Cache강좌(ARM946E-S)

작성자 : 안대현 (luisahn@mtis.co.kr)

1. 메모리 계층구조

컴퓨터 시스템에는 여러 형태의 기억 장치가 존재한다
대부분의 CPU에는 캐시 메모리가 존재하며 이는 CPU외부 버스를 사용하지 않으므로 다른 외부 메모리들에 비해 훨씬 빠른 속도로 접근이 가능하게 된다.
하지만 상당히 비싼 가격으로 인해 그 용량에 제한을 둘 수 밖에 없는 것이 현실이다.
캐시 다음으로 메인-메모리와 보조-메모리(HDD)등이 있을 수 있다.
메인-메모리는 CPU속도에는 못 미치지만 비교적 빠르게 동작한다.
인스트럭션이 메인-메모리에서 동작한다고 가정할 때, 빠른 CPU속도는 그보다 느린 메인-메모리에 의해 제한을 받을 수 밖에 없게 된다.
메인-메모리도 다른 메모리에 비해 가격이 높은 편이기 때문에 그 용량에 제한이 따른다.
이러한 기억용량의 제한을 해결하기 위해 보조-메모리를 두게 된다.
보조-메모리는 메인-메모리보다 훨씬 느리지만 가격이 싸기 때문에 대용량을 저장하기에 적합하다.
이러한 메모리 계층구조(memory hierarchy)를 4가지로 나누어 보면 아래와 같다.

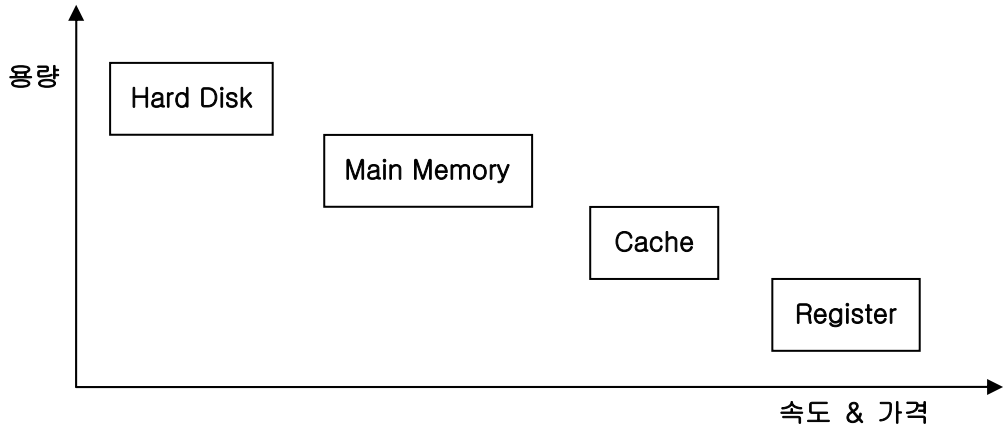
(1) 레지스터 = CPU자체 연산에 필요한 고속의 메모리

(2) 캐시 = CPU와 메인-메모리 간의 속도 차를 개선하기 사용.

(3) 메인-메모리 = 프로그램 실행에 필요한 명령어와 데이터들을 위치시키기 위한 메모리.

(4) 보조-메모리 = 대용량의 데이터를 보존하기 위해 사용. (비휘발성(nonvolatile))

그림 1.1은 이들 장치간의 용량, 속도, 가격에 대한 비교이다.



<그림 1.1> 메모리 비교

2. 캐시 개요

메인-메모리는 CPU에 비해 속도가 상당히 느리기 때문에 CPU가 모든 데이터를 매번 메인-메모리에서 읽어들이게 되면 CPU성능에 큰 손실을 얻게 된다.

이러한 손실을 줄이기 위해 프로그램에서 빈번히 사용되는 데이터(명령어포함)들을 메인-메모리에서 읽어와 캐시에 저장해 놓고, 이후 해당 데이터에 대해서는 메인-메모리를 참조하지 않고 캐시를 참조하게 됨으로써 큰 시스템 성능향상을 얻게 되는 것이다.

CPU가 참조하려는 데이터를 메인-메모리가 아닌 캐시로부터 로딩하려면 해당 데이터가 미리 캐시에 로딩되어 있어야 하겠다.

그러기 위해서는 CPU가 참조하려는 데이터가 무엇인지 예측이 가능해야 메인-메모리로부터 캐시로 미리 로딩해 놓을 수 있을 것이다.

일반적으로 프로그램이 수행될 때 최근에 사용한 인스트럭션과 데이터는 다시 사용될 가능성이 크다.

프로그램이 수행되는 데에는 이러한 지역성의 원칙(principle of locality)이 따르게 된다.

지역성에는 시간과 공간의 2가지 형태가 있다.

(1) **시간적 지역성(temporal locality)**

최근에 사용된 데이터는 가까운 시간 안에 다시 사용될 가능성이 크다.

