CSED311 Lab4-3a

-Cache-

Team ID: 2

20180740 염재후

1. Introduction

이전 랩들까지는 1 cycle만에 메모리에 접근이 가능하다고 가정한, magic memory를 사용했다. 하지만, 실제로는 메모리에 1cycle만에 접근할 수 없고, memory element를 빠른 시간 내에 접근하기 위해서 cache를 사용한다.

이번 랩에서는 Cache와 delay가 있는 Data memory를 사용하여, 현실 세계에서 Memory element를 효율적으로 사용하는 방법에 대해서 알아보도록 한다. 이번 랩의 제한사항은 다음과 같다. 먼저 Data Cache만 구현하고, Cache size는 256Byte이다. Cache의 구현은 Direct mapped와 n-way associativity중 선택하여 구현한다. Cache의 hit/miss policy는 write back, write allocate policy를 가지고 운영된다.

2. Design

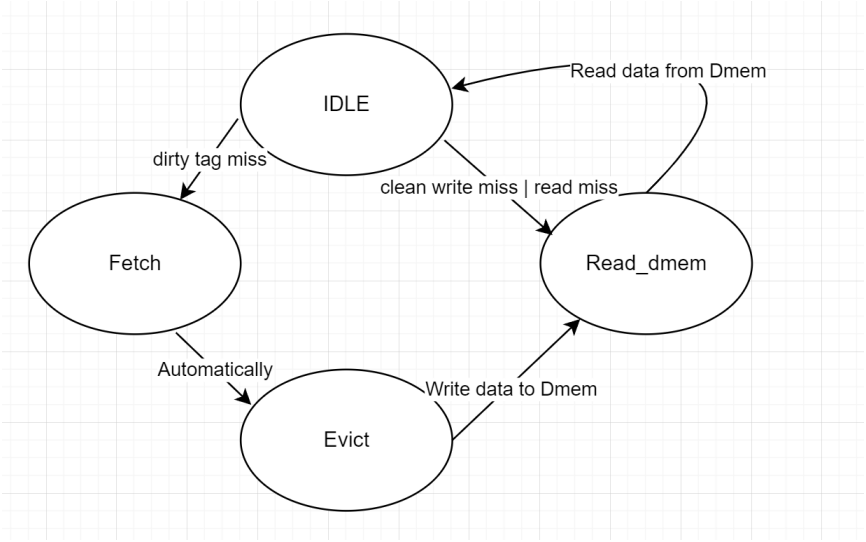
Cache Lab 디자인은 크게 세 가지로 나뉜다. Data Cache, Cache Controller(using Finite State Machine), Cache stall이다. 그 중 Cache Design에는 선택지가 있다. Direct mapped와 n-way associative이다. Direct mapped는 한 index당, 하나의 tag만을 허용하며, n-way associative는 한 index마다 n개의 tag를 허용한다. 단, 실제로 hit하는 것은 그 중 하나의 tag만이 hit하게 된다.

나는 direct mapped cache를 선택하였다. 먼저 Cache Size C = 256 Bytes이고, Block Size B = 16 Byte이다. 따라서, 총 16개의 cache line이 존재하므로, index bit로는 4bits가 필요하다. B=16이기 때문에, block offset, Byte offset으로 각각 2bits씩 필요하다. 따라서, 총 8bits가 필요하므로, tag bit는 24bit가 된다.

Cache Controller로는 총 state가 4개인 Finite State machine을 사용하였다. 각각 Idle, Fetch, Evict, Read이며 각각 state의 의미는 다음과 같다.

1. Idle: write or read hit이며, 어떠한 write도 처리하고 있지 않은 기본 상태
2. Fetch: Idle에서 바로 Evict로 가는 경우, 1 cycle만에 Evict에서 Read로 가는 상황이 발생하여서 넣은 임의의 state이다.
3. Evict: dirty한 Cache line에서 read miss, write miss가 발생하여 Data memory로부터 읽어오기 전에 Evict를 처리하는 state
4. Read_dmem: clean read miss, 혹은 invalid로 인한 write miss, evict가 완료된 후 넘어오는 경우를 다루는 state이다

Finite State Machine을 위한 Transition Diagram의 개요는 다음과 같다.



마지막으로 Cache Stall implementation이다. 우리가 설계하는 CPU는 in-order CPU이기 때문에, Data Cache miss로 인한 stall이 MEM stage에서 나타나게 된다면, EX stage의 instruction이 cache나 memory에 접근하지 않더라도 멈춰야만 한다. 따라서, Stall이 필요한데, 이는 Data hazard(by load instruction)에서 고려해야 할 점과 일부분 다르다. 먼저, data forwarding과 register update를 막아야 한다. 왜냐하면 data forwarding이 진행되는 경우, 원하지 않는 값들이 alu에 들어가서 이상한 연산 결과를 나타낼 수 있기 때문이다. 그 외에는 load stall과 같이 모든 register의 update를 막되, IF/ID, ID/EX, EX/MEM, MEM/WB를 모두 막는다.

