

## 计算机系统 3

### 课程回顾

# 总体内容

- **Ch1 计算机概要**
- **Ch2 指令**
- **Ch3 算术运算**
- **Ch4 处理器**
- **Ch5 存储**

# Ch1

## ■ 8个伟大思想

面向摩尔定律，抽象简化设计，加速大概率事件，并行提高性能  
流水线提高性能，预测提高性能，存储器层次，冗余提高可靠性

## ■ 性能的评价准则

响应时间（运行时间） / 吞吐率 / CPI / ISA

性能 = 1/执行时间

CPU时钟周期数 = 程序的指令数 × 每条指令的平均时钟周期数（CPI）

$$\begin{aligned}\text{CPU 时间 (t}_{\text{CPU}}) &= \frac{\text{CPU 时钟周期数}}{\text{时钟频率}} \\ &= \text{程序的指令数} \times \text{CPI} \times \text{时钟周期时间} \\ &= \frac{\text{程序的指令数} \times \text{CPI}}{\text{时钟频率}}\end{aligned}$$

给出t<sub>CPU</sub>和CPI：量化计算，比较性能

# Ch1

## ■ CPI

$$\text{总CPU时钟周期数} = \sum_{i=1}^n (\text{CPI}_i \times C_i) \quad \text{平均CPI} = \frac{\text{总CPU时钟周期数}}{\text{指令数}n} = \sum_{i=1}^n \left( \text{CPI}_i \times \frac{C_i}{\text{指令数}n} \right)$$

指令类型	A	B	C
CPI	1	2	3
代码序列1的指令数IC	2	1	2
代码序列2的指令数IC	4	1	1

### ■ 序列 1: IC = 5

- CPU时钟周期数  
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$   
 $= 10$
- 平均 CPI =  $10/5 = 2.0$

### ■ 序列 2: IC = 6

- CPU时钟周期数  
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$   
 $= 9$
- 平均 CPI =  $9/6 = 1.5$

# Ch1

## ■ 功耗

$$\text{功耗} = \frac{1}{2} \times \text{负载电容} \times \text{电压}^2 \times \text{开关频率}$$

# Ch2

## ■ MIPS 的寄存器: 32 × 32-bit

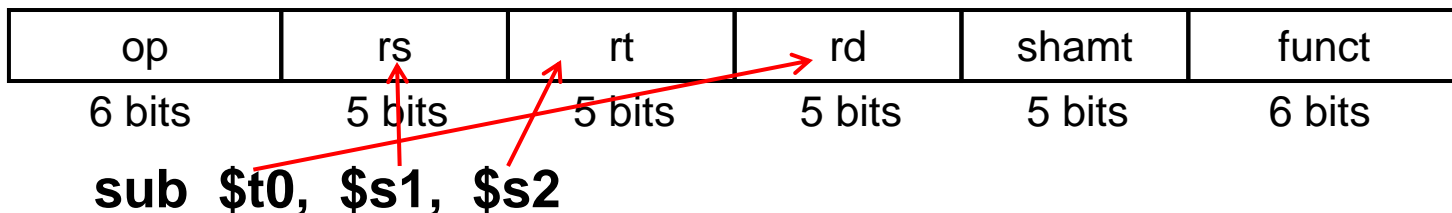
寄存器名字	寄存器编号	寄存器功能
\$zero	\$0	恒等于零
\$at	\$1	被汇编器保留, 用于处理大的常数
\$v0 – \$v1	\$2-\$3	存放函数返回值
\$a0 – \$a3	\$4-\$7	传递函数参数
\$t0 – \$t7	\$8-\$15	存放临时变量
\$s0 – \$s7	\$16-\$23	存放需要保存的临时值
\$t8 – \$t9	\$24-\$25	额外的存放临时变量
\$k0 – \$k1	\$26-\$27	用于操作系统内核
\$gp	\$28	指向全局变量的指针
\$sp	\$29	指向栈顶的指针
\$fp	\$30	指向栈帧的指针
\$ra	\$31	返回地址, 用于函数调用

# Ch2

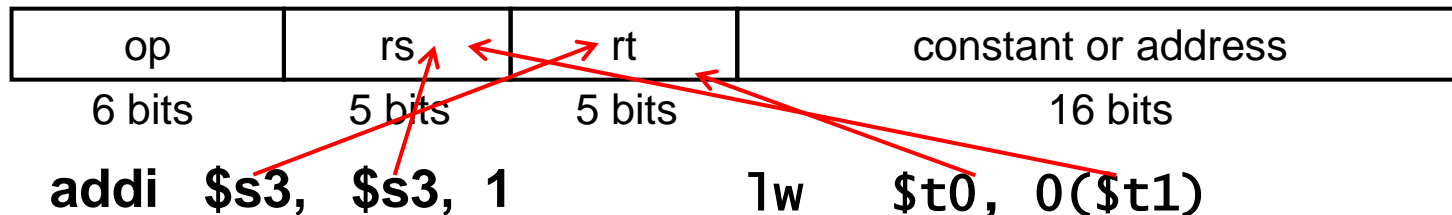
## MIPS 指令

- 2进制编码—机器码
- 每条指令编码为 32位指令字
- 用较小的数值编码为操作代码（**opcode**）、寄存器

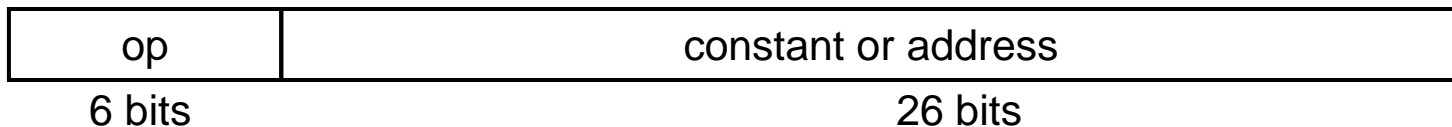
R



I



J



## 指令寻址模式

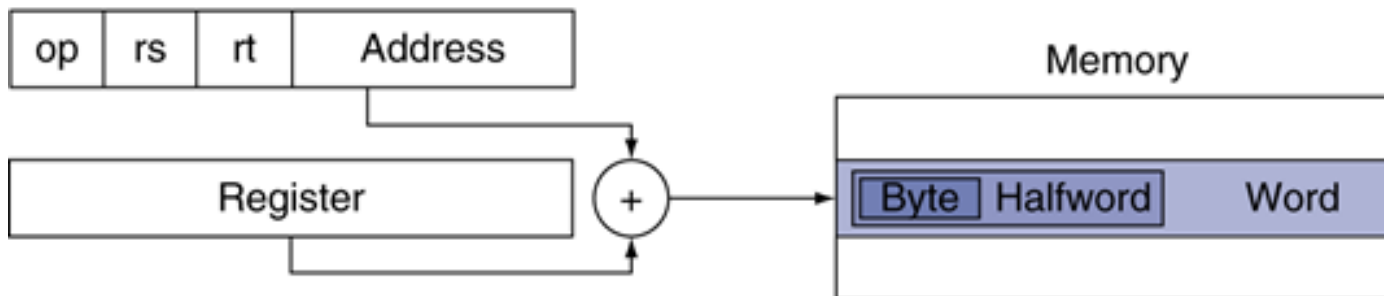
### 1. Immediate addressing



### 2. Register addressing



### 3. Base addressing

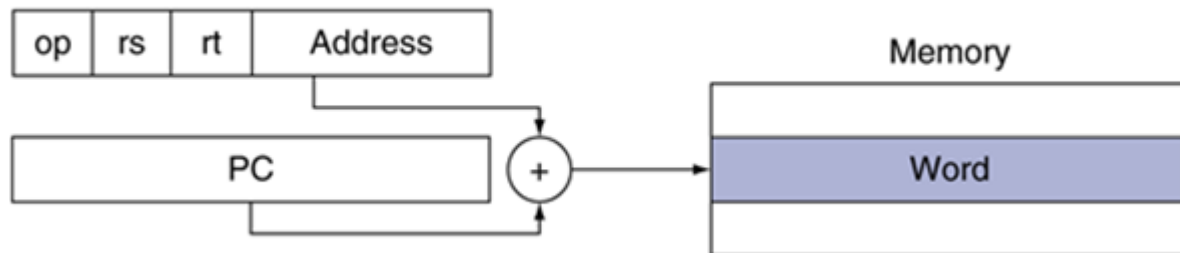




# Ch2

## ■ 指令寻址模式

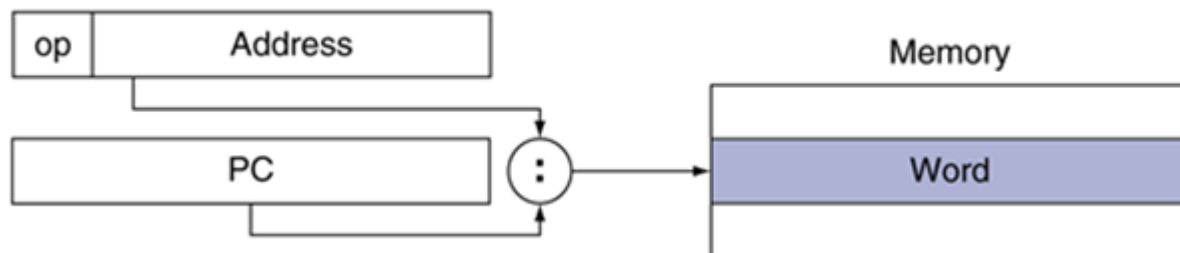
### 4. PC-relative addressing



### ■ PC相对寻址

- 目标地址 =  $(PC+4) + \text{offset} \times 4$

### 5. Pseudodirect addressing



### ■ 直接跳转到地址

- Target address =  $PC_{31...28} : (\text{address} \times 4)$

字操作

# Ch2

## ■ 指令的机器码

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32 <sub>ten</sub>	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34 <sub>ten</sub>	n.a.
add immediate	I	8 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43 <sub>ten</sub>	reg	reg	n.a.	n.a.	n.a.	address

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add \$s1,\$s2,\$s3
sub	R	0	18	19	17	0	34	sub \$s1,\$s2,\$s3
addi	I	8	18	17	100			addi \$s1,\$s2,100
lw	I	35	18	17	100			lw \$s1,100(\$s2)
sw	I	43	18	17	100			sw \$s1,100(\$s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

sub \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	sub
0	17	18	8	0	34
000000	10001	10010	01000	00000	100000

