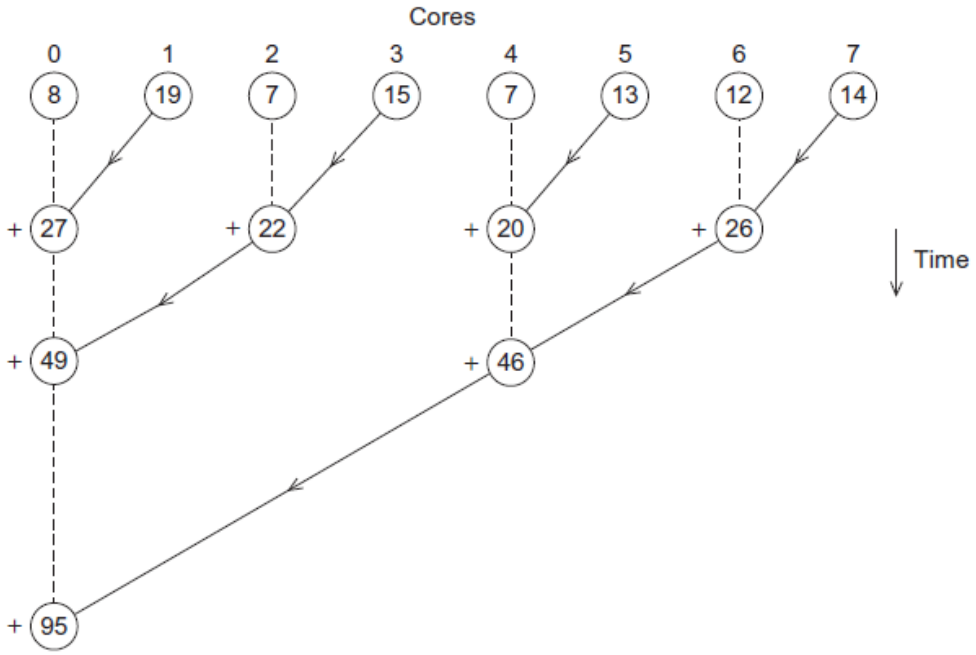
并行计算

（Parallel Computing）

思考题

- 写出树形结构求和的伪代码
  - 假设core的数目为2的幂次（如：1,2,4,8,…）



思考题

- 写出树形结构求和的伪代码
  - 提示：使用变量 divisor 来决定一个core是否应该发送其sum，还是接收并求和
  - divisor 应该从 2 开始，并且每次迭代翻倍
  - 变量 core_difference 决定当前 core 的 partner，从1开始，每次迭代翻倍
  - 如：第一次迭代，0 % divisor = 0 和 1 % divisor = 1，所以 0 接收并累加，1 发送，0 + core_difference = 1 和 1 - core_difference = 0，所以 0 和 1 是一对

思考题

```
divisor = 2;
core_difference = 1;
sum = my_value;
while ( divisor <= number of cores ) {
  if ( my_rank % divisor == 0 ) {
    partner = my_rank + core_difference;
    receive value from partner core;
    sum += received value;
  } else {
    partner = my_rank - core_difference;
    send my sum to partner core;
  }
  divisor *= 2;
  core_difference *=2;
}
```

并行硬件和并行软件（一）

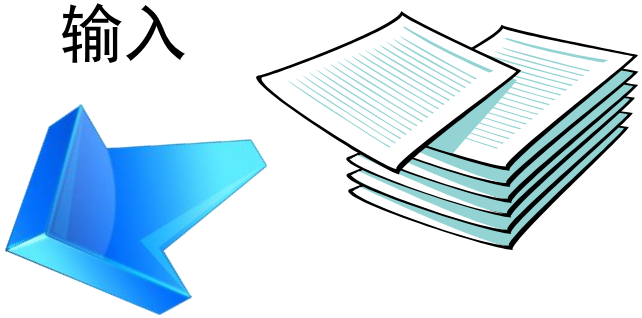
学习内容：

- 背景知识
- 对冯诺依曼架构的改进

1. 背景知识：串行硬件和软件

程序和数据

输入

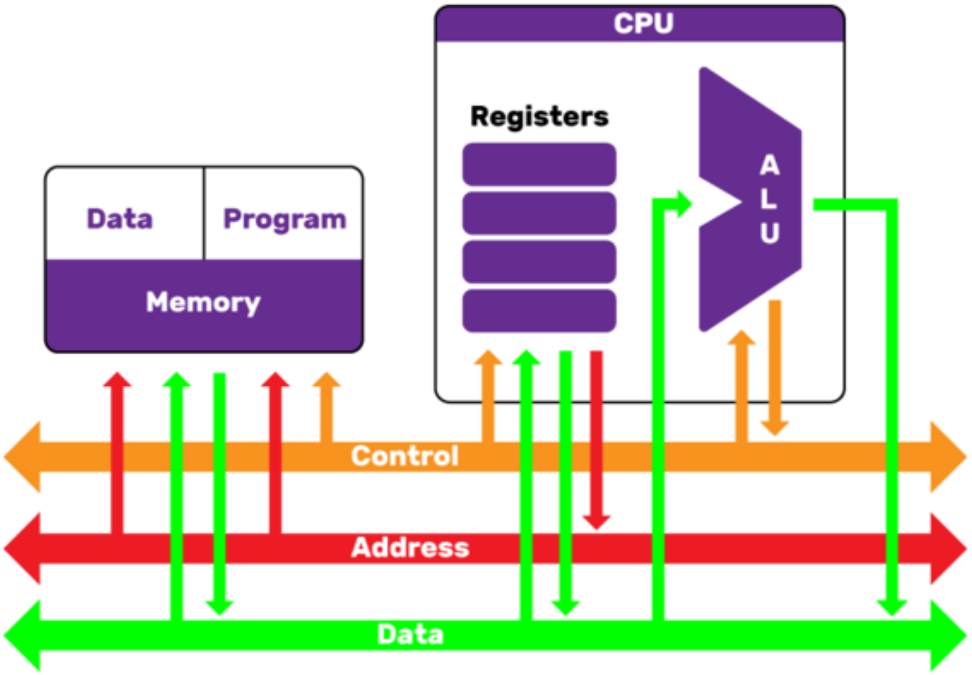


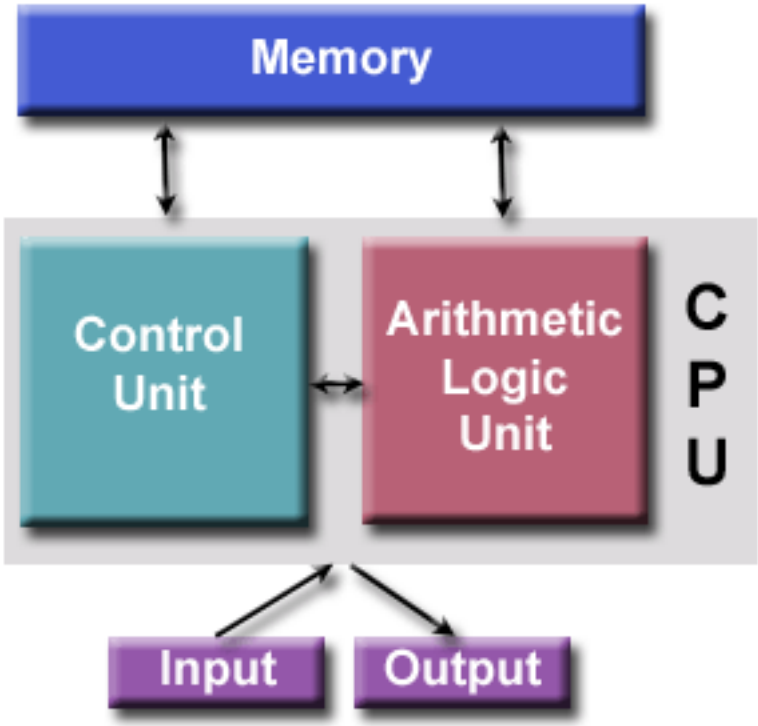
计算机一次运行一个程序

输出

1. 背景知识：冯诺依曼架构（Von Neumann architecture）

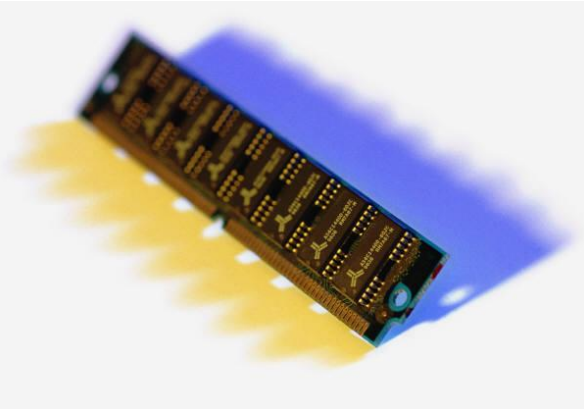
- "stored-program computer"





1. 背景知识：冯诺依曼架构

- 主存（Main Memory）
  - 位置集合，每个位置都能够存储指令和数据
  - 每个位置都包含一个地址，用于访问该位置和其中的内容



1. 背景知识：冯诺依曼架构

- Central processing unit (CPU)
  - 分为两部分：控制单元和算术逻辑单元
  - 控制单元（Control unit）- 负责决定程序中哪一条指令应该被执行（the boss）
  - 算术逻辑单元（ALU - Arithmetic and logic unit）- 负责执行实际的指令（the worker）