3. Implementation

세 가지로 나누어 디자인했으므로, 구현 역시 세 부분으로 나뉜다.

```
// Reg declarations
reg [LINE_SIZE * 8 -1: 0] Cache_line[0: LINE_SIZE-1];
reg valid_bit[0: LINE_SIZE-1];
reg dirty_bit[0: LINE_SIZE-1];
reg [23:0]tag_bit[0: LINE_SIZE-1];

// You might need registers to keep the status.
// 00 is for IDLE, 01 is for evicting data
// 10 is for reading data from DMEM, 11 is for errored state
reg [1:0]state;
reg miss;
reg dmem_input_valid;
reg dmem_read;
reg dmem_write;
reg [31:0]dmem_addr; // tag(24) + index(4) = total 28 bit
reg [LINE_SIZE * 8 -1 : 0] dmem_din;
```

먼저, Cache를 구성하는 reg이다. Cache line과 valid, dirty, tag bit를 위한 각각의 register들과, register 상태를 제어하기 위한 register들이다.

```
always@(*) begin
  // read hit or write hit
  if(is_input_valid && is_valid && tag_hit && !miss) begin
      case (block_offset)
        0 : dout = Cache_line[index][31:0];
        1 : dout = Cache_line[index][63:32];
        2 : dout = Cache_line[index][95:64];
        3 : dout = Cache_line[index][127:96];
      endcase
      is_hit = 1;
      is_output_valid = 1;
      dmem_input_valid = 0;
      dmem_read = 0;
      dmem_write = 0;
  end

  // read miss, write miss
  else if(is_input_valid && (!is_valid || !tag_hit || miss) ) begin
    is_output_valid = 0;
    is_hit = 0;
  end

  else if(!is_input_valid) begin
    is_hit = 1;
    is_output_valid = 1;
  end
```

각각 Cache에서 Asynchronous하게 생성하는 signal들이다. Valid한 경우에 Data를 내보내주고, hit 여부와 output valid를 결정해준다. 또한 input이 valid하지 않은 경우, 즉 read나 write instruction이 아닌 경우, 항상 hit, output valid하다고 만들어준다.

```
else if(is_input_valid)begin //state transition
  if(state == `Idle) begin
    //dirty -> evict
    if(is_input_valid && is_dirty && is_valid && !tag_hit ) begin
      state <= `Fetch;
      miss <= 1;
      dmem_input_valid <= 1;
      dmem_write <= 1;
      dmem_read <= 0;
      dmem_addr <= {4'd0, tag_bit[index], index};
      dmem_din <= Cache_line[index];
      dirty_bit[index] <= 0;
      valid_bit[index] <= 0;
    end

    //clean or read miss
    else if(is_input_valid && (!is_valid || (!is_dirty && !tag_hit) )) begin
      state <= `Read_dmem;
      dmem_input_valid <= 1;
      dmem_read <= 1;
      miss <= 1;
      dmem_addr <= {4'd0, tag, index};
    end

    else if(mem_write && is_valid && tag_hit) begin
      case (block_offset)
        0: Cache_line[index][31:0] <= din;
        1: Cache_line[index][63:32] <= din;
        2: Cache_line[index][95:64] <= din;
        3: Cache_line[index][127:96] <= din;
      endcase
      dirty_bit[index] <= 1;
    end

    else begin
      state <= `Idle;
    end
  end
```

```
  else if(state == `Evict) begin // evict state
    dmem_input_valid <= 0;
    dmem_write <= 0;
    dmem_read <= 0;
    if(is_data_mem_ready) begin
      state <= `Read_dmem;
      dmem_input_valid <= 1;
      dmem_read <= 1;
      dmem_addr <= {4'd0, tag, index};
    end
    else begin
      state <= `Evict;
    end
  end

  else if(state == `Read_dmem) begin // read state
    dmem_input_valid <= 0;
    dmem_write <= 0;
    dmem_read <= 0;
    if(dmem_output_valid && is_data_mem_ready) begin
      state <= `Idle;
      Cache_line[index] <= dmem_dout;
      valid_bit[index] <= 1;
      tag_bit[index] <= tag;
      dirty_bit[index] <= 0;
      miss <= 0;
  end
    else begin
      state <= `Read_dmem;
    end
end


  else if(state == `Fetch) begin
    state <= `Evict;
  end


  else
    state<= state;
  end
d
```

다음은 Synchronous State Transition이다. Cache는 Synchronous하게 transition하며, 조건이 만족하는 경우, data를 써준다. 이 때, 주의해야 할 점은 각 state는 evict 혹은 read하는 단계가 아니라, 그 작업이 data memory에서 완료되기를 기다리는 state라는 것이다. 따라서, state가 transition할 때, 다음 state에서 해야 할 작업에 맞는 signal과 value를 assign해준다. 또한, data memory 내부 코드에서, valid input인 경우, 혹은 read/write signal이 들어온 경우 delay를 시작한다. 따라서, data memory에서 처리하고 있는 동안 계속해서 signal이 asserted된 상태라면, 무의미하게 delay가 2번 연속되는 경우가 발생한다. 따라서, 각 clock cycle마다, transition될 때를 제외한다면, data memory를 작동시키는 signal을 0로 만들어준다.

