Mobile Processor

Project #4

- Cache pipeline MIPS emulator

모바일시스템공학과
32182490
안소민
Dankook University
2020 7 3

목차

1. Project Introduction

Project3에서 했던 pipeline 구조를 기반으로 cache를 구현한 프로그램을 만든다. 구현한 프로그램으로 7개의 input file을 연산하여 결과 값을 출력한다. Cpu에 비해서 속도가 느린 memory의 단점을 보완해줄 수 있는 cache가 존재하는데, 상대적으로 용량이 작은 특징을 이용해서 자주 접근되는 필요한 데이터들을 미리 cache에 옮겨 놓음으로써 메모리만을 사용했을 때보다 프로그램의 효율을 높인다. 3가지 cache 구현 방법 중에서 Fully Associate Cache 방법을 사용했다.

2. Project Goal

- cache의 기본 구조에 대해서 이해한다

- 캐시 구조 중에서 Fully Associative cache, Direct-mapped cache, Set-Associative cache를 이해한다.

- Write policy에 대해 이해한다.

- Replacement Policy를 이해하고 구현한다.

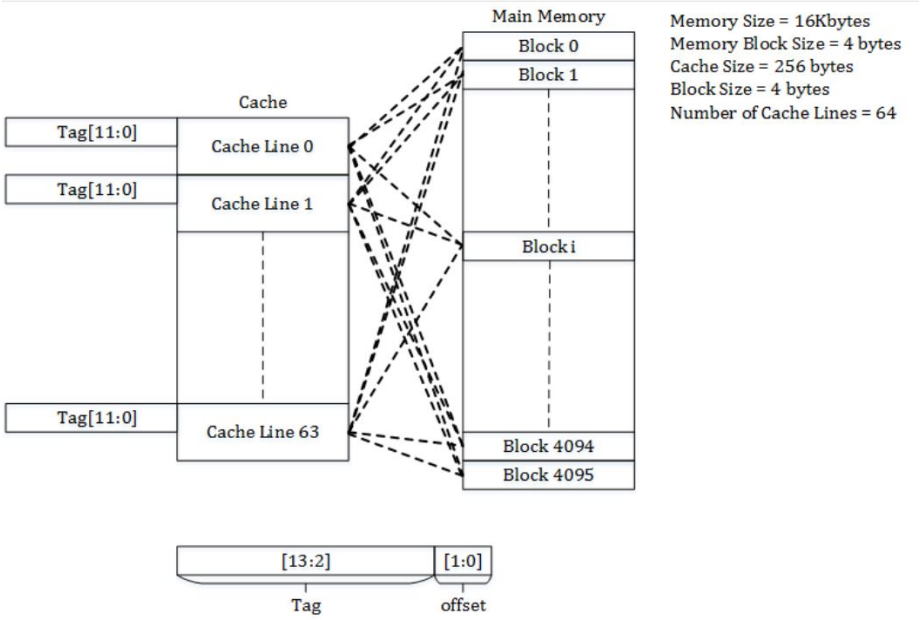
3. Concepts used in Cache Pipeline MIPS emulator

■ Cache의 기본 구조

- ✓ Tag bit: offset bit과 index bit를 제외한 나머지 bit
- ✓ Offset bit: 메모리상에서의 정확한 위치를 나타내는 bit
- ✓ Valid bit: cache상에서 존재여부를 알 수 있게 하는 유효 bit(0 or 1)
- ✓ Dirty bit: 바뀐 사항이 있을 시 표시하는 bit로, cache에 write를 할 때 1을 준다.
- ✓ Index bit: Direct mapped/Set Associative cache 방법에서 쓰이는 bit로, 각 cache line을 가리키는 인덱스 bit이다.

