# Lab 6: Cache

20190084 권민재, 20190335 양승원 `CSED311`

# Introduction

이번 과제에서는 cache가 어떻게 작동하는지 이해하고, Lab 5의 파이프라인 CPU에 cache를 적용시키는 것이 목표이다. 앞선 파이프라인 CPU에서는 프로세서와 메모리 사이의 지연을 가정하지 않았지만, 이번 과제에서는 메모리에 지연이 있는 상황을 가정하였고, 이에 따라 cache가 적용되지 않은 baseline cpu와 2-way set associative single-level cache가 적용된 CPU를 제작하였다.

# Design

Pipeline CPU에서 추가된 메모리에 관련된 부분을 중점적으로 기술하였다.

## Memory Abstraction

이 과제에서 사용하는 메모리에는 read만 가능한 포트와 read와 write가 가능한 포트, 총 2개의 포트가 존재한다. 역할에 따라 포트를 dedicate하는 것은 잘못된 구현이기 때문에, 우리가 임의로 data r/w 과정에서 후자를 포트를 사용하는 식으로 할당하지 않는 동시에 CPU의 메모리 접근을 더 편리하게 만들기 위해서 메모리를 추상화하여 사용하도록 설계하였다.

## Cache

우리는 I-cache와 D-cache를 분리하여 각각이 2-way set associative single-level cache (4 lines, 16 words)가 될 수 있도록 설계하였으며, LRU와 write-through를 채택하였다. 이 cache는 일종의 state machine 처럼 작동하여, IF나 MEM 단계에서 주어지는 요청을 적절히 처리할 수 있도록 설계하였다.

### Indexing

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | T | T | T | T | T | T | T | T | T | T | T | T | I | O | O |

`T`: Tag, `I`: Index, `O`: Offset

cache의 한 line이 4 words (8 byte)이기 때문에, offset을 이용하여 line 안에서 각 word를 선택할 수 있도록 offset은 2비트로 구성하였다. 그리고 이 cache에는 set가 2개 존재하기 때문에, index 비트는 1비트로 구성하였으며, 남는 비트는 tag로 이용하였다.

# Data

이 cache는 기본적으로 cache에 꼭 필요한 정보인 tag, valid, data로 이루어진 table을 2 set 가지고 있으며, 각 line에는 LRU를 위한 counter가 필요하기에 각 line이 이를 포함하도록 설계하였다. 이 cache는 각 세트에 2 lines 밖에 존재하지 않기 때문에, LRU를 위해서는 최근에 접근되었는지 여부만을 기록하도록 설계하였다. 즉, 이 cache에서 LRU를 카운팅하기 위해서는 각 라인마다 1 bit만 사용한다.

# States

## Ready

이 단계는 cache가 현재 외부의 요청을 처리할 수 있음을 의미한다. 이 상태에서 읽기나 쓰기 요청이 들어왔을 때에는 각각 `Read`와 `Write` 상태로 transition한다.

## Read

이 단계는 현재 cache가 읽기 요청을 받아서 이를 처리하고 있음을 의미한다. 요청받은 주소에 대해서 hit 여부를 확인한 후, hit이라면 cache의 databank에서 데이터를 읽어서 돌려주고 `Ready` 상태로 transition한다. 이때, LRU 카운터도 업데이트한다.
만약 miss 였다면, 이제 메모리에서 값을 읽어와야 하기 때문에 memory에 요청할 주소를 세팅하고, 메모리를 기다리는 상태인 `Memory Reading` 상태로 transition 한다.

## Write

이 단계는 현재 cache가 쓰기 요청을 받아서 이를 처리하고 있음을 의미한다. 요청받은 주소가 만약 hit이었다면, hit이 된 cache의 line에 쓰기 요청 받은 데이터를 오프셋을 기반으로 조합하여 메모리에 쓰기 요청을 보낸다. 이것은 cache에 쓰기 요청이 들어온 값을 쓸 수 있기 때문에 line을 valid하게 유지할 수 있음을 뜻하며, 더 나아가 메모리에 값을 쓰는 동안 다른 읽기 요청이 들어왔을 때 캐시가 이를 처리할 수 있음을 의미한다. 그래서 메모리에 값을 쓰는 동안 평행하게 다른 읽기 요청을 처리할 수 있도록 이를 위한 state인 `Ready (Parallel)` 으로 transition한다. 만약 miss였다면, cache에 요청받은 데이터를 메모리에 쓰도록 요청하고, 이를 기다리는 state인 `Memory Writing`으로 transition 한다.

## Memory Reading

이 단계는 memory가 읽기 명령을 수행하는 동안 이를 기다리는 state이다. Memory로부터 처리가 완료되었음을 의미하는 ack 신호가 들어올 때 까지 대기하다가, ack 신호가 들어오면 cache line을 evict하고, `Ready` 단계로 transition 한다.

