单线程=>多线程：并发与并行化

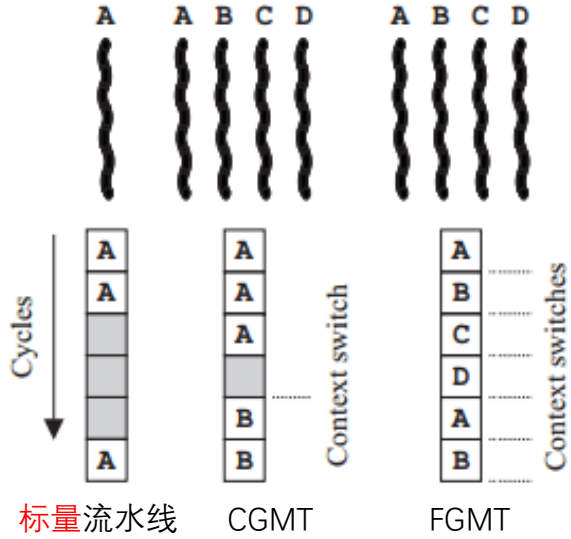
同步指令$2.11
硬件多线程$6.4
多核$6.5，CC（$5.10、$5.12），MC$5.14
GPU$6.6（HMT）

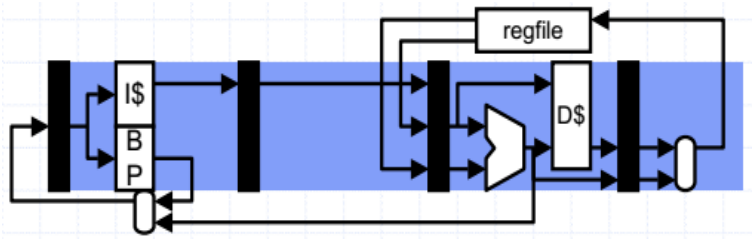
内容：并发与并行系统架构，TLP

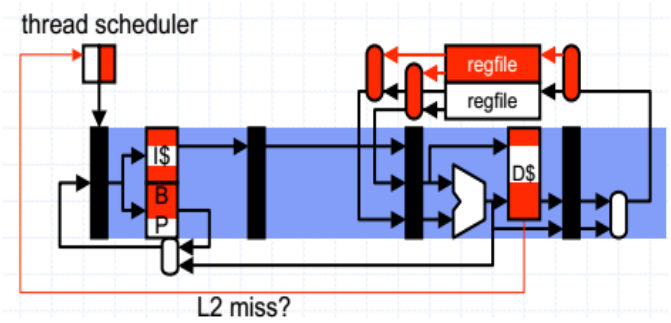
- 从单线程到多线程的需求
  - 功能，性能（上下文切换，吞吐率，空间换时间），【$4.11“多发射/超标量”属ILP技术，单线程内】
  - 挑战：race-condition（data-race，*抢占（调度）*），consistency（PO与EO），coherence（多copy）
- 硬件多线程架构HMT：$6.4
  - 多线程共享功能单元
- 线程同步操作
  - 原子指令：ll-sc，amo，$2.11
    - 数据竞争（data race，一种race condition）：共享数据互斥访问
    - 任务协作：生产者-消费者（flag），会合点（rendezvous，meeting point，barrier）
- RV多核架构：$6.5，hart
  - Cache Coherence：$5.10、$5.12
  - Memory Consistency：$5.14（fence，PO）
- GPU：$6.6，基于HMT架构

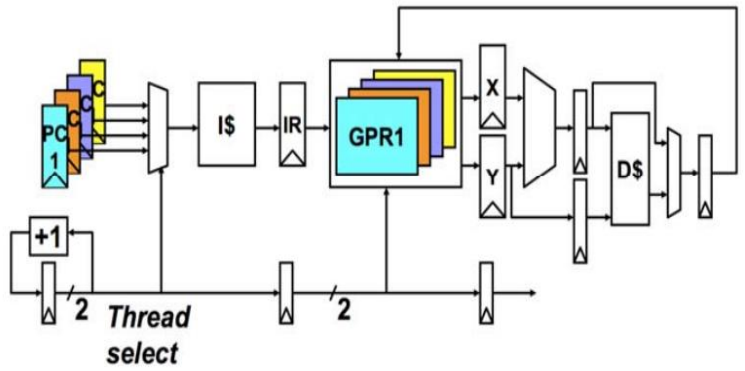
标量流水线HMT技术（硬件调度）：MIMD，$6.4

- 每个cycle一个线程
  - 快速切换：多份线程上下文（PC，RF，stack）
- coarse-grain multithreading：Block Multithreading
  - 隐藏长时延事件【如L2 miss】=>线程切换，下中
- FGMT：每个CC切换线程，RR
  - 隐藏短延时（RAW）、长延时事件，下右

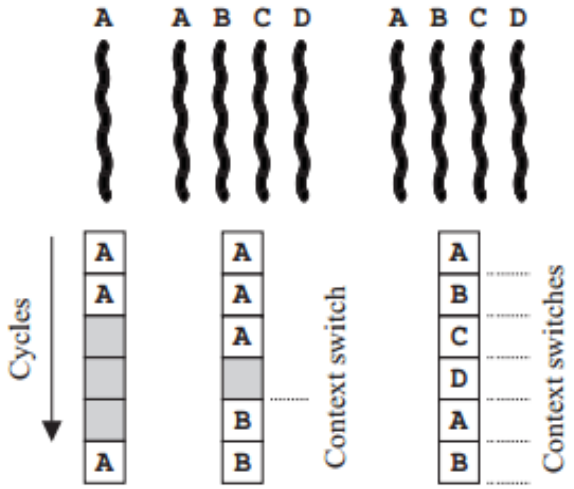








FGMT：消除数据依赖和分支依赖【短时延事件】



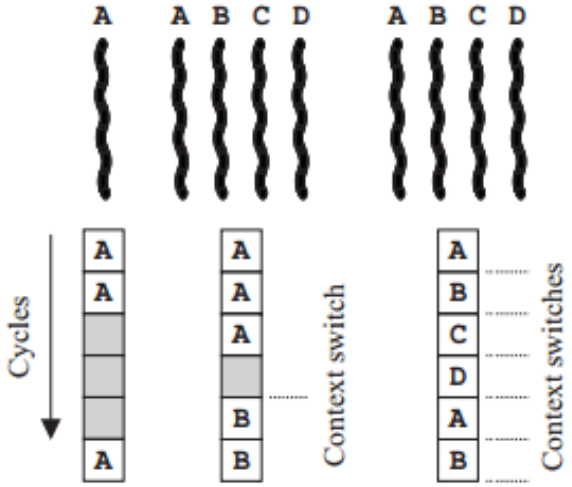
TID	Addr.	Inst.	Cycle							
			1	2	3	4	5	6	7	8
0	0x00	BR 0x0C	F	D	E	M	W			
0	0x04	I		F	D	-	-	-		
0	0x08	I			F	-	-	-	-	
0	0x0C	I				F	D	E	M	W

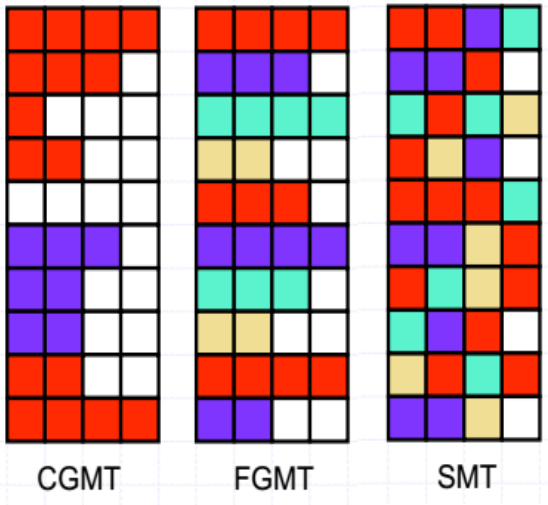
TID	Addr.	Inst.	Cycle							
			1	2	3	4	5	6	7	8
0	0x00	BR 0x0C	F	D	E	M	W			
1	0x30	I		F	D	E	M	W		
2	0x60	I			F	D	E	M	W	
3	0x90	I				F	D	E	M	W
0	0x0C	I					F	D	E	M

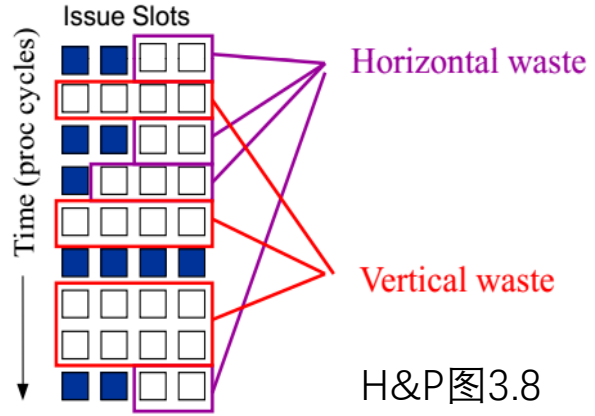
- 细粒度多线程【右下】：提升了总吞吐率，但增加了单线程延迟
  - 下例：0x0C指令晚了一个cycle?

