

메모리 시스템 개요

1. 메모리 특성

- 메모리 시스템 특성에 따라 메모리 시스템을 분류할 수 있다.
 - 내부 메모리, 외부 메모리에 따라 특성이 다름.
- 메모리 위치
 - 내부 메모리 : 레지스터, 메인 메모리, 캐시
 - 외부 메모리 : 주변 메모리 장치(디스크, 테이프), 컨트롤러를 통해 CPU와 통신
- 메모리 용량
 - 내부 메모리 : 바이트나 워드 단위로 표시
 - 외부 메모리 : 바이트 단위로 표시
- 전송 단위
 - 내부 메모리 : 워드 단위
 - 외부 메모리 : 블록(512B, 1KB) 단위

1. 메모리 특성

• 접근 방법

- 순차접근(Sequential) : 테이프, 접근시간 가변
- 직접접근(Direct) : 디스크, 접근시간 가변
- 임의접근(Random) : 메인 메모리, 캐시, 접근시간 일정
- 연관접근(Associative) : 캐시

• 연관 접근 방식

- 기억장소에 데이터와 함께, 키(Key)값을 같이 저장.
- 키값을 비교하여 기억장소를 식별
- 장점 : 기억장소 활용도 증가
 - 메모리 공간중 필요한 장소만 사용할 수 있기 때문에, 기억장소 활용도 증가
- 단점 : 키 저장공간 및 비교시간 부담

1. 메모리 특성

• 메모리 성능

- 접근 시간(Access Time, Latency) : 메모리 읽기/쓰기 작업에 소요되는 시간.
- 메모리 사이클 시간(Memory Cycle Time)
 - 반복적 메모리 접근시, 연속적 접근이 가능한 최소 시간.
 - 접근시간 + 두번째 메모리 접근 준비시간
- 전송률(Transfer Rate)
 - 메모리 시스템에 데이터를 전송할 수 있는 속도
 - 임의접근 메모리 경우, $1/(\text{cycle time})$

1. 메모리 특성

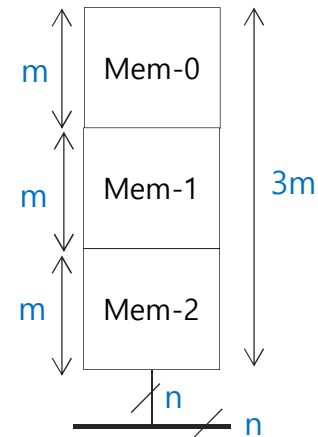
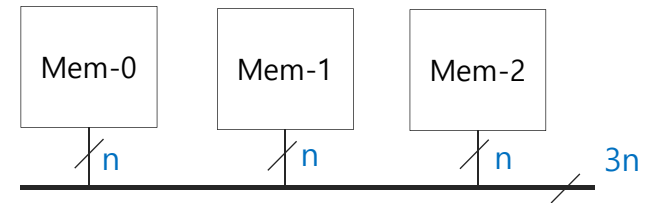
- 물리적 특성

- 휘발성/비휘발성(Volatility) : 전원공급 중단시 저장한 데이터 훼손 여부
- 가변성(Erasability) : 저장된 데이터에 대한 삭제 가능 여부
- 파괴성(Destructiveness)
 - 데이터 접근시 저장된 데이터 파괴 여부.
 - 마그네틱 코어를 제외한 대부분 메모리 비파괴성.

1. 메모리 특성

• 메모리 조직

- 워드를 구성하기 위한 메모리 비트의 배치형태
- Word Expansion : 워드길이 확장, 데이터 폭 확장, 여러 개의 메모리를 병렬접속
- Depth Expansion : 주소길이 확장, 용량(Capacity) 확장, 여러 개의 메모리를 직렬접속



1. 메모리 특성

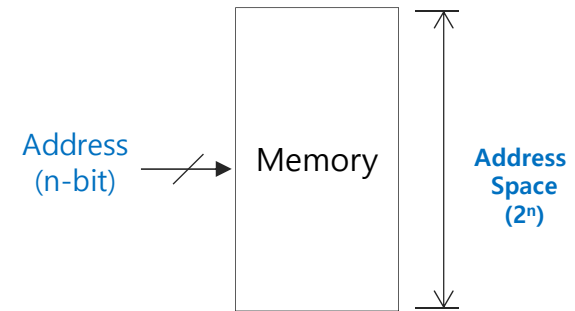
- 메모리 재충전(Refresh)

- DRAM 경우, 데이터를 저장하고 있는 메모리 셀은 시간이 지남에 따라 저장된 전하가 누설
 - 주기적 재충전이 필요
 - 모든 메모리 셀은 정해진 주기내에 반드시 재충전되어야 한다.
- 재충전 빈도와 방법은 반도체 제조기술과 메모리 셀 설계기술에 따라 다름.
- 재충전 작업으로 인한 대역폭 손실 발생

2. 주소 디코딩

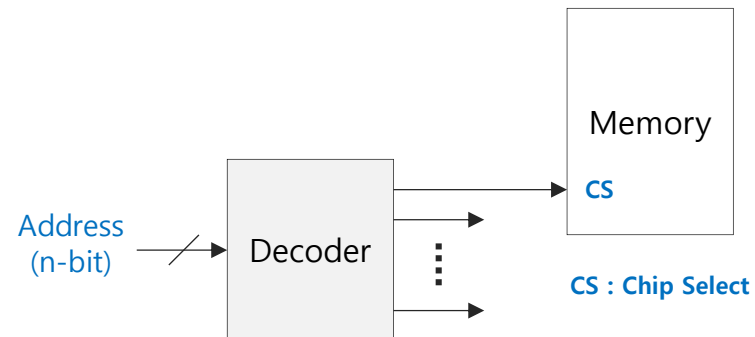
- 주소 공간(Address Space)

- n-bit 어드레스로 접근 가능한 메모리 공간 : 2^n



- 주소 디코딩(Address Decoding)

- 주소 전체 또는 일부를 해석하여 제어신호를 생성하는 작업
 - 제어신호는 물리 메모리의 활성화(Enable) 신호로 활용
 - CS (Chip Select)



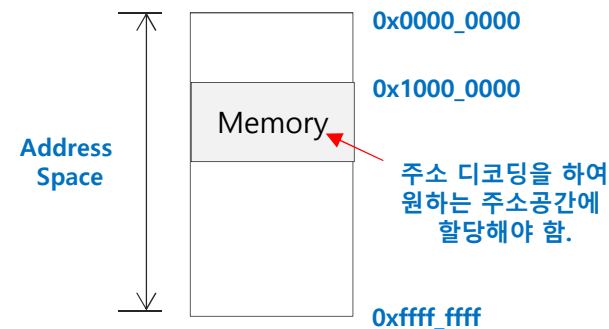
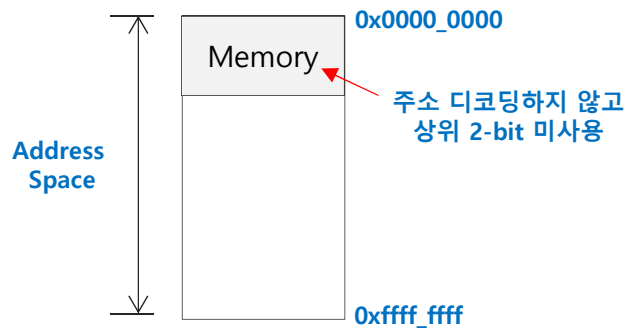
2. 주소 디코딩

• 주소 디코딩이 필요한 경우

- 비연속적 주소공간에 물리 메모리를 할당할 경우
- 물리 메모리의 주소공간을 변경할 경우.
- 특정 주소공간에 물리 메모리를 할당할 경우.

• 주소 디코딩 예제

- 32-bit 주소를 출력하는 CPU에 1GB 물리 메모리를 연결하는 경우,
 - 32-bit 주소 : 4GB 주소 공간 접근 가능.
 - 1GB 메모리 공간 : 30-bit 주소를 사용하여 메모리 셀에 접근.



3. 주소 버스 다중화

- **DRAM 메모리 동향**

- 메모리 용량 증가 → 주소공간 확장 → 주소 핀(Pin) 증가
- 메모리 워드 길이 증가 → 데이터 핀 증가
- 메모리 입출력 핀 수 증가 → 패키지 크기 증가

- **일반적으로 주소 버스 폭 > 데이터 버스 폭**

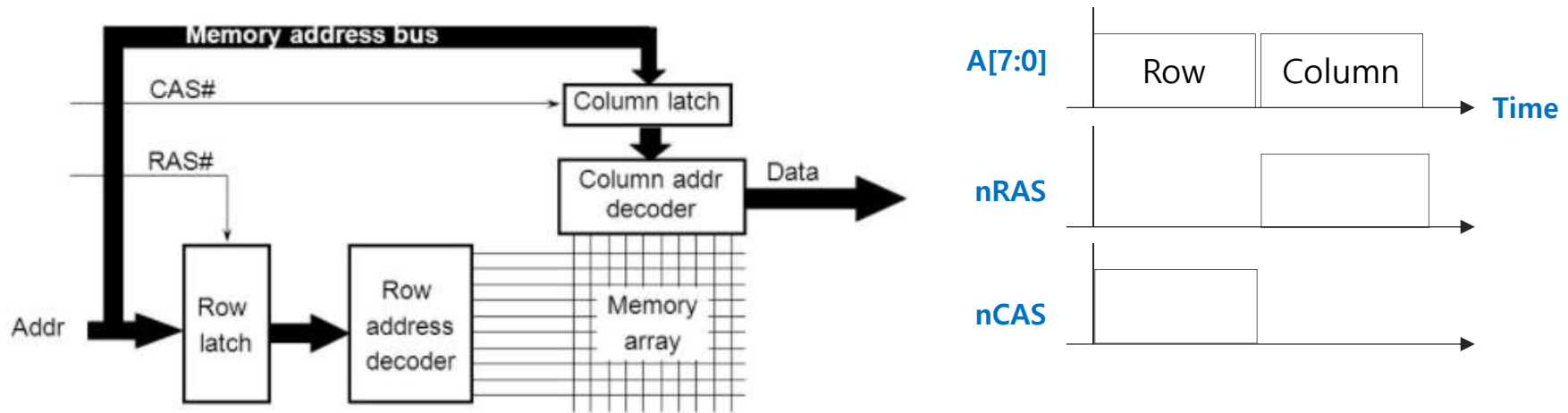
- **주소 버스 다중화(Address Bus Multiplexing)**

- 시간적 다중화(Time-Multiplexing)
 - 주소 버스를 분할해서 사용 : Row address, Column address
 - 주소 버스의 일부와 데이터 버스를 공동으로 사용

3. 주소 버스 다중화

• 주소 버스 분할 다중화 구성

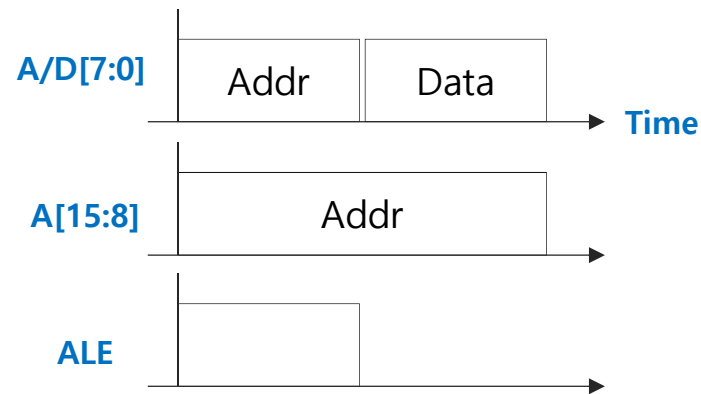
- 메모리 셀이 2차원 배열로 구성된 경우, Row 주소와 Column 주소를 분리함으로써 주소 버스 폭을 절약.
- 주소 버스 제어신호 필요
 - RAS(Row Address Strobe) : 주소 버스에 Row 주소가 출력될 때 '0'로 설정.
 - CAS(Row Address Strobe) : 주소 버스에 Column 주소가 출력될 때 '0'로 설정.



3. 주소 버스 다중화

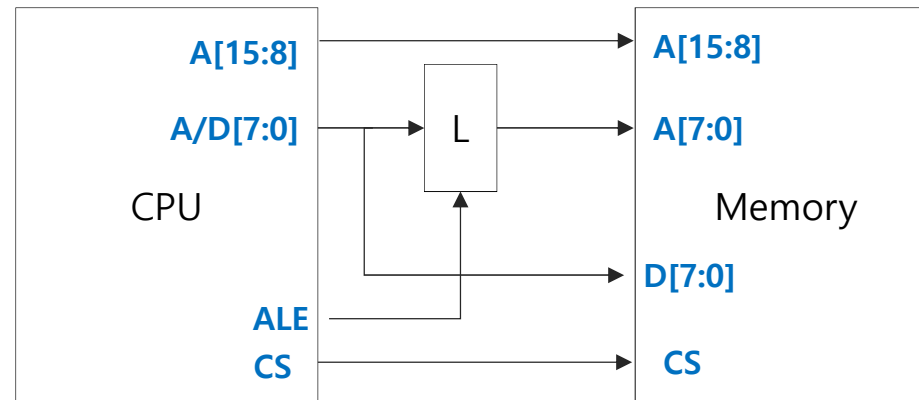
- 주소 버스와 데이터 버스 다중화 구성

- 데이터 버스 : 주소 하위 바이트와 데이터가 공동으로 사용. A/D[7:0]
- 주소 버스 : 주소 상위 바이트 단독 사용. A[15:8]
- 제어 비트 : 데이터 버스상에 출력되는 데이터의 유형을 표시.
 - ALE(Address Latch Enable) : '0' 데이터 출력, '1' 주소 출력



3. 주소 버스 다중화

- 주소/데이터 다중화 주소를 사용할 경우, 외부 메모리 연결할 때 래치(Latch) 필요
 - 하위 바이트 주소를 저장하기 위해 래치를 사용
 - 래치 활성화 신호 : ALE



4. 메모리 내부 조직

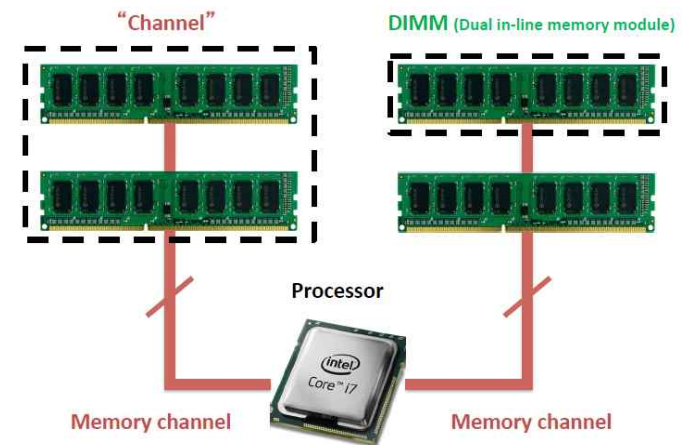
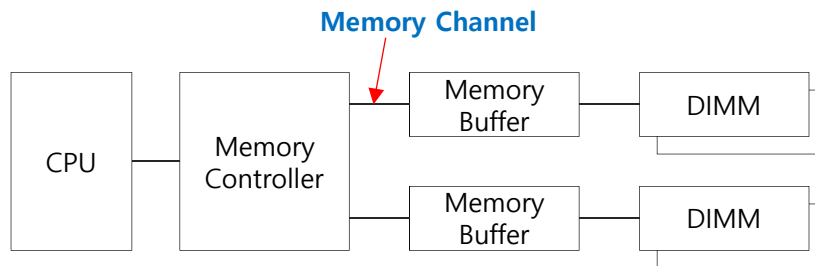
- **메모리 내부 조직(Memory Subsystem Organization)**

- Channel : CPU와 메모리 사이의 데이터 경로
 - Memory Controller : 메모리 제어 신호 발생
 - Memory buffer : 버퍼를 사용해서 메모리 데이터 전송속도를 향상
- SIMM/DIMM(Single/Dual In-line Memory Module) : 메모리 모듈 구조
- Rank : 메모리 칩 배열 구조
- Chip : 물리적 메모리 칩
- Bank : 메모리 배열 구조
- Row/Column : 메모리 셀 배치 구조

4.1 메모리 채널

- 메모리 채널(Memory Channel)

- 메모리에서 I/O 발생시 동일한 메모리 컨트롤러에 연결되며, 하나의 데이터 경로를 공유.
- CPU와 메모리 사이의 데이터 경로
- CPU와 메모리 사이에 메모리 컨트롤러, 메모리 버퍼를 사용



4.2 메모리 랭크

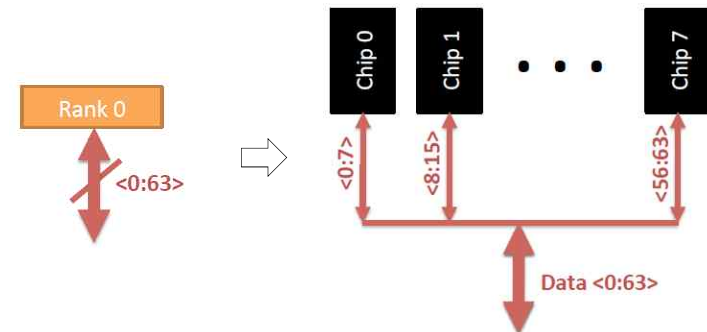
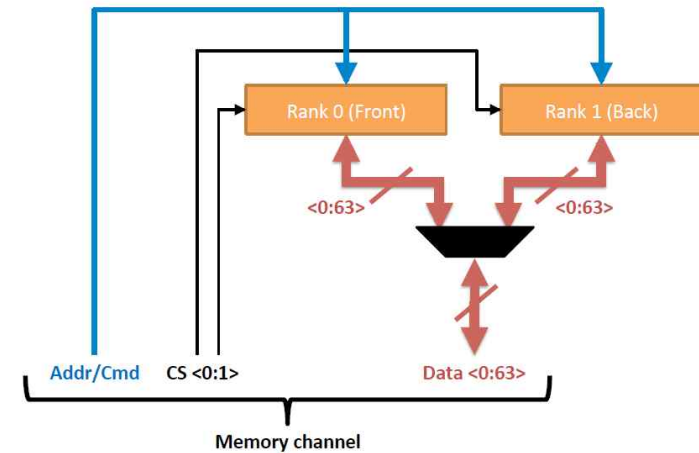
- 메모리 랭크(Memory Rank)

- 동일한 CS(Chip Select) 신호 선을 공유하는 여러 개의 DRAM 집합.
- 일반적으로 64-bit 폭
 - ECC(Error Correction Cod)를 포함하는 경우, 72-bit
 - Single-Rank(1 x 64-bit), Dual-Rank(2 x 64-bit), Quad-Rank(4 x 64-bit)

- 메모리 산업계 표준 그룹(JEDEC)에서 정의한 용어.

