

## 第四章 处理器体系结构

### 4-3 ——顺序执行的处理器

# 本节主要内容

## ■ 回顾

- 指令集回顾
- 逻辑设计回顾、HCL语言

## ■ 顺序执行（Sequential，简称SEQ）的实现

- SEQ的6个阶段
- SEQ的计算过程（微指令）
- SEQ硬件逻辑实现

# Y86-64 指令集 1

1,2,9,10

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	注意: rrmovq rA, rB 也归属此类					
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

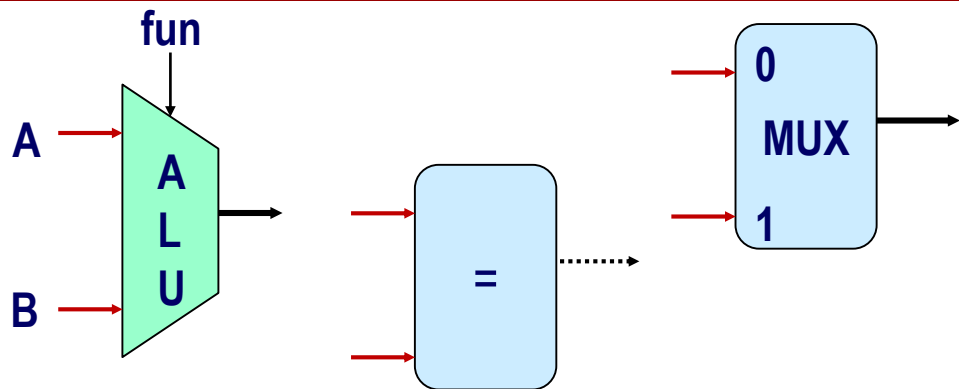
xorq 6 3

# 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								
call Dest	8	0								
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

# 构建CPU的硬件模块

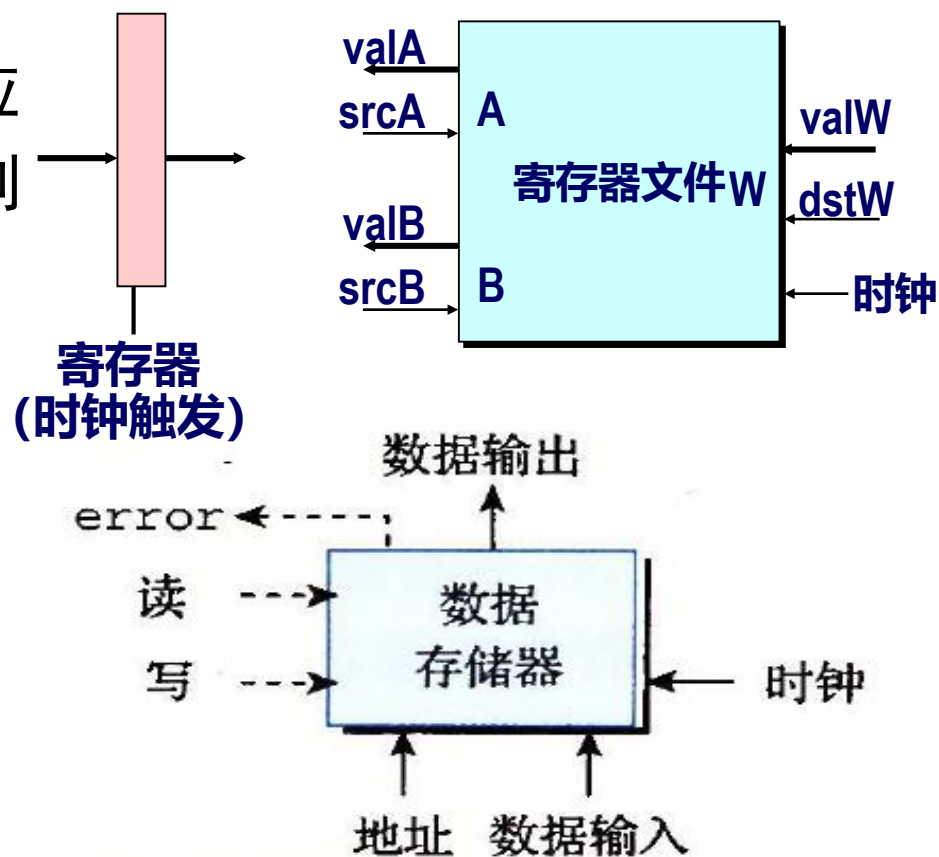


## ■ 组合逻辑

- 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-时序逻辑

- 存储字节
- 可寻址的内存
- 不可寻址的寄存器
- 时钟上升沿触发



# 硬件控制语言HCL

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

## ■ 数据类型

- 布尔型: Boolean
  - a, b, c, ...
- 整型: words
  - 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

- 根据返回值的类型分类

## ■ 布尔表达式

- 逻辑操作
  - `a && b, a || b, !a`
- 字的比较
  - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- 集合关系
  - `A in { B, C, D }`
    - 等同于 `A == B || A == C || A == D`

## ■ 整数表达式

- 表达式实例
  - 情况表达式 `[ a : A; b : B; c : C ]`
  - 依次测试选择表达式 `a, b, c, ...` 等等
  - 当首个选择表达式测试通过后返回相应的情况 `A, B` 或 `C, ...`

H  
C  
L  
程  
序

```

wordsig rB      'rb'      # rB field from instruction
wordsig valC    'valc'    # Constant from instruction
wordsig valP    'valp'    # Address of following instruction
boolsig imem_error 'imem_error' # Error signal from instruction memory
boolsig instr_valid 'instr_valid' # Is fetched instruction valid?

##### Decode stage computations #####
wordsig valA    'vala'    # Value from register A port
wordsig valB    'valb'    # Value from register B port

##### Execute stage computations #####
wordsig valE    'vale'    # Value computed by ALU
boolsig Cnd     'cond'    # Branch test
|
##### Memory stage computations #####
wordsig valM    'valm'    # Value read from memory
boolsig dmem_error 'dmem_error' # Error signal from data memory

#####
# Control Signal Definitions. #
#####

##### Fetch Stage #####

# Determine instruction code
word icode = [
    imem_error: INOP;
    1: imem_icode;      # Default: get from instruction memory
];

# Determine instruction function
word ifun = [
    imem_error: FNONE;
    1: imem_ifun;      # Default: get from instruction memory
];

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

# Does fetched instruction require a regid byte?
bool need_regids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
              IIRMOVQ, IRMMOVQ, IMRMOVQ };

```

# HCL 描述中使用的常数 (P277)

名称	值 (16进制)	含义
IHALT	0	halt 指令的代码
INOP	1	nop 指令的代码
IRRMovQ	2	rrmovq 指令的代码
IIRMOVQ	3	irmovq 指令的代码
IRMMOVQ	4	rmmovq 指令的代码
IMRMOVQ	5	mrmmovq 指令的代码
IOPQ	6	整数运算指令的代码
IJXX	7	跳转指令的代码
ICALL	8	call 指令的代码
IRET	9	ret 指令的代码
IPUSHQ	A	pushq 指令的代码
IPOPQ	B	popq 指令的代码
FNONE	0	默认功能码
RRSP	4	% rsp 的寄存器 ID
RNONE	F	表明没有寄存器文件访问
ALUADD	0	加法运算的功能
SAOK	1	①正常操作状态码
SADR	2	②地址异常状态码
SINS	3	③非法指令异常状态码
SHLT	4	④halt 状态码

这些值表示的是指令、功能码、寄存器ID、ALU操作和状态码的编码

## ■ 取指

- ## ■ 译码

- ## ■ 执行

- ## ■ 访存

- ## ■ 写回

- ## ■ 更新PC

- ## ■ 更新程序计数器PC



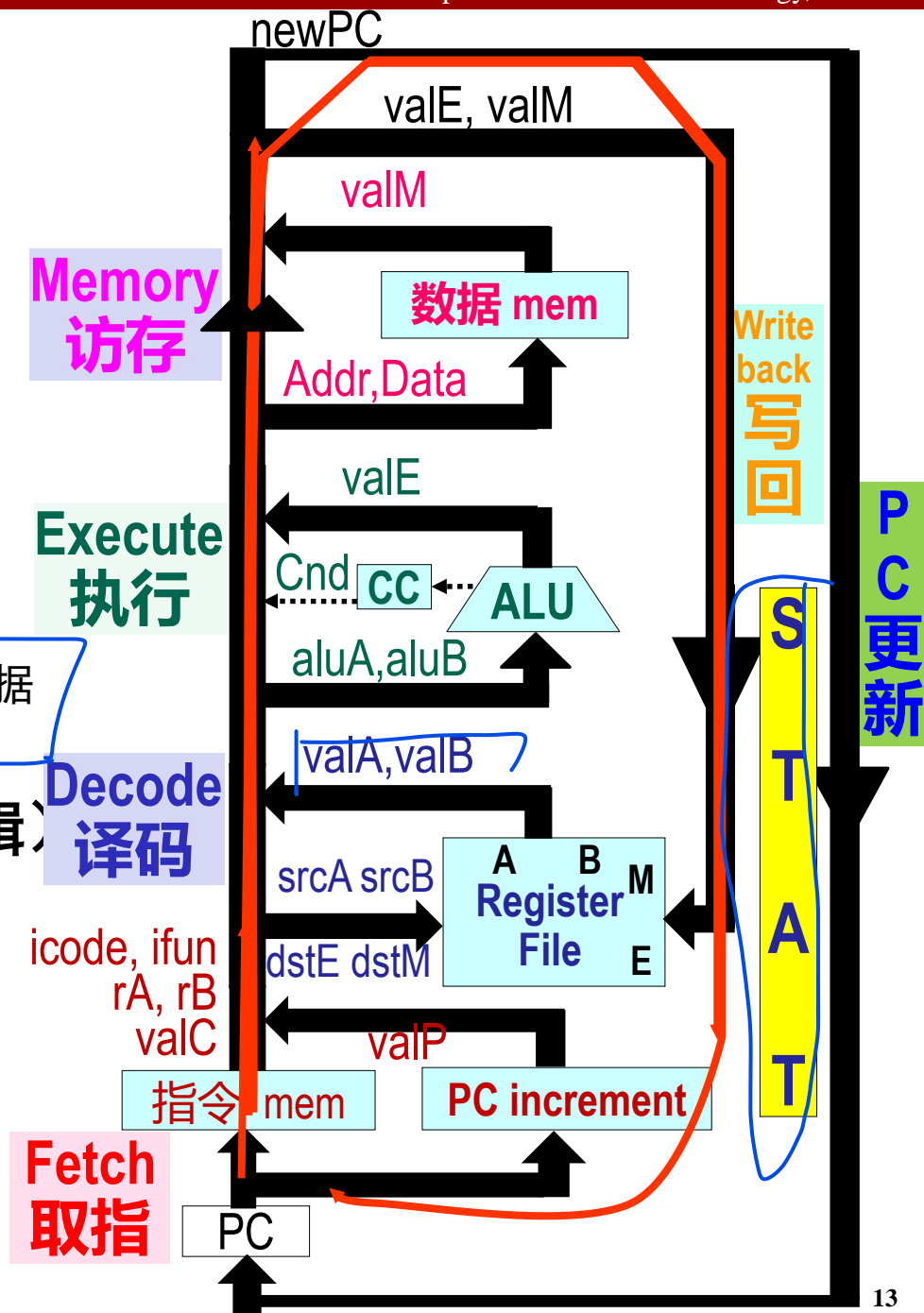
# SEQ 硬件结构

## ■ 状态（数据结构-执行部件）

- 程序计数器 (PC)
- 条件码CC、状态码STAT寄存器
- 寄存器文件RF
- ALU、PC地址增加器
- 内存：访问相同的内存空间
  - 数据：为了读取或写入程序的数据
  - 指令：为了读指令

