Mobile Processor

Cache Memory

Project #4

32171550 박다은

2020 Spring

Contents

1. Project Overview 2

(1) Project Introduction 2

(2) Program Goals 2

(3) Composition in MIPS Simulator 2

(4) Program Structure 4

(5) Build Environment 4

2. Code & Consideration 5

(1) 주요 코드 5

(2) Screenshot 12

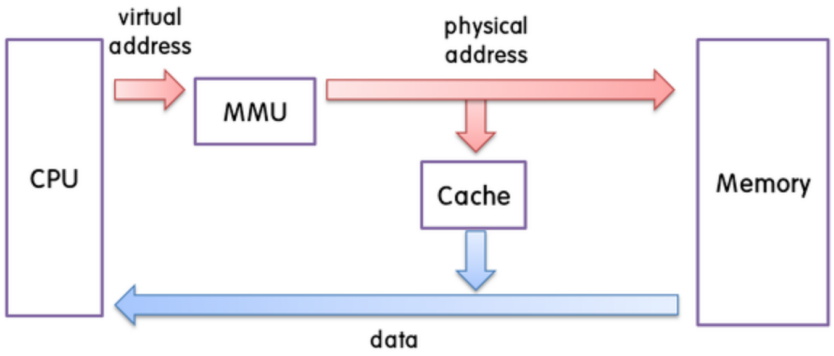
(3) Problems & Solutions 15

(4) Personal Feeling..... 16

1.Project Overview

(1) Project Introduction

기술의 발전으로 프로세서의 처리속도는 빠르게 증가해왔지만, 메모리의 접근 속도는 이를 따라가지 못해 속도의 차이가 발생하게 된다. 이를 개선하기 위해 캐시메모리가 만들어졌다.



캐시는 CPU 칩 안에 들어가는 작고 빠른 메모리이다. 프로세서가 매번 메인 메모리에 접근해 데이터를 받아오면 시간이 오래 걸리기 때문에 캐시에 자주 사용하는 데이터를 담아두고, 해당 데이터가 필요할 때 프로세서가 메인 메모리 대신 캐시에 접근하도록 해 처리 속도를 높인다. 즉, 캐시메모리를 사용하면 주 기억장치를 접근하는 횟수가 줄어들어 컴퓨터의 처리속도가 향상된다.

(2) Project Goals

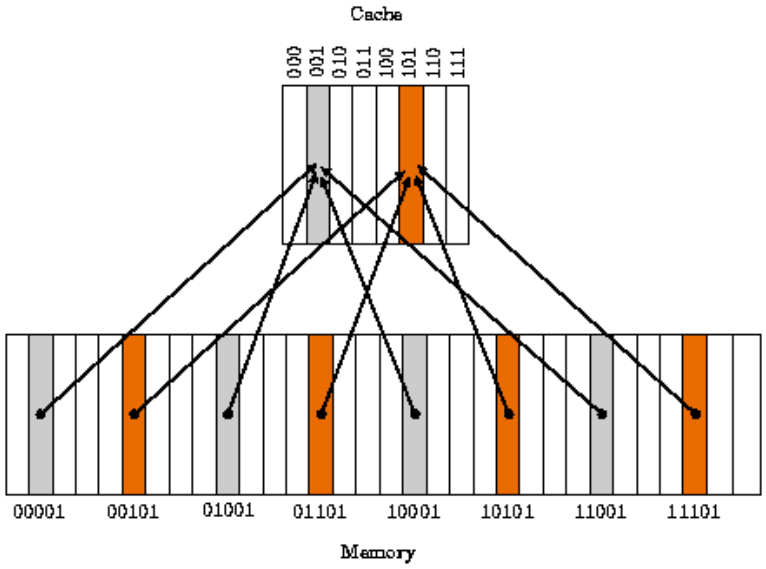
컴퓨터 시스템의 성능을 향상시키기 위한 메모리이자 레지스터와 함께 메모리 계층 구조의 전통적인 핵심 계층 중 하나인 캐시메모리를 직접 구현해보며 구조를 이해하고 캐시 성능을 분석한다. 또한 캐시의 성능을 최적화할 수 있는 방법을 고안하며 시도한다.

(3) Concepts in MIPS simulator

- Cache Hit / Cache Miss
  - CPU 가 데이터를 요청했을 때 캐시메모리가 해당 데이터를 가지고 있다면 이를 'Cache Hit'라 부르고, 해당 데이터가 없어 메모리에서 데이터를 가져와야 한다면 'Cache Miss'라 부른다.
  - Cold miss: 해당 메모리 주소를 처음 불렀기 때문에 발생하는 미스
  - Conflict miss: 두 개의 데이터가 같은 캐시 메모리 주소에 할당되어 발생하는 미스

- ## Associative Cache

  - Direct-mapped Cache:



    주기억장치의 블록들이 지정된 한 개의 캐시라인으로만 사상 될 수 있는 매핑 방법이다. 간단하고 빠르지만, conflict miss 문제가 굉장히 큰 방식이다.

