Processor Arch:SEQ

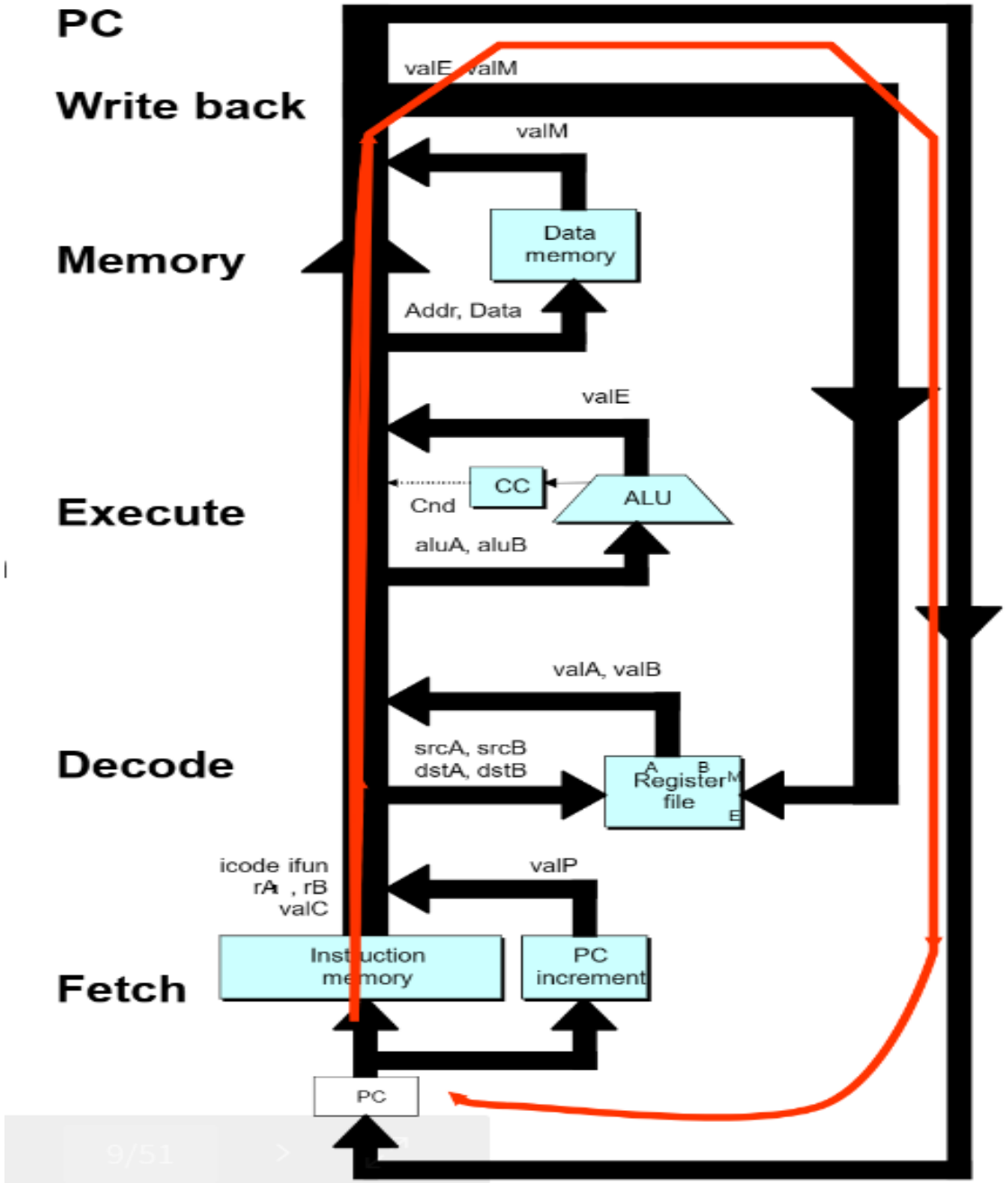
eecs_havefun

目录

- 定义
- 硬件结构与操作
- 硬件实现与时序
- 分阶段实现（HCL）
- 练习
- 数据通路和控制

定义

SEQ：顺序处理器
将每条指令的执行都组织成6个阶段：取指，译码，执行，访存，写回，更新PC，然后顺序执行。每条指令执行完成后顺序执行下一条指令。



SEQ的硬件结构

浅蓝：硬件单元
白色：时钟寄存器
灰色：控制逻辑块
圆圈：线路的标识

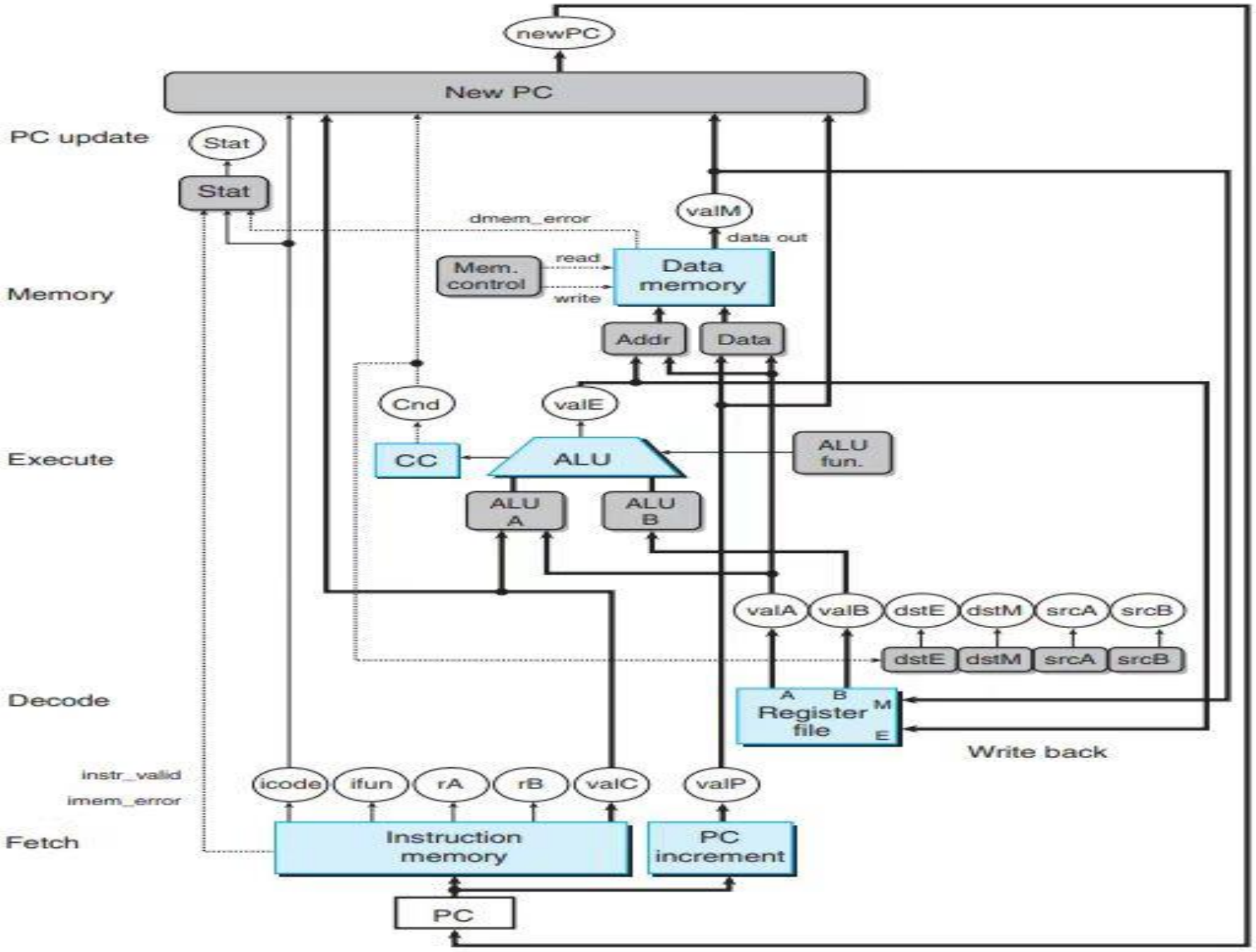


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the

RF：程序寄存器

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

CC：条件码

ZF	SF	OF

Stat：程序状态

PC

DMEM：内存

阶段	计算	OPq rA, rB	mrmovq D(rB), rA
取指	icode:ifun rA, rB valC valP	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA, srcA valB, srcB	valA ← R[rA] valB ← R[rB]	 valB ← R[rB]
执行	valE Cond. codes	valE ← valB OP valA Set CC	valE ← valB+valC
访存	Read/write		valM ← M_8[valE]
写回	E port, dstE M port, dstM	R[rB] ← valE	 R[rA] ← valM
更新 PC	PC	PC ← valP	PC ← valP