## ■ 指令流水（函数过程-控制逻辑）

- 读取由PC指定地址的指令
- 分多个阶段执行
- 更新PC
- 分为6个阶段-子程序



# stat 寄存器的作用

## 1.程序状态:

stat 寄存器可以存储程序的当前状态, 例如程序是否正在运行、是否暂停或是否完成。

## 2.错误信息:

当程序执行过程中发生错误时, stat 寄存器可以存储错误代码或状态信息, 帮助调试和错误处理。

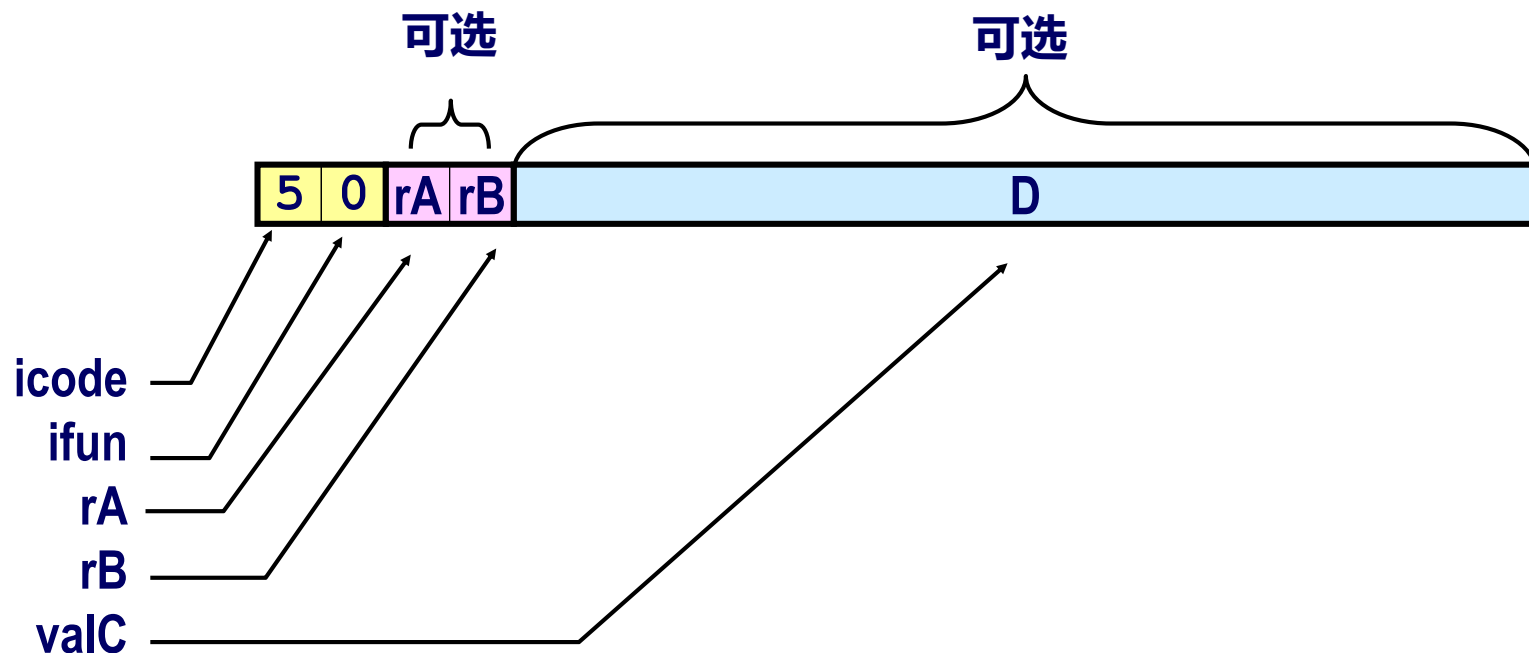
## 3.异常处理:

在异常或中断发生时, stat 寄存器可以记录异常的类型和相关信息, 便于异常处理程序进行处理。

## 4.调试支持:

在调试过程中, stat 寄存器可以提供程序的运行状态, 帮助开发人员理解程序的行为和状态。

# 分析指令编码（以mrmovq为例）



## ■ 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

# 计算的数值

## ■取指

icode	指令码
ifun	功能码
rA	指令中的寄存器A
rB	指令中的寄存器B
valC	指令中的常数
valP	增加后的PC

## ■译码

srcA	寄存器ID A
srcB	寄存器ID B
valA	寄存器值A
valB	寄存器值B
dstE	写入valE的寄存器ID
dstM	写入valM的寄存器ID

## ■执行

- valE ALU运算结果
- Cnd 分支或转移标识

## ■访存

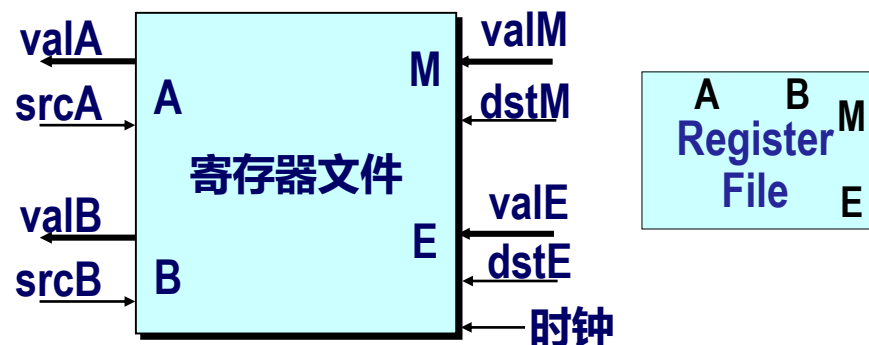
- valM 内存中的数值

## ■写回

- 更新寄存器

## ■更新PC

- PC





# 执行 算术/逻辑 运算

OPq rA, rB

6	fn	rA	rB
---	----	----	----

## ■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

- $PC + 2$

# 计算序列: 算术/逻辑 运算 Ops

OPq rA, rB

取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$
译码	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$
执行	$\text{valE} \leftarrow \text{valB OP valA}$ Set CC
访存	
写回	$R[\text{rB}] \leftarrow \text{valE}$
更新PC	$\text{PC} \leftarrow \text{valP}$

读指令字节  
读寄存器字节

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

结果写回

更新PC

微

操

作

- 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

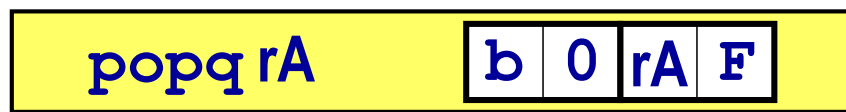
- $PC + 10$

# 计算序列: rmmovq

rmmovq rA, D(rB)		
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 计算下一条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

## ■执行

- 栈指针加8

## ■访存

- 读原来的栈指针（没有加8的）

## ■写回

- 更新栈指针
- 结果写寄存器

## ■更新PC

- PC+2

# 计算序列: popq

	popq rA
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$
译码	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
执行	$\text{valE} \leftarrow \text{valB} + 8$
访存	$\text{valM} \leftarrow M_8[\text{valA}]$
写回	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
更新PC	$\text{PC} \leftarrow \text{valP}$

读指令字节  
读寄存器字节

计算下一条PC

读栈指针  
读栈指针

用法不一样

栈指针加8

valB

从栈里读数据

valA

更新栈指针

结果写回

E和M都更新

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
  - 新的栈指针

注意: pop的时候先读, 再修改指针。  
而push的时候先计算新指针, 后写回寄存器。

OPq rA, rB	
------------	--

译码	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
----	--

读操作数A  
读操作数B

popq rA	
---------	--

译码	$valA \leftarrow R[\%rsp]$ $valB \leftarrow R[\%rsp]$
----	--

读栈指针  
读栈指针

注意：

valA并不总是和rA关联（虽然习惯使用）  
 valB并不总是和rB关联（虽然习惯使用）  
 valA和valB是两个变量，其值可以来自其他寄存器（比如%rsp）的内容

也因此：popq rA指令中并不需要rB寄存器真实出现，因为用到的只是变量valB

# 执行Conditional Move指令



## ■取指

- 读2个字节

## ■译码

- 读操作数寄存器

## ■执行

- 如果条件信号为否, 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



# 计算序列: Cond. Move (另解教材P335习题4.17)

	cmovXX rA, rB
取指	icode:ifun $\leftarrow$ M <sub>1</sub> [PC] rA:rB $\leftarrow$ M <sub>1</sub> [PC+1]
	valP $\leftarrow$ PC+2
译码	valA $\leftarrow$ R[rA] valB $\leftarrow$ 0
执行	valE $\leftarrow$ valB + valA If ! Cond(CC,ifun) rB $\leftarrow$ 0xF
访存	
写回	R[rB] $\leftarrow$ valE
更新PC	PC $\leftarrow$ valP

读指令字节  
读寄存器字节

计算下一条PC  
读操作数A

利用ALU传递数据A  
(阻止寄存器更新)

结果写回

之前的语句得到的CC

cmov不对CC置位

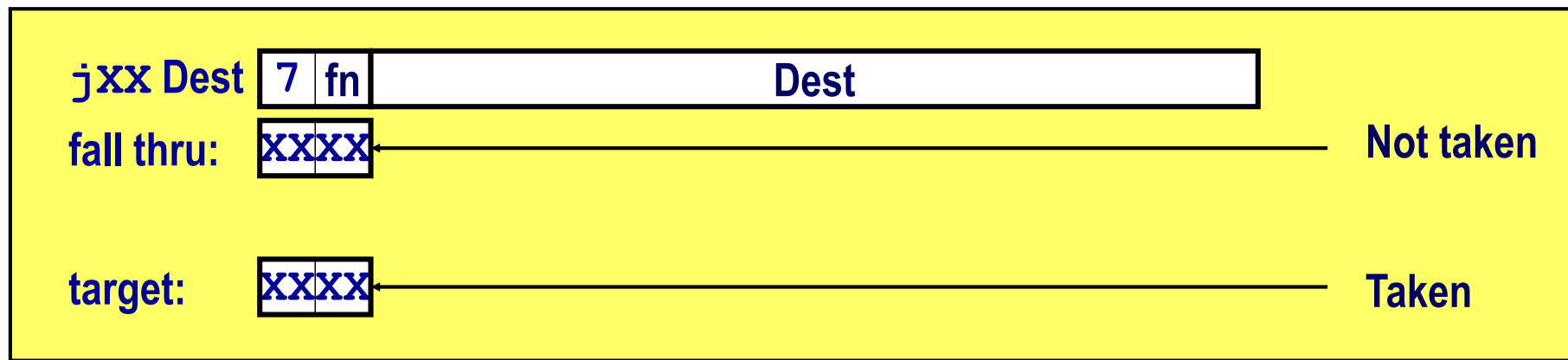
更新PC

- 读rA寄存器并通过ALU传递数据
- 通过将端口值设为0xF来取消数据写入寄存器
  - 如果条件码和传送条件（为false）表明无需传送数据

# 见教材P271页习题，答案在P335

阶段	$\text{cmovXX } rA, rB$
取指	$\text{icode; ifun} \leftarrow M_1[PC]$ $rA; rB \leftarrow M_1[PC+1]$ $\text{valP} \leftarrow PC+2$
译码	$\text{valA} \leftarrow R[rA]$
执行	$\text{valE} \leftarrow 0 + \text{valA}$ $\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$
访存	
写回	$\text{if}(\text{Cnd})$ $R[rB] \leftarrow \text{valE}$
更新 PC	$PC \leftarrow \text{valP}$

