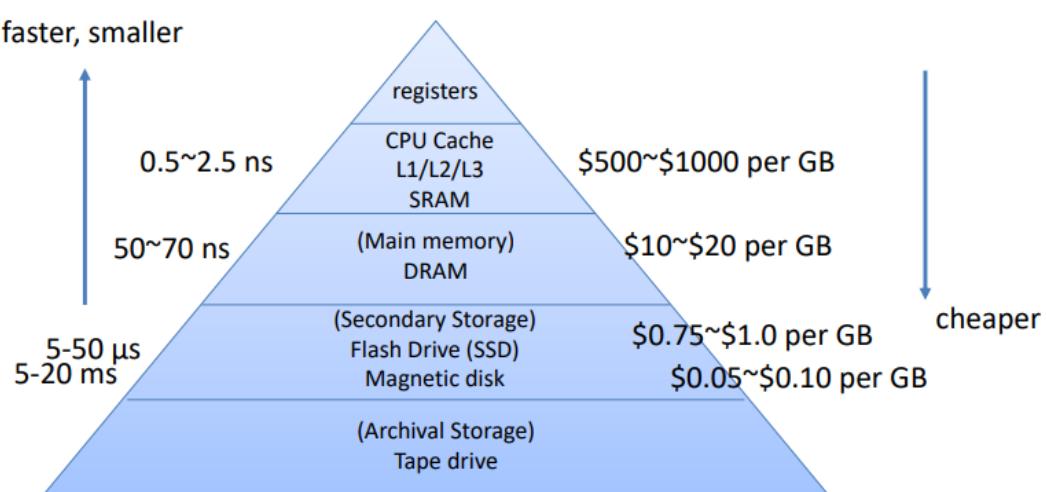




Chapter_05.Memory Hierarch

Memory Hierarchy

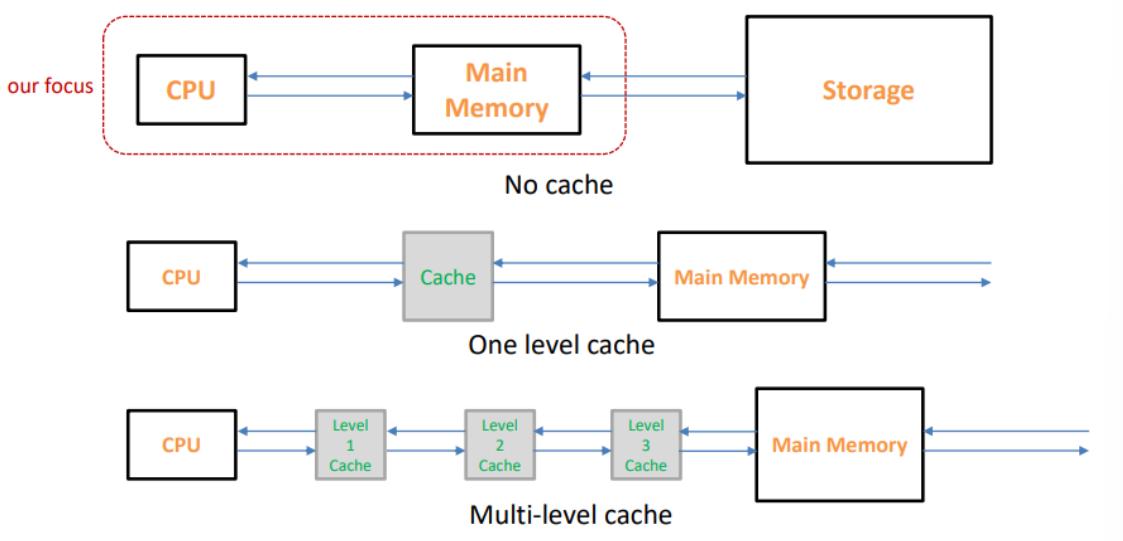
- Multiple levels of memory with different speeds and sizes(속도와 크기가 다른 여러 수준의 메모리)
 - memory: 데이터를 저장 할 수 있는 모든 매체(예를 들어, DRAM, disk,...)



Cache Memory Concept

- Cache memory

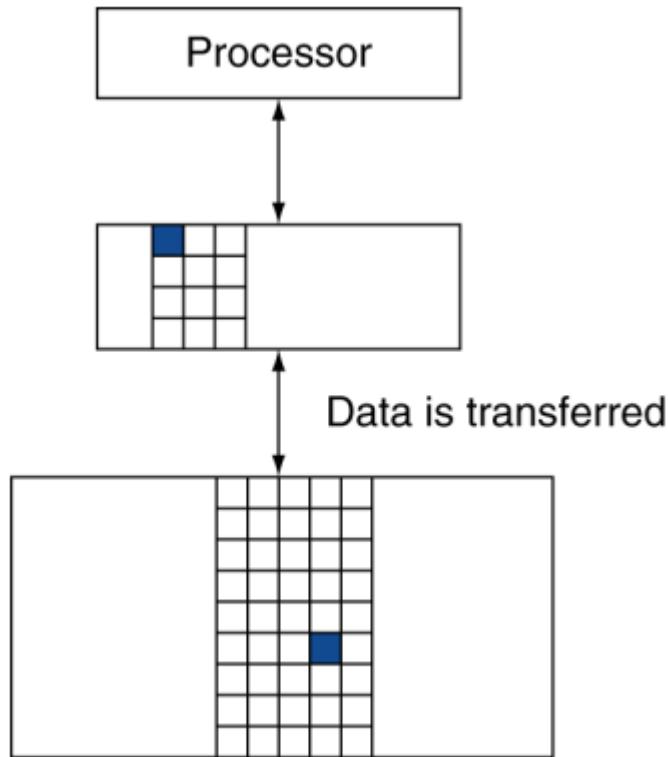
- 크기는 작지만 빠르다. 이는 데이터 유지성을 잡아준다. 그리고 memory로 가는데 오래 걸리는 시간을 절약하기 위해 시간 접근성을 향상시켜 준다.
- Transparent



Cache Memory

- Underlying principle of Cache
 - Principle of Locality
 - 프로그램은 접근했던 데이터의 근처에 또 접근 하려는 경향이 있다
 - 시간의 지역성
 - 만약 데이터가 접근해 올 때, 데이터에 액세스하면 곧 다시 액세스 할 가능성이 있다.
 - 공간의 지역성
 - 만약, 데이터가 접근해 올 때, 데이터에 액세스하면, 주변 데이터에도 액세스 할 가능성이 있다.
- cache 메모리는 Lower level의 메모리보다 크기가 작다. 그래서 데이터를 저장 할 때는 선별적으로 저장한다.
- 캐시를 할 수 있는 이유는 , principle of locality 덕분이다.**

