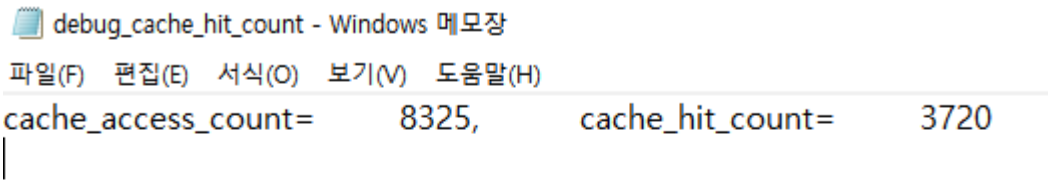
Project 2: Problem 2-1, 2-3

Cache block size 수정 전, cache의 hit ratio 계산

- 먼저, `debug_cache_hit_count_trace.txt`는 다음과 같다.



- 이때, cache의 hit ratio는 다음과 같이 정의된다.

> The fraction of memory access found in a level(upper level) of the memory hierarchy.

- 즉, 다음과 같이 계산할 수 있다.

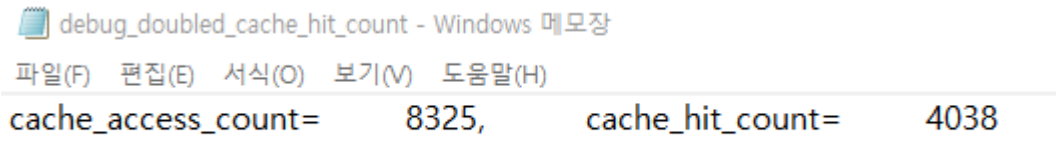
$$(hit\ ratio) = \frac{Number\ of\ cache\ hits}{Number\ of\ cache\ access}$$

- 따라서, cache의 hit ratio는 위의 trace file에서 `cache_hit_count`를 `cache_access_count`로 나눔으로써 계산할 수 있고, 이는 다음과 같다.

$$(hit\ ratio) = \frac{cache\_hit\_count}{cache\_access\_count} = \frac{3720}{8325} \approx 0.446847$$

Cache block size 수정 후, cache의 hit ratio 계산

- 먼저, `debug_doubled_cache_hit_count_trace.txt`는 다음과 같다.



- 따라서, cache hit ratio는 다음과 같이 계산할 수 있다.

$$(hit\ ratio) = \frac{cache\_hit\_count}{cache\_access\_count} = \frac{4038}{8325} \approx 0.485045$$

- 이를 통해, cache block size를 32 byte → 64 byte로 2배 증가시켜준 결과, cache hit ratio가 0.446847 → 0.485045로 증가하였음을 확인할 수 있다.

Cache hit ratio 증가 원인 분석

- 먼저, `loop_test.c`를 살펴보자. 이는 다음과 같다.

```c
#define LOOP 5120

int loop_test()
{
    int result = 0;
    int a[LOOP];

    for (int i = 0 ; i < LOOP; i +=8)
            a[i] = i;


    for (int i = 0 ; i < LOOP; i += 8)
        result = result + a[i];

    return result;
}
```

- 위의 코드에서 주목해야 할 사항은 크게 두 가지 이다.

1. Loop increment stride
    - 반복문이 실행되는 동안 `i` 값은 8만큼 증가한다.
    - 그렇다면, 이 program의 access stride size를 생각해보자.

        - Array의 각 element는 각각 4 byte이며, 8개의 array index를 jump하고 있으므로, 다음 반복에서의 address는 이전 address에서 $4B * 8 = 32B$만큼 더해진 값이다.
        - **이 program의 access increment stride(32B)는 수정 이전의 cache block size(32B)와 정확하게 일치한다.**
    2. 인접한 address에 순차적으로 접근
        - 프로그램에서는 앞서 설명한 stride size만큼 adress를 순차적으로 증가시키며 memory에 접근한다. 즉, stride size만큼 순차적으로 인접한 address에 access하고 있다.
        - 이러한 프로그램의 특성을 고려하는 것은, cache block size의 증가가 hit ratio에 어떤 영향을 끼쳤는지 파악하는 것에 매우 중요하다.

