**CSED211 Homework4**

20210774 김주은

**1. Exercise 6.26 on page 686.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cache | m | C | B | E | S | t | s | b |
| 1. | 32 | 2048 | 8 | 1 | 256 | 21 | 8 | 3 |
| 2. | 32 | 2048 | 4 | 4 | 128 | 23 | 7 | 2 |
| 3. | 32 | 1024 | 2 | 8 | 64 | 25 | 6 | 1 |
| 4. | 32 | 1024 | 32 | 2 | 16 | 23 | 4 | 5 |

**2. Exercise 6.28 on page 686.**

<practice problem 6.12>

테이블이(가) 표시된 사진

자동 생성된 설명

m = 13으로 13bit으로 memory address가 이루어져 있다. s = 3이고, b = 2이기 때문에 tag bit은 8bit으로 이루어져 있다.

그러므로, memory address 총 13bit가 있다면,

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

이라고 할 수 있고, 12~5 로 numbering되어 있는 곳에는 cache tag bit으로 이루어져 있고, 나머지 4,3,2 자리를 set index bit이 차지하고, 마지막 1,0 자리에 block offset bit이 위치하게 된다.

테이블이(가) 표시된 사진

자동 생성된 설명

Cache 내용은 위와 같다.

A.

set2에 있으려면 set index가 2이어야 한다. 하지만 위 table에서는 set index가 2인 line 2개 모두 valid가 0이므로 set2에서 캐시 hit이 되는 memory address는 존재하지 않는다.

B.

Set index 4를 보게 되면, 두 개의 line 모두 valid함을 알 수 있다. Tag bit, set bit, block offset bit 순서대로 적어보면 각 대응되는 memory address를 구할 수 있다.

먼저, line0의 경우는 tag bit이 C7이고 이를 2진수로 바꾸면 11000111이다.

Set index bit은 100이고, block offset은 0~3일 것이다. 이를 다 합치면

1100011110000, 1100011110001, 1100011110010, 1100011110011 이다.

이를 16진수로 정리하면 0x18F0 ~ 0x18F3이다.

Line1의 경우는 tag bit이 05이고, 이를 2진수로 바꾸면 00000101이다.

Set index bit은 100이고, block offset까지 합치면,

0000010110000, 0000010110001, 0000010110010, 0000010110011 이다.

이를 16진수로 변환하면 0x00B0 ~ 0x00B3이다.

C.

Set5이기 때문에 Set index는 101이다. Set 5의 경우는 line0만 valid가 1이므로 line0에 대응되는 주소만 계산한다. Tag bit은 71이므로 이를 이진수로 바꾸면 01110001이다. Set index bit과 block offset bit까지 모두 합쳐 나열하면,

0111000110100, 0111000110101, 0111000110110, 0111000110111이다.

이를 16진수로 변환하면 0x0E34~ 0x0E37이다.

D.

Set7이기 때문에 Set index가 111이다. Set 7의 경우 line1만 valid가 1이며 tag bit은 DE다. 여기에 set index bit, block offset bit 모두 합쳐서 나열하면,

1101111011100, 1101111011101, 1101111011110, 1101111011111 이다.

이를 16진수로 변환하면 0x1BDC ~ 0x1BDF이다.

**3. Exercise 6.31 ~ 6.32 on pages 688 ~ 689.**

Problem 6.31)

테이블이(가) 표시된 사진

자동 생성된 설명

Address : 0x071A

B = 4, S = 8이기 때문에 s = 3, b = 2 이다.

그러므로 tag bit은 13bit에서 나머지부분인 8 bit이다.

A. Address format (1 bit per box)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |

B. Memory reference

|  |  |
| --- | --- |
| Parameter | Value |
| Block offset(CO) | 0x02 |
| Index(CI) | 0x06 |
| Cache tag(CT) | 0x38 |
| Cache hit | Yes |
| Cache byte returned | 0XEB |

Problem 6.32)

Address : 0x16E8

A. Address format (1 bit per box)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

B. Memory reference

|  |  |
| --- | --- |
| Parameter | Value |
| Block offset(CO) | 0x00 |
| Index(CI) | 0x02 |
| Cache tag(CT) | 0xB7 |
| Cache hit | No |
| Cache byte returned | X |

**4. Exercise 6.36 on page 690**

int x[2][128];

int i;

int sum = 0;

for (i = 0; i < 128; i++) {

sum += x[0][i] \* x[1][i];

}

A. Case1 : Assume the cache is 512 bytes, direct-mapped, with 16-byte cache blocks.

C = 512, E = 1, B = 16이다.

C = E\*B\*S 이기 때문에 S = 32 이다.

Sum에 값을 더할 때 x[0][i]와 x[1][i] 값을 각각 load해오는데, 이 때의 miss rate를 계산하기 위해 cache에 어떻게 배열의 값들이 캐시에 로드되는지 확인해본다.