# Ch2

## ■ MIPS 汇编指令 $\Leftrightarrow$ C 语言指令

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

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

```
Loop: sll $t1, $s3, 2
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
```

```
while (save[i] == k) i += 1;
```

```
Exit: ...
```

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

```
if ($s1 < $s2)
goto L
```

# Ch2

## ■ 函数调用

- 命令    过程调用: 跳转和链接 `jal ProcedureLabel`
  - 下一条指令的地址在寄存器 `$ra` 中 (自动)
  - 跳转到目标地址

过程返回: 寄存器跳转 `jr $ra`

- 寄存器
  - 复制 `$ra` 到程序计数器
  - 也可以用于运算后跳转

`$a0 – $a3`: 传递参数(reg's 4 – 7)

`$v0, $v1`: 返回结果值(reg's 2 and 3)

`$t0 – $t9`: 临时寄存器(reg's 8 – 15, 24-25), 可以被调用者改写

`$s0 – $s7`: 保存参数(reg's 16 – 23), 必须被调用者 (过程) 保存和恢复

`$sp`: 堆栈指针寄存器(reg 29)

`$fp`: 帧指针寄存器-保存过程帧的第一个字(reg 30)

`$ra`: 返回地址寄存器 (reg 31)

# Ch3

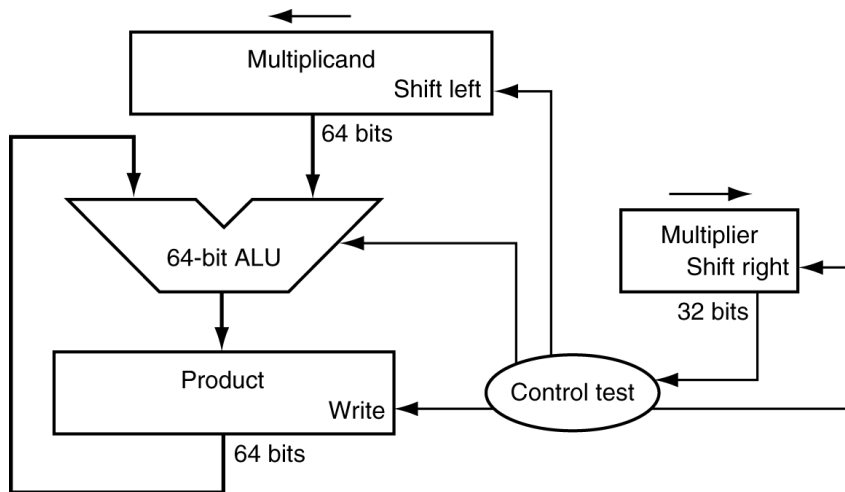
## ■ 加減法

溢出

操作	A	B	溢出条件
A+B	$\geq 0$	$\geq 0$	$< 0$
A+B	$< 0$	$< 0$	$\geq 0$
A-B	$\geq 0$	$< 0$	$< 0$
A-B	$< 0$	$\geq 0$	$\geq 0$

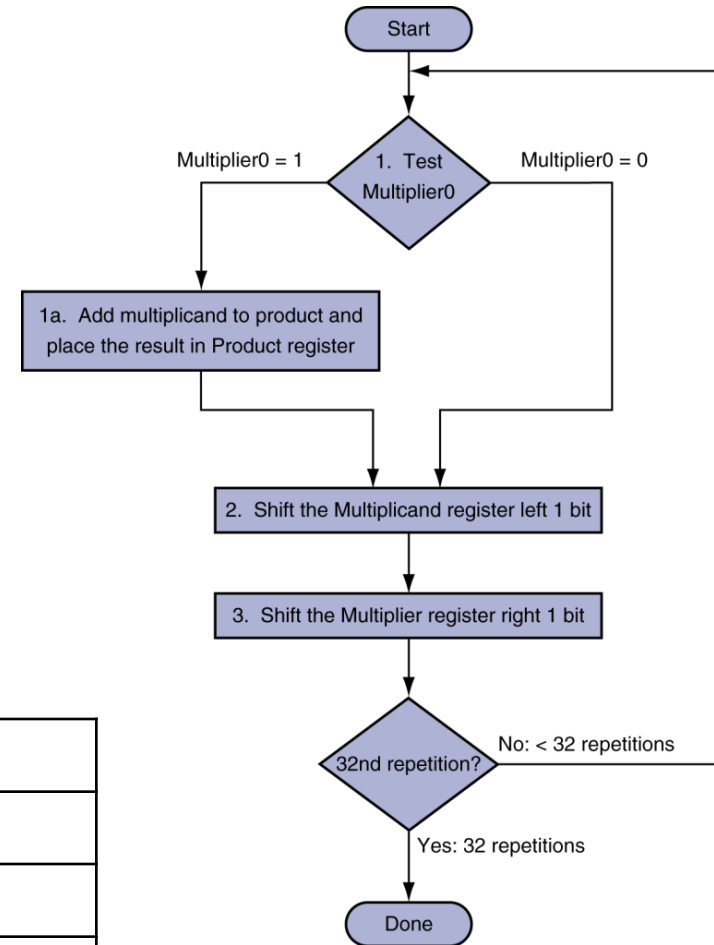
# Ch3

## ■ 乗法



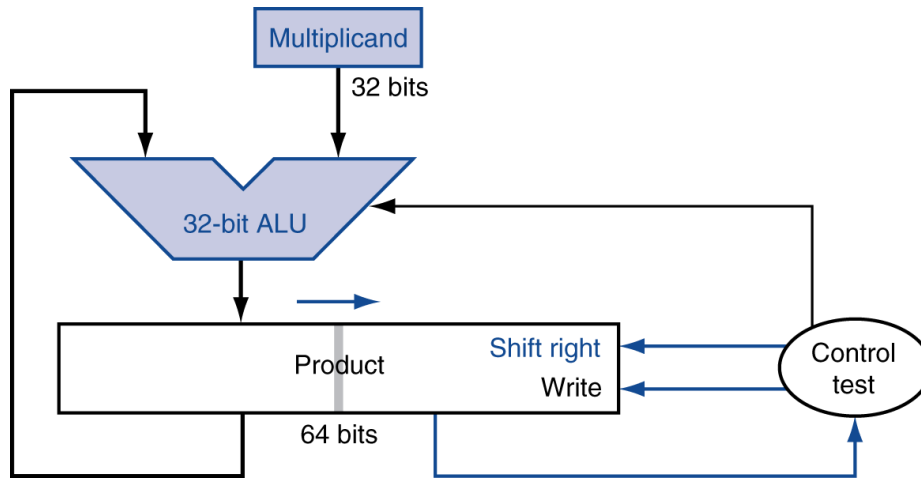
101 X 011

次数	乗数	被乗数	積
0	011	000 101	000 000
1	001	001 010	000 101
2	000	010 100	001 111
3	000	101 000	001 111



