Mobile Processor Programming

assignment #4

: Single Cycle MIPS emulator with cache

모바일시스템공학과 32217259 문서영

[목차]

1. 개요

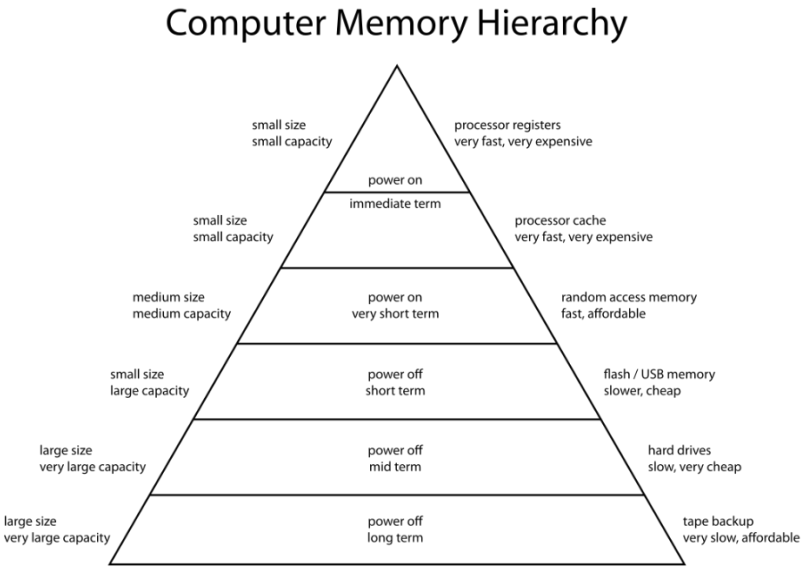
마지막 과제는

언어는 c 언어를 사용하였고, visual studio code 에서 작업하였습니다.

2. 배경지식

1. 메모리 계층 구조

메모리 계층 구조(memory hierarchy)는 메모리를 필요에 따라 여러 가지 종류로 나누는 것을 말합니다.



위의 사진처럼 레지스터와 캐시는 CPU 내부에 존재하기 때문에 매우 빠르게 접근 가능합니다. 메모리는 CPU 외부에 존재하기 때문에 더 느리게 접근합니다. 또한 하드 디스크는 CPU 가 직접 접근할 방법조차 없기에 CPU 가 하드 디스크에 접근하기 위해선 하드 디스크의 데이터를 메모리로 이동시키고 메모리에서 접근하는 매우 느린 접근만 가능합니다.

2. Cache 란

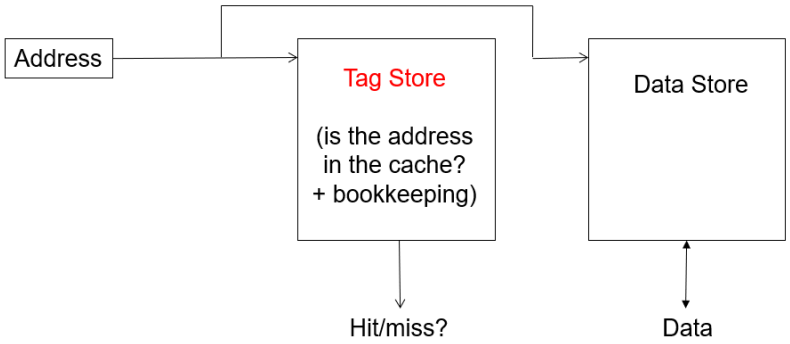
Cache 는 데이터나 값을 미리 복사해 놓은 임시 장소를 말합니다. 캐시에 데이터를 미리 복사해 놓으면 계산이나 접근 시간 없이 더 빠른 속도로 데이터에 접근할 수 있습니다.

CPU 에서 말하는 캐시는 CPU 구조에 메모리로 사용하도록 구성된 하드웨어 캐시로, 크기는 작지만 빠른 메모리입니다. 메모리 계층 구조에 나와있는 것처럼 CPU 내부에 존재하기 때문에 캐시 메모리에 복사해 두면 CPU 밖의 메모리보다 빨리 접근하여 평균 메모리 접근 시간을 아낄 수 있습니다.

캐시는 메모리에 접근할 때 byte 보다 큰 block 단위로 접근합니다.64 byte 또는 32 byte 를 사용합니다. 주소 크기보다 크게 사용하는 이유는 매번 메모리에 접근하여 데이터를 가져오는 것은 비용이 크기 때문에 메모리 한 번 접근할 때마다 많이 가져오는 것입니다.

