



南京大學
NANJING UNIVERSITY



程序转换概述

南京大学

计算机科学与技术系

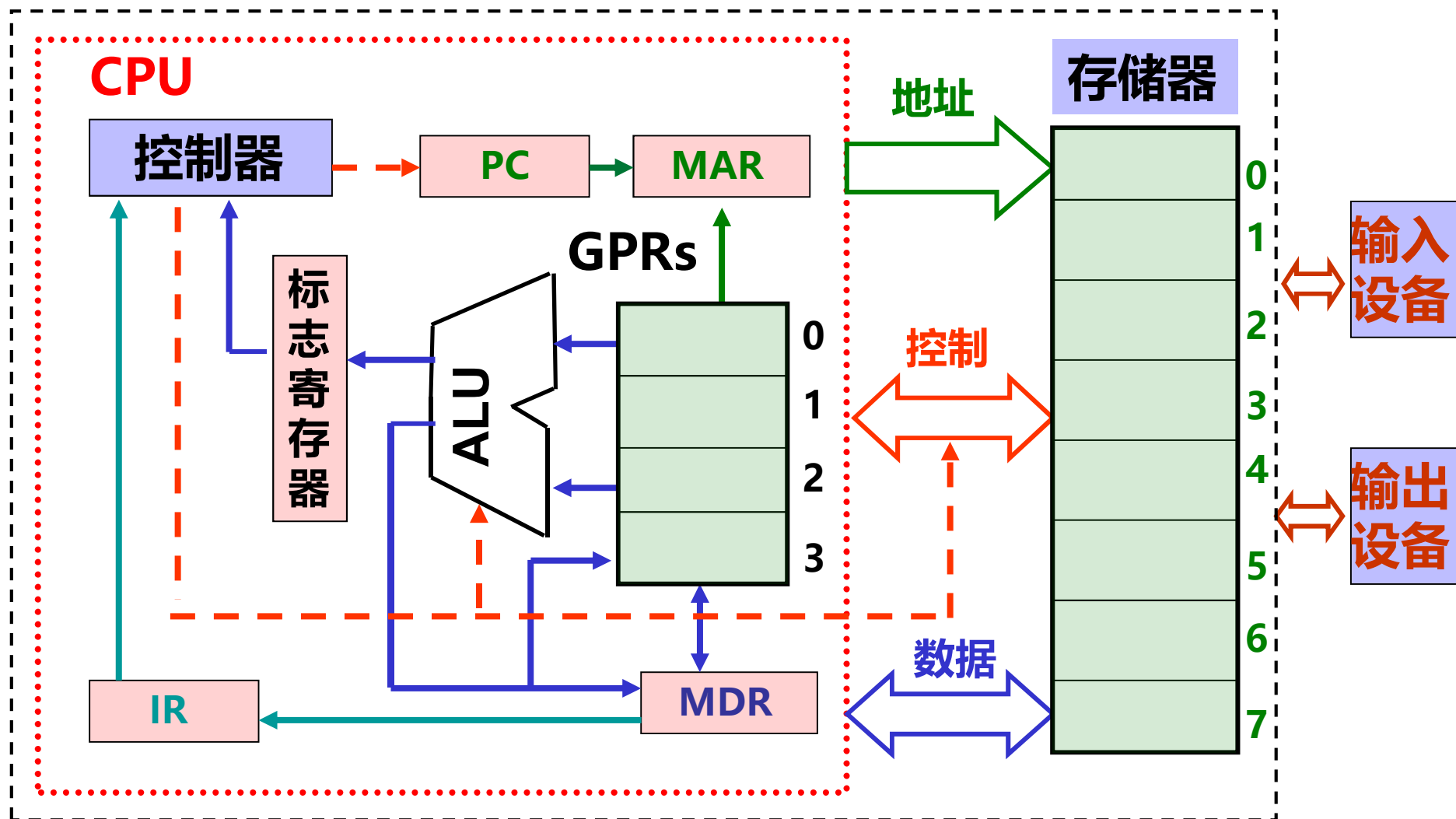
袁春风

email: cfyuan@nju.edu.cn

2015.6

回顾：计算机硬件基本组成

“存储程序”工作方式！



回顾：计算机是如何工作的？

程序由指令组成（菜单由菜谱组成）

- 程序在执行前

数据和指令事先存放在存储器中，每条指令和每个数据都有地址，指令按序存放，指令由OP、ADDR字段组成，程序起始地址置PC（原材料和菜谱都放在厨房外的架子上，每个架子有编号。妈妈从第5个架上指定菜谱开始做）

- 开始执行程序

第一步：根据PC取指令（从5号架上取菜谱）

第二步：指令译码（看菜谱）

第三步：取操作数（从架上或盘中取原材料）

第四步：指令执行（洗、切、炒等具体操作）

第五步：回写结果（装盘或直接送桌）

第六步：修改PC的值（算出下一菜谱所在架子号 $6=5+1$ ）

继续执行下一条指令（继续做下一道菜）

回顾：指令和数据

- **程序启动前**，指令和数据都存放在存储器中，形式上没有差别，都是0/1序列
- 采用“**存储程序**”工作方式：
 - 程序由指令组成，程序被启动后，计算机能自动取出一条一条指令执行，在执行过程中无需人的干预。
- **指令执行过程中**，指令和数据被从存储器取到CPU，存放在CPU内的寄存器中，指令在IR中，数据在GPR中。

