Lecture 11 开发存储器层次结构

1. 介绍

局部性原理

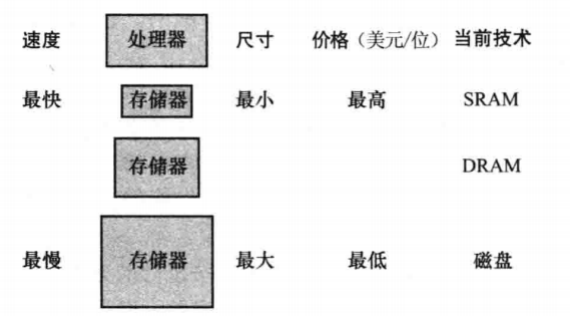
- **时间局部性 temporal locality**：如果某个数据项被访问，那么在不久的将来它可能再次被访问
- **空间局部性 spatial locality**：如果某个数据项被访问，与它地址相邻的数据项可能很快也将被访问

存储器层次结构

层次结构介绍

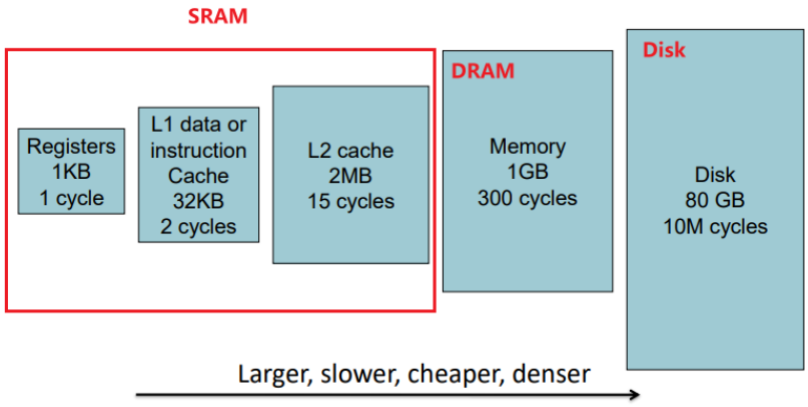
我们可以利用局部性原理将计算机存储器组织成为**存储器层次结构 memory hierarchy**，存储器层次结构由不同速度和容量的多级存储器构成，快速存储器每 bit 的成本比慢速存储器高很多，通常它们的容量也比较小

下图给出了一个存储器层次的基本结构



- 较快的存储器靠近处理器
- 较慢的存储器层次较低，更加便宜，其目的是以最低的价格向用户提供尽可能大的存储容量，同时**存取速度与最快的存储器相当**

存储器层次结构如下：



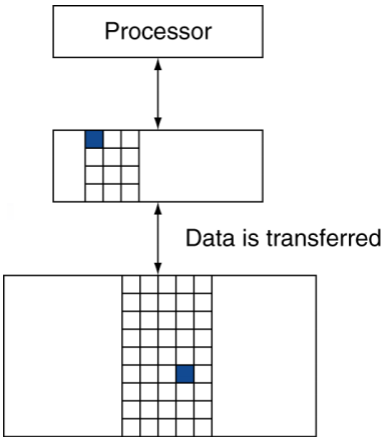
高层的存储器靠近处理器，比底层存储器容量小但是**访问速度更快**

存储器层次结构的思想

- 将所有内容存储在 disk 上
- 将最近访问（和附近）的项目从磁盘复制到较小的 **DRAM memory**
    - **这是主存 main memory**
- 将更频繁访问（和附近）的项目从 DRAM 复制到较小的 SRAM 存储器
    - **这是缓存 Cache memory**

存储器各层次间交互

存储器层次结构可以由多层构成，但数据每次只能在相邻的两个层次之间进行复制



我们将两个层次结构中存储信息交换的最小单元称为**块 block** 或**行 line**

较高层的初始状态是空的

如果访问的数据不在较高层

- Miss：block 从较低的层复制过来
    - **缺失代价 Miss penalty**：从低层复制上来所需要的时间
      将相应的块从底层存储器替换到高层存储器中，以及将该信息块传送给处理器的时间之和
    - **缺失率 Miss ratio**：$\frac{Miss的次数}{总的访问次数}$

如果访问的数据在较高层

- Hit：上层可满足的访问
    - **命中时间 hit time**：指访问存储器层次结构中高层存储器所需要的时间，包括了判断的当前访问是命中还是缺失所需的时间
    - **命中率 Hit ratio**：$\frac{His的次数}{总的访问次数} = 1 - miss\ ratio$

