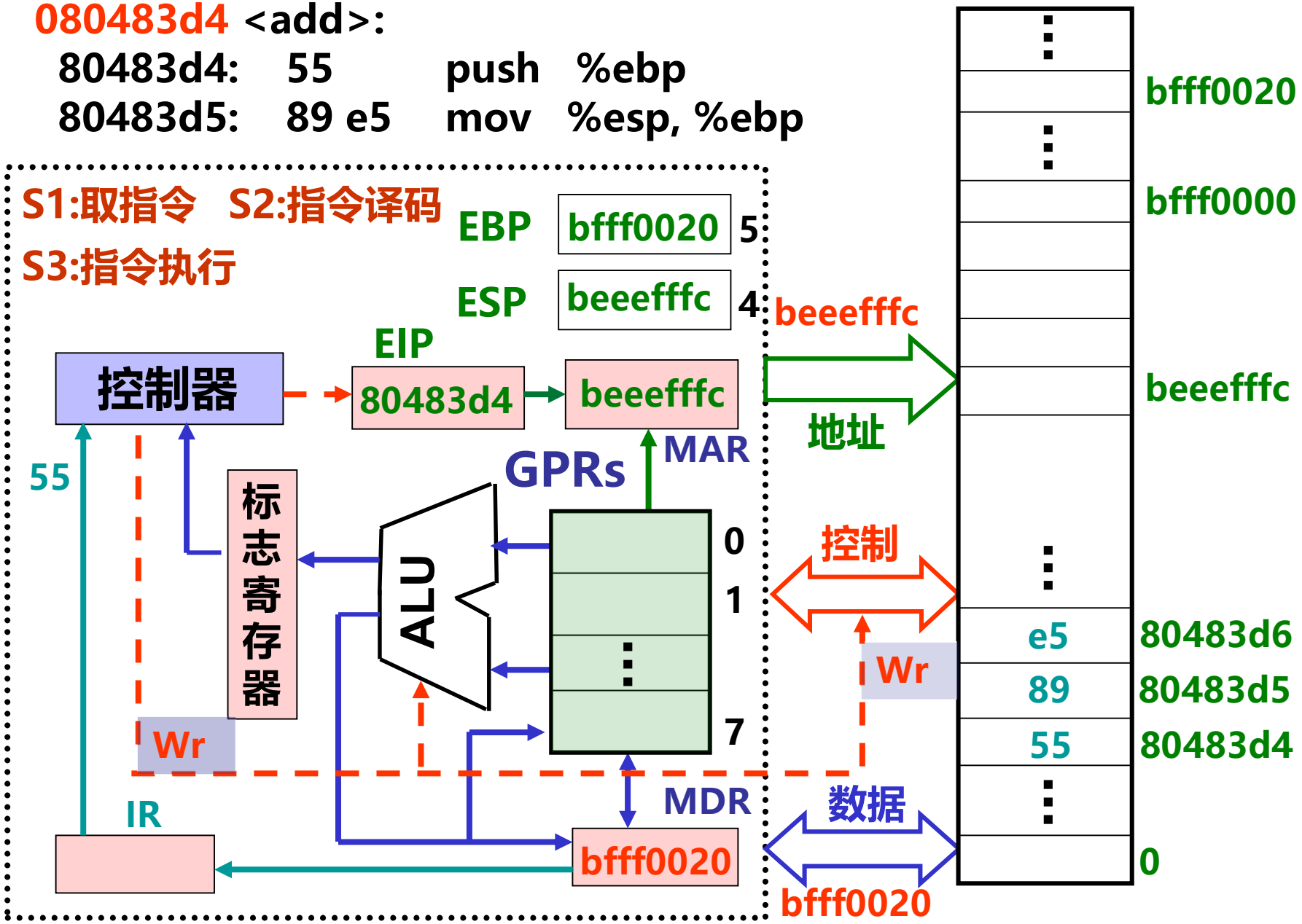
```
080483d4 <add>:
  80483d4:    55          push   %ebp
  80483d5:    89 e5       mov    %esp, %ebp
```



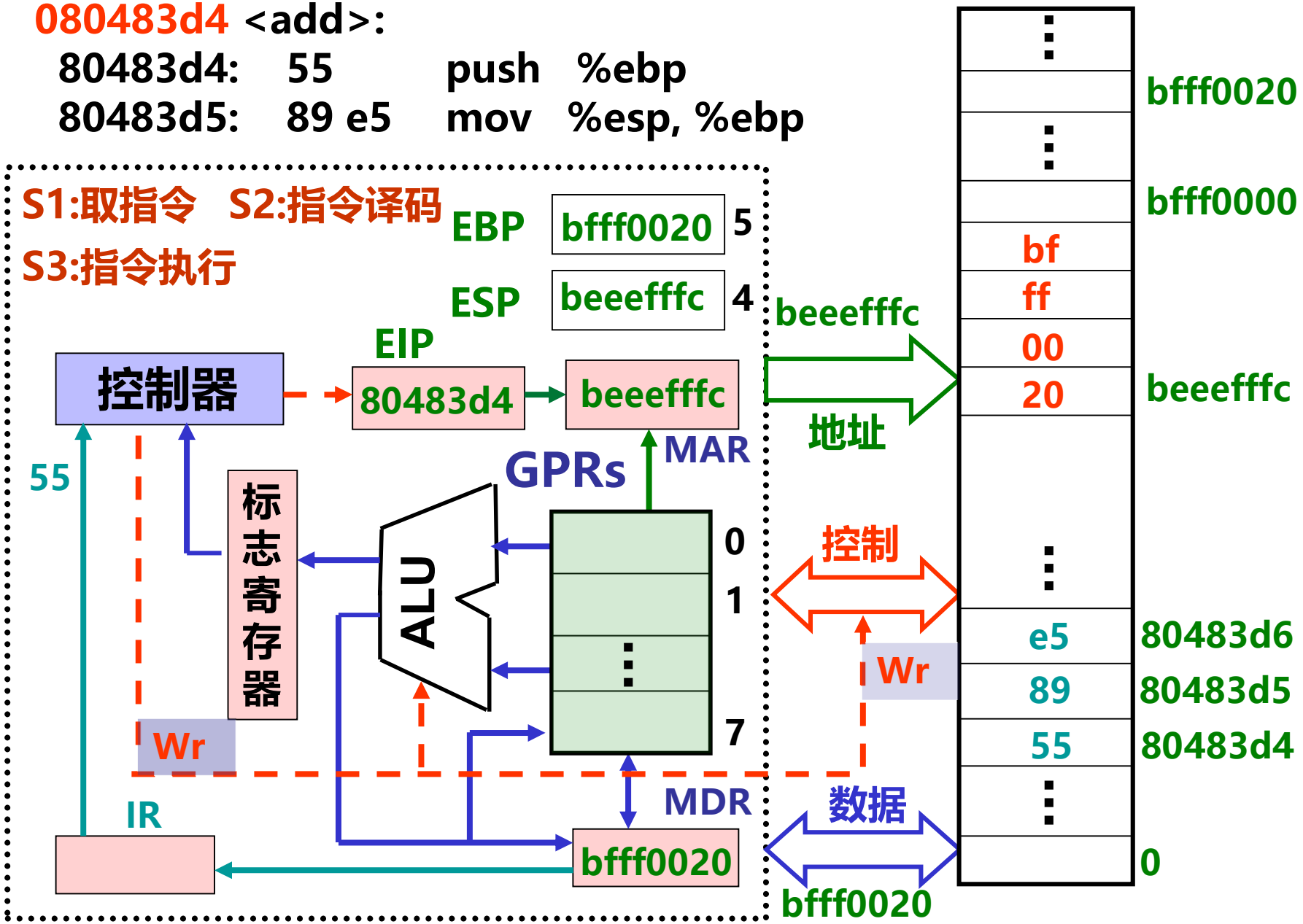
功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]



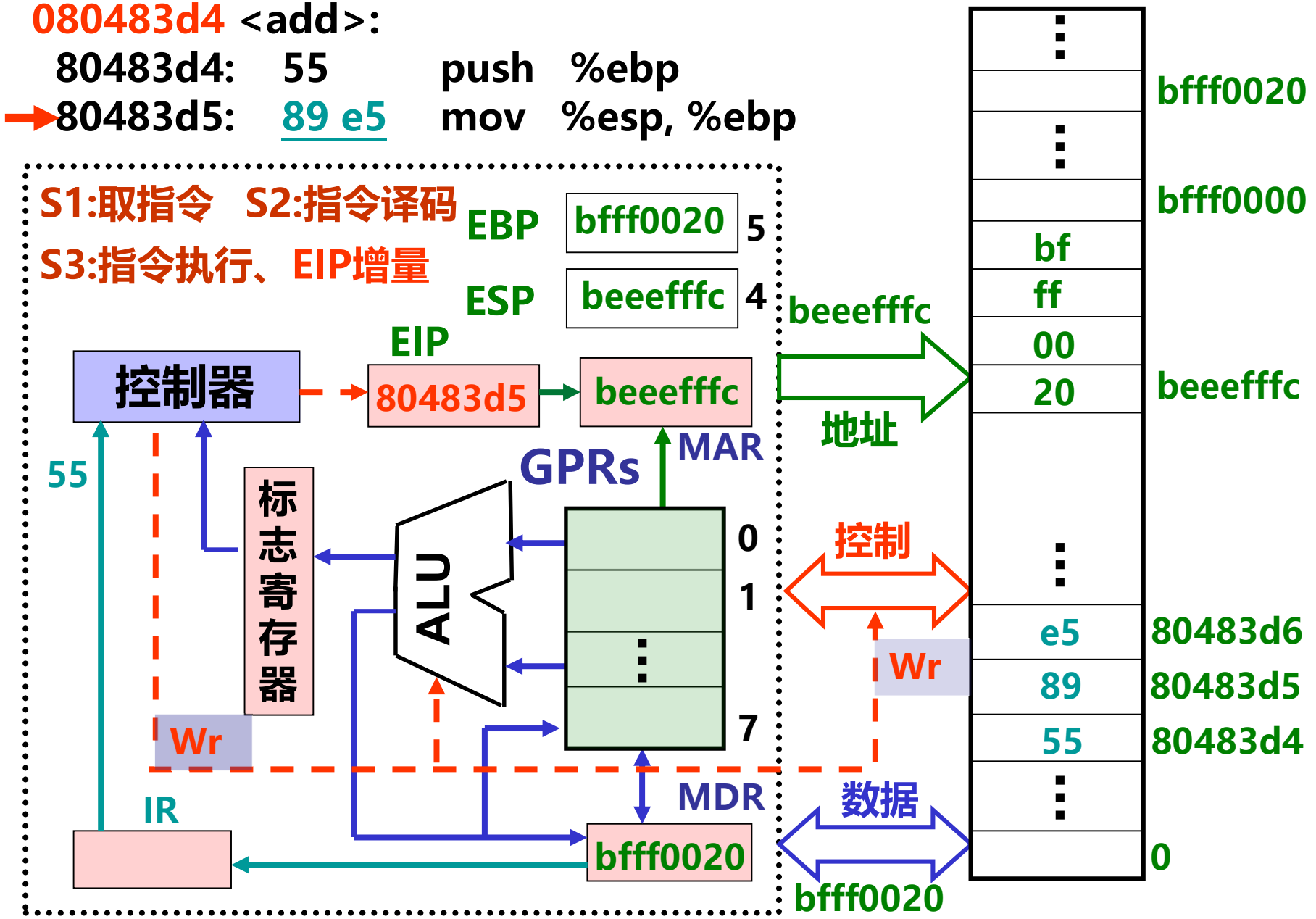
功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]