指令中需给出的信息：

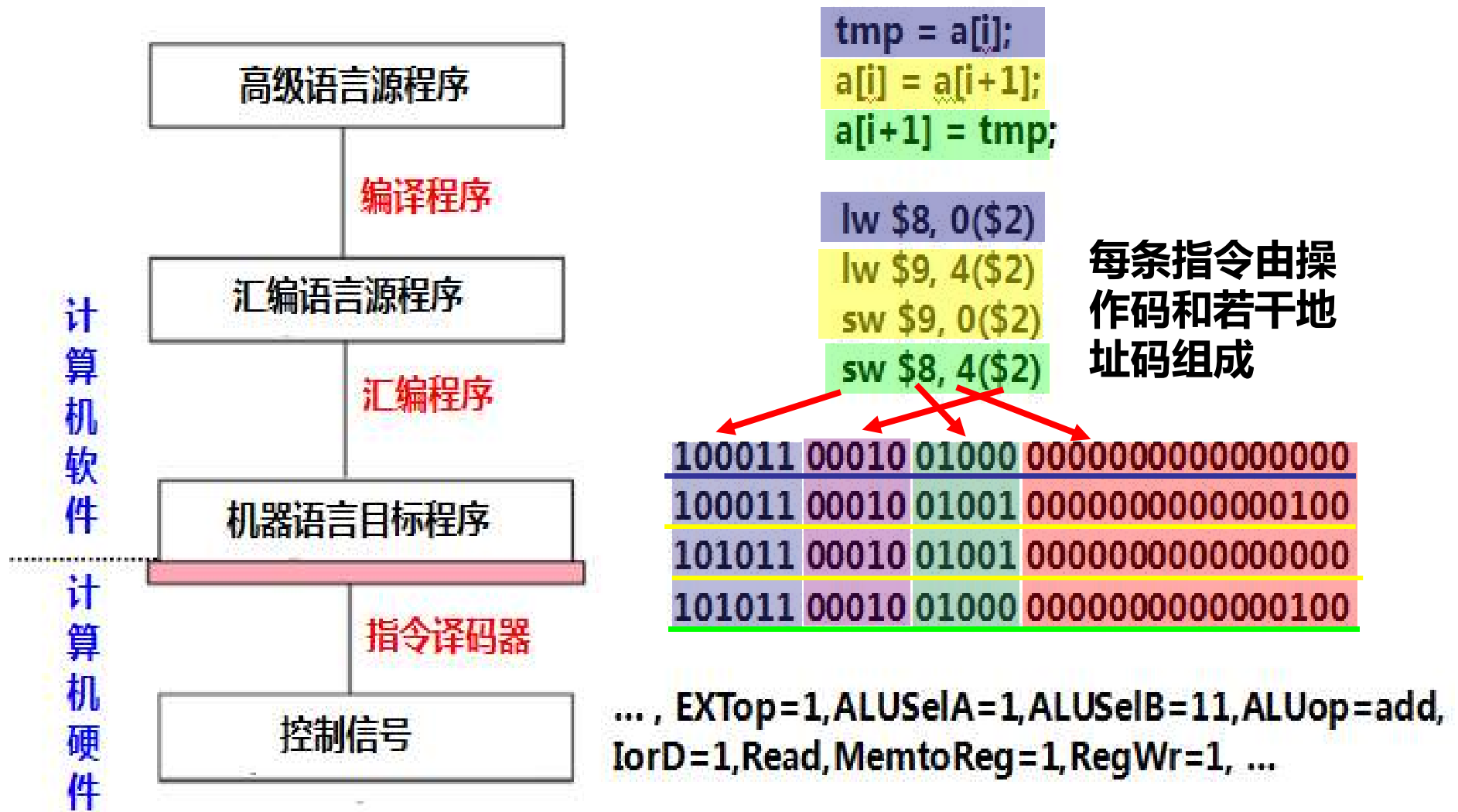
操作性质（操作码）

源操作数1 或/和 源操作数2 （立即数、寄存器编号、存储地址）

目的操作数地址 （寄存器编号、存储地址）

存储地址的描述与操作数的数据结构有关！

回顾：不同层次语言之间的等价转换



任何高级语言程序最终通过执行若干条指令来完成！

“指令”的概念

- 计算机中的指令有**微指令**、**机器指令**和**伪（宏）指令**之分
- **机器指令**处于硬件和软件的交界面
相当于一个菜谱指定的一个完整做菜过程
 - 本章中提及的指令都指机器指令
- **微指令**是微程序级命令，属于硬件范畴
相当于洗、切、煮、炒等做菜“微过程”
- **伪指令**是由若干机器指令组成的指令序列，属于软件范畴
相当于由多个菜谱合成一个“大菜”的过程
- **汇编指令**是机器指令的汇编表示形式，即符号表示
- 机器指令和汇编指令一一对应，它们都与具体机器结构有关，都属于**机器级指令**

机器级指令

- 机器指令和汇编指令一一对应，都是机器级指令
- 机器指令是一个0/1序列，由若干**字段**组成

补码**11111010**
的真值为多少？

100010 DW	mod	reg	r/m	disp8
100010 0 0	01	001	001	11111010

操作码

寻址方式

寄存器编号

立即数(位移量)

- 汇编指令是机器指令的符号表示（**可有不同格式**）

mov [bx+di-6], cl

Intel格式

或

movb %cl, -6(%bx,%di)

AT&T 格式

长度后缀

指令的功能为： **$M[R[bx] + R[di] - 6] \leftarrow R[cl]$**

寄存器传送语言 RTL (Register Transfer Language)

mov、bx、movb、%bx等都是助记符

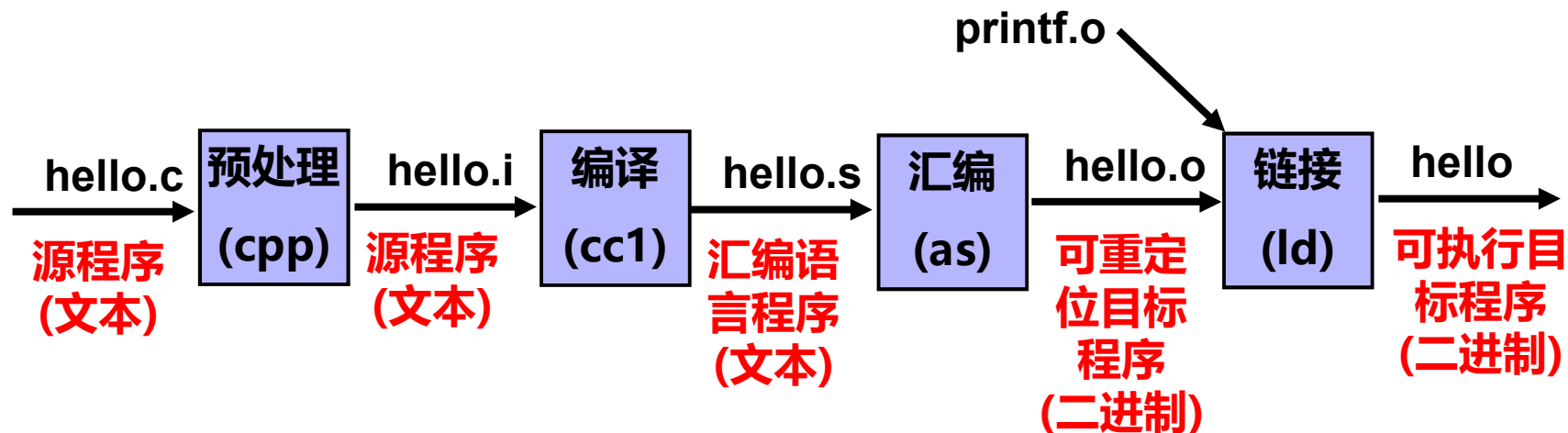
R：寄存器内容

M：存储单元内容

注：也可用(x)表示
地址x中的内容

回顾：高级语言程序转换为机器代码的过程

用GCC编译器套件进行转换的过程



预处理：在高级语言源程序中插入所有用`#include`命令指定的文件和用`#define`声明指定的宏。

编译：将预处理后的源程序文件编译生成相应的汇编语言程序。

汇编：由汇编程序将汇编语言源程序文件转换为可重定位的机器语言目标代码文件。

链接：由链接器将多个可重定位的机器语言目标文件以及库例程（如`printf()`库函数）链接起来，生成最终的可执行目标文件。

GCC使用举例

```
1 // test.c
2 int add(int i, int j)
3 {
4     int x = i + j;
5     return x;
6 }
```