2차원 배열이며 총 2행에 128열로 이루어져 있고, 이는 문제에서 제시된 대로 row-major order로 저장된다. 즉, x[0][i]는 0x0부터 배열이 시작한다고 한다면 4i 가 주솟값이 될 것이다. X[1][i]는 128\*4 + 4i = 512 + 4i의 주소에서 로드될 것이다.

그러나 C = 512이며, E = 1 이기 때문에 4i 와 512+4i의 주소가 캐시에서 같은 블록에 대응된다. 즉, 캐시가 비어있을 때부터 고려해보면 x[0][i]는 miss 되면서 캐시로 로드되지만 다시 같은 블록 자리에 x[1][i]가 miss 되면서 다시 로드되기 때문에 1misses/1references로 miss rate가 100%가 된다.

B. Case 2: What is the miss rate if we double the cache size to 1024 bytes?

C = 1024, E = 1, B = 16이므로 S = 64다.

C = 1024가 되면서 배열 x의 크기인 2\*128\*4=1024와 동일하기 때문에 모든 x 배열 원소에 캐시 블록들이 대응된다. 즉, 같은 블록에 대응되는 배열의 원소들이 존재하지 않는다.

이로써 miss rate는 block size를 사용하여 구할 수 있다.

Block size는 16이며, 총 4 개의 원소가 담긴다. 그러므로 4개의 원소 씩 한 블록에 대응되므로 4개의 원소가 한번에 로드된다. 즉, 4개의 원소당 한 번 꼴로 miss가 발생하며 나머지 3개의 원소에 대해서는 hit이 발생한다.

Miss rate = 1misses/ 4references 이므로 25%이다.

C. Case 3: Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the miss rate?

C = 512, E = 2, B = 16으므로, S = 16이다.

A의 경우와 C 값 즉 Cache size는 동일하나 2-way set associative로 바뀌면서 S 즉, 집합 개수가 16으로 줄어들었다.

E = 2이기 때문에, 즉 line이 2개이므로 같은 set 에 대응되더라도 다른 블록에 대응될 수 있는 것이다. 그러므로 위의 C 코드에서 x[0][i]와 x[1][i]가 같은 set에 대응되더라도 다른 line에 대응되므로 다른 블록에 대응되고 miss rate가 훨씬 감소할 수 있다.

그러나 C = 512이기 때문에, 배열의 크기가 1024임을 감안했을 때 모든 원소가 다른 블록에 대응되지는 않는다. 총 for loop가 128번 반복한다고 할 때, 65번째 열을 반복할 때는 캐시가 다 채워진 상태일 것이며 여기서 LRU replacement policy를 따라 캐시에 로드될 것이다. 즉, LRU strategy에 따라 x[0][i]와 x[1][i]는 각각 x[0][i-64]와 x[1][i-64]가 로드된 블록에 대응되는 것이다.

결국 이것도 마찬가지로 block size가 16이므로 총 4개의 연속된 배열의 원소들이 로드되는 것이 계속하여 반복되게 된다. 그래서 B와 마찬가지로 4개가 연속해서 들어오고 그 중 첫번째 원소에서만 miss가 되므로 1misses/ 4references 에 따라 25% miss rate가 된다.

D. For case 3, will a larger cache size help to reduce the miss rate? Why or why not?

더 큰 cache size는 miss rate를 줄여주지 않는다. 위의 C에서도 계산했던 것처럼 결국 block size에만 영향을 받게 된다.

이는 Cache size가 늘어남에 따라 집합의 갯수가 늘어나는 것은 맞지만 집합의 갯수에 관계없이 블록 단위로 4개의 배열 원소씩 반복하여 가져오며 4개의 원소 중 가장 첫번재 원소에서 miss가 발생하는 구조는 동일하기에 cache size를 늘려서는 더이상 miss rate를 줄일 수 없다.

E. For case 3, will a larger block size help to reduce the miss rate? Why or why not?

더 큰 block size는 miss rate를 줄일 수 있다. 블록 단위로 로드를 해오기 때문에 배열 원소를 4개가 아닌 더 많은 갯수로 한 꺼번에 불러온다면 이는 Miss rate를 줄일 수 있다. 예를 들어 B = 32가 된다면 block 단위로 총 8 개의 원소를 불러오기 때문에 8개의 배열 원소를 불러오고 그 중 첫번째 원소에만 Miss가 발생하고 나머지 7개의 원소는 hit이 일어나므로 결국 miss rate가 1 misses/ 8 references 인 12.5%로 줄어들 수 있다.

**5. Exercise 6.41 on page 694**

텍스트이(가) 표시된 사진

자동 생성된 설명

For loop의 구조를 보게 되면 행을 따라 반복하지 않고 Loop를 stride n의 방식으로 스캔한다. 즉, 내부 Loop를 돌 때마다 miss가 된다고 할 수 있다.