  - Fully Associative Cache:

    비어 있는 캐시메모리가 있으면 아무 장소에 매핑 될 수 있다. 저장할 때는 간단하지만, 찾을 때는 모든 캐시 내 블록을 검사해야 한다. 충돌은 적지만 속도가 느리다.

  - Set Associative Cache:

    Direct-mapped cache 와 fully associative cache 의 장점만을 취한 방식이다. 특정 행을 지정해서 그 행 안의 어떤 열이든 비어 있으면 저장하는 방식이다. Direct 에 비해서 검색은 오래 걸리지만 저장이 빠르며, fully associative 에 비해 저장이 느린 대신 검색이 빠른 중간 형이라 할 수 있다.

- ## 쓰기 정책

  - Write-through: 캐시에 쓰기 동작이 이루어질 때마다 캐시메모리와 주기억장치의 내용을 동시에 갱신하므로 쓰기 동작에 걸리는 시간이 길다.
  - Write-back: 캐시에 쓰기 동작이 이루어지는 동안은 캐시의 내용만이 갱신되고, 캐시의 내용이 캐시로부터 제거될 때 주기억장치에 복사된다.

(4) Program Structure

Open_file

+----------------------------

| fetch()

| ReadMem()

| decode()

| execute()

| memory()

| ReadMem() or WriteMem()

| write_back()

| Update Pipeline Latch

+----------------------------

각 단계에서 각자의 역할은 결과를 출력하는 것이다.

Pipeline 에 캐시메모리를 추가하는 방법으로 구현하였기 때문에, 전체적으로 pipeline 의 structure 와 유사하다. 사이클 당 다섯단계를 실행하는 건 pipeline 과 같지만, 메모리에 접근하는 fetch 와 memory 단계 이후 ReadMem 혹은 WriteMem 이 이루어진다.

(5) Build Environment

Compilation: Linux Assam, with GCC

To compile, please type:

gcc ./direct.c -o pipeline_32171550 or

gcc ./fully.c -o pipeline_32171550

To run, please type:

./cache_32171550 input.bin (input: simple, simple2, fib, gcd …)

2.Code & Consideration

캐시메모리를 구현하는 방식으로 'Direct-mapped Cache'와 'Fully Associative Cache' 두가지를 구현하였다. 캐시메모리의 사이즈를 두 방식 다 Cache line size 는 64bytes, Cache size(data store size)는 128bytes 로 구성하였다. 전체 코드가 pipeline 과 유사하므로 저번 프로젝트에서 작성한 부분은 제외한 채 캐시메모리를 구현하기 위해 추가한 부분만 보고서에 작성하도록 하겠다.

(1) 주요 코드

① Global

```
15 //cache
16 struct cacheline{
17         unsigned int tag : 19;
18         unsigned int sca : 1;
19         unsigned int valid : 1;
20         unsigned int dirty : 1;
21         int data[16];
22 };
23 struct cacheline Cache[128];
24
25 int hit = 0;
26 int miss = 0;
27
28 int check_hit(int c_addr);
29 int ReadMem(int addr);
30 void WriteMem(int addr, int value);
31
32
```

<Direct-mapped Cache>

```
15 //cache
16 struct cacheline{
17         unsigned int tag : 26;
18         unsigned int sca : 1;
19         unsigned int valid : 1;
20         unsigned int dirty : 1;
21         int data[16];
22 };
23 struct cacheline Cache[128];
24
25 int hit = 0;
26 int miss = 0;
27
28 int cache_index = 0;
29
30 int do_miss(int c_addr);
31 int ReadMem(int addr);
32 void WriteMem(int addr, int value);
```

<Fully Associative Cache>

A. struct cacheline

tag	캐시메모리에서 블록을 구분해주는 값이다.
sca	캐시메모리의 블록이 다 차서 replacement 를 해야 할 때, 자주 쓰이는 tag 에 chance 를 주는 replacement algorithm 을 위한 1bit 로, chance 가 주어졌다면 1, 그렇지 않다면 0 을 갖는다.
valid	캐시라인의 접근을 허용하는 구분자로 허용한다면 1, 아니면 0 을 갖는다.
dirty	Write-back 방식을 사용할 때 필요한 구분자로, 데이터가 변경됐다면 1, 아니면 0 을 갖는다.
data	태그에 맞는 데이터를 저장하는 데이터필드이다.

