第1周 程序执行概述

第1讲 程序和指令的关系

第2讲 一条指令的执行过程

第3讲 在IA-32中一条指令的执行过程

第4讲 CPU的基本功能与结构

程序及指令的执行过程

- 程序和指令的关系
  - 程序由一条一条指令组成，指令按顺序存放在内存连续单元
- 程序的执行：周而复始地执行一条一条指令
  - 正常情况下，指令按其存放顺序执行
  - 遇到需改变程序执行流程时，用相应的转移指令（包括无条件转移指令、条件转移指令、调用指令和返回指令等）来改变程序执行流程
- 程序的执行流的控制
  - 将要执行的指令所在存储单元的地址由程序计数器PC给出，通过改变PC的值来控制执行顺序
- 指令周期：CPU取出并执行一条指令的时间

程序及指令的执行过程

程序执行流：

```
……
call outputs
……
call strcpy
……
call printf
……
ret
mov %eax,…
……
```

```
main：
……
call outputs
mov eax,…
……
ret
```

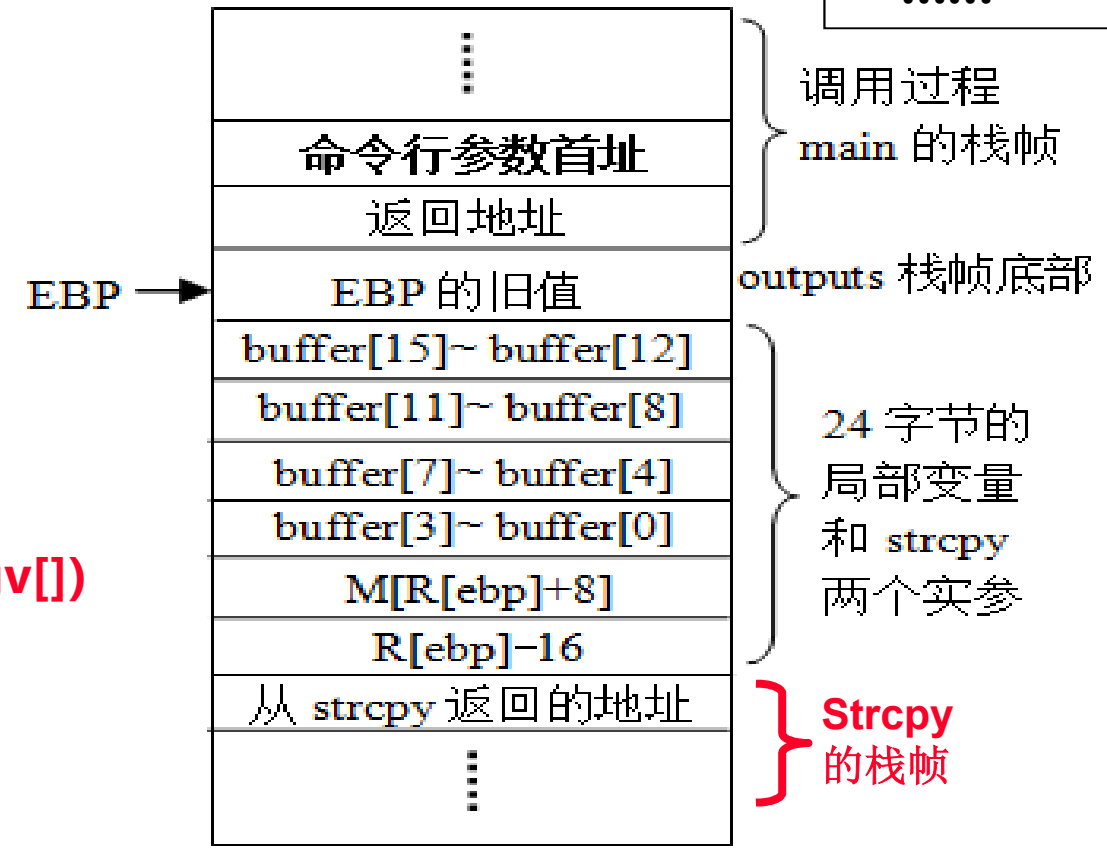
```
outputs：
……
call strcpy
……
call printf
……
ret
```

```
strcpy:
```

对于3.6.1中的例子

```
#include "stdio.h"
#include "string.h"
 void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n", buffer);
}
     ……

int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```



程序及指令的执行过程

反汇编得到的outputs汇编代码

```
080483e4 : push   %ebp
080483e5 : mov    %esp,%ebp
080483e7 : sub    $0x18,%esp
080483ea : mov    0x8(%ebp),%eax
080483ed: mov     %eax,0x4(%esp)
080483f1 : lea    0xfffffff0(%ebp),%eax
080483f4 : mov    %eax,(%esp)
080483f7 : call   0x8048330 <__gmon_start__@plt+16>
080483fc : lea    0xfffffff0(%ebp),%eax
080483ff : mov    %eax,0x4(%esp)
08048403: movl    $0x8048500,(%esp)
0804840a: call    0x8048310
0804840f : leave
08048410: ret
```

将strcpy的两个实参入栈

将printf的两个实参入栈

程序及指令的执行过程

在内存存放的指令实际上是机器代码（0/1序列）

