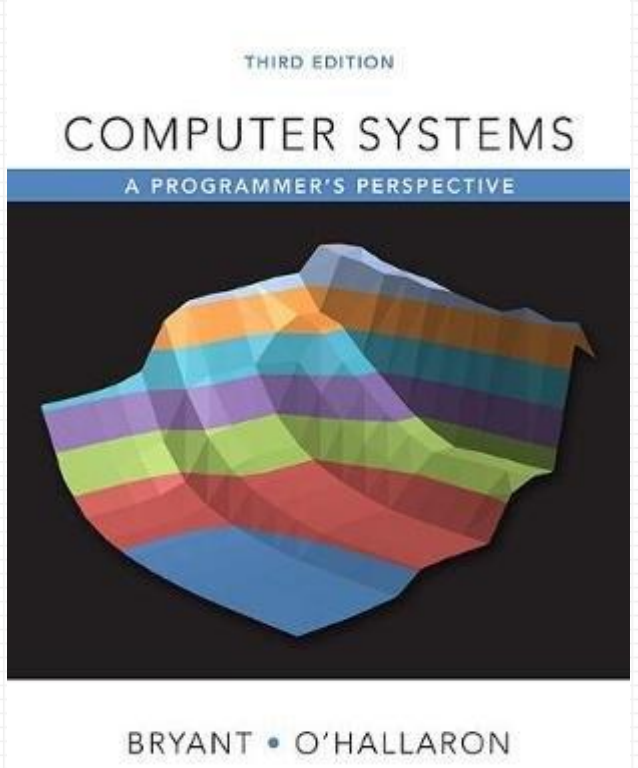
PIPELINED

计算机系统讨论班-回课

常欣海 元培学院



目录

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）

$$\mathrm{CPI} = \frac{\mathrm{C}_i + \mathrm{C}_b}{\mathrm{C}_i} = 1.0 + \frac{\mathrm{C}_b}{\mathrm{C}_i}$$

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

- lp - 加载处罚，mp - 预测错误分支处罚，rp - 返回处罚

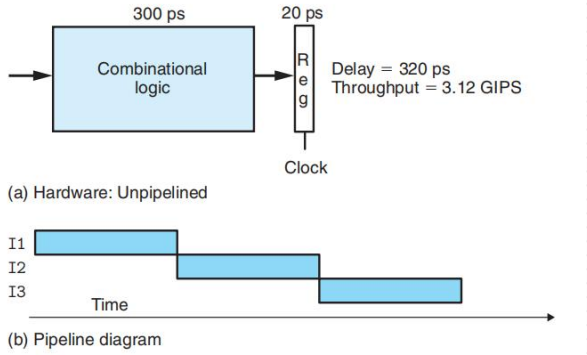
Cause	Name	Instruction frequency	Condition frequency	Bubbles	Product
Load/use	lp	0.25	0.20	1	0.05
Mispredict	mp	0.20	0.40	2	0.16
Return	rp	0.02	1.00	3	0.06
Total penalty					0.27

