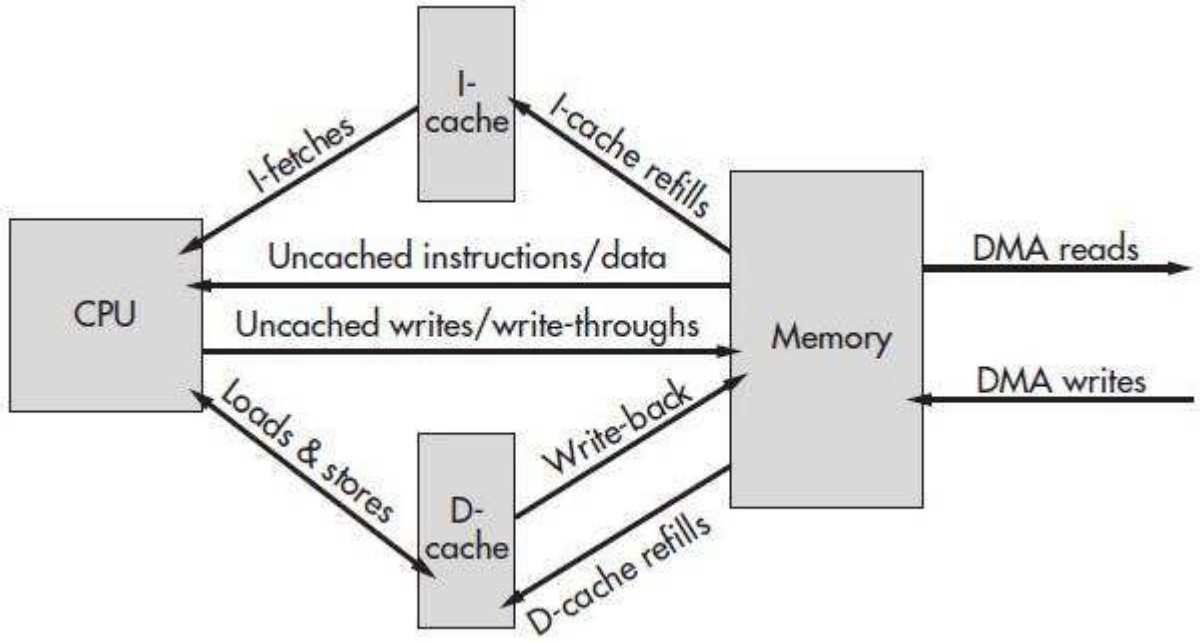
Cache 研究是轉正答辯「MIPS BSP 研究」中重要的一部分。只可惜當時時間緊，沒有能夠總結成文檔。時隔將近一年，這次編寫《CPU 體系架構系列》，對於這一部分內容既是總結整理，又是溫故知新。

概述

Cache 是用來對內存數據的緩存。CPU 要訪問的數據在 Cache 中有緩存，稱為「命中」(Hit)，反之則稱為「缺失」 (Miss)。CPU 訪問它的速度介於寄存器與內存之間（數量級的差別）。實現 Cache 的花費介於寄存器與內存之間。

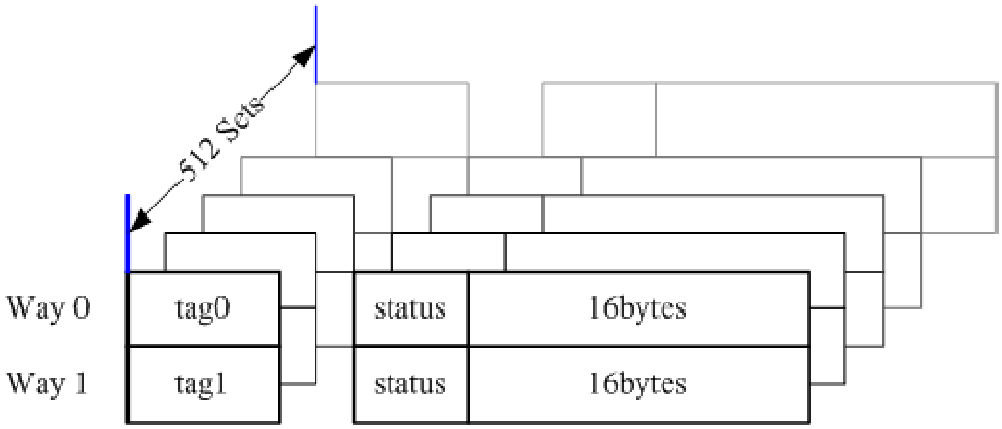
現在 CPU 的 Cache 又被細分了幾層，常見的有 L1 Cache, L2 Cache, L3 Cache，其讀寫延遲依次增加，實現的成本依次降低。現代系統採用從 Register –> L1 Cache –> L2 Cache –> L3 Cache –> Memory –> Mass storage 的層次結構，是為解決性能與價格矛盾改採用的折中設計。下圖描述的就是 CPU、Cache、內存、以及 DMA 之間的關係。程序的指令部分和數據部分一般分別存放在兩片不同的 cache 中，對應指令緩存（I-Cache）和數據緩存（D-Cache）。



