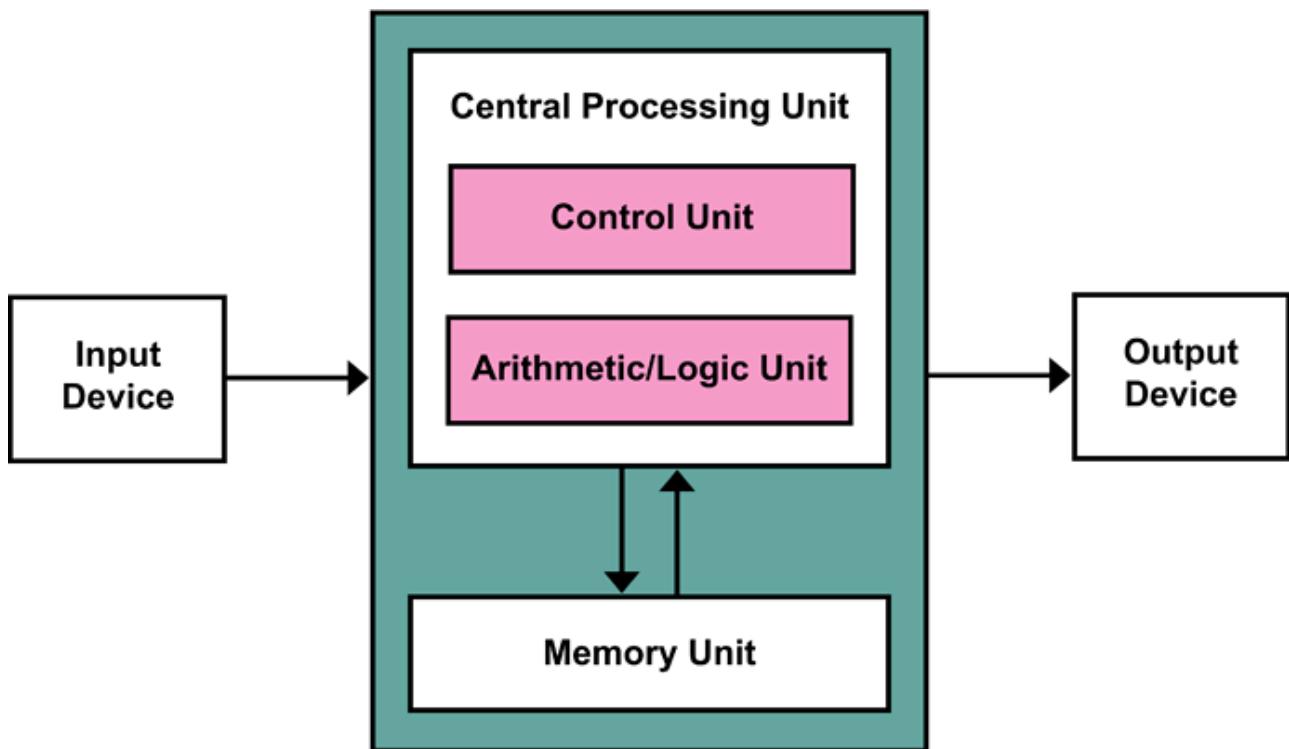


# 基础系统知识

## CPU结构

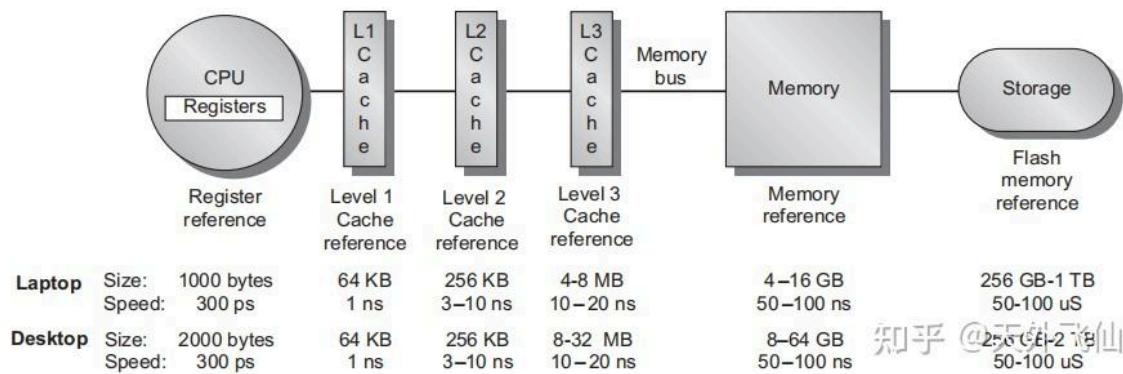
- 冯诺依曼结构（输入，输出，内存单元，计算单元，控制单元）之CPU结构



- 控制单元：CU
- 计算单元：ALU--可以去看~~YSYX~~的数字逻辑电路那一章
- 内存单元：

- CPU内存层级：

每一层的容量都要大于前一层次，但其访问速度也要更慢一点



- 寄存器register：通常由若干个D触发器构成（详细结构请移步《YSYXPart2.md》）
- 高速缓存(Cache)：通常由SRAM组成

### Cache高速缓存

- 引入cache的目的：

知乎 @无边无际

计算机在运行程序时首先将程序从磁盘读取到主存，然后CPU按规则从主存中取出指令、数据并执行指令，但是直接从主存（一般用DRAM制成）中读写是很慢的，所以我们引入了cache。

在执行程序前，首先会试图把要用到的指令、数据从主存移到cache中，然后在执行程序时直接访问cache。如果指令、数据在cache中，那么我们能很快地读取出来，这称为“命中（hit）”；如果指令、数据不在cache中，我们仍旧要从主存中拿指令、数据，这称为“不命中（miss）”。命中率对于cache而言是很重要的。

- 分类：

- L1Cache:单核私有，离CPU核最近，存储信息的读取速度接近CPU核的工作速度，容量较小

- i-cache(指令缓存):

- d-cache(数据缓存):

- L2Cache:单核私有,更远，速度更慢，容量更大

- L3Cache:多核共享，更远，速度最慢，更大

实际上cache是一个广义的概念，可以认为主存是磁盘的cache，而CPU内cache又是主存的cache，使用cache的目的就是伪造出一个容量有低层次存储器（如磁盘）那么大，而速度又有寄存器（如通用寄存器）那么快的存储器，简单来说就要让存储单元看起来又大又快。

- 理论基础：

- 时间局部性：如果一个数据当前被用到，那么接下来一段时间它很可能再次被用到

- 空间局部性：如果一个数据当前被用到，那么其周围的数据很可能被用到（数组）

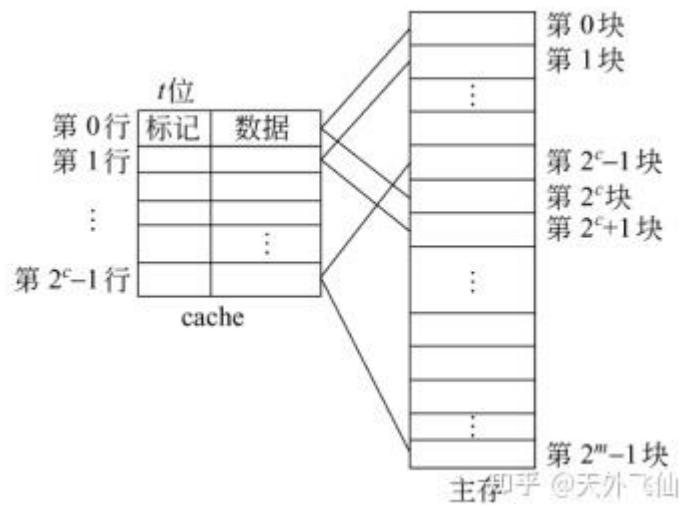
- 组成方式：

cache容量较小，所以数据需要按照一定的规则从主存映射到cache。一般把主存和cache分割成一定大小的块，这个块在主存中称为**data block**，在cache中称为**cache line**。举个例子，块大小为1024个字节，那么data block和cache line都是1024个字节。当把主存和cache分割好之后，我们就可以把data block放到cache line中，而这个“放”的规则一般有三种，分别是“直接映射”、“组相联”和“全相联”。

- 直接映射：

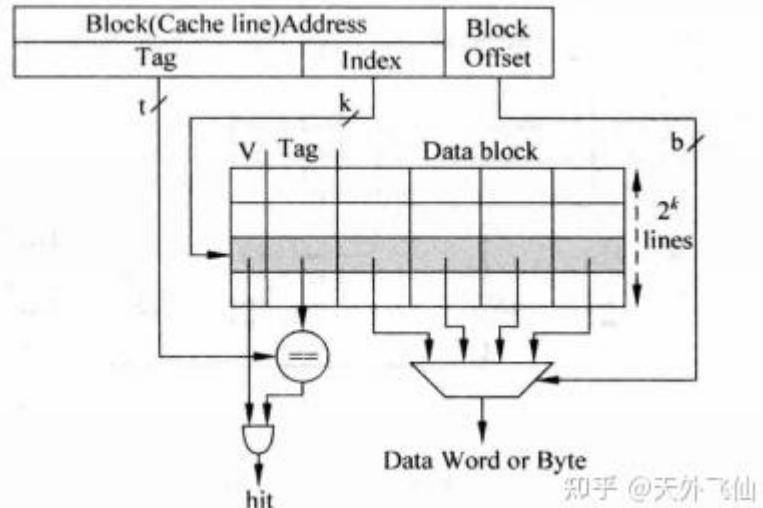
直接映射采用“取模”的方式进行一对映射。举个例子，如果cache中共有8个cache line，那么0、8、16、24...号data block会被映射到0号cache line中，同理1、9、17....号data block会被映射到1号cache line中，具体可以参考下面的关系图。

