

2024학년도 컴퓨터구조 Lab Assignment #5

Cache

김도영, 선민수

2024년 5월 30일

1 Introduction

이 과제에서는 Lab Assignment #4에서 구현했던 파이프라인 CPU에 더해, 주 메모리에 저장된 값을 임시로 저장하는 캐시를 구현하였다. Lab Assignment #4까지의 CPU는 질의를 받으면 한 사이클 내에 해당 메모리의 값을 반환하는, 일종의 '이상화된 메모리'(Magic memory)를 가정하고 구현되었다. 이번 과제에서는 이러한 이상적 메모리 모델에서 벗어나, 지연 시간(Latency) 개념을 도입하였다. 즉, 이번 과제에서는 메모리 접근에 지연 시간이 존재하여 메모리에서 값을 읽어오기까지 CPU의 파이프라인이 멈춰야 하며, 이는 곧 CPU의 성능 하락으로 이어진다. 이러한 메모리 접근 지연시간을 해결하기 위해 캐시를 도입하고, 다양한 캐시의 구현에 따른 CPU의 성능 변화를 알아보는 것이 이번 과제의 골자이다.

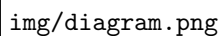
캐시는 메모리 계층구조(Memory hierarchy) 내에서 주 메모리 위, CPU 레지스터 혹은 상위 계층 캐시 아래에 위치한다. 주 메모리는 용량이 크지만 느리며, CPU 레지스터는 빠르지만 개수 및 용량이 제한되어 있다. 캐시는 이 둘 사이에 위치하여 최근에 접근한 값들 또는 이들에 인접한 값들을 미리 저장해두고, CPU의 질의가 있을 때마다 주 메모리 대신 저장한 값을 전달한다. 캐시는 CPU 레지스터보다는 느리지만 메모리에 비해서는 월등히 빠르기 때문에, 프로그래머로 하여금 컴퓨터가 용량이 큰 동시에 빠른, 이상적 주 메모리를 가지고 있는 것 같은 추상화를 제공한다.

캐시는 그 구현에 따라 직접 사상(Direct-Mapped), 집합 연관 사상(Set-Associative), 완전 연관 사상(Fully-Associative) 캐시로 나눌 수 있다. 직접 사상 캐시는 메모리의 여러 주소가 캐시의 특정 라인에 직접 대응되는 방식의 캐시이다. 이러한 캐시는 구현이 간단하고, 읽기 및 쓰기에 별다른 연산을 요하지 않아 속도가 빠르지만, 여러 메모리 주소가 캐시의 한 라인에 대응되어 발생하는 충돌(Conflict)의 발생률이 높아 캐시 적중률이 낮아지는 단점이 있다. 집합 연관 사상 캐시는 여러 캐시 라인을 하나의 집합으로 묶어, 특정 메모리 주소에 대한 질의가 들어왔을 때 그 주소에 해당하는 집합 내 어느 라인에도 데이터를 저장할 수 있도록 만든 캐시이다. 집합 연관 사상 방식은 읽기 및 쓰기 과정에서 집합 내의 각 Way를 탐색해야 하므로 직접 사상 방식에 비해 느리지만, 공간 활용성이 좋아 충돌 발생률이 낮고 캐시 적중률이 높다는 특징이 있다. 집합 연관 사상 방식에서 더 나아가 집합의 개수를 라인의 개수와 같게 만들어, 쓰기 질의가 들어왔을 때 캐시 내의 아무 라인에나 데이터를 저장할 수 있게 만든 방식의 캐시를 완전 연관 사상 캐시라 한다. 이번 과제에서는 직접 사상 및 집합 연관 사상 방식의 캐시를 구현하였다.