引入 Cache 的理論基礎是程序局部性原理，包括時間局部性和空間局部性。即最近被 CPU 訪問的數據，短期內 CPU 還要訪問（時間）；被 CPU 訪問的數據附近的數據，CPU 短期內還要訪問（空間）。因此如果將剛剛訪問過的數據緩存在 Cache 中，那下次訪問時，可以直接從 Cache 中取，其速度可以得到數量級的提高。

Cache 結構

根據 Cache 相聯方式的不同，Cache 有 3 類：直接相聯，全相聯，組相聯。在這裡我們只介紹組相連的 Cache 結構。因為組相聯 Cache 則是直接相聯與全相聯的一個折衷，兼顧性能與價格，也是最常見，最普遍的 Cache 結構。首先給出組相連 Cache 的結構圖。以 harrier 平台的 CPU 芯片 BCM5836 的 Cache 為例。



對照上面的 Cache 結構圖，下面說明如何來說明 Cache 的指標。

1. Cache 的大小，也就是能夠存放多少字節。一般有 16KBytes，32KBytes 等。例如上面的這個 Cache 的大小就是 16KBytes。後面會說明這 16KB 是如何組成的。
2. Cache 有多少路？英文中，「路」表示為 way。例如上面的 Cache 就是兩路相連，分別為 Way0、Way1。
3. Cache 有多少組？英文中，「組」表示為 set。例如上面的 Cache 一共擁有 512 組。
4. Cache 的行大小？英文中，「行」表示為 line。在上圖中，行沒有標示出來，行其實就是某一路的某一組，或者某一組的某一路。上面的雖然有點兒繞，其實就是看到的 tag0、status、16bytes 就是一行。行的大小就是 16bytes。Cache 的大小就是所有的這些行的大小。

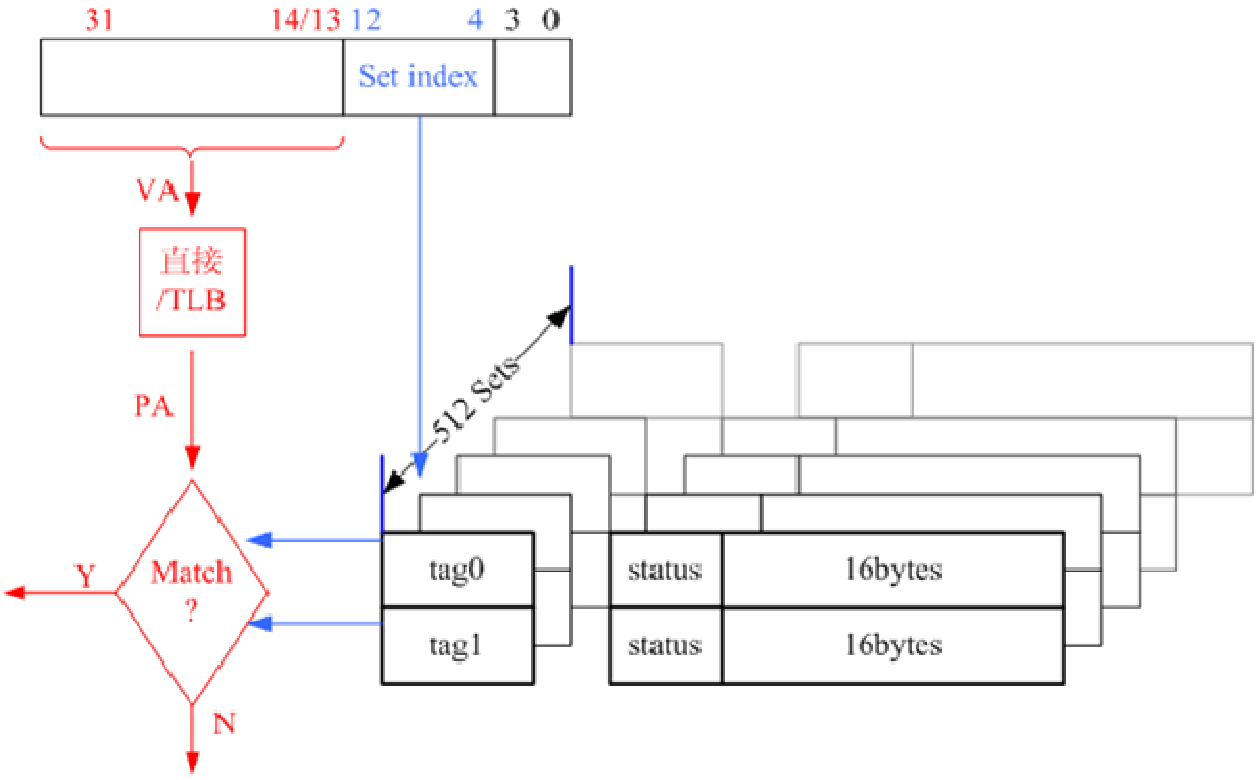
現在就知道 Cache 的大小是如何算得：16Bytes（行大小）*2（路）*512（組）=16KBytes。