注意到上图中的cache除了数据之外还有“标记”位，“标记”可以显示出当前的cache line对应的是主存中的哪一组data block。举个例子，0、8、16.....号data block都可能存入0号cache line，此时标记位可以显示0号cache line到底是哪个data block。



- 实现电路：多路选择器，比较器（详细电路元件可以移步YSYXPart2）

直接映射中cache line一般有三个组成部分，分别是有效位V，标志位Tag，和数据位Data block。实现的电路结构如下。



hit为上面所说的命中位（即命中率）

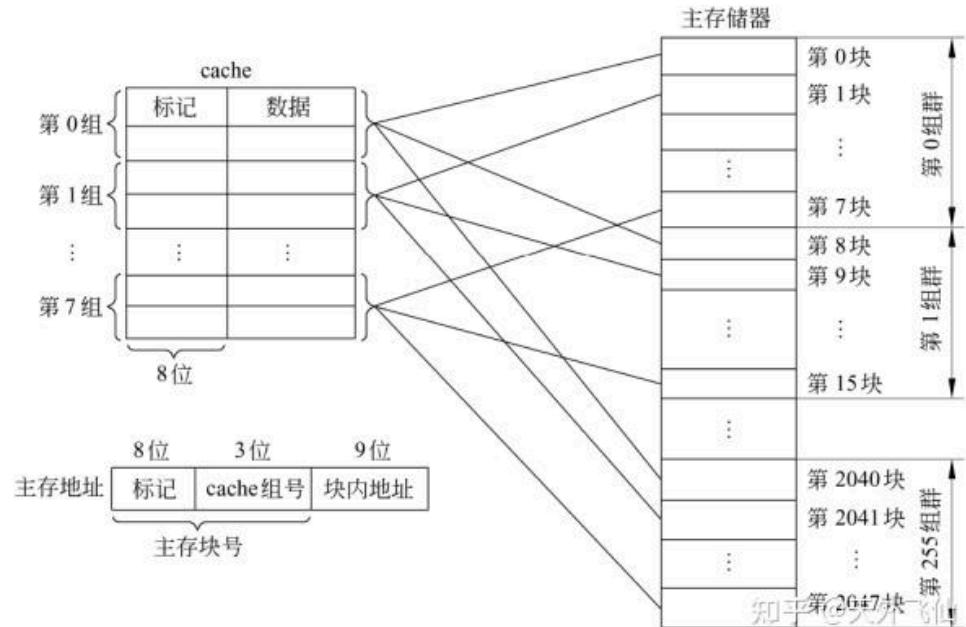
**CPU送来的地址按高低位被分成三部分，tag、index和offset。** index用来指定选中哪一个cache line，tag用来与cache line的tag作比较以生成hit信号，而offset则从选择的cache line中选中部分数据进行输出。

**要注意的是**，index会首先经过一个译码器，译码器生成一段独热码，独热码只会选中SRAM中的某一行，所以在读取data的时候只有对应的cache line会被读出，其他cache不会被读出。

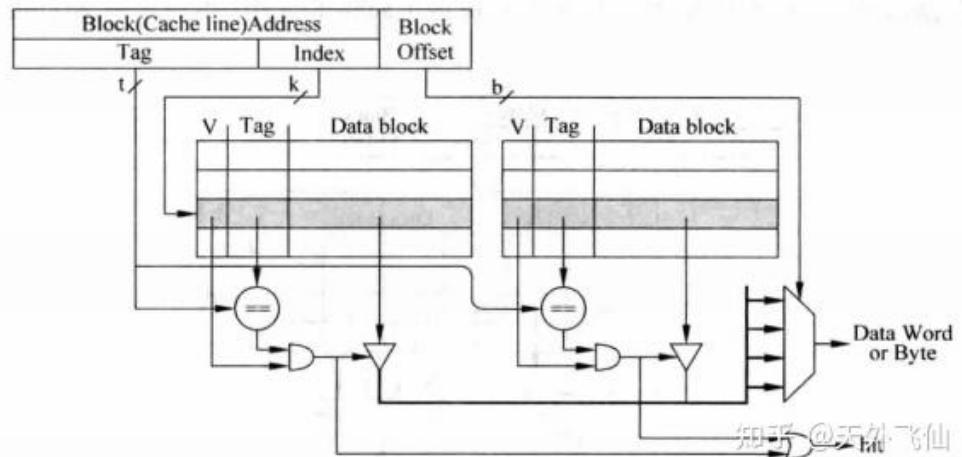
- 组相联：翻倍了每组cache line的数量

直接映射中主存中的每一个data block都有一个确定的cache line进行映射，这是有缺陷的。

当程序连续读取0、8、0、8号data block的数据时，因为只有一个cache line供映射，所以当第二次读取0号block时，第一次读到cache中的0号block早被顶替出去了，这时候又会产生miss，miss会极大地影响执行效率。



下图cache set = 2×cache line

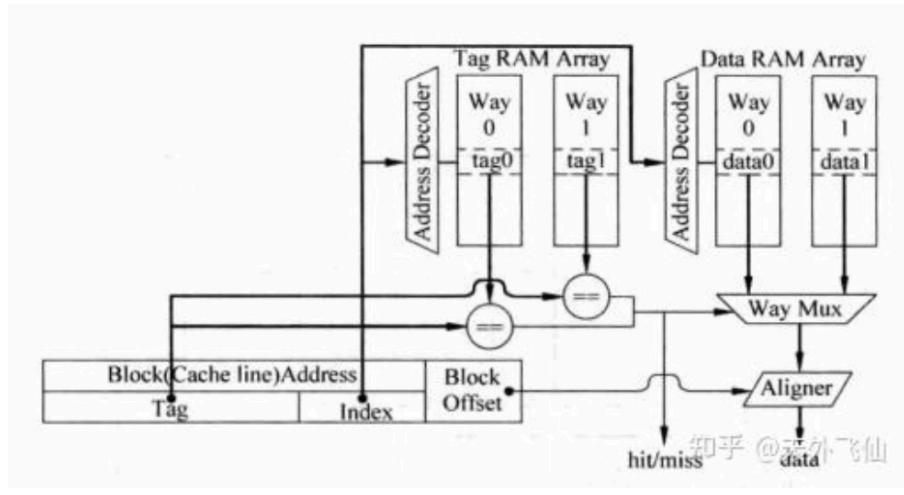


实现电路和直接映射很相似，不同的地方在于直接映射中index只选出一个cache line，而这里选出了两个cache line。两组data根据tag的比较结果来输入到选择器，实现方式是令两组data直接通过三态门连到一组数据线上。不熟悉这个操作的朋友可以先查阅2.5节内容。

在真实场景中组相联cache的tag和data往往被分开存储，因为分开存储，组相联实现电路分化成了并行和串行实现方式。

- 并行方式：

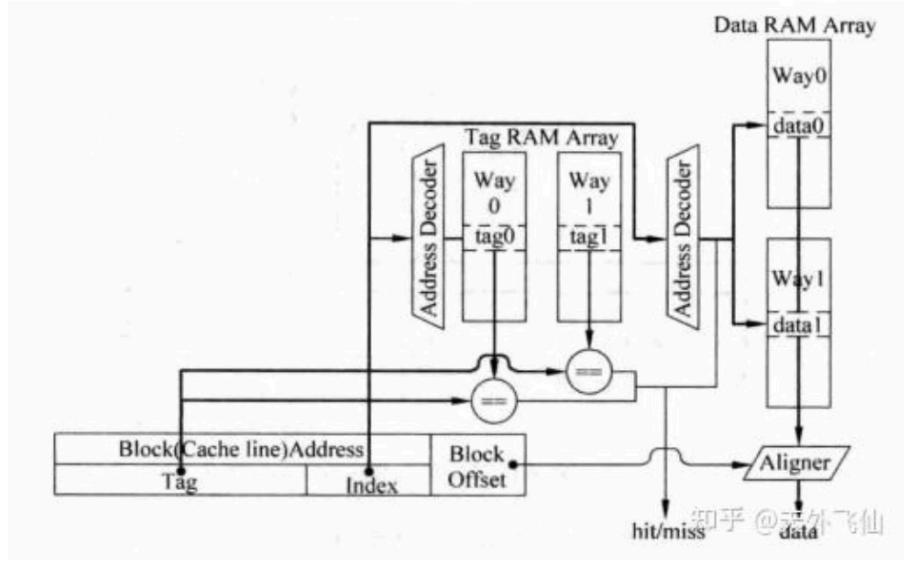
下图是并行实现方式。index同时送到tag ram和data ram，同时译码，同时读取tag和data，并根据tag比较的结果来选择一组data进行输出，aligner是字节选择器。这里关键的地方在于我们看起来就像把cache set中的两路cache line横向拼接起来，然后根据index的译码结果选中某一行，这一行包含两个cache line中data。



并行实现组相联

■ 串行方式：

下图是串行实现方式。相比于并行，这里的关键地方是我们把两路cache line纵向拼接了，这样cache line的数量翻倍，通过tag比较和index译码的综合结果，我们最终只会选中一个cache line，选中的cache line中的数据直接送往aligner。这样的工作过程有明显的串行特征，即首先tag比较，然后才选中某一cache line。



