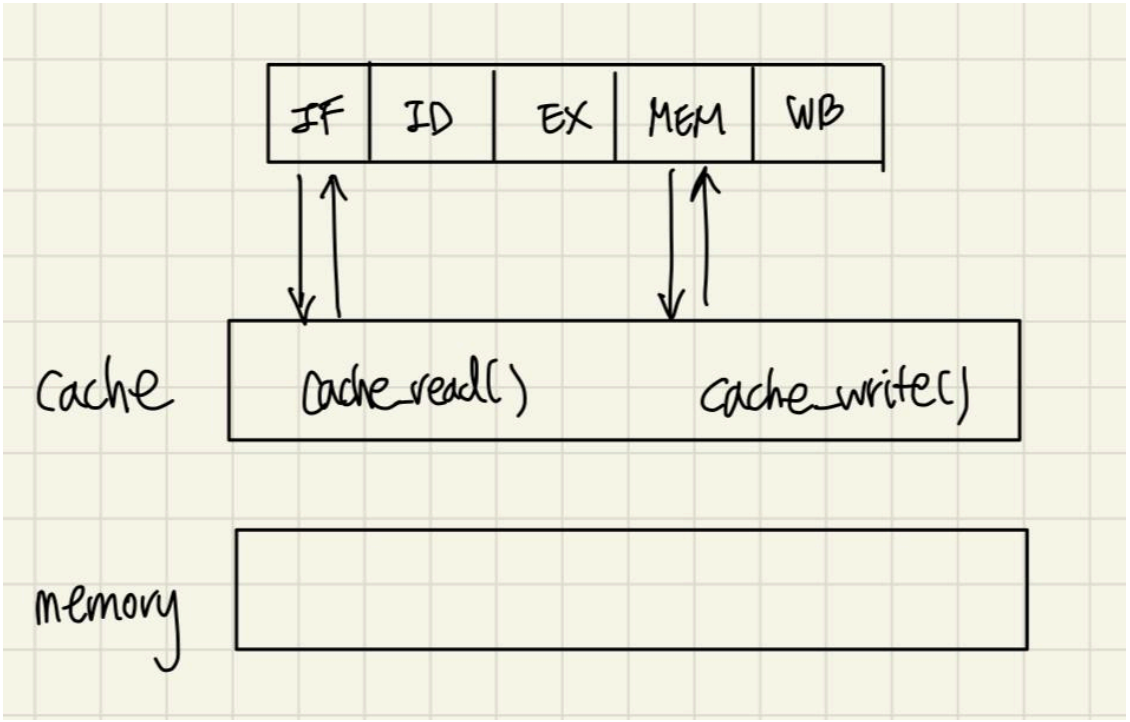
CACHE + MIPS

32214362 모바일시스템공학과 조윤서

Introduction

본 프로젝트에서는 성능 최적화를 위해 메모리 접근 시간을 단축하는 캐시를 구현하였다. 자주 사용되는 데이터에 빠르게 접근하며, 시스템의 전반적인 효율성을 향상시키는 것을 목표로 한다.

Important Concept



single-cycle과 memory 사이에 cache를 추가한다.

Implementation

- **Cache Structure:**
  명령어 메모리와 데이터 메모리의 분리에 따라, 명령어 캐시(inst_cache)와 데이터 캐시(data_cache)도 분리하였다. 명령어 캐시(inst_cache)는 CPU에 의해 실행되는 명령어들을 캐싱하며, 데이터 캐시(data_cache)는 프로그램에 의해 사용되는 데이터를 캐싱한다.

- **Cache Size**:
  - 초기에는 1MB로 설정했으나, 캐시가 너무 커서 자주 플러시하는 일이 없어 캐시의 존재 이유가 퇴색되므로 16KB로 크기를 축소하여 효율성을 높였다.
- **Scheme**
  - 직접 매핑(Direct-mapping) 방식
- **Replacement Policy**:
  - Second Chance Algorithm(SCA)
- **Write Policy**:
  - **Write-back** 정책을 사용하여 변경된 데이터는 캐시 라인이 교체될 때 메모리에 쓰인다.
  - **Write-allocate** 정책은 캐시 미스 시 메모리에서 캐시 라인을 불러오고, 그 위치에 새로운 데이터를 작성한다.

- **init_cache()**
  - 명령어 캐시(inst_cache)와 데이터 캐시(data_cache)를 초기화한다.
  - 모든 캐시 라인을 유효하지 않은 상태로 설정하고, 모든 데이터를 0으로 초기화한다.

- **fetch_from_memory()**
  - 주어진 메모리 주소에서 캐시 라인으로 데이터를 가져온다.
  - 메모리 주소를 기반으로 데이터를 읽어와 지정된 캐시 라인에 저장한다. 명령어 캐시와 데이터 캐시를 구분하여 처리할 수 있다.

- **find_victim()**
  - 새 데이터를 저장할 캐시 라인을 찾는다. (Victim Cache Line).
  - 이미 사용 중인 캐시 라인 중에서 Second Chance 알고리즘을 사용하여 교체될 라인을 선택한다.

