Lab 7: Cache

CSE 4190.308 Computer Architecture
3 Exercises (Total 90 Points)

Received: May 28, 2014
Due: 11:59 p.m., June 13, 2014

TA Office Hours: 7:00 - 8:00 p.m., 6/11 - 6/13

1 Introduction

In this lab, you will implement caches between the 5-stage pipelined processor and the main memory. A 5-staged pipelined implementation with a skeleton code for the caches will be provided. You will notice that when you run the provided code without any modifications, it will take a very long time to finish. This is because the simulated main memory that is accessed in case of cache misses is made slower, in order to make it a bit more realistic.

2 Getting Started

2.1 How to Download the Source Code

Download the `add-lab7.sh` script from the course web page and run it with your ID.

```
$ ./add-lab7.sh YOUR_ID
```

2.2 Directory Structure of Lab 7

Lab 7 실습의 디렉터리 구조는 다음과 같습니다.

```
lab5/
    src/
        LocalCache.bsvv
        CacheTypes.bsv
        Proc.bsv
        y86
    lib/
        ...
```

`src/`
: Lab 7 실습을 진행할 디렉터리입니다.

`src/LocalCache.bsv`
: Cache 모듈의 뼈대 코드입니다. 이 파일을 수정하여 실습을 진행합니다.

`src/CacheTypes.bsv`
: Cache 모듈에서 사용할 파라미터를 정의한 파일입니다.

`src/Proc.bsv`
: 5-stage pipeline y86 프로세서이며 메모리에 직접 접근하는 대신 cache를 사용하도록 구현해야 합니다.

`src/y86`
: 구현한 프로세서를 컴파일하고 벤치마크 프로그램을 실행할 수 있는 스크립트 파일입니다.

`lib/`
: Lab 에서 구현한 y86 프로세서를 동작시키는 데 필요한 라이브러리 및 프로그램을 포함하고 있습니다.

2.3 How to Simulate the Design

Compiling and running the code can be done identically to the previous labs, using the `y86` run script.

2.4 How to Submit Your Design

Changed files will be uploaded to svn server when you type `svn commit` command in your `lab7/src` directory.

3 A 5-stage Pipeline with Caches

제공되는 프로세서 모듈은 메모리에 접근 시, cache를 호출하지 않고 직접 접근하도록 구현되어 있습니다. 이번 실습에서는 cache를 사용하도록 수정하고, 정상적으로 동작하는 cache를 구현해야 합니다. 먼저 direct-mapped cache를 구현하고 이를 바탕으로 cache의 block size와 associativity 를 늘리는 실습을 진행합니다.

3.1 Communication Between Cache and Memory Modules

제공되는 cache와 메모리 모듈의 인터페이스는 다음과 같습니다.

```
interface Cache;
        method Action req(MemReq r);
        method ActionValue#(Data) resp;

          method ActionValue#(CacheMemReq) memReq;
          method Action memResp(Line r);

        method Data getMissCnt;
        method Data getTotalReq;
endinterface

interface Memory;
        method Action iReq(CacheMemReq r);
        method ActionValue#(MemResp) iResp;
        method Action dReq(CacheMemReq r);
        method ActionValue#(MemResp) dResp;
endinterface
```

`Cache`는 각각 `req`와 `resp` method를 통해 프로세서로부터 요청를 받고, 그 요청에 대한 답변을 보내게 됩니다. 요청에서 인자로 받는 `MemReq` 구조체는 `lib/common-lib/MemTypes.bsv`에 정의 되어 있으며, request의 종류 (Store인지, Load인지), 메모리 주소, 그리고 store 시에 사용할 데이터가 포함됩니다. `Memory` 역시 `Cache`와 유사한 request-response 방식의 인터페이스로 구성되어 있습니다. Cache에서

요청받은 작업을 처리하다보면 메모리로의 접근이 필요할 것입니다. 이 때는 `memReq` method를 통해 `CacheMemReq`를 메모리 모듈에 전달하고, 전달한 request가 load인 경우 `memResp` method를 통해 요청한 데이터를 받아오면 됩니다.

