

Processor Arch

— — Sequential & Pipelined

吴思衡 王效乐 许珈铭

2023.10.23

Preface

许珈铭

Processor designer's view

- Sequential
 - design a processor that **can work** with **less hardware**
- Pipelined
 - run the processor **efficiently** with **large throughput**

Sequential (CS:APP Ch. 4.3)

吴思衡

Six Stages

- 取指(fetch):以PC值为地址从内存读取指令字节。**(必须)**
- 译码(decode):从rA和rB指明的寄存器读入操作数，也有读%rsp的。
- 执行(execute):要么根据ifun计算、要么增加或减少栈指针。同时对一条跳转指令来说，这个阶段会决定是不是应该选择分支。**(必须)**
- 访存(memory):将数据写入内存或者从内存读出数据。
- 写回(write back):最多可将两个结果写回寄存器文件。
- 更新PC(PC update):将PC设置成下一条指令的地址。**(必须)**

Y86-64 Instruction Set

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmoveXX 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						

Attention:

1. fn可以替换成不同的数，表示不同的指令
2. 有的指令只需要一个寄存器，剩余的用0xF代替
3. 指令长度取决于第一个byte

M[D]表示访问内存, R[r]表示寄存器的值
CC:ZF、SF、OF

	$OPq\ rA, rB$	$rrmovq\ rA, rB$	$irmovq\ V, rB$
取指	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_8[PC + 2]$ $valP \leftarrow PC + 10$
译码	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	$valA \leftarrow R[rA]$	
执行	$valE \leftarrow valB OP valA$ Set CC	$valE \leftarrow 0 + valA$	$valE \leftarrow 0 + valC$
访存			
写回	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$
更新PC	$PC \leftarrow valP$	$PC \leftarrow valP$	$PC \leftarrow valP$

内存读写指令

阶段	<code>rmovq rA, D(rB)</code>	<code>rmovq D(rB), rA</code>
取指	$\text{icode, ifun} \leftarrow M_1[PC]$ $rA, rB \leftarrow M_1[PC+1]$ $\text{valC} \leftarrow M_8[PC+2]$ $\text{valP} \leftarrow PC + 10$	$\text{icode, ifun} \leftarrow M_1[PC]$ $rA, rB \leftarrow M_1[PC+1]$ $\text{valC} \leftarrow M_8[PC+2]$ $\text{valP} \leftarrow PC + 10$
译码	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$	$\text{valB} \leftarrow R[rB]$
执行	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB} + \text{valC}$
访存	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valE}]$
写回		
		$R[rA] \leftarrow \text{valM}$
更新 PC	$PC \leftarrow \text{valP}$	$PC \leftarrow \text{valP}$

控制转移指令

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

入栈出栈操作

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

取指(fetch)

<code>OPq rA, rB</code>	<code>rrmovq rA, rB</code>	<code>irmovq V, rB</code>	<code>rmmovq rA, D(rB)</code>	<code>mrmovq D(rB), rA</code>
<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>
<code>rA:rB $\leftarrow M_1[PC + 1]$</code>	<code>rA:rB $\leftarrow M_1[PC + 1]$</code>	<code>rA:rB $\leftarrow M_1[PC + 1]$</code>	<code>rA:rB $\leftarrow M_1[PC + 1]$</code>	<code>rA:rB $\leftarrow M_1[PC + 1]$</code>
<code>valP $\leftarrow PC + 2$</code>	<code>valP $\leftarrow PC + 2$</code>	<code>valC $\leftarrow M_8[PC + 2]$</code> <code>valP $\leftarrow PC + 10$</code>	<code>valC $\leftarrow M_8[PC + 2]$</code> <code>valP $\leftarrow PC + 10$</code>	<code>valC $\leftarrow M_8[PC + 2]$</code> <code>valP $\leftarrow PC + 10$</code>
<code>pushq rA</code>	<code>popq rA</code>	<code>jXX Dest</code>	<code>call Dest</code>	<code>ret</code>
<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>	<code>icode:ifun $\leftarrow M_1[PC]$</code>
<code>rA:rB $\leftarrow M_1[PC + 1]$</code>	<code>rA:rB $\leftarrow M_1[PC + 1]$</code>	<code>valC $\leftarrow M_8[PC + 1]$</code>	<code>valC $\leftarrow M_8[PC + 1]$</code>	
<code>valP $\leftarrow PC + 2$</code>	<code>valP $\leftarrow PC + 2$</code>	<code>valP $\leftarrow PC + 9$</code>	<code>valP $\leftarrow PC + 9$</code>	<code>valP $\leftarrow PC + 1$</code>

译码(decode)、执行(execute)

OPq rA, rB	rrmovq rA, rB	irmovq V, rB	rmmovq rA, D(rB)	mrmovq D(rB), rA
$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$ $\text{valE} \leftarrow \text{valB OP valA}$ Set CC	$\text{valA} \leftarrow R[rA]$ $\text{valE} \leftarrow 0 + \text{valA}$	Decode	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$ $\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valB} \leftarrow R[rB]$ $\text{valE} \leftarrow \text{valB} + \text{valC}$
pushq rA	popq rA	jXX Dest	call Dest	ret
$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valA} \leftarrow R[%rsp]$ $\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + 8$	Cnd $\leftarrow \text{Cond(CC, ifun)}$	$\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valA} \leftarrow R[%rsp]$ $\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + 8$

valE \leftarrow valB + valC

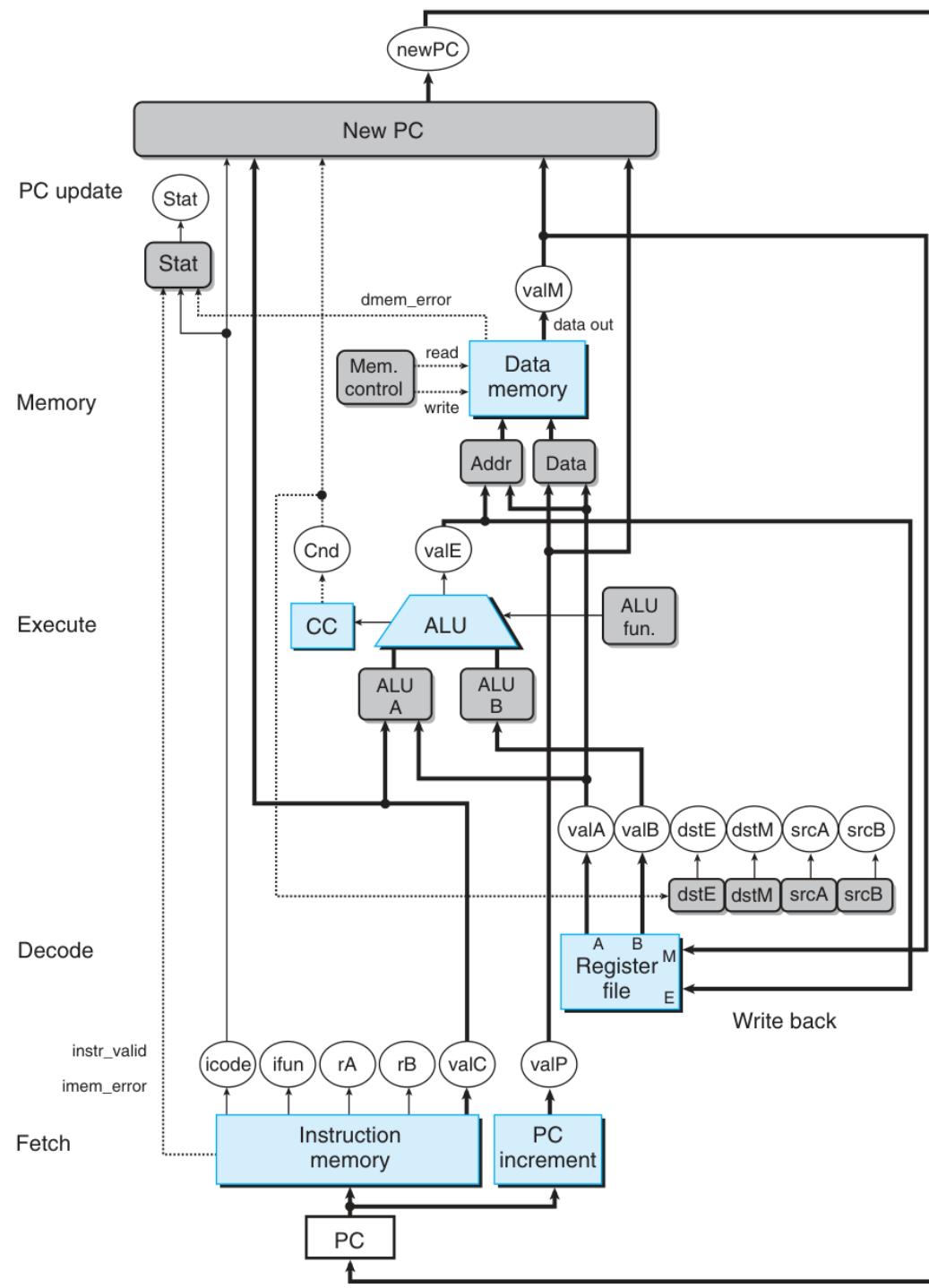
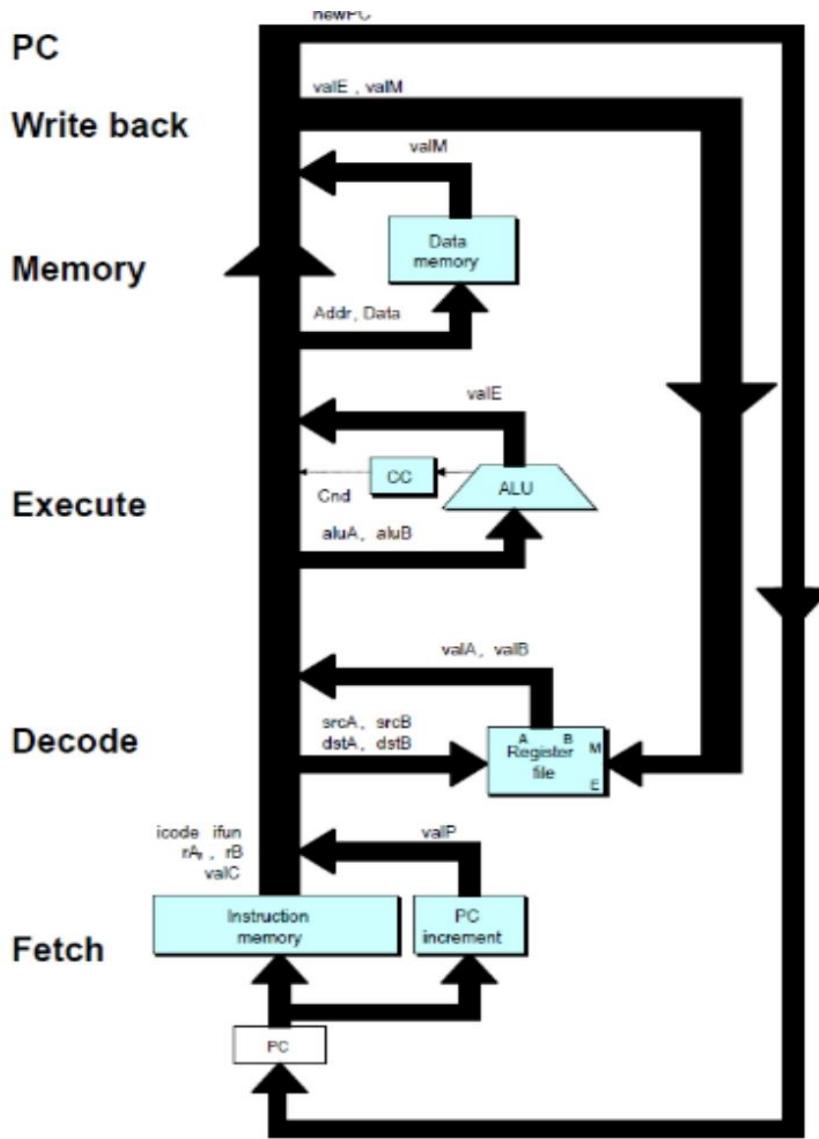


计算地址

访存(memory)、写回(write back)

<code>OPq rA, rB</code>	<code>rrmovq rA, rB</code>	<code>irmovq V, rB</code>	<code>rmmovq rA, D(rB)</code>	<code>mrmovq D(rB), rA</code>
$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$M_8[valE] \leftarrow valA$	$valM \leftarrow M_8[valE]$
<code>pushq rA</code>	<code>popq rA</code>	<code>jXX Dest</code>	<code>call Dest</code>	<code>ret</code>
$M_8[valE] \leftarrow valA$	$valM \leftarrow M_8[valA]$		$M_8[valE] \leftarrow valP$	$valM \leftarrow M_8[valA]$
$R[%rsp] \leftarrow valE$	$R[%rsp] \leftarrow valE$		$R[%rsp] \leftarrow valE$	$R[%rsp] \leftarrow valE$
	$R[rA] \leftarrow valM$			

HARDWARE

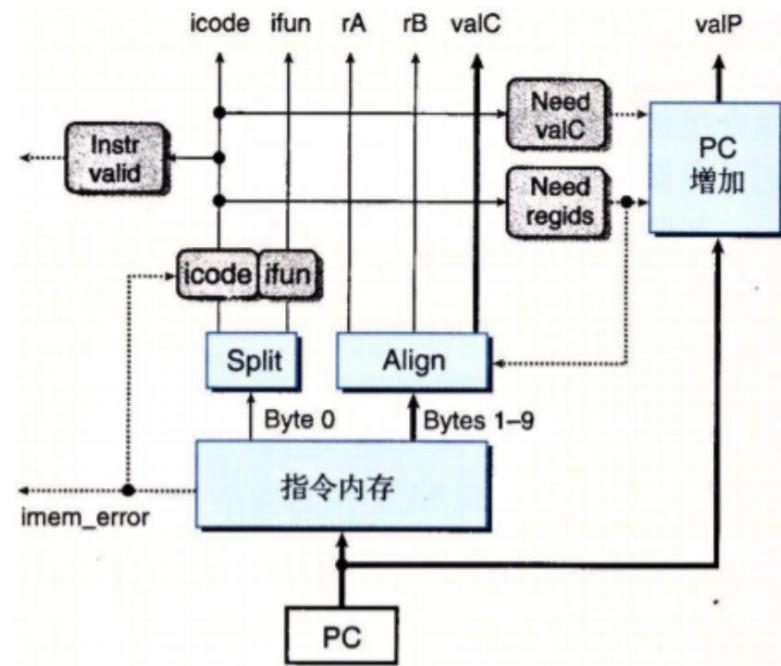


名称	值(十六进制)	含义
IHALT	0	halt 指令的代码
INOP	1	nop 指令的代码
IRRMQVQ	2	rrmqvq 指令的代码
IRMMQVQ	3	irmqvq 指令的代码
IRMMOVQ	4	rmovq 指令的代码
IMRMQVQ	5	mrmvq 指令的代码
IDPL	6	整数运算指令的代码
IJXX	7	跳转指令的代码
ICALL	8	call 指令的代码
IRET	9	ret 指令的代码
IPUSHQ	A	pushq 指令的代码
IPOPQ	B	popq 指令的代码
FNONE	C	默认功能码
RRSP	D	%rsp 的寄存器 ID
RNONE	E	表明没有寄存器文件访问
ALUADD	F	加法运算的功能
SACK	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

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPOQ

Fetch

```
bool need_regids =  
    icode in {  
        ...  
    };  
  
bool need_valC =  
    icode in {  
        ...  
    };  
  
# Determine instruction code  
int icode = [  
    ...  
];  
  
# Determine instruction function  
int ifun = [  
    ...  
];  
  
bool instr_valid = icode in  
{  
    ...  
};
```



)

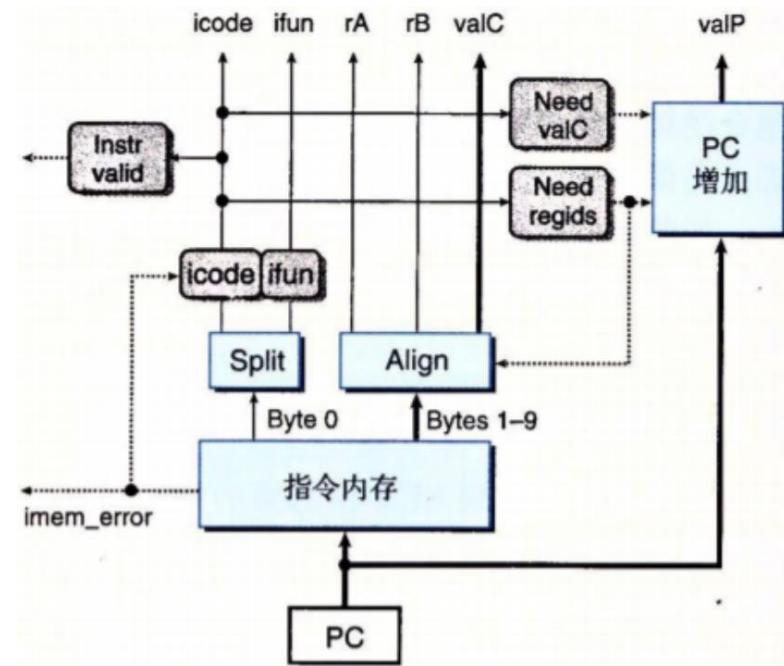
INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Fetch

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

```
bool need_valC =  
    icode in {  
};  
  
# Determine instruction code  
int icode = [  
];
```

```
        bool instr_valid = icode in  
        {  
};  
  
# Determine instruction function  
int ifun = [  
];
```



Fetch

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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

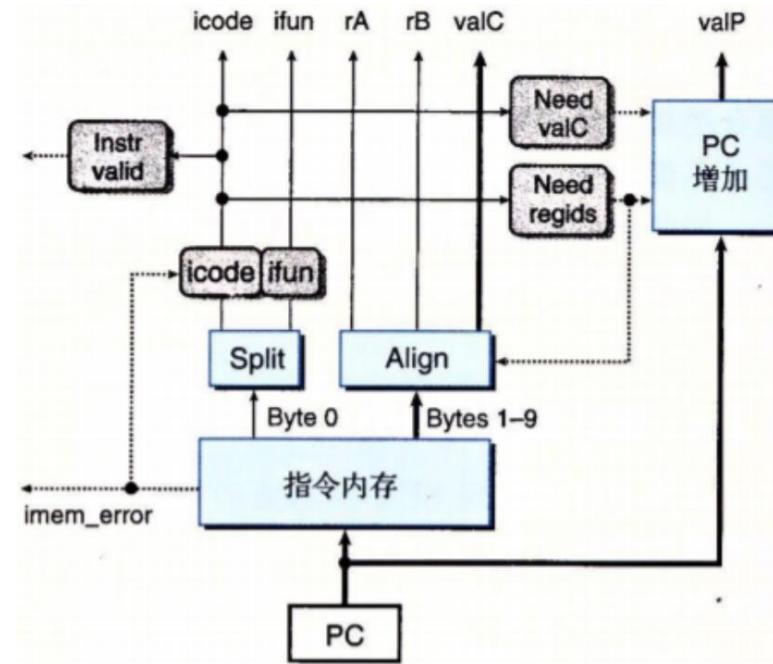
```
bool need_valC =
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };
```

```
# Determine instruction code
int icode = [
```

```
];
# Determine instruction function
int ifun = [
```

```
        bool instr_valid = icode in
        {
    };
```

```
];
```



Fetch

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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

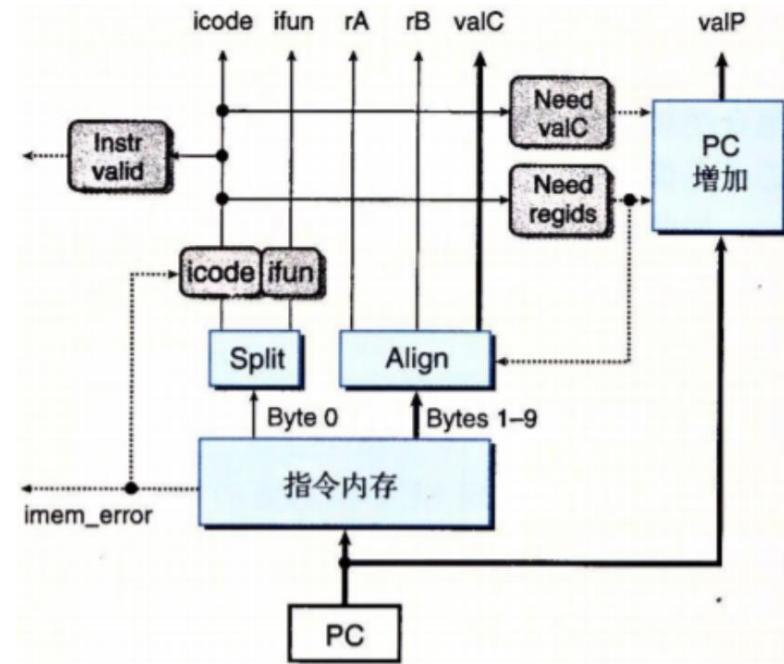
```
bool need_valC =
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };
```

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

```
# Determine instruction function
int ifun = [
```

```
    bool instr_valid = icode in
    {
```

```
};
```



```
];
```

Fetch

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

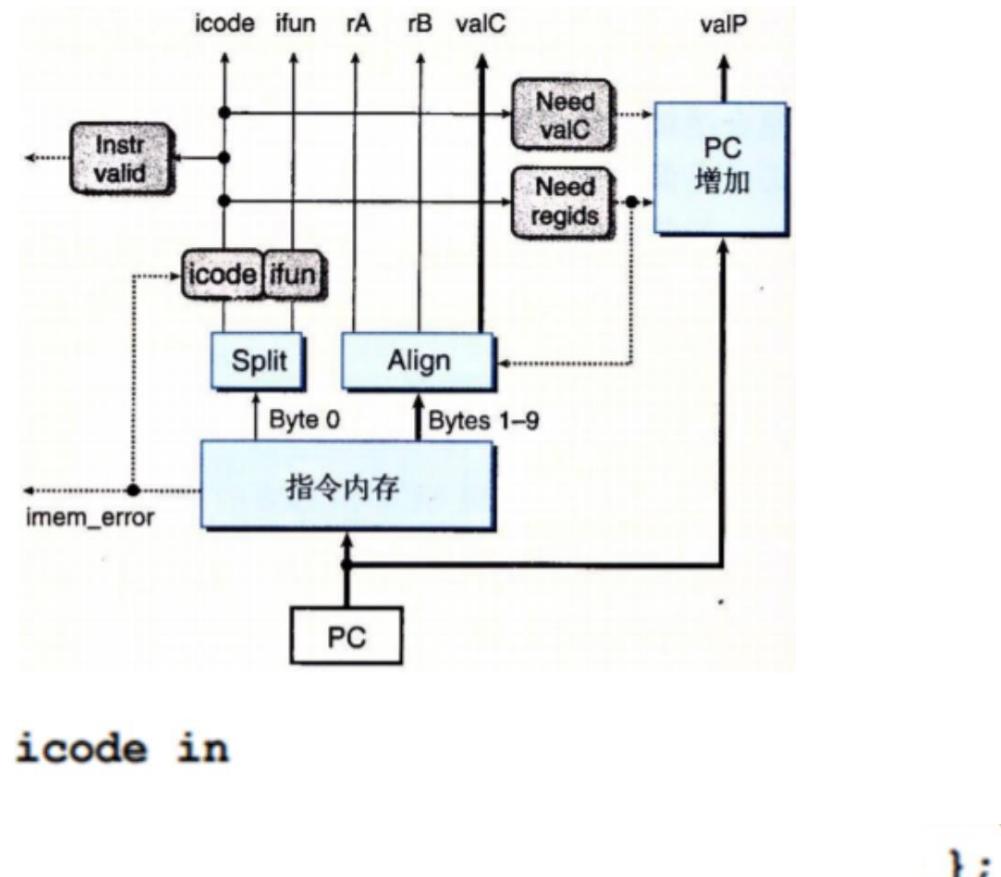
```
bool need_regsids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
    IIRMOVQ, IRMMOVQ, IMRMOVQ };

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
{
    imem_error: FNONE;
    1: imem_ifun;
};
```



Fetch

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

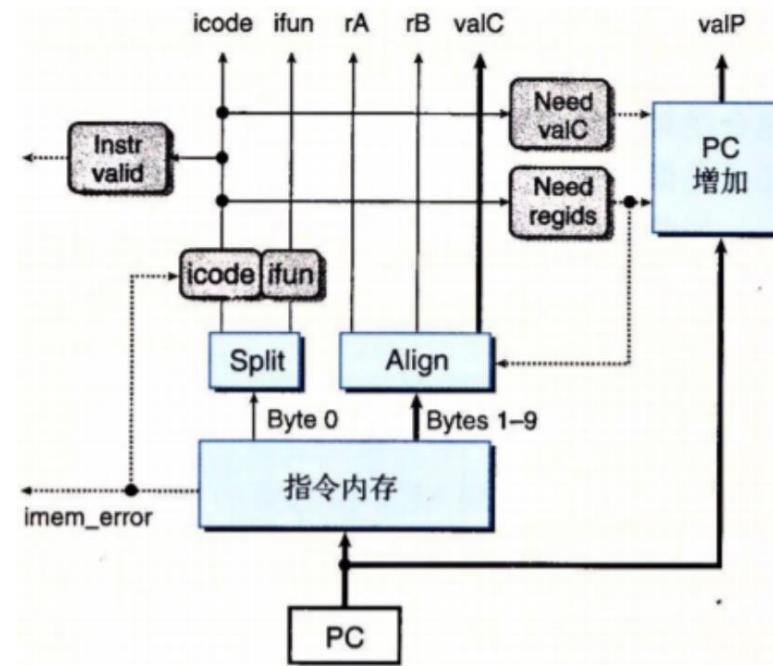
```
bool need_regsids =
    icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
    IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

```
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,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
```



译码(decode)、执行(execute)

OPq rA, rB	rrmovq rA, rB	irmovq V, rB	rmmovq rA, D(rB)	mrmovq D(rB), rA
$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$ $\text{valE} \leftarrow \text{valB OP valA}$ Set CC	$\text{valA} \leftarrow R[rA]$ $\text{valE} \leftarrow 0 + \text{valA}$	Decode	$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[rB]$ $\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valB} \leftarrow R[rB]$ $\text{valE} \leftarrow \text{valB} + \text{valC}$
pushq rA	popq rA	jXX Dest	call Dest	ret
$\text{valA} \leftarrow R[rA]$ $\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valA} \leftarrow R[%rsp]$ $\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + 8$	Cnd $\leftarrow \text{Cond(CC, ifun)}$	$\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valA} \leftarrow R[%rsp]$ $\text{valB} \leftarrow R[%rsp]$ $\text{valE} \leftarrow \text{valB} + 8$

valE \leftarrow valB + valC



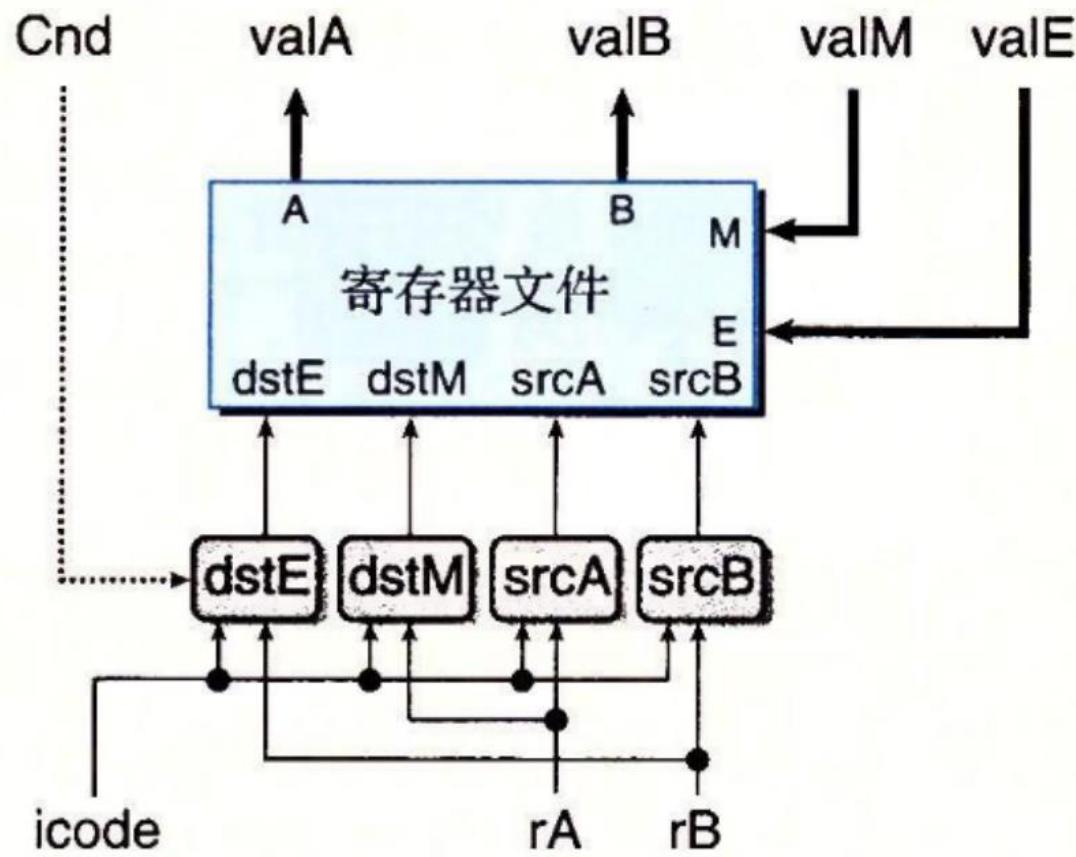
计算地址

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Decode

HCL:

```
word srcA = [
    icode in { } : rA; Cnd
    icode in { } :
    1 :
];
word srcB = [
];
word dstE = [
];
word dstM = [
];
];
```

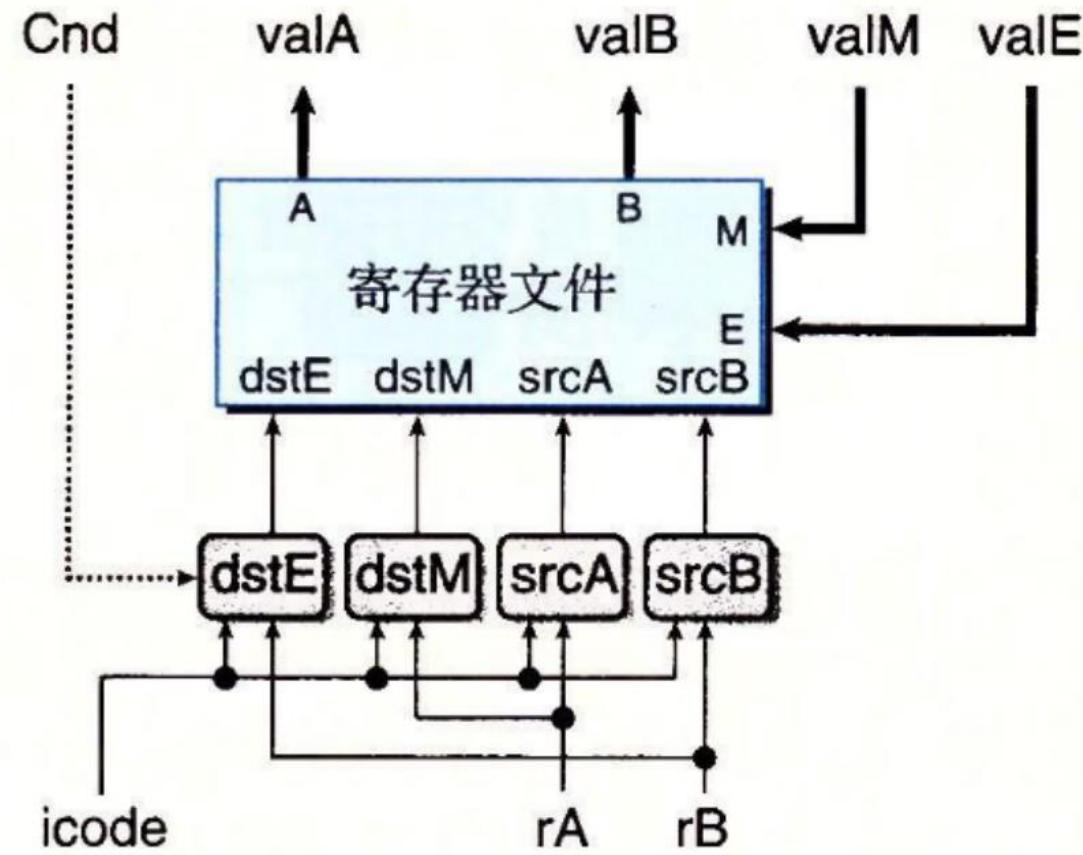


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Decode

HCL:

```
word srcA = [  
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;  
    icode in { } :  
    1 :  
];  
word srcB = [  
];  
word dstE = [  
];  
word dstM = [  
];
```

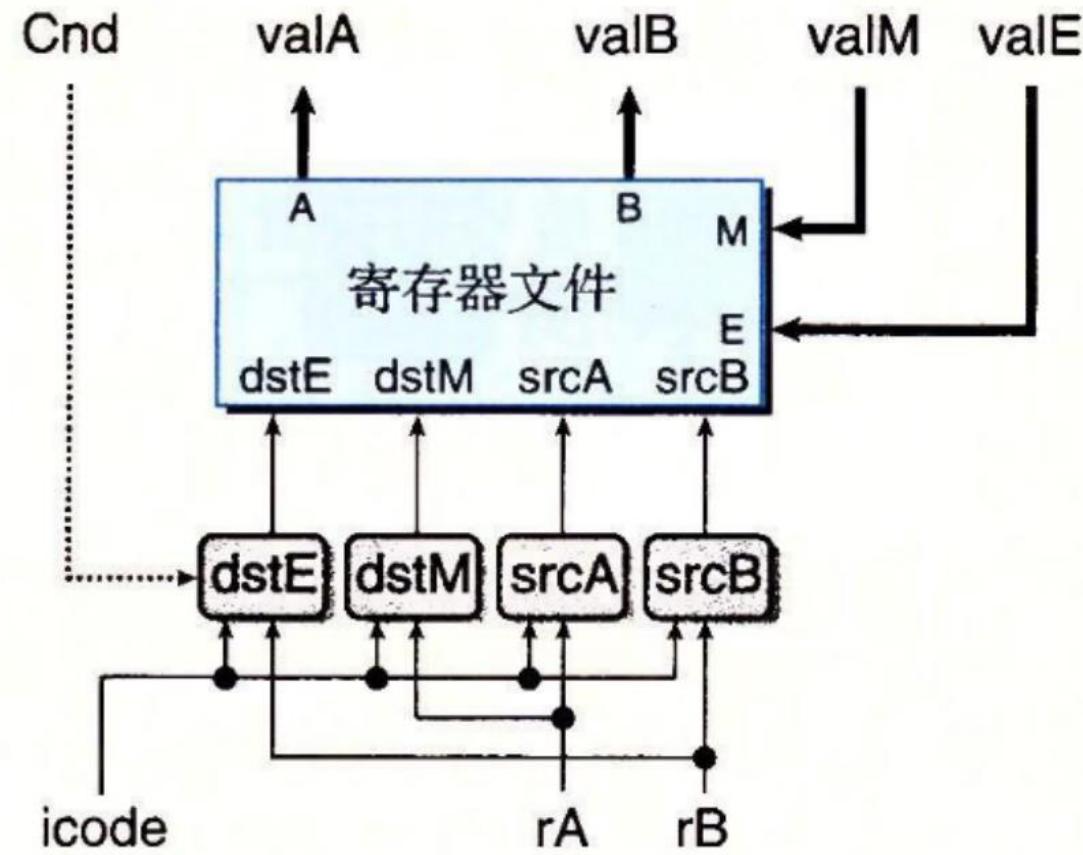


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Decode

HCL:

```
word srcA = [  
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;  
    icode in { IPOPQ, IRET } :  
    1 :  
];  
word srcB = [  
];  
word dstE = [  
];  
word dstM = [  
];
```

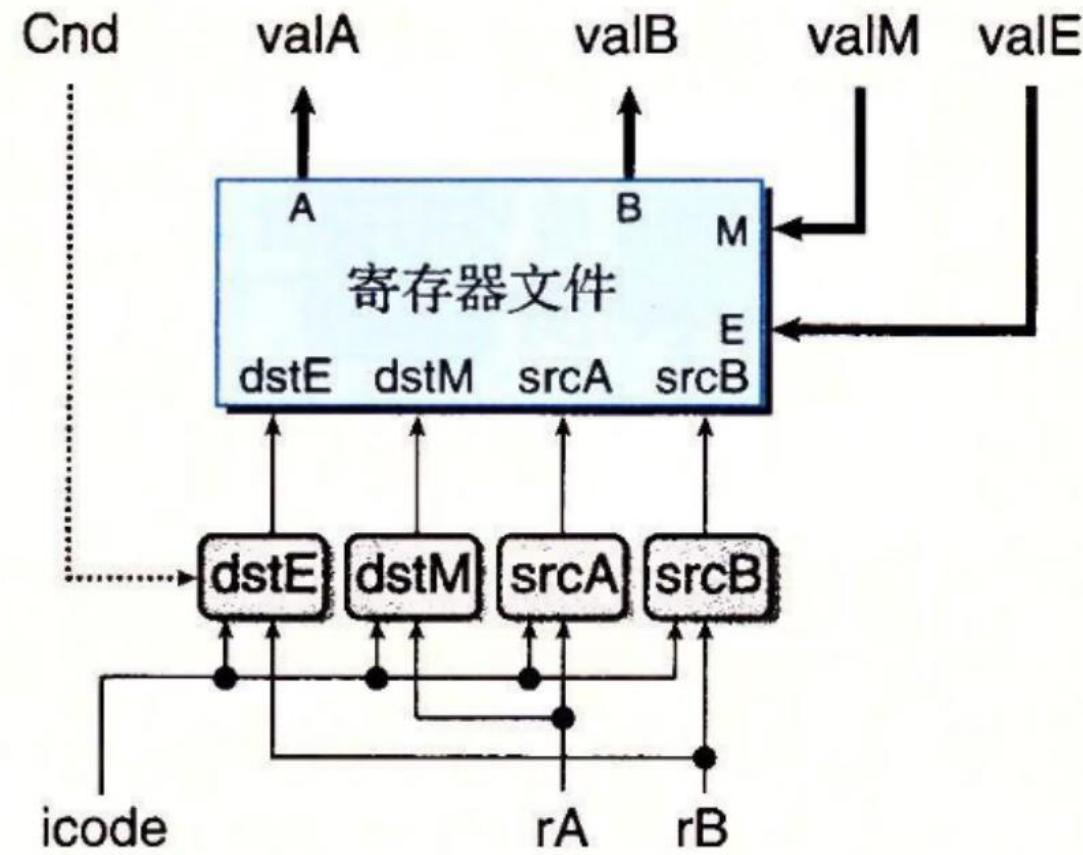


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Decode

HCL:

```
word srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPQ, IRET } : RRSP;
    1 :
];
word srcB = [
];
word dstE = [
];
word dstM = [
];
];
```

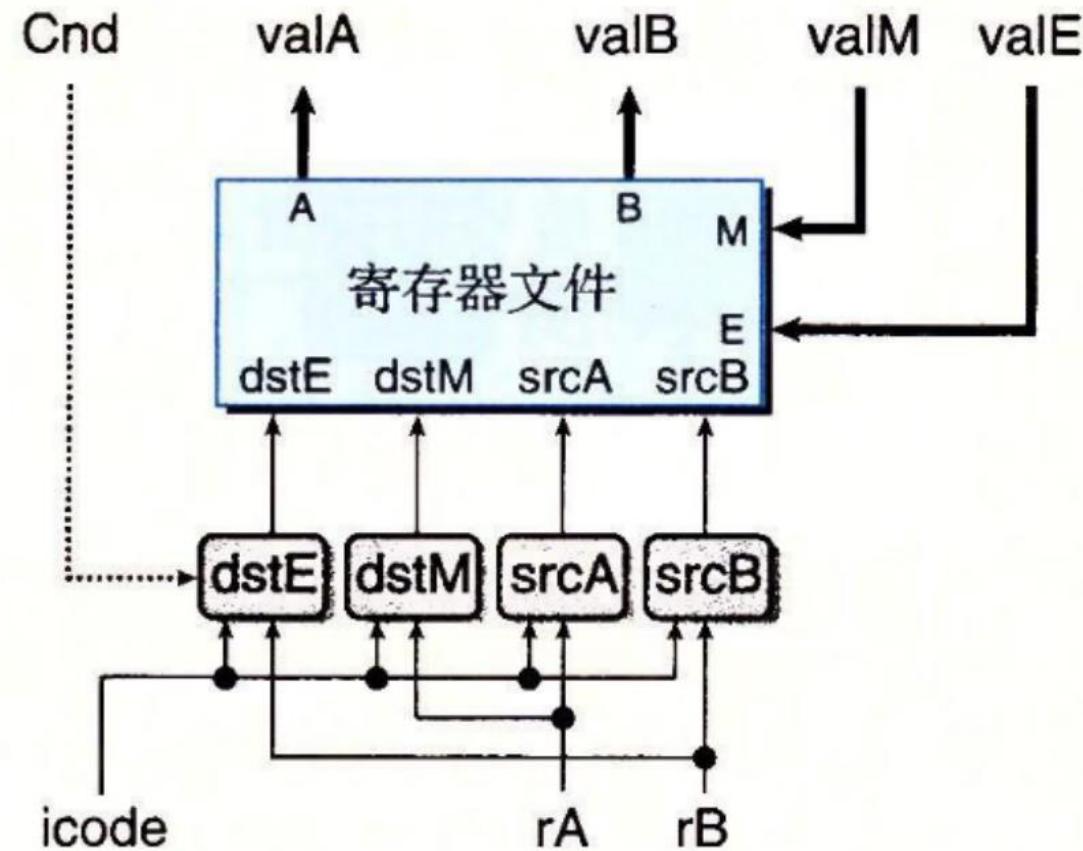


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 { } : rB;
    icode in { } :
    1 :
];
word dstE = [
];
word dstM = [
];
];
```

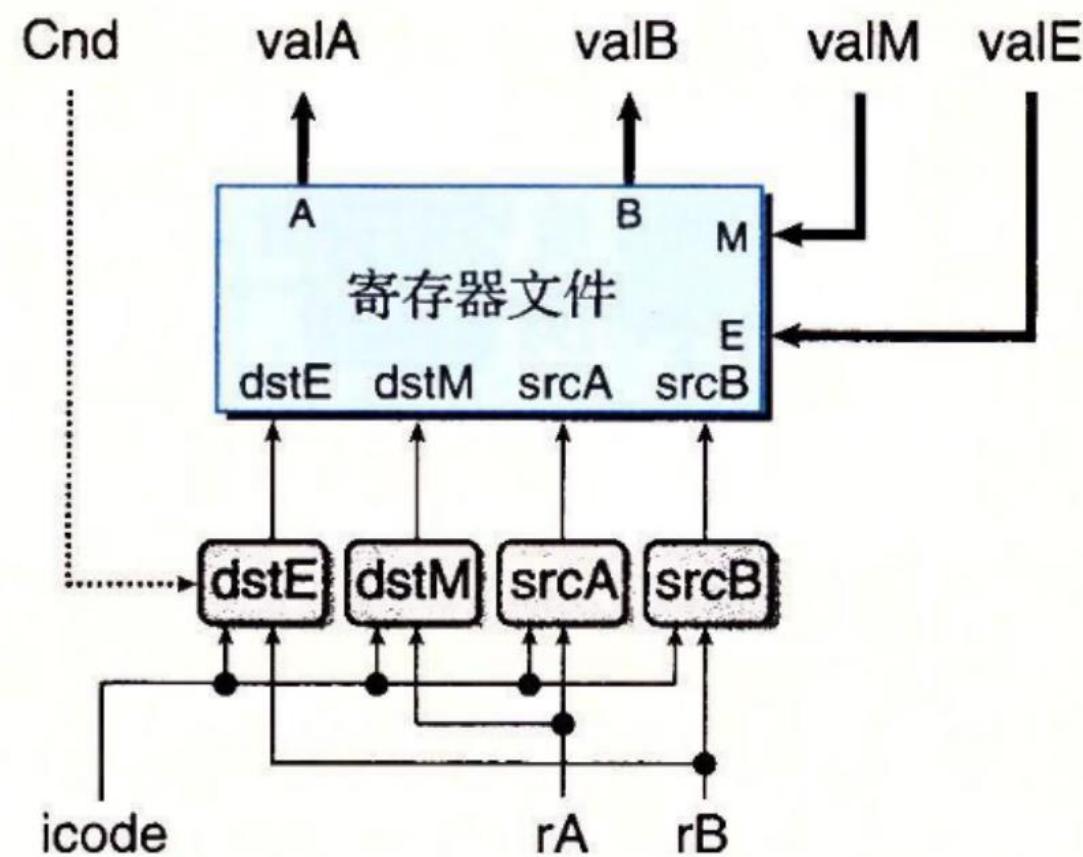


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 { } :  
    1 :  
];  
word dstE = [  
];  
word dstM = [  
];
```

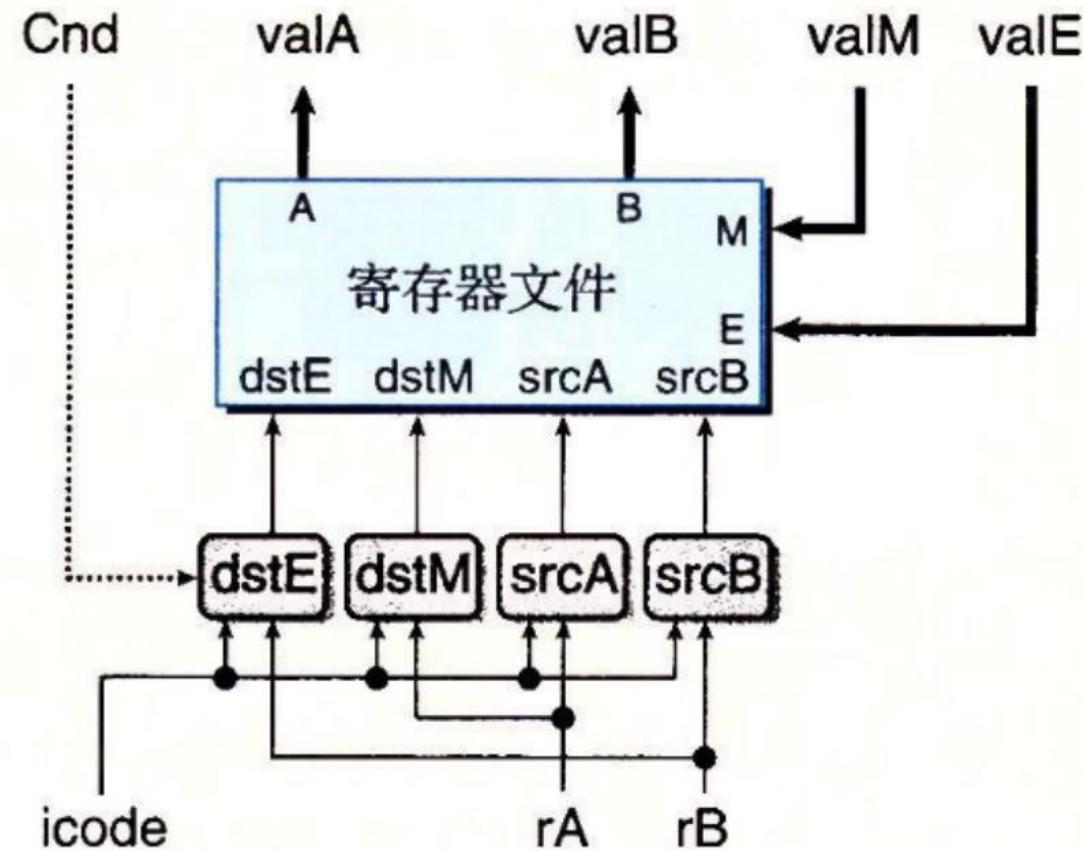


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 } :
    1 :
];
word dstE = [
];
word dstM = [
];
];
```

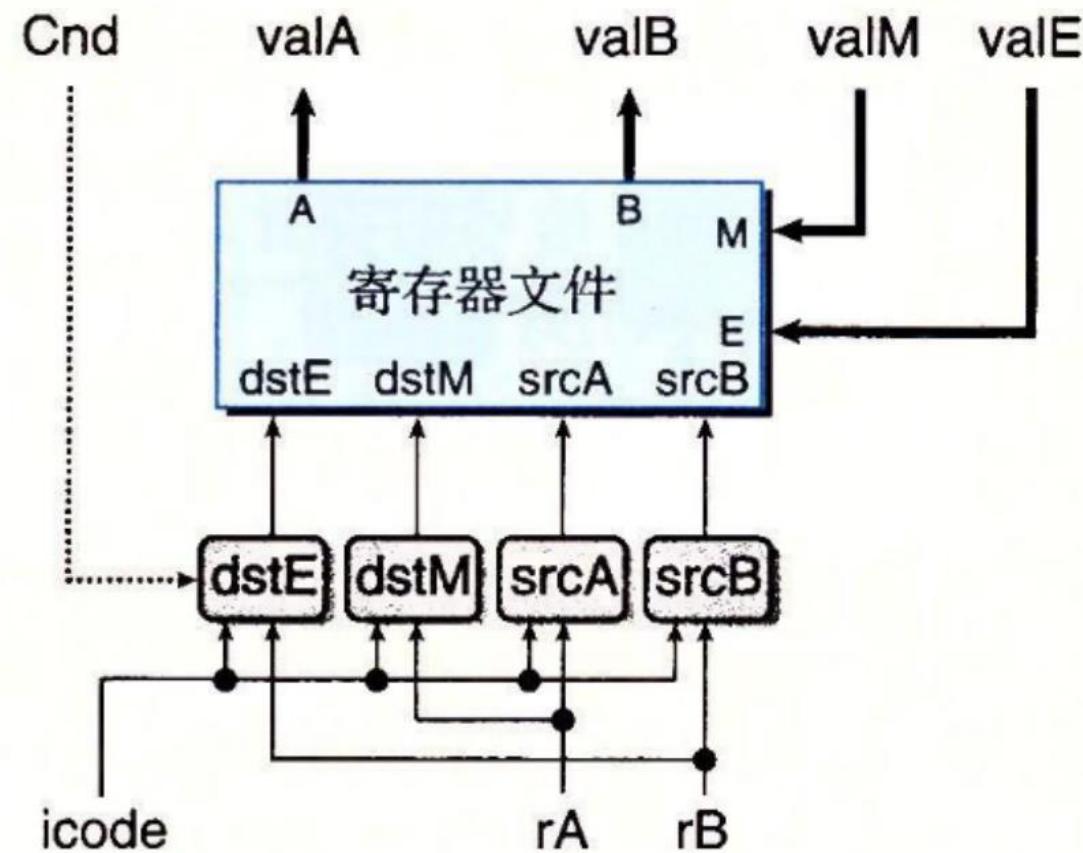


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 :
];
word dstE = [
];
word dstM = [
];
];
```

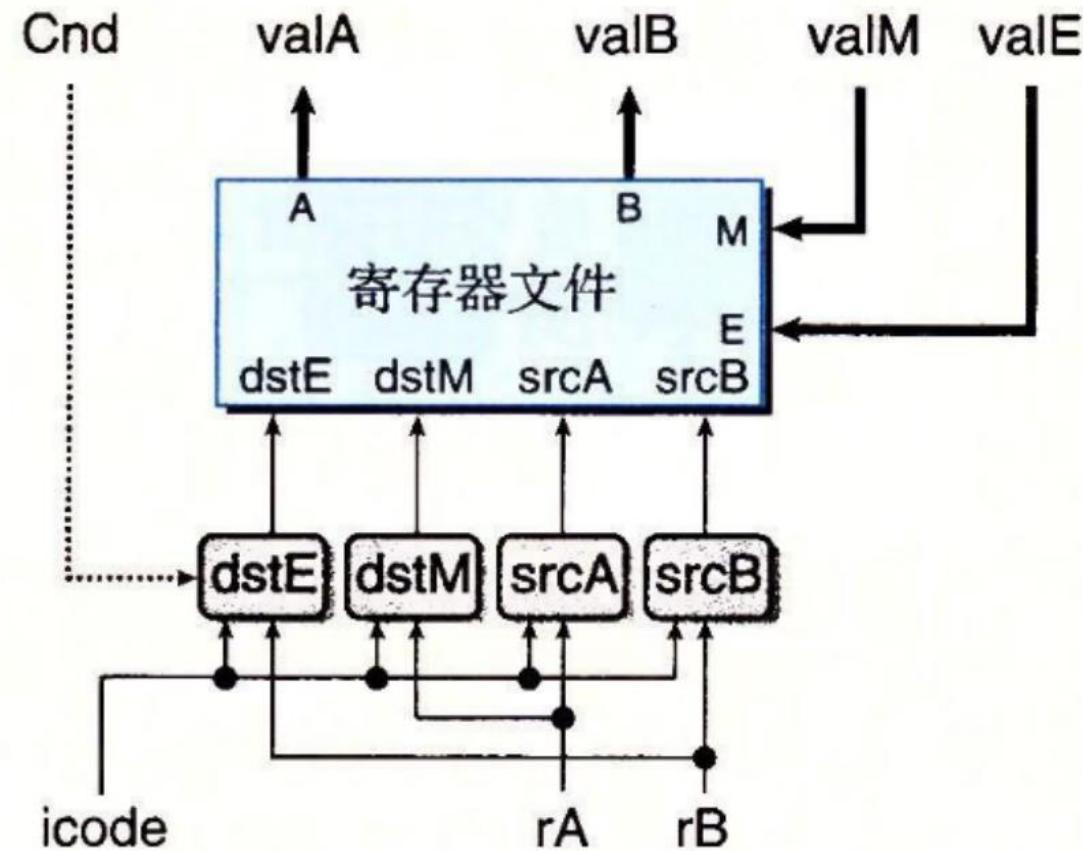


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 { } && Cnd : rB;
    icode in { } : rB;
    icode in { } :
    1 :
];
word dstM = [
];
];
```

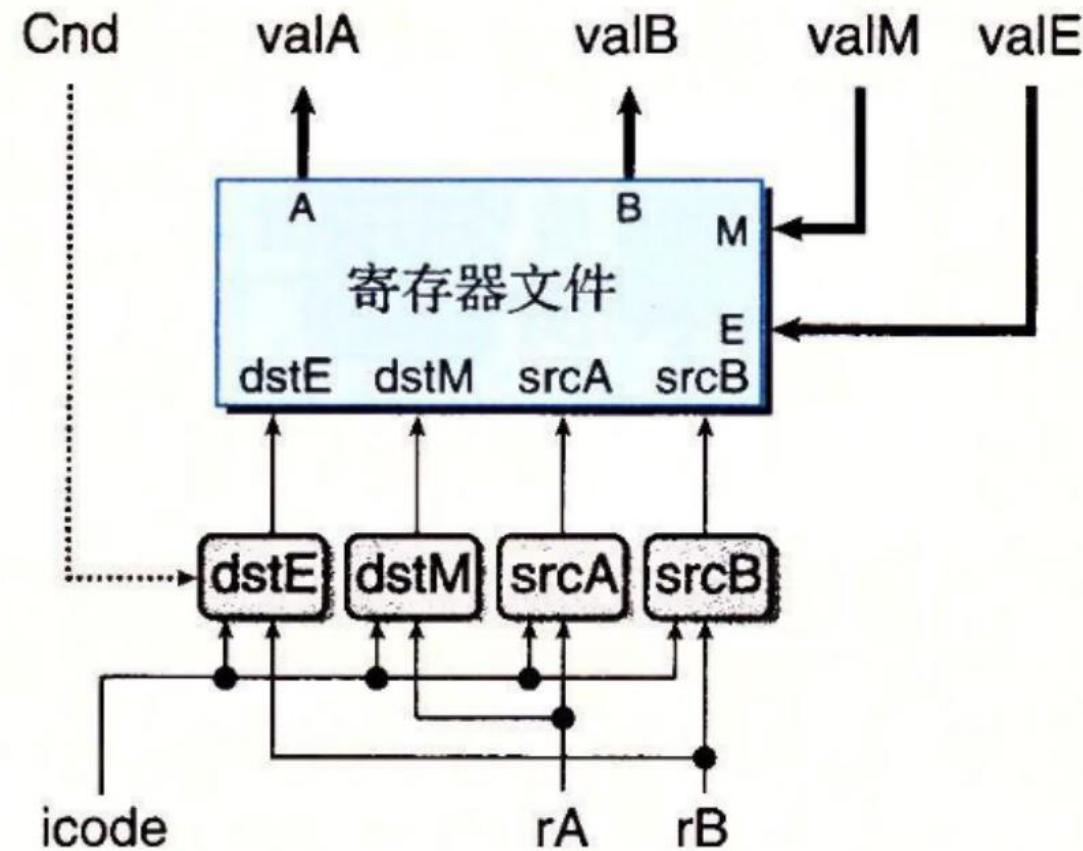


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 { } : rB;
    icode in { } :
    1 :
];
word dstM = [
];
];
```

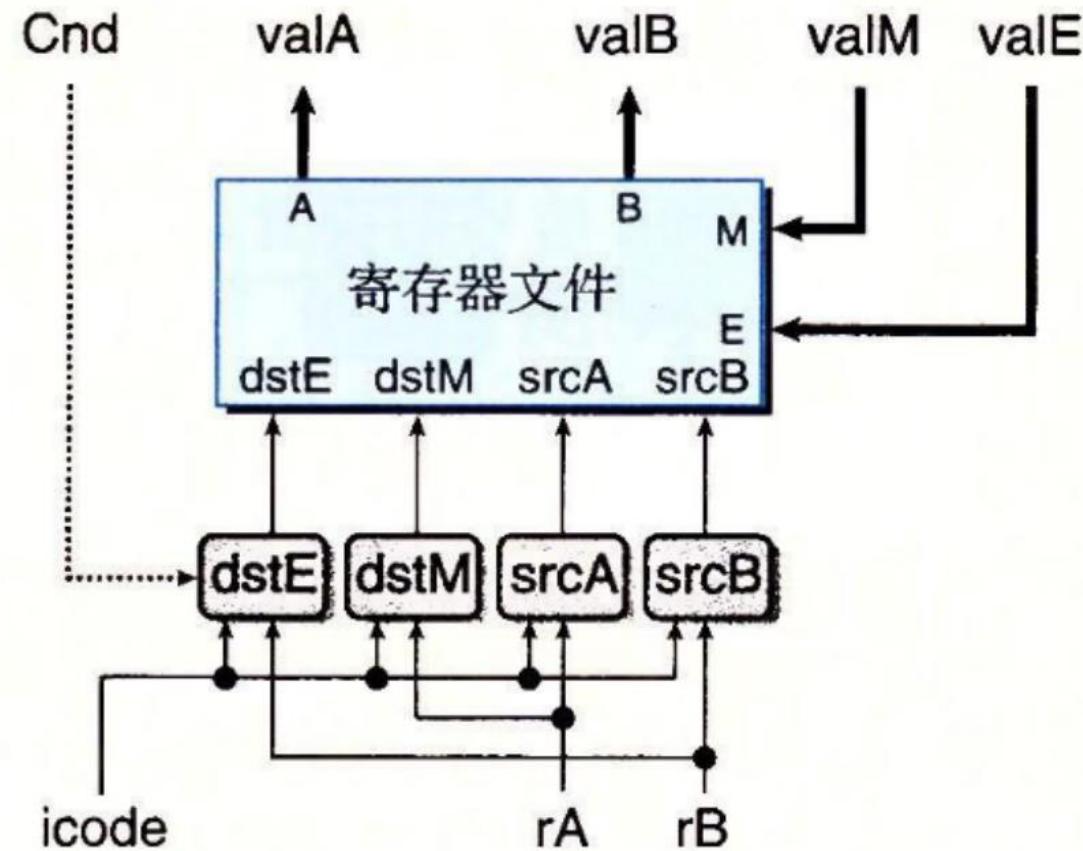


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 {
        ...
        1 :
    };
word dstM = [
];
];
```

