lab 6. cache

20190316 유병호

20190065 강두경

Introduction

이번 Lab 6의 목표는 lab5에서 만들었던 pipelined cpu를 기반으로, cache가 있는 cpu와 cache가 없는 cpu를 구현하는 것이다.

cache가 없는 cpu(baseline cpu)는 memory에서 one word의 data를 받아오고 이때 2 cycles가 걸린다. cache가 있는 cpu의 cache는 2-way set associative, single-level cache로 한 line의 size는 4 words이면서 caceh의 capacity는 32 words이다. 그리고 cache hit일 때는 1 cycle만에 data를 받아올 수 있고, cache miss일 때는 main memory에서 4 words의 data를 6cycle만에 받아온다.
그리고 이 둘의 cycle 차이를 비교하여 cache가 cpu의 성능을 얼마나 상승시킬 수 있을 지 확인해본다.

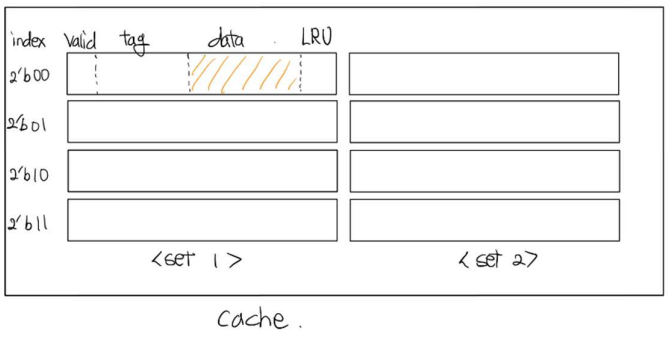
DESIGN

cache가 없는 cpu

cache가 없는 cpu는 1word의 data를 메모리에서 2cycle 만에 가지고 오도록 디자인해야한다. 이를 위해서 memory에 대한 접근이 있을 때 무조건 한 cycle의 delay를 주었다.

cache가 있는 cpu

2-way set associative, single-level cache이며 line size는 4 words이면서 cache의 capacity는 32 words이다. 따라서 이런 캐시는, 1set의 index size가 4인 캐시의 구조를 갖게 된다.



cache의 기초적인 구조

그렇기 때문에 address의 tag, index, offset bit의 모습은 다음과 같다.

|--- tag bit ---||-- index bit --||-- byte bit --|
15 4 3 2 1 0 (bit)

tag bit의 size는 12 bit, index bit의 size는 2bit, byte bit의 size는 2bit이다.

그리고 다음은 cache의 동작과정이다.
1. address의 index bit를 통해 해당하는 set1 cache와 set2 cache의 line을 찾아간다.
2. 해당하는 cache의 line들에서 address의 tag bit를 이용해 hit여부를 판단한다.
3.1.1 만약 두 set의 line중 hit이 발생하면 해당하는 line의 data를 output으로 준다.
3.1.2 그리고 hit이 발생한 set의 line의 lru값을 0으로 바꿔주고 다른 line의 lru 값을 1로 바꿔준다.
3.2.1 만약 miss가 발생하면 lru값이 1인 라인의 값을 update해준다.
3.2.2 그리고 update한 set의 line의 lru값을 0으로 바꿔주고 다른 line의 lru 값을 1로 바꿔준다.

그리고 우리는 write through 방식으로 memory와 cache에 대한 write를 수행해주었다.

cache가 있는 cpu를 작성함에 있어서 가장 중요한 것은 memory와 cpu가 직접 interaction하지 않고, cpu와 cache, cache와 memroy가 interaction해야 한다. 따라서 기존에 cache가 없는 cpu에서 memory에서 data를 읽어오거나 data를 작성할 때 사용되는 read_m1, read_m2, write_m2는 cache에 대한 signal로 사용되고, 그리고 cache는 memory에 대한 mem_read_m1, mem_read_m2, mem_write_m2의 signal과 stall signal을 miss일 경우 생성해주어야 한다.