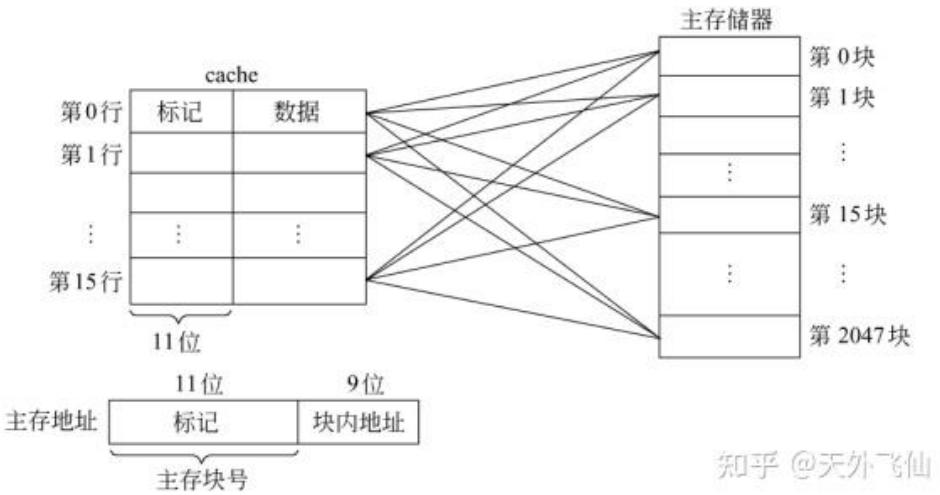
串行实现组相联

■ 比较：

因为并行是Tag和DataRAM同时进行，只需要一个时钟周期，而串行是先进行Tag再进行Data，所以会多一个时钟周期

比较串行和并行实现，并行实现因为比串行多一个路选择器，工作时间会变长，对应的时钟频率会下降，而且每次同时选中多个cache line，功耗较大；而串行实现在用流水线来实现cache时会明显增加所需时钟周期数（多一个时钟周期）。

- 全相联：极端的组相联，即cache只有一个cache set。每一个data block都可以存进任何一个cache line

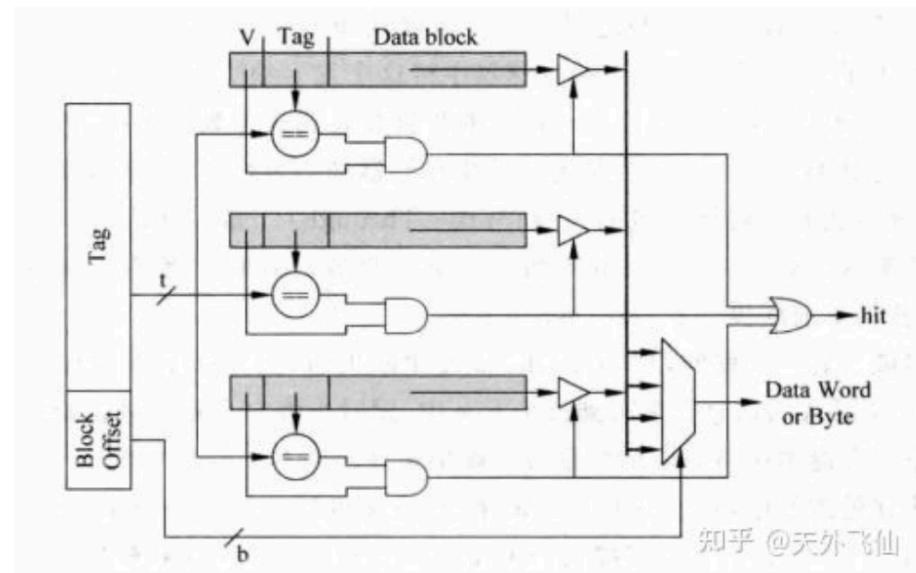


知乎 @天外飞仙

■ 实现电路：

不需要index是因为，index是指示是哪一个cache line的

容易想到，全相联不需要index了，下图是实现电路。我们直接对照每一个cache line的tag并由此控制data的三态门输出。这个实现方法是很简单的，但是这里因为需要做大量的比较电路，所以工作延时也是巨大的。**(why? 因为CPU给出的地址的tag部分需要支持所有的比较电路，负载很大，负载可以简单化成一堆电容，负载很大相当于电容很大，电容很大，充电时间就长，相应的工作时间就长)**



知乎 @天外飞仙

全相联电路实现

- SRAM：读写速度快
- 电路还没看，感觉在晶体管层有点抽象
- 三态门：
- 目前阶段可以理解成是多路选择器（实际上有区别）
- Cache的写入：

注意这里提个醒，注意**cache line**这个概念，这个概念是指**cache**的一行（现代CPU一般为64字节），这一行可以有多个数据，因为下面循环分块和伪共享都用到这个知识点了

1. 将被改写的数据在cache中（cache hit）

1. 写回：只改写cache中对应的cache line

速度快（不用访问主存），但是产生一致性问题（cache与主存的数据不再一致）

需要dirty：

cache line和主存中的数据不再一致，这会产生“一致性”问题，如果有别的核来访问主存中对应的block，那么它将会读到错误的数据。另外，在cache line被替换出去的时候，数据应该被写进主存，这就要求我们能够辨别哪些cache line是被改写过的，反映在电路上就需要增加一个“脏”位，当一个被标记为脏的cache line被替换出去，其内容需要被写入对应的主存。