# 执行Jumps指令



## ■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

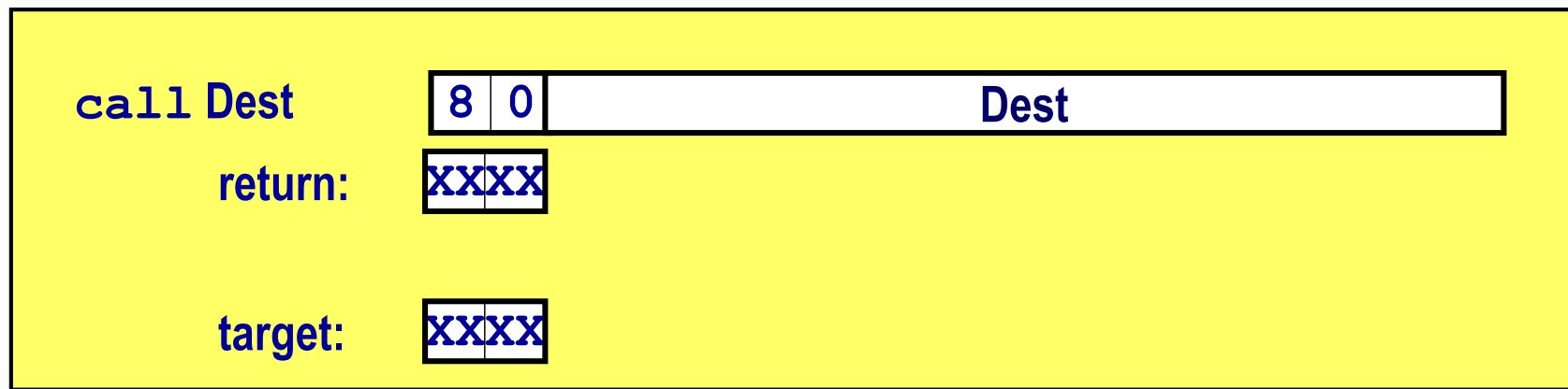
- 如果选择了分支，则把PC值设为分支地址，如果没选择分支，则PC值为增加之后的PC

# 计算序列: 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设为目的地址

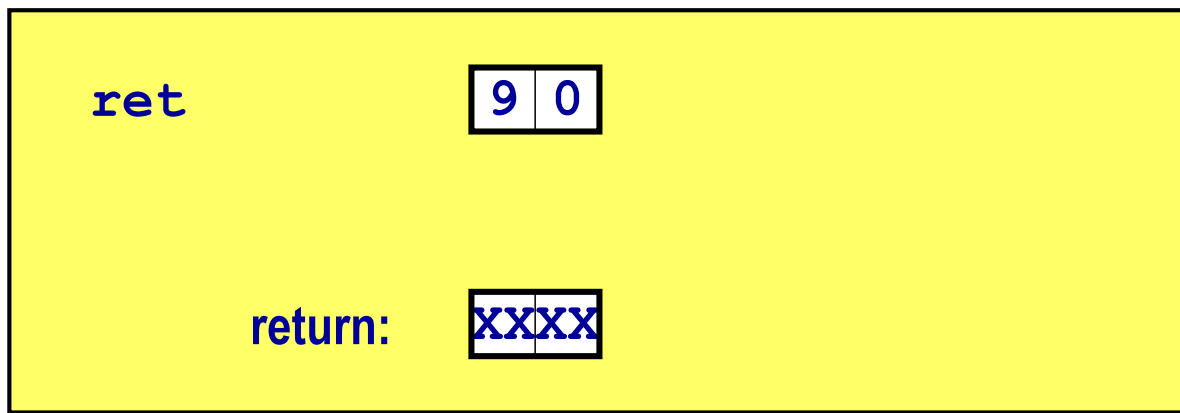
# 计算序列: call

	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

有时valC, 不是valP

# 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

# 计算序列: ret

	ret	
取指	icode:ifun $\leftarrow M_1[PC]$	读指令字节
译码	valA $\leftarrow R[\%rsp]$ valB $\leftarrow R[\%rsp]$	读操作数栈指针 读操作数栈指针
执行	valE $\leftarrow \text{valB} + 8$	栈指针增加
访存	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	$PC \leftarrow \text{valM}$	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



# 计算步骤（以算逻指令与CALL对比）

		OPq rA, rB	
取指	icode,ifun	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	读指令字节
	rA,rB	$\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	读寄存器字节
	valC		[读常数字]
	valP	$\text{valP} \leftarrow \text{PC}+2$	计算下一条PC
译码	valA, srcA	$\text{valA} \leftarrow R[\text{rA}]$	读操作数A
	valB, srcB	$\text{valB} \leftarrow R[\text{rB}]$	读操作数B
执行	valE	$\text{valE} \leftarrow \text{valB OP valA}$	执行ALU的操作
	Cond code	Set CC	设置条件码寄存器
访存	valM		[读写内存]
写回	dstE	$R[\text{rB}] \leftarrow \text{valE}$	ALU的运算结果写回
	dstM		[内存结果写回]
更新PC	PC	$\text{PC} \leftarrow \text{valP}$	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

# 计算步骤

		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	$\text{valP} \leftarrow \text{PC}+9$	计算下一条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

- 所有指令遵循同样的一般格式
- 区别在于每一步计算的不同

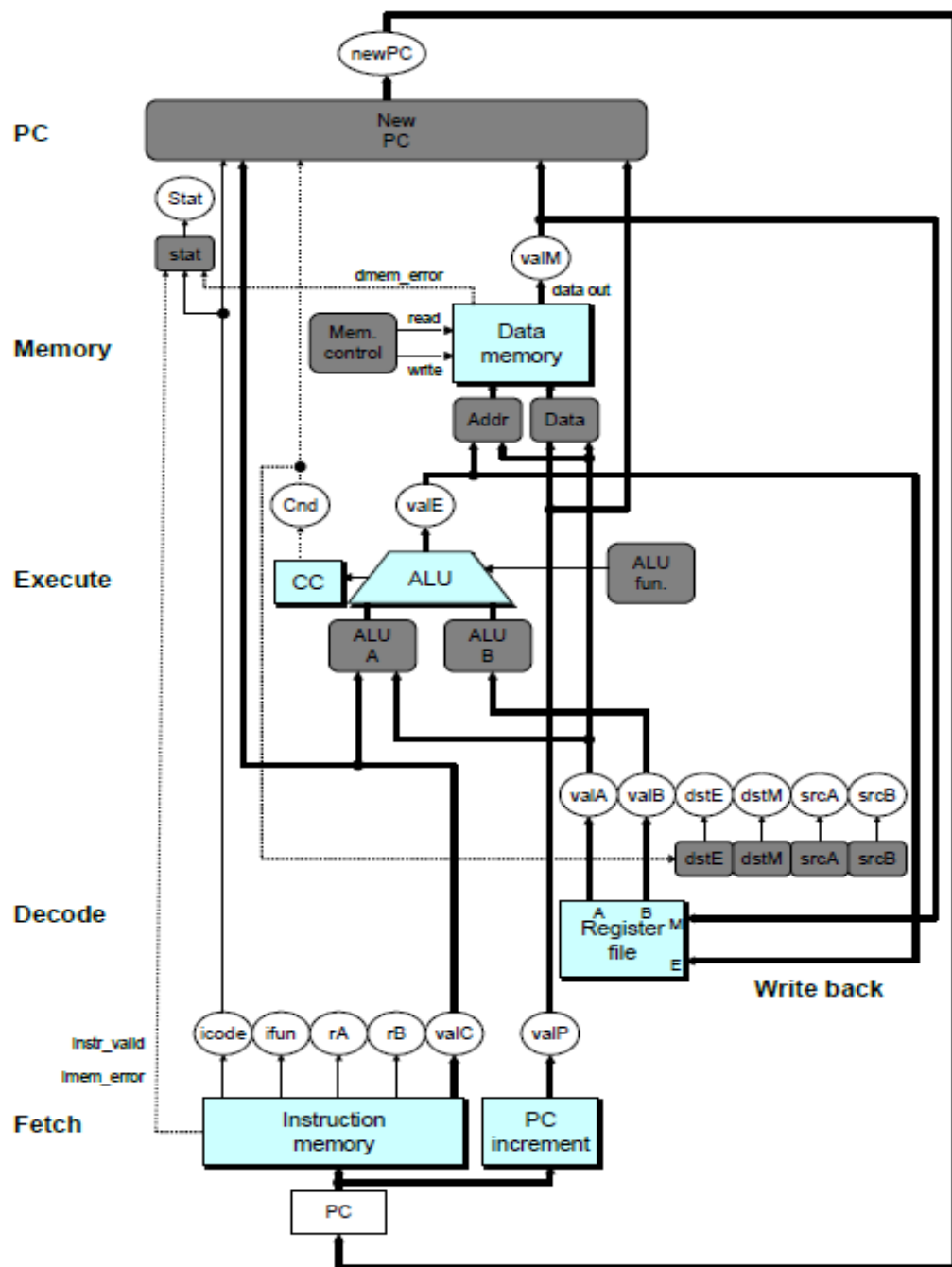
# HCL 描述中使用的常数 (P277)

名称	值 (16进制)	含义
IHALT	0	halt 指令的代码
INOP	1	nop 指令的代码
IRRMovQ	2	rrmovq 指令的代码
IIRMOVQ	3	irmovq 指令的代码
IRMMOVQ	4	rmmovq 指令的代码
IMRMOVQ	5	mrmmovq 指令的代码
IOPQ	6	整数运算指令的代码
IJXX	7	跳转指令的代码
ICALL	8	call 指令的代码
IRET	9	ret 指令的代码
IPUSHQ	A	pushq 指令的代码
IPOPQ	B	popq 指令的代码
FNONE	0	默认功能码
RRSP	4	%rsp 的寄存器 ID
RNONE	F	表明没有寄存器文件访问
ALUADD	0	加法运算的功能
SAOK	1	①正常操作状态码
SADR	2	②地址异常状态码
SINS	3	③非法指令异常状态码
SHLT	4	④halt 状态码

