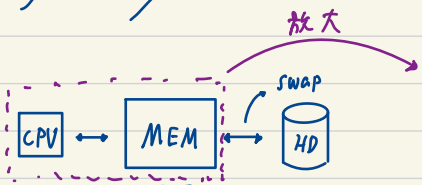
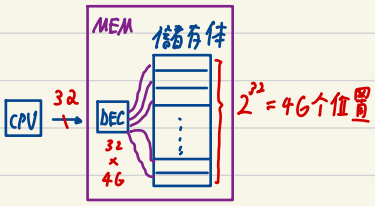
⑪ Memory hierarchy

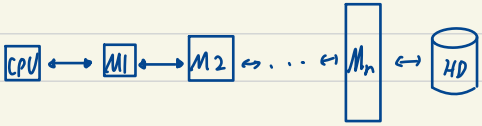




Note:
Decoder有几个輸出，就有几个 AND GATE
故以左圖為例，有4G个 AND GATE
∴ MEM size 越大，decoder 越大
解碼時間越長
CPU存取速度越慢

根據馮·紐曼原則
當要執行某个 executable file 時
需要先將其 swap in 到 MEM 中，再由 CPU 去抓指令來執行
但 swap 速度很慢（∵是 disk I/O operation）
而當 MEM 越小，swap 次數會增加
① 站在減少 swap 次數上，MEM 應越大越好.
② 站在減少 CPU存取時間上，MEM 應越小越好.

∴ 使用 MEM Hierarchy 方法：
靠近 CPU 的越小，靠近 HD 的越大



∴ 設 $M1, M2, \ldots, M_n$ 為資料構成之集合
則：$M1 \subseteq M_2 \subseteq M_3 \cdots \subseteq M_{n-1} \subseteq M_n$

CPU 只能存取 M1 的資料，若欲存取資料不在 M1 的話
需向 M2 一直向後找，找到再向前傳至 CPU
設程式執行時，每一指令，資料執行机率均等
∵ M1很小 ∴ miss rate 高 資料傳輸慢
但事實上，有 locality 的現象 每一指令，資料執行机率不均等
∴ 可把執行机率高的指令，資料放至 M1
降低 miss rate 至很低，則存取速度約等同 M1 速度

Principle of locality

Def: 程式執行時，任一時間只能存取一小部份的位址空間

1. Temporal Locality：
   程式中某些部份被存取後 很快又會被存取到

   e.g.
   ```
   for i=1 to n do {
       \* loop body *\
   }
   ```

2. Spatial Locality：
   程式中某些部份被存取後，它位址附近的部份
   很快又會被存取到

   e.g. Sequential Execution
   Array

Ex. 一段 C code 同時具 spatial locality & temporal locality

```
void clear( int A[], int n ){
    int i;
    for( i=0; i<n; i++ ){
        A[i]=0;
    }
}
```

A[i]=0 指令的抓取具 temporal locality
A[i] 資料的抓取具 spatial locality

MEM 建構技術

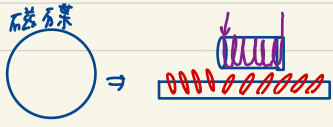
① SRAM：資料存在 Latch；存1 bit 需要6顆電晶体
速度較快，成本高

static：資料不隨時間而變

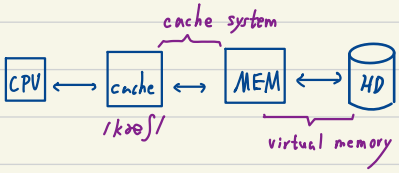
② DRAM：資料存在電容器中；存1 bit 只需一顆電晶体
會有寄生電容效應(充放電太久，∴有資料速度慢)

dynamic：∵電容器會漏電，需要在漏電量沒很多時 補回，稱作 refresh
如果不refresh，存的資料值會改變

③ Magnetic Disk：磁碟盤上有磁性物質
而讀寫頭會有線圈，可通電流產生感應磁場
使磁碟盤上磁性物質沿一方向排列
∴改變電流方向即可改變 0,1

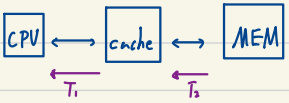


而讀取則是透過將磁碟上欲讀取資料移至讀寫頭正下方
磁場又能產生感應電流
判斷感應電流方向即可知是1或0
但磁化過程很慢，故速度慢



Terminology:

1. block / line：兩個記憶體間資料傳輸最基本單位。
2. hit / miss：若CPU可在上一層記憶體找到資料稱作 hit，反之為 miss
3. hit rate / miss rate 在上一層記憶體找到資料的比例為 hit rate
   反之為 miss rate，∴ hit rate + miss rate = 1
4. hit time：CPU存取上一層資料的需時間，包括找和傳回 CPU 時間
5. miss penalty：設CPU存取上一層資料時 miss 後，需往更上層找資料
   找到了傳回一個 block 予傳回 CPU 的時間



Miss Penalty = $T_1 + T_2$
但 ∵ $T_2 >> T_1$ ∴
計算時為 T_2

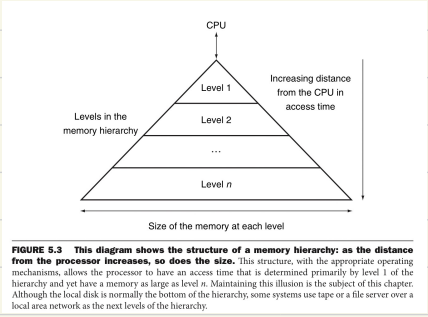
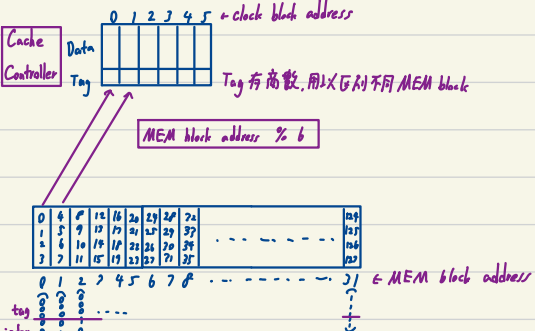


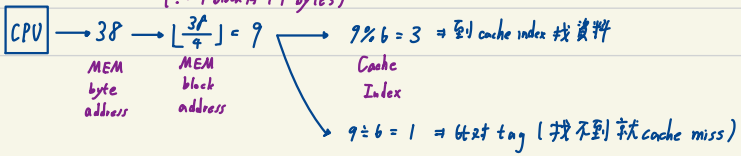
FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

Direct-Mapped Cache



CPU 到 cache 抓資料的位址轉換：

(∵一個block有4個bytes)



共需做 2 次除法 (除法速度慢)

① CPU = 01001|10 (offset)
MEM Block Address
↓
010|01
Tag | Index

⇒ 不需做除法, 只需做 bit 切割即可

Cache miss 處理.

e.g.
MEM Block Address | offset
CPU → 001|01|00
Tag | Index

設 cache miss, CPU會 stall
Cache Controller 會要求 MEM 把5号內容讀到 cache
讀取完成後, Controller 才會通知 CPU
之後, CPU 照 cache index, 到对应编号 offset 抓資料

Memory Address 切割方式:

① 看 MEM 共有几个 blocks 設為 n 个 blocks ⇒ $\lg n$ 个 bit
② 看 cache 共有几个 blocks, 設為 m 个 blocks ⇒ $\lg m$ 个 bit

Memory Address: [$\lg n - \lg m$ | $\lg m$ |]
($\lg n$)

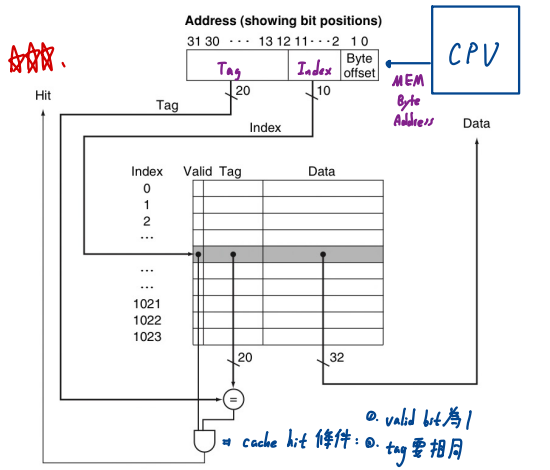


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KiB. We assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving 32 − 10 − 2 = 20 bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

其中, valid bit 用作区別 cache block 內資料是否有效
若為0, 表示不是有效資料
(避免 CPU 用到前面 program 資料)

計算題

給定: ① cache size : cache 可存資料量
② block size
③ address length

求: ① block 總數: $\frac{\text{cache size}}{\text{block size}}$

Tag | Index | offset

② 切位址求 tag bit 數: Address length − lg(block size) − lg(block 總數)

② (1 + tag bit + lg(block size)) × block 總數