

Processor Arch

13 元培数科 常欣海

Here we go! →

2024/10/23



Knowledge Review

前言

before we start

- 本章内容极多，需要至少仔细阅读 CS:APP 两遍
- 对于 SEQ、PIPE 的实现、线是怎么连接的，信号是怎么产生、在什么时候产生的，都需要完全理解、背诵
- 对于冲突的解决，也需要完全理解、背诵
- 参考资料： CMU / HCL Descriptions of Y86-64 Processors，Y86-64 指令集，HCL 完整版，第四章 Arch 复习必备
- 建议大家多开一个 <https://slide.huh.moe/05/> 方便听课时回翻。
- ~~建议变身医学性，全背就完了。符号很多，推荐理解性记忆。~~
- 本次备课花了我大量时间，希望大家好好听讲。
- 看书！看书！看书！

小班回课给分相关

score

考虑到某些同学会想要内卷（虽然我不太鼓励大家卷这个，卷考试会更香，但确实小班给分会有优秀率限制），所以明确一下我的评分标准：

1. 我不太会给同学们太低的分，除非你写的实在过于草率
2. 我希望回课的同学至少能够认真掌握自己回课的部分
3. 为了大家的理解，以及我的身心健康，我希望大家不要大片 copy 大班 PPT 或者书（这部分内容可以有，但必然和我本来就要有的内容会相同很多），更多的给出一些像我一样的便于理解的 tips、一两句话说明一个精髓的点、某些看完书不容易关注的犄角旮旯的考试知识点啥的这些对大伙更实用的东西，具体可以参考我已经公布的我制作的 Slide

Y86-64 的顺序实现

sequential implementation

处理一条指令通常包含以下几个阶段：

1. 取指 (Fetch)
2. 译码 (Decode)
3. 执行 (Execute)
4. 访存 (Memory)
5. 写回 (Write Back)
6. 更新PC (PC Update)

Y86-64 的顺序实现

sequential implementation

1. 取指 (Fetch)

操作：取指阶段从内存中读取指令字节，地址由程序计数器 (PC) 的值决定。

读出的指令由如下几个部分组成：

- icode：指令代码，指示指令类型，是指令字节的低 4 位
- ifun：指令功能，指示指令的子操作类型，是指令字节的高 4 位（不指定时为 0）
- rA：第一个源操作数寄存器（可选）
- rB：第二个源操作数寄存器（可选）
- valC：常数，Constant（可选）

各个不同名称的指令一般具有不同的 icode，但是也有可能共享相同的 icode，然后通过 ifun 区分。

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						

Y86-64 的顺序实现

sequential implementation

1. 取指 (Fetch)

操作：取指阶段从内存中读取指令字节，地址由程序计数器 (PC) 的值决定。

- `ifun` 在除指令为 `OPq`, `jXX` 或 `cmovXX` 其中之一时都为 0
- `rA`, `rB` 为寄存器的编码，取值为 0 到 F，每个编码对应着一个寄存器。注意当编码为 F 时代表无寄存器。
- `rA`, `rB` 并不是每条指令都有的，`jXX`, `call` 和 `ret` 就没有 `rA` 和 `rB`，这在 HCL 中通过 `need_regids` 来控制
- `valC` 为 8 字节常数，可能代表立即数 (`irmovq`)，偏移量 (`rmmovq` `mrmovq`) 或地址 (`call` `jmp`)。`valC` 也不是每条指令都有的，这在 HCL 中通过 `need_valC` 来控制

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB	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						

Y86-64 的顺序实现

sequential implementation

2. 译码 (Decode)

操作：译码阶段从寄存器文件读取操作数，得到 `valA` 和 / 或 `valB`。

一般根据上一阶段得到的 `rA` 和 `rB` 来确定需要读取的寄存器。

也有部分指令会读取 `rsp` 寄存器 (`popq` `pushq` `ret` `call`)。

Y86-64 的顺序实现

sequential implementation

3. 执行 (Execute)

操作：执行阶段，算术/逻辑单元 (ALU) 进行运算，包括如下情况：

- 执行指令指明的操作 (opq)
- 计算内存引用的地址 (rmmovq mrmovq)
- 增加/减少栈指针 (pushq popq) 其中加数可以是 +8 或 -8

最终，我们把此阶段得到的值称为 valE (Execute stage value)。

一般来讲，这里使用的运算为加法运算，除非是在 OPq 指令中通过 ifun 指定为其他运算。这个阶段还会：

设置条件码 (OPq) :

检查条件码和和传送条件 (jXX cmovXX) :

set CC

Cnd \leftarrow Cond(CC, ifun)

Y86-64 的顺序实现

sequential implementation

4. 访存 (Memory)

操作: 访存阶段可以将数据写入内存 (`rmmovq` `pushq` `call`) , 或从内存读取数据 (`mrmovq` `popq` `ret`)

- 若是向内存写, 则:
 - 写入的地址为 `valE` (需要计算得到, `rmmovq` `pushq` `call`)
 - 数据为 `valA` (`rmmovq` `pushq`) 或 `valP` (`call`)
- 若是从内存读, 则:
 - 地址为 `valA` (`popq` `ret`, 此时 `valB` 用于计算更新后的 `%rsp`) 或者 `valE` (需要计算得到, `mrmovq`)
 - 读出的值为 `valM` (Memory stage value)

Y86-64 的顺序实现

sequential implementation

5. 写回 (Write Back)

操作：写回阶段最多可以写 **两个** 结果到寄存器文件（即更新寄存器）。

Y86-64 的顺序实现

sequential implementation

6. 更新PC (PC Update)

操作: 将 PC 更新成下一条指令的地址 `new_pc`。

- 对于 `call` 指令, `new_pc` 是 `valC`
- 对于 `jxx` 指令, `new_pc` 是 `valC` 或 `valP`, 取决于条件码
- 对于 `ret` 指令, `new_pc` 是 `valM`
- 其他情况, `new_pc` 是 `valP`

Y86-64 的顺序实现

sequential implementation

的确有直接传 `valA` 到 `M` 的，但那一般是 `valE` 算别的去了（`rmmovq` `pushq` `popq`）。也可以理解为想要 `rmmovq` 和 `irmovq` 更统一一些所以这么设计。

这里的表中没有写出 `cmoveXX`，因为其与 `rmmovq` 共用同一个 `icode`，然后通过 `ifun` 区分。注意 `OPq` 的顺序，是 `valB OP valA`。

Stage	<code>OPq rA, rB</code>	<code>rmmovq rA, rB</code>	<code>irmovq V, rB</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valA} \leftarrow R[\text{rA}]$	
Execute	$\text{valE} \leftarrow \text{valB OP valA}$ Set CC	$\text{valE} \leftarrow 0 + \text{valA}$	$\text{valE} \leftarrow 0 + \text{valC}$
Memory			
Write back	$R[\text{rB}] \leftarrow \text{valE}$	$R[\text{rB}] \leftarrow \text{valE}$	$R[\text{rB}] \leftarrow \text{valE}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

Y86-64 的顺序实现

sequential implementation

`valC` 被当做偏移量使用，与
`valB` 相加得到 `valE`，然后
`valE` 被当做地址使用。

Stage	<code>rmmovq rA, D(rB)</code>	<code>mrmovq D(rB), rA</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valB} \leftarrow R[\text{rB}]$
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB} + \text{valC}$
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valE}]$
Write back		$R[\text{rA}] \leftarrow \text{valM}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

Y86-64 的顺序实现

sequential implementation

`popq` 中，会将 `valA` 和 `valB` 的值都设置为 $R[\%rsp]$ ，因为一个要用于去当内存，读出旧 $M[\%rsp]$ 处的值，一个要用于计算，更新 $R[\%rsp]$ 。

为了统一，在 `popq` 中，用于计算的依旧是 `valB`。

- `pushq %rsp` 的行为：`pushq` 压入的是旧的 `%rsp`，然后 `%rsp` 减 8
- `popq %rsp` 的行为：`popq` 读出的是旧的 $M[\%rsp]$ ，然后 `%rsp` 加 8

↑ 其他情况：

`pushq` 先 -8 再压栈；`popq` 先读出再 +8

Stage	<code>pushq rA</code>	<code>popq rA</code>
Fetch	$i_{code}:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$	$i_{code}:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC+1]$
	$valP \leftarrow PC + 2$	$valP \leftarrow PC + 2$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[\%rsp]$	$valA \leftarrow R[\%rsp]$ $valB \leftarrow R[\%rsp]$
Execute	$valE \leftarrow valB + (-8)$	$valE \leftarrow valB + 8$
Memory	$M_8[valE] \leftarrow valA$	$valM \leftarrow M_8[valA]$
Write back	$R[\%rsp] \leftarrow valE$	$R[\%rsp] \leftarrow valE$ $R[rA] \leftarrow valM$
PC update	$PC \leftarrow valP$	$PC \leftarrow valP$

Y86-64 的顺序实现

sequential implementation

`ret` 指令和 `popq` 指令类似，`call` 指令和 `pushq` 指令类似，区别只有 PC 更新的部分。

所以，同样注意他们用于计算的依旧是 `valB`。

Stage	jXX Dest	call Dest	ret
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 9$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ <div style="border: 1px solid red; padding: 2px;">$\text{valC} \leftarrow M_8[\text{PC} + 1]$</div> $\text{valP} \leftarrow \text{PC} + 9$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valP} \leftarrow \text{PC} + 1$
Decode			$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
Execute	$\text{Cnd} \leftarrow \text{Cond(CC, ifun)}$	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow \text{valB} + 8$
Memory		$M_8[\text{valE}] \leftarrow \text{valP}$	$\text{valM} \leftarrow M_8[\text{valA}]$
Write back		$R[\%rsp] \leftarrow \text{valE}$	$R[\%rsp] \leftarrow \text{valE}$
PC update	$\text{PC} \leftarrow \text{Cnd ? valC : valP}$	<div style="border: 1px solid red; padding: 2px;">$\text{PC} \leftarrow \text{valC}$</div>	<div style="border: 1px solid red; padding: 2px;">$\text{PC} \leftarrow \text{valM}$</div>

HCL 代码

hardware description/control language

HCL 语法包括两种表达式类型：**布尔表达式**（单个位的信息）和**整数表达式**（多个位的信息），分别用 `bool-expr` 和 `int-expr` 表示。

布尔表达式

逻辑操作

`a && b`, `a || b`, `!a` (与、或、非)

字符比较

`A == B`, `A != B`, `A < B`, `A <= B`, `A >= B`, `A > B`

集合成员资格

`A in { B, C, D }`

等同于 `A == B || A == C || A == D`

字符表达式

case 表达式

```
[  
  bool-expr1 : int-expr1  
  bool-expr2 : int-expr2  
  ...  
  bool-exprk : int-exprk  
]
```

- `bool-expr_i` 决定是否选择该 case。
- `int-expr_i` 为该 case 的值。

依次评估测试表达式，返回第一个成功测试的字符表达式 `A`, `B`, `C`

顺序实现 - 取指阶段

sequential implementation: fetch stage

```
# 指令代码
word icode = [
    imem_error: INOP; # 读取出了问题, 返回空指令
    1: imem_icode; # 读取成功, 返回指令代码
];

# 指令功能
word ifun = [
    imem_error: FNONE; # 读取出了问题, 返回空操作
    1: imem_ifun; # 读取成功, 返回指令功能
];

# 指令是否有效
bool instr_valid = icode in {
    INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
    IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ
};

# 是否需要寄存器
bool need_regs = icode in {
    TRRMOVQ, TPOQ, TPUSHQ, TPPOPO
};
```

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						

顺序实现 - 译码阶段

sequential implementation: decode stage

```
# 源寄存器 A 的选择
word srcA = [
    icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA;
    icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # 不需要寄存器
];
# 源寄存器 B 的选择
word srcB = [
    icode in { IOPQ, IRMMOVQ, IMRMOVQ } : rB;
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # 不需要寄存器
];
```

```
# 目标寄存器 E 的选择
word dstE = [
    icode in { IRRMOVQ } && Cnd : rB; # 支持 cmovXX
    icode in { IIRMOVQ, IOPQ } : rB; # 注意这里!
    icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # 不写入任何寄存器
];
# 目标寄存器 M 的选择
word dstM = [
    icode in { IMRMOVQ, IPOPQ } : rA;
    1 : RNONE; # 不写入任何寄存器
];
```

寄存器 ID `srcA` 表明应该读哪个寄存器以产生 `valA` (注意不是 `aluA`) , `srcB` 同理。

寄存器 ID `dstE` 表明写端口 E 的目的寄存器, 计算出来的 `valE` 将放在那里, `dstM` 同理。

在 SEQ 实现中, 回写和译码放到了一起。

顺序实现 - 执行阶段

sequential implementation: execute stage

```
# 选择 ALU 的输入 A
word aluA = [
    icode in { IRRMOVQ, IOPQ } : valA; # 指令码为 IRRMOVQ 时, 执行 valA + 0
    icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC; # 立即数相关, 都送入的是 aluA
    icode in { ICALL, IPUSHQ } : -8; # 减少栈指针
    icode in { IRET, IPOPQ } : 8; # 增加栈指针
    # 其他指令不需要 ALU
];
# 选择 ALU 的输入 B, 再次强调 OPq 指令中, 是 `valB OP valA`
word aluB = [
    icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL, IPUSHQ, IRET, IPOPQ } : valB; # 大部分都用 valB
    icode in { IRRMOVQ, IIRMOVQ } : 0; # 指令码为 IRRMOVQ 或 IIRMOVQ 时, 选择 0
    # 其他指令不需要 ALU
];
# 设置 ALU 功能
word alufun = [
    icode == IOPQ : ifun; # 如果指令码为 IOPQ, 则使用 ifun 指定的功能
    1 : ALUADD; # 默认使用 ALUADD 功能
];
# 是否更新条件码
bool set_cc = icode in { IOPQ }; # 仅在指令码为 IOPQ 时更新条件码
```

顺序实现 - 访存阶段

sequential implementation: memory stage

```
# 设置读取控制信号
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
# 设置写入控制信号
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };
# 选择内存地址
word mem_addr = [
    icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
    icode in { IPOPQ, IRET } : valA; # valE 算栈指针去了
    # 其它指令不需要使用地址
];

```

```
# 选择内存输入数据
word mem_data = [
    # 从寄存器取值
    icode in { IRMMOVQ, IPUSHQ } : valA; # valB 算地址去了
    # 返回 PC
    icode == ICALL : valP;
    # 默认: 不写入任何数据
];
# 确定指令状态
word Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid : SINS;
    icode == IHALT : SHLT;
    1 : SAOK;
];

```

顺序实现 - 更新 PC 阶段

sequential implementation: update pc stage

```
# 设置新 PC 值
word new_pc = [
    # 调用指令, 使用指令常量
    icode == ICALL : valC;
    # 条件跳转且条件满足, 使用指令常量
    icode == IJXX && Cnd : valC;
    # RET 指令完成, 使用栈中的值
    icode == IRET : valM;
    # 默认: 使用递增的 PC 值
    # 等于上一条指令地址 + 上一条指令长度 1, 2, 9, 10
    1 : valP;
];
```

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
cmovXX rA, rB	2	fn	rA	rB						
irmovq V, rB	3	0	F	rB					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						

顺序实现 - 总结

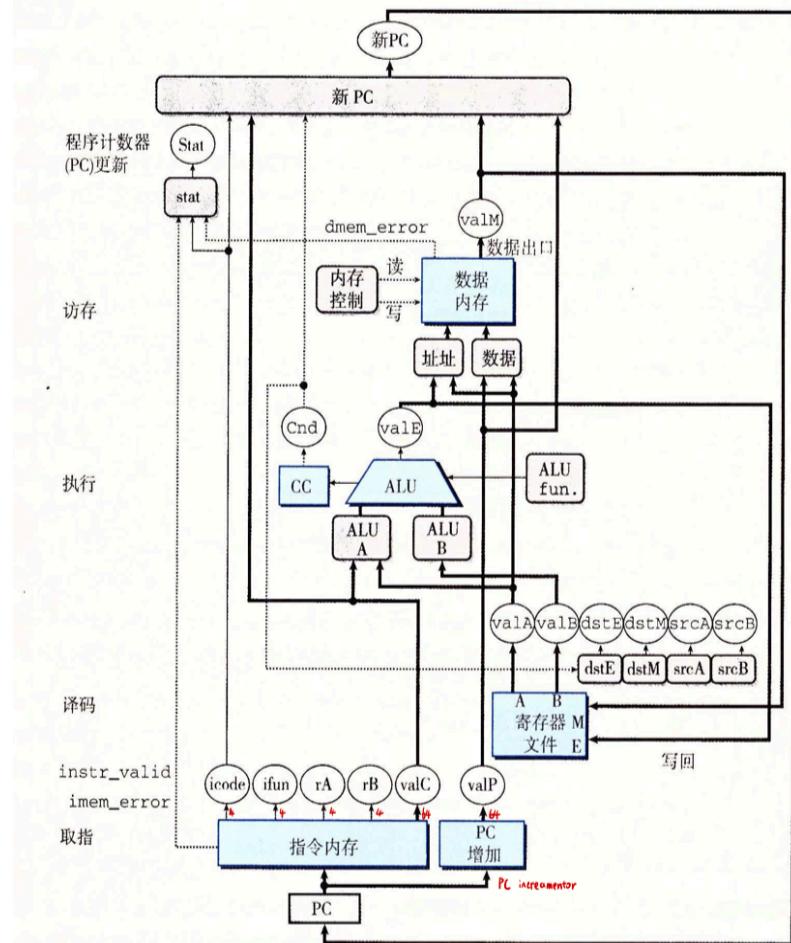
sequential implementation: summary

重点关注：

- valA 和 valB 怎么连的
- 什么时候 valP 可以直传内存
- 什么时候 valA 可以直传内存

答案：

1. call
2. rmmovq pushq popq retq (mrmovq 需要吗？不！)



流水线实现

pipelined implementation

什么是流水线? 答: 通过同一时间上的并行, 来提高效率。

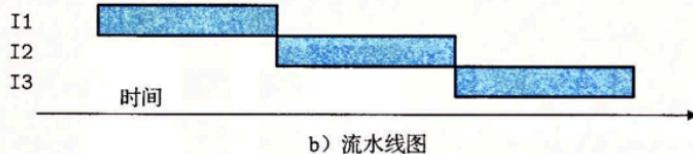
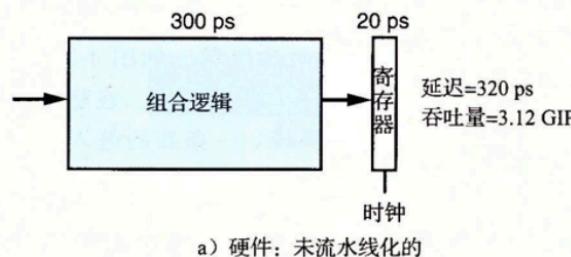


图 4-32 非流水线化的计算硬件。每个 320ps 的周期内, 系统用 300ps 计算组合逻辑函数, 20ps 将结果存到输出寄存器中

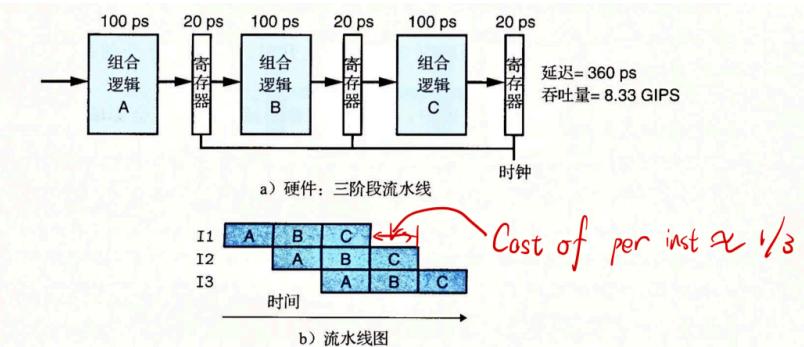


图 4-33 三阶段流水线化的计算硬件。计算被划分为三个阶段 A、B 和 C。每经过一个 120ps 的周期, 每条指令就行进通过一个阶段

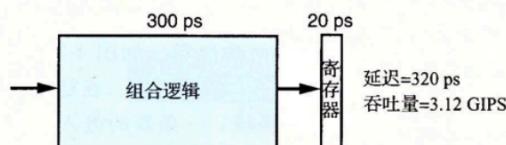
流水线实现

pipelined implementation

吞吐量：单位时间内完成的指令数量。

单位：每秒千兆指令 (GIPS, 10^9 instructions per second, 等于 1 ns (10^{-9} s) 执行多少条指令再加上个 G) 。

$$\text{吞吐量} = \frac{1}{(300 + 20)\text{ps}} \cdot \frac{1000\text{ps}}{1\text{ns}} = 3.125\text{GIPS}$$



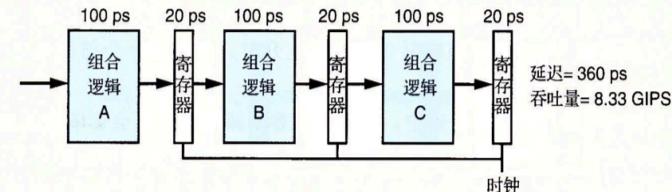
a) 硬件：未流水线化的



b) 流水图

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

$$\text{吞吐量} = \frac{1}{(100 + 20)\text{ps}} \cdot \frac{1000\text{ps}}{1\text{ns}} = 8.33\text{GIPS}$$



a) 硬件：三阶段流水线



b) 流水线图

图 4-33 三阶段流水线化的计算硬件。计算被划分为三个阶段 A、B 和 C。每经过一个 120ps 的周期，每条指令就行进通过一个阶段

Cost of per inst \approx 1/3

流水线实现的局限性

pipelined implementation: limitations

- 运行时钟的速率是由最慢的阶段的延迟限制的。每个时钟周期的最后，只有最慢的阶段会一直处于活动状态
- 流水线过深：不能无限增加流水线的阶段数，因为此时流水线寄存器的延迟占比加大。
- 数据冒险

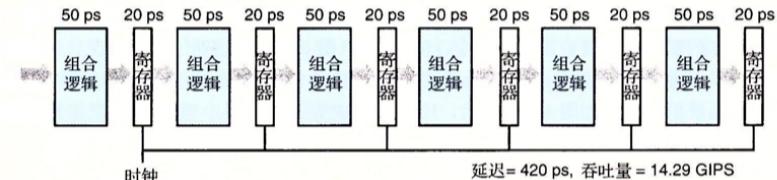
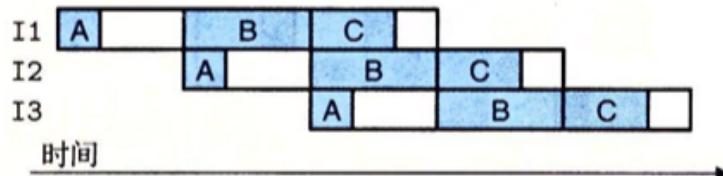


图 4-37 由开销造成的流水线技术的局限性。在组合逻辑被分成较小的块时，由寄存器更新引起的延迟就成为了一个限制因素

```

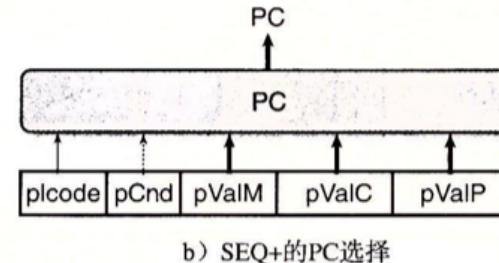
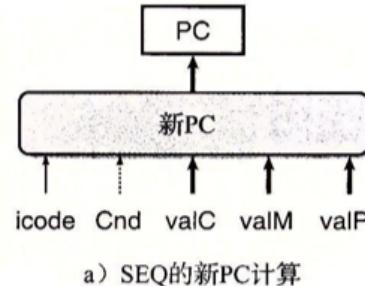
irmovq $50, %rax    ; 将立即数50移动到寄存器rax中
addq %rax, %rbx     ; 将寄存器rax中的值与rbx中的值相加
mrmovq 100(%rbx), %rdx ; 从内存地址rbx+100读取值到寄存器rdx中

