

第四章 处理器体系结构

——处理器的性能

教 师： 郑贵滨
计算机科学与技术学院
哈尔滨工业大学

目录

■ PIPE设计总结

- 异常条件
- 性能分析
- 取指阶段的设计

■ 现代高性能处理器

- 乱序执行

异常

处理器不能继续正常操作的情况

■ 原因

- 停机指令
- 取指或读数试图访问一个非法地址
- 非法指令

■ 期望的典型行为

- 完成一些指令
 - 当前的或者之前的，取决于异常类型
- 抛弃其他指令
- 调用异常处理程序
 - 类似于异常过程调用

(当前)
(当前)
(之前)



■ 我们的实现方法——当指令引起异常时就停机

异常例子

■ 取指阶段的异常

`jmp $-1` # 无效跳转目标

`.byte 0xFF` # 无效指令代码

`halt` # 停止指令

■ 访存阶段的异常

`irmovq $100,%rax`

`rmmovq %rax,0x10000(%rax)` # 无效地址

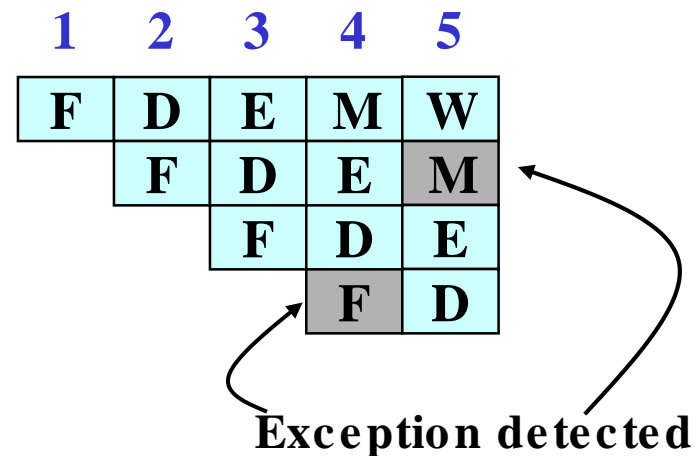
流水线处理器中的异常#1

```
# demo-excl1.js
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # 无效地址
nop
.byte 0xFF                # 无效指令代码
```

```
0x000: irmovq $100,%rax
0x00a: rmmovq %rax,0x1000(%rax)
0x014: nop
0x015: .byte 0xFF
```

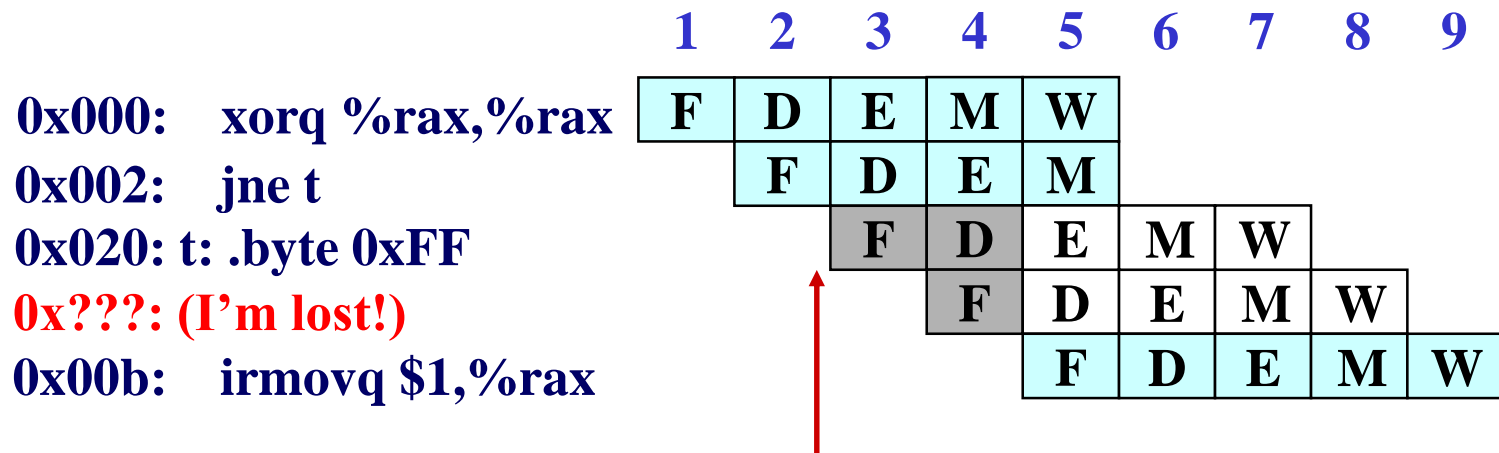
■ 期望的行为

- rmmovq 引起异常
- 其他指令不受它的影响



流水线处理器中的异常#2

```
# demo-exc2.ys
0x000:      xorq %rax,%rax      # Set condition codes
0x002:      jne t              # Not taken
0x00b:      irmovq $1,%rax
0x015:      irmovq $2,%rdx
0x01f:      halt
0x020: t:   .byte 0xFF        # Target
```