B. Cache 변수

hit	Hit 된 개수를 세는 변수이다.
miss	Miss 된 개수를 세는 변수이다.
cache_index	Fully associative cache 에서만 쓰인다. 블록에 쓰이는 위치를 하나씩 증가시키기 위한 변수이다. replacement 할 때 oldest 를 찾기 위해서도 쓰인다.

C. Functions

ReadMem	캐시에 저장된 데이터를 읽는 함수이다.
WriteMem	SW 명령어가 행해졌을 때, 캐시의 데이터필드에 값을 쓰는 함수이다.
check_hit	Direct-mapped cache 에서만 쓰인다. Address 가 hit 인지 miss 인지 검사하고 각각에 맞는 동작을 하는 함수이다.
do_miss	Fully associative cache 에서만 쓰인다. Cache miss 가 일어났을 때 실행되는 함수이다.

② Direct-mapped Cache

A. check_hit 함수

```
159 int check_hit(int c_addr){
160         //divide address
161         int c_tag = c_addr >> 13;
162         int c_index = (c_addr >> 6) & 0x7f;
163         int c_offset = c_addr & 0x3f;
```

- Direct-mapped cache 는 address 를 tag, index, offset 의 세 부분으로 나눈다. Cache line size 가 64bytes 이므로, offset 은 6 이 되고, cache size 가 128bytes 이므로 index 가 7 이된다. 총 address 는 32bit 이므로, tag 는 19bit 가 된다. 즉 tag(19bit) | index(7bit) | offset(6bit)로 이루어진다. 따라서 처음 함수에 address 가 들어오면, 이를 각각의 자리와 비트에 맞게 쪼개 주었다.

```
165         //Cache Hit
166         if((Cache[c_index].valid == 1) && (Cache[c_index].tag == c_tag) && Cache[c
    _index].dirty != 1){
167                 hit++;
168                 //printf("\tHIT!(%x,%x,%x)\n",c_tag,c_index,c_offset);
169                 return 1;
170         }
```

- Valid 가 1 이고, dirty 가 0 이고, tag 가 같을 때 hit 가 되도록 설정했다. Hit 면 변수 hit 의 값을 1 만큼 증가시키고, 리턴 값을 1 로 반환한다. 즉 hit 면 함수의 반환값은 1 이고, miss 면 0 이되도록 설정한 것이다.

```
172         //Cache Miss
173         else{
174                 miss++;
175                 //printf("\tMISS!(%x,%x,%x)\n",c_tag,c_index,c_offset);
176
177                 //write-back
178                 if(Cache[c_index].tag != c_tag){
179                         Cache[c_index].dirty = 1;
180
181                         for(int i=0; i<16; i++){
182                                 int store_addr = (Cache[c_index].tag << 13) | (c_i
    ndex << 6) | (i*4);
183                                 if(Cache[c_index].data[i] != 0){
184                                         Memory[store_addr/4] = Cache[c_index].data
    [i];
185                                 }
186                                 Cache[c_index].data[i] = 0;
187                         }
188                 }
189
190                 Cache[c_index].valid = 1;
191                 Cache[c_index].tag = c_tag;
192                 return 0;
193         }
194 }
```

- if 문을 충족시키지 못한다는 것은 cache miss 라는 뜻이므로, miss 의 값을 1 만큼 증가시킨다.
- 쓰기 정책은 write-back 을 사용했다. Direct-mapped cache 는 index 당 하나의 tag 만 가지므로, 두개의 다른 tag 가 같은 index 를 가지면 충돌이 발생한다. 따라서 index 의 tag 가 달라지면 dirty 를 1 로 바꾸고, 원래 tag 에 맞게 들어있는 data 값을 메모리에 작성하도록 했다. 현재 캐시에 저장 돼있는 값들은 쪼개진 상태이므로, 이를 다시 메모리에 작성할 때는 address 를 다시 합쳐야 한다. 따라서 store_addr 라는 변수를 만들어 원래 address 로 되돌린 후에 그 address 값의 메모리에 데이터를 저장하도록 했다. Data 의 값이 0 이라면 비어있다는 뜻이므로, 데이터값이 0 일 때를 제외하고 메모리에 값을 쓰도록 했다. 마지막으로 메모리에 모두 저장했다면, 그 데이터필드는 다시 0 으로 초기화시켰다.
- 또한 캐시에 접근했으므로 그 라인의 valid 를 1 로 바꿔주고, tag 를 현재 address 의 tag 로 바꿔주었다.