```

SEQ 与 SEQ+

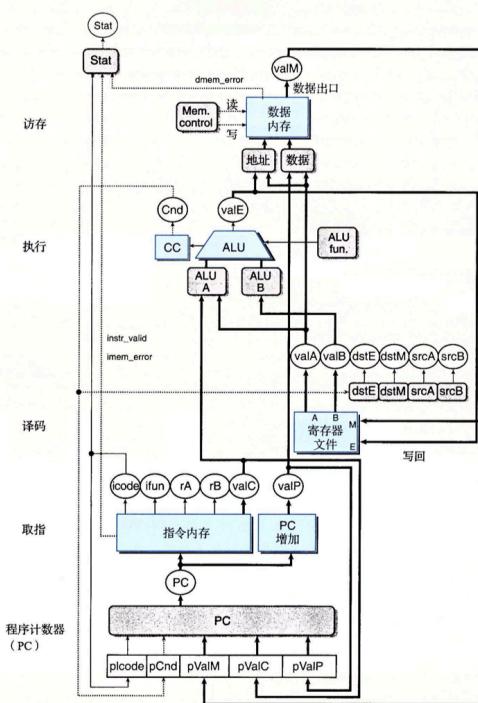
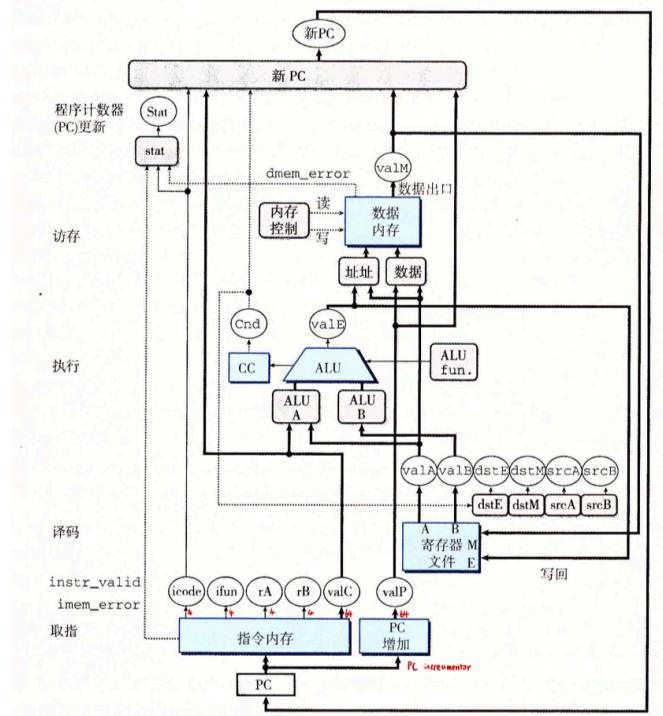
SEQ vs SEQ+

- 在 SEQ 中，PC 计算发生在时钟周期结束的时候，根据当前时钟周期内计算出的信号值来计算 PC 寄存器的新值。💡
- 在 SEQ+ 中，我们需要在每个时钟周期都可以取出下一条指令的地址，所以更新 PC 阶段在一个时钟周期开始时执行，而不是结束时才执行。
- SEQ+ 没有硬件寄存器来存放程序计数器。**而是根据从前一条指令保存下来的一些状态信息动态地计算 PC。



此处，小写的 `p` 前缀表示它们保存的是前一个周期中产生的控制信号。

SEQ vs SEQ+



弱化一些的 PIPE 结构

PIPE-

各个信号的命名：

- 在命名系统中，大写的前缀“D”、“E”、“M”和“W”指的是流水线寄存器，所以 M_stat 指的是流水线寄存器 M 的状态码字段。

可以理解为，对应阶段开始时就已经是正确的值了（且由于不回写的原则，所以该时钟周期内不会再改变，直到下一个时钟上升沿的到来）

- 小写的前缀 f、d、e、m 和 w 指的是流水线阶段，所以 m_stat 指的是在访存阶段中由控制逻辑块产生出的状态信号。

可以理解为，对应阶段中，完成相应运算时才会是正确的值

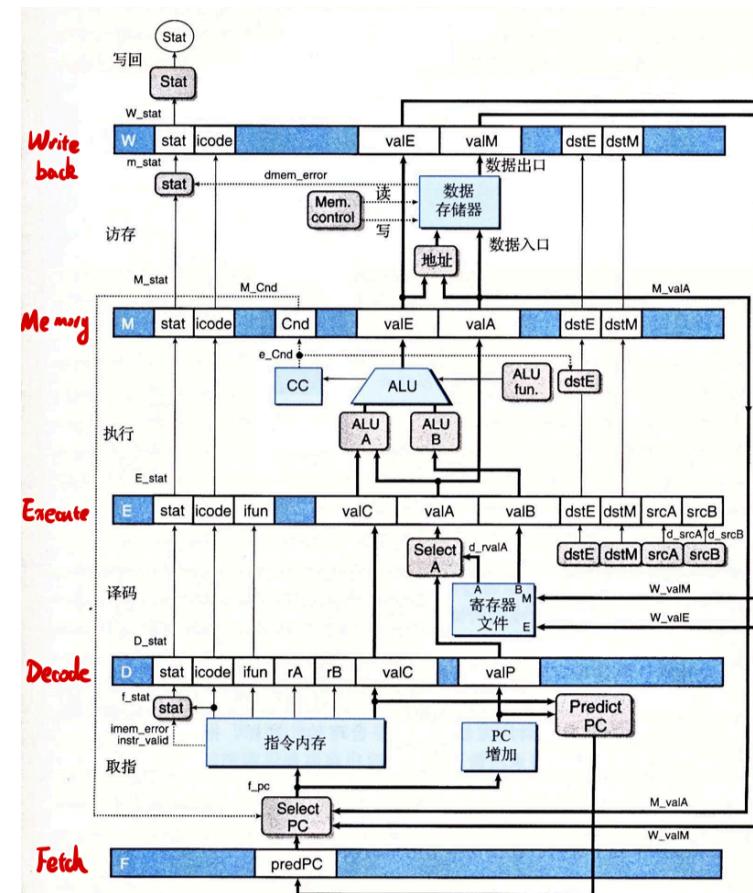


图 4-41 PIPE-1 的硬件结构，一个初始的流水线化实现。通过往 SEQ-(图 4-40)中插入流水线寄存器，我们创建了一个五阶段的流水线。这个版本有几个缺陷，稍后就会解决这些问题

SEQ+ vs PIPE-

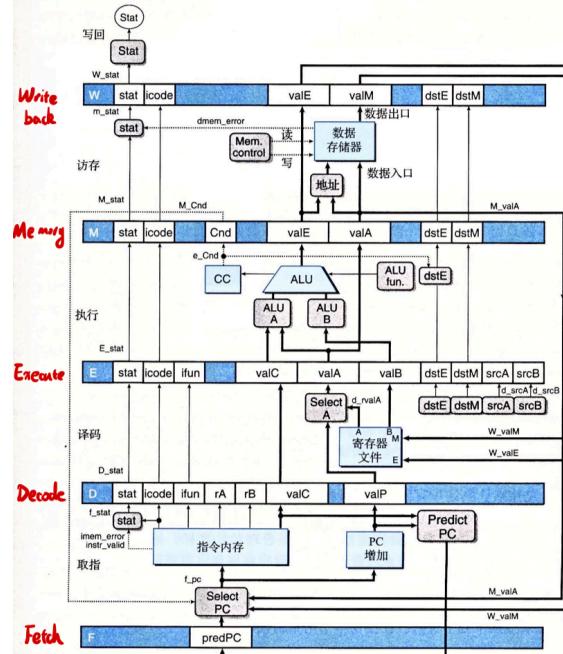
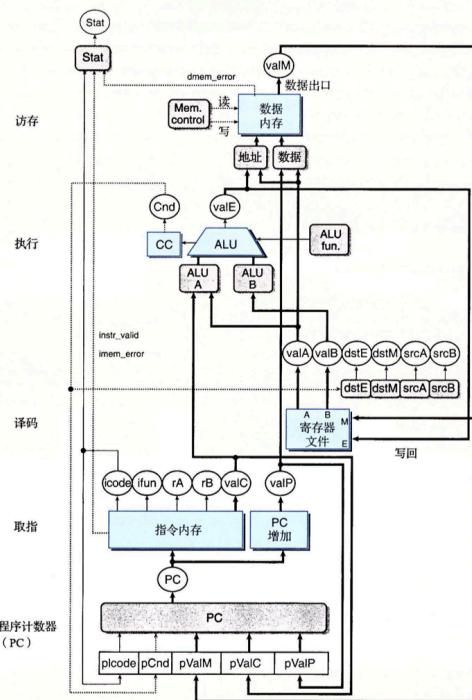


图 4-41 PIPE- 的硬件结构，一个初始的流水线化实现。通过往 SEQ+ (图 4-40) 中插入流水线寄存器，我们创建了一个五阶段的流水线。这个版本有几个缺陷，稍后就会解决这些问题。

弱化一些的 PIPE 结构

PIPE-

- 等价于在 SEQ+ 中插入了流水线寄存器 (**他们都**
是即将由对应阶段进行处理)
- F: Fetch, 取指阶段
- D: Decode, 译码阶段
- E: Execute, 执行阶段
- M: Memory, 访存阶段
- W: Write back, 写回阶段
- 同时, 有个新模块 selectA 来选择 valA 的来
源
 - valP : call jXX (后面讲, 可以想想为
啥, 提示: 控制冒险)
 - d_valA : 其他未转发的情况 (后面讲)

BACK

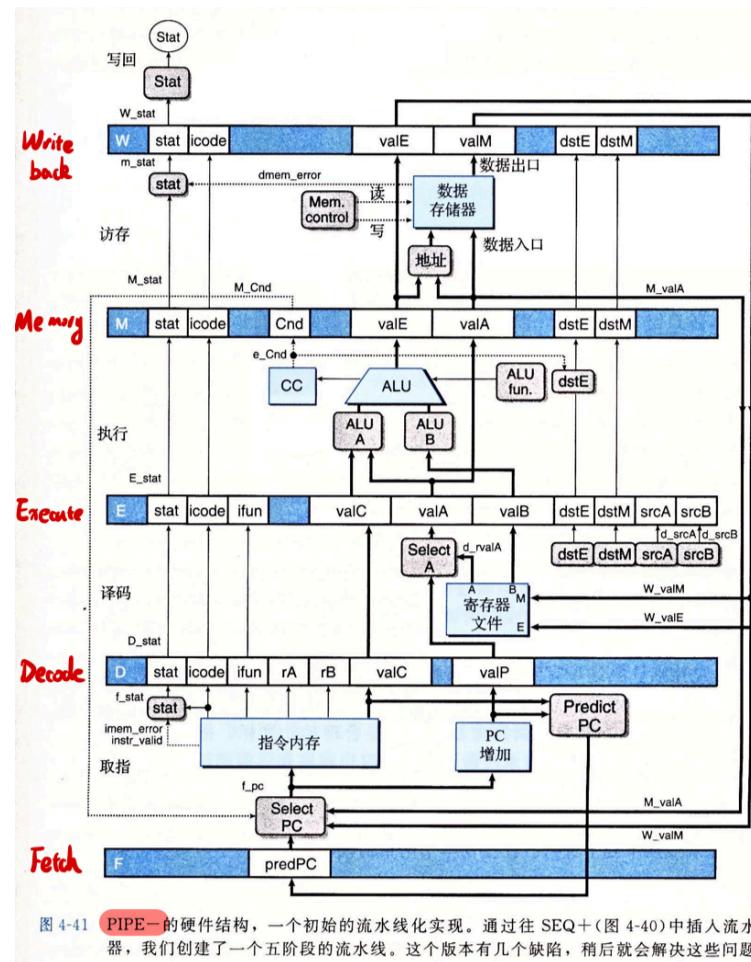


图 4-41 PIPE- 的硬件结构, 一个初始的流水线化实现。通过往 SEQ+(图 4-40)中插入流水线寄存器, 我们创建了一个五阶段的流水线。这个版本有几个缺陷, 稍后就会解决这些问题

PIPE- 分支预测

PIPE- branch prediction

分支预测：猜测分支方向并根据猜测开始取指的技术。

对于 `jXX` 指令，有两种情况：

- 分支不执行：下一条 PC 是 `valP`
- 分支执行：下一条 PC 是 `valC`

由于我们现在是流水线，我们需要每个时钟周期都能给出一个指令地址用于取址，所以我们采用分支预测：

最简单的策略：总是预测选择了条件分支，因而预测 PC 的新值为 `valC`。

对于 `ret` 指令，我们等待它通过写回 `W` 阶段（从而可以从 `M` 中得到之前压栈的返回值并更新 `PC`）。

同条件转移不同，`ret` 可能的返回值几乎是无限的，因为返回地址是位于栈顶的字，其内容可以是任意的。

流水线冒险

hazards

冒险分为两类：

1. **数据冒险 (Data Hazard)**: 下一条指令需要使用当前指令计算的结果。
2. **控制冒险 (Control Hazard)**: 指令需要确定下一条指令的位置，例如跳转、调用或返回指令。

数据冒险

data hazard

数据冒险是相对容易理解的。

在右图代码中，`%rax` 的值需要在第 6 个周期结束时才能完成写回，但是在第 6 个周期内，正处于译码阶段的 `addq` 指令就需要使用 `%rax` 的值了。这就产生了数据冒险。

类似可推得，如果一条指令的操作数被它前面 3 条指令中的任意一条改变的话，都会出现数据冒险。

我们需要满足：当后来的需要某一寄存器的指令处于译码 D 阶段时，该寄存器的值必须已经更新完毕（即已经 **完成** 写回 W 阶段）。

$$5(\text{完成}W) - 1(\text{开始}D, \text{ 即完成}F) - 1(\text{错开一条指令}) = 3$$

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

数据冒险的解决：暂停

data hazard resolution: stall

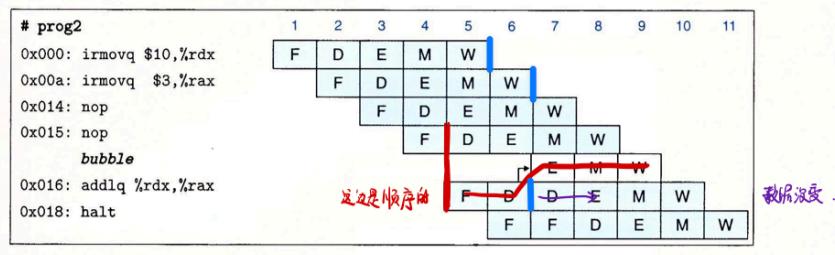
暂停: 暂停时, 处理器会停止流水线中一条或多条指令, 直到冒险条件不再满足。

让一条指令停顿在译码阶段，直到产生它的源操作数的指令通过了写回阶段，这样我们的处理器就能避免数据冒险。（即，下一个时钟周期开始时，此指令开始真正译码，此时源操作数已经更新完毕）

暂停技术就是让一组指令阻塞在它们所处的阶段，而允许其他指令继续通过流水线（如右图 `irmovq` 指令）。

每次要把一条指令阻塞在**译码阶段**，就在**执行阶段**（下一个阶段）插入一个气泡。

气泡就像一个自动产生的 `nop` 指令，**它不会改变寄存器、内存、条件码或程序状态。**



暂停 vs 气泡

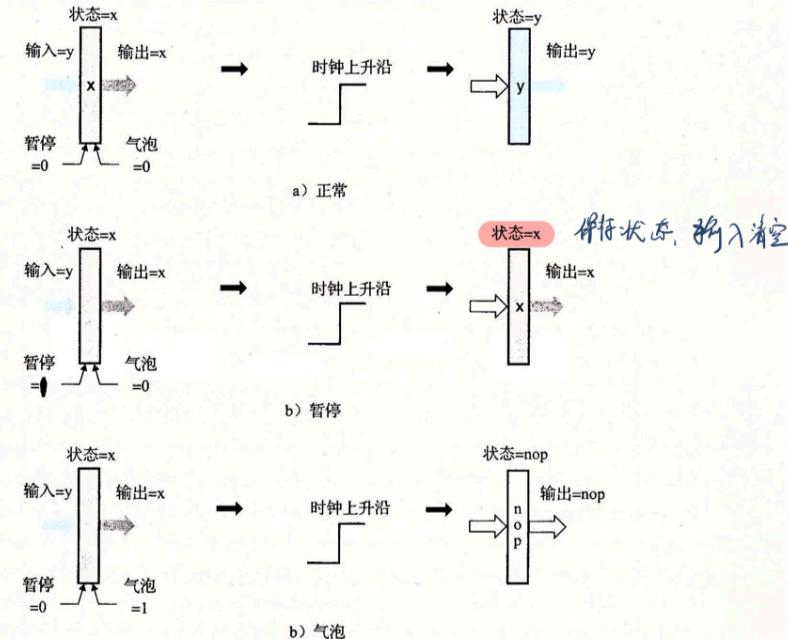
stall vs bubble

- 正常: 寄存器的状态和输出被设置成输入的值
- 暂停: 状态保持为先前的值不变
- 气泡: 会用 `nop` 操作的状态覆盖当前状态

所以, 在上页图中, 我们说:

- 给执行阶段插入了气泡
- 对译码阶段执行了暂停

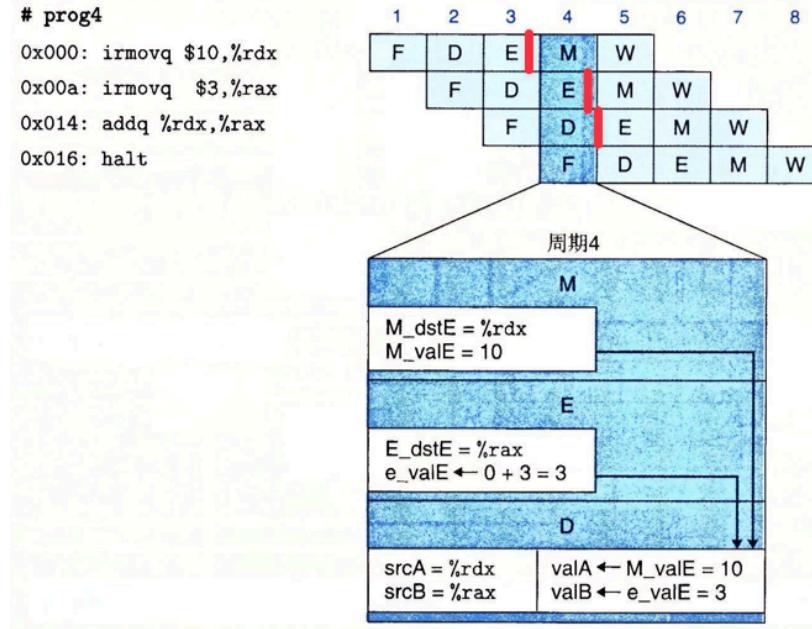
BACK



数据冒险的解决：转发

data hazard resolution: forwarding

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



实际上，在这里，所需要的真实 `%rax` 值，早在 4E 快结束时就已经计算出来了。而我们需要用到它的是 5E 的开始。

回忆：大写的寄存器是在对应阶段开始时就已经是正确的值。

数据冒险的解决：转发

data hazard resolution: forwarding

转发：将结果值直接从一个流水线阶段传到较早阶段的技术。

这个过程可以发生在许多阶段（下图中，要到 6E 寄存器才定下来，所以只要在时钟上升沿来之前，都来得及）。

特殊的数据冒险：加载 / 使用冒险

data hazard: load / use hazard

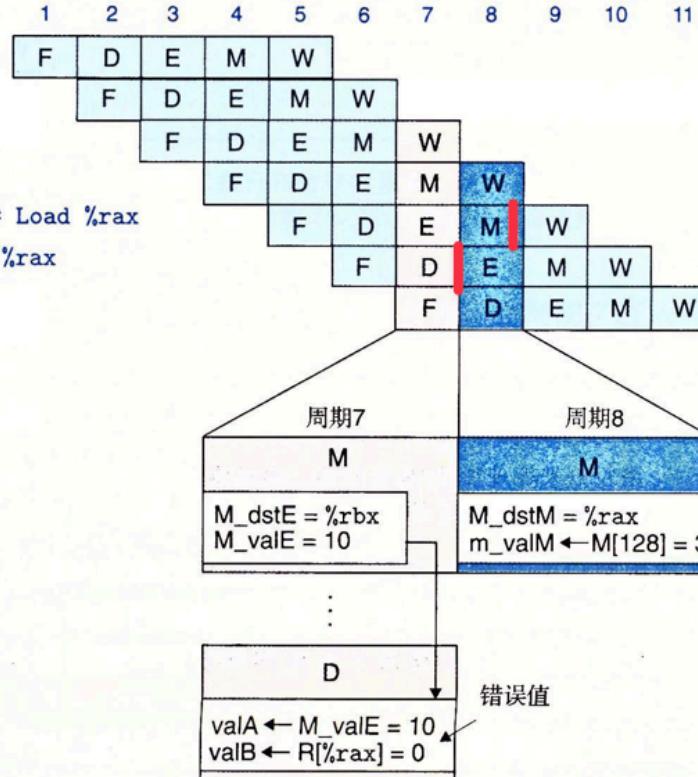
- 如果在先前指令的 E 执行阶段（其内靠后时）就已经可以得到正确值，那么由于后面的指令至少落后 1 个阶段，我们总可以在后面指令的 E 寄存器最终确定之前，将正确值转发解决问题。
- 如果在先前指令的 M 访存阶段（其内靠后时）才能得到正确值，且后面指令紧跟其后，那么当我们实际得到正确值时，必然赶不上后面指令的 E 寄存器最终确定，所以我们必须暂停流水线。
- 所以，加载 / 使用冒险只发生在 `mrmovq` 后立即使用对应寄存器的情况下。

书上老说什么把值送回过去，我觉得第一次读真难明白吧。

特殊的数据冒险：加载 / 使用冒险

data hazard: load / use hazard

```
# 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
0x032: addq %ebx,%eax # Use %rax
0x034: halt
```



加载 / 使用冒险解决方案：暂停 + 转发

load / use hazard solution

依旧是：

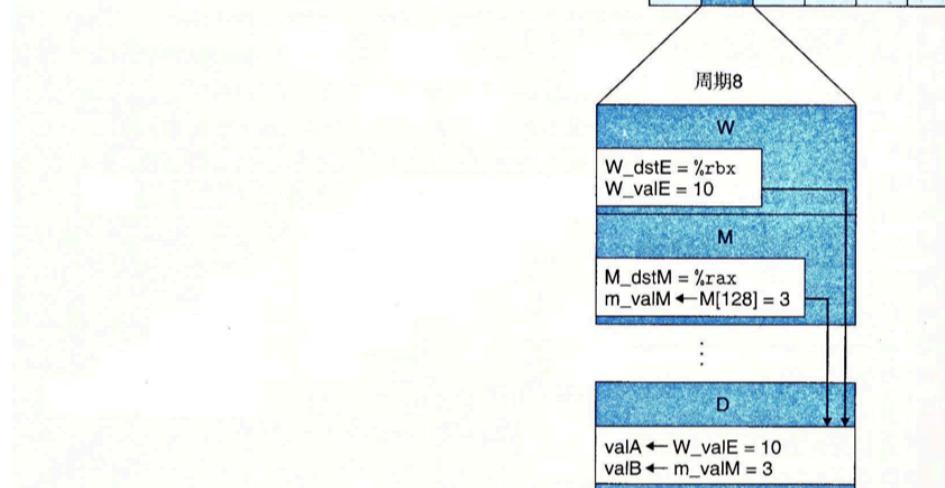
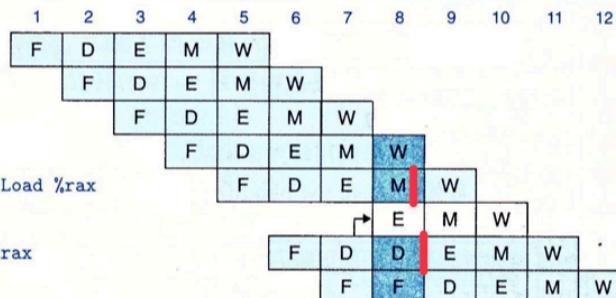
- 译码阶段中的指令暂停 1 个周期
- 执行阶段中插入 1 个气泡

此时，`m_valM` 的值已经更新完毕，所以可以转发到 `d_valA`。

`m_valM`：在 M 阶段内，取出的内存值

`d_valA`：在 D 阶段内，计算得到的即将设置为 `E_valA` 的值

```
# 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
```



