

第四章 处理器体系结构

——顺序执行的处理器

教师：郑贵滨

计算机科学与技术学院

哈尔滨工业大学

Y86-64 指令集 1

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Y86-64 指令集 2

字节	0	1	2	3	4	5	
halt	0	0					
nop	1	0					
cmovXX rA, rB	2	fn	rA	rB			rrmovq 2 0
irmovq V, rB	3	0	F	rB			cmovle 2 1
rmmovq rA, D(rB)	4	0	rA	rB			cmovl 2 2
rrmovq D(rB), rA	5	0	rA	rB			cmove 2 3
OPq rA, rB	6	fn	rA	rB			cmovne 2 4
jXX Dest	7	fn					cmovge 2 5
call Dest	8	0					cmovg 2 6
ret	9	0					
pushq rA	A	0	rA	F			
popq rA	B	0	rA	F			

Y86-64 指令集 3

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB						
rmmovq rA, D(rB)	4	0	rA	rB						
mrmmovq D(rB), rA	5	0	rA	rB						
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

addq 6 0
subq 6 1
andq 6 2
xorq 6 3



Y86-64 指令集 4

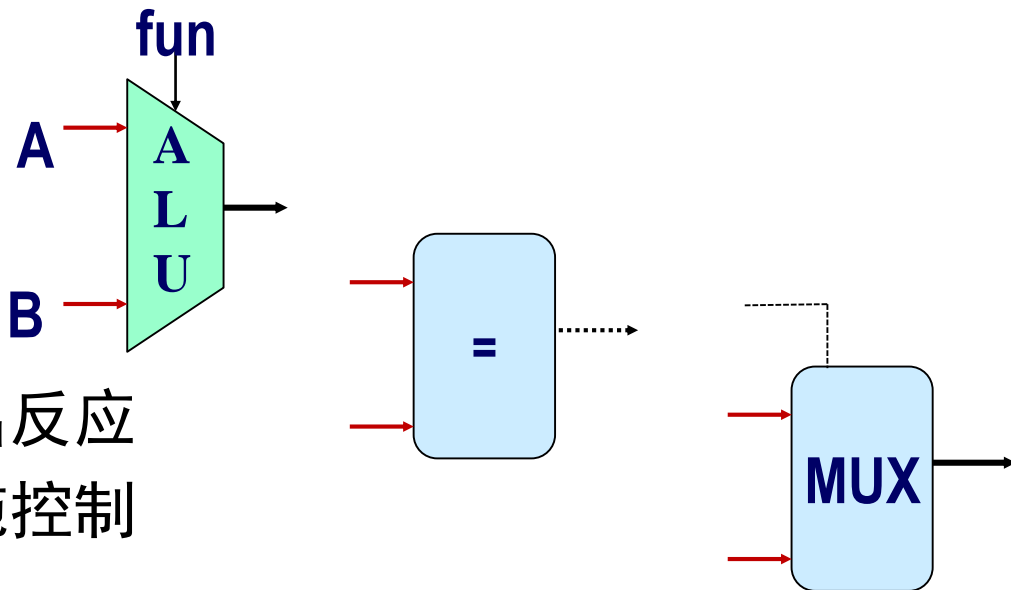
字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB						
rmmovq rA, D(rB)	4	0	rA	rB						
mrmmovq D(rB), rA	5	0	rA	rB						
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn								Dest
call Dest	8	0								Dest
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

jmp	7	0
jle	7	1
jl	7	2
je	7	3
jne	7	4
jge	7	5
jg	7	6

设计硬件模块

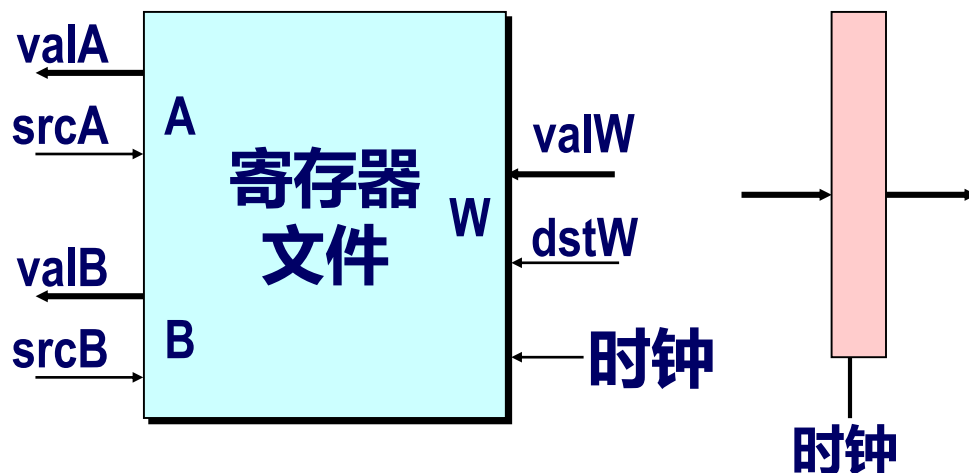
■ 组合逻辑

- 计算输入的布尔函数
- 对输入的变化持续做出反应
- 对数据做出操作并实施控制



■ 存储要素

- 存储字节/位
- 可寻址的内存
- 不可寻址的寄存器
- 时钟上升沿触发写入



硬件控制语言

- 非常简单的硬件描述语言
- 只能表达有限的硬件操作：我们要探索和改进的部分

■ 数据类型

- bool:布尔类型
 - a, b, c, ...
- int:字类型
 - A, B, C, ...
 - 不指定字长---可以是字节, 32-bit的字,等等

■ 声明

- `bool a = 布尔表达式 ;`
- `int A = 整数表达式 ;`

HCL操作

通过返回值的类型分类

■ 布尔表达式

■ 逻辑操作

- $a \ \&\& \ b, a \ || \ b, !a$

■ 字比较

- $A == B, A != B, A < B, A <= B, A >= B, A > B$

■ 集合成员

- $A \text{ in } \{ B, C, D \}$
 - 与 $A == B \ || \ A == C \ || \ A == D$ 一样

■ 字表达式

■ 情况表达式

- $[a : A; b : B; c : C]$
- 按顺序评估测试表达式 a, b, c, \dots
- 返回和首次成功测试对应的字表达式 A, B, C, \dots