例如，在我的個人 PC 上，使用軟件檢測獲得到 Cache 的數據如下圖所示。

工作方式

下面介紹組相連 Cache 的工作方式。

首先看的是 CPU 是如何獲取 Cache 中的數據的。 如下圖所示。



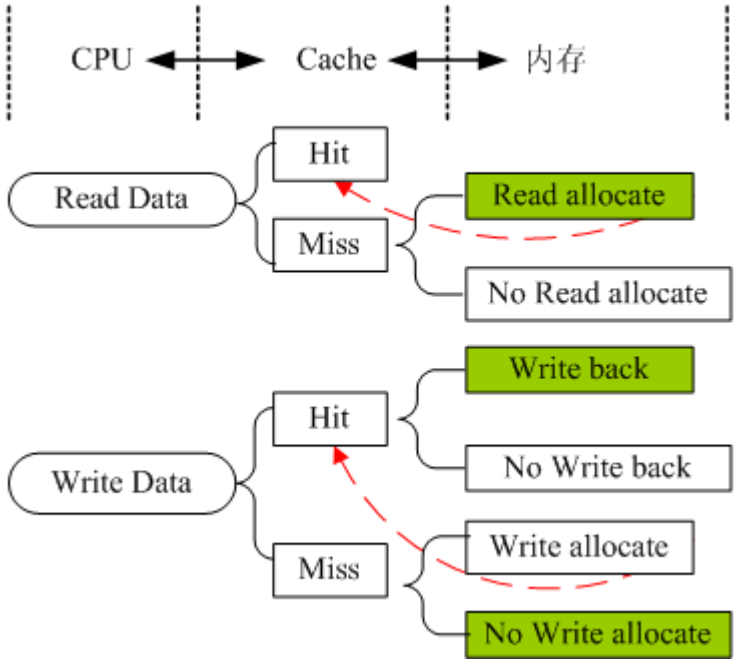
假設這裡是 16KBytes 的指令 Cache（I-Cache），那麼左上角 32 位的虛擬地址就是 PC 指針中的值(如果 PC 指針存在的話)。32 位的地址在圖中使用了 3 中顏色標記(黑色:bit0~bit3；藍色：bit4~bit12；紅色：bit13~bit31）。下面就是 CPU 從 Cache 中獲取指令的過程：

1. 匹配組。藍色的 9bit 選擇某一個組（一共 512 個組，9bit 表示完）；
2. 匹配路。匹配是上一步選定的組中的哪一路。在這裡，有兩種選擇路的方式：index 類型和 hit 類型。
3. 匹配 tag 頭。將紅色部分的虛擬地址（VA）轉化成物理地址（PA），轉化的物理地址和 tag 頭中的匹配,那麼就認為該地址處的值在 Cache 中存在,也就命中(hit)Cache。如果不匹配，那麼就認為該指令在 Cache 中不存在，未命中（miss），此時就需要到內存中取指令。
4. 匹配指令位置。通過上面三步，已經找到 Cache 中的某一行了。但是一行中有 16 個字節，應該取那一個（或者連續幾個）呢？黑色的 4bit 來確定。