```
08048394 <add>:
1 8048394:   55          push  %ebp
2 8048395:   89  e5      mov   %esp, %ebp
3 8048397:   8b  45 0c   mov   0xc(%ebp), %eax
4 804839a:   03  45 08   add   0x8(%ebp), %eax
5 804839d:   5d          pop   %ebp
6 804839e:   c3          ret
```

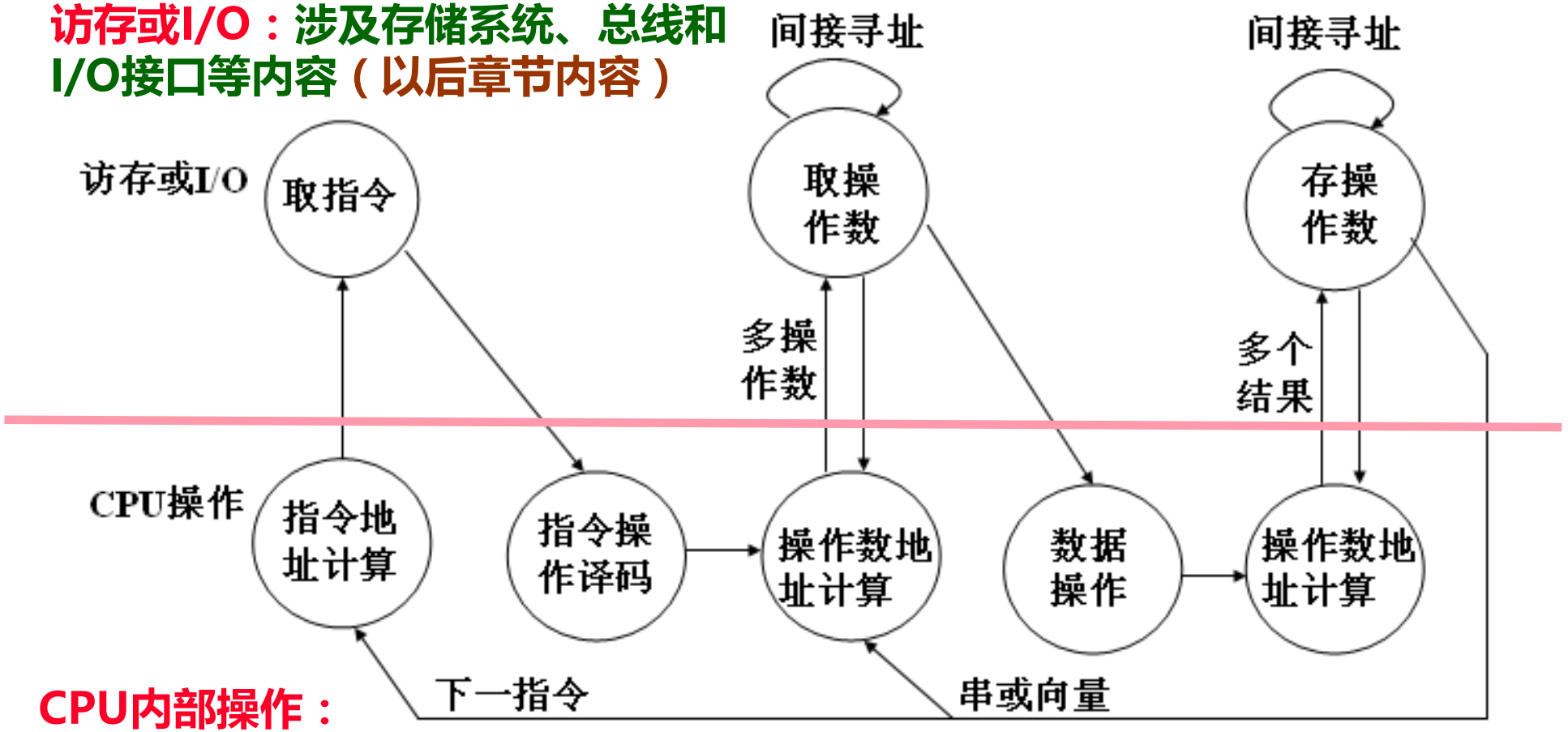
程序执行需要解决的问题：

如何判定每条指令有多长？如何判定操作类型、寄存器编号、立即数等？如何区分第2行和第3行mov指令的不同？如何确定操作数是在寄存器中还是在存储器中？一条指令执行结束后如何正确读取到下一条指令？

- 对于add函数
  - ✓指令按顺序存放在0x08048394开始的存储空间。
  - ✓各指令长度可能不同，如push、pop和ret指令各占一个字节，第2行mov指令占两个字节，第3行mov指令和第4行add指令各占3字节。
  - ✓各指令对应的0/1序列含义有不同的规定，如“push %ebp”指令为01010101B，其中01010为push指令操作码，101为EBP的编号，“pop %ebp”为01011101B，其中01011为pop指令的操作码。

程序及指令的执行过程

访存或I/O：涉及存储系统、总线和I/O接口等内容（以后章节内容）



CPU内部操作：涉及CPU内部数据通路（本章节内容）

CPU运行程序的过程就是执行一条一条指令的过程

CPU执行指令的过程中，包含CPU操作、访问内存或I/O端口的操作两类

机器指令的执行过程

- CPU执行指令的过程
  - 取指令
  - PC+“1”
  - 指令译码
  - 进行主存地址运算
  - 取操作数
  - 进行算术 / 逻辑运算
  - 存结果
  - 以上每步都需检测“异常”
  - 若有异常，则自动切换到异常处理程序
  - 检测是否有“中断”请求，有则转中断处理

取指阶段

执行阶段

指令执行过程

“1”：指一条指令的长度，定长指令字每次都一样；变长指令字每次可能不同

定长指令字通常在译码前做，变长指令字在译码后做！

问题：

“取指令”一定在最开始做吗？PC+“1”一定在译码之前做吗？

“译码”须在指令执行前做吗？

你能说出几种“异常”事件？“异常”和“中断”的差别是什么？

异常是在CPU内部发生的，中断是由外部事件引起的

机器指令的执行过程

- **取指令：**从PC所指单元取出指令送指令寄存器（IR），并增量PC。
