程序的执行

程序执行和指令执行概述

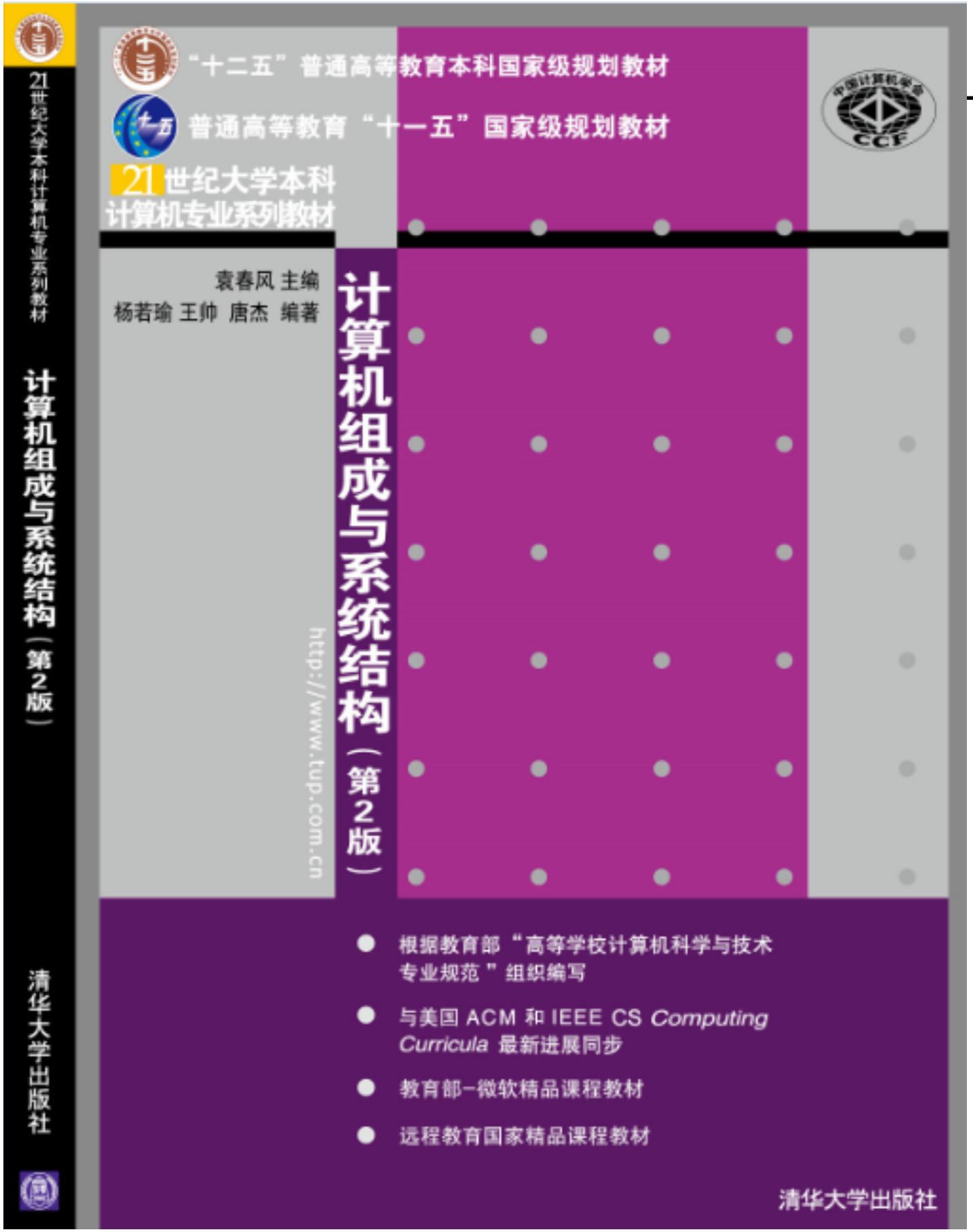
数据通路基本结构和工作原理

流水线方式下指令的执行

程序的执行机制

- 主要教学目标
  - 理解CPU如何控制程序的执行流
  - 了解一条指令的执行过程
  - 了解CPU的主要功能和内部结构
  - 了解数据通路的基本组成和定时方式
  - 理解指令执行时数据通路中信息的流动过程
  - 了解指令流水线的基本概念
  - 了解内部异常和外部中断的基本概念

简要介绍，详细内容参看《计算机组成与系统结构（第2版）》



教材特点

1. 强调软件和硬件的关联
2. 细化流水线CPU设计
3. 注重用实例图表阐述概念
4. 提供丰富的教辅资源

提供配套的辅助教材：

《计算机组成与系统结构习题解答与教学指导》

第2版的改进部分

程序的执行机制

- 分以下三个部分介绍
  - 第一讲：程序执行概述
    - 程序及指令的执行过程
    - CPU的基本功能和基本组成
  - 第二讲：数据通路基本结构和工作原理
    - 数据通路基本结构
    - 数据通路的时序控制
    - 数据通路基本工作原理
  - 第三讲：流水线方式下指令的执行
    - 指令流水线的基本原理
    - 适合流水线的指令集特征
    - CISC和RISC风格指令集
    - 指令流水线的实现
    - 高级流水线实现技术

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

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;
}
```

main：

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

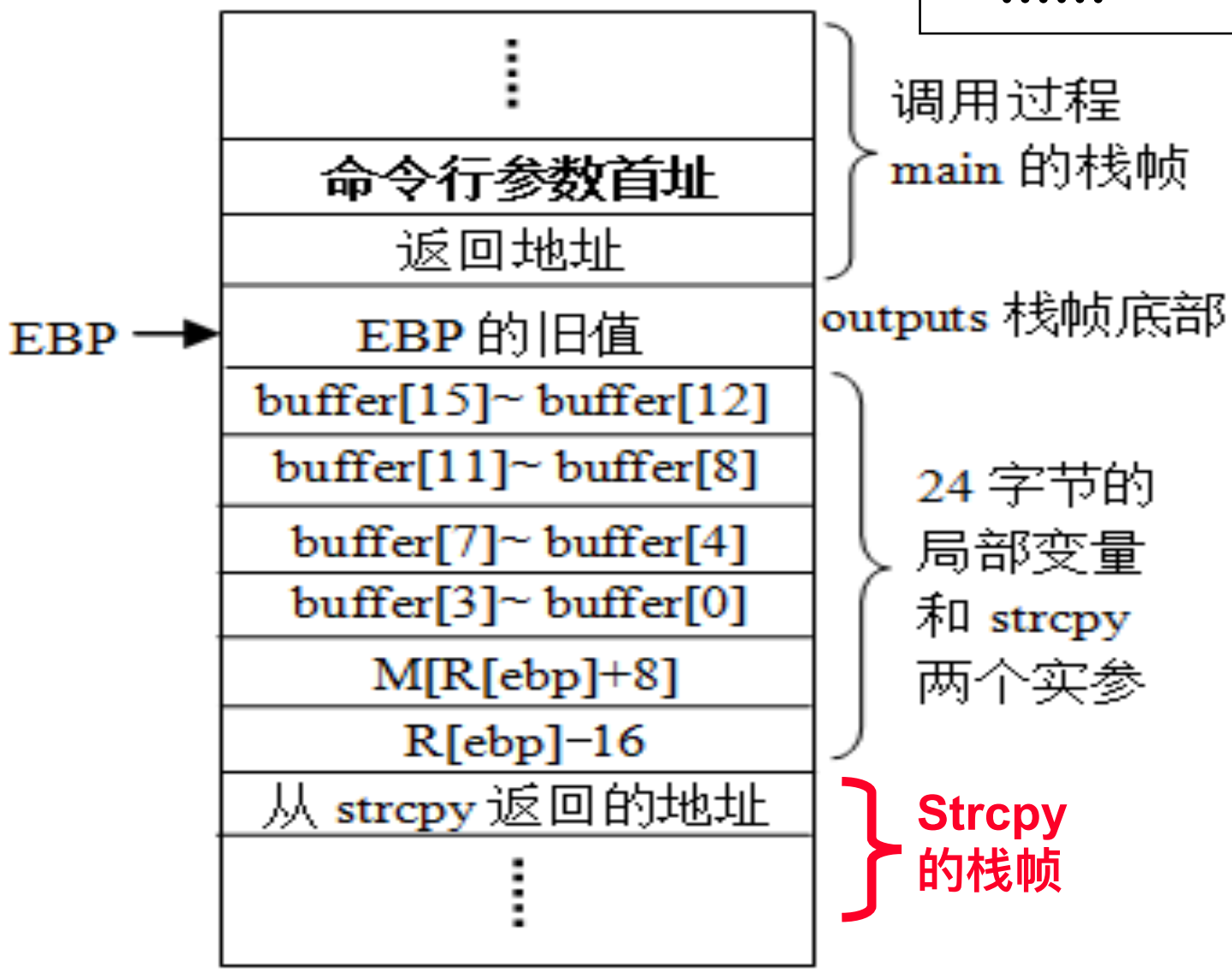
outputs：

```
……
call strcpy
……
call printf
……
ret
```

strcpy:

程序执行流：

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



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
```

双击此处添加文本

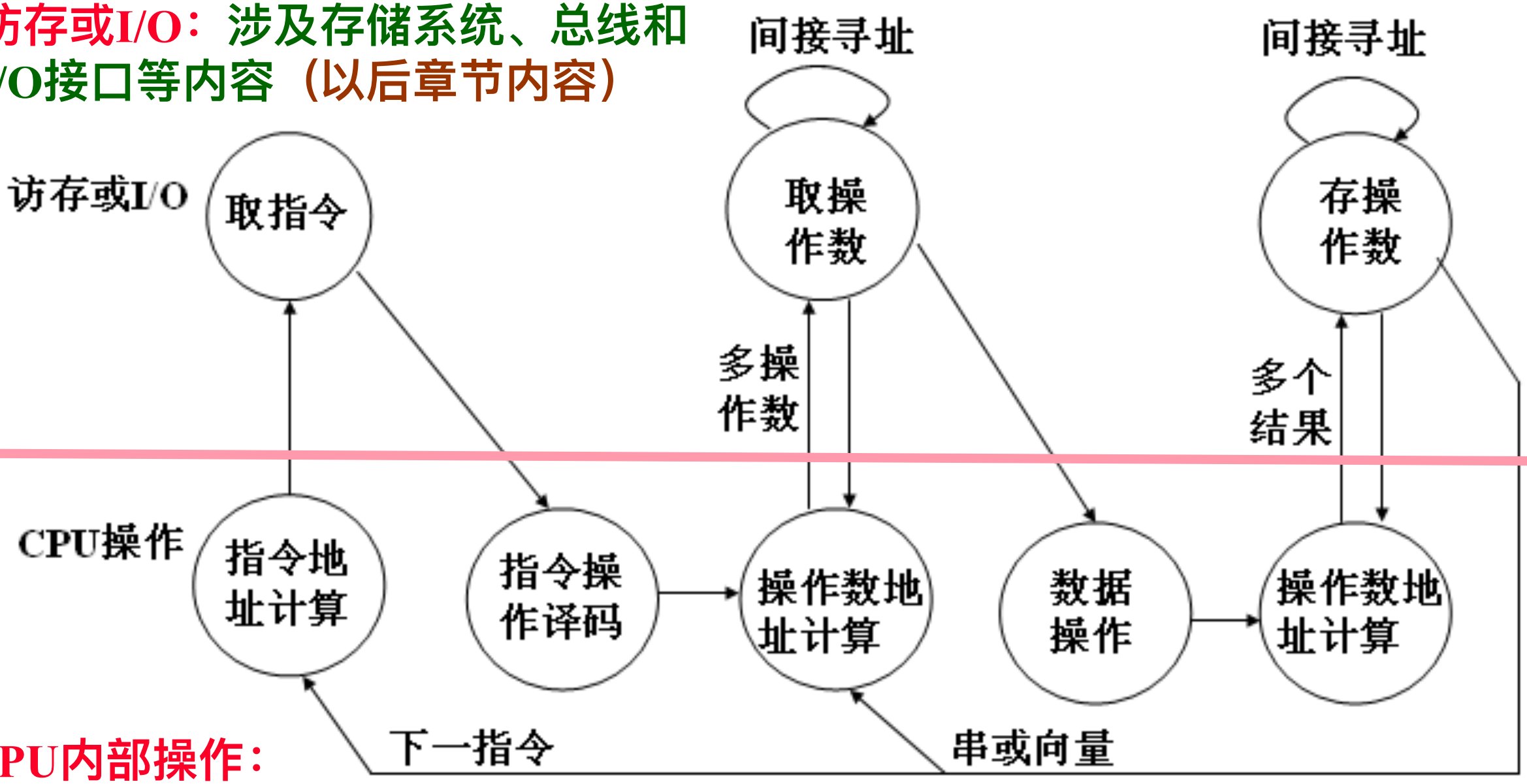
- 对于add函数
  - ✓ 指令按顺序存放在0x08048394开始的存储空间。
  - ✓ 各指令长度可能不同，如push、pop和ret指令各占一个字节，第2行mov指令占两个字节，第3行mov指令和第4行add指令各占3字节。
  - ✓ 各指令对应的0/1序列含义有不同的规定，如“push %ebp”指令为01010101B，其中01010为push指令操作码，101为EBP的编号，“pop %ebp”为01011101B，其中01011为pop指令的操作码。

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？

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

“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

执行阶段

指令执行过程

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

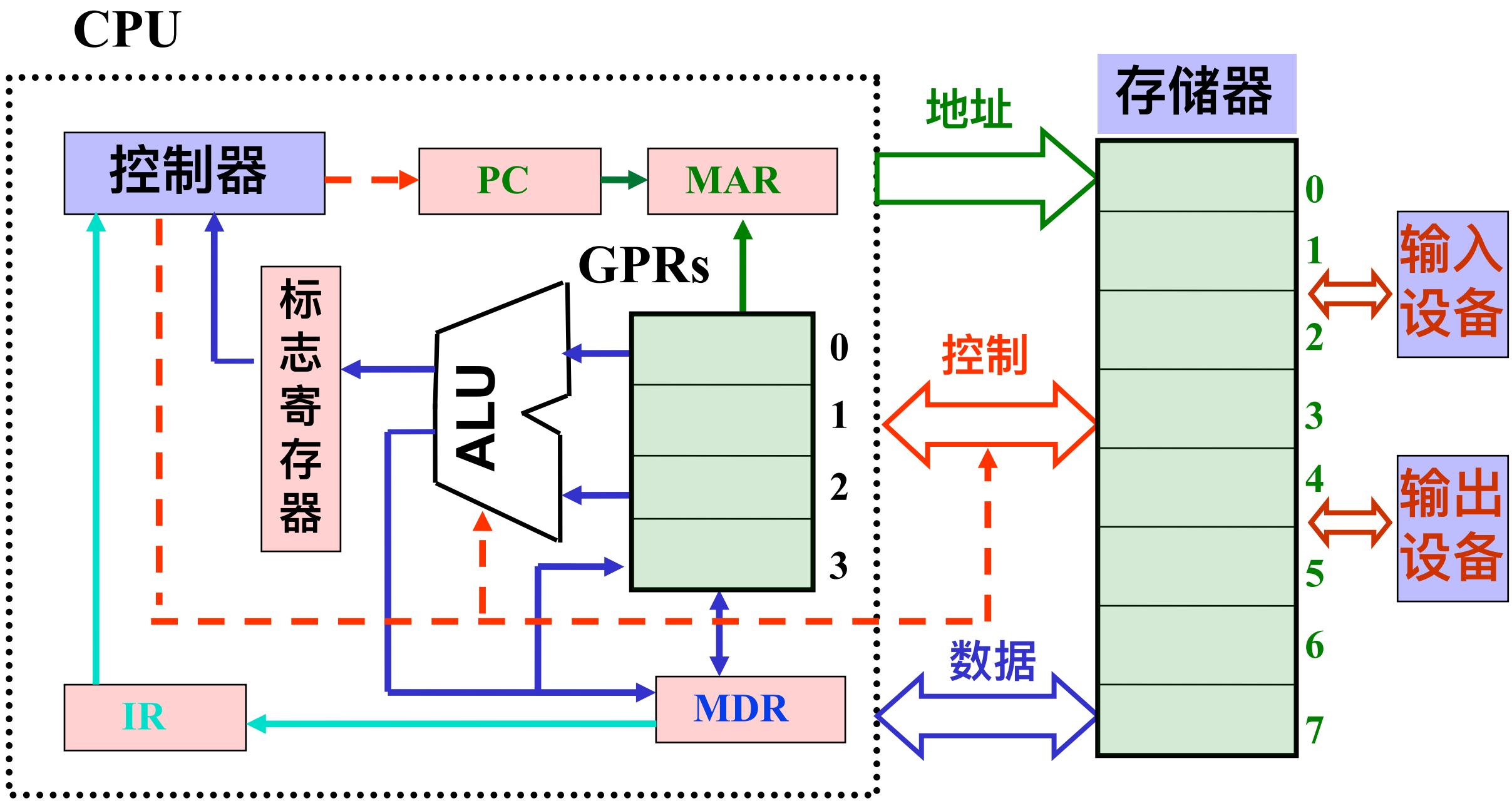
  描述语言称为寄存器传送语言RTL (Register Transfer Language)

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



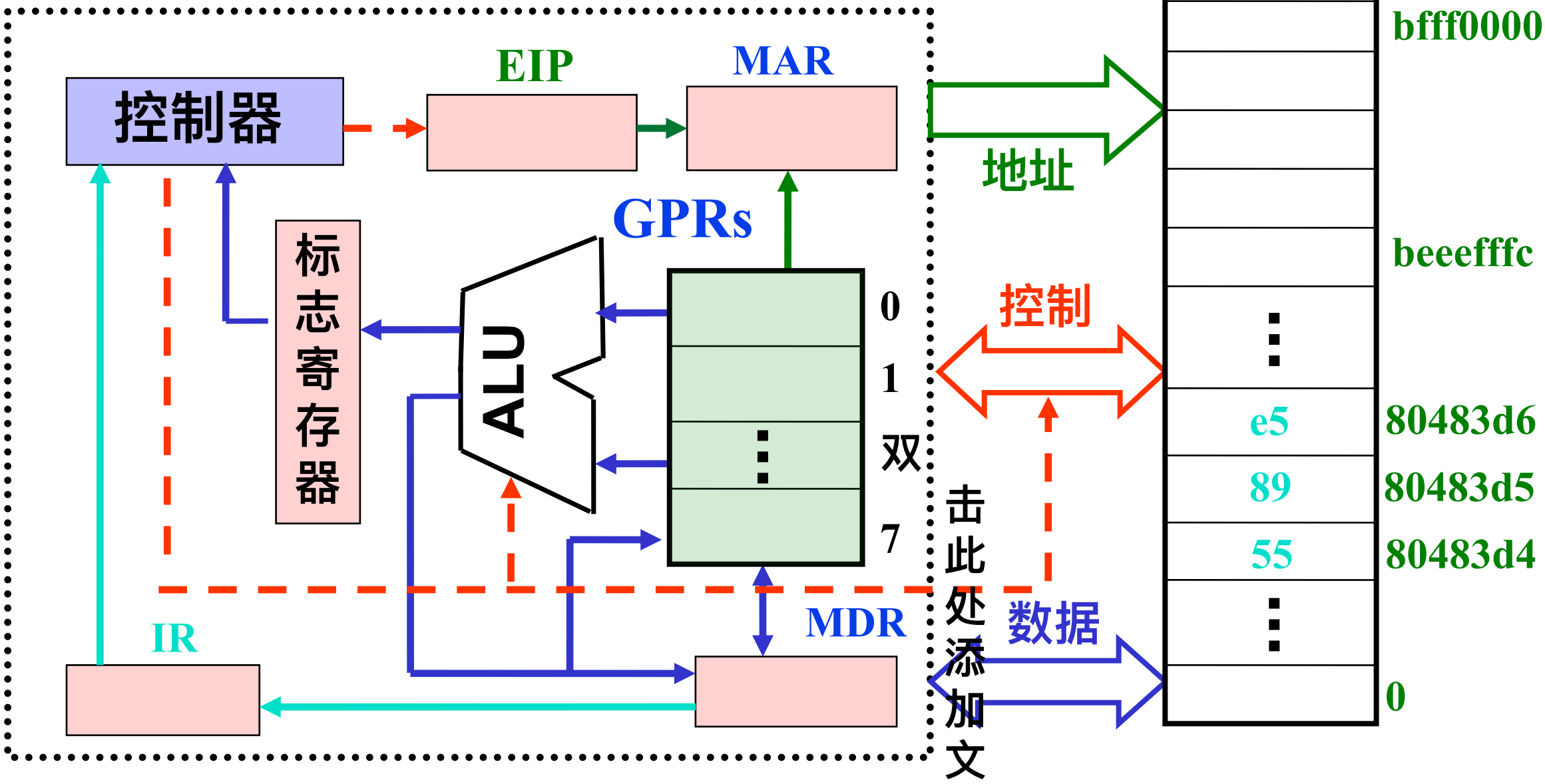
回顾： IA-32的体系结构是怎样的呢?