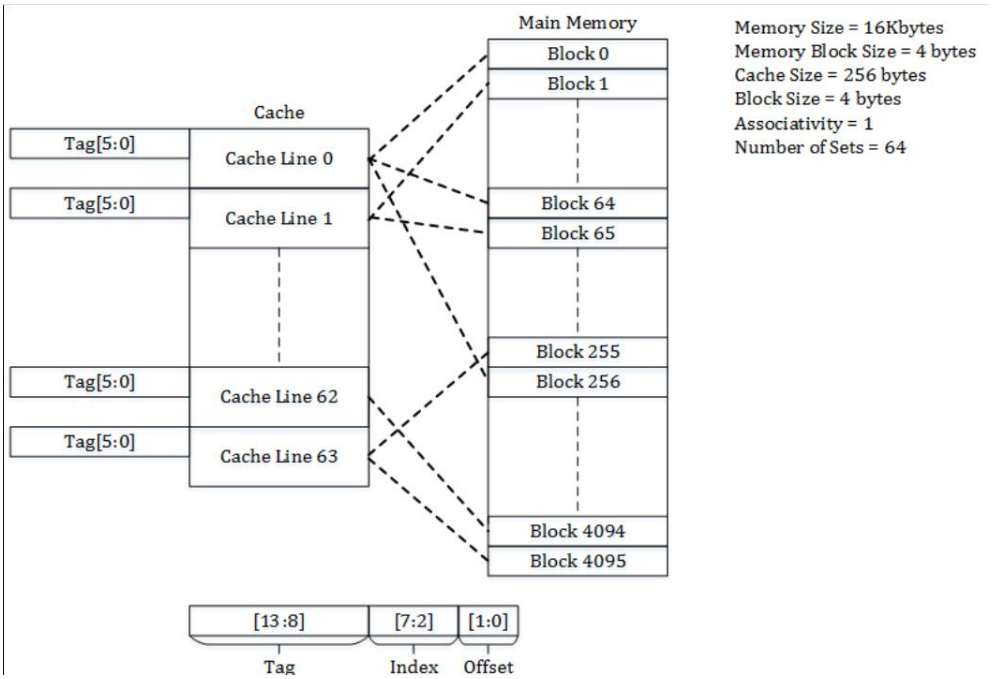
■ 3가지 종류의 cache 구조

1. Full Associative cache



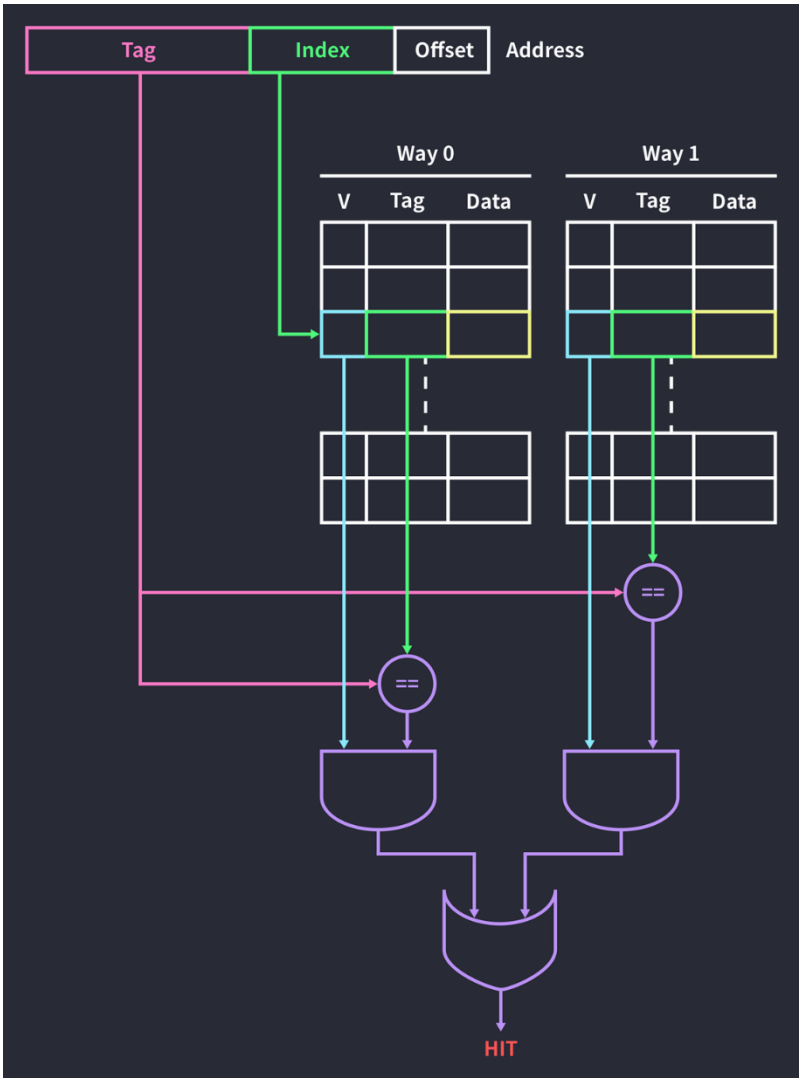
Main memory의 내용 cache line size에 따라 cache 라인에 복사하는데, 그 위치가 정해지지 않아서 아무 캐시 라인에 할당하는 구조이다. 그래서 Direct-mapped cache와 Set Associative cache와 달리 index bit가 없는 구조이다. Index bit가 없기 때문에 서로 다른 주소가 같은 index를 가리킬 때 충돌은 없다는 장점이 있다. 하지만 배정하는 위치가 정해지지 않았기 때문에 cache를 탐색할 때 cache 전체 내에서 찾아야 하기 때문에 속도가 느릴 수 있다는 단점이 있다.