B. ReadMem

```
194 int ReadMem(int addr){
195         int current_tag = addr >> 13;
196         int index = (addr >> 6) & 0x7f;
197         int offset = addr & 0x3f;
198         int dirty_index;
199
200         if(index != dirty_index) Cache[index].dirty = 0;
201
202         if(check_hit(addr) == 1){
203                 if(Cache[index].data[offset/4] == 0)
204                         Cache[index].data[offset/4] = Memory[addr/4];
```

```
205         }
206         else{
207                 if(Cache[index].tag != current_tag)
208                         dirty_index = index;
209
210                 Cache[index].data[offset/4] = Memory[addr/4];
211         }
212         return Cache[index].data[offset/4];
213 }
```

- check_hit 함수와 같은 방법으로 address 를 나눠주었다.
- check_hit 가 1 이면(cache hit 라면) 다른 특별한 과정 없이 원래 캐시에 갖고 있던 데이터를 반환하도록 했다. 캐시에 offset 에 맞는 data 가 들어있지 않을 경우, 메모리에 있는 값을 가져오도록 했다.
- check_hit 함수에서 miss 일 때 dirty 를 1 로 바꿔주었기 때문에, 이를 index 가 바뀌었을 때 다시 0 으로 바꾸기 위해 dirty_index 라는 변수를 만들었다. 따라서 check_hit 이 0 이고(cache miss) 현재 tag 와 index 의 tag 가 다르다면, dirty_index 를 현재 index 로 바꾸어주었다. Miss 이면 data 에는 아무 것도 들어있지 않으므로, 메모리에 있는 값을 캐시로 가져온다.
- 마지막으로, address 의 offset 에 맞는 data 의 값을 반환한다.

C. WriteMem

```
217 void WriteMem(int addr, int value){
218         int current_tag = addr >> 13;
219         int index = (addr >> 6) & 0x7f;
220         int offset = addr & 0x3f;
221
222         if(check_hit(addr) == 1){
223                 Cache[index].data[offset/4] = value;
224                 return;
225         }
226         else{
227                 Cache[index].data[offset/4] = value;
228         }
229
230         //write-through
231         //Memory[addr/4] = value;
232 }
```

- Value 를 address 의 offset 에 맞는 데이터필드에 값을 쓴다.
- Write-back 방법을 사용했으므로, write-through 는 주석처리 시켰다. Write-through 는 캐시의 데이터필드에 값을 쓰는 동시에 메모리에도 값을 쓰므로, 구현방법은 훨씬 간단하다.

③ Fully Associative Cache

A. do_miss

Fully Associative Cache 는 hit 인지 검사하기 위해, 모든 블록을 확인해야 하므로, check_hit 함수를 만드는 대신 do_miss 함수를 만들었다. Hit 인지 check 하는 과정은 ReadMem 과 WriteMem 에서 각각 행해지게 하였고, miss 일때만 이 함수를 호출하도록 했다.

```
161 int do_miss(int c_addr){
162         int c_tag = c_addr >> 6;
163         int c_offset = c_addr & 0x3f;
```

- Direct-mapped cache 와 달리 fully associative cache 는 tag 와 offset 두 가지로 address 를 나눈다. Direct 와 마찬가지로 cache line size 가 64bytes 이므로, offset 은 6bit 이다. 또한, address 의 bit 는 32bit 이므로, tag 는 26bit 이다.

```
165         //capacity miss
166         if(cache_index > 127)
167                 cache_index = 0;
168
169         if(Cache[cache_index].tag != c_tag){
170                 while(Cache[cache_index].sca == 1 && cache_index<127){
171                         Cache[cache_index].sca = 0;
172                         cache_index++;
173                 }
174                 if(cache_index > 127)
175                         cache_index = 0;
176
177                 Cache[cache_index].valid = 0;
178                 Cache[cache_index].dirty = 1;
179
180                 //write-back
181                 for(int i=0; i<16; i++){
182                         int store_addr = (Cache[cache_index].tag << 6) | (i*4);
183                         if(Cache[cache_index].data[i] != 0){
184                                 Memory[store_addr/4] = Cache[cache_index].data[i];
185                         }
186                         Cache[cache_index].data[i] = 0;
187                 }
188         }
189
190         Cache[cache_index].valid = 1;
191         Cache[cache_index].tag = c_tag;
192         return 0;
193 }
```

- Direct-mapped cache 는 address 에서 index 를 가졌으므로, replacement 를 고려할 필요가 없지만, fully associative cache 에서는 index 구분 없이 캐시에 값을 넣으므로 용량이 다 차게 될 수 있다. 따라서 cache_index(캐시라인의 index)가 127 이 넘으면 모든 라인이 꽉 차 있다는 뜻이므로, cache_index 를 0 으로 바꿔준다. Cache_index 는 0 부터 1 씩 늘어났으므로, oldest 가 0 이기 때문이다.