例题1 存储器层次结构带来的好处

假设访问数据需要的平均访问时间是 300 cycles，如果使用一个 hit rate 为 95% 的 cache，它的访问时钟周期为 1 cycle，那么平均访问时间是多少？

$$0.95 \times 1 + 0.05 \times (300 + 1) = 16(cycles)$$

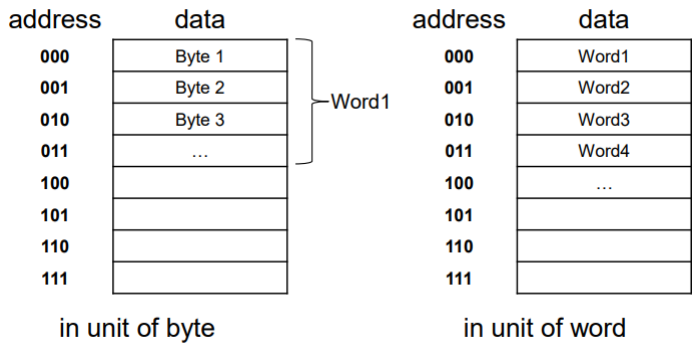
数据层次结构

数据也可以组织成层次化结构

- 靠近处理器那一层中的数据是那些较远层次中的子集
- 所有数据都被存储在最慢的底层
- 离处理器越远的层次访问时间越长

存储器内部结构

存储器中的 address 是一种形式上索引，不会存储在存储器内，一个 address 对应存储的 data 可以是以 byte 为单位的，也可以是以 word 为单位的（通常计算机是以 byte 为单位）

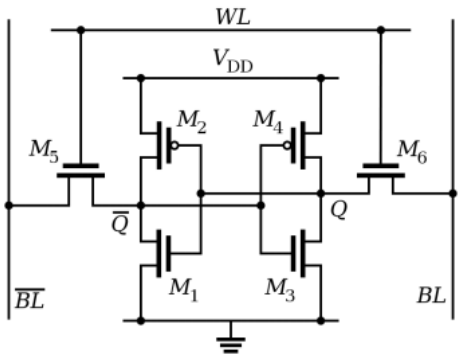


2. 存储器技术

目前，构建存储器层次结构主要有 4 中技术，它们的存储器层次结构自上而下如下表所示：

存储器技术	访问时间(MS)	用途	性质
SRAM 静态随机存取存储器	0.5~2.5	最靠近处理器的那层 cache	易失存储器
DRAM 动态随机存取存储器	50~70	每 bit 成本低于 SRAM 速度更慢，容量更大	易失存储器
Flash 闪存	5000~50000	用作个人移动设备中的二级存储器	非易失存储器
磁盘	5000000~20000000	服务器中容量最大且速度最慢的一层	非易失存储器

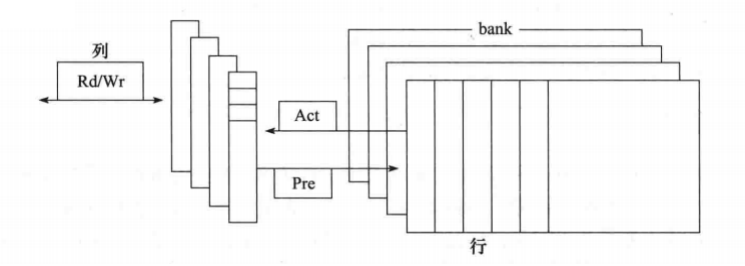
SRAM



- SRAM 是一种组织成存储阵列结构的简单集成电路，通常具有一个读写端口
- 虽然读写访问时间可能不同，但 SRAM 对任何**数据访问时间都是固定的**

- **SRAM 不需要刷新**，并且其访问时间与周期时间非常相近。为了防止读操作时信息丢失，SRAM 的一个基本存储单元通常由 **6-8 个晶体管**组成
- 在空闲模式下，SRAM 只需要最小的功率来保持电荷
- 过去，在大多数 PC 和服务器系统中通常将 SRAM 芯片从它们的一级、二级,甚至三级 cache 中分离出来。由于摩尔定律的推动，当今的处理器芯片中集成了多层次的cache，因此独立的 SRAM 芯片几乎在市场上消失了
- SRAM 只要加电，其中的数值就会保持
- 用于CPU cache，集成在处理器芯片上

DRAM



- 在 DRAM 中，存储单元使用电容保存电荷的方式来存储数据，为了对保存的电荷进行读取或写入，使用一个晶体管对该电容进行访问
- 由于 DRAM 存储的每一个 bit 都只使用**单晶体管**，所以它比 SRAM 的密度要高很多，价格也更便宜
- 由于 DRAM 在电容上保存电荷，不能长久的保持数据，**从而必须周期性地刷新**
- DRAM 单元中的电荷可以保持几微秒，如果 DRAM 中的每个 bit 需要独立的读出后写回，则必须不停地进行刷新操作
- Burst Mode：为了防止没有时间可用于正常的访问操作， DRAM 采用一种两级译码结构，一行的数据可以在一次刷新中被全部 Act 到上级
  （可以通过在一个读周期后紧跟一个写周期的方式一次**刷新一整行**一个单元共用一个字线）