8个GPR（0~7），一个EFLAGs，PC为EIP

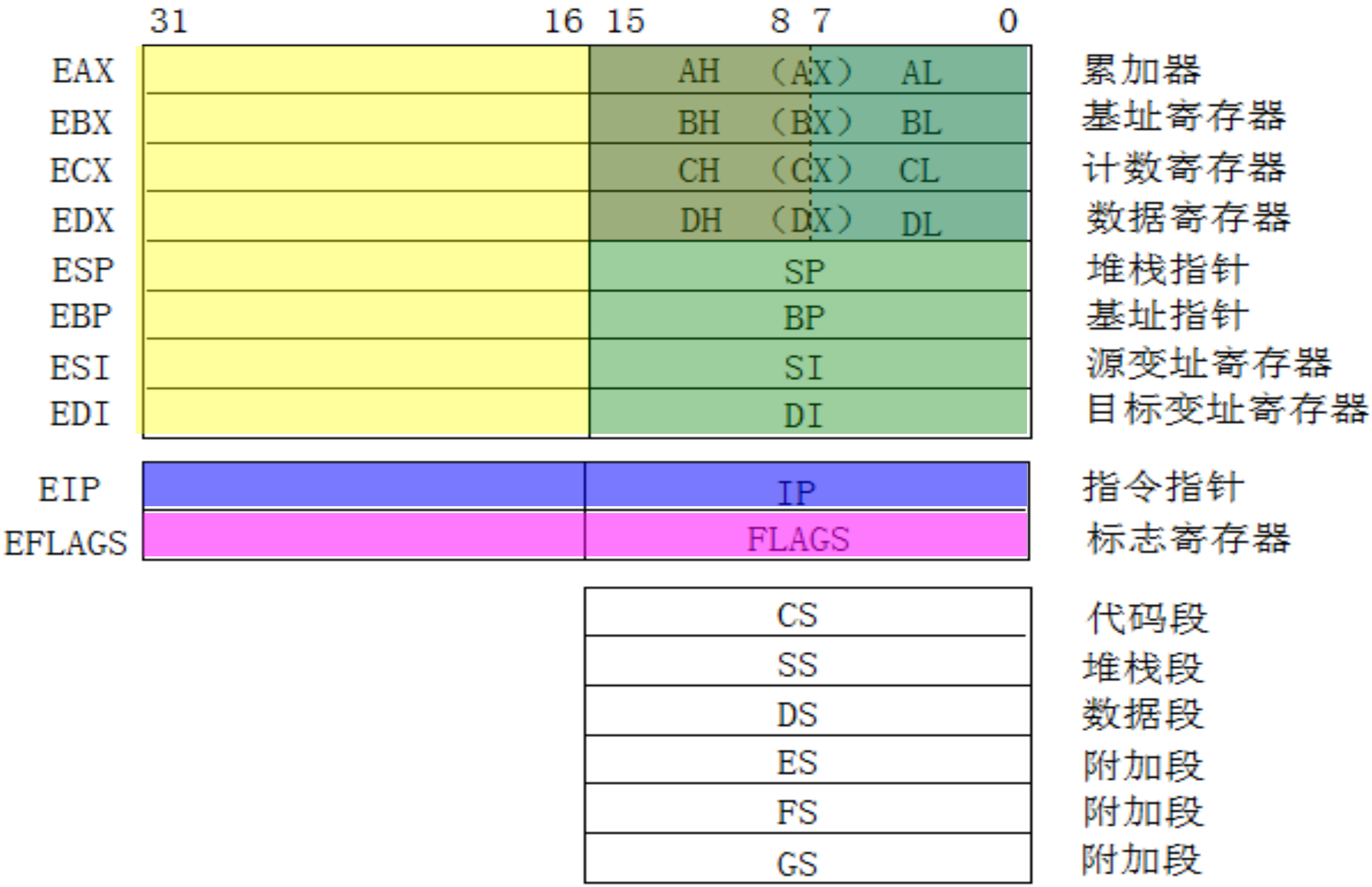
可寻址空间4GB（编号为0~0xFFFFFFFF）

指令格式变长，操作码变长

由若干字段（OP、Mod、SIB等）组成



回顾： IA-32的寄存器组织



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

PA中模拟的IA-32的寄存器组织

```
typedef struct{
union{
    struct {          uint32_t  eax;
                      uint32_t  ecx;
                      uint32_t  edx;
                      uint32_t  ebx;
                      uint32_t  esp;
                      uint32_t  ebp;
                      uint32_t  esi;
                      uint32_t  edi;};
    union{
                      uint32_t  _32;
                      uint16_t  _16;
                      uint8_t  _8[2];
            } gpr[8];
   };
swaddr_t  eip;
} CPU_state;

extern CPU_state cpu;
enum { R_EAX, R_ECX, R_EDX, R_EBX, R_ESP, R_EBP, R_ESI, R_EDI };
enum { R_AX, R_CX, R_DX, R_BX, R_SP, R_BP, R_SI, R_DI };
enum { R_AL, R_CL, R_DL, R_BL, R_AH, R_CH, R_DH, R_BH };

#define reg_l(index)   (cpu.gpr[index]._32)
#define reg_w(index)  (cpu.gpr[index]._16)
#define reg_b(index)  (cpu.gpr[index & 0x3]._8[index >> 2])
```

回顾： IA-32的标志寄存器

31-22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	O	D	I	T	S	Z	0	A	0	P	1	C

8086

80286/386

- 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： IA-32的寻址方式

- 寻址方式
  - 根据指令给定信息得到操作数或操作数地址
- 操作数所在的位置
  - 指令中：立即寻址
  - 寄存器中：寄存器寻址
  - 存储单元中（属于存储器操作数，按字节编址）：其他寻址方式
- 存储器操作数的寻址方式与微处理器的工作模式有关
  - 两种工作模式：实地址模式和保护模式
- 实地址模式（基本用不到）
  - 为与8086/8088兼容而设，加电或复位时
  - 寻址空间为1MB，20位地址：(CS)<<4+(IP)
- 保护模式（需要掌握）
  - 加电后进入，采用虚拟存储管理，多任务情况下隔离、保护
  - 80286以上高档微处理器最常用的工作模式
  - 寻址空间为2^{32}B，32位线性地址分段（段基址+段内偏移量）

回顾：保护模式下的寻址方式

寻址方式	说明
立即寻址	指令直接给出操作数
寄存器寻址	指定的寄存器R的内容为操作数
位移	LA=（SR）+A
基址寻址	LA=（SR）+（B）
基址加位移	LA=（SR）+（B）+A
比例变址加位移	LA=（SR）+(I)×S + A
基址加变址加位移	LA=（SR）+（B）+（I）+A
基址加比例变址加位移	LA=（SR）+（B）+(I)×S + A
相对寻址	LA=(PC)+A

存储器操作数

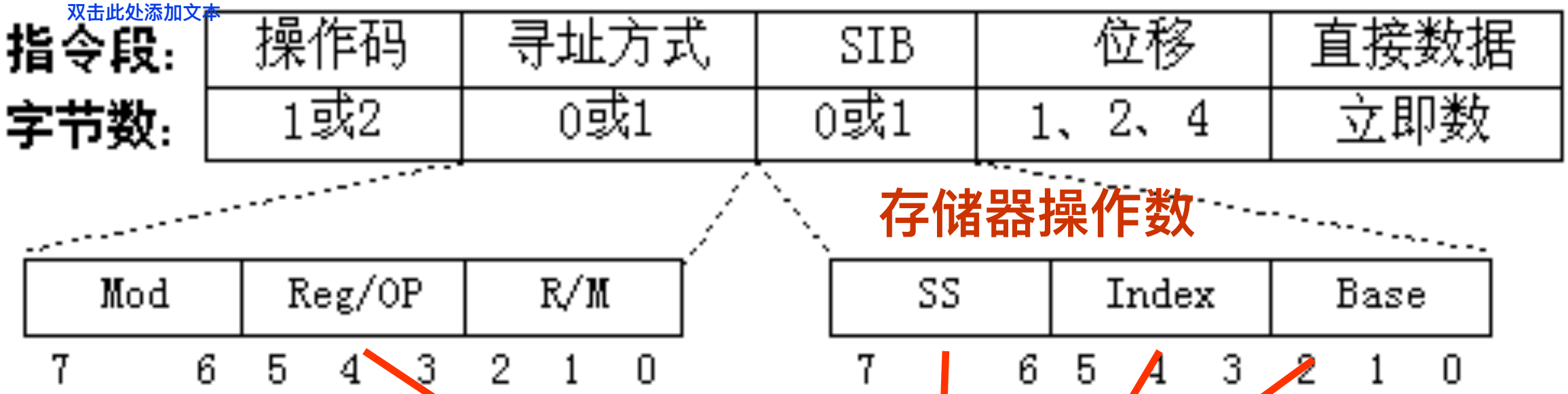
跳转目标指令地址

注：LA:线性地址　(X):X的内容　SR:段寄存器　PC:程序计数器　R:寄存器
A:指令中给定地址段的位移量　B:基址寄存器　I:变址寄存器　S:比例系数

- SR段寄存器（间接）确定操作数所在段的段基址
- 有效地址给出操作数在所在段的偏移地址
- 寻址过程涉及到“分段虚拟管理方式”，将在第6章讨论

回顾： IA-32机器指令格式

双击此处添加文本



位移量和立即数都可以是：1B/2B/4B

SIB中基址B和变址I都可是8个GRS中任一个；SS给出比例因子

操作码：opcode; W：与机器模式（16 / 32位）一起确定寄存器位数（AL / AX / EAX）; D：操作方向（确定源和目标）

寻址方式（ModRM字节）： mod、r/m、reg/op三个字段与w字段和机器模式（16/32）一起确定操作数所在的寄存器编号或有效地址计算方式

8d 04 02　lea (%edx,%eax,1), %eax

1000 1101 00 000 100 00 000 010

PA中模拟的IA-32指令的ModRM和SIB

```
typedef union {
  struct {
        uint8_t R_M         :3;
        uint8_t reg         :3;
        uint8_t mod         :2;
  };
  struct {
        uint8_t dont_care   :3;
        uint8_t opcode      :3;
  };
  uint8_t val;
} ModR_M;

typedef union {
  struct {
        uint8_t base        :3;
        uint8_t index       :3;
        uint8_t ss          :2;
  };
  uint8_t val;
} SIB;
```

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” 结果

```
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
```

EIP←0x80483d4

OP

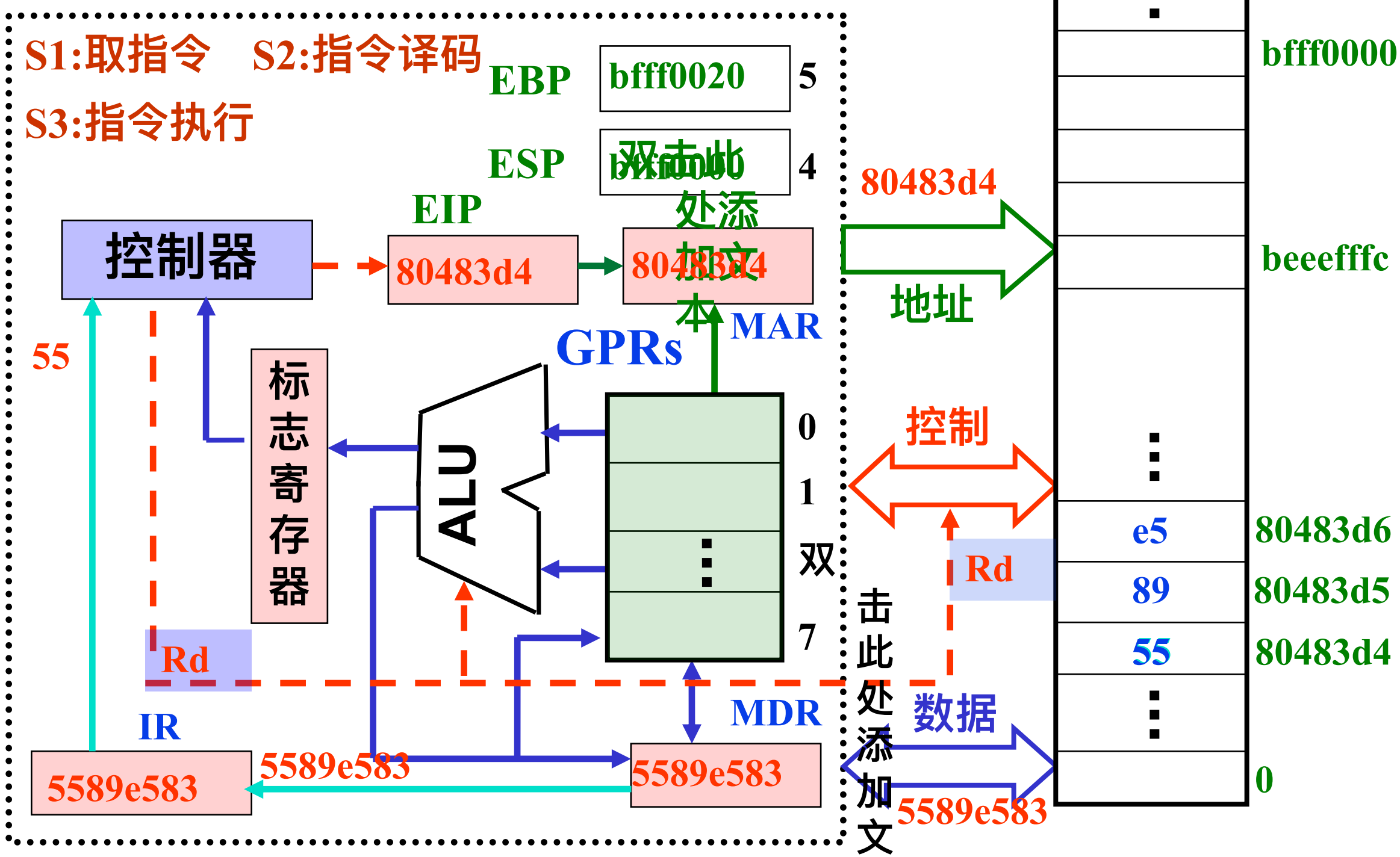
起始EIP=?

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

test代码从80483d4开始!

功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

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

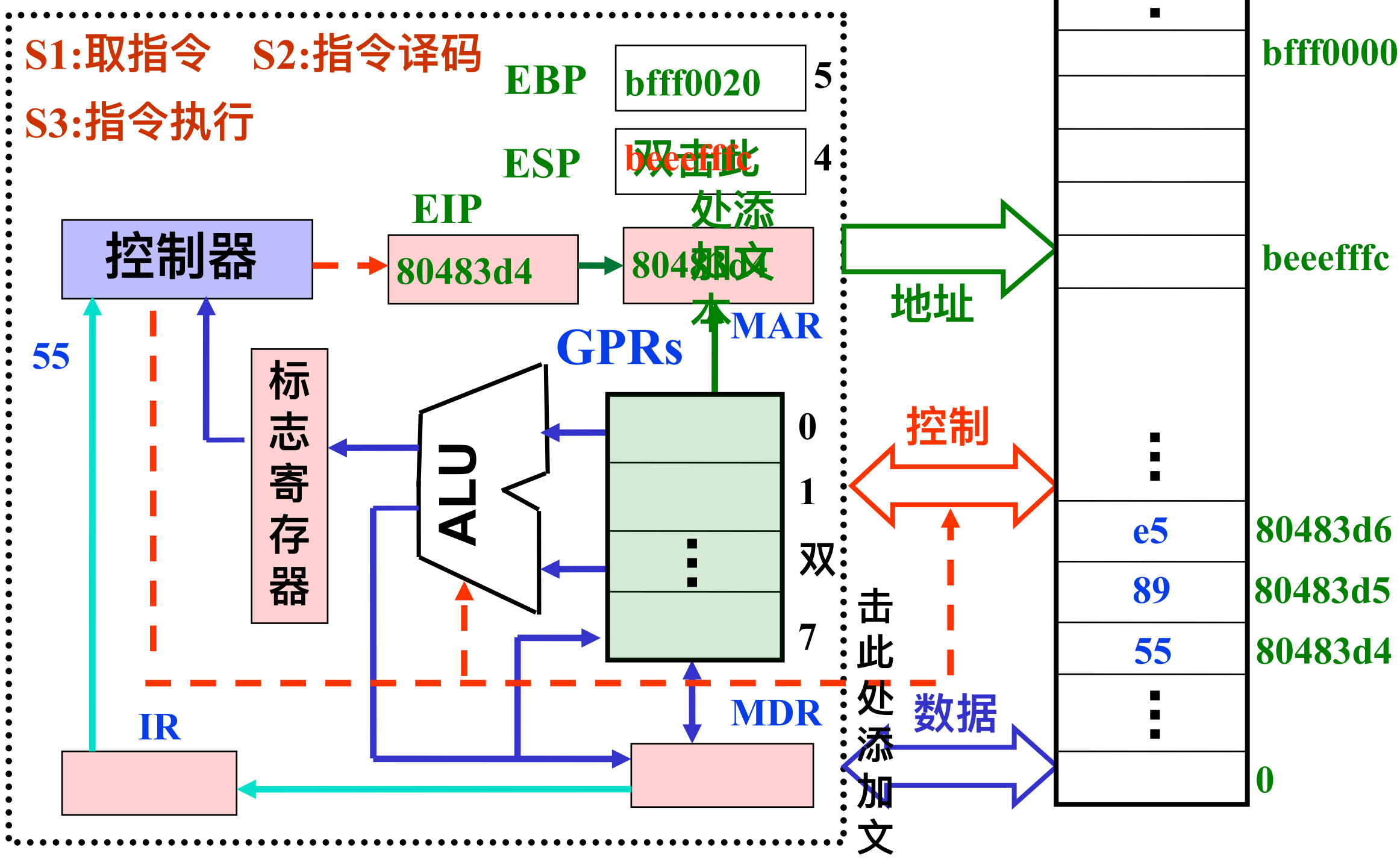


功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

080483d4 <add>:

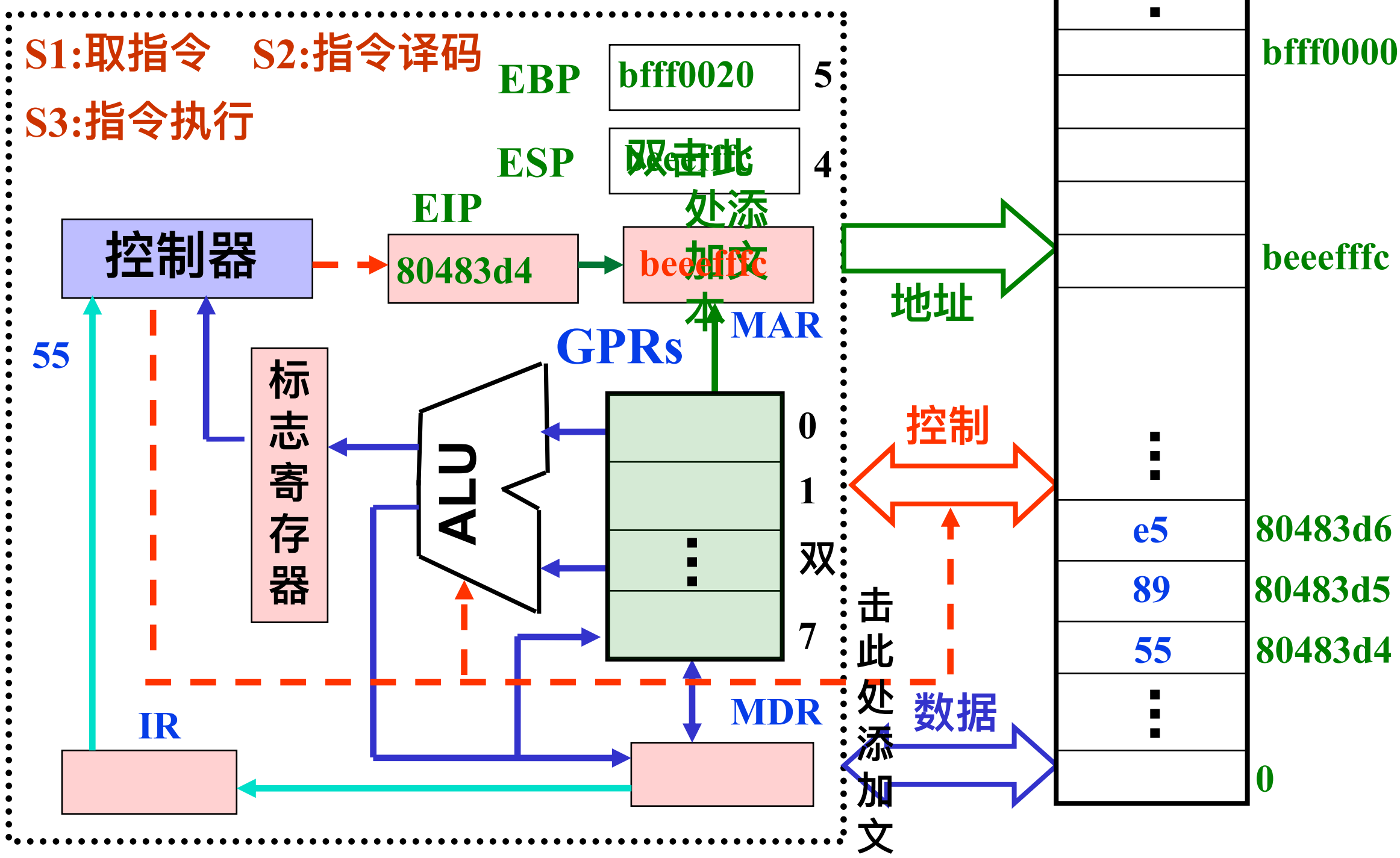
80483d4: 55 push %ebp

80483d5: 89 e5 mov %esp, %ebp



功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

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

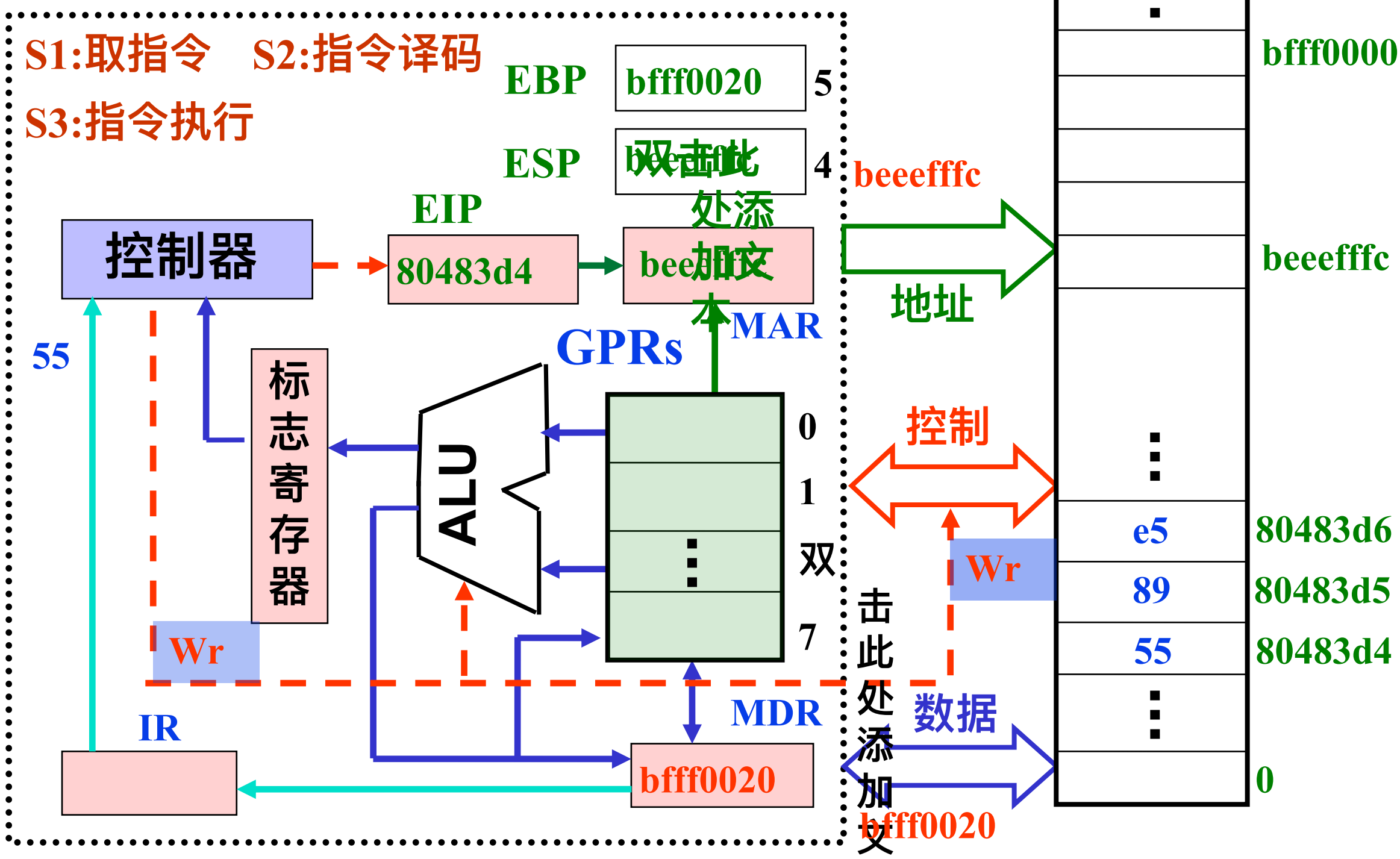


功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

080483d4 <add>:

80483d4: 55 push %ebp

80483d5: 89 e5 mov %esp, %ebp

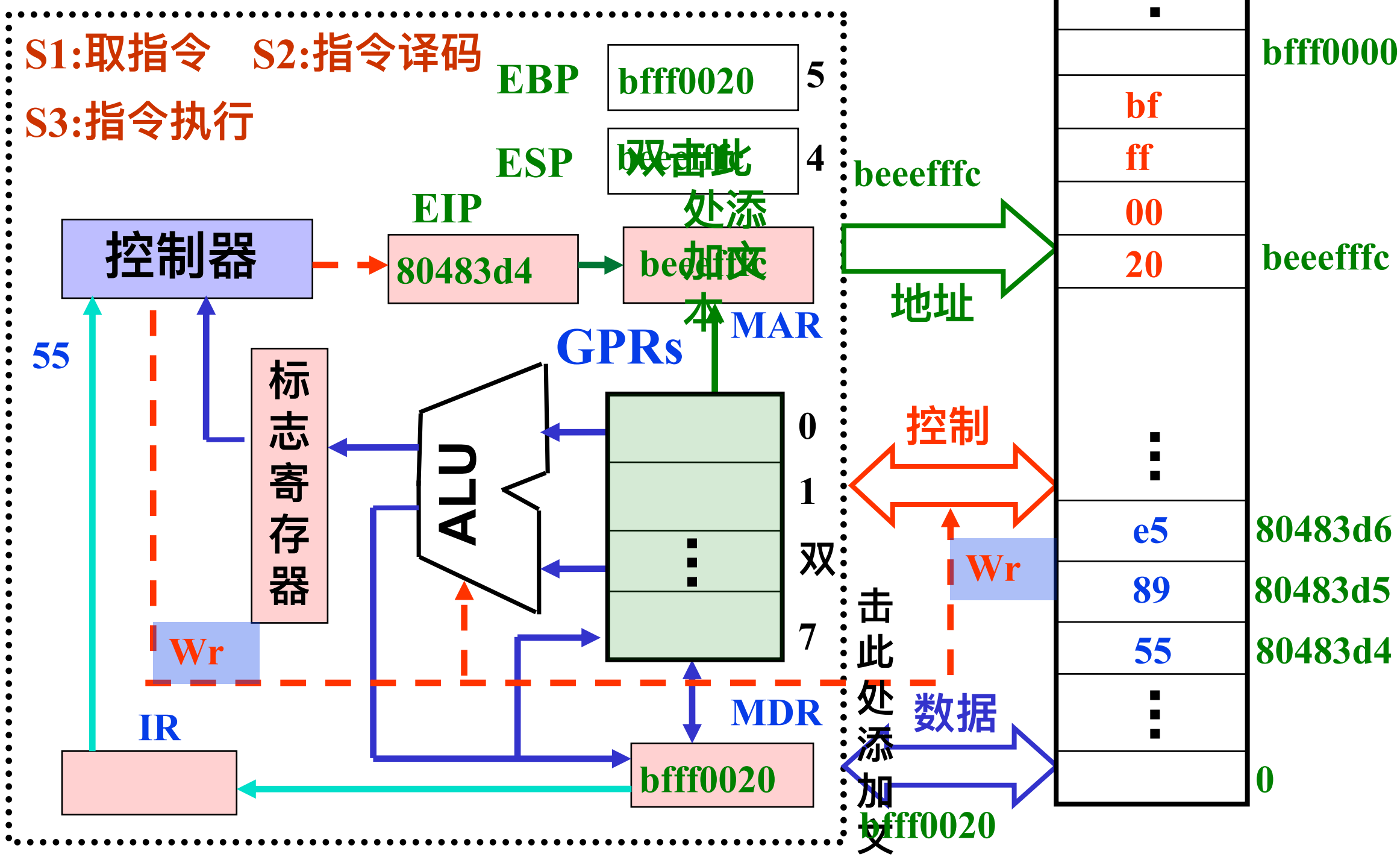


功能：R[esp]← R[esp]-4，M[R[esp]] ←R[ebp]

080483d4 <add>:

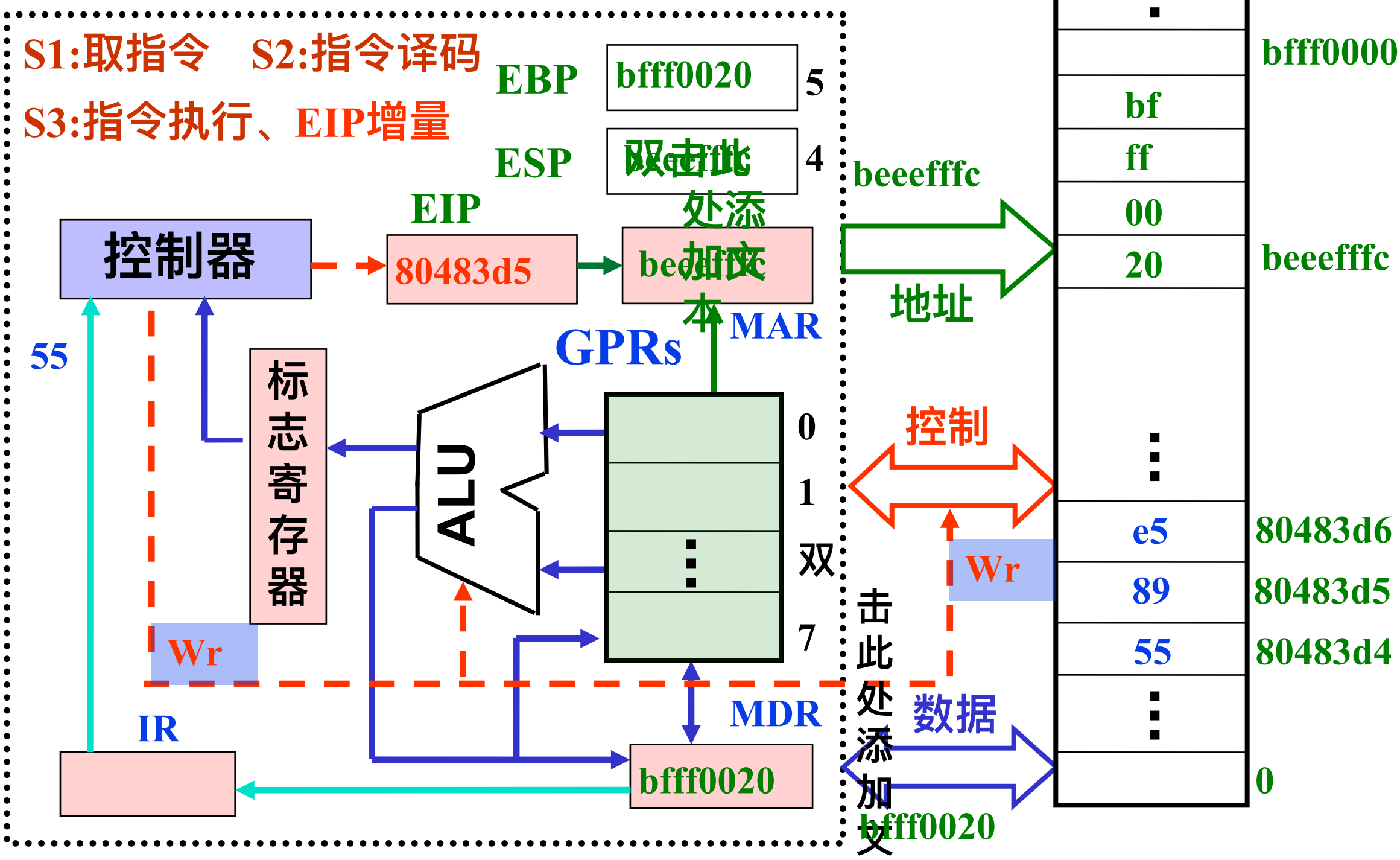
80483d4: 55 push %ebp

80483d5: 89 e5 mov %esp, %ebp



开始执行下一条指令

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



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   }
```

“objdump -d test” 结果

```
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
```

若 i= 2147483647，j=2，

则程序执行结果是什么?

每一步如何执行?

EIP←0x80483d4

EDX和EAX中各是什么?

R[edx]=i=0x7fffffff

R[eax]=j=0x2

OP

test代码从80483d4开始!

执行add时，起始EIP=?

回顾： IA-32的寄存器组织

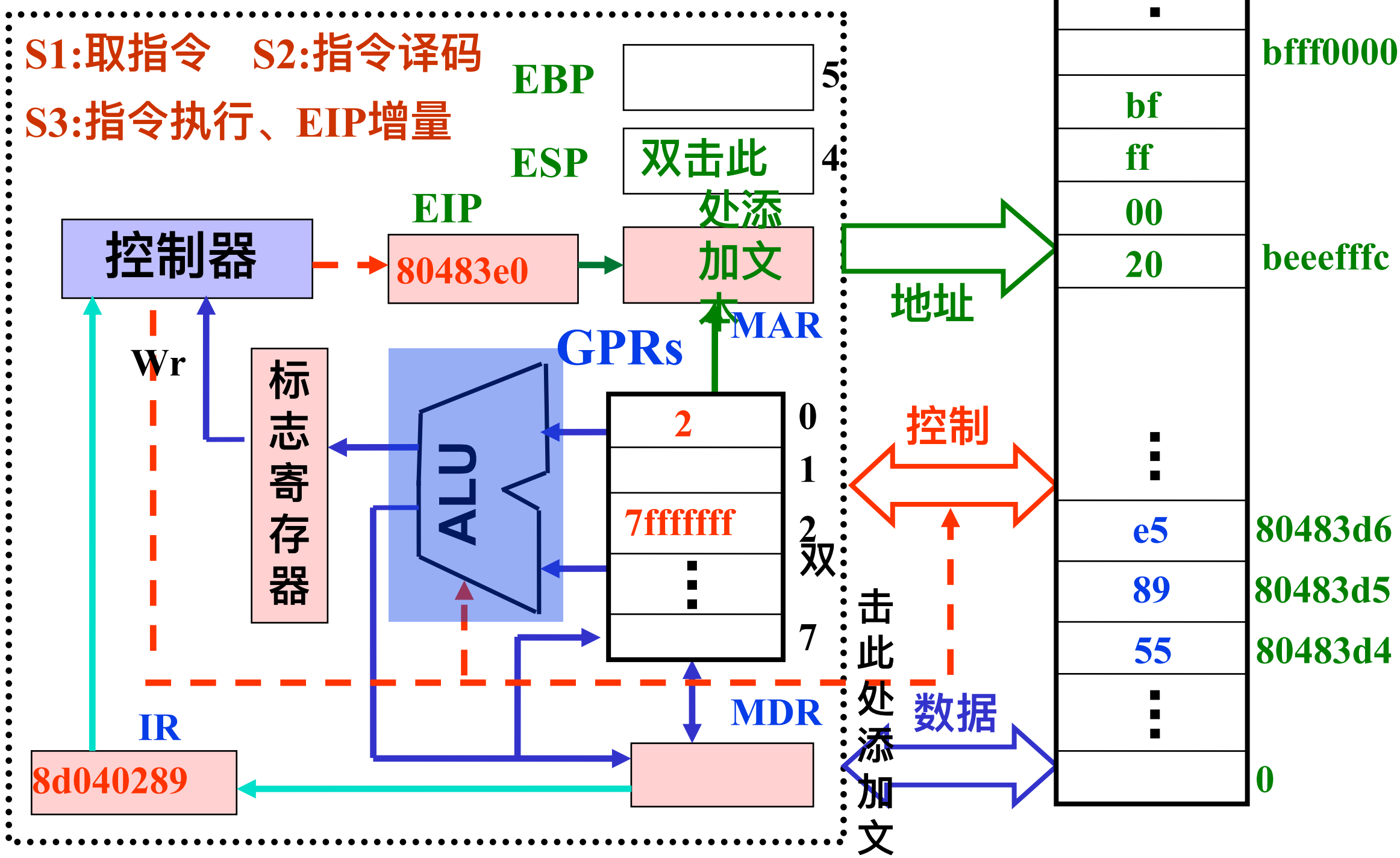
编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

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
```



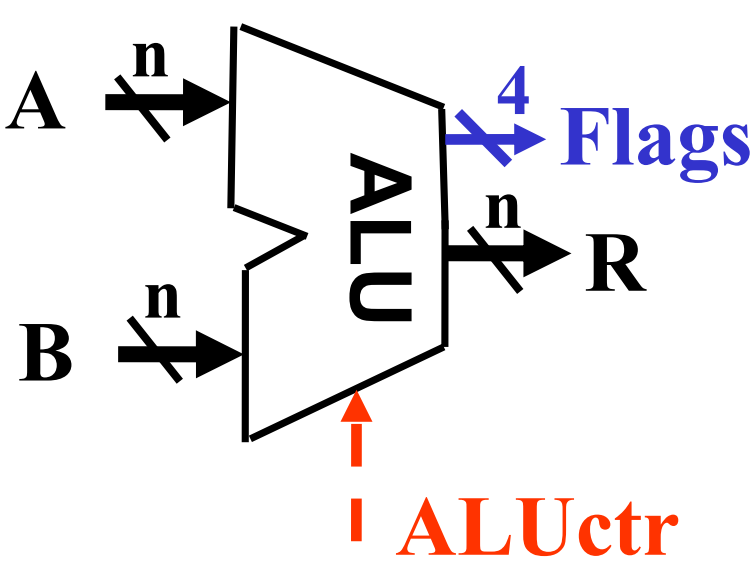
ALU长啥样呢?

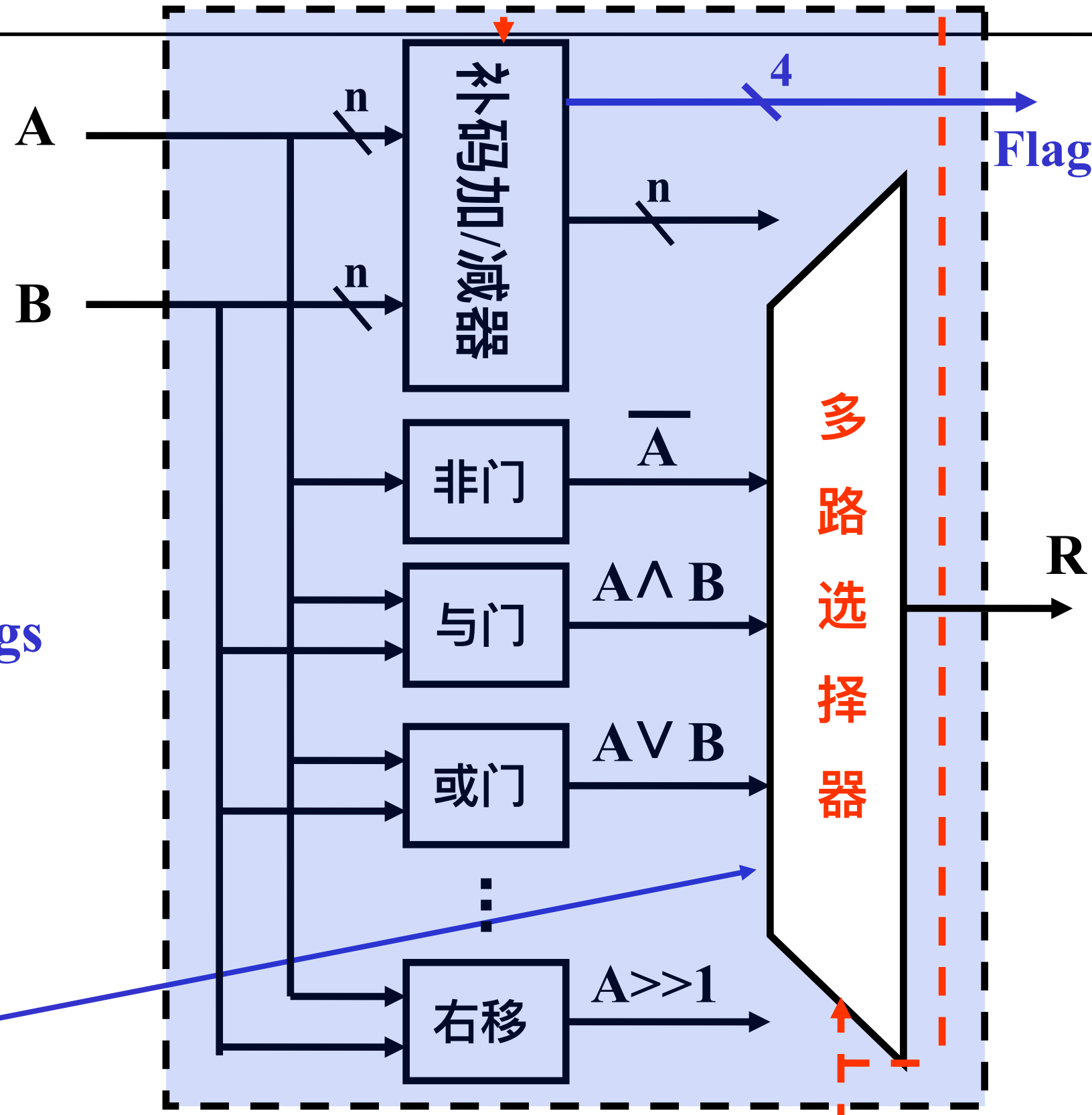
- 试想一下ALU中有哪些部件? （想想厨房做菜用什么工具？）
  - 补码加/减器（可以干什么？）
    - 带符号加、带符号减
    - 无符号加、无符号减
  - ~~乘法器~~? （为什么可以没有？）
    - 可用加/减+移位实现，也可有独立乘法器
    - 带符号乘和无符号乘是独立的部件
  - ~~除法器~~? （为什么可以没有？）
    - 可用加/减+移位实现，也可有独立除法器
    - 带符号除和无符号除是独立的部件
  - 各种逻辑运算部件（可以干什么？）
    - 非、与、或、非、前置0个数、前置1个数.....

大家能否画出ALU框图?

ALU结构原理

ALU的符号是什么样的?

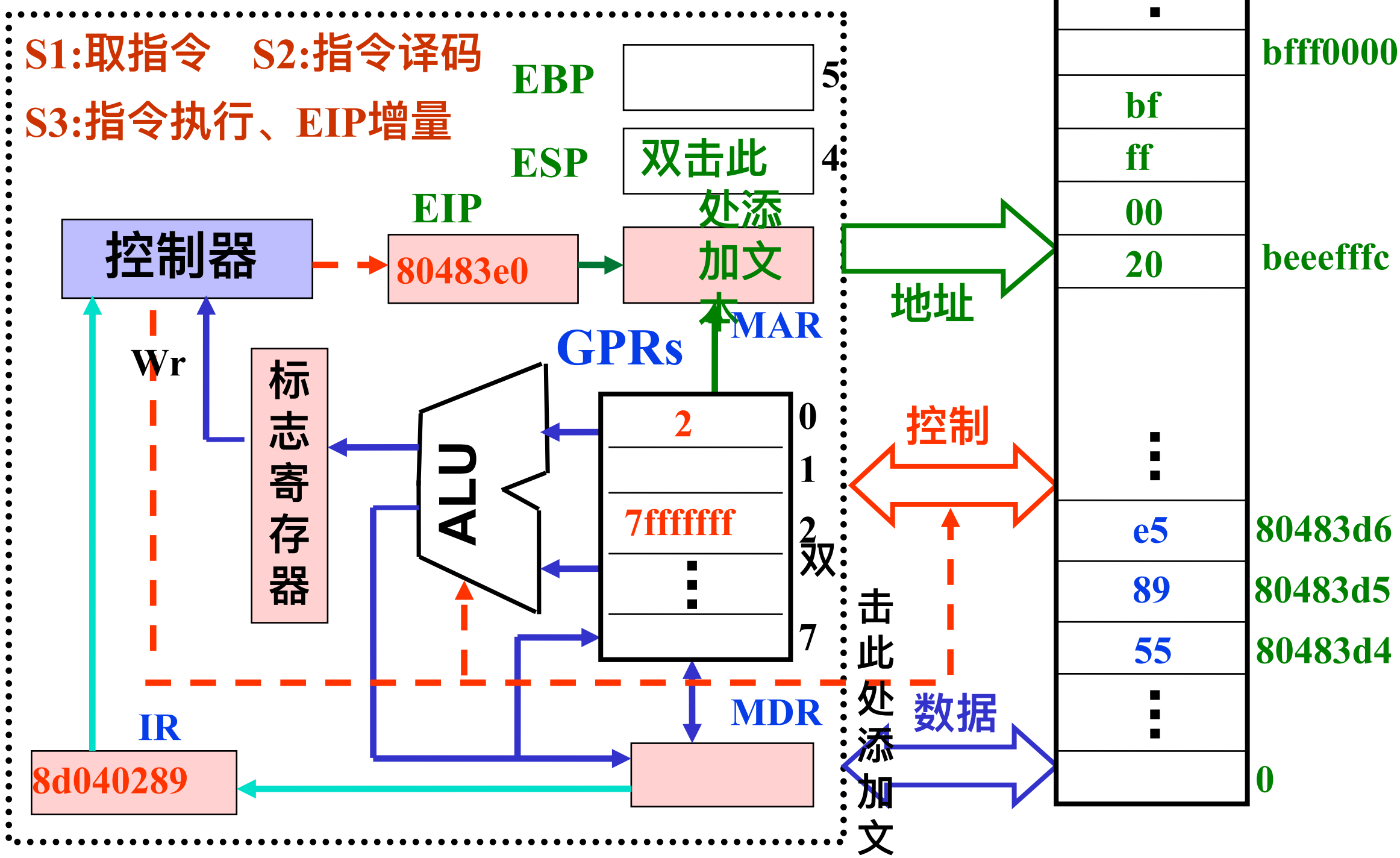




猜猜这是什么?