功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]



开始执行下一条指令


080483d4 <add>:
80483d4: 55 push %ebp
80483d5: 89 e5 mov %esp, %ebp
S1:取指令 S2:指令译码
S3:指令执行、EIP增量
EBP
bfff0020
5
ESP
beeefffc
4
EIP
控制器
80483d5
beeefffc
MAR
GPRs
55
标志寄存器
ALU
0
1
7
Wr
IR
MDR
bfff0020
beeefffc
地址
控制
Wr
数据
bfff0020
bfff0020
bf
ff
00
20
beeefffc
e5
89
55
80483d6
80483d5
80483d4
0

传送指令举例

将以下 Intel格式 指令转换为 AT&T格式 指令，并说明功能。

```
push    ebp
mov     ebp, esp
mov     edx, DWORD PTR [ebp+8]
mov     bl, 255
mov     ax, WORD PTR [ebp+edx*4+8]
mov     WORD PTR [ebp+20], dx
lea     eax, [ecx+edx*4+8]
```

```
pushl   %ebp                    //R[esp]←R[esp]-4，M[R[esp]] ←R[ebp]，双字
movl    %esp, %ebp              //R[ebp] ←R[esp]，双字
movl    8(%ebp), %edx           //R[edx] ←M[R[ebp]+8]，双字
movb    $255, %bl               //R[bl]←255，字节
movw    8(%ebp,%edx,4), %ax     //R[ax]←M[R[ebp]+R[edx]×4+8]，字
movw    %dx, 20(%ebp)           //M[R[ebp]+20]←R[dx]，字
leal    8(%ecx,%edx,4), %eax    //R[eax]←R[ecx]+R[edx]×4+8，双字
```

IA-32中的定点算术运算指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

IA-32常用指令类型

（2）定点算术运算指令

- 加 / 减运算（影响标志、不区分无/带符号）
  - ADD：加，包括addb、addw、addl等
  - SUB：减，包括subb、subw、subl等
- 增1 / 减1运算（影响除CF以外的标志、不区分无/带符号）
  - INC：加，包括incb、incw、incl等
  - DEC：减，包括decb、decw、decl等
- 取负运算（影响标志、若对0取负，则结果为0且CF清0，否则CF置1）
  - NEG：取负，包括negb、negw、negl等
- 比较运算（做减法得到标志、不区分无/带符号）
  - CMP：比较，包括cmpb、cmpw、cmpl等
- 乘 / 除运算（不影响标志、区分无/带符号）
  - MUL / IMUL：无符号乘 / 带符号乘
  - DIV/ IDIV：带无符号除 / 带符号除

整数乘除指令

- 乘法指令：可给出一个、两个或三个操作数
  - 若给出一个操作数SRC，则另一个源操作数隐含在AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。DX-AX表示32位乘积的高、低16位分别在DX和AX中。 n位× n位=2n位
  - 若指令中给出两个操作数DST和SRC，则将DST和SRC相乘，结果在DST中。n位× n位=n位
  - 若指令中给出三个操作数REG、SRC和IMM，则将SRC和立即数IMM相乘，结果在REG中。n位× n位=n位
- 除法指令：只明显指出除数，用EDX-EAX中内容除以指定的除数
  - 若为8位，则16位被除数在AX寄存器中，商送回AL，余数在AH
  - 若为16位，则32位被除数在DX-AX寄存器中，商送回AX，余数在DX
  - 若为32位，则被除数在EDX-EAX寄存器中，商送EAX，余数在EDX

定点算术运算指令汇总

指令	显式操作数	影响的常用标志	操作数类型	AT&T 指令助记符	对应 C 运算符
ADD	2 个	OF、ZF、SF、CF	无/带符号整数	addb、addw、addl	+
SUB	2 个	OF、ZF、SF、CF	无/带符号整数	subb、subw、subl	−
INC	1 个	OF、ZF、SF	无/带符号整数	incb、incw、incl	++
DEC	1 个	OF、ZF、SF	无/带符号整数	decb、decw、decl	−−
NEG	1 个	OF、ZF、SF、CF	无/带符号整数	negb、negw、negl	−
CMP	2 个	OF、ZF、SF、CF	无/带符号整数	cmpb、cmpw、cmpl	<, <=, >, >=
MUL	1 个	OF、CF	无符号整数	mulb、mulw、mull	*
MUL	2 个		无符号整数	mulb、mulw、mull	*
MUL	3 个		无符号整数	mulb、mulw、mull	*
IMUL	1 个		带符号整数	imulb、imulw、imull	*
IMUL	2 个		带符号整数	imulb、imulw、imull	*
IMUL	3 个		带符号整数	imulb、imulw、imull	*
DIV	1 个	无	无符号整数	divb、divw、divl	/，%
IDIV	1 个	无	带符号整数	idivb、idivw、idivl	/，%

程序由指令序列组成

```
1   // test.c
2   #include <stdio.h>
3   int add(int i, int j )
4   {
5       int x = i +j;
6       return x:
7   }
```

若 i= 2147483647，j=2，则执行结果是什么？

```
int main ( ) {
    int  t1 = 2147483647;
    int t2 = 2;
    int  sum = add (t1, t2);
    printf( "sum=%d" ;sum);
}
```

"objdump -d test" 结果

```
080483d4 <add>:    EIP←0x80483d4
 80483d4:    55          push   %ebp
 80483d5:    89 e5       mov    %esp, %ebp
 80483d7:    83 ec 10    sub    $0x10, %esp
 80483da:    8b 45 0c    mov    0xc(%ebp), %eax
 80483dd:    8b 55 08    mov    0x8(%ebp), %edx
 80483e0:    8d 04 02    lea    (%edx,%eax,1), %eax
 80483e3:    89 45 fc    mov    %eax, -0x4(%ebp)
 80483e6:    8b 45 fc    mov    -0x4(%ebp), %eax
 80483e9:    c9          leave
 80483ea:    c3          ret
```

此时，R[eax]=0x2

R[edx]=0x7fffffff

= add %edx , %eax

$2147483647=2^{31}-1$
$=011\cdots1B=0x7fffffff$

add函数从80483d4开始！

执行add时，起始EIP=?

IA-32的寄存器组织

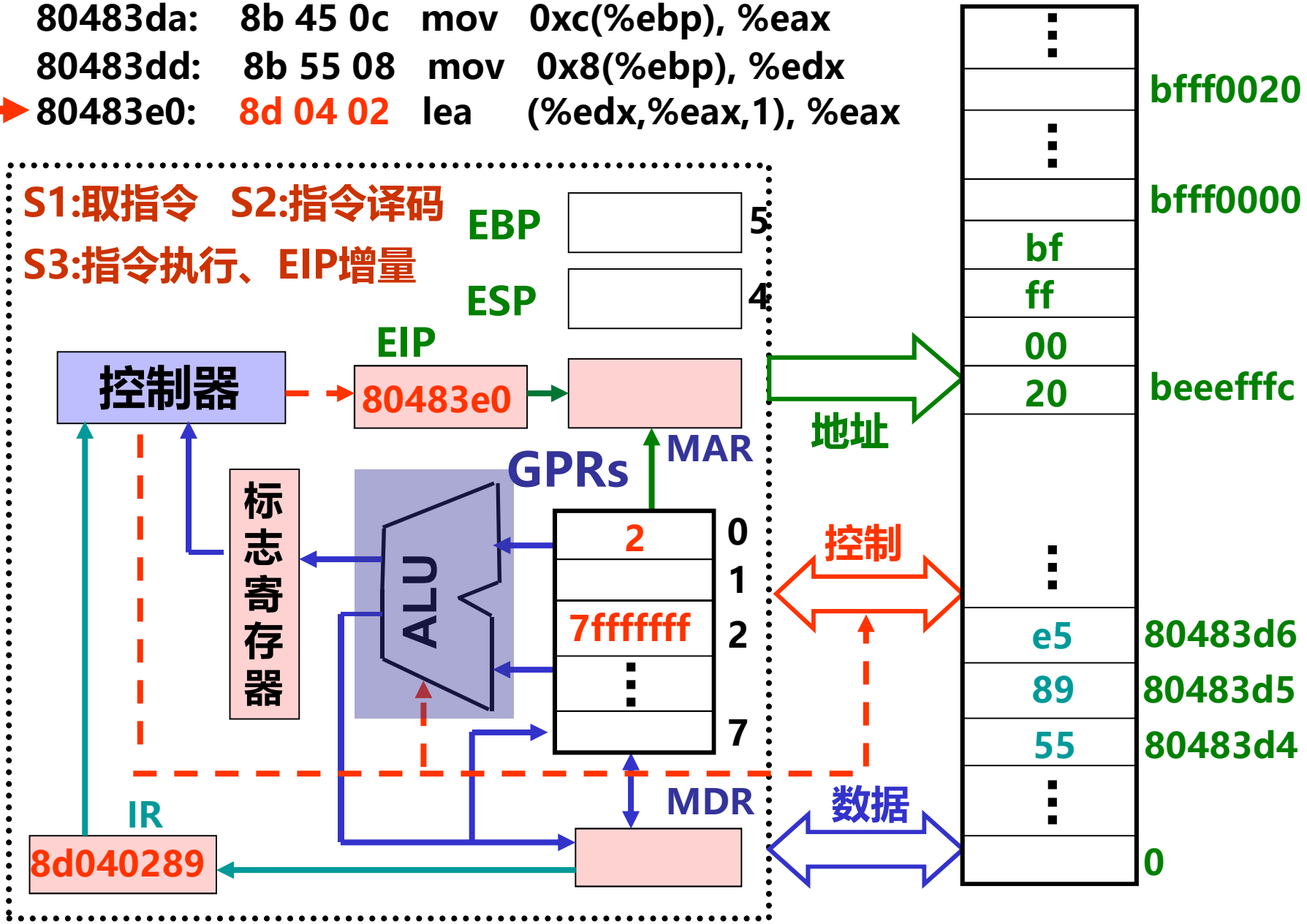
编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

此时，R[eax]=0x2， R[edx]=0x7fffffff

即：0号寄存器中为0x2；2号寄存器中为0x7fffffff

功能：R[eax]← R[edx]+R[eax]*1

```
80483da:   8b 45 0c   mov   0xc(%ebp), %eax
80483dd:   8b 55 08   mov   0x8(%ebp), %edx
80483e0:   8d 04 02   lea   (%edx,%eax,1), %eax
```



ALU长啥样呢？

- 试想一下ALU中有哪些部件？（想想厨房做菜用什么工具？）
  - 补码加/减器（可以干什么？）
    - 带符号整数加、减
    - 无符号整数加、减
  - ~~乘法器~~？（为什么可以没有？）
    - 可用加/减+移位实现，也可有独立乘法器
    - 带符号乘和无符号乘是独立的部件
  - ~~除法器~~？（为什么可以没有？）
    - 可用加/减+移位实现，也可有独立除法器
    - 带符号除和无符号除是独立的部件
  - 各种逻辑运算部件（可以干什么？）
    - 非、与、或、非、前置0个数、前置1个数……