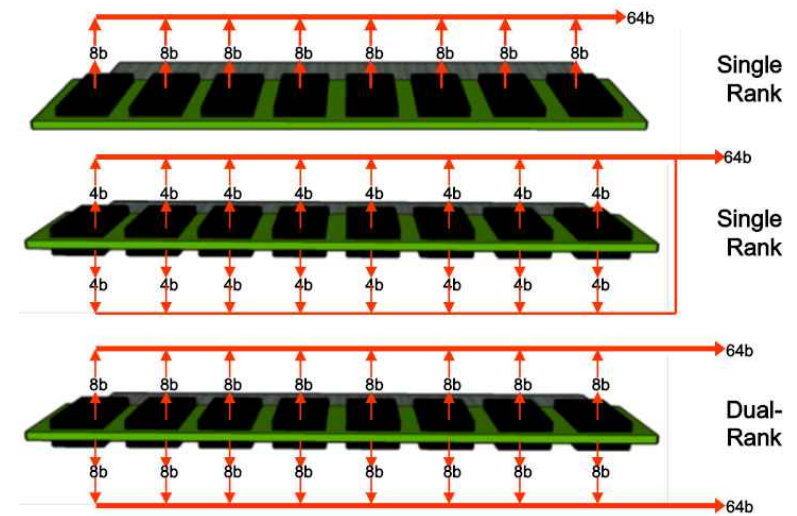
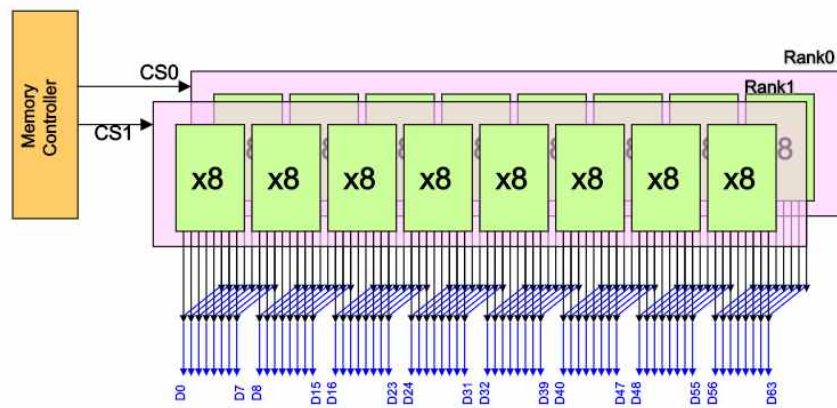
- 메모리 모듈에서의 메모리 블록.

- 메모리 랭크에 속한 메모리는 동시에 접근 가능.





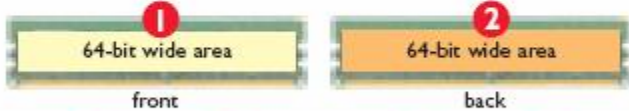

4.2 메모리 랭크

- 메모리 랭크와 메모리 모듈 연결.



4.2 메모리 랭크

- 랭크 구성 예

Single-sided Module	 <p>64-bit wide data area front</p>	1 rank Single-rank
Double-sided Module	 <p>64-bit wide data area front back</p>	1 rank Single-rank
Double-sided Module	 <p>64-bit wide area 64-bit wide area front back</p>	2 ranks Dual-rank
Double-sided Module	 <p>64-bit wide area 64-bit wide area 64-bit wide area 64-bit wide area front back</p>	4 ranks Quad-rank

4.2 메모리 랭크

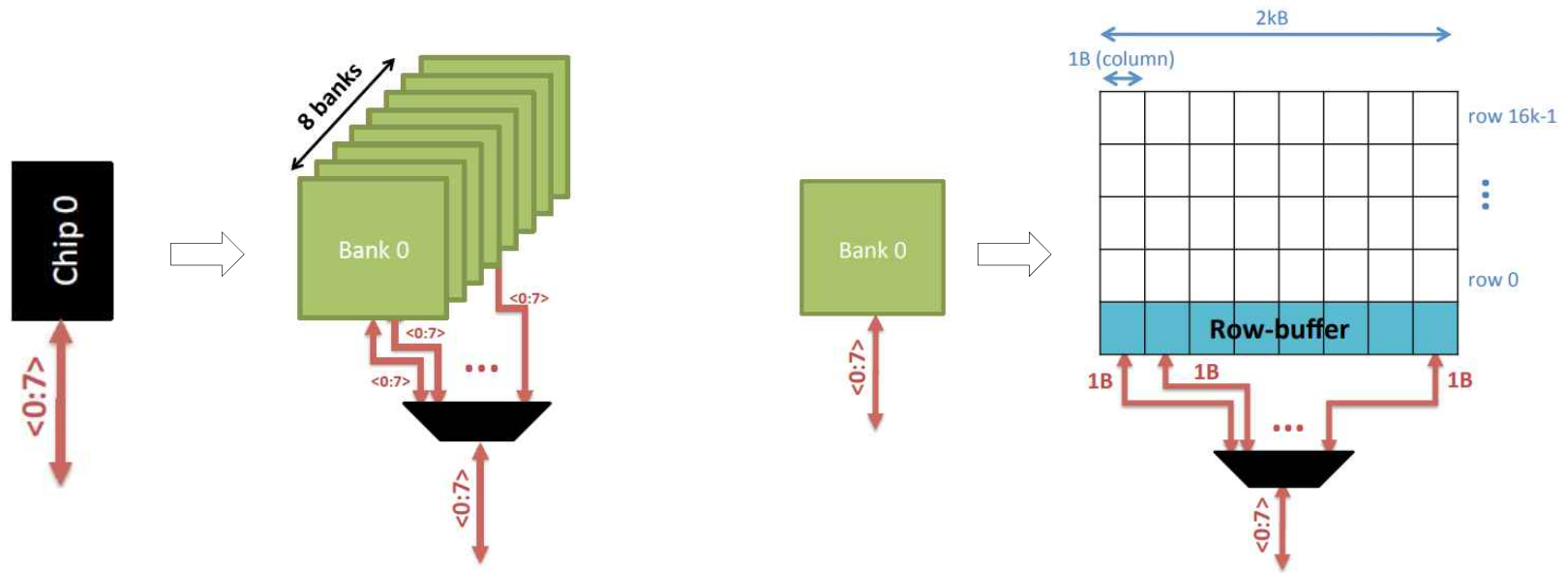
- 메모리 모듈에서의 랭크 표기
 - 1R (Single rank), 2R(Double-rank), 4R(Quad-rank)



1Rx8 : 1 Rank, 메모리 칩의 대역폭(x8 = 8-bit)

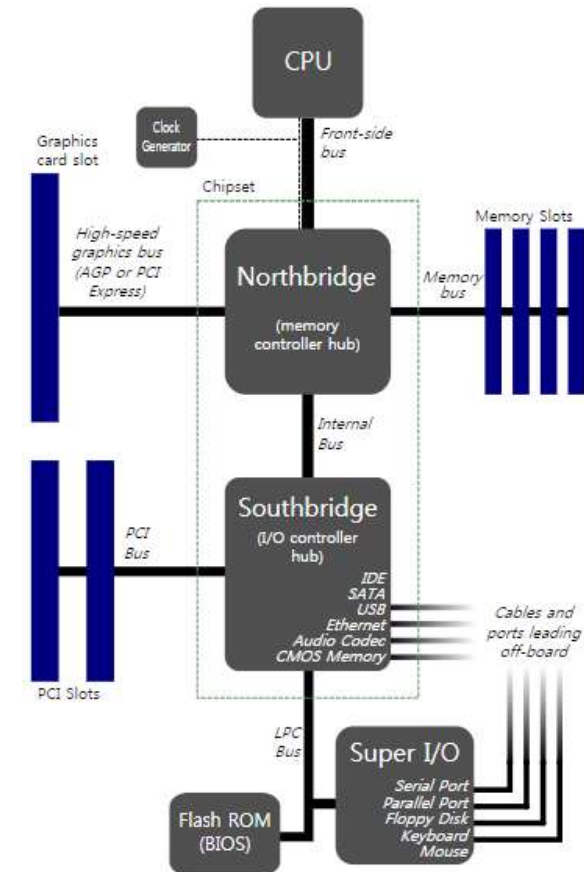
4.3 메모리 뱅크

- 메모리 뱅크(Memory Bank)
 - 메모리 칩에서의 메모리 블록



4. PC 메인보드의 메모리 구조

- PC 메인보드(Main Board, Mother Board)상에서 메모리 구성
 - CPU와 메모리를 Northbridge 를 통해 연결
 - Northbridge : CPU와 고속 버스를 통해 직접 연결되는 칩, MCH(Memory Controller Hub)
 - RAM, PCI Express, AGP, Southbridge 등과 같은 고속 장치들을 FSB를 통해 CPU에 연결
- Southbridge
 - 입출력 장치들을 CPU에 연결에 연결해주는 칩, I/O Controller Hub(ICH)
 - Northbridge를 통해 CPU와 저속장치를 연결



메모리 계층구조

1. 메모리 시스템 설계 요소

- 메모리 시스템 설계 요소

- 용량 : 용량은 커질수록 Good.
- 접근속도 : 빠를수록 Good.
- 비용 : 작을수록 Good

- 메모리 설계자의 고민

- 접근 속도가 빠를수록 비용이 상승
- 용량이 증가할수록 속도는 저하

- 해결방법

- 메모리 특성 차이를 고려하여 메모리를 계층적(Hierarchical)으로 배치하여 최적의 메모리 시스템을 설계
 - 평균적 메모리 접근 속도를 높이면서 가격대 성능비도 적절하게 유지하도록 설계

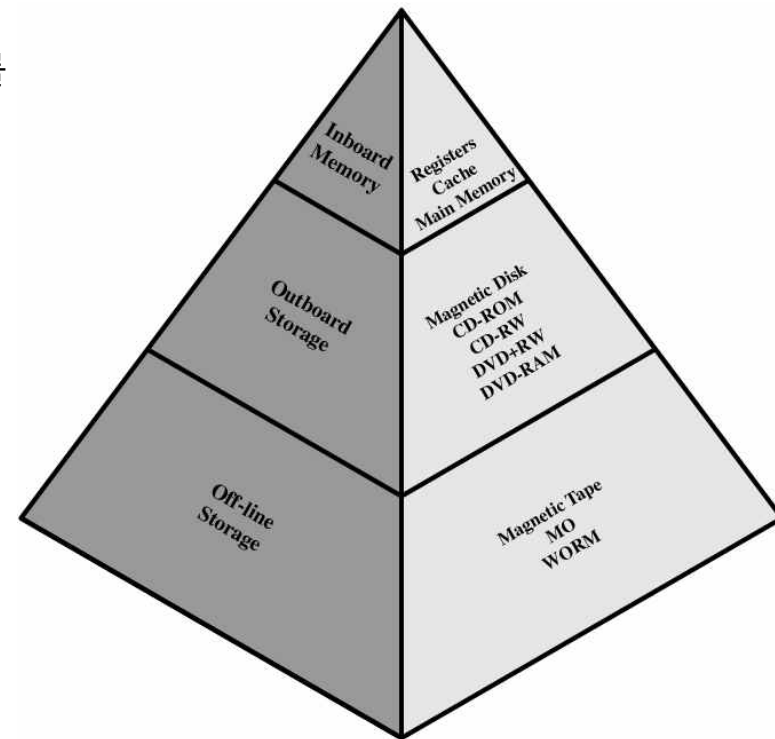
2. 메모리 계층구조

- 메모리 계층구조 (Hierarchy)

- 서로 다른 접근 속도와 용량을 갖는 메모리를 계층적으로 배치한 구조

- 메모리 계층구조 리스트

- Registers
- L1 Cache
- L2 Cache
- Main memory
- Disk cache
- Disk
- Optical
- Tape

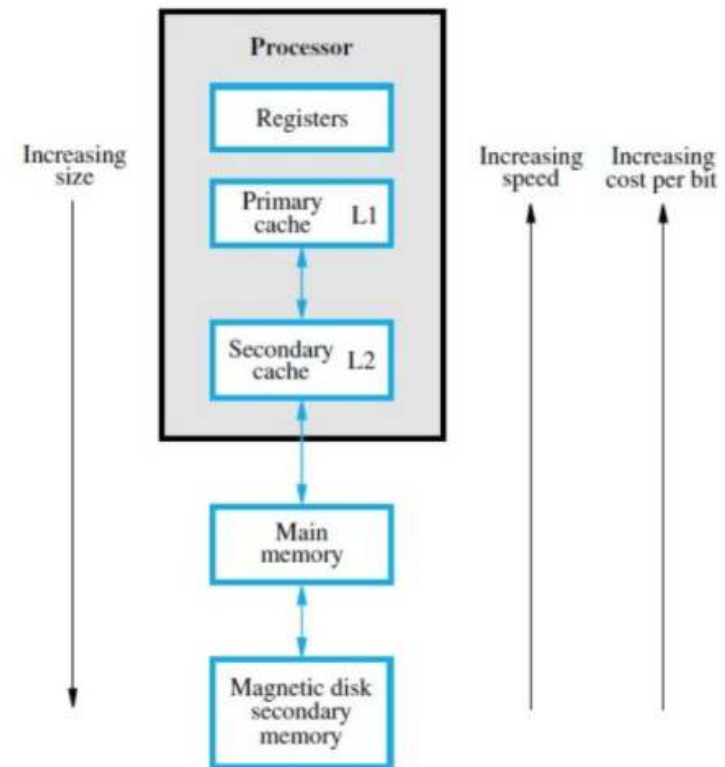


WORM (write once, read many)

3. 메모리 계층구조 특성

- 메모리 계층구조 (Hierarchy) 특성

- 하위 계층으로 내려갈수록
 - 비트당 비용(cost) 감소
 - 용량 (capacity) 증가
 - 접근시간 (access time) 증가
 - CPU의 메모리 접근 빈도 감소



3. 메모리 계층구조 특성

- 상위계층은 하위계층의 일부를 복사한 복사본
- 상위계층과 하위계층간의 데이터의 일관성(Consistency) 유지가 중요.
- 일관성 유지비용이 클 경우, 계층구조가 오히려 더 비효율적 임.
- 계층간의 데이터 전송 단위와 메모리 용량이 계층구조 성능에 중요한 결정요소.

4. 메모리 계층구조 동작 특성

- 메모리 계층구조에서의 데이터 전송단위
 - 워드 : CPU 레지스터와 캐시 메모리 사이
 - 블록 : 캐시 메모리와 메인 메모리 사이
 - 메모리상에서 연속된 어드레스에 저장된 데이터 집합.
 - 페이지 : 메인 메모리와 보조 기억 장치 사이
 - 운영체제에서 관리.
 - 하드웨어를 사용하여 성능향상 가능.

4. 메모리 계층구조 동작 특성

- **적중(Hit)**
 - CPU가 요청한 데이터가 상위 계층에 존재할 때
- **실패(Miss)**
 - CPU가 요청한 데이터가 상위 계층 메모리에 없을 때
 - 요청한 데이터를 하위 계층 메모리로 부터 획득.
- **적중율(Hit Ratio)**
 - $(\text{Hits 횟수}) / (\text{전체 데이터 요청 횟수})$
 - 메모리 계층구조의 성능평가 기준
- **실패율(Miss Ratio)**
 - $(\text{Miss 횟수}) / (\text{전체 데이터 요청 횟수})$
 - $1 - \text{Hit Ratio}$

4. 메모리 계층구조 동작 특성

- **평균 메모리 액세스 시간 (T_a)**

- $T_a = H \times T_c + (1 - H) \times (T_m + T_c)$
 - T_c = 캐시 접근 시간
 - T_m = 메모리 접근 시간
- $T_m \gg T_c$ 인 경우, $T_a \approx H \times T_c + (1 - H) \times T_m$

- **Hit Time**

- Hit/Miss 판단 시간 + 상위 계층 메모리에 접근하는 시간

- **Miss Penalty**

- 상위 계층 메모리의 블록을 하위 계층 메모리로 부터 읽어서 교체하는 시간 + CPU에 해당 블록을 전달하는 시간

5. 참조의 지역성

• 참조의 지역성 (Locality of Reference)

- 기존 프로그램들의 실행패턴을 분석한 결과 프로그램 실행과정에서 메모리 참조는 근거리에서 이루어진다.
- 메모리 액세스가 클러스터를 형성한다.
 - 프로그램 메모리 경우 : loops, subroutines
 - 데이터 메모리 경우 : clustered data (table or array)
- 지역성의 원리(Principle of Locality) 라고도 함.



5. 참조의 지역성

- **시간적 지역성(Temporal Locality)**

- 최근에 사용한 데이터가 가까운 미래에 다시 사용될 확률이 높은 경향.
 - 프로그램의 반복문 : 명령어 Loop 실행
 - 공통 변수

- **공간적 지역성(Spatial Locality)**

- 최근에 사용한 데이터의 근처에 저장된 데이터가 사용될 확률이 높은 경향.
 - 명령어의 순차적 실행
 - 데이터 배열, 테이블

- 명령어와 데이터에서 나타나는 지역성의 정도는 프로그램마다 다르다.