源文件main.c和test.c，最终生成可执行文件为test

gcc -O1 main.c test.c -o test

表示一级优化，-O2为二级优化，选项-o指出输出文件名

- 目标文件可用“**objdump -d test.o**”
反汇编为汇编语言程序

gcc -E test.c -o test.i
gcc -S test.i -o test.s
test.s **gcc -S test.c -o test.s**

```
add:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 12(%ebp), %eax
movl 8(%ebp), %edx
leal (%edx, %eax), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
leave
ret
```

00000000 <add>: **gcc -c test.s -o test.o**

0:	55	push %ebp
1:	89 e5	mov %esp, %ebp
3:	83 ec 10	sub \$0x10, %esp
6:	8b 45 0c	mov 0xc(%ebp), %eax
9:	8b 55 08	mov 0x8(%ebp), %edx
c:	8d 04 02	leal (%edx,%eax,1), %eax
f:	89 45 fc	mov %eax, -0x4(%ebp)
12:	8b 45 fc	mov -0x4(%ebp), %eax
15:	c9	leave
16:	c3	ret

位移量

机器指令

汇编指令

编译得到的与反汇编得到的汇编指令形式稍有差异

两种目标文件

```
1 // test.c
2 int add(int i, int j )
3 {
4     int x = i +j;
5     return x;
6 }
```

test.o : 可重定位目标文件

test : 可执行目标文件

“objdump -d test.o” 结果

00000000 <add>:

0:	55	push	%ebp
1:	89 e5	mov	%esp, %ebp
3:	83 ec 10	sub	\$0x10, %esp
6:	8b 45 0c	mov	0xc(%ebp), %eax
9:	8b 55 08	mov	0x8(%ebp), %edx
c:	8d 04 02	lea	(%edx,%eax,1), %eax
f:	89 45 fc	mov	%eax, -0x4(%ebp)
12:	8b 45 fc	mov	-0x4(%ebp), %eax
15:	c9	leave	
16:	c3	ret	

“objdump -d test” 结果

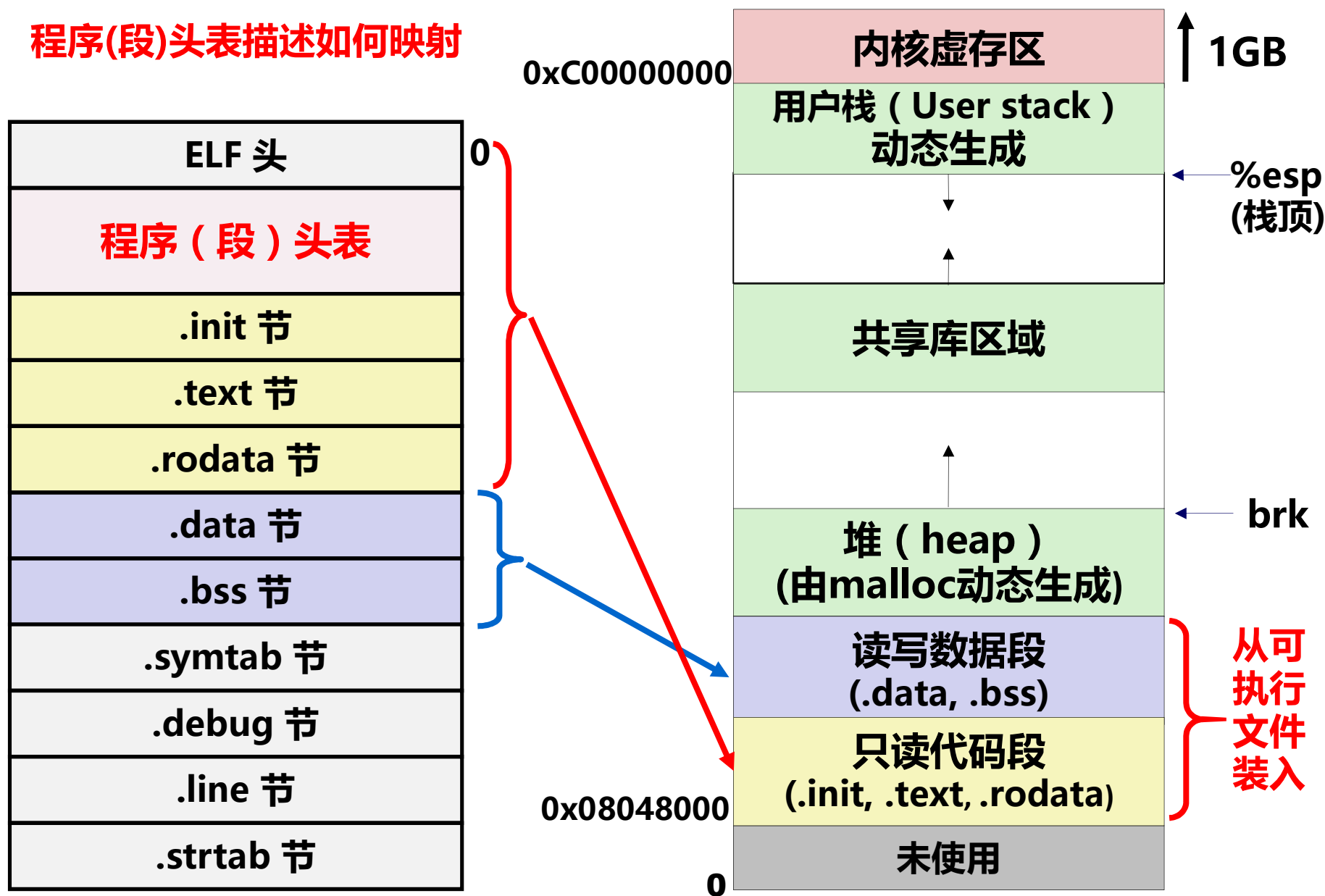
080483d4 <add>:

80483d4:	55	push ...
80483d5:	89 e5	...
80483d7:	83 ec 10	...
80483da:	8b 45 0c	...
80483dd:	8b 55 08	...
80483e0:	8d 04 02	...
80483e3:	89 45 fc	...
80483e6:	8b 45 fc	...
80483e9:	c9	...
80483ea:	c3	ret

test.o中的代码从地址0开始，test中的代码从80483d4开始！

可执行文件的存储器映像

程序(段)头表描述如何映射



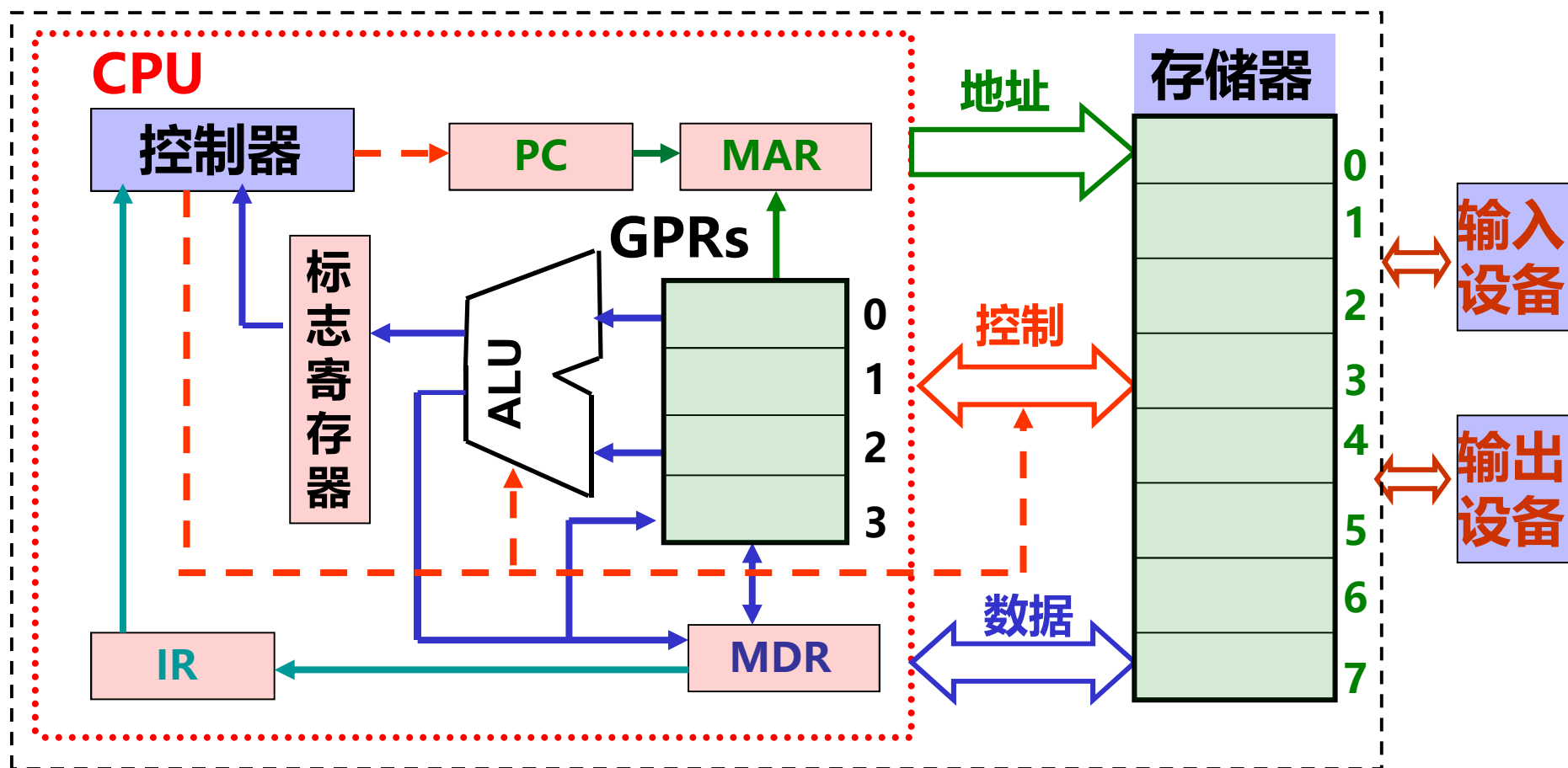
回顾：指令集体系结构（ISA）

- ISA指Instruction Set Architecture，即**指令集体系结构**
- ISA是一种规约（Specification），它规定了**如何使用硬件**
 - 可执行的指令的集合，包括**指令格式、操作种类以及每种操作对应的操作数的相应规定**；
 - 指令可以接受的**操作数的类型**；
 - 操作数所能存放的寄存器组的结构，包括每个寄存器的**名称、编号、长度和用途**；
 - 操作数所能存放的**存储空间的大小和编址方式**；
 - 操作数在存储空间存放时按照**大端还是小端方式存放**；
 - 指令获取操作数的方式，即**寻址方式**；
 - 指令执行过程的控制方式，包括**程序计数器、条件码定义等**。
- ISA在计算机系统中是必不可少的一个**抽象层**，Why？
 - 没有它，软件无法使用计算机硬件！
 - 没有它，一台计算机不能称为“通用计算机”

微体系结构

ISA和计算机组成（Organization，即MicroArchitecture）是何关系？

ISA和计算机组成（微结构）之间的关系



ISA是计算机组成的抽象

不同ISA规定的指令集不同，如，IA-32、MIPS、ARM等
计算机组成必须能够实现ISA规定的功能，如提供GPR、标志、运算电路等
同一种ISA可以有不同的计算机组成，如乘法指令可用ALU或乘法器实现

