1 word = 4 bytes, 1 byte = 8 bits, each element in an array cost 1 word

Datapath

Five stages

1. IF（指令获取）：在这个阶段，计算机从内存中获取下一条指令。这个阶段的主要任务是从程序存储器（通常是指令缓存或主存）中读取指令并将其送入流水线的下一个阶段，即ID阶段。
2. ID（指令译码）：在ID阶段，计算机对从IF阶段获取的指令进行译码和解析。这包括识别指令的操作码（opcode）以及读取和解释操作数。还可能包括寻址模式的计算，以确定需要访问的内存地址或寄存器。
3. EX（执行）：在EX阶段，计算机执行指令的实际操作。这包括算术和逻辑运算，例如加法、减法、乘法、逻辑与或等。这个阶段还包括控制单元生成的控制信号，用于控制数据通路的运算。
4. MEM（内存访问）：在MEM阶段，如果指令涉及内存操作（如加载或存储），则计算机访问内存以读取或写入数据。这个阶段还可以包括访问高速缓存（如果有的话）以提高性能。
5. WB（写回）：在WB阶段，计算机将结果写回寄存器文件（通常是通用寄存器）。这个阶段确保计算的结果可供后续指令使用。

Exception

- 对于大部分指令，其运行过程都要经历IF，ID和EX这三个步骤

EX

在计算机指令执行中，是否需要EX（执行）阶段取决于指令的类型和具体架构。一些简单的指令可能不需要EX阶段，因为它们不涉及复杂的运算。以下是一些示例：

1. 数据传送指令：数据传送指令用于将数据从一个地方传送到另一个地方，通常涉及寄存器之间的数据传输。这些指令通常不需要进行算术或逻辑运算，因此不需要EX阶段。
2. 简单算术指令：某些简单的算术指令，例如加法或减法，可能不需要EX阶段，因为它们可以在ID阶段执行。这通常适用于架构中具有较简单ALU（算术逻辑单元）的情况。
3. 分支指令：分支指令用于控制程序流程，它们不一定需要EX阶段，因为它们的执行是基于条件判断的，而不涉及算术或逻辑运算。

MEM or WB

1. 寄存器-寄存器操作：一些指令只涉及寄存器之间的操作，例如寄存器之间的数据传送、寄存器的加法或减法等。这些操作通常不需要MEM或WB阶段，因为它们不涉及内存或外部设备。
2. 算术和逻辑指令：简单的算术运算（如加法、减法、乘法、除法）和逻辑运算（如与、或、非）通常不需要MEM或WB阶段，因为它们的操作都在寄存器之间完成。
3. 控制指令：控制指令用于修改程序流程，例如条件分支或跳转。这些指令通常不需要MEM或WB阶段，因为它们不涉及数据的读取或写入。

Instruction Running Time

Single Cycle

- 选择**耗时最长的指令**作为cycle time
- 所有指令的运行时间为cycle time

Multi Cycle

- 选择**耗时最长的Stage**作为基准时间
- 所有指令的运行时间根据需要运行的Stage不同，计算出不同的cycle time
- 对于不需要内存操作的non-branch指令：`sll`，`add`，`addi`，`slt`，需要计算IF+ID+EX+WB = 基准时间 * 4
  - 对于branch指令：`beq`，`bne`，`j loop`，需要计算IF+ID+EX = 基准时间 * 3
  - 对于`lw`：需要计算IF+ID+EX+MEM+WB = 基准时间 * 5
  - 对于`sw`：需要计算IF+ID+EX+MEM = 基准时间 * 4

Pipeline

- 选择**耗时最长的Stage**加上可能的pipeline寄存器延迟作为基准时间
- 对于M个指令，假设需要经过N个stage (5)：
- Pipeline time = (M + N - 1) * (基准时间)
  - Number of Cycle = M + N - 1

