컴퓨터구조 (CSED311)

Lab 5a Report

20200220 오상윤

(1) Introduction

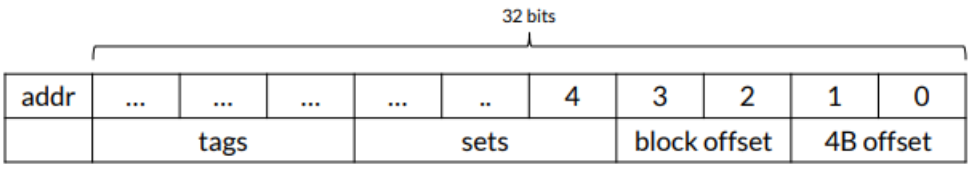
지난 Lab4-2a에서 구현한 5 stage pipelined cpu에서 Data Memory module 자리를 Cache module로 대체하였다. cpu로부터 Memory로의 request가 발생하면 한 cycle 내의 data를 읽고 쓰던 기존의 magic memory와 달리, Cache는 request로 받은 address의 data가 Cache에 존재하는지 여부에 따라 각기 다른 cycle을 소모하여 data를 읽고 쓴다.

Cache size는 256Bytes이며, 각 Cache Line size는 16Bytes로, 총 16개의 Line을 가지도록 하였다. Direct-mapped로 구현하였으며, Write-Miss policy와 Write-Hit policy는 Write-allocate와 Write-back으로 채택하였다. 구현 이후에는 제공된 Naïve Matrix multiplication 파일과 Tiled Matrix multiplication 파일을 각각 사용하여 구현한 cache의 hit ratio를 확인해보고 두 결과를 비교하였다.

파일 구성은 cpu module을 담고 있는 cpu.v, PC module, Adder module을 담고 있는 pc.v, ControlUnit module을 담고 있는 ControlUnit.v, ImmediateGenerator module을 담고 있는 ImmediateGenerator.v, ALUControlUnit module, ALU module을 담고 있는 ALU.v, Mux module, MuxForIsEcall module, Mux2bit module을 담고 있는 mux.v, HazardDetectionUnit module을 담고 있는 HazardDetectionUnit.v, ForwardingUnit module, ForwardingMuxControlUnit module을 담고 있는 ForwardingUnit.v, constant들의 define을 담고 있는 opcodes.v, BTB module, MissPredictionDetector module을 담고 있는 BTB.v, Cache module, CacheDataBank module, CacheTagBank module을 담고 있는 Cache.v, CLOG2의 정의를 담고 있는 CLOG2.v 로 구성되어 있다.

(2) Design

Textbook에 있는 Finite-State Machine for a Simple Cache Controller 디자인을 참고하여 Cache를 설계하였다.



Cache는 cpu로부터 전달받은 address에서 [3:2] 비트는 block offset으로, [7:4] 비트는 index로, [31:8] 비트는 tag로 사용하도록 하였다.

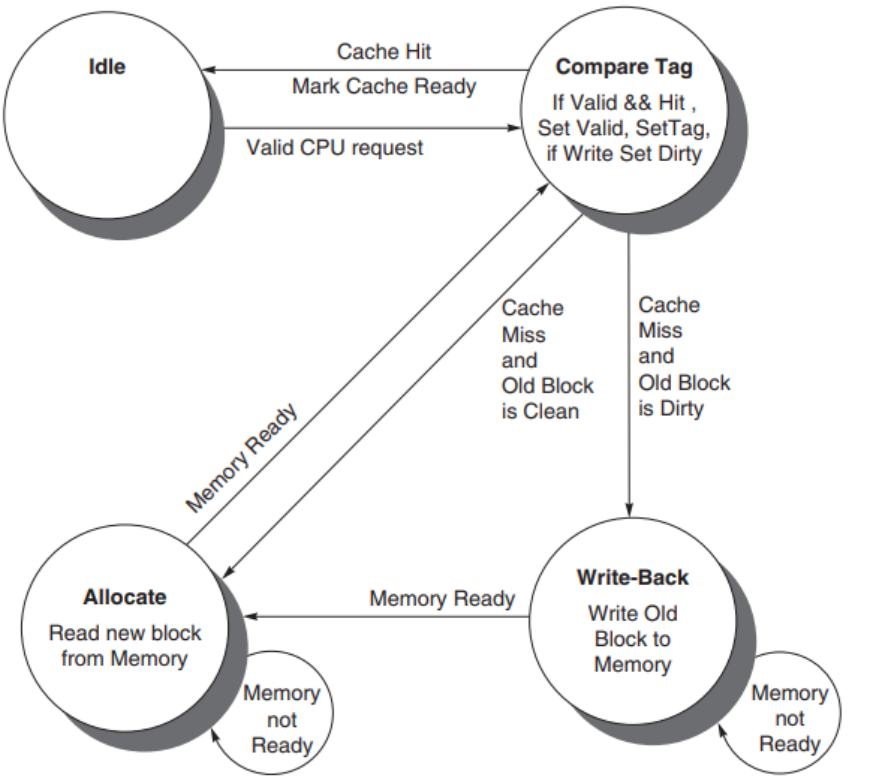


FIGURE 5.39 Four states of the simple controller.

