CS:APP Chapter 4 Computer Architecture

Wrap-Up
结语

任课教师：
宿红毅　张艳　黎有琦　颜珂

原作者：
Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

概述 Overview

流水线设计结语 Wrap-Up of PIPE Design

- 异常情况 Exceptional conditions
- 性能分析 Performance analysis
- 取指阶段设计 Fetch stage design

现代高性能处理器 Modern High-Performance Processors

- 乱序执行 Out-of-order execution

异常 Exceptions

- 在这种情景下处理器不能继续正常操作 Conditions under which processor cannot continue normal operation

原因 Causes

- 停机指令 Halt instruction (Current)
- 指令或数据地址不正确 Bad address for instruction or data (Previous)
- 不合法的指令 Invalid instruction (Previous)

典型期望的动作 Typical Desired Action

- 完成某些指令 Complete some instructions
  - 要么是当前要么是前面指令（取决于异常类型） Either current or previous (depends on exception type)
- 废弃其它指令 Discard others
- 调用异常处理程序 Call exception handler
  - 类似非预期的过程调用 Like an unexpected procedure call

我们的实现 Our Implementation

- 当指令引起异常时停机 Halt when instruction causes exception

异常示例 Exception Examples

在取指阶段检测到异常 Detect in Fetch Stage

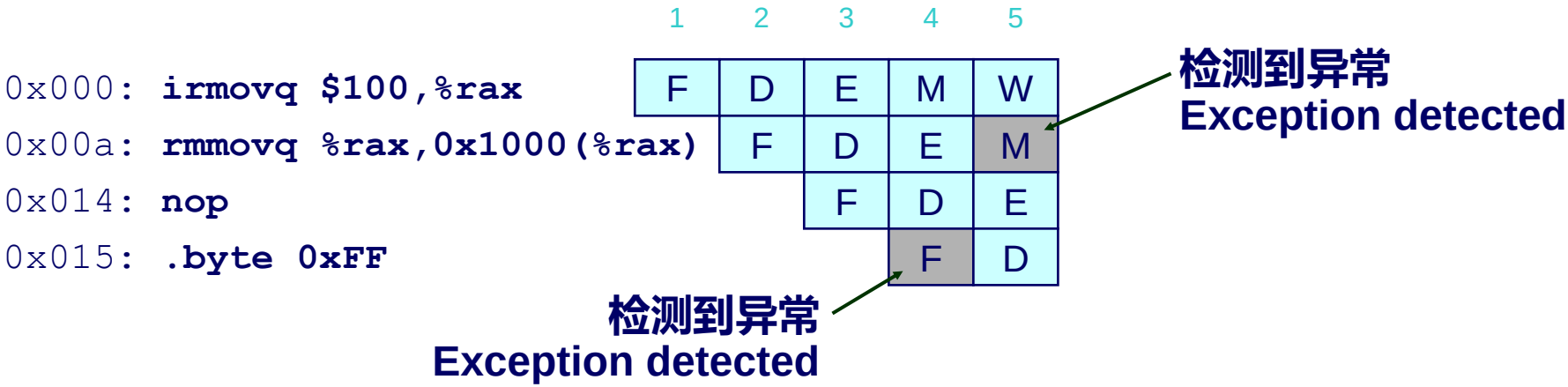
```
jmp $-1                    # Invalid jump target

.byte 0xFF                 # Invalid instruction code

halt                       # Halt instruction
```

在内存阶段检测到异常 Detect in Memory Stage

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
```

流水线处理器中的异常#1
Exceptions in Pipeline Processor #1

```
# demo-exc1.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # Invalid address
nop
.byte 0xFF                # Invalid instruction code
```



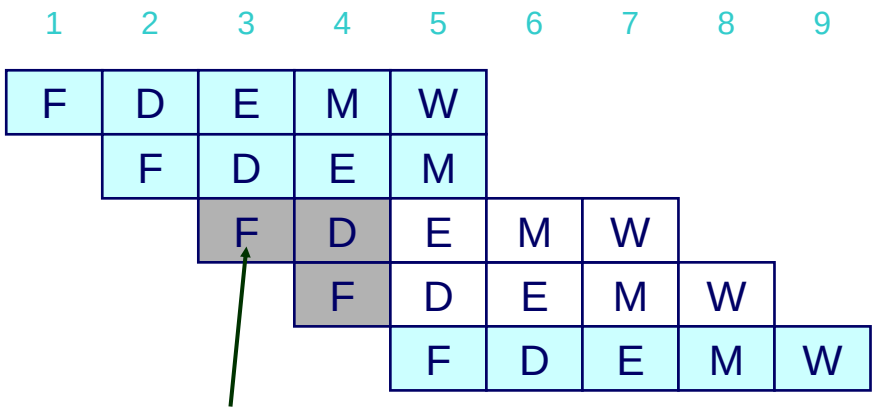
期望的行为 Desired Behavior

- rmmovq应该引起异常 rmmovq should cause exception
- 后续指令应该对处理器状态没有影响 Following instructions should have no effect on processor state

流水线处理器中的异常#2
Exceptions in Pipeline Processor #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
```