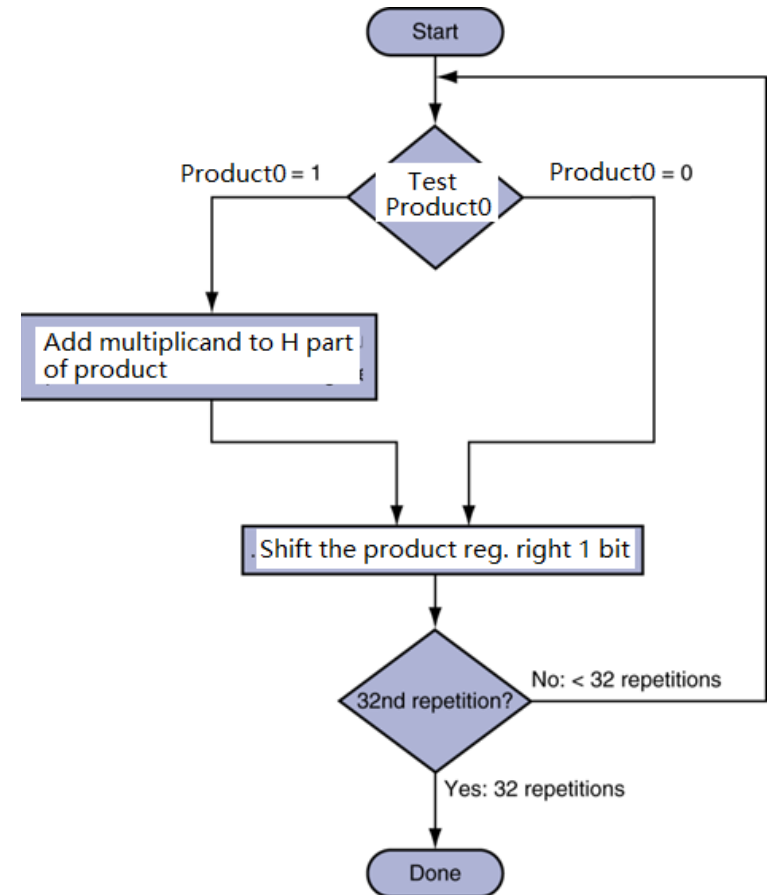
# Ch3

## 改进乘法



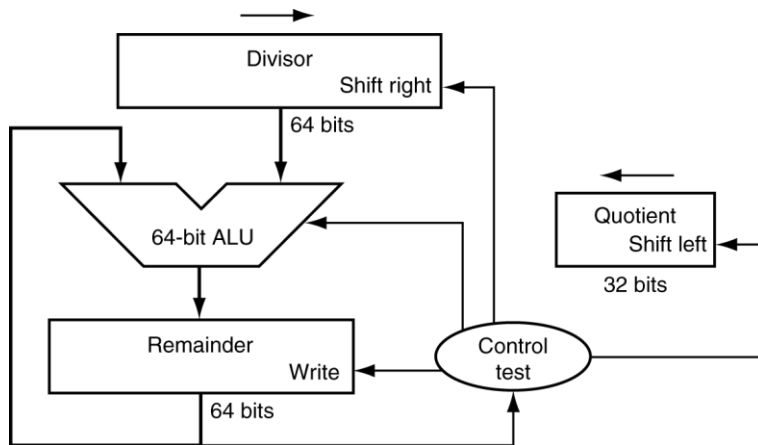
101 X 011

次数	被乘数	积
0	101	000 011
1	101	010 101
2	101	011 110
3	101	001 111

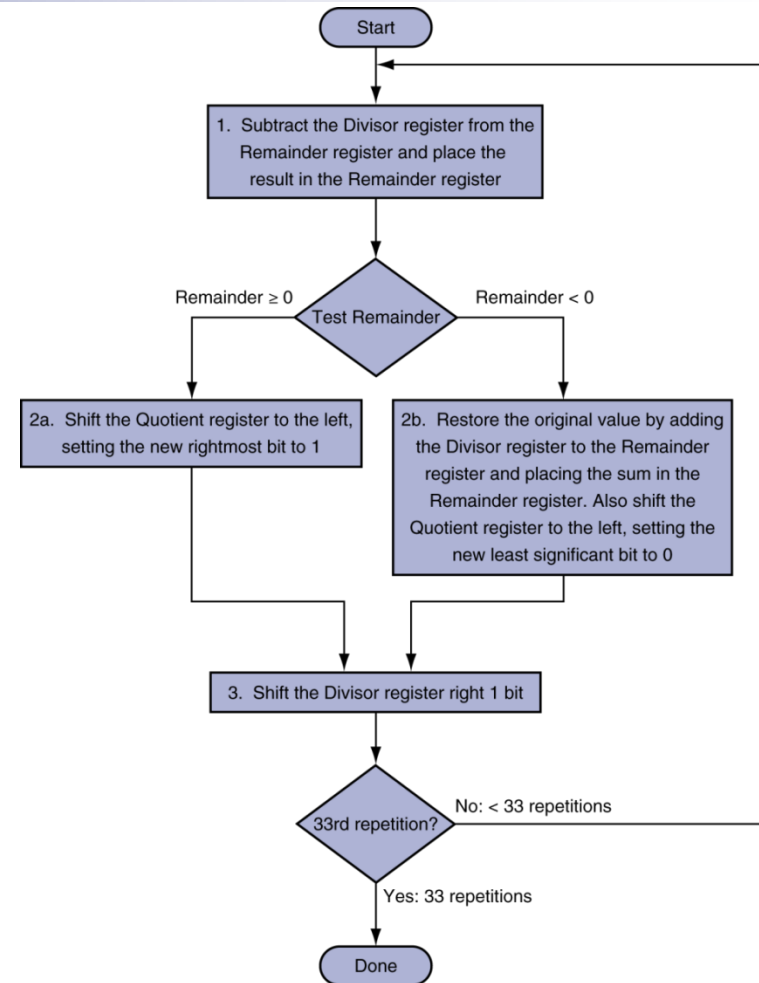


# Ch3

## ■ 除法 101 / 010



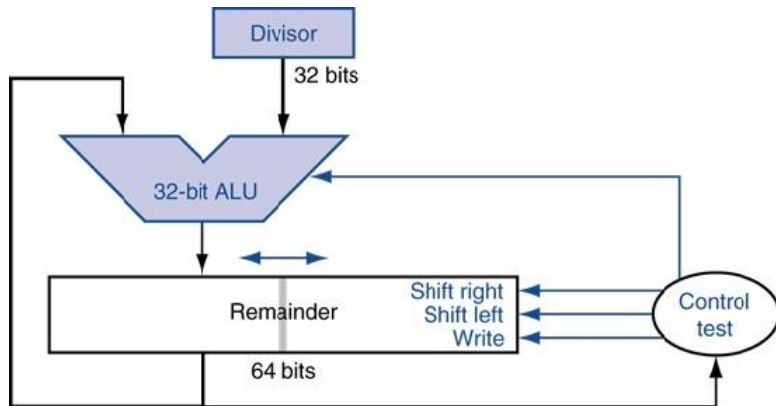
次数	商	除数	余数
0	000	010 000	000 101
1	000	010 000	110 101
	000	001 000	000 101
2	000	001 000	111 101
	000	000 100	000 101
3	000	000 100	000 001
	001	000 010	000 001
4	001	000 010	111 111
	010	000 001	000 001



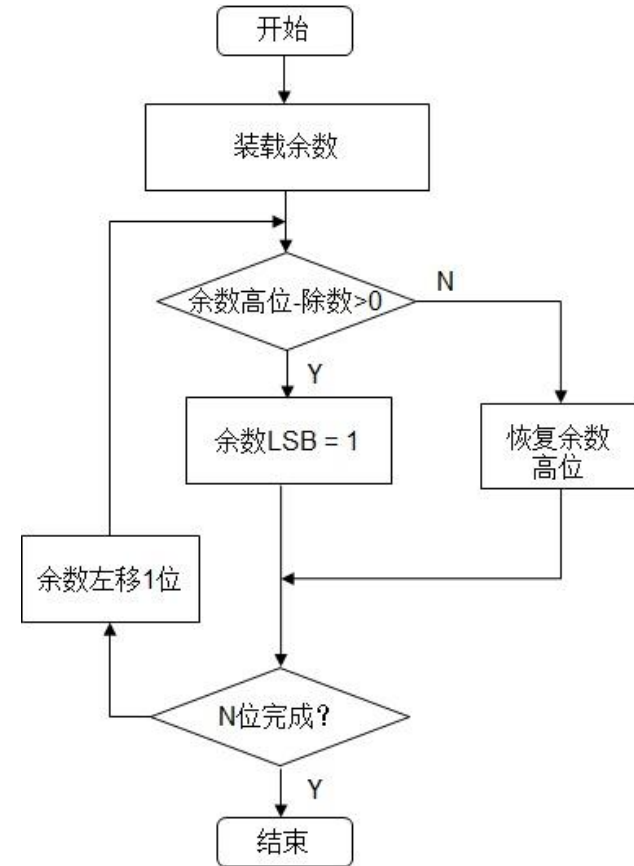


# Ch3

## ■ 优化除法 101 / 010



次数	除数	余数
0	010	000 101
1	010	110 000 101 001 010
2	010	111 001 010 010 100
3	010	000 100 000 101 001 010
4	010	111 001 010



# Ch3

## ■ 浮点数-IEEE 754

S	阶码	尾数
---	----	----

$$x = (-1)^S \times (1 + \text{尾数}) \times 2^{(\text{阶码} - \text{偏移})}$$

S: 符号位 (0  $\Rightarrow$  非负数, 1  $\Rightarrow$  负数), 有效位: "1.尾数"

阶码: 真实指数 + 偏移; 单精度: 偏移 = 127; 双精度: 偏移 = 1203

$$-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$$

S = 1, 尾数 = 1000...00<sub>2</sub>, 阶码 = -1 + Bias

单精度: -1 + 127 = 126 = 01111110<sub>2</sub>

双精度: -1 + 1023 = 1022 = 01111111110<sub>2</sub>

单精度: 1011111101000...00

双精度: 1011111111101000...00

11000000101000...00

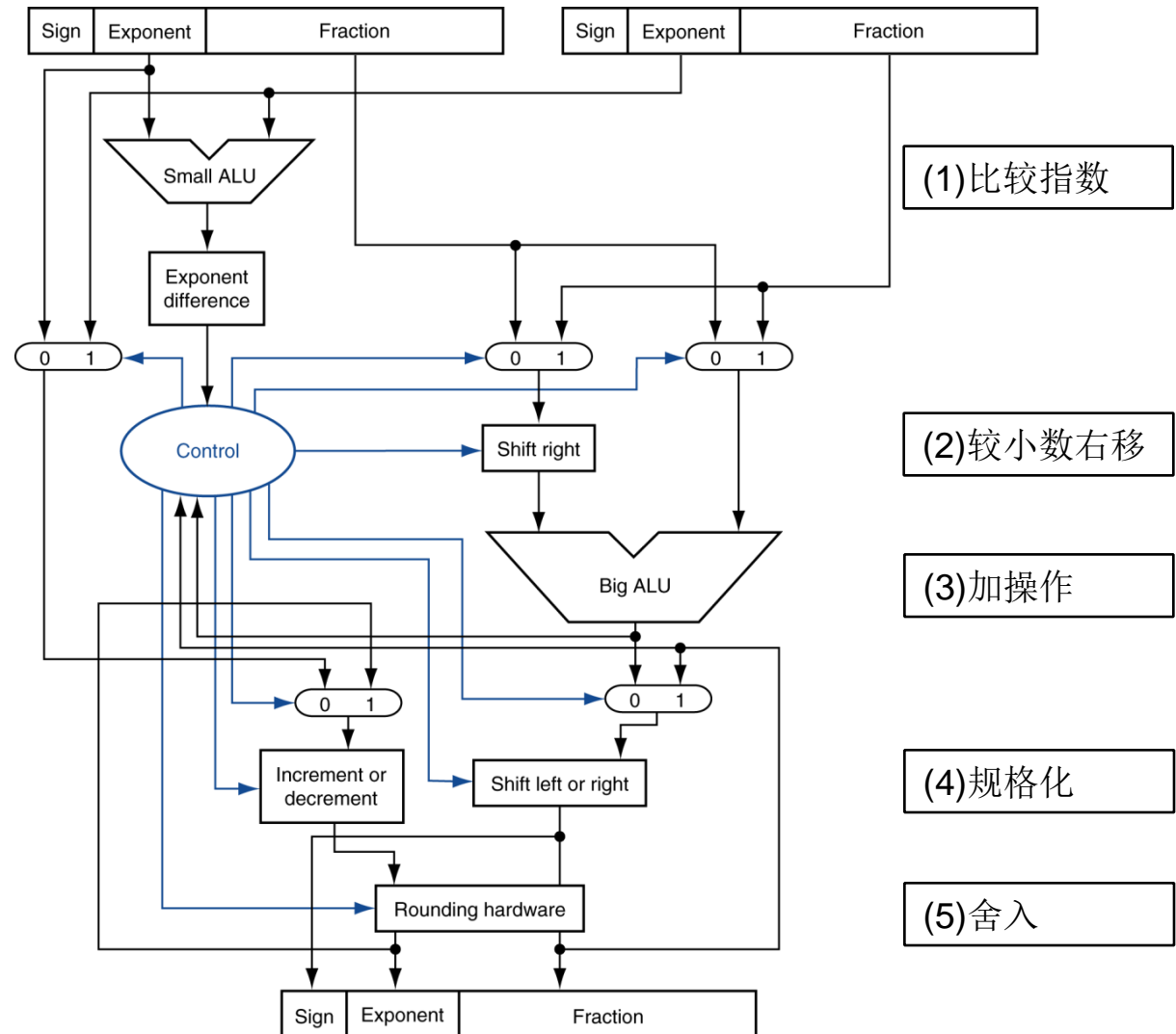
S = 1, 尾数 = 01000...00<sub>2</sub>

阶码 = 10000001<sub>2</sub> = 129

$$\begin{aligned} x &= (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)} \\ &= (-1) \times 1.25 \times 2^2 \\ &= -5.0 \end{aligned}$$

# Ch3

## 浮点加法



# Ch3

## ■ MIPS中的浮点指令

独立浮点寄存器

32 个单精度: \$f0, \$f1, ... \$f31

配对为双精度: \$f0/\$f1, \$f2/\$f3, ...

