第四章 处理器体系结构

4-6 ——处理器的性能

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条件)

- **原因**
  - 停机指令 (当前)
  - 取指或读数试图访问一个非法地址 (之前)
  - 非法指令 (之前)
- **期望行为**
  - 完成一些指令
    - 或者当前或者之前，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
- **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
```

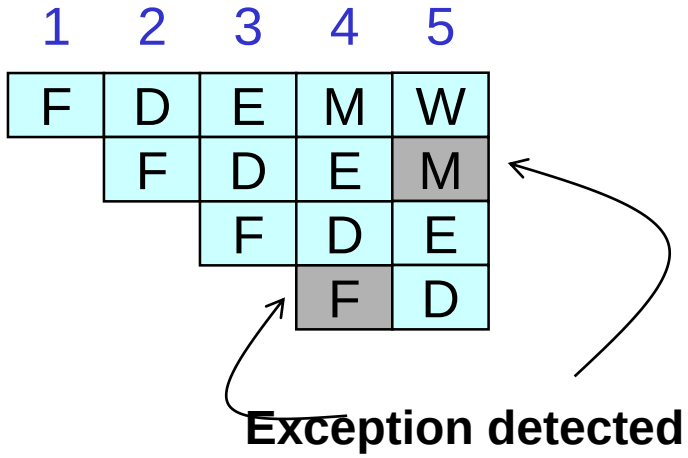
访存阶段的异常

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax)  #假设0x10064是无效地址
```

流水线处理器中的异常#1

```
# demo-exc1.ys
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
```

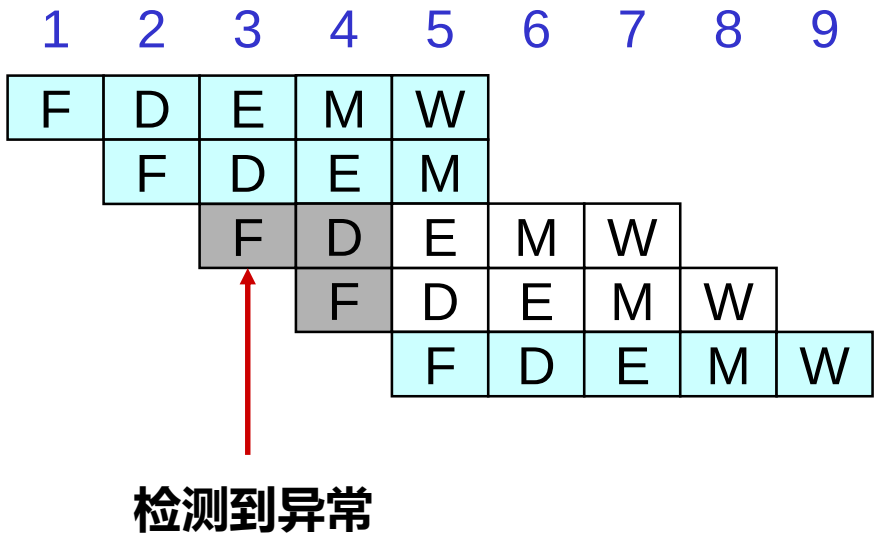


- 期望的行为
  - rmmovq 引起异常
  - 其他指令不受它的影响

流水线处理器中的异常#2

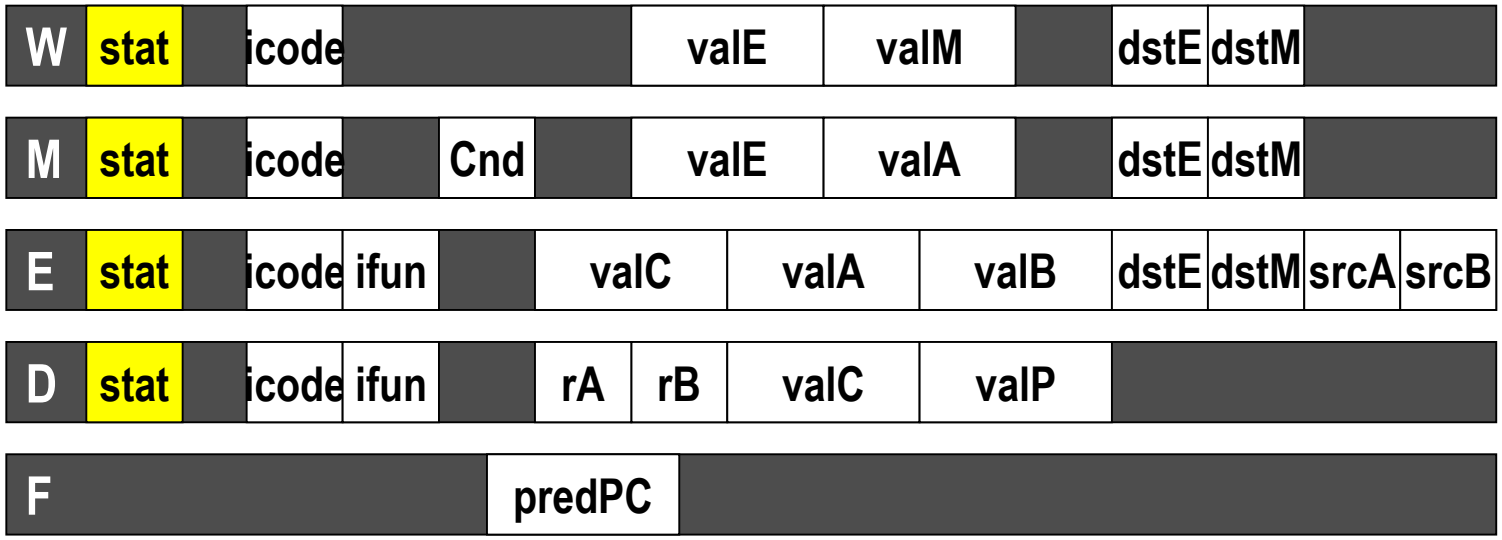
```
# demo-exc2.ys
0x000:     xorq %rax,%rax    # Set condition codes
0x002:     jne t             # Not taken
0x00b:     irmovq $1,%rax
0x015:     irmovq $2,%rdx
0x01f:     halt
0x020: t:  .byte 0xFF        # Target
```

```
0x000:     xorq %rax,%rax
0x002:     jne t
0x020: t:  .byte 0xFF
0x???: (I'm lost!)
0x00b:     irmovq $1,%rax
```