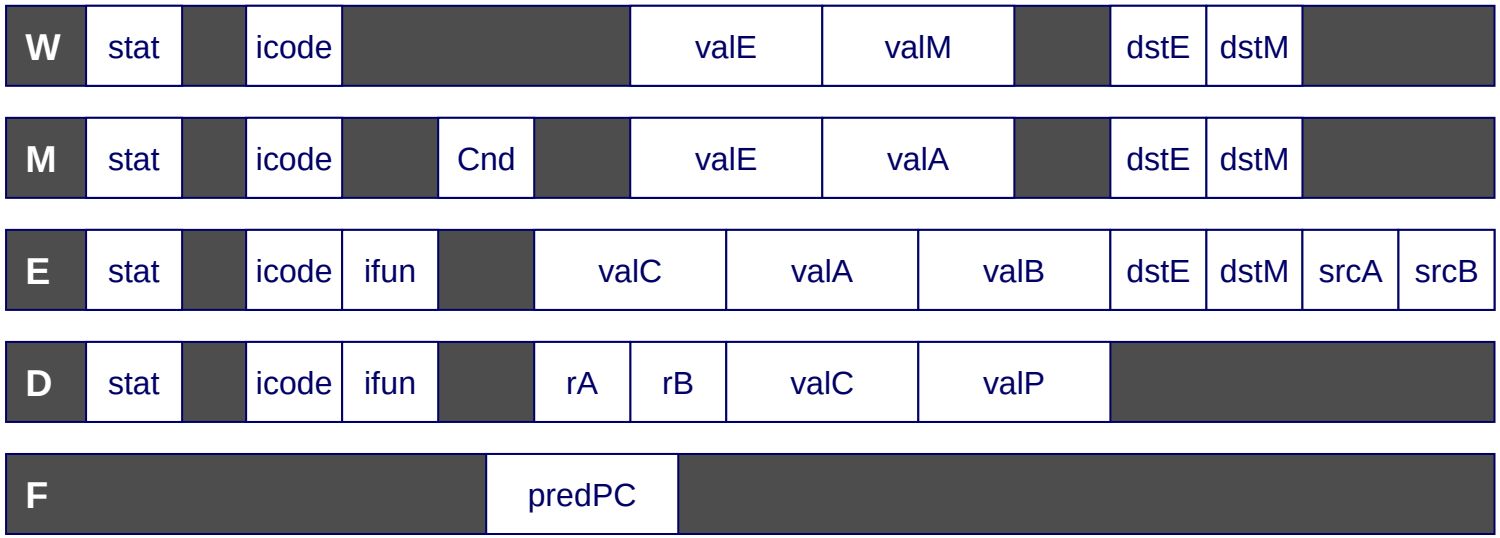


检测到异常 Exception detected

期望的行为 Desired Behavior

- 不应该发生异常 No exception should occur

维护异常排序 Maintaining Exception Ordering



- 流水线寄存器增加状态字段 Add status field to pipeline registers
- 取指阶段设置“AOK”、“ADR”（当取指地址错），“HLT”（停机指令）或“INS”（非法指令）状态之一 Fetch stage sets to either “AOK,” “ADR” (when bad fetch address), “HLT” (halt instruction) or “INS” (illegal instruction)
- 译码和执行阶段直接传递状态值 Decode & execute pass values through
- 内存阶段直接传递或设置成“ADR” Memory either passes through or sets to “ADR”
- 仅当指令到达写回阶段时触发异常 Exception triggered only

异常处理逻辑
Exception Handling Logic

取指阶段 Fetch Stage

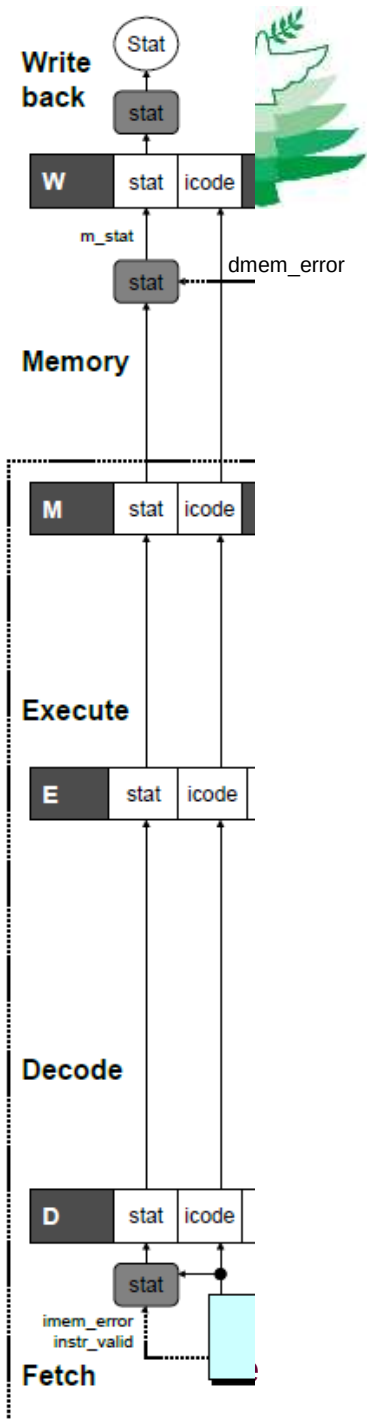
```
# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

内存阶段 Memory Stage

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

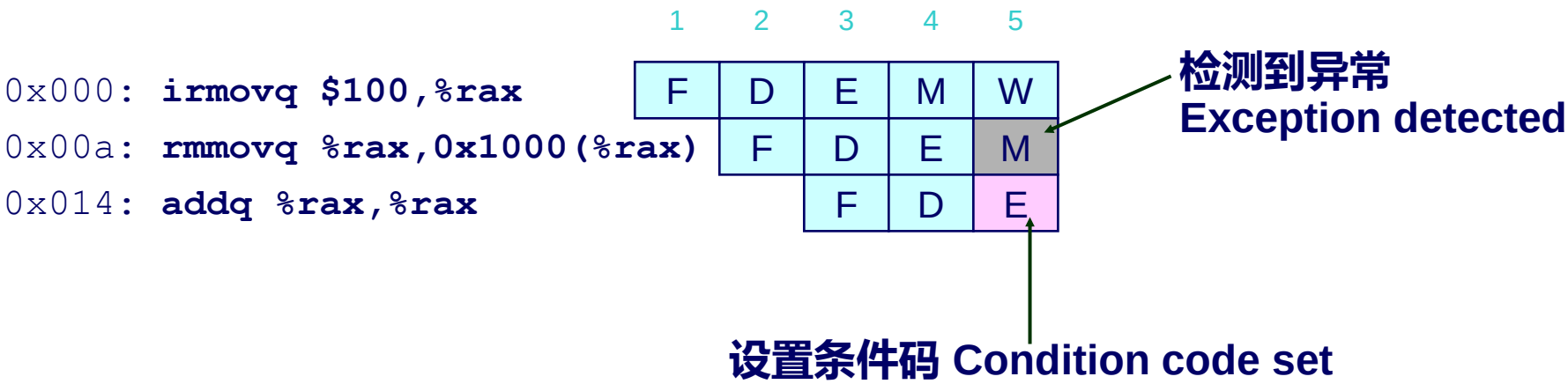
写回阶段 Writeback Stage

```
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
Side Effects in Pipeline Processor

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



期望的行为 Desired Behavior

- rmmovq应该引起异常 rmmovq should cause exception
- 后续指令应该不会有任何影响 No following instruction should have any effect