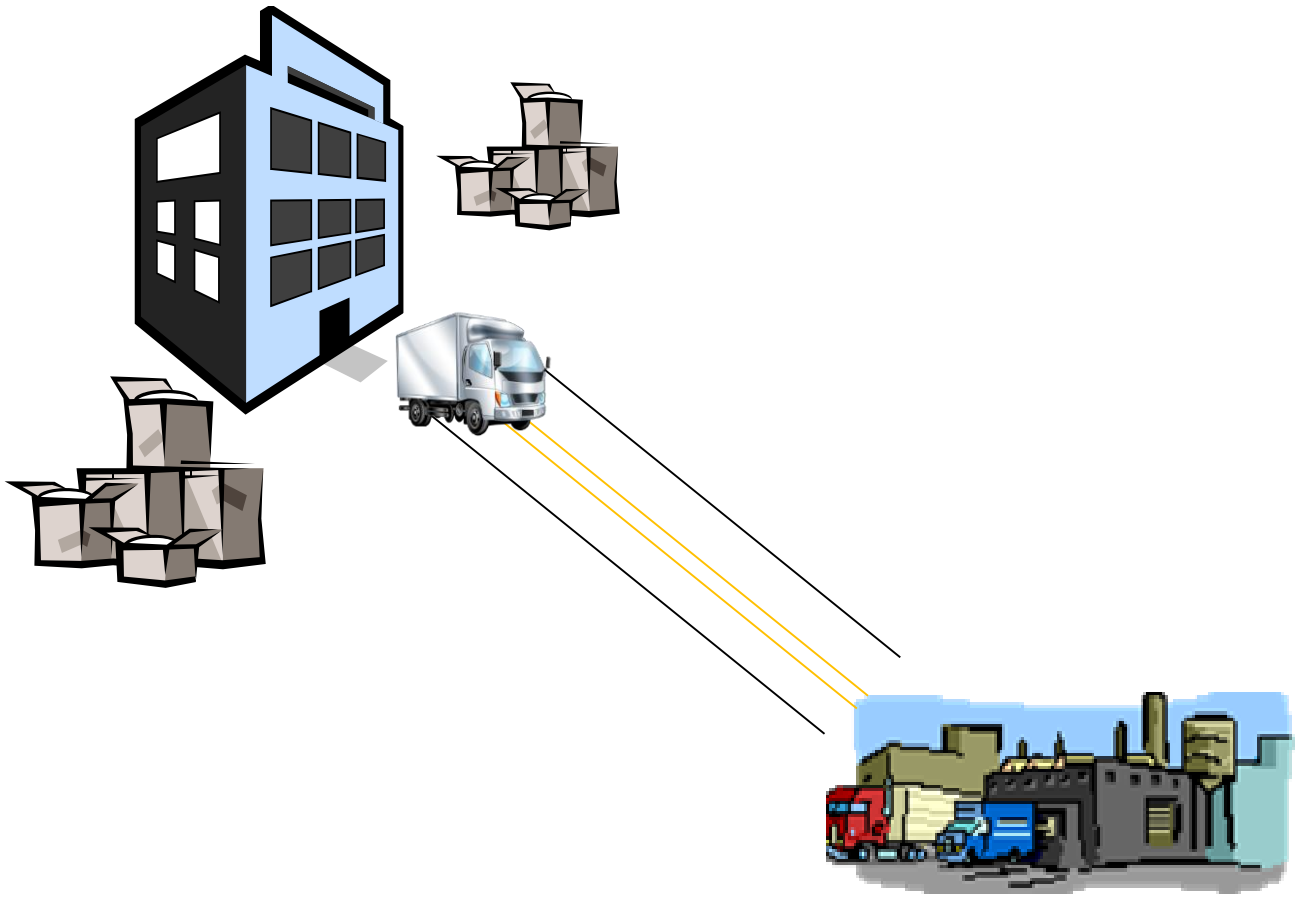
1. 背景知识：冯诺依曼架构

- Central processing unit (CPU)

Registers	Description
MAR (Memory Address Register)	This register holds the memory location of the data that needs to be accessed.
MDR (Memory Data Register)	This register holds the data that is being transferred to or from memory.
AC (Accumulator)	This register holds the intermediate arithmetic and logic results.
PC (Program Counter)	This register contains the address of the next instruction to be executed.
CIR (Current Instruction Register)	This register contains the current instruction during processing.

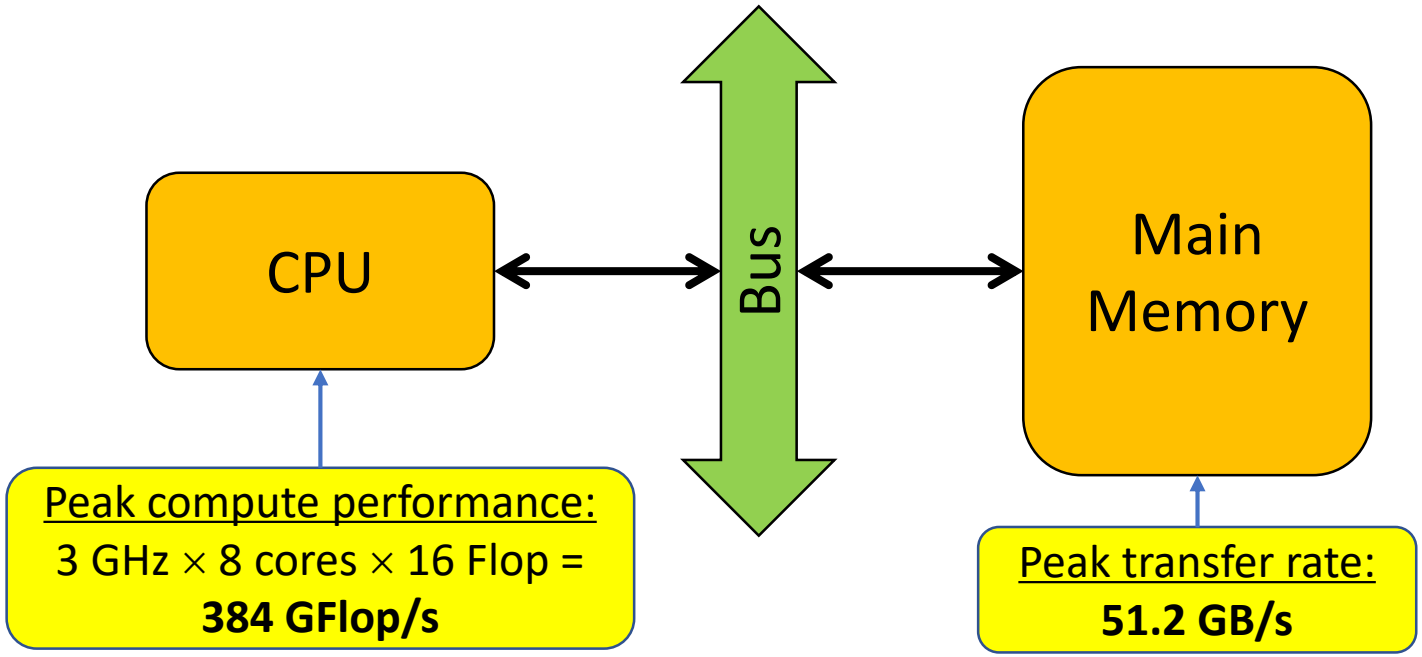
1. 背景知识：冯诺依曼架构

- 瓶颈



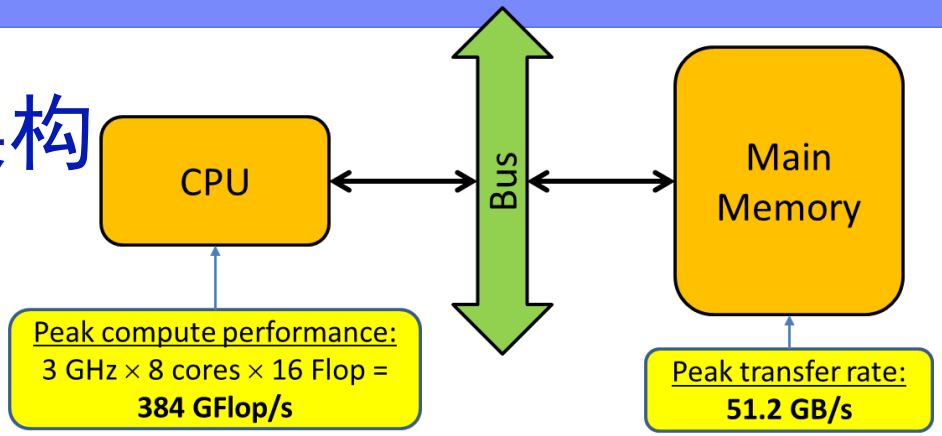
1. 背景知识：冯诺依曼架构

- 瓶颈



1. 背景知识：冯诺依曼架构

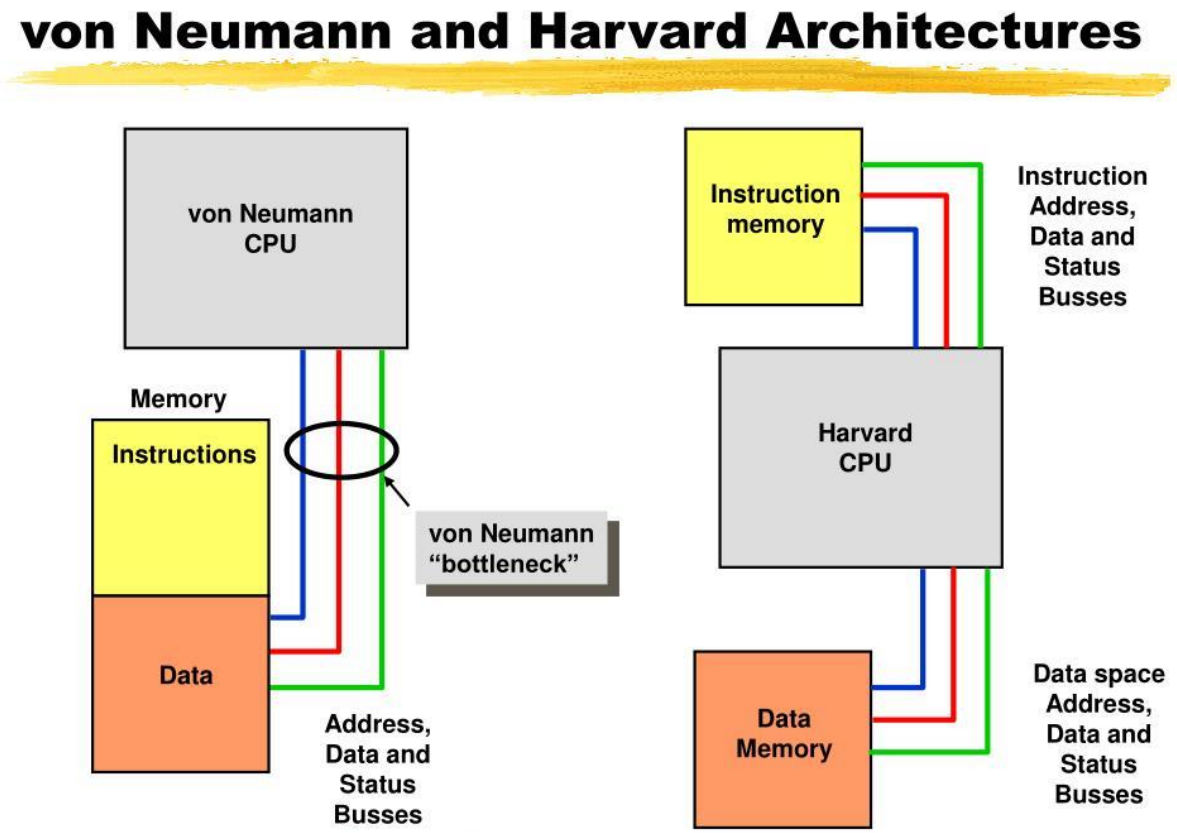
```
// Dot Product
double dotp = 0.0;
for (int i = 0; i<n; i++)
    dotp += u[i] * v[i];
```



