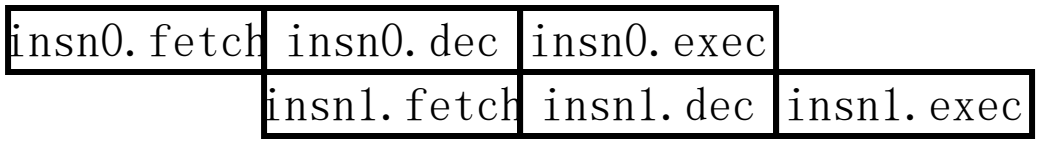
计算机体系结构

期末复习

简单5级流水线
(回顾)

流水线 (pipeline)

流水线



口 核心思想

- **提升指令吞吐量，不是单个指令执行时间**

口 将指令执行划分成多个阶段

- 当指令从第1阶段进入到第2阶段, 下一条指令进入第1阶段
- 每条指令顺序经过所有阶段
- 所有指令经过的阶段数相同
- 并行方式: “指令-阶段 并行”

+相比单周期CPU，指令进入/离开的频率更快

关于流水线的时钟周期

为什么流水线的时钟周期 > (单周期的时钟周期) / (流水段个数)?

- **三个原因**
  - ✓ 每个流水段都增加了寄存器，增加了延迟
  - ✓ 每个流水段的时间长度不同，时钟周期按照最长的计算
  - ✓ 流水线还增加了其他线路（前递旁路等）

***这些因素对于理想的流水线阶段数量有重要影响，随着流水段的级数增加，时钟频率的增益会减少（overhead更高）**

关于流水线的CPI

为什么流水线的CPI > 1?

- 对于简单流水线（scalar in-order pipeline），**CPI = 1 + stall惩罚**
- Stalls是为了解决流水线中的危害（hazards）
  - **Hazard:** 影响流水线正确执行的一些危害（数据相关或者控制相关）
  - **Stall:** 为了让流水线正确执行引发的停顿

指令相关性及危害
(Dependence and Hazards)

指令相关性及危害

- **指令相关性**: 两条指令之间存在关联关系
  - ✓ **数据相关性**: 两条指令使用相同的存储位置
    - 读后写(WAR)，写后读(RAW)，写后写(WRW)
  - ✓ **控制相关性**: 一条指令影响另一条指令是否能够执行
    - 分支语句，中断
  - ✓ **结构相关**：两条指令使用同一种流水线资源

- **危害 (Hazard)**: 有些相关性会导致指令执行错误
  - ✓ 例：一条指令以前一条指令的结果作为操作数，但前一条指令的结果尚未写入寄存器或内存，导致结果错误

数据相关性

数据相关类型

- 写后读 (**RAW, 真相关**)
- 写后写 (**WAW**)
- 读后写 (**WAR**)

数据相关性

哪些数据相关性会导致流水线停顿?

- ✓ “写后读”是对某个值的真正依赖，必须解决
- ✓ “写后写”和“读后写”是因为寄存器数量不够引起的
  - 它们依赖某个寄存器“名字”，而不是一个值
  - 后面会讨论如何解决

存储层次基本概念
(Memory Hierarchy)

内存层次 (Memory Hierarchy)

- **目标：既快又大的内存**

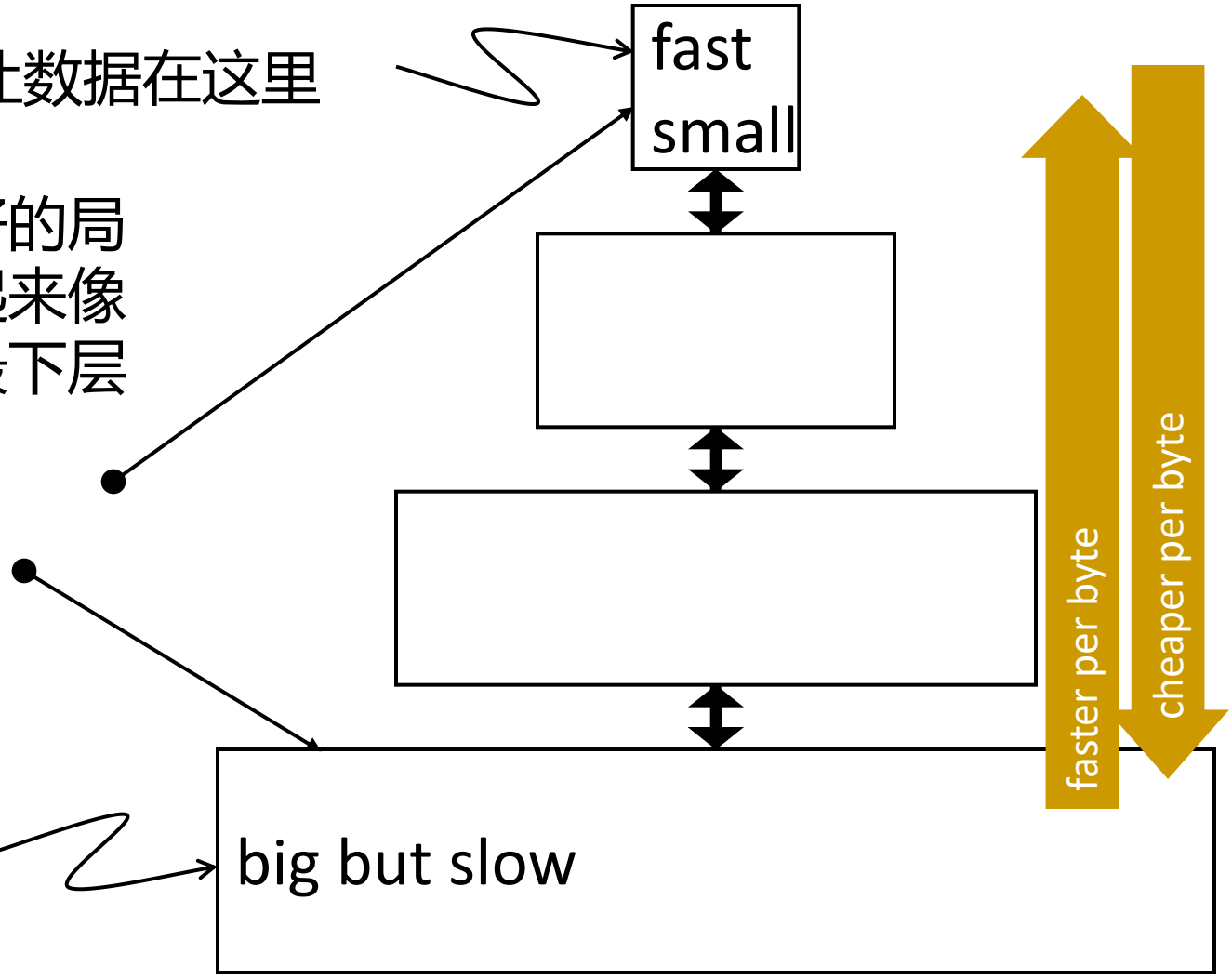
- **采用单层的存储结构很难实现**

- **思路: 采用多层次存储结构**
  - 离处理器越远的层级空间越大、速度越慢
  - 确保处理器需要的大部分数据都保存在速度较快的层级中

内存层次 (Memory Hierarchy)

尽量让数据在这里

当数据访问有很好的局部性时，内存看起来像最上层一样快、最下层一样大



在这里保存所有数据

内存局部性

- 一个典型程序的内存访问会表现出很多局部性
  - 一般程序都有很多“循环”

- **时间局部性:** 一个程序倾向于在一个小的时间窗口内多次访问相同的内存位置

- **空间局部性:** 一个程序倾向于一次访问一片相近的内存位置
  - 1. 指令内存的访问（指令的地址通常是连续的）
  - 2. 数组访问

时间和空间局部性

- 哪里表现出空间局部性?
- 哪里表现出时间局部性?

```
int sum = 0;
int X[1000];

for(int c = 0; c < 1000; c++){
   sum += X[c];
}
```

Caching: 利用时间局部性

- 思路: 将最近访问的数据存放在访问速度快的地方(cache)
- 期望：数据很快会被再次访问

Caching: 利用空间局部性

- **思路: 将与近期访问的数据地址相邻的数据放在访问速度快的地方**
  - 逻辑上将内存划分成大小相等的块(blocks)
  - 访问某个数据，就将数据所在的块整个缓存

- **期望: 相邻的数据很快会被访问**

Cache

Cache 基本概念

- **Block (or Cache Line): Cache中的最小数据单元**
  - Cache被划分为若干个blocks，每个block可以存储从主存中复制过来的一部分数据
  - 主存在逻辑上也被划分为blocks，每个block映射到cache中的一个位置
  - 数据以block为单位进行cache相关操作
- **内存访问时:**
  - HIT: 如果在cache中, 使用cache中的数据(不必再访问内存)
  - MISS: 如果不在cache中, 将数据从内存复制到cache
- **Cache设计中的一些重要决策:**
  - 放置(Placement): 将block放在cache的哪个位置?
  - 替换(Replacement): 为了腾出空间，移除哪些数据?
  - 写策略(Write policy): 如何处理写请求?

Cache寻址

- 内存逻辑上被分成固定大小的blocks
- 每个block映射到cache中的一个位置, 由映射算法根据内存地址中的index bits来决定
- **Cache 访问:**

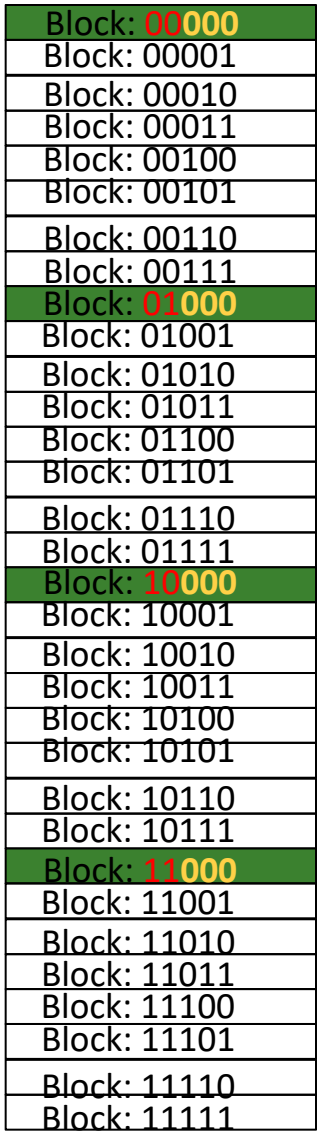
  1) 先根据内存地址中的index bits索引到指定的cache block(包含tag和data)

  2) 然后检查tag中的valid bit

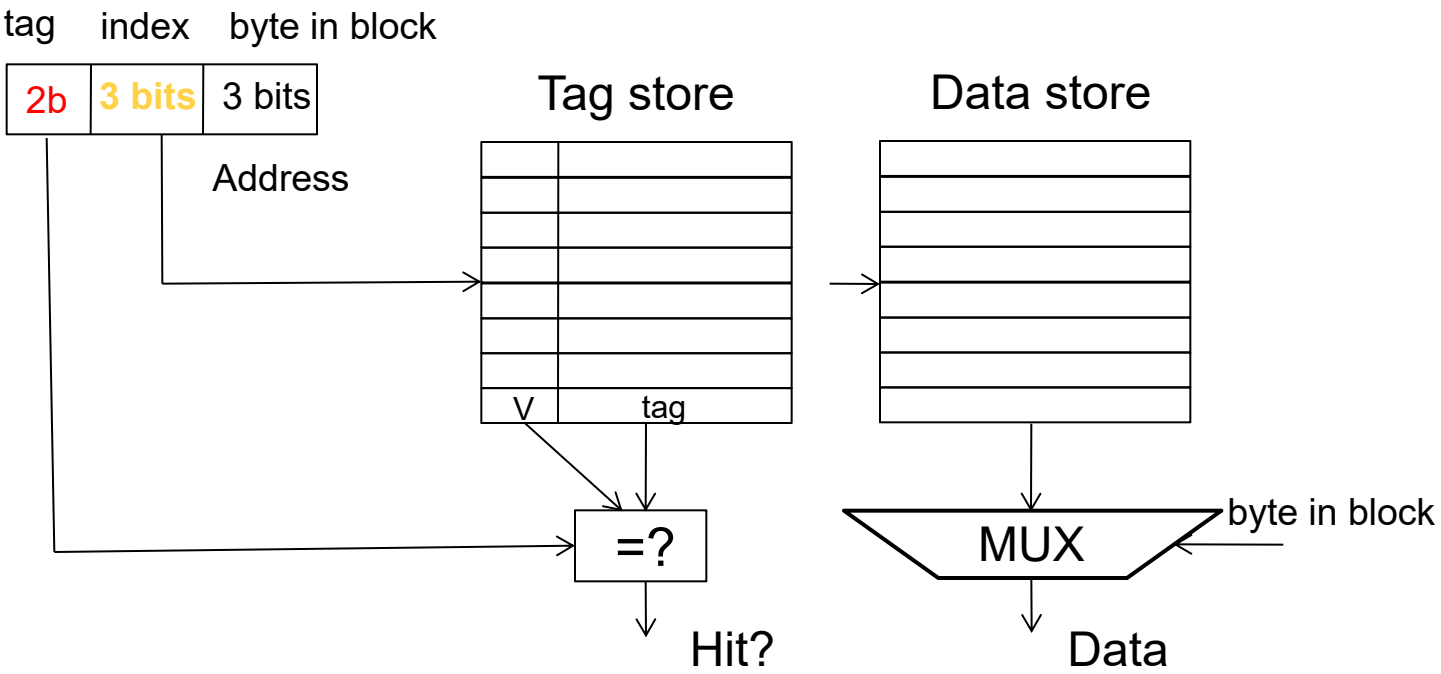
  3) 最后将内存地址中的tag bits和cache中的tag进行比较

- 如果一个block在cache中(cache hit), tag中的valid bit应该置位，并且tag应该与内存地址中的tag bits相同

放置策略：直接映射(Direct-Mapped)



- 假设内存是字节寻址: 大小为256 bytes, block大小为8 bytes，共分为32个blocks
- 假设Cache: 大小为64 bytes, 包含8个blocks
- 直接映射: 一个block只能映射到一个位置



Index相同的内存地址竞争相同的位置

- 导致冲突未命中(conflict misses)

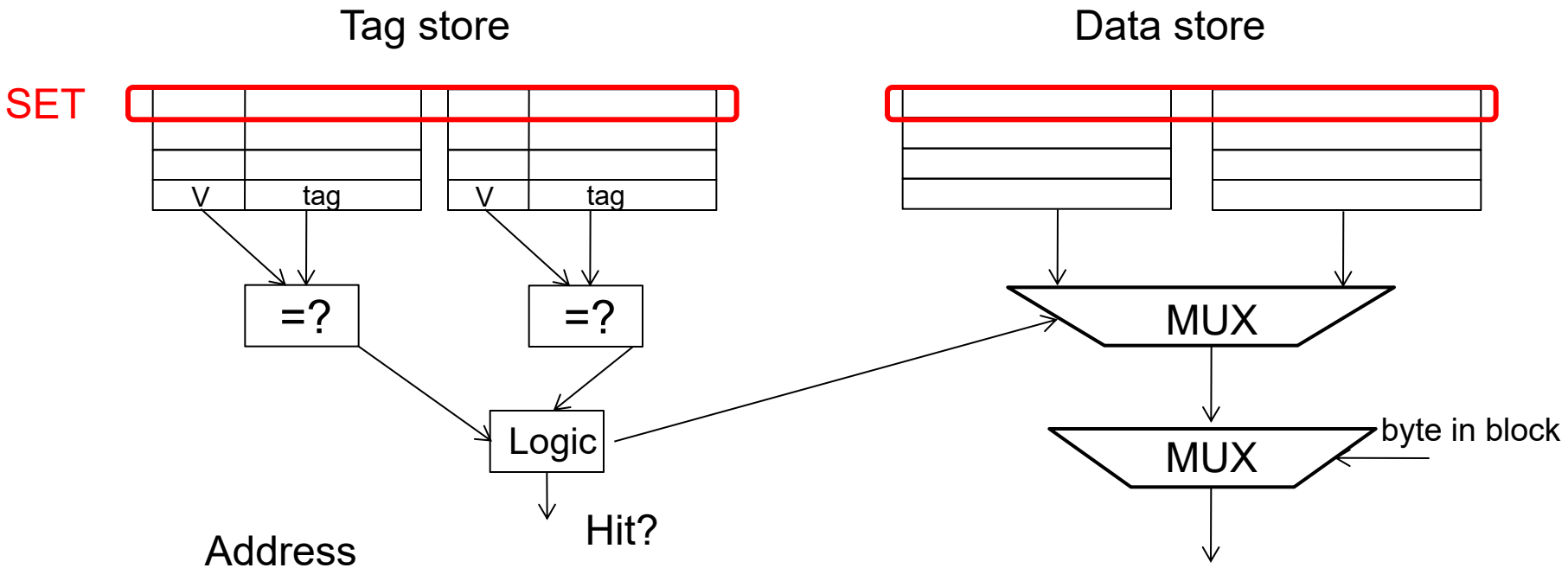
直接映射的 Cache

- **直接映射的 cache:** 内存中index相同的两个blocks不能同时在cache中出现
  - One index → one entry

- **如果交替访问index相同的两个blocks，会导致0%命中率**
  - 假设地址 A and B 有相同的index bits(但tag bits不同)
  - A, B, A, B, A, B, A, B, … → conflict in the cache index
  - 所有访问都是未命中

放置策略：组相联(Set Associativity)

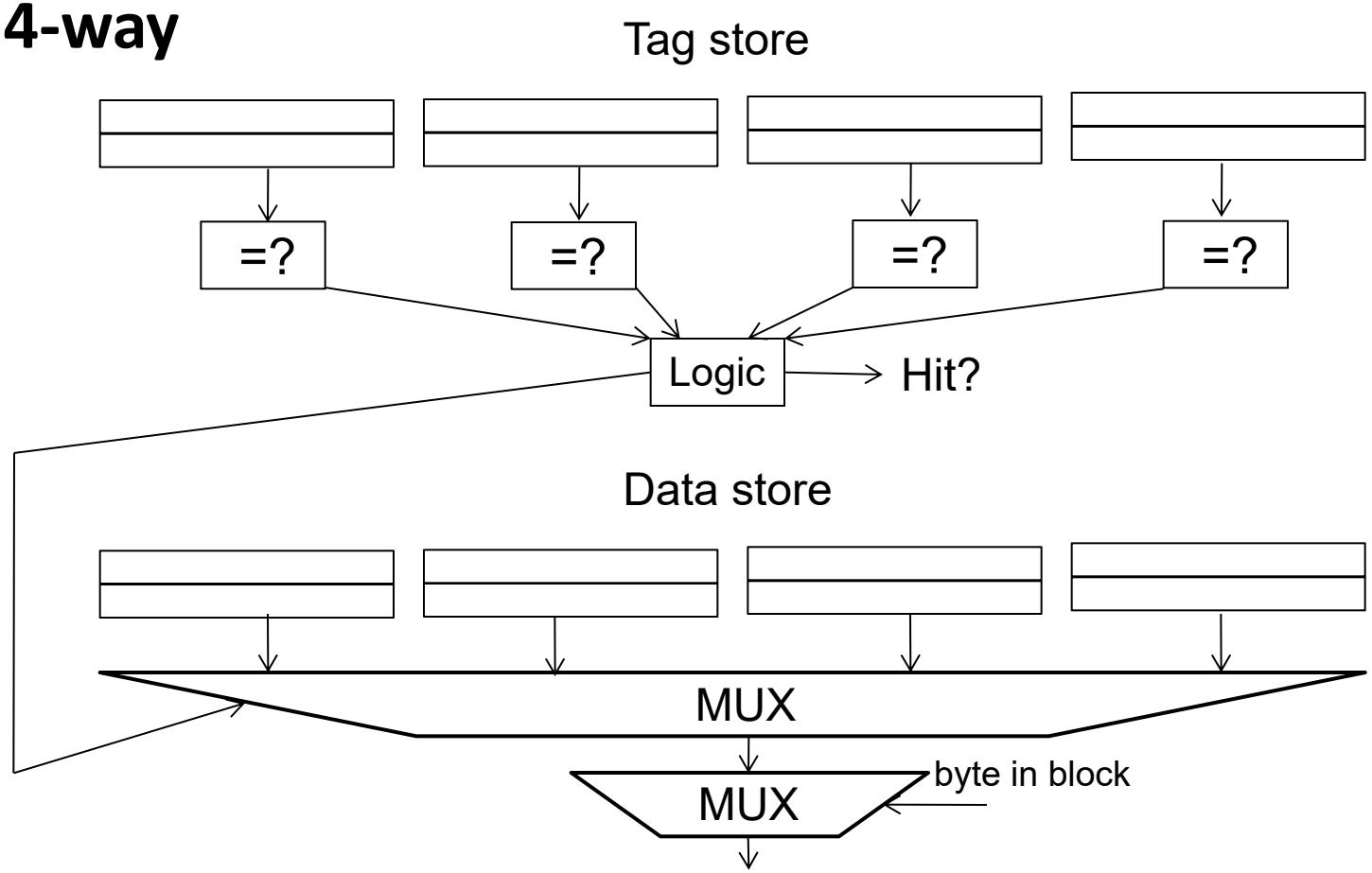
- 在直接映射中，地址 0 and 8 总是冲突
- 不是只有1列(包含8个blocks)，而是有2列(每列包含4个blocks)



Key idea: 根据index bits映射到set，set内的位置任选
+ 减少了冲突未命中
-- 更复杂, 访问更慢, tag更长(index bits变短了)

更高关联度

4-way

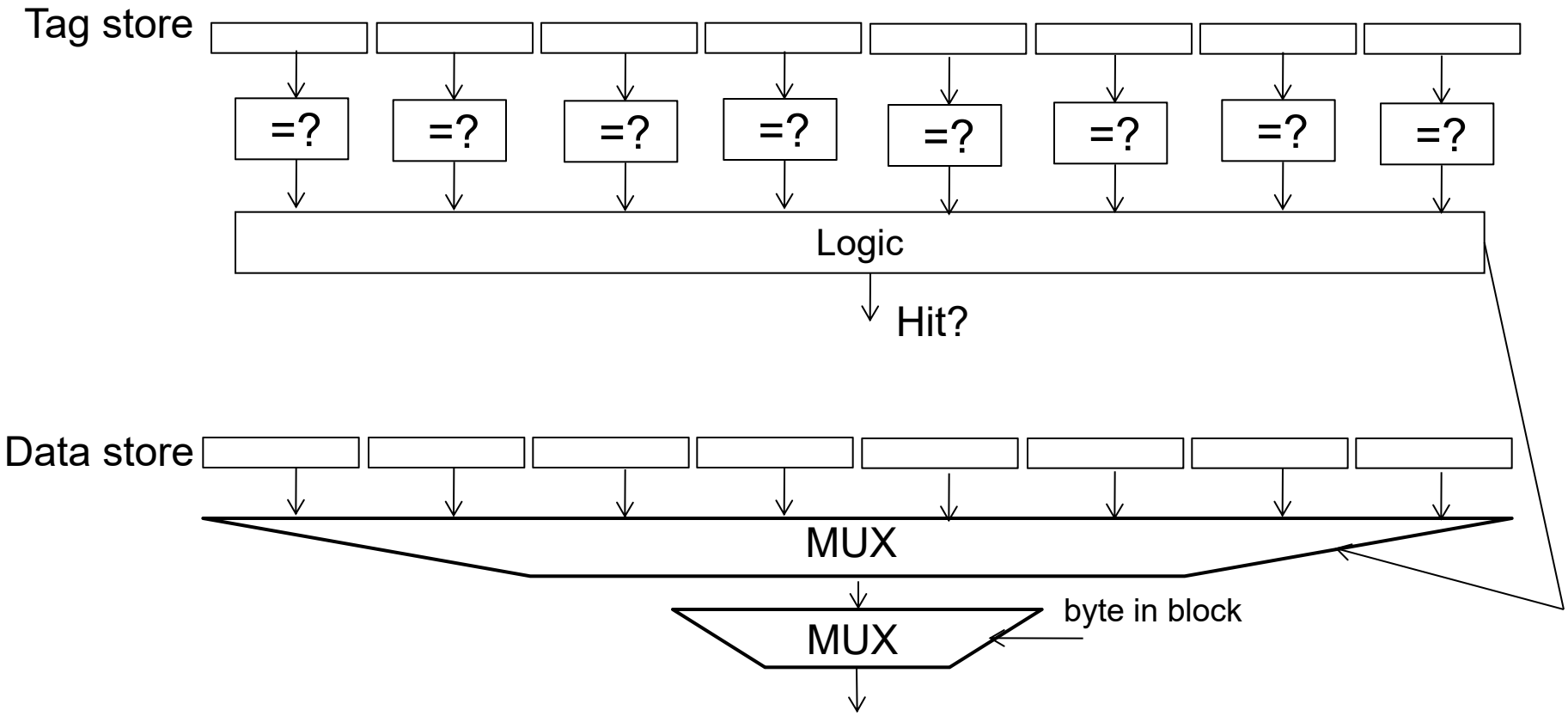


+ 冲突未命中更少了

-- tag比较器增加，data mux变宽，tags更长

放置策略：全相联(Full Associativity)