4. **IPC**：每周期执行指令平均数（单位：指令数）

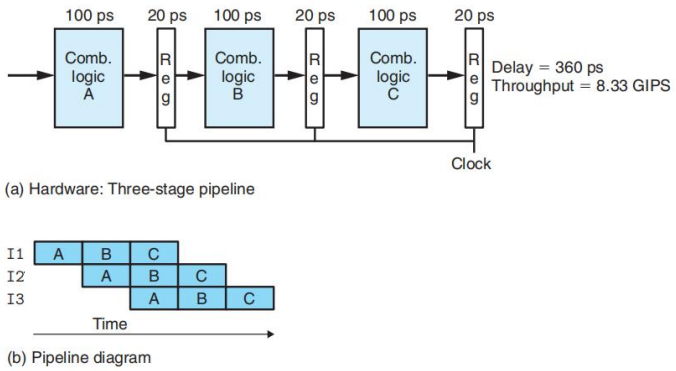
流水线原理基础

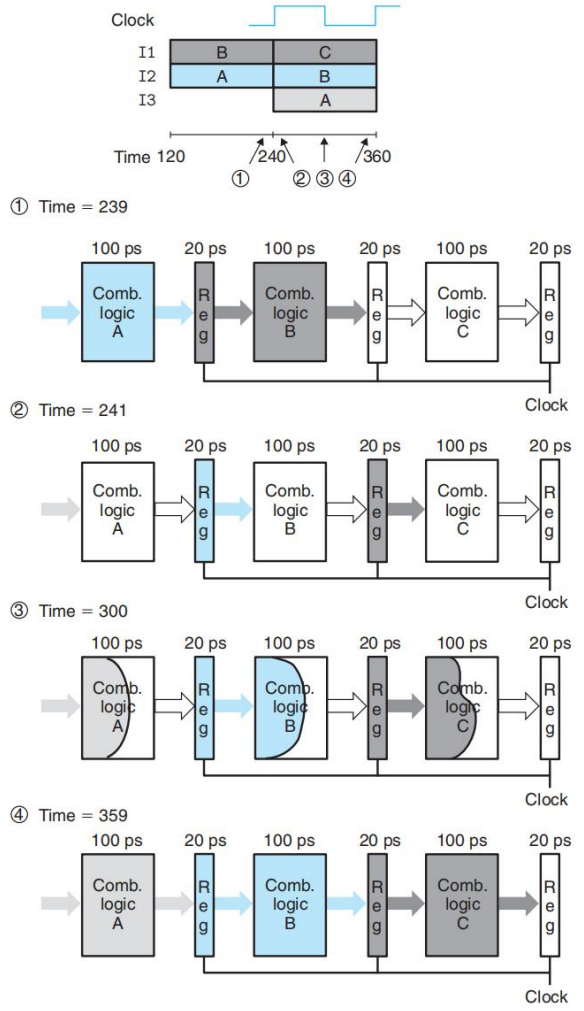
流水线：目的和方法

非流水线化设计



流水线化设计



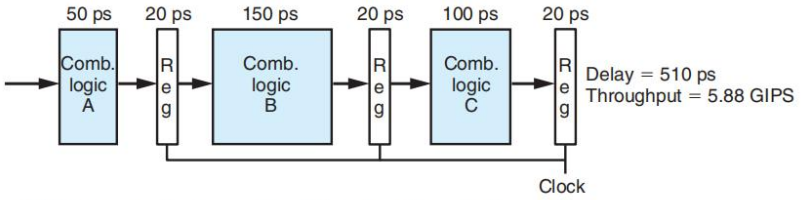


流水线原理基础

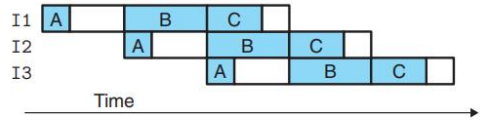
流 水 线 ： 局 限 性

不一致的划分

- 运行时钟的速率是由最慢的阶段的延迟限制的
- 系统的吞吐量由系统的**最慢阶段限制**



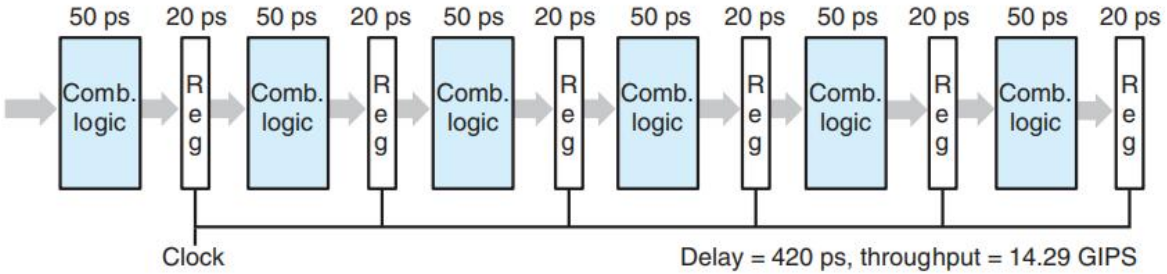
(a) Hardware: Three-stage pipeline, nonuniform stage delays



(b) Pipeline diagram

流水线过深，收益反而下降

- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining
- 设计处理器流水线时，核心思想就是**阶段尽可能多**（有15级或更多的），但是**与寄存器交互的阶段尽可能少**（减少寄存器延迟）



流水线原理基础

流水线：反馈机制

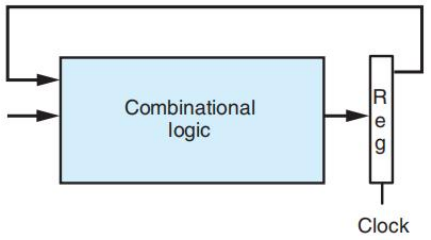
数据相关

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

顺序相关

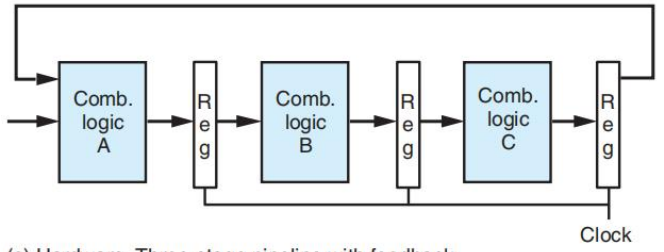
```
1    loop:
2        subq %rdx,%rbx
3        jne targ
4        irmovq $10,%rdx
5        jmp loop
6    targ:
7        halt
```

面临问题

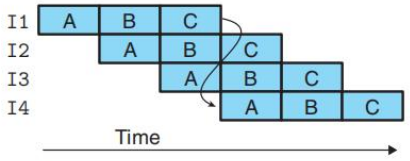


(a) Hardware: Unpipelined with feedback

(b) Pipeline diagram



(c) Hardware: Three-stage pipeline with feedback



(d) Pipeline diagram

Y86-64的流水线实现

SEQ → SEQ+

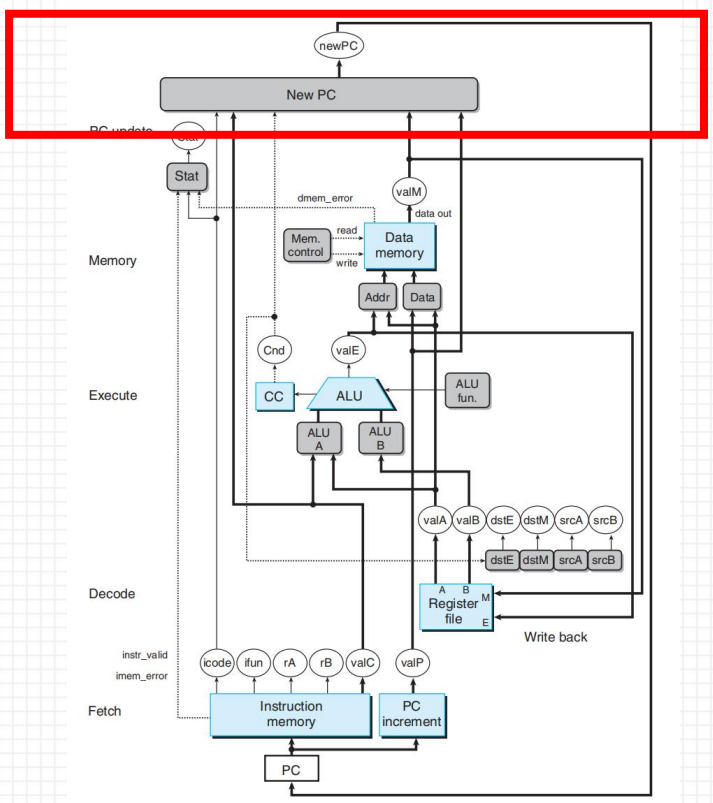


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

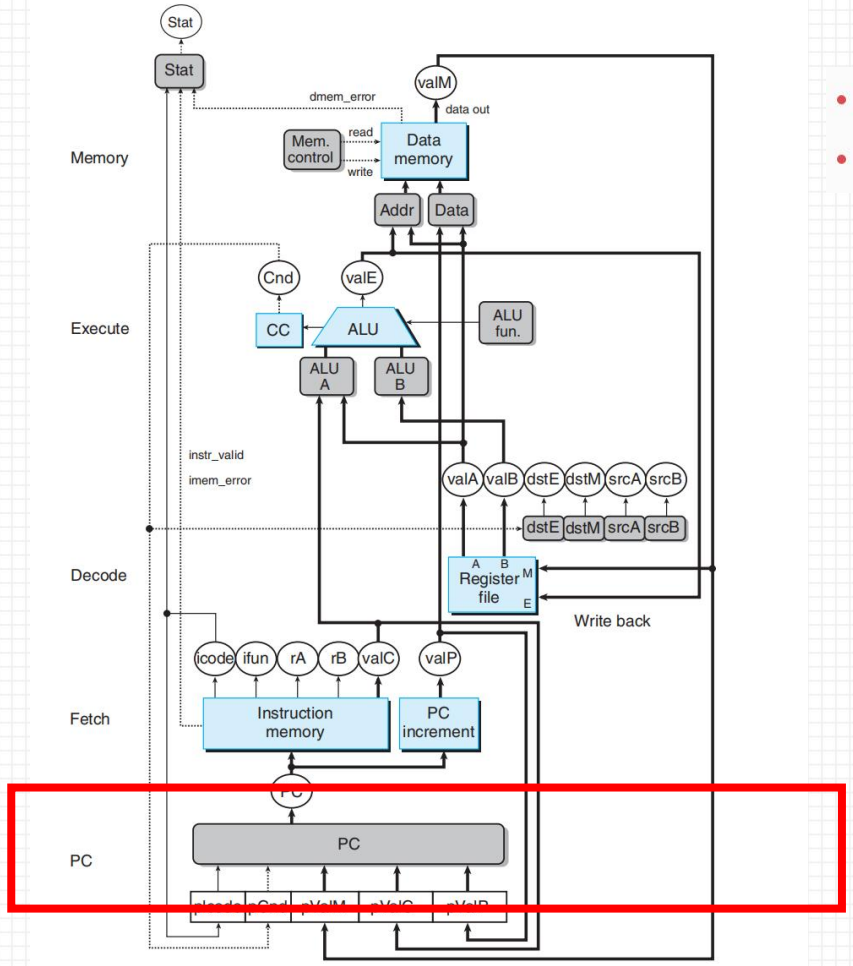


Figure 4.40 **SEQ+ hardware structure.** Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.