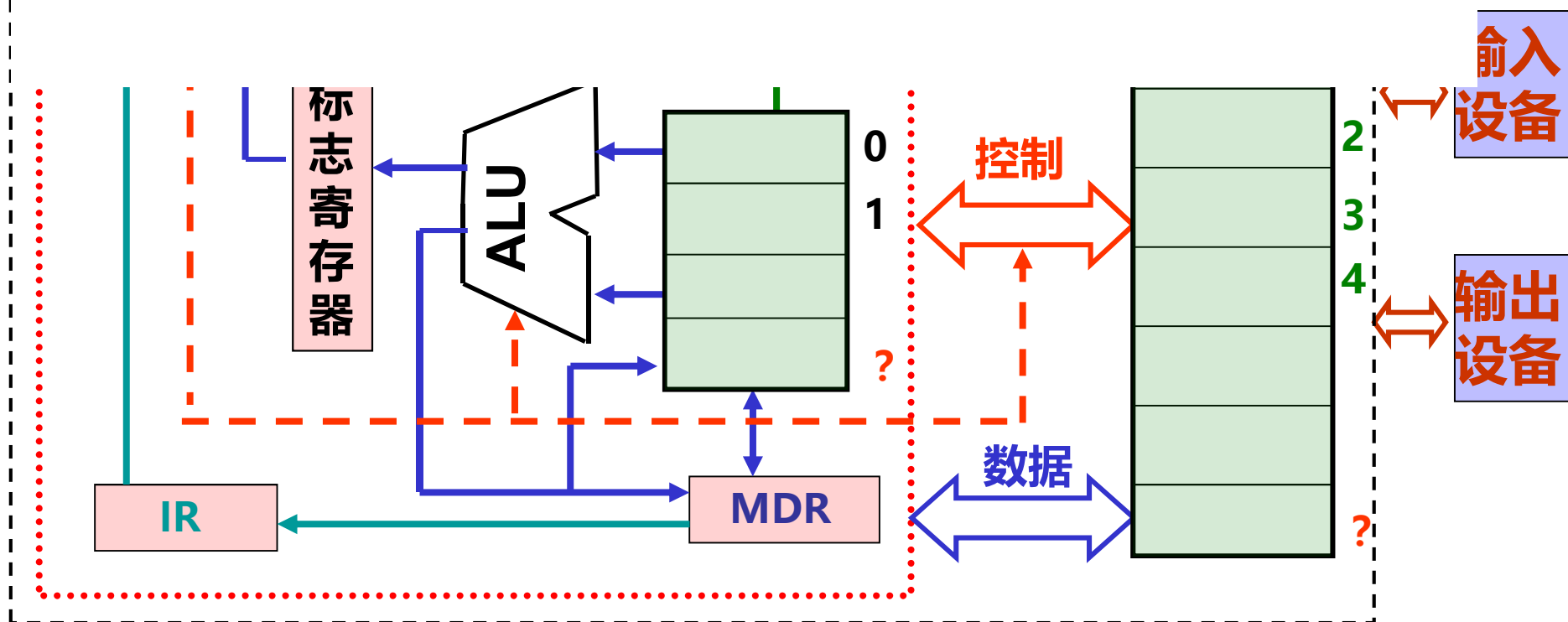
IA-32的体系结构是怎样的呢？

寄存器个数及各自功能？寄存器宽度？存储空间大小？编址单位？

指令格式？指令条数？指令操作功能？寻址方式？数据类型？

小端/大端？标志寄存器各位含义？PC位数？I/O端口编址方式？.....

下一节课开始介绍 IA-32 的指令集体系结构（ISA）



总结

- 高级语言程序总是转换为机器代码才能在机器上执行
- 转换过程：预处理、编译、汇编、链接
- 机器代码是二进制代码，可DUMP为汇编代码表示
- ISA规定了一台机器的指令系统涉及到的所有方面

例如：

- 所有指令的指令格式、功能
- 通用寄存器的个数、位数、编号和功能
- 存储地址空间大小、编址方式、大/小端
- 指令寻址方式



南京大學
NANJING UNIVERSITY



IA-32指令系统概述

南京大学

计算机科学与技术系

袁春风

email: cfyuan@nju.edu.cn

2015.6

Intel处理器

x86前产品	4004 • 4040 • 8008 • 8080 • iAPX 432 • 8085	已停产
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	现有产品
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)	现有产品
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	

IA-32/x64指令系统概述

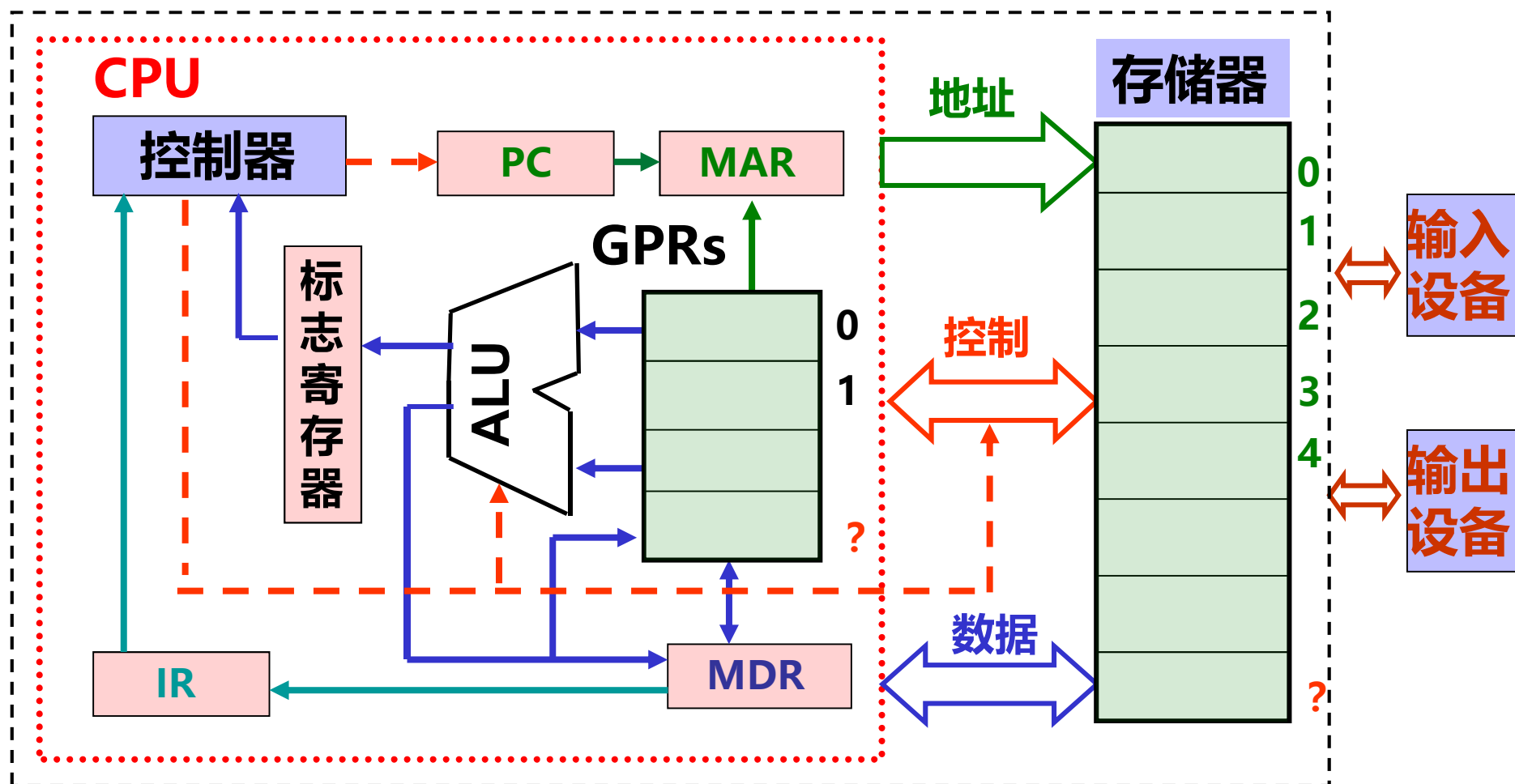
- x86是Intel开发的一类处理器体系结构的泛称
 - 包括 Intel 8086、80286、i386和i486等，因此其架构被称为“x86”
 - 由于数字并不能作为注册商标，因此，后来使用了可注册的名称，如Pentium、PentiumPro、Core 2、Core i7等
 - 现在Intel把32位x86架构的名称x86-32改称为IA-32
 - IA是Intel Architecture的缩写
- 由AMD首先提出了一个兼容IA-32指令集的64位版本
 - 扩充了指令及寄存器长度和个数等，更新了参数传送方式
 - AMD称其为AMD64，Intel称其为Intel64（不同于IA-64）
 - 命名为“x86-64”，有时也简称为x64