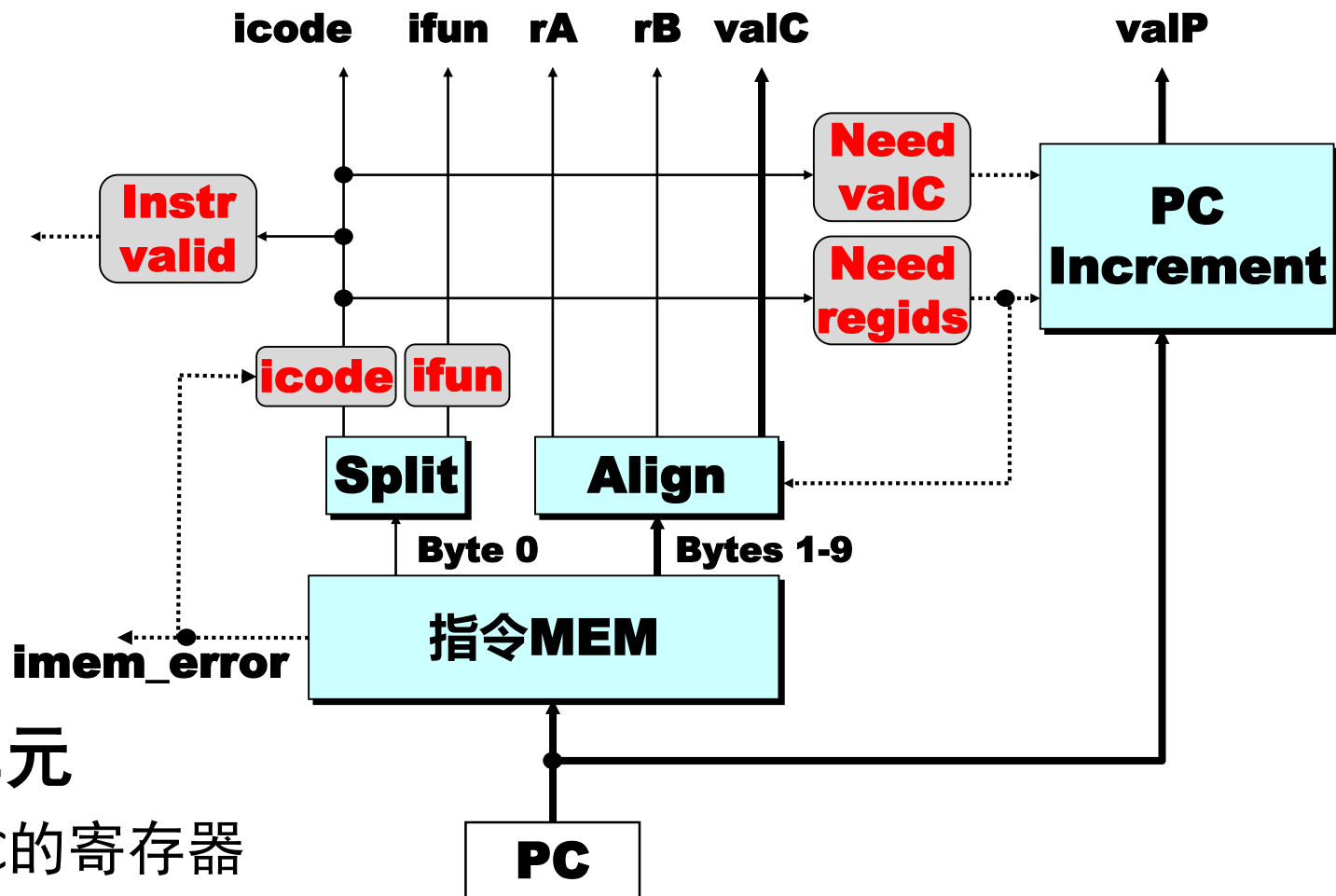
这些值表示的是指令、功能码、寄存器ID、ALU操作和状态码的编码

# SEQ 硬件结构

- 浅蓝色方框: 硬件单元
  - 例如内存、ALU等等
- 灰色方框: 控制逻辑
  - 用HCL语言描述
- 白色的椭圆框:
  - 线路的信号标识
  - 不是硬件单元
- 粗线: 宽度为字长的数据 (64位)
- 细线: 宽度为字节或更窄的数据 (4-8位)
- 虚线: 单个位的数据



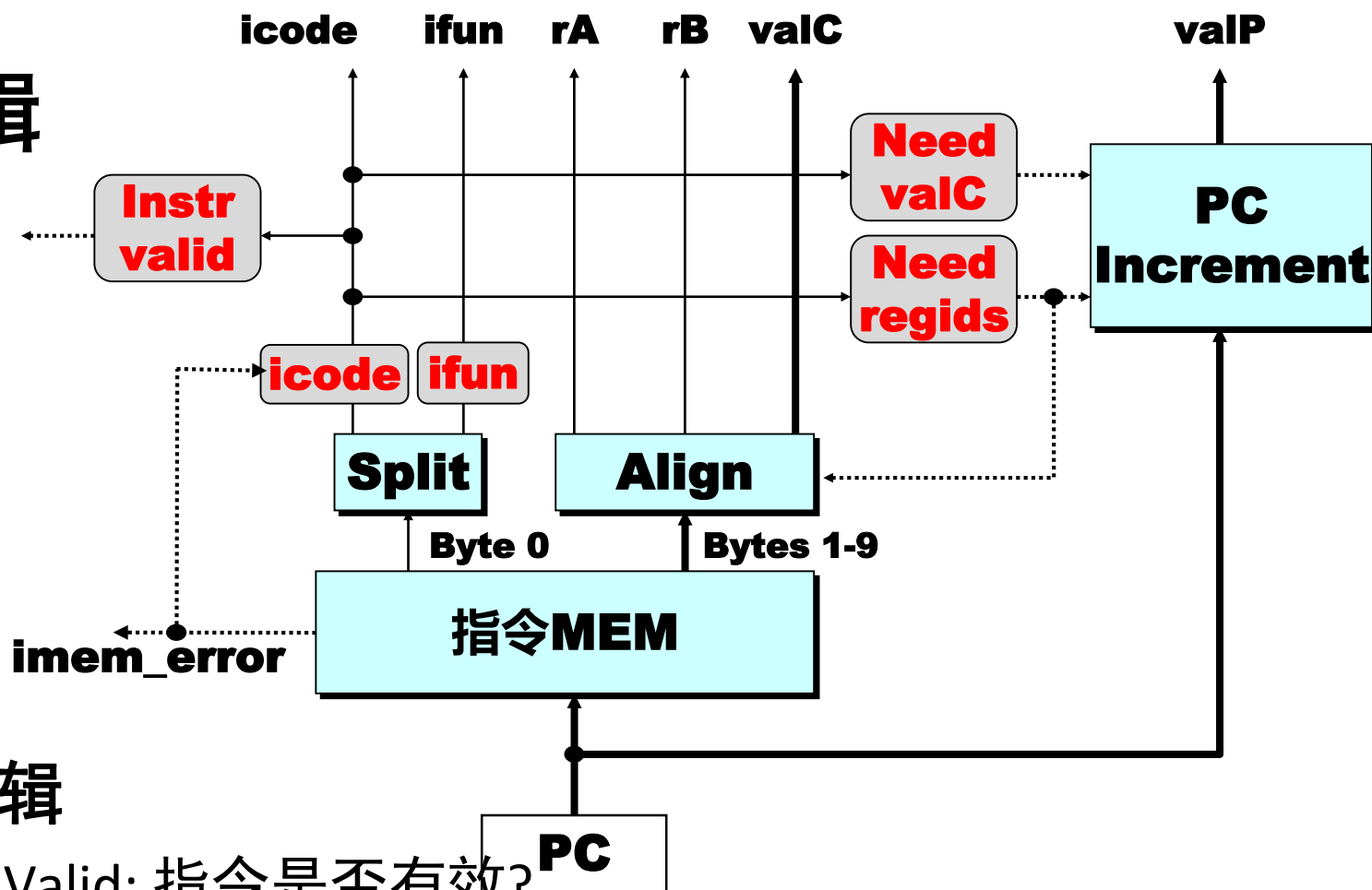
# 取指逻辑



## ■ 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)
  - 发出指令地址不合法的信号imem\_error
- Split: 把指令字节分为icode和ifun
- Align: 把读出的字节放入寄存器和常数字中