2 Design

캐시는 단순히 데이터를 저장하고 출력하는 것 뿐만 아니라, CPU의 질의에 응답하여 요청 블록이 현재 캐시에 존재하는지 확인하고, 만약 요청된 블록이 현재 캐시에 없다면 메모리에 해당 블록이

포함된 라인을 질의하여 메모리가 이에 응답하기까지 기다리는 등, 현재 상태에 따라 다양한 동작을 수행한다. 따라서, 캐시는 내부 상태를 가지고 입력에 따라 상태가 변화하는 유한 상태 기계(Finite State Machine)으로 구현되며, 이번 과제에서 구현된 캐시는 3개의 상태를 가진 유한 상태 기계로 구현되었다.



img/diagram.png

캐시 제어를 위한 유한 상태 기계의 설계

IDLE 상태는 CPU의 요청이 있기 전, 혹은 CPU의 요청을 모두 수행한 후의 유후 상태를 나타낸다. 캐시는 기본적으로 **reset** 신호가 1일 때, 이 상태로 초기화된다. IDLE 상태에서, CPU가 요청을 전달하고, 캐시에 요청 대상 블록이 존재하지 않아 메모리에서 해당 블록을 가져와야 하는 경우 다음 상태로 전이한다. 만약 요청된 주소에 해당하는 라인이 메모리에서 마지막으로 가져온 이후 수정되었을(Dirty) 경우 **WRITE_WAIT** 상태로, 수정되지 않았을 경우 **READ_WAIT** 상태로 전이한다.

WRITE_WAIT 상태는 메모리에 쓰기 요청을 전달하였지만, 메모리가 아직 쓰기가 완료되었고

다음 요청을 받을 수 있다는 신호(`is_ready`)를 보내지 않아 메모리 쓰기 작업이 끝날 때까지 대기하는 상태이다. 만약 해당 신호가 1 이라면 다음 사이클에 `READ_WAIT` 상태로 전이하며, 아니라면 `WRITE_WAIT` 상태에 계속 대기한다.

READ_WAIT 상태는 메모리에 읽기 요청을 전달하였지만, 메모리가 아직 읽기가 완료되었고 현재 출력이 유효하다는 신호(`is_output_valid`)를 보내지 않아 메모리 읽기 작업이 끝날 때까지 대기하는 상태이다. 만약 해당 신호가 1 이라면 다음 사이클에 `IDLE` 상태로 전이하며, 아니라면 `READ_WAIT` 상태에 계속 대기한다.

캐시는 위와 같은 유한 상태 기계의 상태와 메모리, CPU에서 전달되는 신호에 따라 현재 사이클에서 할 일을 결정한다. 예를 들어, `IDLE` 상태에서 CPU 쓰기 요청이 전달되었는데, 요청된 블록이 현재 캐시에 존재하는 캐시 적중(Cache Hit) 상황에서는 CPU에서 전달된 데이터를 캐시의 해당 블록에 쓰고 해당 블록이 수정되었음을 표시한다. 유사하게, `READ_WAIT` 상태에서 메모리가 현재 출력이 유효하다는 신호가 활성화된 경우, 메모리의 데이터 출력을 캐시의 해당 라인에 쓴다.

본 과제에서 구현한 집합 연관 사상 방식 캐시는 강의 교안의 것을 기반으로 구현하였다. 구현된 캐시의 대략적인 구조도는 다음과 같다.

이전 과제에서 구현한 pipelined CPU는 magic memory를 가정하고 구현되었으며, 따라서 메모리가 모든 request에 1사이클만에 응답할 것을 가정하였다. 이번 과제에서는 이와 다르게, 메모리 및 cache가 request에 바로 응답하지 못할 수도 있다. 때문에, CPU에서는 반드시 Cache의 `is_ready`, `is_hit`, `is_output_vaild` signal을 평가하여 pipeline을 stall할지 결정하여야 한다. 따라서, CPU는 현재 메모리의 현재 상태에 따라 새로운 stall signal을 생성하여 pipeline의 이전 stage에 전달하여 만약 stall이 필요한 경우 Cache가 해당 request를 처리할 때까지 대기하게 한다.

이전 과제에서 구현한 파이프라인 CPU는 이상적 메모리를 가정하고 구현되었으며, 따라서 메모리가 모든 요청에 1 사이클만에 응답할 것을 가정하였다.

3 Implementation

CPU module의 구현은 Lab Assignment #4까지 구현한 것과 (Design section에서 설명한 부분을 제외하면) 대동소이하며, 따라서 이 보고서에서는 cache module의 구현에 집중하여 설명한다.