- Example: $n = 2^{30}$
- Computation time: $\boldsymbol{t_{comp} = \frac{2\ GFlop}{384\ GFlop/s} = 5.2\ ms}$
  - Total operations: $2 \cdot n = 2^{31}$ Flops = 2 GFlops
- Data transfer time: $\boldsymbol{t_{mem} = \frac{16\ GB}{51.2\ GB/s} = 312.5\ ms}$
  - Amount of data to be transferred: $2 \cdot 2^{30} \cdot 8$ B = 16 GB
- Execution time: $\boldsymbol{t_{exec} \geq max(5.2ms, 312.5ms) = 312.5ms}$
  - Achievable performance: $\boldsymbol{\frac{2\ GFlop}{312.5\ ms} = 6.4\ GFlop/s}$ (<2% of peak)

1. 背景知识：冯诺依曼架构

- *还有其他架构?*



1. 背景知识： Process, multitasking, and thread

- 进程 (Process)：当用户运行一个程序，操作系统创建一个进程（程序的实例）
  - 内存块（可执行代码、堆栈、堆）
  - 操作系统分配给进程的资源的描述符，如：文件描述符
  - 安全信息，如：哪些硬件和软件资源可以被进程访问
  - 进程的状态信息，如：进程是否准备运行还是在等待某些资源、寄存器的内容、进程内存的信息等

1. 背景知识： Process, multitasking, and thread

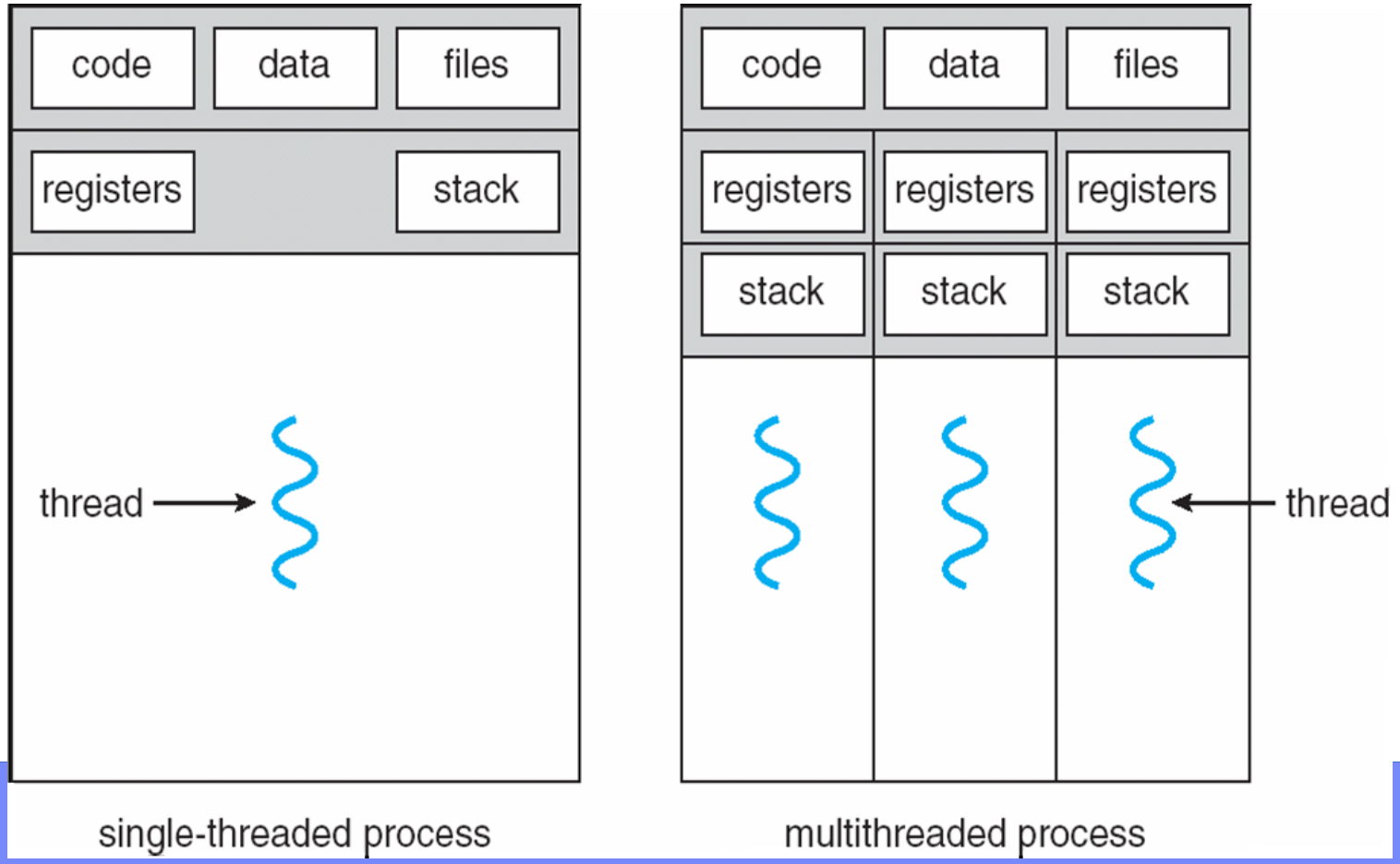
- 多任务 (multitasking)：同时运行多个程序
  - 大多数现代操作系统都是多任务的
  - 给人一种错觉：一个单处理器系统同时运行多个程序
  - 每个进程轮流运行（time slice）
  - 如果进程需要等待资源，将会被阻塞（block）

1. 背景知识： Process, multitasking, and thread

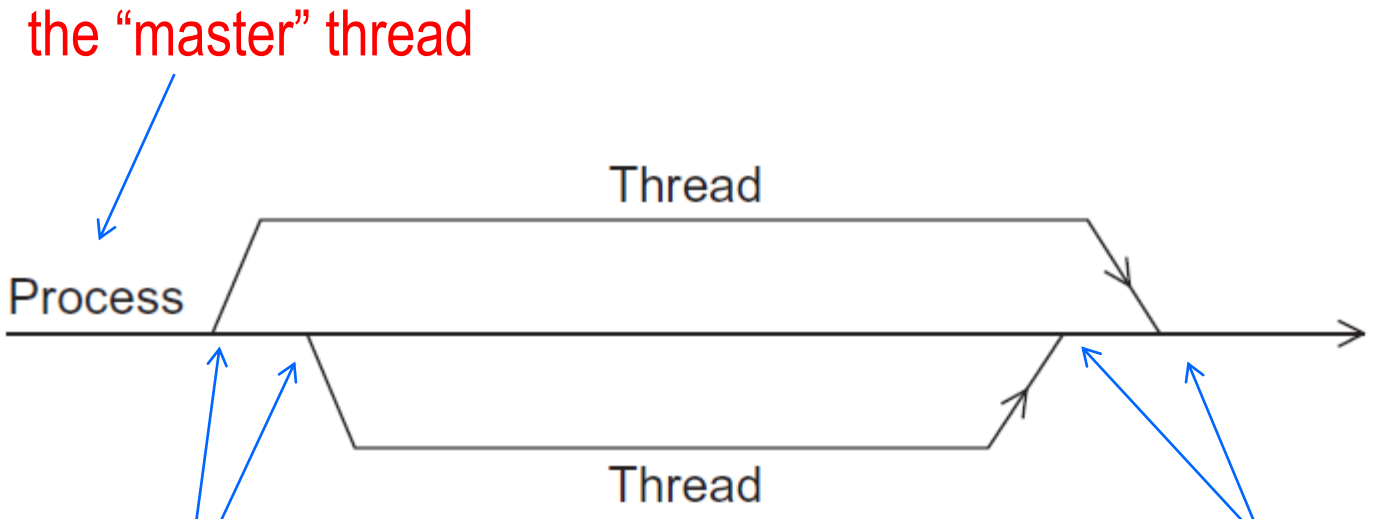
- 线程 (thread)
  - 线程允许程序员将程序划分为独立的任务
  - 当一个线程由于等待资源而阻塞时，其他线程可以继续运行
  - 轻量级，线程间切换要比进程间切换快
  - 线程包含在进程中，可以共享内存的资源（内存、I/O等）
  - 两个例外：线程拥有自己的程序计数器和自己的调用堆栈

1. 背景知识： Process, multitasking, and thread

- 线程 (thread)