# 取指逻辑



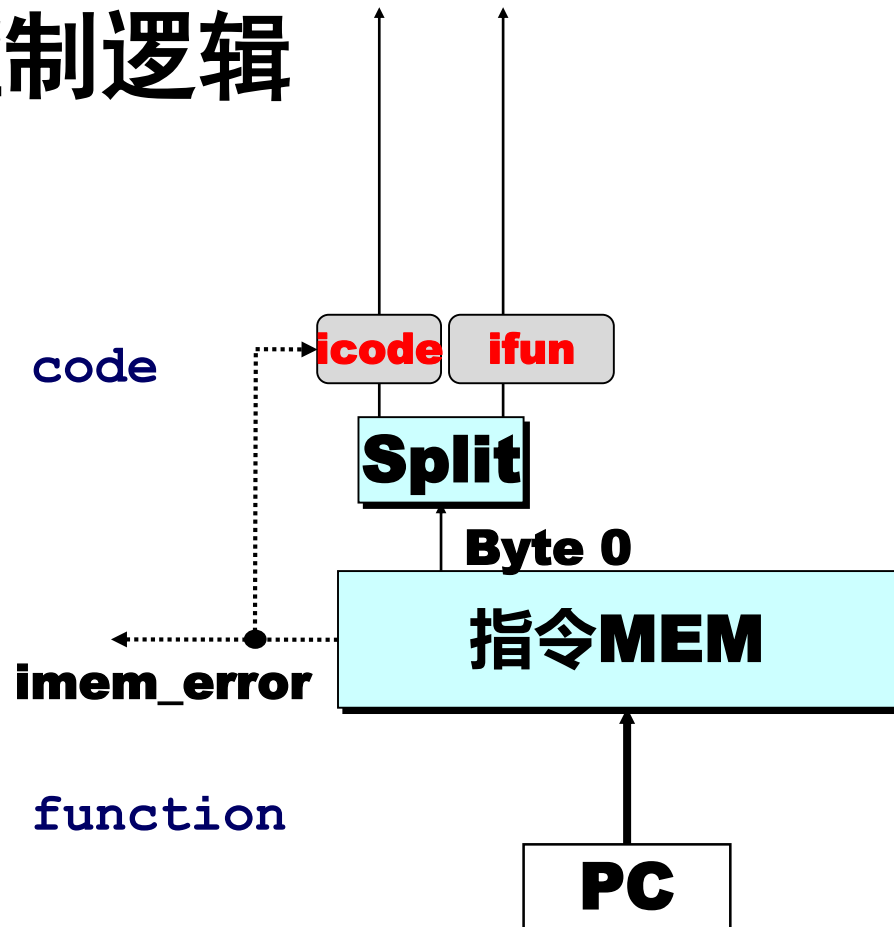
## ■ 控制逻辑

- 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

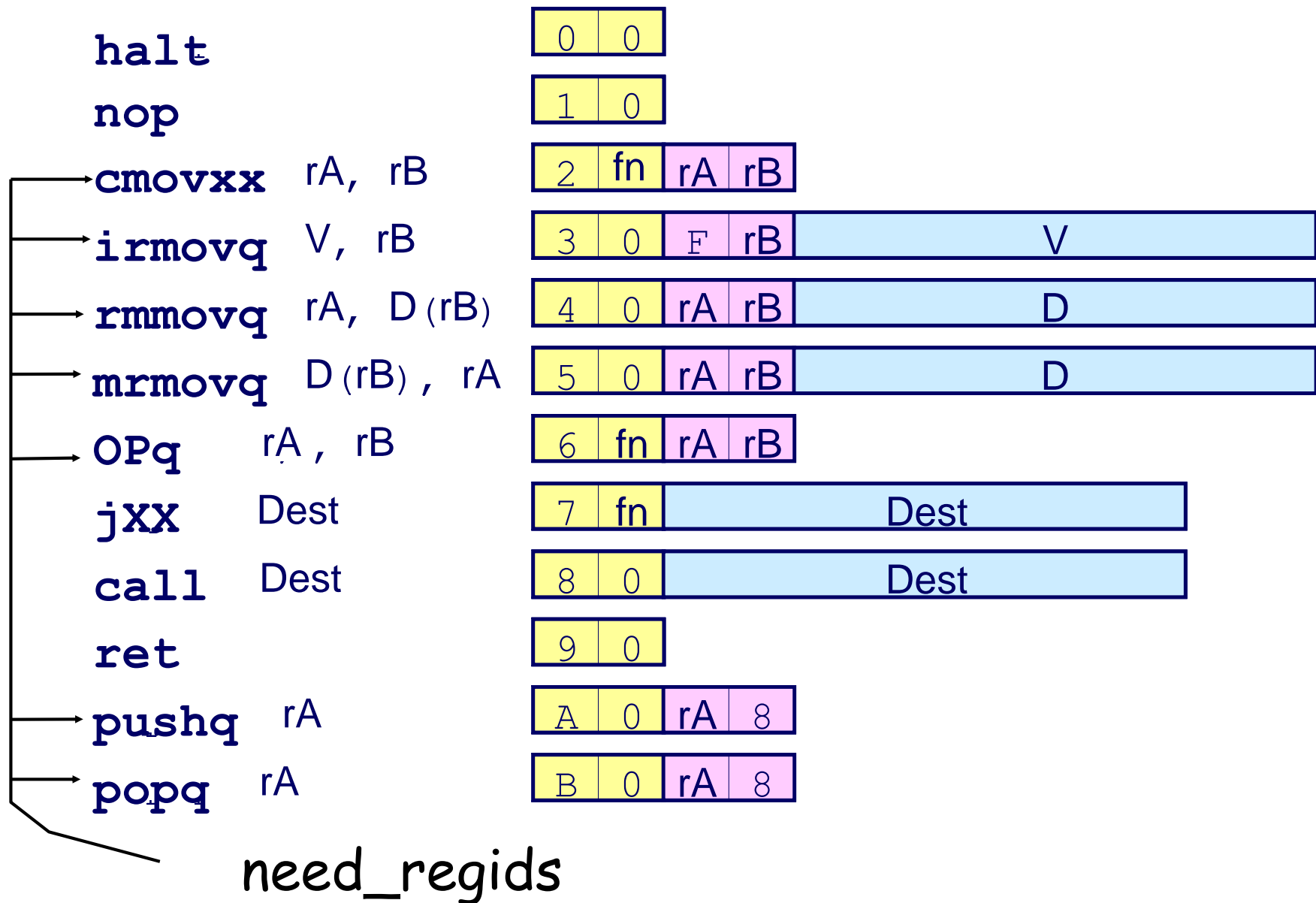
```
# Determine instruction code
int icode = [
    imem_error: INOP;
    1: imem_icode;
];
```

```
# Determine instruction function
int ifun = [
    imem_error: FNONE;
    1: imem_ifun;
];
```



备注：INOP是nop指令的icode并且其值是1，FNONE是默认功能码0

# HCL描述的取指控制逻辑





# 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 };
```

```
bool need_valC = icode in { IIRMOVQ,  
    IRMMOVQ, IMRMVQ, IJXX, ICALL };
```

# 译码逻辑

## ■ 寄存器文件

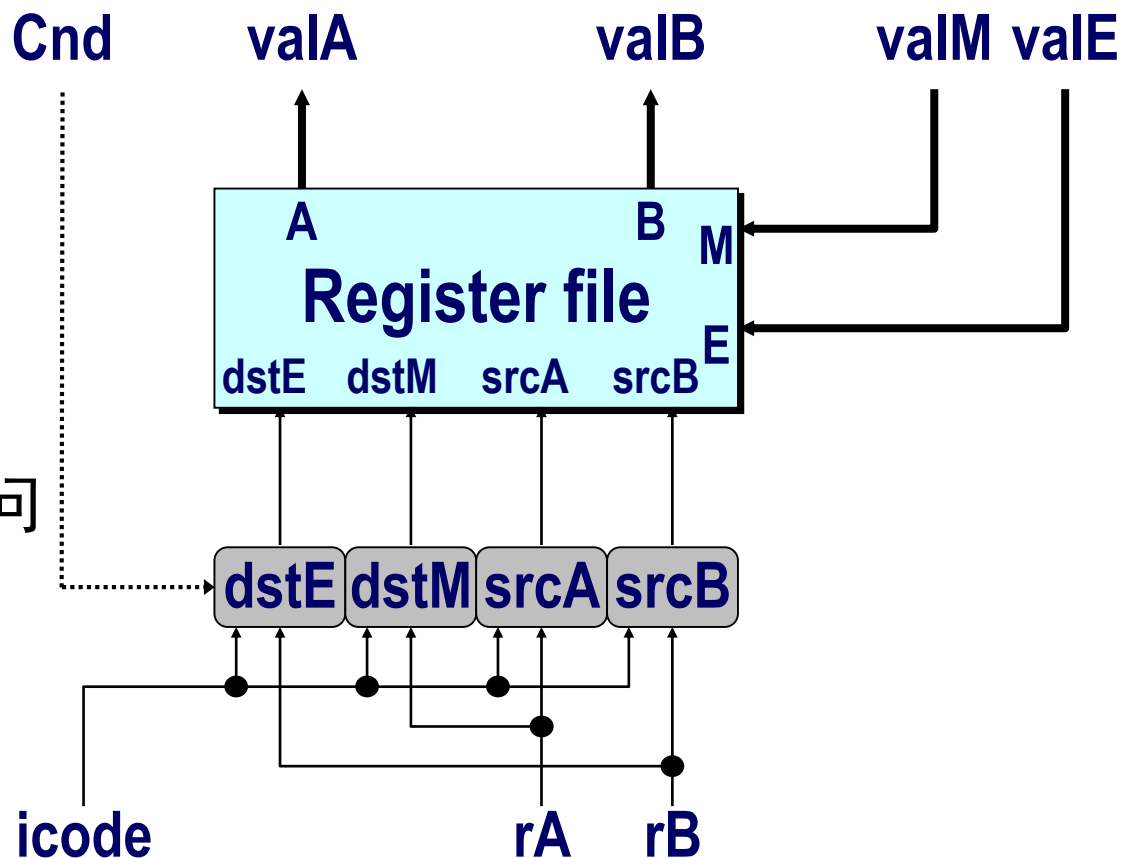
- 读端口 A, B
- 写端口 E, M
- 地址为寄存器的ID  
或 15 (0xF) - 无法访问

## 控制逻辑

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

## 信号

- Cnd: 标明是否触发条件转移 (true/false)
- 在执行阶段计算出Cnd条件信号  $\text{Cond}(\text{CC}, \text{ifun})$



srcA

读RF的第一个端口  
A的地址

	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
	pushq rA	
译码	$valA \leftarrow R[rA]$	读操作数A
	popq rA	
译码	$valA \leftarrow R[\%rsp]$	读栈指针
	ret	
译码	$valA \leftarrow R[\%rsp]$	读栈指针

```

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

```

dstE

写RF的端口E的地址

	OPq rA, rB	
写回	$R[rB] \leftarrow valE$	结果写回
	cmovXX rA, rB	
写回	$R[rB] \leftarrow valE$	有条件的写回结果
	lrmovq V, rB	
写回	$R[rB] \leftarrow valE$	结果写回
	pushq rA	
写回	$R[\%rsp] \leftarrow valE$	更新栈指针
	popq rA	
写回	$R[\%rsp] \leftarrow valE$	更新栈指针
	call Dest	
写回	$R[\%rsp] \leftarrow valE$	更新栈指针
	ret	
写回	$R[\%rsp] \leftarrow valE$	更新栈指针

```
word 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
```

```
];
```

备注：RNONE表示编号F，表示没有寄存器文件写入

```
word srcB = [  
    icode in { IOPQ, IRMMOVQ, IMRMVQ } : rB;  
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;  
    1 : RNONE; # Don't need register  
];
```

```
word dstM = [  
    icode in { IMRMVQ, IPOPQ } : rA;  
    1 : RNONE; # Don't write any register  
];
```

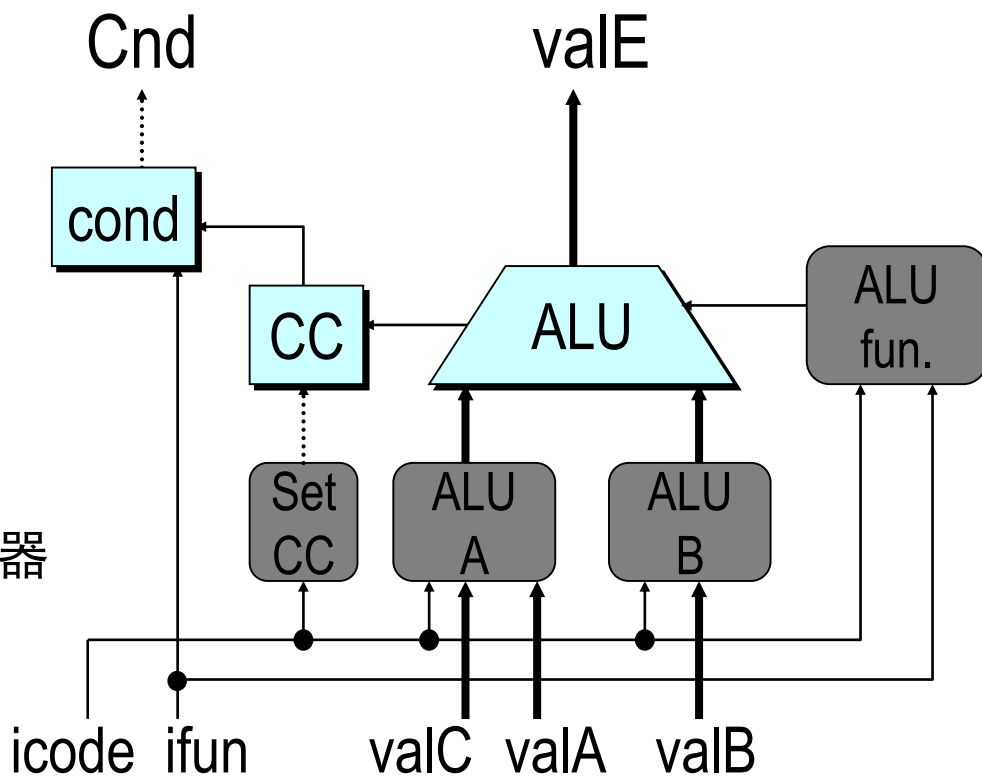
# 执行逻辑

## ■ 单元

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

## ■ 控制逻辑

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



# ALU A 的输入

## ALU的加 数second operand