  - 如add函数，开始PC（IA-32的EIP）中存放的是0x0848394，CPU根据PC取指令送IR，每次总是取最长指令字节数，假定最长指令是4个字节，即IR为32位，此时，也即55 89 E5 8BH被取到IR中。
- **指令译码：**不同指令其功能不同，因而需要不同的操作控制信号。
  - CPU根据不同操作码译出不同控制信号。对于上述取到IR中的55 89 E5 8BH译码时，可根据高5位01010译码得到push指令的控制信号。
- **源操作数地址计算并取操作数：**根据寻址方式确定源操作数地址计算方式，若是存储器数据，则需一次或多次访存；若为间接寻址或两操作数都在存储器的双目运算，则需多次访存；若是寄存器数据，则直接从寄存器取数。
- **执行数据操作：**在ALU或加法器等运算部件中对取出的源操作数进行运算。
- **目的操作数地址计算并存结果：**根据寻址方式确定目的操作数地址计算方式，若是存储器数据，则需要一次或多次访存（间接寻址时）；若是寄存器数据，则在进行数据操作时直接存结果到寄存器。
- **指令地址计算并将其送PC。**顺序执行时，PC加上当前指令长度；遇到转移类指令时，则需要根据条件码、操作码和寻址方式等确定下条指令地址。

机器指令的执行过程

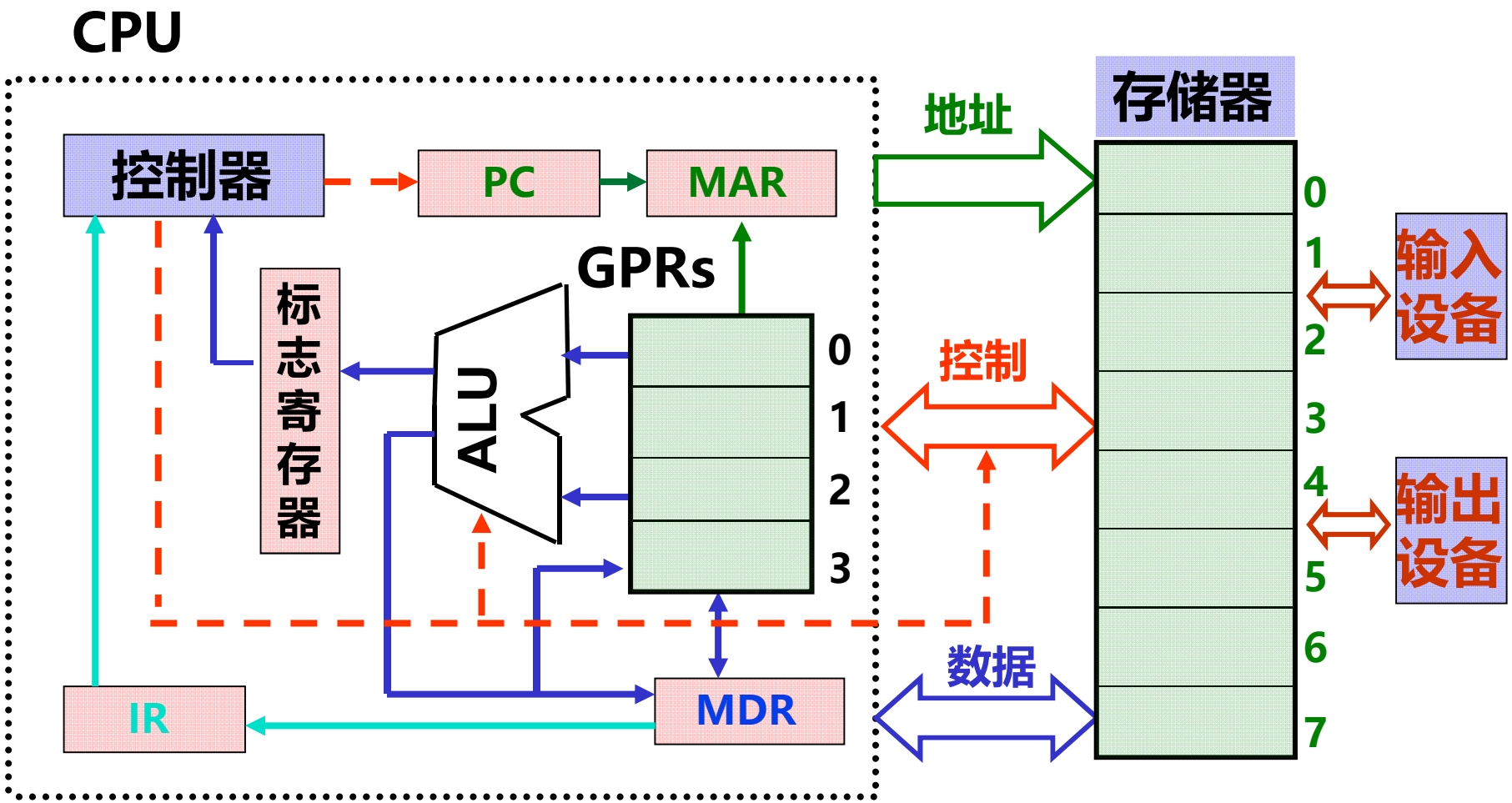
- 每条指令的功能总是由以下四种基本操作来实现：
  - 读取某一主存单元的内容，并将其装入某个寄存器（取指， 取数）
  - 把一个数据从某个寄存器存入给定的主存单元中（存结果）
  - 把一个数据从某寄存器送到另一寄存器或者ALU（取数，存结果）
  - 进行算术或逻辑运算（PC+”1”，计算地址，运算）
  - 指令执行过程中查询各种异常情况，并在发现异常时转异常处理
  - 指令执行结束时查询中断请求，并在发现中断请求时响应中断
- 操作功能可形式化描述
  - 描述语言称为寄存器传送语言RTL (Register Transfer Language)

回顾：冯.诺依曼结构模型机

你妈会做的菜和厨师会做的菜不一样，同一个菜谱的做法也可能不同

如同

不同架构支持的指令集不同，同一种指令的实现方式和功能也可能不同



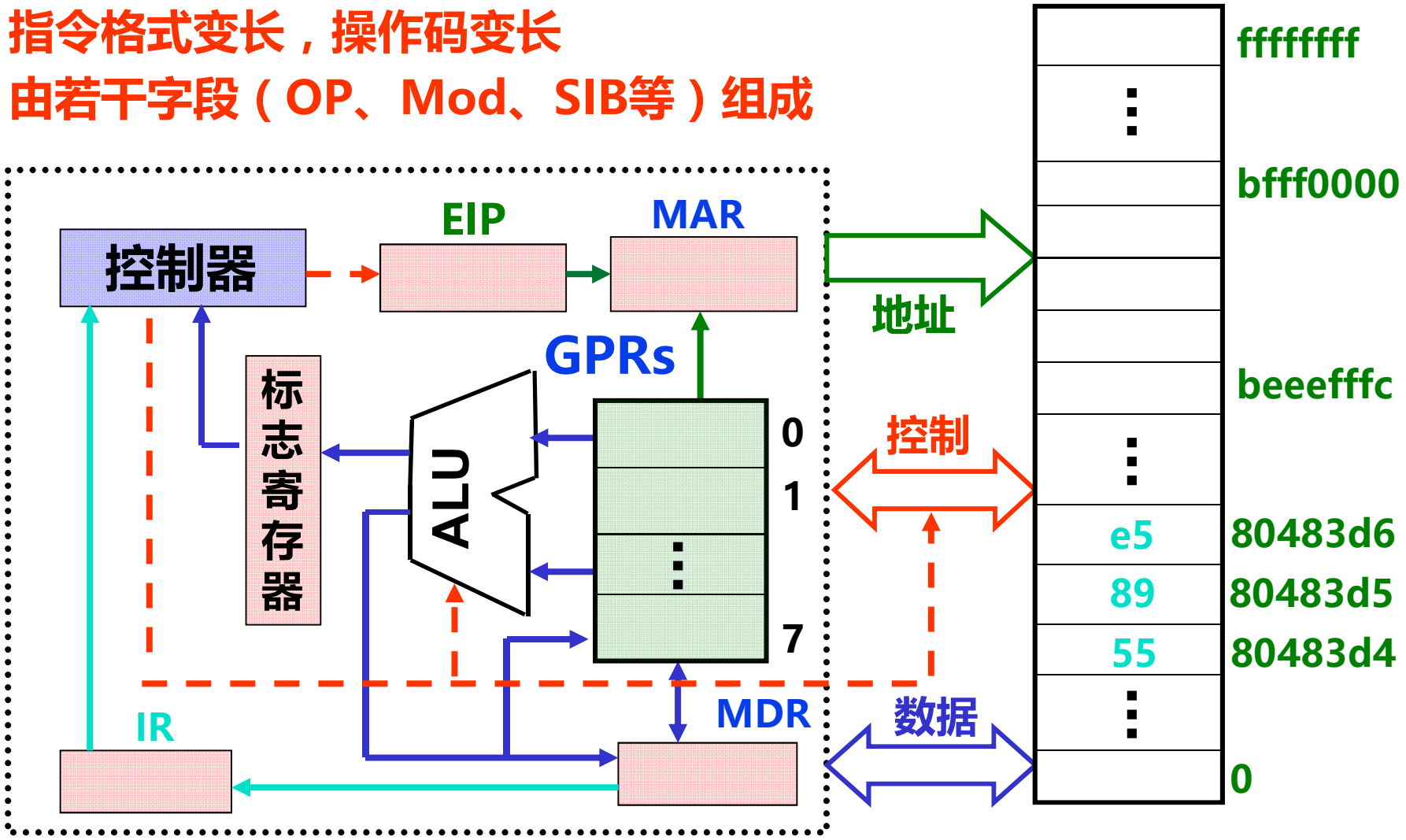
回顾： IA-32的体系结构是怎样的呢？

8个GPR（0~7），一个EFLAGs，PC为EIP

可寻址空间4GB（编号为0~0xFFFFFFFF）

指令格式变长，操作码变长

由若干字段（OP、Mod、SIB等）组成



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，
则程序执行结果是什么？
每一步如何执行？

想想妈妈怎么做菜的？

“objdump -d test” 显示的add函数结果

