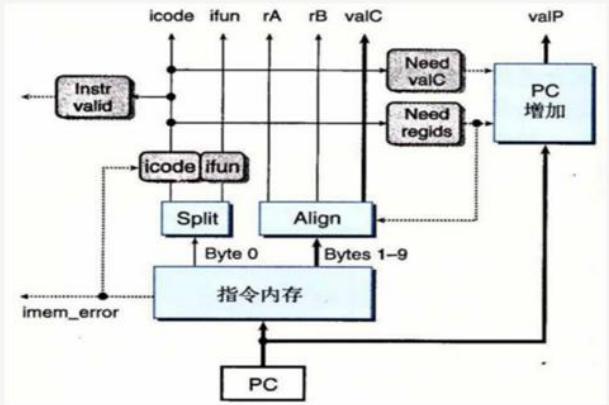
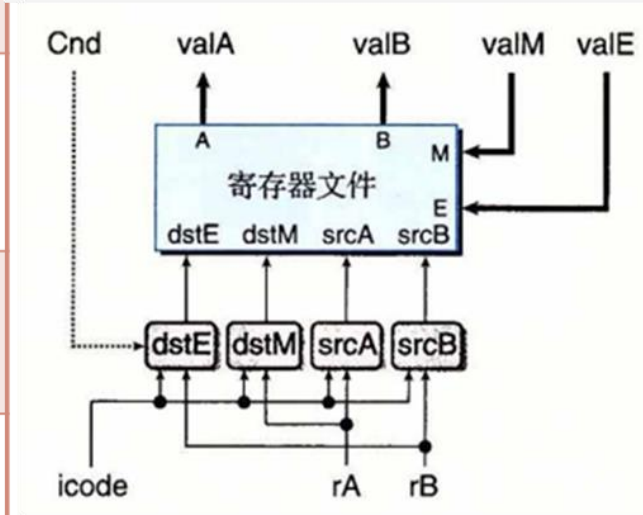
Processor Arch:Pipelined

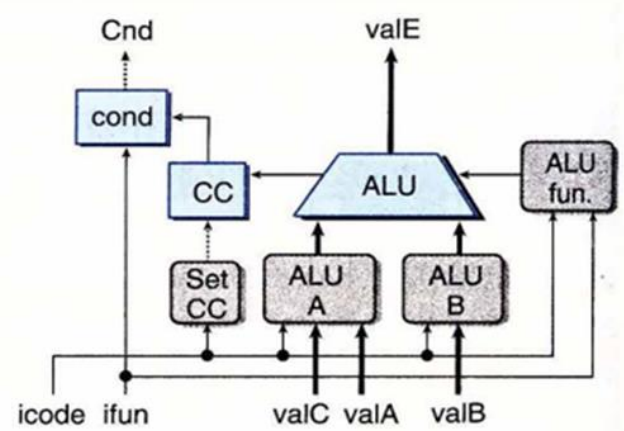
eecs_havefun

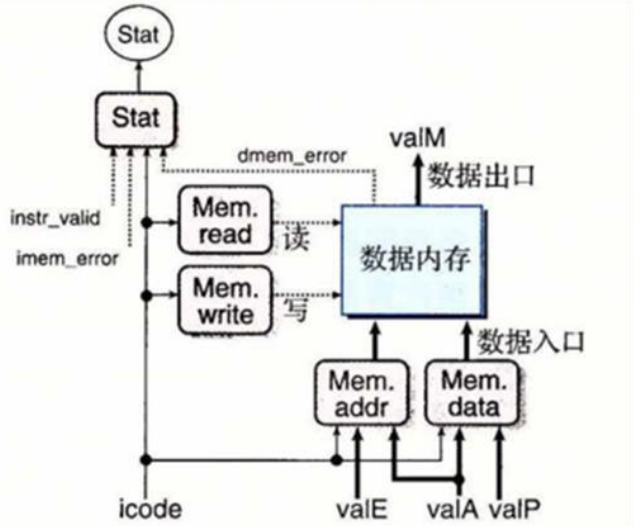
目录

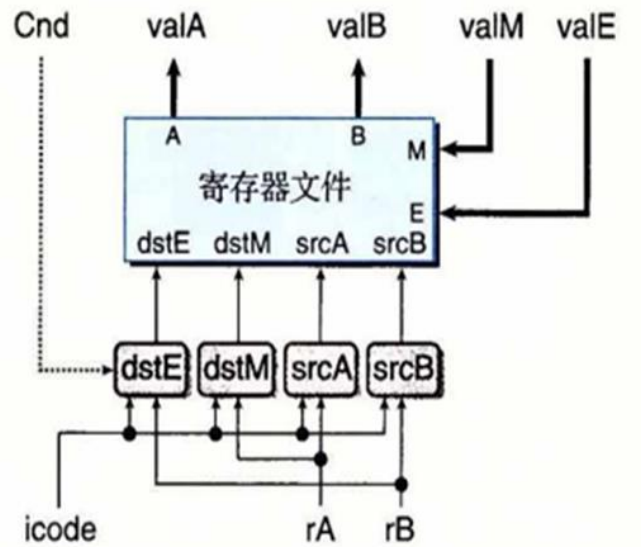
	ret
取指	icode:ifun ← M_1[PC]
译码	valA ←R[%rsp] valB←R[%rsp]
执行	valE ← valB + 8
访存	valM← M_8 [valA]
写回	R[%rsp] ← valE
更新pc	PC ← valM