```
word aluA = [
```

```
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # 其他指令不需要ALU
```

```
];
```

执行	OPq rA, rB $valE \leftarrow valB \text{ OP } valA$
执行	cmovXX rA, rB $valE \leftarrow 0 + valA$
mrmovq D(rB), rA	rmmovq rA, D(rB)
执行	$valE \leftarrow valB + valC$
执行	popq rA $valE \leftarrow valB + 8$
执行	jXX Dest
执行	call Dest $valE \leftarrow valB + -8$
执行	ret $valE \leftarrow valB + 8$
执行	pushq rA $valE \leftarrow valB + -8$

执行ALU的操作

通过ALU传送数据A

irmovq V, rB  
计算有效地址

增加栈指针的值

无操作

减少栈指针的值

增加栈指针的值

减少栈指针的值

# ALU操作

## ALUfun

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

```
word alufun = [
    icode == IOPQ : ifun;
    1 : ALUADD;
];
```



```

word aluB = [
    icode in { IRMMOVQ, IMRMVQ, IOPQ, ICALL,
               IPUSHQ, IRET, IPOPOP }           : valB;
    icode in { IRRMOVQ, IIRMOVQ }               : 0;
    # 其他指令不需要ALU
];

```

```

bool set_cc = icode in { IOPQ };

```

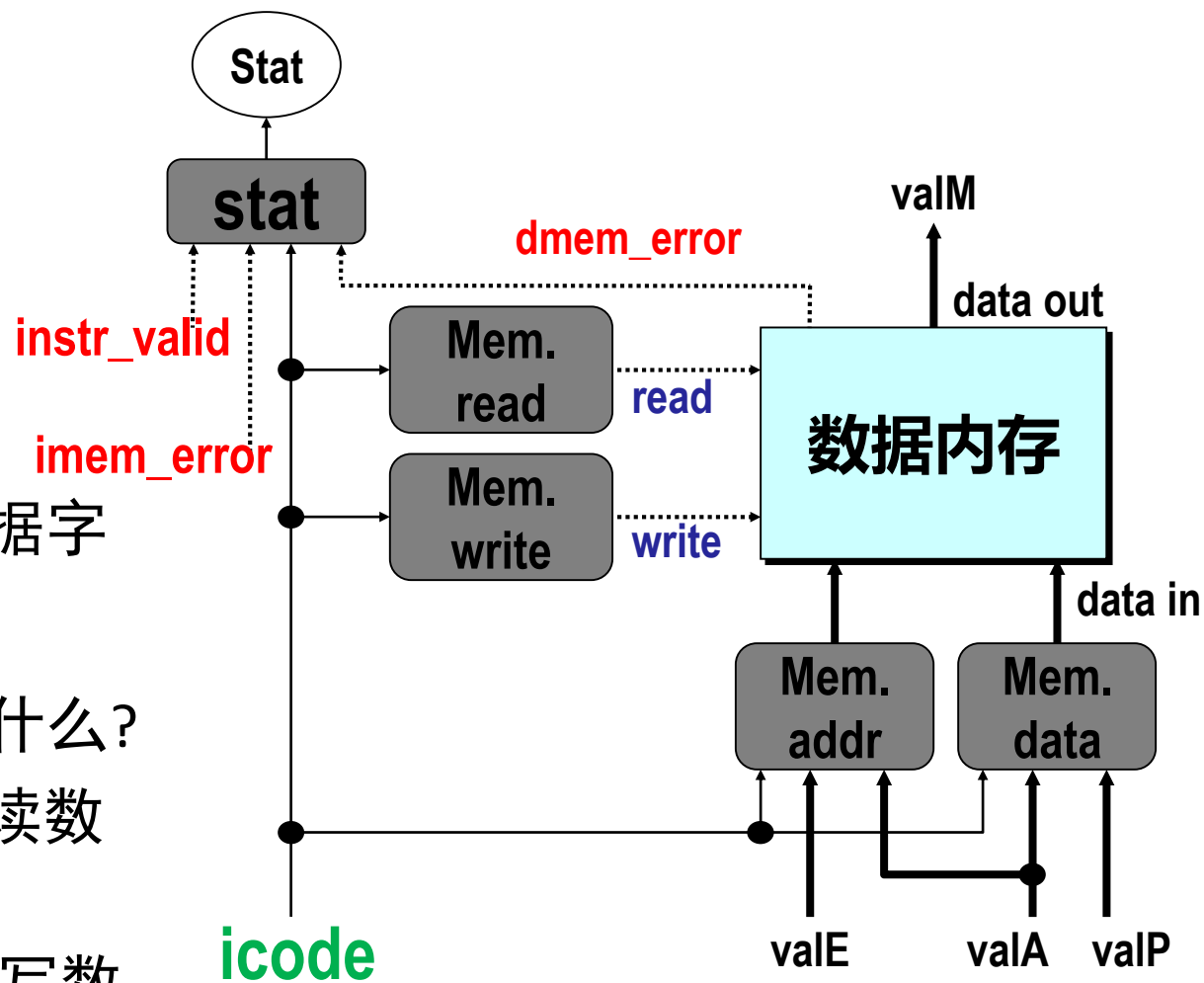
# 访存逻辑

## ■ 访存

- 读写内存里的数据字

## ■ 控制逻辑

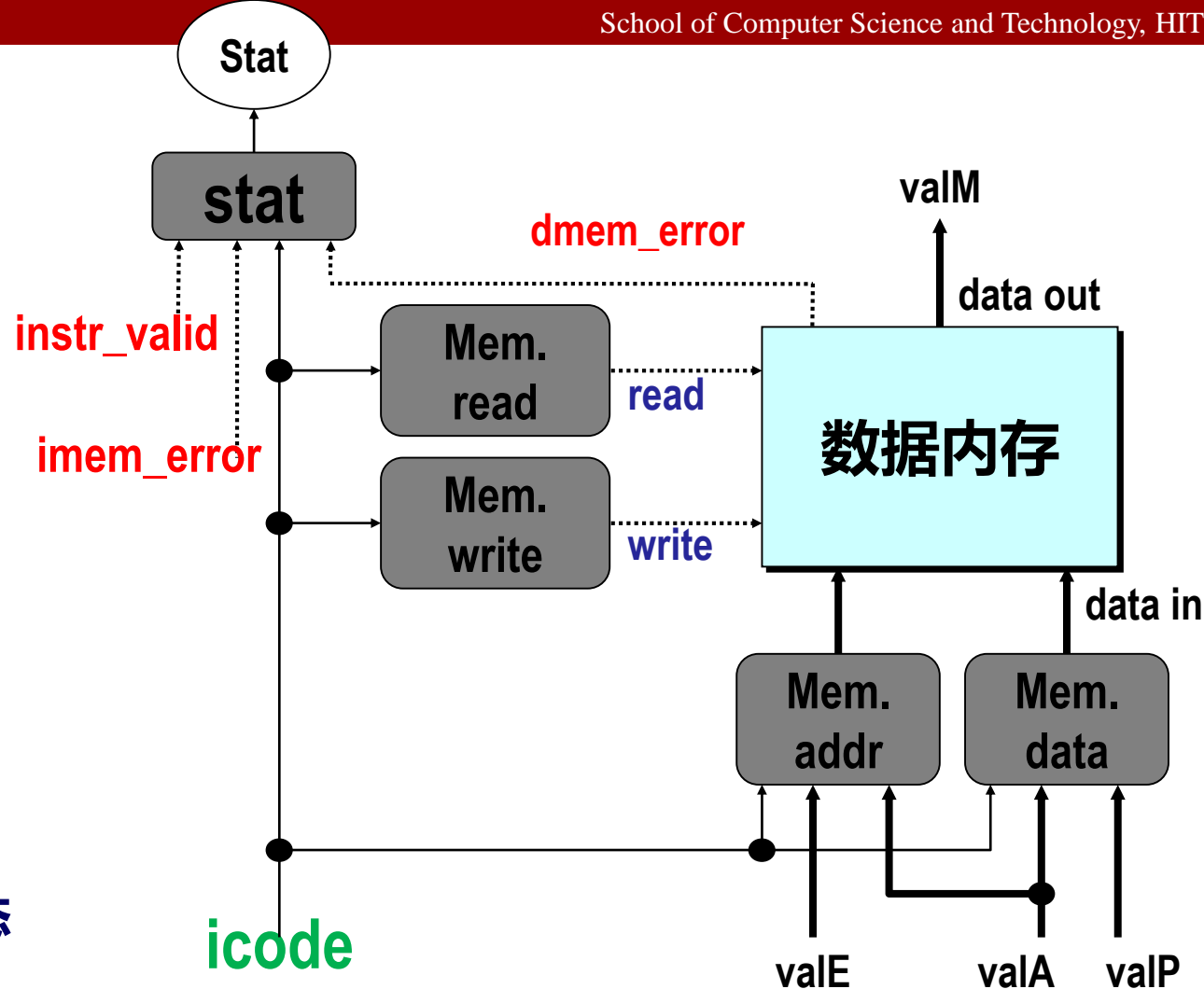
- stat: 指令状态是什么?
- Mem. read: 是否读取数据字?
- Mem. write: 是否写数据字?
- Mem. addr.: 选择地址
- Mem. data.: 选择数据



# 指令状态

## ■ 控制逻辑

- stat: 指令状态是什么?



## ## 决定指令状态

```
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid: SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];
```

SAOK	1	① 正常操作状态码
SADR	2	② 地址异常状态码
SINS	3	③ 非法指令异常状态码
SHLT	4	④ halt 状态码

# 内存地址

	mrmovq D(rB), rA	
访存	$valM \leftarrow M_8[valE]$	读内存数据
	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
	pushq rA	
访存	$M_8[valE] \leftarrow valA$	数据写入内存
	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

```

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

```

# 读内存

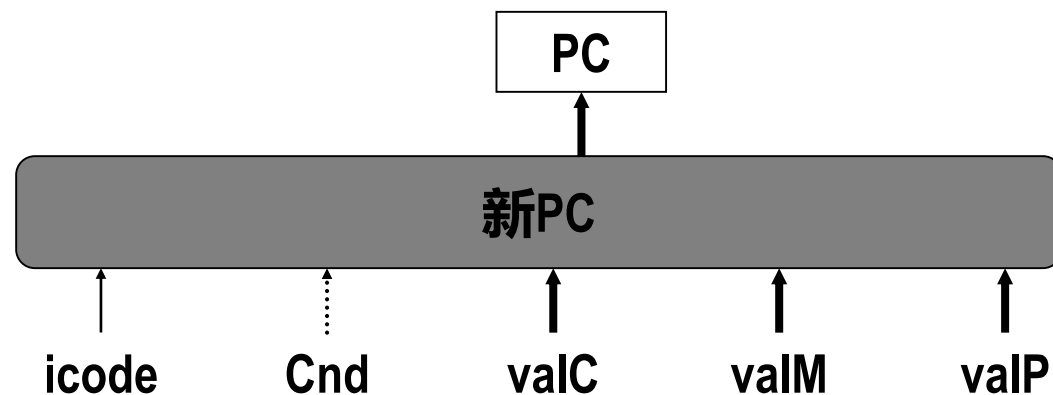
访存	mrmovq D(rB), rA,	从内存读数据
	valM $\leftarrow$ M <sub>8</sub> [valE]	
访存	popq rA	从栈里读取数据
	valM $\leftarrow$ M <sub>8</sub> [valA]	
访存	ret	读返回地址
	valM $\leftarrow$ M <sub>8</sub> [valA]	

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