```
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
```

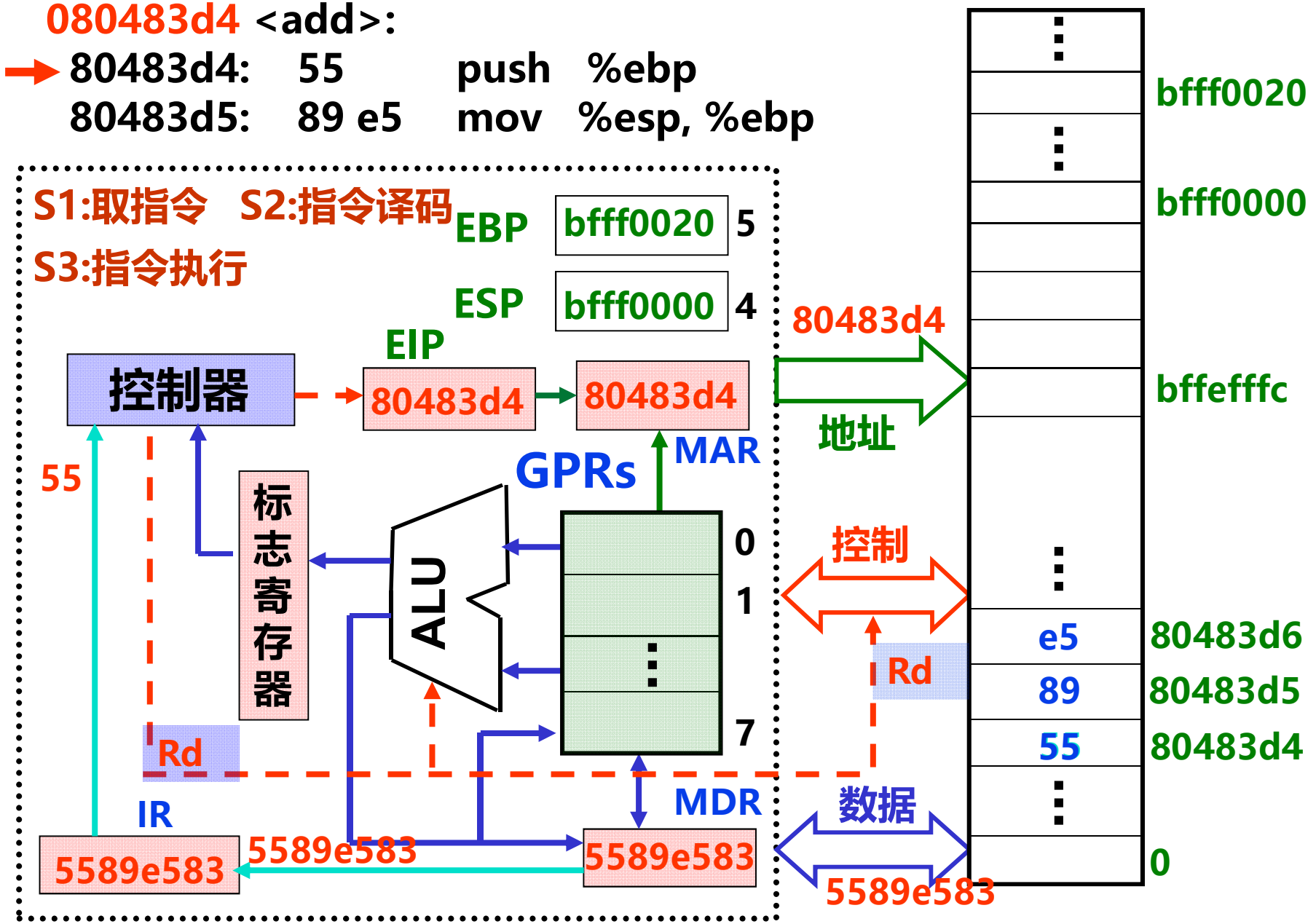
OP

起始EIP=?

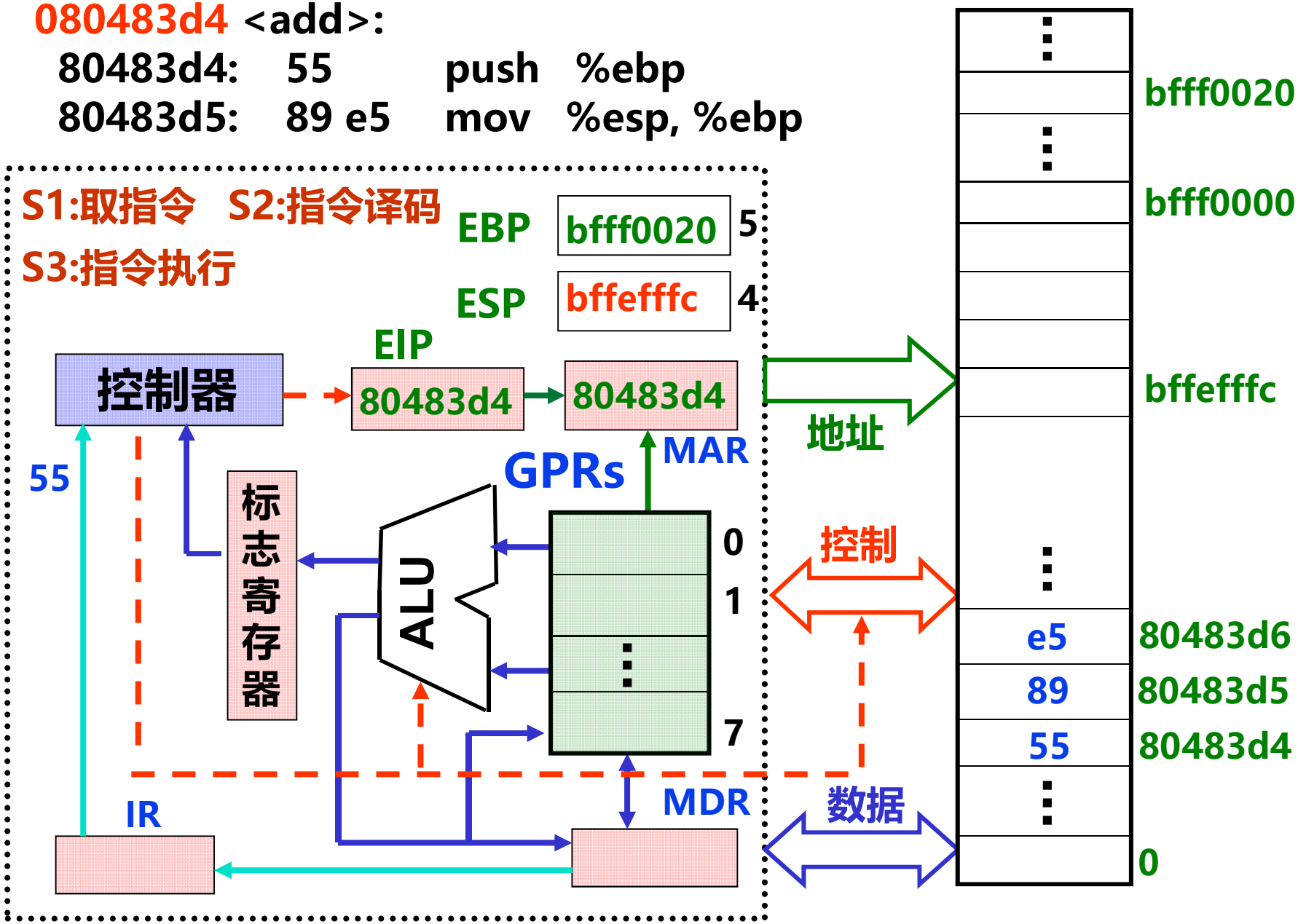
代码执行从80483d4开始！

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

功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

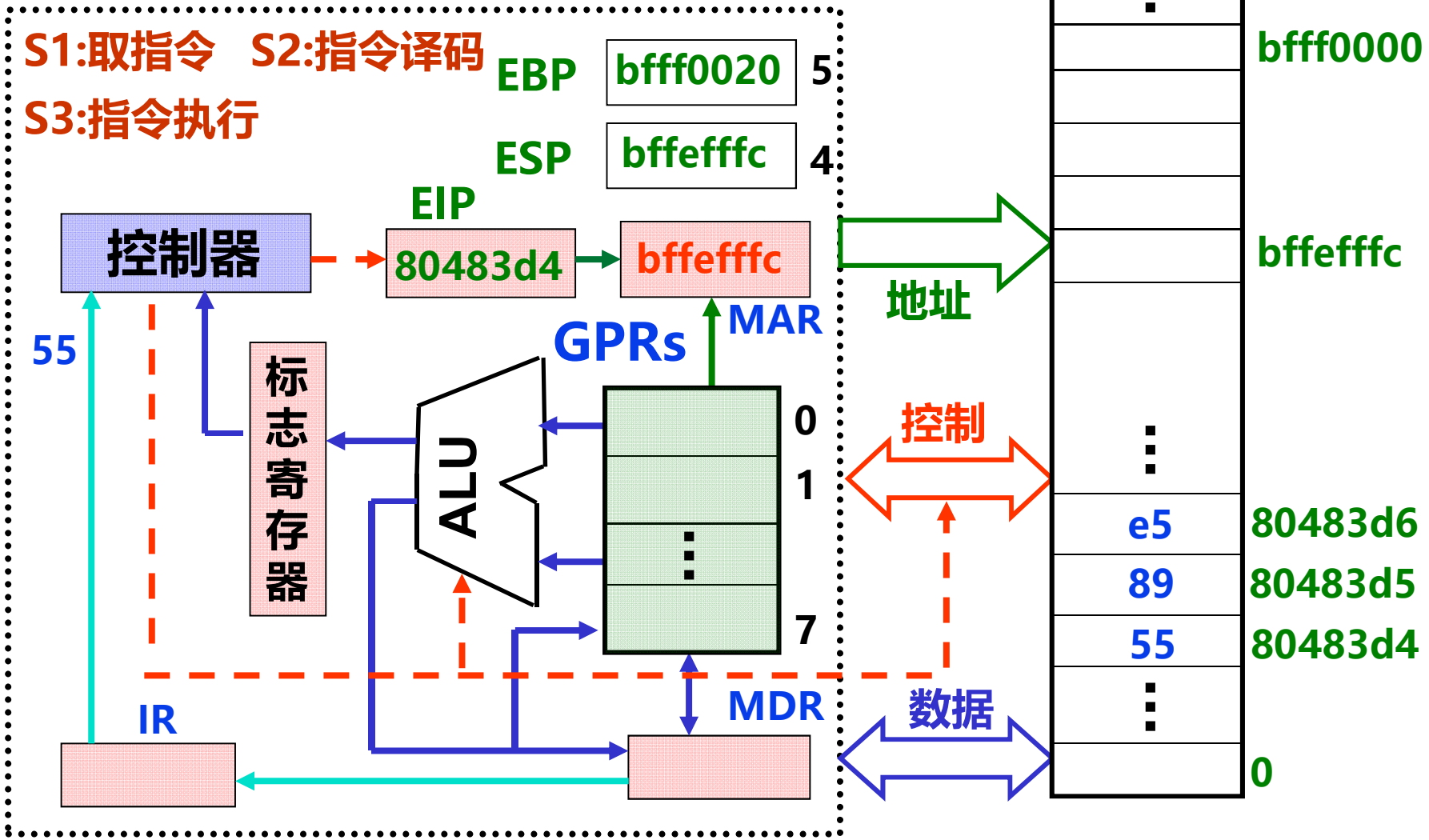


功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]



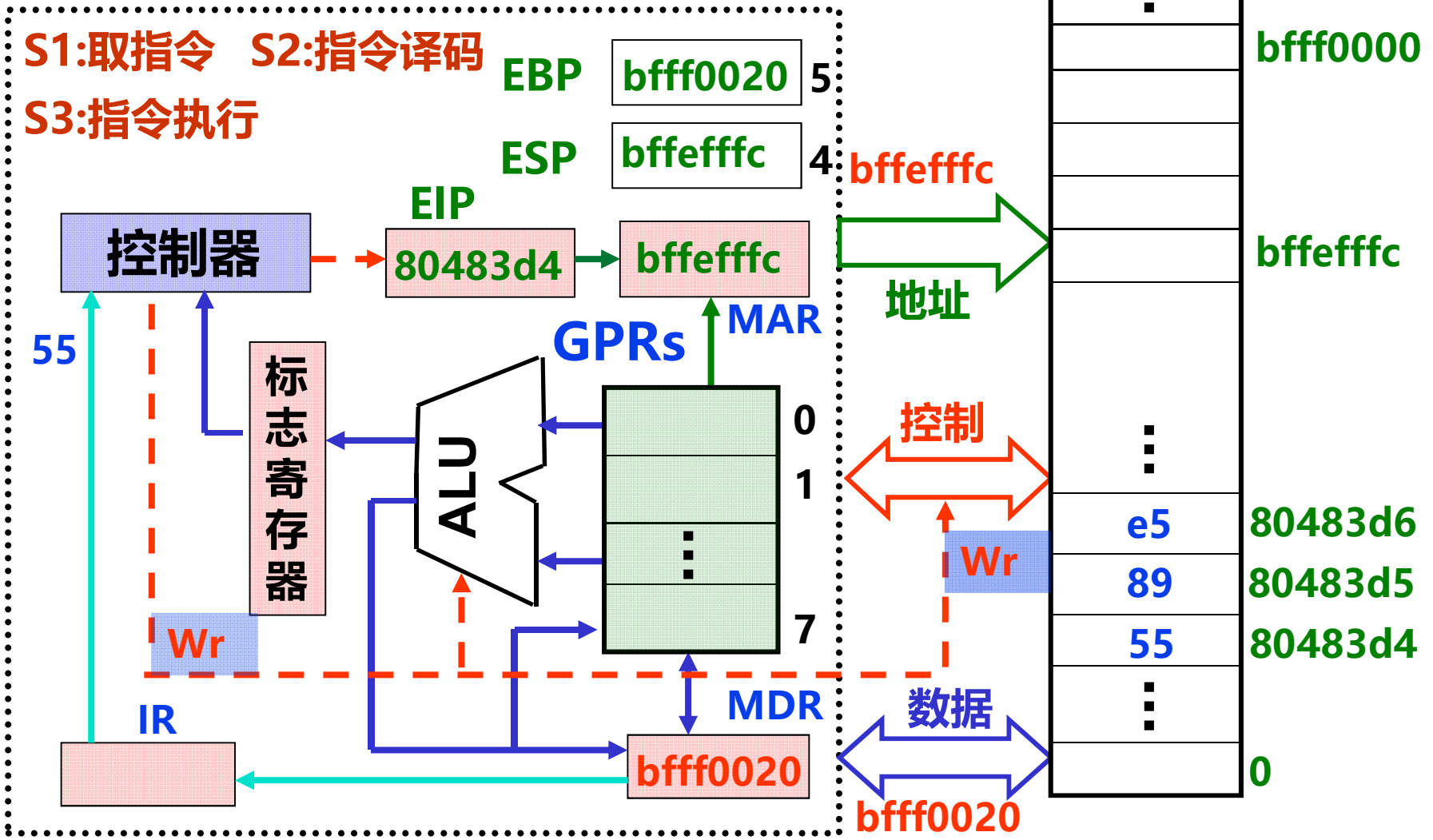
功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

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

