CA5장 메모리 계층구조

5.1 서론

지역성의 원칙 : 프로그램은 어떤 특정 시간에는 주소공간 내의 비교적 작은 부분에만 접근한다는 것.

지역성 종류

시간적 지역성 : 어떤 데이터가 참조되면 곧바로 다시 참조될 가능성이 높다. Ex) 순환문

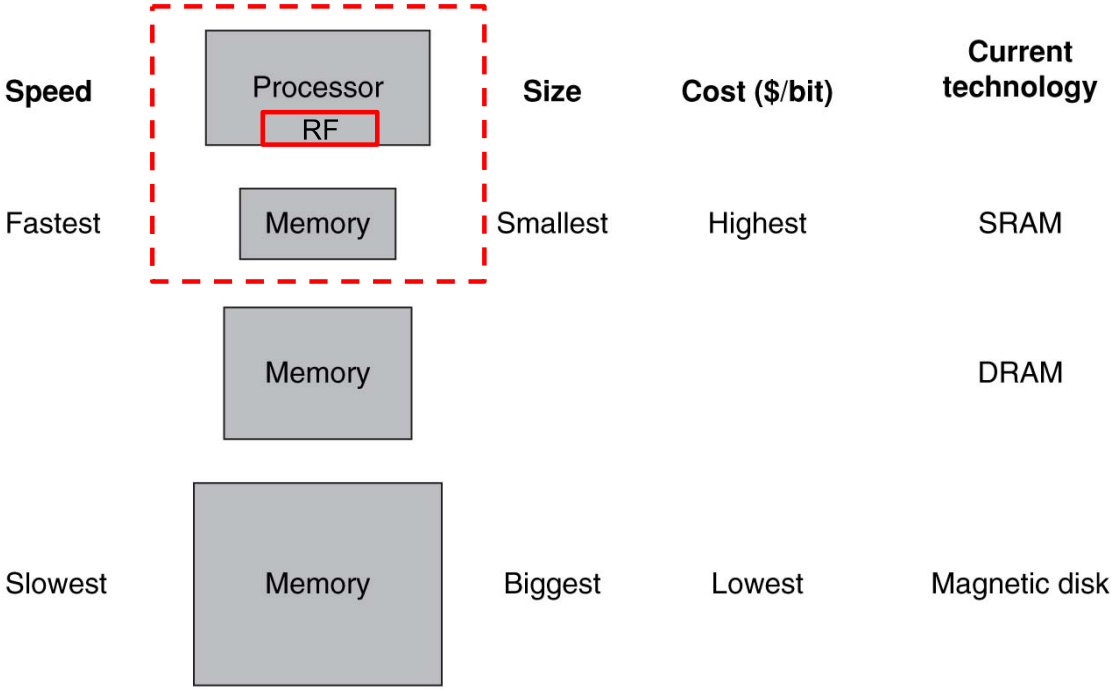
공간적 지역성 : 하나의 데이터가 참조되면 곧바로 그 주위의 데이터가 참조될 가능성이 높다.

Ex) 배열이나 레코드 요소들에 접근할 때

Memory Hierachy

정의 : 여러 계층의 메모리를 사용하는 구조. 프로세서로부터 거리가 멀어질수록 메모리의 크기와 접근 시간이 증가한다. 지역성 원칙 이용

가장 빠른 메모리는 더 느린 메모리보다 비트당 가격이 비싸서 대개 크기가 더 작다.



위의 그림은 메모리 계층구조의 기본 구성이다. 메모리 시스템을 계층적으로 구현하여 사용자는 큰 메모리를 사용하면서도 가장 빠른 메모리에 접근하는 것과 같은 환상을 갖게 해 준다.

데이터 계층 구조

프로세서에 가까운 꼐층은 먼 계층의 부분집합이고, 가장 낮은 계층에는 모든 데이터가 다 저장되어 있다. 프로세서와 멀수록 시간 많이 걸림.

데이터는 인접한 두 계층사이에서만 한번에 복사가 된다.