- 全相联
  - 1个block可以放在cache的任何位置

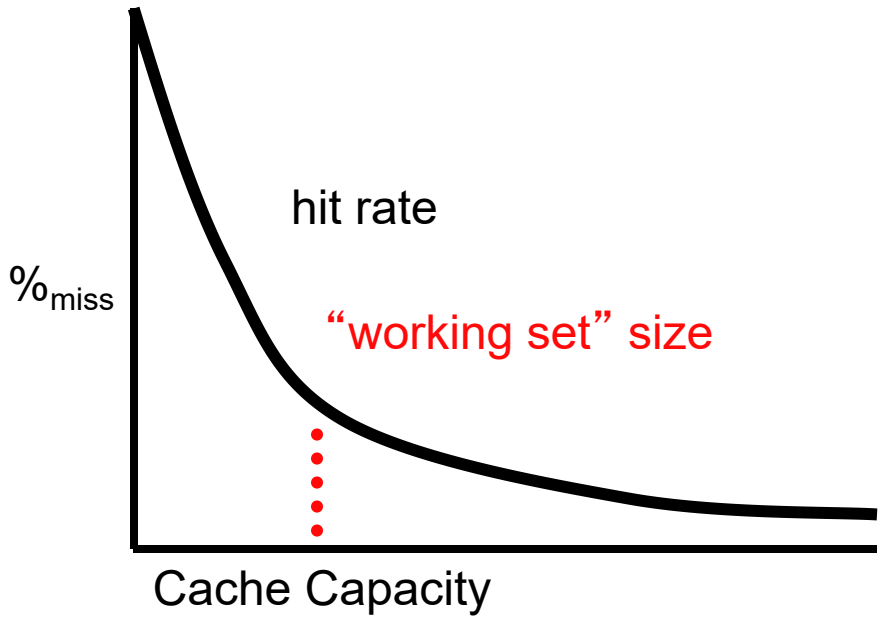


Cache主要参数的影响：Capacity

- **Cache size: 总容量**
  - 更大的缓存可以更好地利用时间局部性

- **Cache增大会增加访问延迟**
  - 越小越快 => 越大越慢
  - 访问延迟可能会增加关键路径的延迟

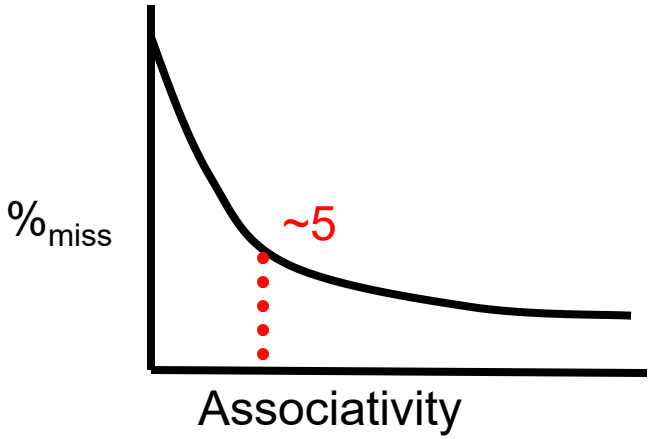
- **Cache太小**
  - 无法有效利用时间局部性
  - 有用的数据频繁被替换



Cache主要参数的影响：关联度

- **关联度:** 每个组(set)里能放几个blocks

- **关联度更高**
  - ++ 命中率越高
  - -- 访问时间更长
  - -- 硬件更复杂 (更多比较器)

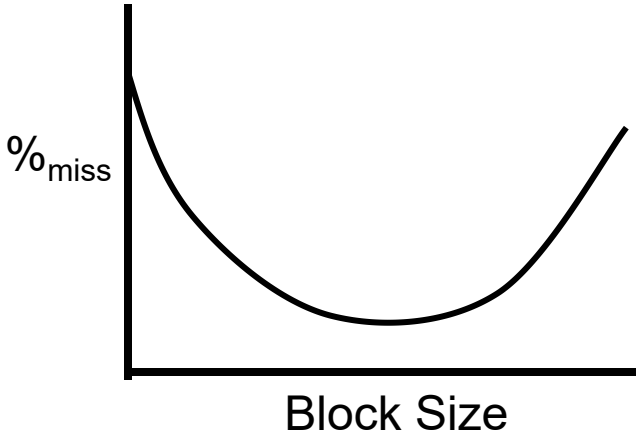


- **随着关联度增加，命中率增长趋缓**

Cache主要参数的影响：Block Size

□ 增大block size有正反两方面影响

+ **空间预取 (正面)**
  - 地址相邻的blocks被提前预取
  - 有利于降低miss rate
- **产生干扰 (反面)**
  - 能放入cache的blocks变少了
  - 可能增加miss rate
  - 特殊情况：全局只有1个block



□ 两方面影响同时存在

- 具体哪个影响大取决于workload

替换策略(Replacement Policy)

- **Cache miss发生时，set中的哪个block将被替换?**
  - 优先替换invalid block (还没放置任何数据)
  - 如果都是valid, 由替换策略决定
    - ✓ **Random**
    - ✓ **FIFO (first-in first-out)**
    - ✓ **LRU (least recently used)**
      - 适合空间局部性 (未来最不可能用到的block)
    - ✓ **NMRU (not most recently used)**
      - LRU的近似策略，更容易硬件实现
      - 2路组相联(2-way set assoc.)时就是LRU
    - ✓ **Belady's**: 替换最晚用到的block
      - 最优策略，但无法实现

写操作如何传递（Cache Hit)

- **何时将新数据传递到更下层的cache或memory?**
- **Option #1: 写穿透(Write-through): 立即传递**
  - Cache命中, 更新cache
  - 立即向下一层写
- **Option #2: 写返回(Write-back): 当block被替换时**
  - 同一个block在不同层的cache和memory中有不同的版本!
  - 每个block需要一个额外的 “**dirty**” 位
    - 替换 **clean** block: 不需要额外操作
      - 这个block只有1个版本
    - 替换 **dirty** block: 需要向下一层cache写
      - 这个 dirty block 是最新版本

Cache写策略对比

写穿透(Write-through)

- 需要额外带宽
  - 考虑反复写命中的情况
- 下层cache需要处理小的写请求 (1, 2, 4, 8-bytes)
+ 不需要dirty bits
+ 不需要处理“写返回”操作
- 常用在GPUs, 因为GPU程序空间局部性低

写返回(Write-back)

+ 优点: 带宽使用少
- 和“写穿透”的优缺点相反
- 常用在CPU中

处理Write Miss

- **写分配(Write-allocate)**: 先从下一层cache将数据读上来，然后再写
  - \+ 减少了读操作的misses (下一次读该block就能命中)
  - – 需要额外的带宽
  - • 很常用 (尤其是和“写返回”搭配使用)

- **写不分配(Write-non-allocate)**: 直接往下一层写，不将数据读入本层cache
  - – 潜在地增加了读操作的misses
  - \+ 使用较少的带宽
  - • 通常和“写穿透”搭配使用

提升Cache性能

提升Cache性能

程序执行时间 = 指令数 * CPI * 时钟周期
CPI (with cache) = CPI_base + CPI_cachepenalty
CPI_cachepenalty = ...

1. 降低 miss rate
2. 降低 miss penalty
3. 降低 hit latency

提升Cache性能

程序执行时间 = 指令数 * CPI * 时钟周期
CPI (with cache) = CPI_base + CPI_cachepenalty
CPI_cachepenalty = ...

1. 降低 miss rate
2. 降低 miss penalty
3. 降低 hit latency

Cache Miss 的三种情况 (3C)

- **Compulsory miss**
  - 第一次访问一个block肯定会发生miss
  - 后续对该block的访问应该命中，除非block因为下面的原因被替换

- **Capacity miss**
  - cache空间不足以缓存所有的数据（小于内存）
  - 即使采用大小相等的全相联cache、以及最优的替换算法，仍然会发生的那些miss

- **Conflict miss**
  - 既不是compulsory miss，也不是capacity miss的那些miss

1. 增大Cache Size

- 可以降低conflict miss和capacity miss

	增大cache size	增加关联度	增加block size
Compulsory misses	=		
Capacity misses	↓		
Conflict misses	↓		
Hit latency	↑		
Miss latency	=		

缺点：

- 访问延迟增加
- 能耗更高

2. 增加关联度

- 可以降低conflict miss

	增大cache size	增加关联度	增加block size
Compulsory misses	=	=	
Capacity misses	↓	=	
Conflict misses	↓	↓	
Hit latency	↑	↑	
Miss latency	=	=	

缺点：

- 访问延迟增加

3. 增加block size?

- 可以降低compulsory miss

	增大cache size	增加关联度	增加block size
Compulsory misses	=	=	↓
Capacity misses	↓	=	↓
Conflict misses	↓	↓	?
Hit latency	↑	↑	=
Miss latency	=	=	↑

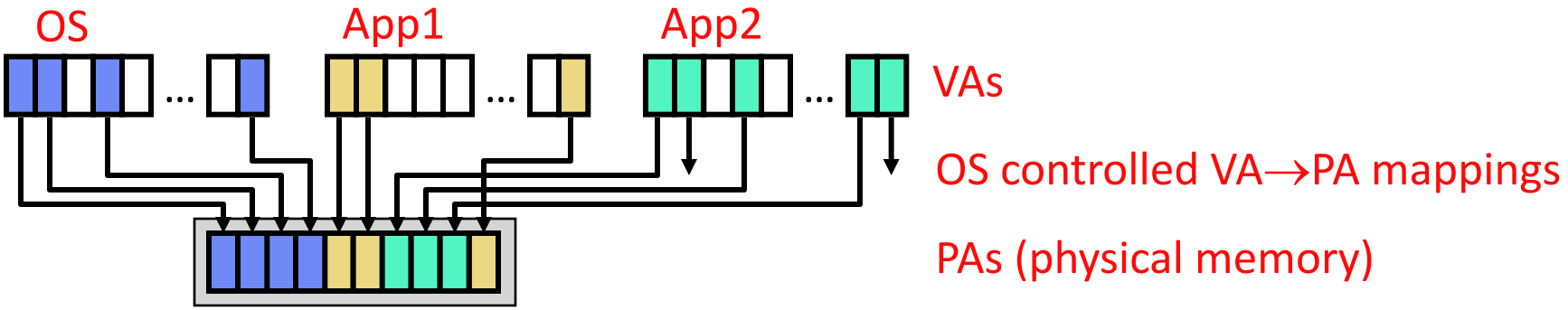
缺点：

- 如果访存空间局部性不高，会浪费cache空间

虚拟内存
(Virtual Memory)

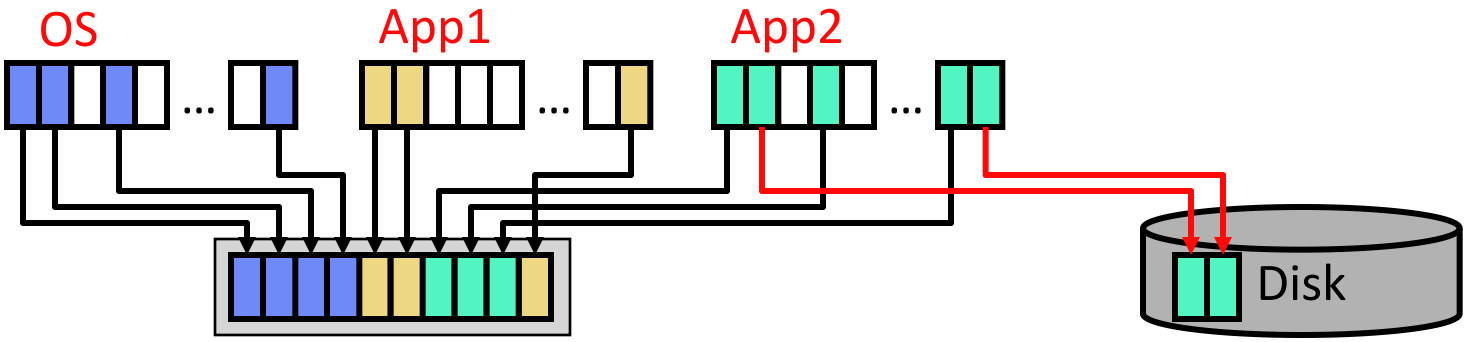
虚拟内存 (Virtual Memory)

- **虚拟内存 (VM)**:
  - Level of indirection
  - 应用产生的地址为虚拟地址 **(VAs)**
    - 每个进程以为自己有2^N 个字节的地址空间
  - 内存访问使用物理地址 **(PAs)**
  - 虚拟地址到物理地址的转换以page为单位(粗粒度)
  - 操作系统负责虚拟地址到物理地址的映射(页分配)
  - 逻辑上: 地址转换需要在每次取指令和访问内存数据之前进行
  - 实际上: 有很多硬件加速措施提高转换速度



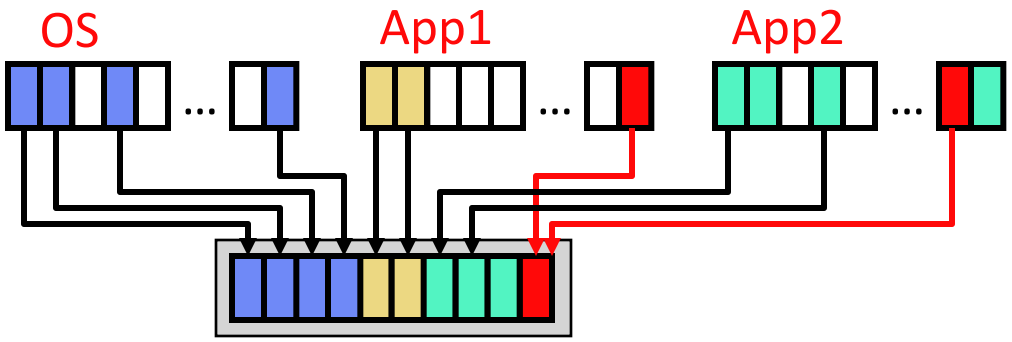
虚拟内存 (Virtual Memory)

- **程序使用虚拟地址 (VA)**
  - 虚拟地址位数为N bits (指针大小, 现在通常为64 bits)
- **内存使用物理地址 (PA)**
  - 物理地址位数为M bits, M通常小于N
  - 2^M 表示能支持最大的物理内存
  - 现代计算机一般采用48 bits, Intel/AMD正在尝试56 bits
- **虚拟地址→物理地址转换以page为粒度**
  - 映射不需要保持物理页连续
  - 虚拟页可能没有映射到任何物理页
  - 没有映射的虚拟页可能在磁盘上(换出去了)或者还未被使用过

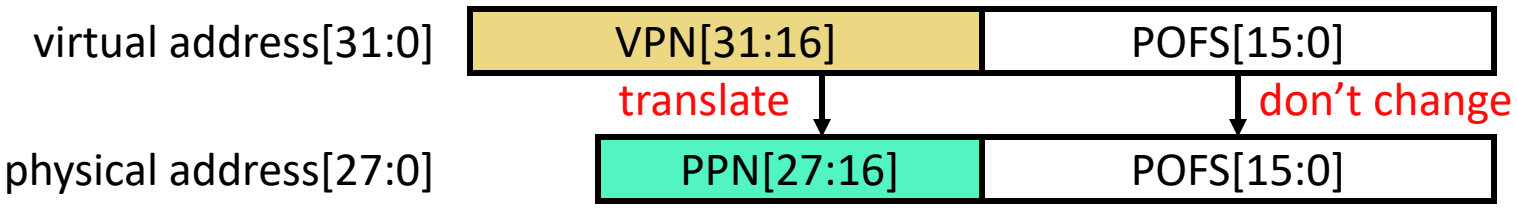


虚拟内存的好处

- **多程序之间隔离**
  - 每个程序以为自己有2^N大小的内存空间
  - 防止程序互相访问彼此的内存空间
    - 无法知道其他程序的物理地址!
- **安全保护**
  - 每个页有读/写/执行的权限 (OS设置)
  - 由硬件保证
- **进程之间通信**
  - 将同一个物理页映射到多个虚拟地址空间
  - 或者通过UNIX **mmap()**共享文件



地址转换

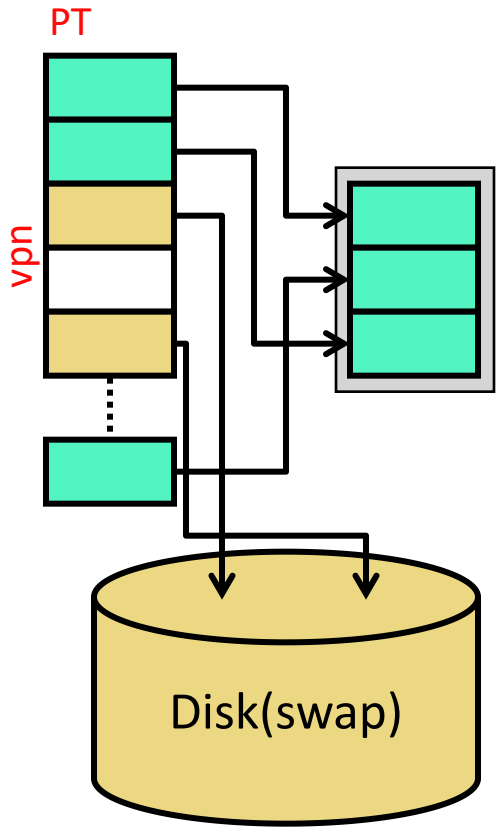


- **虚地址到物理地址的映射称为地址转换**
  - 将虚地址分成**虚拟页号 (VPN)** & **页内偏移 (page offset)**
  - 将虚拟页号转换为**物理页号 (PPN)**
  - 页内偏移不需要转换

- **例如**
  - 页大小为64KB → 16-bit 页内偏移
  - 32-bit计算机 → 32-bit虚拟地址 → 16-bit 虚拟页号
  - 最大256MB物理内存 → 28-bit 物理地址 → 12-bit 物理页号

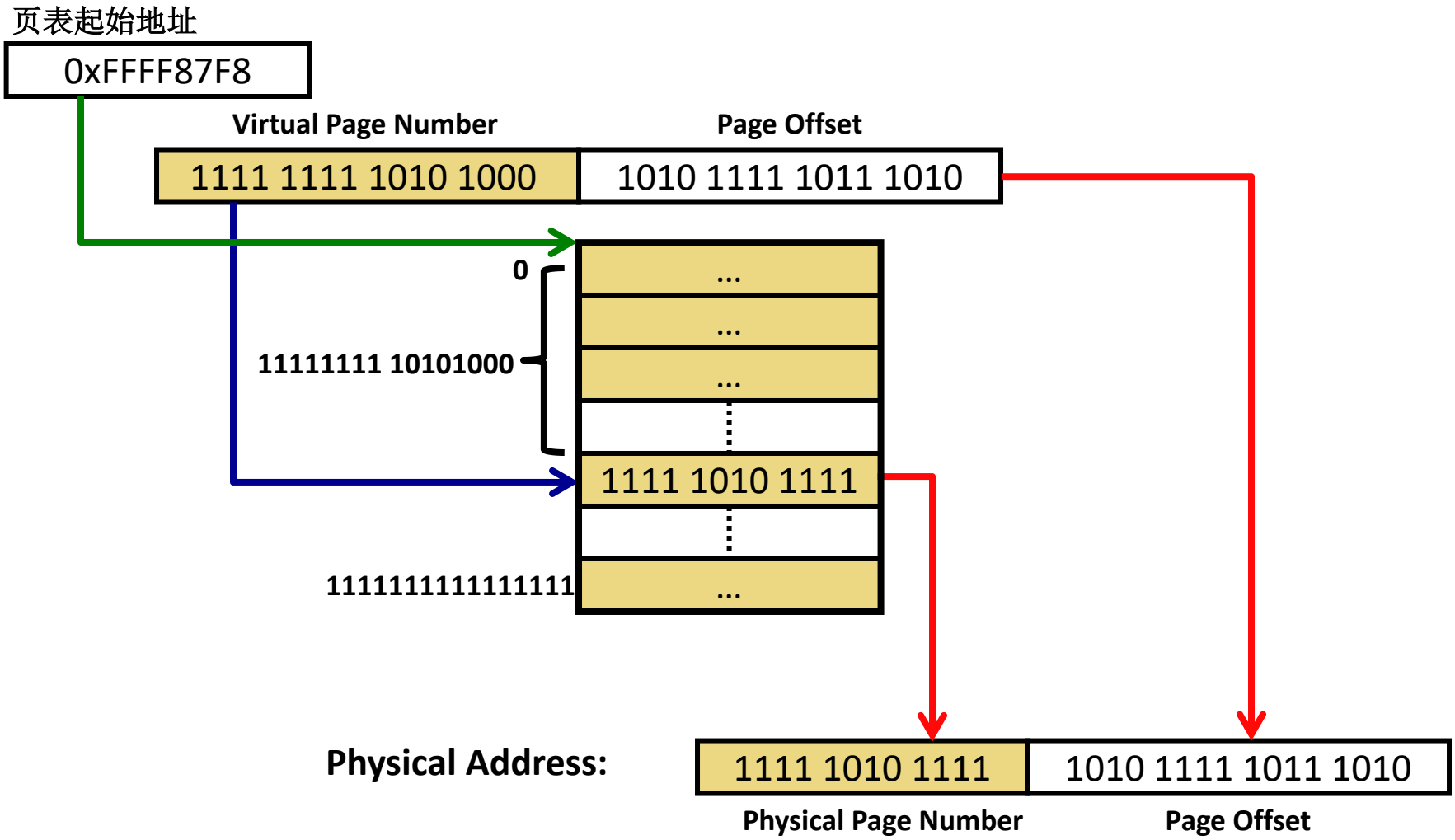
地址转换机制 I

- **地址如何转换?**
  - 软硬件协同完成
- **每个进程有一个页表(page table)**
  - **由操作系统维护的数据结构**
  - 将虚拟页映射到物理页或磁盘地址
    - 如果页从未被访问过, 虚拟页对应的项为空
  - 地址翻译就是查页表



页表示例

Example: 内存访问地址为 0xFFA8AFBA



页表大小

- **以下计算机的页表大小如何计算?**
  - 32-bit 计算机
  - 4B 页表项大小 (PTEs)
  - 4KB 页大小

  - 32-bit 计算机 → 32-bit 虚拟地址 → 2^32 = 4GB 虚拟内存
  - 4GB 虚拟内存 / 4KB 页大小 → 1M 虚拟页
  - 1M 虚拟页 * 4 Bytes/每页表项 → 4MB

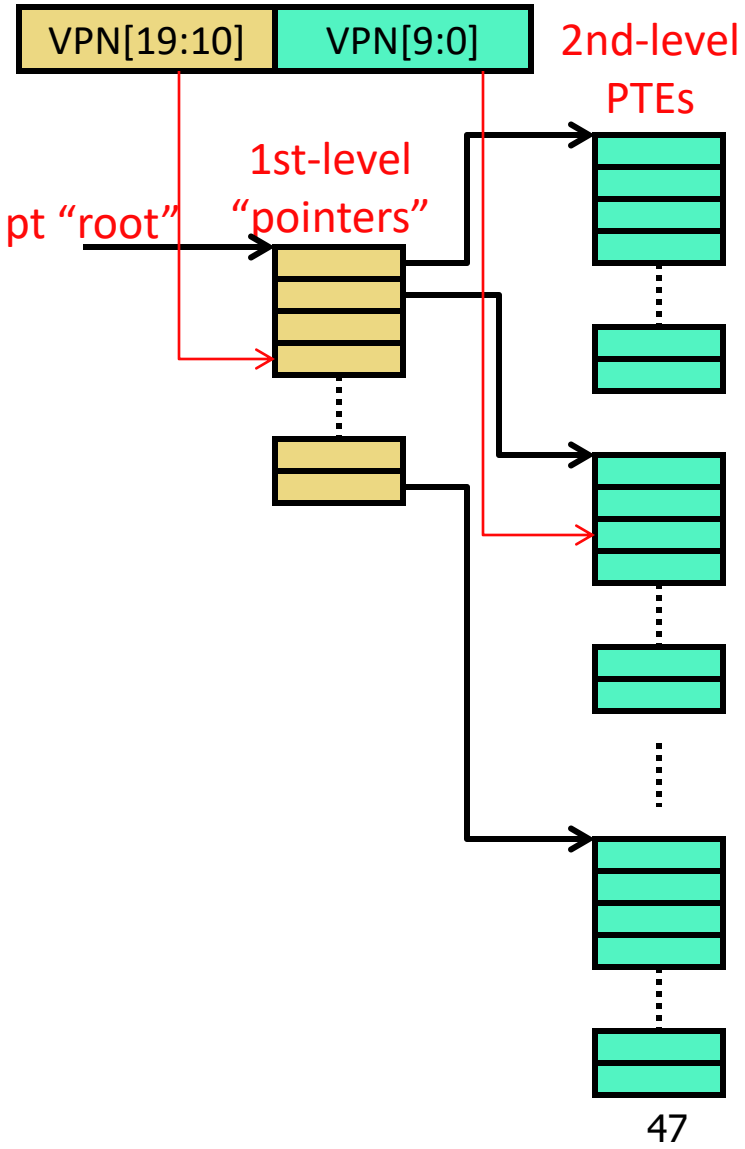
- **如果页大小为 64KB 呢?**
- **如果是 64-bit 计算机呢?**