메모리에서 가져온 데이터를 관리하기 위해 tag 메모리를 사용합니다. Tag 메모리는 캐시의 데이터 메모리의 블록과 쌍을 이룹니다.

Tag 메모리는 주로 tag, valid bit, dirty bit 이 세 가지 정보를 포함합니다. Tag 는 CPU 가 요청한 블록을 찾는데 사용하는 주소 정보의 일부이고, valid bit 는 캐시 블록이 유효한 데이터인지 아닌지를 표시하며, dirty bit 는 메모리에서 캐시로 블록을 가져온 후 CPU 가 블록을 수정했는지 표시합니다.



[캐시 구조의 추상화]

Cache 는 메모리 접근할 때 발생하므로 만약 address 가 cache 에 존재한다면 Hit 로, cache 에서 바로 데이터를 가져옵니다. 반대로 cache 에 데이터가 존재하지 않는다면(tag 와 값이 다르다면) memory 에서 cache 로 데이터를 가져오도록 합니다.

Cache hit rate = (hits 수)/(hits 수 + misses 수) = (hits 수)/(메모리 접근 수)

Average memory access time (AMAT) = (hit-rate * hit-latency) + (miss-rate * miss-latency)

캐시 라인(캐시 블록)은 캐시의 저장 단위로, direct mapping, associate mapping, set associative mapping 이 있습니다.

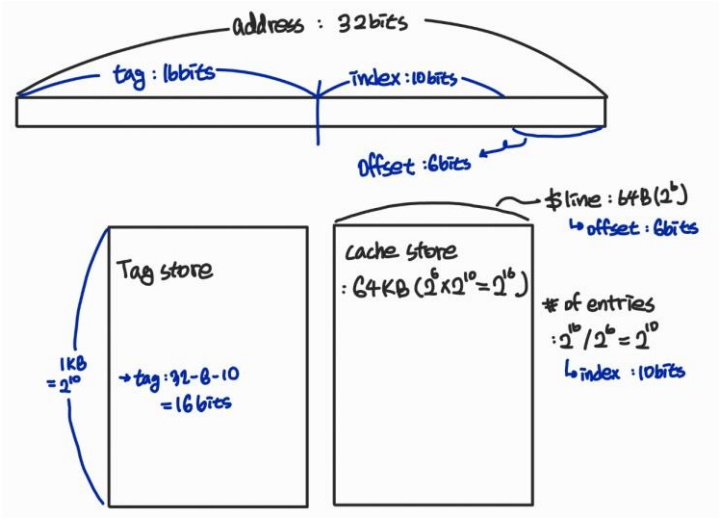
Direct mapping: 가장 간단한 캐시 매핑 방법으로, 메모리 블록은 고정된 위치에만 저장될 수 있습니다. Address 를 tag, index, offset bit 로 나누며, 특정 index 에서 tag 가 같다면, 특정 index 에 대한 특정 offset 에 address 에 대한 data 가 저장되어 있습니다. 구현이 간단하고 비용이 저렴하나, 캐시의 특정 라인에 많은 메모리 블록이 mapping 될 경우, 캐시의 충돌이 빈번하게 발생할 수 있습니다.

Associative mapping: 메모리 블록을 캐시의 어느 위치에나 저장할 수 있는 mapping 방식입니다. 메모리 블록은 캐시의 어느 라인에도 저장될 수 있기에 모든 캐시 라인이 연관된 구조입니다. Tag 와 offset 으로 이루어져 있으며, 원하는 위치에 tag 를 저장하고 data store 안에 offset 만큼 떨어진 곳에 data 를 저장합니다. 그렇기에 캐시 충돌이 거의 발생하지 않고 Hit 의 확률이 많아지나, 구현이 복잡하고 비용이 높습니다. 또한 모든 캐시 라인을 검사해야 하기 때문에 검색 시간이 길어질 수 있습니다.

Set associative mapping: direct mapping 과 associative mapping 의 절충안으로, 캐시를 여러 세트로 나누고 각 세트 내에서 associative mapping 을 사용합니다. 구조는 캐시를 n 개의 세트로 나누고, 각 세트는 m 게의 라인을 가집니다. 메모리 블록은 associative mapping 처럼 특정 세트 내의 어느 라인에도 저장될 수 있습니다. 단점이 없진 않지만 가장 절충안으로 나온 방식이기에 실생활에서 많이 사용됩니다.