3.1 Cache 상태 전이

```

1  always @(*) begin
2      case(state)
3      `IDLE: begin
4          if (is_input_valid && !is_hit)
5              next_state = dirty[index][write_way_index] ? `WRITE_WAIT : `READ_WAIT;
6          else
7              next_state = `IDLE;
8      end
9      `WRITE_WAIT: next_state = is_dmem_ready ? `READ_WAIT : `WRITE_WAIT;
10     `READ_WAIT: next_state = is_dmem_output_valid ? `IDLE : `READ_WAIT;
11     default: next_state = `IDLE;
12     endcase
13 end

```

img/schematic.png

집합 연관 캐시의 구조도

Cache의 다음 상태를 계산하는 logic

Cache의 내부 상태는 위와 같이 IDLE로 초기화되어, CPU의 request 및 cache hit 여부, 쓰고자 하는 line의 dirty 여부, 메모리의 output signal에 따라 WRITE_WAIT, READ_WAIT 상태로 전이한다.

3.2 Hit index, write index의 계산

```
1 // Logic that finds the index of the way that hit
2 always @(*) begin
3     // Notice that hit_way_index may cause unexpected behaviour when used
4     // without testing hit_way_found. Usually you may want to examine is_hit,
5     // which essentially is just an alias for hit_way_found, before using
6     // hit_way_index.
```

```

7   hit_way_found = 0;
8   hit_way_index = 0;
9
10  for (i = 0; i < NUM_WAYS; i = i + 1) begin
11      if (tag == tag_bank[index][i] && valid[index][i]) begin
12          hit_way_index = i[WAY_INDEX_LEN - 1: 0];
13          hit_way_found = 1;
14      end
15  end
16 end
17
18 ...
19
20 // Logic that finds the index of the way to write on
21 always @(*) begin
22     write_way_found = 0;
23     write_way_index = 0;
24
25     for (i = 0; i < NUM_WAYS; i = i + 1) begin
26         if (!valid[index][i]) begin
27             write_way_index = i[WAY_INDEX_LEN - 1: 0];
28             write_way_found = 1;
29         end
30     end
31
32     if (!write_way_found)
33         write_way_index = lru;
34 end

```

Hit index를 계산하는 logic

Cache module 내에서 hit_way_index, write_way_index는 cache hit 상황에서 hit이 발생한 way의 index, cache에 write해야 하는 상황에서 write해야 할 way의 index를 저장하는 레지스터이다. 위 logic은 기본적으로 cache의 각 way를 순회하여, 만약 hit condition tag = tag_bank[index][i] && valid[index][i]를 만족하는 way의 index i를 hit_way_index에 저장한다. write_way_index 또한 비슷하게 cache의 각 way를 순회하여, invalid way를 먼저 찾는다. 만약 invalid way가 없다면, least recently used way를 write_way_index에 저장하여 해당 way에 write와 동시에 evict가 일어나도록 한다.

이때 주의할 점은 hit_way_index는 만약 hit way를 찾지 못하였을 경우 0을 저장하며, 이 때문에 hit_way_found signal을 먼저 테스트하지 않고 사용했을 경우 이 0이 그대로 index로 사용되어 예상하지 못한 현상을 일으킬 수 있다는 점이다. 따라서, hit_way_index는 반드시 hit_way_found, 혹은 이의 별칭인 is_hit을 테스트한 후에 사용되어야 한다.

3.3 Cache write logic

```

1
2 ...
3
4 end else begin
5     state <= next_state;
6
7     if (state == `IDLE && is_input_valid && is_hit && mem_rw) begin

```

```

8     data_bank[index][hit_way_index][32 * offset +: 32] <= din;
9     dirty[index][hit_way_index] <= 1;
10 end
11
12 if (state == `READ_WAIT && is_dmem_output_valid) begin
13     valid[index][write_way_index] <= 1;
14     dirty[index][write_way_index] <= 0;
15     tag_bank[index][write_way_index] <= tag;
16     data_bank[index][write_way_index] <= dmem_dout;
17 end
18
19 ...
20
21 end

```

Tag bank, data bank에 write하는 logic

CPU에서 cache에 write request를 보내거나, 메모리에서 read request에 대한 응답이 도착하였다면 cache의 tag bank, data bank에 해당 데이터를 쓰고 valid, dirty bit를 업데이트해야 한다. 따라서, cache는 매 사이클마다 현재 상태와 CPU request, hit 여부, 메모리 output signal을 점검하여 필요한 경우 해당 line의 값을 업데이트한다.