页表可能很大

多级页表

多级页表

- 将页表变为树形结构
- 最下层保存页表项(PTEs)
- 上层页表保存到下层的指针
- 虚拟页号的不同部分用于索引不同的层

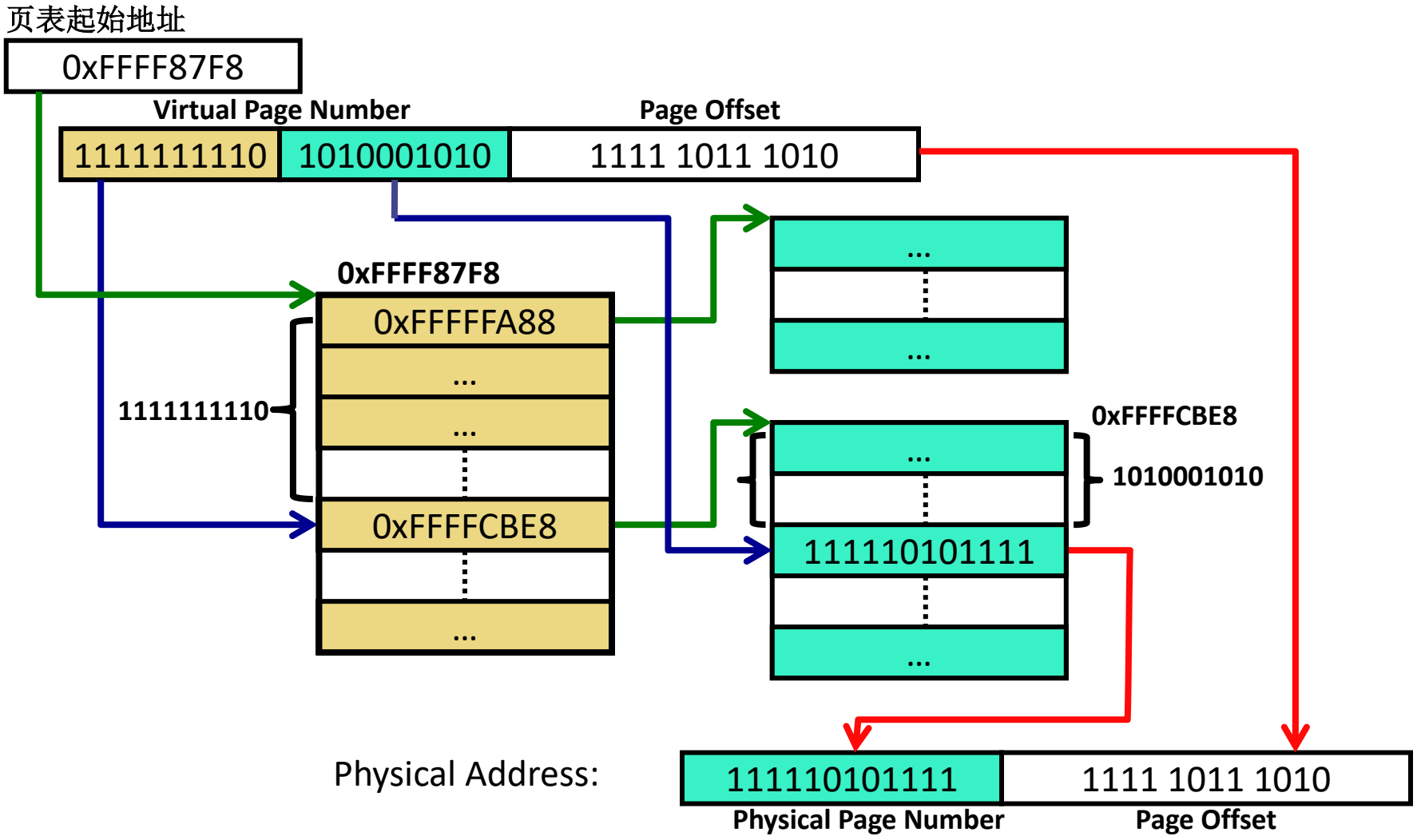


20-bit 虚拟页号

- 高位10 bits索引第1层
- 低位10 bits索引第2层
- 实际中, 通常大于2层

多级地址转换

Example: 内存访问地址 0xFFA8AFBA



页大小

很多ISAs支持多种页大小

- x86: 4KB, 2MB, 1GB

大页的优缺点

+ 减少页表空间
  - 表项变少了
+ 页表层数更少
  - 查找速度快
– 页内空间浪费更严重
  - 分配 2MB 大小的页但只用了 5KB
– 实现更复杂
  - OS 会寻求其他措施提高page利用率

地址转换

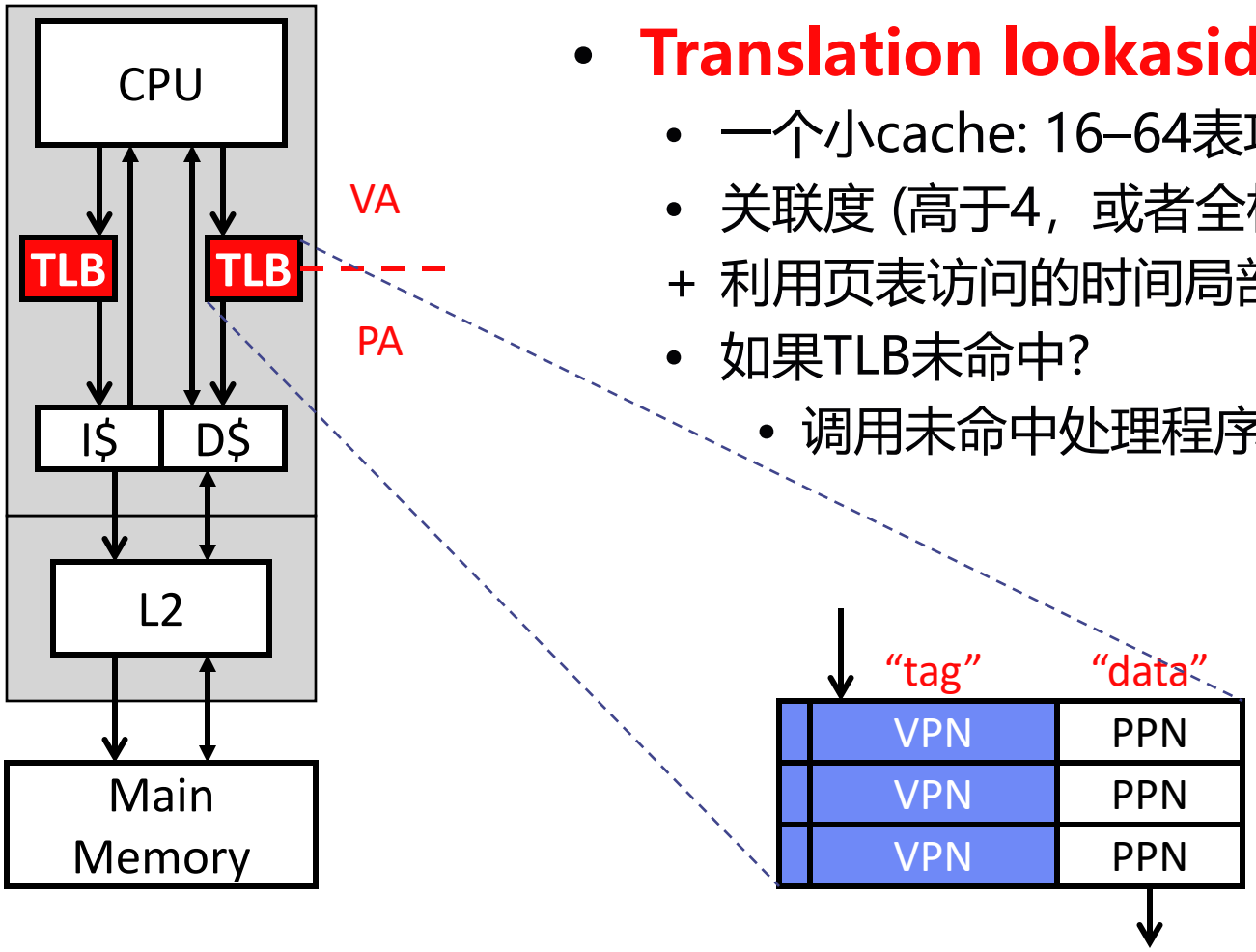
概念上

- 地址转换需要在每一次内存访问(cache访问)之前进行
- 每次转换都需要访问页表
  - 效率非常低

现实中

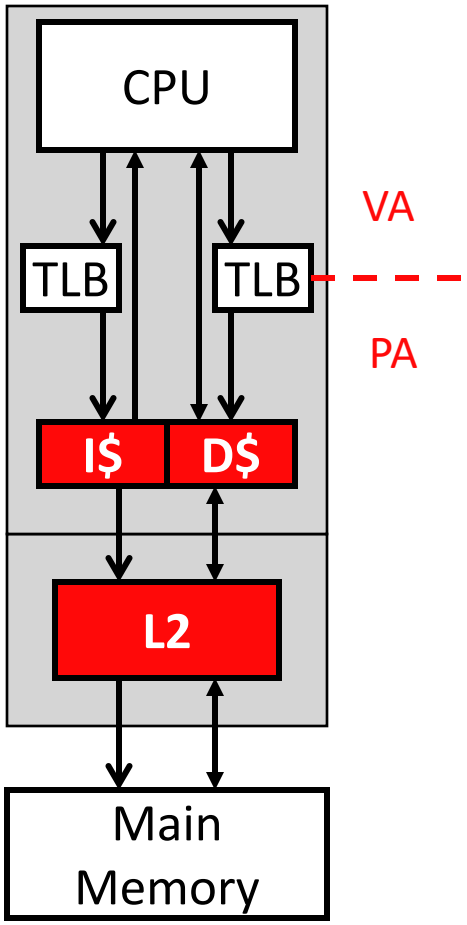
- **Translation Lookaside Buffer (TLB)**: 缓存转换结果
- 只有在TLB未命中的时候才去访问页表

TLB



- **Translation lookaside buffer (TLB)**
  - 一个小cache: 16–64表项
  - 关联度 (高于4，或者全相联)
  + 利用页表访问的时间局部性
  - 如果TLB未命中?
    - 调用未命中处理程序, 遍历页表

TLB & Cache



Caches

- 物理地址作为索引和tag
+ 好处
  - 自然实现隔离(不同任务的物理地址不同)
  - 上下文切换(context switches)无需flush
+ 可以实现基于cache的进程间通信
  - 一个cache block可由多个进程共享
– 慢: 即使TLB命中仍需要增加一个周期

TLBs

- 虚拟地址作为index和tag
- 上下文切换(context switches)需要flush

TLB Miss

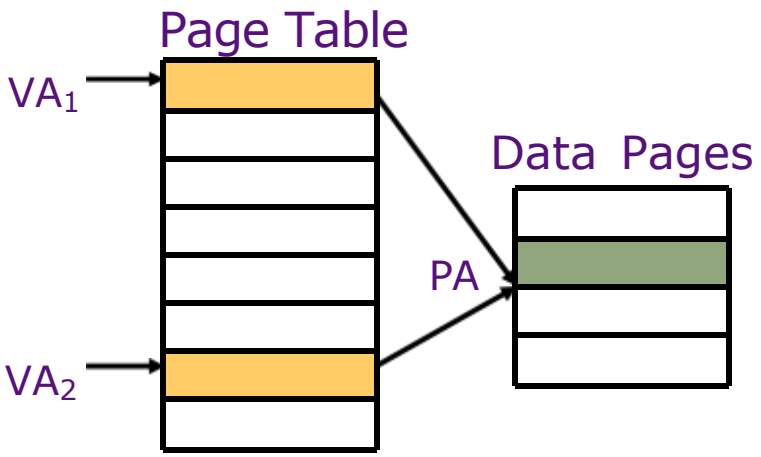
- **TLB miss: 地址转换结果不在TLB, 在页表**
  - 两种处理方式，都相对比较快
- **基于硬件的方式**: e.g., x86, recent SPARC, ARM
  - 页表的基地址由寄存器保存，硬件负责遍历页表取回数据
  + 延迟: 避免了操作系统程序 (avoids pipeline flush)
  – 页表格式必须是硬编码的
- **基于软件的方式**: e.g., Alpha, MIPS
  + 采用简短的（大约10条指令）操作系统程序遍历页表并更新TLB
  + 页表格式可以灵活定义
  – 延迟: 读了1-2次内存访问 + OS call (pipeline flush)
- **当前基于硬件的方式更受欢迎**

缺页异常(Page Faults)

- **Page fault: PTE访问页不在TLB和页表中**
  - 页不在主存中(可能在磁盘，或还没有分配)
  - 如果页还没有分配 → segmentation fault

- **OS 要做的事:**
  - 选择一个要替换掉的物理页
  - 如果要替换的页是“脏”页, 需要先写回磁盘
  - 从磁盘读入缺失的页
    - ✓延迟很长 (~10ms), OS 调度其他任务
  - 更新页表, 刷新TLBs, 重试内存访问操作

虚地址cache引起的Aliasing



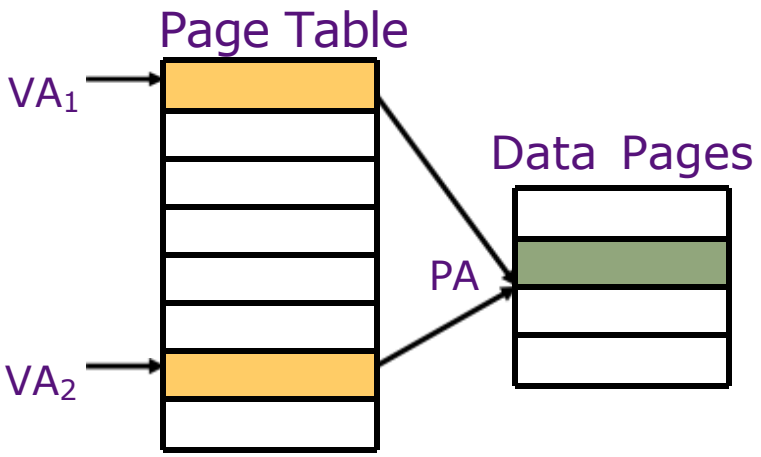
两个虚拟地址共享
同一个物理页

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

- 导致同一块数据在cache中有两个副本
- 写其中一个副本另外一个副本不知道，产生不一致!

因为cache按照虚地址访问，访问VA1时会将数据块放入cache的某个位置，访问VA2时又会将同一个数据块放入cache的另外一个位置，所以同一个数据块有两个copy

虚地址cache引起的Aliasing



两个虚拟地址共享
同一个物理页

Tag	Data
VA_1	1st Copy of Data at PA
VA_2	2nd Copy of Data at PA

- 导致同一块数据在cache中有两个副本
- 写其中一个副本另外一个副本不知道，产生不一致!

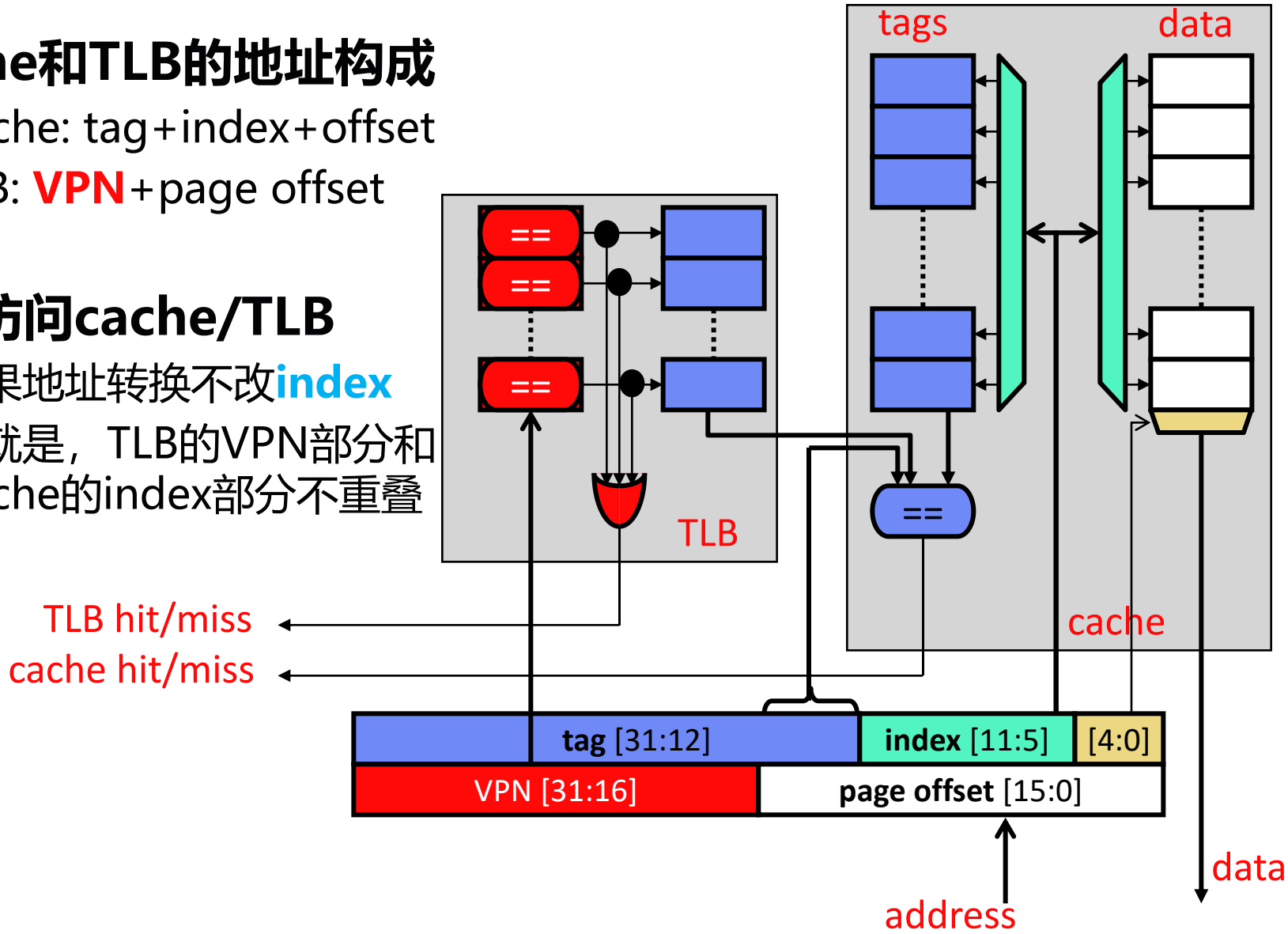
常用解决方案：不允许多个副本同时存在

对于采用直接映射的cache

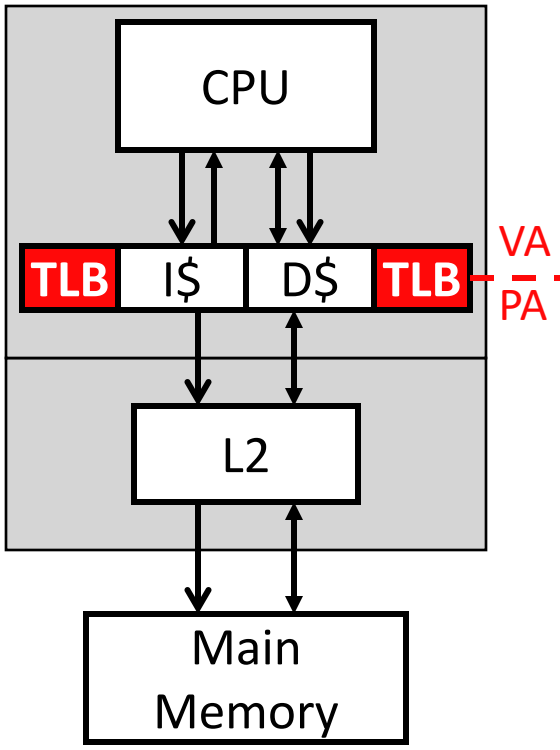
- 要求共享物理页的虚拟地址中的index bits必须相同，这样就会映射到同一个cache block，不会产生多个副本 (early SPARCs)

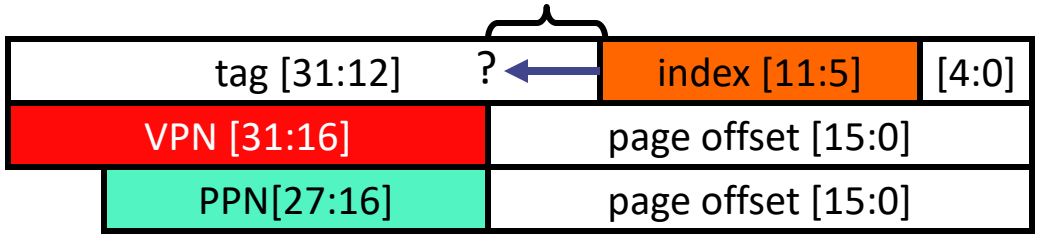
TLB & Cache 并行访问

- **Cache和TLB的地址构成**
  - Cache: tag+index+offset
  - TLB: **VPN**+page offset
- **并行访问cache/TLB**
  - 如果地址转换不改**index**
  - 也就是，TLB的VPN部分和Cache的index部分不重叠



TLB & Cache 并行访问





并行访问的条件

- **(cache size) / (associativity) ≤ page size**
- Index bits在物理地址和虚拟地址中相同!