3. 코드 설명(작성 과정 및 오류 설명)

- 초기 설정



[캐시 구조 초기화]

위의 사진은 캐시 기본 구조는 주소가 32bit 으로 설정되어 있고, cache line 을 64byte, cache store 를 64KB 로 설정했을 때의 모습입니다.

```
struct cacheline {
    unsigned int tag : 16; // among 32 bit, use only a few bits
    unsigned int sca : 1; // 자주 사용하는 값 지우는 거 방지 위해 체크!
    unsigned int valid : 1;
    unsigned int dirty : 1;
    int data[64];

void print_cache(struct cacheline cache, int a);
```

[cache.h]

위와 같이 설정했을 때, 캐시 라인의 구조체는 위와 같습니다.

```
struct cacheline cache[1024]; // offset : 6bits (direct mapping)
void InitCache();
bool AllValid();
void PutCache(int adr, int data);
int ReadFromCacheline(int adr, int offset, int idx);
void DrainTheCacheline(int idx, int offset, int adr);
void FillInCacheline(int idx, int tag, int offset, int adr);
void CheckCache(int* returnVal, int adr);
```

```
int ReadMem(int adr){
        return mem[adr/4];
}

void WriteMem(int adr, int val){
        mem[adr/4] = val;
}
```

캐시는 direct mapping 을 사용하였고, 캐시를 생성하기 위해 사용한 함수는 위와 같습니다. 또한 메모리에서 데이터를 쉽게 가져오기 위해 각 함수를 만들었습니다.

<각 함수 설명>

void InitCache(): cache 구조체에서 valid, dirty, sca bit 를 초기화하였습니다.

bool AllValid(): cache 전체의 valid 를 확인하여 1 이면 true 를 리턴하고, 0 이면 false 를 리턴합니다.

void PutCache(int adr, int data): sw 에서 사용하는 함수로, 만약 tag 가 cache 의 tag 와 다르다면, DrainTheCacheline 과 FillInCacheline 함수를 사용하여 캐시를 메모리에서 다시 가져오도록 합니다. 또한 cache 의 데이터를 채우고, dirty bit = 1, valid bit = 1 로 만듭니다.

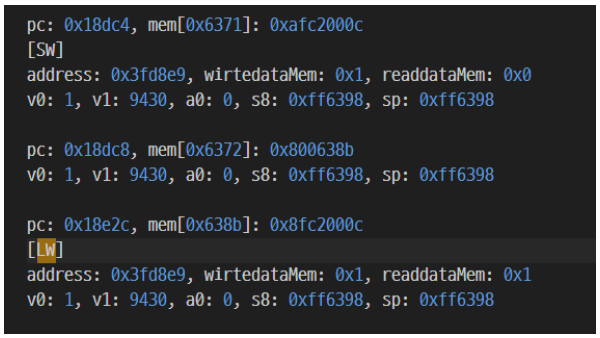
int ReadFromCacheline(int adr, int offset, int idx): cache 에서 데이터를 사용하기 위해 address 에 해당하는 데이터를 리턴합니다.

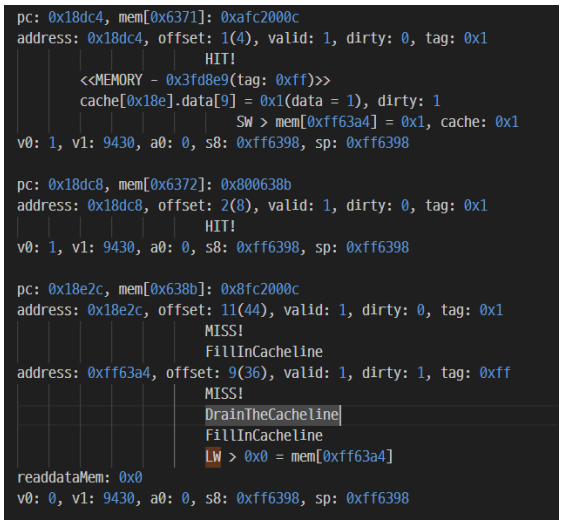
void DrainTheCacheline(int idx, int offset, int adr): 만약 cache 가 dirty 한데 valid 하다면(tag 가 다른 상태), WriteMem 함수를 통해 원래 존재하는 tag 에 대한 데이터를 메모리로 옮겨줍니다.

void FillInCacheline(int idx, int tag, int offset, int adr): address 에 해당하는 tag 를 채워주고, valid = 1 로 만들어줍니다. 또한 현재 address 에 해당하는 index 에 대한 캐시 블록을 채워줍니다.