ALU如何实现呢？

ALU结构原理

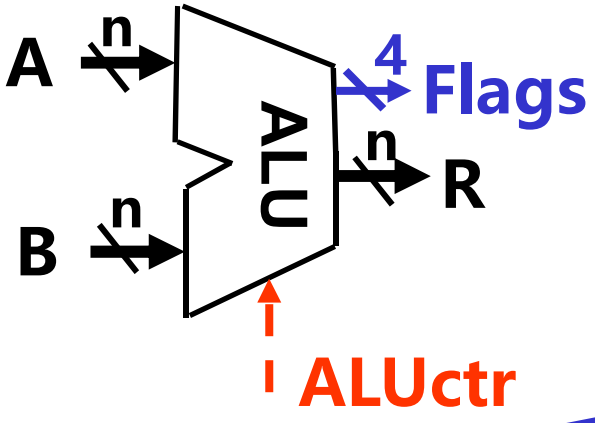
在ALU中执行：

R[eax]=0x2

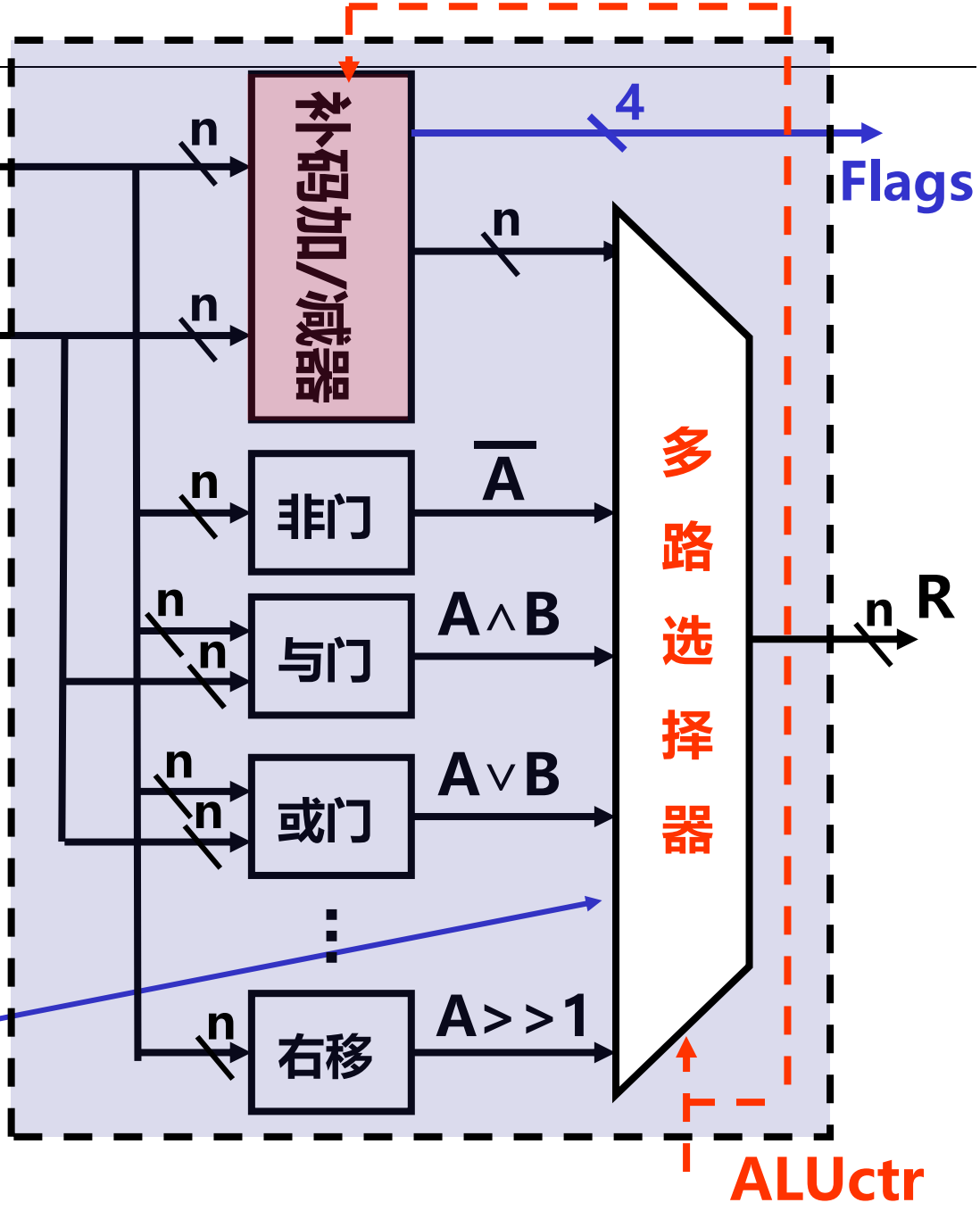
R[edx]=0x7fffffff

R[eax]+R[edx]=?

ALU的符号是什么样的？

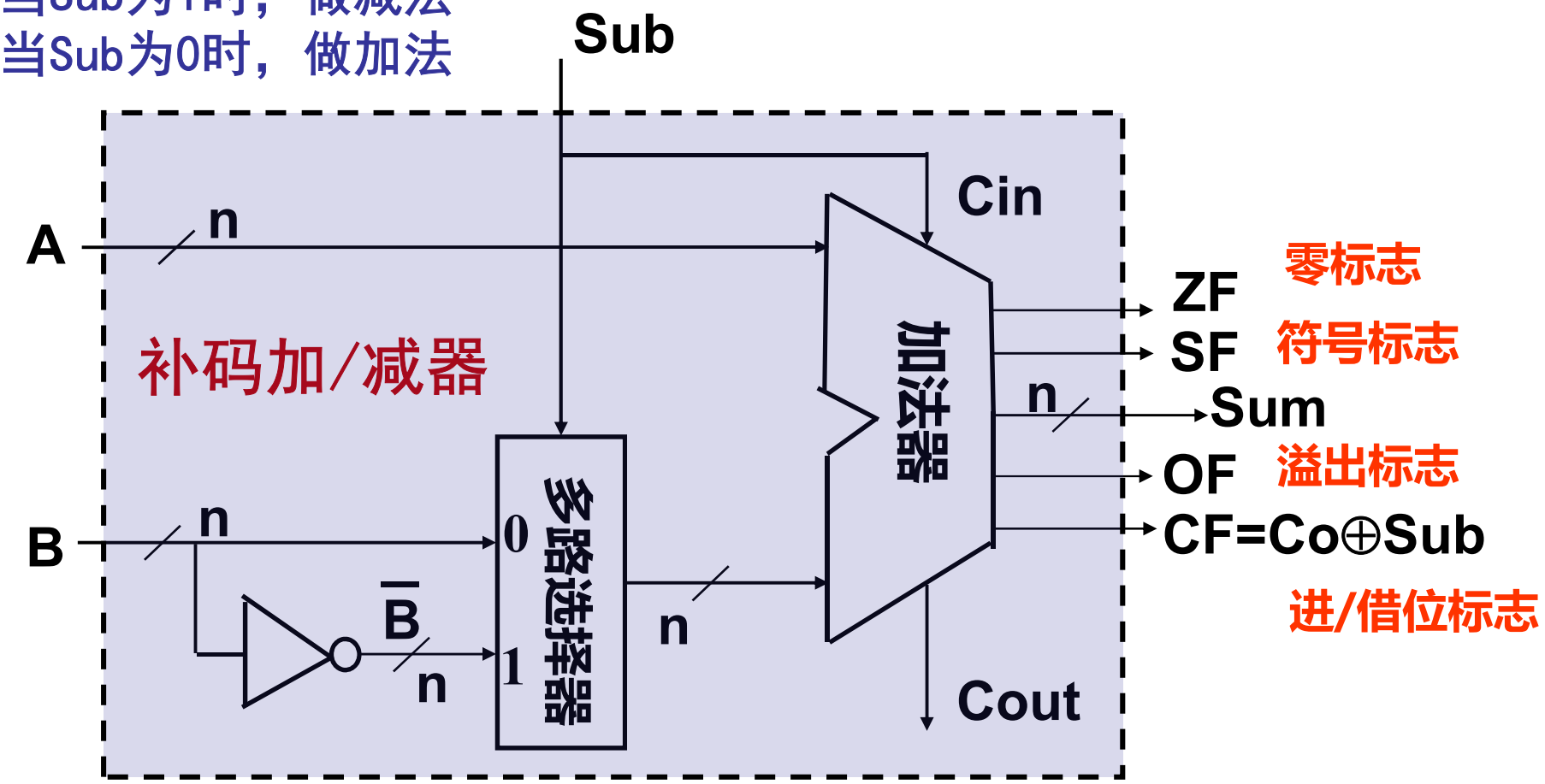


猜猜这是什么？



回顾：补码加/减器

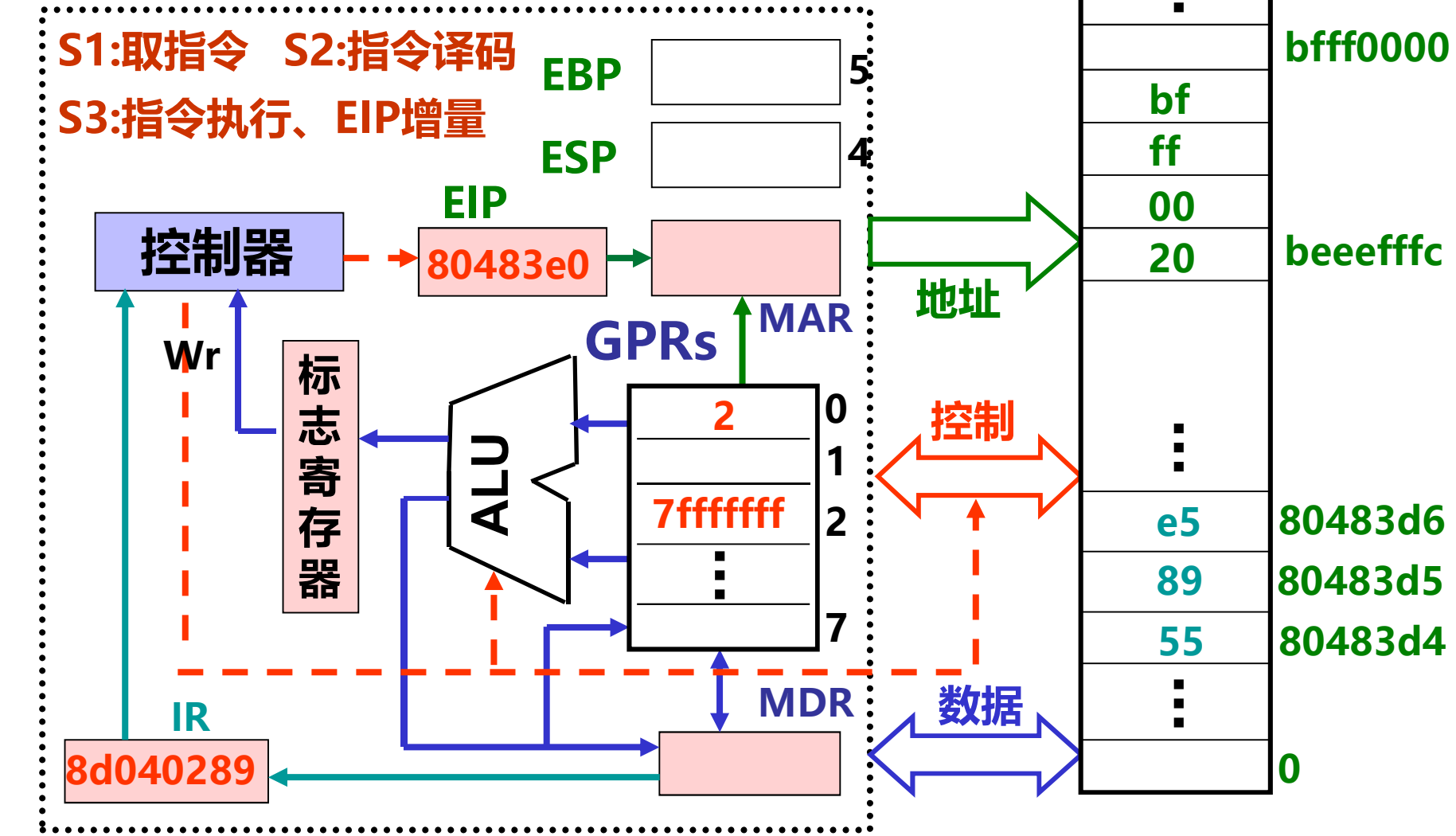
当Sub为1时，做减法
当Sub为0时，做加法



A : 0000 0000 0000 0000 0000 0000 0000 0010
B : 0111 1111 1111 1111 1111 1111 1111 1111
sum : 1000 0000 0000 0000 0000 0000 0000 0001