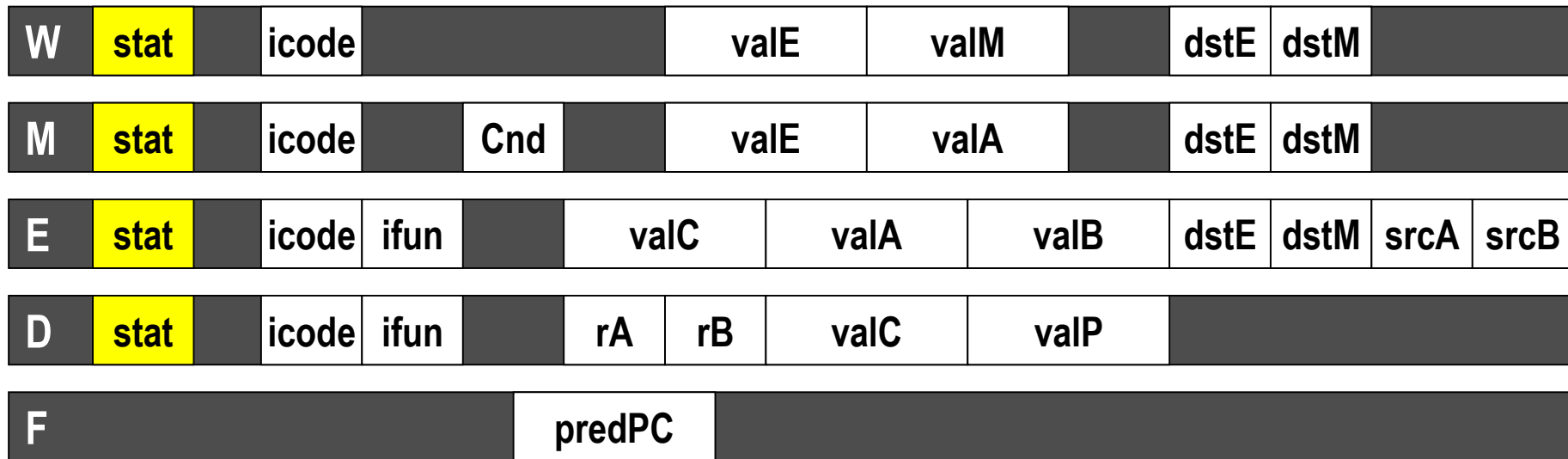


■ 期望的行为

- 没有异常发生

检测到异常

维护异常的顺序



- 为流水线寄存器增加状态字段
- 取指阶段设为“AOK,” “ADR” (当取指地址错误), “HLT” (停机指令) 或 “INS” (非法指令)
- 解码和执行阶段传递值
- 访存阶段传递或设置为“ADR”
- 当指令进入写回阶段时，异常被触发

异常处理逻辑

■ 取指

Determine status code for fetched instruction

```
int f_stat = [
    imem_error: SADR;
    !instr_valid : SINS;
    f_icode == IHALT : SHLT;
    1 : SAOK;
];
```

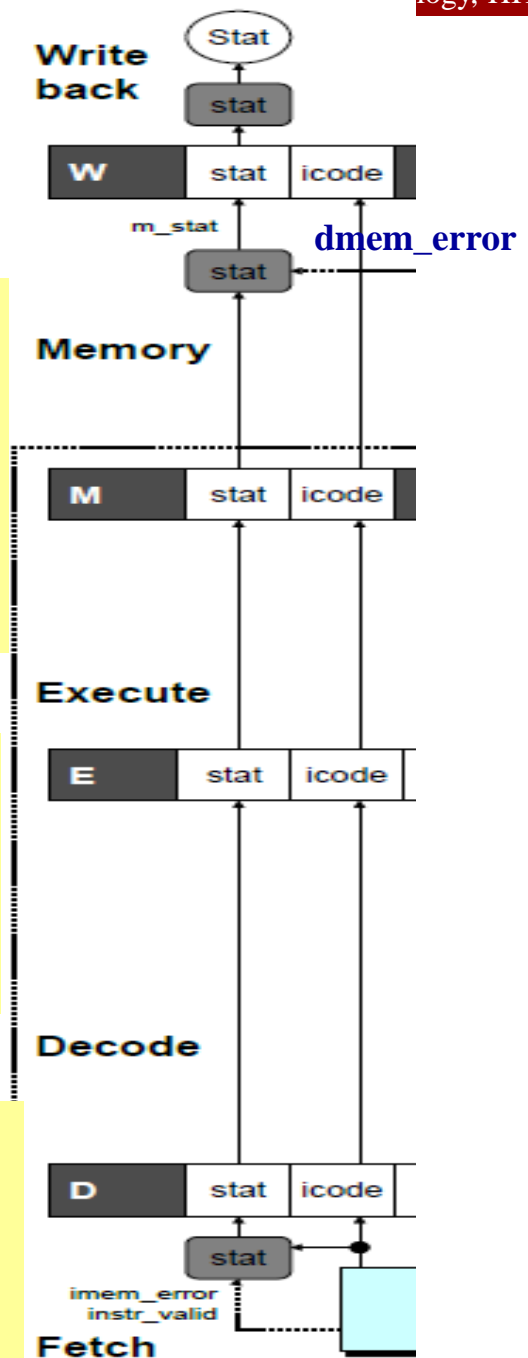
■ 访存

Update the status

```
int m_stat = [
    dmem_error : SADR;
    1 : M_stat;
];
```