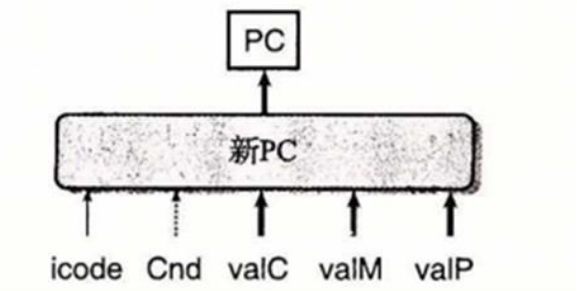




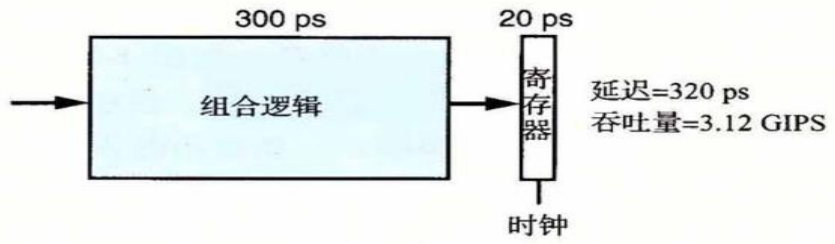




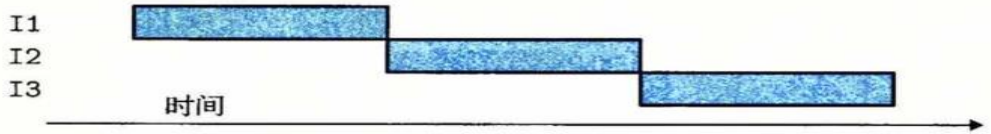




流水线基本原理

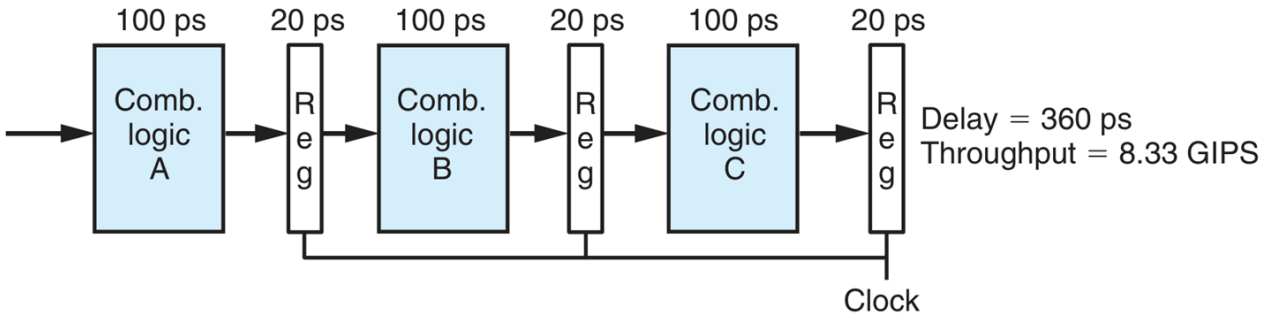


a）硬件：未流水线化的

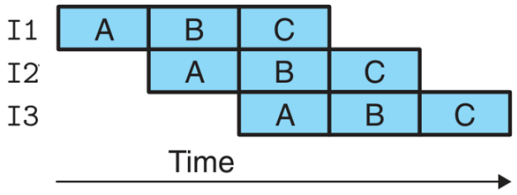


b）流水线图

图 4-32 非流水线化的计算硬件。每个 320ps 的周期内，系统用 300ps 计算组合逻辑函数，20ps 将结果存到输出寄存器中



(a) Hardware: Three-stage pipeline



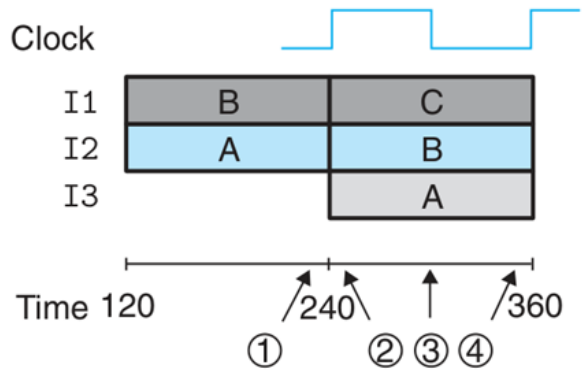
(b) Pipeline diagram

不同的指令可以同时在流水线的不同阶段计算

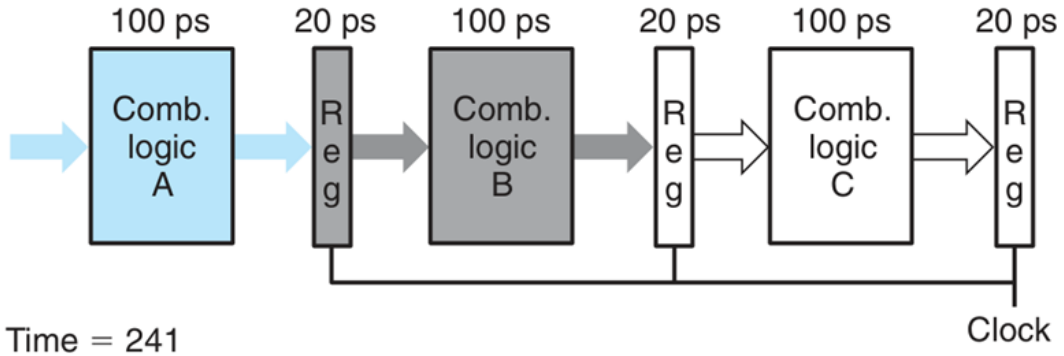
延迟（Latency）：从头到尾执行一条指令所需的时间

吞吐量：每秒能执行指令的数量

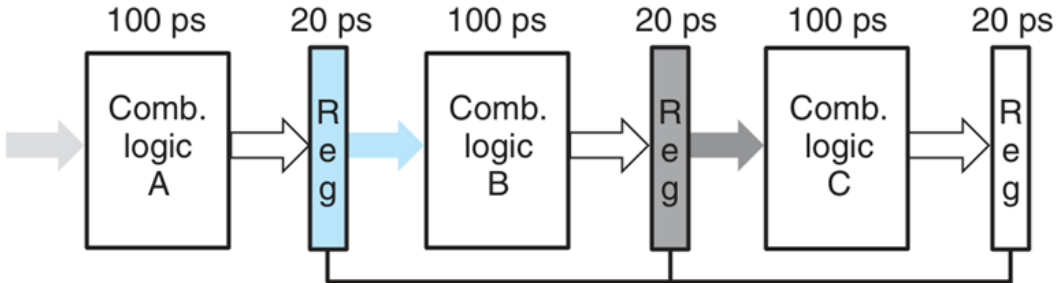
理想化的流水线系统



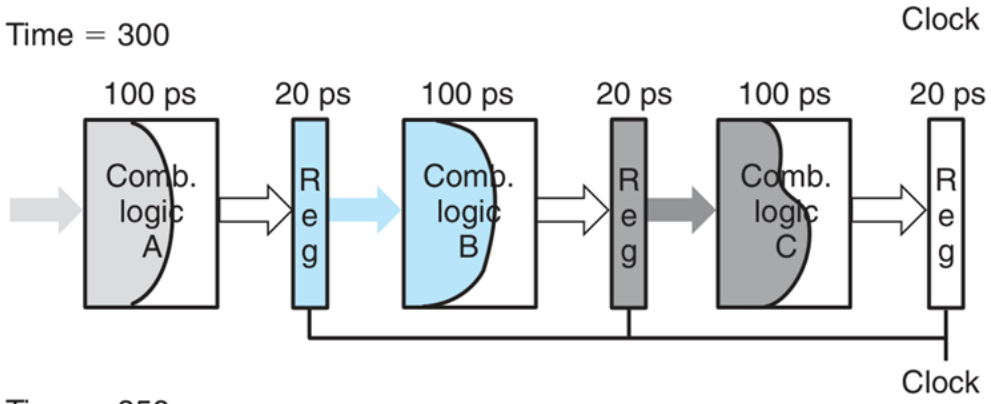
① Time = 239



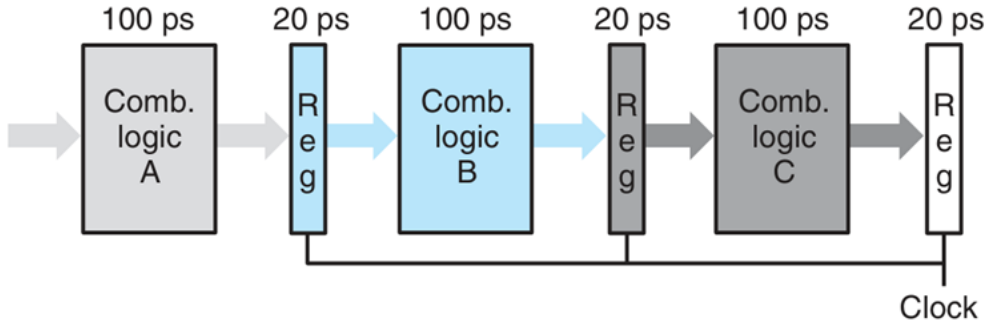
② Time = 241