SDRAM

- 为了进一步优化与处理器的接口，DRAM 增加了时钟，因此称之为 Synchronous DRAM，简写为 SDRAM
- SDRAM 的优势在于使用时钟对存储器和处理器保持同步，其速度上的优势主要源于不需要额外指定地址位以突发方式传送多个数据的能力，而是在时钟的控制之下以 Burst Mode 传送连续的数据
- 允许在时钟上升沿使用 Burst Mode 读取一整行的数据
- 提高带宽
- 最快的版本称为**双数据速率（DDR）SDRAM**，表示在时钟的上升沿和下降沿都使用 Burse Mode 传送数据，因此获得双倍的数据带宽
- 目前最新的版本为 DDR4，每秒可以传输 3200 兆次，时钟频率为 1600MHz

闪存 Flash

- 非易失性半导体存储器
- 闪存是一种电可擦除的可编程只读存储器 EEPROM
- 它不适合用于存储器，大多用于硬盘，因为它是有限次数可写的，不能频繁写
- 与磁盘和 DRAM 不同，而与其他 EEPROM 技术类似，**对闪存的写操作可以使存储位损耗**，为了应对该限制，大多数闪存产品有一个控制器，用来将写操作从已经写入很多次的块中映射到写入次数较少的块中，使写操作尽量分散，这种技术称为**损耗均衡 wear leveling**
- 这种均衡技术会降低缓存的潜在性能，但是不需要在高层次软件中监损耗，损耗均衡也将制造过程中出错的存储单元屏蔽掉，提高成品率

磁盘存储器 Disk

一个硬质磁盘包含

- 非易失性，旋转磁存储器
- 一组圆形磁盘片（绕着轴心每分钟旋转 5400~15000周）

  两侧均被磁性存储材料覆盖，其磁性材料与盒式磁带和录像带材料相同
- 读写磁头，对硬盘上的信息进行读写
- 磁道，每个磁盘的表面划分为同心圆盘，每个面通常由几万条磁道 track，每条磁道同样被划分为存储信息的扇区 sector，每条磁道有几千个扇区，每个扇区的容量为 512~4096 字节

每一个扇区包含了

- 扇区号 Sector ID
- 数据
- 纠错码 Error correcting code（ECC）用于隐藏缺陷和记录错误

访问每个盘面的磁头连在一起相互协调运动，因此每个盘面的磁头位于相同的扇区，为了访问数据，操作系统必须对磁盘进行三步操作

1. **排队延时 Queuing delay**：如果正在进行其他访问，则等待
2. **寻道时间 seek**：将磁头移动到磁道之上所花费的时间
3. **旋转延时 rotational latency**：当磁头到达了正确的磁道，就必须等待要访问的扇区转动到读写头下面，平均延时为磁盘转动一周时间的一半

   每分钟 5400 周的磁盘的平均旋转延时为
   $\frac{0.5}{5400 Rotation\ Per\ Minute} = \frac{0.5}{\frac{5400}{60}} = 0.0056s = 5.6ms$

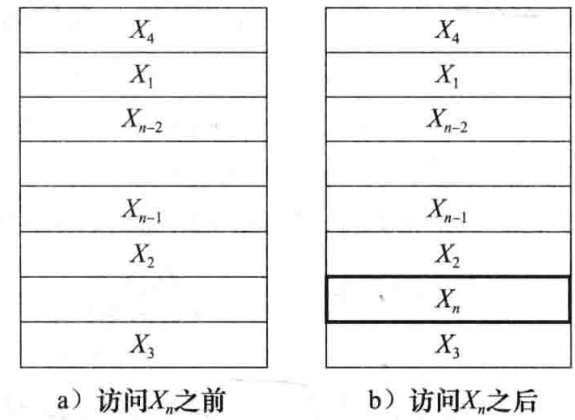
4. **传输时间 data transfer**：传输一块数据需要的时间，传输时间是扇区大小，旋转速度和磁道信息密度的一个函数，2012年，传输速率在每秒 100~200MB
5. **控制器开销 Controller overhead**

3. 缓存 cache

一个简单的 cache

下图为一个简单的 cache，处理器每次请求一个 word，每个 block 由一个单独的 word 构成

要访问的数据项最初不在 cache 中，在请求发出之前，cache 中保存了所有最近访问的数据项 $X_1, X_2, \ldots, X_{n-1}$ 的集合，而当前处理器所要数据项 X_n 不在 cache 中，该请求导致了一次 miss，X_n 被从主存调入 cache 之中