Pixel struct는 char 형 변수 4개로 이루어져있고 8byte 라인이므로 이 r,g,b,a 4개의 변수들은 블록 단위로 가져올 때 다같이 로드될 수 있다. 그러므로 loop를 돌 때마다 가장 처음의 변수 r 에서만 miss가 나며 나머지 g,b,a는 hit이 발생하기에 1 misses / 4 references이므로 25%의 miss rate가 쓰기에서 발생한다.

따라서, hit rate 는 1 – 0.25 로 계산되어 75%이다.

**6. Practice Problem 7.5 on page 731 when the linker relocates .text in m.o to address 0x4004d0 and swap to address 0x4003e8.**

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저,

ADDR(s) = ADDR(.text) = 0x4004d0이다.

ADDR(r.symbol) = ADDR(swap) = 0x4003e8

PC 상대 참조이기 때문에, 참조의 런타임 주소를 계산한다.

refaddr = ADDR(s) + r.offset = 0x4004d0 + 0xa = 0x4004da 이다.

\*refptr = (unsigned)(ADDR(r.symbol) + r.addend – refaddr)

= (unsigned)(0x4003e8 + (- 4) – 0x4004da)

= (unsigned)(-0xF6)

즉, \*refptr값인 -0xF6을 32 byte의 16진수로 표현하면 0x FF FF FF 0a이다.

[Answer]

4004d9: e8 0a ff ff ff callq 4003e8 <swap>

**7. Exercise 7.8 on page 751**

A.

Module1에서는 함수가 정의되었으며, module2에서는 정적 변수가 선언된 것이므로 각자 모듈에서 정의된 것을 선택한다.

(a) REF(main.1) -> DEF(main.1)

(b) REF(main.2) -> DEF(main.2)

B.

Module1과 module2의 x 들은 모두 weak symbol이기 때문에 알 수 없다.

(a) REF(x.1) -> DEF(unknown)

(b) REF(x.2) -> DEF(unknown)

C.

Module1과 Module2의 x들은 모두 strong symbol이며, 이는 복수의 strong symbol이 허용되지 않는 규칙에 어긋나므로 error가 발생한다.

(a) REF(x.1) -> DEF(error)

(b) REF(x.2) -> DEF(error)

**8. Exercise 7.10 on page 752**

A.

p.o가 참조하는 symbol을 libx.a가 정의하기 때문에 libx.a는 맨 뒤에 와야 하며, libx.a 가 나중에 옴으로써 p.o에서 unresolved references를 resolve해준다. libx.a에서 참조하는 symbol을 p.o에서 정의한다고 해도 libx.a는 아카이브이기 때문에 영향을 끼치지 않으므로 p.o가 한 번 써질 필요는 없다.

Answer: gcc p.o libx.a

B.

p.o에서 참조하는 Symbol을 libx.a가 정의하고, libx.a가 참조하는 symbol을 liby.a가 정의하므로 커맨드 라인에는 p.o libx.a liby.a 순서로 써야 된다. 그리고 liby.a가 참조하는 symbol을 libx.a가 정의하므로 liby.a 뒤에 libx.a를 반복하여 써주어야 한다.

Answer: gcc p.o libx.a liby.a libx.a

C.

p.o에서 참조하는 symbol을 libx.a가 정의하기 때문에 p.o를 가장 먼저 쓰고 그 뒤에 libx.a를 쓴다. 여기서 libx.a와 liby.a가 참조하는 Symbol을 libz.a에서 정의하기 때문에 libz.a를 가장 뒤에 써주어야 한다. 그리고 libx.a와 liby.a는 서로 참조하는 Symbol에 대해 정의를 해주기 때문에 libx.a liby.a libx.a의 순서로 써야 한다.

Answer: gcc p.o libx.a liby.a libx.a libz.a

**9. Exercise 7.12 on page 753**

r.offset = 0xa

r.symbol = swap

r.type = R\_x86\_64\_PC32

r.addend = -4

A.

ADDR(s) = ADDR(.text) = 0x4004e0

ADDR(r.symbol) = ADDR(swap) = 0x4004f8

refaddr = ADDR(s) + r.offset = 0x4004e0 + 0xa = 0x4004ea 이다.

\*refptr = (unsigned)(ADDR(r.symbol) + r.addend – refaddr)

= (unsigned)(0x4004f8 + (- 4) – 0x4004ea)

= (unsigned)(0x0a)

[Answer]

4004e9: e8 0a 00 00 00 callq 4004f8 <swap>

B.

ADDR(s) = ADDR(.text) = 0x4004d0

ADDR(r.symbol) = ADDR(swap) = 0x400500

refaddr = ADDR(s) + r.offset = 0x4004d0 + 0xa = 0x4004da 이다.

\*refptr = (unsigned)(ADDR(r.symbol) + r.addend – refaddr)

= (unsigned)(0x400500 + (- 4) – 0x4004da)

= (unsigned)(0x22)

[Answer]

4004d9: e8 22 00 00 00 callq 400500 <swap>