2. 写穿：改写cache line和主存

2. 将被改写的数据不在cache中（cache未命中）

对比第一种就是时刻可以保证存储器的数据一致，但是速度慢有延时代价

1. 写不分配：直接把数据写入主存

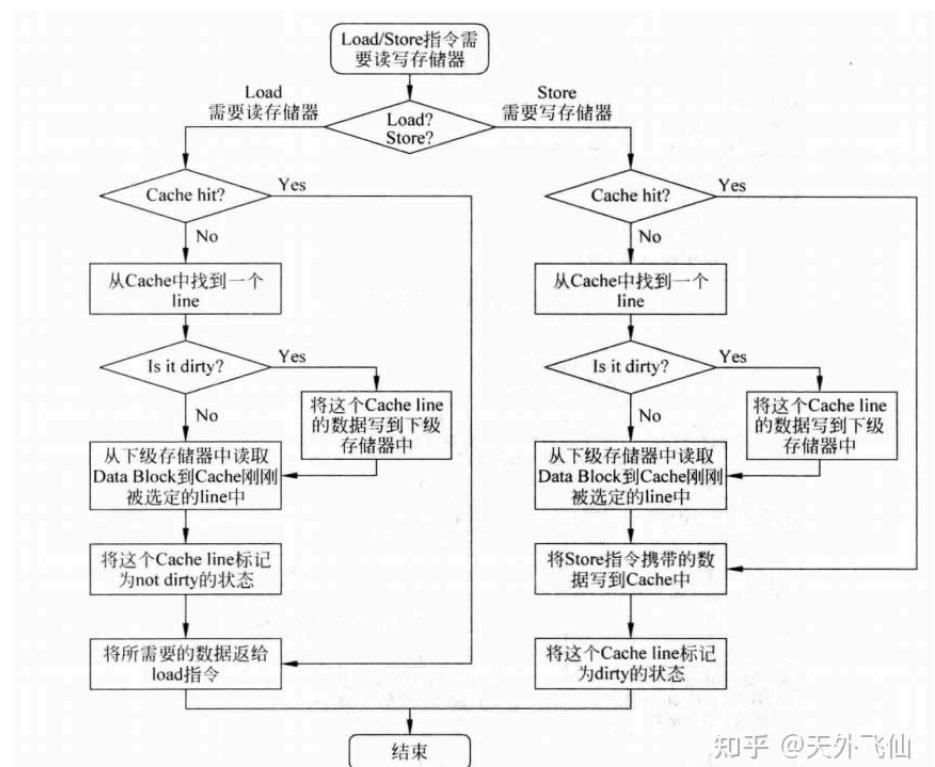
2. 写分配：先把data block放进cache line，然后写回

■ 写入分配策略：

load：读。把内存/缓存中的数据读到寄存器/CPU

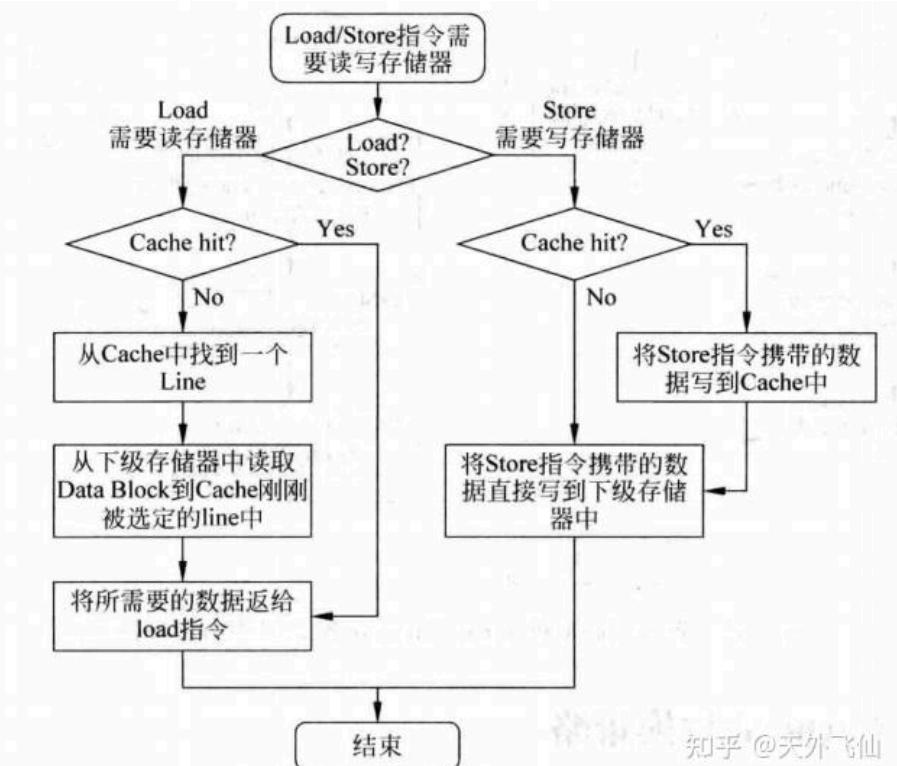
store：写。把寄存器/CPU的值写回内存/缓存

1. 写回+写分配



2. 写穿+写不分配

知乎 @天外飞仙



知乎 @天外飞仙

- Cache的替换策略：无论是读数还是写数，一旦cache未命中就需要做替换

- 方法：

1. LRU (近期最少使用)：选择最近一段时间使用次数最少的cache line进行替换

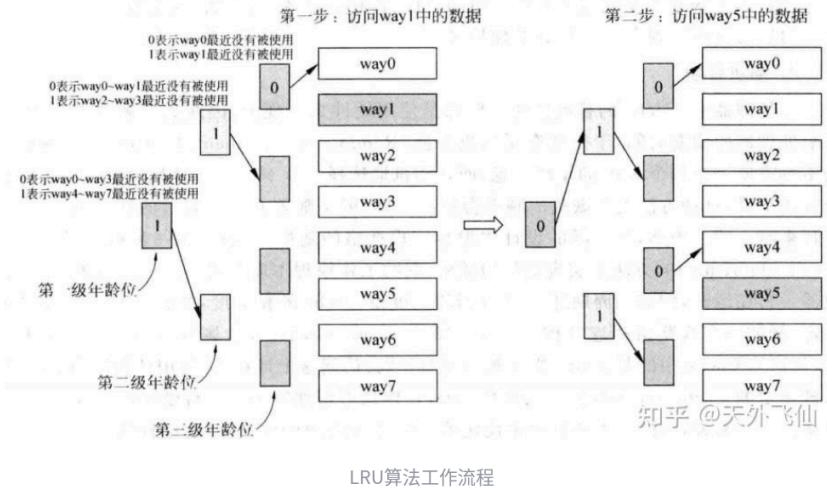
- 具体实现方法：为每一个cache line设置一个“年龄位”

两种年龄位的设置，一种像链表，一种像树

“年龄位”的优化设置可以类比树的设置

如果是2路cache（即每一个cache set只有两个cache line），那么只需要一位“年龄位”。当一个cache line被使用，那么它的年龄为1，另一个line年龄为0。如果是多路cache，那么就需要多位“年龄位”，当一个cache line被使用，那么它对应的年龄就应该被设置为最大，其他cache line的年龄按照之前的顺序排在它之后，这个过程就好像是把单向链表中的某一个节点拿出来放到链表的头，其余节点按照之前的顺序连接在头节点之后。替换的时候总是替换年龄最小的那个cache line，在链表中也就是把表尾去掉，然后把新的块放到表头。

年龄位是通俗易懂的，但是当cache set越来越大，如八路，那么年龄位的实现会变得很复杂，这时候我们有一个简单的方式来实现，首先来大略看看下面一幅图，重点看文字部分。



这个实现方法把八路cache进行分组，第一年齡位把cache set分成两组，一组四个cache line；第二年齡位把四个cache line又分成两组，以此类推。

图上的七个年齡位显示了访问cache line结束后的年齡情况，随着访存的变多，年齡位会慢慢地被填满，然后图中的箭头就会从第一级一路指向某一个way，这个cache line就是最近最少使用的cache line。图中的箭头还没有连完，因为图中只访存了way1和way5。

## 2. 随机替换

### ■ 具体实现方法：只需要一个内置的时钟计数器

在处理器中替换算法都是用硬件直接实现的，硬件复杂度可能会很高，有些情况下我们需要简单地实现替换功能，这时候随机替换就派上用场了。

随机替换不需要记录年齡，它只需要一个内置的时钟计数器，每当要替换cache line，就根据计数器的当前计数结果来替换cache line。

这样的方法优点是实现起来很简单，缺点是它不能体现出数据的使用的规律，因为它可能把最近最新使用的数据给替换出去，不过随着cache的容量越来越大，这个缺点所带来的性能损失也越来越小。总的来说，这是一个折中的办法。

- 主存 (RAM/Main Memory)：一般用DRAM，通常由又称内存或主内存，容量大 (GB级)，速度比缓存慢，用于临时存储当前运行的程序和数据。
- 外存：指硬盘 (HDD)，固态硬盘 (SSD)，容量最大 (TB级)，速度最慢，用于长期存储操作系统，应用和文件等数据。

# CPU--进程和线程/并行和并发

- CPU进程和线程

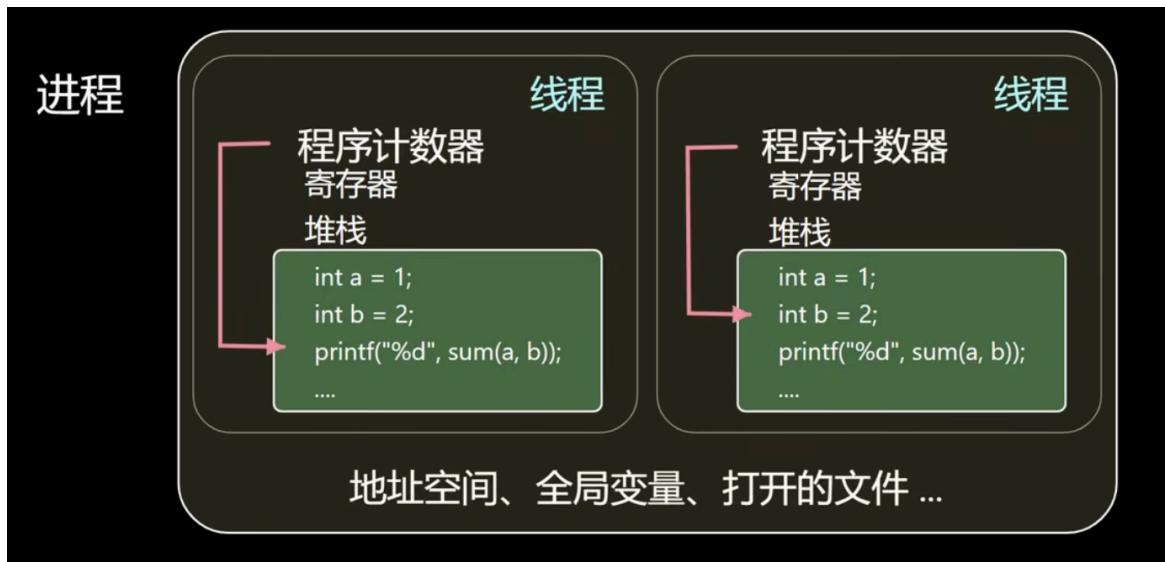
[进程和线程--b站](#)

[多进程vs多线程--b站](#)

- 进程：程序的一次执行过程。是程序在执行过程中分配和管理资源的基本单位

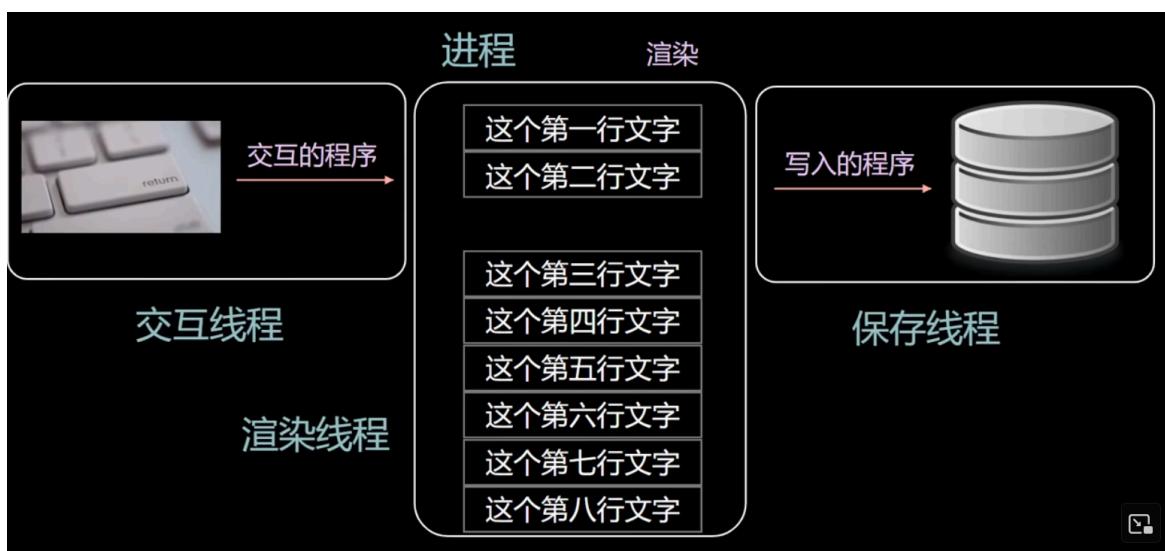
在内存中每个进程都会有一个虚拟地址，每个进程都以为自己独自占用着整个内存空间

- 进程的三种基本状态:
  1. 就绪状态
  2. 执行状态
  3. 阻塞状态
- 线程: 是CPU调度和分派的基本单位, 它可与同属一个进程的其他的线程共享线程所拥有的全部资源
  - 与进程的关系: 被包含在进程之中。一个进程中可以有多个线程, 每条线程并行执行程序中不同的任务



- 为什么要有线程?
 