超标量流水线HMT：SMT

- 提升吞吐率：多发射，动态调度（消除hazard，隐藏延时）
- 例：超标量流水线（4个ppl），4个线程，类似RV图6-5
  - CMT/FMT：每个cycle在一个线程中取指，吞吐率受线程内的ILP限制
  - SMT：每个cycle可在多个线程中取指，充分利用ILP和TLP
- CMT/FMT可标量或超标量，SMT必须超标量！



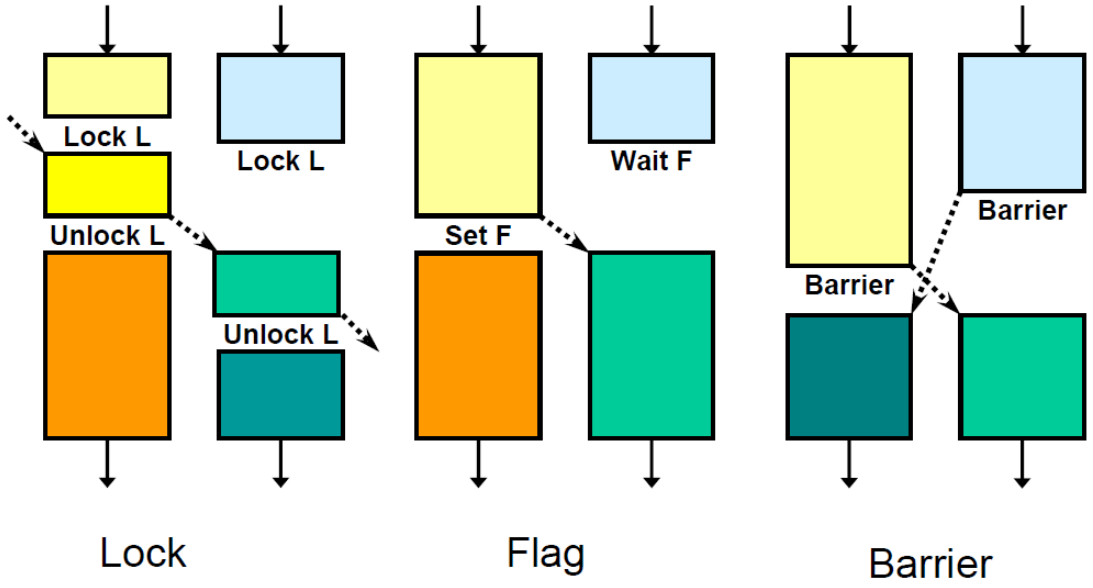




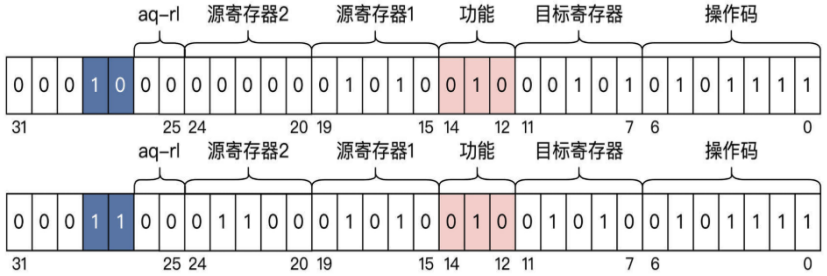
H&P图3.8

Different Synchronization Ops：两个线程

- Data-Race
  - 并发访问同一data，其中至少有一个write，且未串行化
  - 临界区：SWMR
    - lock = 串行化
- 协作
  - 生产者-消费者：flag，序
  - 会合点：barrier
- 同步协议：信号量（共享变量，aq/rl）
  - 原子性：CAS，RMW，lr-sc（ = CAS）
  - 同步点：aq，rl
- 单CPU，多CPU



RV原子指令1：LR/SC指令，RV$2.11



- LR/SC指令：不同 hart 之间同步【针对DR/CS】
  - “保留”：维护LR-SC原子性，如同步变量缓存行的保留状态
- lr.{w/d}.{aqrl} rd，(rs1);
  - {可选内容}：w（32位）、d（64位），aqrl为访存顺序约束，一般使用默认的【下页】
  - rd目标寄存器：锁值
  - rs1源寄存器1：内存地址（锁变量），加载后会设置当前CPU hart读取该地址的保留位
- sc.{w/d}.{aqrl} rd，rs2，(rs1)；先判断rs1对应地址的保留位有没有被设置
  - rd目标寄存器，成功则为0，否则为1（非0）
  - rs1源寄存器1，含保留位，与lr相同
  - rs2源寄存器2，被写入（rs1）
- 指令格式：R-type（图2.18）
  - 上图lr指令，下图sc指令

只有满足以下四个条件，SC才能执行成功：

1. LR和SC访问相同的地址。
2. LR和SC之间没有任何其它的写操作访问同一地址（来自任何一个hart）。
3. LR和SC指令之间没有任何中断与异常发生。
4. LR和SC指令之间没有执行MRET特权指令。

.aqrl：访存序约束标记

- 便于在原子指令上施加额外的内存访问顺序限制
  - .aq，约束为“acquire序”
  - .rl，约束为“release序”
  - .aqrl，约束为“顺序一致（Sequential Consistency，SC）序”

Acquire	Release	含义
0	0	没有顺序限制
0	1	该指令前序所有访问存储的指令的结果必须在该指令执行之前被观察到
1	0	该指令后序所有访问存储的指令必须等该指令执行完成后才开始执行
1	1	该指令前序所有访问存储的指令的结果必须在该指令执行之前被观察到，该指令后序所有访问存储的指令必须等该指令执行完成后才开始执行

RV原子指令1：LR/SC指令，lr=ll，RV$2.11

- 功能：实现原子操作RMW和锁
- lr：装入同步变量（a0）到t0
  - 其后可跟任何对t0的操作指令，即进行RMW的“M”操作
- sc：写回同步变量
  - iff：在lr之后没有其他hart对该单元或缓存块进行写
  - 否则改写a0作废，重新尝试lock
- 用lr-sc实现lock
  - lr本身不是加锁，sc也不是解锁
    - lr完成并不意味着获得了排他访问权
    - 成功的lr-sc：保证两者之间没有发生对同步变量的写
      - 两者间的操作只是针对同步变量，失败时不可见
  - lock()：同步变量（a0），如图
  - unlock()

    unlock: st (a0), 0

    ret
- 多个处理器可能同时执行lock

```
1  .globl lrsc_ins
2  #a0内存地址
3  #a1预期值
4  #a2所需值
5  #a0返回值，如果成功，则为0！否则为1
6  lrsc_ins:
7  cas:
8      lr.w t0, (a0)        #加载以前的值
9      bne t0, a1, fail     #不相等则跳转到fail
10     sc.w a0, a2, (a0)    #尝试更新
11     jr ra                #返回
12 fail:
13     li a0, 1             #a0 = 1
14     jr ra                #返回
15
```

LR/SC指令实现：一种基于Cache的方案

- 执行LR时：将对应Cache行标记为“保留”状态；加载数据到寄存器
  - 【多核】：需通过总线或 Cache 一致性协议确保全局可见性
- 执行SC时：检查Cache行是否仍处于“保留”状态
  - 是 → 执行store，清除保留标记，返回成功；否 → 返回失败
  - 无论成功失败，都通过总线发送 invalidate 消息，确保其他CPU的Cache副本失效
- 保证顺序一致性：LR 和 SC 之间的指令不能重排序
  - LR 和 SC 之间插入隐式的内存屏障fence，防止指令重排序
- 异常处理：在发生中断、异常或上下文切换时需清除保留状态
- 一般不允许“LR/SC对”嵌套，第二次 LR 会清除前一次的保留状态