- **read_from_cache()**

    - 주어진 주소에 해당하는 데이터를 캐시에서 읽어온다.
    - 주소를 분석하여 해당 캐시 라인과 태그를 찾고, HIT와 MISS를 구분하여 처리한다. MISS가 발생하면 fetch_from_memory()를 호출하여 캐시를 업데이트한다.

- **write_to_cache()**
    - 주어진 주소에 데이터를 캐시에 쓰는 함수이다.
    - 적절한 캐시 라인을 찾고, 필요하다면 오래된 데이터를 메모리에 플러시한 후 새 데이터를 캐시에 저장한다.

- **flush_line_to_memory()**
    - 지정된 캐시 라인의 데이터를 메모리에 쓰는 작업을 수행한다.
    - 캐시 라인의 데이터를 메모리의 해당 위치로 복사하고, 캐시 라인의 상태를 업데이트한다.

- **flush_cache()**
    - 전체 캐시를 메모리에 플러시한다.
    - 모든 캐시 라인을 확인하고, 변경된 데이터가 있다면 메모리에 저장한다.

- **cache_full()**
    - 캐시가 가득 찼는지 확인한다.
    - 모든 캐시 라인을 검사하여 유효한지 확인한다.

Build Configuration / Environment

- Development Environment: **Visual Studio Code**
- Programming Language: **C**

- Build Configuration:
    - 컴파일: **gcc main.c src/alu.c src/control_signal.c src/cpu.c src/executionStats.c src/memory.c src/opcode.c src/cache.c**
    - 실행 파일과 ‘simple.bin’ 함께 실행: **./a.out test_prog/simple.bin > a.log**
    - 실행 파일과 ‘simple2.bin’ 함께 실행: **./a.out test_prog/simple2.bin >**

a.log

- 실행 파일과 'simple3.bin' 함께 실행: **./a.out test_prog/simple3.bin > a.log**
- 실행 파일과 'simple4.bin' 함께 실행: **./a.out test_prog/simple4.bin > a.log**
- 실행 파일과 'input4.bin' 함께 실행: **./a.out test_prog/input4.bin > a.log**
- 실행 파일과 'fib.bin' 함께 실행: **./a.out test_prog/fib.bin > a.log**
- 실행 파일과 'linearSearch.bin' 함께 실행: **./a.out test_prog/linearSearch.bin > a.log**

cf) makefile을 사용하여 컴파일하는 것 일반적이나, 시간 제약으로 인해 직접 컴파일을 수행하였다.

Screen Capture / Working Proofs

```
==========================PROGRAM RESULT============================
Return value R[2] : 0
Total Cycle : 8
Executed 'R' instruction : 4
Executed 'I' instruction : 4
Executed 'J' instruction : 0
Number of Branch Taken : 0
Number of inst_cache 'HIT': 7
Number of inst_cache 'MISS': 1
Number of data_cache 'HIT': 1
Number of data_cache 'MISS': 1
Number of Memory Access: 3
====================================================================
```

<simple.bin의 실행 결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 100
Total Cycle : 10
Executed 'R' instruction : 3
Executed 'I' instruction : 7
Executed 'J' instruction : 0
Number of Branch Taken : 0
Number of inst_cache 'HIT': 9
Number of inst_cache 'MISS': 1
Number of data_cache 'HIT': 3
Number of data_cache 'MISS': 1
Number of Memory Access: 3
======================================================================
```

<simple2.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 5050
Total Cycle : 1330
Executed 'R' instruction : 409
Executed 'I' instruction : 920
Executed 'J' instruction : 1
Number of Branch Taken : 102
Number of inst_cache 'HIT': 1328
Number of inst_cache 'MISS': 2
Number of data_cache 'HIT': 612
Number of data_cache 'MISS': 1
Number of Memory Access: 4
======================================================================
```

<simple3.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 55
Total Cycle : 243
Executed 'R' instruction : 79
Executed 'I' instruction : 153
Executed 'J' instruction : 11
Number of Branch Taken : 10
Number of inst_cache 'HIT': 240
Number of inst_cache 'MISS': 3
Number of data_cache 'HIT': 93
Number of data_cache 'MISS': 7
Number of Memory Access: 17
======================================================================
```

<simple4.bin의 실행결과>

```
============================PROGRAM RESULT============================
Return value R[2] : 1
Total Cycle : 1061
Executed 'R' instruction : 359
Executed 'I' instruction : 637
Executed 'J' instruction : 65
Number of Branch Taken : 73
Number of inst_cache 'HIT': 1057
Number of inst_cache 'MISS': 4
Number of data_cache 'HIT': 467
Number of data_cache 'MISS': 19
Number of Memory Access: 42
======================================================================
```

<gcd.bin의 실행결과>