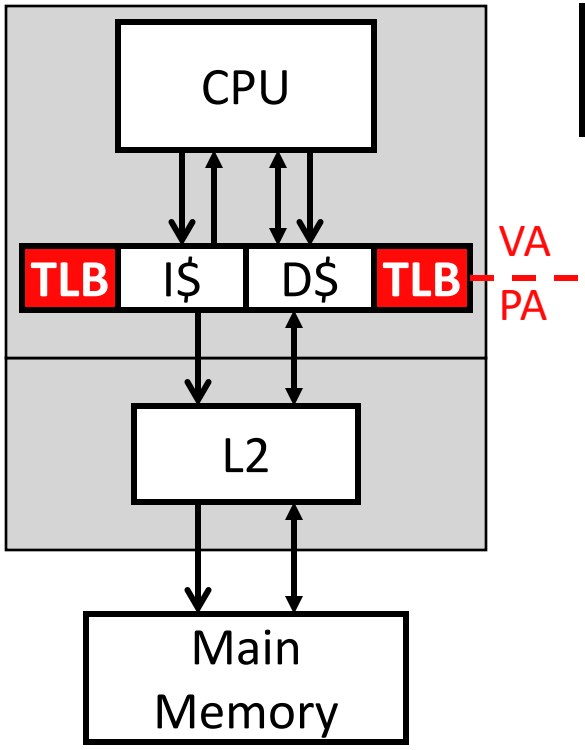
并行访问TLB和Cache

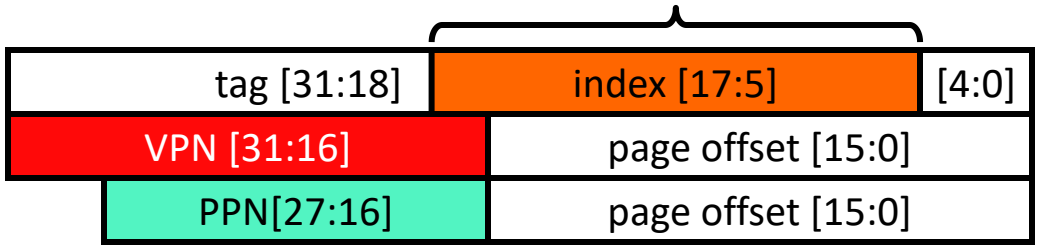
- Cache 访问在后期才需要比较tag
- \+ 快: 访问TLB时可以同时访问Cache(完成set定位)
- \+ 无上下文切换问题
- 当今主流做法

Index bits 限制了cache容量

- 如果index太长，会和VPN重叠
- 可以通过增加关联度缓解

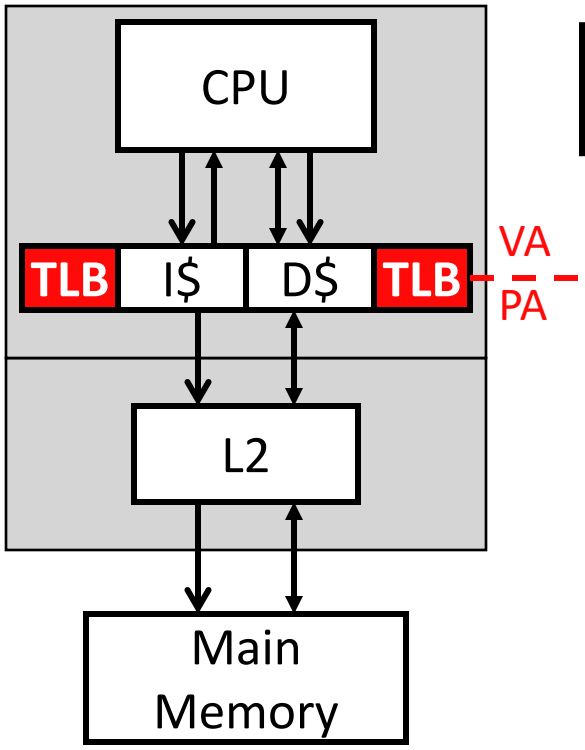
TLB & Cache 并行访问

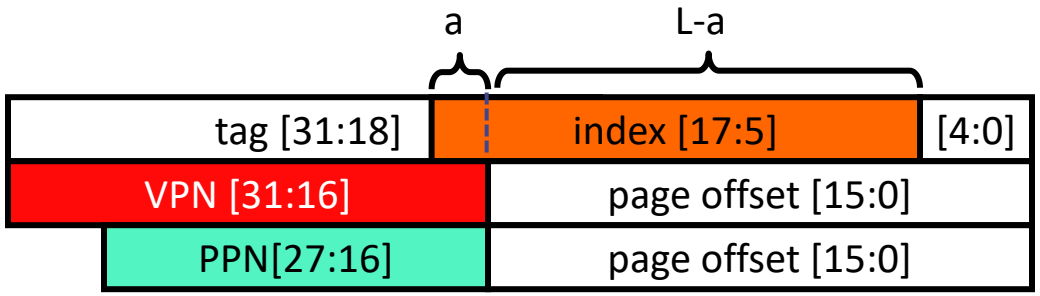




- **如果index太长?**
  - **(cache size) / (associativity) > page size**
  - Index bits在物理地址和虚拟地址中可能不同!
- **导致的问题 -> Aliasing**
  - 假设两个不同的虚拟地址V1, V2共享一个物理页(转换后的物理地址相同)
  - V1和V2的index bits可能不同，导致数据映射到cache不同的block(同一物理地址的数据在cache中有两个不同的副本)

TLB & Cache 并行访问





- **如果index太长?**
  - **(cache size) / (associativity) > page size**
  - Index bits在物理地址和虚拟地址中可能不同!
- **解决方案:virtual-index physical-tag cache**
  - cache同时查找2^a个block，index由两部分构成：低位有(L-a) bits，直接来自虚拟地址；高位有a bits，分别为$2^0,2^1,...,2^{a-1}$
  - TLB和cache并行访问
  - 转换成物理地址后，将物理地址tag和2^a个block中的tag都进行对比

乱序执行
(Tomasulo算法)

乱序执行 (Out of Order Execution)

也称为指令动态调度 (vs. 指令静态调度)

- 由硬件在运行时完成

简单描述:

1. 指令译码完成后，放入保留站（指令缓存空间）
2. 每个周期，检测保留站中每条指令的源操作数
3. 如果指令的源操作数都准备好了，发送指令到执行单元（如果有空闲）
4. 指令完成后，从保留站删除

乱序执行的好处:

- 当某条指令的执行时间很长时，允许后面的不想关指令先执行
- 有相关性的指令给无相关性的指令让路

乱序执行的条件

1. 需要为数据的生产方和消费方建立关联关系

- 寄存器重命名：每个数据都和一个“tag”关联

2. 需要缓存那些操作数还没有准备好的指令

- 放到保留站

3. 指令需要检测其操作数状态（是否准备好）

- 当数据准备好时，会广播其“tag”
- 所有指令将其tag与广播的tag进行比较，如果相同，表明数据准备好

4. 当所有操作数都准备好时，需要将指令发送到执行单元

- 当所有操作数准备好时，指令会被唤醒
- 如果多条指令同时唤醒，需要选择一条发送到执行单元

Tomasulo 算法

核心思想：

- **每个计算单元维护一个保留站(一组物理寄存器)**
  - 用于缓存操作数还没有准备好的指令
- **通过寄存器重命名消除WAW and WAR相关性**
  - 将寄存器ID重命名为保留站ID
- **乱序发送指令**

Tomasulo 算法

如果保留站仍有空余表项

- 将指令送入保留站，并进行寄存器重命名

否则 stall

对于在保留站中的每一条指令:

- 监听总线
- 当发现操作数的tag时, 获取数据并保存在保留站中
- 当所有操作数都准备好了, 指令变为可发送

发送指令到运示例算单元

当指令运算结束

- 将计算结果在总线上广播
- 寄存器文件连接着总线
  - 寄存器增加了一个tag，记录该寄存器在等待哪个保留站ID的值
  - 如果寄存器的tag和总线的tag匹配上了，将值写入寄存器 (同时清除tag)
- 回收保留站表项（清除该指令）

Tomasulo 算法示例

Reservation Station（保留站）

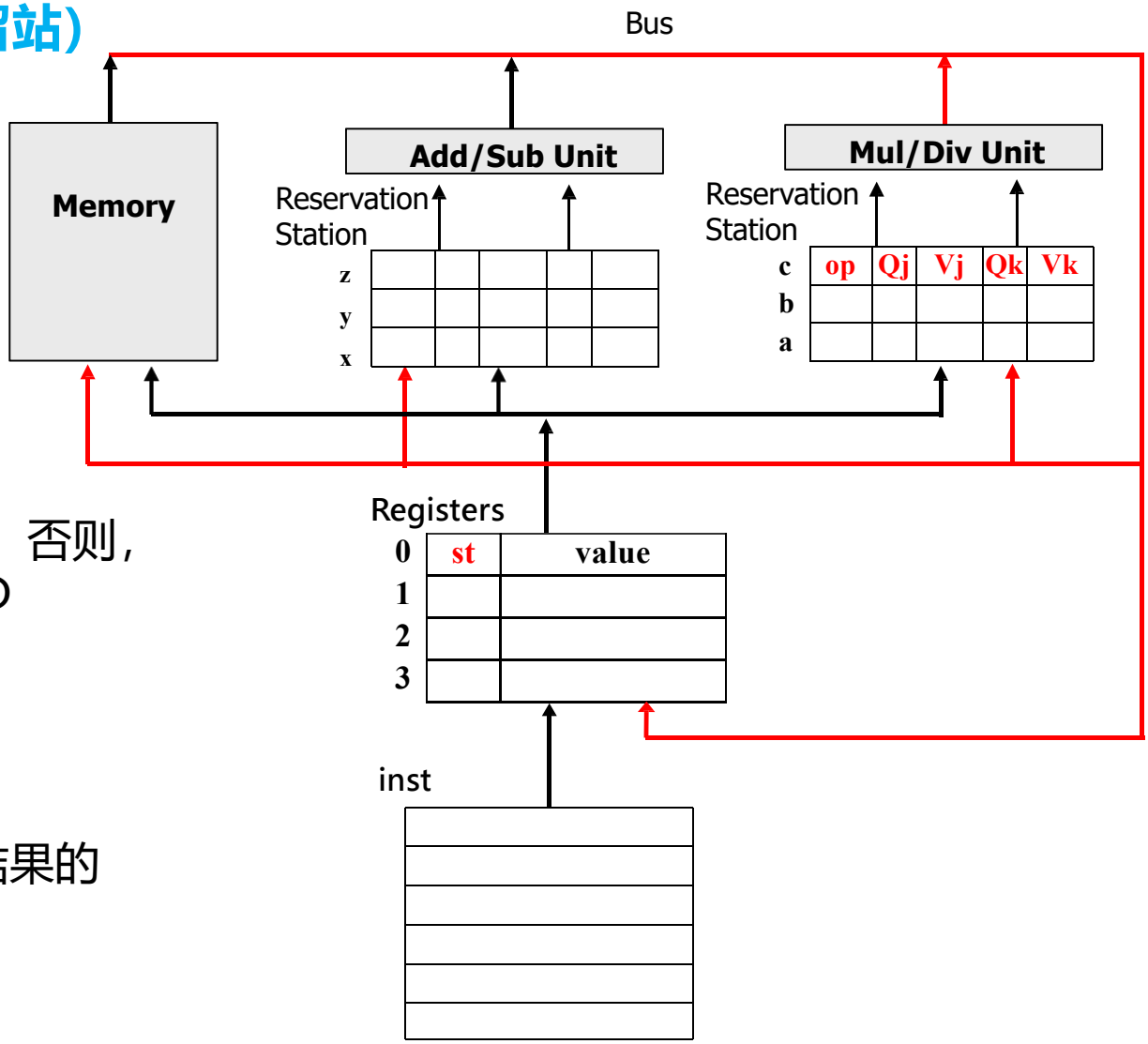
- Op: 指令的op code
- Vj, Vk: 操作数的值
- Qj, Qk: 将要提供操作数值的保留站ID(0表示准备好了)

Registers (增加一列):