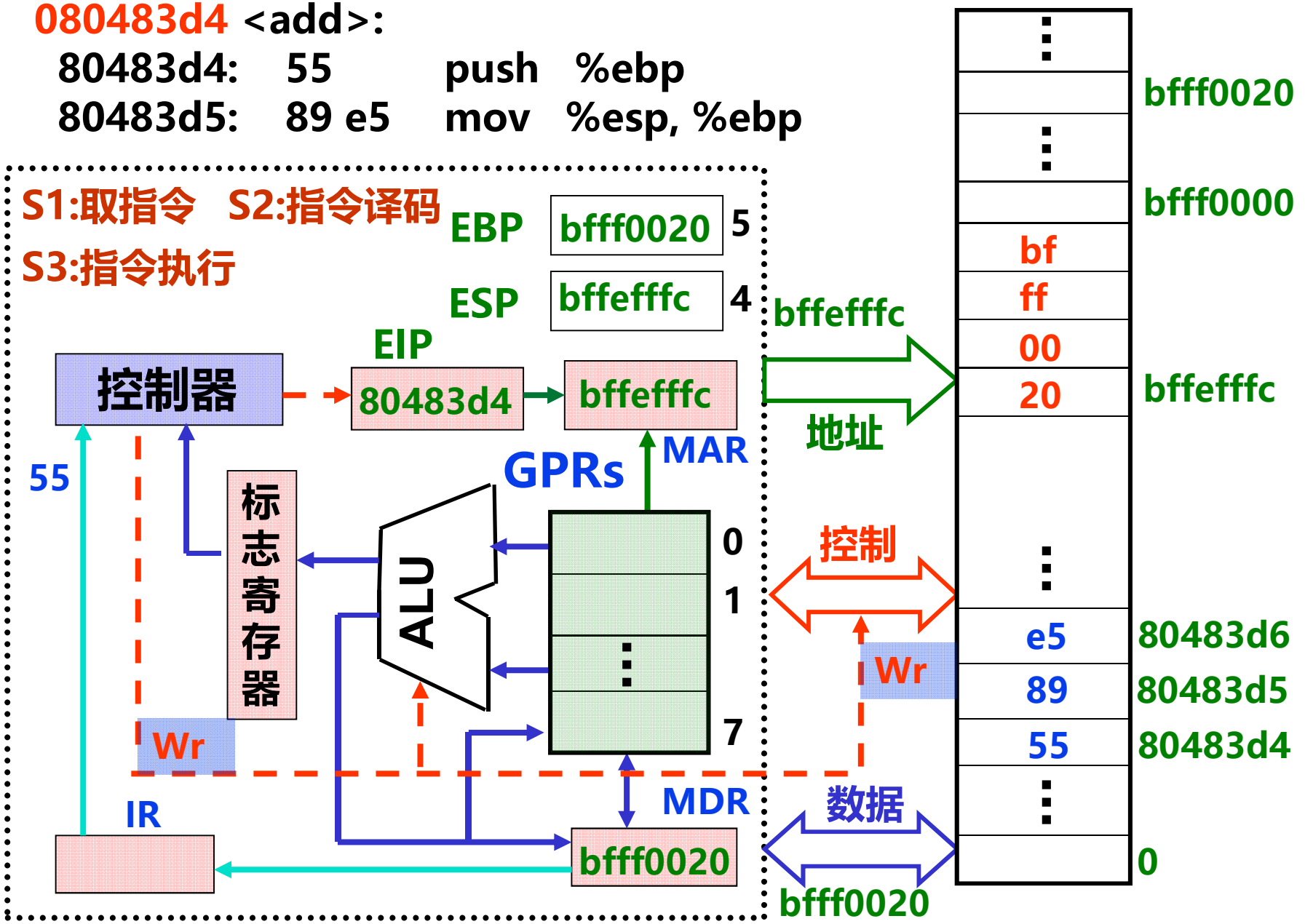


功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

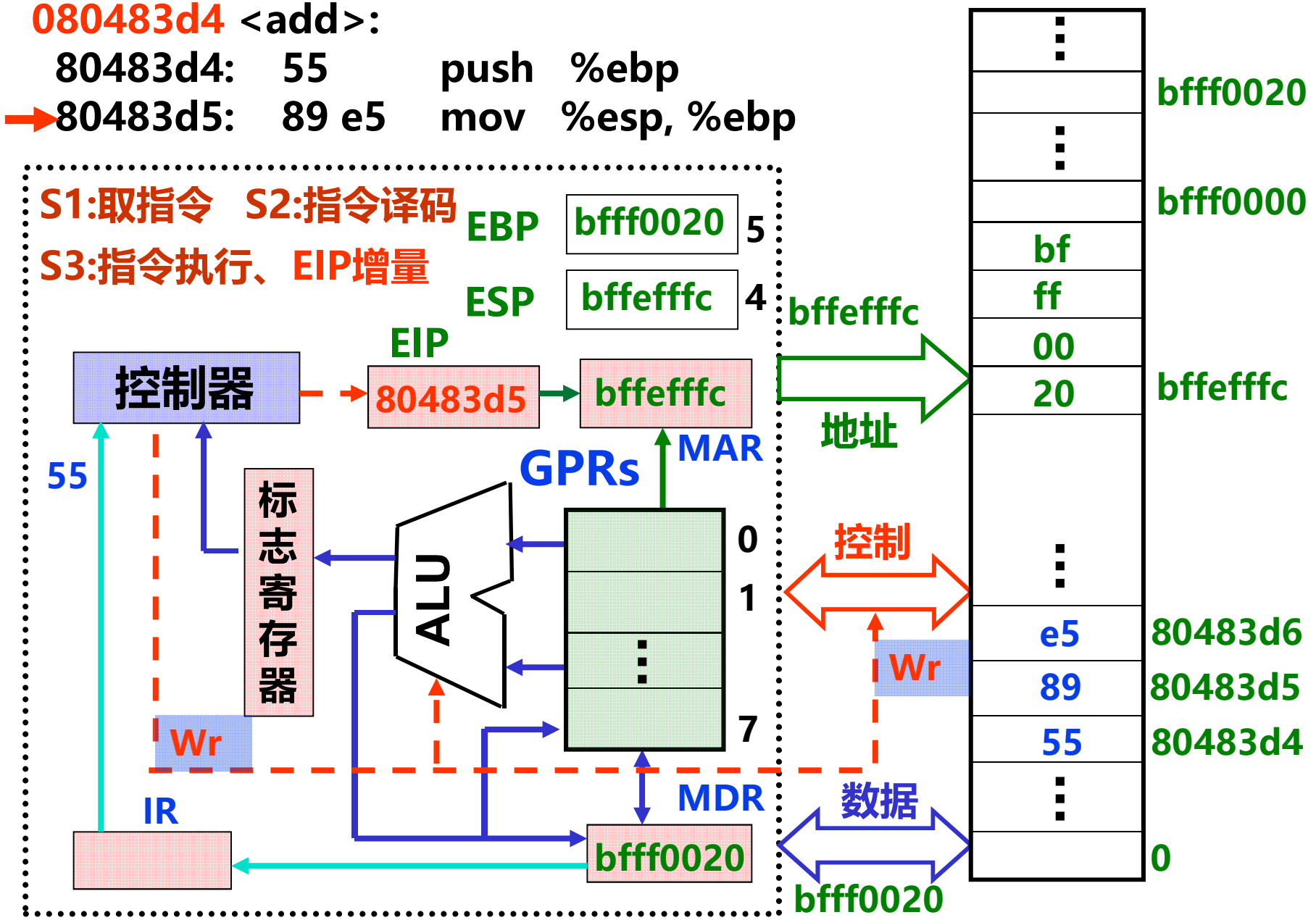
```
080483d4 <add>:
  80483d4:    55          push   %ebp
  80483d5:    89 e5       mov    %esp, %ebp
```



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
bffefffc
4
EIP
控制器
80483d5
bffefffc
MAR
GPRs
55
标志寄存器
ALU
0
1
7
Wr
IR
MDR
bfff0020
bffefffc
地址
控制
Wr
数据
bfff0020
bfff0020
bfff0000
bf
ff
00
20
bffefffc
e5
89
55
80483d6
80483d5
80483d4
0

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，

则程序执行结果是什么？

每一步如何执行？