2. Direct-mapped cache



Fully associative cache 구조와 달리 cache line에 할당하는 위치가 정해져 있는 구조이다. Index로 두어서 해당 위치에 Memory 내용을 옮긴다. Cache를 read할 때 해당 주소가 cache에 존재하는지 여부를 알기 위해서 해당 index만 찾아보면 되기 때문에 탐색하는데 시간이 단축이 된다는 장점이 있다. 하지만 tag bit가 다른 서로 다른 주소인데 index bit가 같게 나오는 경우가 있기 때문에 cache가 어느 데이터를 가져오면 될지 판단이 어려워지는 데이터 충돌이 발생하게 된다. 이는 cache miss을 야기하게 된다. 다시 해당 index cache line을 지우고 다시 쓰기에는 많은 데이터 충돌이 발생하기 때문에 복잡 해지는 단점이 있다.

3. Set-Associative Cache



Address를 인덱스를 통해서 cache에 접근하는 방법은 Direct mapped cache와 동일하다. 하지만 (그림은 two-way set associative cache)cache를 여러 개 사용하는 방법으로, direct mapped cache에서 발생하는 데이터 충돌을 감소시킬 수 있다는 장점이 있다. Way0에서 같은 index bit가 나왔을 시, way2에 같은 index bit를 할당해 준다. 만약 miss가 모든 way에서 발생한다면 하나의 way에서만 교체 정책을 적용한다. 필요한 데이터를 찾을 때 모든 way들을 탐색해야 한다는 특징이 있어 way가 많아질수록 느려진다.

- Cache Hit/Miss

  해당 데이터/주소가 cache에 존재한다면, cache hit 라고 부르며 cache에 존재하지 않을 때는 cache miss라 한다.

  1. Cold miss

     Cache가 초기화가 되지 않은 상태여서 발생하는 miss

  2. Conflict miss

     Write 할 때, tag bit가 다른 서로 다른 데이터/주소가 같은 index 일 때 일어나는 miss(direct mapped, set associative cache에서 발생)

  3. Capacity miss

     Cache의 용량이 부족해서 나타나는 miss

- Write policy

  1. Write through

     쓰기 정책에서 두 가지 방법이 있는데, write through, write back 방법이 있다.

     이 방법은 write를 memory와 cache에 동시에 수행하는 방법으로, cache와 memory의 내용이 항상 동일하기 때문에 따로 memory에 dirty bit가 1인 데이터 내용을 update할 필요가 없고 cache miss가 감소한다. 하지만 이는 많은 write를 하게 되므로 cache를 사용함으로써 효율을 높인다고 하기에는 단점이 될 수 있다.

  2. Write Back

     위의 방법과 달리 cache에만 주로 write를 하며, 바뀐 데이터가 있고 capacity miss가 있을 시에만 memory에도 write를 한다. 따라서 write through 방법 보다는 cache miss가 더 발생할 수 있다.

■ Replacement policy

Capacity miss가 발생했을 때 교체 정책을 이용해서 victim을 선택한 뒤 write를 한다.

