Computer Systems

第六次小班课

助教：罗兆丰

期中复习？

- 没有一定不会考的章节了orz

- 开始做往年题！（2012-2021）
- 一定是**连续的两小时**做一套，**做完再对答案**
- 可以整理错题
- 下周的小班课：模拟考试（50分钟，一半题量）

Processor Arch: Sequential

Basic Stages

- Fetch (icode, ifun, rA, rB, valC, valP)
- Decode (valA, valB)
- Execute (valE)
- Memory (valM)
- Write back
- PC update

Hardware

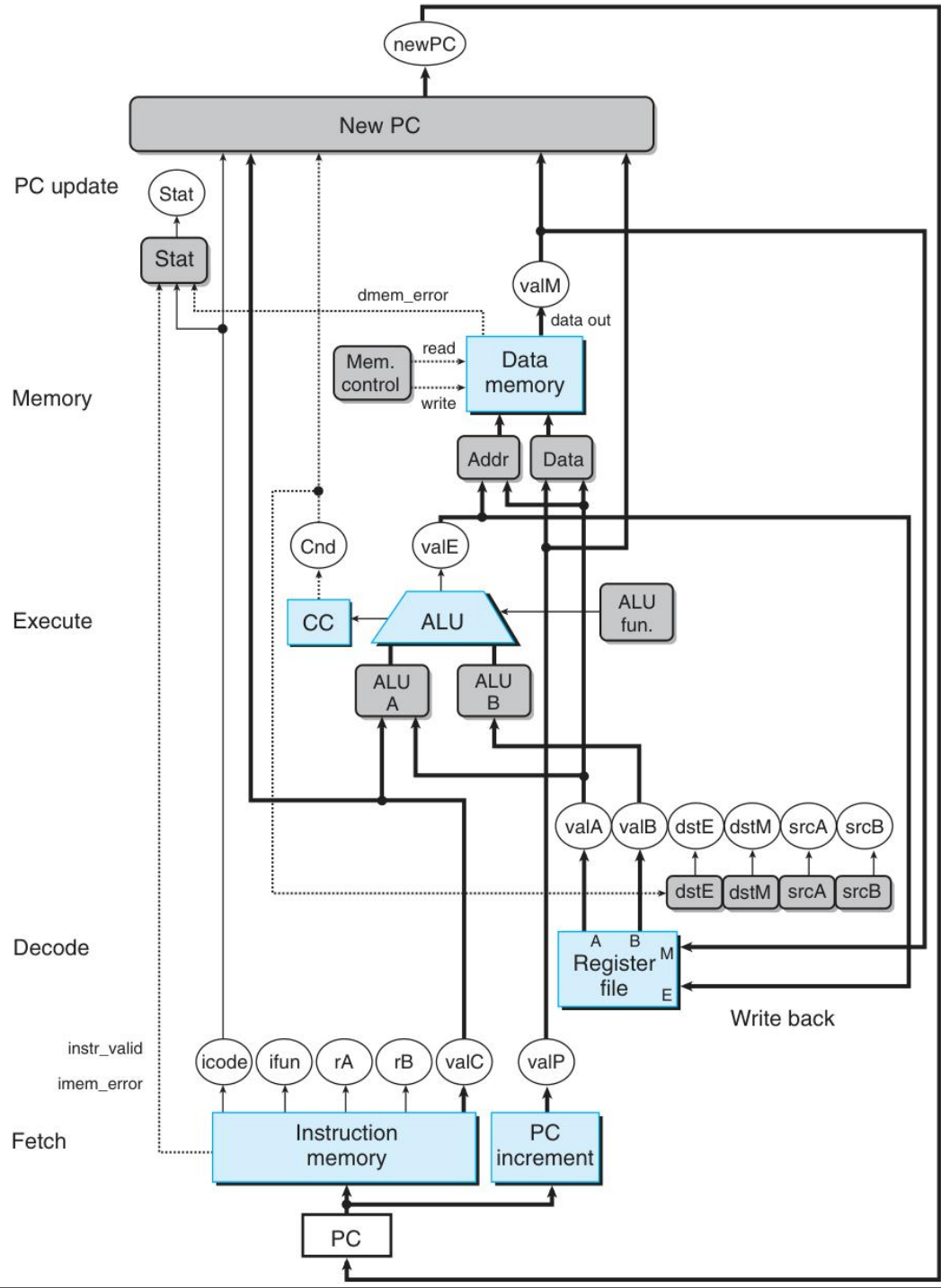
SEQ 中由时钟控制的东西有哪些？

- 程序计数器 PC
- 条件码寄存器 CC
- 寄存器文件的“写”端口
- 数据内存的“写”端口

其他东西都可以看成组合逻辑

- 有些硬件（比如寄存器文件的“读”端口）不是组合逻辑，但是可以“看成”组合逻辑

在将来流水线化之后，流水线寄存器也是时钟控制的



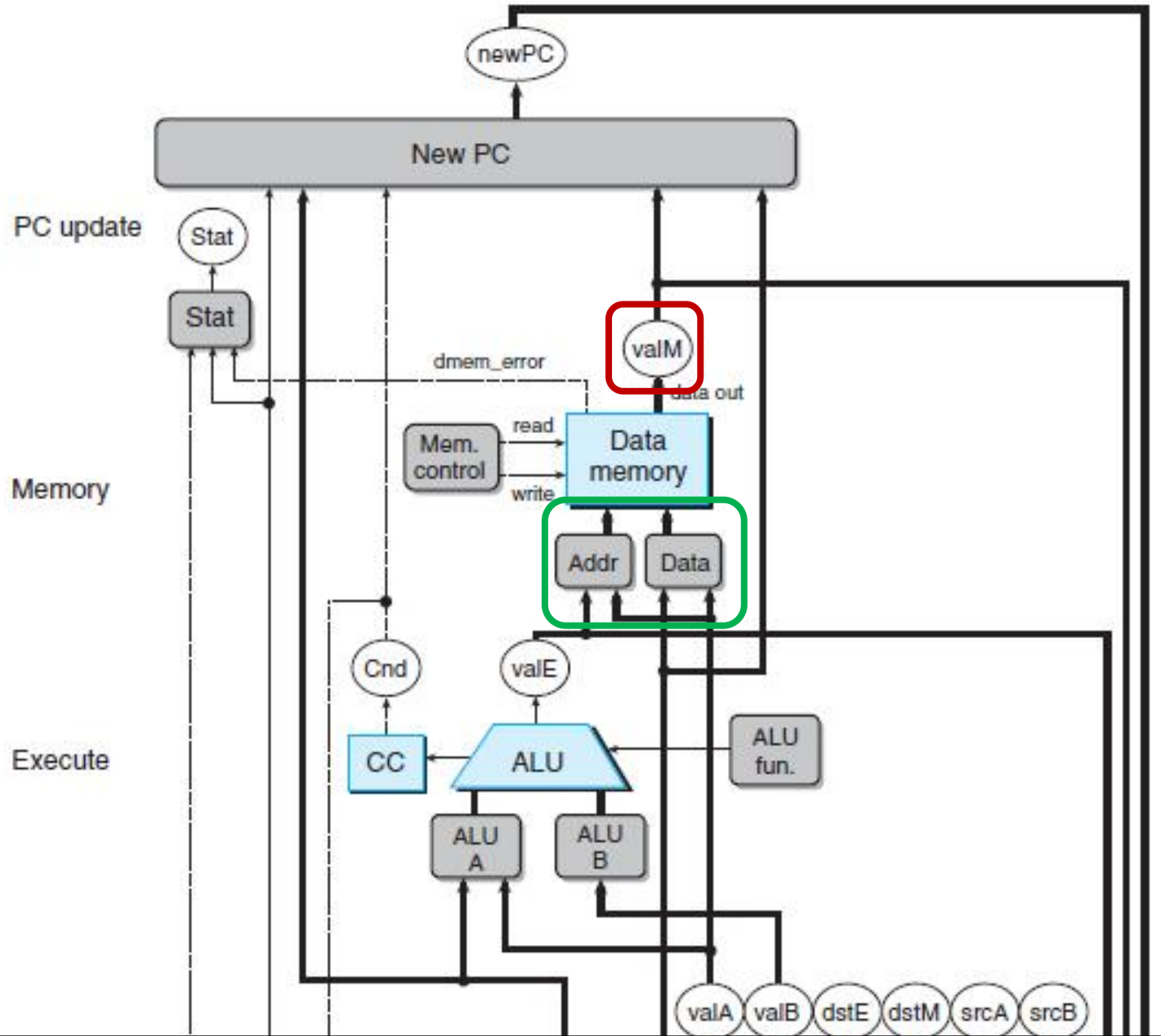
valM用于写回/更新pc

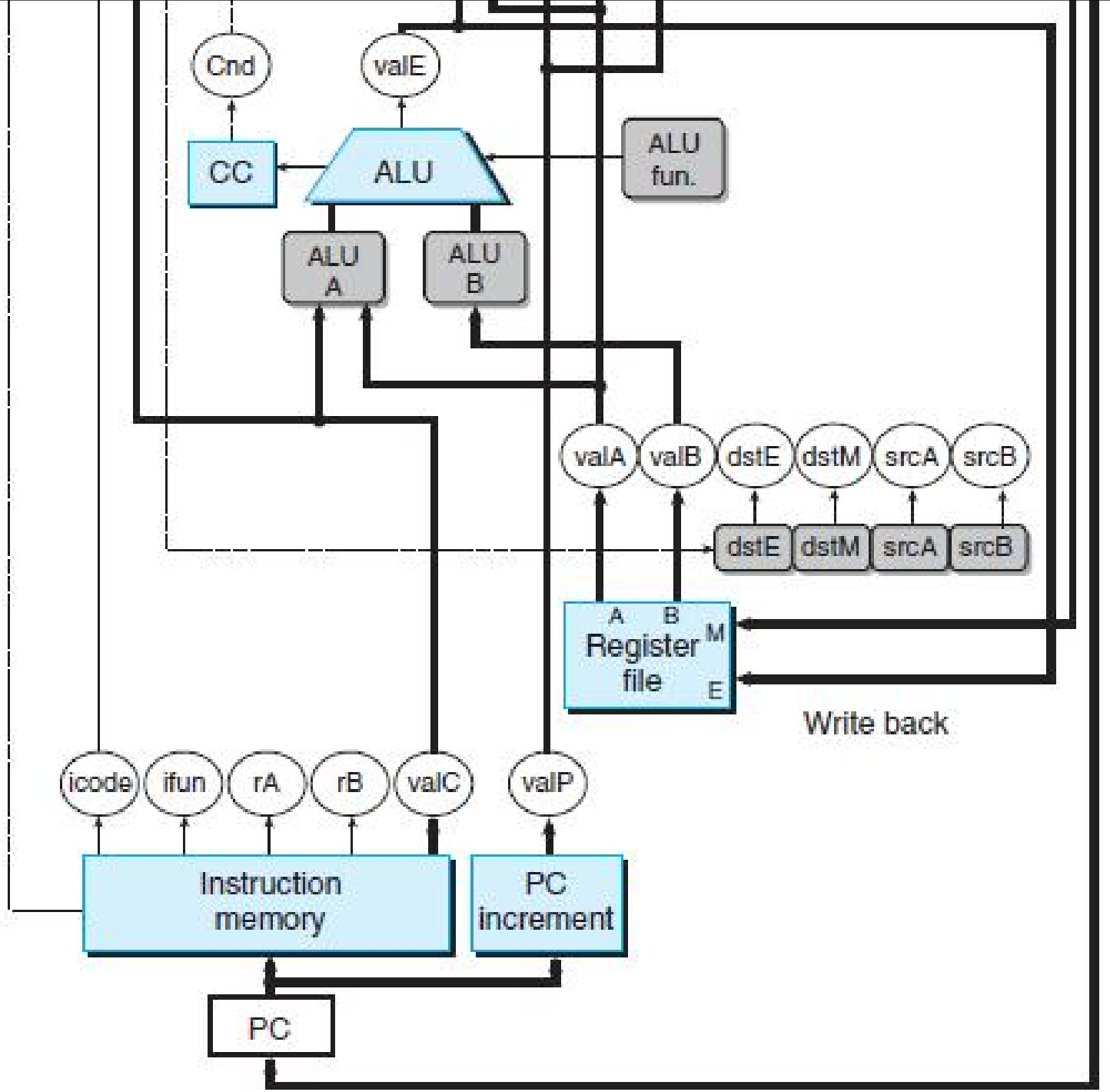
Addr = valE or valA
Data = valP or valA

valE = (valA or val C) OP B

回顾：为什么 mrmovq D(rB), rA?

- 这样就避免了 valA+valC，否则还得多加一条线路





写回只能用valE和valM

valC用于计算valE或更新PC

SEQ的寄存器堆端口允许2读2写

Instruction Stages

计算	OPq rA, rB	rrmovq rA, rB	irmovq V, rB	cmovXX rA, rB
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] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valP ← PC + 2
valA, srcA valB, srcB	valA ← R[rA] valB ← R[rB]	valA ← R[rA]		valA ← R[rA]
valE Cond. codes	valE ← valB OP valA Set CC	valE ← 0 + valA	valE ← 0 + valC	valE ← 0 + valA Cnd ← Cond(CC, ifun)
Read/write				
E port, dstE M port, dstM	R[rB] ← valE	R[rB] ← valE	R[rB] ← valE	if (Cnd) R[rB] ← valE
PC	PC ← valP	PC ← valP	PC ← valP	PC ← valP

Instruction Stages

阶段	rmmovq rA, D(rB)	mrmovq D(rB), rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
执行	valE ← valB+valC	valE ← valB+valC
访存	M_8[valE] ← valA	valE ← M_8[valE]
写回		
		R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

Instruction Stages

阶段	jXX Dest	call Dest	ret
取指	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valP ← PC+1
译码		valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	Cnd ← Cond(CC, ifun)	valE ← valB+(−8)	valE ← valB+8
访存		M_8[valE] ← valP	valM ← M_8[valA]
写回		R[%rsp] ← valE	R[%rsp] ← valE
更新 PC	PC ← Cnd?valC:valP	PC ← valC	PC ← valM

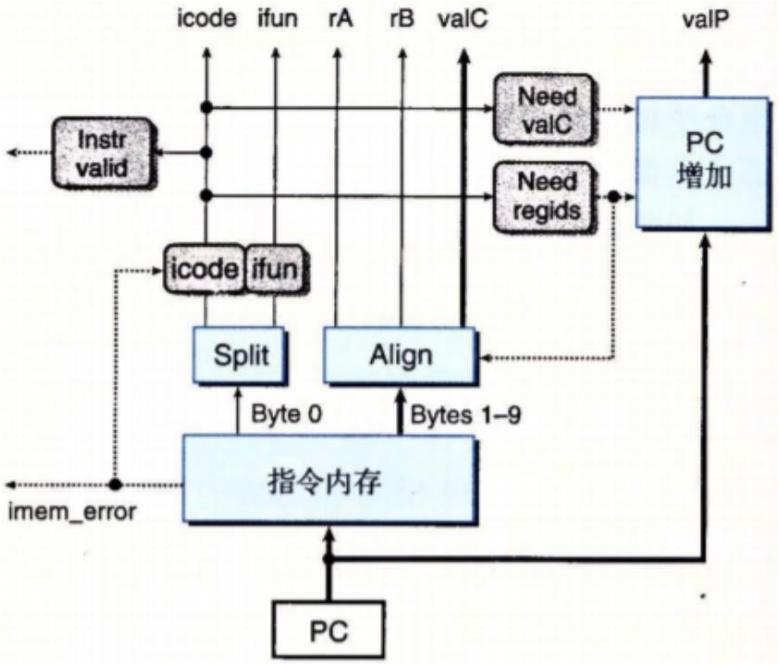
Instruction Stages

阶段	pushq rA	popq rA
取指	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] valP ← PC+2
译码	valA ← R[rA] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	valE ← valB+(−8)	valE ← valB+8
访存	M_8[valE] ← valA	valM ← M_8[valA]
写回	R[%rsp] ← valE	R[%rsp] ← valE R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

