LoongArch CPU 设计实战

第2天：流水线设计简介

教师：杨建磊、万 寒

助教：苏阳、周振源

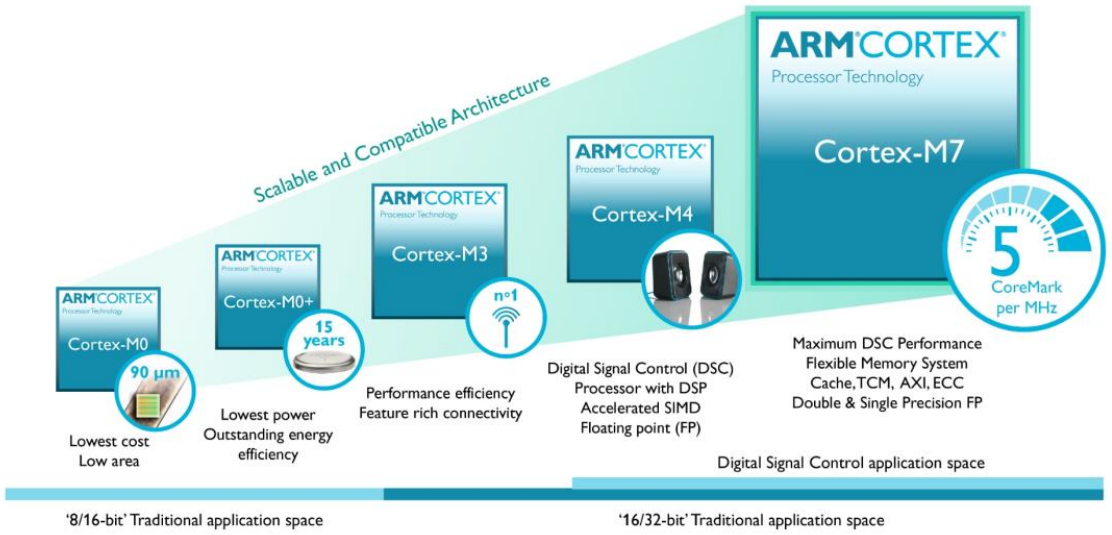
北京航空航天大学 计算机学院

2025年7月29日 四川 成都

Day2 目录

1.1 处理器的性能

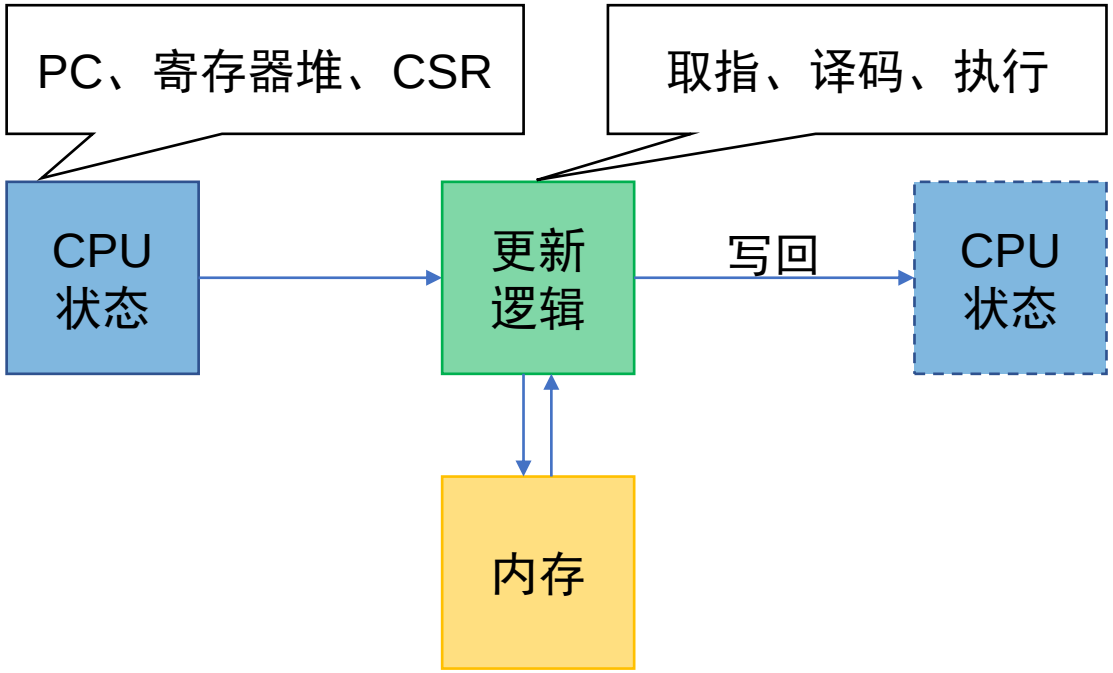
- 吞吐率
  - 单位时间内流水线能处理的任务数量
- IPC
  - 每时钟周期处理器能处理完成的指令数
- DMIPS/CoreMark
  - 单位时间内CoreMark/Dhrystone程序的运行次数，单位是CoreMark/MHz



Core	DMIPS/MHz	CoreMark/MHz
Cortex-M0	0.84	2.33
Cortex-M0+	0.94	2.42
Cortex-M3	1.25	3.32
Cortex-M4	1.25	3.40
Cortex-M7	2.14	5.04

1.2 指令的流水执行

□ 单周期处理器



(・ω・)

减肥

(・ω・)

硬件频率下降
行动迟缓
TT～～TT

中间逻辑
过于肥硕
~大胃袋这一块~

1.2 指令的流水执行

- 处理器设计是一个工程问题
  - 多目标优化 ==> 所谓“众口难调”

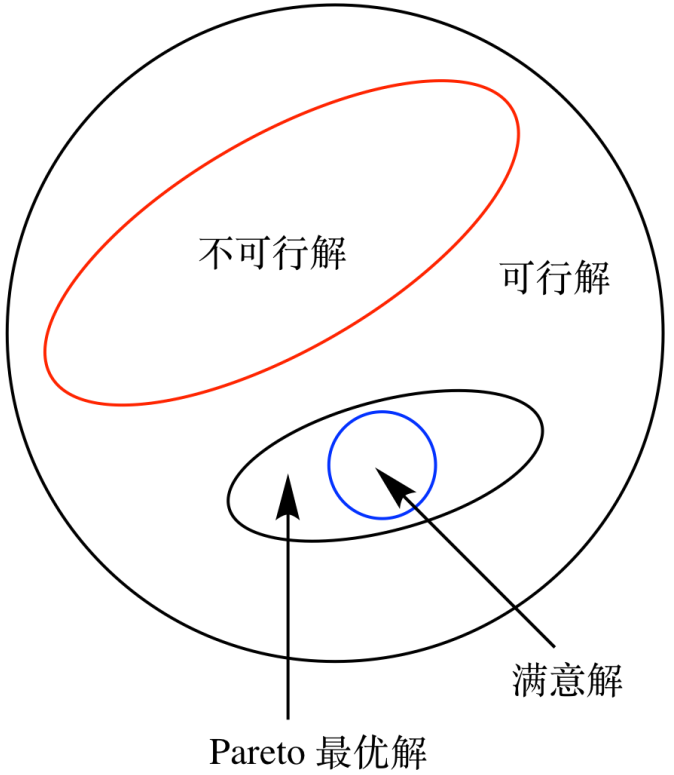
⇩

优化一个目标往往是以牺牲其他目标为代价的

⇩

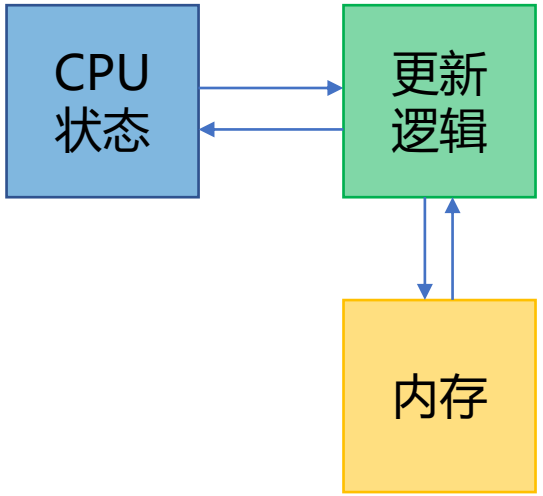
- 大部分工程问题不存在最优解{{{(>_<)}}}

- 复杂/简单的处理器架构无绝对意义的高下之分



1.2 指令的流水执行

- 在硬件上什么样的设计是合适的?
  - 为什么单周期是不合适的?
    - 更新逻辑复杂 ==> 拖累硬件的**时钟频率**
    - 硬件资源浪费 ==> 访存指令ALU，完成计算后闲置
  - 软件的特点 ==> 运行速度 = IPC * 频率、串行
  - 硬件的特点 ==> 运行速度与链路长度正相关、并发
    - 减少每周期组合逻辑规模
    - 探索指令并发的可能性



Tip：不难看出**单周期**的架构设计非常适合构建**软件模拟器**

流水线技术

那么代价是什么呢(⊙_⊙)?

优化一个目标往往是以**牺牲其他目标**为代价的

1.2 指令的流水执行

□ 流水线技术的代价

□逻辑切分的代价

很大程度上依靠经验

⇩

- □ 切哪里？怎么切？ ==> 直接决定流水线架构
- □ 流水线寄存器导致额外的面积开销
- □ 串行的软件编程思想 ==> 并行的硬件编程思想

□指令并行的代价

- □ 数据/状态同步问题 ==> 所有并行处理模型都必须解决的问题

↳ 处理器中一般称之为**冒险**

如何尽量减小冒险风险？
以及
如何从冒险失败中恢复？

仍是值得讨论和研究的问题

- □ 包括但不限于：指令间数据依赖导致的数据同步、分支/特权指令导致的状态同步

设计复杂度↑ (ﾟ - ﾟ)↑ ⇦ **优化一个目标**往往是以**牺牲其他目标**为代价的

1.2 指令的流水执行

□ 流水线技术的代价

我们将主要围绕这两个问题展开

□逻辑切分的代价

很大程度上依靠经验

- □ 切哪里？怎么切？ ==> 直接决定流水线架构
- □ 流水线寄存器导致额外的面积开销
- □ 串行的软件编程思想 ==> 并行的硬件编程思想

□指令并行的代价

- □ 数据/状态同步问题 ==> 所有并行处理模型都必须解决的问题

处理器中一般称之为**冒险**

- □ 包括但不限于：指令间数据依赖导致的数据同步、分支/特权指令导致的状态同步

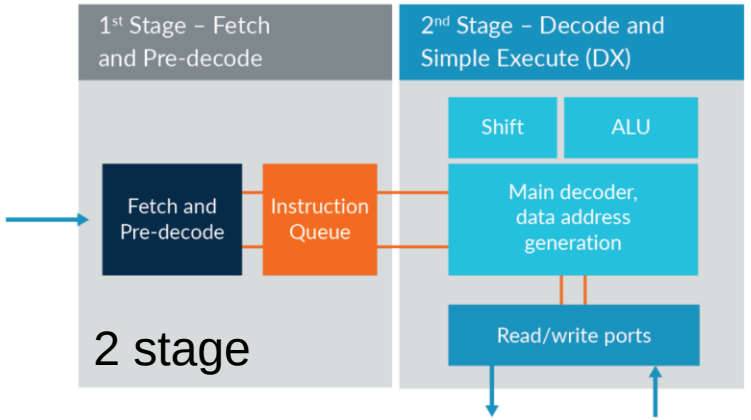
如何尽量减小冒险风险？
以及
如何从冒险失败中恢复？

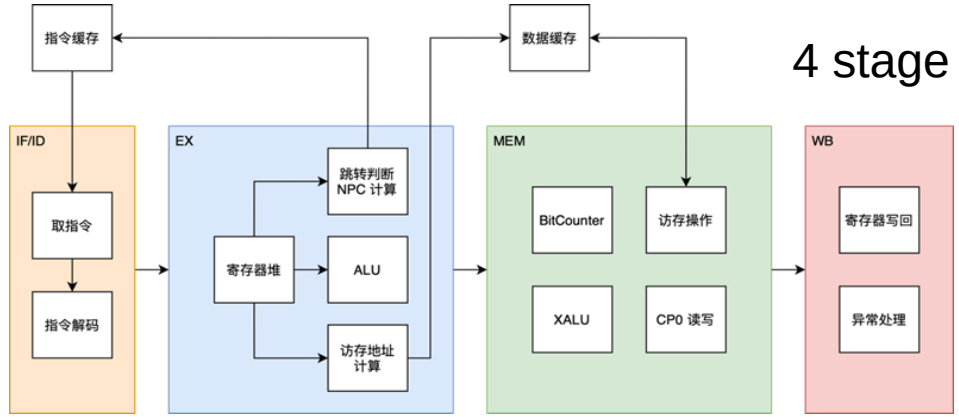
仍是值得讨论和研究的问题

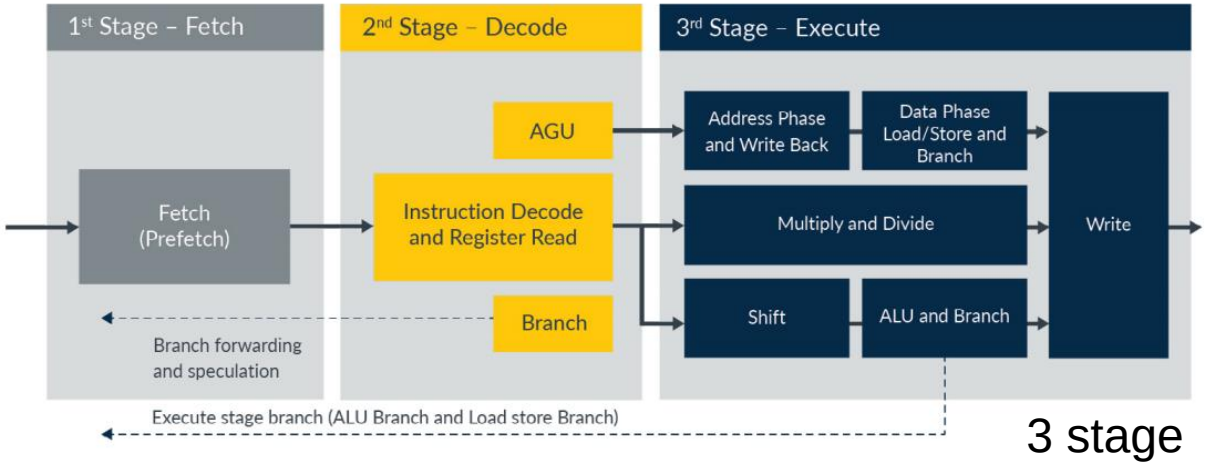
设计复杂度ヽ(ﾟ - ﾟ)ノ ⇐ **优化一个目标**往往是以**牺牲其他目标**为代价的