- 期望的行为
  - 没有异常发生

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) 或者 “INS” (非法指令)
- 译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

- 取指阶段

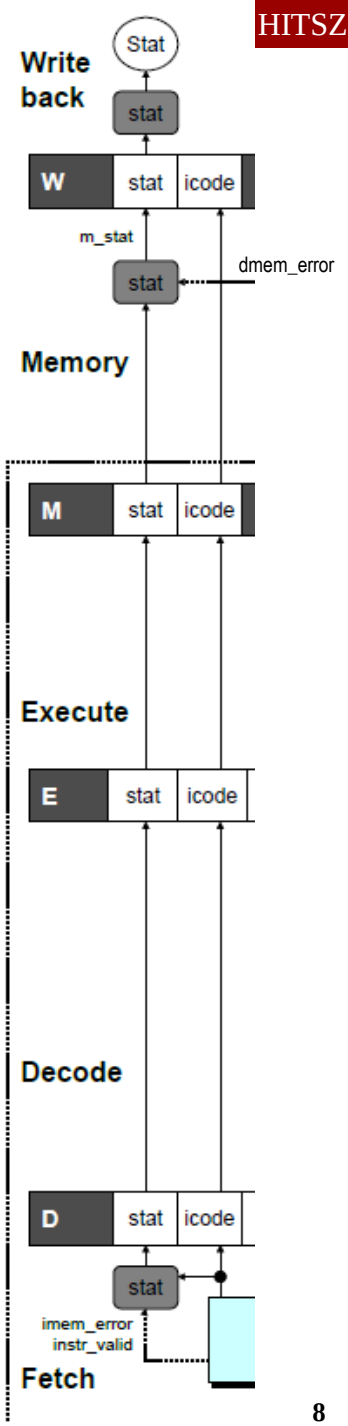
```
# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

- 访存阶段

```
# Update the status
int m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];
```

- 写回阶段

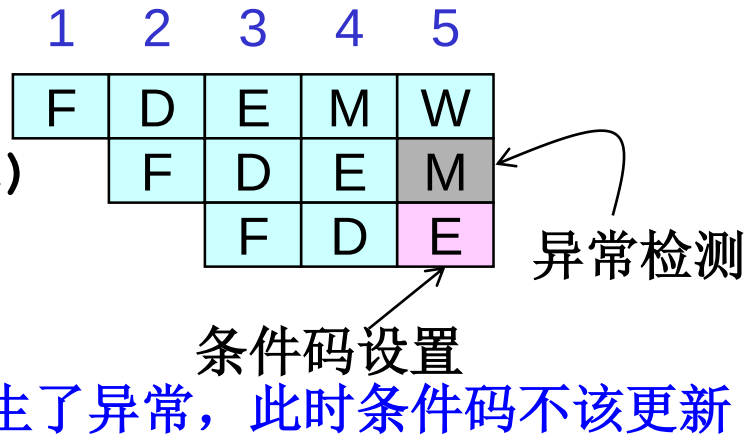
```
int Stat = [
        # SBUB in earlier stages indicates bubble
        W_stat == SBUB : SAOK;
        1 : W_stat;
];
```



流水线处理器中的副作用

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