두 계층간 정보의 최소 단위를 block 또는 line이라고 한다.

프로세서가 요구한 데이터가 상위 계층의 어떤 블록에 있을 때 이를 hit이라고 부르고 상위 계층에서 찾을수 없다면 miss라고 부른다. Miss시 하위 계층 메모리에 접근하게 된다.

Hit ratio: hits/accesses

Hit time : 메모리 계층 구조의 상위 계층에 접근하는 시간, 접근이 적중인지 실패인지 판단하는 시간도 포함

Miss penalty : 하위 계층에서 해당 블록을 가져와서 상위 계층 블록과 교체하는 시간에다 그 블록을 프로세서에 보내는 데 걸리는 시간을 더한 값

모든 프로그램이 메모리 접근에 많은 시간을 쓰기 때문에 메모리 시스템은 성능을 결정하는 중요 요소이다.

요점정리

메모리 계층구조는 최근에 접근했던 데이터들을 프로세서 가까이 적재함으로써 시간적 지역성을 이용. 또한 메모리의 상위 계층으로 필요한 데이터뿐만 아니라 이와 인접한 다량의 데이터들로 이루어진 블록들을 옮김으로써 공간적 지역성 이용.

계층 i+1에 존재하지 않는 데이터는 계층 i에 존재할 수 없다.

5.2 메모리 기술

메모리 계층구조에서는 네 가지 주요 기술이 사용된다. 메인메모리는 DRAM으로 구현된다. 프로세서에 더 가까운 계층인 캐쉬에는 SRAM이 사용된다.

세번째 기술은 플래쉬메모리이다. 이 휘발성 메모리는 개인용 휴대기기에서 2차메모리로 사용.

네 번째 기술은 서버의 가장 크고 가장 느린 계층으로 자기 디스크이다.

Register File – the 1st level : 빠르고, 값비싸고 작다.

SRAM 기술

단순한 집적회로로서, 읽기나 쓰기를 제공하는 접근 포트가 일반적으로 하나 있는 메모리 배열이다. 어떤 데이터든지 접근 시간은 같다. 리프레시가 필요 없으므로 접근 시간은 사이클 시간과 거의 같다.

DRAM 기술

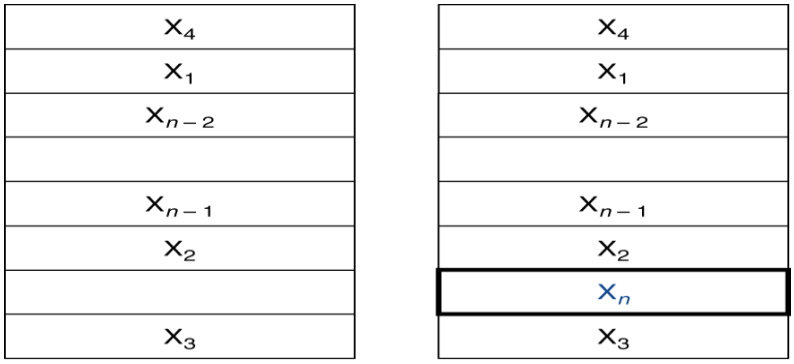
SRAM에서는 전력이 공금되는 한 그 값이 무한히 유지된다. DRAM에서는 셀에 기억되는 값이 전하로 커패시터에 저장된다. 저장된 값을 읽거나 새로 쓰기 위하여 저장된 전하에 접근하는 데 트랜지스터를 하나 사용한다. DRAM은 저장된 비트 하나당 트렌지스터 하나만 사용하므로 SRAM에 비하여 훨씬 더 집적도가 높고 값도 싼다. 주기적인 리프레쉬필요.

플래시 메모리

디스크 메모리

5.3 Cache Memory

CPU에 가까운 메모리 계층의 레벨



a. Before the reference to X_n b. After the reference to X_n

전에는 Xn 이 없는데 요구하여 miss 를 발생하고 워드 Xn 을 메모리로부터 캐시로 가져오게 된다.