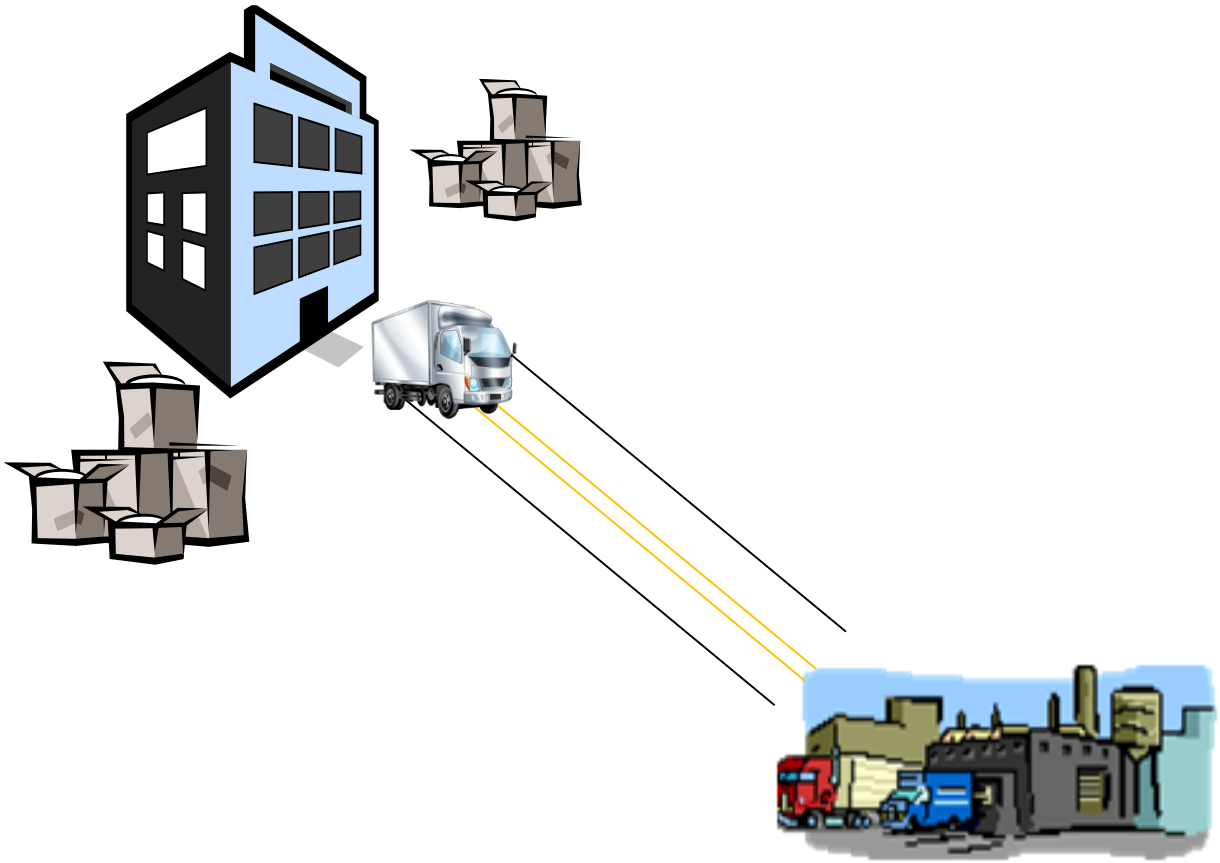


1. 背景知识： Process, multitasking, and thread



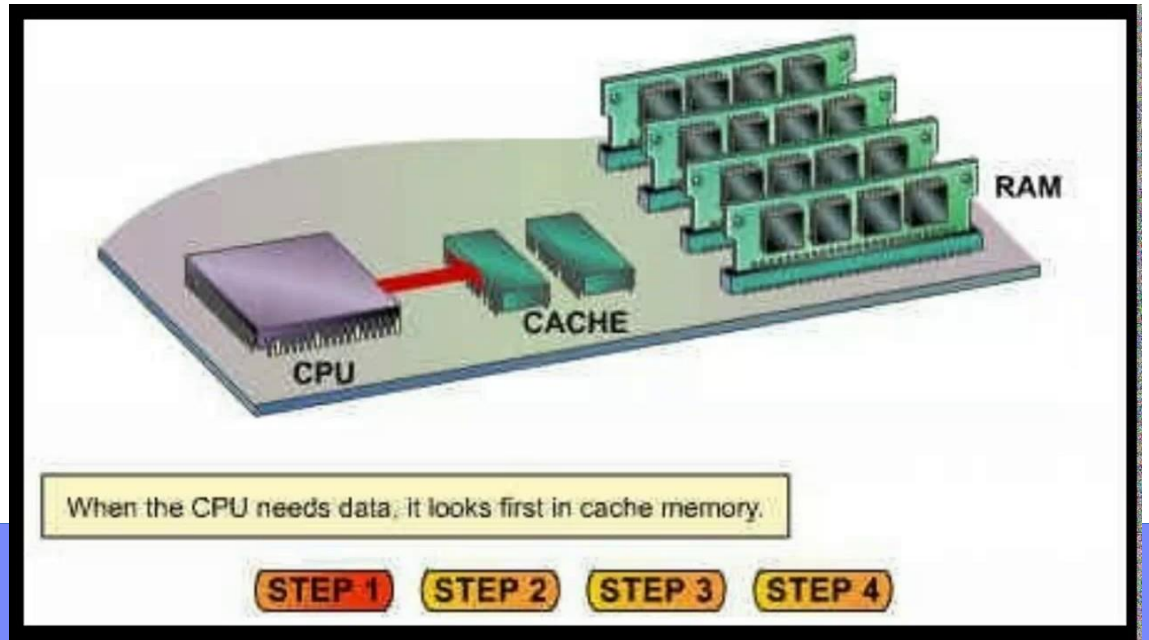
2. 对冯诺依曼架构的改进

- Caching
  - 拓宽道路?
  - 前店后厂?



2. 对冯诺依曼架构的改进

- Caching
  - 一次内存访问不是传输一条指令或数据，而是传送多条指令或数据
  - 不是将所有指令和数据只存储到主存（main memory）中，而是将数据块和指令存储到离CPU的寄存器更近的特殊内存中



2. 对冯诺依曼架构的改进

- Caching
  - 一般来说，缓存（cache）指的是比其他内存访问速度更快的内存位置集合
  - CPU缓存（CPU Cache）可以与CPU位于同一芯片上，也可以位于单独的芯片上，该芯片的访问速度比普通内存芯片快得多
  - *哪些指令和数据放在Cache上？*

2. 对冯诺依曼架构的改进

- Caching
  - 程序倾向于使用与最近使用的数据和指令物理上接近的数据和指令

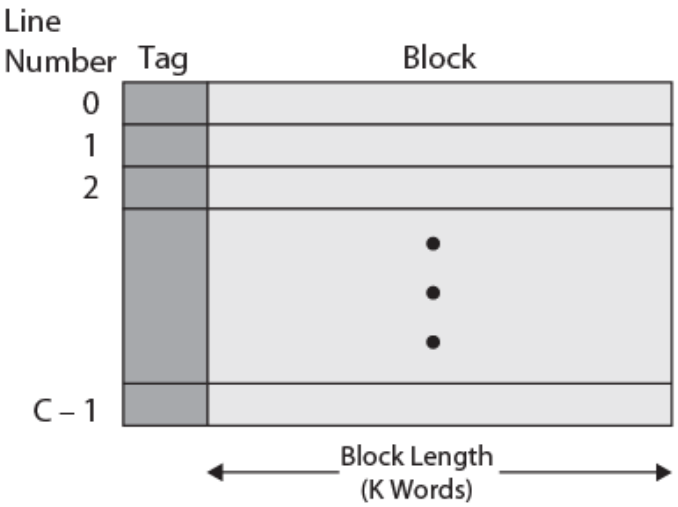
```
float z[1000];
…
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

2. 对冯诺依曼架构的改进

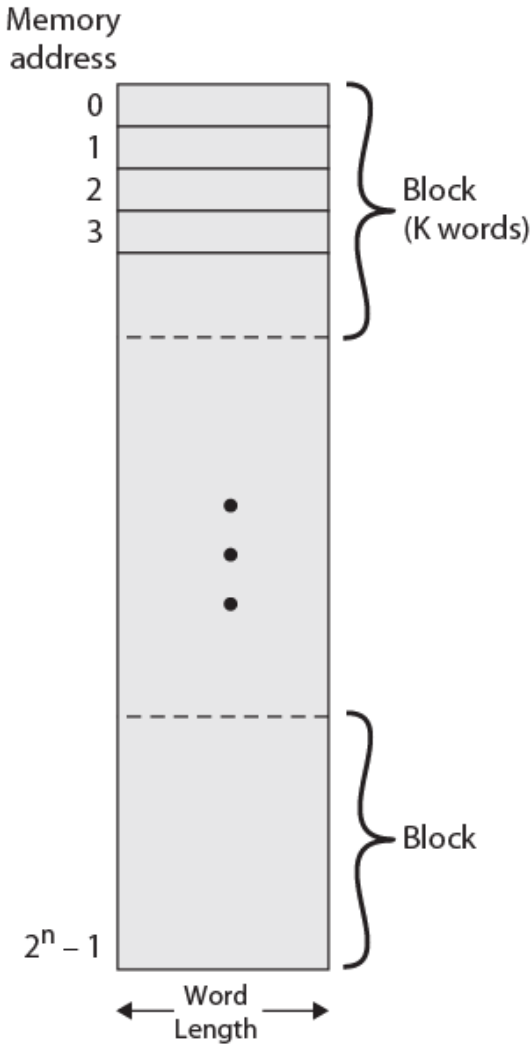
- Caching
  - 局部性原则（Principle of locality）：访问一个内存位置（指令或数据）后，程序通常会在近期（时间局部性）访问附近的内存位置（空间局部性）
  - 根据局部性原则，一次内存访问得到的数据块称为 cache block 或 cache line
  - 通常一个 cache block 或 cache line 是单个内存位置的 8 到 16倍
    - sum += z[0] （如果一个 cache line 存储 16 个 float，则将读取z[0] – z[15]到cache）

2. 对冯诺依曼架构的改进

- Caching



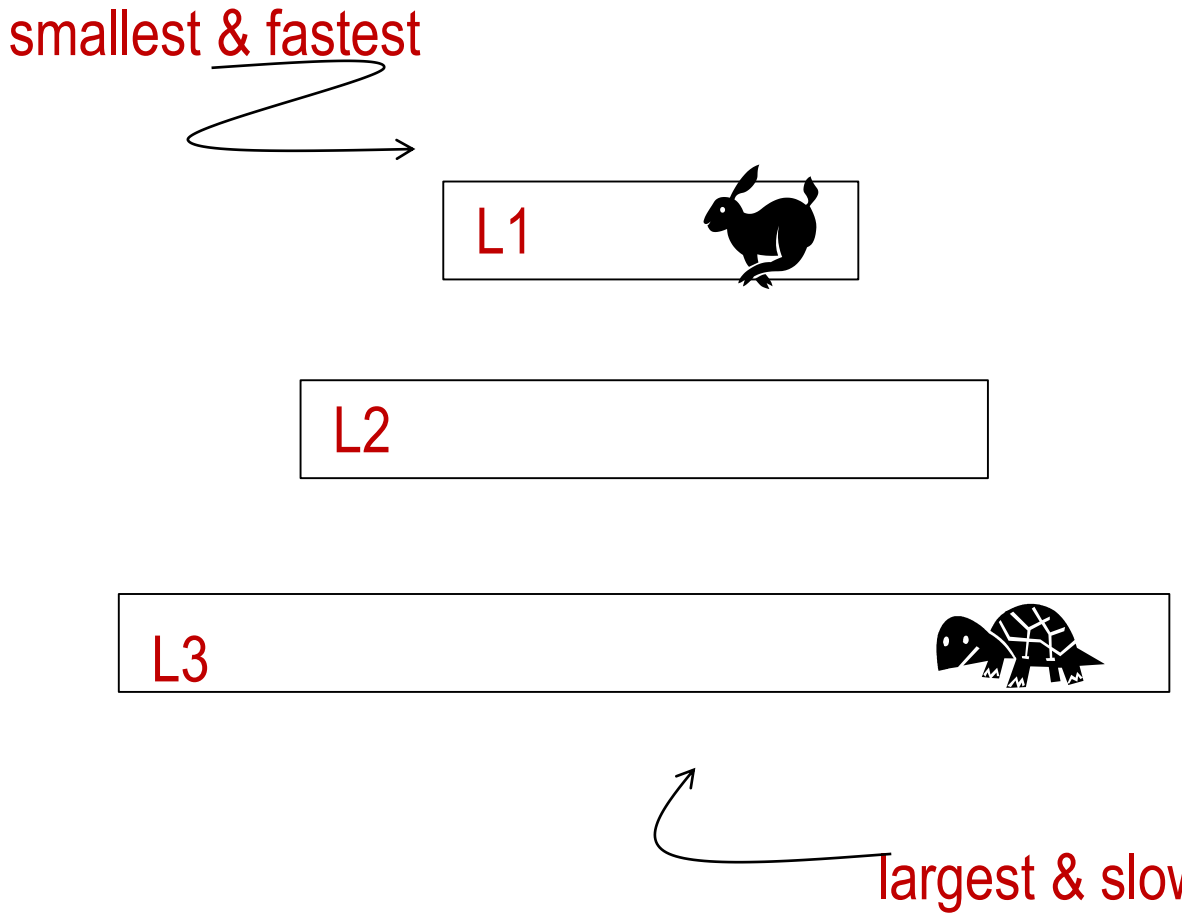
(a) Cache



(b) Main memory

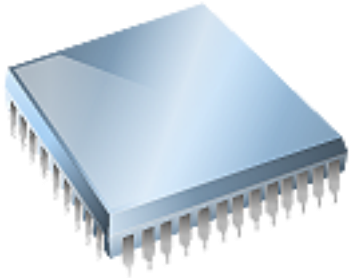
2. 对冯诺依曼架构的改进

- Caching
  - Levels of Cache



2. 对冯诺依曼架构的改进

- Caching
  - Cache hit



fetch x

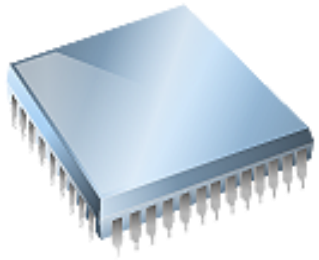
L1 x sum

L2 y z total

L3 A[ ] radius r1 center

2. 对冯诺依曼架构的改进

- Caching
  - Cache miss



2. 对冯诺依曼架构的改进

- Caching
  - 当CPU将数据写入 Cache 时，Cache 中的值可能与主存中的值不一致（inconsistent）
  - 直写（Write-through）：当写入 Cache 时，同时写入主存
  - 写回（Write-back）：当写入 Cache 时，不马上写入主存，而是将 cache 中更新的数据标记为 *dirty*，当该 cache line 被从主存中读取的新的 cache line 所替换时，将该 *dirty line* 写入主存

2. 对冯诺依曼架构的改进

- Caching
  - Cache mapping：从主存中读取的数据存放到 cache 什么位置？
    - 完全关联（Full associative）– 可以存放到缓存的任何位置
    - 直接映射（Direct mapped）– 唯一的对应关系
    - n-way set associative – 可以存放到 n 个不同的位置

2. 对冯诺依曼架构的改进

- Caching
  - Cache mapping
  - 例子：
    - 将16-line主存分配给4-line缓存

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

2. 对冯诺依曼架构的改进

- Caching
  - Cache mapping：当内存中的数据可以映射到缓存中的多个不同位置时，需要能够决定新的数据应该替换哪个 cache line
    - line 0 在位置0，line 2 在位置1，line 4 存放到哪？
  - 替换原则：最近最少使用（least recently used）
    - 缓存中有一个记录，记录块的相对使用顺序
    - 如果 位置 0 最近使用频度高于位置 1，则位置 1 将被替换，即：line 4 存放到位置1

2. 对冯诺依曼架构的改进

- Caching
  - CPU Cache 的工作由系统硬件控制
  - 对于程序员，无法直接决定哪些指令和哪些数据放入 cache
  - 但了解时间和空间局部性原则可以让我们间接控制 caching

2. 对冯诺依曼架构的改进

- Caching
  - 假设 MAX = 4
  - 数组 A 在内存中的存储如下：

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

  - 假设cache为直接映射，且仅有2条cache line，共可存储 8 个数组元素

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];

. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

2. 对冯诺依曼架构的改进

- Caching
  - 哪一个循环更快？
  - 第一个循环会出现几次Cache miss？
  - 第二个循环会出现几次Cache miss？
  - ***4 vs. 16***

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
```