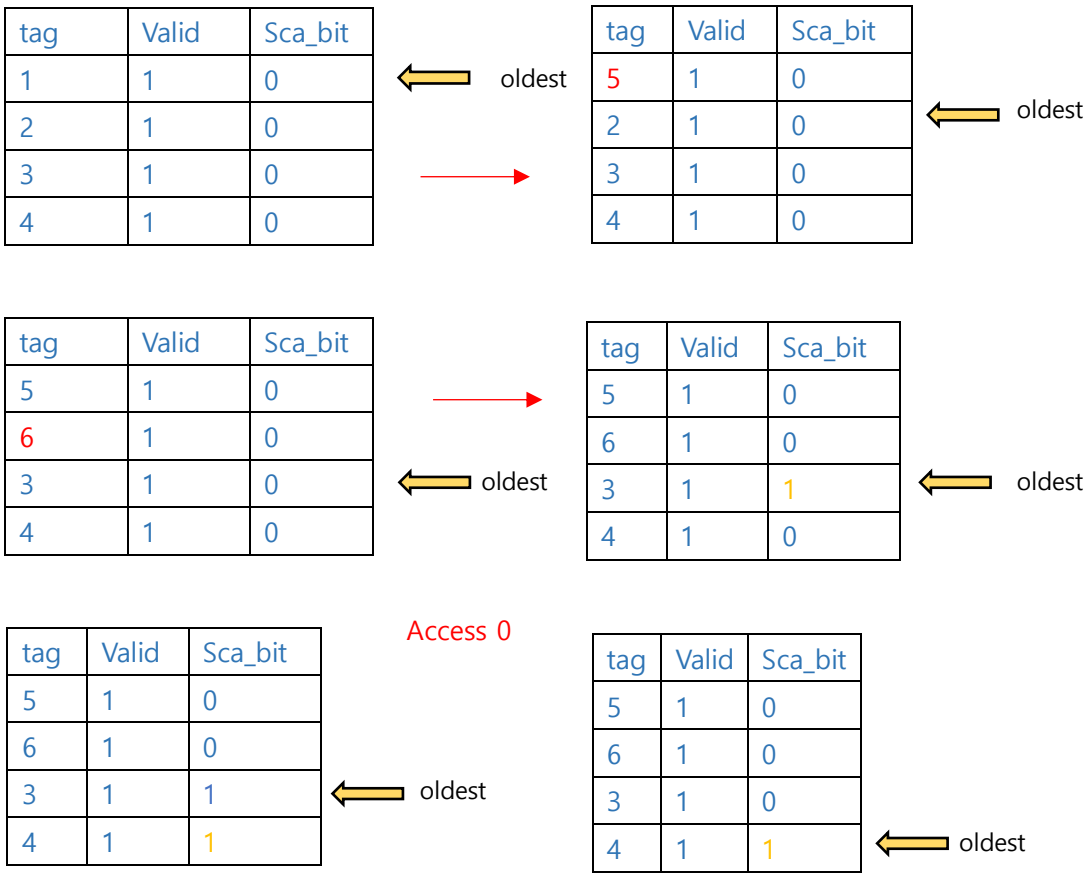
✓ LRU(least recently used)

가장 최근에 사용되지 않은 데이터를 희생시키는 방법으로, 접근했던 횟수가 매겨진다. 이 방법은 많은 데이터의 접근되었던 횟수를 다 기억해야 하기 때문에 복잡하다.

✓ SCA(Second Chance Algorithm)

최근에 접근했던 주소/데이터에게 두 번 기회를 주는 교체 정책이다. Cache의 효율을 높이기 위해서 가장 접근이 되지 않고 오래된 주소 라도 최근 접근이 되었을 때 sca bit를 1을 준다. 이 데이터가 oldest 데이터 일 때 기회를 더 주어서 다른 데이터가 희생되도록 하고 sca bit 가 1이었던 data는 다시 sca bit가 0이 된다.

Access pattern: 1, 2, 3, 4, 5, 6, 3, 4, 0, 6, 3



tag	Valid	Sca_bit
5	1	0
6	1	0
3	1	0
4	1	0

oldest

tag	Valid	Sca_bit
0	1	0
6	1	1
3	1	0
4	1	0

oldest

이와 같은 방법으로 두 번의 기회를 준다.

Capacity miss가 발생했을 때, 만약 victim address의 dirty bit가 1이라면, memory와 다른 데이터가 write된 상태이기 때문에 write back 방법을 쓴다면 memory에 update가 필요하다. 따라서 victim의 dirty bit ==1 이면 memory write + cache write가 같이 일어나야 한다.

4. Program Structure

■ **Cache 구조체**

```
typedef struct cache
{
    int tag;
    int valid;
    int data[64];//line size:64byte
    int dirty;
    int SCA;
    int oldest;
}CACHE;

CACHE cache[4];//data store size:256B, line store size:64B -> # of entries: 4.
```

교체 정책을 위해서 SCA, oldest 라는 멤버를 추가로 두었다. Offset bit은 data[64] 배열의 인덱스 역할을 해준다.

Cache line size, cache size는 정해주신 대로 64byte, 크기는 256B를 선택해서 구현했다. Entry 개수는 256B/64B = 4 이기 때문에 4개 인 것을 알 수 있다. 따라서 구조체 4개를 만들었다.