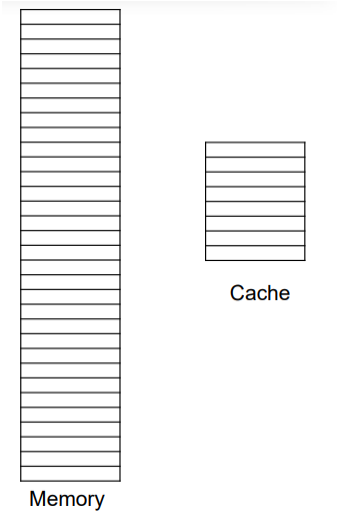
我们需要解决两个问题

- 如何知道一个数据项是否在 cache 中
- 如果在 cache 中，如何找到它

直接映射 direct mapped

索引 index

假设我们的 memory 大小是 32 个 word，cache 的大小是 8 个 word，且 address 是以 word 为单位的



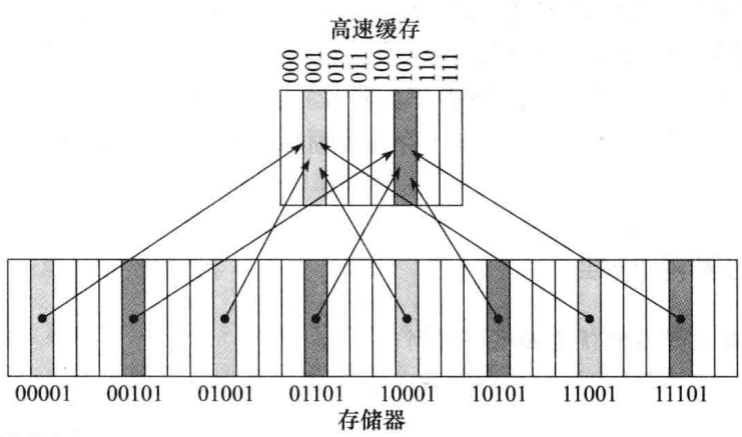
在 cache 中为 memory 每个 word 分配一个位置的最简单的方法就是根据这个 word 的 main memory 地址进行分配

每个存储器地址对应到 cache 中一个确定的地址，对于直接映射来说，主存地址和 cache 之间的典型映射通常很简单

$$(block\ address)\ \ mod\ \ (cache\ 中的\ block\ 数量)$$

如果 cache 中的块数是 2 的幂，那么取模计算很简单，只要去取地址的低 $log_2(cache\ 中的\ block)$ 个 bit 即可

一个有 8 个块的 cache 可以使用块地址中最低的 3 bit（$8 = 2^3$），假设存储器地址从 00001_2 到 11101_2，那么它们被映射到 cache 中的 001_2 到 101_2 中的位置



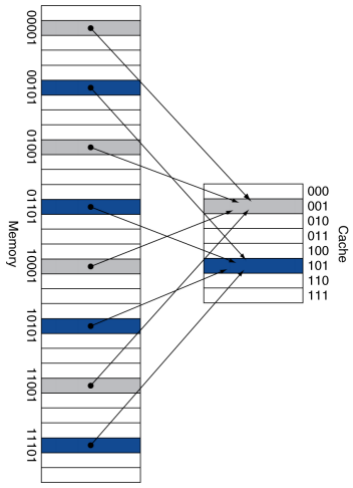
我们把 cache 的地址又称为**索引 index**

标记 tag

由于 cache 中的每个位置对应 memory 中的多个地址，为了确定所请求的 word 是否在 cache 中，我们可以在 cache 中增加一组**标记 tag**

tag 中包含了 memory 的地址信息，这些 地址信息可以用来判断 cache 中的 word 是否就是所请求的 word

tag 只需要包含 memory 地址的高 bit 部分，不需要那些用来检索 cache 的 index



如上图的示例所示，如果我们把 memory 中地址为 00001 的数据放入 cache 中，它放入 cache 中的 index 为 001（低 3 bit），它的 tag 为 00（高 2 bit）

有效位 valid bit

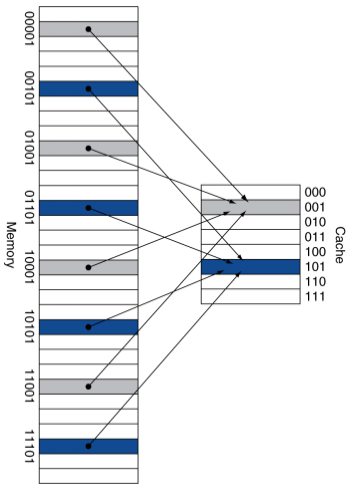
我们还需要一种方法来判断 cache 中的 block 是否包含有效信息，例如，当刚启动时，cache 中没有数据，tag 中的值没有意义，这些 block 的应该被忽略