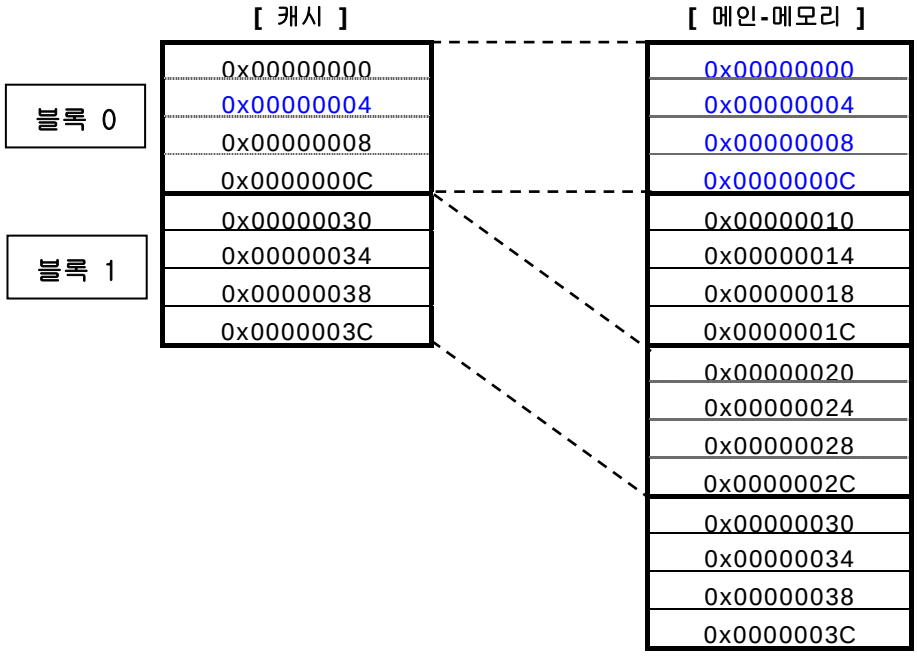
(2) **공간적 지역성(spatial locality)**

가까운 주소영역 내의 데이터들은 작은 시차로 접근될 가능성이 크다.

캐시는 이러한 지역성에 근거하여 CPU가 참조할 인스트럭션과 데이터를 미리 예측할 수 있게 되는 것이다.

지역성에 근거하면 CPU가 데이터를 로드 할 때, 해당 데이터의 주변 데이터도 가까운 시간 안에 다시 참조될 가능성이 크다.
이러한 공간적 지역성에 근거하여 캐시는 데이터 로딩 시, 그 주변 데이터들까지 함께 로딩 하는데 이때 로딩되는 단위를 캐시블록(block) 또는 캐시라인(line)이라고 한다.
캐시블록은 2워드(Word) 이상의 2의 누승으로 구성된다.
만약 캐시블록이 1워드라고 한다면 이는 공간적 지역성을 고려하지 않는 것이 된다.
참고로 ARM946E-S의 캐시블록은 8워드(32 Bytes)이다.

캐시블록이 4워드이고 0x00000004주소에 있는 1워드를 로드 한다고 가정하면,
해당 데이터가 캐시상에 존재하지 않을 때, 캐시는 메인-메모리로부터 0x00000000-0x0000000F의 데이터, 즉 해당 워드를 포함하고 있는 블록내의 데이터를 모두 로드 하게 된다. <그림 1.2참조>



<그림 2.1 캐시블록이 4워드인 캐시에서의 메인-메모리 데이터 로드>

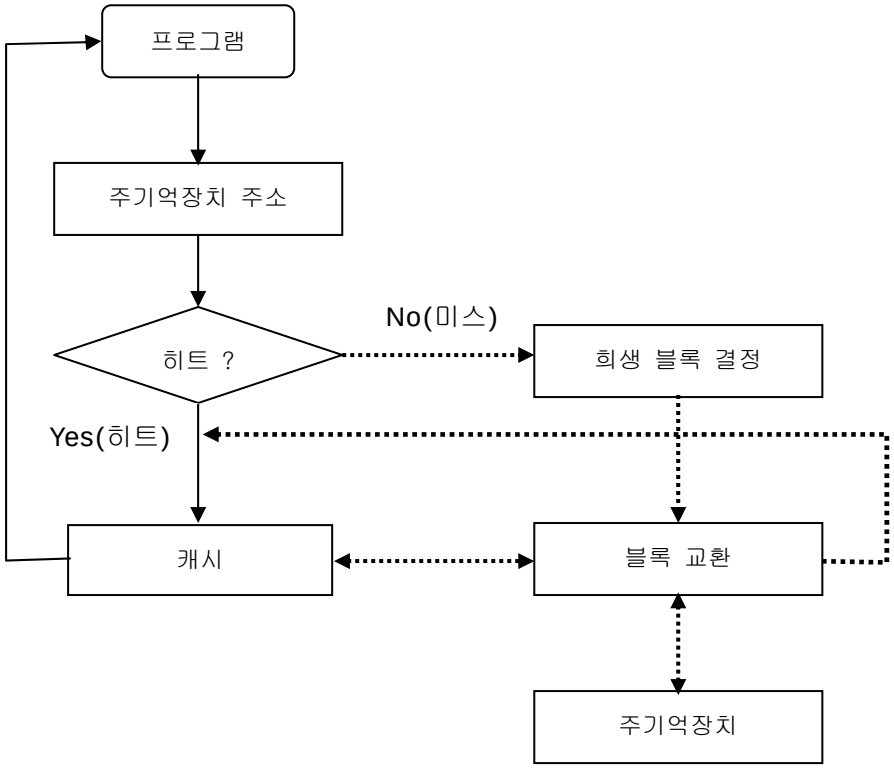
CPU에 의해 요구되는 데이터가 이미 캐시에 로딩 되어 있는 경우를 “캐시 히트(hit)”라고 하고, 캐시에 로딩 되어 있지 않은 경우를 “캐시 미스(miss)”라고 한다.
캐시 히트의 경우,
캐시 블록에 있는 데이터를 로딩 하면 된다.

캐시 미스의 경우,

메인-메모리로부터 데이터를 로딩 한다.