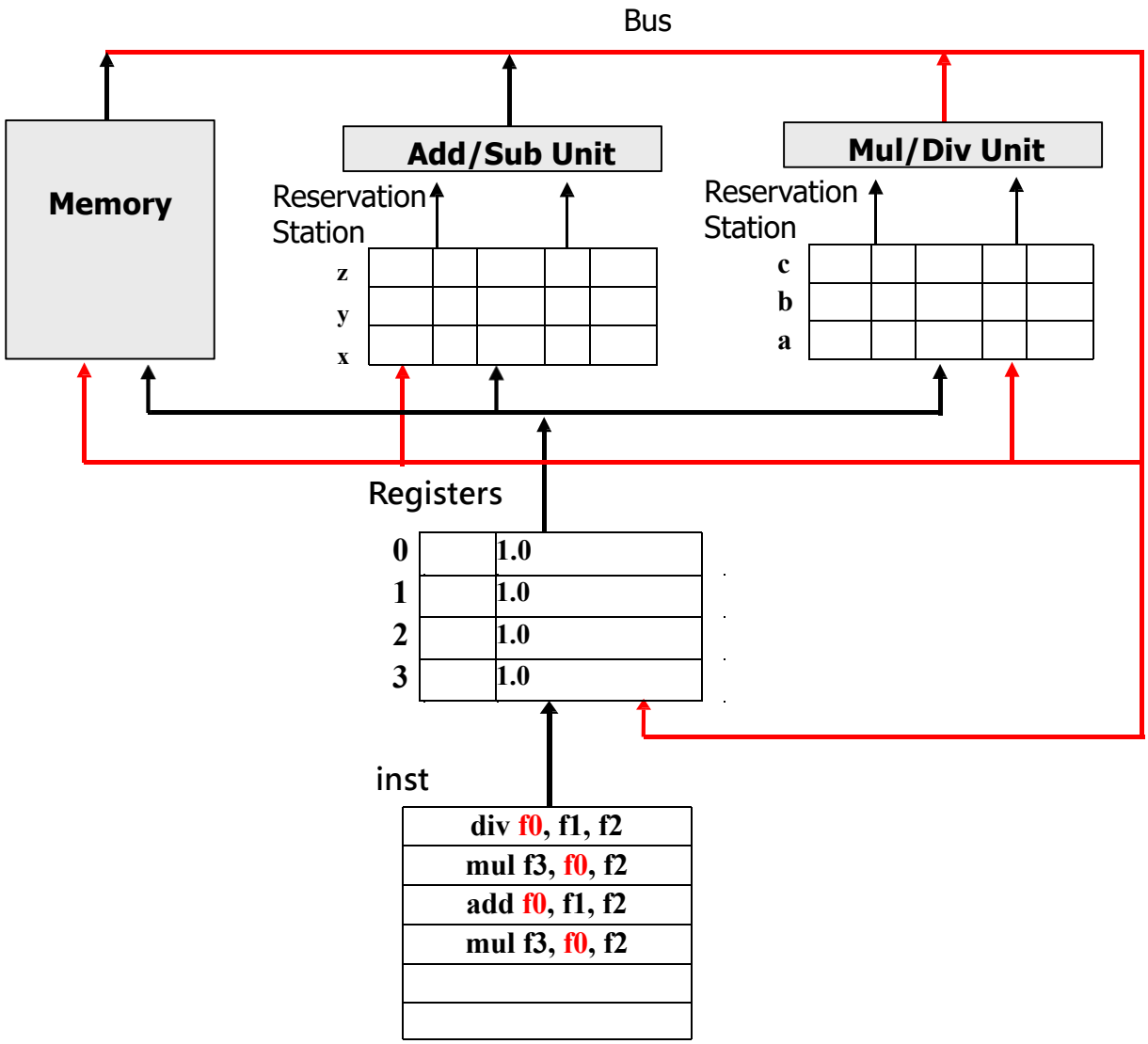
- st: “空”表示该值可以使用，否则，记录将要提供该值的保留站ID

Bus

- 广播结果，同时带上提供该结果的保留站ID

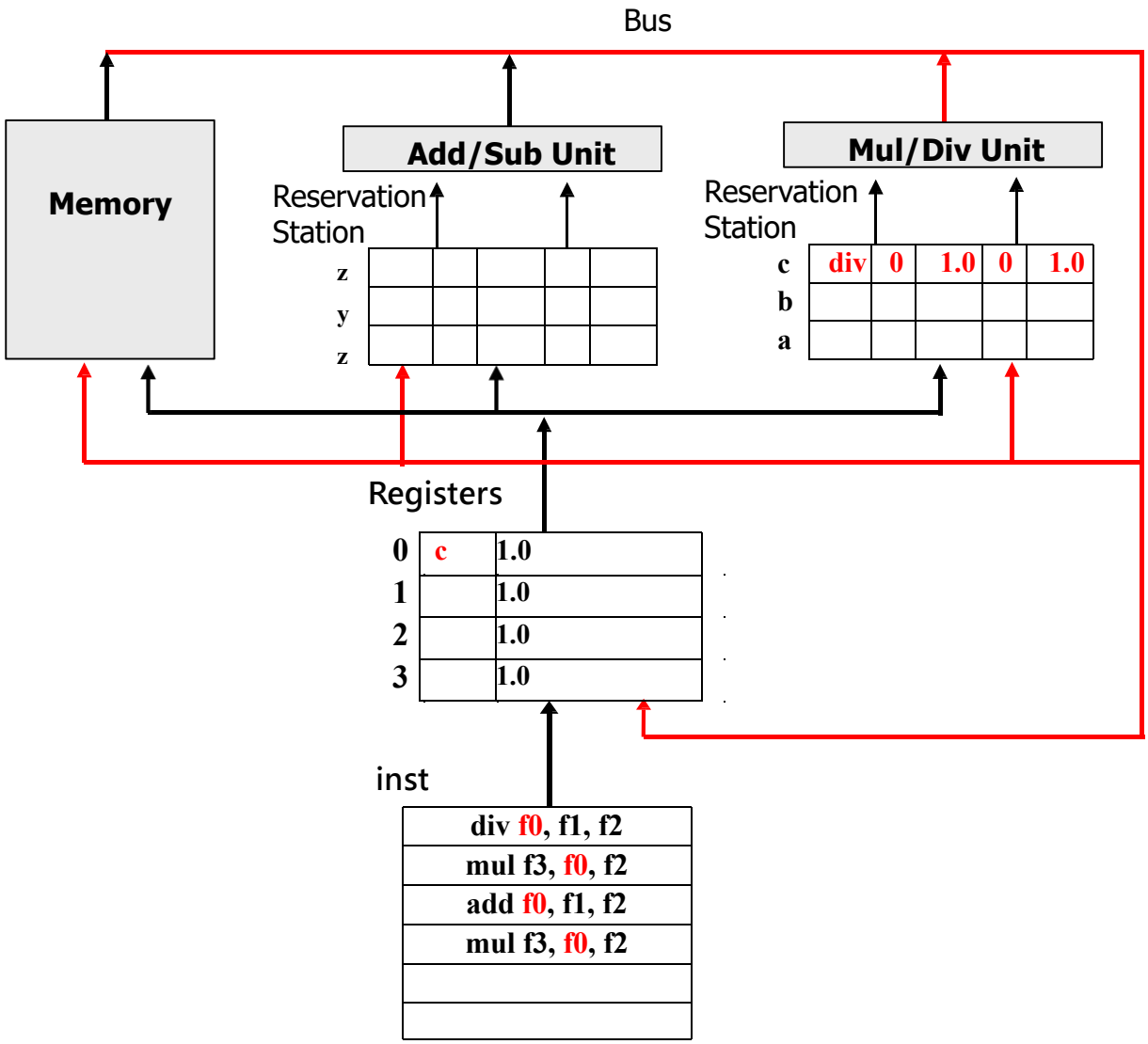


Tomasulo 算法示例



Tomasulo 算法示例

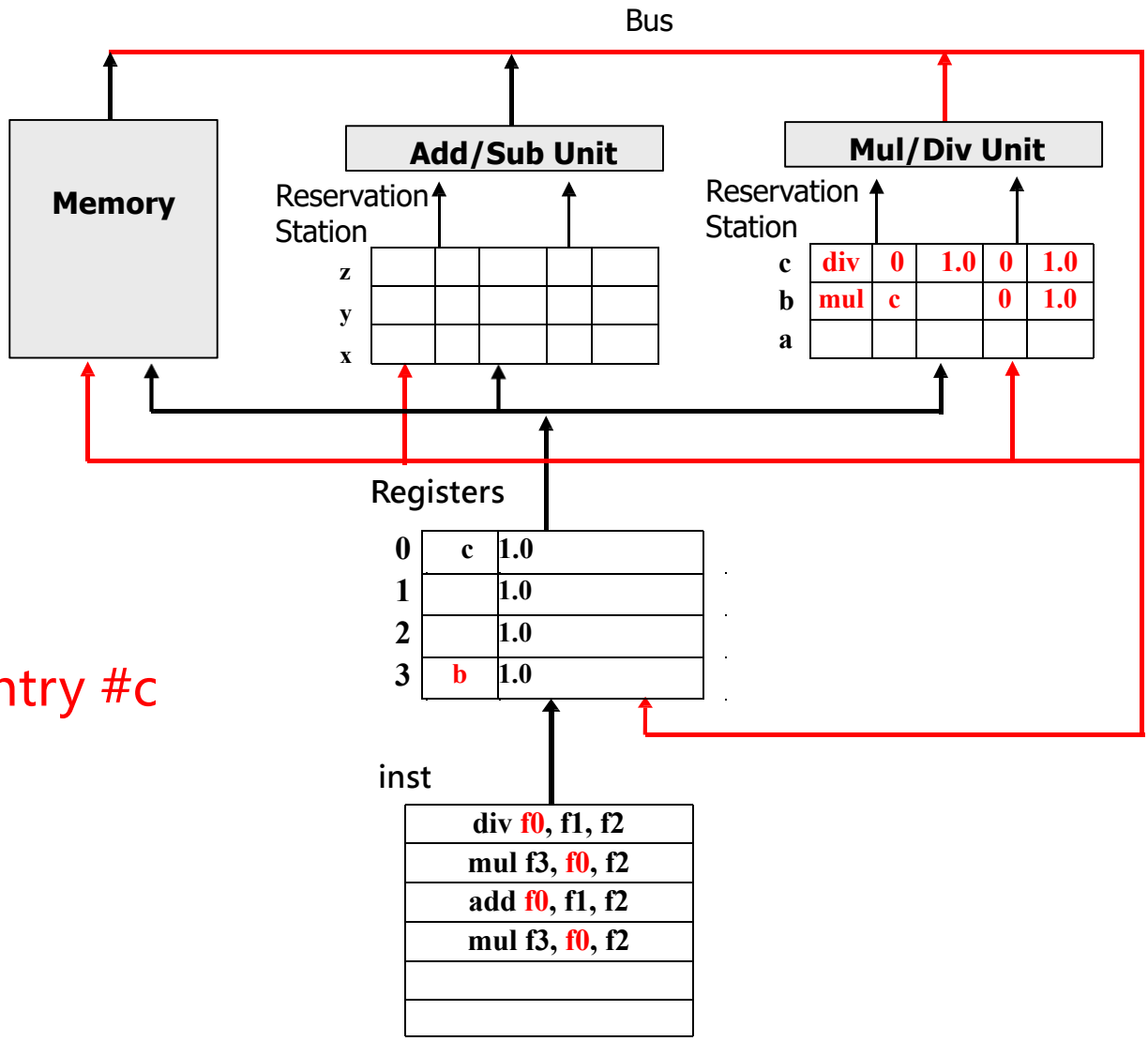
div 发射, f1 和 f2 准备好



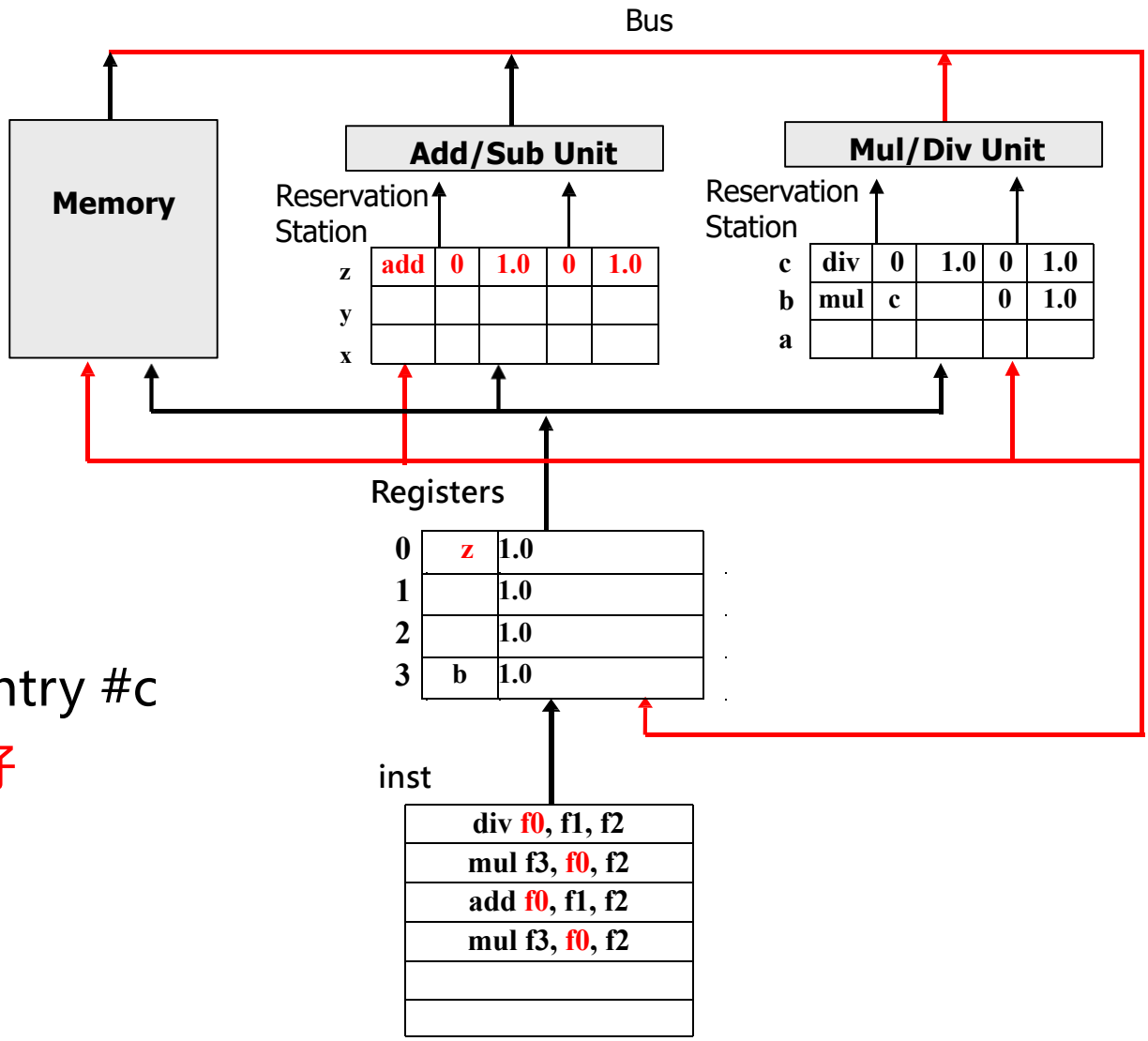
Tomasulo 算法示例

div 发射, f1 和 f2 准备好

mul1 发射, f0 等待 RS entry #c



Tomasulo 算法示例

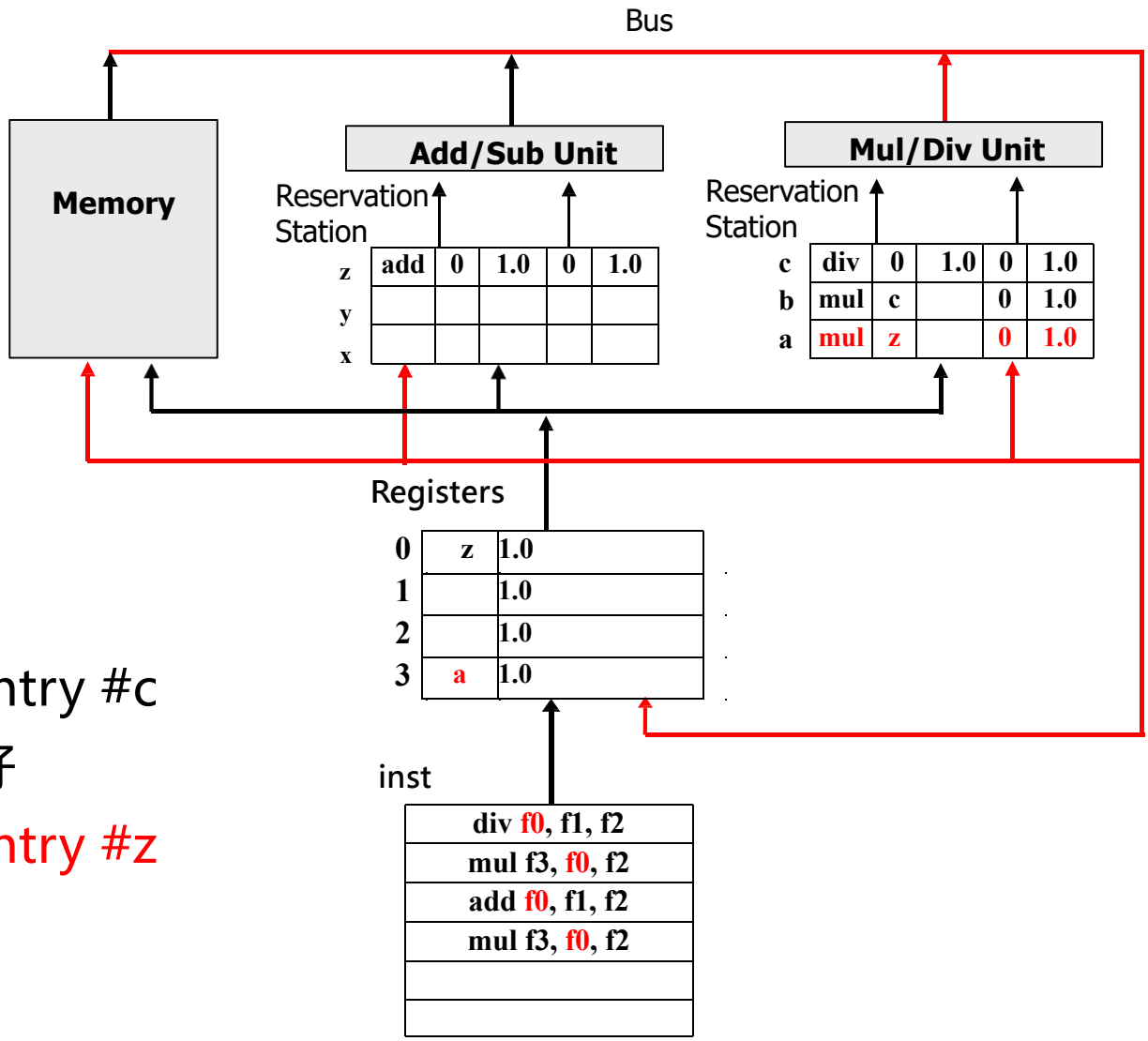


div 发射, f1 和 f2 准备好

mul1 发射, f0 等待 RS entry #c

add 发射, f1 和 f2 准备好

Tomasulo 算法示例



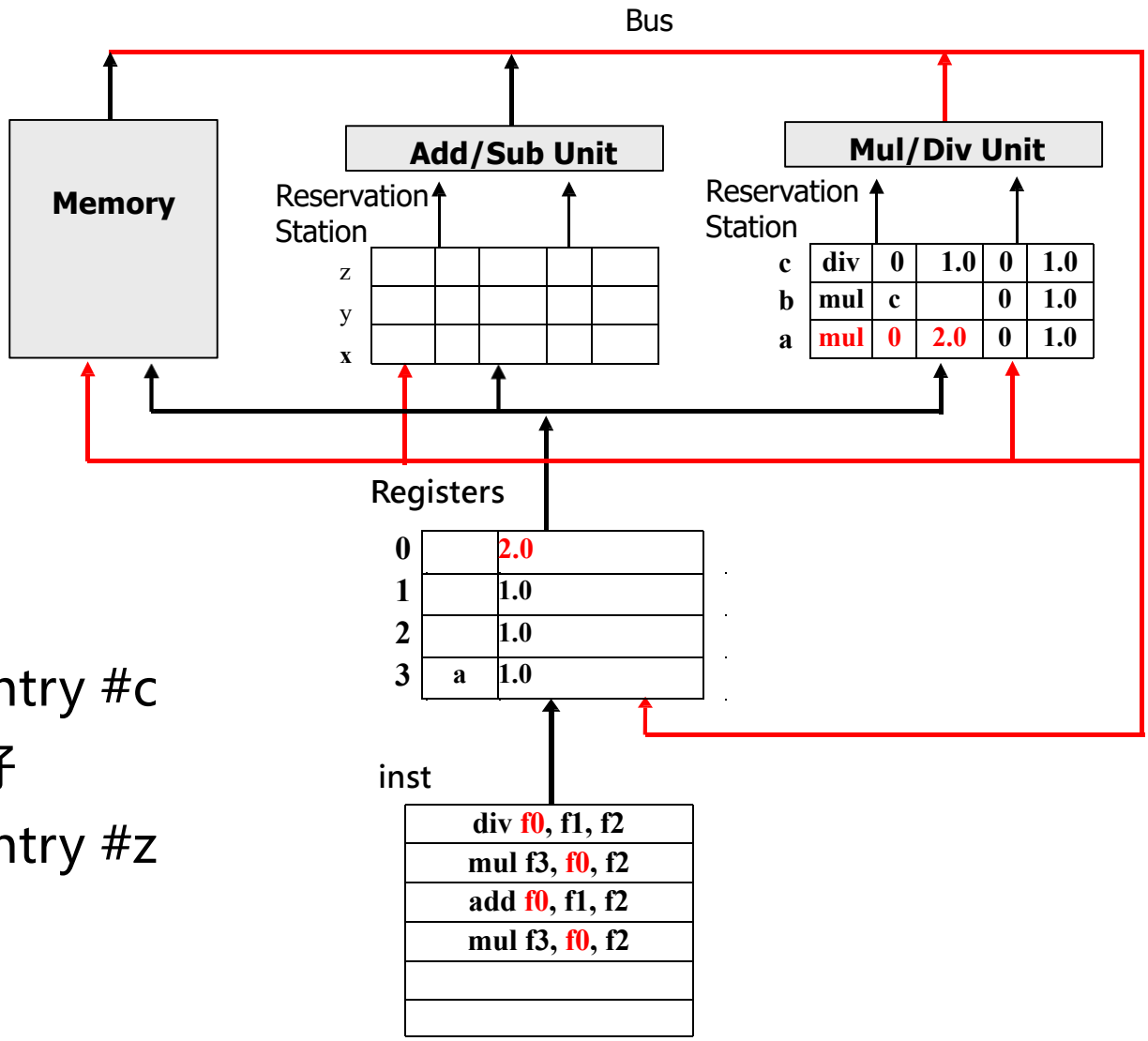
div 发射, f1 和 f2 准备好

mul1 发射, f0 等待 RS entry #c

add 发射, f1 和 f2 准备好

mul2 发射, f0 等待 RS entry #z

Tomasulo 算法示例



div 发射, f1 和 f2 准备好

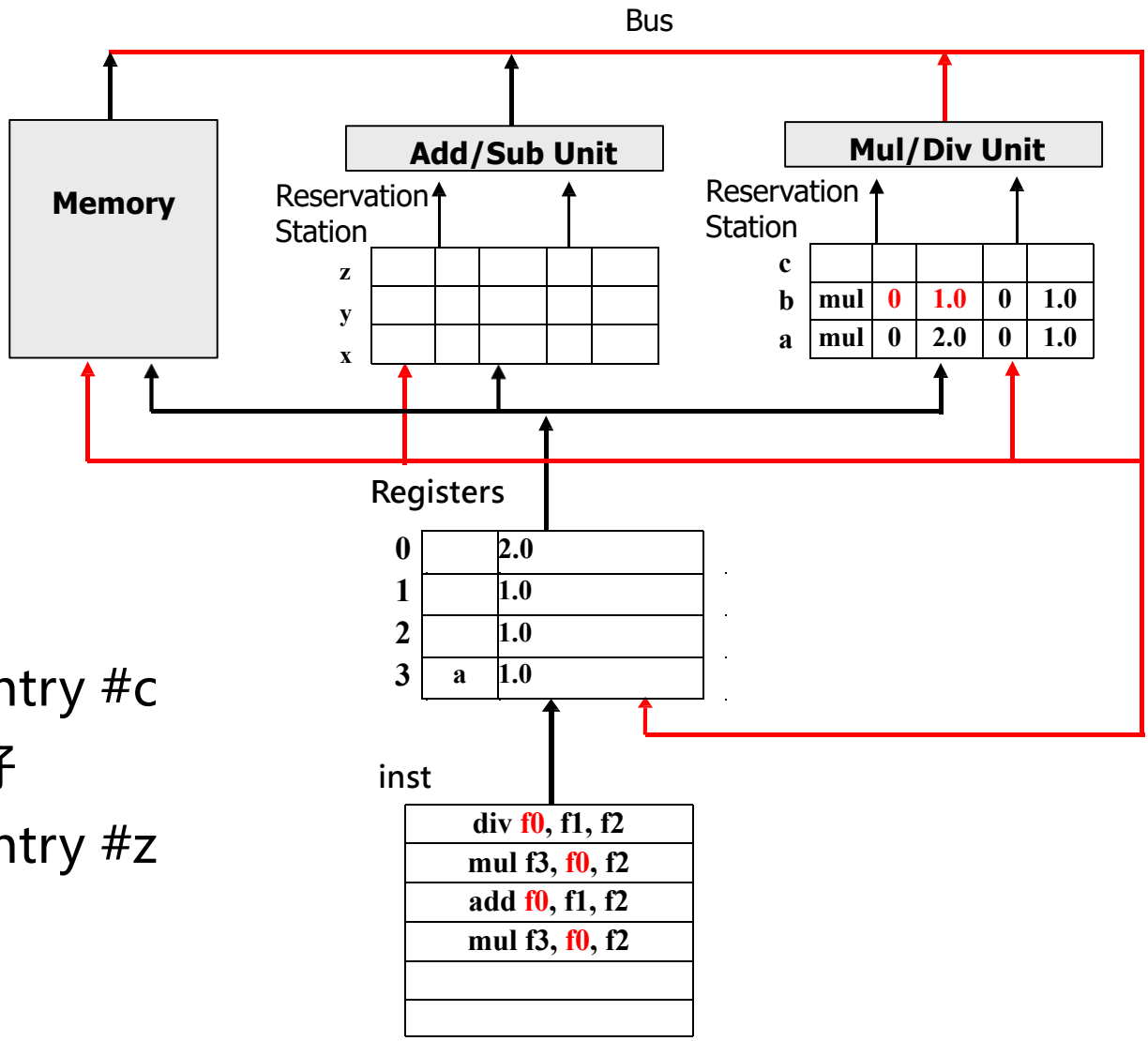
mul1 发射, f0 等待 RS entry #c

add 发射, f1 和 f2 准备好

mul2 发射, f0 等待 RS entry #z

add 写回

Tomasulo 算法示例



div 发射, f1 和 f2 准备好

mul1 发射, f0 等待 RS entry #c

add 发射, f1 和 f2 准备好

mul2 发射, f0 等待 RS entry #z

add 写回

div 写回

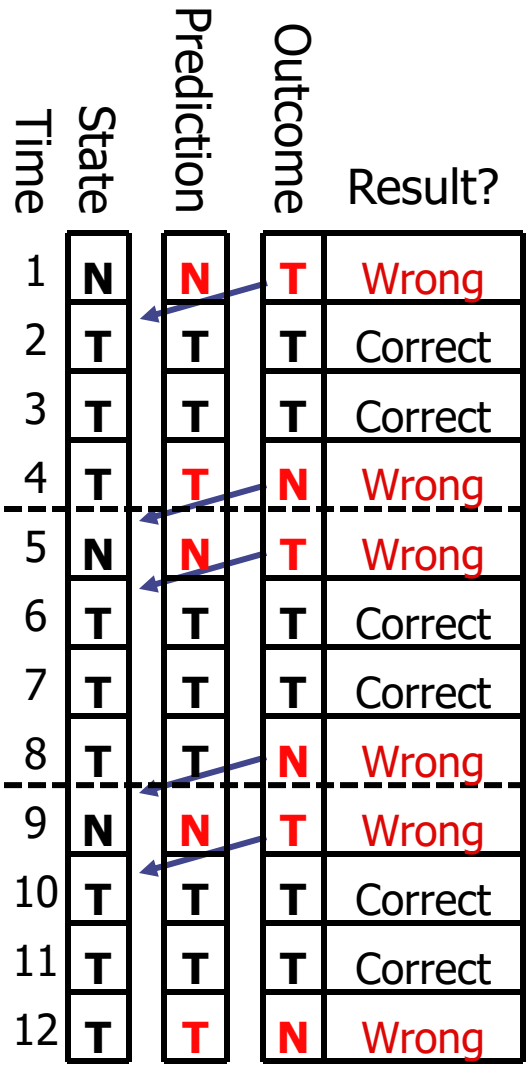
分支预测
(Branch Prediction)

Last Time Predictor

- 问题：双层循环的情况

```
for (i=0;i<100;i++)
    for (j=0;j<3;j++)
        // whatever
```

  - 每次内循环都会导致2次预测错误
  - 预测器状态转换太快了



Last Time Predictor 改进思路

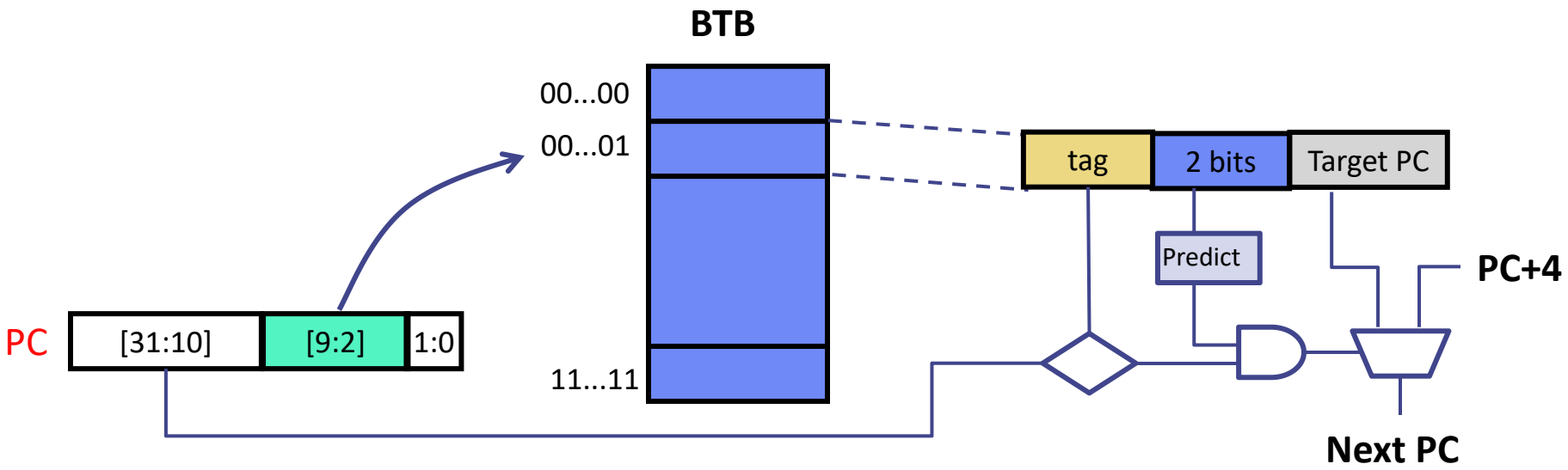
- **问题:** last-time predictor 状态变化太快（T->NT或NT->T）
  - 有些时候分支语句可能是mostly taken 或 mostly not taken

- **改进思路:** 为预测器添加滞后特性，这样在出现单次不同的结果时预测不会立即改变
  - 使用2 bits 记录预测历史
  - T和NT都分别有2个状态表示

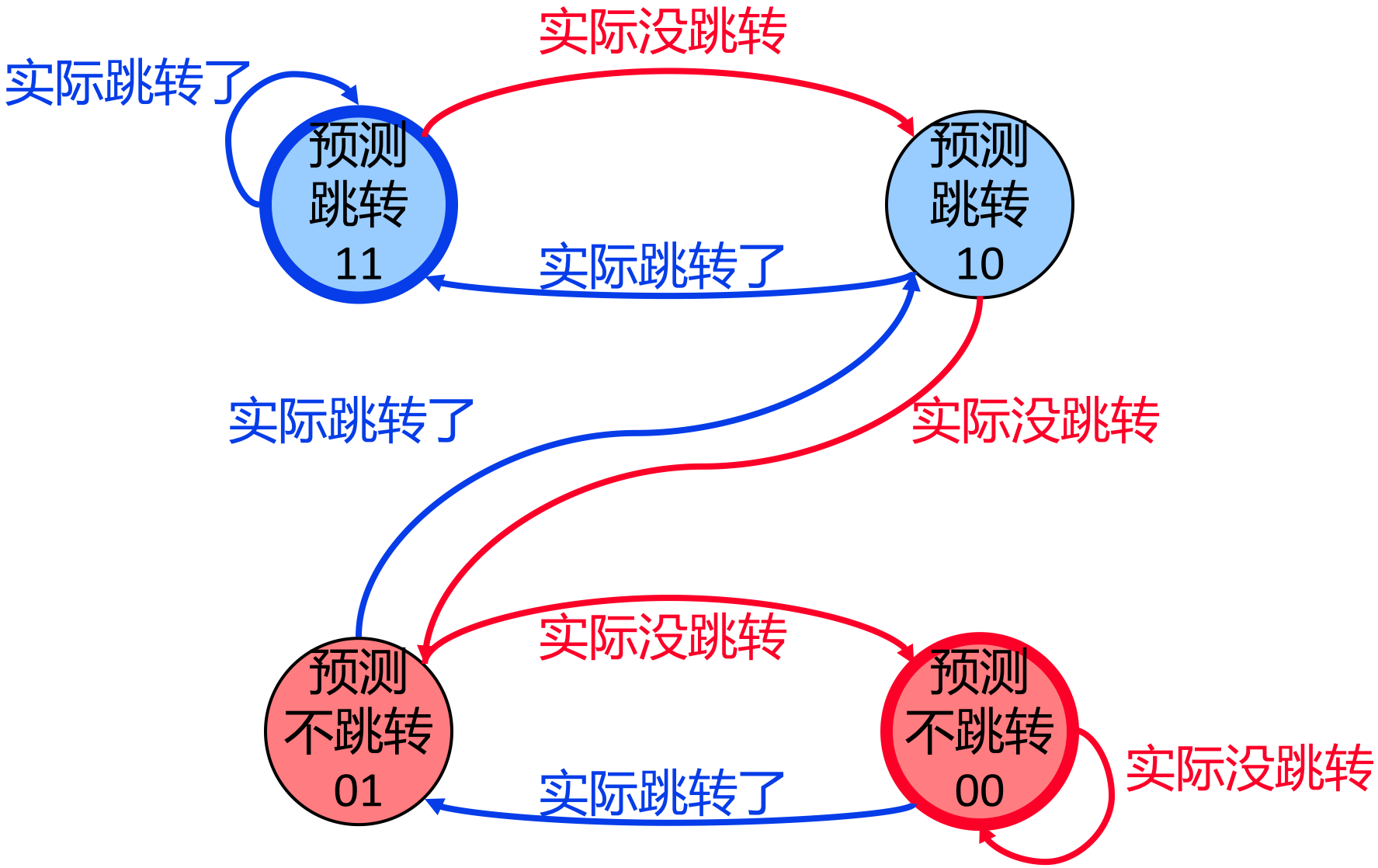
Smith, “A Study of Branch Prediction Strategies,” ISCA 1981.

2 bits 饱和计数器

- 每个分支语句用2 bits计数器进行预测
- **2 bits:** (00->N, 01->n, 10->t, 11->T)



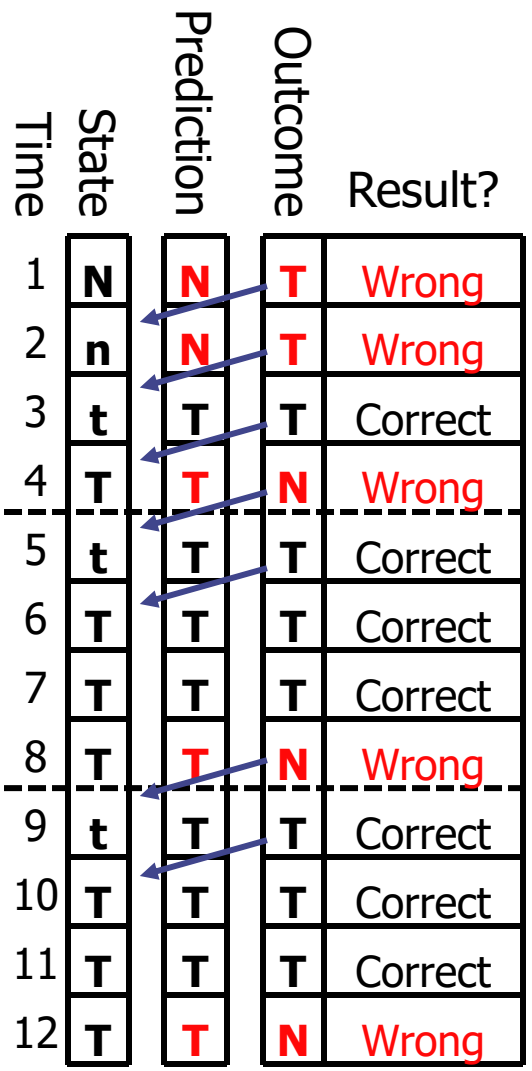
2 bits 饱和计数器


实际跳转了
实际没跳转
预测
跳转
11
预测
跳转
10
实际跳转了
实际跳转了
实际没跳转
实际没跳转
预测
不跳转
01
预测
不跳转
00
实际跳转了
实际没跳转

2 bits 饱和计数器

```
for (i=0;i<100;i++)
    for (j=0;j<3;j++)
        // whatever
```

- **稳定之后每次循环只有1次预测错误**



+ 预测准确率更高
-- 硬件开销更大

2bits饱和计数器的预测准确率在85-90%

多发射
(Multi-Issue)

多发射方案

- **Superscalar(超标量)**: 每周期发射多条指令(个数不固定，1 to 8), 由编译器或者硬件调度
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium 4, i7