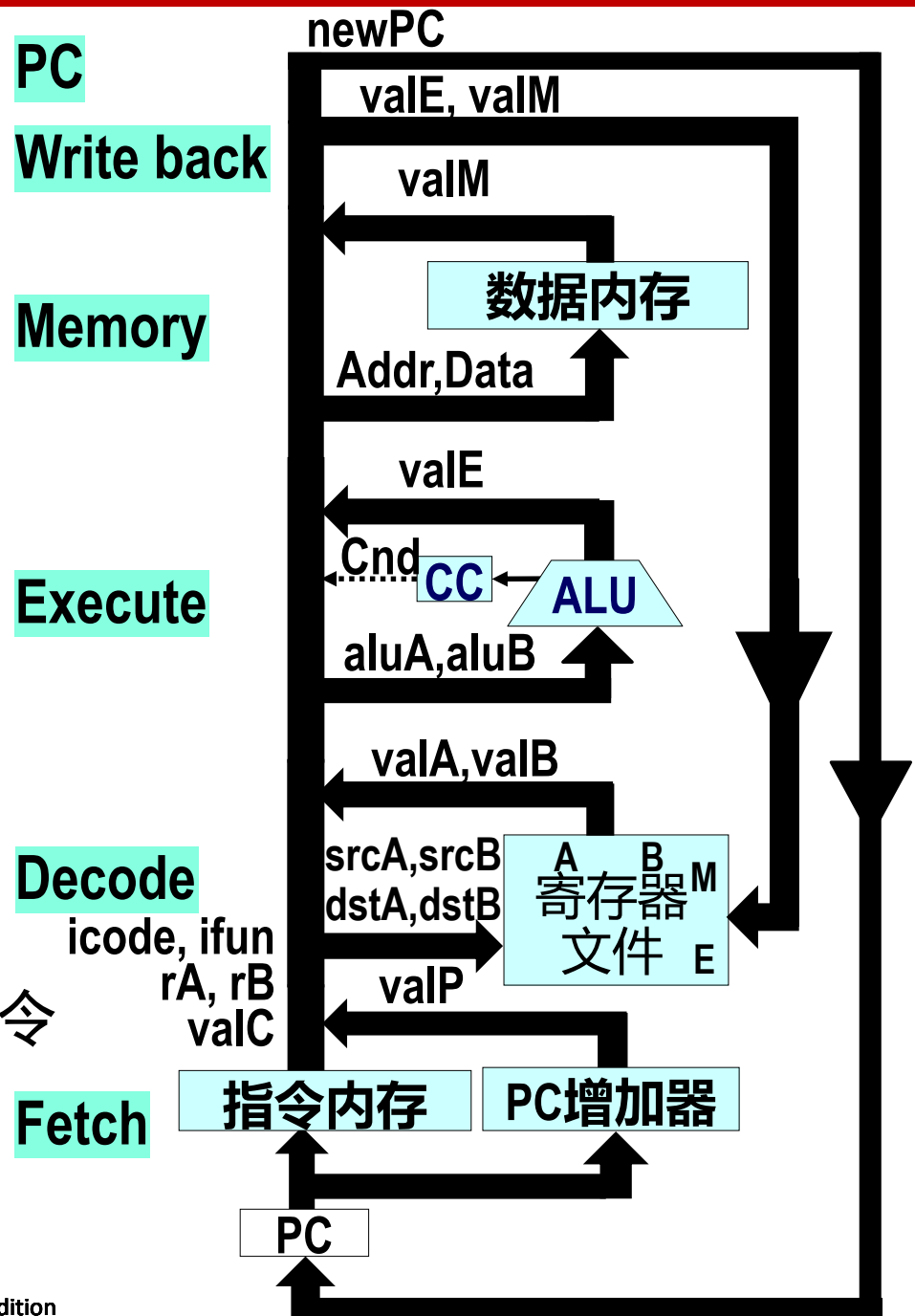
SEQ 硬件结构

■ 状态

- 程序计数器 (PC)
- 条件码寄存器 (CC)
- 寄存器文件
- 内存
 - 访问相同的存储空间
 - 数据: 读/写的程序数据
 - 指令: 读取指令

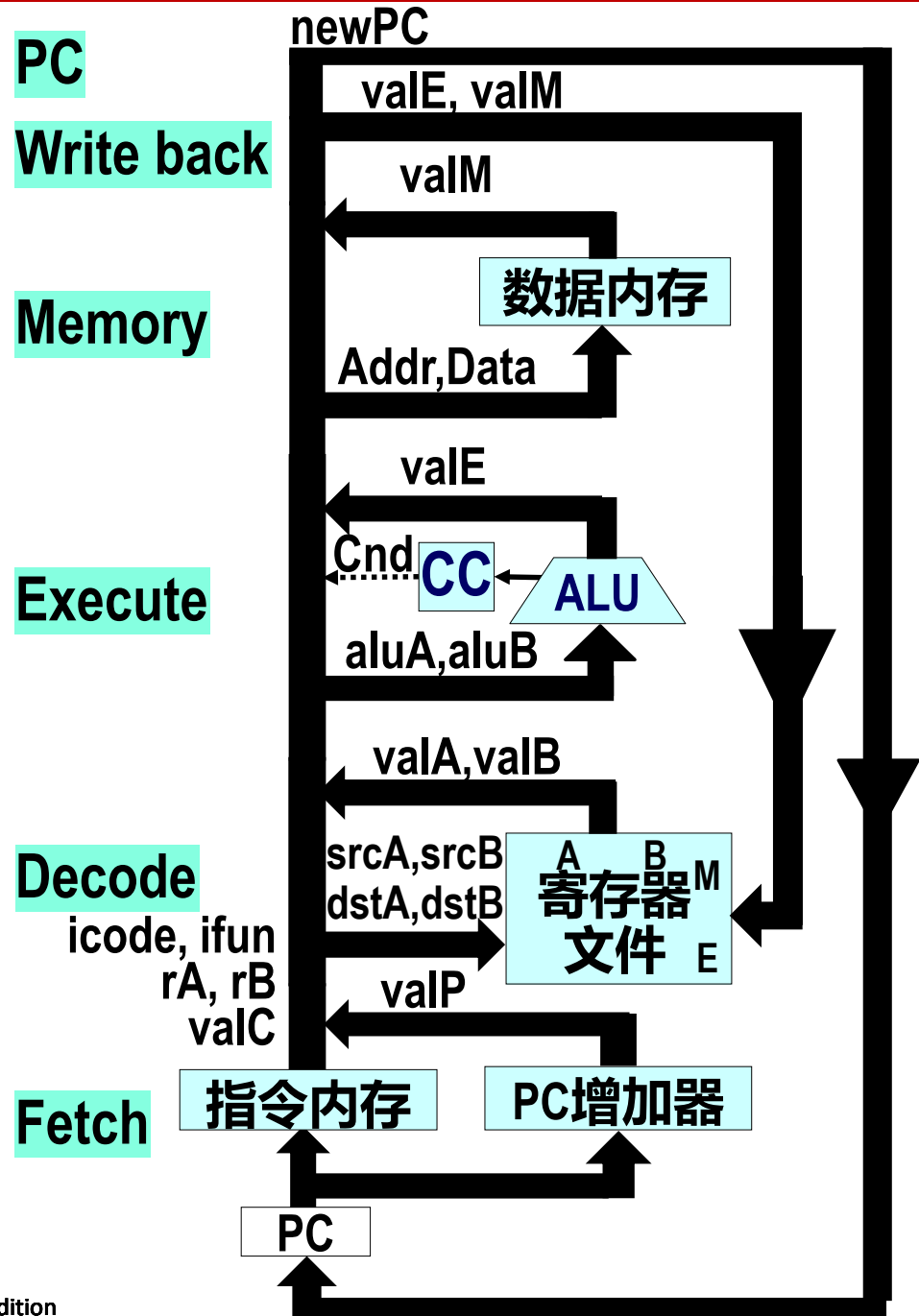
■ 指令流

- 从PC指定的地址读取指令
- 分多个**阶段**执行
- 更新PC

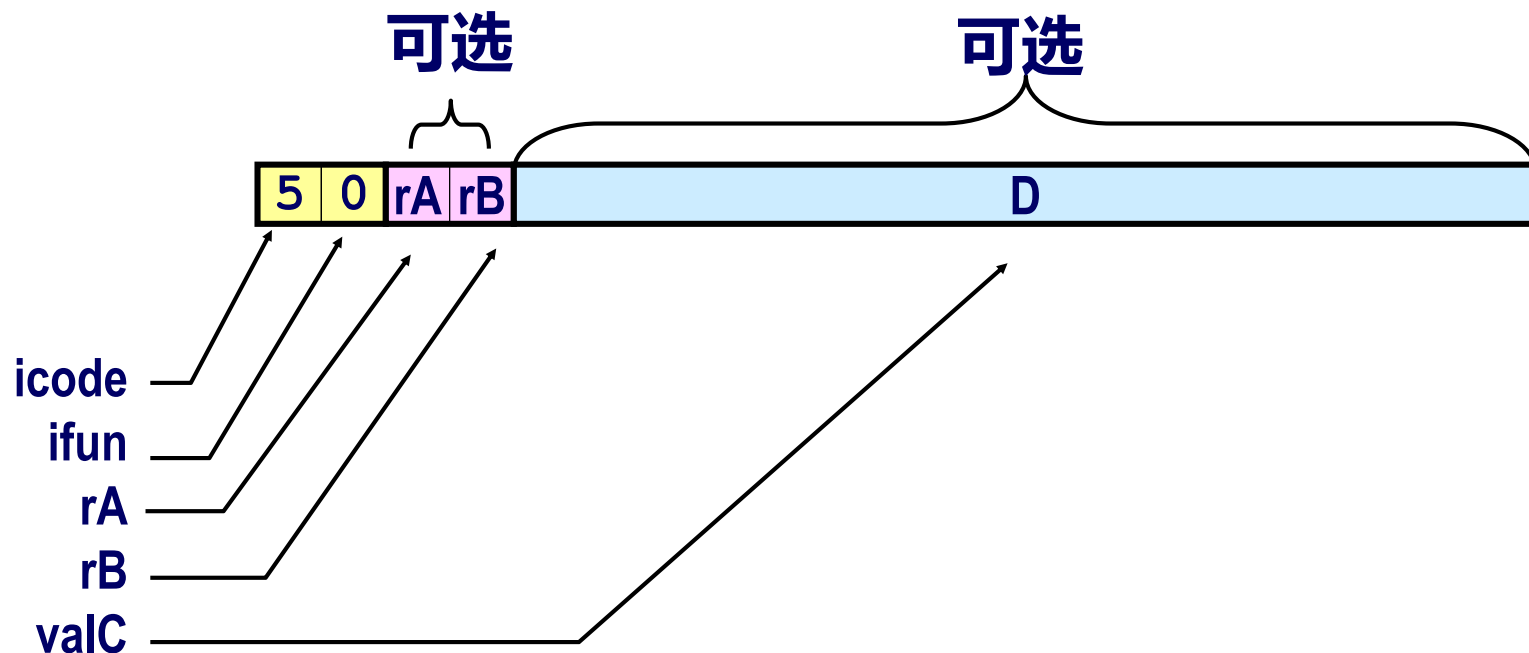


SEQ各阶段

- **取指 - Fetch**
 - 从指令存储器读取指令
- **译码 - Decode**
 - 读程序寄存器
- **执行 - Execute**
 - 计算数值或地址
- **访存 - Memory**
 - 读或写数据
- **写回 - Write back**
 - 写程序寄存器
- **PC更新- PC update**
 - 更新程序计数器



指令译码



■ 指令格式

- 指令字节 *icode:ifun*
- 可选的寄存器字节 *rA:rB*
- 可选的常数字 *valC*

执行 算术/逻辑操作

OP_q rA, rB



均为整数操作

■取指

- 读两个字节

■译码

- 读操作数寄存器

■执行

- 执行操作
- 设置条件码

■访存

- 无操作

■写回

- 更新寄存器

■更新PC

- $PC + 2$

各阶段的运算: 算术/逻辑操作

	OPq rA, rB	
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	读指令字节 读寄存器字节
译码	$\text{valP} \leftarrow \text{PC}+2$ $\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	计算下一个PC 读操作数A 读操作数B
执行	$\text{valE} \leftarrow \text{valB OP valA}$ Set CC	执行ALU的操作 设置条件码寄存器
访存		
写回	$R[\text{rB}] \leftarrow \text{valE}$	结果写回
更新PC	$\text{PC} \leftarrow \text{valP}$	更新PC

- 把指令的执行过程表示为特殊的阶段序列
- 所有的指令都使用相同的格式来表示

执行 `rmmovq` 指令



■ 取指

- 读10个字节

■ 译码

- 读(2个)操作数寄存器

■ 执行

- 计算有效地址

■ 访存

- 写到内存

■ 写回

- 无操作

■ 更新PC

- $PC + 10$

各阶段的运算: `rmmovq`

	<code>rmmovq rA, D(rB)</code>	
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC}+10$	读取指令字节 读寄存器字节 读偏移量D 计算下一个PC
译码	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	读操作数A 读操作数B
执行	$\text{valE} \leftarrow \text{valB} + \text{valC}$	计算有效地址
访存	$M_8[\text{valE}] \leftarrow \text{valA}$	把数值写入内存
写回		
更新PC	$\text{PC} \leftarrow \text{valP}$	更新PC

- 利用ALU计算内存的有效地址

执行 popq



■取指

- 读两个字节

■译码

- 读栈指针(RSP)

■执行

- 栈指针加8

■访存

- 从栈指针旧值(没有加8的)所指内存处读取

■写回

- 更新栈指针
- 结果写到寄存器(rA)

■PC更新

- PC+2

各阶段的运算: popq

	popq rA	
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	读指令字节 读寄存器字节
译码	$\text{valP} \leftarrow \text{PC}+2$ $\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	计算下一个PC 读栈指针 读栈指针
执行	$\text{valE} \leftarrow \text{valB} + 8$	栈指针加8
访存	$\text{valM} \leftarrow M_8[\text{valA}]$	从栈里读数据
写回	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	更新栈指针 结果写回
更新PC	$\text{PC} \leftarrow \text{valP}$	更新PC

- 利用ALU来增加栈指针
- 必须更新两个寄存器
 - 弹出的数据存到rA
 - 新的栈指针值存到rsp

执行条件传送指令

`cmovXX rA, rB`

2

fn

rA

rB

■取指

- 读2个字节

■译码

- 读操作数寄存器

■执行

- 如果条件不成立, 则把目的寄存器设为0xF

■访存

- 无操作

■写回

- 更新寄存器(或无操作)

■更新PC

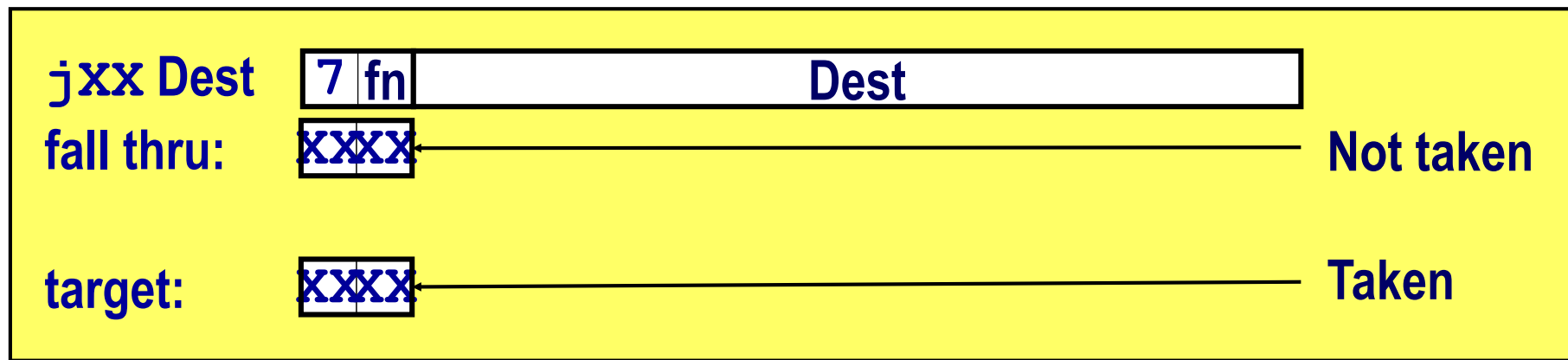
- PC+2

各阶段的运算: Cond. Move

	<code>cmovXX rA, rB</code>	
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	读指令字节 读寄存器字节
	$\text{valP} \leftarrow \text{PC}+2$	计算下一个PC
译码	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow 0$	读操作数A
执行	$\text{valE} \leftarrow \text{valB} + \text{valA}$ $\text{If ! Cond(CC,ifun) rB} \leftarrow 0xF$	利用ALU传递valA (阻止寄存器更新)
访存		
写回	$R[\text{rB}] \leftarrow \text{valE}$	结果写回
更新PC	$\text{PC} \leftarrow \text{valP}$	更新PC