3.4 LRU 업데이트 및 LRU way 계산 logic

```

1 // Logic that finds the least recently used way in a set
2 always @(*) begin
3     lru = 0;
4     for (i = 0; i < NUM_WAYS; i = i + 1)
5         lru = (lru_bank[index][i] == MAX_LRU) ? i[WAY_INDEX_LEN - 1: 0] : lru;
6 end
7
8 ...
9
10 if (state == `IDLE && is_input_valid && is_hit) begin
11     for (i = 0; i < NUM_WAYS; i = i + 1) begin
12         if (lru_bank[index][i] != MAX_LRU)
13             lru_bank[index][i] <= lru_bank[index][i] + 1;
14     end
15     lru_bank[index][hit_way_index] <= 0;

```

LRU way의 index를 계산하고, LRU bank를 업데이트하는 logic

Cache에서는 least recently used way를 계산하기 위해, 각 set 및 way마다 해당 way가 마지막으로 쓰인 시점을 저장하는 lru_bank를 가지고 있다. 어떤 way가 hit될 때마다, 그 set, way의 lru_bank는 0으로 재설정되고 해당 set의 least recently used way를 제외한 나머지 way들은 1씩 늘어난다.

반대로, 어떤 way가 least recently used인지를 계산할 때, cache는 requested set의 각 way를 모두 순회하여 lru_bank가 MAX_LRU의 값을 가지는 way를 찾는다. MAX_LRU는 NUM_WAYS - 1의 값을 가지는 상수로, 어떤 way의 lru_bank가 MAX_LRU라는 것은 해당 way가 least recently used way임을 뜻한다.

쉽게 말해, 영희, 철수, 민수, 길동이 있는 그룹에서 무작위로 한 명씩을 호명한다고 하자. 이때 찾고 싶은 것은 호명된 이후 가장 많은 시간이 지난 사람이다. 이를 위해 ”영희는 1등, 철수는 3등, 민수는 0등, 길동은 2등”과 같은 정보를 저장해두고, 매 호명마다 호명된 사람은 0등으로, 3등을 제외한

나머지의 등수를 1씩 올린다. 이후 "호명된 이후 가장 많은 시간이 지난 사람은 누구인가?"와 같은 질의가 들어오면 현재 3등인 사람을 알려주면 될 것이다. 위 코드는 이와 같은 아이디어를 구현한다.

4 Discussion

4.1 How to Handle Branch Prediction

가장 기본적인 Branch Prediction은 Always Taken과 Always Not Taken 전략이 있다. 이는 다음과 같은 특성을 가진다.

- Always Taken: 프로그램 실행 중 마주치는 모든 Branch Instruction에 대해서 Branch가 Taken이 될 것이라 예상하여 Branch Taken 시의 PC에 대해서 미리 Instruction을 Fetch한다.
- Always Not Taken: 프로그램 실행 중 마주치는 모든 Branch Instruction에 대해서 Branch가 Not Taken이 될 것이라 예상하여 Branch Not Taken 시의 PC인 PC+4에 대해서 미리 Instruction을 Fetch한다.

가장 기본적인 전략이지만, 특정 경우에 편향하여 예측하기 때문에, 이를 보완하기 위해서 Counter 기반의 전략을 선택하게 된다. Counter 기반의 대표적인 예시는 다음과 같다.

- 1-Bit Counter: Predict Take, 1'b1, Predict Not Take, 1'b0의 상태를 가지는 Counter 기반의 전략이다.
- 2-Bit Counter: Strongly Taken, 1'b11, Weakly Taken, 1'b10, Weakly Not Taken, 1'b01, Strongly Not Taken, 1'b00의 4가지 상태를 가지는 Counter 기반의 전략이다. 상태의 전환에 따라서 Saturation Counter와 Hysteresis Counter로 나뉜다.