캐시가 이미 다른 데이터들로 모두 채워져 있다면 캐시블록 중 한 블록을 희생블록(victim block)으로 선택하게 되고 희생블록은 메인-메모리의 해당 블록과 블록교환(block replacement)를 하게 된다.

(블록교환 : 희생블록의 데이터를 메인-메모리에 저장하고 메인-메모리의 데이터를 캐시에 로드)



<그림 2.2 캐시 동작 : 실선 = 캐시 히트, 점선 = 캐시 미스>

<그림 2.2>를 보면 두 가지 의문이 생긴다.

첫째, 캐시 히트와 미스 여부를 무엇으로 판단 하는 것일까?

캐시에는 데이터 영역 이외에 태그(Tag)라는 것이 추가되어 있다.

이는 블록 데이터들에 대한 주소정보를 갖고 있으며 캐시내의 각 블록마다 태그가 존재한다.

따라서 메인-메모리로부터 요청된 주소와 캐시내부의 태그를 비교함으로써 히트 & 미스 여부를 판단할 수가 있는 것이다.

그러나 태그는 유효할 수도 있고 그렇지 않을 수도 있다.

무슨 말인고 하니, 최초 부팅 시 캐시는 비어 있을 것이며, 이때 태그를 비교하는 것은 무의미할 것이다.

이를 위해 각 블록마다 태그와 함께 별도의 Valid비트를 두어 해당 블록이 유효한 지 그렇지 않은 지를 판단하는 것이다.

둘째, 캐시 미스의 경우 희생 블록은 어떻게 결정되는 것일까?

이를 위해 일반적으로 LRU(Least Recently Used)방식과 랜덤(Random)방식이 사용된다.

LRU방식은 블록마다 별도의 카운터를 두어 오랫동안 사용되지 않은 블록을 선택하는 것이고, 랜덤 방식은 여러 블록 중에서 무작위로 선택하는 것이다.

ARM946E-S에서는 랜덤(Random)방식과 라운드-로빈(Round-Robin)방식을 지원하는 데, 라운드-로빈 방식은 블록마다 별도의 카운터를 두어 블록 할당 시마다 1씩 증가시키면서 다음 할당 블록을 결정하는 것이다.

메모리 블록을 캐시 블록에 할당하는 데에는 다음과 같은 방법이 있다.

(1) **직접 매핑(direct mapped)**

메모리의 각 블록이 캐시의 특정 블록에 고정적으로 할당 된다.

(2) **완전 연합(fully associative)**

메모리의 각 블록이 캐시의 모든 블록에 할당 될 수 있다.

(3) **세트 연합(set associative)**

전체 캐시 영역을 세트로 구분하여 메모리의 각 블록을 캐시의 한 세트내의 모든 위치에 할당 될 수 있다.

<직접 매핑>

메인-메모리 내의 각 블록이 캐시블록과 직접적으로 매핑되어 있다.

즉, 메인-메모리의 특정 블록은 캐시내의 한 블록에 정확히 대응되는 것이다.

메인-메모리로부터의 블록을 캐시의 어느 블록에 할당할 지를 선택하기 위해 아래의 식을 따른다.

캐시 블록 번호 = (메인-메모리 블록 번호) Modulo (캐시 블록 수)

Modulo연산에 의해 메인-메모리 블록 0, 4, 8, 12는 캐시내의 같은 블록에 할당 되는 것이다.

<그림 2.3 직접 매핑>

<완전 연합>

메인-메모리 내의 각 블록이 캐시내의 어느 블록에든지 할당 될 수 있다.

<그림 2.4 완전연합>

<세트 연합>

직접매핑과 완전연합의 중간 방식이며 n-way세트연합이라고 부른다.

전체 캐시 블록이 n개의 블록을 가진 여러 개의 세트로 구성된다.

즉, 메인-메모리의 한 블록은 캐시내의 n개의 블록에 할당될 수가 있는 것이다.

n개 중 어느 블록에 할당 될지는 캐시 알고리즘에 의해 적용될 것이다. (LRU, 랜덤…)

사실 직접매핑 캐시는 1-way세트연합과 같은 의미이며, m개의 블록으로 구성된 완전연합 캐시는 m-way세트연합과 같은 의미이다.

즉, 직접매핑은 전체 블록 수만큼의 세트가 존재하고, 완전연합은 한 개의 세트가 존재하는 샘인 것이다.

세트 연합은 메인-메모리로부터의 블록을 캐시의 어느 위치에 할당할 지를 선택하기 위해 먼저 아래의 수식에 따라 해당 블록을 포함하고 있는 세트를 찾게 되며 블록은 계산된 세트 내의 어느 블록에든지 할당 될 수 있다.

캐시 블록을 포함하는 세트 = (메인-메모리 블록 번호) Modulo (캐시 세트 수)

<그림 2.5 세트연합>

참고로 ARM946E-S는 4-way세트연합 캐시이다.