```
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
```

OP

EDX和EAX中各是什么？

R[edx]=i=0x7fffffff

R[eax]=j=0x2

回顾： IA-32的寄存器组织

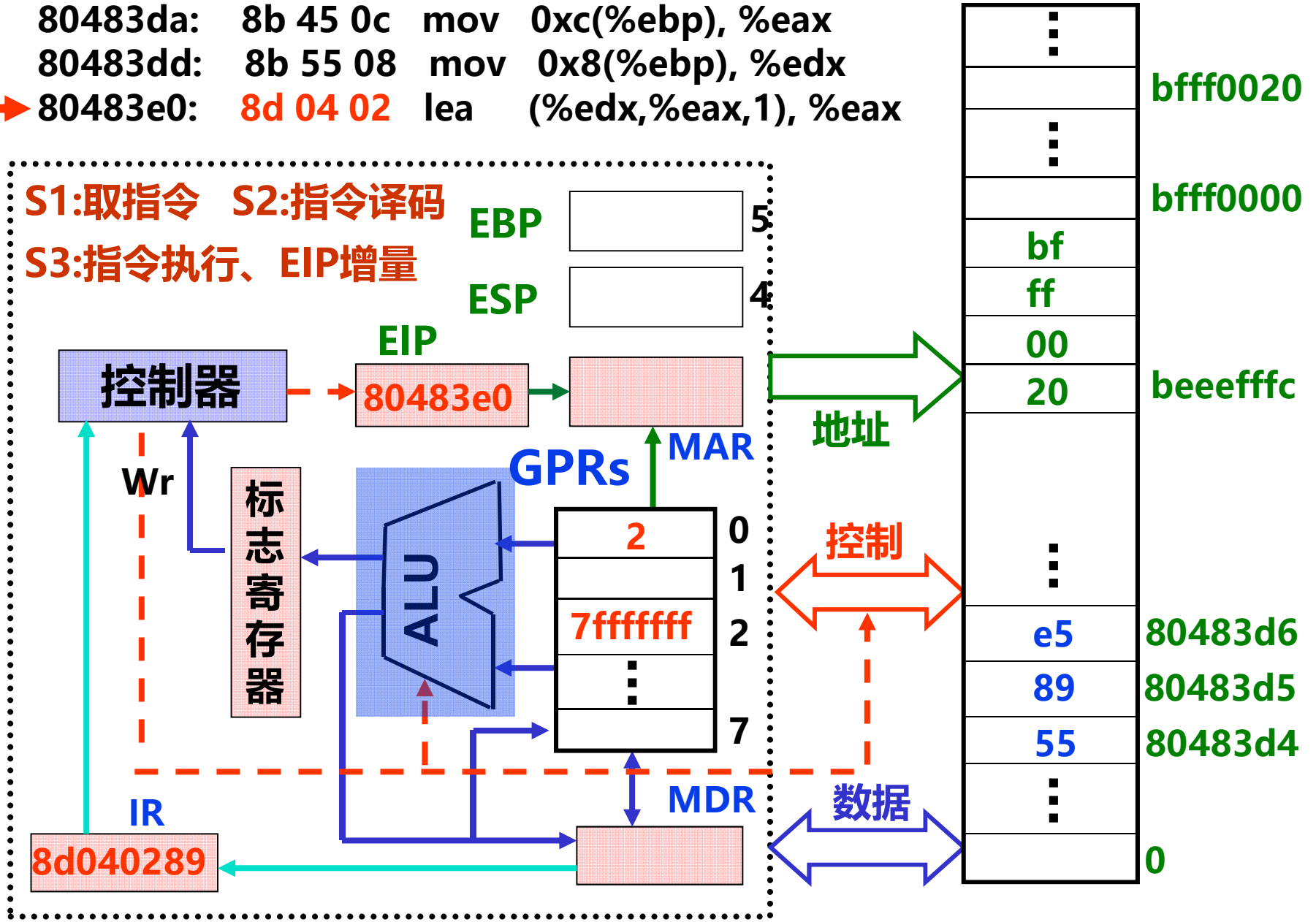
编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

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

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
```