PIPE 最终结构

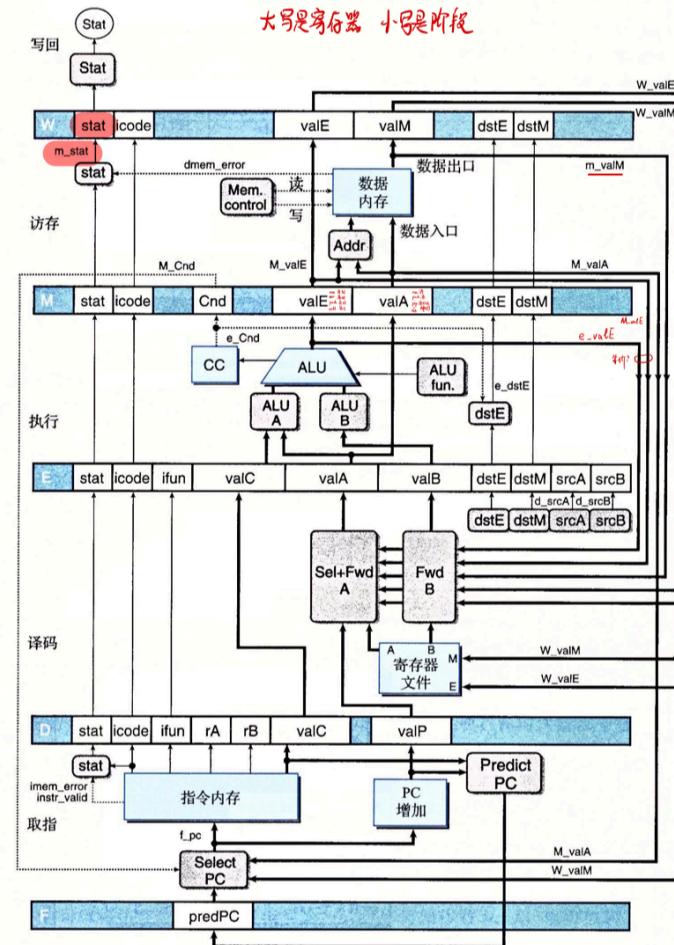
PIPE final structure

把各个转发逻辑都画出来，就得到了最终的结构。

注意：

- Sel + Fwd A：是 PIPE- 中标号为 Select A 的块的功能与转发逻辑的结合。💡
- Fwd B

◀
BACK



PIPE- vs PIPE

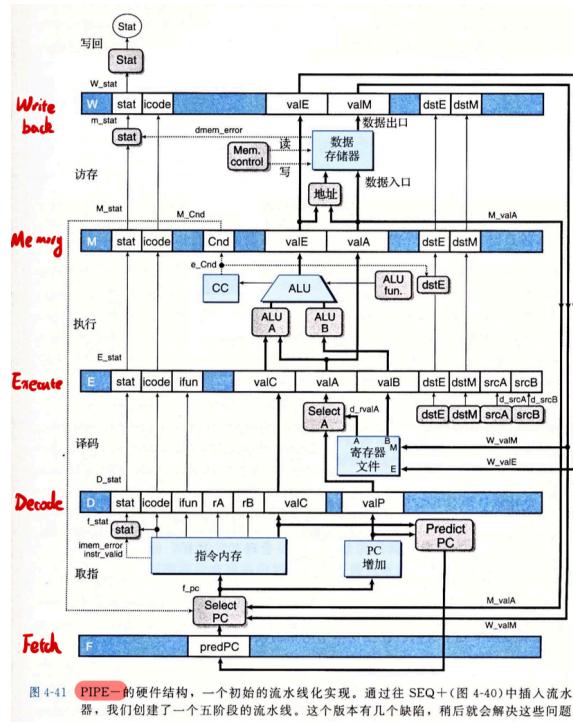
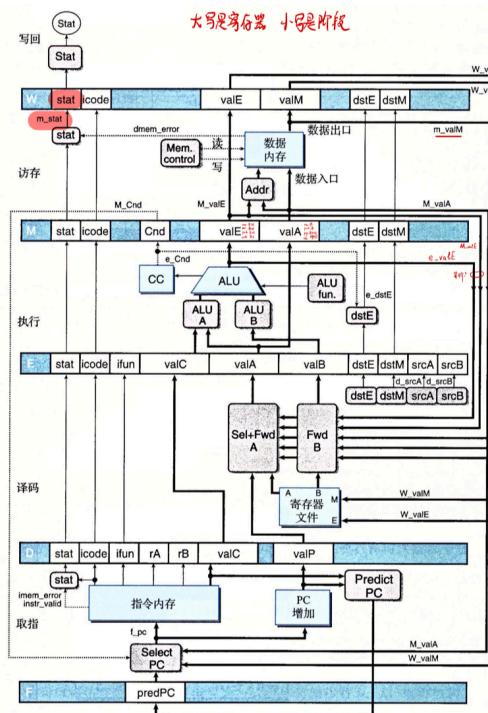


图 4-41 PIPE- 的硬件结构，一个初始的流水线化实现。通过往 SEQ+（图 4-40）中插入流水线寄存器，我们创建了一个五阶段的流水线。这个版本有几个缺陷，稍后就会解决这些问题。



结构之间的差异

differences between structures

SEQ

- 完全的分阶段，且顺序执行
- 没有流水线寄存器
- 没有转发逻辑

SEQ+

- 把计算新 PC 计算放到了最开始
- 目的：为了能够划分流水线做准备，当前指令到 D 阶段时，应当能开始下一条指令的 F 阶段
- 依旧是没有转发逻辑、且顺序执行

 结构差异图

PIPE-

- 在 SEQ+ 的基础上，增加了流水线寄存器
- 增加了一些转发逻辑（但不是所有）
- 新的转发源：`M_valA` `W_valW` `W_valE` (流水线寄存器们)
- 转发目的地：`d_valA` `d_valB`

 结构差异图

PIPE

- 在 PIPE- 的基础上，完善了转发逻辑，可以转发更多的计算结果（小写开头的，而不是只有大写开头的流水线寄存器）
- 新的转发源：`e_valE` `m_valM` (中间计算结果们)

 结构差异图

控制冒险

control hazard

控制冒险：当处理器无法根据处于取指阶段的当前指令来确定下一条指令的地址时，就会产生控制冒险。

发生条件： RET JXX

RET 指令需要弹栈（访存）才能得到下一条指令的地址。

JXX 指令需要根据条件码来确定下一条指令的地址。

- Cnd \leftarrow Cond(CC, ifun)
- Cnd ? valC : valP

```
# 指令应从哪个地址获取
word f_pc = [
    # 分支预测错误时，从增量的 PC 取指令
    # 传递路径: D_valP -> E_valA -> M_valA
    # 条件跳转指令且条件不满足时
    M_icode == IJXX && !M_Cnd : M_valA;
    # RET 指令终于执行到回写阶段时 (即过了访存阶段)
    W_icode == IRET : W_valM;
    # 默认情况下，使用预测的 PC 值
    1 : F_predPC;
];
```

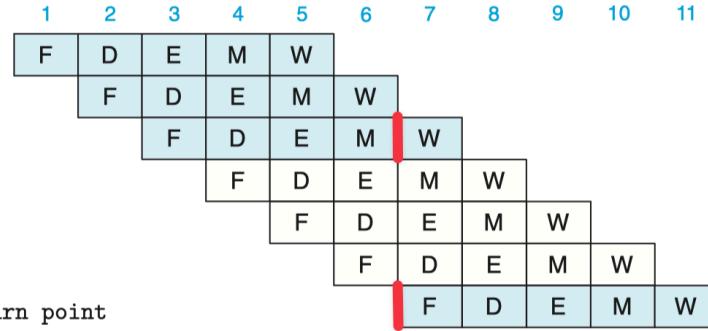
 PIPELINE 电路图

注意，这里用到的都是流水线寄存器，而没有中间计算结果（小写前缀）。

控制冒险：RET

control hazard: RET

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



涉及取指 F 阶段的不能转发中间结果 m_val_M ，必须等到流水线寄存器 W_val_M 更新完毕！

为什么：取址阶段没有相关的硬件电路处理中间结果的转发！必须是流水线寄存器同步。

所以需要插入 3 个气泡：

$$4(\text{RET 完成} M) - 0(\text{开始} F) - 1(\text{错开一条指令}) = 3$$

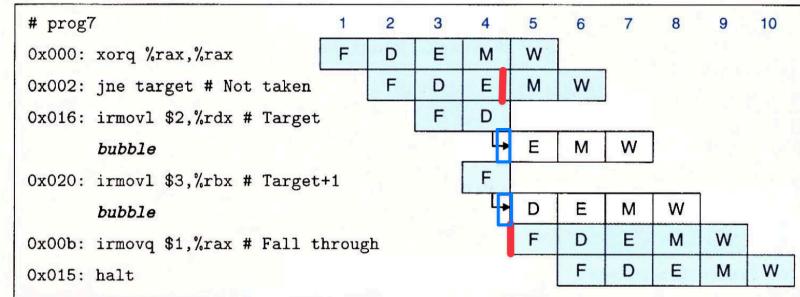
为什么是气泡：💡 暂停 vs 气泡 暂停保留状态，气泡清空状态。

控制冒险：JXX

control hazard: JXX

- 分支逻辑发现不应该选择分支之前（到达执行 E 阶段），已经取出了两条指令，它们不应该继续执行下去了。
- 这两条指令都没有导致程序员可见的状态发生改变（没到到执行 E 阶段）。

```
# 是否需要注入气泡至流水线寄存器 D
bool D_bubble =
    # 错误预测的分支
    (E_icode == IJXX && !e_Cnd) ||
    # 在取指阶段暂停，同时 ret 指令通过流水线
    # 但不存在加载/使用冒险的条件 (此时使用暂停)
    !(E_icode in { IMRMOVQ, IPOPQ } &&
      E_dstM in { d_srcA, d_srcB }) &&
    # IRET 指令在 D、E、M 任何一个阶段
    IRET in { D_icode, E_icode, M_icode };
```

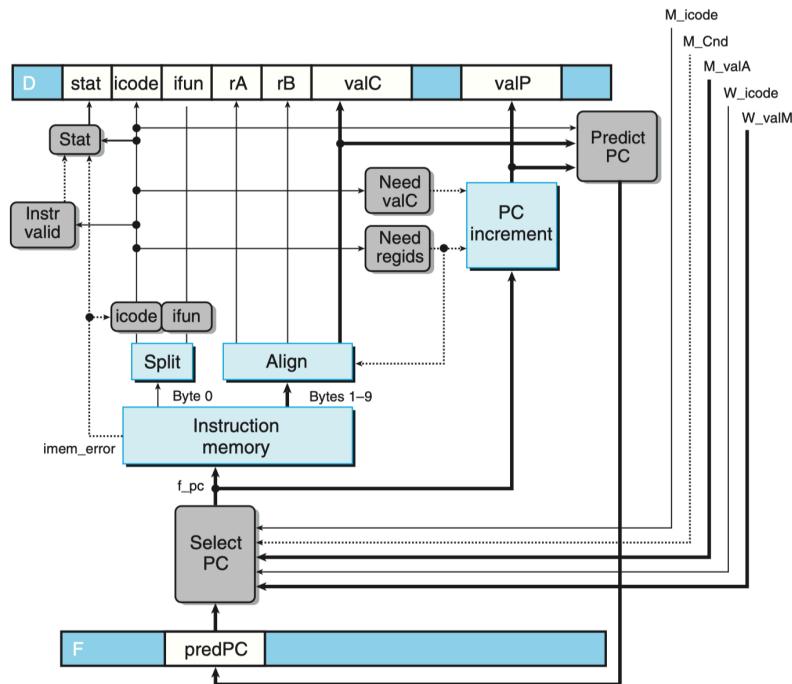


```
# 是否需要注入气泡至流水线寄存器 E
bool E_bubble =
    # 错误预测的分支
    (E_icode == IJXX && !e_Cnd) ||
    # 加载/使用冒险的条件
    E_icode in { IMRMOVQ, IPOPQ } &&
    E_dstM in { d_srcA, d_srcB };
```

PIPELINE 的各阶段实现：取指阶段

pipeline hcl: fetch stage

```
# 指令应从哪个地址获取
word f_pc = [
    # 分支预测错误时, 从增量的 PC 取指令
    # 传递路径: D_valP -> E_valA -> M_valA
    # 条件跳转指令且条件不满足时
    M_icode == IJXX && !M_Cnd : M_valA;
    # RET 指令终于执行到回写阶段时 (即过了访存阶段)
    W_icode == IRET : W_valM;
    # 默认情况下, 使用预测的 PC 值
    1 : F_predPC;
];
# 取指令的 icode
word f_icode = [
    imem_error : INOP; # 指令内存错误, 取 NOP
    1 : imem_icode; # 否则, 取内存中的 icode
];
# 取指令的 ifun
word f_ifun = [
    imem_error : FNONE; # 指令内存错误, 取 NONE
    1 : imem_ifun; # 否则, 取内存中的 ifun
];
```

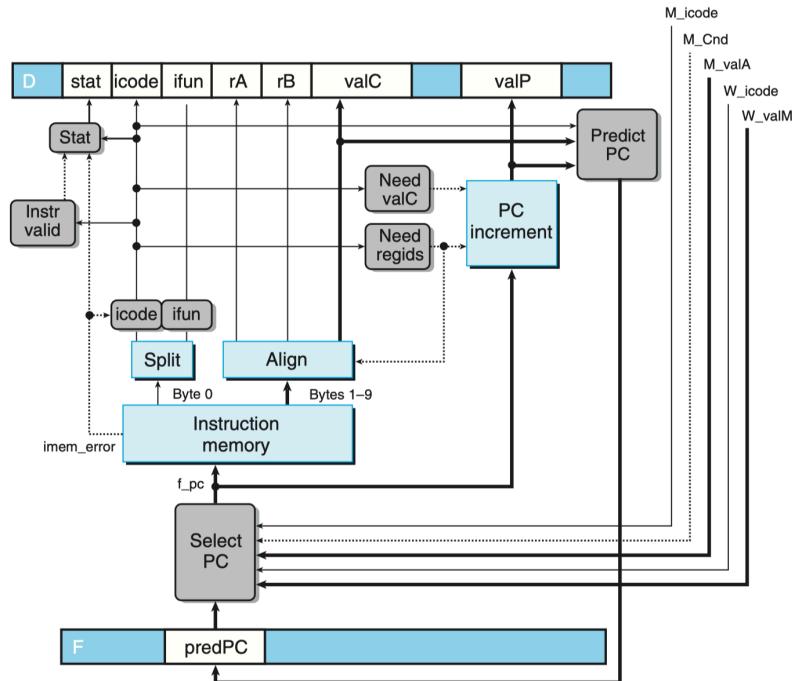


PIPELINE 的各阶段实现：取指阶段

pipeline hcl: fetch stage

```
# 指令是否有效
bool instr_valid = f_icode in {
    INOP, IHALT, IIRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
    IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPOQ
};

# 获取指令的状态码
word f_stat = [
    imem_error : SADR;      # 内存错误
    !instr_valid : SINS;    # 无效指令
    f_icode == IHALT : SHLT; # HALT 指令
    1 : SAOK;              # 默认情况，状态正常
];
```



PIPELINE 的各阶段实现：取指阶段

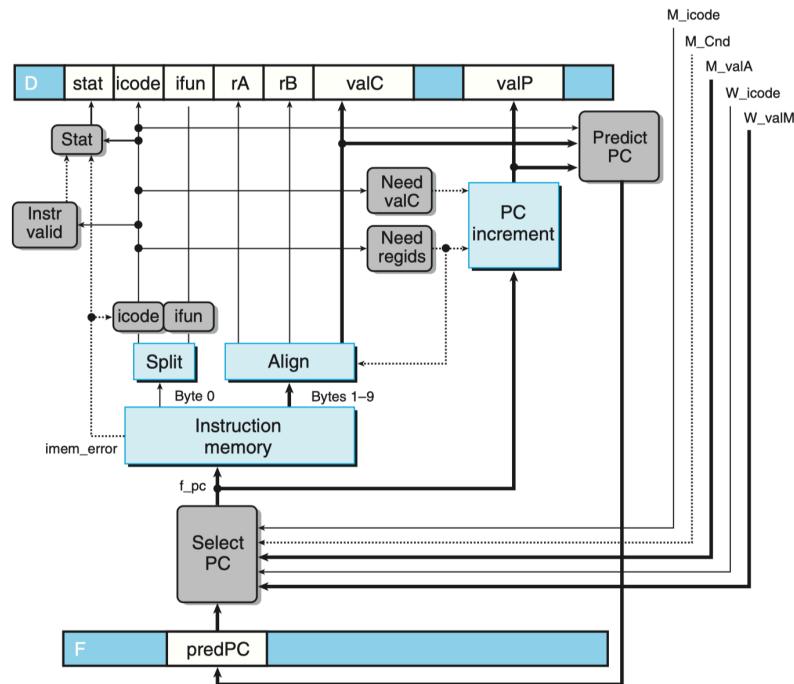
pipeline hcl: fetch stage

```
# 指令是否需要寄存器 ID 字节
# 单字节指令 `HALT`、`NOP`、`RET`；不需要寄存器 `JXX`、`CALL`、
bool need_regids = f_icode in {
    IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
    IIRMOVQ, IRMMOVQ, IMRMOVQ
};

# 指令是否需要常量值
# 作为值；作为 rB 偏移；作为地址
bool need_valC = f_icode in {
    IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL
};

# 预测下一个 PC 值
word f_predPC = [
    # 跳转或调用指令，取 f_valC
    f_icode in { IJXX, ICALL } : f_valC;
    # 否则，取 f_valP
    1 : f_valP;
];

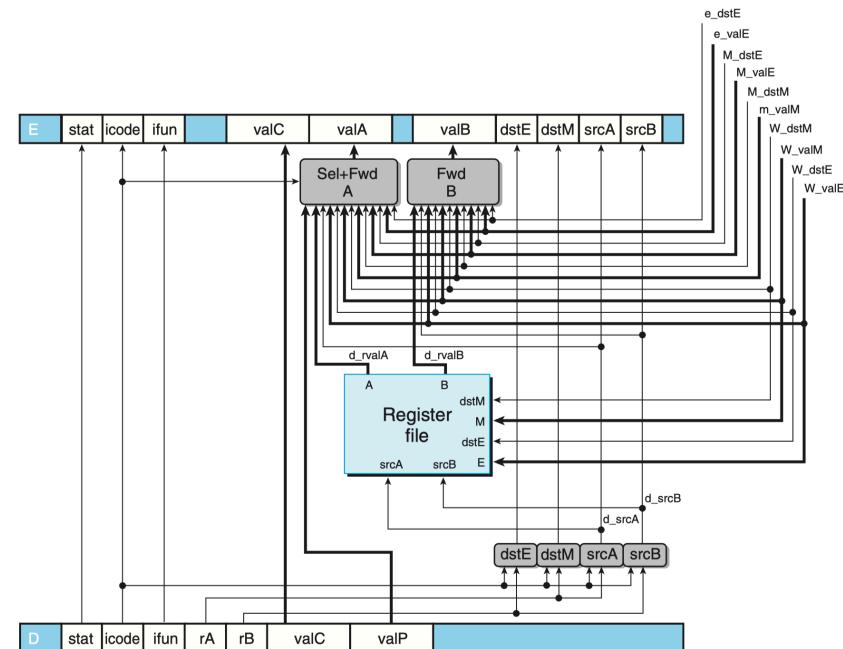
```



PIPELINE 的各阶段实现：译码阶段

pipeline hcl: decode stage

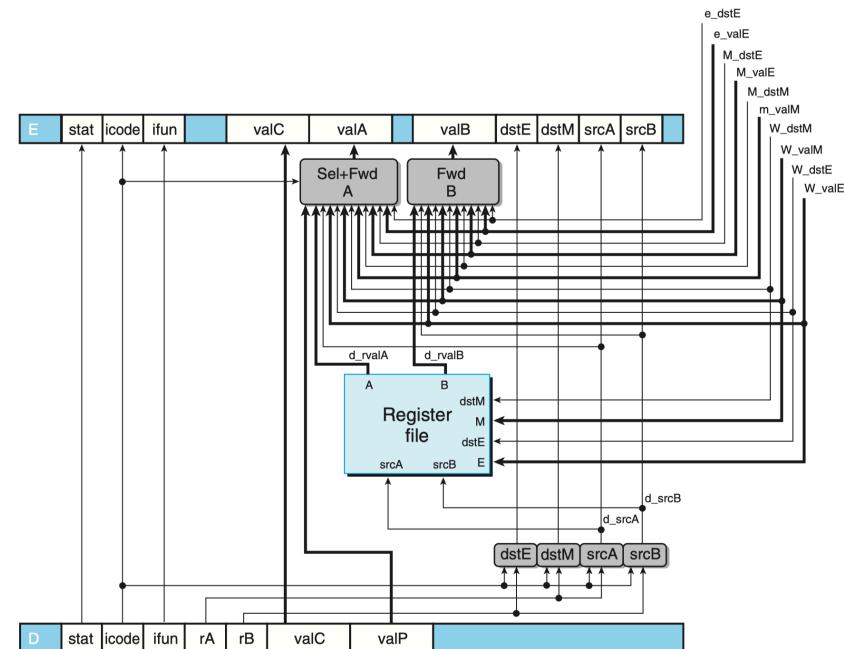
```
# 决定 d_valA 的来源
word d_srcA = [
    # 一般情况, 使用 rA
    D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : D_rA;
    # 此时, valB 也是栈指针
    # 但是同时需要计算新值 (valB 执行阶段计算)、使用旧值访存 (valA)
    D_icode in { IPOPQ, IRET } : RRSP;
    1 : RNONE; # 不需要 valA
];
# 决定 d_valB 的来源
word d_srcB = [
    # 一般情况, 使用 rB
    D_icode in { IOPQ, IRMMOVQ, IMRMOVQ } : D_rB;
    # 涉及栈指针, 需要计算新的栈指针值
    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # 不需要 valB
];
```



PIPELINE 的各阶段实现：译码阶段

pipeline hcl: decode stage

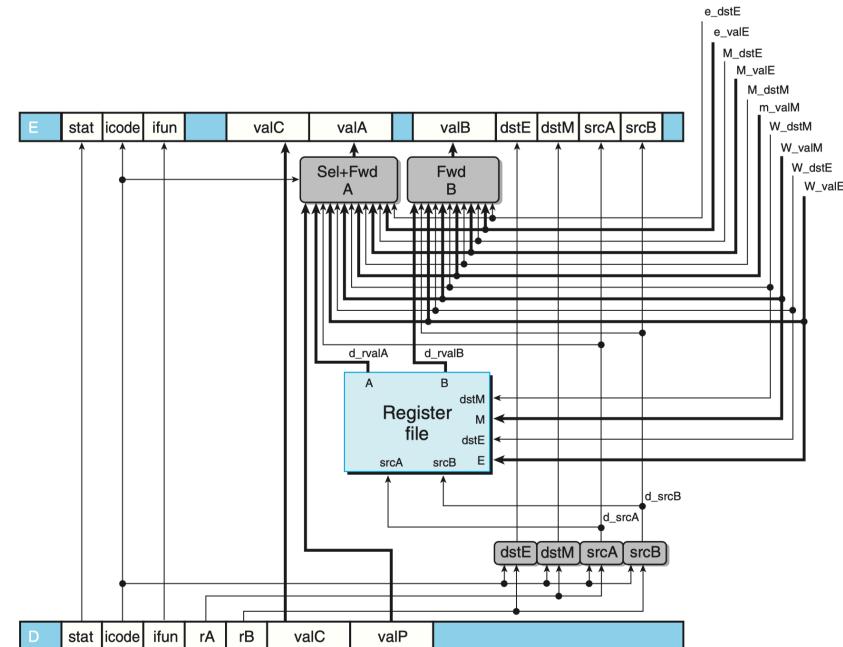
```
# 决定 E 执行阶段计算结果的写入寄存器
word d_dstE = [
    # 一般情况, 写入 rB, 注意 OPQ 指令的 rB 是目的寄存器
    D_icode in { IRRMOVQ, IIRMOVQ, IOPQ } : D_rB;
    # 涉及栈指针, 更新 +8/-8 后的栈指针
    D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
    1 : RNONE; # 不写入 valE 到任何寄存器
];
# 决定 M 访存阶段读出结果的写入寄存器
word d_dstM = [
    # 这两个情况需要更新 valM 到 rA
    D_icode in { IMRMOVQ, IPOPQ } : D_rA;
    1 : RNONE; # 不写入 valM 到任何寄存器
];
```



PIPELINE 的各阶段实现：译码阶段

pipeline hcl: decode stage

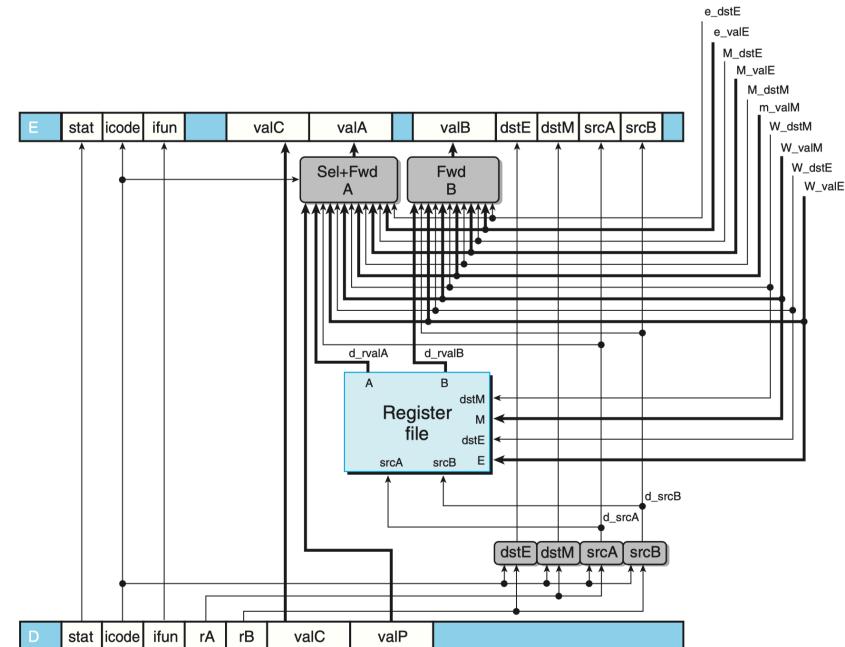
```
# 决定 d 译码阶段的 valA 的最终结果，即将存入 E_valA
word d_valA = [
    # 保存递增的 PC
    # 对于 CALL, d_valA -> E_valA -> M_valA -> 写入内存
    # 对于 JXX, d_valA -> E_valA -> M_valA
    # 跳转条件不满足（预测失败）时，同步到 f_pc
    D_icode in { ICALL, IJXX } : D_valP; # 保存递增的 PC
    d_srcA == e_dstE : e_valE; # 前递 E 阶段计算结果
    d_srcA == M_dstM : m_valM; # 前递 M 阶段读出结果
    d_srcA == M_dstE : M_valE; # 前递 M 流水线寄存器最新值
    d_srcA == W_dstM : W_valM; # 前递 W 流水线寄存器最新值
    d_srcA == W_dstE : W_valE; # 前递 W 流水线寄存器最新值
    1 : d_rvalA; # 使用从寄存器文件读取的值，r 代表 read
];
```