Fetch

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

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



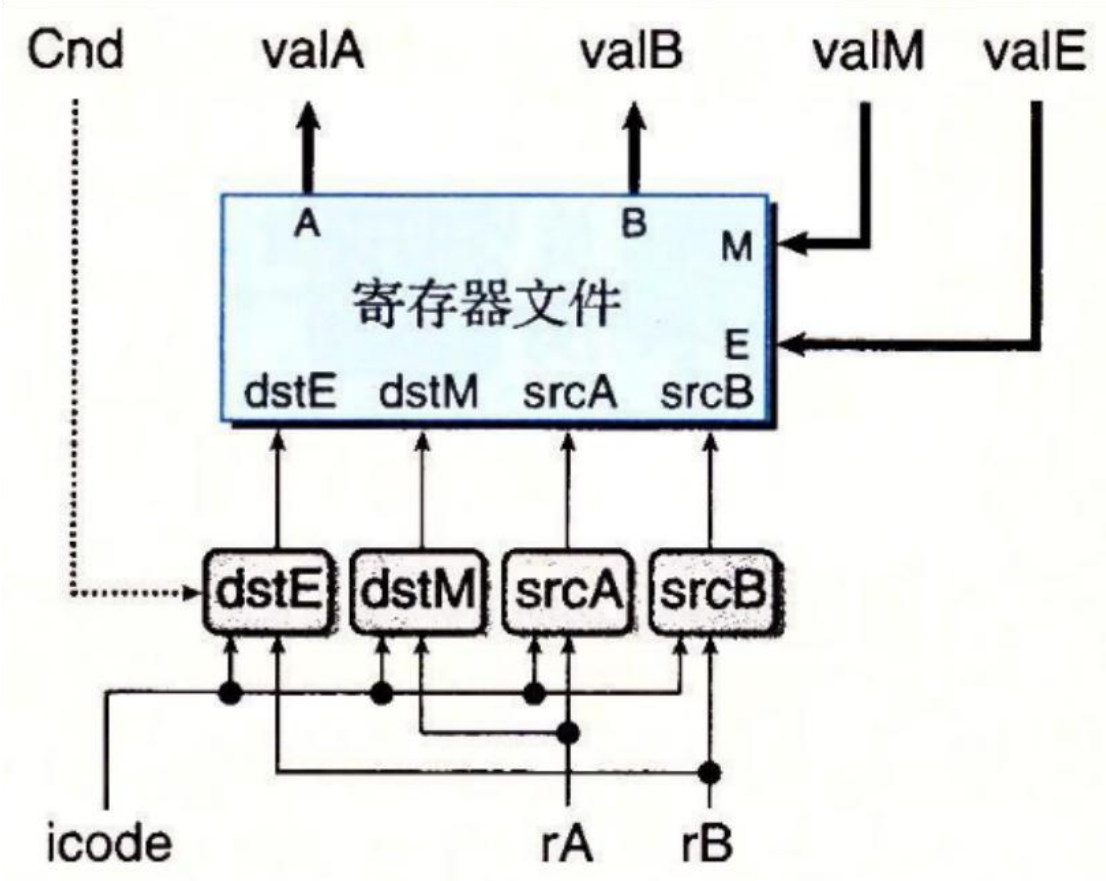

```
# Determine instruction code
int icode = [
        imem_error: INOP;
        1: imem_icode;
];

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

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

Decode

HCL:

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



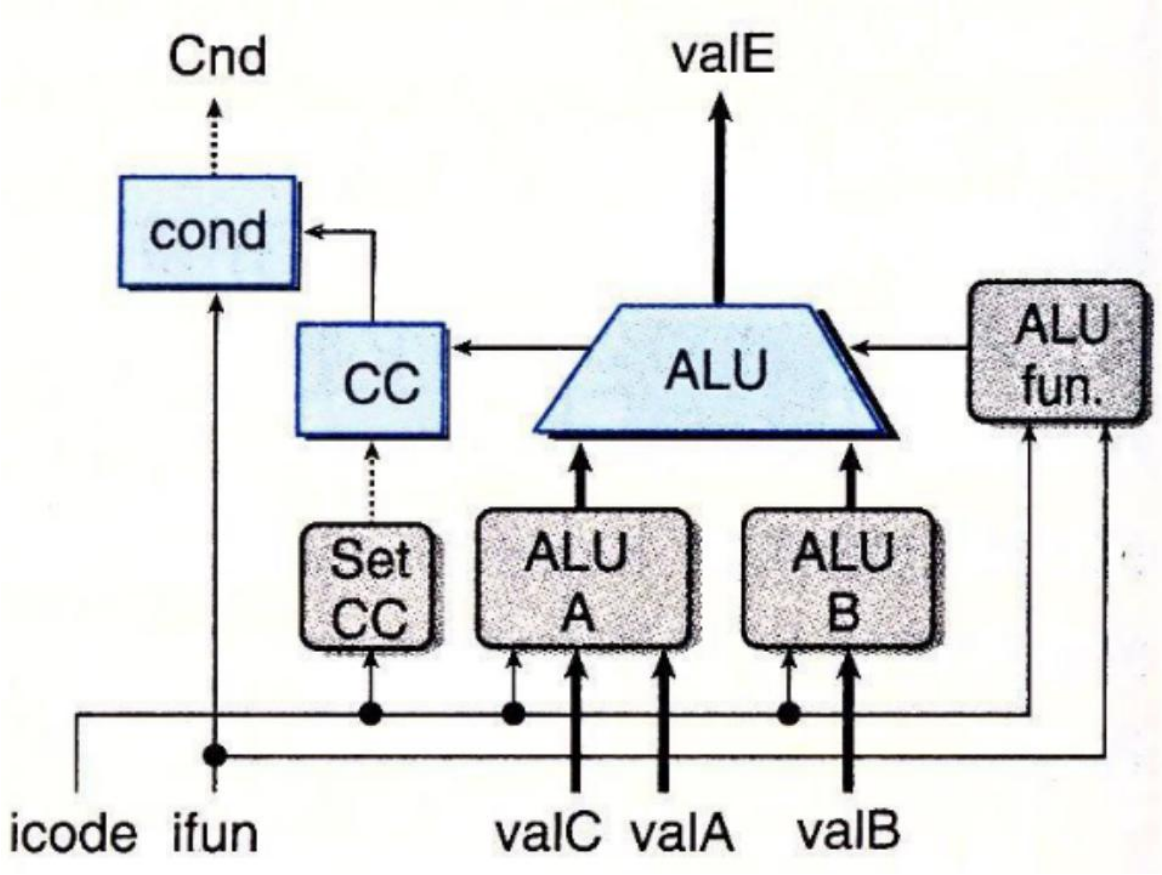
Execute

```
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
];
```

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

```
bool set_cc = icode in { IOPQ };
```

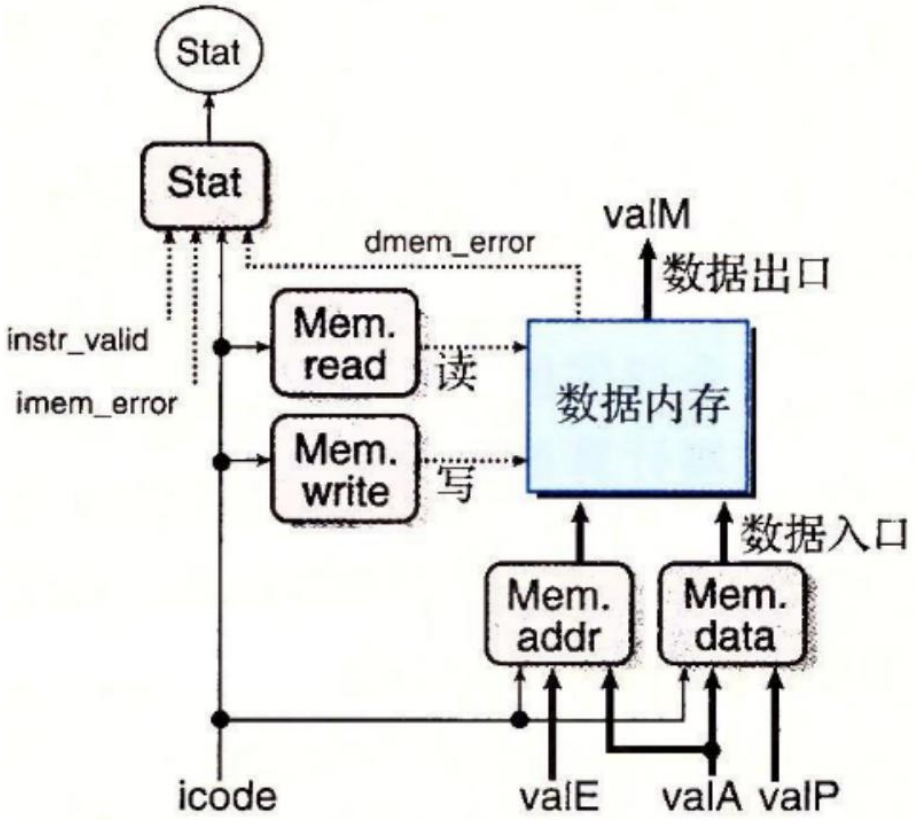


Q:上面这张图里有什么东西是时钟控制的？

Memory

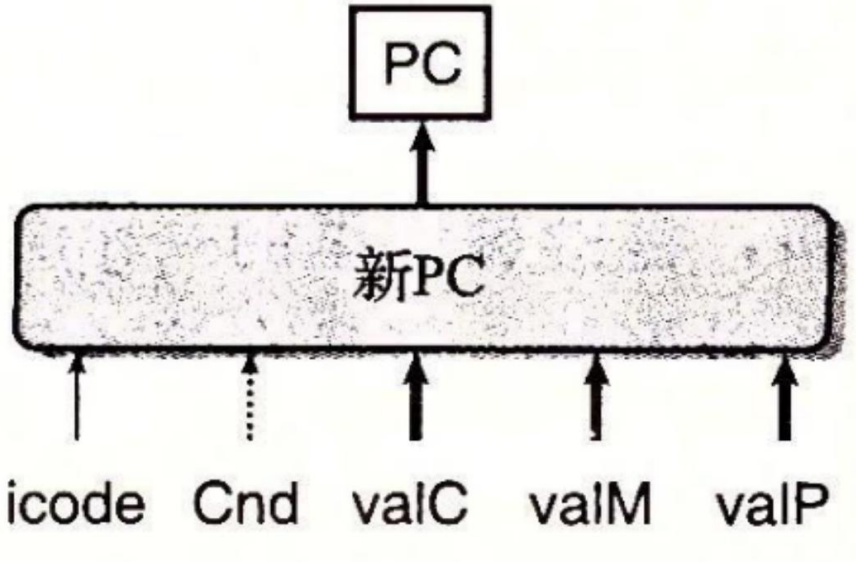
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];
word mem_data = [
  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

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



New 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;
];
```



应试小技巧：如何写出分解指令的阶段级分解？

- 这是助教自己做这类题的方法，仅供参考~
- 第一步：分析指令的运行前后，**有哪些可见状态发生了改变**
  - PC
  - 寄存器（特别是 %rsp）
  - 数据内存
  - CC
- 第二步：根据处理器设计中的一些“原则”，**从后往前**倒推指令的阶段分解
  - 访存的地址只能是 valE 或 valA
  - 存入数据内存的数据只能是 valP 或 valA
  - valE = (valA or valC) OP valB
  - 有时候可能还需要结合一些“记忆”，比如在 Y86-64 的已有指令中，E 阶段用到的常数只有 0 或 ±8
    - 所有涉及到 ±8 的计算（也是更新%rsp的操作）都是 “valE=valB ± 8”
    - 所有涉及到 0 的计算（其实只有 rrmovq 和 irmovq）都是 valE=valA/valC + 0

阶段	pushq rA	popq rA
取指	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] valP ← PC+2
译码	valA ← R[rA] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	valE ← valB+(−8)	valE ← valB+8
访存	M_8[valE] ← valA	valM ← M_8[valA]
写回	R[%rsp] ← valE	R[%rsp] ← valE R[rA] ← valM
更新 PC	PC ← valP	PC ← valP

- 在Y86-64中实现指令

leave，该指令等价于：

```
rrmovq %rbp, %rsp
popq %rbp
```

分析：

$rbp + 8 -> %rsp

M8[%rbp] -> %rbp

- 在Y86-64中实现指令

leave，该指令等价于：

```
rrmovq %rbp, %rsp
popq %rbp
```

分析：　　valE

$rbp + 8 -> %rsp

M8[%rbp] -> %rbp

valM

- 在Y86-64中实现指令leave，该指令等价于：

```
rrmovq %rbp, %rsp
popq %rbp
```

分析：

```
$rbp + 8 -> %rsp
M8[%rbp] -> %rbp
```

valE

valM

Stage	Action
Fetch	icode:ifun <- M1[PC] valP <- PC + 1
Decode	
Execute	
Memory	
Write back	
PC update	PC <- valP

- 在x86-64中也有指令enter，该指令等价于：

```
pushq %rbp
rrmovq %rsp, %rbp
```

Y86-64中能实现吗？

分析：

```
$rbp -> M8[%rsp-
8]
$rsp-8 -> %rbp
$rsp-8 -> %rsp
```

指令特征	指令	说明
写内存	rmmovq，pushq，call	
读内存	mrmovq，popq，ret	
访存计算索引	rmmovq，mrmovq，pushq，call	用valE索引
访存不计算索引	popq，ret	用valA索引
操作栈指针	pushq，popq，call，ret	
设置条件码	OPq	
使用条件码	jXX	
……	……	……

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

valM

6、在Y86-64的PIPE实现中，仅考虑ICALL、IPOPQ、IPUSHQ、IRET指令，对mem_addr的HCL描述正确的是：

```
word mem_addr = [
        M_icode in { ①, ② } : M_valE;
        M_icode in { ③, ④ } : M_valA;
];
```