Cache Control을 위하여 Idle state, Compare Tag state, Allocate state, Write-Back state의 총 4가지 state를 가지도록 하였다.

먼저 Idle state는 cpu로부터 유효한 request가 들어올 때까지 대기하고 있을 때의 state이다. State가 Idle state일 때 cpu는 Cache의 output 값을 사용할 수 있다. Idle state에서 cpu의 request가 들어오면, Compare Tag state로 이동한다.

Compare Tag state에서 Cache module은 cpu로부터 받은 addr에서 index와 tag를 추출한 다음 Tag Bank에 접근하여 해당 index의 tag와 추출한 tag를 비교하고, valid bit를 체크하여 Cache Hit/Miss 여부를 결정한다. 이때 Cache Hit인 경우, read request일 경우 Cache의 Data Bank에서 요청된 값을 꺼내 output으로 내보내고 Idle state로 돌아가며, write request일 경우 Cache의 Data Bank에 din을 통해 들어온 값을 기록하고 dirty bit를 1로 설정한다. 반면 Cache Miss인 경우, dirty bit을 체크하여 clean한 Cache line인 경우(dirty bit이 0) Allocate state로, dirty한 Cache line인 경우(dirty bit이 1) Write-Back state로 이동한다.

Allocate state의 경우 Data Memory에 request로 받은 address의 data를 요청하고, memory가 ready될 때까지 대기하다가 해당 data를 전달받으면 Cache line에 tag와 함께

새롭게 기록한다. 이때 valid bit는 1, dirty bit는 0으로 설정한다. 이후 Compare Tag state로 돌아간다.

Write-Back state의 경우 기존 Cache line에 기록되어 있던 data를 Data Memory에 쓰도록 요청하고, memory가 ready될 때까지 대기하다가 memory가 ready 되면 Allocate state로 이동한다.

Cache의 구현을 위해 설계한 Cache module, CacheDataBank module, CacheTagBank module들에 대한 설명은 아래와 같다.

1. Cache module

   Cpu로부터 address를 받아 tag와 index를 추출하고, input값들과 cache hit 여부 등에 따라 state를 변경해가며 state에 따라 CacheDataBank module, CacheTagBank module, DataMemory module에 적절한 request를 보내서 Cache의 역할을 수행한다. cpu에게 Cache가 Memory로부터 data를 가져오느라 바쁜지, Cache의 output이 유효한지, Cache hit이 떴는지 등을 signal로 보내주며, cpu는 이를 받아 cache에 request를 보냈는데 유효한 respond를 받지 못한 경우 cpu 전체를 stall 한다.

2. CacheDataBank module

   Cache에 저장된 data들을 담고 있는 data bank를 관리한다. Index를 받아 data bank 내의 해당 index 값을 cache module로 보내주며, write enable signal이 1인 경우 해당 index의 data bank line에 새로운 값을 기록한다.

3. CacheTagBank module

   Cache 내의 tag, valid bit, dirty bit들을 관리한다. CacheDataBank와 비슷하게 index를 받아 해당 index의 tag, valid bit, dirty bit을 cache module로 보내주며, write enable signal이 1인 경우 해당 index의 tag, valid bit, dirty bit에 새로운 값을 기록한다.

(3) Implementation

1. Cache module

   input으로 reset, clk, is_input_valid, addr, mem_read, mem_write, din을 가지며, output으로 is_ready, is_output_valid, dout, is_hit을 갖는다. 현재 state와 다음 state를 저장하기 위한 register로 cur_state와 next_state를 가지며, 매 posedge clk 마다

synchronous하게 cur_state에 next_state를 넣어준다. Addr로부터 얻은 block offset 값에 따라 cache에 새롭게 쓰게될 data와 output dout으로 내보낼 data가 asynchronous하게 결정되며, cur_state에 따라 각 state에서 수행해야 할 동작들을 적절하게 수행한다. Cur_state와 input 값에 따라 next_state가 결정된다. Data memory module이 준비되었는지 여부에 따라 is_ready bit을 asynchronous하게 설정하며, cur_state가 Idle state일 때만 asynchronous하게 is_output_valid bit을 1로 설정한다. Addr로부터 얻은 tag와 Tag Bank로부터 읽은 tag가 일치하고, 해당 index의 valid bit가 1일 때만 asynchrounous하게 is_hit을 1로 설정한다. 또한 CacheDataBank, CacheTagBank, DataMemory로 보내는 signal data_we, tag_we, tag_to_write, valid_write, dirty_write, dmem_input_valid, dmem_read, dmem_write 값을 적절하게 조정하여 해당 module들이 적절한 동작을 수행하도록 만든다.

2. CacheDataBank module

   input으로 reset, clk, index, write_enable, data_to_write를 가지며, output으로 data_to_read를 갖는다. Cache에 저장되는 data들을 저장하기 위한 128 bit의 16 line data_bank register를 관리한다. Index에 해당하는 data_bank의 값을 data_to_read로 asynchronous하게 설정한다. 또한 write_enable이 1인 경우 clock synchronous하게 index에 해당하는 data_bank에 data_to_write으로 들어온 data를 저장한다.