PIPELINE 的各阶段实现：译码阶段

pipeline hcl: decode stage

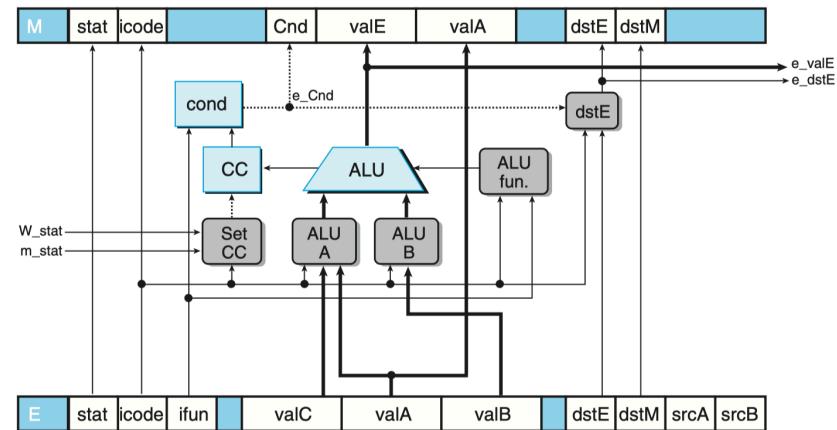
```
# 决定 d 译码阶段的 valB 的最终结果，即将存入 E_valB
word d_valB = [
    d_srcB == e_dstE : e_valE; # 前递 E 阶段计算结果
    d_srcB == M_dstM : m_valM; # 前递 M 阶段读出结果
    d_srcB == M_dstE : M_valE; # 前递 M 流水线寄存器最新值
    d_srcB == W_dstM : W_valM; # 前递 W 流水线寄存器最新值
    d_srcB == W_dstE : W_valE; # 前递 W 流水线寄存器最新值
    1 : d_rvalB; # 使用从寄存器文件读取的值, r 代表 read
];
```



PIPELINE 的各阶段实现：执行阶段

pipeline hcl: execute stage

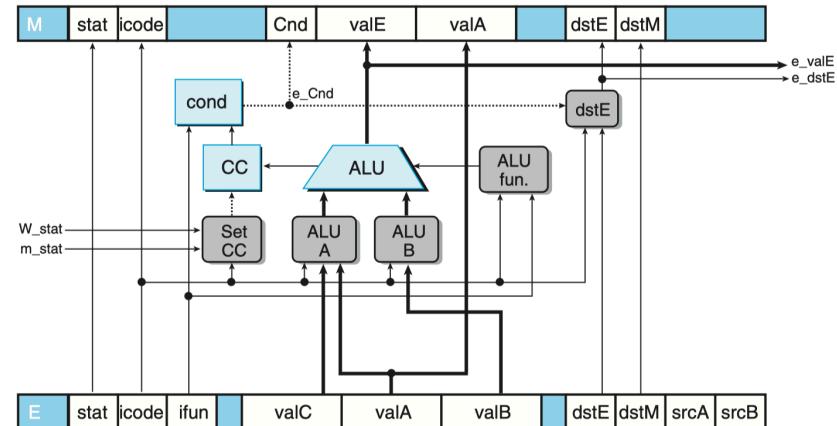
```
# 选择 ALU 的输入 A
word aluA = [
    # RRMOVQ: valA + 0; OPQ: valB OP valA
    E_icode in { IRRMOVQ, IOPQ } : E_valA;
    # IRMOVQ: valC + 0; RMMOVQ/MRMOVQ: valC + valB
    E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : E_valC;
    # CALL/PUSH: -8; RET/POP: 8
    E_icode in { ICALL, IPUSHQ } : -8;
    E_icode in { IRET, IPOPQ } : 8;
    # 其他指令不需要 ALU 的输入 A
];
# 选择 ALU 的输入 B
word aluB = [
    # 涉及栈时, 有 E_valB = RRSP, 用于计算新值
    E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
        IPUSHQ, IRET, IPOPQ } : E_valB;
    # 注意 IRMOVQ 的寄存器字节是 rA=F, 即存到 rB
    E_icode in { IRRMOVQ, IIRMOVQ } : 0;
    # 其他指令不需要 ALU 的输入 B
];
```



PIPELINE 的各阶段实现：执行阶段

pipeline hcl: execute stage

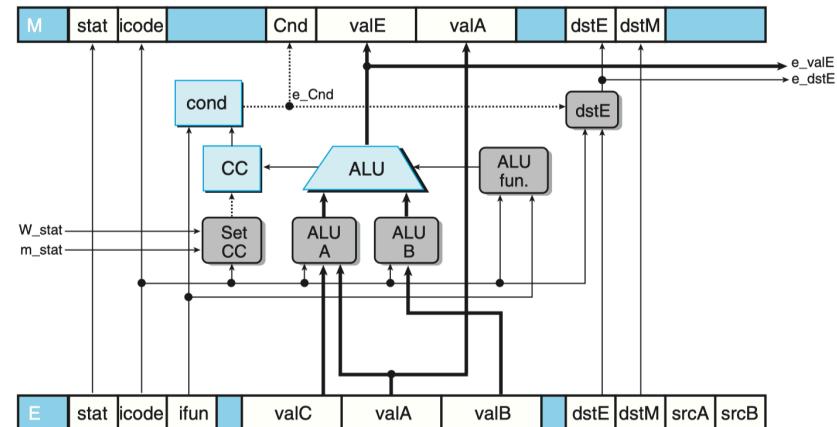
```
# 设置 ALU 功能
word alufun = [
    # 如果指令是 IOPQ, 则选择 E_ifun
    E_icode == IOPQ : E_ifun;
    # 默认选择 ALUADD
    1 : ALUADD;
];
# 是否更新条件码
# 仅在指令为 IOPQ 时更新条件码
# 且只在正常操作期间状态改变
bool set_cc = E_icode == IOPQ &&
    !m_stat in { SADR, SINS, SHLT } &&
    !W_stat in { SADR, SINS, SHLT };
```



PIPELINE 的各阶段实现：执行阶段

pipeline hcl: execute stage

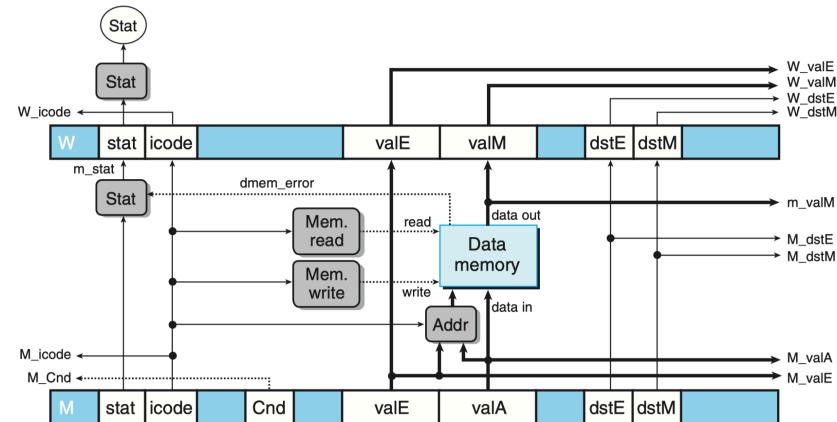
```
# 在执行阶段仅传递 valA 的去向
# E_valA -> e_valA -> M_valA
word e_valA = E_valA;
# CMOVQ 指令，与 RRMOVQ 共用 icode
# 当条件不满足时，不写入计算值到任何寄存器
word e_dstE = [
    E_icode == IRRMOVQ && !e_Cnd : RNONE
    1 : E_dstE;      # 否则选择 E_dstE
];
```



PIPELINE 的各阶段实现：访存阶段

pipeline hcl: memory stage

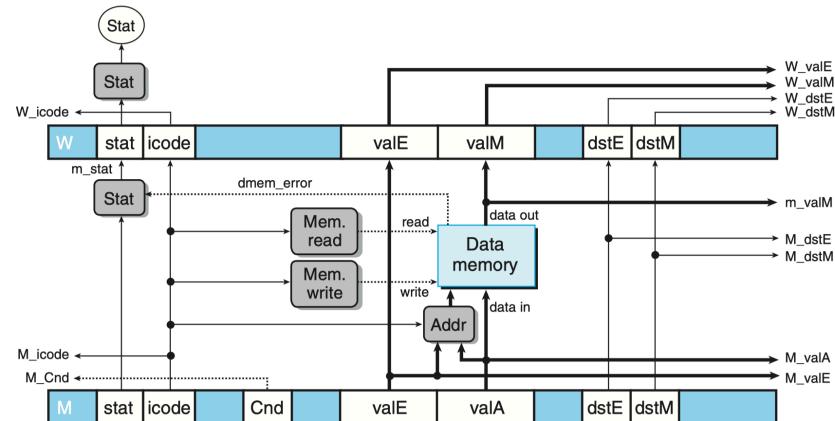
```
# 选择访存地址
word mem_addr = [
    # 需要计算阶段计算的值
    # RMMOVQ/MRMOVQ: valE = valC + valB, 这里 valA/C “统一”
    # CALL/PUSH: valE = valB(RRSP) + 8
    M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
    # 需要计算阶段不修改传递过来的值, 即栈指针旧值
    # d_valA(RRSP) -> E_valA -> M_valA
    M_icode in { IPOPQ, IRET } : M_valA;
    # 其他指令不需要访存
];
# 是否读取内存
bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };
# 是否写入内存
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };
```



PIPELINE 的各阶段实现：访存阶段

pipeline hcl: memory stage

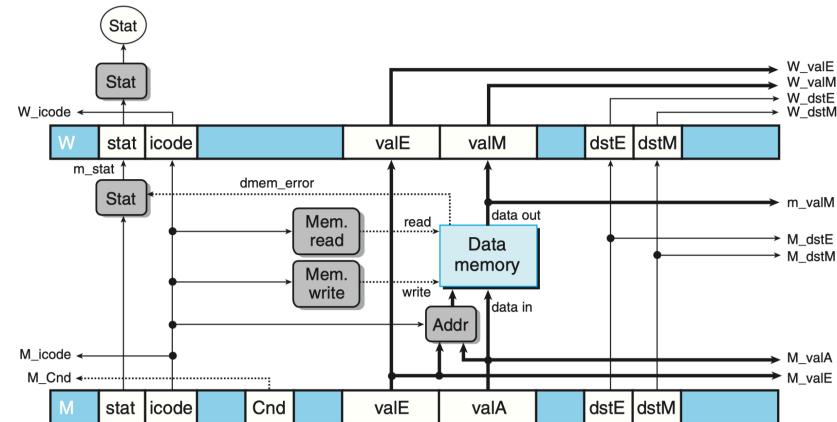
```
# 更新状态
word m_stat = [
    dmem_error : SADR; # 数据内存错误
    1 : M_stat; # 默认状态
];
```



PIPELINE 的各阶段实现：写回阶段

pipeline hcl: writeback stage

```
# W 阶段几乎啥都不干，单纯传递
# 设置 E 端口寄存器 ID
word w_dstE = W_dstE; # E 端口寄存器 ID
# 设置 E 端口值
word w_valE = W_valE; # E 端口值
# 设置 M 端口寄存器 ID
word w_dstM = W_dstM; # M 端口寄存器 ID
# 设置 M 端口值
word w_valM = W_valM; # M 端口值
# 更新处理器状态
word Stat = [
    # SBUB 全称 State Bubble, 即气泡状态
    W_stat == SBUB : SAOK;
    1 : W_stat; # 默认状态
];
```



异常处理（气泡 / 暂停）：取指阶段

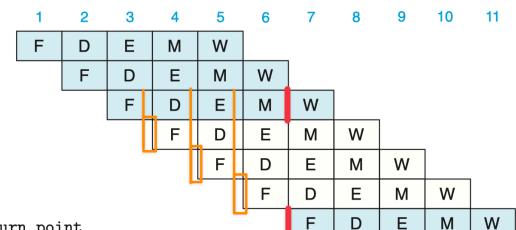
bubble / stall in fetch stage

注意：bubble 和 stall 不能同时为真。

```
# 是否向流水线寄存器 F 注入气泡?
bool F_bubble = 0; # 恒为假
# 是否暂停流水线寄存器 F?
bool F_stall =
    # 加载/使用数据冒险时，要暂停 1 个周期的译码，进而也需要暂停 1 个周期的取指
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
    # 当 ret 指令通过流水线时暂停取指，一直等到 ret 指令得到 W_valM
    IRET in { D_icode, E_icode, M_icode };
```

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

```
# prog7
      1 2 3 4 5 6 7 8 9 10 11
0x000: irmovq Stack,%edx
        F D E M W
0x00a: call proc
        F D E M W
0x020: ret
        bubble
        bubble
        bubble
0x032: irmovq $10,%edx # Return point
        F D E M W
```

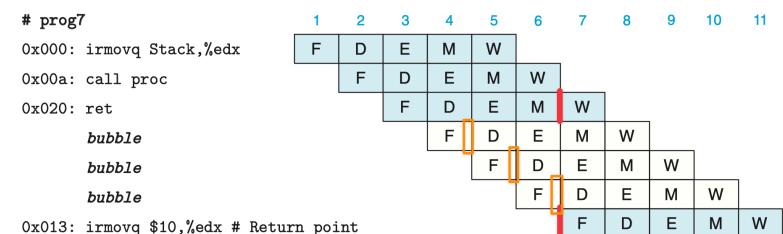
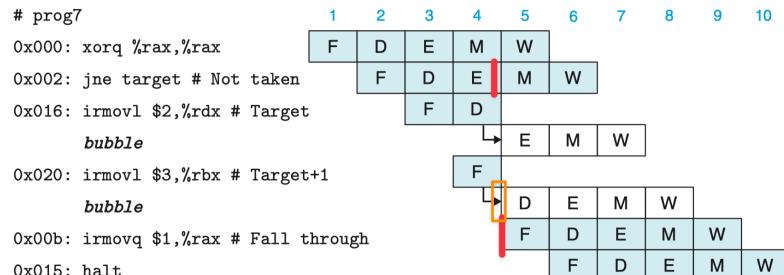


异常处理（气泡 / 暂停）：译码阶段

bubble / stall in decode stage

注意：bubble 和 stall 不能同时为真。

```
# 是否暂停流水线寄存器 D?
# 加载/使用数据冒险
bool D_stall = E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };
# 是否向流水线寄存器 D 注入气泡?
bool D_bubble =
    # 分支预测错误
    (E_icode == IJXX && !e_Cnd) ||
    # 当 ret 指令通过流水线时暂停 3 次译码阶段, 但要求不满足读取/使用数据冒险的条件
    !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) && IRET in { D_icode, E_icode, M_icode };
```

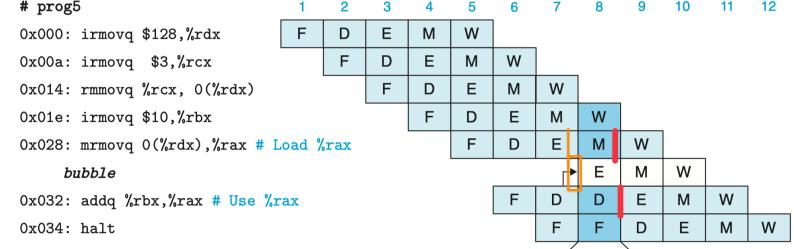
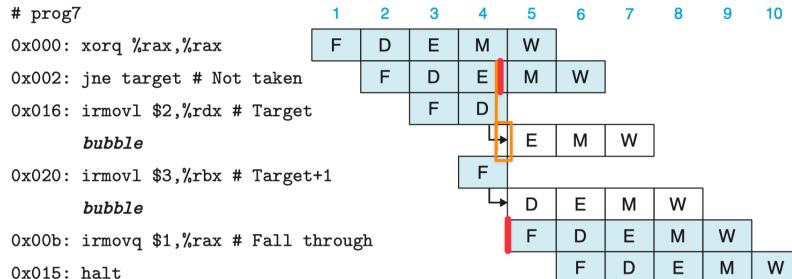


异常处理（气泡 / 暂停）：执行阶段

bubble / stall in execute stage

注意：bubble 和 stall 不能同时为真。

```
# 是否需要阻塞流水线寄存器 E?
bool E_stall = 0;
# 是否向流水线寄存器 E 注入气泡?
bool E_bubble =
    # 错误预测的分支
    (E_icode == IJXX && !e_Cnd) ||
    # 负载/使用冒险条件
    (E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB });
```



异常处理（气泡 / 暂停）：访存阶段

bubble / stall in memory stage

注意：bubble 和 stall 不能同时为真。

```
# 是否需要暂停流水线寄存器 M?  
bool M_stall = 0;  
# 是否向流水线寄存器 M 注入气泡?  
# 当异常通过内存阶段时开始插入气泡  
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
```

异常处理 (气泡 / 暂停) : 写回阶段

bubble / stall in writeback stage

注意: bubble 和 stall 不能同时为真。

```
# 是否需要暂停流水线寄存器 W?  
bool W_stall = W_stat_in { SADR, SINS, SHLT };  
# 是否向流水线寄存器 W 注入气泡?  
bool W_bubble = 0;
```

特殊的控制条件

special control conditions

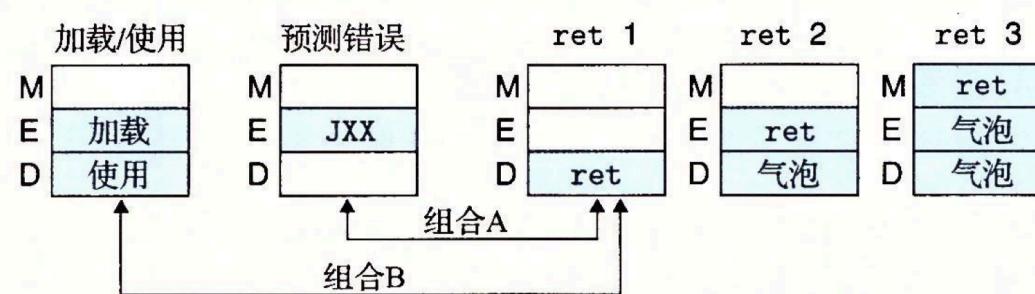


图 4-67 特殊控制条件的流水线状态。图中标明的两对情况可能同时出现

组合 A: 执行阶段中有一条不选择分支 (预测失败) 的跳转指令 JXX , 而译码阶段中有一条 RET 指令。

即, JXX 指令的跳转目标 valC 对应的内存指令是一条 RET 指令。

组合 B: 包括一个加载 / 使用冒险, 其中加载指令设置寄存器 %rsp , 然后 RET 指令用这个寄存器作为源操作数。

因为 RET 指令需要正确的栈指针 %rsp 的值去寻址, 才能从栈中弹出返回地址, 所以流水线控制逻辑应该将 RET 指令阻塞在译码阶段。

特殊的控制条件：组合 A

special control conditions: combination 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

组合情况 A 的处理与预测错误的分支相似，只不过在取指阶段是暂停。

当这次暂停结束后，在下一个周期，PC 选择逻辑会选择跳转后面那条指令的地址，而不是预测的程序计数器值。

所以流水线寄存器 F 发生了什么是没有关系的。

气泡顶掉了 RET 指令的继续传递，所以不会发生第二次暂停。

```
# 指令应从哪个地址获取
word f_pc = [
    # 分支预测错误时，从增量的 PC 取指令
    # 传递路径: D_valP -> E_valA -> M_valA
    # 条件跳转指令且条件不满足时
    M_icode == IJXX && !M_Cnd : M_valA;
    # RET 指令终于执行到回写阶段时 (即过了访存阶段)
    W_icode == IRET : W_valM;
    # 默认情况下，使用预测的 PC 值
    1 : F_predPC;
];
```

特殊的控制条件：组合 B

special control conditions: combination B

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

对于取指阶段，遇到加载/使用冒险或 RET 指令时，流水线寄存器 F 必须暂停。

对于译码阶段，这里产生了一个冲突，制逻辑会将流水线寄存器 D 的气泡和暂停信号都置为 1。这是不行的。

我们希望此时只采取针对加载/使用冒险的动作，即暂停。我们通过修改 D_bubble 的处理条件来实现这一点。

```
# 是否需要注入气泡至流水线寄存器 D
bool D_bubble =
    # 错误预测的分支
    (E_icode == IJXX && !e_Cnd) ||
    # 在取指阶段暂停，同时 ret 指令通过流水线
    # 但不存在加载/使用冒险的条件（此时使用暂停）
    !(E_icode in { IMRMOVQ, IPOPQ } &&
      E_dstM in { d_srcA, d_srcB }) &&
    # IRET 指令在 D、E、M 任何一个阶段
    IRET in { D_icode, E_icode, M_icode };
```

Emphasis

RISC与CISC

- CISC: IA32, AMD64(x86-64)
- RISC: ARM64, RISC-V, MIPS

1. 下列描述更符合(早期)RISC还是CISC?

	描述	RISC	CISC
(1)	指令机器码长度固定	✓	
(2)	指令类型多、功能丰富		✓
(3)	不采用条件码	✓	
(4)	实现同一功能，需要的汇编代码较多	✓	
(5)	译码电路复杂		✓
(6)	访存模式多样		✓
(7)	参数、返回地址都使用寄存器进行保存	✓	
(8)	x86-64		✓
(9)	MIPS	✓	
(10)	广泛用于嵌入式系统	✓	
(11)	已知某个体系结构使用 add R1, R2, R3 来完成加法运算。当要将数据从寄存器 S 移动至寄存器 D 时，使用 add S, #ZR, D 进行操作 (#ZR 是一个恒为 0 的寄存器)，而没有类似于 mov 的指令。	✓	
(12)	已知某个体系结构提供了 xlat 指令，它以一个固定的寄存器 A 为基址，以另一个固定的寄存器 B 为偏移量，在 A 对应的数组中取出下标为 B 的项的内容，放回寄存器 A 中。		✓

RISC vs. CISC

	RISC	CISC
指令(CMD)	种类少，使用频率高 指令编码长度固定(常为4个字节)	种类多，使用频率低 指令编码长度不定(x86-64的指令编码长度为1~15不等)
状态单元(Status)	寄存器数量多(最多32个) 没有条件码	寄存器数量较少 有条件码
函数调用(Procedure)	优先使用寄存器传参(也拥有用栈传参的能力) 可能隐式进行栈操作	完全依靠栈传参 只能显式进行栈操作
内存访问(Memory)	只有读取和写入两条指令 可以访问内存 只支持基址和偏移量寻址	算数运算和逻辑运算指令 也可以访问内存 支持多种寻址方法
其他(Others)	可以更充分地激发硬件性能(用体积更小的芯片跑更高的性能) 更易于进行程序性能优化	抽象程度更高，拥有很多对应高级程序语言典型操作的指令

程序员可见状态

Figure 4.1

Y86-64 programmer-visible state. As with x86-64, programs for Y86-64 access and modify the program registers, the condition codes, the program counter (PC), and the memory. The status code indicates whether the program is running normally or some special event has occurred.

RF: Program registers

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

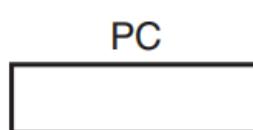
CC:
Condition
codes



Stat: Program status



DMEM: Memory



Y86-64 ISA

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	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					
cmoveq rA, rB	2	fn	rA	rB						
call Dest	8	0			Dest					
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Y86-64 ISA

Operations

addq

6	0
---	---

subq

6	1
---	---

andq

6	2
---	---

xorq

6	3
---	---

Branches

jmp

7	0
---	---

jle

7	1
---	---

jl

7	2
---	---

je

7	3
---	---

Moves

rrmovq

2	0
---	---

cmovele

2	1
---	---

cmovl

2	2
---	---

cmove

2	3
---	---

cmovne

2	4
---	---

cmovge

2	5
---	---

cmovg

2	6
---	---

Y86-64 ISA

Name	Value (hex)	Meaning
IHALT	0	Code for halt instruction
INOP	1	Code for nop instruction
IRRMVQ	2	Code for <code>rrmovq</code> instruction
IIRMOVQ	3	Code for <code>irmovq</code> instruction
IRMMOVQ	4	Code for <code>rmmovq</code> instruction
IMRMOVQ	5	Code for <code>mrmovq</code> instruction
IOPL	6	Code for integer operation instructions
IJXX	7	Code for jump instructions
ICALL	8	Code for <code>call</code> instruction
IRET	9	Code for <code>ret</code> instruction
IPUSHQ	A	Code for <code>pushq</code> instruction
IPOPQ	B	Code for <code>popq</code> instruction
FNONE	0	Default function code
RESP	4	Register ID for <code>%rsp</code>
RNONE	F	Indicates no register file access
ALUADD	0	Function for addition operation
SAOK	1	Status code for normal operation
SADR	2	Status code for address exception
SINS	3	Status code for illegal instruction exception
SHLT	4	Status code for halt

Figure 4.26 Constant values used in HCL descriptions. These values represent the encodings of the instructions, function codes, register IDs, ALU operations, and status codes.

Number	Register name	Number	Register name
0	<code>%rax</code>	8	<code>%r8</code>
1	<code>%rcx</code>	9	<code>%r9</code>
2	<code>%rdx</code>	A	<code>%r10</code>
3	<code>%rbx</code>	B	<code>%r11</code>
4	<code>%rsp</code>	C	<code>%r12</code>
5	<code>%rbp</code>	D	<code>%r13</code>
6	<code>%rsi</code>	E	<code>%r14</code>
7	<code>%rdi</code>	F	No register

Figure 4.4 Y86-64 program register identifiers. Each of the 15 program registers has an associated identifier (ID) ranging from 0 to 0xE. ID 0xF in a register field of an instruction indicates the absence of a register operand.