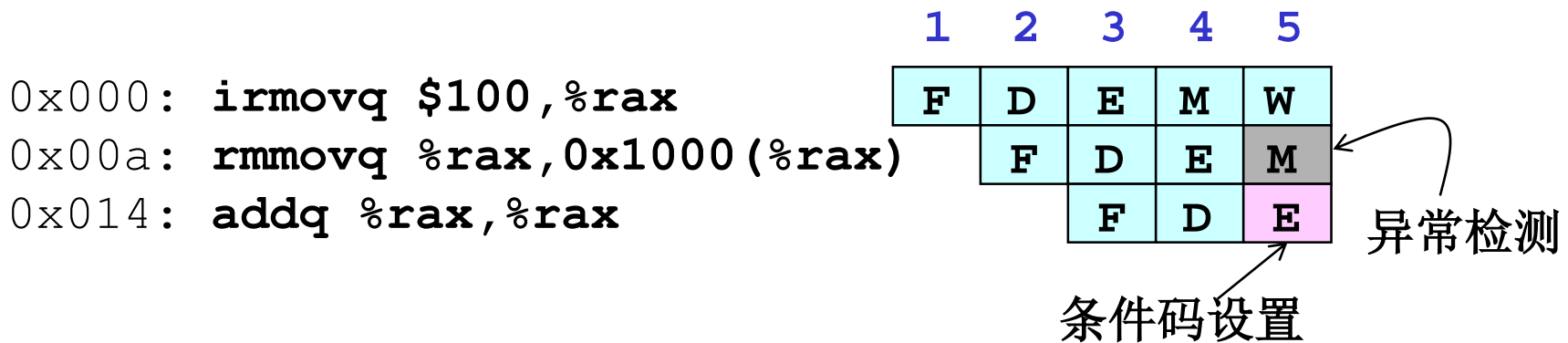
■ 写回阶段

```
int Stat = [
    # SBUB in earlier stages indicates bubble
    W_stat == SBUB : SAOK;
    1 : W_stat;
];
```



流水线处理器中的副作用

```
# demo-exc3.y
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
addq %rax,%rax             # Sets condition codes
```



■ 期望的行为

- `rmmovq` 指令引起异常
- 其他指令不受影响

避免副作用

异常出现应该禁止状态更新

- **非法指令转换为流水线气泡**
 - 除非状态指示异常状态
- **数据不会被写入无效的地址**
- **防止条件码进行无效更新**
 - 在访存阶段检测异常
 - 在执行阶段禁止条件码更新
 - 必须在相同的时钟周期内发生

避免副作用

■ 在最后阶段处理异常

- 当在访存阶段探测到异常时
 - 在下一个时钟周期将气泡插入访存阶段
- 当在写回阶段探测到异常时
 - 停止异常指令

■ 包含在HCL代码中

状态变化的控制逻辑

■ 设置条件码

Should the condition codes be updated?

```
bool set_cc = E_icode == IOPQ &&
    # State changes only during normal operation
    !m_stat in { SADR, SINS, SHLT }
    && !W_stat in { SADR, SINS, SHLT };
```

■ 阶段控制

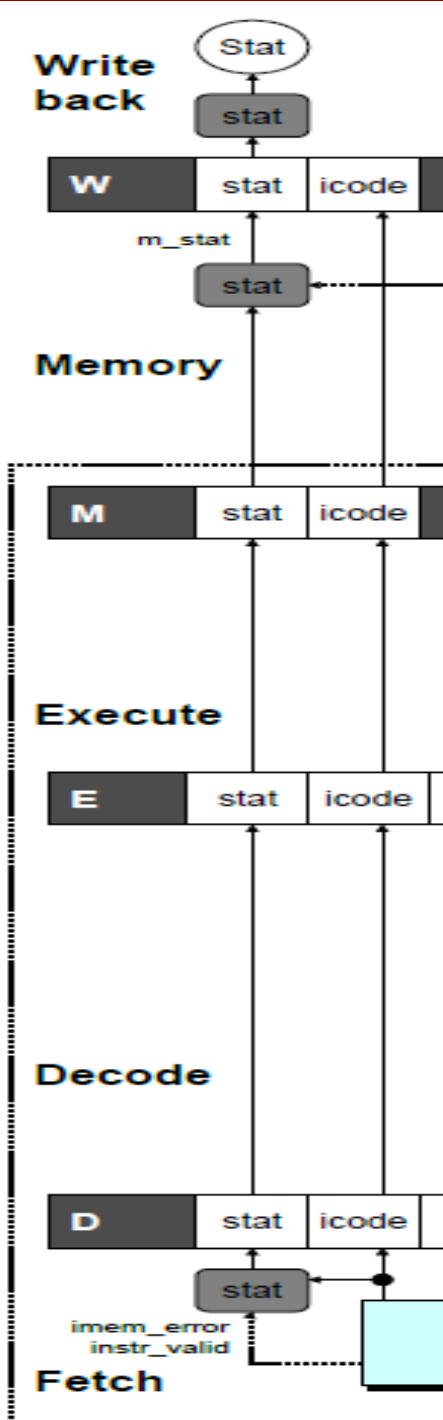
- 也控制访存阶段的更新

Start injecting bubbles as soon as exception passes through memory stage

```
bool M_bubble = m_stat in { SADR, SINS, SHLT }
    || W_stat in { SADR, SINS, SHLT };
```

stall pipeline register W when exception encountered

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



其他实际的异常处理

■ 调用异常处理程序

- 将PC入栈
 - PC指向故障指令或下一条指令
 - 通常和异常状态一起通过流水线传输
- 跳转到处理程序的入口地址
 - 通常是固定地址
 - 被定义为ISA的一部分