## Memory Writing

이 단계는 memory가 쓰기 명령을 수행하는 동안 이를 기다리는 state이다. Memory로부터 처리가 완료되었음을 의미하는 ack 신호가 들어올 때 까지 대기하다가, `Ready` 단계로 transition 한다.

## Ready (Parallel)

이 단계는 메모리가 write-hit을 처리하는 중에 다른 읽기나 쓰기 요청을 대기하는 state이다. 이 상태에서 읽기나 쓰기 요청이 들어왔을 때에는 각각 `Read (Parallel)`와 `Write (Parallel)` 상태로 transition한다.

### Read (Parallel)

이 단계는 메모리가 write-hit을 처리하는 중에 들어온 다른 읽기 요청을 처리하는 state이다. 기본적인 동작은 `Read` state와 같지만, hit가 발생했을 때 `Ready (Parallel)` 단계로 transtion하고, miss가 발생했을 때에는 메모리의 준비 여부에 따라서 `Read` 단계로 transition 한다는 점에서 차이가 있다. 이것은 write-hit을 처리 중인 busy한 메모리를 대기한 이후에 miss가 난 주소에 대해 읽기를 수행하기 위함이다.

### Write (Parallel)

이 단계는 메모리가 write-hit을 처리하는 중에 들어온 다른 쓰기 요청을 처리하는 state이다. write-hit의 처리가 완료되어 ack 신호가 들어오면 그 때 write를 처리할 수 있기 때문에, 그 때까지 대기한 이후에 `Write` 단계로 transition 한다.

# Implementation

## Memory

메모리의 지연을 구현하고 cache를 적용시킬 수 있는 형태가 될 수 있도록 메모리에 약간의 수정을 가했다. 우선, 메모리의 지연과 같은 경우에는 메모리에 timer 레지스터를 두어서 메모리의 지연을 시뮬레이션하였다. 또한, cache는 4 words의 line을 가지기 때문에 한번에 4 words를 읽고 쓸 수 있도록 이에 부합하는 포트를 신설하였으며, 처리가 끝났을 때 이를 CPU에 알릴 수 있도록 ack 포트를 신설하였다.

## Memory Abstraction

사용 목적에 따라 포트를 할당하지 않겠다는 목적을 달성하기 위해서, 메모리 추상화를 담당하는 모듈 `memory_io`에서는 각 포트가 어떤 용도로 사용되고 있는지 기록한다. 이 모듈로 읽기나 쓰기 요청이 들어왔을 때에는, 각 포트가 사용되고 있는 용도와 메모리의 busy 여부에 따라서 요청을 처리할 포트를 결정하고, 해당 포트로 주어진 요청을 보낼 수 있도록 구현하였다.

## Cache

### Indexing

16비트의 주소를 입력받으면 우선 이를 tag, index, offset으로 파싱하도록 구현하였다. 이때, cache의 Read state에서 miss가 발생할 경우에는 입력받은 주소를 별도로 저장하여 후에 다른 state에서 cache table에 접근할 때에 문제가 없도록 만들었다.

### Data

Cache의 table은 tag, valid bit, data, LRU counter로 구성하였으며, 각각은 크기가 4인 배열이 되도록 구현하였다. 이때 인덱스 0, 1은 첫번째 set, 인덱스 2, 3은 두번째 set에서 사용한다. 요청받은 주소에 대해서 table에 접근할 때에는 주소로부터 파싱한 인덱스와 그 인덱스에 2를 더한 값에 접근할 수 있도록 하여 set associative를 달성하였다.

# States

## Ready

읽기나 쓰기 요청이 들어왔을 때 다른 state로 전이해야하는데, ack 신호가 들어오지 않았을 때에만 전이하도록 구현하여 예외 상황을 줄였다.

## Read

Hit이 발생했을 경우에는 주소로 파싱한 값을 기준으로 line으로부터 word size의 데이터를 읽어서 데이터를 레지스터 형식으로 출력한다. 이때, 해당 라인의 LRU counter를 0으로 업데이트 하며, 다른 라인의 counter는 1으로 업데이트한다. Miss가 발생했을 경우에는, 메모리에 읽기 요청을 보낼 준비를 해야한다. Cache의 한 라인 사이즈가 4 words, 즉 8 byte이기 때문에 메모리에 읽을 주소를 요청할 때에는 하위 2비트를 0으로 만들어서 요청하고 4 words를 읽을 것을 메모리에 알려주도록 이 단계에서 준비한 후에 `Memory Reading` state로 전이한다.