最常用的方法就是增加一个有效位 valid bit，来表示一个 block 中是否含有一个valid address，如果不是有效的，则不能使用该 block 中的内容

cache 访问

cache 的访问示例1

- cache 有 8 个 block，memory 大小为 32 个 block
- 每个 block = 1 个 byte



memory 的地址的位数为

$$log_2(32个block \times 1个byte) = 5(bit)$$

cache 的 index 的位数为

$$log_2(8个block \times 1个byte) = 3(bit)$$

cache 的初始状态如下

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

下面是对该 cache 进行 9 次访问的一个序列

访问的十进制地址	访问的二进制地址	在 cache 中命中/缺失	分配的 cache 块（查找或者放置的位置）
22	10110_2	缺失（图 5-9b）	$(10110_2 \bmod 8) = 110_2$
26	11010_2	缺失（图 5-9c）	$(11010_2 \bmod 8) = 010_2$
22	10110_2	命中	$(10110_2 \bmod 8) = 110_2$
26	11010_2	命中	$(11010_2 \bmod 8) = 010_2$
16	10000_2	缺失（图 5-9d）	$(10000_2 \bmod 8) = 000_2$
3	00011_2	缺失（图 5-9e）	$(00011_2 \bmod 8) = 011_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$
18	10010_2	缺失（图 5-9f）	$(10010_2 \bmod 8) = 010_2$
16	10000_2	命中	$(10000_2 \bmod 8) = 000_2$

索引	有效位（V）	标记	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a）上电后 cache 的初始状态

索引	有效位（V）	标记	数据
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	**10_2**	**主存（10110_2）**
111	N		

b）处理地址（10110_2）缺失后的 cache 状态

索引	有效位（V）	标记	数据
000	N		
001	N		
010	Y	**11_2**	**主存（11010_2）**
011	N		
100	N		
101	N		
110	Y	10_2	主存（10110_2）
111	N		

c）处理地址（11010_2）缺失后的 cache 状态

索引	有效位（V）	标记	数据
000	Y	**10_2**	**主存（10000_2）**
001	N		
010	Y	11_2	主存（11010_2）
011	N		
100	N		
101	N		
110	Y	10_2	主存（10110_2）
111	N		

d）处理地址（10000_2）缺失后的 cache 状态

索引	有效位（V）	标记	数据
000	Y	10_2	主存（10000_2）
001	N		
010	Y	11_2	主存（11010_2）
011	Y	**00_2**	**主存（00011_2）**
100	N		
101	N		
110	Y	10_2	主存（10110_2）
111	N		

e）处理地址（00011_2）缺失后的 cache 状态

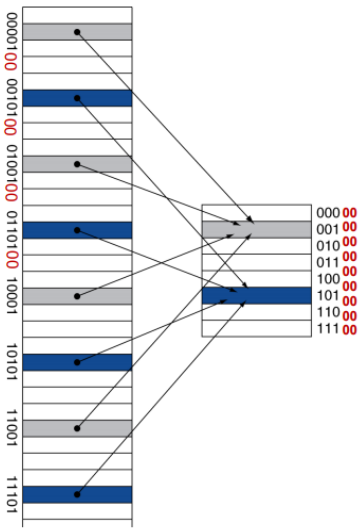
索引	有效位（V）	标记	数据
000	Y	10_2	主存（10000_2）
001	N		
010	Y	**10_2**	**主存（10010_2）**
011	Y	00_2	主存（00011_2）
100	N		
101	N		
110	Y	10_2	主存（10110_2）
111	N		