```
0x000: irmovq $100,%rax
0x00a: rmmovq %rax,0x1000(%rax)
0x014: addq %rax,%rax
```



- 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，此时的执行阶段硬件禁止条件码更新
  - 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 （此时对控制信号进行清0）
    - 当在写回阶段探测到异常时
      - 停止异常指令（让其对寄存器文件写使能失效）
  - 包含在HCL代码中

状态变化的控制逻辑

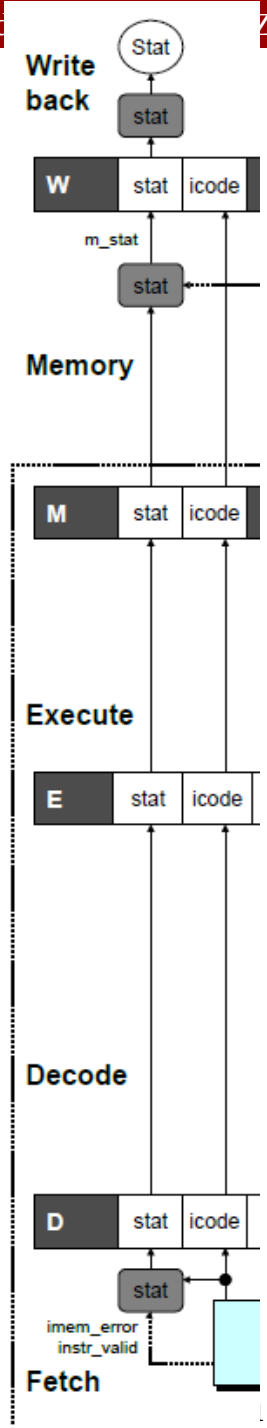
- 设置条件码

```
# 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！---OS部分。很多异常系统仅仅发送了一个信号，交给操作系统来处理。

1. 将 PC 入栈

当异常发生时，当前程序计数器（PC）的值会被保存到栈中。这一步是为了在异常处理完成后能够恢复程序的执行位置。根据异常的类型，PC 可能指向故障指令本身或下一条指令。这取决于异常的性质和设计的异常处理策略。

指向故障指令：如果异常是由于当前指令引起的，PC 通常会指向该指令，以便异常处理程序可以重新执行或跳过该指令。

指向下一条指令：如果异常是由于当前指令的副作用引起的，PC 可能会指向下一条指令，以便程序可以从异常中恢复并继续执行。

异常状态（如异常类型和相关上下文信息）通常会通过流水线传输，以便异常处理程序可以获取这些信息。这些状态信息可能包括异常类型、故障地址等。

2. 跳转到处理程序的入口地址

异常处理程序的入口地址通常是固定的，由指令集架构（ISA）定义。处理器会跳转到这个固定地址，开始执行异常处理程序。异常处理程序的入口地址通常是固定的，由 ISA 定义。例如，在 x86 架构中，异常处理程序的入口地址通常存储在中断描述符表（IDT）中。在 RISC-V 架构中，异常处理程序的入口地址通常是一个固定的地址，如 0x1000。

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（Cycle Per Instruction）

- **CPI ≈ 1.0**
  - 每个周期取一条指令
  - 几乎每个周期都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**

可能会出考试题

  - 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)

$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

  - 因子B/I代表因气泡而产生的平均处罚

PIPE的CPI（续)

B/I = LP + MP + RP

Typical Values

- 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

  ⇒ LP = 0.25 * 0.20 * 1 = 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

  ⇒ MP = 0.20 * 0.40 * 2 = 0.16
- 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

  ⇒ RP = 0.02 * 3 = 0.06

PIPE的CPI（续)

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

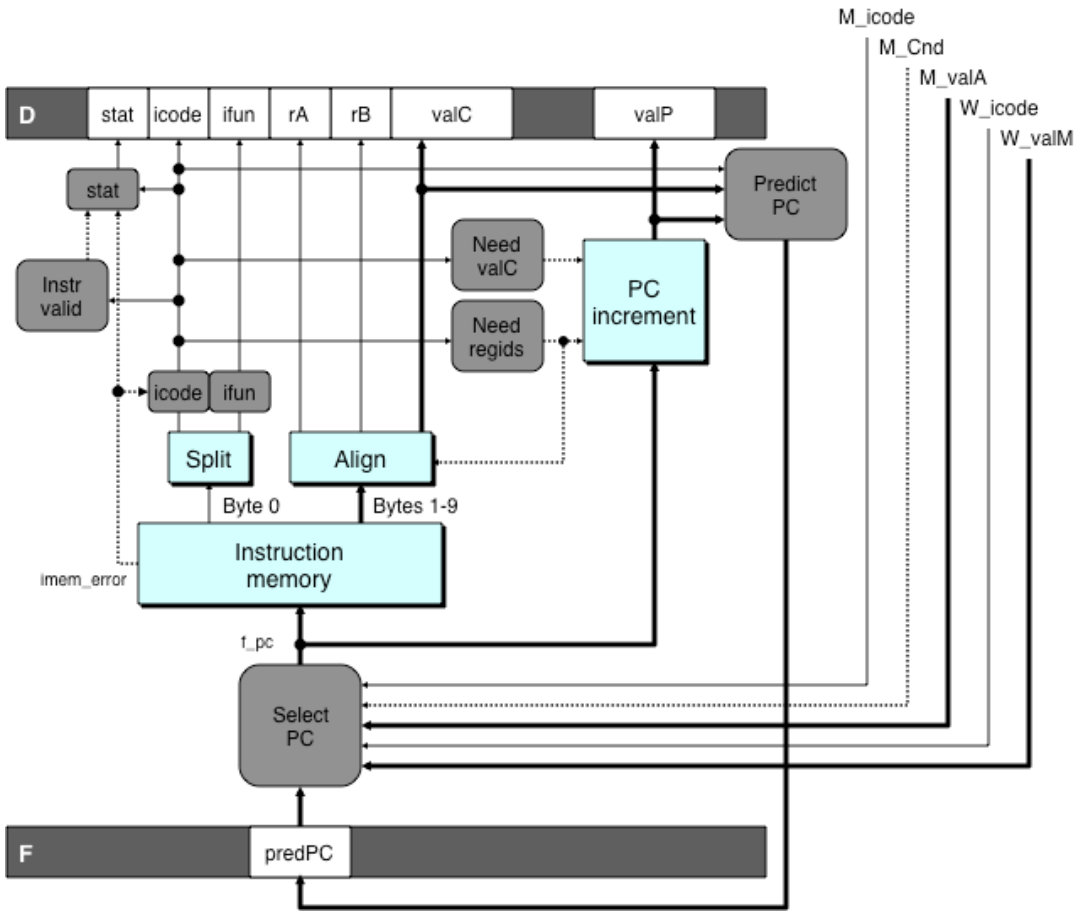
- 处罚造成的影响（三种处罚的总和) 0.05 + 0.16 + 0.06 = 0.27