■ 实现

- 尚未实现！

性能评估

■ 时钟频率

- 以GHz计算
- 阶段功能的划分和电路的设计
 - 保持每个阶段的工作量尽可能的小

■ 指令的执行速率

- CPI: 每指令周期数
- 平均来说, 每条指令需要的时钟周期数
- 流水线功能设计和基准程序
 - 例如: 分支预测错误的频率

PIPE的CPI

■ CPI \approx 1.0

- 每个周期取一条指令
- 几乎每个周期都有有效的执行一条新指令
 - 虽然每个单独的指令具有5个周期的延迟

■ CPI $>$ 1.0

- 有时必须停顿或取消分支

■ 计算 CPI

- C: 时钟周期
- I: 执行完成的指令数
- B: 插入的气泡个数 ($C = I + B$)

$$\text{CPI} = C/I = (I+B)/I = 1.0 + B/I$$

- 因子B/I代表因气泡而产生的平均处罚

PIPE的CPI

Typical Values

■ 计算 CPI(...续)

$$B/I = LP + MP + RP$$

■ LP:由加载/使用冒险停顿产生的处罚

- 加载指令的比例 0.25
- 加载指令需要停顿的比例 0.20
- 每次插入气泡的数量 1

$$\Rightarrow LP = 0.25 * 0.20 * 1 = \mathbf{0.05}$$

■ MP:由错误的分支预测产生的处罚

- 条件转移指令的比例 0.20
- 条件转移预测错误的比例 0.40
- 每次插入气泡的数量 2

$$\Rightarrow MP = 0.20 * 0.40 * 2 = \mathbf{0.16}$$

PIPE的CPI

Typical Values

■ 计算 CPI(...续)

■ RP: 由ret指令产生的处罚

- 返回指令站的比例 0.02
- 每次插入的气泡数量 3

$$\Rightarrow RP = 0.02 * 3 = \mathbf{0.06}$$

$$B/I = LP + MP + RP$$

■ 处罚造成的影响（三种处罚的总和） $0.05 + 0.16 + 0.06 = 0.27$

$$\Rightarrow CPI = 1.27 \quad (\text{Not bad!})$$

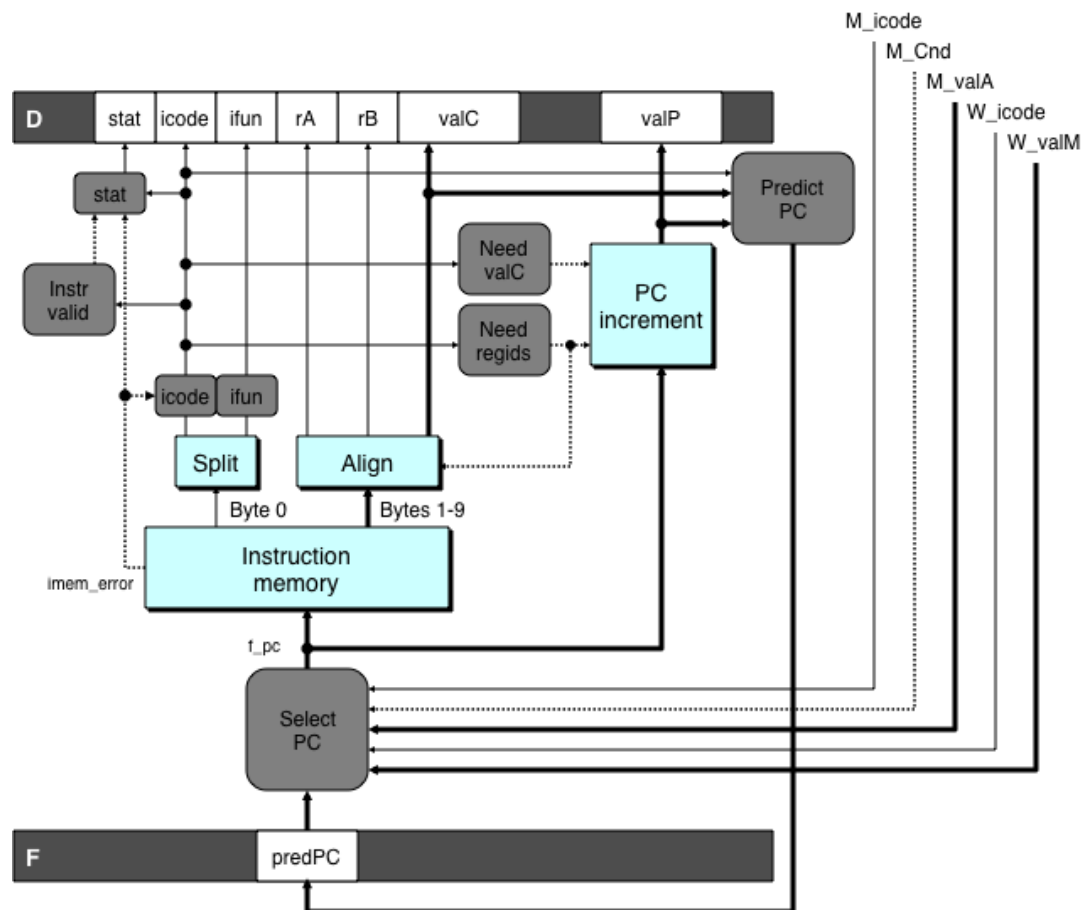
取指逻辑回顾

■ 在取指周期期间

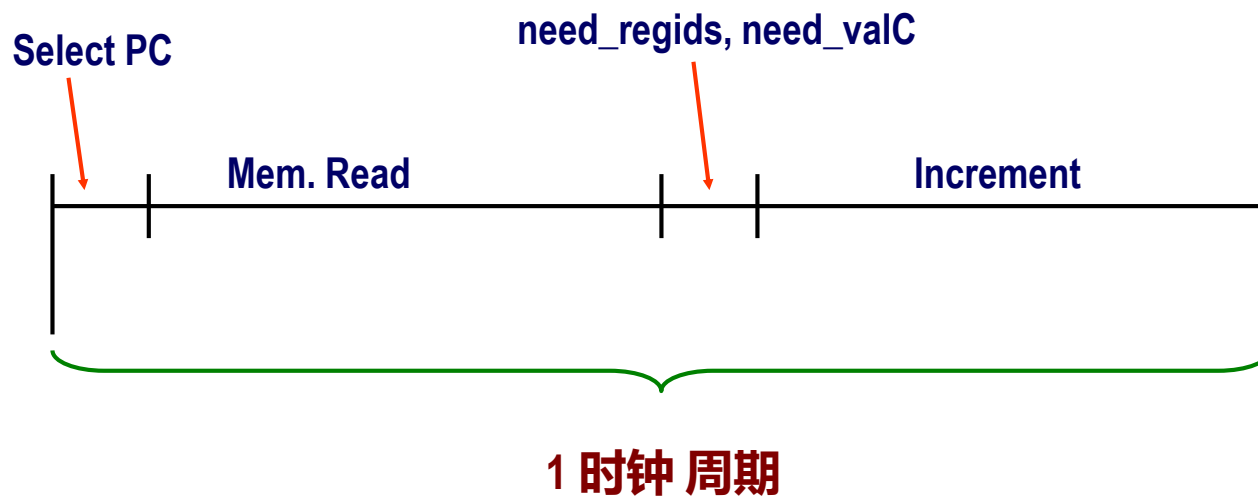
1. 选择PC
2. 从指令存储其中读取指令
3. 检查icode确定指令长度
4. 递增PC