Counter에서 사용되는 bit 수에 따라서 Branch Taken/Not Taken Pattern History를 확인할 수 있다. 초반의 Counter 기반 Branch Predictor는 Branch의 위치(Program Counter)마다 해당 Pattern History를 저장하여 예측하는 방법으로 구현되었다.

이에서 발전된 형태인 Global Path History를 적용하여 McFarling이 제시한 Gshare Branch Prediction은 기존의 PHT와 BTB의 원리처럼 각 PC에 해당하는 Pattern History도 저장하지만, 동시에 Global Path를 확인하여 Branch Prediction에 성능적 향상을 가져왔다.

이를 확장하여 2-Level Branch Direction Predictor 형태로 발전시켰는데, BHT(Branch History Table) + PHT(Pattern History Table)을 결합 형태이다. 각각의 BHT와 PHT의 형태가 Global, per-Set, per-Address인지에 따라서 다양한 형태의 Branch Predictor를 구성한다.

본 과제에서 우리는 2-Bit Saturation Counter 기반의 Branch Predictor를 사용하여 Branch Prediction을 진행하였다.

4.2 2-Bit Saturation Counter Branch Predictor의 구현

본 과제에서 구현한 2-Bit global branch predictor는 2-Bit Saturation Counter를 기반으로 한다. Branch Prediction에 의한 Counter의 유한 상태 기계의 Diagram은 다음과 같다.

Branch Predictor의 구조는 다음 그림과 같으며, BTB의 index bit는 N=4로 설정되어 전체 32개의 Branch Target Entry를 가지도록 디자인하였다.

2-Bit Saturation Counter는 Strongly Taken, Weakly Taken, Weakly Not Taken, Strongly Not Taken의 4가지 상태로 나타나며, 각 상태에 대한 Predictor의 Prediction과 실제 branch의 taken, not taken에 의한 상태 변화는 아래의 표와 같다. (실제 코드에서는 Implementation에서 확인할 수 있듯이, BP_ST, BP_WT, BP_WN, BP_SN으로 각 상태를 나타낸다.)

img/2bit_saturation_counter_FSM_diagram.png

FSM Diagram of 2-Bit Saturation Counter

	Prediction	Taken(Real)	Not-Taken(Real)
Strongly Taken(2'b11)	Taken	Strongly Taken	Weakly Taken
Weakly Taken(2'b10)	Taken	Strongly Taken	Weakly Not Taken
Weakly Not Taken(2'b01)	Not Taken	Weakly Taken	Strongly Not Taken
Strongly Not Taken(2'b00)	Not Taken	Weakly Not Taken	Strongly Not Taken