图 4-24 标识顺序实现中的不同计算步骤。第二栏标识出 SEQ 阶段中正在被计算的值，或正在被执行的操作。以指令 OPq 和 mrmovq 的计算作为示例

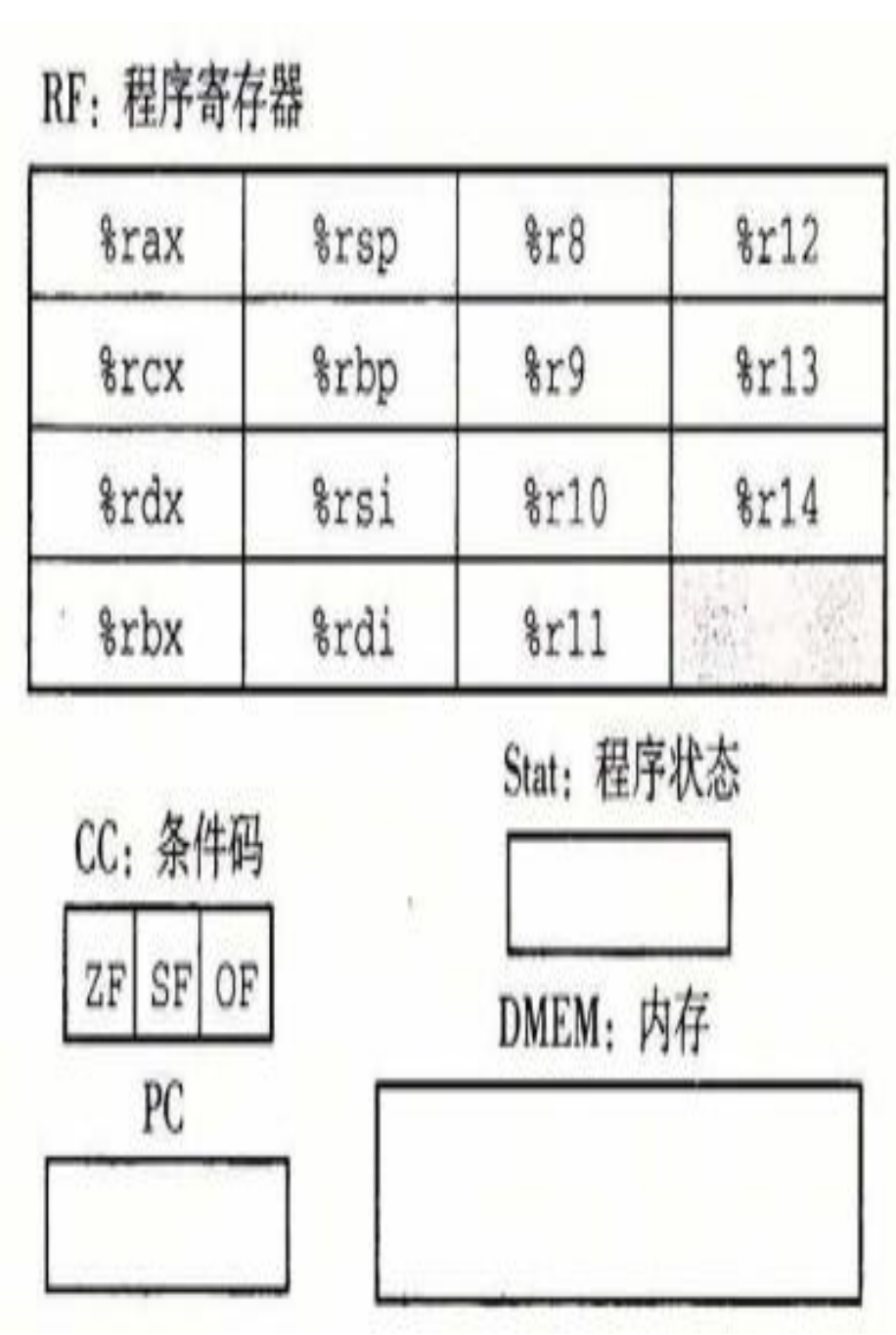
我们的目标：通过指令修改程序员可见的状态

指令拆分成上图所示的阶段，我们需要通过硬件设计实现我们想要完成的操作

各个值、操作的含义

阶段	计算
取指	icode:ifun rA,rB valC valP
译码	valA,srcA valB,srcB
执行	valE Cond. codes
访存	Read/write
写回	E port, dstE M port, dstM
更新 PC	PC

- icode:指令代码（0-b）
- ifun:指令功能
- valC:读入的常数
- valP:保存下条指令的地址
- srcA,srcB：某个寄存器或者为空
- valA/B:从srcA,srcB中读出的数
- valE:经过ALU得到的值
- Cond.codes:修改条件码（Y86中只在有整数操作时进行）
- Read/Write:将前述的某个值写入某内存地址或者从某地址中读出valM。
- dstE:根据指令内容valE应当被写入的寄存器
- dstM:根据指令内容valM应当被写入的寄存器
- PC：保存下一条要执行指令的地址



Stat:

1 AOK 正常操作

2 HLT 遇到halt指令

3 ADR 遇到非法地址

4 INS 遇到非法指令

CC:

ZF： 0

SF： <0

OF:有符号溢出

SF xor OF :判断有符号数是否小于0

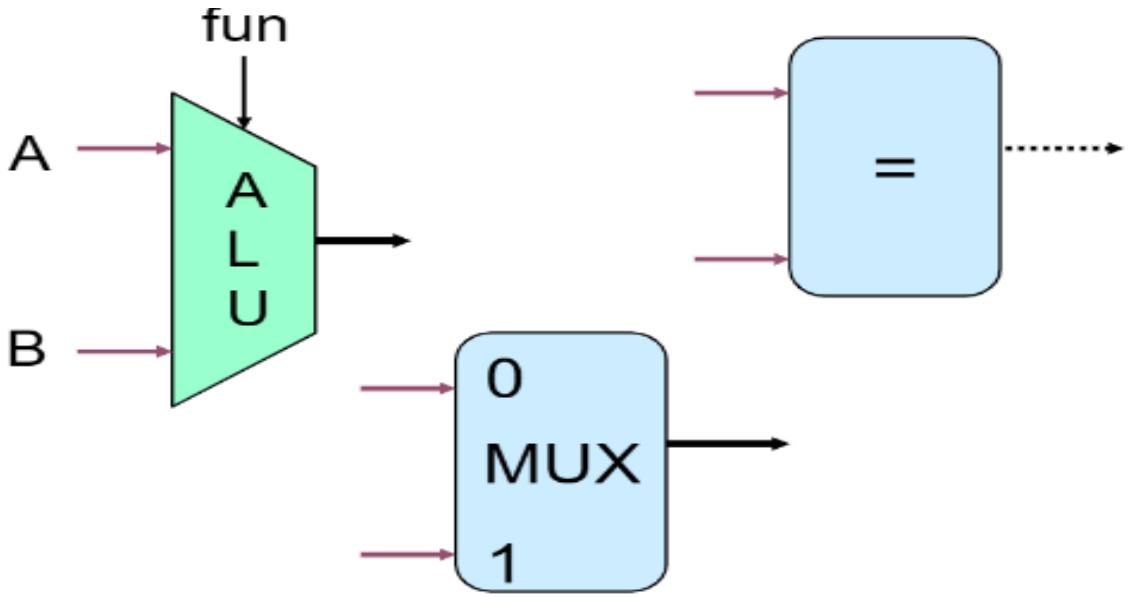
寄存器： 8字节的数

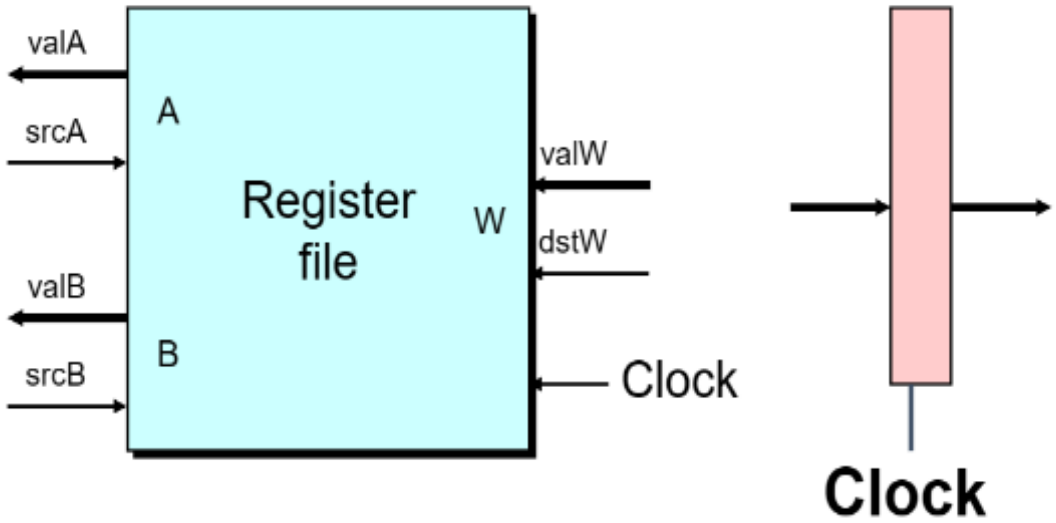
PC： 下一条要执行指令的地址

SEQ的实现以及时序:

组合逻辑;
时钟寄存器（程序计数器和条件码寄存器）；
随机访问存储器（寄存器文件，指令内存，数据内存）；

其中程序计数器，条件码寄存器，寄存器文件，数据内存是状态单元





时序的实现以及原则

要控制处理器中活动的时序，只需要寄存器和内存的时钟控制。硬件获得了如图 4-18～图 4-21 的表中所示的那些赋值顺序执行一样的效果，即使所有的状态更新实际上同时发生，且只在时钟上升开始下一个周期时。之所以能保持这样的等价性，是由于 Y86-64 指令集的本质，因为我们遵循以下原则组织计算：