■ 时间

- 第二、四步需要大量时间

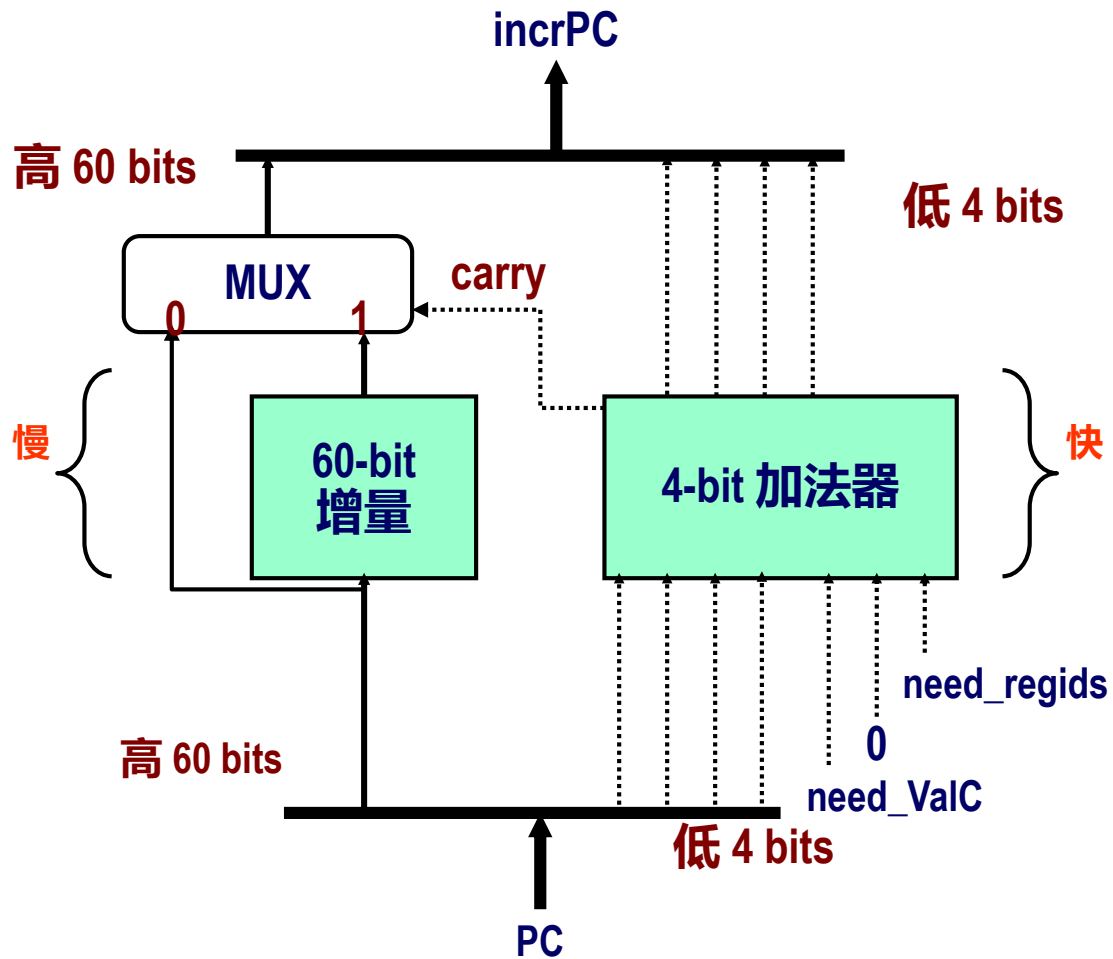


标准取指时序



- 确保每一部分都顺序执行
- 在确定PC增加多少之后才能计算PC的值

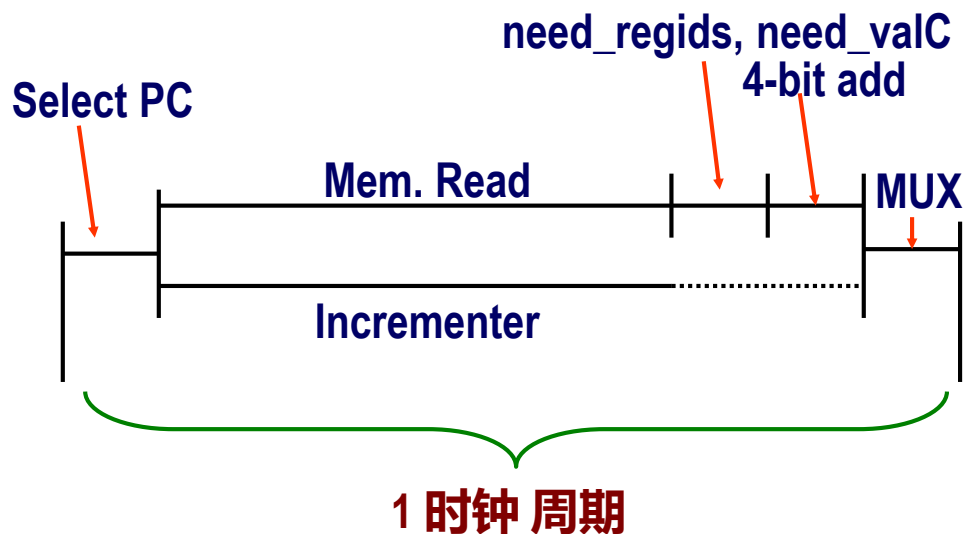