Direct Mapped Cache

각 메모리의 위치가 캐시 내의 정확히 한 곳에만 사상되는 캐시 구조

블록을 찾기 위한 사상 방식: *(Block address) modulo(캐시 내에 존재하는 전체 캐쉬 블록 수)*

캐시내에 태그를 추가하여 프로세서가 요구하는 워드가 캐시 내에 있는지 없는지 알 수 있음.

태그 : 특정 계층에서 해당 블록이 요청워드와 일치한지 알려주는 주소 정보를 담고 있는 필드

Valid bit : 특정 계층에서 해당 블록이 유효한 데이터를 포함하는지를 알려 주는 필드, 이 비트가 1 이면 유효한 블록이 있는 것이다. 초기에는 0 으로 되어 있다.

직접 사상 캐쉬 접근

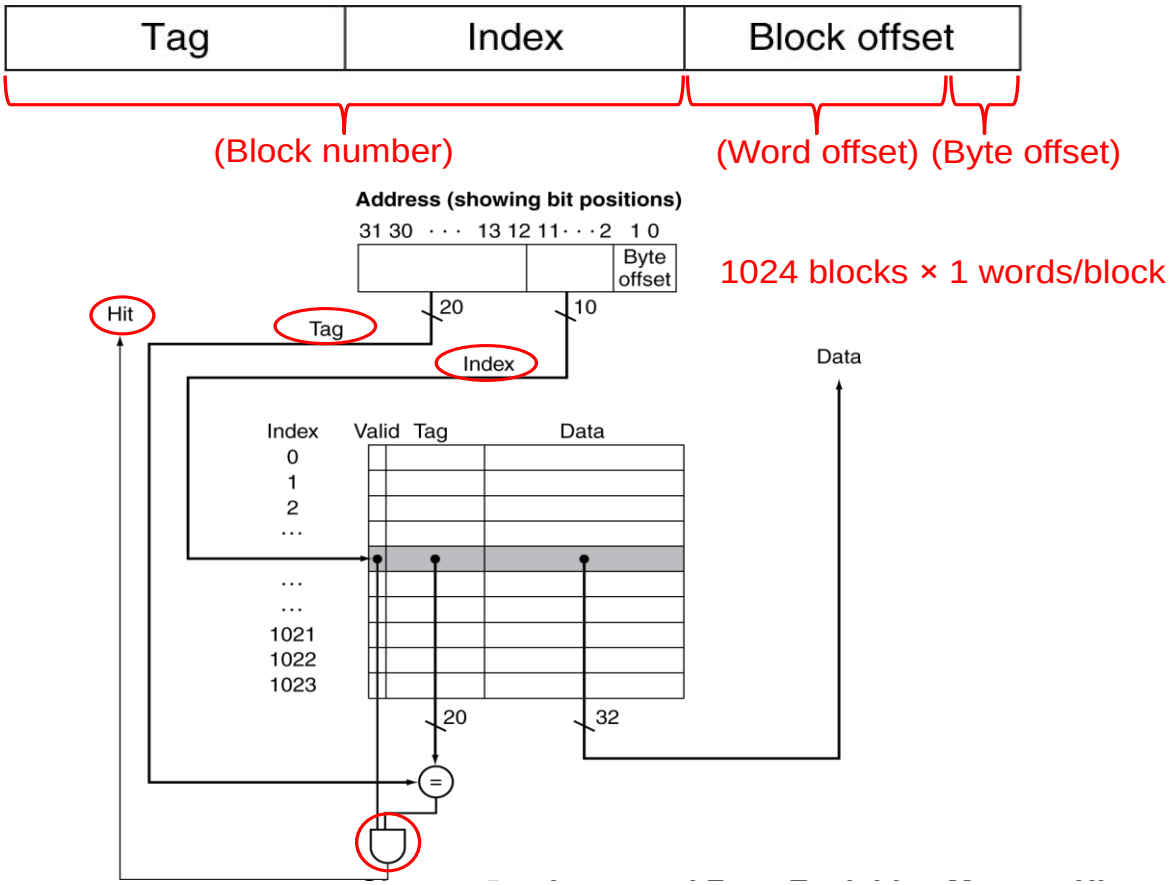
예제에서 캐시 내에 8 개의 블록이 있기 때문에 주소의 하위 3 비트가 블록의 번호를 나타냄.