  ⇒ CPI = 1.27　(Not bad!)

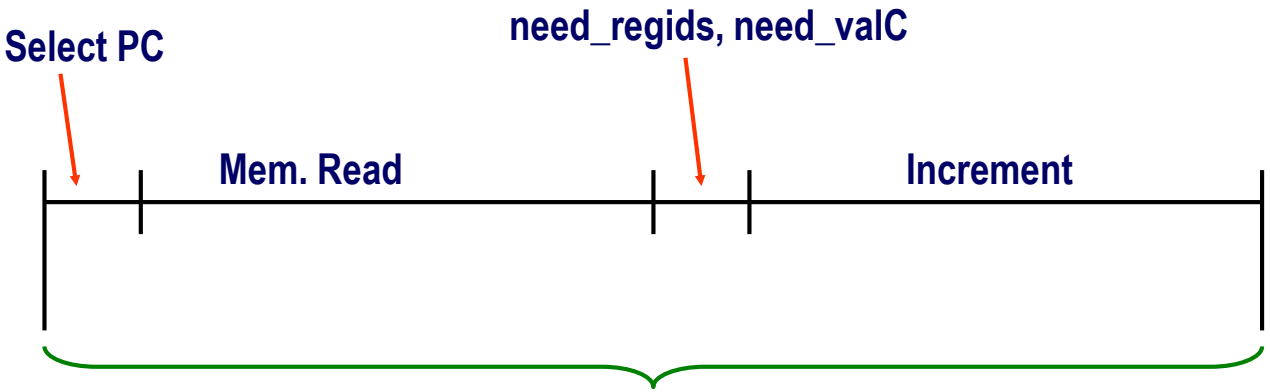
可能出题这里
注意ppt例题，期末考试题

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

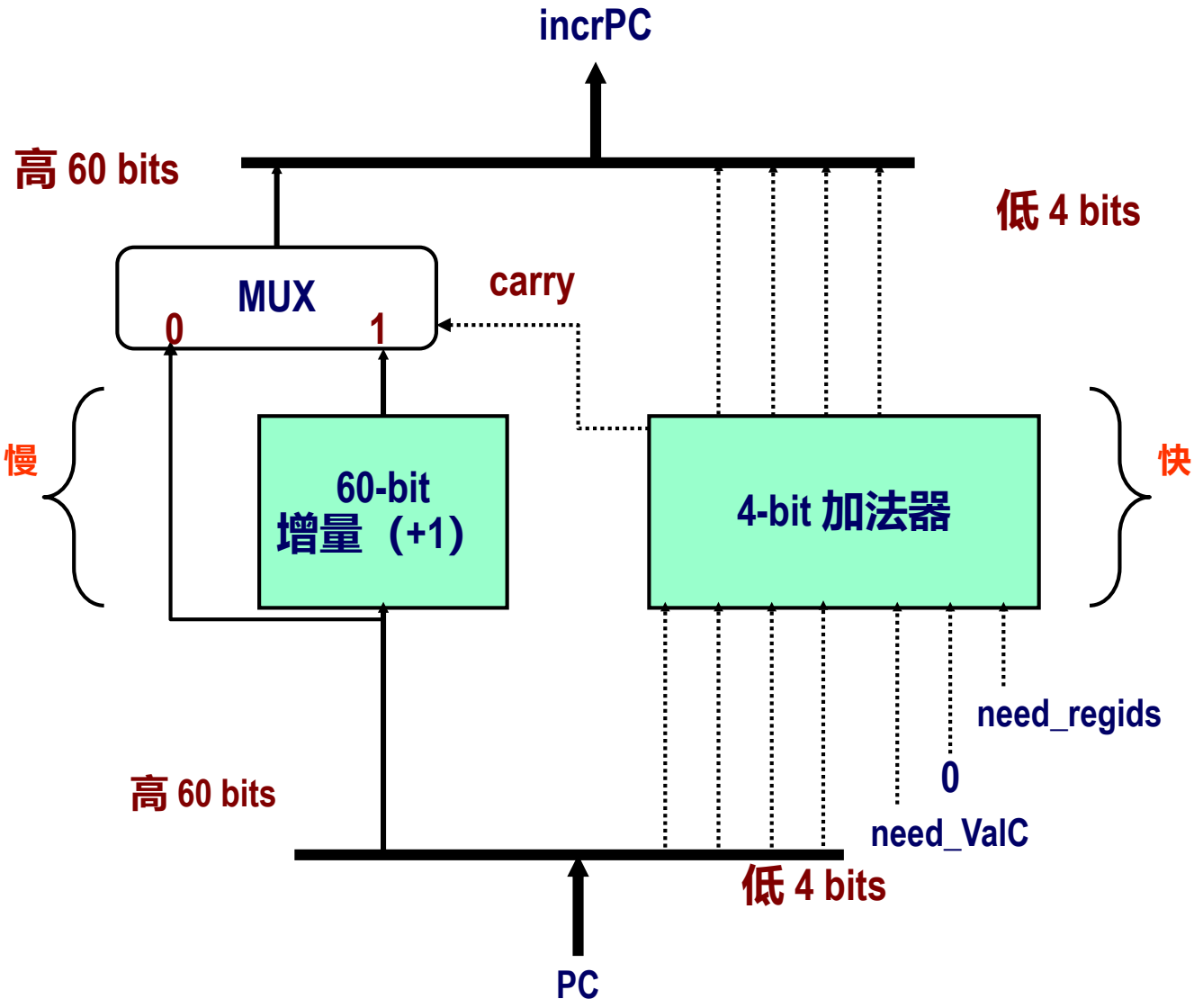