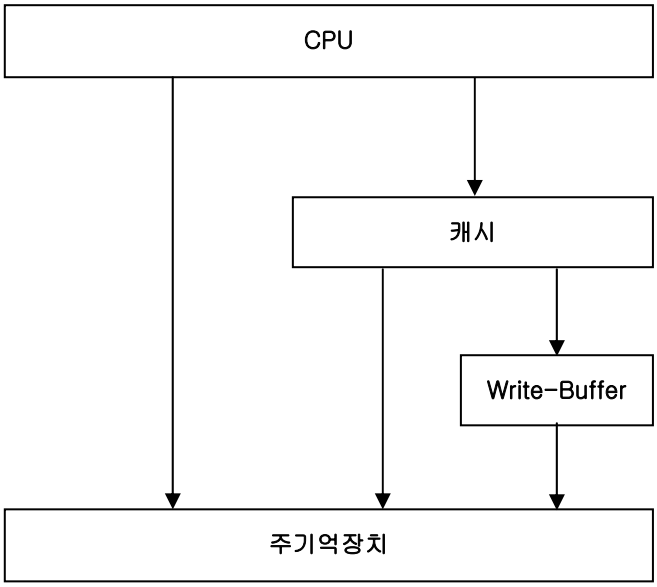
세트연합의 경우, 캐시 블록을 포함하는 세트가 선택되면 세트내의 어느 블록에 할당할 지를 결정해야 한다. 또한 완전연합의 경우에도 메인-메모리의 모든 블록이 캐시의 모든 블록에 할당될 수 있기 때문에 어느 블록에 할당될 지를 결정해야 한다.

이를 위한 방법으로 앞서 언급한 바 있는 랜덤, LRU, RR등의 기법이 사용된다.

(직접매핑의 경우 이미 각 블록이 매핑되어 있으므로 고려되지 않음)

ARM946E-S의 경우 4-way세트연합이므로 한 세트는 4블록으로 구성되며 메인-메모리의 한 블록은 이 4블록(세트)중 한 블록에 할당 될 수 있는 것이다.

지금까지 캐시 Read의 경우를 살펴보았는데 Write의 경우는 어떻게 될까?
캐시는 사실 Read를 위한 장치이다.
거의 모든 프로그램들은 메모리로부터의 Write접근보다 Read접근이 훨씬 많다.
인스트럭션을 수행하는 것 자체가 Read접근이며 데이터의 경우도 대부분 Read접근이기 때문이다.



<그림 2.6 캐시와 Write-Buffer를 통한 CPU의 메인-메모리 접근>

Write의 경우 2가지의 방법으로 나눌 수 있다.

(1) Write-Though

캐시와 메인-메모리 모두에 저장한다.

(2) Write-Back

캐시에만 저장하고 메인-메모리에는 저장하지 않는다.

<Write-Though>

데이터 저장 시에 캐시 히트와 미스의 경우에 대해 각각 생각해 볼 수 있다.
캐시 히트의 경우

캐시에 저장 시킴과 동시에 메인-메모리에도 저장시킨다.

캐시 미스 : 다시 두 가지 방식이 사용될 수 있다.

(1) 희생블록을 선택하여 블록교환을 함과 동시에 데이터를 캐시와 메인-메모리에 모두 저장시킨다.

(2) 저장할 데이터의 블록을 캐시로 가져오지 않고 메인-메모리에만 저장한다.

(ARM946E-S에서는 (2)번 방식을 사용한다.)

<Write-Back>

데이터 저장 시에 캐시 히트와 미스의 경우에 대해 각각 생각해 볼 수 있다.

캐시 히트의 경우

(1) 캐시에만 저장하고 메인-메모리에는 저장하지 않는다.

(2) 블록교환이 발생하게 되면 해당 블록을 메인-메모리로 쓰게 된다.

캐시 미스의 경우

(1) 희생블록을 선택하여 블록교환을 함과 동시에 데이터를 캐시에만 저장시킨다.

(2) 저장할 데이터의 블록을 캐시로 가져오지 않고 메인-메모리에만 저장한다.

(ARM946E-S에서는 (2)번 방식을 사용한다.)

Write-Back의 경우 데이터를 캐시에만 저장시키기 때문에 이후 블록교환이 발생하면 변경된 데이터를 메인-메모리에 저장해야 한다.

하지만 블록교환이 일어난다고 해서 항상 데이터를 메인-메모리에 저장시키는 것은 아니다.

캐시내의 각 블록마다 별도의 Dirty비트라는 것을 두어 블록내의 데이터에 저장이 발생하게 되면 이 Dirty 비트를 세팅 시켜두고 이후 블록교환이 발생하면 Dirty비트를 참조하여 저장이 발생한 블록에 대해서만 메인-메모리로 저장하게 된다.

CPU가 캐시에 데이터를 쓰는 것은 시간비용이 많이 들지 않는다.

하지만 캐시에서 메인-메모리로의 데이터 저장은 시간비용이 상당히 많이 든다.

이 시간비용을 줄이기 위해 존재하는 것이 Write-Buffer이다.

데이터 저장이 발생하게 되면 캐시는 데이터를 Write-Buffer에 저장하고 바로 저장작업을 마칠 수 있다.

이후 Write-Buffer가 메인-메모리로 데이터를 저장하게 된다.