避免副作用 Avoiding Side Effects

出现异常应该取消状态更新 Presence of Exception Should Disable State Update

- 不合法指令转换成流水线气泡 Invalid instructions are converted to pipeline bubbles
  - 除非有状态指明异常状态 Except have stat indicating exception status
- 数据内存不会写到非法地址 Data memory will not write to invalid address
- 防止非法更新条件码 Prevent invalid update of condition codes
  - 在内存阶段检测异常 Detect exception in memory stage
  - 在执行阶段取消条件码 Disable condition code setting in execute
  - 必须在同一个时钟周期发生 Must happen in same clock cycle
- 在最后阶段处理异常 Handling exception in final stages
  - 当在内存阶段检测到异常 When detect exception in memory stage
    - » 下一个周期开始注入气泡到内存阶段 Start injecting bubbles into memory stage on next cycle
  - 当在写回阶段检测到异常 When detect exception in write-back stage
    - » 暂停异常指令 Stall excepting instruction
- 包括在HCL代码中 Included in HCL code

状态改变的控制逻辑
Control Logic for State Changes

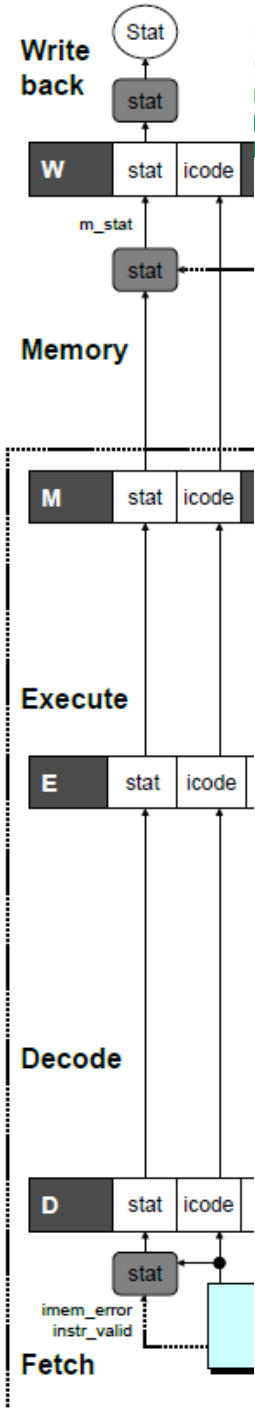
设置条件码 Setting Condition Codes

```
# 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 Stage Control

- 也控制内存更新 Also controls updating of memory