Cache Terminology



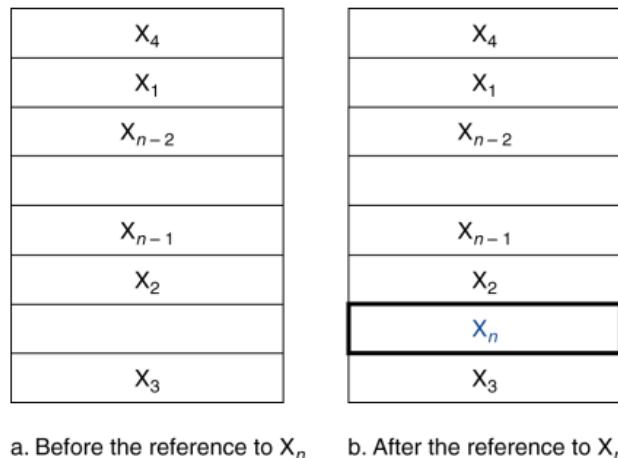
- Block(or line)
 - 정보의 가장 작은 단위이다.
 - cpu가 main memory가 아니라, cache에서 바로 data를 가져오는 경우,
- Hit rate(hit ratio)
 - 요즘은 hit ratio 가 약 99.999...%!
- Miss, miss rate
 - Hit과 반대로 cpu가 main memory 에서 data를 가져오는 경우.
- Hit time
 - 상위 레벨에 접근하는데 걸리는 시간(hit인지, miss인지 결정하는 시간도 포함)
 - Cache 에서 CPU 로 데이터를 전송해 오는데 걸리는 시간
- Miss penalty
 - main memory 로 access해서 data를 가져와 cache에 저장하고, cpu로 가져오는 데 걸리는 시간

- miss는 거의 일어나지 않기 때문에, miss penalty를 줄인다 하더라도 hit time을 줄이는 것에 비해 성능 향상 정도가 작다.

Cache Access

- Assume that
 - CPU가 한 번에 1word씩 요청을 한다고 가정하자.
 - 또한 block size가 1 word라고 하자.
 - memory에 access를 해서, 가져온 data를 cache에 저장하는데 이 때, 어떤 순서로 저장할지에 대한 방법이 필요하다. → cache placement issue

- Memory access: X_1, \dots, X_{n-1}, X_n



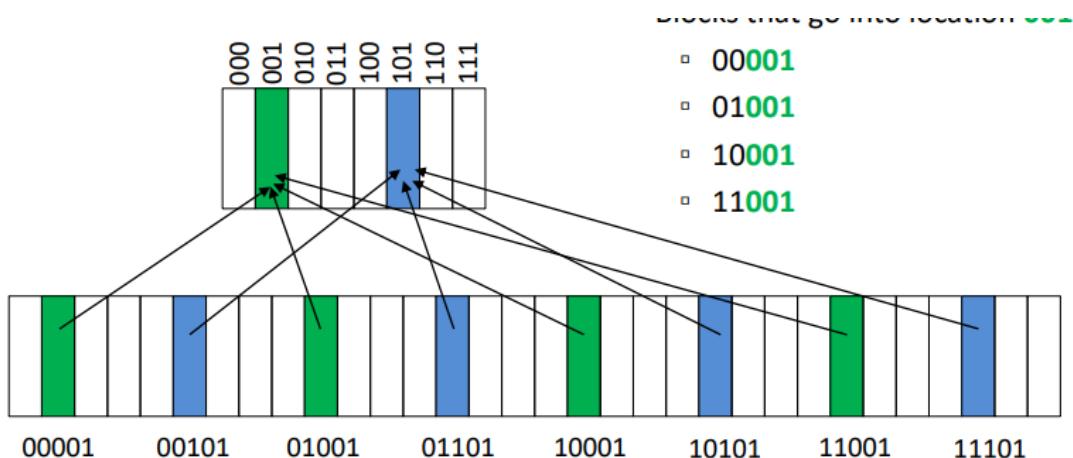
이 그림을 통해 X_n 을 어디에 둬야 할지 정해야 할 필요가 있다.

Direct-mapped Cache

- In direct-mapped cache:
- 캐시의 위치는 메모리 주소의 함수에 의해 결정된다.
- cache location을 memory address의 LSB bits를 이용해 결정한다. 이로 인해, 블럭(word)은 오직, 캐시의 한 부분에만 갈 수 있다. ↔ fully associative cache(캐시 로케이션과 반대되는 개념)
- Cache slot = (word address) mod(#of cache blocks)
- #of cache blocks are in power of 2

- Blocks that go into location 001

- 00001
- 01001
- 10001
- 11001

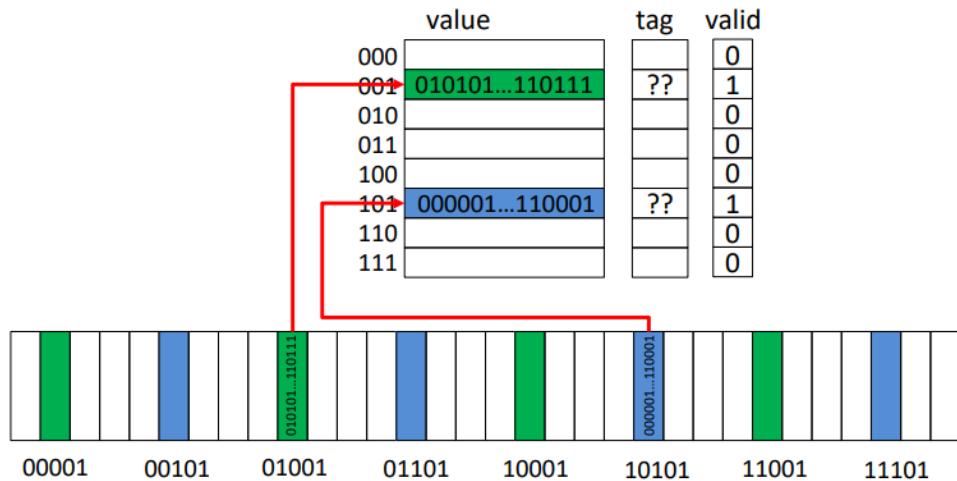


위의 그림은 #of cache blocks = 8

- 이처럼, direct-mapped cache의 경우 캐시에 빈 공간이 발생한다.
- 여러 개의 블럭이 캐시의 한 location으로 mapping
 - 00 001, 01 001, 10 001, 11 001 은 캐시의 001 블럭으로 맵핑된다.
 - 캐시의 001 블럭에 저장된 값이 어디서 왔는지 알 방법도 필요하다. 이때 tag bit 가 등장한다

■ Tags

- High-order bits of the address – **00001**, **01001**, **10001**, **11001**



- tag bit: address 의 high-order bits
 - **00001**, **01001**, **10001**, **11001**
- valid bit: valid bit == 1일 때, 실제 캐싱된 데이터, valid bit가 0인 경우, 쓰레기 값
 - valid bit = 1 → 사용 가능한 cache entry
 - valid bit = 0 → 사용 불가능
- dirty bit : main memory 에 저장된 data와 cache에 저장된 data가 다른 경우, 즉, data 가 modified된 경우 dirty bit가 1이다.
 - dirty bit = 1 → 해당 데이터를 메인 메모리에 업데이트하고, 새로운 값으로 덮어 쓴다.
 - dirty bit = 0 → 새로운 값을 바로 덮어 쓴다.
 - dirty bit 이용 시 memory access 감소