f）处理地址（10010_2）缺失后的 cache 状态

由于 index 用来寻址，且一个 n bit 的 tag 有 2^n 种值，直接映射到 cache 中项的总数必须为 2 的幂

cache 的访问示例2

- cache 有 8 个 block，memory 有 32 个 block
- 每个 block = 1 个 word = 4 个byte



memory 的地址的位数为

$$log_2(32个block \times 4个byte) = 7(bit)$$

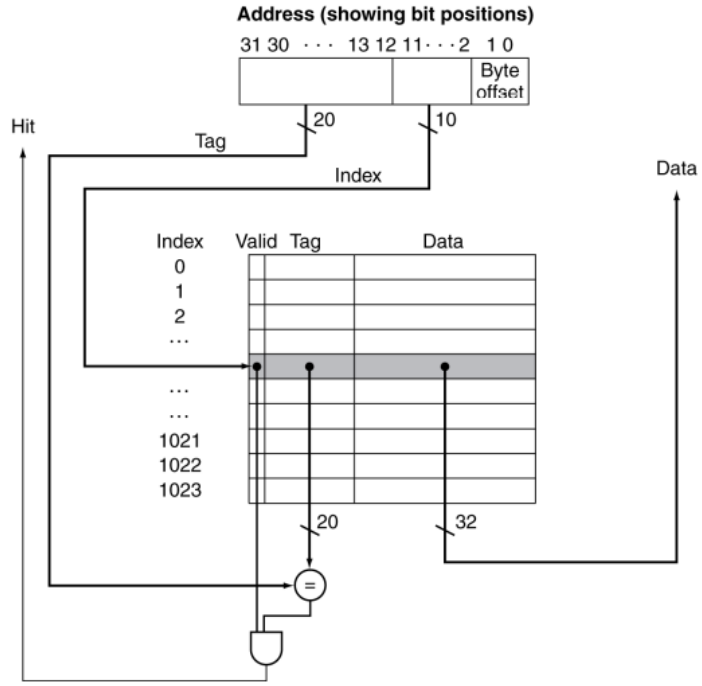
cache 的 index 的位数为

$$log_2(8个block \times 4个byte) = 5(bit)$$

我们的 memory 和 cache 将都在后面补充两个 0

由于我们的搬移是以 block 为单位的，也就是说，其实 memory 和 cache 后面的 2 bit 是不用考虑的，总之它都会搬移到 cache 中，只需要关心是第几个 block，所以每个地址的低 2 bit 可以被忽略，memory 和 cache 中的低 2 bit（新增的）被作为偏移量 offset 不做考虑，其实与之前的没有什么区别

我们的寻址模式如下所示：

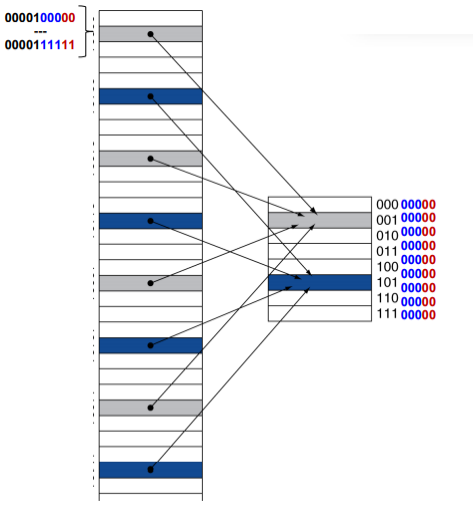


由于这里的 cache 的大小为 1023 个 block = 2^{10}，在 memory 的地址中

- 去掉低 2bit
- 将地址的 address[11:2] 作为 cache 中的 index 进行寻找
- 将地址的 address[12:31] 作为 cache 中的 tag 进行匹配

大容量的 **block** 的 **cache** 访问示例

- cache 有 8 个 block，memory 有 32 个 block
- 每个 block = 8 个 word = 32 个 byte



memory 的地址的位数为

$$log_2(32个block \times 32个byte) = 10(bit)$$

cache 的 index 的位数为

$$log_2(8个block \times 32个byte) = 8(bit)$$

由于我们的搬移是以 block 为单位的，也就是说，其实 memory 和 cache 后面的 5 bit 是不用考虑的，总之它都会搬移到 cache 中，只需要关心是第几个 block

偏移量 Offset

总之因为只需要考虑 block 的数量，memory 和 cache 后面的地址我都不需要考虑

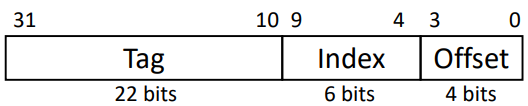
例如

- cache 为 64 个 block
- 每个 block = 4 个 word = 16 个 byte

所以我们有 $log_2 16 = 4(bit)$ 的低 bit 部分不用考虑，我们把它们通称为 offset

- offset：$log_2 16 = 4(bit)$
- index：$log_2 64 = 6(bit)$
- tag：$32 - 6 - 4 = 22(bit)$（memory 的地址是 32-bit）

整体效果如下



所以我就可以具体算出 cache 中具体是哪一个 block

计算 cache 的大小

假设 address 是 32-bit 的，使用直接映射

假设有 2^n 个 block，所以有 index = n (bit)

假设一个 block 存放 2^m 个 word，即一个 block = 2^{m+2} 个 byte

所以我们可以看到

- offset：m+2
- index：n
- tag：32-n-(m+2)