- 캐시의 용량이 다 찼고 현재 tag 와 캐시라인의 tag 가 다르다면, replacement 를 해줘야 한다. while 문을 통해, second chance 가 주어지지 않은 tag 를 찾아주었다. Second chance 가 주어진 tag 면 기회를 쓴 것이기 때문에 sca 를 다시 0 으로 바꿔주었다.
- 또한 tag 를 replacement 한다면, data 의 값도 바뀌어야 하기 때문에 dirty 를 1 로 바꾸어 주었다. Direct-mapped cache 와 같은 방법으로, store_addr 변수를 지정했고 store_addr 값의 메모리에 현재 데이터의 값을 넣었다. 마찬가지로 데이터필드를 0 으로 초기화시켰다.
- 캐시라인에 접근하였으므로 valid 를 1 로 바꾸고, tag 를 현재 tag 값으로 바꾸어 주었다.

B. ReadMem

```
195 int ReadMem(int addr){
196         //divide address
197         int current_tag = addr >> 6;
198         int offset = addr & 0x3f;
199 
200         //dirty
201         int dirty_index = 0;
202         if(cache_index != dirty_index) Cache[cache_index].dirty = 0;
203 
204         //Cache Hit
205         for(int j=0; j<128; j++){
206                 if((Cache[j].valid == 1) && (Cache[j].tag == current_tag) && Cache
    [j].dirty != 1){
207                         hit++;
208                         //printf("\tHIT!(%x,%x,%x)\n",j,current_tag,offset);
209 
210                         if(Cache[j].data[offset/4] == 0)
211                                 Cache[j].data[offset/4] = Memory[addr/4];
212                         Cache[j].sca = 1;
213                         return Cache[j].data[offset/4];
214                 }
215         }
```

- Direct-mapped cache 와 같은 이유로 dirty_index 변수를 만들고, 같은 방법으로 사용했다.
- Fully associative cache 에서는 cache hit 인지 확인하기 위해서 캐시의 모든 라인을 검사해야 한다. 따라서 for 문을 이용해 valid 가 1 이고, dirty 가 1 이 아니고, tag 가 현재 tag 와 같은 라인이 있는지 검사하도록 했다. 존재한다면, 변수 hit 의 값을 1 만큼 증가시켰다. 또한 sca 의 값도 1 로 바꿔주었다.
- Hit 라면, 더 이상 검사할 필요가 없으므로 데이터값을 반환하며 함수를 종료시켰다.

```
217         //Cache Miss
218         cache_index++;
219
220         do_miss(addr);
221
222         miss++;
223         //printf("\tMISS! (%x,%x,%x)\n", cache_index, current_tag, offset);
224
225         if(Cache[cache_index].tag != current_tag)
226                 dirty_index = cache_index;
227
228         Cache[cache_index].data[offset/4] = Memory[addr/4];
229
230         return Cache[cache_index].data[offset/4];
231 }
```

- for 문을 통해 검사하며, hit 가 나오지 않았을 경우에는 cache miss 가 된다. 따라서, 새로운 라인에 tag 와 데이터를 작성할 수 있도록 cache_index 를 1 만큼 증가시키고 do_miss 함수를 호출해주었다.
- ReadMem 함수의 나머지 부분은 direct-mapped cache 와 동일하다.

C. WriteMem

```
233 void WriteMem(int addr, int value){
234         //divide address
235         int current_tag = addr >> 6;
236         int offset = addr & 0x3f;
237
238         //Cache Hit
239         for(int j=0; j<128; j++){
240                 if((Cache[j].valid == 1) && (Cache[j].tag == current_tag) && Cache
    [j].dirty != 1){
241                         hit++;
242                         //printf("\tHIT!(%x,%x,%x)\n",j,current_tag,offset);
243
244                         Cache[j].data[offset/4] = value;
245                         Cache[j].sca = 1;
246                         return;
247                 }
248         }
```

- ReadMem 과 거의 동일하지만, 캐시의 데이터에 값을 작성할 때 메모리에서 값을 가져오는 것이 아닌 함수의 인자인 value 값을 데이터에 작성하도록 했다.
- hit 이면 함수를 종료시켰다.

```
250         //Cache Miss
251         cache_index++;
252
253         do_miss(addr);
254
255         miss++;
256         //printf("\tMISS! (%x,%x,%x)\n", cache_index, current_tag, offset);
257
258         Cache[cache_index].data[offset/4] = value;
259
260         //write-through
261         //Memory[addr/4] = value;
262 }
```

- miss 일 때도 ReadMem 과 비슷하다. 마찬가지로 데이터에 value 를 작성하도록 했다.
- Write-through 방법은 사용하지 않으므로 주석처리 해주었다.

