计 算 机 组 成 与 结 构

彭柯鑫
pkx@cdut.edu.cn

第二章 指令系统体系结构

- 1.设计自己的计算机
- 2. x86体系结构
- 3. x86指令简介
- 4.复杂的x86指令举例
- 5. MIPS体系结构
- 6. MIPS指令简介

硬件攻城狮

指令系统
体系结构

软件程序猿

一个简单的计算机指令系统

ADD R, M

- 功能：将R的内容与M中的内容相加后存入R

运算类指令

LOAD R, M

- 功能：将M中的内容装入R

传送类指令

STORE M, R

- 功能：将R的内容存入M中

JMP L

- 功能：无条件转向L处

转移类指令

*注：M和L为存储器地址，R为寄存器编号

指令的格式

每条指令等长，均为2个字节

第一个字节的高4位是操作码

- LOAD：0000；ADD：0001
- STORE：0010；JMP：0011
- 目前只提供4条指令，最多可扩展到16条

第一个字节的低4位是寄存器号

- R0~R3：0000~0011
- 目前只提供4个寄存器，最多可扩展到16个

第二个字节是存储单元地址

- 最大可以使用256个字节的存储器

指令示例

0001 0010	0000 1001
ADD R2, [9]	

错误指令示例

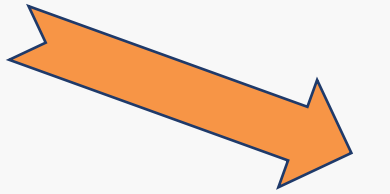
0101 1010	0000 1001
操作码未定义，寄存器号未定义	

运算任务示例

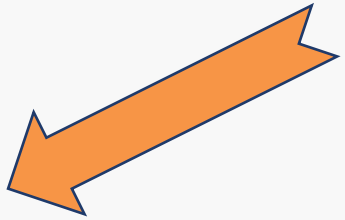
运算任务示例

1. 将M1的内容与M2的内容相加后存入M3
2. 完成运算后，程序转向L处的指令继续执行

- M1、M2、M3和L均为存储单元的地址



可用的指令
ADD R, M
LOAD R, M
STORE M, R
JMP L



程序描述：

1. 将M1的内容送入某个寄存器，记为Rx
2. 将Rx的内容与M2的内容相加，运算结果存入Rx
3. 将Rx的内容送入M3中
4. 转移到L，取出下一条指令继续执行

运算任务对应的程序

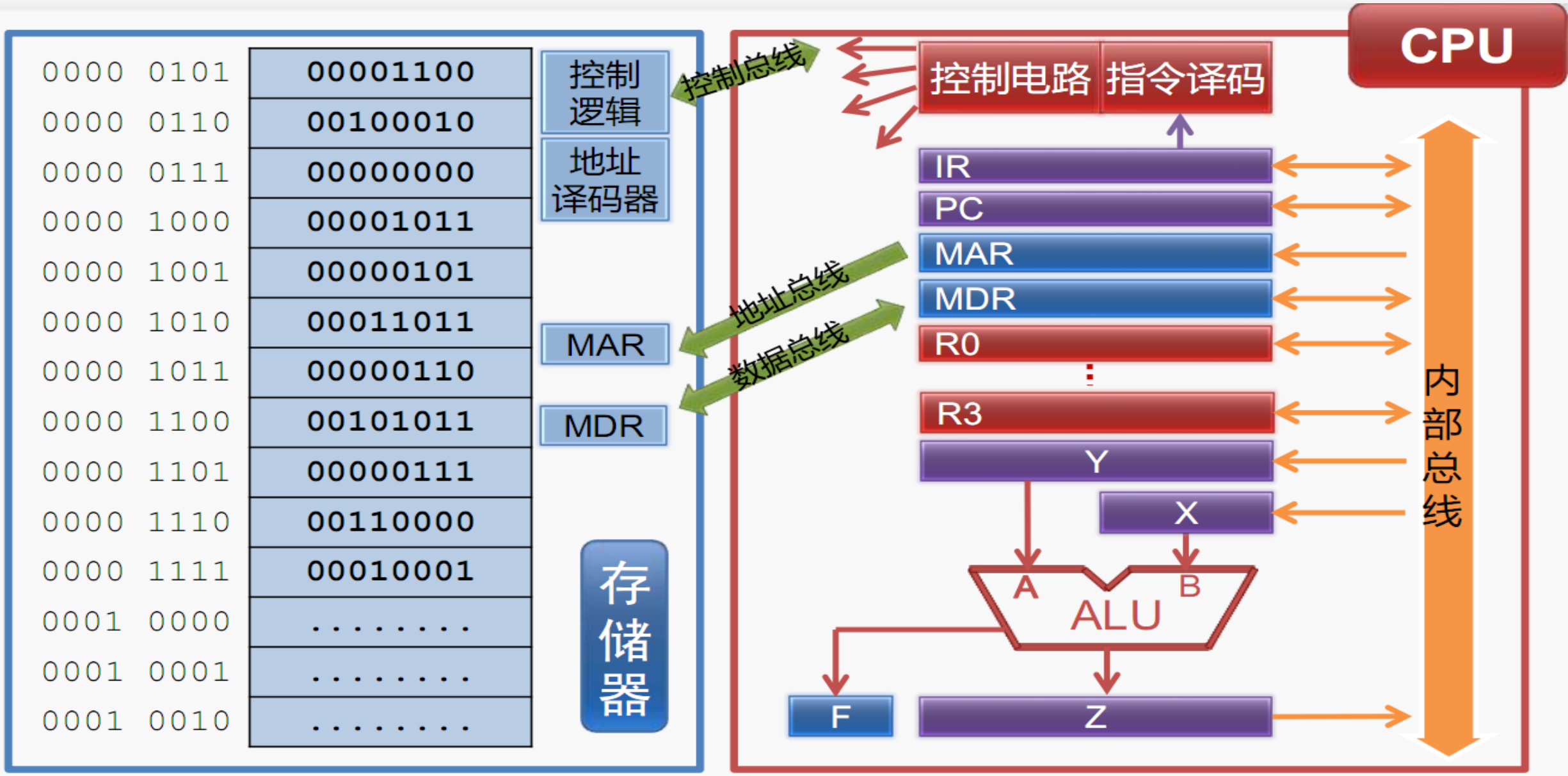
设：M1=5，M2=6，M3=7，L=18

汇编语言程序	机器语言程序	程序的功能
LOAD R3, [5]	00000011 00000101	将存储单元[5]的内容送入寄存器R3
ADD R3, [6]	00010011 00000110	将寄存器R3的内容加上存储单元[6]的内容，再送回R3
STORE [7], R3	00100011 00000111	将寄存器R3的内容送入存储单元[7]中
JMP [18]	00110000 00010010	转向存储单元[18]，取出指令继续执行

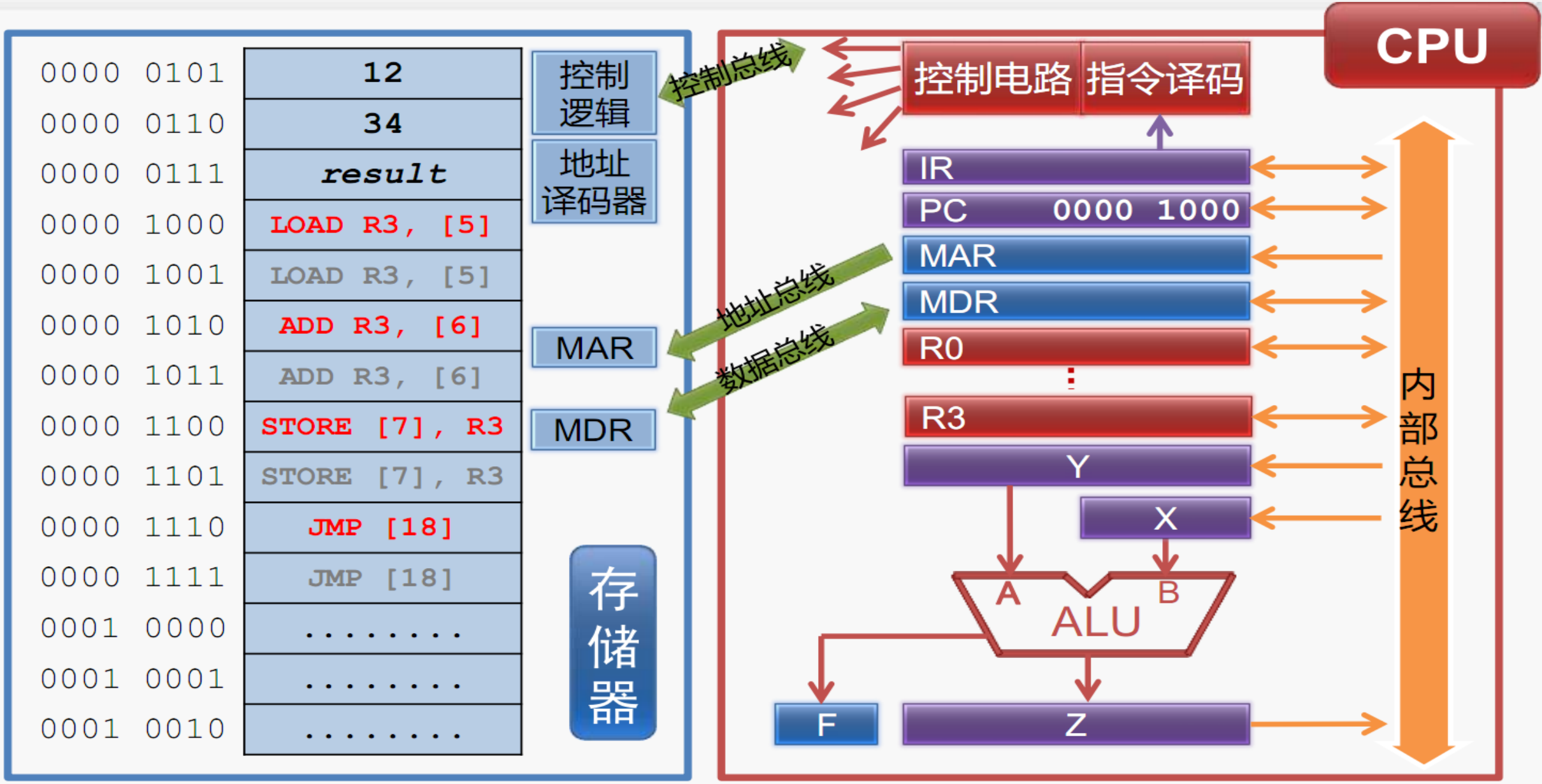
在存储器中的机器语言程序

存储器地址	存储器内容	说明
0000 0101	00001100	地址 [5]，存放了数据12
0000 0110	00100010	地址 [6]，存放了数据34
0000 0111	00000000	地址 [7]，准备存放运算结果
0000 1000	00001011	“LOAD R3, [5]”的第一个字节
0000 1001	00000101	“LOAD R3, [5]”的第二个字节
0000 1010	00011011	“ADD R3, [6]”的第一个字节
0000 1011	00000110	“ADD R3, [6]”的第二个字节
0000 1100	00101011	“STORE [7], R3”的第一个字节
0000 1101	00000111	“STORE [7], R3”的第二个字节
0000 1110	00110000	“JMP [18]”的第一个字节
0000 1111	00010001	“JMP [18]”的第二个字节
0001 0000	.....	第五条指令的第一个字节
0001 0001	.....	第五条指令的第二个字节
0001 0010	.....	第六条指令的第一个字节（地址 [18]）

模型机的CPU和存储器



模型机准备开始运行


0000 0101
12
0000 0110
34
0000 0111
result
0000 1000
LOAD R3, [5]
0000 1001
LOAD R3, [5]
0000 1010
ADD R3, [6]
0000 1011
ADD R3, [6]
0000 1100
STORE [7], R3
0000 1101
STORE [7], R3
0000 1110
JMP [18]
0000 1111
JMP [18]
0001 0000
0001 0001
0001 0010
控制逻辑
地址译码器
MAR
MDR
存储器
控制总线
地址总线
数据总线
CPU
控制电路
指令译码
IR
PC
0000 1000
MAR
MDR
R0
R3
Y
X
A
B
ALU
F
Z
内部总线

第二章 指令系统体系结构

- 1.设计自己的计算机
- 2. x86体系结构
- 3. x86指令简介
- 4.复杂的x86指令举例
- 5. MIPS体系结构
- 6. MIPS指令简介

x86体系结构

体系结构		厂商	微处理器型号	字长	年代
x86	“x86-16” “IA-16”	Intel	8086, 8088, 80186, 80188 80286	16位	1978年起
	IA-32	Intel	80386, 80486, Pentium, Pentium Pro/II/III/4, Core, Atom	32位	1985年起
		AMD	Am386, Am486, AM5x86, K5, K6, Athlon		
		Others	Cyrix 5x86; VIA C3/C7 Transmeta Crusoe, Efficeon		
	x86-64	AMD	Opteron, Athlon 64 Phenom, Phenom II	64位	2003年起
		Intel	Pentium 4 Prescott, Core 2 Core i3/i5/i7		
		Others	VIA Nano		

Intel 8086（1978年）

8086的主要特点

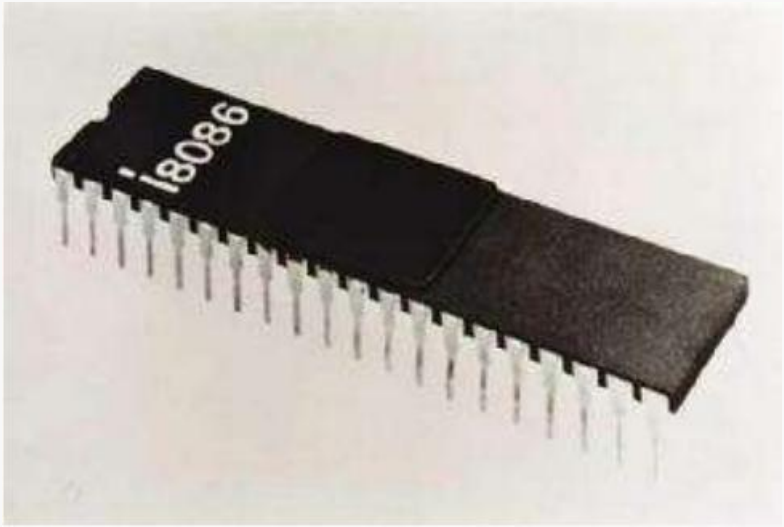
① 内部的通用寄存器为16位

既能处理16位数据，也能处理8位数据

② 对外有16根数据线和20根地址线

可寻址的内存空间为1MByte（2^{20}）

③ 物理地址的形成采用“段加偏移”的方式



Intel 8086（1978年）

8086的主要特点

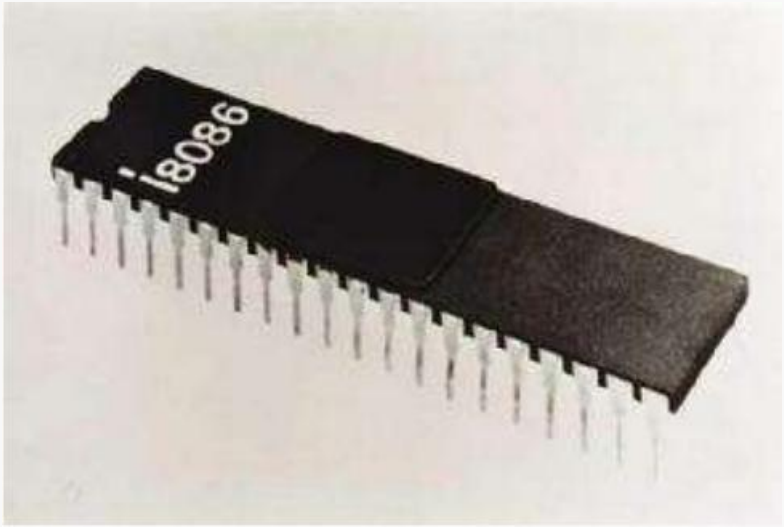
① 内部的通用寄存器为16位

既能处理16位数据，也能处理8位数据

② 对外有16根数据线和20根地址线

可寻址的内存空间为1MByte（2^{20}）

③ 物理地址的形成采用“段加偏移”的方式



微型计算机的早期代表：IBM PC

1981年，IBM PC 5150诞生

- 售价约1600美元
- Intel 8088 CPU，主频4.77MHz，内存16KB
- 因开放性架构逐渐成为个人计算机的制造标准

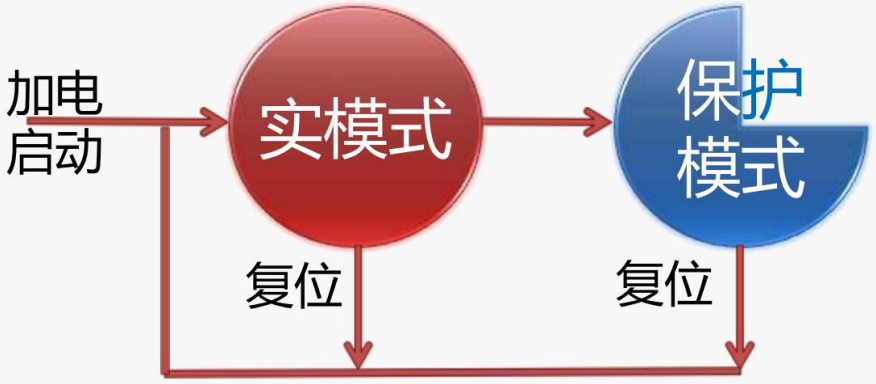
Intel 8088 CPU，1979年推出。8088是8086的简化版本，主要区别是数据总线只有8位宽。

Intel 80286（1982年）

80286的主要特点

- 地址总线扩展到24位，可寻址16MB的内存空间
- 引入了“保护模式”，但是机制有缺陷

  *例如，每个段仍为64KB，严重限制软件规模
- 为保持兼容，保留了8086的工作模式，被称为“实模式”

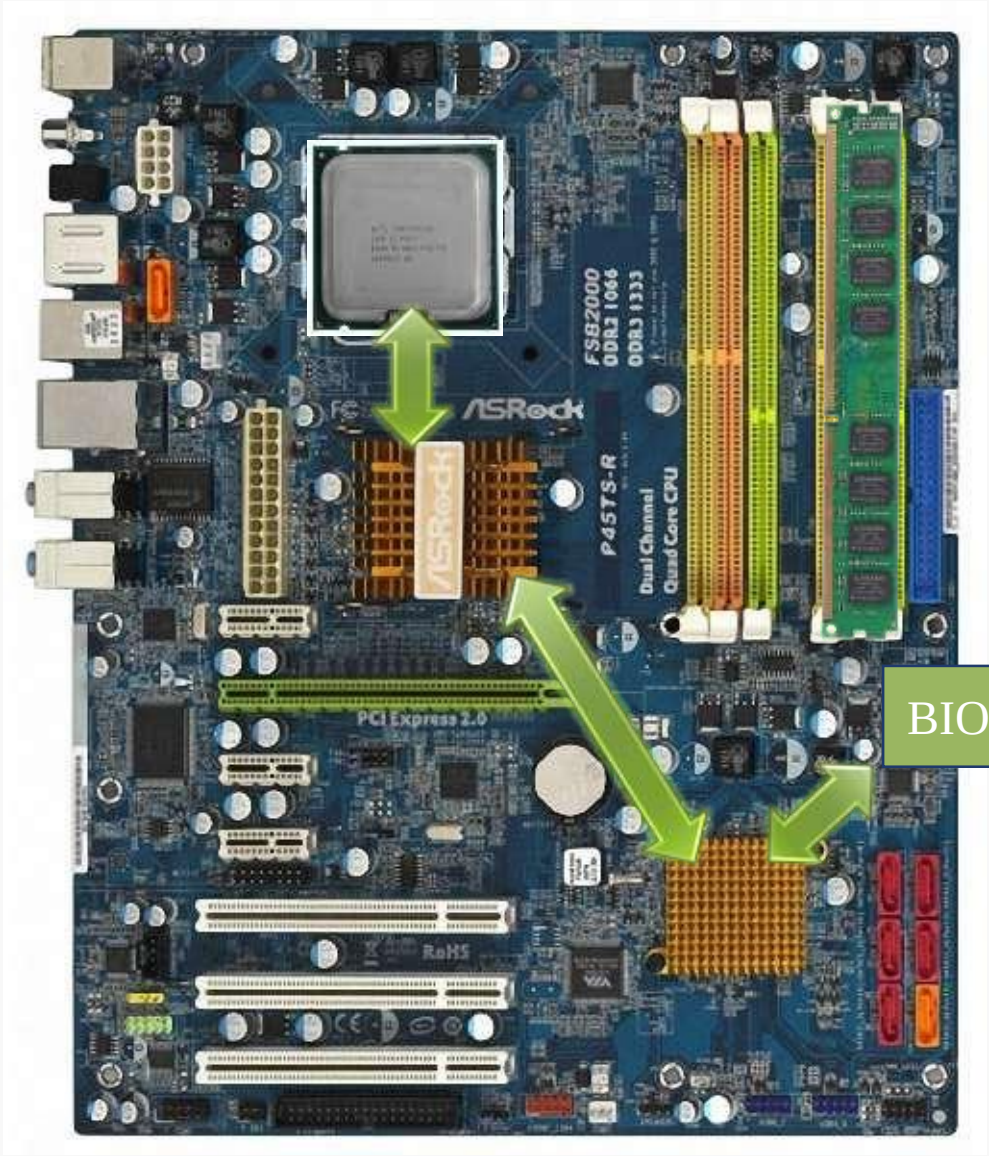


80286
主频6~20MHz
13.4万个晶体管

实模式（Real Mode）

实模式，又称“实地址模式”