- 读rA寄存器并通过ALU向后传递
- 通过将目的寄存器设为0xF来取消传送
 - 如果条件码和传送条件表明无需传送数据

执行跳转指令——Jmps



■取指

- 读9个字节
- PC+9

■译码

- 无操作

■执行

- 根据跳转条件和条件码来决定是否选择分支

■访存

- 无操作

■写回

- 无操作

■更新PC

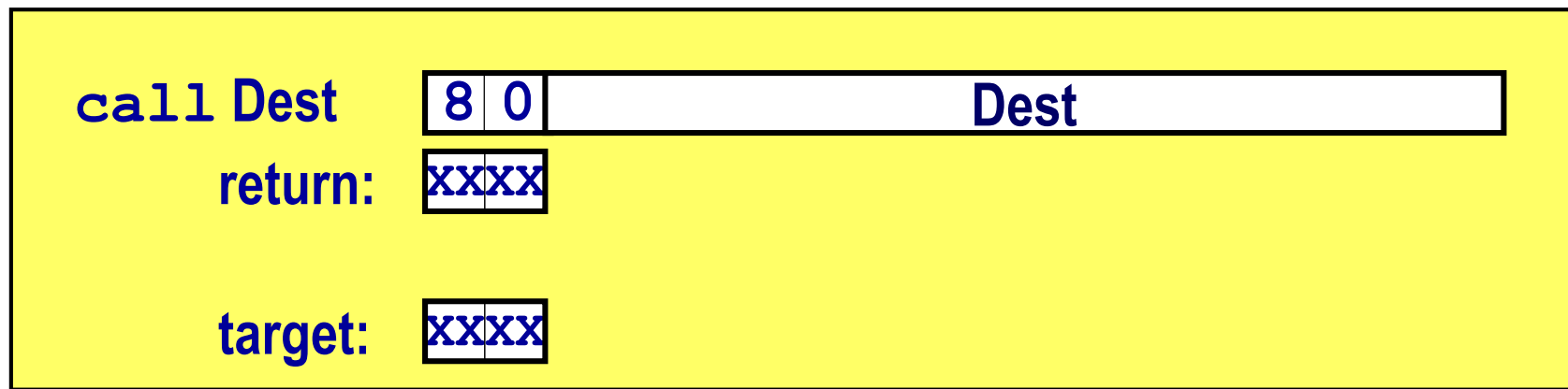
- 如选择分支，将Dest设为PC的值；如不选择分支，则PC设为增加之后的PC值

各阶段的运算: Jumps

	jXX Dest	
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+9$	读指令字节 读目的地址 Fall through address
译码		
执行	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	是否选择分支
访存		
写回		
更新PC	$\text{PC} \leftarrow \text{Cnd} ? \text{valC} : \text{valP}$	更新PC

- 计算两个地址
- 根据条件码和分支条件作出选择

执行 call 指令



■ 取指

- 读9个字节
- PC+9

■ 译码

- 读栈指针

■ 执行

- 栈指针减8

■ 访存

- 把增加后的PC值写到新的栈指针指向的位置

■ 写回

- 更新栈指针

■ 更新PC

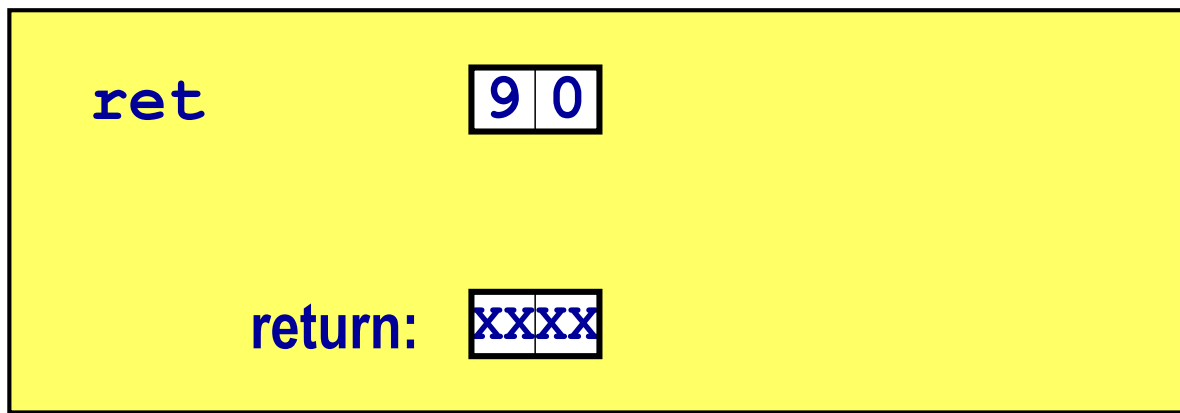
- PC设为Dest(目的地址)

各阶段的运算: call

	call Dest	
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+9$	读指令字节 读目的地址 计算返回指针
译码	$\text{valB} \leftarrow R[\%rsp]$	读栈指针
执行	$\text{valE} \leftarrow \text{valB} + -8$	栈指针减8
访存	$M_8[\text{valE}] \leftarrow \text{valP}$	返回值进栈
写回	$R[\%rsp] \leftarrow \text{valE}$	更新栈指针
PC更新	$\text{PC} \leftarrow \text{valC}$	PC指向目的地址

- 利用ALU减少栈指针
- 存储增加后的PC

执行 ret 指令



■取指

- 读一个字节

■译码

- 读栈指针

■执行

- 栈指针加8

■访存

- 通过原栈指针读取返回地址

■写回

- 更新栈指针

■更新PC

- PC指向返回地址

各阶段的运算: ret

ret		
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	读指令字节
译码	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$	读操作数栈指针 读操作数栈指针
执行	$\text{valE} \leftarrow \text{valB} + 8$	栈指针增加
访存	$\text{valM} \leftarrow M_8[\text{valA}]$	读返回地址
写回	$R[\%rsp] \leftarrow \text{valE}$	更新栈指针
更新PC	$\text{PC} \leftarrow \text{valM}$	PC指向返回地址

- 利用ALU增加栈指针的值
- 从内存中读取返回地址

各阶段的运算

		OPq rA, rB
取指	icode,ifun	icode:ifun $\leftarrow M_1[PC]$
	rA,rB	rA:rB $\leftarrow M_1[PC+1]$
	valC	
	valP	valP $\leftarrow PC+2$
译码	valA, srcA	valA $\leftarrow R[rA]$
	valB, srcB	valB $\leftarrow R[rB]$
执行	valE	valE $\leftarrow valB \text{ OP } valA$
	Cond code	Set CC
访存	valM	
写回	dstE	R[rB] $\leftarrow valE$
	dstM	
更新PC	PC	PC $\leftarrow valP$

读指令字节
 读寄存器字节
 [读常数字]
 计算下一个PC
 读操作数A
 读操作数B
 执行ALU的操作
 设置条件码寄存器
 [读写内存]
 ALU的运算结果写回
 [内存结果写回]
 更新PC

- 所有的指令有相同的格式
- 每一步计算的内容有区别

各阶段的运算

		call Dest	
取指	icode,ifun	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	读指令字节
	rA,rB		[读寄存器字节]
	valC	$\text{valC} \leftarrow M_8[\text{PC}+1]$	读常数字
	valP	$\text{valP} \leftarrow \text{PC}+9$	计算下一个PC
译码	valA, srcA		[读操作数A]
	valB, srcB	$\text{valB} \leftarrow R[\%rsp]$	读操作数B
执行	valE	$\text{valE} \leftarrow \text{valB} + -8$	执行ALU的操作
	Cond code		[设置条件码寄存器]
访存	valM	$M_8[\text{valE}] \leftarrow \text{valP}$	内存读写
写回	dstE	$R[\%rsp] \leftarrow \text{valE}$	ALU的运算结果写回
	dstM		[内存结果写回]
更新PC	PC	$\text{PC} \leftarrow \text{valC}$	更新PC

- 所有指令遵循相同的一般模式
- 区别在于每一步计算的不同

计算的数值

■取指

icode	指令码
ifun	指令功能
rA	指令指定的寄存器A
rB	指令指定的寄存器B
valC	指令中的常数
valP	增加后的PC值

■译码

srcA	寄存器A的ID
srcB	寄存器B的ID
dstE	目的寄存器E
dstM	目的寄存器M
valA	寄存器A的值
valB	寄存器B的值

■执行

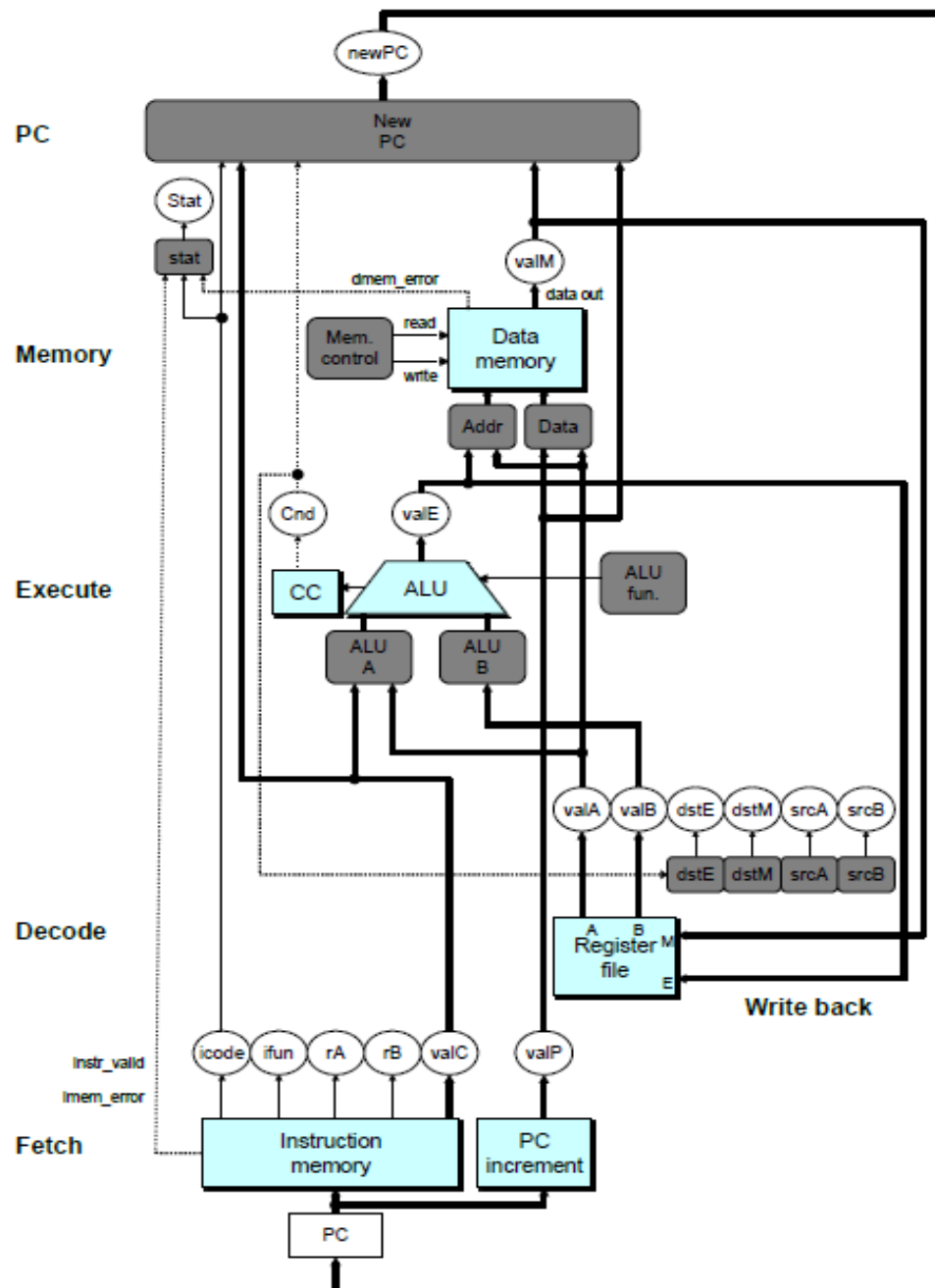
valE	ALU运算结果
Cnd	是否跳转/传送的标识

■访存

valM	来自内存的数值
------	---------

SEQ 硬件结构

- 浅蓝色方框: 硬件单元
 - 如内存、ALU等
- 灰色方框: 控制逻辑块
 - 用HCL语言描述
- 白色的圆圈: 线路名字（信号标签），而非硬件单元
- 粗线: 64位(bit)的字数值
- 细线: 4-8位的数值
- 虚线: 1位的数据