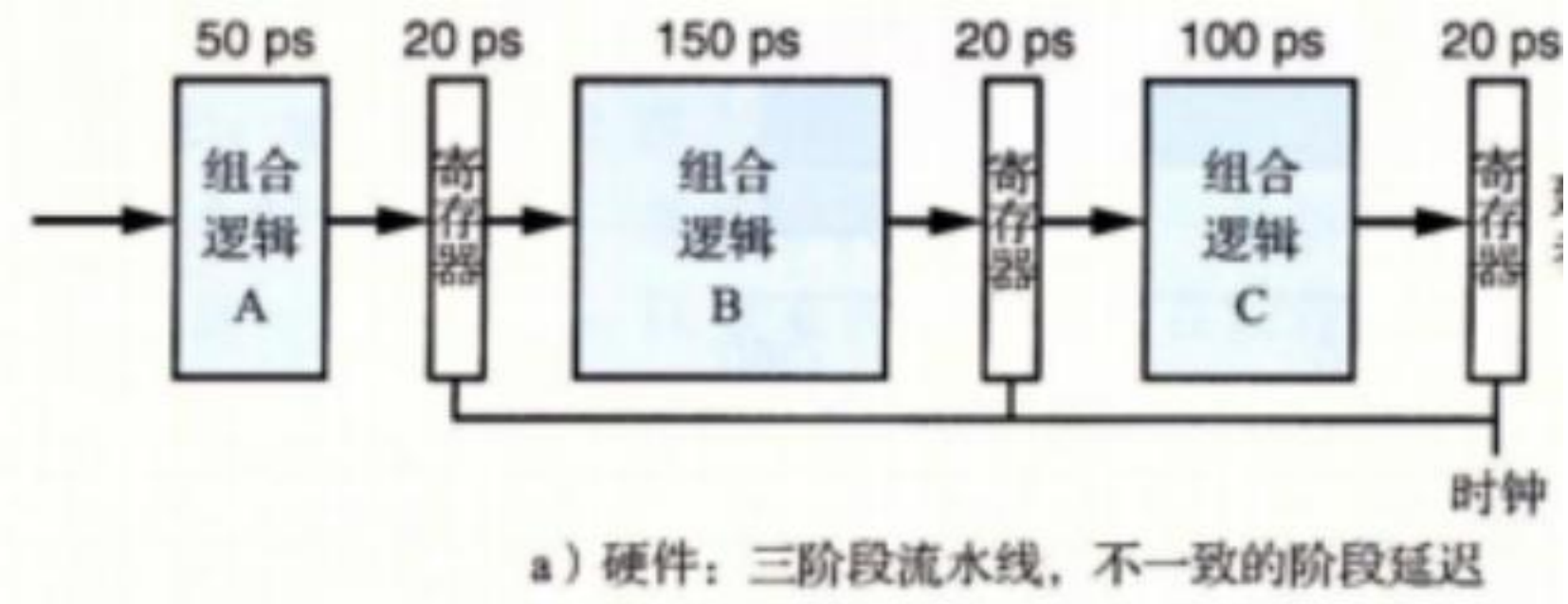
ICALL IPUSHQ

IPOPQ IRET

Processor Arch: Pipelined

Pipeline Basics

- Throughput and latency computation

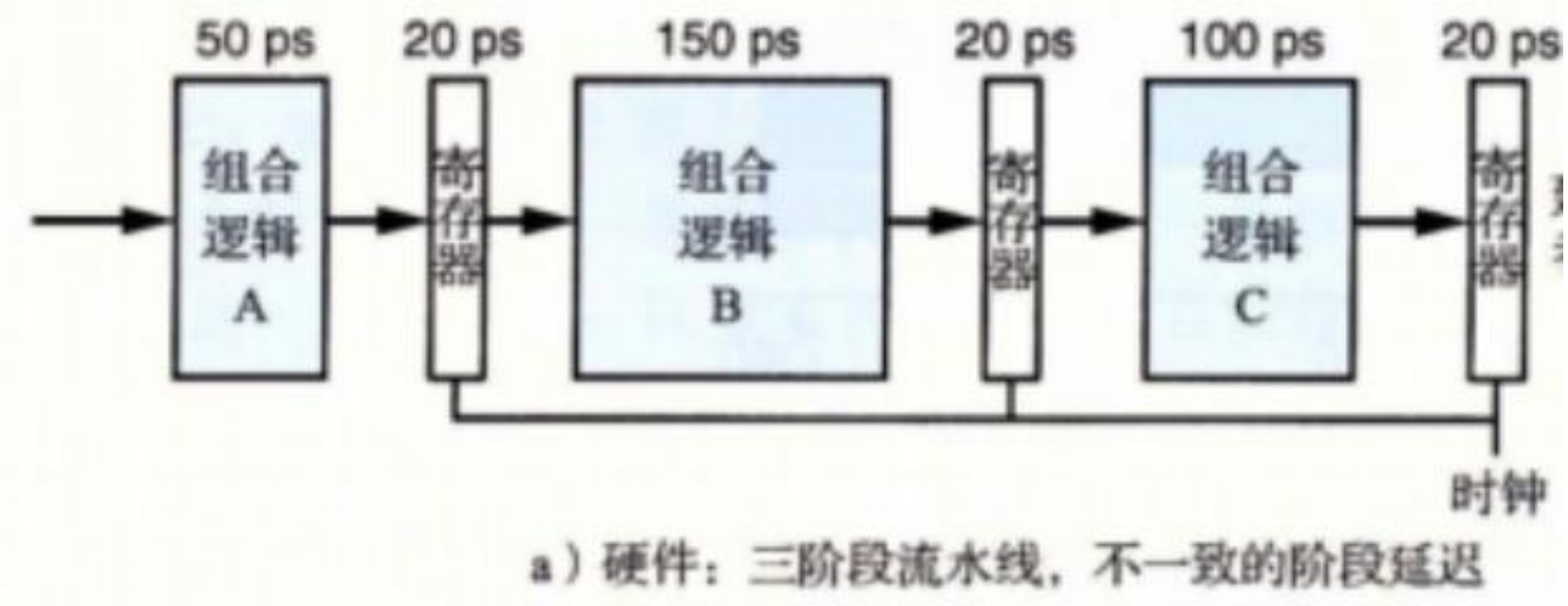


a）硬件：三阶段流水线，不一致的阶段延迟

延迟=?
吞吐量=?

Pipeline Basics

- Throughput and latency computation



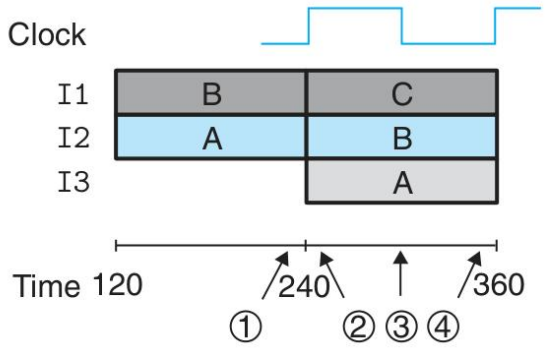
a）硬件：三阶段流水线，不一致的阶段延迟

延迟 = 3 * (max(50, 150, 100) +20) = 3 * (150 + 20) = 510ps

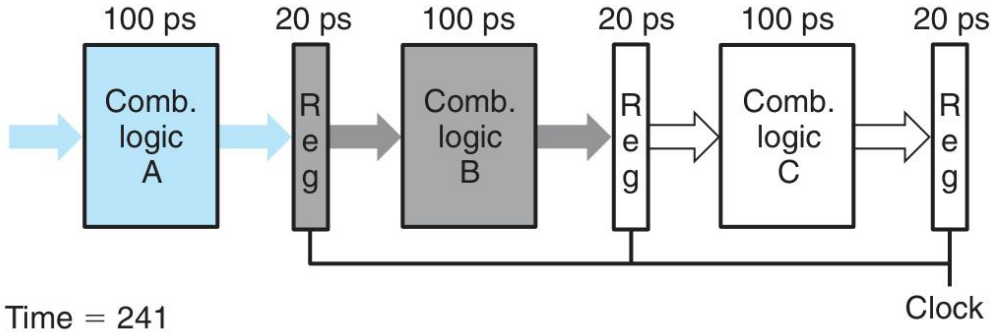
吞吐量 = (1 / 170ps) * (1000ps/1ns) = 5.88GIPS

Pipeline Basics

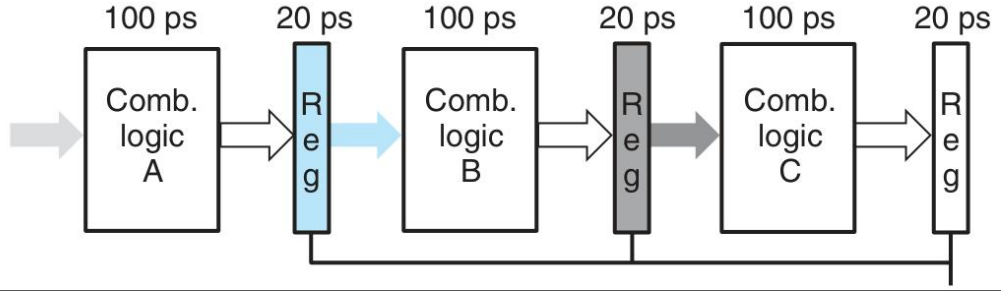
- Critical moments



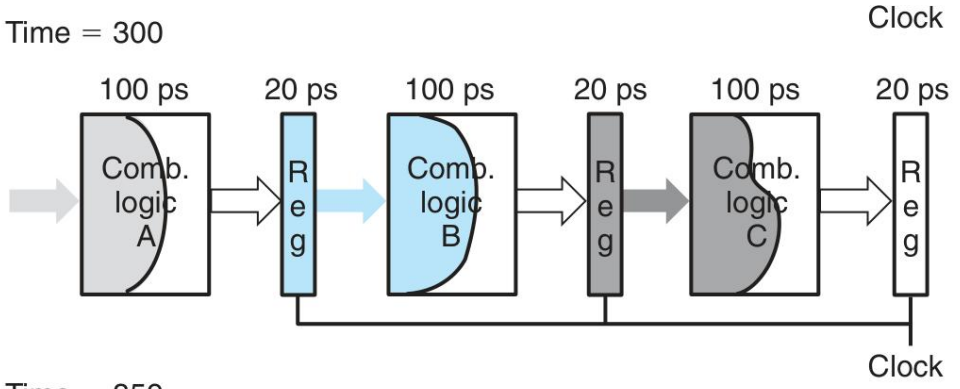
① Time = 239



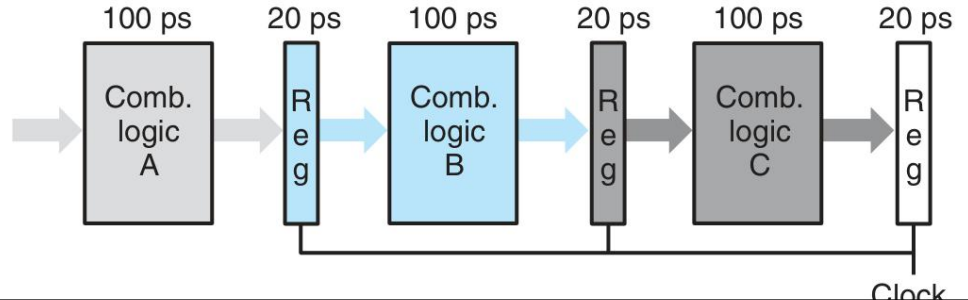
② Time = 241



③ Time = 300



④ Time = 359

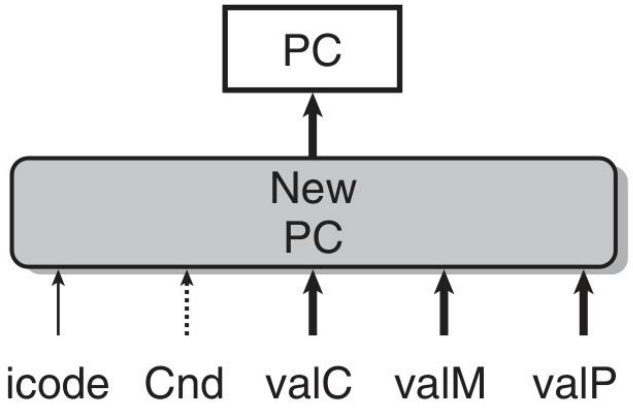


Pipeline Basics

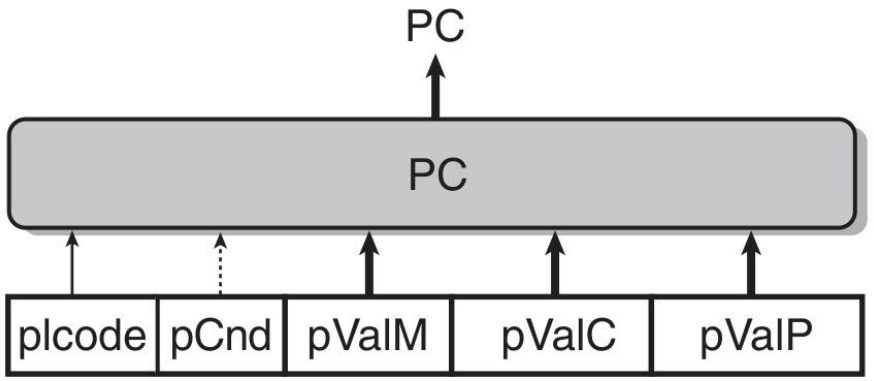
- **Limitations of pipelining**
  - Nonuniform partitioning
  - Decreasing returns to pipeline depth

- **Stages**
  - AMD Zen2(3th Gen Ryzen): 19 stages
  - Intel Ice Lake(10th Gen core): 14-19 stages