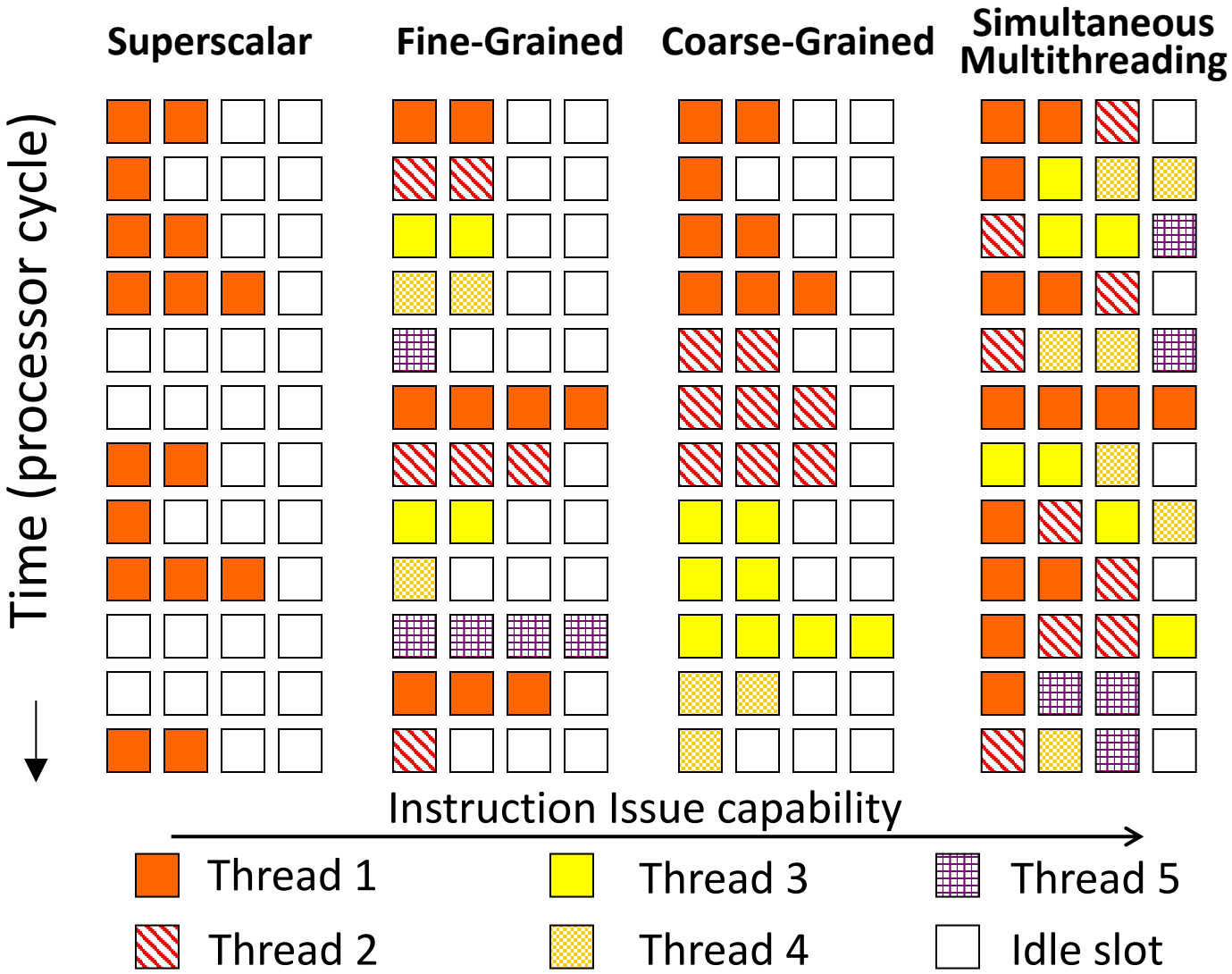
- **Very Long Instruction Words (VLIW, 超长指令字):** 将多条指令(4-16)打包成一条固定格式的长指令,由编译器静态完成
  - Intel architecture-64 (IA-64) 64-bit address (renamed: “Explicitly Parallel Instruction Computer (EPIC)”)

超标量 vs. 超长指令字

Superscalar	VLIW
在一个时钟周期内发射多条指令，多个执行单元并行处理这些指令	将多条指令打包成一个超长指令字
动态调度：硬件在运行时行指令的调度和重排，提高指令级并行性。	静态调度：编译器负责指令打包和调度，硬件设计简化。
复杂的硬件设计	复杂的编译器设计
较高的功耗和热量	低功耗和高效性
灵活性高	灵活性差
适合通用计算任务，如服务器、桌面计算机	适合于特定应用场景，如数字信号处理器（DSP）、嵌入式系统
被广泛采用，如Intel Core、AMD Ryzen	用于一些专用处理器，如Intel Itanium、DSP

多线程
(Multithreading)

单核多线程执行策略

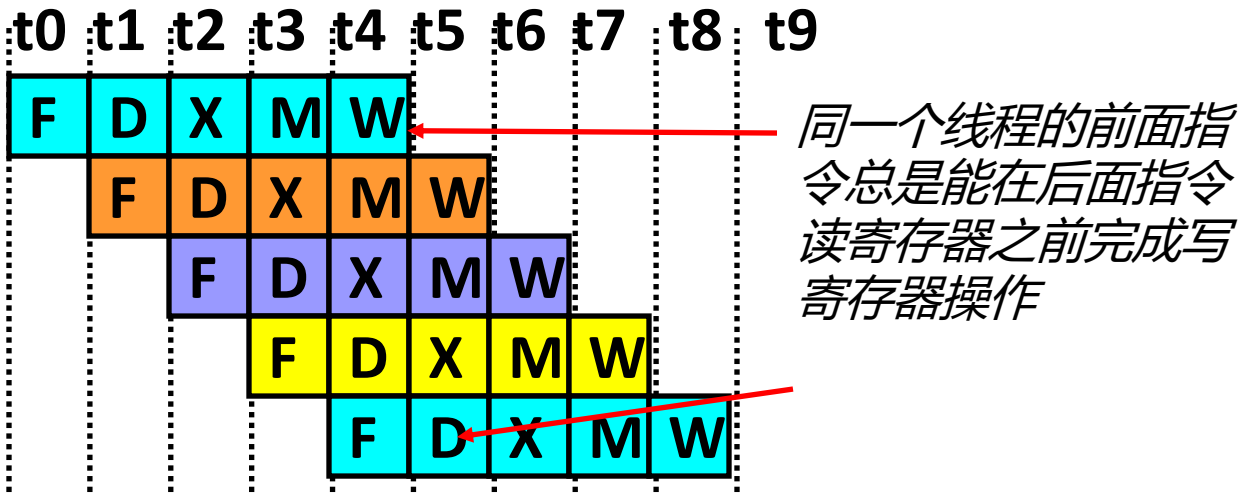

Superscalar
Fine-Grained
Coarse-Grained
Simultaneous Multithreading
Time (processor cycle)
Instruction Issue capability
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Idle slot

细粒度多线程(Fine-Grained Multithreading)

□ 在简单流水线上交错执行不同线程的指令

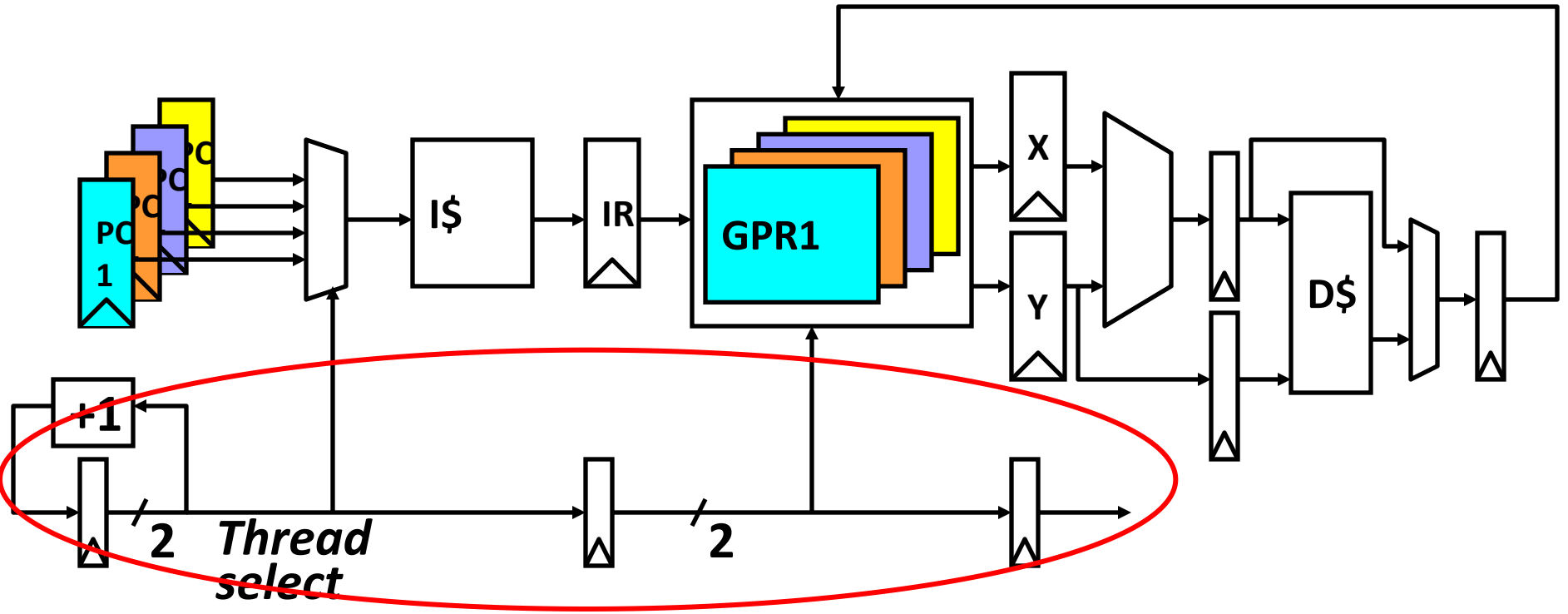
Interleave 4 threads, T1-T4, 5-stage pipe, no internal forwarding

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: SUB r5, r1, r4



细粒度多线程流水线

- 将线程选择信号传递到流水线下游，以确保在每个流水线阶段读取/写入正确的状态位
  - ✓ 在软件看来，好像多个慢速CPUs



细粒度多线程

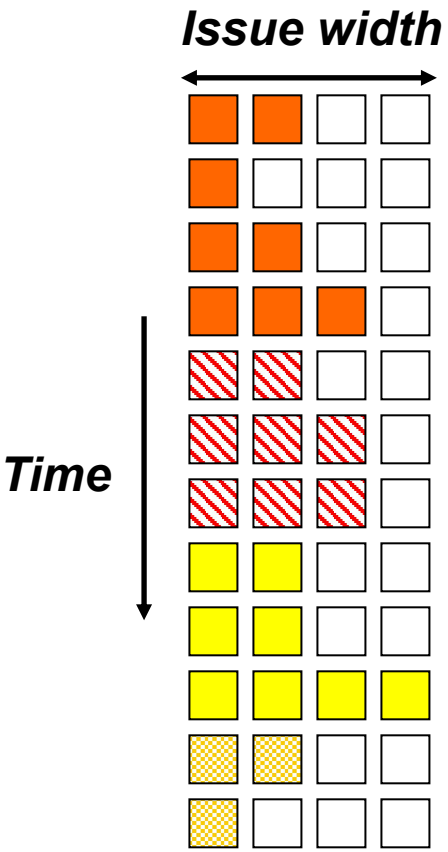
- **优点:**
  - **能掩盖短时间和长时间流水线停顿**
    - e.g., 内存操作, 有依赖关系的指令, 分支指令等
    - 因为线程切换足够频繁，每次切换不需要刷新流水线
  - **更能充分利用硬件资源**

- **缺点:**
  - **单个线程执行时间变长**
    - 一个没有stall的线程可能频繁被切换出去，被来自其他线程的指令拖慢
  - **必须有足够多的线程**
    - 否则无法填不满流水线

粗粒度多线程(Coarse-Grained Multithreading)

- **仅当遇到stall时间较长的事件时才切换线程**
  - 例如cache misses
  - 一个线程可以在连续多个周期使用流水线
  - 可以掩盖长停顿

- **可能的stall事件**
  - Cache misses
  - Synchronization events (e.g., 读取空的队列等待数据而阻塞)
  - FP operations



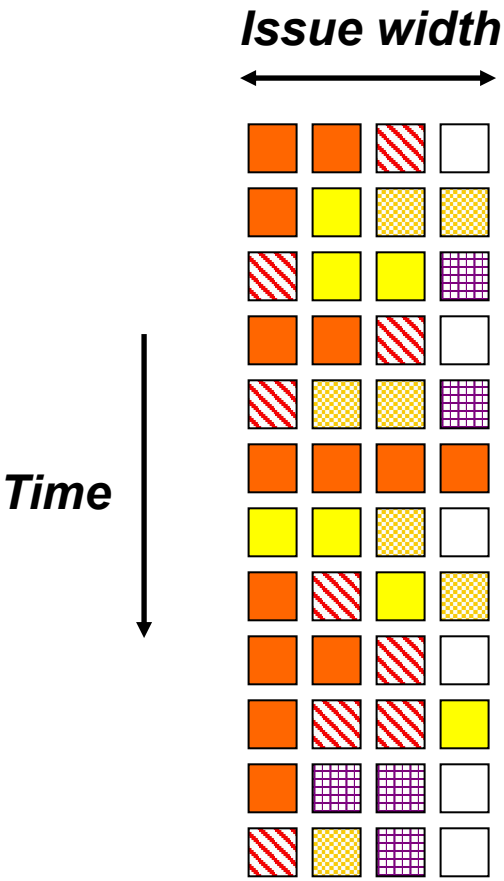
粗粒度多线程

- **优点:**
  - 线程切换不用那么频繁
  - 不会拖慢线程，因为线程只有stall的时候才会被切换出去
  - 关键线程需要高优先级

- **缺点:**
  - 发生切换时，指令必须排空并重新填充 → 流水线较深时影响较大，适合stall时间很长的情况
  - 公平性: stall较少的线程占用流水线时间较长，其他线程可能饿死
    - 解决方案: 占用时间超过一定额度强制切换

同步多线程(Simultaneous Multithreading,SMT)

- **允许同一周期发射的多个指令来自不同线程**
  - 保持多个执行单元的高利用率
  - 更好的流水线效率
  - 多个线程同时在流水线执行，无需上下文切换



SMT的硬件资源

需要复制的资源 (每个线程都有一份)	• return address stack • PC • architected register file • control logic/state
不需要复制，但需要划分的资源 (每个线程分配一部分)	• reorder buffer • load/store buffers • various queues(e.g., scheduling queue)
多个线程共享的资源	• Caches: L1, L2, L3 cache • microarchiteture registers(e.g., physical registers) • execution unit

Cache 一致性协议

Cache一致性问题

- **问题: 如果多个处理器同时拥有同一个cache block, 如何保证一致性(cache coherence)?**

- **本地更新会导致cache状态不一致**
  - ✓write-through和writeback caches都存在该问题

维护Cache一致性(Coherence)

- **一致性定义：** 对于相同的内存位置，所有处理器看到的值是一致的

- **基本策略：** 将最新值(写操作)及时更新到所有cache copy

- 写操作的两种处理方式
  - **Option 1 (update protocol):** 将新数据广播给所有cache copy
  - **Option 2 (invalidate protocol):** 让所有其他copy都无效，只保留一份有效的本地copy, 并更新它

消息如何传播?

- **Snoopy bus: (侦听式)**
  - 处理器在总线上广播数据更新
  - 每个处理器都侦听其他处理器的动向
  - 所有请求通过总线实现序列化→ **完全按顺序**

- **Directory: (目录式)**
  - 通过目录跟踪每个block的所有权
  - 处理器通过目录显式地请求数据
  - 目录协调invalidation/update动作
  - 充当为乱序请求”排序”的角色

总线侦听的作用

- 当处理器写本地copy时，会广播到所有其他cache，所有cache将看到相同的更新
  → **write propagation**

- 当两个处理器几乎同时向各自的本地copy写入相同数据时，一个处理器将会赢得总线，并向所有其他缓存广播其更新信息，另一个处理器必须等待直到它能够获取总线
  → 这保证了所有其他缓存看到相同的顺序
  → **write serialization**

Update vs. Invalidate Tradeoffs

Update-based

+ 如果数据更新不频繁, 可以有效避免invalidate产生的额外开销(下次读要重新load)

- 如果数据在其他处理器介入读取之前又被重写(更新频繁)，那么更新就是无用的

Invalidate-based

+ 使无效广播后, 处理器有唯一有效copy
+ 只有使无效后又读的处理器才会有本地copy

- 下次读取数据时间长(re-load)
- 如果写操作竞争激烈, 产生ping-pong效应 (使无效->读->又使无效)

Write-back Cache：MSI 协议

- **每个cache block有3种状态**
  - ***Invalid***：本地没有copy
  - ***Shared***：本地有一个只读copy，该copy有最新的数据
  - ***Modified***：本地有全局唯一的valid copy(其他cache无copy)，该copy处于dirty状态(可能和内存不一致)，本地可写(不需要通知他人)

- **处理器的动作**
  - load, store, Evict(cache被换出)

- **总线事物**
  - BusRd (load), BusRdX (store), BusWB (write back), …

MSI 协议状态转换

State	*This Processor*		*Other Processor*	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Load Miss ➔ S	Store Miss ➔ M	---	---
Shared (S)	Hit	➔ M	➔ S	➔ I
Modified (M)	Hit	Hit	➔ S	➔ I

MSI 协议存在的问题

- **如果本地有全局唯一的copy，处于shared状态**
  - 此时**发生本地write**，会在总线上广播BusRdX，自己升级为modified，然后再写
  - 因为全局只有一个copy，其实没必要广播消息

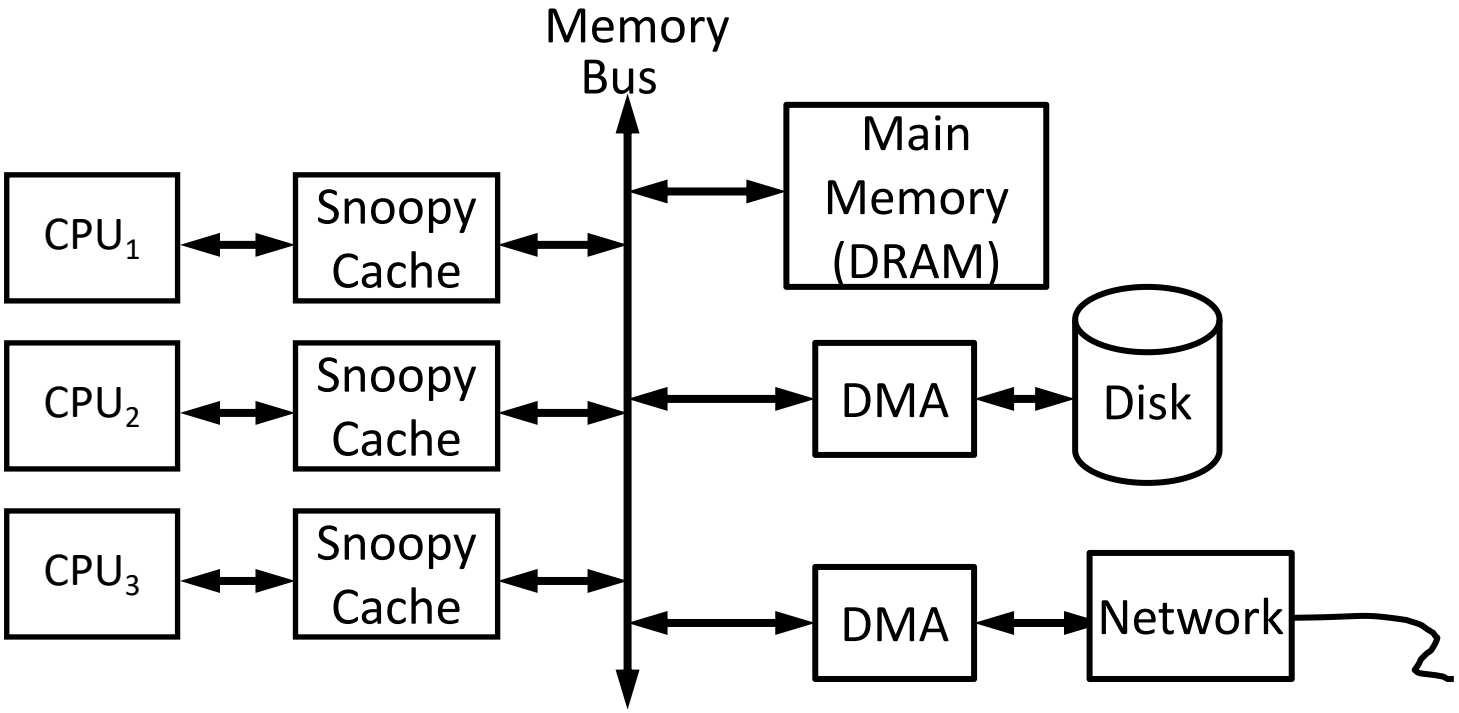
解决方案: MESI

- **Idea:** 增加一个状态(Exclusive)，表明该copy是全局唯一的，并且值是最新的(clean)
  - ***Exclusive* 状态**: 可以直接变为M状态，不需要广播消息

- **如何变为E状态**：全局仅有一个clean的copy时
  - ✓ 如果read一个block的时候其他cache中都没有，那么该block直接为E状态
  - ✓ 系统原本有多个S状态的copy，但是因为cache替换，全局仅剩一个S状态的copy时，该copy也应该变为E状态

侦听式Cache Coherence

- 如果所有cache共享总线，很容易实现
  - 每个cache在总线上广播其read/write操作
  - 每个cache block的有限状态机(FSM)比较简单
  - 适合小规模多处理器



MESI Protocol 状态转换

State	*This Processor*		*Other Processor*	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → **S or E**	Miss → M	---	---
Shared (S)	Hit	→ M	→ S	→ I
Exclusive (E)	Hit	Hit → M	→ S	→ I
Modified (M)	Hit	Hit	→ S	→ I

*注意表格没有体现cache替换引起的状态变化

MESI 存在的问题

- **Shared状态要求数据总是最新的(clean)**
  - i.e., 所有shared copy都有最新的数据，包括内存

- **Problem:** 一个block处于M状态(表明数据dirty)，如果其他cache读该block，那么需要将data先写回内存，否则大家都变为shared状态时，数据都是dirty的

- **为何这会成为一个问题?**
  - 如果每次从M->S都将数据写回内存，其实很多都是没必要的 → 因为很快别的处理器可能又更新了数据

MESI 改进 -> MOESI Protocol

思路：

- **发生M→S转换时，只将最新数据传给请求者，新数据不立即写回内存**
- **包含dirty数据的cache块被换出时再写回内存**
  - ✓ 符合write-back原则

挑战：

- **M->S转换后，出现多个shared copy都包含dirty数据，谁负责写回?**
  - ✓ shared copy的数据是否为dirty无法判断(除非增加标记)
  - ✓ 如果每个shared copy被换出时都写一遍内存->开销大

MESI 改进 -> MOESI Protocol

思路：

- **发生M→S转换时，只将最新数据传给请求者，新数据不立即写回内存**
- **包含dirty数据的cache块被换出时再写回内存**
  - ✓ 符合write-back原则

解决方案：MOESI 协议

- **发生M→S转换时，将其中一个shared copy标记为owner (O)状态，它在被换出的时候写回内存**
  - ✓ O状态的cache copy一定是dirty的
  - ✓ 不用每个shared copy都写回内存

MOESI Protocol 状态转换表

State	*This Processor*		*Other Processor*	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → S or E	Miss → M	---	---
Shared (S)	Hit	→ M	→ S	→ I
Owned (O)	Hit	→ M	→ O	→ I
Exclusive (E)	Hit	Hit → M	→ S	→ I
Modified (M)	Hit	Hit	→ O	→ I

多处理器内存一致性协议

内存一致性模型

- **硬件/编译器和程序员之间的一个约定**
  - 硬件和编译器在指令重排中不能违反个约定
  - 程序员也需要遵循约定的规则

- **Example:** 如果程序员想从P2获得23

P1	P2
`A = 23;`	`while (Flag != 1) {};`
`Flag = 1;`	`... = A; // get 23`

为了满足这个执行顺序, 硬件或编译器不能进行某些优化, e.g. load bypassing ➔ **会影响执行效率**

顺序一致性 (Sequential Consistency)

- **一个多处理器系统被称为sequentially consistent (SC), *如果满足下面两个条件:***
  - 所有处理器的操作按照某种顺序执行
  - 来自同一个处理器的操作按照指令顺序执行

- **这是一种内存顺序模型, 或内存模型**
  - 所有处理器看到的内存指令执行顺序相同, i.e., 所有内存操作按照某种顺序执行 (称为***global total order***)
  - 在这个执行顺中, 每个处理器的操作按照指令顺序执行

顺序一致性

- **简单、直观**
  - 与程序员的直觉一致
  - 容易解释程序行为

- **在同一次执行中，所有处理器看到相同的内存指令执行顺序**
  - 无正确性问题

- **如果多次执行，每次执行可以观察到不同的执行顺序** (每次都是sequentially consistent)
  - 调试仍然很困难 (每次执行指令顺序不一样)

顺序一致性的问题

- **硬件上很难高效实现**
  - 简单实现:
    - 无并发内存访问
    - 每个节点严格按照顺序访问内存
    - 本质上排除了乱序执行处理器

- **不必要的限制**
  - 并行程序很多操作可以安全地重排序，但因为SC限制无法实现

- **如何解决?**
  - 让程序员多做一些工作, e.g. 提供更明确的提示, 让硬件/编译器有更多优化空间

  → 削弱或放宽内存一致性模型

弱一点的模型: Weak Consistency

- **程序员在程序中插入 *fence* 指令:**
  - fence之前的指令一定比fence之后的指令早完成
  - 所有在fence之前的数据操作一定比fence早完成
  - 所有在fence之后的数据操作一定比fence晚完成
  - 所有的fence按照指令顺序完成

- **同步原语, 例如barrier,可以用作fence**
  - Fence在特定的点上传播读写操作,类似刷新内存操作

P1

p = **new A(...)**

FENCE

flag = **true**;

Weak Consistency

- **Advantage**
  - 不用保证严格的内存操作顺序
    - 让硬件有更多的性能优化空间
- **Disadvantage**
  - 程序员负担加重 (need to get the “fences” correct)