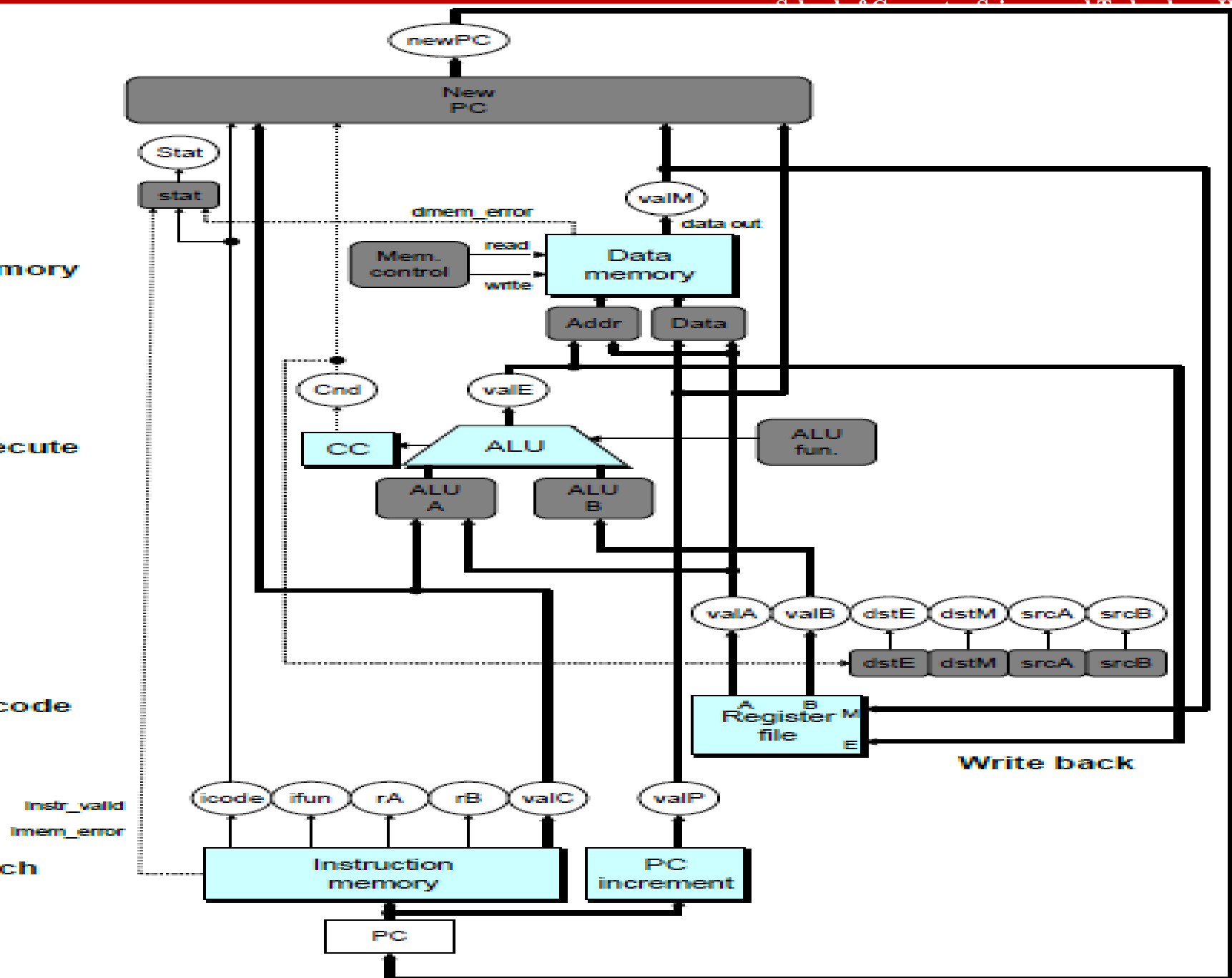
PC

Memory

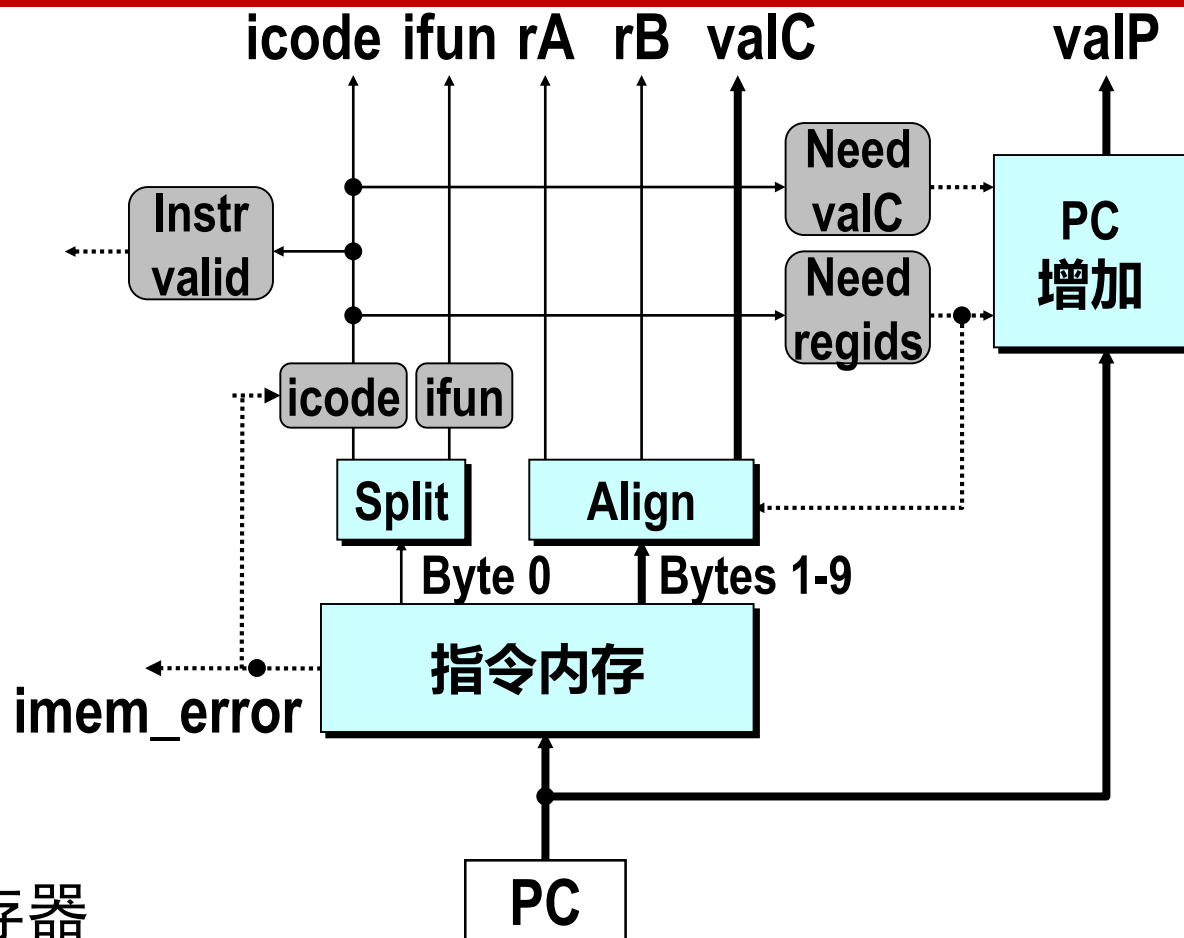
Execute

Decode

Fetch



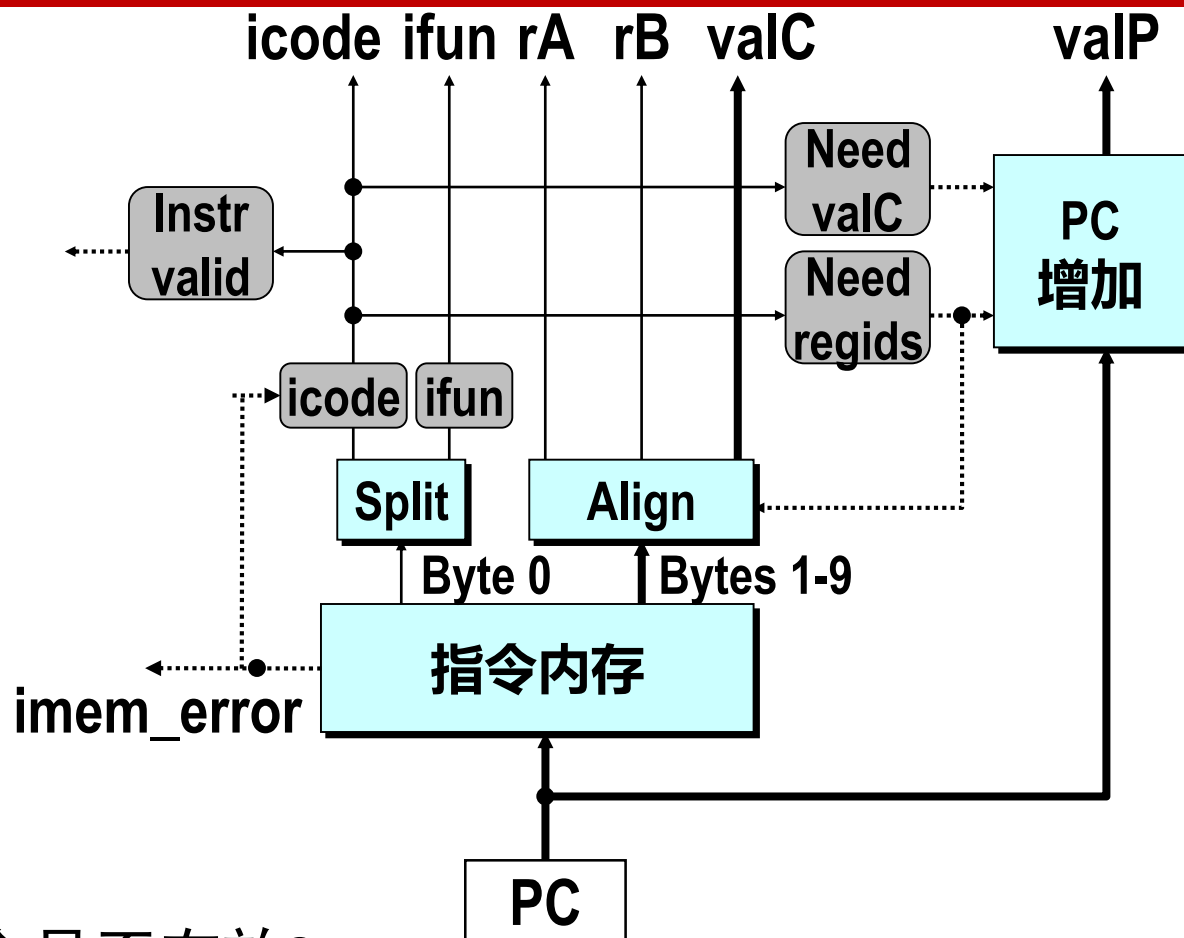
取指逻辑



■ 预定义的单元

- PC: 存储PC的寄存器
- 指令内存: 读十个字节 (PC to PC+9)
 - 发出地址无效的信号
- Split: 把指令字节分为icode和ifun
- Align: 获取指令中rA, rB 和valC的字段

取指逻辑



■ 控制逻辑

- Instr. Valid: 指令是否有效?
- icode, ifun: 指令地址无效时生成no-op指令
- Need regids: 指令是否有寄存器字节?
- Need valC: 指令是否有常数字?