方便并行执行程序中的事件



- 多进程和 (单核/多核) 多线程的区别是什么, 各有什么优缺点
  - 多进程: 同时运行多个进程 (相当于运行多个程序的实例)
    1. 优点:
      1. 各进程之间相互独立互不干扰 (如果一个崩溃了不会影响其他的进程)
      2. 真正实现并行: 在多核心处理器中可以对每个进程分配一个CPU核心用来处理
      3. 对py更加友好
    2. 缺点:

1. 进程间通信 (IPC) 更加复杂
2. 内存开销高--每个进程都需要有自己的内存资源
3. 上下文切换更加消耗资源 (进程切换相比线程切换来说)

- 多线程: 在一个进程中能“同时”执行多个线程

- 1. 优点:

1. 高效共享资源--适合需要频繁交互的任务场景
2. 内存占用少: 因为一个进程分配一个内存, 多个线程共享这一份内存
3. 切换速度快, 线程切换比进程切换快

- 2. 缺点:

1. 管理共享资源访问很棘手
2. 在py等解释型语言中, 全局解释器会阻止这一过程导致无法真正实现并行(GIL:全局解释器锁, 下文有解释)
3. 死锁: 陷入线程无限循环的死循环中导致无法正常进行

- 单核多线程和多核多线程

- 单核多线程: 单核CPU轮流执行多个线程, 通过给每个线程分配cpu时间片来实现
    - 多核多线程: 多个线程分配给多个核心处理, 相当于多个线程并行执行
- 相当于下面的并行和并发的思想

- 应用:

- 1. 多进程: CPU密集型 (计算多) --真正实现并行

- 2. 多线程: I/O密集型 (必须需要长时间等待外部资源, 线程能更加高效的处理这类等待)

- CPU并行和并发--在多任务的条件下

- 并行: 在同一时间真正执行多个任务 (多核/多处理器)

- 计算机三种并行方式:

- 流水线--CPU, 指令级并行技术:

[流水线--b站](#)

[一生一芯--流水线](#)

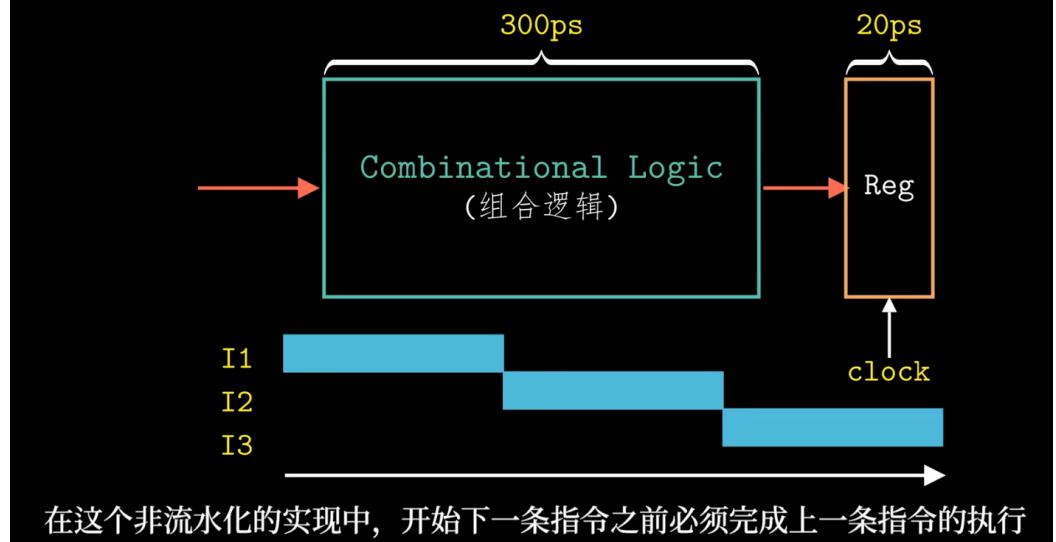
- 非流水线:

延迟: 相当于一条指令执行完的时间

非流水化中, 一条指令执行完才能执行下一条指令, 即, 指令的执行不存在相互重叠的情况

吞吐量: 一秒中能实现多少条指令, 吞吐量越大, 电路效率越好

# General Principles of Pipelining

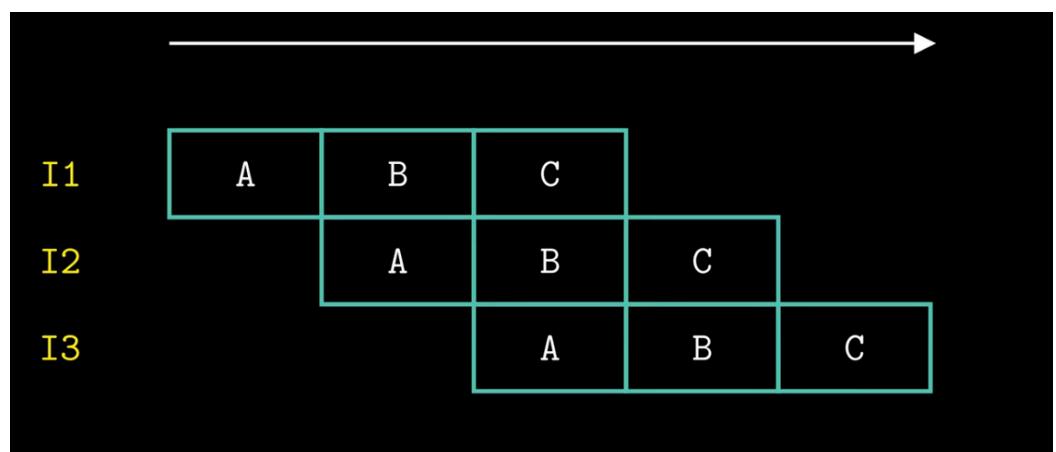


在这个非流水化的实现中，开始下一条指令之前必须完成上一条指令的执行

- 流水线:

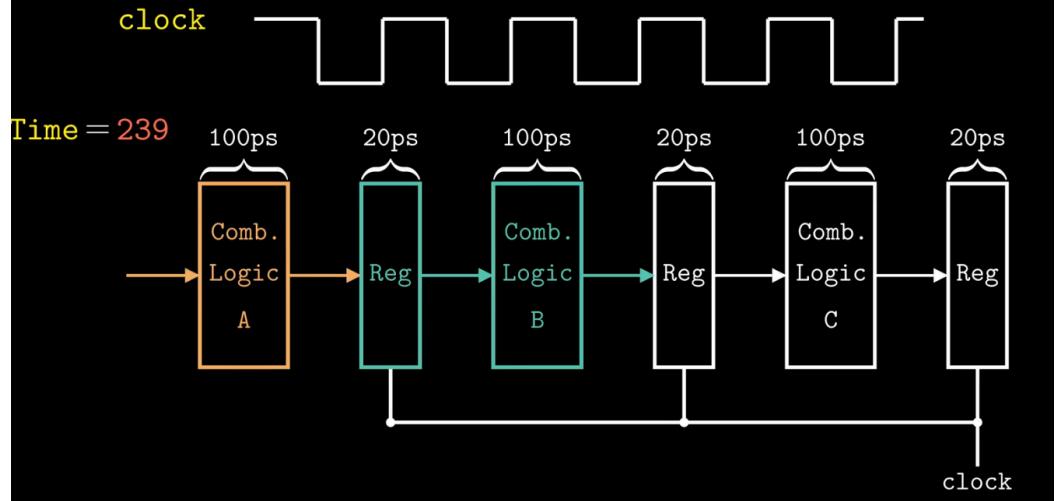
指令不非要执行完才能进入，只需要前一条指令从A阶段到B阶段，那么当前的指令就可以进入A阶段

但是这种必须要求指令之间不能相互依赖（即相互独立）



例子--三级流水线（理想情况下）：

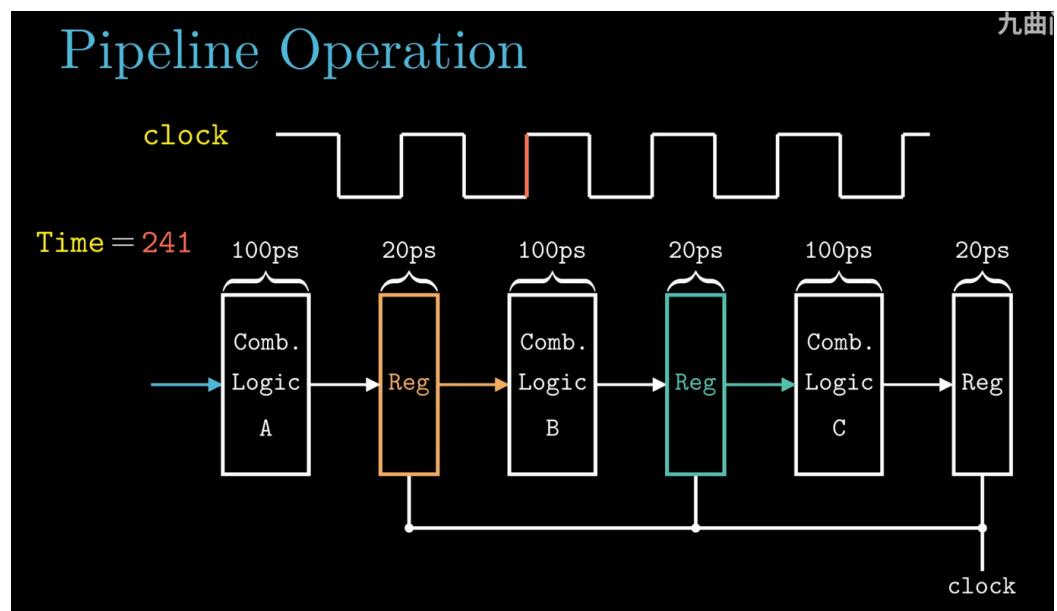
# Pipeline Operation



在300ps的组合逻辑电路中加入三个寄存器，所以这样在第239ps的时候，第一条指令马上进入第二个register，此时第一个寄存器还保存着第一条指令的信息，（如上图，**第一条指令占用这两个部件（青色）**）那么此时第二条指令就可以进入第一个组合逻辑电路进行执行（橙色）