SEQ->SEQ+

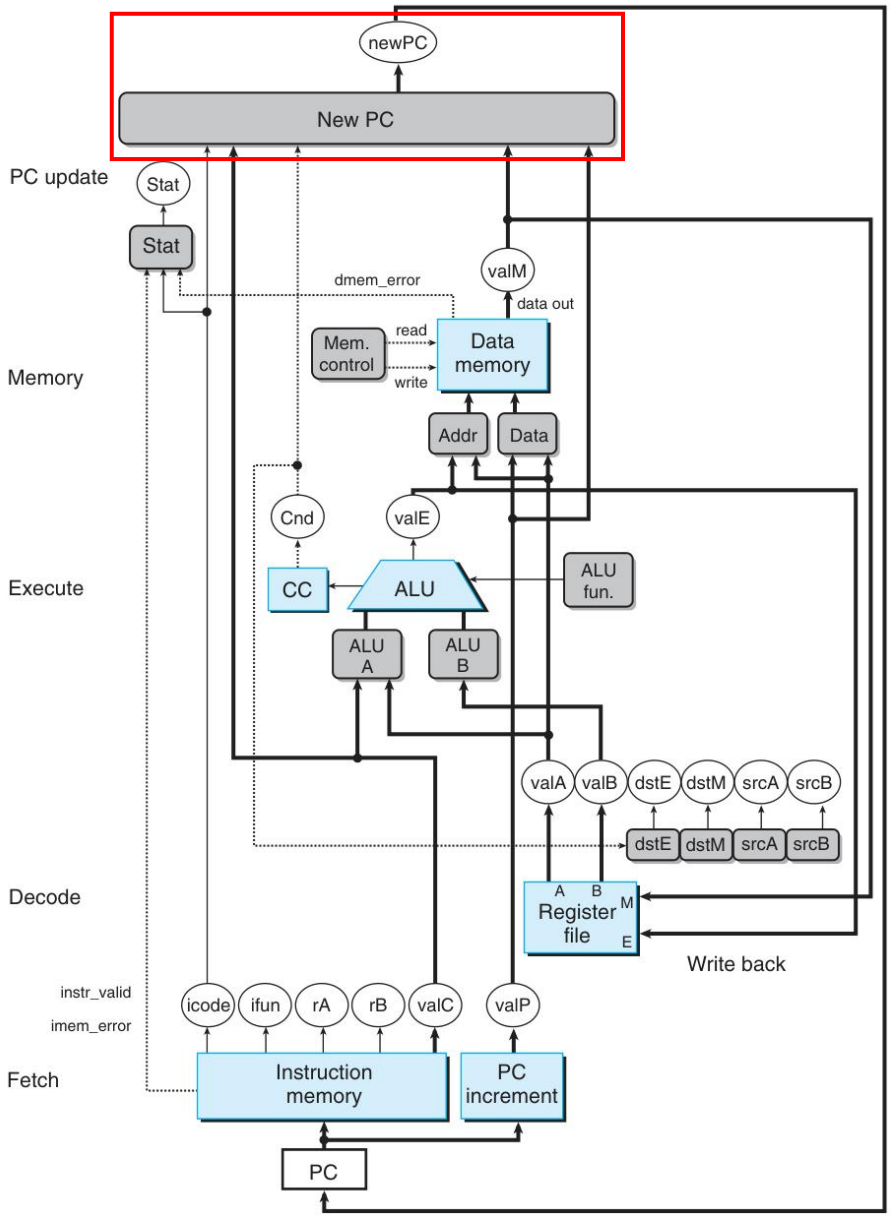


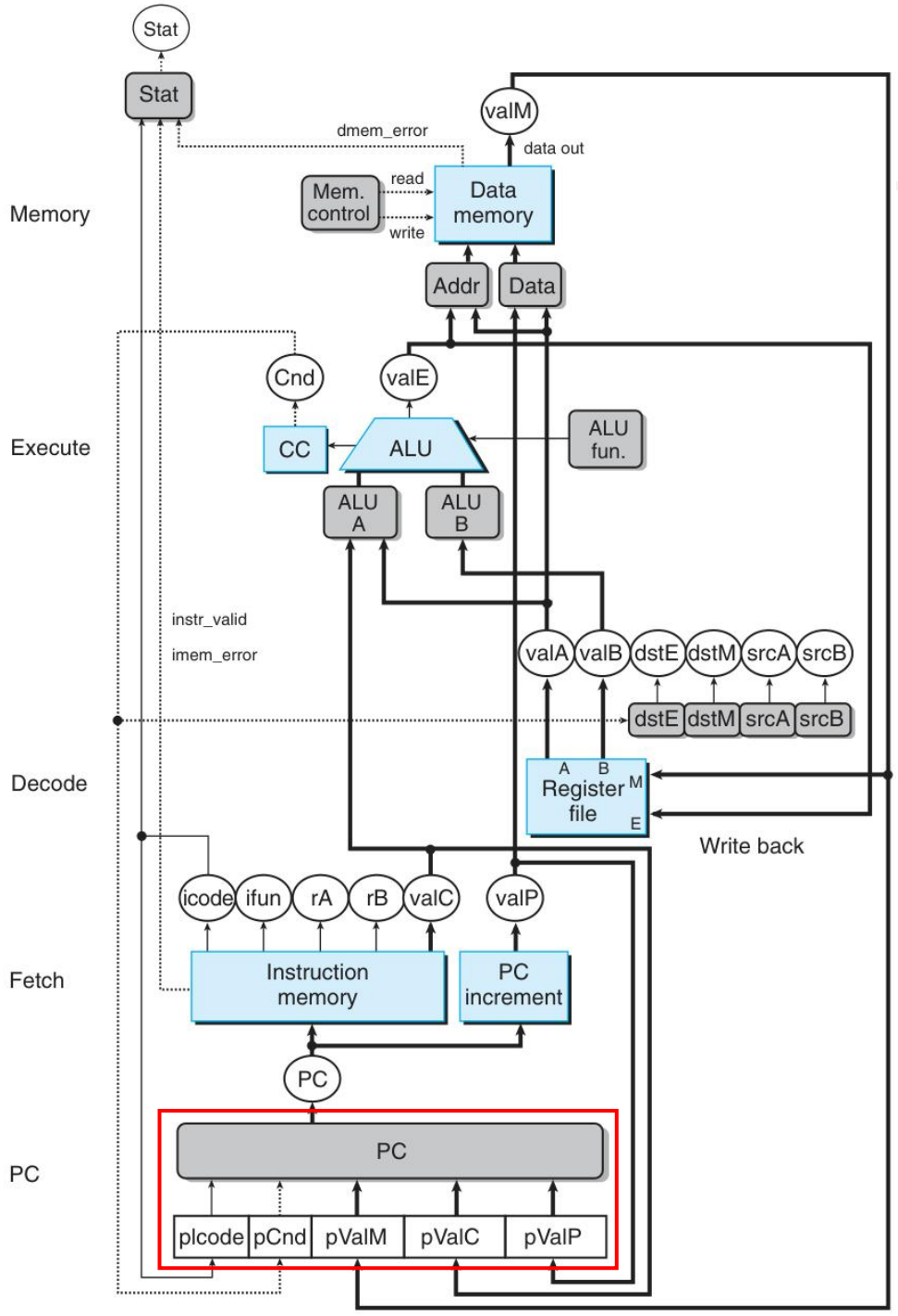
(a) SEQ new PC computation



(b) SEQ+ PC selection

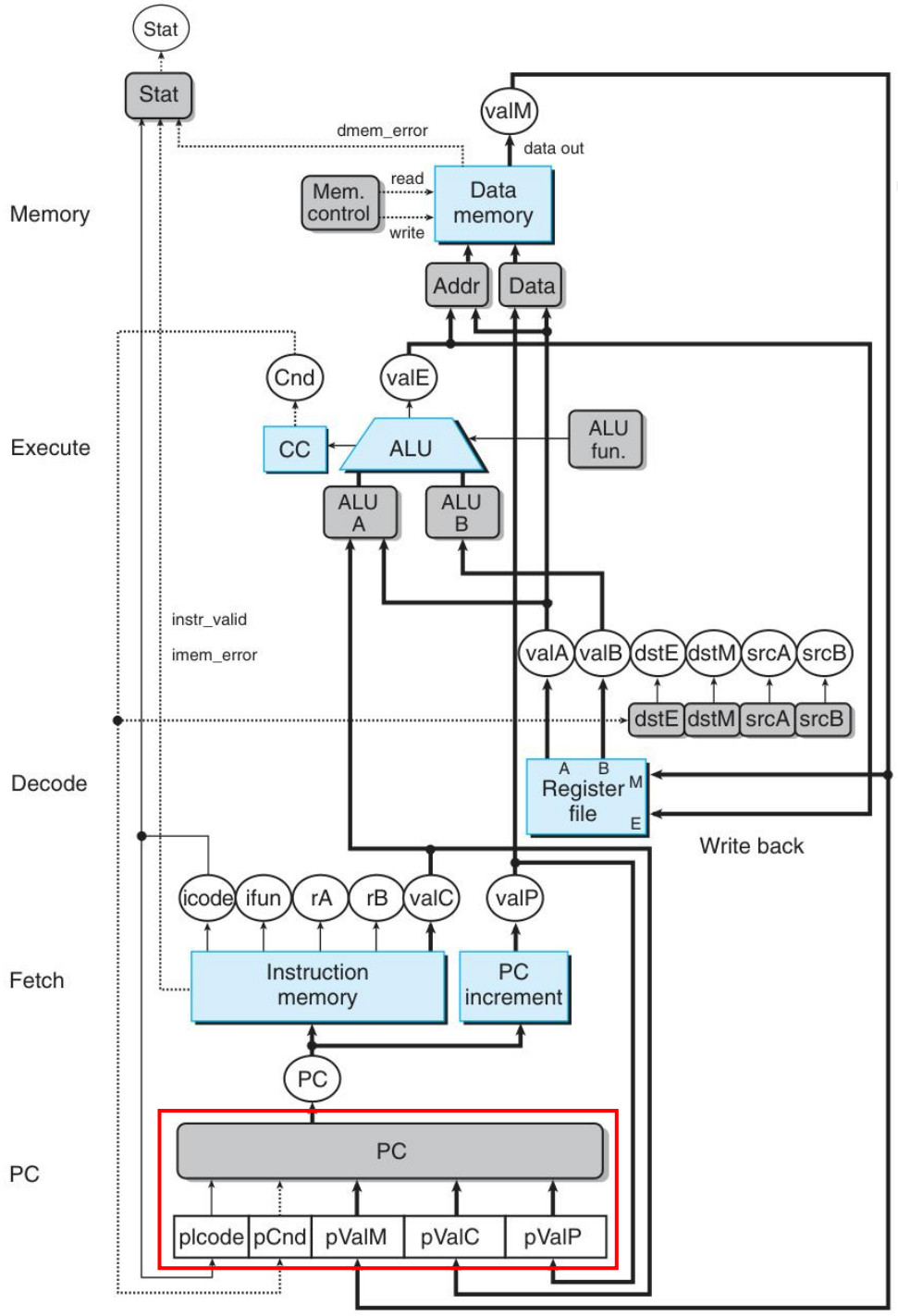
SEQ->SEQ+


newPC
New PC
PC update
Stat
Stat
dmem_error
valM
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
Write back
instr_valid
imem_error
icode
ifun
rA
rB
valC
valP
Fetch
Instruction memory
PC increment
PC

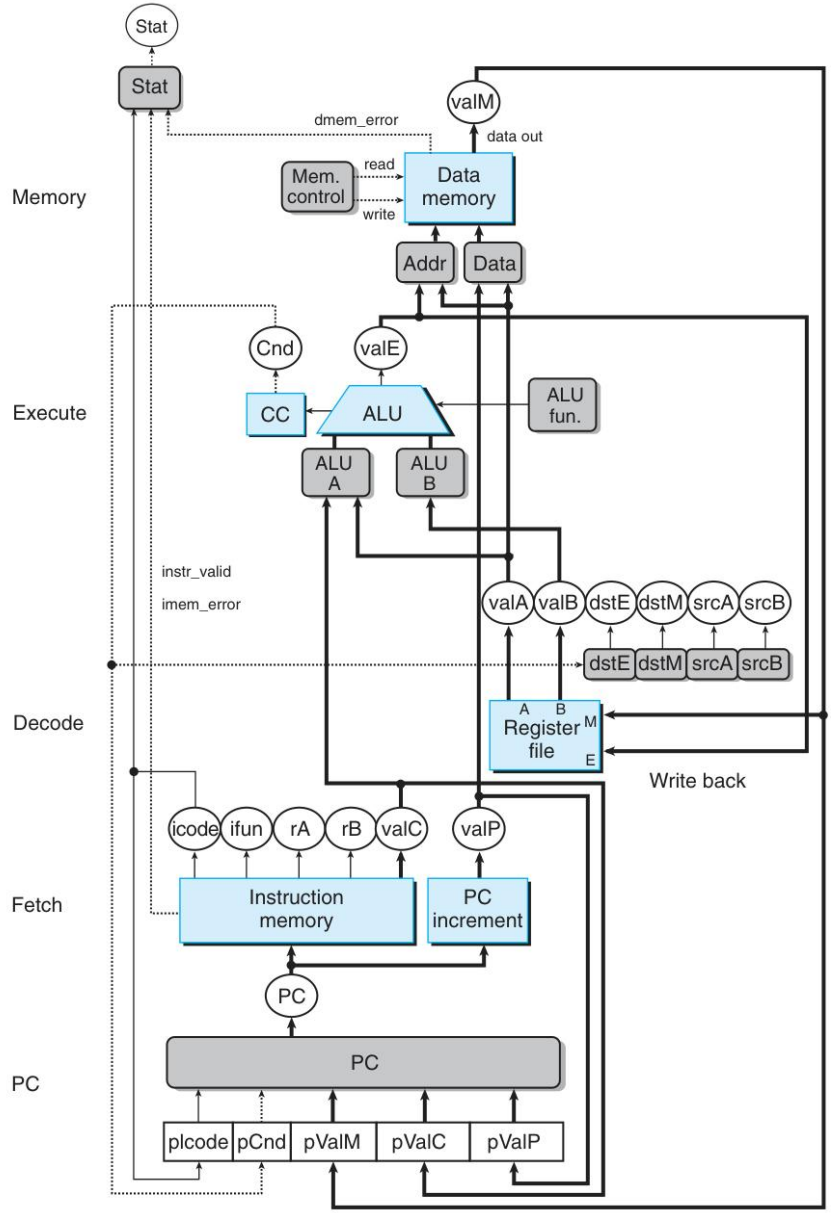

Stat
Stat
valM
dmem_error
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
instr_valid
imem_error
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
Write back
icode
ifun
rA
rB
valC
valP
Fetch
Instruction memory
PC increment
PC
PC
plcode
pCnd
pValM
pValC
pValP