IA-32的体系结构是怎样的呢？

寄存器个数及各自功能？寄存器宽度？存储空间大小？编址单位？

指令格式？指令条数？指令操作功能？寻址方式？数据类型？

小端/大端？标志寄存器各位含义？PC位数？I/O端口编址方式？.....



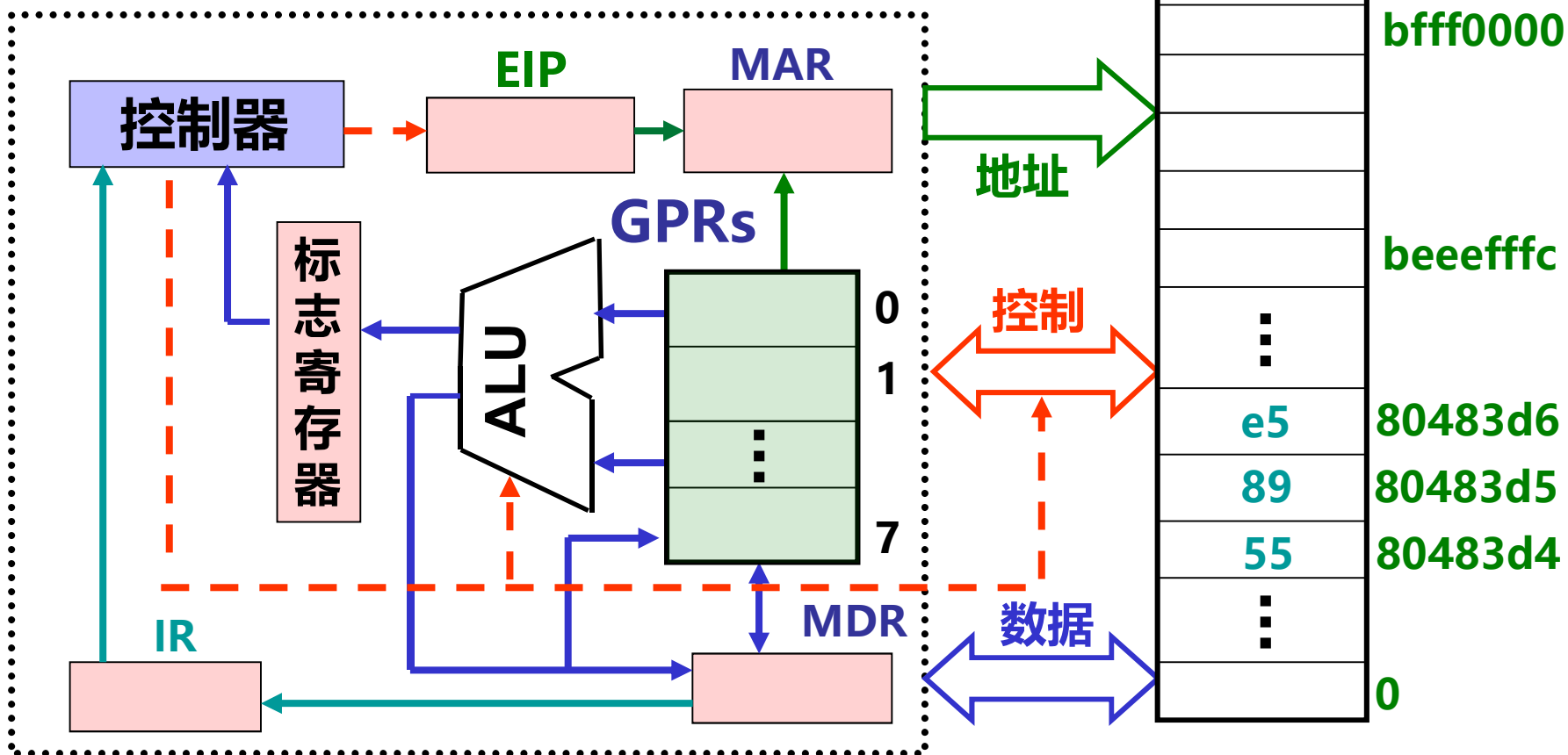
IA-32的体系结构是怎样的呢？

8个GPR (0~7) , 一个EFLAGS, PC为EIP

可寻址空间4GB (编号为0~0xFFFFFFFF)