- 80286及以上的微处理器采用8086的工作模式，即为实模式
- 运行在实模式下的80x86微处理器像是一个更快的8086
- 为兼容8086，所有x86处理器在加电或复位后首先进入实模式
- 系统初始化程序在实模式下运行，为进入保护模式做好准备



x86体系结构

体系结构		厂商	微处理器型号	字长	年代
x86	“x86-16” “IA-16”	Intel	8086, 8088, 80186, 80188 80286	16位	1978年起
	IA-32	Intel	80386, 80486, Pentium, Pentium Pro/II/III/4, Core, Atom	32位	1985年起
		AMD	Am386, Am486, AM5x86, K5, K6, Athlon		
		Others	Cyrix 5x86; VIA C3/C7 Transmeta Crusoe, Efficeon		
	x86-64	AMD	Opteron, Athlon 64 Phenom, Phenom II	64位	2003年起
		Intel	Pentium 4 Prescott, Core 2 Core i3/i5/i7		
		Others	VIA Nano		

Intel 80386（1985年）

80386的主要特点

- 80x86系列中的第一款32位微处理器
- 支持32位的算术和逻辑运算，提供32位的通用寄存器
- 地址总线扩展到32位，可寻址4GB的内存空间
- 改进了“保护模式”（例如，段范围可达4GB）
- 增加了“虚拟8086模式”，可以同时模拟多个8086微处理器

80386
主频12.5~33MHz
27.5万个晶体管

保护模式（Protected Mode）

保护模式，可简写为“pmode”

- 80386及以上的微处理器的主要工作模式
- 支持多任务
- 支持设置特权级
- 支持特权指令的执行
- 支持访问权限检查
- 可以访问4GB的物理存储空间
- 引入了虚拟存储器的概念

保护模式让操作系统加强了对应用软件的控制，使得系统运行更安全高效

虚拟8086模式（Virtual 8086 Mode）

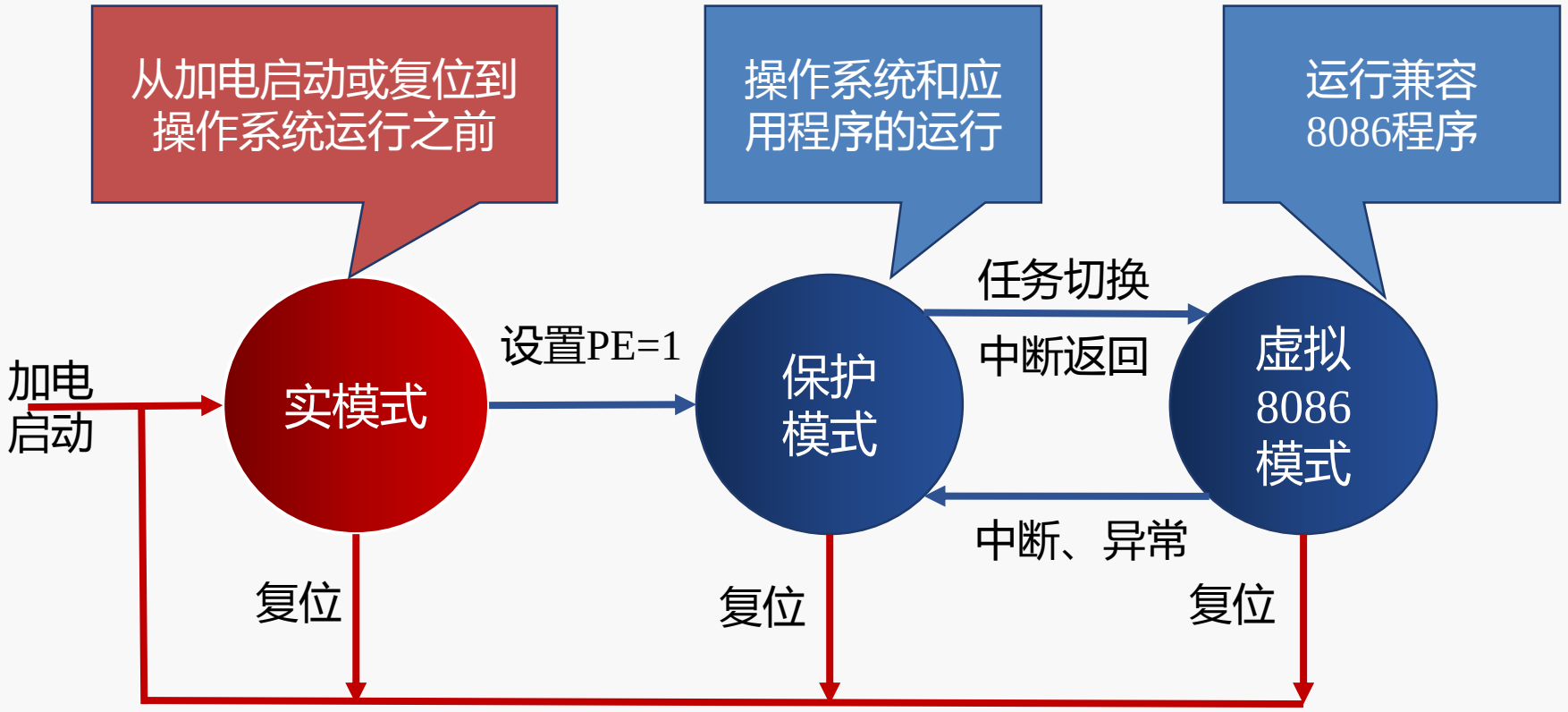
虚拟8086模式，又称“V86模式”

- V86模式实际上是保护模式下一种特殊工作状态
- V86模式下的微处理器类似于8086，但不等同

V86模式 与 实模式 的比较

- 相同点
  - 可寻址的内存空间为1MB
  - “段加偏移”的寻址方式
- 不同点
  - 对中断/异常的响应处理

三种工作模式之间的转换



*注：PE即“保护模式允许”，是80x86控制寄存器CR0中的控制位

x86体系结构

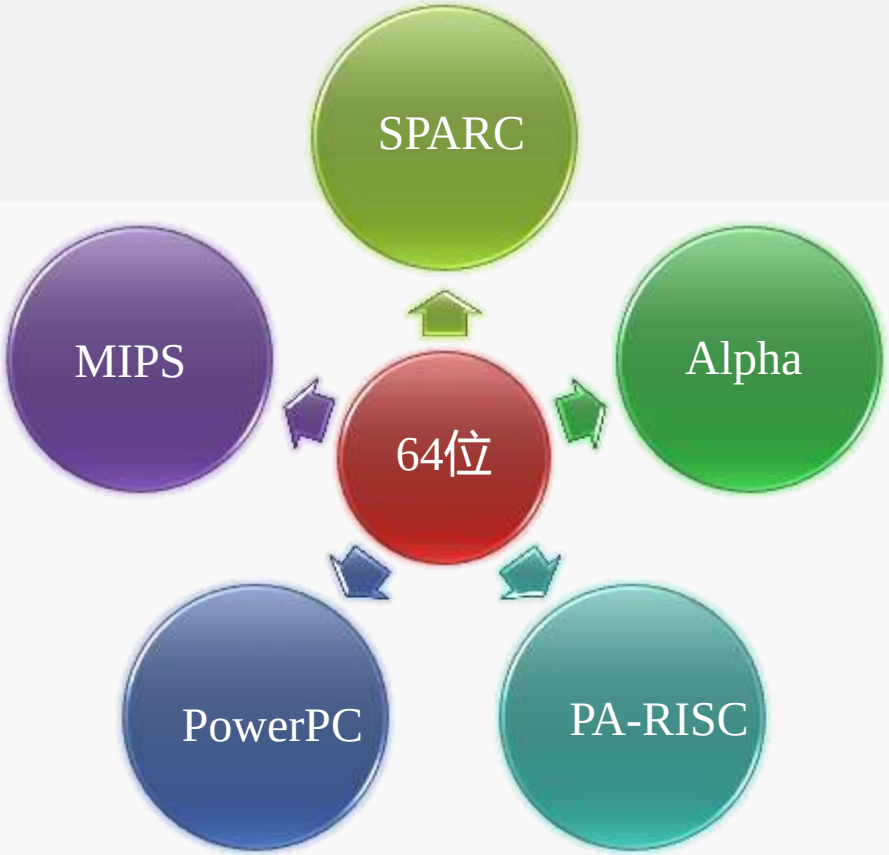
体系结构		厂商	微处理器型号	字长	年代
x86	“x86-16” “IA-16”	Intel	8086, 8088, 80186, 80188 80286	16位	1978年起
	IA-32	Intel	80386, 80486, Pentium, Pentium Pro/II/III/4, Core, Atom	32位	1985年起
		AMD	Am386, Am486, AM5x86, K5, K6, Athlon		
		Others	Cyrix 5x86; VIA C3/C7 Transmeta Crusoe, Efficeon		
	x86-64	AMD	Opteron, Athlon 64 Phenom, Phenom II	64位	2003年起
		Intel	Pentium 4 Prescott, Core 2 Core i3/i5/i7		
		Others	VIA Nano		

注：Intel提出的IA-64是独立于x86的一种新的体系结构，不兼容IA-32

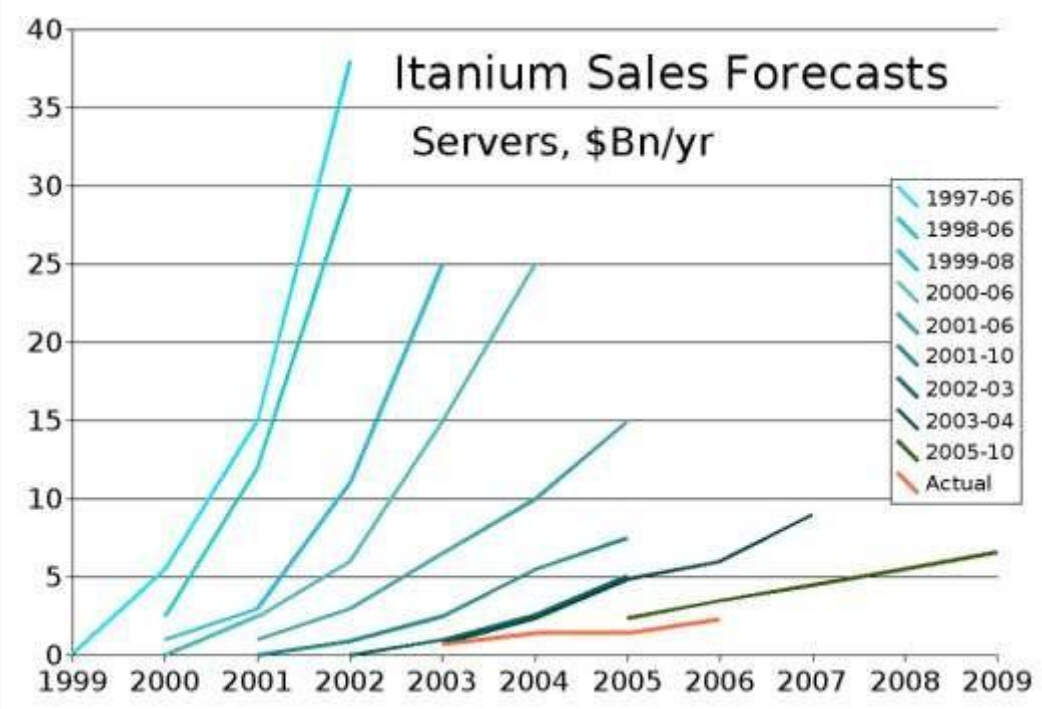
AMD Opteron（2003年）

Opteron的主要特点

- x86扩展到64位的第一款微处理器
- 可以访问高于4GB的存储器
- 兼容32位x86程序，且不降低性能



Opteron
主频1.4~3.5GHz
工艺130~32nm



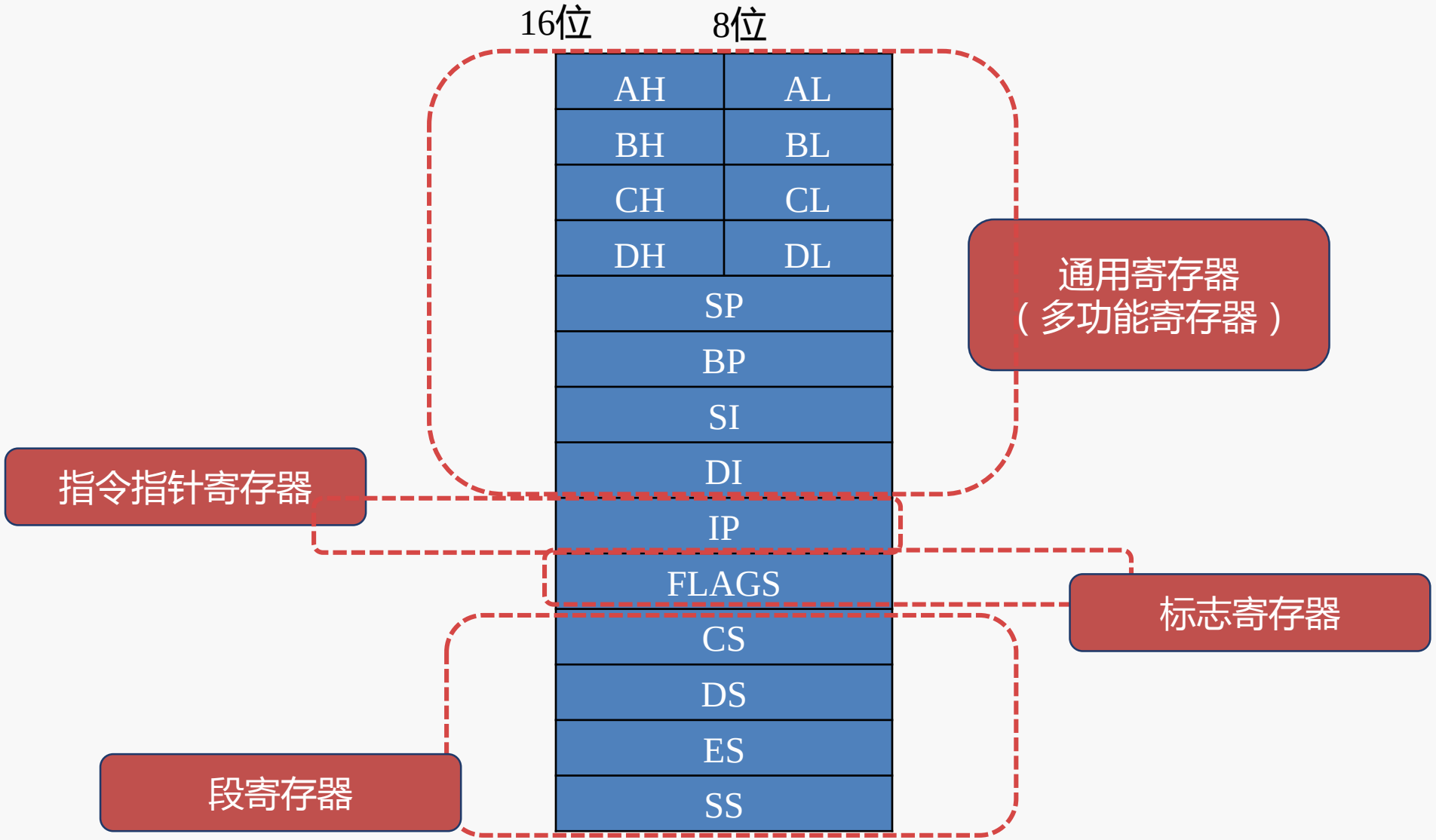
Intel Italium

x86-64的运行模式

运行模式	运行子模式	操作系统	已有程序的支持
长模式 Long mode	64位模式 64-bit mode	64位	需重新编译
	兼容模式 Compatibility mode	64位	不需要重新编译
传统模式 Legacy mode	保护模式 Protected mode	32位或16位	不需要重新编译
	虚拟8086模式 Virtual 8086 mode	32位或16位	不需要重新编译
	实模式 Real mode	16位	不需要重新编译

从16位到64位：x86体系结构的演变

8086的寄存器模型

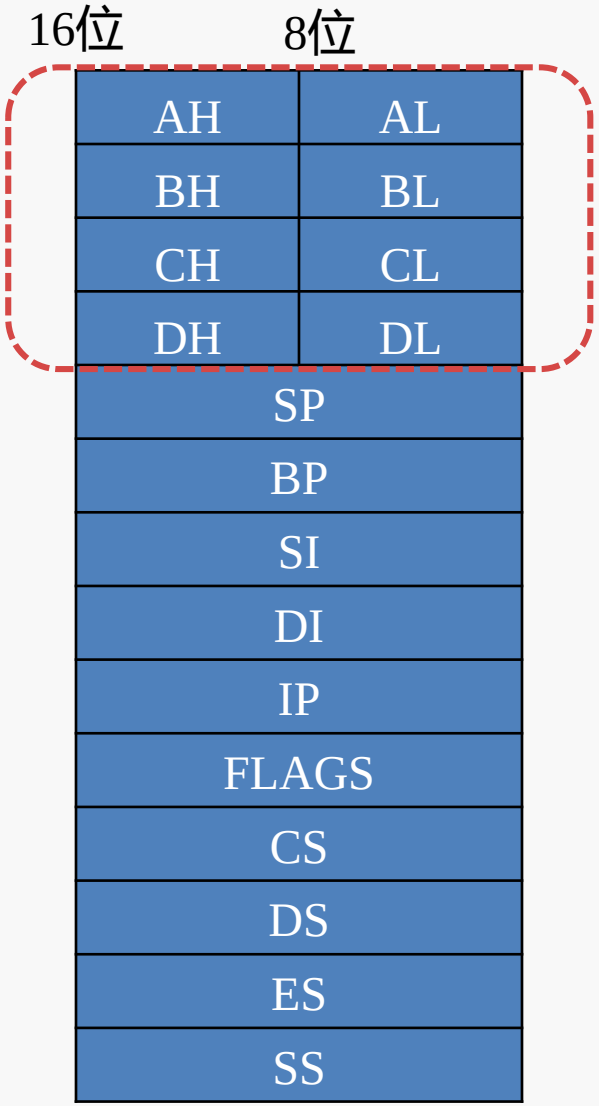

16位
8位
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
IP
FLAGS
CS
DS
ES
SS
通用寄存器
（多功能寄存器）
指令指针寄存器
标志寄存器
段寄存器

通用寄存器（多功能寄存器）（1）

数据寄存器，共有4个

- 均为16位寄存器
- 每个16位寄存器都可分为两个8位寄存器使用
- 适用大多数算术运算和逻辑运算指令
- 除存放通用数据外，各有一些专门的用途：

AX	Accumulator	存放乘除等指令的操作数
BX	Base	存放存储单元的偏移地址
CX	Count	存放计数值
DX	Data	乘法运算产生的部分积 除法运算的部分被除数

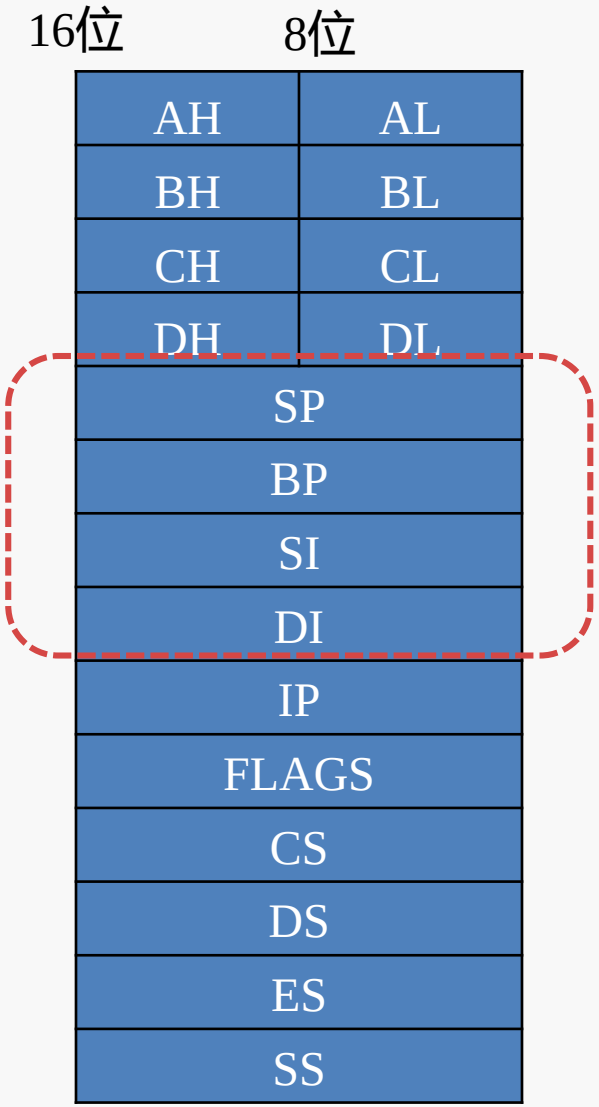


通用寄存器（多功能寄存器）（2）

指针和变址寄存器，共有4个，分为两组

- 均为16位寄存器
- SP和BP用于堆栈操作
- SI和DI用于串操作
- 都可以作为数据寄存器使用

SP	stack pointer	堆栈指针寄存器
BP	(stack)base pointer	(堆栈)基址指针寄存器
SI	source index	源变址寄存器
DI	destination index	目的变址寄存器