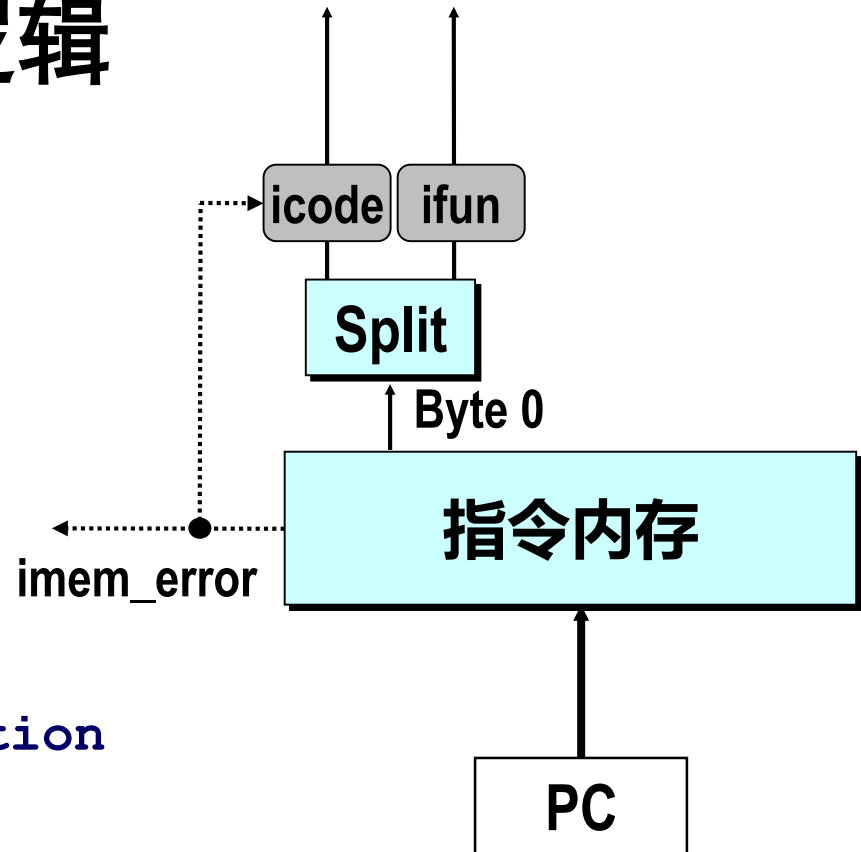
HCL描述的取指控制逻辑

```
# Determine instruction code
```

```
int icode = [
    imem_error: INOP;
    1: imem_icode;
];
```

```
# Determine instruction function
```

```
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```



HCL描述的取指控制逻辑

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

需要寄存器操作数

HCL描述的取指控制逻辑

```
bool need_regids = icode in  
    { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,  
      IIRMOVQ, IRMMOVQ, IMRMVQ };
```

```
bool instr_valid = icode in  
    { INOP, IHALT, IRRMOVQ, IIRMOVQ,  
      IRMMOVQ, IMRMVQ, IOPQ, IJXX, ICALL,  
      IRET, IPUSHQ, IPOPQ };
```

译码逻辑

■ 寄存器文件

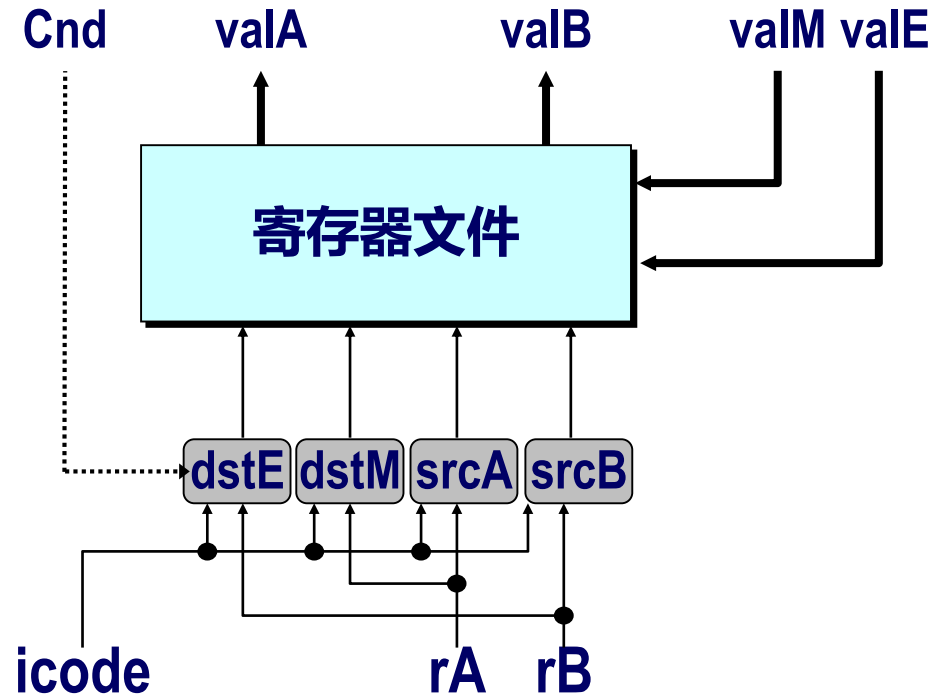
- 读端口 A, B
- 写端口 E, M
- 地址为寄存器的ID
或 15 (0xF,无法访问)

■ 控制逻辑

- srcA, srcB: 读端口地址
- dstE, dstM: 写端口地址

■ 条件信号

- Cnd: 标明是否触发条件传送
 - 在执行阶段计算出Cnd条件信号



srcA——读取valA用的寄存器ID

OPq rA, rB	译码	$valA \leftarrow R[rA]$	读操作数A
cmovXX rA, rB	译码	$valA \leftarrow R[rA]$	读操作数A
rmmovq rA, D(rB)	译码	$valA \leftarrow R[rA]$	读操作数A
popq rA	译码	$valA \leftarrow R[\%rsp]$	读栈指针
jXX Dest	译码		无操作数
call Dest	译码		无操作数
ret	译码	$valA \leftarrow R[\%rsp]$	读栈指针

```
int srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # 不需要寄存器
];
```

dstE——写回的目的寄存器ID

OPq rA, rB	写回	$R[rB] \leftarrow \text{valE}$	结果写回
cmovXX rA, rB	写回	$R[rB] \leftarrow \text{valE}$	有条件的写回结果
rmmovq rA, D(rB)	写回		无
popq rA	写回	$R[\%rsp] \leftarrow \text{valE}$	更新栈指针
jXX Dest	写回		无
call Dest	写回	$R[\%rsp] \leftarrow \text{valE}$	更新栈指针
ret	写回	$R[\%rsp] \leftarrow \text{valE}$	更新栈指针

```

int dstE = [
    icode in { IRRMOVQ } && Cnd : rB;
    icode in { IIRMOVQ, IOPQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # 不写任何寄存器
];

```

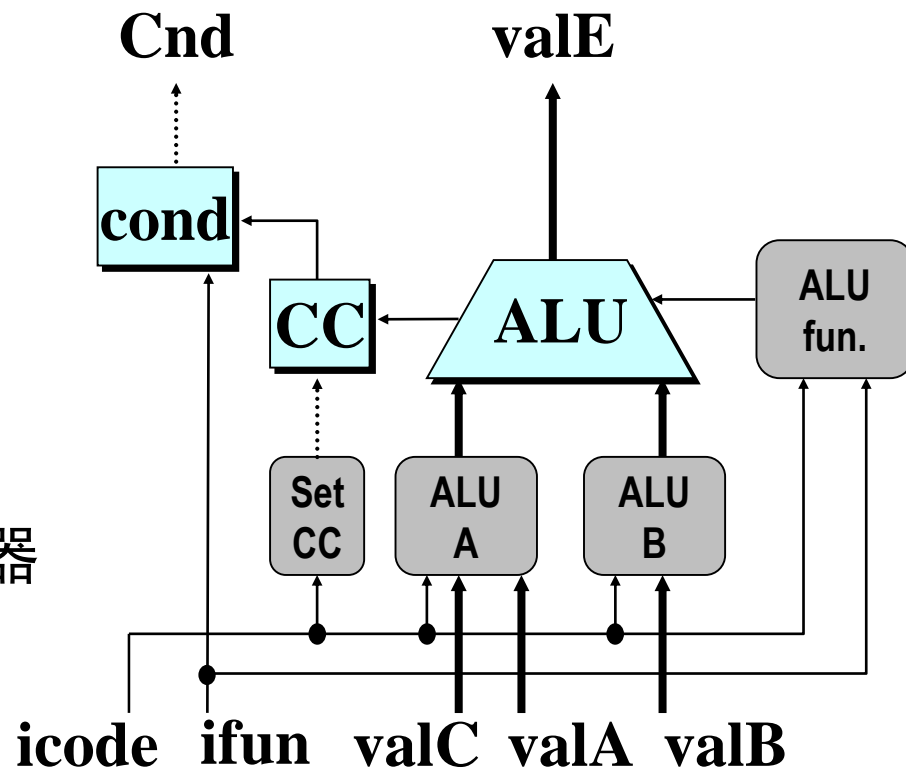
执行逻辑

■ 单元

- ALU
 - 实现四种所需的功能
 - 生成条件码数值
- CC
 - 含三个条件码位的寄存器
- cond
 - 计算条件转移/跳转标识

■ 控制逻辑

- Set CC: 是否加载条件码寄存器
- ALU A: ALU的输入A
- ALU B: ALU的输入B
- ALU fun: ALU执行哪个功能



ALU的输入A

OPq rA, rB	执行	$\text{valE} \leftarrow \text{valB OP valA}$	执行ALU的操作
cmovXX rA, rB	执行	$\text{valE} \leftarrow 0 + \text{valA}$	通过ALU传送数据A
rmmovq rA, D(rB)	执行	$\text{valE} \leftarrow \text{valB} + \text{valC}$	计算有效地址
popq rA	执行	$\text{valE} \leftarrow \text{valB} + 8$	增加栈指针的值
jXX Dest	执行		无操作
call Dest	执行	$\text{valE} \leftarrow \text{valB} + -8$	减少栈指针的值
ret	执行	$\text{valE} \leftarrow \text{valB} + 8$	增加栈指针的值