快速增加PC的电路



调整取指时间

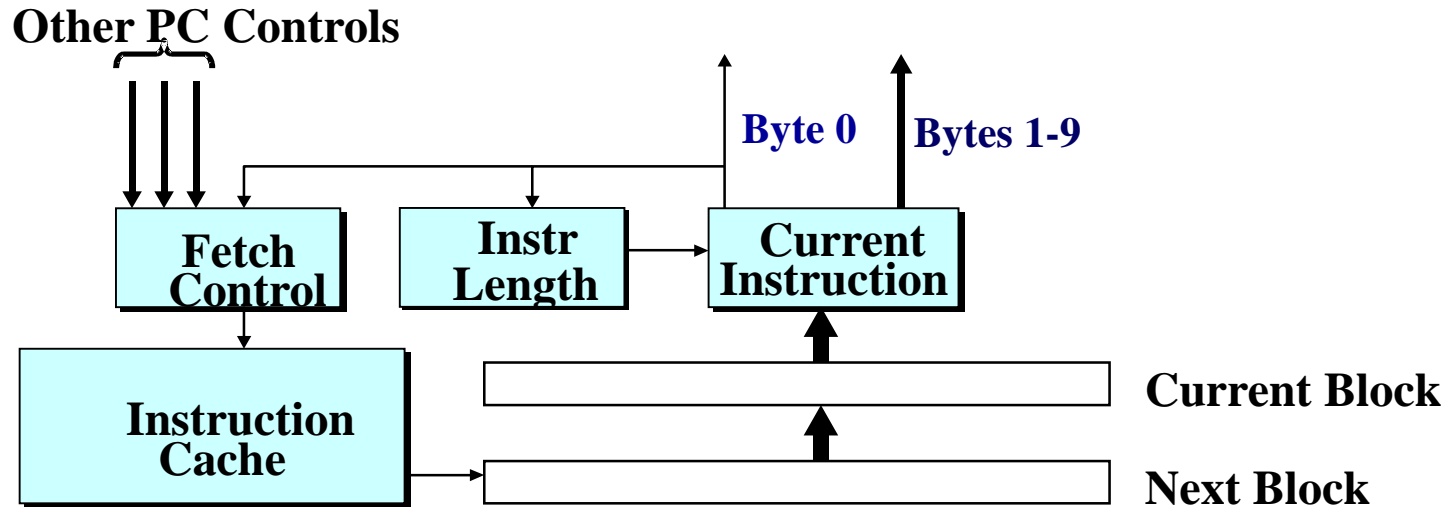
■ 60-Bit 增量

- 当PC被选择时立即执行
- 不必等到MUX再输出
- 和存储器读并行执行



标准周期

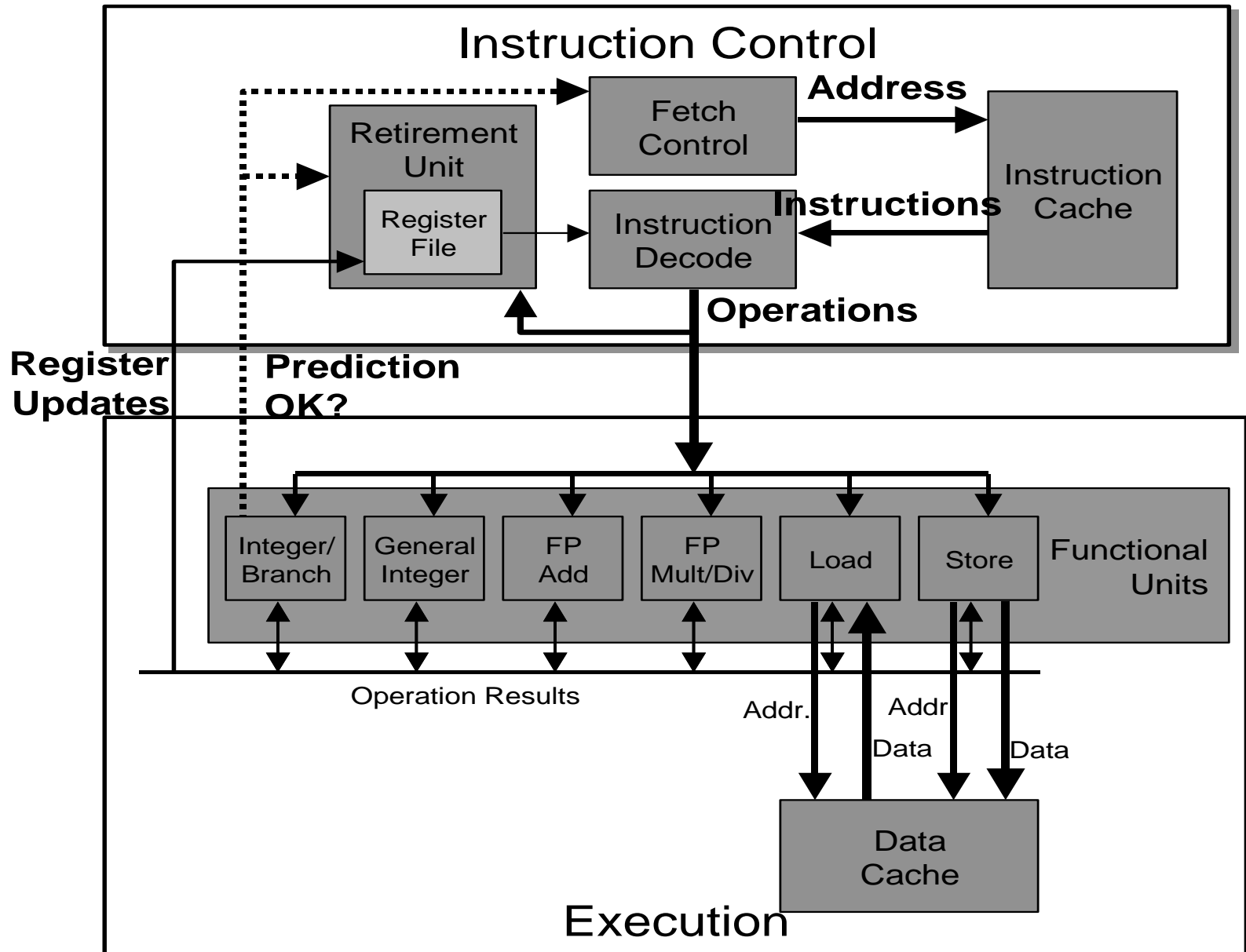