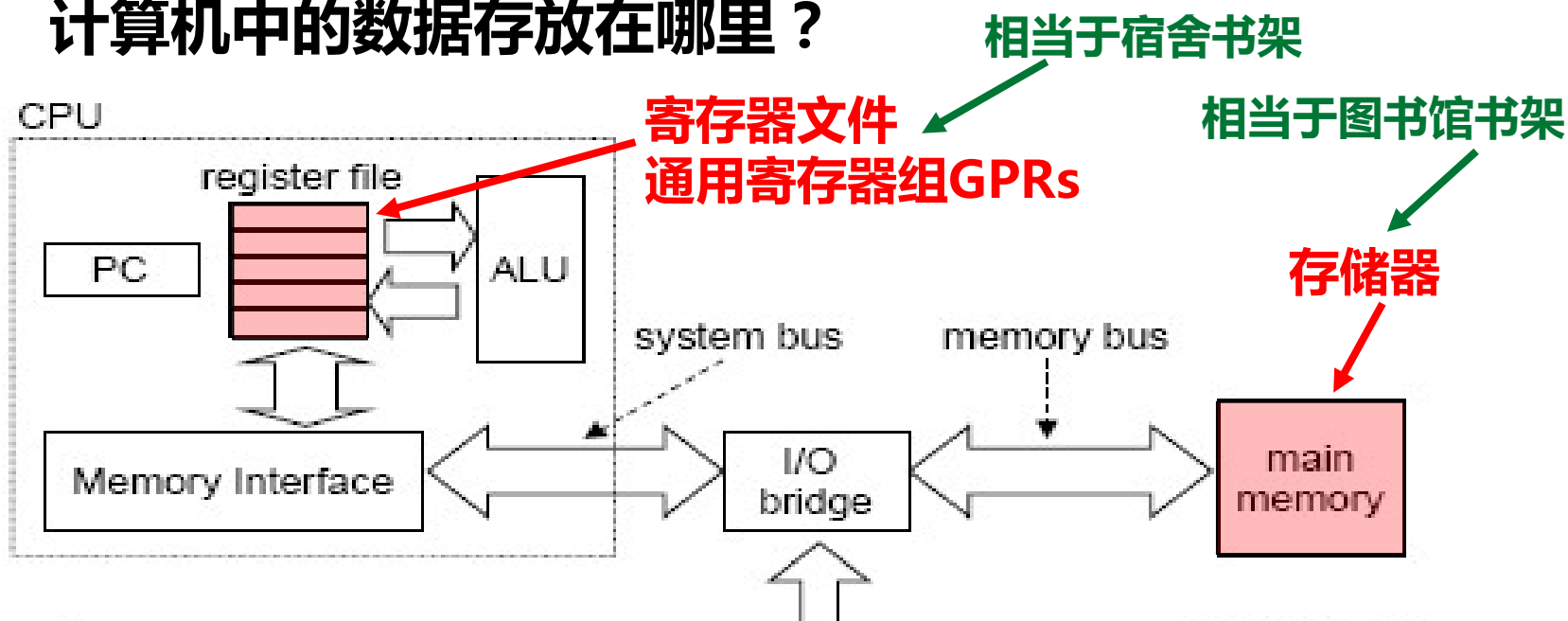
指令格式变长, 操作码变长, 指令由若干字段
(OP、Mod、SIB等) 组成

.....



计算机中数据的存储

- 计算机中的数据存放在哪里？



指令中需给出的信息：

操作性质（操作码）

源操作数1 或/和 源操作数2 （立即数、寄存器编号、存储地址）

目的操作数地址 （寄存器编号、存储地址）

存储地址的描述与操作数的数据结构有关！

IA-32支持的数据类型及格式

C 语言声明	Intel 操作数类型	汇编指令长度后缀	存储长度 (位)
(unsigned) char	整数 / 字节	b	8
(unsigned) short	整数 / 字	w	16
(unsigned) int	整数 / 双字	l	32
(unsigned) long int	整数 / 双字	l	32
unsigned) long long int	-	-	2×32
char *	整数 / 双字	l	32
float	单精度浮点数	s	32
double	双精度浮点数	l	64
long double	扩展精度浮点数	t	80 / 96

IA-32架构由16位架构发展而来，因此，虽然字长为32位或更大，但一个字为16位，长度后缀为 w；32位为双字，长度后缀为 l
long double实际长度为80位，但分配96位=12B（按4B对齐）

IA-32的寄存器组织

	31	16	15	8	7	0	
EAX				AH	(AX)	AL	累加器
EBX				BH	(BX)	BL	基址寄存器
ECX				CH	(CX)	CL	计数寄存器
EDX				DH	(DX)	DL	数据寄存器
ESP				SP			堆栈指针
EBP				BP			基址指针
ESI				SI			源变址寄存器
EDI				DI			目标变址寄存器
EIP				IP			指令指针
EFLAGS				FLAGS			标志寄存器
				CS			代码段
				SS			堆栈段
				DS			数据段
				ES			附加段
				FS			附加段
				GS			附加段