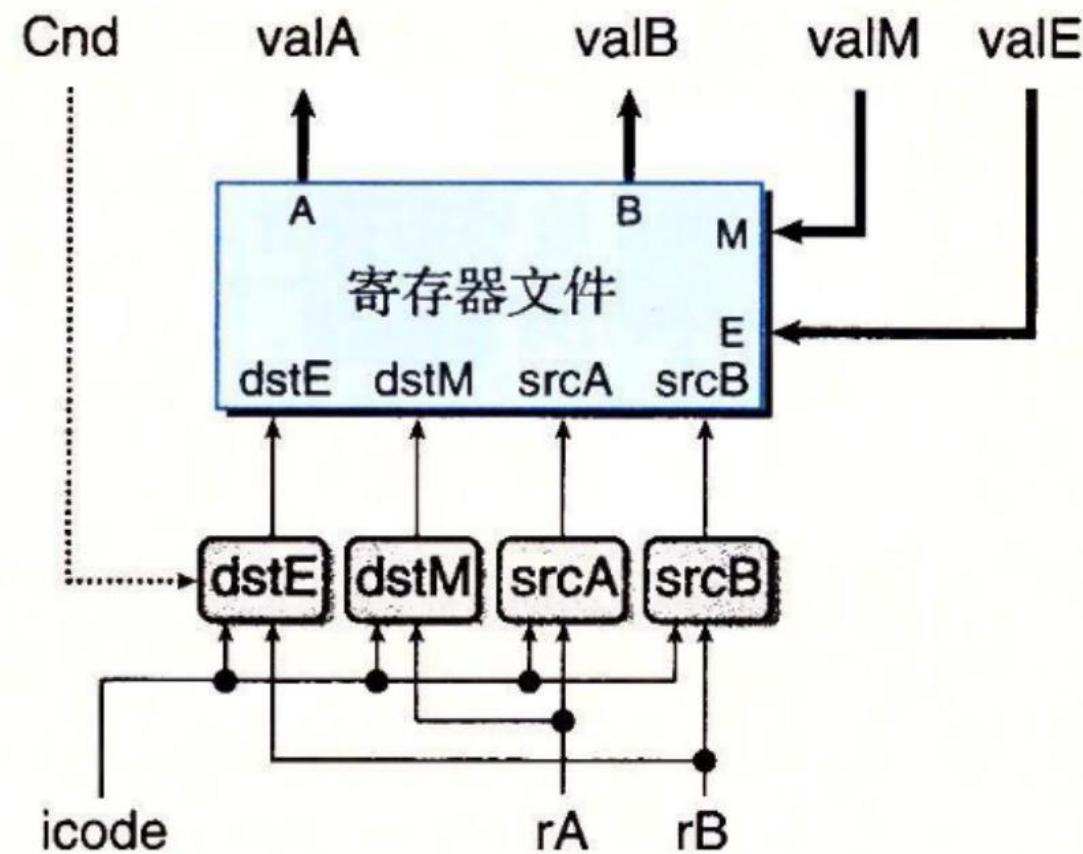


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 } :
    1 :
];
word dstM = [
];
];
```

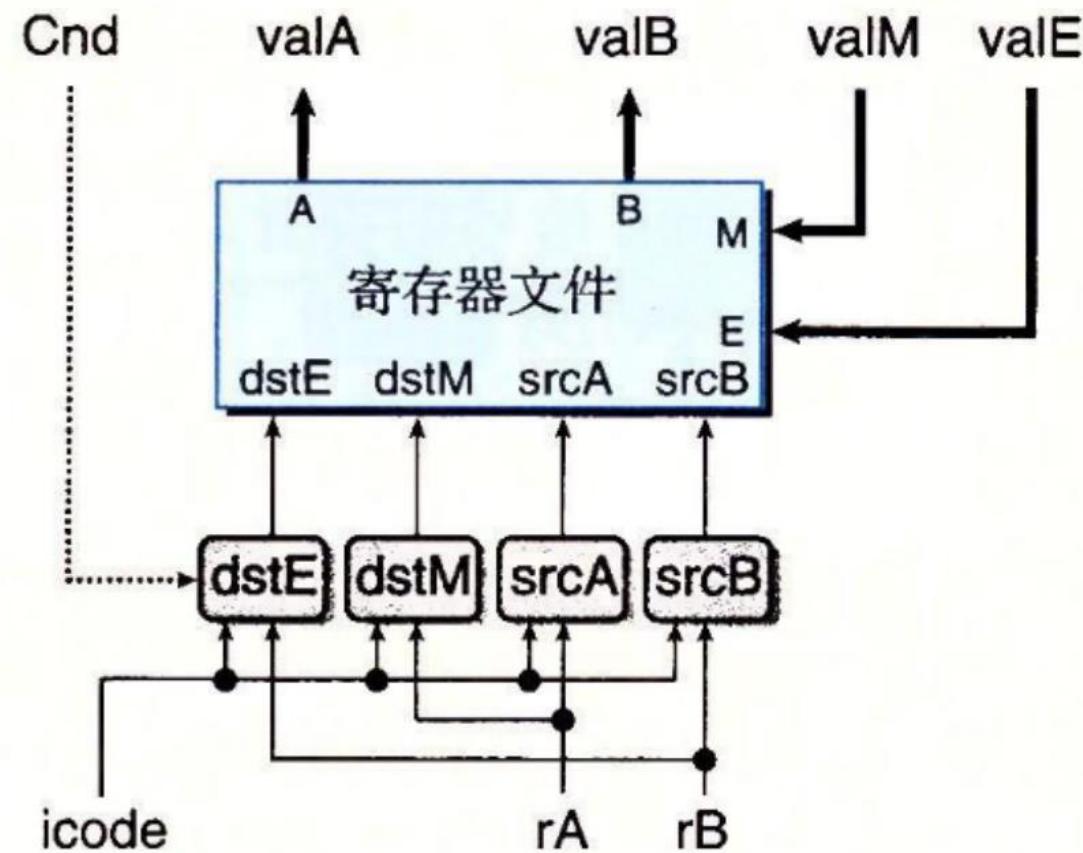


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 :
];
word dstM = [
];
];
```

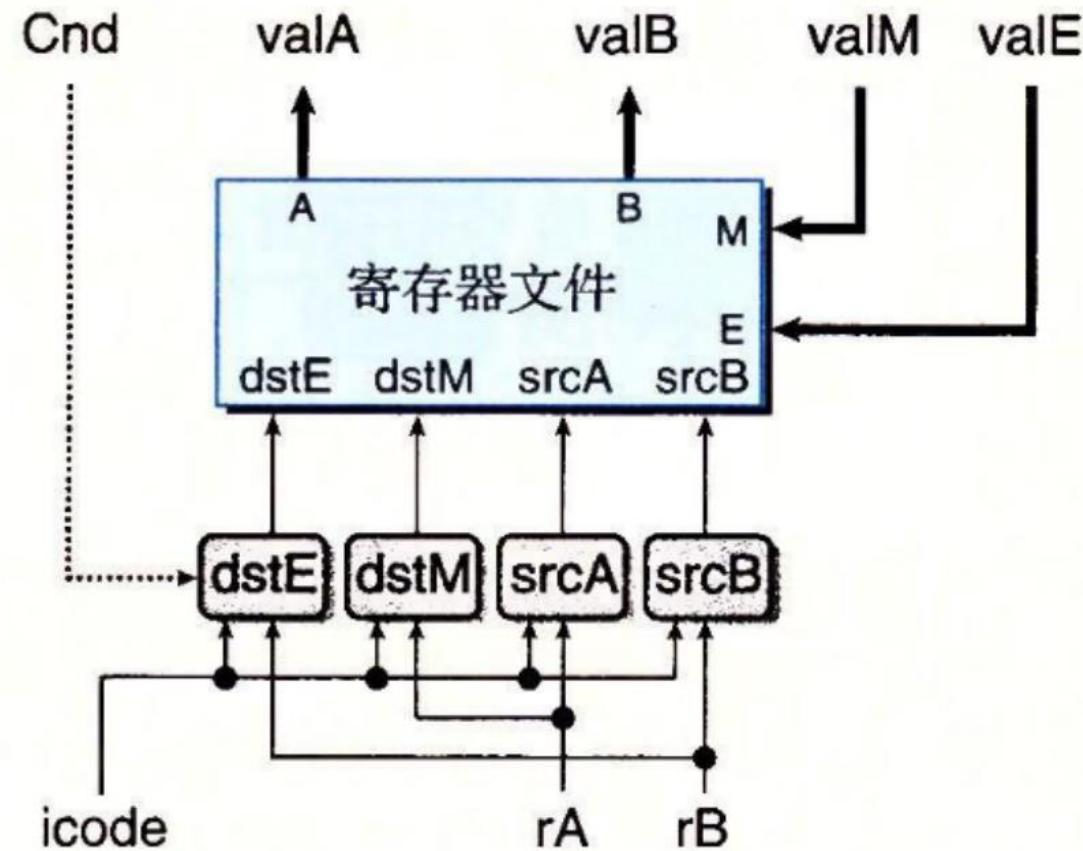


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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 = [
];
];
```

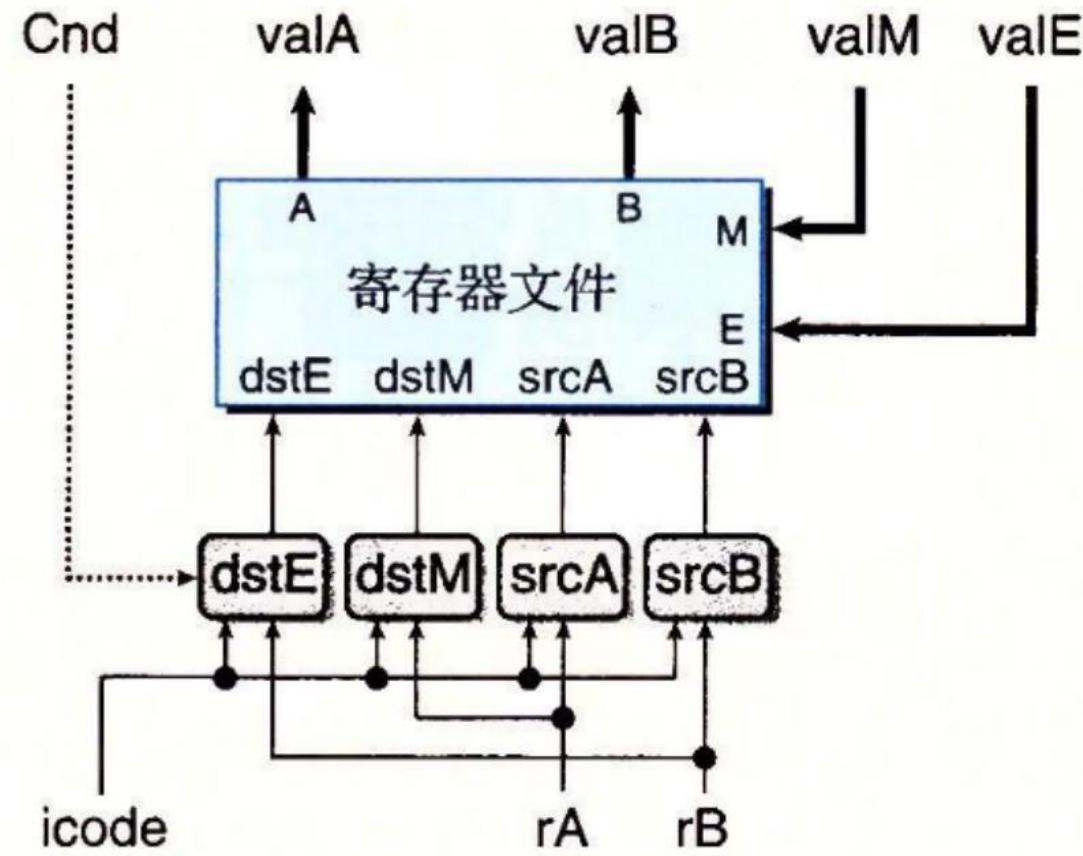


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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
];
```



地址运算往往赋给rsp, 值运算赋给rb, 访存赋给ra

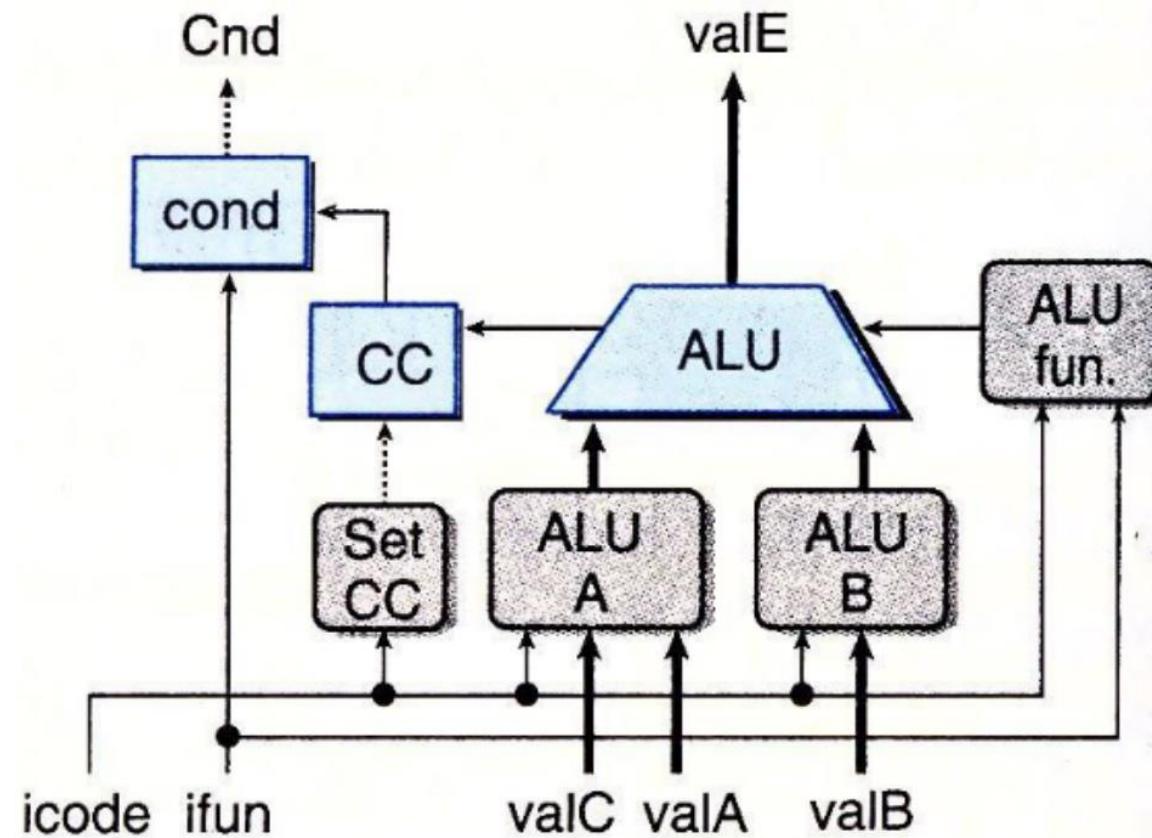
访存(memory)、写回(write back)

<code>OPq rA, rB</code>	<code>rrmovq rA, rB</code>	<code>irmovq V, rB</code>	<code>rmmovq rA, D(rB)</code>	<code>mrmovq D(rB), rA</code>
$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$M_8[valE] \leftarrow valA$	$valM \leftarrow M_8[valE]$
<code>pushq rA</code>	<code>popq rA</code>	<code>jXX Dest</code>	<code>call Dest</code>	<code>ret</code>
$M_8[valE] \leftarrow valA$	$valM \leftarrow M_8[valA]$		$M_8[valE] \leftarrow valP$	$valM \leftarrow M_8[valA]$
$R[%rsp] \leftarrow valE$	$R[%rsp] \leftarrow valE$		$R[%rsp] \leftarrow valE$	$R[%rsp] \leftarrow valE$
	$R[rA] \leftarrow valM$			

Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

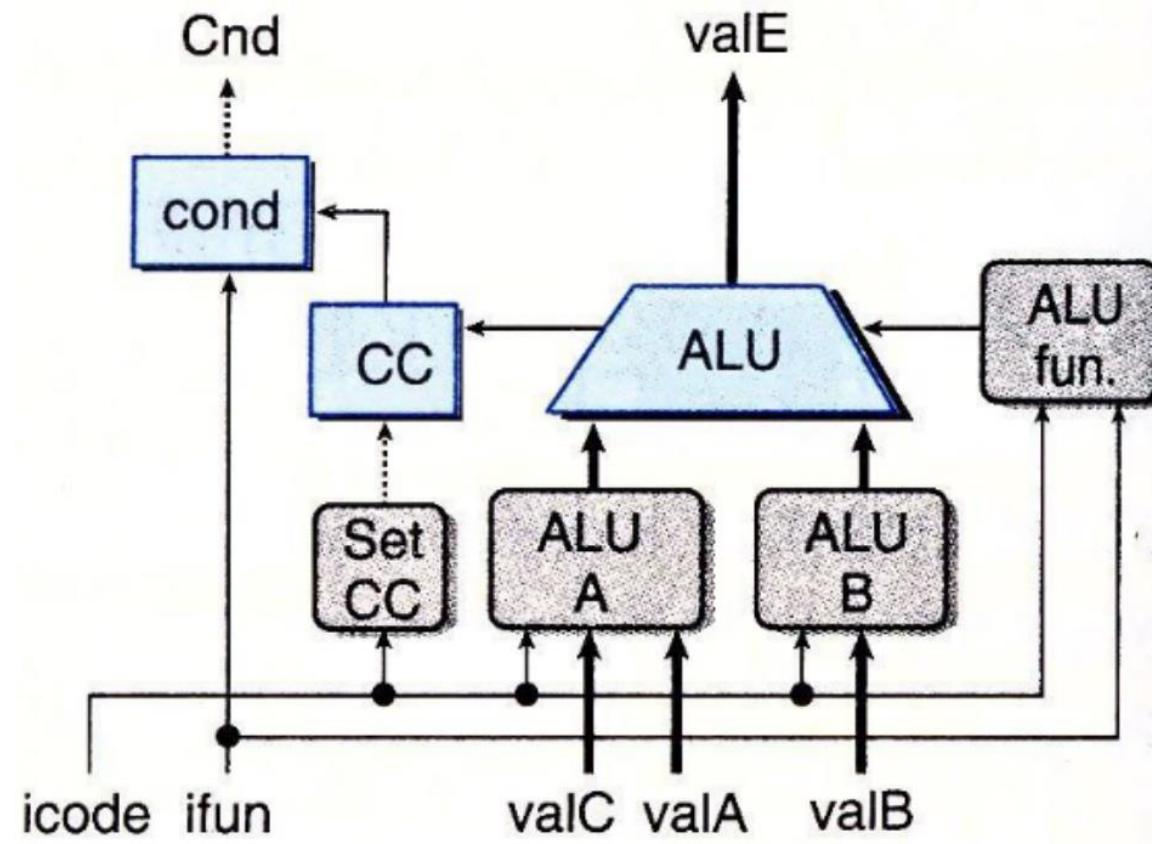
```
word aluA = [
    icode in { } : valA;
    icode in { } : valC;
    icode in { } : -8;
    icode in { } : 8;
];
word aluB = [
];
word alufun = [
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

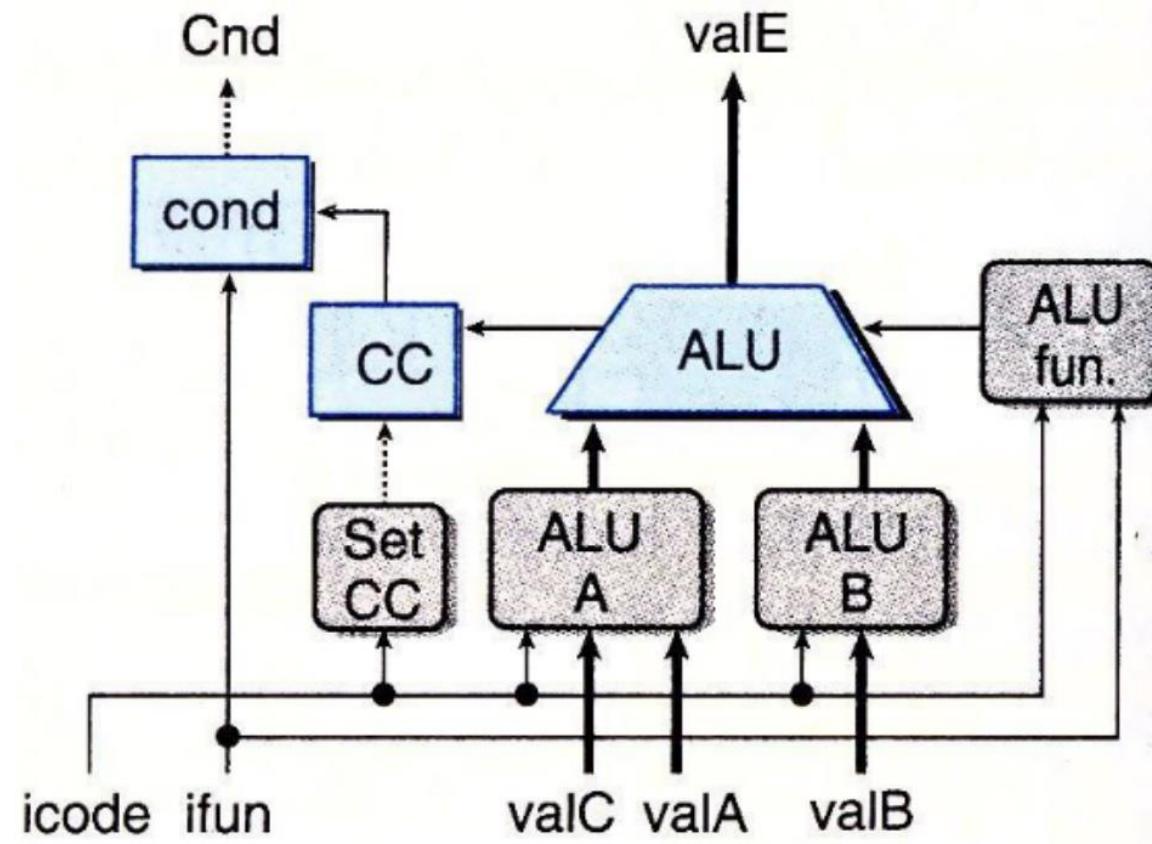
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in {
        } : valC;
    icode in {
        } : -8;
    icode in {
        } : 8;
];
word aluB = [
];
word alufun = [
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

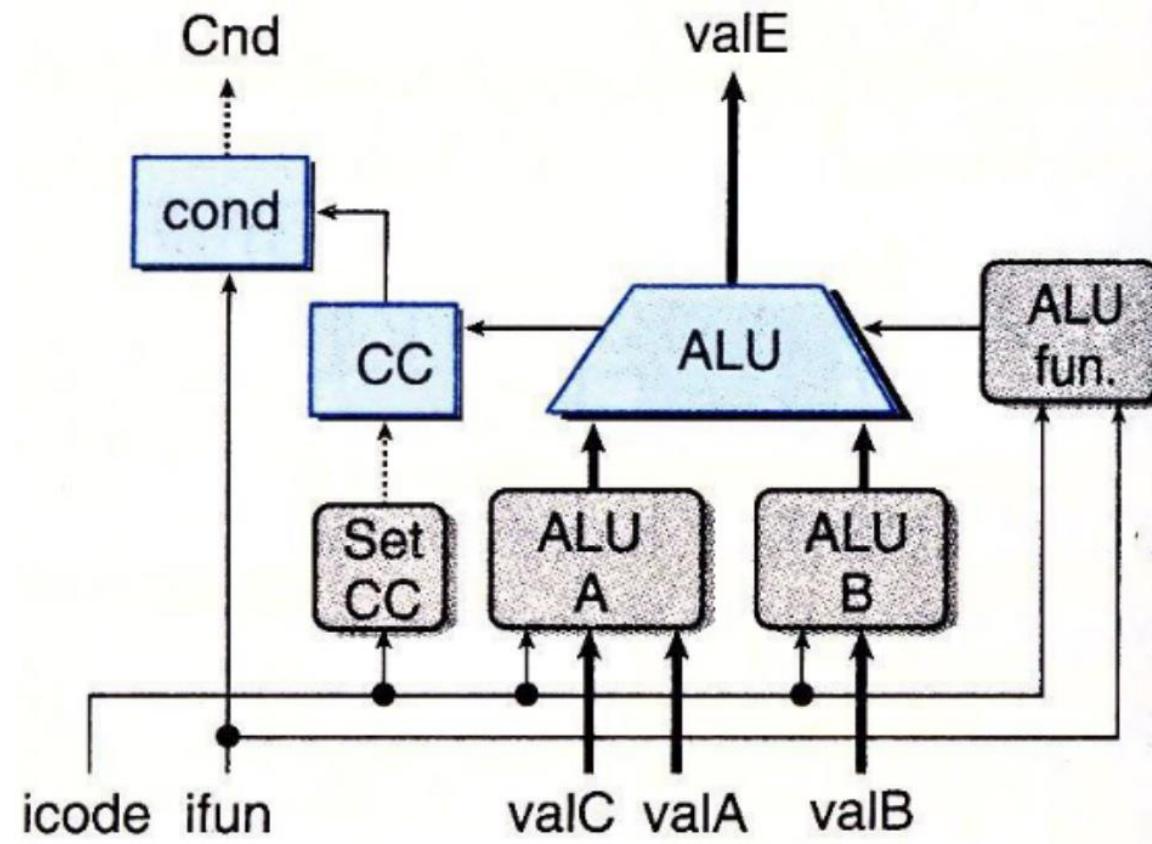
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { } : -8;
    icode in { } : 8;
];
word aluB = [
];
word alufun = [
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

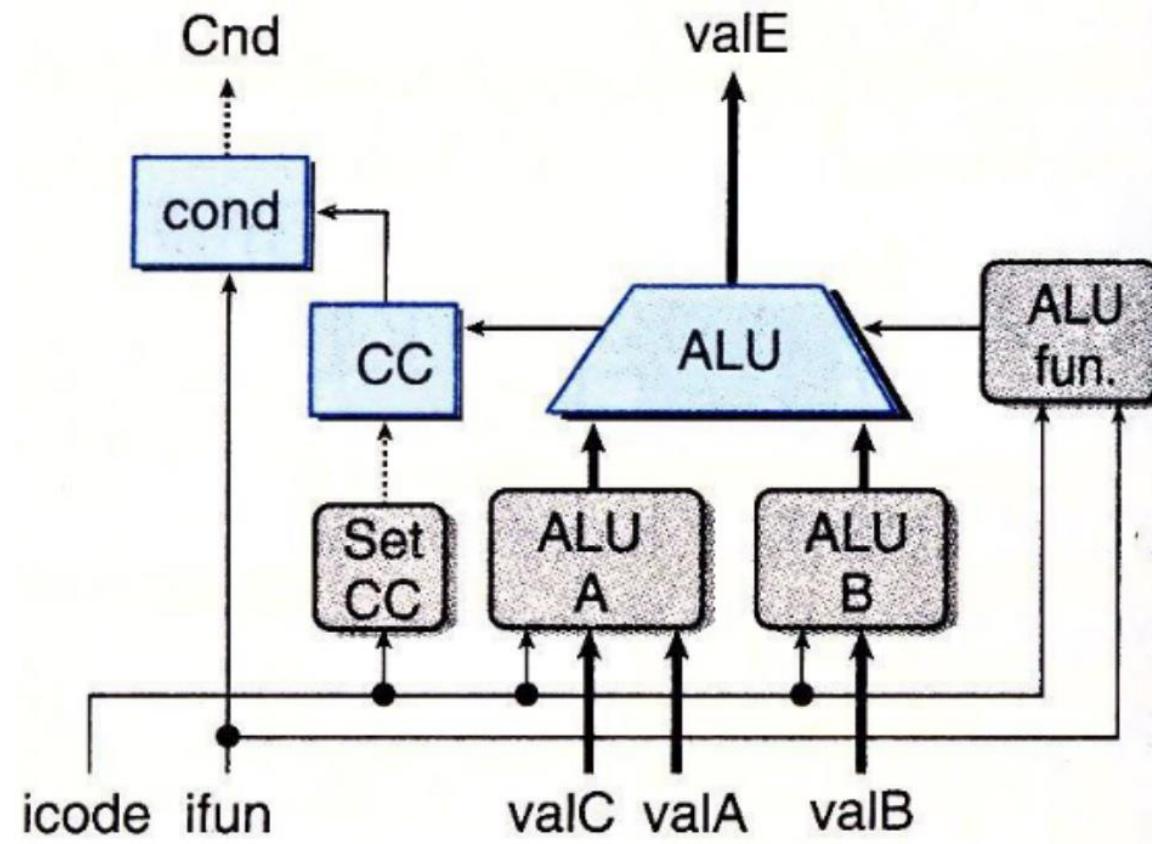
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { } : 8;
];
word aluB = [
];
word alufun = [
];
bool set_cc =
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Execute

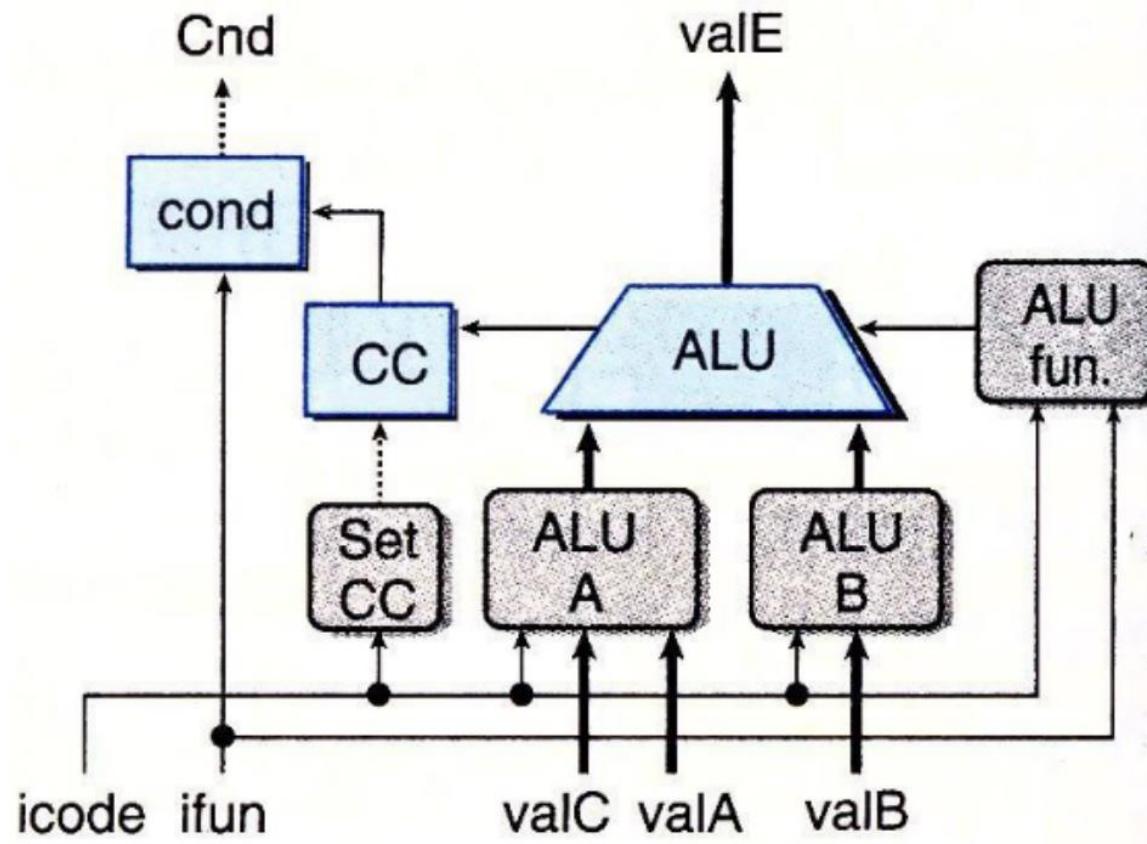
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    #
];
word aluB = [
];
word alufun = [
];
bool set_cc =
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Execute

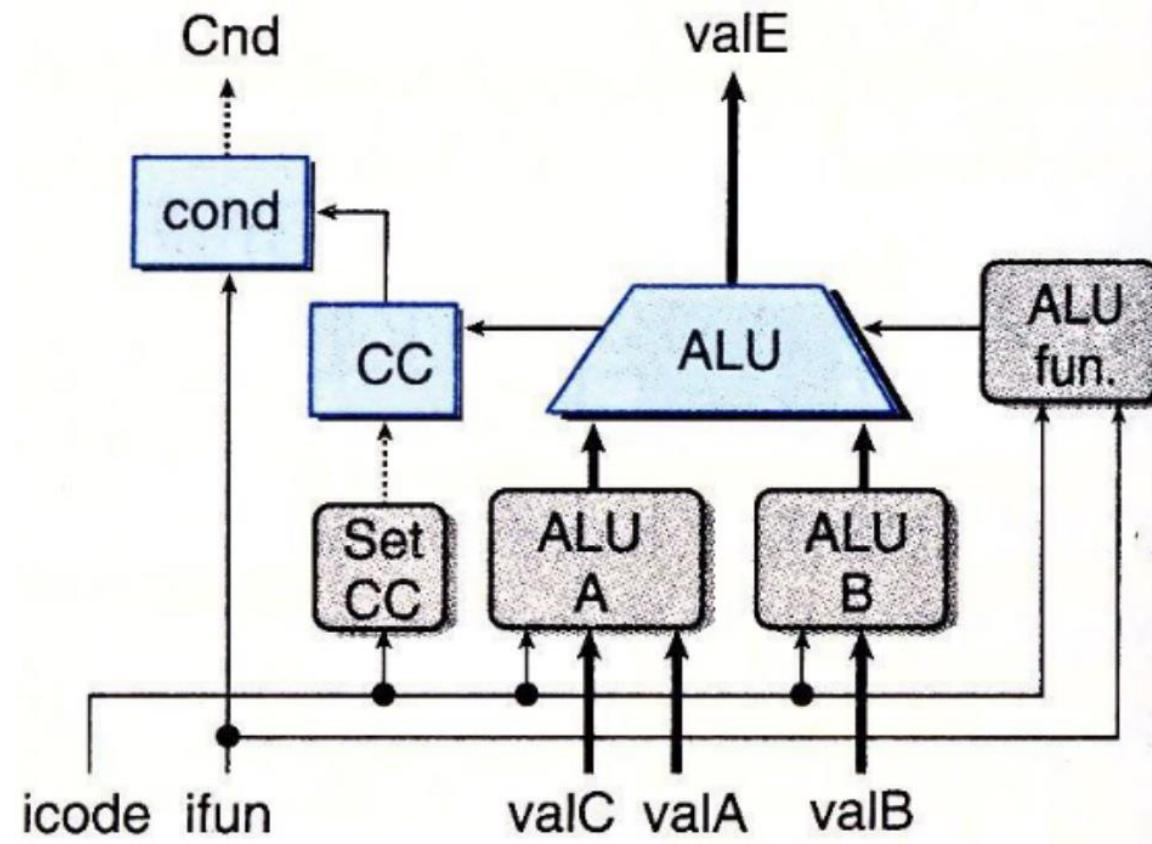
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
word aluB = [
];
word alufun = [
];
bool set_cc =
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Execute

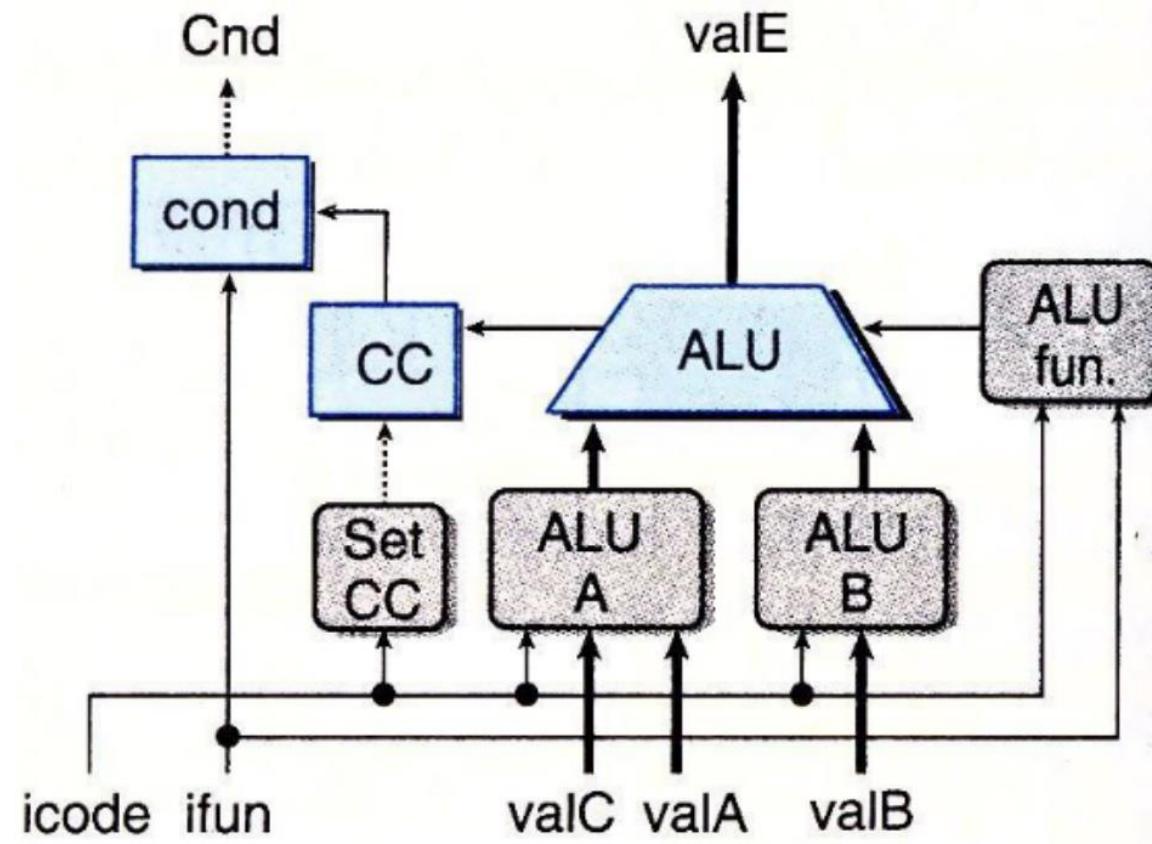
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
word aluB = [
    icode in {
        } : valB;
    icode in {
        } : 0;
    #
];
word alufun = [
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

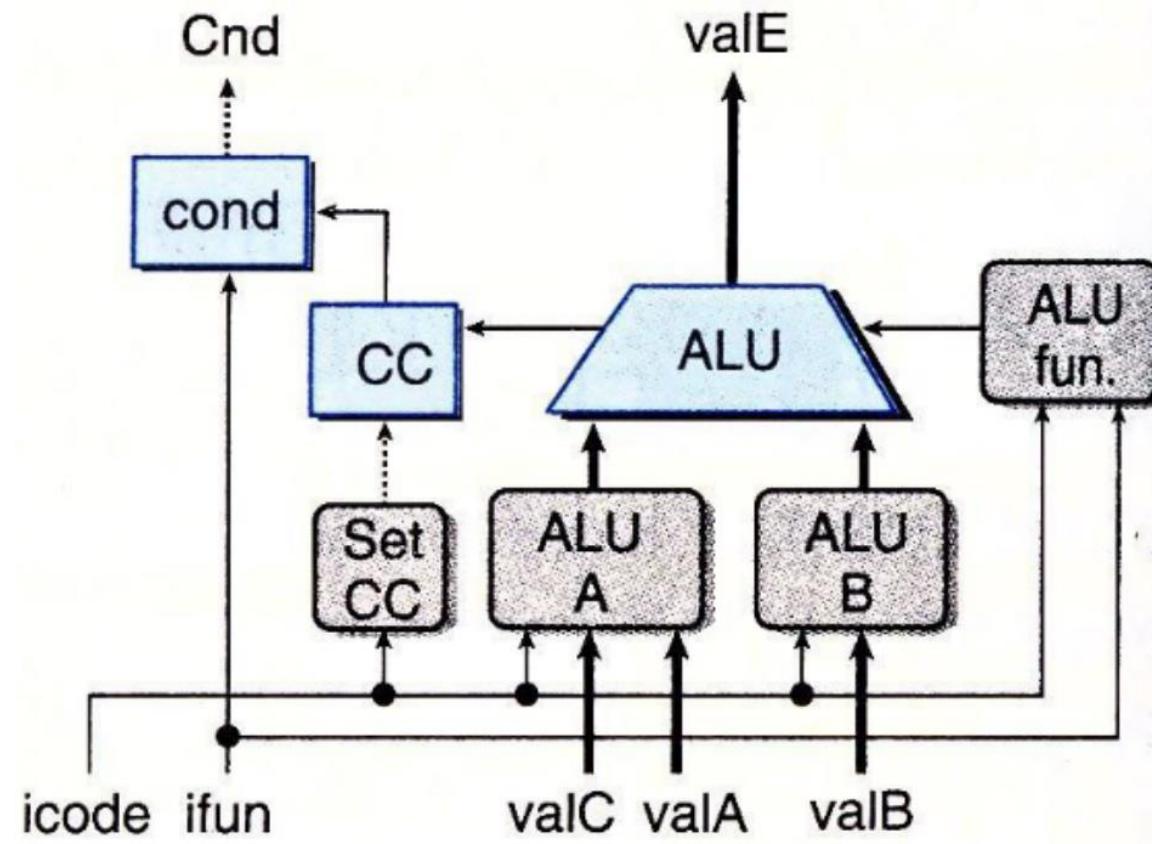
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                IPUSHQ, IRET, IPOPQ } : valB;
    icode in { } : 0;
    #
];
word alufun = [
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

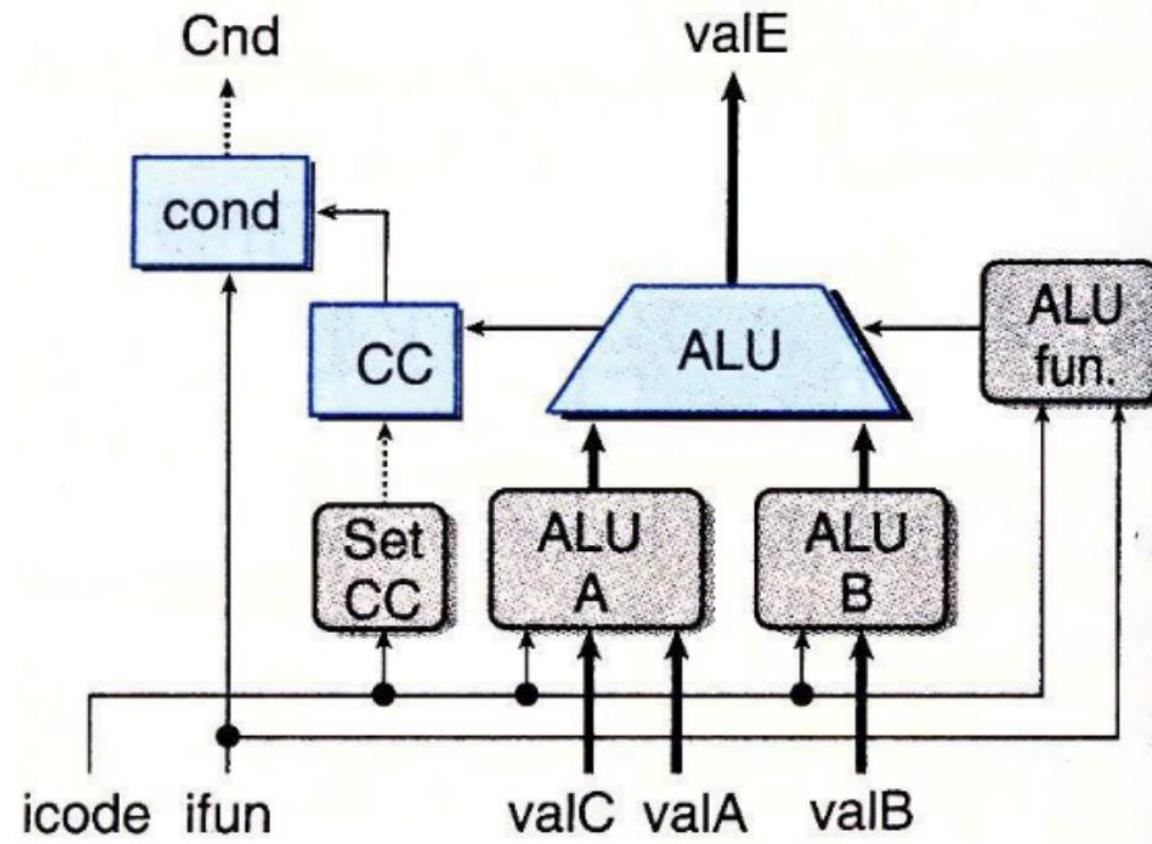
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                IPUSHQ, IRET, IPOPQ } : valB;
    icode in { IRRMOVQ, IIRMOVQ } : 0;
    #
];
word alufun = [
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

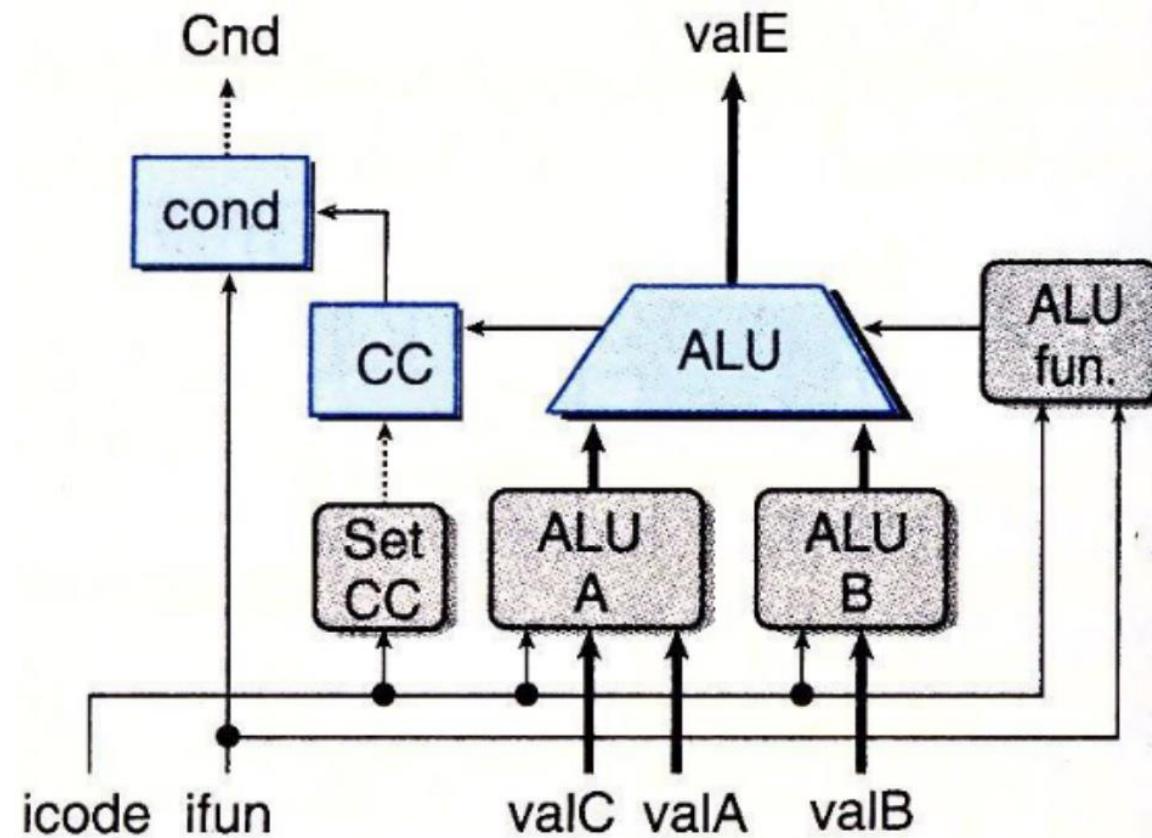
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                IPUSHQ, IRET, IPOPQ } : valB;
    icode in { IRRMOVQ, IIRMOVQ } : 0;
    # Other instructions don't need ALU
];
word alufun = [
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

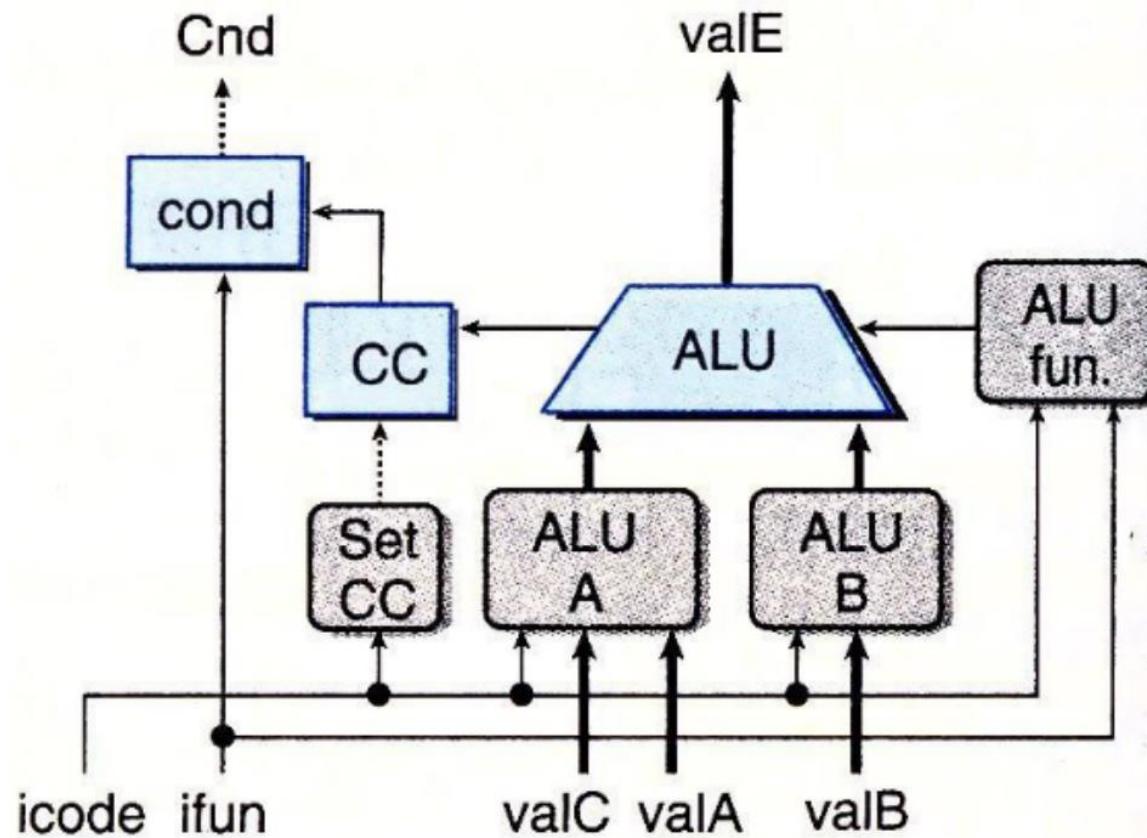
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                IPUSHQ, IRET, IPOPQ } : valB;
    icode in { IRRMOVQ, IIRMOVQ } : 0;
    # Other instructions don't need ALU
];
word alufun = [
    icode == IOPQ : ifun;
    1 : ALUADD;
];
bool set_cc =
```



Execute

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

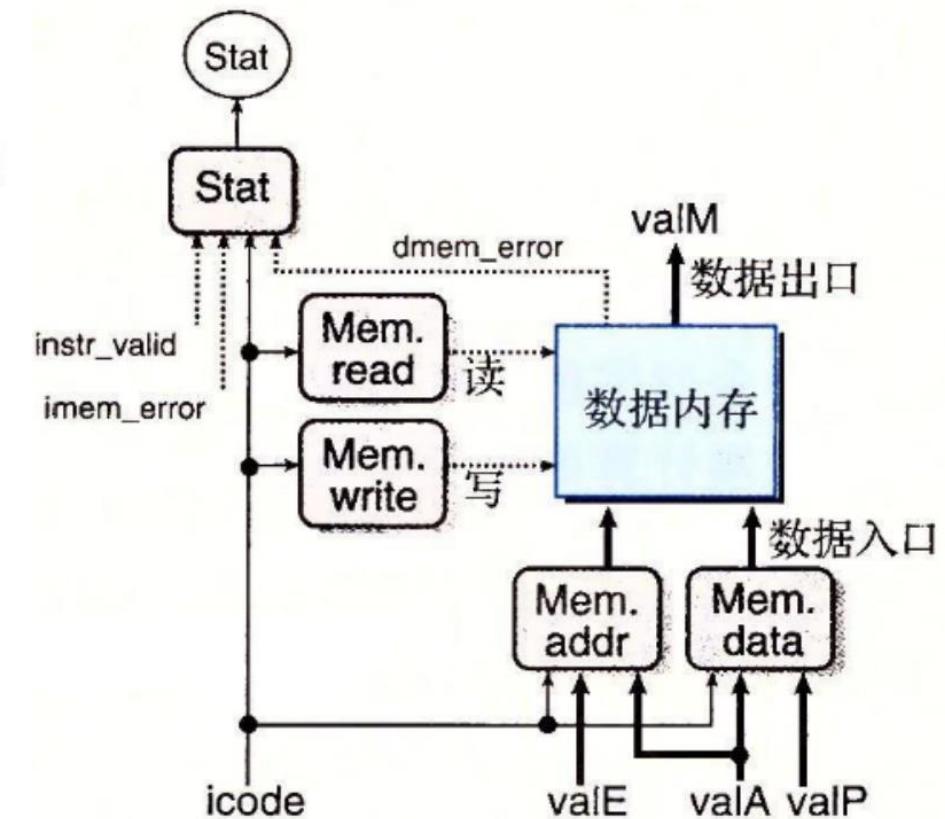
```
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA;
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
    icode in { ICALL, IPUSHQ } : -8;
    icode in { IRET, IPOPQ } : 8;
    # Other instructions don't need ALU
];
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                IPUSHQ,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



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

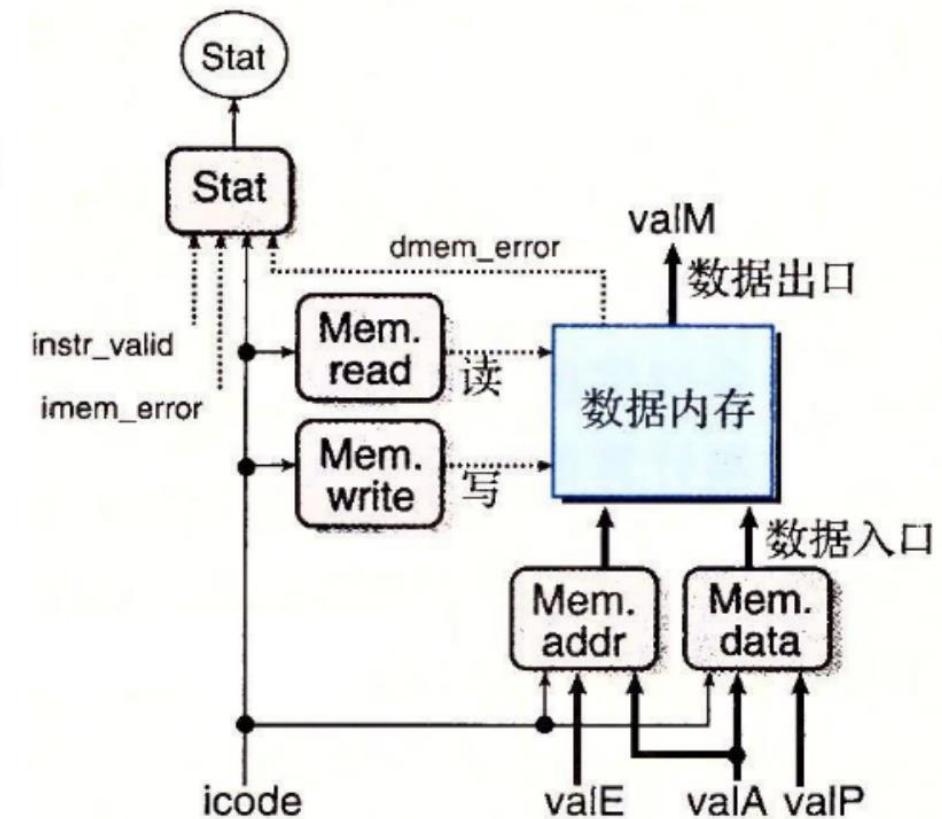
```
bool mem_read = icode in {  
    ...  
};  
  