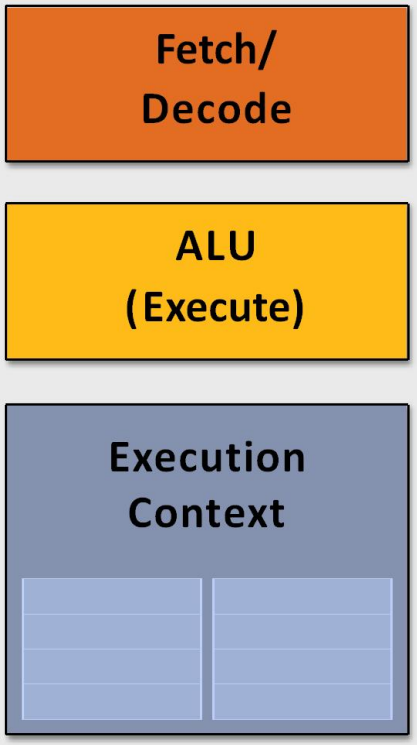
更弱的模型: Release Consistency

- **将fence分成两个: 一个保证在此之前完成，一个保证在此之后完成**
  - ***Acquire***: 在acquire之后的访存指令必须在acquire完成之后完成
  - ***Release***: 在release之前的访存指令必须在release完成之前完成

- **However,**
  - Acquire 之前的指令完成时间无限制
  - Release之后的指令完成时间无限制

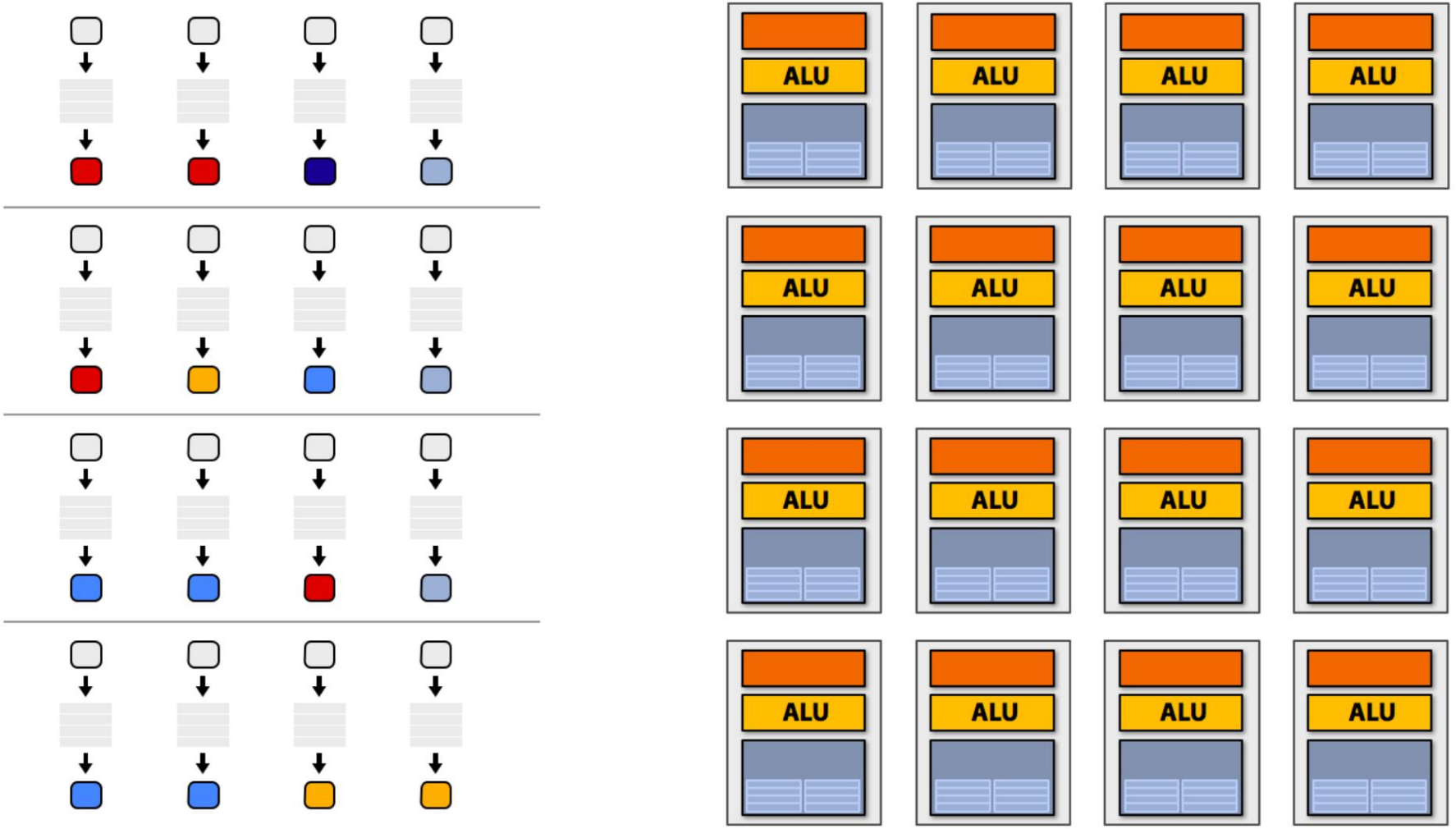
GPU

GPU 设计思路: (1) 简化流水线, 增加核数



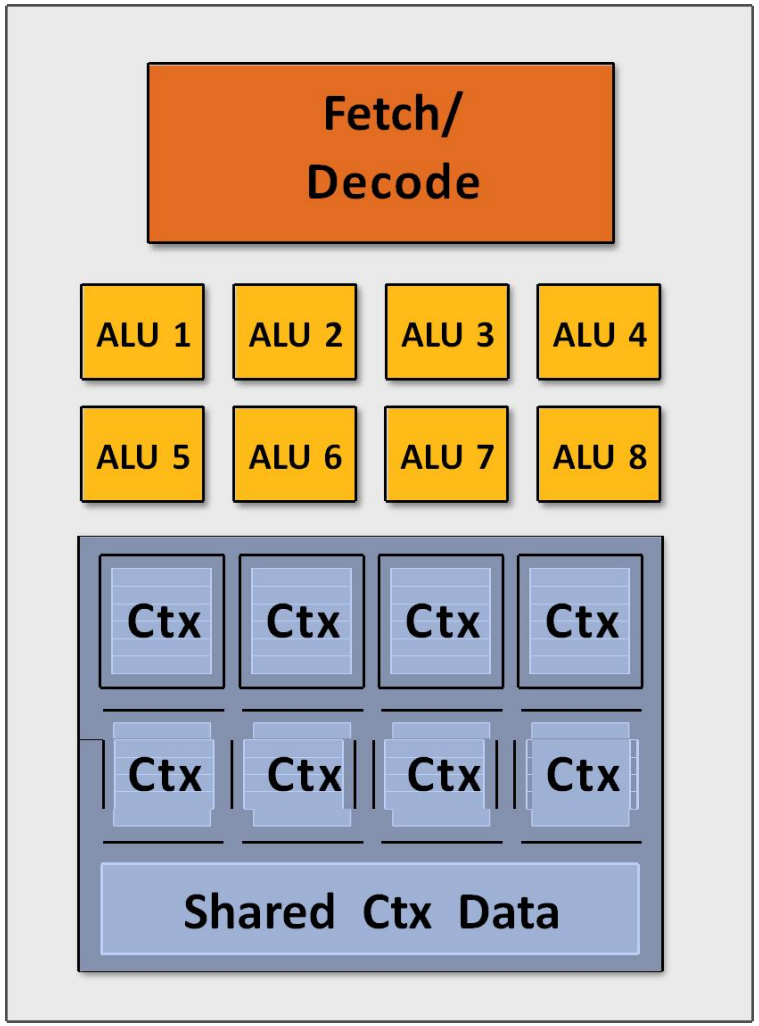
Idea #1:
对流水线进行瘦身, 去掉乱序执行、分支预测等复杂逻辑, 节省的空间用来增加大量核心

GPU 设计思路: (1) 简化流水线, 增加核数



节省的晶体管用来增加核心个数，并行处理多个数据

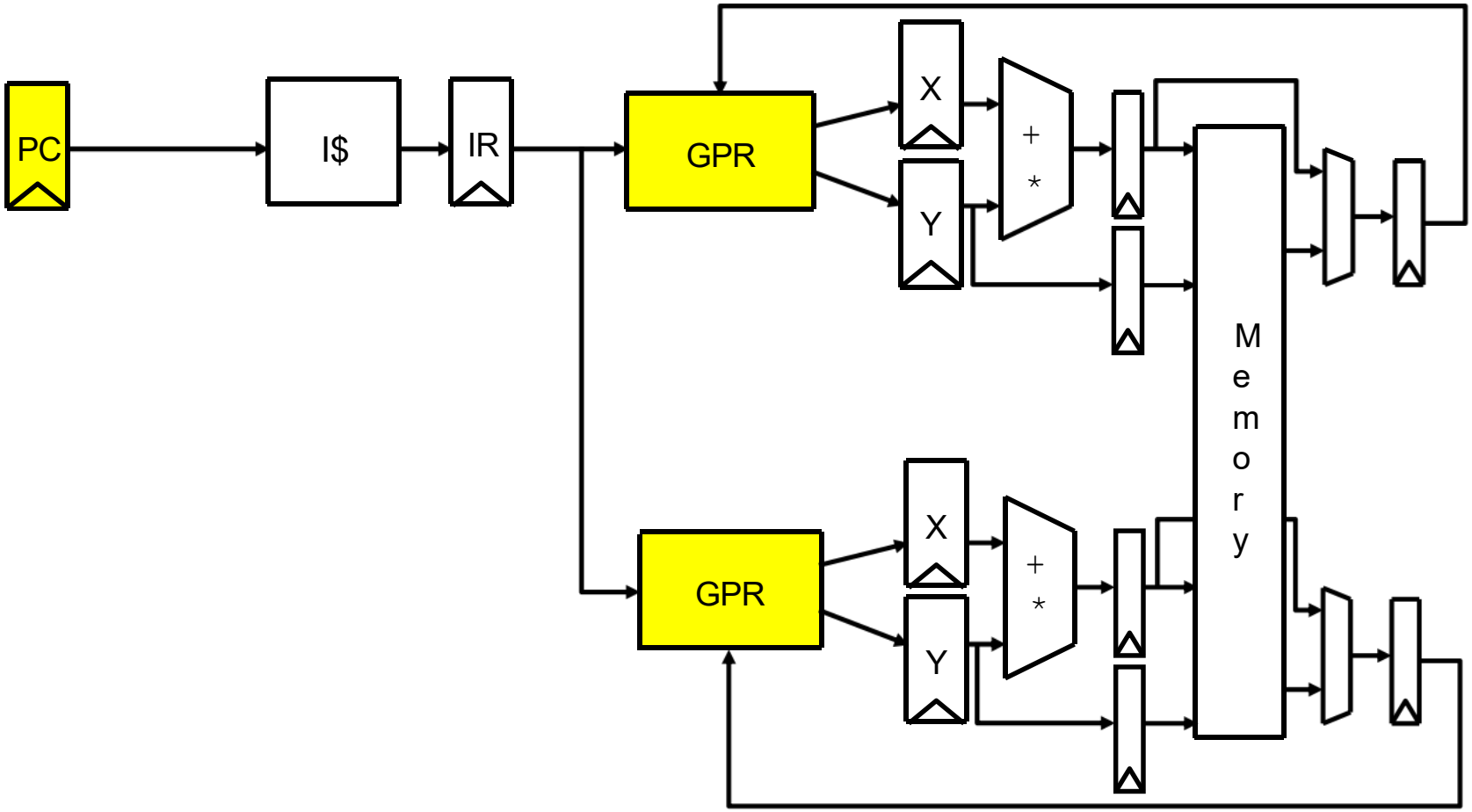
GPU 设计思路: (2) 单指令多线程



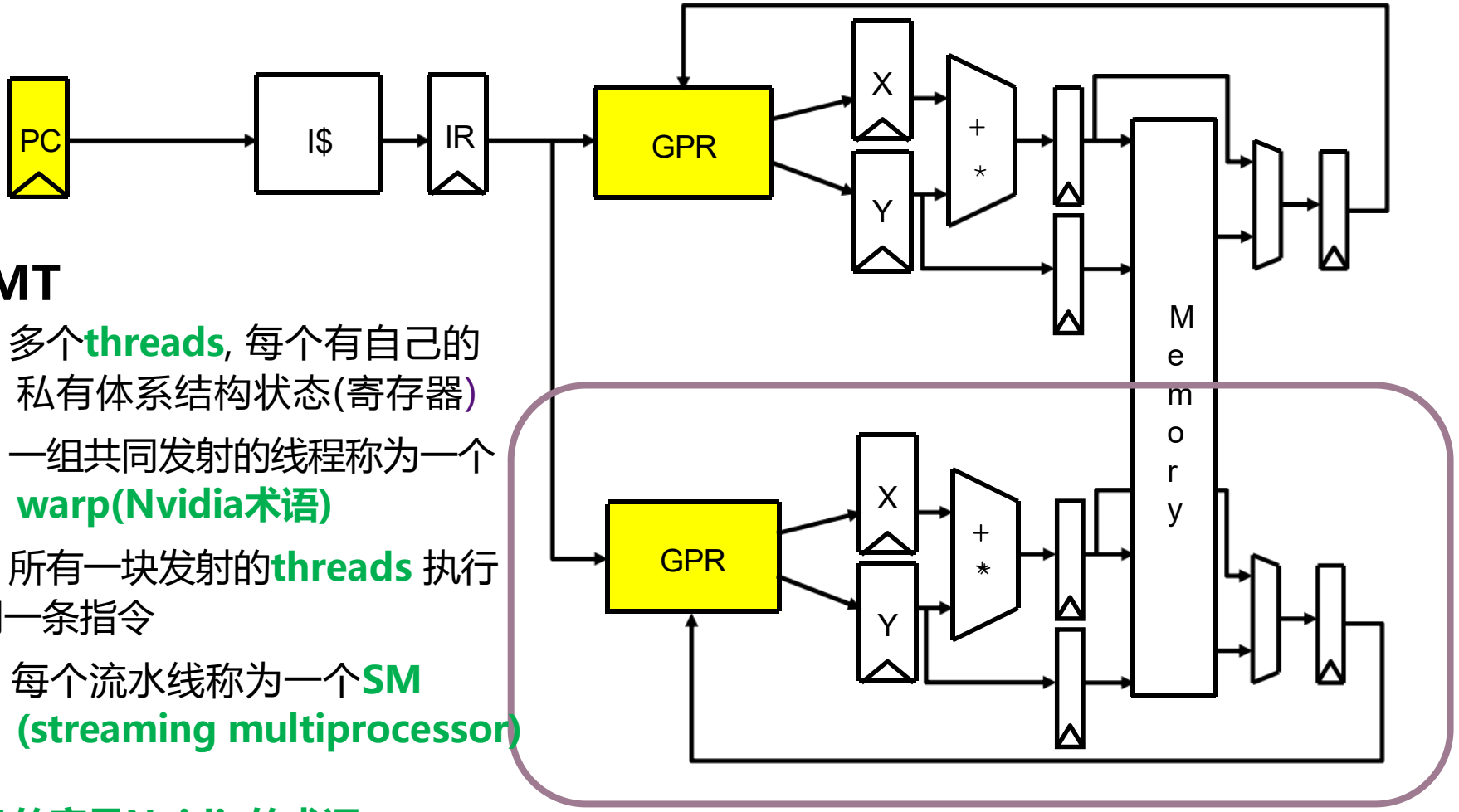
Idea #2:
多个核心共用一条指令, 将管理指令流的成本/复杂性分摊到多个运算单元上(PE)

SIMT(单指令多线程)

单指令多线程 (Single Instruction Multiple Thread)



单指令多线程 (Single Instruction Multiple Thread)



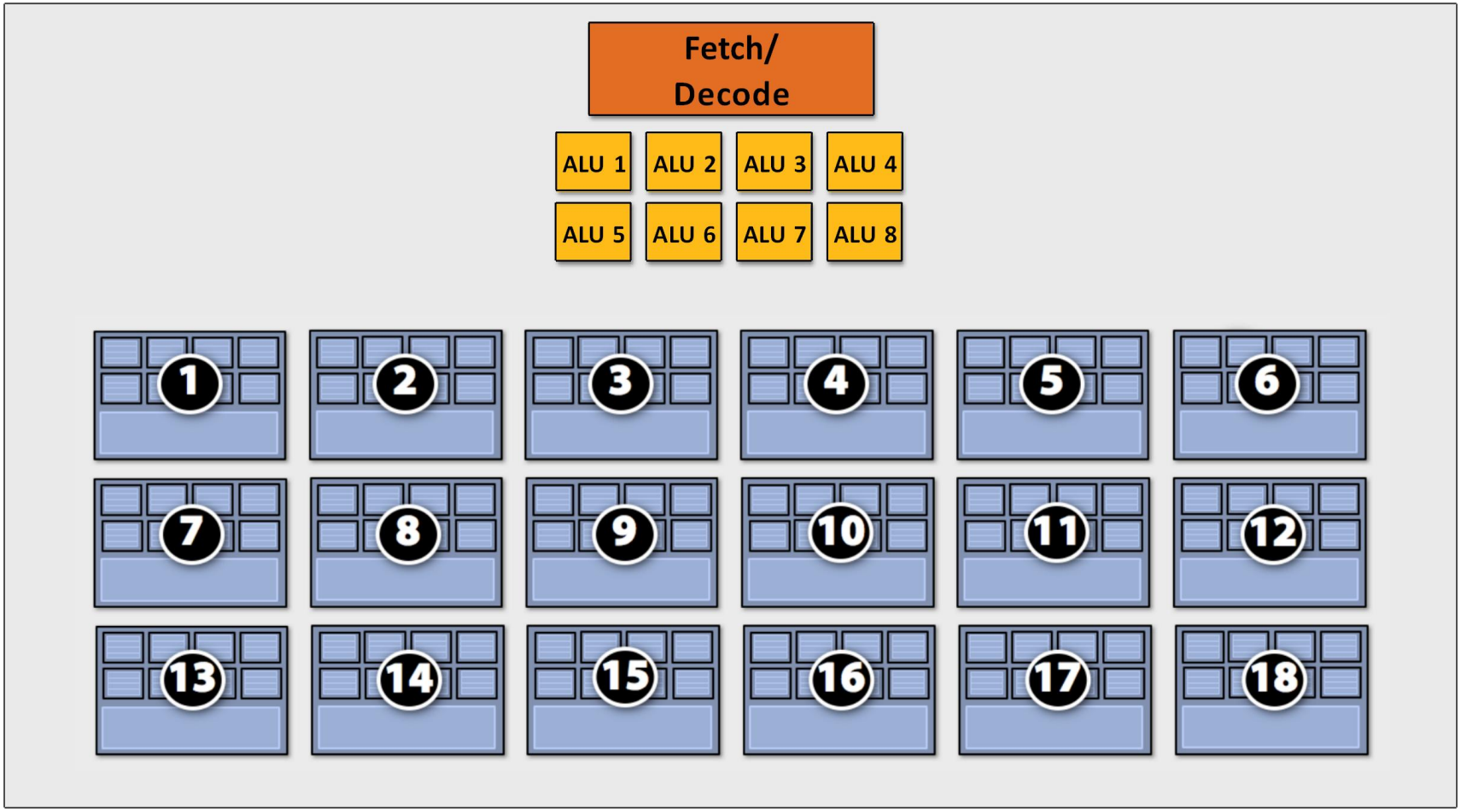
SIMT

- 多个**threads**, 每个有自己的私有体系结构状态(寄存器)
- 一组共同发射的线程称为一个**warp(Nvidia术语)**
- 所有一块发射的**threads** 执行同一条指令
- 每个流水线称为一个**SM (streaming multiprocessor)**

绿色的字是Nvidia的术语

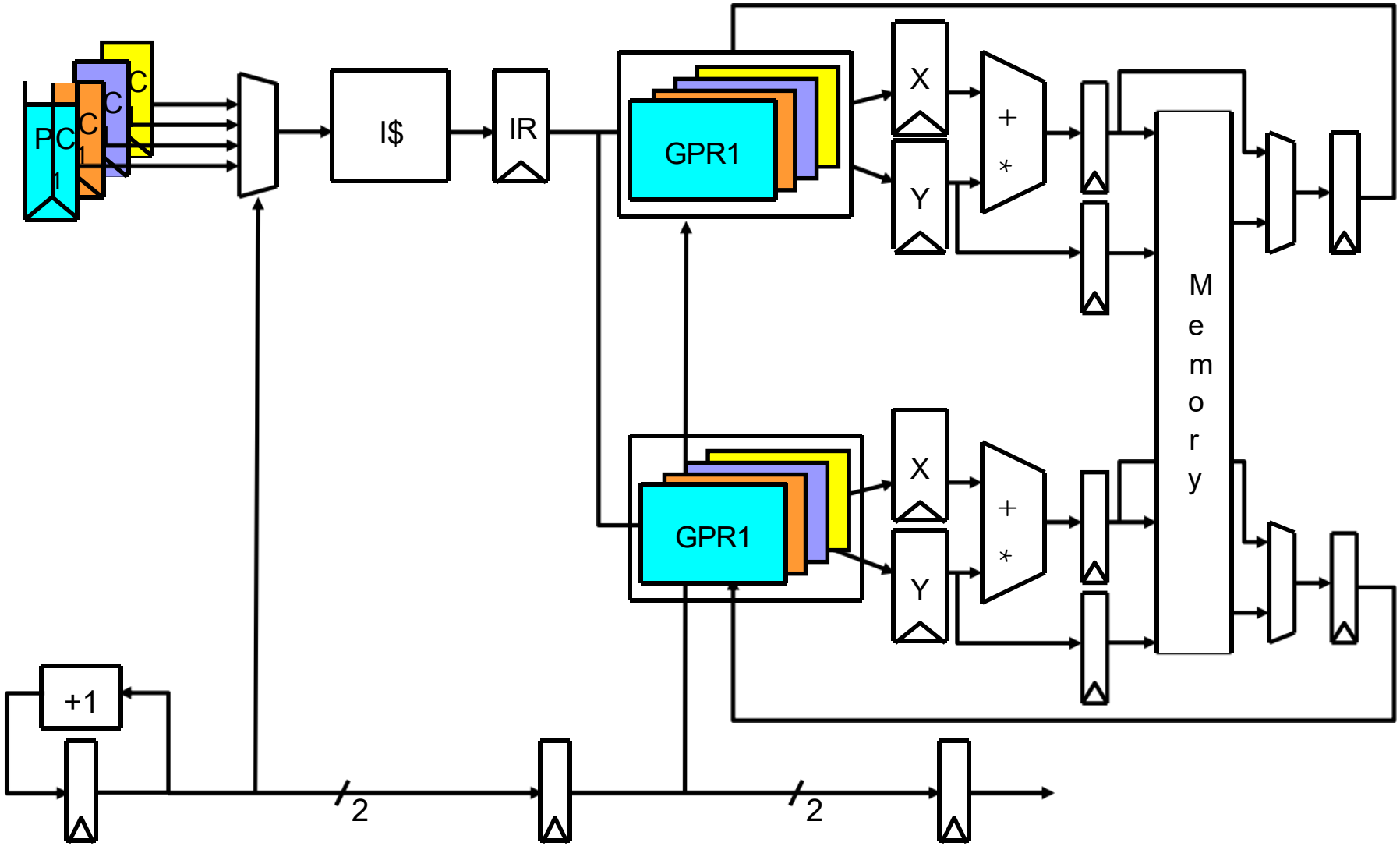
GPU 设计思路: (3) 同时驻留大量线程

Idea #3:
在单核心上维护远多于执行单元的线程数,以实现细粒度线程调度掩盖高延迟操作



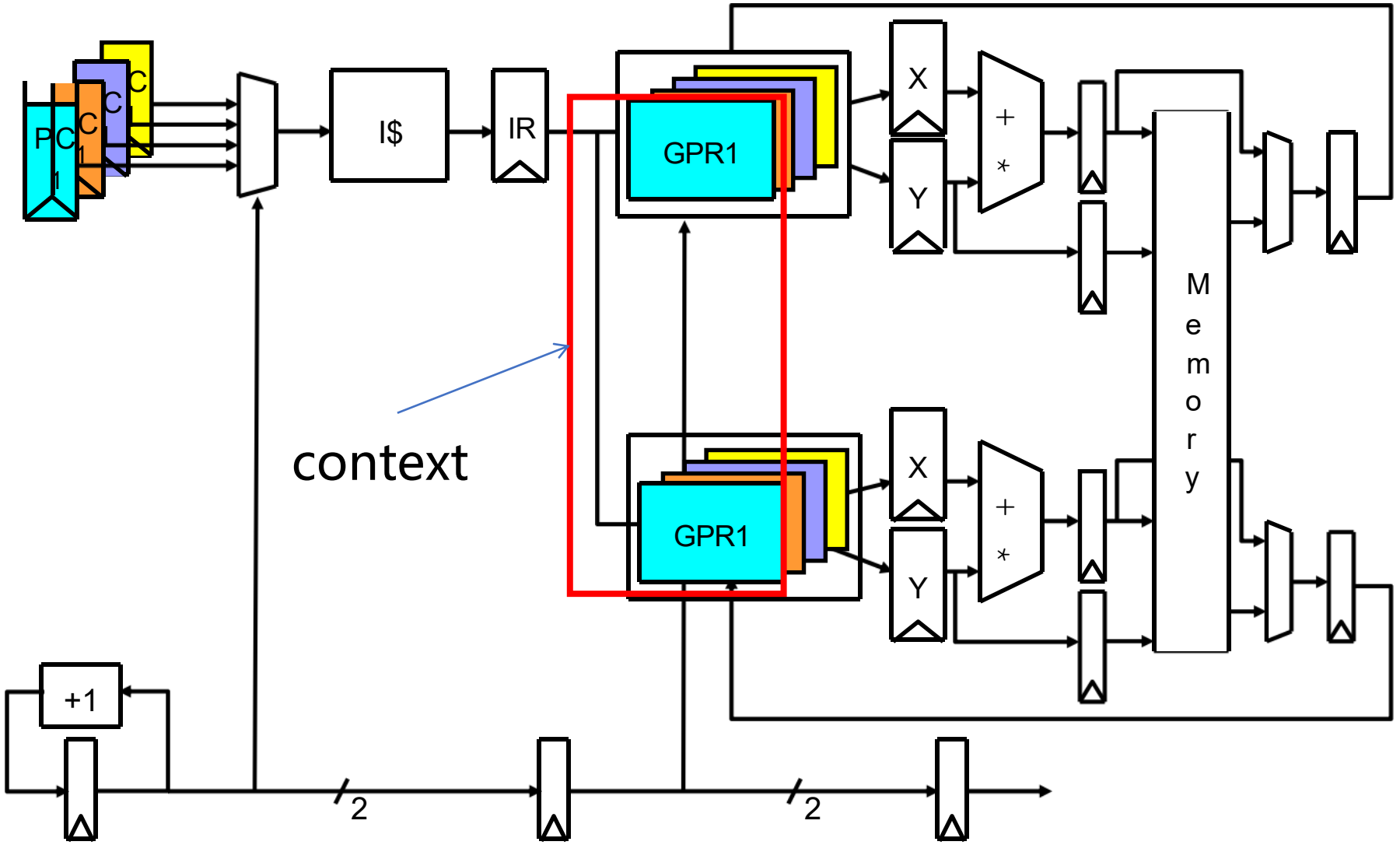
多线程+单指令多线程

(Multithreading +Single Instruction Multiple Thread)