调整顺序

- 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

SEQ+ → PIPE-

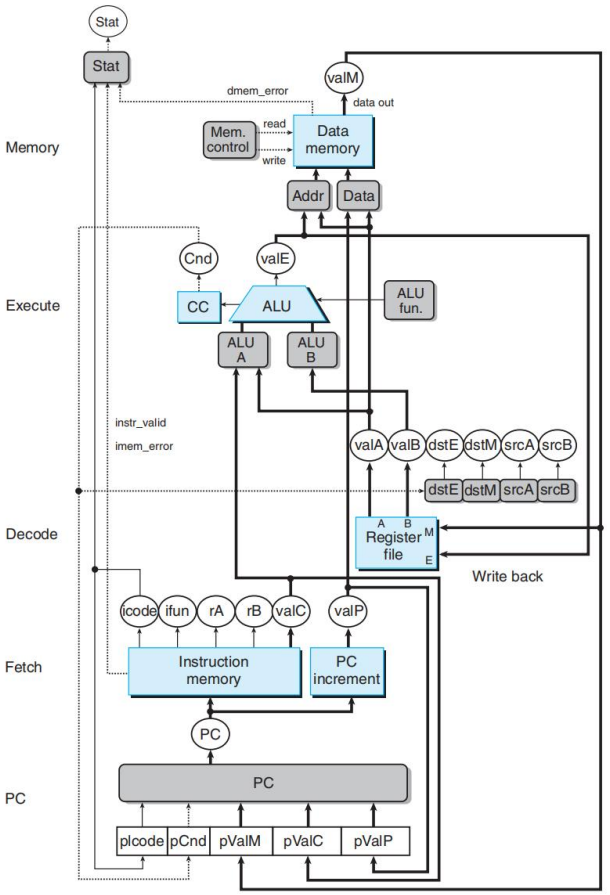


Figure 4.40 **SEQ+ hardware structure.** Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.

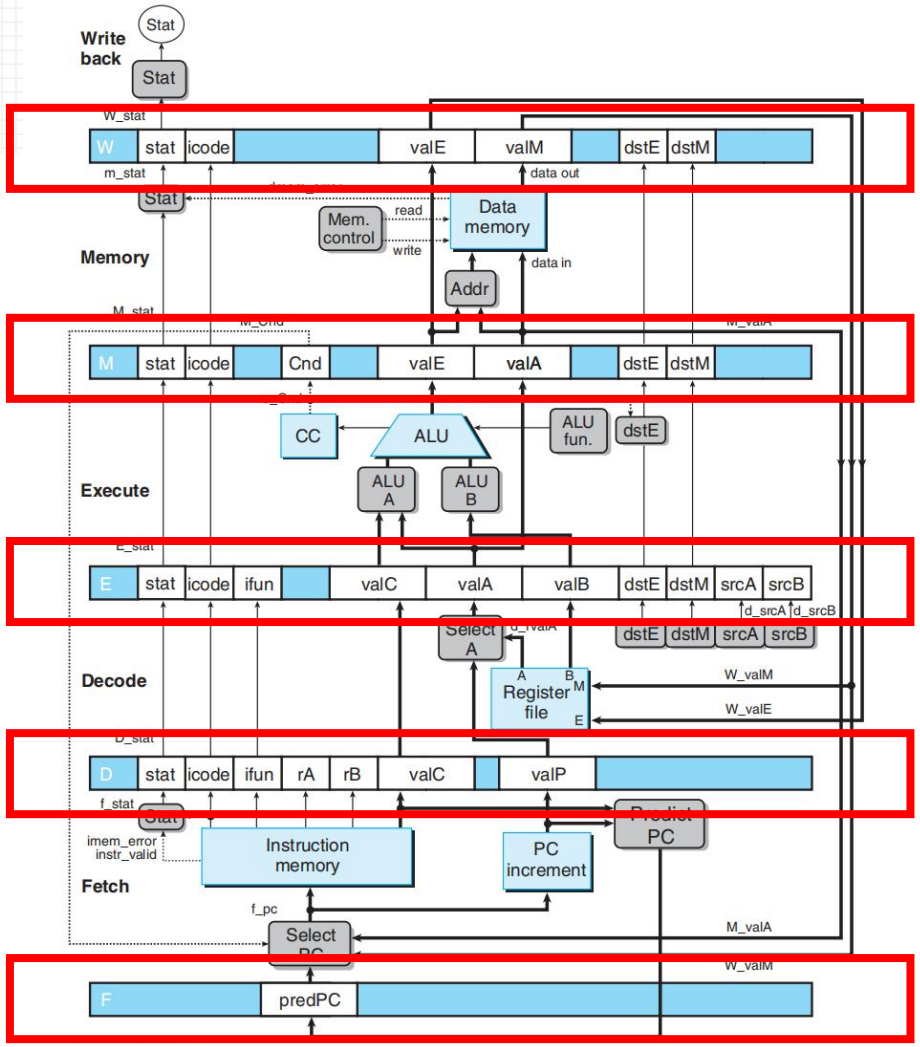


Figure 4.41 **Hardware structure of PIPE−, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

阶段划分

PIPE- detail

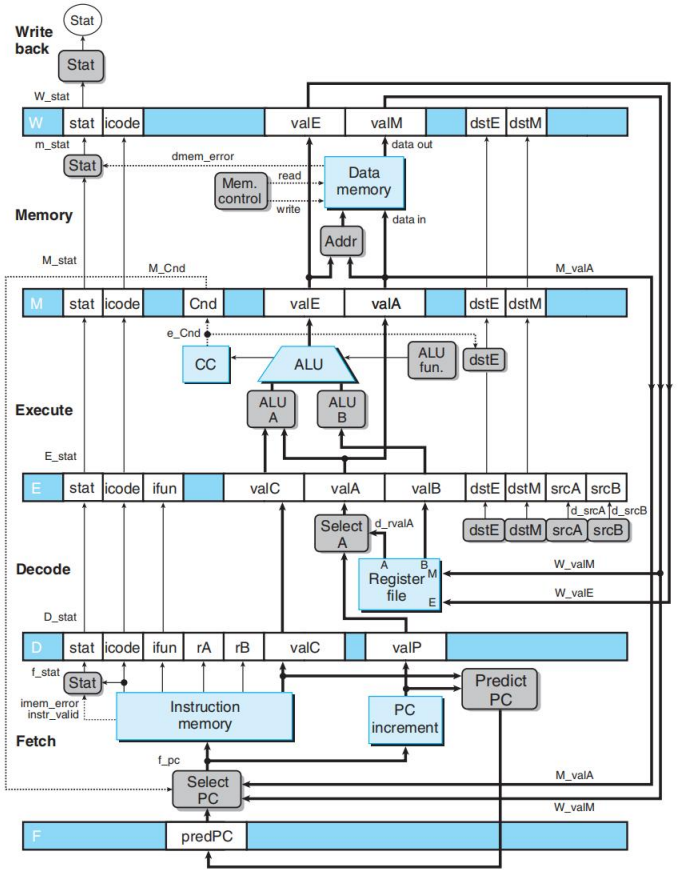


Figure 4.41 **Hardware structure of PIPE−, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