- IOPL改为IOPQ
- RESP改为RRSP

push&pop指令

- push: 先将 %rsp 减 8, 再压栈
- pop: 先弹栈, 再将 %rsp 加 8
- 先这么理解, 原理会在第四章学习 (与表象并不完全一致)
- call: push + jmp, 可进行间接跳转
- ret: pop + jmp

pushq %rsp

效果: 让%rsp的值入栈
入栈的值是%rsp还是%rsp - 8?

popq %rsp

效果: 弹出栈顶的值, 存储到%rsp中
存储到%rsp中的值是%rsp + 8还是(%rsp)?

Instruction	Effect	Description
pushq <i>S</i>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Push quad word
popq <i>D</i>	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Pop quad word

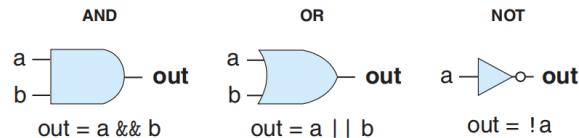
Figure 3.8 Push and pop instructions.

HCL

逻辑门

Figure 4.9

Logic gate types. Each gate generates output equal to some Boolean function of its inputs.



算术/逻辑单元ALU

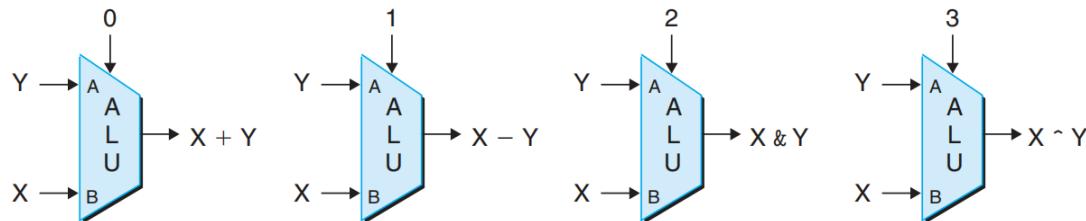


Figure 4.15 Arithmetic/logic unit (ALU). Depending on the setting of the function input, the circuit will perform one of four different arithmetic and logical operations.

Y86-64硬件结构

阶段

- 取址: Fetch
- 译码: Decode
- 执行: Execute
- 访存: Memory
- 写回: Write back
- 更新 PC: PC Update

W: 只能写入valE, valM

(rrmovq, irmovq, cmovXX 需要**+O**做ALU计算)

指令处理

Stage	
Fetch	icode:ifun rA:rB valC valP
Decode	$valA \leftarrow R[rA / \%rsp]$ $valB \leftarrow R[rB / \%rsp]$
Execute	$valE \leftarrow (valB / 0) (OP / +) (valA / valC / \pm 8)$ Set CC $Cnd \leftarrow Cond(CC, ifun)$
Memory	$(M_8[valE] \leftarrow valA / valP) / (valM \leftarrow M_8[valE / valA])$
Write back	$R[rB / \%rsp] \leftarrow (if (Cnd)) valE$ $R[rA] \leftarrow valM$
PC update	$PC \leftarrow valP / (Cnd ? valC : valP) / valC / valM$

指令处理

Stage	$OPq\ rA, rB$	$rrmovq\ rA, rB$	$irmovq\ V, rB$
Fetch	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$ $valP \leftarrow PC + 2$	$icode:ifun \leftarrow M_1[PC]$ $rA:rB \leftarrow M_1[PC + 1]$ $valC \leftarrow M_8[PC + 2]$ $valP \leftarrow PC + 10$
Decode	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	$valA \leftarrow R[rA]$	
Execute	$valE \leftarrow valB\ OP\ valA$ Set CC	$valE \leftarrow 0 + valA$	$valE \leftarrow 0 + valC$
Memory			
Write back	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$	$R[rB] \leftarrow valE$
PC update	$PC \leftarrow valP$	$PC \leftarrow valP$	$PC \leftarrow valP$

指令处理

Stage	rmmovq rA, D(rB)	mrmovq D(rB), rA
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$ $\text{valC} \leftarrow M_8[\text{PC} + 2]$ $\text{valP} \leftarrow \text{PC} + 10$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valB} \leftarrow R[\text{rB}]$
Execute	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB} + \text{valC}$
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valE}]$
Write back		$R[\text{rA}] \leftarrow \text{valM}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

指令处理

Stage	<code>cmoveXX rA, rB</code>
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $rA:rB \leftarrow M_1[\text{PC} + 1]$ $\text{valP} \leftarrow \text{PC} + 2$
Decode	$\text{valA} \leftarrow R[rA]$
Execute	$\text{valE} \leftarrow 0 + \text{valA}$ $\text{Cnd} \leftarrow \text{Cond(CC, ifun)}$
Memory	
Write back	$\text{if } (\text{Cnd}) R[rB] \leftarrow \text{valE}$
PC update	$\text{PC} \leftarrow \text{valP}$

指令处理

Stage	pushq rA	popq rA
Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC} + 1]$
	$\text{valP} \leftarrow \text{PC} + 2$	$\text{valP} \leftarrow \text{PC} + 2$
Decode	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\%rsp]$	$\text{valA} \leftarrow R[\%rsp]$ $\text{valB} \leftarrow R[\%rsp]$
Execute	$\text{valE} \leftarrow \text{valB} + (-8)$	$\text{valE} \leftarrow \text{valB} + 8$
Memory	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valA}]$
Write back	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$	$R[\%rsp] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
PC update	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$

Overview

基本概念

1. 吞吐量：单位时间内完成的指令数（单位：GIPS，每秒十亿条指令）
2. 延迟：完成一条指令需要的时间（单位：ps，皮秒）
3. CPI：执行一条指令所需要的平均时钟周期数（单位：时钟周期）

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

$$\text{CPI} = 1.0 + lp + mp + rp$$

- lp - 加载处罚, mp - 预测错误分支处罚, rp - 返回处罚

Cause	Name	Instruction frequency	Condition frequency	Bubbles	Product
Load/use	lp	0.25	0.20	1	0.05
Mispredict	mp	0.20	0.40	2	0.16
Return	rp	0.02	1.00	3	0.06
Total penalty					0.27

4. IPC：每周期执行指令平均数（单位：指令数）

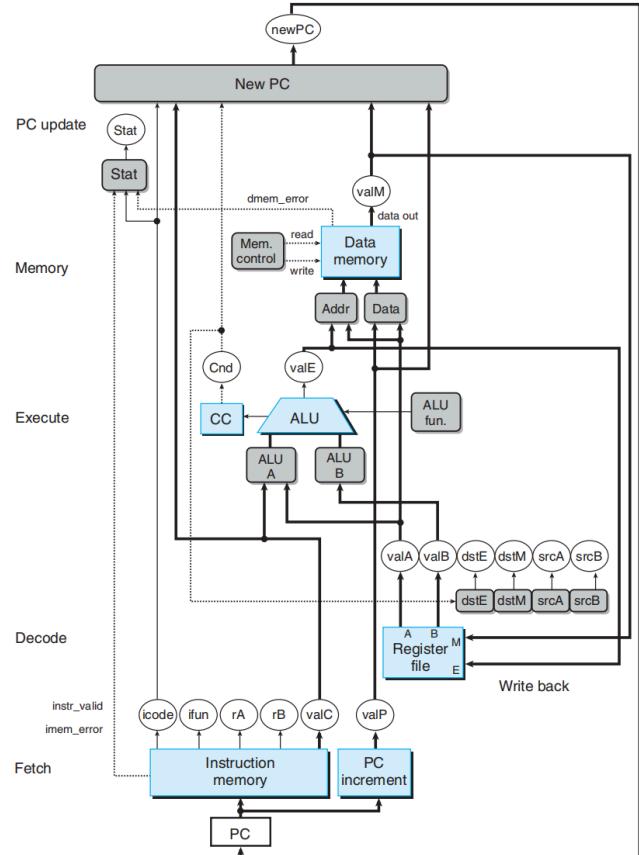


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

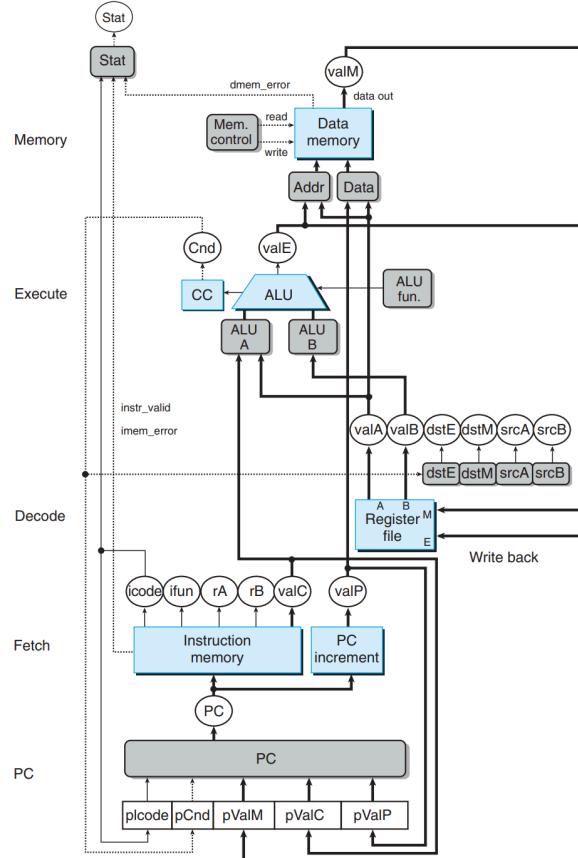


Figure 4.40 SEQ+ hardware structure. Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.

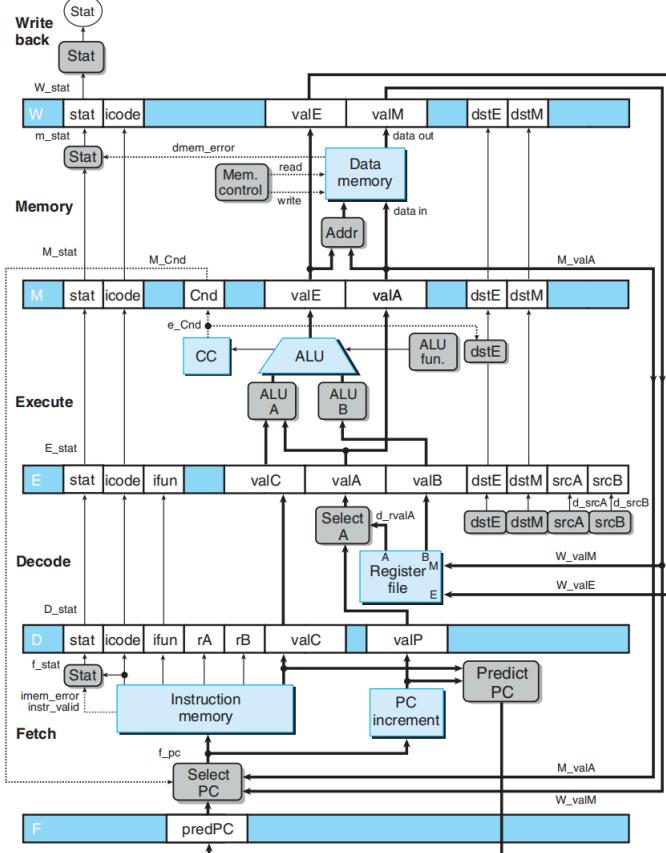


Figure 4.41 Hardware structure of PIPE—, an initial pipelined implementation. By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

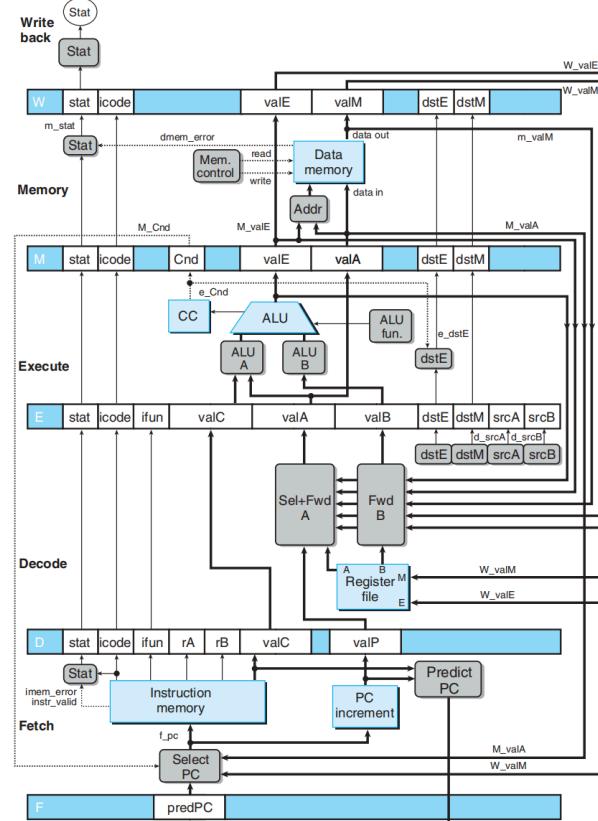


Figure 4.52 Hardware structure of PIPE, our final pipelined implementation. The additional bypassing paths enable forwarding the results from the three preceding instructions. This allows us to handle most forms of data hazards without stalling the pipeline.

Details

SEQ、PIPE的具体实现图、HCL代码都需要详细掌握

此处不再说明，详见【补充资料】部分

SEQ->SEQ+

前置PC

- 电路重定时：改变状态表示而不改变逻辑
- 目的：平衡一个流水线各个阶段之间的延迟
- 在 SEQ+ 的实现里，PC update 的时期从周期的最后被提到了最前，更加接近流水线的形态

SEQ+ -> PIPE-

阶段划分

在 Y86-64 的实现中：

- 正常指令指令默认预测 PC 为下一条指令的地址；
- call 指令和 jxx 指令默认预测 PC 为跳转后地址；
- ret 指令不进行任何预测，直到其对应的写回完成

插入流水线寄存器：分别插入了5个流水线寄存器用来保存后续阶段所需的信号，编号为 F、D、E、M 和 W

- **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

寄存器顺序：F—f—D—d—E—e—M—m—W

PIPE->PIPE

处理冒险

- 硬件：暂停和气泡
 - stall 能将指令阻塞在某个阶段
 - bubble 能使得流水线继续运行，但是不会改变当前阶段的寄存器、内存、条件码或程序状态
- 结构冒险
 - 计算的多时钟周期：采用独立于主流水线的特殊硬件功能单元来处理较为复杂的操作（一个功能单元执行整数乘法和除法，一个功能单元执行浮点操作）
 - 访存的多时钟周期：
 - 翻译后备缓冲器 (TLB) + 高速缓存 (Cache)：实现一个时钟周期内读指令并读或写数据
 - 缺页 (page fault) 异常信号：指令暂停+磁盘到主存传送+指令重新执行

PIPE->PIPE

处理冒险

■ 数据冒险

- **前后使用数据冒险**: 在处理器中, `valA` 和 `valB` 一共有5个转发源:
 - `e_valE` : 在执行阶段, ALU中计算得到的结果 `valE` , 通过 `E_dstE` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
 - `M_valE` : 将ALU计算的结果 `valE` 保存到流水线寄存器M中, 通过 `M_dstE` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
 - `m_valM` : 在访存阶段, 从内存中读取的值 `valM` , 通过 `M_dstM` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
 - `W_valM` : 将内存中的值 `valM` 保存到流水线寄存器W中, 通过 `W_dstM` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。
 - `W_valE` : 将ALU计算的结果 `valE` 保存到流水线寄存器W中, 通过 `W_dstE` 与 `d_srcA` 和 `d_src_B` 进行比较决定是否转发。

PIPE->PIPE

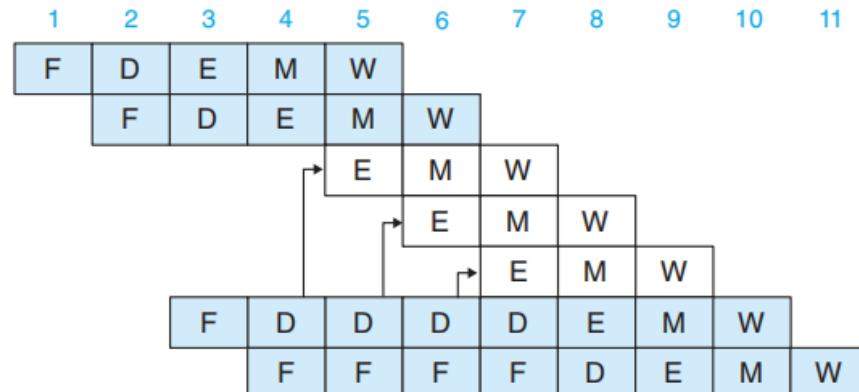
处理冒险

- 数据冒险

- 用暂停来避免数据冒险

- 插入一段自动产生的 `nop` 指令
 - 该方法指令要停顿最少一个最多三个时钟周期，严重降低整体的吞吐量

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



PIPE->PIPE

处理冒险

- 数据冒险
 - 加载/使用数据冒险

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

Condition	Pipeline register				
	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

PIPE->PIPE

处理冒险

- 控制冒险

- ret指令（不预测）：删除后续操作——插入3个bubble

```
# prog7
 1 2 3 4 5 6 7 8 9 10 11
0x000: irmovq Stack,%edx
0x00a: call proc
0x020: ret
    bubble
    bubble
    bubble
0x013: irmovq $10,%edx # Return point
```

The diagram illustrates the insertion of three bubbles after the ret instruction. The pipeline stages are F (Fetch), D (Decode), E (Execute), M (Memory Access), and W (Write). The original code has a ret instruction at stage 6. Three bubbles are inserted after it, pushing the subsequent irmovq instruction to stage 11.