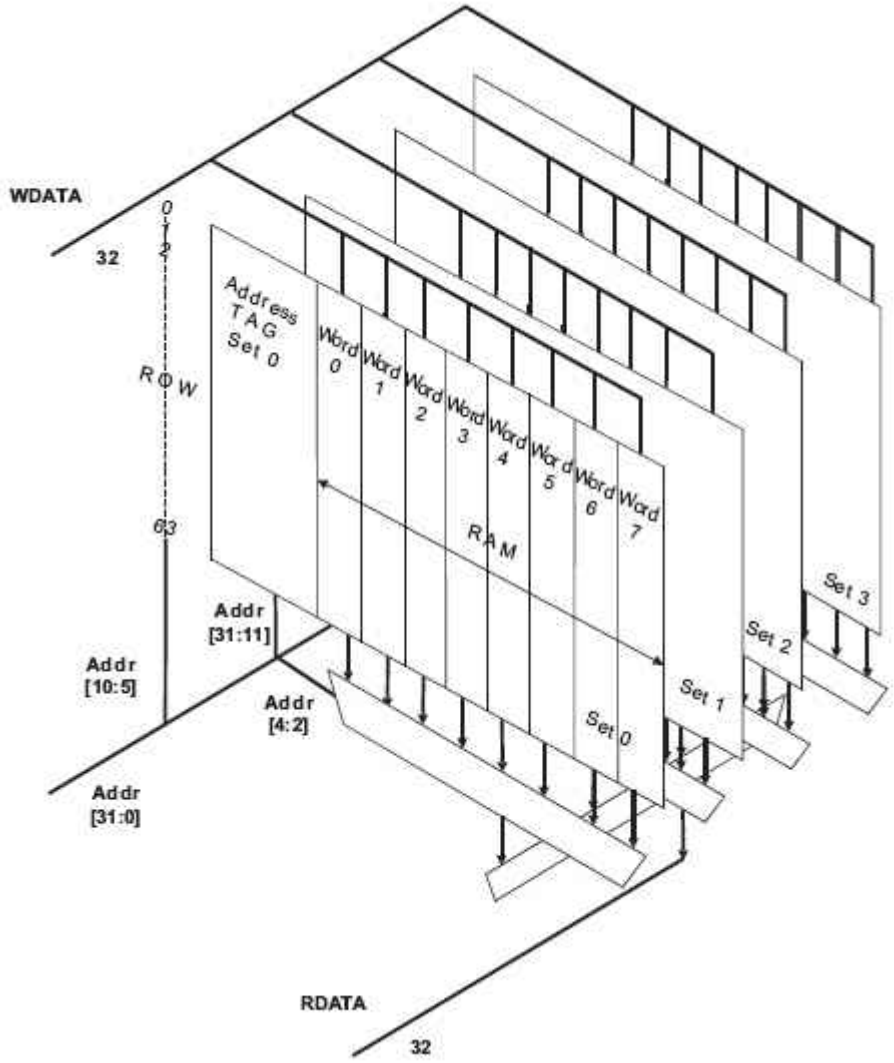
3. 캐시 실례 (ARM946E-S)

ARM946E-S는 Instruction Cache(이하 ICache)와 Data Cache(이하 DCache)가 각각 구분되어 있으며 각 캐시의 크기는 서로 다를 수 있다.

앞서 언급한 바 있듯이 ARM946E-S의 캐시는 4-way세트연합을 사용하며 한 블록은 8워드(32bytes)의 크기를 갖는다. <그림 3.1참조>

8KB 캐시의 경우, 각 세트 내에 4개의 블록(=4-way)과 64개의 세트로 구성된다.

=> 8KB = (4 /*way*/) * (64 /*sets*/) * (32 /*block size*/)



<그림 3.1 8KB 크기의 캐시>

캐시를 접근할 때의 주소포맷은 아래와 같다.

N	9	10	11	12	13	14	15
캐시 크기	4KB	8KB	16KB	32KB	64KB	128KB	256KB

31 ~ N+1	N ~ 5	4 ~ 2	1 ~ 0
태그	**세트 인덱스**	**워드 인덱스**	**바이트**

<그림 3.1>이 8KB캐시를 예로 든 것이므로 Addr[10:5]가 세트 인덱스가 되고 Addr[31:11]가 태그가 되는 것이다.

태그 = 메인-메모리의 블록과 캐시내의 블록이 일치하는 지 비교하기 위해 사용.

세트 인덱스 = 여러 세트들 중 해당 블록이 포함된 세트를 선택하기 위해 사용.

워드 인덱스 = 블록이 8워드로 구성되어 있으므로 블록내의 워드를 선택하기 위해 사용.

세트 내의 4블록 중 한 블록을 선택하는 것은 블록 교환 알고리즘을 따른다. (랜덤, RR)

ARM에서 캐시를 컨트롤 하기 위해 시스템 컨트롤 코프로세서라 불리는 CP15를 엑세스 해야 한다.
이를 위한 명령어로 MRC, MCR이 사용된다.
또한 ARM946E-S에는 MPU(Memory Protection Unit)라는 것이 있는데 MPU는 8개의 영역에 대해 엑세스 권한과 캐시 & Write-Buffer의 사용 유무를 설정할 수 있다.
여기서 MPU에 대해서는 언급하지 않겠지만 특별히 어려운 개념이 아니므로 관련 문서를 참고하기 바란다.
캐시를 Enable하기 위해서는 반드시 MPU의 8개의 영역 중 최소 하나 이상의 영역이 설정되어 있어야 하며 MPU역시 Enable되어야 한다.
CP15의 c1레지스터는 컨트롤 레지스터로써 c1의 Bit[12], Bit[2]가 각각 ICache와 DCache를 Enable/Disable 시킨다.
ICache를 Enable시키는 명령은 아래와 같다.

```
MRC   p15, 0, R0, c1, c0, 0        ; read control register
ORR   r0, r0, #(1<<12)             ; Bit[12] : ICache enable bit
MCR   p15, 0, Rd , c1, c0, 0       ; write control register
```