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**

信号的重新排列和标号

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

PIPE- detail

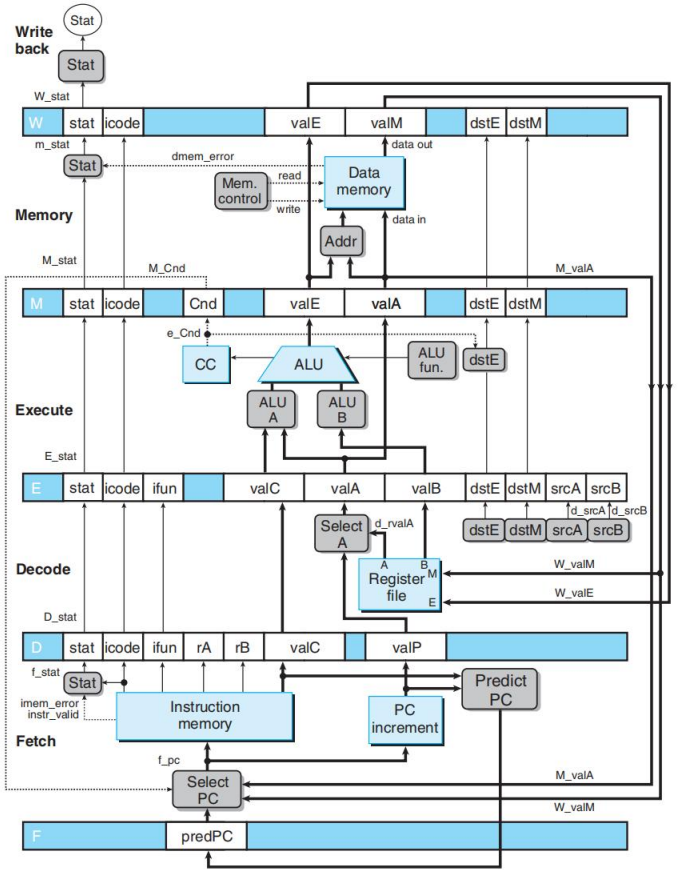


Figure 4.41 **Hardware structure of PIPE−, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

预测下一个PC

- PC Select 从三个程序计数器源中进行选择：
  - F_predPC：一般情况下一条指令
  - M_valA：预测分支错误时，从跳转到valP指向地址
  - W_valM：ret时，读出返回地址
- **条件分支：**我们可以通过**分支预测**技术来预测分支方向，并根据预测开始取值。常见的技术包括：
  - 静态分支预测
    - **总是选择（always taken，AT）：**预测PC值为`valC`，成功率大约为60%。
    - **从不选择（never taken，NT）：**预测PC值为`valP`，成功率大约为40%。
    - **反向选择、正向不选择（backward taken, forward not-taken，BTFNT）：**考虑条件分支通常用于循环操作， 成功率大约为65%。
  - 动态分支预测
    - **分支预测缓存**：包含**一个**或**多个**比特，以表明一个分支是否发生了跳转
    - **锦标赛分支预测器**：对于每个分支具有多种预测，并对预测进行选择
- `ret`指令：常见的技术包括
  - 暂停处理新指令，直到`ret`指令通过写回阶段知道下一条指令的地址
  - 在取指单元中放入一个硬件栈，保存过程调用指令产生的返回地址

PIPE- → PIPE

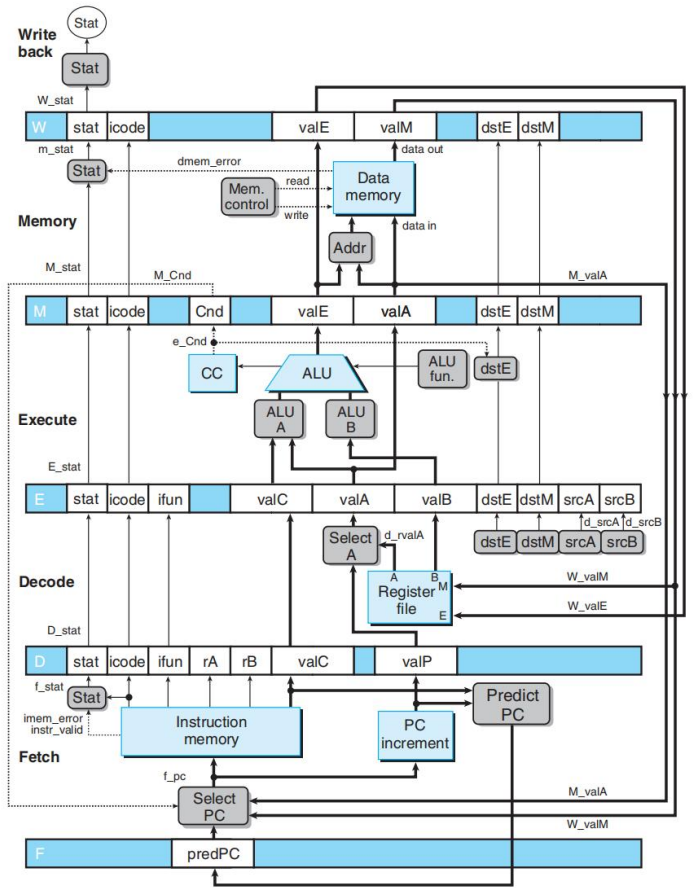


Figure 4.41 **Hardware structure of PIPE−, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