Comparison on the Number of Cycles on Different Branch Predictor

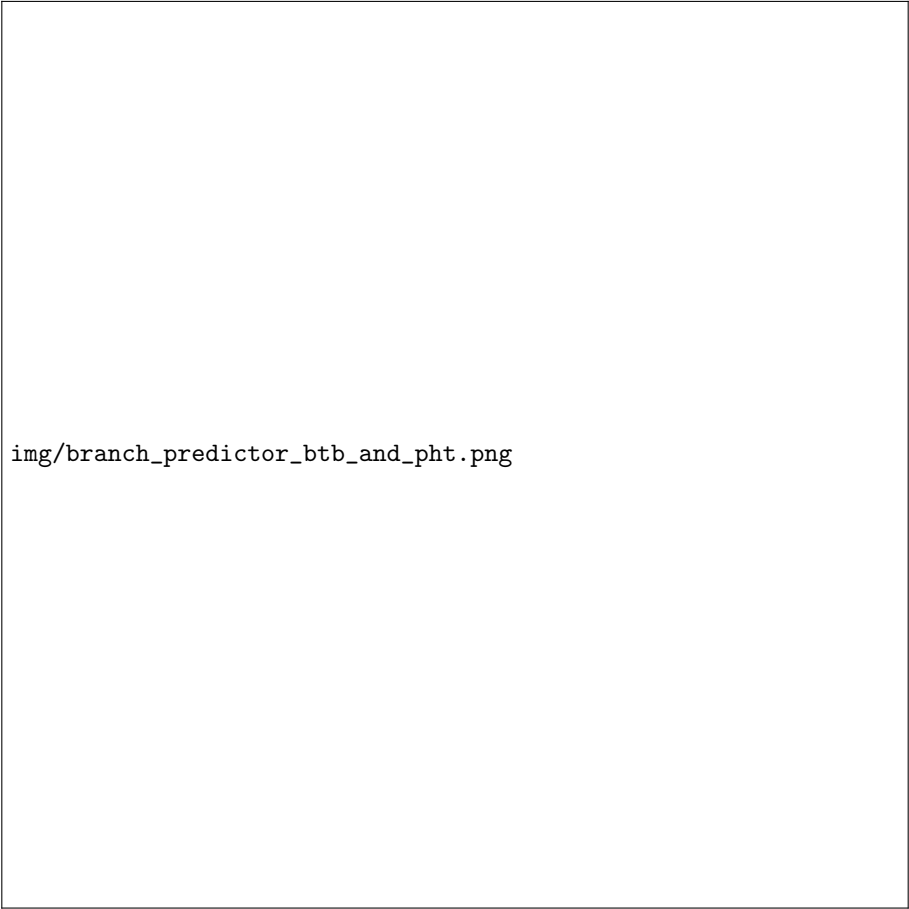
4.3 다른 Branch Predictor와의 비교

다른 방식의 Branch Prediction(Always-Taken, Always-Not-Taken)과 본 과제에서 구현한 2-Bit Saturation Counter Branch Prediction 성능을 비교한 결과는 아래와 같다.

Input File	Always-Taken	Always-Not-Taken	Saturation Counter
basic	37	37	37
non-controlflow	46	46	46
ifelse	51	51	51
loop	354	362	357
recursive	1153	1229	1159

Comparison on the Number of Cycles on Different Branch Predictor

실험 결과에 따르면, 2-bit Saturation Counter는 Always-Not-Taken 기반 Branch Prediction에



img/branch_predictor_btb_and_pht.png

Branch Predictor with BTB and PHT

비해서는 확연히 나은 성능을 보여주나, Always-Taken 기반에 비해서는 구현의 복잡도가 더 높음에도 불구하고, 오히려 실행에 더 많은 사이클이 걸리는 것을 볼 수 있다. 이는 2-bit Saturation Counter의 경우, Always-Taken에 비해 잘못된 예측을 바로잡기 위해 한 번 더 해당 분기에 들어가 Pattern History Table을 수정하는 과정이 필요해서이다. 따라서, 만약 같은 분기를 더 많이 방문하며, 해당 분기의 실제 방향이 가끔씩 변하는 프로그램이라면, 2-bit Saturation Counter가 Always-Not-Taken 기반 Branch Predictor에 비해 실행에 걸리는 사이클 수가 더 적을 것이다.

5 Conclusion

이번 과제에서는 지난 4-1에 이어 Pipelined CPU의 Control Flow DataPath를 구현하는 것을 목적으로 하여, Pipelined CPU를 완성하였다. 기존 Multi-Cycle CPU와는 달리 Branch Taken/Not Taken 여부와 Branch Instruction 다음 Fetch하는 Instruction의 위치에 따라서 Stall이 발생할 수 있기 때문에, Branch Prediction이 중요하다. 2-Bit Saturation Counter 기반의 2-Bit Global Branch Predictor를 구현하여 기존 Always-Taken, Always-Not-Taken 전략과의 비교할 수 있었지만, 주어진 프로그램에 의존하는 특성상 정확한 비교를 할 수 없어 기대한 결과를 확인하지 못하였다. Single-Cycle CPU, Multi-Cycle CPU, Pipelined CPU의 발전 동안 Data Memory 관점에서의 개선은 미비하였는데, Data Memory 접근성을 늘릴 수 있는 Cache의 필요성을 확인할 수 있었다.