(2) Screenshots

① 출력화면

```
final return value(R[2])             : 55
# of cycles                          : 3063
# of instuctions                     : 3058
# of memory operation instructions   : 1095
# of register operation instructions : 1600
# of branch instructions             : 89
# of taken branches                  : 34
# of jump instructions               : 274

*********************************************
# of hits                            : 4142
# of misses                          : 11

Cache Hit Rate                       : 0.997351
Cache Miss Rate                      : 0.002649
Average Memory Access Time(AMAT)     : 4131.058271
*********************************************
```

출력은 전체 사이클이 종료된 후, 결과값(R[2])과, cycle 수, 각 명령어들을 count 한 값을 먼저 출력하고, hit 의 개수 miss 의 개수, 마지막으로 hit rate, miss rate 와 AMAT 를 출력하도록 했다.

② ‘Direct Mapped Cache’ vs ‘Fully Associative Cache’

- 대표로 simple.bin, gcd.bin, input4.bin 파일들에 대해서 single-cycle 과 invalidation, branch prediction 의 cycle 수를 비교해보겠다.

- simple.bin

```
final return value(R[2])              : 0
# of cycles                           : 14
# of instuctions                      : 9
# of memory operation instructions    : 2
# of register operation instructions  : 6
# of branch instructions              : 0
# of taken branches                   : 0
# of jump instructions                : 1

***********************************************
# of hits                             : 9
# of misses                           : 2

Cache Hit Rate                        : 0.818182
Cache Miss Rate                       : 0.181818
Average Memory Access Time(AMAT)      : 7.727273
***********************************************
```

<Direct-mapped Cache>

```
final return value(R[2])              : 0
# of cycles                           : 14
# of instuctions                      : 9
# of memory operation instructions    : 2
# of register operation instructions  : 6
# of branch instructions              : 0
# of taken branches                   : 0
# of jump instructions                : 1

***********************************************
# of hits                             : 9
# of misses                           : 2

Cache Hit Rate                        : 0.818182
Cache Miss Rate                       : 0.181818
Average Memory Access Time(AMAT)      : 7.727273
***********************************************
```

<Fully Associative Cache>

- direct-mapped cache 와 fully associative cache 에서 hit 수와 miss 수의 차이가 없다.
- Hit 는 9 이고, miss 는 2 로 miss rate 가 0.2 정도라는 것을 확인할 수 있다.

gcd.bin

```
final return value(R[2])              : 1
# of cycles                           : 1206
# of instuctions                      : 1201
# of memory operation instructions    : 486
# of register operation instructions  : 545
# of branch instructions              : 67
# of taken branches                   : 39
# of jump instructions                : 103

***********************************************
# of hits                             : 1607
# of misses                           : 80

Cache Hit Rate                        : 0.952579
Cache Miss Rate                       : 0.047421
Average Memory Access Time(AMAT)      : 1534.587433
***********************************************
```

<Direct-mapped Cache>

```
final return value(R[2])              : 1
# of cycles                           : 1206
# of instuctions                      : 1201
# of memory operation instructions    : 486
# of register operation instructions  : 545
# of branch instructions              : 67
# of taken branches                   : 39
# of jump instructions                : 103

***********************************************
# of hits                             : 1664
# of misses                           : 23

Cache Hit Rate                        : 0.986366
Cache Miss Rate                       : 0.013634
Average Memory Access Time(AMAT)      : 1641.627149
***********************************************
```

<Fully Associative Cache>

- direct-mapped cache 의 miss 수가 80 으로 fully associative cache 의 miss 수인 23 과 비교했을 때 더 높다. Miss rate 도 역시 direct-mapped cache 가 더 높다.
- 사이클 수가 증가함에 따라, miss rate 가 0.04 정도로 확 줄어든 것을 확인할 수 있다.
- AMAT 는 direct-mapped cache 에서는 1534, fully associative cache 에서는 1641 으로 direct-mapped cache 에서 더 적은 시간이 드는 것을 확인할 수 있다.

input4.bin

```
final return value(R[2])              : 85
# of cycles                           : 23376940
# of instuctions                      : 23376935
# of memory operation instructions    : 7116607
# of register operation instructions  : 14231347
# of branch instructions              : 2028877
# of taken branches                   : 2028007
# of jump instructions                : 104

***********************************************
# of hits                             : 30254781
# of misses                           : 238762

Cache Hit Rate                        : 0.992170
Cache Miss Rate                       : 0.007830
Average Memory Access Time(AMAT)      : 30019757.974684
***********************************************
```