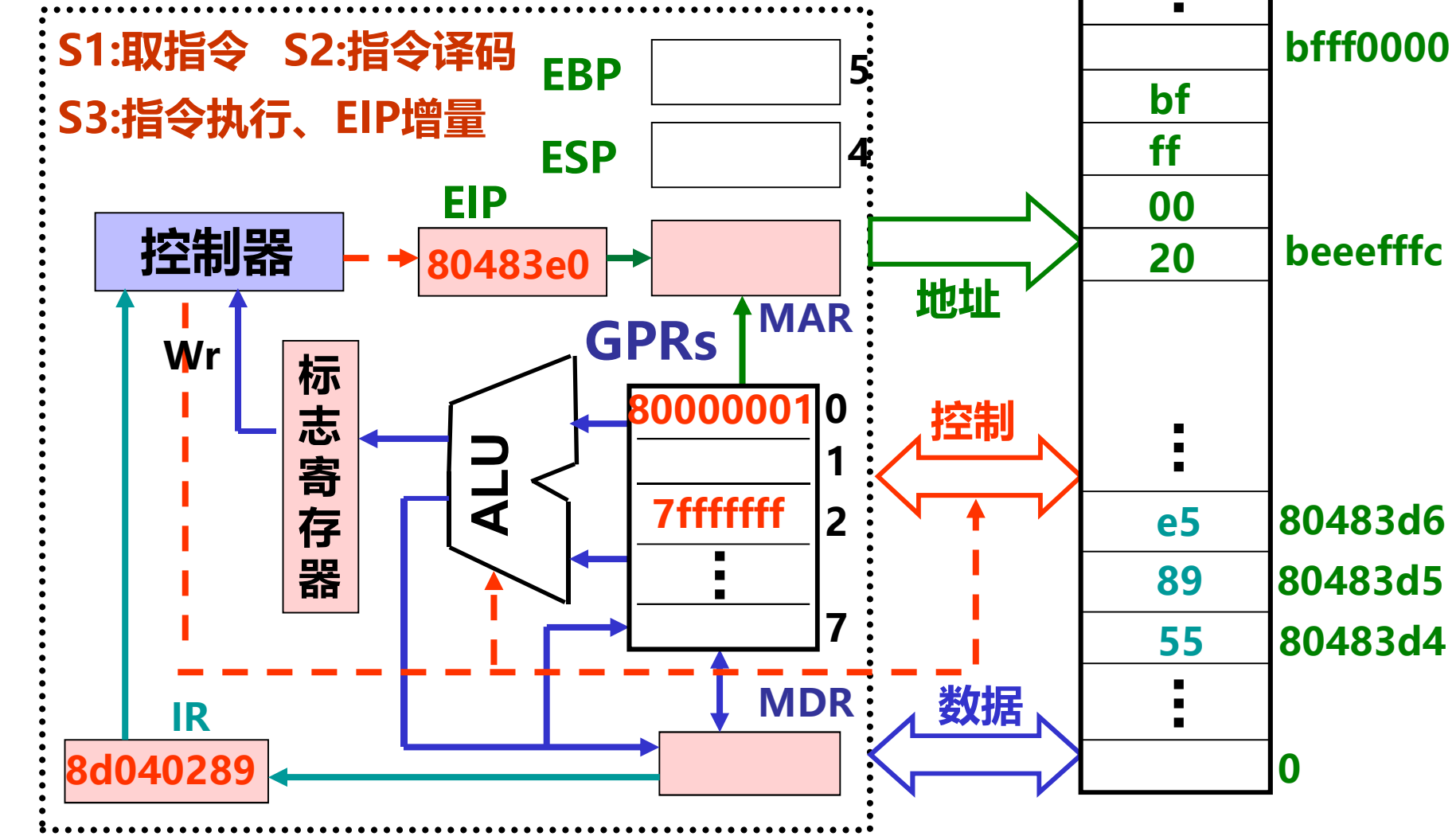
功能：R[eax]← R[edx]+R[eax]*1（执行前）

```
80483da:   8b 45 0c   mov   0xc(%ebp), %eax
80483dd:   8b 55 08   mov   0x8(%ebp), %edx
80483e0:   8d 04 02   lea   (%edx,%eax,1), %eax
```



功能：R[eax]← R[edx]+R[eax]*1 （执行后）

```
80483da:   8b 45 0c   mov   0xc(%ebp), %eax
80483dd:   8b 55 08   mov   0x8(%ebp), %edx
80483e0:   8d 04 02   lea   (%edx,%eax,1), %eax
```



程序执行的结果

```
int add ( int i, int j ) {
    return i+j;
}

int main ( ) {
    int t1 = 2147483647;
     int t2 = 2;
    int sum = add (t1, t2);
    printf(" sum=%d" , sum);
}
```

sum的机器数和
值分别是什么？

sum=0x80000001

sum=-2147483647

两个正数相加结果怎么为负数呢？

因为计算机是一种模运算系统！

高位有效数字被丢失，即发生了溢出

定点加法指令举例

- 假设 R[ax]=FFFAH，R[bx]=FFF0H，则执行以下指令后

“addw %bx, %ax”

AX、BX中的内容各是什么？标志CF、OF、ZF、SF各是什么？要求分别将操作数作为无符号数和带符号整数解释并验证指令执行结果。

解：功能：R[ax]←R[ax]+R[bx]，指令执行后的结果如下

R[ax]=FFFAH+FFF0H=FFEAH ，BX中内容不变

CF=1，OF=0，ZF=0，SF=1

若是无符号整数运算，则CF=1说明结果溢出

验证：FFFA的真值为65535-5=65530，FFF0的真值为65515

FFEA的真值为65535-21=65514≠65530+65515，即溢出

若是带符号整数运算，则OF=0说明结果没有溢出

验证：FFFA的真值为-6，FFF0的真值为-16

FFEA的真值为-22=-6+(-16)，结果正确，无溢出

定点乘法指令举例

- 假设R[eax]=000000B4H，R[ebx]=00000011H，M[000000F8H]=000000A0H，请问：

  (1) 执行指令“mulb %bl”后，哪些寄存器的内容会发生变化？是否与执行“imulb %bl”指令所发生的变化一样？为什么？请用该例给出的数据验证你的结论。

解：“mulb %bl”功能为 R[ax]←R[al]×R[bl]，执行结果如下

R[ax]=B4H × 11H（无符号整数180和17相乘）

R[ax]=0BF4H，真值为3060=180 × 17

无符号乘：

```
      1011 0100
   x  0001 0001
-----------------
      1011 0100
 1011 0100
-----------------
0000 1011 1111 0100
```

AH=？ AL=？

“imulb %bl”功能为 R[ax]←R[al]×R[bl]

R[ax]=B4H × 11H（带符号整数-76和17相乘）

若R[ax]=0BF4H，则真值为3060≠-76 × 17

R[al]=F4H, R[ah]=? AH中的变化不一样！

R[ax]=FAF4H, 真值为-1292=-76 × 17

对于带符号乘，若积只取低n位，则和无符号相同；若取2n位，则采用“布斯”乘法

定点乘法指令举例

- **布斯乘法：** **“imulb %bl”**

R[ax]=B4H × 11H

```
                  1 0 1 1  0 1 0 0
              x   0 0 1-1  0 0 1-1     0001 0001
-------------------------------------------------
  0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0
  1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0
  0 0 0 0 0 1 0 0 1 1 0 0
  1 1 1 1 0 1 1 0 1 0 0
-------------------------------------------------
  1 1 1 1 1 0 1 0 1 1 1 1 0 1 0 0
  ---------------- ----------------
      AH= ?            AL= ?
```

R[ax]=FAF4H, 真值为-1292=-76 × 17

定点乘法指令举例

- 假设R[eax]=000000B4H，R[ebx]=00000011H，M[000000F8H]=000000A0H，请问：

(2) 执行指令“imull $-16, (%eax,%ebx,4), %eax”后哪些寄存器和存储单元发生了变化？乘积的机器数和真值各是多少？

解：“imull -16, (%eax,%ebx,4),%eax”

功能为 R[eax]←(-16)×M[R[eax]+R[ebx]×4] ，执行结果如下

R[eax]+R[ebx]×4=000000B4H+00000011H<<2=000000F8H

R[eax]=(-16)×M[000000F8H]

=(-16)× 000000A0H（带符号整数乘）

=16 ×（-000000A0H）

=FFFFFF60H<<4

=FFFFF600H

EAX中的真值为-2560

整数乘指令

- **乘法指令：可给出一个、两个或三个操作数**
  - **若给出一个操作数SRC，则另一个源操作数隐含在AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。DX-AX表示32位乘积的高、低16位分别在DX和AX中。**
  BACK
  - **若指令中给出两个操作数DST和SRC，则将DST和SRC相乘，结果在DST中。**
  - **若指令中给出三个操作数REG、SRC和IMM，则将SRC和立即数IMM相乘，结果在REG中。**
  BACK

IA-32中的按位运算指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

开场白

上一节课主要介绍了**IA-32**指令系统中的定点算术运算指令，本节课接着介绍位操作指令。

同样，所有**IA-32**指令的细节内容不需要记忆，只要用到某条指令时，会查手册并理解手册中所描述的内容。

IA-32常用指令类型

（3）按位运算指令

- **逻辑运算**

  NOT：非，包括 notb、notw、notl等

  AND：与，包括 andb、andw、andl等

  OR：或，包括 orb、orw、orl等

  XOR：异或，包括 xorb、xorw、xorl等

  TEST：做“与”操作测试，仅影响标志

仅NOT不影响标志，其他指令OF=CF=0，而ZF和SF则根据结果设置：若全0，则ZF=1；若最高位为1，则SF=1

逻辑运算指令举例

- 假设:

M[0x1000]=00000F89H

M[0x1004]=00001270H

R[eax]=FF000001H

R[ecx]=00001000H

说明以下指令的功能

notw %ax

andl %eax, (%ecx)

orb 4(%ecx), %al

xorw %ax,4(%ecx)

testl %eax, %ecx

指令执行结果如下：

notw %ax

R[ax]=not(0001H)=FFFEH

andl %eax, (%ecx)

