Report for Lab6: Cache

20140843 Taekang Eom

20150384 Eunyoung Hyung

1. Introduction

In this lab, we need to understand what the cache is and need to implement a direct-mapped cache on our pipelined CPU. Since the speed needed to access the memory is too slow, we use cache between the CPU and memory. Cache keeps a few data which are frequently and recently accessed and serves these data to CPU in few cycles. It is possible because of the locality.

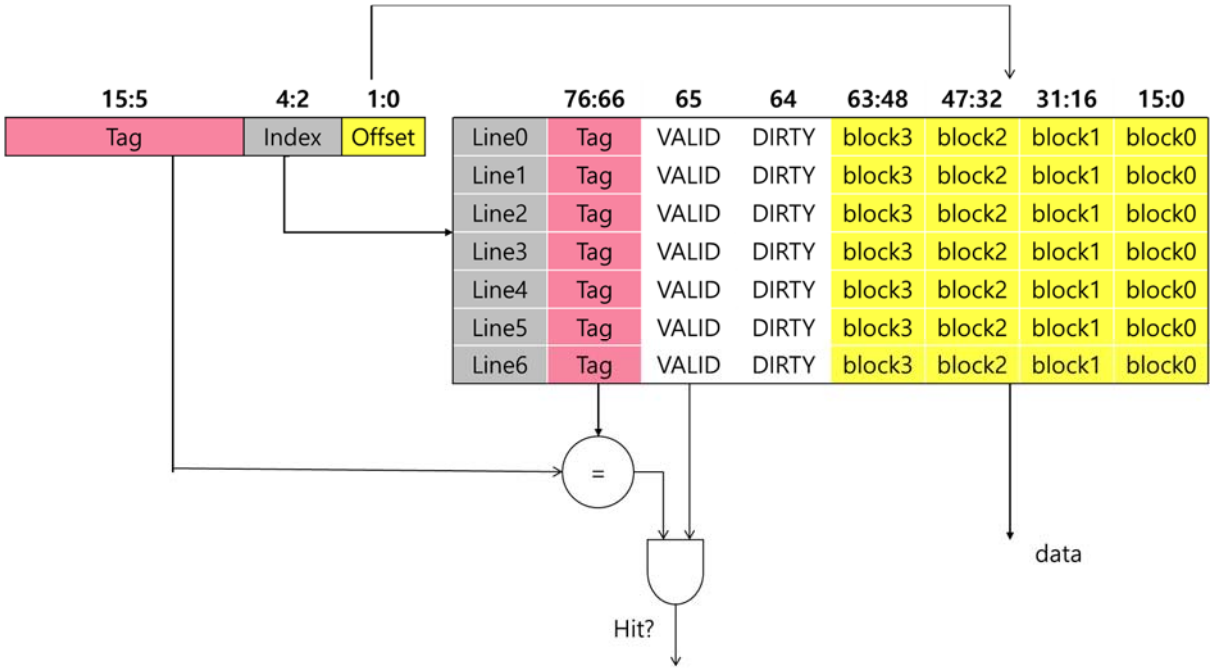
Since cache has the limited space, we need some policies when we access the address which is not in the cache but the cache is full. Also, we need to design when we write a value to the memory when the address is already in the cache or not.

Based on the above, we will compare 2 CPUs: one takes 2 cycles when accessing the memory but it has no cache and another CPU takes 6 cycles when accessing the memory but it has cache.

2. Design

1) Direct-mapped cache

We will implement the Unified I/D cache which has 8 lines and each can contain 4 words. The overall structure looks like the picture below.



2) Cache: Hit, Miss(replacement policy), Write policy

When address[Tag] is same as cache[Index][76:66] and that line is valid, we regard it as cache hit. So we just use the data stored in the cache.

When cache miss occurs, the line that the new address supposed to be will be evited and the whole blocks in that line will be fetched from memory. It's possible because it is direct mapped cache so the location (=line number) is already fixed by the address. Therefore, we do not need any other replacement policies.