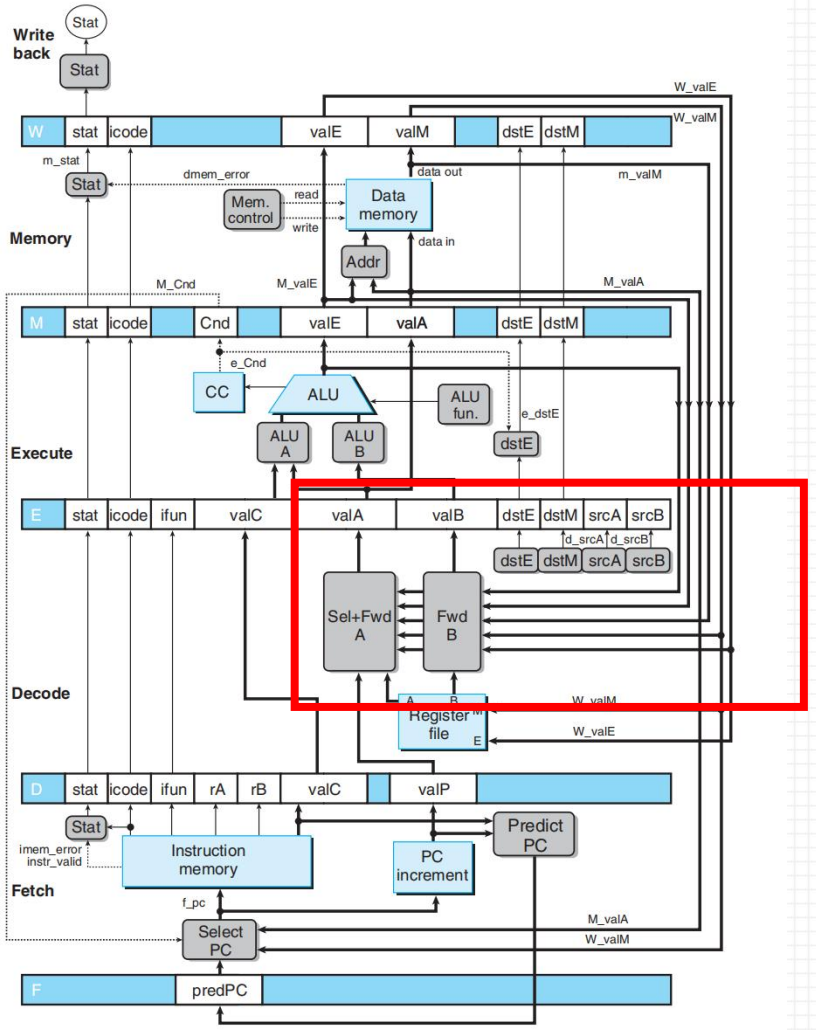


Figure 4.52 **Hardware structure of PIPE, our final pipelined implementation.** The additional bypassing paths enable forwarding the results from the three preceding instructions. This allows us to handle most forms of data hazards without stalling the pipeline.

处理冒险

结构冒险

- 特殊硬件功能单元（计算）
- TLB+Cache（访存）

数据冒险

- **暂停**（通用）
- **转发**（前后使用）
- **加载互锁**（加载/使用）

控制冒险

- **ret指令**（不预测）
- **jXX指令**（预测）

PIPE 数据冒险

若不插入：

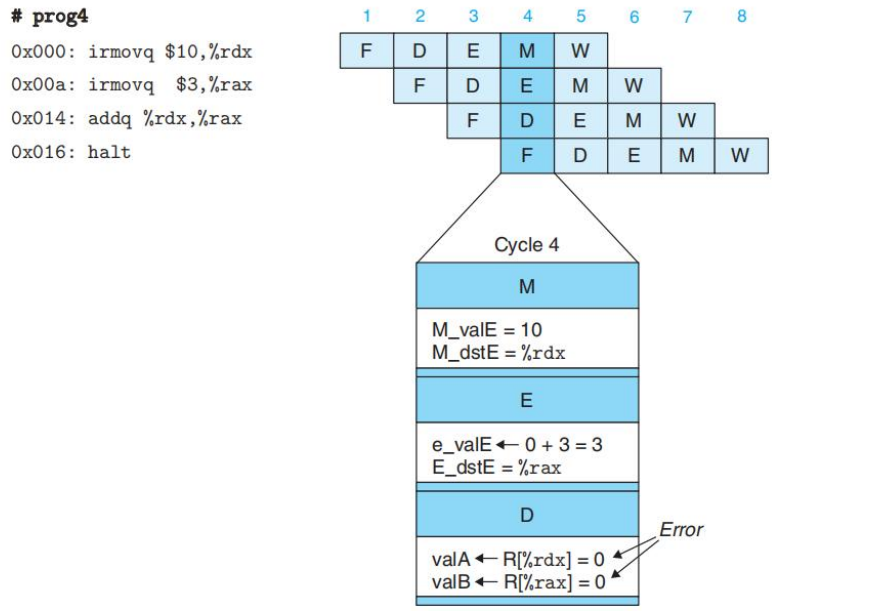


Figure 4.46 Pipelined execution of `prog4` without special pipeline control. In cycle 4, the addq instruction reads its source operands from the register file. The pending write to register %rdx is still in the memory stage, and the new value for register %rax is just being computed in the execute stage. Both operands valA and valB get incorrect values.