- 참조의 지역성은 메모리를 계층적으로 이용하는 근거가 된다.

6. 계층적 메모리 시스템 설계요소

- 배치방법(Mapping)

- 하위계층에서 상위계층으로 데이터를 전송할 때 상위계층의 위치를 결정하는 방법
- 사상(Mapping)이라고도 함.

- 교체방법(Replacement)

- 하위계층에서 상위계층으로 데이터를 전송할 때 상위계층에 저장할 공간이 없는 경우, 상위계층의 공간 마련을 위해 축출할 대상을 선정하는 방법.

- 쓰기 방법

- 상위계층의 데이터를 변경할 경우, 하위계층의 원본을 변경하는 시점을 결정하는 방법.

6. 계층적 메모리 시스템 설계요소

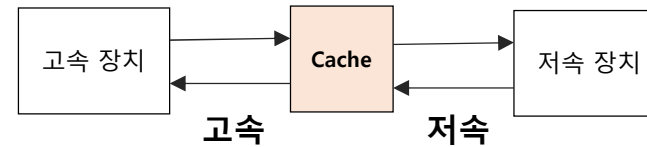
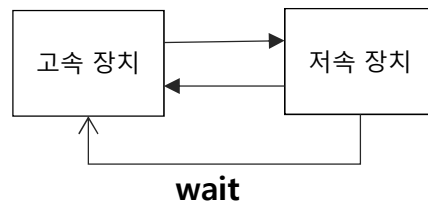
- 상위 계층 메모리의 적중율(Hit Ratio) 향상
 - CPU가 요청하는 데이터가 상위 계층 메모리에 존재 할 확률을 높여야 한다.
- 상위 계층 메모리의 접근 시간 최소화
 - CPU가 요청하는 데이터를 상위 계층에서 읽어 오는데 걸리는 시간을 최소화시켜야 한다.
- 상위 계층 메모리에서 Miss 발생시 시간 지연 최소화
 - 상위 계층에서 Miss 발생한 경우, 하위 계층 메모리로 부터 데이터를 읽어오는데 걸리는 시간을 최소화시켜야 한다.
- 상위 계층과 하위 계층의 데이터 일관성 유지 및 그에 따른 오버헤드 최소화
 - 상위 계층의 데이터를 변경했을 경우, 동일한 위치의 하위 계층의 데이터도 동일하게 변경해야 하며, 그 때 필요한 작업에 걸리는 시간을 최소화시켜야 한다.

Cache 개요

1. Cache 기초

- Cache 란 ?

- 일반적으로 고속 데이터 접근이 가능한 공간
 - 데이터 처리 속도차이가 있는 장치들을 연결하여 사용할 때 속도차이를 극복하는 효과적인 방법.



1.1 Cache 유형

- **Memory Cache**

- CPU와 메인 메모리 사이의 속도 차이 극복

- **Disk Cache**

- 메인 메모리와 하드 디스크 사이의 속도 차이 극복
- 메인 메모리 HE는 디스크 컨트롤러에 위치

- **응용 프로그램/서비스에서 사용하는 Cache**

- 하드 디스크 접근 속도와 네트워크 지연의 속도 차이 극복
- 인터넷 브라우저의 Cache
 - 자주 사용하는 페이지를 서버로 부터 다시 다운로드 하지 않고, cache에 저장된 페이지를 reload.
- Cache server
 - 네트워크 트래픽 부담을 줄이기 위해 사용.

1.2 메모리 Cache

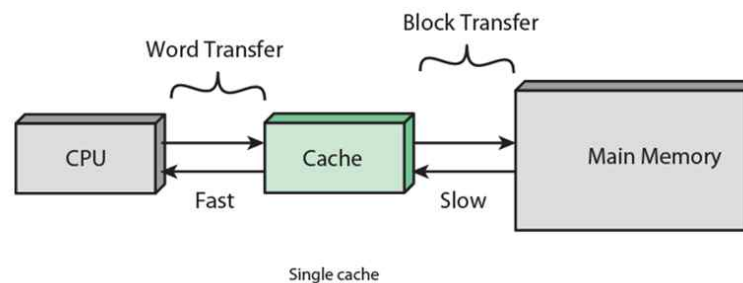
- **메모리 Cache**

- 소용량 고속 메모리를 사용하여 CPU 와 메인 메모리의 속도차이를 해결.
- 메모리 시스템 기능과는 무관하고 성능에만 영향

- **일반적으로 메인 메모리와 CPU 사이에 위치**

- 메인 메모리와 Cache 사이의 데이터 전송 : 블록단위 전송
- Cache 와 CPU 사이의 데이터 전송 : 워드단위 전송

- **CPU 칩 내부에 집적되거나, 보드상에 실장됨.**

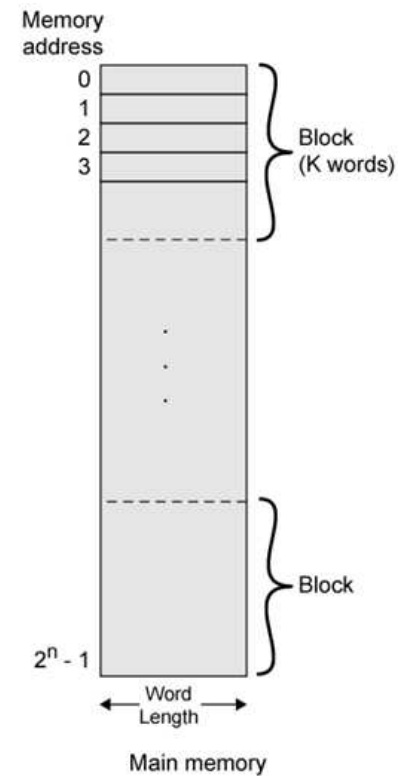
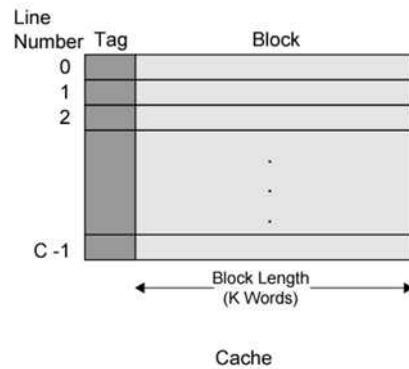


1.2 메모리 Cache

- Cache 성능향상 성공여부는 참조의 지역성에 의존.
- Cache 메모리는 하드웨어적으로 구현되고 프로그래머가 제어할 수 있는 부분은 제한적.
 - Cache 사용여부 제어가능
 - 메인 메모리 영역에 따라 Cache를 사용할지 여부 제어
 - Cacheable/Non-cacheable

2. Cache 구성

- Cache는 라인단위로 관리되는 고속 메모리
- Cache 라인(Line)
 - 메모리 블록, 태그(Tag), 제어비트로 구성.
 - Line size
 - 태그와 제어비트를 제외한 길이
 - 메인 메모리 블록의 크기



2. Cache 구성

- **태그(Tag)**

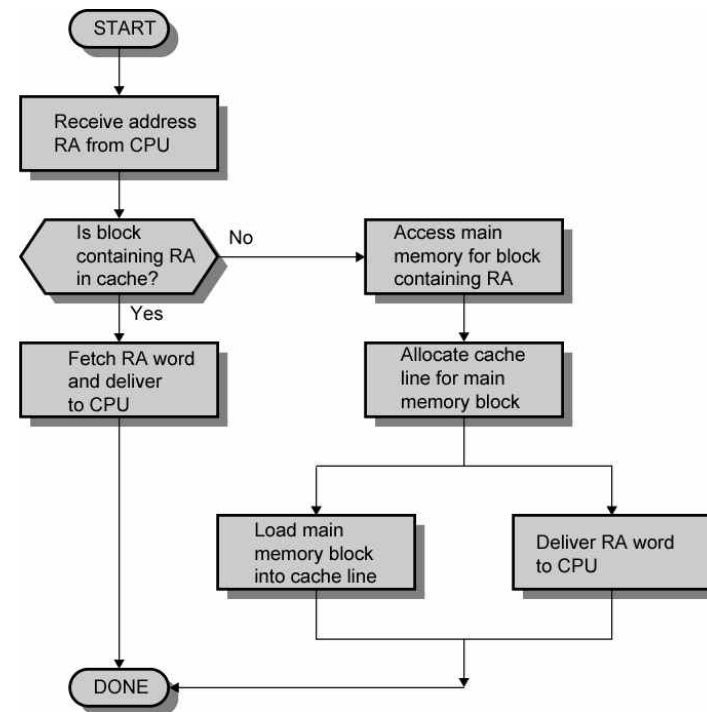
- 현재 저장된 블록에 대한 식별자
- 메인 메모리 어드레스의 일부를 태그로 사용.

- **제어 비트(Control Bit)**

- Cache 관리에 필요한 제어 비트들
- Dirty bit
 - Cache 라인에 포함된 데이터가 Cache에 로딩된 후 변경되었는지 여부를 표시
 - CPU가 라인을 수정했는지 여부를 표시.
- Valid bit
 - Cache 라인에 유효한 데이터 포함 여부를 표시.
 - 부팅시 모두 무효로 설정.

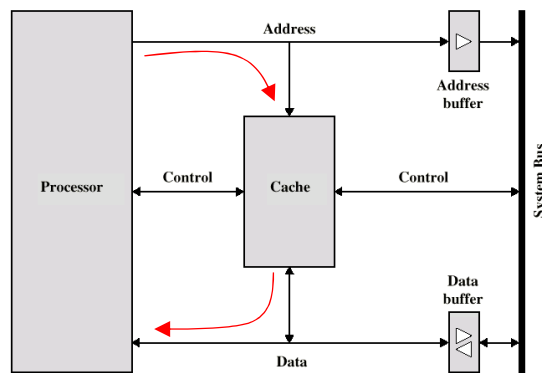
3. Cache 기본동작

- CPU 가 메모리에 저장된 데이터 요청
- 요청한 데이터가 Cache에 있는지 여부 검사
- Cache에 있는 경우(Cache Hit), Cache로부터 데이터 읽기 (고속)
- Cache에 없는 경우(Cache Miss), 메인 메모리로 부터 해당 데이터를 Cache로 복사.
- 복사한 데이터를 CPU 에 전달.

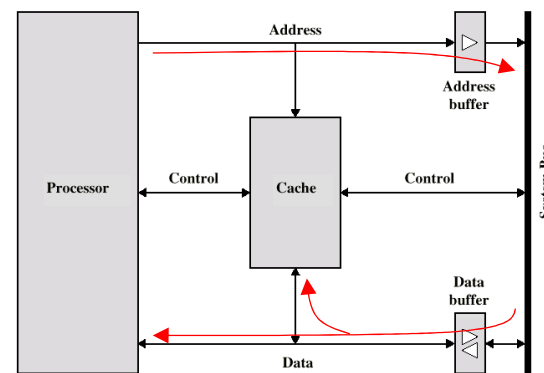


4. Cache Hit/Miss

- CPU가 명령어와 데이터 인출시 먼저 Cache 메모리에 접근해서 원하는 명령어나 데이터 존재 여부를 검사.
 - 원하는 데이터가 Cache에 저장되어 있는 경우 → Cache Hit
 - 원하는 데이터가 Cache에 저장되어 있지 않는 경우 → Cache Miss
 - 메인 메모리로 부터 데이터 로딩.



Cache Hit



Cache Miss

4.1 Cache Miss 처리방법

- **데이터 Cache Miss 처리**

- CPU 정지. 레지스터 값 고정(Freezing)
- 데이터 메모리에서 데이터를 인출 (별도 컨트롤러 사용)
- CPU 재가동.

- **명령어 Cache Miss 처리**

- 현재 PC 값을 메모리에 저장
- 메모리로 부터 명령어를 읽어올 때까지 대기.
- 읽어온 데이터를 Cache 엔트리에 등록 (Tag, Valid bit 처리)
- PC값을 복구하여 명령어 실행을 재개.

- **Write-on-Use 기능**

- Cache miss 패널티(처리시간 및 추가 작업)를 줄이기 위해서, Cache Miss 처리시간동안 다른 명령어를 계속 실행하는 것.
- 데이터 캐시에만 적용, 명령어 캐시에 적용 불가

Cache 설계 요소

1. Cache 설계 목표

- Cache 설계 목표

- Cache 적중율의 극대화
 - 원하는 데이터가 Cache에 존재할 확률을 높여야 한다.
- Cache 접근시간의 최소화
 - Cache 적중시 Cache에서 데이터를 인출하는데 걸리는 시간을 최소화해야 한다.
- Cache 실패에 따른 지연시간의 최소화
 - Cache 실패시 메인 메모리로부터 데이터를 인출하는데 걸리는 시간을 최소화해야 한다.
- 메인 메모리와 Cache 사이의 일관성 유지 및 오버헤드 최소화
 - CPU가 Cache 내용을 변경했을 때, 메인 메모리에 반영하는 절차에 소요되는 시간이 최소화되어야 한다.

2. Cache 설계 요소

- Cache 설계시 고려해야 할 점은 다음과 같다.
 - Cache 크기 (Cache Size)
 - 라인 크기 (Line Size)
 - Cache 수 (Number of Caches)
 - Cache 주소 (Cache Address)
 - 사상 함수 (Mapping Function)
 - 메인 메모리 블록을 Cache 라인에 할당하는 방법
 - 교체 알고리즘 (Replacement Algorithm)
 - Cache 라인을 교체하는 방법
 - 쓰기 정책 (Write Policy)
 - Cache 라인을 메인 메모리에 써넣는 방법

3. Cache 크기

- **Cache 크기에 따른 비용 및 성능**

- Cache 크기가 커질수록 고비용.
 - Cache 주소 지정하는 데 필요한 게이트 수 증가
 - 대용량 Cache가 소 용량 Cache보다 약간 늦어지는 경향.
- Cache 크기가 커질수록 어느 정도까지는 성능향상.
 - Cache에 원하는 데이터가 있는지 여부를 검사하는데 시간 소요.

- **칩 (chip) 이나 보드 (board) 상의 여유 공간의 크기에 따라 Cache 크기를 제한.**

- **Cache 성능은 작업 부하 (workload)에 매우 민감하기 때문에 “optimum” 크기를 결정하는 것은 불가능.**

4. Line 크기

- 블록이 커질수록 Cache에 들어올 수 있는 블록의 수는 감소.
- 블록이 커질수록 원하는 단어로부터 멀리 떨어져 있는 단어들도 같이 읽혀 오며, 지역성의 원리(locality of reference)에 의해 가까운 미래에 사용될 가능성은 낮아진다.
- 블록의 크기와 적중률 (hit ratio) 관계는 특정 프로그램의 지역성 특성에 따라 달라지기 때문에 복잡하며, 최적 값을 찾아내기가 어렵다.
 - 8~64 bytes 정도가 최적에 가까운 근사값으로 알려져 있다.

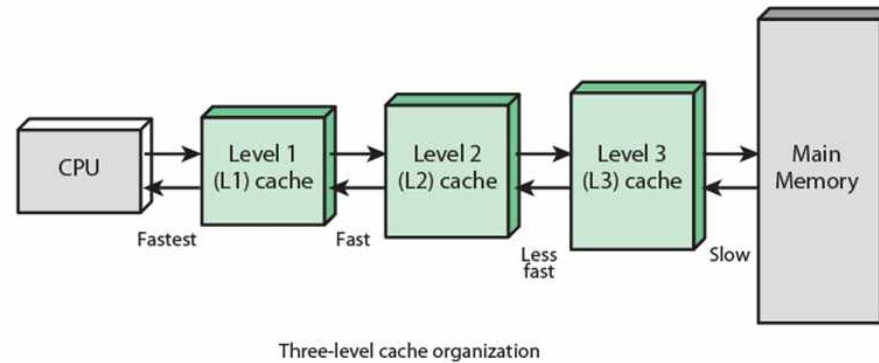
5. Cache 수

- 여러 개의 Cache를 연결하여 계층적으로 운영하는 것이 Cache 접근 속도 향상에 도움.
 - Cache 적중률과 Cache 접근 시간사이의 절충(Trade-off) 필요.
- 복수 개의 Cache 운용 방법
 - 다단계 Cache 사용
 - Cache의 계층적 구성
 - On-chip/Off-chip
 - 통합(Unified) 또는 분리(Split)
 - 명령어 저장 Cache와 데이터 저장 Cache의 분리 여부

5.1 다단계 Cache

- 다단계 Cache (Multilevel cache)

- 여러 단계의 Cache를 cascade 형태로 연결해서 사용
- 소용량 고속 Cache와 대용량 저속 Cache를 계층적으로 연결.
- Cache 접근시간을 단축.



5.1 다단계 Cache

• 다단계 Cache 구현 방법

- 온-칩 Cache (on-chip cache)
 - 프로세서와 동일 칩에 포함되어 있는 Cache.
 - 프로세서와 Cache 사이의 데이터 전송속도 향상.
 - 많은 프로세서들이 오프-칩 Cache (L2/L3 Cache)를 프로세서 칩에 내장.
- 전용 버스 사용
 - 오프-칩 (off-chip) Cache와 프로세서 사이의 전송을 위해 별도의 분리된 데이터 통로를 이용함으로써 시스템 버스의 부담을 줄임.

5.2 통합형 Cache

- **통합형 Cache (Unified Cache)**

- 명령어와 데이터를 모두 같은 Cache에 저장.

- **통합 Cache의 장점**

- 명령어와 데이터 간의 균형을 자동적으로 유지해주기 때문에, 주어진 Cache 크기에 대해 적중률이 (Hit Ratio) 가 높다.
- 하나의 Cache로 구현 가능.

5.3 분리형 Cache

- **분리형 Cache (Split Cache)**

- 명령어와 데이터를 분리해서 명령어는 명령어 Cache (Instruction Cache), 데이터는 데이터 Cache (Data Cache) 에 따로 저장.