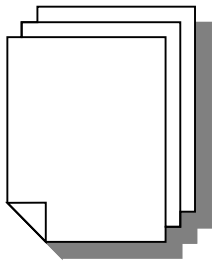
```
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）
  - 如果运行大的程序或访问大的数据集，所有的指令和数据不太可能都装入主存储器
  - 在多任务系统中，许多运行的程序也必须共享可用的主存储器
  - 虚拟内存的作用是将主存储器作为辅助存储（ secondary storage ）的 cache
  - 利用时空局部性原则，仅将运行程序的活动部分保存在主存储器中

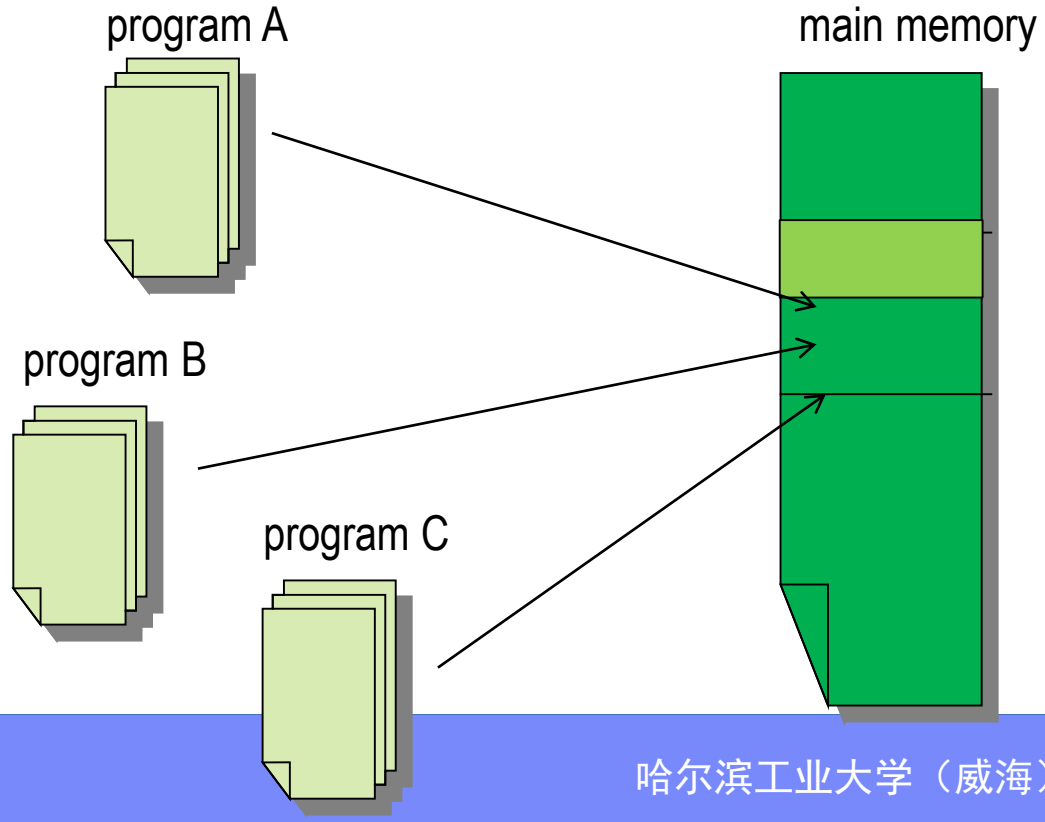
2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）
  - 交换空间（swap space）：其他空闲的，保留在辅助存储器中的部分
  - 页（pages）：blocks of data and instructions
    - 相对较大
    - 大多数系统都有一个固定的 page 大小，范围为4 – 16k



2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）
  - *能否在编译时为page 分配物理内存地址？*

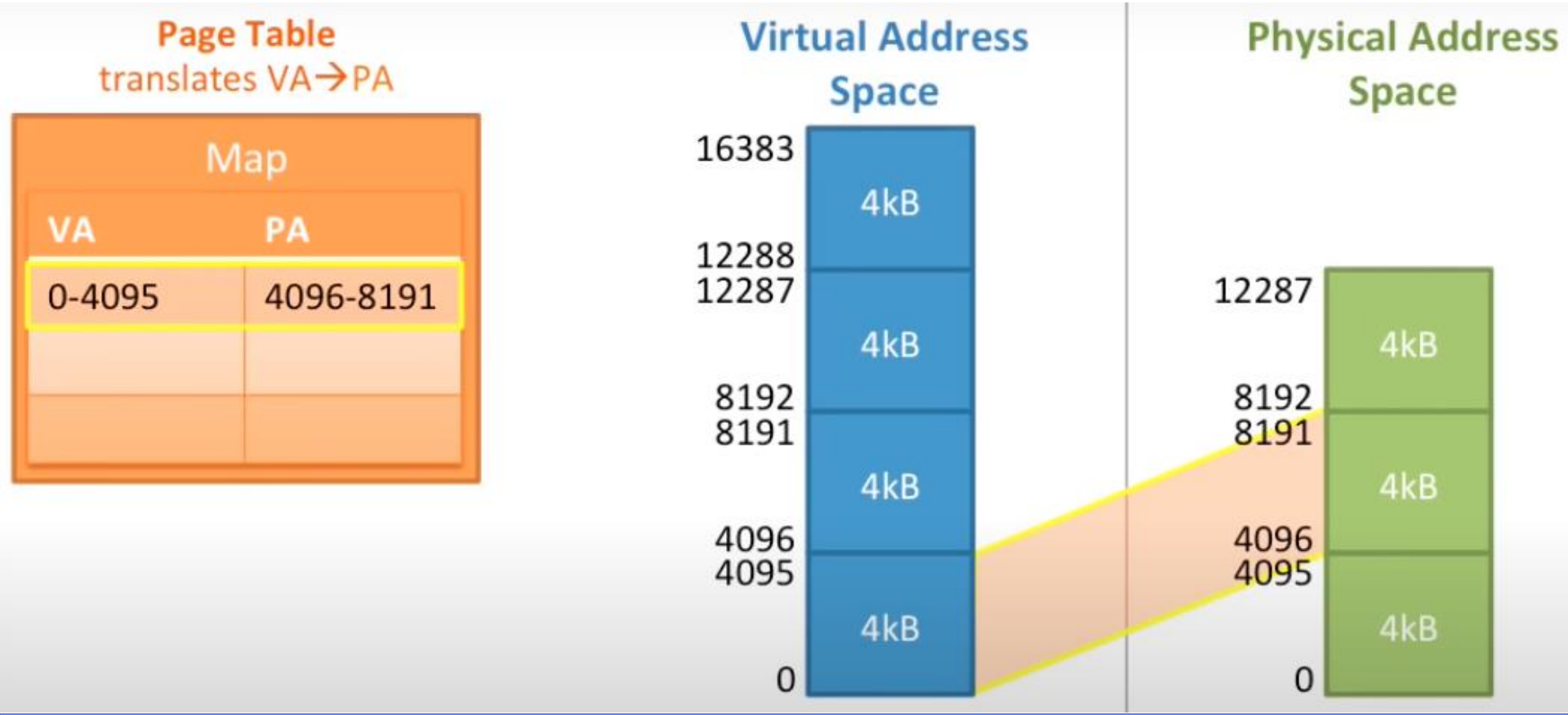


2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）
  - 当程序被编译时，它的 pages 被分配了虚拟页码（virtual page numbers）
  - 当程序运行时，将创建一个表，该表将虚拟页码映射到物理地址
  - Page table 用于将虚拟地址转换为物理地址

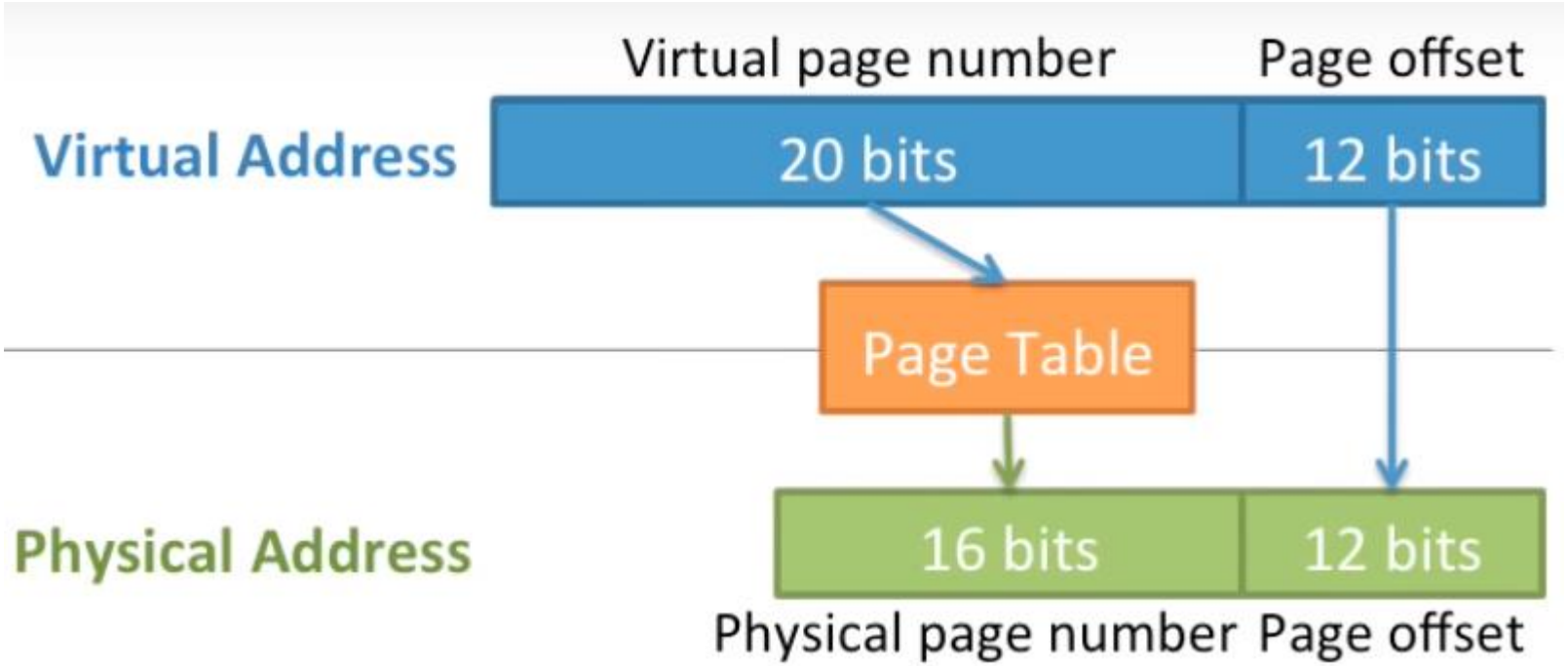
2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）



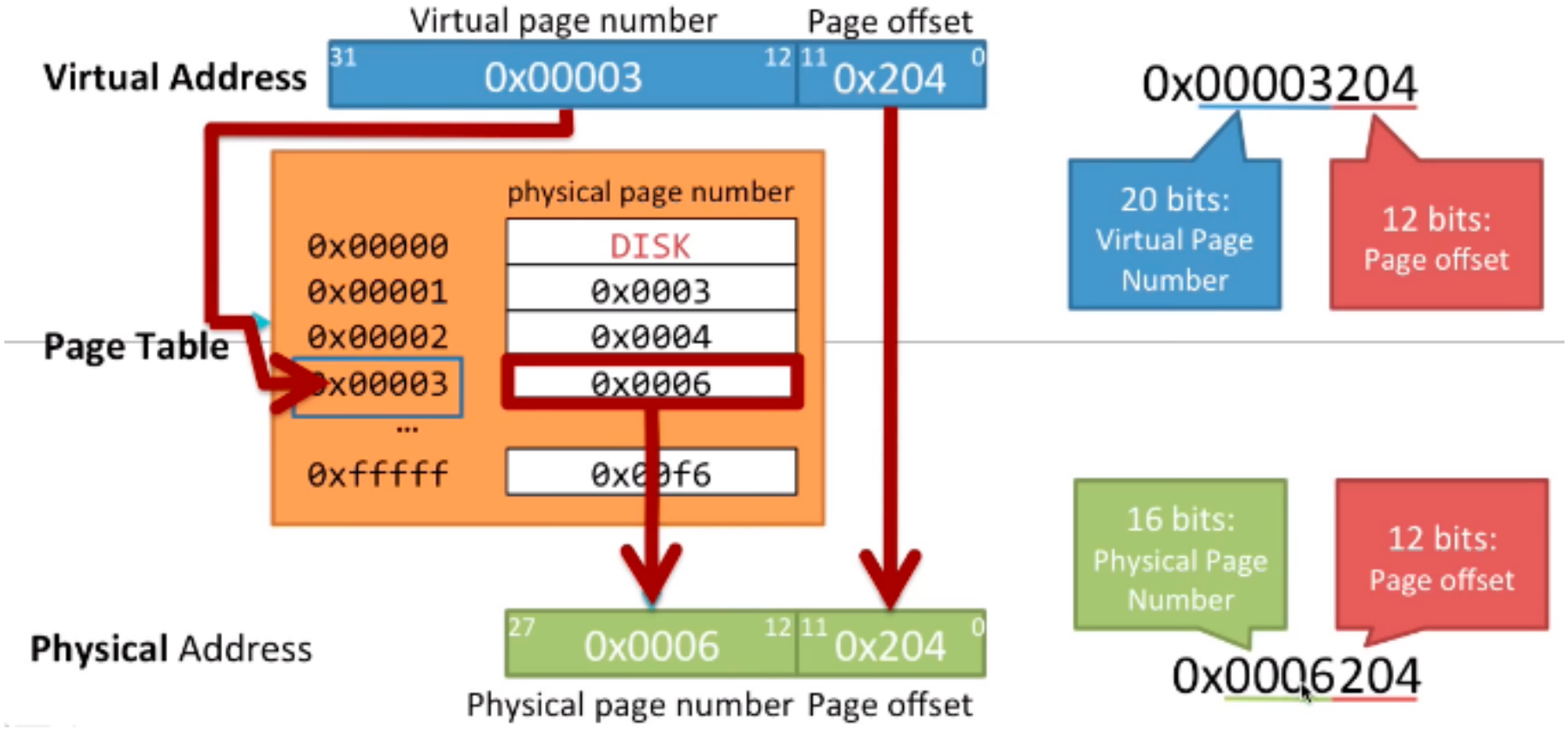
2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）



2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）

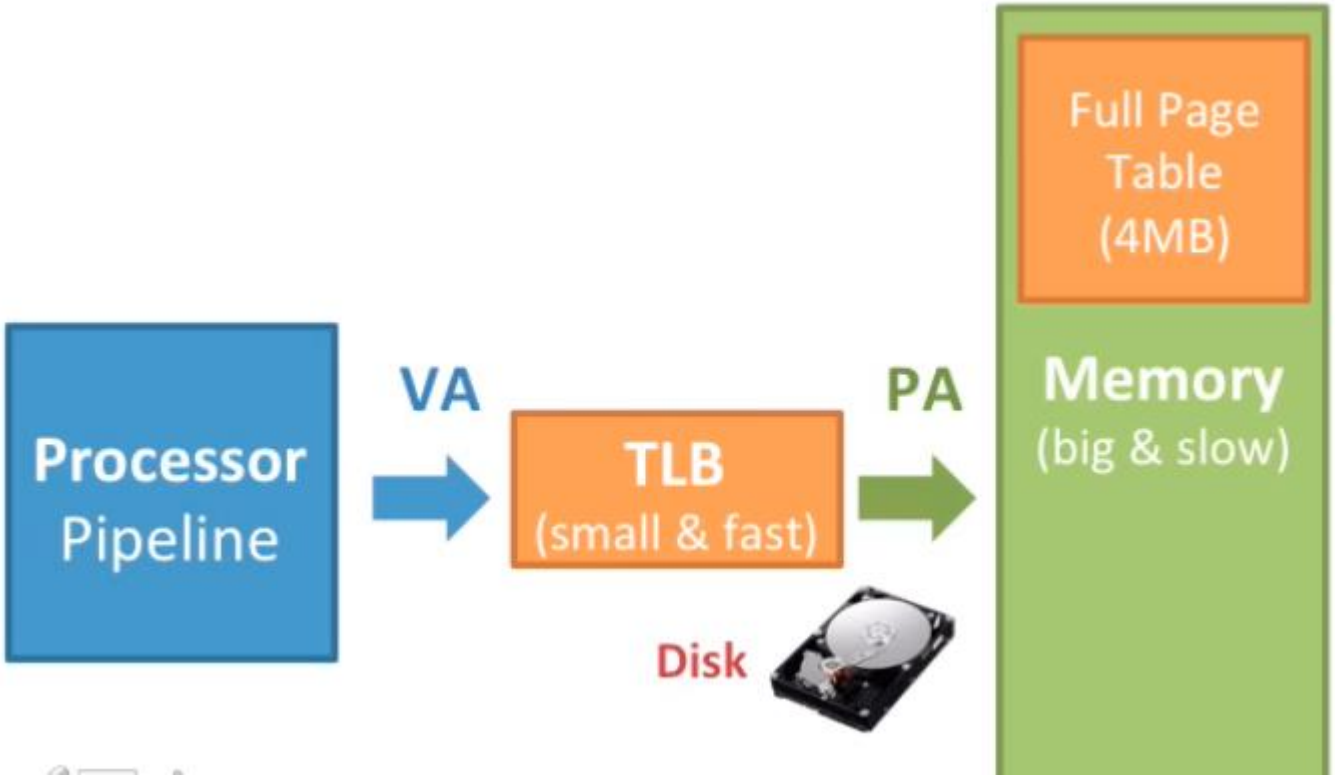


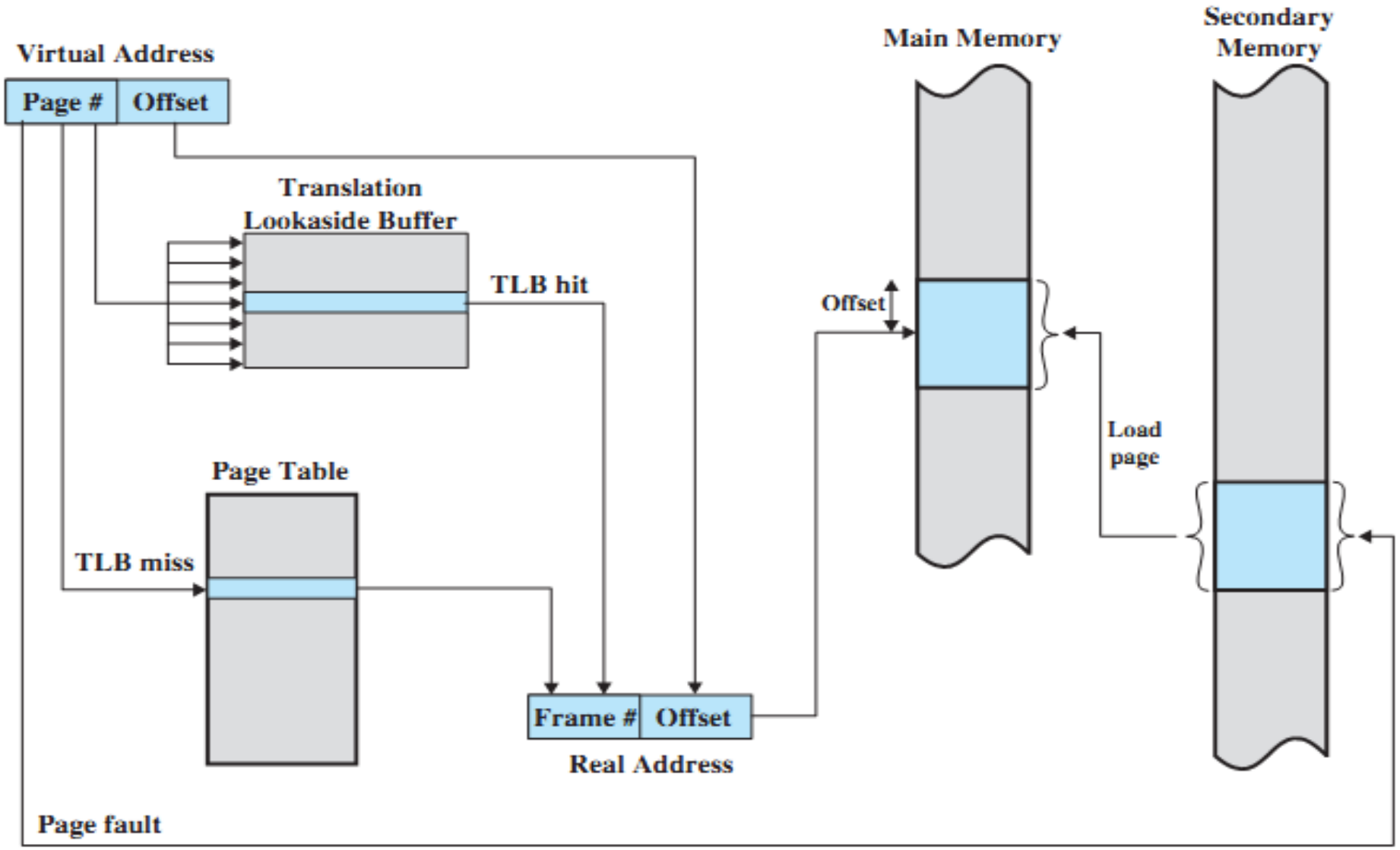
2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）