用暂停来避免数据冒险（通用方法）

- 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

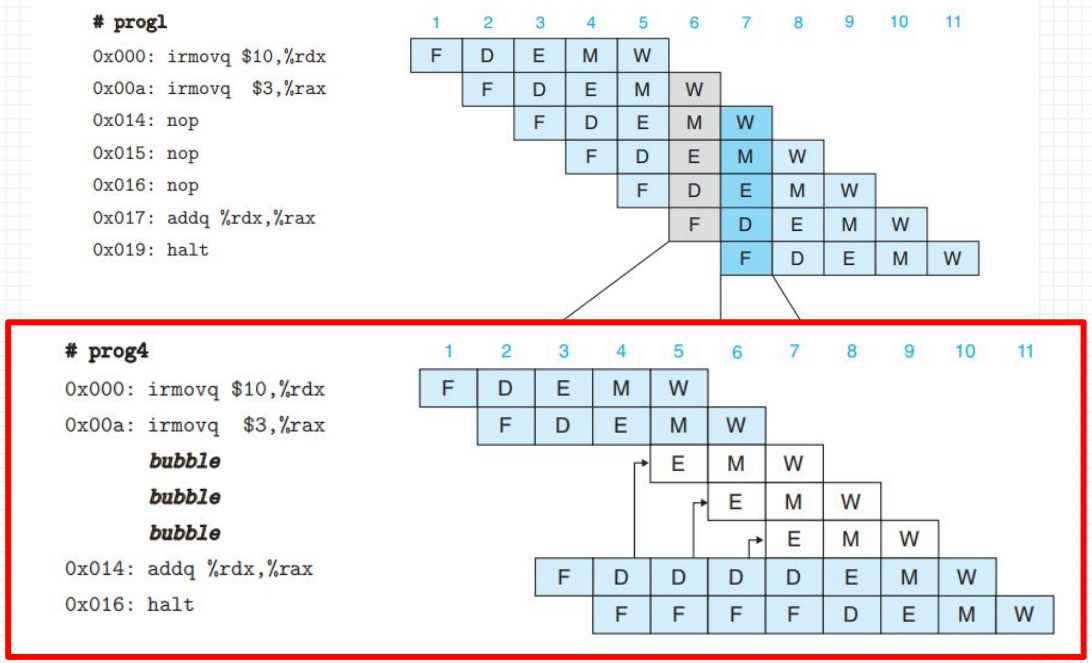
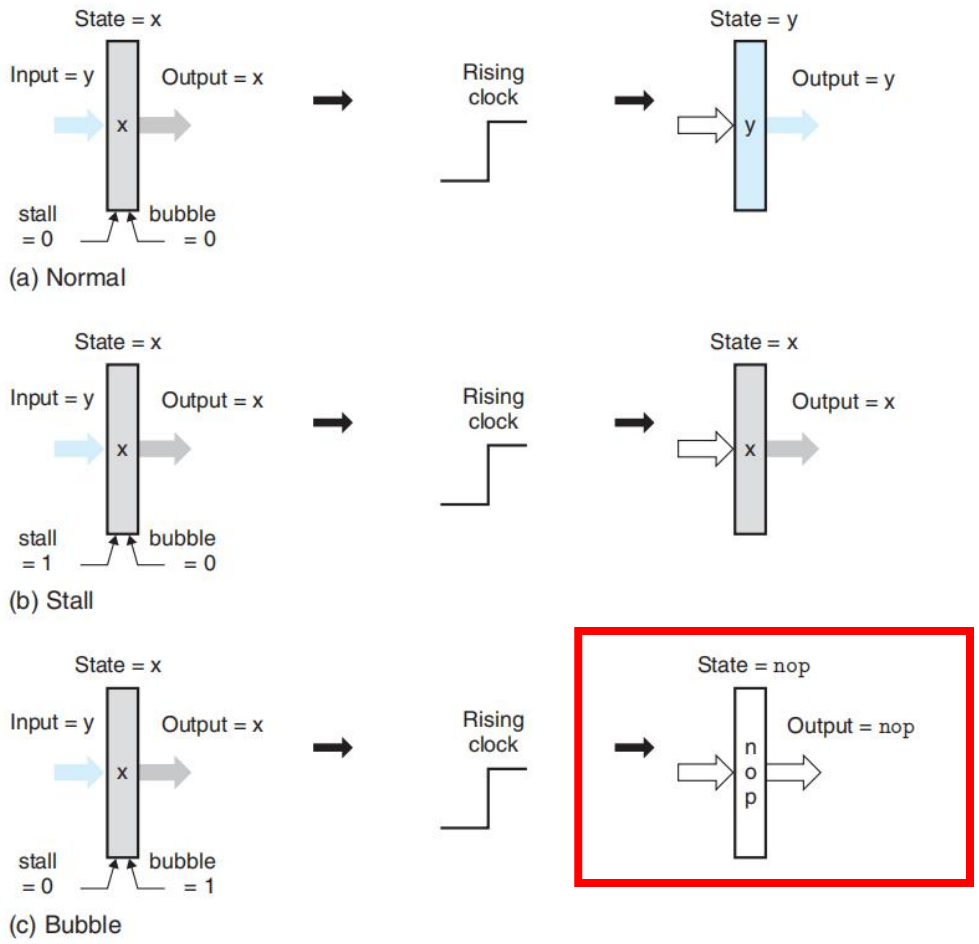


Figure 4.43 Pipelined execution of `prog1` without special pipeline control. In cycle 6, the second irmovq writes its result to program register %rax. The addq instruction reads its source operands in cycle 7, so it gets correct values for both %rdx and %rax.

PIPE 数据冒险



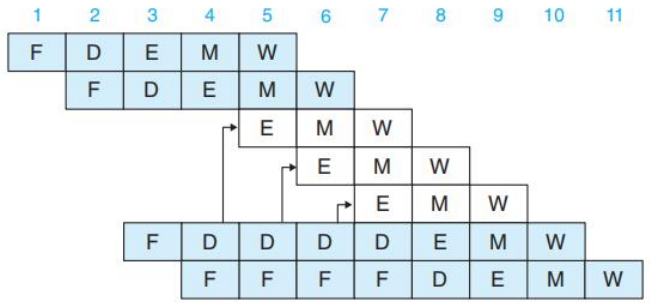
用暂停来避免数据冒险（通用方法）

- 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

暂停和气泡的控制机制

暂停能将指令阻塞在某个阶段，往流水线中插入**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

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



复位配置：INOP(+RNONE)

PIPE 数据冒险

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

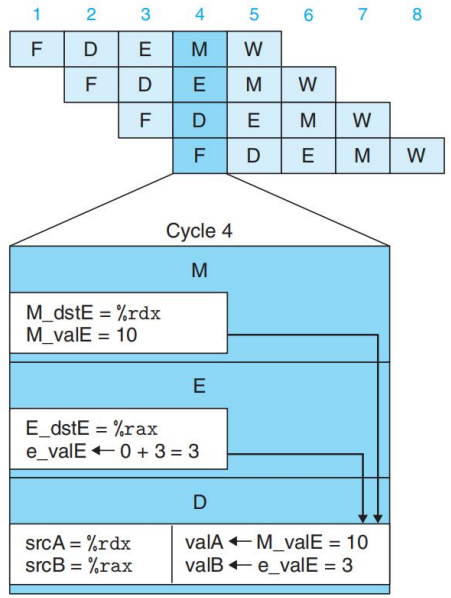


Figure 4.51 **Pipelined execution of `prog4` using forwarding.** In cycle 4, the decode-stage logic detects a pending write to register %rdx in the memory stage. It also detects that a new value is being computed for register %rax in the execute stage. It uses these as the values for valA and valB rather than the values read from the register file.

前后使用数据冒险（用转发来避免数据冒险）

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。

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

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

PIPE 数据冒险

HCL语言判断

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

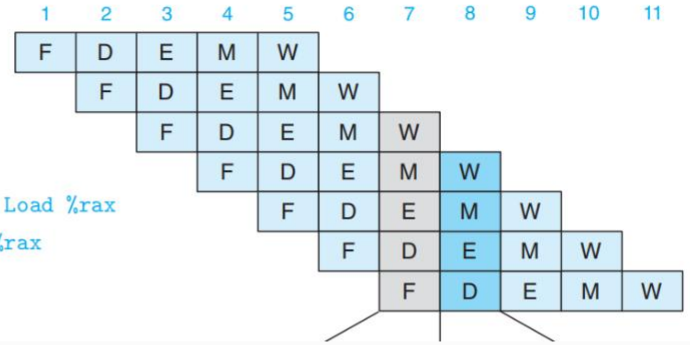
Pipeline 实现

	Pipeline register				
Condition	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~~