1.3 流水线的架构

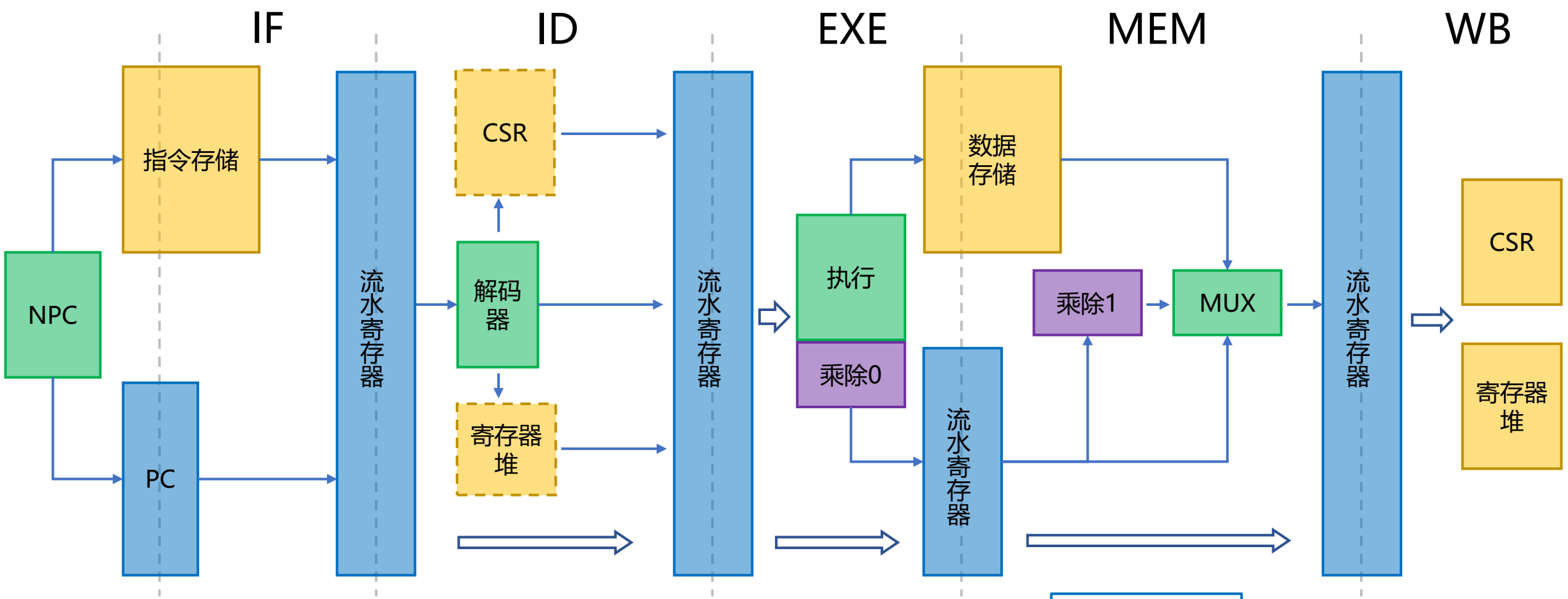
- 2 Stage
  - IF/EXE
- 3 Stage
  - IF/ID/EXE
- 4 Stage
  - IF/ID/EXE/WB
- 5 Stage
  - IF/ID/EXE/MEM/WB







1.3 流水线的架构——经典5级流水线



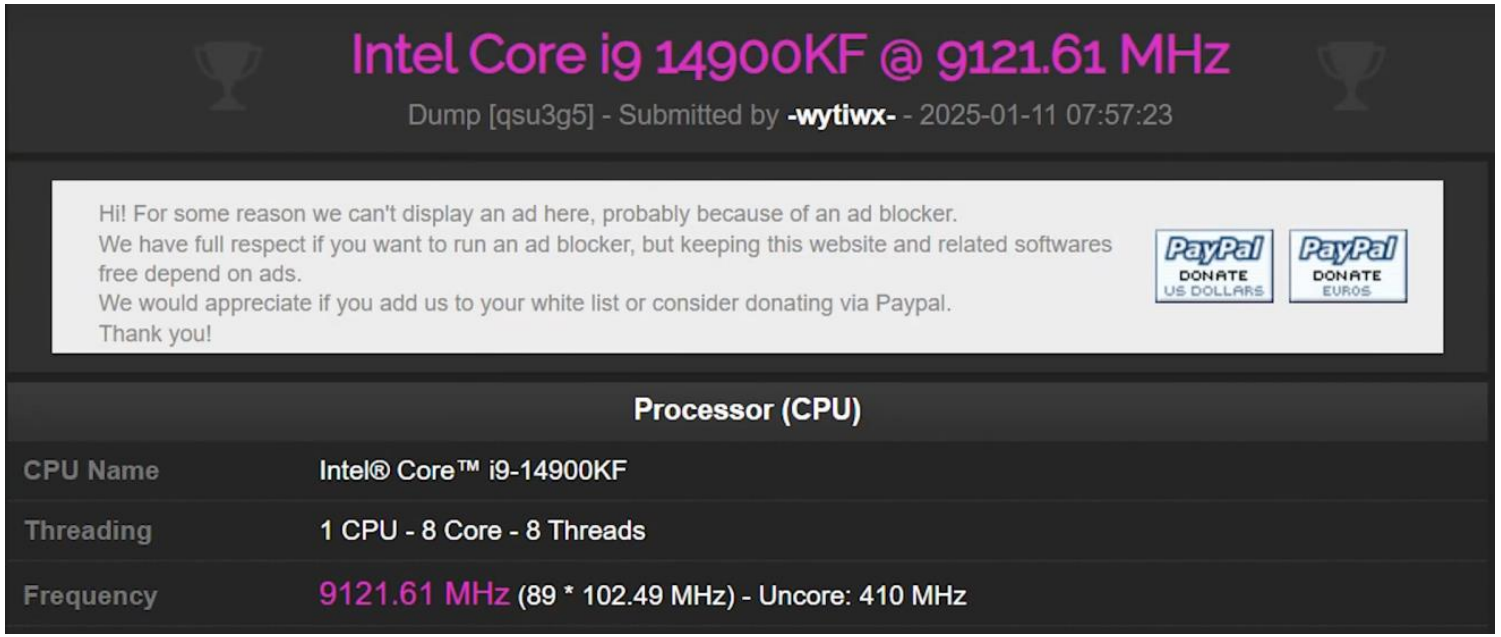
随心所欲
经验主义

1.3 流水线的架构——处理器主频战争

既然增长流水线可以提高主频

极端一点100级流水怎么样(｀ω´)?

让全人类感谢我（doge

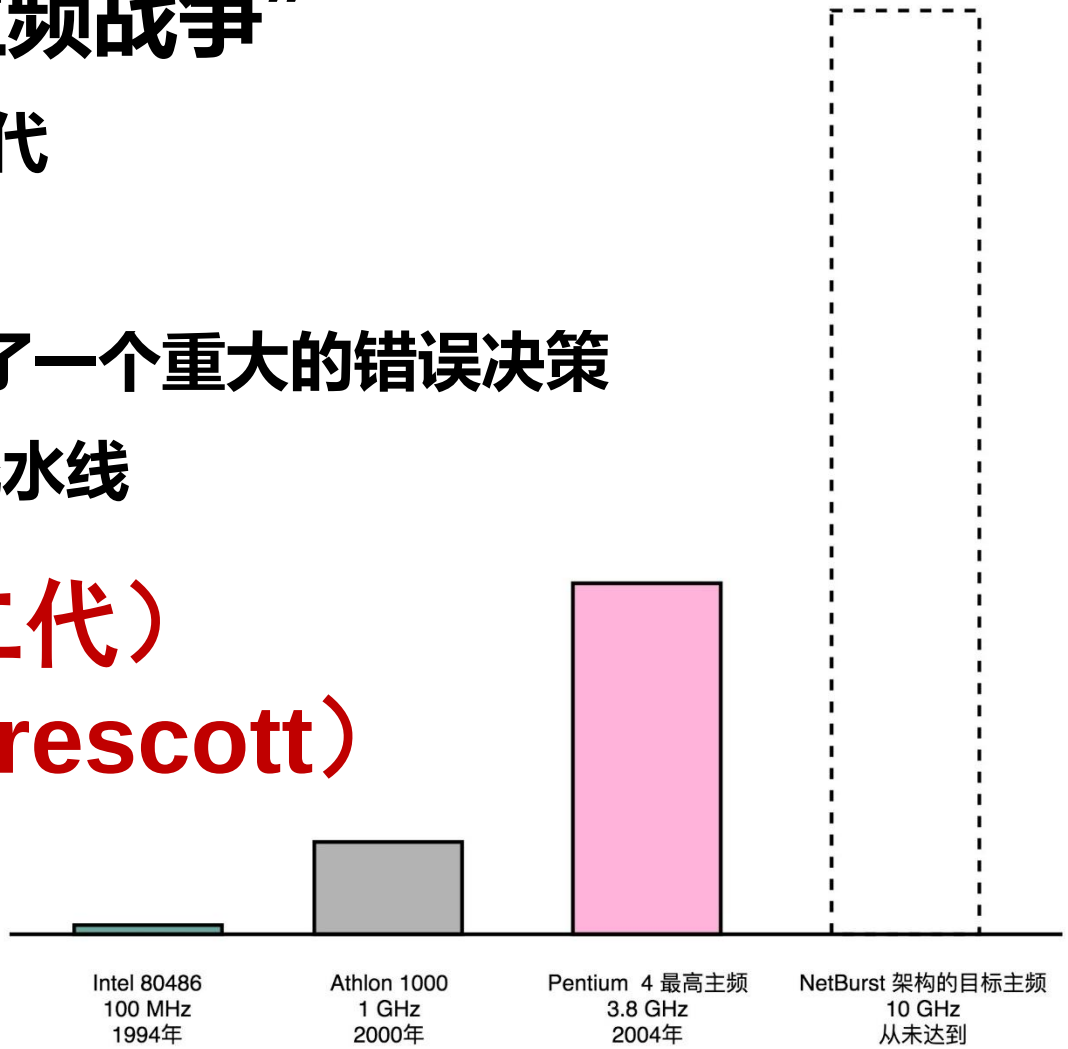


1.3 流水线的架构——处理器主频二三事

- Intel 真的想过这个事——“主频战争”
  - 面对AMD的竞争压力Intel推出了新一代
    NetBurst 架构 CPU
  - 为了达到10GHz，Intel的工程师做出了一个重大的错误决策
    在 NetBurst 架构上，使用了超长的流水线

20级流水（第一、二代）
31级流水（第三代Prescott）

  - 当下的主流处理器多为13-15级流水



1.3 流水线的架构——处理器主频战争

- Intel 真的想过这个事——“主频战争”
  - 面对AMD的竞争压力Intel推出了新一代 NetBurst 架构 CPU
  - 为了达到10GHz，Intel的工程师做出了一个重大的错误决策

    在 NetBurst 架构上，使用了超长的流水线 ==> 31级流水

⇩

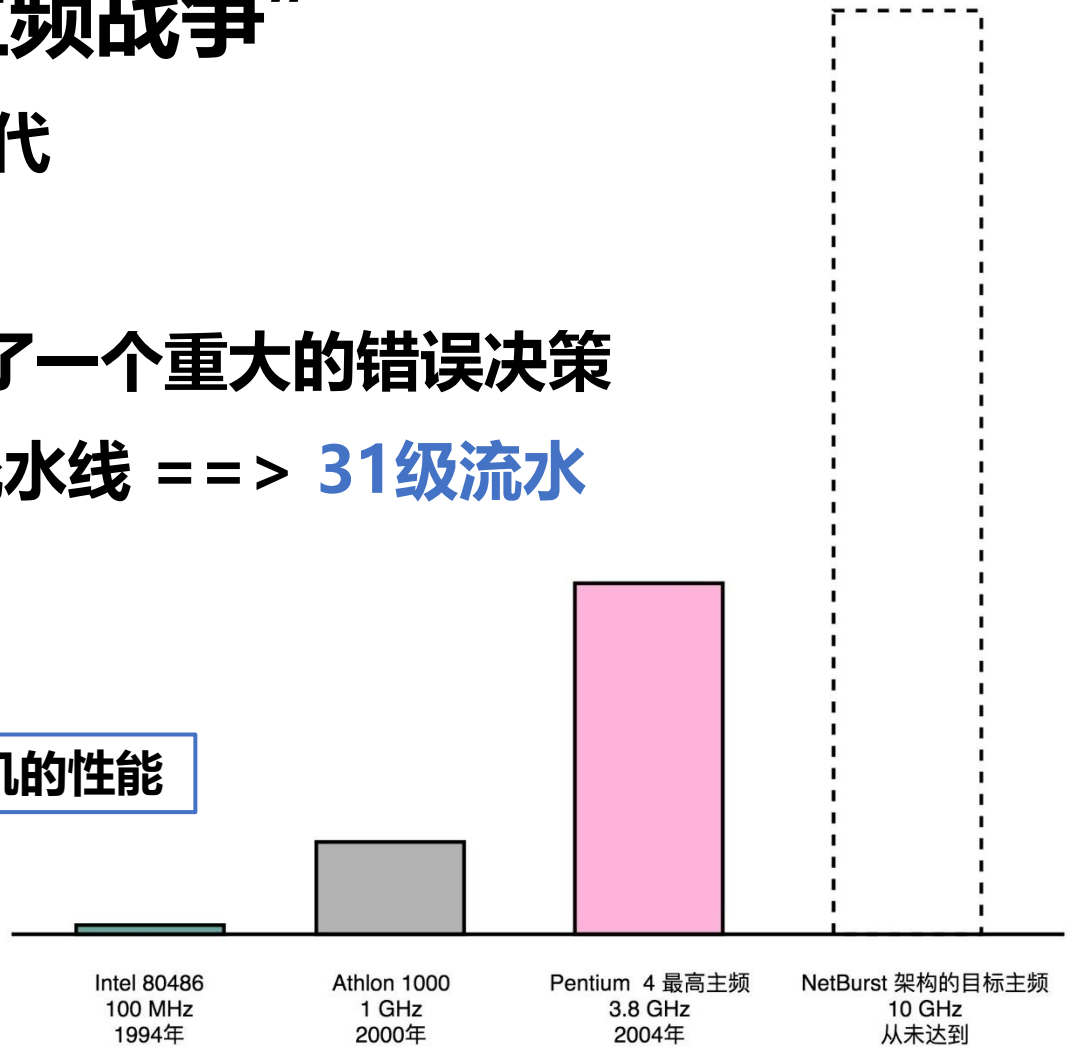
功耗问题、性能降低及冒险依赖难题

⇩

不能简单地通过 CPU 的主频来衡量 CPU 乃至计算机整机的性能

⇩

优化一个目标往往是以牺牲其他目标为代价的



1.3 流水线的架构——处理器主频战争

- Intel 真的想过这个事——“主频战争”
  - 面对AMD的竞争压力Intel推出了新一代 NetBurst 架构 CPU
  - 为了达到10GHz，Intel的工程师做出了一个重大的错误决策