③ Time = 300



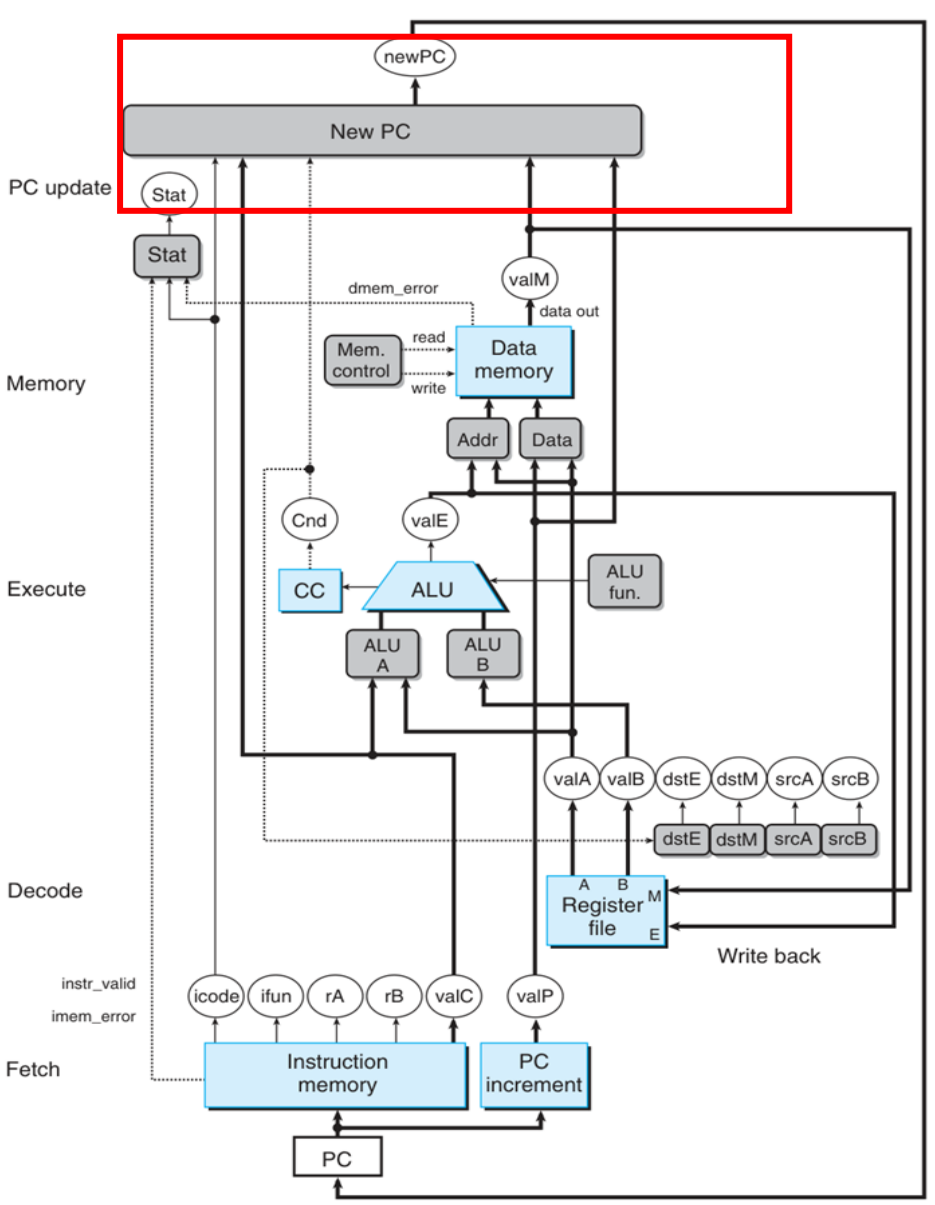
④ Time = 359

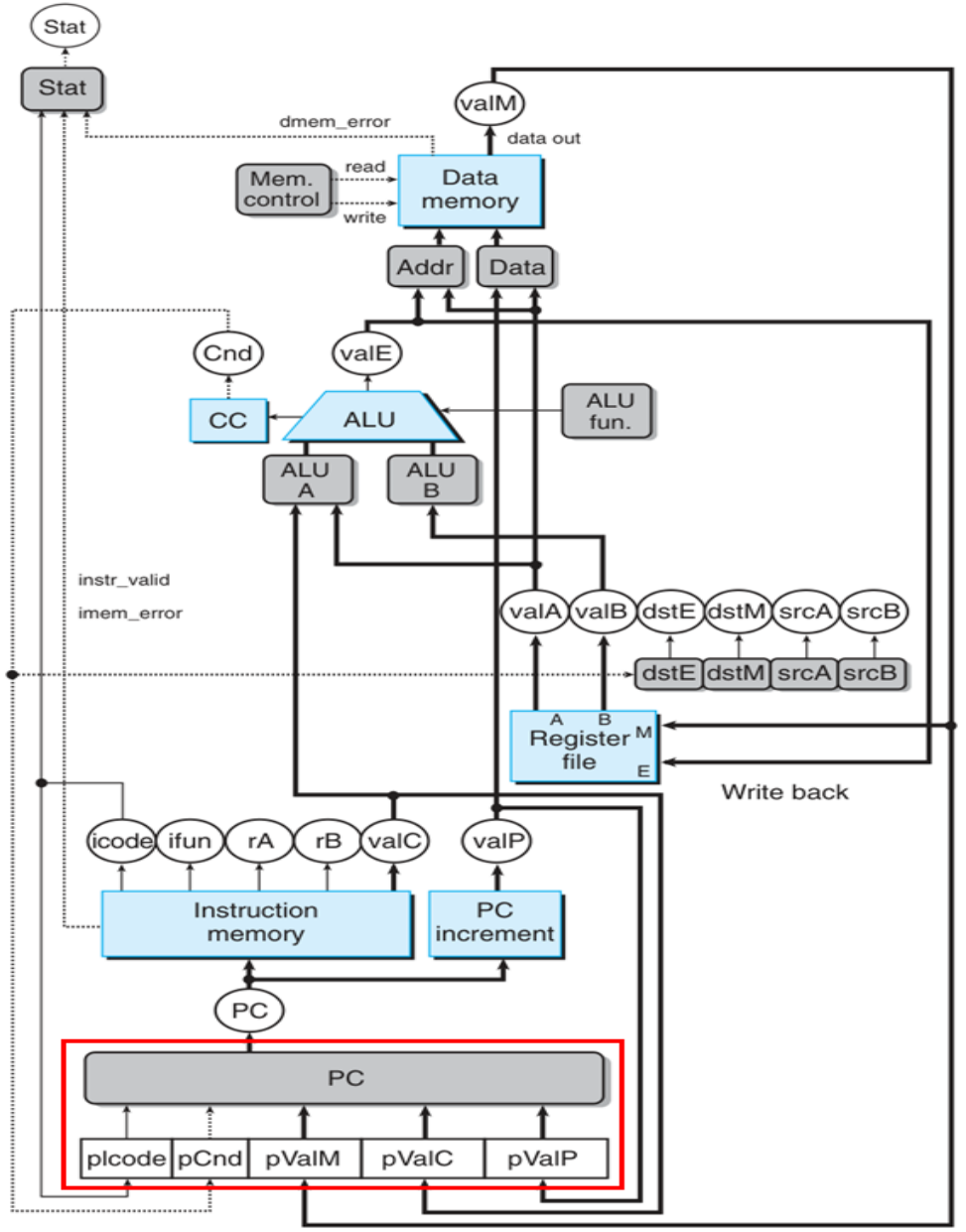


流水线的局限性

- 1.每个阶段延迟不同，系统的吞吐量受最慢的阶段限制。
- 2.流水线阶段划分越多，边际收益越低（流水线寄存器本身的延迟）

SEQ+与SEQ的区别





1.PC计算时机的改变以及各阶段顺序的改变

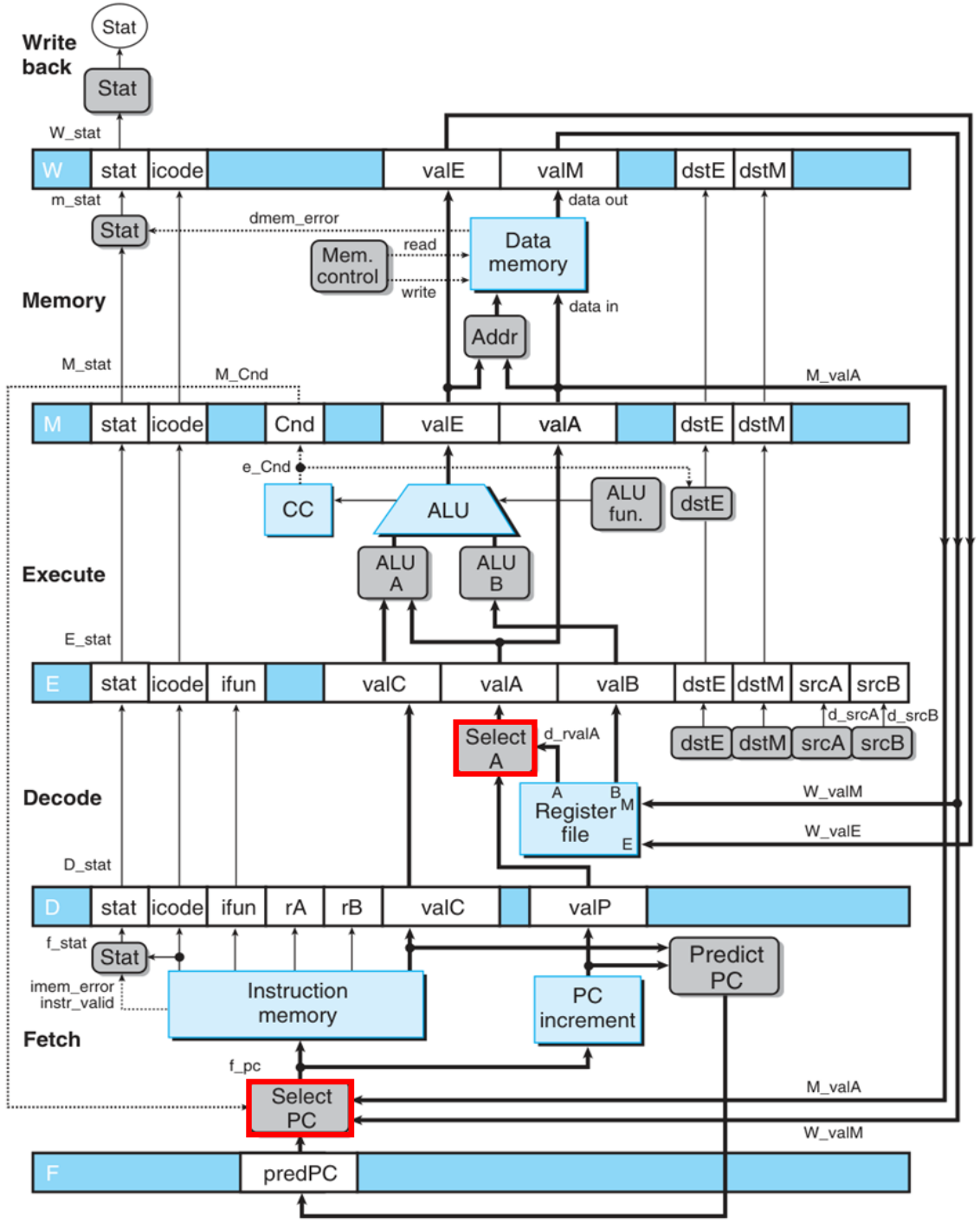
2.不再有硬件寄存器存储PC

PIPE-

插入了5个流水线寄存器，对信号进行了重新排列和标号（大写前缀FDEMW指的是流水线寄存器，小写前缀fdemw指的是流水线阶段），许多信号需要一直携带

增加了SelectA控制块，其中只有调用了call或者jxx指令时valP会写入valA,减少需要携带的信号的数量

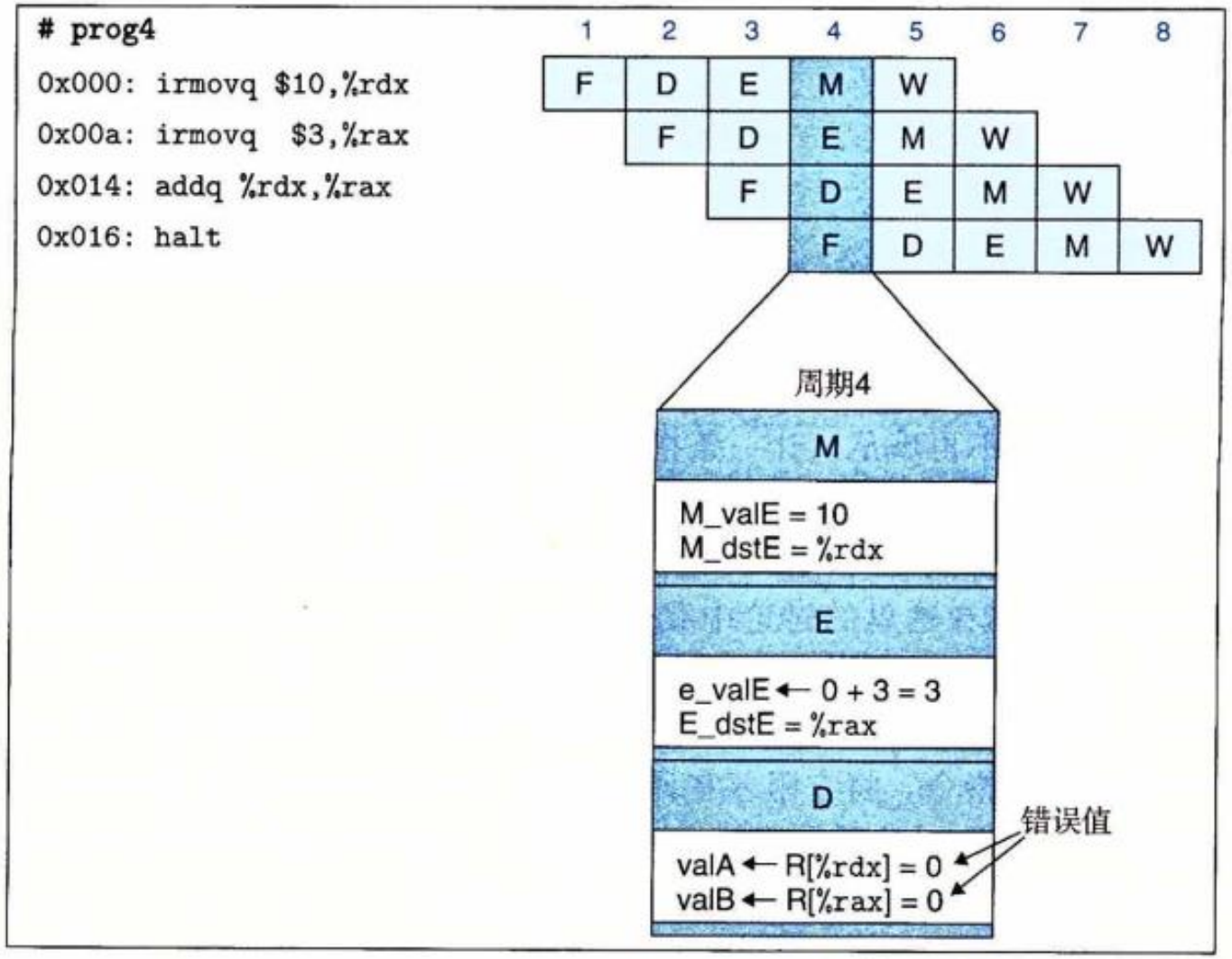
在流水线中，有时我们在一个时钟周期内还没有计算出下一条指令的地址，我们需要预测下一个PC



流水线冒险

- 数据相关：下一条指令要用到这条指令计算出的结果
- Etc：irmovq $1,%rax; addq %rdx,%rax
- 控制相关：一条指令要更改PC
- 这些相关可能会导致流水线产生计算错误，称为冒险（Hazard）
- 译码阶段会发现是否需要处理冒险
- 数据冒险：一条指令会更新后面指令要读到的那些程序状态