标志寄存器

标志位

- FLAGS寄存器中包含若干标志位
- 标志位分为两大类：状态标志和控制标志

状态标志 反映CPU的工作状态

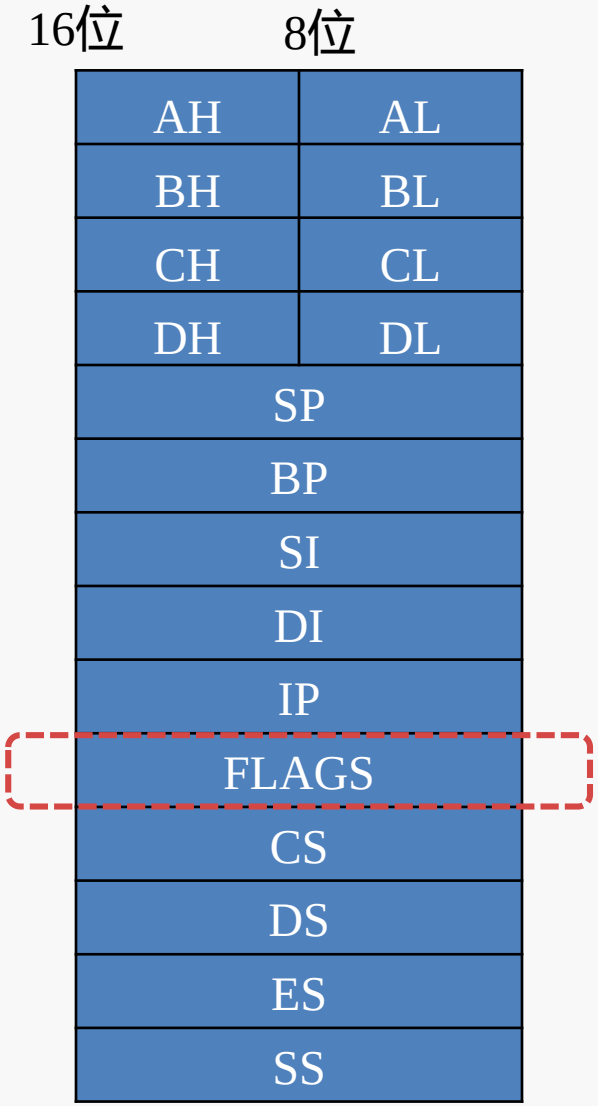
例如：

- 执行加法运算时是否产生进位
- 运算结果是否为零

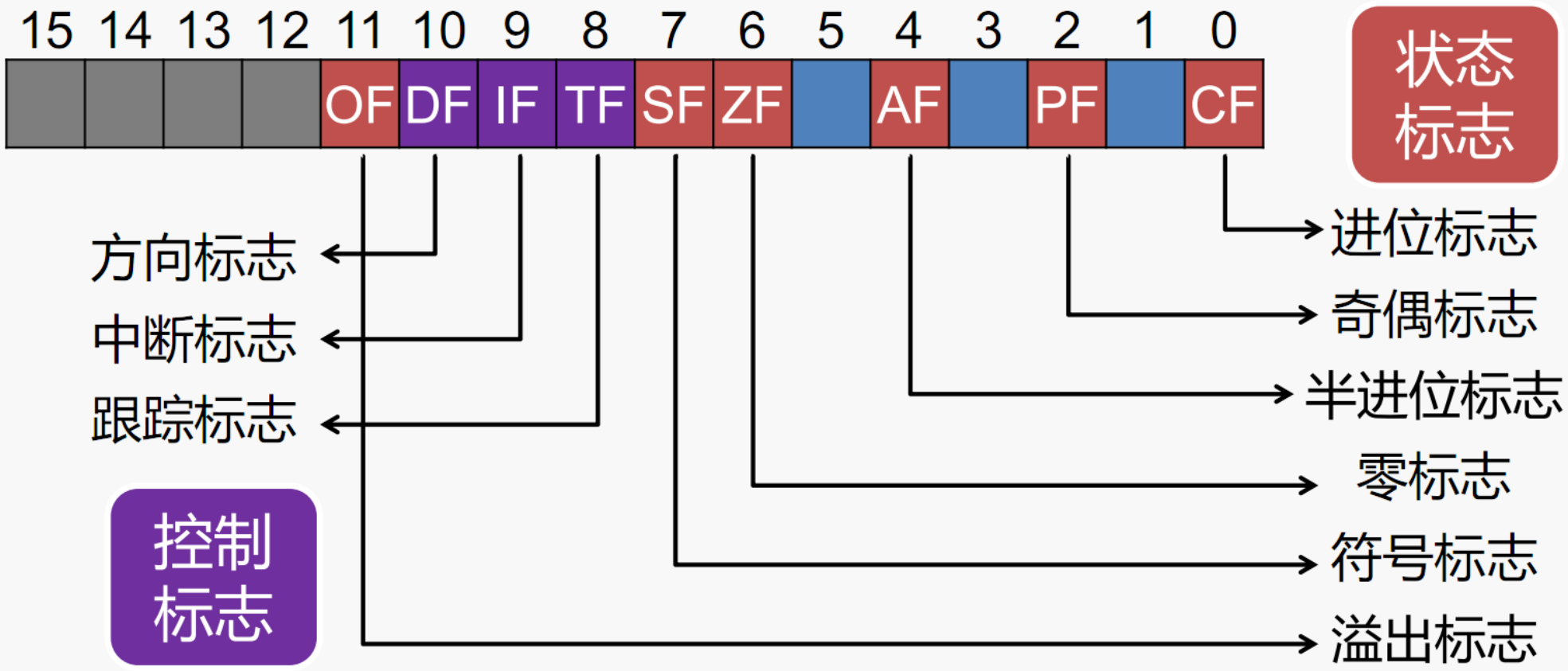
控制标志 对CPU的运行起特定控制作用

例如：

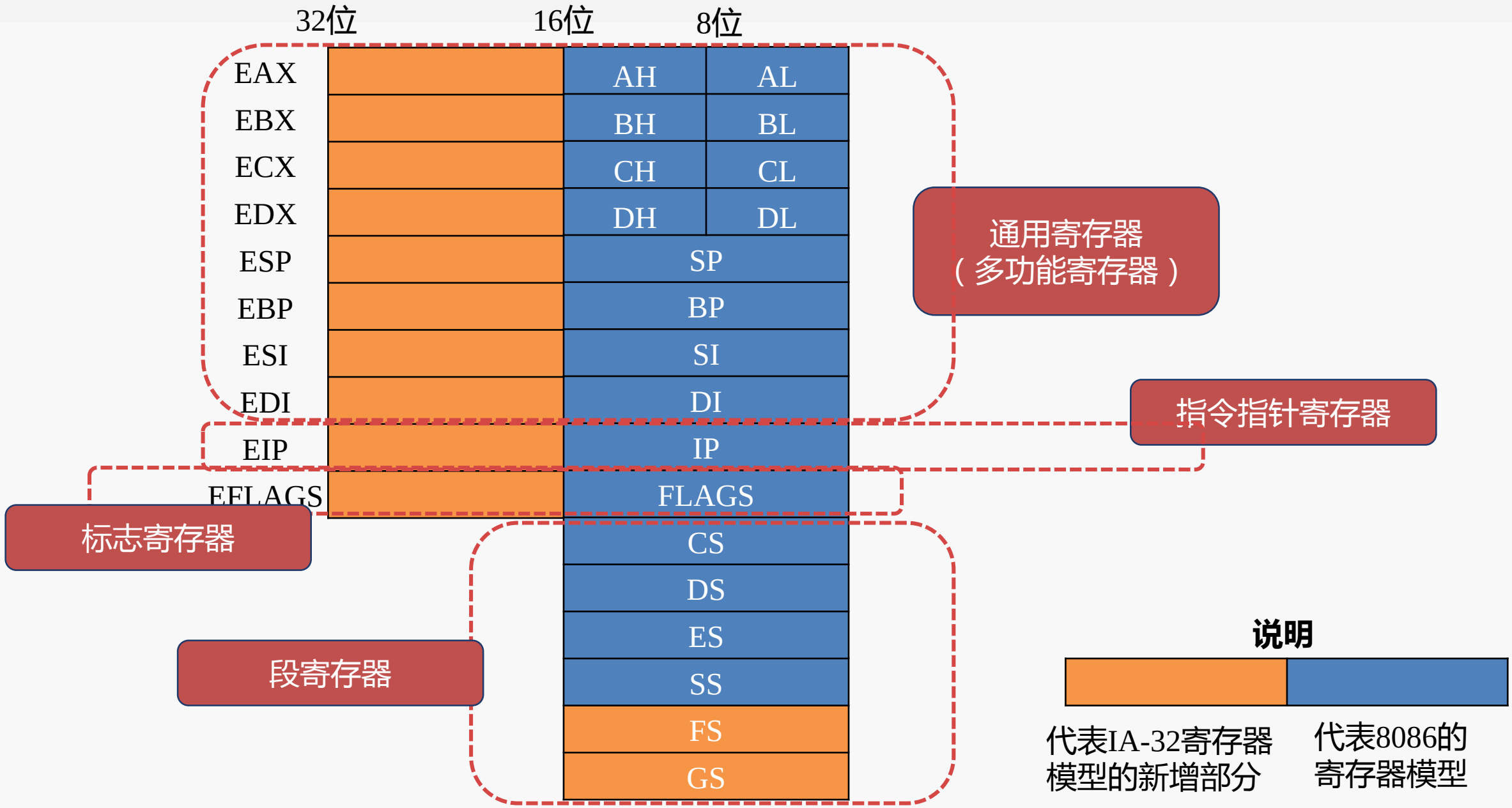
- 以单步方式还是连续方式运行
- 是否允许响应外部中断请求



8086的标志位


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OF DF IF TF SF ZF AF PF CF
状态标志
进位标志
奇偶标志
半进位标志
零标志
符号标志
溢出标志
方向标志
中断标志
跟踪标志
控制标志

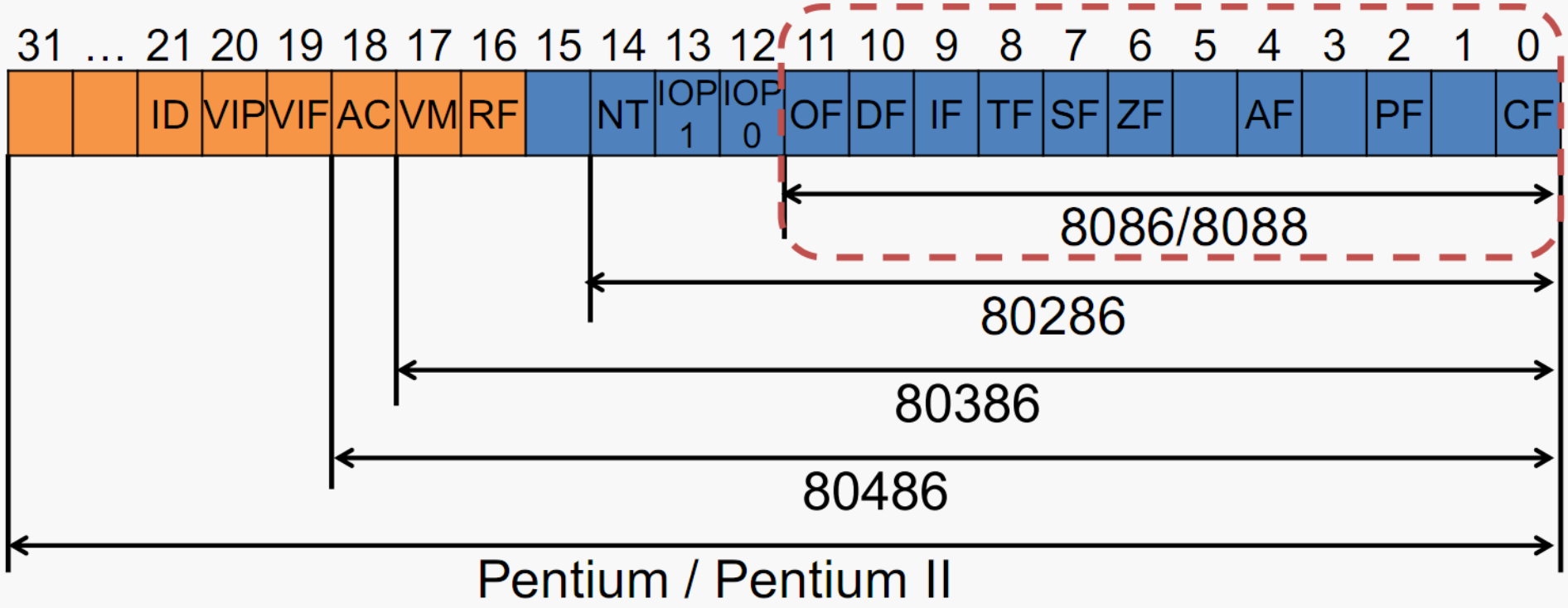
IA-32的寄存器模型


32位
16位
8位
EAX
EBX
ECX
EDX
ESP
EBP
ESI
EDI
EIP
EFLAGS
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
IP
FLAGS
CS
DS
ES
SS
FS
GS
通用寄存器
（多功能寄存器）
指令指针寄存器
标志寄存器
段寄存器
说明
代表IA-32寄存器模型的新增部分
代表8086的寄存器模型

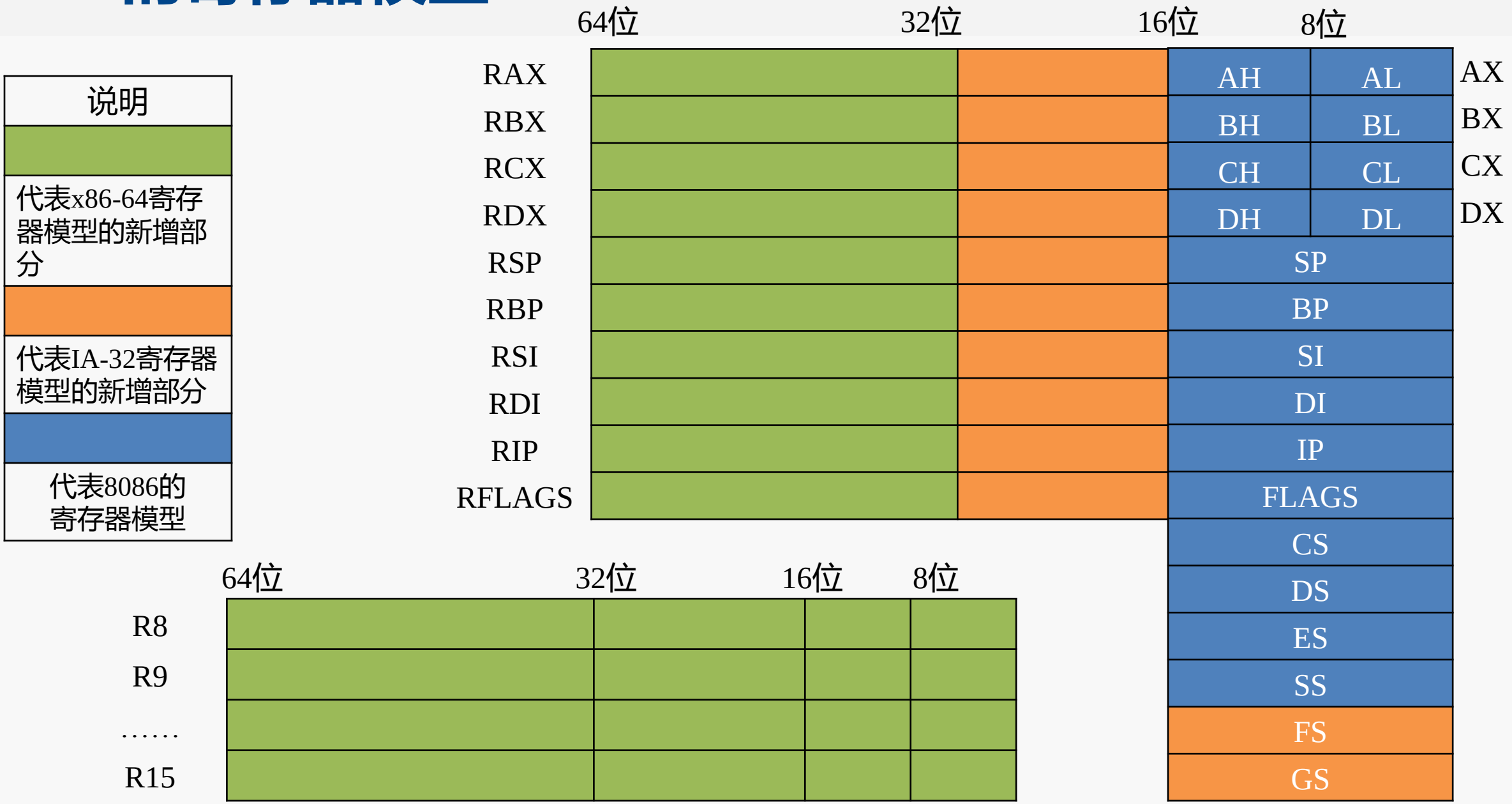
标志寄存器的说明

标志寄存器EFLAGS/FLAGS

- 用于指示微处理器的状态并控制它的操作
- 标志寄存器的内容在不断扩充



IA-32的寄存器模型


说明
代表x86-64寄存器模型的新增部分
代表IA-32寄存器模型的新增部分
代表8086的寄存器模型
64位
32位
16位
8位
RAX
RBX
RCX
RDX
RSP
RBP
RSI
RDI
RIP
RFLAGS
AH
AL
AX
BH
BL
BX
CH
CL
CX
DH
DL
DX
SP
BP
SI
DI
IP
FLAGS
CS
DS
ES
SS
FS
GS
R8
R9
……
R15

从16位到64位：x86体系结构的演变

寄存器模型

存储器寻址

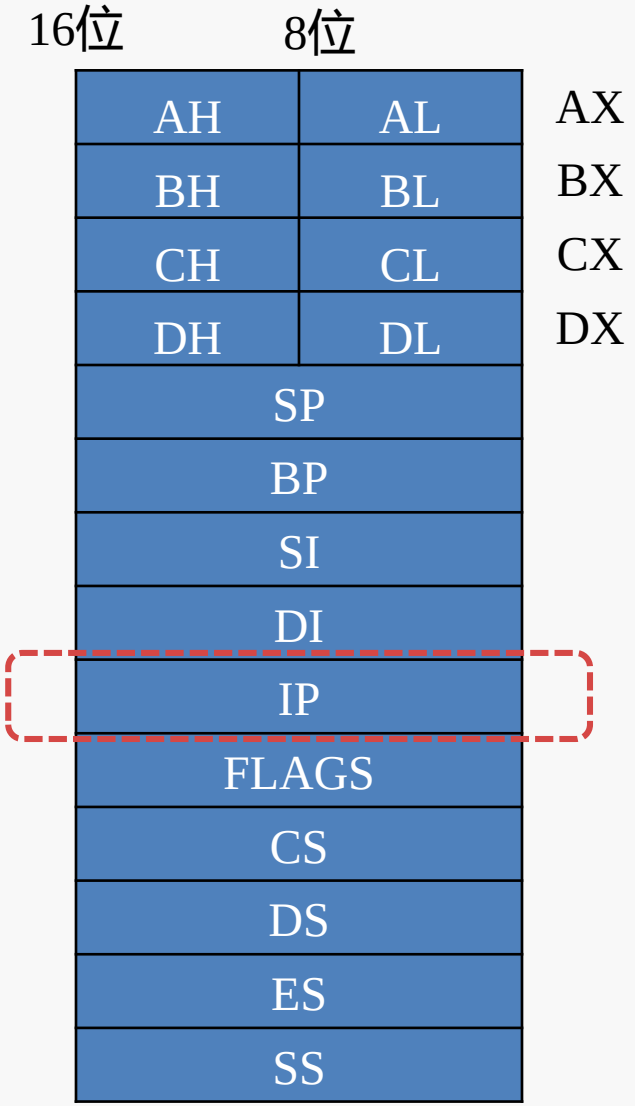
8086的指令指针寄存器

指令指针寄存器 IP（Instruction Pointer）

- 保存一个内存地址，指向当前需要取出的指令
- 当CPU从内存中取出一个指令后，IP会自动增加，指向下一指令的地址（注：实际情况会复杂的多）
- 程序员不能直接对IP进行存取操作
- 转移指令、过程调用/返回指令等会改变IP的内容