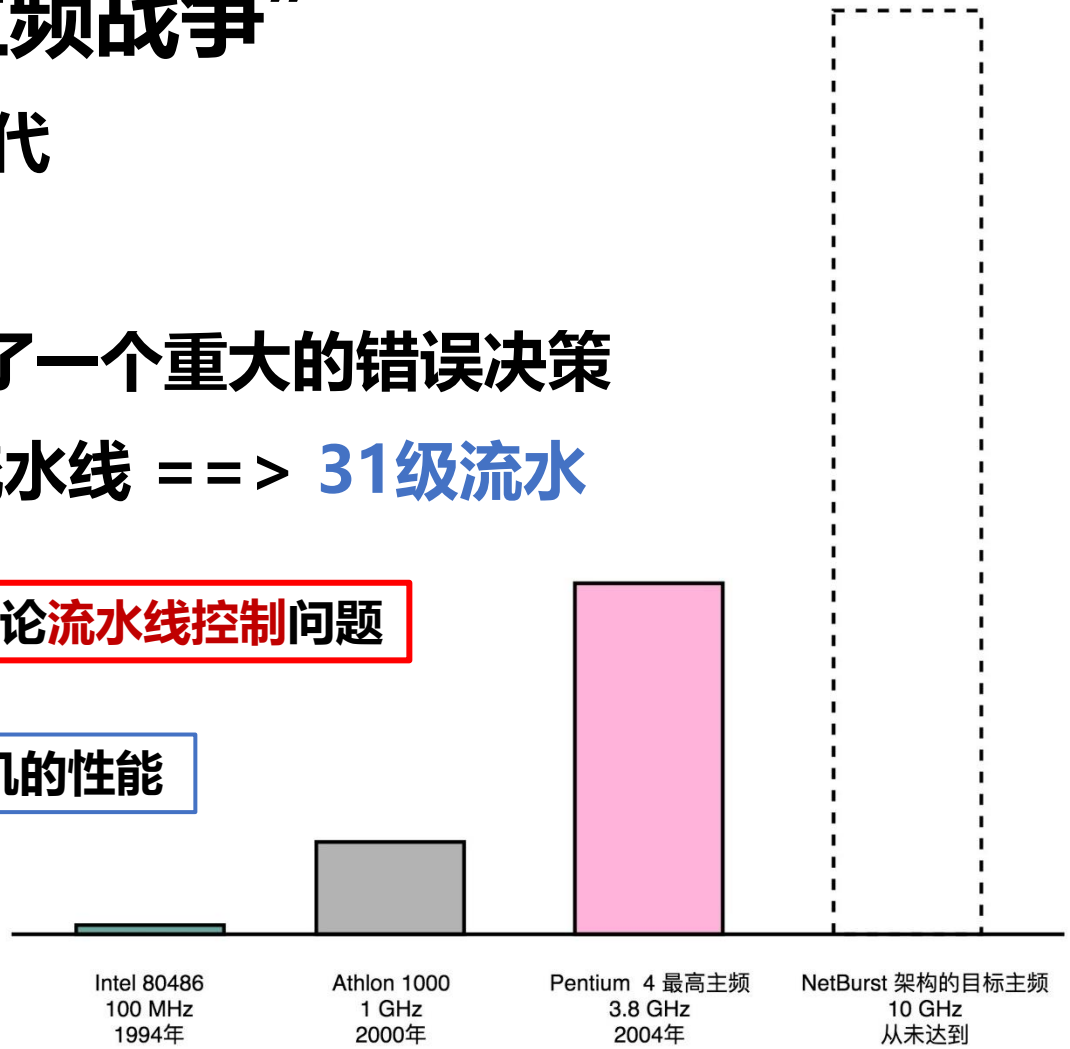
在 NetBurst 架构上，使用了超长的流水线 ==> 31级流水

功耗问题、性能降低及冒险依赖难题

后面将重点讨论流水线控制问题

不能简单地通过 CPU 的主频来衡量 CPU 乃至计算机整机的性能

优化一个目标往往是以牺牲其他目标为代价的



Day2 目录

2.1 流水线握手

握手协议——Stream

注：T指代一个泛型，可以是任意合理数据

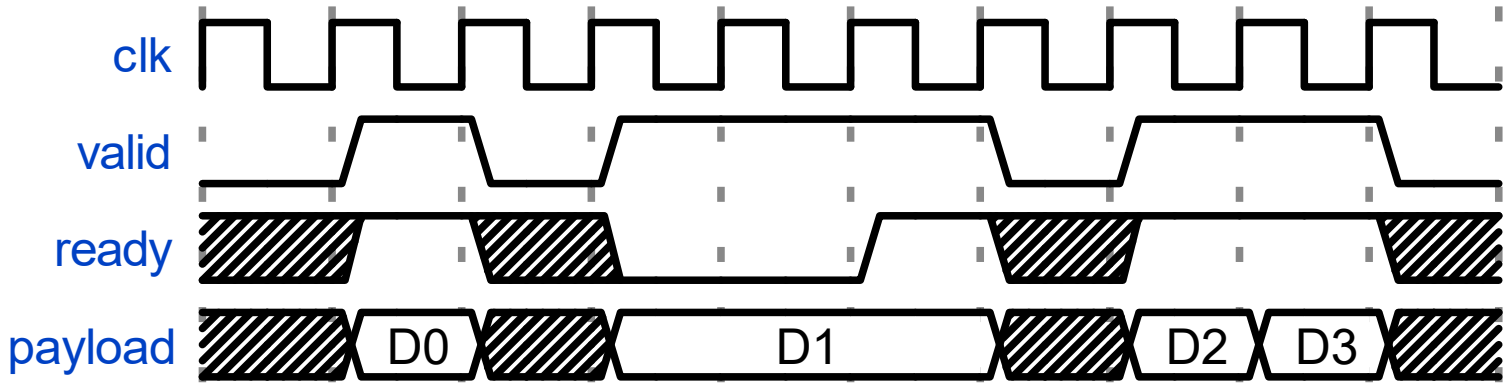
信号	类型	驱动	描述	何时忽略
valid	Bool	Master	当为高时 => 接口上存在有效负载(payload)	
ready	Bool	Slave	当为低时 => 从端不接收传输	valid为低
payload	T	Master	传输任务内容	valid为低

传输仅在**valid和ready同时高**时进行

valid和ready**不以组合逻辑连接**

Tip：可用于
FIFO推入和弹出数据
流水线流动控制
向UART控制器发送请求

时序如图



2.2 流水线控制

□ 指令执行的条件

□ 指令执行的过程

1、获取所有的操作数 => 寄存器 or 立即数

2、按照指令描述的语义获得“计算结果”

3、使用“计算结果”修改处理器状态 => 这里特指ISA规定的状态

Pipe Reg 是优化引入的额外状态
这类状态**对软件不可见**

指令完整执行 ⇩ 对应以下条件

条件一：正确获得操作数

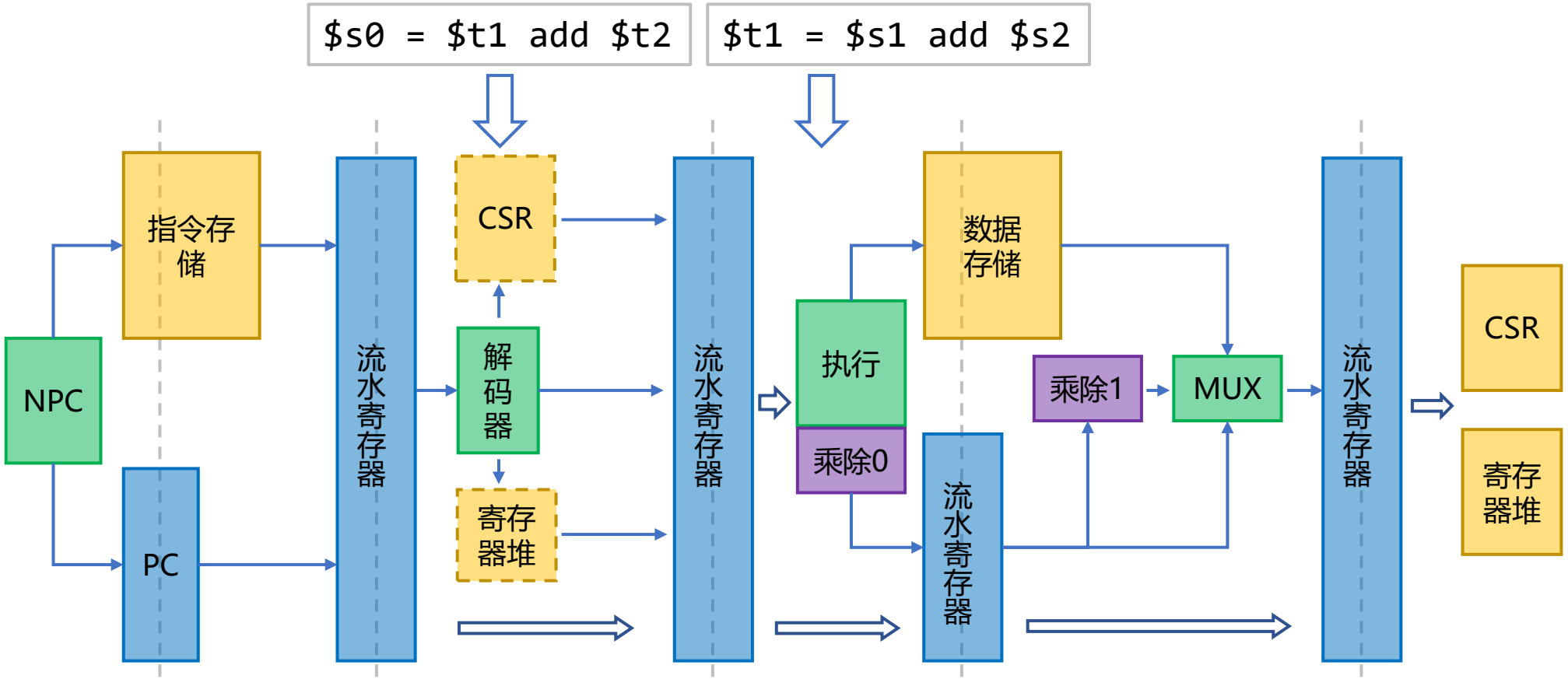
条件二：“资源”就绪（Ready）

条件三：有权修改处理器状态

2.2 流水线控制——写后读

条件一：正确获得操作数（数据同步）

为什么会不正确?