浮点指令只操作浮点寄存器

浮点数读取、存储指令

lwc1, ldc1, swc1, sdc1

单精度运算

add.s, sub.s, mul.s, div.s , add.s \$f0, \$f1, \$f6

双精度运算

add.d, sub.d, mul.d, div.d, mul.d \$f4, \$f4, \$f6

单精度和双精度的比较

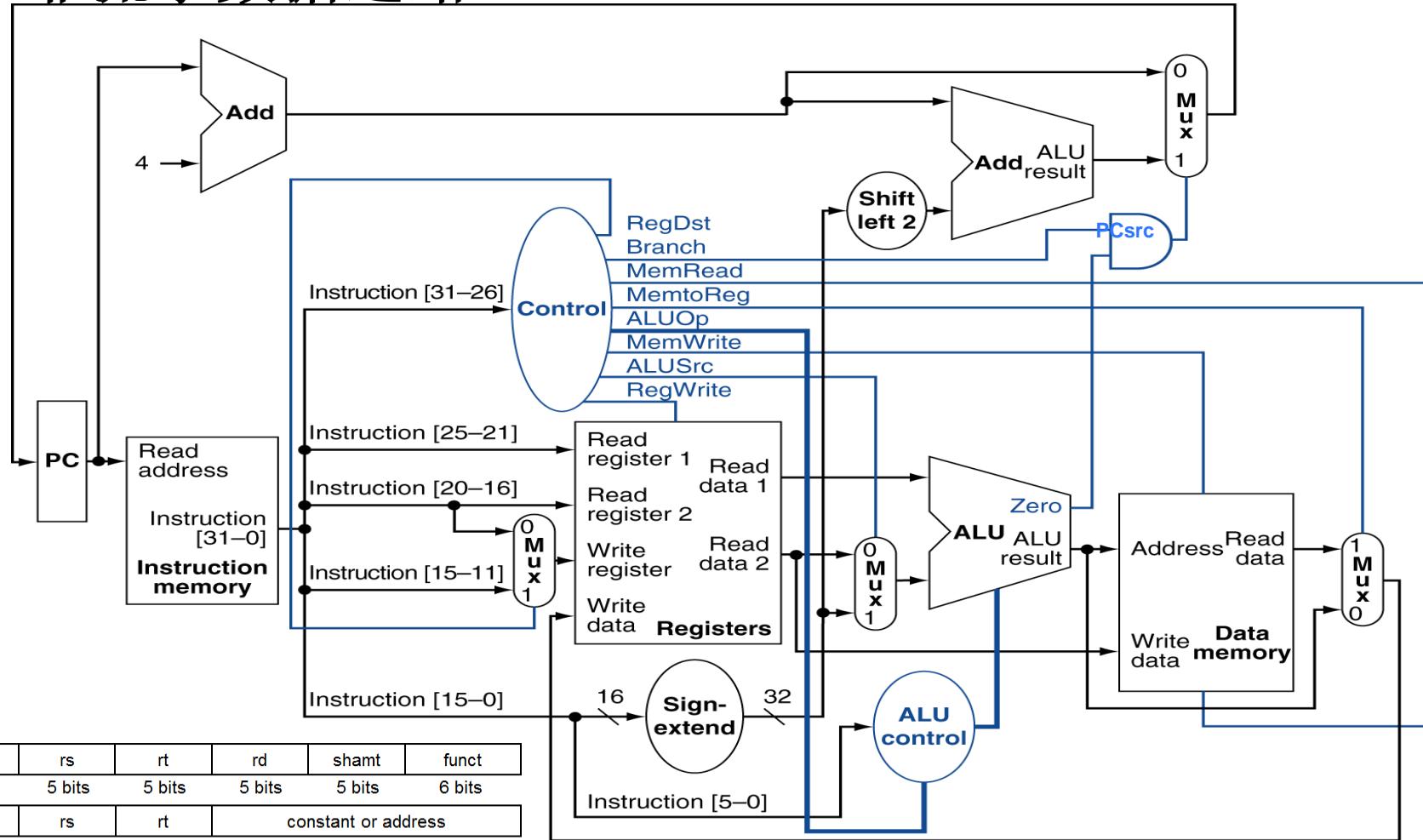
c.xx.s, c.xx.d (xx is eq, lt, le, ...), c.lt.s \$f3, \$f4

浮点条件代码之下的分支

bc1t, bc1f, bc1t TargetLabel

# Ch4

## 非流水数据通路



R	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I	op	rs	rt	constant or address		
	6 bits	5 bits	5 bits	16 bits		
J	op	constant or address				
	6 bits	26 bits				

# Ch4


## ■ 非流水数据通路的控制信号

		信号名	无效时的含义	有效时的含义
MUX 控制	}	RegDst	写寄存器的目标号来源于指令rt字段 (bits 20:16).	写寄存器的目标号来源于指令rd字段(bits 15:11).
		RegWrite	无	寄存器堆写使能
		ALUSrc	ALU第二个输入来源于寄存器堆的 第二个输出	ALU第二个输入来源于指令的低16位（目标地址的偏移量）
分支	}	Branch (PCSrc)	顺序执行，取 $PC + 4$ .	跳转，使用目标地址替代PC+4
		MemRead	无	数据存储器读使能
Mem 读写 控制	}	MemWrite	无	数据存储器写使能
		MemtoReg	写入寄存器的值来源于ALU.	写入寄存器的值来源于数据存储器
		ALUOP	ALU功能控制	

# Ch4

## ■ 非流水数据通路的控制信号

Instruction	RegDst	ALUSrc	Reg-Write	Memto-Reg	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	1	0	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	0	X	0	1	0	0	0
beq	X	0	0	X	0	0	1	0	1

The diagram shows four red brackets below the table, each grouping specific control signals. The first bracket, labeled 'MUX 控制', groups RegDst, ALUSrc, and Reg-Write. The second bracket, labeled 'Mem 读写控制', groups Memto-Reg, Mem-Read, and Mem-Write. The third bracket, labeled '分支', groups the Branch signal. The fourth bracket, labeled 'ALU 控制', groups ALUOp1 and ALUOp0.

MUX 控制

Mem 读写控制

分支

ALU 控制

# Ch4

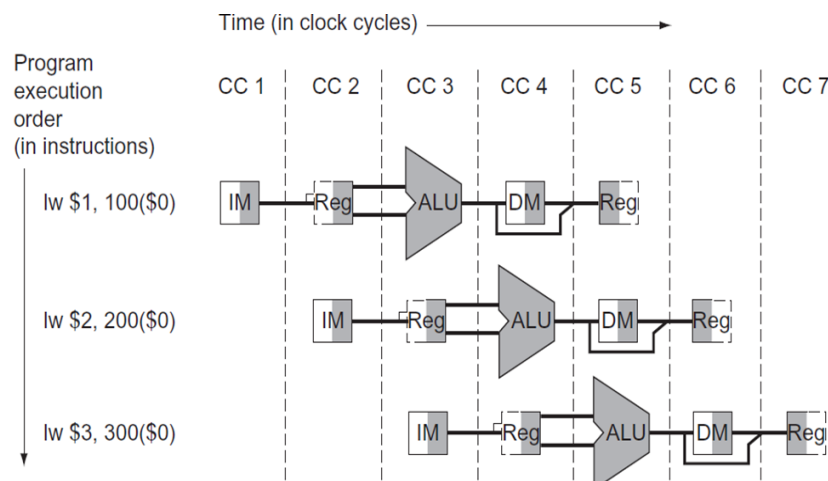
## ■ 流水线

5级, 每级完成一个操作步骤

1. **IF:** 从内存中取指令
2. **ID:** 指令解码& 读寄存器
3. **EX:** 执行运算或计算地址
4. **MEM:** 访问内存操作
5. **WB:** 将结果写回寄存器

## ■ MIPS ISA 设计原则 (也是优点)

- 所有指令**长度**都是**32-bits**等长
- 指令格式规整:
  - 解码的同时读取寄存器操作数
- 访存操作仅出现在**lw/st**指令中
  - 第3级EX可用于计算地址, 供第4级MEM使用。否则需要在EX和MEM之间增加1级地址计算
- 操作数内存地址对齐
  - 内存访问只需一个周期