M[0x1000]=00000F89H∧FF000001H

=00000001H

orb 4(%ecx), %al

R[al]=01H∨70H=71H

xorw %ax,4(%ecx)

M[0x1004]=1270H⊕0001H

=1271H

testl %eax, %ecx

不改变寄存器和存储单元的内容

因为 00001000H∧FF000001H=0

故 ZF=0

IA-32常用指令类型

（3）按位运算指令

- 移位运算（左/右移时，最高/最低位送CF）

SHL/SHR: 逻辑左/右移，包括 shlb、shrw、shrl等

SAL/SAR: 算术左/右移，左移判溢出，右移高位补符

（移位前、后符号位发生变化，则OF=1 ）

包括 salb、sarw、sarl等

ROL/ROR: 循环左/右移，包括 rolb、rorw、roll等

RCL/RCR: 带进位循环左/右移，即：将CF作为操作数

一部分循环移位，包括 rclb、rcrw、rcll等

RCL:

按位运算指令举例

假设short型变量x被编译器分配在寄存器AX中，R[ax]=FF80H，则以下汇编代码段执行后变量x的机器数和真值分别是多少？

```
movw %ax, %dx
salw  $2, %ax
addl  %dx, %ax
sarw  $1, %ax
```

R[dx]←R[ax]

1111 1111 1000 0000<<2　　算术左移，OF=0

1111 1110 0000 0000+1111 1111 1000 0000

1111 1101 1000 0000>>1=1111 1110 1100 0000

sarw $1,%ax 可简写成 sarw %ax

解：$2和$1分别表示立即数2和1 。

x是short型变量，故都是算术移位指令，并进行带符号整数加。

上述代码段执行前R[ax]=x，则执行((x<<2)+x)>>1后，

R[ax]=5x/2。算术左移时，AX中的内容在移位前、后符号未发生变化，故OF=0，没有溢出。最终AX的内容为FEC0H，解释为short型整数时，其值为-320。验证：x=-128，5x/2=-320。经验证，结果正确。

逆向工程：从汇编指令推断出高级语言程序代码

移位指令举例

```
#include <stdio.h>
void main()
{
    int a = 0x80000000;
    unsigned int b = 0x80000000;
    printf("a=   0x%X\n", a >>1);
    printf("b=   0x%X\n", b >>1);
}
```

```
                                    push   %ebp
                                    mov    %esp,%ebp
                                    and    $0xfffffff0,%esp
                                    sub    $0x20,%esp
00 00 00                            movl   $0x80000000,0x1c(%esp)

                                    movl   $0x80000000,0x18(%esp)
00 00 00

19:   8b 44 24 1c                   mov    0x1c(%esp),%eax
1d:   d1 f8                         sar    %eax                      算术
1f:   89 44 24 04                   mov    %eax,0x4(%esp)
23:   c7 04 24 00 00 00 00          movl   $0x0,(%esp)
2a:   e8 fc ff ff ff                call   2b <main+0x2b>
2f:   8b 44 24 18                   mov    0x18(%esp),%eax
33:   d1 e8                         shr    %eax                      逻辑
35:   89 44 24 04                   mov    %eax,0x4(%esp)
39:   c7 04 24 0b 00 00 00          movl   $0xb,(%esp)
40:   e8 fc ff ff ff                call   41 <main+0x41>
45:   c9                            leave
46:   c3                            ret
```

IA-32中的控制转移指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

IA-32常用指令类型

（4）控制转移指令

指令执行可按顺序 或 跳转到转移目标指令处执行

- 无条件转移指令

  JMP DST：无条件转移到目标指令DST处执行

- 条件转移

  Jcc DST：cc为条件码，根据标志（条件码）判断是否满足条件，若满足，则转移到目标指令DST处执行，否则按顺序执行

- 条件设置

  SETcc DST：按条件码cc判断的结果保存到DST（是一个8位寄存器 ）

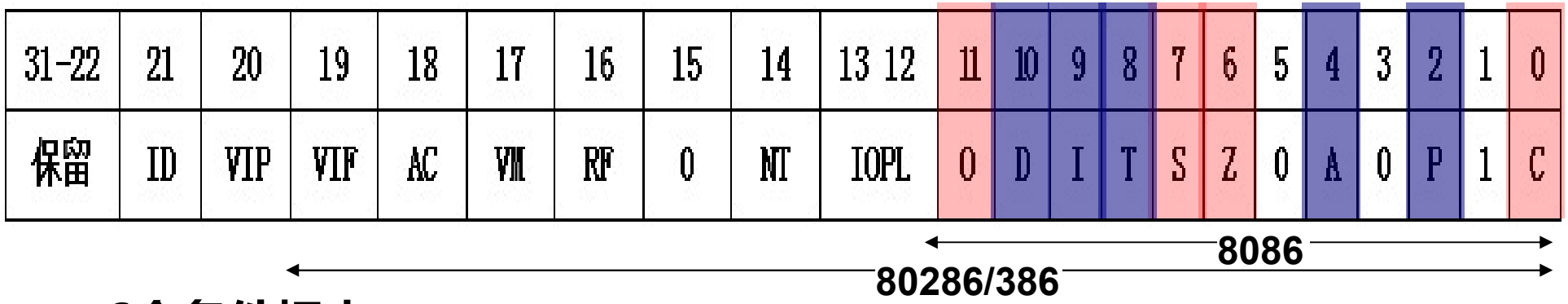
- 调用和返回指令 （用于过程调用）

  CALL DST：返回地址RA入栈，转DST处执行

  RET：从栈中取出返回地址RA，转到RA处执行

- 中断指令 （详见第7、8章）

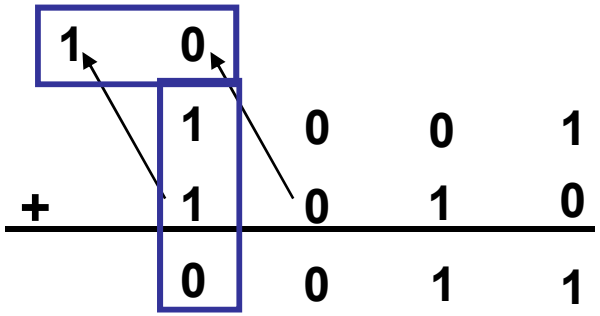
IA-32的标志寄存器



- 6个条件标志
  - OF、SF、ZF、CF各是什么标志（条件码）？
  - AF：辅助进位标志（BCD码运算时才有意义）
  - PF：奇偶标志
- 3个控制标志
  - DF（Direction Flag）：方向标志（自动变址方向是增还是减）
  - IF（Interrupt Flag）：中断允许标志（仅对外部可屏蔽中断有用）
  - TF（Trap Flag）：陷阱标志（是否是单步跟踪状态）
- ……

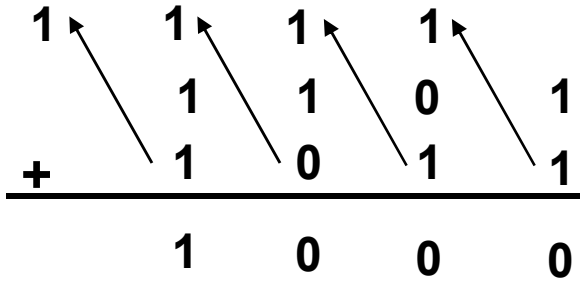
回顾：整数减法举例

-7- 6 = -7 + (-6) = +3 X
9 - 6 = 3 √



OF=1、ZF=0
SF=0、借位CF=0

-3 - 5 = - 3 + (- 5) = - 8 √
13 - 5 = 8 √



OF=0、ZF=0、
SF=1、借位CF=0

可利用条件标志进行大小判断

做减法以比较大小，规则：
Unsigned: CF=0时，大于
Signed：OF=SF时，大于

验证：9>6，故CF=0；13>5，故CF=0

验证：-7<6，故OF≠SF
-3<5，故OF≠SF

分三类：

(1)根据单个标志的值转移

(2)按无符号整数比较转移

(3)按带符号整数比较转移

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 A＞B
10	jae/jnb label	CF=0 OR ZF=1	无符号整数 A≥B
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 A＜B
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 A≤B
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 A＞B
14	jge/jnl label	SF=OF OR ZF=1	带符号整数 A≥B
15	jl/jnge label	SF≠OF AND ZF=	带符号整数 A＜B
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 A≤B