2.2 流水线控制——写后读

- 条件一：正确获得操作数（数据同步）
  - 为什么会不正确？=> 流水化导致获得结果后不第一时间写回
  - 如何解决?
    - 新指令等待，直到数据“可得”（🎵等待~~ 永久的等待~~🎵）
    - 假装写回，新指令认为“可得” => 前递（真写了吗，如写）
  - 前递条件
    - 存在 RAW 冲突 && 新数据Valid
  - 另外：没有什么问题是暂停一拍不能解决的，如果有那就“两”拍

2.2 流水线控制——写后读

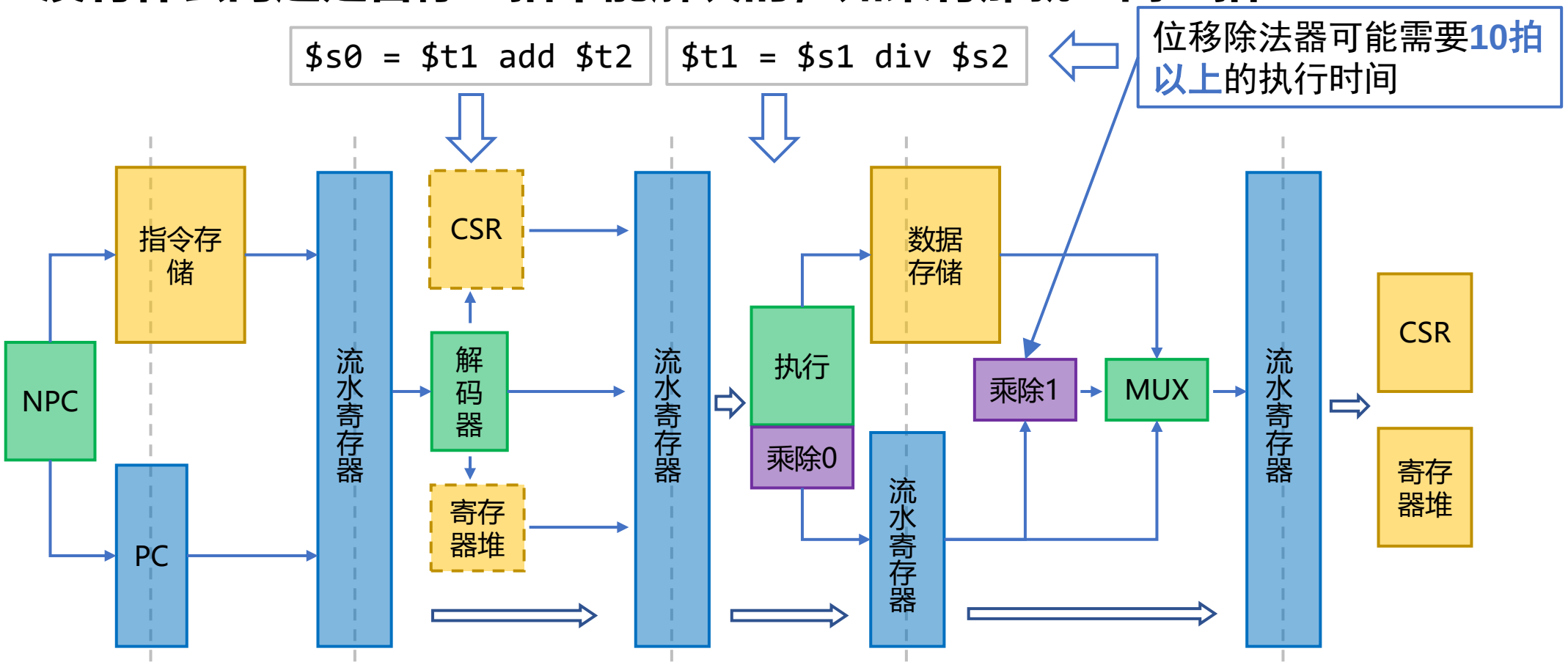
- 条件一：正确获得操作数（数据同步）
  - 为什么会不正确？=> 流水化导致获得结果后不第一时间写回
  - 如何解决?
    - 新指令等待，直到数据“可得”（等待~~ 永久的等待~~）
    - 假装写回，新指令认为“可得” => 前递（真写了吗，如写）
  - 前递条件
    - 存在 RAW 冲突 && 新数据Valid
  - 另外：没有什么问题是暂停一拍不能解决的，如果有那就“两”拍

无法获得有效数据的指令不可以被执行

2.2 流水线控制——写后读

条件一：正确获得操作数（数据同步）

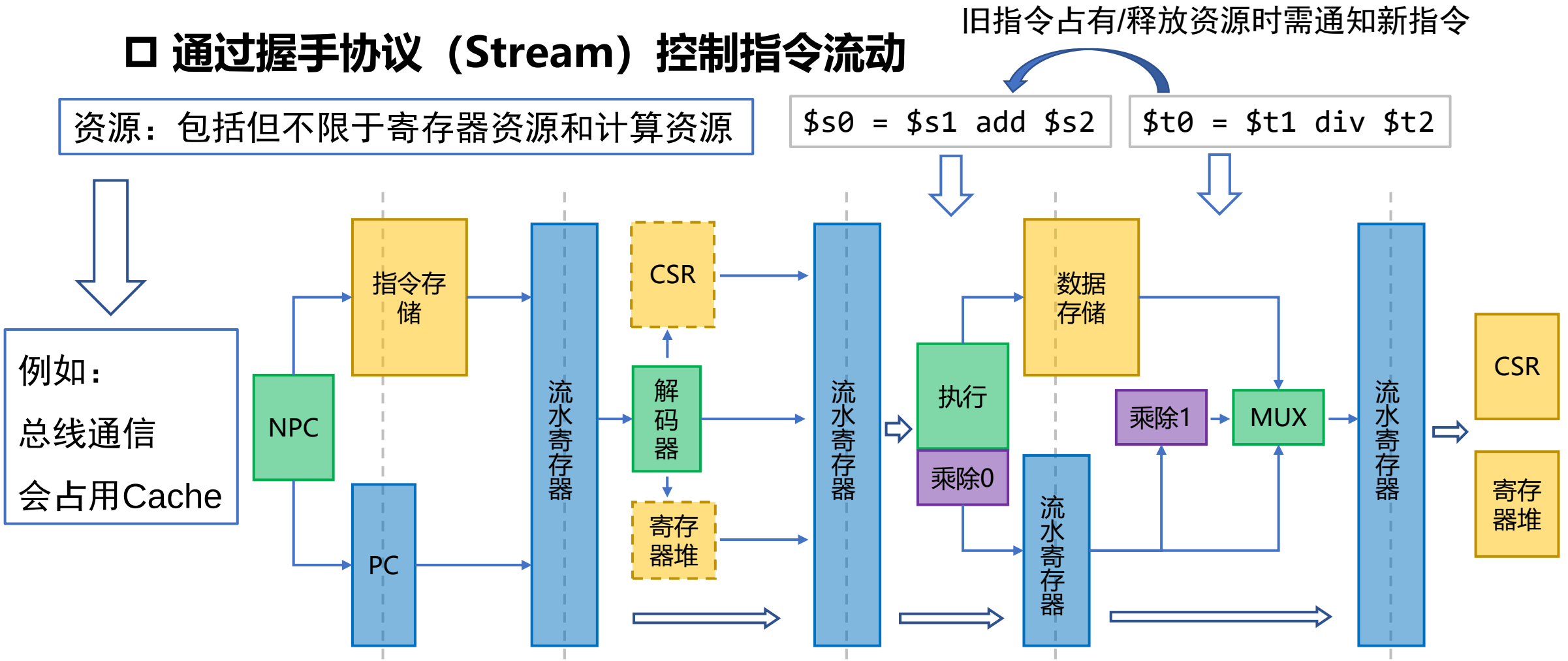
没有什么问题是暂停一拍不能解决的，如果有那就“两”拍



2.2 流水线控制——资源阻塞

□ 条件二：资源就绪（状态同步）

□ 通过握手协议（Stream）控制指令流动



2.2 流水线控制——指令提交

□ 条件三：有权修改处理器状态（状态同步）

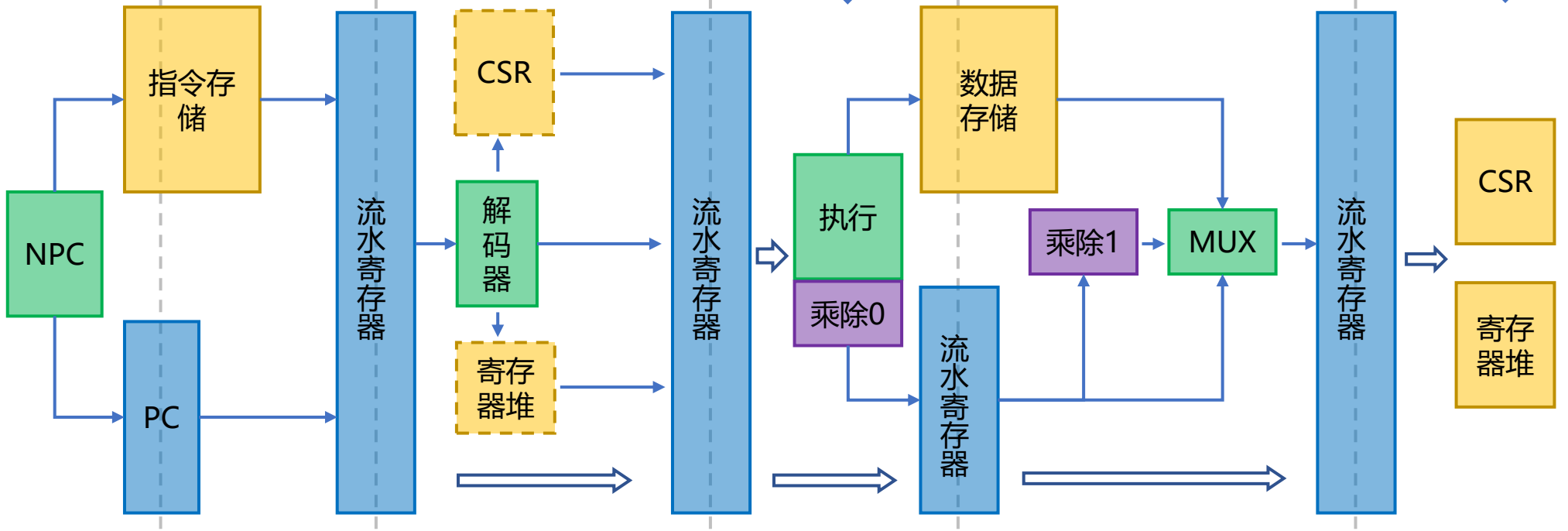
□ 当且仅当**退休或修改可恢复**时指令可修改处理器状态

即使产生结果也无权修改处理器状态 => `$s0 = $s1 add $s2`

跳转到异常入口
冲刷整条流水线

Syscall指令

优先级：
任意退休指令
分支指令



2.3 小结

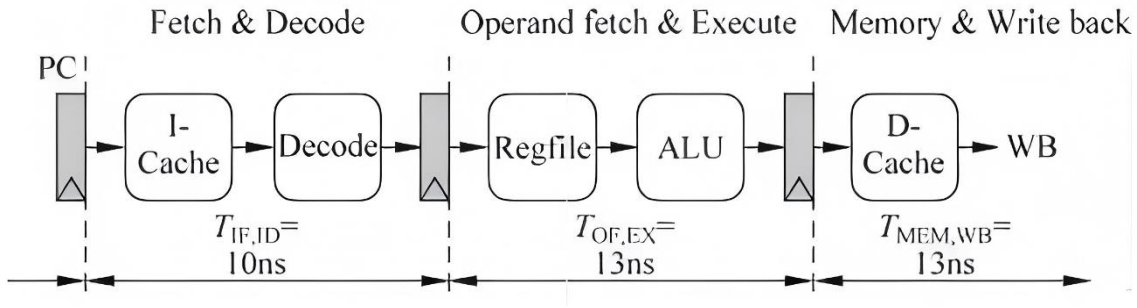
- 一二章规则怪谈(~￣▽￣)~
  - 1、流水线的设计很自由，不同的需求往往有不同的设计
  - 2、流水线的冒险是不可避免的 => 这是流水并行的代价
  - 3、同时使用暂停和前递的方法解决数据同步（冒险）问题
  - 4、指令在退休前一直处于推测执行状态