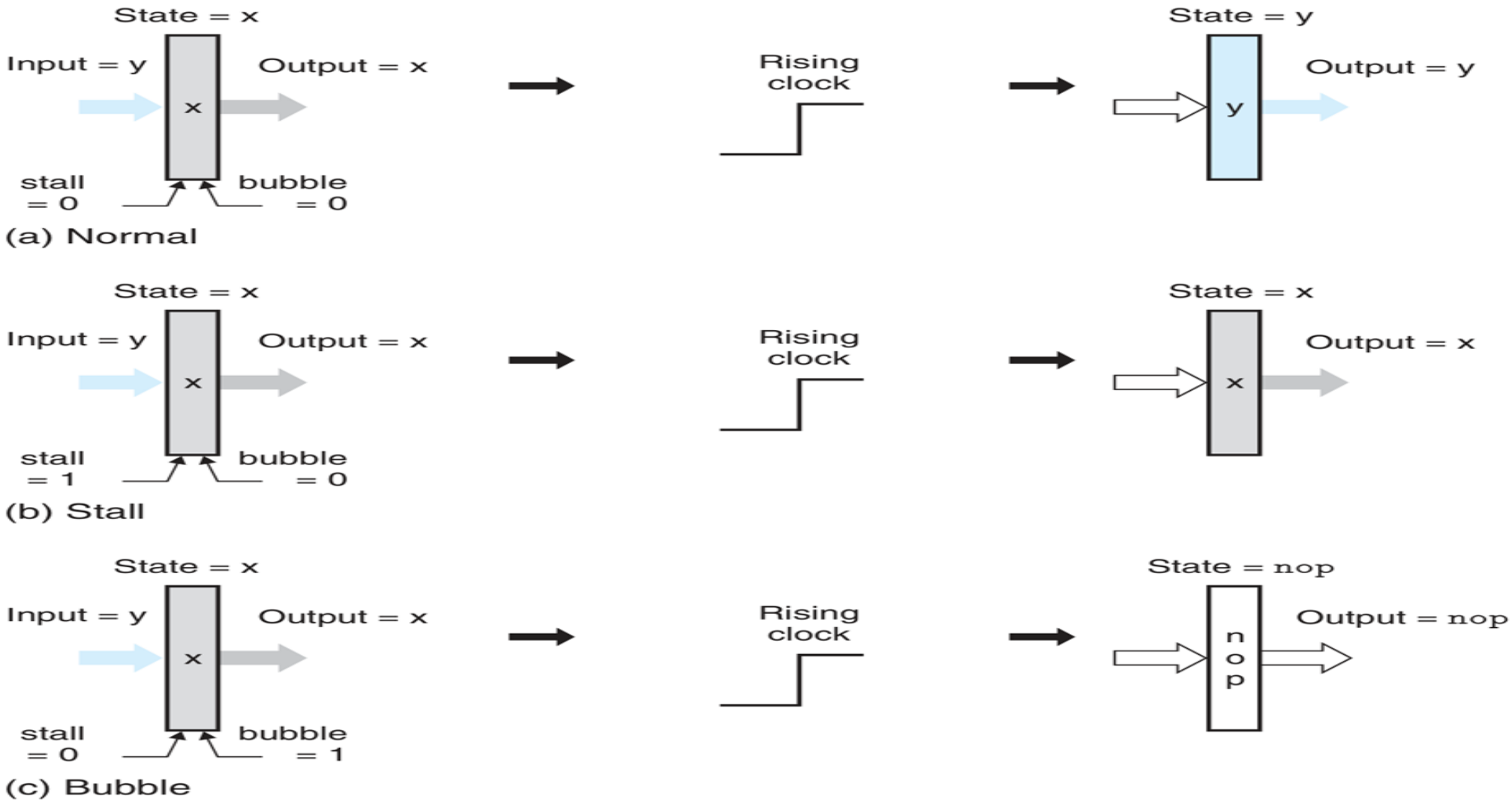
数据冒险（data hazard）

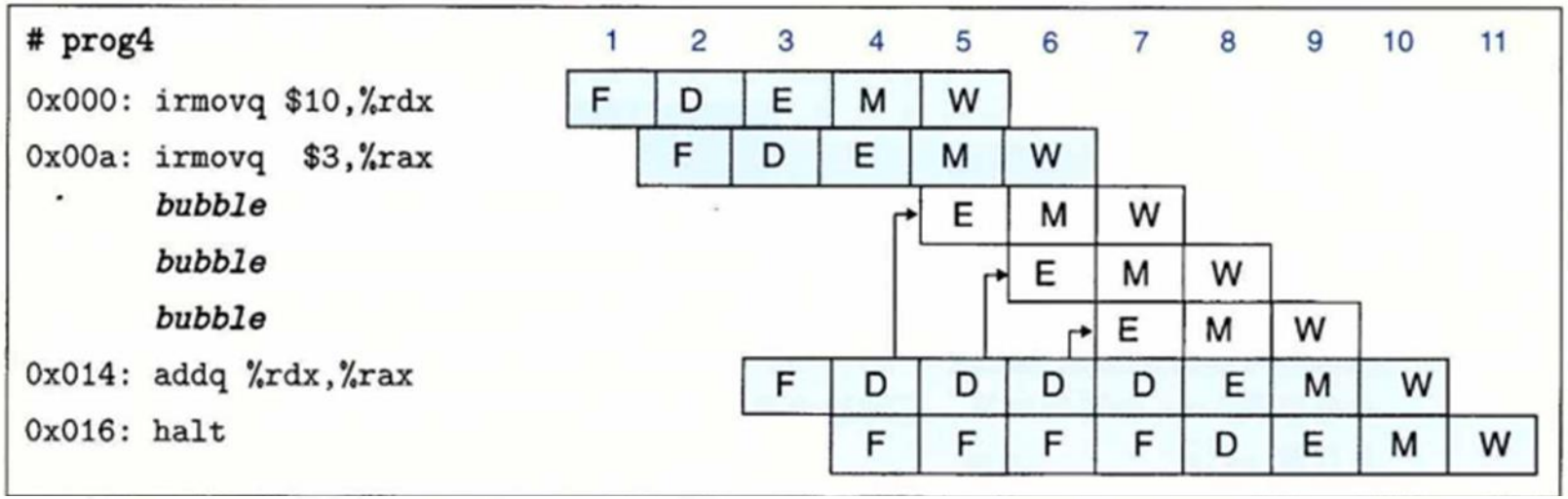


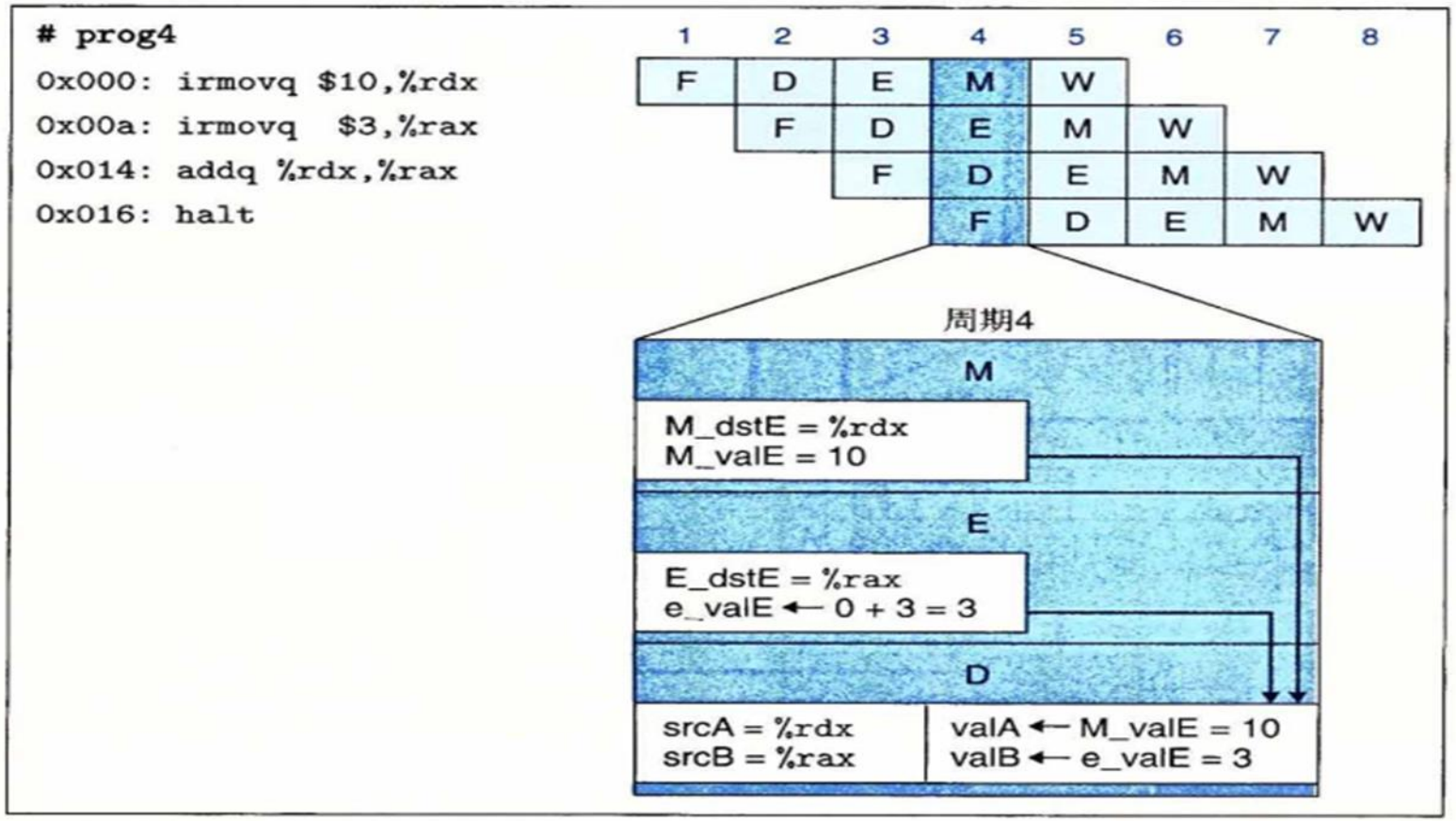
避免数据冒险

- 方式:
- 暂停（stall），插入气泡(bubble)，转发
- 实现转发需要在PIPE-中增加额外的数据连接和控制逻辑

Stall,bubble信号对流水线寄存器的影响





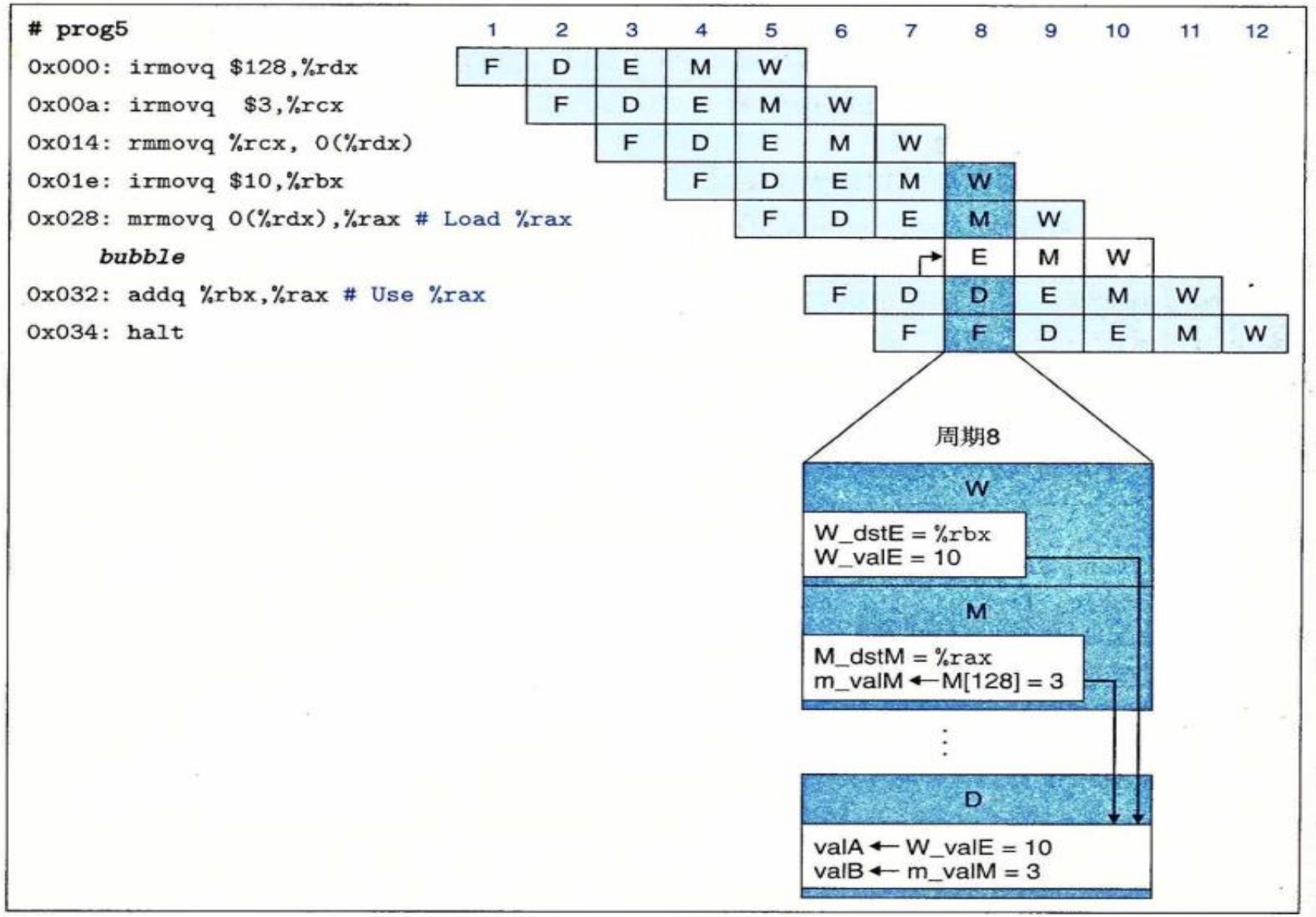


某个位置之前的指令暂停，某个位置处于气泡状态，某个位置之后正常，相当于插入了一条nop指令

加载/使用冒险(load/use hazard)