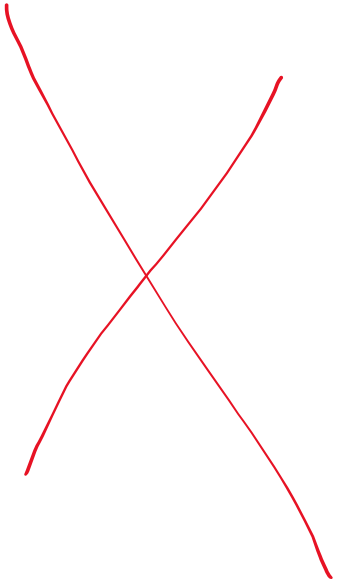
原则：从不回读

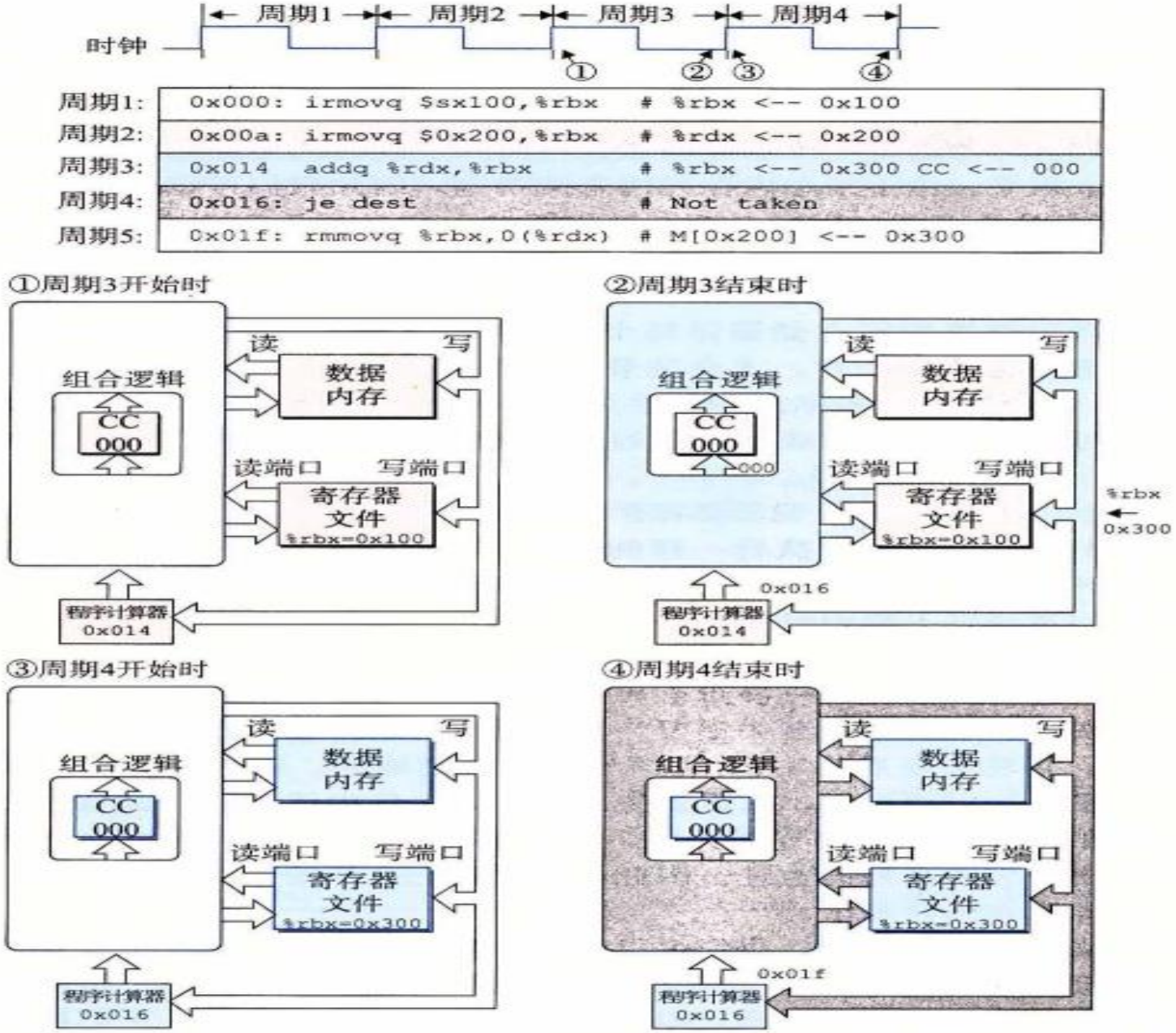
处理器从来不需要为了完成一条指令的执行而去读由该指令更新了的状态。

例子：pushq指令，条件码的设置

阶段	通用
	pushq rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
译码	valA ← R[rA] valB ← R[%rsp]
执行	valE ← valB+(−8)
访存	M_8[valE] ← valA
写回	R[%rsp] ← valE
更新 PC	PC ← valP

- pushq rA
- icode:ifun <--- M1[PC]
- rA:rB <--- M1[PC+1]
- valP <--- PC+2
- Step1:
- valB <--- R[%rsp]
- valE <--- valB + (-8)
- R[%rsp] <--- valE
- Step2:
- valA <--- R[rA]
- valB <--- R[%rsp]
- M8[valB]<---valA
- PC<--- valP





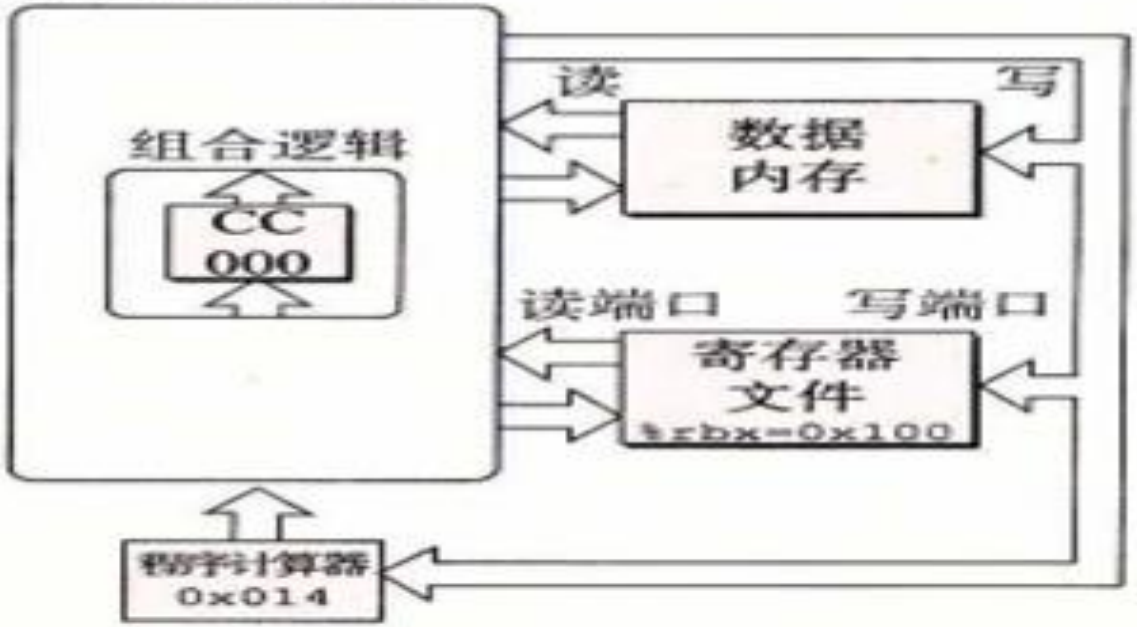
上个时钟周期末组合逻辑产生了所有应该发生的变化并传回给各个状态单元

下个时钟周期开始时状态单元中的内容才正式发生变化

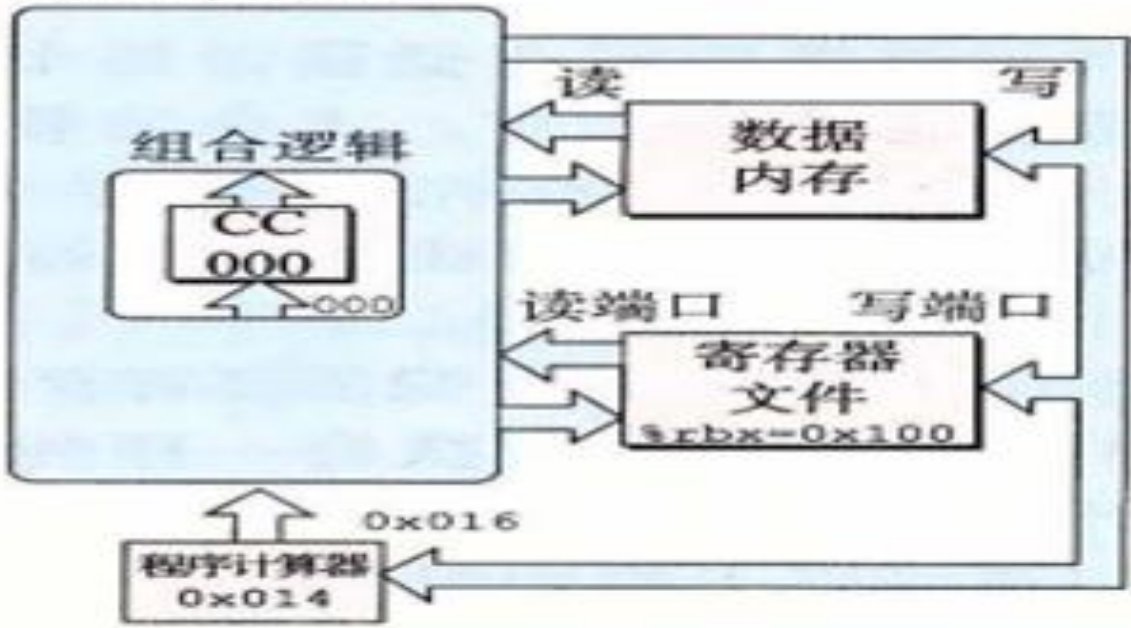
时钟控制状态单元的更新，时钟上升是新的周期开始以及旧的周期结束

1-25 跟踪 SEQ 的两个执行周期。每个周期开始时，状态单元(程序计数器、条件码寄存器、寄存器文件以及数据内存)是根据前一条指令设置的。信号传播通过组合逻辑，创建出新的状态单元的值。在下一个周期开始时，这些值会被加载到状态单元中