例子：程序的机器级表示与执行

```
int sum(int a[ ], unsigned len)
{
   int  i , sum = 0;
   for (i = 0; i <= len–1; i++)
        sum += a[i];
   return sum;
}
```

```
sum:
     …
.L3:
     …
    movl  -4(%ebp),  %eax
    movl  12(%ebp),  %edx
    subl    $1,  %edx
    cmpl  %edx,  %eax
    jbe     .L3
     …
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生了存储器访问异常。 Why?

i 和 len 分别在哪个寄存器中？

i : %eax ; len : %edx

第一次循环，执行结果是什么？

%eax: 0000 …… 0000

%edx: 0000 …… 0000

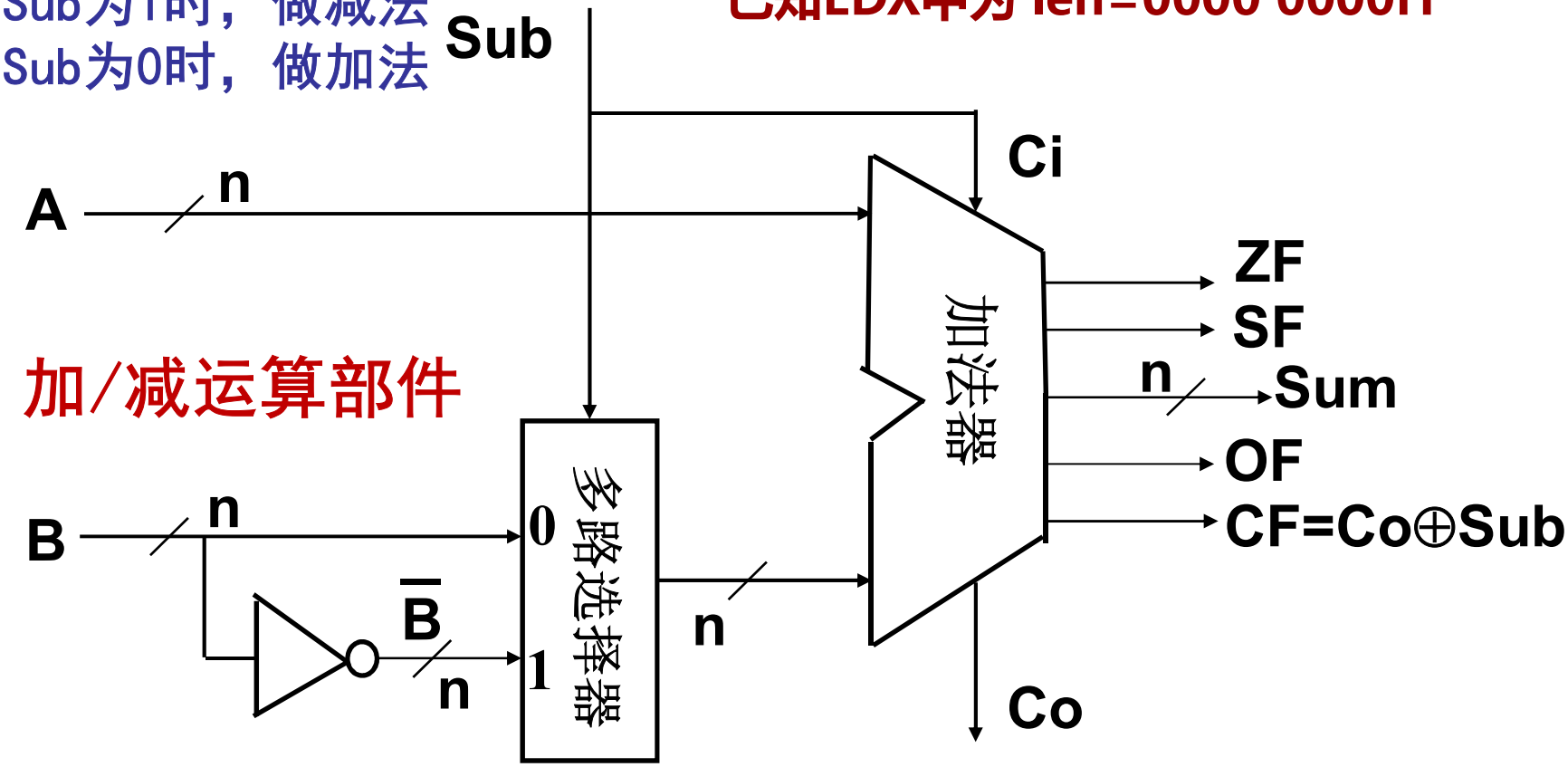
subl 指令的执行结果是什么？

cmpl 指令的执行结果是什么？

subl $1, %edx指令的执行结果

当Sub为1时，做减法
当Sub为0时，做加法

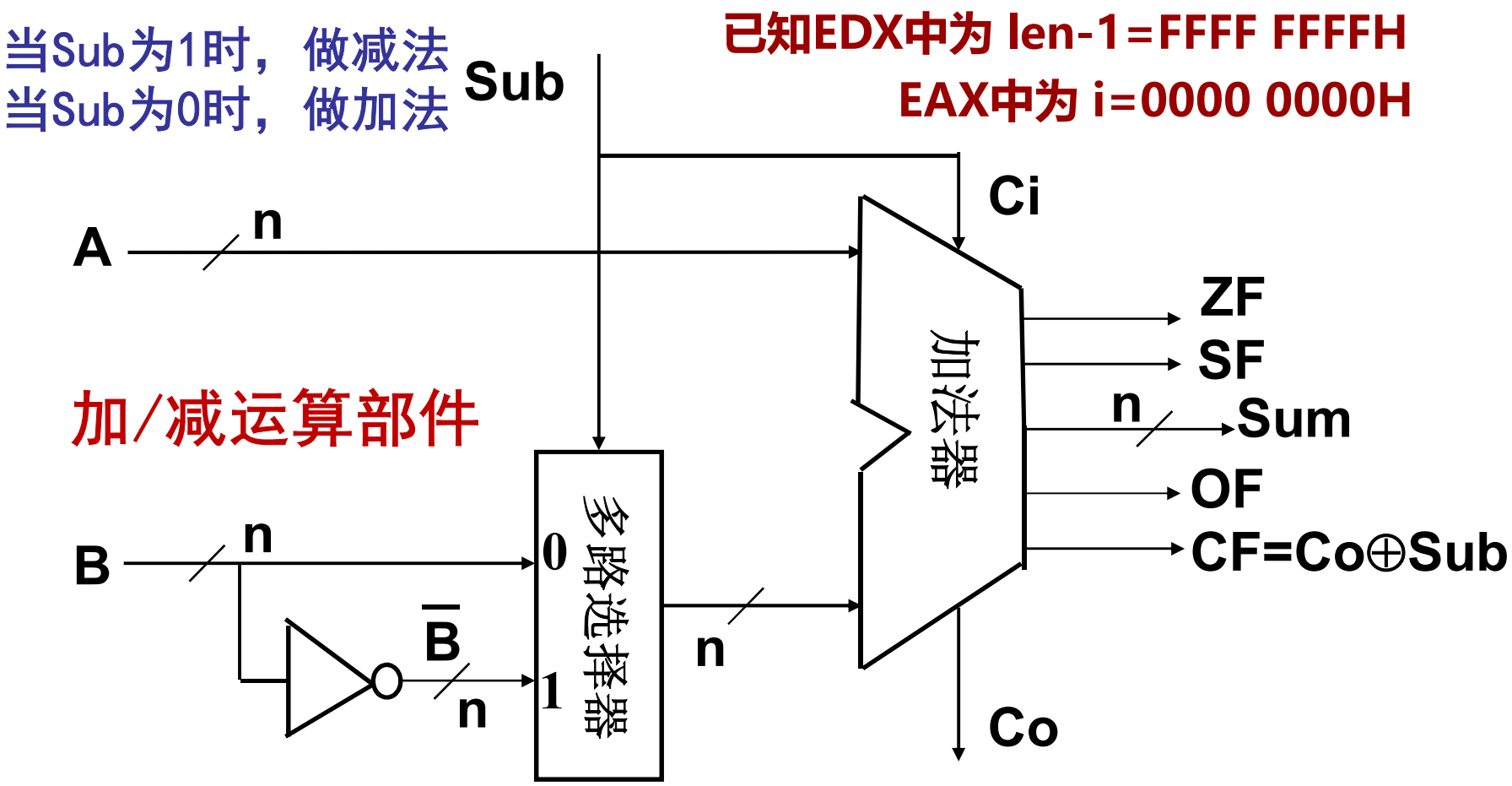
已知EDX中为 len=0000 0000H



“subl $1, %edx”执行时：A=0000 0000H，B为0000 0001H，
Sub=1，因此Sum是32个1，即R[edx]=FFFFFFFFH=0xffffffff

完全等价的两种不同写法！

cpml %edx,%eax指令的执行结果



“cmpl %edx,%eax”执行时：A=0000 0000H，B为FFFF FFFFH，Sub=1，因此Sum是0…01, CF=1, ZF=0, OF=0, SF=0

jbe .L3指令的执行结果

指令	转移条件	说明
JA/JNBE label	CF=0 AND ZF=0	无符号数A > B
JAE/JNB label	CF=0 OR ZF=1	无符号数A≥B
JB/JNAE label	CF=1 AND ZF=0	无符号数A < B
JBE/JNA label	CF=1 OR ZF=1	无符号数A≤B
JG/JNLE label	SF=OF AND ZF=0	带符号整数A > B
JGE/JNL label	SF=OF OR ZF=1	带符号整数A≥B
JL/JNGE label	SF≠OF AND ZF=0	带符号整数A < B
JLE/JNG label	SF≠OF OR ZF=1	带符号整数A≤B

“cmpl %edx,%eax”执行结果是 CF=1, ZF=0, OF=0, SF=0，

因此，在执行“jbe .L3”时满足条件，应转移到.L3执行！

例子：程序的机器级表示与执行

```
int sum(int a[ ], unsigned len)
{
  int  i , sum = 0;
  for (i = 0; i <= len–1; i++)
      sum += a[i];
  return sum;
}
```

```
sum:
    …
.L3:
    …
   movl  -4(%ebp),  %eax
   movl  12(%ebp),  %edx
   subl    $1,  %edx
   cmpl  %edx,  %eax
   jbe     .L3
    …
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生了存储器访问异常。 Why?

“cmpl %edx,%eax”执行结果是 CF=1, ZF=0, OF=0, SF=0，说明满足条件，应转移到.L3执行！ 显然，对于每个 i 都满足条件，因为任何无符号数都比32个1小，因此循环体被不断执行，最终导致数组访问越界而发生存储器访问异常。

例子：程序的机器级表示与执行

例：

```
int sum(int a[ ], int len)
{
   int  i , sum = 0;
   for (i = 0; i <= len–1; i++)
        sum += a[i];
   return sum;
}
```

正确的做法是将参数len声明为int型。 Why?

```
sum:
    …
.L3:
    …
    movl  -4(%ebp),  %eax
    movl  12(%ebp),  %edx
    subl   $1,  %edx
    cmpl  %edx,  %eax
    jle     .L3
    …
```

“sub $1,%edx”和“cmpl %edx,%eax”执行结果与前面一样！

执行到“jle .L3”指令时，也是 CF=1, ZF=0, OF=0, SF=0！

jle .L3指令的执行结果

指令	转移条件	说明
JA/JNBE label	CF=0 AND ZF=0	无符号数A > B
JAE/JNB label	CF=0 OR ZF=1	无符号数A≥B
JB/JNAE label	CF=1 AND ZF=0	无符号数A < B
JBE/JNA label	CF=1 OR ZF=1	无符号数A≤B
JG/JNLE label	SF=OF AND ZF=0	带符号整数A > B
JGE/JNL label	SF=OF OR ZF=1	带符号整数A≥B
JL/JNGE label	SF≠OF AND ZF=0	带符号整数A < B
JLE/JNG label	SF≠OF OR ZF=1	带符号整数A≤B

“cmpl %edx,%eax”执行结果是 CF=1, ZF=0, OF=0, SF=0，因此，在执行“jle .L3”时不满足条件，应跳出循环执行，使得执行结果正常。

例子：C表达式类型转换顺序

unsigned long long

↑

long long

↑

unsigned

↑

int

↑

(unsigned) char

(unsigned) short

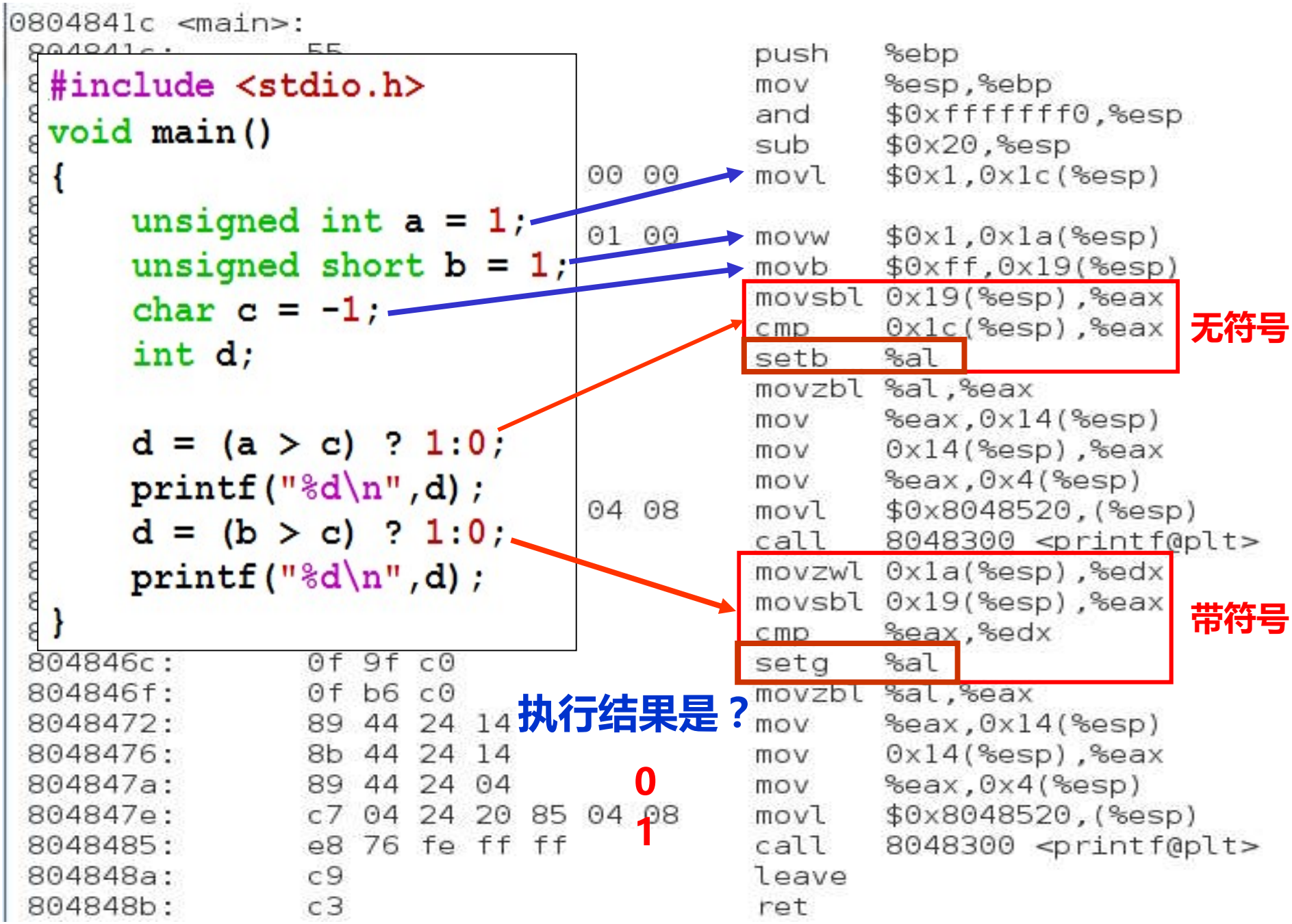
```
#include <stdio.h>
void main()
{
    unsigned int a = 1;
    unsigned short b = 1;
    char c = -1;
    int d;

    d = (a > c) ? 1:0;
    printf("%d\n",d);
    d = (b > c) ? 1:0;
    printf("%d\n",d);
}
```