原因：上一条指令从内存中读出某个寄存器的值，下一条指令需要使用这个值（内存读是确定值的最晚阶段，访存），这样和下一条指令差了两个阶段

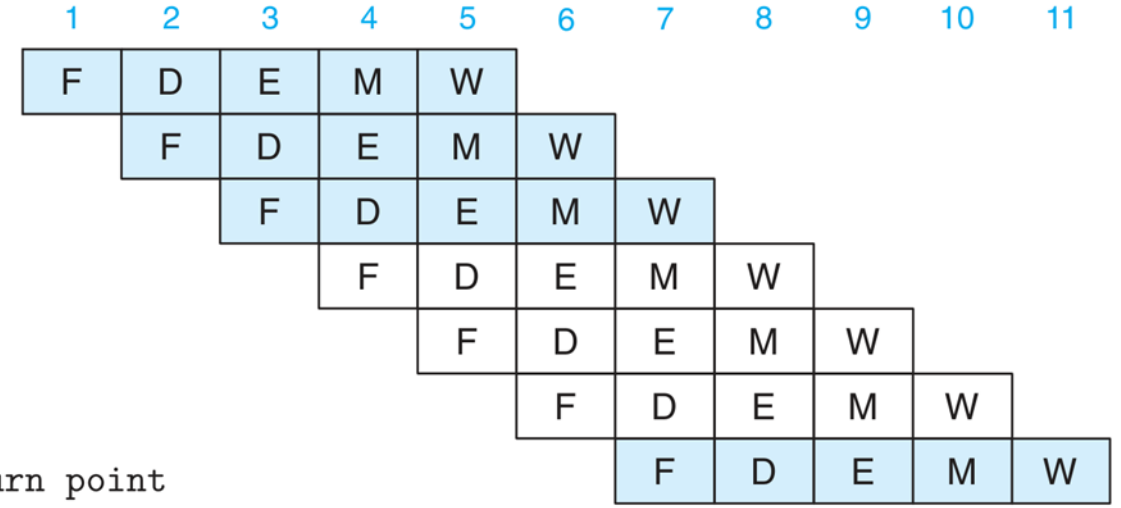
解决方法：用暂停和转发结合处理。用暂停处理加载/使用冒险的方法称为加载互锁（load +interlock）



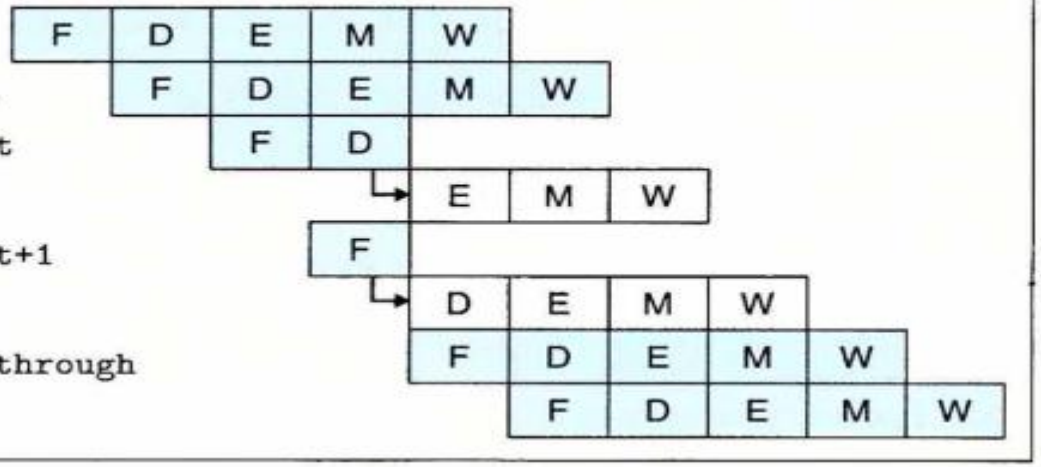
控制冒险（control hazard）

- 处理器无法根据处于取指阶段的当前指令来确定下一条指令的地址
- 例子，return，条件跳转预测分支错误
- 某几个位置设置为气泡，其余位置正常，相当于从某个时间开始删除了指令

```
# prog7
0x000: irmovq Stack,%edx
0x00a: call proc
0x020: ret
       bubble
       bubble
       bubble
0x013: irmovq $10,%edx # Return point
```




```
# prog7
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovl $2,%rdx # Target
       bubble
0x020: irmovl $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

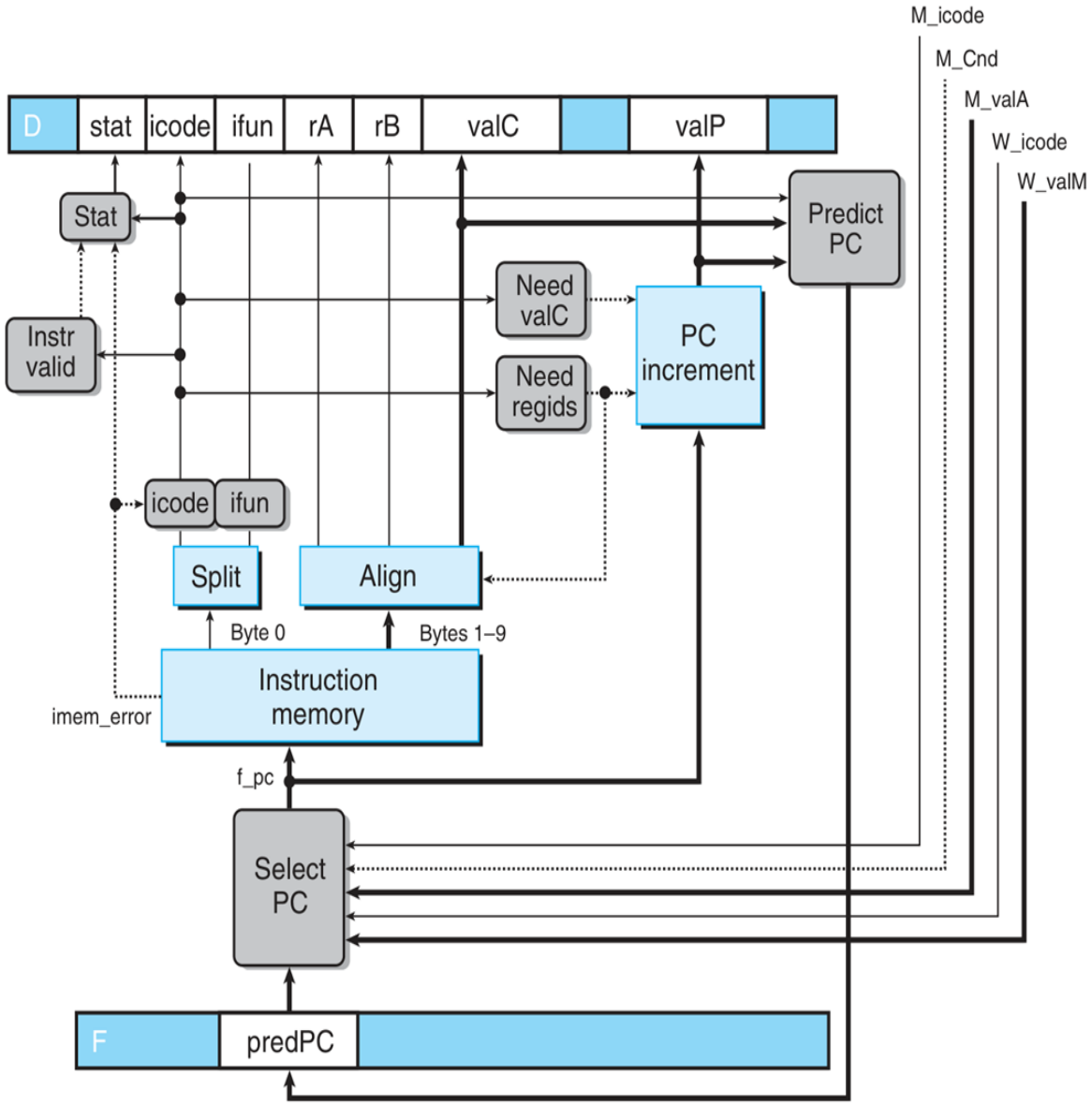


PIPE各阶段的实现

- 1.PC选择和取指阶段

```
word f_pc = [
        # 分支预测错误，这里的valA存着原来的valP
        M_icode == IJXX && !M_Cnd : M_valA;
        # ret执行完毕
        W_icode == IRET : W_valM;
        #默认用预测值
        1 : F_predPC;
];
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

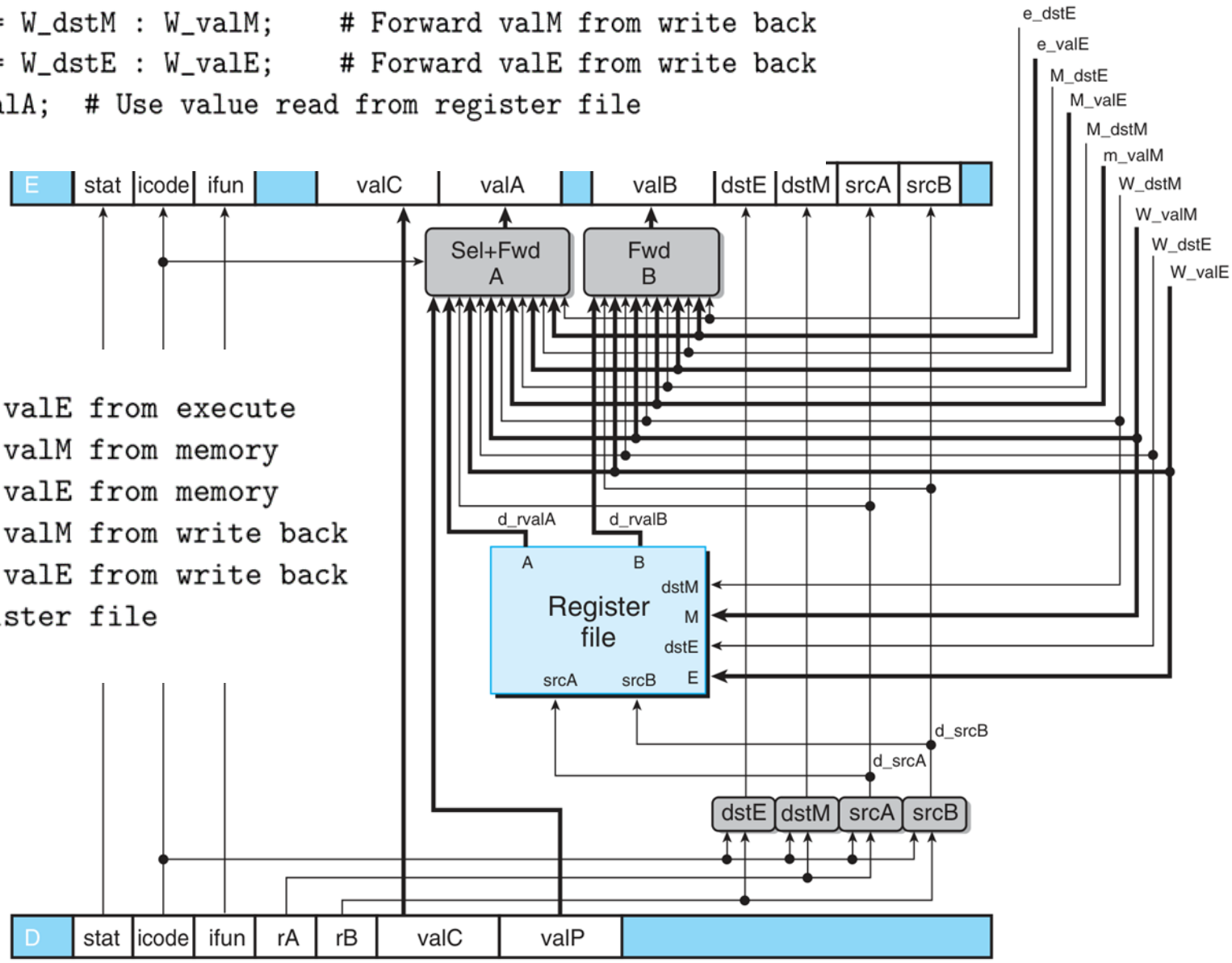
	JXX Dest
取指	icode:ifun ← M_1[PC] valC← M_8 [PC+1] valP←PC+9
译码	
执行	Cnd ← Cond(CC, ifun)
访存	
写回	
更新pc	PC ← Cnd? valC: valP