■ **Cache를 구현하기 위한 함수**

```
//cache
int Readcache(int address);
int Writecache(int address,int value);
void write_data(int vic,int value, int tag, int offset,int where);
void updateMem(int where,int vic);
int max_oldest();
```

Cache는 메모리의 축소판이라고 할 수 있으며 메모리 접근이 필요한 때에 cache에 접근한다. 따라서 Fetch단계에서 메모리로부터 명령어를 읽어 들이기 때문에 Readcache라는 함수를 호출한다. 그리고 Memory access 단계에서 sw(store word), lw(load word)에 해당하는 opcode일 때, 메모리 접근이 필요하다.

sw일 때는 메모리에 data 값을 write 하므로 Writecache를 호출하고 lw는 메모리에 있는 data를 읽어들이는 명령어이므로 Readcache함수를 호출한다.

함수의 길이가 너무 길어지지 않게 writecache 함수안에서 write data함수를 호출하고, capacity miss가 발생시, victim인 cache line이 dirty bit가 1 이라면 Memory에 data update를 하기 위해 updateMem()함수를 만들었다. Victim을 선택하는 과정에서 가장 오래된 cache line을 고르기 위한 max_oldest 함수를 만들었다.

■ **Fully Associative Cache bit 구성(32bit Address Space)**

Cache line size가 64B이므로 2^6 이다. 따라서 6비트가 offset bit에 해당하고 나머지 비트는 tag bit 역할을 한다. Fully associative cache는 index bit가 없기 때문에 이 두 가지 bit만 존재한다.

Cache가 주소가 들어있는지 그 주소가 유효한지를 나타내는 valid bit, write로 인한 cache의 내용이 바뀌었는지 여부를 나타내는 dirty bit도 추가적으로 필요하다.

■ **cache.c- main함수**

```
int main() {

    setting();
    initializeMemory();
    while (1) {

        if(pc == 0xFFFFFFFF){
            //when -1 come in at fetch stage, remain cycles are mem write update write.
            printf("■ ■ ■ ■ ■ ■ ■ ■ ■ cycle%d ■ ■ ■ ■ ■ ■ ■ ■ ■ \n",cycle);
            writeback(memwb_latch[1]);
            printf("pc: 0x%x\n",pc);
            printf("********************terminate***************\n");
            printout();
            break;
        }

        writeback(memwb_latch[1]);
        fetch();
        decode(ifid_latch[1]);
        execute(idex_latch[1]);
        memoryaccess(exmem_latch[1]);
        updatelatch();
    }
}
```

기존의 pipeline 구조에서 cache 기능만 추가한 구조라고 할 수 있다.

Pipeline 구조와 같이 각 latch와 cache를 초기화를 한 뒤, 5단계 Fetch, decode, execute, memory access, write back 함수를 실행한다. decode단계와 write back단계에서의 Data dependency를 막고자 writeback함수를 먼저 실행시켰다.

PC가 -1를 가리켰을 시 남은 cycle 단계가 writeback만 남게 된다. 그래서 writeback 함수를 실행하고 결과값을 출력시키는 구조다.

■ **function_cac.c - fetch함수**

```
int fetch() {
    ifid_latch[0].branchPresig = one_LV_branch_predic();
    ifid_latch[0].inst = Readcache(pc);
    ifid_latch[0].pc4 = pc + 4;
    ifid_latch[0].cycle = cycle;
    printf("■ ■ ■ ■ ■ ■ ■ ■ ■ cycle%d ■ ■ ■ ■ ■ ■ ■ ■ ■ \n",cycle);
    printf("pc: 0x%x\n",pc);
    printf("[Fetch] 0x%08x\n", ifid_latch[0].inst);
    pc = pc + 4;
    mux(ifid_latch[0].branchPresig);
    cycle++;
}
```

ifid_latch[0].inst = Memory[pc/4];

Project3에서 구현했던 pipeline 구조 일 때 와 바뀐 점을 비교해보면 위의 그림과 같다. 메모리에서 instruction을 읽지 않고 cache를 이용해서 읽어온다.

■ **function_cac.c – memory access 함수**

```
void memoryaccess(EXMEM_latch exmem_input) {

    if (exmem_input.control.MemtoReg == 1 && exmem_input.control.MemRead == 1)//lw
    {
        memwb_latch[0].v5 = Readcache(exmem_input.v3);
    }
    else if (exmem_input.control.MemWrite == 1 && exmem_input.control.MemRead == 0)//sw
    {
        Writecache(exmem_input.v3,exmem_input.v2);
    }
    else memwb_latch[0].v5 = exmem_input.v3;//ALU result
```

memwb_latch[0].v5 = Memory[exmem_input.v3 / 4];

Memory[exmem_input.v3 / 4] = exmem_input.v2;

Project3에서의 구조와 바뀐 점은 위의 그림과 같다. Memory access를 하는 lw 때, Readcache를 하여 값을 불러온다. sw 때, Writecache를 호출해서 cache에 data를 write해서 저장한다.