만일 11110 을 하게되면 기존에 있던 110 에 있는 데이터랑 교체하게 된다.

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

word Tag Index: 8 blocks

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		



MIPS 구조에서는 모든 주소가 4 바이트로 정렬되어 4 의 배수 이기 때문에 모든 주소의 맨 오른쪽 2 비트는 워드 내부의 바이트를 나타낸다. 캐시는 1024 워드를 가지고 있을 때 하나의 블록크기가 1 워드이니 10 개의 비트를 캐시를 인덱스하는데 쓰인다. 저기서 태그와 주소의 상위 20 비트가 같고 유효비트가 1 이면 캐시는 hit 되어 읽은 워드를 프로세서에 넘겨준다. 그렇지 않으면 miss 발생

- 32 비트의 주소
- 직접 사상 캐시

● 캐시는 2^n 개 블록을 가지고 있고 n 개 비트는 인덱스를 위하여 사용된다.

● 캐쉬 블록의 크기는 2^m 개 워드이 다. m 개 비트는 블록 내부에서 워드 구별에 쓰이며, 두 비트는 줒소중 바이트 구별용으로 쓰인다.

위와 같은 가정에서 태그 필드의 크기는 **32-(n+m+2)** 이다.

직접 사상 캐시의 전체 비트수는 **2^n * (블록 크기+태그 크기+ 유효 비트 크기)** 이다.

여기서 블록크기는 2^(m+5) bit 이고 태그 크기는 32-(n+m+2) 이고 유효 비트 크기는 1 이다.

따라서 $\boldsymbol{2^n \times (2^m \times 32 + (32 - n - m - 2) + 1)}$ 이다.

하지만 일반적으로 유효비트크기와 태그 필드의 크기는 제외하고 따진다.

Block Size Considerations

공간적 지역성을 이용하기 위해서는 캐시가 한 워드보다 더 큰 크기의 블록을 사용해야 한다. 더 큰 블록의 사용은 실패율을 감소시키며, 캐시의 데이터 저장량에 비해 상대벚그올 태그 저장량을 감소시킴으로써 캐시의 효율을 향상시킬 수 있다. 하지만 블록 하나가 캐시 크기의 상당 부분을 차지하도록 블록을 크게 만들면 실패율이 올라갈 수도 있다. 왜냐하면 캐시 내에 존재할 수 있는 전체 블록의 수가 작여져 블록들 간에 상호 충돌 가능성이 커지기 떄문이다.

블록의 크기가 매우 클 경우에는 블록 내 워드 간의 공간적 지역성이 감소하게 된다. 결과적으로 실패율의 향상은 둔화된다. 블록 크기를 증가시켜서 실패에 따른 비용 증가가 생김. miss penalty는 메모리 계층구조의 다음 하위 계층에서 블록을 가져오고 캐시에 적재하는 데 걸리는 시간에 의해 결정된다. 블록을 가져오는 데 걸리는 시간은 두부분으로 나누어짐. 첫번째 워드를 찾는데 걸리는 접근 지연(latency)과 블록을 이동시키는 데 필요한 전송시간이다. 블록크기가 클수록 전송시간이 커질것임.

결과적으로 실패 손실의 증가가 실패율 감소를 압도하게 되면 캐시의 성능 또한 감소한다.

early restart -> 블록 전체를 기다리지 않고 블록 내의 용청된 워드가 도달하면 곧바로 실행을 시작하는 것이다.(전송 시간을 일부를 감추어서 miss penalty을 효과적으로 줄인것이다.)

critical word first or requested word first -> 요청한 워드를 먼저 메모리에서 캐시로 전송할 수 있도록 하는 방식. 블록의 나머지 워드들은 요청된 워드의 다음 주소부터 순서대로 전송되고, 다시 블록의 처음 워드부터 남은 워드들을 순차적 전송.