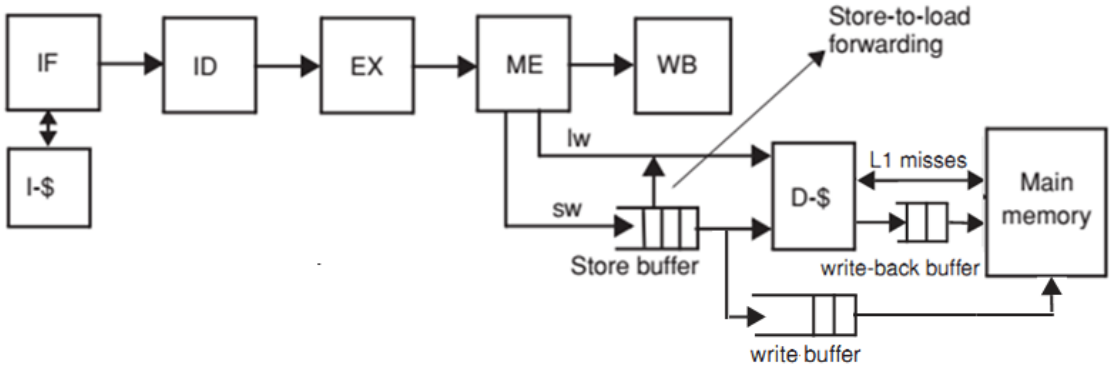
原子指令2：原子内存操作AMO，RV$2.11

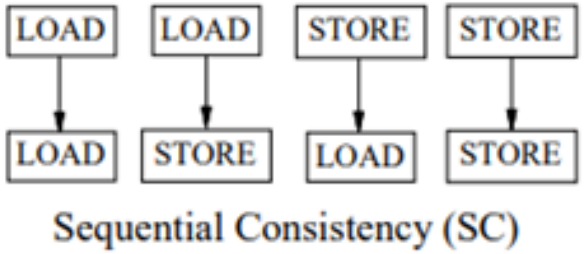
- 对于单个变量的单次操作，使用LR/SC开销大
- 每个AMO指令完成的操作不可分割，不能被外部事件打断
  - 含：原子交换指令、原子加法指令、原子逻辑指令、原子取大小值指令
    - 例：amoswap.{w/d}.{aqrl} rd,rs2,(rs1)
      - 把内存地址rs1中的数据加载到rd寄存器中，然后把rs2中的数据写入到rs1指向的内存单元中
- X86：支持CAS（实现lock）和“原子取后加”
  - PentiumPro之前通过锁Bus阻止其他master（DMA、CPU核）的内存访问，实现lock
  - 从PentiumPro开始
    - LOCK实现：只会阻塞其他CPU核对相关的缓存块访问，而不用锁住整个总线。
      - 只需要保持cacheline的M和E状态，就可以阻止其他cpu核对该块内存的修改
    - CAS实现：只须阻塞其他cpu核对相关内存的缓存块的访问，同样无需锁住总线。
- MIPS、ARM、RV：支持AMO和LL/SC【比实现CAS等简单，$2.11】
  - 基于LL/SC可以实现AMO、CAS等原子操作，不会出现CAS中的ABA问题

同步指令fence：程序序 = 执行序?

- 顺序语义：程序序，单线程，单核
  - 程序执行的“正确性”
    - 单变量读写：按程序序，对任一变量的读都能返回其最新的写值。
    - 多变量读写【锁变量，普通变量】：按程序序
  - in-order执行架构：单周期/多周期/阻塞式流水线，执行序=程序序
  - OoO执行架构：指令调度，执行序≠程序序
    - 静态调度：编译序
    - 动态调度：分支预测，写加速（StoreBuf放松了store-load，TSO），Cache miss（非阻塞）
- fence指令：禁止乱序执行，访存顺序一致性

```
36  sub  x10, x4, x8
40  beq  x1,  x3, 16
44  and  x12, x2, x5
48  or   x13, x2, x6
52  add  x14, x4, x2
56  sub  x15, x6, x7
. . .
72  ld   x4, 50(x7)
```





fence示例：临界区的两种方法

- 可用amoswap【左】，或
- fence【右，加强】

```
    li          t0, 1        # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez        t0, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