<Direct-mapped Cache>

```
final return value(R[2])              : 85
# of cycles                           : 23376940
# of instuctions                      : 23376935
# of memory operation instructions    : 7116607
# of register operation instructions  : 14231347
# of branch instructions              : 2028877
# of taken branches                   : 2028007
# of jump instructions                : 104

***********************************************
# of hits                             : 30426311
# of misses                           : 67232

Cache Hit Rate                        : 0.997795
Cache Miss Rate                       : 0.002205
Average Memory Access Time(AMAT)      : 30359375.465506
***********************************************
```

<Fully Associative Cache>

- cycle 수가 폭발적으로 늘어남으로써 두 코드의 출력도 확실한 차이를 보인다. Direct-mapped cache 의 miss 수는 238762 이고, fully associative cache 의 miss 수는 67232 으로, direct-mapped cache 가 훨씬 많은 miss 수를 보인다는 것을 확인할 수 있다.
- miss rate 역시 0.007 로 확연히 줄어든 것을 확인할 수 있다.
- Direct-mapped cache 의 AMAT 는 30019757 이고, fully associative cache 의 AMAT 는 30359375 로 역시 direct-mapped cache 가 더 빠른 것을 확인할 수 있다.

③ Second Chance Algorithm

```
final return value(R[2])              : 85
# of cycles                           : 23376940
# of instuctions                      : 23376935
# of memory operation instructions    : 7116607
# of register operation instructions  : 14231347
# of branch instructions              : 2028877
# of taken branches                   : 2028007
# of jump instructions                : 104

***********************************************
# of hits                             : 30424912
# of misses                           : 68631

Cache Hit Rate                        : 0.997749
Cache Miss Rate                       : 0.002251
Average Memory Access Time(AMAT)      : 30356589.931905
***********************************************
```

<not sca>

```
final return value(R[2])              : 85
# of cycles                           : 23376940
# of instuctions                      : 23376935
# of memory operation instructions    : 7116607
# of register operation instructions  : 14231347
# of branch instructions              : 2028877
# of taken branches                   : 2028007
# of jump instructions                : 104

***********************************************
# of hits                             : 30426311
# of misses                           : 67232

Cache Hit Rate                        : 0.997795
Cache Miss Rate                       : 0.002205
Average Memory Access Time(AMAT)      : 30359375.465506
***********************************************
```

<sca>

- Fully associative algorithm 에서 second chance algorithm 정책을 사용하지 않았을 때와 사용했을 때를 비교해보았다.
- 사용하지 않았을 때가 miss 의 수가 더 많은 것을 확인할 수 있다.

(3) Problems & Solutions

① write-back

- 처음에는 쓰기정책을 write-through 방법을 사용했고, 이를 구현에 성공하고 나서 write-back 방법으로 바꾸는 과정을 거쳤다. 이 과정속에서 input 파일 절반은 제대로 돌아갔지만, 나머지 절반에서는 오류가 났다. Write-through 에서 성공한 출력값과 비교하며 오류를 찾아갔다. 처음 코드에서는 tag의 data가 0인지 검사하지 않고 data에 있는 모든 값들을 for 문을 사용해 메모리에 저장하고 그 이후 0 으로 초기화 시켰었다. 따라서 이미 메모리에 제대로 저장된 값이 있는데, 0 으로 또 저장이 돼 오류가 났던 것이었다. 따라서 data 가 0 인지 검사하고 0 이 아닌 값들만 메모리에 저장시키도록 했더니 해결되었다.

② Pipeline 의지

- 이번 캐시메모리 프로젝트는 이미 작성한 pipeline 에 캐시메모리를 추가하는 것이 목표였다. 따라서 처음에 pipeline 에서 메모리에 접근하는 fetch 와 memory 부분만 수정하였다. fetch 에서 inst = Mem[PC/4]를 inst = ReadMem(PC)로 수정하였고 memory 에서 LW 에서는 Address = Mem[ALU_result]를 Address = ReadMem(ALU_result)로 수정하였다. 또, SW 에서는 Memory[ALU_result] = v2 를 WriteMem(ALU_result, v2)로 수정하였다.

- 위에서 write-through 에서 write-back 으로 쓰기정책을 바꾸었다고 하였는데, 위 방법으로 고치고 난 후에도 input4.bin 파일이 돌아가지가 않았다. 정확히 25490 번째 cycle 에서 오류가 났어서 올바른 출력값과 비교하며 25000 개의 사이클을 다 확인해 볼 수가 없었다. 따라서 코드만 계속 들여다보았는데, 반나절동안 고민해봐도 어디서 틀린 건지 찾을 수가 없었다. 그러던 와중, 저번 프로젝트 때 만들었던 sum.bin 파일을 넣어봤는데, 이 파일에서도 오류가 났다. 이 파일은 몇백개의 사이클만 가져서 출력값을 비교해볼 수 있었다.