Accessing a Cache

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.9b)	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	miss (5.9c)	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
22	10110_{two}	hit	$(10110_{\text{two}} \bmod 8) = 110_{\text{two}}$
26	11010_{two}	hit	$(11010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	miss (5.9d)	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
3	00011_{two}	miss (5.9e)	$(00011_{\text{two}} \bmod 8) = 011_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$
18	10010_{two}	miss (5.9f)	$(10010_{\text{two}} \bmod 8) = 010_{\text{two}}$
16	10000_{two}	hit	$(10000_{\text{two}} \bmod 8) = 000_{\text{two}}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

Index	V	Tag	Data
000	N		
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

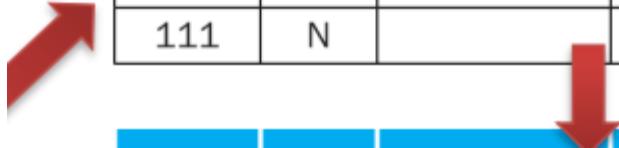
Index	V	Tag	Data
000	Y	10 _{two}	Memory (10000 _{two})
001	N		
010	Y	11 _{two}	Memory (11010 _{two})
011	N		
100	N		
101	N		
110	Y	10 _{two}	Memory (10110 _{two})
111	N		

위의 표를 보면, Hit or miss in cache 라는 부분이 있다. Hit은 캐시에서 데이터를 가져와서 CPU에 저장하는 것, Miss는 MM에서 데이터를 가져와서 CPU에 저장(로드) 하는 것이다.

이 표에선 Hit이 아니라, Miss 된 Data를 저장하는 과정을 나타내고 있다.

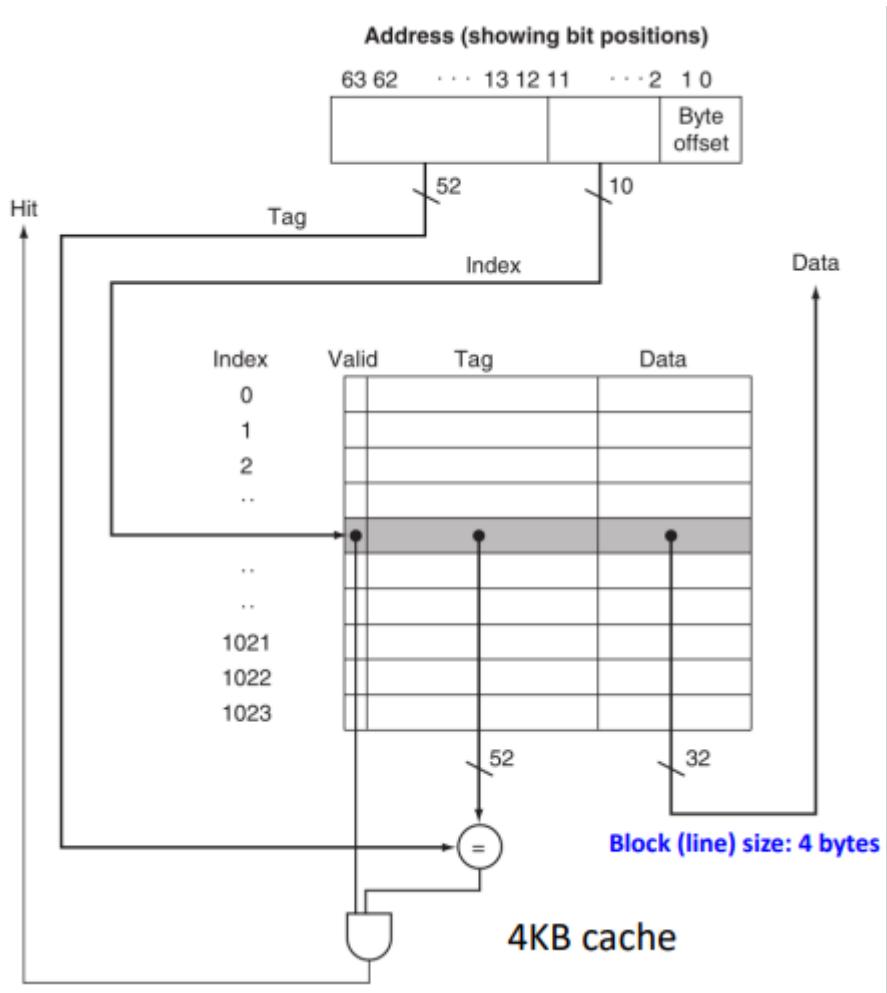
여기서 단계별로 따라가면 되는데, 인덱스 110이 있고, tag bit는 10(2)이다. 메모리에 로드되는 데이터는 10110(2)이다.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		



Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

Address Subdivision



총 64bit 의 Address가 있다.

Data는 4byte == 1word 의 크기를 가지고 있다.

먼저 짚고 가야 할 것은, Byte offset이다. Byte offset은 Data에 특정 바이트를 짚어주는 역할을 하며 크기는 2bit==1byte 이다.

그럼 byte offset옆에 있는 10bit는 Index, 즉 Valid 를 가리키는데, 이건 어떤 특정 한 라인을 가리키기 위한 것이다. 그 크기가 왜 10비트 이냐면 옆에 인덱스의 크기가 총 1024즉 2^{10} 이다. Valid가 10개의 위치를 가리킬 수 있으며, 그 크기가 총 10bit인 것이다.

이제 남은 52bit는 Tag를 가리킨다.

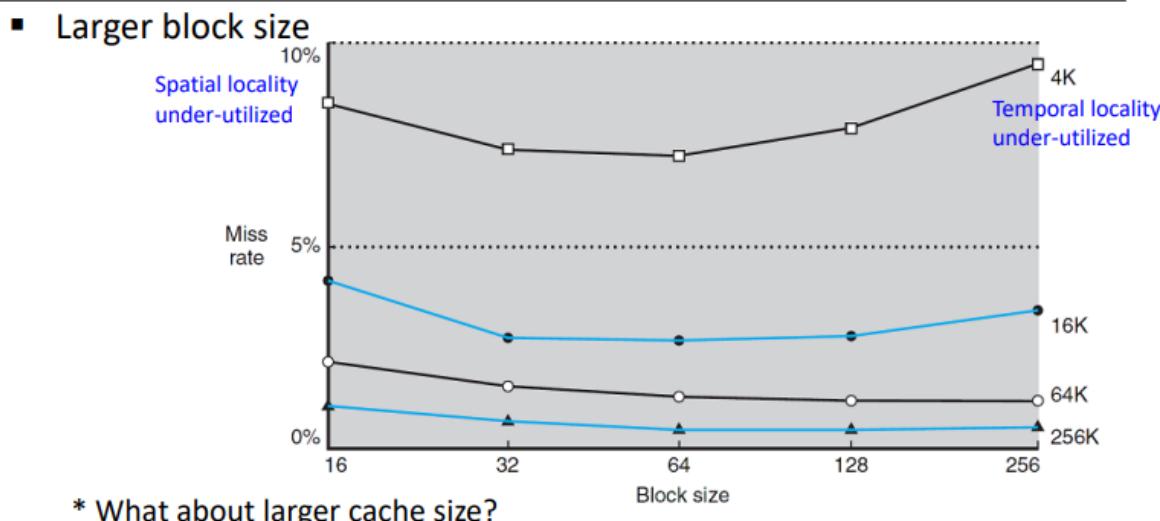
여기서 캐시의 크기를 4KB라고 했는데, 통상 cache의 size를 얘기할 때, 저장 할 수 있는 data의 size만 얘기한다.