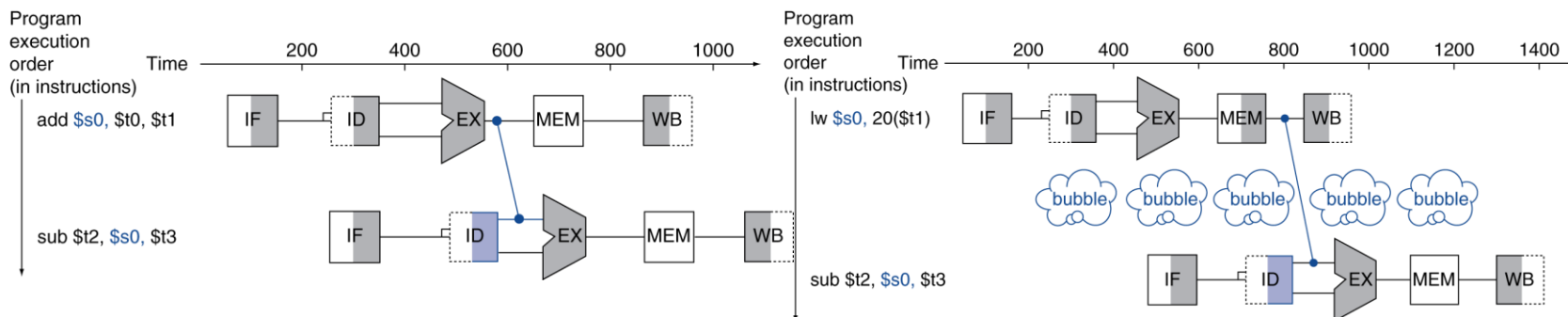
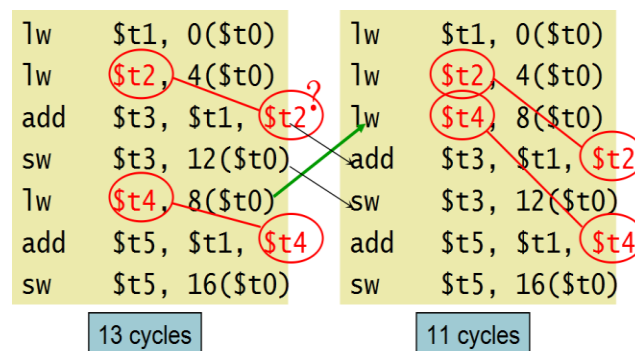
# Ch4

## 冒险

下1周期不能按时执行后1条指令（性能出现下降）

**结构冒险：**所需的部件忙，暂不可用  
分离独立的指令和数据内存

**数据冒险：**需要前面指令的计算结果  
前推/阻塞/指令调整



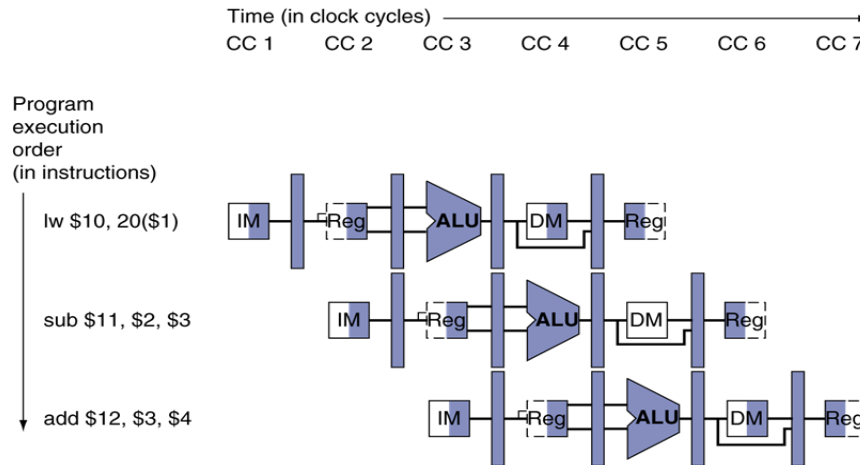
**控制冒险**（控制相关）

需要根据前面某条指令的结果来确定分支的选择执行

# Ch4

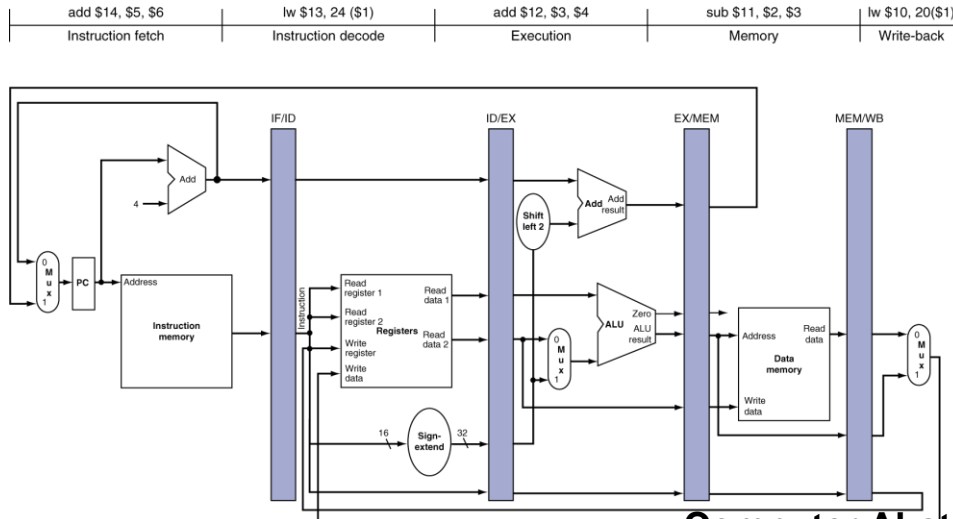
## 流水线寄存器：各级之间，记录前周期结果

多  
时  
钟  
周  
期



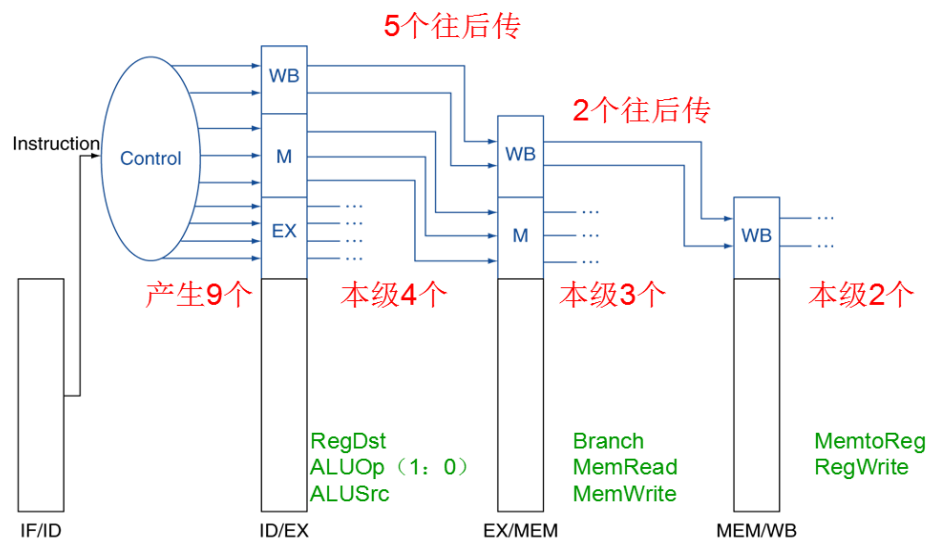
1个时钟周期内：  
前半->写入  
后半->读出

单  
时  
钟  
周  
期



# Ch4

## 流水线控制信号



Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

ID/EX

EX/MEM

MEM/WB

# Ch4

## 数据冒险

判定数据冒险条件（4种类型）

1a. EX/MEM. RegisterRd = ID/EX. RegisterRs

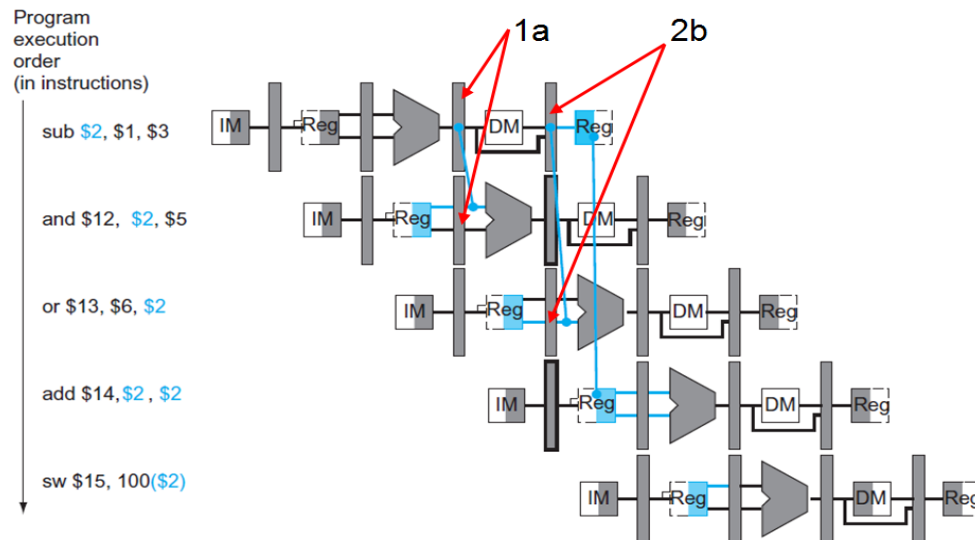
1b. EX/MEM. RegisterRd = ID/EX. RegisterRt

2a. MEM/WB. RegisterRd = ID/EX. RegisterRs

2b. MEM/WB. RegisterRd = ID/EX. RegisterRt

前推最早在EX级，因此旁路单元也放在该级

	Time (in clock cycles)								
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register \$2:	10	10	10	10	10/-20	-20	-20	-20	-20
Value of EX/MEM:	X	X	X	-20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	-20	X	X	X	X



# Ch4

## ■ 数据冒险：前推条件

- (1) 写寄存器:  $\text{RegWrite}=1$
- (2) 所写的寄存器不是0#寄存器:  $\text{RegisterRd} \neq 0$ ,
- (3) 流水线寄存器中读/写寄存器相同

EX/MEM 与 ID/EX: EX型 (1)

MEM/WB与 ID/EX: MEM型 (2)

- (4) 只有EX冒险不成立时, 才对MEM冒险前推

**EX 冒险** (rs/rt在EX/MEM检测到相应的rd)

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

ForwardA = 10

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

ForwardB = 10

**MEM 冒险** ( rs/rt在MEM/WB检测到相应的rd )

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

ForwardA = 01

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd  $\neq$  0)  
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd  $\neq$  0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