IP寄存器的寻址能力：
2^{16}=65536(64K)字节单元

8086对外有20位地址线
寻址范围：2^{20}=1M字节单元

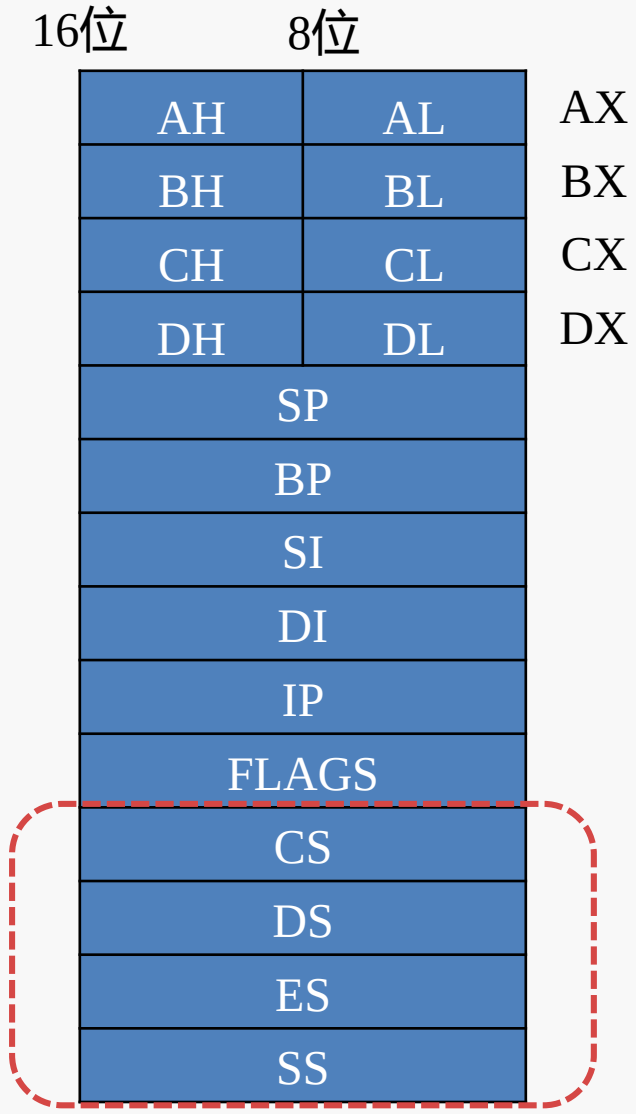


8086的段寄存器

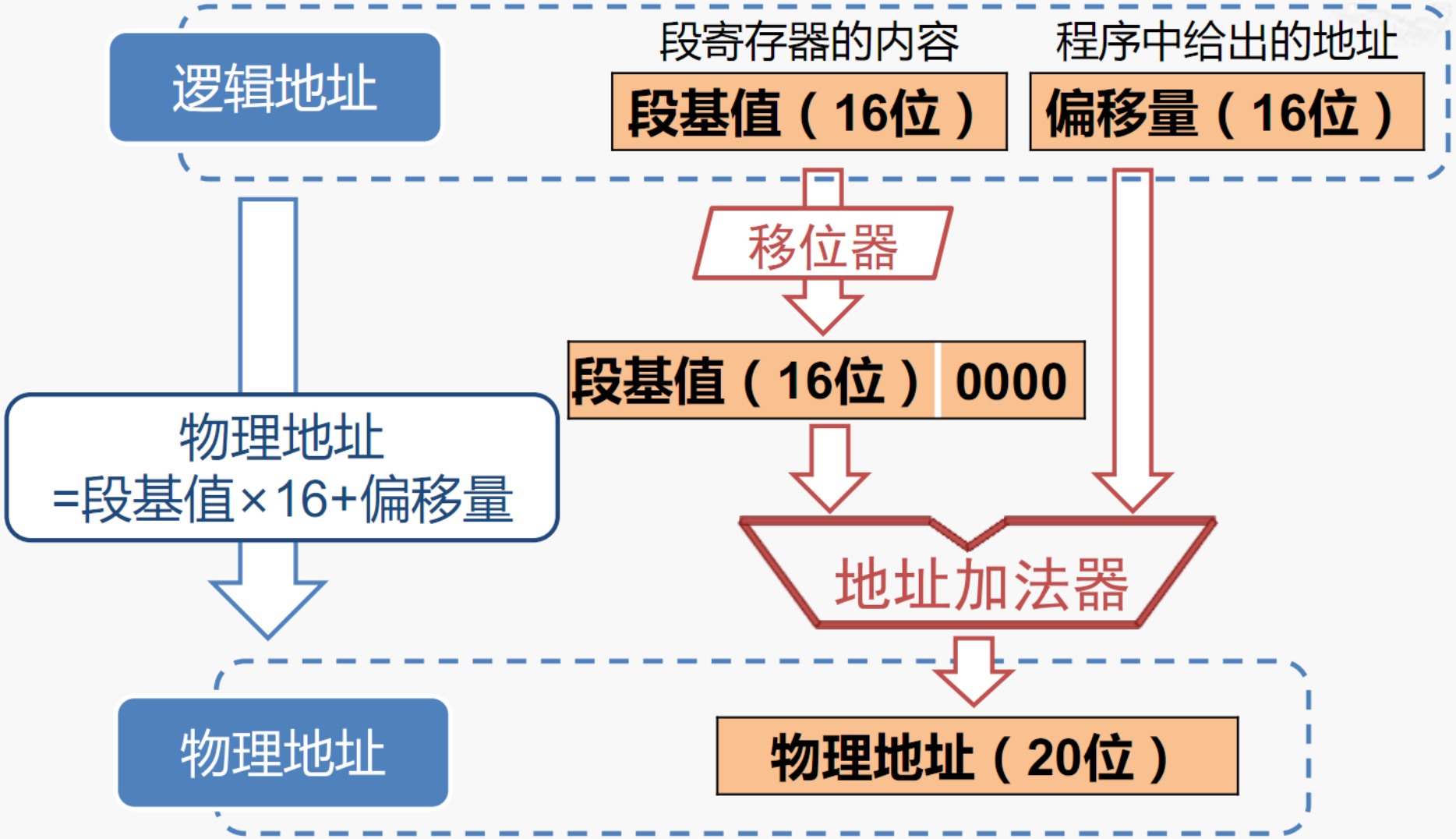
段寄存器（Segment Register）

- 与其它寄存器联合生成存储器地址

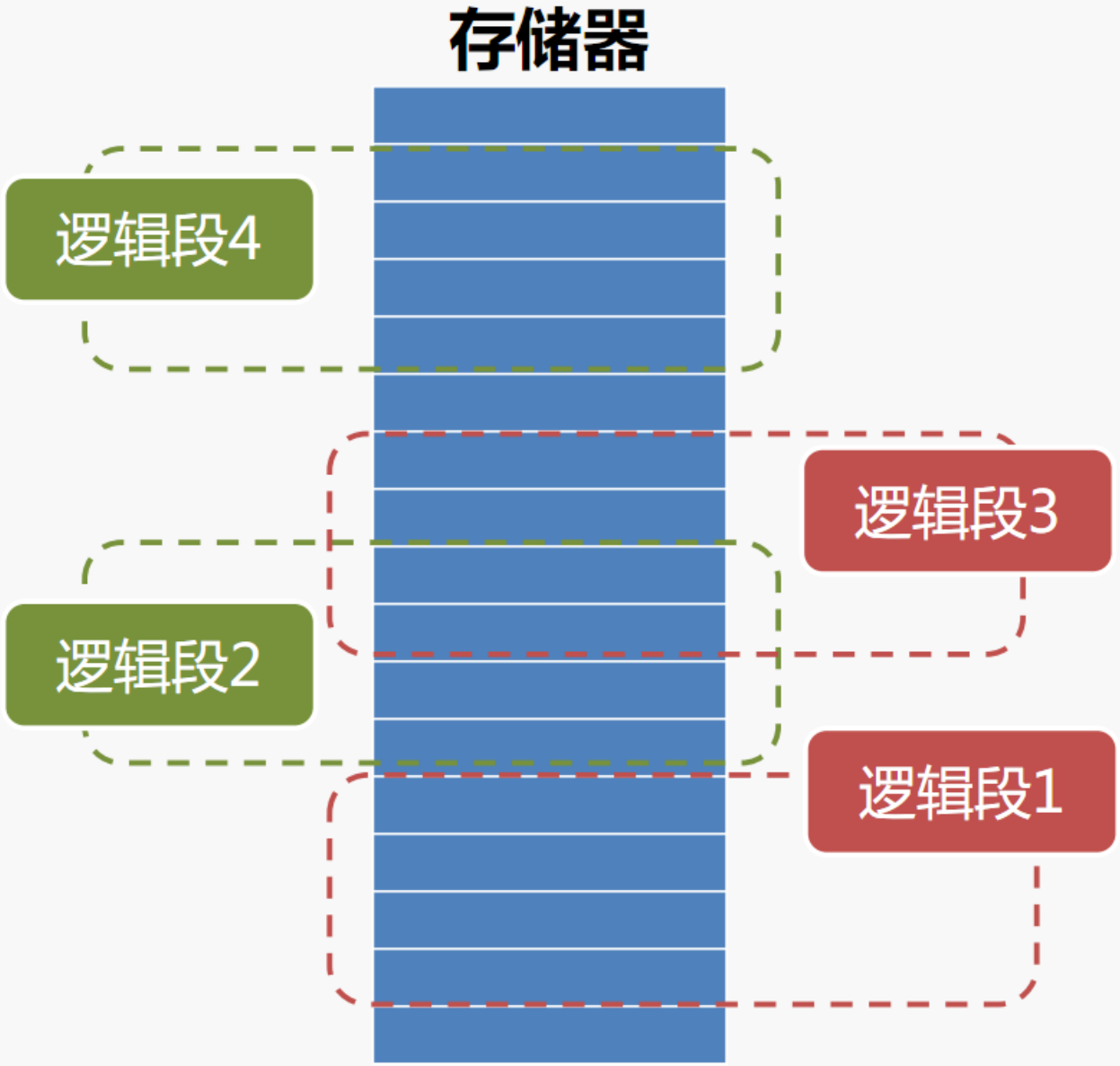
CS	代码段寄存器（Code Segment）
DS	数据段寄存器（Data Segment）
ES	附加段寄存器（Extra Segment）
SS	堆栈段寄存器（Stack Segment）



8086的物理地址生成



逻辑段在物理存储器中的位置

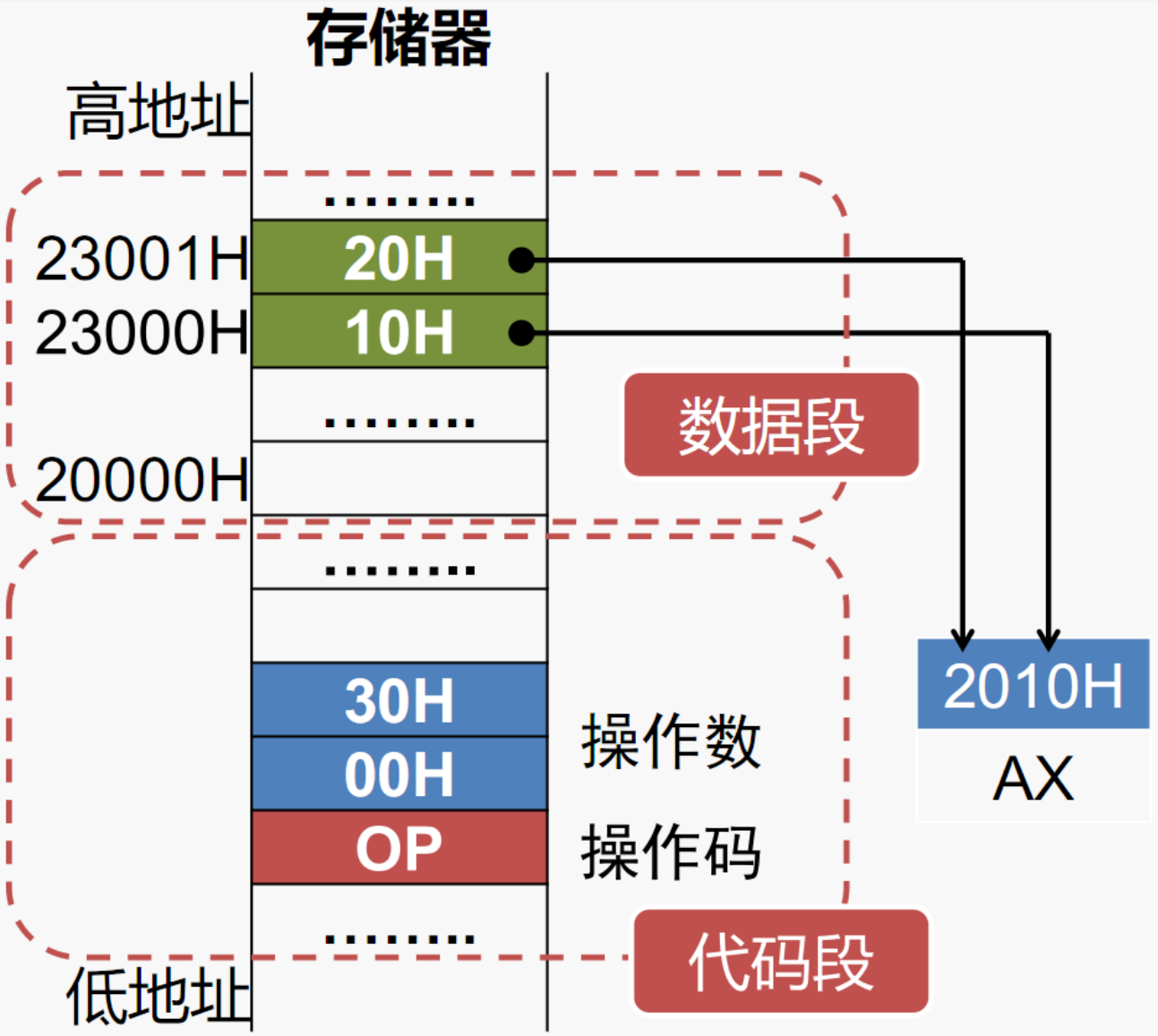


1M字节的存储空间分成许多逻辑段，每段最长64K字节，可以用16位地址进行寻址

编程时使用逻辑地址，不需要知道代码或数据在存储器中的具体物理位置，从而简化存储资源的管理

各个逻辑段在实际存储空间中可以完全分开，也可以部分重叠，甚至完全重叠

“段加偏移”的编程实例



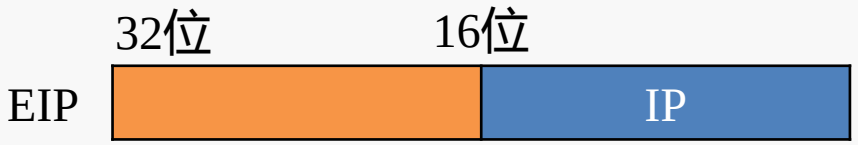
汇编指令 MOV AX, [3000H]

操作数默认存放在DS指向的数据段中，即
[3000H]=DS:[3000H]

设：DS=2000H，
则：物理地址
=2000H×16+3000H
=23000H

IA-32的存储器寻址

以指令的寻址为例



实模式 CS:IP

保护模式 CS:EIP

EIP寄存器的寻址能力：

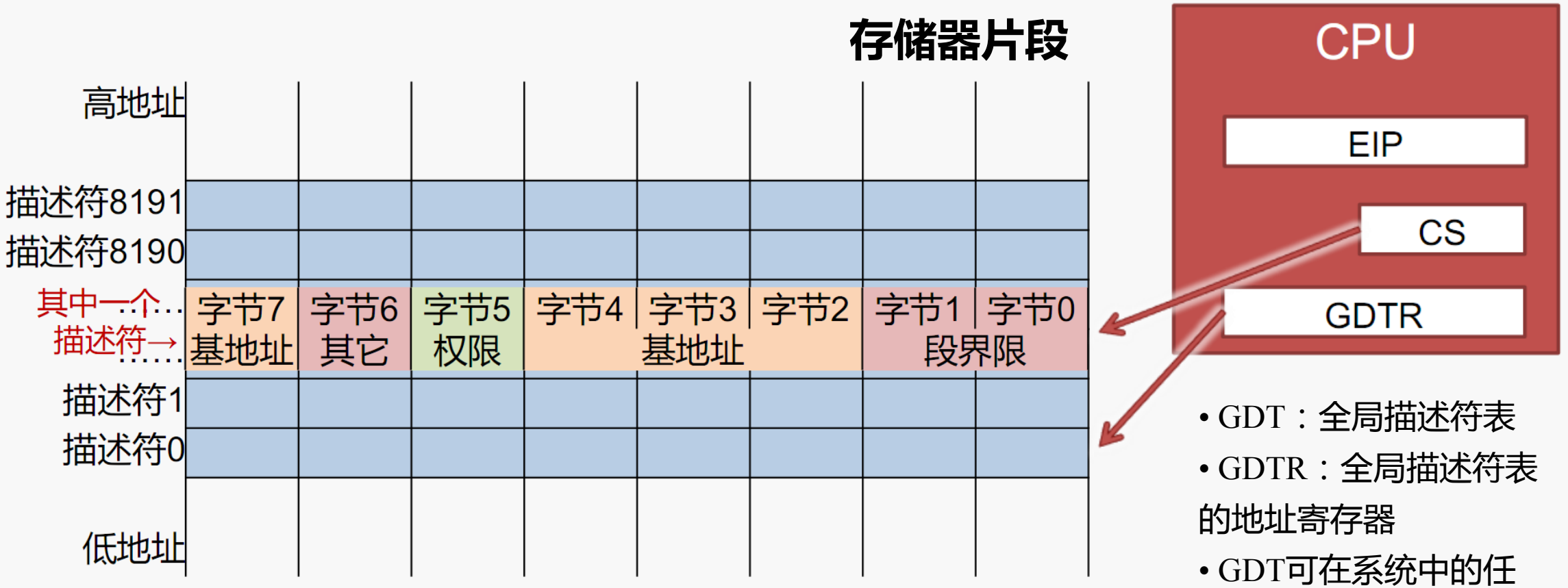
2^{32}=4G字节单元

80386对外有32位地址线

寻址范围：2^{32}=4G字节单元

IA-32的存储器寻址

保护模式下，段基址不在CS中，而是在内存中



- GDT：全局描述符表
- GDTR：全局描述符表的地址寄存器
- GDT可在系统中的任何存储单元，通过GDTR定位

X86-64的描述符

注：描述符中没有了段基址和段界限，只有访问权限字节和若干控制位。所有的代码段都从地址0开始。

第二章 指令系统体系结构

- 1.设计自己的计算机
- 2. x86体系结构
- 3. x86指令简介
- 4.复杂的x86指令举例
- 5. MIPS体系结构
- 6. MIPS指令简介

指令的主要类别

运算类指令

例如：加、减、乘、除，与、或、非等

传送类指令

例如：从存储器到通用寄存器，从通用寄存器到I/O接口等

控制类指令

例如：暂停处理器、清除标志位等

转移类指令

例如：无条件转移、条件转移、过程调用等

指令的运行结果

改变通用寄存器的内容
*如ADD AX, DX

改变存储器单元的内容
*如 MOV [10H], CX

其它
……

改变标志位
*如产生进位

改变指令指针
*如 JMP [BX]

改变外设端口的内容
*如访问显示端口

指令分类举例

1. 传送指令
2. 算术运算指令
3. 逻辑运算和移位指令
4. 转移指令
5. 处理器控制指令

传送指令

作用：把数据或地址传送到寄存器或存储器单元中

分组	助记符	功能	操作数类型
通用数据传送指令	MOV	传送	字节/字
	PUSH	压栈	字
	POP	弹栈	字
	XCHG	交换	字节/字
累加器专用传送指令	XLAT	换码	字节
	IN	输入	字节/字
	OUT	输出	字节/字
地址传送指令	LEA	装入有效地址	字
	LDS	把指针装入寄存器和DS	4个字节
	LES	把指针装入寄存器和ES	4个字节
标志传送指令	LAHF	把标志装入AH	字节
	SAHF	把AH送标志寄存器	字节
	PUSHF	标志压栈	字
	POPF	标志弹栈	字

（1）传送指令

MOV指令（传送）

格式：MOV DST, SRC

操作：DST←SRC

说明：

- DST表示目的操作数，SRC表示源操作数
- MOV指令把一个操作数从源传送至目的，源操作数保持不变

MOV指令和寻址方式的示例

MOV EBX, 40

MOV AL, BL

MOV ECX, [1000H]

MOV [DI], AX

MOV WORD PTR[BX+SI*2+200H], 01H

直接给出操作数

给出存放操作数的寄存器名称

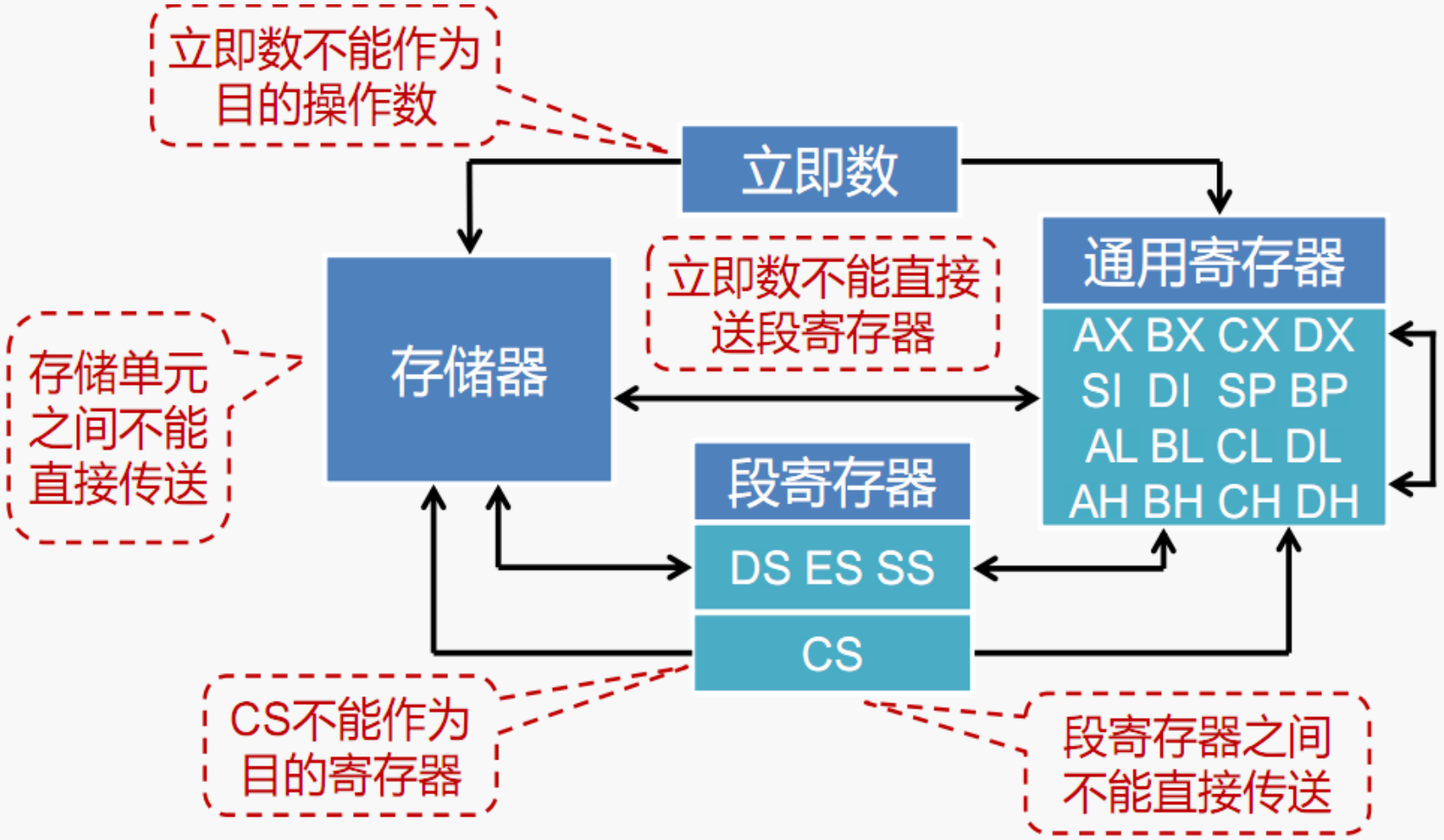
给出存放操作数的存储器地址

给出存放“存放操作数的存储器地址”的寄存器名称

给出“存放操作数的存储器地址”的计算方法

注：BYTE PTR ：字节长度标记
WORD PTR ：字长度标记
DWORD PTR ：双字长度标记

MOV指令的传送方向和限制



不同类型的MOV指令编码

7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
1 1 0 0 0 1 1 w	mod 000 r /m	DISP-LO	DISP_HI	data	data if w=1
1 0 0 0 1 0 d w	mod reg r /m	DISP-LO	DISP_HI		
1 0 0 0 1 1 1 0	mod 0 SR r/m	DISP-LO	DISP_HI		
1 0 0 0 1 1 0 0	mod 0 SR r/m	DISP-LO	DISP_HI		
1 0 1 0 0 0 0 w	addr-lo	addr-hi			
1 0 1 0 0 0 1 w	addr-lo	addr-hi			
1 0 1 1 w r e g	data	data if w=1			