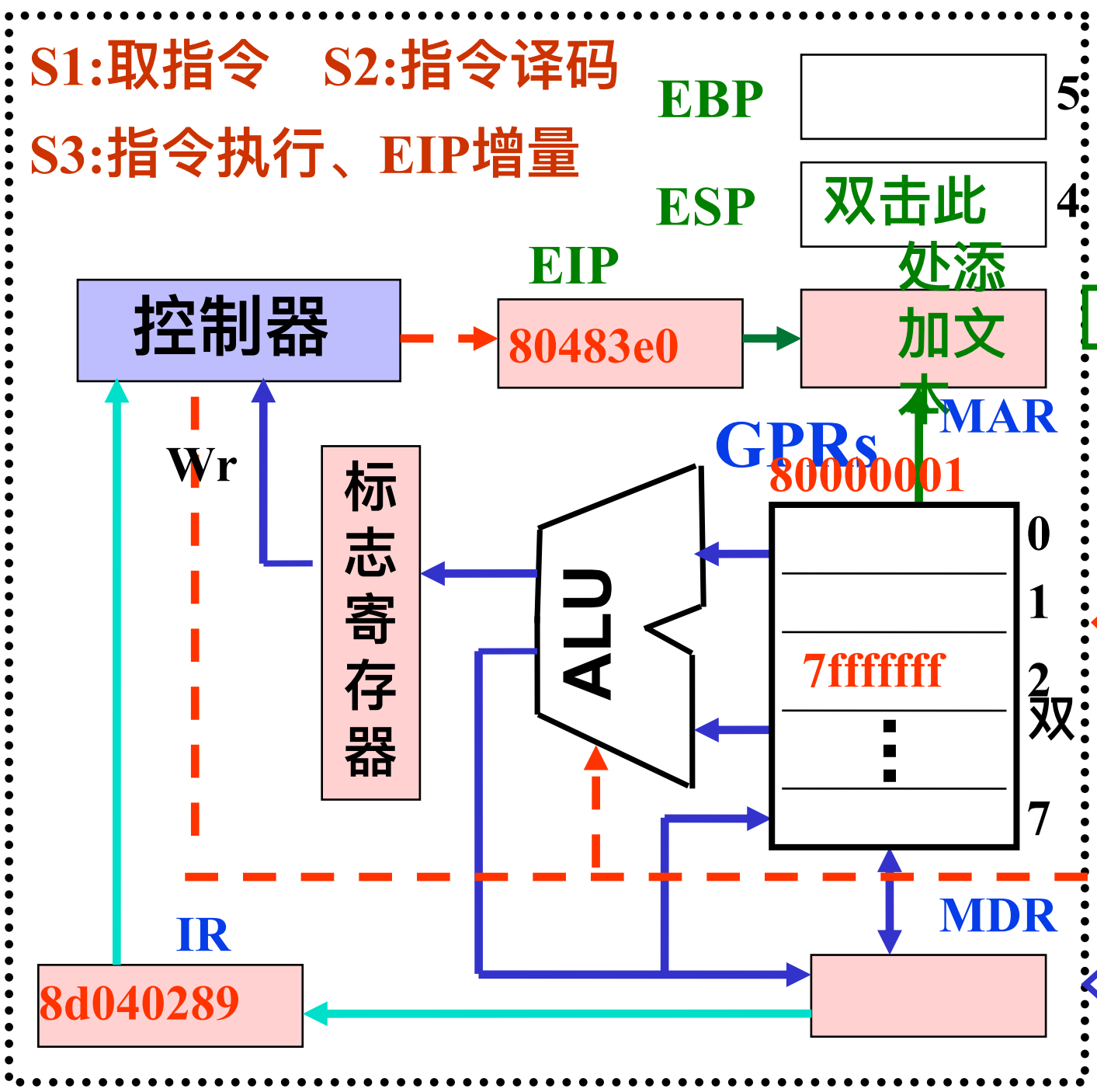
功能：R[eax]← R[edx]+R[eax]*1（执行前）

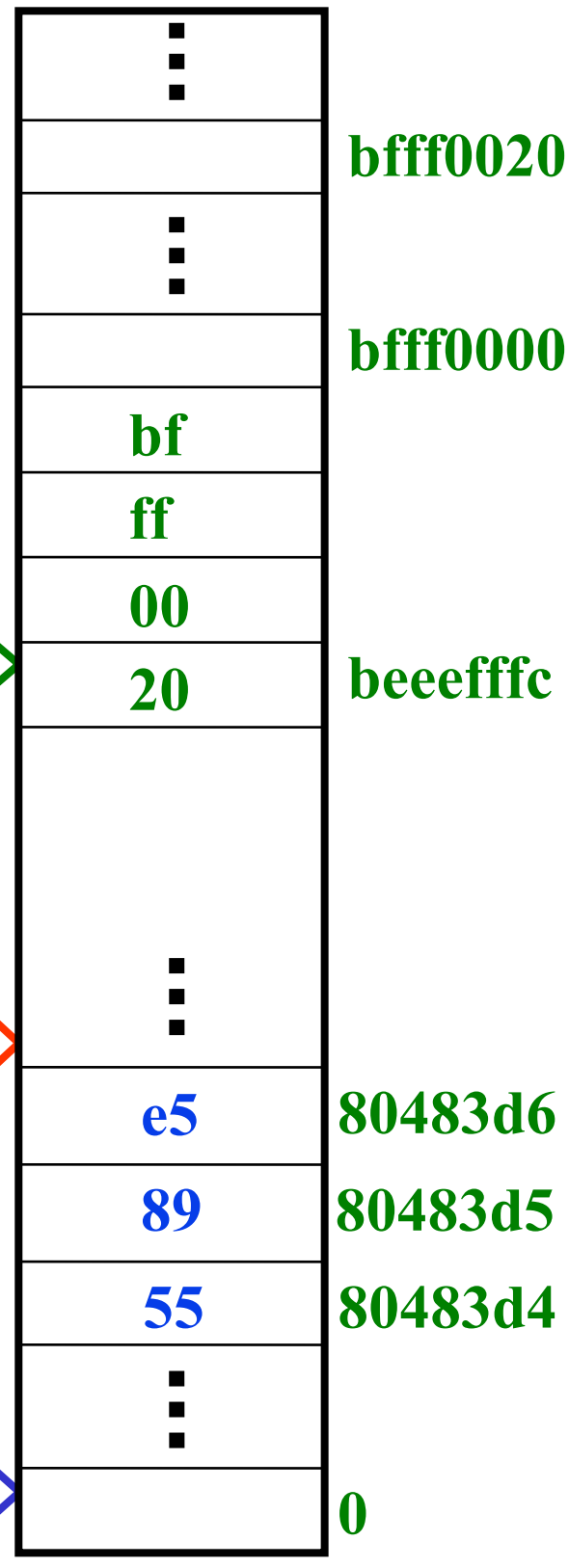
```
80483da:    8b 45 0c    mov   0xc(%ebp), %eax
80483dd:    8b 55 08    mov   0x8(%ebp), %edx
80483e0:    8d 04 02    lea   (%edx,%eax,1), %eax
```