```
1           sd          x1, (a1)     # Arbitrary unrelated store
2           ld          x2, (a2)     # Arbitrary unrelated load
3           li          t0, 1        # Initialize swap value.
4       again:
5           amoswap.w   t0, t0, (a0) # Attempt to acquire lock.
6           fence       r, rw        # Enforce "acquire" memory ordering
7           bnez        t0, again    # Retry if held.
8           # ...
9           # Critical section.
10          # ...
11          fence       rw, w        # Enforce "release" memory ordering
12          amoswap.w   x0, x0, (a0) # Release lock by storing 0.
13          sd          x3, (a3)     # Arbitrary unrelated store
14          ld          x4, (a4)     # Arbitrary unrelated load
```

RV的fence：针对所有指令，不仅访存？，$5.14

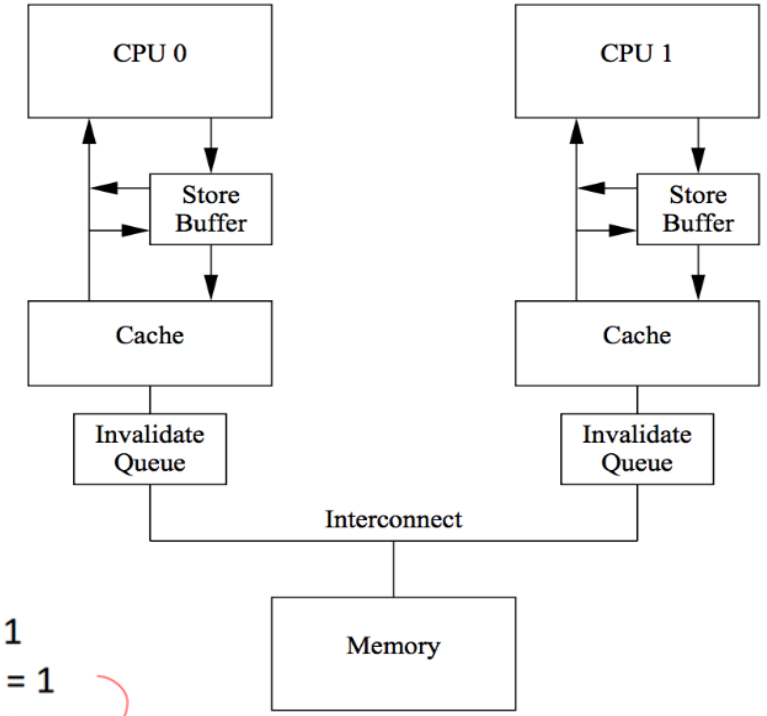
- 内存访问顺序（fence）强调的是同一个 hart 内的执行顺序
- fence [iorw], [iorw]：iorw分别表示fence要约束的前后指令的类型
  - PR/PW/SR/SW位：限制前导集/后续集中所包含指令的类型
    - FENCE RW,RW；相当于TSO
    - FENCE RW,W；之前的所有RW不能后移，之后的W不能前移。acquire语义
    - FENCE R,RW；之前的所有R【load】不能后移，之后的RW不能前移。release语义
    - FENCE R,R
    - FENCE W,W
  - 为了性能，FENCE可以进一步限制前导集和后续集为较小的内存访问集
- fence实现
  - 排空本地Write Buffer到主存
  - 通知其他CPU其缓存副本失效

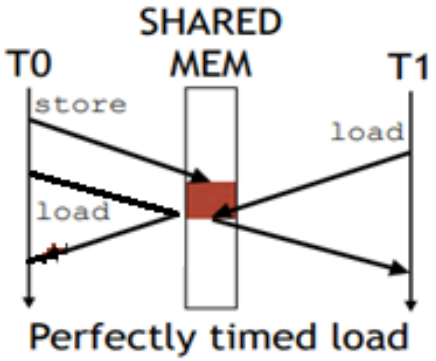
图5-47

Type	Mnemonic	Name
Mem. Ordering	FENCE.I	Instruction Fence
	FENCE	Fence
	SFENCE.VMA	Address Translation Fence

多核架构：SMP（共享内存，每核单线程）

- 程序员角度：“处理器看起来严格按程序序每次执行一条指令”
  - 访存操作立即且原子地执行，读操作返回最近的写值，【零时间、非重叠】
- memory coherence and consistency
  - Coherence：单个数据多个copy的一致性【I/O一致性】
    - Cache coherence：写传播，串行化
  - Consistency：多线程交错访存，不同变量
    - 访存序：global memory order