- **분리 Cache의 장점**

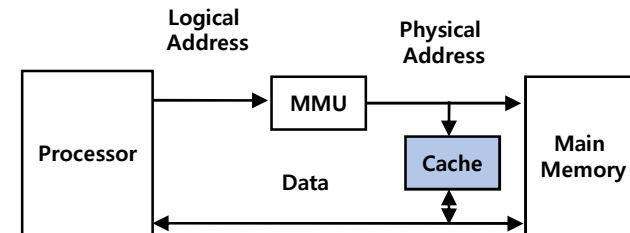
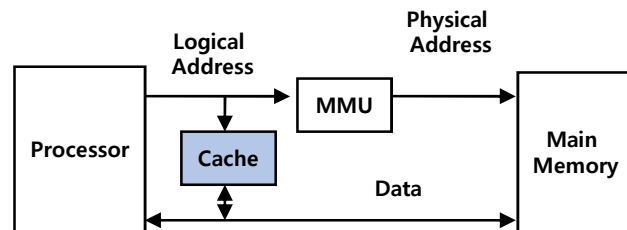
- 명령어 인출 (fetch) 유닛과 오퍼랜드 인출 유닛 사이의 Cache 경쟁 (Cache Contention) 을 피할 수 있다.
- 명령어 파이프라인에 효과적.
- Cache 메모리의 대역폭 확장 효과.

- **계층적 Cache를 사용할 경우, 최상위 계층 Cache는 분리형 캐시, 최하위 계층은 통합형 Cache로 구성 → 대역폭 확장과 적중률 향상 효과**

6. 논리적/물리적 Cache

- Cache가 사용하는 주소에 따라 구분

- 논리적(Logical) Cache
 - 논리 주소를 사용하는 Cache, 가상(Virtual) Cache
- 물리적(Physical) Cache
 - 물리 주소를 사용하는 Cache.



6. 논리적/물리적 Cache

- 논리적 Cache와 물리적 Cache 비교

- 논리적 Cache의 액세스 속도가 더 빠름.
 - MMU 가 주소변환을 수행하기 전에 응답가능
- 논리적 Cache 경우, Cache 액세스할 때 가상주소를 사용하기 때문에 각 프로그램이 사용하는 가상주소를 구분할 필요가 있다.

Cache 성능 평가

1. Cache 메모리 성능

- Cache 메모리 성능 요소

- Cache 메모리의 적중율 (Hit Ratio)
 - 요청한 데이터가 캐시에 존재할 확률 = (캐시 메모리 적중 횟수)/(캐시 메모리 참조 횟수)
- 실패율(Miss Ratio) = $1 - (\text{Hit Ratio})$
- Cache 유효 접근 시간
 - Cache 메모리에 접근할 때 실제로 필요한 시간.

2. Cache Miss 유형

- Cache 성능개선 : Cache Hit Ratio를 향상시켜 개선
- Cache Miss 유형
 - 강제 실패, 용량 실패, 충돌 실패로 구분
- Cache Miss Penalty
 - 메모리로 부터 블록을 읽어오는 시간
 - 블록의 첫 번째 워드에 접근하는 시간(Latency)
 - 읽어온 블록에 Cache에 저장하는 시간

2. Cache Miss 유형

- 강제 실패(Compulsory Miss)

- 모든 메모리 블록의 최초 접근에 의해 발생하는 실패.
- Cache 용량과 강제 실패 횟수와는 무관.
- 블록 크기를 증가시키거나, 선인출 정책(Prefetch Policy) 사용하여 강제 실패 회수를 줄일 수 있음.
- 필수 실패, Cold Miss, 최초 참조 실패(First Reference Miss)라 함.

2. Cache Miss 유형

- 용량 실패(Capacity Miss)

- Cache 메모리의 용량 한계 때문에 발생하는 실패.
- 블록이 교체된 후 나중에 다시 블록을 가져올 때 발생.
- 프로그램 크기가 Cache 용량보다 클 경우 발생.
- Cache 블록의 크기와 Miss 발생 횟수와 무관.

- 충돌 실패(Collision Miss)

- 여러 개의 블록들이 Cache의 동일한 라인에 저장될 경우 발생하는 실패.
 - Conflict Miss라고도 함.
- 메인 메모리 블록을 캐시 라인에 연결하는 사상(Mapping) 방식에 따라 충돌 실패 발생 여부가 결정.
 - Fully-associative 사상방법에서는 충돌실패 발생하지 않음.

3. Cache 유효 접근 시간

- 유효 접근 시간(Effective Access Time)

- $T_e = H \times T_c + (1-H) \times (T_c + T_p) = T_c + (1 - H) \times T_p$
 - T_e : Effective Access Time
 - T_c : Cache Access Time (Hit Time)
 - T_p : Cache Miss Penalty Time
- If $H \approx 1$, $T_e \approx T_c$

- Cache 실패 패널티 타임

- Cache 실패가 발생했을 경우 지연시간
- Cache 실패가 발생했을 경우, 메인 메모리에서 블록을 가져오는데 걸리는 시간.
- $T_p = T_m + T_t$
 - T_m : 메인 메모리 접근시간
 - T_t : 블록 전송 시간, Block Transfer Time

4. Cache 성능개선

- Cache 성능개선

- 유효 접근 시간을 줄이는 것
- 유효 접근 시간에 영향을 주는 요소
 - 적중율, 실패 패널티, Cache 접근시간 : 절충(Trade-off) 가 필요

- 적중율 향상방법

- Cache 용량 증가
- 블록 크기 증가
- 효율적 사상 방법 사용
- 프로그램 최적화
- 컴파일러 개선

4. Cache 성능개선

• 실패 패널티 축소 방법

- 메모리 대역폭(Bandwidth) 확장
- Cache와 메인 메모리 전송단위인 블록 크기를 줄여 전송시간 절약
- 메인 메모리로 부터 캐시로 블록이 적재될 때, CPU가 요청한 워드(Word)가 도착하자마자 CPU로 전송하는 방법 사용
 - Early Start : 블록을 Cache 적재할 때, 첫 번째 워드부터 순차적으로 적재하다, 요청된 워드가 Cache에 도착하면, 바로 CPU로 보낸다.
 - Critical Word First : 요청한 워드를 가장 먼저 CPU로 보내고, 나머지 블록 데이터를 차례로 적재하는 방식. Requested Word First 방식.

4. Cache 성능개선

- Cache 접근 시간 단축 방법

- 소용량 Cache 사용해서 접근 시간 단축 가능
- 직접 사상 방법 사용함으로써 태그(Tag) 비교시간 단축 가능
- 캐시 메모리 접근시 파이프라인 사용해서, Cache 접근 Latency를 단축할 수 있다.

Cache 사용 사례

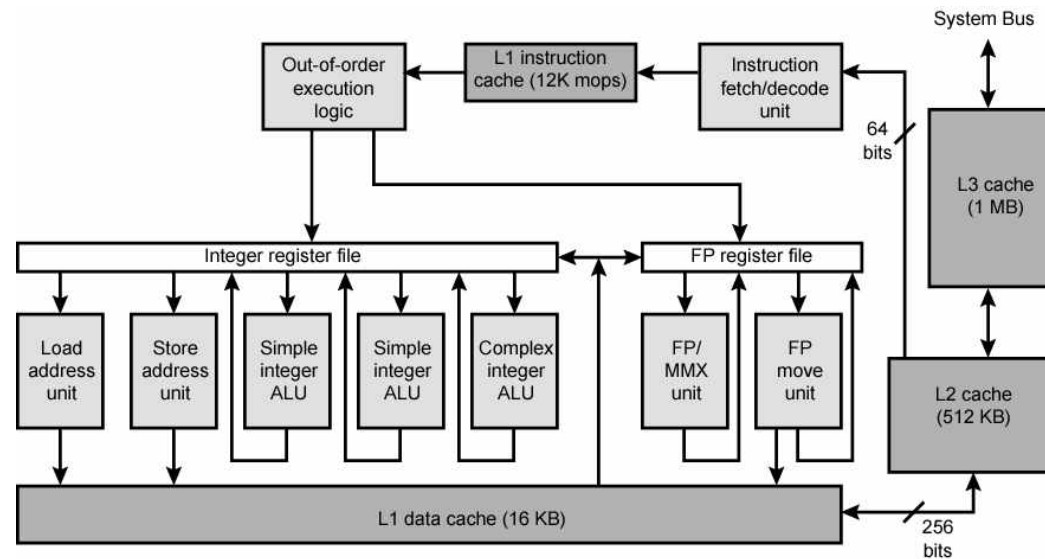
1. Intel CPU Cache

- **Intel CPU의 Cache**

- 80386 : no on-chip cache
- 80486 : 8K Cache
 - 16 byte Block
 - 4-way set associative mapping
- Pentium : 2 개의 on-chip L1 caches (split cache)
- Pentium III : L3 off-chip Cache 추가
- Pentium 4
 - L1 caches : 8kB, 64 byte lines, 4-way set associative
 - L2 cache : 256kB, 128 byte lines, 8-way set associative
 - L3 cache on-chip

1. Intel CPU Cache

- Pentium 4 블록 다이어그램



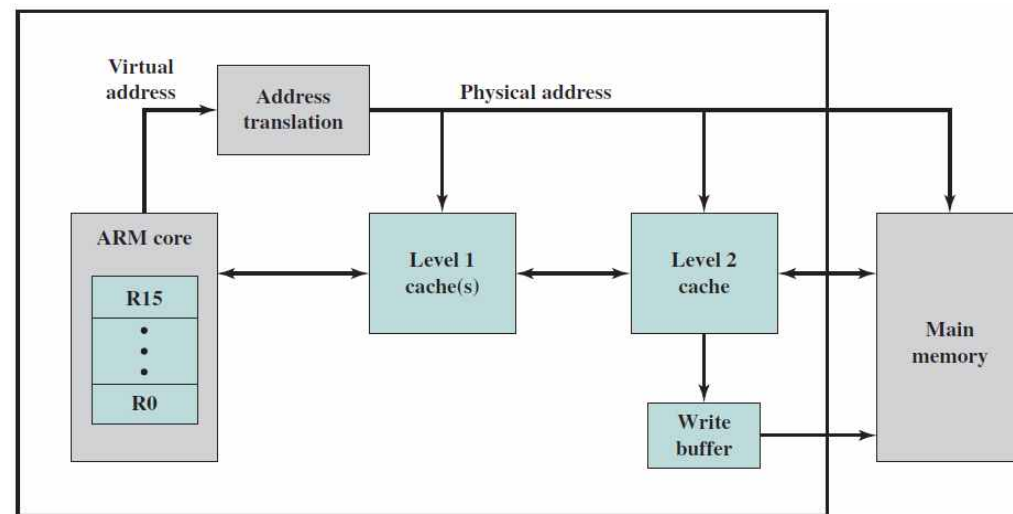
2. ARM Cache

- ARM cache features

Core	Cache Type	Cache Size (kB)	Cache Line Size (words)	Associativity	Location	Write Buffer Size (words)
ARM720T	Unified	8	4	4-way	Logical	8
ARM920T	Split	16/16 D/I	8	64-way	Logical	16
ARM926EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	16
ARM1022E	Split	16/16 D/I	8	64-way	Logical	16
ARM1026EJ-S	Split	4-128/4-128 D/I	8	4-way	Logical	8
Intel StrongARM	Split	16/16 D/I	4	32-way	Logical	32
Intel Xscale	Split	32/32 D/I	8	32-way	Logical	32
ARM1136-JF-S	Split	4-64/4-64 D/I	8	4-way	Physical	32

2. ARM Cache

- ARM Cache 구조



* Write buffer : 메모리 쓰기 속도 향상을 위한 소용량 FIFO 메모리

사상 함수

1. Cache 알고리즘 개요

- **사상 함수 (Mapping Function)**

- 메인 메모리 블록을 Cache 라인에 할당하는 방법
 - 직접 사상 (Direct), 연관 사상 (Associative), 세트-연관 (Set-Associative)

- **교체 알고리즘 (Replacement Algorithm)**

- Cache 라인을 교체하는 방법
 - LRU (Least Recently Used), FIFO, LFU (Least Frequently Used), Random

- **쓰기 정책 (Write Policy)**

- Cache 라인을 메인 메모리에 써넣는 방법
 - Write-Through, Write-Back

2. 사상 함수

- **사상 함수(Mapping Function)**

- Cache 라인의 수가 주 기억장치 블록의 수보다 적기 때문에, 주 기억장치 블록을 Cache 라인으로 매핑해주는 알고리즘이 필요.

- **사상 방법 (Mapping Techniques)**

- 직접 (Direct) : 메모리 블록이 정해진 Cache 라인에만 적재될 수 있다.
- 연관 (Associative, Fully associative) : 메모리 블록이 임의의 Cache 라인에 적재될 수 있다.
- 세트-연관 (Set Associative) : 직접 사상과 연관 사상의 절충.

2.1 사상 함수 예제

- 사상 함수 설명을 위한 가정 (Case Study)

- Cache 크기 : 64 KB
- Cache 블록 크기 : 4 B
 - Cache 블록 크기 = Cache 라인 크기
 - Cache에 포함되는 전체 라인 수 : $64 \text{ KB} / 4 \text{ B} = 16\text{K lines}$
- 메인 메모리 크기 : 16 MB
 - $2^{24} = 16\text{M}$: 24 bit 주소공간

3. 직접 사상

- 직접 사상(Direct Mapping)

- 가장 간단한 매핑 방법
- 메인 메모리의 각 블록 (block) 이 오직 한 군데의 Cache 라인 (cache line) 에만 매핑.
 - 메인 메모리 블록이 적재되는 Cache 라인의 위치가 고정.

- Cache 라인번호

- $(\text{메인 메모리 블록 번호}) \% (\text{Cache 내 라인들의 수})$

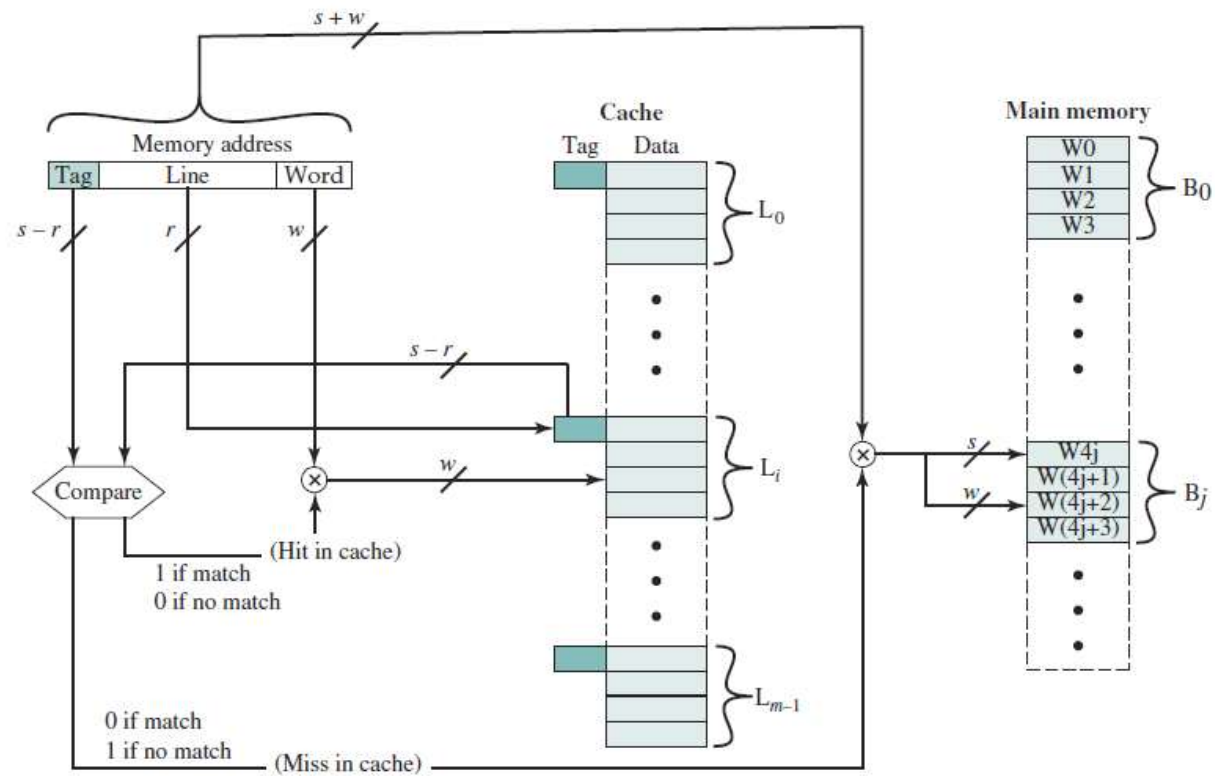
3. 직접 사상

- Case Study에 적용한 결과의 주소 구성
 - 24-bit 주소
 - 2-bit 워드 식별자 (Word Identifier) (4 byte block)
 - 22-bit 블록 식별자 (Block Identifier)
 - 8-bit tag (=22-14)
 - 14-bit 라인 번호

Tag s-r	Line r	Word w
8	14	2

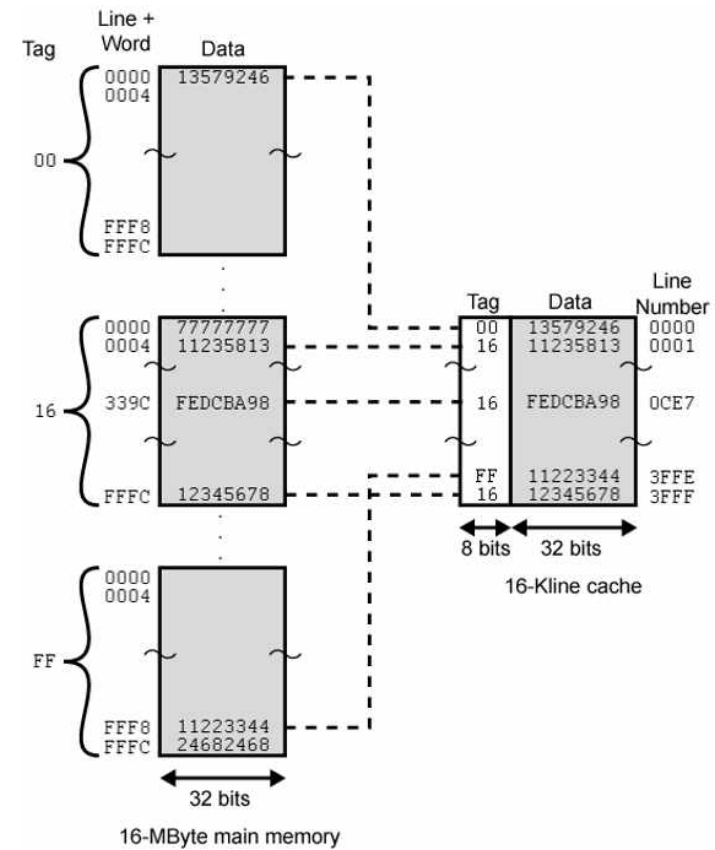
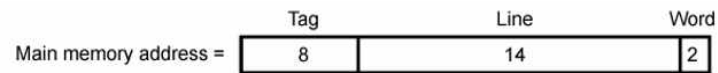
3. 직접 사상

- 직접 사상에서의 Cache 조직



3. 직접 사상

- Case Study의 Memory map



3. 직접 사상

- 장점

- 하드웨어 구현이 간단하고 접근 속도가 빠르다.
- 태그 비교 회로가 간단 → 대용량 캐시에 적합.