标准取指时序

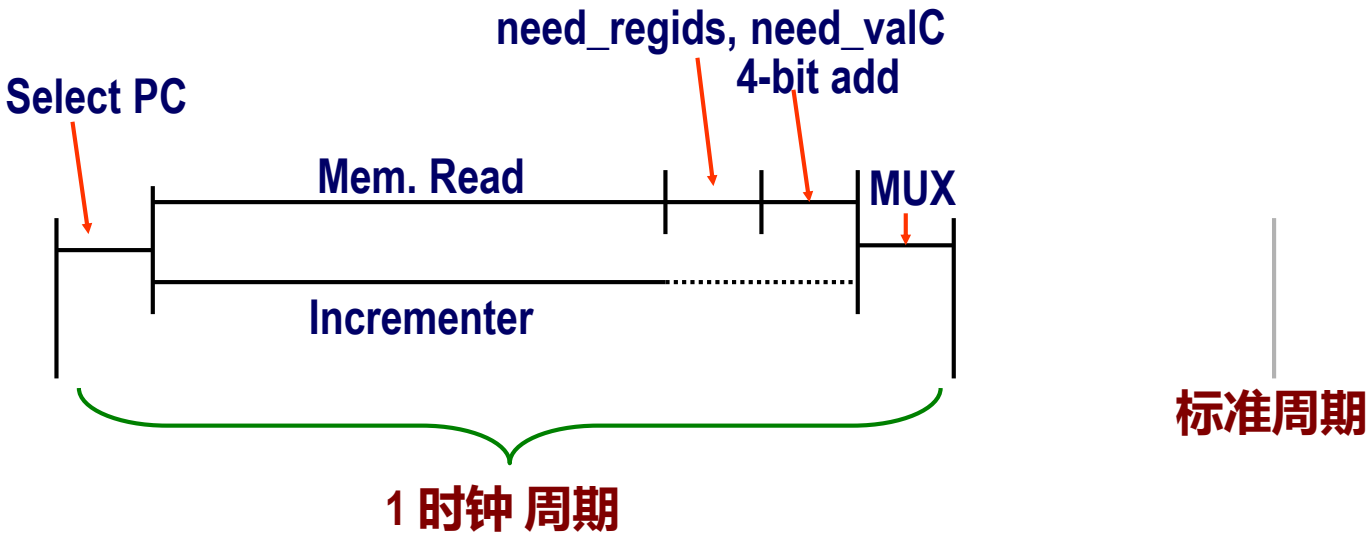


- 确保每一部分都顺序执行
- 在确定PC增加多少之后才能计算PC的值

快速增加PC的电路

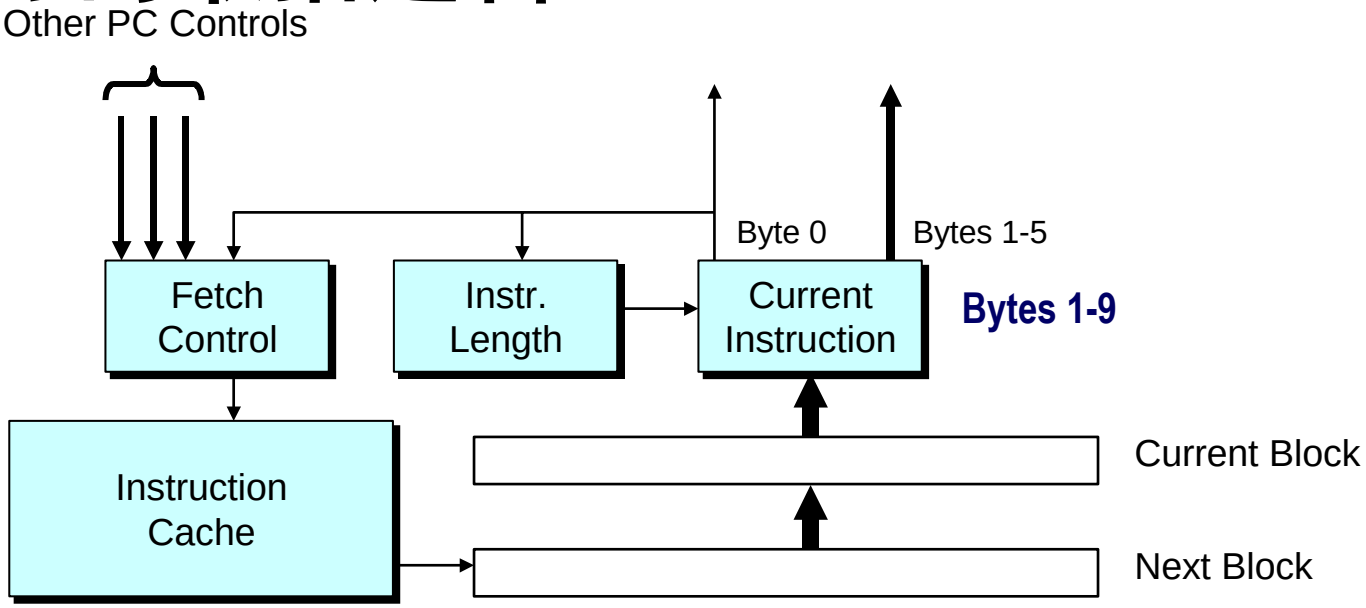

incrPC
高 60 bits
低 4 bits
MUX
0
1
carry
慢