이번 lab에서 사용할 메모리는 cache에서 한번에 다루는 데이터 크기(*Cache Block*)가 Load/Store instruction에서 다루는 데이터 크기(*Word*)보다 큰 경우를 고려하여 burst access를 지원합니다. 실제의 메모리 동작과는 차이가 있지만, 사용할 메모리는 request에 한번에 읽어올 word의 개수를 지정할 수 있습니다. `lib/common-lib/MemTypes.bsv`에 정의된 `CacheMemReq` 구조체의 `burstLength`가 그것입니다. 메모리는 `burstLength`에 따라 순서대로 `Vector#(n, Data)`의 구조로 데이터를 전해주게 됩니다.

cache와 메모리는 프로세서 모듈인 `mkProc`에서 선언되어 초기화 됩니다. cache와 메모리가 서로 같은 레벨에 존재하기 때문에 cache가 메모리의 interface를 부를 수 없고, 메모리가 cache의 interface를 사용할 수 없습니다. 이렇게 같은 레벨의 모듈을 서로 연결할 때, Bluespec에서는 `mkConnection`이란 모듈을 사용합니다.

현재 cache와 메모리의 작동 메커니즘은 아래와 같습니다 (dCache의 경우).

1. CPU가 `Cache` 인터페이스의 `req` method를 호출

2. dCache가 요청받은 내용을 `CacheMemReq`로 변환하고 이 request data를 `memReqQ`에 enqueue

3. dCache의 `memReq` method를 통해 `memReqQ`에서 request data가 빠져나감

4. Memory의 `dReq` method를 통해 위의 request data가 메모리 모듈의 `dMemReqQ`로 enqueue

5. Memory가 요청을 처리 후, Load 요청인 경우 `dMemRespQ`에 response data를 enqueue

6. Memory의 `dResp` method를 통해 `dMemRespQ`에서 response data가 빠져나감

7. dCache의 `memResp` method를 통해 `memRespQ`로 response data가 enqueue

8. dCache가 `memRespQ`에서 response data를 dequeue하여 얻어냄

문제는 dCache와 Memory는 서로 interface를 호출할 수 없기 때문에, 3, 4, 6, 7은 상위 모듈인 `mkProc`에서 다음과 같이 처리해주어야 합니다.

```
rule connectReq;
        let req <- dCache.memReq;
        mem.dReq(req);
endrule

rule connectResp;
        let resp <- mem.dResp;
        dCache.memResp(resp);
endrule
```

이처럼 단순히 `ActionValue` method와 `Action` method를 잇는 rule을 일일이 만들지 않도록

```
mkConnection(mem.dReq, dCache.memReq);
mkConnection(mem.dResp, dCache.memResp);
```

위와 같은 코드를 이용하면 `mkConnection`모듈이 위 rule의 동작을 자동으로 수행하게 됩니다. 이러한 연결은 `Proc.bsv`에 주석처리(369, 370라인) 되어 있으므로, 실습 시 주석을 해제하시고, 여러분들은

cache에서 메모리에 request를 보낼때는 `memReqQ`에 request data를 enqueue하고 메모리로 부터 response를 받는 것은 `memRespQ`에 데이터가 enqueue되기를 기다리면 됩니다.

예를 들어, 메모리로 부터 `0xF0` 번째 부터 4개의 word를 읽어오고 싶다면

```
memReqQ.enq(CacheMemReq{op: Ld, addr: 0xF0, data: ?, burstLength: 4});
```

위와 같은 코드를 어떤 rule에서 실행 한 후 다른 rule에서는

```
let resp = memRespQ.first;
```

위와 같은 코드로 response를 기다리면 됩니다. 그리고 return 값인 `resp`에는 앞에서 부터 순서대로 `0xF0, 0xF4, 0xF8, 0xFC`의 데이터가 저장되어 있습니다.

3.2 Implementing Direct-mapped Cache Memory

강의자료를 참고하여 block size 가 1 word (`Data`) 인 가장 간단한 형태의 direct-mapped cache 를 구현합니다.

cache access 와 miss 수는 각 cache 모듈의 상단에 선언된 `reqCnt`, `missCnt` 레지스터를 사용하여 count 할 수 있습니다. 이 값은 벤치마크 프로그램 수행 종료 시에 화면에 출력됩니다.

Exercise 1 (30 points) : `Proc.bsv`, `LocalCache.bsv` 파일을 수정하여 `mkCacheDirectMap` 모듈이 single-word direct-mapped cache로 동작하도록 완성하시오.

정확히 구현한 경우 benchmark 시뮬레이션 시 아래표와 유사한 결과를 얻을 수 있습니다.