bool mem_write = icode in {  
    ...  
};  
  
word mem_addr = [  
    ...  
];  
  
word mem_data = [  
    ...  
],  
    int Stat = [  
    ...  
];
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { };
word mem_addr = [
];
word mem_data = [
];
int Stat = [
];
```

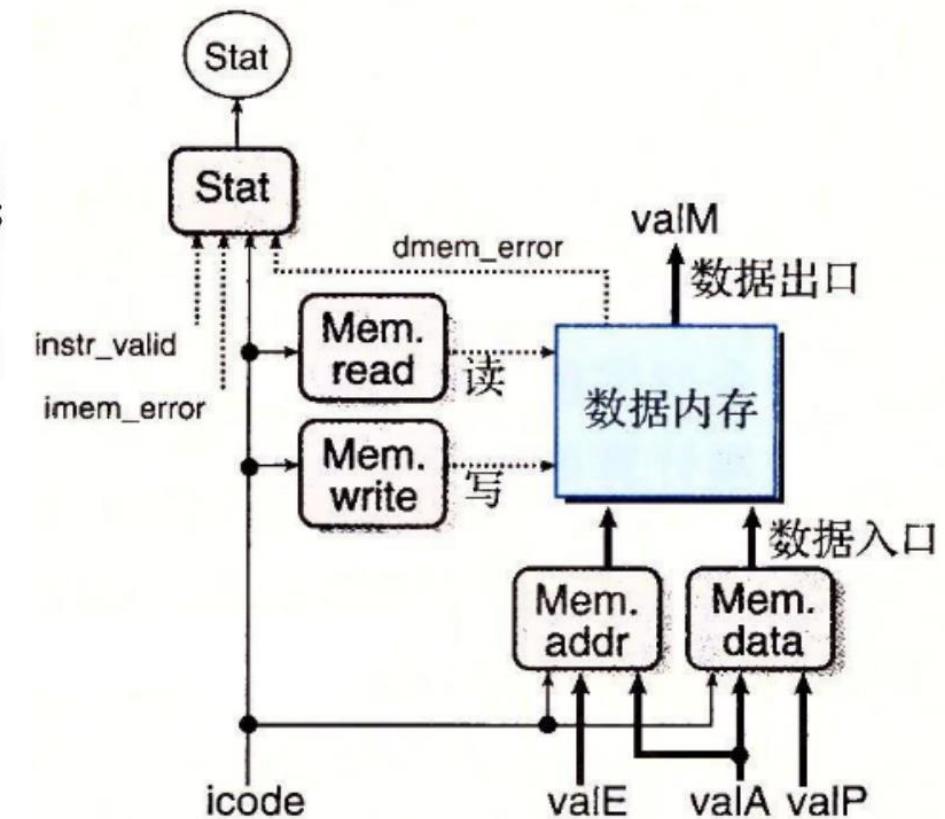


INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { } : valE;
    icode in { } : valA;
    #
];
word mem_data = [
];
int Stat = [ ];

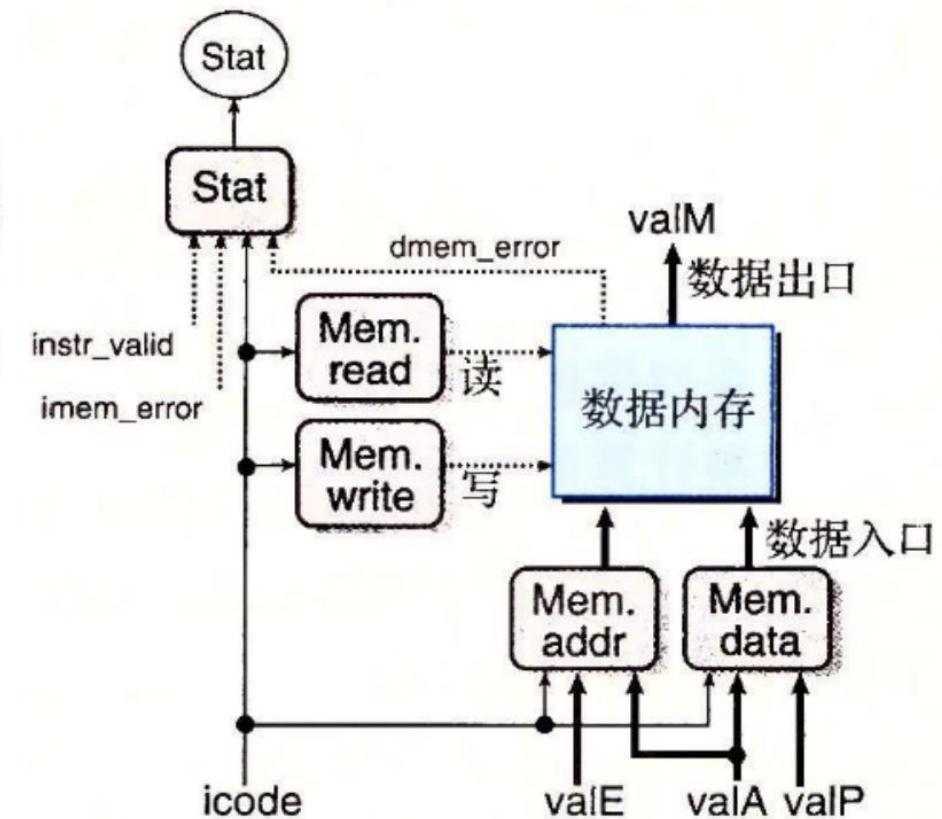
];
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

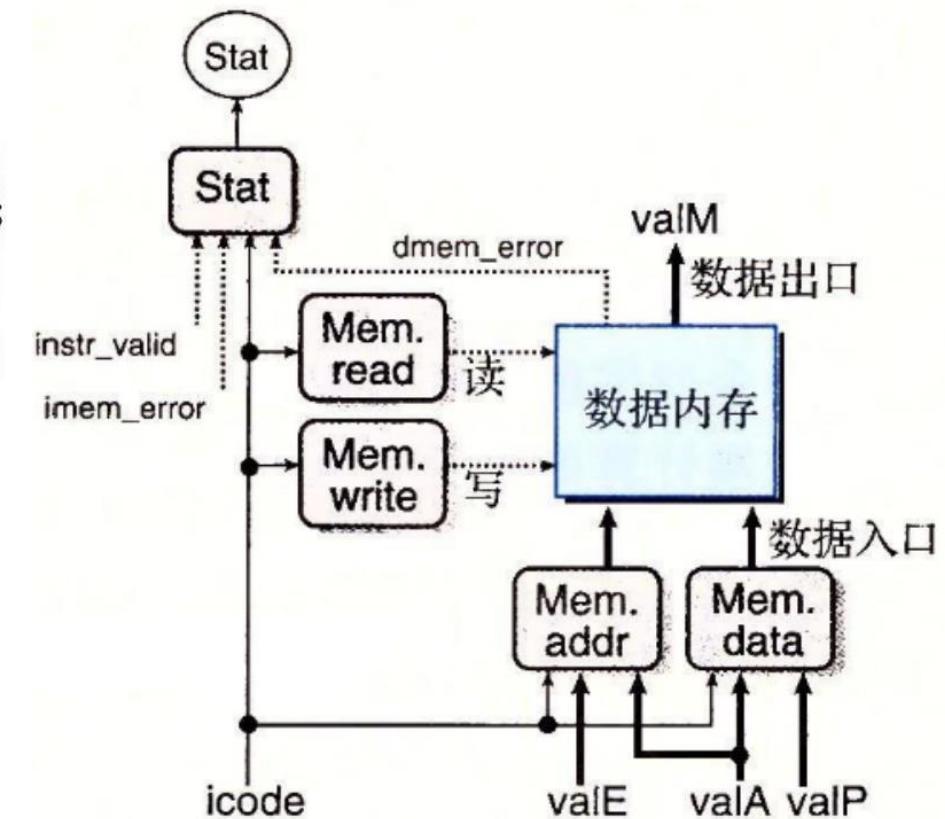
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { } : valA;
    #
];
word mem_data = [
];
int Stat = [
];
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

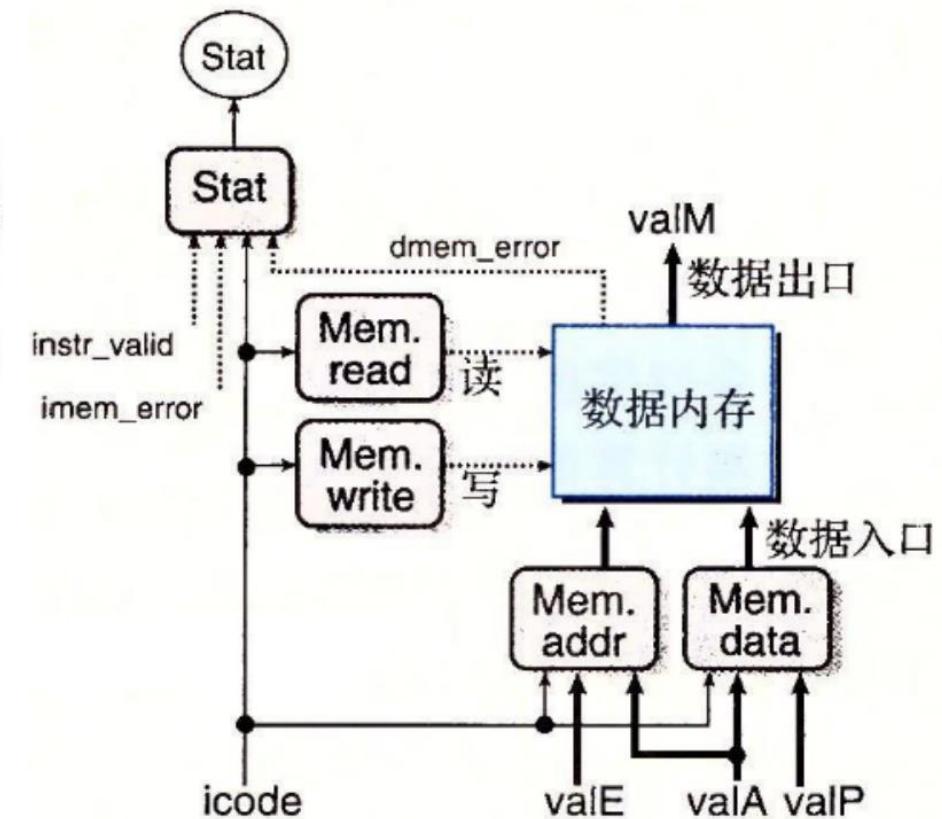
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    #
];
word mem_data = [
];
int Stat = [
];
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

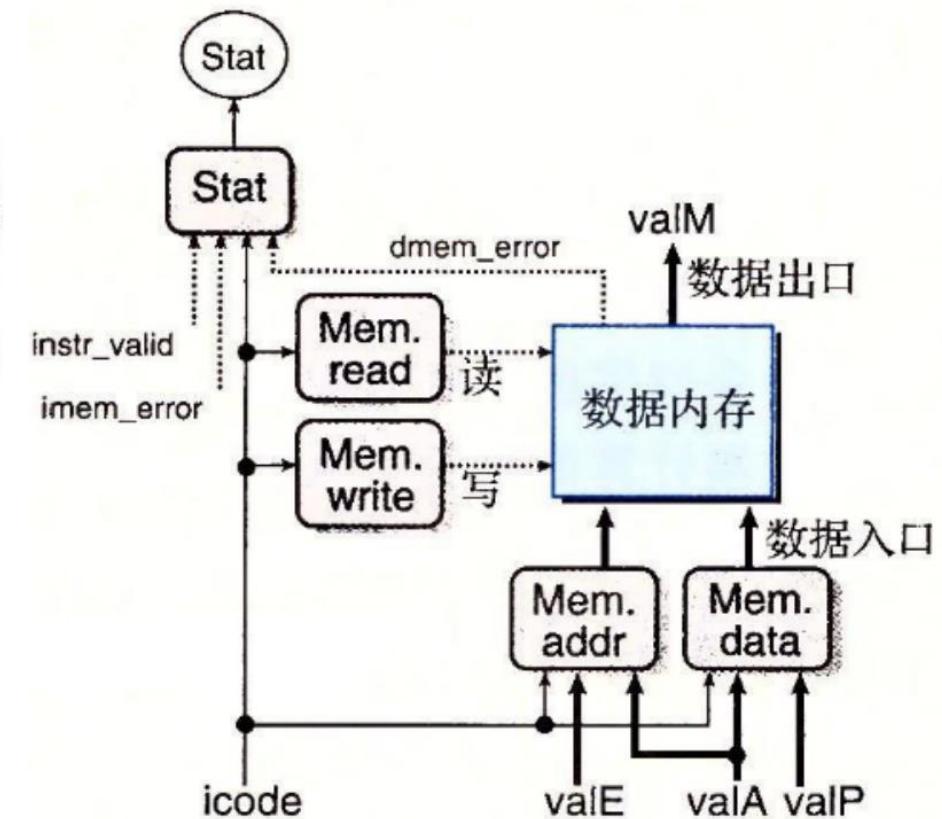
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
word mem_data = [
    # Value from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



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

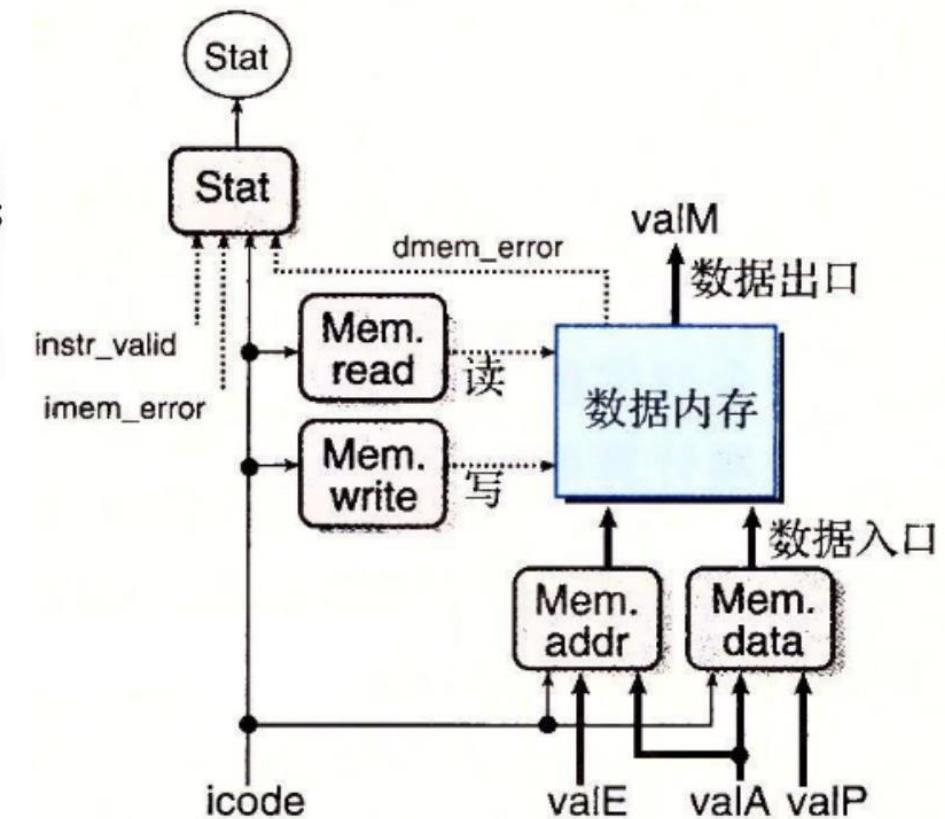
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
word mem_data = [
    # Value from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



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

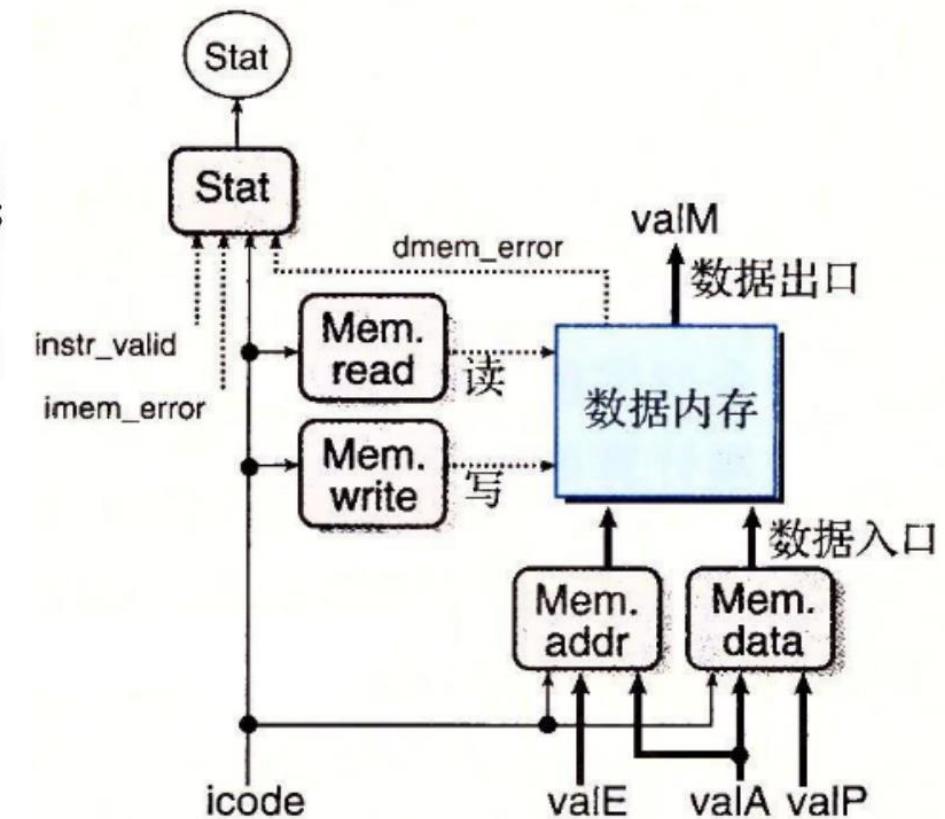
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
word mem_data = [
    # Value from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



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

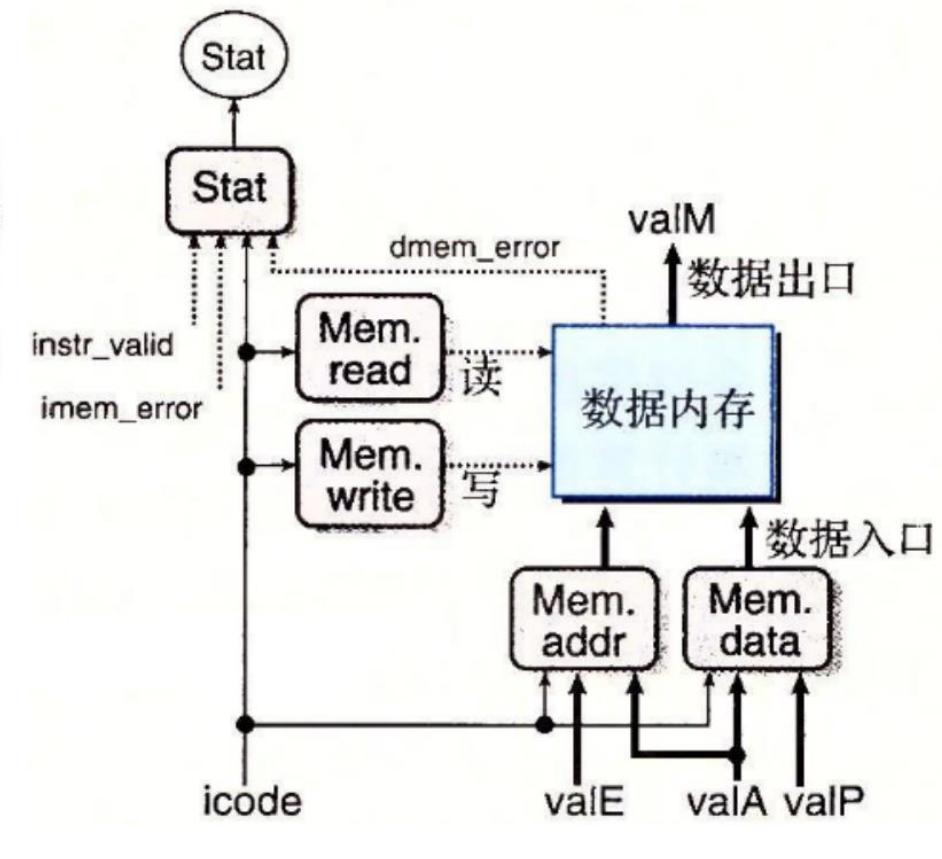
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
word mem_data = [
    # Value from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



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

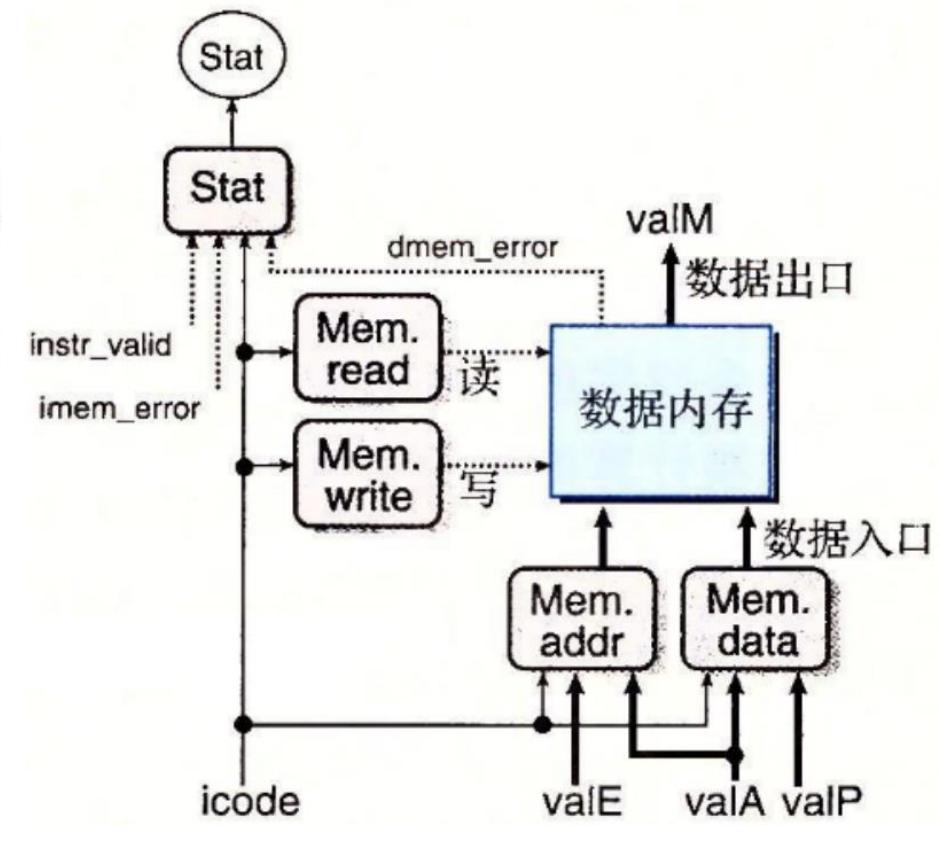
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
word mem_data = [
    # Value from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



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

Memory

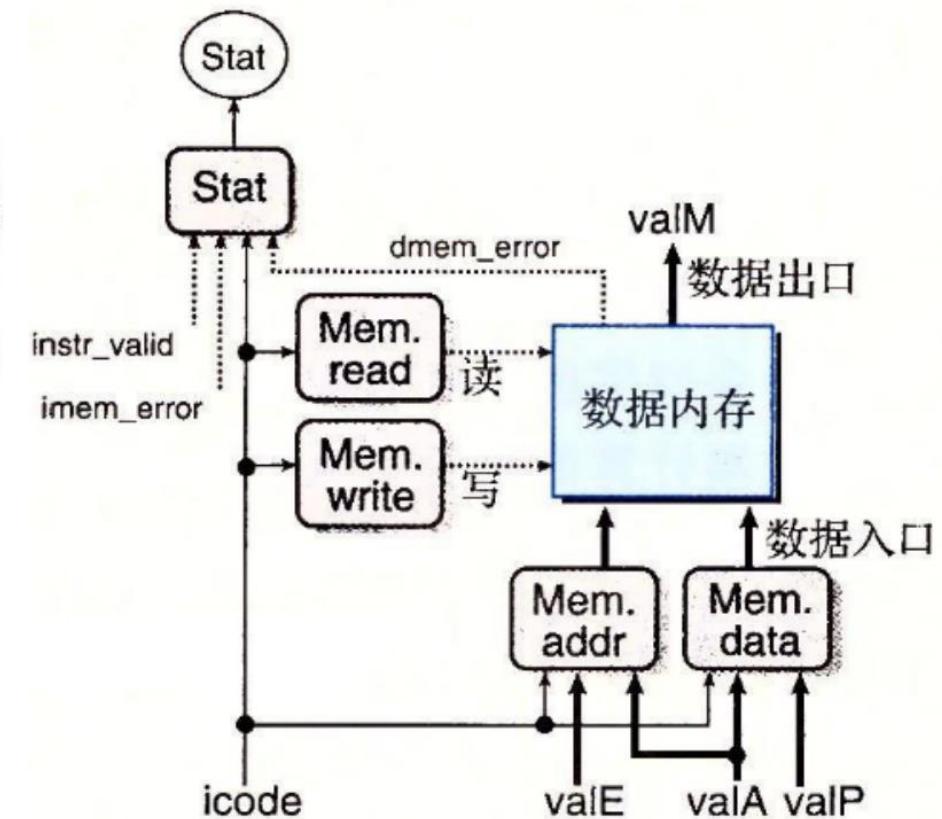
```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA;
    # Other instructions don't need address
];
word mem_data = [
    # Value from 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;
        : SHLT;
    : SAOK;
];
```



INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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;
    : SAOK;
];
```



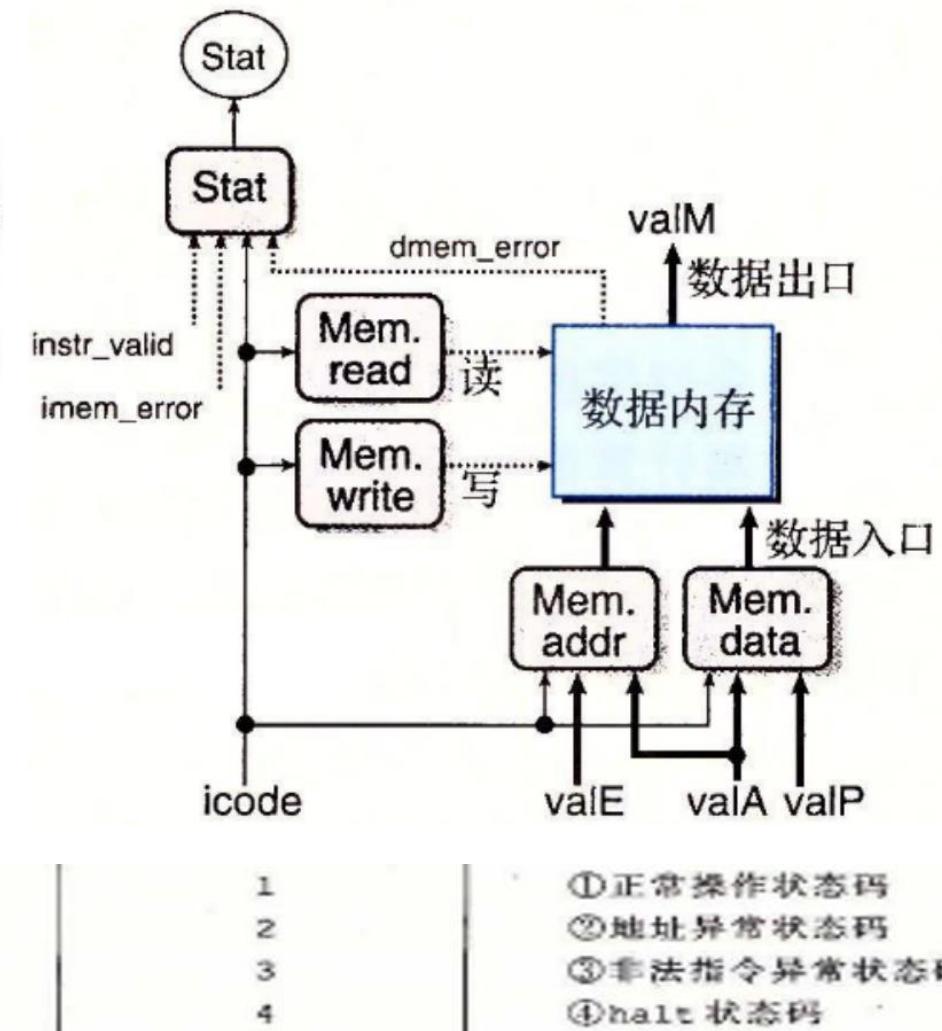
INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