60-bit
增量（+1）
4-bit 加法器
快
need_regids
0
need_ValC
高 60 bits
低 4 bits
PC

调整取指时间



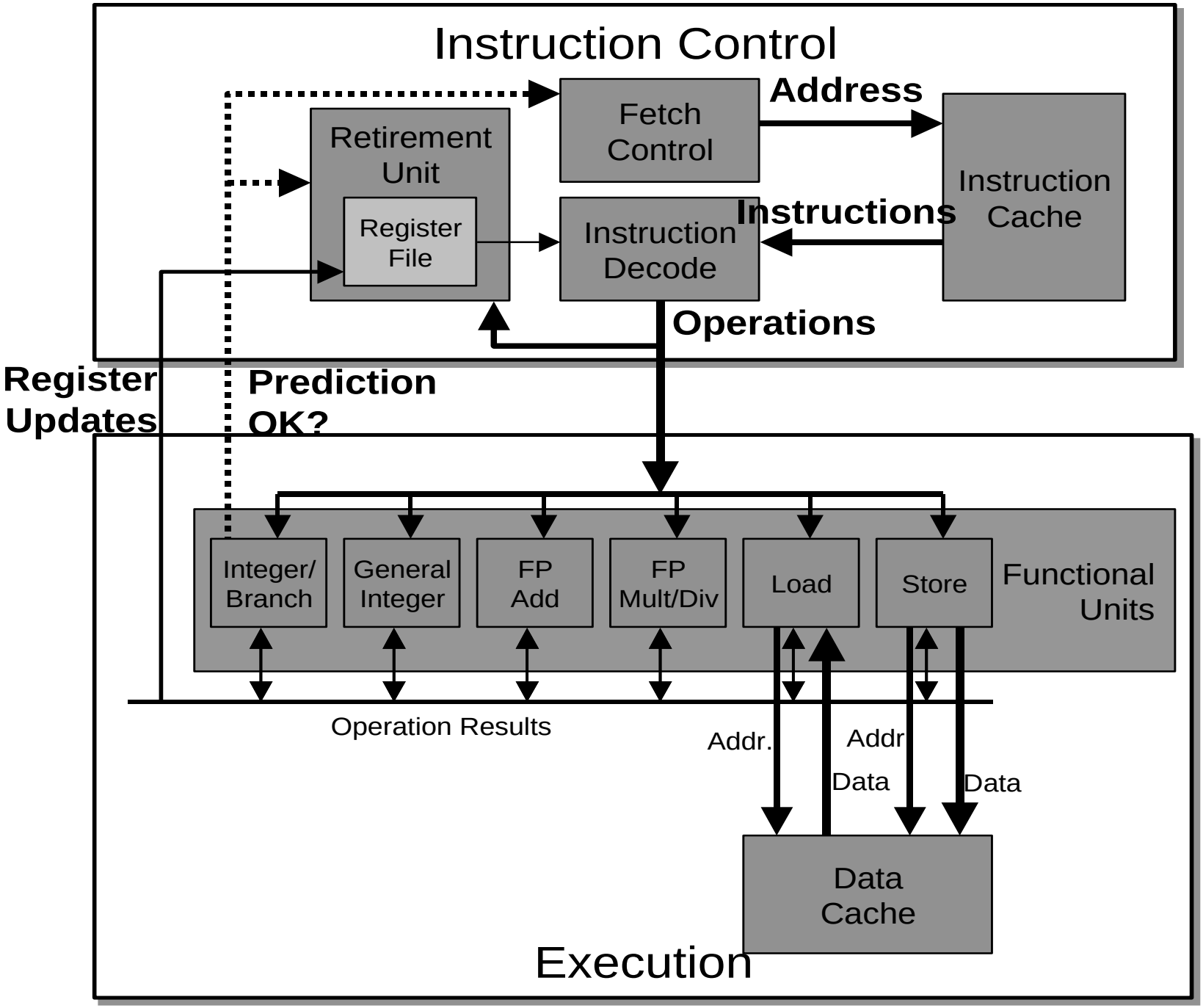
- **60-Bit 增量**
  - 当PC被选择时立即执行
  - 不必等到MUX再输出
  - 和存储器读并行执行