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

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		0	D	I	T	S	Z	0	A	0	P	1	C

← 80286/386
← 8086 →

- 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

SKIP

- 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：计算机中的算盘长啥样？

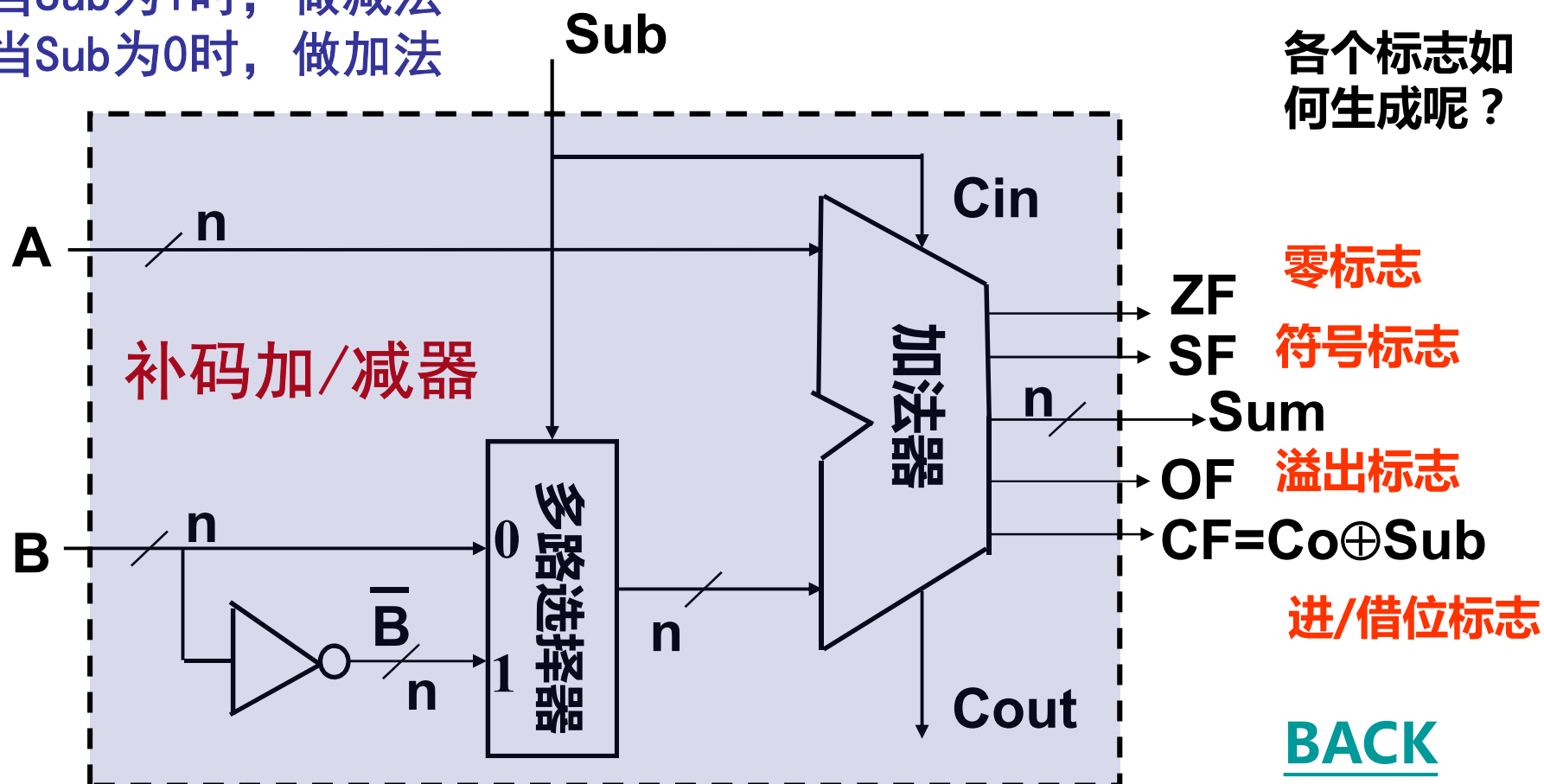
重要认识1：计算机中所有运算都基于加法器实现！

计算机中的算盘就是加法器！

当Sub为1时，做减法
当Sub为0时，做加法

重要认识2：加法器不知道所运算的是带符号数还是无符号数。

重要认识3：加法器不判定对错，总是取低n位作为结果，并生成标志信息。



IA-32的寻址方式

- 寻址方式
 - 如何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： $(CS) \ll 4 + (IP)$
- 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以上微处理器的工作模式
 - 寻址空间为 $2^{32}B$ ，32位线性地址分段（段基址+段内偏移量）

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	$LA = (SR) + A$		} 存储器操作数
基址寻址	$LA = (SR) + (B)$		
基址加位移	$LA = (SR) + (B) + A$		
比例变址加位移	$LA = (SR) + (I) \times S + A$		
基址加变址加位移	$LA = (SR) + (B) + (I) + A$		
基址加比例变址加位移	$LA = (SR) + (B) + (I) \times S + A$		
相对寻址	$LA = (PC) + A$	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

SKIP

IA-32的寄存器组织

	31	16	15	8	7	0	
EAX				AH	(AX)	AL	累加器
EBX				BH	(BX)	BL	基址寄存器
ECX				CH	(CX)	CL	计数寄存器
EDX				DH	(DX)	DL	数据寄存器
ESP				SP			堆栈指针
EBP				BP			基址指针
ESI				SI			源变址寄存器
EDI				DI			目标变址寄存器
EIP				IP			指令指针
EFLAGS				FLAGS			标志寄存器

	CS	代码段
	SS	堆栈段
	DS	数据段
	ES	附加段
	FS	附加段
	GS	附加段

返回寻址方式

返回寻址方式

存储器操作数的寻址方式

int x;

float a[100];

short b[4][4];

char c;

double d[10];

Linux系统：
double型变量
按4B边界对齐

windows系统：
double型变量
按8B边界对齐

a[i]的地址如何计算？

104+i×**4**

i=99时，104+99×4=500

b[i][j]的地址如何计算？

504+i×**8**+j×**2**

i=3、j=2时，504+24+4=532

d[i]的地址如何计算？

544+i×**8**

i=9时，544+9×8=616

b31		b0	
d[9]			616
⋮			
d[0]			544
		c	536
b[3][3]	b[3][2]		532
⋮			
b[0][1]	b[0][0]		504
a[99]			500
⋮			
a[0]			104
x			100
⋮			

存储器操作数的寻址方式

```
int x;
float a[100];
short b[4][4];
char c;
double d[10];
```

各变量应采用什么寻址方式？

x、c：位移 / 基址

a[i]： $104 + i \times 4$ ，比例变址+位移

d[i]： $544 + i \times 8$ ，比例变址+位移

b[i][j]： $504 + i \times 8 + j \times 2$ ，
基址+比例变址+位移

将**b[i][j]**取到AX中的指令可以是：

“**movw** **504**(%ebp,%esi,**2**), %ax”

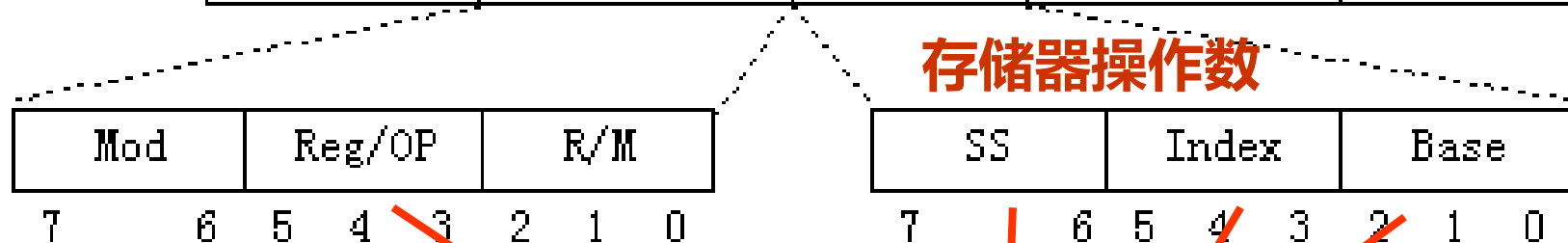
其中， $i \times 8$ 在EBP中，j在ESI中，

2为比例因子

b31		b0	
d[9]			616
⋮			
d[0]			544
		c	536
b[3][3]	b[3][2]		532
⋮			
b[0][1]	b[0][0]		504
a[99]			500
⋮			
a[0]			104
x			100
⋮			

IA-32机器指令格式

指令段:	操作码	寻址方式	SIB	位移	直接数据
字节数:	1或2	0或1	0或1	1、2、4	立即数



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 leal (%edx,%eax,1), %eax

1000 1101 00 000 100 00 000 010

总结

- IA-32是典型的CISC (复杂指令集计算机) 风格ISA
 - 8个通用寄存器 (8位、16位、32位)
 - 2个专用寄存器 : EIP (PC)、标志寄存器EFLAGS
 - 6个段寄存器 (间接给出段基址)
 - 存储器地址空间为4GB , 按字节编址 , 小端方式
 - 寻址方式
 - 立即、寄存器、存储器 (段基址+有效地址 (偏移量))
 - 相对寻址
 - 变长指令字、变长操作码
- 汇编语言格式
 - Intel格式汇编
 - AT&T格式汇编 (本课程使用)

8(%edx,%eax,4)