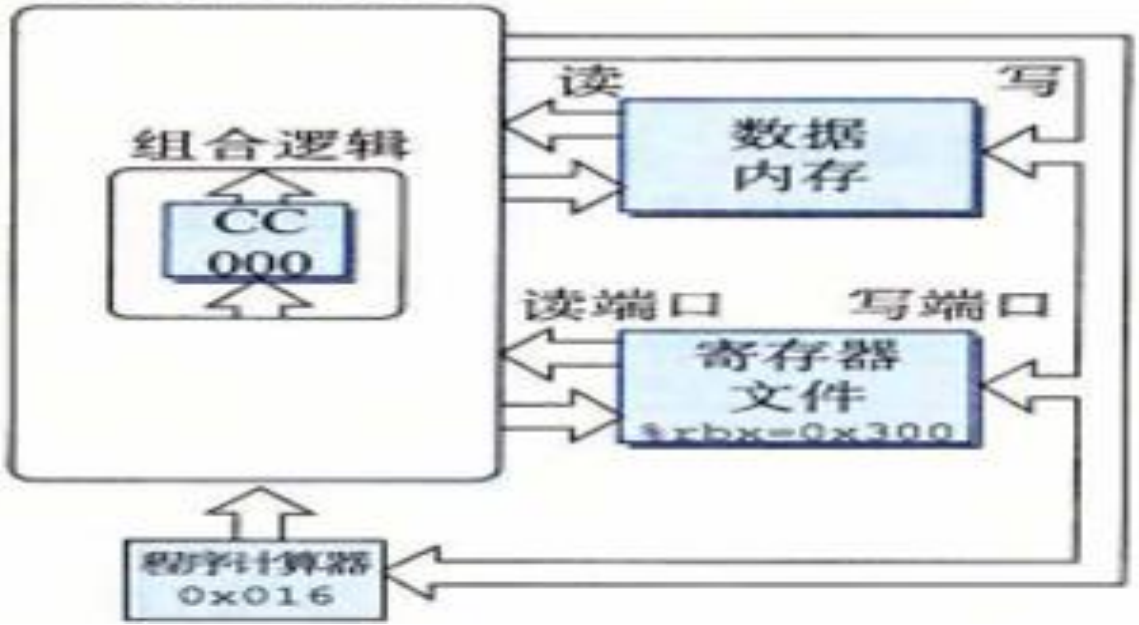
①周期3开始时



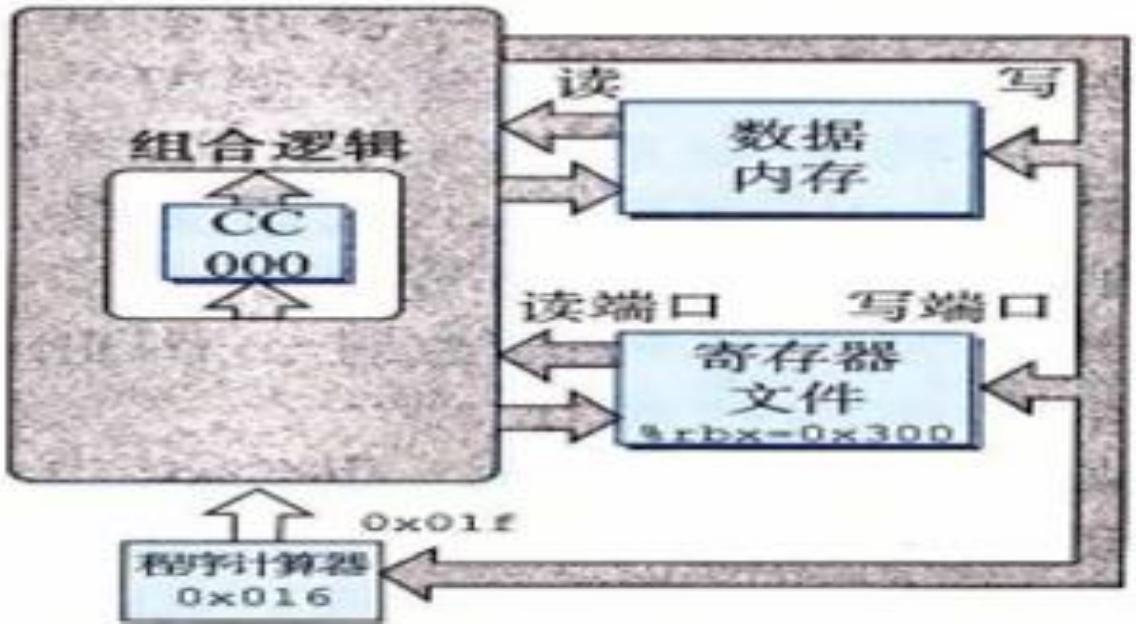
②周期3结束时



③周期4开始时



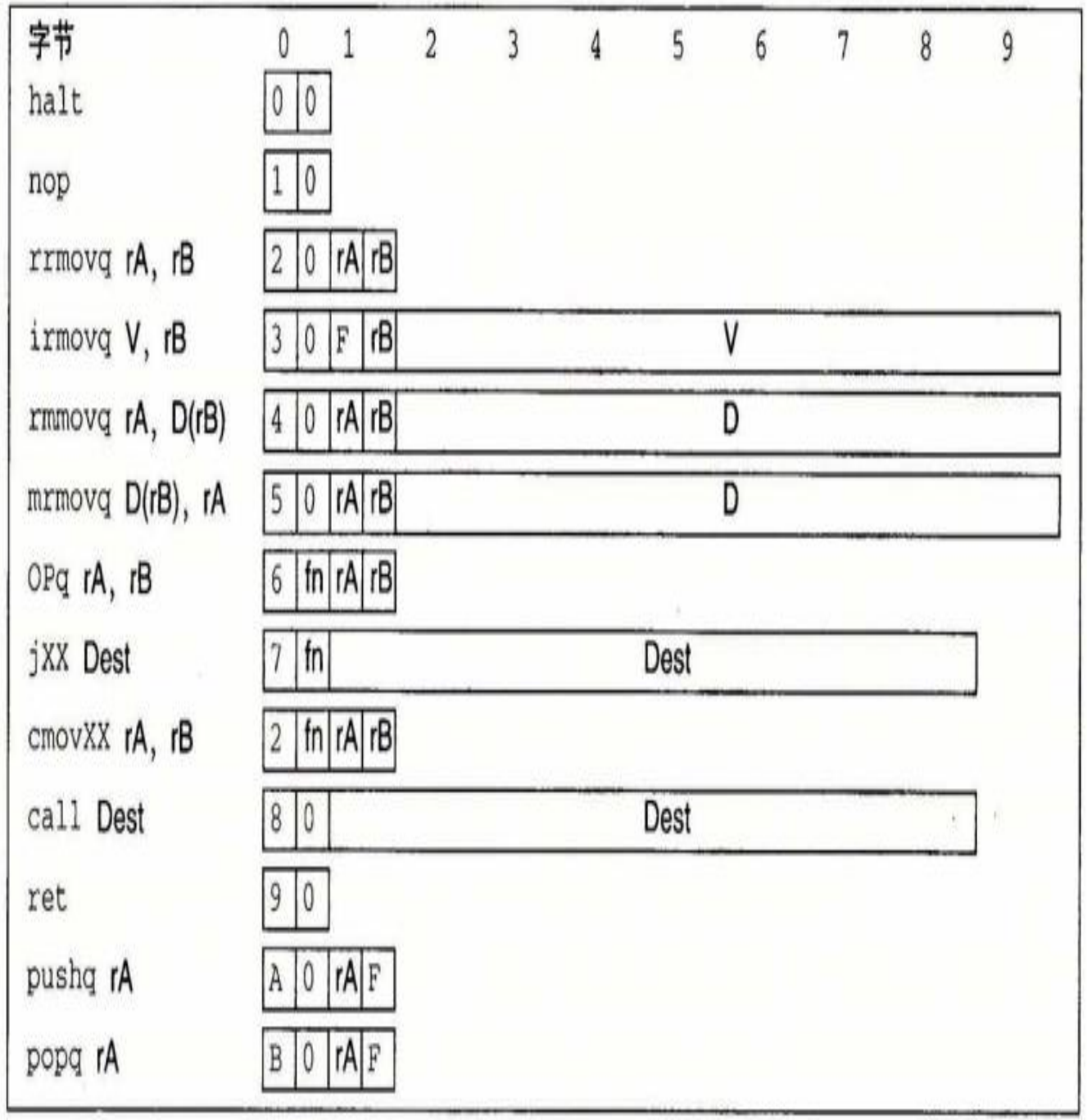
④周期4结束时



分阶段实现（HCL）

- HCL（硬件控制语言）
- 本节会设计实现SEQ所需要的控制逻辑块的HCL描述

名称	值(十六进制)	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	mrmovq 指令的代码
IOPL	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