      - （普通/同步）访存序与程序序的一致性：SC > TSO > RC
      - “程序员需要知道编译器和处理器支持什么内存模型”：fence



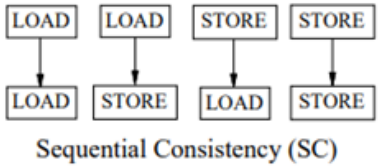


Initial state: x=0; y=0;	
Thread 0	Thread 1
x=1; r0=y	y=1; r1=x

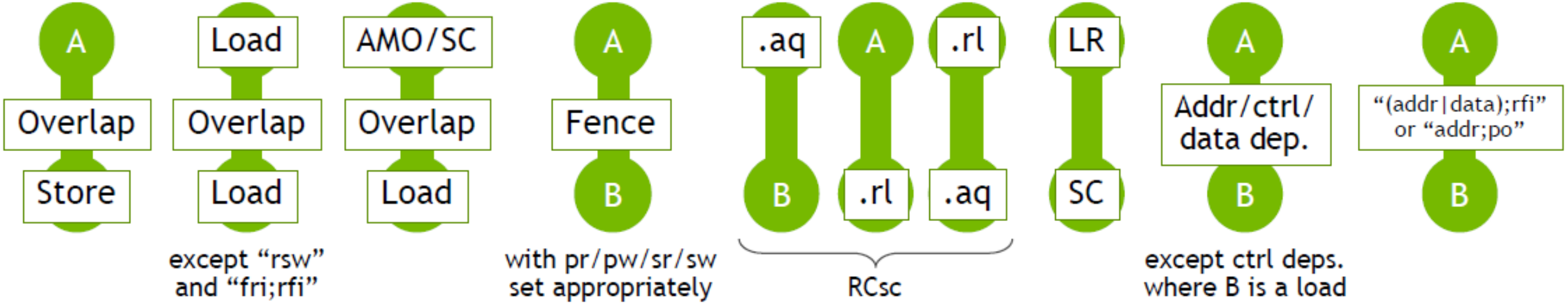
Initially A = B = 0

Proc 0	Proc 1
(1) A = 1	(3) B = 1
(2) print B	(4) print A

RVWMO：主要遵循RC模型，13条PPO规则、3条公理

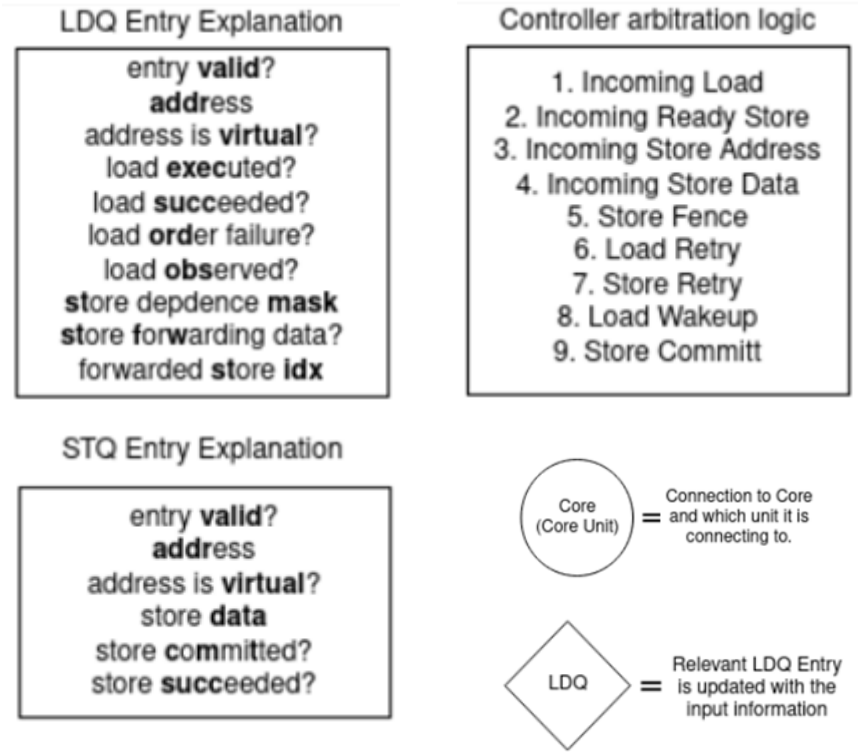


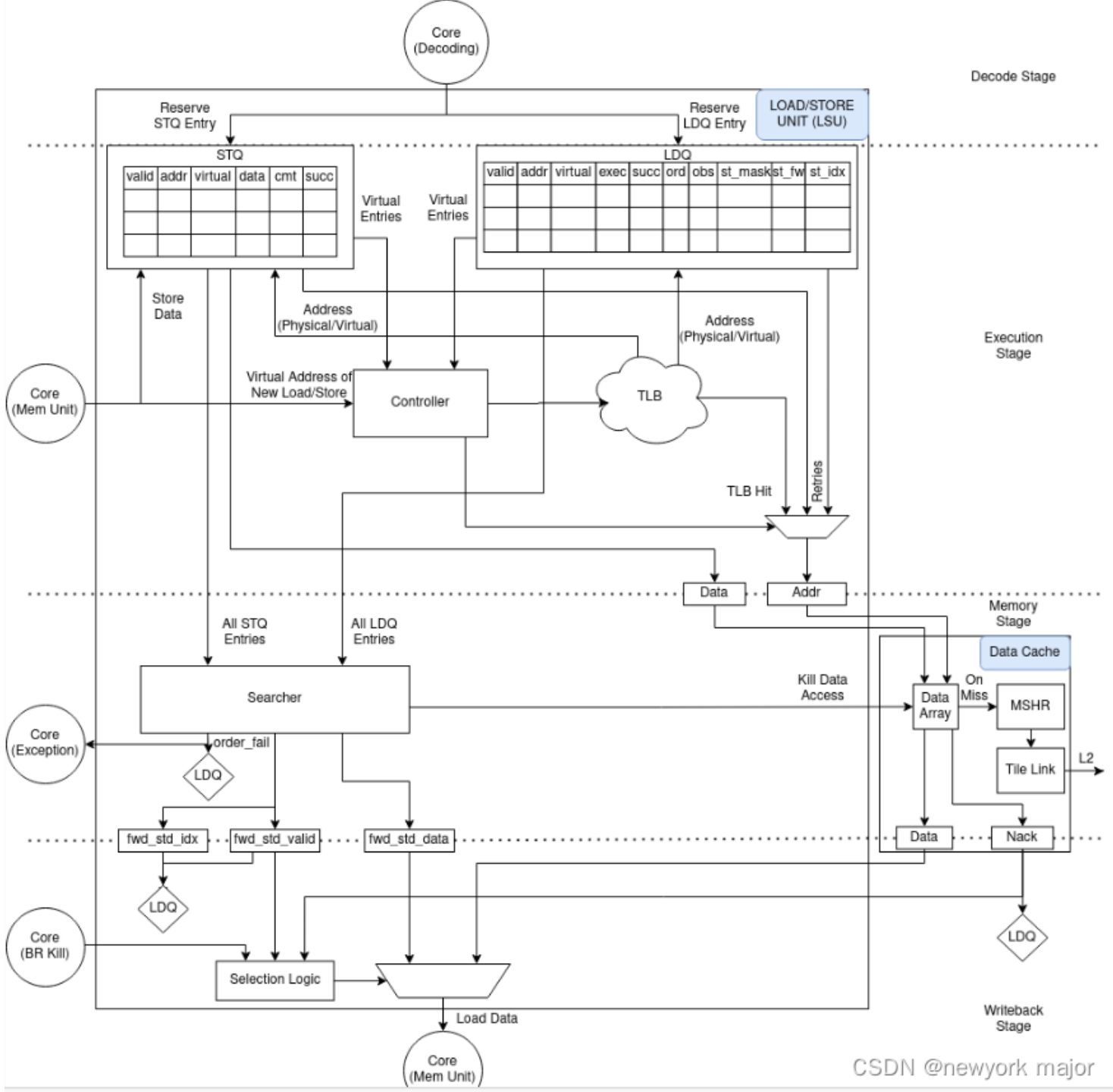
- PPO RULES：即“不允许乱序的情况”【对RV机器的要求】
  - 程序序中A在B之前，且符合以下模式之一，则在全局访存序中A必须在B之前
    - 地址重叠（1/2/3），同步指令（4/5/6/7/8）
    - 语法依赖（9/10/11）：指“地址/控制/数据”依赖
    - 流水线依赖（12/13）：前两条指令语法依赖，第三条为**相关读**或不知是否相关的**写**，【3有4无？】
- 三个Axiom：Load Value，Atomicity（一对），Progress（store最终可见）



LSU：BOOM架构

- 地址生成，支持OOO
- 实现store-load前推：Searcher
- 检测RVWMO**违例**：order failure





数据一致性问题

- 同一data（X）的多个copy
  - 写传播
  - 读写操作的原子性（写操作）
- 访存操作的全局序不确定性
  - P1的wr与P2的wr先后顺序，下左？
  - 事务串行化，下右
    - 串行化点：总线广播保证每个操作同时被其他节点看见

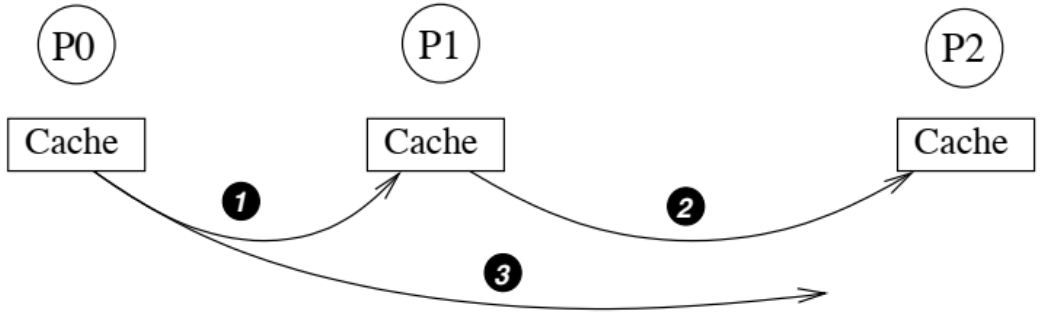
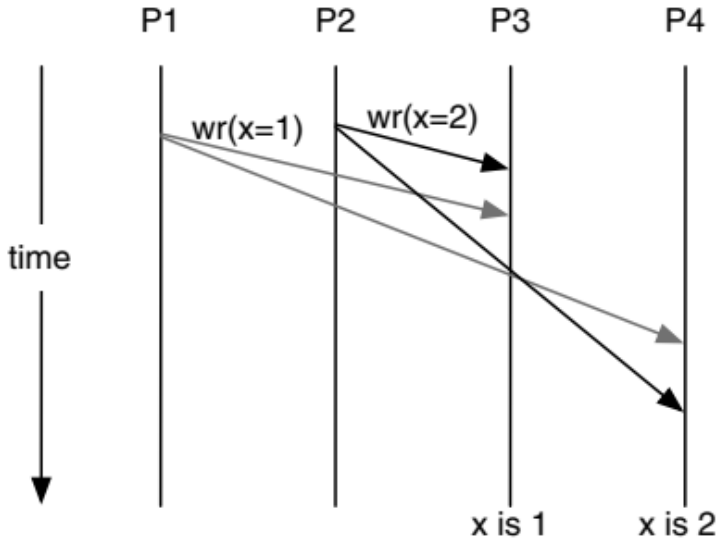
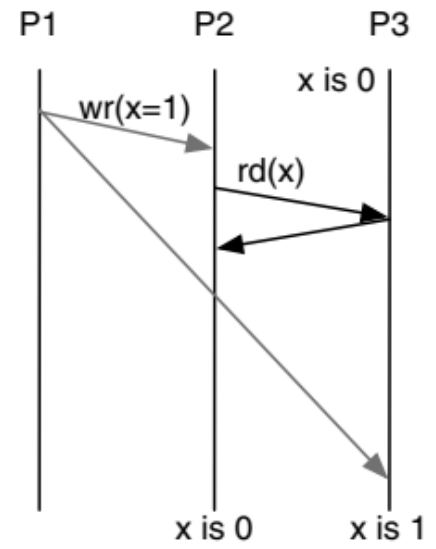


图5.39：写直达，T3时CPU_A写1，A$=1，而B$=0

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0		inv	inv	0
1	CPU A reads X	0	inv	0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1



汤孟岩图6-2　违反写操作串行化



违反读写操作串行化

Cache Coherence：$5.10，$5.12

- 嗅探协议snoopy，目录协议dir：使无效式，写更新式
- Snooping Protocols：a write invalidate protocol
  - 分布式一致性状态维护（VI，MSI，MESI）
    - 写传播：写操作独占访问【M】，并使其他副本无效【I，广播应答】
    - 写串行化（SWMR）：对同一data，写竞争时只允许有一个写【只能有一个M态的copy】

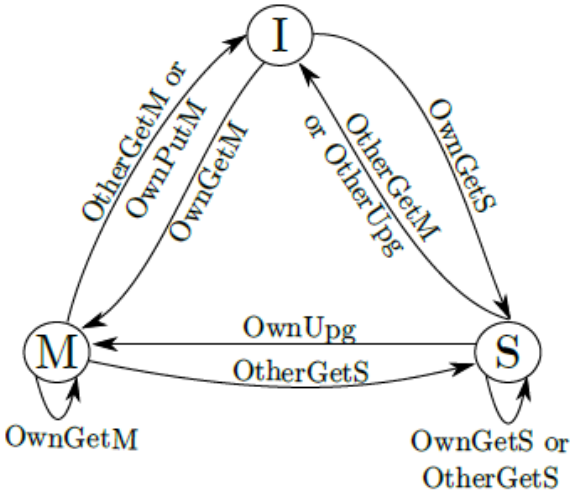


FIGURE 5.40 write-back，invalidation protocol，snooping bus