In the case of writing to the memory, if the address is already in the cache, we just update the cache and set the DIRTY bit. When the line is evicted and if the DIRTY bit is set, the values in the cache are written to the memory then; this policy is called Write-Back.

3. Implementation

We only address the newly added or updated modules.

1) Opcode.v

These are just for readability and convenience.

```
`define LINE_SIZE 77
```

```
`define BLOCK_NUM 8
`define BLOCK_SIZE 4
`define DIRTY 64
`define VALID 65
```

2) Cache.v

You can refer the comments below.

```
`timescale 1ns/1ns
`include "opcodes.v"
`define PERIOD1 100
`define MEMORY_SIZE 256 //  size of memory is 2^8 words (reduced size)
`define WORD_SIZE 16    //  instead of 2^16 words to reduce memory

module Cache(clk, reset_n, readM1, address1, data1, readM2, writeM2, address2,
data2, finish,access_num,hit_num);
    input clk;
    wire clk;
    input reset_n;
    wire reset_n;

    // 1 is for Instructions
    input readM1;
    wire readM1;
    input [`WORD_SIZE-1:0] address1;
    wire [`WORD_SIZE-1:0] address1;
    output [`WORD_SIZE-1:0] data1;
    reg [`WORD_SIZE-1:0] data1;
    // 2 is for Data
    input readM2;
    wire readM2;
    input writeM2;
    wire writeM2;
    input [`WORD_SIZE-1:0] address2;
    wire [`WORD_SIZE-1:0] address2;
    inout [`WORD_SIZE-1:0] data2;
    wire [`WORD_SIZE-1:0] data2;
    // when finish is 0, cache is being written; otherwise we can access the
cache
    output finish;
    reg finish;
    // for counting the # of access and hit
    output [`WORD_SIZE-1:0] access_num;
    reg [`WORD_SIZE-1:0] access_num;
    output [`WORD_SIZE-1:0] hit_num;
    reg [`WORD_SIZE-1:0] hit_num;
```

```
    // signal from memory after reading
    wire finish_mem;

    reg [`WORD_SIZE-1:0] write_data;
    wire [`WORD_SIZE*`BLOCK_SIZE-1:0] data1_mem;
    wire [`WORD_SIZE*`BLOCK_SIZE-1:0] data2_mem;
    reg [`LINE_SIZE-1:0] cache [0:`BLOCK_NUM-1];
    reg [`WORD_SIZE-1:0] outputData2;
    reg [`LINE_SIZE-1:0] evict1;
    reg [`LINE_SIZE-1:0] evict2;
    reg readM1_mem;
    reg readM2_mem;
    reg writeM2_mem;
    assign data2 = readM2? outputData2:`WORD_SIZE'bz;

    wire I_hit;
    wire D_hit;
    // when the Tag are same and the cache line is Valid, it's hit.
    assign I_hit = (address1[15:5] == cache[address1[4:2]][76:66]) &&
cache[address1[4:2]][`VALID];
    assign D_hit = (address2[15:5] == cache[address2[4:2]][76:66]) &&
cache[address2[4:2]][`VALID];

    Memory NUUT(!clk, reset_n, readM1_mem, address1, data1_mem, readM2_mem,
writeM2_mem, address2,write_data, data2_mem, evict1,evict2, finish_mem);

    always@(posedge clk)
        if(!reset_n)
            begin
                access_num <= 16'b0;
                hit_num <=16'b0;
                finish <= 1'b1;
                // initialize all the cache line with the value 0
                cache[0] = 77'b0;
                cache[1] = 77'b0;
                cache[2] = 77'b0;
                cache[3] = 77'b0;
                cache[4] = 77'b0;
                cache[5] = 77'b0;
                cache[6] = 77'b0;
                cache[7] = 77'b0;
            end
        else
            begin
                // if cache hit
                if(I_hit && (D_hit || !(readM2 || writeM2))) begin
                    hit_num <= hit_num +1;
                    access_num <= access_num + 1;
```

```
                    if(readM1) begin // CPU requires instruction from memory
                        // just for in case when the pc is same as the address
that CPU writes a value to
                        if(writeM2 && address1==address2) begin
                            data1 <= data2;
                        end
                        else begin
                            case(address1[1:0]) // find the proper cache line
and return the data1(=instruction)
                                2'b00 : data1 <= cache[address1[4:2]][15:0];
                                2'b01 : data1 <= cache[address1[4:2]][31:16];
                                2'b10 : data1 <= cache[address1[4:2]][47:32];
                                2'b11 : data1 <= cache[address1[4:2]][63:48];
                            endcase
                        end
                    end