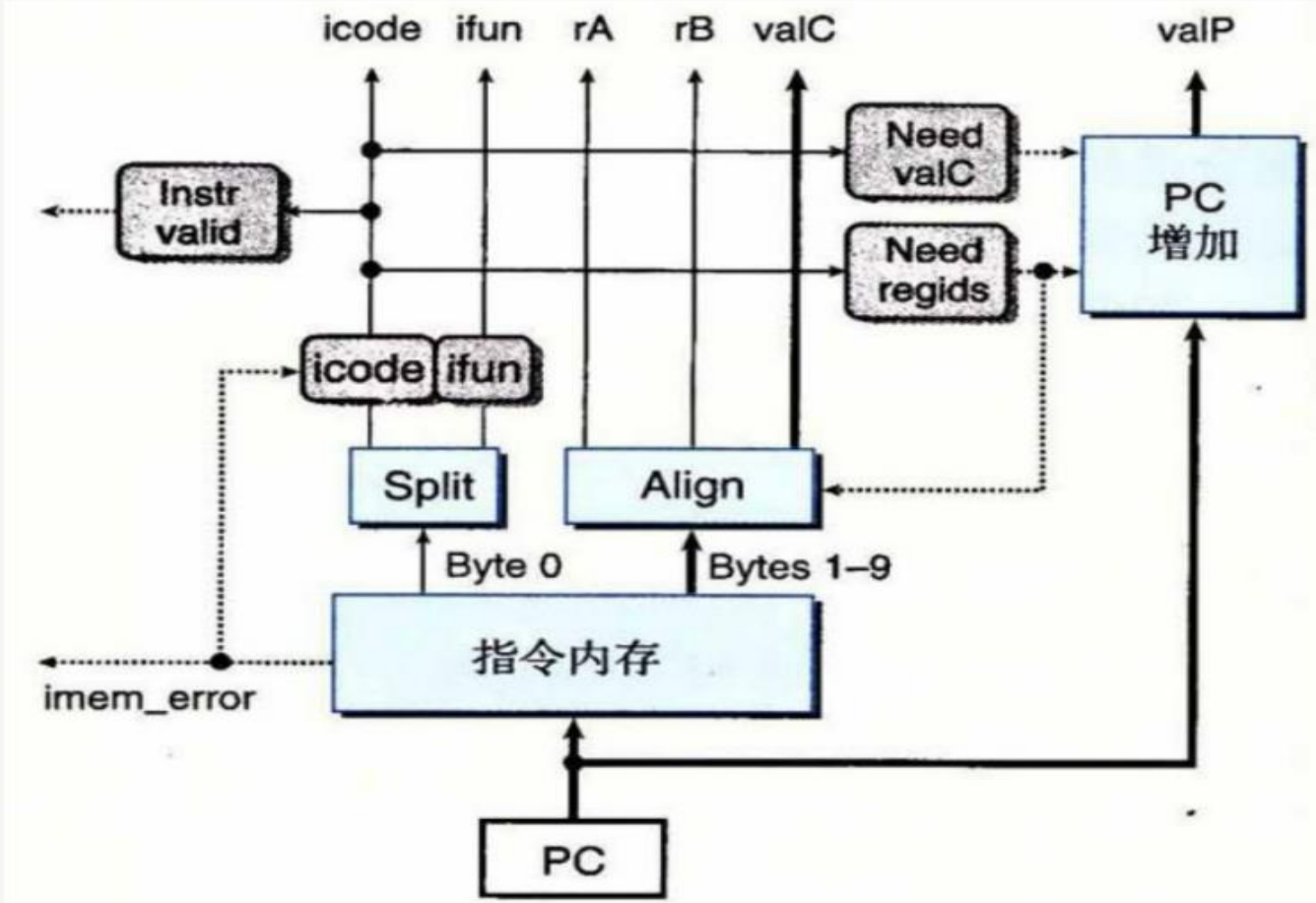


取指

Imem_error:指令地址的合法性
Instr_valid:指令的合法性
Need valC:指令是否包含常数
Need regids:指令是否包含寄存器指示字节
Align:如果Need regids=1,第一个字节的高4位为rA,低4位为rB
Split:如果不出现指令越界，第0个字节的高4位为icode,低4位为ifun

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

PC增加器硬件单元根据当前的PC以及两个信号need_regids和need_valC的值，产生信号valP。对于PC值p、need_regids值r以及need_valC值i，增加器产生值$p+1+r+8i$。



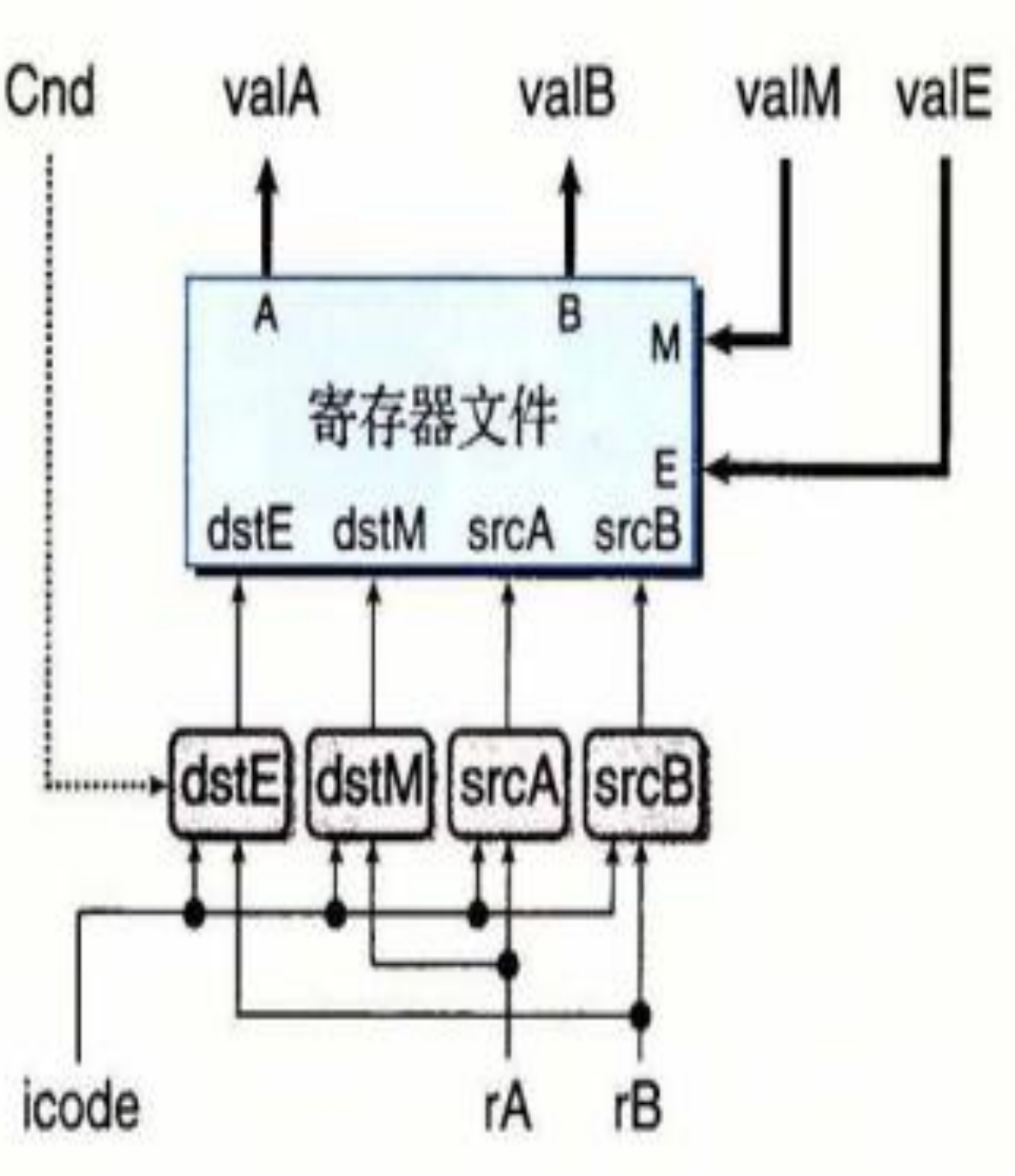
阶段	计算	OPq rA, rB	mrmovq D(rB), rA
取指	icode:ifun rA, rB valC valP	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10

译码

从读端口srcA，srcB读入寄存器标识符（若为0xF表示不需要访问寄存器），并读出valA,valB。

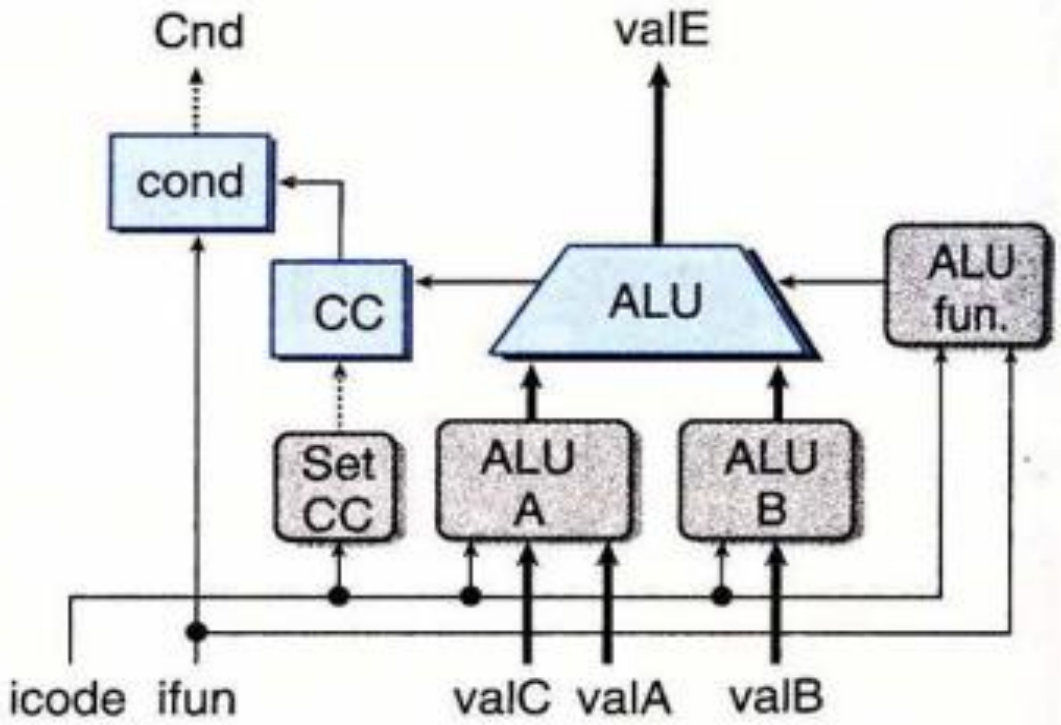
```
word srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```

译码	valA,srcA valB,srcB	valA ← R[rA] valB ← R[rB]	valB ← R[rB]



执行

ALUfun是给ALU提供的信号，由icode,ifun决定
ALUA和ALUB的来源通过icode决定



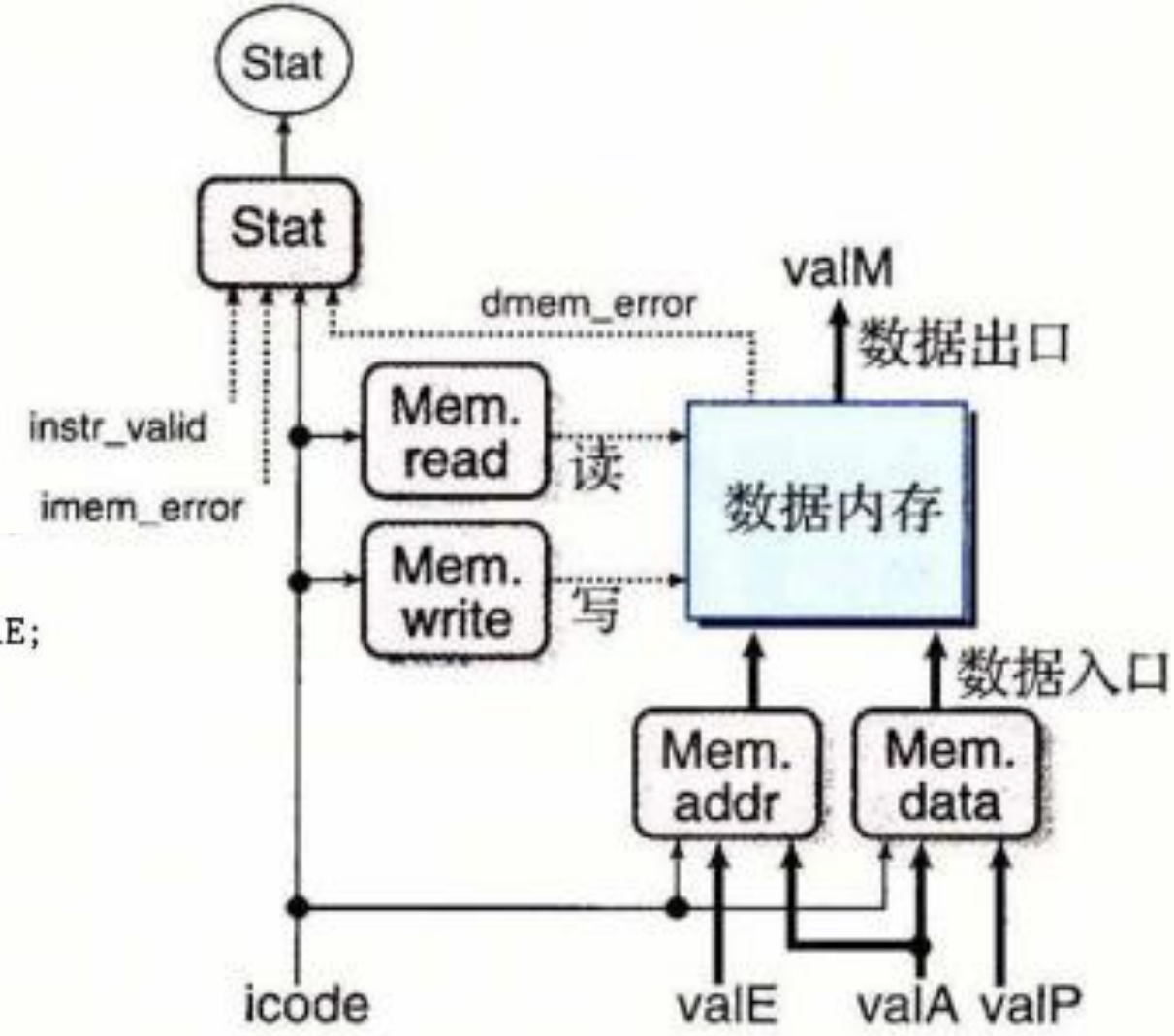