    => 指令自以为看到了处理器最新且正确的状态
  - 5、退休的指令一定看到了处理器最新且正确的状态

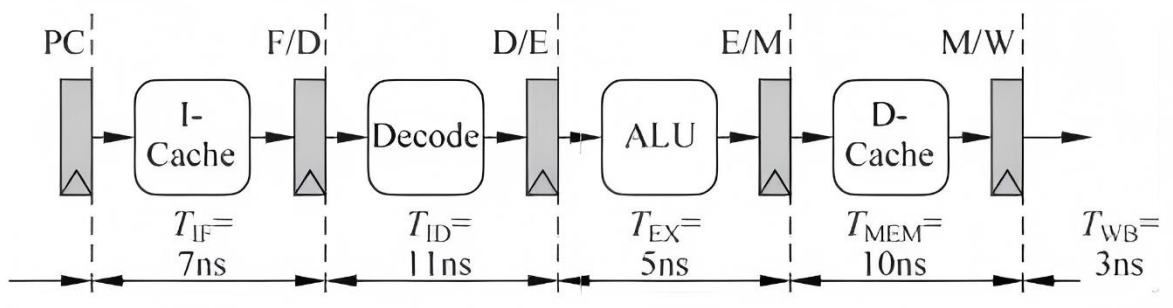
Day2 目录

3.1 超标量处理器简介

□ 流水线纵向切分

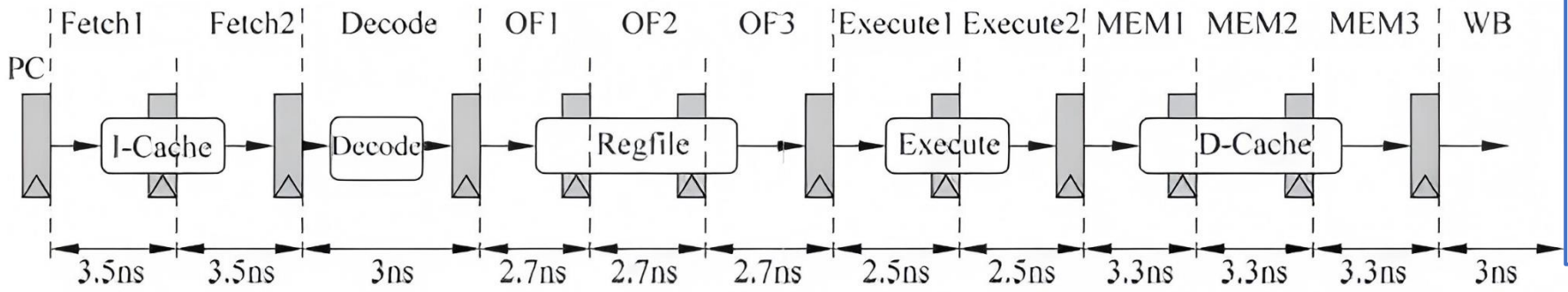


3 Stage



5 Stage

流水线横切



12 Stage

长流水线引发的
数据依赖等问题
使得**一味加长流水线**
难以获得理想的收益

3.1 超标量处理器简介——回忆：流水线控制

□ 条件二：资源就绪（状态同步）

□ 通过握手协议（Stream）控制指令流动

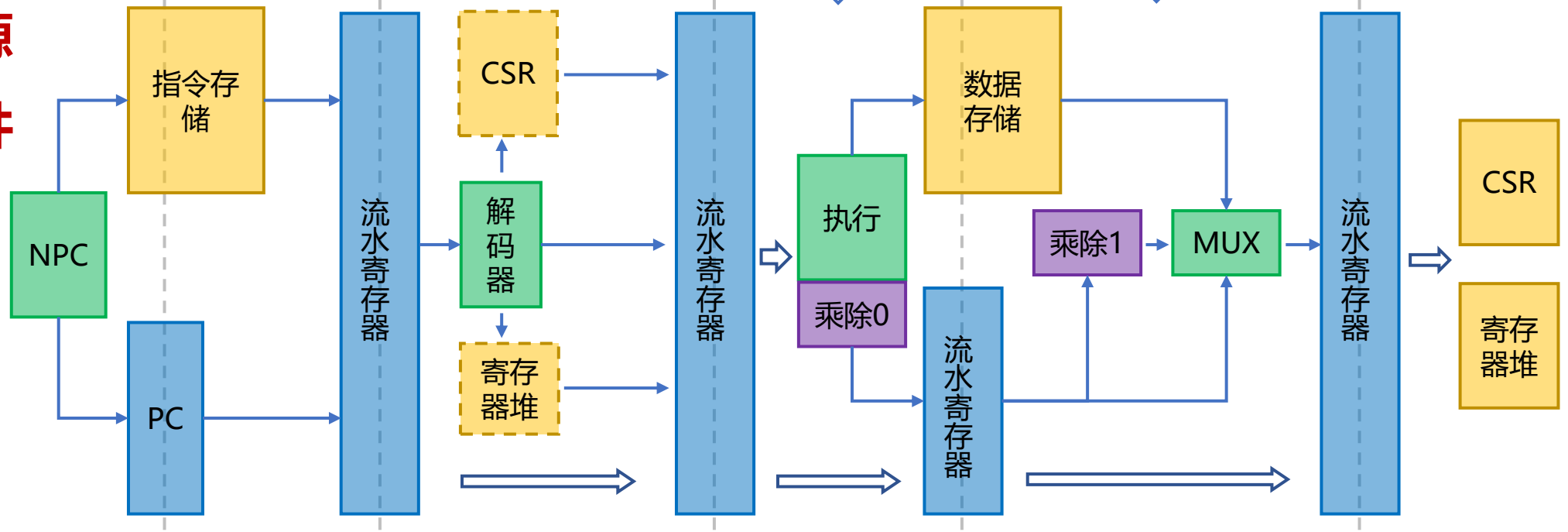
旧指令占有/释放资源时需通知新指令

事实上这两条指令是**可以并行执行**的 => `$s0 = $s1 add $s2` `$t0 = $t1 div $t2`

增加流水资源

放松执行条件

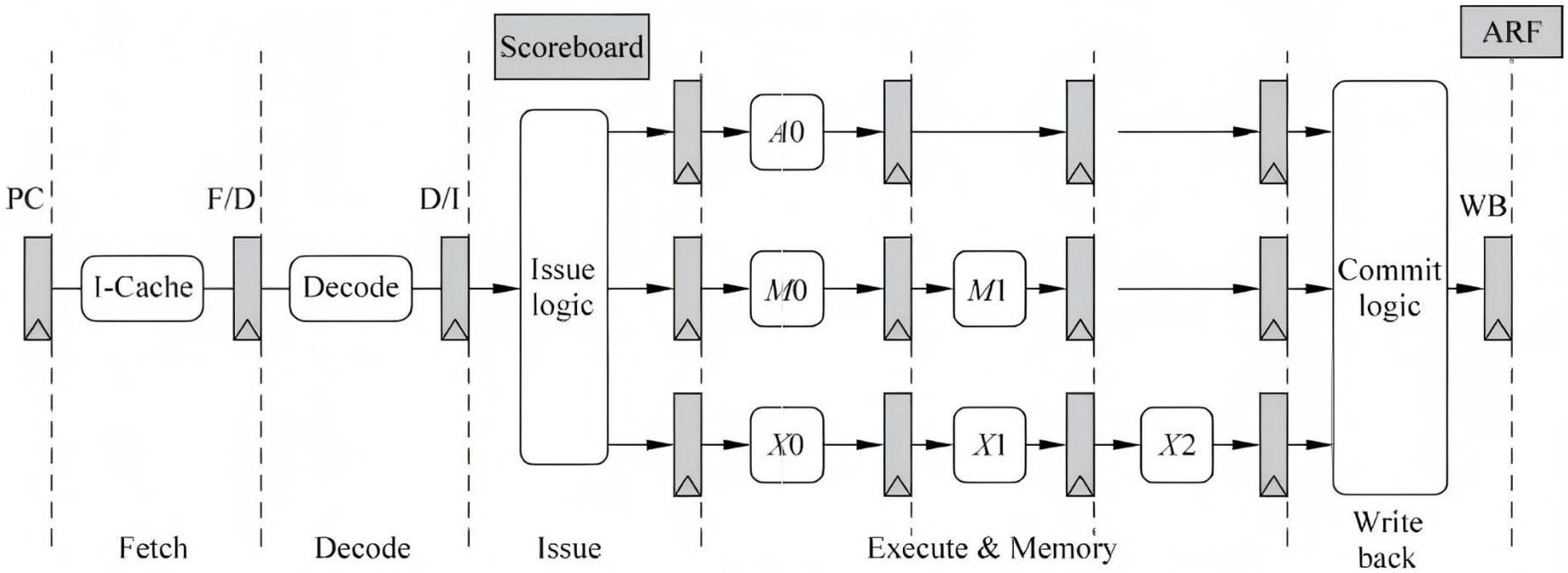
进一步挖掘
指令并行性



3.2 超标量处理器简介

□ 从纵向切分到横向切分——顺序超标量

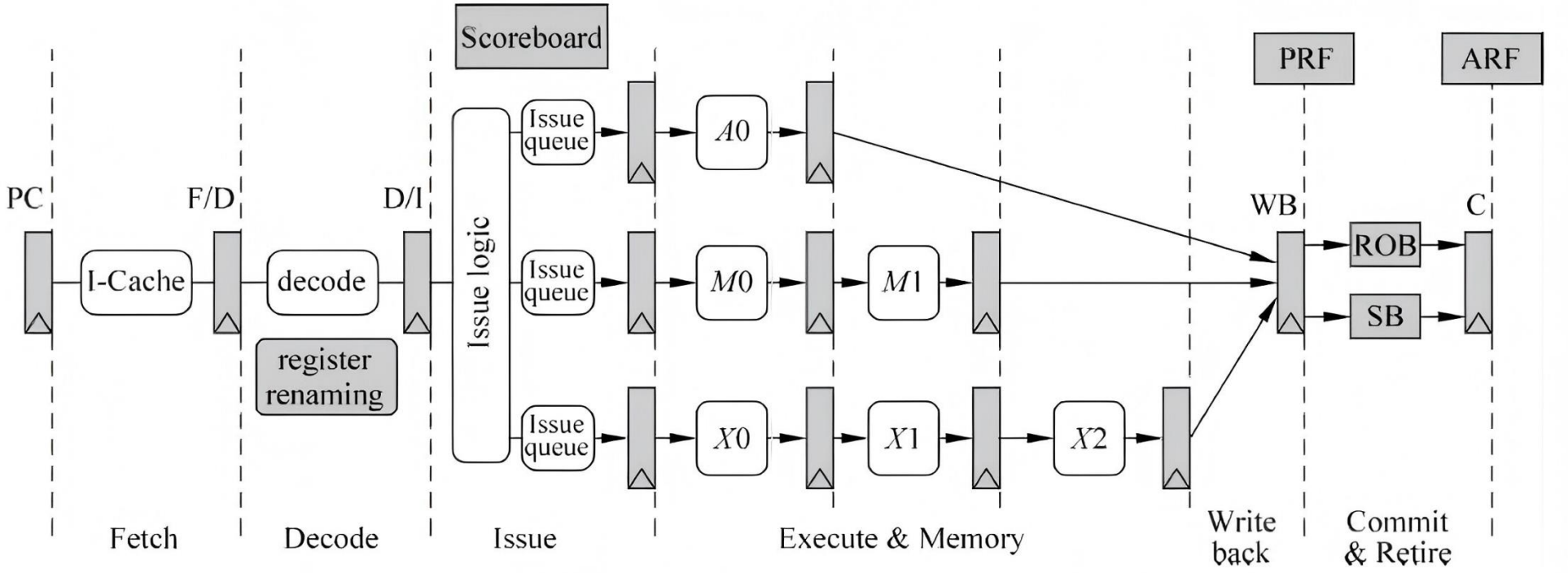
	Frontend	Issue	Write back	Commit
顺序超标量	**in-order**	**in-order**	**in-order**	**in-order**
乱序超标量	in-order	out-of-order	out-of-order	in-order



3.2 超标量处理器简介

□ 从纵向切分到横向切分——乱序超标量

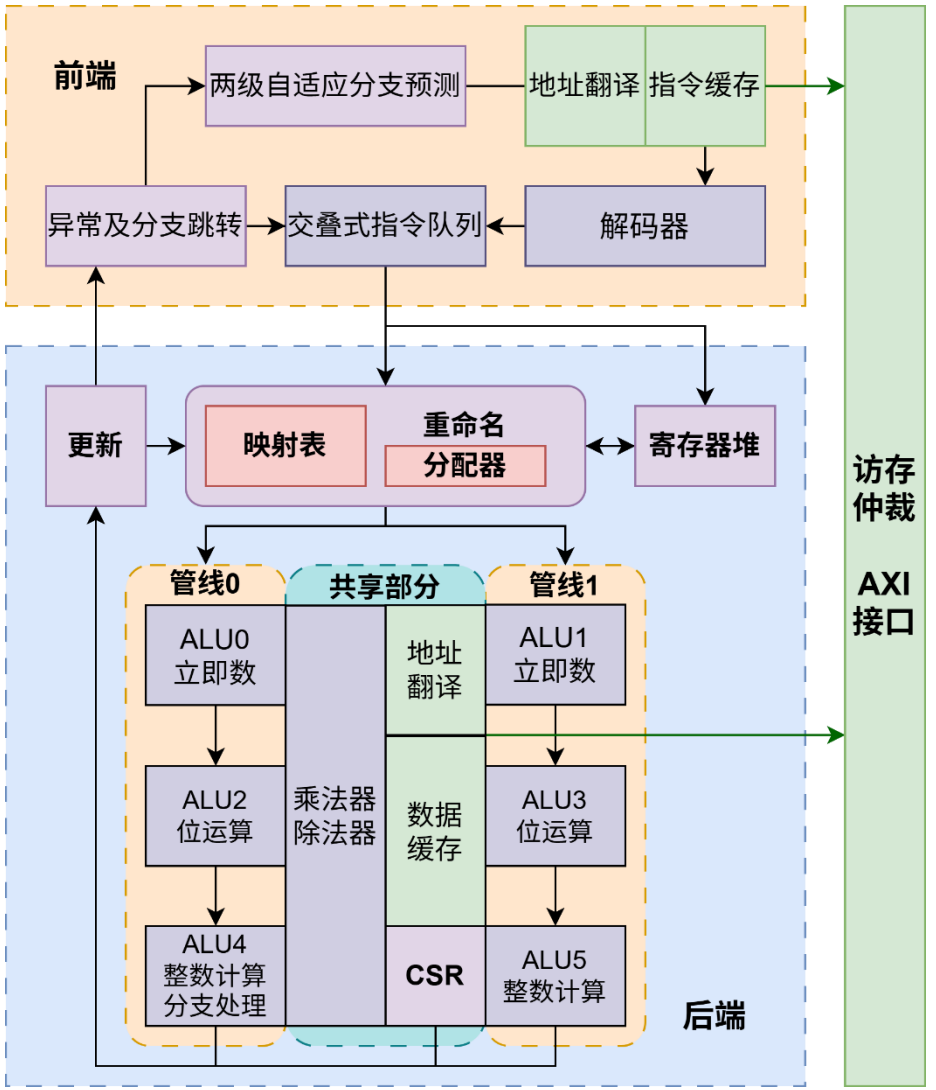
	Frontend	Issue	Write back	Commit
顺序超标量	in-order	in-order	in-order	in-order
乱序超标量	in-order	out-of-order	out-of-order	in-order