ForwardB = 01

# Ch4

## ■ 取数-使用：冒险与阻塞

取数指令

在ID阶段检测寄存器的使用

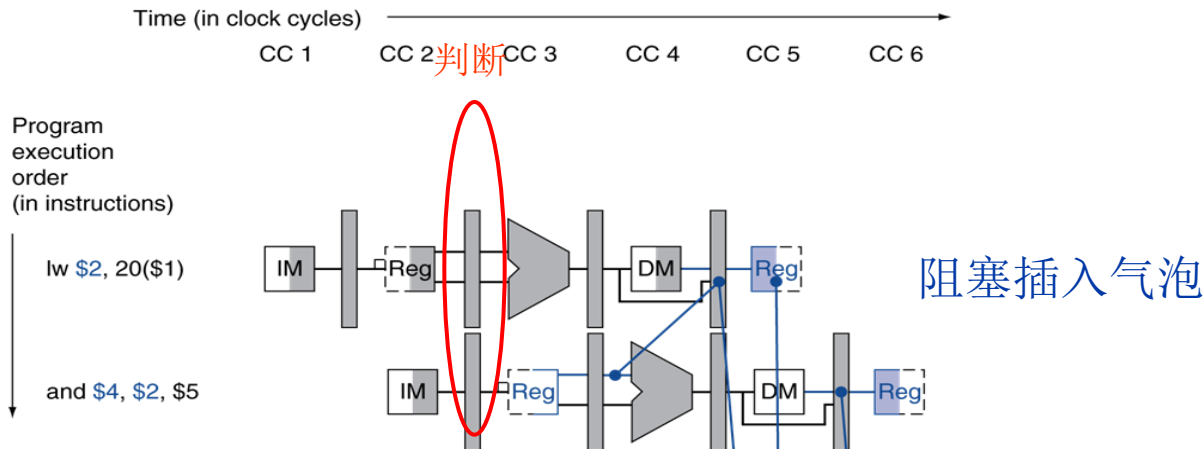
“取数-使用”冒险判定

ID/EX. MemRead and

((ID/EX. RegisterRt = IF/ID. RegisterRs)

or (ID/EX. RegisterRt = IF/ID. RegisterRt))

检测在ID级，因此冒险  
检测单元也放在该级



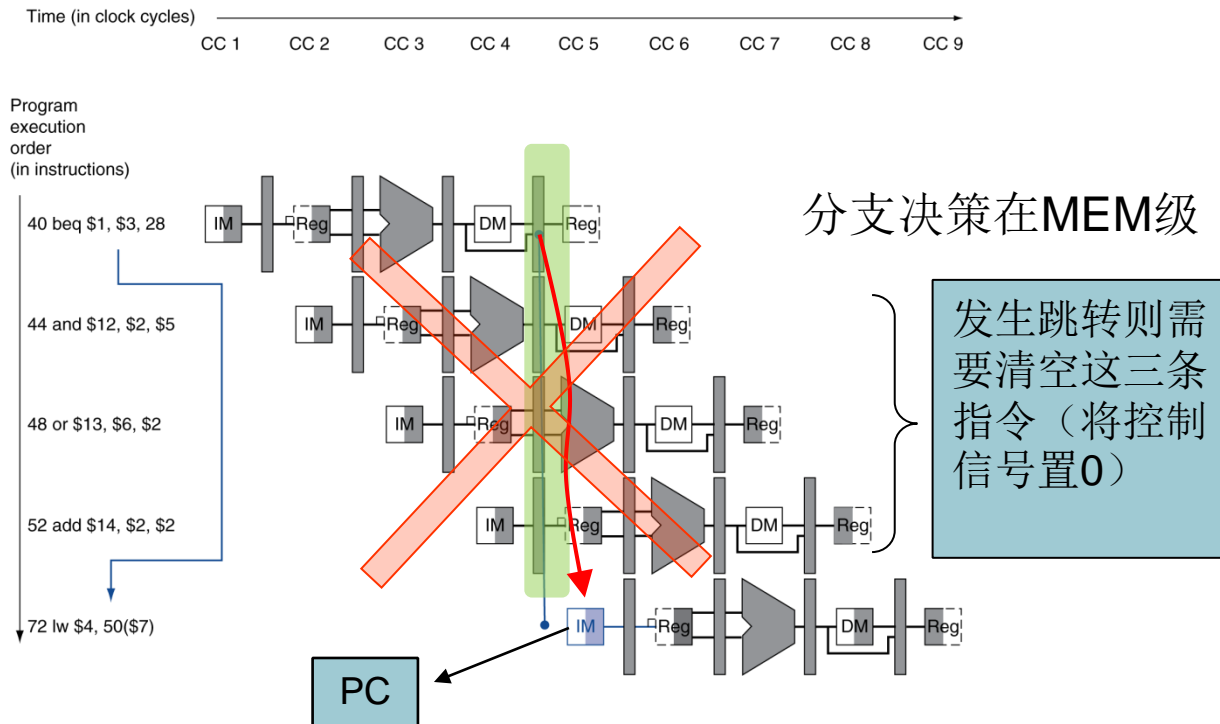
# Ch4

## 控制冒险/分支冒险

分支决策从EX提前到ID级

目标地址计算（加法器）

比较器（两个寄存器异或、然后逐位或）



## ■ 异常与中断

打断原来正常执行的指令流

异常Exception: 源于CPU内部事件

例如, 无效指令, 溢出, 系统调用指令, ...

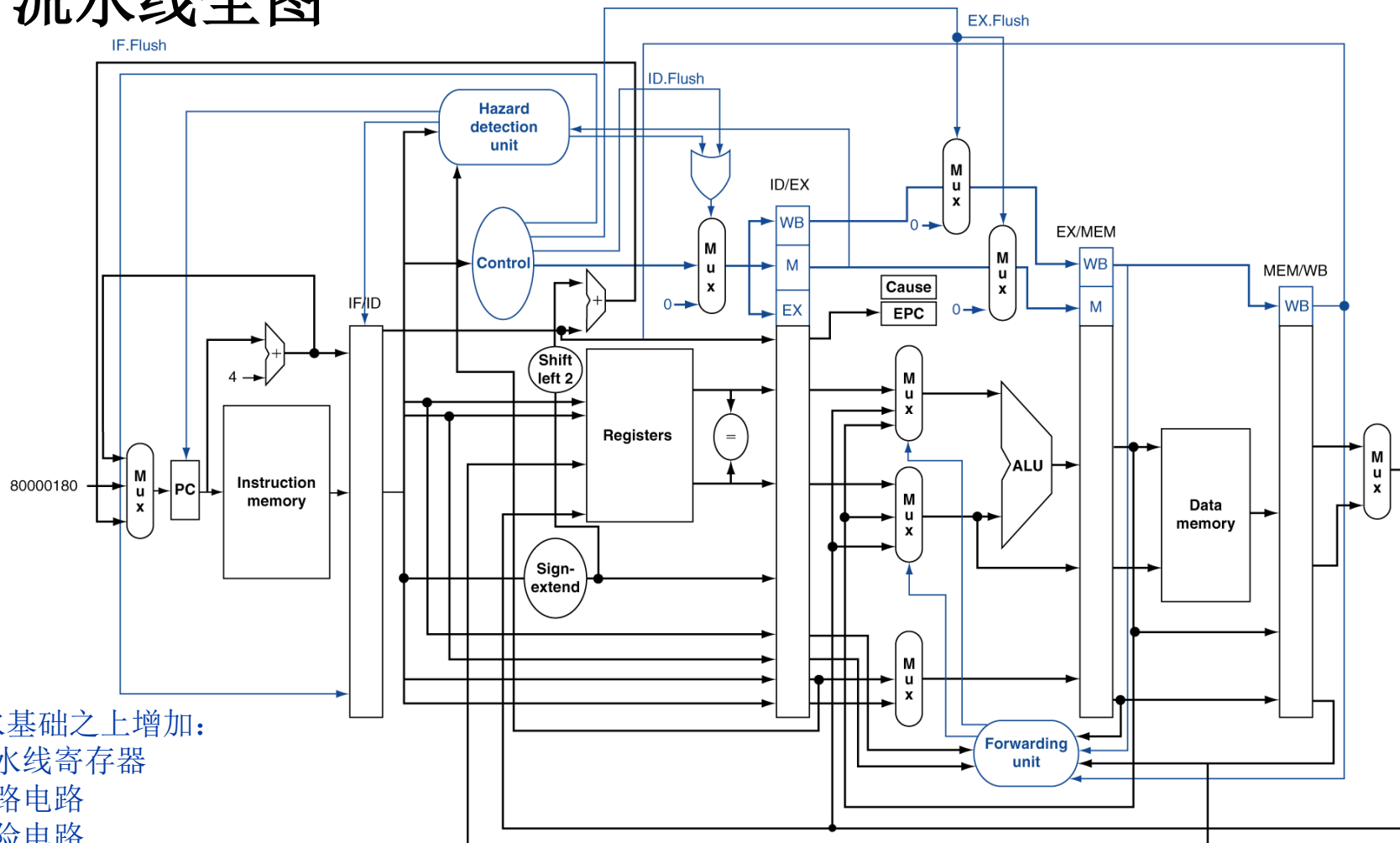
中断: 源于外部I/O 控制器

跳转到异常处理入口代码, 地址: 8000 00180



# Ch4

## ■ 流水线全图



非流水基础之上增加:

- 1) 流水线寄存器
- 2) 旁路电路
- 3) 冒险电路
- 4) 分支预测及撤销
- 5) 异常处理

# Ch5

- **局部性**：在任何时间内，程序访问的只是地址空间相对较小的一部分内容
  - **时间局部性 (Temporal locality)**：如果某个数据项被访问，在不久的将来它可能再次被访问
  - **空间局部性 (Spatial locality)**：如果某个数据项被访问，与它地址相邻的数据项可能很快也将被访问
- **存储技术**
  - 缓存 Static RAM (SRAM), 0.5ns – 2.5ns
  - 主存 Dynamic RAM (DRAM), 50ns – 70ns
  - 磁盘, 5ms – 20ms
- **关键问题**
  - 数据在不在
  - 怎么找到

# Ch5

## ■ 缓存Cache

### 直接映射

块地址 = 地址 / 块的大小

块号(位置/索引)=(块地址) mod  
(cache中的总块数)