CPU 和數據 Cache（D-Cache）的交互基本上也是這樣的。存在差別的地方時第三步，匹配 tag 頭。對於指令 Cache，不允許修改的(絕大部分情況)，所以匹配到 tag 頭，就認為 Cache 命中了，可以把 Cache 中的指令讀取到 CPU 執行了。但是數據 Cache 可能被修改了（corrupt），內存中的和 Cache 中的不相同了，這個時候，還需要查看數據 Cache 是否有效（valid）。

還需要強調的一點是，上面的過程完全由硬件完成，程序員不能夠操控其中的任何環節。

然後要看的是 Cache 和內存如何交互數據。 如下圖所示。



在 CPU 如何獲取 Cache 中的數據的問題中，還遺留有三個問題：第一，CPU 讀取 Cache 中的數據，沒有命中（miss）應該怎麼辦？第二，對於數據 Cache，CPU 修改了某個數據的值，並把它寫回到 Cache 中，造成 Cache 中的數據和內存中的數據不一致怎麼辦？第三，對於數據 Cache，CPU 修改了某個數據的值，並把它寫回到 Cache 中，結果沒有命中（miss）應該怎麼辦？這三個問題，就是這一段 Cache 和內存如何交互需要解決的問題。

首先看圖中 Read Data->Miss 的過程，CPU 讀取 Cache 中的數據，沒有命中應該怎麼辦？

讀數據時 Cache miss，實際實現中有 2 種策略：Read-allocate 和 No read-allocate(Read through)。現代的實現一般皆為 Read-allocate，即：先從 Cache 中分配一行，後從 RAM 中讀數據填充之，爾後將數據傳給 CPU。No read-allocate 則是直接從 RAM 取數據到 CPU（不經 Cache）。若非特別指出，讀策略皆默認為 Read-allocate。

然後看圖中 Write Data->Hit 的過程，CPU 修改數據 Cache，命中，可能造成數據 Cache 和內存中的值不一致應該怎麼辦？

在寫命中(store hit)時，Cache 的實現亦有兩種策略：Write-back 和 Write-through。Write-through 的 Cache，在 write hit 時，會將數據更新到 Cache 和 RAM。Write-back 的 Cache，在 write hit 時，則僅將數據更新到 Cache 且將被更新的行標為'dirty'，當該行被替換時控制器才將該行數據寫回到內存。對於 Write-back 的 Cache，在連續多次寫數據時可以節約總線帶寬，性能要好於 Write- through，但由於其緩存的數據往往是最新的，與內存中的數據多數時候是不一致的，因此需要軟件來維護其一致性。

最後看圖中 Write Data->Miss 的過程，CPU 修改數據 Cache，未命中，這時應該如何處理呢？

寫數據時 Cache miss，實現的策略和讀數據時 Cache miss 類似，有兩種：Write-allocate 和 No write-allocate。前者的處理方式是先分配一行，後從 RAM 中讀數據填充之（相當於一個 read miss refill 過程），最後才將數據寫入 Cache（到此，亦會根據寫命中 store hit 進行後面的操作）。No read-allocate (Write-around)的處理方式則是繞過這一級 Cache（不分配 Cache line），直接將數據送到下一級 Cache/Memory。有些 MIPS 的實現兩種寫策略皆支持。

其實，上面的 Cahce 和內存交互是一個策略問題。一般都使用 Read-allocate，Write-back，No write-allocate 的策略，對於具體芯片實現，需要參考其 data sheet。

未深入的技術

虛地址 Cache：Cache 裡面的地址索引是使用物理地址索引，還是用虛地址索引？導致的問題有：錯誤共享問題和別名問題。

Cache 在 MIPS 下的重影問題。

Cache 實例（MIPS Cache）

對於 Cache，軟件能夠做什麼呢，程序員應該做什麼呢？本節就已 MIPS Cache 為實例來說明其中一些常見的操作。

在上文的 Cache 和內存的交互描述中，有這麼一段話： *對於 Write-back 的 Cache，在連續多次寫數據時可以節約總線帶寬，性能要好於 Write- through，但由於其緩存的數據往往是最新的，與內存中的數據多數時候是不一致的，因此需要軟件來維護其一致性。* 這就是程序員在編程時，需要對 Cache 進行操作的地方。考慮這麼一種情況：