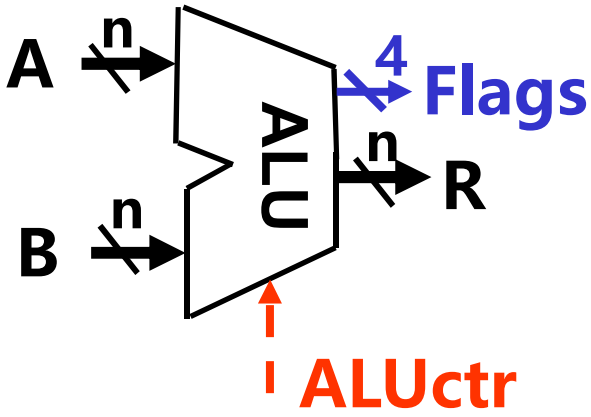
ALU长啥样呢？

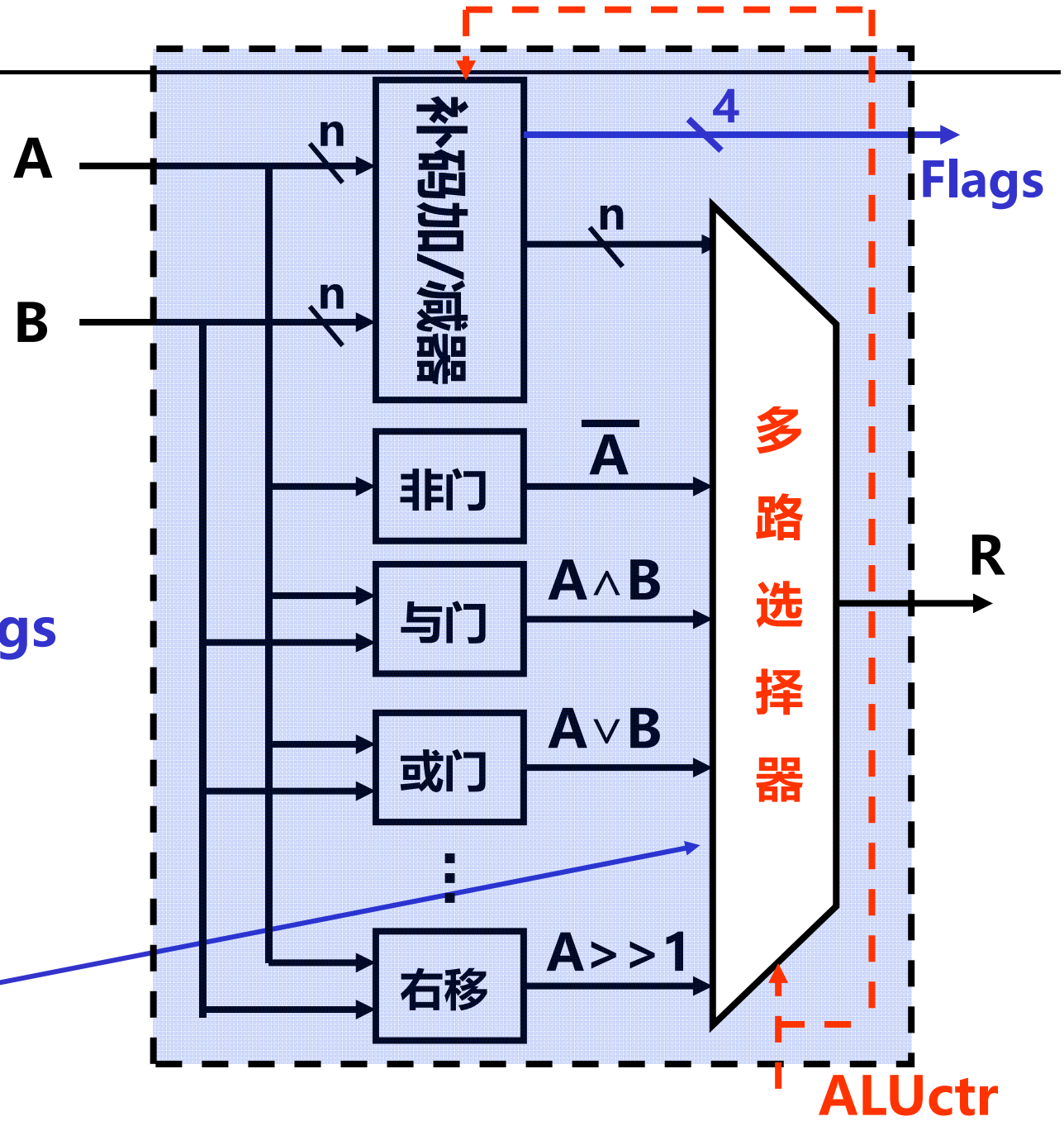
- 试想一下ALU中有哪些部件？（想想厨房做菜用什么工具？）
  - 补码加/减器（可以干什么？）
    - 带符号加、带符号减
    - 无符号加、无符号减
  - ~~乘法器~~？（为什么可以没有？）
    - 可用加/减+移位实现，也可有独立乘法器
    - 带符号乘和无符号乘是各自独立的部件
  - ~~除法器~~？（为什么可以没有？）
    - 可用加/减+移位实现，也可有独立除法器
    - 带符号除和无符号除是各自独立的部件
  - 各种逻辑运算部件（可以干什么？）
    - 非、与、或、非、前置0个数、前置1个数……