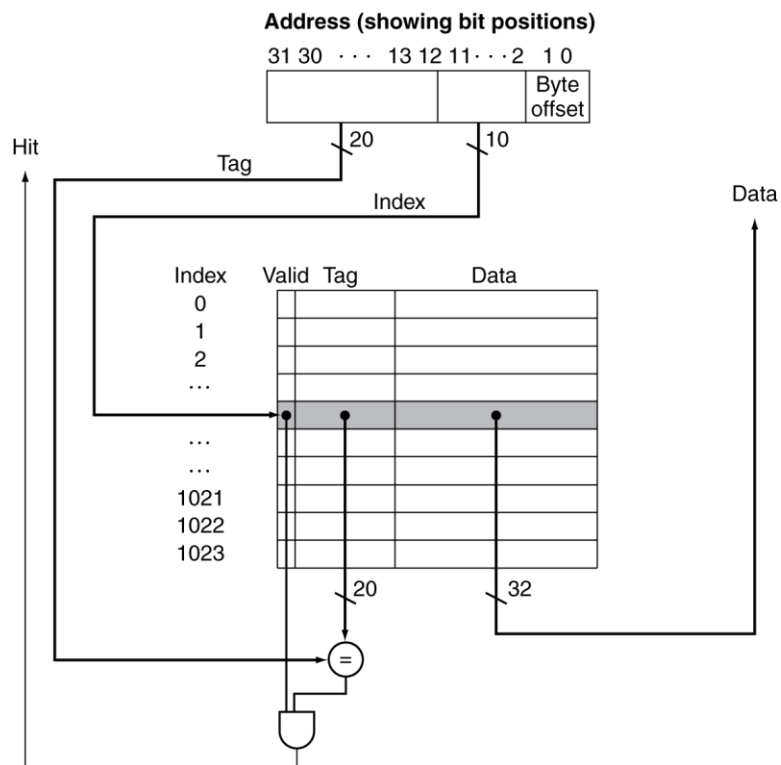
Index 索引	V 有效位	Tag 标记	Data 数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache 命中：则CPU正常处理

Cache 缺失：造成CPU流水线阻塞，从下一级存储器中取数据块

指令cache 缺失：重新进行指令获取

数据cache 缺失：完成数据的访问



# Ch5

## ■ 缓存Cache—量化计算

I-cache 失效率 = 2%， D-cache 失效率 = 4%

缺失代价 = 100 个时钟周期

理想 CPI = 2

Load & stores 占全部36%的指令  
则：

指令缺失的时钟周期数：

I-cache:  $0.02 \times 100 = 2$

D-cache:  $0.36 \times 0.04 \times 100 = 1.44$

实际 CPI =  $2 + 2 + 1.44 = 5.44$

理想 CPU 是  $5.44/2 = 2.72$  倍

平均存储器访问时间Average memory access time (AMAT)

AMAT = 命中时间+ 缺失率  $\times$  缺失代价

CPU时钟周期时间为1ns

命中时间为1个时钟周期, 缺失代价为20 时钟周期

I-cache的缺失率为 5%

AMAT =  $1 + 0.05 \times 20 = 2\text{ns}$

# Ch5

## ■ 缓存Cache—相联

分组，记录标号+数据

组号=(块号) modulo (cache中的组数)

拥有8个块的cache的不同配置：

直接映射

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

2路组相联

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

4路组相联

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

全相联

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

最近最少使用  
(Least-recently used, LRU)

# Ch5

## ■ 缓存Cache—多级

L1-cache: 直接连接 CPU, 容量小, 但速度快

L-2 cache: 处理L1-cache 的失效, 容量较大, 速度较慢, 但仍比主存要快

CPU 基准 CPI = 1, 时钟频率 = 4GHz

L1-cache缺失率/指令 = 2%

主存访问时间 = 100ns

L2-cache访问时间 = 5ns

L2-cache对于主存的整体失效率 = 0.5%

仅有L1-cache

缺失代价 =  $100\text{ns}/0.25\text{ns} = 400$  时钟周期

有效 CPI =  $1 + 0.02 \times 400 = 9$

L1失效& L2 命中

缺失代价 =  $5\text{ns}/0.25\text{ns} = 20$  时钟周期

L1失效& L2 失效

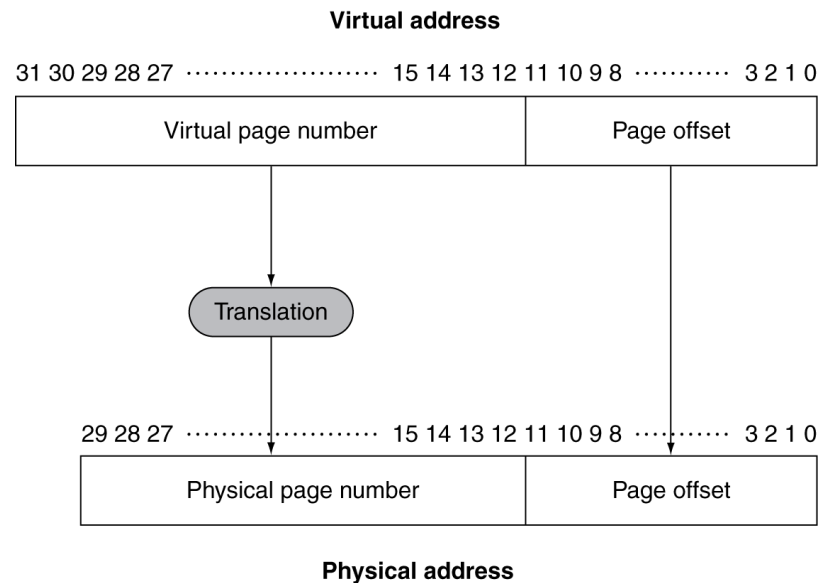
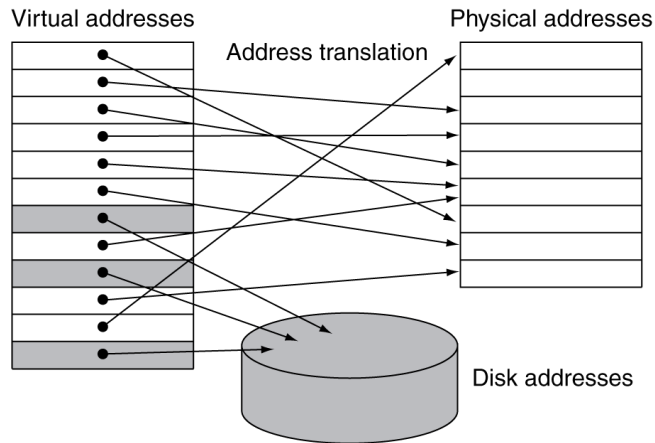
CPI =  $1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$