Cache 안의 total number of bits

- Address is divided into tag part and index part
 - tag part: 52 bits
 - index part: 10 bits
 - byte offset: 2 bits
- Total number of bits in the cache
 n : number of index bits
 m : 2^m is the # of words
 $(m=0 \text{ if 1 word in a line})$

$$2^n \times \sum \begin{array}{l} \text{valid bit: 1} \\ \text{tag bits: } 64-(n+m+2) \\ \text{data bits: } 2^m \times 32 \end{array}$$

Block Size and Miss Rate



그래프를 해석해 보면, y축이 두 개가 있다. 왼쪽 y축 변량은 Miss rate인데, 이건 %가 작을 수록 좋다. 오른쪽 y축 변량은 Temporal locality under-utilized인데, cache의 크기이다. 이 크기가 세로로 긴 직사각형인지, 혹은 정사각형인지, 가로로 긴 직사각형 인지에 따라, 성능의 향상 또한 차이가 난다.

cache 가 세로로 길면 그래프의 X축 상에서 왼쪽 방향에 있게 되는데, 가져와야 할 데이터는 Tag bit, Index, Address등 data가 길어야 하는데, 이걸 충분히 가져오지 못하면, Miss

rate가 올라간다. 즉, 데이터 에러떠서 메인 메모리까지 가야 한다는 소리다.

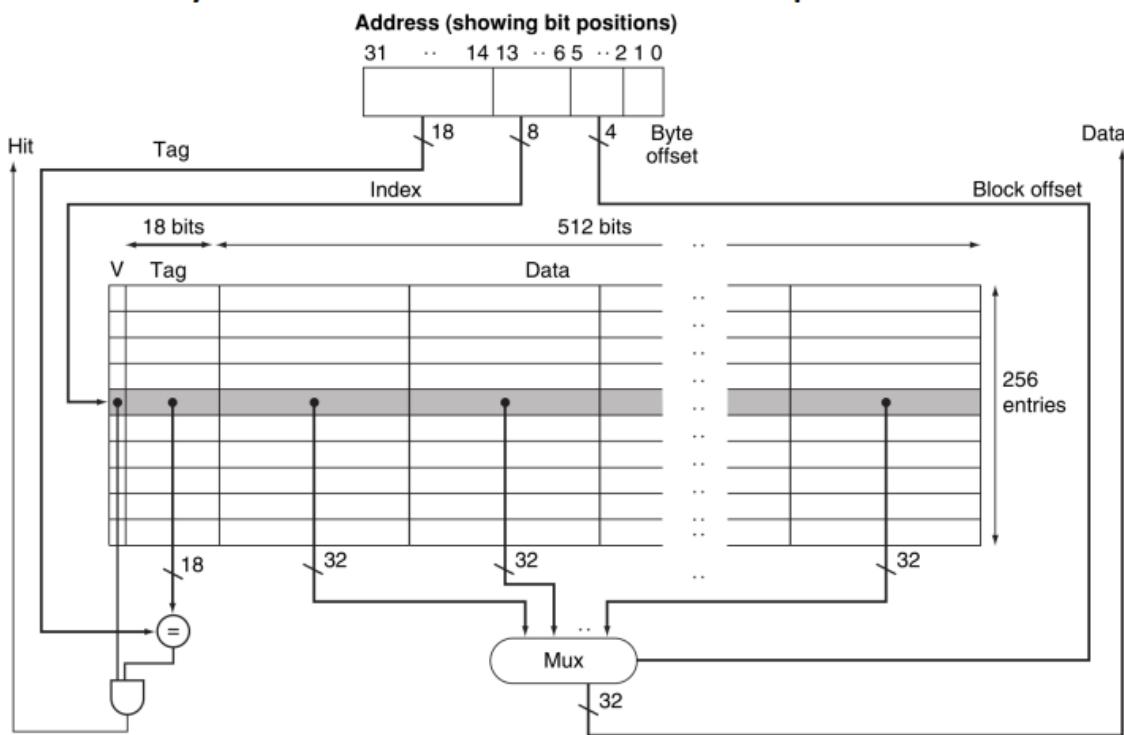
cache가 가로로 길면 반대로 그래프의 X축 상에서 오른쪽 방향에 있게 되는데, 대신, 가져올 수 있는 데이터의 수, 그러니깐, 가로로 길면 해당 Index의 정보는 충분히 가져올 수 있지만, 갯수 index는 줄어들어서 이 또한 Miss rate가 증가한다.

Miss penalty

- Time to load a block from MM(Main memory) to Cache
 - Set-up time + transfer time
 - Set-up time → 데이터가 이동하기 전, 메모리상에서 데이터를 준비하고 로드하는 시간
 - Transfer time → 데이터가 MM에서 Cache로 운송되는 시간
- Reducing the miss penalty(by hiding the transfer time)
- Cache에서 CPU로 데이터가 로드되어야 동작을 수행한다. 이 때, FM으로, 앞에서부터 단위 크기 만큼 데이터를 순차적으로 보내준다. 그래서 입력을 받고 순차적으로 동작을 수행한다.
 - “early - restart” → 데이터를 다 받을 때까지 걸리는 시간이 매우 기니깐, 데이터를 앞에서부터 순차적으로 전달해줘서 받은 순서대로 동작을 실행한다.
 - “critical word first” → CPU가 원하는 데이터가 들어올 때 까지 수행을 안 하고 있으니깐 Cache에서 CPU가 원하는 데이터부터 먼저 주고, 다시 앞에서부터 데이터를 순차적으로 주는 것이다.