    - 프로그램의 코드를 보면, 인접한 address에 순차적으로 접근하도록 동작하는 것을 알 수 있다.
        - 이러한 프로그램의 특성을 고려하면, **spatial locality가 performance에 critical하게 영향을 끼칠 것**이라고 예상할 수 있다.
    - 또한, spatial locality가 performance에 얼마나 영향을 끼칠지는 **프로그램의 access stride size와 cache block size의 상대적인 크기**에 의해 큰 영향을 받을 것임을 알 수 있다.
        - 즉, access stride size에 비해 cache block size가 상대적으로 클수록, 한 번의 miss로 가져온 cache block에 포함된 address의 범위에 더 많이 access하므로, 더 많은 hit이 발생할 것이다. 즉, spatial locality의 이점이 performance에 더 크게 영향을 끼칠 것이다.
- 위의 두 가지 사항을 고려할 때, cache block size가 32 byte → 64 byte로 두 배 증가함에 따라 cache hit ratio가 증가한 원인은 다음과 같다.
    1. 수정 전. (Cache block size = 32B)
        - 앞서 살펴봤듯이, program의 access stride size와 cache block size가 정확하게 동일하다.
        - 즉, 처음에 compulsory miss가 발생하게 되면, cache block size(32B)만큼 해당 address에 인접한 word를 cache로 가져온다. (Spatial locality)
        - 허나, access stride size가 cache block size와 동일하기 때문에, 다음 access 시에 이전에 가져온 cache block에 포함되어 있지 않은 address에 access한다.

    - 따라서, 또 다시 compulsory miss가 발생하게 되고, spatial locality의 이점이 성능 향상에 큰 영향을 미치지 않는다.
2. 수정 후. (Cache block size = 64B)
    - 허나, 수정 후 cache block size는 64B로, access stride size의 두 배의 값을 가진다.
    - 즉, 처음에 compulsory miss가 발생하게 되면, cache block size(64B)만큼 해당 address에 인접한 data를 cache에 가져온다. (Spatial locality)
    - 이번에는 cache block size가 access stride size의 두 배이기 때문에, 다음 access 시에 이전에 가져온 cache block에 포함된 address에 access하게 된다.
    - 따라서, 이러한 경우에 hit이 발생하고, spatial locality의 이점을 바탕으로 hit ratio가 증가하게 된다.
    - 물론, 다음 access 시에는 또 다시 이전에 가져온 cache block에 포함되어 있지 않은 address에 access하기 때문에, compulsory miss가 발생하게 된다.

- 위와 같은 이유로 cache block size가 32B일 때 보다, 64B일 때 hit ratio가 더 높을 것임을 이론적으로 예측할 수 있다.
    - 이는 물론, `debug_cache_hit_count_trace.txt`과 `debug_doubled_cache_hit_count_trace.txt`를 통해 확인할 수 있는 시뮬레이션 결과와 일치한다. 따라서, 위와 같은 이론적인 예측 결과를 실험적으로도 확인할 수 있었다.
- 마지막으로, Problem 1의 riscv_cache_test.S (input_inst.dat)를 통해 수정된 cache의 eviction size가 64B임을 확인하였다.
    - 이렇게 eviction 동작이 정상적으로 수행되는지 확인함으로써, cache block size가 32B → 64B로 올바르게 수정하였음을 한번 더 검증할 수 있었다.
    - 이는 `debug_cache_eviction_trace.txt` 및 `debug_doubled_cache_eviction_trace.txt`를 통해 확인할 수 있다. (해당 파일들은 Problem 2의 reference 폴더에서 확인하실 수 있습니다.)