  - 使用Page table的一个缺点是使访问主存中某个位置所需的时间增加一倍
  - Translation-lookaside buffer (TLB)：处理器中一种特殊的地址转换缓存
  - TLB 缓存 page table 中的少量记录（通常为16 - 512）
  - 页错误（Page fault）– 访问 page table 中页的有效物理地址，但该页存储在磁盘上
  - *Write-through or Write-back ?*

2. 对冯诺依曼架构的改进

- 虚拟内存（Virtual memory）



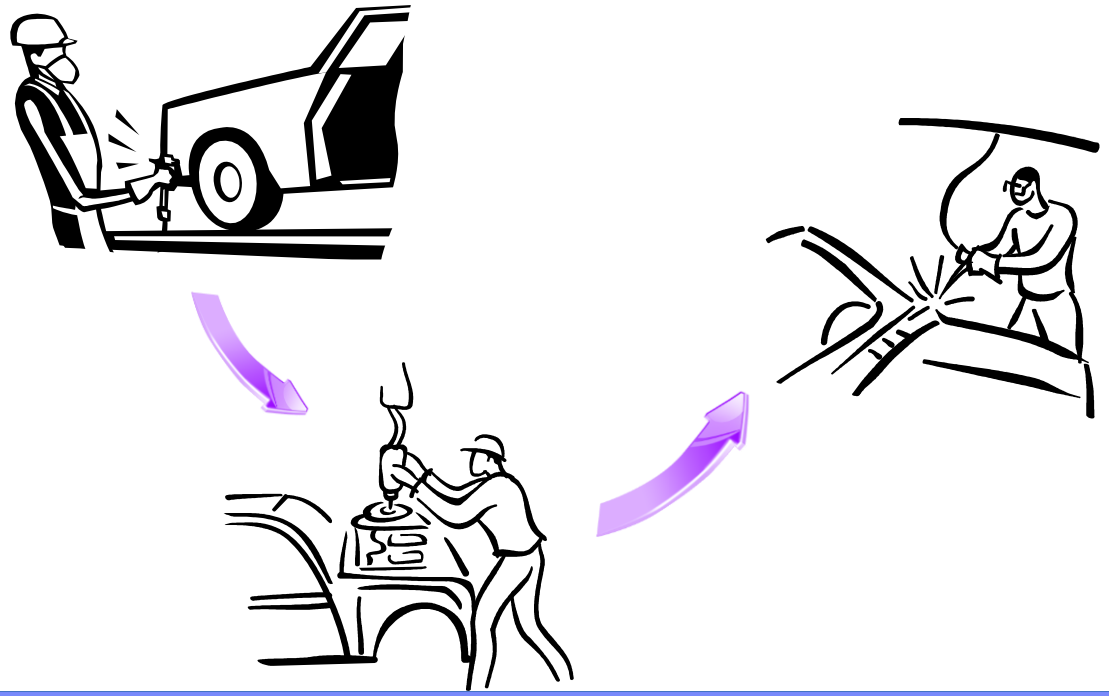

Virtual Address
Page #
Offset
Translation
Lookaside Buffer
TLB hit
Page Table
TLB miss
Frame #
Offset
Real Address
Page fault
Main Memory
Secondary
Memory
Offset
Load
page

2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 试图通过让多处理器组件或功能单元同时执行指令来提高处理器性能
  - 流水线（Pipelining）- 功能单元分阶段布置
  - 多发（Multiple issue）- 可以同时启动多条指令

2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 流水线（Pipelining）- 功能单元分阶段布置



2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

执行两个浮点数相加操作：$9.87 \times 10^4 + 6.54 \times 10^3$

2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 假设每一个操作需要 1 纳秒（ 10^{-9} seconds ）
  - 则循环需要 7000 纳秒完成

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 流水线（Pipelining）
    - 将前面的浮点加法分成 7 个独立的硬件或功能单元
    - 第一个单元：取操作数，第二个单元：比较指数，……
    - 一个功能单元的输出是下一个功能单元的输入

2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 流水线（Pipelining）