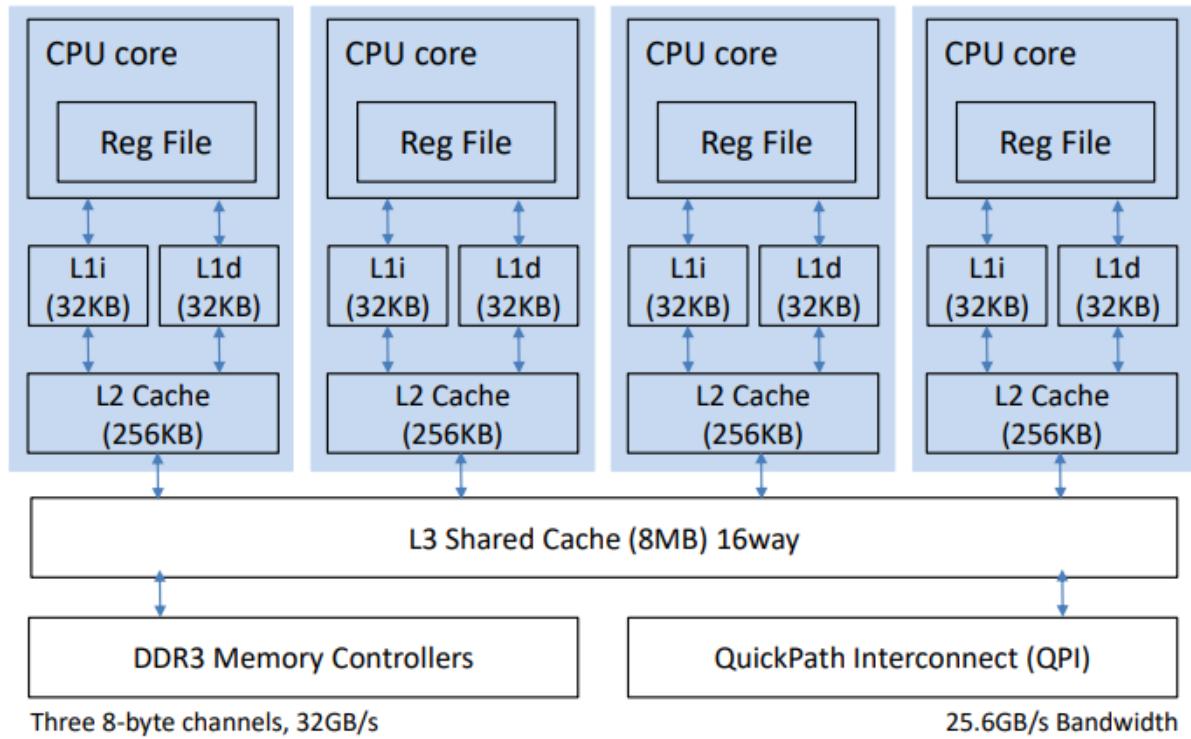
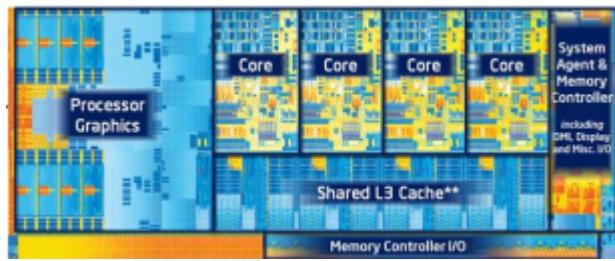
Example Cache

- Intrinsity FastMATH embedded microprocessor



- Block offset이 등장한다. 이 Block offset은 CPU내의 어떤 한 특정한 Data를 지정해 준다. 여기선 Block offset이 4라고 적혀있는데, $2^4 = 16$, 즉 16개의 data가 존재하며, Block offset은 한 개의 Data 즉, 4byte(=1word)크기의 한 Data를 지정해 준다.
- 2bit는 Byte offset이다.
- 8bit는 $2^8=256$ 으로, 8개의 index가 존재한다.
- 나머지 18bit는 Tag 부분으로 들어간다.
- 여기서 Mux는 어떤 단일 혹은 여러 개의 입력 값이 들어오면, 무조건 한 개만 통과시켜 주는 소자이다. 여기서 16개의 data 중 1개의 data 만 통과시켜준다.
- Tag 부분에서 = 기호가 있는데, 이는 0/1을 반환하는 것으로 True면 1, False 면 0를 반환한다.
- 그 후, Vaild와 아까 반환한 0 혹은 1의 값을 AND 연산해서 1이면 Hit , 0이면 Miss를 반환한다.

Intel core i7 Block Diagram



Four x86 SMT(Simultaneous Multithreading) Processors Dedicated L1, L2 Cache Shared L3 Cache

- 네 개의 x86의 SMT processors즉, 4개의 Core가 있다. 그리고 L1, L2 캐시가 있고, 이들은 모두 L3 Cache를 공유한다.
- 우선 4개의 CPU Core가 있다. 안에 Reg File은 여러 Reg들의 집합체이다.
- L1을 L1i, L1d로 구분해 놓았는데, i 는 instruction이 송신되고, d 는 data, instruction 이 아닌 data들이 교환된다. 이렇게 나눈 이유는 i와 d가 받아들이는 정보에 따라 처리하는 방식에서 차이가 나기 때문에 성능 향상을 위해서 구분해 놓은 것이다.
- 그리고 L1i, L1d의 Cache들은 L2 Cache와 교환하고 있다.
- 이것이 한 Core이며, 이 여러개의 Core들이 L3 Cache와 교환하고 있다. 그리고 L3 캐시는 Memory 와 교환하고 있는데, MM에는 DDR3, QPI(CPU간의 interconnect하기

위한 존재) 가 존재한다.

Cache Miss Handling in Processor

- Separate cache for instruction and data stream
 - Instruction and data cache at L1

Processors	MIPS32 74K Atheros AR9344	Intel i7-4770	Apple A10	ARM Cortex-A57
Instruction cache	32KB, 32B line, 4-way	32KB, 64B line, 8-way	64KB, 64B line, 4-way	48KB, 64B line, 3-way
Data cache	32KB, 32B line, 4-way	32KB, 64B line, 8-way	64KB, 64B line, 4-way	48KB, 64B line, 2-way

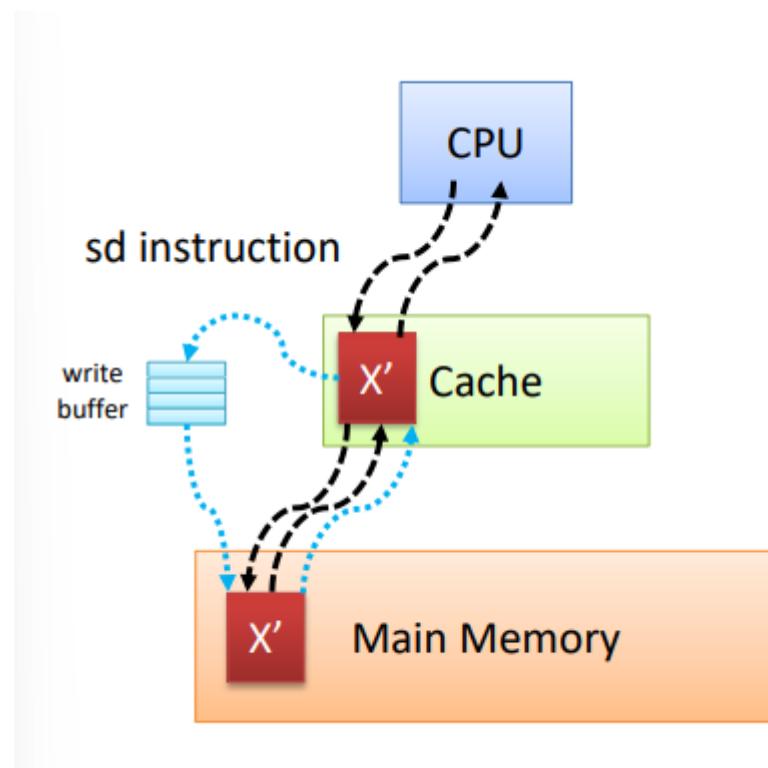
- Cache miss on instruction(identical for data miss)
 - Cache miss 가 일어났을 때, 소자에서 일어나는 일련의 과정들이다.
 1. Send PC-4 memory unit : Program Counter은 어떤 데이터를 전송하면 다음 데 이터를 전송 하도록 1word(4byte)의 크기만큼 현재 PC에서 더해준다. 하지만 여기 선 아까 보낸 값이 미스가 났기 때문에, 아까 전의 값을 되돌려 받아야 하므로 PC+4 가 되어있던 것을 PC로 만들기 위해, PC에 -4 을 더해준다.
 2. Instruct MM(main memory) to perform read : Main memory 에 아까 잘못 전송 한 값의 address 를 보내고, data를 read한다.
 3. Wait until data is ready : data가 준비될 때까지 대기한다.
 4. Write data to cache entry, fill up the tag and valid fields : 가져온 data를 cache entry 에 write한다. tag와 valid field 를 채운다.
 5. Re-send instruction address to the memory - hit! : 채운 instruction address 를 다시 memory 에 전송한다. 이번에는 cache에 해당 address와 동일한 instruction 이 존재한다 → Hit!