```
# 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 };
```



现实生活中的其他异常处理 Rest of Real-Life Exception Handling

调用异常处理程序 Call Exception Handler

- PC压入栈 Push PC onto stack
  - 故障指令或下一条指令的PC Either PC of faulting instruction or of next instruction
  - 通常随着异常状态直接在流水线传递 Usually pass through pipeline along with exception status
- 跳转到处理程序地址 Jump to handler address
  - 通常是固定地址 Usually fixed address
  - 定义为ISA的一部分 Defined as part of ISA

实现 Implementation

- 还没有实现 Haven't tried it yet!

性能度量 Performance Metrics

时钟频率 Clock rate

- 用GHz度量 Measured in Gigahertz
- 阶段的功能划分和电路设计 Function of stage partitioning and circuit design
  - 保持每个阶段工作量较小 Keep amount of work per stage small

指令执行的速率 Rate at which instructions executed

- CPI：每条指令占用的周期数 CPI: cycles per instruction
- 平均每条指令需要多少时钟周期 On average, how many clock cycles does each instruction require?
- 流水线功能设计和基准测试程序 Function of pipeline design and benchmark programs
  - 例如分支预测错误的频率 E.g., how frequently are branches mispredicted?

流水线处理器的CPI CPI for PIPE

CPI ≈ 1.0

- 每个时钟周期取一条指令 Fetch instruction each clock cycle
- 几乎每个周期有效处理一条新指令 Effectively process new instruction almost every cycle
  - 尽管每条单独指令都有5个周期的时延 Although each individual instruction has latency of 5 cycles

CPI > 1.0

- 有时必须暂停或取消分支 Sometimes must stall or cancel branches

计算CPI Computing CPI

- C是时钟周期数 C clock cycles
- I是执行完成的指令数 I instructions executed to completion
- B是注入的气泡气泡数 B bubbles injected (C = I + B)

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