## Write

Hit이 발생했을 경우에는 캐시에 있는 값을 포함하여 메모리에 4 words의 데이터를 쓸 수 있도록 한다. Read의 경우와 마찬가지로, 현재 주소의 하위 2비트를 0으로 바꾼 후에 메모리에 주소를 요청하며, hit가 발생한 line의 데이터에 오프셋에 따라 요청이 들어온 word를 삽입한 형태로 메모리에 요청을 보낸다. 즉, 만약 주소의 오프셋이 0이었다면, 메모리에 데이터를 보낼 때에는 line의 가장 하위 word가 cache에 쓰기가 요청된 데이터로 변경된 형태로 메모리에 쓰기를 요청한다. 이때, 해당 cache line은 valid bit를 0으로 설정하여 evict 시킨다. Miss가 발생했을 경우에는, cache가 요청 받은 값을 그대로 memory에 적어야 하기 때문에, 앞서 Design 파트에서 언급한 것과 같이 요청 받은 값을 memory에 word size를 적을 것이라는 사실과 함께 전달한다.

## Memory Reading

메모리에서 ack 신호가 왔을 때, cache를 업데이트한다. 이때, 메모리로부터 도착하는 정보는 4 words이며, 해당 데이터로 cache line을 replace 하고 LRU counter를 업데이트 하는 방식으로 구현하였다.

## Memory Writing

메모리에서 ack 신호가 왔을 때, 바로 Ready state로 transition 하도록 구현하였다.

## Ready (Parallel)

읽기 신호가 들어왔을 때에는 `Read (Parallel)`로 전이하고, 쓰기 신호가 들어왔을 때에는 `Write (Parallel)`로 전이한다.

## Read (Parallel)

Hit가 발생하였을 경우에는, `Read`와 똑같이 동작하여 프로세서에 그 결과를 전달한다. 이 state의 이전에 있었을 `Write` state에서 write-hit을 처리해야하는 주소가 포함된 cache line의 valid bit를 0으로 설정하는 식으로 evict 하였기 때문에, 해당 케이스는 여기서 고려하지 않아도 된다. Miss가 발생하였을 때에는, 메모리에서 실제 값을 읽어와야하지만 메모리는 write-hit을 처리하느라 busy한 상태이므로 ack 신호가 올 때 까지 대기한다.

### Write (Parallel)

Ack 신호가 올 때 까지 대기한다.

# Discussion

- 이 구현에서는 Write Buffer를 사용하지 않았으나, write-hit을 state에 기반해서 처리하지 않고 Write Buffer를 이용하여 처리하면 캐시의 동작이 block되는 경우가 조금 줄어들 수 있다.
- 메모리의 포트를 instruction과 data 접근에 대해서 dedicate하지 않도록 메모리를 추상화한 점에서 확장성이 높다.
- Write-Through 방식으로 캐시를 구현하여, 동일한 주소에 write하더라도 메모리에 계속해서 write를 수행하게 되었다. 하지만 write hit가 발생하는 경우에는 그래도 cache만 이용하고 memory 접근이 필요하지 않은 동작들은 처리할 수 있기에 단점이 상쇄된 것 같다.
- write through에서 partial write가 필요했기에 메모리에서 QWORD size input과 WORD size input을 모두 받을 수 있도록 구현하였는데, write back을 이용해 구현하면 memory에서 QWORD size input만 받을 수 있으면 되기 때문에 메모리가 간단해지는 장점이 있었을 것 같다.

# Conclusion

- Without Cache (Full)

```
# Clock # 2657
# The testbench is finished. Summarizing...
# All Pass!
```

- With Cache (Full)

```
# Clock #  895
# The testbench is finished. Summarizing...
# Test # 19-3 :     Wrong
# Test #   20 : No Result
# Pass : 54/56
```

- Without Cache (Until 19-2)

```
# Clock #  281
# The testbench is finished. Summarizing...
# Test # 19-3 : No Result
# Test #   20 : No Result
# Pass : 54/56
```

- With Cache (Until 19-2)

```
# Clock #   391
# The testbench is finished. Summarizing...
# Test # 19-3 : No Result
# Test #   20 : No Result
# Pass : 54/56
```

- Cache Hit Rate (Until 19-2)

```
# hit:    486, miss:     59
# hit:     18, miss:     30
```

I-Cache 에서 486 hit, 59 miss가 발생하고, D-Cache 에서 18 hit와 30 miss가 발생하였다.
총 504 hit와 89 miss가 발생하였으므로

$$\text{hit ratio} = \frac{504}{504+89} \times 100 \simeq 85.00(\%)$$

테스트 19-2까지에 대해 baseline cpu보다 오히려 클럭 사이클 수가 늘어난 것을 확인할 수 있는데, 이것은 지역성을 잘 활용하는 가장 마지막 테스트를 진행하지 못했기 때문이라고 생각된다. 이번 과제를 통해 캐시로 메모리 입출력 과정을 최적화할 수 있음을 알게되었다.