Write Handling in Cache

- Write-through cache
 - **sd instruction** 은 data cache에 data를 write한다. → data cache와 MM와 다른 value를 가진다.
 - 이러한 경우, cache와 MM가 inconsistent 한다고 한다.

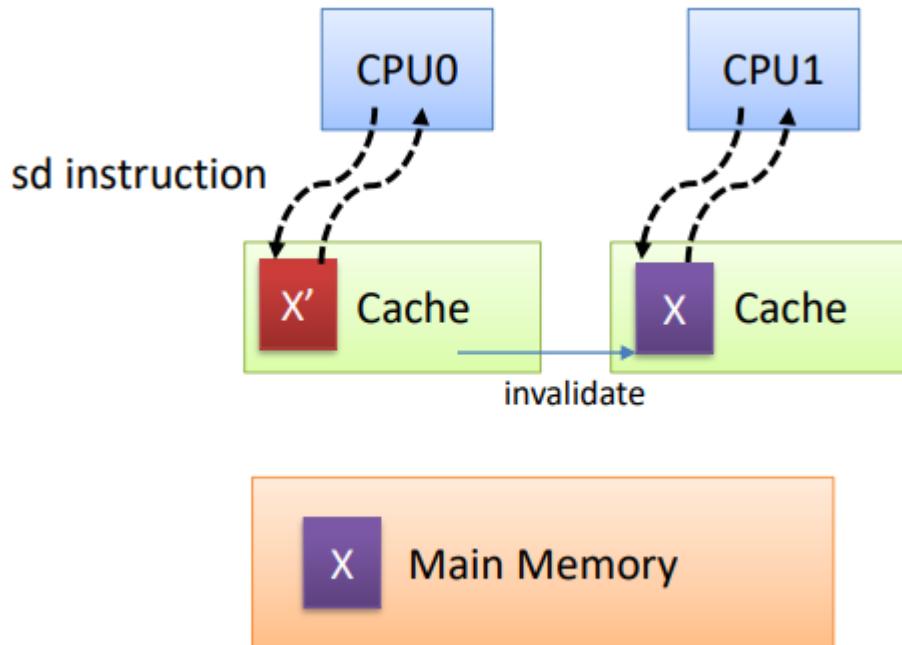
- cache와 memory 를 consistent하게 유지하는 방법은, cache와 memory 둘 다 data를 write하는 것이다 . → 이 방법이 write-through 라고 한다.
 - 이 방법에는 cache와 MM 둘 다 update 한다. → data consistency 가 유지된다.
 - 하지만 매번 MM에 access 하면 속도가 매우 느려지기 때문에, write buffer을 사용 한다.
- Write-back cache
 - write-back cache 는 새로운 value가 cache의 block에만 write 된다.
 - 수정된 block은 교체될 때, MM에 write 된다.
 - MM에 매번 access 하지 않아도 되기 때문에, 빠르게 write 할 수 있다. 하지만 구현이 매우 어렵다.

Write-through cache	Division	Write-back cache
Cache와 MM가 업데이트 가 된 후, write를 수행한다.	Way	Cache에만 업데이트가 완료된다면, Write 를 수행한다.
Data 가 유지된다.	Data	MM은 오직, block이 cache로부터 교체된 후, 동기화된다.



- 위에는 Write-through cache다. 여기서 주목할것은 Cache 의 존재 여부는 메인 메모리로 가지 않는 것에 목적이 있다. 하지만, 여기선 MM로 데이터가 옮겨간다. 이렇게 되

면 수행 시간이 매우 느려지게 되는데, 이 때, Write buffer가 존재해서 이곳에 Cache의 데이터를 차곡차곡 쌓는다. 그리고 CPU와 Cache 간에는 활동을 계속 하고 있고, MM에 아까 write buffer에서 받은 데이터를 순차적으로 load한다.



- 위의 그림은 Write-back cache 를 나타낸 것이다. 여기엔 CPU가 여러 개가 있다. 그림을 잘 보면 Cache에 적혀있는 값이 다르다. 여기서 원하는 것을 X' 인데, 이렇게 되면, CPU1 밑단 Cache에 CPU0가 가지고 있는 Cache 값을 옮겨줘야 한다. 즉, Cache 간 교류가 일어나야 한다는 것이다. 만약, CPU가 여러개면 Cache는 더 많은 교류를 통해 값을 옮겨줘야 한다. 그리고 이후, 수정이 완료되면, MM에 write한다.

Cache Performance

- CPU time
 - Normal execution : includes cache hit time : Miss없이 Hit만 일어났을 때, 메모리로 가지 않을 때의 수행을 말한다.
 - Memory stall cycle : cache miss penalty
 - stall : 어떠한 Action이 delay or post 되는 것.

$$\begin{aligned} \text{CPU time} &= \text{total clock cycles} \times \text{clock cycle time} \\ &= (\text{CPU cycles} + \text{Memory stall cycles}) \times \text{clock cycle time} \end{aligned}$$

- 갯수 * 길이 → total clock cycles x clock cycle time
- Memory stall cycle : Miss 를 처리하는 , 즉, 메모리가 기다리는 cycle , 이것만 잘 파악 하면, Hit, Miss를 파악하는데 도움이 된다. 그래서 이것을 어떻게 알 수 있을지 밑에 간략히 소개해놓았다. 즉 Cache Miss 때문에 얼마나 cycle가 소비되고 추가되는지에 관한 것이다.(일단 구현 자체는 매우 어렵다)
- Assume the read and write miss penalties are the same
 - Write-through : complication from write buffer → MM까지 갔다오는 시간 write, miss 등등이 얼마나 걸리는지 단순히 계산 할 수 있지만, write buffer가 있을 경우, CPU 입장에선 소비하는 memory cycle 로 가는 것이 줄어들었지만, 얼만큼 줄었는지는 예측 불가능하다.
 - Write-back : potential stall from sync when a block is replaced → Cache 다 차서 밀려나야 하는 시점에는 메모리에 가야하는데, 이게 언제 얼마나 발생할지는 미지수이다. 즉, 데이터마다 다르고, 예측하기도 어렵다.