IMPLEMENTATION

Baseline CPU

```
always@(posedge clk)
  if(!reset_n)
```

```
        begin ...
        end
    else begin
        // if(read_m2)output_data2 <= memory[address2];
        // if(write_m2)memory[address2] <= data2;

        if (read_m1_delay == 0) begin
            if (read_m1)
                read_m1_delay <= 1;
        end
        else if (read_m1_delay == 1) begin
            data1 <= (write_m2 & address1 == address2) ? data2 : memory[address1];
            read_m1_delay <= 0;
        end

        if (read_m2_delay == 0) begin
            if (read_m2)
                read_m2_delay <= 1;
        end
        else if (read_m2_delay == 1) begin
            output_data2 <= memory[address2];
            read_m2_delay <= 0;
        end

        if (write_m2_delay == 0) begin
            if (write_m2)
                write_m2_delay <= 1;
        end
        else if (write_m2_delay == 1) begin
            memory[address2] <= data2;
            write_m2_delay <= 0;
        end
    end
```

memory에서 data를 가져오는 한 cycle의 delay(data를 가져오는데 총 2 cycle)은 다음과 같이 memory에 대한 접근에 대한 signal인 read_m1, read_m2, write_m2가 1일 때 delay를 주고 있다.

```
        if (!mem_stall) begin
            if (!datahazard) begin
                PRE_PC <= next_PC;
                IFID_PC <= PRE_PC;
                IFID_INSTR <= (is_flush) ? 16'h0000 : data1;
            end
            //mem_stall <= IDEXC_MEMREAD || IDEXC_MEMWRITE;
            mem_stall <= 1;
                    .....
        end
        else begin
            mem_stall <= 0;
            MEMWBC_NEWINST <= 0;
        end
```

그리고 datapath의 sequential logic에서 mem_stall이라는 register의 값을 1과 0을 반복하며, 1일 때는 정상적으로 pipeline이 수행되고 0일 때는 stall되게 되도록 하여, 의도적으로 stall이 한 번씩 이루어지게했고 이를 통해 memory에서 data를 가져오는 delay에 대한 pipeline의 stall을 구현하였다.

Pipelined CPU with Cache

다음은 우리가 구현한 cache의 input과 output에 대한 설명이다.

```
    input         clk,
    input         reset_n,
    input [15:0]   address1,
    input [15:0]   address2,
    input         cpu_read_m1,
    input         cpu_read_m2,
    input         cpu_write_m2,
    input         mem_signal,
    input [63:0]   mem_data1, //from memory, load
    input [63:0]   mem_data2,
    input [15:0]   cpu_data, //from cpu, store

    output reg     hit,
    output         stall,
    output         mem_read_m1,
    output         mem_read_m2,
    output         mem_write_m2,
    output [15:0]  mem_address1,
    output [15:0]  mem_address2,
    output [15:0]  mem_write_data,
    output [15:0]  wb_data1,
    output [15:0]  wb_data2
```

<input>

address1: instruction에 대한 address

address2: data에 대한 address

cpu_read_m1, cpu_read_m2, write_m_2 : cpu의 read_m1, read_m2, write_m2

mem_signal: memory에서 cache를 update해줄 data를 받아왔음을 알려주는 signal

mem_data1: memory에서 받아온 instruction에 대한 data

mem_data2: memry에서 받아온 data 에 대한 data.