MOV指令编码示例

76543210	76543210	76543210	76543210	76543210	76543210
1011 w reg	data	data if w=1			
1011 1 000	11101110	00010000			
B8	EE	10			

立即数到寄存器

MOV AX,10EEH

76543210	76543210	76543210	76543210	76543210	76543210
100010dw	mod reg r/m	DISP-LO	DISP_HI		
10001011	000 011 11	00000100	00010000		
8B	0F	04	10		

存储器到寄存器

MOV CX,[BX+1004H]

（2）栈操作指令

PUSH指令（压栈）

格式：PUSH SRC

说明：

- SRC表示寄存器操作数或存储器操作数

POP指令（弹栈）

格式：POP DST

说明：

- DST表示寄存器操作数或存储器操作数
- DST也可以是除CS寄存器以外的段寄存器

PUSH和POP指令操作示意

(a) 空堆栈

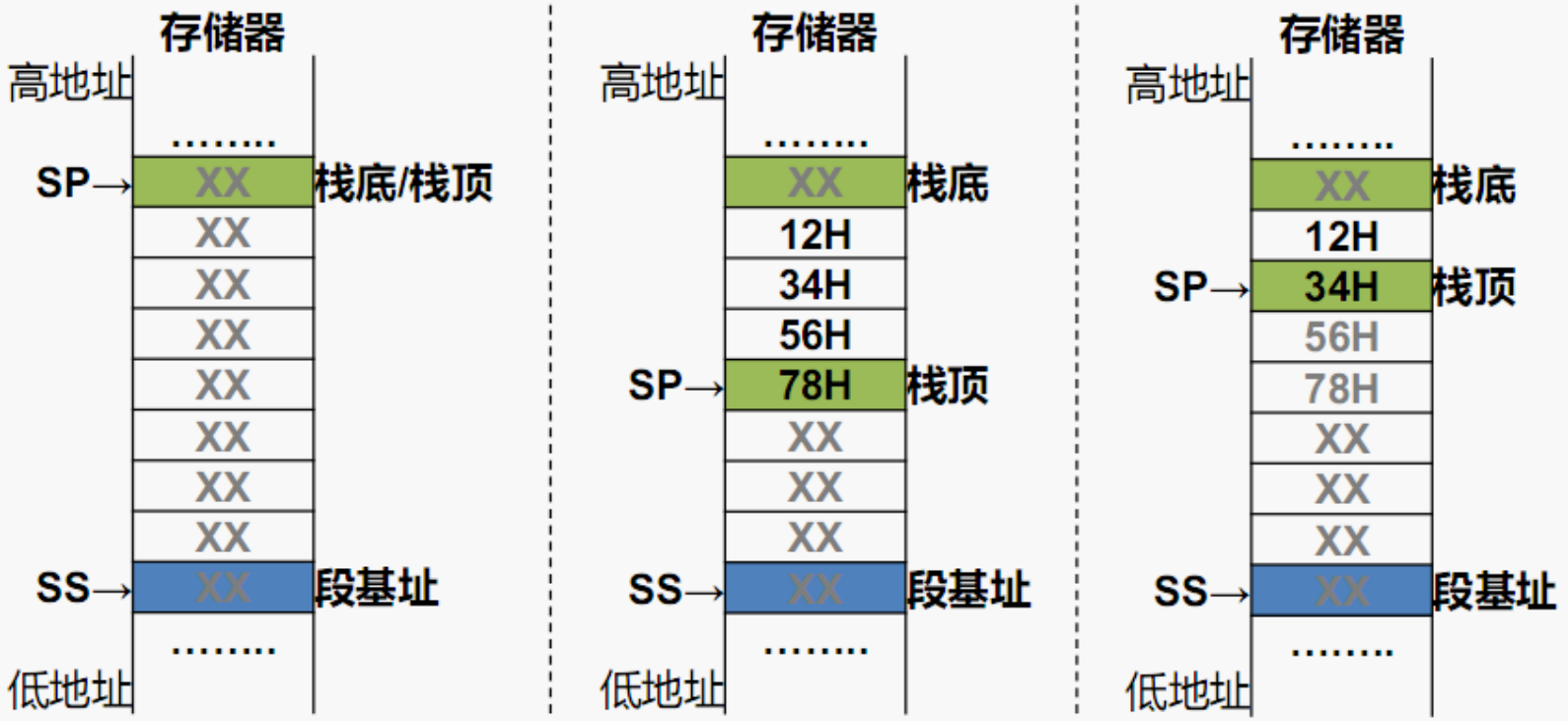
(b) 压栈

PUSH AX; AX=1234H

PUSH BX; BX=5678H

(c) 弹栈

POP CX



注：堆栈结构是所谓“向下生长”，即栈底在堆栈的高地址端。而堆栈段的段基址（由SS寄存器确定）并不是栈底。

（3）LEA指令说明

LEA指令（Load Effective Address）

格式：LEA REG，SRC

操作：

- 把源操作数（SRC）的有效地址（即偏移地址）装入指定寄存器（REG）

说明：

- 源操作数必须是存储器操作数
- 目的操作数必须是通用寄存器

LEA指令示例

LEA BX，[BX+DI*4+6H]

注意区别于：

MOV BX,[BX+DI*4+6H]

思考：

LEA指令的巧妙用途

指令分类举例

1. 传送指令
2. 算术运算指令
3. 逻辑运算和移位指令
4. 转移指令
5. 处理器控制指令

算术运算指令

作用

- 完成加、减、乘、除等算术运算
- 提供运算结果调整、符号扩展等功能

操作数的限制

- 目的操作数不能是立即数或CS寄存器
- 两个操作数不能同时为存储器操作数

分组	助记符	功能
加法	ADD	加
	ADC	加（带进位）
	INC	加1
减法	SUB	减
	SBB	减（带借位）
	DEC	减1
	NEG	取补
	CMP	比较
乘法	MUL	乘（不带符号）
	IMUL	乘（带符号）
除法	DIV	除（不带符号）
	IDIV	除（带符号）

算术运算指令

作用

- 完成加、减、乘、除等算术运算
- 提供运算结果调整、符号扩展等功能

操作数的限制

- 目的操作数不能是立即数或CS寄存器
- 两个操作数不能同时为存储器操作数

分组	助记符	功能
符号扩展	CBW	将字节扩展为字
	CWD	将字扩展为双字
十进制调整	AAA	加法的ASCII调整
	DAA	加法的十进制调整
	AAS	减法的ASCII调整
	DAS	减法的十进制调整
	AAM	乘法的ASCII调整
	AAD	除法的ASCII调整

（1）加法类指令

ADD指令（加）

格式：ADD DST, SRC

操作：DST←DST+SRC

ADC指令（带进位的加）

格式：ADC DST, SRC

操作：DST←DST+SRC+CF

INC指令（加1）

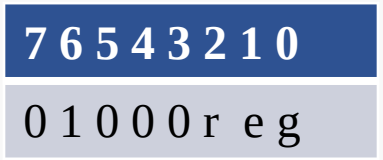
格式：INC OPR

操作：OPR←OPR+1LEA指令（Load Effective Address）

```
ADD BL, 8
ADD WORD PTR[BX], DX


ADD EAX, ECX
ADC EBX, EDX


INC CL
```

示例

INC Reg
指令长度为1字节



（2）减法类指令

SUB指令（减）

格式：SUB DST, SRC

操作：DST←DST-SRC

SBB指令（带借位的减）

格式：SBB DST, SRC

操作：DST←DST-SRC-CF

DEC指令（减1）

格式：DEC OPR

操作：OPR←OPR-1

示例

```
SUB BL, 8
SUB WORD PTR[BX], DX


SUB EAX, ECX
SBB EBX, EDX


DEC ECX
```

（2）减法类指令

CMP指令（比较）

格式：CMP DST, SRC

操作：DST - SRC

说明：减法操作，但不回写结果，仅影响标志位

指令分类举例

1. 传送指令
2. 算术运算指令
3. 逻辑运算和移位指令
4. 转移指令
5. 处理器控制指令

逻辑运算和移位指令

作用

- 实现对二进制位的操作和控制
- 又称“位操作指令”

操作数的限制

- 对于单操作数指令，操作数不能是立即数
- 对于双操作数指令，限制与MOV指令相同

分组	助记符	功能
逻辑运算	NOT	逻辑非
	AND	逻辑与
	OR	逻辑或
	XOR	逻辑异或
	TEST	逻辑测试
移位	SHL	逻辑左移
	SAL	算术左移
	SHR	逻辑右移
	SAR	算术右移
循环移位	ROL	循环左移
	ROR	循环右移
	RCL	带进位循环左移
	RCR	带进位循环右移

（1）逻辑运算指令

NOT指令（逻辑非）

格式：NOT OPR

操作：OPR按位求反，送回OPR

AND指令（逻辑与）

格式：AND DST, SRC

操作：将DST和SRC的内容按位进行“与”操作，结果送到DST中

```
MOV  AL, 10101010B
NOT  AL
;now: AL=01010101B
```

示例

```
MOV  BL, 11111010B
AND  BL, 0FH
;now: BL=00001010B
```

示例

（2）移位指令

SHL指令（左移）

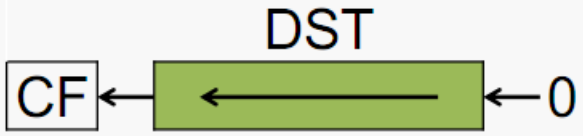
格式：SHL DST, CNT

操作：

- 将DST的内容按图中方式移动
- 移动位数由CNT指定

说明：

- DST可以是寄存器或存储器操作数
- CNT可以是立即数1或CL寄存器
- 相当于无符号数乘以2n的运算



示例：设AL中有一个无符号数X，用移位指令求10X

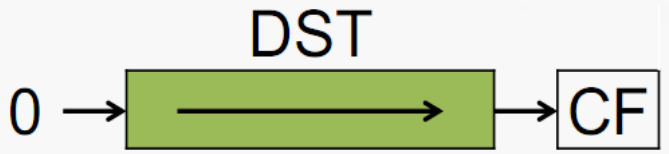
MOV AH, 0
SHL AX, 1 ；得到2X
MOV BX, AX
MOV CL, 2
SHL AX, CL ；得到8X
ADD AX, BX ；得到10X

（2）移位指令

SHR指令（逻辑右移）

格式：SHR DST, CNT

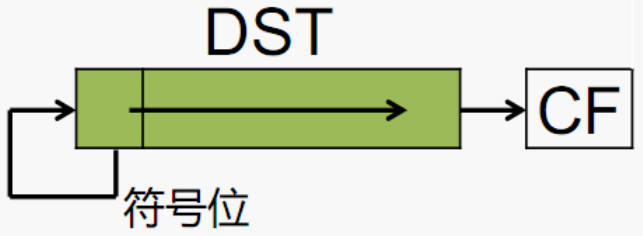
说明：相当于无符号数除以2n的运算



SAR指令（算术右移）

格式：SAR DST, CNT

说明：相当于带符号数除以2n的运算



指令分类举例

1. 传送指令
2. 算术运算指令
3. 逻辑运算和移位指令
4. 转移指令
5. 处理器控制指令

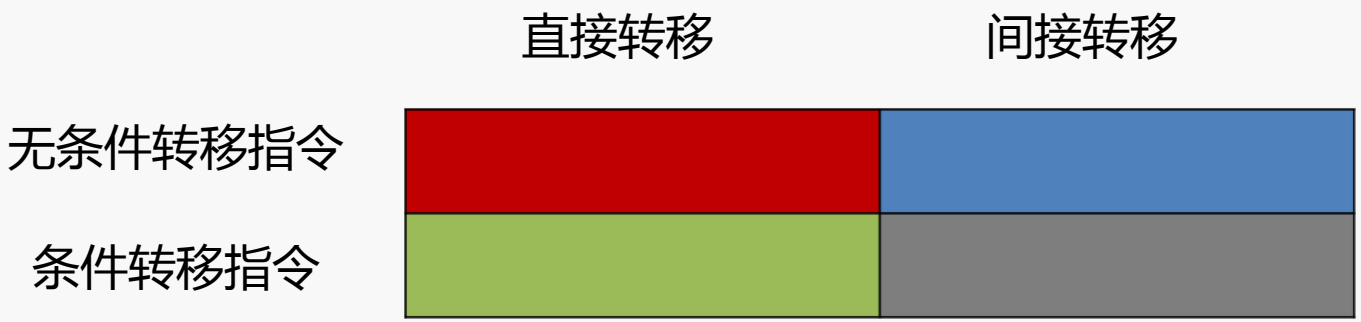
转移指令

作用

- 改变指令执行顺序

说明

- 根据是否有判断条件，分为无条件转移指令和条件转移指令两大类
- 根据转移目标地址的提供方式，可分为直接转移和间接转移两种方式



（1）无条件转移指令 - 直接转移

短转移：JMP SHORT LABEL

- 操作：IP←IP+8位的位移量（-128~127Byte）

近转移：JMP NEAR PTR LABEL

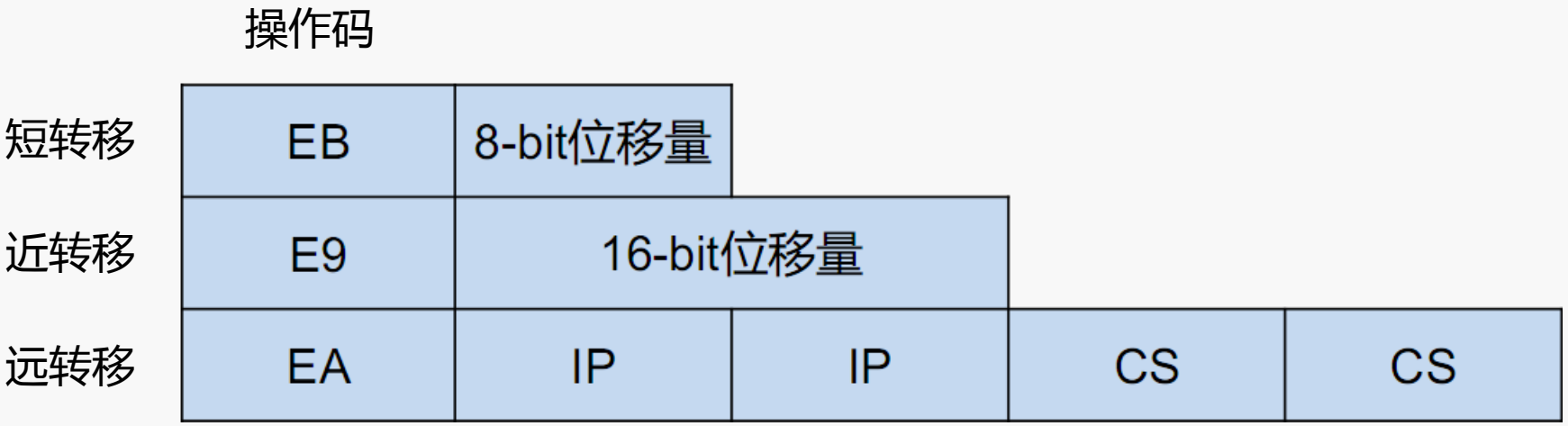
- 操作：IP←IP+16位的位移量（±32KByte）

远转移：JMP FAR PTR LABEL

- 操作：IP←LABEL的偏移地址；CS ←LABEL的段基值

1. 位移量是一个带符号数，为LABEL的偏移地址与当前EIP/IP值之差
2. 从80386开始，近转移可以使用32位的位移量

说明

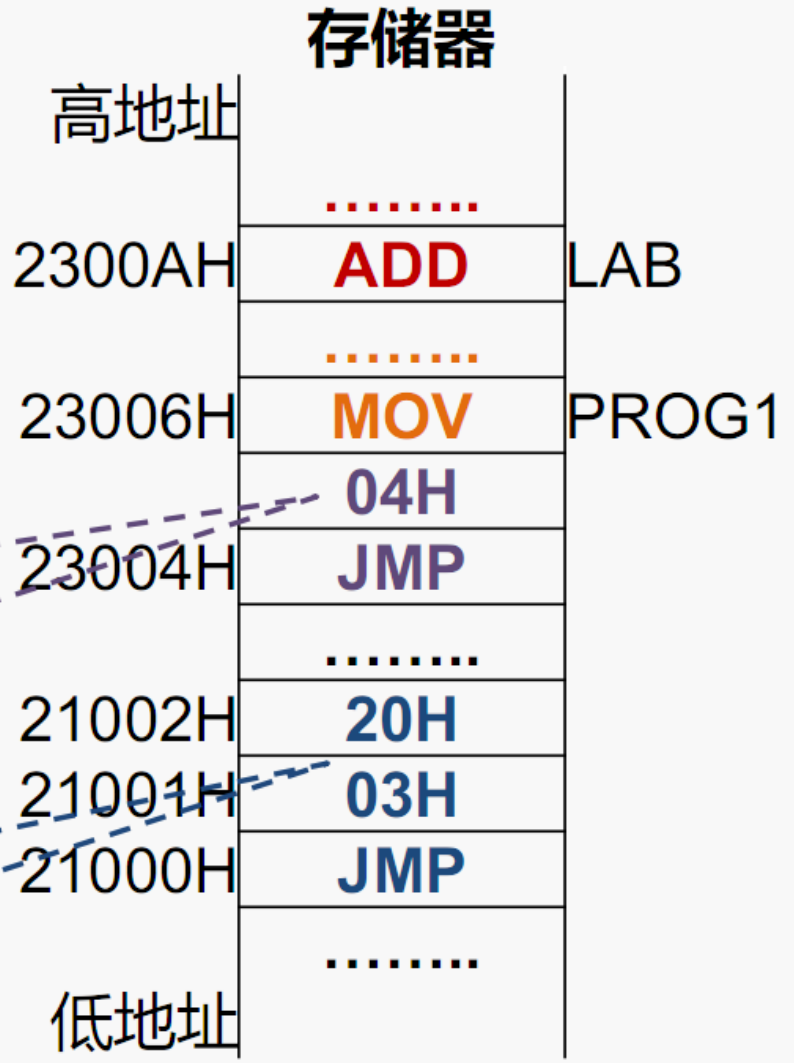


短/近转移的执行过程

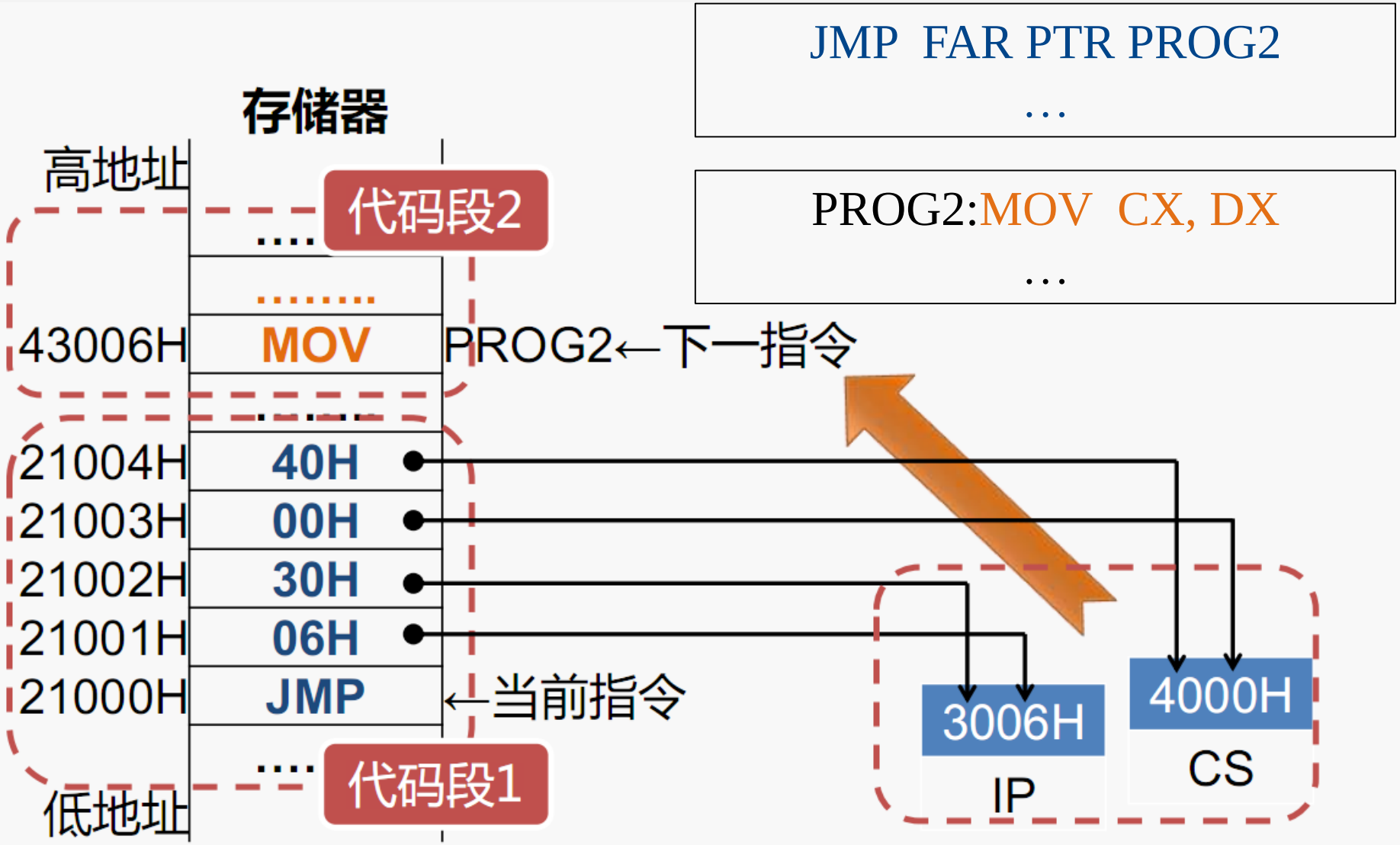
```
        JMP   NEAR PTR PROG1
                ...
        JMP   SHORT LAB
PROG1:  MOV   CX, DX
LAB:    ADD   AX, BX
                ...
```