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

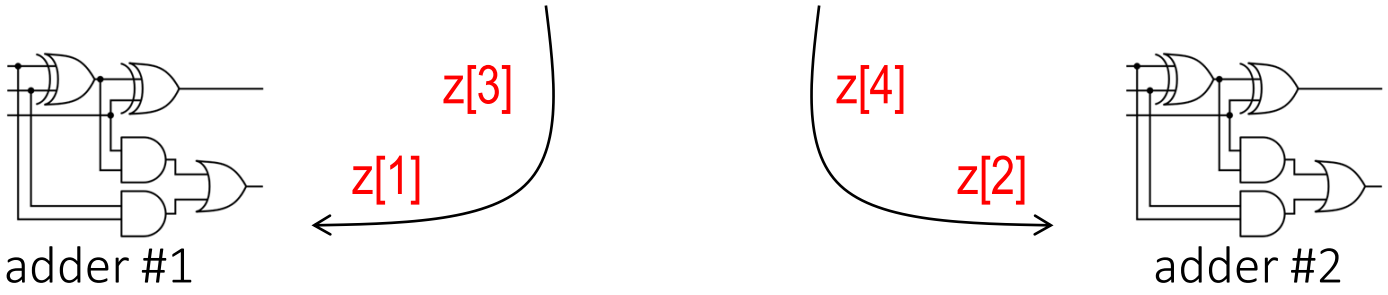
2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 流水线（Pipelining）
    - 一个加法仍用时 7 纳秒
    - 但1000个浮点数的加法用时1006纳秒

2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 多发（Multiple issue）- 复制功能单元并尝试在程序中同时执行不同的指令

$$\text{for } (i = 0;\ i < 1000;\ i{+}{+})$$
$$z[i] = x[i] + y[i];$$

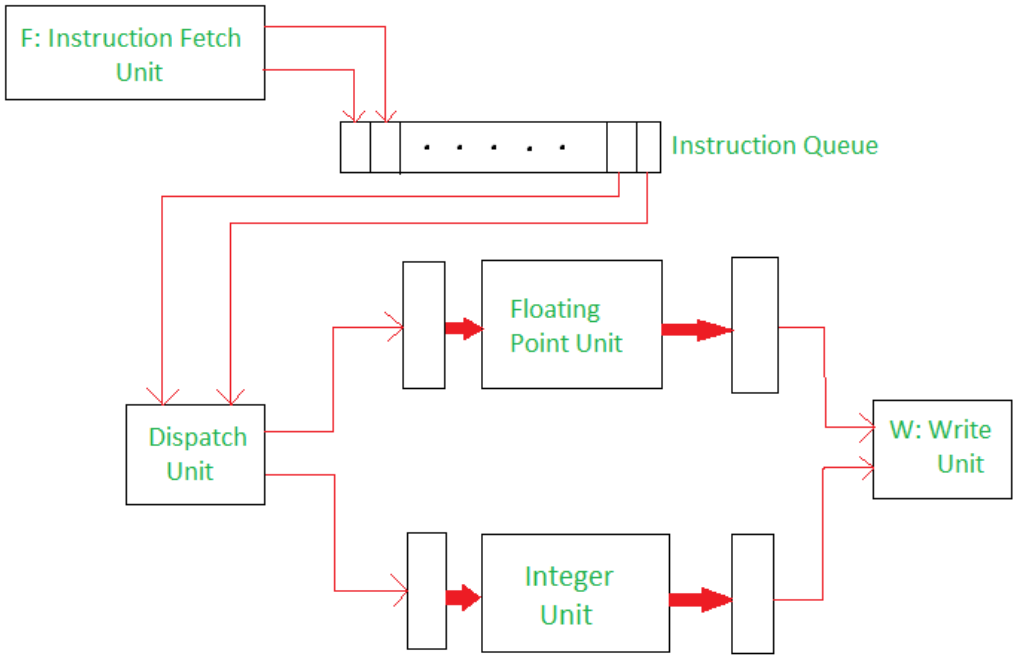


2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 多发（Multiple issue）- 复制功能单元并尝试在程序中同时执行不同的指令
    - static multiple issue – 功能单元在编译时调度
    - dynamic multiple issue – 功能单元在运行时调度
    - 支持 dynamic multiple issue 的处理器称为超标量（superscalar）处理器

2. 对冯诺依曼架构的改进

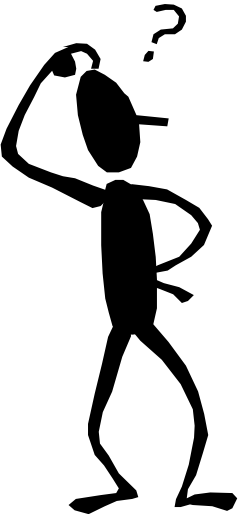
- 指令级并行（Instruction-level parallelism or ILP）
  - 多发（Multiple issue）- 复制功能单元并尝试在程序中同时执行不同的指令



Processor with Two Execution Units

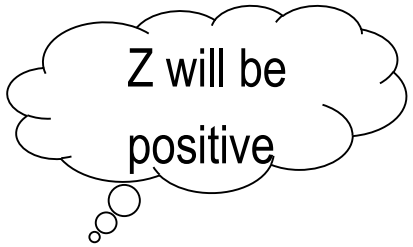
2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 多发（Multiple issue）- 复制功能单元并尝试在程序中同时执行不同的指令
    - 为了利用多发，系统必须找到可以同时执行的指令
    - 推测（speculation）：编译器或处理器对一条指令进行推测，然后根据推测执行该指令



2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 多发（Multiple issue）
    - 推测（speculation）