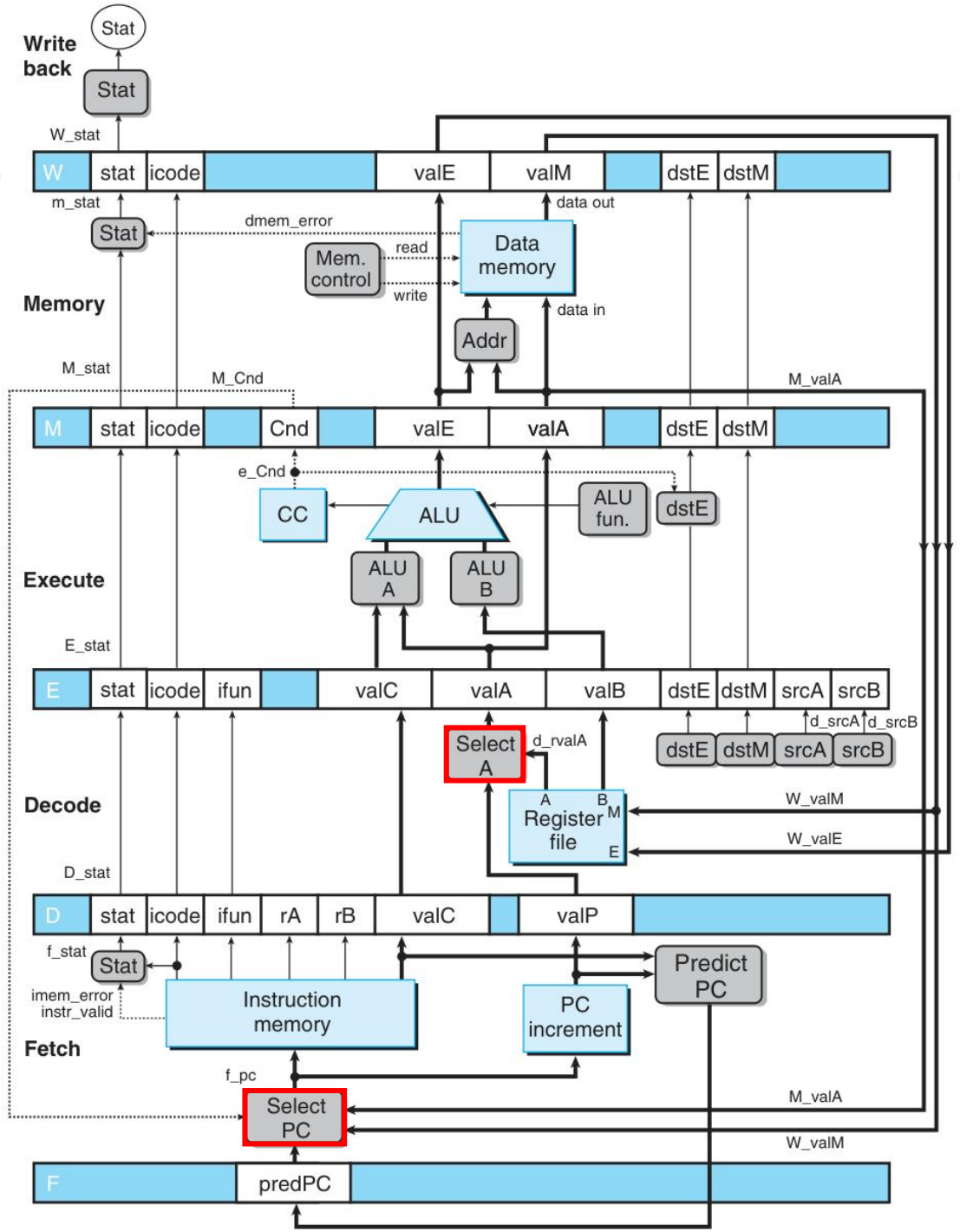
SEQ->SEQ+

- Circuit retiming
- PC update stage
- No hardware PC registers



SEQ+->PIPE-

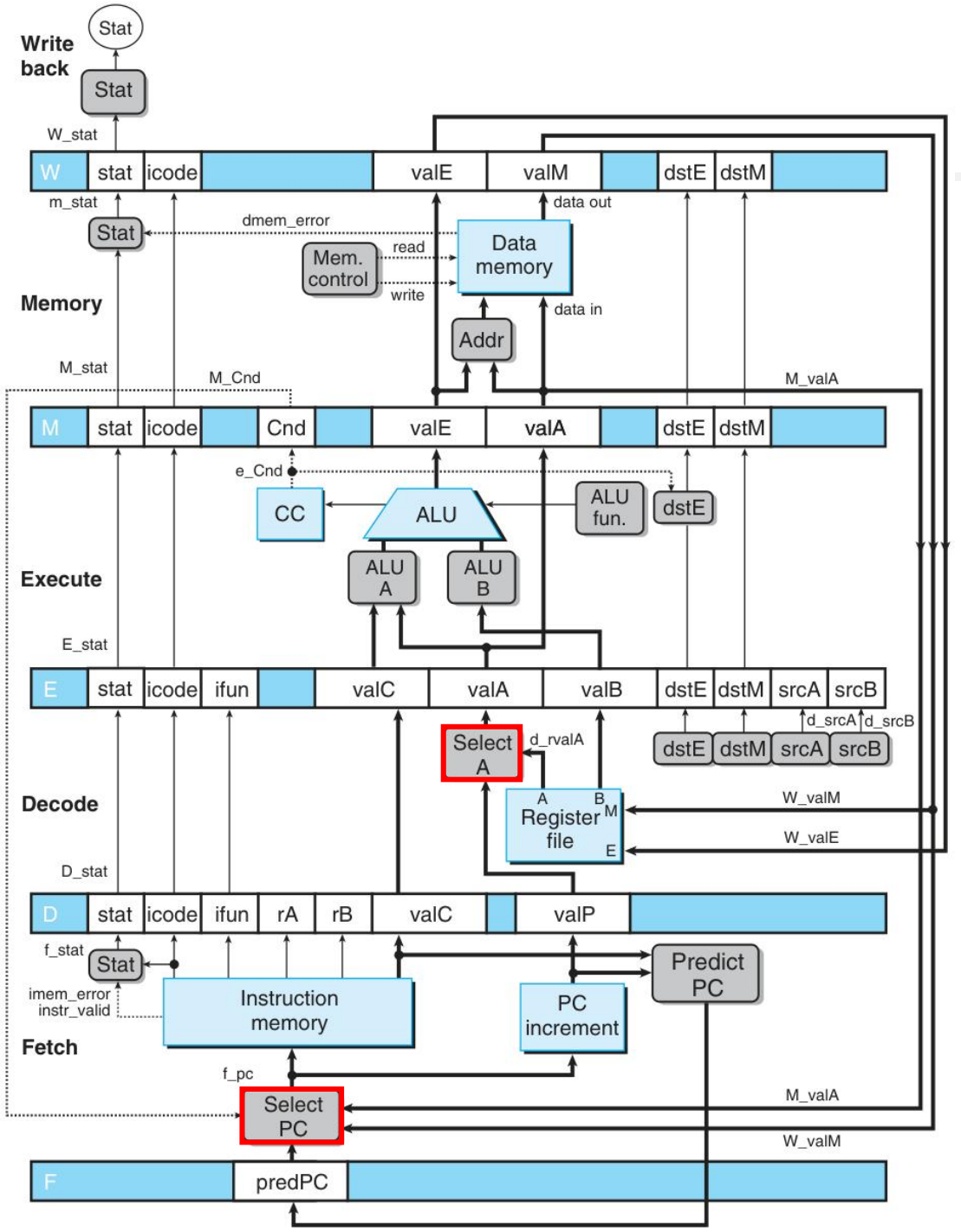

Stat
Stat
valM
dmem_error
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
instr_valid
imem_error
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
Write back
icode
ifun
rA
rB
valC
valP
Fetch
Instruction memory
PC increment
PC
PC
PC
plcode
pCnd
pValM
pValC
pValP


Write back
Stat
Stat
W_stat
W
stat
icode
valE
valM
dstE
dstM
m_stat
Stat
dmem_error
data out
Mem. control
read
write
Data memory
Memory
data in
Addr
M_stat
M_Cnd
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
dstE
Execute
ALU A
ALU B
E_stat
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
Select A
d_rvalA
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
W_valM
W_valE
D_stat
D
stat
icode
ifun
rA
rB
valC
valP
f_stat
Stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
Fetch
f_pc
M_valA
Select PC
W_valM
F
predPC

SEQ+->PIPE-

- Adding pipeline registers
- **Select PC**
- **Select A:** Since valP is only used in the Memory period of call and in the Execute period of jXX and both of them do not need valA, Select A module is used to reduce the number of registers

Select A 体现了“尽量减少寄存器状态和线路的数量”这个设计哲学



SEQ+->PIPE-

各寄存器的作用

F　　保存程序计数器的预测值

D　　保存取指信息

E　　保存关于最新译码的指令、从寄存器文件中读出的值

M　　保存最新执行指令的结果、 关于处理条件转移的分支条件和目标信息

W　　位于访存和反馈路径之间
ret指令会向PC选择逻辑提供返回地址

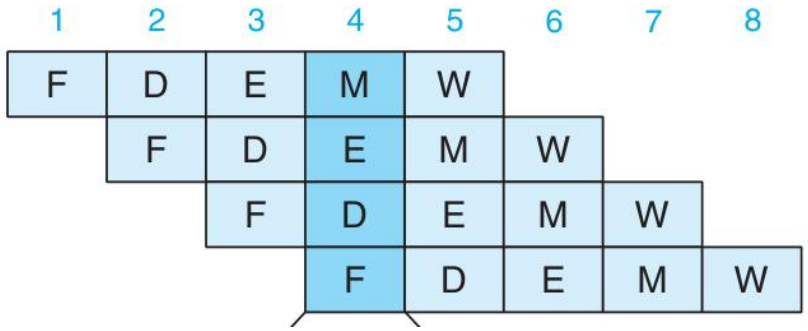
- S_Field
  - Value of Field held in stage S pipeline register
- s_Field
  - Value of Field computed in stage S

Data word	Register ID	Source description
e_valE	e_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage
W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage

Problems: Data/Control Dependency

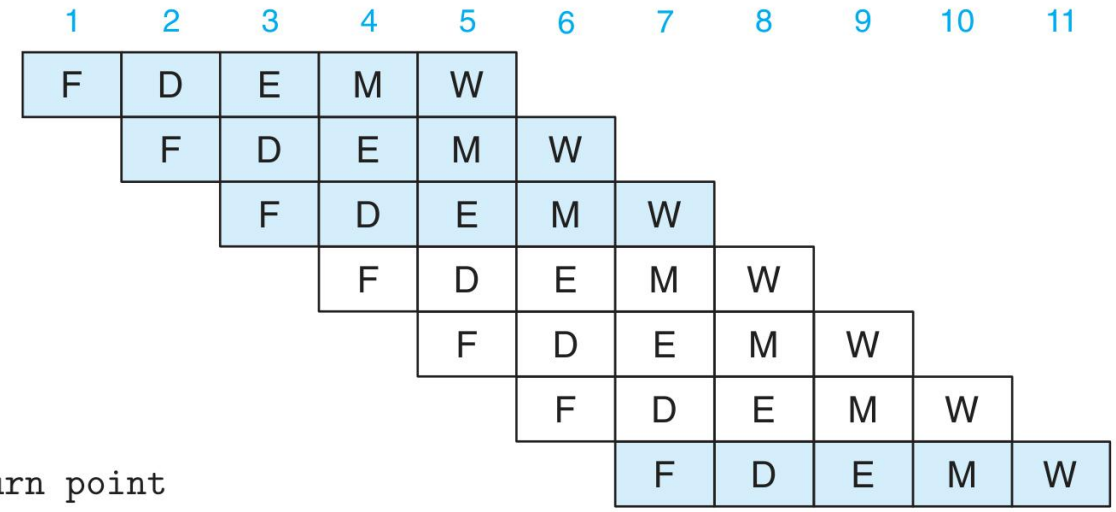
- Data hazard

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



- Control hazard

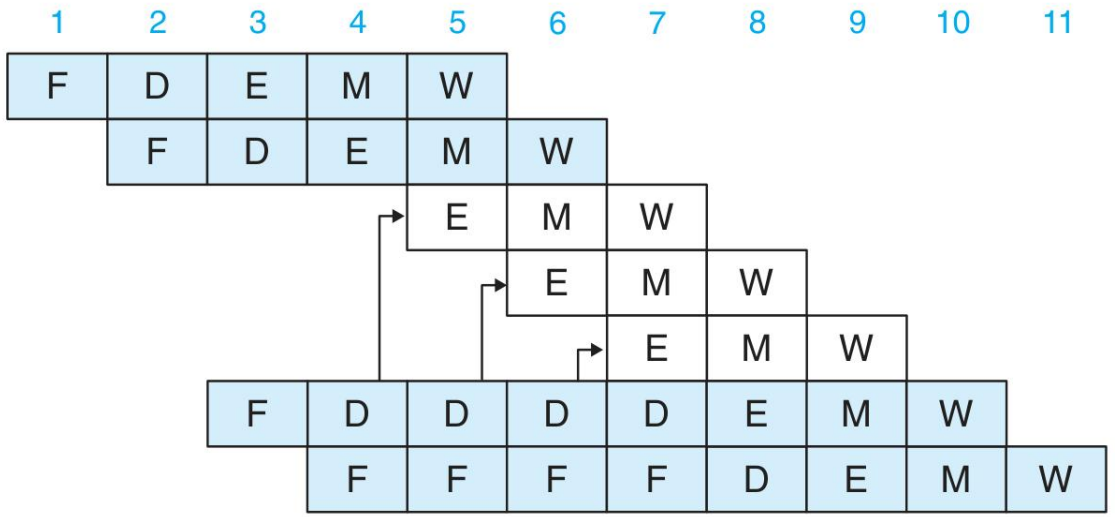
```
# prog7
0x000: irmovq Stack,%edx
0x00a: call proc
0x020: ret
       bubble
       bubble
       bubble
0x013: irmovq $10,%edx # Return point
```