猜测：各用哪种条件设置指令？

条件设置指令：

SETcc DST：按条件码cc判断的结果保存到DST


0804841c <main>:
#include <stdio.h>
void main()
{
unsigned int a = 1;
unsigned short b = 1;
char c = -1;
int d;
d = (a > c) ? 1:0;
printf("%d\n",d);
d = (b > c) ? 1:0;
printf("%d\n",d);
}
804846c: 0f 9f c0
804846f: 0f b6 c0
8048472: 89 44 24 14
8048476: 8b 44 24 14
804847a: 89 44 24 04
804847e: c7 04 24 20 85 04 08
8048485: e8 76 fe ff ff
804848a: c9
804848b: c3
00 00
01 00
04 08
push %ebp
mov %esp,%ebp
and $0xfffffff0,%esp
sub $0x20,%esp
movl $0x1,0x1c(%esp)
movw $0x1,0x1a(%esp)
movb $0xff,0x19(%esp)
movsbl 0x19(%esp),%eax
cmp 0x1c(%esp),%eax
setb %al
movzbl %al,%eax
mov %eax,0x14(%esp)
mov 0x14(%esp),%eax
mov %eax,0x4(%esp)
movl $0x8048520,(%esp)
call 8048300 <printf@plt>
movzwl 0x1a(%esp),%edx
movsbl 0x19(%esp),%eax
cmp %eax,%edx
setg %al
movzbl %al,%eax
mov %eax,0x14(%esp)
mov 0x14(%esp),%eax
mov %eax,0x4(%esp)
movl $0x8048520,(%esp)
call 8048300 <printf@plt>
leave
ret
无符号
带符号
执行结果是？
0
1

x87浮点处理指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

开场白

前面几节课主要介绍了**IA-32**指令系统中各种类型的指令，包括传送指令、定点算术运算指令、位操作指令和控制转移指令。

本节课主要介绍**x87 FPU** 浮点处理指令。

同样，所有**IA-32**指令的细节内容不需要记忆，只要用到某条指令时，会查手册并理解手册中所描述的内容。

IA-32的浮点处理架构

- IA-32的浮点处理架构有两种

  (1) x87FPU指令集（gcc默认）

  (2) SSE指令集（x86-64架构所用）

- IA-32中处理的浮点数有三种类型
  - float类型：32位 IEEE 754 单精度格式
  - double类型：64位 IEEE 754 双精度格式
  - long double类型：80位双精度扩展格式

    1位符号位s、15位阶码e（偏置常数为16 383）、1位显式首位有效位（explicit leading significant bit）j 和 63位尾数f。它与IEEE 754单精度和双精度浮点格式的一个重要的区别是：它没有隐藏位，有效位数共64位。

x87 FPU指令

- 早期的浮点处理器是作为CPU的外置协处理器出现的
- x87 FPU 特指与x86处理器配套的浮点协处理器架构
  - 浮点寄存器采用栈结构
    - 深度为8，宽度为80位，即8个80位寄存器 SKIP
    - 名称为 ST(0) ~ ST(7)，栈顶为ST(0)，编号分别为 0~7
  - 所有浮点运算都按80位扩展精度进行
  - 浮点数在浮点寄存器和内存之间传送
    - float、double、long double型变量在内存分别用IEEE 754单精度、双精度和扩展精度表示，分别占32位（4B）、64位（8B）和96位（12B，其中高16位无意义）
    - float、double、long double类型变量在浮点寄存器中都用80位扩展精度表示
    - 从浮点寄存器到内存：80位扩展精度格式转换为32位或64位
    - 从内存到浮点寄存器： 32位或64位格式转换为80位扩展精度格式

Intel处理器

x86前产品	4004 • 4040 • 8008 • 8080 • iAPX 432 • 8085
x87（外置浮点运算器）	8/16位总线: 8087 16位总线: 80187 • 80287 • 80387SX 32位总线: 80387DX • 80487
x86-16（16位）	8086 • 8088 • 80186 • 80188 • 80286
x86-32/IA-32（32位）	80386 • 80486 • Pentium（OverDrive、Pro、II、III、4、M） • Celeron（M、D） • Core
x86-64/Intel 64（64位）	Pentium（4（部份型号）、Pentium D、EE） • Celeron D（部份型号） • Core 2
EPIC/IA-64（64位）	Itanium
RISC	i860 • i960 • StrongARM • XScale
微控制器	8048 • 8051 • MCS-96
x86-32/IA-32	EP80579 • A100 • Atom（CE、SoC）
x86-64/Intel 64	Xeon（E3、E5、E7、Phi） • Atom（部分型号） • Celeron • Pentium • Core（i3、i5、i7）
EPIC/IA-64	Itanium 2

已停产

现有产品

BACK

X87 FPU指令

- **数据传送类**

(1) 装入（转换为80位扩展精度）

FLD：将数据从存储单元装入浮点寄存器栈顶 ST(0)

FILD：将数据从int型转换为浮点格式后，装入浮点寄存器栈顶

(2) 存储（转换为IEEE 754单精度或双精度）

FSTx：x为s/l时，将栈顶ST(0)转换为单/双精度格式，然后存入存储单元

FSTPx：弹出栈顶元素，并完成与FSTx相同的功能

FISTx：将栈顶数据从int型转换为浮点格式后，存入存储单元

FISTP：弹出栈顶元素，并完成与FISTx相同的功能

带P结尾指令表示操作数会出栈，也即ST(1)将变成ST(0)

X87 FPU指令

- **数据传送类**

(3) 交换

FXCH：交换栈顶和次栈顶两元素

(4) 常数装载到栈顶

FLD1 ：装入常数1.0

FLDZ ：装入常数0.0

FLDPI ：装入常数pi (=3.1415926...)

FLDL2E ：装入常数log(2)e

FLDL2T ：装入常数log(2)10

FLDLG2 ：装入常数log(10)2

FLDLN2 ：装入常数Log(e)2

X87 FPU指令

- **算术运算类**

(1) 加法

FADD/FADDP： 相加 / 相加后弹出栈

FIADD：按int型转换后相加

(2) 减法

FSUB/FSUBP ： 相减 / 相减后弹出栈

FSUBR/FSUBRP：调换次序相减 / 相减后弹出栈

FISUB：按int型转换后相减

FISUBR：按int型转换并调换次序相减

若指令未带操作数，则默认操作数为ST(0）、ST(1）

带R后缀指令是指操作数顺序变反，例如：

fsub执行的是x-y，fsubr执行的就是y-x

X87 FPU指令

- **算术运算类**

(3) 乘法

FMUL/FMULP: 相乘/相乘后弹出栈

FIMUL：按int型转换后相乘

(4) 除法

FDIV/FDIVP：相除/相除后弹出栈

FIDIV：按int型转换后相除

FDIVR/FDIVRP：调换次序相除 / 相减后弹出栈

FIDIVR：按int型转换并调换次序相除

IA-32浮点操作举例

问题：使用老版本gcc –O2编译时，程序一输出0，程序二输出是1，是什么原因造成的？ **f(10)的值是多少？机器数是多少？**

程序一：

```
#include <stdio.h>
double f(int x) {
        return 1.0 / x ;
}
void main() {
        double a, b;
        int i ;
        a = f(10) ;
        b = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

程序二：

```
#include <stdio.h>
double f(int x) {
        return 1.0 / x ;
}
void main() {
        double a, b, c;
        int i ;
        a = f(10) ;
        b = f(10) ;
        c = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

IA-32浮点操作举例

```
double f(int x)
{
    return 1.0 / x ;
}
```

```
8048328:    55          push   %ebp
8048329:    89 e5       mov    %esp,%ebp
804832b:    d9 e8       fld1
804832d:    da 75 08    fidivl 0x8(%ebp)
8048330:    c9          leave
8048331:    c3          ret
```

两条重要指令的功能如下

入口参数：int x=10

fld1：将常数1.0压入栈顶ST(0)

fidivl：将指定存储单元操作数M[R[ebp]+8]中的int型数转换为double型，再将ST(0)除以该数，并将结果存入ST(0)中

f(10)=1.0(80位扩展精度)/10(转换为double)=0.1

0.1=0.00011[0011]B= 0.00011 0011 0011 0011 0011 0011 0011…B

IA-32浮点操作举例

```
...
a = f(10) ;
b = f(10) ;
i = a == b;
...
```

```
08048334 <main>:
 8048334:      55                    push   %ebp
 8048335:      89 e5                 mov    %esp,%ebp
 8048337:      83 ec 08              sub    $0x8,%esp
 804833a:      83 e4 f0              and    $0xfffffff0,%esp
 804833d:      83 ec 0c              sub    $0xc,%esp
 8048340:      6a 0a                 push   $0xa
 8048342:      e8 e1 ff ff ff        call   8048328 <f> //计算a=f(10)
 8048347:      dd 5d f8              fstpl  0xfffffff8(%ebp) //a存入内存
 804834a:      c7 04 24 0a 00 00 00  movl   $0xa,(%esp,1)
 8048351:      e8 d2 ff ff ff        call   8048328 <f> //计算b=f(10)
 8048356:      dd 45 f8              fldl   0xfffffff8(%ebp) //a入栈顶
 8048359:      58                    pop    %eax
 804835a:      da e9                 fucompp      //比较ST(0)a和ST(1)b
 804835c:      df e0                 fnstsw %ax  //把FPU状态字送到AX
 804835e:      80 e4 45              and    $0x45,%ah
 8048361:      80 fc 40              cmp    $0x40,%ah
 8048364:      0f 94 c0              sete   %al
 8048367:      5a                    pop    %edx
 8048368:      0f b6 c0              movzbl %al,%eax
 804836b:      50                    push   %eax
 804836c:      68 d8 83 04 08        push   $0x80483d8
 8048371:      e8 f2 fe ff ff        call   8048268 <_init+0x38>
 8048376:      c9                    leave
 8048377:      c3                    ret
```