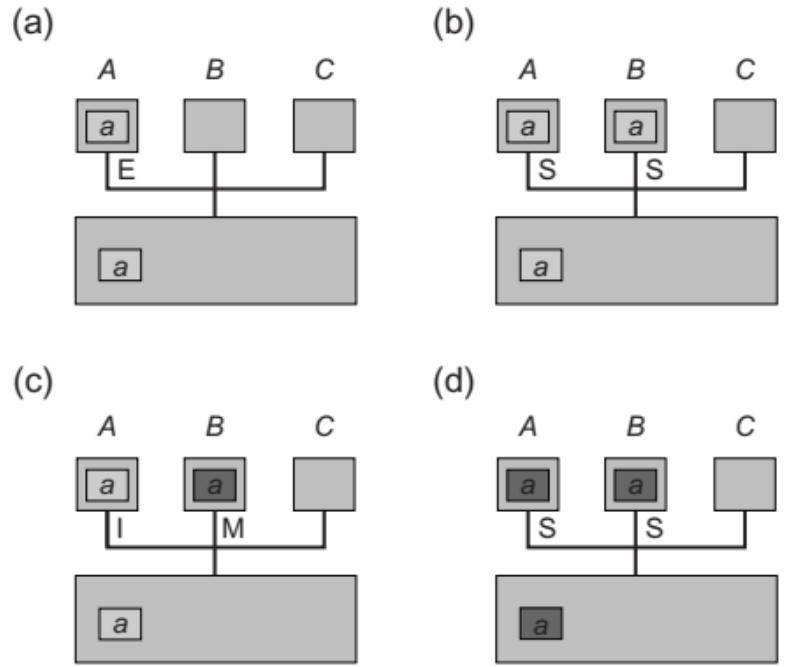
图5.39：写直达，T3时CPU_A写1，A$=1,而B$=0

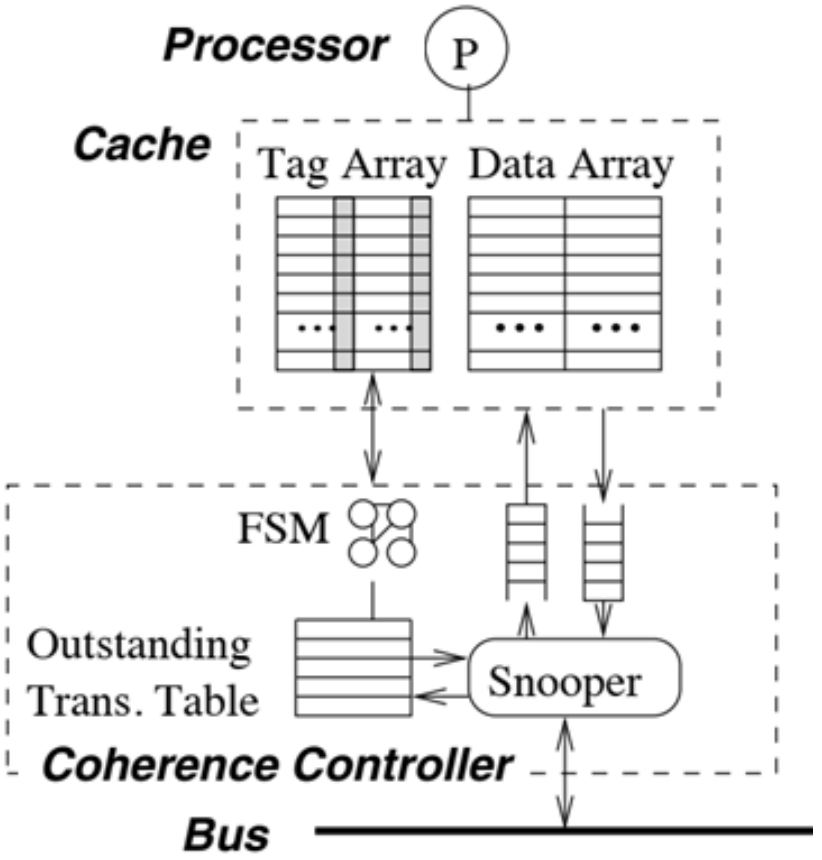
Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0		inv	inv	0
1	CPU A reads X	0	inv	0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
		inv	inv	0
CPU A reads X	Cache miss for X	0	inv	0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1	inv	0
CPU B reads X	Cache miss for X	1	1	1 写回

VI (写透写无效) => MSI => MESI =>MOESI =>

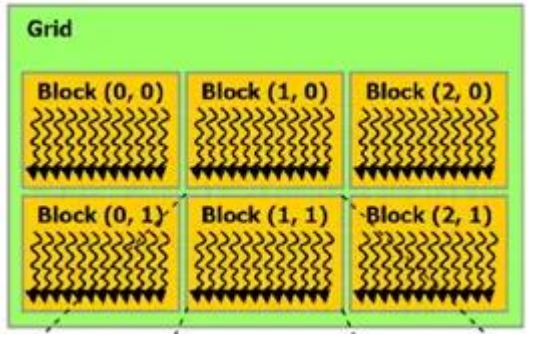
- MESI：减少总线通信
  - E：独占，a只在本Cache中。本CPU可任意读写
    - 状态迁移：发现其他CPU访问时须将E改为S。
  - 例：A读a，B读a，B写a，A读a





汤：图7-2

GPU：一个MIMD处理器，RV$6.6



- 数据级并行DLP：由众多HMT SIMD处理器核组成一个MIMD处理器
  - SIMD线程：仅由SIMD指令组成的线程，如指令1 → 数据A, 数据B, 数据C, 数据D；【同一条指令，4个元素】
- 存储器面向带宽（指令宽度）而非延迟，使用专用DRAM（HBM，die堆叠）
  - SIMD处理器本地内存，GPU全局内存：依靠FGMT技术隐藏访存长时延，而非Cache（新版GPU使用Cache）
  - 如图：设指令宽度32【32个元素】，每个数据32b
    - 每个通道执行一个元素计算，故一条指令须 2cc

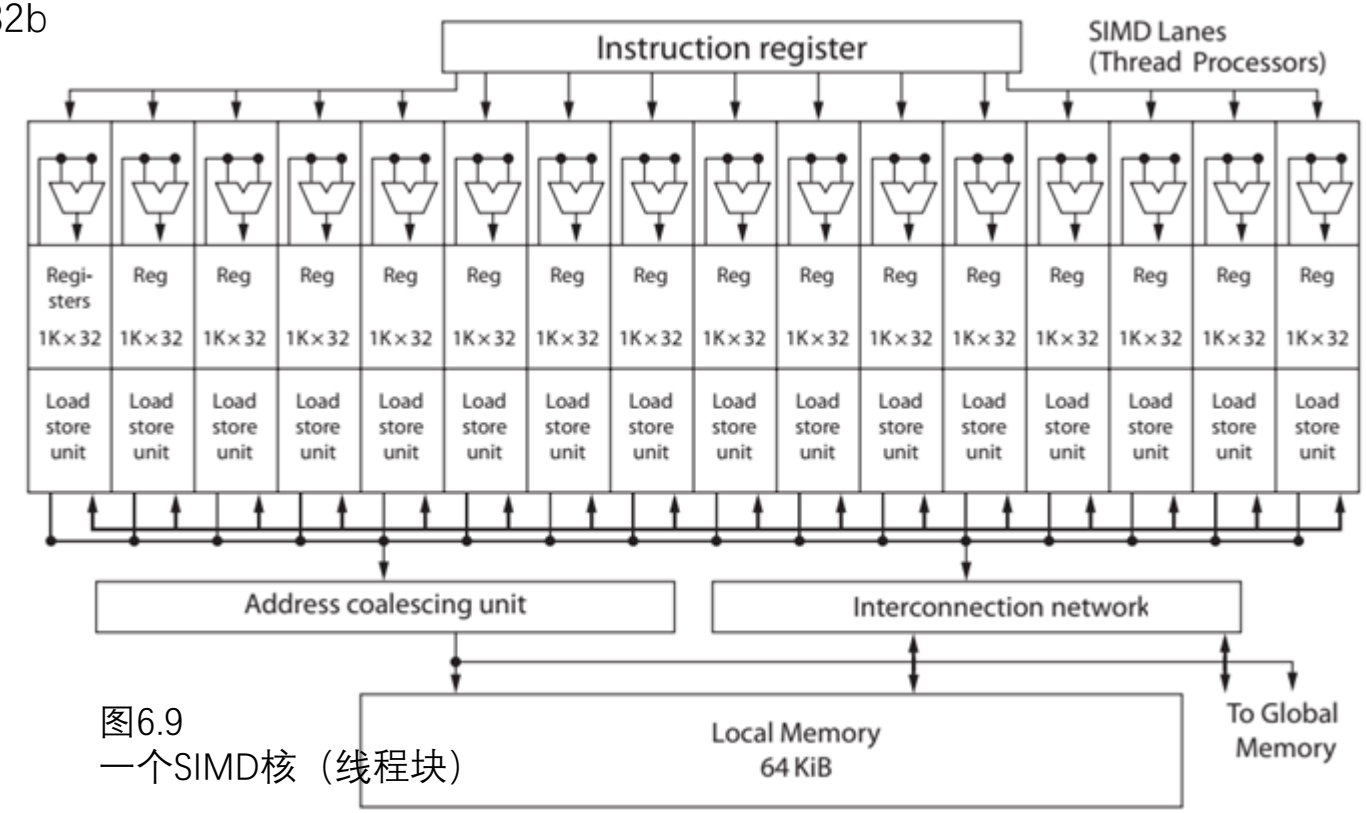
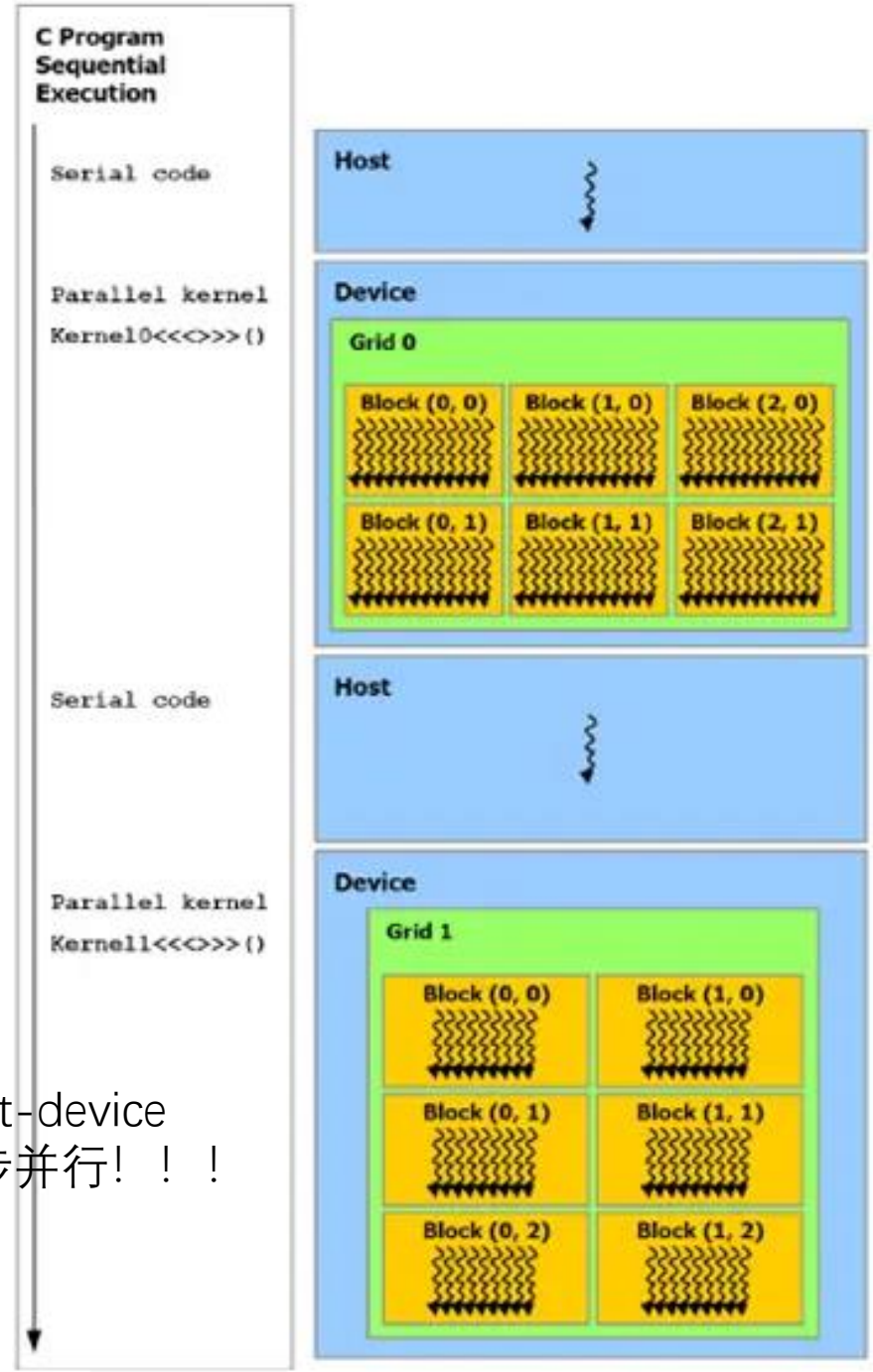


图6.9
一个SIMD核（线程块）

- CUDA编程：多线程并行计算
  - 网格grid：多个线程块，可按一维、二维或三维组织
    - 线程块block：n个SIMD线程，每个线程唯一ID
      - 一个块中的线程可以同步，或通过shared memory通信
      - 执行时，线程块被划分为m个线束
  - 内存分配：host内存，device内存
    - 统一内存：自动在CPU内存和GPU内存间进行数据传输
  - CUDA函数类型：device、global（核函数）、host
- 执行：一个kernel（核函数）启动一个grid
  - 线程束wrap：含32个线程，是SM的调度执行单元
    - 一个warp的线程对不同数据执行相同的指令，即 SIMT

CUDA异构编程：一个求和程序

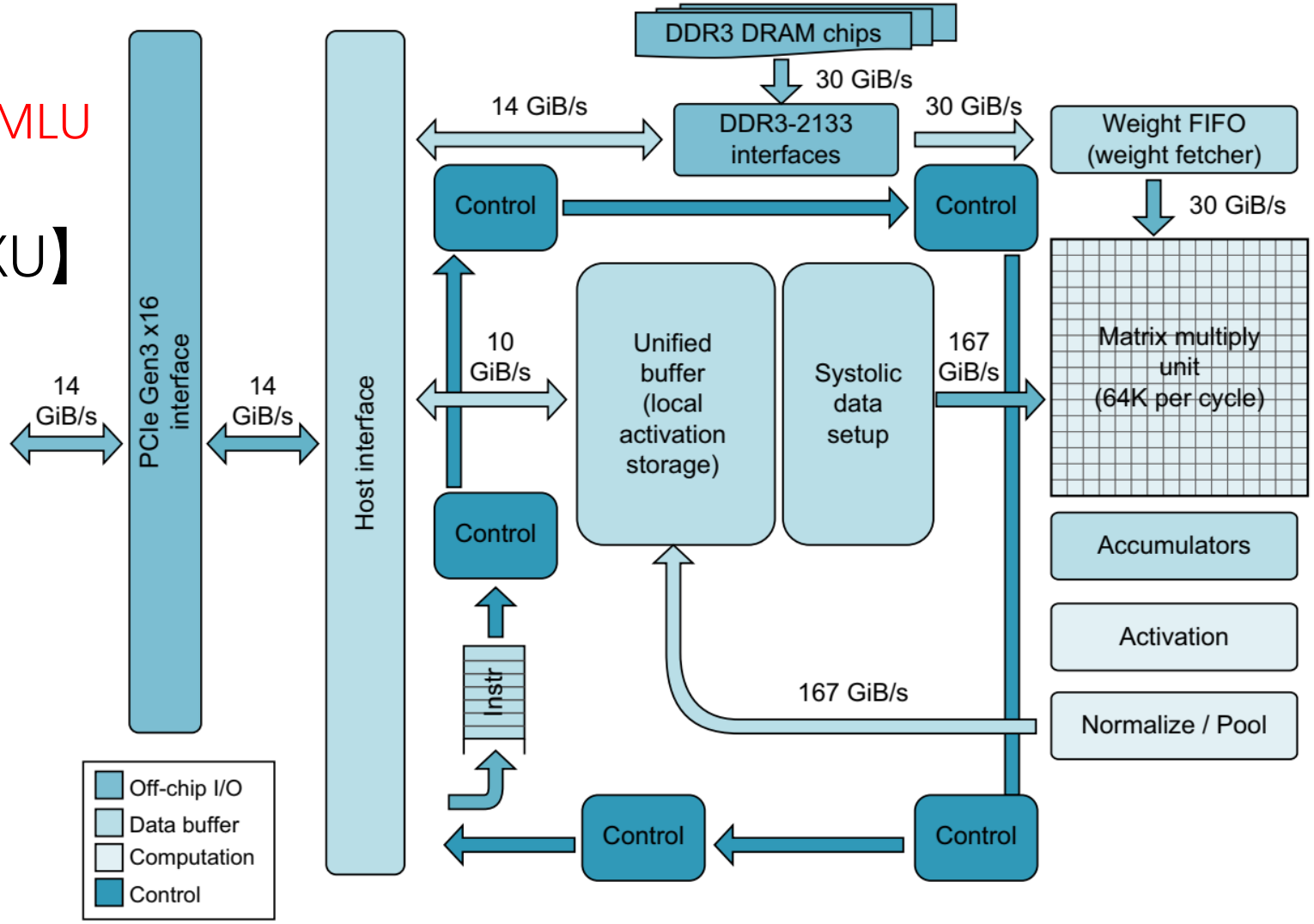
```
#define N 10
int main(void ){
    int a[N],b[N],c[N];
    int *dv_a,*dev_b,*dev_c;
     //创建GPU内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N*sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N*sizeof(int) ) );