3.2 超标量处理器简介

□ 顺序超标量——举个例子

https://github.com/LainChip/LainCore



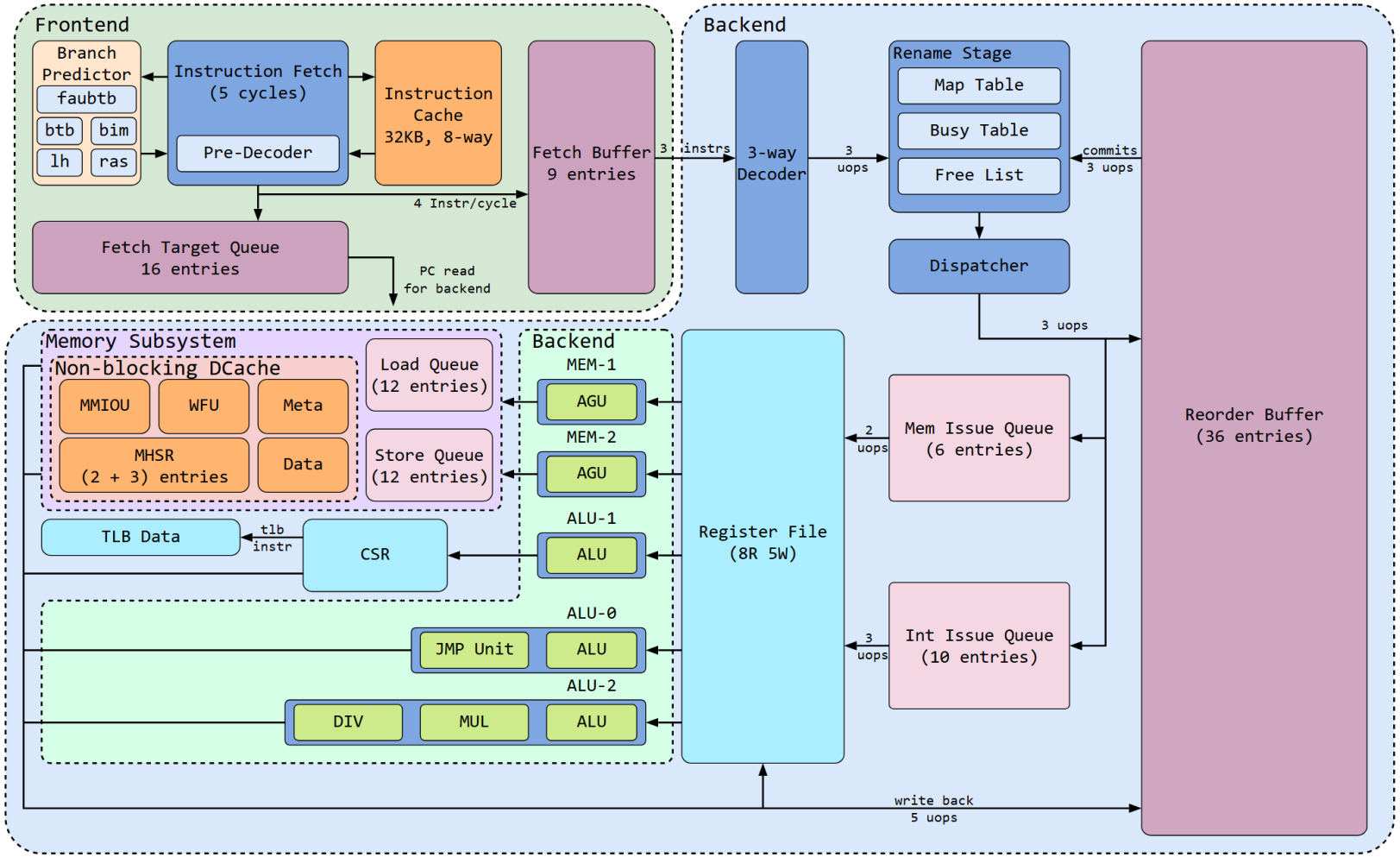
LainCore 北航2023龙芯杯一等奖作品 ⇨

顺序取指、顺序多发、顺序执行、顺序写回、顺序提交

3.2 超标量处理器简介

□ 乱序超标量——举个例子

顺序取指、乱序多发、乱序执行、乱序写回、顺序提交



复旦2024龙芯杯一等奖作品

3发射 后端5执行

https://github.com/iFuProcessor

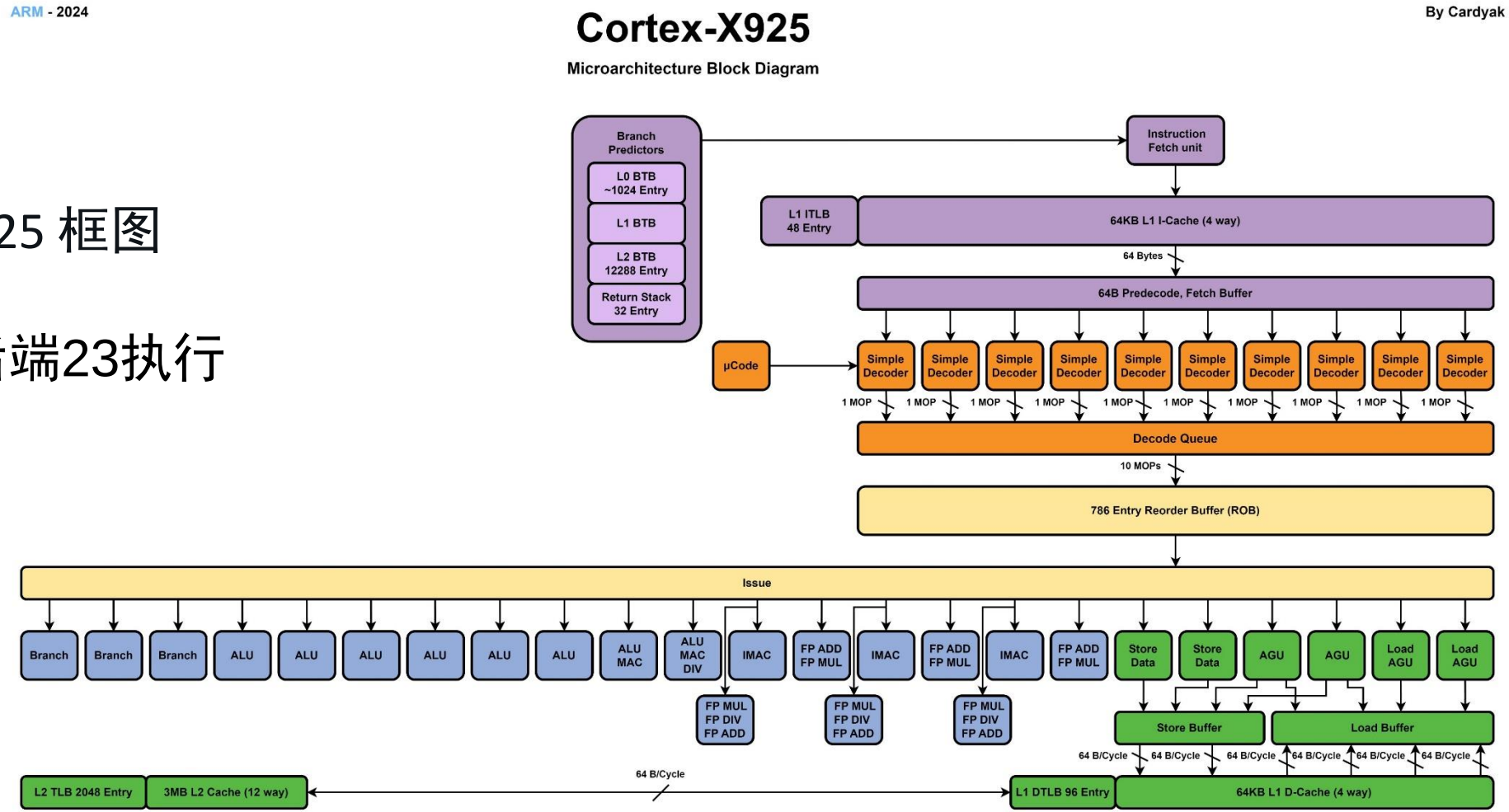
3.2 超标量处理器简介

□ 乱序超标量—工业界例子

顺序取指、乱序多发、乱序执行、乱序写回、顺序提交

Arm x925 框图

10取指 后端23执行



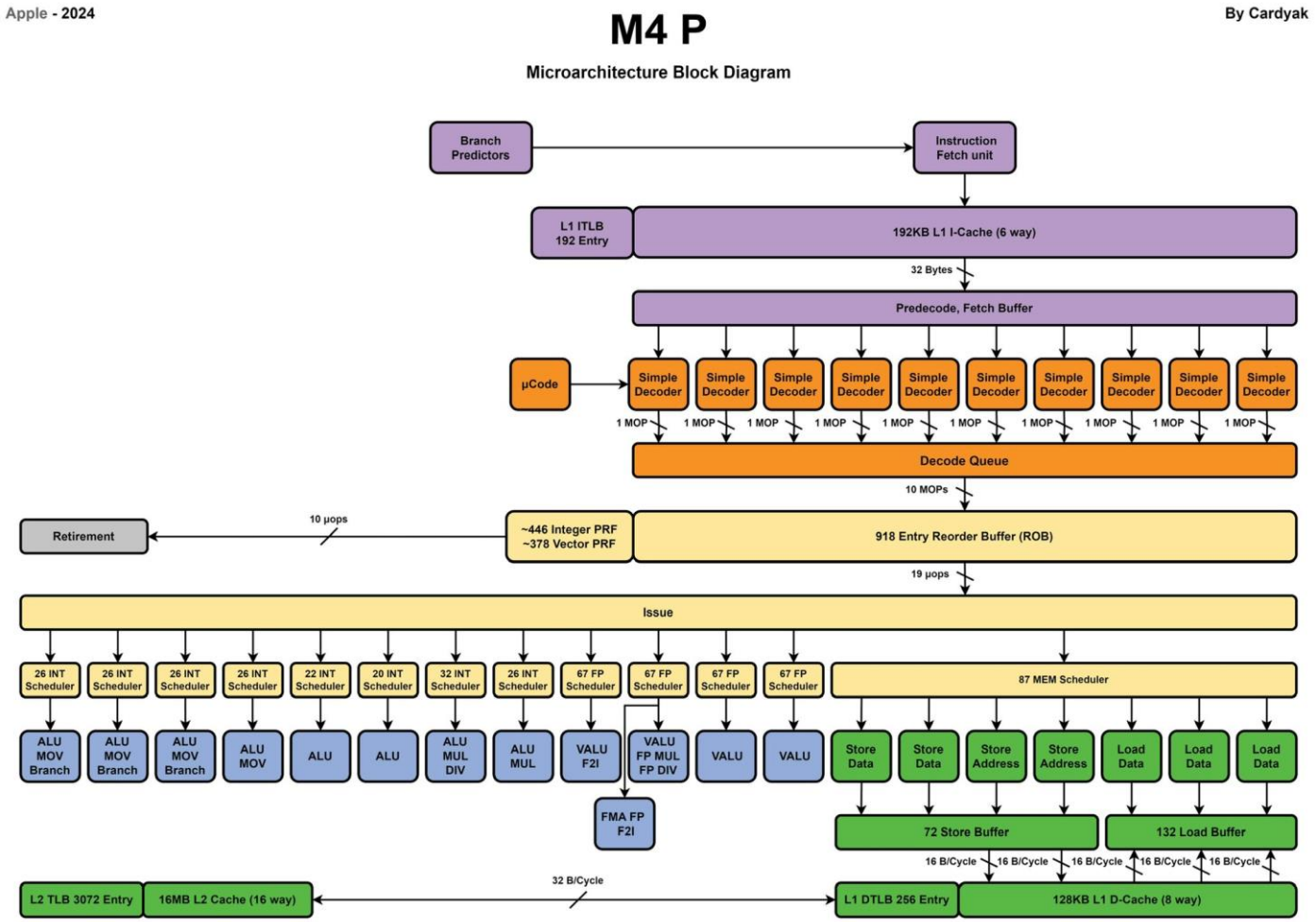
3.2 超标量处理器简介

□ 乱序超标量—工业界例子

顺序取指、乱序多发、乱序执行、乱序写回、顺序提交