<코드 오류 설명>

void CheckCache(int* returnVal, int adr): pc 값과 lw 에 해당하는 값을 채울 때 사용하는 함수로, address 에 해당하는 데이터는 returnVal 로 가져옵니다. 만약 address 에 대해 cache 의 tag 와 같고, valid = 1 이며 dirty 하지 않으면 Hit 로, ReadFromCacheline 함수를 통해 값을 바로 가져옵니다. 그렇지 않다면 Miss 로, DrainTheCacheline 과 FillinCacheline 함수를 통해 캐시를 다시 채워주고 그에 대한 데이터를 가져옵니다.





PC 에서는 아무런 문제없이 잘 작동되는 캐시가 input4.bin 을 동작시키면 무한 루프에 빠져 single mips 와 cache mips 에서의 값 비교를 통해 오류를 확인했더니

18dc4:	afc2000c	sw	v0,12(s8)
...			
18e2c:	8fc2000c	lw	v0,12(s8)

위의 부분에서 문제가 발생하였습니다. sw 명령어를 통해 mem[0xff63a4]에 0x1 의 값을 넣었는데, lw 를 통해 v0 에 저장되는 값은 0x0 였습니다.

```
pc: 0x18e2c, mem[0x638b]: 0x8fc2000c
address: 0x18e2c, offset: 11(44), valid: 1, dirty: 0, tag: 0x1
                        MISS!
        cache[0x238].data[11] = 0x24021dc2
        cache[0x238].data[11] = 0x24021dc2
                        FillInCacheline
        cache[0x238].data[11] = 0x8fc2000c
address: 0xff63a4, offset: 9(36), valid: 1, dirty: 1, tag: 0xff
                        MISS!
        cache[0x18e].data[9] = 0x1
                        DrainTheCacheline
        cache[0x18e].data[9] = 0x1
                        FillInCacheline
        cache[0x18e].data[9] = 0x0
                        LW > 0x0 = mem[0xff63a4]
readdataMem: 0x0
v0: 0, v1: 9430, a0: 0, s8: 0xff6398, sp: 0xff6398
```

위의 사진처럼 FillInCacheline 함수를 지나고 값이 0x1 에서 0x0 으로 변하는 것을 볼 수 있었습니다.

```
if(cache[idx].tag != tag) FillInCacheline(idx, tag, offset, adr);
```

위의 과정을 통해 v0 의 값은 해결했는데, v1 의 값에 문제가 생겼습니다.

```
v0: 12, v1: -1404862460, a0: 0, s8: 0x0, sp: 0x1000000
```

HEX	FFFF FFFF AC43 8004
DEC	-1,404,862,460

확인해보니 원래 나와야 하는 값에서 앞의 bit 에 f 가 붙어 음수화 되어 있었습니다. FillInCacheline 에서 원래 cache address 의 시작점을 ppt 에 나온 대로 *adr & 0xffffffc0* 을 사용했다가,

```
int dst = (tag << 16) | (idx << 6);
```

원래 있는 데이터인 tag 와 index 정보를 사용하여 캐시 데이터 시작점을 지정하니 음수값이 나오지 않았습니다.

4. 출력 결과

1. simple.bin

```
----------------------------------------
return value(value in r2) : 0(0x0)
total number of executed instruction : 8
number of memory (load/store) operations : 2
number of register operations : 6
number of branches (total/taken) : 0
cache hit/miss (cold miss or conflict miss) : Hit-7, Miss-2
----------------------------------------
```

```
  // printf("dst: 0x%x\n", dst);
  if(cache[idx].tag != tag) {
          cache[idx].tag = tag;
          for(int i=0;i<16;i++){
f DEBUG_CACHE2
                  printf("cache data: 0x%x -> mem[0x%x] = 0x%x\n
f
                  cache[idx].data[i] = ReadMem(dst + i*4);
f DEBUG_CACHE23
   printf("adr: 0x%x -> cache[0x%x].data[0x%x] = 0x%x\n", dst+i*4
f
          }
  }
```

위와 같은 구조로 작성하면 simple.bin과 simpl2.bin만 결과가 나오고, instruction 개수도 다르지 않게 나오지만, 위의 사진에서 if문을 빼고 작성하면 instruction 개수가 매우 이상하게 나오지만 inpu4.bin 빼고 결과가 잘 나옵니다..

```
----------------------------------------
return value(value in r2) : 0(0x0)
total number of executed instruction : 4195260
number of memory (load/store) operations : 2
number of register operations : 4195258
number of branches (total/taken) : 0
cache hit/miss (cold miss or conflict miss) : Hit-3933055, Miss-262206
----------------------------------------
```