- 因子B/I表示气泡的平均惩罚 Factor B/I represents average penalty due to bubbles

流水线处理器的CPI CPI for PIPE (Cont.)

B/I = LP + MP + RP

典型值 Typical Values

- LP是由于装载/使用冒险暂停导致的惩罚 LP: Penalty due to load/use hazard stalling
  - Load指令的占比 Fraction of instructions that are loads 0.25
  - Load指令需要暂停的比例 Fraction of load instructions requiring stall 0.20
  - 每次注入的气泡数 Number of bubbles injected each time 1

  ⇒ LP = 0.25 * 0.20 * 1 = 0.05
- MP是由于分支预测错误导致的惩罚 MP: Penalty due to mispredicted branches
  - 条件跳转指令的占比 Fraction of instructions that are cond. jumps 0.20
  - 条件跳转预测错误的比例 Fraction of cond. jumps mispredicted 0.40
  - 每次注入的气泡数 Number of bubbles injected each time 2

  ⇒ MP = 0.20 * 0.40 * 2 = 0.16

流水线处理器的CPI CPI for PIPE (Cont.)

B/I = LP + MP + RP

典型值 Typical Values

- RP是返回指令导致的惩罚 RP: Penalty due to `ret` instructions
  - 返回指令的占比 Fraction of instructions that are returns 0.02
  - 每次注入的气泡数 Number of bubbles injected each time 3

  ⇒ RP = 0.02 * 3 = 0.06
- 惩罚的净效果 Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27

  ⇒ CPI = 1.27 (还不赖 Not bad!)

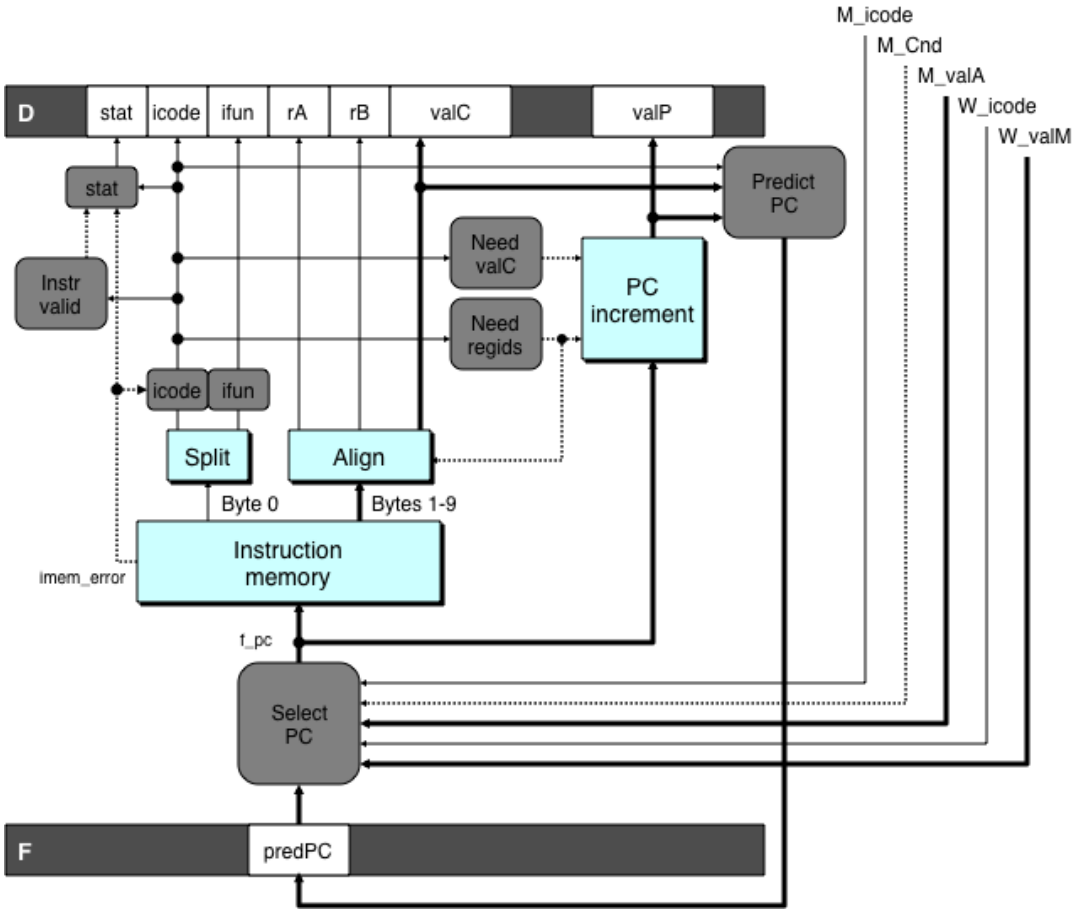
取指逻辑修正 Fetch Logic Revisited

在取指周期期间 During Fetch Cycle

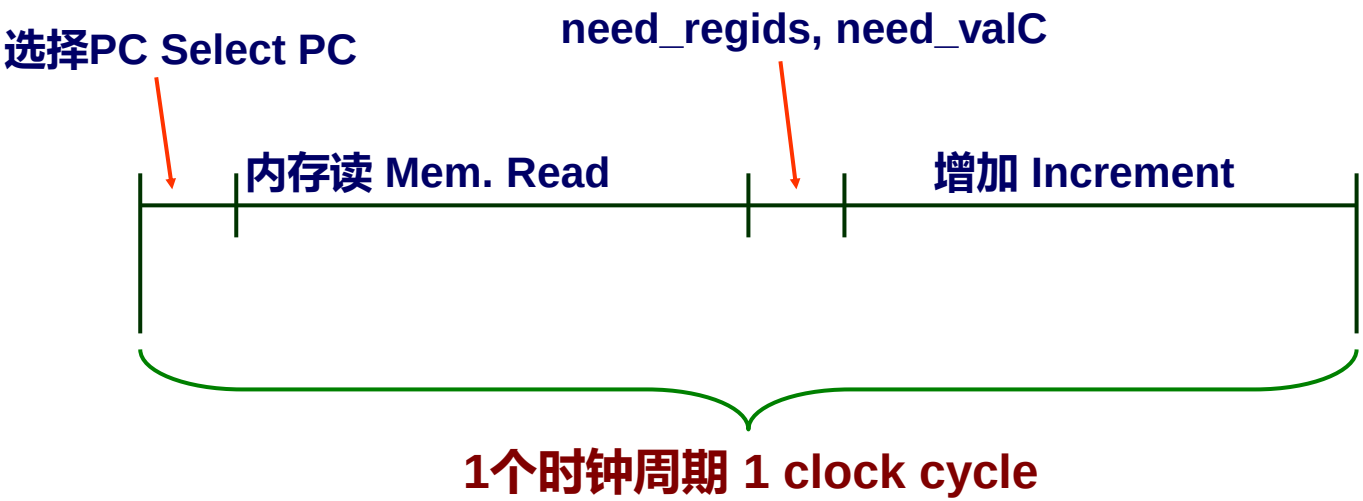
1. 选择PC Select PC
2. 从指令内存读字节 Read bytes from instruction memory
3. 检查icode确定指令长度 Examine icode to determine instruction length
4. 增加PC Increment PC

时序 Timing

- 步骤2和4需要较长的时间 Steps 2 & 4 require significant amount of time

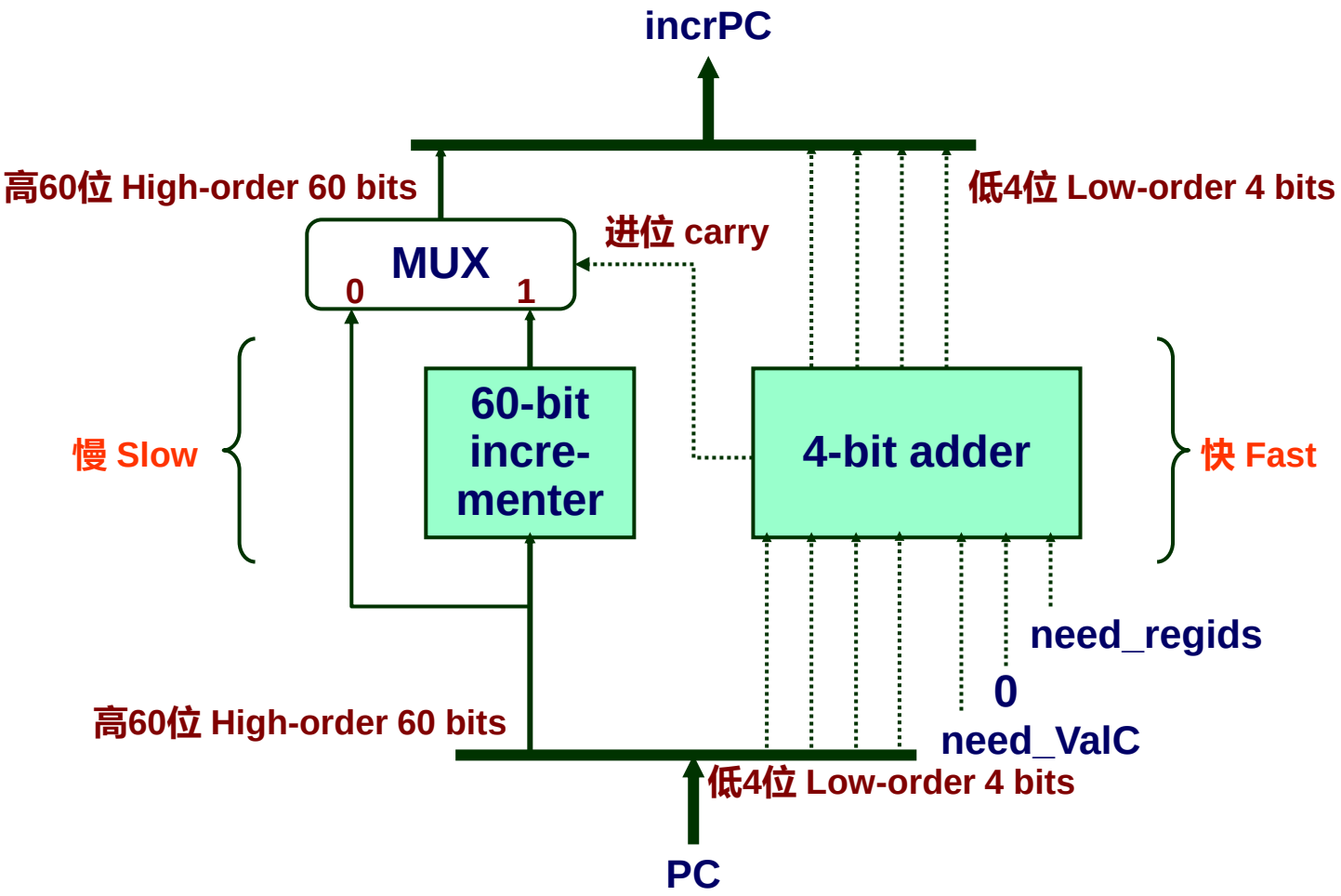


标准取指时序 Standard Fetch Timing

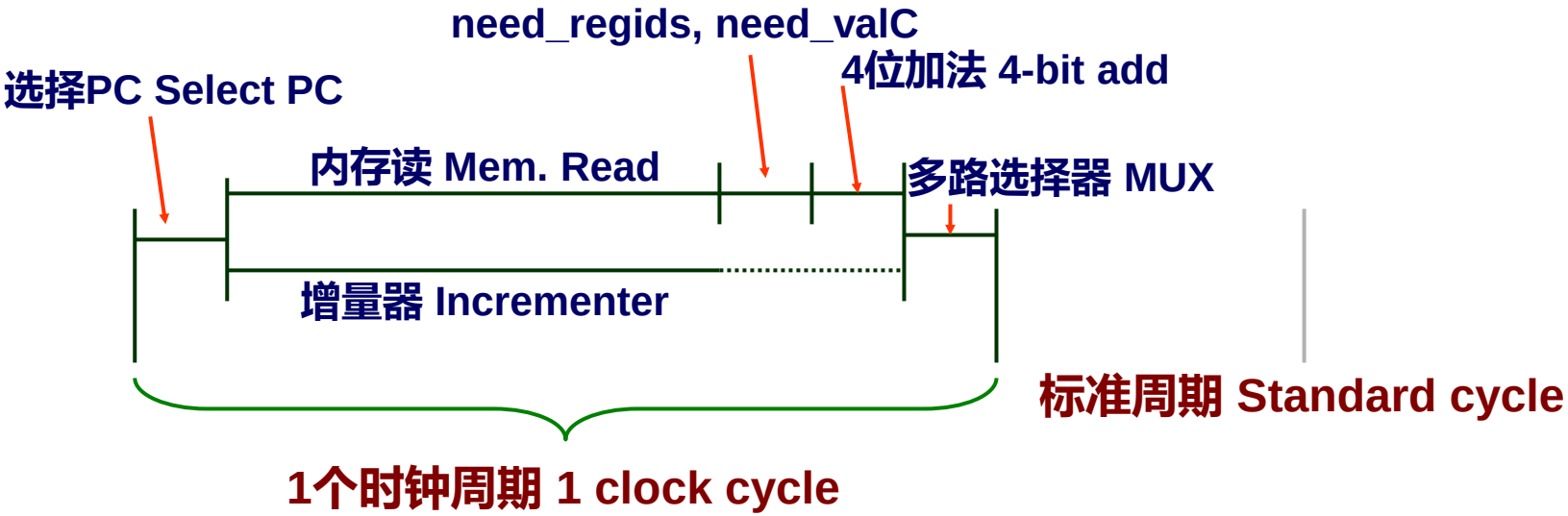


- 必须顺序执行每个操作 Must Perform Everything in Sequence