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
```





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?

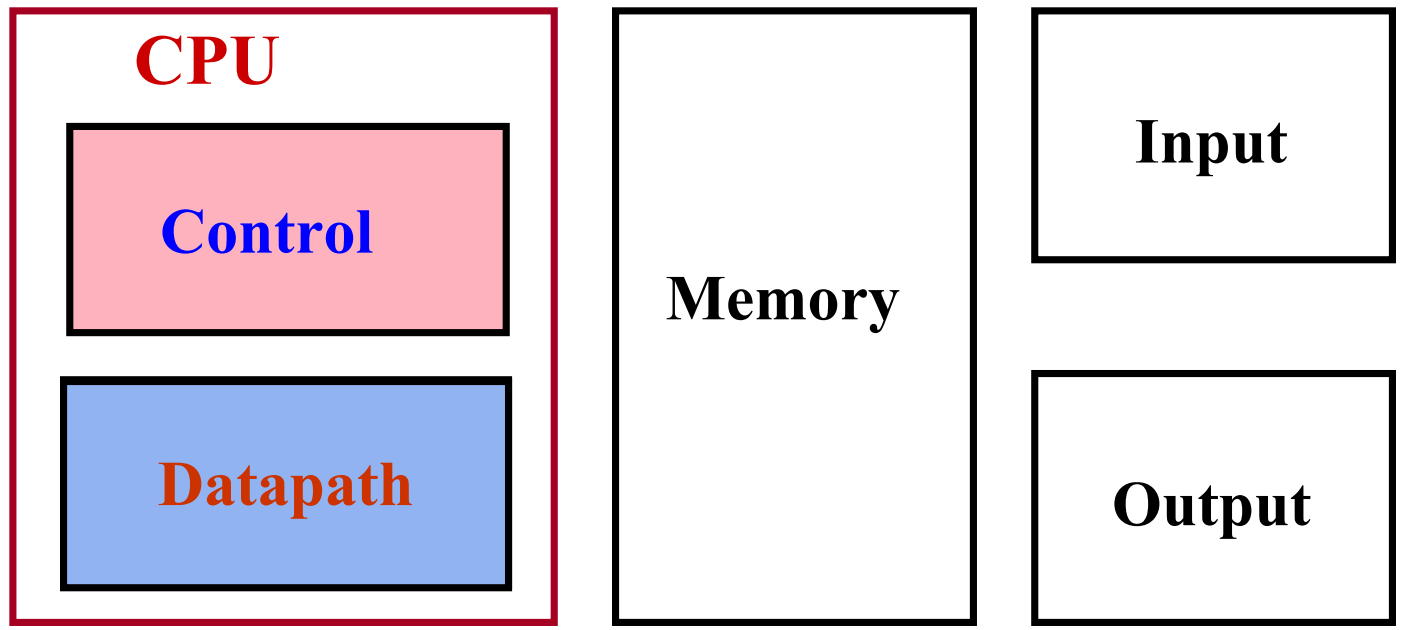
为什么?

程序的执行机制

- 分以下三个部分介绍
  - 第一讲：程序执行概述
    - 程序及指令的执行过程
    - CPU的基本功能和基本组成
  - 第二讲：数据通路基本结构和工作原理
    - 数据通路基本结构
    - 数据通路的时序控制
    - 数据通路基本工作原理
  - 第三讲：流水线方式下指令的执行
    - 指令流水线的基本原理
    - 适合流水线的指令集特征
    - CISC和RISC风格指令集
    - 指令流水线的实现
    - 高级流水线实现技术

数据通路的位置

- 计算机的五大组成部分：



- 什么是数据通路（DataPath）？
  - 指令执行过程中，数据所经过的路径，包括路径中的部件。它是指令的执行部件。
- 控制器（Control）的功能是什么？
  - 对指令进行译码，生成指令对应的控制信号，控制数据通路的动作。能对执行部件发出控制信号，是指令的控制部件。

数据通路的基本结构

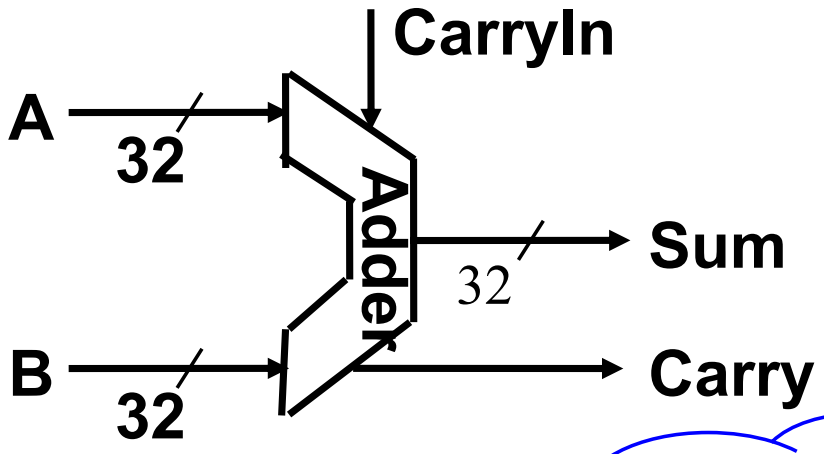
- 数据通路由两类元件组成
  - 组合逻辑元件（也称操作元件）
  - 时序逻辑元件（也称状态元件，存储元件）
- 元件间的连接方式
  - 总线连接方式
  - 分散连接方式
- 数据通路如何构成?
  - 由“操作元件”和“存储元件”通过总线方式或分散方式连接而成
- 数据通路的功能是什么?
  - 进行数据存储、处理、传送

因此，数据通路是由操作元件和存储元件通过总线方式或分散方式连接而成的进行数据存储、处理、传送的路径。

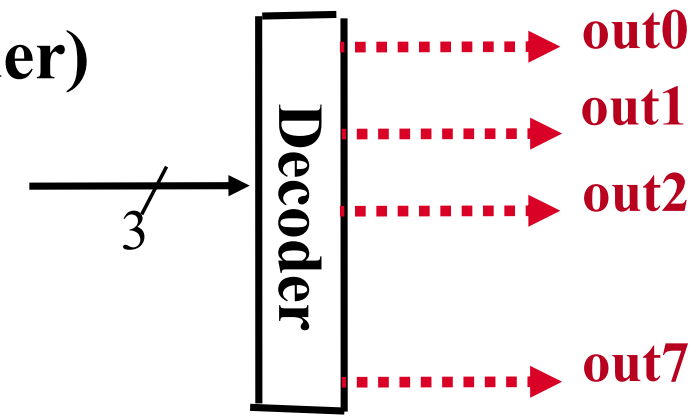
操作元件：组合逻辑电路

控制信号

- 加法器 (Adder)



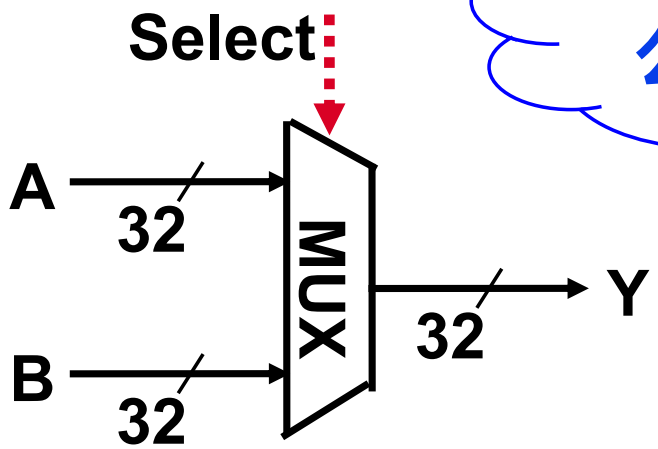
- 译码器 (Decoder)



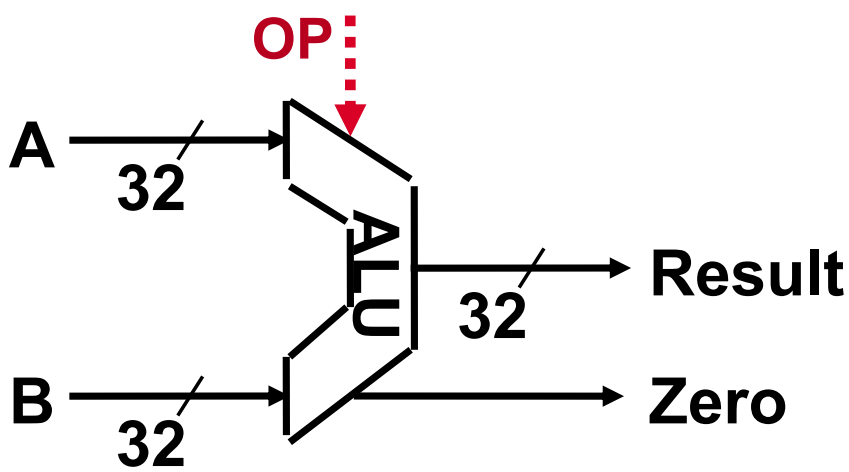
加法器需要什么控制信号?

何时要用到adder, ALU, MUX or Decoder?

多路选择器 (MUX)



算逻部件 (ALU)



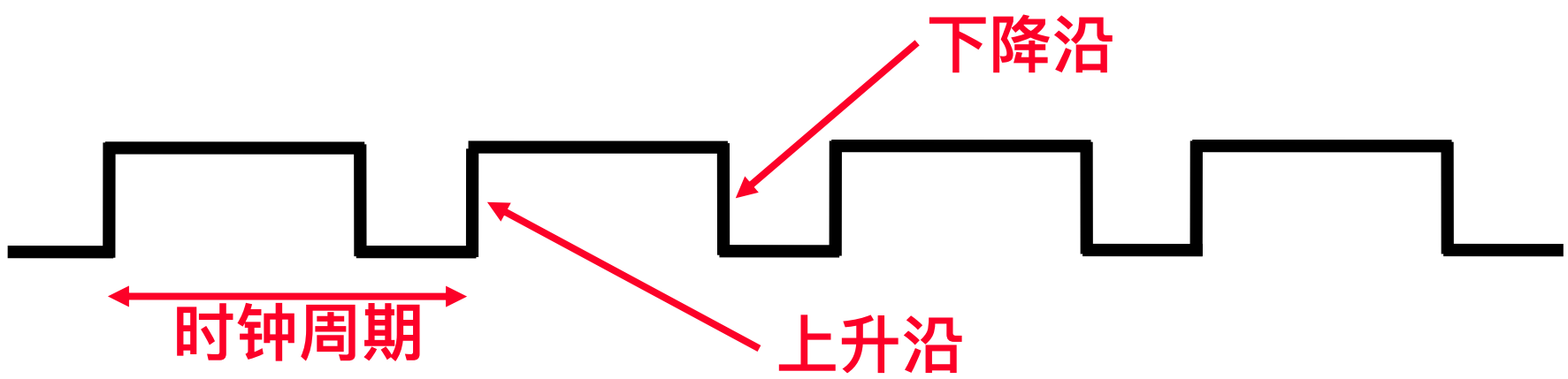
组合逻辑元件的特点：

其输出只取决于当前的输入。即：若输入一样，则其输出也一样

定时：所有输入到达后，经过一定的逻辑门延时，输出端改变，并保持到下次改变，不需要时钟信号来定时

状态元件：时序逻辑电路

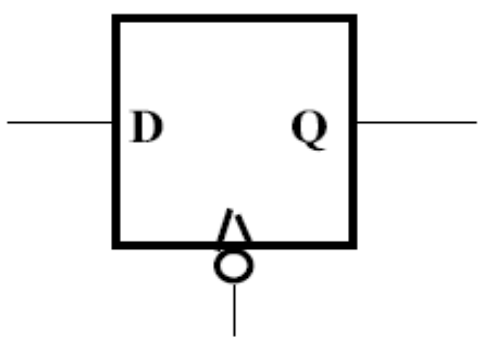
- 状态（存储）元件的特点：
  - 具有存储功能，在时钟控制下输入被写到电路中，直到下个时钟到达
  - 输入端状态由时钟决定何时被写入，输出端状态随时可以读出
- 定时方式：规定信号何时写入状态元件或何时从状态元件读出
  - 边沿触发（edge-triggered）方式：
    - 状态单元中的值只在时钟边沿改变。每个时钟周期改变一次。
      - 上升沿（rising edge） 触发：在时钟正跳变时进行读/写。
      - 下降沿（falling edge）触发：在时钟负跳变时进行读/写。

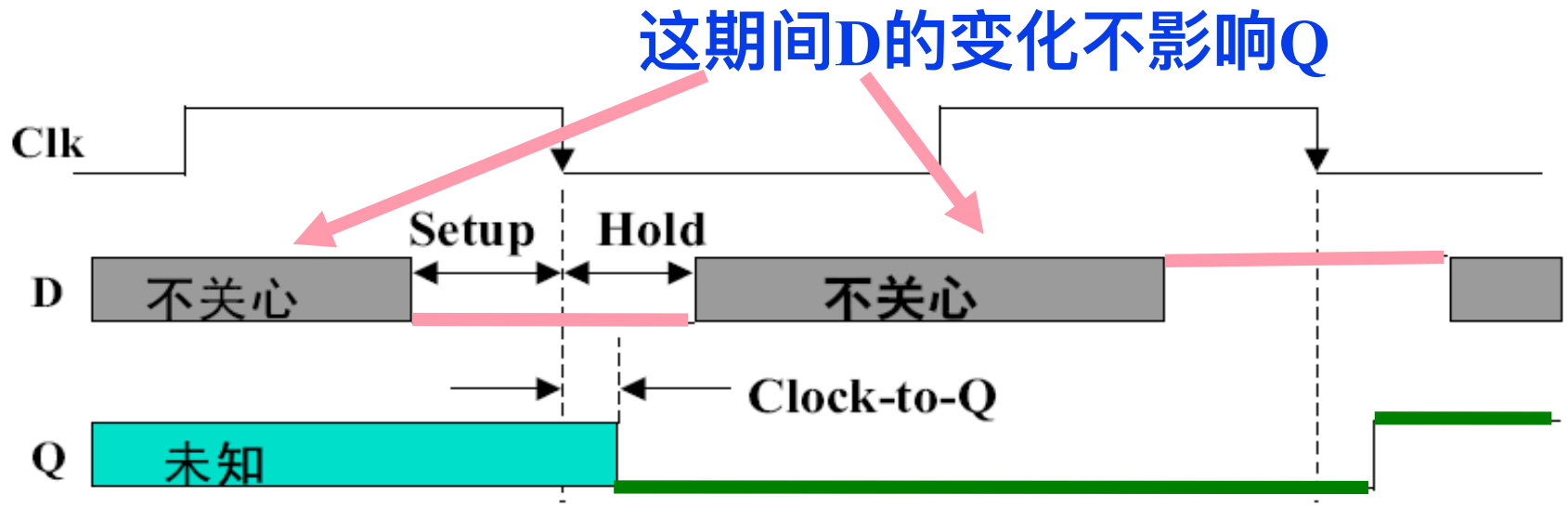


- 最简单的状态单元（回顾：数字逻辑电路课程内容）：
  - D触发器：一个时钟输入、一个状态输入、一个状态输出

存储元件中何时状态被改变?

双击此处添加文本





Q总是在clock-to-Q后跟着D变化

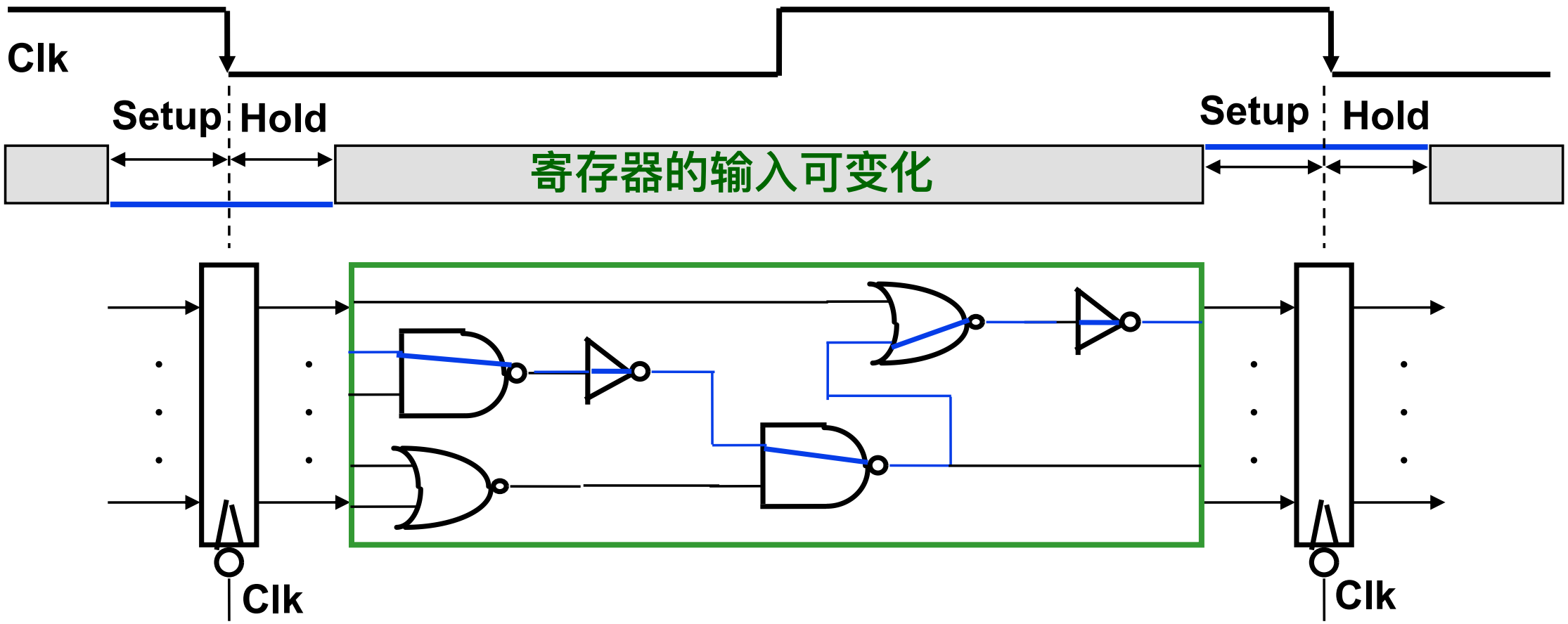
- 建立时间（**Setup Time**）：在触发时钟边沿 之前 输入必须稳定
- 保持时间（**Hold Time**）：在触发时钟边沿 之后 输入必须保持
- **Clock-to-Q time:（Latch Prop - 锁存延迟）**
  - 在触发时钟边沿，输出并不能立即变化

切记：状态单元的输入信息总是在一个时钟边沿到达后的“Clk-to-Q”时间才被写入，此时的输出才反映新的状态值

数据通路中的状态元件有两种：寄存器(组) + 存储器

数据通路与时序控制

现代计算机的时钟周期



数据通路由“ … + 状态元件 + 操作元件(组合电路) + 状态元件 + …” 组成

只有状态元件能存储信息，操作元件须从状态元件接收输入，并将输出写入状态元件。其输入为前一时钟生成的数据，输出为当前时钟所用的数据

Cycle Time = Clk-to-Q时间+Longest Delay+建立时间+时钟偏移

早期累加器型指令系统数据通路

AC：累加器
MQ：乘商寄存器
PC、IR、ALU、
IBR、MBR：？

- 最简单的数据通路结构
- 取指令数据路径为：

  PC→MAR,

  Read M,

  M→MBR→IBR→IR
- 取操作数、运算、送结果的数据路径为：

  操作数地址→MAR,

  Read M,

  M→MBR→ALU输入端,

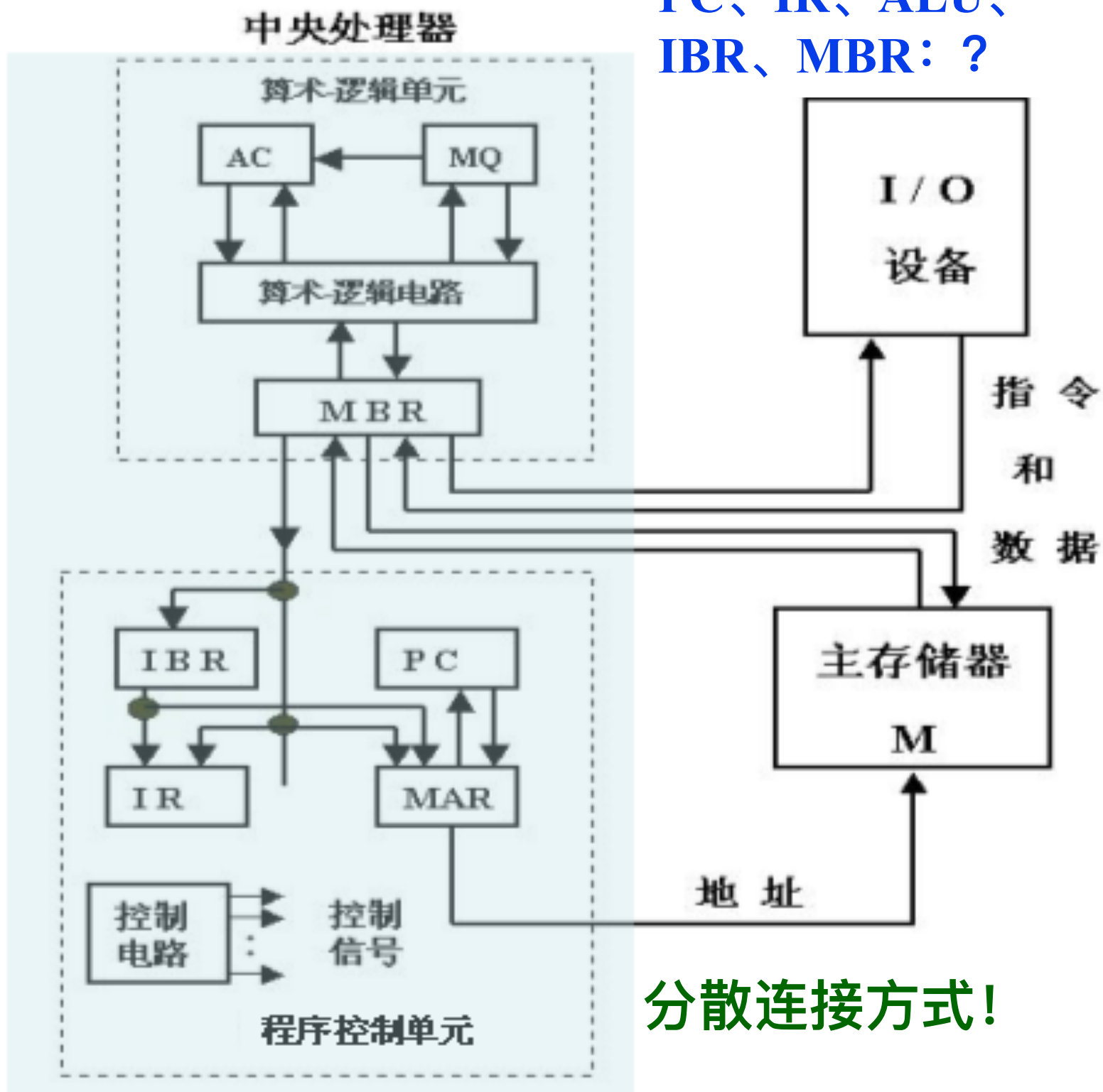
  AC→ALU输入端,

  ALU操作,

  ALU结果→AC,

  AC → MBR,

  Write M



分散连接方式！

IAS计算机（冯.诺依曼等设计）是现代计算机的原型

单总线数据通路

总线连接方式！

四种基本操作的时序控制信号

在寄存器之间传送数据　1Cycle

R0out，Yin

完成算术、逻辑运算　3Cycles

R1out，Yin

R2out，Add，Zin

Zout，R3in

从主存取字　R[R2]←M[R[R1]]　>3Cycles

R1out，MARin

Read, WMFC (等待MFC)

MDRout，R2in

写字到主存 M[R[R1]] ← R[R2]　>3Cycles

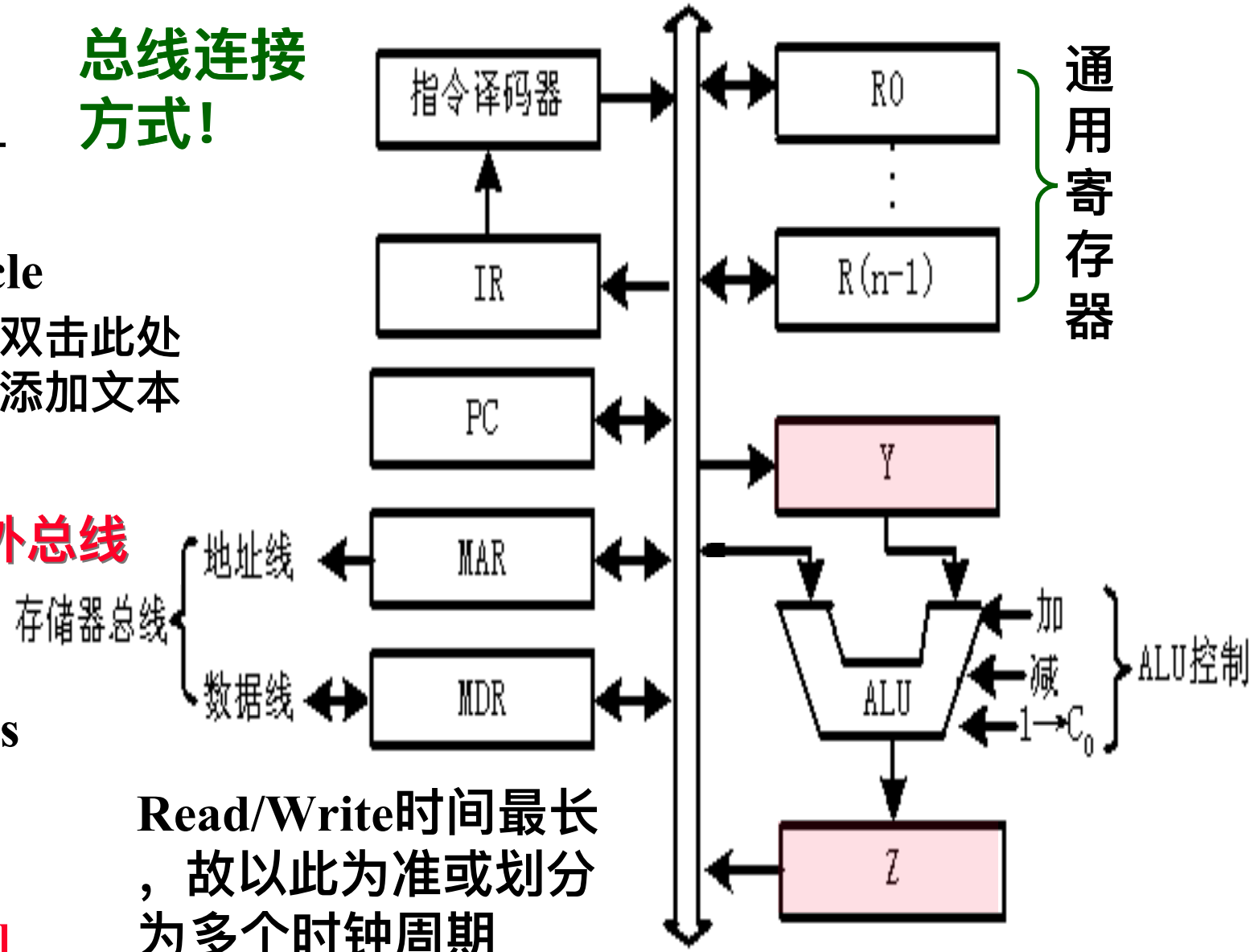
R1out，MARin

R2out，MDRin

Write, WMFC

CPU访存有两种通信方式

早期：直接访问MM，“异步”方式，用MFC应答信号；现在：先Cache后MM，“同步”方式，无需应答信号。



Read/Write时间最长，故以此为准或划分为多个时钟周期

双击此处添加文本

问题：时钟周期的宽度如何确定？

以上四种操作各需要几个时钟周期？

取指阶段的操作与时序控制信号？

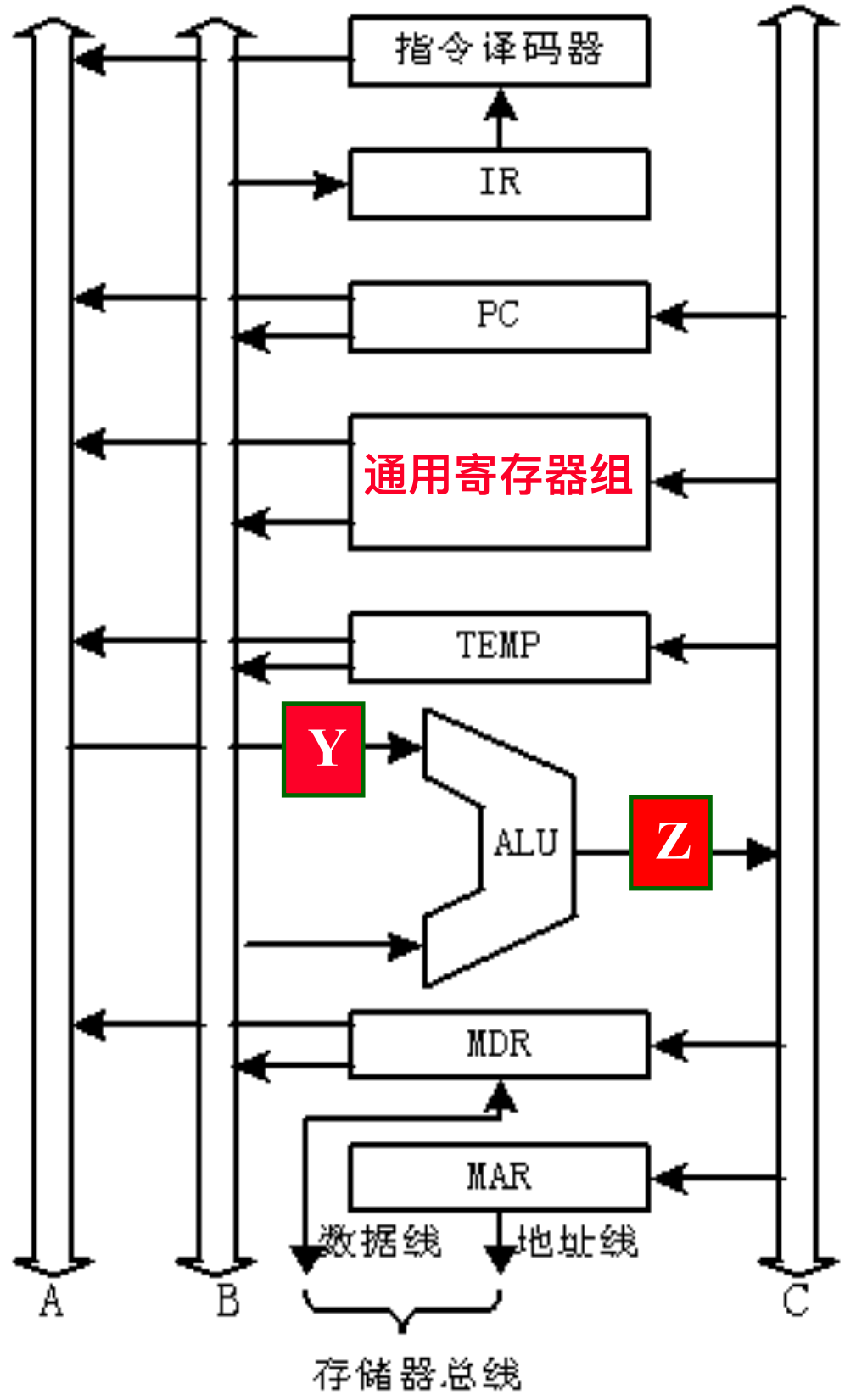
IR ←M[PC]，PC ←PC+“1”

三总线数据通路

- 单总线中一个时钟内只允许传一个数据，因而指令执行效率很低
- 可采用多总线方式，同时在多个总线上传送不同数据，提高效率
- 例如：三总线数据通路
  - 总线A、B分别传送两个源操作数，总线C传送结果
  - 单总线中的暂存器Y和Z在此可取消，Why?　　三个总线各自传不同数据，不会发生冲突，故无需Y和Z
  - 采用双口通用寄存器组
  - 如何实现 R[R3] ←R[R1] op R[R2]

    R1outA，R2outB，op，R3inC

    只要一个时钟周期（节拍）即可！



目前大都采用流水线方式执行指令，单总线或三总线的总线式数据通路很难实现指令流水执行。

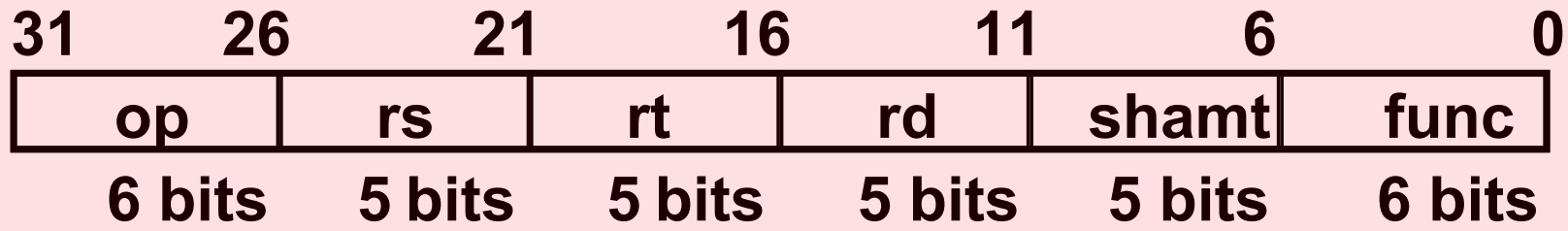
以下以MIPS指令系统为例简介CPU的工作原理。

MIPS的三种指令类型

MIPS有三种指令格式：R-型、I-型、J-型

本节内容无需掌握，仅为理解指令的执行过程而补充

- ADD and SUBSTRACT
  - add rd, rs, rt
  - sub rd, rs, rt



R-型指令格式

- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16

I-型指令格式

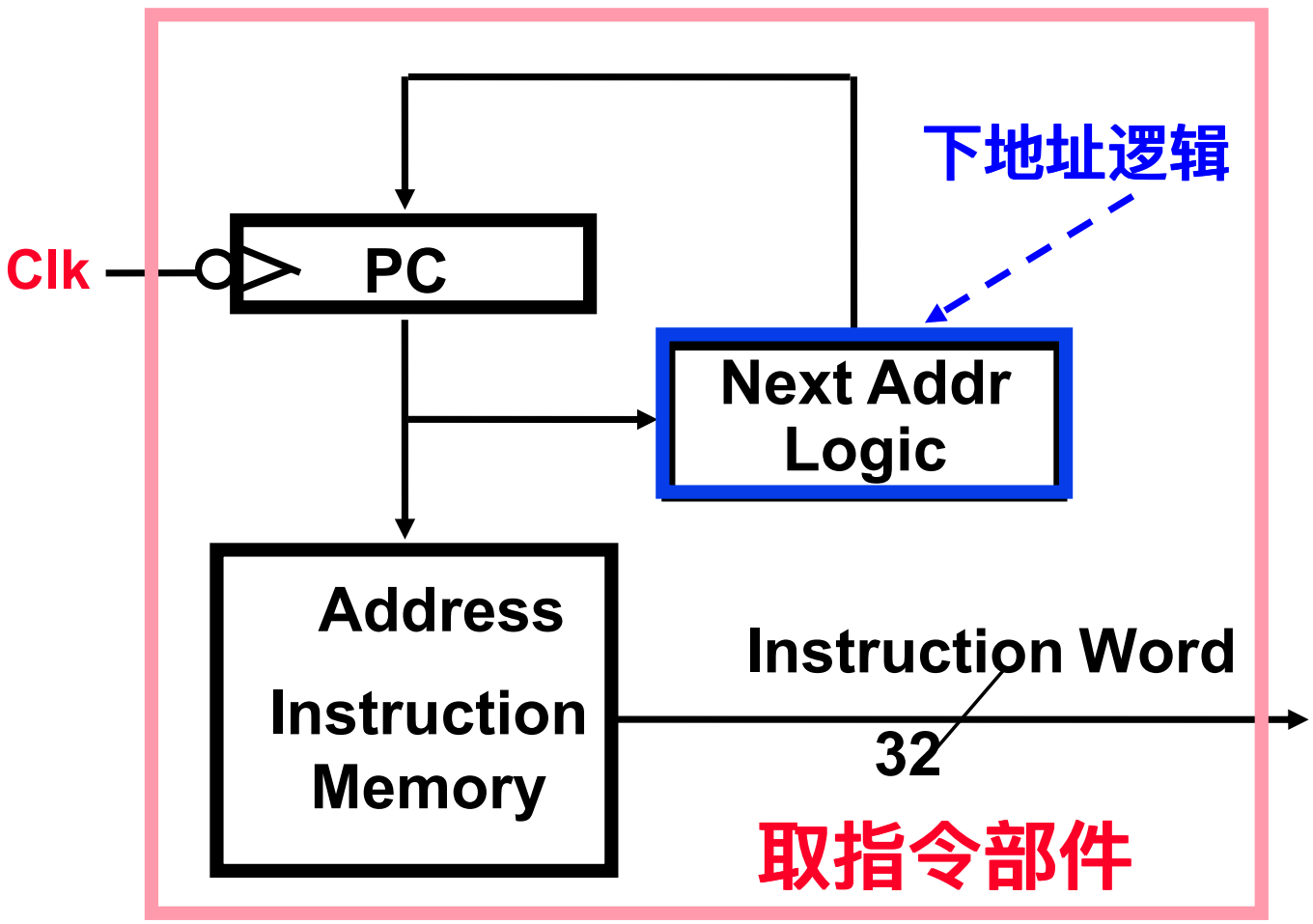
- JUMP:
  - j target

J-型指令格式

取指令部件(Instruction Fetch Unit)

- 每条指令都有的公共操作：
  - 取指令： M[PC]
  - 更新PC：PC ← PC + 4

转移（Branch and Jump）时，PC内容再次被更新为 “转移目标地址”

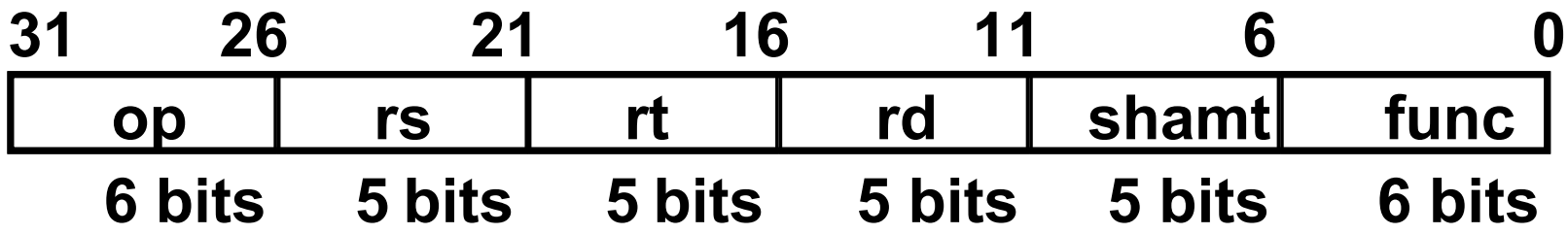


顺序：先取指令，再改PC的值
（具体实现时，可以并行）

绝不能先改PC的值，再取指令

取指后，各指令功能不同，数据通路中信息流动过程也不同

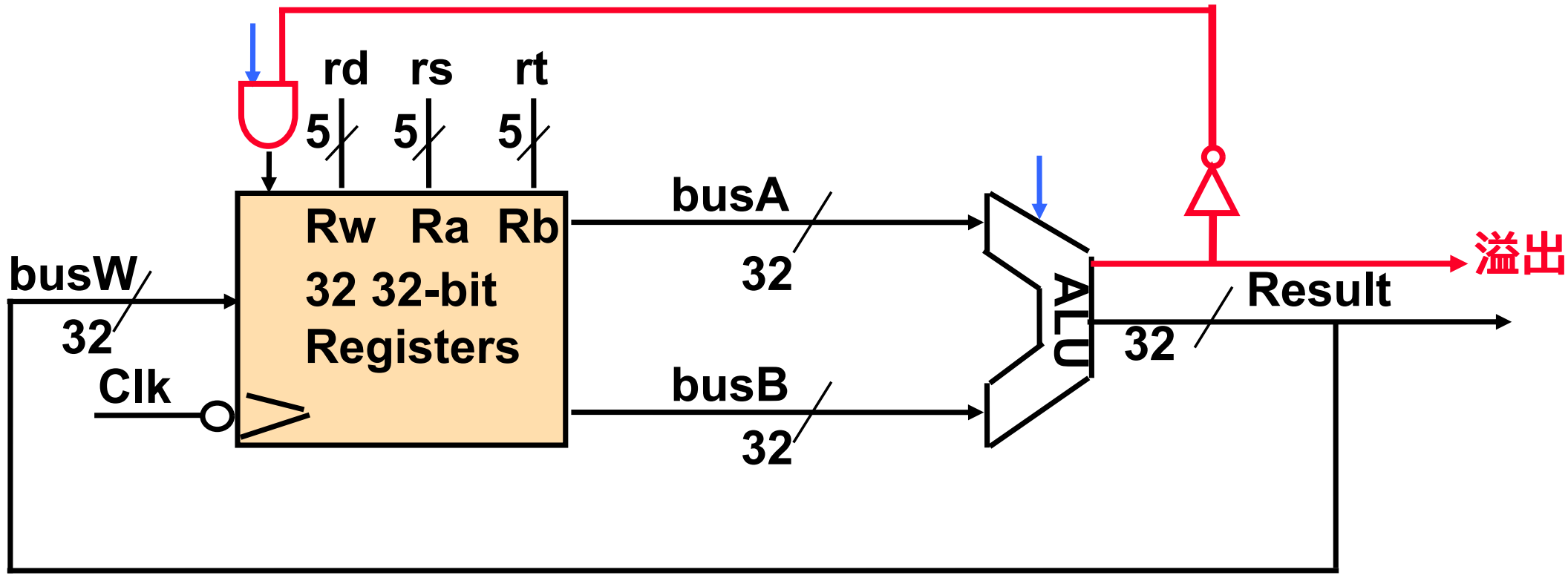
RR（R-type）型指令的数据通路



功能：R[rd] ← R[rs] op R[rt]，如：add　rd, rs, rt

溢出时，不写结果
并转异常处理程序

不考虑公共操作，仅R-Type指令执行阶段的数据通路如下：



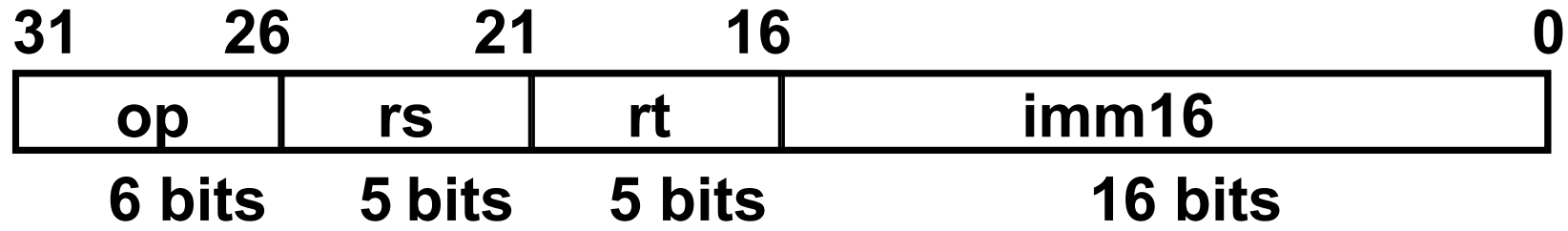
Ra, Rb, Rw 分别对应指令的rs, rt, rd

典型的三总线结构！

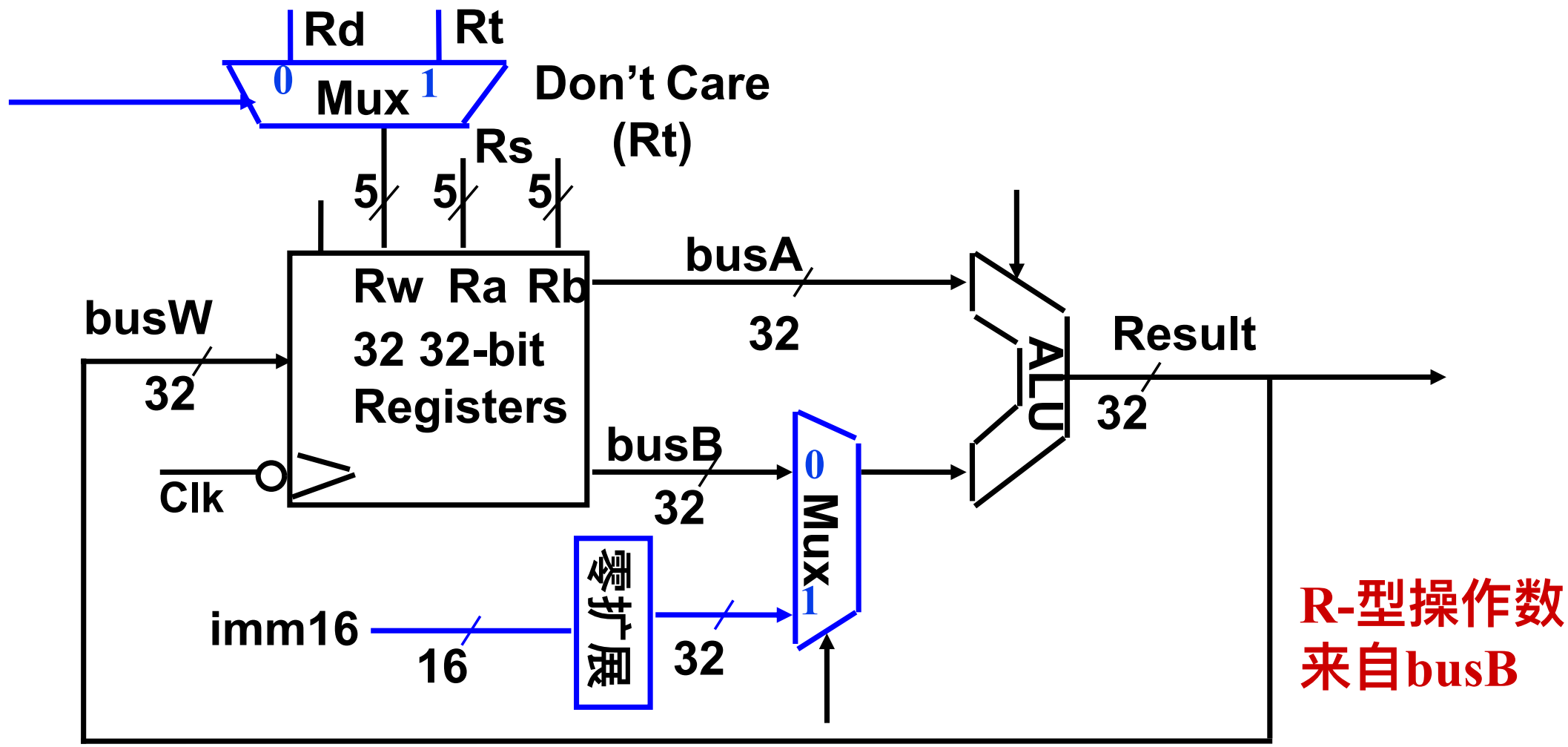
busA、busB、busW

带立即数的逻辑指令的数据通路

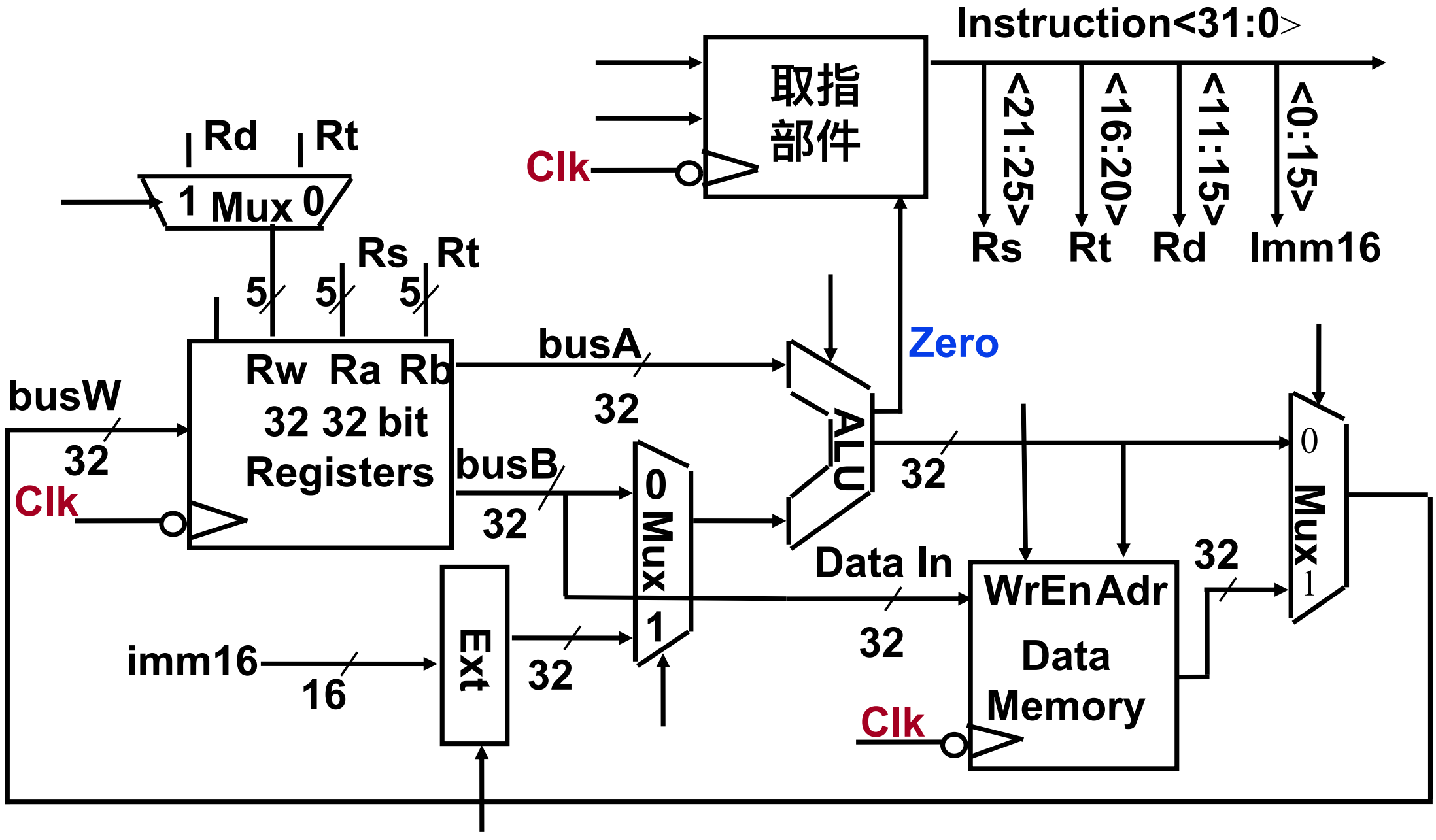
R[rt] ← R[rs] op ZeroExt[imm16]]　　Ex: ori　rt, rs, imm16



R-型结果写入Rd



单周期数据通路的基本结构

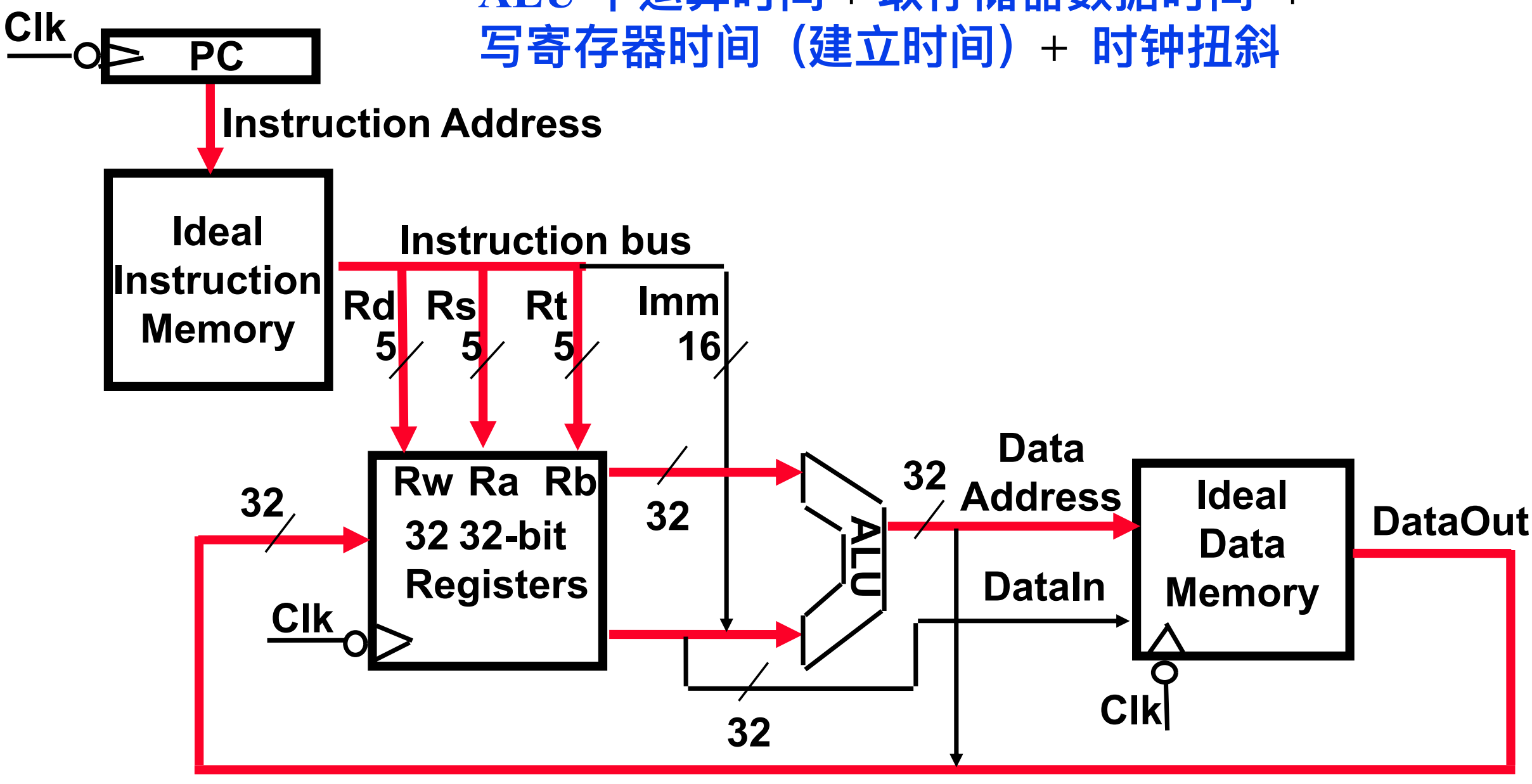


指令执行结果总是在下个时钟到来时开始保存在 寄存器 或 存储器 或 PC 中!

单周期数据通路中的关键路径（Load操作）

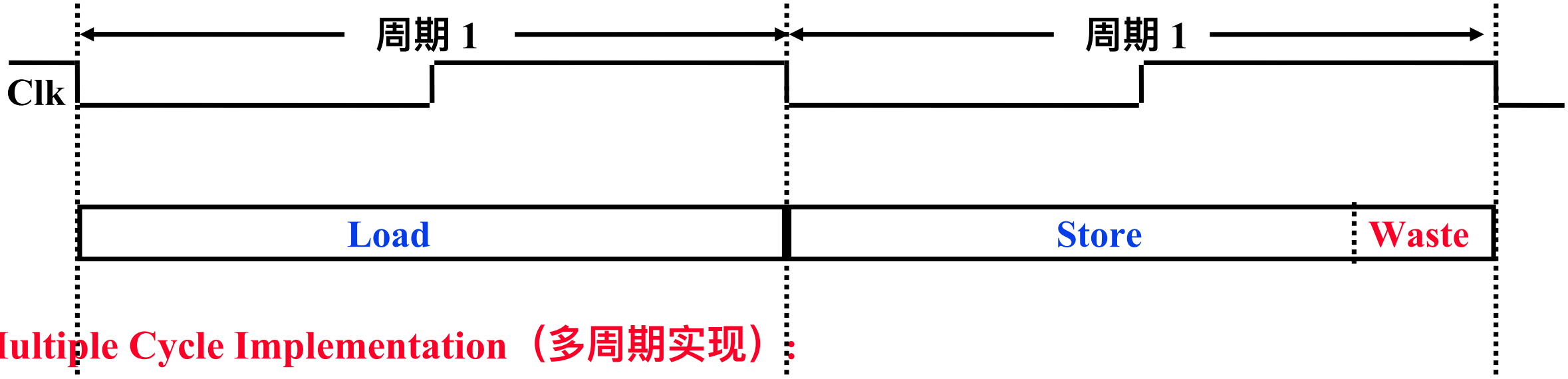
Load操作：R[Rt] ← M[R[Rs]+Imm16]

关键路径 (Load) =
Clk-to-Q + 取指令时间 + 取寄存器数据时间 +
ALU 中运算时间 + 取存储器数据时间 +
写寄存器时间（建立时间）+ 时钟扭斜

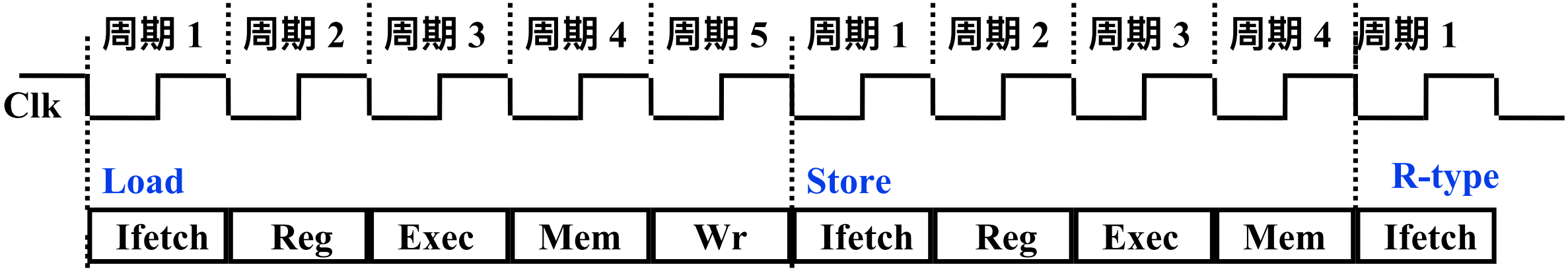


单周期, 多周期 和 流水线比较

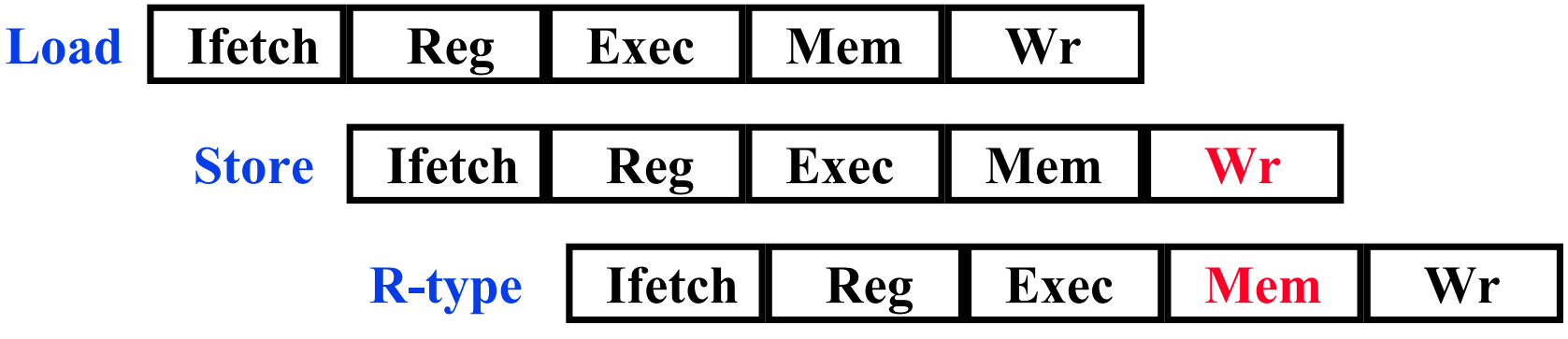