ICache를 사용하는 것은 코드 수행에 있어 대단한 효율을 얻게 된다.
예를 들어, memcpy()함수는 아래와 같이 구현될 수 있으며 이어 생성된 어셈블리어 코드를 보여준다.
(RVCT 2.0에서 컴파일 됨.)

```
void memcpy( void *dst, const void *src, size_t len )
{
   unsigned char *s1 = dst;
   unsigned char *s2 = src;

   while ( len-- )
   {
      *s1++ = *s2++;
   }
}
```

```
memcpy
    SUB         r2, r2, #1
    CMN         r2, #1
    LDRNEB   r3, [r1], #1
    STRNEB   r3, [r0], #1
    BNE         memcpy
    BX          r14
```

memcpy()함수는 6 * 4의 총 24바이트의 코드를 생성하며 이는 캐시의 한 블록에 모두 들어 갈 수 있는 크기이다.
따라서 memcpy()함수를 수행 시 함수전체를 캐시 블록에 로딩할 수 있으며 함수 수행을 위한 인스트럭션

인출(fetch)이 모두 캐시로부터 가능하게 되는 것이다.
만약 ICache를 사용하지 않는다면 매번 메인-메모리로부터 인스트럭션을 인출해야 하므로 상당한 시간비용이 소요될 것이다.
특히, memcpy()함수와 같이 동일 루틴의 루프 횟수가 상당히 많은 코드에서 ICache는 대단한 효율을 발휘할 수 있는 것이다.

DCache를 Enable시키는 명령은 아래와 같다.

```
MRC p15, 0, Rd, c1, c0, 0        ; read Control Register
ORR  r0, r0, #(1<<2)             ; bit[2] : DCache Enable/Disable
MCR p15, 0, Rd , c1, c0, 0       ; write Control Register
```

위 memcpy()함수에서 DCache역시 시스템의 성능의 효율을 높인다.
0x00000000번지의 데이터를 0x100크기만큼 복사한다고 가정해 보자.
데이터를 0x00000000번지에서 1바이트 읽어 들이게 되면 캐시는 메인-메모리로부터 블록사이즈(32bytes)만큼을 한꺼번에 읽어 들인다.
따라서 0x00000000번지를 읽고 난 이후부터 0x0000001F까지의 데이터는 모두 캐시에서 읽어 오게 되는 것이다.
이러한 동작에는 여러 가지 환경적인 요인이 시스템 성능에 영향을 줄 수가 있다.
바이트 단위로 32번을 읽어오는 행위와 32바이트를 한꺼번에 읽어오는 행위는 시스템에 따라 다를 수 있겠지만 거의 대부분의 시스템에서 32바이트를 한꺼번에 읽어오는 것이 더 효율적일 것이다.
또한 ARM946E-S에서는 스트리밍을 지원한다.
스트리밍이란 캐시가 캐시 블록을 채우는 동시에 코어가 요청한 데이터를 로딩해 주는 것을 말한다.
0x00000000번지에서 한 바이트를 읽을 경우, 캐시는 해당 블록(32바이트)을 로딩하게 되는데 코어가 요청한 0x00000000번지의 데이터를 메인-메모리로부터 로딩함과 동시에 코어에게 전달해 줌으로써 코어는 다음 명령을 수행할 수가 있게 되며 캐시는 나머지 데이터를 로딩하게 되는 것이다.
만약, 이때 캐시가 비어 있지 않다면 로딩작업에 앞서 캐시미스에 블록 교환에 의한 추가 손실이 발생할 것이다.

Write동작에 있어서 ARM946E-S는 Write-Though와 Write-Back을 모두 지원한다.
이를 위해 c2레지스터(Cache Configuration Registers)와 c3레지스터(Write Buffer Control Register)가 존재한다.
c2와 c3의 조합으로 MPU의 각 영역에 대해 Write-Though와 Write-Back을 별도로 설정할 수 있다.
Write동작은 아래의 설정에 따라 결정된다.

	Cache Control	**Write-Buffer Control**
Write-Though	Enable	Disable
Write-Back	Enable	Enable

MPU 영역0에 대해 Write-Back을 설정하는 명령은 다음과 같다.

```
MRC   p15, 0, Rd, c2, c0, 0        ; read Cache Configuration
ORR   r0, r0, #(1<<0)              ; bit[0] : Region 0
MCR   p15, 0, Rd, c2, c0, 0        ; DCache Enable
MRC   p15, 0, Rd, c3, c0, 0        ; read Write-Buffer Configuration
ORR   r0, r0, #(1<<0)              ; bit[0] : Region 0
MCR   p15, 0, Rd, c3, c0, 0        ; write-Buffer Enable
```

Write-Buffer를 사용할 경우,
메인-메모리로의 Write동작은 Write-Buffer로 이루어지며, 이후 Write-Buffer는 코어를 정지시키지 않고 해당 데이터를 메인-메모리로 전달한다.

ARM946E-S는 캐시 할당에 있어 “Read-allocate”를 사용한다.
이것은 Read동작에 대해서만 캐시블록을 할당하는 것이다.
캐시 미스의 경우, Write할 데이터가 캐시내에 존재하지 않기 때문에 해당 블록을 캐시로 로딩하느냐 그렇지 않으냐의 문제가 생기는데 Read-allocate는 캐시로 로딩하지 않고 메인-메모리에만 Write하는 것이다.

캐시의 Read와 Write동작에는 Flush와 Clean이라는 중요한 개념이 있다.
앞서 캐시의 Valid비트와 Dirty비트에 대해서 언급한 바 있듯이 Valid비트는 캐시블록의 데이터가 유효하다는 것을 알려주며 Dirty비트는 데이터가 변경되었다는 것을 알려준다.
Flush란 캐시내의 데이터가 유효하지 않다는 것을 알려주며 Clean이란 캐시내의 데이터를 메인-메모리로 저장하는(비우는) 작업을 말한다.
Valid비트를 0으로 세팅하므로써 이후 해당 블록을 엑세스시 캐시 미스를 발생시킬 것이다.
Dirty비트를 0으로 세팅하게 되면 캐시는 해당 블록을 메인-메모리에 저장하게 된다.