LAB-IP
=2300AH-(23004H+2)
=04H

PROG1-IP
=23006H-(21000H+3)
=2003H



远转移的执行过程


JMP FAR PTR PROG2
…
PROG2:MOV CX, DX
…
存储器
高地址
代码段2
43006H
MOV
PROG2←下一指令
21004H
40H
21003H
00H
21002H
30H
21001H
06H
21000H
JMP
←当前指令
代码段1
低地址
3006H
IP
4000H
CS

（2）无条件转移指令 - 间接转移

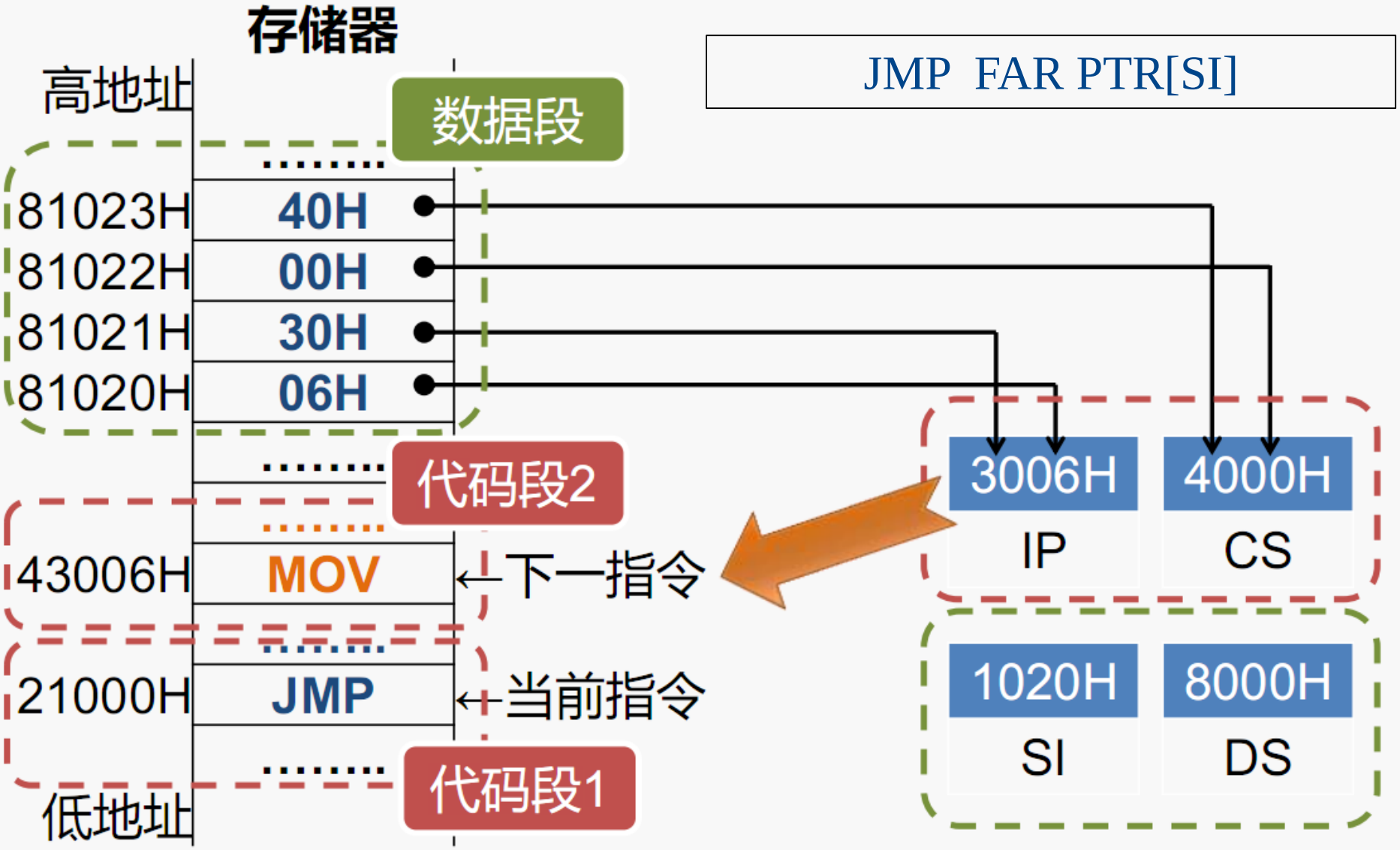
转移目标地址在寄存器中

- JMP AX　　; AX → IP
- JMP EAX　; EAX → EIP

转移目标地址在存储器中

- JMP [SI]　　; [SI] → IP
- JMP FAR PTR[SI]　; [SI] → IP, [SI+2] → CS

间接转移示例


存储器
高地址
数据段
81023H
40H
81022H
00H
81021H
30H
81020H
06H
代码段2
43006H
MOV
下一指令
21000H
JMP
当前指令
代码段1
低地址
JMP FAR PTR[SI]
3006H
IP
4000H
CS
1020H
SI
8000H
DS

（3）条件转移指令

操作

- 根据当前的状态标志位决定是否发生转移

说明

- 一般在影响标志位的算术或逻辑运算指令之后
- 8086中，所有的条件转移都是短转移
  - 同一代码段内，-128~127字节范围内
- 从80386起，条件转移指令可以使用32位的长位移量
  - 同一代码段内，±2G字节范围

条件转移指令

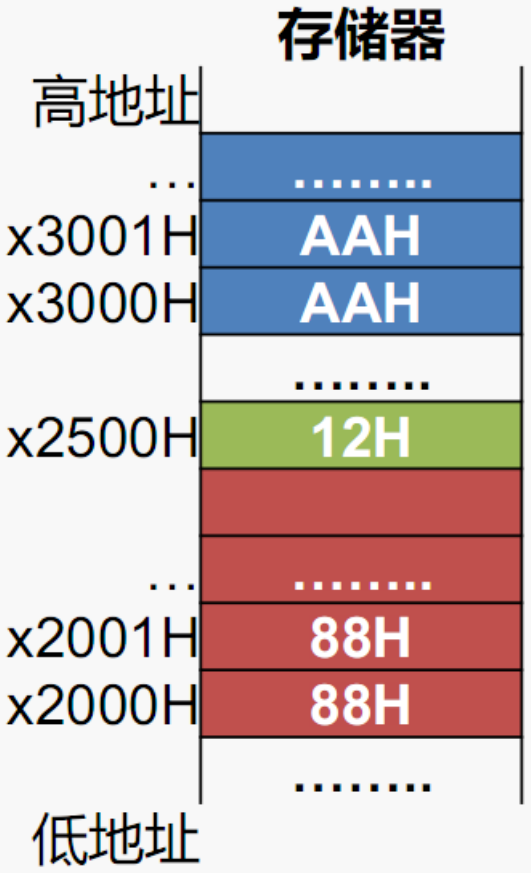
分组		格式	功能	测试条件
条件转移指令	根据某一状态标志转移	JC LABEL	有进位时转移	CF=1
		JNC LABEL	无进位时转移	CF=0
		JP/JPE LABEL	奇偶位为1时转移	PF=1
		JNP/JPO LABEL	奇偶位为0时转移	PF=0
		JZ/JE LABEL	为零/相等时转移	ZF=1
		JNZ/JNE LABEL	不为零/不相等时转移	ZF=0
		JS LABEL	负数时转移	SF=1
		JNS LABEL	正数时转移	SF=0
		JO LABEL	溢出时转移	OF=1
		JNO LABEL	无溢出时转移	OF=0

条件转移指令

		格式	功能	测试条件
条件转移指令	对无符号数	JB/JNAE LABEL	低于/不高于等于时转移	CF=1
		JNB/JAE LABEL	不低于/高于等于时转移	CF=0
		JA/JNBE LABEL	高于/不低于等于时转移	CF=0且ZF=0
		JNA/JBE LABEL	不高于/低于等于时转移	CF=1或ZF=1
	对有符号数	JL/JNGE LABEL	小于/不大于等于时转移	SF≠OF
		JNL/JGE LABEL	不小于/大于等于时转移	SF=OF
		JG/JNLE LABEL	大于/不小于等于时转移	ZF=0且SF=OF
		JNG/JLE LABEL	不大于/小于等于时转移	ZF=1或SF≠OF

程序示例

计算存储器中[2000H]和[3000H]起始的两个数之和，数的长度存放在[2500H]字节单元




```
        MOV   CL, [2500H] ;
        MOV   SI, 2000H
        MOV   DI, 3000H
        CLC                    ; 将标志位CF清零
LOOP1:  MOV   AX, [SI]
        ADC   AX, [DI]
        MOV   [SI], AX
        INC   SI
        INC   SI               ; 可否使用“ADD SI，2”
        INC   DI
        INC   DI
        DEC   CL
        JNZ   LOOP1            ; 循环执行[2500]次
        MOV   AX, 0H
        ADC   AX, 0H
        MOV   [SI], AX
```

指令分类举例

1. 传送指令
2. 算术运算指令
3. 逻辑运算和移位指令
4. 转移指令
5. 处理器控制指令

逻辑运算和移位指令

作用

- 控制CPU的功能
- 对标志位进行操作

分组	格式	功能
标志操作指令	STC	把进位标志CF置1
	CLC	把进位标志CF清0
	CMC	把进位标志CF取反
	STD	把方向标志DF置1
	CLD	把方向标志DF清0
	STI	把中断标志IF置1
	CLI	把中断标志IF清0
外同步指令	HLT	暂停
	WAIT	等待
	ESC	交权
	LOCK	封锁总线（指令前缀）
空操作	NOP	空操作

第二章 指令系统体系结构

1.设计自己的计算机

2. x86体系结构

3. x86指令简介

4.复杂的x86指令举例

5. MIPS体系结构

6. MIPS指令简介

复杂的X86指令举例

1. 串操作指令
2. 循环控制指令
3. 查表指令
4. 十进制调整指令

串操作指令

作用

- 对存储器中的数据串进行每次一个元素的操作
- 串的基本单位是字节或字（即“一个元素”）
- 串长度可达64KB

分类

- 共5条串操作指令
- 另有3种重复前缀，与串操作指令配合使用

串操作指令

分组	助记符	功能
串操作指令	MOVS （MOVSB，MOVSW）	串传送 （字节串传送，字串传送）
	CMPS （CMPSB，CMPSW）	串比较 （字节串比较，字串比较）
	SCAS （SCASB，SCASW）	串扫描 （字节串扫描，字串扫描）
	LODS （LODSB，LODSW）	取串 （取字节串，取字串）
	STOS （STOSB，STOSW）	存串 （存字节串，存字串）
重复前缀	REP	无条件重复前缀
	REPE / REPZ	相等/为零重复前缀
	REPNE / REPNZ	不相等/不为零重复前缀

串传送指令说明

MOVSB指令（字节串传送）

格式：MOVSB

操作：在存储器中将指定位置的一个字节单元传送到另一个指定的位置

REP前缀（无条件重复）

格式：REP 串操作指令

操作：当CX≠0时，重复执行串操作指令

串操作指令的特性

隐含操作数

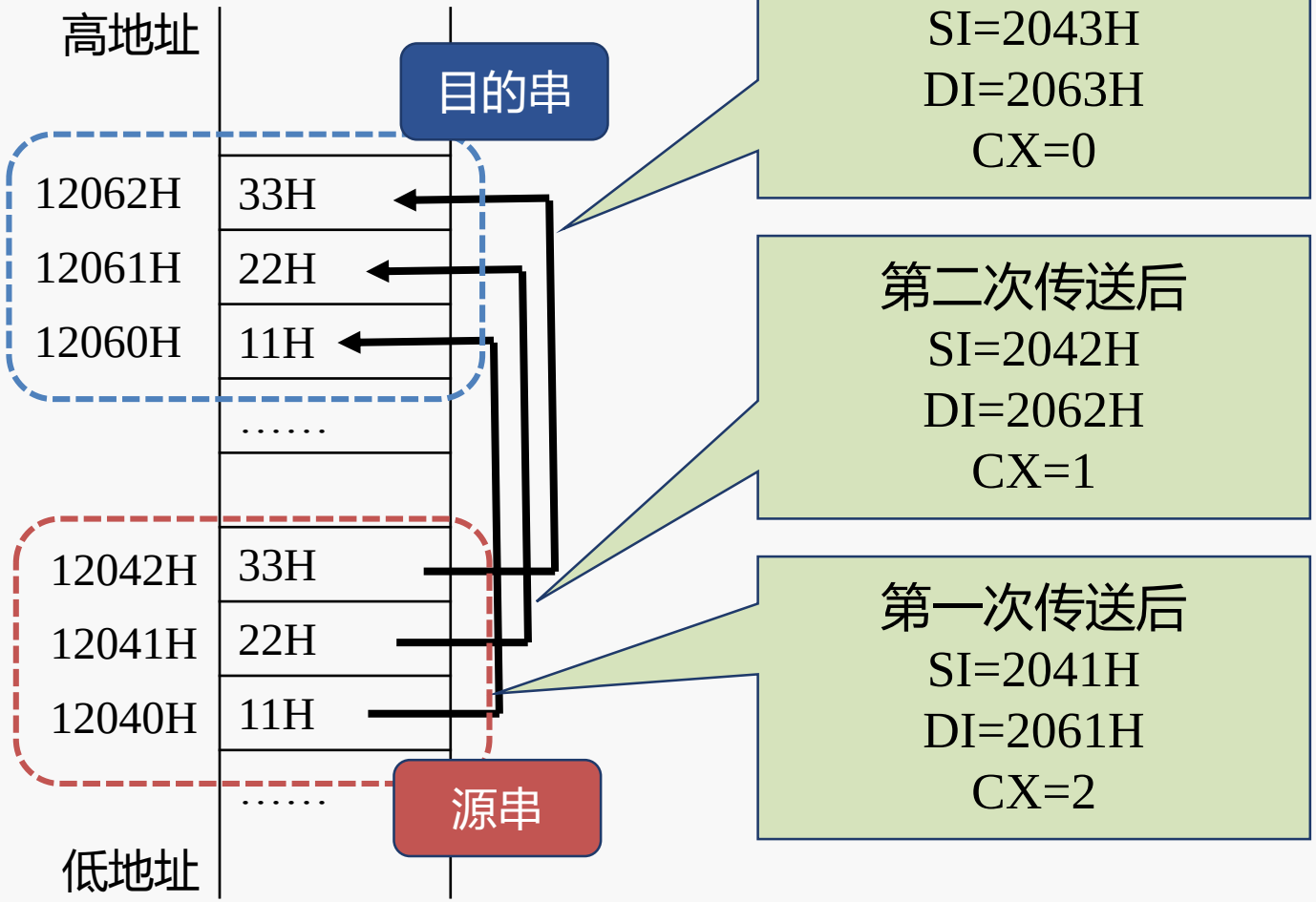
- 源串地址为DS:SI，目的串地址为ES:DI
- 串的长度在CX寄存器中

处理完一个串元素后的操作（硬件自动完成）

① 修改SI和DI，指向下一个串元素

② 若使用重复前缀，则CX ←CX-1

MOVSB指令示例



设 DS=1000H

```
MOV AX, DS
MOV ES, AX
MOV SI, 2040H
MOV DI, 2060H
CLD
MOV CX, 3
REP MOVSB
```

```
MOVSB;第一次传送
MOVSB;第二次传送
MOVSB;第三次传送
```

串传送方向（标志寄存器中的DF标志位）

设置DF=0

- 从“源串”的低地址开始传送
- 传送过程中，SI和DI自动增量修改

设置DF=1

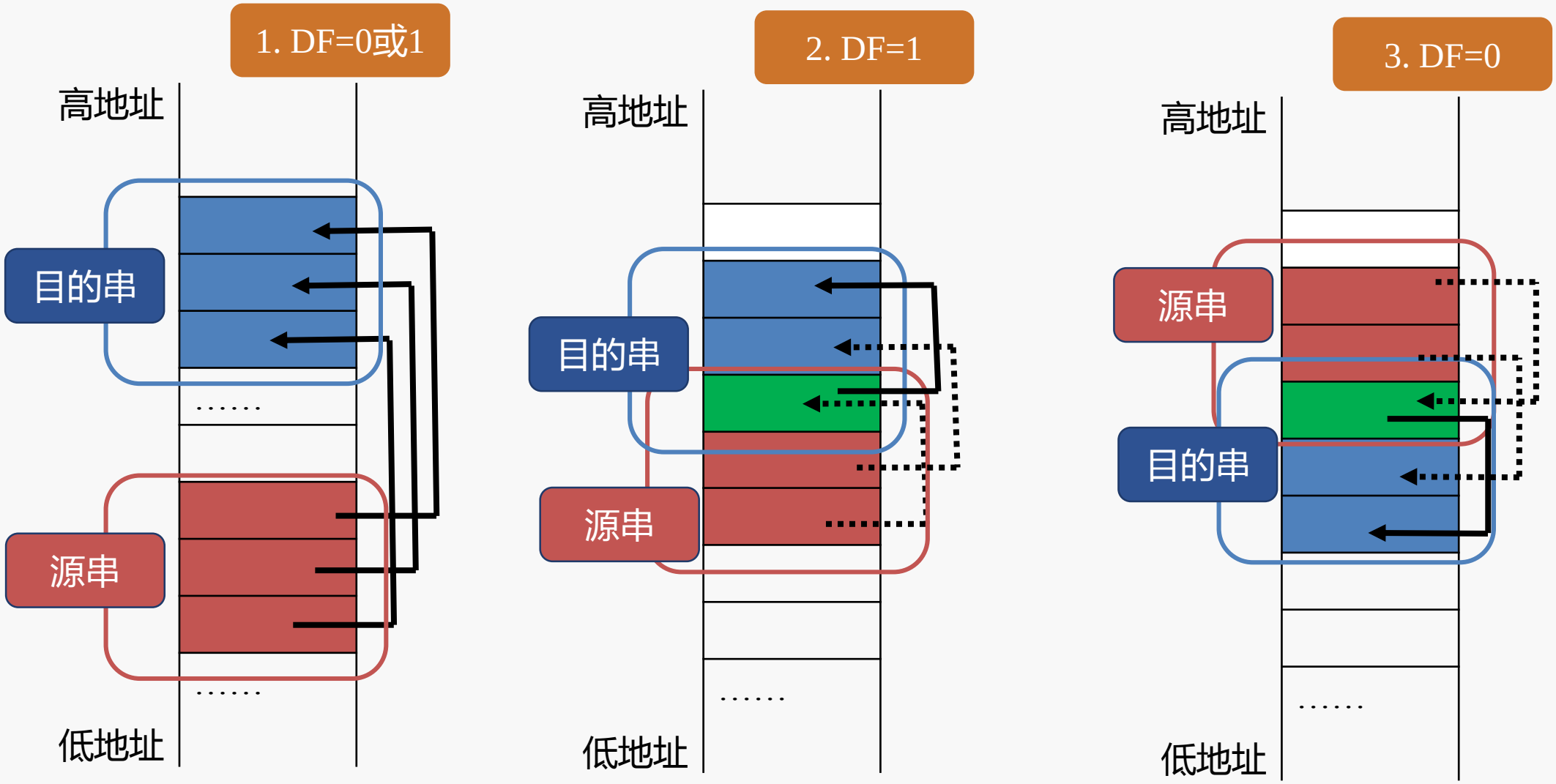
- 从“源串”的高地址开始传送
- 传送过程中，SI和DI自动减量修改

标志操作指令	
STD	把方向标志DF置1
CLD	把方向标志DF清0

串元素位宽 / 标志位	字节	字
方向标志DF=0	SI←SI+1; DI←DI+1	SI←SI+2; DI←DI+2
方向标志DF=1	SI←SI-1; DI←DI-1	SI←SI-2; DI←DI-2

方向标志的作用

应对“源串”和“目的串”的存储区域部分重叠的问题



复杂的X86指令举例

1. 串操作指令
2. 循环控制指令
3. 查表指令
4. 十进制调整指令

循环控制指令

格式	功能	测试条件
LOOP LABEL	循环	CX≠0
LOOPZ/LOOPE LABEL	为零/相等时循环	CX≠0且ZF=1
LOOPNZ/LOOPNE LABEL	不为零/不相等时循环	CX≠0且ZF=0
JCXZ	CX值为0时循环	CX=0

LOOPNZ/LOOPNE指令说明

LOOPNZ/LOOPNE指令（不为零/不相等时循环）

- 格式：LOOPNZ　LABEL
  或 LOOPNE　LABEL

- 操作

① CX←CX-1

② 若CX ≠0，转移到LABEL处继续执行
否则，结束循环，顺序执行下一条指令

循环控制指令示例

在100个字符的字符串中寻找第一个$字符

```
        MOV  CX，100
        MOV  SI，0FFFH
NEXT:INC SI
        CMP BYTE PTR [SI] ，’$’
        LOOPNZ NEXT
```

在循环出口
分析查找情况

ZF=0　CX=0	查找完毕，在串中没有$字符
ZF=1　CX≠0	已找到$字符，通过CX的内容可确定位置
ZF=1　CX=0	已找到$字符，在串的最后一个字符处