<output>

hit: cache에서 hit이 되어 memory에 대한 접근이 필요없음을 알려주는 signal

stall: miss가 발생하여 stall 이루어져야 한다는 것을 알려주는 signal

mem_read_m1, mem_read_m2, mem_write_m2: cache에서 miss가 발생했을 때 instruction을 load해야 하는지, data를 load해야하는지 아니면 memory에 data를 write해야하는지 memory에게 알려주는 signal.

wb_data1, wb_data2: datapath의 wb_data1, wb_data2.

```
  reg                set1_valid  [0:`IDX_SIZE - 1];
  reg [11:0]          set1_tag    [0:`IDX_SIZE - 1];
```

```
reg [`WORD_SIZE - 1:0]  set1_data   [0:`IDX_SIZE - 1][0:`OFFSET_SIZE - 1];
reg                     set1_lru    [0:`IDX_SIZE - 1];
reg                     set2_valid  [0:`IDX_SIZE - 1];
reg [11:0]               set2_tag    [0:`IDX_SIZE - 1];
reg [`WORD_SIZE - 1:0]  set2_data   [0:`IDX_SIZE - 1][0:`OFFSET_SIZE - 1];
reg                     set2_lru    [0:`IDX_SIZE - 1];
```

cache 내부에서 문제조건에 부합하는 2 set의 구현은 다음과 같이 해주었다.

```
assign addr1_set1_hit = set1_valid[address1_index] && set1_tag[address1_index] == address1_tag;
assign addr1_set2_hit = set2_valid[address1_index] && set2_tag[address1_index] == address1_tag;
assign addr1_hit = addr1_set1_hit || addr1_set2_hit;
```

cache 내부에서 hit 판별은 다음과 같이 해당하는 index의 line에 저장되어 있는 tag가 현재 address의 tag와 같은지로 판단한다.

```
assign mem_read_m1 = cpu_read_m1 && !addr1_hit;
assign mem_read_m2 = cpu_read_m2 && !addr2_hit;
```

main memory에서 load하여 data를 가져와야할 때는 cache에서 miss가 발생했을 경우 이지만,

```
assign mem_write_m2 = cpu_write_m2;
```

write through를 구현했기 때문에 store instruction에 대해서는 cache와 memory 모두 data를 stoare해준다.

```
assign stall = (mem_read_m1 || mem_read_m2 || mem_write_m2) && !mem_signal;
```

이 stall signal은 cache에서 miss가 났을 경우 cache에서 datapath를 stall 시키기 위해 내보내는 signal이기 때문에 1. memory에 대한 접근 이루어지고 있고, 2. 아직 memory가 load나 store insturction을 마치지 못했을 때 1의 값을 가져 stall 시키게 된다.

```
assign addr1_mem_data = address1_offset == 0 ? mem_data1[15:0] :
                        address1_offset == 1 ? mem_data1[31:16] :
                        address1_offset == 2 ? mem_data1[47:32] :
                        mem_data1[63:48];
```

cache가 있는 cpu의 memory는 한번에 4word의 data를 output으로 주기 때문에 해당하는 offset의 1word data를 cpu에게 주게 된다. 이 때 Verilog에서 배열을 input port로 매핑하는 것이 불가능하기에 16비트 wire 4개를 64비트 wire로 flatten하는 과정을 수행하였다.