如果要计算 cache 的大小，我们知道，cache 中的一个 index 存储的信息有

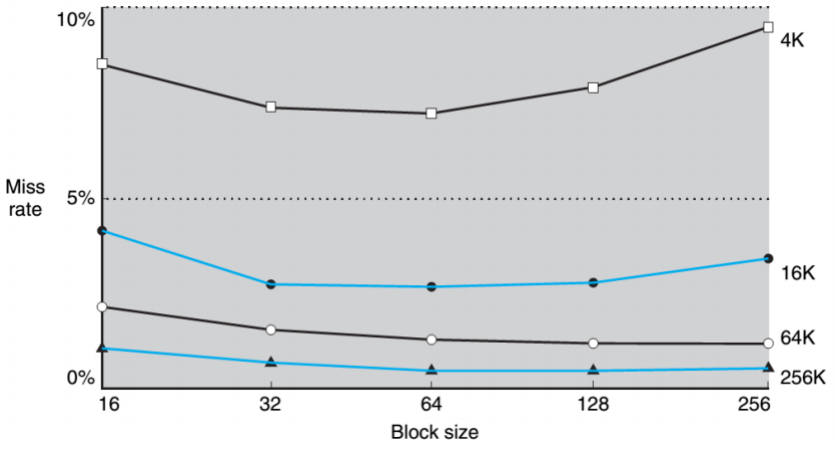
- block：$2^m \times 32$（2^m 个 word，每个 word = 32 bit）
- tag：32-n-(m+2)
- valid bit：1

故整个 cache 的大小为（以 bit 记）

$$2^n(个block) \times ([2^m \times 32] + [32 - n - (m + 2) + 1] + [1])$$

缺失率与块大小的协调

较大的 cache 块能够更好地利用空间局部性以降低缺失率，而当块大小在 cache 容量中所占比例增加到一定程度时，缺失率也随之增加



cache 缺失

- 如果 cache 命中，计算机继续使用该数据
- 如果 cache 缺失，控制单元必须能够检测到缺失的发生，然后从主存（或者较低一级的 cache）中取回所需的数据来处理缺失

cache 缺失处理由两部分共同完成

- 处理器控制单元
- 进行初始化主存访问和重新填充 cache 的独立控制器

当 cache 缺失，我们等待主存操作完成时，整个处理器阻塞，临时寄存器和程序员可间寄存器中的内容被冻结（我们假定均为顺序执行处理器，当 cache 缺失时被阻塞）

Read miss

如果指令访问引起一次缺失，那么指令寄存器中的内容无效，为了将正确的指令取回 cache，我们必须通知存储器层次结构中的较低层次执行一次读操作，由于在执行的第一个时钟周期，程序计数器 PC 产生了新的增量，因此产生缺失的指令地址等于程序计数器的值 -4，当地址产生时，就可以通知主存进行一次读操作，并等待存储器相应（可能需要多个时钟周期访问主存），然后把取回的 word 写入 cache

1. 把程序计数器 PC 的原始值（PC - 4）送到存储器中
2. 通知主存执行一次读操作，并等待主存访问完成
3. 写 cache 项，将从主存取回的数据写入 cache 中存放数据的部分，并将地址的高 bit 部分写入 tag 域，设置有效位 valid bit
4. 重启指令执行第一步，重新取指，这次指令在 cache 中

写操作

如果有一个 store 指令，我们只将该数据写入数据 cache 的话，可能会导致主存与 cache 相应位置中的值，这种情况下，cache 和主存被认为不一致 inconsistent

下面的部分是当写回的数据在 cache 里时

写直达 **write-through**

- 保持主存和 cache 一致性最简单的方法就是将这个数据同时写入到主存和 cache 中
- 这种方法无法提供良好的性能，这些写操作将花费大量的时间

写缓冲 write buffer

- 当一个数据等待被写入主存时，先把它放入写缓冲中
- 当把数据写入 cache 和写缓冲后，处理器可以继续执行
- 当写主存操作完成后，写缓冲的数据项可以得到释放
- 如果写缓冲已经满了，那么处理器执行到一个写操作时就必须停下来直到写缓冲中有一个空位置

写回 write back

- 当发生写操作时，新值仅仅被写入 cache 块中
- 只有当修改过的块被替换时才需要写到较低层的存储结构中
- 有时候下写回机制也会用到写缓冲，当发生修改时，被修改的数据块移入缓冲器