```
z = x + y ;
if ( z > 0)
    w = x ;
else
    w = y ;
```

如果推测错误呢？

必须退回重新计算 w = y

2. 对冯诺依曼架构的改进

- 指令级并行（Instruction-level parallelism or ILP）
  - 多发（Multiple issue）
    - 推测（speculation）：如果编译器进行推测，通常会插入测试推测是否正确的代码，如果不正确，则采取纠正措施；如果硬件进行推测，处理器通常将推测执行的结果存储在缓冲区中

2. 对冯诺依曼架构的改进

- 硬件多线程（Hardware multithreading）
  - 对于程序中相互依赖的语句，ILP很难被利用

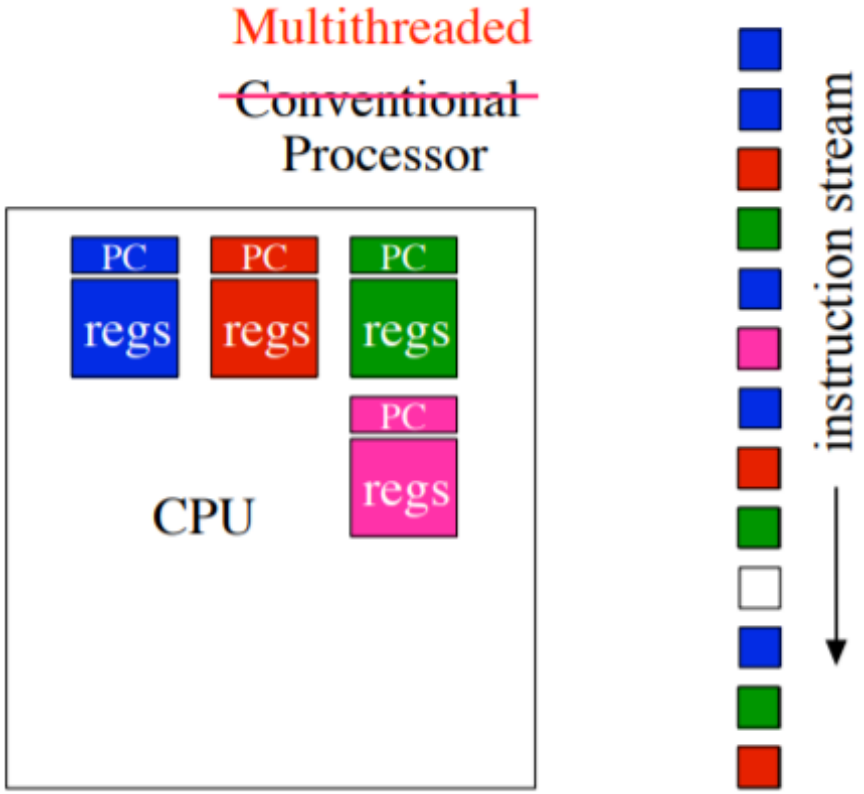
```
f[0] = f[1] = 1;
for (i = 2; i <= n; i++)
    f[i] = f[i - 1] + f[i - 2];
```

2. 对冯诺依曼架构的改进

- 硬件多线程（Hardware multithreading）
  - 线程级并行（Thread-level parallelism, TLP）：通过同时执行不同的线程来提供并行性，提供了比 ILP 更粗的粒度（Coarse-grained）
  - 硬件多线程为系统提供了一种在当前执行的任务暂停时继续执行有用工作的方法，如：当前任务等待从内存加载数据
  - 不在当前执行的线程中寻找并行性，而是运行另一个线程

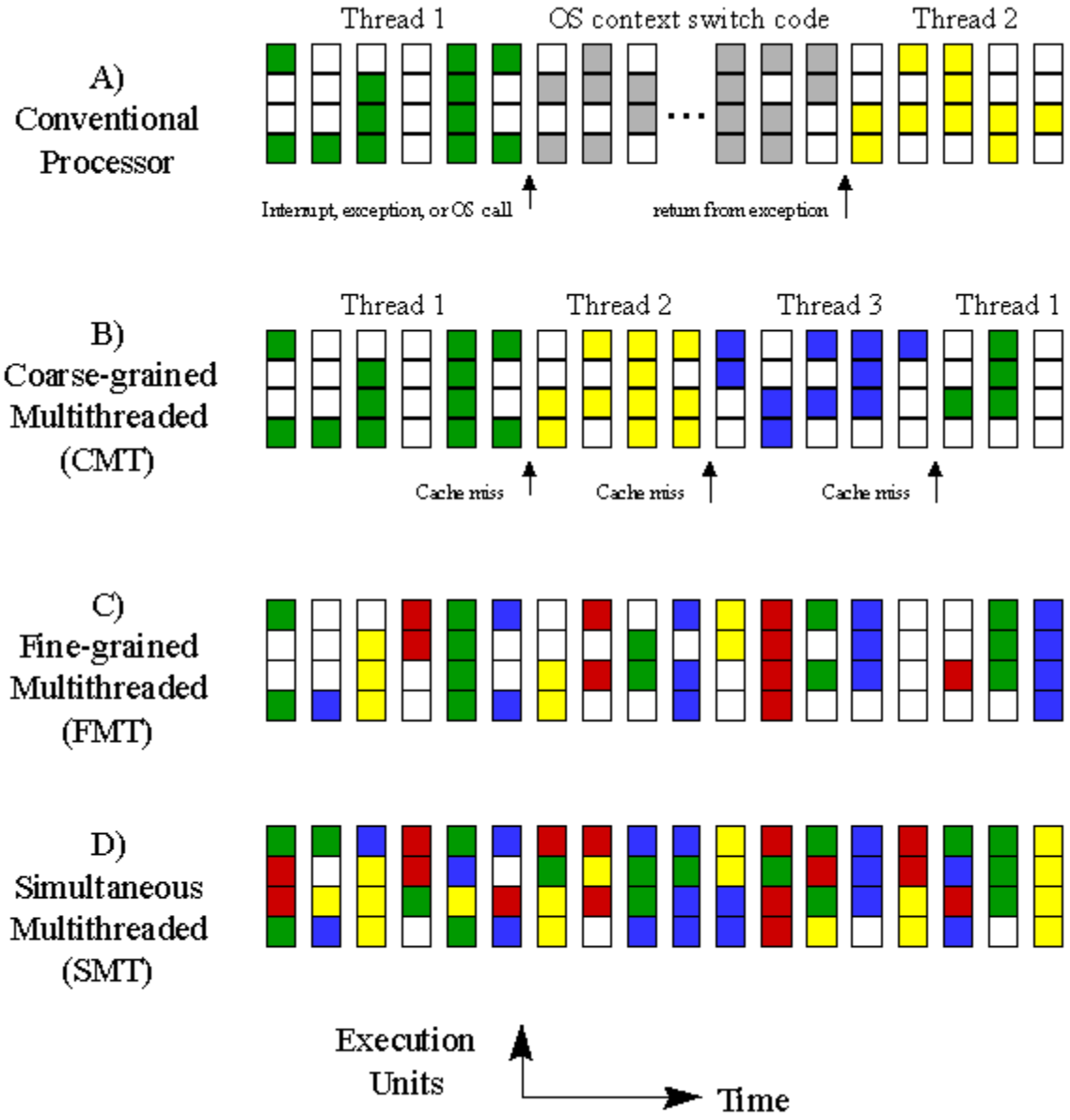
2. 对冯诺依曼架构的改进

- 硬件多线程（Hardware multithreading）



2. 对冯诺依曼架构的改进

- 硬件多线程（Hardware multithreading）
  - 细粒度多线程（ Fine-grained multithreading）- 处理器在每条指令之后在线程之间切换，跳过暂停的线程
  - 粗粒度多线程（ Coarse-grained multithreading）- 只切换那些停下来等待耗时操作完成的线程
  - 同时多线程（Simultaneous multithreading，SMT）是细粒度多线程的一种变体。允许多个线程在超标量处理器上利用多个功能单元


Thread 1
OS context switch code
Thread 2
A)
Conventional
Processor
Interrupt, exception, or OS call
return from exception
Thread 1
Thread 2
Thread 3
Thread 1
B)
Coarse-grained
Multithreaded
(CMT)
Cache miss
Cache miss
Cache miss
C)
Fine-grained
Multithreaded
(FMT)
D)
Simultaneous
Multithreaded
(SMT)
Execution
Units
Time

小结

- 背景知识
  - 冯诺依曼架构；进程、线程
- 对冯诺依曼架构的改进
  - Cahche
  - 虚拟内存
  - 指令级并行（ILP）
  - 硬件多线程