A Simple Solution: Bubbles and Stalls

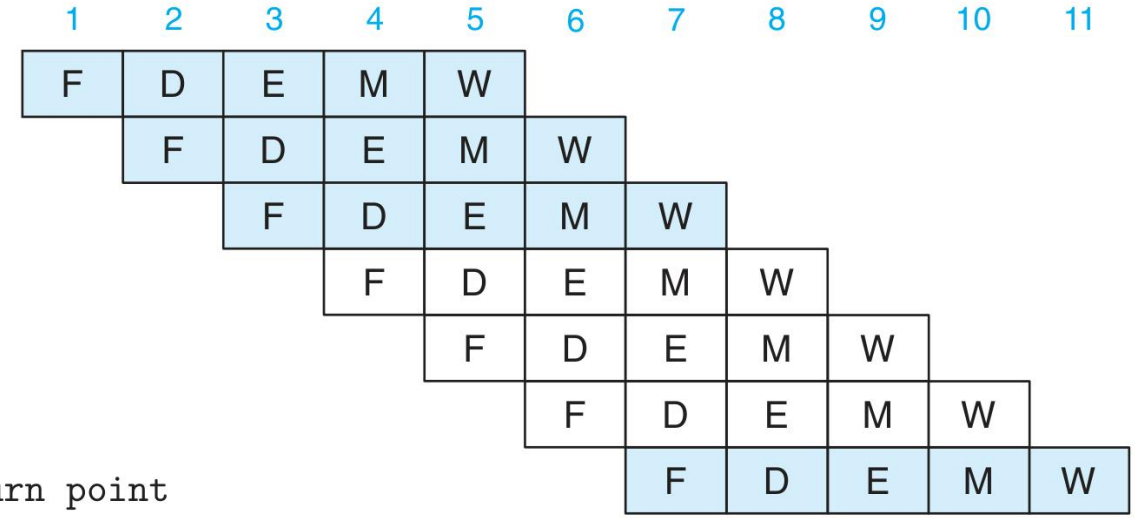
Handling data hazard

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



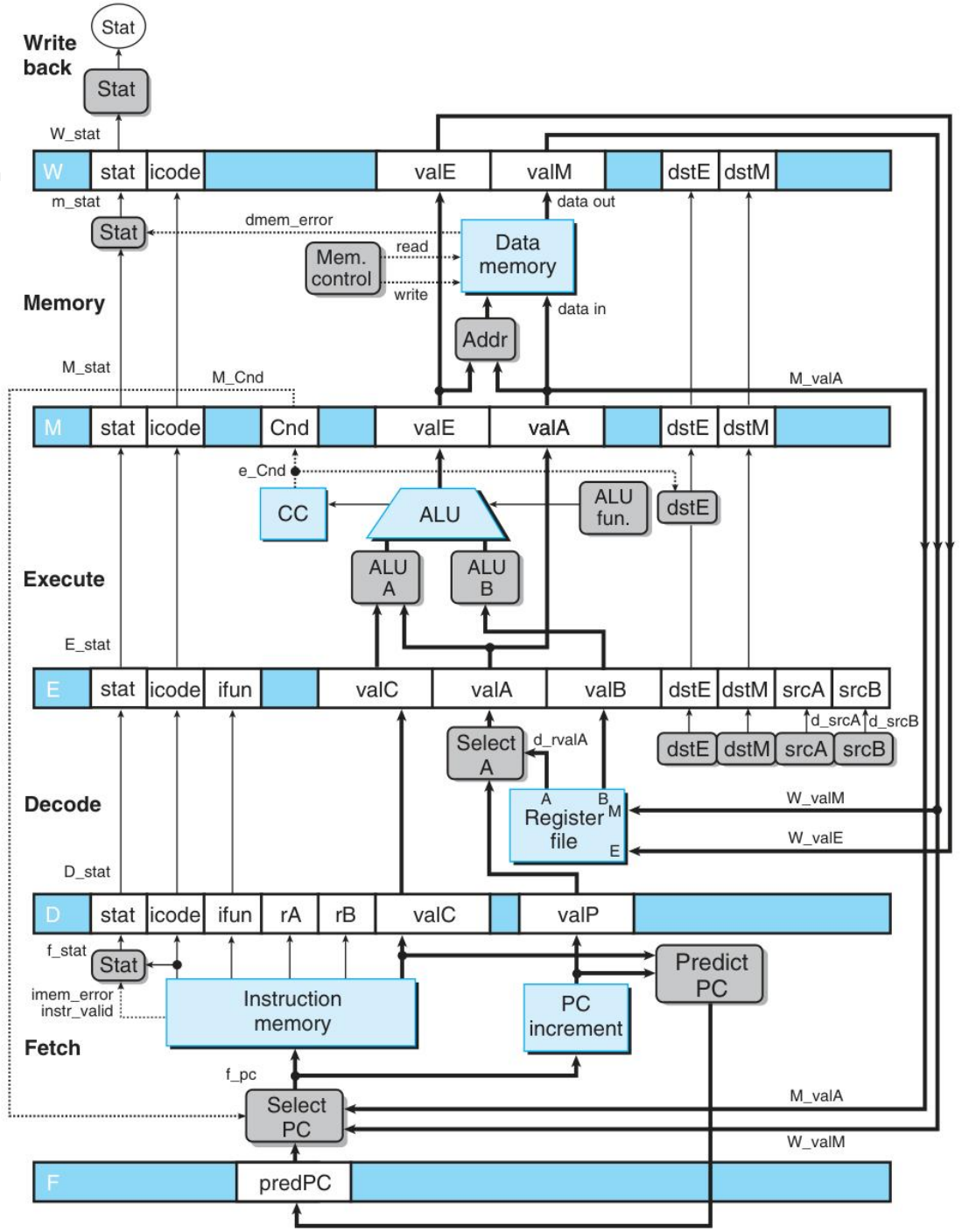
Handling control hazard

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



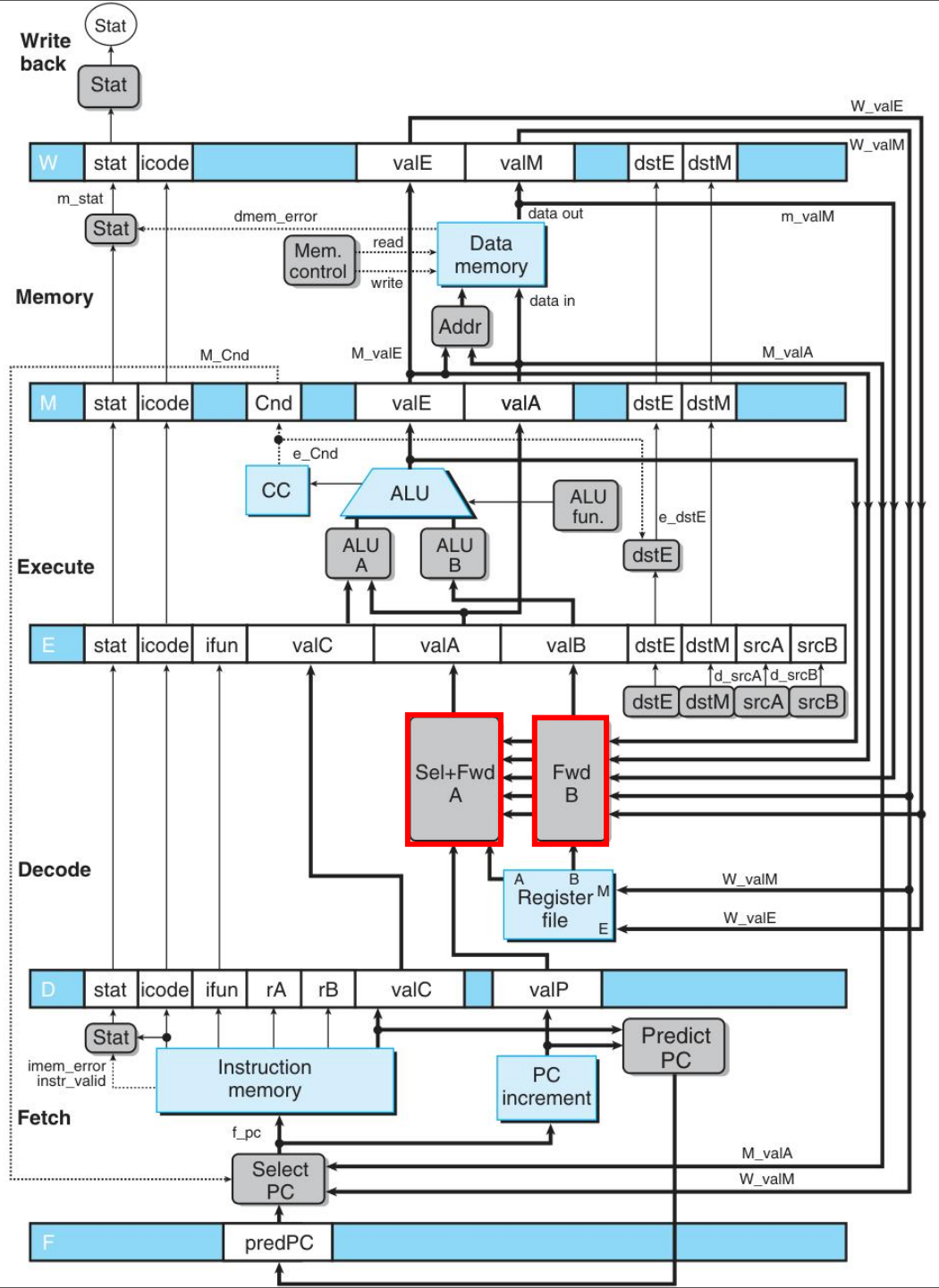
Another Solution: Forward

- Need the data that has not written back to the registers when decoding.
- **Principle: Try to use forward. If failed, use stall.**
- Sel+Fwd A
- Fwd B

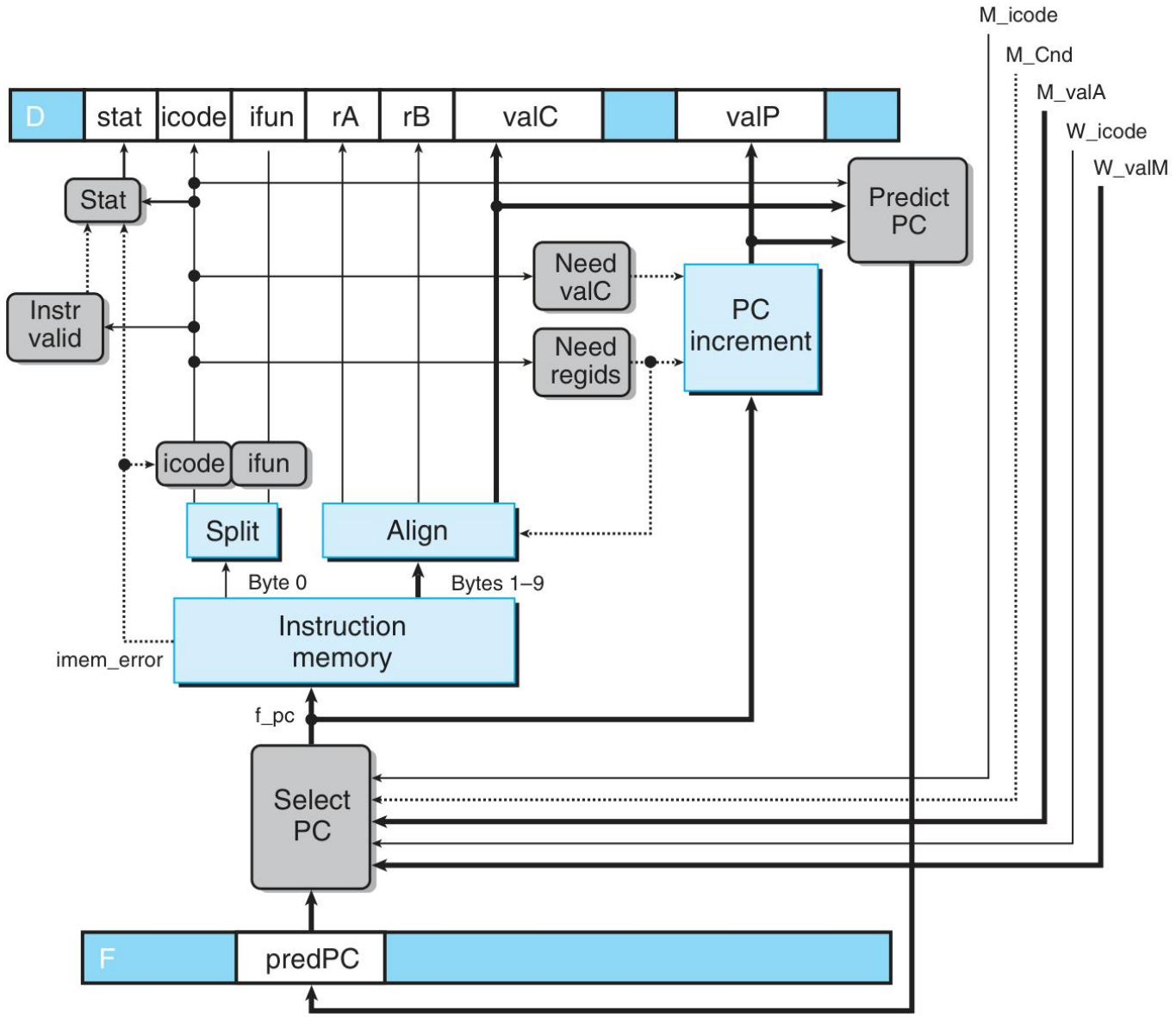


Another Solution: Forward

- Need the data that has not written back to the registers when decoding.
- **Principle: Try to use forward. If failed, use stall.**
- Sel+Fwd A
- Fwd B