更实际的取指逻辑



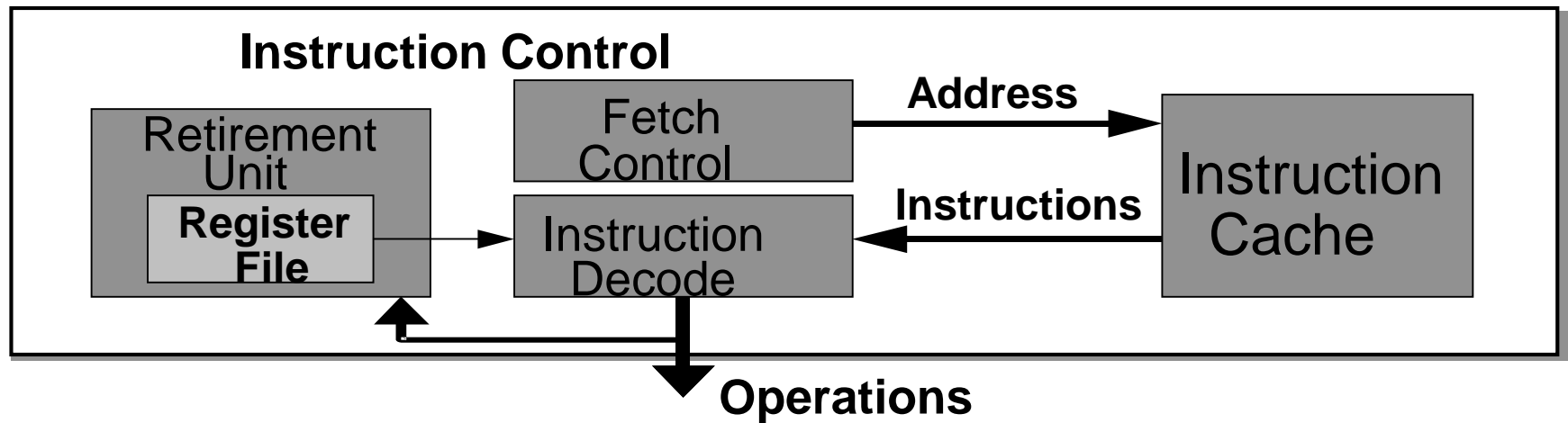
■ 取指盒子

- 集成到指令缓存
- 取整个cache块 (16 or 32 bytes)
- 从当前块选择当前指令
- 提前获取下一个块
 - 当到达当前块的末尾
 - 或在分支目标处