RAM(메인-메모리)상에서 동작하는 코드들은 수행되기 이전에 ROM(보조-메모리)으로부터 코드를 로딩해야 한다.
이후 RAM상의 코드를 수정할 경우 간혹 문제가 발생한다.
코드를 수정하기 위해 RAM에 코드 데이터를 저장하게 되면 이것은 실제로 DCache에 저장이 될 것이다.
이후 RAM코드를 실행하면 이는 수정된 RAM코드가 실행되는 것이 아니라 ICache내에 있던 기존 코드가 실행될 것이다.
이러한 문제를 막기 위해 수정할 코드 데이터를 RAM(실제로는 DCache)에 저장한 이후에 DCache를 Clean해 주어야 한다.
이로써 수정된 코드 데이터는 DCache로부터 RAM에 저장이 될 것이다.
또한 수정된 코드를 실행하기에 앞서 ICache를 Flush해 주어야 한다.
수정된 코드를 실제로 RAM에 저장되어 있지만 코드는 여전히 ICache내에 있던 기존 코드를 실행할 수 있기 때문이다.

ICache를 Flush하기 위한 명령은 아래와 같다.

```
MOV R0, #0
MCR p15, 0, R0, c7, c5, 0
```

DCache를 Flush하기 위한 명령은 아래와 같다.

```
MOV R0, #0
MCR p15, 0, R0, c7, c6, 0
```

Write-Though의 경우는 캐시와 메인-메모리가 항상 일치하므로 Clean작업이 필요하지 않다.
하지만 Write-Back의 경우, Flush에 앞서 Clean작업이 수행되어야 한다.
DCache를 Clean & Flush하기 위해 ARM946E-S는 각 블록에 대해 명령을 수행해야 한다.
아래 코드는 인라인 어셈블리어로 작성되어 이해가 쉬우리라 생각하고 설명은 생략한다.

```
void CleanFlushDCache( void )
{
   unsigned int way=0, Rd, line;
   const unsigned int size=0x1000; // 16KB DCach

   do {
      line = 0;

      do {
         Rd = way | line;
         __asm { MCR   p15, 0, Rd, c7, c14, 2 }    // clean & Flush the line

         line += 0x20;    // next line
      } while ( line <= size );    // complete all entries in one way?

      way += 0x40000000;    // next way
   } while ( way );
}
```

또한 데이터의 주소를 이용하여 하나의 캐시블록을 Clean & Flush 할 수도 있다.

```
MOV R0, =Address
MCR p15, 0, R0, c7, c14, 1
```

ARM946E-S는 인터럽트 처리루틴과 같이 빠른 엑세스를 필요로 하는 코드 및 데이터들의 빠른 수행을 위해 캐시 락다운(Lockdown)이라는 것을 지원한다.

ARM946E-S는 1-way단위로 락다운 영역이 할당 되는데 이 영역은 캐시 동작에 있어 블록 교환이 일어나지 않기 때문에 캐시 미스가 발생하지 않는다.
캐시 락다운은 S/W적으로 쉽게 구현이 되며 몇 가지만 주의하면 된다.
첫째, 락다운 명령은 캐시 설정이 되어 있지 않은 영역에서 수행되어야 한다.
둘째, 캐시는 Enable되어야 하며 인터럽트는 Disable되어야 한다.
셋째, 락다운을 하기위한 코드 및 데이터는 이전에 캐시에 로딩되어 있지 않아야 한다.
넷째, 리셋 이후에 캐시가 사용되었다면 Clean 또는 Flush를 해 주어야 한다.
참고로 캐시 전체를 락다운 할 수는 없다. (일반적인 캐시 동작을 위해 하나 이상의 way를 남겨두어야 한다.)

ICache 락다운을 위해 c7 & c9 레지스터가 사용되며, DCache 락다운을 위해 c9레지스터가 사용된다.

```
MCR p15, 0, Rd, c9, c0, 0;    write data lockdown
MRC p15, 0, Rd, c9, c0, 0;    read data lockdown
MCR p15, 0, Rd, c9, c0, 1;    write instruction lockdown
MRC p15, 0, Rd, c9, c0, 1;    read instruction lockdown
```

C9레지스터는 어느 Way를 락다운 할 것인 지를 결정한다.
C9레지스터의 포맷은 아래와 같다.

31	30 ... 2	1 ... 0
L	**SBZ** (Should be zero)	**Way**

L : 락다운 코드 및 데이터를 로딩하겠다는 의미. (로딩 이전에 1로 셋팅, 로딩 후 0으로 셋팅)
Way : 4개의 Way중 락다운 할 Way를 결정한다.