```

int aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # 其他指令不需要ALU
];

```


ALU操作

OPl rA, rB	执行	$\text{valE} \leftarrow \text{valB} \text{ OP } \text{valA}$	执行ALU的操作
cmovXX rA, rB	执行	$\text{valE} \leftarrow 0 + \text{valA}$	通过ALU传送数据A
rmmovl rA, D(rB)	执行	$\text{valE} \leftarrow \text{valB} + \text{valC}$	计算有效地址
popq rA	执行	$\text{valE} \leftarrow \text{valB} + 8$	增加栈指针的值
jXX Dest	执行		无操作
call Dest	执行	$\text{valE} \leftarrow \text{valB} + -8$	减少栈指针的值
ret	执行	$\text{valE} \leftarrow \text{valB} + 8$	增加栈指针的值

```
int alufun = [
    icode == IOPQ : ifun; # IOPQ表示Opq指令
    1 : ALUADD;
];
```

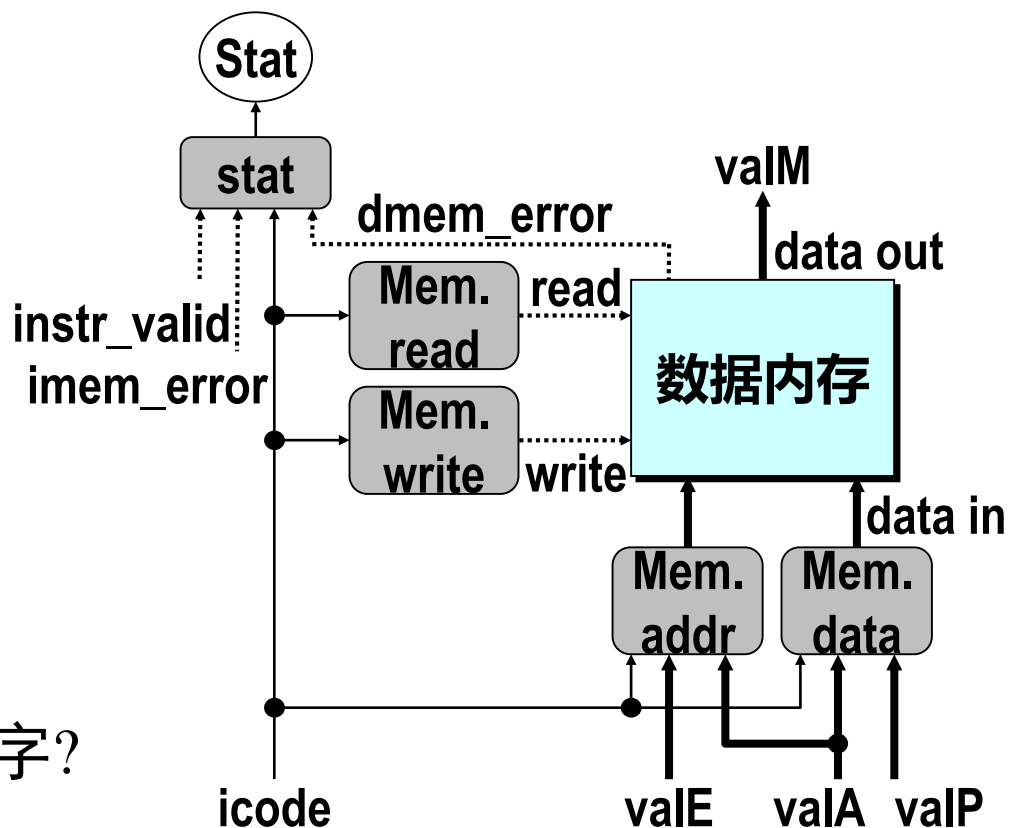
访存逻辑

■ 访存

- 读、写内存里的数据字

■ 控制逻辑

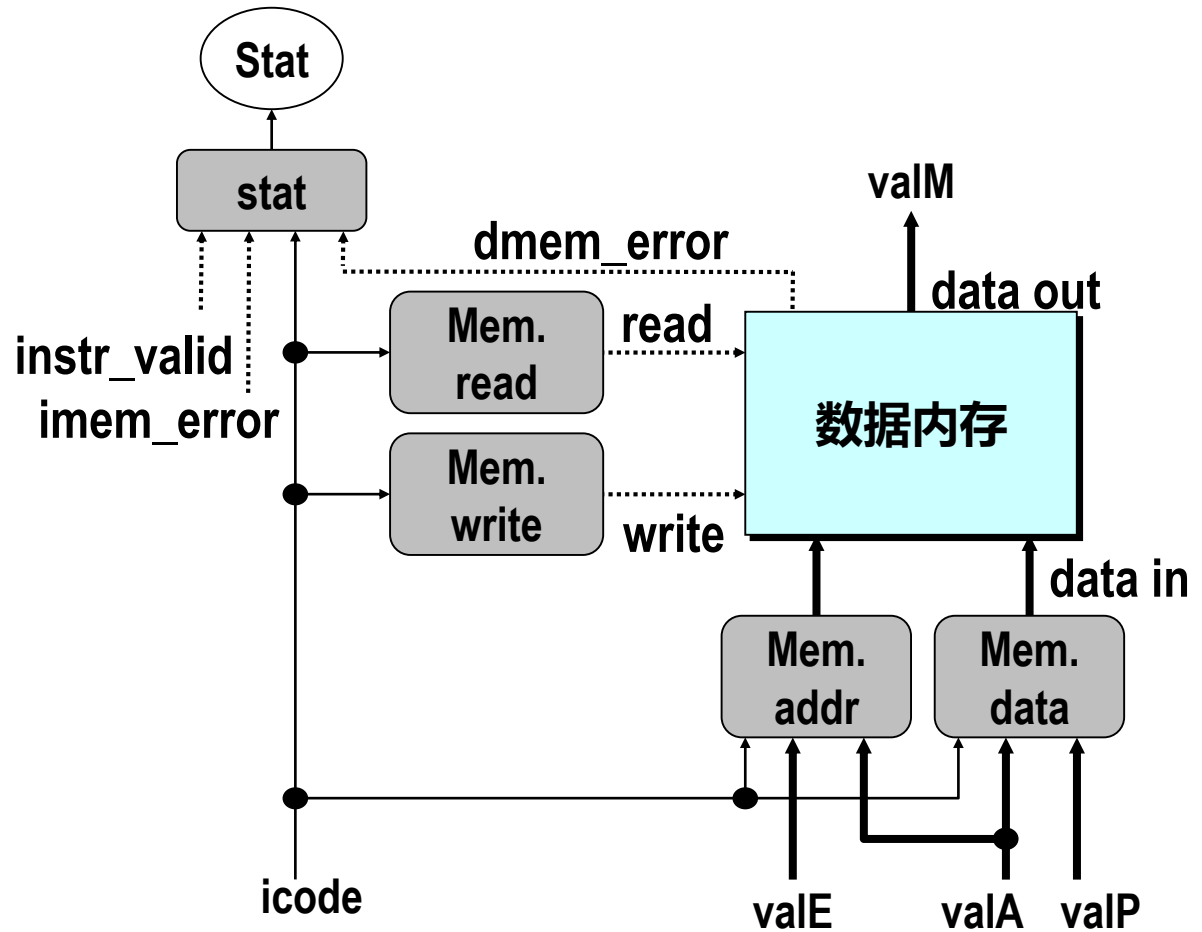
- stat: 指令状态是什么?
- Mem. read: 是否读数据字?
- Mem. write: 是否写数据字?
- Mem. addr.: 选择地址
- Mem. data.: 选择数据



指令状态

■ 控制逻辑

- stat: 指令状态是什么?



确定指令状态码

```
int Stat = [
    imem_error || dmem_error : SADR; #地址异常
    !instr_valid: SINS; #非法指令异常
    icode == IHALT : SHLT; #halt状态
    1 : SAOK; #正常操作
];
```

内存地址

<code>OPq rA, rB</code>	访存		无操作
<code>rmmovq rA, D(rB)</code>	访存	$M_8[\text{valE}] \leftarrow \text{valA}$	数据写入内存
<code>popq rA</code>	访存	$\text{valM} \leftarrow M_8[\text{valA}]$	从栈里读取数据
<code>jXX Dest</code>	访存		无操作
<code>call Dest</code>	访存	$M_8[\text{valE}] \leftarrow \text{valP}$	返回值入栈
<code>ret</code>	访存	$\text{valM} \leftarrow M_8[\text{valA}]$	读返回地址

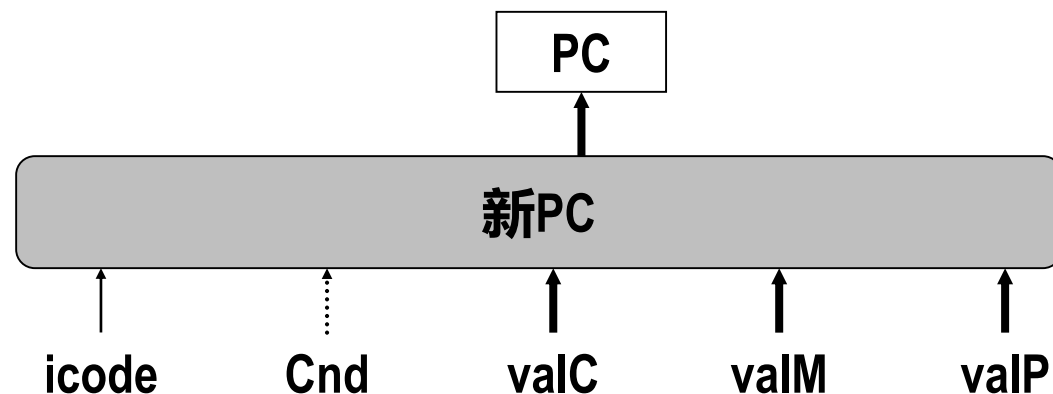
```
int mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # 其他指令不需要地址
];
```

读内存

OPq rA, rB	访存		无操作
rmmovq rA, D(rB)	访存	$M_8[valE] \leftarrow valA$	数据写入内存
popq rA	访存	$valM \leftarrow M_8[valA]$	从栈里读取数据
jXX Dest	访存		无操作
call Dest	访存	$M_8[valE] \leftarrow valP$	返回值入栈
ret	访存	$valM \leftarrow M_8[valA]$	读返回地址

`bool mem_read = icode in { IMRMOVQ, IPOPOPQ, IRET };`

更新PC的逻辑



■ 新PC

- 选取下一个PC值

■ 条件信号

- Cnd: 标明是否触发条件跳转
 - 在执行阶段计算出Cnd条件信号

更新PC

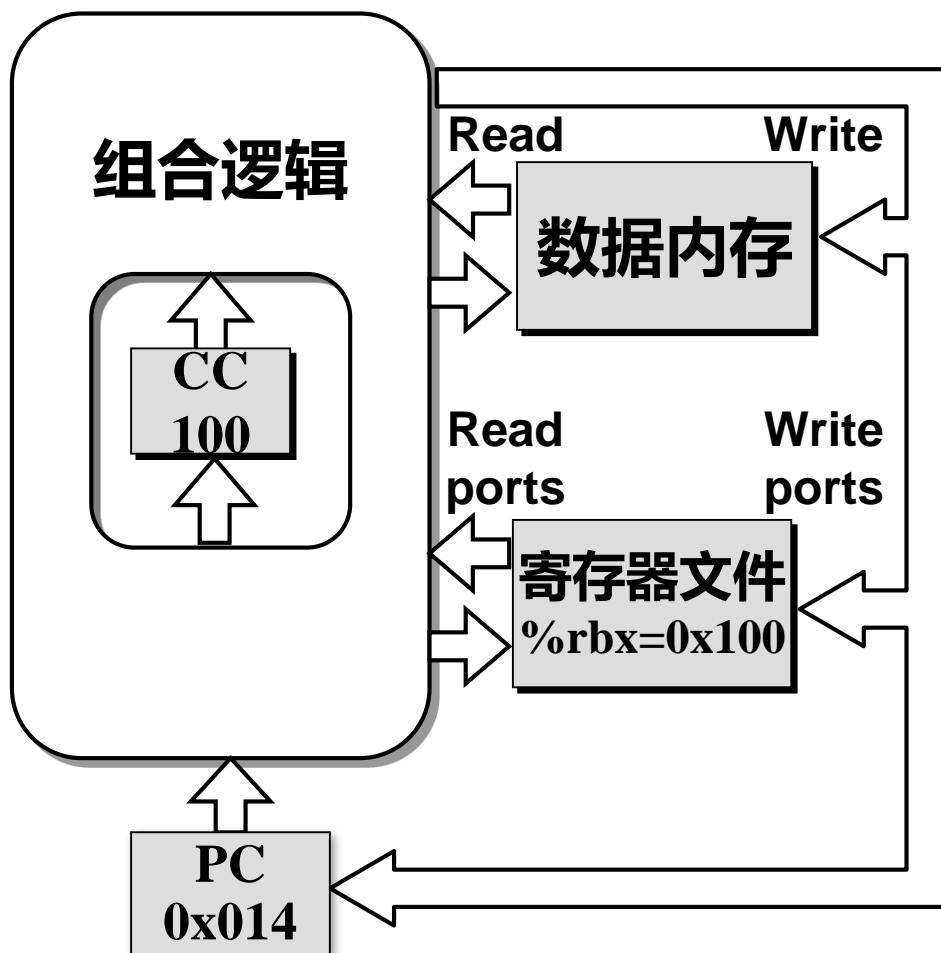
<code>OPq rA, rB</code>	更新PC	$PC \leftarrow valP$	更新PC
<code>rmmovq rA, D(rB)</code>	更新PC	$PC \leftarrow valP$	更新PC
<code>popq rA</code>	更新PC	$PC \leftarrow valP$	更新PC
<code>jXX Dest</code>	更新PC	$PC \leftarrow Cnd ? valC : valP$	更新PC
<code>call Dest</code>	更新PC	$PC \leftarrow valC$	PC设为目的地址
<code>ret</code>	更新PC	$PC \leftarrow valM$	PC设为返回地址