Cache

- 内存地址的存放一般为Tag + Index + Offset

Direct Mapped Cache

在直接映射缓存中，主存储器（main memory）被划分为多个块（blocks），而缓存（cache）则被划分为多个线（lines）或槽（slots）。这些缓存线用于存储来自主存储器的数据块。当CPU需要读取特定地址的数据时，系统会检查这些数据是否已在缓存中。
为了决定一个内存块应该存储在缓存的哪个位置，以及如何在缓存中找到这些块，系统会使用一种映射策略。在直接映射缓存中，使用了内存地址的一部分来确定一个特定的缓存线。

内存地址通常包含以下三个部分：

1. **标记（Tag）**：用于唯一标识一个内存块。当缓存检查某个特定缓存线时，它会比较地址标记和缓存线中存储的标记，以确定所需数据是否在该缓存线中。
2. **索引（Index）**：这是直接决定内存块在缓存中位置的部分。系统通过内存地址的索引字段来选择应该使用哪个缓存线。换句话说，索引用于“直接映射”内存块到特定的缓存线。
3. **块内偏移（Block Offset）**：当存储块大小大于一个字（word）时，块内偏移用于在一个块内部定位特定的字。

计算Tag

"标记"用于在缓存中唯一标识一个数据块。由于缓存容量比整个内存小，因此不可能将所有内存块同时加载到缓存中。标记用于区分不同的内存块，即使它们可能映射到缓存的同一索引位置。

```
Tag = Block Number / Number of Total Cache Block
```

计算Index

即我们需要最少的比特位数来表示所有的block。如果一共有2^M个内存块block，那么我们需要M个bit位才能表示所有的block，所以block number的最后M位即位索引位

```
Cache Index = Block Number % Number of Total Cache Block
```

计算Offset

假设一个内存块的大小为2^N bytes，则offset为N bit

Valid Bit

valid bit代表这个slot有没有被写入过数据。在起始状态，所有的valid bit都为0.在判断缓存命中时需要判断valid bit为1才能正确命中

Set Associative Cache

> Set Associative (SA) Cache是一种折中的缓存映射策略，结合了直接映射缓存 (Direct Mapped Cache) 的高效性和全关联缓存 (Fully Associative Cache) 的灵活性。它试图在这两种策略的优势之间找到平衡，减少缓存未命中的次数，同时保持合理的硬件复杂度和成本。

> Set Associative缓存的工作原理如下：
>
> 1. **集合（Set）：** 缓存被划分为多个集合（sets），每个集合包含几个缓存行（cache lines）或块（blocks）。这些块是缓存中可以存储数据的单位。
> 2. **关联度（Associativity）：** 每个集合中的块数定义了缓存的“关联度”。例如，如果每个集合有四个块，则该缓存是4路组相联的。
> 3. **映射：** 当主存中的一个块需要被加载到缓存中时，首先会根据该块的地址计算出它应该存储在哪个集合中。这个计算过程通常基于地址的某些位，并且每个集合对应主存中的多个块。然而，一个给定的块只能映射到一个特定的集合。
> 4. **替换策略（Replacement Policy）：** 如果计算出的目标集合已满（即每个块都已被其他数据占用），缓存必须决定哪个块将被替换以腾出空间。这是通过所谓的替换策略来完成的，常见的替换策略有最近最少使用（LRU）、随机（Random）等。
>
> 通过这种方式，Set Associative缓存降低了发生冲突缺失（多个内存地址映射到同一缓存位置）的可能性，因为现在每个内存块有多个潜在的缓存位置可供选择。这增加了一些查找所需数据的复杂性（因为现在必须在一个集合的所有块中查找），但通常能够显著提高缓存的命中率，尤其是在工作集大小适中，且访问模式较为分散的情况下。

计算Offset, (set) index, Tag

- 假设内存块的大小为2^N，则offset为N bits
- 假设集合(Set)的数量为2^M，则index为M bits
- Tag = 32 - (N+M) bits