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

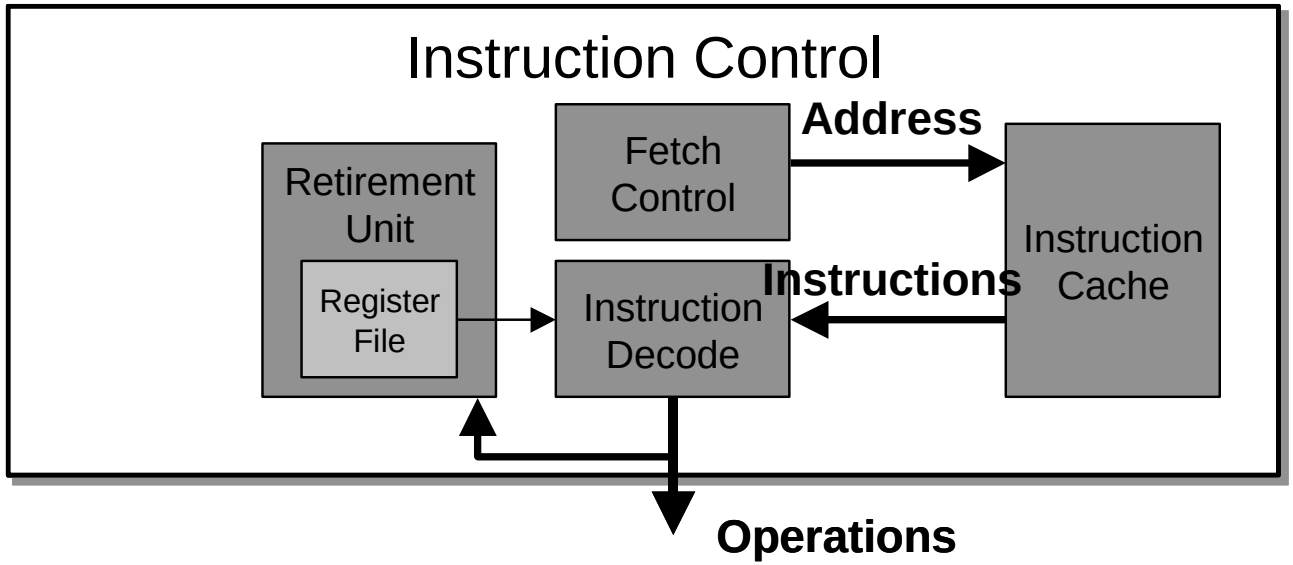
现代CPU设计


Instruction Control
Retirement Unit
Register File
Fetch Control
Address
Instruction Cache
Instructions
Instruction Decode
Operations
Register Updates
Prediction OK?
Integer/ Branch
General Integer
FP Add
FP Mult/Div
Load
Store
Functional Units
Operation Results
Addr.
Data
Addr
Data
Data Cache
Execution

在现代处理器架构中，**Retirement Unit（退休单元）**是流水线中的一个关键组件，负责处理指令的最终提交（commit）和异常处理。退休单元的主要作用是确保指令在完成所有必要的操作后，能够正确地更新程序状态，或者在发生异常时进行适当的处理。主要功能包括：