■ **Cache_func.c – Readcache 함수**

✓ Cache hit

```
for(int i = 0; i < 4; i++){

    if(cache[i].valid == 1 && cache[i].tag == tag)//if there the address in the cache
    {//cache hit
        miss = 0;
        cache[i].oldest = oldest;
        oldest++;
        cache[i].SCA = 1;//when cache hit, value is
        cache_hit++;
        return cache[i].data[offset];
    }
}
```

Cache 엔트리 수가 4 이므로 for 문을 이용해서 해당 주소를 가진 명령어가 있는지 확인한다. 만약 있으면 cache hit 이며, 해당 데이터 값을 반환한다. 나중에 희생자를 고르기 위해서 SCA, oldest 순위를 준다. 여기서 oldest 변수는 전역변수로, hit 나 write 가 될 때 갱신되고 가장 오래된 cache line 은 가장 작은 oldest 순위를 가진다.

✓ Cache miss -capacity miss

```
if(miss == 1){
    int sig = 0;
    for(int i = 0; i < 4; i++){// but if cache is full(capacity miss) --> make victim
        if(cache[i].valid == 0){
            sig = 1;//if cache is not full
        }

    }
    if(sig == 0){//if cache is full-capacity miss
        des = Writecache(address,Memory[address/4]);
        cache_miss++;
        return cache[des].data[offset];
    }
```

Cache miss 가 발생시, capacity miss 인지 cold miss 인지 구분하기 위해 for 반복문으로 모든 cache line 의 valid bit 가 1 인지 확인한다. 하나라도 0 이 있다면 cold miss 이므로 sig 라는 변수를 두어 각 다른 miss 일 때 실행되는 부분을 구분했다.

Capacity miss 일 때는 victim 을 정하고 write 를 해야 하므로 Writecache 함수를 호출했다.

✓ Coldmiss

```
else{
    cache_miss++;
    cache[index_cache].valid = 1;
    cache[index_cache].tag = tag;
    cache[index_cache].oldest = oldest;
    cache[index_cache].dirty = 0;
    cache[index_cache].SCA = 0;
    for(int i = 0; i < 64; i ++){

        cache[index_cache].data[i] = Memory[where+i];
    }
    result =  cache[index_cache].data[offset];
    index_cache++;
    oldest++;
    return result;
}
```

Cold miss 가 났을 때, 다시 새로 write 를 해야 하므로 각 필요한 cache 구조체 변수들에 값을 할당한다. 이 때 cache line size 2^6 이므로 offset 이 6bit 였으므로 64 개의 데이터가 들어갈 수 있게 되기 때문에 for 반복문을 이용해서 해당 Memory 주소로부터 cache 에 값을 채운다. Fully associative cache 가 index bit 가 없지만 index 가 있는 것처럼 index_cache 를 전역변수로 두어 순서대로 값을 write 했다.

■ **Cache_func.c – Writecache 함수**

✓ Cache hit

```
if(hit == 1)//when cache hit
{
    cache[locate].data[offset] = value;// write the value
    cache[locate].dirty = 1;
    cache[locate].oldest = oldest;
    oldest++;
    cache[locate].SCA = 1;
    cache_hit++;
}
```

Cache hit 일 때, 맞는 cache 위치에 offset bit 에 해당하는 곳에 value 를 write 한다.

✓ Cache miss – capacity miss

```
else{//capacity miss
    //select victim
    cache_miss++;
    least = -1;
    int old = max_oldest();
    victim = max_oldest();//give the oldest one to victim

    while(1){
        int i;
        for(i = 0; i < 4 ; i++){
            if((victim > cache[i].oldest) && (cache[i].oldest > least))
            {
                victim = cache[i].oldest;
                vic = i;
            }
        }
        if(cache[vic].SCA == 1){//if sca ==1, have to choose another victim(the next oldest victim)
            cache[vic].SCA = 0;
            least = victim;
            victim = old;
            continue;
        }
        else break;
    }//while
    check = 1;
    if(cache[vic].dirty == 1)//have to write memory
    {
        updateMem(where,vic);
    }
    write_data(vic,value,tag,offset,where);//write the value
    printf("vic: %d\n",vic);
    return vic;
}
```

Capacity miss 가 생겼을 때는 victim 을 정해야 하는데,

① 가장 oldest 가 작은 수를 찾는다.