3. CacheTagBank module

   input으로 reset, clk, index, write_enable, tag_to_write, valid_write, dirty_write를 가지며, output으로 tag_to_read, valid_read, dirty_read를 갖는다. Cache 각 index의 tag, valid bit, dirty bit을 저장하는 tag_bank, valid_table, dirty_table register들을 관리한다. Index에 해당하는 tag_bank, valid_table, dirty_table의 값을 각각 tag_toread, valid_read, dirty_read로 asynchronous하게 설정한다. 또한 write_enable이 1인 경우 clock synchronous하게 index에 해당하는 tag_bank, valid_table, dirty_table에 tag_to_write, valid_write, dirty_write으로 들어온 값들을 저장한다.

(4) Discussion

구현하고 디버깅을 하던 중, register 들의 결과 값이 계속 터무니없는 값이 등장하였다. Cache 의 문제가 아닌 input 으로 들어오는 address 에 문제가 있는 것 같아 다른 module 들을 살펴본 결과, DataForwarding 을 위한 ForwardingMuxControlUnit 이 잘못된 mux signal 을 발생시켜 잘못된 타이밍에 DataForwarding 이 발생하는 것을 확인할 수 있었다. ForwardingMuxControlUnit 이 rs1, rs2, rd 값을 비교한 결과만을 바탕으로 mux

signal 을 발생시키도록 설계한 것이 문제였다. 따라서 mem_wb_reg_write 값도 함께 참고하여 rd 값이 유효한 경우에만 비교한 결과를 바탕으로 mux signal 을 발생시키도록 수정하였다. 이번 랩이 이전에 구현한 cpu 위에 추가적으로 구현하는 랩인 만큼 이전의 작은 실수가 나중에 발견하기 어려운 치명적인 문제가 될 수 있다는 점을 느끼게 되었다.

주어진 naïve_matmul 파일과 opt_matmul 파일을 구현한 cpu 를 통해 실행시켜보았다. Naïve_matmul 의 경우 총 cycle 수는 71,567 cycles 이며, Cache hit 횟수는 1687 회, Cache miss 횟수는 812 회가 나왔다. Opt_matmul 의 경우 총 cycle 수는 76,800 cycles 이며, Cache hit 횟수는 1605 회, Cache miss 횟수는 894 회가 나왔다. Hit ratio 를 계산했을 때, naïve_matmul 은 0.68, opt_matmul 은 0.64 로 계산된다. 두 hit ratio 의 차이가 나타나는 이유는 각 address 에 대한 접근순서가 달라 각각 Cache 사용에 있어서 양상을 보이기 때문이다. Opt_matmul 의 hit ratio 가 더 높을 것으로 예상하였으나, 예상과는 달리 naïve_matmul 의 hit ratio 가 더 높게 측정되었다. 이것의 원인은 다음과 같을 것으로 보인다. 우리의 cache 는 한 cache line 이 16 bytes, 4 개의 int 형(4bytes) 자료를 담을 수 있으며 cache 가 총 16 개의 line 을 가지므로 결국 총 64 개의 int 형 자료를 담을 수 있다. 이때 matrix a, b, c 의 크기가 8x8 로 64 개의 entry 를 가지기 때문에 각 matrix 의 같은 i,j 의 entry 끼리 동일한 cache line 을 공유하기 때문인 것으로 생각된다. 결국 반복문 내에서 매번 a, b, c 3 개의 matrix 에 모두 접근하기 때문에 tiling 을 이용한 opt_matmul 에서 더 많은 cache miss 가 발생한 것이라고 추측된다.

동일한 cache size 와 cache line size 를 유지할 때, # of ways 를 늘리기 위해서는 # of sets 를 줄여야 한다. Opt_mamul 의 tiled implementation 이 그 장점을 살리기 위해서는 ways 를 늘려 cache miss 를 줄일 필요가 있다. 현재 3 개의 matrix 에 모두 접근하는데 이때 각각의 matrix 가 동일한 cache line 을 공유하는 것이 원인이 되어 cache miss 가 지속적으로 발생하는 것으로 보이므로, 4-way associative cache 로 구현한다면 cache miss 를 줄일 수 있을 것이라고 생각된다.

(5) Conclusion

이번 Lab 을 통해 Cache 를 직접 구현해보면서 Cache 내부에서의 데이터 처리 방식, 특히 Write-allocate 와 Write-back 의 Write-Hit/Miss policy 에 대한 이해를 높일 수 있었다. 또한 주어진 naïve_matmul 파일과 opt_matmul 파일을 사용하여 직접 cache 를 구현한 cpu 에서 돌려보며 그 성능을 비교해보고 성능 차이의 원인을 분석해보면서 실제 캐시의 활용 측면에서 캐시가 어떻게 이용되며 어떤 조건에서 성능을 최대로 향상 시키는지 생각해볼 수 있었다.