Apple M4 框图

10取指 后端19执行



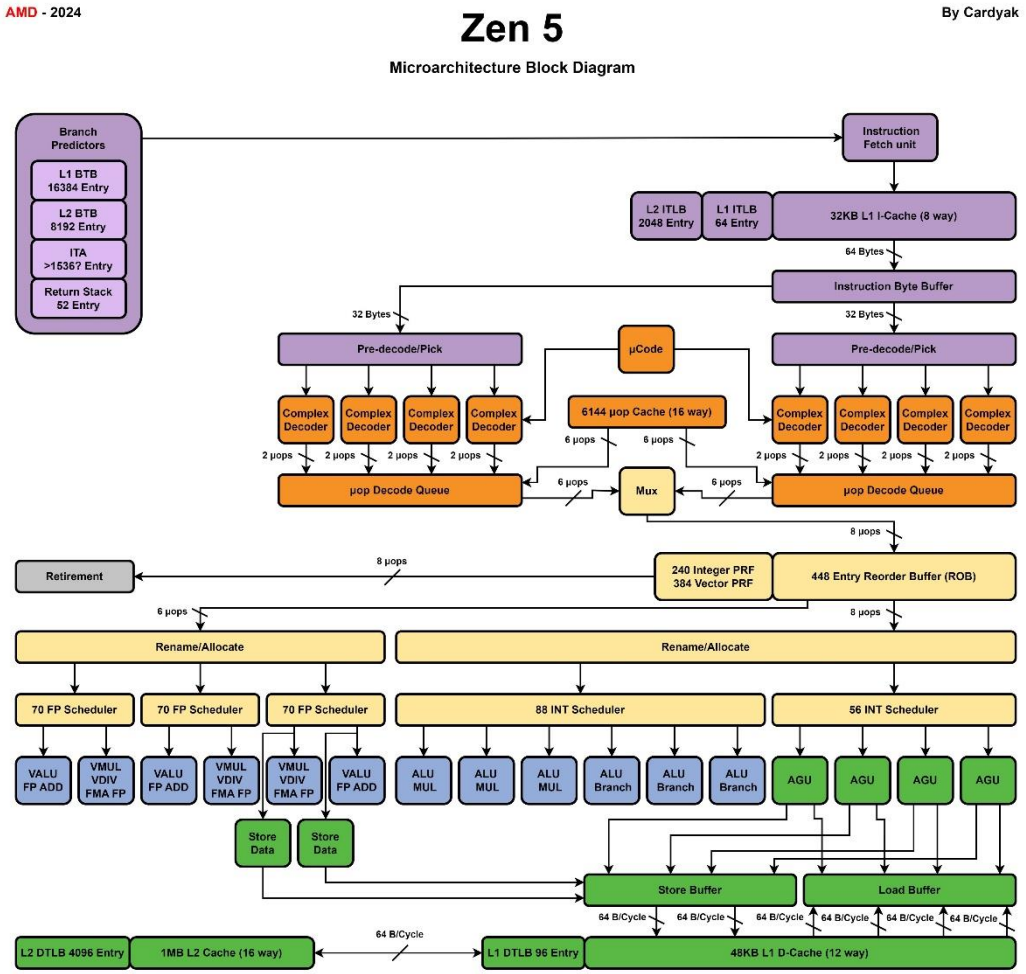
3.2 超标量处理器简介

□ 乱序超标量—工业界例子

顺序取指、乱序多发、乱序执行、乱序写回、顺序提交

AMD Zen5 框图

8取指 后端14执行



3.2 超标量处理器简介

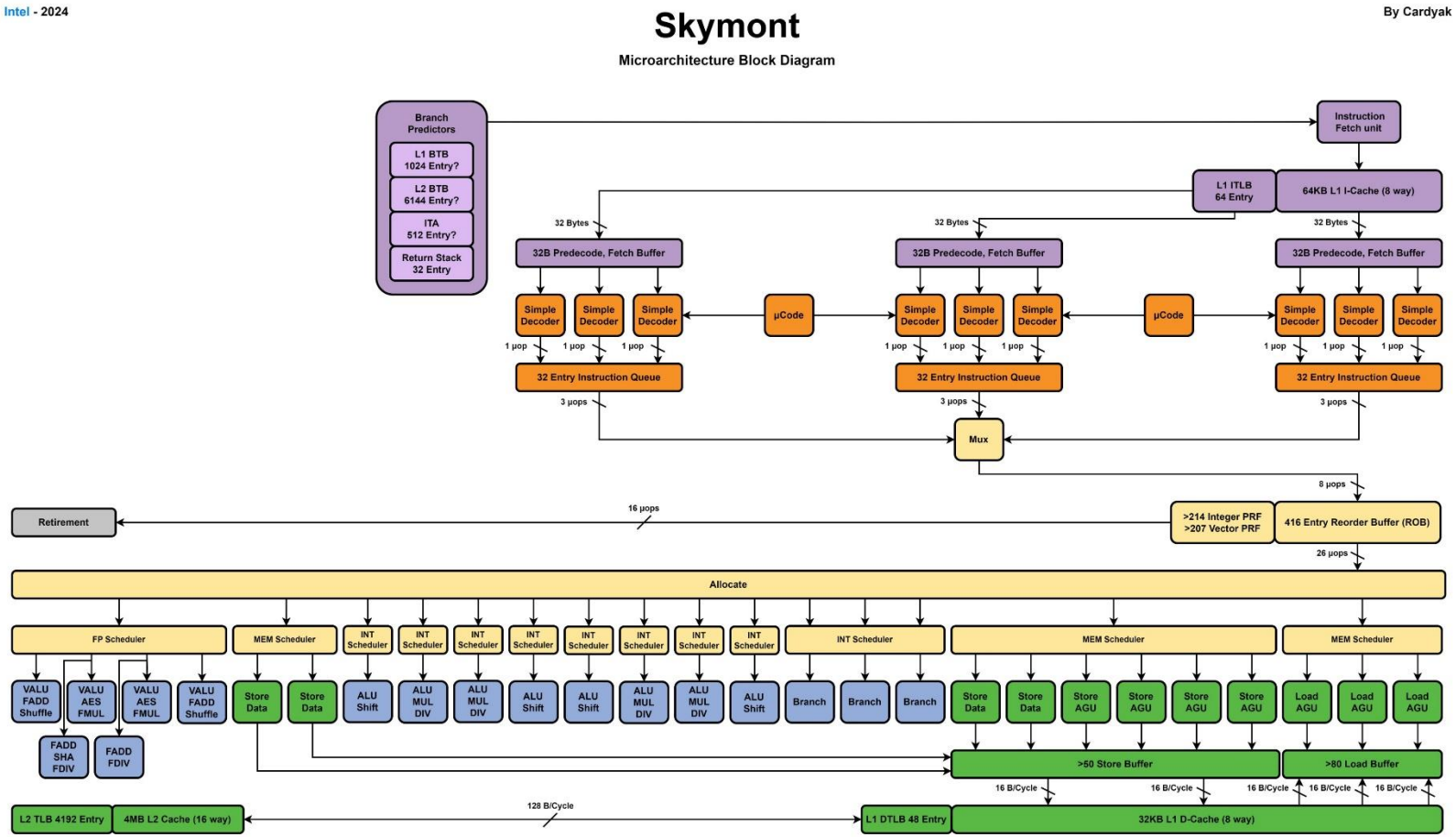
□ 乱序超标量—工业界例子

顺序取指、乱序多发、乱序执行、乱序写回、顺序提交

英特尔 Skymont 框图

8取指 后端26执行

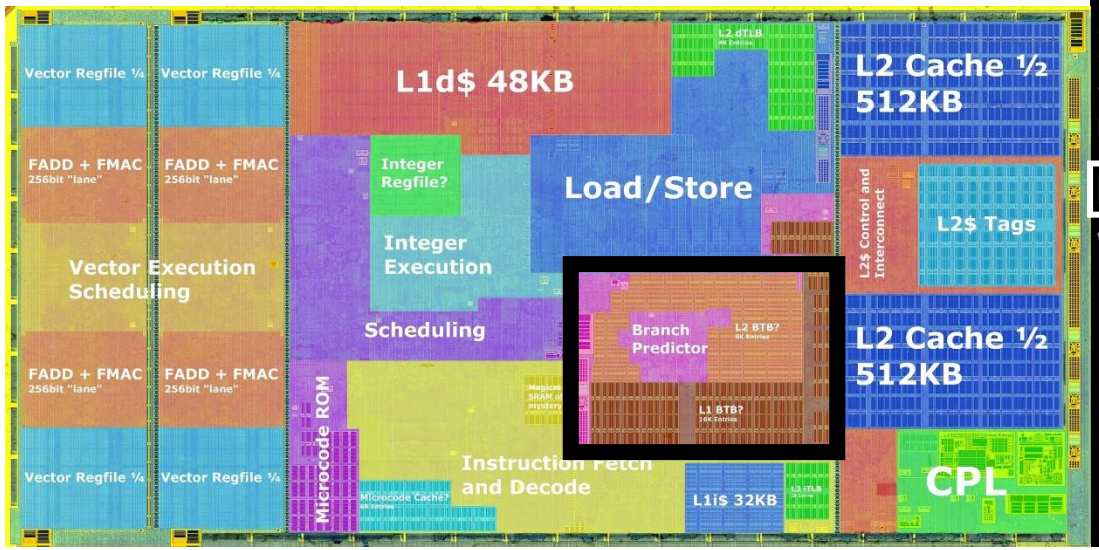
非常夸张

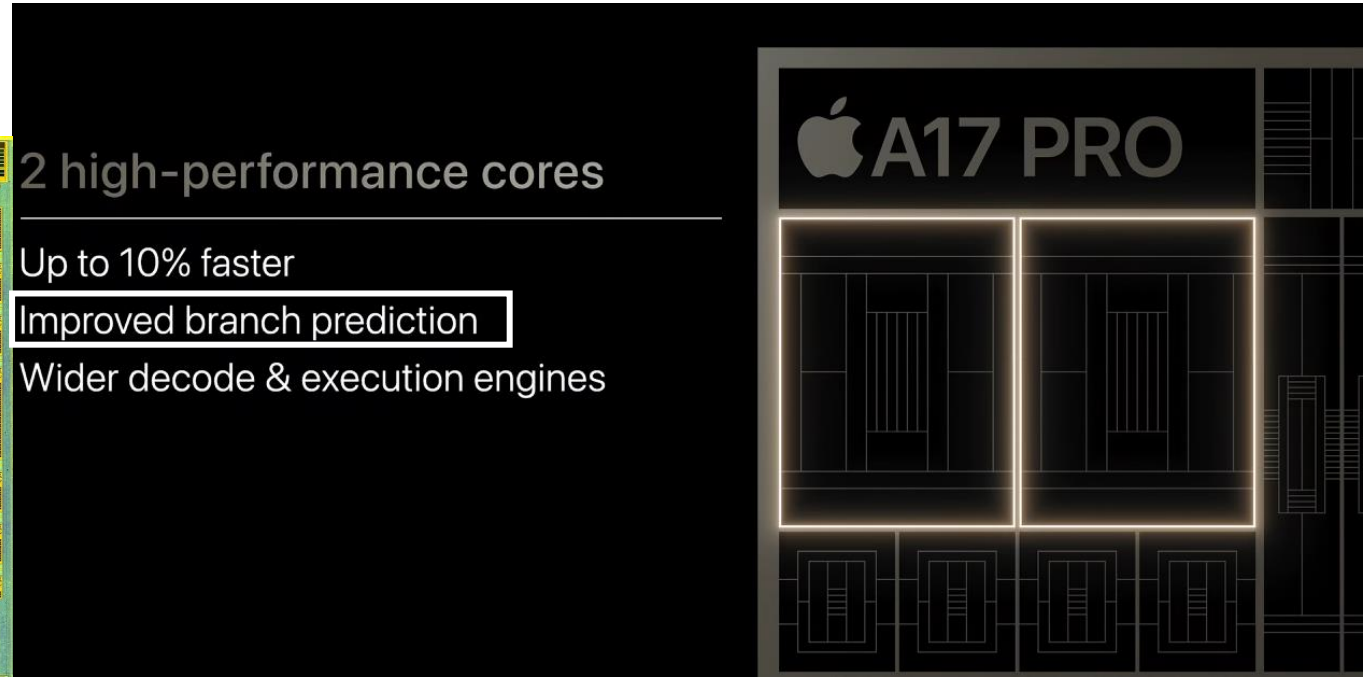


3.3 分支预测

- 继Cache之后的又一大力作
  - 上述的处理器中都会有名为Branch-Predictor的部件
  - 长流水 且 前后端分离 => 分支指令代价沉重
  - 通过预测分支规避风险
  - 成功率 > 90%