② 해당 cache line 의 SCA bit 가 1 이면

③ Cache 내용을 tag 에 해당하는 메모리 주소에 update 한다. (updateMem 함수 호출)

④ Cache 에 새로운 tag bit 와 data 들을 할당하고 필요한 value 를 write 한다. (write data 함수 호출)

Cold miss 일 때 write 하는 방식은 Readcache 와 동일한 방법으로 구현했다.

5. Problems and solutions

1. For 반복문을 이용한 cache size 지정 실수

   Fully associative cache 구조를 구현하려고 했기 때문에 for 반복문을 이용해서 전체 캐시 내용을 탐색하는 방법을 썼었다. 처음에는 cache 크기대로 상수로 반복 횟수를 지정했으나, 다시 size of (cache)/size of(int)를 이용해서 반복횟수를 정했다. 여기서 cache는 구조체라, 가장 큰 멤버 크기로 size of 값이 나오는 것을 잊고 cache를 배열처럼 생각했다. 그래서 처음에 결과 값이 다른 값이 나와서 곧바로 수정할 생각을 못했다. 알고 보니 사이즈를 잘못 생각해서 그런 오류가 발생한 거였다. 그래서 다시 상수로 고쳐서 문제를 해결했다.

2. Replacement policy 어떻게 구현할 것인지

SCA 방법을 적용했지만 oldest인 캐시라인도 구분을 해줘야 하기 때문에 어떻게 할지 고민이 많았다. 그래서 oldest라는 전역변수를 지정해서 cache에 새로운 라인이 write 될 때마다 oldest++을 해줘서 겹치는 oldest 순위가 없도록 만들고 oldest 수가 가장 작은 캐시라인을 victim으로 만드는 방식으로 구현했다.

그 다음으로는 oldest인 해당 라인이 SCA bit가 1일 때의 경우가 있을 때는 어떻게 구현할지 고민했다. 그래서 조건문 if로 해당 oldest 다음으로 작은 oldest를 찾는 반복문을 썼다. 처음에 비교하는 대상을 잘못 지정해서 조금 오랫동안 디버깅을 해야 했다.

3. Write back 구현 실수

처음에 캐시를 구현하려고 생각한 함수 구조는 대략 잘 갖췄지만 memory update 함수를 만들 때 실수가 있었다. victim이 되는 cache tag bit에 해당되는 메모리 주소에 update를 해야 하는 데 그 주소를 새로 캐시에 쓰려는 address 주소로 잘못 코드를 구현해서 capacity miss가 발생하는 input4 파일에서 결과 값이 다르게 나오는 오류가 발생했다. 차근차근 어느 부분에서 오류가 났는지 확인했고 capacity miss가 시작되는 부분을 찾은 덕분에 수정을 할 수 있었다.

6. Build Environment

Compilation: Linux Assam, with GCC

To compile, please type:

gcc -o cache cache.c branch.c function_cac.c cache_func.c pipeline.h

To run, please type:

./cache

Input4파일에서는 실행파일을 만들어서 실행했습니다.

- 실행파일 실행 환경

  gcc branch.c function_cac.c cache.c cache_func.c pipeline.h -o cache_pip.o

  ./cache_pip.o> cache_input4

  vi cache_input4

7. Screen capture

① Simple.bin

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle10 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback-------------------
pc: 0x18
Reg[29]: 0xffff8 ----> 0x100000
pc: 0xffffffff
*******************terminate*****************
******************result*********************
reg[2] (final return)value: 0
 # of instructions: 12
 # of memory operation instructions: 2
 # of register operation instructions: 8
 # of cache hit: 10
 # of cache miss: 2
 # of branch instructions: 0
 # of not-taken branches: 0
 # of jump instructions: 0
*****************************************
```

② Simple2.bin

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle12 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback-------------------
pc: 0x20
Reg[29]: 0xfffe8 ----> 0x100000
pc: 0xffffffff
*******************terminate*****************
******************result*********************
reg[2] (final return)value: 100
 # of instructions: 12
 # of memory operation instructions: 4
 # of register operation instructions: 9
 # of cache hit: 14
 # of cache miss: 2
 # of branch instructions: 0
 # of not-taken branches: 0
 # of jump instructions: 0
*****************************************
```

③ Simple3.bin

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle1339 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback-------------------
pc: 0x64
Reg[29]: 0xfffe8 ----> 0x100000
pc: 0xffffffff
*******************terminate*****************
******************result*********************
reg[2] (final return)value: 5050
 # of instructions: 28
 # of memory operation instructions: 614
 # of register operation instructions: 1029
 # of cache hit: 1950
 # of cache miss: 2
 # of branch instructions: 102
 # of not-taken branches: 1
 # of jump instructions: 1
*****************************************
```

④ Simple4.bin