加载/使用数据冒险

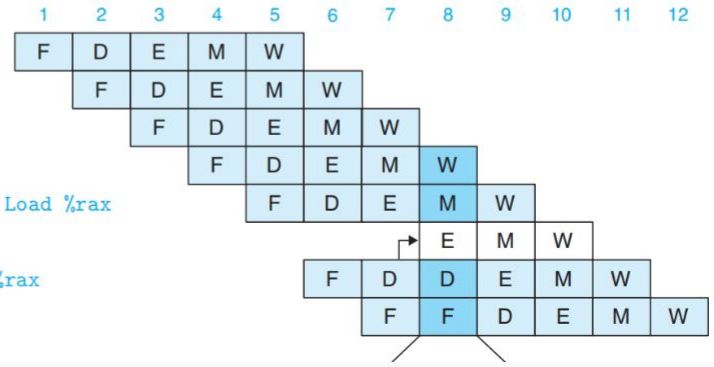
- 不能单独通过转发实现（M,D——中间跨阶段）

```
# 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
```



- **加载互锁（Load Interlock）方法**
  - 核心思想：通过暂停+转发组合实现
  - 形式：**(Install)+bubble** ——最后一定要插入bubble避免后续在流水线中继续进行

```
# 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 控制冒险

HCL语言判断

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

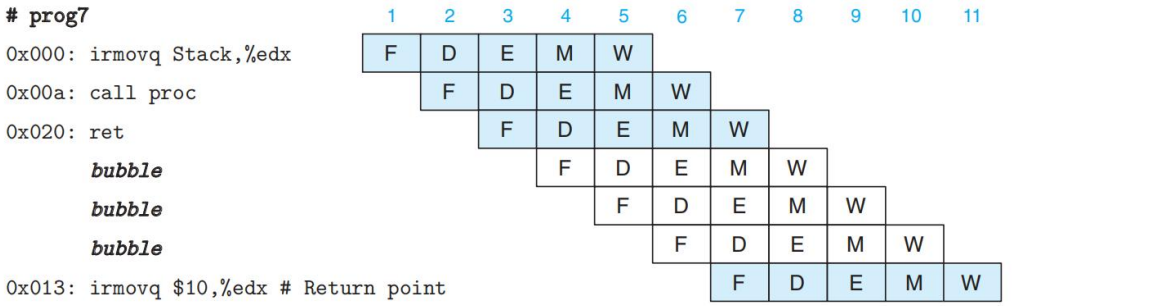
Pipeline 实现

	Pipeline register				
Condition	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

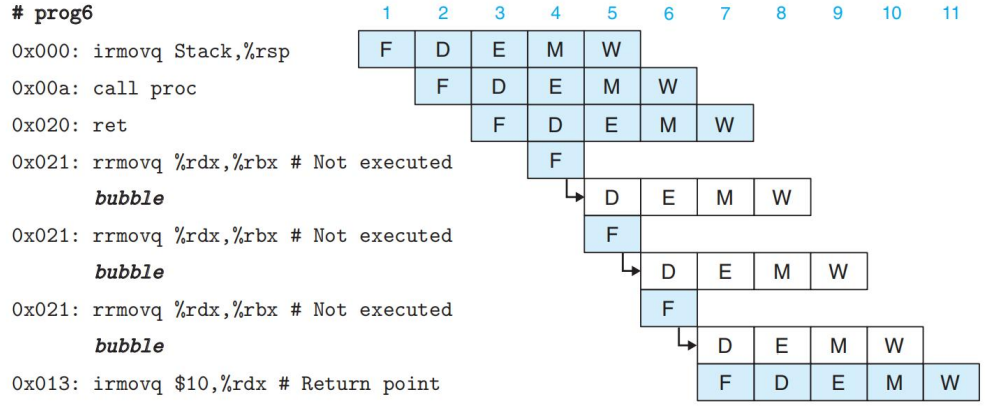
ret指令（不预测）

- 删除后续操作--**插入3个bubble**

版本1：



版本2：



PIPE 控制冒险

HCL语言判断

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

Pipeline 实现

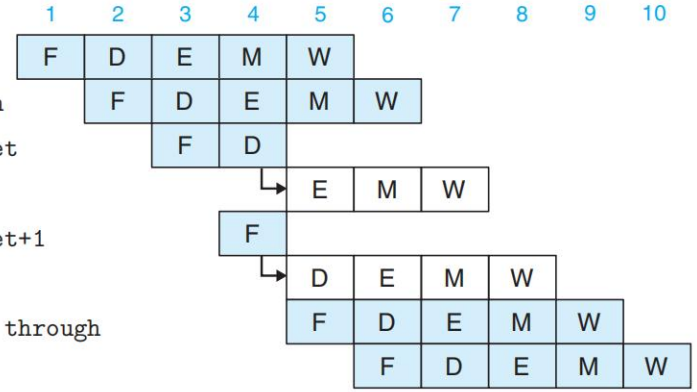
	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
~~Load/use hazard~~	~~stall~~	~~stall~~	~~bubble~~	~~normal~~	~~normal~~
Mispredicted branch	normal	bubble	bubble	normal	normal

跳转指令（预测）

（处理预测错误的分支）

- 删除后续操作——**插入2个bubble**

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



PIPE 具体实现

HCL Descriptions of Y86-64 Processors

PC选择和取指阶段

- What address should instruction be fetched at **[f_pc]**
- Determine icode of fetched instruction **[f_icode]**
- Determine status code for fetched instruction **[f_stat]**
- Predict next value of PC **[f_predPC]**

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

译码和写回阶段

- What register should be used as the A source? **[d_srcA]**
- What register should be used as the B source? **[d_srcB]**
- What register should be used as the E destination? **[d_dstE]**
- What register should be used as the M destination? **[d_dstM]**
- What should be the A value? **[d_valA]**
- What should be the B value? **[d_valB]**

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

执行阶段、访存阶段……

PIPE 全结构图

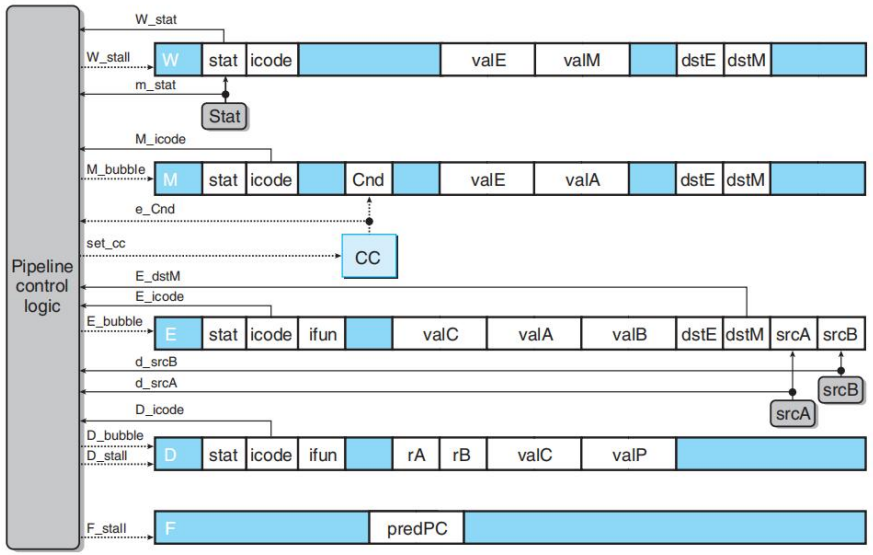


Figure 4.68 PIPE pipeline control logic. This logic overrides the normal flow of instructions through the pipeline to handle special conditions such as procedure returns, mispredicted branches, load/use hazards, and program exceptions.

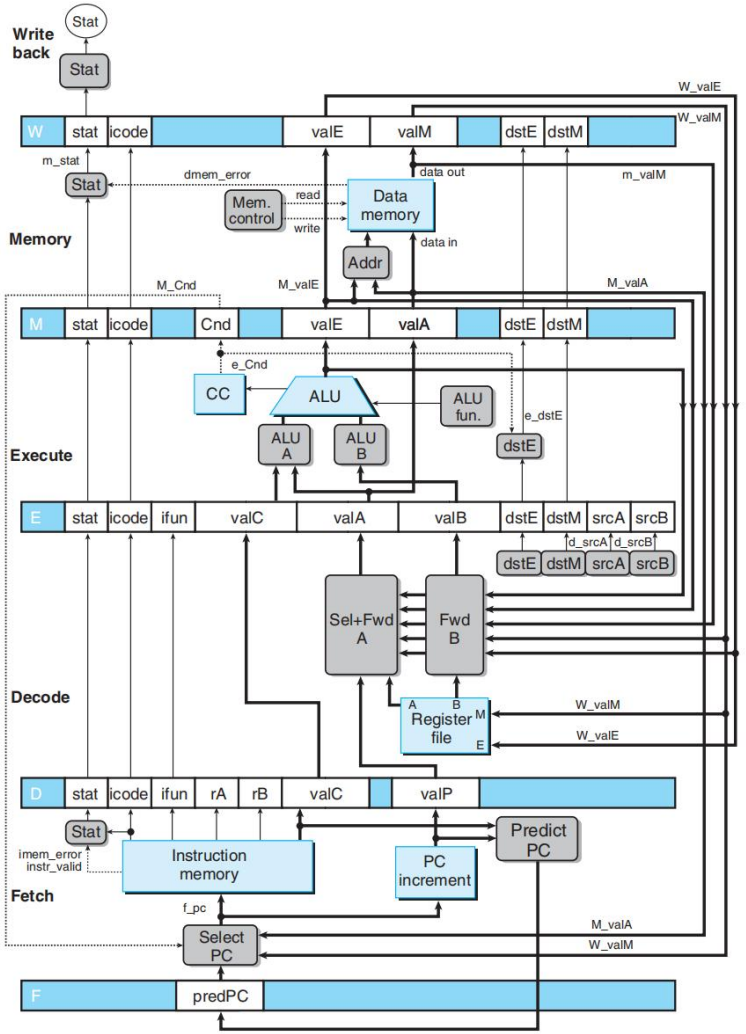
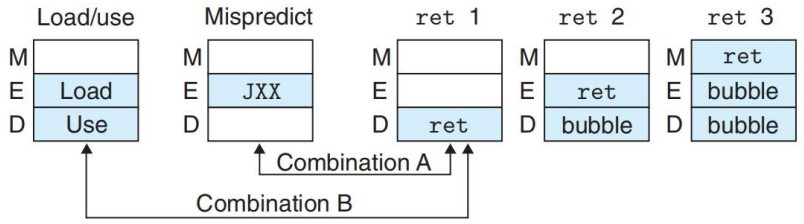


Figure 4.52 Hardware structure of PIPE, our final pipelined implementation. The additional bypassing paths enable forwarding the results from the three preceding instructions. This allows us to handle most forms of data hazards without stalling the pipeline.

机制检验

流水线：控制条件组合



控制条件组合

- 有限性组合：**Combination A + B**
  - Conbination A：**ret位于不选择分支** ——简单叠加
  - Conbination 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

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

机制检验

流水线：异常处理