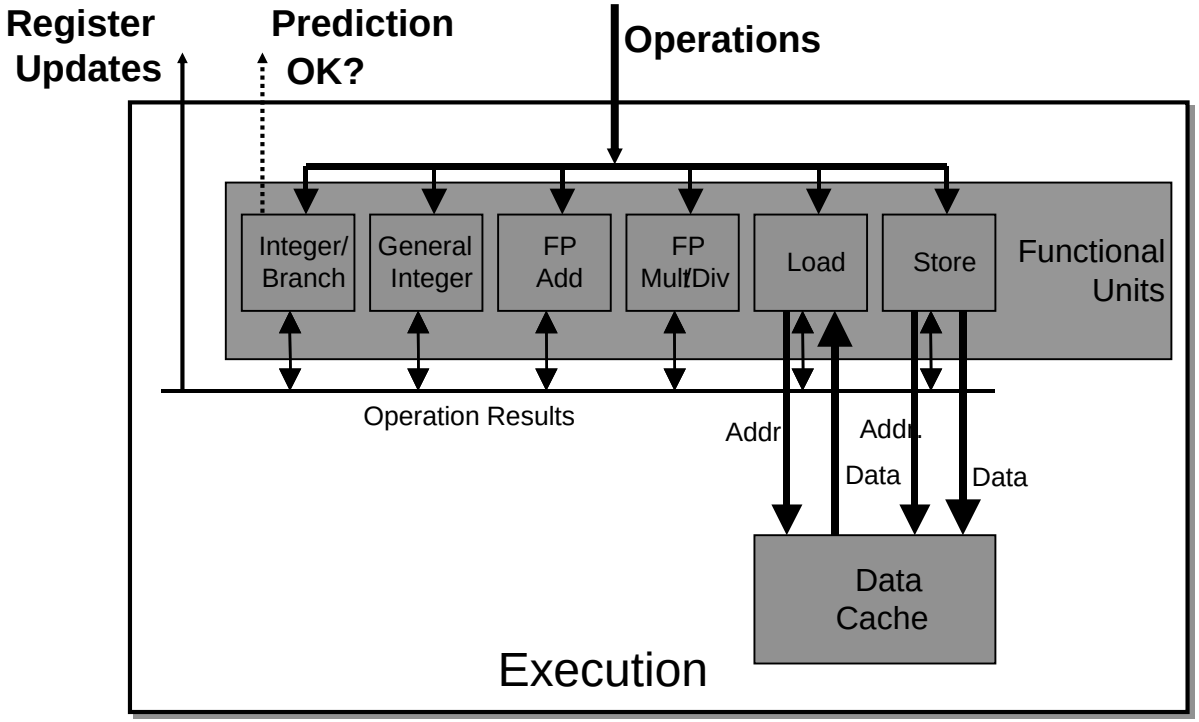
- 指令提交：当指令完成所有阶段（如执行、访存等）后，退休单元负责将指令的结果提交到程序状态中，更新寄存器和内存。
- 异常处理：如果指令在执行过程中触发了异常，退休单元会负责处理这些异常，包括保存当前状态、跳转到异常处理程序等。
- 指令顺序保证：确保指令按照程序的顺序正确提交，即使在乱序执行的处理器中，退休单元也会保证指令的提交顺序与程序顺序一致。

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**
  - 抽象的标识符将一个操作的目的和后一个操作的源相连接

执行单元

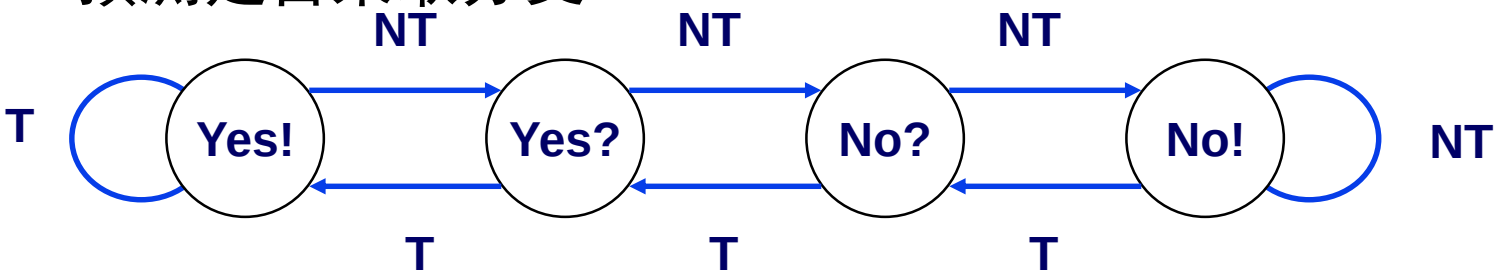


- 多功能单元
  - 每一个可以独立操作
- 一旦操作数就绪操作就可以执行
  - 不一定依据程序顺序执行
  - 受限于功能单元
- 控制逻辑
  - 确保执行结果和顺序执行一样

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

NT代表not true



- **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

4-6 习题

判断对错：

现代超标量 CPU 指令的平均周期通常小于 1 个时钟周期。（ 正确 ）

本章总结（分为三个阶段）

- **第一阶段**
  - 指令集体系结构
  - 逻辑设计
- **第二阶段**
  - 串行实现
  - 流水线和初级流水线实现
- **第三阶段**
  - 使流水线运行
  - 现代处理器设计

再次强调，微指令部分，要会默写。

Enjoy!