如果发生了 write miss，即写回去 memory 的内容不在 cache 里面

写分配 write allocate

- 一种机制是，先从 memory 中把里面的内容写入 cache 中后，再按照写回、写直达、写缓冲的机制来实现

写不分配 write around

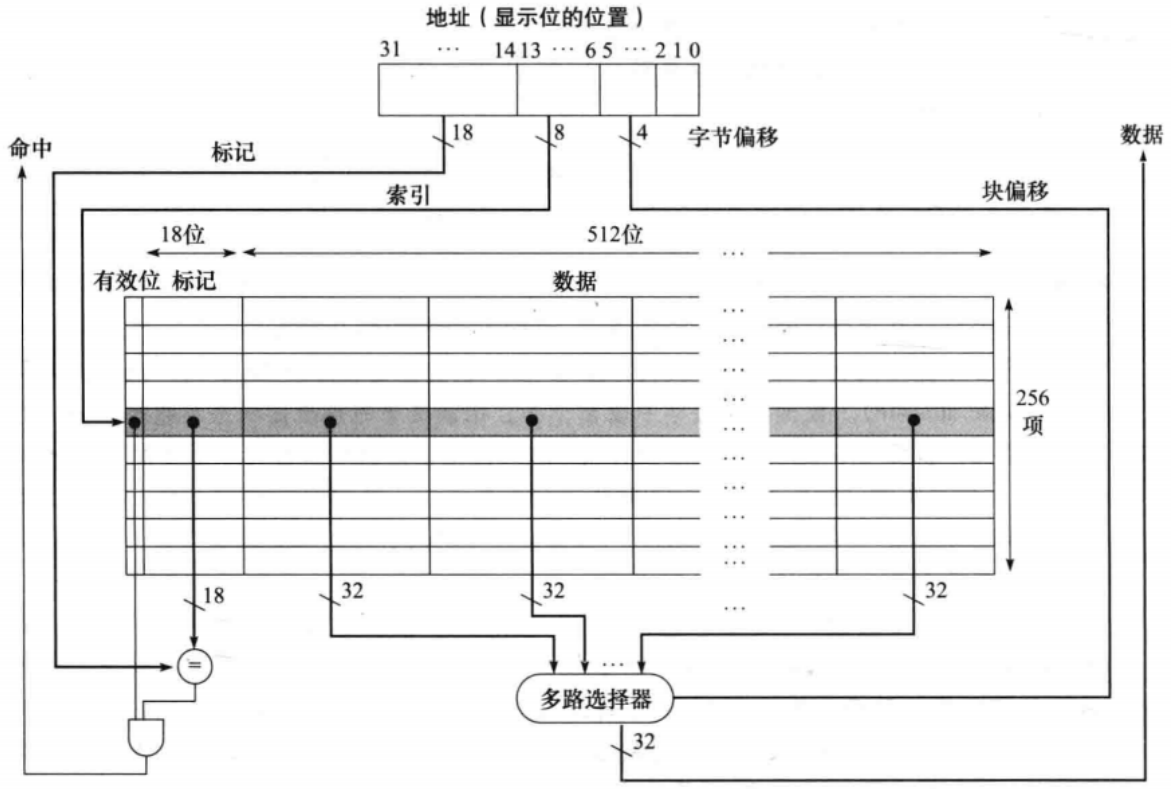
- 直接写入 memory，不用再放回 cache 里了
- 通常用于写和度分离的比较开的时候

写操作的流程

- 如果写 memory 的内容在 cache 中
    - 把它发送到 cache
    - 我们应该马上把它存入 memory？（**write-through**）
    - 等我们把 block 移出去？（**write-back**）
- 如果写 memory 的内容不在 cache 中
    - 把它放在 cache 中？（**write allocate**）
    - 直接写到 memory 里，不会放入 cache 中？（**no write allocate**）

FastMATH 处理器

内置 FastMATH 处理器是一个快速的嵌入式微处理器，它采用 MIPS 结构



cache 中有 256 块，每块 16 个 word

- offset = $log_2(16 \times 4) = 6(bit)$
- index = $log_2 256 = 8(bit)$
- tag = $32 - 6 - 8 = 18(bit)$

该处理器采用 12 级流水线结构，处理器每个时钟周期可以请求 1 个指令 word 和一个数据 word，为了不满足阻塞流水线的需求，使用了分离的指令 cache 和数据 cache，每个 cache 容量为 16KiB，即 4096 个 word

读请求

1. 将地址送到适当的 cache 中去，该地址来自程序计数器（对于指令访问），或者来自于 ALU（对于数据访问）
2. 如果 cache 发出命中信号，请求的 word 就出现在数据线上，由于在请求的数据块中有 16 个 word ，因此需要选择那个正确的字，tag 用来控制多路选择器，从检索到的 block 中选择 16 个字中的某个 word
3. 如果 cache 发出缺失信号，我们把地址送到 memory中，当主存返回数据时，把它写入 cache 后再读出以满足请求

写操作

- FastMATH 处理器同时提供写直达和写回机制，由操作系统决定某种应用应该使用哪种机制

缺失率

FastMATH 处理器执行 SPEC CPU 2000 测试程序时指令和数据的近似缺失率

它分离了指令 cache 和数据 cache

指令缺失率	数据缺失率	综合缺失率
0.4%	11.4%	3.2%

通常，混合 cache 具有较高的命中率，但是很多处理器使用分离的 cache 提高 cache 的带宽（近似加倍）