- 跳转指令（预测）：删除后续操作——插入2个bubble

```
# 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+
    bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

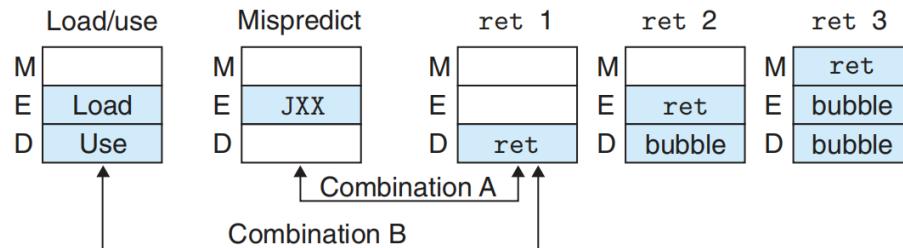
The diagram illustrates the insertion of two bubbles after the jne instruction. The pipeline stages are F (Fetch), D (Decode), E (Execute), M (Memory Access), and W (Write). The original code has a jne instruction at stage 2. Two bubbles are inserted after it, pushing the subsequent irmovl instruction to stage 10.

PIPE->PIPE

处理冒险

- 检验自洽

- 控制条件组合——有限性组合： **Combination A + B**
 - Combination A: **ret位于不选择分支** ——简单叠加
 - Combination B: **加载/使用+ret** ——取”**stall**”
 - 加载互锁核心思想：通过暂停+转发组合实现
 - 合理性： Install后，下一条指令无法进入寄存器，当前指令因为bubble并未成功下传
 - 有效性： Install后，当前指令ret依然存在于流水线中，加载/使用语句后可进一步执行



PIPE->PIPE

处理冒险

- 检验自洽

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	Pipeline register				
	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

PIPE->PIPE

处理冒险

- 异常处理
 - 内部异常：
 - HLT：执行halt指令
 - ADR：从非法内存地址读或向非法内存地址写
 - INS：非法指令
 - 外部异常
 - 系统重启
 - I/O设备请求
 - 硬件故障
- 要求：异常指令之前的所有指令已经完成，后续的指令都不能修改条件码寄存器和内存。

PIPE- ->PIPE

处理冒险

- 异常处理

1. 当同时多条指令引起异常时，处理器应该向操作系统报告哪个异常？

基本原则：由流水线中最深的指令引起的异常，表示该指令越早执行，优先级最高。

2. 在分支预测中，当预测分支中出现了异常，而后由于预测错误而取消该指令时，需要取消异常。
3. 如何处理不同阶段更新系统状态不同部分的问题？

- 异常发生时，记录指令状态，继续取指、译码、执行
- 异常到达 **访存阶段**：
 1. 执行阶段，禁止设置条件码 ($set_cc \leftarrow m_stat, W_stat$)
 2. 访存阶段，插入气泡，禁止写入内存
 3. 写回阶段，暂停写回，即暂停流水线

Homework Review

HW4

学号	HW-分数
2300012932	10
2300012935	10
2300012950	10
2300012951	10
2300013083	10
2300013115	10
2300013158	10
2300013201	10
2300013222	10
2300013230	10
2300013272	10
2300017788	10
2300094610	10

- 可能确实不存在很好的办法验证Y86-64的正确性
- 暂不存在 C 到 Y86-64 的转换工具
- 但可以通过手动模拟的方式，通过Y86-64模拟器验证正确性（如果你真的想这么干）

Exercises

Processor Arch: ISA & Logic

E1

10. 下面有三组对于指令集的描述，它们分别符合 ①_____，②_____，③_____ 的特点。

- ① 某指令集中，只有两条指令能够访问内存。
- ② 某指令集中，指令的长度都是 4 字节。
- ③ 某指令集中，可以只利用一条指令完成字符串的复制，也可以只利用一条指令查找字符串中第一次出现字母 K 的位置。

- A. CISC, CISC, CISC
- B. RISC, RISC, CISC
- C. RISC, CISC, RISC
- D. CISC, RISC, RISC

E1

10. 下面有三组对于指令集的描述，它们分别符合 ①_____，②_____，③_____ 的特点。

- ① 某指令集中，只有两条指令能够访问内存。
- ② 某指令集中，指令的长度都是 4 字节。
- ③ 某指令集中，可以只利用一条指令完成字符串的复制，也可以只利用一条指令查找字符串中第一次出现字母 K 的位置。

- A. CISC, CISC, CISC
- B. RISC, RISC, CISC
- C. RISC, CISC, RISC
- D. CISC, RISC, RISC

【答】B。 ①的访存模式单一，更加符合 RISC 的特点；②的指令长度固定，更加符合 RISC 的特点；③的指令功能丰富而复杂，更加符合 CISC 的特点。

E2

9. 请比较 RISC 和 CISC 的特点，回答下述问题：

假设编译技术处于发展初期，程序员更愿意使用汇编语言编程来解决实际问题，那么程序员会更倾向于选用 _____ ISA。

假设你设计的处理器速度非常快，但存储系统设计使得取指令的速度非常慢（也许是处理单元的十分之一）。这时你会更倾向于选用 _____ ISA。

- A) RISC、RISC
- B) CISC、CISC
- C) RISC、CISC
- D) CISC、RISC

E2

9. 请比较 RISC 和 CISC 的特点，回答下述问题：

假设编译技术处于发展初期，程序员更愿意使用汇编语言编程来解决实际问题，那么程序员会更倾向于选用 _____ ISA。

假设你设计的处理器速度非常快，但存储系统设计使得取指令的速度非常慢（也许是处理单元的十分之一）。这时你会更倾向于选用 _____ ISA。

- A) RISC、RISC
- B) CISC、CISC
- C) RISC、CISC
- D) CISC、RISC

答案：B

//知识点 1：CISC 有更多的指令，有些更接近高级语言

//知识点 2：CISC 的指令功能更复杂，指令执行需要更多的周期，一定程度上可以平衡处理速度与指令访存的速度差异。不过，通常处理器设计中，主要通过多层次的存储体系结构来弥补两者之的速度差异。

E3

9. 下面关于 RISC 和 CISC 的描述中，正确的是：
- A. CISC 和早期 RISC 在寻址方式上相似，通常只有基址和偏移量寻址
 - B. CISC 指令集可以对内存和寄存器操作数进行算术和逻辑运算，而 RISC 只能对寄存器操作数进行算术和逻辑运算
 - C. CISC 和早期的 RISC 指令集都有条件码，用于条件分支检测
 - D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问

E3

9. 下面关于 RISC 和 CISC 的描述中，正确的是：
- A. CISC 和早期 RISC 在寻址方式上相似，通常只有基址和偏移量寻址
 - B. CISC 指令集可以对内存和寄存器操作数进行算术和逻辑运算，而 RISC 只能对寄存器操作数进行算术和逻辑运算
 - C. CISC 和早期的 RISC 指令集都有条件码，用于条件分支检测
 - D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问

答案： B

E4

8. 下面对指令系统的描述中，错误的是：（ ）

- A. CISC 指令系统中的指令数目较多，有些指令的执行周期很长；而 RISC 指令系统中通常指令数目较少，指令的执行周期都较短。
- B. CISC 指令系统中的指令编码长度不固定；RISC 指令系统中的指令编码长度固定，这样使得 CISC 机器可以获得了更短的代码长度。
- C. CISC 指令系统支持多种寻址方式，RISC 指令系统支持的寻址方式较少。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问。

E4

8. 下面对指令系统的描述中，错误的是：（ ）

- A. CISC 指令系统中的指令数目较多，有些指令的执行周期很长；而 RISC 指令系统中通常指令数目较少，指令的执行周期都较短。
- B. CISC 指令系统中的指令编码长度不固定；RISC 指令系统中的指令编码长度固定，这样使得 CISC 机器可以获得了更短的代码长度。
- C. CISC 指令系统支持多种寻址方式，RISC 指令系统支持的寻址方式较少。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数，避免了不必要的存储访问。

答案： D

E5

11. 下面有关指令系统设计的描述正确的是：

- A. 采用 CISC 指令比 RISC 指令代码更长。
- B. 采用 CISC 指令比 RISC 指令运行时间更短
- C. 采用 CISC 指令比 RISC 指令译码电路更加复杂
- D. 采用 CISC 指令比 RISC 指令的流水线吞吐更高

E5

11. 下面有关指令系统设计的描述正确的是：

- A. 采用 CISC 指令比 RISC 指令代码更长。
- B. 采用 CISC 指令比 RISC 指令运行时间更短
- C. 采用 CISC 指令比 RISC 指令译码电路更加复杂
- D. 采用 CISC 指令比 RISC 指令的流水线吞吐更高

答案：C，CISC 的比 RISC 代码复杂（如不定长），因此译码也更复杂，优点是代码更短。运行时间和吞吐都谁更高要依据实际应用。

E6

11、关于 RISC 和 CISC 的描述，正确的是：

- A. CISC 指令系统的指令编码可以很短，例如最短的指令可能只有一个字节，因此 CISC 的取指部件设计会比 RISC 更为简单。
- B. CISC 指令系统中的指令数目较多，因此程序代码通常会比较长；而 RISC 指令系统中通常指令数目较少，因此程序代码通常会比较短。
- C. CISC 指令系统支持的寻址方式较多，RISC 指令系统支持的寻址方式较少，因此用 CISC 在程序中实现访存的功能更容易。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数。

E6

11、关于 RISC 和 CISC 的描述，正确的是：

- A. CISC 指令系统的指令编码可以很短，例如最短的指令可能只有一个字节，因此 CISC 的取指部件设计会比 RISC 更为简单。
- B. CISC 指令系统中的指令数目较多，因此程序代码通常会比较长；而 RISC 指令系统中通常指令数目较少，因此程序代码通常会比较短。
- C. CISC 指令系统支持的寻址方式较多，RISC 指令系统支持的寻址方式较少，因此用 CISC 在程序中实现访存的功能更容易。
- D. CISC 机器中的寄存器数目较少，函数参数必须通过栈来进行传递；RISC 机器中的寄存器数目较多，只需要通过寄存器来传递参数。

答案：C

考查对 CISC 和 RISC 基本特点的描述，A 和 B 都是描述反了，D 则是太绝对，RISC 也有可能用栈来传递参数。

E7

4. 下述关于 RISC 和 CISC 的讨论，哪个是错误的
- A. RISC 指令集包含的指令数量通常比 CISC 的少
 - B. RISC 的寻址方式通常比 CISC 的寻址方式少
 - C. RISC 的指令长度通常短于 CISC 的指令长度
 - D. 手机处理器通常采用 RISC，而 PC 采用 CISC

E7

4. 下述关于 RISC 和 CISC 的讨论，哪个是错误的
- A. RISC 指令集包含的指令数量通常比 CISC 的少
 - B. RISC 的寻址方式通常比 CISC 的寻址方式少
 - C. RISC 的指令长度通常短于 CISC 的指令长度
 - D. 手机处理器通常采用 RISC，而 PC 采用 CISC
4. C。CISC 变长指令，有不少指令的长度是要比 RISC 短的。当代手机常常采用 ARM 架构，这是 RISC；实际上对能耗要求高或者结构简单的设备一般都用 RISC。

E8

5. 下面对指令系统的描述中，错误的是：（ ）
- A. 通常 CISC 指令集中的指令数目较多，有些指令的执行周期很长；而 RISC 指令集中指令数目较少，指令的执行周期较短。
 - B. 通常 CISC 指令集中的指令长度不固定；RISC 指令集中的指令长度固定。
 - C. 通常 CISC 指令集支持多种寻址方式，RISC 指令集支持的寻址方式较少。
 - D. 通常 CISC 指令集处理器的寄存器数目较多，RISC 指令集处理器的寄存器数目较少。

E8

5. 下面对指令系统的描述中，错误的是：（ ）
- A. 通常 CISC 指令集中的指令数目较多，有些指令的执行周期很长；而 RISC 指令集中指令数目较少，指令的执行周期较短。
 - B. 通常 CISC 指令集中的指令长度不固定；RISC 指令集中的指令长度固定。
 - C. 通常 CISC 指令集支持多种寻址方式，RISC 指令集支持的寻址方式较少。
 - D. 通常 CISC 指令集处理器的寄存器数目较多，RISC 指令集处理器的寄存器数目较少。
5. D。送分题，RISC 中寄存器一般更多。

E9

1. 下列描述更符合（早期）RISC 还是 CISC？

	描述
(1)	指令机器码长度固定
(2)	指令类型多、功能丰富
(3)	不采用条件码
(4)	实现同一功能，需要的汇编代码较多
(5)	译码电路复杂
(6)	访存模式多样
(7)	参数、返回地址都使用寄存器进行保存
(8)	x86-64
(9)	MIPS
(10)	广泛用于嵌入式系统
(11)	已知某个体系结构使用 add R1,R2,R3 来完成加法运算。当要将数据从寄存器 S 移动至寄存器 D 时，使用 add S,#ZR,D 进行操作（#ZR 是一个恒为 0 的寄存器），而没有类似于 mov 的指令。
(12)	已知某个体系结构提供了 xlat 指令，它以一个固定的寄存器 A 为基址，以另一个固定的寄存器 B 为偏移量，在 A 对应的数组中取出下标为 B 的项的内容，放回寄存器 A 中。

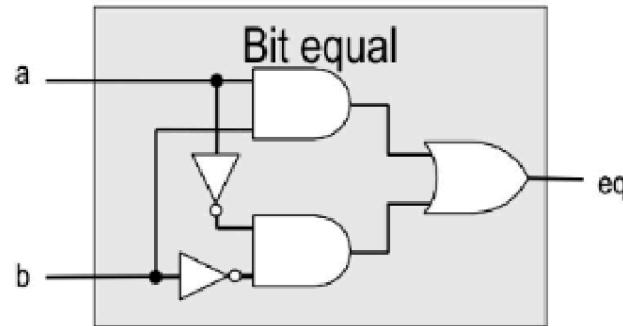
E9

1. 下列描述更符合（早期）RISC 还是 CISC？

	描述	RISC	CISC
(1)	指令机器码长度固定	✓	
(2)	指令类型多、功能丰富		✓
(3)	不采用条件码	✓	
(4)	实现同一功能，需要的汇编代码较多	✓	
(5)	译码电路复杂		✓
(6)	访存模式多样		✓
(7)	参数、返回地址都使用寄存器进行保存	✓	
(8)	x86-64		✓
(9)	MIPS	✓	
(10)	广泛用于嵌入式系统	✓	
(11)	已知某个体系结构使用 add R1, R2, R3 来完成加法运算。当要将数据从寄存器 S 移动至寄存器 D 时，使用 add S, #ZR, D 进行操作（#ZR 是一个恒为 0 的寄存器），而没有类似于 mov 的指令。	✓	
(12)	已知某个体系结构提供了 xlat 指令，它以一个固定的寄存器 A 为基址，以另一个固定的寄存器 B 为偏移量，在 A 对应的数组中取出下标为 B 的项的内容，放回寄存器 A 中。		✓

E10

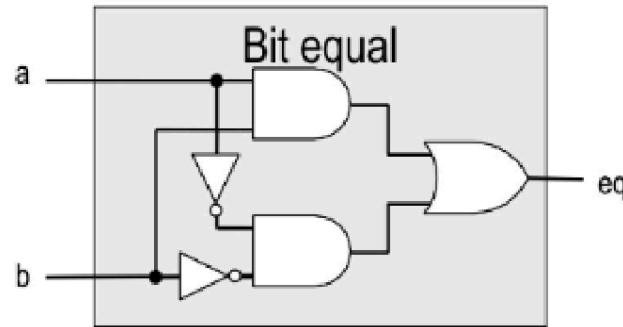
10. 对应下述组合电路的正确 HCL 表达式为：



- A. Bool eq = (a or b) and (!a or !b)
- B. Bool eq = (a and b) or (!a and !b)
- C. Bool eq = (a or !b) and (!a or b)
- D. Bool eq = (a and !b) or (!a and b)

E10

10. 对应下述组合电路的正确 HCL 表达式为：

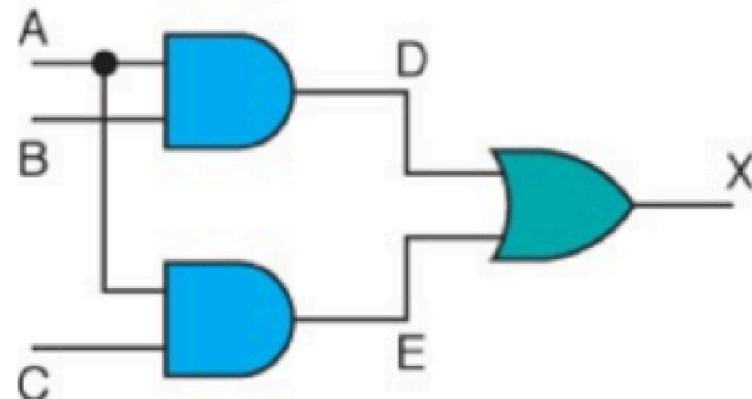


- A. Bool eq = (a or b) and (!a or !b)
- B. Bool eq = (a and b) or (!a and !b)
- C. Bool eq = (a or !b) and (!a or b)
- D. Bool eq = (a and !b) or (!a and b)

答案： B

E11

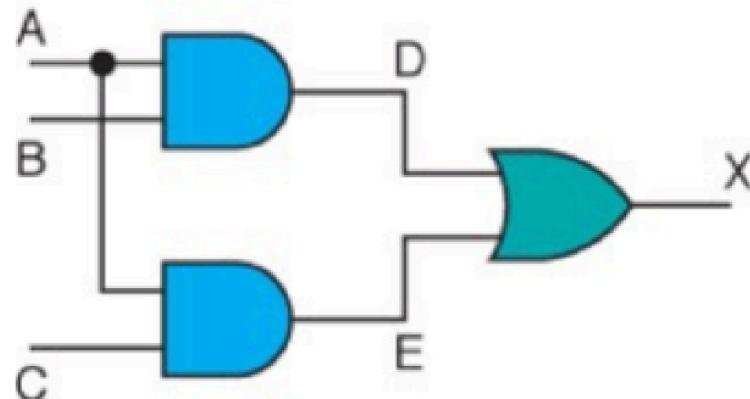
9、对应下述组合电路的正确 HCL 表达式为



- A. Bool X = (A || B) && (A || C)
- B. Bool X = A || (B && C)
- C. Bool X = A && (B || C)
- D. Bool X = A || B || C

E11

9、对应下述组合电路的正确 HCL 表达式为

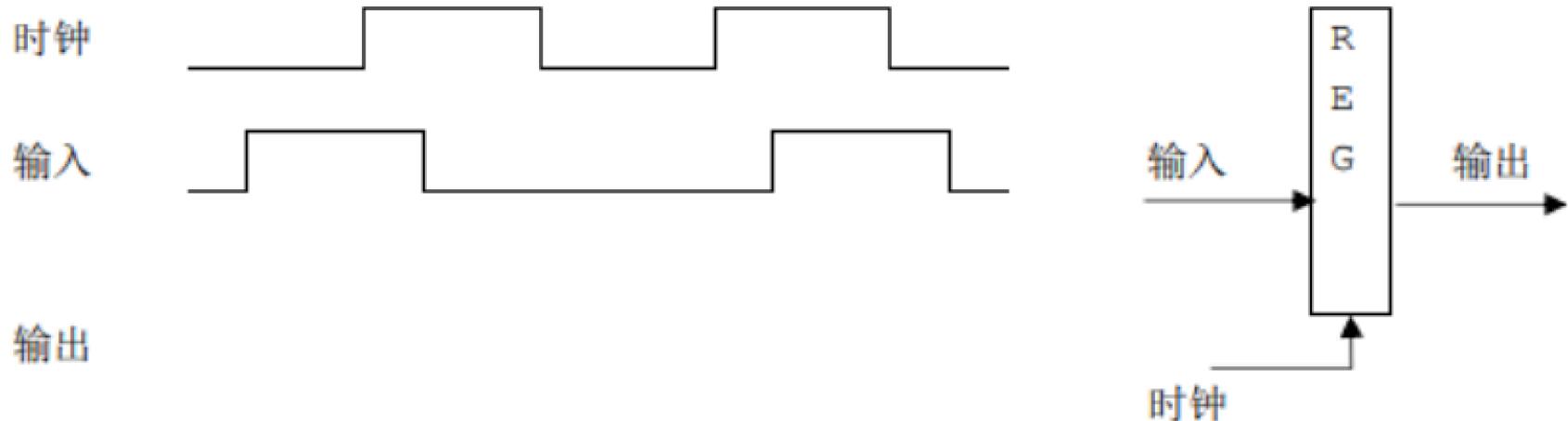


- A. Bool X = (A || B) && (A || C)
- B. Bool X = A || (B && C)
- C. Bool X = A && (B || C)
- D. Bool X = A || B || C

答案: C

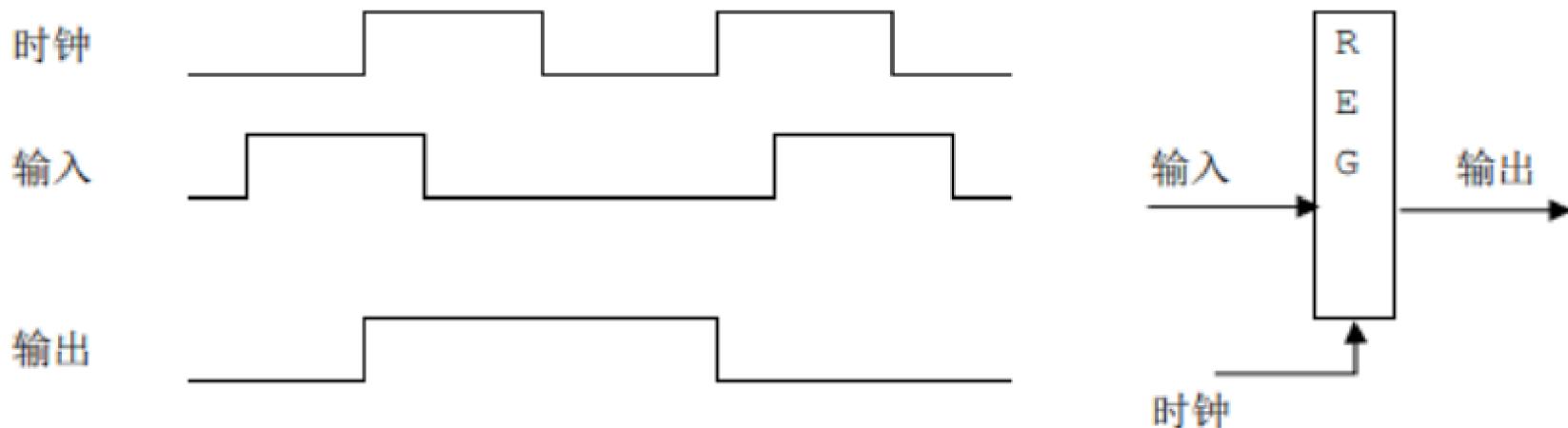
E12

3. 下列寄存器在时钟上升沿锁存数据，画出输出的电平（忽略建立/保持时间）



E12

3. 下列寄存器在时钟上升沿锁存数据，画出输出的电平（忽略建立/保持时间）



E13

第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
$(x \geq 0) \ \ (3*x < 0)$	Y N
$(x \geq 0) \ \ (x < ux)$	Y N
$((x \gg 1) \ll 1) \leq x$	Y N
$((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2$	Y N
$(x - y > 0) == ((y+x+1)>>31 == 1)$	Y N
$dx + dy == (double)(y+x)$	Y N
$dx + dy + dz == dz + dy + dx$	Y N
$(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))$	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位 (s): 1-bit; 阶码字段 (exp): 6-bit; 小数字段 (frac): 5-bit。

- 1) 若将该格式下能表示的所有正规格数从小到大依次排列，则
相邻两数之间差值的最小值为 _____，最大值为 _____

- 2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

E13

第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
(x >= 0) (3*x < 0)	Y N
(x >= 0) (x < ux)	Y N
((x >> 1) << 1) <= x	Y N
((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2	Y N
(x - y > 0) == ((y+x+1)>>31 == 1)	Y N
dx + dy == (double)(y+x)	Y N
dx + dy + dz == dz + dy + dx	Y N
(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位(s): 1-bit; 阶码字段(exp): 6-bit; 小数字段(frac): 5-bit。

1) 若将该格式下能表示的所有正规格数从小到大依次排列，则

相邻两数之间差值的最小值为_____，最大值为_____

2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

答案

N 考虑 $x = (1<<31)>>1$

N 考虑 $x = -1$

Y

N 考虑 $x = -1 \ y = 0$

N 考虑 $x-y = T_{min}$

N 考虑 $x+y$ 溢出

Y 恒成立，每一个 int 型都可以由一个 double 型精确表示

Y 恒成立

答案：

1)

2^{-35}

2^{26}

2)

1 100011 00000 (发生进位)

0 111011 00010 (发生舍入)

1 111111 00000 (溢出, $+\infty$)

E14

第三题 (20 分)

(1) 观察下面C语言函数和它相应的x86-64汇编代码

```
int foo(int x, int i)
{
    switch(i)
    {
        case 1:
            x -= 10;
        case 2:
            x *= 8;
            break;
        case 3:
            x += 5;
        case 5:
            x /= 2;
            break;
        case 0:
            x *= 1;
            default:
                x += i;
    }
    return x;
}

00000000004004a8 <foo>:
4004a8: mov %edi,%edx
4004aa: cmp $0x5,%esi
4004ad: ja 4004d4 <foo+0x2c>
4004af: mov %esi,%eax
4004b1: jmpq *0x400690(%rax,8)
4004b8: sub $0xa,%edx
4004bb: shr $0x3,%edx
4004be: jmp 4004d6 <foo+0xe>
4004c0: add $0x5,%edx
4004c3: mov %edx,%eax
4004c5: shr $0x1f,%eax
4004c8: lea (%rdx,%rax,1),%eax
4004cb: mov %eax,%edx
4004cd: sar %edx
4004cf: jmp 4004d6 <foo+0xe>
4004d1: and $0x1,%edx
4004d4: add %esi,%edx
4004d6: mov %edx,%eax
4004d8: retq
```

调用gdb命令`x/kg $rsp`将会检查从`rsp`中的地址开始的k个8字节字，请填写下面gdb命令的输出（每空一分）。

>(gdb) x/6g 0x400690

0x400690: 0x_____ 0x_____
0x4006a0: 0x_____ 0x_____
0x4006b0: 0x_____ 0x_____

(2) 右边的汇编代码是由左边程序中的`m`函数编译而成。回答如下问题。

<pre>typedef struct _list { struct _list* next; int value; } list; int m(list* p) { int r = 100; while (_①_) { r = __②__; p = __③__; } if (p != 0) r = __④__; return r; }</pre>	<pre>m: testq %rdi, %rdi je .L6 movq (%rdi), %rdx movl \$100, %eax testq %rdx, %rdx jne .L4 jmp .L3 .L14: movq (%rdi), %rdx testq %rdx, %rdx je .L3 movl 8(%rdx), %r8d __⑤__ 8(%rdi), %r8d __⑥__ %r8d, %eax movq (%rdi), %rdi testq %rdi, %rdi jne .L14 ret .L3: __⑦__ 8(%rdi), %eax ret .L6: movl \$100, %eax ret</pre>
--	---

已知访存延迟 1 个指令周期。不访存的时候 `addl` 延迟 4 个指令周期, `imull` 延迟 8 个指令周期, 其他指令延迟 1 个指令周期。指令访存时延迟的执行周期为不访存的时候延迟的指令周期和访存延迟的周期之和。函数`m`中的循环在处理链表`p`的时候 CPE 为 4 个指令周期。⑤处的指令为`movl`, `addl`, `imull`中的一个。请填写①-⑦处的代码。

① _____

② _____

③ _____

④ _____

⑤ _____

E14

答案：

(1) 每空 1 分

0x400690:	0x00000000004004d1	0x00000000004004b8
0x4006a0:	0x00000000004004bb	0x00000000004004c0
0x4006b0:	0x00000000004004d4	0x00000000004004c3

(2) ⑤空 2 分, 其他 3 分

① $p \neq 0 \ \&\ p->next \neq 0$

② $r * (p->value * p->next->value)$ 注意：答案中运算可以交换，但不可结合

③ $p->next->next$

④ $r * p->value$

⑤ imull (因为每次处理两个元素, 所以关键路径的时间周期为 8。关键路径要么为两次访存和两次 mov, 要么为⑤处的运算。前者不可能达到 8, 所以只能是 imull。)

给分说明：如果②④的答案和⑤一致，但是⑤答错了就②④各给 2 分。

E15

第三题 (15 分)

分析下面C语言程序和相应的x86-64汇编程序。其中缺失部分代码（被遮挡），请在对应的横线上填写缺失的内容。

```
#include <stdio.h>
#include "string.h"

void myprint(char *str)
{
    char buffer[16];
    [REDACTED](buffer,str);
    printf("%s \n",buffer);
}

void alert(void)
{
    printf("[REDACTED] \n");
}

int main(int argc,char *argv[])
{
    myprint("123456712345671234567\xaa\x84\x04\x08");
    return 0;
}
*****.section .rodata
.LCO:
    .string "[REDACTED]"
    .text
    .globl myprint
    .type myprint, @function
myprint:
.LFBO:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    subq $48, %rbp
    movq %rdi, -40(%rbp)
    movq %fs:40, [REDACTED]
    movq %rax, -8(%rbp)
    xorl %eax, %eax
    movq %rax, %rdx
    leaq -32(%rbp), %rax
    movq %rdx, [REDACTED]
    [REDACTED]
    call strcpy
    [REDACTED] -32(%rbp), %rax
    movq %rax, %rsi
    movi $LCO, %edi
    movi $0, %eax
    call printf
```

```
nop [REDACTED], %rax
    [REDACTED]
    xorq [REDACTED]
    je [REDACTED]
    call _stack_chk_fail
.L2:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFBO:
    .size myprint, .-myprint
    .section .rodata
.LC1:
    .string "Where am I?"
    .text
    .globl alert
    .type alert, @function
alert:
.LFBI:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq %rsi, %rbp
    .cfi_def_cfa_register 6
    movl $.LC1, %edi
    call puts
    nop
    popq %rbp
    .cfi_def_cfa 7, 8
    [REDACTED]
    .cfi_endproc
.LFE1:
    .size alert, .-alert
    .section .rodata
    .align 8
.LC2:
    .string "1234567123456712345671234567\252[REDACTED]\004\b"
    .text
    .globl main
    .type main, @function
main:
.LFBS2:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq %rbp, %rbp
    .cfi_def_cfa_register 6
    subq $16, %rsp
    movl %edi, -4(%rbp)
    movq %rsi, -16(%rbp)
    movl $.LC2, %edi
    [REDACTED]
    movl $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE2:
    .size main, .-main
```

E15

第三题 (15 分)

分析下面C语言程序和相应的x86-64汇编程序。其中缺失部分代码（被遮挡），请在对应的横线上填写缺失的内容。（注：每格1分）

```
#include <stdio.h>
#include "string.h"

void myprint(char *str)
{
    char buffer[16];
    [REDACTED] (buffer,str);          @strcpy _____
    printf("%s \n",buffer);
}

void alert(void)
{
    printf("[REDACTED] \n");          @Where am I?
}

int main(int argc,char *argv[])
{
    myprint("123456712345671234567\xaa\x84\x04\x08");
    return 0;
}
*****+.section .rodata
.LCO:
.string "[REDACTED]"          @%s \n _____
.text
.global myprint
.type myprint, @function
myprint:
.LFBO:
.cfi_startproc
pushq %rbp
.cfi_offset %rbp, -16
movq %rbp, %rbp
.cfi_def_cfa_register 6
subq $48, %rbp
movq %rdi, -40(%rbp)
movq %fs:40, [REDACTED]          @%rsp _____
movq %rax, -8(%rbp)
xorl %eax, %eax
movq [REDACTED], %rdx
leaq -32(%rbp), %rax
movq %rdx, [REDACTED]
call strcpy
[REDACTED] -32(%rbp), %rax
movq %rax, %rsi
movl $.LC0, %edi
movl $.S, %eax
call printf
```

```
nop [REDACTED], %rax
xord [REDACTED]
je [REDACTED] _stack_chk_fail
.L2:
Leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size myprint, .-myprint
.section .rodata
.LC1:
.string "Where am I?"
.text
.global alert
.type alert, @function
alert:
.LFB1:
.cfi_startproc
pushq %rbp
.cfi_offset %rbp, -16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC1, %edi
call puts
nop
popq %rbp
.cfi_def_cfa 7, 8
[REDACTED]          @ret _____
.cfi_endproc
.LFE1:
.size alert, .-alert
.section .rodata
.align 8
.LC2:
.string "123456712345671234567\x25\x004\b"          @204 _____
.text
.global main
.type main, @function
main:
.LFB2:
.cfi_startproc
pushq %rbp
.cfi_offset %rbp, -16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rbp
movl %edi, -4(%rbp)
movq %rsi, -16(%rbp)
movl $.LC2, %edi
[REDACTED]          @call myprint _____
movl $.S, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE2:
.size main, .-main
```

注：下面这种回答也对：