- 2.译码和写回阶段

```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
```

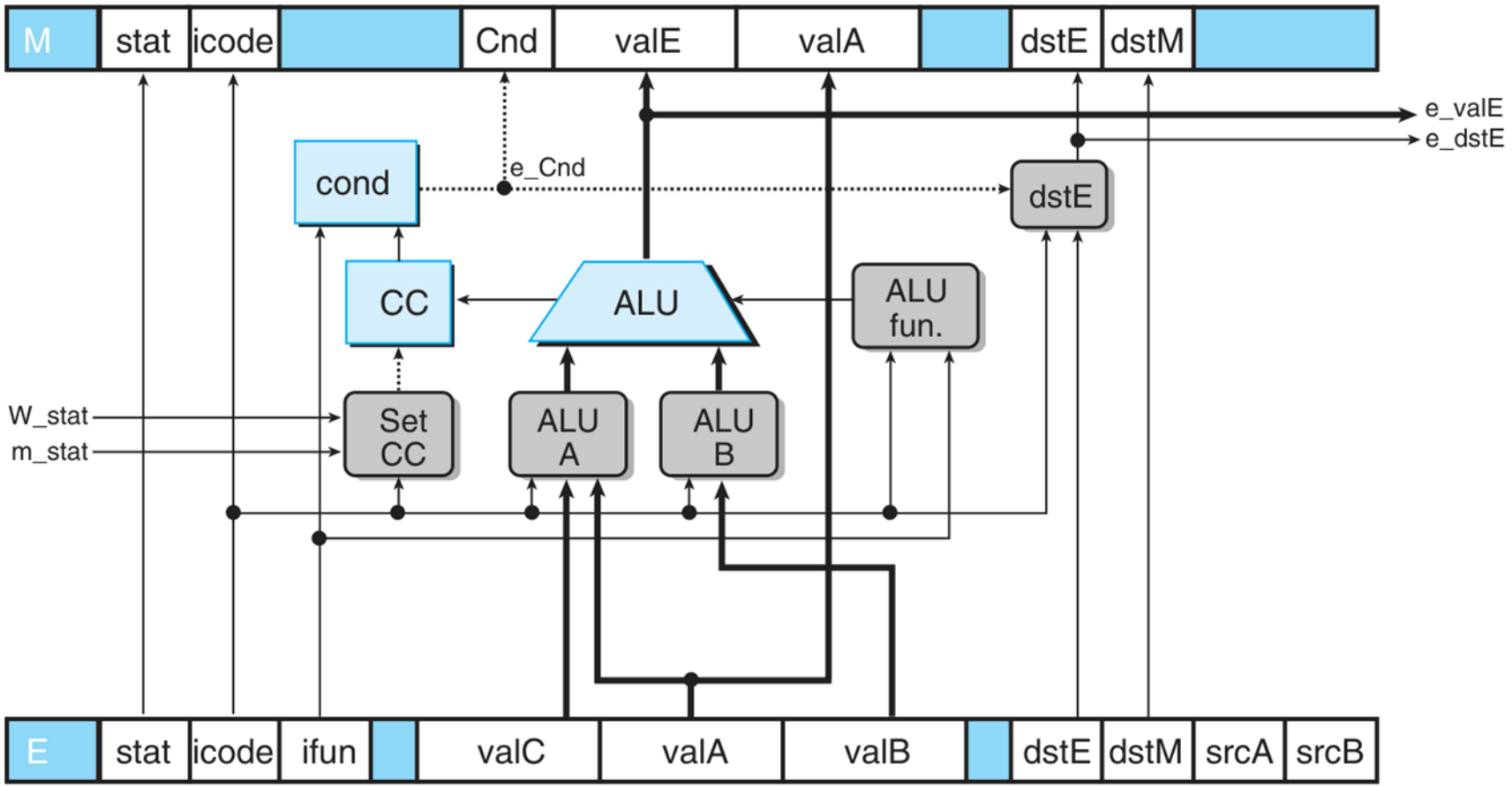
```
word d_valB = [
        d_srcB == e_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];
```



Sel+FwdA(Select+Forward)合并了valP，valA两个信号，并实现了valA的转发逻辑

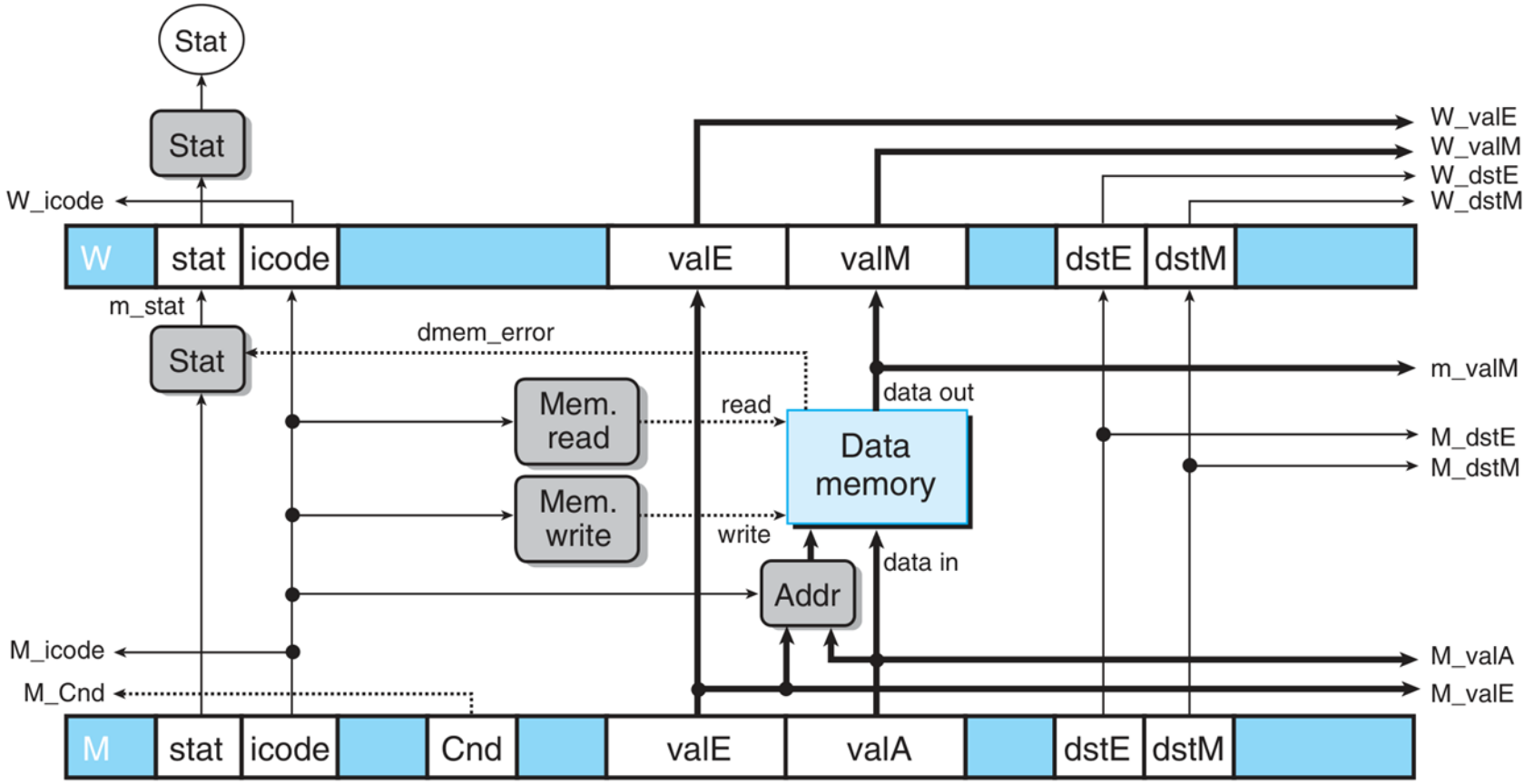
- 3.执行阶段

区别：Set CC 被信号m_stat和W_stat控制原因和异常控制的机制有关



- 4.访存阶段

区别：删去了Data块；许多信号作为转发源被传回



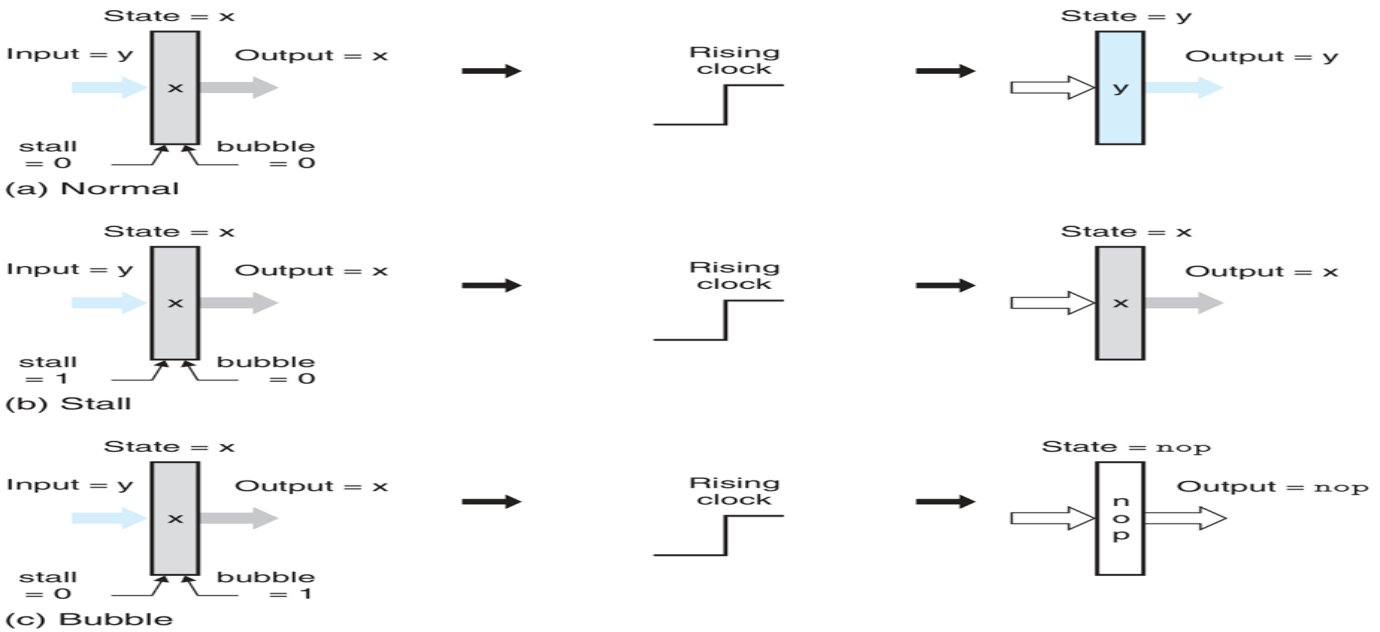
流水线控制逻辑

- 必须处理其他机制无法处理的下列4种情况：
- 加载/使用冒险，处理ret，处理预测错误的分支，异常
- 异常发生后的处理：

在我们的阶段设计中，每个流水线寄存器中会包含一个状态码 stat，随着每条指令经过流水线阶段，它会记录指令的状态。当异常发生时，我们将这个信息作为指令状态的一部分记录下来，并且继续取指、译码和执行指令，就好像什么都没有出错似的。当异常指令到达访存阶段时，我们会采取措施防止后面的指令修改程序员可见的状态：1）禁止执行阶段中的指令设置条件码，2）向内存阶段中插入气泡，以禁止向数据内存中写入，3）当写回阶段中有异常指令时，暂停写回阶段，因而暂停了流水线。