                    if(readM2) begin // CPU requires data from memory
                        case(address2[1:0]) // find the proper cache line and
return the outputData2(=data)
                            2'b00 : outputData2 <= cache[address2[4:2]][15:0];
                            2'b01 : outputData2 <=
cache[address2[4:2]][31:16];
                            2'b10 : outputData2 <=
cache[address2[4:2]][47:32];
                            2'b11 : outputData2 <=
cache[address2[4:2]][63:48];
                        endcase
                    end
                    if(writeM2) begin // CPU wants to write data to memory
                        case(address2[1:0])
                            2'b00 : cache[address2[4:2]][15:0] <= data2;
                            2'b01 : cache[address2[4:2]][31:16] <= data2;
                            2'b10 : cache[address2[4:2]][47:32] <= data2;
                            2'b11 : cache[address2[4:2]][63:48] <= data2;
                        endcase
                        cache[address2[4:2]][`DIRTY] =1'b1; // We need to
indicate Dirty bit for later update in memory
                    end
                    finish <= 1'b1;
                end
                else if(finish) begin // cache miss(evict and memory call)
                    access_num <= access_num + 1; // need to access memory
                    evict1 <= cache[address1[4:2]]; // select the proper line
of the address for instruction
                    evict2 <= cache[address2[4:2]]; // select the proper line
of the address for data
                    write_data <= data2; // value that cpu wants to write
```

```
                    readM1_mem <= readM1; // for instruction
                    readM2_mem <= readM2; // for data
                    writeM2_mem <= writeM2 && !D_hit; // in the case of write
&& the address is not in the cache
                    if(writeM2 && D_hit) begin // If the address for the data
is in the cache,
                        case(address2[1:0]) // update the cache and set the
dirty bit
                            2'b00 : cache[address2[4:2]][15:0] <= data2;
                            2'b01 : cache[address2[4:2]][31:16] <= data2;
                            2'b10 : cache[address2[4:2]][47:32] <= data2;
                            2'b11 : cache[address2[4:2]][63:48] <= data2;
                        endcase
                        cache[address2[4:2]][`DIRTY] = 1'b1;
                    end
                    finish <= 1'b0;
                end
                else if(finish_mem) begin // it's set after reading is done in
memory
                    // update the cache with values from the memory read
before
                    cache[address1[4:2]] <=
{address1[15:5],1'b1,1'b0,data1_mem};
                    if(address1[4:2] != address2[4:2]) begin
                        cache[address2[4:2]] <=
{address2[15:5],1'b1,1'b0,data2_mem};
                    end
                    // set the signals back
                    readM1_mem <= 1'b0;
                    readM2_mem <= 1'b0;
                    writeM2_mem <= 1'b0;
                    finish <= 1'b1;
                    // output the data from memory
                    if(readM1) begin
                        if(writeM2 & address1==address2) begin
                            data1 <= write_data;
                        end
                        else begin
                            case(address1[1:0])
                                2'b00 : data1 <= data1_mem[15:0];
                                2'b01 : data1 <= data1_mem[31:16];
                                2'b10 : data1 <= data1_mem[47:32];
                                2'b11 : data1 <= data1_mem[63:48];
                            endcase
                        end
                    end
                    if(readM2) begin
                        case(address2[1:0])
```

```
                            2'b00 : outputData2 <= data2_mem[15:0];
                            2'b01 : outputData2 <= data2_mem[31:16];
                            2'b10 : outputData2 <= data2_mem[47:32];
                            2'b11 : outputData2 <= data2_mem[63:48];
                        endcase
                    end
                end

            end
endmodule
```