DCache의 경우, 락다운을 위해 데이터를 캐시로 로딩하는 것은 LDR명령으로 수행된다.
하지만 ICache의 경우는 C7레지스터를 사용하여 인스트럭션을 로딩한다.

```
__asm{ MCR p15, 0, Rd, c7, c13, 1 }    // Rd = instruction's address
```

아래의 코드는 ICache를 락다운하는 코드이다.
인라인 어셈블리어로 작성되어 이해가 쉬우리라 생각하고 설명은 생략한다.
DCache의 경우도 데이터 로딩 명령 이외에는 동일하므로 DCache코드는 생략한다.

```
unsigned int LockDownICache( unsigned int startAddr, unsigned int endAddr )
{
   #define N_WAY 4    // 4-way way associative
   #define ALIGN_CACHELINE 0xFFFFFFE0
   #define SIZE_PER_WAY 0x1000    //16KB
```

```
#define CACHELINE_SIZE 32 // byte
#define LOCKDOWN_BIT ((unsigned int)(1<<31))
#define WAY_BITS 0x03
unsigned int Rd;
unsigned int addr;
unsigned int way_num;
unsigned int way_endAddr;

endAddr -= CACHELINE_SIZE;
if ( startAddr > endAddr ) return 0xFFFFFFFF;

way_num = ((endAddr - startAddr) / SIZE_PER_WAY) + 1;
if ( way_num > N_WAY ) return 0xFFFFFFFF;

addr = startAddr & ALIGN_CACHELINE;

__asm{ MRC p15, 0, Rd, c9, c0, 1 }     //get current ICache index

Rd &= WAY_BITS;
if ( (N_WAY - Rd) < way_num ) // check for available way
    return 0xFFFFFFFF;

Rd |= LOCKDOWN_BIT;    // enter lockdown mode

while ( way_num-- )
{
    __asm{ MCR p15, 0, Rd, c9, c0, 1 }     //write instruction lockdown

    if ( way_num == 0 )
        way_endAddr = endAddr + CACHELINE_SIZE;
    else
        way_endAddr = addr + SIZE_PER_WAY;

    while ( addr < way_endAddr )
    {
        __asm{ MCR p15, 0, addr, c7, c13, 1 }     //prefetch instruction
        addr += CACHELINE_SIZE;
    }
```

```
        Rd++;    //increment ICache index
    }

    Rd &= ~LOCKDOWN_BIT;   // exit lockdown mode
    __asm{ MCR p15, 0, Rd, c9, c0, 1 }    //write instruction lockdown

    return Rd;
}
```

ARM946E-S에는 디버깅을 위해 c15레지스터가 존재한다.
c15를 컨트롤 하여 캐시내의 각 블록(라인)의 태그와 데이터를 Read & Write할 수 있으며 이때 각 블록을 접근하기 위해 인자로 전달되는 레지스터의 포맷은 아래와 같다.

31 30 29 ~ N+1	N ~ 5	4 ~ 2	1 ~ 0
Way / **SBZ**	**Index**	**Word**	**SBZ**

또한, 태그의 포맷은 아래와 같다.

31 ~ N+1	N ~ 5	4	3	2 ~ 1	0
TAG address	**Index**	**Valid**	**Dirty**	**Way**	

마지막으로 캐시내의 태그 및 데이터를 덤프하는 코드를 소개하며 강좌를 마치겠다.
역시 별로 어려운 코드가 아니므로 충분히 이해할 수 있으리라 생각되어 설명은 생략한다.

```
#define WriteCacheIndex(Rd)   __asm{ MCR p15, 3, Rd, c15, c0, 0}
#define ReadCacheIndex(Rd)    __asm{ MRC p15, 3, Rd, c15, c0, 0}
#define WriteICacheTag(Rd)    __asm{ MCR p15, 3, Rd, c15, c1, 0}
#define ReadICacheTag(Rd)     __asm{ MRC p15, 3, Rd, c15, c1, 0}
#define WriteDCacheTag(Rd)    __asm{ MCR p15, 3, Rd, c15, c2, 0}
#define ReadDCacheTag(Rd)     __asm{ MRC p15, 3, Rd, c15, c2, 0}
#define WriteICacheData(Rd)   __asm{ MCR p15, 3, Rd, c15, c3, 0}
#define ReadICacheData(Rd)    __asm{ MRC p15, 3, Rd, c15, c3, 0}
#define WriteDCacheData(Rd)   __asm{ MCR p15, 3, Rd, c15, c4, 0}
#define ReadDCacheData(Rd)    __asm{ MRC p15, 3, Rd, c15, c4, 0}

typedef struct
{
```

```
    unsigned int tag;
    unsigned int data;
} CACHE_ENTRY;

void DumpICache( CACHE_ENTRY *entry )
{
    #define N_WAY 4
    #define CACHELINE 8
    unsigned int way, set_index, word_addr;
    unsigned int way_i, index_i, word_i;

    unsigned int tag, data;
    unsigned int way_set_word;
    unsigned int index_num;
    unsigned int Rd;

    CACHE_ENTRY *pCacheBuf;

    pCacheBuf = (CACHE_ENTRY*)&entry;

    __asm { MRC p15, 0, Rd, c0, c0, 1 }
    Rd = (Rd >>6) & 0x0f;
    index_num = (1<<(Rd+2));    // calculate index from size

    for ( way_i = 0; way_i < N_WAY; way_i++ )
    {
        way = way_i << 30;

        for ( index_i = 0; index_i < index_num; index_i++ )
        {
            set_index = index_i << 5;

            WriteCacheIndex( way | set_index );
            ReadICacheTag(tag);

            for( word_i = 0; word_i < CACHELINE; word_i++ )
            {
                word_addr = word_i << 2;
```

```
                way_set_word = way | set_index | word_addr;

                WriteCacheIndex( way_set_word );
                ReadICacheData( data );

                pCacheBuf->tag = tag;
                pCacheBuf->data = data;
                pCacheBuf++;
            }
        }
    }
}
```

[참고 문헌]

컴퓨터 구조학 - 조정완

ARM Architecture Reference Manual

ARM946E-S Technical Reference Manual