Single Cycle Implementation（单周期实现）：



Multiple Cycle Implementation（多周期实现）：



Pipeline Implementation（流水线实现）：



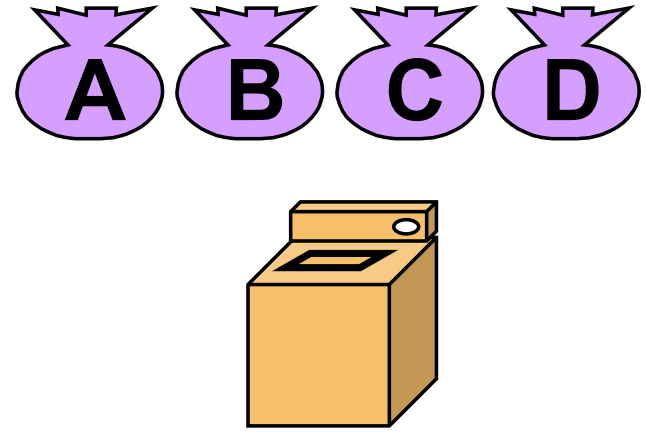
前述的单总线结构CPU一定是多周期实现的CPU!

程序的执行机制

- 分以下三个部分介绍
  - 第一讲：程序执行概述
    - 程序及指令的执行过程
    - CPU的基本功能和基本组成
  - 第二讲：数据通路基本结构和工作原理
    - 数据通路基本结构
    - 数据通路的时序控制
    - 数据通路基本工作原理
  - 第三讲：流水线方式下指令的执行
    - 指令流水线的基本原理
    - 适合流水线的指令集特征
    - CISC和RISC风格指令集
    - 指令流水线的实现
    - 高级流水线实现技术

一个日常生活中的例子—洗衣服

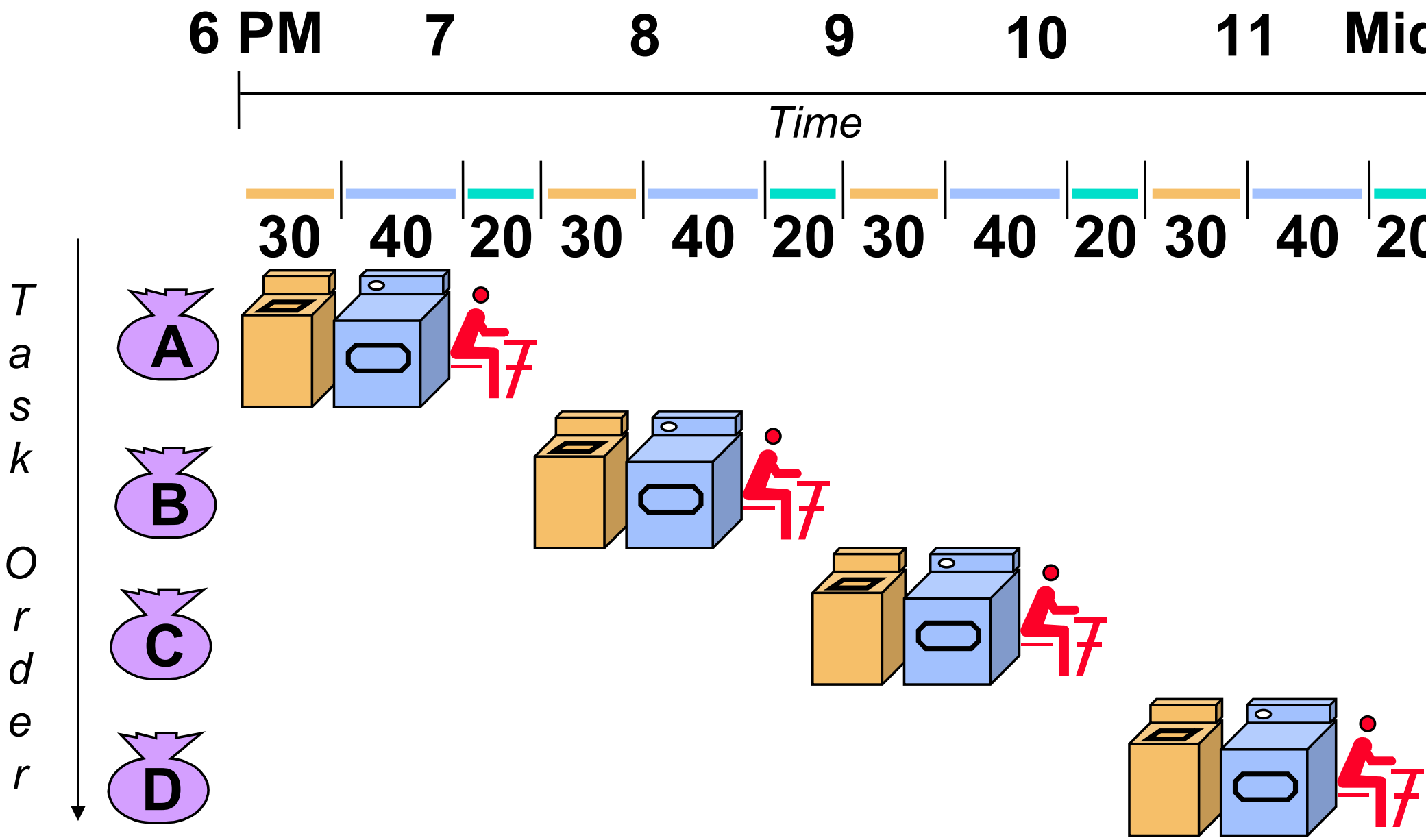
- **Laundry Example**
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes



如果让你来管理洗衣店，你会如何安排?

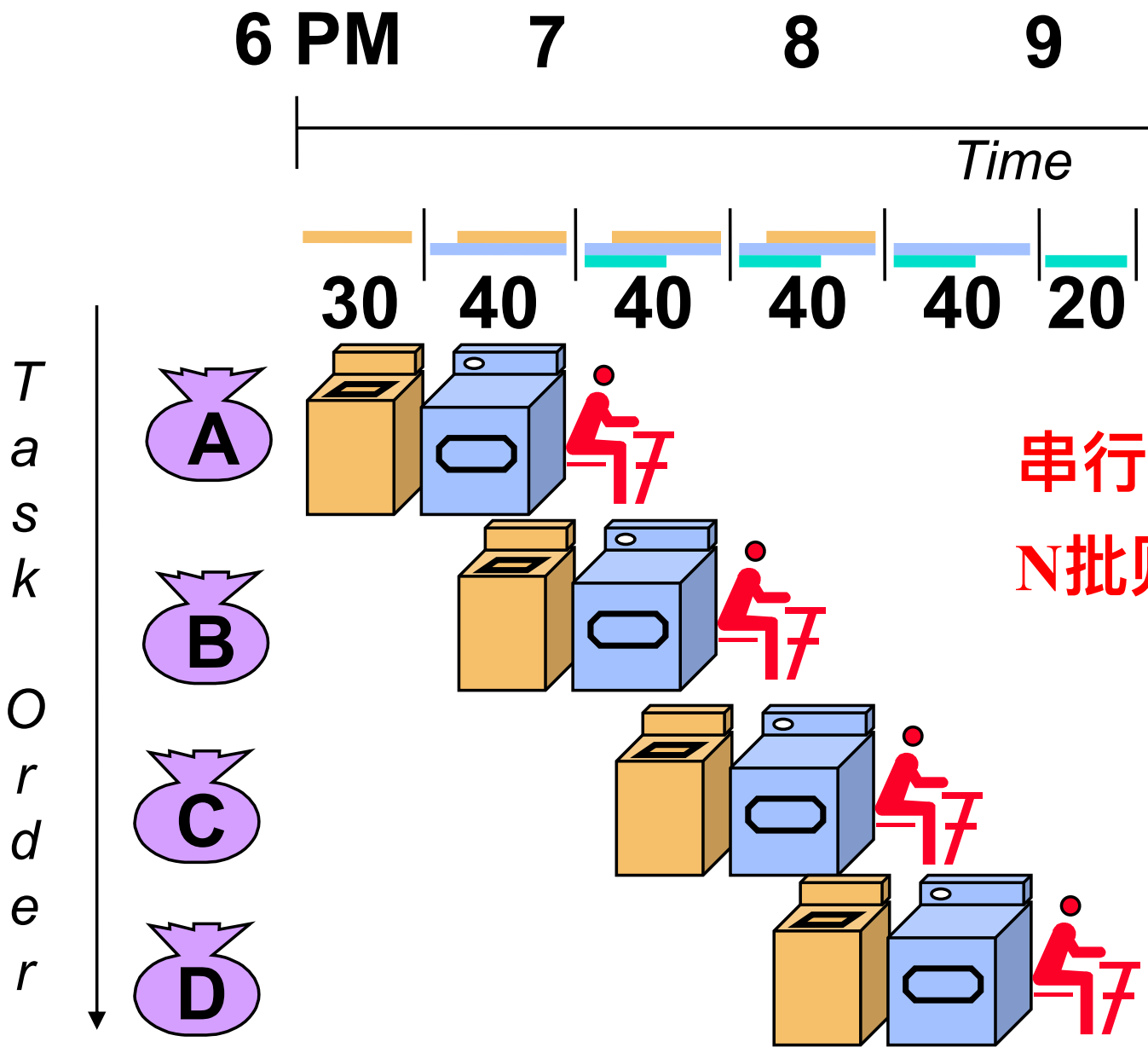
Pipelining: It’s Natural !

Sequential Laundry（串行方式）



- 串行方式下 4 批衣服需花费 6 小时（4x(30+40+20)=360分钟）
- N批衣服，需花费的时间为Nx(30+40+20) = 90N
- 如果用流水线方式洗衣服，则花多少时间呢?

Pipelined Laundry: (Start work ASAP)



串行为90分钟x4=6小时

N批则为90xN分钟

流水线方式下，需
30+4x40+20
=210分 (3.5小时)

如果有N批衣服呢?

30+Nx40+20分钟

假定每一步时间均衡，则比串行方式提高约3倍!

流水方式下，所用时间主要
与最长阶段的时间有关!

指令流水线的基本概念

- 五段流水线

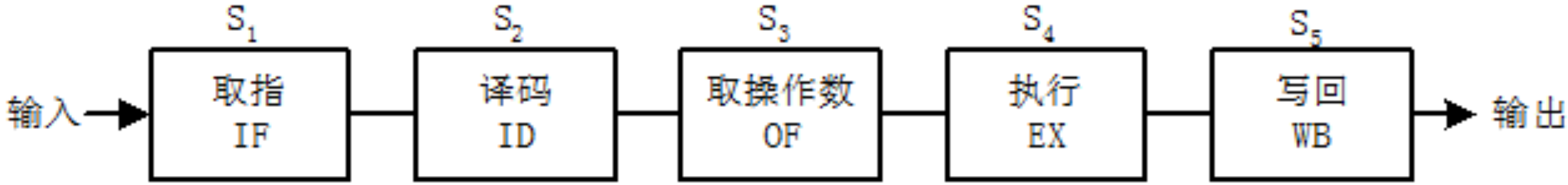
取指令(IF)：根据PC的值从存储器取出指令。

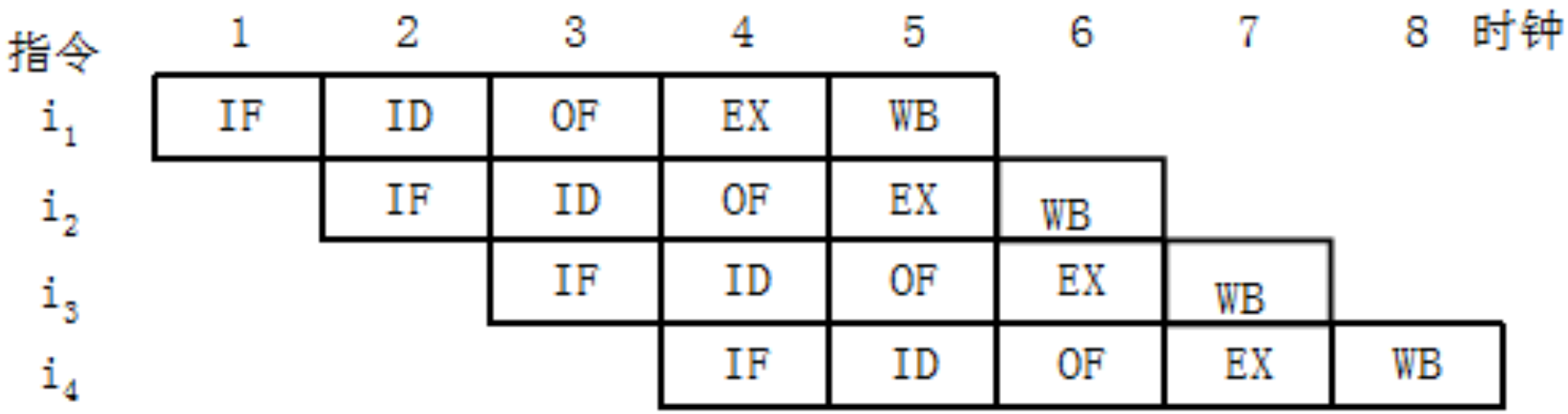
指令译码(ID)：产生指令执行所需的控制信号。

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。



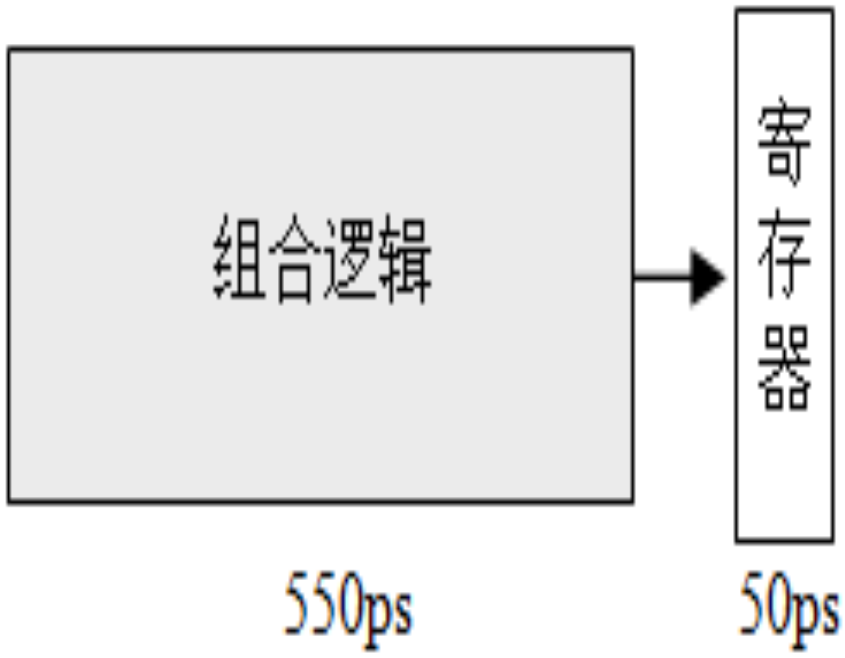


单周期数据通路中指令的执行

假定：最复杂指令执行过程 ① 取指：200ps；②译码和读操作数：50ps；
③ALU操作：100ps；④读存储器：200ps；⑤结果写寄存器：50ps。

200+50+100+200=550

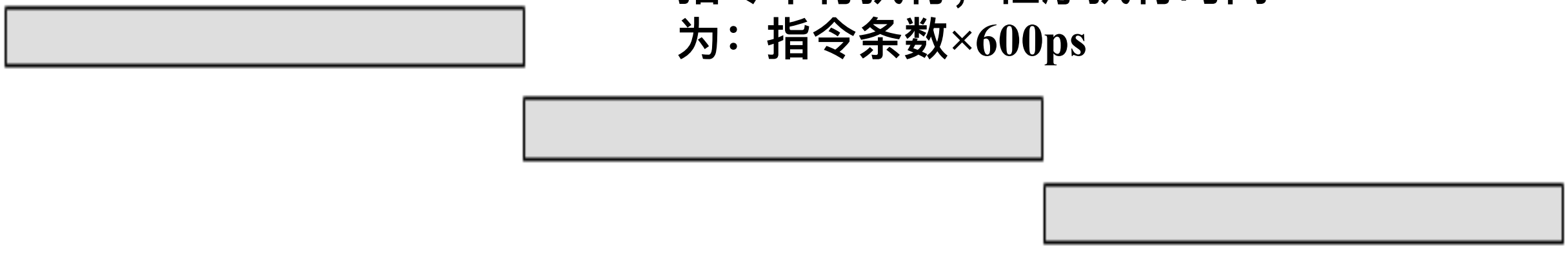
单周期：每条指令在单个时钟周期内完成，
故CPI=1，时钟周期=600ps



CPI=1，指令延时为600ps
指令吞吐率为1.67GIPS

每秒执行指令条数：
$1/600ps=1/(600\times10^{-15})=1.67\times10^{12}$

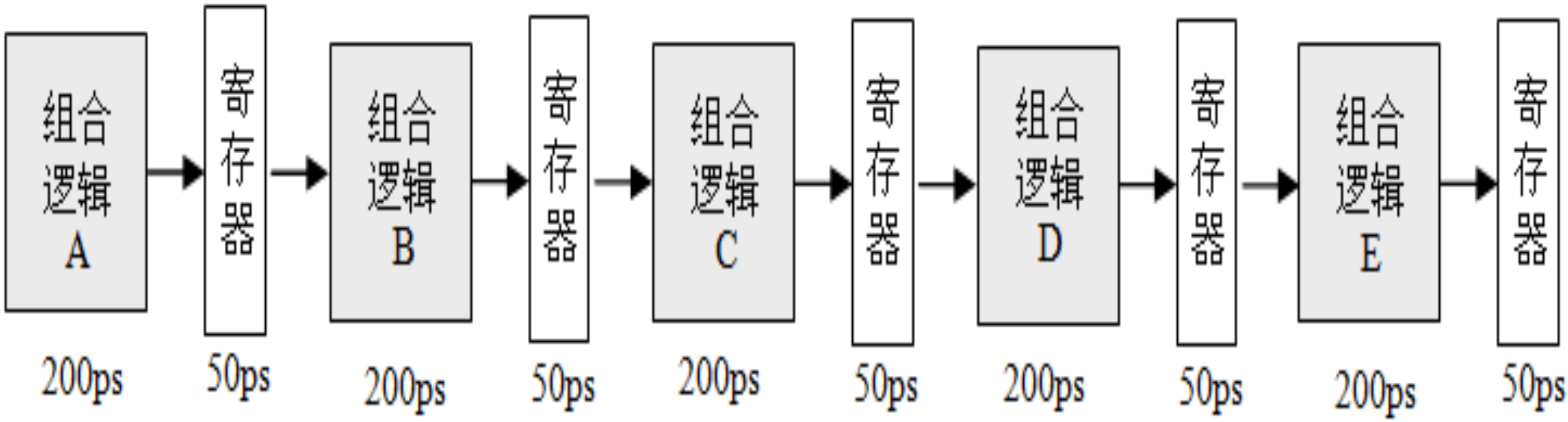
指令串行执行，程序执行时间为：指令条数×600ps

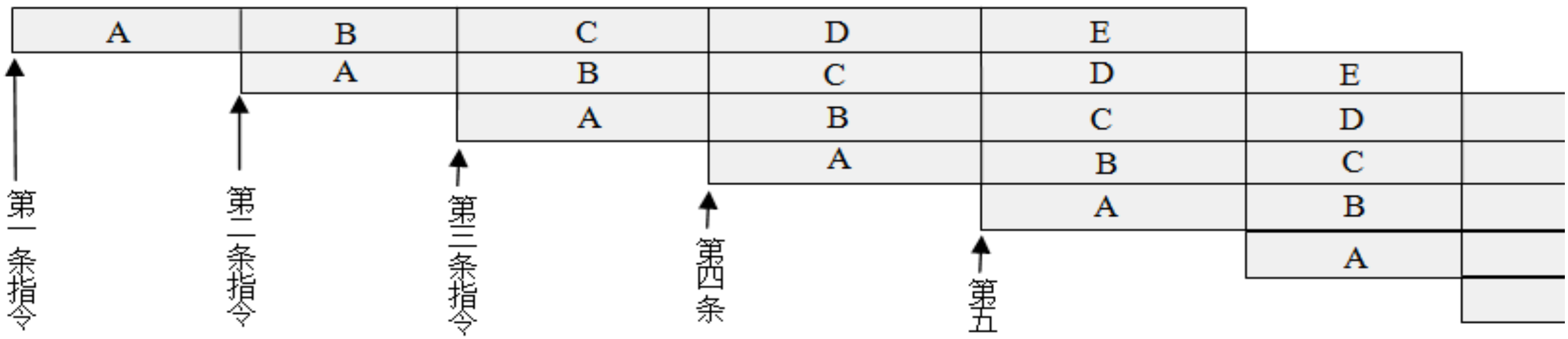


流水线数据通路中指令的执行

假定：最复杂指令执行过程 ① 取指：200ps；②译码和读操作数：50ps；
③ALU操作：100ps；④读存储器：200ps；⑤结果写寄存器：50ps。

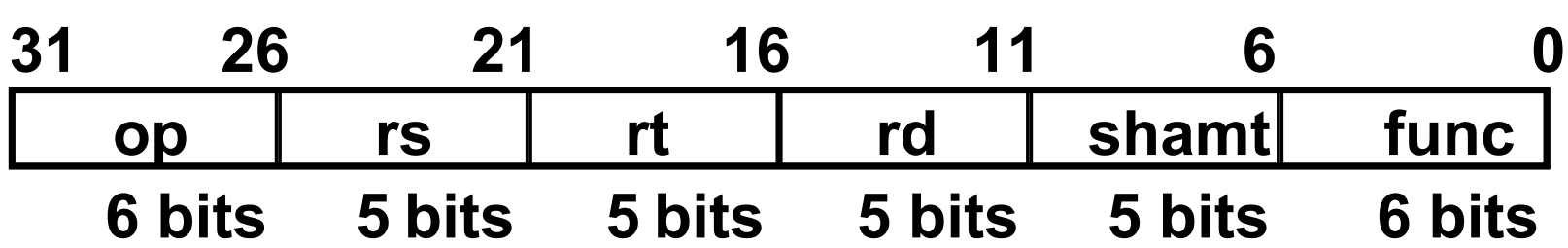
最长段为200ps　　**指令延时为：250ps×5=1.25ns**　　**指令吞吐率为4GIPS**





流水线指令集的设计

- 具有什么特征的指令集有利于流水线执行呢?
  - 长度尽量一致，有利于简化取指令和指令译码操作
    - MIPS指令32位，下址计算方便: PC+4
    - X86指令从1字节到17字节不等，使取指部件极其复杂
  - 格式少，且源寄存器位置相同，有利于在指令未知时就可取操作数
    - MIPS指令的rs和rt位置一定，在指令译码时就可读rs和rt的值



若位置随指令不同而不同，则需先确定指令类型才能取寄存器编号

- load / Store指令才能访问存储器，有利于减少操作步骤，规整流水线
  - lw/sw指令的地址计算和运算指令的执行步骤规整在同一个周期
  - X86运算类指令操作数可为内存数据，需计算地址、访存、执行
- 内存中”对齐”存放，有利于减少访存次数和流水线的规整

总之，规整、简单和一致等特性有利于指令的流水线执行

按指令格式的复杂度来分

按指令格式的复杂度来分，有两种类型计算机：

复杂指令集计算机CISC (Complex Instruction Set Computer)

精简指令集计算机RISC (Reduce Instruction Set Computer)

早期CISC设计风格的主要特点

(1) 指令系统复杂

变长操作码 / 变长指令字 / 指令多 / 寻址方式多 / 指令格式多

(2) 指令周期长

绝大多数指令需要多个时钟周期才能完成

(3) 各种指令都能访问存储器

除了专门的存储器读写指令外，运算指令也能访问存储器

(4) 采用微程序控制

(5) 有专用寄存器

(6) 难以进行编译优化来生成高效目标代码

例如，VAX-11/780小型机

16种寻址方式；9种数据格式；303条指令；一条指令包括1～2个字节的操作码和下续N个操作数说明符。一个说明符的长度达1 ～10个字节。

复杂指令集计算机CISC

- CISC的缺陷
  - 日趋庞大的指令系统不但使计算机的研制周期变长，而且难以保证设计的正确性，难以调试和维护，并且因指令操作复杂而增加机器周期，从而降低了系统性能。
- 1975年IBM公司开始研究指令系统的合理性问题，John Cocks提出精简指令系统计算机 RISC (Reduce Instruction Set Computer)。
  - 对CISC进行测试，发现一个事实：
    - 在程序中各种指令出现的频率悬殊很大，最常使用的是一些简单指令，这些指令占程序的80%，但只占指令系统的20%。而且在微程序控制的计算机中，占指令总数20%的复杂指令占用了控制存储器容量的80%。
  - 1982年美国加州伯克利大学的RISCI ，斯坦福大学的MIPS，IBM公司的IBM801相继宣告完成，这些机器被称为第一代RISC机。

SKIP

Top 10 80x86 Instructions

° Rank	instruction	Integer Average	Percent total executed
1	load	MOV M to R	22%
2	conditional branch	Jcc	20%
3	compare	CMP	16%
4	store	MOV R to M	12%
5	add		8%
6	and		6%
7	sub		5%
8	move register-register		4%
9	call		1%
10	return		1%
	Total		96%

° Simple instructions dominate instruction frequency

（简单指令占主要部分，使用频率高！）

BACK

RISC设计风格的主要特点

(1) 简化的指令系统

指令少 / 寻址方式少 / 指令格式少 / 指令长度一致

(2) 以RR方式工作

除Load/Store指令可访存外，其余指令都只访问寄存器

(3) 指令周期短

以流水线方式工作， 因而除Load/Store指令外，其他简单指令都只需一个或一个不到的时钟周期就可完成

(4) 采用大量通用寄存器，以减少访存次数

(5) 采用硬连线路控制器，不用或少用微程序控制

(6) 采用优化的编译系统，力求有效地支持高级语言程序

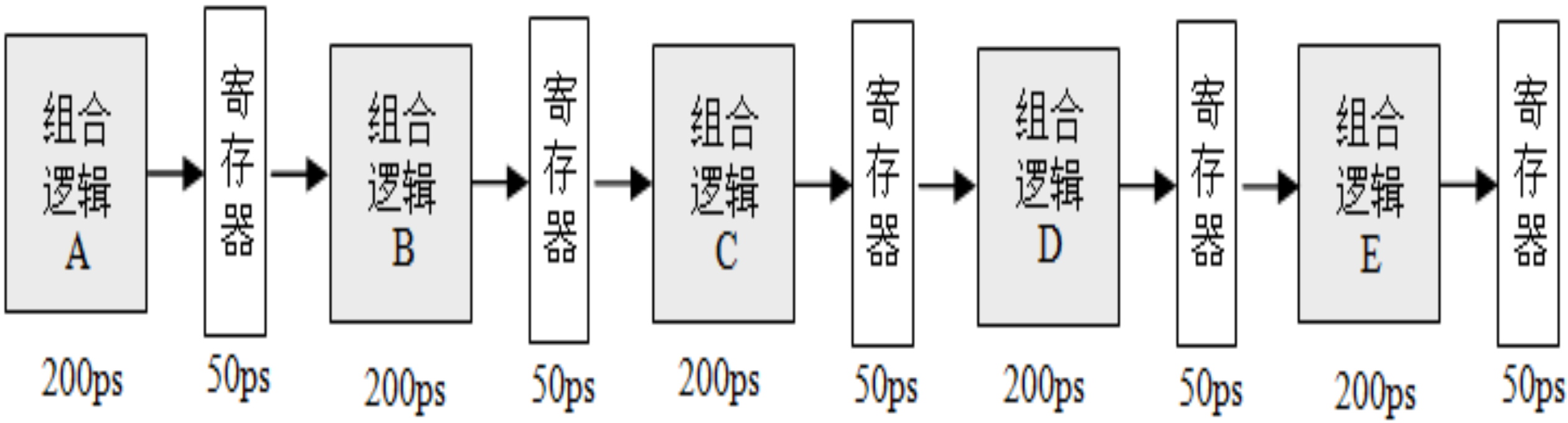
MIPS是典型的RISC处理器，82年以来新的指令集大多采用RISC体系结构

x86因为“兼容”的需要，保留了CISC的风格，同时也借鉴了RISC思想

指令流水线的实现

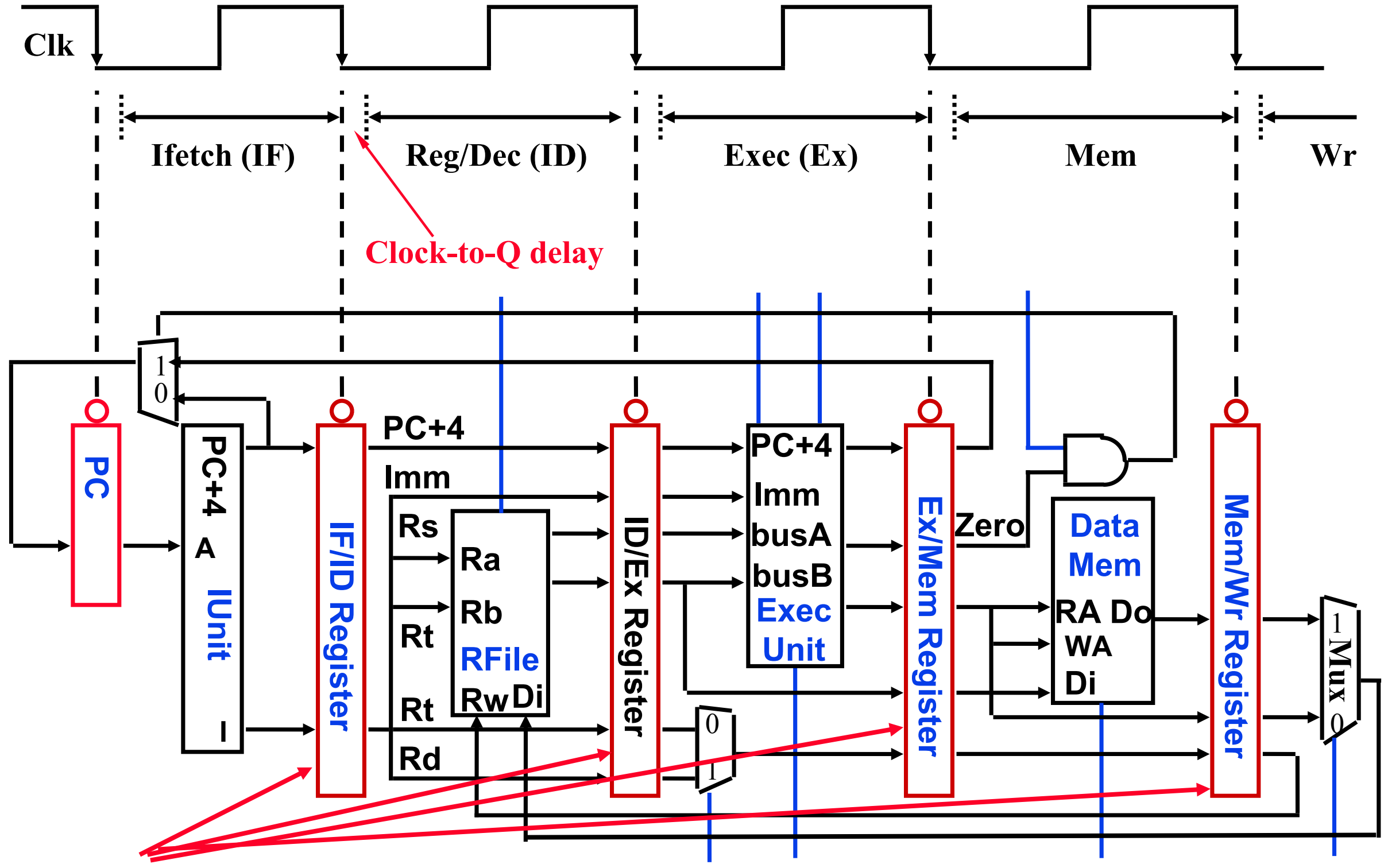
假定：最复杂指令执行过程 ① 取指：200ps；②译码和读操作数：50ps；
③ALU操作：100ps；④读存储器：200ps；⑤结果写寄存器：50ps。

可以分5个流水段，最长阶段为200ps



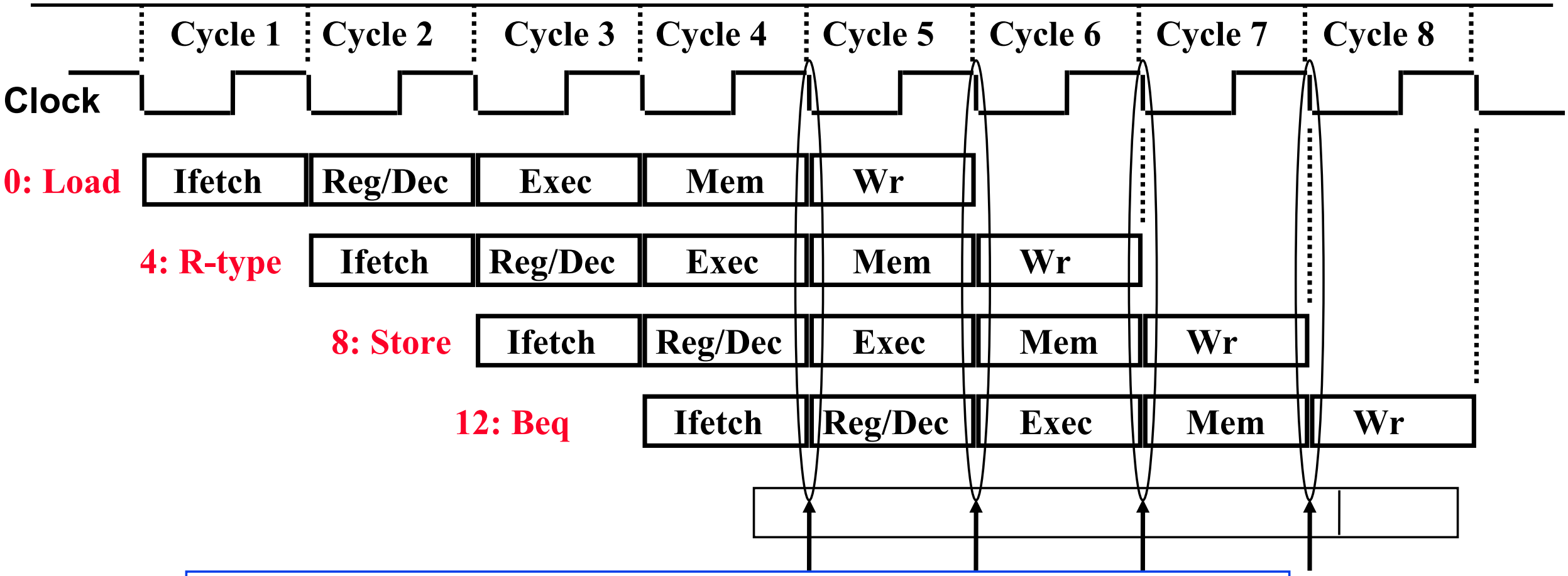
取指令 IFetch　　读数/译码 Reg/Dec　　ALU运算 Exec　　读/写存储器 Mem　　写结果 Write

五段流水线数据通路



流水段寄存器：保存每个时钟周期执行的结果!

指令流水线的执行举例



考察以下几个点的情况：

End of Cycle 4　End of Cycle 5　End of Cycle 6　End of Cycle 7

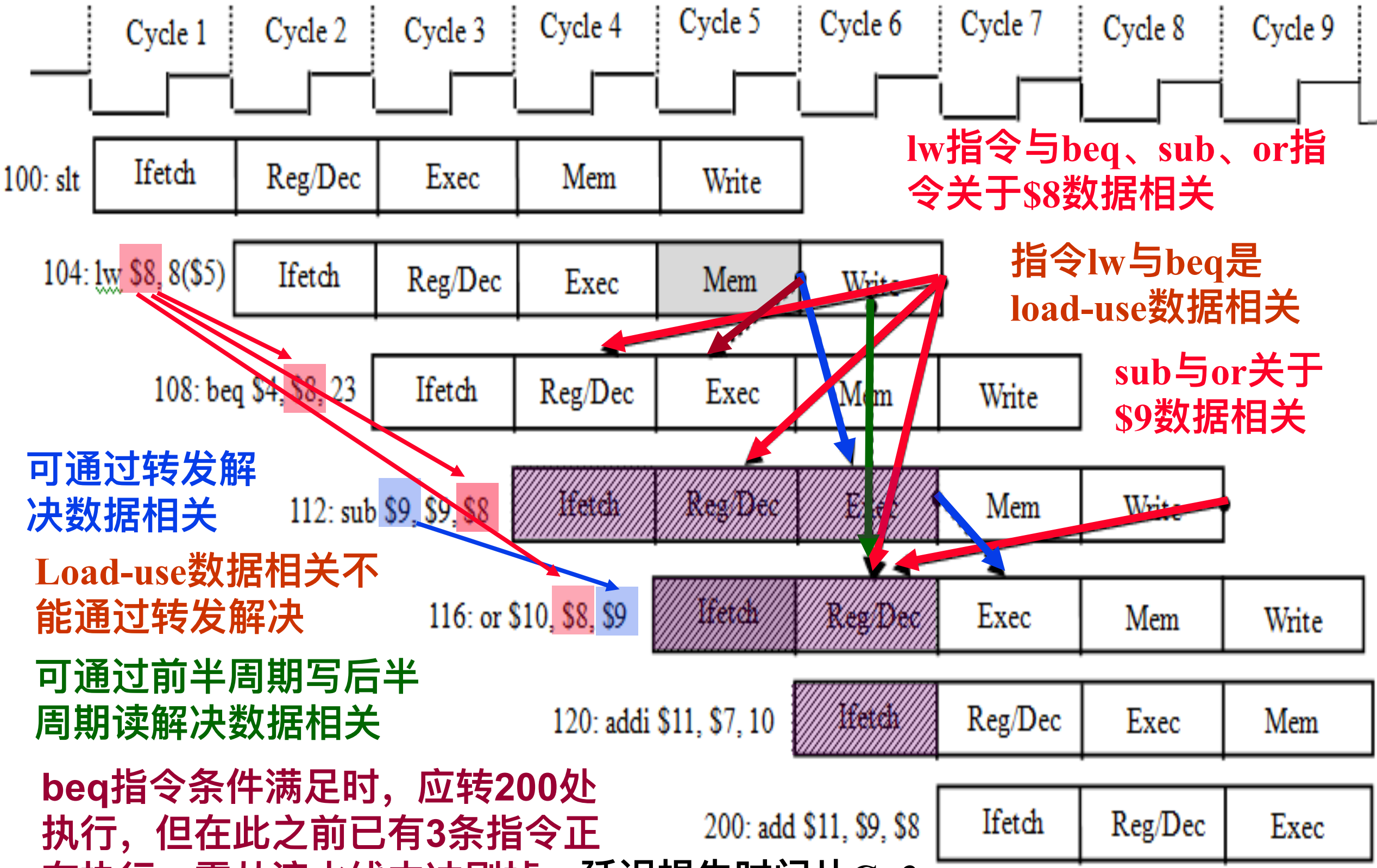
周期4结束: Load 的 Mem, R-type 的 Exec,　Store 的 Reg,　Beq 的 Ifetch

周期5结束: Load 的 Wr,　R-type 的 Mem,　Store 的 Exec,　Beq 的 Reg

周期6结束:　R-type 的 Wr,　Store 的 Mem, Beq 的 Exec

周期7结束:　Store 的 Wr,　Beq 的 Mem

指令流水线的执行举例



lw指令与beq、sub、or指令关于$8数据相关

指令lw与beq是load-use数据相关

sub与or关于$9数据相关

可通过转发解决数据相关

Load-use数据相关不能通过转发解决

可通过前半周期写后半周期读解决数据相关

beq指令条件满足时，应转200处执行，但在此之前已有3条指令正在执行，需从流水线中冲刷掉　延迟损失时间片C=3

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)

本PPT内容只需大概了解

编译器优化指令顺序解决数据冒险

以下源程序可生成两种不同的代码，优化的代码可避免Load阻塞

a = b + c;

d = e – f;

假定 a, b, c, d ,e, f 在内存

编译器的优化很重要!

Slow code:

```
lw      $2, b
lw      $3, c
add     $1, $2, $3
sw      $1, a
lw      $5, e
lw      $6, f
sub     $4, $5, $6
sw      $4, d
```

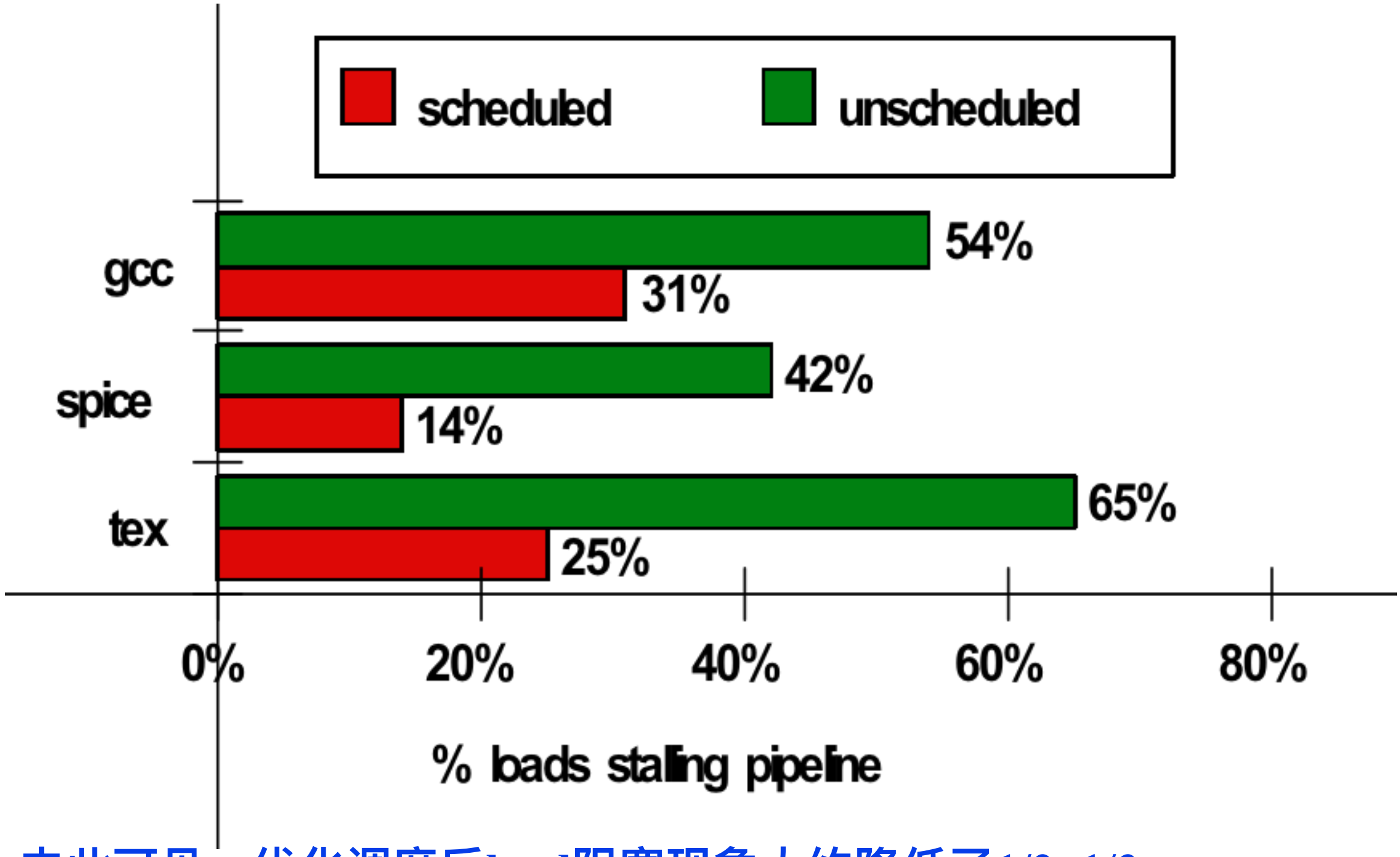
调整后

Fast code:

```
lw      $2, b
lw      $3, c
lw      $5, e
add     $1, $2, $3
lw      $6, f
sw      $1, a
sub     $4, $5, $6
sw      $4, d
```

如果硬件不支持阻塞处理的话，则编译器可以将顺序调整和插入NOP指令结合起来，在找不到可插入的指令时，插入NOP指令!

编译器优化以避免阻塞的情况调查



由此可见，优化调度后load阻塞现象大约降低了1/2~1/3

编译器优化指令顺序解决控制冒险

- 基本思想：把分支指令前面的与分支指令无关的指令调到分支指令后面执行，以填充延迟时间片（也称分支延迟槽Branch Delay slot），不够时用nop填充

如何对以下程序段进行分支延迟调度？（假定时间片为2）

```
lw $1, 0($2)
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 2
add $3, $3,$2
sw $1, 0($2)
……
```

调度后

```
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 2
lw $1, 0($2)
nop
add $3, $3,$2
sw $1, 0($2)
```

若分支延迟时间片减少为1

```
lw $3, 0($2)
add  $6, $4, $2
beq $3, $5, 2
lw $1, 0($2)
add $3, $3,$2
sw $1, 0($2)
……
```

调度后可能带来其他问题：
产生新的load-use数据冒险

调度后，无需在硬件线路中阻塞branch指令后面指令的执行

提高性能措施—实现指令级并行

- 实现指令流内部的并行流水线称为指令级并行（ILP）
- 有两种指令级并行策略
  - 超流水线（Super- pipelining）
    - 级数更多的流水线　　CPI =?　　CPI =1
    - 理想情况下，流水线的加速比与流水段的数目成正比

（即：理想情况下，流水段越多，时钟周期越短，指令吞吐率越高）

但是，它会增加开销，且是有极限的！可以怎样突破极限呢?

N段流水线说明一个时钟周期内最多有几条指令同时并行执行?

N条！故N越大并行度越高！

  - 多发射流水线（Multiple issue pipelining ）
    - 多条指令(如整数、浮点、装入/存储等) 同时启动并独立运行
    - 前提：有多个执行部件。如定点、浮点、乘/除、取数/存数部件等
    - 结果：能达到小于1的CPI，定义CPI的倒数为IPC

（例如：理想的四路多发射流水线的IPC为4）

    - 两种实现方法
      - 静态多发射：由编译器在编译时静态完成指令打包和冒险处理
      - 动态多发射：由硬件在执行时动态完成指令打包和冒险处理

静态多发射处理器

- 由编译器在编译时进行相关性分析和静态分支预测，以静态完成指令打包
  - 指令打包（将同时发射的多条指令合并到一个长指令中）
    - 将同一个时钟周期内发射的多个指令看成是一条多个操作的长指令，称为一个“发射包”
    - “静态多发射”也被称为“超长指令字”（VLIW-Very Long Instruction Word），采用这种技术的处理器被称为VLIW处理器
    - 在同一个周期内发射的指令类型是受限制的（举例:干洗/水洗）

      例如，只能是一条ALU指令/分支指令、一条Load/Store指令
    - IA-64采用这种方法，Intel称其为EPIC（Explicitly Parallel Instruction Computer—显式并行指令计算机）　　安腾、安腾2

静态打包时，一定要保证指令内部不会出现冒险！

动态多发射处理器

- 由硬件在执行时动态完成指令打包或冒险处理
- 通常被称为超标量处理器（Superscalar）
  - 同一个时钟动态发射多条指令，一个周期内可执行一条以上指令
- 与VLIW处理器的不同点：
  - VLIW处理器：与机器结构密切相关，在结构有差异机器上要重新编译
  - 超标量处理器：编译器仅进行指令顺序调整（还是串行序列），不进行指令打包，而是由硬件根据机器结构决定同时发射哪几条指令。因此，编译后的代码能够被不同结构的机器正确执行
- 超标量多结合动态流水线调度（Dynamic pipeline scheduling）技术
  - 通过指令相关性检测和动态分支预测等手段，投机性地不按指令顺序执行，当发生流水线阻塞时，可以到后面找指令来执行（乱序执行）
  - 举例说明动态流水线调度技术：

```
lw      $t0, 20($s2)
addu    $t1, $t0, $t2
sub     $s4, $s4, $t3
slti    $t5, $s4, 20
```

左边指令序列中，哪条指令可以提前执行?

sub指令可提前到addu指令前执行

如果不将sub调到前面，则会影响slti指令的执行，而且还会发生load-use冒险

本章小结

- CPU的基本功能是周而复始地执行指令，并处理异常和中断。
- 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。
- 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。

本章小结

- 现代计算机的每个指令周期直接由时钟周期（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。
- 早期计算机中数据通路采用总线方式，通过CPU的内部总线把CPU中的通用寄存器、ALU、暂存器、指令寄存器等互连，有单总线、二总线和三总线结构数据通路。
- 现代计算机的数据通路都采用流水线方式实现，将每条指令的执行过程分解成功能段相同的几个流水段，每个流水段的执行时间也被设置成完全相同。
- 流水线方式下，同时有多条指令重叠执行，因此程序的执行时间比串行执行方式下缩短很多。在有些情况下会发生流水线冒险，包括结构冒险、数据冒险和控制冒险三类。
- 高级流水线：超流水、静态多发射（VLIW）、超标量、动态调度
- 编译器优化：静态调度、循环展开、…