3) Memory.v

We omit the part that saves the data in memory.

```
`timescale 1ns/1ns
`include "opcodes.v"
`define PERIOD1 100
`define MEMORY_SIZE 256 //  size of memory is 2^8 words (reduced size)
            //  requirements in the Active-HDL simulator

module Memory(clk, reset_n, readM1, address1, data1, readM2, writeM2,
address2,write_data, data2, evict1, evict2, finish);

// Omit
// ……
// Omit

            memory[16'hc5] <= 16'hf819;
            memory[16'hc6] <= 16'hf01d;
        end
        else if(readM1 || readM2 || writeM2) begin
            if(finish) begin // writeback; update the memory if the evicted
line is dirty.
                if(evict1[`DIRTY]) begin
                    memory[{evict1[76:66],address1[4:2],2'b00}] =
evict1[15:0];
                    memory[{evict1[76:66],address1[4:2],2'b01}] =
evict1[31:16];
                    memory[{evict1[76:66],address1[4:2],2'b10}] =
evict1[47:32];
                    memory[{evict1[76:66],address1[4:2],2'b11}] =
evict1[63:48];
                end
                if(evict2[`DIRTY]) begin
                    memory[{evict2[76:66],address2[4:2],2'b00}] =
evict2[15:0];
```

```
                    memory[{evict2[76:66],address2[4:2],2'b01}] =
evict2[31:16];
                    memory[{evict2[76:66],address2[4:2],2'b10}] =
evict2[47:32];
                    memory[{evict2[76:66],address2[4:2],2'b11}] =
evict2[63:48];
                end
                // in the case of write and !D_hit, update the memory in
place.
                if(writeM2) memory[address2] <= write_data;
                finish <= 1'b0;
                timer <= 16'b100;

            end
            else if(timer != 16'b0) begin // timer is for the satisfying the
latency regulation
                    timer <= timer-1;
            end
            else begin
            // output the values which will be written to the cache
                    data1 <= {memory[{address1[`WORD_SIZE-
1:2],2'b11}],memory[{address1[`WORD_SIZE-1:2],2'b10}],
                              memory[{address1[`WORD_SIZE-
1:2],2'b01}],memory[{address1[`WORD_SIZE-1:2],2'b00}]};
                    data2 <= {memory[{address2[`WORD_SIZE-
1:2],2'b11}],memory[{address2[`WORD_SIZE-1:2],2'b10}],
                              memory[{address2[`WORD_SIZE-
1:2],2'b01}],memory[{address2[`WORD_SIZE-1:2],2'b00}]};
                    finish <= 1'b1;
            end
        end
endmodule
```

4. Discussion

Cache needs 1959 clocks while non-cache needs 2873 clocks even it needs 4 fewer cycles when accessing the memory. So we could find how the cache works efficiently even when the memory access takes 6 cycles. Also, we could get efficient result, because we choose Write-back policy which makes fewer access to the memory. And the hit ratio of our cache is 0.94.

```
VSIM 2> run -all
# Clock # 1959
# Access # 1437, Hit # 1349
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/Modeltech_pe_edu_10.4a/examples/lab6/cache/cpu_TB.v(157)
#    Time: 196050 ns  Iteration: 2  Instance: /cpu_TB
```

<With cache>

```
VSIM 5> run -all
# Clock # 2873
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/Modeltech_pe_edu_10.4a/examples/lab6/non_cache/cpu_TB.v(154)
#    Time: 287450 ns  Iteration: 2  Instance: /cpu_TB
```

<Without cache>

5. Conclusion

We understand what the Cache is and how the CPU with cache is better than the CPU without cache. We can also learn the policies needed for the implementation of Cache.