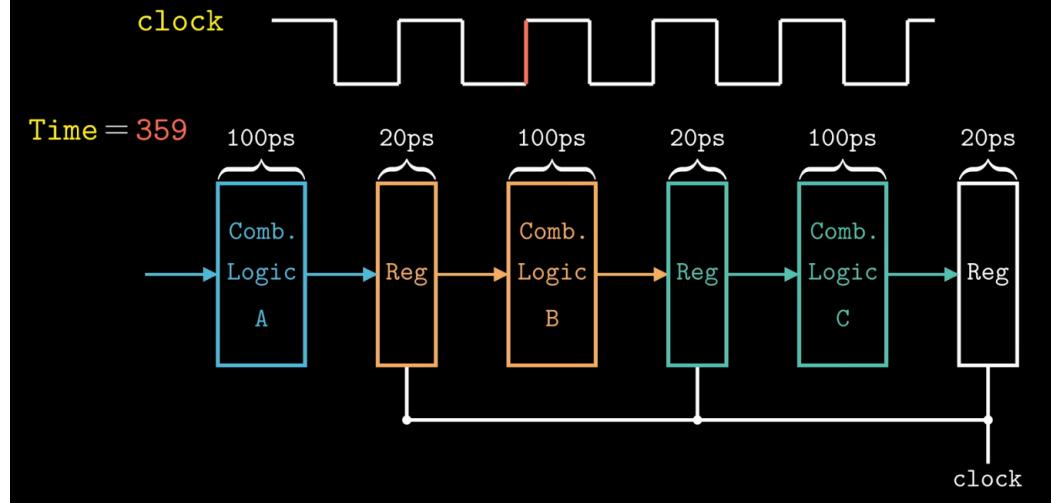
如上图，这样就充分利用了电路

第241ps的时候



第359ps的时候

# Pipeline Operation



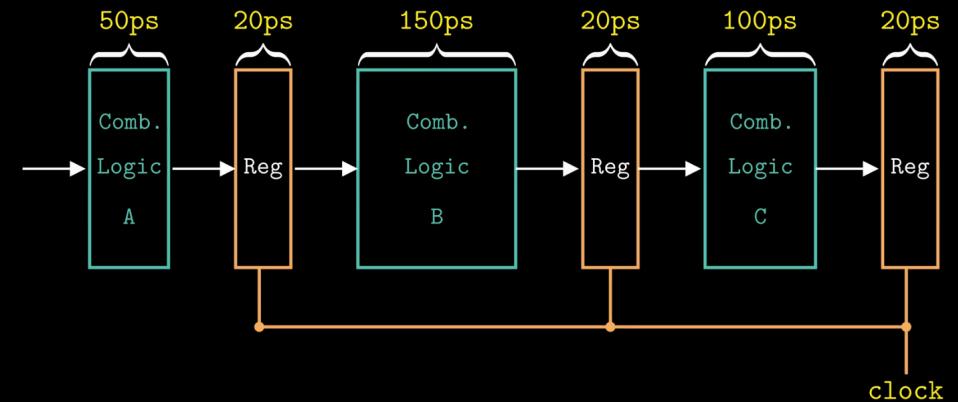
## ■ 现实情况下的流水线

### 1. 组合逻辑电路无法平均划分的问题

“如何分割组合逻辑电路使其达到最优的程度是一种设计流水线的挑战”

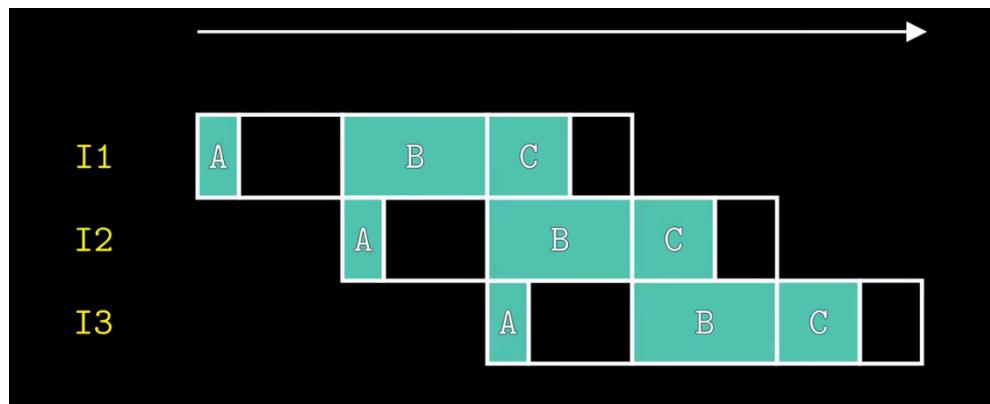
每个阶段的延迟可能不相等，这时候的状态转换就必须依赖于最长延迟的那组部件

## Nonuniform Partitioning



虽然三个阶段的延迟加起来仍旧是300ps

即：第二个指令的进入必须要等B阶段这么长的时间，即使A阶段有100ps的空闲



## 2. 指令之间相互依赖的问题

数据依赖/数据冒险：当前指令需要依赖于上面的指令

### Data Dependency

```
irmovq $50, %rax  
addq %rax, %rbx  
mrmovq 100(%rbx), %rdx
```

控制依赖/控制冒险：条件测试的结果决定要实现哪一条指令

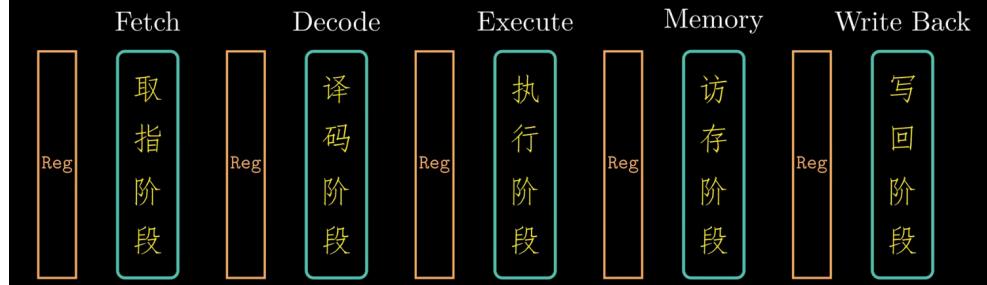
### Control Dependency

```
loop:  
    subq %rdx, %rbx  
    jne targ  
    irmovq $10 %rdx  
    jmp loop  
targ:  
    halt
```

五级流水线结构：

### Hardware structure of Pipeline

九曲阑干 



## 3. 结构冒险--硬件资源冲突 (单个端口无法同时取指和访存)

比如1指令在B阶段需要访问寄存器x

2指令在A阶段也需要访问寄存器x

在同一时刻同时发生，这样就会产生冒险

## 如何解决这一问题还没学

### ■ 超标量--指令级并行：

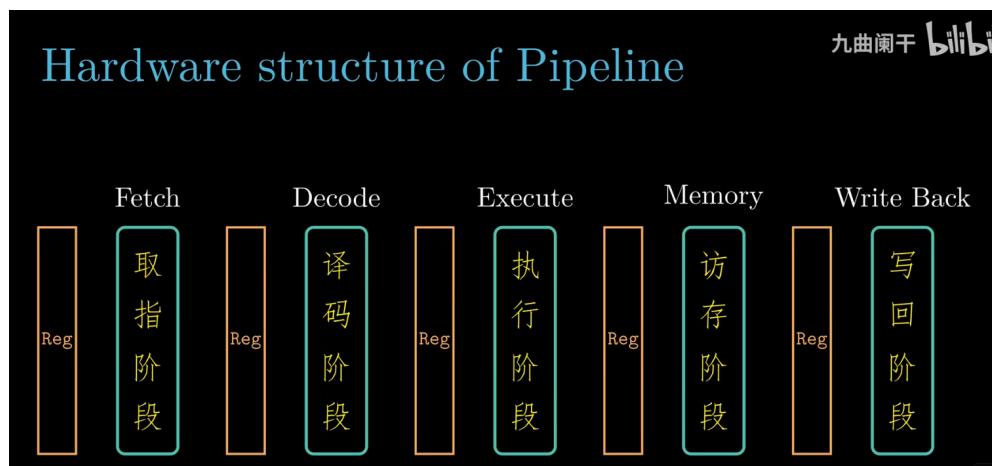
■ 概念：CPU的一个核心在一个时钟周期内获取，执行，提交多条指令

■ 实现：

1. 硬件基础：CPU中存在多个功能单元（比如多个ALU，多个浮点FPU，以及专用的加载/存储单元）
2. 是流水线处理器的扩展：每一个流水线阶段可以并行实现多条指令

比如在这一个阶段，流水线可能在执行阶段只允许一条指令存在，而经过超标量之后，可以在执行阶段，可以执行多条指令（流水线是让不同阶段容纳指令，超标量是让同一阶段容纳多条指令）

也可以理解成多条流水线



### ■ 多核--线程级/进程级并行：

参考上面介绍的多线程和多进程，

每个核心可以有自己的流水线/超标量逻辑，核心分别执行不同的任务

- 并发：在时间上交替执行多个任务（单核常用，时间片切换，即不断切换，使其看起来像是同时执行多个任务）

## 对CPU来说如何计算一个 $a[0:31] = b[0:31] + d[0:31]$ 的过程？

### 1. 标量处理：

1. 取指--译码
2. 加载数据：首先是内存管理单元计算实际物理地址，然后CPU向其发出请求进行加载数据，如果cache hit就直接加载到register，如果miss就访问主存（现代CPU都采用写回+写分配的cache写入策略）
3. 执行计算：ALU执行运算
4. 写回结果--进行下一次循环