```
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };

word mem_data = [
    # Value from register
    icode in { IRMMOVQ, IPUSHQ } : valA;
    # Return PC
    icode == ICALL : valP;
    # Default: Don't write anything];
```

# 更新PC的逻辑



## ■ 新PC

- 选取下一个PC的值

# 更新PC

	OPq rA, rB
更新PC	PC $\leftarrow$ valP

更新PC

	rmmovq rA, D(rB)
更新PC	PC $\leftarrow$ valP

更新PC

	popq rA
更新PC	PC $\leftarrow$ valP

更新PC

	jXX Dest
更新PC	PC $\leftarrow$ Cnd ? valC : valP

更新PC

	call Dest
更新PC	PC $\leftarrow$ valC

PC设为目的地址

	ret
更新PC	PC $\leftarrow$ valM

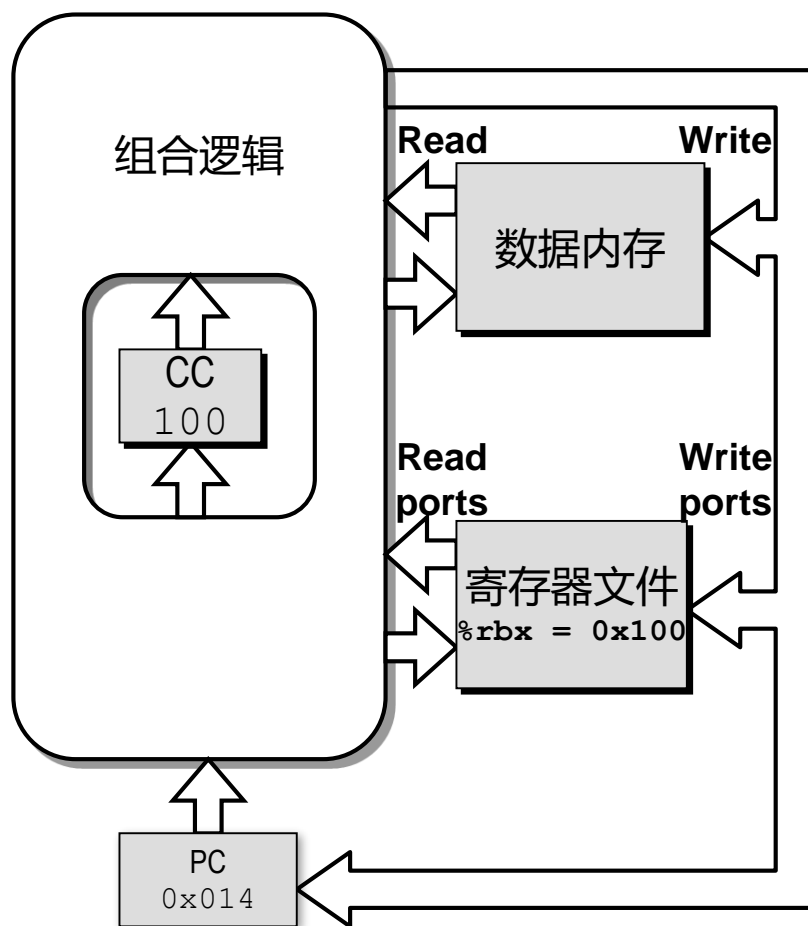
PC设为返回地址

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

注意: Cnd <- Cond(CC, ifun)



# SEQ 操作



## ■ 时序逻辑

- PC寄存器
- 条件码寄存器
- 数据内存
- 寄存器文件

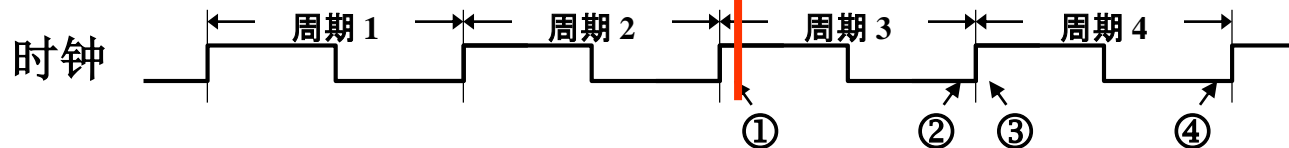
*都在时钟上升沿时更新*

## ■ 组合逻辑

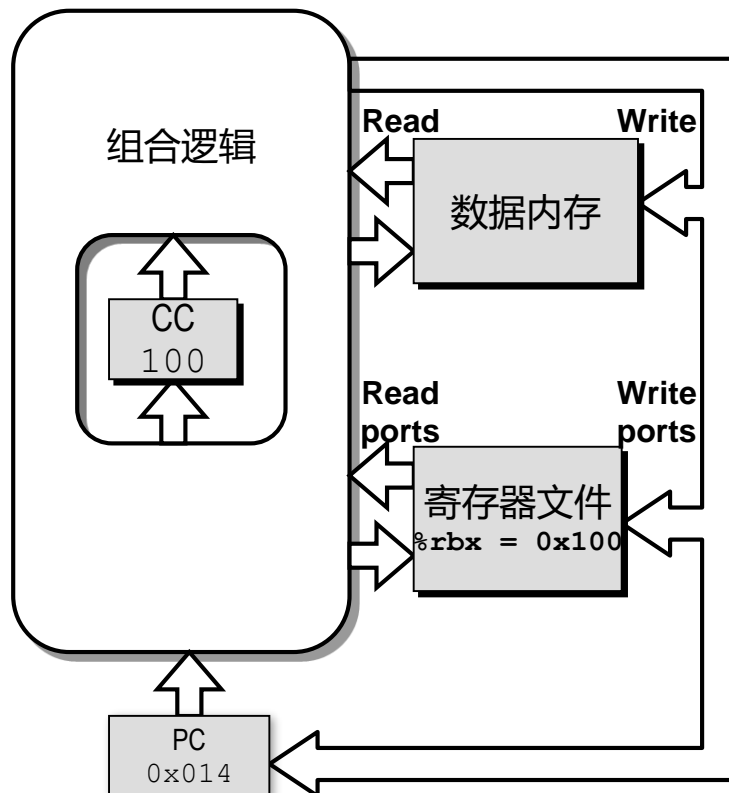
- ALU
- 控制逻辑
- 读操作 (相当于组合逻辑)
  - 指令内存
  - 寄存器文件
  - 数据内存

# SEQ 操作

## #2



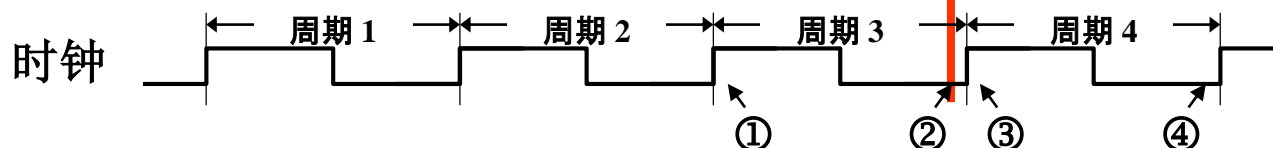
周期 1:	<b>0x000: irmovq \$0x100,%rbx # %rbx &lt;-- 0x100</b>
周期 2:	<b>0x00a: irmovq \$0x200,%rdx # %rdx &lt;-- 0x200</b>
周期 3:	<b>0x014: addq %rdx,%rbx # %rbx &lt;-- 0x300 CC &lt;-- 000</b>
周期 4:	<b>0x016: je dest # Not taken</b>
周期 5:	<b>0x01f: rmmovq %rbx,0(%rdx) # M[0x200] &lt;-- 0x300</b>



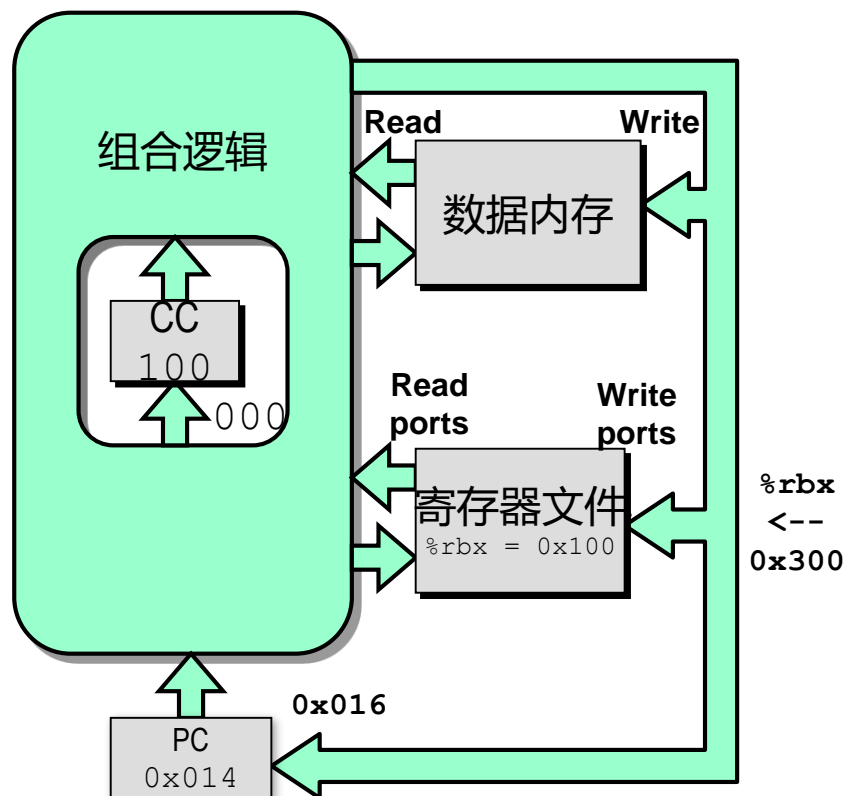
- 依据第二条irmovq指令来设置状态单元-时序逻辑(CLK)
- 组合逻辑开始对状态的变化作出反应

# SEQ 操作

## #3

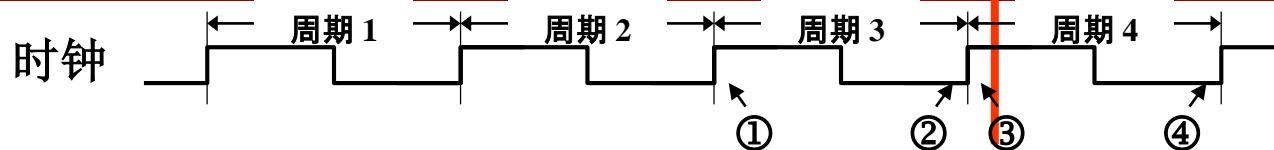


周期 1:	<b>0x000:  irmovq \$0x100,%rbx  # %rbx &lt;-- 0x100</b>
周期 2:	<b>0x00a:  irmovq \$0x200,%rdx  # %rdx &lt;-- 0x200</b>
周期 3:	<b>0x014:  addq %rdx,%rbx      # %rbx &lt;-- 0x300 CC &lt;-- 000</b>
周期 4:	<b>0x016:  je dest             # Not taken</b>
周期 5:	<b>0x01f:  rmmovq %rbx,0(%rdx) # M[0x200] &lt;--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

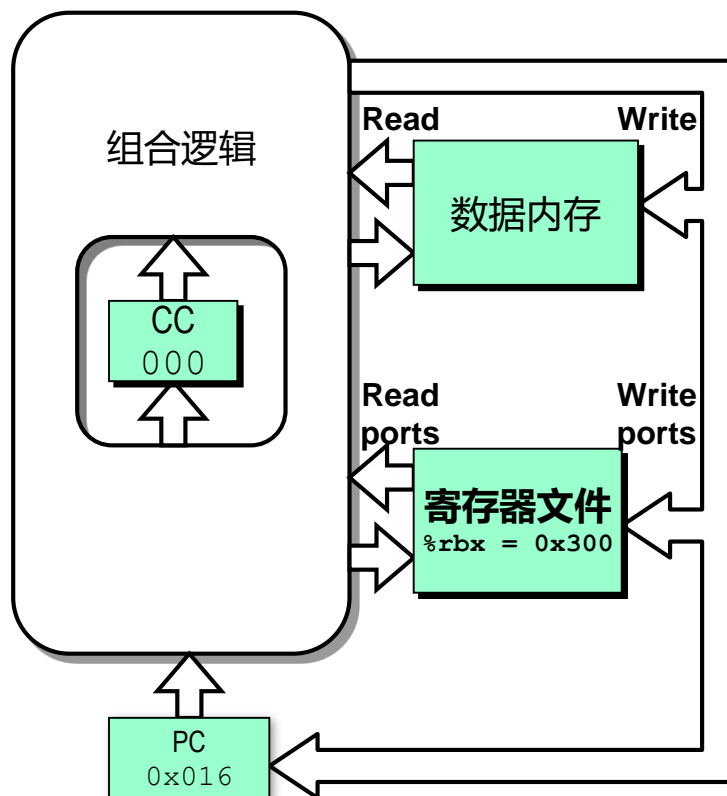


- 依据第二条irmovq指令来设置状态
- 组合逻辑为addq指令生成结果

# SEQ 操作 #4

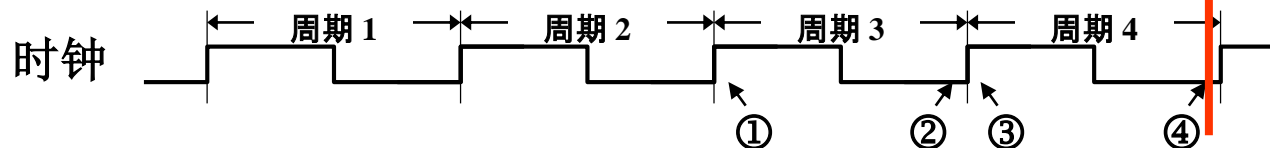


周期 1:	0x000: <code>irmovq \$0x100,%rbx</code> # <code>%rbx &lt;-- 0x100</code>
周期 2:	0x00a: <code>irmovq \$0x200,%rdx</code> # <code>%rdx &lt;-- 0x200</code>
周期 3:	0x014: <code>addq %rdx,%rbx</code> # <code>%rbx &lt;-- 0x300 CC &lt;-- 000</code>
周期 4:	0x016: <code>je dest</code> # Not taken