Sequential Logic

```
if (!stall) begin
                if (cpu_read_m1) begin
                    if (addr1_hit)
                        num_hit <= num_hit + 1;
                    else
                        num_miss <= num_miss + 1;
                end
```

```
            if (cpu_read_m2 || cpu_write_m2) begin
                if (addr2_hit)
                    num_hit <= num_hit + 1;
                else
                    num_miss <= num_miss + 1;
            end
            if (mem_read_m1) begin
                if (set1_lru[address1_index] >= set2_lru[address1_index]) begin
                    set1_data[address1_index][0] <= mem_data1[15:0];
                    set1_data[address1_index][1] <= mem_data1[31:16];
                    set1_data[address1_index][2] <= mem_data1[47:32];
                    set1_data[address1_index][3] <= mem_data1[63:48];
                    set1_tag[address1_index] <= address1_tag;
                    set1_valid[address1_index] <= 1;
                    set1_lru[address1_index] <= 0;
                    set2_lru[address1_index] <= 1;
                end
                else begin
                    set2_data[address1_index][0] <= mem_data1[15:0];
                    set2_data[address1_index][1] <= mem_data1[31:16];
                    set2_data[address1_index][2] <= mem_data1[47:32];
                    set2_data[address1_index][3] <= mem_data1[63:48];
                    set2_tag[address1_index] <= address1_tag;
                    set2_valid[address1_index] <= 1;
                    set2_lru[address1_index] <= 0;
                    set1_lru[address1_index] <= 1;
                end
            end
            else if (mem_read_m2) begin
                if (set1_lru[address2_index] >= set2_lru[address2_index]) begin
                    set1_data[address2_index][0] <= mem_data2[15:0];
                    set1_data[address2_index][1] <= mem_data2[31:16];
                    set1_data[address2_index][2] <= mem_data2[47:32];
                    set1_data[address2_index][3] <= mem_data2[63:48];
                    set1_tag[address2_index] <= address2_tag;
                    set1_valid[address2_index] <= 1;
                    set1_lru[address2_index] <= 0;
                    set2_lru[address2_index] <= 1;
                end
                else begin
                    set2_data[address2_index][0] <= mem_data2[15:0];
                    set2_data[address2_index][1] <= mem_data2[31:16];
                    set2_data[address2_index][2] <= mem_data2[47:32];
                    set2_data[address2_index][3] <= mem_data2[63:48];
                    set2_tag[address2_index] <= address2_tag;
                    set2_valid[address2_index] <= 1;
                    set2_lru[address2_index] <= 0;
                    set1_lru[address2_index] <= 1;
```

```
            end
        end
        if (mem_write_m2) begin
            if (addr2_set1_hit)
                set1_data[address2_index][address2_offset] <= cpu_data;
            else if (addr2_set2_hit)
                set2_data[address2_index][address2_offset] <= cpu_data;
        end
```

sequential logic에서는 cache의 update나 store가 이루어지게 된다.
cache에 대한 update는 memory에서 값을 읽어왔을 때 이루어져야 하므로 stall 의 값이 0일 때 이루어져야 한다. 그리고 lru를 통해 두개의 set에서 한 line을 선택하여 이를 가져온 memory의 data를 통해 update해준다.

store일 때는 hit일 경우에만 해당하는 set의 line의 offset에 data를 store 해준다

끝으로 cache hit rate를 구하기 위해 위에서 cache에 접근할 경우 cache hit 여부를 판별하여 hit or miss 변수를 증가시키는 logic을 추가하였다.

MEMORY

```
if (signal == 1)
signal <= 0;

if (read_m1_delay == 0)
   read_m1_delay <= read_m1;
else if (read_m1_delay > 0 && read_m1_delay < `DEF_DELAY)
   read_m1_delay <= read_m1_delay + 1;
else if (read_m1_delay == `DEF_DELAY) begin
data1_out[15:0] <= (write_m2 & address1 == address2) ? data2_out[15:0] : memory[{address1[15:2], 2'b00}];
data1_out[31:16] <= (write_m2 & address1 == address2) ? data2_out[31:16] : memory[{address1[15:2], 2'b01}];
data1_out[47:32] <= (write_m2 & address1 == address2) ? data2_out[47:32] : memory[{address1[15:2], 2'b10}];
data1_out[63:48] <= (write_m2 & address1 == address2) ? data2_out[63:48] : memory[{address1[15:2], 2'b11}];
read_m1_delay <= 0;
signal <= 1;
end
```

cache에서 memory에서 data를 load해오라는 signal(read_m1)을 받게 되면 delay를 갖다가 6cycle 이 됐을 때 data를 output으로 cache에게 전해준다. 이 때 signal은 cache의 mem_signal과 같은 signal인데, 평소에는 0의 값을 갖고 있지만, data를 output으로 주었을 때 signal을 1로 만들어준다.

```
        if (write_m2_delay == 0)
           write_m2_delay <= write_m2;
        else if (write_m2_delay > 0 && write_m2_delay < `DEF_DELAY)
```

```
            write_m2_delay <= write_m2_delay + 1;
        else if (write_m2_delay == `DEF_DELAY) begin
            memory[address2] <= data2_in;
            write_m2_delay <= 0;
            signal <= 1;
        end
```

store 일 때도 기본적으로 load instruction과 같은 구조를 갖게 된다.