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;
];

```

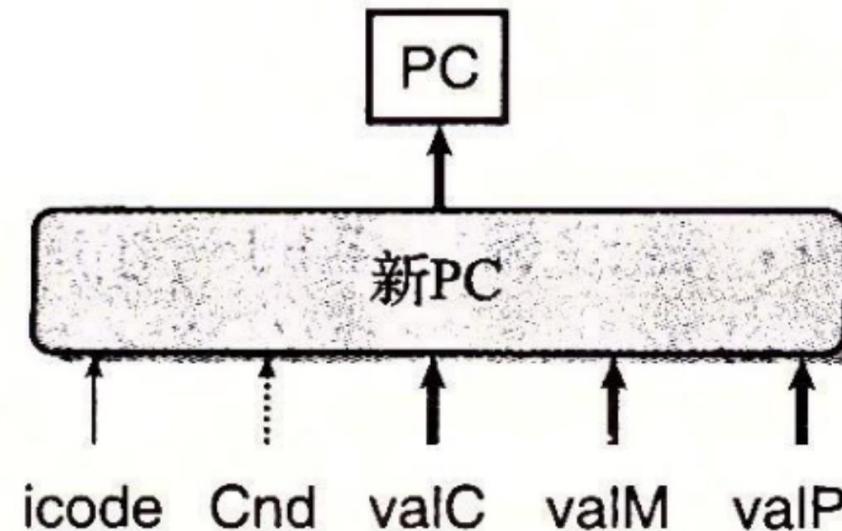


New PC

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

```
word new_pc = [
```

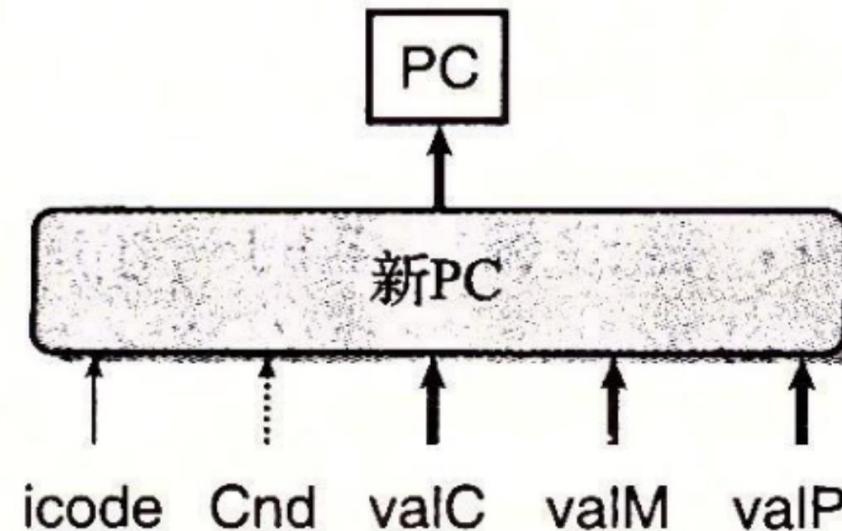
```
];
```



New PC

INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ

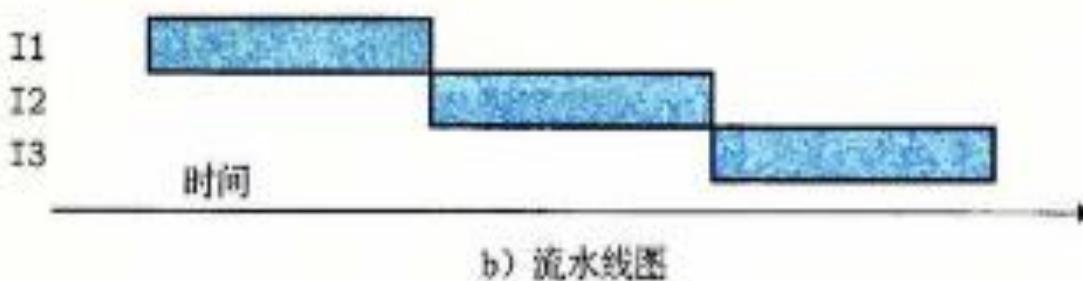
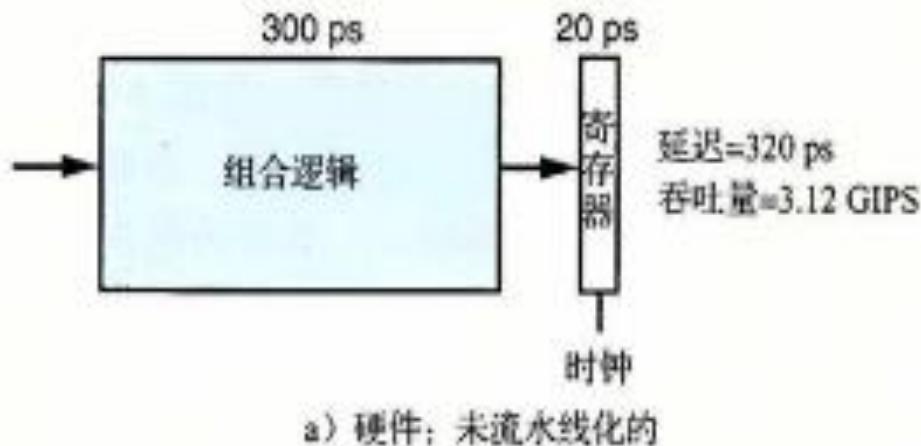
```
word new_pc = [  
    # Call.  Use instruction constant  
    icode == ICALL : valC;  
    # Taken branch.  Use instruction constant  
    icode == IJXX && Cnd : valC;  
    # Completion of RET instruction.  Use value from stack  
    icode == IRET : valM;  
    # Default: Use incremented PC  
    1 : valP;  
];
```



Pipelined (CS:APP Ch. 4.4)

王效乐

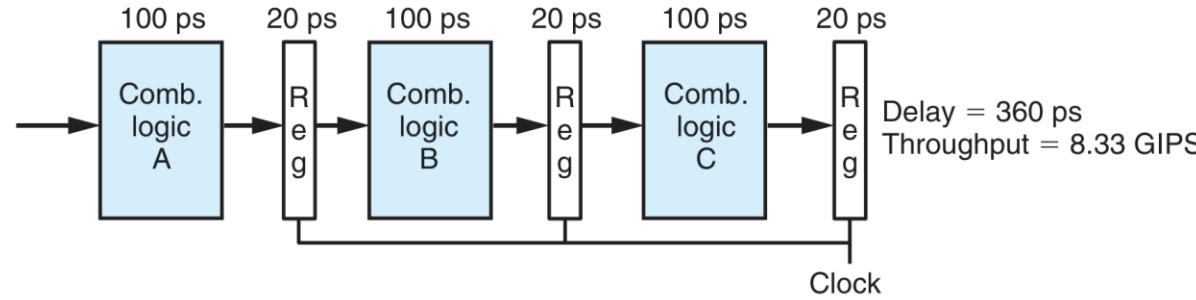
计算流水线



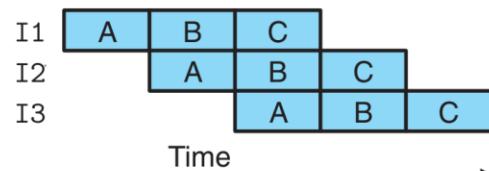
吞吐量 = 1 / 320ps (延迟) = 3.12GIPS

1GIPS 就是每秒十亿条指令

流水线操作



(a) Hardware: Three-stage pipeline



(b) Pipeline diagram

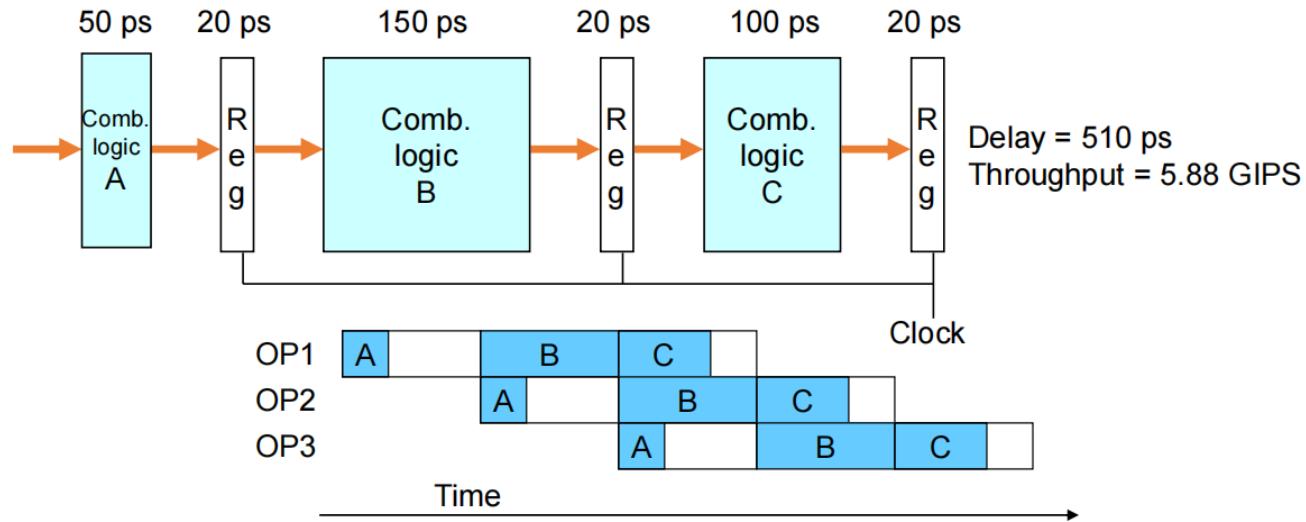
总时长更长，但是吞吐量更多

假设将系统执行的计算分成三个部分，在各阶段之间放上流水线寄存器，就能在第一条指令执行B阶段时让第二条指令执行A阶段，减少了浪费。

吞吐量 = $1/120\text{ps} = 8.33\text{GIPS}$ 是原来的2.67倍

Limitations: Nonuniform Delays

——不一致划分

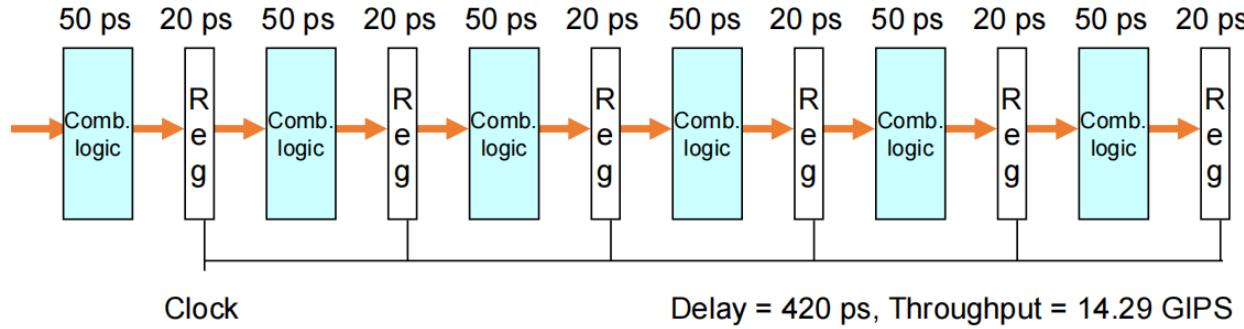


- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

- 实际操作中，不易划分为一组具有相同延迟的阶段
- 如处理器中的某些内存单元，ALU和内存无法被划分为多个延迟较小的单元
- 一不一致的阶段延迟，使系统的吞吐量受最慢阶段的速度限制

Limitations: Register Overhead

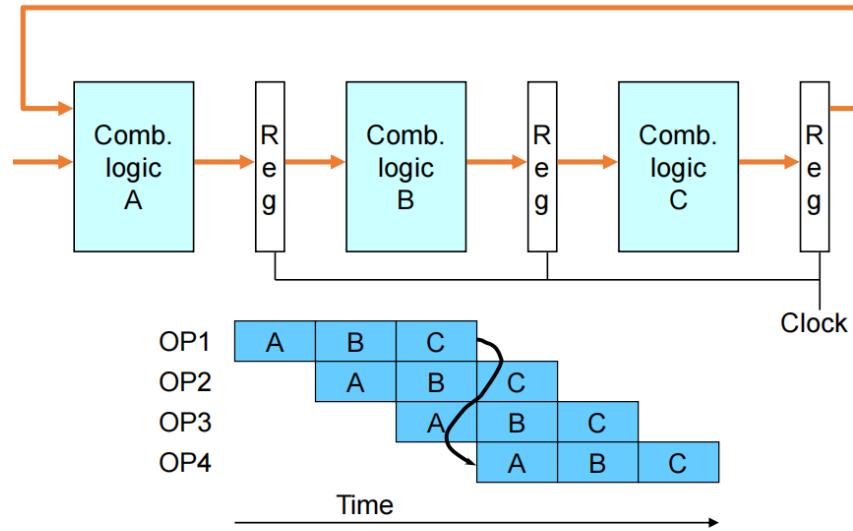
——流水线过深



- 流水线寄存器延迟限制效率

- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
 - 1-stage pipeline: 6.25%
 - 3-stage pipeline: 16.67%
 - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Instruction Dependencies in Processors —— 指令相关性



- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

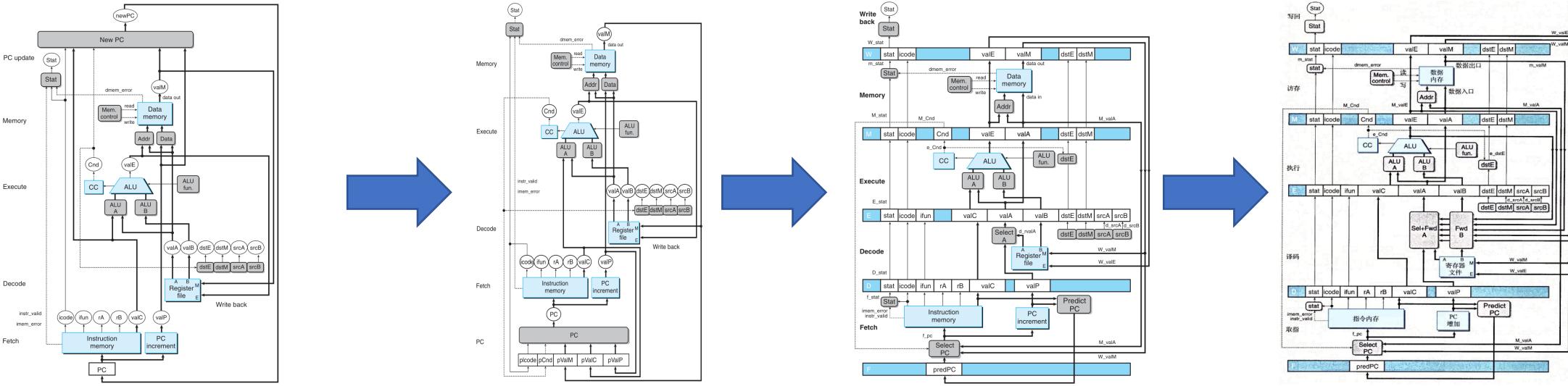
需要正确处理反馈的影响!

```
1  irmovq $50, %rax
2  addq %rax, %rbx
3  mrmovq 100(%rbx), %rdx
```

Textbook-p287

- **数据冒险**: 每条指令的结果不能即时反馈给下一条指令，在流水线的过程中，我们改变了系统的行为
- 另一种是指令控制流造成的顺序相关，称之为**控制冒险**, jxx、ret

设计流水线化的Y86-64处理器



SEQ

- 重新安排计算阶段
- 工作: 将PC计算阶段提前

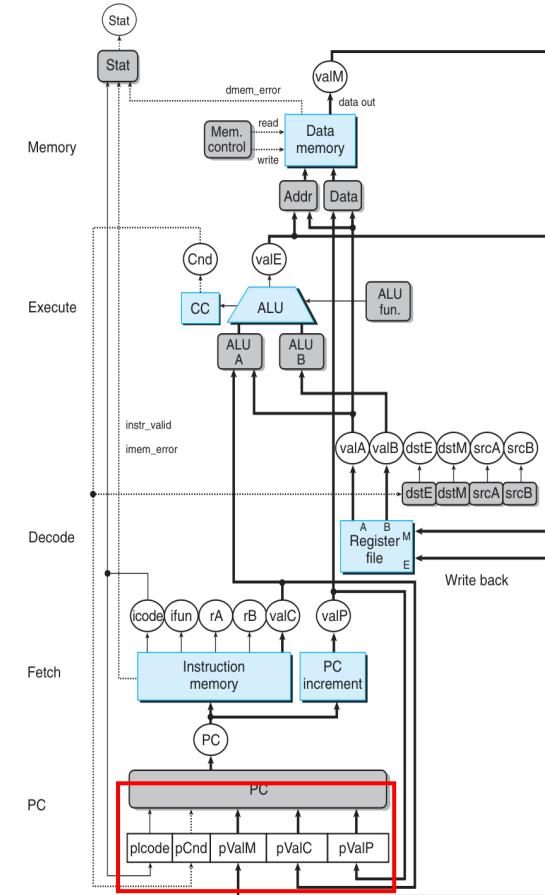
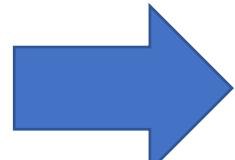
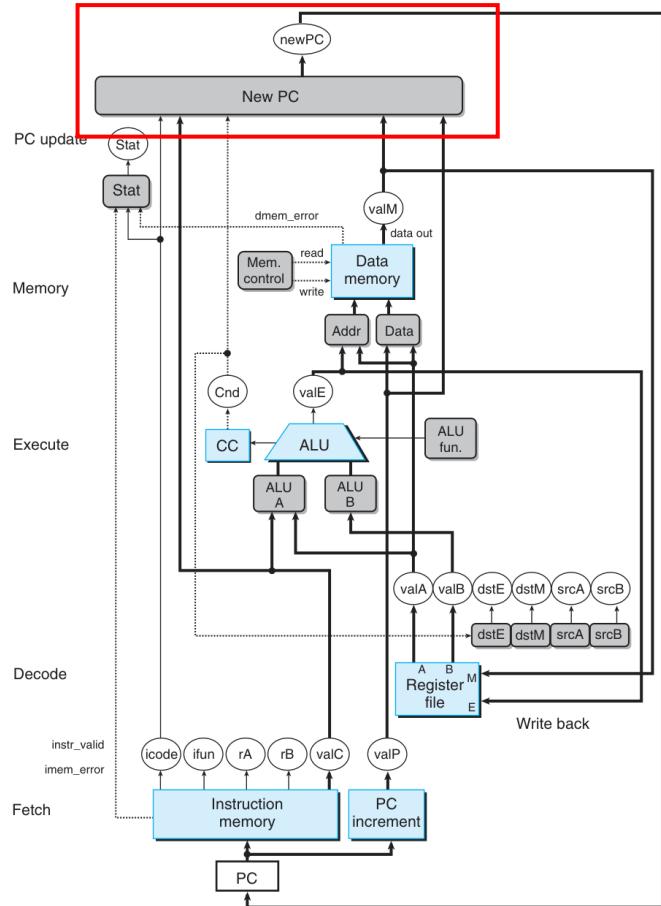
SEQ+

- 插入流水线寄存器
- 信号重新排列
- 工作: 初步建立流水线

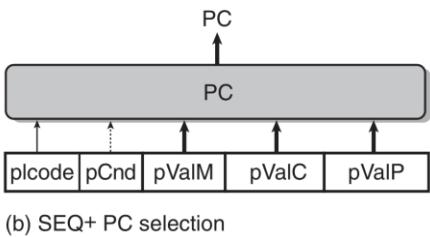
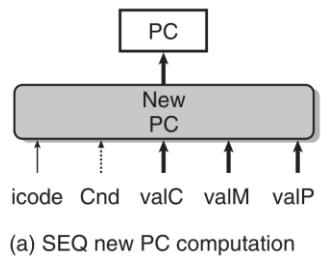
PIPE-

- 实现流水线控制逻辑
- 解决冒险、处理异常

设计流水线化的Y86-64处理器

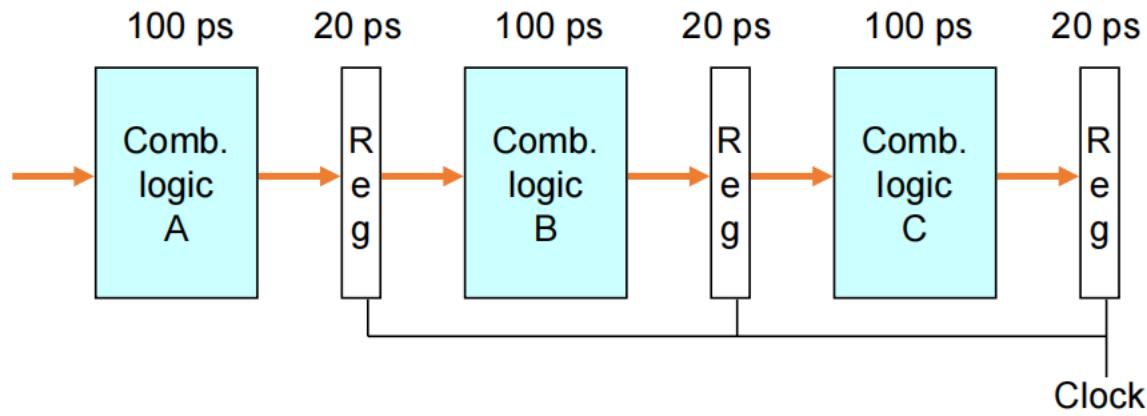


由SEQ > SEQ+：重新安排计算阶段



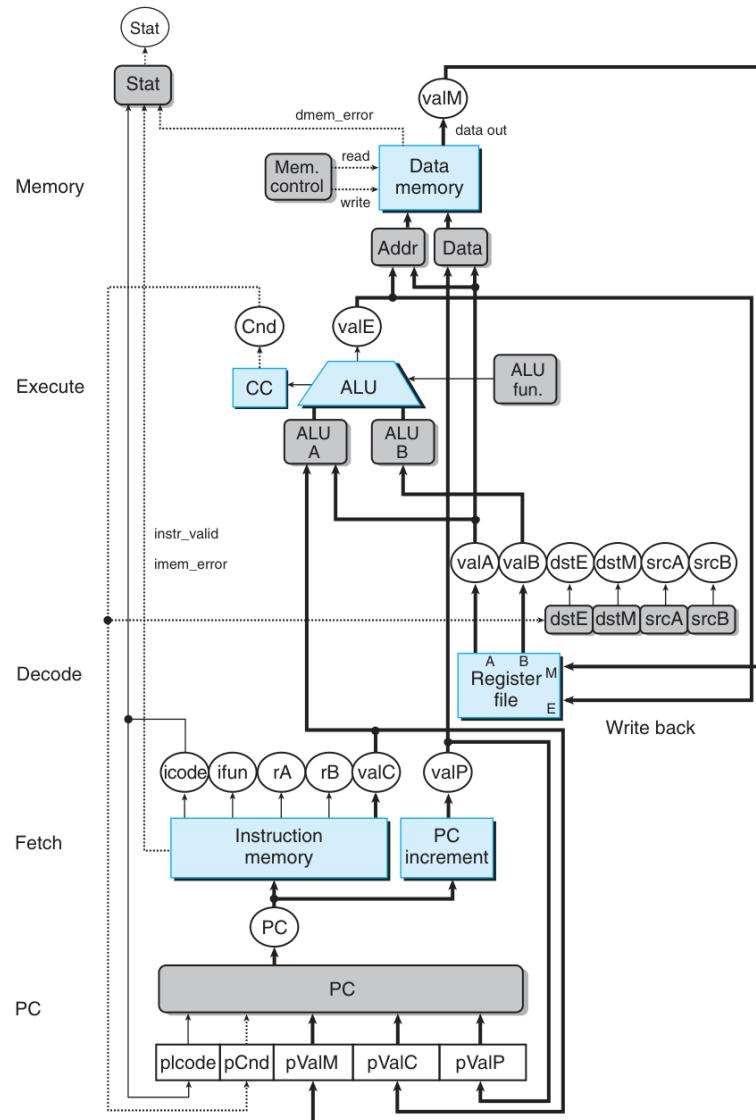
- 仍然是SEQ
- 在SEQ中，PC计算发生在时钟周期结束的时候
- SEQ+中，创建状态寄存器来保存在一条指令执行过程中计算出来的信号，当一个新的时钟周期开始时，通过同样的逻辑计算当前指令的PC
- 优点：更好地安排流水线阶段中的活动的时序

一个疑问：SEQ+的PC存储在哪里？

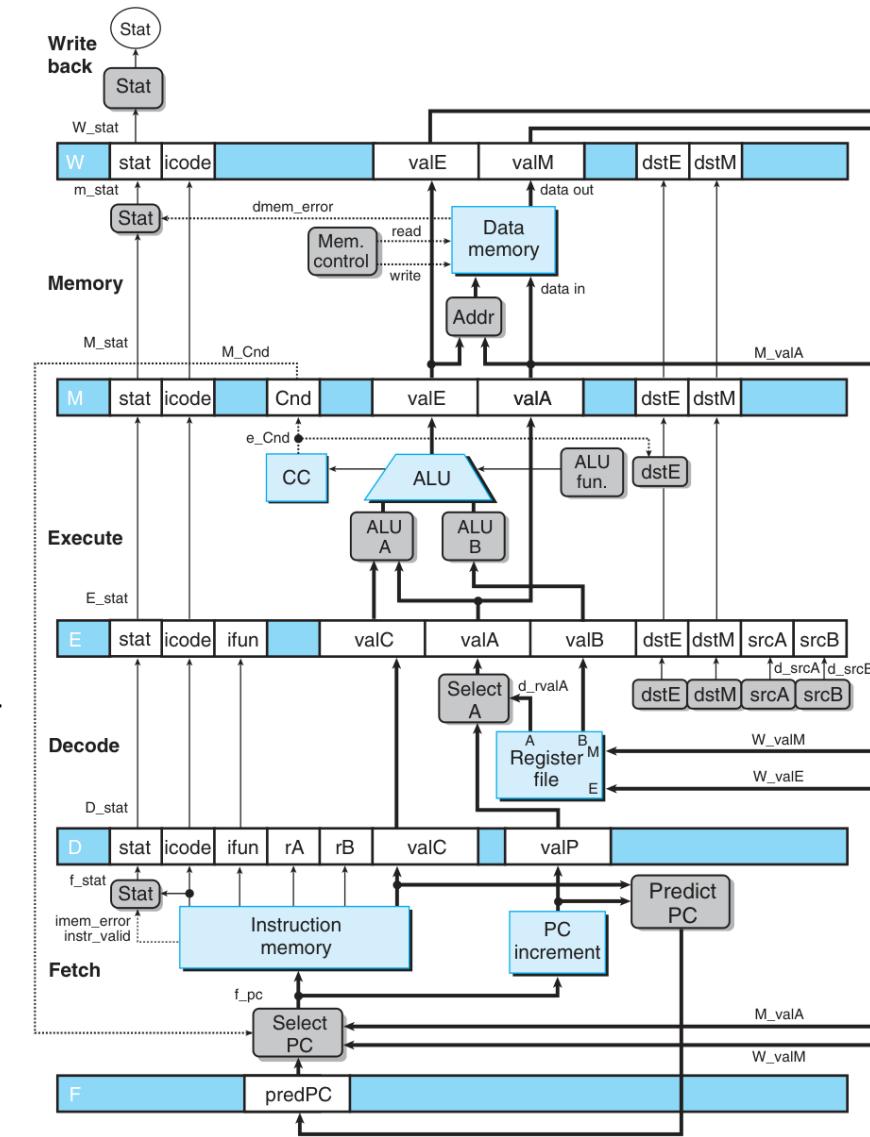


- 一般流水线示意图
 - 流水线寄存器
 - 多个指令的不同阶段同时在执行，互不干扰

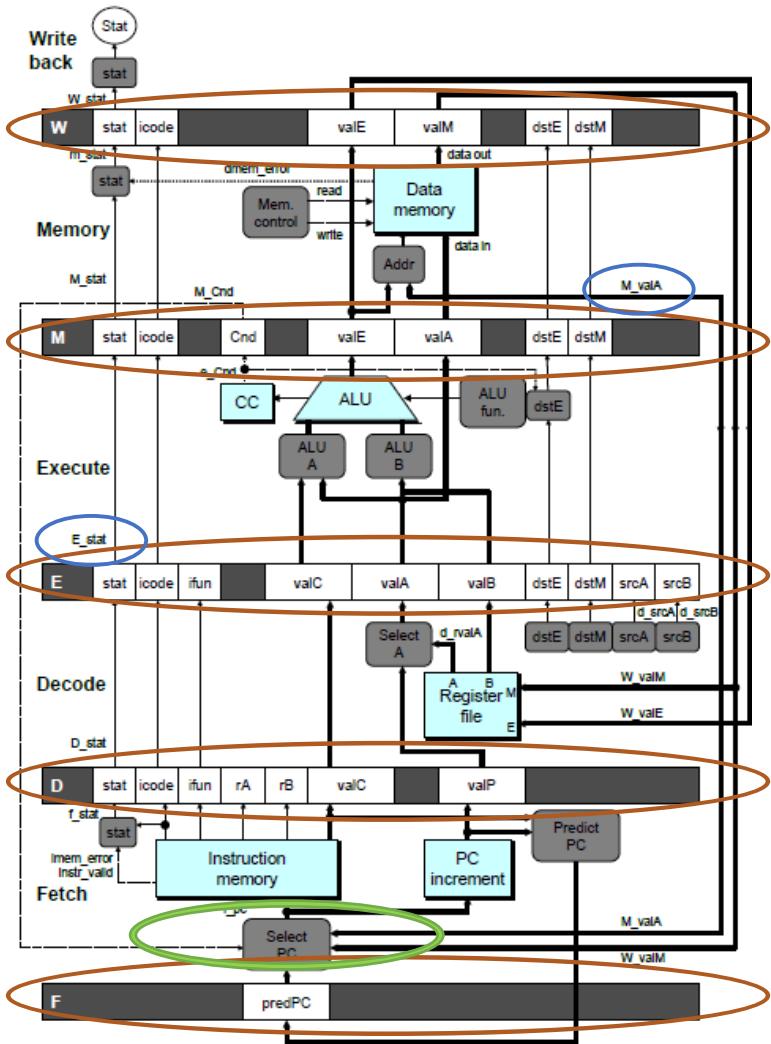
继续优化：插入流水线寄存器，将信号重新排列



SEQ+ > PIPE-



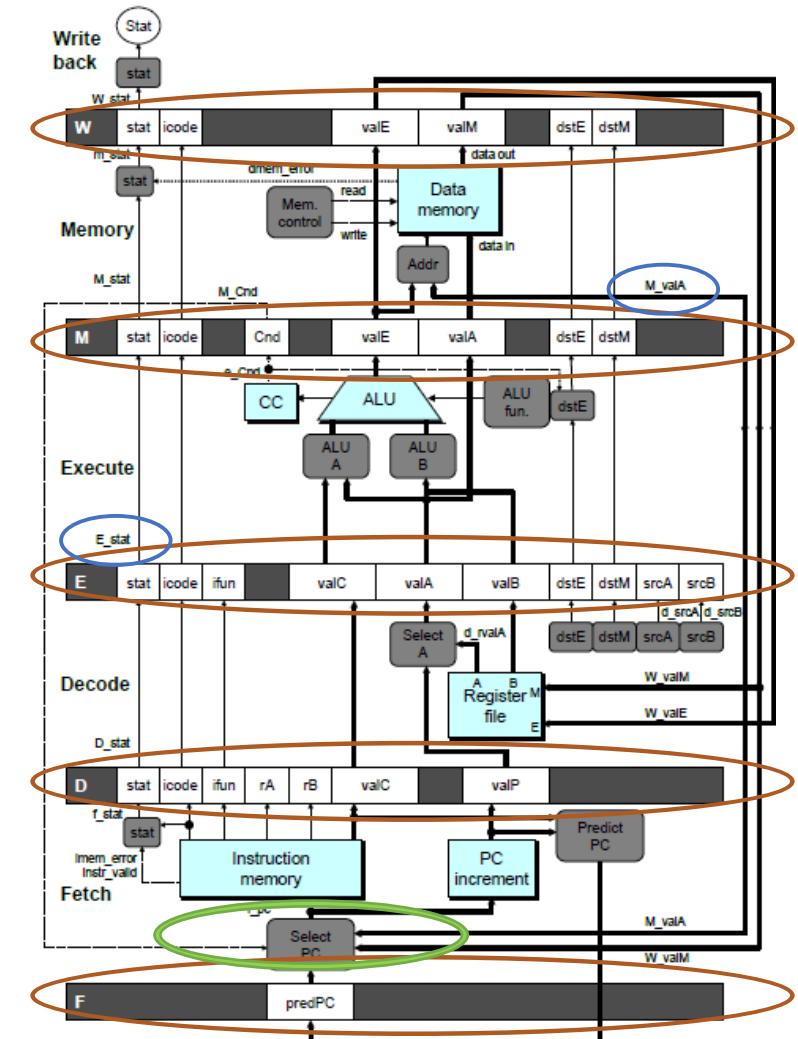
插入流水线寄存器



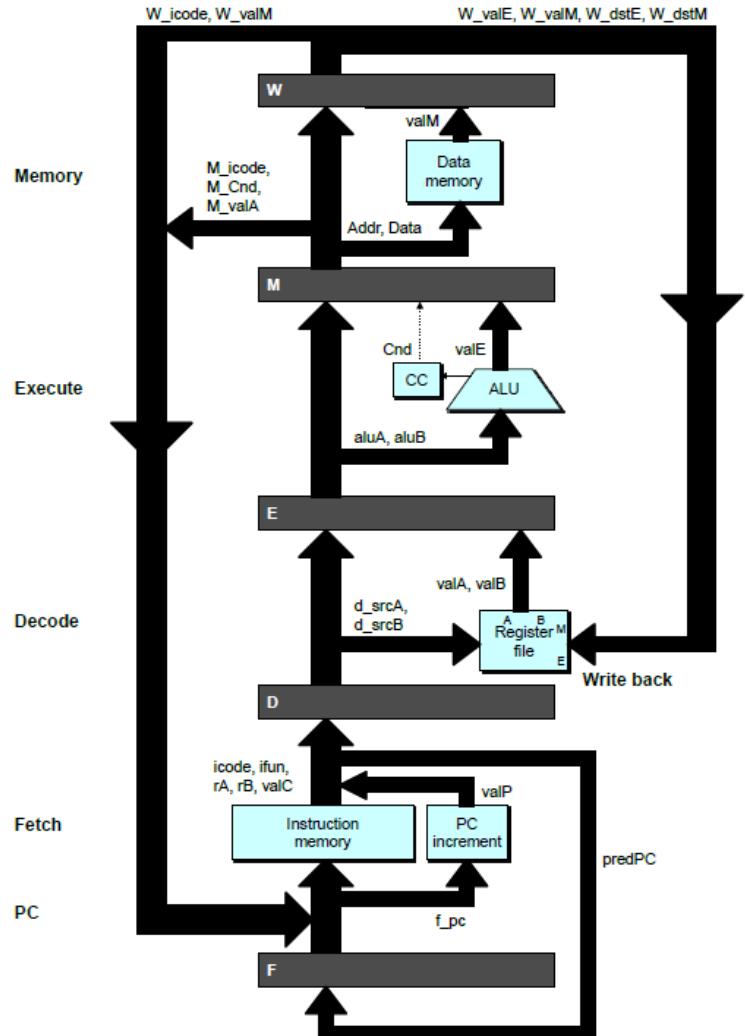
- **Fetch**
 - Select current PC
 - Read instruction
 - Compute incremented PC
- **Decode**
 - Read program registers
- **Execute**
 - Operate ALU
- **Memory**
 - Read or write data memory
- **Write Back**
 - Update register file

信号重新排列并标号

- SEQ: 一个时刻只处理一条指令，诸如valC、srcA等信号值唯一且确定
- PIPE: 同时处理多个指令，每个指令对于同一信号值有自己的版本。
 - 大写代表流水线寄存器，小写代表流水线阶段
 - 根据流水线寄存器建立命名机制，如将状态码分别命名为D_stat\E_stat\M_stat\W_stat，表示流水线寄存器的状态码字段
 - 小写前缀f、d等指流水线阶段，如m_stat指在访存阶段由控制逻辑块产生出的状态信号。
- 运行顺序：
- F—f—D—d—E—e—M—m—W—w

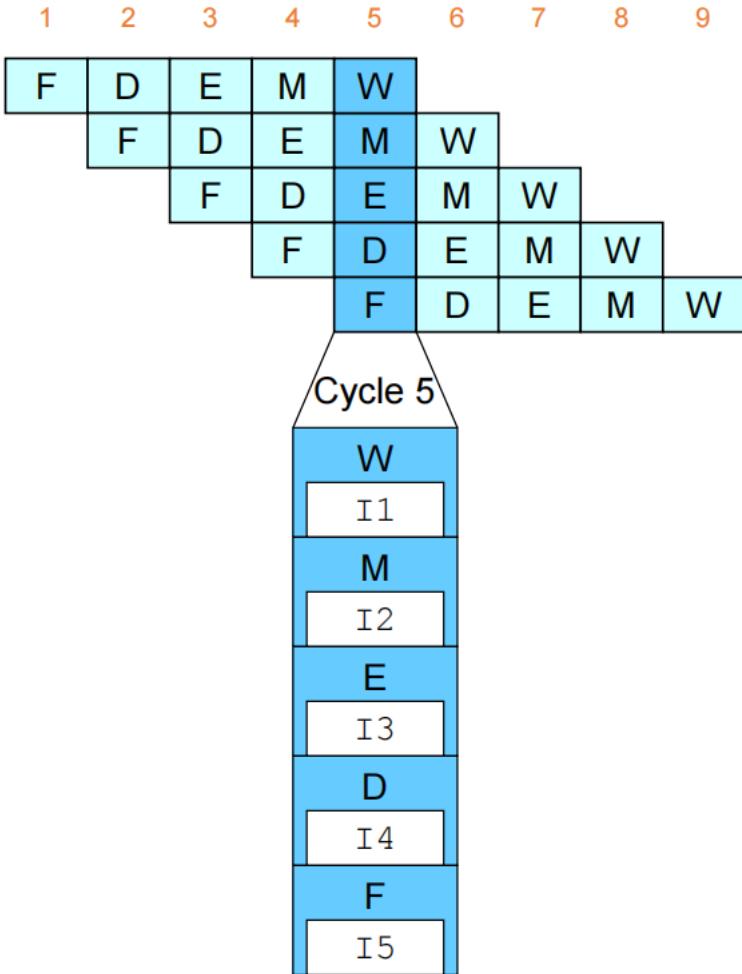


插入流水线寄存器



- 各个寄存器作用
- 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

example:



```
irmovq    $1, %rax    #I1  
irmovq    $2, %rcx    #I2  
irmovq    $3, %rdx    #I3  
irmovq    $4, %rbx    #I4  
halt          #I5
```

Textbook-p292

大致有了流水线模样，但还需要处理一些细节

- 从PIPE-到PIPE, 还需要做的工作:
- 流水线控制逻辑
 - 解决冒险
 - 处理异常
 - **加载/使用冒险（数据冒险）**：在一条从内存中读出一个值的指令和一条使用该值的指令之间，流水线必须暂停一个周期
 - **处理ret（控制冒险）**：流水线必须暂停直到ret指令到达写回阶段
 - **预测错误的分支（控制冒险）**：在分支逻辑发现不应该选择分支之前，分支目标处的几条指令已经进入流水线了，必须取消这些指令，并从跳转指令后面的那条指令开始取指
 - **异常**：当一条指令导致异常，我们想要禁止后面的指令更新程序员可见的状态，并且在异常指令到达写回阶段时，停止执行

首先解决一个问题——如何处理异常？

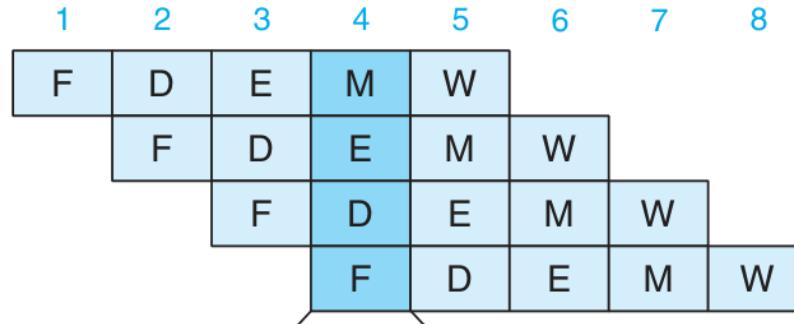
——异常处理

- 三种内部异常：
 - halt指令
 - 有非法指令和功能码组合的指令
 - 取值或数据读写试图访问一个非法地址
- 几个细节问题：
 - 多条指令引起异常——报错时深度优先
 - 选择错误分支并出现异常
 - 处理器在不同阶段更新系统状态的不同部分——如异常指令之后的某些指令同处于流水线上时改变了系统状态（条件码等）

如何解决冒险?

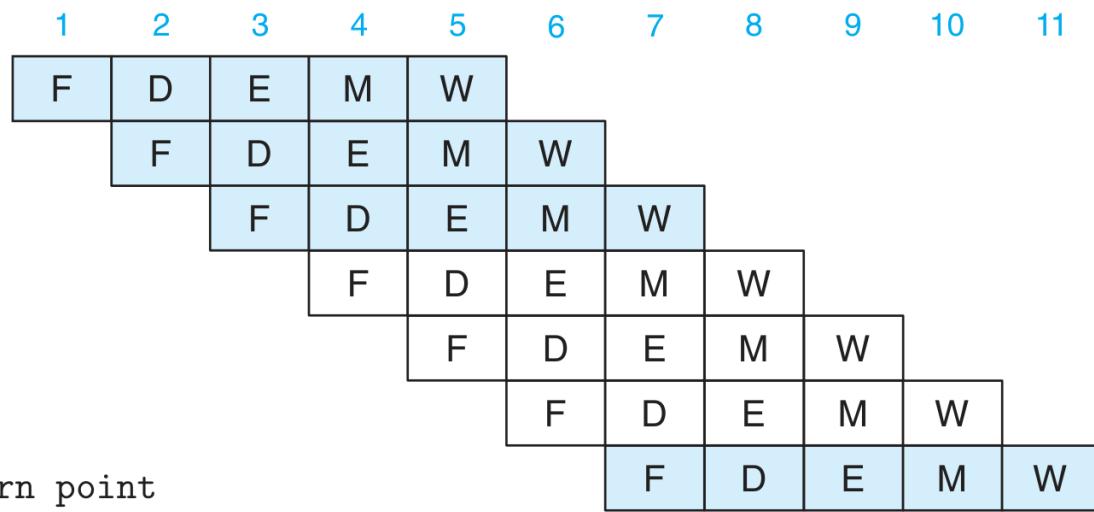
数据冒险

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



控制冒险

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

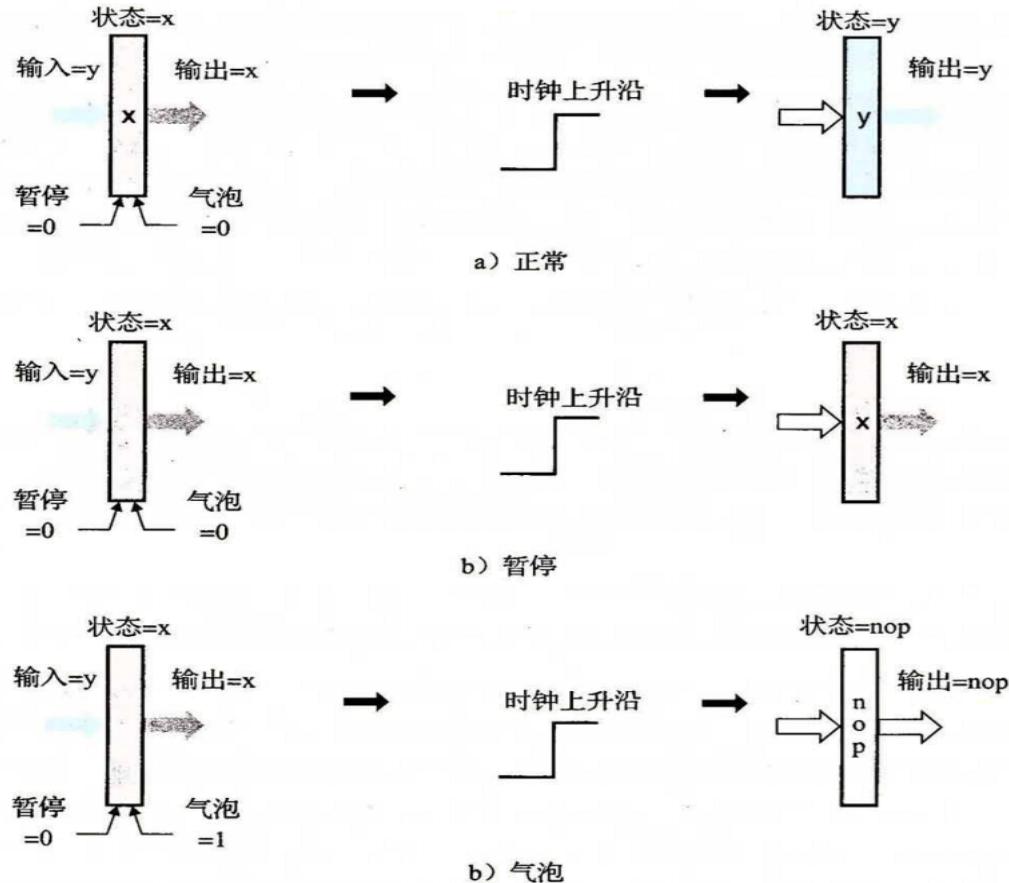


什么是bubble (气泡) ?

- 流水线控制机制

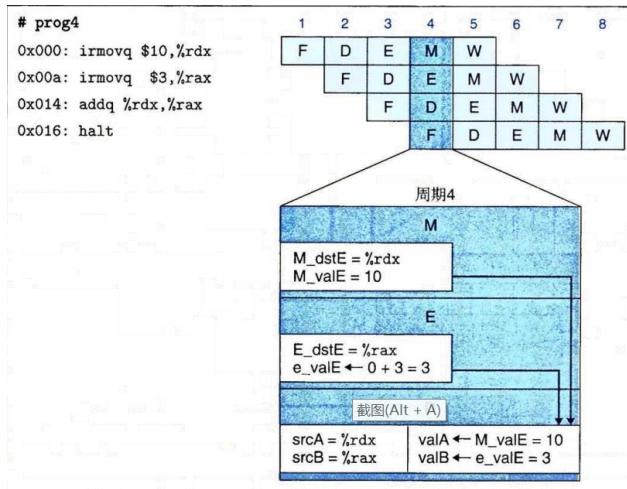
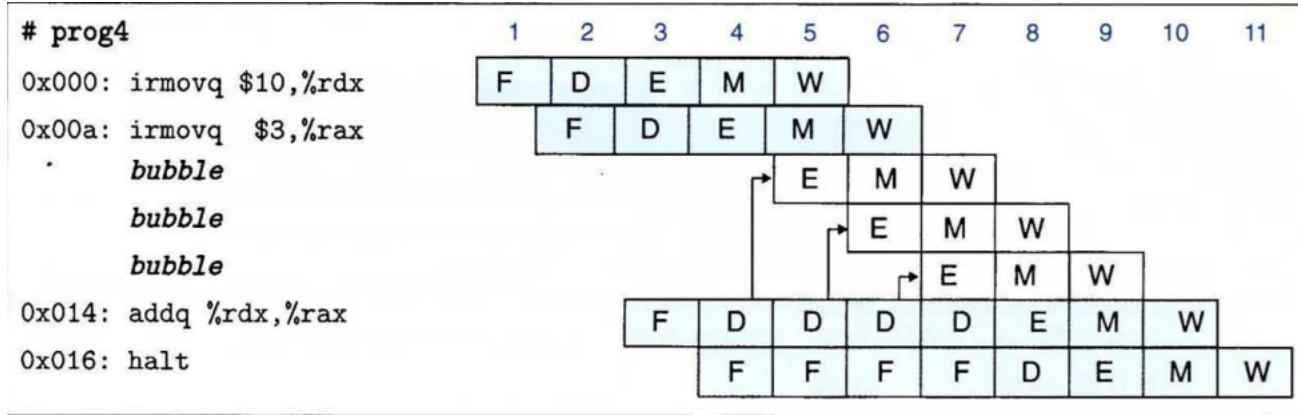
流水线控制机制

- 假设流水线寄存器有两个控制信号：
暂停(stall)和气泡(bubble)
- 暂停信号为1时，禁止更新状态
- 气泡信号为1时，用nop的操作覆盖当前状态
- 两者不同时为1



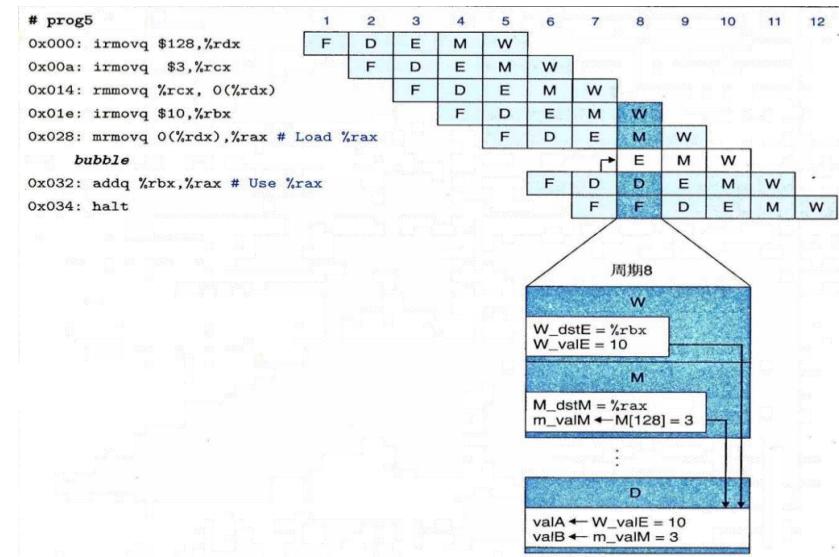
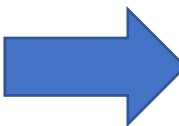
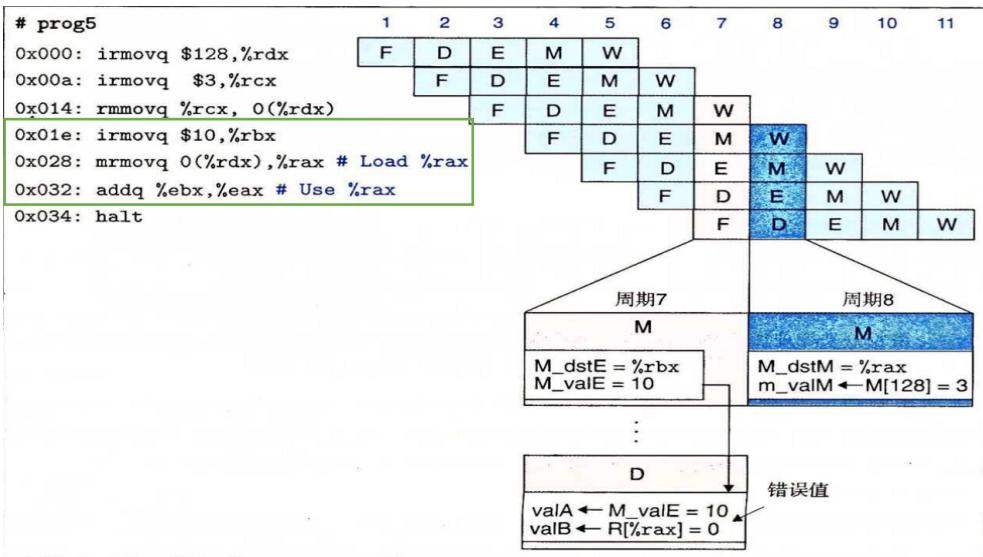
条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
加载/使用冒险	暂停	暂停	气泡	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常

数据冒险——“暂停”与转发结合



- “暂停”
- 机器通过动态地在执行阶段插入三个 bubble，得到类似三条nop指令的执行流
- 不足：使流水线暂停n个周期，降低了吞吐量，性能不佳。

- 转发
- 利用流水线寄存器
- 将访存阶段中的值 (M_valE) 作为操作数valA;
将ALU的输出 (e_valE)作为操作数valB
- 存在问题：流水线冒险——加载/使用数据



- 两条指令距离比较紧凑时
- 周期7需用到%rax，周期8才读出正确的%rax
- 转发不及时

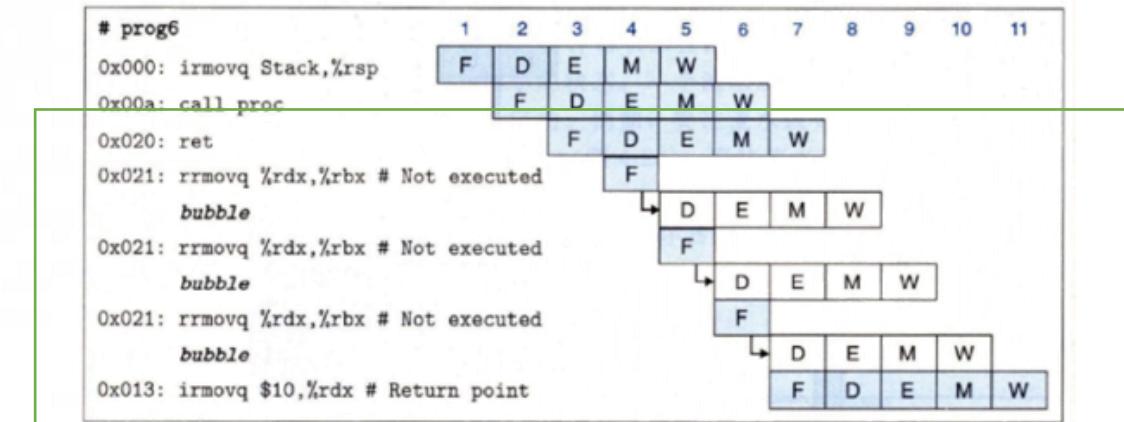
“暂停”与转发相结合

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
预测错误的分支	正常	气泡	气泡	正常	正常

控制冒险——ret

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

```
0x000:  irmovq stack,%rsp # Initialize stack pointer
0x00a:  call proc       # Procedure call
0x013:  irmovq $10,%rdx # Return point
0x01d:  halt
0x020: .pos 0x20
0x020: proc:           # proc:
0x020:   ret            # Return immediately
0x021:   rrmovq %rdx,%rbx # Not executed
0x030: .pos 0x30
0x030: stack:          # stack: Stack pointer
```



Textbook-p305

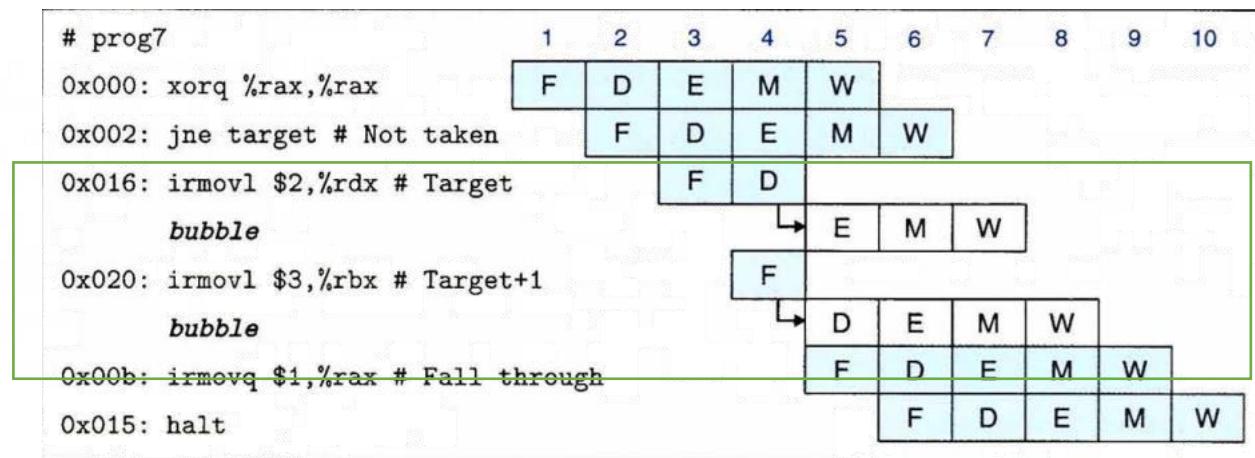
在译码、执行、访存阶段时暂停，插入三个
bubble，当ret达到写回阶段时，PC选择返回地
址作为指令的取指地址

控制冒险——jxx

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

跳转出现错误