- 단점

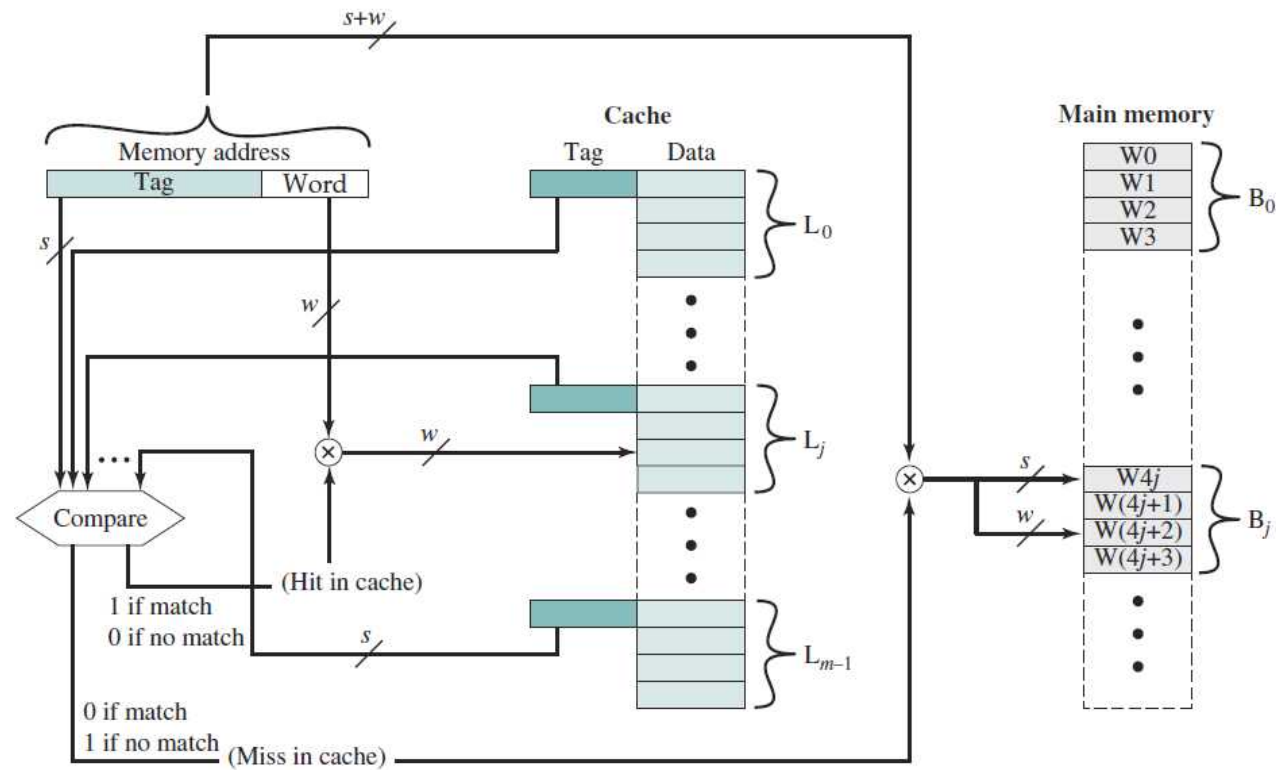
- 주어진 블록에 대해 고정 위치 → 평균 적중률 저하
- 프로그램이 같은 Cache 라인에 매핑된 2개의 블록을 연속적으로 액세스 하는 경우, Cache 적중율 저하. (Thrashing)

4. 완전 연관 사상

- **완전 연관 사상(Fully Associative Mapping)**
 - 메인 메모리 블록이 적재되는 Cache라인이 정해져 있지 않다.
 - Cache 라인 번호가 없음.
 - 태그를 사용하여 메인 메모리 블록을 식별.
- **원하는 블록을 찾기 위해 모든 Cache라인의 태그를 검사해야 한다.**
 - Cache 탐색 비용 증가

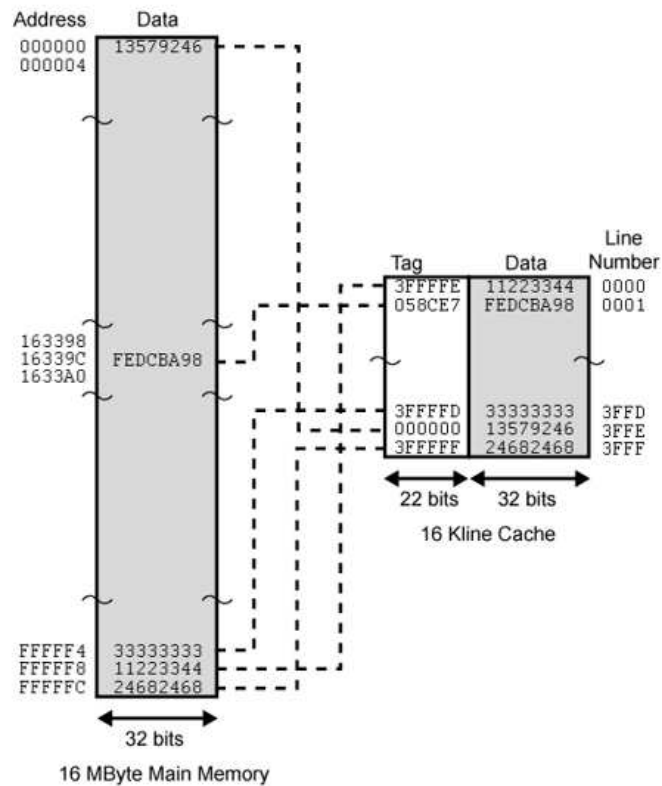
4. 완전 연관 사상

- 완전 연관 사상에서의 Cache 조직



4. 완전 연관 사상

- Case Study의 Memory map



4. 완전 연관 사상

- Case Study에 적용한 결과의 주소 구성

- 메인 메모리 블록은 22-bit 태그와 함께 저장.
- Cache에서 데이터를 탐색할 경우, 주소의 태그 필드와 Cache 라인의 모든 태그 필드를 비교
- 하위 2-bit 은 메모리 블록내에서의 워드 식별자



4. 완전 연관 사상

• 장점

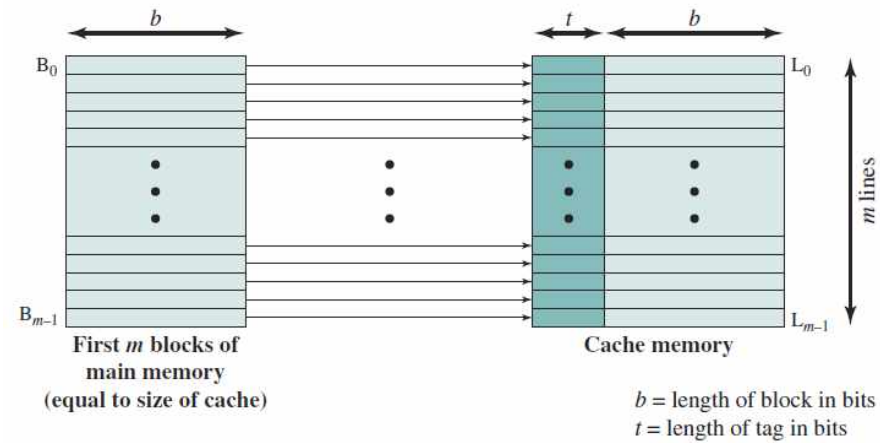
- 새로운 블록을 Cache에 적재하거나 교환할 때, Cache라인 선택 폭이 넓어짐
 - 메모리 블록에 대해 적재 융통성이 커짐.
- 소용량 Cache 를 사용해도 적중율이 높음.

• 단점

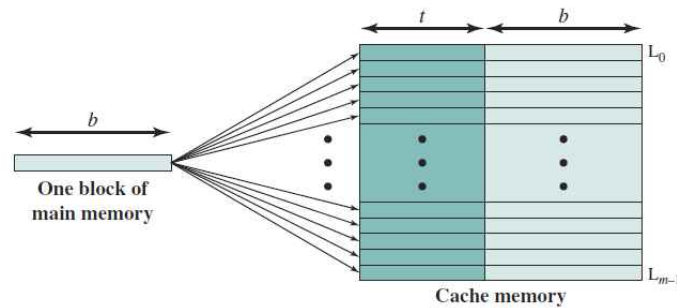
- 태그 길이 증가
- Cache 라인 탐색 비용 증가
 - Cache 라인들의 태그들을 병렬로 검사하기 위한 복잡한 회로가 필요.
- 고가의 연관 메모리 필요

4.1 직접 사상과 완전 연관 사상 비교

직접 사상



완전 연관 사상



5. 집합 연관 사상

- 집합 연관 사상(Set Associative Mapping)

- 직접 사상과 연관 사상의 장점을 취하고 단점을 줄인 절충형.

- Cache를 여러 개의 세트로 분리

- 각 세트는 여러 개의 라인으로 구성
 - 메인 메모리 블록은 정해진 세트의 모든 블록에 실장될 수 있다.
 - 메인 메모리 블록이 적재될 세트는 고정.
 - 세트내에서 라인 할당은 자유.

- Cache의 Set 번호 = (메모리 블록 번호) % (Cache의 전체 Set 수)

5. 집합 연관 사상

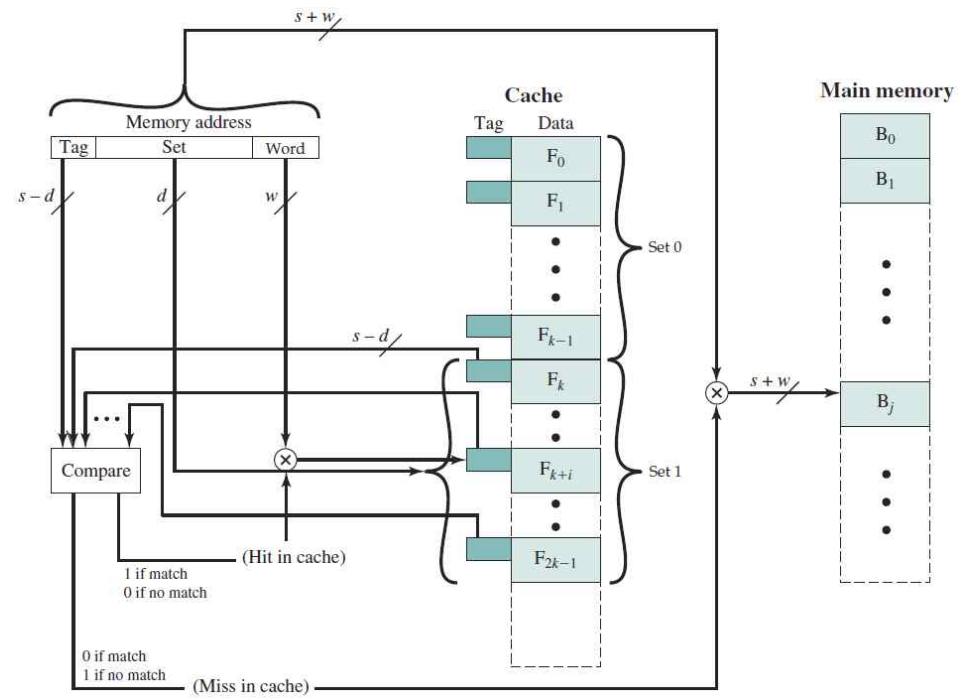
- K-way 세트-연관 사상

- 한 세트를 구성하는 Cache라인의 수가 k 개라면, k-way 세트-연관 사상이라 함.
 - k-way 의 집합 연관성(Set Associativity)은 'k' 가 된다.
- 예를 들면,
 - 2-way 세트-연관 사상 : 세트 당 2 개의 Cache라인 보유.
 - 메모리 블록은 2 개의 Cache 라인중 어느 한 곳에 저장.
 - Set Associativity = 2

-

5. 집합 연관 사상

- 집합 연관 사상에서의 Cache 조직



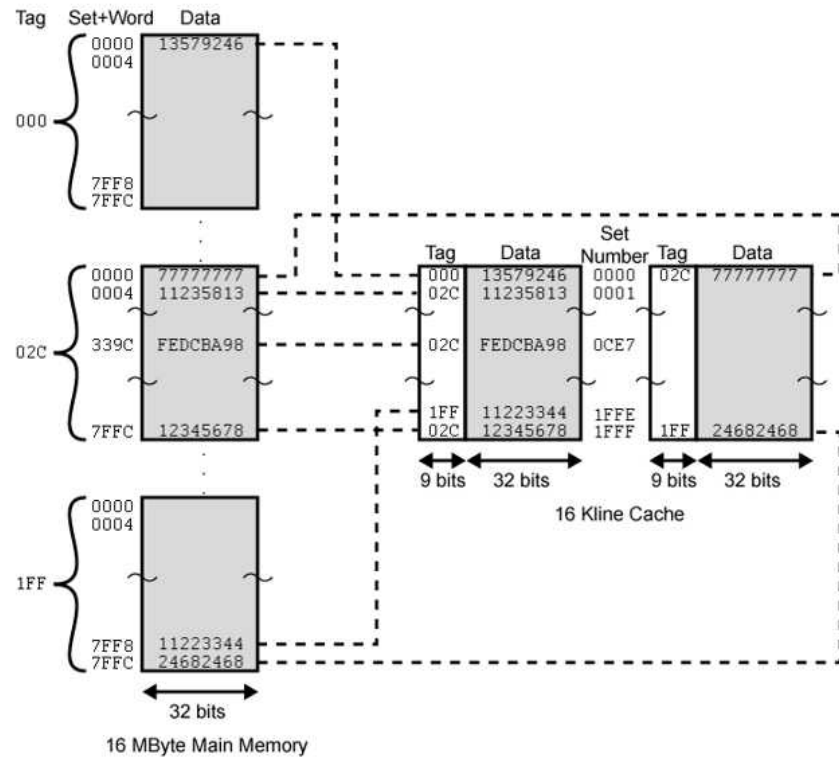
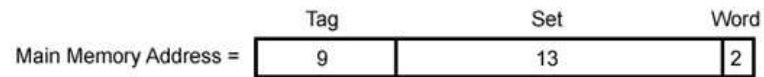
5. 집합 연관 사상

- Case Study에 적용한 결과의 주소 구성
 - 24-비트 어드레스
 - 9-비트 Tag 필드
 - 세트에 저장된 Cache라인의 식별자
 - 13-비트 Set 필드
 - 세트 식별자 : Direct 매핑의 line 필드와 유사
 - 2-비트 Word 필드
 - Cache 라인내의 워드 또는 바이트 식별자

Tag (9)	Set (13)	Word (2)
---------	----------	----------

5. 집합 연관 사상

- Case Study의 Memory Map
 - 2-Way



5. 집합 연관 사상

• 장점

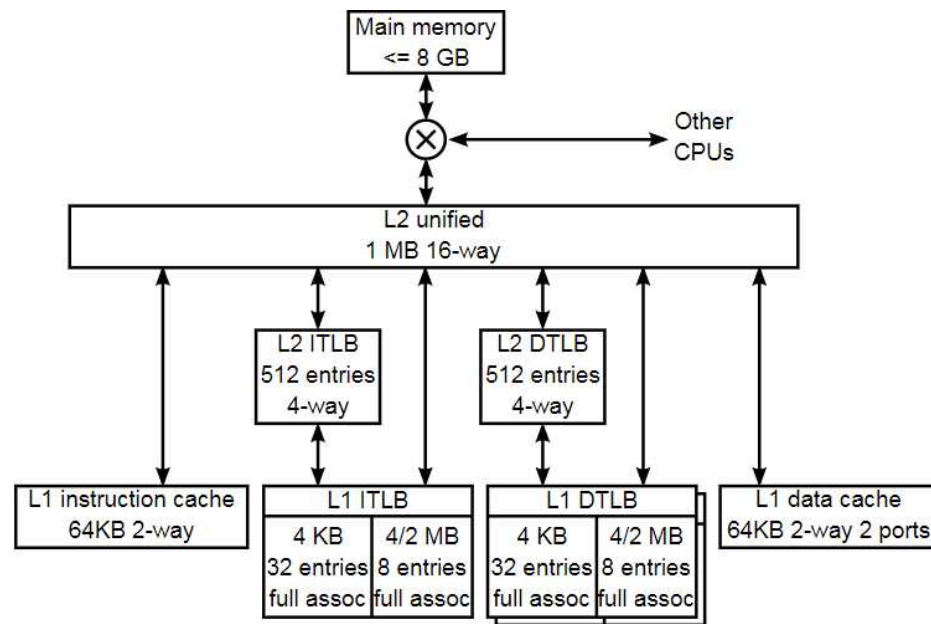
- 집합 연관 사상은 완전 연관 사상에 비해 상대적으로 태그의 길이가 짧고, 비교 횟수도 짧아 하드웨어 비용도 작고 사상 작업 속도도 빠르다.
- 적중율은 완전 연관사상 보다 떨어지지만, 직접 사상보다는 높다.