性能比 =  $9/3.4 = 2.6$

# Ch5

## ■ 虚拟存储器—磁盘的Cache

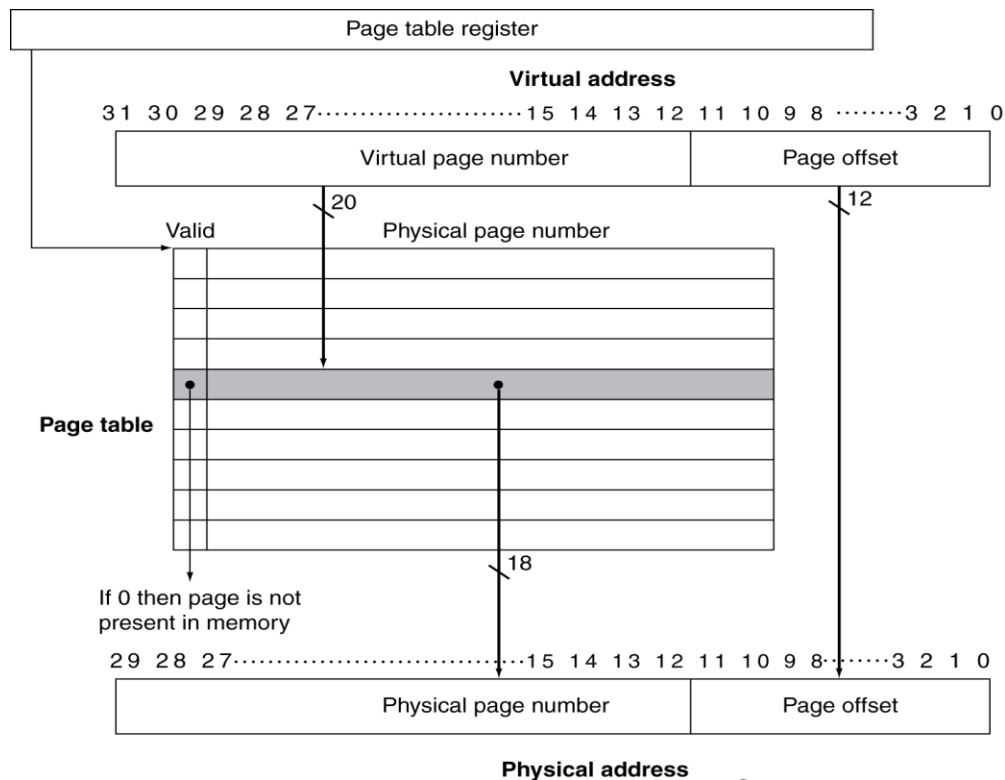
- 程序共享地址空间，用来保存经常访问的代码和数据
- 扩展存储空间



# Ch5

## 虚拟存储器—页表

- 保存虚拟地址和物理地址之间转换关系
- 使用虚页号来索引
- CPU中的页表寄存器指向页表位置

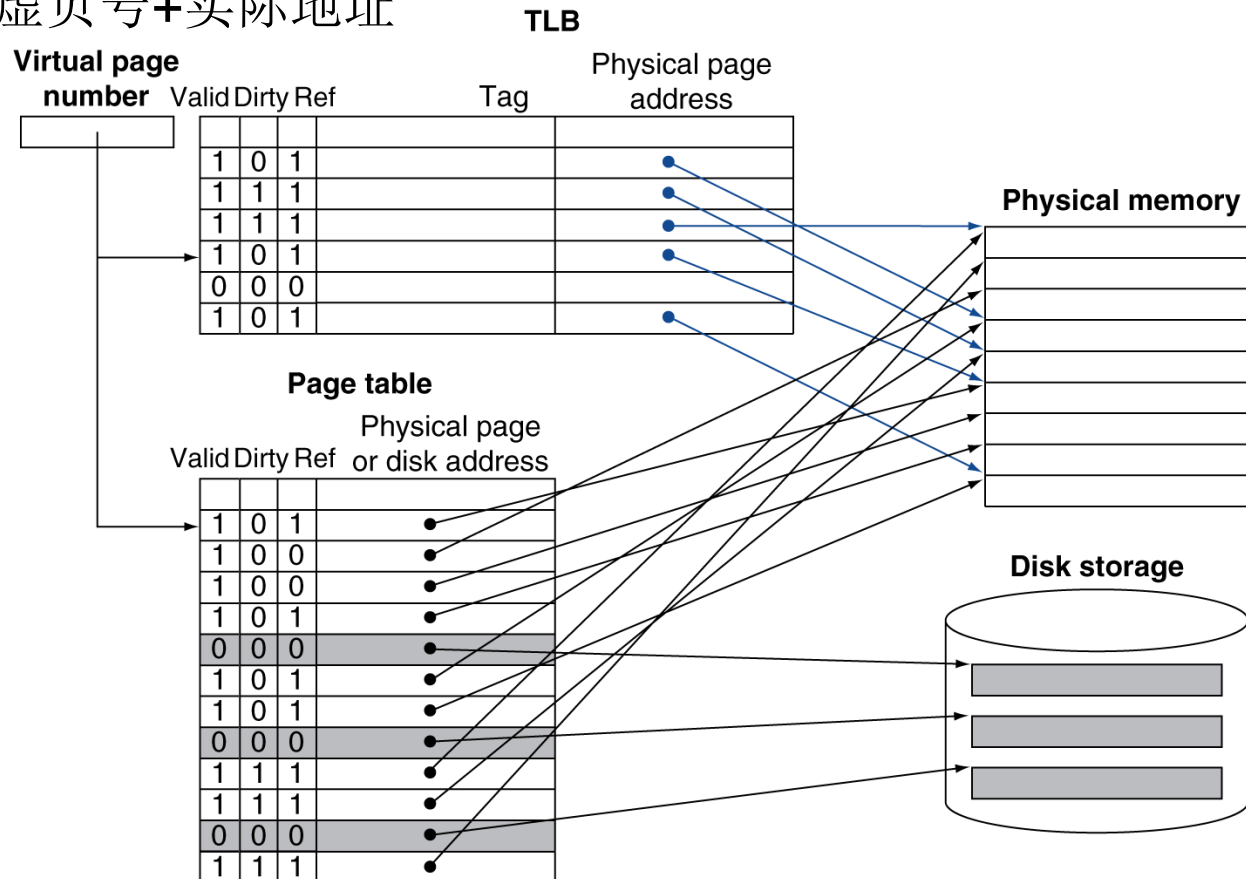




# Ch5

## 虚拟存储器—TLB

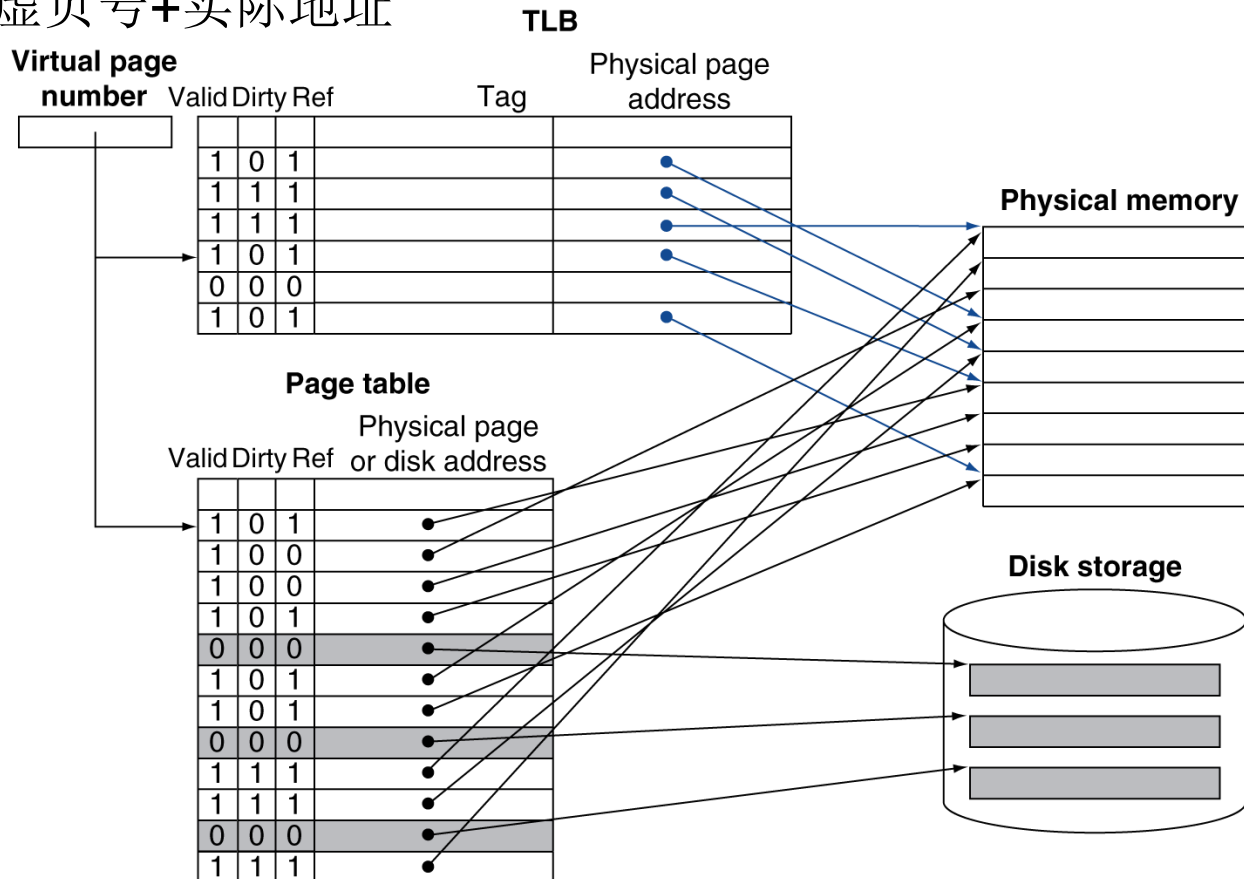
- 页表的缓存
- 虚页号+实际地址



# Ch5

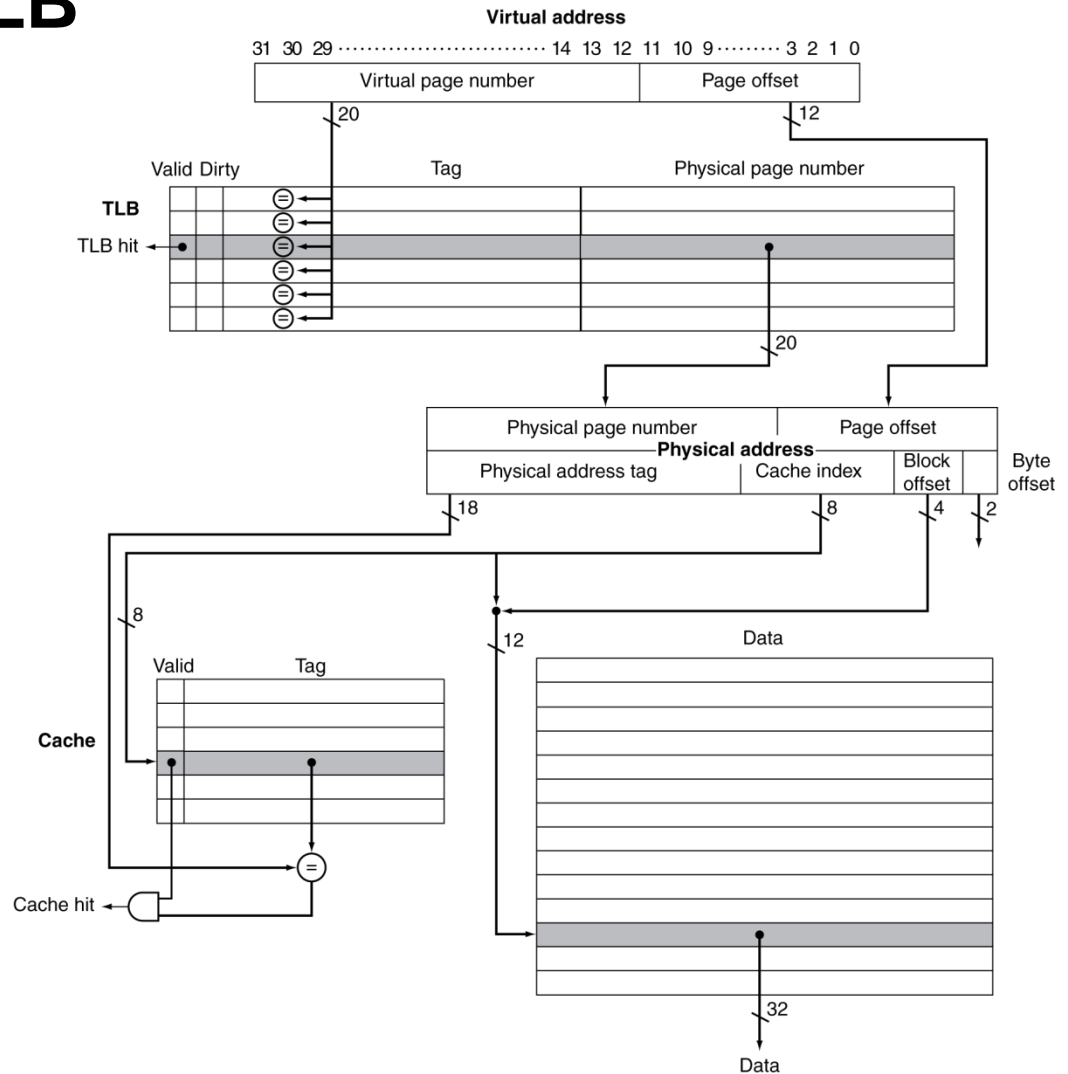
## 虚拟存储器—TLB

- 页表的缓存
- 虚页号+实际地址



# Ch5

## ■ 虚拟存储器—TLB



# Ch5

## 存储器工作流程

### CPU访存过程