```
0x000: xorq %rax,%rax
0x002: jne target      # Not taken
0x00b: irmovq $1, %rax # Fall through
0x015: halt
0x016: target:
0x016: irmovq $2, %rdx # Target
0x020: irmovq $3, %rbx # Target+1
0x02a: halt
```



Textbook-p306

在周期4，已经取出来两条错误的指令，此时程序员可见的状态尚未发生变化，在周期5的译码和执行阶段插入气泡，同时取出正确的指令，从而取消预测错误的指令（又称指令排除）。

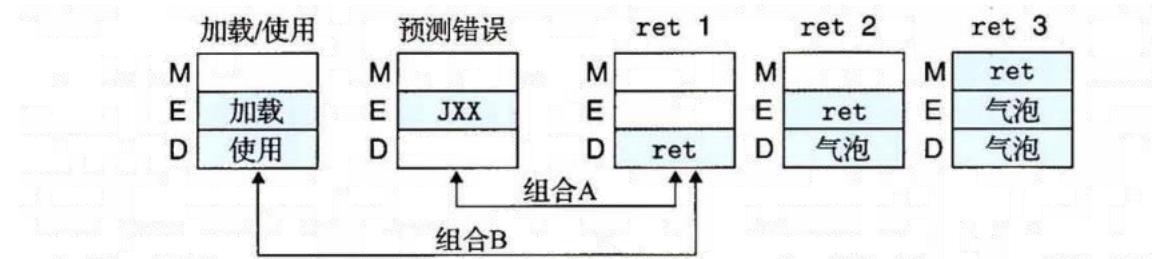
- 机器如何知道需要处理特殊情况

条件	触发条件
处理 ret	$IRET \in \{D_icode, E_icode, M_icode\}$
加载/使用冒险	$E_icode \in \{IMRMOVL, IPOPL\} \quad \& \quad E_dstM \in \{d_srcA, d_srcB\}$
预测错误的分支	$E_icode = IJXX \quad \& \quad ! e_Cnd$
异常	$m_stat \in \{SADR, SINS, SHLT\} \quad \quad W_stat \in \{SADR, SINS, SHLT\}$

- 到达时钟周期末尾时， d_srcA 和 d_srcB 会设置成译码中指令的源操作数的寄存器ID
- 当跳转指令时，信号 e_Cnd 指明是否要选择分支

- 若是在一个时钟周期内出现多个特殊情况

- 控制条件的组合
 - 去除无法同时出现的情况
 - 考虑A、B两种情况



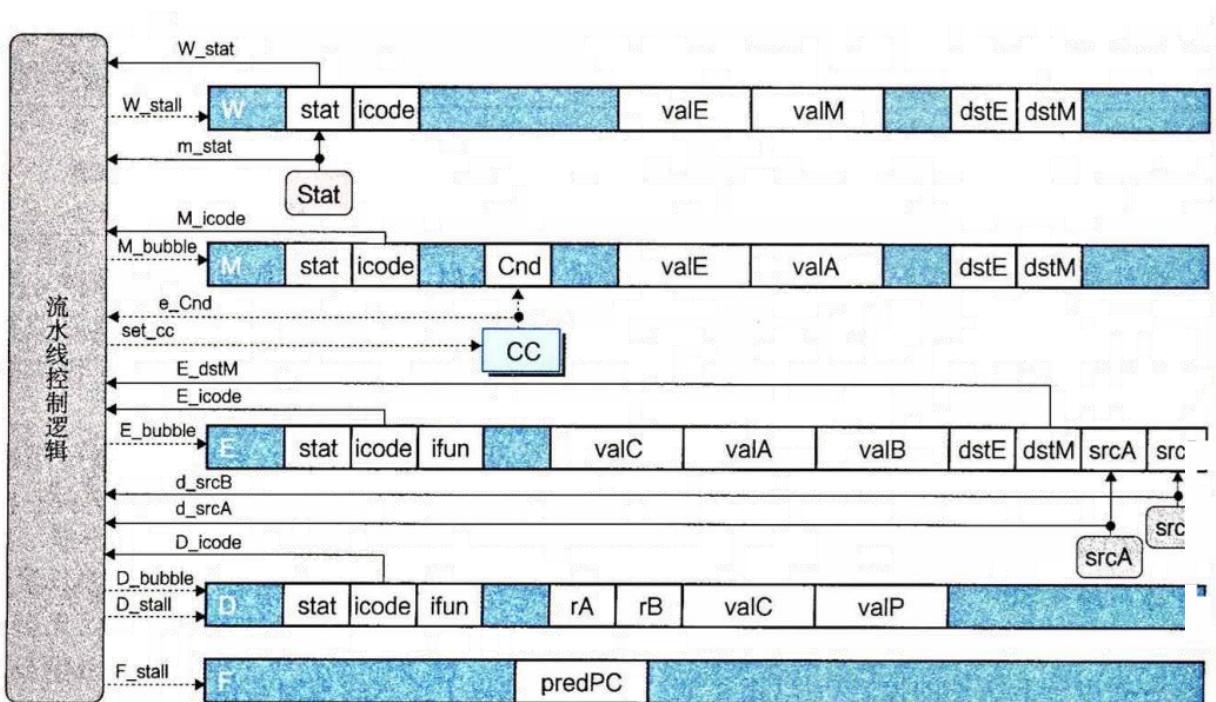
A:

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	暂停	气泡	气泡	正常	正常

B: 需特殊处理

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	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	暂停	暂停	气泡	正常	正常

PIPE控制逻辑实现



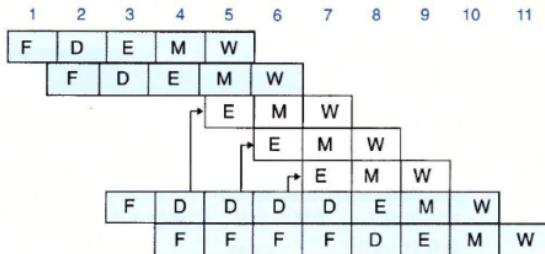
```
bool F_stall =  
    # Conditions for a load/use hazard  
    E_icode in { IMRMOVQ, IPOPQ } &&  
    E_dstM in { d_srcA, d_srcB } ||  
    # Stalling at fetch while ret passes through pipeline  
    IRET in { D_icode, E_icode, M_icode };  
  
bool D_bubble =  
    # Mispredicted branch  
    (E_icode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

重点题型B:流水线冒险和避免机制(无转发下通常要加3个气泡)

1.暂停避免数据冒险

解决方案: 转发避免数据冒险, 需要5个转发源(e_valE, M_valE, W_valE, m_valM, W_valM)和2个转发目的(ValA, Val B)

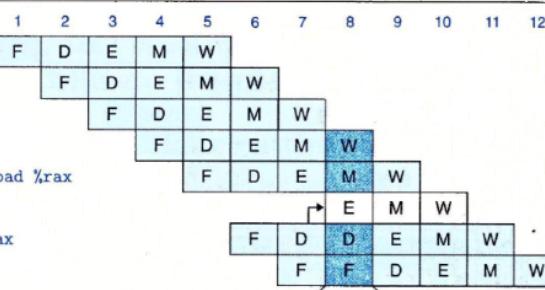
```
# prog4
1 2 3 4 5 6 7 8 9 10 11
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
    bubble
    bubble
    bubble
0x014: addq %rdx,%rax
0x016: halt
```



2.加载/使用冒险(属于数据冒险)

解决方案: 加载互锁,1个气泡

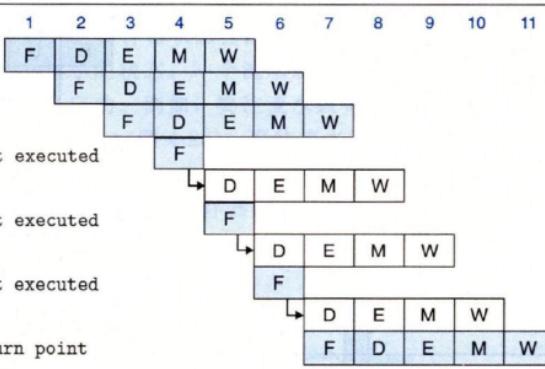
```
# prog5
1 2 3 4 5 6 7 8 9 10 11 12
0x000: irmovq $128,%rdx
0x00a: irmovq $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
    bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```



3.处理ret(属于控制冒险)

解决方案: 3个气泡

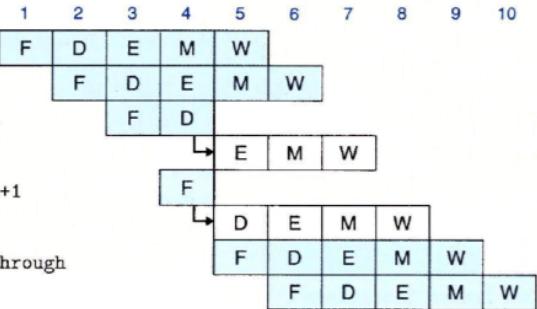
```
# prog6
1 2 3 4 5 6 7 8 9 10 11
0x000: irmovq Stack,%rsp
0x00a: call proc
0x020: ret
0x021: rrmovq %rdx,%rbx # Not executed
    bubble
0x021: rrmovq %rdx,%rbx # Not executed
    bubble
0x021: rrmovq %rdx,%rbx # Not executed
    bubble
0x013: irmovq $10,%rdx # Return point
```



4.预测错误的分支(属于控制冒险)

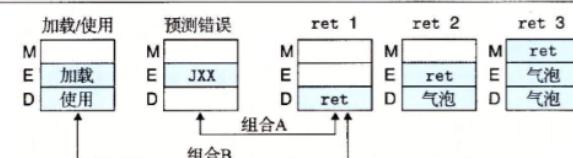
解决方案: 2个气泡

```
# prog7
1 2 3 4 5 6 7 8 9 10
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
```



5.小结和多种冒险的组合

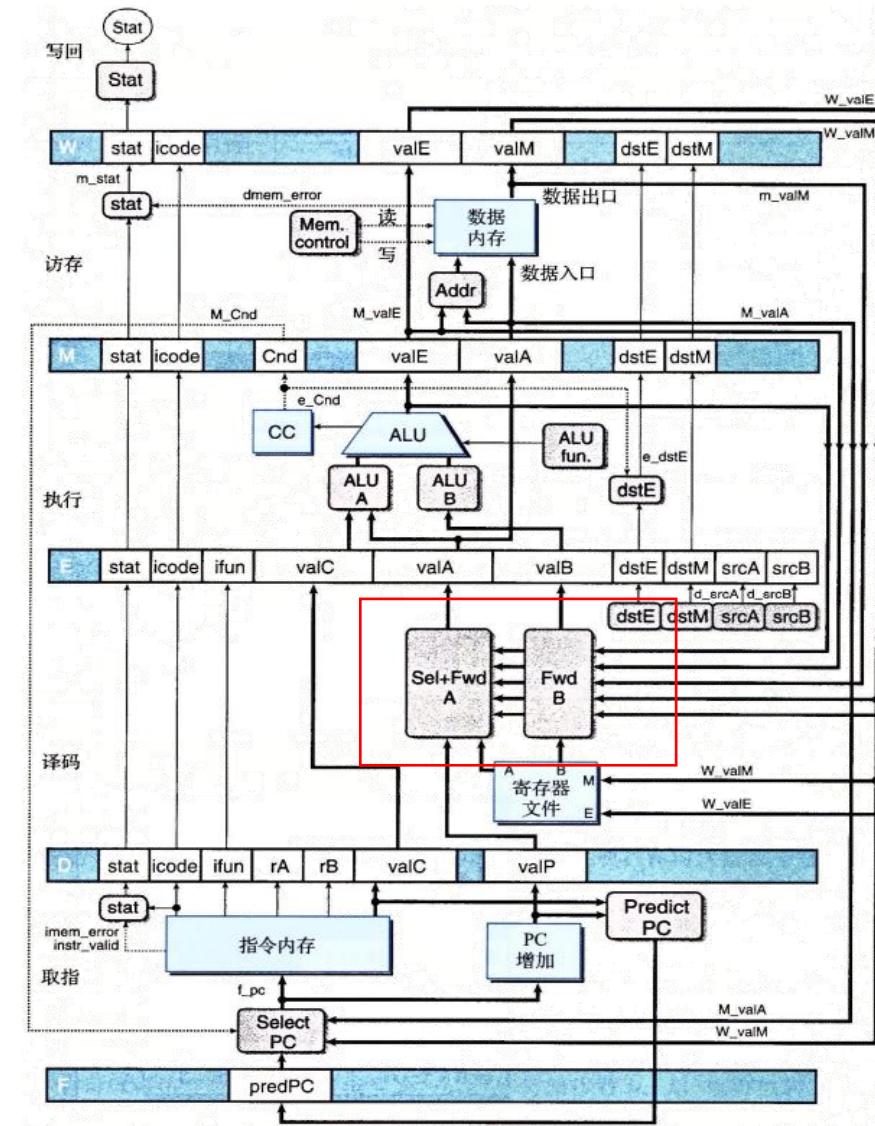
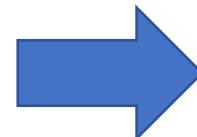
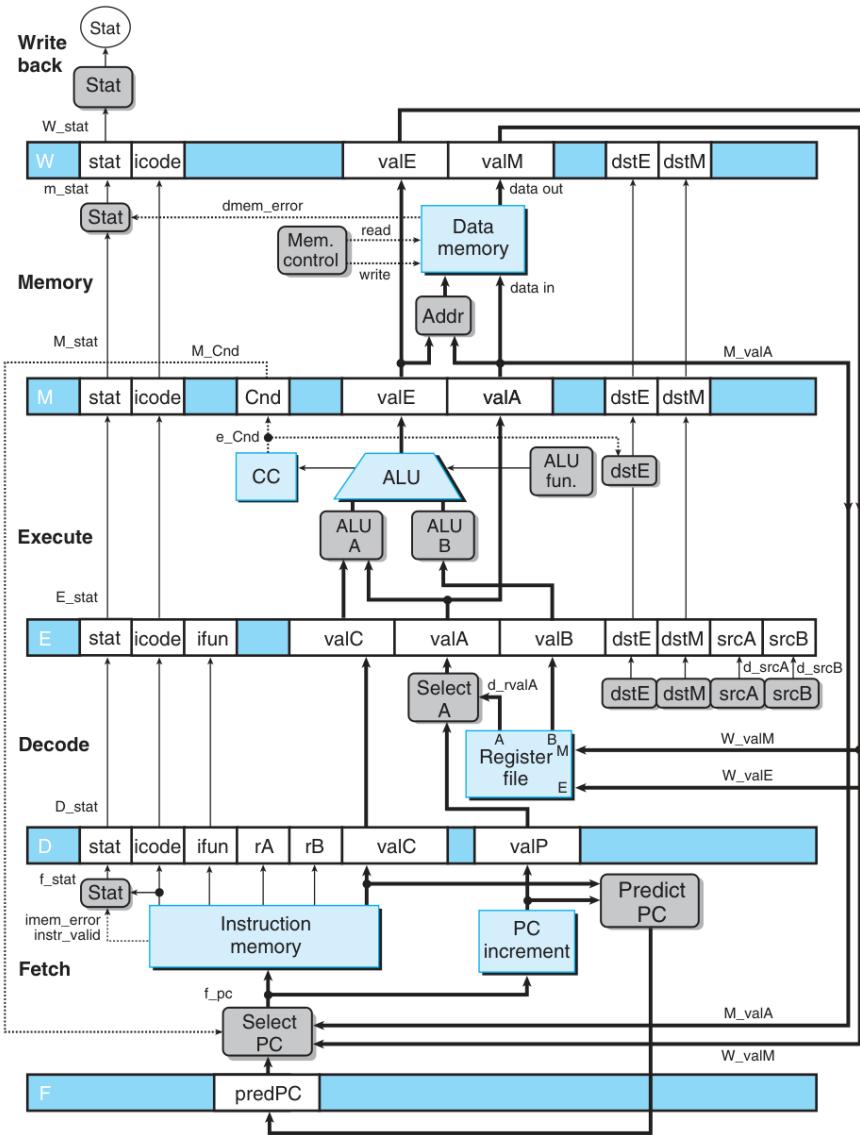
条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
加载/使用冒险	暂停	暂停	气泡	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常



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	暂停	气泡	气泡	正常	正常

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	暂停	暂停	气泡	正常	正常

PIPE- > PIPE



PIPE各阶段

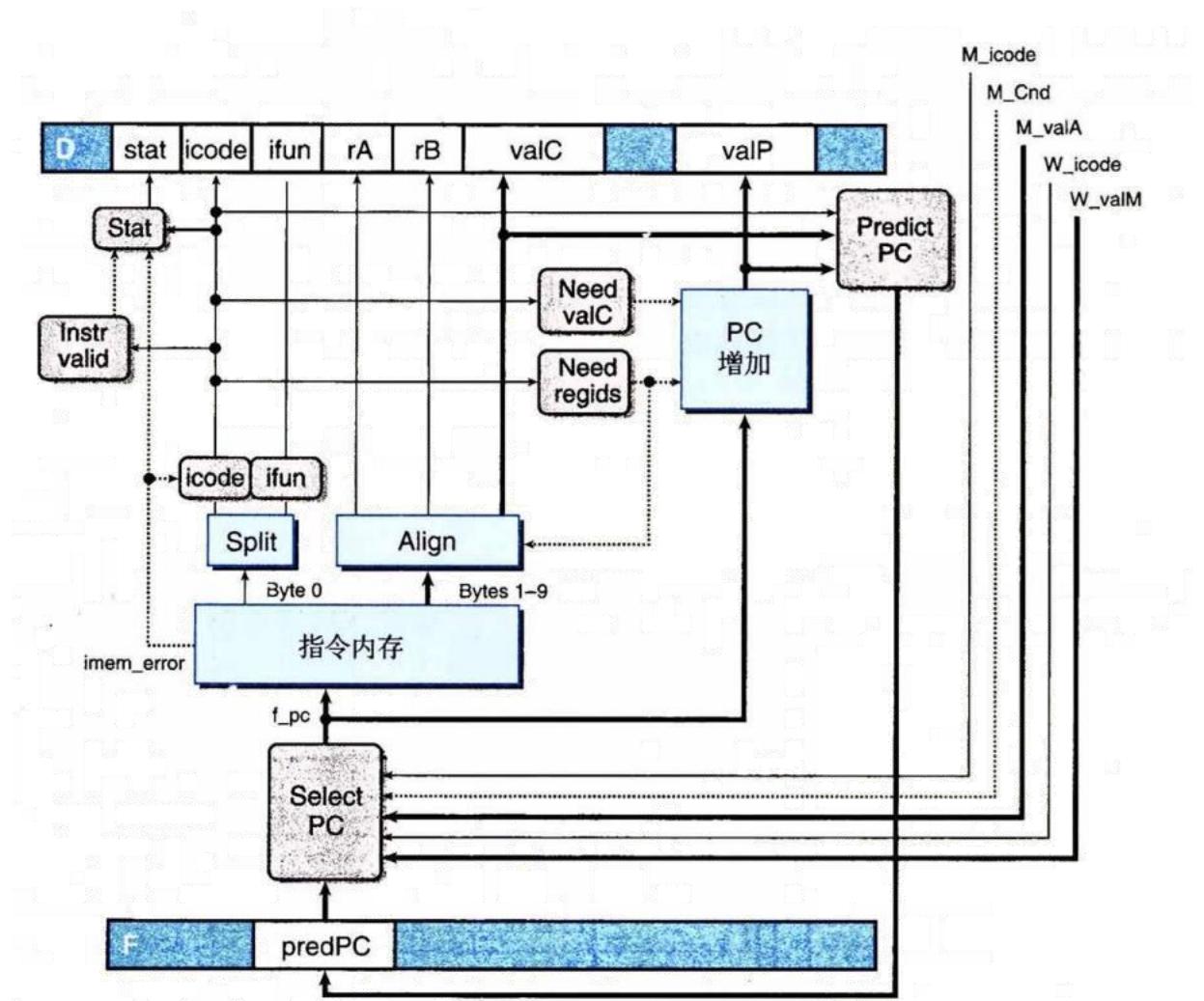
- PC选择和取指阶段 (Fetch)
 - 选择PC当前值，并且预测下一个PC值

```
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;
];
```

选择逻辑

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

预测逻辑

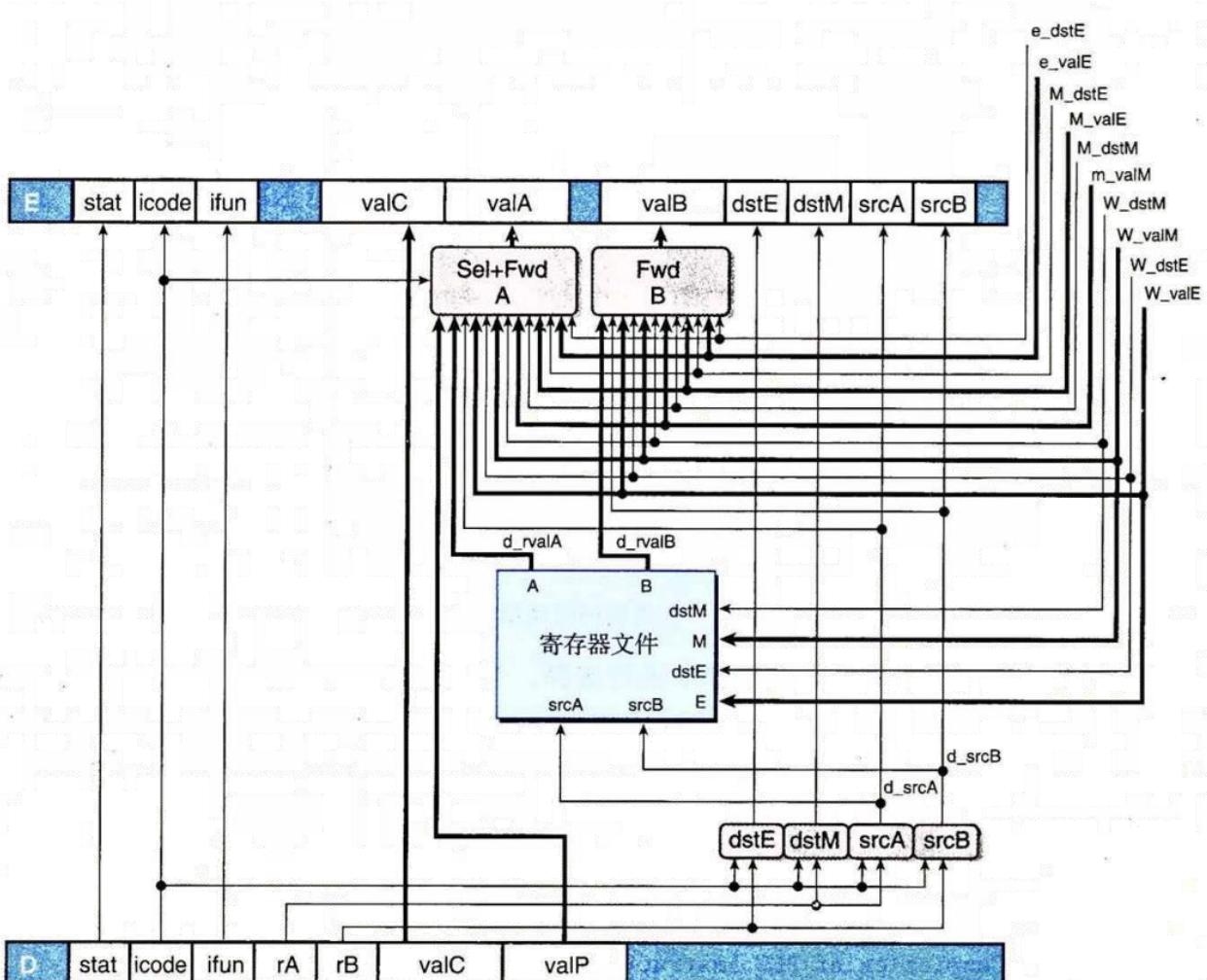


PIPE各阶段

- 译码和写回阶段 (Decode & Write back)
 - 复杂性主要与转发逻辑有关
 - 合并valP与valA信号，实现valA的转发逻辑

```

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
];
  
```

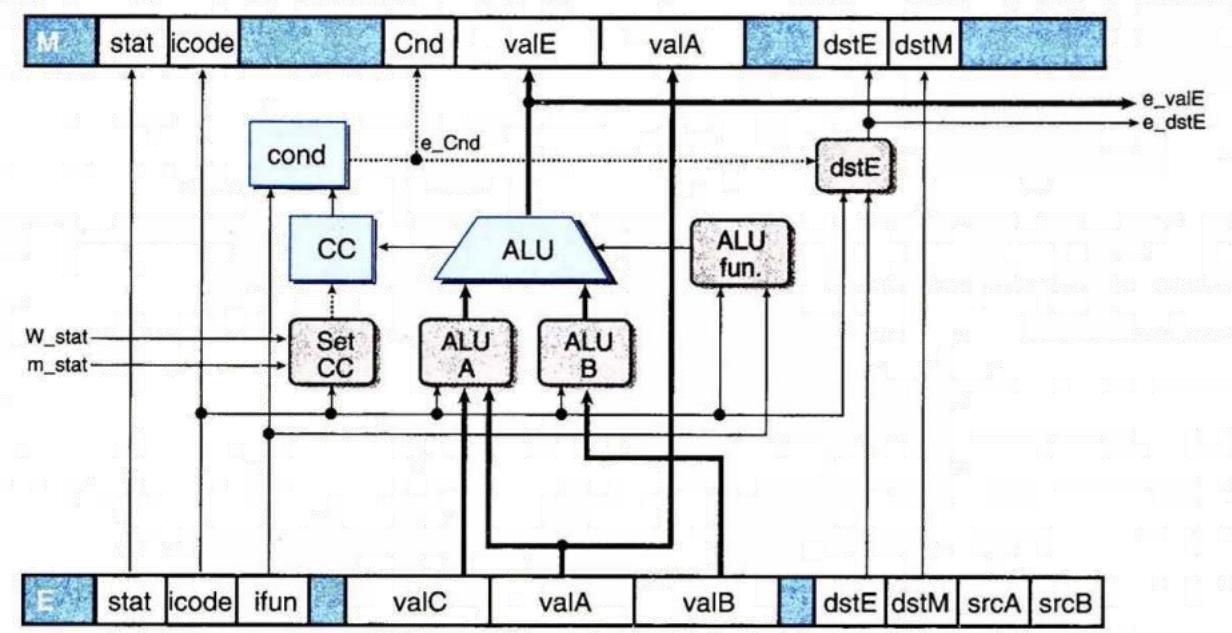


- 转发源具有不同的优先级
- 状态码设置，写回阶段有气泡时也是AOK

数据字	寄存器 ID	源描述
e_valE	e_dstE	ALU 输出
m_valM	M_dstM	内存输出
M_valE	M_dstE	访存阶段中对端口 E 未进行的写
W_valM	W_dstM	写回阶段中对端口 M 未进行的写
W_valE	W_dstE	写回阶段中对端口 E 未进行的写

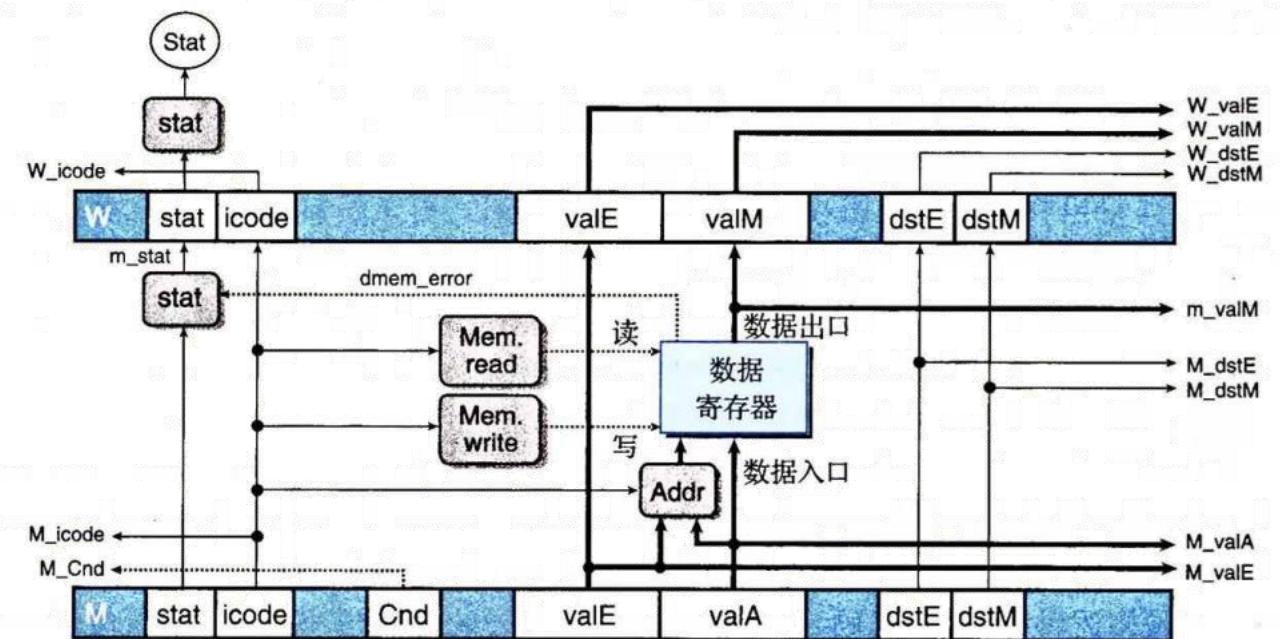
PIPE各阶段

- 执行阶段 (Execute)
 - 与SEQ十分相似
 - e_valE 与 e_dstE 作为转发源，指向访存阶段
 - Set CC以 m_stat 以及 W_stat 作为输入，以决定是否要更新条件码



PIPE各阶段

- 访存阶段 (Memory)
- 无Data阶段，由Sel+Fwd进行选择
- M与W信号作为转发和**流水线控制逻辑**的一部分，提供给其他部分



性能分析

$$CPI = \frac{C_i + C_b}{C_s} = 1.0 + \frac{C_b}{C_s}$$

- CPI: 每指令周期数

$$CPI = 1.0 + lp + mp + rp$$

- lp : 加载/使用冒险造成暂停时插入气泡的平均数
- mp : 由于预测错误取消指令时插入气泡的平均数
- rp : 是当由于ret指令造成暂停时插入气泡的平均数
- 超标量操作: 并行地取指、译码和执行, 使得CPI小于1.0
- $IPC = 1/CPI$

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

未竟事宜

- 与存储系统的接口
 - Cache & TLB
 - 高速缓存不命中
 - 缺页
- 多周期指令
 - 整数乘法、除法、浮点运算
 - 独立于主流水线的单元，也是流水线化的
 - 同步化

Practice

The End