Cache Misses

제어 유닛은 실패를 탐지해야 하며 메모리로부터 데이터를 가져와서 실패를 처리해야한다.

cashe hit시 CPU는 아무일도 없는 것처럼 데이터 사용가능, cashe miss시 – 파이프 라인 지연,

Instruction cache miss

1. 원래의 PC 값(현재 PC값 -4)을 메모리로 보낸다.

2. 메인 메모리에서 읽기 동작을 지시하고 메모리가 접근을 끝날 때까지 기다린다.

3. 메모리에서 인출한 데이터를 데이터 부분에 쓰고, 태그 필드에 주소의 상위 비트를 쓰고, 유효 비트를 1로 만듦으로써 캐쉬 엔트리에 쓰기를 수행한다.

4. 명령어 수행을 첫 단계부터 다시 시작하여 캐시에서 명령어를 가져온다. 이제는 필요한 명령어를 캐쉬에서 찾을 수 있다.

데이터 접근을 위한 캐시의 제어는 근본적으로 똑같다. 실패 발생시 메모리가 데이터를 보내 줄 때까지 프로세서를 지연시키기만 하면된다.

Write-Through

정의 : 쓰기가 항상 캐시와 다음 하위 계층을 갱신하는 방식. 항상 두 계층의 데이터가 일치함을 보장.

그러나 이것은 쓰는데 시간이 엄청 오래 걸린다. 만약 이것이 최소한 100개의 프로세서 클럭 사이클 이 필요하고 명령어의 10%가 저장명령어이고 CPI가 1이면 90%*1+(10%*1+10%*100)=11 즉 약 10배정도의 성능저하가 발생.

=> Solution : **Write buffer**는 메모리에 쓰이기 위해 기다리는 동안 데이터를 저장하는 큐. 메인 메모리에 쓰기를 완료하고 나면, 쓰기 버퍼에 있는 엔트리는 다시 비게 된다. 프로세서가 쓰기 처를 하려고 할 때, 쓰기 버퍼가 모두 차 있으면 쓰기 버퍼에 빈 공간이 생길 때까지 멈춰 있어야 한다.

Write-Back

정의 : 쓰기가 발생했을 때 새로운 값은 캐시 내의 블록에만 쓴다. 그러다가 나중에 캐시에서 쫓겨날 때 쓰기에 의해 내용이 바뀐 블록이면 메모리 계층구조의 더 낮은 계층에 써진다. Write-Back 방식은 특히 메인 메모리가 처리할 수 있는 속도보다 프로세서가 쓰기를 더 빠르게 발생시키는 경우에 성능 향상시킬수 있다. 하지만 Write-Thorough 방식보다 구현이 까다로움.

쓰기 실패시의 정책

- What should happen on a write miss?
- For write-through
  - Allocate on miss: fetch the block
    - The block is in cache now
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block

Write Allocation : 전체 블록을 메모리에서 읽어 온 후, 블록 중에서 쓰기 실패를 발생시킨 워드만 덮어 쓴다.

no Write allocation : 메모리에 있는 블록의 해당 영역만 갱신하고 캐시에는 쓰지 않는 방식

Example: Intrinsity FastMATH 프로세서

-MIPS 구조와 단순한 캐시 형식을 취한 빠른 임베디드 마이크로 프로세서이다.

-12단계의 파이프라인으로 구성.

-각 사이클당 명령어 한 개와 데이터 워드 한 개 요청가능

-지연 없는 파이프랑인 구현을 위해 분리된 명령어 캐시와 데이터 캐시가 사용되어짐.

각 캐시의 크기는 16KiB(즉 4096워드) 이고 16워드 크기의 블록을 갖는다.