```
word alufun = [                    bool set_cc = icode in { IOPQ };
        icode == IOPQ : ifun;
        1 : ALUADD;
];
```

```
word aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];
```

```
word aluB = [
        icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                   IPUSHQ, IRET, IPOPQ } : valB;
        icode in { IRRMOVQ, IIRMOVQ } : 0;
        # Other instructions don't need ALU
];
```

访存

- Mem.read Mem.write是控制读写的信号，Mem.addr是内存地址，Mem.data是要写的数据




```
word mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];
  word mem_data = [
          # Value from register
          icode in { IRMMOVQ, IPUSHQ } : valA;
          # Return PC
          icode == ICALL : valP;
          # Default: Don't write anything
  ];
```

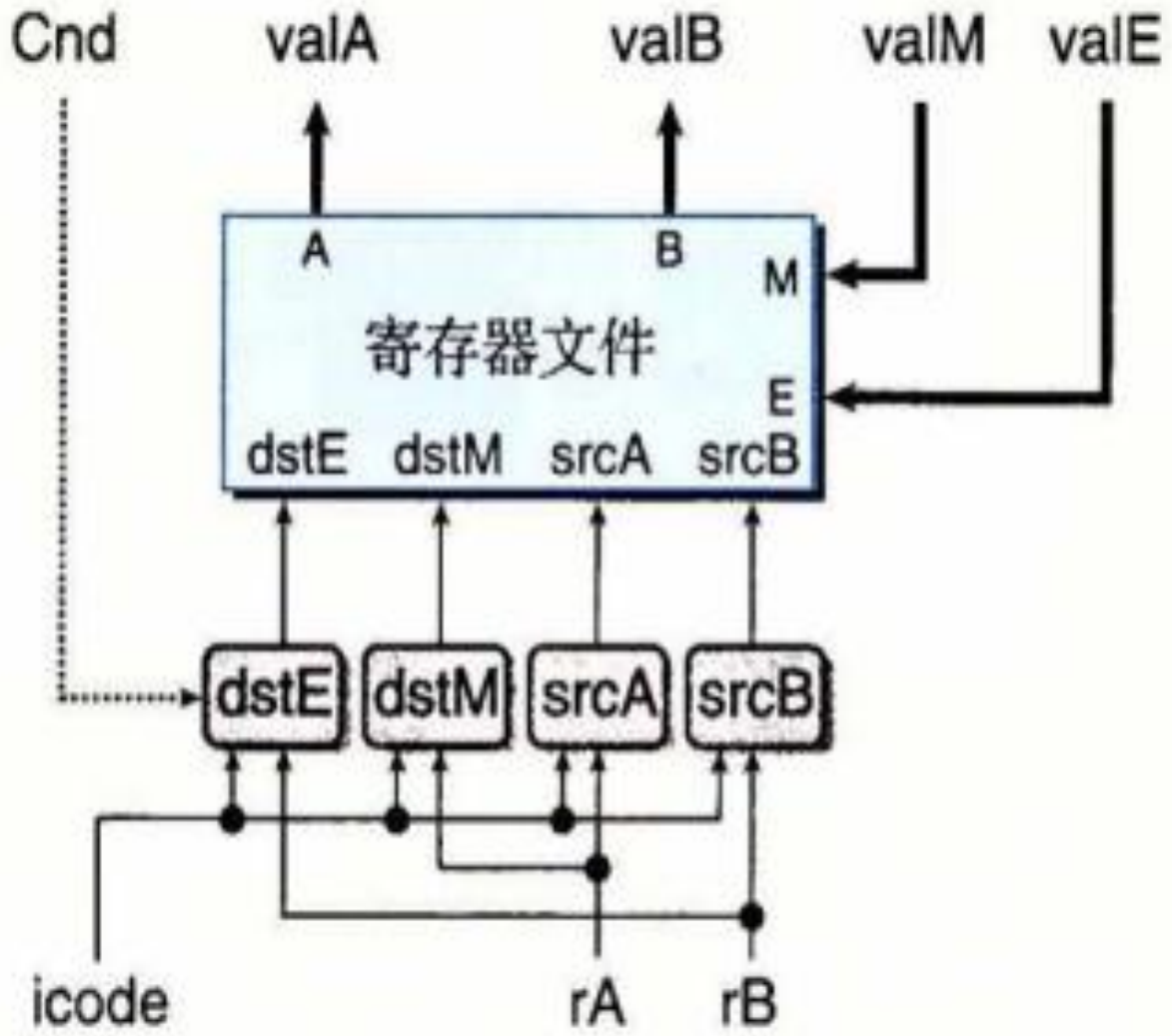
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };

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

写回

- 根据指令类型以及执行产生的Cnd确定dstE以及dstM，写入valE,valM。
- Cnd=Cond(CC,ifun)
- 不考虑cnd时hcl代码如下

```
word dstE = [
        icode in { IRRMOVQ } : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

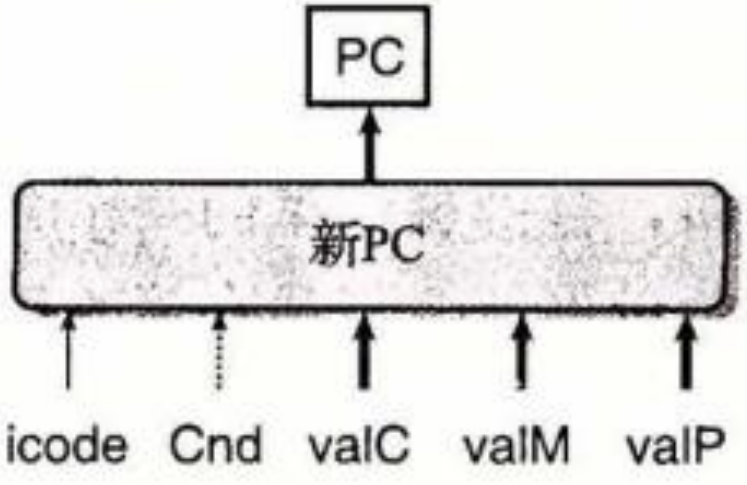
```
word dstM = [
        icode in { IMRMOVQ, IPOPQ } : rA;
        1 : RNONE;  # Don't write any register
];
```



写回	E port, dstE M port, dstM	R[rB]← valE	R[rA]← valM

更新PC

```
word new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
        icode == IJXX && Cnd : valC;
        # Completion of RET instruction.  Use value from stack

                icode == IRET : valM;
                # Default: Use incremented PC
                1 : valP;
        ];
```



PC的值大多数情况下是valP
执行call指令时为valC
执行条件跳转指令且符合条件时为valM

练习

- T1：valA/valB是在什么阶段产生的？valE呢？
- T2：哪几个值可能被写入PC？
- T3：请填充下一页给出的表格中的空缺

		`call` Dest
Fetch	icode,ifun	icode:ifun ← M_1[PC]
	rA,rB	
	valC	valC ← M_8[PC+1]
	valP	valP ← PC+9
Decode	valA, srcA	
	valB, srcB	valB ← R[`%rsp`]
Execute		
Memory		
Write back		
PC update		