```

int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Cnd : valC;
    icode == IRET : valM;
    1 : valP;
];

```

SEQ 操作#1



■ 组合逻辑 无需时序控制

- ALU
- 控制逻辑
- 读内存
 - 指令内存
 - 寄存器文件(读)
 - 数据内存(读)

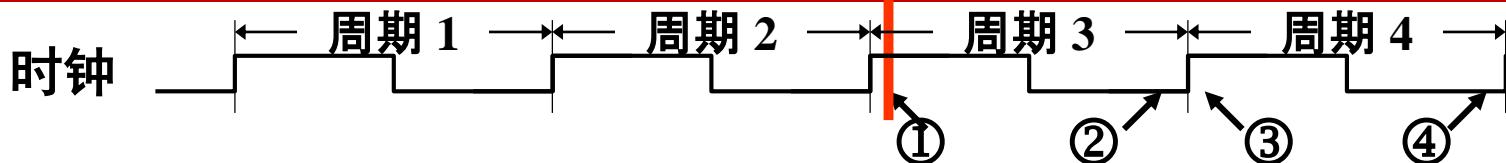
■ 状态单元(存储设备)

时钟寄存器

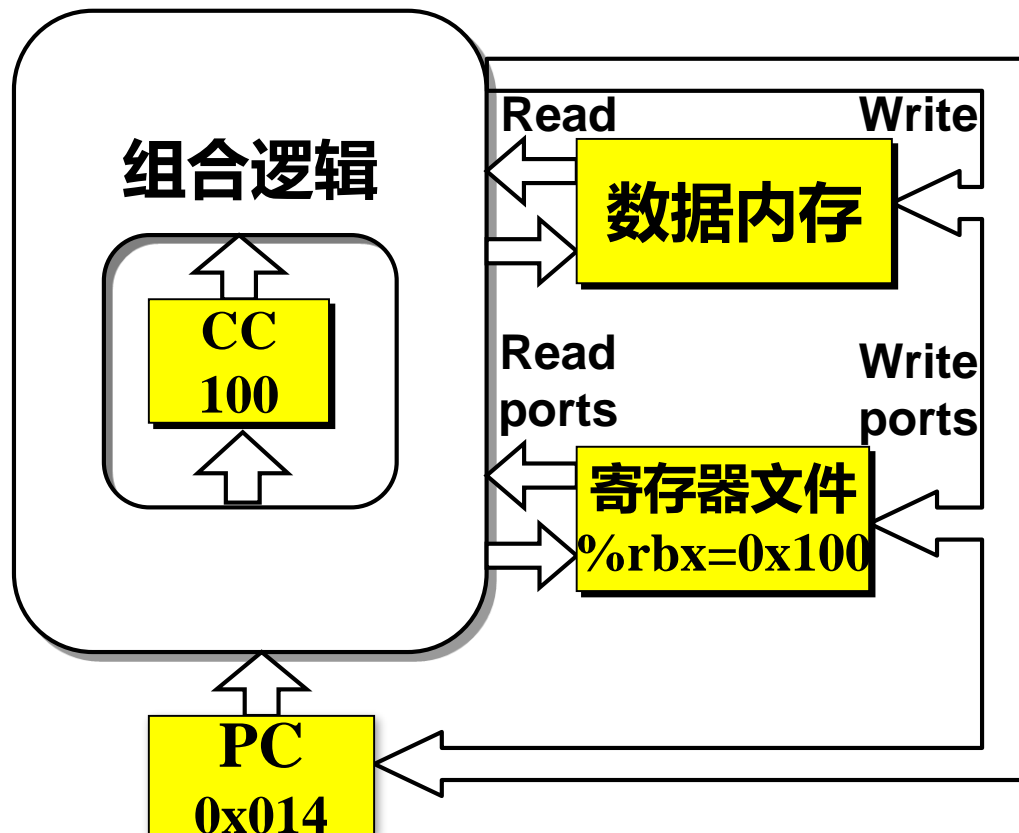
- 程序计数器PC
- 条件码寄存器
- 随机访问存储器
- 数据内存(写)
- 寄存器文件(写)

都在时钟上升沿时更新

SEQ 操作 #2

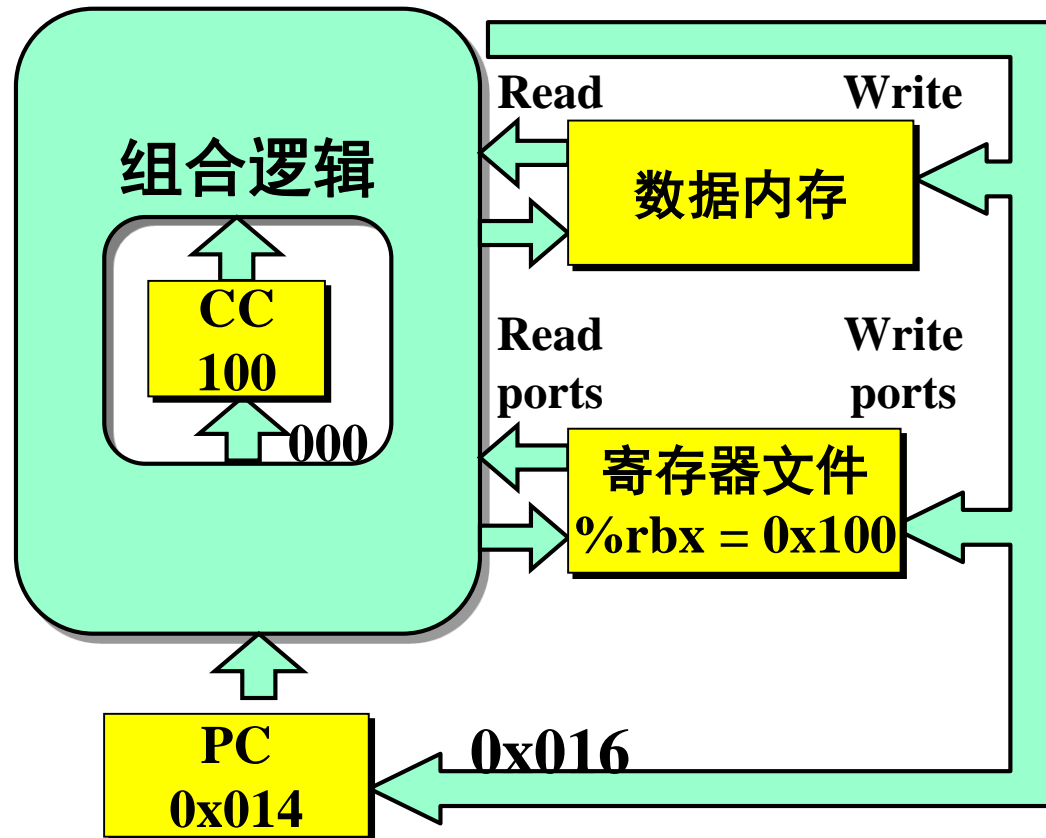
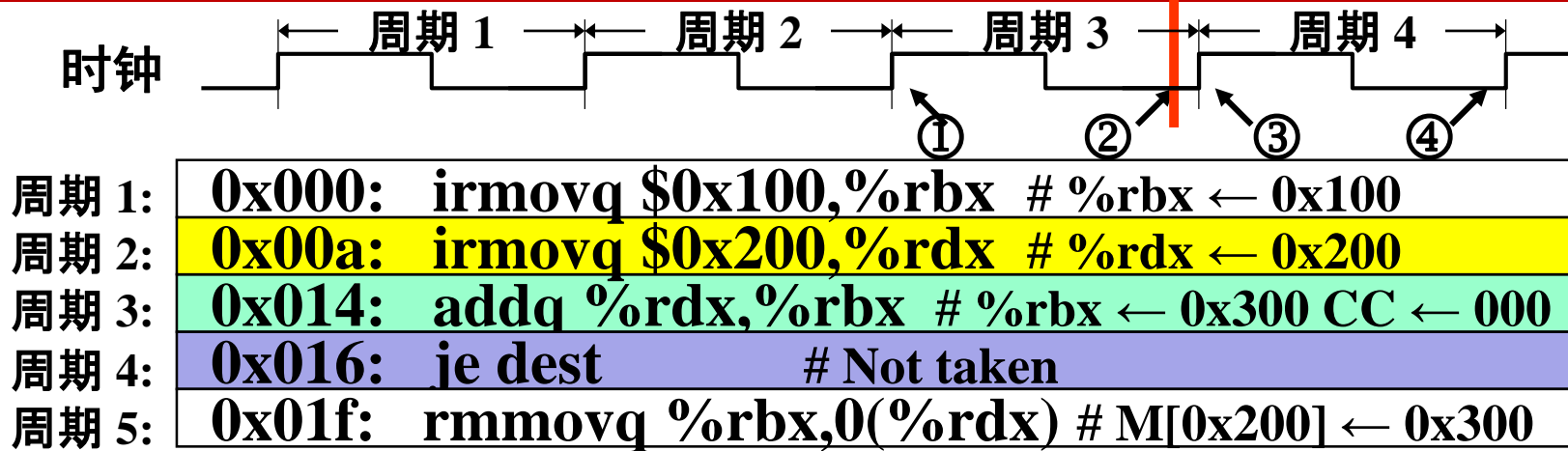


周期 1:	0x000: irmovq \$0x100,%rbx # %rbx ← 0x100
周期 2:	0x00a: irmovq \$0x200,%rdx # %rdx ← 0x200
周期 3:	0x014: addq %rdx,%rbx # %rbx ← 0x300 CC ← 000
周期 4:	0x016: je dest # Not taken
周期 5:	0x01f: rmmovq %rbx,0(%rdx) # M[0x200] ← 0x300



- 依据第二条irmovq指令设置的状态
- 组合逻辑开始对状态的变化作出反应