-D-cache: write-through or write-back

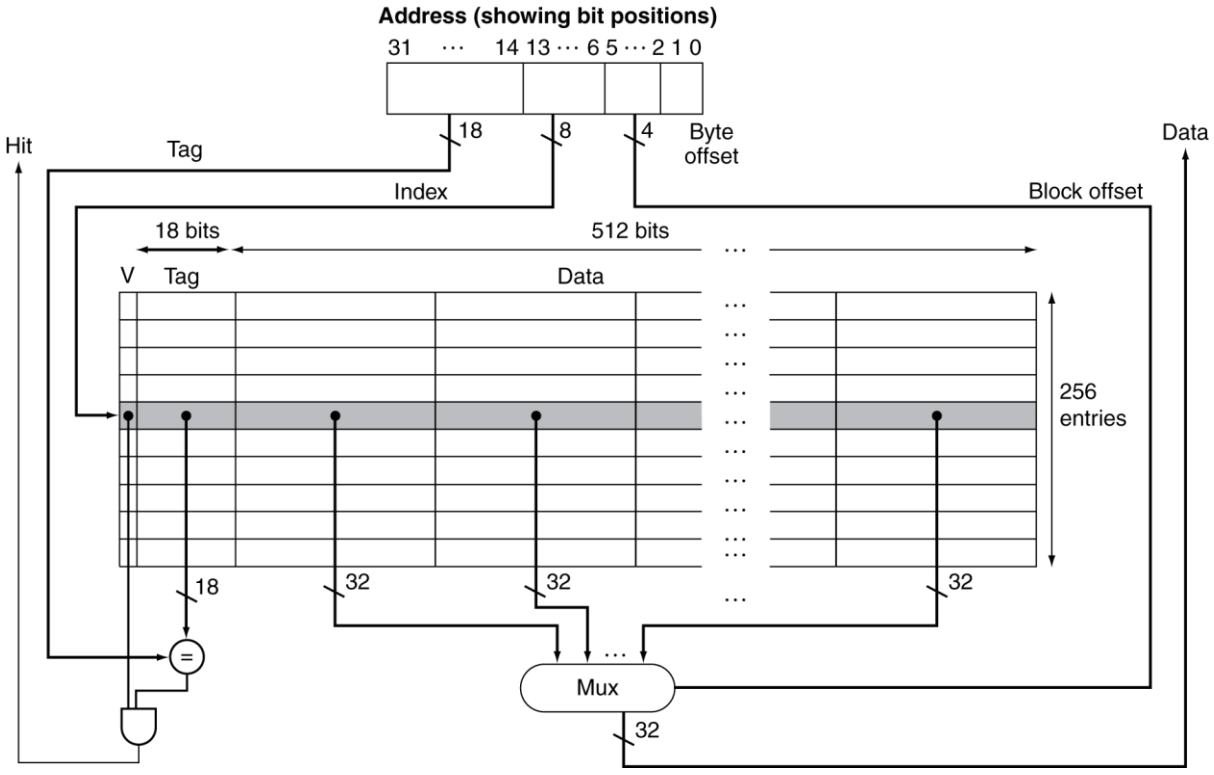
- 한 항목만 담을 수 있는 쓰기 버퍼 존재

각 캐시에 대한 읽기 요청에 대한 처리 단계

1. 주소를 해당 캐시에 보낸다. 주소는 PC(명령어인 경우)나 ALU(데이터인 경우)로부터 나온다.

2. 캐시 적중일 경우 요청된 워드가 캐시의 데이터 선에 나온다. 요청된 블록에는 16개의 워드가 존재하므로 정확한 워드를 선택하여야한다. 블록 인덱스 필드가 멀티플렉서 제어를 위해 사용된다.(Block offset 신호) 이 신호는 블록 내 16개 워드 중에서 요청한 워드를 선택한다.

3. 캐시 실패일 경우 메인 메모리로 주소를 보내야 한다. 메모리가 데이터를 보내주면, 캐시에 쓰고 난 뒤 요청을 처리하기 위해 읽는다.



5.4 캐시 성능의 측정 및 향상

두가지 방법으로 캐시의 성능 향상.