现代CPU设计



指令控制



■ 从内存中获取指令字节

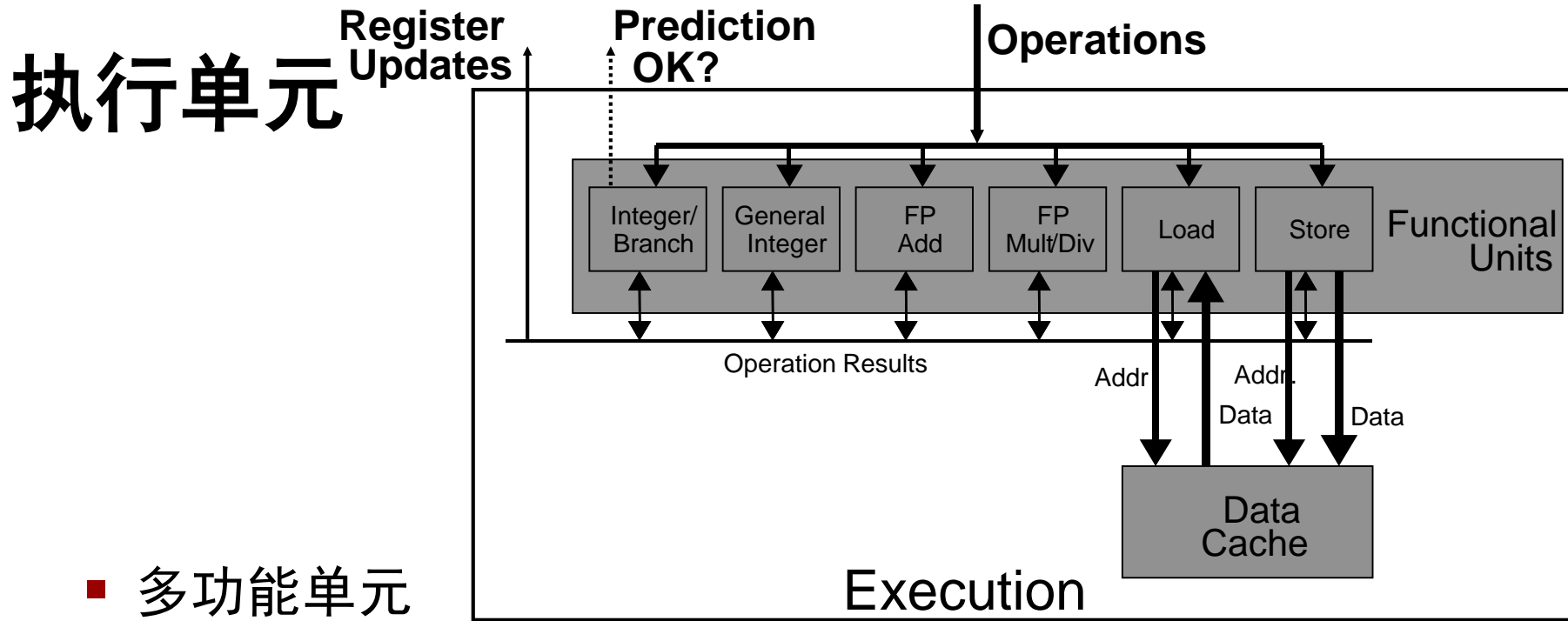
- 当前PC值加预测目标得到预测分支
- 使用硬件动态猜测是否采取/不采取分支

■ 将指令转换为操作

- 指令执行所需的基本步骤
- 典型指令需要1-3个操作

■ 将寄存器转换为标签

- 抽象标识符将一个操作的目的和后一个操作的源相连接



- 多功能单元
 - 每一个可以独立操作
- 一旦操作数就绪操作就可以执行
 - 不一定依据程序顺序执行
 - 受限于功能单元
- 控制逻辑
 - 确保执行结果和顺序执行一样

Intel Haswell CPU的能力

■ 多条指令可以并行执行

- 2 load (加载)
- 1 store (存储)
- 4 integer (整型)
- 2 FP multiply (浮点乘)
- 1 FP add / divide (浮点加/除)

■ 有些指令花费多于一个周期，但是仍可以流水化

指令	延迟	周期数 (周期s/Issue)
Load / Store	4	1
Integer Multiply	3	1
Integer Divide	3—30	3—30
Double/Single FP Multiply	5	1
Double/Single FP Add	3	1
Double/Single FP Divide	10—15	6—11

Haswell 操作

- 将指令动态转换为“Uops”
 - ~118 bits 宽
 - 保持操作，两个源，一个目的
- 使用乱序引擎执行Uops
 - 执行Uop当
 - 操作数可用
 - 功能单元可用
 - 执行由“预约站”控制
 - 跟踪uops之间的数据相关
 - 分配资源

高性能分支预测

■ 影响性能的关键

- 处理预测错误通常需要11-15个周期

■ 分支目标缓存

- 512 个目的地址
- 4 bits 用于历史信息
- 自适应算法
 - 可以识别重复的模式，例如交替跳转或不跳转

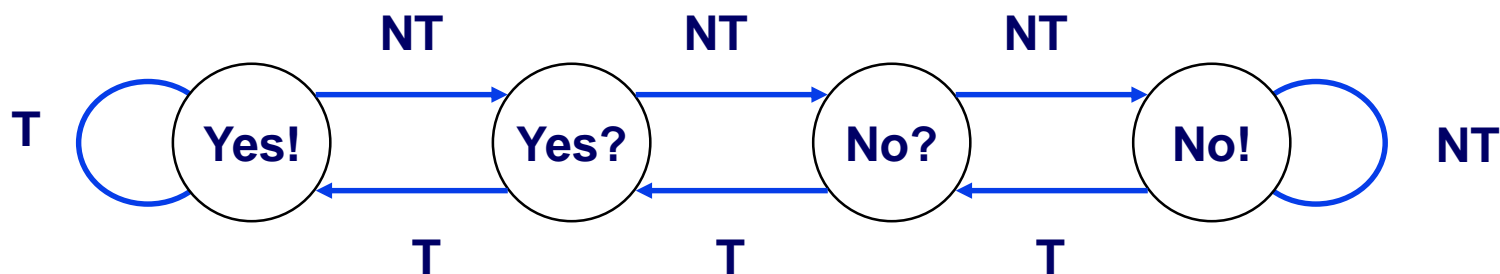
■ 处理BTB未命中

- 在第六个周期检测
- 负偏移地址时采用预测，正偏移时不采用预测
 - 循环vs条件

分支预测示例

■ 分支历史

- 编码分支指令先前的历史信息
- 预测是否采取分支



■ 状态机

- 每次采取分支后，向右过渡
- 不采取则向左过渡
- 在状态 “Yes!” 或 “Yes?” 下，预测采取分支

处理器总结

■ 设计技术

- 对所有的指令建立统一的框架
 - 便于在指令之间共享硬件
- 将标准逻辑块与控制逻辑位连接起来

■ 操作

- 状态被保存在存储器或时钟寄存器
- 组合逻辑进行计算
- 寄存器/存储器时钟用于控制整体的行为

■ 提高性能

- 流水化提高了吞吐量和资源利用率
- 必须保证服从ISA行为