复杂的X86指令举例

1. 串操作指令
2. 循环控制指令
3. 查表指令
4. 十进制调整指令

XLAT指令说明

XLAT指令（换码，查表）

- 格式：XLAT

- 操作

（事先在数据段中定义了一个字节型数据表）

① 从BX中取得数据表起始地址的偏移量

② 从AL中取得数据表项索引值

③ 在数据表中查得表项内容

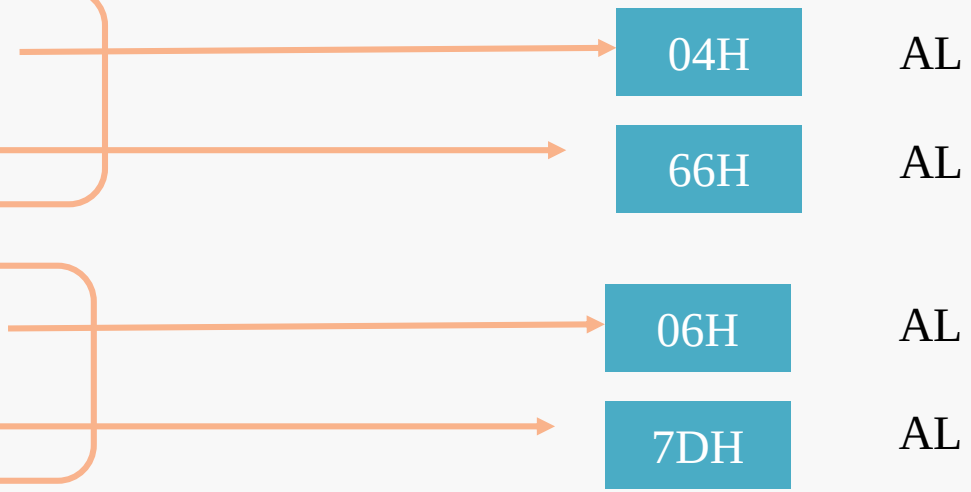
④ 将查得的表项内容存入AL

XLAT指令示例

```
TAB    DB  3FH，06H，5BH，4FH，66H
       DB  6DH，7DH，07H，7FH，6FH
……
MOV   BX，OFFSET        TAB
……
MOV   AL，4
XLAT
……
MOV AL，6
XLAT
```

字节型数据表

数据表起始地址的偏移量



复杂的X86指令举例

1. 串操作指令
2. 循环控制指令
3. 查表指令
4. 十进制调整指令

十进制调整指令

分组	助记符	功能
十进制调整	AAA	加法的ASCII调整
	DAA	加法的十进制调整
	AAS	减法的ASCII调整
	DAS	减法的十进制调整
	AAM	乘法的ASCII调整
	AAD	乘法的十进制调整

十进制调整指令说明

DAA指令（加法十进制调整指令）

- 格式：DAA

- 操作

（事先在数据段中定义了一个字节型数据表）

① 跟在二进制加法指令后

② 将AL中的“和”数调整为压缩BCD数据格式

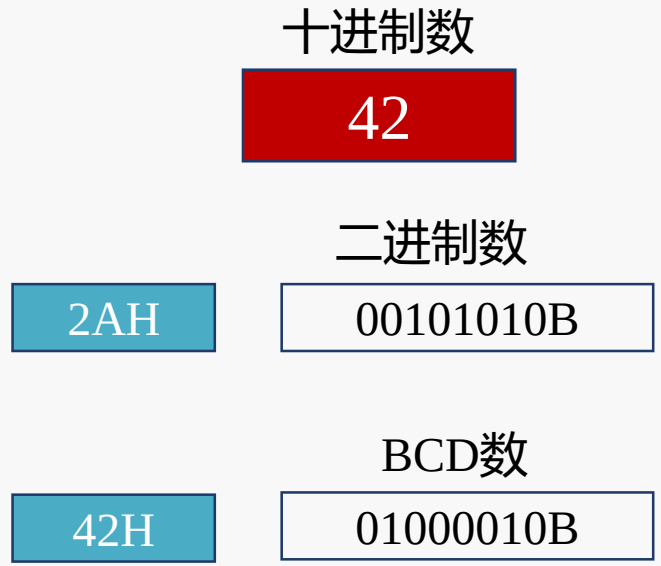
③ 调整结果送回AL

BCD（Binary-Coded Decimal）

BCD数具有二进制编码的形势，又保持了十进制的特点，可以作为人与计算机联系的中间标志

示例

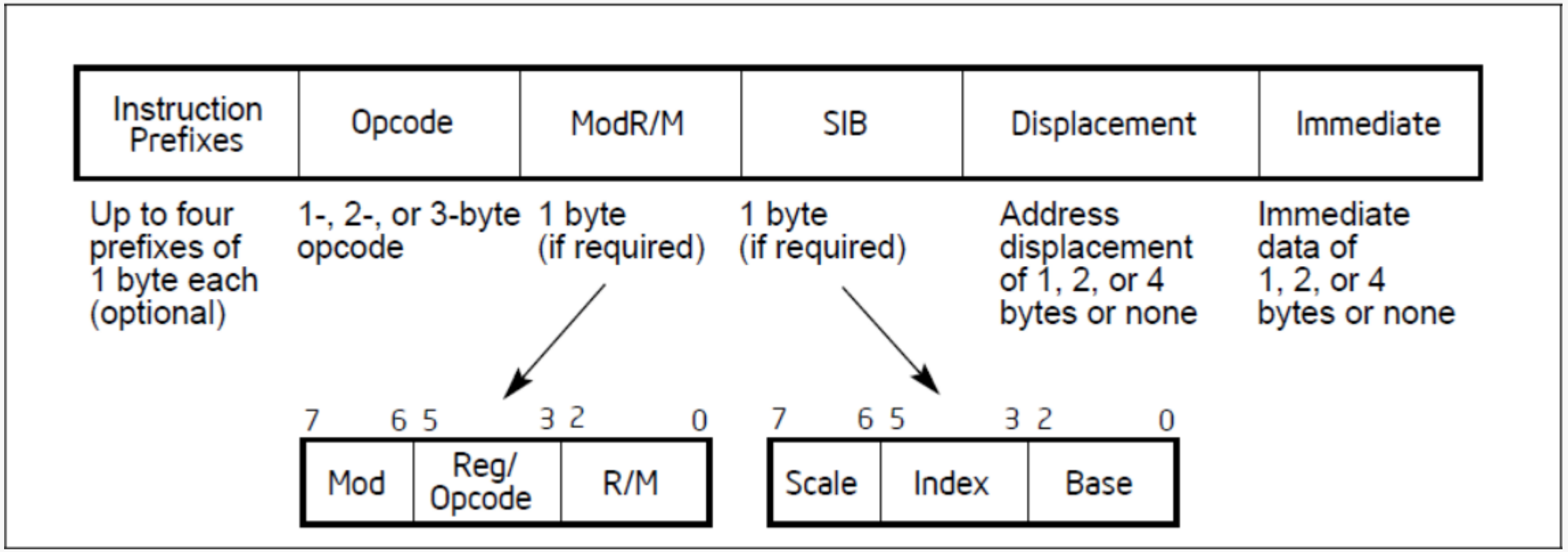
```
MOV  AL, 27H      ;  AL=27H
ADD  AL,15H       ;  AL=3CH
DAA               ;  AL=42H
;now: AL=01010101B
```



“最长的指令”

```
LOCK   ADD   DWORD   PTR   ES:[EAX+ECX*8+11223344H]，12345678H
```

指令编码（15个字节）：26 66 67 F0 81 84 C8 44 33 22 11 78 56 34 12



第二章 指令系统体系结构

1.设计自己的计算机

2. x86体系结构

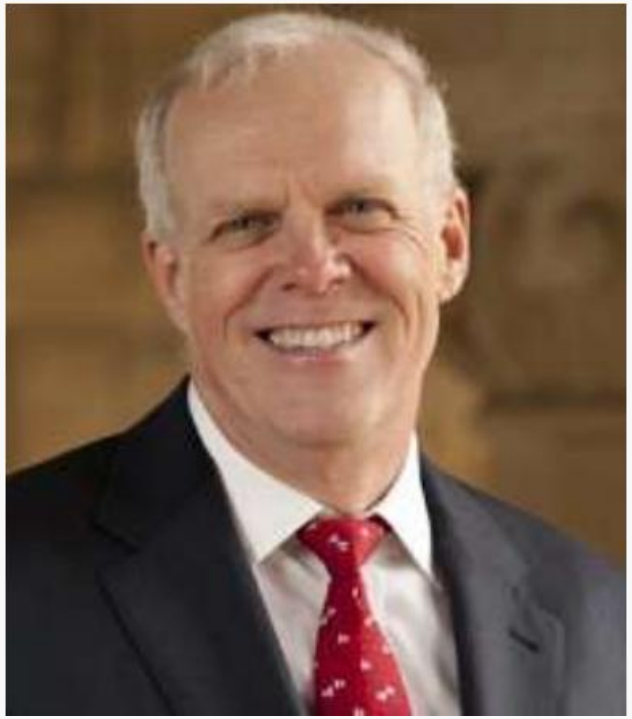
3. x86指令简介

4.复杂的x86指令举例

5. MIPS体系结构

6. MIPS指令简介

MIPS的设计者 和 RISC的先驱



约翰·亨尼西
John Hennessy
1953年出生

- 1977年，进入斯坦福大学任职
- 1981年，领导RISC微处理器的研究小组
- 1984年，共同创立MIPS计算机系统公司
- 1989年~1999年，先后担任斯坦福大学计算机系统实验室主任、计算机系主任和工程学院院长等
- 2000年起，任斯坦福大学校长

RISC：Reduced Instruction Set Computer，精简指令系统计算机

CISC：Complex Instruction Set Computer，复杂指令系统计算机

MIPS公司的商业兴衰

1984年，MIPS计算机系统公司成立

1988年，SGI公司在其计算机产品中采用MIPS处理器

1989年，MIPS第一次上市

1992年，SGI收购MIPS，更名为MIPS技术公司

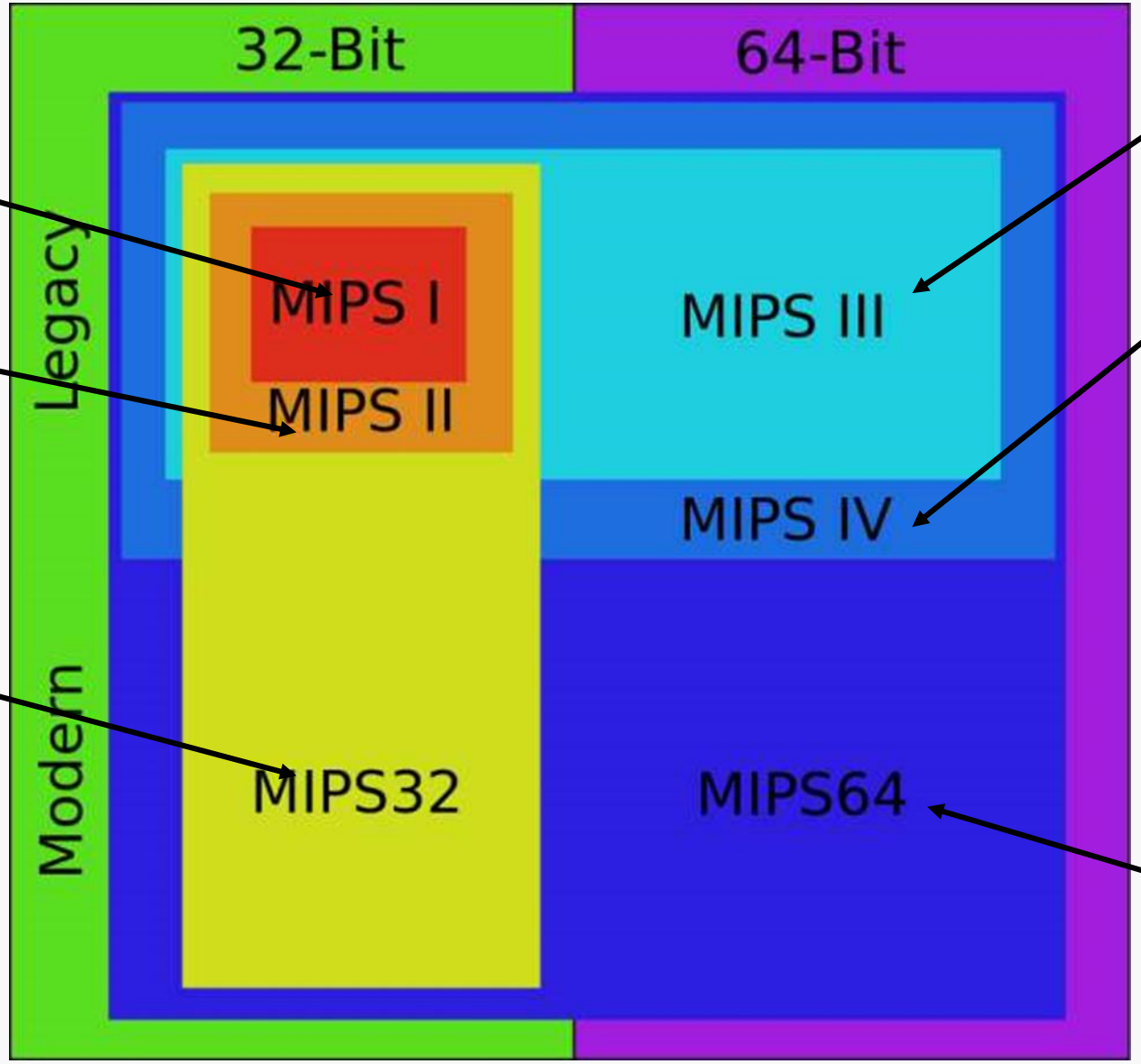
1998年，MIPS再次上市

2012年，Imagination Technologies收购MIPS

MIPS处理器广泛应用的领域：

- 数字电视、机顶盒、蓝光播放器、游戏机、网络设备等

MIPS指令的发展



1985年，R2000

1990年，R3000

1999年
以MIPS II为基础，
增加了MIPS
III/IV/V的部分特性

1992年，R4000
扩展到64位

1994年，R8000

MIPS V
1996年

1999年
以MIPS V为基础

MIPS的设计指导思想

MIPS的全称：

- Microprocessor without Interlocked Piped Stages

主要关注点

- 减少指令的类型
- 降低指令复杂度

基本原则

- A simpler CPU is a faster CPU.

MIPS指令的主要特点

固定的指令长度（32-bit，即1 word）：

- 简化了从存储器取指令

简单的寻址模式

- 简化了从存储器取操作数

指令数量少，指令功能简单（一条指令只完成一个操作）

- 简化指令的执行过程

只有Load和Store指令可以访问存储器

- 例如，不支持x86指令的这种操作：ADD AX,[3000H]

需要优秀的编译器支持

MIPS指令示例（运算指令）

加法指令

格式： add a, b, c

操作：将b和c求和，结果存入a中

算术运算	逻辑运算	移位运算
add a，b，c	and a，b，c	all a，b，c
sub a，b，c	or a，b，c	srl a，b，c
mul a，b，c		
div a，b，c		

MIPS指令示例（访存指令）

假设

- A是一个100个字（word）的数组，首地址在寄存器$19中
- 变量h对应寄存器$18
- 临时数据存放在寄存器$8简单的寻址模式

那么

- A[10]=h+A[3] 对应的MIPS指令为：

```
lw      $8,12($19)     #        t0=A[3]
add     $8,$18,$8      #        t0=h+A[3]
sw      $8,40($19)     #        A[10]=h+A[3]
```

MIPS的通用寄存器（32个，每个都是32位宽）

编号	名称	用途	编号	名称	用途
0	$zero	The Constant Value 0	24-25	$t8-$t9	Temporaries
1	$at	Assembler Temporary	26-27	$k0-$k1	Reserved for OS Kernel
2-3	$v0-$v1	Values for Function Results and Expression Evaluation	28*	$gp	Global Pointer
4-7	$a0-a3	Arguments	29*	$sp	Stack Pointer
8-15	$t0-$t7	Temporaries	30*	$fp	Frame Pointer
16-23*	$s0-$s7	Saved Temporaries	31*	$ra	Return Address

* Preserved across a call

通用寄存器使用示例

以下指令与对应注释中的指令相同

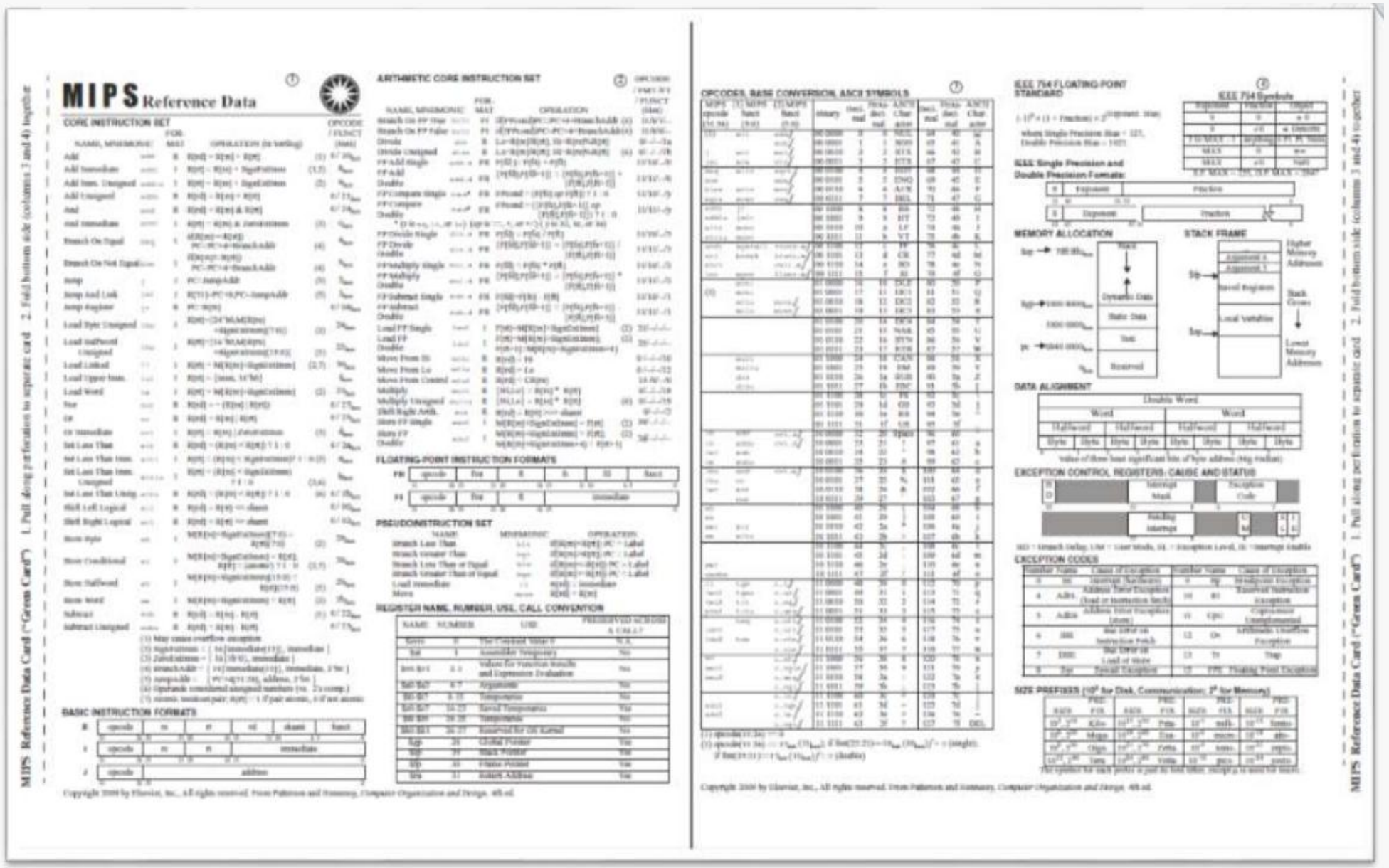
```
lw      $t0，12($s3)
#   lw   $8，12($19)

add      $t0，$s2，$t0
#   add $8，$18，$8

sw      $t0，40($s3)
#   sw   $8，40($19)
```

编号	名称	用途
8-15	$t0-$t7	Temporaries
16-23*	$s0-$s7	Saved Temporaries

第二章 指令系统体系结构

- 1.设计自己的计算机
- 2. x86体系结构
- 3. x86指令简介
- 4.复杂的x86指令举例
- 5. MIPS体系结构
- 6. MIPS指令简介

MIPS指令



MIPS Reference Data

CORE INSTRUCTION SET

ARITHMETIC CORE INSTRUCTION SET

FLOATING-POINT INSTRUCTION FORMATS

PSEUDOINSTRUCTION SET

REGISTER NAME, NUMBER, USE, CALL CONVENTION

BASIC INSTRUCTION FORMATS

OPCODES, BASE CONVERSION, ASCII SYMBOLS

IEEE 754 FLOATING POINT STANDARD

IEEE 754 Symbols

IEEE Single Precision and Double Precision Formats:

MEMORY ALLOCATION

STACK FRAME

DATA ALIGNMENT

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS

EXCEPTION CODES

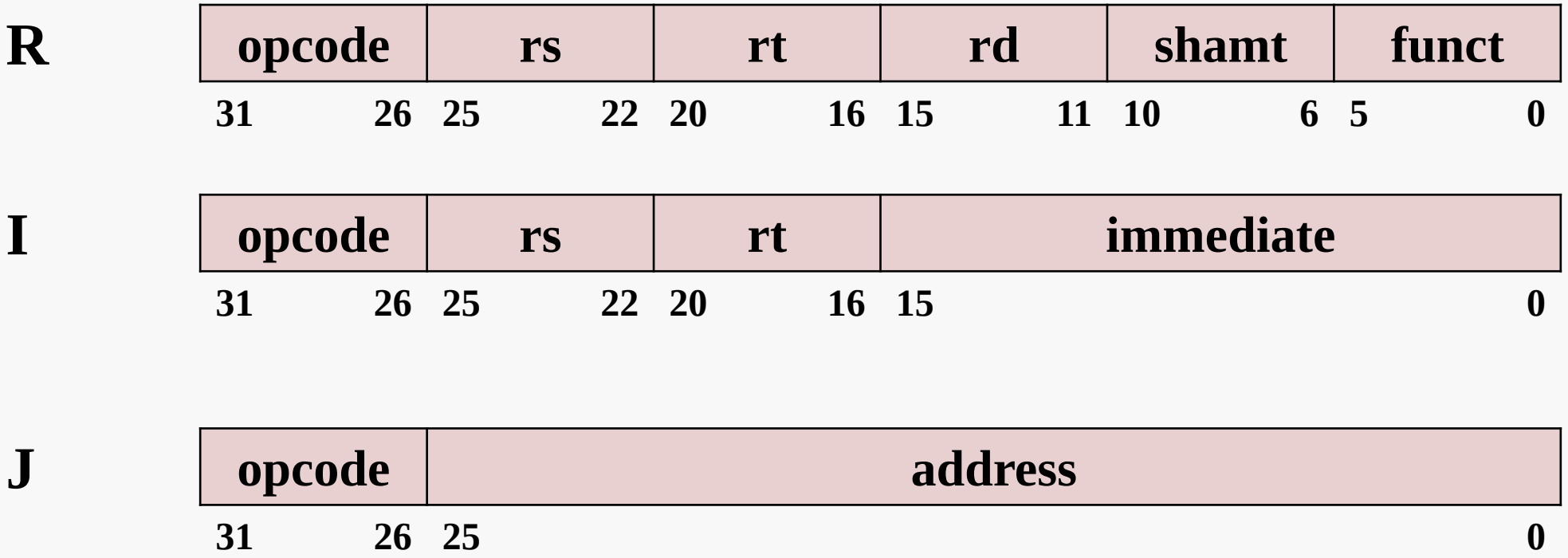
SIZE PREFIXES (10^n for Disk, Communication; 2^n for Memory)

MIPS指令的基本格式

R：Register，寄存器

I：Immediate，立即数

J：Jump，无条件转移



不同维度的指令分类（示例）

运算指令	add rd,rs,rt sll rd,rt,shamt	addi rt,rs,imm slti rt,rs,imm	/
访存指令	/	lw rt,imm(rs) sw rt,imm(rs)	/
分支指令	jr rs	beq rs,rt,imm	j addr
	R型指令	I型指令	J型指令

R型指令的格式（1）

R型指令格式包含6个域

- 2个6-bit域，可表示0~63的数
- 4个5-bit域，可表示0~31的数

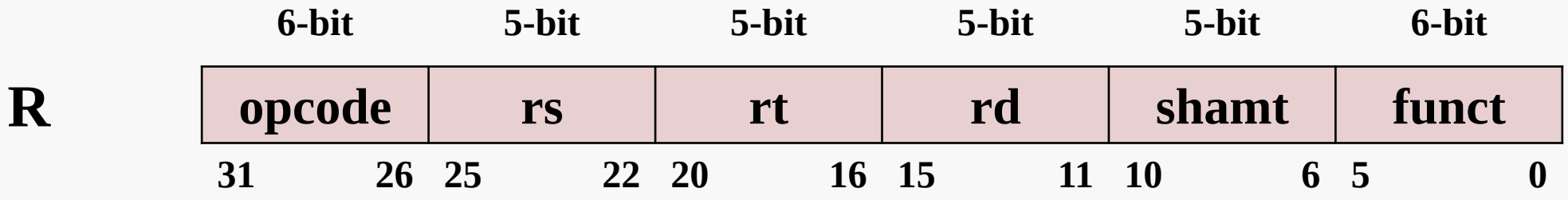
思考：
为什么不将opcode域和funct域合并成一个12-bit的域？

用于指定指令的类型。对于所有R型指令，该域的值均为0

opcode

与opcode域组合，精确地指定指令的类型

funct



R型指令的格式（2）

rs Source Register

◦ 通常用于指定第一个源操作数所在的寄存器编号

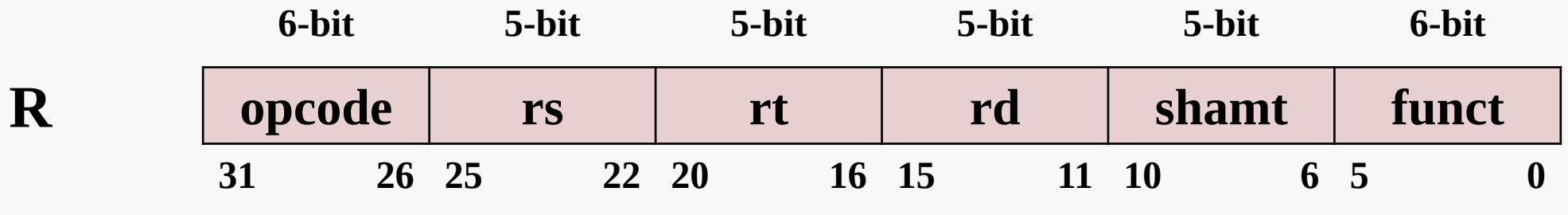
rt Target Register

◦ 通常用于指定第二个源操作数所在的寄存器编号

rd Destination Register

◦ 通常用于指定目的操作数（保存运算结果）的寄存器编号

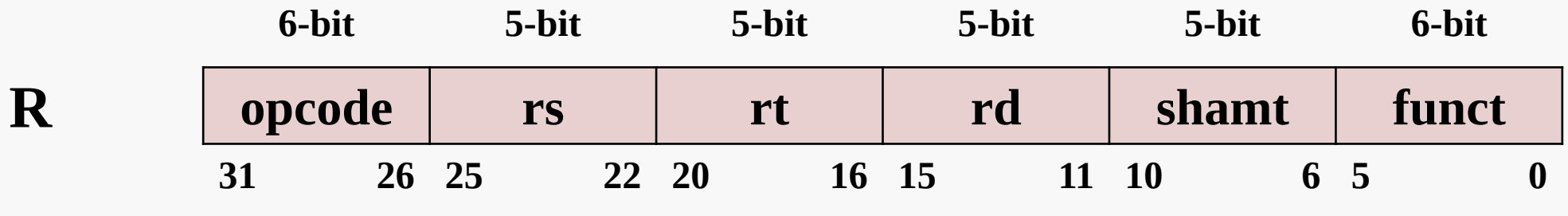
5-bit的域可表示0~31，对应32个通用寄存器



R型指令的格式（3）

shamt shift amount

- 用于指定移位指令进行移位操作的位数
- 5-bit的域可表示0~31，对于32-bit数，更多移位没有实际意义
- 对于非移位指令，该域设为0



R型指令的编码示例

add　$8，$9，$10

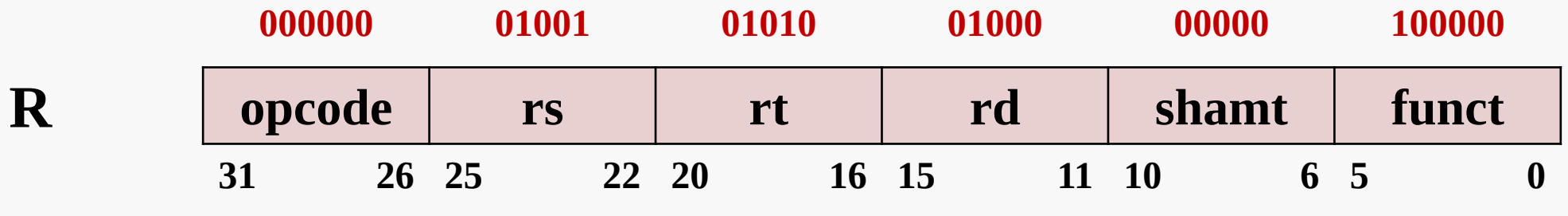
◦ 查指令编码表得到：

opcode = 0 ，funct = 32，shamt = 0（非移位指令）

◦ 根据指令操作数得到：

rd = 8（目的操作数），rs = 9（第一个源操作数）

rt = 10（第二个源操作数）



不同维度的指令分类（示例）

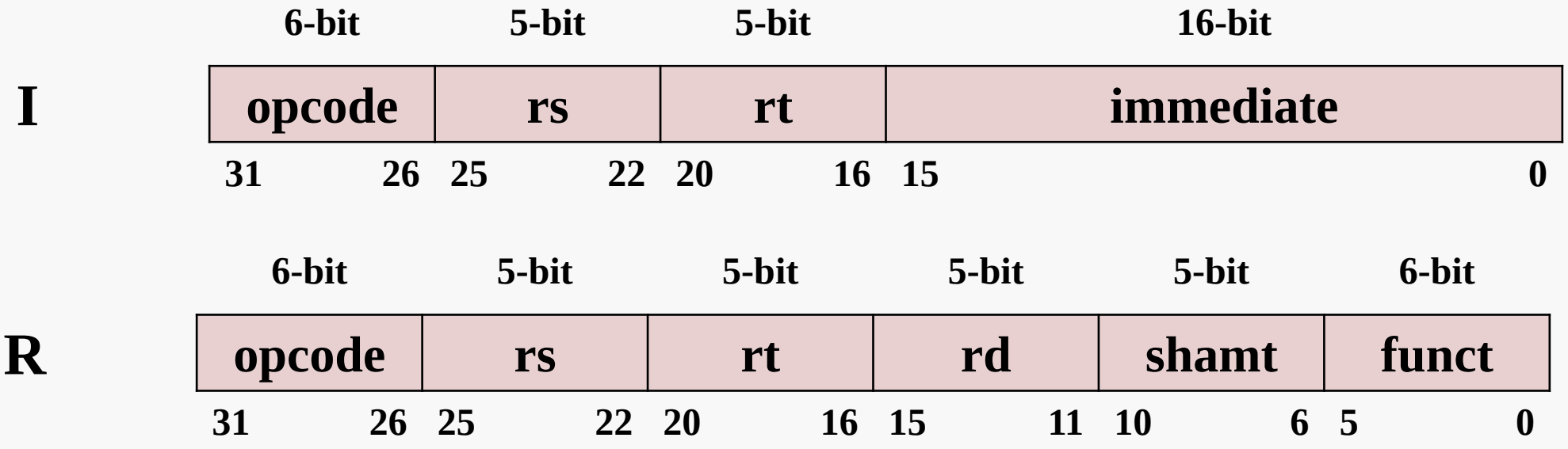
运算指令	add rd,rs,rt sll rd,rt,shamt	addi rt,rs,imm slti rt,rs,imm	/
访存指令	/	lw rt,imm(rs) sw rt,imm(rs)	/
分支指令	jr rs	beq rs,rt,imm	j addr
	R型指令	**I型指令**	**J型指令**

I型指令的格式（1）

R型指令只有一个5-bit域表示立即数，范围为0~31

常用的立即数远大于这个范围，因此需要新的指令格式

I型指令的大部分域与R型指令相同



I型指令的格式（2）

opcode

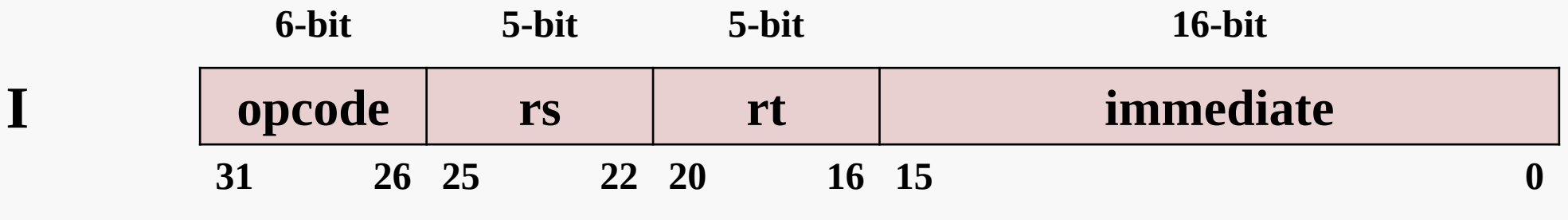
- 用于指定指令的操作类型（但没有funct域）

rs Source Register

- 指定第一个源操作数所在的寄存器编号

rt Target Register

- 指定用于目的操作数（保存运算结果）的寄存器编号
- 对于某些指令，指定第二个源操作数所在的寄存器编号



I型指令的格式（3）

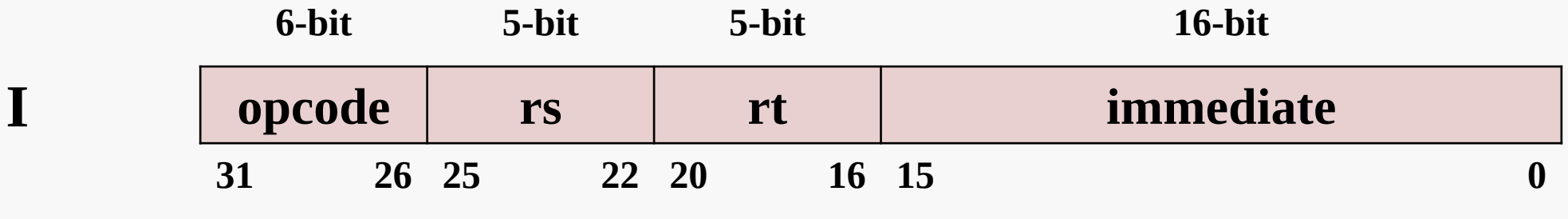
immediate

- 16-bit的立即数，可以表示216个不同数值
- 对于访存指令，如lw rt,imm(rs)

通常可以满足访存地址偏移量的需求（-32768~+32767）

- 对于运算指令，如addi rt,rs,imm

无法满足全部需求，但大多数时候可以满足需求



I型指令的编码示例

addi　$21，$22，-50　　#　　$21=$22+(-50)

◦查指令编码表得到：

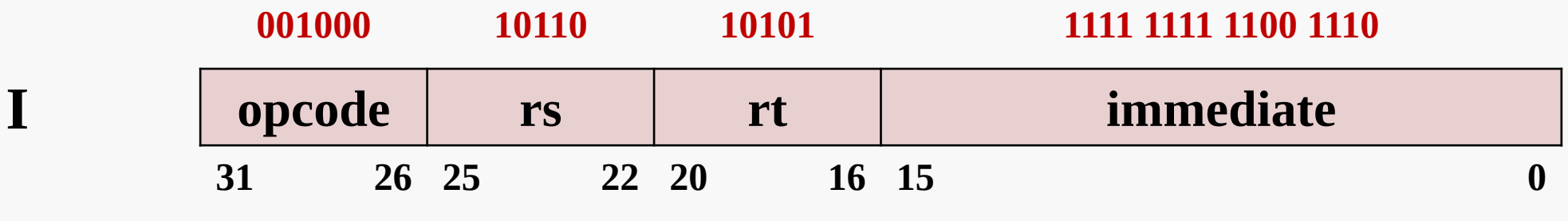
opcode = 8

◦分析指令得到：

rs = 22（源操作数寄存器编号）

rt = 21（目的操作数寄存器编号）

immediate = -50（立即数）



不同维度的指令分类（示例）

	R型指令	I型指令	J型指令
运算指令	add rd,rs,rt sll rd,rt,shamt	addi rt,rs,imm slti rt,rs,imm	/
访存指令	/	lw rt,imm(rs) sw rt,imm(rs)	/
分支指令	jr rs	beq rs,rt,imm	j addr

分支指令的分类

Branch

- 分支：改变控制流

Conditional Branch

- 条件分支：根据比较的结果改变控制流
- 两条指令：branch if equal (beq) ；branch if not equal (bne)

Unconditional Branch

- 非条件分支：无条件地改变控制流
- 一条指令： jump (j)

条件分支指令（I型）

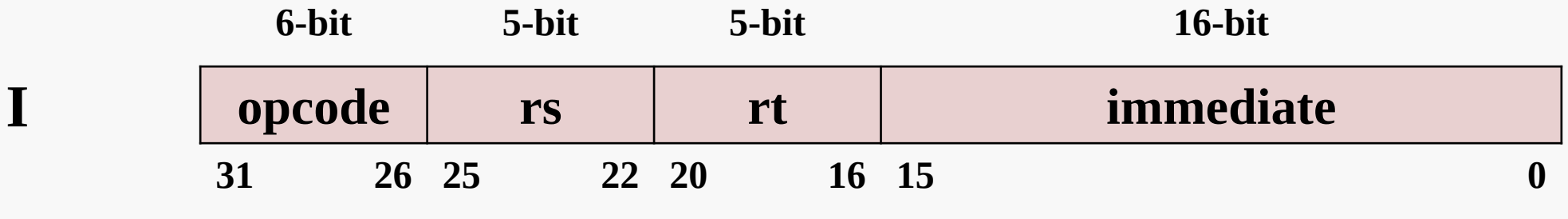
条件分支

- beq rs，rt，imm　　# opcode=4
- bne rs，rt，imm　　# opcode=5

格式：beq reg1，reg2，L1

if (value in reg1)==(value in reg2)

goto L1



条件分支指令的示例

```
if(i==j)
        f=g+h;
else
        f=g-h;
```

C语言代码

```
beq $s3，$s4，True          #   branch  i==j
sub $s0，$s1，$s2           #   f=g-h(false)
j Fin                       # goto  Fin

True: add $s0，$s1，$s2                 # f=g+h (true)
Fin: …
```

MIPS汇编语言代码

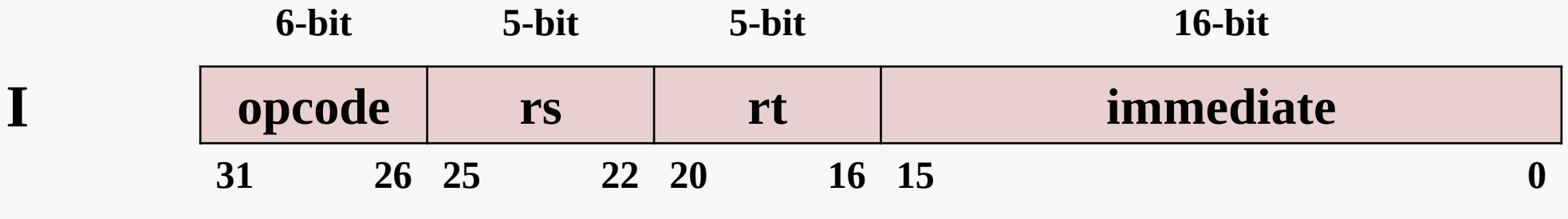
条件分支指令的目标地址范围

如何充分发挥16-bit的作用？

- 以当前PC为基准，16-bit位移量可以表示$\pm 2^{15}$ bytes
- MIPS的指令长度固定为32-bit（word）
- 16-bit位移量可以表示 $\pm 2^{15}$ words = $\pm 2^{17}$ bytes（±128KB）

目标地址计算方法：

- 分支条件不成立，PC = PC + 4 = next instruction
- 分支条件成立，PC = (PC+4) + (immediate*4)



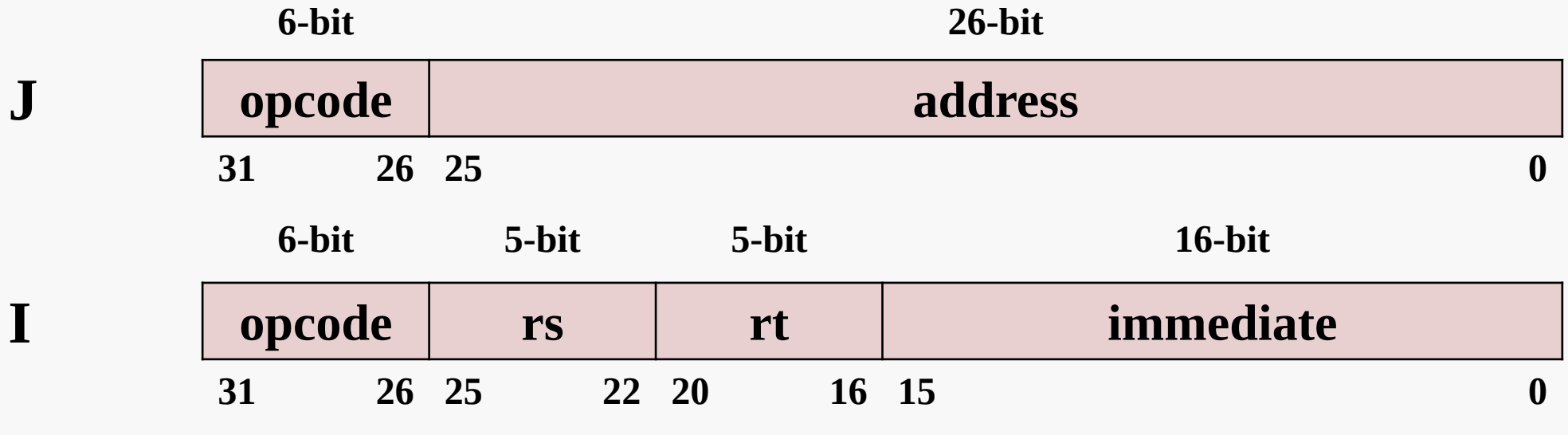
非条件分支指令（J型）

在不需要条件判断的情况下，如何扩大目标地址范围

- 理想情况，直接使用32-bit地址
- 冲突：MIPS的指令长度固定为32-bit，opcode占用了6-bit

目标地址计算方法：

- New PC ={(PC+4)[31..28], address, 00}



两种分支指令示例

假设变量和寄存器的对应关系如下

f → $s0　　g → $s1　　h → $s2

i → $s3　　j → $s4

```
if  (i == j)
    f  = g + h;
Else
    f = g – h;
```

```
    bne  $s3，$s4，Else
    add  $s0，$s1，$s2
    j Exit
Else: sub $s0，$s1，$s2
Exit:
```

非条件分支指令（R型）

J型指令的目标地址范围：$\pm 2^{28}$bytes（±256MB）

如何到达更远的目标地址

- 2次调用j指令
- 使用jr指令：jr rs