Cache Miss

1. **Compulsory Misses（强制性缺失）：**
    - 这类缺失发生在对一个数据块的首次访问；因为数据块还没有被加载到缓存中，所以必须从更低一级的内存（如主内存）中取出数据块。
    - 这也被称为冷启动缺失（cold start misses）或首次引用缺失（first reference misses），因为它们通常发生在程序刚开始运行时，此时缓存未被充分利用。
2. **Conflict Misses（冲突缺失）：**
    - 这类缺失发生在直接映射缓存（direct-mapped cache）或组相联缓存（set-associative cache）中，当多个数据块映射到同一个缓存块或集合（set）时。如果新来的数据块与已经在缓存位置的数据块发生冲突，已在缓存中的数据块将被替换出去。
    - 这也被称为碰撞缺失（collision misses）或干扰缺失（interference misses），因为它们是由于不同数据块之间的映射冲突导致的。
3. **Capacity Misses（容量缺失）：**
    - 这类缺失发生在缓存无法容纳所有需要的数据块时。如果程序访问的数据块数量超过了缓存的容量，缓存中的一些数据块将被替换出去，以便为新的数据块腾出空间。当再次需要被替换出去的数据块时，就会发生容量缺失。
    - 这种情况表明，即使缓存中没有冲突，由于缓存容量本身的限制，也无法避免缺失的发生。

Cache Write Policy

当数据被修改后，缓存中的信息和主内存中的信息可能会不一致。为了处理这种不一致性，有两种常见的策略：

1. **写直达（Write-through）缓存：**
    - 在这种策略中，当缓存中的数据被修改时，同时也会将修改的数据写入到主内存中。这样可以确保主内存中的数据始终是最新的，也就是说，缓存和主内存中的数据始终是一致的。
    - 优点是简单、一致性好，当缓存中的数据发生更改时，不需要额外的操作就能保证与主内存的同步。
    - 缺点是每次写操作都需要访问主内存，这会带来额外的时间开销，尤其是当写操作非常频繁时。
2. **写回（Write-back）缓存：**
    - 在这种策略中，修改后的数据仅仅被写回到缓存中，而不是立即写入主内存。只有当缓存中的数据需要被替换，也就是说，当新的数据需要加载到缓存中而缓存已满时，缓存中被修改过的数据（脏数据）才会被写回主内存。
    - 优点是减少了对主内存的写入操作，因为不是每次缓存数据更新都要写入主内存，这可以提高系统的整体性能，特别是在写操作频繁的场景下。
    - 缺点是一致性问题更加复杂。在多核或多处理器系统中，如果不同的处理器缓存中存储了同一主内存位置的不同副本，就需要更复杂的一致性协议来避免数据不一致的问题。

问题：

- 使用写直达策略时，每次写操作都会同时更新缓存和主存（main memory）。由于主存的写速度通常远低于缓存和处理器的速度，因此每次写操作都会被主存的慢速度拖慢，这影响了整个系统的性能。

解决方案：

- 为了解决这个问题，系统中引入了一个“写缓冲区”（write buffer）。写缓冲区是一种快速存储器，位于缓存和主存之间。
  - 当处理器执行写操作时，它只需要将数据写入缓存和写缓冲区。由于这两者的速度都很快，处理器不会因为等待写操作完成而闲置，从而可以继续执行后续操作。
  - 同时，内存控制器（memory controller）会在后台处理写缓冲区中的数据，将其写入较慢的主存中。这一步是异步进行的，不会影响处理器的正常工作。

问题：

- 使用写回策略时，如果我们在替换缓存块时每次都将其写回主存，这将非常浪费资源，因为不是所有被替换的缓存块都包含更新过的数据。在一些情况下，缓存块的数据可能自从被加载到缓存以来并没有被修改过，因此将其写回主存是不必要的。

解决方案：

- 为了更有效地管理缓存内容的写回，系统引入了“脏位”（dirty bit）这一概念。脏位是附加在每个缓存块上的一个额外的位。
  - 当处理器写入缓存时，它只更新缓存块中的数据，并将对应的脏位设置为1，表明该缓存块已被修改，与主存中的相应块内容不一致。这个过程中，并不会有数据写回到主存。
  - 当需要替换缓存块时，系统会检查脏位。如果脏位为1（表示缓存块已被修改），系统才将该缓存块的内容写回主存。如果脏位为0（表示缓存块自加载后未被修改），则无需进行写回操作。