• 단점

- 직접 사상과 비교할 때 탐색속도나 하드웨어 비용이 상대적으로 떨어진다.
 - 직접 사상보다 태그 길이 증가
 - 직접 사상보다 Cache 라인 탐색 비용 증가

5. 집합 연관 사상

- AMD Athlon 64 CPU (K8 Core)의 Cache 계층 구조



교체 알고리즘

1. 교체 알고리즘 개요

- **교체 알고리즘 (Replacement Algorithm)**

- Cache가 전체가 채워졌거나, 사상 함수에 의해 할당된 Cache 라인이 선점되어 있을 경우,
 - 새로운 블록을 Cache로 적재할 때는 현재 Cache에 저장되어 있는 블록들 중 하나를 선택해서 교체하는 방법.

- **직접(Direct) 사상 경우**

- 선택의 여지가 없음 : 각 블록이 저장되는 위치가 고정되어 있기 때문에, 해당 라인에 저장된 블록과 교체.

- **완전 연관 / 집합 연관 사상인 경우**

- 교체 알고리즘 (Replacement Algorithm) 적용하여 교체 대상을 선정.
- 성능향상을 위해, 알고리즘은 하드웨어로 구현
- 완전 연관 사상은 교체 알고리즘이 복잡하지만, 집합 연관은 집합내에서만 교체대상을 선정하기 때문에 간단하다.

2. 교체 알고리즘 유형

- **Least Recently Used (LRU)**

- Cache내에서 사용되고 있지 않은 채로 가장 오랫동안 있었던 블록을 교체
 - USE 비트 사용
- 가장 널리 사용되는 알고리즘
- 하드웨어 구현비용이 크다.

- **First in first out (FIFO)**

- Cache내에서 가장 오랫동안 머물렀던 블록을 교체.
 - 먼저 적재된 블록을 먼저 축출.
- Round-robin 또는 Circular Queue 로 구현
- 타임 스탬프 (Time-stamp) 사용.
- 구현비용이 상대적으로 저렴.

2. 교체 알고리즘 유형

- **Least frequently Used (LFU)**

- 가장 적게 참조되었던 블록을 교체.
- 참조 카운터 (Reference Counter) 사용.

- **Random**

- 무작위로 교체.
- 사용 횟수에 근거한 다른 방법에 비해, 성능이 크게 떨어지지 않는다.
- 구현비용이 저렴.

쓰기 정책

1. 쓰기 정책 개요

- 쓰기 정책(Write Policy) 필요성

- CPU가 데이터를 갱신할 경우, Cache에 있는 데이터만을 갱신한다.
- Cache에 적재된 데이터는, 메인 메모리에도 저장되어 있기 때문에, CPU가 데이터 갱신시, 일관성(Consistency)을 유지하기 위해서는 Cache와 함께 메인 메모리에도 갱신해야 한다.
 - Cache = 복사본, 메인 메모리 = 원본

- Cache 블록 갱신 방법은 갱신 시점에 따라 2가지로 나눈다.

- Write-Back 방식 : Cache만 갱신 하는 방법, 메인 메모리는 나중에 갱신.
- Write-Through 방식 : Cache와 메인 메모리 모두 갱신하는 방법.

1. 쓰기 정책 개요

- **Cache 블록 갱신시 고려해야 할 점**

- 여러 개의 디바이스가 메인 메모리를 같이 사용할 경우, 데이터 일관성 유지 필요.
 - CPU 와 I/O 모듈이 메인 메모리에 직접 액세스할 경우.
- 같은 버스에 연결된 여러 개의 CPU 가 각자 Cache를 가지고 있을 경우, 데이터 일관성 유지 필요.

2. Write-Back 방식

- 데이터 갱신이 Cache에서만 일어난다.

- Cache 데이터 변경시 Dirty-bit (또는 Use-bit) 를 사용하여 데이터 변경을 표시한다.
- 블록이 교체될 때, Dirty-bit 가 세트된 경우에만 메인 메모리를 갱신한 후 교체한다.

- 메인 메모리 갱신 시점

- 메인 메모리로부터 메모리 블록을 읽어 Cache에 적재하기 전에, Cache를 선점하고 있던 블록이 변경되었을 경우, 메인 메모리에 저장할 필요가 있다.
- Cache 블록이 변경되지 않는 경우, 새로운 블록을 덮어쓴다. (Overwrite)

2. Write-Back 방식

- 장점

- 메모리에 대한 쓰기 동작 부담을 최소화한다.
- 데이터 갱신 속도가 빠르고, 메모리 트래픽 감소.

- 문제점

- 메인 메모리의 일부분이 무효 (Invalid) 상태에 있다.
 - I/O 모듈에 의한 메모리 접근은 항상 Cache를 통해서 이루어져야 한다.
 - 회로가 복잡해지고 성능저하 (Bottle-neck 발생)

3. Write-Through 방식

- 메인 메모리와 Cache의 일관성을 유지하기 위하여 모든 쓰기 동작들이 Cache뿐만 아니라 메인 메모리에도 같이 이루어진다.
 - 모든 쓰기 동작에 대해 메인 메모리 접근 필요
- 장점
 - 메인 메모리의 내용을 항상 유효하도록 보장한다.
 - 로컬 Cache를 사용하는 멀티 프로세서 시스템에서 Cache 일관성 유지가 간단.
 - Cache의 일관성을 유지하기 위하여 메인 메모리와의 트래픽(Traffic) 을 감시 필요.
- 단점
 - 기억장치 사용량이 많아져서 병목현상 (bottle-neck) 을 야기할 수 있다.
 - 쓰기에 소요되는 시간이 증가한다.

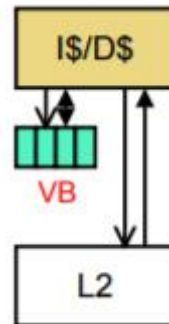
4. Write Buffer

- 메인 메모리에 쓰여질 데이터들을 임시로 저장하고 있는 버퍼
 - 캐시와 메인 메모리 사이에 위치
 - 메인 메모리에 데이터 쓰는 것이 늦을 때 사용
 - Buffered write-through
- 데이터 Write 하면서 Read 실행 가능
- 버퍼 Full 시, 슬롯이 빌 때까지 Write 동작이 정지.

4. Write Buffer

- **Victim Buffer**

- Write buffer의 일종으로 Write-back 캐시에서, 사용후 축출된(Dirty Evicted) 캐시 라인을 임시로 저장하는 공간.
- Cache Miss 시, Victim Buffer 를 검사해서, hit 하면, 캐시로 복사.
- 임시 백업 공간으로 활용 가능
 - Victim 버퍼에 저장된 데이터를 다시 참조할 경우, 효과적.



5. Write Miss

- 캐시에 포함되지 않은 어드레스에 데이터를 쓸 때 발생
- Write Miss 처리 방법
 - No-Write-Allocation 방식
 - 써야할 데이터를 캐시에 쓰지 않고, 메인 메모리에 직접 써넣는다.
 - 바로 사용하지 않을 데이터를 쓸 때 효과적
 - Write-Allocation
 - 메인 메모리에서 라인을 읽어서 캐시에 할당한 후, 써 넣는다.
 - 바로 사용할 데이터인 경우 효과적
 - Allocate-on-Write라고 함.

Cache 일관성 유지

1. Cache 일관성 개요

- **Cache 일관성 (cache coherency)**

- 메인 메모리를 공유하는 여러 개의 Cache사이에는 일관성 (coherency) 이 유지되어야 한다.

- **Cache 일관성을 유지하기 위해 사용되는 방법**

- 주소 버스 감시를 통한 일관성 유지.
- 하드웨어를 사용한 일관성 유지.
- Non-Cacheable 메모리 영역 지정.

2. 주소 버스 감시

- 주소 버스를 감시하여 다른 Cache 컨트롤러가 메모리 쓰기 작업을 실행하는지 여부를 감시.
- 만약, 다른 Cache 컨트롤러가 메인 메모리에 쓰고 있는 데이터가 자신의 Cache에도 적재되어 있다면, 그 데이터를 무효화(Invalidate) 시킨다.
 - 모든 Cache 제어기들이 write-through 정책을 사용할 경우에만 적용 가능.

3. 하드웨어를 사용한 일관성 유지

- 메인 메모리에 대한 모든 갱신들이 다른 모든 Cache들에도 반영될 수 있도록 별도의 하드웨어를 추가하여, 데이터 갱신에 대한 하드웨어적 투명성을 확보하는 것이다.
- 어떤 프로세서가 자신의 Cache에 있는 단어를 수정한다면, 그 내용이 메인 메모리에도 갱신되고, 다른 Cache들에도 해당 데이터가 있다면 모두 갱신된다.

4. Non-Cacheable 메모리 영역 지정

- 메인 메모리의 일부분을 캐싱 불가능 (non-cacheable) 하게 만들고 여러 개의 프로세서가 그 부분만을 공유하게 한다.
- 캐싱 불가능한 메모리 영역에 대한 접근 (access) 은 모두 Cache 미스 (cache miss) 처리한다.
 - 해당 영역은 Cache에 복사되지 않기 때문.
- 캐싱 불가능한 메모리 영역은 칩-선택 (chip-select) 신호 또는 어드레스의 상위주소 비트들을 이용하여 구분할 수 있다.

메모리 관리

1. 메모리 관리

• 메모리 관리 필요성

- 메모리는 컴퓨터 시스템에서 가장 중요한 자원
- 프로그램 크기가 메모리 용량보다 더 빠르게 증가.
- 여러 프로그램의 동시 실행
- 여러 사용자의 동시 사용

• 메모리의 효과적인 관리 필요성

- 여러 개의 프로그램이 동시에 실행될 수 있는 메모리 공간을 제공해야 한다.
- 각 프로그램의 고유 메모리 공간이 보호되어야 한다.
- 프로그램사이의 메모리 공간이 공유될 수 있어야 한다.

1. 메모리 관리

- **메모리의 관리 방법**

- 가상 메모리, 메모리 보호, 메모리 공유, 메모리 재배치
- 메모리 리소스 관리 : 운영체제(Operating System)

- **운영체제에서 메모리 관리 → 메모리 관리 기법의 복잡화 → 하드웨어를 사용한 성능향상 필요.**

- 메모리 관리 전용 하드웨어가 컴퓨터 시스템에 추가
 - MMU(Memory Management Unit)

2. 메모리 관리 기법

- 하나의 프로그램만을 실행하는 단일 프로그램(Uni-program) 컴퓨터 시스템에서 메모리는 2개 영역으로 나누어서 사용
 - 운영체제 또는 모니터 프로그램 영역 : 컴퓨터 시스템 자원관리
 - 사용자 프로그램 영역
- 여러 개의 프로그램을 동시에 실행하는 다중 프로그램(Multi-program) 컴퓨터 시스템에서는 사용자 프로그램 영역은 다른 프로그램들과 메모리를 공유하게 된다.

2. 메모리 관리 기법

- 메모리 공간의 분리하고 공유하는 작업을 메모리 관리(Memory Management)라고 하고, 운영체제에서 수행한다.
- 주요 메모리 관리 기법
 - 스와핑(Swapping)
 - 파티셔닝(Partitioning)
 - 페이징(Paging)
 - 세그멘테이션(Segmentation)

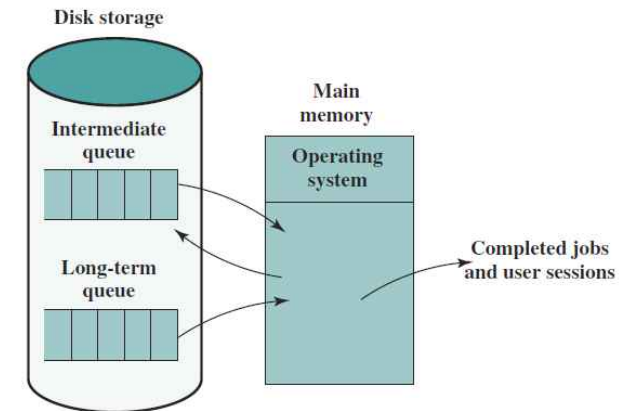
3. 스와핑

- 프로그램이 I/O 작업을 포함하고 있을 경우, I/O 장치의 처리속도는 CPU 속도보다 느리기 때문에, CPU는 휴지(Idle) 상태에 있게 된다.
- 다중 프로그램 시스템에서는, I/O 작업 종료를 대기하는 프로그램이 늘어나고, 이 프로그램들이 메인 메모리 공간을 차지하게 되어, 새로운 프로그램 실행 공간확보가 어려워지게 된다.
- 해결책
 - 메인 메모리 용량 증가 : 하드웨어 비용 증가
 - 스와핑(Swapping) 기법 사용 : CPU를 필요로 하는 프로그램만 메모리, I/O 작업 종료를 대기하는 프로그램은 보조 기억장치로 이동해서 대기.

3. 스와핑

- 스와핑 절차

- CPU 대기 시간이 긴 프로그램(long-term queue에서 대기)은 보조 기억장치(하드 디스크)에 저장.
- 필요시에 메인 메모리로 로딩해서 실행.
- 프로그램 종료시, 메인 메모리에서 삭제
- 메인 메모리 공간이 부족할 경우 메인 메모리에 적재된 프로그램중 하나를 선택해서, 보조 기억장치로 이동
 - Intermediate queue로 이동



Swapping

4. 파티셔닝

- 파티셔닝(Partitioning)

- 메인 메모리를 파티션(Partition) 단위로 분할하여 사용하는 것
- 분할 방법에 따라
 - 고정길이 분할
 - 가변길이 분할

- 고정길이 분할

- Equal-Sized : 동일한 크기의 섹션으로 분할
- Unequal-Sized : 다양한 섹션 크기로 분할

- 가변길이 분할

- 섹션의 길이와 수가 가변, 동적 분할
- 프로그램 로딩시, 필요한 메모리 만큼만 분할해서 할당.

4.1 고정길이 파티셔닝

- 고정길이(Fixed-sized) 파티션

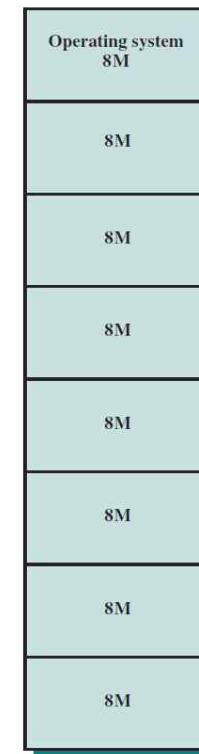
- 파티션의 길이가 사전에 고정
 - 파티션의 길이는 동일(Equal-Sized)하거나 다를 수 (Unequal-Sized) 있다.
- 메인 메모리에 프로그램 로딩시, 최적의 크기를 찾아서 할당. (Best-Fit)

- 장점

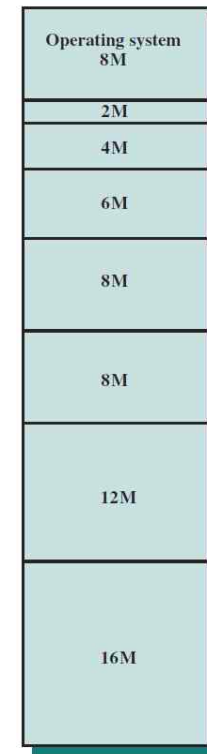
- 구현하기가 쉽다.
- 운영체제 부담이 적다

- 단점

- 메모리 낭비가 심하다.
 - 쓰여지지 않고 남은 공간(Internal Fragmentation)이 발생.



Equal-Sized



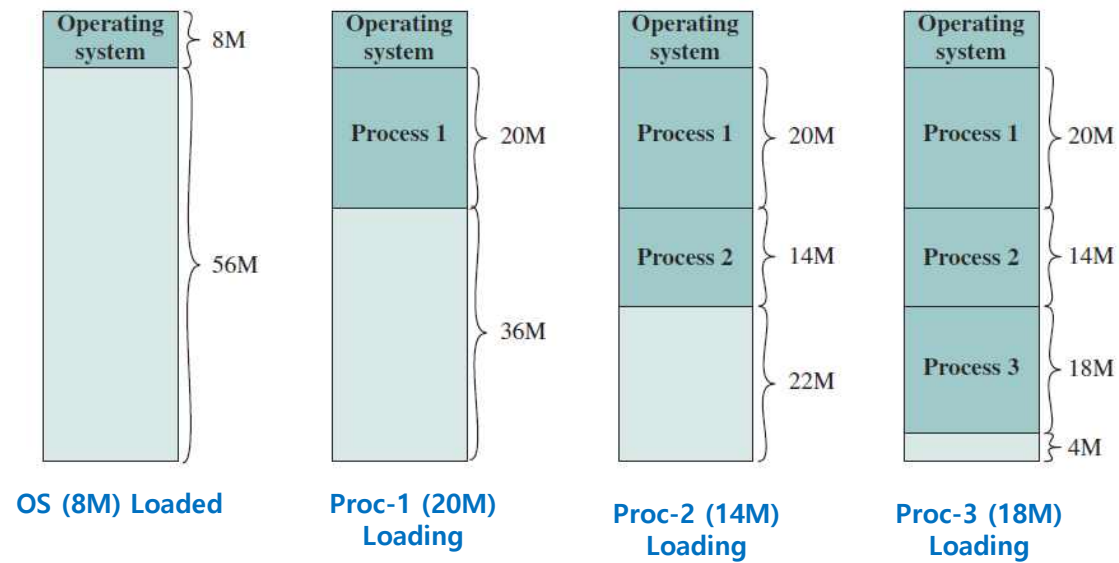
Unequal-Sized

4.2 가변 길이 파티셔닝

- 프로그램이 메인 메모리에 로딩시, 필요한 만큼만 할당받아서 사용.
- 시간이 지나면서, 사용할 수 없는 작은 공간(Hole)들이 발생
- Hole 제거 방법
 - Coalesce(병합)
 - 연속된 2개의 Hole을 합쳐 1개로 만듦.
 - Compaction(압축) :
 - 운영체제는 사용하지 않는 메모리 공간을 합치거나, 기존 할당된 공간을 이동시켜서, 보다 큰 공간을 확보해 둔다.
 - 이 작업은 시간이 많이 걸려서(Time-consuming), 결국 CPU 성능을 저하시킨다.