- 在知道PC需要增加多少前无法计算PC增加 Can't compute incremented PC until know how much to increment it by

快速的PC增加电路
A Fast PC Increment Circuit



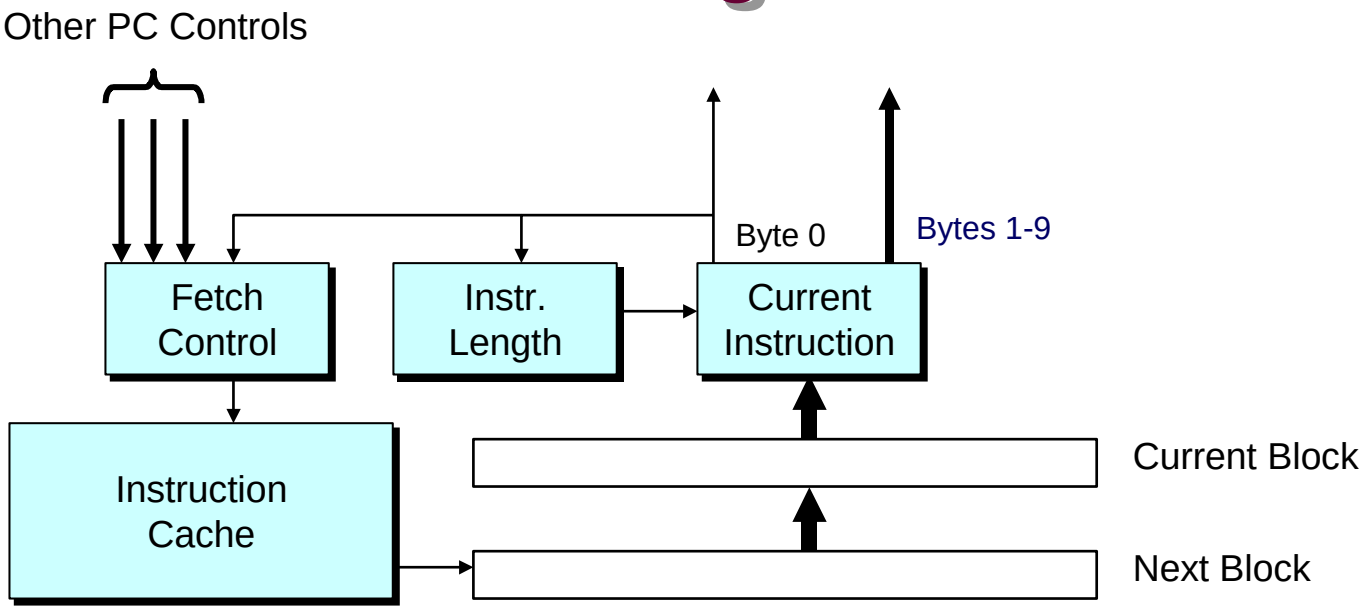
修改的取指时序 Modified Fetch Timing



60位的增量器 60-Bit Incrementer

- 只要选择PC后，立即开始行动 Acts as soon as PC selected
- 直到最后多路选择器才需要输出 Output not needed until final MUX
- 与内存读并行工作 Works in parallel with memory read

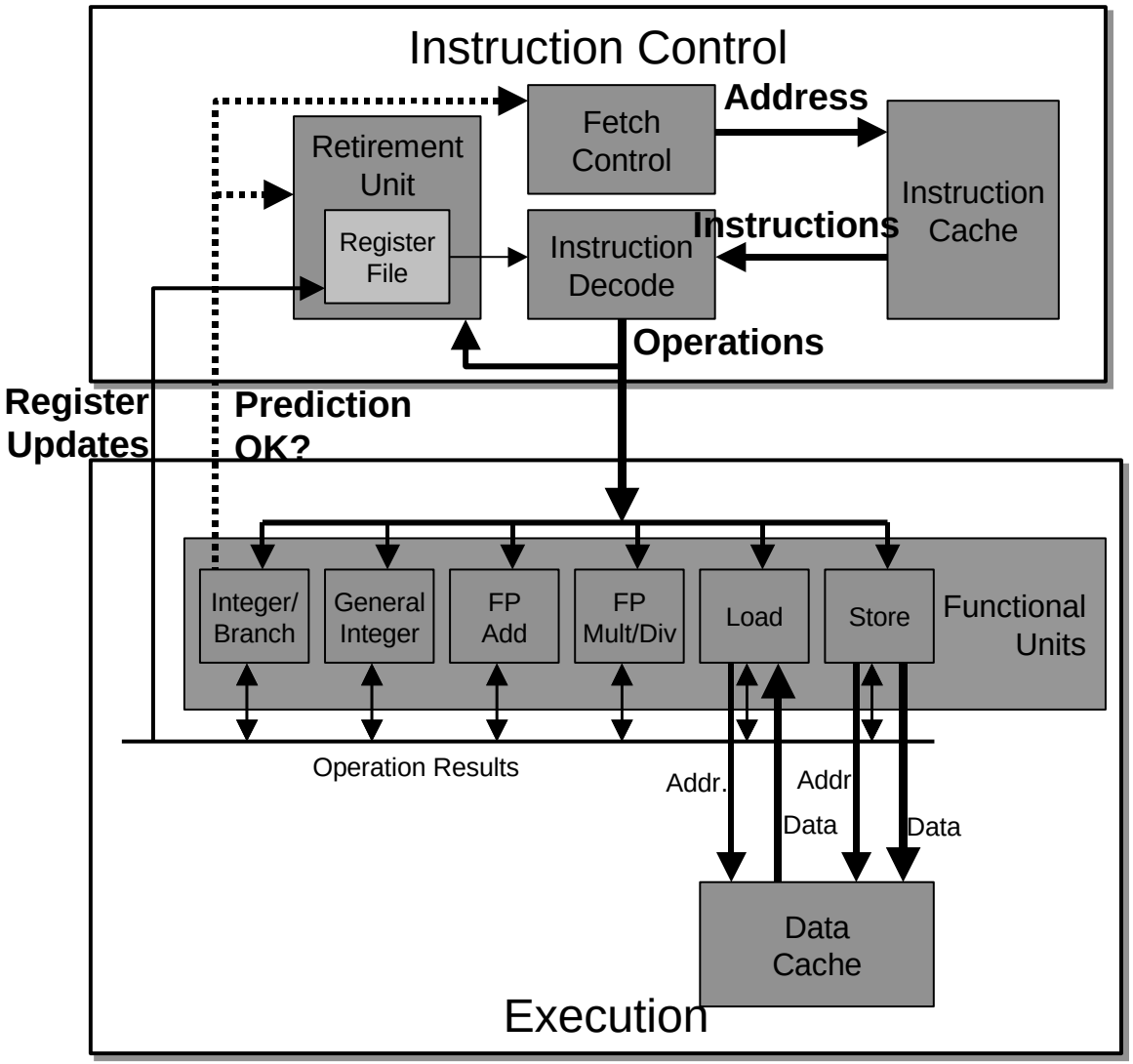
更真实的取指逻辑
More Realistic Fetch Logic



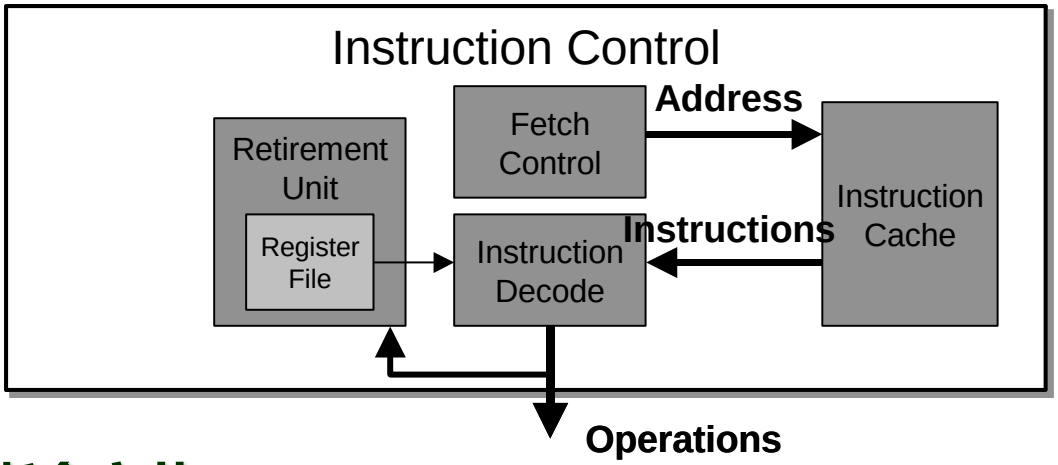
取指框 Fetch Box

- 集成进指令高速缓存 Integrated into instruction cache
- 取整个cache块（16或32字节） Fetches entire cache block (16 or 32 bytes)
- 从当前块选择当前指令 Selects current instruction from current block
- 提前工作取下一个块 Works ahead to fetch next block
  - 当到达当前块的尾部 As reaches end of current block
  - 在分支目标处 At branch target

现代CPU设计 Modern CPU Design



指令控制 Instruction Control



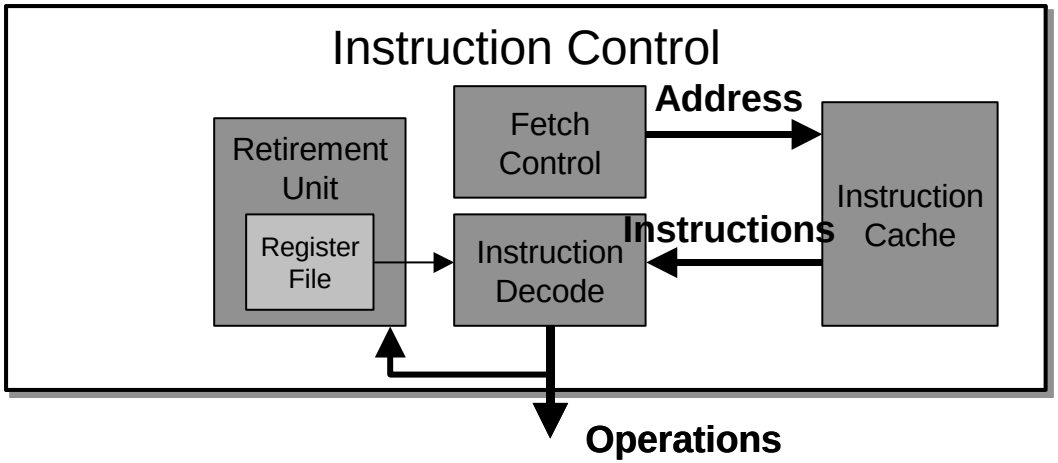
从内存抓取指令字节 Grabs Instruction Bytes From Memory

- 基于当前PC+分支预测的预测目标 Based on Current PC + Predicted Targets for Predicted Branches
- 硬件动态猜测是否选择/不选择分支和（可能的）分支目标 Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

翻译指令成操作 Translates Instructions Into *Operations*

- 执行指令所需要的原语步骤 Primitive steps required to perform instruction
- 典型指令需要1-3个操作 Typical instruction requires 1–3 operations

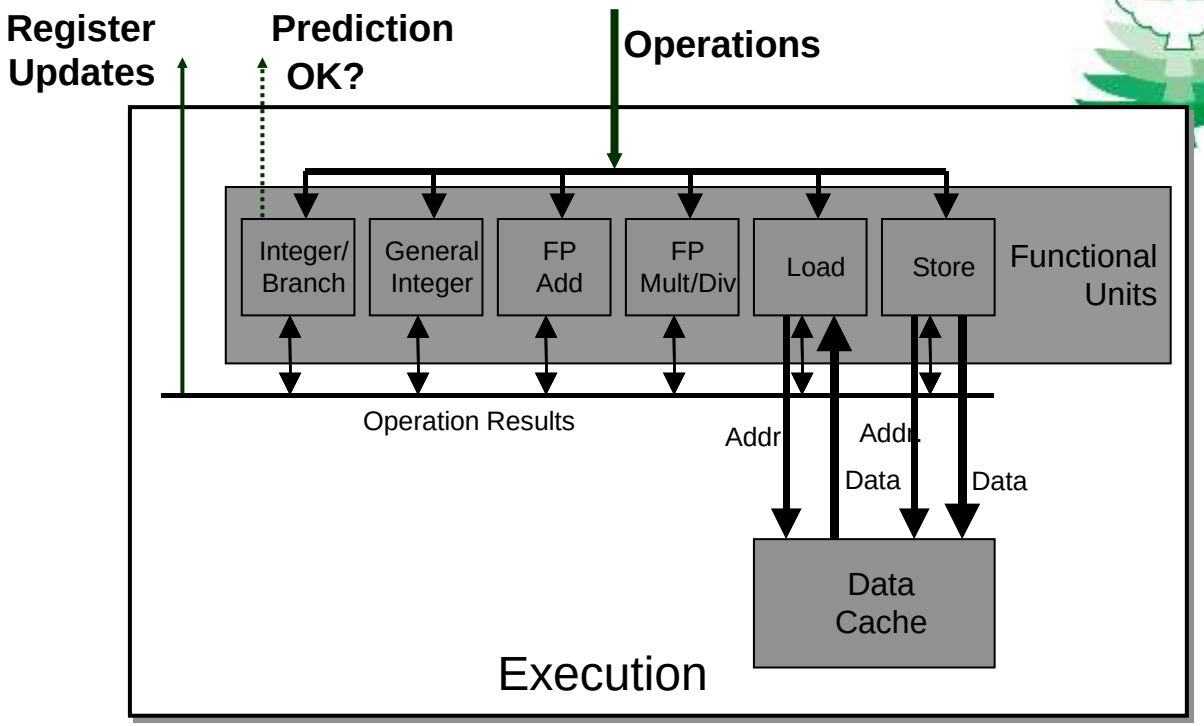
指令控制 Instruction Control



转换寄存器引用成标记 Converts Register References Into *Tags*

- 将一个操作的目的地址和后续操作的源地址链接在一起的抽象标识符
  Abstract identifier linking destination of one operation with sources of later operations