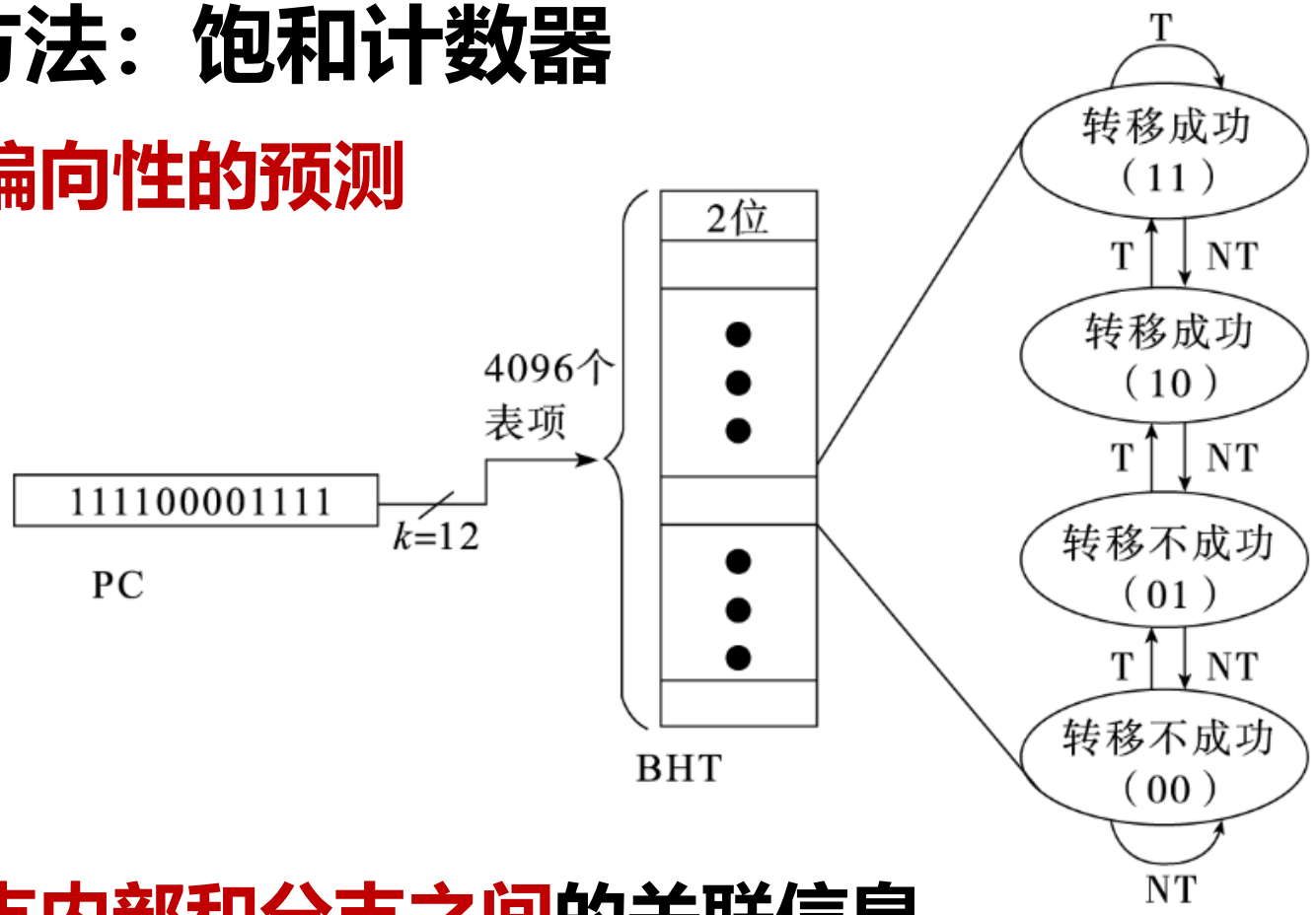
对于**程序特性**的深刻洞察





3.3 分支预测

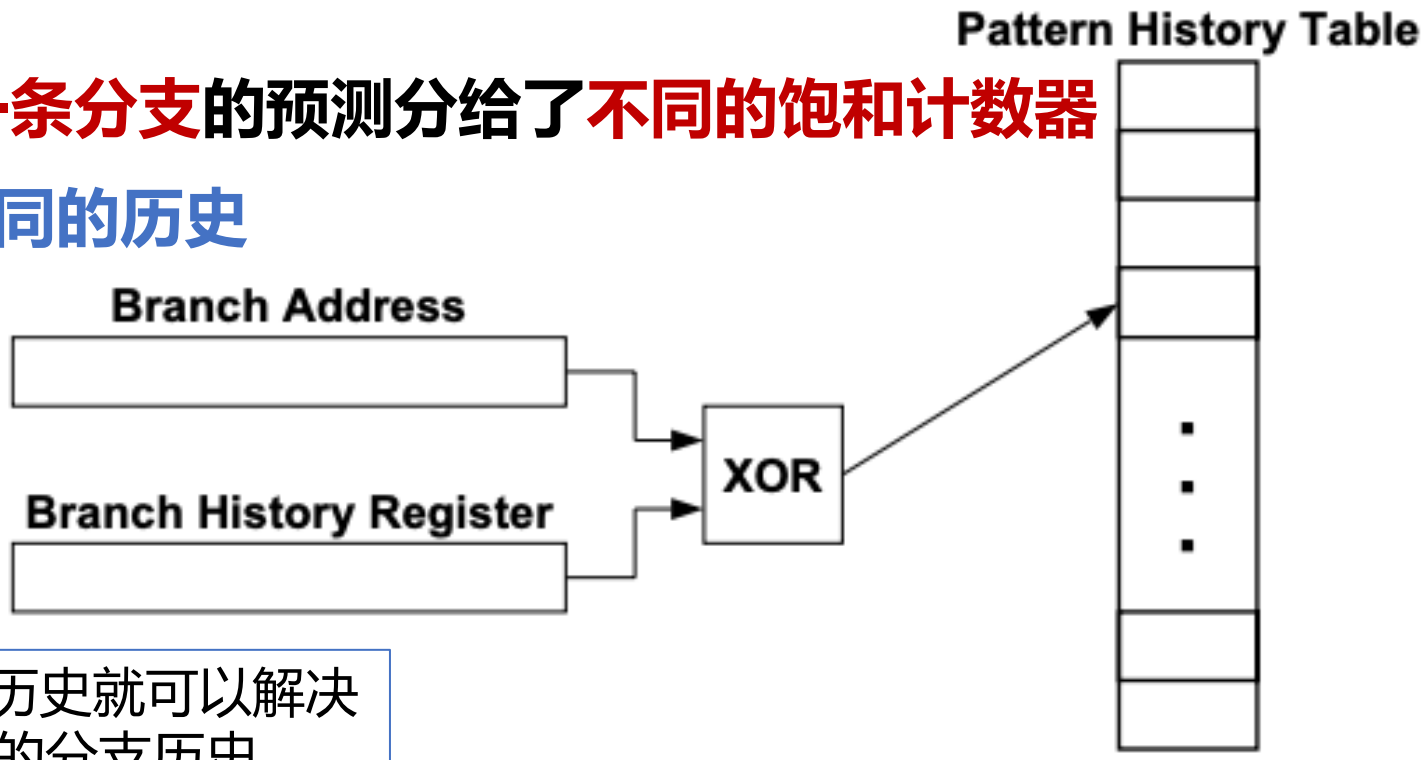
- 最古老的分支预测方法：饱和计数器
  - 饱和计数器只能作出**偏向性的预测**



  - 不能充分利用单个**分支内部和分支之间**的关联信息

3.3 分支预测

- **利用分支历史信息的预测器：GShare 预测器**
  - 将分支的 PC 与一个**全局历史寄存器** XOR 以后作为 index 来选择 PHT 中的一个饱和计数器
  - **不同全局分支历史下的同一条分支的预测分给了不同的饱和计数器**
  - GShare 预测器能够**区分不同的历史**
  - 正确预测循环退出



循环退出的预测只需要单个 PC 的历史就可以解决
而分支间相关的例子则必须要全局的分支历史。

3.3 分支预测

□ 锦标赛预测器

循环退出的预测只需要单个 PC 的历史就可以解决
而分支间相关的例子则必须要全局的分支历史。

局部历史和全局历史
两个预测器**分别独立预测**

再由一个饱和计数器**选择**
子预测器提供的方向预测

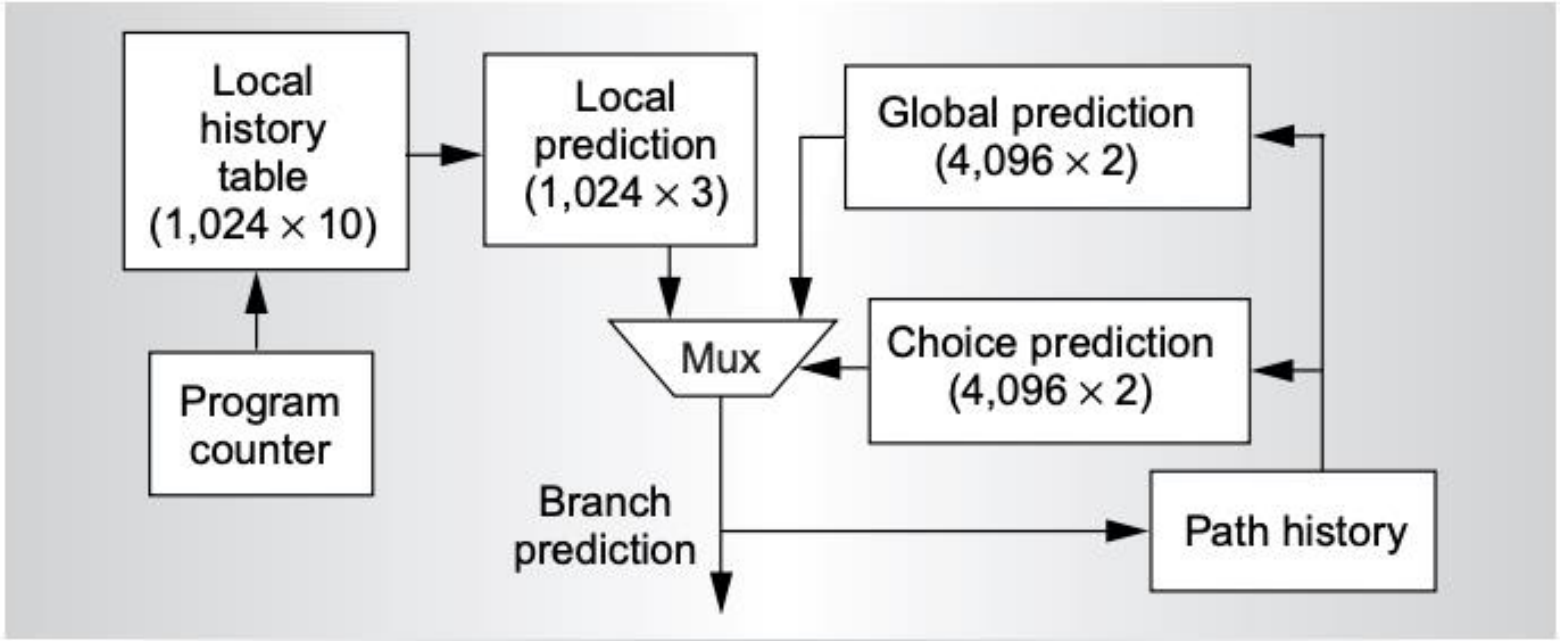


Figure 4. Block diagram of the 21264 tournament branch predictor. The local history prediction path is on the left; the global history prediction path and the chooser (choice prediction) are on the right.

R. E. Kessler, "The Alpha 21264 microprocessor," in IEEE Micro, vol. 19, no. 2, pp. 24-36, March-April 1999

3.3 分支预测

□ 现代分支预测算法——TAGE

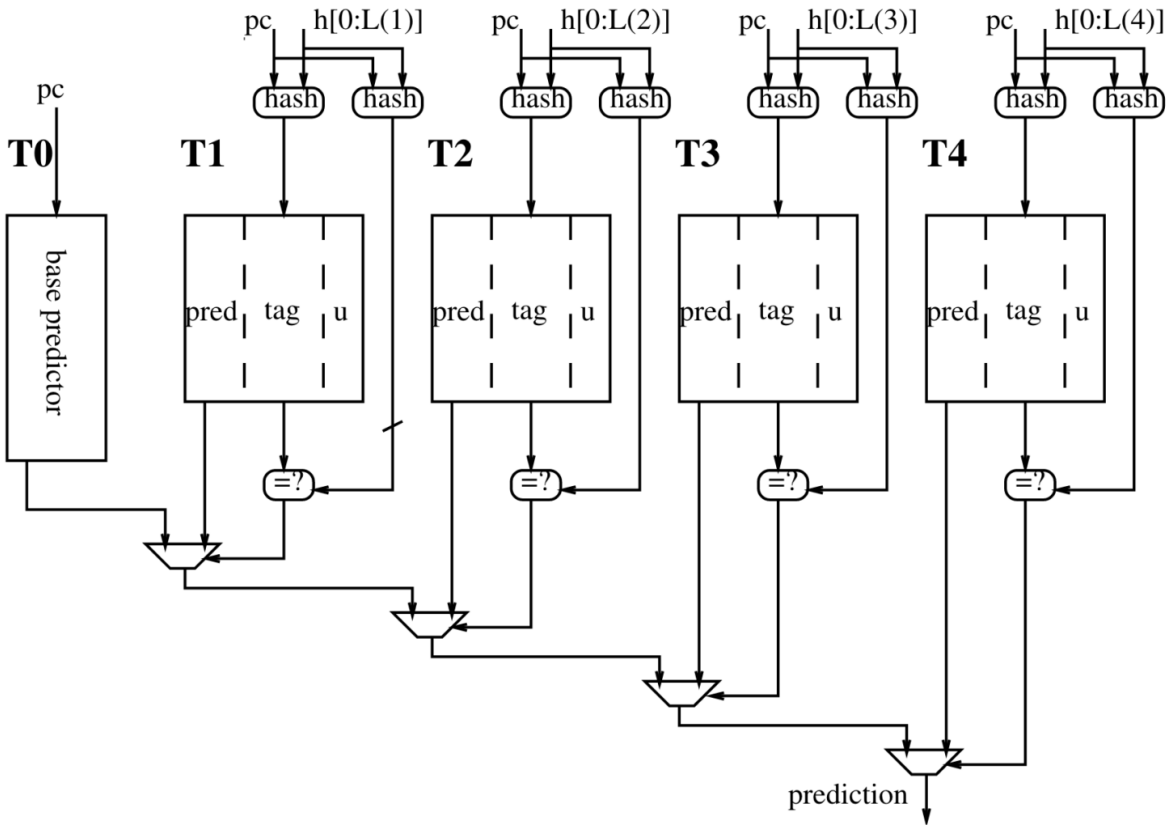


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

现代的分支预测器算法只有一种——TAGE

几乎所有的学术研究和商业高性能处理器都使用 TAGE 或者 TAGE 的变种

短历史预测简单分支
长历史预测复杂分支

Branch prediction research is basically about separating easier and more difficult branches, and using a simple predictor for the easy branches and a complex predictor for the difficult ones.
— Onur Mutlu (Computer Architecture and Digital Design, ETHZ)

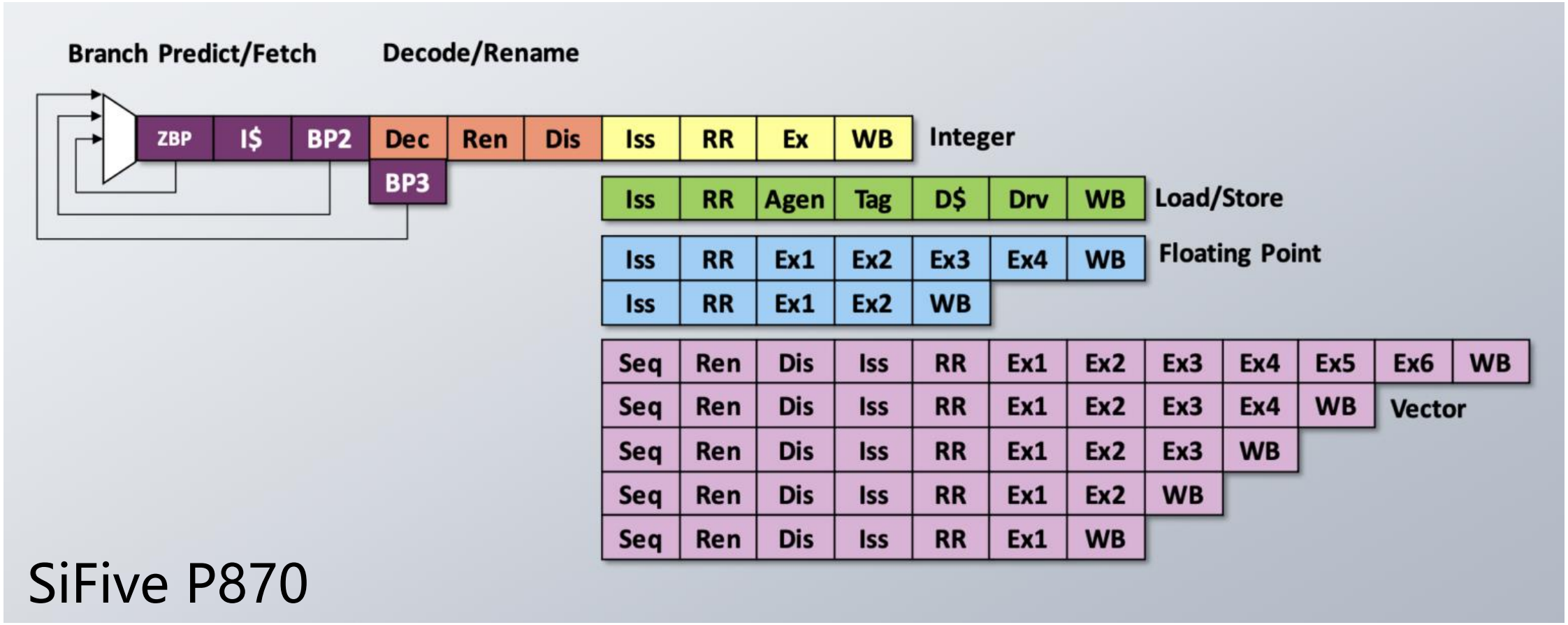
André Seznec, Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 2006, 8, pp.23.

3.3 分支预测

覆盖预测器

使用更强的预测器**覆盖**简单预测器的结果

- 像 TAGE 这种复杂程度的预测器是**没有可能**当拍就能得到预测结果



3.4 小结

- 取指 => 译码 => 发射 => 执行 => 写回
- 五级流水线 is all Lab1 need
- 流水线 = 数据通路 + 控制