DISCUSSION

cache가 없는 cpu의 cycle

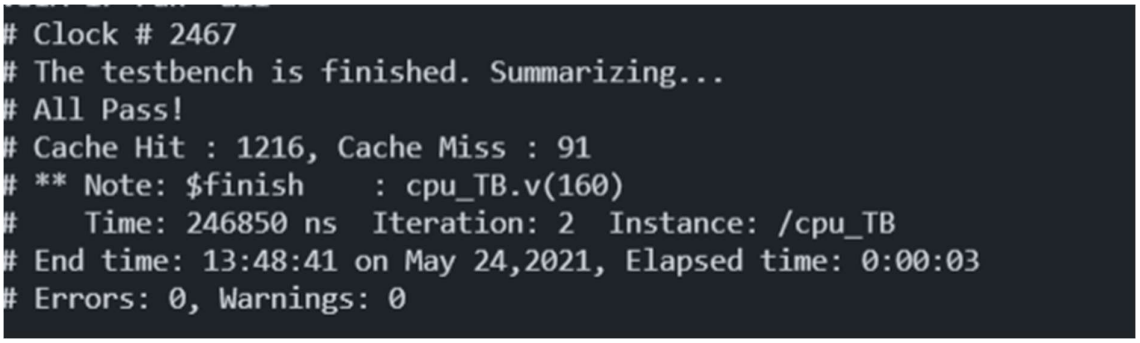
```
# Loading work.Memory
VSIM 5> run -all
# Clock # 2534
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/Users/bhyu4/Desktop/Lab6_Team44_20190065_20190316 (1)/cpu
#    Time: 253550 ns  Iteration: 2  Instance: /cpu_TB
```

cache가 있는 cpu의 cycle

```
VSIM 9> run -all
# GetModuleFileName: ÁöÁ¤µÈ ¸ðµâÀ» Ã£À» ¼ö ¾ø½À´Ï´Ù.
#
#
# Clock # 2467
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/intelFPGA_pro/20.4/lab6/cpu_TB.v(155)
#    Time: 246850 ns  Iteration: 2  Instance: /cpu_TB
```

cache가 있는 cpu의 cycle이 2467로 cache가 없는 cpu의 cycle인 2534 보다 67 cycle 이 줄어든 모습을 알 수 있다.

cache가 있는 cpu에서 memory에 대한 접근이 2cycle에서 6cycle로 늘어났지만, cache의 존재가 이 penalty를 줄이고 오히려 성능개선을 이루어냈음을 알 수 있다.



또한 cache의 sequential logic에서 계산한 num_hit, num_miss를 cpu_TB에 연결하여 display한 결과 cache hit은 1216, miss는 91로 hit ratio는 약 0.9304임을 알 수 있다.

Conclusion

본 랩에서는 cache가 없는 cpu와 cache가 있는 cpu간의 성능 비교를 통해서 cache가 성능개선의 어떤 역할을 하게 되는지 알아보는 것이 목적이었다. 우리는 직접 cache가 있는 cpu에서 속도 향상이 이루어진 것을 확인할 수 있었고, 그 성능 개선이 매우 강력함을 확인할 수 있었다.

하지만 우리가 구현한 cache는 그 size가 작은 cache이고 single level cache이기 때문에 multi level cache와 size가 달라졌을 때는 성능 개선이 어떻게 이루어질지 알 수 없다. 추후에 이러한 cache를 베릴로그를 통해서 만들어볼 수 있다면 이들의 비교를 통해 현대 프로세서에서 적용되고 있는 multilevel cache의 성능 개선에 대한 이해를 직접 해볼 수 있을 것이라고 생각한다.