周期 5:	0x01f: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

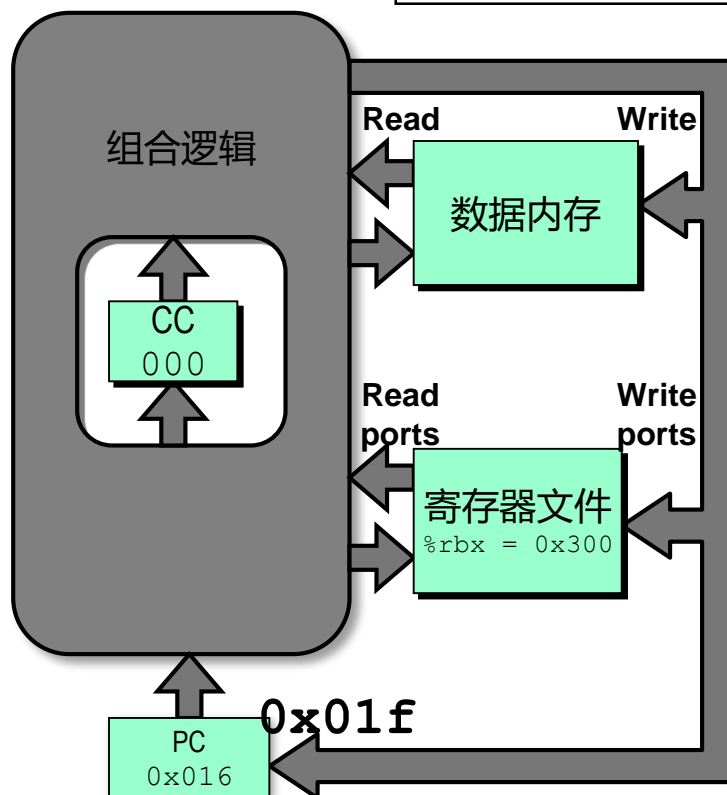


- 依据addq指令设置状态
- 组合逻辑开始对状态的变化作出反应

# SEQ 操作 #5



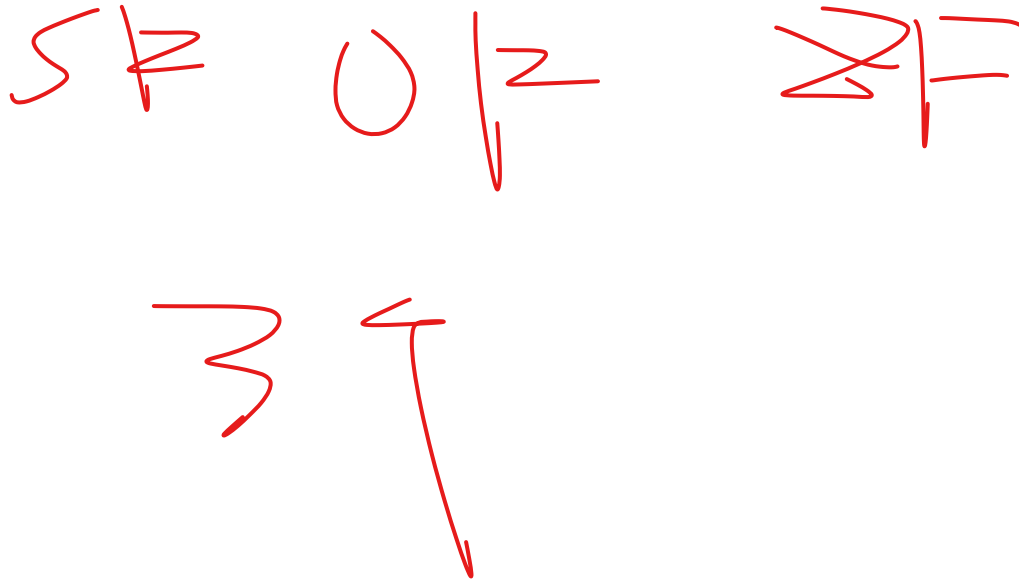
周期 1:	0x000:  irmovq \$0x100,%rbx  # %rbx <-- 0x100
周期 2:	0x00a:  irmovq \$0x200,%rdx  # %rdx <-- 0x200
周期 3:	0x014:  addq %rdx,%rbx      # %rbx <-- 0x300 CC <-- 000
周期 4:	0x016:  je dest             # Not taken
周期 5:	0x01f:  rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300



- 依据addq指令设置状态
- 组合逻辑为je指令生成结果

# SEQ操作例题的总结：

- Mov类指令不改变条件码，只有算术逻辑类指令会有SetCC改变条件码操作。
- 算数逻辑运算每次都会对条件码进行更新
- 时序逻辑器件需要始终上升沿触发才更新里面的数



## 习题

复习PC

1.Y86-64的CPU顺序结构设计与实现中，分成（ **B** ）个阶段

A.5      B.6      C.7      D.8

考试可能没有前两列

2.为Y86-64 CPU增加一指令“iaddq V,rB”，将常量数值V加到寄存器rB。参考irmovq、OPq指令，请设计iaddq指令在各阶段的微操作。(10分)  
大图见下页。

指令	irmovq V,rB	OPq rA, rB	iaddq V,rB
取指	icode:ifun $\leftarrow$ M1[PC]	icode:ifun $\leftarrow$ M1[PC]	
	rA:rB $\leftarrow$ M1[PC+1]	rA:rB $\leftarrow$ M1[PC+1]	
	valC $\leftarrow$ M8[PC+2]		
	valP $\leftarrow$ PC+10	valP $\leftarrow$ PC+2	
译码		valA $\leftarrow$ R[rA]	
	valB $\leftarrow$ 0	valB $\leftarrow$ R[rB]	
执行	valE $\leftarrow$ valB+valC	valE $\leftarrow$ valB OP valA Set CC	
访存			
写回	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE	
更新PC	PC $\leftarrow$ valP	PC $\leftarrow$ valP	

指令	irmovq V,rB	OPq rA, rB	iaddq V,rB
取指	icode:ifun $\leftarrow$ M1[PC]	icode:ifun $\leftarrow$ M1[PC]	icode:ifun $\leftarrow$ M1[PC]
	rA:rB $\leftarrow$ M1[PC+1]	rA:rB $\leftarrow$ M1[PC+1]	rA:rB $\leftarrow$ M1[PC+1]
	valC $\leftarrow$ M8[PC+2]		valC $\leftarrow$ M8[PC+2]
	valP $\leftarrow$ PC+10	valP $\leftarrow$ PC+2	valP $\leftarrow$ PC+10
译码		valA $\leftarrow$ R[rA]	
	valB $\leftarrow$ 0	valB $\leftarrow$ R[rB]	valB $\leftarrow$ R[rB]
执行	valE $\leftarrow$ valB+valC	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ valB+valC Set CC
访存			<del>算数指令</del>
写回	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE
更新PC	PC $\leftarrow$ valP	PC $\leftarrow$ valP	PC $\leftarrow$ valP



# SEQ 总结

## ■ 实现

- 把每条指令表示成一个特殊的阶段序列
- 每种指令类型都遵循统一的序列
- 把寄存器、内存、预设的硬件单元整合到指令的执行过程中
- 再在这个过程中嵌入控制逻辑

## ■ 不足的地方

无流水线

- 实际使用起来太慢
- 信号必须能在一个周期内传播所有的阶段，其中要经过指令内存、寄存器文件、ALU以及数据内存等
- 时钟必须非常慢
- 硬件单元只在时钟周期的一部分时间内被使用

***Enjoy!***