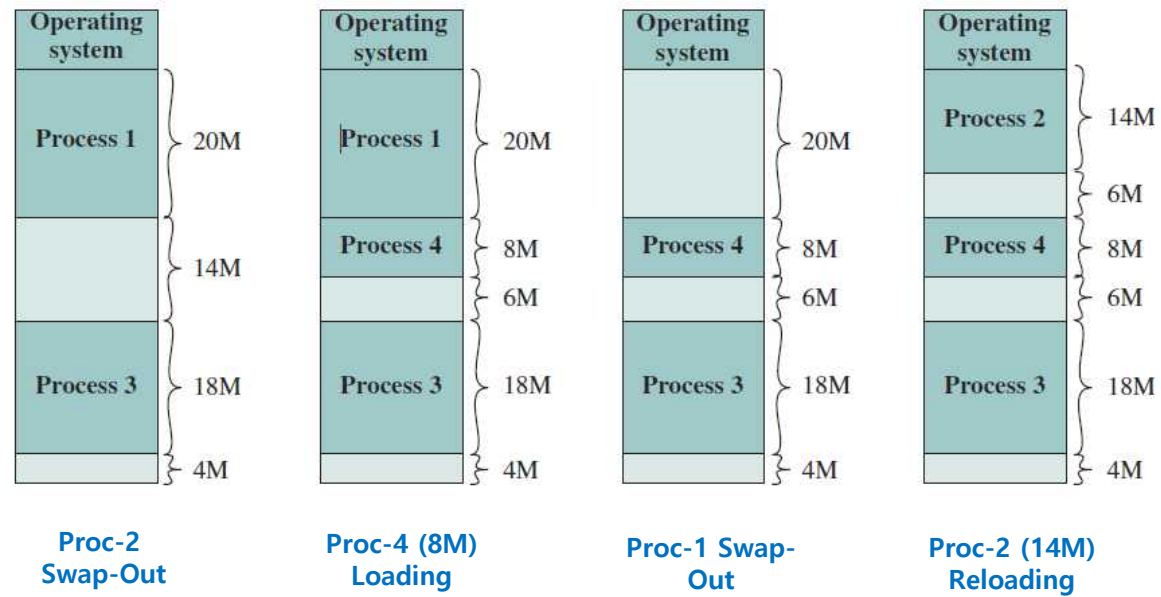
4.2 가변 길이 파티셔닝

- 적용례 : 64M 메인 메모리에 4개의 프로그램이 로딩되어 실행되는 경우.



4.2 가변 길이 파티셔닝

- 적용례



4.3 재배치

- 프로그램 재배치(Relocation)

- 프로그램이 보조 기억장치로 Swap-Out 된 후, 다시 메인 메모리로 리로딩(Reloading) 될 때, 같은 장소를 사용한다는 보장은 없음.
- 로딩시 재배치

- 따라서, 프로그램이 사용하는 주소(Address)는 물리 주소가 아닌 논리 주소를 사용.

- 논리 주소(Logical Address) : 프로그램 시작 위치를 원점(Origin)으로 한 상대 주소(Relative Address) 체계를 사용.

- 물리 메모리 로딩시 자동으로 물리 주소로 변환.

- 프로그램이 로딩 시작 주소 (물리주소) = Base Address
- 물리 주소 = (Base Address) + (프로그램내의 논리 주소)

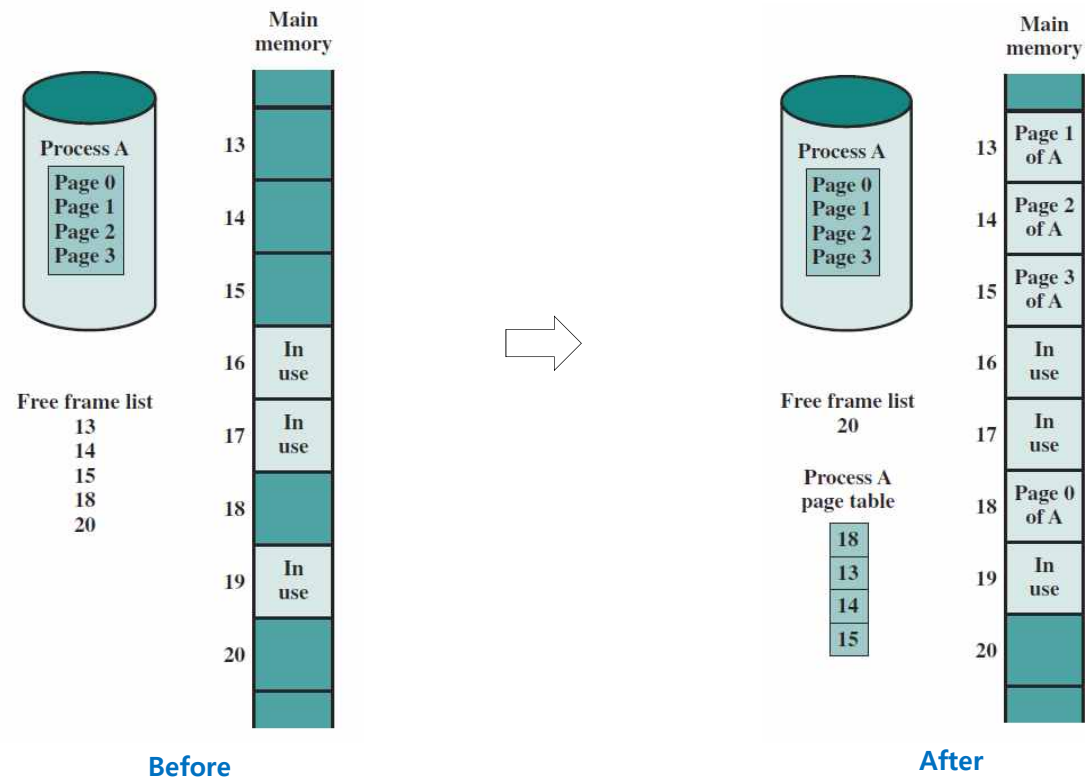
- 변환 작업은 운영체제에 의해 소프트웨어적으로 실행될 수 도 있고, 메모리 관리 유닛에 의해 하드웨어적으로 실행될 수 있다.

5. 페이징

- 고정길이, 가변길이 분할의 문제점인 파편화(Fragmentation)를 해결하기 위한 방법
- 페이징(Paging)
 - 메모리뿐 만아니라 프로그램도 동일하게 분할하는 방법
 - 메모리 분할 : 동일한 크기의 소용량 공간으로 분할
 - 메모리의 소용량 공간 = 페이지 프레임(Page Frame)
 - 프로그램 분할 : 동일한 크기로 작게 분할
 - 분할된 작은 프로그램 = 페이지(Page)
- 프로그램 로딩시 필요한 페이지만큼 페이지 프레임을 할당
 - 파편화(Fragmentation) 최소화.
- 페이지 프레임 할당 및 미사용 페이지 프레임(Free Frame) 관리는 운영체제.
 - - 페이지 테이블(Page Table)을 사용하여 관리

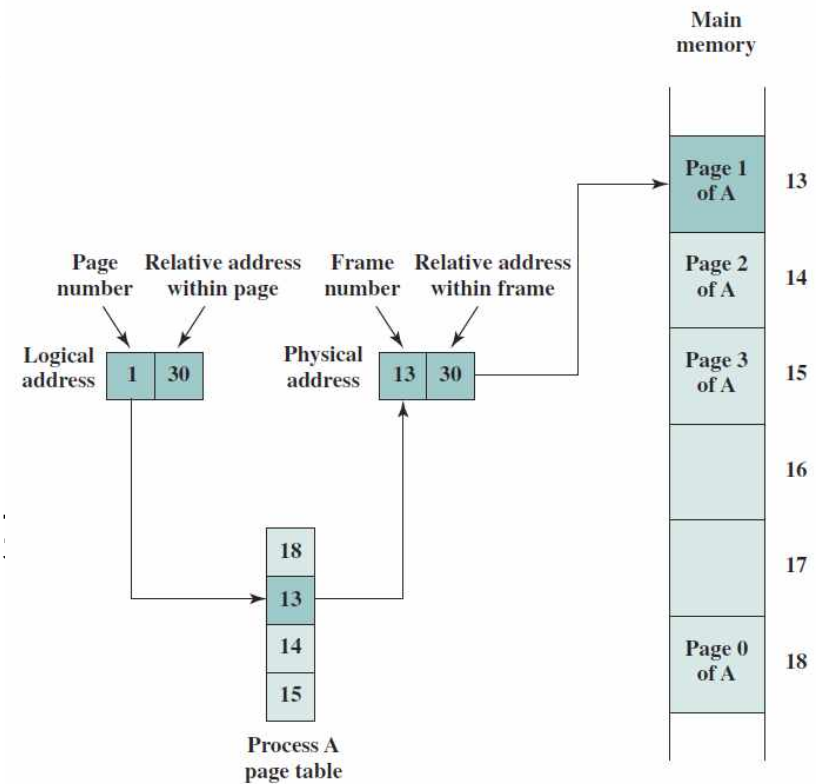
5. 페이징

- 미사용(Free) 페이지 프레임 할당 예



5. 페이징

- 논리 주소에서 물리 주소로 변환하는 과정
 - 페이지 테이블(Page Table) 사용
- 프로그램당 페이지 테이블 할당
 - 페이지 테이블 관리의 운영체제.
 - 변환 작업은 전용 하드웨어를 사용.
- 프로그램이 메인 메모리에 로딩될 때, 프로세스 프레임에 로딩되고, 페이지 테이블이 설



6. 세그먼테이션

- 세그먼테이션(Segmentation)

- 프로그램을 세그먼트(Segment)라는 논리적 단위로 분할하여 관리하는 방식.

- 페이징과 세그먼트 차이

- 페이징

- 프로그래머에게 Invisible
- 메모리 공간의 확장

- 세그먼트

- 프로그래머에게 Visible
- 프로그램과 데이터를 용도 또는 특성에 따라, 논리적으로 구조화 가능
- 논리단위로 관리 가능 (접근 권한, 보호 속성 설정 등)

6. 세그먼테이션

- 세그먼테이션 장점

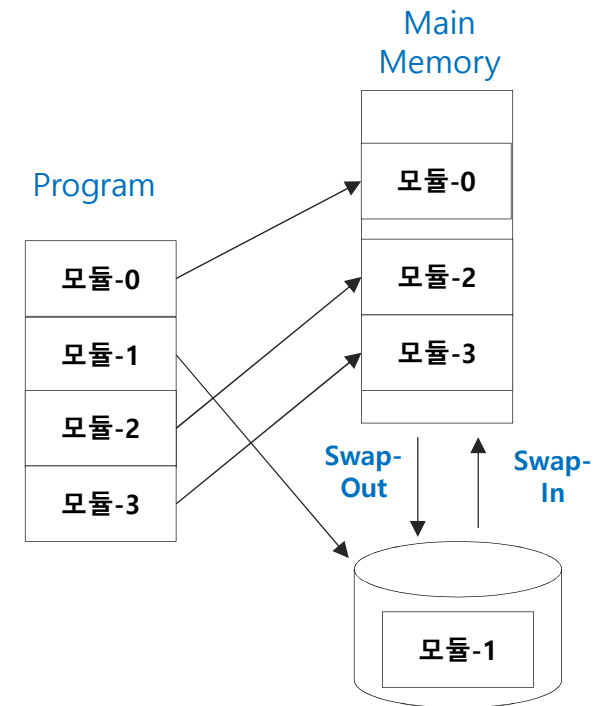
- 데이터 구조체의 확장이 용이
- 세그먼트를 독립적으로 컴파일하거나, 로딩할 수 있다.
- 세그먼트 단위로 공유, 보호 기능을 사용할 수 있다.

- 대부분의 컴퓨터 시스템은 세그먼테이션과 페이징을 같이 사용.

가상 메모리 구조

1. 메모리 Overlay

- 초기 컴퓨터 시스템의 메모리 관리 방법
- 메인 메모리 용량보다 큰 프로그램을 실행하기 위한 방법
 - 프로그램을 여러 개의 모듈로 분리한 후, 모듈들을 메인 메모리와 보조 기억장치(주로 하드 디스크) 사이에서 공간적으로 이동(Swapping)시키면서 프로그램을 실행하는 방법.
 - Swap-Out : 메인 메모리에 로딩된 모듈을 보조 기억장치로 이동시키는 것.
 - Swap-In : 보조 기억장치에 저장된 모듈을 메인 메모리로 로딩시키는 것.



1. 메모리 Overlay

- 실행에 필요한 모듈만을 메모리에 로딩해서 사용함으로써, 메모리 공간을 효율적으로 사용.
- 장점
 - 메모리 공간의 효율적 사용
- 단점
 - 프로그래머가 Overlay 공간을 직접 관리해야 한다.
 - 프로그램 이식성(Portability) 저하 : 메모리 용량이 다른 시스템으로 이전시 다시 코딩.
 - 다중 프로그램 실행시 메모리 공간 보호가 필요.

2. 가상 메모리

- 메인 메모리의 용량에 의해 제한을 받는 메모리 공간 → Real/Physical 메모리
- 물리 메모리와 상관없이 논리 주소 공간에 정의되는 메모리 → 가상(Virtual) 메모리
 - 물리 메모리 용량 제한을 넘어서 프로그래밍 가능.
- 가상 메모리(Virtual Memory) 필요성
 - 여러 개의 프로그램의 메모리 자원의 효율적이고 안전한 공유
 - 메모리 용량 부족으로 인한 프로그래머 부담 경감.
 - 메모리 관리는 운영체제로 넘김.
- 프로그램이 사용하는 메모리 공간과 실제 물리 메모리 공간을 분리
 - 프로그래머의 메모리 관리 부담을 제거

2. 가상 메모리

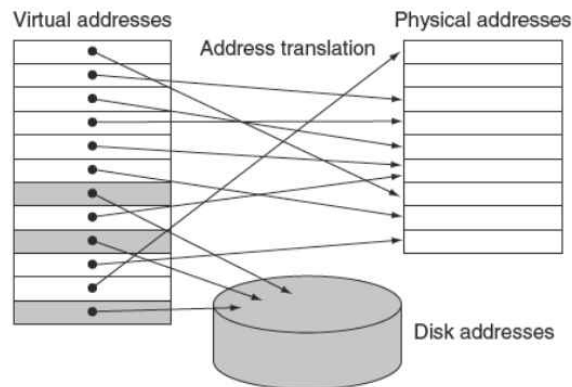
- 가상 주소(Virtual Address)

- 프로그램이 사용하는 주소, 논리(Logical)주소.

- 물리 주소

- 물리 메모리에서 사용하는 주소, 실제(Real) 주소

- 주소 변환(Address Translation) : CPU에서 프로그램을 실행하기 위해서는 가상 주소를 물리 주소로 변환하는 과정이 필요하다.



2.1 Demand 페이징

- **Demand paging**

- 프로그램의 모든 페이지를 한번에 메모리에 로딩하지 않고, 필요한 경우에만 메모리에 로딩.
 - 메모리 공간 절약
 - 프로그램 로딩 시간 절약.
- CPU가 원하는 페이지가 메모리에 로딩되어 있지 않은 경우 "Page Fault"를 발생시켜, 해당 페이지를 메모리에 로딩.
 - Page fault는 운영체제가 처리.

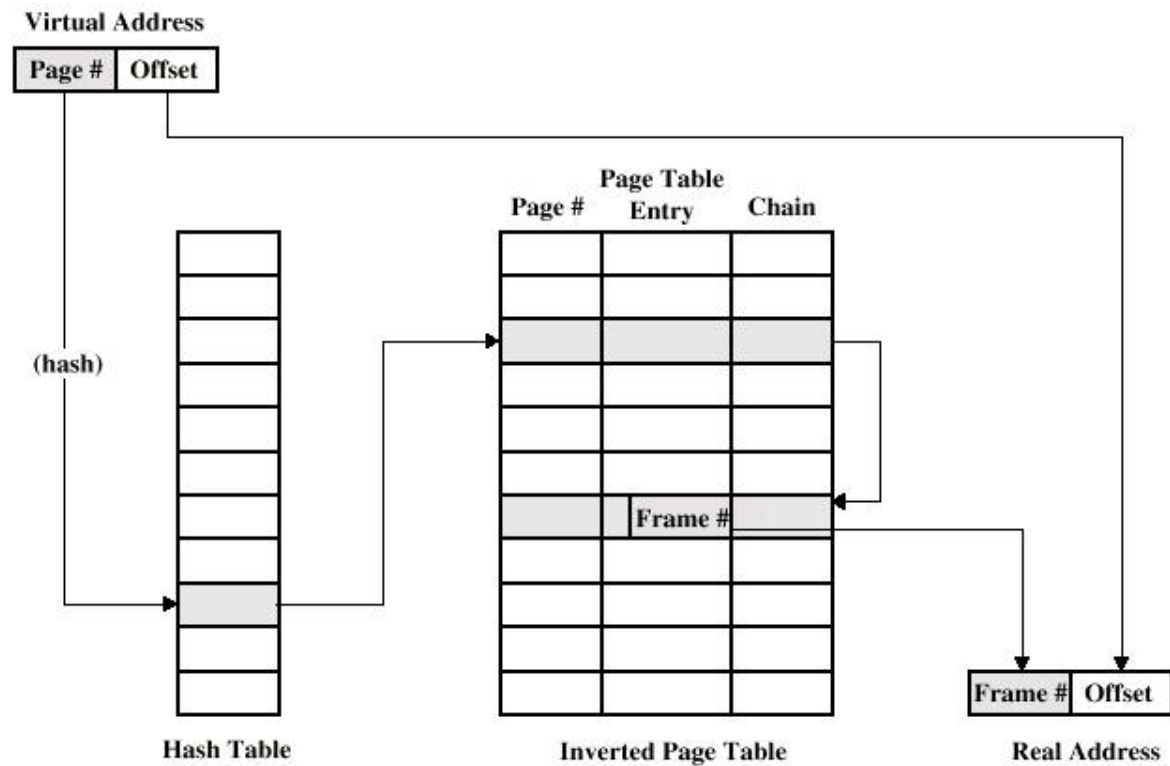
- **Page fault 처리**

- 운영체제는 해당 페이지를 보족 기억장치로부터 메인 메모리에 로딩.
 - 메인 메모리에 여유 공간이 없을 경우, 이미 로딩된 페이지를 방출
 - 페이지 교체 알고리즘 적용.