80位→64位

64位→80位

0.1是无限循环小数，无法精确表示，比较时，a舍入过而b没有舍入过，故 a≠b

IA-32浮点操作举例

```
...
a = f(10) ;
b = f(10) ;
c = f(10) ;
i = a == b;
...
```

```
8048342:    e8 e1 ff ff ff    call   8048328 <f> //计算a
8048347:    dd 5d f8          fstpl  0xfffffff8(%ebp) //把a存回内存
                                     //a产生精度损失
804834a:    c7 04 24 0a 00 00 00   movl   $0xa,(%esp,1)
8048351:    e8 d2 ff ff ff    call   8048328 <f> //计算b
8048356:    dd 5d f0          fstpl  0xfffffff0(%ebp) //把b存回内存
                                     //b产生精度损失
8048359:    c7 04 24 0a 00 00 00   movl   $0xa,(%esp,1)
8048360:    e8 c3 ff ff ff    call   8048328 <f> //计算c
8048365:    dd d8             fstp   %st(0)
8048367:    dd 45 f8          fldl   0xfffffff8(%ebp) //从内存中载入a
804836a:    dd 45 f0          fldl   0xfffffff0(%ebp) //从内存中载入b
804836d:    d9 c9             fxch   %st(1)
804836f:    58                pop    %eax
8048370:    da e9             fucompp //比较a，b
8048372:    df e0             fnstsw %ax
```

0.1是无限循环小数，无法精确表示，比较时，a和b都是舍入过的，故 a=b！

IA-32浮点操作举例

- 从这个例子可以看出
  - 编译器的设计和硬件结构紧密相关。
  - 对于编译器设计者来说，只有真正了解底层硬件结构和真正理解指令集体系结构，才能够翻译出没有错误的目标代码，并为程序员完全屏蔽掉硬件实现的细节，方便应用程序员开发出可靠的程序。
  - 对于应用程序开发者来说，也只有真正了解底层硬件的结构，才有能力编制出高效的程序，能够快速定位出错的地方，并对程序的行为作出正确的判断。

IA-32浮点操作举例

C/C++ code

```
1  #include "stdafx.h"
2  int main(int argc, char* argv[])
3  {
4      int a=10;
5      double *p=(double*)&a;
6      printf("%f\n",*p);              //结果为0.000000
7      printf("%f\n",(double(a)));     //结果为10.000000
8
9      return 0;
10 }
11 为什么printf("%f",*p)和printf("%f",(double)a)结果不一样呢？
```

不都是强制类型转换吗？怎么会不一样

关键差别在于一条指令：

fldl 和 fildl

IA-32浮点操作举例

```
int a = 10;
 8048425:  c7 44 24 28 0a 00 00 00     movl   $0xa,0x28(%esp)
double *p = (double *)&a;
 804842d:  8d 44 24 28                 lea    0x28(%esp),%eax
 8048431:  89 44 24 2c                 mov    %eax,0x2c(%esp)
```

可以看到关于指针的类型转换在汇编层次并没有体现出来，都是直接 mov 过去

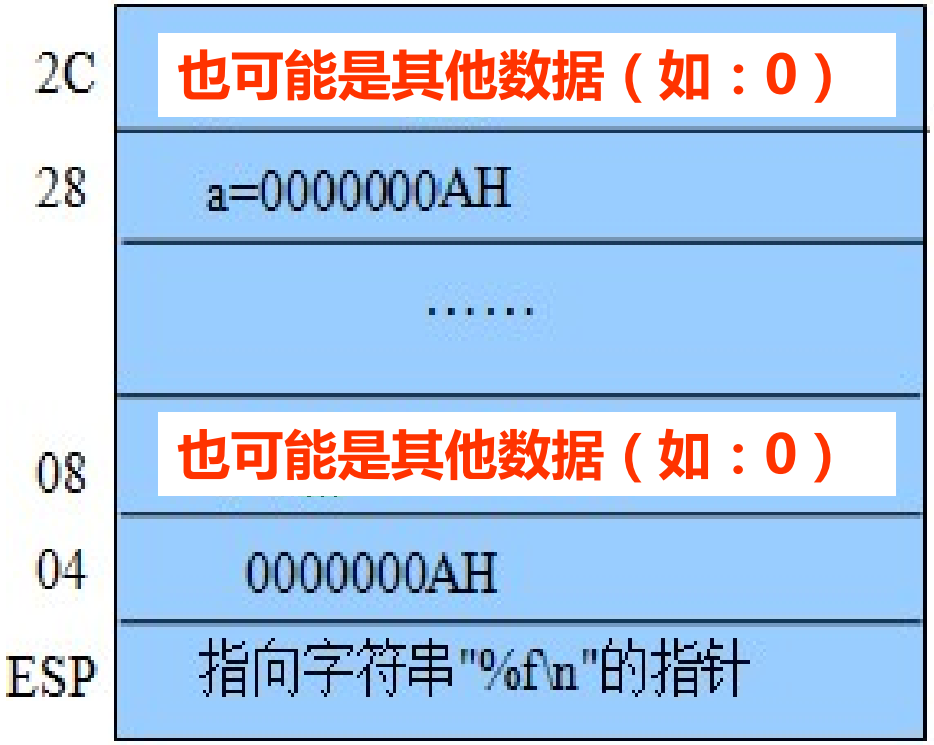
```
printf("%lf\n", *p);
                                      mov    0x2c(%esp),%eax
                                      fldl   (%eax)
```

度加载到浮点栈顶 S`0(7)）

```
                                      fstpl  0x4(%esp)
```

p 的类型是 double，故按 64 位压栈）

```
                                      movl   $0x8048500,(%esp)
                                      call   8048300 <printf@plt>
```




```
                                      mov    0x28(%esp),%eax
                                      mov    %eax,0x1c(%esp)
```

v 操作，把变量 a 的值移来移去

```
                                      fildl  0x1c(%esp)
```

的是 fildl 指令，和上面用的 fldl 指令不一样!

IA-32浮点操作举例

- **有一个跟帖的解释如下**

请问：这个帖子的回答中，哪些是正确的？哪些是错误的？

（1）

10=0000000AH，即0A是LSB，所以00H、00H、00H、0AH是printf所打印的double数据的低四字节，高四字节不确定

a是int型，内存（小端）中的表示是

0000 1010 0000 0000 0000 0000 0000 0000 后面的位不确定

double型占用8个字节，如果将上述字节看做double，第一位是符号位，第2~12位是阶码，第12~64是位数，

0000 1010 0000 0000 0000 0000 0000 0000 后面的不确定，

那么你可以转换成实数算下，应该很小（小数点的后6位肯定都是零），输出的时候默认为6位小数，发生截断，所以是

0.000000

（2）printf("%f\n",(double(a)));发生类型转化，这个可以，一般sizeof比较小的类型可以转换成size比较大的类型，或者是类型提升或者是转换

MMX及SSE指令

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

MMX/SSE指令集的由来

- **由MMX发展而来的SSE架构**
  - ✓ **MMX指令使用8个64位寄存器MM0~MM7，借用8个80位寄存器ST(0)~ST(7)中64位尾数所占的位，可同时处理8个字节，或4个字，或2个双字，或一个64位的数据**
  - ✓ **MMX指令并没带来3D游戏性能的显著提升，故推出SSE指令，并陆续推出SSE2、SSE3、SSSE3和SSE4等采用SIMD技术的指令集，这些统称为SSE指令集**
  - ✓ **SSE指令集将80位浮点寄存器扩充到128位多媒体扩展通用寄存器XMM0~XMM7，可同时处理16个字节，或8个字，或4个双字（32位整数或单精度浮点数），或两个四字的数据**
  - ✓ **从SSE2开始，还支持128位整数运算，或同时并行处理两个64位双精度浮点数**

IA-32中通用寄存器中的编号

编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

反映了体系结构发展的轨迹，字长不断扩充，指令保持兼容

ST（0）~ ST（7）是80位，MM0 ~MM7使用其低64位

SSE指令（SIMD操作）

- 用简单的例子来比较普通指令与数据级并行指令的执行速度
  - ✓为使比较结果不受访存操作影响，下例中的运算操作数在寄存器中
  - ✓为使比较结果尽量准确，例中设置的循环次数较大: 0x4000000=2^{26}
  - ✓例子只是为了说明指令执行速度的快慢，并没有考虑结果是否溢出

以下是普通指令写的程序

```
080484f0 <dummy_add>:                                   所用时间约为22.643816s
 80484f0:  55              push  %ebp
 80484f1:  89 e5           mov  %esp, %ebp
 80484f3:  b9 00 00 00 04  mov  $0x4000000, %ecx
 80484f8:  b0 01           mov  $0x1, %al
 80484fa:   b3 00          mov   $0x0, %bl
 80484fc: 00 c3            add   %al, %bl
 80484fe: e2 fc            loop  80484fc <dummy_add+0xc>
8048500:  5d               pop           %ebp
 8048501: c3               ret
```

循环400 0000H=2^{26}次，每次只有一个数（字节）相加

SSE指令（SIMD操作）

以下是SIMD指令写的程序

所用时间约为1.411588s

```
08048510 <dummy_add_sse>:
 8048510:  55                   push  %ebp
 8048511:  b8 00 9d 04 10       mov $0x10049d00, %eax
 8048516:  89 e5                mov   %esp, %ebp
 8048518:  53                   push   %ebx
 8048519:   bb 20 9d 04 14      mov   $0x14049d20, %ebx
 804851e:  b9 00 00 40 00       mov   $0x400000, %ecx
 8048523:  66 0f 6f 00          movdqa  (%eax), %xmm0
 8048527:  66 0f 6f 0b          movdqa  (%ebx), %xmm1
 804852b: 66 0f fc c8           paddb    %xmm0, %xmm1
 804852f: e2 fa                 loop   804852b <dummy_add_sse+0x1b>
 8048531:  5b                   pop    %ebx
 8048532:  5d                   pop    %ebp
 8048533:  c3                   ret
```

SIDM指令

22.643816s/ 1.411588s ≈16.041378,与预期结果一致! SIMD指令并行执行效率高!

循环400000H=2^{22}次，每次同时有128/8=16个数（字节）相加

SSE指令（SIMD操作）

- paddb指令（操作数在两个xmm寄存器中）
  - 一条指令同时完成16个单字节数据相加
  - 类似指令paddw同时完成8个单字数据相加
  - 类似指令psubl同时完成4个双字数据相减
- movdqa指令
  - 将双四字（128位）从源操作数处移到目标操作数处
  - 用于在 XMM 寄存器与 128 位存储单元之间移入/移出双四字，或在两个 XMM 寄存器之间移动
  - 源操作数或目标操作数是存储器操作数时，操作数必须是 16 字节边界对齐，否则将发生一般保护性异常 (#GP)
- movdqu指令
  - 在未对齐的存储单元中移入/移出双四字

更多有关SSE指令集的内容请参看Intel的相关资料