    for(int i=0; i<N; i++){
            a[i]=-i;
            b[i]=i*i;
     }

    //从主机空间的数据复制到设备上
    HANDLE_ERROR( cudaMemcpy (dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy (dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy (dev_c, c, N*sizeof(int), cudaMemcpyHostToDevice ) );

    add<<N, 1>> (dev_a, dev_b, dev_c); //调用处理函数，并声明N个线程block，每个block有1个线程。

    HANDLE_ERROR( cudaMemcpy( c, dev_c, N*sizeof(int) cudaMemcpyDeviceHost ) );
    for(int i=0; i<N; i++)
        printf("%d+%d =%d\n",a[i],b[i],c[i]);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
_global_ void add( int *a, int *b, int *c){
        int tid = blockIdx.x; //blockId是CUDA一个关键字
        if(tid<N)
                c[tid] = a[tid] + b[tid];
}
```



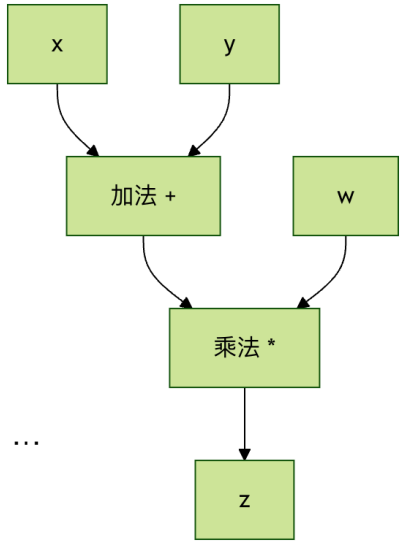
Host-device
异步并行！！！

Domain-Specific Architectures：CPU、GPU、TPU、NPU，$6.7

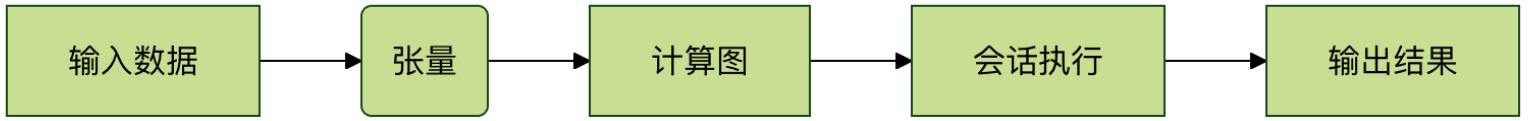
- DSA：面向能效
  - *FPU*、DSP、GPU、TPU、MLU
  - 实时反应式RPU?
- DSA五原则：如TPU【MXU】
  - 使用专用存储器
    - 最小化数据迁移
  - 将HW用于ALU和存储器
    - 放弃微结构过度优化
  - 采用专门的并行方式
    - 领域专用
  - 减少数据大小和类型
    - 领域专用
  - 使用领域专用语言
    - 如TensorFlow（DNN）



TensorFlow



- 高性能计算编程框架
  - 张量(Tensor)：0~n维数组
    - TensorFlow的基本数据单位
      - 标量0维，向量1，矩阵2，图像3，视频4，…
  - 计算图(Computational Graph)
    - “模型”：描述数据(Tensor)流动(Flow)的DAG，张量流
  - 会话(Session)
    - 执行计算图的运行时环境
      - TensorFlow 1.x：计算图的构建和执行是分离的【称“静态图”】
      - TensorFlow 2.x：即时执行【称“动态图”】
  - 可在 CPU、GPU、TPU 上运行
- 应用：分布式训练（TensorFlow的Keras）

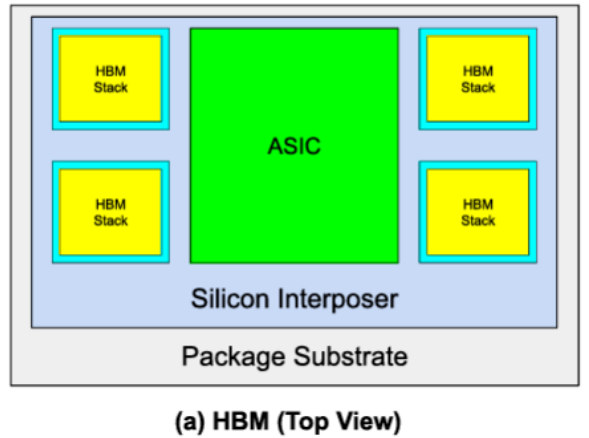