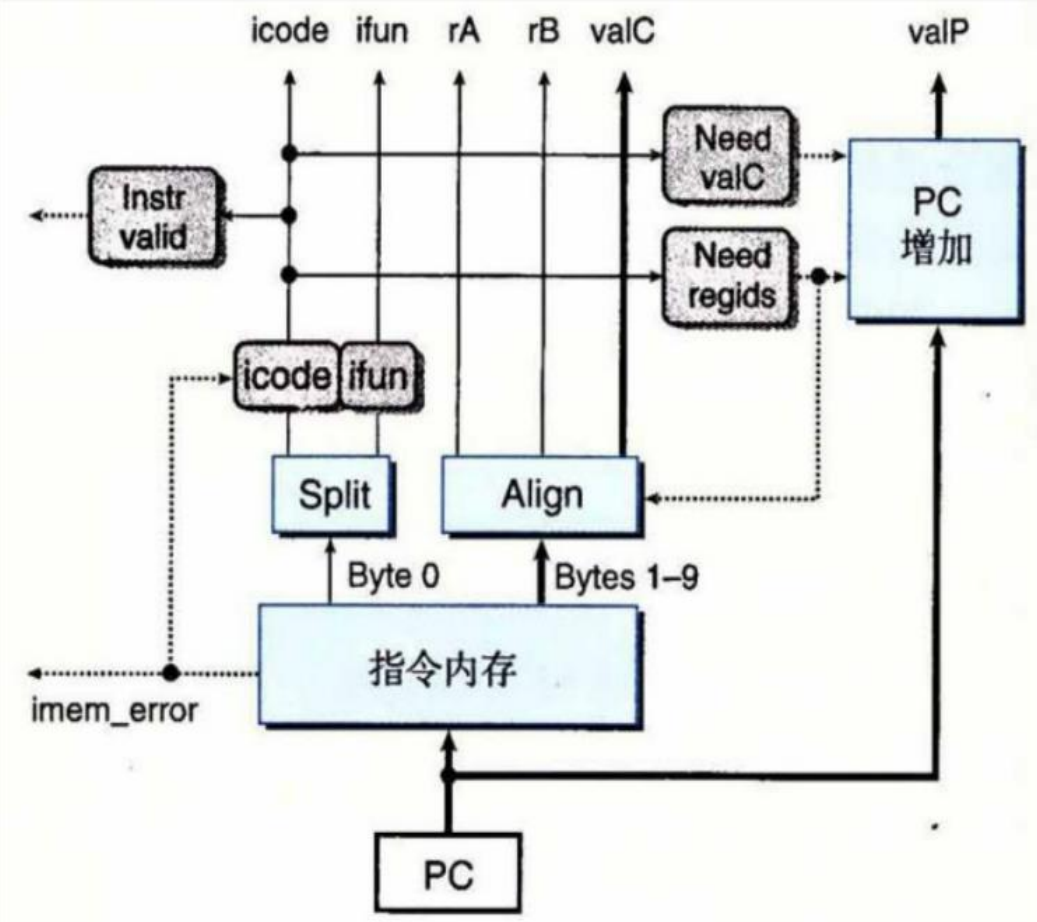
执行下面一条指令：0x100:6001
(已知%rax中是1,%rcx中是2，条件码原来全为0，stat=1)
6001：addq　%rax,%rcx

取指：

icode:ifun ←M1[PC]
rA:rB←M1[PC+1]
valP←PC+2

icode:ifun <-- M1[0x100]
6:0
rA:rB <-- M1[0x100+1]
%rax:%rcx
valP <-- 0x100+2

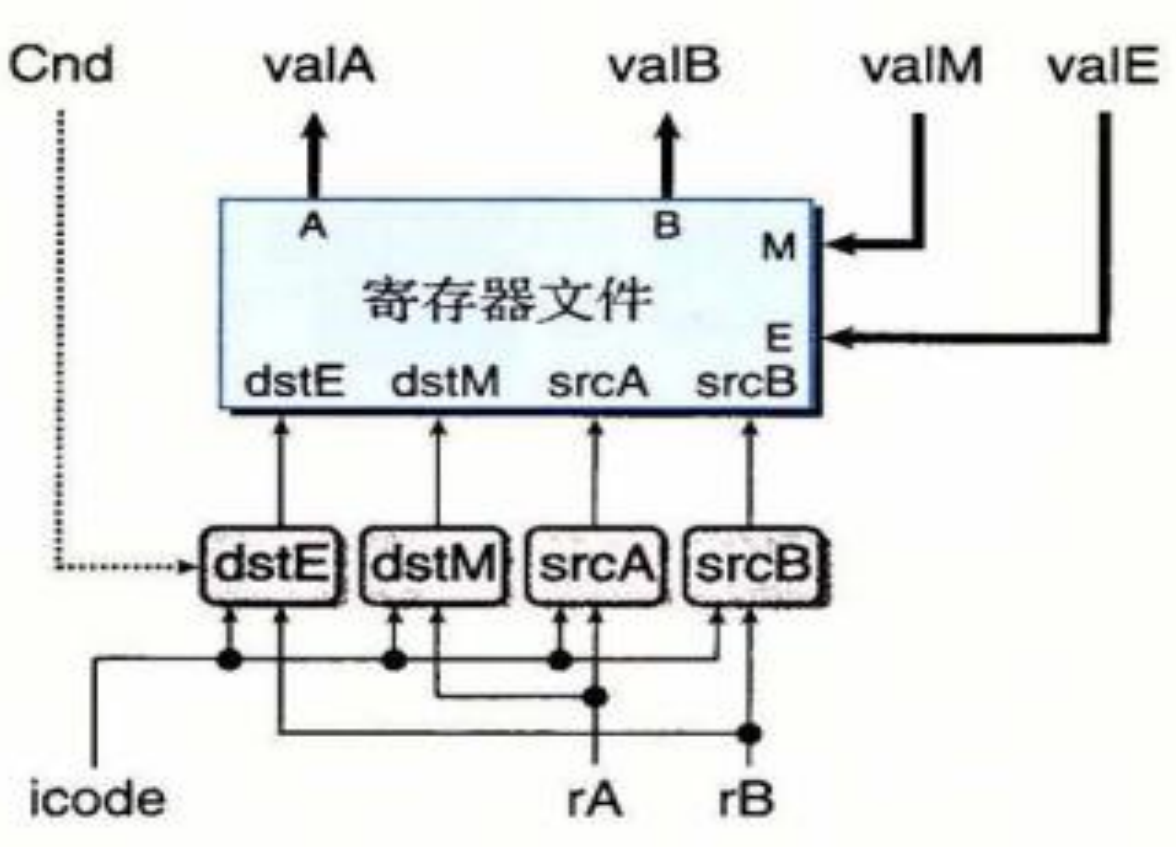
icode	6	%rax	0x1	
ifun	0	%rcx	0x2	
rA	%rax	ZF	0	
rB	%rcx	SF	0	
valP	0x102	OF	0	
valC		PC	0x100	
valA		Stat	1	
valB		（一些程序员可见状态）		



译码

- valA<--R[rA]　　valA <-- %rax 1
- valB<--R[rB]　　valB <-- %rcx 2

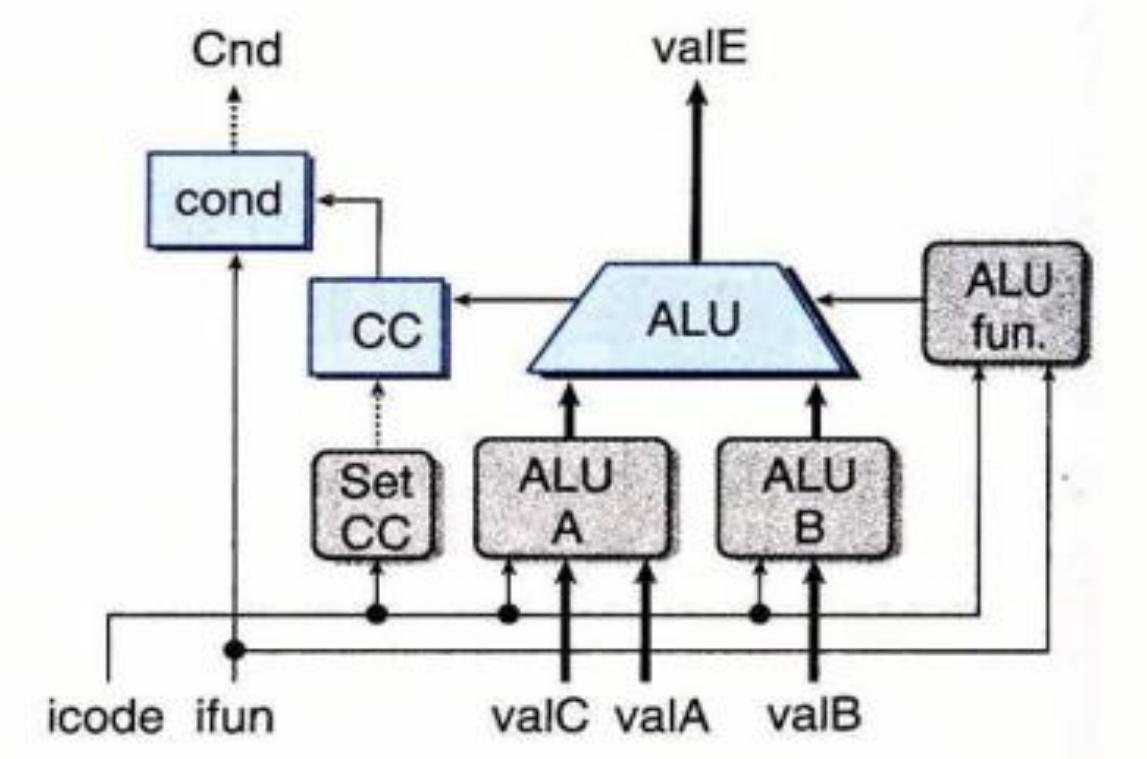
icode		6	%rax	0x1	
ifun		0	%rcx	0x2	
rA	%rax		ZF		0
rB	%rcx		SF		0
valP	0x102		OF		0
valC			PC	0x100	
valA		1	Stat		1
valB		2	（一些程序员可见状态）		
srcA	%rax				
srcB	%rcx				
dstE	%rcx				



执行

- valE<--valB OP valA valE <-- 2+1
- Set CC Set CC
  ZF=0,SF=0,OF=0

icode	6	%rax	0x1	
ifun	0	%rcx	0x2	
rA	%rax	ZF	0	
rB	%rcx	SF	0	
valP	0x102	OF	0	
valC		PC	0x100	
valA	1	Stat	1	
valB	2	（一些程序员可见状态）		
valE	3			
srcA	%rax			
srcB	%rcx			
dstE	%rcx			