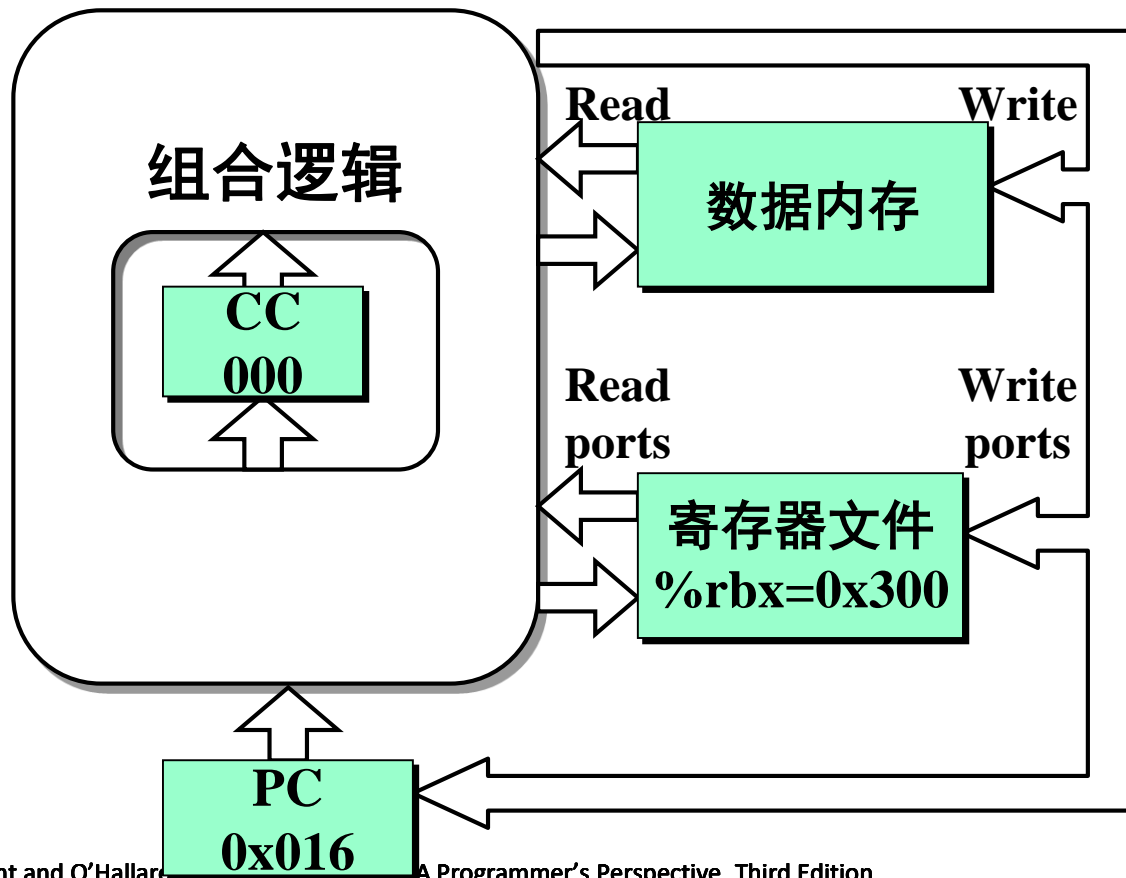
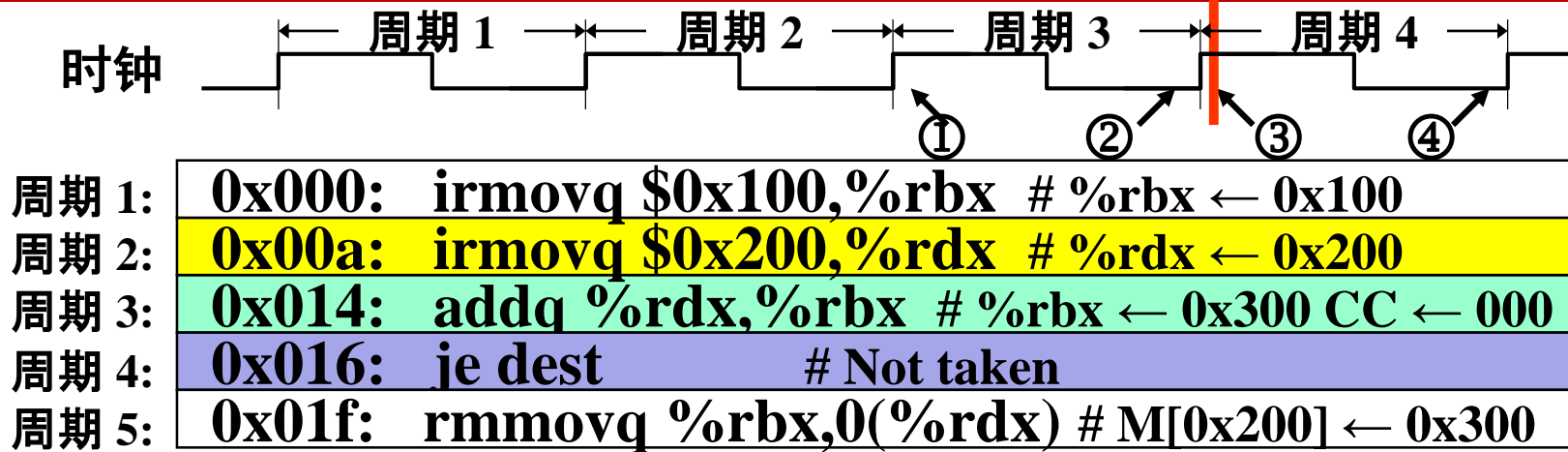
SEQ 操作 #3



- 依据第二条irmovq指令设置的状态
- 组合逻辑为addq指令生成结果

%rbx ← 0x300

SEQ 操作 #4

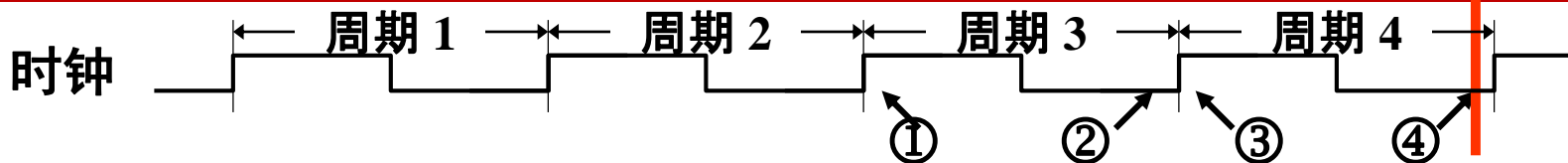


- 依据addq指令设置状态
- 组合逻辑**开始**对状态的变化作出反应

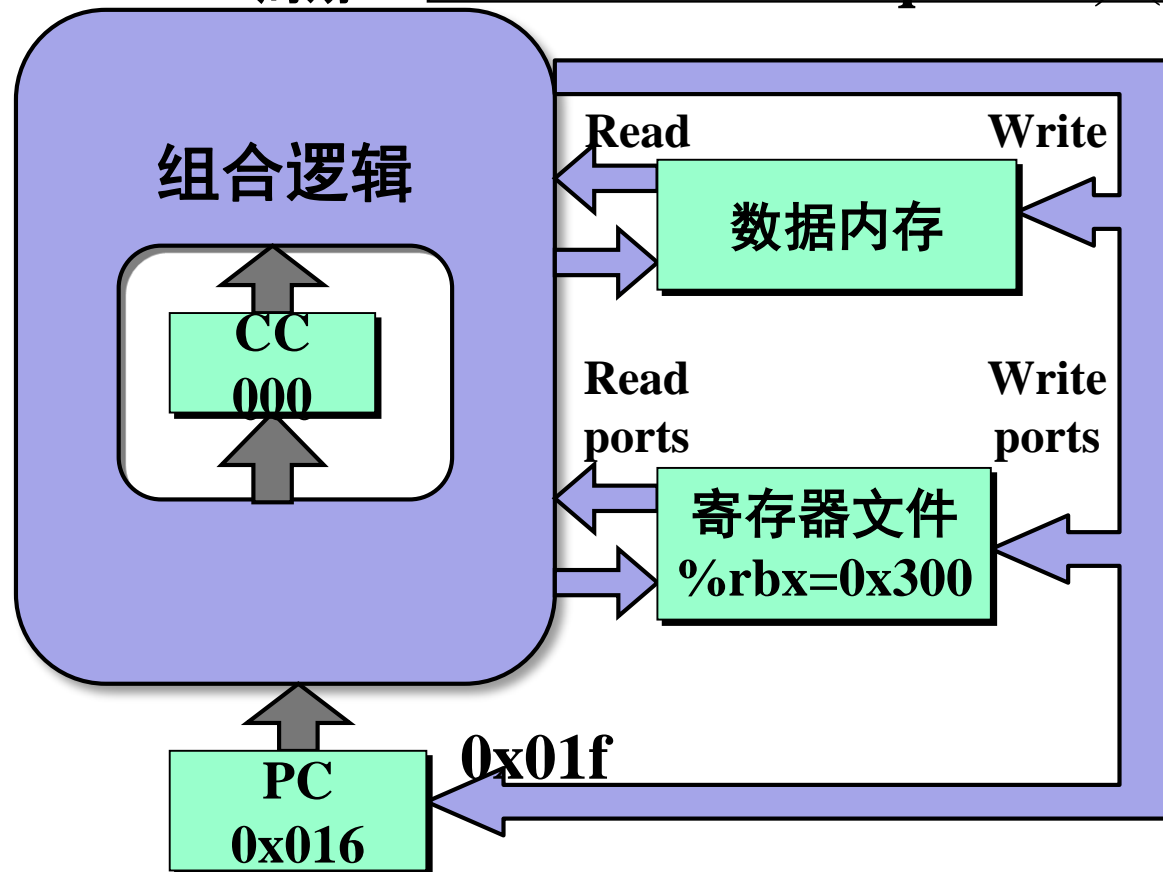
SEQ

操作

#5



周期 1:	0x000: irmovq \$0x100,%rbx # %rbx ← 0x100
周期 2:	0x00a: irmovq \$0x200,%rdx # %rdx ← 0x200
周期 3:	0x014: addq %rdx,%rbx # %rbx ← 0x300 CC ← 000
周期 4:	0x016: je dest # Not taken
周期 5:	0x01f: rmmovq %rbx,0(%rdx) # M[0x200] ← 0x300



- 依据addq指令设置状态
- 组合逻辑为je指令生成结果

SEQ 总结

■ 实现

- 把每条指令表示成一系列简单的阶段（步骤）
- 每种指令类型都遵循相同的统一流程
- 整合寄存器、内存、预设的组合逻辑块
- 用控制逻辑连接成一个整体

■ 不足的地方

- 实际使用起来太慢
- 信号必须在一个周期内，传播经过所有阶段：指令内存、寄存器文件、ALU以及数据内存
- 时钟必须非常慢
- 硬件单元只在时钟周期的一部分时间内活动（被使用）