```python
# TensorFlow 1.x 风格（仅作理解，不推荐使用）
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

# 定义计算图
x = tf.placeholder(tf.float32, shape=[None, 784])
W = tf.Variable(tf.random.normal([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.matmul(x, W) + b

# 创建会话并执行
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    result = sess.run(y, feed_dict={x: input_data})
```

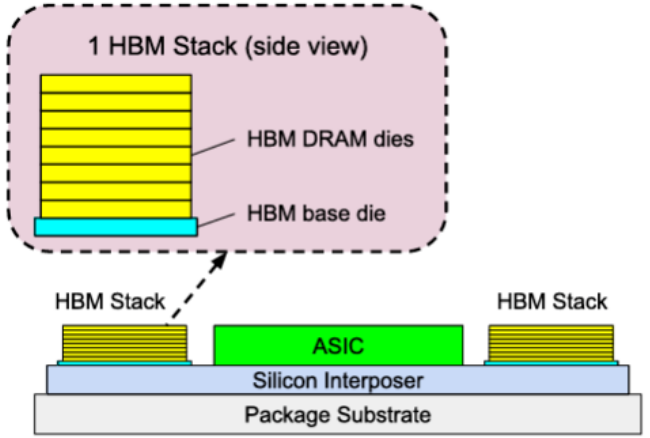
```python
# TensorFlow 2.x 风格（推荐）
import tensorflow as tf

# 直接执行运算
x = tf.constant([[1.0, 2.0, 3.0]])
W = tf.Variable(tf.random.normal([3, 2]))
b = tf.Variable(tf.zeros([2]))
y = tf.matmul(x, W) + b

print(y)  # 立即得到结果
```

David Patterson：“AI推理不需要更强的GPU”

- LLM 推理的两个阶段
  - Prefill（预填充）阶段：处理输入序列的所有 token
    - 计算密集型，高带宽内存HBM，GPU/TPU 可以较好地应对
  - Decode（解码）阶段：自回归，每步只输出一个 token
    - 属“内存带宽受限的”，低延迟
  - “频繁、小消息、大网络”
- 挑战：内存墙（容量），延迟墙（访存，通信）
  - 方向1：近内存计算PNM
    - 在 HBM 的 base die 上集成计算逻辑
  - 方向2：网络内处理PIN



(a) HBM (Top View)

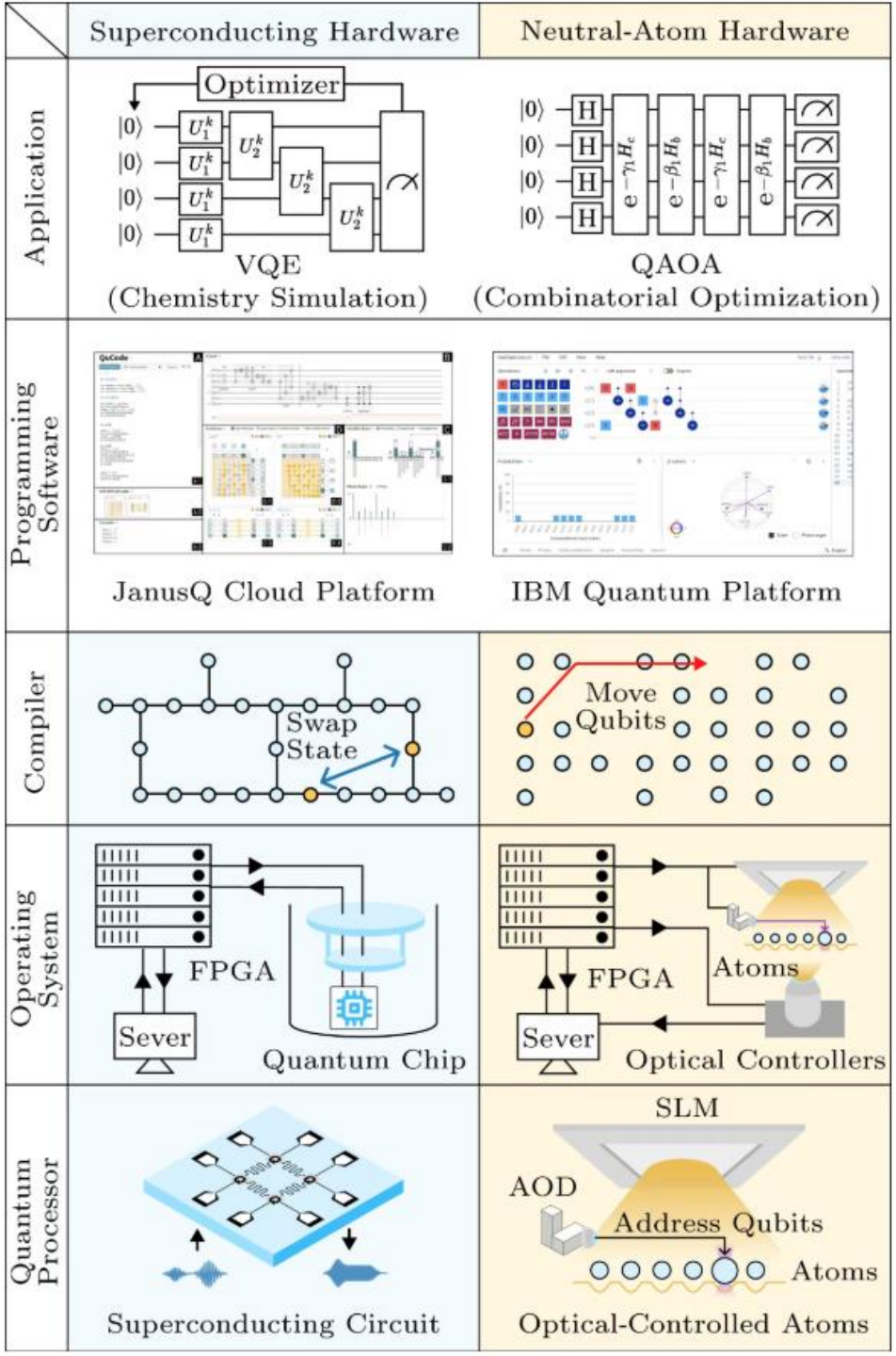


(b) HBM (Side View)

David Patterson, 马晓宇, “关于大语言模型推理硬件的挑战与研究方向”, IEEE Computer, 2026

量子计算系统架构

- 应用层（Applications）
  - 软件开发工具包（SDK）和编程语言接口
- 编程软件层（Programming Software）
  - 验证和程序修复方法，可编程性和可靠性
- 编译器层（Compiler）：
  - 逻辑到物理比特映射、幺正矩阵分解、门级优化及噪声缓解等
- 操作系统层（Operating Systems）：
  - 调度、标定和纠错等服务，运行稳定性和吞吐率。
- 量子处理器层（Quantum Processors）：
  - 两类工艺：超导电路，光控制



Yin JW, Chen ZR, Peng S *et al*. A review of quantum computing systems and software. *Journal of Computer Science and Technology*, 41(1): 147−169, Jan. 2026.

休息是为了走更远的路!