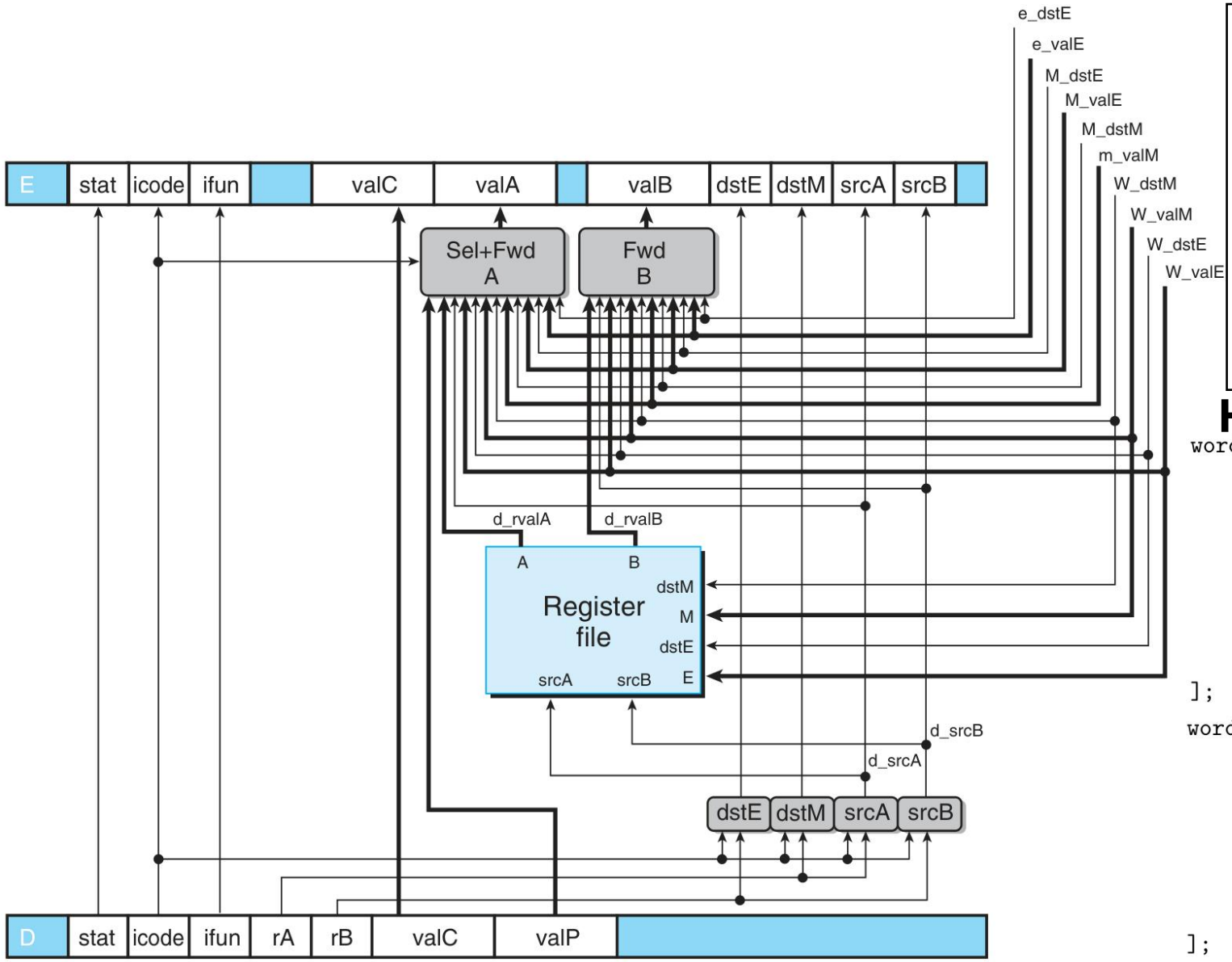
Modified HCL: Select PC & Fetch



HCL:

```
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];

word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

Modified HCL: Decode & Write back



Pay attention to the order!

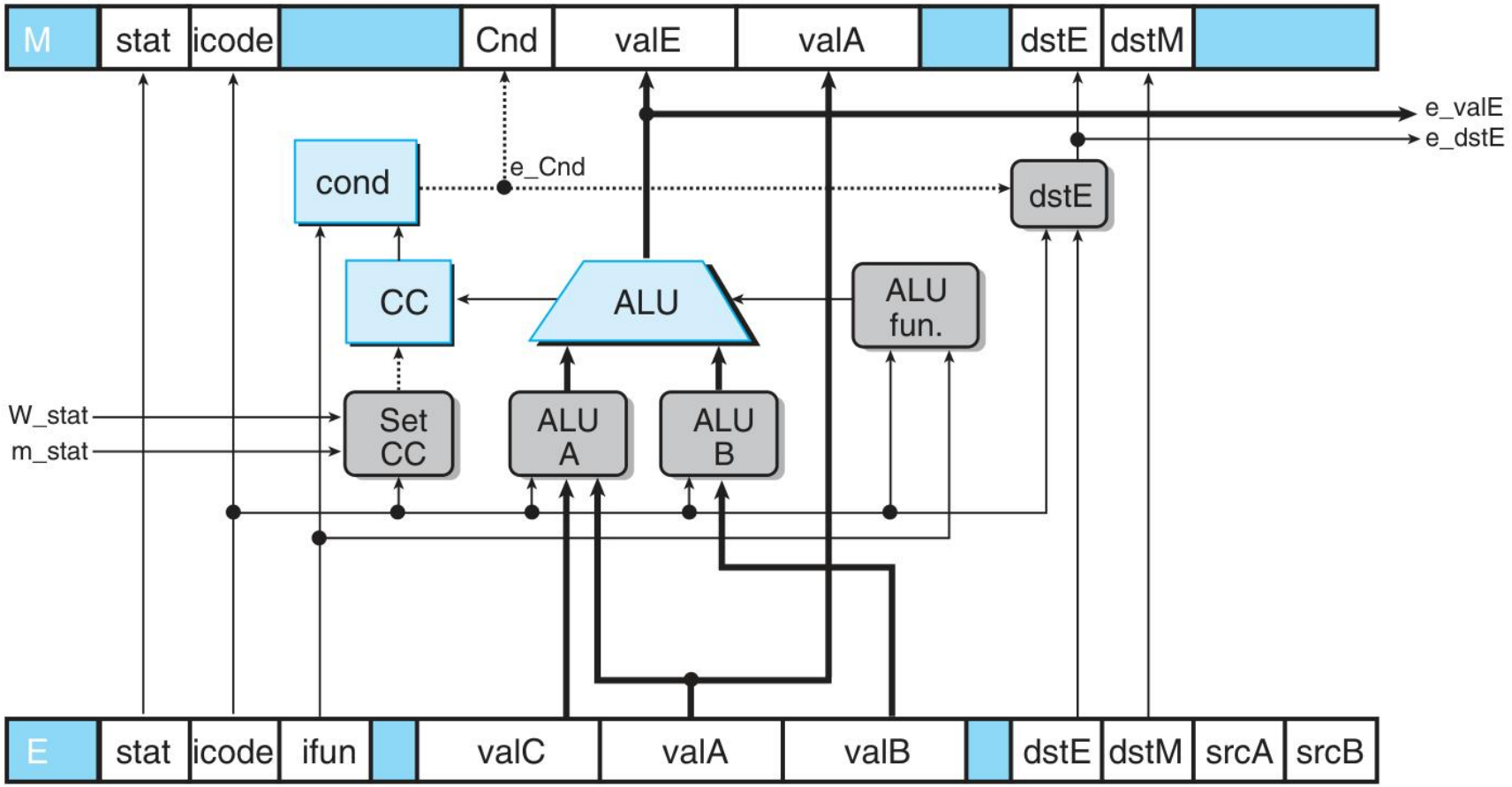
思考：不同转发源的先后顺序可以更改吗?

- 不能!
- 更靠后的指令的结果被优先采用，所以优先级 E阶段 > M阶段 > W阶段
- 为了让 popq %rsp 行为正确，所以优先级 valM > valE

HCL

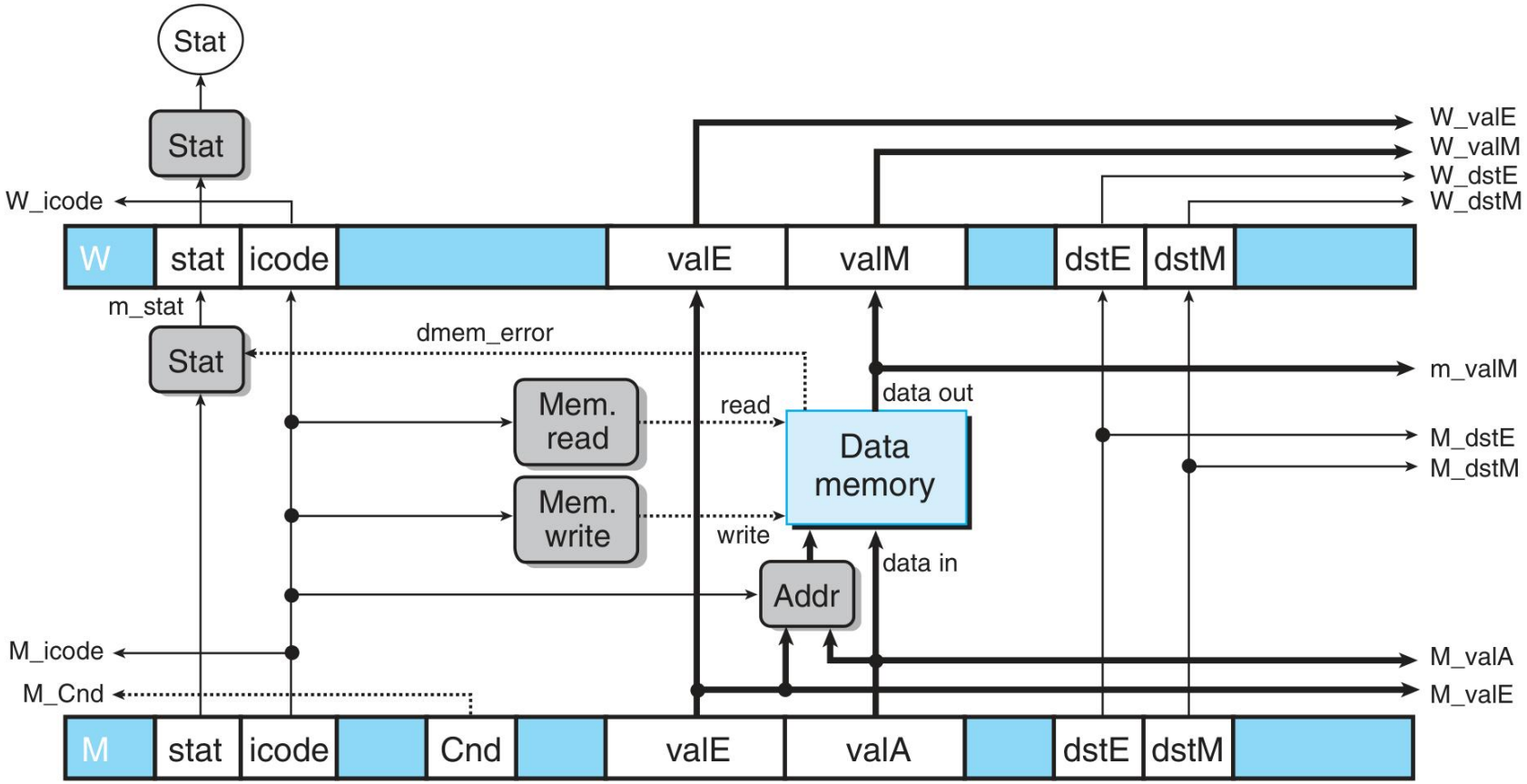
```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : 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
];
```

Modified HCL: Execute



HCL: left out

Modified HCL: Memory

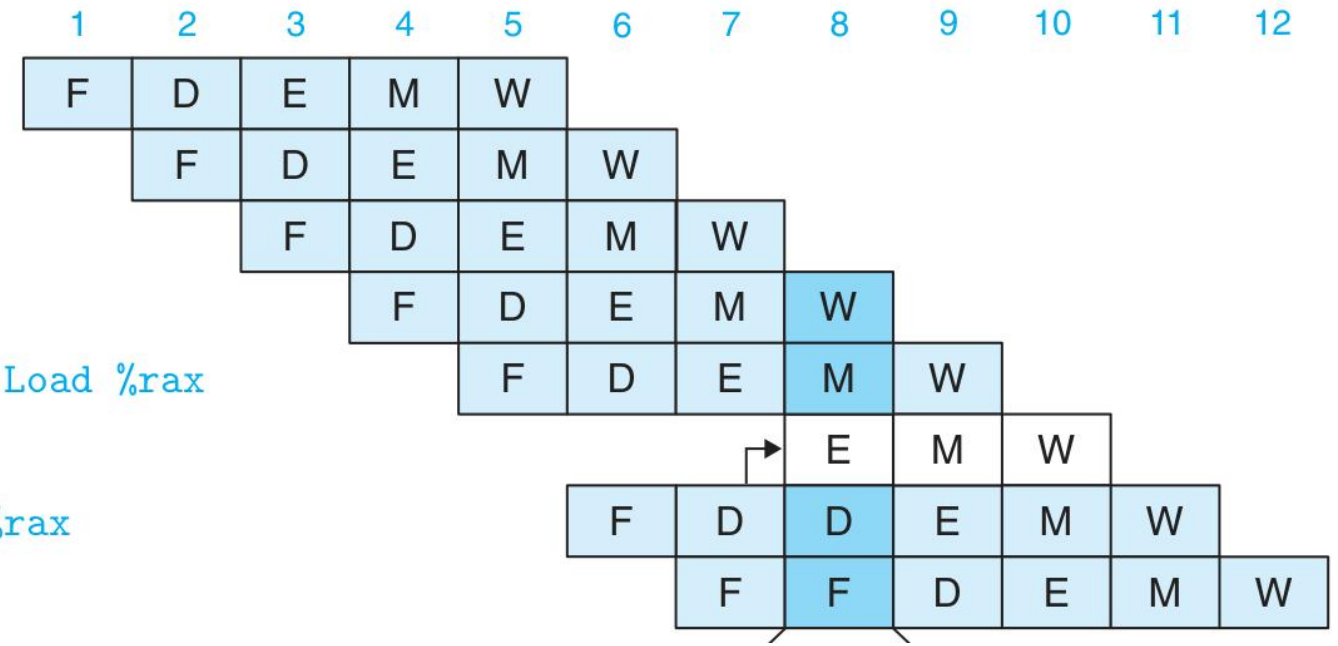


HCL: left out

Hazard: Load/Use

- **You cannot just use forwarding to solve all the problems…**
- Last instruction reads data from memory to a register, and present instruction needs the data in this register.
- Must stall and insert a bubble, then forward from memory stage.

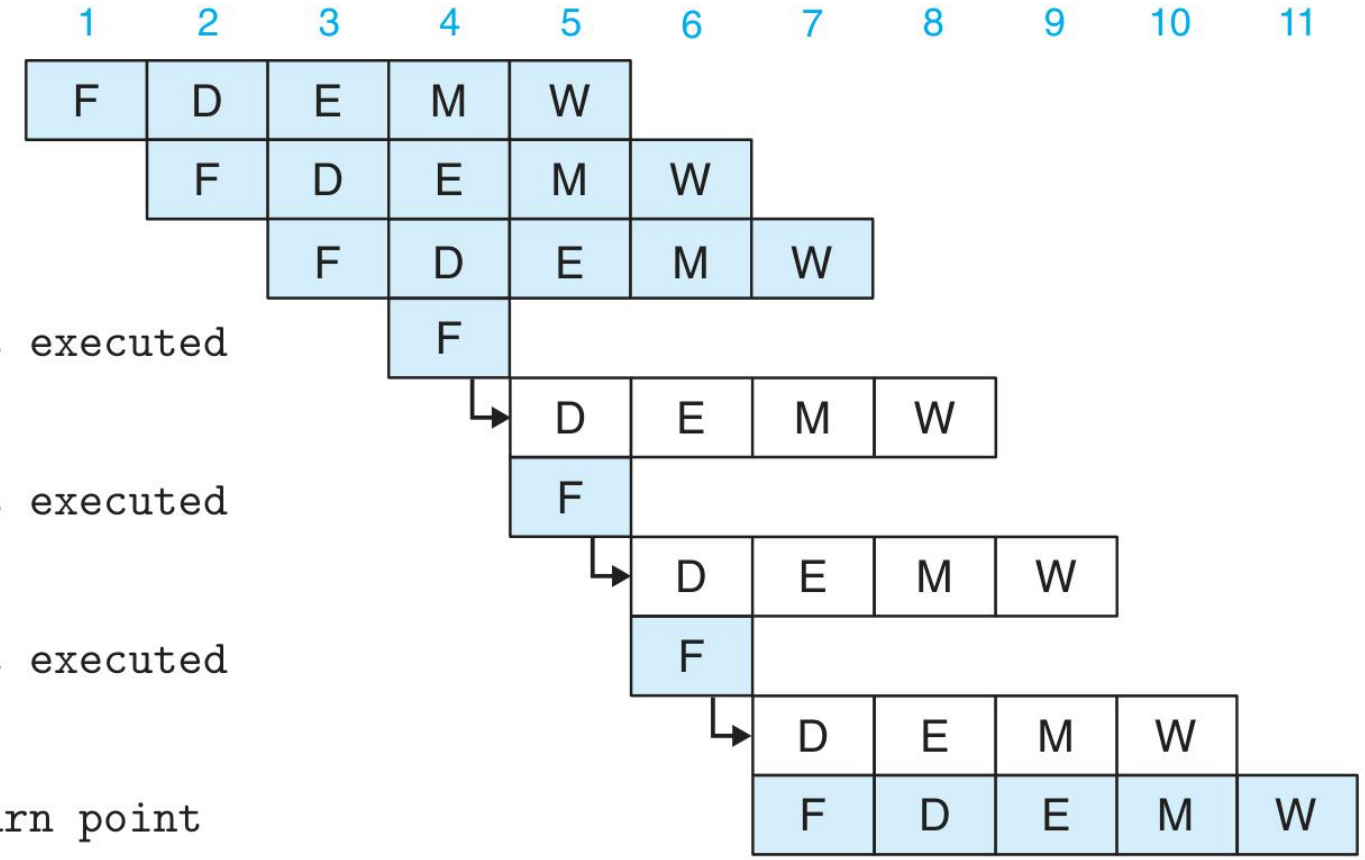
```
# prog5
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
     bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```



Hazard: ret

- The PC of the next instruction of ret will be known until memory stage.
- Insert three bubbles.

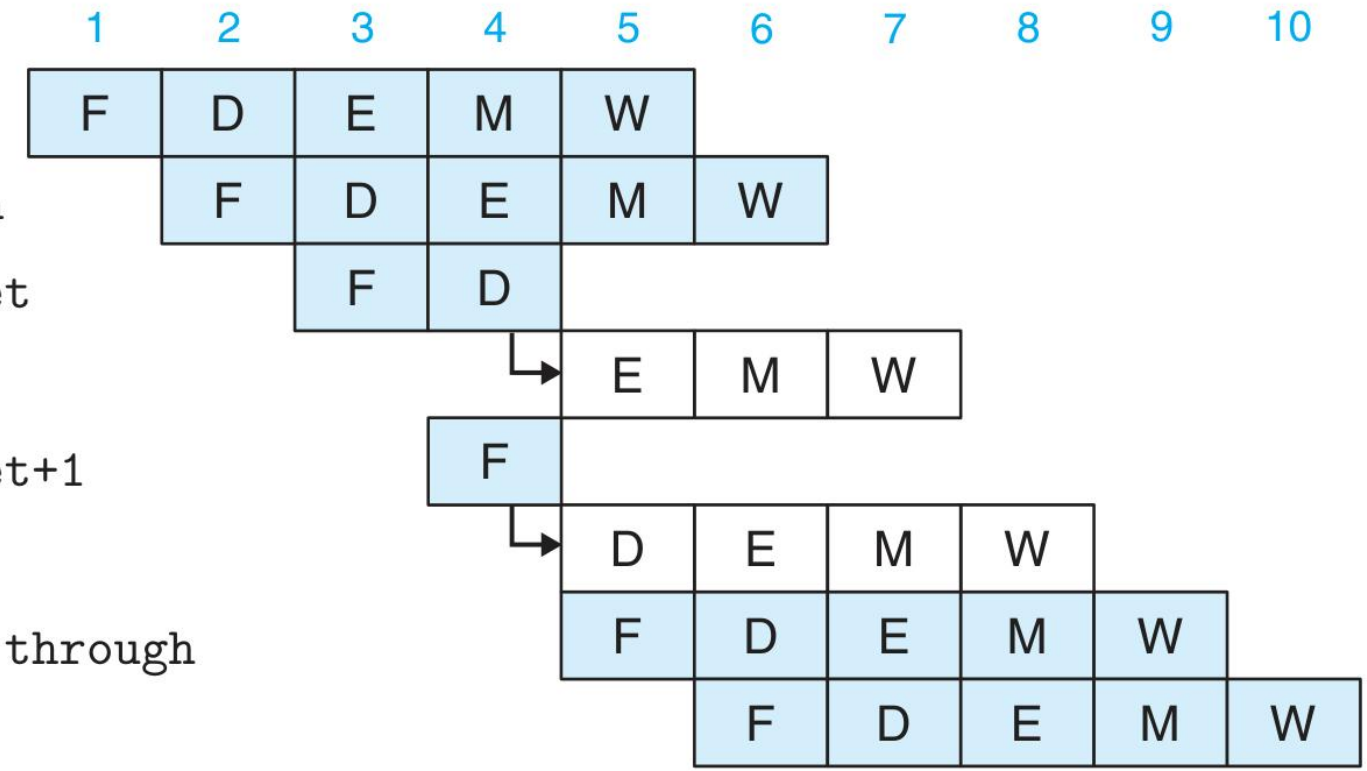
```
# prog6
0x000: irmovq Stack,%rsp
0x00a: call proc
0x020: ret
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x013: irmovq $10,%rdx # Return point
```



Hazard: Branch Misprediction

- After Execute stage, the right branch will be known.
- Insert two bubbles.

```
# 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
```



Questions

(13)　　判断下列说法的正确性

(1)（ F ）流水线的深度越深，总吞吐率越大，因此流水线应当越深越好。

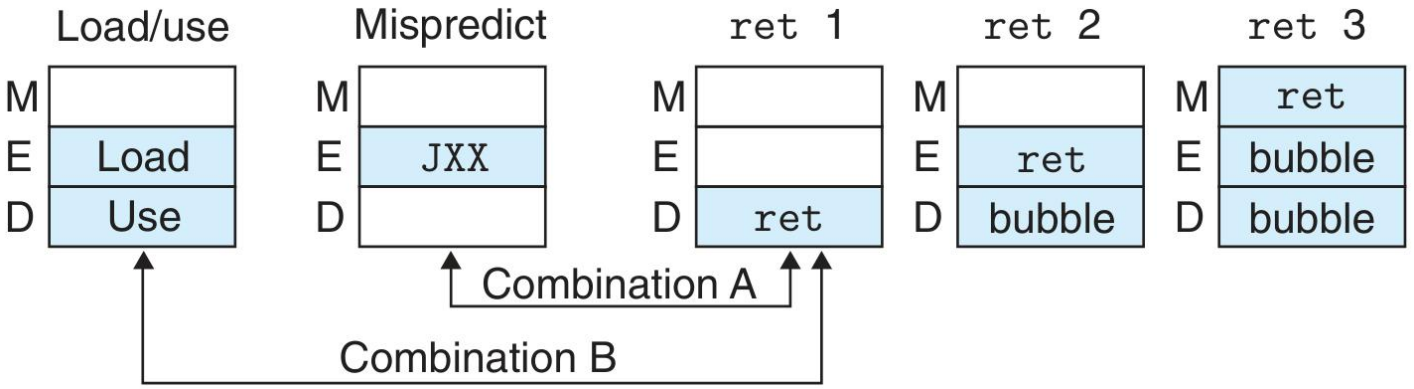
(2)（ T ）流水线的吞吐率取决于最慢的流水级，因此流水线的划分应当尽量均匀。