Benchmark	Clock Cycles	Insts	Cache Req Count	Cache Miss Count	Cache Miss Rate
asum	6727	341	57	57	100%
Array32	4246	238	36	36	100%
Array_ij	34927	1986	292	292	100%
Array_ji	38574	2115	352	311	88.4%
bubble	101027	13843	3650	452	12.4%
fibonacci	67235	9881	5115	294	5.7%
htower	88908	2732	1271	641	50.4%

Table 1: Benchmark Result with single-word block, Direct-mapped Cache

3.3 Increasing Block Size

Single-word block cache 에서는 메모리 Instruction 요청한 데이터만 cache로 읽어와 저장하는데, cache 의 block size 를 늘리면 instruction 에서 요청하는 데이터보다 많은 데이터를 읽어와 cache 에 저장함으로써 성능향상을 기대할 수 있습니다. 이는 대부분의 프로그램이 메모리 접근 패턴에서 spatial locality 특성을 보이기 때문입니다. 메모리 주소상으로 인접해있는 데이터를 접근할 확률이 높고 이 데이터를 미리 한꺼번에 읽어옴으로써 cache hit rate 을 증가시킬 수 있습니다.

Exercise 2 (30 points) : **Exercise 1** 에서 구현했던 single-word direct-mapped cache 를 수정하여 multiple-word block size 를 지원하는 direct-mapped cache 를 구현하시오 (단, cache 의 전체 크기는 변경하지 않음). block size 는 다음과 같이 정의된 `typedef` 를 통해 자유롭게 변경할 수 있도록 구현해야 합니다. Cache 모듈과 관련된 정의들은 `CacheTypes.bsv` 파일에 있습니다.

```
typedef WordsPerBlock 4
```

다음은 block size를 4-word로 설정하고 benchmark 프로그램을 실행시킨 결과입니다.

Benchmark	Clock Cycles	Insts	Cache Req Count	Cache Miss Count	Cache Miss Rate
asum	2809	341	57	20	35.1%
Array32	4237	238	36	35	97.2%
Array_ij	12354	1986	292	79	27.1%
Array_ji	25893	2115	352	188	53.4%
bubble	33137	13843	3650	23	0.6%
fibonacci	30048	9881	5115	75	1.5%
htower	14205	2732	1271	94	7.4%

Table 2: Benchmark Result with 4-word block

3.4 Increasing Associativity

cache 의 associativity 는 특정 메모리 주소값의 데이터가 cache 에 들어갈 때 어느 위치에 저장될 수 있는지, 그 제한된 공간의 수를 의미합니다. associativity 가 1인 direct-mapped cache 는 임의의 주소값의 데이터가 cache 내에서 한 곳에만 위치할 수 있습니다. 다시 말해 associativity 가 n 인 cache 는, 임의의 주소값의 데이터가 특정 n 개의 cache entry 중 하나에 선택적으로 위치할 수 있는 구조입니다.

Set associative cache 를 구현하려면, set 내에 더이상 공간이 없는데 새 데이터를 저장해야 할 때 cache 에서 쫓아낼 victim 을 선정하는 알고리즘을 결정해야 합니다. 가장 먼 과거에 접근되었던 entry 를 쫓아내는 LRU 방식을 생각할 수 있겠으나, 정확한 LRU 를 구현하려면 추가적으로 유지해야 하는 정보의 양이 많고 로직의 복잡도 또한 크게 증가합니다. 따라서 이번 실습에서는 Exact LRU 가 아닌 Approximate LRU 를 적용하여, 가장 최근에 접근했던 entry 를 제외하고 victim 을 선정하는 정책을 취합니다. 2-way set associative cache 의 경우는 이 정책이 Exact LRU 와 동일하며 associativity 가 더 큰 경우에도 다시 접근될 확률이 가장 높은 entry 를 cache 에서 쫓아내는 상황을 방지할 수 있습니다.

Exercise 3 (30 points) : 위 **Exercise 2**에서 구현한 multiple-word direct-mapped cache 를 바탕으로, `LocalCache.bsv` 파일 하단에 선언되어 있는 `mkCacheSetAssociative` 모듈에 n-way set associative cache 를 구현하시오. `mkCacheSetAssociative` 모듈은 multiple-word block size 와 multi-way 를 모두 지원해야 합니다. 여기서 way의 수는 block size 와 마찬가지로, `CacheTypes.bsv`에 선언된 `CacheSets` 를 통해 자유롭게 변경이 가능하도록 구현해야 합니다.

예를 들어, block size가 4-word인 2-way set associative cache는 다음과 같은 결과를 보입니다.

Benchmark	Clock Cycles	Insts	Cache Req Count	Cache Miss Count	Cache Miss Rate
asum	2809	341	57	20	35.1%
Array32	4237	238	36	35	97.2%
Array_ij	12354	1986	292	79	27.1%
Array_ji	13953	2115	352	91	25.9%
bubble	33137	13843	3650	23	0.6%
fibonacci	30048	9881	5115	75	1.5%
htower	14205	2732	1271	94	7.4%

Table 3: Benchmark Result with 4-word block, 2-way Set Associative Cache