```
# prog10
0x000: irmovq $1,%rax
0x00a: xorq %rsp,%rsp #CC = 100
0x00c: pushq %rax
0x00e: addq %rax,%rax
0x010: irmovq $2,%rax
```

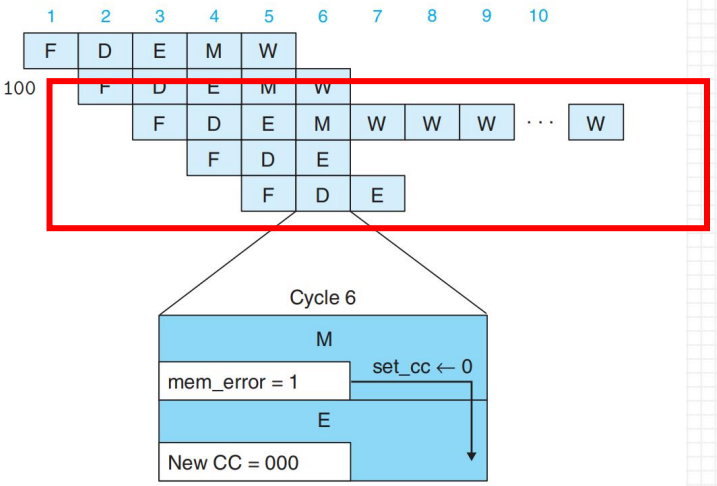


Figure 4.63 **Processing invalid memory reference exception.** On cycle 6, the invalid memory reference by the `pushq` instruction causes the updating of the condition codes to be disabled. The pipeline starts injecting bubbles into the memory stage and stalling the excepting instruction in the write-back stage.

异常处理

- 内部异常：**Stat：{HLT, ADR, INS}**
  - **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

控制权移交操作系统

- 系统例外程序计数器（SEPC）
- 系统例外原因寄存器（SCAUSE）

PC的更新？

- 非精确中断（PIPE）
- 精确中断（RISC-V）

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。

问题：

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 ← 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

机制检验

流水线：更多工作

硬件设计——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

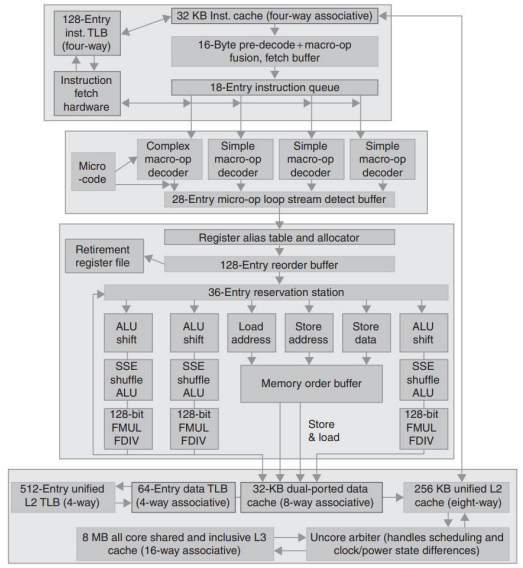


FIGURE 4.74 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready micro-operation each clock cycle.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

谢　谢　大　家　！

元培学院 常欣海

汇 报 时 间：2023.10.25