```
============================PROGRAM RESULT============================
Return value R[2] : 55
Total Cycle : 2679
Executed 'R' instruction : 818
Executed 'I' instruction : 1697
Executed 'J' instruction : 164
Number of Branch Taken : 109
Number of inst_cache 'HIT': 2675
Number of inst_cache 'MISS': 4
Number of data_cache 'HIT': 1088
Number of data_cache 'MISS': 7
Number of Memory Access: 18
======================================================================
```

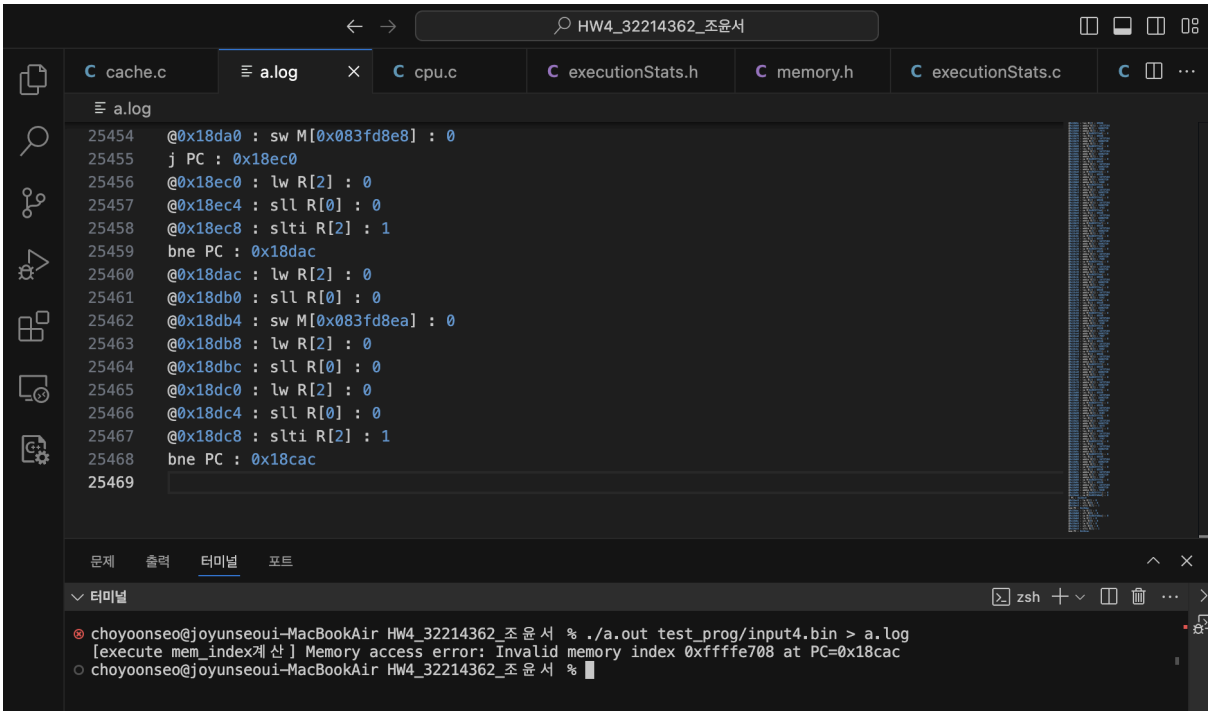
<fib.bin의 실행결과>

```
============================PROGRAM RESULT============================
Return value R[2] : 55
Total Cycle : 2896
Executed 'R' instruction : 1091
Executed 'I' instruction : 1750
Executed 'J' instruction : 55
Number of Branch Taken : 109
Number of inst_cache 'HIT': 2892
Number of inst_cache 'MISS': 4
Number of data_cache 'HIT': 1087
Number of data_cache 'MISS': 8
Number of Memory Access: 20
======================================================================
```

<fib_edited_final.bin의 실행결과**>**

```
============================PROGRAM RESULT============================
Return value R[2] : 1
Total Cycle : 65
Executed 'R' instruction : 11
Executed 'I' instruction : 54
Executed 'J' instruction : 0
Number of Branch Taken : 7
Number of inst_cache 'HIT': 61
Number of inst_cache 'MISS': 4
Number of data_cache 'HIT': 31
Number of data_cache 'MISS': 2
Number of Memory Access: 7
======================================================================
```

<linearSearch.bin의 실행결과**>**



<input4.bin의 실행결과(실패)>

Trial & Errors

Error 1: memory access오류

Trial 1:BLOCK_SIZE를 바이트로 설정해놓은 것을 까먹고 sizeof(int)로 나누었다. 선언을 할 때 바이트로 선언했는지 워드로 선언했는지 제대로 확인하는 것의 중요성을 깨달았다.

Error2: Return value가 제대로 업데이트되지 않는다.

Trial 2: 메모리에서 cache로 값을 가져올 때 indexing이 원활하게 이루어지는지 하나하나 출력하여 확인하여 오류를 잡았다.

Error3: input4 실행 제대로 안됨

Lesson

캐시를 단순히 이론적으로만 학습하는 것이 아니라 직접 구현해봄으로써 캐시의 중요성과 효율성을 깊이 있게 이해할 수 있었다. 시간부족으로 인해 이번에는 direct-mapping 방식의 캐시밖에 구현하지못했으나 향후에는 더 복잡하지만 효율성이 높은 캐싱 기법인 fully associative와 set associative 캐시를 구현해볼 것이다.