### 2. 向量化并行：

1. 取指--译码
2. 向量加载: 和上述的流程差不多, 只不过这里是加载到向量寄存器中
3. 执行计算: ALU向量单元
4. 写回结果--进行下一次循环

## 使用C++多线程/多进程加速这一个计算过程

- 多线程: 使用< Thread >库

[C++多线程--菜鸟教程](#)

[C++实现多线程](#)

这里使用十亿个元素, 充分体现加速的优势

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
#include <immintrin.h> // AVX2 指令集头文件

// 单线程标量相加版本
void scalar_add(const std::vector<int>& b, const std::vector<int>& d, std::vector<int>& a) {
    for (size_t i = 0; i < b.size(); ++i) {
        a[i] = b[i] + d[i];
    }
}

// 多线程 AVX2 向量化相加版本
void multithread_avx2_add(const std::vector<int>& b, const std::vector<int>& d,
                           std::vector<int>& a, int num_threads) {
    size_t total_size = b.size();
    size_t chunk_size = total_size / num_threads;

    // 确保 chunk_size 是 8 的倍数, 以便 AVX2 正确处理
    chunk_size = (chunk_size + 7) / 8 * 8;

    std::vector<std::thread> threads;

    for (int i = 0; i < num_threads; ++i) {
        size_t start = i * chunk_size;
        size_t end = std::min((i + 1) * chunk_size, total_size);

        if (start >= total_size) break;

        threads.emplace_back([&, start, end]() {
            for (size_t j = start; j + 7 < end; j += 8) {
                __m256i b_vec = _mm256_loadu_si256(reinterpret_cast<const __m256i*>
(&b[j]));
                __m256i d_vec = _mm256_loadu_si256(reinterpret_cast<const __m256i*>
(&d[j]));
                __m256i result = _mm256_add_epi32(b_vec, d_vec);
            }
        });
    }
}
```

```

        _mm256_storeu_si256(reinterpret_cast<__m256i*>(&a[j]), result);
    }

    // 处理剩余元素
    for (size_t j = (end / 8) * 8; j < end; ++j) {
        a[j] = b[j] + d[j];
    }
});

for (auto& thread : threads) {
    thread.join();
}
}

int main() {
    // 设置较大的数组大小以便更好地测量性能
    const size_t SIZE = 10000000000;
    const int NUM_THREADS = std::thread::hardware_concurrency();

    std::cout << "数组大小: " << SIZE << " 个元素" << std::endl;
    std::cout << "可用线程数: " << NUM_THREADS << std::endl;

    // 初始化数据
    std::vector<int> b(SIZE);
    std::vector<int> d(SIZE);
    std::vector<int> a1(SIZE); // 用于标量版本
    std::vector<int> a3(SIZE); // 用于多线程 AVX2 版本

    for (size_t i = 0; i < SIZE; ++i) {
        b[i] = i % 100;
        d[i] = (i + 1) % 100;
    }

    // 测试单线程标量版本
    auto start = std::chrono::high_resolution_clock::now();
    scalar_add(b, d, a1);
    auto end = std::chrono::high_resolution_clock::now();
    auto scalar_time = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "单线程标量版本时间: " << scalar_time.count() << " 毫秒" << std::endl;

    // 测试多线程 AVX2 版本
    start = std::chrono::high_resolution_clock::now();
    multithread_avx2_add(b, d, a3, NUM_THREADS);
    end = std::chrono::high_resolution_clock::now();
    auto multithread_time = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "多线程 AVX2 版本时间: " << multithread_time.count() << " 毫秒" << std::endl;

    // 计算加速比 (标量时间 / 多线程时间)
}

```

```

    double speedup = static_cast<double>(scalar_time.count()) /
multithread_time.count();
    std::cout << "多线程加速比: " << speedup << "x" << std::endl;

    // 验证结果是否正确
    bool correct = true;
    for (size_t i = 0; i < SIZE; ++i) {
        if (a1[i] != a3[i]) {
            std::cout << "错误: 多线程 AVX2 版本结果不正确!" << std::endl;
            correct = false;
            break;
        }
    }

    if (correct) {
        std::cout << "结果验证: 正确" << std::endl;
    }

    return 0;
}

```

结果：

```

lyjy@Lenovo:~/LYJY_MLSys/Basic System Knowledge$ g++ -O3 -mavx2 -std=c++11 -pthread -o
MultiThreading MultiThreading.cpp
lyjy@Lenovo:~/LYJY_MLSys/Basic System Knowledge$ ./MultiThreading
数组大小: 10000000000 个元素
可用线程数: 32
单线程标量版本时间: 617 毫秒
多线程 AVX2 版本时间: 210 毫秒
多线程加速比: 2.9381x
结果验证: 正确

```

- 多进程：

通过fork创建子进程， mmap用来创建共享内存（用来实现进程间通信）

```

#include <iostream>
#include <vector>
#include <chrono>
#include <sys/mman.h>
#include <sys/wait.h>
#include <unistd.h>
#include <cstring>
#include <immintrin.h> // AVX2 指令集头文件

// 单线程标量相加版本
void scalar_add(int* b, int* d, int* a, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        a[i] = b[i] + d[i];
    }
}

```

```

// AVX2 向量化相加函数
void avx2_add(int* b, int* d, int* a, size_t start, size_t end) {
    size_t i = start;

    // 使用 AVX2 处理大部分数据
    for (; i + 7 < end; i += 8) {
        __m256i b_vec = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(&b[i]));
        __m256i d_vec = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(&d[i]));
        __m256i result = _mm256_add_epi32(b_vec, d_vec);
        _mm256_storeu_si256(reinterpret_cast<__m256i*>(&a[i]), result);
    }

    // 处理剩余的元素
    for (; i < end; ++i) {
        a[i] = b[i] + d[i];
    }
}

int main() {
    const size_t SIZE = 10000000000; // 1亿个元素
    const int NUM_PROCESSES = 8; // 进程数量

    std::cout << "数组大小: " << SIZE << " 个元素" << std::endl;
    std::cout << "使用进程数: " << NUM_PROCESSES << std::endl;

    // 使用mmap分配共享内存
    size_t data_size = SIZE * sizeof(int);
    int* shared_b = (int*)mmap(NULL, data_size, PROT_READ | PROT_WRITE,
                                MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    int* shared_d = (int*)mmap(NULL, data_size, PROT_READ | PROT_WRITE,
                                MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    int* shared_a = (int*)mmap(NULL, data_size, PROT_READ | PROT_WRITE,
                                MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    // 初始化数据
    for (size_t i = 0; i < SIZE; ++i) {
        shared_b[i] = i % 100;
        shared_d[i] = (i + 1) % 100;
    }

    // 预热
    scalar_add(shared_b, shared_d, shared_a, SIZE);

    // 测试单线程标量版本
    auto start = std::chrono::high_resolution_clock::now();
    scalar_add(shared_b, shared_d, shared_a, SIZE);
    auto end = std::chrono::high_resolution_clock::now();
    auto scalar_time = std::chrono::duration_cast<std::chrono::milliseconds>(end -
start);
    std::cout << "单线程标量版本时间: " << scalar_time.count() << " 毫秒" << std::endl;

    // 测试多进程 + AVX2 版本
    start = std::chrono::high_resolution_clock::now();

```

```

// 计算每个进程处理的数据块大小
size_t chunk_size = SIZE / NUM_PROCESSES;
// 确保块大小是8的倍数，以便AVX2正确处理
chunk_size = (chunk_size + 7) / 8 * 8;

// 创建子进程
for (int i = 0; i < NUM_PROCESSES; ++i) {
    pid_t pid = fork();

    if (pid == 0) { // 子进程
        size_t start_idx = i * chunk_size;
        size_t end_idx = (i == NUM_PROCESSES - 1) ? SIZE : (i + 1) * chunk_size;

        // 使用 AVX2 向量化执行计算
        avx2_add(shared_b, shared_d, shared_a, start_idx, end_idx);

        exit(0); // 子进程退出
    } else if (pid < 0) {
        std::cerr << "创建进程失败" << std::endl;
        return 1;
    }
}

// 等待所有子进程完成
for (int i = 0; i < NUM_PROCESSES; ++i) {
    wait(NULL);
}

end = std::chrono::high_resolution_clock::now();
auto multiprocess_avx2_time = std::chrono::duration_cast<std::chrono::milliseconds>
(end - start);
std::cout << "多进程+AVX2版本时间: " << multiprocess_avx2_time.count() << " 毫秒" <<
std::endl;

// 计算加速比
double speedup = static_cast<double>(scalar_time.count()) /
multiprocess_avx2_time.count();
std::cout << "多进程+AVX2加速比: " << speedup << "x" << std::endl;

// 验证结果
bool correct = true;
int* test_result = new int[SIZE];
scalar_add(shared_b, shared_d, test_result, SIZE);

for (size_t i = 0; i < SIZE; i += SIZE / 1000) { // 只检查部分样本
    if (test_result[i] != shared_a[i]) {
        std::cout << "错误: 结果不正确 at index " << i << std::endl;
        correct = false;
        break;
    }
}

```

```

if (correct) {
    std::cout << "结果验证: 正确" << std::endl;
}

delete[] test_result;