Memory stall cycles = Read stall cycles + Write stall cycles

$$= \text{inst count} \times \frac{\text{memory access}}{\text{inst count}} \times \text{miss rate} \times \text{miss penalty}$$

Memory access 는 Id, sd 로 한다.

Cache Performance

- cache performance를 향상시키기 위한 방법에는 두 가지가 있다.
 1. 두 개의 다른 block 이 cache의 같은 location에 load될 가능성을 줄여서 miss rate를 줄인다.
 2. hierarchy에 추가적인 level을 더해 miss penalty 를 줄인다.
→ multi-level caching
- CPU time : 실제 프로그램을 실행한 CPU execution cycle 와 memory system을 기다리는 memory-stall cycle로 나눌 수 있다.
 - CPU execution cycle : cache access가 hit했을 때, 소모되는 시간, 즉 cache hit time 또한 포함된다.
 - CPU time = (CPU execution cycles + Memory-stall cycle) x clock cycle time
- Memory-stall cycle : memory-stall cycles = read stall cycles + write stall cycles

- **Read stall cycle** : $\text{read stall cycles} = (\text{reads}/\text{program}) \times \text{read miss rate} \times \text{read miss penalty}$
- 하지만 write의 경우 더 복잡하다.
 - write-through : write buffer 가 가득찬 경우 , buffer에 자리가 날 때까지 stall 해야 한다.
 - write-back : block이 replace 되는 경우, MM에 write 하는 동안 stall
 - 그렇지만, 단순한 계산을 위해 read penalty 와 write penalty 를 같다고 하고 계산 을 한다.
- **memory-stall cycles = $N \times (\text{memory access}/N) \times \text{miss rate} \times \text{miss penalty}$**
 - N : of instructions
- Instruction cache miss rate: 2% ▪ Miss penalty: 100 cycles
- Data cache miss rate: 4% ▪ Load and store: 36% of instructions
- Ideal case CPI: 2
- Q. How much faster is the processor with perfect cache(Miss 발생 X)?
 - 모든 instruction은 적어도 한 번은 값을 가져오기 위해 MM에 access 해야 한다.
 - Number of instructions: N
 - Instruction miss and data miss are separate
 - total penalty cycles = inst miss cycles + data miss cycles
 - inst miss cycles = $N \times 2\% \times 100 = 2N$
 - data miss cycles = $N \times 36\% \times 4\% \times 100 = 1.44N$
 - Total cycles = $2N + 2N + 1.44N = 5.44N$
 - 계산 전반적으로 재검토
Data miss 시 cache hit 사이클도 추가해야 함.
 - Total cycles for perfect cache = $2N$
 - Performance is better by $5.44N/2N = 2.72$

Data miss 시 cache hit 시간도 같이 계산에 포함해야 할까? 답은 F, 왜냐면, CPU execution cycle에서 cache access가 hit했을 때, 소모되는 시간, 즉 cache hit time 또한 포함되었기 때문이다.

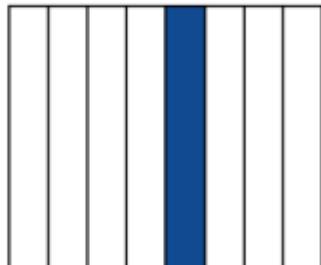
Associative Cache

1. Direct-mapped cache : 블락이 어떤 한 위치에만 지정 될 수 있다.

Direct mapped

Block # 0 1 2 3 4 5 6 7

Data



Tag

Search

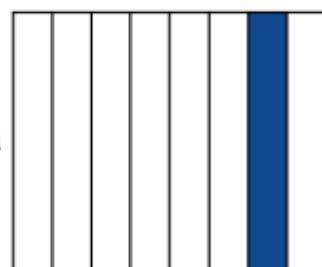
1
2



2. Fully associative cache : 블락이 캐시의 어느 위치에나 올 수 있다.

Fully associative

Data



Tag

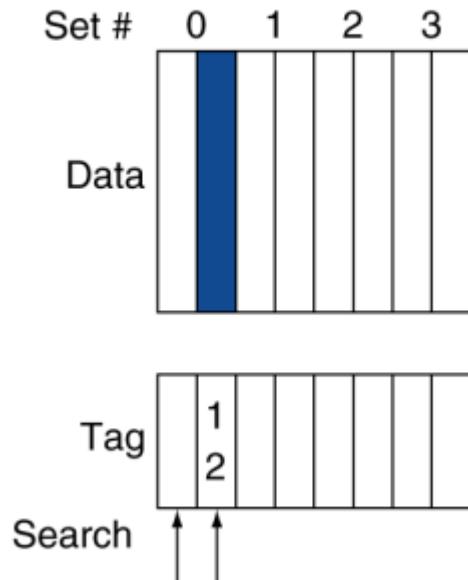
Search

1
2



3. Set associative cache : 한 개의 위치보단 선택지가 2개가 있어서 선택의 폭이 1 증가 하지만, 어느 위치에나 올 수 있는 것은 아니다. Set 이기에 Set 안에서만 움직일 수 있다.

Set associative



Associative Cache

- Direct-mapped cache 와 fully associative cache 는 모두 associative cache 의 special cases다.
- Advantage of associative cache
 - Decreased miss rate