對於某片內存區域，剛開始在內存和 Cache 中是一致的。現在程序運行，對這篇數據進行了修改，由於 Cache 是 Write-back 類型的，修改的數據都保存在 Cache 中，而內存中的數據仍然是舊的數據。此時，程序啟動 DMA，將內存中的數據傳送到外部設備。無疑，此時傳送出去的數據並不是我們想要的數據（我們想要傳送出去的是通過計算，仍然保存在 Cache 中的新數據），這就是問題出錯的地方。我們需要在完成數據計算之後，啟動 DMA 之前，將 Cache 中的數據寫回到（flush）到內存中。這就是所說的需要軟件來維護其一致性。

當然，上面是一種很常見的需要軟件參與的情況，還有很多其它類似的情況。下面我們首先來看看 MIPS 留給程序員的 Cache 接口是怎麼樣的。

MIPS Cache 軟件接口

寄存器 TagHi 和 TagLo 用於暫存 Cache 行的 Tag 中的數據，都是 32 位 CP0 寄存器。

Cache 指令 MIPS 體系結構只引入了 cache 指令作為軟件控制 cache 的統一接口。Cache 指令的格式如下所示：

```
cache ops, addr
```

其中 ops 在指令中佔據 5 位，低 2 位指定 Cache 的類型，高 3 位指定執行的操作。下表中列出了 ops 低 2 位指定的 Cache 的類型，一般我們使用到的只有一級 I-Cache 和 D-Cache。

ops 低 2 位 Cache 的類型

00	L1 I-Cache
01	L1 D-Cache
10	L2 Cache
11	L3 Cache

下表中列出了 ops 高 3 位指定該 Cache 命令執行的操作。

Ops 高 3 位 執行的操作

000	Index Invalidate
001	Index Load Tag
010	Index Store Tag
011	
100	Hit Invalidate
101	Fill/Hit Writeback Invalidate
110	Hit Writeback
111	Fetch and Lock

有關 MIPS Cache 的軟件接口，可以參考《See MIPS Run》一書的 Chapter 4-How Caches Work on MIPS Processors 4.9-Programming MIPS32/64 Caches，也可以參考《The MIPS Cache Architecture》一文的 Chapter 4-MIPS Cache 控制接口。

初始化 Cache

Cache 的初始化分兩步：第一步開啟 Cache 功能，設置寄存器 CP0 config 0、CP0 brcm config 0 等，使能 I-Cache 和 D-Cache 功能，開啟 Kseg0 的 Cache 功能。第二步初始化 Cache，將未知狀態的 Cache 行的 Tag 域置為 0。下面的代碼就是設置 Tag 頭為 0 的過程。

```
    ...
    /* Calc an address that will correspond to the last cache line  */
    addu    a3, a2, a0
    subu    a3, a1

    /* Loop through all lines, invalidating each of them */
1:
     cache  ICACHE_INDEX_STORE_TA 0(a2)     /* clear tag */
    bne     a2, a3, 1b
    addu    a2, a1
    ...
```

DMA 操作和 Cache

最後回答一下在本節開始舉的那個問題，Cache 和內存中的數據不一致，導致 DMA 需要軟件參與 Cache 的操作。代碼如下所示，如果是 DMA 要將內存中的數據傳送到外設（DMA_TX），那麼需要將 Cache 中的數據沖刷到內存（cacheFlush 函數），保證內存中的數據是最新的數據；如果 DMA 將外設的數據傳送到內存了，那麼需要將 Cache 中的數據置位無效（cacheInvalidate 函數），這樣下次 CPU 取該部分數據時，就不會到 cache 中獲取，而是獲取內存中的最新數據。

```
void*
osl_dma_map(void *dev, void *va, uint size, uint direction)
{
    if (direction == DMA_TX)
        cacheFlush(DATA_CACHE, va, size);
    else
        cacheInvalidate(DATA_CACHEva, size);
    return ((void*)CACHE_DMA_VIRT_TO_PHYS(va));
}
```

該段代碼選自網絡驅動模塊，DMA 將網絡數據包在內存和網卡之間傳送。