- 문제는 write-back 에 있지 않았다. Write-through 에서는 input4.bin 파일도 돌아갔으므로 당연히 write-back 의 오류라 생각 해 그 부분만 계속 수정했던 것이 거의 하루동안 오류를 찾지 못했던 원인이 되었다.

- 문제는 data dependency 에 있었다. 앞서 말했듯이, fetch 와 memory 에서만 ReadMem과 WriteMem함수를 호출했는데, data dependency에서도 memory에 접근하는 과정이 있었기 때문에 오류가 났던 것이었다. Memory 에도 data 를 쓴 write-through 방법에서는 오류가 나지 않았지만, 데이터값이 캐시에만 써있는 write-back 에서는 오류가 난 것이었다. 따라서 파일내에서 'Memory'을 검색하여 메모리에 접근하는 모든 경우들에서 함수를 호출했더니 오류가 해결되었다.

(4) Personal Feeling

세 번의 프로젝트를 끝을 냈고, 이번이 드디어 마지막 프로젝트였다. 이번 프로젝트는 기말고사와 각종 과제들과 완전히 겹쳐 많은 시간을 쏟을 수가 없었다. 따라서 확실히 개념을 습득한 후에 코드를 작성해야겠다고 생각했고 그렇게 했다. 세번의 프로젝트들로 개념을 정확히 모른 채 코드를 작성하면 훨씬 많은 시간이 걸리고, 오류도 훨씬 많이 생긴다는 것을 깨달았기 때문이다. 정말 작은 실수도 용납할 수 없었으므로 더욱 꼼꼼히 확인하고 작성했던 것 같다.

캐시메모리는 다른 프로젝트들보다는 쉽다는 얘기를 듣기도 했고, 기간도 일주일 정도밖에 되지 않으니 수월하게 할 수 있을 거라 생각했다. 처음엔 direct-mapped cache 와 fully associative cache 그리고 set associative cache 까지 전부 구현해내는 것이 목표였다. 이 프로젝트를 끝으로 한 학기가 끝나므로, 마지막은 더 완벽하게 마무리하고 싶었기 때문이다. 그러나 전혀 수월하지 않았다. 위 Problems & Solutions 에서 작성한 두번째 오류가 정말 많은 시간을 잡아먹었다. 하루종일 그림도 그려보고 구글에 찾아도 보며 계속 고민한 문제였는데, 생각보다 별 게 아닌 오류였어서 많이 허무했다. 하지만 위에 따로 작성한 오류들 말고는 크게 기억나는 오류가 없을 정도로 많은 오류가 발생하지는 않았다. 처음 프로젝트에서는 사소한 오류가 매우 많았는데 점점 프로젝트들을 진행하면서, 작성할 때 꼼꼼하게 확인하는 과정을 거쳐 작은 오류들은 거의 나타나지 않을 정도로 발전하게 되었다.

한 학기동안 네개의 프로젝트를 진행하면서 바쁘게 달려왔고, 체력적으로도 많이 힘들었지만 결과적으로는 얻은 게 훨씬 많은 것 같다. 몇백줄이나 되는 코드를 처음부터 끝까지 온전히 내 힘만으로 작성해본 것은 처음이었다. 그 과정에 전에는 느껴보지 못했던 엄청난 뿌듯함과 성취감을 느꼈고, 자신감도 많이 높아지게 되었다. 처음 프로젝트를 진행할 때만 해도 이러한 긴 코드를 작성하는 것은 상상도 하지 못할 일이었다. 예전에는 진로에 대해 별 생각없이 살아왔는데, 3 학년이 되었고 또 이런 프로젝트들도 하면서 처음으로 진로를 오랫동안 고민해보기도 했다. 이번학기는 고 3 때보다 더 열심히 살았던 유일한 해였던 것 같다. 이 수업이 어쩌면 인생에 큰 커닝포인트가 되는 시점이 될 지도 모르겠다.

네번째 프로젝트를 마지막으로, 한 학기를 마무리하며 복합적인 느낌이다. 가장 큰 감정은 무엇보다 성취감과 후련함이다. 여러가지로 아쉬움도 있지만, 한학기라는 시간동안 얻어간 게 큰 것 같아 많이 뿌듯하다. 여태까지는 방학을 의미없이 보냈지만, 이번 방학에는 이 기분을 이어가 헛되이 보내지 않는 것이 목표이다. 스스로도 한 학기동안 정말 수고 많았다고 칭찬해주고 싶다.

한 학기동안 너무 감사했습니다 교수님. 앞으로도 열심히 하겠습니다!