```
■ ■ ■ ■ ■ ■ ■ ■ cycle292 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback-------------------
pc: 0x28
Reg[29]: 0xfffe0 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 55
 # of instructions: 44
 # of memory operation instructions: 101
 # of register operation instructions: 219
 # of cache hit: 389
 # of cache miss: 3
 # of branch instructions: 10
 # of not-taken branches: 1
 # of jump instructions: 11
******************************************
```

⑤ Fib.bin

```
■ ■ ■ ■ ■ ■ ■ ■ cycle3282 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback-------------------
pc: 0x34
Reg[29]: 0xfffd8 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 55
 # of instructions: 56
 # of memory operation instructions: 1116
 # of register operation instructions: 2405
 # of cache hit: 4374
 # of cache miss: 3
 # of branch instructions: 109
 # of not-taken branches: 55
 # of jump instructions: 164
******************************************
```

⑥ Gcd.bin

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle1267 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback-------------------
pc: 0x40
Reg[29]: 0xfffd0 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 1
 # of instructions: 64
 # of memory operation instructions: 489
 # of register operation instructions: 938
 # of cache hit: 1744
 # of cache miss: 12
 # of branch instructions: 73
 # of not-taken branches: 28
 # of jump instructions: 65
*****************************************
```

⑦ Input4.bin

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle23374691 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback-------------------
pc: 0x18eec
Reg[29]: 0xf8010 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 85
 # of instructions: 25536
 # of memory operation instructions: 7117466
 # of register operation instructions: 20318758
 # of cache hit: 28499389
 # of cache miss: 3983569
 # of branch instructions: 2029699
 # of not-taken branches: 869
 # of jump instructions: 103
*****************************************
```

8. Personal feeling

캐시를 어떻게 함수로 구현할지는 개념을 다 이해하고 나니 바로 코드로 구현을 할 수 있을 것 같았다. 하지만 언제 어느 부분에서 적용해야 할지 초반에는 감이 오지 않았다. 처음에 캐시가 어떤 것인지 제대로 와 닿지 못했던 것 같다. 가장 최근 강의 자료를 보면 바로 알 수 있지만 바로 확인하지 않았다. 캐시가 메모리 사용의 효율을 높이기 위해서 사용하는 것인데 당연히 메모리 접근이 필요할 때 캐시를 사용하면 될 거라는 생각을 못하고 모든 5단계를 생각하면서 이 모든 단계에서 어떻게 다 캐시를 적용할까 고민을 했던 것이 코드 구현시작을 할 때 더디게 만든 것 같다.

파이프라인 구현을 할 때 수정을 많이 했던 경험 때문에 캐시를 구현하는 것은 조금 더 수월했다. 제일 처음 프로젝트1을 시작할 때와 비교했을 때 이제는 개념만 이해하면 어떻게 구현할지 머리 속에서 대략 구조를 만들 수 있게 된 것 같다. 이렇게 차근차근 한 단계씩 생기는 프로젝트를 할 때마다 게임에서 점점 높은 스테이지를 깨는 느낌이었다. 그만큼 재밌게 구현을 한 것 같고 실력도 조금은 좋아지지 않았을까 스스로 기대를 했다. 이번 프로젝트4에서 아쉬운 것은 오랜 시험기간 준비로 인해 너무 지쳤고 건강이 안 좋아져서 더 여러가지 방법으로 구현하지 못한 것이다. Direct-mapped cache나 set associative cache와의 성능에 대해 생각을 해본다면 일단 fully associative cache가 conflict miss가 발생하지 않기 때문에 direct-mapped cache보다는 cache hit율이 높을 것 같고 index를 사용하지 않아서 데이터가 존재하는지 확인하는데 걸리는 시간은 더 오래 걸릴 것이다. Set associative(2way 나 4way)cache는 fully에 비해 더 많은 data를 저장할 수 있고 index를 사용하기 때문에 탐색시간도 덜 걸릴 것이고 capacity miss 도 줄어들고 conflict miss도 way가 커질수록 줄어들 것이다. 좀 더 빠르고 miss을 낮추기 위한 방법은 set associative cache구조가 적합한 것 같다. 더 복잡한 구조가 역시 성능에 신경을 많이 쓰는 것 같다.

이번 학기 컴퓨터 구조 수업 덕분에 컴퓨터 구조에 대한 개념 이해와 코드 구현이 모두 필요했던 프로젝트를 하게 되었고 이 과제는 개인적으로 도움이 많이 되어서 좋았다. 또 끈기도 기를 수 있었던 것 같다.