2. simple2.bin

```
----------------------------------------
return value(value in r2) : 0(0x0)
total number of executed instruction : 10
number of memory (load/store) operations : 4
number of register operations : 6
number of branches (total/taken) : 0
cache hit/miss (cold miss or conflict miss) : Hit-10, Miss-2
----------------------------------------
```

```
----------------------------------------
return value(value in r2) : 100(0x64)
total number of executed instruction : 4195262
number of memory (load/store) operations : 4
number of register operations : 4195258
number of branches (total/taken) : 0
cache hit/miss (cold miss or conflict miss) : Hit-3933058, Miss-262206
----------------------------------------
```

3. simple3.bin

```
----------------------------------------
return value(value in r2) : 5050(0x13ba)
total number of executed instruction : 4196582
number of memory (load/store) operations : 613
number of register operations : 4195866
number of branches (total/taken) : 102
cache hit/miss (cold miss or conflict miss) : Hit-3919220, Miss-277769
----------------------------------------
```

4. simple4.bin

```
----------------------------------------
return value(value in r2) : 55(0x37)
total number of executed instruction : 4195495
number of memory (load/store) operations : 100
number of register operations : 4195374
number of branches (total/taken) : 10
cache hit/miss (cold miss or conflict miss) : Hit-3917968, Miss-277586
----------------------------------------
```

5. fib.bin

```
----------------------------------------
return value(value in r2) : 55(0x37)
total number of executed instruction : 4197931
number of memory (load/store) operations : 1095
number of register operations : 4196563
number of branches (total/taken) : 109
cache hit/miss (cold miss or conflict miss) : Hit-3920786, Miss-277746
----------------------------------------
```

5. gcd.bin

```
----------------------------------------
return value(value in r2) : 1(0x1)
total number of executed instruction : 4196313
number of memory (load/store) operations : 486
number of register operations : 4195689
number of branches (total/taken) : 73
cache hit/miss (cold miss or conflict miss) : Hit-3919039, Miss-277607
----------------------------------------
```

6. input4.bin

```
----------------------------------------
return value(value in r2) : -1346240076(0xafc201b4)
total number of executed instruction : 23371126
number of memory (load/store) operations : 7107626
number of register operations : 14233698
number of branches (total/taken) : 2029699
cache hit/miss (cold miss or conflict miss) : Hit-28420567, Miss-1040640
----------------------------------------
```

답을 구하려 하였으나, 시간이 부족하고 도대체 어디서 잘못된 결과를 내는 건지 알 수 없어 이대로 찍어 보냅니다.

5. 어려웠던 점 및 후기

캐시에 대한 MIPS 가 파이프라인 MIPS 보다 쉬울 것이라고 생각하여 처음엔 Set Associative Mapping 을 선택하려 했던 만큼 가볍게 여겼습니다. 그러나 메모리만 건드렸을 뿐인데도 오류가 발생하여 더 어렵게 느껴졌습니다. 파이프라인은 전체적인 구조를 생각하며 만들면 되기에 오류를 비교적 쉽게 찾을 수 있었지만, 캐시는 개념 자체를 이해하기가 힘들었습니다. 특히, 적용할 때 PC 는 잘 작동하는데 LW 에서는 작동하지 않아 어디서 문제가 생긴 것인지 알기 어려웠습니다. 파이프라인보다 쉽고, 개념도 적으니 만만하게 본 결과였던 것 같습니다.

마지막 과제라 가장 완벽하게 제출하고 싶었지만, 가장 만족스럽지 않은 결과가 나와 아쉬웠습니다. 학기를 마무리하면서 MIPS 의 구조를 대략적으로 알게 되었고, 코드가 실행되는 동안 어떤 일이 발생하는지 자세히 살펴볼 수 있는 기회를 갖게 되어 어렵지만 유익한 수업이었습니다. 과제가 진행되는 동안 이렇게 C 언어로 긴 코드를 작성하는 게 처음이었기에, 차근차근 쌓아 올린 코드가 아니라 오류를 고쳐가면서 중구난방으로 펼쳐져 있어 알아보기 힘들기도 했습니다. 이 과정을 통해 어떤 식으로 코드 작업을 해야 하는지도 알게 되었습니다.

이런 과제를 하면서 긴 코드에 대한 두려움이 줄어든 것 같습니다. 비록 마지막 과제가 만족스럽지 못한 결과를 냈지만, 이를 통해 많은 것을 배울 수 있었고 이번 학기의 수업 중 가장 유익한 수업이었던 것 같습니다. 앞으로의 학습과 프로젝트에 큰 도움이 될 것입니다.