// 释放共享内存
munmap(shared_b, data_size);
munmap(shared_d, data_size);
munmap(shared_a, data_size);

return 0;
}

```

结果：

```

lyjy@Lenovo:~/LYJY_MLSys/Basic System Knowledge$ g++ -O3 -mavx2 -std=c++11 -o
MultiProcess MultiProcess.cpp
lyjy@Lenovo:~/LYJY_MLSys/Basic System Knowledge$ ./MultiProcess
数组大小: 1000000000 个元素
使用进程数: 8
单线程标量版本时间: 374 毫秒
多进程+AVX2版本时间: 239 毫秒
多进程+AVX2加速比: 1.56485x
结果验证: 正确

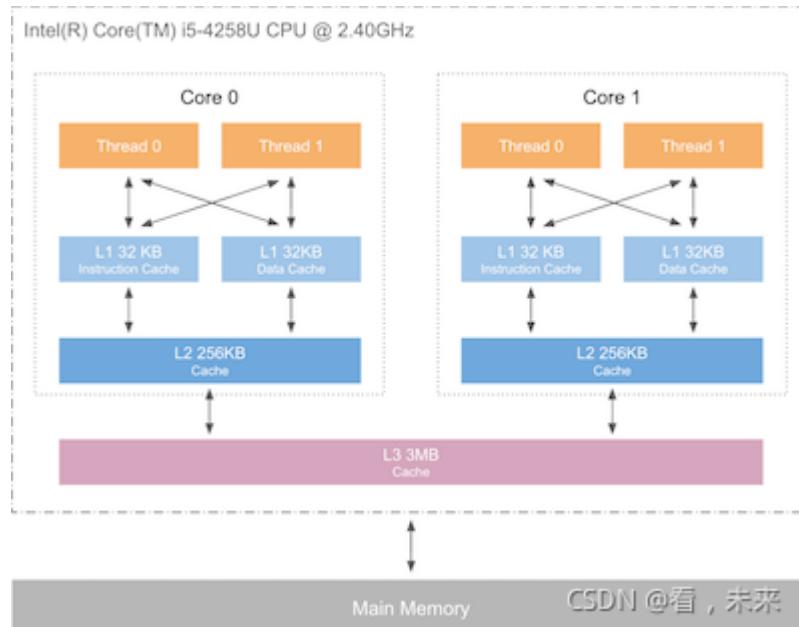
```

- 底层逻辑：

多线程和多进程的底层逻辑，都是利用**细分**的思想，将大任务划分为小任务，然后真正实现**并行计算**，同时利用**多个cpu核心和缓存层次结构**来实现的

- cache在多核层面的结构

和上面说的一样，1,2单核独享，3多核共享



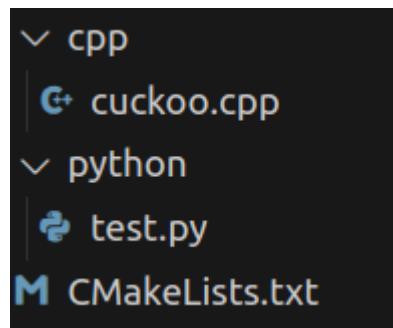
# Pybind/nanobind--cpp,py混合编程

CPP->py

[nanobind官方手册](#)

[nanobind第三方博客](#)

- 是什么：是用于创建python扩展模块的C++库，允许将C++代码暴露给python调用，从而兼顾C++性能和py的易用性
- 对比：pybind生态更加完善，更加适合初学者，nanobind适合性能有极致要求的项目
- 尝试
  - 项目目录



- 编写CMakeLists.txt和cpp文件

除了project名称和最后的模块需要修改项目名称，其他都是模板，以后直接套用

```
cmake_minimum_required(VERSION 3.18)
project(cuckoo)#这里是你的项目名称，这里的名称可以和下面写的名称不一样

#根据CMAKE的版本不同设置一个DEV_MODULE的值，这里的作用是基于CMAKE对于py包的查找在3.18这个版本做了重大改进
if (CMAKE_VERSION VERSION_LESS 3.18)
  set(DEV_MODULE Development)
else()
  set(DEV_MODULE Development.Module)
endif()

find_package(Python 3.8 COMPONENTS Interpreter ${DEV_MODULE} REQUIRED)

#添加以下几行代码。这些代码会将 CMake 配置为默认执行优化的发布构建，除非指定了其他构建类型。如果不添加这些代码，绑定代码可能会运行缓慢并生成较大的二进制文件。
if (NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Choose the type of build." FORCE)
  set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS "Debug" "Release"
"MinSizeRel" "RelWithDebInfo")
endif()

#如果您将 nanobind 安装为 Pip 或 Conda 包，请在 CMakeLists.txt 的末尾添加以下几行。它们会查询包以确定其安装路径，然后将其导入
execute_process(
  COMMAND "${Python_EXECUTABLE}" -m nanobind --cmake_dir
  OUTPUT_STRIP_TRAILING_WHITESPACE OUTPUT_VARIABLE nanobind_ROOT)
```

```
find_package(nanobind CONFIG REQUIRED)

#这里指定名字
nanobind_add_module(cuckoo cpp/cuckoo.cpp)
```

cpp文件

```
#include <nanobind/nanobind.h>

int add(int a, int b) { return a + b; }

// 注意这里的名字要和cmake中nanobind_add_module的名字相同
// 这里是一个宏，用来定义py模块
// m是一个对象，表示当前正在创建的py模块，大括号里面是py语法，用来def（定义一个py可以调用的函数，“”双引号里面的是py使用的函数名，&后面的是指向C++函数add的指针）
NB_MODULE(cuckoo, m) { m.def("add", &add); }
```

○ 编写完成后

1. 生成构建系统

-S指定源代码目录（即cmake所在的目录）

-B指定在当前目录下创建名为build的构建目录

```
cmake -S . -B build
```

2. 构建项目

```
cmake --build build
```

3. 使用

第一种

进入build去使用

```
cd build
python3

Python 3.11.1 (main, Dec 23 2022, 09:28:24) [Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
import my_ext
my_ext.add(1, 2)
3
```

第二种：创建软链接

在py的目录下执行

这种方式可以直接右键运行py，更加高效

```
ln -s ../build/cuckoo.cpython-39-x86_64-linux-gnu.so cuckoo.so
```

- nano模块中的编写比如编写类等还有待学习

## py中torch矩阵转化为C++数组

py->CPP

py中一个torch的矩阵如何转化为C++/C中的数组

1. 方法一：利用libtorch (Pytorch的C++接口，实现在C++中进行网络训练，网络推理的功能)

注意，使用libtorch的时候一定要在CMake中正确链接libtorch库 (链接第三方库)

[convert tensor into an array2 --pytorch](#)

```
#include <torch/torch.h>
#include <iostream>
#include <vector>

int main() {
    auto tensor = torch::rand({1, 2, 3, 5});
    tensor = tensor.view({tensor.size(0), -1});
    tensor = tensor.contiguous();
    std::vector<float> v(tensor.data_ptr<float>(), tensor.data_ptr<float>() + tensor.numel());
    std::cout << "v.size() = " << v.size() << std::endl;
    return 0;
}
```

```
cmake_minimum_required(VERSION 3.10)
project(libtorch_example)

set(CMAKE_CXX_STANDARD 17)
# 把下面路径替换成你解压 libtorch 的路径
set(CMAKE_PREFIX_PATH "/home/lyjy/libtorch_cuda/libtorch")

find_package(Torch REQUIRED)
add_executable(main main.cpp)
target_link_libraries(main "${TORCH_LIBRARIES}")
set_property(TARGET main PROPERTY CXX_STANDARD 17)
```

2. 如果是利用py文件里面的东西，就是要将py保存成文件，然后通过C++指针访问

## Python: GIL 是什么，为什么会有 GIL，如何缓解 GIL

GIL是什么？为什么会有GIL？如何缓解GIL？

[GIL--知乎](#)

- GIL：全局解释器锁，是py解释器中的一个机制，用于限制解释器在任意时刻只能在一个线程执行py字节码，从而防止多个线程同时执行py代码时发生数据竞争
- 为什么会有GIL：为了简化py解释器的设计，并保护对共享资源的访问不会因并发执行而导致数据不一致，尤其是增加了线程安全而不影响性能
- 如何缓解GIL：

## 1. 使用多进程

- **multiprocessing 模块**: Python的 `multiprocessing` 模块允许你创建进程池，每个进程都运行在自己的Python解释器实例中，并且拥有自己的GIL。因为每个进程都是独立的，所以它们可以并行运行在多核CPU上，不受GIL的限制。
- **适用场景**: 适合于计算密集型任务，特别是当任务可以被分解成多个独立子任务时。

## 2. 利用专门的库

- **NumPy和SciPy**: 这些科学计算库可以绕过GIL，因为它们的底层实现（通常用C或Fortran写成）执行重量级的计算任务时不需要Python的GIL。
- **TensorFlow、PyTorch等**: 这些深度学习和机器学习库在进行大规模矩阵计算或数据处理时，也能绕过GIL，因为它们同样使用了自己的并行计算技术。
- **适用场景**: 特别适合于数据分析、科学计算、机器学习和深度学习等任务。

## 3. 使用Jython<sup>+</sup>或IronPython

- **Jython和IronPython**: 这些是Python的替代实现，分别运行在Java虚拟机（JVM）和.NET环境中。它们没有GIL，因此可以更好地利用多核CPU。
- **适用场景**: 当你的项目可以从Java或.NET生态系统中受益，或当你需要与这些平台的组件交互时。

## 4. 异步编程

- **asyncio 库**: 虽然这不是直接绕过GIL的方法，但 `asyncio` 可以帮助你以单线程并发的方式执行I/O密集型任务，提高程序的整体效率和响应性。
- **适用场景**: 适合于I/O密集型任务，如网络应用、并发请求处理等。