마지막으로 Cache stall이다. Cache stall은 CPU는 Data memory가 새 request를 받을 준비가 되었으며, hit고, output이 valid한 경우에만 진행된다. 따라서 다음과 같은 논리식을 통해 Cache가 stall할 조건을 특정할 수 있다. 또한 앞서 언급했듯, cache가 stall하게 된다면 Data forwarding을 막아서, ALU가 정확하지 않은 값을 연산하는 것을 막아주도록 한다.

```
//Cache_stall fowarding blocking
assign rs1_dout_fwd = Cache_stall ? rs1_dout : (internal_forwardA == 1) ? rd_din : (is_ecall && EX_MEM_rd == 17 && EX_MEM_reg_write) ? EX_MEM_alu_out : rs1_dout;
assign rs2_dout_fwd = Cache_stall ? rs2_dout : (internal_forwardB == 1) ? rd_din : rs2_dout;
assign tmp_halt = is_ecall && (rs1_dout_fwd == 10);
```

```
assign Cache_stall = (is_ready && is_output_valid && is_hit) == 0;
```

```
PC pc(
  .reset(reset),       // input (Use reset to initialize PC. Initial value must be 0)
  .clk(clk),          // input
  .next_pc(next_pc),     // input
  .nextPCWrite((nextPCWrite && !Cache_stall)),
  .current_pc(current_pc)   // output
);
```

또한, 당연하게도, Stall이 되는 동안, 새로운 PC로 update되는 것을 막도록 한다.

4. Discussion

Cache를 구현하면서 fowarding과 관련된 문제를 겪었다. Stall을 하는 도중 data forwarding이 계속 진행되며 register output이 xxxx xxxx로 나타나는 현상이 나타나게 되었고, 이를 막기 위해서, cache로 인한 stall 중에는 data forwarding을 막아두었다.

다음으로는 Cycle수를 비교해보도록 하겠다.

naïve mat mul: 70322 cycles

opt mat mul: 75656 cycles

가 나오게 되었다. 또한 각 Test Bench의 hit ratio는 다음과 같다.

	Hit	Miss	Hit-ratio
Naïve	1687	812	67%
Opt	1605	894	64%

일반적으로는 blocking method를 사용한 opt mat mul의 cycle이 더 적게 나와야 한다. 그 이유는, block size를 1 word가 아닌 4 word로 디자인했기 때문에, 한 번 cache에서 load하게 되면, 다른 block들까지 같이 prefetching하게 되는데, 이를 잘 사용하는 방법이 blocking method이다. Blocking method에 대해서 간단히 설명하자면, 일반적인 matrix multiplication, A * B = C는 A의 row와 B의 column간 dot product로 C의 element를 계산하게 되는데, B의 column 원소들을 fetching하게 되면, 4words를 fetching한 후, 1word만 사용하고 다른 block을 fetch해야하기 때문이다. 반면, blocking method는 한 번 불러온 4words를 다 활용한 뒤 새로운 block을 불러오기 때문에, cache를 fully utilize 한다고 할 수 있다.

하지만 Test Bench의 결과는 그렇지 않다고 말해주고 있다. Blocking method를 사용한

경우, 더 많은 miss가 일어나는 것을 확인할 수 있었다. 이론적인 부분에서는 최적화가 잘 된 경우, Blocking method는 더 좋은 성능을 보장한다. 따라서, opt가 더 많은 miss를 만들어낸 이유는 잘못된 최적화라고 결론지을 수 있다.

만약, set이나 way의 숫자를 바꾼다고 가정해보자. Fully associated set의 경우와 비교할 경우, 그 차이를 바로 알 수 있다. 이 경우, tag가 다르고, index가 같은 경우, evict하지 않고 cache를 다른 way에 저장하게 된다. 따라서 evict가 줄어들기 때문에, 전체 cycle의 수 자체는 줄어든다고 볼 수 있다. 즉 hit ration의 증가로 이어질 것이다. 하지만, 실제 경우에는, mux와 같은 다른 cost consuming하는 경우가 많기 때문에 특별한 경우에만 사용한다.

5. Conclusion

이번 랩을 통해서, data cache를 구현하는 방법에 대해서 알 수 있었다. 또한 cache controller를 위한 finite state machine을 구현하면서 write back, write allocate cache의 작동 semantics에 대해서 고민할 수 있는 기회가 되었다고 생각한다.