- 异常指令之前的指令都完成了，而后面的指令对程序员可见的状态都没有影响

Condition	Trigger
Processing `ret`	IRET ∈ {D_icode, E_icode, M_icode}
Load/use hazard	E_icode ∈ {IMRMOVQ, IPOPQ} && E_dstM ∈ {d_srcA, d_srcB}
Mispredicted branch	E_icode = IJXX && !e_Cnd
Exception	m_stat ∈ {SADR, SINS, SHLT} || W_stat ∈ {SADR, SINS, SHLT}



条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
加载/使用冒险	暂停	暂停	气泡	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常

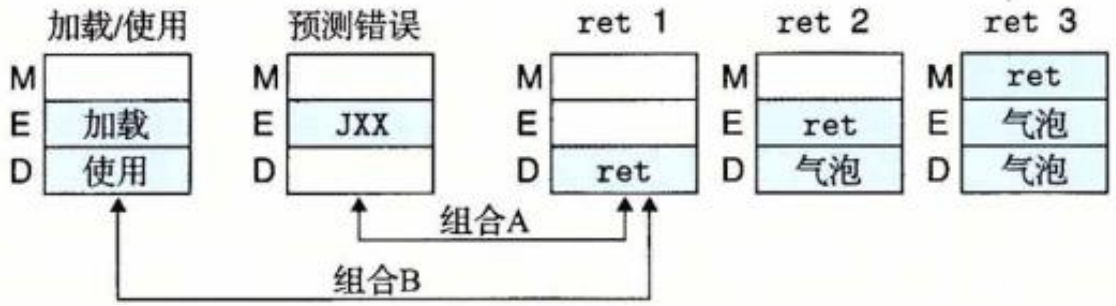
正在处理ret指令（ret处于D,E,M阶段开始时）

加载/使用冒险：即将进行执行阶段的指令是mrmovq或者popq并且该指令从内存中读出的数的目的地与译码阶段翻译得到的某个source一致

分支预测错误：进行执行阶段的指令是jxx并且执行阶段得到的条件码不符合条件

异常：访存阶段状态码异常或者写入阶段的流水线寄存器状态码异常

控制条件组合（Control combination）



条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常
组合					

条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
加载/使用冒险	暂停	暂停	气泡	正常	正常
组合	暂停	气泡＋暂停	气泡	正常	正常
期望的情况					

控制逻辑实现

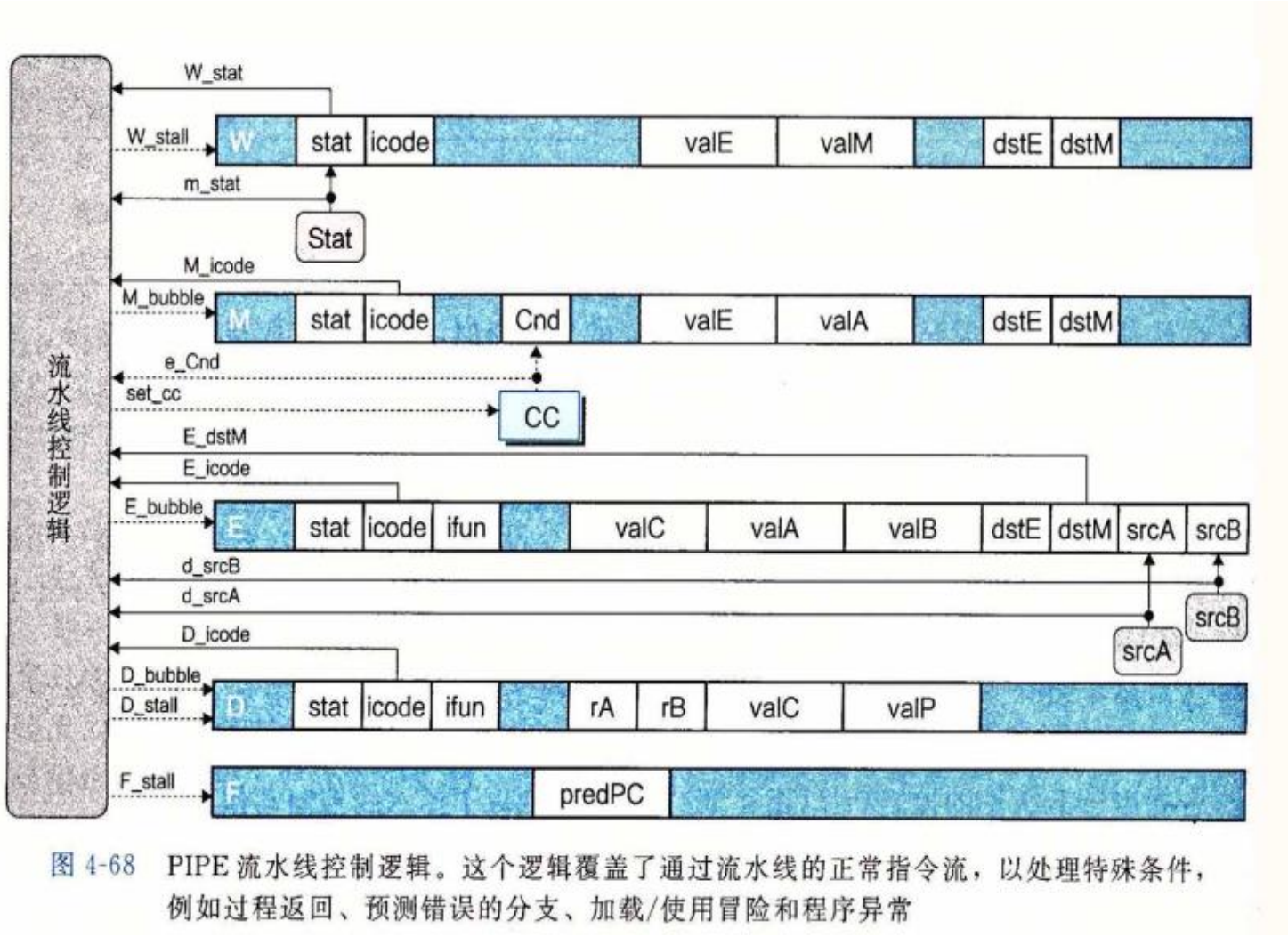


图 4-68 PIPE 流水线控制逻辑。这个逻辑覆盖了通过流水线的正常指令流，以处理特殊条件，例如过程返回、预测错误的分支、加载/使用冒险和程序异常

- bool F_stall =
- # Conditions for a load/use hazard
- E_icode in { IMRMOVQ, IPOPQ } &&
  E_dstM in { d_srcA, d_srcB } ||
  # Stalling at fetch while ret passes through pipeline
  IRET in { D_icode, E_icode, M_icode };

```
bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } &&
     E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
    # but not condition for a load/use hazard
    !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
      IRET in { D_icode, E_icode, M_icode };
bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } &&
     E_dstM in { d_srcA, d_srcB};
```

```
# Start injecting bubbles as soon as exception passes through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS,
SHLT };
```

```
bool W_stall = W_stat in { SADR, SINS, SHLT };
```

性能分析

原因	名称	指令频率	条件频率	气泡	乘积
加载/使用	*lp*	0.25	0.20	1	0.05
预测错误	*mp*	0.20	0.40	2	0.16
返回	*rp*	0.02	1.00	3	0.06
总处罚					0.27

三种处罚的总和是 0.27，所以得到 CPI 为 1.27。

我们的目标是设计一个每个周期发射一条指令的流水线，也就是 CPI 为 1.0。虽然没有完全达到目标，但是整体性能已经很不错了。我们还能看到，要想进一步降低 CPI，就应该集中注意力预测错误的分支。它们占到了整个处罚 0.27 中的 0.16，因为条件转移非常常见，我们的预测策略又经常出错，而每次预测错误都要取消两条指令。

旁注 其他的分支预测策略

我们的设计使用总是选择(always taken)分支的预测策略。研究表明这个策略的成功率大约为 60%[44, 122]。相反，从不选择(never taken, NT)策略的成功率大约为 40%。稍微复杂一点的是反向选择、正向不选择(backward taken, forward not-taken, BTFNT)的策略，当分支地址比下一条地址低时就预测选择分支，而分支地址比较高时，就预测不选择分支。这种策略的成功率大约为 65%。这种改进源自一个事实，即循环是由后向分支结束的，而循环通常会执行多次。前向分支用于条件操作，而这种选择的可能性较小。在家庭作业 4.55 和 4.56 中，你可以修改 Y86-64 流水线处理器来实现 NT 和 BTFNT 分支预测策略。

正如我们在 3.6.6 节中看到的，分支预测错误会极大地降低程序的性能，因此这就促使我们在可能的时候，要使用条件数据传送而不是条件控制转移。

```
loop:
        mrmovq (%rdi),%r10      # x = *start
        xorq %r11,%r11          # Constant 0
        subq %r10,%r11          # -x
        jle pos                 # Skip if -x <= 0
        rrmovq %r11,%r10        # x = -x
```