大家能否画出ALU框图？

ALU结构原理

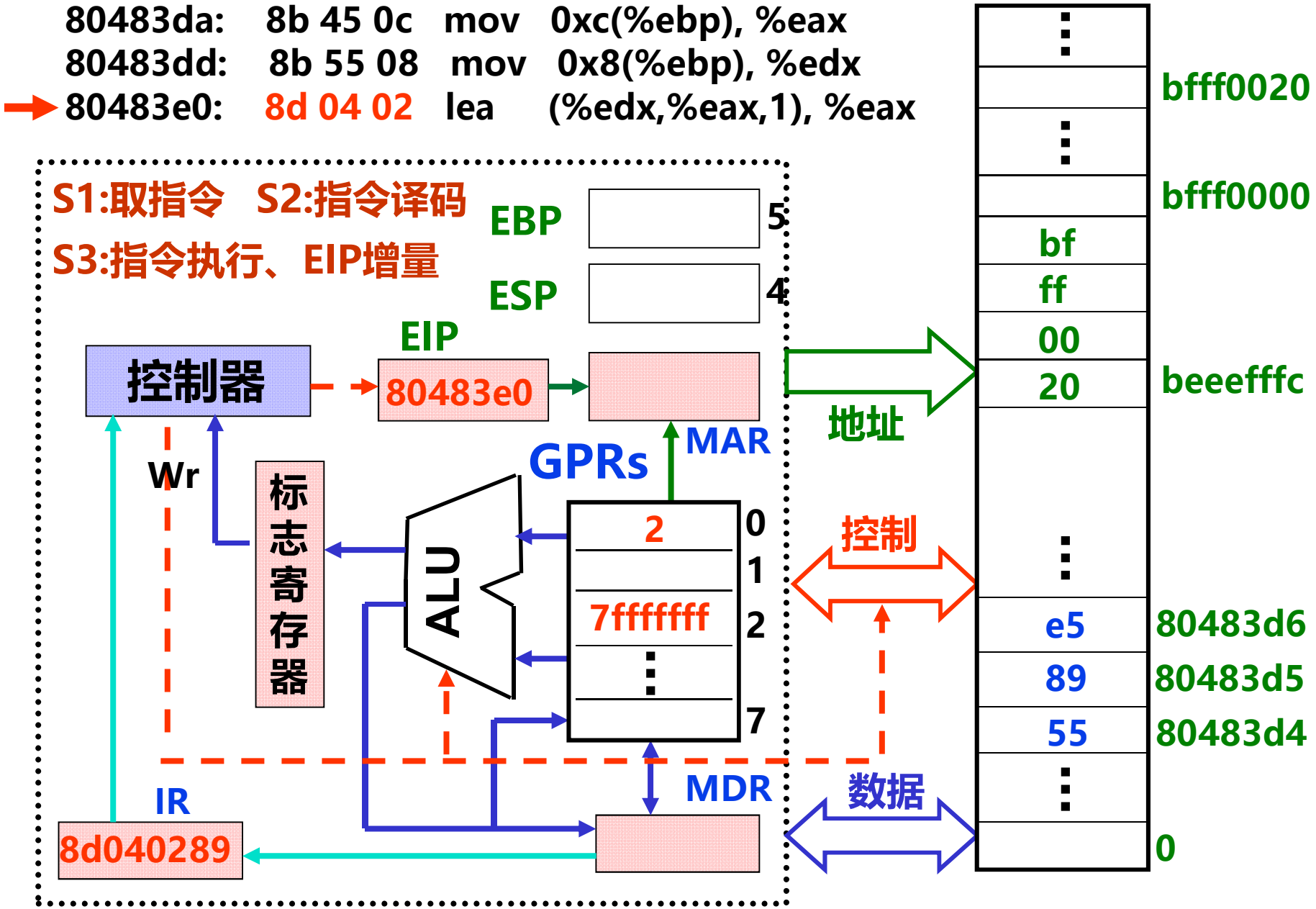
ALU符号是怎样的？



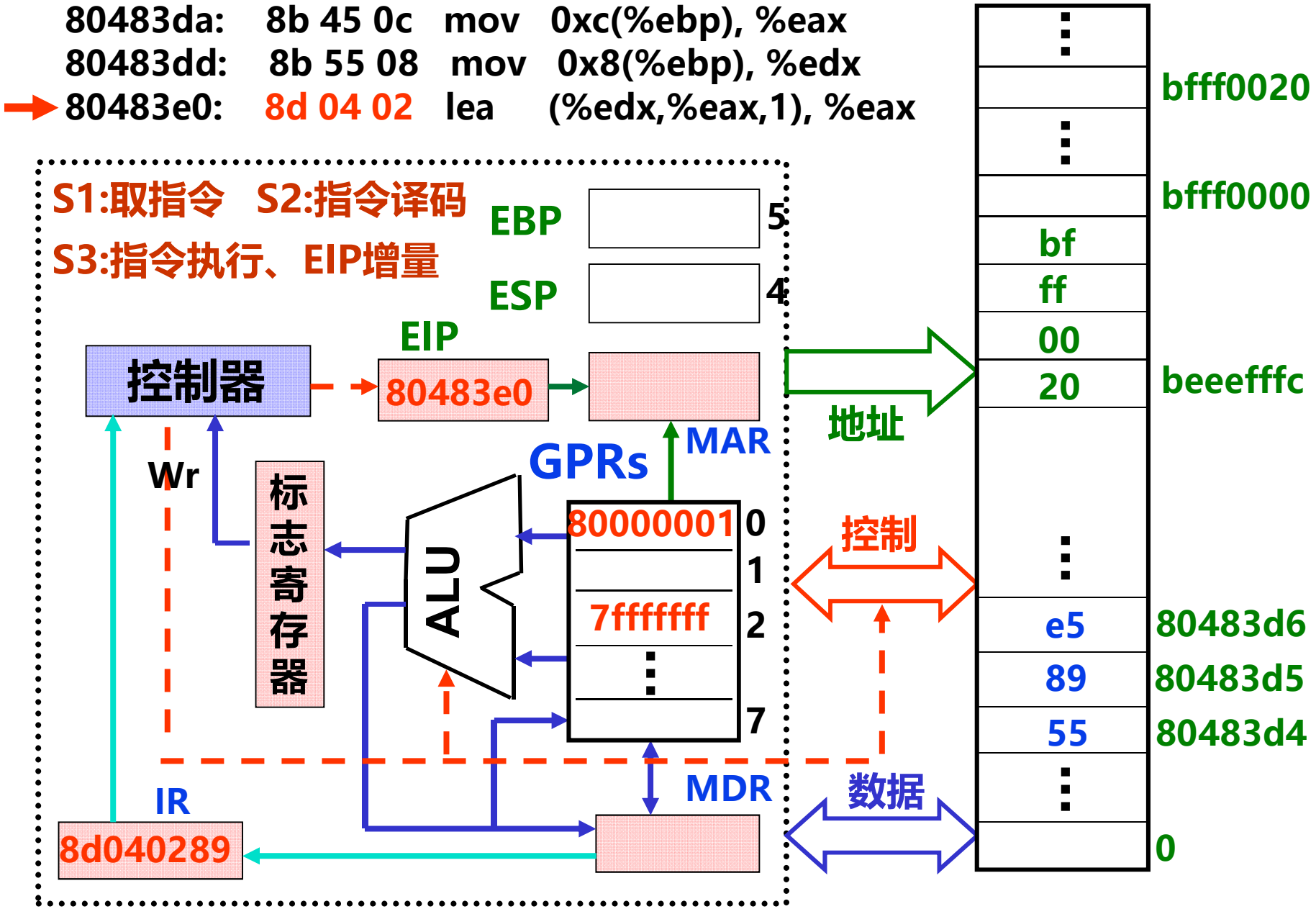


猜猜这是什么？

功能：R[eax]← R[edx]+R[eax]*1（执行前）



功能：R[eax]← R[edx]+R[eax]*1 （执行后）



lea指令执行的结果

```
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

咦，怎么会两个正数相加结果为负数呢？

为什么？

本周小结

- CPU的基本功能是周而复始地执行指令。
- CPU最基本的部分是数据通路和控制单元
  - 数据通路（datapath）中包含组合逻辑元件和存储信息的状态元件。
    - 组合逻辑（如加法器、ALU、扩展器、多路选择器以及状态元件的读操作逻辑等）用于对数据进行处理；
    - 状态元件包括触发器、寄存器和存储器等，用于对指令执行的中间状态或最终结果进行存储。
  - 控制单元（control unit）：对取出的指令进行译码，与指令执行得到的条件标志或当前机器的状态、时序信号等组合，生成对数据通路进行控制的控制信号，如读信号Rd、写信号Wr、ALU控制信号ALUctr等。
- 指令执行过程主要包括取指、译码、取数、运算、存结果。
- 通常把取出并执行一条指令的时间称为指令周期，它由机器周期或直接由时钟周期组成。现代计算机已经没有机器周期的概念。
- 现代计算机的每个指令周期直接由一个或若干个时钟周期（节拍）组成。
- 时钟信号是CPU中用于控制同步的信号。
- 每条指令功能不同，因此每条指令执行时数据在数据通路中所经过的部件和路径也可能不同。但是，每条指令在取指令阶段都一样。