```
@movq %fs:40
@-8(%rbp), %rax
```

Processor Arch: SEQ/PIPE

E1

10. Y86 指令 `popl rA` 的 SEQ 实现如下图所示，其中①和②分别为：

Fetch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{ra:rb} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \textcircled{1}$
Decode	$\text{valA} \leftarrow R[\%esp]$ $\text{valB} \leftarrow R[\%esp]$
Execute	$\text{valE} \leftarrow \textcircled{2}$
Memory	$\text{valM} \leftarrow M_4[\text{valA}]$
Write Back	$R[\%esp] \leftarrow \text{valE}$ $R[\text{ra}] \leftarrow \text{valM}$
PC Update	$\text{PC} \leftarrow \text{valP}$

- A) $\text{PC} + 4 \quad \text{valA} + 4$
- B) $\text{PC} + 4 \quad \text{valA} + (-4)$
- C) $\text{PC} + 2 \quad \text{valB} + 4$
- D) $\text{PC} + 2 \quad \text{valB} + (-4)$

E1

答案: C

//知识点: popl 弹栈指令, 栈结构

- 1) popl 是双字节指令, 下一条指令位置 PC+2
- 2) Y86 是 32 位体系结构, popl 弹出 4 字节, 弹栈栈指针增加, valA + 4

E2

10. 在 Y86 的 SEQ 实现中，对仅考虑 IRMMOVQ, ICALL, IPOPQ, IRET 指令，对 mem_addr 的 HCL 描述正确的是：

```
word mem_addr = [
    icode in { (1), (2) } : valE;
    icode in { (3), (4) } : valA;
];
```

- A. (1) IRMMOVQ (2) IPOPQ (3) IRET (4) ICALL
- B. (1) IRMMOVQ (2) IRET (3) IPOPQ (4) ICALL
- C. (1) ICALL (2) IPOPQ (3) IRMMOVQ (4) IRET
- D. (1) IRMMOVQ (2) ICALL (3) IPOPQ (4) IRET

E2

10. 在 Y86 的 SEQ 实现中，对仅考虑 IRMMOVQ, ICALL, IPOPQ, IRET 指令，对 mem_addr 的 HCL 描述正确的是：

```
word mem_addr = [
    icode in { (1), (2) } : valE;
    icode in { (3), (4) } : valA;
];
```

- A. (1) IRMMOVQ (2) IPOPQ (3) IRET (4) ICALL
- B. (1) IRMMOVQ (2) IRET (3) IPOPQ (4) ICALL
- C. (1) ICALL (2) IPOPQ (3) IRMMOVQ (4) IRET
- D. (1) IRMMOVQ (2) ICALL (3) IPOPQ (4) IRET

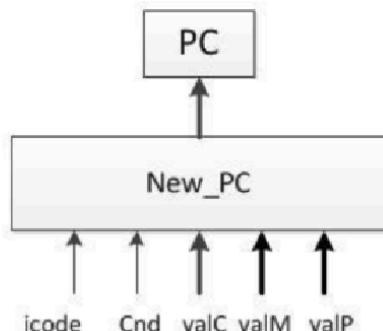
答案： D

E3

14. 在 Y86 的 SEQ 实现中，PC（Program Counter，程序计数器）更新的逻辑

结构如下图所示，请根据 HCL 描述为①②③④选择正确的数据来源。

```
Int new_pc = [
    # Call.
    Icode = ICALL : ①;
    # Taken branch.
    Icode = IJXX && Cnd : ②;
    # Completion of RET instruction.
    Icode = IRET : ③;
    # Default.
    1 : ④;
```



其中：Icode 为指令类型，Cnd 为条件是否成立，valC 表示指令中的常数值，
valM 表示来自返回栈的数据，valP 表示 PC 自增。

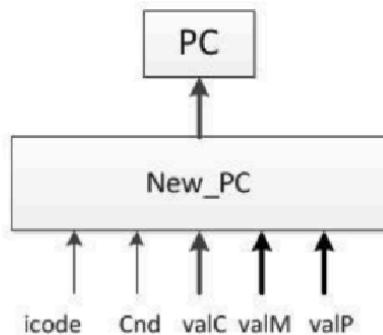
- A. valC, valM, valP, valP
- B. valC, valC, valP, valP
- C. valC, valC, valM, valP
- D. valM, valC, valC, valP

E3

14. 在 Y86 的 SEQ 实现中，PC（Program Counter，程序计数器）更新的逻辑

结构如下图所示，请根据 HCL 描述为①②③④选择正确的数据来源。

```
Int new_pc = [
    # Call.
    Icode = ICALL : ①;
    # Taken branch.
    Icode = IJXX && Cnd : ②;
    # Completion of RET instruction.
    Icode = IRET : ③;
    # Default.
    1 : ④;
```



其中：Icode 为指令类型，Cnd 为条件是否成立，valC 表示指令中的常数值，
valM 表示来自返回栈的数据，valP 表示 PC 自增。

- A. valC, valM, valP, valP
- B. valC, valC, valP, valP
- C. valC, valC, valM, valP
- D. valM, valC, valC, valP

答：C

E4

11. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间
- B. 增加流水线级数，不一定能获得总体性能的提升
- C. 指令间数据相关引发的数据冒险，不一定可以通过暂停流水线来解决。
- D. 流水级划分应尽量均衡，吞吐率会受到最慢的流水级影响，均衡的流水线能提高吞吐量。

E4

11. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间
- B. 增加流水线级数，不一定能获得总体性能的提升
- C. 指令间数据相关引发的数据冒险，不一定可以通过暂停流水线来解决。
- D. 流水级划分应尽量均衡，吞吐率会受到最慢的流水级影响，均衡的流水线能提高吞吐量。

答案：C

E5

13. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间。
- B. 减少流水线的级数，能够减少数据冒险发生的几率。
- C. 指令间数据相关引发的数据冒险，都可以通过 data forwarding 来解决。
- D. 现代处理器支持一个时钟内取指、执行多条指令，会增加控制冒险的开销。

E5

13. 关于流水线技术的描述，错误的是：

- A. 流水线技术能够提高执行指令的吞吐率，但也同时增加单条指令的执行时间。
- B. 减少流水线的级数，能够减少数据冒险发生的几率。
- C. 指令间数据相关引发的数据冒险，都可以通过 data forwarding 来解决。
- D. 现代处理器支持一个时钟内取指、执行多条指令，会增加控制冒险的开销。

答：C，load-use 冒险 不能通过 data forwarding 来解决

E6

12、关于流水线技术的描述，正确的是：

- A. 指令间数据相关引发的数据冒险，一定可以通过暂停流水线来解决。
- B. 流水线技术不仅能够提高执行指令的吞吐率，还能减少单条指令的执行时间。
- C. 增加流水线的级数，一定能获得性能上的提升。
- D. 流水级划分应尽量均衡，不均衡的流水线会增加控制冒险。

答：()

E6

12、关于流水线技术的描述，正确的是：

- A. 指令间数据相关引发的数据冒险，一定可以通过暂停流水线来解决。
- B. 流水线技术不仅能够提高执行指令的吞吐率，还能减少单条指令的执行时间。
- C. 增加流水线的级数，一定能获得性能上的提升。
- D. 流水级划分应尽量均衡，不均衡的流水线会增加控制冒险。

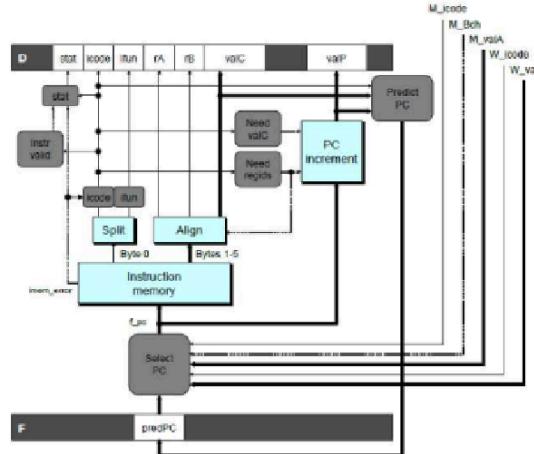
答：()

答案：A

说明：B 会增加单条指令的执行时间，C 可能会降低性能，D 和控制冒险没有关系

E7

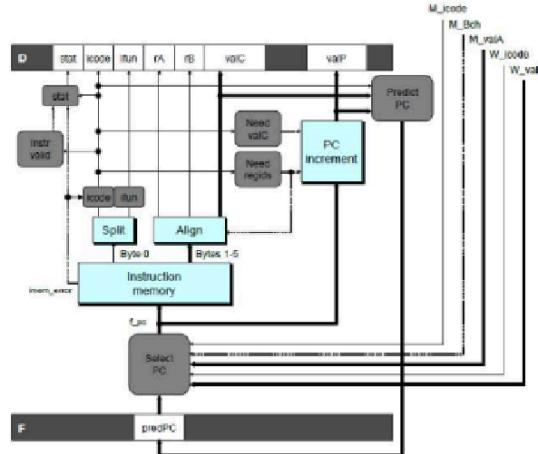
11. 流水线数据通路中的转移预测策略为总是预测跳转。如果转移预测错误，需要恢复流水线，并从正确的目标地址开始取值。其中，用来判断转移预测是否正确的信号是_①_和_②_，用来获得正确的目标地址的信号是_③_。



- A. ① M_icode ② M_Bch ③ M_valA
- B. ① W_icode ② M_Bch ③ M_valA
- C. ① W_icode ② M_Bch ③ W_valM
- D. ① M_icode ② M_Bch ③ W_valM

E7

11. 流水线数据通路中的转移预测策略为总是预测跳转。如果转移预测错误，需要恢复流水线，并从正确的目标地址开始取值。其中，用来判断转移预测是否正确的信号是_①_和_②_，用来获得正确的目标地址的信号是_③_。



M_icode
M_Bch
M_valA
W_0040
W_valM

- A. ① M_icode ② M_Bch ③ M_valA
- B. ① W_icode ② M_Bch ③ M_valA
- C. ① W_icode ② M_Bch ③ W_valM
- D. ① M_icode ② M_Bch ③ W_valM

E8

第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
$(x \geq 0) \ \ (3*x < 0)$	Y N
$(x \geq 0) \ \ (x < ux)$	Y N
$((x \gg 1) \ll 1) \leq x$	Y N
$((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2$	Y N
$(x - y > 0) == ((y+x+1)>>31 == 1)$	Y N
$dx + dy == (double)(y+x)$	Y N
$dx + dy + dz == dz + dy + dx$	Y N
$(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))$	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位 (s): 1-bit; 阶码字段 (exp): 6-bit; 小数字段 (frac): 5-bit。

1) 若将该格式下能表示的所有正规格数从小到大依次排列，则

相邻两数之间差值的最小值为 _____，最大值为 _____

2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

E8

第二题 (20 分)

1. 在 64 位机器上，判断下列等式是否恒成立

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
double dy = (double)y;
double dz = (double)z;
```

Expression	Always True?
(x >= 0) (3*x < 0)	Y N
(x >= 0) (x < ux)	Y N
((x >> 1) << 1) <= x	Y N
((x-y)<3) + (x>>1) - y == 8*x - 9*y + x/2	Y N
(x - y > 0) == ((y+x+1)>>31 == 1)	Y N
dx + dy == (double)(y+x)	Y N
dx + dy + dz == dz + dy + dx	Y N
(int)((lx+ly)>>1) == ((x&y) + ((x^y)>>1))	Y N

2. 假设 C 语言中新定义了一种数据类型 T，该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：

符号位 (s): 1-bit; 阶码字段 (exp): 6-bit; 小数字段 (frac): 5-bit。

1) 若将该格式下能表示的所有正规格数从小到大依次排列，则

相邻两数之间差值的最小值为 _____，最大值为 _____

2) 现定义了如下变量：

```
T a = -15.875;
T b = (1<<28) + (1<<24) + (1<<22);
T c = a*b;
```

请写出各变量的二进制表示：

变量	二进制表示
a	
b	
c	

答案： A, B, B, B

答案: A,A,A,B

答案: A,A,B,C

E9

12. 若处理器实现了三级流水线，每一级流水线实际需要的运行时间为 1ns、2ns 和 3ns，则此处理器不停顿地执行完毕 10 条指令需要的时间为：
A. 21 ns B. 12 ns C. 24 ns D. 36 ns

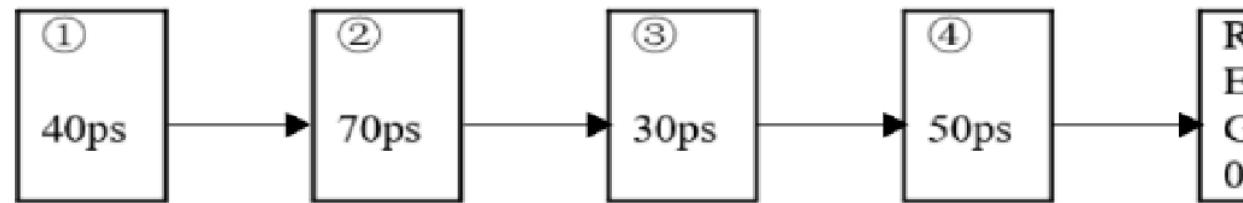
E9

12. 若处理器实现了三级流水线，每一级流水线实际需要的运行时间为 1ns、2ns 和 3ns，则此处理器不停顿地执行完毕 10 条指令需要的时间为：
A. 21 ns B. 12 ns C. 24 ns D. 36 ns

答案 D $3+3+10*3=36\text{ns}$

E10

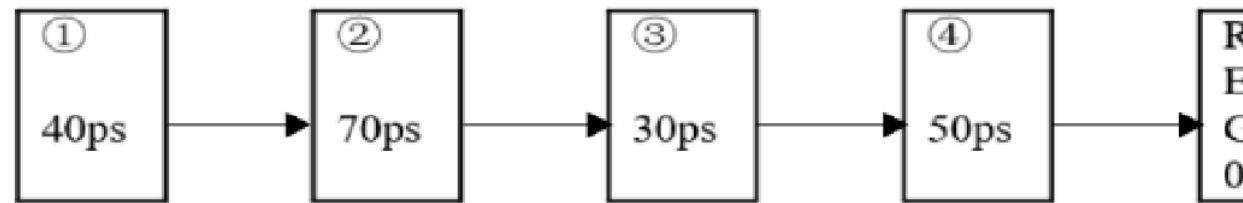
11. 如下图所示，①~④为四个组合逻辑单元，对应的延迟已在图上标出，REG0 为一寄存器，延迟为 20ps。通过插入额外的 2 个流水线寄存器 REG1、REG2（延迟均为 20ps），可以对其进行流水化改造。改造后的流水线的吞吐率最大为 _____ GIPS。



- A. 7.69 B. 8.33 C. 10.00 D. 11.11

E10

11. 如下图所示，①~④为四个组合逻辑单元，对应的延迟已在图上标出，REG0 为一寄存器，延迟为 20ps。通过插入额外的 2 个流水线寄存器 REG1、REG2（延迟均为 20ps），可以对其进行流水化改造。改造后的流水线的吞吐率最大为 _____ GIPS。



- A. 7.69 B. 8.33 C. 10.00 D. 11.11

【答】C。 额外的 2 个寄存器应当插入①②之间、②③之间，最慢的一级延迟为 $30+50+20=100\text{ps}$ ，吞吐率为 $1000/100=10\text{GIPS}$

E11

12. 一个功能模块包含组合逻辑和寄存器，组合逻辑单元的总延迟是 100ps，单个寄存器的延时是 20ps，该功能模块执行一次并保存执行结果，理论上能达到的最短延时和最大吞吐分别是多少？
- A. 20ns, 50GIPS
 - B. 120ns, 50GIPS
 - C. 120ns, 10GIPS
 - D. 20ps, 10GIPS

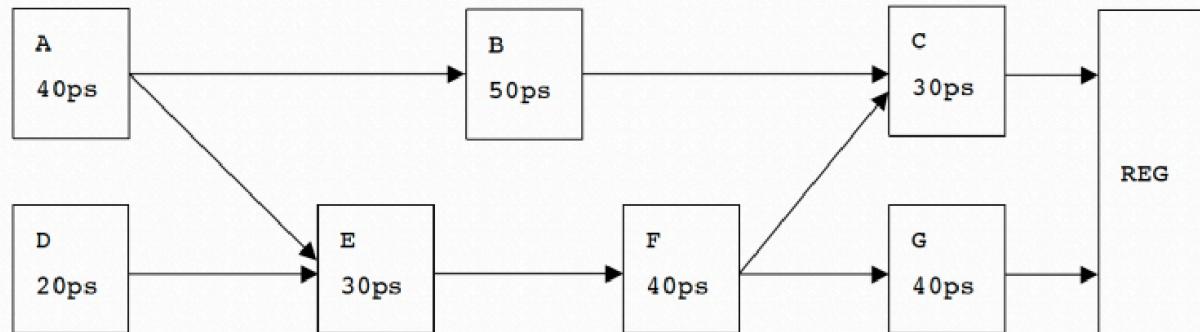
E11

12. 一个功能模块包含组合逻辑和寄存器，组合逻辑单元的总延迟是 100ps，单个寄存器的延时是 20ps，该功能模块执行一次并保存执行结果，理论上能达到的最短延时和最大吞吐分别是多少？
- A. 20ns, 50GIPS
 - B. 120ns, 50GIPS
 - C. 120ns, 10GIPS
 - D. 20ps, 10GIPS

答案：B，主要考察延时和吞吐的区别，延时最小是 logic + 1 个 register，吞吐最高时是把 logic 划分成无穷多份，每一级需要 20ps， $1000/20ps = 50GIPS$

E12

8. A-G 为 7 个基本逻辑单元，下图中标出了每个单元的延迟，以及用箭头标出了单元之间所有的依赖关系。寄存器的延迟均为 20ps，在图中以 REG 符号表示。假设流水线寄存器只能添加在有直接依赖关系的基本逻辑单元之间，而不能在 C 或 G 与 REG 之间。以下说法正确的是：



- A. 原电路的吞吐量 (throughput) 舍入后大约是 $1000/150=6.667$ GIPS。
- B. 将该电路改造成 2 级流水线有 8 种方法
- C. 如果将该电路改造成 3 级流水线，延迟最小可以到 80 ps。
- D. 不论实现该电路时遇到怎样的数据冒险和控制冒险，一定可以对流水线寄存器使用暂停 (stalling) 解决。

E12

- A. 错。 $1000/170 = 5.882$ 而非 $1000/150 = 6.667$ 。不要漏掉寄存器。
- B. 错。考虑 D-E-F-G 这条路，要么插 D-E，要么 E-F，要么 F-G。如果是 D-E，那么为了使每条极大路径上都恰有一个新插入的寄存器，考虑 D-E-F-C，那么 F-C 之间也不能插入了。但是 A-E-F-G 上又必须有一个，所以只能插 A-E。这样之后不管是插在 A-B 还是 B-C 所得方案都是合法的。对称地，如果是 F-G，结果也是如此。最后考虑 E-F 的情况，此时 A-E，D-E，F-C，F-G 之间均不能再插入，但 A-B 和 B-C 任选一个插入得到的方案都合法。因此一共有 $2+2+2=6$ 种。
- C. 错。3 阶段里的最优为 90ps。
- D. 对。最朴素的情况每条指令 stall 足够多次，电路回到 SEQ 的状态。

E13

第四题 (15 分)

请分析 Y86-64 ISA 中加入的一族间接跳转指令: `jxx *rB`, 其格式如下:

C	Fn	F	rB
---	----	---	----

该指令的功能是跳转到寄存器 `R[rB]` 所存放的地址。类似于直接跳转指令，间接跳转指令也包括无条件跳转和条件跳转，通过不同的功能码 Fn 来指示。为了和直接跳转区别，icode 为 IJREGXX。时钟周期适当进行延长，在不修改原有的硬件线路和信号设置的前提下，只增加和新指令有关的逻辑，回答以下问题。

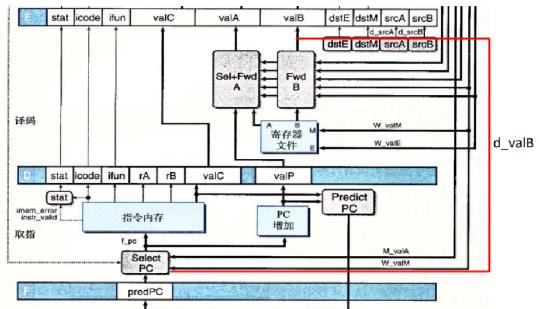
1. 在教材中的 SEQ 处理器上实现该指令，请补全下表中每个阶段的操作。需要说明的信号可能有: icode, ifun, rA, rB, valA, valB, valC, valE, valP, Cnd, R[], M[], PC, CC

Stage	<code>jxx *rB</code>
Fetch	<code>icode : ifun ← M1[PC]</code>
Decode	
Execute	
Memory	
Write Back	
Update PC	<code>PC ← Cnd ? valE : valP</code>

在 foo 函数运行过程中，计算以下情况 foo 函数的执行周期数。（周期数计算从执行 foo 第一条指令开始，直到其返回指令 ret 完全通过流水线为止。另外假设 foo 函数开始的若干条指令不会和 foo 函数体外的指令形成冒险。）

`n=-1: _____; n=0: _____; n=2: _____`

2. 考虑在教材中的 Pipeline 处理器上实现该指令，采用总是选择分支（预测下一条指令时使用跳转地址 `R[rB]`）的预测策略。



由于 `R[rB]` 需要到译码阶段才能得到，需要增加一条从 `Fwd B` 输出信号 `d_valB` 到 `Select PC` 的旁路通路，增加线路如图所示。增加旁路后，为了预测 `jxx *rB` 的下一条 PC，_____（需要/不需要）在该指令和下一条指令间插入气泡。

`Select PC` 的 HCL 代码如下图所示：

```
word f_pc = [
    ①
    (M_icode == IJXX || M_icode == IJREGXX) && !M_cnd : M_valA;
    ②
    W_icode == IRET : W_valM;
    ③
    1 : F_predPC;
    ④
]
```

为了预测下一条 PC，需要修改 `Select PC` 的 HCL 代码，增加一行 _____，增加的位置可以是 _____（写出所有可能的位置，错填不得分，漏填可得部分分）

3. 请将该指令预测错误的触发条件，以及此时流水线的控制逻辑补充完整。

触发条件：（如果有多种可能请任写一种）

`== IJREGXX &&`

控制逻辑：（如果有多种可能请任写一种）

F	D	E	M	W

4. 基于改造后的 Y86-64 PIPE 考虑如下代码片段，回答问题。

```
# Array of 3 elements
array:
.quad return
.quad L1
.quad L2

# void foo(long n, long *arr)
# n in %rdi, arr in %rsi
foo:
    rmovq %rdi, %rdx          # line 1
    addq %rdx, %rdx            # line 2
    addq %rdx, %rdx            # line 3
    addq %rdx, %rdx            # line 4
    irmovq array, %rcx         # line 5
    addq %rcx, %rcx            # line 6
    andq %rdi, %rdi            # line 7
    jge *%rcx                  # line 8
    return:                   #
    ret                         # line 9
L2: #
    rmovq 16(%rsi), %rcx      # line 10
    rmovq %rcx, 8(%rsi)        # line 11
L1: #
    rmovq 8(%rsi), %rcx        # line 12
    rmovq %rcx, (%rsi)         # line 13
    jmp return                 # line 14
```

E13

1.