执行单元 Execution Unit



- 多功能单元 Multiple functional units
  - 每个单元可以独立地运行 Each can operate in independently
- 一旦操作数可用，就立即执行操作 Operations performed as soon as operands available
  - 没必要按照程序的顺序 Not necessarily in program order
  - 在功能单元范围内 Within limits of functional units
- 控制逻辑 Control logic
  - 确保行为等价于顺序程序执行 Ensures behavior equivalent to sequential program execution

Intel Haswell的CPU能力
CPU Capabilities of Intel Haswell

多条指令可以并行执行 Multiple Instructions Can Execute in Parallel

- 2条load指令 2 load
- 1条store指令 1 store
- 4条整数指令 4 integer
- 2条浮点乘法 2 FP multiply
- 1条浮点加/除 1 FP add / divide

有些指令占用1个以上周期，但是可以流水线化 Some Instructions Take > 1 Cycle, but Can be Pipelined

指令 Instruction	时延 Latency	周期/发射 Cycles/Issue
装载/存储 Load / Store	4	1
整数乘 Integer Multiply	3	1
整数除 Integer Divide	3—30	3—30
双/单精度浮点乘 Double/Single FP Multiply	5	1
双/单精度浮点加 Double/Single FP Add	3	1
双/单精度浮点除 Double/Single FP Divide	10—15	6—11

Haswell操作 Haswell Operation

- 动态翻译指令成“Uops” Translates instructions dynamically into “Uops”
  - 超过118位宽 ~118 bits wide
  - 包括操作、两个源地址和目的地址 Holds operation, two sources, and destination
- 执行Uops用“乱序”引擎 Executes Uops with “Out of Order” engine
  - Uop执行，当以下时候 Uop executed when
    - 操作数可用 Operands available
    - 功能单元可用 Functional unit available
  - 由“预留站”控制执行 Execution controlled by “Reservation Stations”
    - 保持跟踪uops之间的数据相关 Keeps track of data dependencies between uops
    - 分配资源 Allocates resources

高性能分支预测
High-Perforamnce Branch Prediction

对性能至关重要 Critical to Performance