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

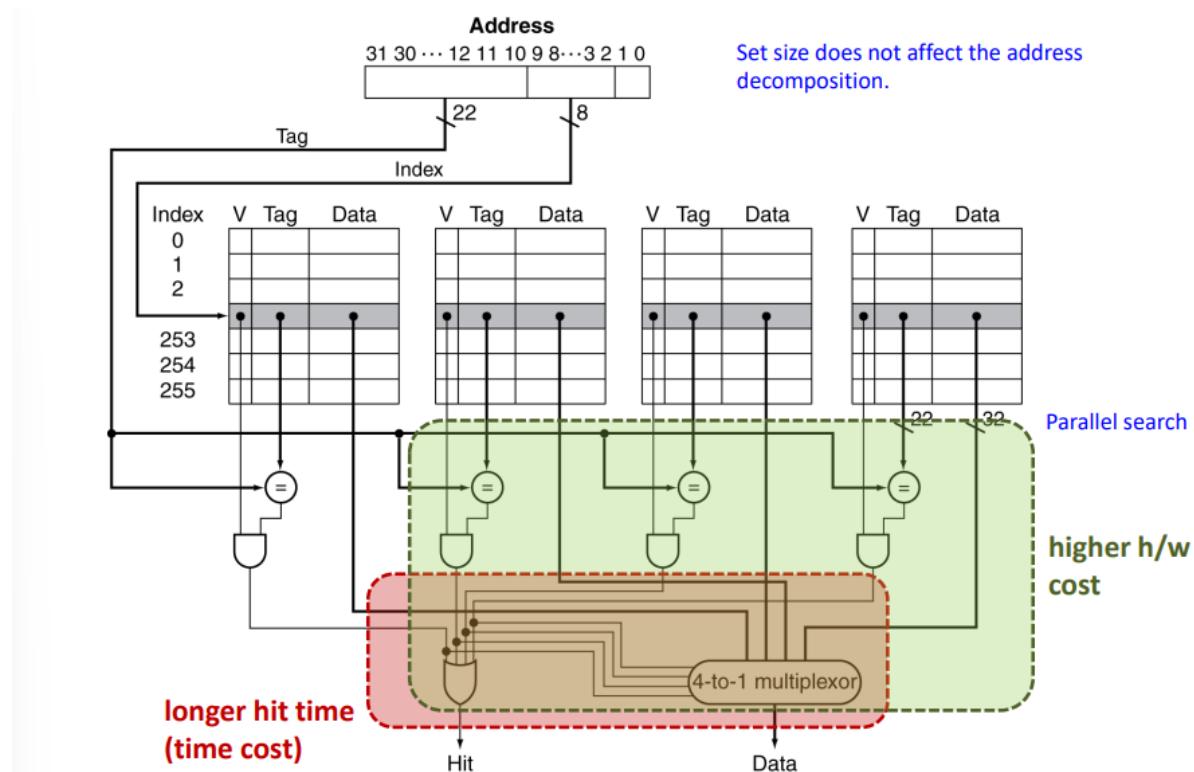
64KB data cache
16-word line
Intrinsity FastMATH
processor for SPEC
CPU2000 benchmarks

→ Associativity 가 2배씩 커질 수록 Data miss rate도 2배씩 감소 할 것이라고 생각하지만, 사실 Associativity가 커질수록 2배씩 감소 하지는 않는다.

- Disadvantage
 - Longer hit time
 - Set안에 있는 모든 태그들은 반드시 선택과 비교를 해야 한다. 그래서 Hit time 이 길다.
 - Higher H/W cost(HW 비용이 비싸다.) → 선택해 준 범위 내에서만 search를 진행하기 때문에, 선택의 범위를 더 늘리기 위해, HW를 더 추가해야 하기 때문이다.

4-way Set Associative Cache Schematic

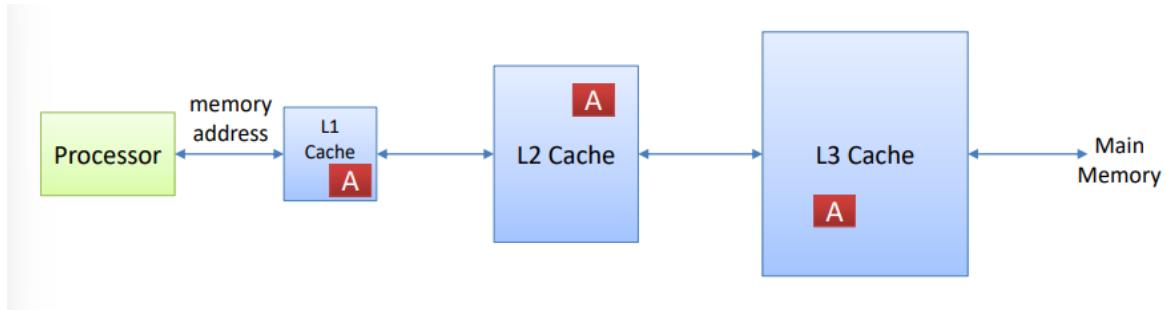
- search 자체는 Hit time을 늘리지 않는다.
- Fully associative → Hit time 의 영향을 가장 많이 받는다.
- Direct associative → 공간 효율이 안 좋다.
- 이것의 절충안으로 Set-Associative 가 있다.
- 계산은 병렬적으로 일어나기 때문에, 한 개를 계산하는 것과 똑같다



- 위에서 했던 설명을 그림으로 보여준건데, Longer hit time(time cost) 이건, 위에선 data가 단일로 있었지만, 여기선 4-way로, Tag의 0/1연산을 한 후 나온 값을 Index(valid)와 AND 연산 후에, 이렇게 나온 총 4개의 값을 OR 연산을 통해, Hit인지 Miss인지 결정한다. → longer hit time
- Higher h/w cost : 소자가 4개가 들어가며, 그에 따라 연산 처리는 더욱 복잡해지며, 필요한 HW 부품의 개수도 증가한다.

Multi-level Cache

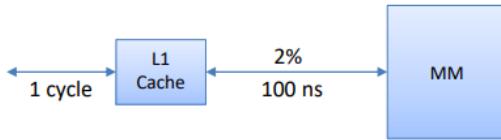
- Cache 들의 기본 소자는 SRAM으로 만들어 진다.
- L1 cache : 속도가 가장 중요하다. CPU에 붙어있기에 **Hit time이 좋아야 한다.** 여기엔 **Direct associative**로 하는 것이 좋다. 크기도 작기 때문이다.
- L3 cache : Hit time이 조금 느려도 괜찮다. Data가 이쪽으론 많이 안 오고, L1, L2 에서 걸러진게 나온다. 대신 MM에 안 가기 위해 **Miss rate를 줄여야 한다.** **Fully associative**를 써야한다. 또한 가로, 세로 크기도 잘 조절해야 한다.
- Modern processors 는 1레벨 이상의 cache를 가지고 있다. 즉, 여러 단계로 cache를 구성하고 있다. L1, L2, cache와 Shared cache인 L3이다.
- L1 : small, fast
- L2: larger, slower
 - 높은 associativity, 큰 블럭 사이즈
 - L1 보다는 Access time에 대한 critical이 적다.
- L1 cache miss → Served by L2 cache
- L2 cache miss → served by L3 cache



Multilevel Cache performance → 연습 문제

Multilevel Cache Performance

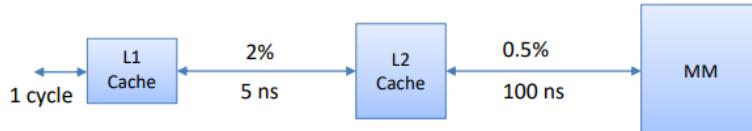
■ Performance with L1 Cache only



Base CPI	1.0
Clock rate	4 Ghz
L1 Miss rate	2%
Main memory access latency	100 ns

- i) Avg CPI = 1 cycle x 1.0 + 400 cycles x 0.02 = 9
- ii) Avg CPI = 1 cycle x 0.98 + (400+1) cycles x 0.02 = 0.98 + 8.02 = 9

■ Performance with L2 cache



- i) Avg CPI = 1 cycle x 1.0 + 20 cycles x 0.02 + 400 cycles x 0.005 = 3.4
- ii) Avg CPI = 1 cycle x 0.98 + (20+1) cycles x 0.015 + (400+20+1) cycles x 0.005 = 3.4

■ Performance comparison

- $\frac{9}{3.4} = 2.6$



이건 시험 반드시 출제 반드시 안 되면 무조건 암기라도 하고 시험장 들어가자