Stage	jxx *rB
Fetch	icode : ifun $\leftarrow M_1[PC]$ xA : rB $\leftarrow M_1[PC+1]$ valP $\leftarrow PC+2$
Decode	valB $\leftarrow R[rB]$
Execute	valE $\leftarrow valB + 0$ Cnd = Cond(CC, ifun)
Memory	/
Write Back	/
Update PC	PC $\leftarrow Cnd ? valE : valP$

2. 不需要。间接跳转在 decode 阶段时 SelectPC 正好利用最新转发的 valB 的值作为预测地址访问指令内存。

1 2 3 处都可以插入。插入的内容见下。④是无效的插入位置。注意，原来的 M_XXX 条件和 W_XXX 条件互斥，所以其顺序任意，但它们都不会和新加入的条件冲突。例如 mispredict 发生时，Decode 阶段的指令已经被清空，所以最终不会导致多个条件成立。

```
word f_pc = [
    // D_icode == IREGXX: d_valB;
    (M_icode == IJXX || M_icode == IREGXX) && !M_cnd : M_valA;
    // D_icode == IREGXX: d_valB;
    W_icode == IRET : W_valM;
    D_icode == IREGXX: d_valB;
    1 : F_predPC;
    ④
]
```

3. E_icode == IJREGXX && !e_Cnd

F	D	E	M	W
normal/stall	bubble	bubble	normal	normal

分析方法同 IJXX。注意，触发条件是在 E 阶段，因为 E 阶段结束、M 阶段开始时就要控制流水线寄存器。

4. 15 13 20

n == -1: line 8 分支预测错误，惩罚 2 个周期。一共 9 + 2 + 4 (trailing cycles for ret) = 15。
n == 0: 一共 9 + 4 (trailing cycles for ret) = 13。
n == 1: line 12 和 13 发生 load/use hazard (不考虑加载转发)，惩罚 1 个周期。一共 12 + 1 + 4 (trailing cycles for ret) = 17。
n == 2: line 12 和 line 13, 以及 line 10 和 line 11 各发生一次 load/use hazard, 共惩罚 2 个周期。一共 14 + 2 + 4 (trailing cycles for ret) = 20。

E14

【综合】

20. 以下提供了一段代码的 C 语言、汇编语言以及运行到某一时刻栈的情况

汇编:

```
000000000400596 <func>:
400596: sub    $0x28,%rsp
400598: mov    %fs:0x28,%rax
4005a3: mov    %rax,0x18(%rsp)
4005a8: xor    %eax,%eax
4005aa: mov    (%rdi),%rax
4005ad: mov    0x8(%rdi),%rdx
4005b1: cmp    %rdx,%rax
4005b4: jge    .(1)
4005b6: mov    %rdx,(%rdi)
4005b9: mov    %rax,0x8(%rdi)
4005bd: mov    0x8(%rdi),%rax
4005c1: test   %rax,%rax
4005c4: jne    4005cb <func+0x35>
4005c6: mov    (%rdi),%rax
4005c9: jmp    .(2)
4005cb: mov    (%rdi),%rdx
4005ce: sub    %rax,%rdx
4005d1: mov    %rdx,(%rsp)
4005d5: mov    %rax,0x8(%rsp)
4005da: mov    .(3),%rdi
4005dd: callq  400596 <func>
4005e2: mov    0x18(%rsp),%rcx
4005e7: xor    .(4),%rcx
4005f0: .(5) 4005f7 <func+0x61>
4005f2: callq  400460 <_stack_chk_fail@plt>
4005f7: add    .(6),%rsp
4005fb: retq

0000000004005fc <main>:
4005fc: sub    $0x28,%rsp
400600: mov    %fs:0x28,%rax
400609: mov    %rax,0x18(%rsp)
40060e: xor    %eax,%eax
400610: movq   0x69,%rsp
400618: movq   0xfc,0x8(%rsp)
400621: mov    %rsp,%rdi
400624: callq  400596 <func>
400629: mov    %rax,%rsi
```

```
40062c: mov    $0x4006e4,%edi
400631: mov    $0x0,%eax
400636: callq  400470 <printf@plt>
40063b: mov    0x18(%rsp),%rdx
400640: xor    .(4),%rdx
400649: .(5) 400650 <main+0x54>
40064b: callq  400460 <_stack_chk_fail@plt>
400650: mov    $0x0,%eax
400655: add    .(6),%rsp
400659: retq
```

C 语言与堆栈:

```
typedef struct{
    long a;
    long b;
} pair_type;
long func(pair_type *p) {
    if (p -> p < p -> b) {
        long temp = p -> a;
        p -> a = p -> b;
        p -> b = temp;
    }
    if ((7) ) {
        return p -> a;
    }
    pair_type np;
    np.a = .(8);
    np.b = .(9);
    return func(&np);
}
int main(int argc, char* argv[]) {
    pair_type np;
    np.a = .(10);
    np.b = .(11);
    printf("%ld", func(&np));
    return 0;
}
```

一些可能用到的字符的 ASCII 码表:

换行	空格	"	%	()	,	0	A	a
0x0a	0x20	0x22	0x25	0x28	0x29	0x2c	0x30	0x41	0x61

回答问题:

I. gdb 下使用命令 x/4b 0x4006e4 后 (即查看 0x4006e4 开始的 4 个字节, 用 16 进制表示) 得到的输出结果是

0x4006e4: 0x_____ 0x_____ 0x_____ 0x_____

II. 互相翻译 C 语言代码和汇编代码, 补充缺失的空格 (标号相同的为同一格)。

- | | |
|------|--------------|
| (1) | <func+_____> |
| (2) | <func+_____> |
| (3) | _____ |
| (4) | _____ |
| (5) | _____ |
| (6) | _____ |
| (7) | _____ |
| (8) | _____ |
| (9) | _____ |
| (10) | _____ |
| (11) | _____ |

III. 补充栈的内容。使用 16 进制, 可以不写前导多余的 0; 对于给定已知条件后仍无法确定的值, 填写“不确定”; 已知程序运行过程中寄存器%rs 的值没有改变。

- | | |
|-----|-------|
| (a) | _____ |
| (b) | _____ |
| (c) | _____ |
| (d) | _____ |
| (e) | _____ |
| (f) | _____ |
| (g) | _____ |
| (h) | _____ |
| (i) | _____ |
| (j) | _____ |
| (k) | _____ |

IV. 程序运行结果为_____。

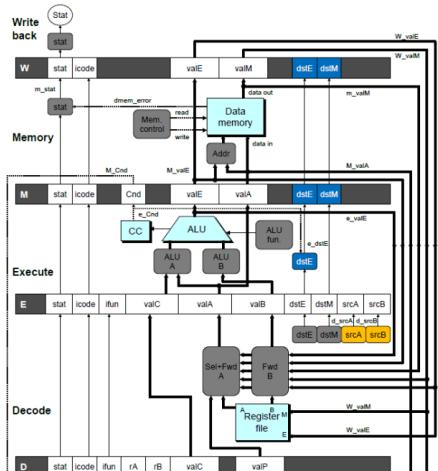
.....
0x0000000000000000
0xc76d5add7bbeaa00
0x00007fffffdff60
(a) _____
(b) _____
(c) _____
(d) _____
(e) _____
(f) _____
(g) _____
(h) _____
(i) _____
(j) _____
(k) _____
(l) [栈顶] {低地址}

E14

得分

第四题（15分）

这是一款 Y86-32 流水线处理器的结构图（局部），请以此为基础，依次回答下列问题。



1、该处理器设计采用了前递 (forwarding) 技术, 一定程度上解决了数据相关的问题, 在上图中体现在 Sel+FwdA 和 FwdB 部件上。前者输出的信号会存到流水线寄存器 F 的 va1A 域 (即 F.va1A 信号), 请补充该信号的 HCL 语言描述。

```

int E_valA = [
    D_icode in { ICALL, IJXX } : _____; # ① 答案: D_valP
    d_srcA = e_dstE : _____; # ② 答案: e_valE
    d_srcA = M dstM : _____; # ③ 答案: m_valM
]

```

```
d_srcA == M_dstE : M_valE      ;  
d_srcA == W_dstM : W_valM      ;  
...  
;
```

2、如果在该处理器上运行下面的程序，每条指令在不同时钟周期所处的流水线阶段如下表所示。在这种情况下，哪条指令的执行结果会有错误？写出该指令的地址：

0x01e。 (1分)

3、如需检测出这个情况，需要增加逻辑电路，用 HCL 语言表达如下：

E_icode in {IMRMOVL, IPOPL} && _____ in { _____ }

答案: E_icode in {IMRMOVL, IPOPL} && E_dstM in { d_srcA, d_srcB }, 2分,
全对才得分

4、当新增的电路检测出这个情况后，应对各流水线寄存器进行不同的设置，以便在尽可能少影响性能的前提下解决该问题。请填写下表，可选的设置包括normal/bubble/stall三种。

F	D	E	M	W

答案: stall stall bubble normal normal- 3分, 全对才得分

5、如果遇到下面程序代码所示的情况，该处理器运行时仍然存在问题。因此，还需要新增检测电路。当新增的电路检测出这个情况后，应对各流水线寄存器进行不同的设置，以便在尽可能减少影响性能的前提下解决该问题。请填写下表，可选的设置包括 normal/hubble/stall 三种。

```
demo2.ys  
...  
0x018: rmmovl %ecx, 0(%edx)  
0x01e: irmovl $10, %ebx  
0x024: popl %esp  
0x026: ret
```

F	D	E	M	W

第六章 一元一次方程 1.1.1 从实际问题到方程 1 分一个对由得分

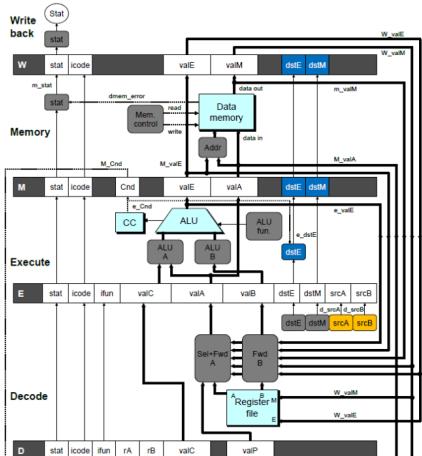
E15

得分

【2018年期中】

第四題（15分）

这是一款 Y86-32 流水线处理器的结构图（局部），请以此为基础，依次回答下列问题。



1、该处理器设计采用了前递 (forwarding) 技术，一定程度上解决了数据相关的问题，在上图中体现在 Sel+FwdA 和 FwdB 部件上。前者输出的信号会存到流水线寄存器 E 的 valA 域（即 E_valA 信号），请补全该信号的 HCL 语言描述。

```
int E_valA = [  
    D_icode in { ICALL, IJXX } : _____ ; # ①  
    d_srcA == e_dstE : _____ ; # ②  
    d_srcA == M_dstM ; _____ ; # ③
```

```
d_srcA == M_dstE : M_valE ;  
d_srcA == W_dstM : W_valM ;  
...
```

3

2、如果在该处理器上运行下面的程序，每条指令在不同时钟周期所处的流水线阶段如下表所示。在这种情况下，哪条指令的执行结果会有错误？写出该指令的地址：_____。

1	2	3	4	5	6	7	8	9	10	11	12
F	D	E	M	W							
	F	D	E	M	W						
		F	D	E	M	W					
			F	D	E	M	W				
				F	D	E	M	W			
					F	D	E	M	W		
						F	D	E	M	W	

3、如需检测出这个情况，需要增加逻辑电路，用 HCL 语言表达如下：
Eicode in {TMRMOVL, TPOPL} && _____ in { }

4、当新增的电路检测出这个情况后，应对各流水线寄存器进行不同的设置，以便在尽可能少影响性能的前提下解决该问题。请填写下表，可选的设置包括normal/hubble/stall三种。

F	D	E	M	W

5、如果遇到下面程序代码所展示的情况，该处理器运行时仍然存在问题。因此，还需要新增检测电路。当新增的电气检测出这个情况后，应对各流水线寄存器进行不同的设置，以便在尽可能少影响性能的前提下解决该问题。请填写下表，可选的设置包括 normal / bubble / stall 三种。
dom0_vs

demo2. ys

```
0x018: rmmovl %ecx, 0(%edx)
0x01e: irmovl $10, %ebx
0x024: popl %esp
0x026: ret
```

F	D	E	M	W

4.57 在我们的 PIPE 设计中，只要一条指令执行了 load 操作，从内存中读一个值到寄存器，并且下一条指令要使用这个寄存器作为源操作数，就会产生一个暂停。如果要在执行阶段中使用这个源操作数，暂停是避免冒险的唯一方法。对于第二条指令将源操作数存储到内存的情况，例如 `imovq` 或 `push` 指令，是不需要这样的暂停的。考虑下面这段代码示例：

```
1    nrmovq 0(%rcx),%rdx    # Load 1
2    pushq %rdx                # Store 1
3    nop
4    popq %rdx                # Load 2
5    rmovq %rax,0(%rdx)      # Store 2
```

在第1行和第2行，`mmxmovq`指令从内存读一个值到`rdx`，然后`pushq`指令将这个值压入栈中。我们的PIPE设计会让`pushq`指令暂停，以避免装载/使用冒险。不过，可以看到，`pushq`指令要到访存阶段才会需要`rdx`的值。我们可以再添加一条旁路旁路，如图4-70所示，将内存输出（信号`v_valm`）转发到流水线寄存器M中的`vals`字段。在下一个时钟周期，被传送的值就能写入内存了。这种技术称为真转发（load forwarding）。

注意，上述代码序列中的第二个例子(第4行和第5行)不能利用加载转发。`popq`指令加载的值是作为一条指令地址计算的一部分的，而在执行阶段而非访存阶段就需要这个值了。

A. 写出描述发现加载/使用冒险条件的逻辑公式，类似于图4-64所示。除了能用加载转发时不会

B. 文件 pipe1.hcl 包含一个 PIPE 控制逻辑的修改版。它含有信号 e_valu 的定义，用来实现图 4-70 中标号为 “Fwd_A”的块。它将流水线控制逻辑中的加载/使用冒险的条件设置为 0，因此流水线控制逻辑将不会发现任何形式的加载/使用冒险。修改这个 HCL 描述以实现加载转发。请参考实验资料获得如何为你的解答生成模拟器以及如何测试模拟器的指导。

E15

回答问题：

1. gdb 下使用命令 `x/4b 0x4006e4` 后（即查看 `0x4006e4` 开始的 4 个字节，用 16 进制表示）得到的输出结果是？

`0x4006e4: 0x25 0x6c 0x64 0x00`

2. 互相翻译 C 语言代码和汇编代码，补充缺失的空格（标号相同的为同一格）。

- (1) `4005bd <func+0x27>`
- (2) `4005e2 <func+0x4c>`
- (3) `%rsp`
- (4) `%fs:0x28`
- (5) `je`
- (6) `$0x28`
- (7) `p -> b == 0`
- (8) `p -> a - p -> b`
- (9) `p -> b`
- (10) `105`
- (11) `252`

3. 补充栈的内容。使用 16 进制，可以不写前导多余的 0；对于给定已知条件后仍无法确定的值，填写“不确定”；已知程序运行过程中寄存器`%fs` 的值没有改变。

- (a) `0x0000000000000009`
- (b) `0x00000000000000fc`
- (c) `不确定`
- (d) `0xc76d5add7bbeaa00`
- (e) `0x00000000004005e2`
- (f) `0xc76d5add7bbeaa00`
- (g) `0x000000000000002a`
- (h) `0x0000000000000069`
- (i) `0x00000000004005e2`
- (j) `0xc76d5add7bbeaa00`
- (k) `不确定`

4. 程序输出结果是？

21

【注意】

- 1 最后一空为 `0x00`，因为要用`\0`作字符串结尾
- 2 (1) (2) 较难
- 2 (10) (11) 不能颠倒，因为汇编如此写
- 3 (a) (b) 两空不能颠倒，因为 func 函数会修改内存的值
程序的作用是用（类似于）更相减损术求最大公约数

E16

得分

【2022年期中】

第四题(15分) 请分析 Y86-64 ISA 中新加入的一组条件 POP 指令: cpopqXX。 cpopqXX 和 popq 指令的机器码格式如下。

	B	fun	rA	F
popq	B	0	rA	F

类似cmovXX，该组cpopqXX指令在条件码(Cnd)满足时，会将栈顶的8字节数据读入到寄存器rA中，并且使栈指针增加8；如果条件不满足，则不执行该指令，即对rA、栈指针都无影响。

1. 在教材所描述的SEQ处理器上实现该指令，请按下表补全每个阶段的操作。需说明的信号可能会包括: icode, ifun, rA, rB, valA, valB, valC, valE, valP, Cnd: the register file R[], data memory M[], Program counter PC, condition codes CC。其中对存储器的引用必须标明字节数。如果在某一阶段没有任何操作，请填写none指明。

Stage	cpopqXX rA
Fetch	<code>icode:ifun ← M₁[PC]</code> <code>rA:rB ← M₁[PC+1]</code> <code>valP ← PC+2</code>
Decode	_____
Execute	_____
Memory	_____
Write back	_____
PC update	<code>PC ← valP</code>

2. 考虑在教材中的 Pipeline 处理器上实现该指令，需要改进教材所描述的 PIPE 处理器在执行 (E:Execute) 阶段的处理逻辑，使其能够在条件不满足时不修改 rA 和栈指针的值。请依据改进后的处理器，补全执行阶段的部分 HCL 代码：

```
word e_dstE = [
    (E_icode in {IRRMOVQ, ① } && ② ) : RNONE;
    1 : E_dstE;
];

word e_dstM = [
    (E_icode == ① && ② ) : RNONE;
    1 : E_dstM;
]
```

① _____ ② _____

3. 和 popq 与 mrmovq 相同，在执行该指令的过程中有可能会引发加载-使用冒险 (load-use hazard)。请将在引入了 cpopqXX 指令后的 Y86-64 PIPE 处理器中，加载-使用冒险的触发条件以及处理冒险的控制逻辑补充完整。如有多种可能，任写一种即可：

触发条件: E_icode in {IMRMOVQ, IPOPQ} && e_dstM in {_____, _____}
控制逻辑: (选填 stall, bubble 和 normal)

F	D	E	M	W
			normal	normal

4. 基于改造后的 Y86-64 PIPE 考虑如下代码片段，回答问题。

```
# long foo(long n)
# n in %rdi
foo:
    xor    %rax, %rax          # Line 1
    pushq  %rsp               # Line 2
    irmovq $1, %rsi            # Line 3
L1:
    subq    %rsi, %rdi          # Line 4
    andq    %rdi, %rdi          # Line 5
    cpopqe %rsi                # Line 6
```

addq %rsi, %rax # Line 7
andq %rdi, %rdi # Line 8
jne L1 # Line 9
ret # Line 10

假设条件分支总是预测跳转，计算在输入的n值取1和3时，函数foo执行的周期数。(周期数计算从执行foo第一条指令开始，直到其返回指令ret完全通过流水线为止。另外在调用foo之前，处理器已经执行过足够多的nop指令。)
n为1时候需要执行 _____ 个周期；n为3时候需要执行 _____ 个周期。

E16

答案：

1.

Stage	cpopqXX rA
Fetch	<u>icode:ifun</u> $\leftarrow M_1[PC]$ <u>rA:rB</u> $\leftarrow M_1[PC+1]$ <u>valP</u> $\leftarrow PC+2$
Decode	<u>valB</u> $\leftarrow R[\%rsp]$ <u>valA</u> $\leftarrow R[\%rsp]$
Execute	<u>valE</u> $\leftarrow valB + 8$ <u>Cnd</u> $\leftarrow Cond(CC, ifun)$
Memory	<u>valM</u> $\leftarrow M_8[valA]$
Write back	<u>if (Cnd) R[\%rsp] $\leftarrow valE$</u> <u>if (Cnd) R[rA] $\leftarrow valM$</u>
PC update	<u>PC $\leftarrow valP$</u>

(每空1分，共7分)

2. ①IPOPQ ②!e_Cnd (每空1分，共2分)，第1空填ICPOPOQXX也算对

3.

触发条件：E_icode in {IMRMOVQ, IPOPQ} && e_dstM in {d_srcA, d_srcB}

控制逻辑：(选填 stall, bubble 和 normal)

F	D	E	M	W
Stall	Stall	Bubble	normal	normal

(共3分，其中 e_dstM in {d_srcA, d_srcB} 1分，全对才得分；控制逻

辑2分，全对才得分)

4.

解析：循环过程为Line 4~9，共6条

n=3 : 4(填充流水线)+3(Line 1~3)+6*3(共三次循环)+1(Line 7~8:load-use hazard)+2(Line 9:branch misprediction)+1(ret指令)=29

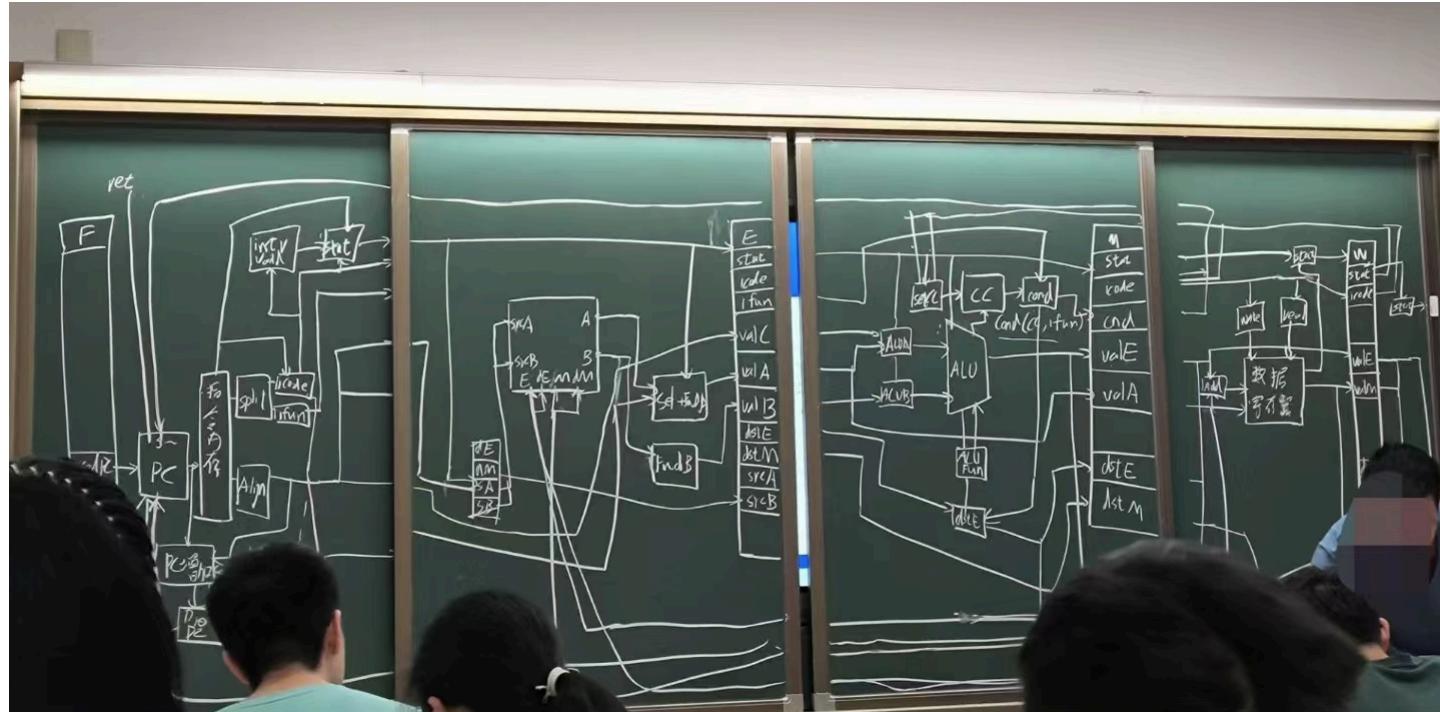
n=1 : 4(填充流水线)+3(Line 1~3)+6 (第一次循环)+1(Line 7~8:load-use hazard)+2(Line 9:branch misprediction)+1(ret指令)=17

(对任意一空得2分，两空全对得3分)

Notices

记忆

ICS 去年小班同学默写盛况



时间轴

周次	大班	节次	主题 (放假安排以学校通知为准)	小班	节次	小班对应大班	LAB日程(参考)	回课同学
一	9月9日	1	Overview	9月11日	1	大班1		
	9月12日	2	Bits and Bytes/Integers				L1 (datalab) out	
二	9月14日	3	Floating Point	9月18日	2	大班2/3		A/B
	9月19日	4	Machine Prog: Basics					
三	9月23日	5	Machine Prog: Control	9月25日	3	大班4/5	L2 (bomblab) out	C/D
	9月26日	6	Machine Prog: Procedures					
四	9月30日	7	Machine Prog: Data					
	10月3日		国庆节快乐!					
五	10月7日		国庆节快乐!	10月9日	4	大班6/7	L3(attacklab) out	E/F
	10月10日	8	Machine Prog: Advanced					
六	10月14日	9	Processor Arch: ISA&Logic	10月16日	5	大班8/9	L4 (archlab) out	G/A
	10月17日	10	Processor Arch: Sequential					
七	10月21日	11	Processor Arch: Pipelined	10月23日	6	大班10/11		B/C
	10月24日	12	The Memory Hierarchy				L5 (cachelab) out	
八	10月28日	13	Cache Memories	10月30日	7	大班12/13/期中复习		D/E/F
	10月31日	14	Program optimization					
九	11月4日	15	期中考试	11月6日	8	期中讲解		
	11月7日	16	专题讲座					
十	11月11日	17	Linking I	11月13日	9	大班14/17		G/A
	11月14日	18	Linking II					
十一	11月18日	19	ECF: Exceptions & Processes	11月20日	10	大班18/19	L6 (tshlab) out	B/C
	11月21日	20	ECF: Signals & Nonlocal Jumps					
十二	11月25日	21	System Level I/O	11月27日	11	大班20/21		D/E
	11月28日	22	Virtual Memory: Concepts				L7(malloclab) out	
十三	12月2日	23	Virtual Memory: Systems	12月4日	12	大班22/23		F/G
	12月5日	24	Dynamic Memory Allocation					
十四	12月9日	25	Network Programming I	12月11日	13	大班24/25	L8 (proxylab) out	A/B
	12月12日	26	Network Programming II					
十五	12月16日	27	Concurrent Programming	12月18日	14	大班26/27		C/D
	12月19日	28	Synchronization: Basic					
十六	12月23日	29	Synchronization: Advanced	12月25日	15	大班28/29/期末复习	期末核查	E/F/G
	12月26日	30	期末复习					

补充资料

- HCL语言: [HCL Descriptions of Y86-64 Processors.pdf](#)

CS:APP3e Web Aside ARCH:HCL:
HCL Descriptions of Y86-64 Processors*

Randal E. Bryant
David R. O'Hallaron

December 29, 2014

- 我的Y86-64学习笔记: [Y86-64 Note.html](#)

- Y86-64 Note
 - Y86-64指令集体系结构
 - Y86-64程序实现
 - Y86-64硬件结构
 - SEQ
 - SEQ+
 - PIPE-
 - PIPE

Y86-64 Note

Y86-64指令集体系结构

Y86-64指令

- YAS: Y86-64 assembler ($ys \rightarrow yo$)
- YIS: Y86-64 simulator (run yo programs)

- 我的回课: [Pipelined Review.pdf](#) \\ 往年补充资料: [pipelined res.pdf](#)

THANKS

Made by WalkerCH

changxinhai@stu.pku.edu.cn

Reference: [Weicheng Lin]'s presentation.

Reference: [Arthals]'s templates and content.