- 두개의 다른 메모리 블록이 하나의 캐시 위치를 두고 경쟁하는 확률을 줄여서 실패율을 줄임

- Multilevel caching : 메모리 계층구조에 새로운 계층을 추가함으로써 실패 손실을 줄임.

*CPU 시간 = (CPU 클럭 사이클 + 메모리 지연 클럭 사이클) * 클럭 사이클 시간*

메모리 지연 클럭 사이클은 주로 캐시 실패 때문에 생긴다.

메모리 지연 클럭 사이클 = 읽기 지연 사이클 + 쓰기 지연 사이클

쓰기 버퍼 지연시간은 무시할 수 있어서 읽기 지연과 쓰기 지연을 합치면 밑에와 같이 나온다.

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Average Access Time

- Hit time is also important for performance

Average memory access time (AMAT) : Hit time(캐시 적중시간) + Miss rate × Miss penalty

When CPU performance increased -> Miss penalty becomes more significant

Decreasing base CPI -> Greater proportion of time spent on memory stalls

Increasing clock rate -> Memory stalls account for more CPU cycles

Can't neglect cache behavior when evaluating system performance

Alternative Placement Scheme

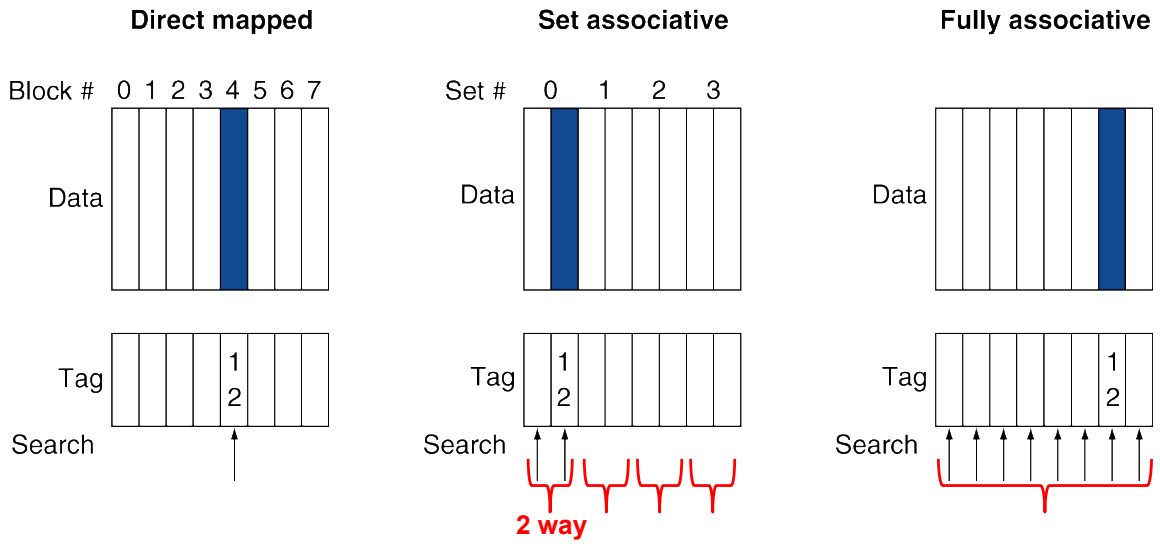
directed mapped 는 메모리 블록을 캐시에 넣을 때 각 블록이 캐시의 딱 한 곳에만 들어 갈 수 있는 배치방법이다.

fully associative 는 블록이 캐시 내의 어느 곳에나 위치할 수 있는 방식, 검색을 효율적으로 하기윟서 각 캐시 엔트리와 연결도니 비교기를 이용하여 병렬로 검색한다. 이러한 비교기는 비싸다.

set associatvie 는 각 블록이 배치 될 수 있는 위치의 개수가 고정되어 있는 캐시 구조

n-way set associatvie 는 각 블록당 n개의 배치 가능한 위치를 갖는 것이다.

Associative Cache Example



스스로 점검하기 및 예제

p384, **p396-397**, p405, p407, p409