2.2 Thrashing

- 메인 메모리 용량에 비해 너무 많은 프로그램이 동시에 실행될 경우, 페이지 교체가 너무 빈번하게 발생하는 현상
 - 운영체제가 페이지 교체에 대부분의 시간을 소모.
- 해결 방법
 - 페이지 교체 알고리즘의 효율을 향상
 - 사용 이력(History)를 기반으로 한 교체대상 선정
 - 동시에 실행되는 프로그램 수를 축소.

2.3 Inverted Page Table Structure

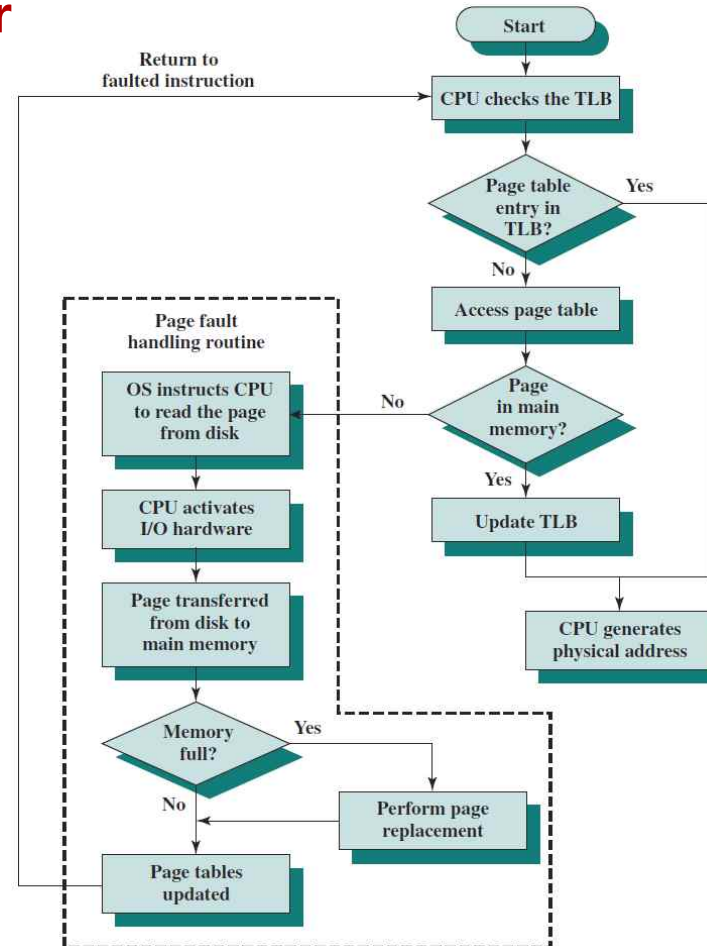


2.4 Translation Lookaside Buffer

- 가상 메모리 사용시 2번의 물리 메모리 접근이 필요
 - 1st 메모리 접근 : 페이지 테이블 인출
 - 2nd 메모리 접근 : 데이터 인출
 - 메모리 접근 속도 저하
- 해결 방법
 - 자주 사용되는 페이지 테이블 엔트리를 저장하는 소용량 Cache를 사용
 - TLB(Translation Lookaside Buffer)

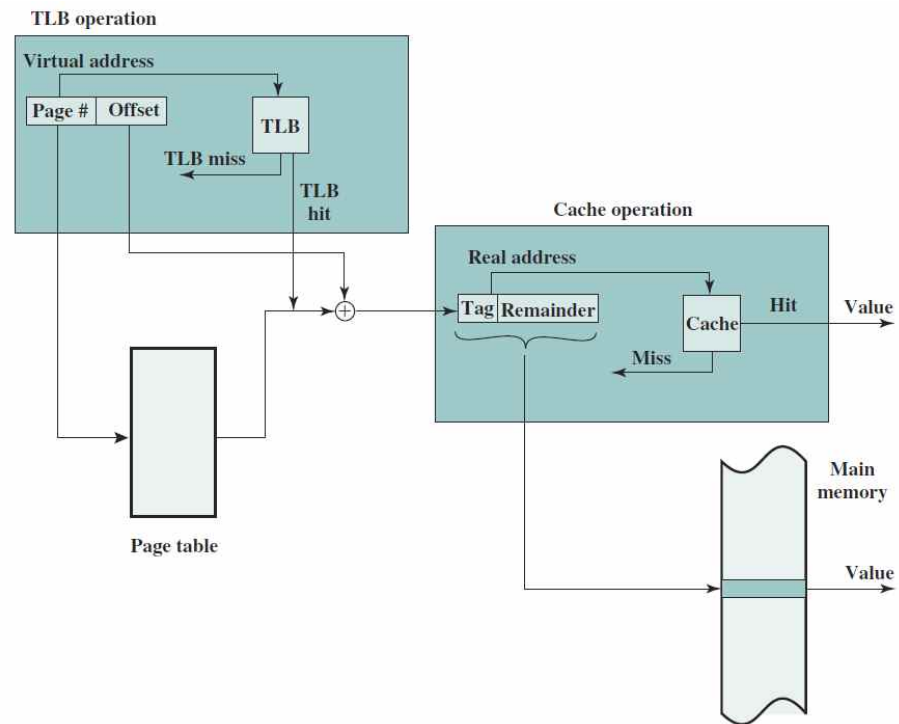
2.4 Translation Lookaside Buffer

- TLB를 사용한 페이징 흐름



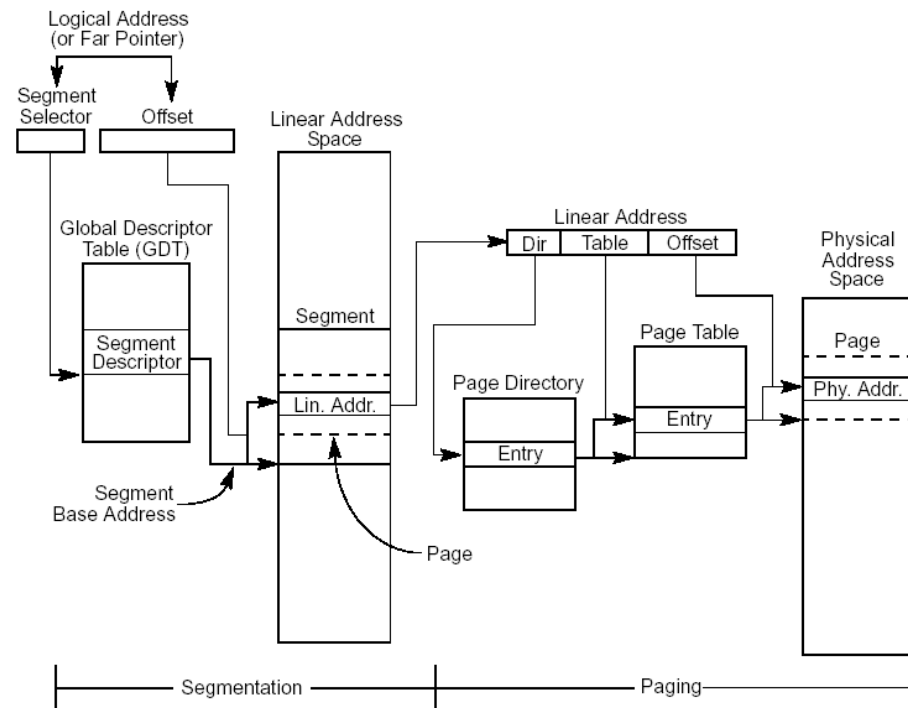
2.4 Translation Lookaside Buffer

- TLB와 Cache 연동



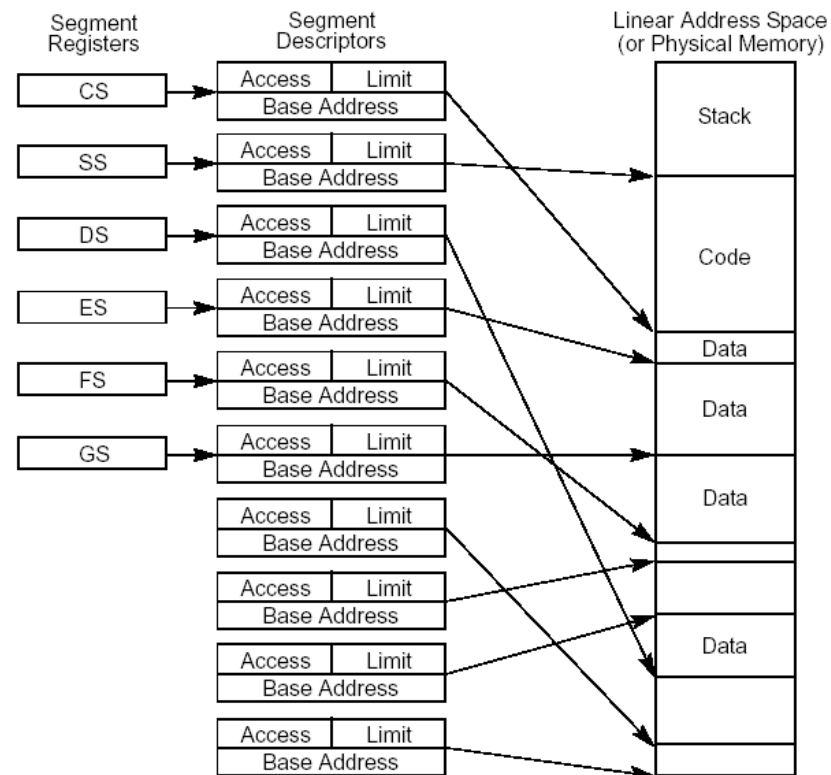
3. x86 프로세서의 가상 메모리

- x86 프로세서 가상 주소 변환 과정



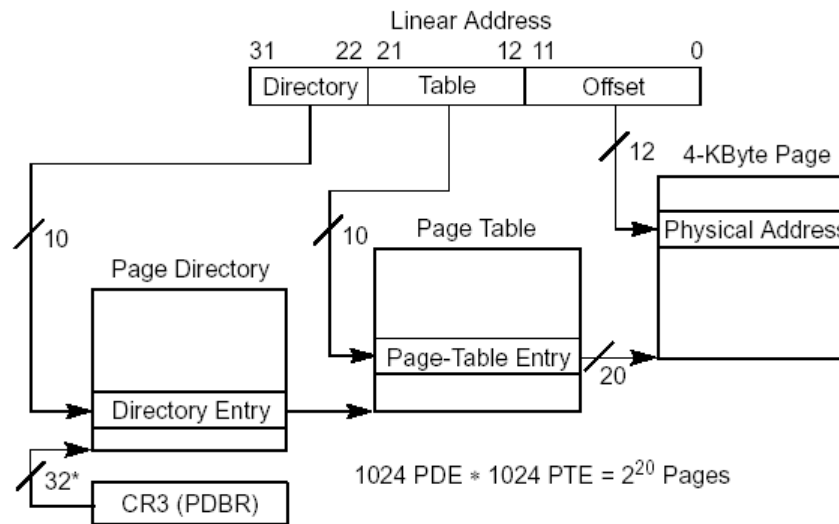
3. x86 프로세서의 가상 메모리

- x86 세그먼트 레지스터 사용



3. x86 프로세서의 가상 메모리

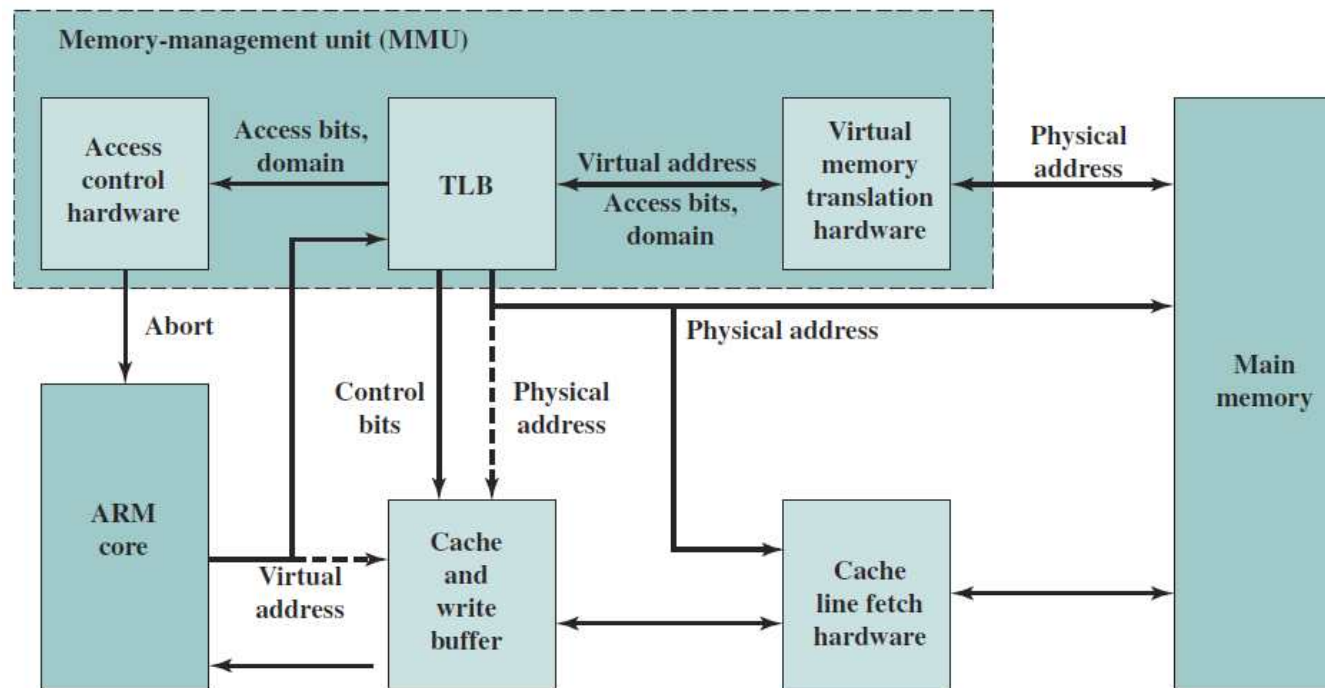
- x86 페이지 테이블 구조



*32 bits aligned onto a 4-KByte boundary.

4. ARM 프로세서의 가상 메모리

- ARM 프로세서의 메모리 시스템

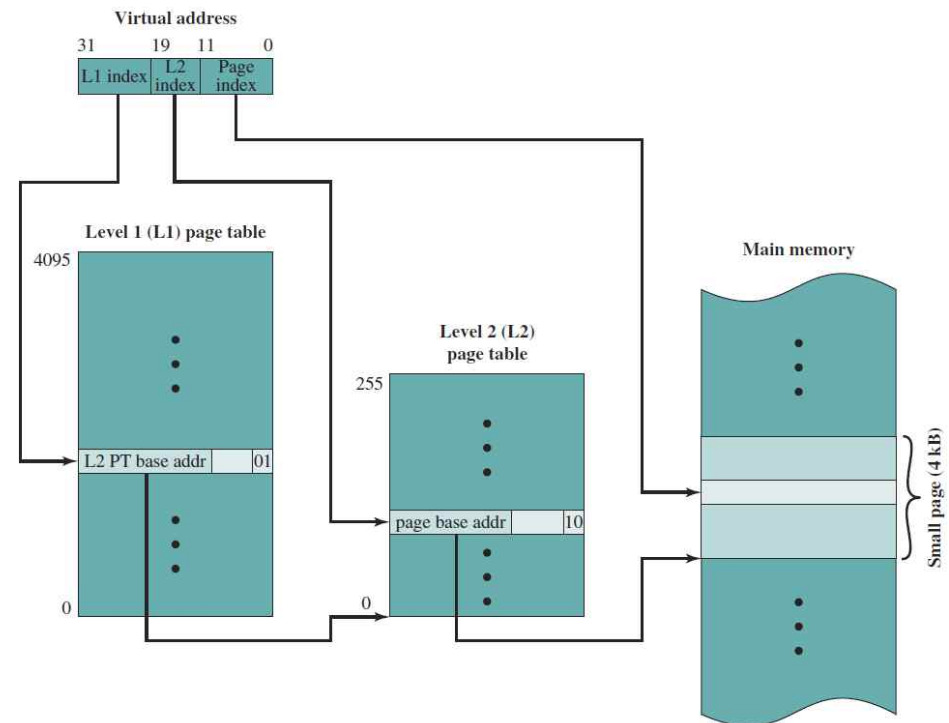


4. ARM 프로세서의 가상 메모리

- ARM 프로세서의 페이지 테이블
 - 2-level 변환

- 페이지 유형

- Supersection : 16 MB
- Section : 1 MB
- Large Page : 64 kB
- Small Page : 4 kB



end