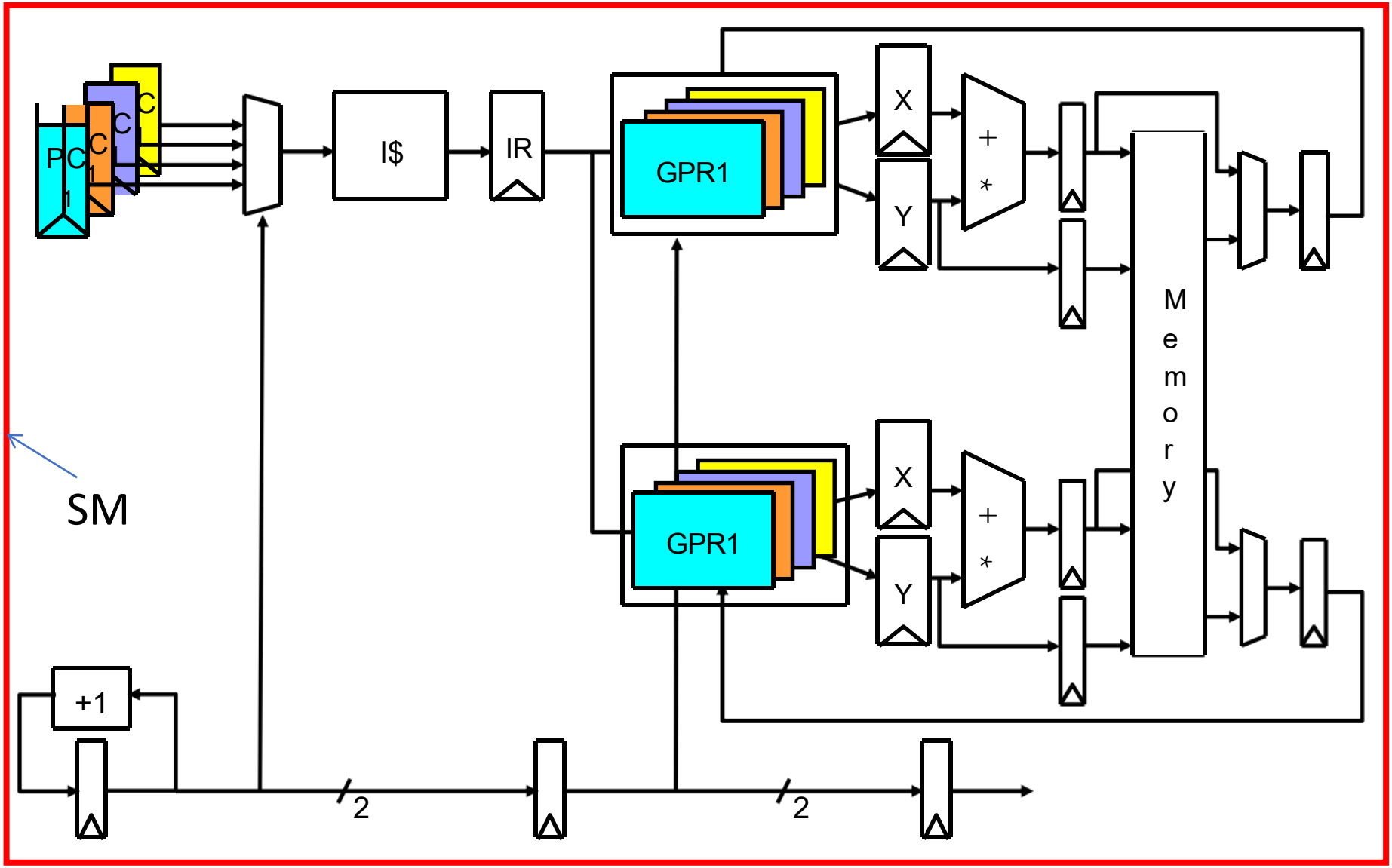
多线程+单指令多线程

(Multithreading +Single Instruction Multiple Thread)



多线程+单指令多线程

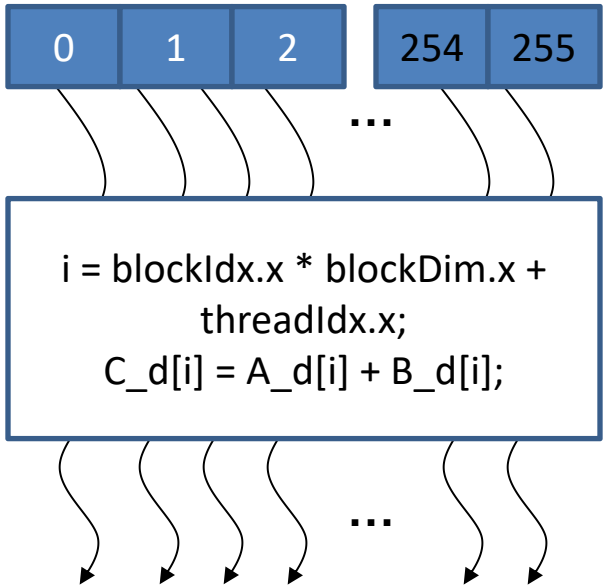
(Multithreading +Single Instruction Multiple Thread)



GPU 线程调度

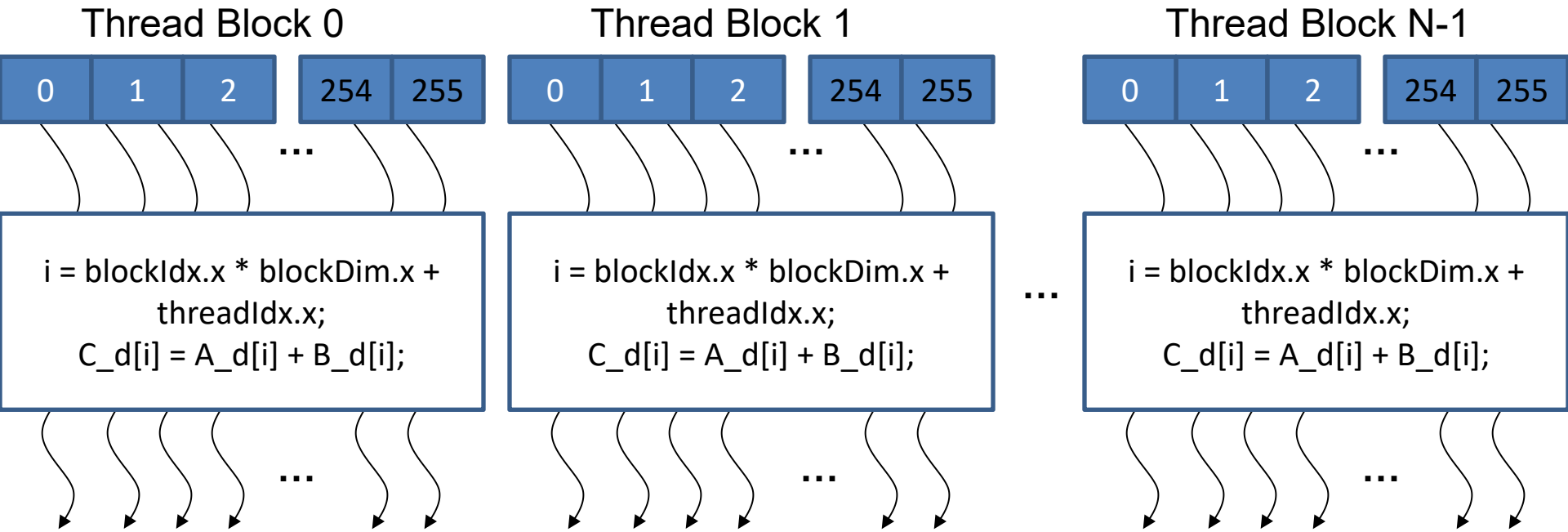
GPU 线程

- 一个GPU程序对应一个**device kernel**
- Kernel以一组线程的方式执行(**称为Grid**)
- 所有线程执行相同的kernel代码
- 每个线程使用自己的编号，计算不同数据



线程块 (Block)

- 一个Grid的所有线程分成若干块(**blocks**)
- **Block内线程协同计算：**共享内存、原子操作、同步机制
- Block间不能协作



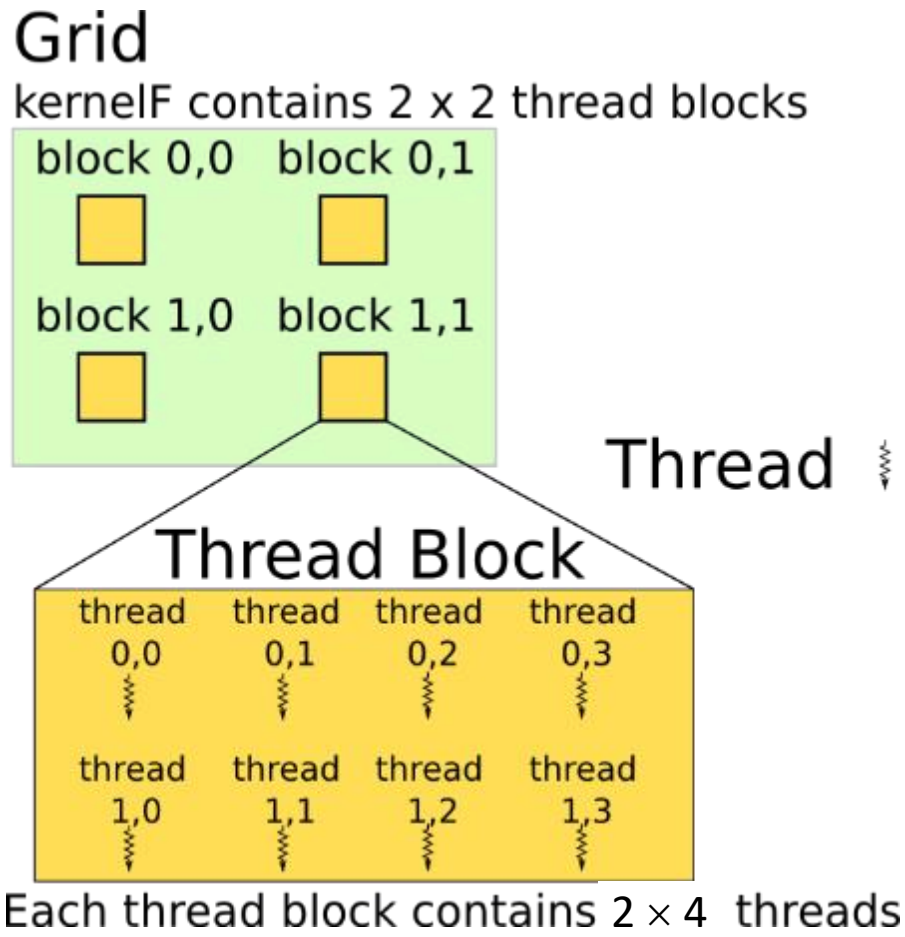
线程和Block编号

- Block和thread采用**多维编号**(便于映射到数据)

Kernel包含2x2个blocks

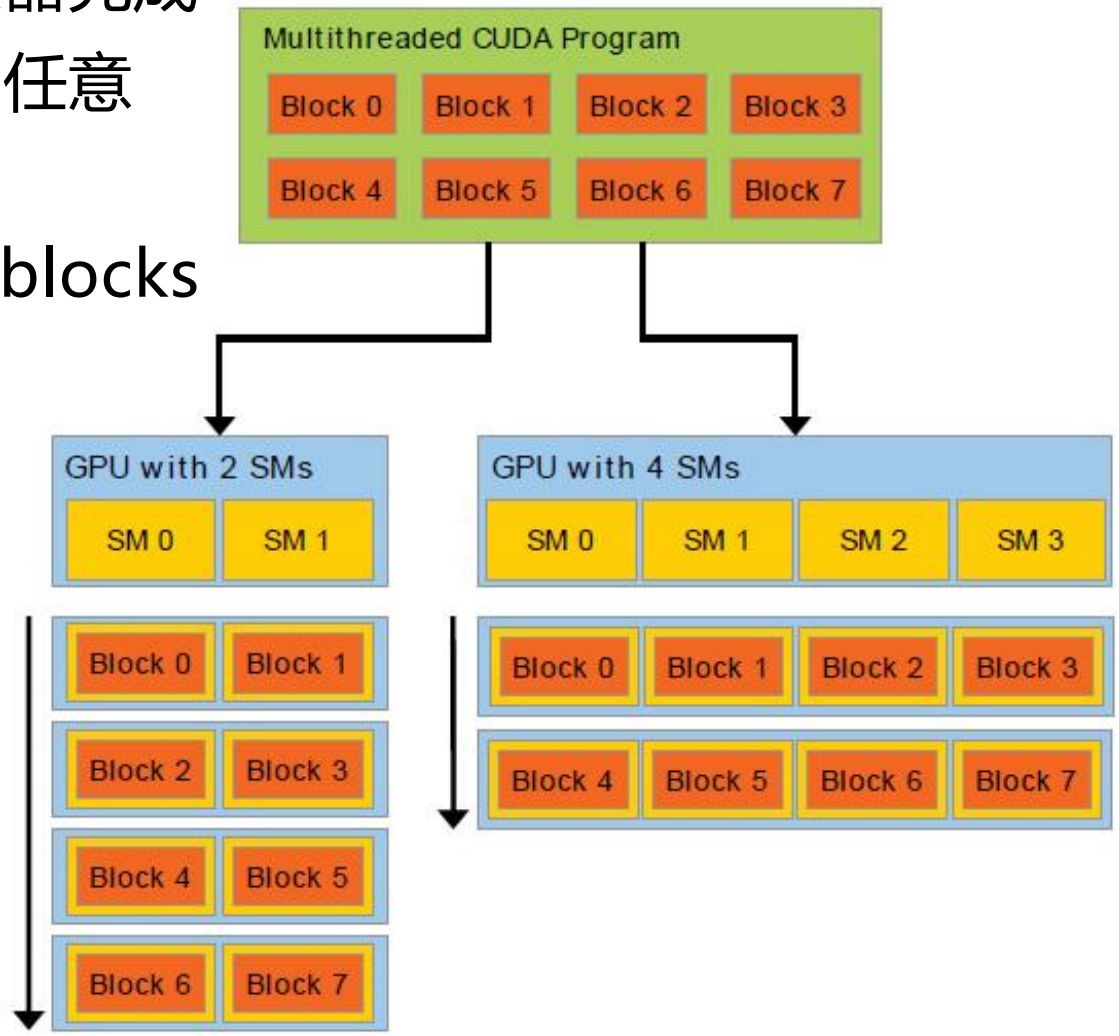
每个block包含2x4个threads

```
kernelF<<<(2,2),(2,4)>>>(A);
__device__    kernelF(A){
    i = blockDim.x * blockIdx.x
        + threadIdx.x;
    j = blockDim.y * blockIdx.y
        + threadIdx.y;
    A[i][j]++;
}
```



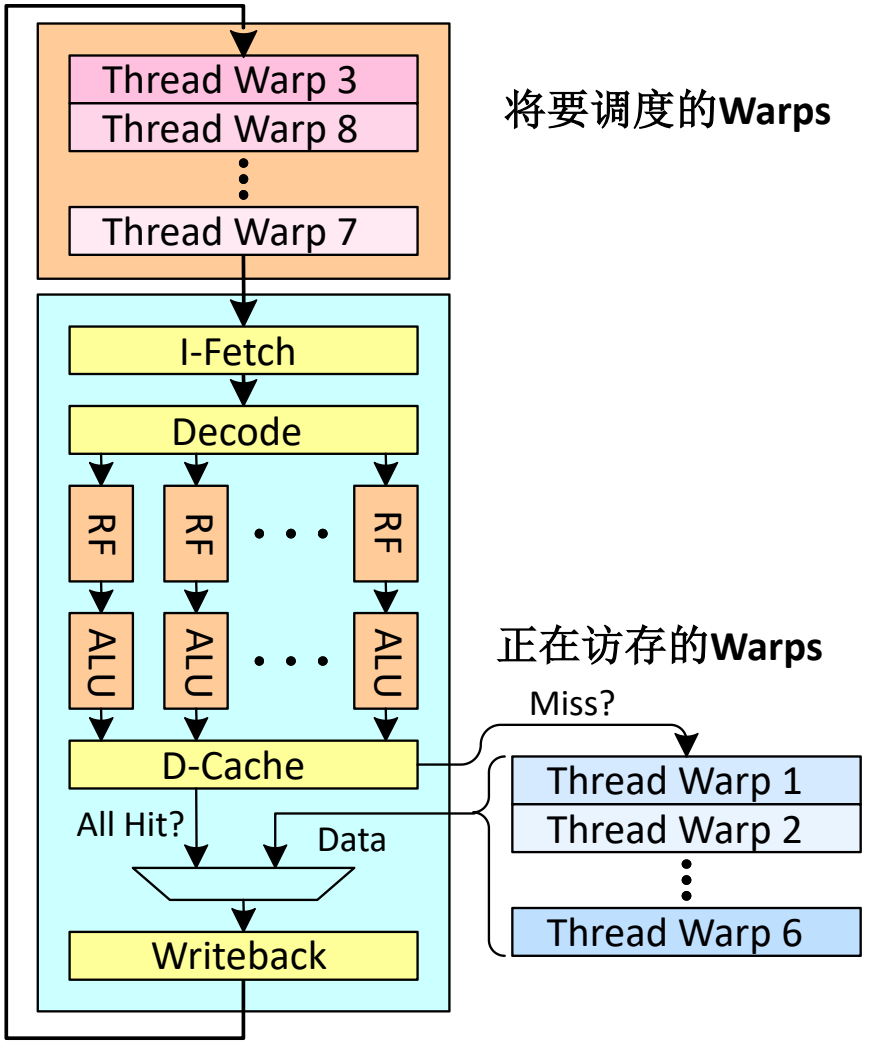
Block 调度

- block调度由硬件调度器完成
- 每个block可以调度到任意SM上
- 每个SM上可驻留多个blocks
- 多组context



线程调度

- block调度到SM之后，以warp为单位进行调度执行
- 一个block被划分为多个warp
- 每个warp中的线程并发执行，运行同一条指令，但处理不同的数据
- 多个warps交替执行(每个cycle选一个warp)
  - ✓ fine-grained multithreading，但以warp为单位

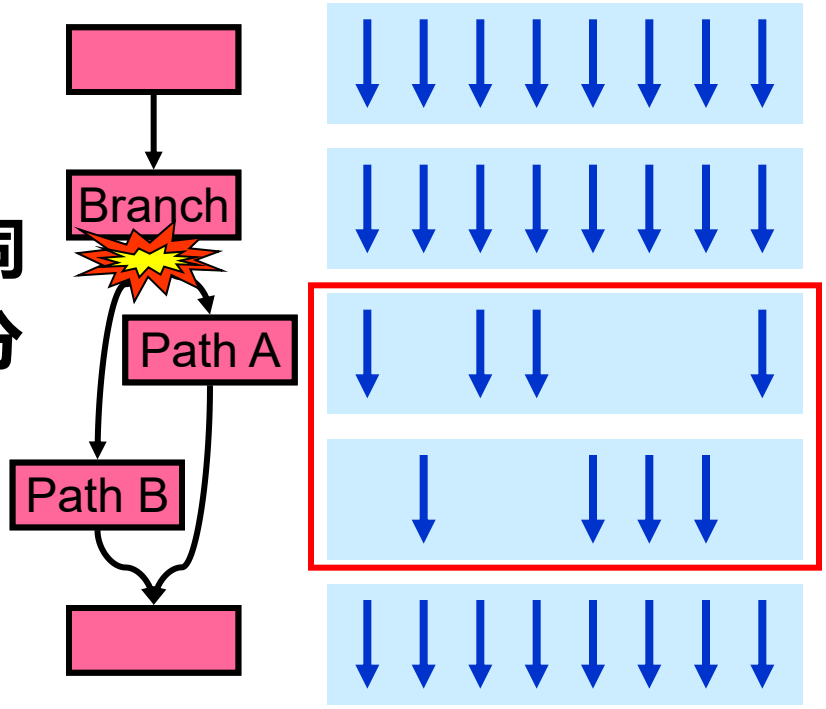


GPU架构关键问题

GPU的控制流问题

- GPU采用SIMD流水线以节省控制逻辑的面积
  - 将线程分成组(warps)

- 当warps内的线程分支到不同的执行路径时，会发生分支分歧（**branch divergence**）

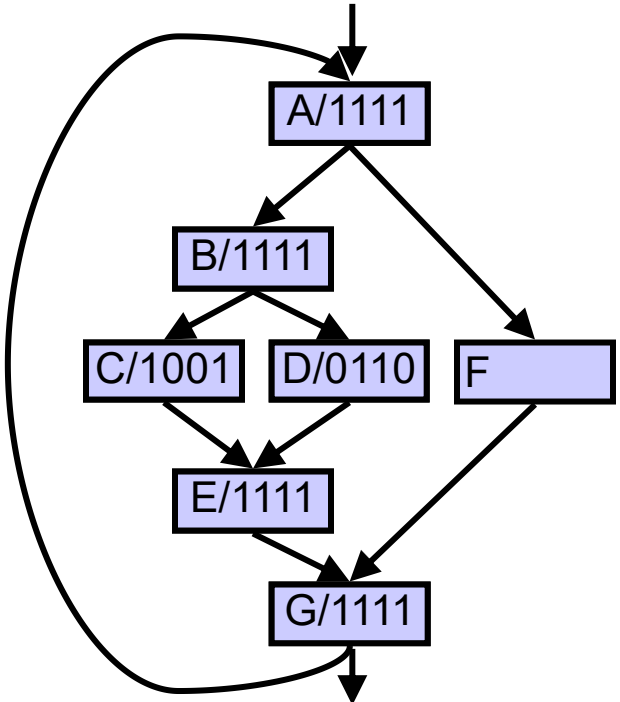


本来warps内的线程执行相同的指令，分支分歧后线程执行的指令不一样！

处理分支分歧

- **每个warp用stack存储不跳转分支的PCs和掩码**
- **遇到分支时**
  - 将当前掩码入栈
  - 将不跳转分支的掩码和PC入栈
  - 设置当前掩码为跳转分支的掩码
- **分支执行完时**
  - 将不跳转分支的掩码和PC出栈，并执行分支
- **不跳转分支执行完时**
  - 将分支之前的源掩码弹出
- 如果一个分支的掩码都是0，跳过该代码块

分支分歧解决方案



Stack

Reconv. PC	Next PC	Active Mask
-	E	1111
E	D	0110
E	C	1001