- 典型地预测错误需要11-15周期惩罚 Typically 11–15 cycle penalty for misprediction

分支目标缓冲 Branch Target Buffer

- 512个条目 512 entries
- 4位历史 4 bits of history
- 自适应算法 Adaptive algorithm
  - 能够识别重复的模式，例如替换选择-不选择 Can recognize repeated patterns, e.g., alternating taken–not taken

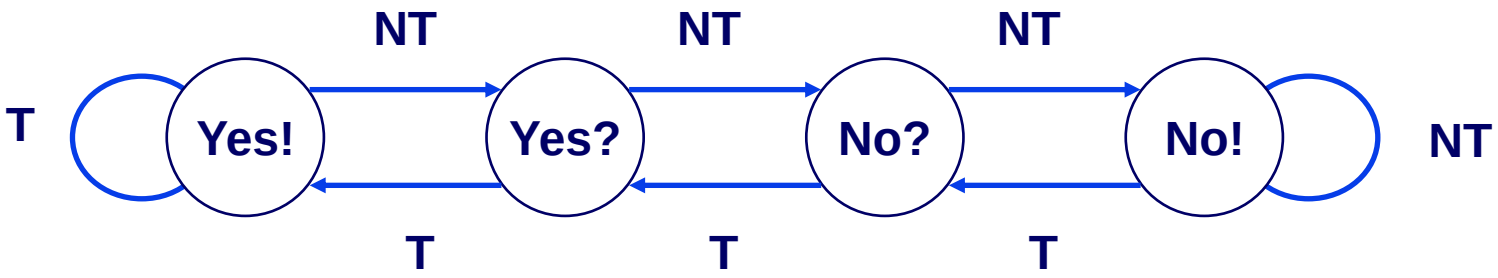
处理BTB缺失 Handling BTB misses

- 检测大约第6个周期 Detect in ~cycle 6
- 对负偏移预测选择，对正偏移预测不选择 Predict taken for negative offset, not taken for positive
  - 循环对条件分支 Loops vs. conditionals

示例分支预测 Example Branch Prediction

分支历史 Branch History

- 对有关以前分支指令的历史进行信息编码 Encode information about prior history of branch instructions
- 预测分支是否选择 Predict whether or not branch will be taken



状态机 State Machine

- 每次分支选择，向右转换 Each time branch taken, transition to right
- 当不选择，向左转换 When not taken, transition to left
- 预测分支选择当处于状态Yes！或Yes？ Predict branch taken when in state Yes! or Yes?

处理器小结 Processor Summary

设计技术 Design Technique

- 为所有指令创建统一的框架 Create uniform framework for all instructions
  - 想要在指令之间共享硬件 Want to share hardware among instructions
- 用控制逻辑位连接标准逻辑块 Connect standard logic blocks with bits of control logic

操作 Operation

- 状态存储在内存和时序寄存器中 State held in memories and clocked registers
- 由组合逻辑完成计算 Computation done by combinational logic
- 寄存器/内存时钟足以控制总体行为 Clocking of registers/memories sufficient to control overall behavior

处理器小结 Processor Summary

增强性能 Enhancing Performance

- 流水线增加了吞吐量和改进了资源利用率 Pipelining increases throughput and improves resource utilization
- 必须确保维持ISA行为 Must make sure to maintain ISA behavior