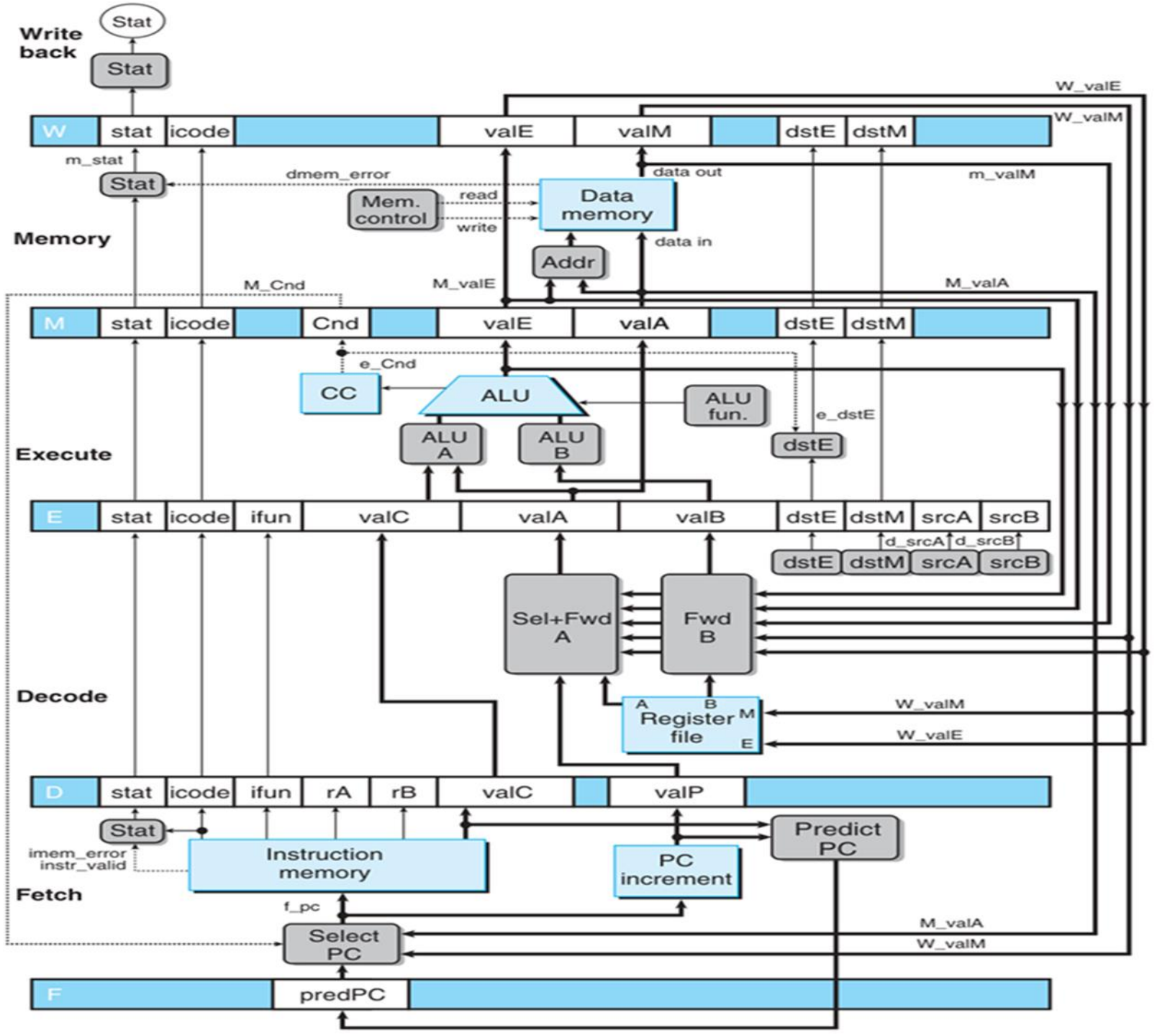
```
loop:
        mrmovq (%rdi),%r10      # x = *start
        xorq %r11,%r11          # Constant 0
        subq %r10,%r11          # -x
        cmovg %r11,%r10         # If -x > 0 then x = -x
```

练习

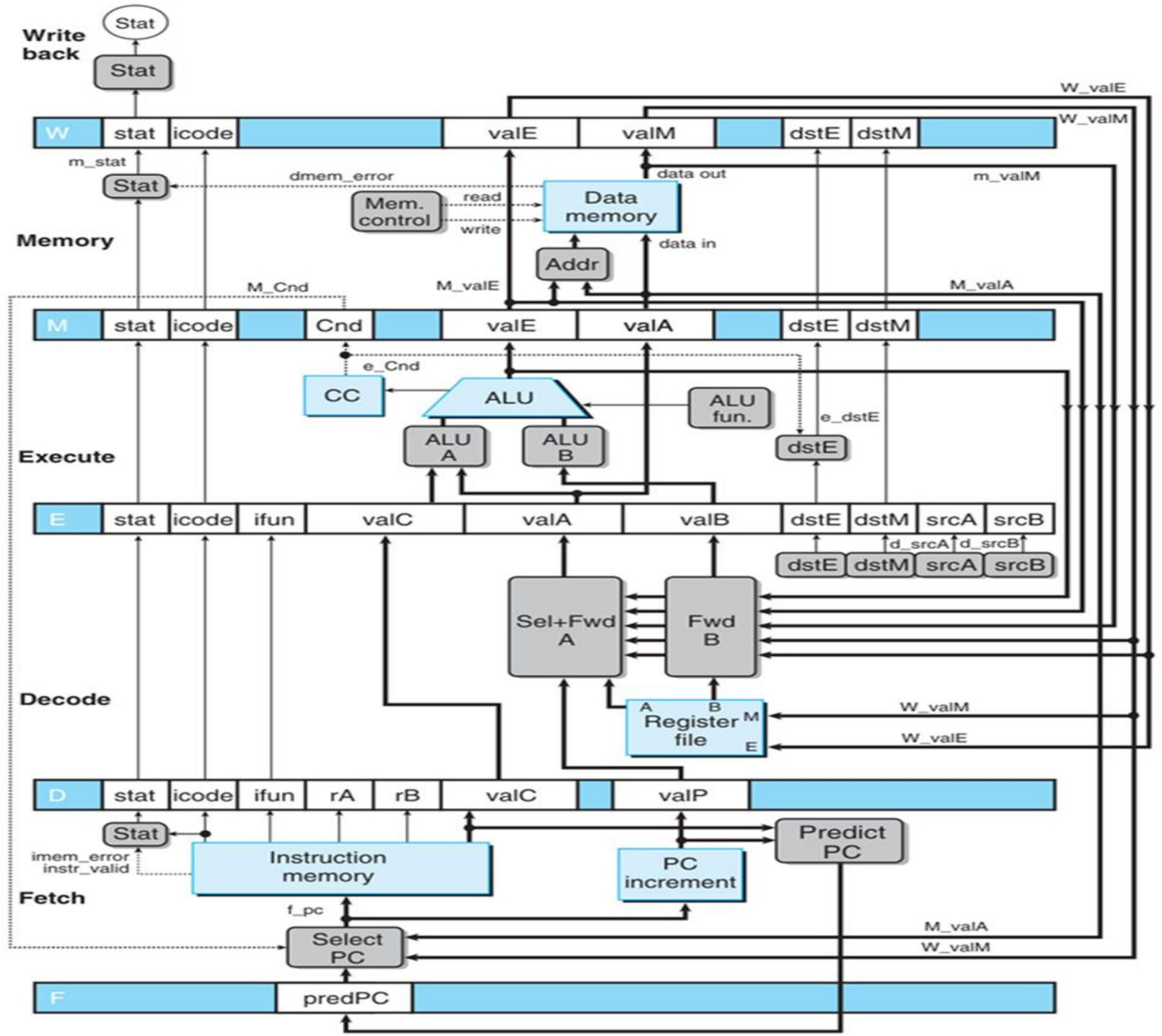
- 模拟在PIPE中执行指令:
- ret; popq rA
- 0x0:irmovq $3,%rax
- 0xa:xorq %rdx,%rdx
- 0xc:addq %rdx,%rax
- 参考：seq时每条指令的行为

Irmovq $3,%rax	F	D	E	M	W		
Xorq %rdx,%rdx		F	D	E	M	W	
Addq %rdx,%rax			F	D	E	M	W

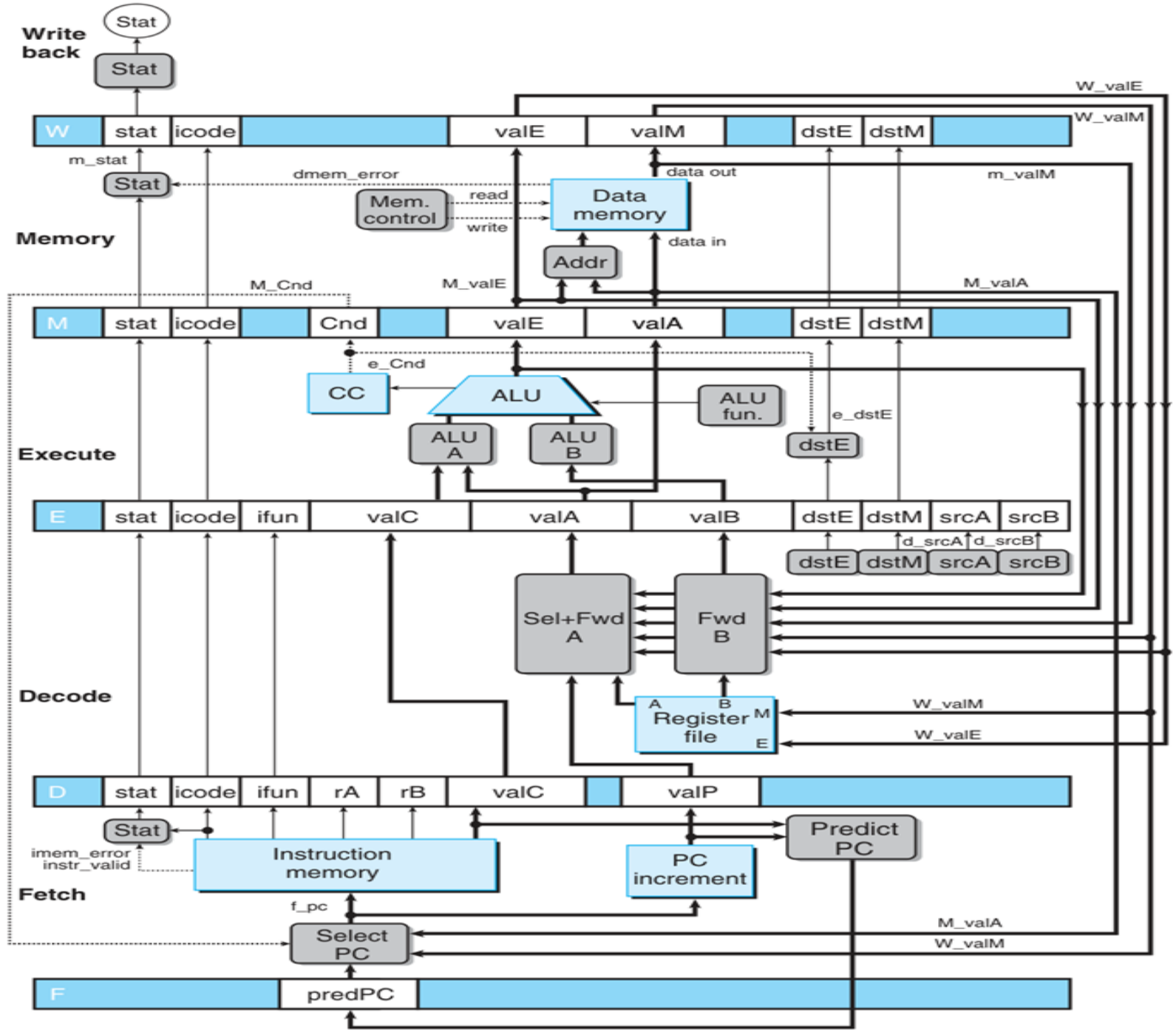
```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
word d_valB = [
        d_srcB == e_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];
```



	popq rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1 [PC+1] valP←PC+2
译码	valA ←R[%rsp] valB ← R[%rsp]
执行	valE ← valB + 8
访存	valM← M_8 [valA]
写回	R[%rsp] ← valE R[rA] ← valM
更新 pc	PC ← valP



	ret
取指	icode:ifun ← M_1[PC]
译码	valA ←R[%rsp] valB←R[%rsp]
执行	valE ← valB + 8
访存	valM← M_8 [valA]
写回	R[%rsp] ← valE
更新 pc	PC ← valM



	JXX Dest
取指	icode:ifun ← M_1[PC] valC← M_8 [PC+1] valP←PC+9
译码	
执行	Cnd ← Cond(CC, ifun)
访存	
写回	
更新pc	PC ← Cnd？ valC: valP