(3)（ T ）假设寄存器延迟为 20ps，那么总吞吐率不可能达到或超过 50 GIPS。

(4)（ F ）数据冒险总是可以只通过转发来解决。

(5)（ T ）数据冒险总是可以只通过暂停流水线来解决。

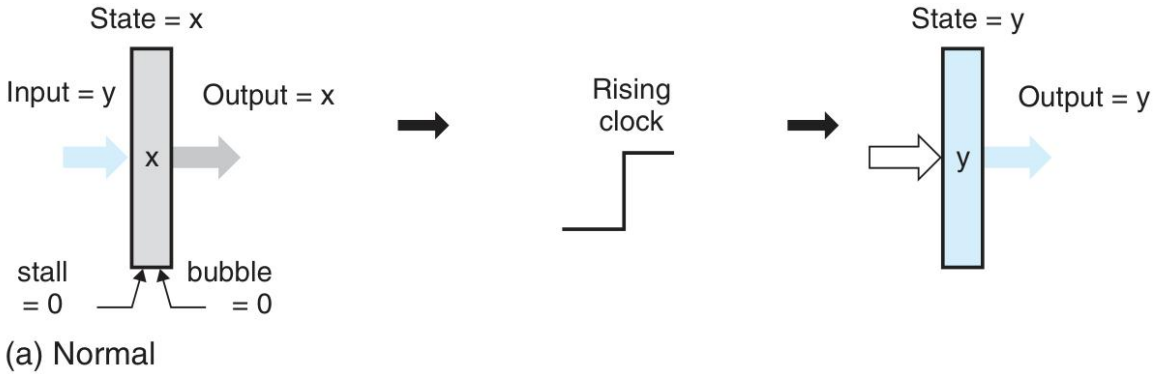
Hazard Combination



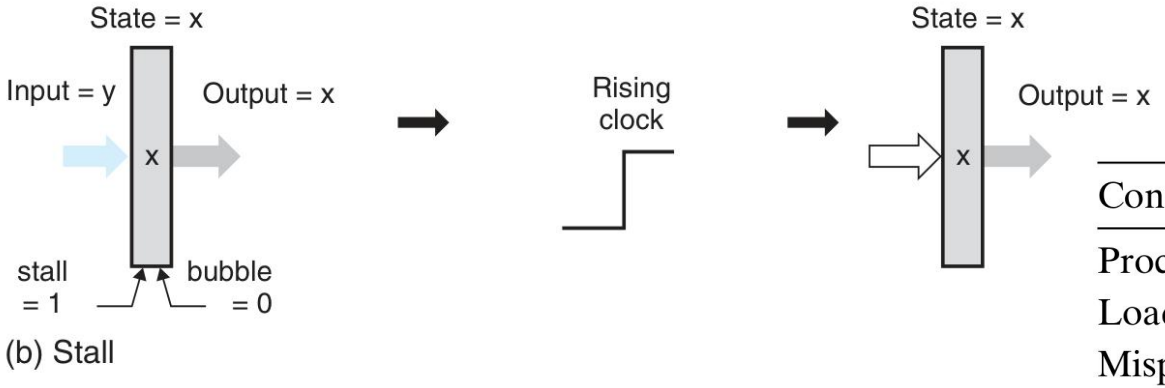
Condition	Pipeline register F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

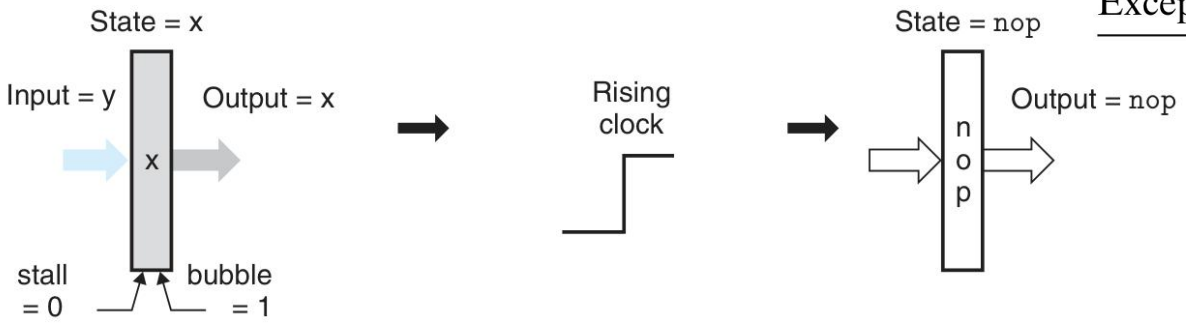
Hazard Detection & Control



(a) Normal



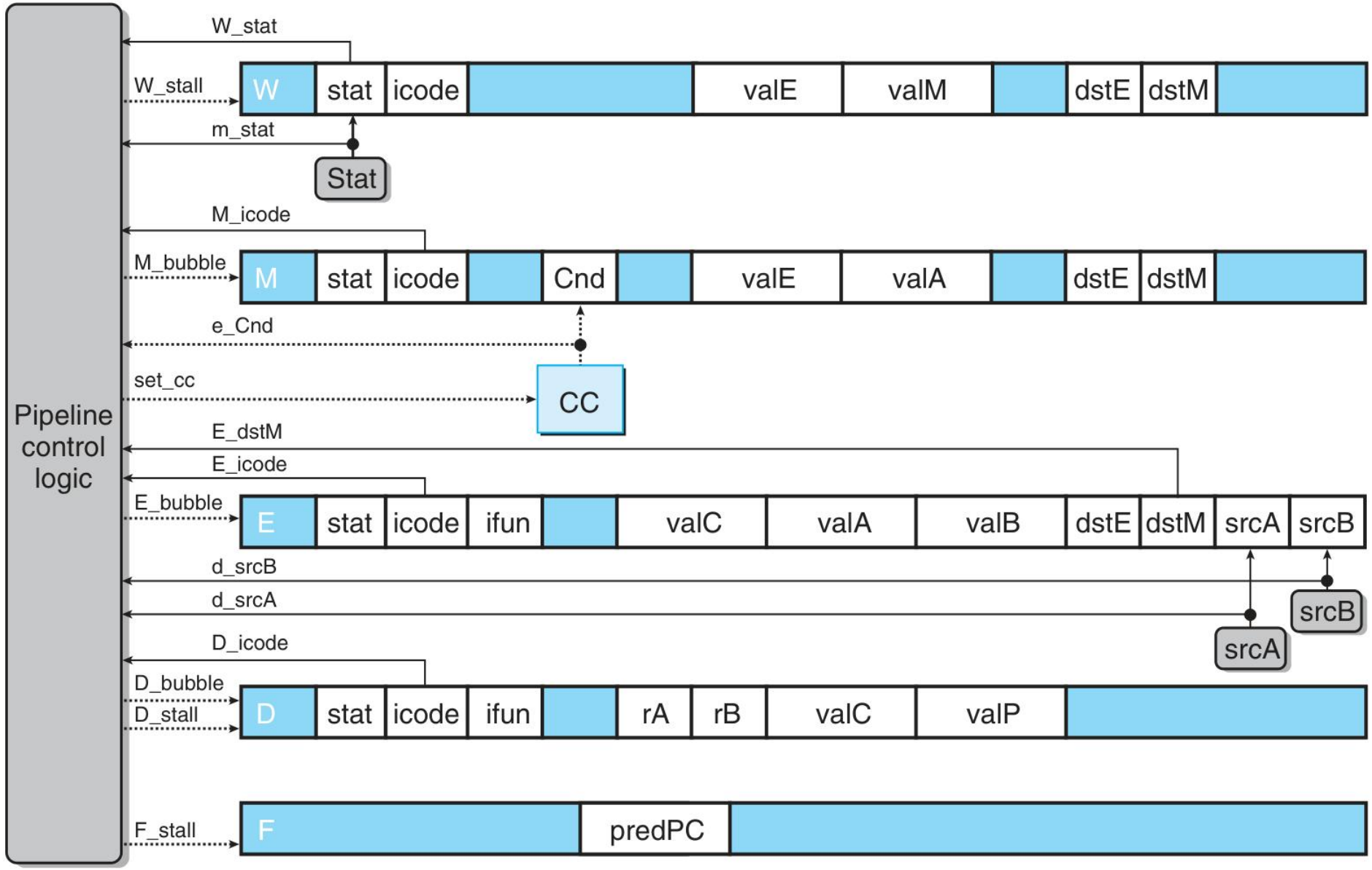
(b) Stall



(c) Bubble

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}

Implementing Pipeline Control



深入理解 Y86-64！

Q：pushq %rsp 压入栈里的值是原来的 %rsp ，还是 -8 以后的 %rsp？这在指令的分阶段描述上是怎么反映的？

A：原来的 %rsp。这与 pushq 的行为描述是一致的

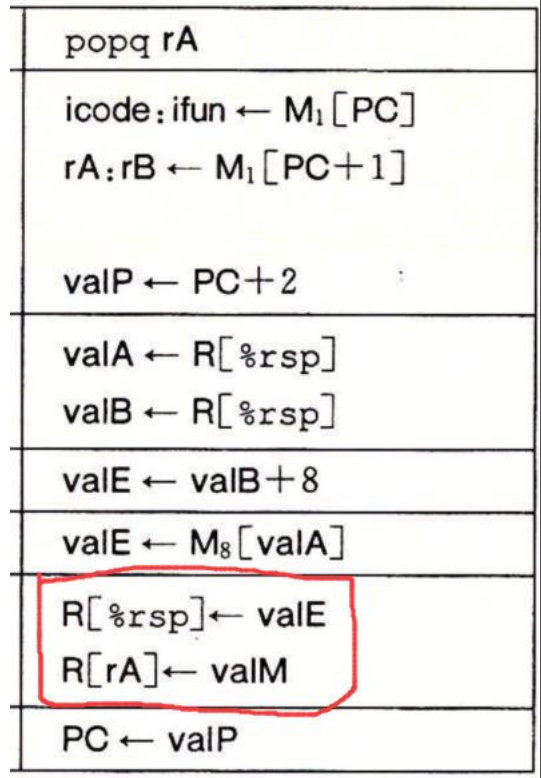
阶段	pushq rA
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
译码	valA ← R[rA] valB ← R[%rsp]
执行	valE ← valB+(−8)
访存	M_8[valE]← valA
写回	R[%rsp]← valE
更新 PC	PC ← valP

深入理解 Y86-64！

Q：popq %rsp 的行为是什么？这个行为在 popq 的分阶段实现上怎么描述？在硬件实现上有哪些设计是为了适应这个行为而做的？

A：

- popq %rsp 的行为是：把**从内存中读出的数**写入 %rsp，而不是把 （%rsp+8）写入 %rsp
- 在 popq 的分阶段实现上，表现为在 W 阶段，先写 R[dstE]←valE，再写 R[dstM]←valM，这样的话，按顺序一条一条执行下来，最后 %rsp 里的值就是 valM
- 在硬件实现上，
  - 在设计电路时，需要让写寄存器时，写 M 端口的优先级比写 E 端口的优先级更高
  - 在设计转发逻辑时，需要让来自同一阶段的转发源中优先级 valM＞valE



深入理解 Y86-64！

Q：在 SEQ 的取址阶段，一次性会从内存中读出几个字节？

A：固定地会读 10 个字节！先全部读进来然后再做指令的解析。

事实上，一次性读连续的 10 个字节和读 1 个字节的时间消耗其实差不多

深入理解 Y86-64！

```
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

Q：在 SelectPC 中，为什么使用 M_valA, M_cnd 和 W_valM，而非 e_valA, e_cnd 和 m_valM 呢？

A：（这是助教的个人理解）

- **好处：**控制冒险（包括ret和预测错误的jxx）所产生的bubble都可以少一个，因而可以少浪费一个时钟周期。
- **坏处：**m_valM 和 e_cnd 分别是需要等待 M阶段 和 E阶段执行完毕之后得到的结果。如果用这两个信号来更新pc，那么时钟周期就至少需要 F+E 或 F+M 的时间总和，太久了。时钟周期变大也会导致吞吐量下降。
- **正确性方面：**正如在“好处”中所说的，如果采用这个方案，只要相应地修改一下控制冒险的应对方案（就是减少一个气泡），正确性也是没问题的。

- **总结：**所以这其实是一个 **“浪费的周期数量（bubble个数）”与“时钟周期大小”的**trade-off的问题。在这个例子中，F阶段本身就是需要访存取址的一个阶段（访存往往可能带来很大的延迟），如果再加上E/M阶段的延迟（尤其是M阶段，因为也需要访存），可能会让时钟周期变得超大，所以没有采用这种方案。

深入理解 Y86-64！

A：（续上问）

- **总结：**所以这其实是一个 **“浪费的周期数量（bubble个数）”与“时钟周期大小”的**trade-off的问题。在这个例子中，F阶段本身就是需要访存取址的一个阶段（访存往往可能带来很大的延迟），如果再加上E/M阶段的延迟（尤其是M阶段，因为也需要访存），可能会让时钟周期变得超大，所以没有采用这种方案。

- **进一步思考：**为什么在做“转发”的时候，转发源之中又有 e_valE 和 m_valM 了呢？我个人觉得这是因为，D阶段是一个很轻量级的阶段，哪怕需要看E和M的计算结果来完成最后valA和valB的计算，其实也就只是在E/M的基础上又经过了一个组合逻辑而已，不会增加太多的时间开销。这时候使用e_valE和m_valM可以减少一个bubble，带来的收益就更大了。

深入理解 Y86-64！

Q：流水线控制逻辑表格里面，从正确性的角度上来看，ret 的 F 能不能改成”正常“？ 预测错误的 F 能不能改成”暂停“？

A：我认为，从正确性的角度上看，都是可以的！
不过同学们在背流水线控制逻辑的时候还是按照书上背，这个问题只是为了增进理解。

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

图 4-66 流水线控制逻辑的动作。不同的条件需要改变流水线流，或者会暂停流水线，或者会取消部分已执行的指令

深入理解 Y86-64！

Q：在时钟上升沿，流水线寄存器（等时钟控制的东西）会接收输入。但是，流水线寄存器从接收到输入，到输出变化，也需要经过一个微小的时延。如果在流水线寄存器的输出变化之前，后续的组合逻辑就已经读取了流水线寄存器的输出，会不会出问题？

A：不会！
如果发生这种情况，因为组合逻辑是实时响应的，所以在写入完成之后，组合逻辑会响应新的输入。只要时钟周期足够长，保障所有的组合逻辑能够在一个周期内达到稳态，就不会出问题

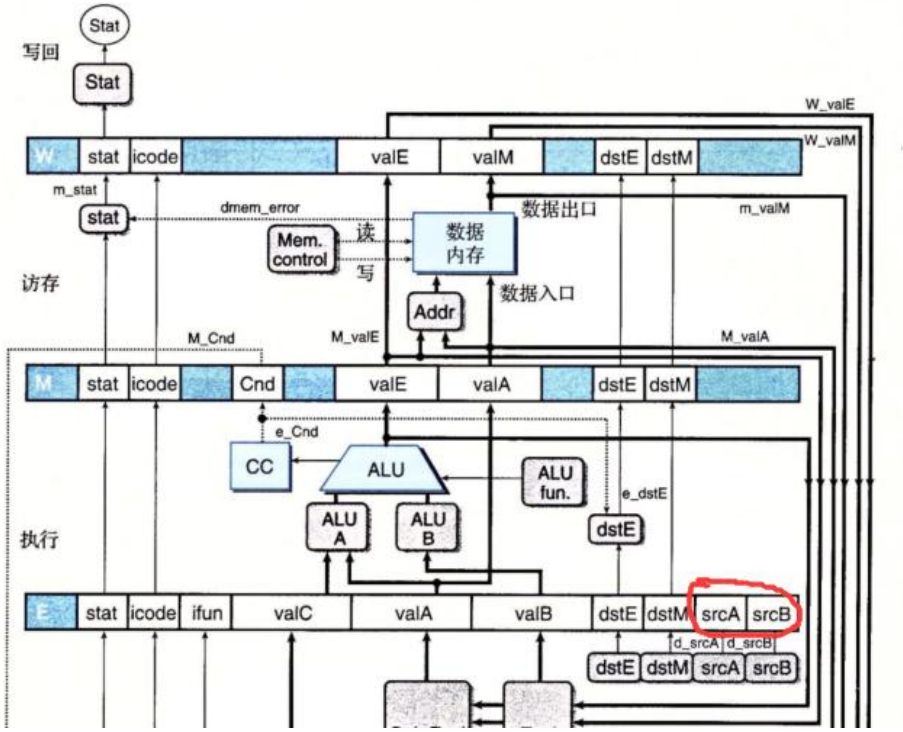
深入理解 Y86-64！

Q：为什么 PIPE 的 E阶段时钟寄存器还有 srcA 和 srcB这两个字段？好像根本没用到过？

A：

在我们的 PIPE 里面确实没有用到，不过在课后练习题 4.57 里用到了！

课后练习题 4.57 介绍了“加载/转发”机制，建议同学们有空的话学习一下，有可能可以用在 Archlab 中哦



THANK YOU

加油！