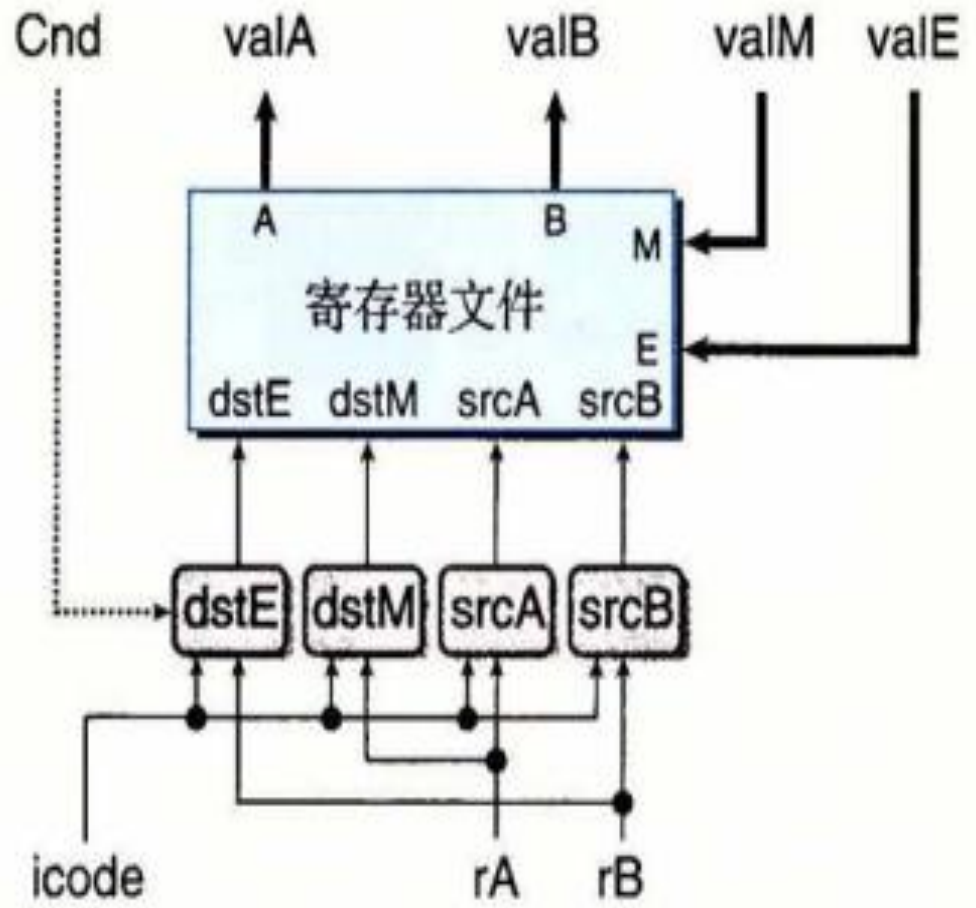


ALUfun=0
ALUA=valA
ALUB=valB
ZF=0
SF=0
OF=0

访存（该指令没有访问内存）
写回

- R[rB]<--valE
- %rcx <-- 3

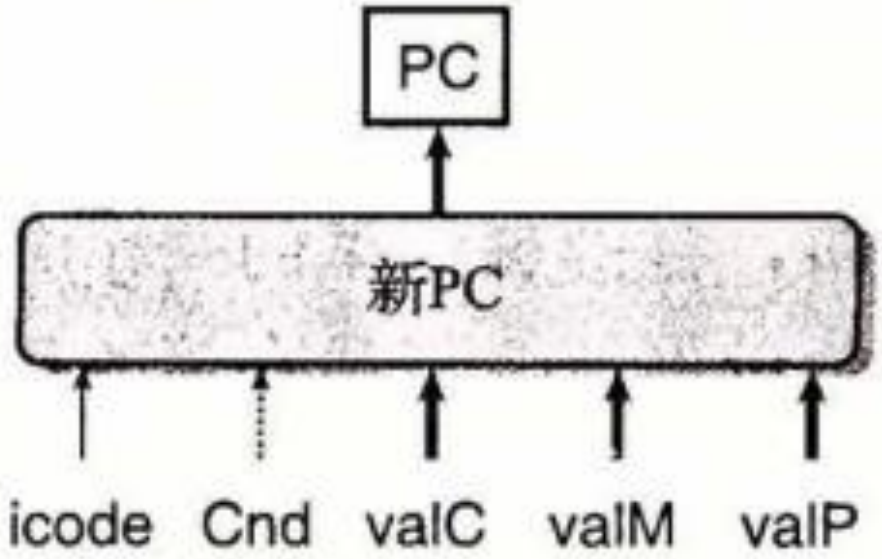
icode	6	%rax	1	
ifun	0	%rcx	3	
rA	%rax	ZF	0	
rB	%rcx	SF	0	
valP	0x102	OF	0	
valC		PC	0x100	
valA	1	Stat	1	
valB	2	（一些程序员可见状态）		
valE	3			
srcA	%rax			
srcB	%rcx			
dstE	%rcx			



更新PC

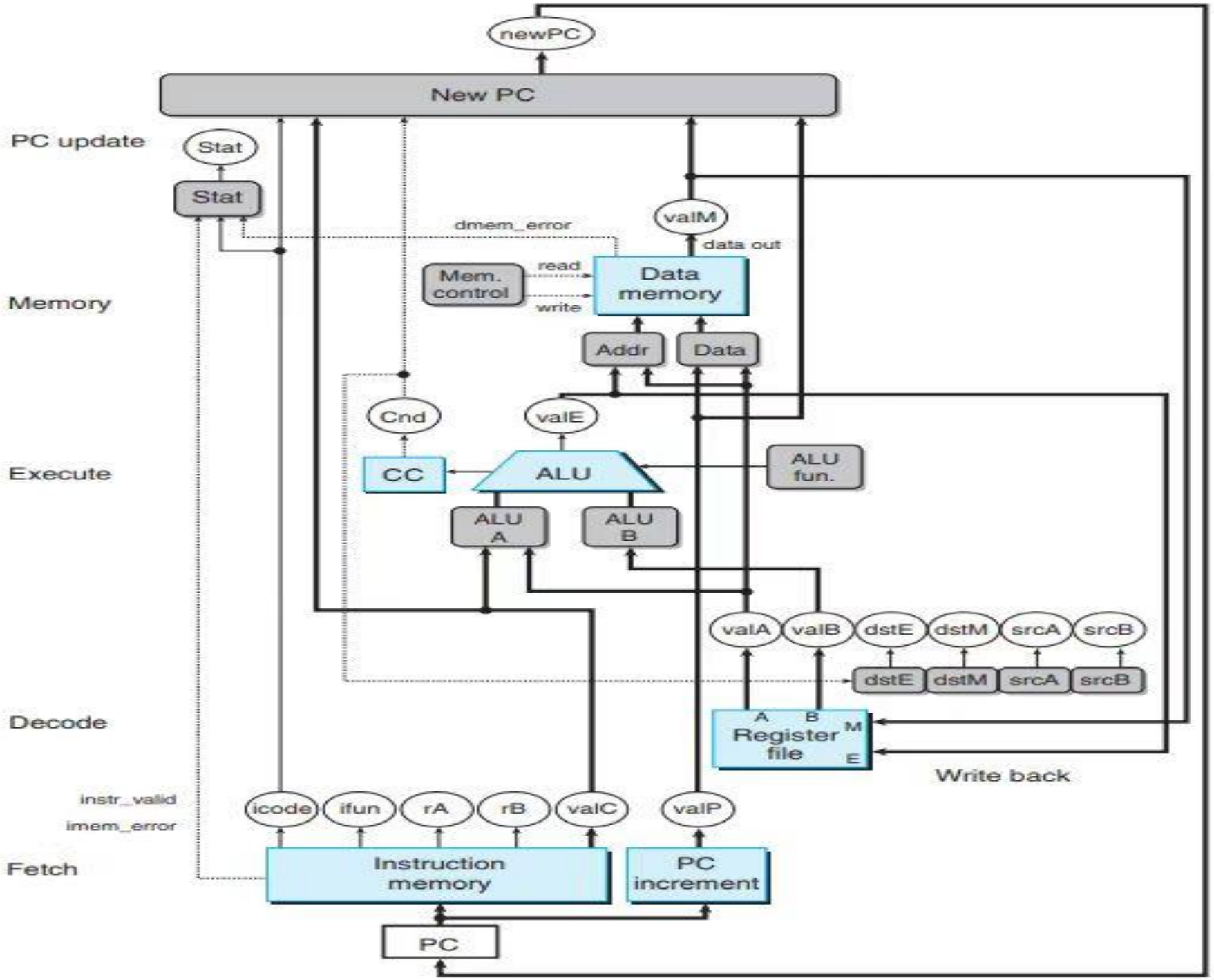
- PC<--valP
- PC <-- 0x102

icode	6	%rax	1	
ifun	0	%rcx	3	
rA	%rax	ZF	0	
rB	%rcx	SF	0	
valP	0x102	OF	0	
valC		PC	0x102	
valA	1	Stat	1	
valB	2	（一些程序员可见状态）		
valE	3			
srcA	%rax			
srcB	%rcx			
dstE	%rcx			



另一条指令:0x8:5030080000000000000000
mrmovq 0x8(%rbx),%rax
(%rbx 中存了0x800,0x808后8个字节存了4)

- icode:ifun <-- M1[0x8]
- rA:rB <-- M1[0x9] %rbx:%rax
- valC <-- M1[0xa] 0x8
- valP <-- 0x12
- valB <-- %rbx 0x800
- valE <-- 0x800+0x8
- valM <-- M8[0x808] 4
- %rax <-- 4
- PC <-- 0x12



- 数据通路（Data Path）：指令的执行部件（组合逻辑与存储器）
- 可以经过存储，PC,寄存器，ALU，CC，数据内存（硬件单元）以及灰色的逻辑部分
- 控制（Control）：控制数据通路的部分
- 所有灰色的部分